ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Master project report

DynaProg for Scala

A Scala DSL for Dynamic Programming on CPU and GPU

Laboratory	Programming Methods Laboratory, LAMP, EPFL
Professor	Martin Odersky
Supervisors	Vojin Jovanovic, Manohar Jonnalagedda
Expert	Mirco Dotta, Typesafe
Student	Thierry Coppey
Semester	Autumn 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 project, we present *DynaProg*, a Scala DSL for dynamic programming on heterogeneous platforms which allow to write concise programs and execute them efficiently on GPUs.

Existing work is a DSL embedded in Haskell [**?**] with possible conversion to CUDA code [**?**], a compiler for a dynamic programming external DSL into C code [**?**] or ad-hoc CUDA implementations for specific problem classes [**?**], [**?**].

Our contributions are:

- A systematic approach to process data (top-down/bottom-up) and backtracking information (focus on running time and memory efficiency)
- A language embedded in Scala (DSL) to express DP problems concisely (based on ADP)
- Two implementations: Scala for CPU (features) and an CUDA for GPU (efficiency)

This project has been achieved in collaboration with Manohar Jonnalagedda. I also would like to thank the LAMP team, including Eugene Burmako, Andro Stucki, Vojin Jovanovic and Tiark Rompf who provided insightful advices and suggestions. I hope you will enjoy your reading.

Thierry Coppey

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 intermediate results), the solution of $F(n)$ would be obtained after n additions whereas without memoization it requires $F(n) - 1$ additions !

Formally, dynamic programming problems respect the Bellman's principle of optimality: *«An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision»*. This means that every intermediate result is computed only once, although it might be reused as basis for multiple larger problems, hence our first observation.

There exist various categories of dynamic programming:

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

Since the first category is inherently sequential (progress cannot be faster than one element at a time) 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 the previous category, 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 for problems like RNA folding, input are usually around few thousands. Multiple input problems also require more memory: for instance matching 3 sequences is $O(n^3)$-space complex. Since we target a single computer with one or more attached devices (GPUs, FPGAs), and since we plan to maintain data in memory (due to the multiple reuse of intermediate solutions) the storage complexity must be relatively limited, compared to other problem that could leverage the disk storage. Hence in general, we focus on problems that have $O(n^2)$-space complexity whereas time complexity is usually $O(n^3)$ or larger. We encourage you to refer to the section 2 for further classification and examples.

1.2 Scala and LMS

1.2.1 Scala

«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]

As the Scala programming language is initially developed by the LAMP, it seems natural to use it as host language for our project, however, we would list its features that makes it an interesting development language for this project:

- The functional programming style and syntactic sugar offered by Scala allows concise writing of implementation, analysis and transformations of our DSL, which would have been much more complex and tiresome in an imperative language like C.
- Scala is largely adopted in the industry, which makes both the adoption of related project easier and offer a steeper learning curve to their potential users.
- Finally, through the Java VM and JNI interface, Scala offers the possibility to load dynamically external libraries, to leverage best underlying hardware by mixing with CUDA kernels to obtain optimal performance.

1.2.2 LMS

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.

LMS can be leveraged to transform Scala code into its C-like equivalent. However, the concern in this project is that the code for the GPU would be sensibly different from the original CPU code as both implementation serve different purposes: CPU version (Scala) is more general whereas the GPU version trades some functionalities for performance and suffer additional restrictions, in particular for memory management and alignment. Hence the use of LMS would be restricted to user-specific function, over which our DSL has no control.

[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 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)

$$M_{(i,j)} = \max \left\{ \begin{array}{l} \dots \\ M_{(i,j-1)} \\ \max\limits_{i<k<j} [M_{(i,k)} + M_{(k+1,j)}] \end{array} \right.$$

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:

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

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:

- Vectors of size n are indexed from 0 to $n-1$, matrices follow the same convention ($M_{(m,n)}$ is indexed from $(0,0)$ to $(m-1, n-1)$)
- Matrices dimensions are implicitly specified by number of indices and their number of elements is usually the same as the input length (possibly with 1 extra row/column).
- 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.

Although not necessary to understand the project, the description of some of the most common dynamic programming problem is relevant to capture the essence of the dynamic programming processes and be able to compare and search for similarities among problems. Would the reader be familiar with dynamic programming, he could immediately jump to the next section.

A tighter analysis on the alphabet and intermediate results size is done because FPGA was also considered as a possible execution platform.[2]

[2]Initially, the project envisioned a collaboration with Nithin George, a PhD student at Processor Laboratory (LAP), EPFL. Unfortunately, at the time of writing, progress in this direction is not sufficient to be covered in this report. However, FPGA is considered as a future deployment platform for a particular class of problems.

2.2.1 Smith-Waterman (simple)

1. Problem: matching two strings S, T with $|S| = m, |T| = n$.
2. Matrices: $M_{(m+1)\times(n+1)}, B_{(m+1)\times(n+1)}$
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:

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

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

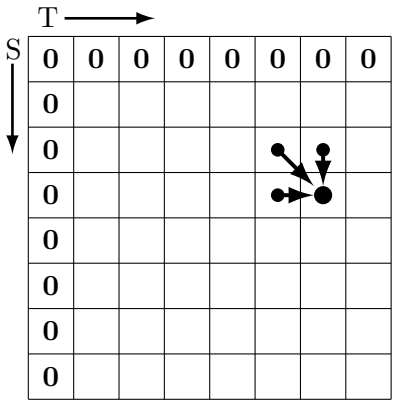


Figure 1: Smith-Waterman (affine gap cost) dependencies (serial)

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.
   - Since first line and column of the matrix (corresponding to a matching with one string being empty) are zeroed, their initialization might be omitted, but this would implies more involved initialization and computations, which is cumbersome.
   - Padding: since to fill the i^{th} row we refer to the $(i-1)^{\text{th}}$ character of string S, we could prepend to both S and T an unused character, so that matrix and input lines are aligned. Hence valid input indices would become $S[1 \cdots m]$ and $T[1 \cdots n]$.

2.2.2 Smith-Waterman with affine gap extension cost

1. Problem: matching two strings S, T with $|S| = m, |T| = n$.
2. Matrices: $M_{(m+1)\times(n+1)}, E_{(m+1)\times(n+1)}, F_{(m+1)\times(n+1)}, B_{(m+1)\times(n+1)}$
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:

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

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

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

   That can be written alternatively as:

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

   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

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

6. Backtracking: similar to Smith-Waterman (simple)
7. Visualization: same as Smith-Waterman (simple)
8. Optimizations: 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 respectively horizontal and vertical wavefront, hence removing the need of two additional matrices.

2.2.3 Smith-Waterman with arbitrary gap cost

1. Problem: matching two strings S, T with $|S| = m, |T| = n$ with an arbitrary gap function $g(x) \geq 0$ where x is the size of the gap. Without loss of generality, let $m \geq n$ (swap inputs if necessary). Example penalty function could be[3] $g(x) = m - x$.
2. Matrices: $M_{(m+1)\times(n+1)}, B_{(m+1)\times(n+1)}$
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:

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

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

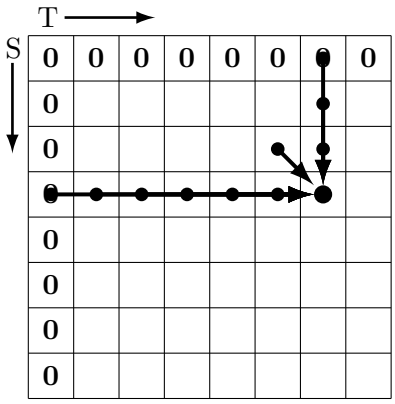


Figure 2: Smith-Waterman (arbitrary gap cost) dependencies

8. Optimizations: The dependencies here are non-serial, there is no optimization that we can apply out of the box here. In general, this problem has an $O(n^3)$ complexity (whereas simple and affine gap variants are $O(n^2)$).

[3] Intuition: long gaps penalize less, at some point, one large gap is better than matching with 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 a symmetric matrix (can be stored as a triangular matrix with 0 diagonal that can be omitted), hence $|S| = \frac{n(n-1)}{2} = N$.
2. Matrices: $M_{(n+1)\times(n+1)}, B_{(n+1)\times(n+1)}$ *indices denote «first vertex, last vertex».*
   Upper triangular matrices including main diagonal. Note that the vertex n is the same as the vertex 0 due to the cyclic nature of the problem.
3. Alphabets:
   - Input: $\Sigma(S_{(i,j)}) = \{0..m\}$ with $m = \max_{i,j} S_{(i,j)}$ determined at runtime[4].
   - Cost matrix: $\Sigma(M) = \{0..z\}$ with $z = m \cdot (n-2)$ (a triangulation of a polygon of n edges adds at most $n-2$ edges).
   - Backtrack matrix: $\Sigma(B) = \{stop, 0..n\}$ (index of intermediate edge)
4. Initialization: $M_{(i,i)} = M_{(i,i+1)} = 0, B_{(i,i)} = B_{(i,i+1)} = stop \quad \forall i$
5. Recurrence:
$$M_{(i,j)} = \left\{ S(i,j) + \max_{i<k<j} M_{(i,k)} + M_{(k,j)} \;\middle|\; k \right\} = B_{(i,j)}$$
   Intuition: triangulate the partial polygon $(i, ..j)$ recursively. 3 cases for the last triangle:
   - Given 2 triangulations $(1..k)$ and $(k..n)$, we close the polygon with $\triangle(1, k, n)$
   - Given a triangulation $(1..n-1)$, we close the polygon with $\triangle(1, n-1, n)$
   - Given a triangulation $(2..n)$, we close the polygon with $\triangle(1, 2, n)$

   Since the edge to close the last triangle is already part of the polygon, its cost is 0.
6. Backtracking: Add the edges in the set given by the set $\mathrm{BT}(B_{(0,n)})$ where
$$\mathrm{BT}(B_{(i,j)} = k) \mapsto \begin{cases} \{\} & \text{if } k = stop \\ \{(i,j)\} \cup \mathrm{BT}(B_{(i,k)}) \cup \mathrm{BT}(B_{(k,j)}) & \text{otherwise} \end{cases}$$
7. Visualization:

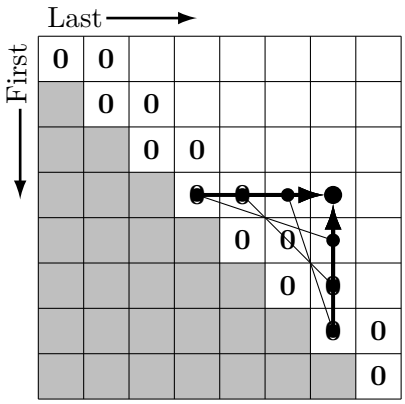


Figure 3: Convex polygon triangulation dependencies

8. Optimizations: if the cost of edges between contiguous vertices is 0, we do not need to handle special cases in the DP program (i.e. existing edges cannot be added).

