DP Problems of Interest

Manohar Jonnalagedda, Thierry Coppey, Nithin George

1 Introduction

1.1 Definitions

- Block of computation: a block is simply a part of the DP matrix that we want to compute.
- Wavefront: this is the place around which computation happen, typically. There should be some memory to store intermediate information between block of computations

1.2 Problems classification

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

Monadic/polyadic

- **Monadic:** on the right hand-side of the recurrence formula, only one term appears. For instance, Smith-Waterman with constant penalty is monadic

$$M_{(i,j)} = \max \begin{cases} 0 \\ M_{(i-1,j-1)} + \text{cost}(S(i), T(j)) \\ M_{(i-1,j)} - d \\ M_{(i,j-1)} - d \end{cases}$$

- **Polyadic:** when multiple terms of the recurrence occur in the right and-side of the recurrence formula. For instance Fibonacci is polyadic:

$$F(n) = F(n-1) + F(n-2)$$

Serial/non-serial

- **Serial:** when the solution depends only of a fixed number of immediately previous solutions (i.e. neighbor cells). For instance Fibonacci is serial (it accesses only 2 cells backward).
- **Non-serial:** when the solution depends of an arbitrary number of previous solutions. Typically Smith-Waterman with arbitrary gap penalty and Nussinov are non-serial:

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

1.3 Simplifications

1.3.1 Calculus

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

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

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

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

This simplification avoids the non-serial dependencies at the cost of extra storage in the wavefront, unfortunately, it might be applicable only for some special cases.

1.3.2 Precomputations

When a calculus transformation is impossible, it might be worth to interleave a computation phase that will aggregate some of the results that are necessary to the computation block. For instance, for Nussinov term $\max_{i<k<j}[M_{(i,k)} + M_{(k+1,j)}]$, we can precompute it over all rows and columns of the block, and for all elements that are not part of the block, then pass these partial results together at the block launch to finish the computation.

On GPU, this could be done by interleaving a new kernel for this specific purpose, on FPGA, this could be done by preparing the maximums in another memory area whose pointer will later be passed to the co-processor. We may notice that since this phase is at the same time necessary for both architecture, and can be run independently, we can both execute them concurrently and mix between architectures (use CUDA for pre-computation and FPGA for actual tile computation for instance).

2 Problems of interest

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

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

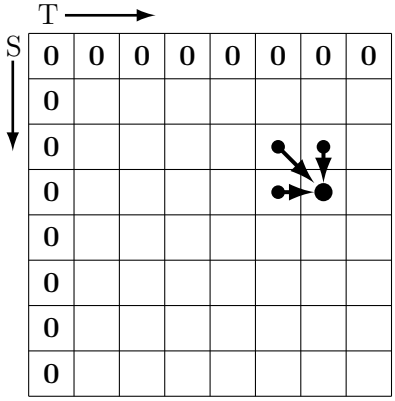
We describe the problem processing in terms of both initialization and recurrences.

2.1 Smith-Waterman (simple)

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

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

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



8. Optimizations:
   - In serial (monadic) problems we can avoid building the matrix M by only maintaining the 3 last diagonals in memory (one for the diagonal element, one for horizontal/vertical, and one being built). This construction extends easily to polyadic problems where we need to maintain $k + 2$ diagonals in memory where k is the maximum backward lookup.

2.2 Smith-Waterman (with gap extension at different cost)

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

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

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

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

That can be written alternatively as:

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

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

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

Notice that this recurrence is very similar to Smith-Waterman (simple) except that we propagate 3 values (M, E, F) instead of a single one (M).

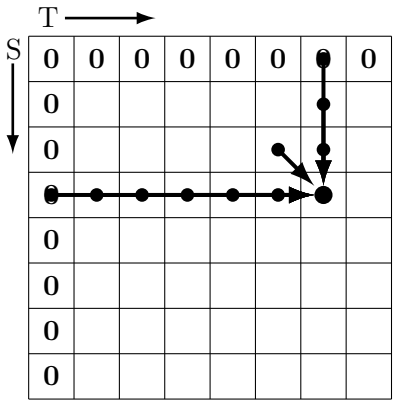
6. Backtracking: same as Smith-Waterman (simple)
7. Visualisation: same as Smith-Waterman (simple)
8. Optimizations: same as Smith-Waterman (simple)

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 $m \geq n$[1]. Example penalty function could be[2] $g(x) = m - x$.
2. Matrices: $M_{m \times n}, B_{m \times n \times m}$
3. Alphabets:
   - Input: $\Sigma(S) = \Sigma(T) = \{a, c, g, t\}$.
   - Cost matrix: $\Sigma(M) = [0..z], z = \max(\text{cost}(\_)) \cdot \min(m, n)$
   - Backtrack matrix: $\Sigma(B) = \{stop, NW, N_{\{0..m\}}, W_{\{0..n\}}\}$
4. Initialization:
   - Match cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
   - Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = stop$.
5. Recurrence:

