Analysis on GPU parallel computing using CUDA Programming model

Huanzhou Huang

Abstract

GPU technologies are creating amazing breakthroughs in many important fields over the last decade. The parallel computing is one of the most important part in GPU programming, which can obtain better performance on average than CPU for large blocks of data. In this paper, the parallel programming model of GPU using CUDA and their parallel optimization strategy was presented. Then, in order to analysis the advantages of GPU parallel computing and storage structure, this paper provided several algorithms including Linear Search algorithm, Breadth First Search, and K-means algorithm using GPU programming and CUDA implementation. Finally their results and the future studies were discussed.

Key words: GPU, parallel computing, CUDA, Linear Search algorithm, Breadth First Search, and K-means

1. Introduction

1.1 Graphics processing unit

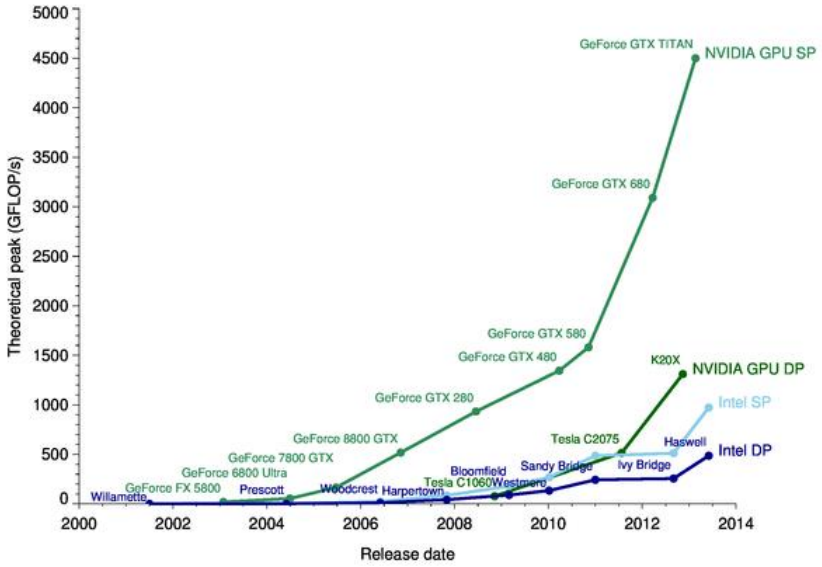
Graphics processing unit (GPU) technology is one of the milestones in the history of development of computer system architecture. Since the graphics and video processing becomes increasingly important for modern computer, there needs a specialized processor to meet these demands. GPU is heart of the graphics card as well as the graphic processing and its role is equivalent to the CPU. It is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles.[1] The reason why GPU can replace CPU as the graphics processor is that CPU not only needs to do the render, but also take the responsibility of memory management, input response and other complicated work, which will significantly reduce the performance on graphic. Additionally, the special architecture of the GPU, especially the multicore structure, can greatly help it to process the basic element of the graphic, such as pixel, improving the rendering efficiency.

Modern GPUs possess the following functions: 1. Support vertex programmability and fragment programmability. 2. Support IEEE 32 bit floating point arithmetic. 3. Support four-element vector, fourth-order matrix calculation. 4. Provide branch instructions and loop and control statements. 5. Possess high-bandwidth memory transfer capability. 6. Support the query and use for 1D, 2D, 3D Pixel and Texture. 7. Provide Render to Texture (RTT) function.

1.2 General-purpose computing on graphics processing units

Its application for computer graphics and image processing is well known, however what has only recently been explored is that modern GPUs are more powerful than CPU in general parallel computation for large blocks of data due to their highly parallel structure. It can be seen in Figure1. General-purpose computing on graphics processing units (GPGPU) is the utilization of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU).[2] The specialization in each chip of GPU can make great contribution to the performance, and may provide advantages to multiple CPUs device. Therefore the GPU programming concepts, which is based on multithreaded parallel computing, supersedes the stream processing on the CPU and plays the most important role on GPGPU.

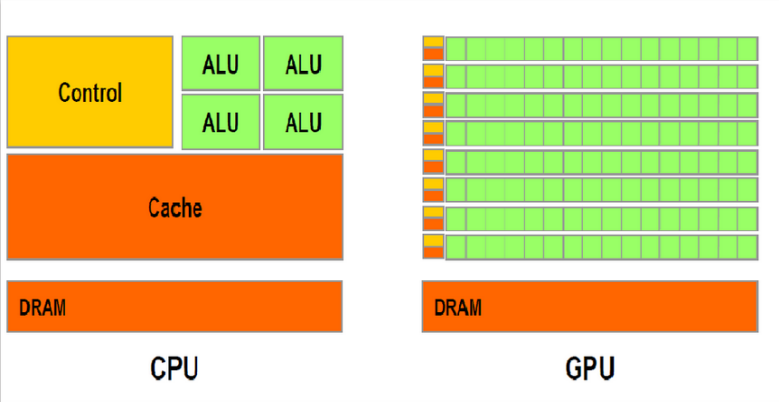
Figure1. Performance comparison between GPU and CPU from 2000 to 2014



1.3 Compute Unified Device Architecture

Figure1 shows the basic architectures of CPU and GPU. Since their difference in the number of cores, operations, and memory handling, the parallel programming and algorithm for GPU is quite different from general CPU.

