Accelerator Language Reference Manual

Avi Chad-Friedman UNI: ajc2212
Alan McNaney UNI: apm2144
Evan Charles O’Connor UNI: eco2116

Table of Contents

1. Introduction

1.1 What is Accelerator?

Accelerator consists of a subset of the syntax of the R language. We will implement basic mathematical operators matrices, boolean operators, if-else structures, for loops, and imperative functions. Our compiler will translate this subset of R to Accelerator enabled C++ to accelerate matrix mathematics and statistical analysis computation. Accelerator will implement a small standard library of functions to assist with printing, meta information regarding matrices, and file I/O with a focus on reading and writing CSV data sets from file to matrices. This will allow programmers and researchers familiar with R to write programs using a subset of known R syntax and still gain the performance improvements made possible via Accelerator's access to parallel computation resources. R is not typesafe. We intend to make Accelerator type safe.

1.2 Why R and OpenMP?

Large scale data is collected continuously from the internet and other sources by businesses, research organizations, and government agencies. This data can necessitate databases with records numbering into the hundreds of millions. This presents the very real challenge of efficiently and meaningfully interpreting that collected data. For instance, how do we sort it? CPU's are designed for general processing, and as such can carry out sorting and analysis on large scale data but are not specialized to this demanding and increasingly frequent task. What hardware resources are readily available and well suited to the task of large scale data manipulation?

Many hardware architectures already contain a large-scale parallel processing hardware device which is overlooked for the purpose of data analysis - the Graphics Processing Unit. The laptop on which this proposal is being written has 384 graphical shading cores clocked to 1029 MHz, and 128 ALU's in it's GPU alone. For reference, this constitutes a mid-level Nvidia GPU. If we can leverage the parallel processing power already available within a system's GPU, we can make large scale matrix manipulation and the application of statistical methods to large scale data sets more efficient than would be possible with a traditional CPU.

Why did we choose to begin with a subset of R's syntax? R is a widely used statistical programming language, and over the last year has greatly increased in popularity. R has no native ability to access the parallel processing resources. Several major industry leaders, such as AMD, ARM, Cray, HP, IBM, Intel, NVIDIA, Oracle, and others have been working since 2008 to create OpenMP - an API and language extension implemented in C, C++, and Fortran which allows access to the large scale parallel processing power of GPUs and other processing cores available to a specific architecture. Our intention is to allow access to the raw power of the GPU from the already familiar and easy to use context of R syntax, improving performance in matrix mathematics and statistical analysis of datasets by leveraging the parallel processing power available in OpenMP enabled C++.

2. Data Types and Literals

- Note: all data types are immutable

2.1 Primitive Types

bool

- 1 byte width boolean data type, represented internally as 0 as a constant for `FALSE` and all other values as `TRUE`.
- bool literal: a constant indicating true or false
- Examples of bool literals:
    - `TRUE`
    - `FALSE`

int

- 32 bit integer data type, represented internally as a binary signed 2's complement bitstring, with min/max values as limited by a 32 bit 2's complement bitstring.
- int literal: an optionally signed string of digits with min/max values bounded that does not contain a "."
- Examples of int literals:
    - `0`
    - `1`
    - `-1`
    - `123456789`
    - `-123456789`

NA

- `NA` is identical functionally equivalent to R's `NA`, which is a representation of value which is not available. It is essentially a null value. This can be assigned to variables or exist in any element in any matrix data type, and returns `NA` when evaluated using any operator against any other data type.
- Example of NA literal:
    - `NA`

double

- Double is a double precision 64 bit floating point type, as defined by the IEEE 754 standard with 1 bit for sign, 11 bits for an exponent part, and a 52 bit significand. Minimum and maximum values are identical to those of a 64 bit floating point type.

