Formal Description and Optimization Based High –Performance Computing on CUDA

Huacheng Zhao
College Of Electronic and Information Engineering, xi'an JiaoTong University Shaanxi,China
Z151614650@126.com

Bo Li
College Of Electronic and Information Engineering, xi'an JiaoTong University Shaanxi,China
boblee@xjtu.edu.cn

Jingjing Liang
College Of Electronic and Information Engineering, xi'an JiaoTong University Shaanxi,China
liangjingj4@sina.com

***Abstract*-In recent years, with the development of GPU, based on the general purpose computation on graphics processors has become a new field. NVIDIA's CUDA is a hardware-software architecture that enables high-performance computing developers to use the computational power and memory bandwidth of the GPU in a familiar programming environment –the C programming language. Aiming at the processing of GPU, this paper provides the formal description for data parallel mode, a detailed description of the CUDA programming mode and the principle of optimization. It shows by the comparative experiment that CUDA owns strongly of the ability to the parallel processing and provides new methods and ideas to GPGPU.**

Keywords- formal description ; CUDA; Optimization design

I. INTRODUCTION

Recently,graphic processing units(GPUs) originally purposed for graphic application have emerged as the most powerful computing .NVIDIA's CUDA is released to enable high-performance computing becoming easier[1][2][3]. CUDA treats the GPU as a dedicated coprocessor of CPU, and allows the same codes to run on the different GPU cores. CPUs own multi-cores ,but GPUs own many cores. CUDA is a extension of the C programming language, and completing the program for the implementation of the chip don't need to learn the specific commands of displaying chips. The most advantage of CUDA is that it greatly reduces the use of GPU programming the threshold of entry. As long as programmers master C language, people are able to write a CUDA program.

CUDA includes the main functions[4][5] :

- providing standard C language in the GPU;
- supporting the CUDA parallel computing to providing a unified hardware and software of the solution;
- owning the library of FFT and BLAS;
- supporting the direct access to the driver procedures;
- standard numerical FFT and BLAS library;
- for calculating the special CUDA driver.

The features of CUDA computing combine the characteristics of GPU through a standard C language to create the thread applications. How much can GPU computing speed up a real-world science? Researchers and companies finished speedups ranging from 10 to 100 by using CUDA , across domains from computational chemistry to physics model, to CT and MRI, to sorting and searching .

The rest of the paper is organized as follow. Sec 2 presents the data parallel model of the formal description.Sec.3 provides the CUDA programming model.Sec4 proposes several methods to optimize performance. Experimental results are presented in Sec5, and we conclude the paper in Sec6.

II. DATA PARALLEL MODEL OF THE FORMAL DESCRI-PTION

CUDA is a model based on data parallel. It is particularly suitable for parallel computing , which structure can be used to support a very high processing speed of this calculation. For practical applications, this article uses the formal description for the data parallel model.

A. Formal definition of data parallel

Assuming the size of data set size is l .The data set is:

$$F = \{ f_i \mid i = 0,1,2,3....l-1 \} \quad (1)$$

(This project is supported by the Electronic Information Industry Foundation of China under the Government)

where f_i is a single data. There are some types of operations:

$$P = \{ p_j \mid j = 0,1...m-1 \} \quad (2)$$

$N_k(f_k, T_k)(k = 0,1,2..m-1)$ is that the f_k data under the T_k operation . PV indicates the data parallel processing in n tuple.

Definition: Data set is divided into l / n parts according to certain rules, and each part is composed by the same operations, then:

$$PV = (N_0(f_{i_0}, T), N_1(f_{i_1}, T), ..., N_{n-1}(f_{i_{n-1}}, T)) \quad (3)$$

where $f_{i_0} \bigcup f_{i_1} \bigcup ..... \bigcup f_{i_{n-1}} = f_i, T \subseteq P$.