[4]We need to have statistics about S, this is where dynamic compilation might play 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 $B_{(0,n-1)}$. Use the following recursive function for parenthesizing

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

7. Visualization:

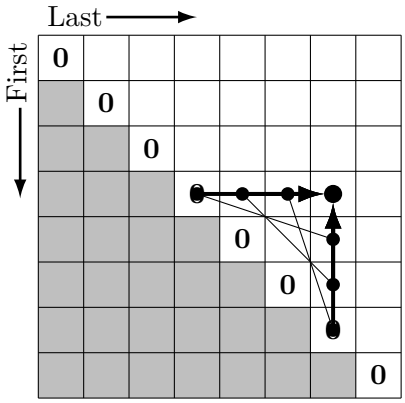


Figure 4: Matrix chain multiplication dependencies

8. Optimizations: we could normalize the semantics of indices and use $(n+1) \times (n+1)$ matrices where the meaning of cell (i, j) would be $\underset{i \leq k < j}{\mathrm{chain}}(A_k)$.

2.2.6 Nussinov algorithm

1. Problem: folding a RNA string S over itself $|S| = n$.
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:

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

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

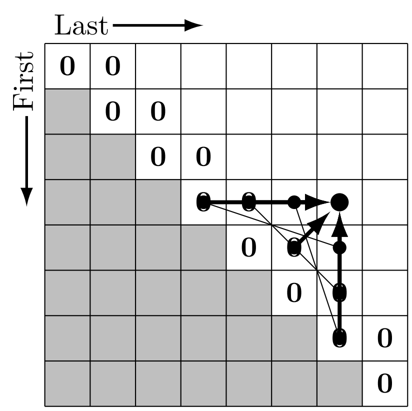


Figure 5: Nussinov dependencies

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 $|S| = n$.
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 $z \leq n \cdot b + c$
     - $\Sigma(F) = \{0..y\}$ with $y \leq \min(F_0, z \cdot n)$
   - Backtrack matrices:
     - $\Sigma(BW) = \{stop, L, R, V, k\}$
     - $\Sigma(BV) = \{stop, hairpin, stack, (i, j), k\}$ with $0 \leq i, j, k < n$
     - $\Sigma(BF) = \{stop, P, k\}$ 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]:

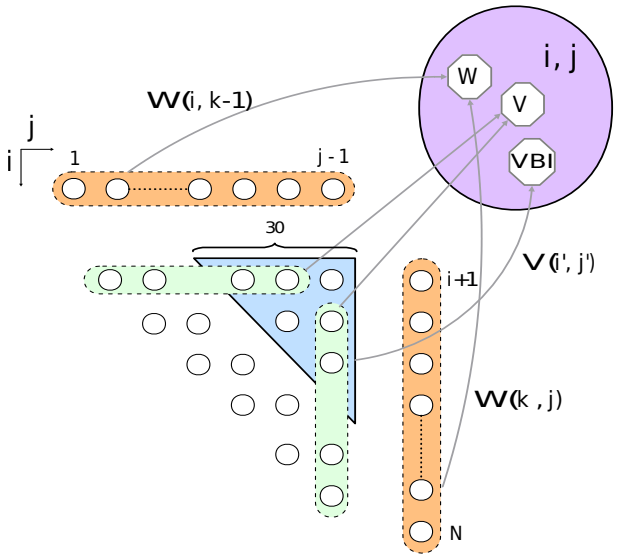
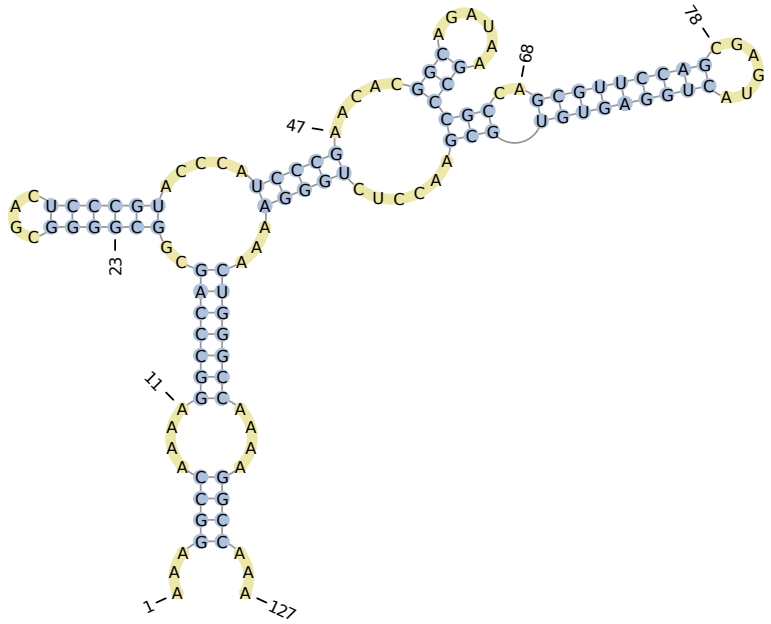


Figure 6: Zuker folding dependencies

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



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).

Figure 7: An example of an RNA folded into a secondary structure

8. Optimizations: 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. 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 aim of this section is to demonstrate that the problems described above are very similar or encompass a significant part of the common dynamic programming problems. (There is an hyperlink on the problems name to their detailed description).

Serial problems	**Shape**	**Matrices**	**Wavefront**
Smith-Waterman simple	rectangle	1	–
Smith-Waterman affine gap extension	rectangle	3	(can replace 2 matrices)
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	–

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

Table 1: Classification of related problems

2.3.1 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 (binary search is more efficient).
- Coin Change: 1D non-serial

These algorithm also involve dynamic programming. However, we do not evaluate thoroughly their shape and number of matrices as this is not relevant in the scope of this project.

- Floyd-Warshall
- 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

- RNA structure prediction
- Recursive least squares
- Bitonic tour
- Shortest path, Shortest path in DAGs, All pair shortest paths, Independent sets in trees
- Subset Sum, Family Graph
- Optimal Binary Search Trees
- Independent set on a tree
- More dynamic programming problems from Wikipedia

2.3.2 Conclusion

In the rest of the project, we use a different description of the problems that is based on ADP [**?**], which is more convenient but does not share much with the above description (even though ultimately the executed computations are very similar). Although not of immediate use, the description of the above problem and ad-hoc CUDA implementation of three of them (Smith-Waterman with arbitrary gap cost, Matrix chain multiplication and Convex polygon triangulation) helped us to understand:

1. There is a difference between dynamic programming as seen in algorithmic classes and their concrete implementation, mainly because special care must be taken for correct indices and preventing off-by-one errors.
2. Problems can be classified in two categories: single track (input) and two-tracks (2 input sequences). Most of the common dynamic programming problems fall in these two categories.
3. Sometimes matrices are initially padded with zeroes (or initial value), although this might be ignored at algorithm design, care must be taken for these special values and their inclusion in the matrix should be decided according to the complexity of the recurrence formula.
4. Incidentally, we proposed a cyclic variant of the convex polygon triangulation, which uses a parallelogram matrix instead of a triangular one (upper right triangle extended by a lower triangle on the right). Unfortunately, this proved to be based on an erroneous recurrence relation analysis, which could be reduced to the more natural triangular case.
   Although the polygon triangulation problem can be solved with a regular triangular matrix and we have not found real problem requiring a parallelogram matrix, we still present this version in the analysis part of the project, which could be justified for cyclic problems that require dynamic programming for each unfolding position (that is for all position in the circular structure, break the cycle at this position, and solve the dynamic programming problem on the resulting flattened sequence). For example, one could be interested in finding the longest subsequence verifying some property in a cycle, such that the subsequence score changes if it is rotated.