- examples of double literal
  - `123456.789`
  - `0.0`
  - `-123.456789`

2.2 Non-Primitive Types

vector

- Used to represent vectors. A vector is an array type composed of primitives. They are an internal representation, and not available to the user outside of parameter passing inside of matrix creation.
- General Vector Creation Syntax
  - vectorIdentifier <- c(comma separated, same type literals or NAs)
  - whitespace ignored
- Examples of vector literals (literal bolded for emphasis)
  - intMatrix <- matrix(**c(1,2,3,4)**)
  - boolMatrix <- matrix(**c(TRUE,FALSE,TRUE,FALSE)**)

matrix

- Used to represent matrices. A matrix is composed of vectors. Vectors of vectors are allowed, but no further nesting is allowed. Elements are represented by values stored contiguously in memory in column major order. An individual matrix may only contain elements of one primitive type, and may contain `NA`'s. The type contained in a matrix is inferred at matrix assignment, and the type of a matrix cannot be changed once it has been created. Matrices cannot be declared without instantiation, and cannot be declared with only `NA` types.
- General Array Creation Syntax
  - `matrixIdentifier <- matrix( c( comma separated, same type literals or NAs), optional (nrow = int, ncol  = int) )`
  - Whitespace is ignored.
- examples of matrix literals:
  - `boolMatrix = matrix(c(TRUE,TRUE,FALSE,TRUE)) # array like matrix containing booleans`
  - `intMatrix = matrix( c(1,2,3,4,5))  # array like matrix containing ints`
  - `charMatrix = matrix( c('a','b','c'))  # array like matrix containing chars, equivalent to a string type`
  - `naMatrix = matrix( c(1,NA,3,NA,5,NA)) # array like matrix containing ints and NA values`
  - `doubleMatrix = matrix( c(1.1,2.2,3.3,4.4,5.5)) # array like matrix containing double values`
  - `twoByTwoMatrix = matrix( c(1,2,3,4), nrow = 2, ncol = 2)`
  - `twoByWithNAMatrix = matrix( c(1,NA,3,NA), nrow = 2, ncol = 2)`

character

- equivalent to a string data type, and is internally composed of char primitives in the style of R. Allows for escaping meaningful characters
    - Escapable characters
        - `'\n' # newline escaping`
        - `'\r' # carriage return escaping`
        - `'\t' # horizontal tab escaping`
        - `'\\' # backslash escaping`
        - `'\"' # double quote escaping`
- character literal: A single column matrix containing an ASCII char in each element.
    - Examples:
        - `"This is a character literal."`
        - `"a"`
        - `"A character literal with an escaped newline \n escaped tab \t and escaped backslash \\"`

2.3 Casting

Casting between types is not permitted in Accelerator. The only exceptions to this are that each data type can be represented as a character data type for printing, and that any variable of any type may contain the value `NA`.

3. Lexical Conventions

3.1 Identifiers

Identifiers refer to variables, functions, or formal arguments for parameter passing to functions. Identifiers are composed of alphanumeric characters, and must begin with a letter of the alphabet. Alphabetic characters can be upper or lower case.

3.2 Keywords

1. `if`
    - Used in `if ( boolean expression ) { statements }` and `if (boolean expression ) { statements } else { statements }`
    - The first set of statements are only executed if the boolean expression evaluates to `TRUE`, and the second only if the boolean expression evaluates to `FALSE`.
2. `else`
    - Used in `if ( boolean expression) { statements } else { statements }`
    - The second set of statements following else are executed should the boolean expression evaluate to FALSE.

3. `function`
    - Used in `functionName <- function ( formal params ) { statements }`
    - Indicates that the `functionName` identifier represents a function identifier.
4. `for`
    - Used in `for ( identifier in int range operator int ) { statements }`
    - Indicates a looping structure, causing the above statements to be executed repeatedly as many times as is indicated by the integer range statement

5. `in`
    - Used in `for ( identifier in int range operator int ) { statements }` looping structures
    - Associates an index variable for looping with a described range
6. `next`
    - used in `for ( identifier in int range operator int ) { statements containing next }`
    - Causes the current loop's execution of statements to be terminated, and control to return to the top of the for loop to evaluate whether new iterations across the statement are necessary.
7. `break`
    - used in `for( identifier in int range operator int ) { statements containing break }`
    - Causes all execution of the for loop to cease.
8. `TRUE`
    - Boolean constant indicating true used in boolean expressions.
9. `FALSE`
    - Boolean constant indicating false used in boolean expressions.
10. `NA`
    - Indicates that a value is not available. Can be assigned to a variable of any data type, or as the value held in the element of any type of matrix.

