ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

DynaProg for Scala

A Scala DSL for Dynamic Programming on GPU and FPGA

Thierry Coppey, Manohar Jonnalagedda
School of Computer and Communication Sciences, EPFL

December 2, 2012

Abstract

Dynamic programming is a common pattern of Computer Science used in various domains. Yet underlying matrix recurrences might be difficult to express and error prone. Additionally, domain experts might not have the skills to make an efficient parallel implementation. In this paper, we leverage the Scala / LMS infrastructure to implement a domain specific language (DSL) for dynamic programming on heterogeneous platforms, which allows to write concise and efficient programs. Our contributions are:

- Reuse of existing compiler technology for a specific purpose
- Grammar embedded in Scala (DSL) to express DP problems efficiently
- Systematic approach to generate and process backtracking information (better than gapc)
- Code generator to transform a grammar into efficient code for CPU, GPU and FPGA

Contents

1 Introduction

1.1 Dynamic programming

Dynamic programming consists of solving a problem by reusing subproblems solutions. A famous example of dynamic programming is the Fibonacci series that is defined by the recurrence

$$F(n+1) = F(n) + F(n-1) \qquad \text{with } F(0) = F(1) = 1$$

which expands to (first 21 numbers)

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ...$$

A typical characteristic is that an intermediate solution is reused multiple times to construct larger solutions (here $F(3)$ helps constructing $F(4)$ and $F(5)$). Reusing an existing solution avoid redoing expensive computations: with memoization (memorizing all the intermediate results we obtain) the solution of $F(n)$ would be obtained after only n additions whereas without memoization it would require $F(n) - 1$ additions !

Given that characteristic, there exist various categories of dynamic programming:

- Series that operates usually on a single dimension
- Sequences alignment (matching two sequences at best), top-down grammar analysis, ...
- Tree-related algorithms: phylogenetic, trees raking, maximum tree independent set, ...

Since the first category is inherently sequential and the third category is both hard to parallelize efficiently (similar to a sparse version of the second category) and does not share much with other classes, we focus on the second type of problems, which is also the most common.

Taking real-world examples, the average input size for sequence alignment is around 300K whereas problems like RNA folding input are usually around few thousands. Multiple input problems also require more memory: for instance matching 3 is $O(n^3)$-space complex. Since we target a single computer with one or more attached devices (GPUs, FPGAs), the available storage is relatively small if we compare it to a cluster of machines. Hence in general, we would focus on problems that are around $O(n^2)$-space and $O(n^3)$-time complexity. We encourage you to refer to the section 2 for further classification and examples.

1.2 Scala and LMS

«Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling programmers to be more productive. Many companies depending on Java for business critical applications are turning to Scala to boost their development productivity, applications scalability and overall reliability.»[1]

Lightweight Modular Staging (LMS) is a runtime code generation framework built on top of Scala that uses only types to distinguish between code being transformed at compilation and at runtime. Through extensive use of component technology, lightweight modular staging makes an optimizing compiler framework available at the library level, allowing programmers to tightly integrate domain-specific abstractions and optimizations into the generation process.

[1] `http://www.scala-lang.org`

2 Dynamic programming problems

2.1 Problems classification

2.1.1 Definitions

- **Alphabets:** the alphabet represent an enumeration of the possible values. Their size helps determining how many bits are required in the implementation to store the values. Alphabets must be defined for input, wavefront, cost and backtrack.
- **Dimensions:** let n the size of the input and d the dimension of the underlying matrix.
- **Matrices:** we refer indifferently by the matrix or the matrices to all the intermediate cost- and backtrack-related informations that are necessary to solve the dynamic programming problem of interest. Matrices elements are usually denoted by $M_{(i,j)}$ (i^{th} line , j^{th} column).
- **Computation block:** this is a part of the DP matrix (cost and or backtrack) that we want to compute. A block might be either a sub-matrix (rectangular) or a parallelogram, possibly cropped at its parent matrix boundaries.
- **Wavefront:** the wavefront consists of all the data necessary to reconstruct a computation block of the DP matrix. It might include some previous lines/columns/diagonals as well as line-/column-/diagonal-wise aggregations (min, max, sum, ...).
- **Delay:** we call delay the maximum distance between an element and its dependencies along column and lines (ex: recurrence $M_{(i,j)} = f\big(M_{(i-1,j)}, M_{(i-2,j-1)}\big)$ has delay 3).

2.1.2 Litterature classification

In the literature, dynamic programming problems (DP) are classified according to two criteria:

- **Monadic/polyadic:** a problem is monadic when only one of the previously computed term appears in the right hand-side of the recurrence formula (ex: Smith-Waterman). When two or more terms appear, the problem is polyadic (ex: Fibonacci, $F_n = F_{n-1} + F_{n-2}$). When a problem is polyadic with index p, it also means that its backtracking forms a p-ary tree (where each node has at most p children).
- **Serial/non-serial:** a problem is serial ($s = 0$) when the solutions depends on a fixed number of previous solutions (ex: Fibonacci), otherwise it is said to be non-serial ($s \geq 1$), as the number of dependencies grows with the size of the subproblem. That is computing an element of the matrix would require $O(n^s)$. (ex: Smith-Waterman with arbitrary gap is $s = 1$; we can usually infer s from the number of bound variables in the recurrence formula)

$$M_{(i,j)} = \max \begin{cases} \dots \\ M_{(i,j-1)} \\ \max\limits_{i<k<j} [M_{(i,k)} + M_{(k+1,j)}] \end{cases}$$

Note that the algorithmic complexity of a problem is exactly $O\big(n^{d+s}\big)$.

2.1.3 Calculus simplifications

In some special case, it is possible to transform a non-serial problem into a serial problem, if we can embed the non-serial term into an additional aggregation matrix. For example:

$$M_{(i,j)} = \max \begin{cases} \max\limits_{k<i} M_{(k,j)} \\ \sum\limits_{k<i,l<j} M_{(k,l)} \end{cases} \implies M_{(i,j)} = \max \begin{cases} C_{(k,j)} \\ A_{(i-1,j-1)} \end{cases}$$

Where the matrix C stores the maximum along the column and matrix A stores the sum of the array of the previous elements. Both can be easily computed with an additional recurrence:

$$\begin{aligned} C_{(i,j)} &= \max(C_{(i-1,j)}, M_{(i,j)}) \\ A_{(i,j)} &= A_{(i-1,j)} + A_{(i,j-1)} - A_{(i-1,j-1)} + M_{(i,j)} \end{aligned}$$

Although this simplification removes some non-serial dependencies at the cost of extra storage in the wavefront, it is not sufficient to transform all non-serial monadic problems into serial problems (ex: this does not apply to Smith-Waterman with arbitrary gap cost).

2.2 Problems of interest

We usually focus on problem that have an underlying bi-dimensional matrix ($d = 2$) because they can be parallelized (as opposed to be serial if $d = 1$) and can solve large problems (of size n). Problems of higher matrix dimensionality ($d \geq 3$) require substantial memory which severely impacts their scalability. Also we tend to limit algorithmic complexity of the problems as from $O(n^4)$ on, running time becomes a severely limiting factor.

We describe problems structures: inputs, cost matrices and backtracking matrix. These all have an alphabet (that must be bounded in terms of bit-size). Unless otherwise specified, we adopt the following conventions:

- Matrices dimensions are implicitly specified by number of indices and their number of elements is usually the same as the input length.
- Number are all unsigned integers
- Problem dimension is m, n (or n) indices i, j ranges are respectively $0 \leq i < m$, $0 \leq j < n$.
- Unless otherwise specified, the recurrence applies to all non-initialized matrix elements.

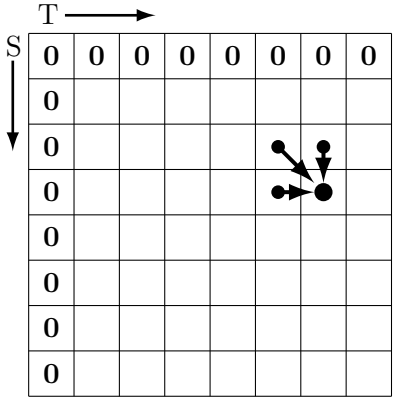
We describe the problem processing in terms of both initialization and recurrences.