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

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



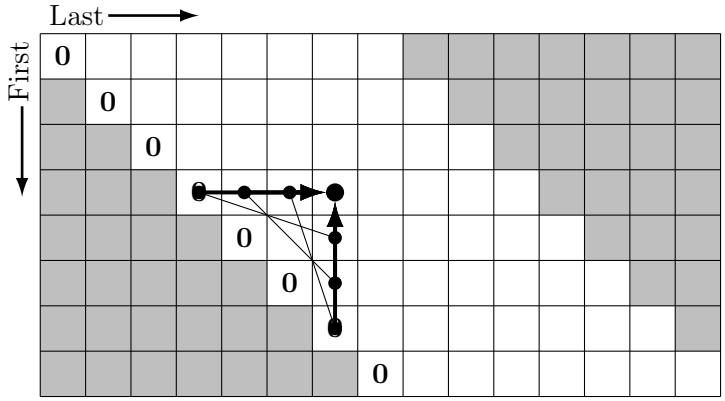
8. Optimizations: The dependencies here are non-serial, there is no optimization that we can apply out of the box here.

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

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

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 lower triangular matrix compacted in memory (rows are contiguous) with a 0 diagonal that is omitted [3], hence $|S| = \frac{n^2}{2} = N$.
2. Matrices: $M_{n\times 2n}, B_{n\times 2n}$ *«first edge, last edge»* upper triangular including main diagonal
3. Alphabets:
   - Input: $\Sigma(S_{(i,j)}) = \{0..m\}$ with $m = \max_S(i, j)\forall i, j$ determined at runtime[4].
   - Cost matrix: $\Sigma(M) = \{0..z\}$ with $z = m \cdot (n - 2)$ (we add at most $n - 2$ edges).
   - Backtrack matrix: $\Sigma(B) = \{stop, 0..n\}$ (the index of the edge we add)
4. Initialization: $M_{(i,i)} = 0, M_{(i,i+1)} = 0, B_{(i,i)} = stop \quad \forall i$
5. Recurrence:
$$M_{(i,j)} = \left\{ \max_{i<k<j} M_{(i,k)} + M_{(k+1,j)} + S(i, k) \;\middle|\; k \right\} = B_{(i,j)}$$
   It is interesting to note that even in the sequential world, this problem is solved by filling the diagonals, ie. computing sub-solutions for all polygons of size k before those of size $k + 1$.
6. Backtracking: Start at $B_{(1,n)}$. Use the following recursive function for the smaller polygons:
$$\mathrm{BT}(B_{(i,j)} = k) \mapsto \begin{cases} A_i & \text{if } k = 0 \vee k = j \\ \Big(\mathrm{BT}(B_{(i,k)})\Big) \cdot \Big(\mathrm{BT}(B_{(k+1,j)})\Big) & \text{otherwise} \end{cases}$$
7. Visualisation: the layout is the a matrix of size $n \times (2n - 2)$, because of polygons being "cyclical" in nature.



8. Optimizations: we need to rotate that matrix to progress in the same direction as usual, that is towards bottom right.

[3] Arbitrary convention for both architectural implementation and code generator. Rationale: in lower triangular matrix, element address is independent of the matrix size.

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

2.5 Matrix chain multiplication

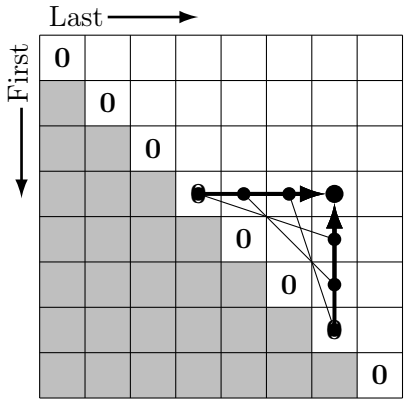
1. Problem: find an optimal parenthesizing of the multiplication of n matrices A_i. Each matrix A_i is of dimension $r_i \times c_i$ and $c_i = r_{i+1} \forall i$. *«r=rows, c=columns»*
2. Matrices: $M_{n\times n}, B_{n\times n}$ *(first, last matrix)*
3. Alphabets:
   - Input: matrix A_i size is defined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

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

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

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

7. Visualisation:



8. Optimizations:
   - We need to swap vertically the matrix to have a normalized progression towards bottom right. To do that, we need to map all indices $i \mapsto n-1-i$, but since we want to store the matrix sparsely, we might want to transform it into a lower triangular matrix as we can provide a size-independent mapping of element indices. XXX: what's the best trade off ? progress towards bottom right VS map indices more efficiently ?