Figure 1 can be used to describe:

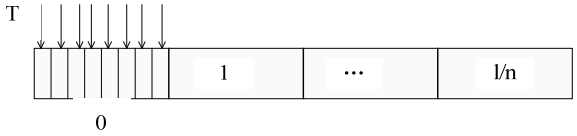

Figure 1 Data parallel model

B. Time Function of Data parallel

$time(N_k(f_k, T_k))$ that is a function of time in dealing with the implementation of the data f_k . In order to the convenience of computing , assuming the same size of the data spends the same time to process, then

$$time(N_i(f_0, T_j)) = time(N_i(f_1, T_j)) = ... = time(N_i(f_{l-1}, T_j))$$

Parallel processing in the data model is defined, we can see that every data is processing by the m types of operations. Under ideal circumstances, data-parallel processing time is

$$t = l / n * \sum_{j=0}^{j=m-1} time(N_j(f_{i_0}, T_j)) \quad (4)$$

Data processing time is proportional to the data set size for the smallest unit of data. Then

$$time(N_j(fi_0, T_j)) = 1/n * time(N_j(f_i, T_j)) \quad (5)$$

Therefore, the time function t can be transformed into:

$$t = l / n * (1/n) * \sum_{j=0}^{j=m-1} time(N_j(f_i, T_j)). \quad (6)$$

C. Formal definition of task parallel

Definition: The same or different data on different nodes are imposed by different operations. These operations are independent on each other. We express the task parallel with PV.

$$PV = (N_0(d_0, T_0), N_1(d_1, T_1), ..., N_{m-1}(d_{m-1}, T_{m-1}))$$

, $d_i \subseteq F, T_i \subseteq P.$ (7)

D. Time Function of task parallel

In the task parallel processing mode, the data size is l .We allocate the operations to the nodes. And the number of operations and nodes are m. Processing the j node on the execution time is $time(N_j(F, T_j)$. In an ideal case, then :

$$time(N_j(F, T_j) = l * time(N_j(f_i, T_j), (j = 0,1,2..m-1) \quad (8)$$

The task parallel processing time is

$$t2 = \max_{j=0}^{j=m-1} time(N_j(F, T_j)) = \max_{j=0}^{j=m-1} l * time(N_j(f_i, T_j)) = l * \max_{j=0}^{j=m-1} time(N_j(f_i, T_j)). \quad (9)$$

But, $t = l / n * (1/n) \sum_{j=0}^{j=m-1} time(N_j(f_i, T_j))$.

t< l / n^2 *m* $\max_{j=0}^{j=m-1} time(N_j(f_i, T_j)$ <m/ n^2 *t2,

n>>m , so t<t2.

Through this analysis, the following conclusions can be: data parallel is more suitable for large-scale data processing than task parallel.

III BACKGROUND AND RELATION WORK

This section presents the basic concept of the CUDA of programming model.

A. The CUDA of programming model

The first generation of GPGPU requires that non-graphics application is mapped through the graphics application. Recently, one of the major GPU vendors—NVIDIA released their new parallel programming model, named Compute Unified Device Architecture (CUDA) that extends the C programming language. Another GPU vendor AMD announced Close To Metal(CTM) programming model that uses an assembly language for application development. Intel provides

Larrabee, a new mutli-core GPU architecture for GPU computing.

Currently, CUDA is the best available programming model, and is the most widely accepted model. Since the release of CUDA, it has been used for a large number of applications[6][7][8][9]. For these reasons, we choose to use CUDA in our research.

CUDA programming model of the GPU itself has great ability to parallel processing. The CUDA programming model allows developers to exploit parallelism by writing natural. In the CUDA model ,GPU is viewed as a co-processor, to perform a large number of parallel threads. A source program consists of the host code and the kernel code. The host code is run on the CPU, and the kernel code is executed on the GPU. Matrix addition will serve as a simple example. In order to implement two N*N matrices on the CPU in C programming language, we should write a two nested loop:

```
        void addMatrix (int *a , int b, int *c,int N)
{
            int index;
            for(int i=0;i<N; i++){
            for(int j=0;j<N; j++)
            {
                        index=i*N+j;
                        c[index]=a[index]+b[index];
                        }
            }
}
 int main()
{
  ……..
  addMatrix(a,b,c,N);
   return 0;
}
```

In the CUDA, we write a C function, called a kernel that computes one element in the Matrix. We use CUDA to rewrite ,only to modify the computing on GPU.

```
__global__ void addMatrix(int *a,int *b,int *c,int N)
{
            int i=threadIdx.x;
       int j=threadIdx.y;
       int index=i+j*N;
       c[index]=a[index]+b[index];
}
Void main()
{
   …….
            dim3 blocksize(N,N);
  addMatrix<<<1,blocksize>>>(a,b,c,N);
}
```

Here the _global_ declaration indicates a kernel function that run on the GPU. The <<< N,N>>invokes a group of threads run in parallel.