3 Architecture design and technical decisions

3.1 User facing language requirements

The field of dynamic programming has been influenced in the recent years by a methodology known as Algebraic Dynamic Programming which uses a grammar and an algebra to separate between the parsing and the score computation:

> *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.*

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 can extend parsers to backtracking parsers or pretty-printers. Additionally, we want to support the following features:

1. **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 $\Omega(n^3)$ storage requirements that limits the problems size more dramatically.
2. **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.
3. **Input 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.
4. **Single-result on devices:** The general ADP framework supports multiple solutions for intermediate results. Such functionality is easy to support in Scala; however, memory management hampers the performance of the GPU implementation. To overcome this issue, the user could manually manage the memory, but this would defeat most of the benefits of automatic code generation. Hence the trade-off solution we propose is to restrict ADP to only one optimal result on CUDA, while leaving the freedom to obtain co-optimal (or even all possible solutions) with the Scala version.
5. **Automatic backtracking:** Since efficient code has to be devised we imposed 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 would like to to automate the backtracking to fulfill goals of usefulness, efficiency and ease-of-use in device-specific implementation:

- Leaving the backtrack implementation to the user would force him to memoize the backtracking information together with the results (backtrack would grow towards final result and duplicate unnecessarily information), hence requiring both $O(n^3)$ space and memory management features on devices.
- Enforcing automatic backtracking presents the advantage to ensure constant size for intermediate results, hence ensuring an $O(n^2)$ storage requirement. 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).

6. **Yield analysis:** in the vanilla ADP, the user has to define for each concatenation the minimal and maximal length of the subsequence on each side. Although non-emptiness information is necessary to avoid infinite recursion in the parsers, forcing an explicit definition can become cumbersome for the DSL user. We want to provide an automatic computation of concatenation boundaries, while at the same time leaving the possibility to manually specify it for maximum flexibility.

The support of these features has the following implications:

- **Dependency analysis:** Since we target GPUs (and FPGAs) which are massively parallel architecture, a top-down execution using hash tables is impractical (fallback computation if element is not present is hard to parallelize), hence we need to construct the result tree bottom-up, therefore ensure that the (partial) evaluation order between rules is respected.
- **Normalization:** in order to automate the backtracking, we need the rule to present a certain shape so that we can define uniquely the backtracking information (in particular we want to distinguish between alternatives). Also we need to maintain coherency between the Scala and the CUDA version so that they can inter-operate: we would like to reuse the backtracking information (from CUDA) to do actual processing in Scala (for instance pretty-printing or effectively multiplying matrices).
- **Optimizations:** An possible optimization 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. Unfortunately this analysis is very involved: we need to solve the following problem:
  Given f , find a pair of functions (f_1, f_2) or (f_3, f_4) such that[6]

$$\begin{array}{rcll} f(i,k_1,k_2,j) & = & f_1(i,f_2(i,k_1,k_2),k_2,j) & \wedge \\ \min\limits_{i<k_1<k_2<j}\big[f(i,k_1,k_2,j)\big] & = & \min\limits_{i<k_2<j}\big[f_1(i,\min\limits_{i<k_1<k_2}\big[(f_2(i,k_1,k_2)\big],k_2,j)\big] & \vee \end{array}$$

$$\begin{array}{rcll} f(i,k_1,k_2,j) & = & f_3(i,k_1,f_4(k_1,k_2,j),j) & \wedge \\ \min\limits_{i<k_1<k_2<j}\big[f(i,k_1,k_2,j)\big] & = & \min\limits_{i<k_2<j}\big[f_3(i,k_1,\min\limits_{k_1<k_2<j}\big[(f_4(k_1,k_2,j)\big],j)\big] & \end{array}$$

  Since this require complex mathematical analysis that are out of the scope of the project, we leave this optimization to the responsibility of the user.
  Another optimization that is easier to implement is dead rule elimination. Traditional dead code elimination only reduces the size of the generated code. In the GPU implementation, this optimization will not only reduce the binary size but also the CUDA memory consumption as well as speed-up computation (since all rules are executed together).

[6]The first equation denotes breaking in two functions, the second is Bellman's optimality principle.

3.2 Parsing grammar (ADP)

In this section, we provide a concise description of the Algebraic Dynamic Programming. Would the reader be interested in the mathematical details of the notation, we encourage him to read the section 3 of [**?**].

ADP is a formalization of parsers that introduces a distinction between the **parsing grammar** (recognition phase) and an associated **algebra** (evaluation phase). Such separation makes possible to define multiple algebra for the same grammar. This has two main applications:

1. Experiment variants with the same grammar: for example, Needleman-Wunsch and Smith-Watermann share the same grammar but have a different evaluation algebra
2. Use an evaluation and execution algebra: a dynamic programming problem is solved in two steps: computing one optimal solution and applying it over actual data. For example in matrix chain multiplication, the first step solves the underlying dynamic problem by evaluating the number of necessary multiplications, the second step *effectively* multiplies matrices according to the order previously defined.

Practically, an ADP program is constituted of 3 components: a **signature** containing functions signatures, which are implemented by **algebrae** and a **grammar** containing parsers that make use of the functions defined in the signature. The concrete program instance mixes-in the algebra with the grammar. The grammar parsers intermediate results are memoized in an array (tabulation parser). A parser usually consist of a tree of:

- **Terminal:** operates on a subsequence of input elements and returns either its content or position (or a failure if the sequence does not fit the terminal).
- **Filter:** accepts only subsequences matching a certain predicate. The condition is evaluated ahead of its actual content evaluation.
- **Or:** expresses alternative between two different parsers and returns their result union.
- **Concatenation:** constructs 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.
- **Map:** this parser transform its input using a user-defined function. It is typically used to transform a subword into a score that can later be aggregated.
- **Aggregation:** the aggregation applies a functions that reduces the list of results, typically minimum or maximum, but the function can be arbitrarily defined.
- **Tabulation:** the tabulation's primary function is to store intermediate results and possibly serve as connection point between different parsers.

Additionally, the signature must define input alphabet (`Alphabet`), and output alphabet (`Answer`) can be defined either in the signature or in the algebra. Finally, the grammar needs to have a starting point, denoted as axiom. Finally, the default aggregation function h must be defined[7]. To make it more clear, we propose an example of the matrix chain multiplication problem[8].

[7] Although aggregation usage is not mandatory in the framework, we force the existence of an aggregation function over the ouput type so that we can use it to aggregate windowed results.

[8] The original ADP framework was embedded in Haskell, however, we assume that the reader is more familiar with Scala notation and immediately present the syntax of our implementation.

```
trait MatrixSig extends Signature {
  type Alphabet = (Int,Int) // Matrix(rows, columns)
  val single:Alphabet=>Answer
  val mult:(Answer,Answer)=>Answer
}

trait MatrixAlgebra extends MatrixSig {
  type Answer = (Int,(Int,Int)) // Answer(cost, Matrix(rows, columns))
  override val h = minBy((a:Answer) => a._1)
  val single = (a: Alphabet) => (0, a)
  val mult = (a:Answer,b:Answer) =>
    { val ((m1,(r1,c1)),(m2,(r2,c2)))=(a,b); (m1+m2+r1*c1*c2, (r1,c2)) }
}

trait MatrixGrammar extends ADPParsers with MatrixSig {
  val axiom:Tabulate = tabulate("M",
    (el ^^ single | axiom ~ axiom ^^ mult) aggregate h)
}

object MatrixMult extends MatrixGrammar with MatrixAlgebra with App {
  println(parse(Array((10,100),(100,5),(5,50)))) // List((7500,(10,50)))
}
```

Listing 1: Matrix chain mulitiplication DSL implementation
with or: | map: ∧∧ concatenation: ∼

This program grammar can also be expressed in BNF:

$$\begin{array}{rcl} axiom & ::= & \text{matrix} \\ & | & axiom\ \ axiom \end{array}$$

and it encodes the following recurrence (cost only):

$$M_{(i,j)} = \begin{cases} 0 & \text{if } i+1=j \\ \min_{i<k<j} M_{(i,k)} + M_{(k,j)} + r_i \cdot c_k \cdot c_j & \text{otherwise} \end{cases}$$

Notice that the semantics of indices differ slightly from the problem presented in 2.2.5; this is because empty chain are made expressible (denoted $M_{(i,i)}$, single matrices are denoted $M_{(i,i+1)}$).

3.3 Recurrences analysis

3.3.1 Dead rules elimination

Dead rules[9] elimination analysis is straightforward: starting from the grammar's axiom, recursively collect all tabulations involved in the computation in R (set of rules that are mandatory in the grammar). The dead rules $D = S \setminus R$ (where S is the set of all tabulations) can safely be removed from the grammar rules. Although seemingly useless for the Scala implementation, this step is necessary to maintain coherency between Scala and CUDA rules numbering (that happen in a later stage on the valid rules). In CUDA, this analysis not only provides dead code elimination, but it also prevents useless computation execution, since all rules are computed sequentially for a particular subsequence before the next subsequence is processed.

[9] *Rule* denotes a tabulation belonging to the grammar; both terms refer to the same concept, with the subtle difference that *rule* emphasizes its grammar membership.

3.3.2 Yield analysis

Since the original ADP introduces many concatenation combinators[10] to differentiate empty/non-empty, and floating/fixed-length concatenations, it is quite involved for the programmer to make sure that the concatenation operators exactly match the size of each pair of subsequence involved. Additionally the priority of operators varies in Scala, depending the operator's first character whereas it is possible to specify arbitrary priorities in Haskell. To overcome these issues, we propose to automate the computation of the minimum/maximum length of (subsequences) parsers.

