Parallel Computing Using GPUs

Haizhen Wu

March 1, 2011
Summary

Many computational intensive algorithms, such as Markov Chain Monte Carlo (MCMC) in Statistics and evolutionary computation in Computer Science, are essentially parallel or parallelizable algorithms. On the other hand, Graphics Processing Units (GPUs), which were originally designed to handle graphic related tasks, are naturally massively parallel structures and with huge potential in (massively) parallel computing. Driven by the technology leaders such as Nvidia, this General-Purpose computation on Graphics Processing Units (GPGPU) approach is getting rapid popularity than other parallel computing methods, and its applications in a wide range of research fields are also expanding dramatically.

The purpose of this manual is to show the impressive potential speed-up of GPU computing in many complex scientific computing problems, introduce the most popular GPGPU architecture - Compute Unified Device Architecture (CUDA), provide detailed instructions for setting up a CUDA system and programming under CUDA C, and explore the applications in statistical computing through some real examples.

The structure of this manual is as below:

In Chapter 1, we introduce the motivation of turning from serial computing to parallel computing and the advantages of GPGPU compared to other methods. Then we introduce the CUDA architecture, the method of setting up a CUDA system, and the access to CUDA-enabled systems over the School of Engineering and Computer Science (ECS)/School of Mathematics, Statistics and Operations Research (MSOR) network at Victoria University of Wellington (VUW). In Chapter 3, we introduce basic CUDA C programming and discuss the integration of CUDA to other languages. Finally, in Chapter 4, we show some applications of GPU computing in Statistics based on real examples.
Contents

List of Figures iii
Abbreviations iii

1 Parallel Computing and GPGPU 1
   1.1 Introduction .............................................. 1
   1.2 Parallel Computing vs Sequential Computing .............. 2
   1.3 Amdahl's law ............................................ 3
   1.4 Development of parallel computing architecture .......... 4

2 CUDA Architecture and System Build-up 5
   2.1 CUDA Architecture ........................................... 5
       2.1.1 Advantages of CUDA Architecture .................. 5
       2.1.2 Typical CUDA Architecture - Tesla 10 series ....... 6
   2.2 Setting up a CUDA System ................................ 6
   2.3 Software Configuration .................................... 7
   2.4 Access to Red-tomatoes over ECS/MSOR network .......... 8

3 Basic CUDA C programming 9
   3.1 Model of CUDA Programming ............................... 9
       3.1.1 Host and Device ..................................... 9
       3.1.2 Thread, Block and Grid ............................. 9
       3.1.3 Scalability ....................................... 11
   3.2 Basics of CUDA C programming ........................... 11
       3.2.1 Host, Kernel and Device Functions ................. 12
       3.2.2 Memory Management .................................. 13
       3.2.3 Example - Moving Arrays ........................... 14
       3.2.4 Error handling ................................... 17
   3.3 CUDA libraries ........................................... 17
   3.4 Integration of CUDA with Other Languages ............... 17
       3.4.1 Integrating CUDA C with R ..................... 18
       3.4.2 Integration of Other languages with CUDA ....... 21
       3.4.3 Pros and Cons of integration ...................... 22

4 GPU Computing in Statistics 23
   4.1 Random Numbers Generation on GPU ...................... 23
   4.2 Random Walk Generation on GPU .......................... 23
   4.3 Metropolis-Hastings Markov Chain Monte Carlo ........... 26
List of Figures

1.1 Performance of GPU and CPU .............................................. 2
1.2 Serial loop and parallel threads ......................................... 3

2.1 Typical CUDA architecture - Tesla 10 series .......................... 6
2.2 Hardware Choices .......................................................... 6
2.3 Device Query Result of Tesla T10 on red-tomatoes .................. 8

3.1 Host and Device ............................................................. 9
3.2 Thread ............................................................................ 10
3.3 Execution of typical CUDA program ..................................... 10
3.4 Block ............................................................................. 11
3.5 Scalability ........................................................................ 11
3.6 Comparison between C and CUDA C code ............................ 12
3.7 Data Copies between CPU and GPU .................................... 14
3.8 Data Copies between CPU and GPU - Code ......................... 14
3.9 CUDA Software Development Architecture ........................... 17
3.10 CUDA C function to be wrapped in R ................................. 19
3.11 CUDA C function to be wrapped in R (Cont) ....................... 20
3.12 R code to wrap and call CUDA C function ......................... 21

