FAST VECTOR PROCESSING LANGUAGE

LANGUAGE REFERENCE MANUAL

Gowri Kanugovi <gk2263@columbia.edu>
Pratap V Prabhu <pvp2105@columbia.edu>
Ravindra Babu Ganapathi <rg2547@columbia.edu>

TABLE OF CONTENTS

1. Introduction

Fast Vector Processing Language (FVPL) allows programmers to transparently and efficiently utilize the power of SIMD instructions (such as SSE, SSE2) to compute large amount of sequential data at higher speeds. Manipulation of large sequential data is common in various domains like image processing, databases and cryptography, but most of the current C/C++ compilers generate slow native code for x86 family of processors. The new processors include a vector processing unit which has the capability to compute data in parallel. These instructions known as SIMD (Single Instruction Multiple Data) operate on 128 bits of data at a time.

FVPL aims to make use of the power of the new generation processors.

2. Lexical Conventions

FVPL comprises tokens such as: keywords, identifiers, comments, integer constants, floating point constants, operators and separators. It is a free form language; spaces, tabs and new lines are ignored and considered to only serve as delimiters between tokens.

2.1. Comments

Single line comments in FVPL are begin with the characters // like in the C-language. Multi-line comments are also supported by FVPL. Such comments begin with /* and should be terminated by */

2.2. Identifiers

Identifiers are sequence of letters, digits and the underscore ('_') character. The first letter however has to be only either an alphabet or the underscore character. Identifiers cannot begin with a digit. FVPL identifiers are case-sensitive. Following are some examples of FVPL identifiers

Valid identifiers: A, foo, a, _bar, bar_foo, count2
Invalid identifiers: 1A, a#, foo-bar

2.3. Keywords

The following table summarizes the identifiers used as keywords in FVPL. These keywords cannot be used as otherwise.

int	Float	double	dynamic	main
for	dynamic	void	return	sizeof
static	int8	if	else	

2.4. Constants

Constants provide programmers the ease of initializing any of their identifiers to one of the supported primitives. The different types of constants supported by FVPL are:

2.4.1. *Integer constants:* Integers of FVPL consists of an optional '+' or '-' sign followed by any number of digits in the range of 0-9.

2.4.2. *Floating point constants:* Floating point numbers in FVPL comprises of an optional '+' or '-' sign followed by an integer of one or more digits. This is followed by a decimal point which is then followed by an integer of one or more digits.

2.5. Operators

FVPL supports operations on both scalars and vectors.

2.5.1. *Operators on Scalars:* A programmer can perform the following actions on a scalar type in FVPL:

- Arithmetic operators: The operators '+, '-', '*' and '/' are supported by FVPL. The semantics of the operators are similar to those of addition, subtraction, multiplication and division respectively. The multiplication and division operators are however supported only for the int and float data types in FVPL. The precedence and associativity of operators follows the same conventions as in the C-language.
- Bitwise logical operators: FVPL supports bitwise logical AND, OR, XOR and NOT operations. These are denoted by '&', '|', '^' and '~' respectively.
- Assignment operators: The assignment operator in FVPL is denoted the '=' symbol. This operator assigns the value of the right hand side expression to the left hand side expression.
- Sizeof operator: The sizeof operator returns the size of the operand in bytes. The result is an integer value and thus can be used for assigning any integer variable.

2.5.2. *Operators on Vectors:* FVPL, being a vector processing language tries to provide the programmer the maximum benefit of vector operations. These operators can be used transparently by the programmer as though he is performing the operation on a scalar. For example, there is no need for him to maintain a loop to perform operation on each of the vector member. The following operators are supported by FVPL:

- Vector arithmetic operators: The operators '+', '-', '*' and '/' are supported by FVPL. The semantics of the operators are similar to those of addition, subtraction, multiplication and division respectively. Addition of two vectors implies that each element of one vector is added to the corresponding element of the other vector.

  For e.g.: Three vectors A,B and C

  *The operation (A+B)*C implies that the sum of each of the element in A and B is multiplied by the corresponding element in C*
- Vector logical operators: FVPL supports bitwise logical operations on the vector elements. The logical AND, OR, XOR and NOT are represented as '|&', '|', '^' and '~' respectively. Any logical vector operation implies that the operator is applied on each of its corresponding elements.

For e.g.: Two vectors A and B

The operator A&B implies a bitwise logical AND between every element of A and the corresponding element of B.

- Vector assignment operator: The vector assignment operator denoted by '=' operator implies that every element of the vector is initialized with the value of the scalar element on the right hand side of the operator.

  For e.g.: Vector A

  A=5 assigns the value 5 to every element of A

- Vector initialization: At the time of the creating the vector, FVPL allows the programmer to initialize every element of the vector. This is particularly useful when the programmer is trying to create small vectors and wants to assign distinct values to it manually.