Parsers are made of terminals, concat, operations (aggregate, map, filter) and tabulations; minimum/maximum yield of terminals is set, hence it only remains to assign appropriate yield sizes to tabulations; other operations simply propagate that information. It is possible to obtain the yield size of tabulations using the following algorithm (assuming recursive parsers contain at least one tabulation that is part of the loop):

1. Set the yield minimum size of all tabulations to a large number M_0 (such that all tabulation would reasonably have a minimum yield size smaller than M_0)
2. Repeat k times (k is the number of rules of the grammar): for each rule, compute its minimum yield size and update its value (without recursion at tabulations). This would lead to a correct minimum yield size because the terminals provide a minimum size and this might need at most k iteration to propagate across all rules.
3. Set the maximal yield of all the rules to the minimal value. For each rule, compute recursively up to depth k (where the depth is computed as the number of tabulation traversed) the maximum yield size. If the depth reaches the maximum k, there is a loop between tabulations, hence return infinity.

The last part of this algorithm seems to have an exponential complexity, however, if we consider depth-first search and return as soon as we reach infinity, we might reduce its complexity to $O(k^2)$. Obtaining the yield size of tabulations provides the following benefits:

- Minimum size: prevents self-reference parsers (on the same subsequence) and avoid considering subsequences yielding empty results (hence slightly reducing time complexity).
- Maximum size: allows to reduce the size of the result and backtrack matrix to $O(m \cdot n)$ instead of $O(n^2)$ (where m is the maximum yield size), possibly providing substantial space savings. As the rules with bounded maximum yield are very rare, we did not implement this optimization, although we might consider it for future work.

3.3.3 Dependency analysis

Let us introduce the concept of dependency: a dependency between tabulations A, B denoted $A \to B$ exists if $B_{(i,j)} = f(A_{(i,j)})$, that is if the result of B depends of the result on the *same* subproblem computed in A. A grammar is unsolvable if there exist a dependency loop between parsers ($A \to ... \to A$). Such case only happen when there is no concatenation or a concatenation with an empty word. Being able to track the dependencies of tabulations and infer a computation order between them has two benefits:

- Although seemingly unnecessary in a top-down approach (Scala), this analysis detects dependency loops which would result in infinite call loops (stack overflow) at execution.
- Ordering tabulations is critical in a bottom-up approach (CUDA) to make sure that all dependencies are valid before an element computation is triggered.

[10] ADP's original combinators are: $\sim\!\sim, \sim\!\sim *, * \sim\!\sim, * \sim *, - \sim\!\sim, \sim\!\sim -, + \sim\!\sim, \sim\!\sim +, + \sim +$.

3.4 Backtracking

In order to produce an efficient transformation from ADP-like problem description 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 parsers 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 which is not implemented in ADP, unless we consider the whole backtrack being part of the scoring (which has a big performance impact and non-constant memory requirement issues that make such GPU implementation hard and not desirable). Additionally, since the backtracking data 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 sub-solutions combination that contribute to the problem solution. To do that we need to apply the following transformations
  1. Normalize rules and identify them uniquely by exploding alternatives: each rule is decomposed into the union of multiple sub-rules uniquely identified by an index, where sub-rules do not contain alternatives (Or parsers). Let s a subrule and r_s its identifier, we also establish a mapping T from identifier to subrule: $(r_s \to s) \in T$.
  2. Let $\text{cc}(r_s)$ the number of concatenation contained in the sub-rule r_s. The data element corresponding to a rule is a pair (score, backtrack) and is named after the tabulation.
     - The score part consists of a user-defined type (a composite of primitive types, case classes and tuples)
     - The backtrack part is a tuple $(r_s, (k_1, k_2, \ldots, k_m))$ where m is the maximal number of concatenations occurring in the enclosing rule of r_s; more formally $m = \max_z \left[\text{cc}(r_z) | r_z \in \text{rule}(r_s)\right]$, and let $m_s = \text{cc}(r_s) \leq m$.
  3. During the matrix computation of cell (i, j), if the sub-rule r_s applies, the backtrack will be set as $(r_s, (k_1, k_2, \ldots, k_{m_s}))$; with $i \leq k_1 \leq k_2 \leq \ldots k_{m_s} \leq j$. Note that if the backtrack occupies a fixed-length memory, the backtrack will contain exactly m indices, hence $k_i \,|\, m_s < i \leq m$ will be unspecified.
  4. During backtracking, when reading the cell (i, j) with backtrack $(r_s, (k_1, k_2, \ldots, k_m))$, given r_s, we recover $s = T(r_s)$, the sub-rule that applies. Hence we can determine m_s, which allows us to enqueue the subsequences $(i, k_1), (k_1, k_2), ..., (k_{m_s}, j)$ for recursive backtracking. If s 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 $((i, j), r_s, (k_1, k_2, \ldots, k_{m_s}))$ where (i, j) denotes the subsequence on which the sub-rule $T(r_s)$ has to be applied with concatenation indices $(k_1, k_2, \ldots, k_{m_s})$.

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

In order to reduce the storage required by the backtracking indices, we can avoid storing fixed indices (where at least one of the two subsequences involved in a concatenation has a fixed size) and leverage the knowledge contained in s to reconstruct the appropriate backtrack.

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 in-order, applying for each rule the underlying transformation, and storing intermediate results (i.e. in a hash map) until they are processed by another rule. Since there is only one consumer for each intermediate result, every read in the hash map can remove the corresponding value, hence reducing the memory consumption. Ultimately, only the problem solution will be stored in the hash map.

3.4.1 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 intermediate 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 traces. To do that, we maintain a multiplicity counter in each backtrack path:

- While there is an unique 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 explore the k first paths with multiplicity 1 and safely ignore the other (as we only need k distinct results).

Now it remains the problem to generate all possible results and check whether they are valid candidates. To do that we 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 alternatives: since we know exactly (by the subrule id r_s, 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 $O(f(n))$ 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 set by the backtrack, the filter must have been passed at matrix construction time.

3.4.2 Backtracking complexity

Since the ADP parsers can store multiple results, we are interested in measuring the overhead of k-best backtracking (compared to single-element backtracking).

For single-element backtrack, we only need to «revert» the parser to find involved subsequences, which is linear in the parser size (because the backtracking identifies uniquely the alternative and concatenations indices).

At every backtrack step, either:

- The sequence is removed at one element, which leads to maximal backtrack length of n.
- The sequence is split in k subsequences, with recurrence $f(n) = k \cdot f(n/k) + 1$ by solving this recurrence we see that there can be at most n final nodes and n intermediate nodes (when $k = 2$). 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[11].

Another special case we might be interested in is to obtain all co-optimal solutions (all the solutions that have an optimal score). We need here to notice that in the parser reversal, no sub-solution is discarded, because either it is not co-optimal (and would have been discarded at an earlier stage) or it is co-optimal, hence contributes to create a co-optimal result. It follows that the complexity of co-optimal solutions backtrack is proportional to the number of solutions.

3.4.3 Backtrack utilization

Since the dynamic programming problem may help solving a larger problem[12], we need to be able to apply the result of the dynamic programming computation in a different domain. The easiest way to do that, is to reuse the same input and grammar, but use a different output domain, and only compute the result of the trace obtained from the DP backtrack.

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 algebra to operate on another domain, and reuse the same grammar. The key point here is to notice that a backtrack trace of a parser can be reused by another, providing that they share 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 another parser operating on «concrete matrices» domain that will do the actual matrix multiplication (instead of the cost estimation).

3.5 CUDA storage: from list to optional value

Since lists are natively supported, it is natural to gather parsers results into lists in Scala implementation. However, when it comes to efficient CUDA implementation, lists must be avoided because memory allocation (and management) is not very efficient [**?**]. A workaround might be to use fixed-length lists but we assume here that the programmer is most often interested in a single optimal solution (this also alleviates the complexity of constructing multiple distinct backtrack traces). Even if this restriction apparently greatly simplifies the design, issues might arise for how to represent and deal with empty lists and how to minimize the amount of used memory:

- **Minimizing memory consumption:** Under the restriction that we only store the best result, we first need to transform aggregation such that they return at most one element. Useful aggregator belonging to this class are quite limited: minimum/maximum (optionally

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

[12] 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.

with respect to an object's property), count and sum[13], hence it is possible to provide the user with a tailored implementation. To benefit of this fixed memory aggregation, we need to do some structural transformation of inner parsers. In general, a tabulation T is the root of its evaluation tree, with leaves being either other tabulations or terminals, note that any of the 5 intermediate element can appear multiple times (or not being present) and in any order:

$$T < \textbf{Aggregate} < \textbf{Or} < \textbf{Filter} < \textbf{Map} < \textbf{Concat} < (\text{Tabulation} \mid \text{Terminal})$$

For obvious performance reasons, we want to maintain aggregations wherever they are present. However, we can partially normalize the rest of the evaluation tree:

- We must ensure that all parsers potentially generating multiple possibilities are aggregated. To do that, we simply wrap the original parser in an h-aggregation (where h is the default aggregation function that must be specified by the user)
- Since the aggregation now operates on a single element, we want to push it as close to the leaves as possible, as long as we do not change operational domain[14].
- Filters can be hoisted within the same concatenation / alternative
- Alternatives must be hoisted outside of maps and concatenations, the reason being that we need to avoid maintaining lists of candidates (that will be later aggregated).

We summarize the required transformations in the following table:

Outer ╲ inner	Aggregate	Or	Map	Filter	Concat
Aggregate	merge?	swap_R	—	swap_P	—
Or	—	simplify?	—	swap_P?	—
Map	—	swap_R	fuse	swap_P	—
Filter	—	—	—	merge?	—
Concat	—	swap_R	—	—	—

R = required, P = performance optimization

Table 2: Parsers normalization towards CUDA code generation

Note that there can be no swap with Map and Concat internal parsers due to domain change. Fusing is done by the C compiler (declaring mapping functions inline).