3.3 Punctuation

1. `,`
    - separates formal parameters in function definition
    - separates actual parameters in function calls
2. `[ ]`
    - matrix access
    - Examples
        - `matrixName[a,b]`
            - `a` and `b` represent valid indices, otherwise this will raise a compilation error

        - `matrixName[a,]`
            - `a` must be a valid row in `matrixName`
            - omission of an int value following the "`,`" requests an entire row's worth of data, returned as a single row matrix
        - `matrixName[,b]`
            - `b` must be a valid column in `matrixName`
            - omission of an int value preceding the "`,`" requests an entire column's worth of data, returned as a single column matrix

3. `()`
    - normal expression evaluation precedence override
    - identifying boolean expressions for `if - else` statements
    - identifying iteration identifiers and ranges in for statements
4. `{}`
    - delineation of a scoped block of statements
5. `“”`
    - character data type declaration

3.4 Comments

- `#`
    - In keeping with R's syntax, there are only single line comments
    - Examples
        - `# comment text`
            - At start of line ignores entire line until the next newline character
        - `statement # comment text`
            - Ignores the remainder of the line until the next newline character

3.5 Operators

Arithmetic Operators

- Applicable to int, double, matrix (element wise, must match type)
    - `+      # addition`
    - `-      # subtraction`
    - `*      # multiplication`
    - `/      # division`
    - `^      # exponentiation`
- Applicable to only integer, matrix of int (element wise, must match type)
    - `%%     # modulus operator`

Assignment Operator

- Applicable to all types
  - `<-    # assignment`

Comparison Operators

- Applicable to int, double, boolean, character, matrix (element wise, must match type)
  - `==    # equality`
  - `!=    # non-equality`
- Applicable to int, double, character, matrix (element wise, must match type)
  - `>     # greater than`
  - `>=    # greater than or equal to`
  - `<     # less than`
  - `<=    # less than or equal to`

Logical Operators

- Applicable to bool, bool matrices
  - `||          # OR`
  - `&&          # AND`
  - `!           # logical negation`

Range Operator

- Applicable to int
  - `:     # range`

Matrix / Matrix Operations

- Applicable to matrix
  - `+     # addition`
  - `-     # subtraction`
  - `*     # multiplication`

Operator Limitations

- Expressions may contain Matrix / Matrix operations, or any other kind, but not both.
- Matrices must be of appropriately matching dimensionality in order for Matrix / Matrix operations to be used. Incorrectly dimensioned matrices will cause a compiler error.
- The use of exponents with matrix data types only allows of the following
  - matrix ^ int
    - Any value

Operator Precedence

- Operator precedence is the same as in C, both for regular operators and Matrix / Matrix operators.

3.6 Whitespace

- Includes
  - `' '   # space`
  - `"\t'  # tab`
- Does not include
  - `'\n'`
    - Newline characters are meaningful as statement separators

4. Syntax

4.1 Program Structure

Programs in Accelerator consist of a series of statements, executed procedurally, which consist of variable assignments, function declarations, and function calls. In keeping with R's syntax, there is no enclosing function, simply statements in a text file with the file extension ".acc". Accelerator is not object oriented, therefore there are no classes.

4.2 Expressions

Assignment Expressions

Assignment in Accelerator is carried out via the <- operator. It is right binding, taking a value from the right hand side and assigning it to the identifier on the left.

Example:

```
      myVar <- "a string of characters"
```

Arithmetic Expressions

These operators represent basic mathematical operations, and are left associative.

Example:

```
      1 + 2
      10 %% 2
      5 * 5
```

In scalar matrix operations, we can apply an elementwise mathematical operation to a matrix. This results in a matrix where the appropriate operation has been carried out on each element in the matrix. This does not mutate the dimensionality of the matrix. These operators are also left associative.

