Dynamic Programming Alignment Language Specification

1. Introduction

This manual describes the programming language DPAL: dynamic programming alignment language, a simple C-like domain-specific language for dynamic programming sequence alignment algorithms. It is inspired by variations in the Smith-Waterman alignment algorithm, an algorithm less commonly used for DNA sequence alignment due to its slow run-time, but whose high sensitivity would provide immense benefits to the bioinformatics community if accelerated. The language is designed to efficiently describe such an algorithm, but also to be limited enough so that it can be automatically implemented by a compiler in a CPU-FPGA hybrid architecture.

2. Language Description

A DPAL program consists of five fields:

1. An alphabet for the sequence characters
2. Constant declarations (optional)
3. 2D dynamic programming matrix declarations
4. DP matrix cell score computation function
5. Alignment reporting condition function

DPAL is a strongly typed language with three different data types: `unsigned`, `signed`, and `bool`. The bitwidths of the first two, representing unsigned and signed integers respectively, can be specified by the user. For example, the following declares an 8-bit signed integer variable:

```
signed<8> var;
```

An unspecified bitwidth defaults to 32 bits. Adjusting the bitwidth could significantly improve the performance of the generated FPGA implementation, as the produced circuits could be significantly smaller, resulting in many more processing elements fitting in the FPGAs. `Bool` data take either `true` or `false` values, and since DPAL is strongly typed, they are not interoperable with integers.

2.1 Alphabet

The user specifies the alphabet for the sequences. This alphabet declaration is very similar to an enum in C, where the alphabet character identifiers are listed. For example, the alphabet for the four basic nucleotides would be declared as:

```
alphabet = {A, C, G, T};
```

Similarly, the alphabet of the twenty amino acids could be declared as:

```
alphabet = {ala, arg, asn, asp, cys, glu, gln,
            gly, his, ile, leu, lys, met, phe,
            pro, ser, thr, trp, tyr, val};
```

As with enums in C, each character in the alphabet has an implicit integer representation. In DPAL, this integer representation is unsigned, and the default representation of each character is its index in the list. Just as in C, the integer representations of the characters can also be explicitly specified. For example, the following is a more transparent way to specify the nucleotides in the standard .2bit format:

```
alphabet = {A=2, C=1, G=3, T=0};
```

2.2 Constants

Constants are declared after the alphabet, and can be used globally throughout the program. Constant values must be defined at declaration, and both scalar and matrix constants are supported. The following are examples of scalar and matrix constants, used for a fixed HOXD55 scoring scheme (see section 4.4).

```
const unsigned BITWIDTH = 18;
const unsigned<BITWIDTH> HOXD55[4][4] = {
    {91, -90, -25, -100},
    {-90, 100, -100, -25},
    {-25, -100, 100, -90},
    {-100, -25, -90, 91}
};
const unsigned<BITWIDTH> GAP_OPEN = -400;
const unsigned<BITWIDTH> GAP_EXTEND = -30;
```

2.3 Dynamic Programming Matrix Declarations

The core of dynamic programming alignment algorithms is the computation of 2D dynamic programming matrices. The user declares these matrices after the alphabet and constant declarations. A DP matrix declaration consists of the data type, matrix identifier, and double brackets `[][]`, as in:

```
signed<8> score[][];
```

DP matrix cell scores are assigned in the cell score computation function (section 2.5), and are referenced in the alignment reporting condition function (section 2.6).

Cell scores are initialized to 0 or `false`. (User-specifiable cell score initialization may be implemented in the future.)

2.4 Cell Score Computation Function

The computation of each cell score in the 2D dynamic programming matrices is performed in the `cell()` function. The indices of the current cell are accessed using the keywords `row` and `col`.

The arguments to the `cell()` function are run-time loadable parameters. These are typically used for parameterizable scoring schemes. At run-time, modification of these parameters is done through the generated software API, and are updated asynchronously from any alignments. Fixed implementations without run-time loadable parameters would likely achieve higher performance than those with the loadable parameters, as more FPGA resources would be allocated for processing elements.

Cell scores must be assigned from the result of a `max()` function call. This is because each cell score computation is treated as a decision between multiple options. For example, in the following Smith-Waterman case (see section 4.2), the computation of cell `H[row][col]` chooses between 0, a substitution, an insertion, and a deletion:

```
H[row][col] = max(0,
                  H[row-1][col-1] + substitute,
                  H[row-1][col] + insert,
                  H[row][col-1] + delete);
```

The arguments for the `max()` functions must abide by the following restrictions:

- Each argument can be dependent upon constants, parameters, and up to one other DP matrix cell.
- The DP matrix cell dependency must have indices less than or equal to the indices of the current cell.
- No two arguments can be dependent upon the same DP matrix cell.
- Circular dependencies are not allowed.