4.1 Random Walk Generation on GPU ....................................... 24
4.2 Comparison Between CPU and GPU ..................................... 24
4.3 Random Walk Generation on GPU - CUDA C code ............... 25
4.4 Metropolis-Hastings Markov Chain Monte Carlo ................... 26
4.5 MCMC on GPU ............................................................... 26
4.6 MCMC on GPU - CUDA C code ......................................... 27
Abbreviations

API Application Programming Interface.
CUDA Compute Unified Device Architecture.
ECS School of Engineering and Computer Science.
FFT Fast Fourier Transform.
FLOPS FLoating point OPerations per Second.
GPGPU General-Purpose computation on Graphics Processing Units.
GPUs Graphics Processing Units.
MCMC Markov Chain Monte Carlo.
MSOR School of Mathematics, Statistics and Operations Research.
VUW Victoria University of Wellington.
Chapter 1
Parallel Computing and GPGPU

1.1 Introduction

The speed of computation plays a vital role in many fields of quantitative sciences. For example, several seconds advancing in earthquake early warning may save dozens of lives, while in financial markets, a tenth of a second can mean hundreds of thousands of dollars. There are also many scientific problems which can be solved in theory, but are impossible in practice without sufficiently powerful computer. One striking example comes from a genetic programming problem describe in [1]: for certain level of complexity (20-bits Multiplexor), the problem can be solved with estimated time of 4 years, while for a more complicated problem (37-bits Multiplexer), it has never been attempted before.

Moore’s Law is an empirical law proposed by Gordon Moore in 1965, which claimed that the transistor density in a microprocessor doubles every 18 to 24 months. In the 1990’s, Moore’s law has been used to describe the increase in CPU power or speed. Although Moore’s law works well so far, it has expected to stop in the near future due to the physical and economical limitations. Bill Dally, the chief scientist and senior vice president of research at Nvidia, has claimed that Moore’s Law is going to die for CPUs. Instead, the GPUs will play the key role on the future computing performance improvement.

Figure 1.1 shows that performance of GPUs, in terms of FLoating point OPerations per Second (FLOPS), has been about one order of magnitude higher than CPUs in recent years. It is reasonable to predict that, as the slowing down of Moore’s law on CPUs, the difference of computational performance between GPUs and CPUs will increase dramatically in the near future.
1.2 Parallel Computing vs Sequential Computing

The reason why a GPU can achieve high performance is due to its (massively) parallel structure. In contrast to CPUs with only one or at most several processors/cores, a GPU consists of hundreds, even thousands of multi-processors/cores. Although the performance of each multi-processor/core is usually inferior to CPU, the overall performance can outreach it. However, this performance improvement only applies to those parallel algorithms.

Traditionally, computer software has been written serially. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed sequentially on computer. Only one instruction can execute at a time and after that instruction is finished, the next is executed.

Parallel computing, on the other hand, uses many parallel processing streams to solve a problem. This is realized by breaking down the problem into parallel parts such that each part of the algorithm can be executed simultaneously and often independently. This parallel computing can be implemented by a single computer with multiple processors, several networked computers, specialized hardware, or the combination of the above.

Apart from hardware support, the realization of parallel computing depends on the nature of the problem. Only parallelizable algorithms can take advantage of parallel computing, which, fortunately, covers a large proportion of scientific computation. A far-from-complete list includes:

- Vector and matrix operations
- Image processing
- Simulation of large number of random variable/ random processes
- Fast Fourier transform
• SQL database operations
• Markov Chain Monte Carlo simulation
• Genetic programming

Typically, the parallelizable algorithms in sequential computing are usually written as serial loop; while in parallel computing, many parallel threads can be excited simultaneously.

Figure 1.2: Flowchart of serial loop and parallel threads.

1.3 Amdahl’s law

The speed-up of turning sequential computing to parallel computing is governed by Amdahl’s law. If we denote $P$ the proportion of the execution time which can be parallelized and $N$ the number of parallel threads available, then Amdahl’s law implies, the speed-up of parallel computing compare to sequential computing is:

$$S = \frac{1}{1 - P + \frac{P}{N}}.$$

For example, if the parallelizable proportion accounting for 95% of the whole algorithm and the speed-up is 18.5; while for algorithms with 99% parallelizable proportion, the speed-up could reach 70.8. This suggests that the key of performance improvement in parallel computing is the parallelizable proportion of the algorithm.
It is also noteworthy that as the number of parallel threads $N$ increases, there is a limit speed-up, which is $1/(1 - P)$. This implies that a finite number of parallel threads will always be sufficient to achieve best performance for many parallel computing applications.