Figure 2.the basic architectures of CPU and GPU is shown in figure1.



In order to build a parallel computing platform and programming model for GPU, NVIDIA created CUDA(Compute Unified Device Architecture), which provides C-style programming API and tools for GPU, and makes GPGPU(General-purpose computing on graphics processing units) more easier. To minimize the barrier to programming entry, CUDA use C language as the fundamental programming language and extend it to realize the operation of the chip, rather requiring of learning the specific instruction and architecture of chips. Therefore, the developer who has the experience in C language can quickly learn and develop GPU applications using the environment and SDK provided by CUDA.

The main functions and features provided by CUDA includes: 1. standard c language for GPU programming; 2.unified hardware and software solutions to support GPU parallel computing; 3.

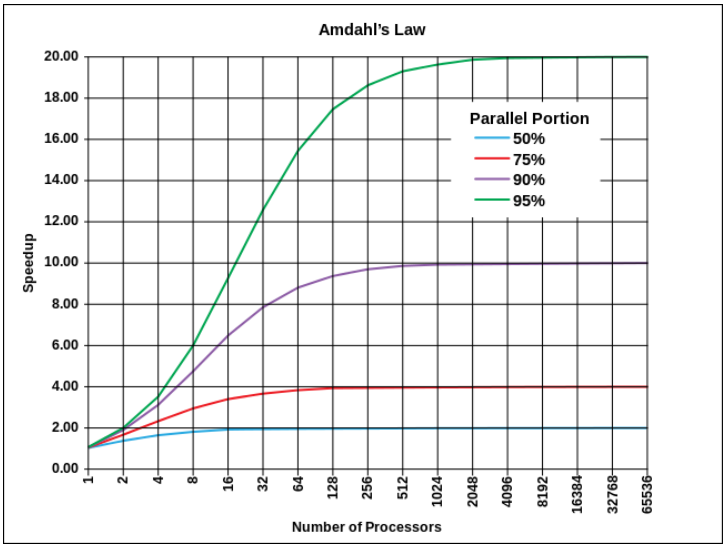
the standard fast Fourier transform and basic Linear algebra subroutine libraries; 4. compatible drivers and interfaces for OpenGL and DirectX; 5. Support direct Access to drivers.

As a parallel computing architecture combined with hardware and software, CUDA take full advantage of the large-scale computing resources of GPU, which includes several hundred cores and thousands of threads, to realize the parallel operation. CUDA and its GPU parallel computing is now widely used in picture processing, bioinformatics, video transcode, dynamics simulation, oil exploration, scientific calculation, medical diagnosis and other fields, and it also has made great contribution to improve the efficiency on different applications.

1.4 parallel computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently.[3] The speed-up of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10x no matter how many processors are used, which can be seen in Figure3. [4]

Figure3. The graphical representation of Amdahl's law.



Parallel computing implementation on algorithms has various ways. From a hardware perspective, the current parallel computing can be classified as two kinds. One is independent of parallelism, which can be divided into parallelism on multi-core and parallelism on heterogeneous pattern (GPU CPU combined). The other is based on parallel clusters, and currently the typical case is symmetrical multi-cluster system, which connects processingNodes though high-speed networks connectivity equipment to establish a multistage system structure of parallel computing. It performs coarse-grained parallelism among nodes, however performs fine-grained and simulative parallelism inside the nodes. From the perspective of software programming, the typical parallel programming model can be divided into the following kinds: MPI(Message Passing Interface) model, OpenMP multithreading model, CUDA heterogeneous parallel programming model, and mixed isomerism model. Among all the parallel programming model, the CUDA based on GPU is most popular and fastest-development technology.

Although there has been lots of accomplishments, parallel computing still remains great potential for development because it is far beyond serial algorithms in research. The common methods of parallel computing design is called PCAM: division, communication, assembly, mapping. For division, a problem is divided into several parts on average, and allow each processor to execute at the same time. Communication is aim to analyze the data which is to be exchanged their coordination. Combination require the small combinatorial problems to combine together to improve performance and reduce the overhead of task. Mapping is to assign tasks to each processor. In summary, there are still many room for improvement in parallel computing and the main difference between parallel computing and serial algorithm is that parallel computing not only needs to consider itself, but also consider the parallel model being used, network connections, and so on.

With the diversification and discretization of the practical problem, the solution time trend for combinatorial optimization problems, especially NP ((Non-deterministic Polynornial) complete problems, has grown exponentially with its size. Although the classical optimization algorithm such as Genetic Algorithms, Neural Networks and Swarm Intelligence have developed quickly in in theory and practical applications, the efficiency solution is still low and has room for improvement because of the complexity of NP problem and high-density of the data processing. Therefore, to perform parallel computing for complex algorithm is very important to solve the problems and improve efficiency.