Example:
```
A <- matrix(c(1,2,3,4),nrow=2,ncol=2)
A + 2
A / 2
A * 2
A - 2
```

Matrix Arithmetic Expressions
These represent linear algebra operations on matrices. They must occur between matrices which are of the correct dimensionality for the matrix arithmetic operation.

Examples:
```
A <- matrix(c(1,2,3,4),nrow=2,ncol=2)
# Result:
#     A = [ 1, 3 ]
#         [ 2, 4 ]

B <- matrix(c(4,9,1,11),nrow=2,ncol=2)
# Result:
#     B = [ 4, 1  ]
#         [ 9, 11 ]
# Matrix addition
A + B
# Result:
#     [ 5,   4 ]
#     [ 11, 15 ]

# Matrix multiplication
A * B
# Result:
#     [ 31, 34 ]
#     [ 44, 46 ]

# Matrix subtraction
A - B
# Result:
#     [ -3, 2  ]
#     [ -7, -7 ]
```

Comparison Expressions
All comparison operators are binary operators comparing the left and right operands. They result in a bool. Comparison operators can only be applied to ordered types, such as int, double, and character, and matrices of those types which engages in element-wise comparison. Matrices

must match type with the variable they are compared with in comparisons between matrices and variables.

Example:
```
1 < 2               # bool (TRUE)
intMatrix <- matrix(c(1,2,3,4))
2 < intMatrix       # results in a bool matrix [FALSE,FALSE,TRUE,TRUE]
```

Logical Expressions
Logical expressions evaluate to bool data types and result in a bool. They can be applied to a bool data type and a matrix of type bool resulting in a matrix of type bool. They can be applied to two matrices of type bool, causing matching element logical evaluation.

Example:
```
a <- TRUE
b <- FALSE
a && b              # evaluates FALSE
a || b              # evaluates TRUE
boolMatrix <- matrix(c(TRUE,FALSE))
boolMatrix && a     # evaluates to a matrix containing [TRUE,FALSE]
anotherMatrix <- matrix(c(FALSE,FALSE))
boolMatrix || anotherMatrix    # evaluates to [TRUE, FALSE]
```

4.3 Statements
Statements in Accelerator are syntactically correct instructions, and are executed sequentially.

Expression Statements
Expression statements contain a single expression as defined in the expressions segment above and are terminated by a newline character.

Conditional Statements
Accelerator allows if-else control flow statements. A boolean expression is evaluated. If the boolean expression evaluates TRUE, one block of statements is executed. There is an optional else clause and second block of statements. If these are included and the boolean evaluates FALSE, the second block of statements is executed

Examples:
```
if (boolean expression)
      { block of statements executed on TRUE evaluation }
```

```
if (boolean expression)
      { block of statements executed on TRUE evaluation }
else
      { block of statements executed on FALSE evaluation }
```

Looping Structures
Accelerator has one looping structure, a for loop which uses an index variable which iterates across an indicated range, executing an attached block of statements as many times as is indicated by the range.

Example
```
for ( identifier in range expression )
      { statements }
```

Loop Interruption
- Individual iterations across the statement block can be terminated with the "next" keyword. Following a "next" control returns to the top of the loop to evaluate whether the loop should continue or not. Next expressions return type NA.
- The entire execution of the for statement can be terminated using the "break" keyword. Break also returns type NA.
- Any code placed following either the "next" or "continue" keyword is ignored.

Function Statements
Function statements invoke a previously defined function, causing flow of execution to transfer to the block of statements defined within the function. Functions return, by default, the evaluated value of the final expression in their block of statements.

Return Values
Each statement is evaluated to a value. Functions, loops, and if-else statements return the value of the last statement evaluated within their associated block of statements.

4.4 Scope

Scope in Accelerator is divided between normal CPU memory space, and shared memory scope as defined by OpenMP enabled C++. These memory spaces are disjoint. It is possible to transfer data from the CPU scope to the GPU scope and vice versa. This is handled automatically by Accelerator, and is carried out when operations are performed that involve a matrix. This implementation is intended to shield the programmer from being forced to manage disparate memory spaces, concurrent memory access, and synchronization issues. From the perspective of the programmer, there is one memory space.