1.4 Development of parallel computing architecture

Originally, computers had only one single-core processor and didn’t have any parallel computing ability. Later on, with the demand for parallel tasks, Multi-core/Multiprocessor computers have been emerged. Since the number of processors is limited in this architecture (typically 2 or 4), only very limited parallel computing ability can be provided. Another popular parallel computing architecture, which has been developed for many years, is grid computing.

Grid computing refers to utilising many computers from multiple administrative domains to implement a single specific parallelizable computing task. The number of computers in a grid computing architecture can be dozens to hundreds, which provide sufficient parallel threads for some parallel computing algorithms. However, one distinct deficiency of grid computing is its non-interactive feature, which restricted the application of grid computing from many parallel computing algorithms with interaction among threads.

However, the modern graphics processor (GPU) on video cards has evolved into an extremely powerful and flexible processor. This has enabled the architecture of General Purpose Computation on Graphics Processing Units (GPGPU) to be established, and provided massively parallel computing ability with certain level of interaction among threads.
Chapter 2

CUDA Architecture and System Build-up

2.1 CUDA Architecture

CUDA (Compute Unified Device Architecture) is one of the most popular GPGPU architecture introduced by NVIDIA in 2007. It is designed from the ground-up for efficient general purpose computation on GPUs.

2.1.1 Advantages of CUDA Architecture

Compare to previous GPGPU architectures, CUDA features the following advantages:

• It provides a minimal extension to the familiar C/C++ environment. Developers can compile C for CUDA to avoid the tedious work of remapping their algorithms to graphics card concepts.

• It supports Heterogeneous Computing (computing with both CPU and GPU). For most algorithms, only parts of them are parallelizable. Since the speed of CPUs are usually much faster than individual GPUs processors, it will more efficient to execute the parallelizable parts on GPUs, while allocate the non-parallelizable parts to CPUs. CUDA is the first GPGPU architecture which is designed to support joint CPU/GPU execution of an application.

• CUDA architecture has good scalability. The programs written and tested in a lower end GPUs can be easily extended to higher end GPUs.

• Both a low level Application Programming Interface (API) and a high level API are available in CUDA architecture, which provide flexibility to experienced programmer as well as a relatively gentle learning curve to the beginner.

• There are also expanding development tools and libraries in CUDA to help controlling GPUs through existing functions, which reduces programming needs enormously.
2.1.2 Typical CUDA Architecture - Tesla 10 series

Figure 2.1 shows a typical CUDA architecture. It consists of an array of highly threaded streaming multiprocessors (SMs). Each SM has a number of streaming processors (SPs) that share control logic and instruction cache and each SP can run thousands of threads in applications.

![Figure 2.1: Typical CUDA architecture - Tesla 10 series.](image)

The Tesla 10 has 240 SPs (30 SMs, each with 8 SPs). Suppose we have a parallel task requiring execution of \( n = 1920 \) jobs, we might group every 32 threads into a block, then the number of blocks to be executed is \( B = 60 \). Since we have 30 SMs with each containing 8 SPs, the number of blocks in each SM is \( 60/30 = 2 \) and the number of threads executed in each SP is \( 1920/240 = 8 \).

2.2 Setting up a CUDA System

Setting up a CUDA system is relatively straightforward, there are several approaches to establish a CUDA-enabled system: You can either buy a dedicated CUDA-Computing system which was designed from the ground up for high performance computing, such as The NVIDIA Tesla 20-series, or buy an additional CUDA-enabled GPU to install on to an existing computer. In fact, many computers already have CUDA-enabled GPUs: such as all Apple laptop or desktop computers.

![Figure 2.2: Hardware Choices](image)
However, there are some specifications which need particular attention: such as the number of threads per block, amount of local memory per thread and number of instructions per kernel, etc. More detailed descriptions of the compute capability can be found in Appendix G of [2]. Beware that only CUDA devices with compute capability greater than version 1.3 support double-precision instructions.

