**Acceleration on Naïve Bayes with GPUs**

Yi Zhou

Abstract

Data Mining is an important technology in today’s world. Some researches focus on the algorithm design in this field. Some focus on the ability for the algorithm to solve problems effectively when facing a huge dataset. This paper is going to follow the idea that using a parallelism way could improve the ability. And it will implement acceleration on Naïve Bayes with Commodity GPUs and CUDA.

The experiment will be done through a small test on a real-life problem. And it will compare the time cost between the original code and modified parallelism running code.

**contents**

[**1.** **Introduction** 4](#_Toc414471371)

[**2.** **Background** 5](#_Toc414471372)

[**2.1.** **CPU and GPU** 5](#_Toc414471373)

[**2.2.** **CUDA** 7](#_Toc414471374)

[**2.3.** **NaiveBayes** 10](#_Toc414471375)

[**3.** **The approach** 15](#_Toc414471376)

[**3.1.** **Overall approach** 15](#_Toc414471377)

[**3.2.** **Real-life problem** 15](#_Toc414471378)

[**3.3.** **Detailed design** 18](#_Toc414471379)

[**4.** **Experiments** 20](#_Toc414471380)

[**4.1.** **Environment** 20](#_Toc414471381)

[**4.2.** **Result** 20](#_Toc414471382)

[**5.** **Conclusions** 22](#_Toc414471383)

[**Reference** 23](#_Toc414471384)

[**Source code** 24](#_Toc414471385)

1. **Introduction**

Data Mining had a wildly use in many fields in today’s world. Some researches had been done to improve Data Mining algorithm’s performance. They might use different ways to implement the Data Mining procedures in order to acquire a higher correctness. However, when facing a huge amount of data, the speed of the procedures cost should also be a considerable issue. There are several ways to improve this situation. One of them is using GPU. GPU has a quite different architecture compared to CPU, which we will see in 2.1.. It is able to complete faster computations through this designing.

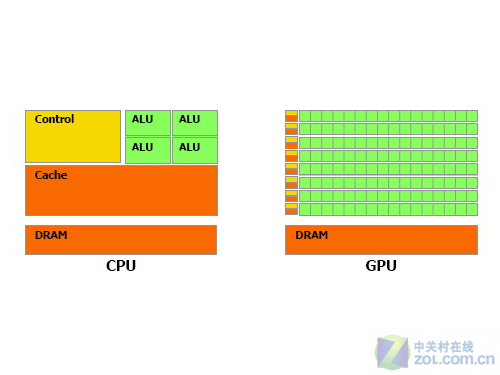
One of famous Data Mining algorithm is NaiveBayes. It could use arithmetic ways to get a classifier from massive data and then through the classifier we could get an approximate result about those unclassified data.

To achieve a complete version of GPU paralleled NaiveBayes algorithm needs more time and work. So, this paper is going to do an real life experiment about how to improve the performance on NaiveBayes.

1. **Background**
2. **CPU and GPU**

CPU (Central Processing Unit） is the compute center of a computer. It contains the ALU (Arithmetic Logic Unit) which could execute fixed-point or floating-point arithmetic operations. CPU’s Architecture will benefit the serial architecture of x86 instruction set. This allowed CPU to finish one work efficiently.

The term GPU (Graphic Processing Unit) was first proposed by NVIDIA in 1999. It reduced the graphic card’s dependence on the central processor, and share the work which was originally done by CPU, especially in 3D graphics processing.



G2.1.1 Architecture of CPU and GPU

As G2.1.1 shows, Architecture of CPU and GPU differed greatly. CPU has a lot of function modules and it can adapt to the complex operation environment. GPU instead has a relatively simple composition. Stream processor and memory controller occupied the vast majority of transistor. Most of transistors in CPU are used to construct control circuit (such as branch prediction) and Cache, only a small part of transistor will to the actual operation work.