2. Programming model of CUDA

2.1 Basic structure of CUDA Programming model

Figure4. CUDA hardware interface

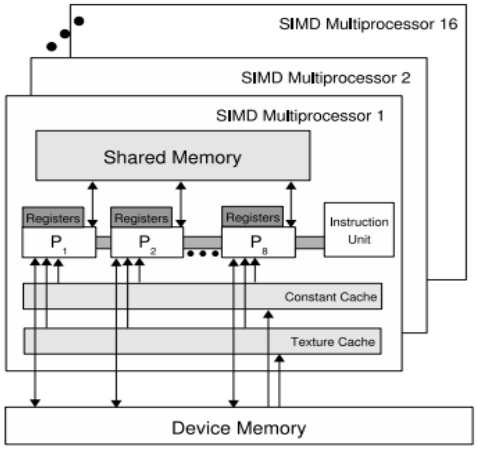
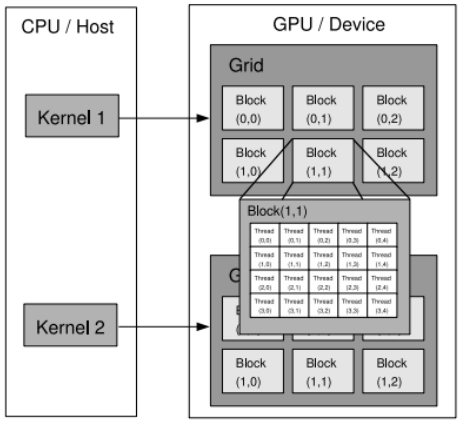


Figure5. CUDA programming model



GPU composed of many transistors, which primarily are used for data processing, not for data caching and flow control. This structure is designed for compute-intensive, highly parallel computing. Therefore, in CUDA programming model, GPU can generate massive threads as the co-processor of CPU and hide the access delay of memory by the parallel computing of these threads, rather using high-capacity cache. Under the multi-GPU memory architecture, which can be seen in Figure 4, the smallest computing unit of the chip is thread, and each thread has a private register. Multiple threads can form a thread block thread and block will be divided into smaller warps in actual operation, which can be seen in Figure 5. All the threads within the block can access and operate the shared memory. All the blocks in the multiple processors form a grid. The instruction which is executed by a single thread in GPU will turn into a kernel function. Different kernel functions will execute serially and the threads inside the kernel functions will execute in parallel. Additionally, in this structure, a program can be divided into two parts: host-side and device-side. Host-side is the operations which execute on the CPU while device-side is

on the GPU. CUDA will compile the program into the form which GPU can execute and transfer it to GPU. CUDA uses a Single Instruction Multiple execution model, this model can be summarized to following steps:

1. Data type initialization: definition and declaration of variables.

```
int *a_h;    //host-side variable
float *a_d;    // device-side variables
```

2. Storage space allocation: Allow storage space for for host-side and device-side variables in different types.

```
A_h= (int )malloc(N sizeof(int));     //host-side variable
cudaMalloc( (void**)&a_d) , N*sizeof(float) ) ;    // device-side variables
```

3. Data transfer: transfer data form the host to GPU.

```
cudaMemcpy(a_d ,a_h , Size, cudaMemcpyHostToDevice);
```

4. Parallel execution: call kernel function, which will be executed once on each thread in GPU.

```
__global__ void kernel <<<GridDim, BlockDim>>> (float* a float* b float* c);
```

5. Return result: Return the result on the GPU-side to CPU.

```
cudamemcpy(b_h, b_d, Size, cudaMemcpyDeviceToHost) ;
```

6. Release GPU memory: Call the cudaFreeO function to reclaim the space in GPU global memory space.

The following code is a matrix addition function in c language.

```
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]； } }
```

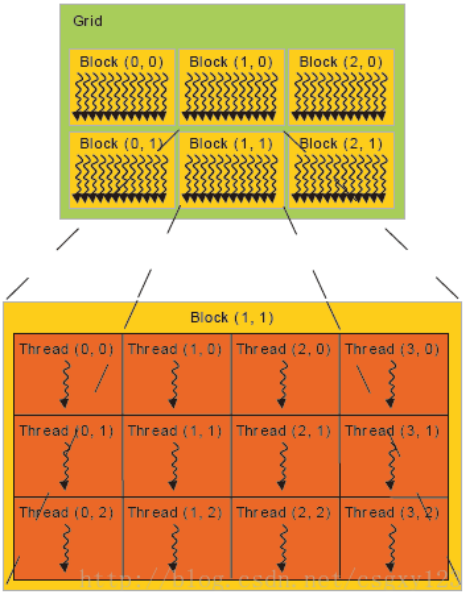
When rewriting using CUDA, it is necessary to modify the calculation part and take full advantage of the GPU parallel computing ability.

```
__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);  }
```

2.2 Thread block

Thread block is a set of threads that coordinate works. They can effectively share data though high-speed shared memory and synchronize the execution to coordinate memory accesses. More precisely, the user can specify synchronization points in the kernel and threads will hang up when they all reach the points. The number of threads in the block is limited, of which the maximum number is 1024. Fortunately the thread blocks which perform in the same kernel and have the same dimensionality and size can form into a grid of thread blocks. Each thread is identified by a thread ID. To help the addressing which is based on thread ID, the application can assign the thread block into arrays in any form such as one-dimensional, two-dimensional or three-dimensional. Then using one to three index components to identify each thread. For example, given a two-dimensional thread block (Bx,By), the ID of the thread of which the index is (x,y) is (x+yBx).