2.3 Software Configuration

The CUDA architecture supports all three of the most popular operating systems - Windows, Mac OS and Linux. Although there are slight differences in installation on different operating systems, the basic steps include:

- Verify the system has a CUDA-enabled GPU and install the NVIDIA GPU driver.
- Verify that a C/C++ compiler is available (gcc on MacOS and Linux, MS Virtual C/C++ on Windows).
- Install CUDA software (CUDA toolkit/GPU computing SDK).
- Verify the Installation.

The detailed installation instructions can be found in [3, 4, 5]

After installation, running the SDK example deviceQuery should give the following output (Figure 2.3), which shows the specification of the CUDA device.
2.4 Access to Red-tomatoes over ECS/MSOR network

There is a sever (red-tomatoes), which includes a CUDA-enabled GPU, in ECS (School of Engineering and Computer Science, Victoria University of Wellington) workstation. The instructions for accessing this sever can be found here: http://homepages.ecs.vuw.ac.nz/~kevin/Tesla_C1060/TeslaGpuKMBnote1.html
Chapter 3

Basic CUDA C programming

3.1 Model of CUDA Programming

As mentioned, the CUDA architecture supports heterogeneous computing with both CPU and GPU involved. This introduces a more complicated computing model than traditional CPU computing.

3.1.1 Host and Device

In CUDA architecture, the host refers to the CPU while the device refers to GPUs. Apart from executing serial instructions, the CPU is also responsible for managing and allocating instructions and data. Figure 3.1 shows the connection between host and device. As a device, GPUs execute the part of the parallelizable application, which is allocated by CPU. Both host and device have their own memory space. When a set of data needs to be processed by the GPU, it firstly allocated in host memory, and then copied to the device. After processing, the result will be copied to device memory and then transferred back to the host.

3.1.2 Thread, Block and Grid

The parallel portions of a program executed on GPU are called kernels. A CUDA kernel is executed by an array of threads, which run the same code but each of which has

Figure 3.1: Host and Device.
distinct ID that it uses to compute memory addresses and make control decisions. One kernel is executed at a time on the GPU.

Figure 3.2: Threads are the separate tasks executed by the individual core on the GPU.

The execution of a typical CUDA program in GPUs is illustrated in Figure 3.3. The execution starts with host (CPU) execution. When a kernel is launched in GPU, a large number of threads are generated to implement parallel computing, and they are collectively called a grid. When all threads of a kernel complete, the corresponding grid terminates, and the execution continues on the host until another kernel is executed.

Figure 3.3: Execution of typical CUDA program

The grids are usually divided into many blocks, which are the groups of threads, to achieve scalability. The maximum size of a block is specified by the compute capability of the device, and for the Tesla T10 Processor, this number is 512.
The threads can communicate and cooperate with each other. This is a powerful feature of CUDA. Cooperation between threads of different blocks is not scalable but cooperation within smaller batches of threads is scalable.

### 3.1.3 Scalability

For programs not allowing threads communication with each other in different blocks, the CUDA runtime system can execute blocks in any order. This flexibility enables scalable implementations as shown in Figure 3.5. For a program with parallel part divided into 8 blocks, it can be run on a low-end GPU running 2 blocks at a time, or can be run on a high-end device by running 4 blocks at a time, without any rewriting code.

### 3.2 Basics of CUDA C programming

CUDA C is similar to the normal C programming language in its basic syntax. However, there are some extensions such that CUDA C can control the GPU device. Figure 3.6 shows difference between C and CUDA C code for a simple task in which every element of an array \( a \) is incremented by a scalar value \( b \).
3.2.1 Host, Kernel and Device Functions

There are three types of functions which play important roles in CUDA C programming. The **host** functions are those function executed on the CPU. The **kernel** functions specify the code to be executed on the GPU, but are called by the host functions. The kernel functions play the roles of the interface between the CPU and the GPU. There is another type of function, which executed on GPU and can only be called by those functions running on GPUs and is called a **device** functions. The following code shows these three types of functions.
Note that the kernel execution configuration parameters surrounded by `<<<` and `>>>`. These execution configuration parameters specified the dimension of the grid (number of blocks in each grid) and dimension of each block (number of threads in each block).

### 3.2.2 Memory Management

Unlike CPU computing with unique memory space in CUDA, the GPU and the CPU both have their own memory space (see Figure 3.1), which are separate from each other. The basic steps of transferring data are:

- Allocate memory for data on host and device;
- Copy data to device;
- Process data in device;
- Copy data from device;
- Free memory on both host and device.

