COMS 4115: Proposal

Caleb Babatunde, Avi Chad-Friedman, Alan McNaney, Evan O'Connor
UNIs: cba2117, ajc2212, apm2144, eco2116

Edwards

Accelerator

Contents

What is the Accelerator language?

Accelerator consists of a subset of the syntax of the R language. We will implement basic mathematical operators for vectors and matrices, boolean operators, if-elseif-else structures, for loops, and imperative functions. Our compiler will translate this subset of R to OpenMP enabled C++ to accelerate matrix mathematics and statistical analysis computation. Accelerator will implement a small standard library of functions, including commonly used statistical analysis functions from R such as mean, median, standard deviation into OpenMP. This will allow programmers and researchers familiar with R to write programs using 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.

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? CPUs 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 ALUs in its GPU alone[1]. For reference, this constitutes a mid-level Nvidia GPU. If we can leverage the parallel processing power already available within a systems 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 Rs 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++.

Notes

[1]http://www.notebookcheck.net/NVIDIA-GeForce-840M.105681.0.html

Data Types

Listing 1: Accelerator's syntax is a subset of R

```
# boolean
x <- TRUE      # 1 not promotable to TRUE
y <- FALSE     # 0 not promotable to FLASE
TRUE == FALSE  # returns FALSE; comparison operators (<, >, <=, >=, !=, ==)
```

```
# double
x <- 3.4    # decimal indicates to compiler that value is double not integer
x * 10.0    # double 34. returned; standard algebraic operations (+, -, /, *, ^) valid
x < 100.0   # returns TRUE; comparison operators (<, >, <=, >=, !=, ==)

# integer
x <- 3      # lack of decimal indicates to compiler that value is integer not double
x * 2       # integer 6 returned; standard algebraic operations (+, -, /, *, ^) valid
3 %% 2      # integer 1 returned; modulo only valid for integers, not doubles
5 %/% 2     # integer 2 returned; value floored for integer divison
x < 100     # returns TRUE; comparison operators (<, >, <=, >=, !=, ==)

# character
fname <- "Joe"; lname <- "Smith"            # assignment of character strings
paste(fname, lname)                         # returns "Joe Smith"; concatenation
sprintf("Hello World!")                     # prints "Hello World!" to the stdout
substr("Hello World!", start=1, stop=5)     # returns "Hello"; substring
fname < 'ZZ'                                # returns TRUE; lexical comparison operators
                                            # (<, >, <=, >=, !=, ==)

# matrix
A <- matrix( c(1, 2, 3, 4, 5, 6),           # elemental data
   nrow=2, ncol=3,                          # no. rows and columns
   byrow=TRUE)                              # fill matrix by rows

A[2,3]      # returns 6; elemental access
A[2,]       # returns 2nd row of A; row and column access valid
A ^ 2       # integer exponentiation of matrices valid
A + A       # +, -, and * matrix operations valid
```

Operators and Functions

Listing 2: Some of the built in functionalities of Accelerator

```
# logical operators (applied to raw or logical 'number-like' objects)
!x            # logical negation
x & y         # logical AND
x | y         # logical OR
xor(x, y)     # elementwise exclusive OR
isTRUE(x)     # evaluates whether x is TRUE

# looping structure
for (i in 1:50){
   sprintf("Hello World!")
}

# recursive gcd
gcd <- function(a,b) ifelse (b==0, a, gcd(b, a %% b))

# Not Available Value (equiv to Null)
a <- c(1,3,NA,9)

# Conversion to character; will be used with sprintf() for printing to stdout
x <- as.character(3.14)
```

```
# passing matrices as arguments to functions
sin(a) # applies sin() function for each element in the matrix

# cross product
v1 <- c(1,2)
v2 <- c(2,3)
A <- crossprod(v1, v2)

# transpose v1
v1 <- t(v1)

# Accelerator is type safe, unlike R
x <- "1"
x + 4 # returns an error
```

Code Examples

Listing 3: Handling matrix multiplication in Accelerator

```
# Matrix Manipulation

# matrix creation
A <- matrix(c(1,2,3,4), nrow=2, ncol=2, byrow=TRUE)

# populates matrix by columns
# A =         [,1]  [,2]
#        [1,]    1    2
#        [2,]    3    4

B <- matrix(c(1,2,3,4), nrow=2, ncol=2, byrow=TRUE)

# matrix multiplication
C <- A %*% B

# C =         [,1] [,2]
#        [1,]    7   10
#        [2,]   15   22
```

Listing 4: Compiled OpenMP matrix manipulation code

```
//
//  Hello.cpp
//  OpenMP
//

#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <iostream>

using namespace std;

void fill(float ** cols){
    for(int i = 0; i<4; i++){
        //allocate memory for each row
        cols[i] = (float *)malloc(4*sizeof(float));
    }
}
```

```
void populate(float ** A, int cols, int rows){
    for(int i = 0; i<cols; i++){
        for(int j = 0; j<rows; j++){
           float f = static_cast <float> (rand()) / static_cast <float> (RAND_MAX/10.0);
           A[i][j] = floor(f);
        }
    }
}

//Source: http://www.appentra.com/parallel-matrix-matrix-multiplication/
int alg_matmul2D(int m, int n, int p, float** a, float** b, float** c)
{
    int i,j,k;
#pragma omp parallel shared(a,b,c) private(i,j,k)
    {
#pragma omp for  schedule(static)
        for (i=0; i<m; i=i+1){
            for (j=0; j<n; j=j+1){
                a[i][j]=0.;
                for (k=0; k<p; k=k+1){
                    a[i][j]=(a[i][j])+((b[i][k])*(c[k][j]));
                }
            }
        }
    }
    return 0;
}

void print_matrix(float ** A, int cols, int rows){
    for(int i = 0; i<cols; i++){
        cout<<endl;
        for(int j = 0; j<rows; j++){
            cout << A[i][j] << " ";
        }
    }
}

int main()
{
    float **a = (float **)malloc(4*sizeof(float *));
    float **b = (float **)malloc(4*sizeof(float *));
    float **c = (float **)malloc(4*sizeof(float *));
    fill(a);
    fill(b);
    fill(c);
    populate(a, 4, 4);
    populate(b, 4, 4);
    print_matrix(a, 4, 4);
    cout << endl << "x";
    print_matrix(b, 4, 4);
    cout <<endl <<"=";
    alg_matmul2D(4, 4, 4, c, a, b);
    print_matrix(c, 4, 4);
    cout << endl;
    return 0;
}
```

Listing 5: Sample output

```
0 1 7 4
5 2 0 6
6 9 3 5
8 0 0 5
x
6 0 3 0
4 6 5 9
8 5 0 6
4 7 9 7
=
76 69 41 79
62 54 79 60
116 104 108 134
68 35 69 35
```