2.2.1 Smith-Waterman (simple)

1. Problem: matching two strings S, T with $|S_{\text{padded}}| = m, |T_{\text{padded}}| = n$.
2. Matrices: $M_{m \times n}, B_{m \times n}$
3. Alphabets:
   - Input: $\Sigma(S) = \Sigma(T) = \{a, c, g, t\}$.
   - Cost matrix: $\Sigma(M) = [0..z], z = \max(\text{cost}(\_)) \cdot \min(m, n)$
   - Backtrack matrix: $\Sigma(B) = \{stop, W, N, NW\}$
4. Initialization:
   - Cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
   - Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = stop$.
5. Recurrence:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} 0 & stop \\ M_{(i-1,j-1)} + \text{cost}(S(i), T(j)) & NW \\ M_{(i-1,j)} - d & N \\ M_{(i,j-1)} - d & W \end{array} \right\} = B_{(i,j)}$$

6. Backtracking: starts from the cell $\max\{M_{(m,j)} \cup M_{(i,n)}\}$, stops at the first cell containing a 0.
7. Visualization: by convention, we put the longest string vertically ($m \geq n$):



8. Optimizations:
   - In serial (monadic) problems we can avoid building the matrix M by only maintaining the 3 last diagonals in memory (one for the diagonal element, one for horizontal/vertical, and one being built). This construction extends easily to polyadic problems where we need to maintain $k + 2$ diagonals in memory where k is the maximum backward lookup.
   - Padding: since first line and column of the matrix are zeroes, their initialization might be omitted, but this would implies more involved initialization and computations, which is cumbersome. Also since to fill the i^{th} row we refer to the $(i-1)^{\text{th}}$ character of string S thus we prepend to both S and T an unused character, so that matrix and input lines are aligned. Hence valid input indices are $S[1 \cdots m-1]$ and $T[1 \cdots n-1]$. We refer as such strings as padded strings hereafter (with $|S_{\text{padded}}| = |S| + 1$).

2.2.2 Smith-Waterman with affine gap extension cost

1. Problem: matching two strings S, T with $|S_{\text{padded}}| = m, |T_{\text{padded}}| = n$.
2. Matrices: $M_{m\times n}, E_{m\times n}, F_{m\times n}, B_{m\times n}$
3. Alphabets:
   - Input: $\Sigma(S) = \Sigma(T) = \{a, c, g, t\}$.
   - Cost matrices: $\Sigma(M) = \Sigma(E) = \Sigma(F) = [0..z], z = \max(\text{cost}(\_)) \cdot \min(m, n)$
   - Backtrack matrix: $\Sigma(B) = \{stop, W, N, NW\}$
4. Initialization:
   - No gap cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
   - T-gap extension cost matrix: $E_{(i,0)} = 0$ *«eat S chars only»*
   - S-gap extension cost matrix: $F_{(0,j)} = 0$
   - Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = stop$.
5. Recurrence for the cost matrices:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} 0 & stop \\ M_{(i-1,j-1)} + \text{cost}(S(i), T(j)) & NW \\ E_{(i,j)} & N \\ F_{(i,j)} & W \end{array} \right\} = B_{(i,j)}$$

$$E_{(i,j)} = \max \left\{ \begin{array}{l|l} M_{(i,j-1)} - \alpha & NW \\ E_{(i,j-1)} - \beta & N \end{array} \right\} = B_{(i,j)}$$

$$F_{(i,j)} = \max \left\{ \begin{array}{l|l} M_{(i-1,j)} - \alpha & NW \\ F_{(i-1,j)} - \beta & W \end{array} \right\} = B_{(i,j)}$$

   That can be written alternatively as:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} 0 & stop \\ M_{(i-1,j-1)} + \text{cost}(S(i), T(j)) & NW \\ \max_{1\le k\le j-1} M_{(i,k)} - \alpha - (j-1-k)\cdot\beta & N \\ \max_{1\le k\le i-1} M_{(k,j)} - \alpha - (i-1-k)\cdot\beta & W \end{array} \right\} = B_{(i,j)}$$

   Although the latter notation seems more explicit, it introduces non-serial dependencies that the former set of recurrences is free of. So we need to implement the former rules whose kernel is

$$[M; E; F]_{(i,j)} = f_{\text{kernel}}([M; E]_{(i,j-1)}, [M; F]_{(i-1,j)}, M_{(i-1,j-1)})$$

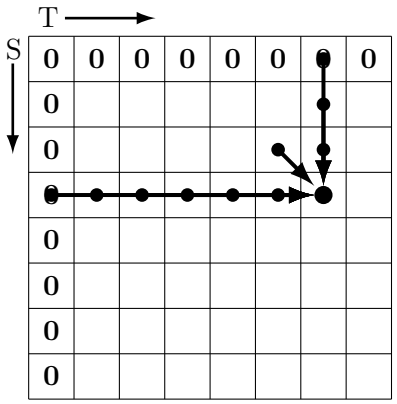
   Notice that this recurrence is very similar to Smith-Waterman (simple) except that we propagate 3 values (M, E, F) instead of a single one (M). Also notice that it is possible to propagate E and F inside a resp. horizontal and vertical wavefront.
6. Backtracking: same as Smith-Waterman (simple)
7. Visualization: same as Smith-Waterman (simple)
8. Optimizations: same as Smith-Waterman (simple)

2.2.3 Smith-Waterman with arbitrary gap cost

1. Problem: matching two strings S, T with $|S_{\text{padded}}| = m, |T_{\text{padded}}| = n$ with an arbitrary gap function $g(x) \geq 0$ where x is the size of the gap. Without loss of generality, let $m \geq n$[2]. Example penalty function could be[3] $g(x) = m - x$.
2. Matrices: $M_{m \times n}, B_{m \times n \times m}$
3. Alphabets:
   - Input: $\Sigma(S) = \Sigma(T) = \{a, c, g, t\}$.
   - Cost matrix: $\Sigma(M) = [0..z], z = \max(\text{cost}(\_)) \cdot \min(m, n)$
   - Backtrack matrix: $\Sigma(B) = \{stop, NW, N_{\{0..m\}}, W_{\{0..n\}}\}$
4. Initialization:
   - Match cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
   - Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = stop$.
5. Recurrence:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} 0 & stop \\ M_{(i-1,j-1)} + \text{cost}(S(i), T(j)) & NW \\ \max_{1 \leq k \leq j-1} M_{(i,j-k)} - g(k) & N_k \\ \max_{1 \leq k \leq i-1} M_{(i-k,j)} - g(k) & W_k \end{array} \right\} = B_{(i,j)}$$

6. Backtracking: similar to Smith-Waterman (simple) except that you can jump of k cells.
7. Visualization:



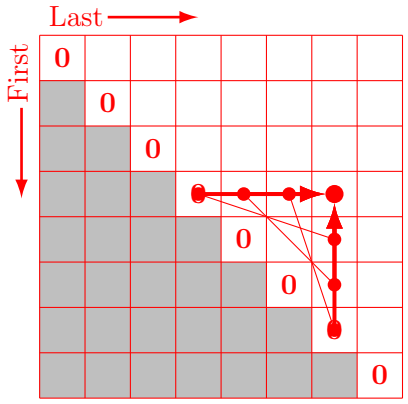
8. Optimizations: The dependencies here are non-serial, there is no optimization that we can apply out of the box here.

[2] Otherwise if $|T| > |N|$ we only need to swap both the inputs and backtracking pairs.

[3] Intuition: long gaps penalize less, at some point, one large gap is better than matching and smaller gaps.

2.2.4 Convex polygon triangulation

1. Problem: triangulating a polygon of n vertices with least total cost for added edges. We denote the cost of adding an edge between the pair of edges i, j by $S(i, j)$, Where $S_{n\times n}$ is an upper triangular matrix with a 0 diagonal that is omitted, hence $|S| = \frac{n^2}{2} = N$.
2. Matrices: $M_{n\times n}, B_{n\times n}$ *«first vertex, last vertex»* upper triangular including main diagonal. Note that if the vertex n is the same as the vertex 0.
3. Alphabets:
   - Input: $\Sigma(S_{(i,j)}) = \{0..m\}$ with $m = \max_S(i, j) \forall i, j$ determined at runtime[4].
   - Cost matrix: $\Sigma(M) = \{0..z\}$ with $z = m \cdot (n - 2)$ (we add at most $n - 2$ edges).
   - Backtrack matrix: $\Sigma(B) = \{stop, 0..n\}$ (the index of the edge we add)
4. Initialization: $M_{(i,i)} = 0, M_{(i,i+1)} = 0, B_{(i,i)} = stop \quad \forall i$
5. Recurrence:
$$M_{(i,j)} = \left\{ S(i, j) + \max_{i<k<j} M_{(i,k)} + M_{(k,j)} \;\middle|\; k \right\} = B_{(i,j)}$$
6. Backtracking: Start at $B_{(1,n)}$. Use the following recursive function for the smaller polygons:
$$\mathrm{BT}(B_{(i,j)} = k) \mapsto \begin{cases} A_i & \text{if } k = 0 \vee k = j \\ \left(\mathrm{BT}(B_{(i,k)})\right) \cdot \left(\mathrm{BT}(B_{(k+1,j)})\right) & \text{otherwise} \end{cases}$$
7. Visualization: the layout is the a matrix of size $n \times (2n - 2)$, because of polygons being "cyclical" in nature.



8. Optimizations: we need the cost of edges between contiguous polygon vertices to have a cost of 0, so that we do not need to have special cases in the DP program (i.e. edges are «already present», hence we cannot add them).