  For e.g.: Vector A

  A = {1, 2, 3, 4, 5} initializes the vector with 5 elements each with the value given in the braces.

- Vector concatenation: FVPL allows two vectors to be concatenated with each other by using the '@' operator. At the end of this operation we will have one vector which contains all the elements of the two vectors involved in the operation.

  For e.g.: Vector A = {1, 2, 3}

  Vector B = {4, 5, 6}

  A@B returns {1, 2, 3, 4, 5, 6}

- Vector copy. In FVPL, elements of one vector can be copied into the other vector by using the vector copy operator. The operator used is same as the assignment operator, but in this case the right hand side of the operation has to be a vector.

  For e.g.: Vector A, B

  A = B implies every element of B is copied into the corresponding position in A i.e. A[0]=B[0], A[1]=B[1]… A[999]=B[999]

- Vector casting: FVPL supports casting of vector from float to integer or vice-versa. Casting operation is applied to each of the element of the vector.

  For e.g.: Vector A = {1.2, 2.3, 3.4}

  Vector B = (int)A then B will contain {1,2,3}

- Vector sizeof operator: The sizeof operator on vectors returns the size in bytes of the total memory allocated to the vector. For example, if the vector contains 'n' integer elements then the sizeof operator returns (4*n) bytes.

2.6. Separators

The two separators supported by FVPL are comma (',') and semi-colon (';'). The separator ';' is used to indicate the end of a statement whereas the ',' separator separates two identifiers.

2.7. Block delimitation

In FVPL, the opening flower braces '{' indicates the beginning of a block of code and the closing flower braces '}' indicate the end of the block of code. Blocks play a major role in defining the scope of variables. All variables defined within a block are visible only to the code in that block. Scoping is explained in more detail in the next section.

2.8. Scoping

Scope of variables can be defined in two different contexts in FVPL:

- *Local scope*: Variables declared in a block of code are visible only within that block. This is the local scope of the variable. Accessing the variable anywhere outside its scope will result in a compilation error indicating that the variable is undefined.
- *Global scope*: A global variable is declared outside all functions. These variables can be accessed by any part of the program. If any function alters the value of this variable, then the altered variable is seen by all other functions. However, local variables with the same name override the global variable within that block.

3. Data Types

Every identifier in FVPL is associated with a type which indicates the way the identifier is interpreted by the program.

3.1. Basic Type

FVPL allows data manipulation on both the scalar and vector types.

- Scalar data types: FVPL supports the following basic data types on scalars:
  int8 is a sequence of 8 bits
  int is a sequence of 32 bits
  float is a single precision floating number, having a size of 32 bits
  double is a double precision floating number, having a size of 64 bits

- Vector data types: FVPL supports the following basic data types on vectors:
  int8 creates a vector such that each of its element is a sequence of 8 bits
  int creates a vector such that each of its element is a sequence of 32 bits
  float creates a vector such that each of its element is a single precision floating point number with size of 32 bits
  double creates a vector such that each of its element is a double precision floating point number with size of 64 bits

4. Branching Construct

In FVPL, the following constructs are used to control the flow of code:

- *if: 'if*' is the keyword used for conditional execution of code. If the condition associated with the 'if' statement is true, then the code block associated with it is executed
  Syntax: if(condition) { /* statements to be executed */ }

- *else:* 'else' is the keyword used in conjunction with 'if'. When the condition associated with the 'if' statement evaluates to false then the code block associated with 'else' is executed.
  Syntax: else { /* statements to be executed */ }

- *return:* The 'return' keyword passes the flow of control to the statement which called for the execution of a particular function. This keyword is specifically used to return from the called function back to the calling function.
  Syntax: return;

5. Looping Construct

The looping construct in FVPL is the keyword '*for'*. The semantics of FVPL '*for'* is same as the C language 'for' loop. It is used to execute the same piece of code till some condition is met.

```
Syntax: for(initialization; condition; looping)
        {
            /* statements to be executed till
               the termination condition is
               reached */
        }
```

6. Declarations

Any variable used in the FVPL program needs to be declared before it is used in the program. The declaration of a variable should include the data type, the identifier name and also additionally whether it requires memory to be allocated to it dynamically. If the programmer does not request the variable to be allocated dynamically, it is assumed that the variable will be allocated memory in a static manner.

Syntax: (dynamic/static) data-type identifier, identifier;

Two identifiers of the same type can be declared in a single line and separated with a comma. Declaration statements can optionally include the initialization of the variable.

7. Functions

FVPL supports function calls like the C-language. Functions are written to perform a particular sub-task which can be reused. In FVPL, functions need to be declared like any other variable before they are actually used.