- **Handling nested aggregations:** since a tabulation might contain nested aggregation, they must be preserved in order not to increase its time complexity. To do that, each internal aggregation has to define its own intermediate score and backtrack variables, whereas outermost aggregation can directly write in the cell of the cost/backtrack matrix.
- **Failure handling:** a parser can either be successful and return a valid result or fail and return no result; failure can happen in terminals, (input) tabulations and filters. These 3 cases can be reduced to one by wrapping terminals and tabulations into a filter that checks validity conditions. It remains to discuss failure encoding strategies:
  - **Special «empty» value:** The benefit of such encoding is a reduced number of memory accesses; indeed since we anyway need to access the value to make computations,

[13] Notice that all these operations can be implemented with a folding operation on a single variable.

[14] We cannot push an aggregation through a mapping or a concatenation operation.

we do not generate additional memory accesses to check the validity of the value. The drawback of such approach is that it becomes necessary to specify a special `empty` value that cannot be used, except to denote the absence of result. Since types can be arbitrary at every step of the parser, it becomes cumbersome to ask the DSL user to provide a special value for every intermediate result.

- **Backtrack encoding:** Reusing the backtrack to encode the validity of a result is a more general approach, and allow greater flexibility for the user. Indeed, since we maintain sub-rules identifiers in the backtrack, it suffice to use a special identifier to denote that related value is invalid. This approach also work with nested aggregations by storing intermediate sub-rule identifiers that would only grant the validity of the related value.

Since backtrack encoding comes at the price of an additional memory access to test the validity (memory accesses usually accounts for most of the time on CUDA devices), it is also relevant to allow the user to completely disable this test to speed-up computations.

3.6 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. Without loss of generality, let the underlying dynamic programming matrices be of dimension $m \times n$.

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 aggregations that are made along the dimensions of the matrix do 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 symbol (from input alphabet) is denoted by S_I.

3.6.1 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:

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

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.

3.6.2 Large problems

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

3.6.3 Non-serial problems

Non-serial problems need to potentially access all elements that have been previously computed. We restrict ourselves to the following dependencies[15]:

- 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). 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 $S_C = 4, S_B$[16] $= 4, S_I = 1, S_W = 0$ and $M_D = 2^{30}$:

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

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

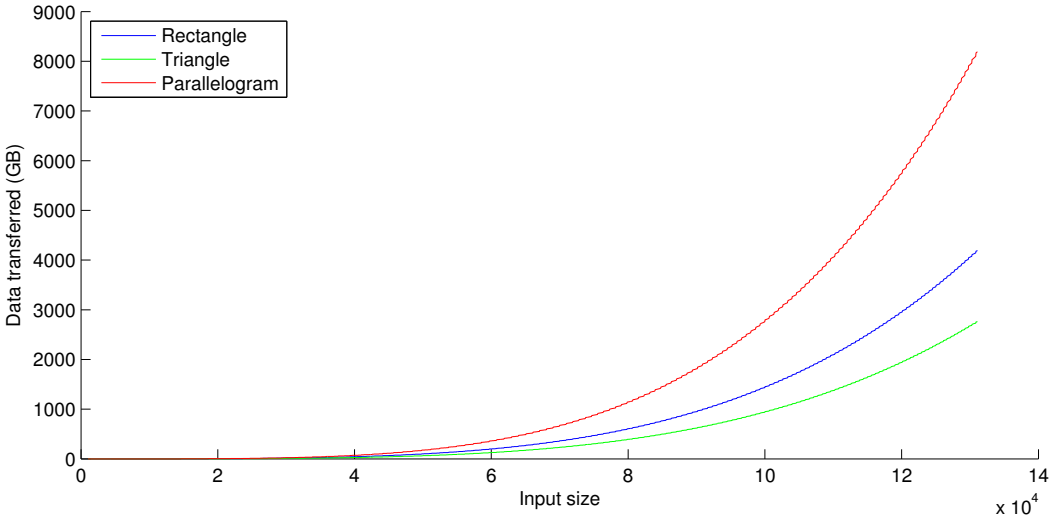


Figure 8: Transfer overhead for non-serial problems larger than device memory

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 and 6 days $10h^{(P)}$, assuming there is no other scalability issues, hence transfer would account respectively for $0.1\%^{(R)}$, $0.2\%^{(T)}$ and $0.3\%^{(P)}$ of the running time. Although this overhead seems appealing compared to the computation time, the total running 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.

3.6.4 Serial problems

The serial problems have the interesting property to access a fixed number of previous elements. These elements can be either stored either explicitly in a scoring matrix or implicitly as moving aggregation into the 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), line-wise or column-wise. 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 given the same running time. Hence, it seems natural to let serial problems grow larger than the memory.

Mixing the dependency property and size requirements, we can split the matrix into submatrices, 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 $4\times$ more storage space would only result in a $2\times$ speedup (measuring only the computation time). Hence a hybrid approach would be to decide a k such that at each step, we partition the desired sub-matrix 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.

Finally, although such problem is interesting because targeted platforms could include FPGA, where efficient implementations exist [**?**], several reasons made us considering this class of problem as a future work:

- The most prominent problem in this category is Smith-Waterman, for which an ad-hoc efficient implementation already exists[**?**],[**?**]. Additionally, the authors are planning to write extensions to support some variants of this problem like Needleman-Wunsch.
- The implementation sensibly differs from the class of small problems, as the solving strategy is completely different from non-serial small problems, hereby requiring larger development time that would be out of the scope of this project.
- Finally, such implementation would be only valuable for problems that are larger than the memory device, whereas smaller problem could perfectly use the existing implementation.

3.7 Memory layout

A major bottleneck on massively parallel architecture with shared memory (like CUDA) is the global memory access. To address it, according to the manufacturer documentation, it would be best if all threads access contiguous memory at the same time (coalesced memory access). This is justified by the memory hardware architecture, where additional latency (precharge rows and columns of the memory chip) is mandatory to access data at very different positions. Since all thread share the same global memory (and most of the time the same scheduler), accessing non-contiguous memory cumulates latencies before progress can be made.

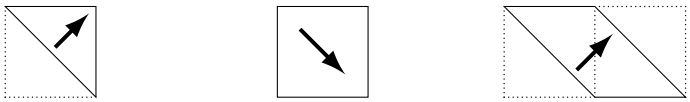


Figure 9: Matrix shapes, the arrow indicates the computation progress direction

Since progress is made along the diagonal of the matrices, a naive row/column addressing of the matrix elements would result in no coalesced access. Hence instead of addressing by row (or column) the matrix elements, we address them by diagonal. Let M_W the width of the matrix and M_H its height. Coalesced addressing is most easy in the parallelogram matrix, as we could pretend that the parallelogram is simply a tilted rectangle (with diagonal elements being contiguous in memory) as follows:

$$(i, j) \rightarrow i + (j - i) * M_H$$

Noticing that the triangle is just half the parallelogram, we could use the same addressing, but we would use twice the necessary memory. Hence we need a more involved formula: the size of the triangle embedded in a square of side k (incl. diagonal) is $\frac{k \cdot (k+1)}{2}$; knowing that fact, the index of the element (i, j) of the triangle (starting with the main diagonal when $i = j$), we obtain the following formula (with $M_H = M_W = n$):

$$(i, j) \rightarrow M - T + i \text{ with } \begin{cases} M &= \dfrac{(n \cdot (n+1)}{2} & \textit{total } \triangle \\ d &= n + 1 + i - j & \textit{diagonal of } (i, j) \\ T &= \dfrac{(d \cdot (d+1)}{2} & \triangle \textit{ of current diagonal} \end{cases}$$

Finally, for the rectangular matrix, since the parallelogram indexing looks efficient, we want to reuse the same idea. However, embedding the rectangle within a parallelogram would have a very large overhead ($(M_H)^2$, by adding a triangle on each side of the rectangle). The solution consists into breaking the rectangle into multiple horizontal stripes of fixed height B_H, hereby dramatically reducing the size of the additional triangles. Finally, the stripes can be stitched together to form a single parallelogram continuing along the next stripe. Noticing that beyond a certain number of lines, coalescing access does not improve latency as memory is anyway not stored contiguously, we can make B_H constant .

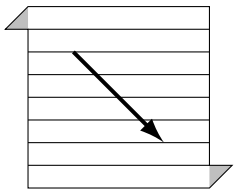


Figure 10: Parallelogram memory representation of a rectangular matrix

It follows that the total memory required to store the matrix is:

$$M_W \cdot \left\lceil \frac{M_H}{B_H} \right\rceil \cdot B_H + (B_H)^2$$

and the mapping of indices is given by:

$$(i,j) \rightarrow (B_H \cdot \underbrace{(j+m)}_{\text{diagonal}} + m + \underbrace{\left\lfloor \frac{i}{B_H} \right\rfloor}_{\text{stripe}} \cdot \underbrace{M_W \cdot B_H}_{\text{stripe size}}) \quad \text{with } m = i \mod B_H$$

3.8 Compilation stack

Since the generated code sensibly differs from the Scala version, due to the reasons previously discussed, we cannot reuse LMS, although we borrow some of its ideas. Having the full control on the compilation stack also provides us the following benefits:

- Since CUDA target and runtime compilation/execution is not supported in LMS, but in Delite [**?**]: an additional framework that runs on top of LMS. Although conceptually very similar, implementing our own stack reduces the number of dependencies, hence possible sources of misconfiguration errors for the final user of our framework. Additionally, since we do not share much of the functionality of Delite, an ad-hoc stack helps keeping the project featherweight.
- LMS can generate code for monadic functions that operate on arrays. However dynamic programming problems might require multiple inputs (for multi-track grammars) and special scheduling (to respect dependencies in the matrix), hence we need to issue specific C/CUDA code to handle problems correctly.
- LMS works on array of primitive types, possibly array of structures broken into array of simple types. Since in DP problems, composite types are representing a single logical element, and since we want to benefit from coalesced accesses (and possibly storing structures into efficient on-chip shared memory), we do not want to break structures. Also we want to offer support for tuples, which are a convenient way to write data containers (also we want to support case classes and composites types).
- Some information is only known at run-time (for instance input and matrix dimensions), hence we want to benefit from this knowledge as it helps computing the matrix indices more efficiently. Since such information could possibly be reused, we want to make the process as transparent as possible for the DSL user.