Figure6. Block and grid architecture of CUDA



Therefore, it can add the thread block and grid to improve the program's efficiency.

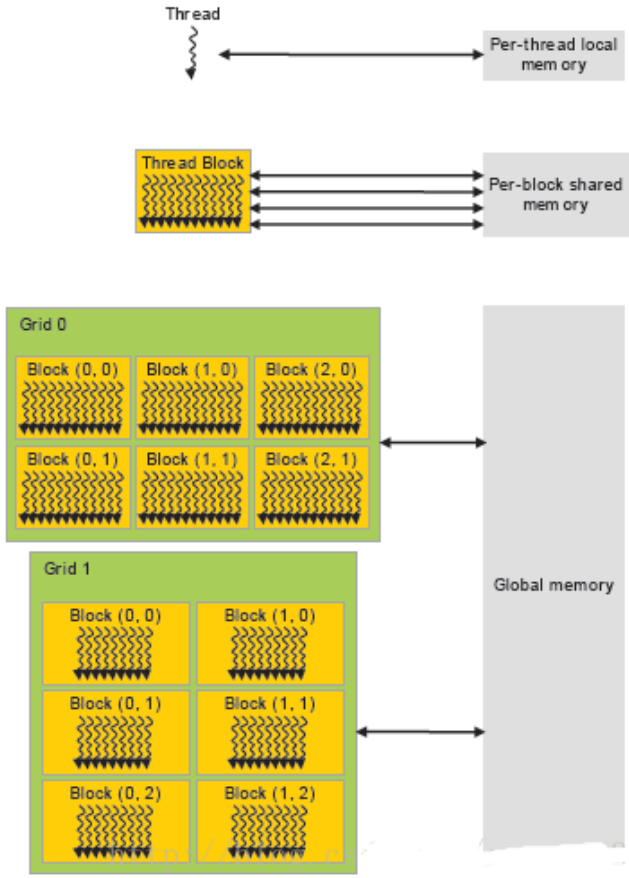
```
__global__void addMatrix(int *a, int b, int *c, int N){
int i = blockIdx.x*blockDim.x + threadIdx.x;
int j = blockIdx.*blockDim.y + threadIdx.y;
int index = i+j*N;
if(i<N && j<N)
c[index] = a[index] + b[index]; }
void main(){
dim3 blocksize(16,16);
dim3 dimGrid(N/dimBlk.x, N/dimBlk.y);
   addMatrix<<<1,blocksize>>>(a,b,c,N); }
```

2.3 Shared memory

In CUDA, the memory can be divided into two categories: system memory and device memory. Because the reading and writing speed of memory for GPU computing in device is faster than it in system, it is necessary to allocate the system memory for data and allocate the device memory for threads to do the read-write operation before the kernel function calls. The threads in the device can use the following types of memory: registers, local memory, shared memory, global memory, constant memory and texture memory. In the above example for matrix addition, the threads can run effectively and independently and no thread know which element can be accessed by other threads. It is because CUDA use shared memory to achieve this feature. Kernel stores the data into shared memory, which can be used in the communication of threads which are in the same thread block. A thread block is only processed by a multiple-processor,

therefore the shared memory resides on the chip, resulting in a quick access for memory. Additionally, because the register and shared memory of multiple-processor allocate threads to the thread block, the number of thread block which can be processed for each multiple-processor depends on how many the registers and shared memory each thread blocks need. If there is not enough available registers or shared memory to process at least one thread block, the kernel will not run.

Figure7. Memory architecture of CUDA



2.4 Thread Synchronization

Once threads are run in parallel on the same memory, it should provide a mechanism to ensure that before a thread is completed, another thread cannot access the result in memory. CUDA provide a __syncthreads() function to guarantee this mechanism. __syncthreads() is an atomic operation, which keep program continue only if threads in the block must be all synchronized.

2.5 Optimization strategy

Although the GPU can perform better than CPU under the suitable parallel computing condition, its performance mainly depends on the algorithm and program design. The division of the program module, the data structure design and the access of threads and memory are crucial parts for GPU parallel computing. Therefore how to optimize the algorithm to suit the GPU architecture is an important subject which can help improve the efficiency greatly. Performance optimization can be mainly considered by three aspects: 1. Maximize the parallel threads to take full advantage of the parallel computing. 2. Map the operations to the graphics hardware as much as possible. 2. Optimize memory usage to maximize the memory bandwidth and avoid the operations on the low-bandwidth memory 3. Change the data in the shared memory of the multiprocessors 4. Optimize the instruction usage to achieve maximum instruction throughput. 5. Choose the best memory access mode.

