COMS 4115: Proposal

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

Edwards

Accelerator

Contents

Why R and OpenACC? **3**

What is the Language? **3**

Data Types **3**

Functions and Operators **4**

Code Examples **5**

Why R and OpenACC?

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 pure 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.

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 within a GPU. Cray, CAPS, Nvidia, and PGI have recently come together to create OpenACC - a cross platform API built on top of the C language which allows access to the large scale parallel processing power of GPUs and other processing hardware. 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 large-scale matrix mathematics and statistical analysis of large datasets by leveraging the large scale parallelism available in OpenACC.

Notes

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

What is the 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 and while loops, and imperative functions. Our compiler will translate this subset of R to OpenACC enabled C containing compiler directives to accelerate matrix mathematics operators. Accelerator will implement functions, specifically porting commonly used statistical analysis functions from R such as mean, median, and standard deviation into OpenACC C, again taking advantage of compiler directives to parallelize calculation and provide performance increases. This will allow programmers and researchers familiar with R to write programs using R syntax and still gain the performance improvements made possible via OpenACC compiler directives. Additionally, Accelerator will add type safety to its subset of R syntax.

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
```

```
# double
x <- 3.4    # decimal indicates to compiler that value is double not integer
x * 10      # double 34. returned; standard algebraic operations (+, -, /, *, ^) valid

# 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

# 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

# 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
```

Functions and Operators

Listing 2: Some of the built in functionalities of Accelerator

```
# logical operators (applied to raw or logical 'number-like' objects and vectors)
!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){
   print(i)
}

# 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
x = as.character(3.14)

# passing vectors as arguments to functions
sin(a) # applies sin() function

# 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 Mainpulation

# matrix creation
A = matrix(c(1,2,3,4), nrows=2, ncols=2)

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

B = matrix(c(1,2,3,4), nrows=2, ncols=3)

# matrix multiplication
C <- A %*% B

# C =         [,1] [,2]  [,3]
#        [1,]    7   15   23
#        [2,]   10   22   34
```

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++){
            A[i][j] = 1;
        }
    }
}
```

```
//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);
    populate(c, 4, 4);
    alg_matmul2D(4, 4, 4, a, b, c);
    print_matrix(c, 4, 4);
    return 0;
}
```