Draft

Manohar Jonnalagedda, Thierry Coppey, Nithin George

Contents

1 Planning/draft

Contributions

- Identify DP classes and give a generalization of their shape/dependency graph
- Provide state of the art/on par parallel implementations for all these classes
- Generalize/simplify expression by using a DSL to implement these problems
- Provide multi-platform support (CPU/GPU/FPGA)

Todo @TCK

```
- continue performance evaluation
- class problem into categories
- think of hashmap implementation. is it useful ?
- CPU+GPU implementation for serial (NS?) problems larger than device memory
- optimizations for in-memory computations
```

Todo @Manohar

```
- generate chain matrix recurrences to generate implementation
- more theory behind what we want to support, how we encompass all cases, ...
```

Plan

1. **Problems description:** parallel tree-raking, does not share much with other algorithms (sparse version of computations, might not scale efficiently). Most common patterns are already enclosed by the above problems. Real input size is around 300K. We might want to also look at an $O(n^3)$-space-complex problem (like matching 3 strings S, T, U).
   All the problems we consider use 2D storage matrix, their dependencies are an union of:
   - Serial dependencies
   - Non-serial horizontal or vertical dependencies (1D non-serial)
   - Non-serial horizontal+vertical dependencies in the form $M_{(i,j)} = \mathrm{op}_k f(M_{(i,k)}, M_{(k,j)})$
   - We have not found other type of dependencies in the literature
2. **User facing language:** goals are flexibility and compactness.
   - User-facing language should be similar to related paper [?] or *ADP fusion*. We want to reuse the transformation mapping (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).
   - 3 different cases: we care about backtrack cost or both.
   - Backtracking: create an operator that produces the whole backtrack sequence 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 implementation 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:** describe the intermediate representation, formalize the framework 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. **Benchmark:** make sure implementations are correct, compare them other papers.
8. **Optimizations:** improve as much as possible / as long as time permits

2 Problems classification

2.1 Definitions

- **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).

2.2 Classification

2.2.1 Litterature classification

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

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

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

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

2.2.2 Calculus simplifications

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

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

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

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

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).

3 Problems of interest

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

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

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

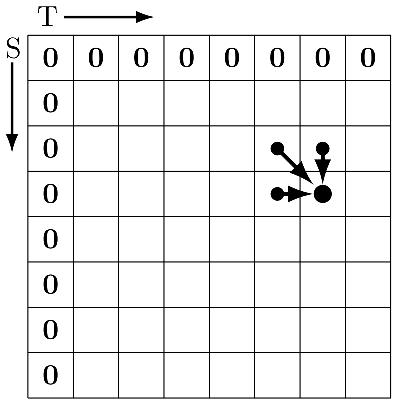
We describe the problem processing in terms of both initialization and recurrences.

3.1 Smith-Waterman (simple)

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

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

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



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

3.2 Smith-Waterman with affine gap extension cost

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

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

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

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

   That can be written alternatively as:

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

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

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

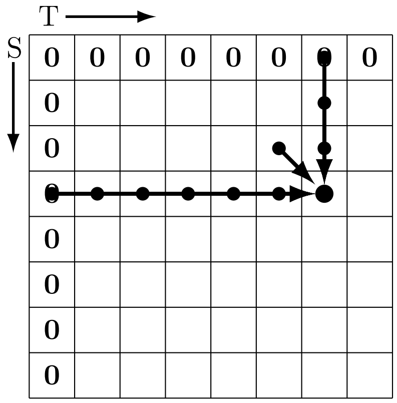
   Notice that this recurrence is very similar to Smith-Waterman (simple) except that we propagate 3 values (M, E, F) instead of a single one (M). Also notice that it is possible to propagate E and F inside a resp. horizontal and vertical wavefront.
6. Backtracking: same as Smith-Waterman (simple)
7. Visualization: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

