GPU-accelerated Computation for Statistical Analysis of the Next-Generation Sequencing Data

Yilan Liu

May 2014

A Major Qualifying Project Report:
submitted to the Faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Yilan Liu

Date: May 2014

Approved:

Professor Zheyang Wu, Advisor

This report represents the work of one or more WPI undergraduate students.
Submitted to the faculty as evidence of completion of a degree requirement.
WPI routinely publishes these reports on its web site without editorial or peer review

Acknowledgments

I would like to thank Professor Zheyang Wu for his generous help in guiding me through my research with his experience, expertise and patience. During the course of the project, he met with me every week to make sure I accomplished good quality of work while patiently helping me find my way to solutions.

I would also like to thank WPI Mathematics Department for preparing me with sufficient background to complete this project.

Abstract

The next-generation sequencing technologies are pouring big data and pushing the frontier of life sciences toward new territories that were never imagined before. However, such big data impose great computational challenges to statistical analysis of these data. It is important to utilize Graphics Processing Unit (GPU)'s large throughput and massive parallelism to process large data with extremely high efficiency. In this project we develop GPU based tools to address the statistical computation challenges in analyzing the next-generation sequencing data. Our work contains three components. First, we accelerate general statistical analysis in R, a generic environment for statistical computation, which is often limited to using Central Processing Unit (CPU) for computations. After studying various approaches of using GPU in R, we adopted the best solution to combine R with GPU. An R package is created to shift a set of critical R functions onto GPU computation. It allows users to run R code with GPU extensions that enable much faster large-data computation. Second, we address a set of specific computation-intensive problems in simulating genetic variants in whole-genome sequencing data. A GPU-based R package is created to facilitate some typical simulations in genetic association studies. Third, we break the CPU limitation of Variant Tools, a popular toolkit for the next-gen sequencing analysis, by extending its functionality to more the powerful parallel computation of GPU. For this purpose an R-function interface is created so that we can connect Variant Tools' sophisticated data processing and annotation to the powerful GPU-accelerated data analysis. The work of this project is valuable to whole-genome sequencing studies, as well as to general statistical computational need. It is part of the research funded to the WPI Department of Mathematical Sciences by Major Research Instrumentation Program of National Science Foundation. The R packages and the interfacing code as well as their documentation will be available to view and download at users.wpi.edu/~zheyangwu/.

Table of Contents

1 Background 1

1.1 Graphics Processing Unit (GPU) 1

1.2 General-Purpose Computing on Graphics Processing Units (GPGPU) 1

1.3 CUDA (Compute Unified Device Architecture) 2

1.4 Big Data from the Next Generation Sequencing 4

1.5 R 5

1.6 Variant Tools 5

2 Problem Statement 7

3 Design Considerations 8

3.1 Perspective from GPGPU Standpoint 8

3.2 Perspective from CUDA Standpoint 9

3.3 Perspective from R and Variant Tools 10

4 Procedure 12

4.1 CUDA Programming 12

4.1.1 How CUDA Program Works 12

4.1.2 An Example of CUDA Program 13

4.2 Variant Tools Source Code 15

4.3 Interfacing between R and C/C++ 16

4.4 Interfacing between R and CUDA 18

5 Results 20

5.1 CUDA Extensions for R 20

5.2 Performance Advantage of Using CUDA Extensions for R 20

5.2.1 gpuMatMult() versus %*% 21

5.2.2 rpuDist() versus dist() 22

5.3 GRRNG (GPU-R Random Number Generator) Package........ 23

5.4 Performance Advantage of Using GRRNG 24

5.4.1 gpuRnorm() versus rnorm() 24

5.4.2 gpuRunif() versus runif() 25

5.4.3 gpuRsort() versus sort() 26

5.5 GpuRInterface Package 27

5.6 GRGDS (GPU-R Genetic Data Simulator) Package 27

5.7 Interface between Variant Tools and GPU-based R function 28

5.8 Analysis of performance of GPGPU 29

6 Future Work 30

7 References 32

Appendix I: List of Functions in CUDA Extensions for R 35

Appendix II: Source Code for gpuRnorm() 40

List of Figures

1 Unified Array of Processors for NVIDIA GeForce 8800 2
2 Accelerating HMMER using GPUs Scalable Informatics 3
3 MUMmerGPU: High-through DNA Sequence Alignment Using GPUs 4
4 GPU Design vs. CPU Design 8
5 Design of CUDA 9
6 How CUDA Executes Programs 10
7 Code from RTest class in RTester.py 16
8 Source Code of add.c 17
9 Source Code of add.r 17
10 Result Shown in R terminal 17
11 An RInside Example Code 18
12 Granger Times Performance Comparison 19
13 Code in R to compare the performance of gpuMatMult() and %*% 21
14 Runtime Comparison between %*% (CPU) and gpuMatMult() (GPU) 21
15 Runtime Comparison between dist() and rpuDist() 22
16 Runtime Comparison between rnorm() and gpuRnorm() 25
17 Runtime Comparison between runif() and gpuRunif() 26
18 Runtime Comparison between sort() and gpuRsort() 26

1 Background

1.1 Graphics Processing Unit (GPU)

First introduced in 1991 with the technical definition of "a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second" [1], GPU has greatly changed how we interact and experience with modern computers. Before GPU was invented, transformation from numbers in computer to pixels on screen is done on Central Processing Unit (CPU), and CPU computed each pixel in series, which was very inefficient. GPU, though similar to CPU, is designed specifically for performing the complex mathematical and geometric calculations that are necessary for graphics rendering. [2]

Though GPUs were originally used for and still are most commonly seen in 2D/3D graphics rendering, now their capabilities are being utilized more broadly to accelerate computational workloads in areas such as financial modeling, cutting-edge scientific research, and oil and gas exploration [3].

1.2 General-Purpose Computing on Graphics Processing Units (GPGPU)

In 2006, NVIDIA GeForce 8800 mapped separate graphics stage to a unified array of processors. The logical graphics pipeline for GeForce 8800 is a recirculating path that visits the processors three times, and there was much fixed-function graphics logic between visits [7]. This design is illustrated in Figure 1. It allowed dynamic partitioning of the array to vertex shading, geometry processing, and pixel processing, which enabled massive parallelism. This marked the birth of GPGPU [4]. GPGPU refers to the use of GPU for general purpose such as mathematical and scientific computing, instead of simply for graphics rendering.

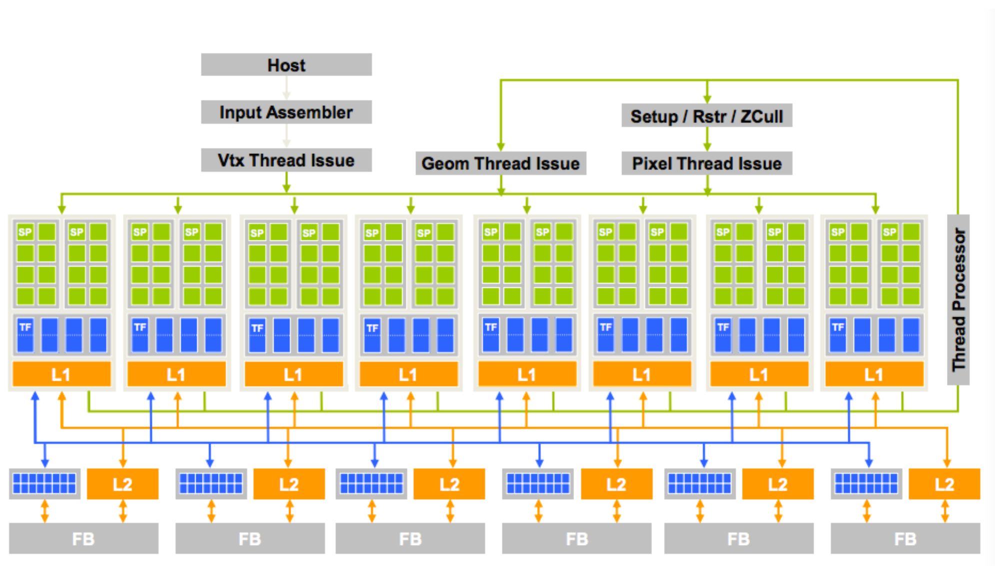

Figure 1. Unified Array of Processors for NVIDIA GeForce 8800 [4]

After GeForce 8800 marked the beginning of GPGPU, GPGPU still faced many constraints, such as working with the corner cases of the graphics API, limited texture size/dimension, limited outputs, etc. Many steps have taken to reduce the constraints: Designing high-efficiency floating-point and integer processors; Exploiting data parallelism by having large number of processors; Introducing shader processors fully programmable with large instruction cache, instruction memory, and instruction control logic; Reducing the cost of hardware by having multiple shader processors to share their cache and control logic; Adding memory load/store instructions with random byte addressing capability; Developing CUDA C/C++ compiler, libraries, and runtime software models [4]. After these steps were taken, more benefits of GPGPU have shown when compared with CPU computing and GPGPU has then become a popular topic of research. GPGPU takes great advantage of GPU's data-parallelism and large throughput, and is most commonly used in fields that benefit from large data-parallelism, such as data mining, statistics and financial analytics.