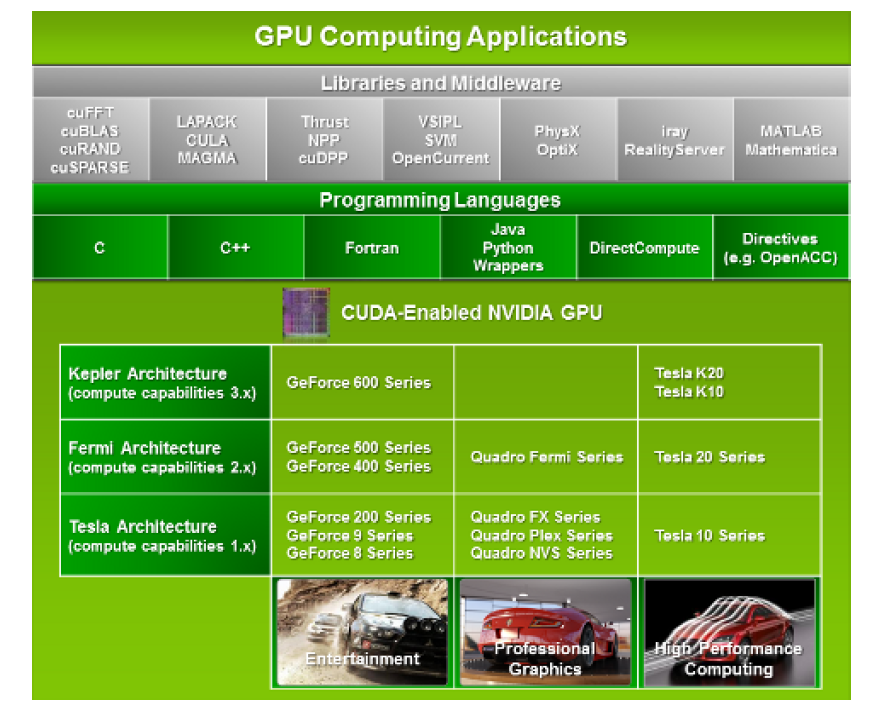
The GPU control is relatively simple, it has a small demand for Cache. So most of the transistors can be composed of various types of special circuit, multi line, which makes the calculation speed of GPU faster. Now CPU technology progress is slower than Moore's law, while the GPU speed has more than Moore's law, and its performance was doubled every 6 months. Owens etc. has a comprehensive introduction of GPU’s history and architecture in their paper.

Today’s CPU has a complex designing in the front part, the instruction decoding, branch prediction and etc. consumed a huge number of transistors. The designing goal of CPU is not only to have a high throughput, but also have application environment compatibility. CPU applications have to face far more than GPU. CPU is designed to handle general task processing, processing, operation and system core control etc.. The most basic components of an CPU is the arithmetic logic unit and a control unit, CPU micro architecture is optimized for efficient computing data correlation which is not too complex and other non-computing tedious works, the purpose is to do well in the treatment of complicated task in daily.

GPU design's purpose is to achieve accelerated graphics. For now, it focus on the implementation of 3D graphics acceleration. The reason is that its designing is basically used for optimizing 3D graphics acceleration, such as the z-buffering blanking, texture mapping (texture mapping), graphics coordinates transform and lighting calculations (Transforming & Lighting) and etc.. The object of this calculation is for a large number of parallel data, the amount of data that calculated is huge. But the data type GPU faced is single, the vast majority of data it processed is single-precision floating-point number. When it came to GTX200 and HD 4800 series cards, GPU could support the calculation of double- precision floating-point number.

1. **CUDA**

CUDA (Compute Unified Device Architecture) is a general computational framework based on a new parallel programming model and instruction set architecture. It is an integrated technology developed by NVIDIA, an official name of the company for GPGPU. Through this technology, users can be calculated by using the NVIDIA GeForce 8 after GPU and the newer Quadra GPU. It is also the first time when GPU can be used as the C- compiler development environment. When NVIDIA marketing, they often mixing the promotion of compiler and architecture, which caused confusion. In fact, CUDA can be compatible with OpenCL or their C- compiler. Instruction will eventually be drivers into PTX code on Both the CUDA C- language and OpenCL compiler, and be computed by the core.



