Matrix Language (ML) Project Proposal
COMS W4115

Kyle Jackson (kdj2109) • Jessica Valarezo (jgv2108) • Jared Greene (jmg2227) • Alex Barkume (ajb2233) • Caroline Trimble (cdt2132)

February 10th, 2016

1. Introduction & Motivation

ML is a parallelized programming language with a focus on 1D, 2D or 3D matrix operations, designed for computational efficiency and a strong focus on real-time applications. This language has the ability to segment large matrices into rows, columns or individual indices and run them independently on various threads. This ability is particularly applicable to applying computer vision algorithms to images, including filtering, transformations, color space conversions, thresholding, and various others. When a number of these algorithms are applied sequentially to a very large image, their total execution time can be considerably long. As such, our language focuses on reducing the operational time of matrix algorithms.

2. Language Overview

Built-in Types

Type	**Description**
`int`	integer value
`float`	floating point value
`bool`	1 or 0
`str`	a text value - can be single character or longer string
`matrix`	matrix of values - can be `matrix, tuple, int, float`
`struct`	C-type struct

```
// Variable declaration and assignment
int x

float y = 5.0

str z = “Hello, World!”
```

```
bool b = 0

// Matrix declaration and assignment
matrix M0
M0 = {}

// Structs
struct rect:
      float x = 0
      float y = 0
      float area = x * y

struct rect rectA

// Accessing fields
rect.x = 5.5
rect.y = 10.25
```

Variable names can begin with any lowercase or uppercase letter (`^[A-Za-z]`). Thereafter, a variable can have any combination of lowercase and uppercase letters, numbers, and underscore (`[0-9A-Za-z_]+$`).

Keywords

Keyword	Description
`async <function_name>`	Executes a function/block of code in parallel on a new thread. Returns a value.
`sync`	Waits for all running threads to stop before continuing on.
`pfor`	Every iteration of the for loop is run on a different pthread.
`print`	Prints out any data type
`if`	If in if-else statement
`else if`	Strings together multiple ifs in if-else statement
`else`	Else in if-else statement
`main`	Main function
`return`	Return a value from a function. In the case that the function has spawned threads, waits for all threads to terminate.

Comments

```
// - line comment
/** **/ - block comment
```

Operators

Operators	Description
=	Assignment operator
>	Greater than operator
<	Less than operator
>=	Greater than or equal to operator
<=	Less than or equal to operator
&&	Logical AND operator
||	Logical OR operator
!	Logical NOT operator
+, -, *, /	Arithmetic operators
==	Returns an int, 1 if the values are equal, 0 otherwise
!=	Returns an int, 0 if the values are equal, 1 otherwise
:	Function definition Returns a list of ints, upper bound is exclusive.

Matrix Operators	Description
,	Column
{}	Row
()	n-tuple as an single entry in a matrix
[]	Matrix indexing - begins at 0, 0
+, -, *, /	Matrix and scalar operations
<, <=, >, >=	Compares each entry in matrix to a single constant float or int, or to each corresponding entry in another matrix.

```
//matrix operators:  +, - , *, /
M2 = M1 * M2

//matrix scalar operators: +, - , *, /
M1 = 3 * M1
```

Primitive Transformations

Primitive transformation can be used in combination to allow the programmer to express any arbitrarily complex matrix transformation. All of these transformations return a `matrix`.

```
// Concatenating matrices - must have same # of rows, appended horizontally
M2 = [M1, M2}

// Returns a specified region of a matrix using python-like indexing
M2 = M1.[][1:3]    // Returns a matrix w/ 2nd and 3rd column of M1

M2 = M1.[1:3][]    // Returns a matrix with 2nd and 3rd rows of M1

M2 = M1.[0][]      // Returns a matrix with 1st row of M1

M2 = M!.[][!0]     // Returns matrix without the 1st column of M1

// Returns matrix of the intersection of 1st 3 columns and rows
M2 = M1.[0:3][0:3]
```