[4] We need to scan/have stats about S and that's where LMS plays a role

2.2.5 Matrix chain multiplication

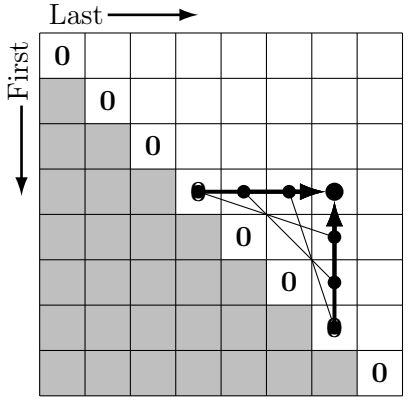
1. Problem: find an optimal parenthesizing of the multiplication of n matrices A_i. Each matrix A_i is of dimension $r_i \times c_i$ and $c_i = r_{i+1} \forall i$. *«r=rows, c=columns»*
2. Matrices: $M_{n \times n}, B_{n \times n}$ *(first, last matrix)*
3. Alphabets:
   - Input: matrix A_i size is defined as pairs of integers (r_i, c_i).
   - Cost matrix: $\Sigma(M) = 1..z$ with $z \leq n \cdot \left[\max_i(r_i, c_i)\right]^3$.
   - Backtrack matrix: $\Sigma(B) = \{stop\} \cup \{0..n\}$.
4. Initialization:
   - Cost matrix: $M_{(i,i)} = 0$.
   - Backtrack matrix: $B_{(i,i)} = stop$.
5. Recurrence: $c_k = r_{k+1}$

$$M_{(i,j)} = \min_{i \leq k < j} \left\{ \; M_{(i,k)} + M_{(k+1,j)} + r_i \cdot c_k \cdot c_j \;\middle|\; k \; \right\} = B_{(i,j)}$$

6. Backtracking: Start at $B_{(1,n)}$. Use the following recursive function for parenthesizing

$$\mathrm{BT}(B_{(i,j)} = k) \mapsto \begin{cases} A_i & \text{if } k = 0 \vee k = j \\ \Big(\mathrm{BT}(B_{(i,k)})\Big) \cdot \Big(\mathrm{BT}(B_{(k+1,j)})\Big) & \text{otherwise} \end{cases}$$

7. Visualization:



8. Optimizations:
   - We need to swap vertically the matrix to have a normalized progression towards bottom right. To do that, we need to map all indices $i \mapsto n - 1 - i$ and we obtain a new recurrence relation:

$$M_{(i,j)} = \min_{i \leq k < j} \left\{ \; M_{(i,k)} + M_{(2i-1-k,j)} + r_i \cdot c_k \cdot c_j \right.$$

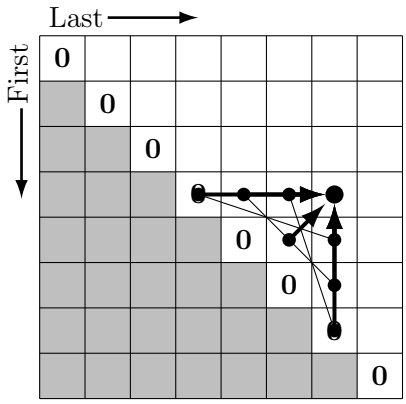
   With the initialization at $M_{(i,n-i-1)}$

2.2.6 Nussinov algorithm

1. Problem: folding a RNA string S over itself $\lfloor |S|/2 \rfloor = n$.
2. Matrices: $M_{n \times n}, B_{n \times n}$
3. Alphabets:
   - Input: $\Sigma(S) = \{A, C, G, U\}$.
   - Cost matrix: $\Sigma(M) = \{0..n\}$
   - Backtrack matrix: $\Sigma(B) = \{stop, D, 1..n\}$
4. Initialization:
   - Cost matrix: $M_{(i,i)} = M_{(i,i-1)} = 0$
   - Backtrack matrix: $B_{(i,i)} = B_{(i,i-1)} = stop$
5. Recurrences:

$$M_{(i,j)} = \max \left\{ \begin{array}{l|l} M_{(i+1,j-1)} + \omega(i,j) & D \\ \max_{i \leq k < j} M_{(i,k)} + M_{(k+1,j)} & k \end{array} \right\} = B_{(i,j)}$$

   With $\omega(i,j) = 1$ if i, j are complementary. 0 otherwise.
6. Backtracking: Start the backtracking in $B_{(1,n)}$ and go backward. The backtracking is very similar to that of the matrix multiplication, except that we also introduce the diagonal matching.
7. Visualization:



8. Optimizations: note that this is very similar to the matrix multiplication except that we also need the diagonal one step backward, so the same optimization can apply.

2.2.7 Zuker folding

1. Problem: folding a RNA string S over itself $\lfloor |S|/2 \rfloor = n$.
2. Matrices: $V_{n\times n}, W_{n\times n}, F_n$ (Free Energy), $BV_{n\times n}, BW_{n\times n}, BF_n$
3. Alphabets:
   - Input: $\Sigma(S) = \{A, C, G, U\}$.
   - Cost matrices:
     - $\Sigma(W) = \Sigma(V) = \{0..z\}$ with $z \leq n * b + c$
     - $\Sigma(F) = \{0..y\}$ with $y \leq \min(F_0, z \cdot n)$
   - Backtrack matrices:
     - $\Sigma(BW) = \{stop, S, W, V, k\}$
     - $\Sigma(BV) = \{stop, HL, IL, SW, (i,j), k\}$ with $0 \leq i, j, k < n$
       HL=HairpinLoop, IL=InteriorLoop, (i,j)=MultiLoop
     - $\Sigma(BF) = \{stop, L, k\}$ with $0 \leq k < n$
4. Initialization:
   - Cost matrices: $W_{(i,i)} = V_{(i,i)} = 0, F_{(0)} =$ energy of the unfolded RNA.
   - Backtrack matrices: $BW_{(i,i)} = BV_{(i,i)} = BF_{(0)} = stop$.
5. Recurrence:

$$W_{(i,j)} = \min \left\{ \begin{array}{l|l} W_{(i+1,j)} + b & L \\ W_{(i,j-1)} + b & R \\ V_{(i,j)} + \delta(S_i, S_j) & V \\ \min_{i<k<j} W_{(i,k)} + W_{(k+1,j)} & k \end{array} \right\} = BW_{(i,j)}$$

$$V_{(i,j)} = \min \left\{ \begin{array}{ll|l} \infty & \text{if}(S_i, S_j) \text{ is not a base pair} & stop \\ eh(i,j) + b & \text{otherwise} & hairpin \\ V_{(i+1,j-1)} + es(i,j) & & stack \\ VBI_{(i,j)} & & (i',j') \\ \min_{i<k<j-1}\{W_{(i+1,k)} + W_{(k+1,j-1)}\} + c & & k \end{array} \right\} = BV_{(i,j)}$$

$$VBI_{(i,j)} = \min \left\{ \min_{i<i'<j'<j} V_{(i',j')} + ebi(i,j,i',j') \right\} + c \;\middle|\; (i',j') \Big\} = BV_{(i,j)}$$

$$F_{(j)} = \min \left\{ \begin{array}{l|l} F_{(j-1)} & P \\ \min_{1\leq i<j}(F_{(i-1)} + V_{(i,j)}) & i \end{array} \right\} = BF_{(j)}$$

With δ a lookup table. In practice, we don't go backward for larger values than 30, so we can replace $\min_{i<k<j}$ by $\min_{\max(i,j-30)<k<j}$ in the expressions of VBI.

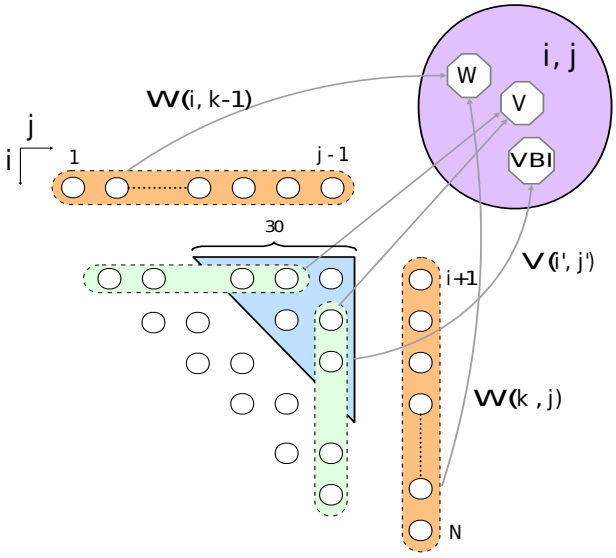
6. Backtracking: Start at $BF_{(n)}$ using the recurrences

$$BF_{(j)} = \begin{cases} P & \Longrightarrow & BF_{(j-1)} \\ i & \Longrightarrow & BF_{(i-1)} + BV_{(i,j)} \end{cases}$$

$$BV_{(i,j)} = \begin{cases} hairpin & \Longrightarrow & \langle \text{hairpin}(i,j) \rangle \\ stack & \Longrightarrow & \langle \text{stack}(i,j) \rangle \oplus BV_{(i+1,j-1)} \\ (i',j') & \Longrightarrow & \langle \text{bulge from } (i,j) \text{ to } (i',j') \rangle \oplus BV(i',j') \\ k & \Longrightarrow & BW_{(i+1,k)} \oplus BW_{(k+1,j-1)} \end{cases}$$