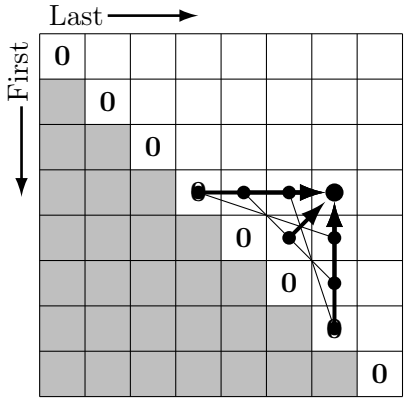
[5] Integer multiplication might blow up, use float or doubles and addition of logarithms instead.

2.6 Nussinov algorithm

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

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

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



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.

2.7 Zuker folding

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

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

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

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

$$VBI(i,j) = \min_{i<i'<j'<j}\{V_{(i',j')} + ebi(i,j,i',j')\} + c \quad | \quad (i',j') = BV_{(i,j)}$$

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 W,V, VBI and F.

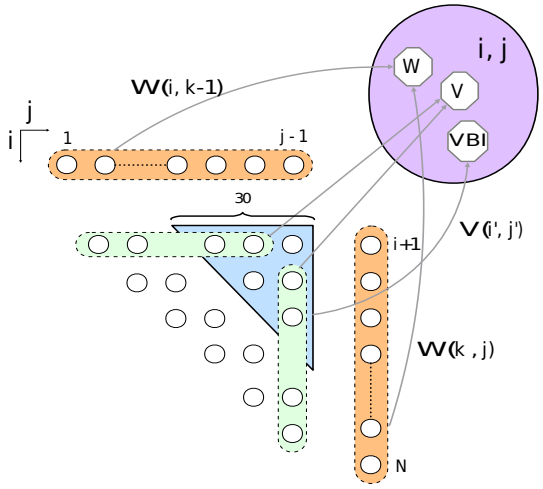
6. Backtracking: Start at $BF_{(n)}$ using the recurrences

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

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

$$BW_{(i,j)} = \begin{cases} S & \Longrightarrow & \langle bulge(i) \rangle BW_{(i+1,j)} \\ W & \Longrightarrow & \langle bulge(j) \rangle BW_{(i,j+1)} \\ V & \Longrightarrow & BV_{(i,j)} \\ k & \Longrightarrow & BW_{(i+1,k)} BW_{(k+1,j-1)} \end{cases}$$

7. Visualisation:



8. Optimizations: XXX

3 Plan

1. **Problems:** finish the problem description.
   We might look towards parallel tree-raking, but it does not share much with the above algorithms (sparse version of above computations, might not scale that well). Most of the common patterns are already enclosed by the above problems. Real input size is around 300'000 (we may want to target the million, using disk). We might want to also look at an $O(n^3)$-space complexity problem (like matching 3 strings S, T, U).
2. **User facing language:** goals are flexibility and compactness.
   - Design the user-facing language that we want to support, which should be similar to related papers «Algebraic Dynamic Programming» and «ADP fusion». We want to reuse their transformation to map (problem description) $\mapsto$ (kernel implementation) for a single element.
   - We also may want to try to make implicit transformation for code like `@DP def Fib(n:Int) = if (n<=2) return 1 else Fib(n-1)+Fib(n-2).`
   - Windowing: the user should be able to force a windowing (i.e. force a non-serial problem to be a k-polyadic serial problem).
   - Consider 3 different cases: we care about backtrack, the costs or both.
   - Backtracking: we might want to create an operator/class that given an item produces the previous in the backtrack, or the whole sequence in correct order, or only indices.

   $\Longrightarrow$ *end of October.*
3. **Prototyping:** get a prototype to understand difficulties and share common base.
   Implement a working prototype of Smith-Waterman with arbitrary gap cost on CPU (for correctness), and specific platform (CUDA/FPGA). This will give us an idea of how to implement the general case. We also need to benchmark and compare both implementations to see how we compare to existing implementations and see the direction to take (which decide is faster and by how much). Here we aim to do as good an implementaion for the specific platform (CPU/GPA) as possible.
   $\Longrightarrow$ *end of October.*
4. **Baseline:** Also use benchmarks provided by existing implementations as baselines.
   $\Longrightarrow$ *end of October.*
5. **Formalize IR:** According to experiences, describe the intermediate representation, also formalize the framework that will be provided to the code generators (i.e. memory management, ...).
6. **Full compiler stack:** enrich the compiler stack from both top-down (translate best user-facing language parsers) and bottom-up (parametric code generators), core of the work.
7. **Test and benchmark:** make sure our implementations are correct and compare them with previous papers implementations.

3.1 LMS compiler stack

User-language: define additional parameters for the recurrence

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

⇓ Conversion (using an existing technique)

Intermediate representation

⇓ Optimizations

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

Code specification

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

⇓ Generation

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

Binary program