Michigan Technological University
Digital Commons @ Michigan Tech

Dissertations, Master's Theses and Master's Reports - Open

Dissertations, Master's Theses and Master's Reports

2013

EFFECTIVE FUNCTION CHOICE IN THE R SCRIPTING LANGUAGE

Trevor D. Fisher
Michigan Technological University

Follow this and additional works at: https://digitalcommons.mtu.edu/etds

Copyright 2013 Trevor D. Fisher

Recommended Citation
Fisher, Trevor D., "EFFECTIVE FUNCTION CHOICE IN THE R SCRIPTING LANGUAGE", Master's report, Michigan Technological University, 2013.
https://digitalcommons.mtu.edu/etds/663

Follow this and additional works at: https://digitalcommons.mtu.edu/etds

EFFECTIVE FUNCTION CHOICE IN THE R SCRIPTING LANGUAGE

By

Trevor D. Fisher

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2013

©2013 Trevor D. Fisher

This report has been approved in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Report Adviser: *Dr. Zhenlin Wang*

Committee Member: *Dr. Steven Carr*

Committee Member: *Dr. Saeid Nooshabadi*

Committee Member: *Dr. Qiuying Sha*

Department Chair: *Dr. Charles Wallace*

To my mother, father, and little brother. Thank you for all of you support.

Abstract

This project examines the current available work on the explicit and implicit parallelization of the R scripting language and reports on experimental findings for the development of a model for predicting effective points for automatic parallelization to be performed, based upon input data sizes and function complexity. After finding or creating a series of custom benchmarks, an interval based on data size and time complexity where replacement becomes a viable option was found; specifically between $O(N)$ and $O(N^3)$ exclusive. As data size increases, the benefits of parallel processing become more apparent and a point is reached where those benefits outweigh the costs in memory transfer time. Based on our observations, this point can be predicted with a fair ammount of accuracy using regression on a sample of approximately ten data sizes spread evenly between a system determined minimum and maximum size.

Chapter 1

Introduction

Due to its simplicity and ease of use, R has become a major tool in the fields of data mining and scientific data processing. However, this simplicity comes at a cost in performance. While R may be easy to learn and implement, and have heavily optimized built in functions, its execution speed can be limited as function complexity increases. The results of this are simulations and analysis that take days instead of hours or minutes.

Current work on optimizing R through parallelization follows one of two paths. The first is to use a fine grained, explicit approach, and parallelize specific functions for execution in multicore CPU and/or GPU environments. Multiple libraries, and projects such as R+GPU [7] and PLASMA/MAGMA [8] [9] focus on implementing parallel versions of already available or common R functions in order to achieve higher performance. With this method, individual function calls, which can take tremendous amounts of time can be manually replaced with parallelized versions that can perform the same calculations at a significant speedup. Overall, the script is executed sequentially and individual functions take advantage of parallelization.

The advantage of this approach is that the individual functions can be fine tuned to get the maximum use out of the hardware that they are being run on. The disadvantage is the narrow focus of the individual projects. Only individual functions are parallelized, and special function calls are needed. This means that in order to take advantage of possible parallelization, researchers need to go through the process of modifying their code to use the new functions. Along with this, there is also a lack of flexibility. If a parallel version of a function is not available researchers either have to write their own or are out of luck.

The second path focuses on coarse grained implicit parallelization. Rather than explicitly parallelize individual functions, projects such as pR [12] focus on automatically parallelizing entire scripts through limited applications of compiler analysis and optimization, taking advantage of R's simplicity which allows for some analysis that is not possible using more complex languages such as C. Other projects such as fork [10] and Rdsm [11] modify R to add MPI and multithreading directly into the language.

The upside of this implicit approach is that the parallelization is done automatically or with the inclusion of a few new keywords. Rather than using parallelized versions of specific functions, researchers can parallelize their scripts automatically, possibly requiring only the inclusion of a set of directives in the code for the compiler. This allows for much more flexibility if explicitly parallelized functions are not already available. The disadvantage of this is that automatic parallelization is still a complicated subject, and while this process does not require researchers to rewrite their code in the same way as explicit parallelization, it cannot produce the same level of performance that an experienced programmer can.

Chapter 2

Goals

The goal of this project is to develop a model of R function execution time based on data size, transfer time, and algorithm complexity that will allow for the effective choice of when to replace a sequential R function with a parallel counterpart.

Chapter 3

Background

R is an open source implementation of the S programming language developed for statistical data processing. It specializes in data manipulation and display, with many built in graphics and data analysis functions. Along with this, it is easily expanded with custom code packages available either on the Comprehensive R Archive Network (CRAN) [1] or through individual development. Packages allow R users to develop and make available code for functions and actions that are not part of the standard R package and consist of R scripts and external code accessed through a built in interface for several languages including C++ and FORTRAN.

R is an interpreted language and is designed to operate efficiently on vectors and matrices. R being an interpreted language means that rather than having to compile an R script before running it, the script is simply fed into an interpreter that automatically converts it to bit code for execution. The advantage of this is that scripts can be tested and run as they are being developed without the need for recompilation. The downside is that being interpreted introduces bottlenecks to execution time and speed. For many basic functions, the back end code is written in heavily optimized C and FORTRAN, however functions developed externally are not guaranteed to be so. R's use of vector and matrix operations also allows for significant code simplification through the avoidance of loops and the use of optimized code.

R has gained popularity as a statistical programming and calculating environment due to this ease of use and expansion. It allows for a fairly inexperienced person to create and use complicated functions in a simple manner and comes with many built in options for the graphical display of data. Actions that can take tens of lines of code in C can be performed with one simple line in R, and if the R base package does not contain a required function, the CRAN contains hundreds of external packages that could contain the necessary function.

One of the many areas of work on R is in parallel processing. While certain R functions have been optimized to run as efficiently as possible on a standard processor, it has no built in options for parallelization. This is where several external packages come into play. Available on the Comprehensive R Archive Network [2], there are

several packages that add external parallel capability to the R language. Located in the CRAN High-Performance and Parallel Computing in R section of the main CRAN page, these packages range from individual functions that have been implemented using an external parallel framework to expansions to the language that allow for explicit parallelization. While there are multiple ways to achieve parallelism, this project focused on the use of a GPU as a parallel processor. It is also possible to achieve parallel execution on the CPU, but a study of that fell outside of the time scope of this project.

Beginning in the 1980s, graphics processing units were introduced as a specialized addition to computers designed specifically to handle the calculations involved in graphical tasks. Driven by gamer's demand for more graphical power, each generation of GPUs were more powerful than the last. Finally, in 2006, NVIDIA recognized that GPUs had essentially become large scale parallel processors that pretty much anyone had access to and decided to capitalize on this by introducing CUDA and shifting GPU design to a form more suitable for parallel code execution along with graphical tasks.

Parallel code for execution on an NVIDIA GPU is written using CUDA C, parallel computing platform based upon C. Each CUDA program is divided into two separate parts. Host code which is executed on the CPU of the computer and device code which is executed on the GPU. The host code is responsible for setting up and initializing the CUDA environment, doing things such as determining what resources are available, transferring data to and from the GPU for processing, and calling the device code to get everything started. The device code is executed on the GPU and consists of a kernel and any functions written to be called from within the kernel. CUDA follows the single instruction, multiple data method of parallelism. This means that parallelism is achieved by executing the same piece of code in parallel with the only difference between individual executions being the piece of data being worked on.

Individual pieces of work are carried out by threads, which as explained above each execute the same instruction on a unique piece of data. Threads are grouped into blocks that can be one, two, or three dimensional depending on the needs of the program, and blocks are in turn divided into a grid that can also be one, two, or three dimensional. The number of threads per block and blocks per grid is specified by the programmer in the host code, allowing for control of the exact execution conditions of a given kernel.

Chapter 4

Related Works

As mentioned before, the parallelization of R can be split into two methods, fine grained parallelization where individual functions are explicitly parallelized and coarse grained parallelization where R scripts themselves are parallelized manually or automatically. Currently, the main focus of the work on parallelizing R is on actual parallelization, either explicitly or via script analysis.

"Automatic Parallelization of Scripting Languages: Toward Transparent Desktop Parallel Computing" [12] is one of the earliest works on the subject of automatically analyzing and parallelizing R. The paper describes how the recent increase in the amount of data being analyzed has created a need for transparent parallelization in R and other scripting languages in order to handle analysis in a reasonable amount of time. It puts forward a system called pR, which dynamically analyzes R script code and automatically parallelizes portions of it for execution on multiprocessor environments. As of the time of this publication, pR was more theoretical and was limited to some function calls and for/while loops. Overall, a speedup of up to 25x over base execution time of an R script was shown.

"Transparent Runtime Parallelization of the R Scripting Language" [13] continues where the previous work left off. It describes the more recent progress on the pR project, going into more detail and depth about what is possible and what has been done than the last paper. While pR is more fleshed out in this paper, it still produces a speedup of 25x at the best, as in the last paper. However, it still compares positively to the results of similar projects such as the snow (Simple Network of Workstations for R) package which allows for parallelization using a network of two workstations, as its name suggests.

Finally, "Efficient Statistical Computing on Multicore and MultiGPU Systems" [14] provides a view of current research into more fine grained parallelization. In this case, rather than parallelizing the entire script, individual functions are explicitly parallelized to execute on multicore CPUs and GPUs; specifically the Pearson correlation coefficient , the chi-squared distribution, and the unary linear regression model. The user is given with the same standard R functions to use in the script, but

when executed the parallel version of the function is used rather than the standard one. This provided varying speedups from 2x to 30x for those individual functions depending on the data sizes and execution location.

Chapter 5

Methods

5.1 Coding

Data was gathered from seven functions; Sum, Squaresum, Cubesum, Vector Addition, Vector Sorting, Matrix Multiplication, and Matrix Transposition. These seven functions were chosen for their simplicity of implementation and diversity of time complexity. Each function was implemented in R as a function and a compiled bytecode function, in C or C++ using Rcpp and/or the .Call function, and in CUDA either by hand, or as an adaption of previously existing code. It was decided that wherever possible, preexisting optimized code would be used in order to achieve the best results. To that end, multiple sources and libraries were used, ranging from R packages to the CUDA SDK provided by NVIDIA. Each non-native function is written in either CUDA or C++ and integrated into R using either Rcpp or the .Call function.

Integrating outside code into an R function can be done in several ways. In this case the Rcpp and RcppArmadillo packages were used, along with the .Call function wherever necessary. .Call is a built in R function designed as a way to call external code precompiled into a shared object file from R. It is one of the most basic ways to call external functions from R and comes with only the minimum amount of support. It is up to the programmer to handle everything and make sure there are no problems. It works by simply loading an externally compiled shared object file (.so) and in the case of C/C++ choosing the desired function and passing it any arguments in the form of pointers to R objects. As long as the code has been correctly compiled, .Call will work. However, it is then the external code's job to correctly interpret the R objects, using functions provided by the R.h library, and carry out whatever actions are necessary. Below is an example of the use of .Call and some of the possible code modifications that were needed to handle R access.

```
# R .so initialization functions
ex_init <- function() { dyn.load("ex.so")}
```

```
ex_cln  <- function() { dyn.unload("ex.so")}

# .Call to external function
ex_cpp <- function(vec, iter) {
    # Setup code here
    .Call("ex_cpp", vec, iter, y)
    # Return code here
}

# External Code
extern "C" SEXP ex_cpp(SEXP v, SEXP ret) {
    int numElements = length(v);
    int *vec = INTEGER(v);
    double *retVal = REAL(ret);
    *retVal = 1;
    return R_NilValue;
}
```

On top is the R script for loading an external .so file into the R environment, allowing access to any functions it may include. The initializing function must be called before .Call is used to access one of the shared object's functions and the cleanup function should be called once the shared object is no longer needed. Next, is the actual R script that uses .Call. Its use is fairly simple, requiring the function name, and any arguments being passed in. Due to its simplicity, it is up to the programmer to ensure that the correct arguments are being provided. Finally comes the external code. In this case, it is C code, but calling CUDA functions follows the same pattern. In order for .Call to recognize the function in the shared object, the *extern* keyword must be used to extend the visibility of the function so that .Call can access it. Next, the function return type is set to SEXP (S-Expression), the base data type used by R, as are the function's arguments. Each SEXP is passed to the function as a pointer to a piece of data in the memory being use by R. Inside the function, it is the programmer's responsibility to use functions provided in R.h to correctly unpack the SEXP variables into ones that can be used by the C function. Finally, once any calculations are done, a value is returned. In this case, the result is stored in one of the arguments and R_NilValue is returned.

This is the method that was used to include all of the CUDA parallel function benchmarks, as Rcpp is currently not capable of handling CUDA code. The same process used in creating and calling the example C function is applied to the CUDA host code which, along with the removal of extraneous code, was one of the two usual modification needed to integrate benchmark code for use in R.

While the .Call method works and has an advantage in flexibility, it has the undesirable property of being fairly complex. Since many R users are not actual computer scientists, this put a limit on how much code could be exported to C/C++

functions. In order to simplify this, the Rcpp and RcppArmadillo packages were created. The Rcpp package serves as in intermediate between external C/C++ code and the .Call function, automating much of the code generation and compilation. Using Rcpp, a user needs only have enough of a basic understanding of C/C++ to write the function that they want.

```
# Matrix Multiplication example C code using Rcpp
cppFunction('
    NumericMatrix ex_cpp(SEXP matA, SEXP matB) {
    Rcpp::NumericMatrix a(matA);
        Rcpp::NumericMatrix b(matB);
        Rcpp::NumericMatrix ab(a.nrow(), b.ncol());

        for(int i = 0; i < a.nrow(); i++) {
            for(int j = 0; j < b.ncol(); j++) {
                ab(i,j) = 0;
                for(int k = 0; k < a.ncol(); k++) {
                    ab(i,j) += a(i,k)*b(k,j);
                }
            }
        }
        return ab;
    }'
)
```

Above is an example of the use of Rcpp. In this example, rather than having to use .Call, you simply call the *cppFunction* function of Rcpp and give it the code to be used. This code can either be in the form of a separate string stored in another R variable, or as in the case of this example, as is. The R environment treats the code between the two apostrophes as one string which is passed into cppFunction. cppFunction takes that string, and adds any necessary header files and declarations before compiling the code and performing all of the functions done manually for the .Call example. The end result is the ability to call *ex_cpp* directly from the R prompt. The programmer still needs to know a little bit about the R environment and data types, but the whole process is far simpler than that of manually using .Call.

Along with the Rcpp package, the RcppArmadillo package is available. RcppArmadillo is an addition to Rcpp that adds wrappers for the Armadillo C++ linear algebra library, a set of heavily optimized linear algebra functions. The following code example shows the use of RcppArmadillo and a second method of using Rcpp to generate a function.

```
# Example Armadillo C code. This is for matrix multiplication.
code <- '
```

```
    arma::mat A = Rcpp::as<arma::mat>(a);
    arma::mat B = Rcpp::as<arma::mat>(b);
    arma::mat C = A * B;
    return Rcpp::wrap(C);
'

# Armadillo code compilation
ex_arma <- cxxfunction(signature(a="numeric",b="numeric"), code,
                                                        plugin="RcppArmadillo")
```

Rather than providing the code directly in the function call, here the code is stored in the *code* variable. Next, *cxxFunction* is called. The example functions signature, code, and external plugins used are passed in as an argument and as before, a function is returned.

In the functions used in this project, .Call, Rcpp, and RcppArmadillo have all been used where appropriate. C/C++ code is split between Rcpp code and .Call based solely on when it was written and what the most efficient method was. As stated before, for the parallel implementations of functions written using CUDA and CUDA Thrust/CUBLAS, the .Call convention was exclusively used due to the need for manual compilation.