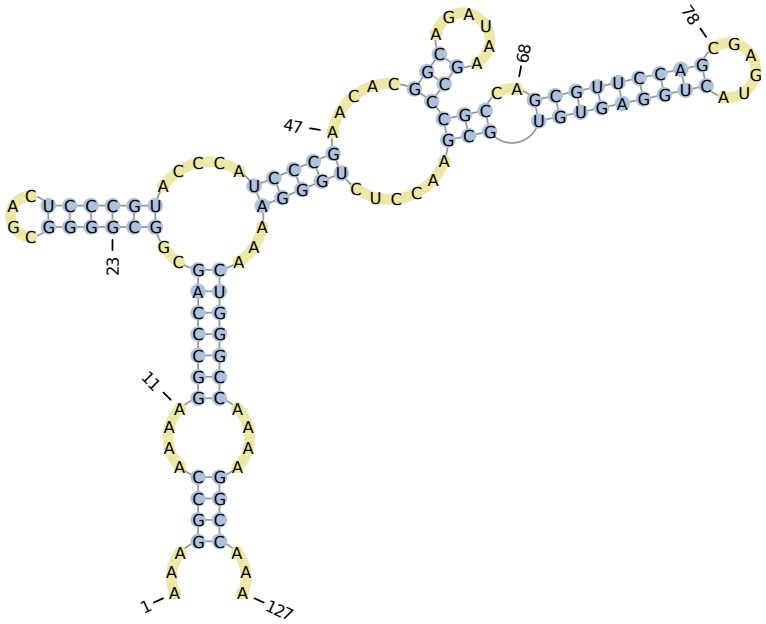
$$BW_{(i,j)} = \begin{cases} L & \Longrightarrow & \langle multi\_loop(i) \rangle \oplus BW_{(i+1,j)} \\ R & \Longrightarrow & \langle multi\_loop(j) \rangle \oplus BW_{(i,j+1)} \\ V & \Longrightarrow & BV_{(i,j)} \\ k & \Longrightarrow & BW_{(i+1,k)} \oplus BW_{(k+1,j-1)} \end{cases}$$

7. Visualizations[5]: The recurrence consists of two non-serial dependencies as in «Smith-Waterman with arbitrary gap cost» plus a bounded 2-dimensional dependency for bulges.



Since this problem is non-trivial to understand from the recurrences, we propose an additional illustration of a RNA chain folded according to the Zuker folding algorithm.

[5]Reproductions of the illustrations from [**?**] pp.148,149



An example of an RNA folded into a secondary structure. Types of structural features modeled by the Zuker folding algorithm include: dangling ends (1), internal loop (11), stack (23), multi-loop (47), bulge (68) and hairpin loop (78).

8. Optimizations: XXX: notice that there are 3 matrices: W,V (VBI is part of V) that can be expressed using regular matrix, and F that is of different dimension than W and V and requires a special construction (in the wavefront?). We need to find a nice way to encode both its construction and backtrack into the existing framework (implement 1D DP recursively?) Also notice that the k of BV and BW describe almost the same backtrack, but there is an additional cost c in BV.

2.3 Related problems

The goal of this section is to demonstrate that our framework can accommodate with many problems that we have not considered at the design time.

2.3.1 Serial problems

Problem	Shape	Matrices	Wavefront
Smith-Waterman simple	rectangle	1	–
Smith-Waterman affine gap extension	rectangle	3	–
Needleman-Wunsch	rectangle	1	–
Checkerboard	rectangle	1	–
Longest common subsequence	rectangle	1	–
Longest common substring	triangle	1	–
Levenshtein distance	rectangle	1	–
De Boor evaluating B-spline curves	rectangle	1	–

2.3.2 Non-serial problems

Problem	Shape	Matrices	Wavefront
Smith-Waterman arbitrary gap cost	rectangle	1	–
Convex polygon triangulation	parallelogram	1	–
Matrix chain multiplication	triangle	1	–
Nussinov	triangle	1	–
Zuker folding	triangle	3?	0?
CYK Cocke-Younger-Kasami	triangle	#rules	–
Knapsack (pseudo-polynomial)	rectangle	1	–

2.3.3 Other problems

- Dijkstra shortest path: we need a $E \times V$ matrix, along E forall V reduce its distance, problem is serial along E, non-serial along V hence of complexity $O(|E| \cdot |V^2|)$ which is far worse than both $O(|V|^2)$ (min-priority queue) and $O(|E| + |V| \log |V|)$ (Fibonacci heap).
- Fibonacci: this problem is serial 1D. Could be implemented using a placeholder element in one of the matrix dimension.
- Tower of Hanoi: 1D non-serial
- Knuth's word wrapping: 1D non-serial
- Longest increasing subsequence: serial but binary search algorithm more efficient: $O(n \log n)$.
- Coin Change: 1D non-serial

XXX: Fill up to 20-50 related problems to see how general we are

- Floyd-Warshall: <it possible to move the k external loop inside?> Serial with n iterations
- Viterbi (hidden Markov models): T non-serial iterations over a vector
- Bellman-Ford (finding the shortest distance in a graph)
- Earley parser (a type of chart parser)
- Kadane maximum subarray 1D serial, look at Takaoka for 2D
- Structural alignment (MAMMOTH, SSAP), RNA structure prediction.
- Recursive least squares

- Bitonic tour

```
- Balanced 0-1 matrix
- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-hori
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth
- Transposition tables and refutation tables in computer chess
- Pseudo-polynomial time algorithms for the subset sum and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- Duckworth-Lewis method for resolving the problem when games of cricket are interrupted
- The Value Iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Pho
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the traveling salesman problem, either exactly (in exponential time)
- Beat tracking in music information retrieval.
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision.
- Seam carving (content aware image resizing)
- Some approximate solution methods for the linear search problem.
=====> http://en.wikipedia.org/wiki/Dynamic_programming#A_type_of_balanced_0.E2.80.931_matrix
- Shortest path in DAGs
- Shortest path
- All pair shortest paths
- Independent sets in trees
=> also see exercises for more problems
=====> http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf
- Subset Sum, Coin Change, Family Graph
=====> http://www.algorithmist.com/index.php/Dynamic_Programming
- Optimal Binary Search Trees
=====> http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/05-dynprog.pdf
- Independent set on a tree
=====> http://www.cs.ucsb.edu/~suri/cs130b/NewDynProg.pdf
```

3 Architecture design and technical decisions

3.1 User facing language requirements

The Algebraic Dynamic Programming approach (ADP) introduces a conceptual splitting of a DP algorithm into a recognition and an evaluation phase. The evaluation phase is specified by an evaluation algebra, the recognition phase by a yield grammar. Each grammar can be combined with a variety of algebras to solve different but related problems, for which heretofore DP recurrences had to be developed independently. Grammar and algebra together describe a DP algorithm on a high level of abstraction, supporting the development of ideas and the comparison of algorithms. A notation for yield grammars is provided that serves as a domain-specific language.

Given such formalization [**?**] of dynamic programming on sequences, it seems natural to borrow from it and extend it to other types of DP problems. In short, this framework allow the user to define a grammar using parsers, which are then run over an input string and produce intermediate results that are memoized into a table, when multiple solutions are possible, the user can define an aggregation function (h) to retain only some candidates for further combination.

The benefits of ADP framework is that it does not constraint the result of the evaluation to be a single value, but could extend parsers to backtracking parsers or pretty-printers. Such functionality is easy to support in plain Scala, however, it could hampers the performance of the GPU implementation if we leave to the user the duty of representing and processing all the information within the parsers. Its major shortcoming in expressivity is that it only operates on strings, making only triangular-matrix problem easily representable.

The additional features we want to support are:

1. **Cyclic problems:** are inherently very similar to string problems, except that the input is cyclic. To support such problem efficiently, we only need to mark the grammar cyclic such that it would apply on any unfolding of the cyclic input string.
2. **Input pair algebra:** the original ADP framework only support single input, we want to support a pair of inputs such that we can treat problem such as Smith Waterman or Needleman-Wunsch. However, it does not make sense to treat more than two sequences because of the $O(n^3)$ storage requirements that limits the problems size more dramatically.
3. **Windowing**: this can be easily encoded by passing the windowing parameter that limits the computation, then it could be possible to collect either the best or k-best results.
4. **Device restrictions:** since CUDA (and FPGA) cannot operate on an arbitrary Scala classes, we need to restrict the language to primary types (int, float, ... and structures of them). However, we want to preserve the expressiveness available for Scala and impose restrictions on the input and answers available to CUDA. A typical restriction we want to make is that processed structures are of fixed size so that we avoid memory management issues and thread divergence.