Temporary variables may be used, and the scope of a variable only includes the current iteration of the function. Variables must be assigned values before being referenced.

2.5 Alignment Reporting Condition Function

The `condition()` function represents the condition in which an alignment will be reported. Typically, this function is very simple – just a check for when a cell score exceeds a given threshold. Arguments to the `condition()` are run-time loadable parameters, such as the threshold value. These parameters are updated on every alignment.

Temporary variables may be used, and as with the `cell()` function, the scope only includes the current iteration of the function, and values must be assigned before referencing. A call to the `report()` function indicates the alignment is to be reported.

3. Formal Specification

Syntax specification key:

`x`	(in `courier` font) means `x` is a keyword terminal
`x`	(in *`italic courier`* font) means *`x`* is a non-keyword terminal
x	(in *italic*) means *x* is a nonterminal
(x**)**	(parentheses in **bold**) means zero or one occurrence of x (x is optional)
x+	means one or more occurrences of x
x*	means zero or more occurrences of x
[xyz**]**	(brackets in **bold**) groups together grammar symbols
|	separates production alternatives

Program	::=	*AlphabetDecl*
		*ConstDecl**
		DPMatrixDecl+
		CellFuncDecl
		CondFuncDecl
AlphabetDecl	::=	`alphabet={` *CharacterDecl* **[**`,` *CharacterDecl***]*** `};`
CharacterDecl	::=	*`id`***(**`=`*`int_const`***)**
ConstDecl	::=	*ConstScalarDecl* | *ConstMatrixDecl*
ConstScalarDecl	::=	`const` *Type* *`id`* `=` *Constant* `;`
ConstMatrixDecl	::=	`const` *Type* *`id`***[**`[]`**]**+ `= {`
		`{` *Constant* **[**`,` *Constant***]*** `}`
		[`,` `{` *Constant* **[**`,` *Constant***]*** `}`**]***
		`};`
Constant	::=	*ConstInt* | *ConstBool*
ConstInt	::=	*`int_const`* | *`id`*
ConstBool	::=	`true` | `false` | *`id`*
Type	::=	*TypeName* **(**<*ConstInt*>**)**
TypeName	::=	`unsigned` | `signed` | `bool`
DPMatrixDecl	::=	*Type* *`id`*`[][];`
CellFuncDecl	::=	`cell(`*Parameter* **[**`,` *Parameter***]***`)` `{` *Stmt+* `}`
CondFuncDecl	::=	`condition(`*Parameter* **[**`,` *Parameter***]***`)` `{` *Stmt+* `}`
Parameter	::=	*Type* *`id`* **[**`[`*ConstInt*`]`**]***
Stmt	::=	*VariableDecl*
	|	*IfStmt*
	|	*AssignStmt*
	|	*SwitchStmt*
	|	*ReportStmt*
VariableDecl	::=	*Type* *`id`* **(**=*Constant***)**`;`
IfStmt	::=	`if(`*Expr*`)` `{`*Stmt**`}`

```
                          [else if(Expr){Stmt*}]*
                          (else{Stmt*})
AssignStmt        ::=     id[[Expr]]* = Expr;
SwitchStmt        ::=     switch(Expr){CaseStmt+}
CaseStmt          ::=     case Expr:Stmt* | default: Stmt*
ReportStmt        ::=     report();
Expr              ::=     max(Expr[, Expr]*)
                  |       Expr + Expr
                  |       Expr – Expr
                  |       Expr < Expr
                  |       Expr <= Expr
                  |       Expr > Expr
                  |       Expr >= Expr
                  |       Expr == Expr
                  |       Expr != Expr
                  |       !Expr
                  |       Expr && Expr
                  |       Expr || Expr
                  |       Expr << Expr
                  |       Expr >> Expr
                  |       Expr & Expr
                  |       Expr ^ Expr
                  |       Expr | Expr
                  |       ~Expr
                  |       Constant
                  |       id[[Expr]]*
                  |       query_char
                  |       ref_char
                  |       row
                  |       col
```

4. Example Programs

The following are a series of example implementations in DPAL, including:

1. Simple Smith-Waterman
2. Run-time paramaterizable penalties with an ambiguous base character
3. Affine gap penalty
4. Fixed substitution matrix
5. Run-time parameterizable substitution matrix

DPAL keywords are bolded.