GPU Memory Scope
When legal operations are carried out on two matrices or on a matrix and a scalar data type, all information relevant to the operation is transferred to the GPU. These operations are then parallelized, results obtained, and transferred back into main CPU memory space. As such, operations involving matrices block until return values are obtained from the GPU.

5. Standard Library Functions

Accelerator provides a lean set of standard library functions to create vectors and matrices, and perform essential input and output operations.

Vector

- `c <- function(arg0, arg1, ...)`
  - Creates a vector from the given arguments. The arguments must all be of the same type and can be integers, doubles, booleans, or characters.
- `matrix <- function(vector, nrow, ncol)`
  - Creates a `nrow` by `ncol` matrix and populate it by column from the elements of vector.
  - `vector`: An optional vector
  - `nrow`: desired number of rows
  - `ncol`: desired number of columns
- `length <- function(vector)`
  - Returns the length of the vector
  - `vector`: a vector of any type
- `dimensions <- function(matrix)`
  - Returns an integer vector of length 2 with the number of rows in matrix as the first element and the number of columns in the second
  - `matrix`: a matrix of any type
- `reverse <- function(vector)`
  - Reverses the order of the elements in vector
  - `vector`: a vector of any type

FILE I/O

- `print <- function(vector)`
  - Writes the characters of vector to standard output
  - vector: a character vector

- `write <- function(matrix, csv_file_path)`
    - Writes the elements of matrix to the csv file `csv_file_path`.
        - If there is already an open file descriptor for `csv_file_path`, then it will append, otherwise it will open a new file descriptor.
    - `matrix`: a matrix of any type
    - `csv_file_path`: a file path
- `read <- function(csv_file_path)`
    - Returns a matrix populated with the contents of the csv file at `csv_file_path`.
    - `csv_file_path`: a file path

6. Parallelizable Operations

User Experience

- Accelerator automatically parallelizes all matrix operations, including element wise operations. This limits flexibility, but completely abstracts the decision of whether a section of code is safe or wise to parallelize. The tradeoff heavily favors the programmer, who may be familiar with R syntax but not well versed, if at all, in multi-threaded programming. For small matrices, the overhead of sending data to and from the GPU is large; however, the vast majority of Accelerator use cases will involve large data sets and, therefore, large matrices. R itself already exists as a robust and efficient tool for the purpose of small matrix manipulation in a non-parallel computation environment. Automatically parallelizing all matrix operations covers a large subset of the cases in which multi-threading would be beneficial.

OpenMP Implementation

- Accelerator takes advantage of a core subset of compiler directives OpenMP provides. Again, the advantage of using relatively simple compiler directives is that the programmer can write useful parallelizable code without concerning his or herself with where or how the parallelization should occur.
- Matrix Operations
    - Matrix operations of the form A <operation> B, or A <operation> n where A and B are matrices and n is a scalar
        -

```
#define CHUNK_SIZE 100;
int chunk = CHUNK_SIZE;
float** a, b, c; \\initialization of a and b
\\or int n; if A <operation> n
\\or #pragma omp parallel shared(a,n,c,chunk)
```

```
#pragma omp parallel shared(a,b,c,chunk)
{
#pragma omp for  schedule(static)
\\for loops to compute c
}
```

- An operation within a loop structure of the form a = a <operation> b, where a and b are scalars

  -
    ```
    int a, b;
    \\some initialization
    #pragma omp parallel for reduction(<operation> : a)
    \\for loop with a = a <operation> b in the body
    ```

7. Grammar

- The below is our grammar, the contents of our parser.mly
- It has been tested with menhir, and presently contains no shift/reduce errors.