The following figures (Figure 3.7 and 3.8) show the conceptual scheme and the CUDA C code for these steps.
3.2.3 Example - Moving Arrays

The following shows an example that the CUDA code of moving an array from host to device, updating, and then moving back to the host.
// moveArrays.cu
//
// demonstrates CUDA interface to data allocation on device (GPU)
// and data movement between host (CPU) and device.

#include <stdio.h>
#include <assert.h>
#include <cuda.h>

int main(void) {
    float *a_h, *b_h;  // pointers to host memory
    float *a_d, *b_d;  // pointers to device memory
    int N = 14;
    int i;
    // allocate arrays on host
    a_h = (float *)malloc(sizeof(float)*N);
    b_h = (float *)malloc(sizeof(float)*N);
    // allocate arrays on device
    cudaMalloc((float **)&a_d, sizeof(float)*N);
    cudaMalloc((float **)&b_d, sizeof(float)*N);
    // initialize host data
    for (i=0; i<N; i++) {
        a_h[i] = 10.f + i;
        b_h[i] = 0.f;
    }
    for (i=0; i<N; i++)
        printf("%f ", a_h[i]);
    printf("\n");
    for (i=0; i<N; i++)
        printf("%f ", b_h[i]);
    printf("\n");
    // send data from host to device: a_h to a_d
    cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
    // copy data within device: a_d to b_d
    cudaMemcpy(b_d, a_d, sizeof(float)*N, cudaMemcpyDeviceToDevice);
    // retrieve data from device: b_d to b_h
    cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    // check result
    for (i=0; i<N; i++)
        if (a_h[i] == b_h[i]) printf("y");
        else printf("n");
    printf("\n");
    for (i=0; i<N; i++)
        printf("%f ", b_h[i]);
    printf("\n");
    // cleanup
    free(a_h); free(b_h);
    cudaFree(a_d); cudaFree(b_d);
}
/* Error Handling Example*/
// includes, system
#include <stdio.h>
#include <assert.h>

// Simple utility function to check for CUDA runtime errors
void checkCUDAError(const char* msg);

// Part3: implement the kernel
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    int inOffset = blockDim.x * blockIdx.x;
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int in = inOffset + threadIdx.x;
    int out = outOffset + (blockDim.x - 1 - threadIdx.x);
    d_out[out] = d_in[in];
}

/////////////////////////////////////////////////////////////////////
// Program main
/////////////////////////////////////////////////////////////////////
int main( int argc, char** argv)
{
    // pointer for host memory and size
    int *h_a;
    int dimA = 256 * 1024; // 256K elements (1MB total)

    // pointer for device memory
    int *d_b, *d_a;

    // define grid and block size
    int numThreadsPerBlock = 256;

    // Part 1: compute number of blocks needed based on
    // array size and desired block size
    int numBlocks = dimA / numThreadsPerBlock;

    // allocate host and device memory
    size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
    h_a = (int *) malloc(memSize);
    cudaMalloc((void **) &d_a, memSize);
    cudaMalloc((void **) &d_b, memSize);

    // Initialize input array on host
    for (int i = 0; i < dimA; ++i)
    {
        h_a[i] = i;
    }

    // Copy host array to device array
    cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice );

    // launch kernel
    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);
    reverseArrayBlock<<< dimGrid, dimBlock >>>( d_b, d_a );

    // block until the device has completed
    cudaThreadSynchronize();

    // check if kernel execution generated an error
    // Check for any CUDA errors
    checkCUDAError("kernel invocation");

    // device to host copy
    cudaMemcpy( h_a, d_b, memSize, cudaMemcpyDeviceToHost );

    // Check for any CUDA errors
    checkCUDAError("memcpy");

    // verify the data returned to the host is correct
    for (int i = 0; i < dimA; i++)
    {
        assert(h_a[i] == dimA - 1 - i );
    }

    // free device memory
    cudaFree(d_a);
    cudaFree(d_b);

    // free host memory
3.2.4 Error handling

The code above shows an example of handling error in CUDA C program.