G2.2.1 interface designed in CUDA

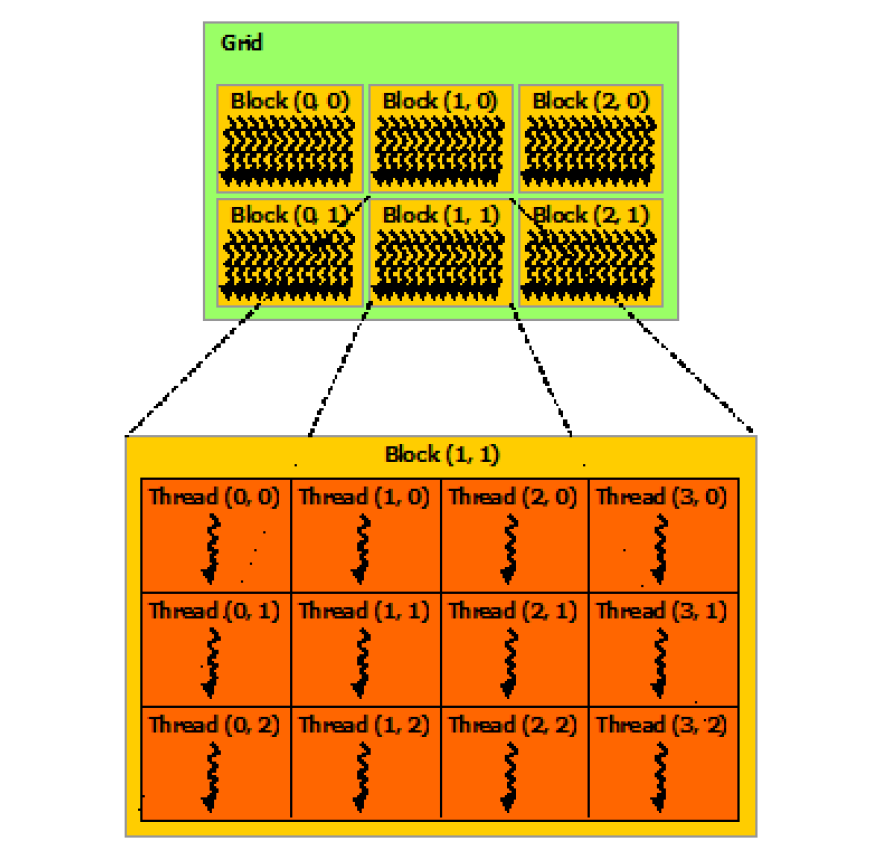
In G2.2.1 we could see that CUDA contains a software environment which allows developers to use C or other programming languages.

By the emergence of multicore CPU and GPU, building parallel system on processor cores has become the mainstream designing. Furthermore, degree of parallelism of CPU and GPU will continue to expand by Moore’s law. The challenge is how to develop transparent extended degree of parallelism in order to use the increasing number of processor cores. CUDA parallelism programming was designed to overcome this challenge and at a same time maintain a relatively low learning curve for those programmer who was familiar to standard programming language like C.

CUDA is now used widely in Bioinformatics, Computational Chemistry, Machine Learning and etc.

G2.2.2 shows the Grid and Block architecture in CUDA programming. This paper is going to use CUDA C language to do the experiment. CUDA C is an extension to C language. There are five inner variables that user could use in CUDA C:

* gridDim, gridDim is a structure contains three elements x, y, z, which indicates the dimension of grid on directions x, y, z. now, we could only use 2D coordinates.
* blockDim, blockDim is a structure contains three elements x, y, z, which indicates the dimension of block on directions x, y, z. it matches the first and second parameters of kernel function’s running configuration.
* blockIdx, blockIdx is a structure contains three elements x, y, z, which indicates the index on dimension x, y, z of grid where the current thread running.
* threadIdx, threadIdx is a structure contains three elements x, y, z, which indicates the index on dimension x, y, z of block where the current thread running.
* warpSize. warpSize indicates the dimension of warp.



G2.2.2 Grid of Threads Block

1. **NaiveBayes**

Machine Learning (ML) is one of the important branch in computer science. The main research issues for ML is how to teach machine “study”.

NaïveBayes is one of the popular ML algorithm. This classifier’s principle is to calculate the object’s posterior probability using its prior probability and Bayes formula. The posterior probability is used to decide which class this object belongs to.

NaiveBayes is based on a simple assumption: attributes are relatively independent between each other given the object’s value.

Theoretically, probability model classifier is a conditional probability model

Independent category variable C has a number of categories，they are conditionally dependent on a number of featured variables F1, F2, …, Fn. The question is that if the number of features n is relatively big or every feature could have a wide range of value, it is impossible to list all the probability based on the probability model. So we change this model to make it workable. Bayes theorem has these formula:

It could also be writen as:

Actuallly, we only care about the numerator. Because the denominator is independent on C and value of feature Fi is fixed. The denominator could be considered as a constant. Then the numerator is equal to Joint distribution model.