CUDA provides three key abstractions—hierarchical thread blocks, shared memory and barrier synchroniza-tion-that provide a clear parallel structure. Many levels of threads provide fine-grained data parallelism and nested within coarse-grained data parallelism .

B. Thread blocks

Thread blocks are a group of co-ordination of threads through high-speed shared memory to share effectively data in parallel. More specifically, the users can specify in the kernel synchronization point, and all of thread blocks achieve synchronization point to hang . Thread blocks contain up to 512 threads on an NVIDIA ‘s architecture GPU[11][12]. Performing the same cores have the same dimensions and size of the thread blocks which can be combined into a grid. Each thread by the thread ID to identify. To help address based on its thread ID, the application can be threaded into a block of any size specified one-dimensional, two-dimensional or three-dimensional array of threads and the use of one, two or three index components to identify each thread. For example, the size of the thread for the two-dimensional block (Bx, By), the index for the (x, y) of the thread ID for the thread (x + yBx). Therefore, the GPU on the above-mentioned operative require only a slight modification can be achieved.

```
__global__ void addMatrix(int *a,int *b,int *c,int N)
{
     int i=blockIdx.x*blockDim.x+threadIdx,x;
    int j=blockIdx.y*blockDim.y+threadIdx.y;
    int index=i+j*N;
     if(i<N&&j<N)
          {
                    c[index]=a[index]+b[index];
          }
}
Void main()
{
   …….
          dim3 dimBlock(16,16);
          dim3 dimGrid(N/dimBlk.x,N/dimBlk.y);
  addMatrix<<<dimGrid,dimBlk>>>(a,b,c,N);
}
```

Here a thread block size of 16*16=256 threads. A grid is created with blocks to have one thread per matrix element as before. The threads in each thread block are run in parallel.

C. Shared memory

In the CUDA architecture, the memory is divided into two categories: system memory and device memory. GPUs as a result of the computing of the equipment read and write memory faster than system memory. So it is responsible for the computing of the kernel function call before the division is not only the needs of system memory for data, but also the device memory to the memory for the thread to read and write operation.

CUDA threads in the implementation process can access multiple data memory space, as shown in Figure 2. Each thread has a private local memory. Each thread block has a shared memory, the memory block for all threads are visible, and the block has the same life cycle. Ultimately, all threads can access the same global memory.

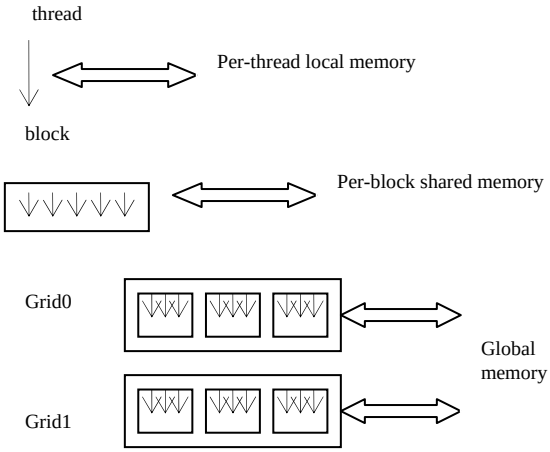

Figure 2 hierarchical memory

Implementing on the device only through the following several threads memory: registers, local memory, shared memory, global memory, constant memory and texture memory. Combined in the matrix example, an independent thread to run effectively, but there is no thread to know which elements can be access by other threads. CUDA solve to this problem through the shared memory to be completed. Kernel data store in shared memory. Threads in the same thread block are carried out through the shared memory to communicate only by a thread block to deal with more than one processor, so the presence of shared memory on the chip, resulting in a very express memory access. Because of multi-processor registers and shared memory block allocated to the thread of all threads, so each group of multi-processor can handle the number of kernel threads in each block depends on the number of registers and the number of shared memory. If every multi-processor does not have enough available memory to register or to share at least one thread processing block, the implementation of the kernel will not start. At present, the shared memory size of 16K bytes.

D. Barrier synchronization

Threads within a block can collaborate with each other through a number of shared memory to share data and to coordinate the implementation of synchronous memory access. More specifically, you can call __syncthreads()inner core function which is specified in the synchronization point. __syncthreads () play the role of the fence, and block all the threads here have to wait for further processing. To achieve effective collaboration, shared memory should be close to low-latency memory of the processor core.__syncthreads () should be lightweight, and a block all threads are must reside in the same processor core.