1.3 CUDA (Compute Unified Device Architecture)

CUDA is the software development platform developed by NVIDIA for general purpose computing on NVIDIA GPUs. It is a parallel computing extensions to many

popular languages and a powerful drop-in accelerated libraries to turn key applications and cloud based compute appliances. It is widely used by researchers since its introduction in 2006 [5]. It is used across many domains: Bioinformatics, computational chemistry, computational fluid dynamics, computational structural mechanics, data science, defense, etc [6]. It has shown great success in demonstrating its massive computing power in those fields. As we can see from Figure 2 and Figure 3, there is a large performance benefit by using a CUDA-enabled GPU in Bioinformatics and Life Sciences.

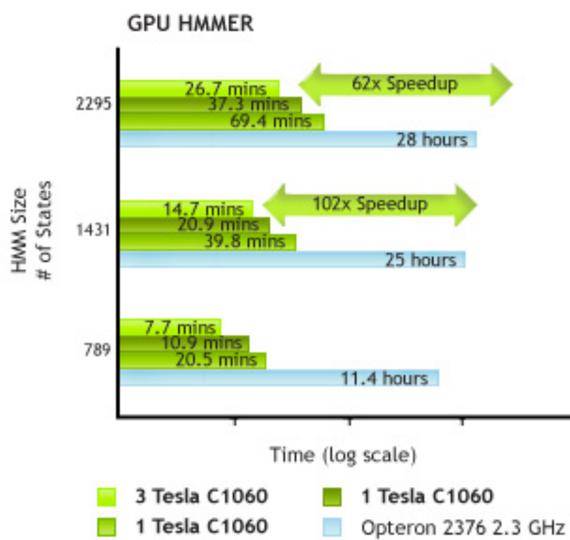

Figure 2. Accelerating HMMER using GPUs Scalable Informatics [6]

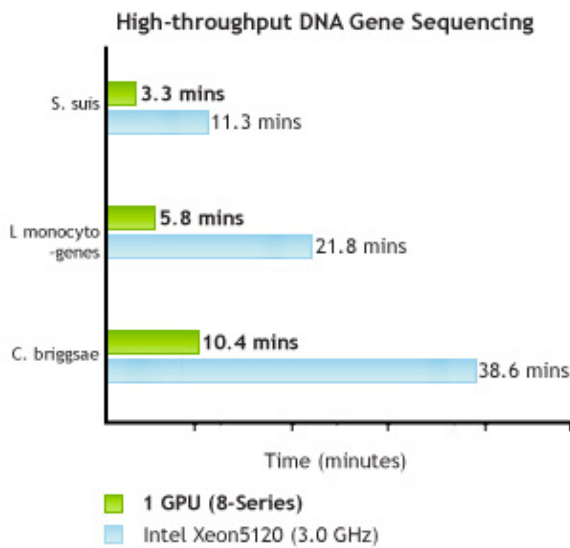

Figure 3. MUMmerGPU: High-through DNA Sequence Alignment Using GPUs [6]

1.4 Big Data from the Next Generation Sequencing

The next-generation deep-sequencing technology has led to exponentially increasing genetic data with a significantly reduced cost. Such data provide unprecedentedly rich information on complex genetic system, but also impose great challenges on data processing and analysis. We are confronting the issue of one-thousand-dollar genome but hundred-thousand-dollar interpretation [30]. To address this issue, both methodology innovation and computational power are highly demanded, especially in genome-wide association studies (GWAS) that aim to detect causative genetic factors for common human diseases from millions of genetic variations, such as the single nucleoid polymorphisms (SNPs). The power of the GPU computation is a key to address the heavy computational need. Computation for many analyses of the next-generation sequencing data is highly parallel in nature because the huge amount of genetic association tests can be independently calculated by an identical procedure. Literature have shown that the similar problems can be implemented to exploit GPU computation that increases the speed 10 to 100 times faster than a comparable CPU [31].

1.5 R

R is a software environment and language for statistical computations. It is free and highly extensible. The R software contains a base program to provide basic program functionality, and smaller specialized program modules called packages can be added on top of it. The R environment includes [11]:

- "An effective data handling and storage facility",
- "A suite of operators for calculations on arrays, in particular matrices",
- "A large, coherent, integrated collection of intermediate tools for data analysis",
- "Graphical facilities for data analysis and display either on-screen or on hardcopy", and
- "A well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities".

R is the most popular open source statistical environment in the biomedical research community at the moment [16].

1.6 Variant Tools

Variant Tools is an open-source software tool for the manipulation, annotation, selection, and analysis of variants in the context of next-gen sequencing analysis [29]. It is project-based with a flexible command line interface. With more than 20 commands and more than 200 registered users, it is certainly one of the most powerful and widely used toolkit in genome sequencing analysis [8].

Variant Tools contains commands to call variants, import variants, export variants,

annotate variants, reference genome, conduct association analysis, etc. It is a platform under which we can analyze our data using methods, compare and analyze results, and re-compare and re-analyze using different methods or annotation sources, based on the information obtained from previous analyses [29].

2 Problem Statement

R, the core of Variant Tools' association analysis framework to support customized testing, is widely used in statistics and it is playing a very important role in managing biological big data. Most of R functions and libraries run on CPU. However, the performance benefit in using GPU for computing started to shine in recent years. R, as a tool that relies heavily on computation, has hardly benefited from the emerging of GPGPU since it is not highly extensible with GPGPU. CUDA, the most widely used development kit for GPGPU, as well as other common development platforms for GPGPU, are not highly compatible with R. In order to benefit from both the tremendous computing power of GPU and the massive statistical analysis power presented by R, it is necessary to come up with a solution to run R conveniently together with C/C++, the main development language for GPGPU programming platforms such as CUDA.

In order to run R with C/C++ conveniently while maximizing the performance advantage, a simple integration between R and C/C++ will not be enough. Our integration needs to fit the parallelism design concept of GPU while enabling users to keep as much code in R as possible since we rely extensively on R libraries for statistical analysis. Therefore, the ultimate solution needs to be both R-friendly and parallelism-friendly. This requires us to dive deep and study the basic concepts of GPGPU, learn how to program using CUDA, how to write R code, how to communicate between R and C/C++ and how to create a product that is user-friendly for R and Variant Tools users.

3 Design Considerations

3.1 Perspective from GPGPU Standpoint

GPU is the master of parallel programing, since it was originally designed for highly parallel tasks like image rendering. As we can see in Figure 4, GPU possesses much more processing units than CPU. It possesses independent vertices and fragments. Temporary registers are zeroed; there are no shared or static data among registers and there are no read-modify-write buffers. In short, there is no communication between vertices or fragments [9]. Each GPU's processing unit is based on the traditional core design, can do integer/floating point arithmetic, logical operations and many other operations. From hardware design point of view, GPU is a complete multi-threaded processing design, with complex pipeline platform and full power to process tasks in each thread.

Figure 4. GPU Design vs. CPU Design [9]

By the design concept of GPU, GPU is more suitable for data-parallel processing. GPU can handle lots of data on which the same computation is being executed, with no dependencies between data elements in each step in the computation.

Therefore, our extension between C/C++ and R must be dividing large data into smaller, independent chunks to process in parallel instead of dividing a huge task into smaller, dependent subtasks. For instance, wrapping our R core in a thread and dividing it into sub-processes won’t benefit from GPU’s large throughput;

pre-processing our data into smaller parts and running a CUDA call separately on each segment of data will.

3.2 Perspective from CUDA Standpoint

CUDA is a parallel computing platform and programming model for using NVIDIA GPUs for general purpose computing. The organization of GPU explains the structure of CUDA programs. The main function is executed by a single thread on the CPU. Kernels, which consist of many lightweight threads, are executed by functions on the host. Kernel launches change control of the program from CPU to GPU, and the CPU pauses while the GPU is running. After the kernel finishes executing, the CPU resumes running the program [10].

NVCC is both a preprocessor and a compiler for CUDA. When it encounters CUDA code, NVCC automatically compiles the code into GPU executable code, which means generating code to call the CUDA Driver. If it encounters Host C/C++ code, it will call the platform's C/C++ compiler to compile, for example Visual Studio's own Microsoft C/C++ Compiler. Then it will call Linker to put together all compiled modules, along with CUDA libraries and standard C/C++ libraries to become the ultimate CUDA Application. Therefore, NVCC mimics the behavior of the GCC/G++ compiler. This process is demonstrated in Figure 5.

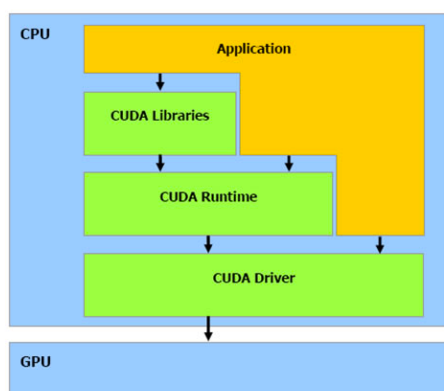

Figure 5. Design of CUDA [32]

As shown in Figure 6, while executing programs, CUDA lets each kernel in the host to be executed in a grid in the GPU. Every grid contains multiple blocks, and every block can contain multiple threads.