Repetitively using the chain rule, the formula could be:

Now the assumption about conditional independent begin to work: assuming that every feature Fi is independent on the others Fj (j ≠ i). This means:

For those i ≠ j, joint distribution model could be written as:

This means that distribution of category variable C under the assumption could be:

Where evidence Z is a scaling factor which only dependent on F1,…, Fn. When the value of feature variables is known, Z is a constant. Because the formula is separated to [prior probability](http://www.iciba.com/prior_probability) of category p(C) and independent probability distribution p(Fi­|C), the controllability of this probability model has been improved. If this is a classification problem of k and every p(Fi|C=c) could be r parameters, the corresponding NaiveBayes model has (k-1)+nrk parameters. In the practical applications, people usually take k as 2, r as 1 (Using Bernoulli distribution as attributes), so the number of parameters in this model is 2n+1, where n is the number of attributes in bi - level classification.

Simply saying, the steps of NaiveBayes algorithm could be:

1. Assume x = {a1, a2, …, am}, where x is a instance waiting to be classified. Each a is one of x’s attributes.
2. There is the set of class C = {y1, y2, …, yn}.
3. Computing p(y1|x), p(y2|x), …, p(yn|x)
4. Finding a set of instances of which class is given. Such set is called training set.
5. Counting the conditional probability for every characteristic attributes under every class. That is:

p(a1|y1), p(a2|y1), …, p(am|y1); p(a1|y2), p(a2|y2), …, p(am|y2);…;p(a1|yn), p(a2|yn), …, p(am|yn);

1. If every characteristic attributes are conditional independent, according to Bayes Theorem,
2. Because denominator is constant for every class, the only thing needed to do is Maximized the nominator. Also the characteristic attributes are independent, so

1. If p(yk|x) = max{ p(y1|x), p(y2|x), …, p(yn|x)}, then x ∈ yk

G2.3.1 shows the procedure of NaiveBayes Classifier, it is separated into three important stages: preparation stage, classifier training stage, applying stage.

Preparation

Determine the characteristic attributes

Acquire training instances

Compute conditional probability for every characteristic attribute

Compute P(yi) for every class

Compute P(x|yi)p(yi) for every class

Consider the max P(x|yi)p(yi) to be the class of x

Training

Applying

G2.3.1 Procedure of NaiveBayes classifier

1. **The approach**
2. **Overall approach**

Having Known the NayeBayes classifier, this paper is going to find out an approach to optimize its running time. A Bayes network S=<G, Θ> consist of the network topology and the local probability distribution. G stands for a conditional independent network of X, where X = {X1, X2, …, Xn}. Θ stands for distribution of local conditional probability which related to every variables.

When classing the instance, the costly steps is updating weight and updating parameters. Updating weight is to compute weight on every class, which is the sum of probabilities on every training instance. Updating parameters is to use the new weight to update the parameters used to do classification in next stage.

Both of these two updating could be optimized utilizing the parallel technology. Through 2.2, it is a common view that CUDA could be used to do parallel development. This paper is going to do the optimization in CUDA C language.

1. **Real-life problem**

Because the limited time and resources, this paper was not going to completely rewrite NaiveBayes Algorithm in parallel using CUDA C. Instead, a real-life problem will be presented and solved by both general C and CUDA C.

For some BBS website, using fake accounts is a common problem. As the administrator of BBs website, it will be really nice to find out these accounts and delete them. This will avoid the intervention for operation analysis reports. And it will also enhance the understanding and monitoring of BBS websites.

If all the checking work were done by people, it will be costly and low efficient. An automatic detection mechanism will improve this situation. Simply saying, we need to find a way to classifier the real accounts and fake accounts.

First assuming C = 0 stands for real accounts and C = 1 stands for fake accounts.

1. Defining characteristic attributes and partition

This step is going to find the attributes that we could use to distinguish real and fake accounts.

Here are three attributes we choose: a1: number of bogs/registration days, a2: number of friends/registration days, a3: whether using the real head portrait. In a BBS website, this three attributes could be calculated or transferred from database. Now the partition is given:  
a1: {a<=0.05, 0.05<a<0.2, a>0.2}, a2: { a<=0.1, 0.1<a<0.8, a>=0.8} a3: {a=0 (head portrait is fake), a=1}

1. Acquiring training set

Here we use 10 thousand accounts checked by administrator as training set.