   The general ADP framework supports multiple solutions for intermediate results. Similarly, we want to impose the restriction that only the best result would be stored on device while preserving the support for multiple partial results in Scala.
5. **Automatic backtracking:** Since efficient code has to be devised we impose restrictions on the output that could be generated by the parsers on devices. However, on the other side, the backtracking information would be of primary interest for the DSL user, hence we need to automate the backtracking to both fulfill the goals of efficiency and usefulness

of the device specific implementation.
Enforcing automatic backtracking presents the advantage to ensure constant size for intermediate results, hence ensuring an $O(n^2)$ storage requirement while memoizing the backtracking with the results would have the drawback of growing towards final result and duplicated unnecessary information, hence requiring both $O(n^3)$ space and memory management features on devices. Collecting the backtracking list can be easily done in $O(n)$ and then inverted whether we prefer bottom-up or top-down construction (the backtrack is usually a lattice of nodes that constitute a tree whose leaves are input elements).

Since we construct the dependency tree bottom up, and since we need to work with a fixed-size memory, we need to do the additional steps in our pre-processing:

- **Normalization:** in order to automate the backtracking, we need the rule to present a certain shape (in particular we want to remove alternatives) so that we can define uniquely the backtracking information. Another step we can make in the process is to break down complex rules into simpler ones if they are expressible so, hereby reducing the complexity of the overall algorithm, would the user production grammar not be optimal. Also we can strip away unnecessary rules.
- **Dependency analysis:** a precise order between the rules application must be given. This is required as we use a bottom-up approach where all previous results must be computed versus a top-down approach with a table where a fallback to computation always exist if the desired element is not present.

Size analysis to know what storage size we require: ex: Zucker requires $O(n^2) + O(n)$ storage...

3.2 Parsing grammar (ADP)

Borrowing from the ADP grammar, parsers operate on a subsequence of the input (sequence) and return a list of parse results on these subsequences. These results are obtained by the internal transformation that is defined within a parser. The parsers are classified by types:

- **Terminal:** operates on a subsequence and returns its content or position (or nothing if the sequence does not fit the terminal).
- **Filter:** this parser accepts only subsequences matching a certain predicate
- **Union:** this parser expresses alternative between two different parsers
- **Concatenation:** construct a larger sequence from two subsequences. The subsequences can be of fixed or varying size and concatenation operators might impose restrictions on the subsequences length to be considered.
- **Transformation:** this parser transform its input using an arbitrary function. It is typically used to transform a parse list into a score that can later be aggregated.
- **Aggregation:** the aggregation reduces the list of results, typically using minimum or maximum but can be arbitrarily defined.

XXX: explain ADP concepts

explain split into signature and grammar

XXX: normalization, transforms applied

```
Optimizations: we want to split grammar into binary productions, to do that we need
to have equality on the following formula. We can solve it more easily if functions are linear
```

$$\min_{i<k_1<k_2<j} \left[f(i,k_1,k_2,j)\right] \leq \min_{i<k_2<j} \left[g_2(i, \min_{i<k_1<k_2} \left[(g_1(i,k_1,k_2)\right], k_2, j)\right]$$

3.3 Backtracking

In order to do a clean and efficient transformation from ADP-like language to plain C recurrences, we need to construct bottom-up recurrences from top-down parser rules. To do that, we slightly need to modify the ADP language in order to separate the backtracking and the scoring, because we want to obtain an efficient algorithm: backtrack writes are in $O(n^2)$ whereas score reads are proportional to the algorithmic complexity ($O(n^3)$ or more for non-serial). To deal with this problem, we are facing two options:

- **Explicit backtracking:** requires clear syntactical separation between the score and the backtrack and is not implemented in vanilla ADP (unless the whole backtrack is made part of the scoring: big performance impact and non-constant memory requirement issues make its GPU implementation hard and not desirable). Since the backtracking is user-defined, there is no way to generate the backtracking algorithm automatically, hence the user also needs to provide it.
- **Implicit backtracking:** implies that every rule needs to be normalized, and transformed such that given a rule identifier and a set of indices (subproblems breaking), it is possible to retrieve the subproblems combination that build the problem. To do that we need to apply the following transformations
  1. Normalize rules and identify them uniquely by
     - Removing all the «or» operators by breaking a parser into multiple subrules
     - Forwarding names: replace a rule that is a single invocation by its content (i.e. name a terminal parser with its own name instead of its containing parser's)
     - Assigning each subrule an unique identifier r and create a mapping table $T : r \rightarrow$ user-defined name of the parser.
  2. The data elements corresponding to a rule are (scoring, backtracking) and named after the tabulation. The scoring is a user-defined primary type (int, double, ...) and the backtracking is a tuple $(\mathrm{id}_{\mathrm{subrule}}, k_1, k_2, \ldots, k_m)$ where m is the maximal number of splits that can occur in the subrules.
  3. During the matrix computation of cell (i, j), if the subrule r applies, the backtrack will be set as $(r, k_1, k_2, \ldots, k_{m_r})$ (obviously with $i \leq k_1 \leq k_2 \leq \ldots k_{m_r} \leq j$ and $m_r \leq m$).
  4. During backtracking, when reading the cell (i, j) with backtrack $(r, k_1, k_2, \ldots, k_m)$, given r, we recover the subrule hence the user-defined name of the rule that applies and k_{m_r}, which allows us to enqueue the subwords $(i, k_1), (k_1, k_2), ..., (k_{m_r})$ for further backtracking. If r refers to a terminal, we stop the backtracking.
  5. The backtracking can be returned to the user as a mapping table T and a list of triplets (r, i, j) where r is the index of the sub-rule that can be easily mapped into the user name (such that $T : r_u \rightarrow$ user-defined rule name) and (i, j) is the subword on which the rules applies.

  In short, we break parsers into normalized rules, the backtracking information is the rule id (which rule to unfold) and a list of indices (how to unfold it).

Assuming that the backtracking information is meant to guide further processing, we provide this information into a list constructed bottom-up: it can be simply processed by the user program in-order, applying for each user-defined rule the underlying transformation, and storing intermediate results (i.e. in a hash map) until they are processed by another rule. Since consumed sub-result

can be dropped and since the backtracking contains only relevant sub-problems, ultimately, only the result will be stored.

3.3.1 Backtracking with multiple backtrack elements

The backtracking technique described above work fine when there is a single element stored per matrix cell (which is usually the case with min/max problems). However, in the generalization introduced by ADP, it is possible that a matrix cell stores multiple results. In such case, we need to select a correct result to avoid backtracking inconsistencies.

Additionally, we need to keep track of the multiplicities of the solutions, that is if we want to obtain the k best solutions, we need to make sure that we return k different backtracks. To do that, we maintain a multiplicity counter in each backtrack process:

- While there is a single solution for all possible incoming paths, we continue in this direction with the same multiplicity (we have no choice).
- When there is r different solutions available, and the path multiplicity at this point is k we have the following cases:
  1. If $k \geq r$: we explore all paths with multiplicity $k - r + 1$. This is because each branch may produce only one solution and we don't know ahead of time which path will provide multiple solutions. Finally, we retain only the k best solutions.
  2. If $k < r$ (there is more paths than needed): we can only explore the k first paths with multiplicity 1 and safely ignore the other (as we only need k results).

Now it remains the problem to generate all possible results and check whether they are correct correct. To do that we can simply re-apply the parsers while maintaining the source elements of all production and then retain only those with desired score and backtrack. Since we know the backtrack for one element, we can do the following optimization at backtrack parser computation:

1. Defuse or's: since we know exactly (by the subrule id, maintained in the backtrack) which alternative has been taken to obtain the result, we can skip undesired branches of or parsers.
2. Feed concatenation indices: since the backtrack stores the concatenation indices, we can reuse in the concatenation parsers. This removes the $f(n)$ factor in the backtrack complexity (as concatenation backtrack parsers «know» where to split).
3. Skip filters: since filter are applied before their inner solution is computed, their are only position-dependent. Hence if a backtrack involves a filter, since its position is fixed, the filter must have been passed at matrix construction time.

3.3.2 Backtracking complexity

We want to have an estimation of the complexity of the two operations to see what is the overhead of multiple backtracking. For single-element backtrack, we only need to revert the parser to find involved subwords, which is linear in the parser size (because the backtracking identifies uniquely the alternative in or's and indices in concatenations). At every backtrack step, either:

- The word is removed at least one character, which leads to maximal backtrack length of n.
- The word is split in s subwords, with recurrence $f(n) = 2f(n/2) + 1$ by solving this recurrence we see that there can be at most n final nodes and n intermediate nodes (when $s = 2$). Hence the backtrack length is at most $2n$.

Let one parser reversal complexity be $O(p)$, single backtrack has $O(2n \cdot p)$ complexity. For the k-elements backtrack, since we regenerate all possible solutions, that is $O(k^{c+1})$ candidates (with

c the maximal number of concatenation in the parser), the overall complexity is $O(2n \cdot k^{c+1} \cdot p)$. Hence there is a k^{c+1} factor to pay if we want to backtrack the k best solutions[6].