Finally, we reap most of the LMS benefits in the generation of user-defined functions, as they are completely independent of the rest of the generated program thus could be generated without additional processing.

Although our compilation stack might look quite similar to that of Delite, we do not share any component but LMS for the user functions generation. The compilation and execution process works as follows:

- Compilation: LMS generates the C code corresponding to the user-defined function and embeds them into the program bytecode (note that we present them separately for clarity).
- Runtime, for each parser (grammar+algebra): recurrences analysis is done in order to generate code (with placeholders for constants)
- Runtime, at every parse function call: the input size is known, hence replaced into the generic problem code, which is then processed by CUDA, C++ and Scala compilers. Then the JNI library resulting of the compilation is loaded and its corresponding Scala wrapper is invoked on the data to be processed.

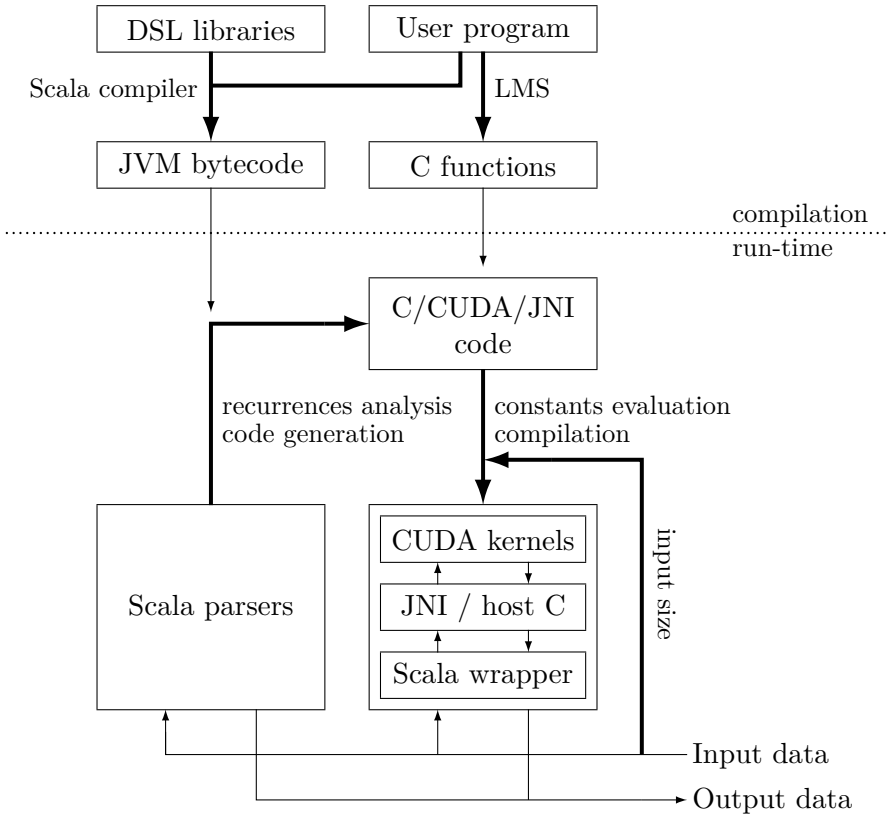


Figure 11: Compilation and execution scheme of a parser

4 Implementation

4.1 Ad-hoc CUDA

In the project planning, an ad-hoc implementation phase immediately followed the problem analysis (we also present the parallelogram matrix case). The goal of this phase is threefold:

1. Better understand the challenges in CUDA implementation of dynamic programming problems and get on par with state-of-art implementations.
2. Have a baseline implementation that is independent of the hardware and that could be benchmarked. We also tried to contact the authors of [**?**] and [**?**] to obtain their implementation. The former provided us with their implementation, which turned out to address large serial problems whereas our focus was on smaller non-serial problems, the latter did not respond to our solicitations.
3. Have an optimal implementation that can serve as target to be imitated and generalized by the code generation.

Leveraging the insights provided by [**?**] and [**?**], we started with a basic implementation (where each CUDA thread processes one matrix line) with three additional optimizations:

- Memory accesses must be coalesced (memory accesses account for a significant part of the total running time, according to both manufacturer documentation and experiments [**?**])
- Synchronization between threads can be done according to [**?**], additionally, we can slightly loosen the synchronization restrictions, as the paper describes a thread barrier whereas we only require a condition on previous thread progress (except for the parallelogram case, where we still require a barrier).
- Computation progresses element-wise along the diagonal (maximizes the parallelism level)
- Thread block size = warp size (32) to benefit from implicit synchronization within warps

4.1.1 Related work

Since [**?**] focuses on a different class of problem, we compare our implementation against [**?**], which provides an efficient matrix multiplication implementation. However, since we have neither the source code (or binary) nor the same evaluation hardware, we need to normalize the results. To do that, we present hardware differences and their result:

Graphic card		**Our**	**ATLP[?]**
Model		GeForce GT 650M	Tesla C1060
Architecture, capability		Kepler (3.0)	GT200 (1.3)
Memory	Mb	1024	4096
CUDA cores		384	240
Clock (core, memory)	MHz	756, 1953	1300, 1600
Memory bus	bit	128	512
Memory bandwidth	GB/s	28.8	102.4
Processing power	GFLOPS	564.5	622.08
Processing speedup		1	1.07
Memory speedup		1	3.55

Table 3: Graphic cards technical specifications (source: Wikipedia)

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

Table 4: ATLP[**?**] results: matrix chain multiplication, execution time (in seconds)

4.1.2 Results

We present here the timings of our implementation. For correctness, we first implemented a CPU version that we used to compare CUDA results against. Input data is made of random numbers. The implemented dynamic programming problems are:

- Rectangle: Smith-Waterman with arbitrary cost
- Triangle: matrix chain multiplication
- Parallelogram: polygon triangulation using a matrix larger than necessary. Note that this implementation uses at most 32 blocks to prevent dead locks on our hardware (restriction due to the number of concurrent threads on the device).

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

Table 5: Execution time (in seconds) for R=rectangle, T=triangle, P=parallelogram

4.1.3 Results discussion

- **User interface:** It has been put in evidence in [**?**] that using the GPU exclusively for CUDA or in combination with UI display (Mac OS) affects the performance (GeForce 330M). With the newer architecture, this difference has been reduced to less than 3.5%, decoupled UI and CUDA performing best. So we can safely ignore this issue.
- **Blocks synchronization:**
  - Removing `__threadfence()` before the synchronization is not syntactically correct but results still remains valid, this confirms the observation made by [**?**]. Speedup for matrix size of 1024 are 67ms (parallelogram) 100ms (triangle) 180ms (rectangle).
  - In the parallelogram case, using all threads to monitor other blocks status instead of the first one only results in a 6.4x speedup (22.72→3.52ms) for the parallelogram.
- **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 complex-

ity 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

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

  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 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.

- **Threads number:** 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:** 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 → 2.464), 14.1% (triangle, 1.427 → 1.225) and 9.7% (parallelogram 1.539 → 1.389). The best experimental number of unrolling is 5.

4.2 Scala parsers

The Scala parsers consist in 4 traits that are used to construct a DSL program:

- **Signature:** abstraction to define input (`Alphabet`) and output (`Answer`) types, and the aggregation function. The signature is implemented by all other traits (in particular algebras and grammars).
- **BaseParsers:** serves as basis for the two other traits and defines common features. It implements the `Parser` abstraction and all its inheriting classes: `Tabulate`, (abstract) `Terminal`, `Aggregate`, `Filter`, `Map`, `Or`, `Concat`. Terminals are further specialized in the two other traits (ADPParsers and TTParsers). The parser abstraction specifies 3 methods:
  - `apply(subword)` computes the parser result; it is used to obtain the corresponding results.

  - `unapply(subword,backtrack)` computes the previous step of the backtrack by returning subsequences at the origin of the result; it is invoked recursively to obtain the full backtrack trace.
  - `reapply(subword,backtrack)` is very similar to apply, except that it computes only the results matching the backtrack. It is used to construct the result corresponding to a backtrack trace (possibly in a different domain, pretty printing, ...).

  To support analysis, the parsers carry additional values:
  - Minimum and maximum yield size: functions evaluated recursively except for tabulations where value is attributed in the yield analysis phase.
  - Number of inner alternatives: helps counting alternatives, hereby guaranteeing an unique number for each (provided that parsers obtain non-overlapping ranges).
  - Number of inner moving concatenations: helps determining required storage for the backtrack as well as retrieving the appropriate index in the backtrack phase

  Additionally, the BaseParser implements the analysis that are shared by both the Scala and the CUDA version: dead rules elimination, yield analysis and dependencies ordering. Finally, it provides some implicit functions to flatten nested tuples (that are constructed by multiple concatenations).
- **ADPParsers:** used as basis for a single track DP grammar (using one input sequence). It defines the concatenation operator $\sim$ (`Concat` wrapper), and the terminals (empty, element and sequence). Additionally, it defines the interface functions `parse(input)`, `backtrack(input)` and `build(in,backtrack)` that respectively compute the result, the backtrack and the result corresponding to a trace.
- **TTParsers:** used to define two-track DP grammar (using a pair of sequences as input). Similarly, this class defines concatenations $-\!\sim$ and $\sim\!-$, terminals (for each track) and the `parse(in1,in2)`, `backtrack(in1,in2)` and `build(in1,in2,backtrack)` functions.