Along with the two external functions (C/C++ and CUDA) used for timing, two implementations of the R internal functions were tested; the base R function and the bytecode compiled function. The base R function is either one provided directly by R or written as an R script. The bytecode function is the result of running the base R function through the R compiler package. In certain cases, this compilation is reported to provide a speedup over a base R script by eliminating the need to interpret the function on the go. These two functions are fairly simple to create compared to the external functions, as shown below.

```
# R example function
ex_r <- function(vec, iter) {
    #Code Here
}

# R bytecode example function
ex_bc <- cmpfun(ex_r)
```

In the above example, the first function is a pure R function. The call to *function* creates an R function taking whatever arguments are listed in the call to *function* and assigns the resulting function to *ex_r*. The functions code goes between the two curly bracers. Next, once an R function has been created, it is a simple matter of passing that function into the *cmpFun()* function which will return the R bytecode implementation. Either function can then be called using *ex_r* and *ex_bc* respectively.

5.2 Testing Methods

Initial planning for this project called for the determination of when to substitute parallel functions to be based solely upon data size. In this case, data size is defined as the number of elements in a matrix or vector. This choice was made to simplify calculating data sizes as the actual object that stores data in R contains much more book keeping data than its C or CUDA counterpart. Given a new function to check for, the profiler would benchmark it's performance using several data sizes and use the results to predict when it would become profitable to introduce parallel functions. However, based on the achieved results, it was decided that the focus should not be solely on the data size, but also on the time complexity of the algorithm being profiled. As the initial profiling proceed, it became apparent that data size was not the sole factor in determining when the parallel code would surpass the performance of its sequential counterpart. Our example of this would be the difference between Sum and Matrix multiplication. For Sum, a simple $O(N)$ operation, the overhead associated with initializing and running a CUDA function overshadows any gains in speed to be had from parallel execution, causing the sequential function to outperform the parallel one in all tests. On the other hand, for Matrix Multiplication, an $O(N^3)$ function, the advantage of parallel execution occurs so early on that it would be practical to replace nearly every call with its parallel counterpart.

Each code option was tested using a variety of data sizes ranging from 100 elements, up to the maximum number of elements that could be used, constrained by either memory limits or time. The chosen data sizes remained consistent within individual function tests, but could not be as closely matched between different functions. As viewable in the code section, the tests all consist of a similar R script that generates a random data set for testing use. Once that data set has been generated, each benchmark is run and timed using R timing methods for a predetermined number of iterations, with the the resulting timings being averaged out.

The exact limits on number of iterations and maximum data size were determined by observing tests individually and gauging how much memory and time would be needed for each run. The full results for the various benchmarks can be found in the Data section, and all tests were run on the same system:

```
OS:             ubuntu 12.04 LTS
Memory:         11.7 GiB
Processor:      Intel Core i7 CPU 960 @ 3.20GHz x 8
OS Type:        64 bit
Graphics:       GeForce GTX 570
CUDA Version:   4.2
```

Overall, each test was carried out using an R script similar to the following example. Minor differences were necessary, but in general, each script followed the same general pattern. First, the benchmarking function checks and waits fore 20 seconds if

the caller specified a wait. This is to allow the caller to lock the system to minimize the need for system interrupts. Next, a loop is entered. Each iteration of the loop corresponds to one of the data sizes being tested. Inside each loop, a random set of data is generated and then fed to the benchmarks being tested, possibly multiple times to ensure a more accurate timing result is returned. Once the timing of each function is finished, the results are written out to a text file and the loop iterates to the next data size. A more detailed example of this can be found in the Code chapter.

As mentioned before, external code was either written and compiled into a shared object file outside of the timing script and then called, or written in the script and compiled for use when the script was initially loaded, depending on the complexity of the code and time it was written. Whether it was generated using Rcpp and R package functions, or was compiled externally has no effect on the execution. Each method results in a C function being compiled and called from inside R.

5.3 Regression and Prediction

Once testing was complete, the resulting timing data was analyzed and regression was used to approximate the time complexity of each function. In the interest of finding a minimum sample size to generate a close regression model of the time complexity, the timing data was sampled in two ways; a naive method where sample size was controlled by using only every nth element of the timing data, and a more advanced method where the samples were selected so that their values were be as evenly spread between the minimum and maximum values as possible. In order to generate a regression model of a function, a number of samples equal to the degree of the time complexity of the function are needed. Using the simple sample selection method, this minimum size is detected by checking the resulting sample size before begining the regression. For the even sample selection method, a sample size of at least two plus the degree is guaranteed to be returned.

```
# Naive sample selection. Simply taking every ith
# element of data$DATA.SIZE.
for(i in 1:(length(data$DATA.SIZE)-1)) {
  sequence <- seq(1, length(data$DATA.SIZE), i)
  d_s <- data$DATA.SIZE[sequence]
}

# Even sample selection. Complexity is the degree of the
# time complexity of the function being examined. A
# minimum of complexity samples are needed to generate
# a regression model of a function.
for(i in (length(data$DATA.SIZE)-2):complexity) {
    sequence <- sample_gen(data$DATA.SIZE, i)
```

```
    d_s <- data$DATA.SIZE[sequence]
}
```

As shown, with the naive method, a simple loop is used to sample varying numbers of elements from the data by selecting every nth element. The even sampling method is more complex, but provides a more evenly spread sample.

```
# Sample Selection Function
#  sample_gen(data_i, size): Generates a set of indices into
#                            data_i that approximate a sample
#                            with an even distribution of points
#                            between the min and max values found
#                            in data_i.
#  data_i: The set of data sizes associated with the timing results
#  size:   The number of samples to be found between the first and
#          last sample
sample_gen <- function(data_i, size) {
  # Find the minimum and maximum data sizes
  min  <- data_i[1]
  max  <- data_i[length(data_i)]

  # Calculate a step size that will result in size number of
  # samples between min and max
  step <- (max-min)/(size+1)

  # Calculate the next desired data size after the minimum
  check_val <- min+step

  # Add the first index to retval
  retval <- 1

  # Set the index used to add values to retval to 2
  index_val <-2

  # Iterate through the available data sizes
  for(u in 2:length(data_i)) {

    # As soon as the data size stored in data_i[u] is larger than
    # that found in check_val, add a new index to retval
    if(data_i[u] > check_val) {

      # Determine if u-1 or u should be added to retval based on
      # which size stored in data_i is closer to the desired size
```

```
      # stored in check_val
      if(abs(data_i[u-1]-check_val) < abs(data_i[u]-check_val)) {
        retval[index_val] <- u-1
      } else {
        retval[index_val] <- u
      }

      # Update index_val and check_val
      index_val <- index_val+1
      check_val <- check_val+step
    }
    if((size+1) < index_val) {
      break
    }
  }

  # Finally, add the last index to retval and return it
  retval[index_val] <- length(data_i)
  return(retval)
}
```

Given the full set of input data sizes from the profiling results, the minimum and maximum values are found. Next, the minimum is subtracted from the maximum and divided by the number of intermediate samples desired plus one, producing a data size step. This step, when added to the minimum value repeatedly, will result in a series of data sizes evenly spaced between the minimum and maximum values. The minimum sample size of this method is two, the first and last values available. This was chosen as a minimum sample size for simplicity and to ensure that the regression functions received at least two samples at a minimum. Initially, the calculated step is added to the minimum value and the result is compared to the input data. The index of the element closest to that result is recorded, the step value is added in again, and the next index is found. This process repeats itself until either the total desired number of samples is found or it becomes impossible to generate more samples due to the contents of the input data. An example of this would be the dataset [100, 200, 500, 1000]. If a sample size of three is desired, the indexes [1, 3, 4] would be returned. However, if a sample size of four were requested, the same indices [1, 3, 4] would be returned due to the gap between elements three and four preventing an even spread. Given a dataset [100 200 300 400 500 600 700 800 900 1000] on the other hand, and [1, 4, 7, 10] would be returned by the same request. This limitation occasionally lead to multiple samples of the same size with little difference in element choice between them. In those case, when the results were being summarized, the values reported for samples with identical data sizes were averaged together to reduce them to one usable number.

Once each sample had been taken, R's *lm* regression function was used to generate a regression model based upon the data. Due to the nature of the polynomials being approximated, a more advanced set of arguments were used with the *lm* function provided by R in order to generate the appropriate curves.

```
data  <- read.csv("filename.csv")

# O(N log(N)) approximation for the Vector Sorting function
fit_r <- lm(data$R~data_m$DATA.SIZE*log(data_m$DATA.SIZE))

# O(N) - O(N^poly_degree) fitting for the remaining functions
fit_r <- lm(data$R~poly(data_m$DATA.SIZE, poly_degree, raw=TRUE))
```

Each of the above examples follow the same general formula. In R, the *lm* function is used to carry out regression, requiring a formula and the data being worked on as arguments. In these cases, the formula and data are one and the same. The arguments to each function consists of a left hand side and a right hand side, separated by a ˜. In these cases, the left hand side is the same for both arguments; *data$R*, the timing results for the R implementation of whatever function is being analyzed. The right hand sides differ in that the first one is meant for modeling a log based time complexity, in this case using the natural logarithm for simplicity, while the second version is for specifying polynomial arguments. The first argument can be read as the elements of *data$R* are the result of carrying out the function *x*log(x)* on the corresponding elements of *data$DATA.SIZE*. The second argument is slightly more complicated, using the *poly* function, but is read as the elements of *data$R* are the result of carrying out a polynomial function of degree *poly_degree* on the corresponding elements of *data$DATA.SIZE*. The *poly* function, with the use of the $raw\bar{T}RUE$ argument, generates a raw polynomial over the set of points found in *data$DATA.SIZE*. Without the *raw* argument, the result will be an orthogonal polynomial. The choice the set *raw=TRUE* was made for consistency simply because it produced a more accurate model in all cases but Vector Sorting, where the differences between the raw and orthogonal results were minimal.

A model of time complexity was generated for each of the four implementations of each function. Those models were then used to generate a set of predicted execution times based upon the full set of data sizes profiled for each function. The predicted results were then analyzed to determine how close the model was to the actual data by calculating the root mean squared error and maximum error values which were recorded for each sample size. Finally, a quick check was made to find the approximate point of intersection between the R, BC, and CPP regression models and the GPU regression models to ad another indicator of how accurate they were.

Chapter 6

Results

6.1 Sum, Squaresum, and Cubesum

The Sum, Squaresum, and Cubesum functions are all versions of vector summation; the process of summing the contents of a vector to produce one number. Sum, figure 6.1, is the simplest version, taking $O(N)$ time, and simply iterating through a given vector, adding each component to the sum. Squaresum, figure 6.2, and Cubesum, figures 6.3 and 6.4, are modifications of Sum that take $O(N^2)$ and $O(N^3)$ time respectively. All three external CUDA and C++ functions were coded using either CUDA or the C++ Algorithm [5] library, with modifications to increase time complexity. These three functions were created due to their simplicity and ease of modification to allow for the study of the effects of different time complexities on execution time.

Overall, these three functions share similar code, with the only difference being the number of iterations the sum function is run for.

As the plots show, as the time complexity increases from $O(N)$ to $O(N^3)$, the opportunity to achieve a speedup using function replacement appears. Shown in figure 6.1, the GPU function is initially the slowest of the four implementations. However, once a data size of twenty thousand elements is reached, the performance of the C implementation drops below that of the GPU implementation. From that point on, the GPU implementation executes faster than the C code. Throughout the entire dataset, the R and Bytecode implementation both took similar amounts of time and were the fastest implementations. In this case, the parallel functions poor performance compared to R, BC, and C for the early data sizes is due to the time needed to transfer data from the host device to the GPU and back again. In those cases, the CPU can perform the entire calculation well before the parallel function can transfer the data, process it, and transfer it back. The difference between the R functions and the C function is simply caused by the fact that the R functions are built on optimized C and FORTRAN libraries that perform the vector summation more efficiently than the C implementation being used.

As the time complexity increases, the performance of the R, BC, and C implemen-

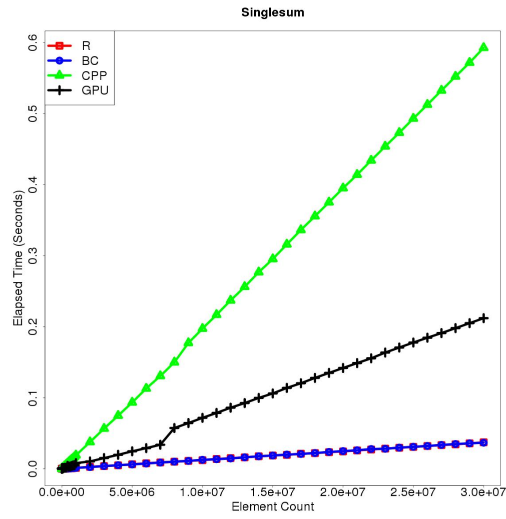

Figure 6.1: Sum

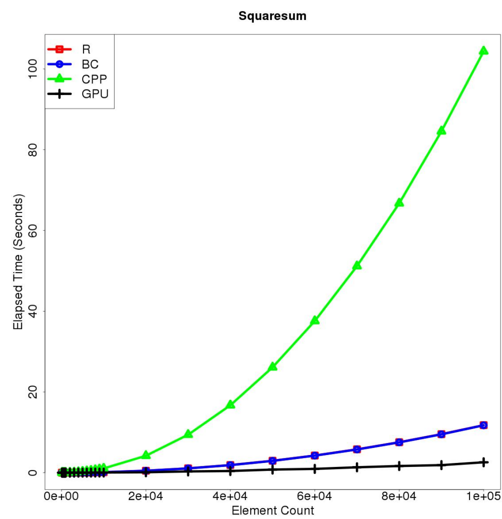

Figure 6.2: Squaresum

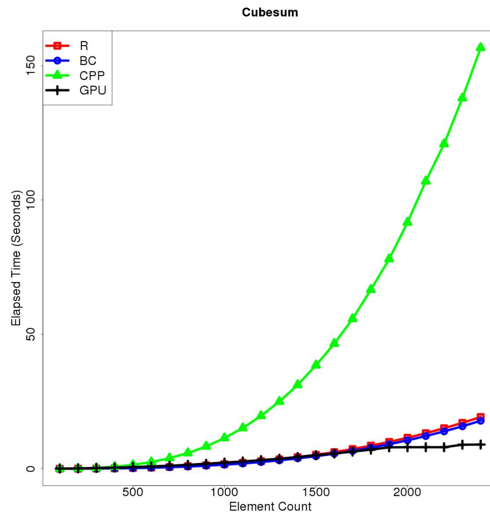

Figure 6.3: Cubesum

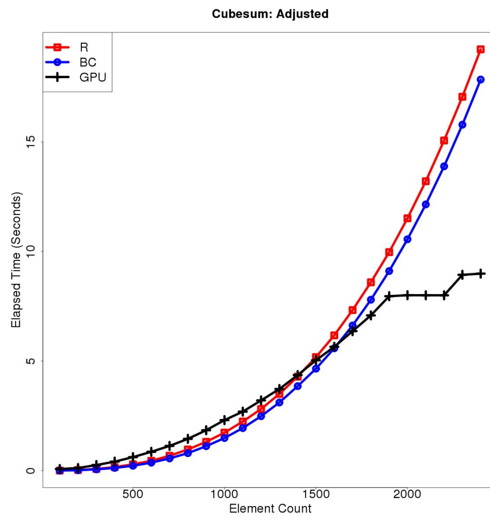

Figure 6.4: Cubesum: Adjusted

tations of the vector summation function degrades further. By figure 6.2, the parallel function has pulled ahead even further and replacement becomes reliably profitable starting at vectors of four hundred elements for the C implementation and two thousand elements for the R and BC implementations. Up until those points, the C, R, and BC functions execute faster. However, at those points, the benefits of parallel execution begin to outweigh the costs of transfer times.