Without parallel threads, GPU cannot compete with CPU because the unit frequency of GPU is much lower than CPU. Additionally, CUDA need to use CPU to copy data to GPU memory, which is on the global memory. There is no cache architecture in global memory and it

will take thousands cycle to access and wait for the addition result to continue if using only one thread. Using multithreading can avoid the delay because when one thread access the memory and wait the result, CPU will switch another thread and access the next location on memory. Therefore the operation on the threads can hide the delay on the operations on global memory.

Using the maximum memory bandwidth to transfer data is a good optimization strategy for memory because the bandwidth between CPU and device is much low than it between device and global memory. Moreover, different memory types have their different access patterns, and their efficient bandwidth may on different order of magnitude. Therefore to choose the best memory access mode is important for access performance of memory. The memory of graphics card is DRAM, of which the best access mode is continuous access mode. Specifically given an array, let thread0 read the first element and let thread1 read the second and so on.

3. Algorithms using CUDA parallel computing

3.1 Linear Search algorithm

Linear search is a common task in computer programming, which is to find one element from the start to end. In text preprocessing or indexing, it is common for us to use Linear search to get one specific word or phrase in an unsort document. Using serial algorithm on CPU will take too much time when the document or text file reach a big size. Fortunately, it is perfectly suited for parallel computing on CUDA to deal with linear search problems.

3.1.1 Serial algorithm for linear search on CPU.

This algorithm is very straightforward:

```
boolean LinearSearch(char *file, int length, char *target, int targetLen){
  for (int i=0; i < length-targetLen; i++) {
    boolean match = ture;
    for(int j=0; j < targetLen && match; j++)
      if (file[i+j] != target[j]) match = false;
    if(match) return I;}}
```

3.1.2 Linear search algorithm on CUDA

For parallel computing on GPU, it requires spitting the task into many pieces and translates every pieces to running separately and simultaneously on different threads. One good approach is to map each character in the document to a single thread. Therefore you will generate a few million threads if your document has characters in that order of magnitudes. Then it can be determined by each thread if the characters match the target string from a corresponding index. However, because the termination condition of this program is finding one correct target,

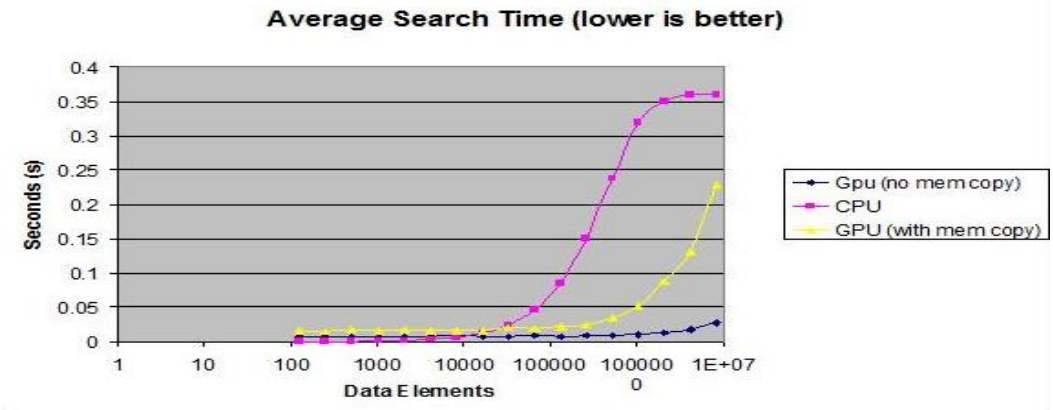
it is necessary to do thread communication for terminating the procedure. CUDA provide a standard reduction function called atomicMin function to store data on the global memory, which can be used in storing the first occurrence of the target. Then each thread should check the global memory before doing search.

```
__global__ void cudaSearch (char *file, int length, char *target, int targetLen, int *found){
  int start = blockDim.x*blockIdx.x + threadIdx.x;
  if(*found > start) {
    boolean match = true;
    for (int i=0; i < targetLen; i++)
      if (file[start+i] != target[i]) match = false;
    if (fMatch) atomicMin(found, start);}}
```

3.1.3 Result of Linear search algorithm

CUDA experiment is conducted on a PC with core i7 930, and a GTX 470.

Figure 8. Average search time for GPU programming and CPU programming



Since the GPU has to run more threads for the grown elements, the CUDA kernels still need to take increasingly more time. However, the parallel computing performance of the search has increased a lot compare to the serial algorithm on the CPU. Additionally, it is found that to copy a portion of the data on the same memory and then doing search, rather doing memcopy function, can reduce time greatly for GPU.