The starting point of an FVPL program, like in C, is the *main* function. Therefore, any functions that are used in the program need to be either defined or declared before it.

Functions can optionally return some value to the caller. After returning from the function, the calling function resumes execution. Given below is the syntax of function declaration

Syntax: return-type function-name(function-parameters) { //body of the function }

Return-type: It can be any one of the data types supported by FVPL i.e. int, float, double. If the function is not returning a value to the caller, then the return type should be “void”.

Function-name: It is the identifier of the function. This name is used as the reference to the function when the function is to be called.

Function-parameters: A list of variables that are passed to the function body by the caller. The parameters can be of any data type but the order in which these data types are passed to the function is important.

Function-body: A block of code, which performs the task of the function. Any variable declared within the function exist only within the function and cannot be viewed anywhere outside. But the function can access and modify the global variables in the program.

8. Example program

Following is an example FVPL program.

```
int main()
{
   int A[1024];
   int B[1024];
   int C[1024];
   int i;

   /*array initializing code goes here. Either load from file or
   initialize in program */

   //Every element of the vector B is multiplied by 2 and added to
   //the corresponding element of vector A. The result is stored in
   //the corresponding position in vector C.

   C = A + B * 2;

 }
```

9. Lexer

{
type token = EOF | MAIN |IF |ELSE |WHILE |FOR |RETURN| LBRACKET | RBRACKET | STATIC |INT |FLOAT |DOUBLE |INT8 |VOID |LPARAN |RPARAN |LBRACE |RBRACE |PLUS |MINUS |MUL|DIV |OR |AND |XOR |NOT |ASN |SEP |COMMA |LT |GT |LTEQ |GTEQ |EQ |NOTEQ |DIGIT | ID
}

let digit=['0'-'9']

rule token = parse
| eof { EOF }
| "main" { MAIN}
| "if" { IF}

```
| "else"                                  { ELSE}
| "while"                                 { WHILE}
| "for"                                   { FOR}
| "return"                                { RETURN}
| "static"                                { STATIC}
| "int"                                   { INT}
| "float"                                 { FLOAT}
| "double"                                { DOUBLE}
| "int8"                                  { INT8}
| "void"                                  { VOID}
| '('                                     { LPARAN}
| ')'                                     { RPARAN}
| '{'                                     { LBRACE}
| '}'                                     { RBRACE}
| '['                                     { LBRACKET}
| ']'                                     { RBRACKET}
| '+'                                     { PLUS}
| '-'                                     { MINUS}
| '*'                                     { MUL}
| '/'                                     { DIV}
| '|'                                     { OR}
| '&'                                     { AND}
| '^'                                     { XOR}
| '~'                                     { NOT}
| '='                                     { ASN}
| ';'                                     { SEP}
| ','                                     { COMMA}
| '<'                                     { LT }
| '>'                                     { GT }
| "<="                                    { LTEQ }
| ">="                                    { GTEQ}
| "=="                                    { EQ }
| "!="                                    { NOTEQ }

| digit+                                  { DIGIT }
| ['a'-'z''_' 'A'-'Z']+(digit|['a'-'z''A'-'Z'])* { ID}
| _                                       { token lexbuf }


{

  }
```

10. Grammar

File Consists of start symbol Main and user defined functions

Main -> INT MAIN LPARAN RPARAN LBRACE (STATEMENT)* RBRACE

STATEMENT -> (DECL| EXPR| COND| LOOP| FUNCALL| LBRACE STATEMENT

RBRACE) SEP

DECL-> TYPE VAR_DECL SEP

TYPE-> INT| FLOAT| DOUBLE| INT8|

VAR_DECL -> VAR COMMA VAR | VAR

VAR -> ID | ID LBRACKET DIGIT RBRACKET

EXPR -> ID ASN TERM | TERM

TERM -> TERM OP TERM | ID

OP -> PLUS | MINUS | MUL | DIV | OR | AND | XOR

COND -> IF LPARAN BOOL_EXPR RPARAN STATEMENT (ELSE STATEMENT)?

BOOL_EXPR -> ID RELOP ID | DIGIT

RELOP -> LT | GT | LTEQ | GTEQ | EQ | NOTEQ

LOOP -> FOR LPARAN INIT SEP COND SEP INC RPARAN

INIT -> ID ASN DIGIT |

INC -> ID ASN ID (PLUS|MINUS) DIGIT

FUNCALL-> ID LPARAN ARG RPARAN

ARG -> ARG COMMA ARG| ID

FUNDEF-> RETTYPE ID LPARAN PARAM RPARAN LBRACE STATEMENT RBRACE

PARAM -> PARAM COMMA PARAM | TYPE ID

RETTYPE -> TYPE | VOID