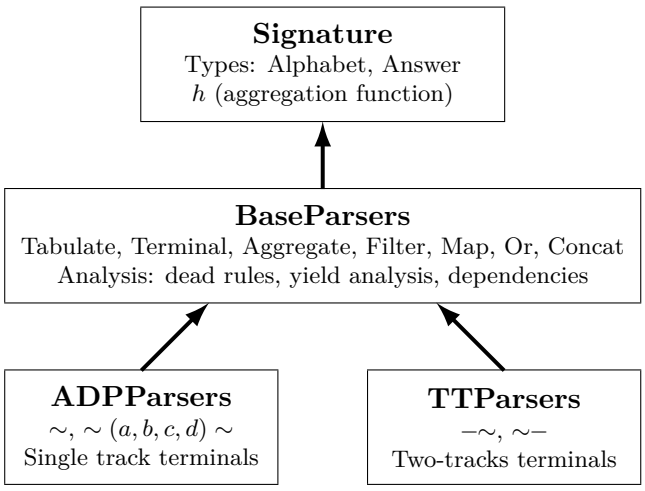


Figure 12: Scala parsers class diagram (simplified)

4.3 Code generation

The code generation step produces multiple outputs that are tightly bound to each other. Besides the Scala wrapper (a simple JNI interface), in the C/CUDA code generated we distinguish:

1. JNI input and output conversion functions
2. Host helpers for memory management and scheduling of CUDA kernels
3. CUDA matrix computation, which can be further decomposed into matrix scheduling (loops) and (matrix cell) computation.
4. CUDA backtrack collection kernel

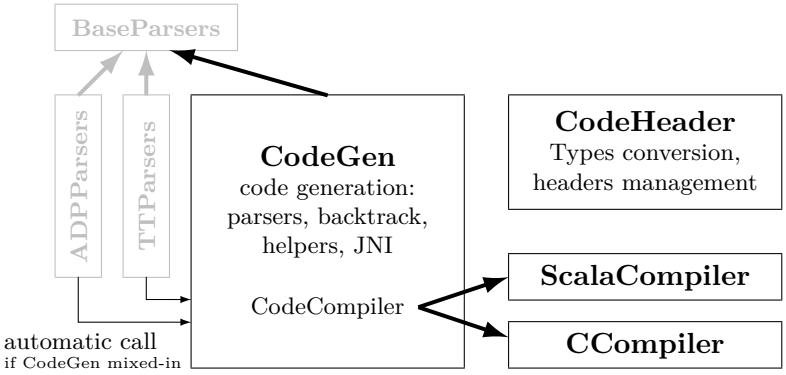


Figure 13: Code generation and runtime engine class diagram (simplified)

4.3.1 Scala structures conversion (JNI)

Since Scala general types can be extremely complex and might heavily depend of the JVM, we want to restrict the supported types; additionally types should be of fixed size for more efficient processing and easier memory allocation. We support the following types:

- **Primitive types:** natively supported in both Java and C. Since there is some little semantics difference between these two languages types, we used C (signed) types as reference. Supported types are: boolean, byte (unsigned char), char, short, int (32bit), long (64bit), float and double.
- **Empty case classes:** user-defined types might be more complex, so we allow users to define case classes that serve as data container and would be translated into C `structs`.
- **Tuples:** if the user-defined type is fairly simple, a named case class might be cumbersome. Tuples are a syntactical lightweight alternative to case classes, although they translate very similarly. Since Tuple classes are generic and can carry different member types; need to name tuple types uniquely, according to their arity and inner types.

Currently we use `Manifests` and reflection to extract types, and convert their string representation into our restricted subset. Manifests expands tuple inner types and reflection can be used to find class member's types. This imposes the additional restriction that we can not nest tuples into case classes, because generic types are then erased. However, the same effect could be achieved with Scala 2.10 `TypeTags` although we then need to rely on macros expansion to convert them properly into concrete classes. [see hint from Eugene, `https://gist.github.com/4407488`].

The JNI functions are involved at input to decode sequences arrays and at output, to encode the result and possibly its corresponding trace. Input method is constructed in two steps:

- Recursively obtain the classes and accessor methods of the composite input type. A subtle variation is that case classes primitive types are immediately converted into native types whereas tuple members are boxed in their respective class (i.e. `java.lang.Integer`, ...).
- For each element of the input array, retrieve the objects recursively and write their primitive values in the corresponding `struct` array.

The output method consist of two different steps:

- Converting the result into its JVM counterpart by using the opposite rule as for decoding input (but with JNI types specified in the constructor lookup instead of accessors).
- Optionally encoding the backtrack: this is pretty straightforward as the structure is more regular (and make uses of Lists); additional care should be taken to avoid bloating concatenation indices lists with unnecessary elements (as C uses fixed memory whereas Scala lists length might vary).

4.3.2 Host wrappers

The host wrappers are the functions bridging between JNI and CUDA; their duties are:

- Exposing JNI parsing and backtracking functions
- Calling appropriate conversion methods
- Allocating host and CUDA memory (and managing transfers between them)
- Launching CUDA kernels: matrix computation, backtrack, and possibly aggregation within window (additional aggregation among window results, would this option be set)

One peculiarity of our execution environment, is that the kernel execution duration is bound to approximately 10 seconds[17]. To solve this issue, we estimate the overall complexity of matrix computation, which allows us to estimate running time, then break computation into multiple kernels sufficiently small to fit in the time limit.

Since computations are made diagonal-by-diagonal (see 4.3.3), we can easily decompose the matrix computation by adapting the number of diagonals computed per kernel. The global complexity being the product of the number of elements and the complexity per element, the latter being equal to the number of unbounded concatenations (where maximal size is infinite).

Bonus: problems larger than device memory

Problems larger than the device memory can actually be processed on recent CUDA devices (architecture ≥ 2.0) as these are able to address the main memory from the device. However, since the distance between CUDA processors and memory is increased, there is a $5\times$ slowdown penalty to be paid in this configuration (experimentally, on a 1024×1024 triangular matrix). Nevertheless, this workaround implementation has 2 benefits:

- It allows larger problem to be solved, with very little implementation effort, would the user be patient enough for the computation to terminate
- It provides a good estimation of the main memory usage penalty, and hereby a strong argument in favor of the implementation described in 3.6.3 (with only up to $\leq 0.3\%$

[17] Hard limit imposed by the operating system. Although workarounds exist for Linux and Windows (requiring a second graphic card to display the UI), none of them is compatible with Mac OS. Eventually, a hack has been devised to force the UI on CPU while keeping the dedicated CUDA card powered; unfortunately this does not alleviate the kernel execution timeout.

transfer overhead). However, since we have not found concrete applications with such matrix size, the benefit of supporting large matrices is unclear, hence we leave the optimal implementation for future work.

4.3.3 Matrix computation scheduling

Similarly as in the ad-hoc implementation, progress is made along the diagonal (see 3.7) and each thread is responsible of one line. That is, the matrix is swept horizontally by a «diagonal of threads», that are enabled only if they are within a valid matrix cell.

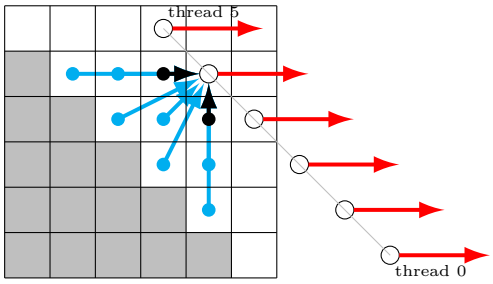


Figure 14: «Diagonal of threads» and maximal dependencies

Special care must be taken to handle computation dependencies: within a warp, all threads are executed at the same time, hence no synchronization is necessary. To benefit from this implicit synchronization, we set block size being equal to wrap size. It remains to provide inter-block synchronization: dependencies are along line, column and possibly intermediate elements. By induction on rows and columns, it suffice to have the last column and row element valid. Since line is computed by the current thread (hereby valid), it only remains to guarantee that the column element of the previous line is valid (in figure 14, previous refers to the line immediately below). To do that, each block writes last valid diagonal in a «lock» array, and next block need only to wait (polling) until desired element is marked valid. Notice that `__threadfence` is not mandatory (hereby slightly improving performance), verifying the observation of [**?**].

```
__global__ void gpu_solve(/*...*/ volatile unsigned* lock, // = {0}
    unsigned d_start, unsigned d_stop) {
  const unsigned tB = blockIdx.x;
  unsigned tP=d_start; // block progress

  for (unsigned diag=d_start; diag<d_stop; ++diag) {
    /* ... compute diagonal values ... */

    // __threadfence();
    if (threadIdx.x == 0) {
      lock[tB] = ++tP;
      if (tB > 0) while(lock[tB-1]<tP) {}
    }
    __syncthreads();
  }
}
```

Listing 2: Synchronization with previous thread block (active waiting)

4.3.4 Parsers code generation

Parsers generation is independent of user-defined function generation (see 4.3.6). Tabulation inner parsers are first wrapped in additional aggregation (by h, hereby ensuring they produce at most one result) and normalized (according to 3.5); code generation then occurs recursively, producing a list of loops and conditions, and body (possibly with a hoisted part). Additionally, position variables are maintained and subrule index and concatenation indices are propagated. We give an overview of each parser transformation:

- **Terminal:** provides its own C code, which correspond usually to the input element value, its position or the position of the matching range.
- **Tabulate:** is a simple value load, possibly wrapped into a validity check. Useless validity verification can be removed by marking the tabulation as «always valid».
- **Aggregate:** corresponds to an intermediate (value,backtrack) pair where inner parsers write their result; outermost aggregation is written back to corresponding (cost, backtrack) matrices. Validity information, and concatenation indices are propagated within backtrack. To preserve a correct semantic, inner aggregations body is hoisted outside loops and condition checks of the enclosing parser.
- **Or:** since parsers are normalized and operate on a single aggregation result, it suffice to emit sequentially code of alternatives.
- **Map:** wraps its argument into a the user-defined function call
- **Filter:** wraps its body into user-defined condition check
- **Concat:** fixed size concatenation are wrapped in simple conditions; moving concatenations are wrapped in a `for` loop. The loops and conditions are further simplified to reduce range and remove useless conditions before actual code is emitted.