Finally, by figures 6.3 and 6.4, the parallel function begins outperforming the sequential implementation quickly, starting at three hundred element vectors for the C implementation, fifteen hundred element vectors for the R implementation, and seventeen hundred element vectors for the BC implementation. However, by this point the parallel function begins to show some signs of irregularity. As the plot shows, at nineteen hundred elements, the performance of the parallel function levels off and then jumps up at twenty three hundred elements. As shown in the Cubesum Thread and Block Count table 9.8, this jump and leveling out is associated with a change in the number of blocks being used by the CUDA kernel function that occurs automatically as part of the function's built in optimizations. At the point of the jump, the number of blocks used by the CUDA kernel is increased to handle larger data sizes. The following plateau is the result of the system taking similar amounts of time to handle pieces of data that while different in size do not require another increase in block size. A similar pattern is likely present in Singlesum and Squaresum, but is not as apparent due to the larger spread of data sizes used and the lower time complexity.

6.2 Vector Addition

Vector Addition is the process of adding two vectors, in this case of equal length, together and returning the result as a third vector, taking $O(N)$ time. The parallel code for this is based on the code provided by the NVIDIA CUDA 4.2 SDK [3], with modifications made to simplify its host code and interface with the R objects being passed in, while the C++ and R code is hand written. In this case, the results for Singlesum are supported as it is always better to use the R implementation of vector addition. For all data sizes tested, it is simply faster to do this addition in R than it is to transfer the vectors to the GPU, due to data transfer times between devices.

Figure 6.5 is significantly rougher than its counterpart, Singlesum. This is due to the minimal amount of time necessary to carry out the function. The initial data sizes are all small enough that it is possible to perform the addition in so little time that it does not always register to the timing functions. However, the results still show the same general trend found in Singlesum. For all data sizes, it takes longer to transfer data from host memory to the device, perform the calculations, and than transfer the data back than to perform the calculation sequentially. The most likely cause of the nearly consistent gap between the R/bytecode and parallel function is the lack of optimization in the CUDA function which simply generates one thread per

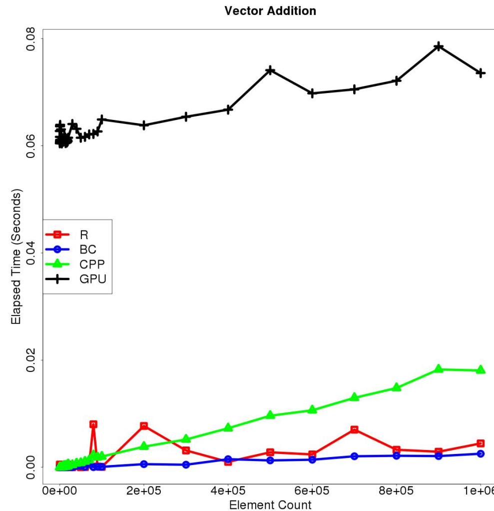

Figure 6.5: Vector Addition

vector element, rather than perform any tiling or memory optimizations. As with the adjusted Cubesum plot, figure 6.4, the execution time of the parallel implementation of this function shows sever regular irregularities in the form of occasional jumps in execution time. Given the simplicity of the CUDA kernel and the way the host code lays out the kernels grid and block structure, the cause of these jumps is currently unknown.

6.3 Vector Sorting

The Sorting function takes as input an integer vector and sorts its elements into ascending order taking $O(Nlog(N))$ time. This code is hand written using the CUDA Thrust library [4] and the optimized C++ Algorithm library [5]. For sorting, it becomes beneficial to swap over to the GPU implementation early on, after the vector grows to a length of several hundred elements. While the performance increase is not tremendous, it is there and in applications requiring large amounts of sorting, replacement would be beneficial. As expected, as the time complexity approaches $O(N^2)$, the benefits of parallel execution start to show up.

While Vector Sorting does not have a corresponding function in the Sum set, its time complexity comes midway between Singlesum and Squaresum, and as figure 6.6 shows, there is a point where the parallel code overtakes the R and bytecode functions. Specifically, this happens at vectors of size seven thousand for the C implementation, one million for the R implementation, and nine hundred thousand for the BC implementation. As before, the R and bytecode functions took approximately the same time and up to a point were more efficient than their parallel counterparts. Finally, the C function ended up being the slowest overall, again due simply to the fact

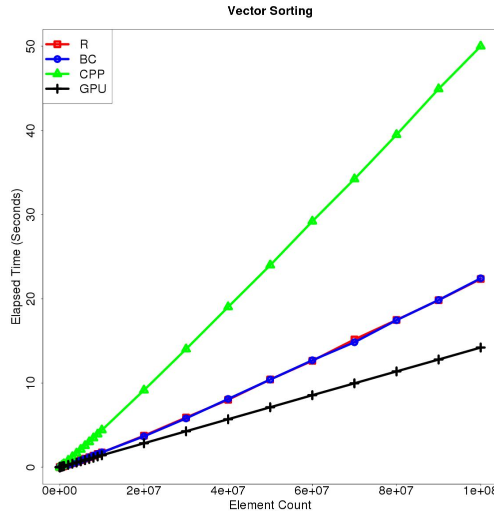

Figure 6.6: Vector Sorting

that R's implementation is designed to handle sequential functions such as sorting as efficiently as possible. In all cases, the plot shows a linear increase in processing time as data sizes increase, with no bumps or irregularities. In this case, this is due to the optimized nature of the functions used. The R and Bytecode functions are essentially identical and the C and CUDA functions both use libraries designed to either be as efficient as possible in the given circumstances.

6.4 Matrix Transposition

Matrix Transposition takes as input an NxN double matrix and generates its transpose, taking $O(N^2)$ time. The external code for this is based on the NVIDIA CUDA 4.2 SDK example [3], again modified for simplicity and to interact with R. Matrix transposition takes slightly longer to become beneficial, but the benefits are still apparent.

As seen in figure 6.7, by $O(N^2)$ time complexity, the point of transition occurs once the matrix size hits six thousand by six thousand for the C implementation, and eight thousand by eight thousand for the R and BC implementations. In this case, the cause of the spike in the plot of CPU timings at a matrix size of sixteen thousand by sixteen thousand is currently unknown. At that point, the matrix takes up approximately one point nine gigabytes of space on a system with eleven gigabytes of memory. The fact that there is a similar, if smaller bump in the R and bytecode timings at the same location appears to indicate that this is likely the work of the system adjusting to the increase in the amount of memory being used at that point. This feature is present in the data across multiple separate runs, ruling it out as being caused by a system interrupt. Without a more thorough examination of the R base

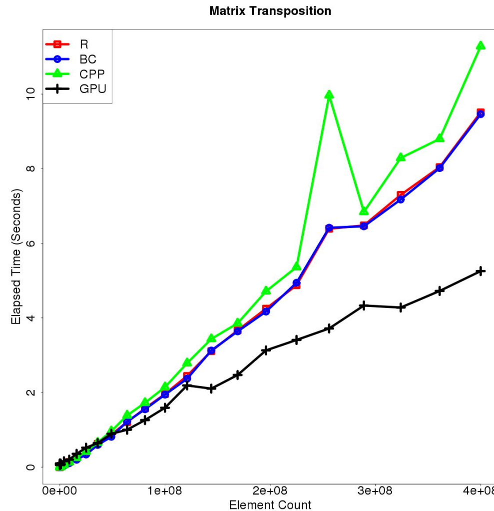

Figure 6.7: Matrix Transposition

code, the exact cause is unclear.

6.5 Matrix Multiplication

Matrix Multiplication is exactly what its name suggests, taking two matrices as input, it produces a third matrix that is the result of multiplying the two initial ones together, taking $O(N^3)$ time. The code for this benchmark is the most complex of all of the tests. As stated before, it was part of the GPUTools package and was chosen so time was not wasted reinventing the wheel. Specifically, the external parallel code uses the matrix multiplication code from the R GPUTools package [6], modified to be compiled on its own, outside of its package. The original code can be found on the GPUTools page of CRAN.

As figure 6.8 shows, the C implementation of the function has the worst performance of the five functions tested, as expected, with the R and BC implementations having much better performance and the GPU implementation being the fastest of all. For matrix multiplication, replacement is a foregone conclusion. As figures 6.8 and 6.9 support, once the time complexity of a function reaches $O(N^3)$, the data sizes where replacement becomes profitable become trivially small; in this case, immediately starting with the one hundred by one hundred element matrix size for both the R and C implementations and at the two hundred by two hundred element matrix for the BC implementation. For small data sizes, the execution time differences are minimal, and for larger ones, the parallel multiplication code provides a clear benefit. Again, the results from the Sum benchmarks are supported. As the CUDA function used in this benchmark was designed to work with R from the start, it produces a smooth timing result and as predicted outperforms the other three implementations.

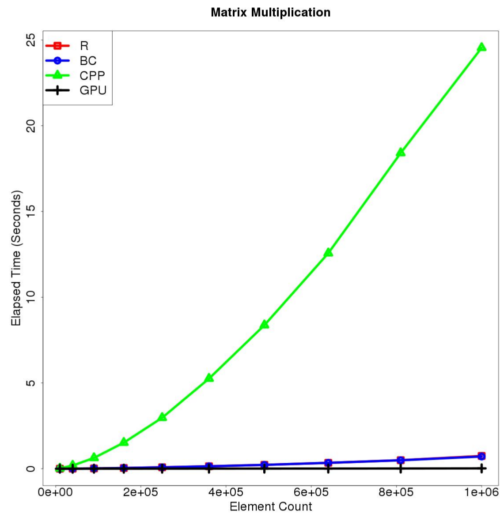

Figure 6.8: Matrix Multiplication

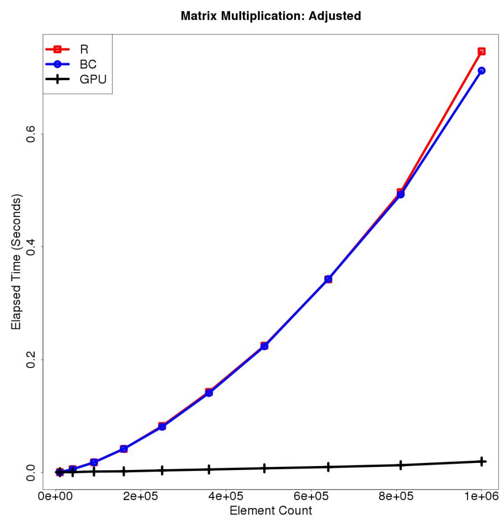

Figure 6.9: Matrix Multiplication: Adjusted

6.6 Transfer Timing

Transfer Times is another modification of sum that only measures the times used to transfer data back and fourth between R and the C++ code and between the system and GPU devices. Full data can be found in the Data section of the report.

Initial results show that transfer time between R and C++ code make up a minimal part of function execution, taking so little time that it did not register using the C++ time functions unless something interrupted the program, causing it to take longer, resulting in the spikes in the data. This is due to the fact that data transfers between R and C++ code consist simply of passing a pointer. The data is not copied or moved unless absolutely necessary, so the transfer takes almost no time. On the other hand, transfer times between the system and the GPU can take up varying portions of the execution time based upon function time complexity and data sizes, on average approximately seventeen percent of execution time for each data structure that needs to be transferred, with some variance for individual data sizes.

Overall, what this means is that the only transfer time that affects the parallel functions is that of transfers between the device and the host, which take up varying portions of the functions execution time depending on the data size and the time complexity of the parallel algorithm.

6.7 Profiling

As the previous section has shown, it is not always the case that a sequential R function can be replaced with a parallel implementation to produce a useful speedup. Functions with a time complexity of $O(N)$ take so little time that the sequential implementation outperforms any tested parallel implementation simply because of the necessary data transfer times. As mentioned in the Background section, this project only tested parallel execution using a GPU, so this may not hold true for parallel execution on a CPU where data transfer times will be different. On the other end of the spectrum, functions with a time complexity of $O(N^3)$ up are so complex that the time lost to memory transfers to and from the GPU are more than made up for by the speedup produced, so much so that it is practical to replace all sequential functions of such a complexity with a parallel counterpart.

In between those two extremes is where there is the possibility of improving execution time by dynamically switching between parallel and sequential functions based upon data sizes and computation complexity. While the results rarely show an execution time improvement outside of the double digits, the benefit of replacement would come from instances where large amounts of data are being analyzed, rather than in cases of individual bits. On a single piece of data, reducing the time a function takes from minutes to seconds may be convenient, but it is hardly worth the amount of effort that may be required to get that improvement. However, when going over hundreds of gigabytes of data, automatically switching functions to use the most ef-

ficient one based on the size of the individual piece of data being used could save a tremendous amount of time.

Given the benchmarking results, there is definitely the possibility of improving the performance of R scripts through automatic function choice based upon data sizes and time complexity. However, before this can be realized, there are still several hurdles that need to be overcome. While it is possible to use the benchmark results currently available to predict the performance of those specific functions based upon data size, the differences between the squaresum and matrix transposition show that time complexity is not the only factor in determining where the transition from sequential to parallel code should be. The desired end result of this work would be a system that can analyze a given set of sequential and parallel functions and determine the points where one function should be replaced with another. However, at the moment, the only way to do this wold be to actually run the function over various pieces of data to generate a predictive dataset for that specific function. This process can take an extended amount of time for some functions, and while it will allow fore profitable speedups later on, not everyone may be willing to wait. To this end, the next section discusses function time complexity modeling, performance prediction, and sample requirements.

6.8 Regression

Once function profiling was completed, we moved on to regression and execution time prediction. As explained in Methods, the timing data gathered for each function was used to generate a series of models of each function's time complexity. Two separate sampling methods were tested before the even method was chosen due to its more consistent results. The purpose of this modeling was to determine approximately how many samples were necessary to generate an accurate model of each function's time complexity. Initial testing was done using the naive sampling method. When it became apparent that this lead to unsatisfactory results, the even sampling method was created and chosen as a way of generating a better set of samples. Based upon the resulting observations, it was determined that the the overall required number of sample timings needed to accurately approximate a functions time complexity depends on two factors, the degree of a function's time complexity and a function's regularity, and that at least ten samples were necessary to achieve an acceptable result.

The first set of time complexity models examined were based upon the full data sets gathered previously. Given access to all of the gathered data, the R regression functions were able to consistently generate accurate approximations of the execution times of each function. As shown in figures 6.10, 6.11, 6.12, 6.13, 6.14, 6.15, and 6.16, when given access to the full set of available data, it is possible to generate accurate approximations of each function's time complexity, with most deviations being the result of irregularities in the data such as in figures 6.13 and 6.15 where execution

time spikes. As expected, given as much data as it was possible to generate in the time available, the resulting models were fairly accurate.

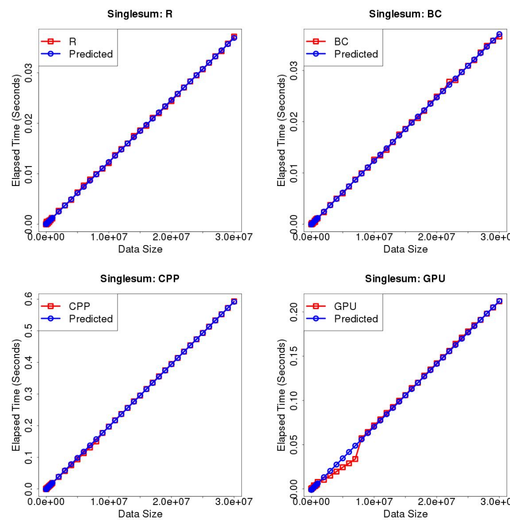

Figure 6.10: Singlesum Prediction, 66 Elements

However, as sample size was reduced, data regularity begin to have a greater effect on the resulting time complexity approximations. This does not mean that generating an accurate model of time complexity requires a large amount of profiling data, but it does mean that samples need to be chosen with care. An extreme example of bad sample choice would be figure 6.17. Here, only two samples data sizes were chosen, 100 and 20000 elements, and the results are noticeable. For the R and GPU models, the two points chosen were both well below the average actual timing results, producing models that underestimate the execution time of the R implementation of vector addition and predict that the GPU implementation will take less time as the data size increases, all the way down to zero seconds. For the BC and CPP implementations, the chosen data sizes produce models that do the opposite and overestimate the function's execution time. Based on this and similar results, it was determined that the most likely method of generating a usable model of function execution time using regression would be to take a set of samples consisting of a minimum and maximum data size and a set of data sizes spaced as evenly between the two as possible.