Once threads in the same memory space to operate in parallel, we must provide a mechanism to ensure that a thread do not read a result before another thread has finished writing it. CUDA provided in __syncthreads () built-in function to guarantees such a purpose. __syncthreads () function is an atomic operation that block all of the thread must be synchronized in order to continue.

IV PERFORMANCE OPTIMIZATION

GPU performance is very good, but to a large extent the performance should be restricted by the structure of the algorithm. In the use of CUDA, the data structure and memory access performance of the GPU have a great impact on, and sometimes play a decisive factor.

Performance is optimized from three main aspects:

- optimizing the largest parallel implementation;
- optimizing memory in order to make the most use of memory band-width;
- to optimize the use of instructions to get maximum throughput of instructions.

Follow to sum of squares as an example of performance optimization.

```
_global_ static void SquareSum1(int*num,int *result)
{
```

```
        int sum=0;
      clock_t start=clock();
for(int i=0;i<DATA_SIZE;i++)
{
        sum+=num[i]*num[i];
}
clock time=clock()-start;
*result=sum;
}
```

In the program, don't use in parallel processing. we remove computing on the CPU to implement.

Comparison of the experimental test results, as shown in table I:

TABLE I. THE PERFORMANCE TEST RESULTS

	CPU	GPU
time/s	0.15	0.27
throughput(GB/s)	0.0267	0.0148

We can see clearly that performance is bad on GPU. The main reason is that the architecture characteristics of GPU caused. In CUDA, copying data from the CPU to the GPU memory is called global memory. There is no cache in GPU. And the global memory access time latency is very long. It spends several hundred cycles . In the program, we use only a thread, so every time it read the data in global memory and the data must be read and compute before the next operation. This version of the program spend about 407Mcycles.

GeForce 8600GT and the frequency of the implementation unit is 1.5GHZ. This process takes about 0.27 seconds or so. From the table, we can see that in the calculation of square and, using CPU would be faster than the GPU, which is calculated because the square do not need much computing power. It is almost limited by memory bandwidth.

A. Process in parallel

There is only one thread to use and there is no cache in the global memory. Following the use of multi-threads to avoid the enormous costs of latency. When a large number of threads start, then when a thread to read memory, the beginning of time to wait for the results, CPU immediately switches to the next thread and read the next memory location. In theory, with enough threads can hide the latency. In order to parallel, the array can be divided into blocks, and each block calculates values. At the end , the result is by adding all of blocks .

```
_global_ static void SquareSum1(int *num,int *result)
{
        int sum=0;
   int tid=threadIdx.x;
   int size=DATA_SIZE/THREAD_NUM;
   clock_t start=clock();
     for(int i=tid*size;i<(tid+1)*size;i++)
     {
        sum+=num[i]*num[i];
     }
     *result=sum;
     clock time=clock()-start;
}
```

DATA_SIZE is the size of the array. THREAD_NUM is the number of threads. The number of threads is 256. After compiling the implementation, using the processing in parallel need 8.2Mcycles.The bandwidth is about 536MB/s with the GeForce 8600GT .

B. Optimizing memory in order to make the most use of memory band-width.

Optimizing memory in order to get the most use of memory bandwidth to transmit high-bandwidth data. Between the CPU and device the transmission of data is much lower than between global memory and the device. Therefore we minimize to transfer data between CPU and device. Different types of memory have different access patterns. the effective bandwidth may also be an order of magnitude worse, and optimize the memory of the next visit is to take advantage of the best model. GPU is a DRAM .The best way is continuous access mode. The model make thread0to read the first figure and thread1 to read the second figure and so on.

```
  _global__ static void SquareSum1(int *num,int
*result)
  {
        int sum=0;
     int tid=threadIdx.x;
     int size=DATA_SIZE/THREAD_NUM;
     clock_t start=clock();
for(int i=tid*size;i<DATA_SIZE;i+=THREAD_NUM)
  {
        sum+=num[i]*num[i];
  }
  *result=sum;
```

```
clock time=clock()-start;
}
```

The above program need 2.1Mcycles.It is faster. than the first version to use CUD. The speed has been improved. There is a better performance. If increasing the number of threads to 512, the implementation of the reduction to 1.7M cycles. The current bandwidth is 3.54GB/s, which is larger than the previous version. But there is a gap between the bandwidth and the actual memory bandwidth.