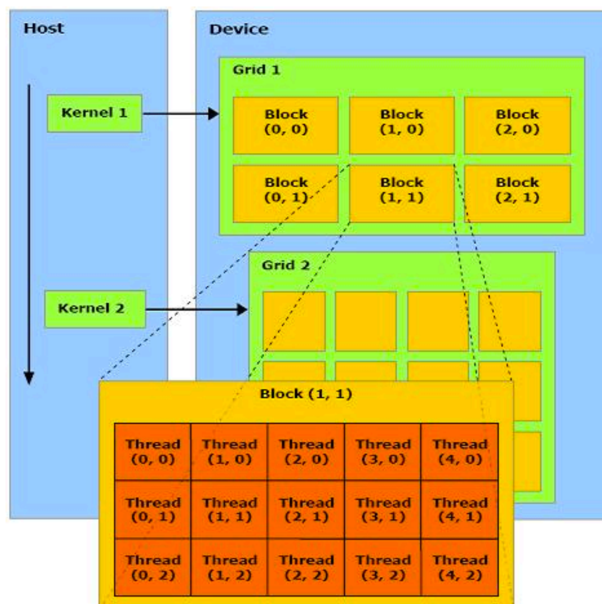

Figure 6. How CUDA Executes Programs [32]

The nature of CUDA limits our main coding languages for GPGPU computation to be C/C++. Also, the design of CUDA made it not possible for dividing an R thread into several sub-processes. Dividing data into smaller chucks and process them in parallel has to happen at a low level, and at an early stage. A wrapper around R core does not fit into CUDA's design concept.

3.3 Perspective from R and Variant Tools

R is a powerful language and environment for statistical analysis, including both computing and graphics. With many built-in and download packages and libraries, R provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, etc.) and graphical techniques, and is highly extensible [11]. We certainly want to utilize the benefits of using R in our statistical analysis. In other words, we want to write code in R as much as possible when the tradeoff in performance does not go beyond a threshold.

Although the CUDA programming interface is vector oriented and fits perfectly with the R language paradigm [19], R, as a standalone integrated suite, cannot be integrated with CUDA unless we replace some existing R methods with methods partly written for CUDA computations.

As for Variant Tools, we only concern about its R Testing interface. Its interface to R on the R core, and it is highly extensible and flexible. Therefore, as long as our extension runs with R, it will run on Variant Tools.

4 Procedure

During the course of this project, I first familiarized myself with CUDA environment and practiced programing in CUDA. Then I studied the source code of Variant Tools to see from where I can build the extension. Next, I did research on interfacing between R and C/C++ as well as how to write CUDA code for R functions. Then I dived deep into studying and testing existing CUDA related packages for R. After I finished studying CUDA R package written by others, I tried to come up with my own CUDA package for R, along with tests to show their performance.

4.1 CUDA Programming

4.1.1 How CUDA Program Works

CUDA programs execute on two different processing units: the host (CPU) and the device (GPU). The host relies on traditional random access memory (RAM) and the device typically uses Graphical Double Data Rate (GDDR) RAM, which is designed for use on graphics cards. Since the device can only access GDDR RAM and the host can only access the traditional RAM, we need to call "special" functions in CUDA that transfers data between the two. These functions must be launched on the host. Examples of such functions include `cudaMalloc`, very is similar to `malloc` in C, and `cudaMemcpy`, which transfers data between the host and the device [14].

Typically a CUDA program begins with initializing variables and allocating memory, followed by inputting data on the host. Then, these data are copied to the device using the `cudaMemcpyHostToDevice` function. Next, kernels, which are the functions that run on the device, will be launched. Kernels contain all the code to do the computations/calculations that we want to run on the device. Information stored in memory on the device will remain for the duration of the program; it is preserved between kernel launches. After kernels have run, data are copied back from the device to the host using `cudaMemcpyDeviceToHost` [14].

4.1.2 An Example of CUDA Program

Here is an example for using CUDA to compute sum of cubes.

First, we start with writing sum_cubes1.cu.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#define MAX_SIZE 1048576

int data[MAX_SIZE];

// Generates random numbers from 0-9
void generateNumbers(int *number, int size) {
    for(int i = 0; i < size; i++) {
        number[i] = rand() % 10;
    }
}

// Kernel function to compute the sum of cubes
__global__ static void sumOfCubes(int *num, int *result,
                                  clock_t *time)
{
    int sum = 0;
    clock_t start = clock();
    for(int i = 0; i < MAX_SIZE; i++) {
        sum += num[i] * num[i] * num[i];
    }

    *result = sum;
    *time = clock() - start;
}

int main(int argc, char **argv) {
    int *gpudata, *result;
    clock_t *time;

    // generate the input array on the host
    generateNumbers(data, MAX_SIZE);

    // allocate GPU memory
    cudaMalloc((void**) &gpudata, sizeof(int) * MAX_SIZE);
    cudaMalloc((void**) &result, sizeof(int));
    cudaMalloc((void**) &time, sizeof(clock_t));

      // transfer the input array to the GPU
    cudaMemcpy(gpudata, data, sizeof(int) * MAX_SIZE,
               cudaMemcpyHostToDevice);

    // launch the kernel
    sumOfCubes<<<1, 1, 0>>>(gpudata, result, time);

    int sum;
    clock_t time_used;

    // copy back the result to the CPU
    cudaMemcpy(&sum, result, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&time_used, time, sizeof(clock_t),
           cudaMemcpyDeviceToHost);

    // free memory on GPU
```

```
    cudaFree(gpudata);
    cudaFree(result);
    cudaFree(time);

    printf("The sum of squares is %d. Total execution time: %d\n", sum, time_used);
    return 0;
}
```

In the above code, `sumOfCubes()` is the simple kernel function to calculate the sum of cubes. `main()` is the function for the host to do all the pre-process and post-process for CUDA. `cudaMalloc` is the function for allocating bytes of linear memory on the device and returns in a pointer to the allocated memory, like `malloc` in C. `cudaMemcpy` copies bytes from source to destination, where kind is one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`, which specifies the direction of the copy [14].

However, this is rather inefficient since we only have one thread in total, and the total elapsed GPU clock time is 108079606. So we can make improvements by adding `#define THREAD_NUM 256`. We want to have 256 threads to process the data in parallel. Then we change the kernel function to be:

```
__global__ static void sumOfCubes(int *num, int* result,
                                  clock_t *time)
{
    const int tid = threadIdx.x;
    const int size = MAX_SIZE / THREAD_NUM;
    int sum = 0;
    clock_t start;
    if(tid == 0) start = clock();
    for(int i = tid * size; i < (tid + 1) * size; i++) {
        sum += num[i] * num[i] * num[i];
    }

    result[tid] = sum;
    if(tid == 0) *time = clock() - start;
}
```

In the above code, `threadIdx` is the CUDA built-in parameter representing the thread number of the current thread. The total number of threads is 256, so there are 256 threads processing in parallel, each represented by its thread number. In the main function, we change `sumOfCubes<<<1, 1, 0>>>(gpudata, result, time);` to `sumOfCubes<<<1, THREAD_NUM, 0>>>(gpudata, result, time);`. This ran more efficiently than the previous program with a total elapsed GPU clock time of 4476286, which is a lot less than 108079606.

To achieve even better parallelism, we can define multiple blocks in each grid. Doing so we are fully utilizing the design of CUDA, by diving our data into grids, then into blocks, and then into threads [13]. Instead we have `sumOfCubes<<<1, THREAD_NUM, BLOCK_NUM>>>(gpudata, result, time);` in the main function to launch the kernels. It will be even more efficient than the second version shown above.

4.2 Variant Tools Source Code

After carefully studying the source code of Variant Tools, I found out most of its R-Testing related content is written in Python, and the main Python classes for parsing R-Testing related arguments and running R tests are called RTest and SKAT. Both of them are in RTester.py. Variant Tools simply parses the arguments that we have passed in, helps generating the .R source file, and then as we can see in Figure 7, sends it as a command using `runCommand()` function in Python. This suggests that Variant Tools' R-Testing related content solely depends on the R environment installed, so as long as we write our own package and extend R with it, we will be able to utilize it in Variant Tools. This R interfacing mechanism is flexible because the RTest method of `vtools associate` will have no control over what are implemented inside the R program or what packages we have installed on top of R.