1. Computing probabilities for every class on all instances

Dividing real and fake accounts by 10 thousand we get:

P(C = 0) = 8900/100000 = 0.89

P(C = 1) = 110/100000 = 0.11

1. Computing probabilities for every attribute on all classes

P(a1 <= 0.05|C = 0) = 0.3

P(0.05<a1 <=0.2|C = 0) = 0.5

P(a1>0.2|C=0) = 0.2

P(a1 <= 0.05|C = 1) = 0.8

P(0.05<a1 <=0.2|C =1) = 0.1

P(a1>0.2|C=1) = 0.1

P(a1 <= 0.1|C = 0) = 0.1

P(0.1<a1 <=0.8|C = 0) = 0.7

P(a1>0.8|C=0) = 0.2

P(a1 <= 0.1|C = 1) = 0.7

P(0.1<a1 <=0.8|C = 1) = 0.2

P(a1>0.8|C=1) = 0.1

P(a3 = 0|C = 0) = 0.2

P(a3 = 1|C = 0) = 0.8

P(a3 = 0|C = 1) = 0.9

P(a3 = 1|C = 1) = 0.1

1. Applying classifier

Next we are going to use the classifier we made to evaluate an account. This account has a fake head portrait. Its number of bogs/registration days is 0.1. Its number of friends/registration days is 0.2.

P(C = 0)P(x|C = 0) = P(C = 0)P(0.05<a1<0.2|C = 0)P(0.1<a2<0.8|C=0)P(a3 = 0|C = 0) = 0.89\*0.5\*0.7\*0.2 = 0.0623

P(C = 0)P(x|C = 0) = P(C = 0)P(0.05<a1<0.2|C = 0)P(0.1<a2<0.8|C=0)P(a3 = 0|C = 0) = 0.89\*0.5\*0.7\*0.2 = 0.0623

P(C = 1)P(x|C = 1) = P(C = 1)P(0.05<a1<0.2|C = 1)P(0.1<a2<0.8|C=1)P(a3 = 0|C = 1) = 0.11\*0.1\*0.2\*0.9 = 0.0623

So, according to this classifier, this account is likely fake.

1. **Detailed design**

3.2 shows an example of the application on one instance. However, there should not be only one case to be classified. Usually there are millions of instances waiting to be handled. So we need to make use of programing to do this massive work.

// CUDA kernel. Each thread takes care of one element of c

\_\_global\_\_ void classify(double \*a, int \*c, int n)

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

…

}

//main function, the entry of the program

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

{

…

int n = INSTANCES\_NUM;

int blockSize, gridSize;

blockSize = 1024;

gridSize = (int)ceil((float)n/blockSize);

// Execute the kernel

classify<<<gridSize, blockSize>>>(d\_a, d\_c, n);

…

}

* n: number of instances need to be classified
* blockSize: number of threads in each thread block
* gridSize: number of thread blocks in grid
* i: id of each thread

This is a simple structure of CUDA C program. The kernel function can be used to run the code in GPU. And GPU will create considerable threads to do the work in parallel, which makes the optimization of NaiveBayes classifier possible.

For every instance, it will do these work in the kernel function:

1. Do partition on instance;
2. Compute probabilities for each class of this instance;
3. Choose the max probability as the right class and return.
4. **Experiments**
5. **Environment**

Machine information:

1. Commodity laptop
2. Processor: Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHZ
3. RAM: 8G
4. 64bit
5. GPU: NVIDIA GeForce GTX 860M

Library dependency:

1. Could work on both linux and windows system.
2. CUDA version: v6.5
3. C compiler: gcc 4.8.1(Find more information about windows gcc on <http://www.mingw.org/>)

Dataset:

We use different numbers of dataset randomly generated from program to form the test set.

1. **Result**

NBC: NaiveBayes classifier

T4.2.1 Compare of running time around Normal NBC and optimized NBC

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number of instances | 10,000 | 100,000 | 1,000,000 | 10,000,000 |
| Normal | **0.4014** | **4.1028** | **40.5300** | **404.3862** |
| optimized | **0.0736** | **0.6173** | **6.0161** | **57.5090** |

Unit:ms. Each result is an average on 10 same experiments.

G4.2.1 Compare of running time around Normal NBC and optimized NBC

1. **Conclusions**

From T4.2.1 we could see that time cost by normal NBC and parallel NBC are both increasing when numbers of instances become larger. And for each specific number of instances, parallel NBC is faster than normal NBC. In G4.2.1, we could find that the running time of two programs are all relatively short. However, the parallel NB program is increasing slowly while normal NBC has a sharp increase after the number of instances become over 1,000,000. So, we could believe that it is possible to make NaiveBayes classifier running in a parallel way and the paralleled NBC program is more efficient than previous one.