Creating Matrices

```
M1 = {1, 2, 3}
print M1
M1 =
1  2  3
// 1 row, 3 columns

M2 = {1,2,3}{1,2,3}
print M2
M2 =
1  2  3
1  2  3
// 2 rows, 3 columns

M3 = {(255 255 255),(0 0 0),(255 255 255)}
print M3
M3 =
255 255 255  0 0 0  255 255 255
```

```
// 1 row, 3 columns

M4 = {(255 255 255),(0 0 0),(255 255 255)}{(255 255 255),(0 0 0),(255 255 255)}
print M4
M4 =
255 255 255  0 0 0  255 255 255
255 255 255  0 0 0  255 255 255
// 2 rows, 3 columns

M5 = M1 M1  // {1,2,3}{1,2,3}
print M5
M5 =
1  2  3
1  2  3
// 2 rows, 3 columns
```

Functions

```
// Built-in functions

// Read in the image file, return a 2D matrix
read(img_file_name):

// Write matrix M1 to file named img_file_name
out(M1,img_file_name):

// Return the length of a row or column in a matrix
len(M1.[0][])

// Return 1D matrix of integers. If start argument is omitted, defaults to 0
// range(stop)
// range(start,stop)
range(len(M1.[0][]))

// Multiply each entry in matrix by 3
3 * M1

// Standard matrix multiplication
M1 * M1

// The argument <matrix_name> is passed by value to the function.
<matrix_name> = <function_name> <matrix_name>


// Same as above but function is run on a parallel thread.
<matrix_name> = async <function_name> <matrix_name>
```

```
//*****************************************************************************
// Function Declaration

// function with no return type (void)
func <function_name> (<parameters>):
      // indented function block, where indentation is two spaces

// function with a return type
func <return_type> <function_name> (<parameters>):
      // indented function block, where indentation is two spaces
```

3. Target Language

Our compiler will produce C combined with OpenMP and pthreads for parallel capabilities.

4. Sample Program

```
main:
  matrix M1 = {(255 255 255),(0 0 0),(255 255 255)}
  matrix M2 = M1 M1 M1
  print M2
  M2 = convertToBinary M2
  print M2

func matrix convertToBinary(matrix M0):
  int thresh = 128

  pfor i in 0:len(M0.[][0]):
    for j in 0:len(M0.[i][]):
      if M0.[i][j] < thresh
        M0.[i][j] = 0
      else
        M0.[j][i] = 1
  return M0
```

Output…

```
  M2 =
    255 255 255  0 0 0  255 255 255
    255 255 255  0 0 0  255 255 255
    255 255 255  0 0 0  255 255 255
  M2 =
    1 0 1
    1 0 1
    1 0 1
```

```
main:
  matrix M1 = {(0 0 0),(0 0 0),(255 255 255)}
  matrix M2 = M1 M1 M1
  print M2

  /** RGB values are converted to binary. The current thread waits for
      convertToBinary to return **/
  M2 = convertToBinary M2

  // Two independent functions are run on parallel threads
  int sum = async getSum M2
  int avg = async getAvg M2

  // The current thread waits for all parallel threads to terminate
  sync

  print sum
  print avg

func matrix convertToBinary(matrix M0):
  int thresh = 128

  pfor i in 0:len(M0.[][0]):
    for j in 0:len(M0.[i][]):
      if M0.[i][j] < thresh
        M0.[i][j] = 0
      else
        M0.[j][i] = 1
  return M0

func float getAvg(matrix M0):
  int sum = getSum M0
  int total = len(M0.[][0])* len(M0.[0][])
  return float(sum)/float(total)

func int getSum(matrix M0):
  int sum = 0
  pfor i in 0:len(M0.[][0]):
    for j in 0:len(M0.[0][]):
      sum += M[0].[i][j]
  return sum
```

Output…

```
6
0.333
```

5. Foreseeable Challenges

- Parallelism implementation
    - Determining when code can be parallelized and how to check when a the programmer mistakenly tries to parallelize serial work
- Thread management
    - Contention scope and limiting open threads so underlying processor is never overwhelmed
    - Thread safety: managing shared resources is pfor loops and race conditions