```
#
@timectrl(tmout, 'time expired')
def timed_command(cmd, instream = None, msg = ''):
    return runCommand(cmd, instream, msg)
#
self.loadData()
# write data and R script to log file for this group
if self.data_cache_counter < self.data_cache or self.data_cache < 0:
    try:
        with open(os.path.join(env.cache_dir, '{0}.dat.R'.format(self.gname)), 'w') as f:
            f.write('\n'.join(self.Rscript + self.Rdata))
        self.data_cache_counter += 1
    except:
        pass
# print self.formatOutput()
cmd = "R --slave --no-save --no-restore"
cmd_logger = env.logger.debug
na_value = float('nan')
try:
    out = timed_command(cmd, "{0}".format('\n'.join(self.Rscript + self.Rdata) + self.formatOutput()),
                        "R message for {0}".format(self.pydata['name']))
    if out == 'time expired':
        cmd_logger = env.logger.warning
        na_value = -9
        raise ValueError("running time {}s has expired".format(tmout))
    else:
        out = out.split()
    if len(self.outvars):
        out = out[out.index("BEGIN-VATOUTPUT") + 1:out.index("END-VATOUTPUT")]
        res = [typemapper(x[1])(y) for x, y in zip(self.outvars, out)]
    else:
        res = [self.pydata['name']]
except Exception as e:
    cmd_logger("Association test {} failed while processing '{}': {}".\
                        format(self.name, self.gname, e))
    res = [na_value]*len(self.outvars)
return res
```

Figure 7. Code from RTest class in RTester.py

4.3 Interfacing between R and C/C++

There are two simplest ways of interfacing between C/C++ and R: Wrapping C/C++ code in an R wrapper, or running R code as a command in C/C++.

The following is one of the most straightforward approaches to wrap C code in R. All we need to do is to use .C(), .Call() or .External() function provided by R to extend R with compiled C code. First we write our C code, then we compile the C code by typing `R CMD SHLIB <file_name.c>` in the command prompt. A .so object file will then be generated, and we will use `dyn.load(<file_name.so>)` in R to load the object file. After loading the .so shared object file in R, we call the C function that we have written using .C(), .Call() or .External(). If we use .C(), the C function we are calling should be of return type void, which means the compiled code should not return anything except through its arguments [15].

The following code, as shown in Figure 8 and Figure 9, is an example to do simple addition. add.c is the C file we use to do a simple addition of two numbers. Then we

use `R CMD SHLIB add.c` to generate add.so. After add.so is generated, all we need to do is to source add.r in R and type `add(<number1>, <number2>)`, and their sum will be returned. The result in shown in Figure 10.

```
#include <stdlib.h>
#include <R.h>

void add(double *a, double *b, double *c) {
        *c = *a + *b;
        //prints to R console
        Rprintf("%.0f + %.0f = %.0f\n", *a, *b, *c);
}
```

Figure 8. Source Code of add.c [33]

```
add <- function(a,b) {
  ##check to see if function is already loaded
  if(!is.loaded("add"))
    dyn.load("add.so")

  c <- 0
  z <- .C("add",a=a,b=b,c=c)
  c <- z$c
  return(c)
```

Figure 9. Source code of add.r [33]

```
> source ("add.r")
> add(1,1)
1 + 1 = 2
[1] 2
> add(1012398, 3481)
1012398 + 3481 = 1015879
[1] 1015879
> 
```

Figure 10. Result Shown in R terminal

To run R script in C, we can use the `system()` call in C to run the R script as a command. For example, if we have a .R source file called "calculate.R", then we can use `system("R CMD BATCH calculate.R");` in C to run it.

In addition to the two straightforward approaches mentioned above, we may also use the many power libraries for interfacing between R and C/C++. One of them is Rcpp, it provides a powerful API on top of R, allowing direct interchange of rich R objects between R and C++. Rcpp modules provide easy extensibility using declarations and Rcpp attributes greatly facilitates code integration. It utilizes .Call() interface provided by R and build the whole package on top of it [16].

Another library is RInside. RInside wraps the existing R embedding API in an easy-to-use C++ class, thus makes it much easier to embed R in C++ code on Linux, OS X, or Windows [17]. The code in Figure 11 demonstrates how to display "Hello, word!" in R from C++.

```
// -*- mode: C++; c-indent-level: 4; c-basic-offset: 4;  tab-width: 8; -*-
//
// Simple example showing how to do the standard 'hello, world' using embedded R
//
// Copyright (C) 2009 Dirk Eddelbuettel
// Copyright (C) 2010 Dirk Eddelbuettel and Romain Francois
//
// GPL'ed

#include <RInside.h>                    // for the embedded R via RInside

int main(int argc, char *argv[]) {

    RInside R(argc, argv);              // create an embedded R instance

    R["txt"] = "Hello, world!\n";       // assign a char* (string) to 'txt'

    R.parseEvalQ("cat(txt)");           // eval the init string, ignoring any returns

    exit(0);
}
```

Figure 11. An RInside Example Code [17]

4.4 Interfacing between R and CUDA

In order to communicate and exchange data between R and CUDA, the simplest way is to link CUDA code with C or C++ code, then use C/C++ to interface with R. Simply convert the C functions which does the computation to CUDA functions, then use nvcc to compile it instead of using `R CMD SHLIB`. nvcc will generate the .so file, and you can use the same approach mentioned in Section 4.3 to wrap the .so file in R. However, most of the popular R function implementations involve no parallelism and they can only be executed as separate instances on multicore or cluster hardware for large data-parallel analysis tasks. I have failed to find a way to simply wrap the R core in C and divide the process into parallel. Therefore, it is necessary to replace existing R functions with ones specifically written for CUDA in order to use CUDA for

computation. Rewriting functions in R and using CUDA to do the parts that can benefit from parallelism is unavoidable.

While we can always write our own R wrapper and CUDA code, there are R libraries that have already been developed using CUDA. They contain many useful R routines. A list of these libraries can be seen on http://cran.r-project.org/web/views/HighPerformanceComputing.html under section "Parallel computing: GPUs".

One of the most popular packages among them is gputools, developed by Josh Buckner, Mark Seligman and Justin Wilson at the University of Michigan [18]. It moves frequently used R functions in our work to run on GPU by using CUDA. It contains functions such as gpuCor, gpuCrossprod, gpuDist, gpuGlm, gpuGranger, gpuLm, etc. Its performance versus CPU is extremely impressive, as we can see in Figure 12. Figure 12 shows the correlation between R's granger.test in package 'MSBVAR' and gputools' gpuGranger function, while the granger.test is running on a single CPU thread on Intel Core i7 920 and gputools is running on a GTX 260 GPU. As we can see, as the number of total points tested increases, the runtime of granger.test grows exponentially but gpuGranger does not.

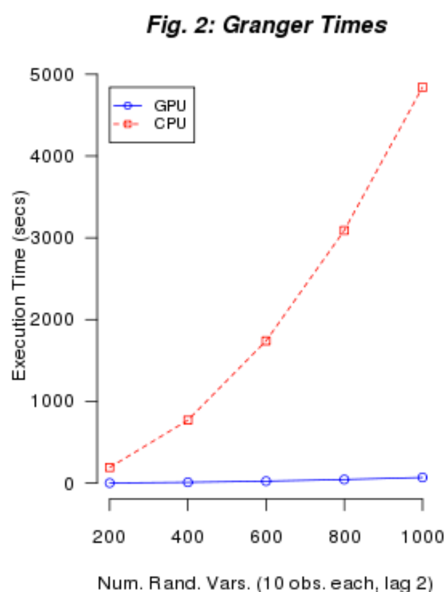

Figure 12. Granger Times Performance Comparison [18]

5 Results

After conducting an extensive amount of research and experiments, I found eight good R libraries for CUDA. I installed them successfully and used timed tests to analyze their performance. I have also developed my own CUDA extension for R focusing on simulations, called GRRNG (GPU-R Random Number Generator). A series of performance tests were also conducted to show the time efficiency of functions in my package. On the other hand, I developed an interface package called GpuRInterface to make calling functions in CUDA extensions easier. I used both of my packages to improve the performance of certain functions in the R Genetic Data Simulation library provided by Professor Wu, as well as to reduce the total runtime of R Testing commands in Variant Tools.

5.1 CUDA Extensions for R

Here we provide a list of R-CUDA libraries and functions as a reference for R users to adapt the GPU power into their own R script, after carefully selecting from all available R libraries for CUDA. They are:

- gputools [20]
- HiPLARM [21]
- Rpud [22]
- Magma [23]
- gcbd [24]
- WideLM [25]
- cudaBayesreg [26]
- permGPU [27]

Methods contained in these packages are documented in Appendix I. They can replace existing R functions that use CPU for computation.

5.2 Performance Advantage of Using CUDA Extensions for R

In order to show the efficiency of GPGPU, I tested several functions provided by the libraries listed in Section 5.1 using a personal computer, with Intel Core i7 4770k as its CPU and NVIDIA GeForce GTX770 as its GPU.

5.2.1 gpuMatMult() versus %%*

I used a sample program, as shown below in Figure 13, to test the performance of gpuMatMult() in gputools with %*% in R.