4.1 Simple Smith-Waterman

```
alphabet = {A, C, G, T};
signed<10> H[][];
```

```
cell() {
  signed<10> substitute = 2;
  if (query_char != ref_char) {
    substitute = -2;
  }
  H[row][col] = max(0,
                  H[row-1][col-1] + substitute,
                  H[row-1][col] - 1,
                  H[row][col-1] - 1);
}
condition(signed<10> threshold) {
  if (H[row][col] >= threshold) {
    report();
  }
}
```

4.2 Run-time parameterizable penalties with ambiguous base

```
alphabet = {A, C, G, T, N}
const unsigned BITWIDTH = 10;
signed<BITWIDTH> H[][];
cell(signed<BITWIDTH> match, signed<BITWIDTH> mismatch,
     signed<BITWIDTH> insert, signed<BITWIDTH> delete,
     signed<BITWIDTH> N_penalty) {
  signed<BITWIDTH> substitute;
  if (query_char == N || ref_char == N) {
    substitute = N_penalty;
  } else if (query_char == ref_char) {
    substitute = match;
  } else {
    substitute = mismatch;
  }
  H[row][col] = max(0,
                  H[row-1][col-1] + substitute,
                  H[row-1][col] + insert,
                  H[row][col-1] + delete);
}
condition(signed<BITWIDTH> threshold) {
  if (H[row][col] >= threshold) {
    report();
  }
}
```

4.3 Affine gap penalty

```
alphabet = {A, C, G, T, N}
const unsigned BITWIDTH = 18;
signed<BITWIDTH> M[][];
signed<BITWIDTH> I[][];
```

```
signed<BITWIDTH> D[][];
cell(signed<BITWIDTH> match, signed<BITWIDTH> mismatch,
     signed<BITWIDTH> gap_open,
     signed<BITWIDTH> gap_extend,
     signed<BITWIDTH> N_penalty) {
  signed<BITWIDTH> substitute;
  if (query_char == N || ref_char == N) {
    substitute = N_penalty;
  } else if (query_char == ref_char) {
    substitute = match;
  } else {
    substitute = mismatch;
  }
  I[row][col] = max(M[row-1][col] + gap_open,
                    I[row-1][col] + gap_extend);
  D[row][col] = max(M[row][col-1] + gap_open,
                    D[row][col-w] + gap_extend);
  M[row][col] = max(0,
                    M[row-1][col-1] + substitute,
                    I[row][col],
                    D[row][col]);
}
condition(signed<BITWIDTH> threshold) {
  if (M[row][col] >= threshold) {
    report();
  }
}
```

4.4 Fixed substitution matrix

```
alphabet = {A, C, G, T}
const unsigned BITWIDTH = 18;
const unsigned<BITWIDTH> HOXD55[4][4] = {
    {91, -90, -25, -100},
    {-90, 100, -100, -25},
    {-25, -100, 100, -90},
    {-100, -25, -90, 91}
};
const unsigned<BITWIDTH> GAP_OPEN = -400;
const unsigned<BITWIDTH> GAP_EXTEND = -30;
signed<BITWIDTH> M[][];
signed<BITWIDTH> I[][];
signed<BITWIDTH> D[][];
cell() {
  signed<BITWIDTH> substitute =
                       HOXD55[query_char][ref_char];
  I[row][col] = max(M[row-1][col] + GAP_OPEN,
                    I[row-1][col] + GAP_EXTEND);
  D[row][col] = max(M[row][col-1] + GAP_OPEN,
                    D[row][col-1] + GAP_EXTEND);
  M[row][col] = max(0,
                    M[row-1][col-1] + substitute,
                    I[row][col],
                    D[row][col]);
}
condition(signed<BITWIDTH> threshold) {
  if (M[row][col] >= threshold) {
    report();
  }
}
```

4.5 Run-time parameterizable substitution matrix

```
alphabet = {A, C, G, T, N}
const unsigned BITWIDTH = 18;
signed<BITWIDTH> M[][];
signed<BITWIDTH> I[][];
signed<BITWIDTH> D[][];
cell(signed<BITWIDTH> sub_mat[5][5],
     signed<BITWIDTH> gap_open,
     signed<BITWIDTH> gap_extend,
     signed<BITWIDTH> N_penalty) {
  signed<BITWIDTH> substitute;
  if (query_char == N || ref_char == N) {
    substitute = N_penalty;
  } else {
    substitute = sub_mat[query_char][ref_char];
  }
  I[row][col] = max(M[row-1][col] + gap_open,
                    I[row-1][col] + gap_extend);
  D[row][col] = max(M[row][col-1] + gap_open,
                    D[row][col-1] + gap_extend);
  M[row][col] = max(0,
                    M[row-1][col-1] + substitute,
                    I[row][col],
                    D[row][col]);
}
condition(signed<BITWIDTH> threshold) {
  if (M[row][col] >= threshold) {
    report();
  }
}
```