3.3.3 Backtrack utilization

Since the dynamic programming input may be a subset of the larger problem that we want to resolve[7], we need to be able to apply the result of the dynamic programming computation on values in a different domain. The easiest way to do that, is to reuse the same grammar on a different domain, and only compute the resulting trace obtained from the DP solver.

This step is pretty straightforward: since ADP parsers emphasize on the split between signature and grammar and decouples them, we only need to modify the signature to operate on another domain, and reuse the same grammar. The key point here is to notice that a backtrack trace on one parser can be reused on another, providing that they have the same grammar. For instance, to compute optimally a matrix chain multiplication, we solve the DP problem in a domain where matrices are represented by their dimensions, we obtain an optimal trace and feed it to a parser operating on the «real matrices» domain that will do the actual computation.

3.4 Memory constraints

We denote by *device* the computational device on which the processing of the DP matrix (or of a computational block) is done and M_D its memory. This can be the GPU or the FPGA internal memory. Usually the main memory is larger than device memory and can ultimately be extended by either disk or network storage.

We propose to evaluate the device memory requirements to solve the above problem classes. We need first to define additional problem properties related to implementation:

- **Number of matrices:** multiple matrices can be encoded as 1 matrix with multiple values per cell. Hence the implementation differentiates only between cost and backtrack matrices with respective element sizes S_C and S_B.
- **Delay of dependencies:** In case the problem does not fit into memory, partial matrix content needs to be transferred across sub-problems. Such data amount is usually proportional to the delay of dependencies. If this delay is small, it might be worth to duplicate matrix data in the wavefront, otherwise it might be more efficient to allow access to the previous computational blocks of the matrix.
- **Wavefront size:** Finally some aggregation that is made along some dimension of the matrix does not need to be written at every cell but can be propagated and aggregated along with computation (ex: maximum along one row or column). Hence such information can be maintained in a single place (in the wavefront) and progress together with the computation. We denote by S_W the size of wavefront elements.
- **Input size:** the size of an input letter (from input alphabet) is denoted by S_I.

3.4.1 Small problems (in-memory)

Problem that can fit in memory can be solved in a single pass on the device. Such problem must satisfy the equation:

$$(S_I + S_W) \cdot (m + n) + (S_C + S_B) \cdot (m \cdot n) \leq M_D$$

[6]Note that the same k^{c+1} factor lies in the forward matrix computation complexity.

[7]For instance in matrix chain multiplication, we only care about matrix dimensions for dynamic programming, however, we ultimately want to multiply the real matrices and obtain a result.

For instance, assuming that $m = n$, $M_D = 1024$Mb, that backtrack is 2b (<16384, 3 directions) and that the cost can be represented on 4 b (int or float), that input is 1b (char) and that there is no wavefront, we can treat problems of size n such that $2n + 5n^2 \leq 2^{30} \implies n \leq 14650$. We might also possibly need to take into account extra padding memory used for coalesced accesses. But it is reasonable to estimate that problems up to 14K fit in memory.

3.4.2 Large problems

To handle large problems, we need to split the matrix into blocks of size $B_H \times B_W$. For simplification, we assume a square matrix made of square blocks with b blocks per row/column.

3.4.3 Non-serial problems

Non-serial problems need to potentially access all elements that have been previously computed. We restrict[8] ourselves to the following dependencies:

- Non-serial dependencies along row and column
- Serial dependencies along diagonal, with delay smaller or equal to one block size

Such restriction implies that all the block of the line and the row, and one additional block to cover diagonal dependencies must be held in memory (independently of the matrix shape).

For simplification, let $m = n$ and assume that we have b square blocks per row and per column. Hence we have the following memory restriction:

$$2\frac{n}{b}(S_I + S_W) + 2 \cdot \frac{n^2}{b}S_C + \frac{n^2}{b^2}S_B \leq M_D$$

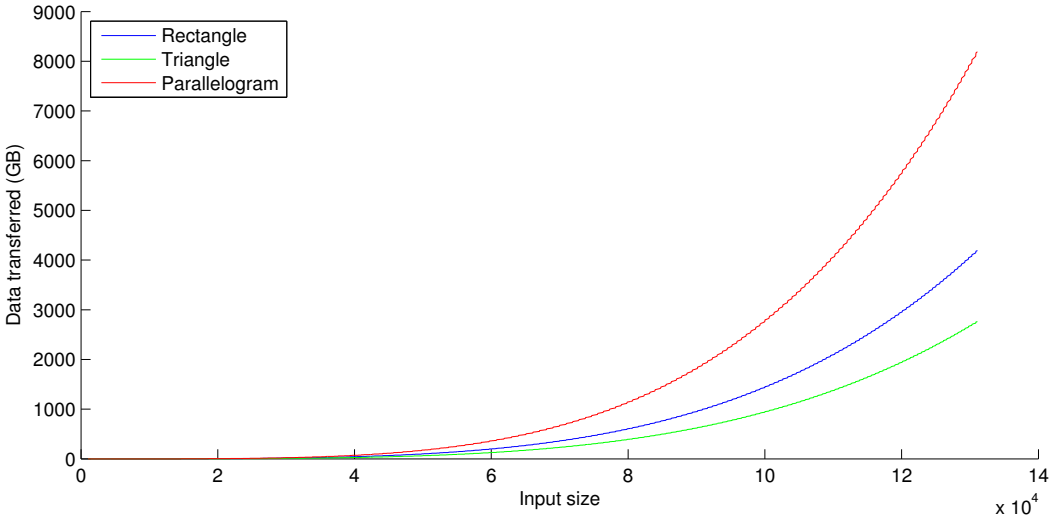
We also need to take into account the transfer between main memory (or disk) and device memory. Dependency blocks only need to be read, computed blocks need to be written back. Ignoring the backtrack and focusing only on the cost blocks, the transfers (in blocks) are:

$$\begin{aligned} b^2 + (b-1)^2 + \sum_{i=0}^{b-1} i \cdot b &= \tfrac{1}{2}b^3 + \tfrac{3}{2}b^2 - 2b + 1 && \text{(Rectangle)} \\ \sum_{i=1}^{b} \Big(1 + 2 \cdot (i-1)\Big) \cdot (b + 1 - i) &= \tfrac{1}{3}b^3 + \tfrac{1}{2}b^2 + \tfrac{1}{6}b && \text{(Triangle)} \\ \sum_{i=1}^{b} \Big(1 + 2 \cdot (i-1)\Big) \cdot b &= b^3 && \text{(Parallelogram)} \end{aligned}$$

Putting these two formula together, and using most of the device memory available, we obtain the following results with $S_C = 4, S_B$[9] $= 4, S_I = 1, S_W = 0$ and $M_D = 2^{30}$:

[8] As we have not encountered a problem with non-serial dependencies along the diagonal.

[9] To deal with larger matrices, backtrack data need to be extended.



Given an experimental bandwidth of 5.3743 Gb/s between CPU and GPU, processing matrices one order of magnitude larger (128K) would result in respectively $13^{(R)}$, $8.5^{(T)}$ and $25.4^{(P)}$ minutes of transfer delay. Extrapolating the preliminary results of small problems, a computation on input of size 128K would require respectively 7 days $13h^{(R)}$, 2 days $22h^{(T)}$ and 6 days $10h^{(P)}$, assuming there is no other scalability issues. Although this overhead seems appealing compared to the computation time, the total time blows up (because of the $O(n^3)$ complexity) and make the processing of such large problem less relevant. Given that real problems – like RNA folding – operate at input sizes up to 4096, it would not be of much relevancy to implement a version for larger cases, although perfectly feasible.

3.4.4 Serial problems

The serial problem have the interesting property to access to a fixed number of previous elements. These elements can be either stored either explicitly in a scoring matrix or implicitly as moving aggregation into a wavefront. Since the dependencies are fixed, the computation direction gains an additional degree of freedom: matrix can be solved in diagonal (as non-serial problems) or line-wise or column-wiser. This allows to store the whole necessary state to make progress into a limited number of lines (or columns), and sweep vertically (resp. horizontally) along the matrix.

Since serial problems are of complexity $O(n^2)$ (due to the matrix dimension and the finite number of dependencies), it is possible to tackle much larger problem than non-serial during the same running time. Hence, it seems obvious to let serial problems grow larger than the memory.

Mixing the dependency property and size requirements, we can split the matrix into sub-matrices, store special lines (and/or columns) into memory (or hard disk), and repeat computations to solve the backtrack (similarly as in [**?**],[**?**], but this implementation use problem-specific knowledge that might not generalize).

To store intermediate lines and columns, we are facing two different strategies to explore:

- **Fixed subproblem size:** we decompose the algorithm as follows
  1. Define a grid of «major column and rows», where each cell's data (input, output, cost and backtrack matrices) fits into the device memory.
  2. Compute the values of the grid's major columns and rows in one pass.
  3. Second (on-demand) computation to process backtracking inside relevant cells.