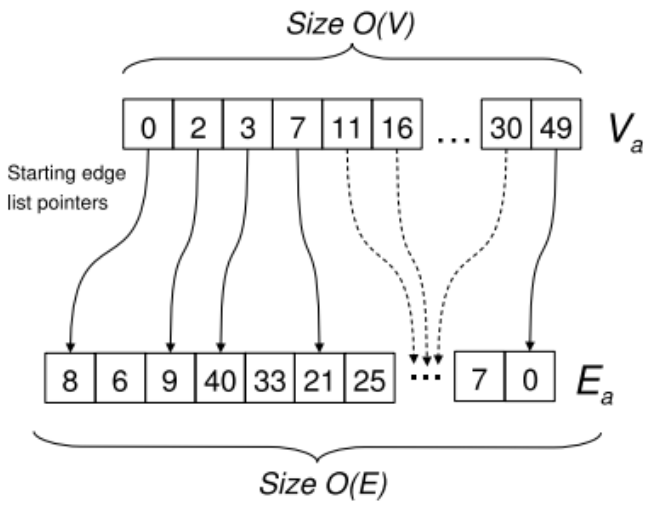
3.2 BFS Algorithms

Graph algorithms are also very suitable in CUDA parallel computing model. Since large graph involves millions of vertices, using stream processing on CPU may suffer from the scale of the graph and will take too much time. Although using high-end computing resource is a good choice, they are only a few to access. GPU is a practical and efficient platform to deal with the graph problems because it is easy accessible for researchers and all the elements in the graph problems can be disguised and interpreted as graphics rendering.

3.2.1 Graph Representation

Graph can be represent in compact adjacency list. In the list, each elements, which stands for the vertex, points to the starting position of its adjacency. Va is an array which represented as the graph G(V,E) and Ea is an array which store the edges. Each element in Va points to the corresponding index of adjacency in the edge array Ea. Vice versa. See Figure 9.

Figure 9. Graph Representation



3.2.2 Breadth First Search on CUDA

Breadth first search starts with any node, which is designated as the root node. When exploring a node, all unvisited neighbors are designated as the child nodes of this node, and all child nodes are "scanned" before doing a deep dive into any of those node. Once all the neighbors have been scanned, then the control shifts to the child nodes, and all of the child nodes are explored before any of the grand child nodes are explored. Therefore, this traversal mechanism ensures that nodes are explored in the order of their "level", that is, their distance from the designated root node.[5] At this level, using level synchronization is a good solution for BFS problem in CUDA implementation.

It is inappropriate to create a queue to store each vertex during traverse the nodes because it will cause grid configuration to change at every level and overheads of maintaining indices of queue. In CUDA implementation, mapping each vertex to one thread is a good way because it can maximize the using of threads. Additionally, it is necessary to use two arrays Fa, Xa to store the access information of frontier and the visited nodes, which can be defined as Boolean type

and the index of array stands for vertex. Moreover, to save the result, using an array Ca to store the minimal number of edges of each vertex from the root node S.

Algorithm 1 CUDA_BFS (Graph $G(V,E)$, Source Vertex S)

```
1: Create vertex array V_a from all vertices in G(V,E),
2: Create edge array E_a from all edges in G(V,E)
3: Create frontier array F_a, visited array X_a and cost array C_a of size V.
4: Initialize F_a, X_a to false and C_a to ∞
5: F_a[S] ← true
6: C_a[S] ← 0
7: while F_a not Empty do
8:     for each vertex V in parallel do
9:         Invoke CUDA_BFS_KERNEL(V_a, E_a, F_a, X_a, C_a) on the grid.
10:    end for
11: end while
```

Algorithm 2 CUDA_BFS_KERNEL (V_a, E_a, F_a, X_a, C_a)

```
1: tid ← getThreadID
2: if F_a[tid] then
3:     F_a[tid] ← false, X_a[tid] ← true
4:     for all neighbors nid of tid do
5:         if NOT X_a[nid] then
6:             C_a[nid] ← C_a[tid]+1
7:             F_a[nid] ← true
8:         end if
9:     end for
10: end if
```

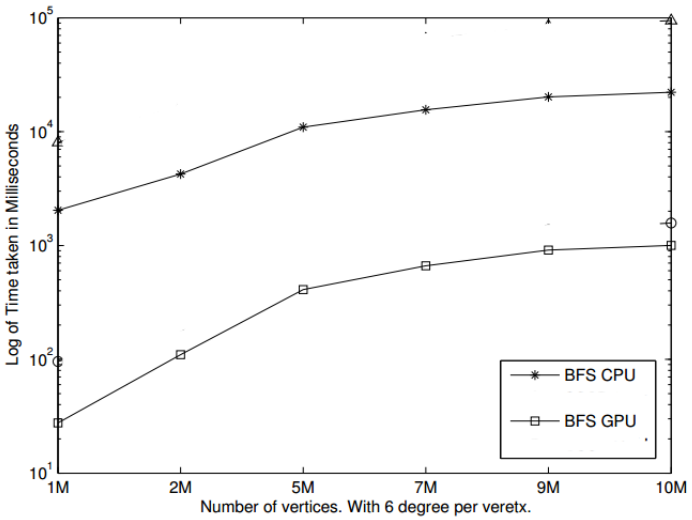
In the algorithm, each vertex will traverse its node at Fa. If there exist an edge, it will update the number of edges in Ca using edge list Ea. Then the traversed node in Fa will remove its own entry and add the current nodes into the visited array Xa. It will also detect its neighbor and put the unvisited nodes to the Fa. Finally do the loop until the Fa is empty.