In this case, the minimum and maximum data sizes were determined by what timing data was available. In an actual implementation of a system for automatic effective function choice, these values would be determined based upon the limitations of the available systems. The minimum, based upon a functions time complexity would need to be a value that consistently results in an execution time that is measurable by the system. for each of the tested functions, a data size of one hundred elements was chosen for vector operations and ten thousand elements for matrix op-

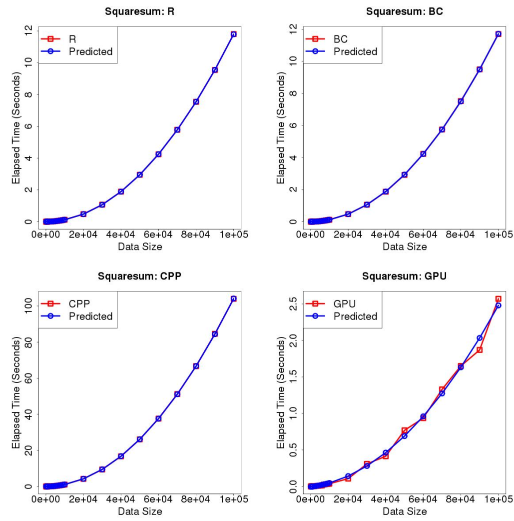

Figure 6.11: Squaresum Prediction, 28 Elements

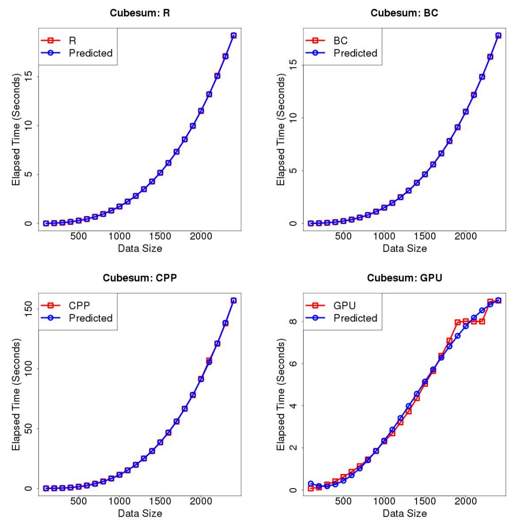

Figure 6.12: Cubesum Prediction, 24 Elements

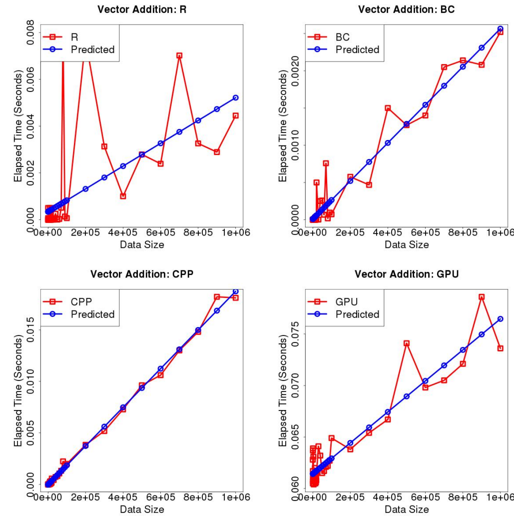

Figure 6.13: Vector Addition Prediction, 46 Elements

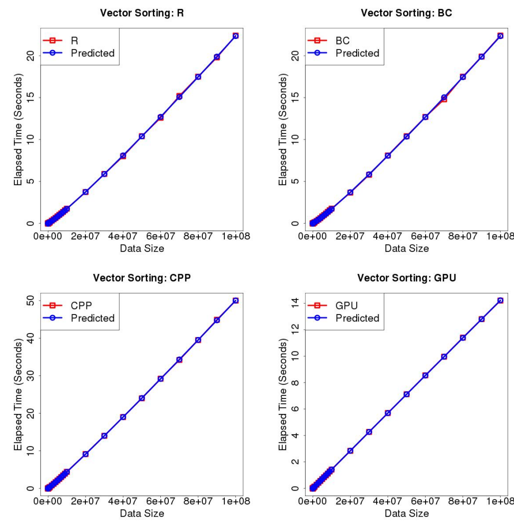

Figure 6.14: Vector Sorting Prediction, 64 Elements

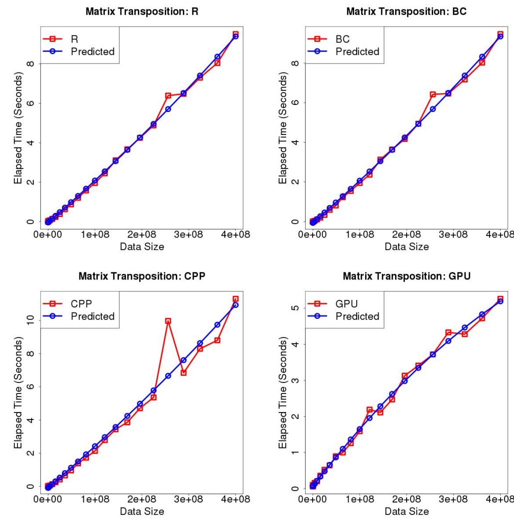

Figure 6.15: Matrix Transposition Prediction, 29 Elements

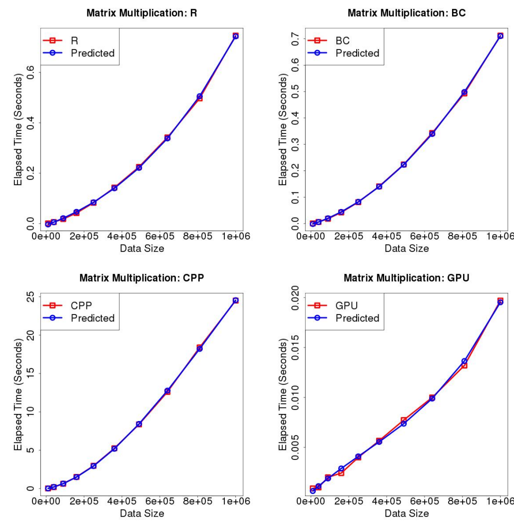

Figure 6.16: Matrix Multiplication Prediction, 10 Elements

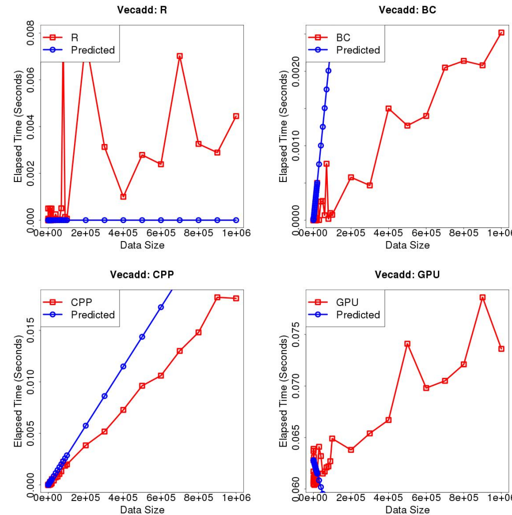

Figure 6.17: Vector Addition Prediction, 2 Elements, Inaccurate Results

erations. In this case, that size was not always large enough, as shown by the timing results for Vector Addition. The exact size necessary would depend on the system being used, but could be determined easily by testing small data sizes until a suitable value is found. The maximum data size on the other hand would be determined by the amount of memory available on the system. More specifically by whichever is smaller, main memory or the memory available on the GPU, since the data being used needs to be able to fit on both devices. While an exact method of determining the amount of available memory to use was not developed, a good starting point would be as close to one half of the capacity as possible. If main memory is the limiting factor, this leaves space for the system and anything else that needs it, and for both main memory and GPU memory, this avoids the problems associated with completely filling memory.

Given the available timing data, the even sample selection method was chosen as it produced the closest approximation of the desired sampling method that was available. With this method, a variety of sample sizes were tested, with a summary of the results in table 6.1 which reports on the Normalized Root Square Mean Errors of each function for various data sizes. The NRMSE is a measure of the difference between the values predicted by the model of function time complexity and the observed results, with values ranging from zero to one hundred percent. The lower the value, the closer the predicted values are to the observed ones.

For the variety of sample sizes tested, most functions produced normalized root mean squared errors of eight percent or less. Matrix Transposition produced slightly higher values, but still stayed under twenty percent. Finally, Vector Addition produced much higher values ranging between one hundred and thirty eight point two percent and five point nine percent. For NRMSE measurements, values below twenty

percent are acceptably accurate and below five percent are great. Larger values than that indicate that the model does not accurately represent the available data. This does not necessarily mean that the model is good or bad, just that it cannot produce the same results as the profiling provided. The results for Vector Addition are a good example of this. Looking at figure 6.13, the predicted results follow the same general trend as the observed timing, but it is impossible for a polynomial of degree one to accurately model the amount of variance that is found in the observed results.

Based upon the summary in table 6.1, it is apparent that the regularity of the data provided as a sample is very important. The artificial complexity functions Singlesum, Squaresum, and Cubesum produced the most regular data and this results in the NRMSE of the resulting model being very low for any sample size; less than two percent for the R, BC, and CPP implementations and less than ten percent for the GPU implementations. The confirmation functions however provide more varied results. Vector Sorting and Matrix Multiplication, both being fairly regular with only a few highs and lows in their data both produced minimum and maximum NRMSE values similar to the artificial complexity functions with only a few higher or lower, and with higher standard deviations. The Matrix Transposition function, being the next most regular produces minimum and maximum NRMSE values that are higher than the previously mentioned functions, but still within the realm of acceptability. Finally, the Vector Addition function produced the largest NRMSE values due to the variance of it's data.

6.9 Prediction Accuracy

Given the variance in the NRMSE of each function for the various sample sizes, the next step taken was to examine the results of each regression and see how closely each approximation came to the observed values when predicting where to swap sequential functions for their parallel counterparts. For each available sample size of each function, the intersections between the R, BC, and CPP implementations and the GPU implementation were found, in the tested data sizes and to the nearest integer, and the absolute value of the difference between the observed and predicted intersections was calculated. The results of this can be seen in figures 6.18 through 6.24 which show the R function results. Since these plots show the difference between the predicted and observed points of intersection for each function implementation, the closer a point is to zero the better. Finally, table 6.2 summarizes the resulting differences from observed intersections for all functions and implementations, reporting the minimum, median, and maximum values alone with the standard deviation of the differences found among the varying sample sizes.

Overall, some differences are to be expected when there is an intersection to find. The observed points of intersection are based on a finite set of data sizes and represent the first data size where the GPU implementation became consistently faster than the R, BC, or CPP implementation. Given the gaps in between individual data sizes, even

Table 6.1: Regression Normalized Root Square Mean Error Summary

FUNCTION	MIN	MED	MAX	STD DEV
R				
Singlesum	1.10%	1.10%	1.60%	0.11%
Squaresum	0.10%	0.10%	0.20%	0.03%
Cubesum	0.30%	0.30%	0.40%	0.04%
Vector Addition	77.90%	90.75%	138.20%	18.15%
Vector Sorting	0.60%	0.90%	4.60%	1.01%
Matrix Transposition	5.40%	5.98%	17.00%	2.86%
Matrix Multiplication	1.70%	1.75%	2.10%	0.16%
BC				
Singlesum	1.30%	1.30%	2.10%	0.22%
Squaresum	0.20%	0.20%	0.20%	0.00%
Cubesum	0.30%	0.30%	0.50%	0.06%
Vector Addition	23.10%	24.20%	28.68%	2.03%
Vector Sorting	0.80%	1.20%	11.70%	2.75%
Matrix Transposition	0.65%	6.43%	10.30%	1.81%
Matrix Multiplication	1.00%	1.10%	1.40%	0.14%
CPP				
Singlesum	0.90%	1.30%	1.50%	0.17%
Squaresum	0.20%	0.20%	0.20%	0.00%
Cubesum	0.70%	0.70%	0.90%	0.06%
Vector Addition	5.90%	6.15%	7.20%	0.41%
Vector Sorting	0.20%	0.40%	2.30%	0.53%
Matrix Transposition	0.71%	20.75%	33.80%	6.23%
Matrix Multiplication	1.00%	1.10%	1.30%	0.10%
GPU				
Singlesum	4.60%	6.10%	7.20%	0.67%
Squaresum	6.20%	6.30%	7.80%	0.68%
Cubesum	7.10%	7.30%	9.00%	0.41%
Vector Addition	34.80%	41.00%	79.30%	11.85%
Vector Sorting	0.20%	0.40%	1.90%	0.47%
Matrix Transposition	3.26%	5.95%	7.30%	0.82%
Matrix Multiplication	4.40%	4.70%	5.20%	0.37%

if the predictions were perfect, the actual point of intersection still falls somewhere in between the observed point and the previous data size. When examining the plots, the data sizes being worked with and the variance in the differences should be taken into account. As an example, for Matrix Multiplication a difference from observed of fourteen thousand elements seems fairly large until you take into account that data sizes of up to four hundred million elements were tested.

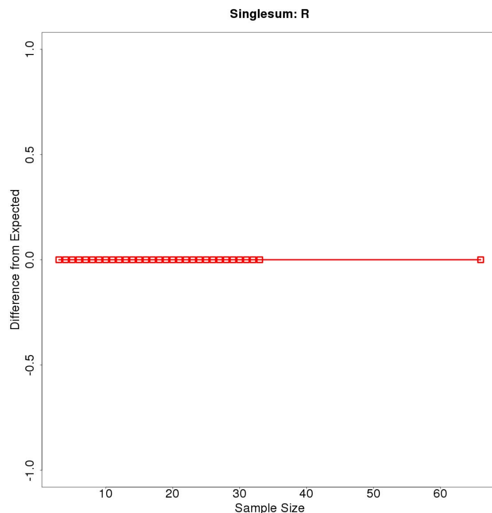

Figure 6.18: Singlesum Differences

For Singlesum, figure 6.18 the observed results were that there was no intersection between either the R or BC implementations and the GPU implementation and an intersection somewhere around twenty thousand elements for the CPP implementation. For R and BC, the predicted results were correct. No intersection was found. However, the predicted results also indicated that there was either no intersection between the CPP and GPU implementation, the majority of the results, or that there was an intersection at a much higher data size than expected in a few instances. The predicted results for Vector Addition, figure 6.19, on the other hand were exactly the same as the observed results, despite the high degree of inaccuracy reported by the NRMSE. No intersections between the R, BC, and CPP implementations and the GPU implementation.

For Vector Sorting, figure 6.20, the calculated intercepts between the R, BC, and CPP implementations and the GPU implementations were fairly regular for most sample sizes, with a few increases in difference at the smaller sample sizes.

As the degree of the time complexity of the functions increases further, the resulting differences begin to change. Both Squaresum, figure 6.21, and Matrix Transposition, figure 6.22 show significantly more variance in the difference between expected and predicted intersections, but show a general trend of either a leveling out of or a decreasing of differences as sample size increases.