```
gpu.matmult <- function(n) {
    A <- matrix(runif(n * n), n ,n)
    B <- matrix(runif(n * n), n ,n)

    tic <- Sys.time()
    C <- A %*% B
    toc <- Sys.time()
    comp.time <- toc - tic
    cat("CPU: ", comp.time, "\n")

    tic <- Sys.time()
    C <- gpuMatMult(A, B)
    toc <- Sys.time()
    comp.time <- toc - tic
    cat("GPU: ", comp.time, "\n")
}
```

Figure 13. Code in R to compare the performance of gpuMatMult() and %*% [28]

Add the result is the following:

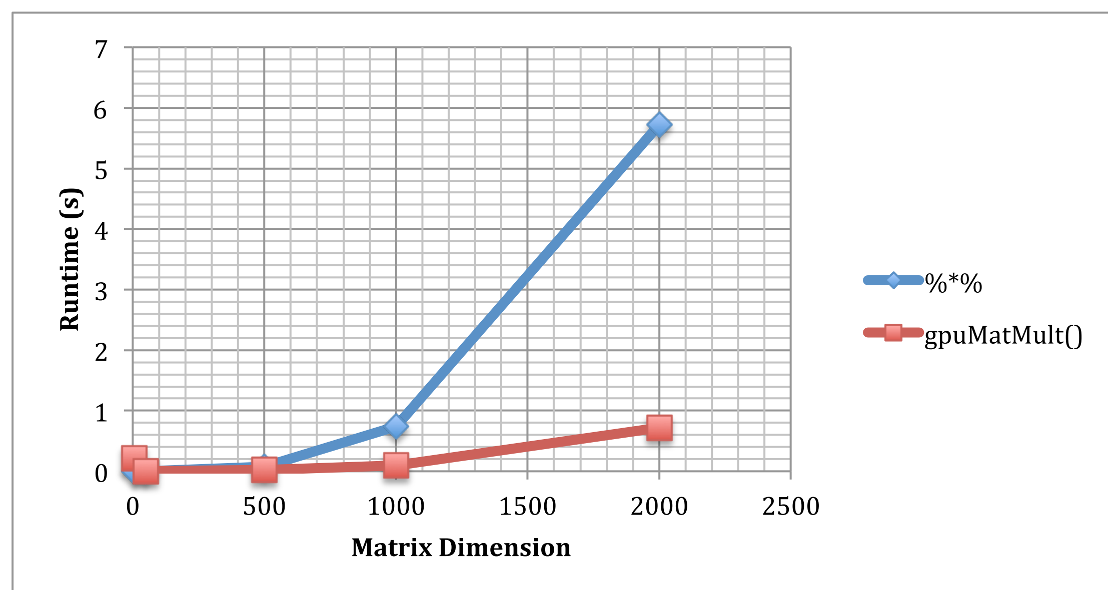

Figure 14. Runtime Comparison between %*% (CPU) and gpuMatMult() (GPU)

As we can see from Figure 14, when our data size is relatively small, gpuMatMult() runs slightly slower than %*%, but both take a very short time to complete the task. This has to do with the mechanism of having to copy data between the host and the device both before and after the computation. When data size is small, it takes more time to do memory copying than to do the actual computation. However, when the matrix lets larger, the advantage of GPGPU begins to show. Eventually, when the matrix size is very big, gpuMatMult() runs much faster than %*%.

5.2.2 rpuDist() versus dist()

rpuDist() is the function for calculating matrix distance in the package rpud. In order to compare its performance with dist() in R, I ran the following code in R:

```
library(rpud)
test.data <- function(dim, num, seed = 17) {
    set.seed(seed)
    matrix(rnorm(dim * num), nrow = num)
}
system.time(dist(m))
system.time(rpuDist(m))
```

And the runtime comparison is shown below in Figure 15.

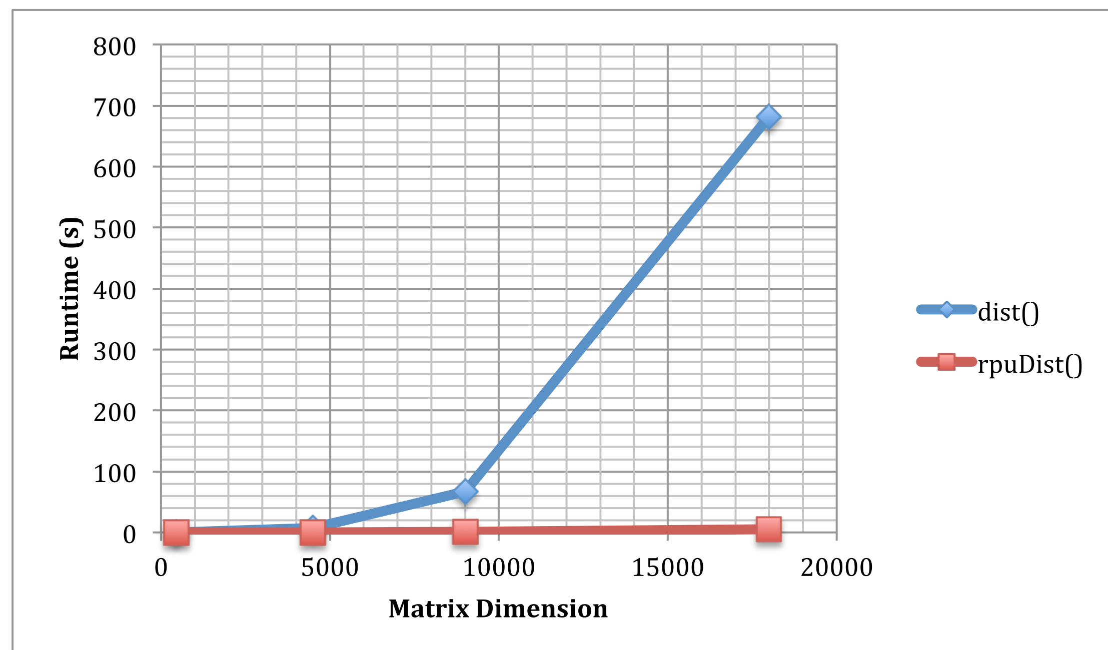

Figure 15. Runtime Comparison between dist() and rpuDist()

As we can see from Figure 15, rpuDist(), which uses CUDA to calculate distance matrix, performs much better than R's built-in function dist(), which uses CPU. It is especially noticeable that the performance of rpuDist() is more than 100 times better than the performance of dist() when the dimension of the matrix became as large as 18000.

5.3 GRRNG (GPU-R Random Number Generator) Package

In addition to installing and testing CUDA extensions developed by others, I also implemented my own CUDA extension for R. GRRNG is developed based on libraries cuRAND and THRUST. cuRAND is a C library that uses CUDA for generating high-quality pseudorandom and quasirandom numbers efficiently [35]. THRUST is a C++ library that provides a rich collection of data parallel algorithms such as scan, sort, and some useful data structure for parallelism [34].

GRRNG contains the following R functions:

- gpuRnorm – GPU accelerated rnorm() function, for generating random numbers from normal distribution
- gpuRlnorm – GPU accelerated rlnorm() function, for generating random numbers from log-normal distribution
- gpuRpois – GPU accelerated rpois() function, for generating random numbers from Poisson distribution
- gpuRunif – GPU accelerated runif() function, for generating random numbers from uniform distribution
- gpuRbinom – GPU accelerated rbinom() function, for generating a sequence of random integers from a binomial distribution
- gpuRsample – GPU accelerated sample() function for uniformly distributed probability
- gpuRsort – GPU accelerated sort() function
- gpuSetSeed – Set the random seed for GPU

The package GRRNG contains the source code for making CUDA calls in C/C++, the wrapper functions in R and a Makefile for generating the .so shared library. My code

for gpuRnorm() is shown in Appendix II as an example.

In statistics, we often need massive amount of randomly generated data for us to conduct test on, and simulation is almost always necessary. In 2006 September issue of the Journal of the American Statistical Association, there are 25 theory and methods articles, 16 of them used simulation [36]. The functions provided by GRRNG are essential for using CUDA in R simulation, and we'll discuss their advantage in performance when compared to their CPU version in Section 5.4.

Installation for GRRNG is easy. If you have a CUDA supported GPU with CUDA properly installed on your machine, simply download GRRNG_<version>.tar.gz from http://users.wpi.edu/~zheyangwu/, set the basic environment variables, and run `R CMD INSTALL GRRNG_<version>.tar.gz`. Afterwards, all you need to do is to include `library(GRRNG)` in your R scripts.

5.4 Performance Advantage of Using GRRNG

I tested a few functions in my GRRNG package with different size of data and compared the performance with their corresponding CPU-based R functions. Here I chose three of them as examples to demonstrate their performance.

5.4.1 gpuRnorm() versus rnorm()

rnorm(n, mean = 0, sd = 1) is the function for generating n random numbers from a normal distribution with the given mean and standard deviation. gpuRnorm() is the version of rnorm() in my GRRNG, which uses C library cuRAND to make CUDA calls to accelerate the random number generation process. I called both rnorm() and gpuRnorm() with an increasing n, and the result is shown in Figure 16.

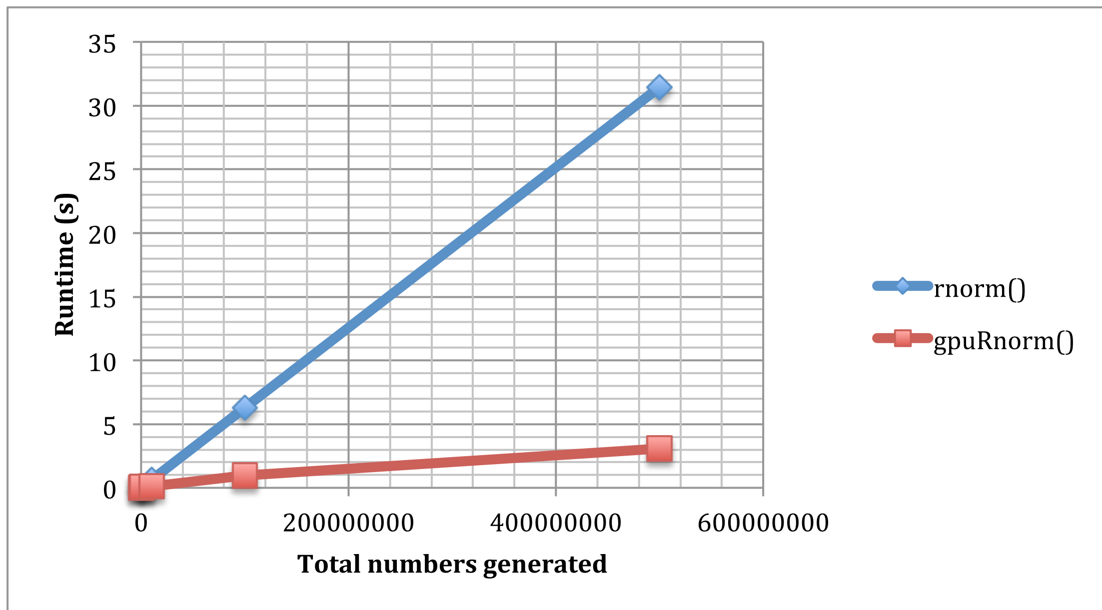

Figure 16. Runtime Comparison between rnorm() and gpuRnorm()

As we can see above, when n is big, gpuRnorm() runs much faster than rnorm(). When n=500000000, it took rnorm() 31.45 seconds to run but gpuRnorm() only used 3.058 seconds! This is 10 times faster.

5.4.2 gpuRunif() versus runif()

runif(n, min = 0, max = 1) generates n random numbers from the uniform distribution on the interval from min to max. Its corresponding function in GRRNG is gpuRunif(). Similarly, I recorded and compared their execution time for different n's, and the result is demonstrated in Figure 17 below. Similarly, we found out the runtime performance of gpuRunif() is a lot better than runif() on large data sets.

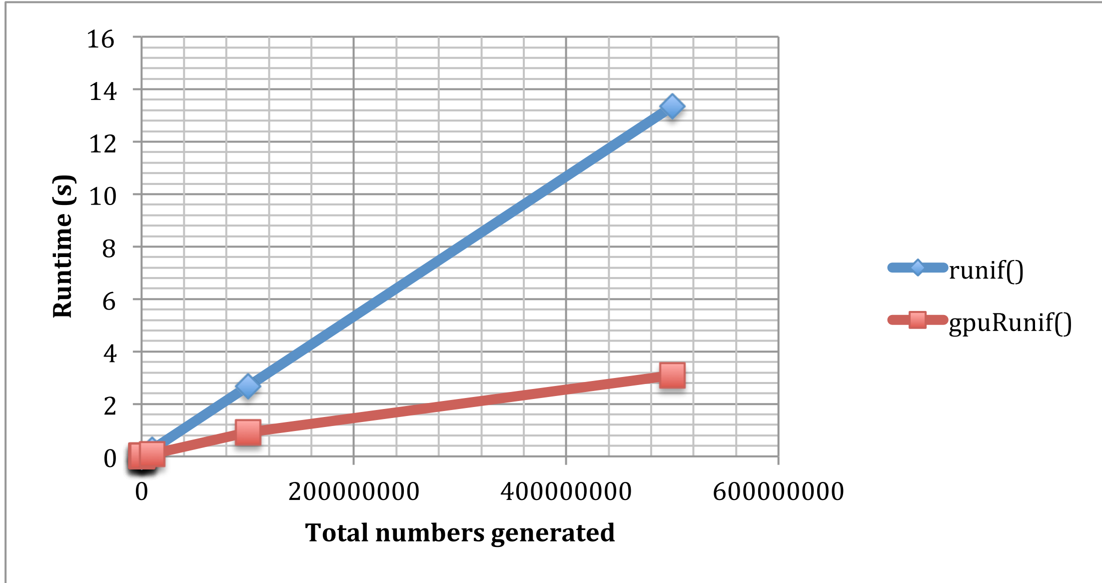

Figure 17. Runtime Comparison between runif() and gpuRunif()

5.4.3 gpuRsort() and sort()

gpuRsort() is the GPU accelerated version of sort() in R. It is build based on the sort() call in the THRUST library. I ran both functions on different sets of random numbers with varying sizes, and their runtime performance comparison is shown below in Figure 18. We can see that gpuSort() also outperforms sort() on a large data set.

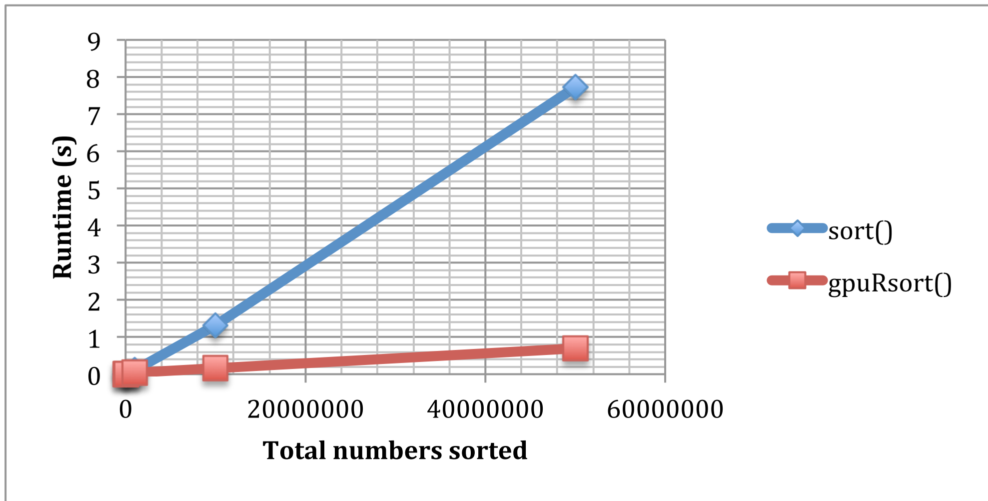

Figure 18. Runtime Comparison between sort() and gpurRsort()

5.5 GpuRInterface Package

I also wrote a simple interface package in R to make calling functions in CUDA extensions easier. This interface is essentially a parser that parses function name as an argument, and calls the GPU version of that function as long as the related CUDA extension is installed. For example, if we want to call the GPU version of the rnorm() function with n=10000, all you need to do is to import GpuRInterface using `library(GpuRInterface)` and call `gpuCallFunc("rnorm", 10000)`, then GpuRInterface will call gpuRnorm(10000) in the GRRNG package for you and return the result. This interface helps users call functions in the installed CUDA extensions easily without having to remember the corresponding functions names in those extensions. This package will also be available to download on http://users.wpi.edu/~zheyangwu/.

5.6 GRGDS (GPU-R Genetic Data Simulator) Package

I adapted some functions in the R Genetic Data Simulation library provided by Professor Wu using functions in GRRNG. These functions are wrapped up to build a new R package GRGDS (GPU-R Genetic Data Simulator). For example, in Simulation_Quanti_BinaryTraits.R, I improved the function genYMarks() by changing the line `epsilon <- rnorm(numb, sd=errorStd)` to `epsilon <- gpuCallFunc("rnorm", numb, sd=errorStd)`, which calls `gpuRnorm(numb, sd=errorStd)`. Then I tested genYMark() on a 100000000*10 matrix and the result is shown below:

```
> system.time(genYMarks(A, B))
   user  system elapsed 
  0.992   0.476   1.468 
> system.time(genYMarks2(A, B))
   user  system elapsed 
  6.705   0.358   7.058 
```

genYMarks() uses gpuRnorm() and genYMarks2() is the original version. As we can see, the improved version of genYMarks() runs almost seven times faster than the original version. This implies that the Simulations Package can be largely benefited from GRRNG as well as other CUDA based R extensions. The R package is built up

following the standard procedure at http://cran.r-project.org/doc/manuals/R-exts.html.

5.7 Interface between Variant Tools and GPU-based R function

We want an interface that connects the functionality of Variant Tools (e.g., data management and annotation) to the GPU-powered R functions for realizing the computational needs in genetic data analysis (e.g., the association tests). We call this interface R function gpuCallRFunc, for which we demonstrate the coding idea as below:

```
library(GpuRInterface)
# BEGINCONF
# [sample.size]
# [result]
# n=2
# columns=2
# name=beta0, beta1
# column_name=estimate, p.value
# ENDCONF
gpuCallRFunc = function (dat, phenotype.name, family = "gaussian",
funcName="glm", ...) {
  y = dat@Y[, phenotype.name]
  x = apply(dat@X, 1, function(i) sum(i, na.rm=T))
  ...
  m = call(funcName, y~x, family=family)
  ...
  return (list(sample.size=length(y), result=summary(m)$coef[,c(1,4)]))
}
```

The corresponding Variant Tools command calling this R interface is something like `vtools associate rare status -m 'RTest gpuCallRFunc.R --name demo --phenotype.name "age" --family "binomial"' --... -j8 -g variant.chr --to_db demo > demo.txt`. Here ... represents the options required by the specific engine function that drives the specific data analysis procedure.

The interface is to be further complete and tested. And the final version will be available online at users.wpi.edu/~zheyangwu/. We have tested some examples (not provided here) and did significantly improve the computation performance for Variant Tools by replacing CPU based R function by the corresponding GPU accelerated R function.

5.8 Analysis of performance of GPGPU

As we can see from the runtime comparison between functions implemented using GPGPU and functions that do not, GPGPU's parallel data processing is very efficient especially when is data set is large. However, many times we count on libraries written by others and most of these libraries are not GPGPU oriented. So, it is hard to convert our code to use GPGPU as much as we can. But the advantages of fully utilizing the power of GPGPU are rather obvious. Therefore, even though the shifting from using CPU for computing large data to using GPU for these computations will be not an easy process, it is still worth our effort.

6 Future Work

The GRRNG library contains eight functions in total at the moment, but there are many more that can be added to it. Here is a short list of R functions that are related to simulation that can be converted to their GPU version [36]:

- rbeta (for the beta random variable)
- rexp (for the exponential random variable)
- rf (for the F random variable)
- rgamma (for the gamma random variable)
- rgeom (for the geometric random variable)
- rhyper (for the hypergeometric random variable)
- rlogis (for the logistic random variable)
- rmvbin (for the multivariate binary random variable)
- rnbinom (for the negative binomial random variable)
- rweibull (for the weibull random variable)

Also, we can submit both the GRRNG package and the GpuRInterface package to CRAN so that we can make them open to more users and more developers. Opening its source can attract more contributors and granting more people with access to them can help more users benefit from GPGPU.

In the future, we can write even more R functions using CUDA to replace as many commonly used functions as we can. We know in our own field of research what are the most commonly used functions, so simply convert these functions using CUDA and create their wrappers in R, then we can always reference them in the future. Also, we can help write open source R libraries for CUDA by adding other useful routines to the packages.

On the other hand, we can start to replace some old code in the existing R source with calls to the new functions in the CUDA extensions. Although this process is tedious, the improvement in runtime performance will be unimaginable. Also, while writing R code, we need to keep in mind that we should always call functions in the CUDA extensions as opposed to their CPU version, when possible.

Last but not the least, testing of functions that are fully or partly rewritten with CUDA extensions can be done at a larger scale. We should not only test more functions, but also test them with real data – Find some existing R code of ours, replace as many functions with their CUDA version as we can, and then run the R code on a large biological data set and compare the performances.

7 References

1 NVIDIA. "Graphics Processing Unit (GPU)." *NVIDIA*. NVIDIA Corporation, 31 Aug. 1999. Web. 04 May 2014. <http://www.nvidia.com/object/gpu.html>.

2 Tyson, Jeff, and Tracy V. Wilson. "How Graphics Cards Work." HowStuffWorks. HowStuffWorks, Inc, n.d. Web. 04 May 2014. <http://computer.howstuffworks.com/graphics-card1.htm>.

3 Krewell, Kevin. "WHAT'S THE DIFFERENCE BETWEEN A CPU AND A GPU?" *NVIDIA*. NVIDIA Corporation, 16 Dec. 2009. Web. 04 May 2014. <http://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/>.

4 Zahran, Mohamed. "History of GPU Computing." Cs.nyu.edu. Web. 4 May 2014. <http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture2.pdf>.

5 NVIDIA. "About CUDA." NVIDIA Developer Zone. NVIDIA Corporation, n.d. Web. 04 May 2014. <https://developer.nvidia.com/about-cuda>.

6 NVIDIA. "Bioinformatics and Life Sciences." NVIDIA. NVIDIA Corporation, n.d. Web. 04 May 2014. <http://www.nvidia.com/object/bio_info_life_sciences.html>.

7 Kirk, David, and Wen-mei Hwu. "2.1 Evolution of Graphics Pipelines." Programming Massively Parallel Processors: A Hands-on Approach. 2nd ed. Burlington, MA: Morgan Kaufmann, 2010. N. pag. Print.

8 Peng, Bo. "Integrated Annotation and Analysis of Genetic Variants from Next-generation Sequencing Studies with Variant Tools." The University of Texas MD Anderson Cancer Center. 03 Oct. 2013. Web.

9 Houston, Mike. "General Purpose Computation on Graphics Processors (GPGPU)." Http://graphics.stanford.edu/. Web. 4 May 2014. <http://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf>.

10 Kerzner, Ethan, and Timothy Urness. *GPU Programming for Mathematical and Scientific Computing*. Rep. N.p.: n.p., n.d. <http://artsci.drake.edu/urness/download/MICS2010Kerzner.pdf>.

11 "The R Project for Statistical Computing." *The R Project for Statistical Computing*. N.p., n.d. Web. 03 May 2014. <http://www.r-project.org/>.

12 Kou, Qiang. "Talk about R and GPU." Capital Os Statistics. N.p., 07 Oct. 2013. Web. 04 May 2014. <http://cos.name/2013/10/gossip-r-gpu/>.

13 Yang, Hangjun. "CUDA Programming." Web log post. Yanghangjun's Blog. N.p., 10 Dec. 2010. Web. 04 May 2014. <http://blog.csdn.net/yanghangjun/article/details/6067534>.

14 NVIDIA. "NVIDIA CUDA Library Documentation." *NVIDIA CUDA Library*. NVIDIA Corporation, n.d. Web. 04 May 2014. <http://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/index.html>.

15 Blay, Sigal. *Calling C Code from R, an Introduction*. Tech. N.p.: n.p., 2004. Print.<http://www.sfu.ca/~sblay/R-C-interface.txt>

16 Eddelbuettel, Dirk. "Overview: Rcpp: Seamless R and C++ Integration."*Rcpp: Seamless R and C++ Integration*. N.p., 14 Mar. 2014. Web. 04 May 2014. <http://dirk.eddelbuettel.com/code/rcpp.html>.

17 Eddelbuettel, Dirk. "Overview: RInside: Easier Embedding of R in C++ Applications." *RInside: Easier Embedding of R in C++ Applications*. N.p., 05 Dec. 2012. Web. 04 May 2014. <http://dirk.eddelbuettel.com/code/rinside.html>.

18 University of Michigan. "R+GPU." R+GPU. University of Michigan, n.d. Web. 04 May 2014. <http://brainarray.mbni.med.umich.edu/brainarray/rgpgpu/>.

19 Yau, Chi. "GPU Computing with R." R Tutorial: An R Introduction to Statistics. N.p., n.d. Web. 04 May 2014. <http://www.r-tutor.com/gpu-computing>.

20 Buckner, Josh, Mark Seligman, and Justin Wilson. *Package 'gputools'*. N.p.: n.p., 09 May 2013. PDF.

21 Nash, Peter, and Vendel Szeremi. Package 'HiPLARM'. N.p.: n.p., 18 Oct. 2012. PDF.

22 Yau, Chi. *Package 'rpud'*. N.p.: n.p., 15 Sept. 2010. PDF.

23 Smith, Brian J. *Package 'magma'*. N.p.: n.p., 03 Apr. 2013. PDF.

24 Eddelbuettel, Dirk. *Package 'gcbd'*. N.p.: n.p., 12 Dec. 2013. PDF.

25 Seligman, Mark, and Chris Fraley. *Package 'WideLM'*. N.p.: n.p., 17 Feb. 2012. PDF.

26 Da Silva, Adelino Ferreira. Package 'cudaBayesreg'. N.p.: n.p., 03 Mar. 2014. PDF.

27 Shterev, Ivo D., Sin-Ho Jung, Stephen L. George, and Kouros Owzar.*Package 'permGPU'*. N.p.: n.p., 27 Dec. 2012. PDF.

28 "R and GPU." Statistical Laboratory. Statistical Laboratory, n.d. Web. 04 May 2014. <http://statlab.nchc.org.tw/rnotes/?page_id=163>.

29 F. Anthony San Lucas, Gao Wang, Paul Scheet, and Bo Peng (2012) Integrated annotation and analysis of genetic variants from next-generation sequencing studies with variant tools, Bioinformatics 28 (3): 421-422.

30 Elaine r Mardis ET Al. The 1, 000 Genome, The 100,000 Analysis. Genome Med, 2(11):84, 2010; N. Siva. 1000 Genomes Project. Nature Biotechnology, 26(3):256–256, 2008

31 Gibran Hemani, Athanasios Theocharidis, Wenhua Wei, and Chris Haley. EPIGPU: Exhaustive Pairwise Epistasis Scans Parallelized on Consumer Level Graphics Cards. Bioinformatics, 27(11):1462–1465, 2011

32 Etong. "CUDA: Leading GPGPU Revolution." *Expreview*. N.p., 24 Oct. 2008. Web. 4 May 2014. <http://www.expreview.com/5240-all.html>.

33 Simpson, Matt. "Writing CUDA C Extensions for R." *Matt Simpson*. N.p., 29 June 2012. Web. 04 May 2014. <http://www.themattsimpson.com/2012/06/29/writing-cuda-c-extensions-for-r/>.

34 NVIDIA. "Thrust :: CUDA Toolkit Documentation." *NVIDIA Developer Zone*. NVIDIA Corporation, n.d. Web. 04 May 2014. <http://docs.nvidia.com/cuda/thrust/#introduction>.

35 NVIDIA. "CuRAND :: CUDA Toolkit Documentation." *NVIDIA Developer Zone*. NVIDIA Corporation, n.d. Web. 04 May 2014. <http://docs.nvidia.com/cuda/curand/introduction.html#introduction>.

36 "Lesson 4: Simulation Using R." *Lesson1n*. N.p., 30 Nov. 1999. Web. 04 May 2014. <http://www.esg.montana.edu/R/yang4.htm>.

Appendix I: List of Functions in CUDA Extensions for R

- gputools [20]

gpuCor()：Calculate correlation coefficient

gpuDist()：Calculate distance matrix

gpuDistClust()：Hierarchical clustering

gpuFastICA()：Independent Component Analysis

gpuGlm()：GPU accelerated glm()

gpuGranger()：Perform Granger Causality Tests for Vectors on a GPU

gpuHclust()：GPU accelerated hclust()

gpuLm()：GPU accelerated lm()

gpuLsfit()：GPU accelerated lsfit()

gpuMatMult()：Matrix multiplication

gpuMi()：B spline based mutual information

gpuQr()：Estimate the QR decomposition for a matrix

gpuSolve()：GPU accelerated solve()

gpuSvd()：GPU accelerated svd()

gpuSvmPredict()：A support vector machine style binary classifier

gpuSvmTrain()：Train a support vector machine on a data set

gpuTtest()：T-Test Estimator with a GPU

- HiPLARM [21]

checkFile: Startup function that reads in the optimised crossover points

chol: Cholesky Decomposition using GPU or multi-core CPU

chol2inv-methods: Inverse from Cholesky

crossprod: Crossproduct using GPU or multi-core CPU

determinant: Calculate the determinant using GPU and multi-core CPU

hiplarSet: Methods for Function hiplarSet in Package HiPLARM

hiplarShow: Shows the crossover points for all functions

lu: (Generalized) Triangular Decomposition of a Matrix

norm: Matrix Norms

OptimiseAll: Optimise all given routines

OptimiseChol: Optimise the chol routine for dpo Matrices

OptimisecrossprodDge: Optimise the crossprod routine for a single dge matrix

OptimisecrossprodDgeDge: Optimise the crossprod routine for two dge matrices

OptimisecrossprodDgemat: Optimise the crossprod routine for a dge matrix and an R base matrix

OptimisematmulDtrDtr: Optimise the matmul routine for two dtr matrices

OptimisematmulDtrmat: Optimise the matmul routine for a dtr matrix and an R base matrix

OptimisenormDge: Optimise the norm routine for a dge matrix

OptimisercondDge: Optimise the rcond routine for a dge matrix

OptimisercondDpo: Optimise the rcond routine for a dpo matrix

OptimiseSolveDge: Optimise the solve routine for a dge matrix

OptimiseSolveDgemat: Optimise the solve routine for a dge matrix and an R base matrix

OptimiseSolveDpo: Optimise the solve routine for a dpo matrix

OptimiseSolveDpomat: Optimise the solve routine for a dpo matrix and an R base matrix

OptimiseSolveDtr: Optimise the solve routine for a dtr matrix

OptimiseSolveDtrmat: Optimise the solve routine for a dtr matrix and an R base matrix

rcond: Estimate the Reciprocal Condition Number using GPU and multi-core CPU

solve: Solve a linear system Ax=b using GPU or multi-core architectures

tcrossprod: tcrossproduct using GPU or multi-core CPU

%*% : Matrix Multiplication of two matrices using GPU or multi-core architectures

- Rpud [22]

rpuDist: Compute the distance matrix with GPU

plot.rpusvm: Plot an SVM Model Trained by rpusvm

plot.rvbm: Diagnostics Plots for Variational Bayesian Multiclass Probit Regression

predict.rpusvm: Predict Method for Support Vector Machines

predict.rvbm: Predict Method for Variational Bayesian Multiclass Probit Regression

read.svm.data: Reading SVM Training Data from a File

rhierLinearModel: Gibbs Sampler for Hierarchical Linear Model

rhierMnlRwMixture: MCMC Algorithm for Hierarchical Multinomial Logit with Mixture of Normals Heterogeneity

rmultireg: Draw from the Posterior of a Multivariate Regression

rpuchol: GPU Accelerated Cholesky Decomposition

rpucor: Kendall’s Tau-b Correlation Coefficient on GPU

rpucor.test: Compute the p-values of the correlation matrix

rpuDist: Compute the Distance Matrix on GPU

rpuGetDevice: Id of the GPU device in use by the active host thread

rpuHclust: Hierarchical Clustering

rpuScale: Scaling SVM Training Data

rpuSetDevice: Select GPU for use by the current thread

rpusvm Support: Vector Machines on GPU

rvbm: GPU Accelerated Variational Bayesian Multiclass Probit Regression

rvbm.sample.train: Example Data Sets for Variational Bayesian Multiclass Probit Regression

summary.rvbm: Summary Statistics of Variational Bayesian Multiclass Probit Regression

- Magma [23]

Backsolve: Solve an Upper or Lower Triangular System

Lu: The LU Decomposition

Magma: Matrix Class "magma" Constructor

magma-class: Matrix Class "magma", including a sets of methods

magmaQR-class: Class "magmaQR", including a set of methods

magmaLU-class: Class "magmaLU", including a set of methods

- gcbd [24]

analysis: Analysis functions for GPU/CPU Benchmarking

benchmark: Benchmarking functions for GPU/CPU Benchmarking

figures: Figures from the corresponding vignette

utilities: Utility functions for GPU/CPU Benchmarking

- WideLM [25]

wideLM: Fitting Multiple Models of Modest Size

- cudaBayesreg [26]

buildzstat.volume: Build a Posterior Probability Map (PPM) NIFTI volume

cudaMultireg.slice: CUDA Parallel Implementation of a Bayesian Multilevel Model for fMRI Data Analysis on a fMRI slice

cudaMultireg.volume: CUDA Parallel Implementation of a Bayesian Multilevel Model for fMRI Data Analysis on a fMRI NIFTI volume

plot.bayesm.mat: Plot Method for Arrays of MCMC Draws

plot.hcoef.post Plot Method for Hierarchical Model Coefficients

pmeans.hcoef: Posterior mean for each regression variable

post.overlay: Rendering a Posterior Probability Map (PPM) volume

post.ppm: Posterior Probability Map (PPM) image

post.randeff: Plots of the random effects distribution

post.shrinkage.mean: Computes shrinkage of fitted estimates over regressions

post.shrinkage.minmax: Computes shrinkage of fitted estimates over regressions

post.simul.betadraw: Postprocessing of MCMC simulation

post.simul.hist: Histogram of the posterior distribution of a regression coefficient

post.tseries: Show fitted time series of active voxel

premask: Mask out voxels with constant time-series

read.fmrislice: Read fMRI data

read.Zsegslice Read brain segmented data based on structural regions for CSF, gray, and white matter.

Regpostsim: Estimation of voxel activations

- permGPU [27]

permgpu: Conduct permutation resampling analysis using permGPU

scoregpu: Computes score test statistic using permGPU

test.permgpu: Conduct permutation resampling analysis using permGPU

test.scoregpu: Conduct permutation resampling analysis using permGPU

Appendix II: Source Code for gpuRnorm()

Code in rnorm.cu:

```
#include <R.h>
#include <cuda.h>
#include <curand.h>

extern "C" void gpuRnorm (double *nums, int *seed, double *mu, double *sd, double
*hostData) {
    size_t num = (size_t) (*nums);
    size_t n;
    curandGenerator_t gen;
    double *devData;

    // TODO Change this hardcoded 100000000 to be MAX_SIZE_T/sizeof(double)
    while (1) {
        if (num > 100000000) {
            n = 100000000;
        } else {
            n = num;
        }
        cudaMalloc((void **)&devData, n * sizeof(double));
        curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
        curandSetPseudoRandomGeneratorSeed(gen, *seed);
        curandGenerateNormalDouble(gen, devData, n, *mu, *sd);
        cudaMemcpy(hostData, devData, n * sizeof(double),
cudaMemcpyDeviceToHost);
        curandDestroyGenerator(gen);
        cudaFree(devData);
        if (num > 100000000) {
            num = num - 100000000;
            hostData = hostData + 100000000;
        } else {
            break;
        }
    }
}
```

Code in rnorm.R:

```
gpuRnorm <- function(n, mu, sd){
    if(! is.loaded("gpuRnorm"))
        dyn.load("rnorm.so")

    a <- 1:n

    if (! exists("gpuSeed"))
        gpuSeed <- 0

    if (missing(mu) || missing(sd)) {
        mu <- 0
        sd <- 1
    }

    out <- .C("gpuRnorm", as.double(n), as.integer(gpuSeed), as.double(mu),
as.double(sd), hData = as.double(a))

    return(out$hData)
}
```