3.3 CUDA libraries

Apart from the standard libraries, CUDA also provides several useful libraries, which can improve the programming efficiency significantly. These include:

- CURAND library, which provides facilities that focus on the simple and efficient generation of high-quality pseudo-random and quasi-random numbers. A pseudo-random sequence of numbers satisfies most of the statistical properties of a truly random sequence but is generated by a deterministic algorithm. A quasi-random sequence of n-dimensional points is generated by a deterministic algorithm designed to fill an n-dimensional space evenly([6]).

- CUBLAS, an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime. It allows access to the computational resources of NVIDIA GPUs. The library is at the low API level: it can control the GPUs directly and doesn’t need to call other CUDA driver functions. However, CUBLAS attaches only to a single GPU and does not automatically work across multiple GPUs([7]).

- CUFTT, The CUFFT library provides a simple interface for computing parallel Fast Fourier Transform (FFT) on an NVIDIA GPU, which allows users to leverage the floating point power and parallelism of the GPU without having to develop a custom, GPU based FFT implementation (8).

3.4 Integration of CUDA with Other Languages

Figure 3.9: CUDA Software Development Architecture.
From Figure 3.9, we can see that besides CUDA C, CUDA also supports OpenCL and DirectX Compute and many other high level programming languages such as C/C++, Fortran, Java, Python, and the Microsoft .NET Framework.

3.4.1 Integrating CUDA C with R

R[9] is a cross-platform programming language and software environment for statistical computing and graphics. After its appearance in 1993, the R language has become a de facto standard among statisticians for developing statistical software and is widely used for statistical software development and data analysis. Hence, the integration of CUDA C with R is of great interest in the field of Statistics. There are several R packages which provided functionality of GPU computing. The principle of the integrating CUDA C with R is: first write a function with CUDA C, then compile the CUDA C code into a shared library, link this shared library to R and finally write a R function to wrap the CUDA C function [10, 9].

The following function is a CUDA C function (Random.Walk.simulation.GPU.cu), which can be compiled and link to R.
/*
* Simulate Random Walk on GPU computing by CUDA C
* Update: 05/12/2010
*/

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>
#include <R.h>

// Simple utility function to check for CUDA runtime errors
void checkCUDAError(const char* msg);

//Thread Per Block
#define THREADS_PER_BLOCK 256

//define kernel functions
//this kernal function simulate a random Processes in device
__global__ void kernel_simulation(unsigned long long seed,
     double* Data, int NumChain, int NumStep, int Thin)
{
    int id = threadIdx.x + blockIdx.x * THREADS_PER_BLOCK;
    /* Each thread gets same seed, a different sequence number,
     * no offset */
    curandState localState;
    curand_init(seed, id, 0, &localState);

    int i, j;
    double increment;
    double Stddev = 1 / sqrtf((NumStep - 1) * Thin);

    Data[id * NumStep + 0] = 0;
    for (i = 1; i < NumStep; i++)
    {
        increment = 0;
        for (j = 0; j < Thin; j++)
        {
            increment += Stddev * curand_normal_double(&localState);
        }
        Data[id * NumStep + i] = Data[id * NumStep + i - 1] + increment;
    }
}

Figure 3.10: CUDA C function to be wrapped in R
extern "C" {
void RandWalkGPU(int *pNumChain, int *pNumStep, int *pThin, double *pDataMatrix) {

  int NumChain=*pNumChain;
  int NumStep=*pNumStep;
  int Thin=*pThin;
  double *h_Data=pDataMatrix;

  /*allocate space memory for a random Processes on device*/
  double* d_Data;
  size_t size=NumChain*NumStep*sizeof(double);
  cudaMalloc((void **)&d_Data,size);
  cudaMemcpy(d_Data,h_Data,size,cudaMemcpyHostToDevice);

  //define the block dimension each chain simulated in 1 thread
  int blocksPerGrid=(NumChain+THREADS_PER_BLOCK-1)/THREADS_PER_BLOCK;

  //set seed
  unsigned long long seed=time(NULL);

  kernel_simulation<<<blocksPerGrid,THREADS_PER_BLOCK>>>(seed, d_Data,NumChain,NumStep,Thin);

  // check if kernel execution generated an error
  checkCUDAError("kernel invocation");
  cudaMemcpy(h_Data,d_Data,size,cudaMemcpyDeviceToHost);

  // Check for any CUDA errors
  checkCUDAError("memcpy");

  // Free device memory
  cudaFree(d_Data);