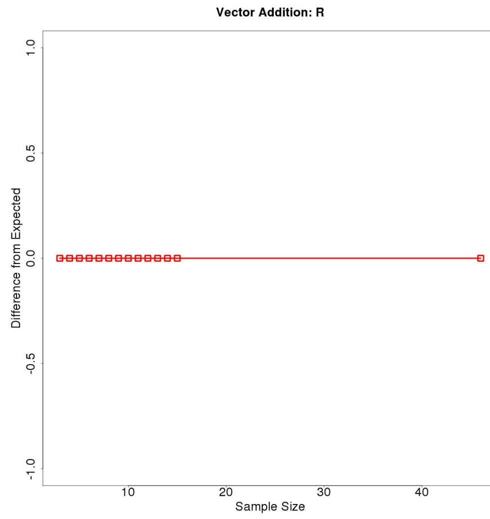

Figure 6.19: Vector Addition Differences

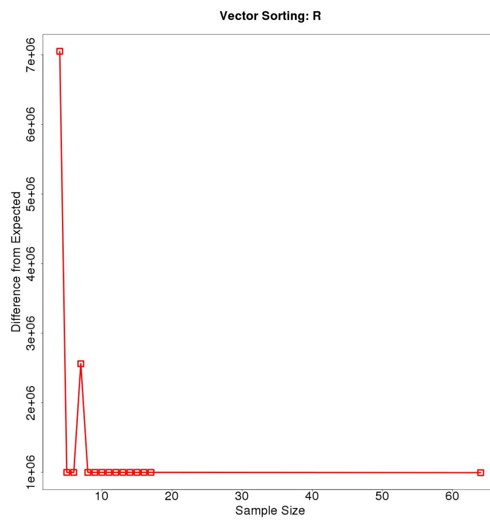

Figure 6.20: Vector Sorting Differences

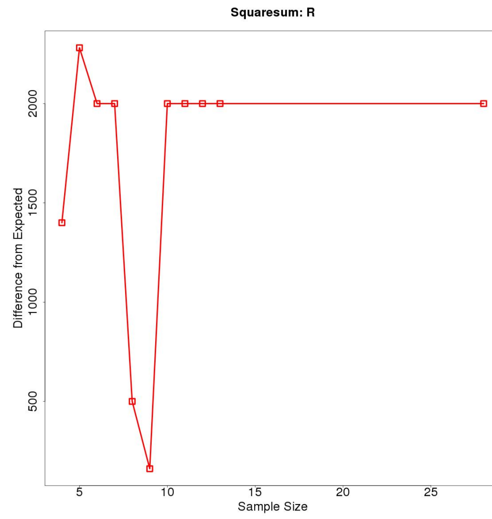

Figure 6.21: Squaresum Differences

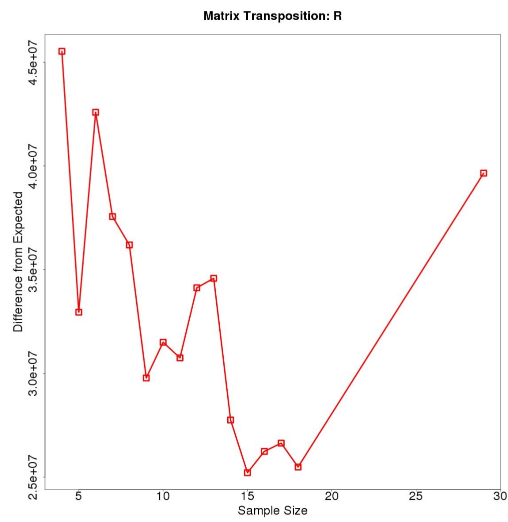

Figure 6.22: Matrix Transposition Differences

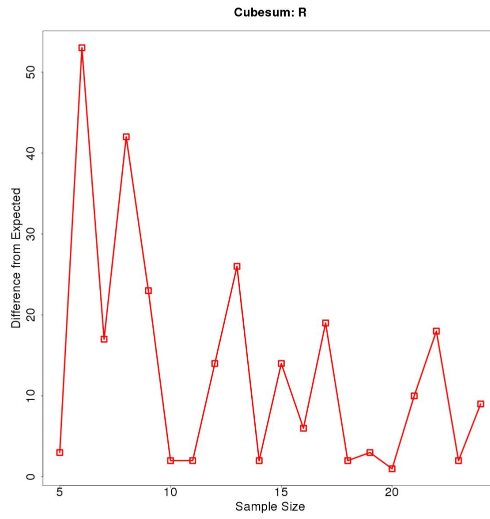

Figure 6.23: Cubesum Differences

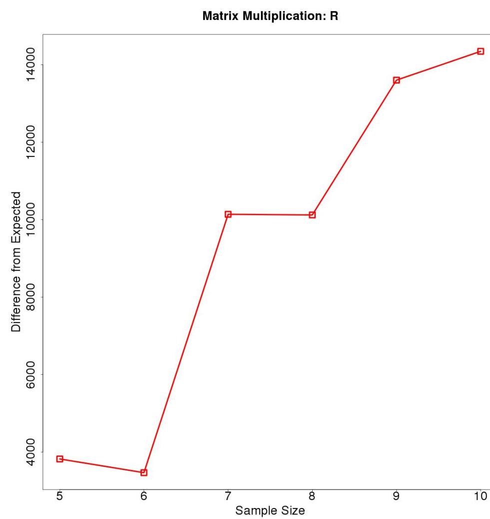

Figure 6.24: Matrix Multiplication Differences

Finally, the results of the functions with the highest time complexities, Cubesum and Matrix Multiplication show even more variance. Cubesum, given its larger sample size, exhibits a very irregular plot, while Matrix Multiplication, with its smaller sample size is slightly more regular, but is still far from perfect.

Table 6.2: Summary of Differences

FUNCTION	MIN	MED	MAX	STD DEV
R				
Singlesum	0.00	0.00	0.00	0.00
Squaresum	161.00	2000.00	2281.00	696.74
Cubesum	1.00	9.50	53.00	14.16
Vector Addition	0.00	0.00	0.00	0.00
Vector Sorting	995627.00	999872.60	7049989.00	1585270.44
Matrix Transposition	25215215.50	32227443.50	45529804.00	6198733.23
Matrix Multiplication	3467.00	10129.75	14346.00	4677.06
BC				
Singlesum	0.00	0.00	0.00	0.00
Squaresum	8.00	2000.00	2997.00	885.05
Cubesum	15.00	70.50	103.00	17.17
Vector Addition	0.00	0.00	0.00	0.00
Vector Sorting	895630.00	899874.00	6514824.00	1533151.32
Matrix Transposition	20615306.00	29084161.00	46009942.00	7535456.10
Matrix Multiplication	19933.00	23368.50	26800.00	2932.33
CPP				
Singlesum	20000.00	20000.00	296581.00	49791.12
Squaresum	458.00	1268.00	1640.00	311.04
Cubesum	67.00	79.00	300.00	68.71
Vector Addition	0.00	0.00	0.00	0.00
Vector Sorting	2582.00	6857.00	6894.00	1098.48
Matrix Transposition	808835.33	3745195.50	36000000.00	13875004.73
Matrix Multiplication	3621.00	8483.50	10000.00	2517.30

Looking back at the results, there are several noticeable patterns. The accuracy of each prediction appears to either increase as sample size increases or to reach a plateau where the difference between observed and predicted intersections stays approximately the same between samples. However, in several cases, the accuracy suddenly drops again for the full sized sample, as seen in figure 6.22, or continually decreases as sample size increases as seen in figure 6.24. These two results are both the result of the sensitivity of regression modeling to the input data.

In the case of Vector Sorting, the large variances are caused by the nature of the model. Since Vector Sorting has a time complexity of $O(Nlog(N)$, the function used to model its timing results is log based. This, coupled with the fact that early on

the models of the R and GPU execution times are close to parallel means that small changes introduced by taking different sample sizes, causing different elements to be chosen, produce large changes in the resulting point of intersection. The difference between figure 6.25, created using a sample size of six, and figure 6.26, created using a sample size of seven is tremendous. Both figures show a close up of the predicted intersection between the R and GPU implementations of Vector Sorting and clearly show the asymptote around zero caused by the nature of the log function that is not clearly shown when examining the full set of sample sizes. With smaller sample sizes, the general location of the point of intersection switches between the type founds in figure 6.25 and figure 6.26. As the sample size increases, the results constantly follow figure 6.25 which produces an intersection at almost the same point each time that is slightly closer to the observed value than the intersections produced by models like 6.26. However, as sample size increases from seventeen to sixty four, the intersection type jumps to that of figure 6.26, causing the observed increase in the difference between figure 6.27 and figure 6.28.

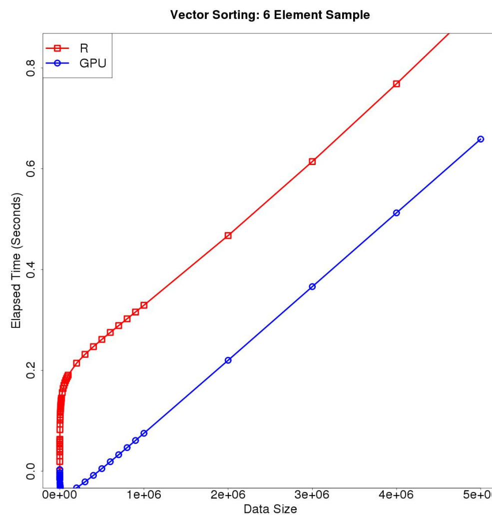

Figure 6.25: Vector Sorting A

In the case of Matrix Multiplication, the continuous increase in difference between observed and predicted point of intersection is also the result of small changes in the model as the sample size is increased. The Matrix Multiplication tests were run on matrices ranging in element count from ten thousand elements to four hundred million elements. What this means is that small changes in the model as the number of samples increases produce large changes in the actual predicted points of intersection. What look like minor differences between models created using different sample sizes result in noticeable changes in the point of intersection between the R and GPU implementations. In this case, it just so happens that those changes cause the difference between the expected and observed points of intersection to increase.

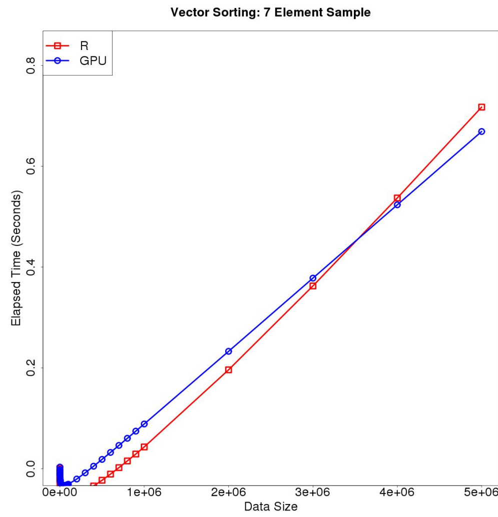

Figure 6.26: Vector Sorting B

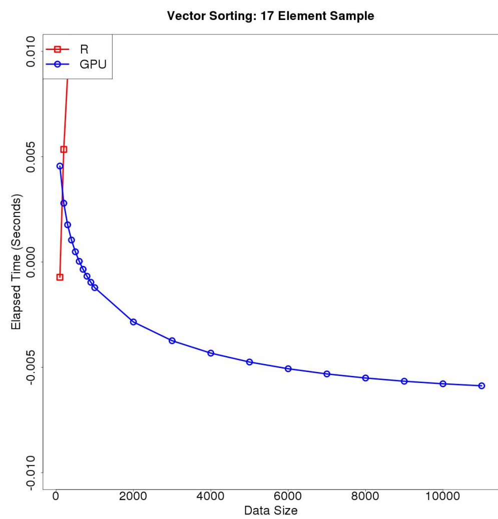

Figure 6.27: Vector Sorting, 17 Element Sample

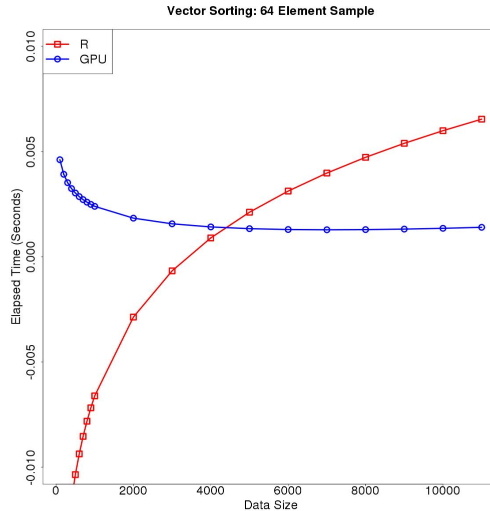

Figure 6.28: Vector Sorting, Full Sample

6.10 Summary

Overall, two measures of accuracy were used to try and determine what an acceptable sample size would be for regression modeling. First, the Normalized Root Mean Squared Error was examined, and while there was some variance in the NRMSE between the maximum possible sample size for each function and the minimum, it was never large enough to suggest that a large sample was necessary. While the exact sample size that produced the minimal NRMSE for each function was not exact between implementations, a sample size of approximately nine elements on average appears to produced minimal values quite often. However, since there is rarely a large difference between the minimum and maximum NRMSE, with Vector Addition being the exception, the minimum data size to produce an accurate approximation would appear to be a function's degree plus either two or three.

Next, the difference between the predicted and observed points of intersection between the R, BC, and CPP implementations of each function and the GPU implementation were examined. This produced a much different set of results than the NRMSE analysis. Given that the observed points of intersection are based off of a limited data set and are only approximations based upon where the GPU implementation was observed to begin performing faster than the other implementations, some variance was to be expected. This can be seen in figures 6.18 through 6.24 and in table 6.2, where as sample size increases, in most cases the differences between the observed and predicted results begin to either stabilize at a set of similar values or decrease towards the expected value. Based upon the provided plots, it appears that a sample size of at least ten elements is needed before the accuracy of the predicted points of intersection begins to level out.

Based upon this data, it is therefore recommended that a sample of at least ten data sizes, evenly spaced between a system dependent minimum and maximum be taken when preparing to perform regression modeling of a function. Relying only upon the NRMSE, a number of samples equal to one plus the degree of the function would appear to be all that is needed to create an accurate model. However, when the resulting differences between the predicted and observed intersections are taken into account that number is too small.

Chapter 7

Future Work

While the initial goal of this project was to implement a modified form of the R bytecode compiler that dynamically replaced sequential functions with parallel counterparts, it quickly became apparent that this was not feasible within the available time constraints. Based upon the gathered data, this would be the next step. Given a description of the time complexity of a sequential function and a parallel counterpart, the R bytecode Compiler could be modified to dynamically replace sequential functions with their parallel counterparts based upon data sizes compared to time complexity. Depending on implementation, this choice could either be based upon a static set of rules based upon current data or on a set of benchmarks run during the installation of the modified compiler package.

Once profiling data has been gathered, it becomes a matter of approximating the time complexity of a function using regression. The complexity in this action comes from gathering a set of profiling data of sufficient size and regularity that an accurate regression can be made using the approximation. While the work in this paper only scratches the surface of this topic, current results wold seem to indicate that a sample of at least ten data points is needed to generate an accurate model of a function. The next step would be to continue to investigate various methods of data sampling to see if the process can be simplified further, and to implement a profiling function that can gather this data with minimal input from the user.

Chapter 8

Conclusion

This project, Effective Function Choice in the R Scripting Language began as an investigation into the effectiveness of automatically parallelizing the R scripting language by replacing sequential functions with parallel counterparts. In order to test this possibility, several benchmarks were created or modified and measured. Sum, Squaresum, and Cubesum were created as a control to be used to measure the results of different time complexities and data sizes on code that was as identical as possible. Following that, Vector Addition, Vector Sorting, Matrix Transposition, and Matrix Multiplication were chosen as real world examples of the time complexities found in the Sum benchmarks. Finally, Load Times was created to examine data transfer times.

After the benchmarking was complete, it became apparent that there is an interval between $O(N)$ and $O(N^3)$ exclusive where an R scripts execution time could be improved by function replacement. At $O(N)$, for our functions, data transfer times for parallel CUDA functions are longer than the time needed to sequentially perform the desired calculations, and at $O(N^3)$ Time complexities replacement becomes beneficial so early on that it is basically a given that it should occur. Between those two extremes, the exact point where function replacement can provide an improvement in performance varies based on exact time complexity and data sizes.