Though, the unit of all the time we calculated is microseconds, which means they are all relatively small for today’s high speed computer. This happens because we are just programming with a real-life problem, which is simple and don’t need too much fussy operations. However, there are more complicated problems and the time may be far more costly than this one. At that time, a more efficient implementation of a classifier will be more worthy for people to use.

**Reference**

* Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, *96*(5), 879-899.
* Nickolls, J., & Dally, W. J. (2010). The GPU computing era. *IEEE micro*,*30*(2), 56-69.
* Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *Ieee Micro*, *28*(2), 39-55.
* Fang, J., Varbanescu, A. L., & Sips, H. (2011, September). A comprehensive performance comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on* (pp. 216-225). IEEE.
* Garland, M. (2010, April). Parallel computing with CUDA. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (pp. 1-1). IEEE.
* Kim, S. B., Han, K. S., Rim, H. C., & Myaeng, S. H. (2006). Some effective techniques for naive bayes text classification. *Knowledge and Data Engineering, IE*
* Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., ... & Volkov, V. (2008). Parallel computing experiences with CUDA. *IEEE micro*, (4), 13-27.

**Source code**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

// CUDA kernel. Each thread takes care of one element of c

\_\_global\_\_ void vecAdd(double \*a, int \*c, int n)

{

const double classValue[2] = {0.89,0.11};

const double attribute\_1[6] = {0.3,0.8,0.5,0.1,0.2,0.1};

const double attribute\_2[6] = {0.1,0.7,0.7,0.2,0.2,0.1};

const double attribute\_3[4] = {0.2,0.9,0.8,0.1};

int i = blockIdx.x \* blockDim.x + threadIdx.x;

if (i<n&&i%3==0){

if(a[i]<=0.05){

a[i]=0;

}else if(a[i]>0.05&&a[i]<0.2){

a[i]=2;

}else{

a[i]=4;

}

if(a[i+1]<=0.1){

a[i+1]=0;

}else if(a[i+1]>0.1&&a[i+1]<0.8){

a[i+1]=2;

}else{

a[i+1]=4;

}

if(a[i+2]==0){

a[i+2]=0;

}else{

a[i+2]=2;

}

double temp[2];

for(int j = 0;j < 2; j++){

temp[j] = classValue[j] \* attribute\_1[(int) (a[i]+j)] \* attribute\_2[(int) (a[i+1]+j)] \* attribute\_3[(int) (a[i+2]+j)];

}

if(temp[0]>temp[1]){

c[i/3] = 0;

}else{

c[i/3] = 1;

}

}

\_\_syncthreads();

}

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

{

// Size of vectors

int numInstance = 100000000;

int n = numInstance\*3;

float time\_elapsed[10];

cudaEvent\_t start,stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

// Host input vectors

double \*h\_a;

//Host output vector

int \*h\_c;

// Device input vectors

double \*d\_a;

//Device output vector

int \*d\_c;

// Size, in bytes, of each vector

size\_t bytes = n\*sizeof(double);

size\_t bytes\_ = numInstance\*sizeof(int);

// Allocate memory for each vector on host

h\_a = (double\*)malloc(bytes);

h\_c = (int\*)malloc(bytes\_);

// Allocate memory for each vector on GPU

cudaMalloc(&d\_a, bytes);

cudaMalloc(&d\_c, bytes\_);

int i;

// Initialize vectors on host

for( i = 0; i < n; i = i+3 ) {

h\_a[i] = rand()%50/100.0;

h\_a[i+1] = rand()%10/10.0;

h\_a[i+2] = rand()%2/10.0;;

}

// Copy host vectors to device

cudaMemcpy( d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

int blockSize, gridSize;

// Number of threads in each thread block

blockSize = 1024;

// Number of thread blocks in grid

gridSize = (int)ceil((float)n/blockSize);

int j = 0;

for(j = 0;j < 10;j++){

cudaEventRecord( start,0);

// Execute the kernel

vecAdd<<<gridSize, blockSize>>>(d\_a, d\_c, n);

cudaEventRecord( stop,0);

cudaEventSynchronize(start);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time\_elapsed[j],start,stop);

}

cudaEventDestroy(start);

cudaEventDestroy(stop);

float sum;

for(j = 0;j < 10;j++){

sum+=time\_elapsed[j];

}

printf("time：%f(ms)\n",sum/10.0);

// Copy array back to host

cudaMemcpy( h\_c, d\_c, bytes\_, cudaMemcpyDeviceToHost );

// Release device memory

cudaFree(d\_a);

cudaFree(d\_c);

// Release host memory

free(h\_a);

free(h\_c);

return 0;

}