  //return EXIT_SUCCESS;
}
} //extern "C"

void checkCUDAError(const char *msg) {
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err) {
    fprintf(stderr, "Cuda error: %s: %s.
", msg, cudaGetErrorString( err ));
    exit(EXIT_FAILURE);
  }
}

Figure 3.11: CUDA C function to be wrapped in R (Cont)

Figure 3.12 shows the structure of wrapper function in R. The code
compile CUDA C function to a shared library file using CUDA C compiler “nvcc”. This is equivalent to

R CMD SHLIB *.c

in compiling normal C function to shared library to be linked by R.

```
system("nvcc -arch=sm_13 -I/usr/share/R/include -Xcompiler -fpic -g -02 -c
Random.Walk.simulation.GPU.cu -o Random.Walk.simulation.GPU.o")
```

Figure 3.12: R code to wrap and call CUDA C function

### 3.4.2 Integration of Other languages with CUDA

The integration of Other languages with CUDA has similar principle. Some information can be found from the following website.

- Fortran:
- PGI Fortran to CUDA compiler [http://www.pgroup.com/resources/accel.htm](http://www.pgroup.com/resources/accel.htm)

- **Java:**

- **Python:**
  - PyCUDA Python wrapper [http://mathema.tician.de/software/pycuda](http://mathema.tician.de/software/pycuda)

- **.NET languages:**

### 3.4.3 Pros and Cons of integration

**Pros:**

- Essential wrappers of the CUDA C functions
- Friendly to non-C programmers
- Focus on algorithm instead of programming

**Cons:**

- Limited control to the GPU device
- Performance loss due to extra data transfer between CPU and GPU
Chapter 4

GPU Computing in Statistics

Many statistical analysis are involved in large number of data processing or simulating, hence are naturally parallelizable. An incomplete list of the parallelizable statistical algorithms include:

- Random numbers/processes simulation
- Markov Chain Monte Carlo (MCMC)
- Maximum likelihood optimisation
- Optimisation with dynamic programming
- Kernel density estimation
- Multivariate statistical analysis

4.1 Random Numbers Generation on GPU

Random number generation is usually the most time-consuming procedure in many statistical applications, however this are naturally parallel if the random numbers are independent. The speed-up will be expected high since the proportion of parallelizable part is relatively high.

4.2 Random Walk Generation on GPU

Another feasible applications of GPU computing is the random walk generations. Each chain is and must be computed by a single thread. However, if multiple chains are repeated, these are independent. So this is a parallelizable task. Since the random chains generated are independent, the generation can be easily parallelized. Figure 4.1 and 4.2 showed the output of random walk generation by GPU computing and the time consumption between CPU and GPU. The result shows that the speed-up was 19.5 for 1000 chains.
The code of the random walk generation is as following.
/*
* Simulate Random Walk on GPU computing by CUDA C
* Update: 05/12/2010
*/

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>
#include <R.h>

// Simple utility function to check for CUDA runtime errors
void checkCUDAError(const char* msg);

//Thread Per Block
#define THREADS_PER_BLOCK 256

//define kernel functions
//this kernel function simulate a random Processes in device
__global__ void kernel_simulation(unsigned long long seed,
      double* Data, int NumChain, int NumStep,int Thin)
{
  int id = threadIdx.x + blockIdx.x * THREADS_PER_BLOCK;
  /* Each thread gets same seed, a different sequence number, 
     no offset */
  curandState localState;
  curand_init(seed, id, 0,&localState);
  int i,j;
  double increment;
  double Stddev=1/sqrtf((NumStep-1)*Thin);
  Data[id*NumStep+0]=0;
  for(i=1;i<NumStep;i++)
  {
      increment=0;
      for(j=0;j<Thin;j++)
      {
          increment+=Stddev*curand_normal_double(&localState);
      }
      Data[id*NumStep+i]=Data[id*NumStep+i-1]+increment;
  }
}

extern "C"
{
  void RandWalkGPU(int *pNumChain, int *pNumStep, int *pThin, double *pDataMatrix)
  {
      int NumChain=*pNumChain;
      int NumStep=*pNumStep;
      int Thin=*pThin;
      double *h_Data=pDataMatrix;
      
      //allocate space memory for a random Processes on device
      double* d_Data;
      size_t size=NumChain*NumStep*sizeof(double);
      cudaMalloc((void **)&d_Data,size);
      cudaMemcpy(d_Data,h_Data,size,cudaMemcpyHostToDevice);
      