Once the benchmarking results were gathered, they were used to begin a study into the best methods of generating approximations of each function's time complexity using regression. Various sample sizes were tested to determine what effect sample size and choice have on the accuracy of the predicted results. Both the Normalized Root Mean Squared Error and difference between expected and observed intercepts were examined. In the end, the NRMSE analysis suggesting a sample size per function of time complexity plus between two and three samples at a minimum, due to the limited variance found associated with sample size. The difference analysis on the other hand showed that for samples where the difference between predicted and observed intersections was going to level off, a minimum ten samples would be recommended.

Chapter 9

Data

9.1 Introduction

This section contains the raw data gathered for each benchmark. Each table is split into several sections. INPUT is the number of elements in the data structure being used. OUTPUT is the number of elements in the returned data structure. CALC.COUNT is the approximate number of calculations done to produce the results. The final four sections are the timing results for each benchmark, R, BC(bytecode), CPP, and GPU. Each time is reported in seconds unless otherwise noted. If a result is not available, it will be labeled with *N/A*. The only reason that this will happen is if obtaining that result would have taken an inordinate amount of time. Any timing results of zero are the result of the action being measured taking so little time that the timing functions could not return a result.

GPU transfer time is a recording of the time needed to transfer data for the sum benchmark from main memory to the GPU and back. GPU transfer timing is divided into four sections. TOTAL.ELAPSED is the total elapsed time in seconds spend performing memory transfers. TO.DEVICE and TO.HOST are the sub times spent transferring data specifically to the GPU or back to main memory respectively. FUNCTION.ELAPSED is the average time previously found for each execution of the Sum function on the GPU.

CPP transfer time is a recording of the time needed to transfer data for the sum benchmark from R to the external C code. TRANSFER.TIME is the time necessary to pass the data pointer to the C code and back to R again.

Table 9.1: Singlesum Timing Results

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
100	1	100	3.91E-006	0.00E+000	0.00E+000	4.16E-004
200	1	200	0.00E+000	0.00E+000	6.25E-005	6.45E-005
300	1	300	0.00E+000	0.00E+000	0.00E+000	3.32E-005
400	1	400	0.00E+000	0.00E+000	0.00E+000	2.58E-004
500	1	500	0.00E+000	0.00E+000	0.00E+000	6.45E-005
600	1	600	0.00E+000	0.00E+000	0.00E+000	5.33E-004
700	1	700	0.00E+000	0.00E+000	0.00E+000	5.02E-004
800	1	800	0.00E+000	0.00E+000	0.00E+000	1.76E-005
900	1	900	0.00E+000	0.00E+000	1.95E-006	3.32E-005
1000	1	1000	0.00E+000	0.00E+000	0.00E+000	2.58E-004
2000	1	2000	0.00E+000	0.00E+000	5.00E-004	6.64E-005
3000	1	3000	0.00E+000	0.00E+000	1.25E-004	1.56E-005
4000	1	4000	0.00E+000	0.00E+000	1.56E-005	3.32E-005
5000	1	5000	0.00E+000	0.00E+000	5.02E-004	6.64E-005
6000	1	6000	0.00E+000	0.00E+000	7.81E-006	2.68E-004
7000	1	7000	6.25E-005	0.00E+000	1.56E-005	5.33E-004
8000	1	8000	0.00E+000	0.00E+000	5.00E-004	6.25E-005
9000	1	9000	0.00E+000	0.00E+000	1.29E-004	5.35E-004
10000	1	10000	3.91E-006	0.00E+000	3.32E-005	5.33E-004
20000	1	20000	1.95E-006	1.95E-006	2.91E-004	2.85E-004
30000	1	30000	0.00E+000	0.00E+000	6.78E-004	6.43E-004
40000	1	40000	2.50E-004	0.00E+000	9.36E-004	6.46E-004
50000	1	50000	0.00E+000	0.00E+000	1.00E-003	6.66E-004
60000	1	60000	1.95E-006	1.95E-006	1.02E-003	6.66E-004

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
70000	1	70000	5.02E-004	2.50E-004	1.29E-003	4.20E-004
80000	1	80000	3.13E-005	1.56E-005	1.42E-003	7.15E-004
90000	1	90000	1.95E-006	1.25E-004	1.87E-003	9.34E-004
1e+05	1	1e+05	2.52E-004	1.56E-005	2.00E-003	9.34E-004
2e+05	1	2e+05	1.43E-004	6.64E-005	4.12E-003	1.33E-003
3e+05	1	3e+05	1.60E-004	2.87E-004	5.69E-003	1.50E-003
4e+05	1	4e+05	6.66E-004	3.32E-004	7.67E-003	4.10E-003
5e+05	1	5e+05	4.28E-004	6.78E-004	9.54E-003	3.00E-003
6e+05	1	6e+05	8.05E-004	4.34E-004	1.13E-002	3.17E-003
7e+05	1	7e+05	7.46E-004	9.36E-004	1.31E-002	3.94E-003
8e+05	1	8e+05	1.00E-003	1.00E-003	1.47E-002	4.67E-003
9e+05	1	9e+05	1.25E-003	1.13E-003	1.67E-002	5.03E-003
1e+06	1	1e+06	1.13E-003	1.13E-003	1.90E-002	7.93E-003
2e+06	1	2e+06	2.66E-003	2.32E-003	3.76E-002	1.03E-002
3e+06	1	3e+06	3.72E-003	3.75E-003	5.65E-002	1.50E-002
4e+06	1	4e+06	4.81E-003	5.00E-003	7.47E-002	1.96E-002
5e+06	1	5e+06	6.31E-003	6.00E-003	9.33E-002	2.45E-002
6e+06	1	6e+06	7.67E-003	7.33E-003	1.13E-001	2.90E-002
7e+06	1	7e+06	8.86E-003	8.71E-003	1.31E-001	3.38E-002
8e+06	1	8e+06	9.94E-003	9.95E-003	1.50E-001	5.73E-002
9e+06	1	9e+06	1.10E-002	1.10E-002	1.77E-001	6.43E-002
1e+07	1	1e+07	1.21E-002	1.26E-002	1.97E-001	7.17E-002
1.1e+07	1	1.1e+07	1.37E-002	1.34E-002	2.17E-001	7.86E-002
1.2e+07	1	1.2e+07	1.49E-002	1.45E-002	2.37E-001	8.59E-002

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
1.3e+07	1	1.3e+07	1.60E-002	1.60E-002	2.56E-001	9.25E-002
1.4e+07	1	1.4e+07	1.75E-002	1.75E-002	2.77E-001	9.97E-002
1.5e+07	1	1.5e+07	1.86E-002	1.86E-002	2.95E-001	1.06E-001
1.6e+07	1	1.6e+07	1.95E-002	1.99E-002	3.16E-001	1.14E-001
1.7e+07	1	1.7e+07	2.11E-002	2.07E-002	3.36E-001	1.20E-001
1.8e+07	1	1.8e+07	2.20E-002	2.21E-002	3.56E-001	1.28E-001
1.9e+07	1	1.9e+07	2.33E-002	2.36E-002	3.75E-001	1.35E-001
2e+07	1	2e+07	2.44E-002	2.49E-002	3.95E-001	1.42E-001
2.1e+07	1	2.1e+07	2.58E-002	2.60E-002	4.14E-001	1.49E-001
2.2e+07	1	2.2e+07	2.71E-002	2.78E-002	4.34E-001	1.56E-001
2.3e+07	1	2.3e+07	2.83E-002	2.81E-002	4.54E-001	1.64E-001
2.4e+07	1	2.4e+07	2.95E-002	2.97E-002	4.73E-001	1.71E-001
2.5e+07	1	2.5e+07	3.07E-002	3.09E-002	4.93E-001	1.78E-001
2.6e+07	1	2.6e+07	3.20E-002	3.20E-002	5.13E-001	1.85E-001
2.7e+07	1	2.7e+07	3.33E-002	3.35E-002	5.33E-001	1.91E-001
2.8e+07	1	2.8e+07	3.43E-002	3.48E-002	5.52E-001	1.98E-001
2.9e+07	1	2.9e+07	3.58E-002	3.58E-002	5.72E-001	2.05E-001
3e+07	1	3e+07	3.72E-002	3.66E-002	5.93E-001	2.12E-001

Table 9.2: Squaresum Timing Results

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
100	1	10000	1.95E-006	7.81E-006	7.81E-006	3.50E-004
200	1	40000	2.58E-004	5.02E-004	2.91E-004	6.66E-004
300	1	90000	2.68E-004	6.64E-005	6.25E-004	9.92E-004
400	1	160000	6.46E-004	2.85E-004	1.73E-003	1.25E-003
500	1	250000	6.78E-004	3.50E-004	2.43E-003	1.14E-003
600	1	360000	8.67E-004	8.38E-004	3.94E-003	1.67E-003
700	1	490000	1.25E-003	8.71E-004	5.13E-003	1.93E-003
800	1	640000	1.06E-003	1.25E-003	6.87E-003	1.69E-003
900	1	810000	1.33E-003	1.00E-003	8.26E-003	2.51E-003
1000	1	1000000	1.93E-003	1.42E-003	1.06E-002	2.67E-003
2000	1	4000000	5.98E-003	5.55E-003	4.19E-002	4.35E-003
3000	1	9000000	1.23E-002	1.19E-002	9.42E-002	6.99E-003
4000	1	16000000	2.07E-002	1.93E-002	1.68E-001	9.00E-003
5000	1	25000000	3.15E-002	3.09E-002	2.61E-001	1.15E-002
6000	1	36000000	4.51E-002	4.35E-002	3.77E-001	1.36E-002
7000	1	49000000	6.08E-002	5.99E-002	5.12E-001	2.60E-002
8000	1	64000000	7.96E-002	7.83E-002	6.70E-001	2.97E-002
9000	1	81000000	9.93E-002	9.83E-002	8.49E-001	3.33E-002
10000	1	100000000	1.22E-001	1.21E-001	1.05E+000	3.73E-002
20000	1	400000000	4.77E-001	4.72E-001	4.19E+000	1.07E-001
30000	1	900000000	1.07E+000	1.06E+000	9.40E+000	3.09E-001
40000	1	1600000000	1.89E+000	1.87E+000	1.67E+001	4.14E-001
50000	1	2500000000	2.95E+000	2.92E+000	2.61E+001	7.69E-001
60000	1	3600000000	4.24E+000	4.23E+000	3.76E+001	9.36E-001

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
70000	1	4900000000	5.78E+000	5.75E+000	5.12E+001	1.33E+000
80000	1	6400000000	7.54E+000	7.52E+000	6.67E+001	1.65E+000
90000	1	8100000000	9.54E+000	9.50E+000	8.46E+001	1.87E+000
1e+05	1	10000000000	1.18E+001	1.17E+001	1.04E+002	2.57E+000

Table 9.3: Cubesum Timing Results

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
100	1	1000000	6.43E-003	4.06E-003	1.23E-002	7.49E-002
200	1	8000000	3.09E-002	2.20E-002	9.39E-002	1.21E-001
300	1	27000000	7.99E-002	5.91E-002	3.12E-001	2.57E-001
400	1	64000000	1.63E-001	1.26E-001	7.36E-001	4.12E-001
500	1	125000000	2.91E-001	2.23E-001	1.43E+000	6.18E-001
600	1	216000000	4.50E-001	3.67E-001	2.47E+000	8.62E-001
700	1	343000000	6.81E-001	5.59E-001	3.93E+000	1.13E+000
800	1	512000000	9.67E-001	7.99E-001	5.84E+000	1.45E+000
900	1	729000000	1.32E+000	1.12E+000	8.29E+000	1.85E+000
1000	1	1000000000	1.73E+000	1.49E+000	1.14E+001	2.31E+000
1100	1	1331000000	2.24E+000	1.94E+000	1.51E+001	2.69E+000
1200	1	1728000000	2.81E+000	2.49E+000	1.97E+001	3.21E+000
1300	1	2197000000	3.50E+000	3.11E+000	2.50E+001	3.73E+000
1400	1	2744000000	4.29E+000	3.86E+000	3.12E+001	4.36E+000
1500	1	3375000000	5.18E+000	4.65E+000	3.85E+001	5.04E+000
1600	1	4096000000	6.18E+000	5.59E+000	4.65E+001	5.65E+000
1700	1	4913000000	7.33E+000	6.63E+000	5.58E+001	6.37E+000
1800	1	5832000000	8.60E+000	7.80E+000	6.66E+001	7.09E+000
1900	1	6859000000	9.97E+000	9.11E+000	7.80E+001	7.96E+000
2000	1	8000000000	1.15E+001	1.06E+001	9.16E+001	8.01E+000
2100	1	9261000000	1.32E+001	1.22E+001	1.07E+002	8.00E+000
2200	1	10648000000	1.51E+001	1.39E+001	1.21E+002	8.00E+000
2300	1	12167000000	1.71E+001	1.58E+001	1.38E+002	8.94E+000
2400	1	13824000000	1.92E+001	1.78E+001	1.57E+002	8.99E+000

Table 9.4: Vector Addition Timing Results

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
100	100	100	0.00E+000	0.00E+000	0.00E+000	6.28E-002
200	200	200	5.00E-004	0.00E+000	0.00E+000	6.05E-002
300	300	300	6.25E-005	0.00E+000	1.95E-006	6.17E-002
400	400	400	0.00E+000	0.00E+000	0.00E+000	6.07E-002
500	500	500	0.00E+000	0.00E+000	0.00E+000	6.37E-002
600	600	600	0.00E+000	0.00E+000	0.00E+000	6.07E-002
700	700	700	0.00E+000	0.00E+000	0.00E+000	6.09E-002
800	800	800	6.25E-005	0.00E+000	1.56E-005	6.37E-002
900	900	900	1.95E-006	0.00E+000	0.00E+000	6.11E-002
1000	1000	1000	0.00E+000	0.00E+000	3.12E-005	6.39E-002
2000	2000	2000	0.00E+000	0.00E+000	0.00E+000	6.08E-002
3000	3000	3000	0.00E+000	0.00E+000	0.00E+000	6.31E-002
4000	4000	4000	0.00E+000	0.00E+000	7.81E-006	6.07E-002
5000	5000	5000	0.00E+000	0.00E+000	3.12E-005	6.04E-002
6000	6000	6000	0.00E+000	0.00E+000	3.13E-005	6.14E-002
7000	7000	7000	0.00E+000	7.81E-006	1.33E-004	6.09E-002
8000	8000	8000	0.00E+000	0.00E+000	7.81E-006	6.07E-002
9000	9000	9000	0.00E+000	0.00E+000	3.12E-005	6.10E-002
10000	10000	10000	5.02E-004	0.00E+000	2.83E-004	6.09E-002
11000	11000	11000	0.00E+000	0.00E+000	1.29E-004	6.20E-002
12000	12000	12000	3.91E-006	1.95E-006	1.43E-004	6.09E-002
13000	13000	13000	0.00E+000	3.13E-005	2.68E-004	6.15E-002
14000	14000	14000	2.50E-004	0.00E+000	7.23E-005	6.05E-002
15000	15000	15000	0.00E+000	0.00E+000	3.16E-004	6.10E-002