3.3 Smith-Waterman with arbitrary gap cost

1. Problem: matching two strings S, T with $|S_{\text{padded}}| = m, |T_{\text{padded}}| = n$ with an arbitrary gap function $g(x) \geq 0$ where x is the size of the gap. Without loss of generality, let $m \geq n$[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: $\Sigma(B) = \{\mathit{stop}, NW, N_{\{0..m\}}, W_{\{0..n\}}\}$
4. Initialization:
   - Match cost matrix: $M_{(i,0)} = M_{(0,j)} = 0$.
   - Backtrack matrix: $B_{(i,0)} = B_{(0,j)} = \mathit{stop}$.
5. Recurrence:

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

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



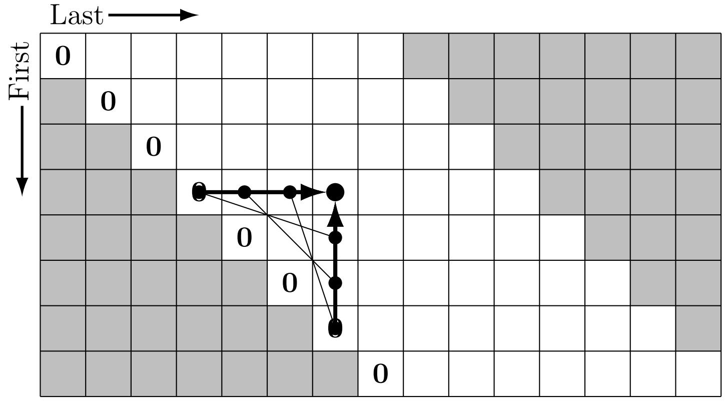
8. Optimizations: The dependencies here are non-serial, there is no optimization that we can apply out of the box here.

[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.

3.4 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: $\Sigma(B) = \{\mathit{stop}, 0..n\}$ (the index of the edge we add)
4. Initialization: $M_{(i,i)} = 0, M_{(i,i+1)} = 0, B_{(i,i)} = \mathit{stop} \quad \forall i$
5. Recurrence:
$$M_{(i,j)} = \left\{ S(i,j) + \max_{i<k<j} M_{(i,k)} + M_{(k,j)} \,\middle|\, k \right\} = B_{(i,j)}$$
   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. Visualization: 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

3.5 Matrix chain multiplication

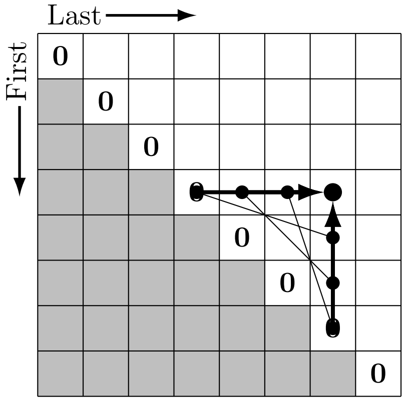
1. Problem: find an optimal parenthesizing of the multiplication of n matrices A_i. Each matrix A_i is of dimension $r_i \times c_i$ and $c_i = r_{i+1} \forall i$. *«r=rows, c=columns»*
2. Matrices: $M_{n\times n}, B_{n\times n}$ *(first, last matrix)*
3. Alphabets:
   - Input: matrix A_i size is defined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

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

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

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

7. Visualization:



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

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

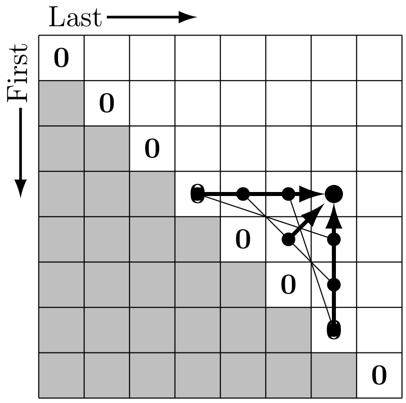
   With the initialization at $M_{(i,n-i-1)}$

3.6 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:

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

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



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

3.7 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:

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

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

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

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

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.