Intermediate types must be correctly declared. To do that each user-defined function provides its input and output types. Aggregation temporary values declaration is ensured by a *exists-or-declare* header policy that is called for every type declaration.

4.3.5 Backtracking on the GPU

The backtracking is processed similarly as with the Scala parser, the major difference being that since we are generating the C code, we can provide an immediate mapping from the subrule index to the backtrack elements to add to the trace. The backtrack is done in 3 steps:

- If window is set, the windowing aggregation kernel is run to determine the position of the best result within the matrix. Otherwise the best position can be found at the last computed element of the matrix.
- For a $m \times n$ matrix, allocate a $m + n$ vector with two heads (reading, writing, initialized at the same position). Write the best element in the vector.
- While there is a vector element that has been written but not read
  - From the parser id and its position retrieve the corresponding (subrule, concatenation indices) pair by reading in the corresponding matrix cell
  - Using this, write new backtrack items that are at the origin of the current element.

Since code is generated, it is possible to write the last step using a switch case, hereby flattening the writes in the vector (compared to recursive calls in Scala). Finally, since the trace has to be reversed, we can obtain this transformation for free by constructing the trace list from the end in the JNI conversion. Reversing the list present the advantage that trace is immediately usable to construct the desired element. It might be possible that Scala and CUDA parser provide

different traces to construct the same result, because the trace verifies the dependency order, which is only a partial order.

4.3.6 User functions generation

XXX: How LMS is used to generate function
Possibly write that we can do the same with macros
Why cannot use LMS for everything
Where and how is LMS useful
Also user function code generation with macros ?

4.4 Runtime execution engine

The runtime execution engine is made of two instrumented compilers:

- A wrapper for `g++` and `nvcc` that can combine different file types (`.h`, `.c`, `.cu`) into a JNI library which is then loaded into the current JVM instance. If necessary, paths can be customized to fit the user environment.
- A wrapper for the Scala compiler, which allow the creation of Scala interface to the freshly compiled JNI libraries. It should be noted there that using `VirtualDirectory` as compilation target prevents the interaction with JNI, hence physical path has to be used.

These two compilers interfaces are then mixed in another class that transform the previously (see 4.3) generated code, fixing input sizes and splits (number of kernels to launch to respect the time limit) constants, and execute it.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
CONTINUE HERE
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

4.5 LibRNA (?)

XXX

5 Benchmarks

XXX: explain why slower: approx 3x more memory used per score (maintain matrix dimensions)

XXX: continue benchmarking here

Matrix size	1024	2048	4096	8192
Splits	1	8	64	512
Analysis	0.007	0.007	0.007	0.007
Code generation	0.071	0.072	0.073	0.071
C/CUDA compilation	3.025	1.836	1.832	1.841
Scala compilation	1.837	1.853	1.830	1.767
- JNI read	0.027	0.028	0.029	0.058
- CUDA compute	0.868	3.378	19.782	139.697
- CUDA backtrack	0.009	0.012	0.023	0.045
- JNI output	0.005	0.009	0.018	0.032
Total execution	0.915	3.432	19.860	139.846

Table 6: Preliminary results for MatrixMultGen

XXX: Compare current implementation versus ad-hoc implementation. Compare CUDA vs Scala (we might need to ad-hoc fix stack overflows in Scala).If Zuker coefficients can be fixed, compare performance with [**?**] by rescaling numbers wrt to bandwidth and computation performance.

6 Future work

We consider several directions and possible extensions for our work. We briefly describe each of them and give an idea of how they could be implemented:

1. **Non-serial scheduling for problems larger than the device memory:** as described in 3.6.3, it could be possible to handle problems that are larger than the device memory in an efficient way, hereby dramatically reducing the memory transfer penalties introduced by the main memory usage. However, this comes at the price of a more involved kernel scheduling and a complex element indexing strategy (since we first need to find the enclosing matrix block before addressing the element relatively to it).
2. **Algorithmic analysis:** so far, we considered that the DSL user would write an optimal program. Another direction in which we could improve the parsers is the recurrence analysis, either by removing serial dependencies when possible (see 2.1.3) or reducing the algorithmic complexity by creating intermediate tabulations (see 3.1). These analysis would certainly involve a strong mathematical analysis and the ratio benefit over implementation complexity would be quite small under the initial assumption that the DSL users are experts in their field (hereby knowing how to optimize manually the grammar).
3. **Serial problems larger than memory:** As discussed in 3.6.4, this class of problems require a completely different implementation. Since the authors of [**?**] are planning to write extensions to their implementation, duplicating the effort might not be worth the price; however, would their future implementation be sufficiently modular, we could integrate it in our framework and redirect compatible grammars to this state of art implementation.
4. **Add FPGA as target platform:** Initially envisioned a second target platform by George Nithin, a PhD student of the LAP (Laboratory of processors architecture), the underlying complexity of transforming DP recurrences into VHDL code made us leave this platform

aside for the scope of this project. The reconfigurability possibilities of FPGA make them attractive whenever it comes to very simple and massively parallel computations where the memory can be pipelined; this makes serial dynamic programming problems good candidates for such implementation.

5. **Multi-dimensional matrices:** in the current implementation, all the matrices are encoded such that they are of the same size. Leveraging the yield analysis, we could reduce the dimension of matrices that are of smaller dimension (for tabulations with bounded maximal size). In the problems we have analyzed, no such special case appeared, this is why we do not support this optimization at present.

   ```
   How to Encode multi-dimensional matrices efficiently
   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
   ```

6. **Independent computations:** The attentive reader might have noticed a discrepancy between the problem analysis and the recurrence rules of the Zuker problem implementation. This is due to the usage of multiple source for the problem description.
7. Pack the data => less memory transfer (i.e. GATC=>4 letters in 1 char), but only applies to input; score is hardly compressible if appropriate type is initially chosen
8. Operate on larger words (ex 64 bits) to increase thread locality and reduce memory accesses

7 Conclusion

XXX

Planning

Todo @TCK

- Test/proof parsers are correct – make sure implementation is correct
- Automate test to compare against implementation
- Benchmarks – use CUDA profiler(?)
- Write report
- Port LibRNA for CUDA?

Todo @Manohar

- Integrate LMS code generation into v4.
- Fix Zuker coefficients

Legacy roadmap deadlines:

Nov 16	Rules normalization and automatic backtracking GenScala on LMS + GenCuda + LMS CudaCompiler
Nov 23	Problem generalization: "cyclic keyword", Zucker problem / CudaLoop optimization
Nov 30	— gap due to LMS missing knowledge —
Dec 7	Benchmarking, grammar analysis
Dec 14	First thoughts for larger than device memory
Dec 21	Writing report
Jan 4	— holiday —
Jan 18	Writing report: implementation description and plan for future work

Journal

Sep. 17	Getting started with the project, reading related work on hash maps.
Sep. 24	Parallel hashing paper solve the problem for 32bit key/value pair. Stripped CudPP. Devised (but not implemented) extension beyond 64bit using memory areas locks.
—	Change of project suggested by Vojin (supervisor): joint work with Manohar on DP
Oct. 01	Problems specifications: serial/non-serial, started CUDA implementation with blocks.
Oct. 08	CudAlign solves serial monadic, might adapt it (but complicated / ad-hoc / in progress). Focus on non-serial problems that fit in GPU memory.
Oct. 15	First working implementation for non-serial problems (rectangle, triangle, parallelogram)
Oct. 22	No workaround for timeout, fixed by multiple kernels. Implemented backtrack on GPU.
Oct. 29	Scala/C compiler engine, to use Scala/CUDA, C code must be provided.
Nov. 05	Multiple fixes and rework of ADP parsers to aim at generating C-like code.
Nov. 12	Rework ADP parsers: cyclic, two-track grammars and automatic backtracking discussion.
Nov. 19	Explorations LMS and Macros for C code, implementation is quite different, hence ad-hoc.
Nov. 26	Full backtracking stack: apply / unapply / reapply, refactoring of the classes.
Dec. 03	JNI for LibRNA to get coefficients for Zuker, errors in algebra (quite hairy code).
Dec. 10	Rework concatenation operators, yield analysis, code generation.
Dec. 17	Detupling, generic backtrack (vs. ad-hoc), nested aggregates, empty results support.
Dec. 24	(sick 4 days), code generation: full JNI conversion support, rework CodeCompiler.
Dec. 31	Writing report (design, implementation), fixing various issues, complexity analysis.
Jan. 7	XXX
Jan. 14	XXX