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
16000	16000	16000	1.56E-005	0.00E+000	1.43E-004	6.12E-002
17000	17000	17000	0.00E+000	3.91E-006	2.83E-004	6.06E-002
18000	18000	18000	5.00E-004	0.00E+000	3.48E-004	6.10E-002
19000	19000	19000	7.81E-006	3.91E-006	1.60E-004	6.08E-002
20000	20000	20000	0.00E+000	5.00E-004	5.72E-004	6.15E-002
30000	30000	30000	0.00E+000	1.95E-006	4.28E-004	6.41E-002
40000	40000	40000	2.54E-004	2.50E-004	7.25E-004	6.32E-002
50000	50000	50000	3.91E-006	2.58E-004	8.13E-004	6.15E-002
60000	60000	60000	3.13E-005	6.64E-005	1.06E-003	6.17E-002
70000	70000	70000	5.08E-004	7.58E-004	1.32E-003	6.21E-002
80000	80000	80000	8.02E-003	1.76E-005	2.24E-003	6.22E-002
90000	90000	90000	1.37E-004	9.57E-005	1.86E-003	6.27E-002
1e+05	1e+05	1e+05	6.64E-005	7.23E-005	1.99E-003	6.49E-002
2e+05	2e+05	2e+05	7.70E-003	5.76E-004	3.84E-003	6.38E-002
3e+05	3e+05	3e+05	3.13E-003	4.67E-004	5.18E-003	6.54E-002
4e+05	4e+05	4e+05	1.00E-003	1.50E-003	7.28E-003	6.67E-002
5e+05	5e+05	5e+05	2.79E-003	1.27E-003	9.62E-003	7.41E-002
6e+05	6e+05	6e+05	2.39E-003	1.40E-003	1.06E-002	6.98E-002
7e+05	7e+05	7e+05	7.02E-003	2.05E-003	1.30E-002	7.05E-002
8e+05	8e+05	8e+05	3.26E-003	2.14E-003	1.48E-002	7.21E-002
9e+05	9e+05	9e+05	2.89E-003	2.08E-003	1.82E-002	7.86E-002
1e+06	1e+06	1e+06	4.45E-003	2.52E-003	1.81E-002	7.36E-002

Table 9.5: Vector Sorting Timing Results

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
100	100	200	1.95E-006	0.00E+000	0.00E+000	3.19E-003
200	200	461	0.00E+000	0.00E+000	1.95E-006	3.34E-004
300	300	744	1.56E-005	6.25E-005	1.95E-006	3.38E-004
400	400	1041	1.95E-006	0.00E+000	0.00E+000	3.22E-004
500	500	1350	1.25E-004	2.50E-004	1.56E-005	8.30E-004
600	600	1667	3.91E-006	7.81E-006	6.25E-005	4.28E-004
700	700	1992	1.25E-004	1.25E-004	1.95E-006	7.13E-004
800	800	2323	1.95E-006	3.91E-006	3.32E-005	8.55E-004
900	900	2659	1.56E-005	1.56E-005	2.58E-004	8.57E-004
1000	1000	3000	6.25E-005	0.00E+000	3.32E-005	7.13E-004
2000	2000	6603	5.16E-004	5.16E-004	6.64E-004	5.00E-004
3000	3000	10432	2.83E-004	2.83E-004	4.34E-004	1.25E-003
4000	4000	14409	2.60E-004	3.22E-004	1.00E-003	1.03E-003
5000	5000	18495	5.84E-004	6.46E-004	1.27E-003	1.64E-003
6000	6000	22669	8.34E-004	3.34E-004	1.42E-003	1.67E-003
7000	7000	26916	7.09E-004	6.78E-004	1.97E-003	1.86E-003
8000	8000	31225	4.34E-004	4.28E-004	2.01E-003	1.48E-003
9000	9000	35589	9.67E-004	8.09E-004	2.16E-003	2.05E-003
10000	10000	40000	9.84E-004	8.75E-004	2.42E-003	2.00E-003
11000	11000	44456	1.00E-003	9.38E-004	3.00E-003	2.03E-003
12000	12000	48951	1.00E-003	1.00E-003	3.07E-003	2.57E-003
13000	13000	53482	1.02E-003	1.04E-003	3.83E-003	2.33E-003
14000	14000	58046	1.53E-003	1.13E-003	4.00E-003	2.93E-003
15000	15000	62642	1.20E-003	1.16E-003	4.03E-003	2.87E-003

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
16000	16000	67266	1.29E-003	1.65E-003	4.66E-003	2.69E-003
17000	17000	71918	1.33E-003	1.33E-003	4.93E-003	3.13E-003
18000	18000	76595	1.68E-003	2.18E-003	5.06E-003	3.14E-003
19000	19000	81297	1.46E-003	1.43E-003	5.32E-003	3.16E-003
20000	20000	86021	1.86E-003	2.25E-003	5.44E-003	3.67E-003
30000	30000	134314	2.48E-003	2.25E-003	8.93E-003	5.00E-003
40000	40000	184083	3.50E-003	3.98E-003	1.20E-002	6.16E-003
50000	50000	234949	4.88E-003	5.00E-003	1.51E-002	7.93E-003
60000	60000	286690	6.01E-003	6.03E-003	1.83E-002	9.26E-003
70000	70000	339157	7.28E-003	7.41E-003	2.20E-002	1.05E-002
80000	80000	392248	8.15E-003	8.59E-003	2.56E-002	1.20E-002
90000	90000	445882	9.70E-003	9.19E-003	2.83E-002	1.36E-002
1e+05	1e+05	500000	1.04E-002	1.09E-002	3.23E-002	1.49E-002
2e+05	2e+05	1060206	2.29E-002	3.00E-002	6.80E-002	2.93E-002
3e+05	3e+05	1643137	3.58E-002	3.52E-002	1.06E-001	4.30E-002
4e+05	4e+05	2240824	5.67E-002	4.91E-002	1.45E-001	5.78E-002
5e+05	5e+05	2849486	6.25E-002	6.29E-002	1.83E-001	7.22E-002
6e+05	6e+05	3466891	8.49E-002	7.72E-002	2.21E-001	8.58E-002
7e+05	7e+05	4091569	9.26E-002	9.20E-002	2.61E-001	1.00E-001
8e+05	8e+05	4722472	1.11E-001	1.05E-001	3.02E-001	1.15E-001
9e+05	9e+05	5358819	1.22E-001	1.29E-001	3.39E-001	1.27E-001
1e+06	1e+06	6000000	1.43E-001	1.43E-001	3.81E-001	1.43E-001
2e+06	2e+06	12602060	3.22E-001	2.90E-001	7.94E-001	2.80E-001
3e+06	3e+06	19431364	4.86E-001	4.52E-001	1.22E+000	4.20E-001

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
4e+06	4e+06	26408240	6.55E-001	6.22E-001	1.66E+000	5.60E-001
5e+06	5e+06	33494851	8.35E-001	8.25E-001	2.10E+000	7.02E-001
6e+06	6e+06	40668908	1.02E+000	9.94E-001	2.54E+000	8.45E-001
7e+06	7e+06	47915687	1.19E+000	1.18E+000	3.00E+000	9.83E-001
8e+06	8e+06	55224720	1.37E+000	1.38E+000	3.46E+000	1.12E+000
9e+06	9e+06	62588183	1.57E+000	1.57E+000	3.92E+000	1.28E+000
1e+07	1e+07	70000000	1.75E+000	1.76E+000	4.41E+000	1.42E+000
2e+07	2e+07	146020600	3.74E+000	3.67E+000	9.14E+000	2.84E+000
3e+07	3e+07	224313638	5.90E+000	5.80E+000	1.40E+001	4.27E+000
4e+07	4e+07	304082400	8.01E+000	8.10E+000	1.90E+001	5.70E+000
5e+07	5e+07	384948501	1.04E+001	1.04E+001	2.40E+001	7.12E+000
6e+07	6e+07	466689076	1.26E+001	1.27E+001	2.92E+001	8.55E+000
7e+07	7e+07	549156863	1.52E+001	1.48E+001	3.42E+001	9.96E+000
8e+07	8e+07	632247199	1.75E+001	1.75E+001	3.95E+001	1.14E+001
9e+07	9e+07	715881826	1.98E+001	1.99E+001	4.49E+001	1.28E+001
1e+08	1e+08	800000000	2.24E+001	2.24E+001	5.00E+001	1.42E+001

Table 9.6: Matrix Transpostion Timing Results

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
100 x 100	100 x 100	10000	0.00E+000	0.00E+000	0.00E+000	1.02E-001
200 x 200	200 x 200	40000	0.00E+000	0.00E+000	0.00E+000	6.10E-002
300 x 300	300 x 300	90000	0.00E+000	1.00E-003	0.00E+000	6.20E-002
400 x 400	400 x 400	160000	0.00E+000	2.00E-003	2.90E-002	6.40E-002
500 x 500	500 x 500	250000	3.00E-002	1.00E-003	2.00E-003	6.50E-002
600 x 600	600 x 600	360000	3.00E-002	3.00E-003	3.00E-003	6.60E-002
700 x 700	700 x 700	490000	3.20E-002	4.00E-003	4.00E-003	6.90E-002
800 x 800	800 x 800	640000	3.50E-002	6.00E-003	7.00E-003	7.30E-002
900 x 900	900 x 900	810000	3.60E-002	8.00E-003	8.00E-003	9.90E-002
1000 x 1000	1000 x 1000	1000000	3.80E-002	1.00E-002	1.20E-002	7.30E-002
2000 x 2000	2000 x 2000	4000000	7.10E-002	4.20E-002	5.80E-002	1.60E-001
3000 x 3000	3000 x 3000	9000000	1.40E-001	1.07E-001	1.35E-001	2.03E-001
4000 x 4000	4000 x 4000	16000000	2.38E-001	2.00E-001	2.51E-001	3.59E-001
5000 x 5000	5000 x 5000	25000000	3.84E-001	3.40E-001	4.17E-001	5.23E-001
6000 x 6000	6000 x 6000	36000000	6.27E-001	5.97E-001	6.60E-001	6.45E-001
7000 x 7000	7000 x 7000	49000000	8.76E-001	8.15E-001	9.64E-001	8.96E-001
8000 x 8000	8000 x 8000	64000000	1.21E+000	1.23E+000	1.39E+000	1.00E+000
9000 x 9000	9000 x 9000	81000000	1.58E+000	1.55E+000	1.73E+000	1.26E+000
10000 x 10000	10000 x 10000	100000000	1.96E+000	1.95E+000	2.14E+000	1.59E+000
11000 x 11000	11000 x 11000	121000000	2.45E+000	2.37E+000	2.79E+000	2.19E+000
12000 x 12000	12000 x 12000	144000000	3.11E+000	3.13E+000	3.44E+000	2.11E+000
13000 x 13000	13000 x 13000	169000000	3.67E+000	3.64E+000	3.86E+000	2.47E+000
14000 x 14000	14000 x 14000	196000000	4.25E+000	4.17E+000	4.71E+000	3.13E+000
15000 x 15000	15000 x 15000	225000000	4.88E+000	4.94E+000	5.36E+000	3.41E+000

INPUT	OUTPUT	CALC.COUNT	R	BC	CPP	GPU
16000 x 16000	16000 x 16000	256000000	6.39E+000	6.42E+000	9.97E+000	3.72E+000
17000 x 17000	17000 x 17000	289000000	6.48E+000	6.46E+000	6.84E+000	4.33E+000
18000 x 18000	18000 x 18000	324000000	7.30E+000	7.17E+000	8.29E+000	4.28E+000
19000 x 19000	19000 x 19000	361000000	8.04E+000	8.02E+000	8.80E+000	4.72E+000
20000 x 20000	20000 x 20000	400000000	9.51E+000	9.46E+000	1.13E+001	5.26E+000

Table 9.7: Matrix Multiplication Timing Results

INPUT	OUTPUT	CALC.COUNT	R	BC	ARMA	CPP	GPU
100 x 100	100 x 100	1000000	9.34E-004	7.36E-004	4.02E-004	2.33E-002	8.98E-004
200 x 200	200 x 200	8000000	6.19E-003	5.98E-003	5.08E-003	1.87E-001	1.00E-003
300 x 300	300 x 300	27000000	1.83E-002	1.83E-002	1.79E-002	6.29E-001	2.01E-003
400 x 400	400 x 400	64000000	4.20E-002	4.22E-002	4.16E-002	1.52E+000	2.43E-003
500 x 500	500 x 500	125000000	8.27E-002	8.12E-002	8.10E-002	2.97E+000	4.00E-003
600 x 600	600 x 600	216000000	1.43E-001	1.41E-001	1.39E-001	5.25E+000	5.67E-003
700 x 700	700 x 700	343000000	2.25E-001	2.24E-001	2.24E-001	8.37E+000	7.74E-003
800 x 800	800 x 800	512000000	3.42E-001	3.43E-001	3.42E-001	1.26E+001	1.00E-002
900 x 900	900 x 900	729000000	4.97E-001	4.93E-001	5.03E-001	1.84E+001	1.32E-002
1000 x 1000	1000 x 1000	1000000000	7.46E-001	7.12E-001	7.17E-001	2.45E+001	1.97E-002

Table 9.8: Cubesum Thread and Block Count

Vector Size	Thread Count	Block Count
100	128	1
200	256	1
300	256	2
400	256	2
500	256	2
600	256	3
700	256	3
800	256	4
900	256	4
1000	256	4
1100	256	5
1200	256	5
1300	256	6
1400	256	6
1500	256	6
1600	256	7
1700	256	7
1800	256	8
1900	256	8
2000	256	8
2100	256	9
2200	256	9
2300	256	9
2400	256	10

Table 9.9: Vector Addition Thread and Block Count

Threads	Blocks	Multiprocessors	Load	Threads	Blocks	Multiprocessors	Load
100	1	1	1	15000	59	10	1
200	1	1	1	16000	63	11	1
300	2	1	1	17000	67	12	1
400	2	1	1	18000	71	12	1
500	2	1	1	19000	75	13	1
600	3	1	1	20000	79	14	1
700	3	1	1	30000	118	20	2
800	4	1	1	40000	157	27	2
900	4	1	1	50000	196	33	3
1000	4	1	1	60000	235	40	3
2000	8	2	1	70000	274	46	4
3000	12	2	1	80000	313	53	4
4000	16	3	1	90000	352	59	4
5000	20	4	1	100000	391	66	5
6000	24	4	1	200000	782	131	9
7000	28	5	1	300000	1172	196	14
8000	32	6	1	400000	1563	261	18
9000	36	6	1	500000	1954	326	22
10000	40	7	1	600000	2344	391	27
11000	43	8	1	700000	2735	456	31
12000	47	8	1	800000	3125	521	35
13000	51	9	1	900000	3516	586	40
14000	55	10	1	1000000	3907	652	44

Table 9.10: GPU Transfer Timing Results

INPUT	OUTPUT	CALC.COUNT	TOTAL.ELAPSED	TO.DEVICE	TO.HOST	FUNCTION.ELAPSED
100	1	1	6.76E-004	8.17E-006	1.05E-005	4.16E-004
200	1	1	6.45E-005	8.38E-006	1.13E-005	6.45E-005
300	1	1	1.33E-004	8.74E-006	1.08E-005	3.32E-005
400	1	1	5.18E-004	8.62E-006	1.10E-005	2.58E-004
500	1	1	6.25E-005	9.14E-006	1.11E-005	6.45E-005
600	1	1	1.33E-004	9.04E-006	1.12E-005	5.33E-004
700	1	1	2.68E-004	9.31E-006	1.14E-005	5.02E-004
800	1	1	3.32E-005	9.31E-006	1.14E-005	1.76E-005
900	1	1	3.91E-006	9.62E-006	1.17E-005	3.32E-005
1000	1	1	2.85E-004	1.11E-005	1.52E-005	2.58E-004
2000	1	1	6.64E-005	1.11E-005	1.29E-005	6.64E-005
3000	1	1	1.29E-004	1.24E-005	1.46E-005	1.56E-005
4000	1	1	1.33E-004	1.35E-005	1.56E-005	3.32E-005
5000	1	1	2.60E-004	1.48E-005	1.72E-005	6.64E-005
6000	1	1	2.68E-004	1.63E-005	1.93E-005	2.68E-004
7000	1	1	2.68E-004	1.73E-005	1.97E-005	5.33E-004
8000	1	1	2.68E-004	1.85E-005	2.12E-005	6.25E-005
9000	1	1	1.35E-004	1.99E-005	2.25E-005	5.35E-004
10000	1	1	1.35E-004	2.12E-005	2.33E-005	5.33E-004
20000	1	1	5.66E-004	3.10E-005	3.59E-005	2.85E-004
30000	1	1	1.27E-004	4.12E-005	4.87E-005	6.43E-004
40000	1	1	5.72E-004	6.06E-005	6.21E-005	6.46E-004
50000	1	1	3.22E-004	6.28E-005	7.31E-005	6.66E-004
60000	1	1	5.80E-004	7.98E-005	8.66E-005	6.66E-004