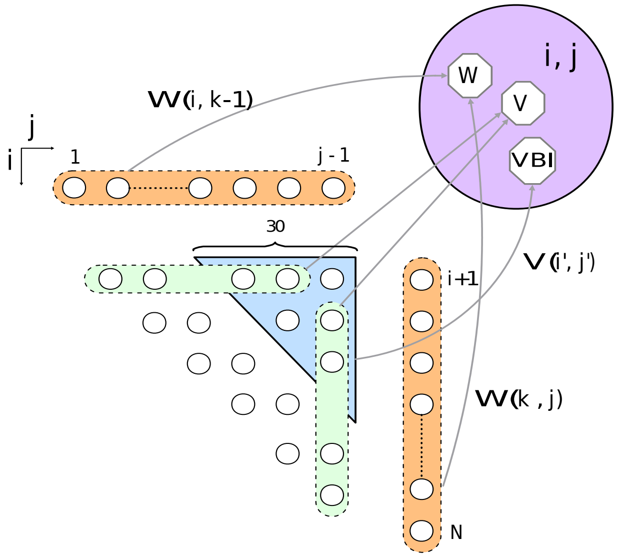
6. Backtracking: Start at $BF_{(n)}$ using the recurrences

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

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

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

7. Visualization:



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

4 Related problems

The goal of this section is to demonstrate that our framework can accommodate with many (20-50) problems that we have not considered at the design time.

4.1 Serial problems

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

4.2 Non-serial problems

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

4.3 Other problems

- Dijkstra shortest path: we need a $E \times V$ matrix, along E forall V reduce its distance, problem is serial along E, non-serial along V hence of complexity $O(|E| \cdot |V^2|)$ which is far worse than both $O(|V|^2)$ (min-priority queue) and $O(|E| + |V| \log |V|)$ (Fibonacci heap).
- Fibonacci: this problem is serial 1D. Could be implemented using a placeholder element in one of the matrix dimension.
- Tower of Hanoi: 1D non-serial
- Knuth's word wrapping: 1D non-serial
- Longest increasing subsequence: serial but binary search algorithm more efficient: $O(n \log n)$.
- Coin Change: 1D non-serial
- Floyd-Warshall: <it possible to move the k external loop inside?> Serial with n iterations
- Viterbi (hidden Markov models): T non-serial iterations over a vector
- Bellman-Ford (finding the shortest distance in a graph)
- Earley parser (a type of chart parser)
- Kadane maximum subarray 1D serial, look at Takaoka for 2D
- Structural alignment (MAMMOTH, SSAP), RNA structure prediction.
- Recursive least squares

- Bitonic tour

```
- Balanced 0-1 matrix
- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-hori
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth
- Transposition tables and refutation tables in computer chess
- Pseudo-polynomial time algorithms for the subset sum and knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- Duckworth-Lewis method for resolving the problem when games of cricket are interrupted
- The Value Iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Pho
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the traveling salesman problem, either exactly (in exponential time)
- Beat tracking in music information retrieval.
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision.
- Seam carving (content aware image resizing)
- Some approximate solution methods for the linear search problem.
=====> http://en.wikipedia.org/wiki/Dynamic_programming#A_type_of_balanced_0.E2.80.931_matrix

- Shortest path in DAGs
- Shortest path
- All pair shortest paths
- Independent sets in trees
=> also see exercises for more problems
=====> http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf
-
- Subset Sum, Coin Change, Family Graph
=====> http://www.algorithmist.com/index.php/Dynamic_Programming

- Optimal Binary Search Trees
=====> http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/05-dynprog.pdf

- Independent set on a tree
- 0-1 Knapsack
=====> http://www.cs.ucsb.edu/~suri/cs130b/NewDynProg.pdf
```

5 Implementation

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.

5.1 Memory requirements

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

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

5.2 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.

5.3 Large problems

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

5.3.1 Non-serial problems

Non-serial problems need to potentially access all elements that have been previously computed. We restrict[5] ourselves to the following dependencies:

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

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

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

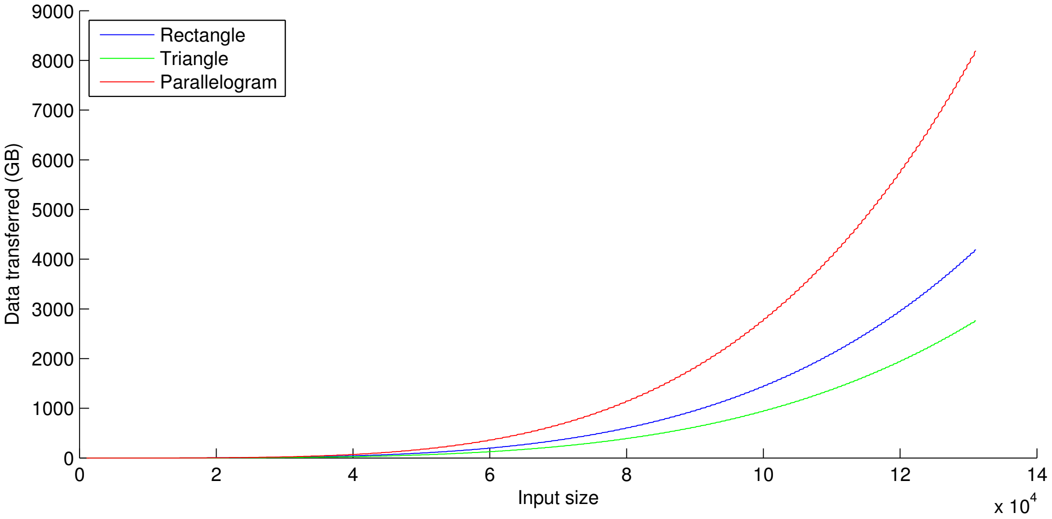
For simplification, let $m = n$ and assume that we have b square blocks per row and per column. Hence we have the following memory restriction:

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

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

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

Putting these two formula together, and using most of the device memory available, we obtain the following results with $S_C = 4, {S_B}^6 = 4, S_I = 1, S_W = 0$ and $M_D = 2^{30}$:



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

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

on input of size 128K would require respectively 7 days 13h$^{(\mathrm{R})}$, 2 days 22h$^{(\mathrm{T})}$ and 6 days 10h$^{(\mathrm{P})}$, assuming there is no other scalability issues. Although this overhead seems appealing compared to the computation time, the total time blows up (because of the $O(n^3)$ complexity) and make the processing of such large problem less relevant. Given that real problems – like RNA folding – operate at input sizes up to 4096, it would not be of much relevancy to implement a version for larger cases, although perfectly feasible.

5.3.2 Serial problems

The serial problem have the interesting property to access to a fixed number of previous elements. These elements

propose a memory consumption scheme

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

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

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

Split into blocks:

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

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

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

all the problem are subject to the input dimensions n.

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

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

5.4 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

6 Benchmarks

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

Graphic cards[7]

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

Results

ATLP[?]

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

Matrix multiplication timing in seconds

SWMB[?]

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

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

Smith-Waterman, timing in seconds

Intermediate results

Matrix	Block	Comment	R(s)	T(s)	P(s)
1024	1	CPU	1.965	1.191	6.069
2048	1	CPU	27.229	15.296	57.323
4096	1	CPU		177.608	
1024	32	GPU baseline	0.838	0.500	0.516
1024	32	GPU sync improved	0.642	0.316	0.343
2048	32	GPU P $\leq$ 32 blocks	2.864	1.427	2.096
4096	32	GPU 8 splits	21.902	9.285	16.767
8192	32	GPU 64 splits	159.058	62.064	135.793
12288	32	GPU 256 splits	419.030	196.971	460.912

Best timings for R=rectangle, T=triangle, P=parallelogram[8]

Micro-benchmarks [9]

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

- It has been put in evidence in a previous work[10] that with Mac OS operating system, using the GPU exclusively for CUDA or combining with UI display may affect the performance (GeForce 330M, architecture 1.2). With the new architecture (3.0, GeForce 650M), this difference has been reduced to less than 3.5% with the decoupling of UI and CUDA performing best. So in micro-benchmarks, we can safely ignore the graphic card usage.
- Synchronization between blocks
  - Removing `__threadfence()` before the synchronization is not syntactically correct but results still remains valid, this confirms the observation made by [?]. Speedup for <1024/32/*> are 67ms (parallelogram) 100ms (triangle) 180ms (rectangle).
  - In the parallelogram case, using all threads to monitor other blocks status instead of 1 thread results in a 6.4x speedup (22.72→3.52ms) on <1024/32/para>.

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

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

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