Let b the number of cells that we have on each row/column, the total computation running time would be $(b^2 + 2b) \cdot t_b$ where t_b is the time to compute one cell's matrix. This division has the advantage of providing the minimal computation time at the expense of external storage proportional to $O(n)$ (if we store only lines or columns) or $O(n^2)$ (if we store both).

- **Myers and Miller's algorithm:** (divide and conquer) This algorithm break the DP problem into 2 (or 4) subproblems such that once the middle line/column is computed, the problem can be solved for 1 submatrix and backtracking among up to 2 of the 3 other. This breaking is applied recursively until the submatrix data fits into memory. The storage requirements are $4 \cdot O(n)$ (we store along both dimension $1 + \frac{1}{2} + \frac{1}{4} + ...$ lines/columns). The algorithm proceeds as follows: first it solves the problem to obtain the first backtracking element, then it breaks the matrix in 4 submatrices, and refine it until backtrack is tractable. Since there is at most $\log n/b$ refinements and since every part of the matrix may be involved in backtrack, running time is $O(n^2 \log_2 n)$.
- **Hybrid approach:** a hybrid approach might be created to take advantage of additional available memory, however, the running time decreases logarithmically to the space used, this means that using twice more storage space would only result in a $2\times$ speedup (measuring only the computation time). Hence an hybrid approach would be to decide a k such that at each step we partition the desired submatix into a intermediate grid of k rows/columns. The space usage would be in $2k \log_k(n/b)$ and the running time complexity would be $O(n^2 \cdot \log_k n)$. Then the user would be able to fix a storage space $S \geq 4 \log_2(n/b)$ and obtain the corresponding k for a given n.

try to setup wavefront size (if needed) => just enlarge the matrix by 1 so that we go wavefront-to-wavefront

XXX: what's the maximal size of the wavefront ??

XXX: can we avoid to store some matrices and put them in the wavefront ??

Split into blocks:

- Decide the shape of the blocks
- Decide the size of the blocks
- Decide of a strategy to store intermediate lines/columns: space/time tradeoff.

XXX: make it work up to 14K for all 3 problems using multiple kernels XXX: example of non-symmetric serial problem

The three last elements, combined with the above one, provide a precise estimation of the memory consumption, and the implementation difficulty

all the problem are subject to the input dimensions n.

the two latter one gives an estimation of the constant factor.

The delay of dependencies might also have an impact: if the matrix is too large to fit in the memory (device or main memory), it becomes necessary to maintain partial matrix content (all the intermediate elements) within the wavefront. Also the number of cost matrices might affect the performance, simply because maintaining them requires computations and memory accesses.

3.5 Memory layout

XXX

3.6 LMS compiler stack

User-language: define additional parameters for the recurrence

- Windowing (to convert non-serial into serial problems)
- Input sizes, and alphabets (backtrack, input, cost)
- Backtrack (implicitly by backtrack alphabet size) and cost matrices bit-sizes (cost maximum may be inferred using «Yield size/grammar analysis»)
- Recurrence functions, devices available
- What to keep in memory (cost, backtrack or both).

⇓ Conversion (using an existing technique)

Intermediate representation

⇓ Optimizations

- Transform non-serial into serial
  - Use aggregation functions/transformations
  - Use windowing from user (if no other technique succeed)
- Define the wavefront depth
- Avoiding the cost matrix by moving it into the wavefront

Code specification

- Kernel function (1-element function), inputs, outputs, wave front, dependencies, bit sizes
- Device-level interface => setup the block sizes(w/h), input and memory sizes
- Define the device-specific implementation of the block (CPU/FPGA/CUDA)
- Define the co-processor memory aggregation function
- Define the scheduling of the blocks and aggregation (software pipelining)
- Define the data movement back and forth to disk

⇓ Generation

- Generate the kernel for specific device
- Generate the scheduling and barriers

Binary program

4 Implementation

4.1 ADP parsers

4.2 Transformations and simplifications

```
 * Parser construction:
 * 1. Assign a "OR_id" (sub_rule_id) and "CONCAT_id" to all parsers so that we know which rule a
 *    How to skip some indices ?
 *
 * Scala running:
 * 2. Provide meaningful backtrack: rule_id and list of concat indices
 *
 * Code generation:
 * 1. Normalize rules (at hash-map insertion (?))
 * 2. Compute dependency analysis between rules => order them into a list/queue
 *    - If rule R contains another rule S unconcatenated (or concatenated with empty)
 *      then we have S -> T (S before T)
 * 3. Compute the maximal number of concatenation among each rule (field in Treeable(?))
 * 4. Break rules into subrules (at each Or, which must be at top of the rule)
```

4.3 LMS integration

4.4 CUDA implementation

5 Benchmarks

5.1 Related work

The goal of this section is to see how we are comparing with other papers in terms of performance and to note performance progress of the improvements.

Graphic cards[10]

Paper		–	**ATLP[?]**	**SWMB[?]**
Serie		GeForce	Tesla	GeForce
Model		GT 650M	C1060	GTX 560
Architecture		Kepler	GT200	GF114
Capability		3.0	1.3	2.1
Memory	Mb	1024	4096	1024
CUDA cores		384	240	384
Core clock	MHz	756	1300	822
Memory clock	MHz	1953	1600	4008
Memory bus	bit	128	512	256
Memory bandwidth	GB/s	28.8	102.4	128.26
Processing power	GFLOPS	564.5	622.08	1263.4
Processing speedup		1	1.07	
Memory speedup		1	3.55	

Results

ATLP[?]

Matrix size	128	256	512	1024	1536	2048	2560	3072	3584	4096
No split	0.07	0.09	0.19	0.59	1.27	2.25	3.51	5.07	6.92	9.06
Split at 1	0.06	0.07	0.08	0.14	0.26	0.47	0.77	1.21	1.80	2.57

Matrix multiplication timing in seconds

SWMB[?]

Matrix size	**Sequences**	**No pruning**	**Pruning**
162K × 172K	NC_000898.1, NC_007605.1	1.2	1.2
543K × 536K	NC_003064.2, NC_000914.1	10.8	10.8
1044K × 1073K	CP000051.1, AE002160.2	40.3	36.2
3147K × 3283K	BA000035.2, BX927147.1	363.6	363.2
5227K × 5229K	AE016879.1, AE017225.1	962.4	469.5
7146K × 5227K	NC_005027.1, NC_003997.3	1309	1309
23012K × 24544K	NT_033779.4, NT_037436.3	19701	19694
59374K × 23953K	NC_000024.9, NC_006492.2	49634	46869
32799K × 46944K	BA000046.3, NC_000021.7	53869	29133

[10]Source: `http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units`

Smith-Waterman, timing in seconds

5.2 Results

Intermediate results

Matrix	Block	Comment	R(s)	T(s)	P(s)
1024	1	CPU	1.965	1.191	6.069
2048	1	CPU	27.229	15.296	57.323
4096	1	CPU		177.608	
1024	32	GPU baseline	0.838	0.500	0.516
1024	32	GPU sync improved	0.642	0.316	0.343
2048	32	GPU P $\leq$ 32 blocks	2.864	1.427	2.096
4096	32	GPU 8 splits	21.902	8.841	16.767
8192	32	GPU 64 splits	159.058	62.064	135.793
12288	32	GPU 256 splits	419.030	196.971	460.912

Best timings for R=rectangle, T=triangle, P=parallelogram[11]

Micro-benchmarks [12]

Problem are described by the triplet <matrix size/block size/shape>.

- It has been put in evidence in a previous work[13] that with Mac OS operating system, using the GPU exclusively for CUDA or combining with UI display may affect the performance (GeForce 330M, architecture 1.2). With the new architecture (3.0, GeForce 650M), this difference has been reduced to less than 3.5% with the decoupling of UI and CUDA performing best. So in micro-benchmarks, we can safely ignore the graphic card usage.
- Synchronization between blocks
  - Removing `__threadfence()` before the synchronization is not syntactically correct but results still remains valid, this confirms the observation made by [**?**]. Speedup for <1024/32/*> are 67ms (parallelogram) 100ms (triangle) 180ms (rectangle).
  - In the parallelogram case, using all threads to monitor other blocks status instead of 1 thread results in a 6.4x speedup (22.72→3.52ms) on <1024/32/para>.
- Using multiple threads per matrix cell: In the case of a triangular matrix, at each step, the number of cells to be computed (on the diagonal) decrease while the computation complexity increase (there is one more dependency). According to [**?**], the solution lies in adaptive thread mapping, using more than one thread to compute one matrix cell, depending on the complexity. However, in our setup (memory layout+algorithm+hardware), we did not found any improvement by doing so. We want to explore the reason for that: we pose as hypothesis that the bandwidth is the bottleneck of our setup and test it.
  - First we need to prove that we use almost all the available memory bandwidth: for matrix multiplication, in a triangular matrix, we have

$$\text{Total transfer} = \frac{n(n+1)}{2} \text{ writes} + \sum_{i=0}^{n-1} 2i \cdot (n-i) \text{ reads}$$

[11] $\leq$ 32 blocks on my GPU to prevent deadlock

[12] Micro-benchmark do not provide extensive results as they stand only to validate implementation strategies.