      //define the block dimension each chain simulated in 1 thread
      int blocksPerGrid=(NumChain+THREADS_PER_BLOCK-1)/THREADS_PER_BLOCK;
      
      //set seed
      unsigned long long seed=time(NULL);
      kernel_simulation<<<blocksPerGrid,THREADS_PER_BLOCK>>>(seed, d_Data,NumChain,NumStep,Thin);
      
      // check if kernel execution generated an error
      checkCUDAError("kernel invocation");
      cudaMemcpy(h_Data,d_Data,size,cudaMemcpyHostToDevice);
      
      // Check for any CUDA errors
      checkCUDAError("memcpy");
      
      // Free device memory
      cudaFree(d_Data);
      
      //return EXIT_SUCCESS;
  }
} //eDataern "C"

Figure 4.3: Random Walk Generation on GPU - CUDA C code

25
4.3 Metropolis-Hastings Markov Chain Monte Carlo

Markov Chain Monte Carlo with many independent chains is another viable example. The Metropolis-Hastings Markov Chain Monte Carlo algorithm is as follows:

1. Initialise $x^{(0)}$.

2. For $i = 0$ to $N - 1$
   - Sample $u \sim U[0, 1]$.
   - Sample $x^* \sim q(x^* | x^{(i)})$ for some (Markovian) proposal density $q(\cdot | x^{(i)})$.
   - If $u < A(x^{(i)}, x^*) = \min\left(1, \frac{p(x^*)q(x^{(i)} | x^*)}{p(x^*)q(x^{(i)} | x^{(i)})}\right)$, where $p(\cdot)$ is the target density,
     
     $x^{(i+1)} = x^*$
     
     otherwise
     
     $x^{(i+1)} = x^i$

The output and code are shown below.
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <cuda.h>
#include <curand_kernel.h>

#define THREADS_PER_BLOCK 256
#define NO_CHAIN 5
#define NO_STEP 1000
#define NO_SUBSTEP 1
#define NO_PARAMETER 2
#define NO_OBSERVATION 1

typedef struct {
  float data[NO_PARAMETER];
} MCMC_PAR_VEC;

void ini_chains(MCMC_PAR_VEC *theta, int No_chain, int No_step)
{
  theta[0*No_step].data[0]=0.0;
  theta[0*No_step].data[1]=0.0;
  theta[1*No_step].data[0]=3.0;
  theta[1*No_step].data[1]=3.0;
  theta[2*No_step].data[0]=-3.0;
  theta[3*No_step].data[0]=3.0;
  theta[3*No_step].data[1]=-3.0;
  theta[4*No_step].data[0]=-3.0;
  theta[4*No_step].data[1]=-3.0;
}

MCMC_PAR_VEC mcmc_jump(MCMC_PAR_VEC theta, curandState *r)
{
  MCMC_PAR_VEC theta_star;
  float sigma_0=1, sigma_1=1;
  theta_star.data[0]=theta.data[0]+0.2*curand_normal(r)*sigma_0;
  theta_star.data[1]=theta.data[1]+0.2*curand_normal(r)*sigma_1;
  return theta_star;
}

float mcmc_p(MCMC_PAR_VEC theta)
//standard bivariate normal
{
  float y=0;
  float mcmc_p;
  float sigma_sq_0=1, sigma_sq_1=1;
  mcmc_p=exp(-pow(theta.data[0]-y,2)/(2*sigma_sq_0)-pow(theta.data[1]-y,2)/(2*sigma_sq_1))/((2*M_PI)*sqrt(sigma_sq_0*sigma_sq_1));
  return mcmc_p;
}

float mcmc_q(MCMC_PAR_VEC theta_a, MCMC_PAR_VEC theta_b)
//standard bivariate normal
{
  float mcmc_q;
  float sigma_sq_0=1, sigma_sq_1=1;
  mcmc_q=exp(-pow(theta_a.data[0]-theta_b.data[0],2)/(2*sigma_sq_0)-pow(theta_a.data[1]-theta_b.data[1],2)/(2*sigma_sq_1))/((2*M_PI)*sqrt(sigma_sq_0*sigma_sq_1));
  return mcmc_q;
}

Figure 4.6: MCMC on GPU - CUDA C code
Bibliography