3.2.3 Result of Breadth First Search

CUDA experiment is conducted on a PC with Intel Core 2 Duo E6400 2.3GHz processor, and an NVidia GeForce 8800 GTX GPU.

In figure 10, it can be seen that GPU is capable of performing in large scale of graph and it is better than CPU clearly.

Figure 10. Result of BFS algotirhm using CPU programming and GPU programming



3.3 K-means algorithm

Clustering is an important technology in information retrieval and data mining. It is an effective mean to analyze the data and find useful information in large-scale data. Clustering classify the data object into many classes or clusters, making the objects in the same cluster have a high degree of similarity and the objects in the different clusters have great difference.

3.3.1 K-means algorithm on CPU

Among all the Clustering algorithms, the K-means algorithm, k-means algorithm is one of the most common and most typical algorithm which use distance as the evaluation for similarity. The main flow of K-means algorithm is as follows: 1. Randomly selected k objects in the given objects of n as the initial mean value of each cluster. 2. For the remaining n-k objects, calculates a distance from the center of k and assign the object to the closet cluster. 3. Update each new cluster and figure out k new center. 4. According to the new center of k, repeat 2 and 3 steps until the criterion function become convergence.

Although the flow of k-means algorithm is straight forward and easy to realize, the time complexity of this algorithm is O (N*M*k), in which N stands for the data size, M is the number of traversal, k is the calculation times for each data. Meanwhile, it needs to repeatedly read the original data and write the comparing results to classification set for many times. Therefore it may low the efficiency and take too much time when the dataset is large.

3.3.2 K-means algorithm based on GPU

In order to take full advantage of the threads and parallel computing, it is a good strategy to map each thread into each data object. Then using each data object to compute the center distance from all the clusters and assign the data object to the nearest cluster. (Figure 11)

Figure11 Data allocation strategy on GPU

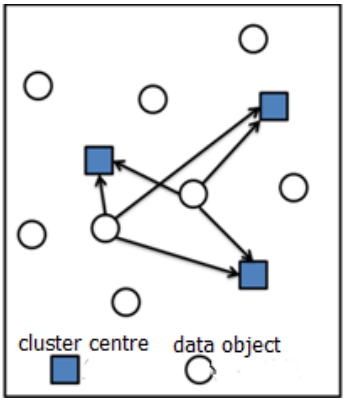
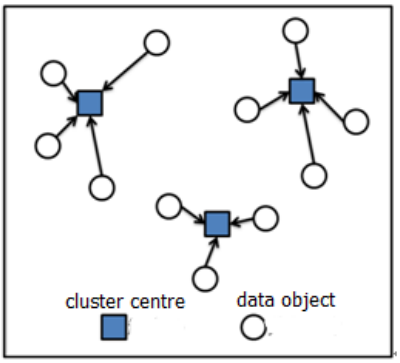


figure12 update the center of clusters



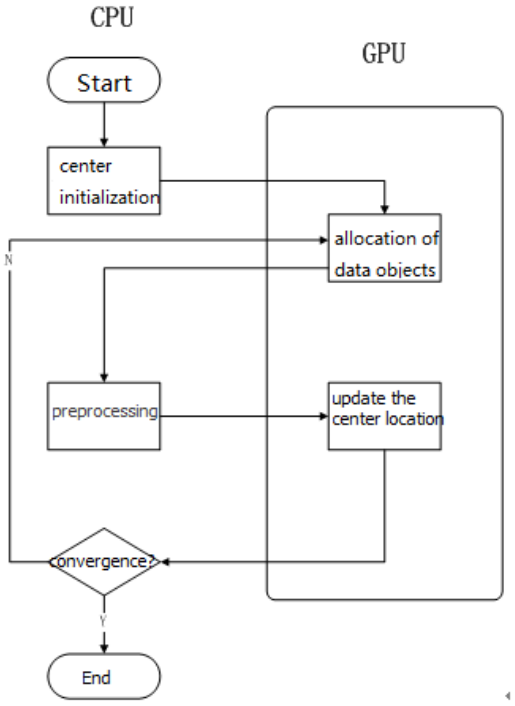
The new cluster center is calculated by the mean of data objects in the original cluster. The location of the K cluster can be calculated by GPU where each thread in CUDA is responsible for a cluster center.