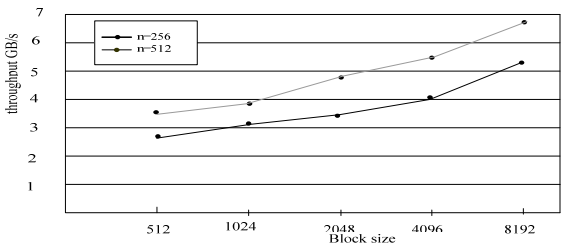

Figure3. the use of throughput

Figure 3 gives the block of threads is 256 and 512 to compare the memory bandwidth. We can see that when threads increase and the bandwidth increases .

C. Optimizing the use of instructions to get maximum throughput of instructions.

Optimizing the command is that we don't use the command with a low throughput, such as use single precision instead of double precision. As a result of SIMD, in the CUDA programming, it is necessary to avoid to use the branch prediction. For example, if stream will be interrupted by the procedure, we used the replacement of equivalent sentences.

For example, if(a>0){b+=a; c-=a}.The sentence is modified: d=(a>0); b+=a*d; c-=a*d. Rewritten the program and the source program is equivalent. Avoid – ing to use statements interrupts a program.

V. EXPERIMENTAL RESULT

All experiments use CUDA platform: GPU: GeForce 8600GT;CPU: pentium Dual E2160 1.8GHZ。CUDA parallel experiments use to rewrite and the implement-tation of the CPU .

TABLE II. EXPERIMENT RESULT

	time	Band-width
CPU	0.15 s	26.7MB/S
GPU	0.00113s	3.54GB/s

From Table II, we can see that the calculation based on the GPU than the CPU greatly improve. This shows that GPU is more suitable for large-scale computing than CPU.

VI CONCLUSIONS

Along with pioneering the field of application and development of GPU-based computing will become increasingly common maturity. CUDA-based high-performance scientific computing, data parallel processing to speed up the ability to make cheap to build high-performance computing platforms become a reality.

VII. ACKNOWLEDGEMENT

This work is Supported by the Electronic Information Industry Foundation of China under the Government. Project: Research and Application on Automatic Telemetering and Emergency Response System for Basin Water Pollution with 3S.

VIII. REFERENCES

[1] NVIDIA Corporation, CUDA Programming Guide, February 2007.

[2] NVIDIACUDA.http://developer.nivida.com/ object/cuda.html.

[3] Ye Zhao, Yiping Han, Zhe Fan, Feng Qiu, Yu-Chuan Kuo, Arie E. Kaufman, and Klaus Mueller. Visual simulation of heat shimmering and mirage.

[4] David A. Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing, IEEE Computer Society. Washington, DC, USA,2006:539–550.

[5] Buatois, L., Caumon, G., Lévy, B. 2007. Concurrent number cruncher: An efficient sparse linear solver on the GPU. Proceedings of the High-Performance Computation Conference (HPCC), Springer LNCS.

[6] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in Proceed -ings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2008.

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A united graphics and computing architecture," IEEE Micro, vol. 28, In press 2008

[8] Intel, Intel 64 and IA-32 Architectures Software Developer'sManual, May 2007.

[9] AMD CTM Guide: Technical Reference Manual. 2006. http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

[10] MacEachren, A.M., I. Brewer, and E. Steiner,(in press). Geovisualizationto Mediate Collaborative Work:Tools to

Support Different-place Knowledge Construction and Decision-Making, 20th International Cartographic Conference. ICA, Beijing,China, August 6-10, 2001.Mark, David M., Christian Freksa.

[11] Frank Losasso,Hugues Hoppe.Geometry clipmaps:terrain rendering using nested regular grids.ACM Transactions on Graphics,23(3):769-776,2004

[12] Daniel Cohen-Or,Yiorgos Chrysanthou,Claudio Silva and Fredo Durand.A survey of visibility for walkthrough applications.IEEE Transaction on Visualization and

Computer Graphics,9(3):412-431,2003

[13] Daniel Cohen-Or,Yiorgos Chrysanthou,Claudio Silva and Fredo Durand.A survey of visibility for walkthrough applications.IEEE Transaction on Visualization and

Computer Graphics,9(3):412-431,2003

[14] Daniel Cohen-Or,Yiorgos Chrysanthou,Claudio Silva and Fredo Durand.A survey of visibility for walkthrough applications.IEEE Transaction on Visualization and

Computer Graphics,9(3):412-431,2003

[15] Sud A,Otaduy MA,Manocha D.and DiFi.“Fast 3D distance field computation using graphics hardware,”Proceedings of the Eurographics,2004.