```
%{ open Ast %}

%token NA ASSIGN PLUS MINUS TIMES EOF IF FOR ELSE NOELSE COLON IN
%token DIVIDE EQ NEQ LT LEQ GT GEQ LPAREN RPAREN LBRACE RBRACE FUNCTION
%token MOD EXP AND OR NOT LBRACK RBRACK NEXT BREAK DLIN COMMA
%token <string> CHARACTER
%token <int> INT
%token <float> DOUBLE
%token <bool> BOOL
%token <string> ID

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc COLON
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left EXP
%right NOT

%start program

%type <Ast.program> program
```

```
%%

program :
 | decls EOF { $1 }

decls:
 | /* nothing */ { [],[] }
 | decls stmt    { ($2 :: fst $1), snd $1 }
 | decls fdecl   { fst $1, ($2 :: snd $1) }

fdecl:
 | ID ASSIGN FUNCTION LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE DLIN
    { { fname = $1;
        formals = $5;
        body = List.rev $8 } }

formals_opt:
  | /* nothing */            { [] }
  | formal_list              { List.rev $1 }

formal_list:
  | ID                       { [$1] }
  | formal_list COMMA ID     { $3 :: $1 }

actuals_opt:
  | /* nothing */            { [] }
  | actuals_list             { List.rev $1 }

actuals_list:
  | expr                     { [$1] }
  | actuals_list COMMA expr  { $3 :: $1 }

data:
  | BOOL                     { BoolLit($1) }
  | ID                       { Id($1) }
  | CHARACTER                { Character($1) }
  | DOUBLE                   { DoubleLit($1) }
  | INT                      { IntLit($1) }
  | NA                       { Na }

stmt:
  | expr DLIN                                       { Expr($1) }
  | LBRACE stmt_list RBRACE                         { Block($2) }
  | LBRACE loop_stmt_list RBRACE                    { Block($2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE         { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt            { If($3, $5, $7) }
  | FOR LPAREN ID IN expr RPAREN loop_block_body    { For($3, $5, $7) }
```

```
expr:
  | data                                        { $1 }
  | bool_expr                                   { $1 }
  | arith_expr                                  { $1 }
  | ID ASSIGN expr                              { Assign($1, $3) }
  | LPAREN expr RPAREN                          { $2 }
  | ID LBRACK expr COMMA expr RBRACK            { MatrixAcc($1, $3, $5) }
  | ID LBRACK COMMA expr RBRACK                 { MatrixCol($1, $4) }
  | ID LBRACK expr COMMA RBRACK                 { MatrixRow($1, $3) }
  | ID LPAREN actuals_opt RPAREN                { FuncCall($1, $3) }

arith_expr:
  | expr COLON  expr    { DualOp($1, Range, $3) }
  | expr PLUS   expr    { DualOp($1, Add, $3) }
  | expr MINUS  expr    { DualOp($1, Sub, $3) }
  | expr TIMES  expr    { DualOp($1, Mult, $3) }
  | expr DIVIDE expr    { DualOp($1, Div, $3) }
  | expr MOD    expr    { DualOp ($1, Mod, $3) }
  | expr EXP    expr    { DualOp ($1, Exp, $3) }

bool_expr:
  | expr EQ     expr    { DualOp($1, Equal, $3) }
  | expr NEQ    expr    { DualOp($1, Neq, $3) }
  | expr LT     expr    { DualOp($1, Lthan, 3) }
  | expr LEQ    expr    { DualOp($1, Leq, $3) }
  | expr GT     expr    { DualOp($1, Gthan, $3) }
  | expr GEQ    expr    { DualOp($1, Geq, $3) }
  | expr AND    expr    { DualOp ($1, And, $3) }
  | expr OR     expr    { DualOp ($1, Or, $3) }
  | NOT expr            { SingOp (Not, $2) }

loop_block_body:
  | LBRACE loop_stmt_list RBRACE      { Block(List.rev $2) }

loop_stmt_list:
  | /*nothing*/                       { [] }
  | stmt_list DLIN                    { $1 }
  | loop_stmt_list loop_expr DLIN     { $2 :: $1 }

loop_expr:
  | NEXT                              { Next }
  | BREAK { Break }

stmt_list:
  | stmt                              { [$1] }
  | stmt_list stmt                    { $2 :: $1 }
```