After the distribution of the data objects, it can get the index of the data objects in their clusters. When recalculate the center location of each cluster, an intuitive idea is to read all the data objects and then determine which cluster the objects belong to. However, if a large number of threads on the GPU require access to shared memory simultaneously, it could cause the access delay. Therefore it is necessary to transfer the allocation result of objects form the GPU to CPU and then let the host to relocate and calculate the number of the data objects in each cluster. After that, the processed data will send to shared memory and it will not cause delay.

The main difference between the K-means on CPU and GPU is that GPU-based algorithm make the data independent of the set and transfer the high-density computing part from the host to the device. More specifically, although the process of data assignment and updating of clusters of k need do many times, the calculations and the data is independent each other.

Therefore it can be separated into two single kernel functions and let the GPU do parallel computing.

Figure13 the main flow of K-means algorithm based on GPU



As the figure13 shows, CPU is responsible for randomly generating the initial cluster centers of K and the preprocessing of result in the allocation of data objects. In addition, the convergence judgement is operated by CPU. The GPU is responsible for the high-density computing for each independent data. As mentioned above, although this CPU and GPU-based heterogeneous pattern will generate additional data I / O time-consuming, the multithreading and parallel computing can reduce the delay of the memory access. Moreover, the data of the cluster center, which is transfer between host and device should be as small as possible because the memory bandwidth of GPU is much higher than memory bandwidth of CPU.

4. Conclusion

This paper first introduces the key concepts of GPU, GPGPU, CUDA and parallel computing, which gives the researchers a blueprint of the GPU parallel computing structure. Then it provides detailed information on basic structure of CUDA Programming model including the CUDA hardware interface, CUDA Single Instruction Multiple execution model, the structure of Thread block and Shared memory, Thread Synchronization and their optimizing strategy. In order to analysis the advantages of GPU parallel computing and storage structure, this paper provided several algorithms including Linear Search algorithm, Breadth First Search, and K-means algorithm using GPU programming and CUDA implementation.

The key for GPU parallel programming is to carefully design the algorithm to expose substantial fine-grained parallelism and decompose the computation into independent tasks that perform minimal global communication. [6] Specifically, how to map threads into the elements and how to control memory access is a crucial part GPU parallel programming. In the Linear Search algorithm, each character in the document is given a thread to do the search. In the Breadth First Search algorithm, each vertex in graph is performed by a thread and be present in the global edge array. In K-means algorithm, each data object is independent by a thread and the CPU and GPU have the division of works. If any of those roles change, it may cause great performance reduction to the result. In addition, the problem may be limited by the size of the device memory, thus how to design the workflow to steam the data from CPU to GPU and handle problem by a CPU and GPU based heterogeneous pattern is also a breakthrough point.

In the future, the GPU technology will no doubt keeping on developing. Since the original objective of GPU is to do graphic rendering, lacking of high precision calculation is a bottleneck for GPU to compete with CPU. However modern GPU are continuously improving its

computing part and GPU companies has announced that they are developing double precision calculation support for GPU now. In addition, the NVidia has announced that its Tesla range of GPUs will have larger memory capacity, which makes the approaches of external memory become feasible. Therefore the application and algorithm using the GPU parallel computing will benefit from the improvement of ability of calculation and change its program structure to fit the new feature. Moreover, to further improve the efficiency of parallel processing and deal with big data, using multiple GPUs in parallel is a good way and CUDA provided the multi-GPU bridges to establish the connection. Therefore the design for program using multi-GPU is also a good research area.

5. Reference

[1] Graphics_processing_unit. (n.d.). In Wikipedia. Retrieved March 14, 2015, from http://en.wikipedia.org/wiki/Graphics_processing_unit

[2] Fung etal. (2002) Mediated Reality Using Computer Graphics Hardware for Computer Vision.: Proceedings of the International Symposium on Wearable Computing 2002 (ISWC2002), Seattle, Washington, USA, pp. 83–89.

[3] Gottlieb, Allan; Almasi, George S. (1989). Highly parallel computing. Redwood City, Calif.: Benjamin/Cummings. ISBN 0-8053-0177-1.

[4] Parallel_computing. (n.d.). In Wikipedia. Retrieved March 14, 2015, from http://en.wikipedia.org/wiki/Parallel_computing

[5] Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Proc. 14th International Conference on High Performance Computing, pp. 197–208 (2007)

[6] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In IPDPS ’09: Proceedings of the 23th IEEE International Symposium on Parallel&Distributed Processing, pages 1–10, Washington, DC, USA, 2009. IEEE ComputerSociety.

[7] Li, Y., Zhao, K., Chu, X., & Liu, J. (n.d.). Speeding up k-Means algorithm by GPUs. Journal of Computer and System Sciences, 216-229.

[8] The Supercomputing Blog. (n.d.). Retrieved March 18, 2015, from http://supercomputingblog.com/cuda/search-algorithm-with-cuda/

[9] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for highperformance computing. In SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, page 47, Washington, DC, USA, 2004. IEEE Computer Society

[10] NVIDIA Corporation. NVIDIA CUDA Programming Guide, Nov. 2007. Version 1.1.