INPUT	OUTPUT	CALC.COUNT	TOTAL.ELAPSED	TO.DEVICE	TO.HOST	FUNCTION.ELAPSED
70000	1	1	3.22E-004	9.67E-005	1.03E-004	4.20E-004
80000	1	1	1.62E-004	1.04E-004	1.09E-004	7.15E-004
90000	1	1	2.91E-004	1.14E-004	1.25E-004	9.34E-004
1e+05	1	1	3.32E-004	1.25E-004	1.35E-004	9.34E-004
2e+05	1	1	7.46E-004	2.46E-004	2.59E-004	1.33E-003
3e+05	1	1	1.00E-003	3.38E-004	3.63E-004	1.50E-003
4e+05	1	1	1.13E-003	4.12E-004	4.53E-004	4.10E-003
5e+05	1	1	1.57E-003	4.74E-004	4.99E-004	3.00E-003
6e+05	1	1	1.17E-003	5.44E-004	5.78E-004	3.17E-003
7e+05	1	1	1.86E-003	6.15E-004	6.28E-004	3.94E-003
8e+05	1	1	1.49E-003	6.61E-004	7.38E-004	4.67E-003
9e+05	1	1	1.81E-003	7.57E-004	7.97E-004	5.03E-003
1e+06	1	1	2.00E-003	8.00E-004	8.42E-004	7.93E-003
2e+06	1	1	3.59E-003	1.53E-003	1.51E-003	1.03E-002
3e+06	1	1	5.26E-003	2.23E-003	2.26E-003	1.50E-002
4e+06	1	1	6.86E-003	2.93E-003	2.88E-003	1.96E-002
5e+06	1	1	8.51E-003	3.62E-003	3.60E-003	2.45E-002
6e+06	1	1	9.35E-003	4.24E-003	4.26E-003	2.90E-002
7e+06	1	1	1.10E-002	5.04E-003	4.94E-003	3.38E-002
8e+06	1	1	2.13E-002	5.65E-003	1.30E-002	5.73E-002
9e+06	1	1	2.50E-002	6.38E-003	1.57E-002	6.43E-002
1e+07	1	1	2.74E-002	7.08E-003	1.73E-002	7.17E-002
2e+07	1	1	5.48E-002	1.40E-002	3.48E-002	1.42E-001
3e+07	1	1	8.14E-002	2.09E-002	5.20E-002	2.12E-001

Table 9.11: CPP Transfer Timing Results

INPUT	OUTPUT	CALC.COUNT	TRANSFER.TIME
100	1	1	0.00E+000
200	1	1	0.00E+000
300	1	1	0.00E+000
400	1	1	0.00E+000
500	1	1	0.00E+000
600	1	1	0.00E+000
700	1	1	0.00E+000
800	1	1	0.00E+000
900	1	1	0.00E+000
1000	1	1	0.00E+000
2000	1	1	6.25E-005
3000	1	1	7.81E-006
4000	1	1	0.00E+000
5000	1	1	1.95E-006
6000	1	1	0.00E+000
7000	1	1	0.00E+000
8000	1	1	0.00E+000
9000	1	1	5.00E-004
10000	1	1	0.00E+000
20000	1	1	0.00E+000
30000	1	1	0.00E+000
40000	1	1	0.00E+000
50000	1	1	1.95E-006
60000	1	1	0.00E+000

INPUT	OUTPUT	CALC.COUNT	TRANSFER.TIME
70000	1	1	0.00E+000
80000	1	1	3.91E-006
90000	1	1	0.00E+000
1e+05	1	1	0.00E+000
2e+05	1	1	0.00E+000
3e+05	1	1	0.00E+000
4e+05	1	1	0.00E+000
5e+05	1	1	0.00E+000
6e+05	1	1	0.00E+000
7e+05	1	1	0.00E+000
8e+05	1	1	0.00E+000
9e+05	1	1	0.00E+000
1e+06	1	1	0.00E+000
2e+06	1	1	0.00E+000
3e+06	1	1	0.00E+000
4e+06	1	1	0.00E+000
5e+06	1	1	0.00E+000
6e+06	1	1	0.00E+000
7e+06	1	1	3.12E-005
8e+06	1	1	0.00E+000
9e+06	1	1	0.00E+000
1e+07	1	1	1.95E-006
2e+07	1	1	0.00E+000
3e+07	1	1	0.00E+000

Chapter 10

Code

10.1 Timing Script

Below is an example of the timing scripts used, including examples of both Rcpp and .Call function usage.

```
# Necessary R Packages
library(Rcpp)
library(inline)
library(compiler)
library(mail)

# R example function
ex_r <- function(vec, iter) {
  #Code Here
}

# R bytecode example function
ex_bc <- cmpfun(ex_r)

# Example Armadillo C code. This is for matrix multiplication.
code <- '
 arma::mat A = Rcpp::as<arma::mat>(a);
 arma::mat B = Rcpp::as<arma::mat>(b);
 arma::mat C = A * B;
 return Rcpp::wrap(C);
'

# Armadillo code compilation
ex_arma <- cxxfunction(signature(a="numeric",b="numeric"), code,
                       plugin="RcppArmadillo", verbose=TRUE)

# Matrix Multiplication C code
cppFunction('
  NumericMatrix matmul_cpp(SEXP matA, SEXP matB) {
    Rcpp::NumericMatrix a(matA);
```

```
    Rcpp::NumericMatrix b(matB);
    Rcpp::NumericMatrix ab(a.nrow(), b.ncol());

    for(int i = 0; i < a.nrow(); i++) {
      for(int j = 0; j < b.ncol(); j++) {
        ab(i,j) = 0;
          for(int k = 0; k < a.ncol(); k++) {
            ab(i,j) += a(i,k)*b(k,j);
          }
        }
    }
    return ab;
  }'
)

# Example RCPP function. This is for vector addition.
cppFunction('
  NumericVector ex_cpp(SEXP vecA, SEXP vecB) {
    Rcpp::NumericVector a(vecA);
    Rcpp::NumericVector b(vecB);
    Rcpp::NumericVector ab(a.size());

    for(int i = 0; i < a.size(); i++) {
      ab(i) = a(i) + b(i);
    }
    return ab;
  }'
)

# External .Call initialization and cleanup functions
ex_init <- function() { dyn.load("ex.so")}
ex_cln  <- function() { dyn.unload("ex.so")}

# External .Call to C function
ex_cpp <- function(vec, iter) {
  # Setup code here
  .Call("ex_cpp", vec, iter, y)
  # Return code here
}

# External .Call to CUDA example function
ac_gpu <- function(vec, iter) {
  # Setup code here
  .Call("ex_gpu", vec, iter, y)
  # Return code here
}



# Example benchmarking code
ex_bench <- function(init=100, goal=1000, step=100, reps=10, wait=TRUE) {
```

```
# If necessary, give the user 20 seconds to lock the computer
if(wait) {
  tmp <- proc.time()[3]
  while((proc.time()[3]-tmp) <= 20) {}
}

# Iterate through sizes that are multiples of N, used for Artificial
# Complexity testing. If this was not used, the loop was removed.
# The elements iterated through have to be hard coded, but this was
# just for simplicity.
for(it in seq(0.01, 1, by=0.01)) {
  # Write output to the R benchmark csv file
  output <- "INPUT,OUTPUT,CALC.COUNT,N.FACTOR,ELAPSED\n"
  cat(output, file="ex_bench_r.csv", append=TRUE)

  # Iterate through the desired vector sizes. This changes
  # depending on whether multiple vectors, or matrices are
  # used.
  for(i in seq(init, goal, by=step)) {
    # Create a random vector
    A <- sample(5, i, replace=TRUE)

    # Next, time the execution of one of the example
# functions
    start <- proc.time()[3]
    ex_r(A, as.integer(floor((it*i)*i*i)))
    average <- proc.time()[3]-start
    # If desired, repeat the timing multiple times for better
    # accuracy
    if(reps > 1) {
      for(j in c(1:(reps-1))) {
        start <- proc.time()[3]
        ex_r(A, as.integer(floor((it*i)*i*i)))
        average <- (average+(proc.time()[3]-start))/2
      }
    }

    # Format the results and write them to the csv file.
    # The exact output from this was changed to accommodate
    # the varying test outputs.
    input <- toString(i)
    out <- toString(1)
    ccount <- toString(i)
    nfac <- toString(as.integer(floor((it*i)*i*i)))
    elapsed <- toString(average)
    output <- c(input, ",", out, ",", ccount, ",", nfac, ",", elapsed, "\n")
    cat(output, file="ex_bench_r.csv", append=TRUE)
    rm(A)
    print(toString(i/(goal/it)))
  }
```

```
    # Repeat the process for the bytecode, CPP, and CUDA functions
  }
}
```

10.2 CUDA and C/C++ Code

Below is an example of external function code, for both CUDA and C/C++ functions.

```
// CUDA Library
#include<CUDA.h>

// R Libraries
#include <R.h>
#include<Rinternals.h>
#include<Rmath.h>
#include<R_ext/BLAS.h>

// C sorting
#include<vector>
#include<algorithm>
#include<numeric>
#include<math.h>

// sum: Sequential sum code
int sum(int *v, int l) {
    int retVal = 0;
    for(int i = 0; i < l; i++) {
        retVal += v[i];
    }
    return retVal;
}

// vecCopy: Copies size elements from in to out
void vecCopy(int *in, int *out, int size) {
    for(int i = 0; i < size; i++) {
        out[i] = in[i];
    }
}

// Parallel vector reduction
__global__ void reduce(int *idata, int *odata, int numElements, int iter) {
    extern __shared__ int temp[];
    unsigned int i = threadIdx.x;
    unsigned int j = blockDim.x*blockIdx.x;

    for(int x = 0; x < iter; x++) {
        // Handle parallel reduction where the full block is involved.
        if(numElements-j > blockDim.x) {
            // Load elements into shared memory
```

```
            temp[i] = idata[i+j];
            __syncthreads();

            // Sum the block's contents
            for(unsigned int stride = blockDim.x>>1; stride > 0; stride >>=1) {
                __syncthreads();
                if(i < stride) {
                    temp[i] += temp[i+stride];
                }
            }
        }
        // Handle parallel reduction on the final, incomplete block
        else {
            // Load elements into shared memory
            if(i+j < numElements) {temp[i] = idata[i+j];}
            else    {temp[i] = 0;}
            __syncthreads();

            // Sum the block's contents
            for(unsigned int stride = blockDim.x>>1; stride > 0; stride >>=1) {
                __syncthreads();
                if(i < stride && i+j+stride < numElements) {
                    temp[i] += temp[i+stride];
                }
            }
        }
        __syncthreads();
    }

    // Write the results back to output
    if(i == 0) {
        odata[blockIdx.x] = temp[i];
    }
}

extern "C" SEXP ac_gpu(SEXP v, SEXP i, SEXP ret) {
    // Error checking
    int numElements = length(v);
    if(numElements <= 1) {
        return R_NilValue;
    }

    // Data and memory setup
    const int memSize = numElements*sizeof(int);
    int *temp = INTEGER(v);
    int *iter = INTEGER(i);
    int *h_idata = (int *)malloc(memSize);
    vecCopy(temp, h_idata, numElements);
    int *h_odata = (int *)malloc(memSize);
    int *d_idata, *d_odata;
    cudaMalloc((void**)&d_idata, memSize);
```

```
    cudaMalloc((void**)&d_odata, memSize);

    // Copy input data to device
    cudaMemcpy(d_idata, h_idata, memSize, cudaMemcpyHostToDevice);

    // Loop over the data until a single value has been returned
    bool flip = false;
    while(numElements > 1) {
        int i = 0;
        int g = 0;
        // Calculate the block size
        for(i = 2; i <= 256; i <<=1) {if(numElements < i){break;}}
        // Calculate the grid size
        if(numElements % i == 0) {
            g = ((int)floor(numElements/i));
        } else {
            g = ((int)floor(numElements/i))+1;
        }
        dim3 grid(g, 1), block(i,1);

        // Run the kernel
        if(flip) {reduce<<<grid, block, i*sizeof(int)>>>(d_odata, d_idata,
                                                         numElements, *iter);}
        else {reduce<<<grid, block, i*sizeof(int)>>>(d_idata, d_odata,
                                                         numElements, *iter);}

        // Update the control variables for the next iteration
        numElements = g;
        flip = !flip;
    }
    // Retrieve the results
    if(!flip) {
        cudaMemcpy(h_odata, d_idata, memSize, cudaMemcpyDeviceToHost);
    } else {
        cudaMemcpy(h_odata, d_odata, memSize, cudaMemcpyDeviceToHost);
    }

    // Copy the result into the return variable
    int *retVal = INTEGER(ret);
    *retVal = (int)h_odata[0];

    // Cleanup
    free(h_idata);
    free(h_odata);
    cudaFree(d_idata);
    cudaFree(d_odata);

    // Return the nil value
    return R_NilValue;
}
```

```
// singlesum_cpu
// Sequential sum on the cpu
extern "C" SEXP ac_cpp(SEXP v, SEXP i, SEXP ret) {
    // Initialize data
    int numElements = length(v);
    int *vec = INTEGER(v);
    int *iter = INTEGER(i);
    std::vector<int> temp(numElements);
    for(int i = 0; i < numElements; i++) {
        temp[i] = vec[i];
    }

    // Sum vec using accumulate from the algorithm library
    // and store the result in the return variable
    int *retVal = INTEGER(ret);
    int average = std::accumulate(temp.begin(), temp.end(), 0);
    for(int i = 1; i < *iter; i++) {
        average = std::accumulate(temp.begin(), temp.end(), 0);
    }
    *retVal = average;

    // Return a nil value
    return R_NilValue;
}
```

Bibliography

[1] *The Comprehensive R Archive Network*: http://cran.us.r-project.org/

[2] *CRAN High-Performance and Parallel Computing in R*: http://cran.r-project.org/web/views/HighPerformanceComputing.html

[3] *CUDA 4.2 SDK*: https://developer.nvidia.com/cuda-toolkit-42-archive

[4] *CUDA Thrust Library*: http://docs.nvidia.com/cuda/thrust/

[5] *C++ Algorithm Library*: http://www.cplusplus.com/reference/algorithm/

[6] *GPUTOOLS*: http://cran.r-project.org/web/packages/gputools/index.html

[7] *R+GPU*: http://brainarray.mbni.med.umich.edu/brainarray/rgpgpu/

[8] *PLASMA*: http://icl.eecs.utk.edu/plasma/

[9] *MAGMA*: http://icl.eecs.utk.edu/magma/

[10] *fork*: http://cran.r-project.org/web/packages/fork/index.html

[11] *Rdsm*: http://cran.r-project.org/web/packages/Rdsm/index.html

[12] Xiaosong Ma; Jiangtian Li; Samatova, N.F., "Automatic Parallelization of Scripting Languages: Toward Transparent Desktop Parallel Computing," Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International , vol., no., pp.1,6, 26-30 March 2007 doi: 10.1109/IPDPS.2007.370488

[13] Jiangtian Li, Xiaosong Ma, Srikanth Yoginath, Guruprasad Kora, and Nagiza F. Samatova. 2011. Transparent Runtime Parallelization of the R Scripting Language. J. Parallel Distrib. Comput. 71, 2 (February 2011), 157-168. DOI=10.1016/j.jpdc.2010.08.013 http://dx.doi.org/10.1016/j.jpdc.2010.08.013

[14] Yulong Ou; Bo Li; Hailong Yang; Zhongzhi Luan; Depei Qian, "Efficient Statistical Computing on Multicore and MultiGPU Systems," Network-Based Information Systems (NBiS), 2012 15th International Conference on , vol., no., pp.709,714, 26-28 Sept. 2012 doi: 10.1109/NBiS.2012.89