[13] Performance Evaluation 2012 miniproject, *Performance Evaluation of Mersenne arithmetic on GPUs*

where each write is 10 bytes (long+short), and each read is 8 bytes (long). For $n = 4096$ we transfer 183'352'614'912 bytes which corresponds to 183.35GB. In 8.841 seconds, we can transfer theoretically at most $8.841 \cdot 28.8 = 254$GB. Hence we can say that 72% of the algorithm running time is spent into memory accesses.

– On a 4096 matrix, if we assume that ATLP card would have the same bandwidth as our card, their running time would be

$$2.57 \cdot (1 - .72) + 2.57 \cdot 0.72 \cdot \frac{102.4_{GB/s}}{28.8_{GB/s}} = 9.43\text{s}_{\text{ ATLP}} > 8.84\text{s}_{\text{ our}}$$

Which shows that our algorithm is comparable to theirs. However, we must avoid a close comparison because the fundamental hardware differences would make a tight computation almost intractable (additionally, we do not have ATLP source code).

As a conclusion, (1) we must remain away to invalidate their result as previous hardware generations might be subject to more constraint to our hardware and (2) we are on par, if not better with one of the best current implementation.

- Reducing the number of threads launched at different splits of the algorithm (especially in latest splits in rectangular and triangular shapes) does not bring any speedup. Even worse, it slows down slightly the computation. We might attribute this to a better constant transformation by the compiler. Hence, having many idle threads does not impede performance.
- Unrolling the inner loops (non-serial dependencies) a small number of time provide some speedup, for a 2048-matrix respectively 10.9% (rectangle, $2.765 \rightarrow 2.464$), 14.1% (triangle, $1.427 \rightarrow 1.225$) and 9.7% (parallelogram $1.539 \rightarrow 1.389$).

6 Future work

- Serial for problems larger than memory, use hybrid (Myers and Miller's algorithm with $\log_k$ with k depending on available memory) approach depending available memory
- Annotation on recursive functions to use dynamic programming like `@DynaProg def Fib(n:Int) = if (n<=2) return 1 else Fib(n-1)+Fib(n-2)`.

7 Conclusion

8 Planning — Work in progress

8.1 Write introduction

```
what exists
problem
how we solve it compared to other
how to evaluate
contributions (3): 3 tensed sentences

related work
benchmark => prove by evaluation intro statements
```

Steps

1. Re-implement vanilla combinators with lists
2. Add following problems and features:
   - Polygon triangulation (cyclic problem) (parallelogram matrix)
   - SWat with arbitrary gap (multi-track grammar) (rectangular matrix)
   - Zucker (rules of different complexity $O(n^2)/O(n)$)
   - Rules normalization
   - Rules dependency
   - Automatic backtracking
3. Test/proof parsers are correct — make sure implementation is correct
4. Automate test to compare against implementation
5. Build bottom-up vanilla version
   - Separate initialization (terminals) and non-terminals (speed up?)
   - In parallel implement the code generator for CUDA, hardcode in text function bodies for CUDA
6. LMS implementation: from normal to Rep/Exp
   - Re-implement bottom-up vanilla into LMS
   - Re-implement CUDA version and use functions body to infer CUDA code
7. Benchmarks — compare also versus other papers.
8. Write report

Roadmap

16.11	Rules normalization and automatic backtracking as in 5.2 GenScala on LMS + GenCuda + LMS CudaCompiler
23.11	Problem generalization: "cyclic keyword", Zucker problem / CudaLoop optimization
30.11	— Gap due to LMS missing knowledge
7.12	Benchmarking, grammar analysis
14.12	First though for larger than mem
21.12	Writing report
28.12	— holiday —
4.01	— holiday —
11.01	Writing report: implementation description and plan for future work
18.01	Writing report

Todo @TCK

Todo @Manohar

```
Look at string templating. """xxx $var xxx"""
- Cleanup LMS CUDA generator

Separate initialization (base cases, atoms) and processing (rules, recurrences) => less cases to

XXX: how to encode multi-dimensional matrices
1. assume they have the same type put one after another => different dimensions ok
2. assume of same size => put into a struct
=> but using different pointers seems more reliable => completely different matrices => fixed li
```

OldPlan

1. **User facing language:** similar to [**?**] or [**?**] *ADP fusion*. We want to reuse the transformation mapping (problem description) $\mapsto$ (kernel implementation) for a single element.
2. **Prototyping:** understand difficulties and share common base. Prototype of Smith-Waterman with arbitrary gap cost on CUDA/FPGA. Give an idea of how to implement the general case. Benchmark and compare both implementations. Aim to do as good an implementation for the specific platform (CPU/GPA) as possible.
3. **Formalize IR:** describe the intermediate representation, formalize the framework provided to the code generators (i.e. memory management, ...).
4. **Full compiler stack:** enrich the compiler stack from both top-down (translate best user-facing language parsers) and bottom-up (parametric code generators), core of the work.

```
1. optimize: terminals of bounded yield + binary decomposition if possible
2. transform into: axioms (init/fixed) + rules (iterations)
- define optimizations to IR
  - making serializable by using aggregation functions/transformations
  - avoiding the cost matrix by moving it into the wavefront
- scheduling (with memory loads for non-serializable)
- code generation for both platforms
  - possibly choice to use which platform for what part (load/compute @ cpu/gpu/fpga)

---------------------------------
Core function F:
- in: s,t strings
- in: neighbor costs: top, left, top+left
- in: neighbor stats: top, left, top+left
- out: backtrack information
- out: cost(i,j)
- out: stats

we might want to pack into memory the matrix data and make threads operate on more than one cell

Some ideas:
```

```
- define a way to pack the characters => less memory transfer (i.e. GATC=>4 letters in 1 char)
- operate on some larger word (ex 64 bits) to increase thread locality and reduce memory accesse
- write a kernel that takes stats from left and r

        stats (x)
         ||
         vv
stats -> KK -> new_stats (y')
(y)      || \
         vv  --+ backtrack info (Bxy)
     new_stats (x')

and compute the backtracking for its cell (optional as template boolean)

- multi-grain
  + within the 64b-words : multiple letters at once (thread level)
  + group threads into blocks that operate on (block level)
  + kernel that operates one after another (GPU/CPU level)
    => can we catch a stream's event at CPU level to alloc memory to get back results ?
- keep track of
  + stats O(m+n)
  + backtrack O(m*n)

- are we sure we need squares ?
  - rectangles can increase the full usage of all threads during (x-y) runs
  - assuming we put longest word vertically we play hot potato for stats
    left->right => block.x exchanges within block, last thread writes to
    global exchange memory (if short, so that it does not penalize running time)
  - we may go up to running in stripes
- if we put in diagonal-major data in the memory, why not stream all chunks like
  that using a cyclic buffer (?) => no empty slot
  ==> this might be useful for the backtrack information

- use extensively profiling : CUDA profiler

- Robust DP paper provides 2 insights: coalesced access + GPU synchronization
  => can we do a producer-consumer scenario at the block border so that we can
     execute a large diagonal horizontal/vertical swipe made of multiple blocks?
  => multiple swipes at different delays (so that we avoid syncing at every step

          s-->
  +----------------/------------+
t | B1            /             |
| +------/-------/              | KERNEL1 (keep benefit of t in shared mem)
| | B2  /                       | delay between b1 and b2 can be 1% of line length
v +----/------------------------+
```

```
  |                              |
  |                              | KERNEL2
Memory organized as
b1 -- --
b1 b1 --
b1 b1 b1
b1 b1 b1
.. .. ..
b1 b1 b1
b2 b1 b1
b2 b2 b1
b2 b2 b2
.. .. ..

For stats (aggregation of the table): we need to maintain both an horizontal and vertical front
For backtracking we need to maintain the whole table
... | . |
..A | B | min/max: Min(D)=Min(Min(C),Min(B))
----+---+ sum    : Sum(D)=Sum(C)+Sum(B)-Sum(A)
..C | D | avg    : sum both #cells and values then divide at appropriate cell
----+---+
we many not need to store what's the previous cell, however, the previous
cell information (usually limited range) is much more compact than the score(32-64 bits)

-------------------------------------------------------------------------
Some GPU algorithms: http://hgpu.org/?cat=11
Translation into C++: http://bibiserv.cebitec.uni-bielefeld.de/macports/resources/download/
CUDPP libraries (but awfully big resulting binary): http://code.google.com/p/cudpp/
13 dwarfs: http://developer.amd.com/afds/assets/presentations/2155_final.pdf
http://tutorials.jenkov.com/java-reflection/fields.html
http://lampwww.epfl.ch/~michelou/scala/scala-reflection.html

XXX: use TypeClass to put a predicate on types
-----------------------------------------
def fun[T: CanTranslateToC](...)
def fun[T](implicit ev:CanTranslateToC[T])

class CanTranslateToC[T] { def translate:String }
implicit def canTranslateInt = new CanTranslateToC[Int] = { def translate = "Int" }
```