A CPU-GPU Data Transfer Optimization Approach Based On Code Migration and Merging

Cong Fu*, Yanlong Zhai*, Zhenhua Wang†, Muhammad Mudassar*, Zishuo Wang*
*Beijing Engineering Research Center of Massive Language Information Processing and Cloud Computing Application
School of Computer Science, Beijing Institute of Technology, Beijing, China
{fucong, ylzhai}@bit.edu.cn
†Beijing Institute of Control Engineering, Beijing, China
zhenhuaw233@hotmail.com

Abstract—**Recent development in graphic processing units (GPUs) has subverted the traditional CPU-only computing. Because of the high performance and parallelism, GPUs are widely used as coprocessors to run highly parallel computations. However, porting applications to CPU-GPU architecture remains a challenge to average programmers, which have to explicitly manage data transfers between the host and device memories, and manually optimize GPU memory utilization. In this paper, we proposed an approach to optimize the data transfer operations between CPU and GPU by analysing the data dependency and reorganizing source code. Based on our previous work, we found that not only the data transmission through PCIe bus is time consuming, but also the preparation and cleaning work for data transfer operations. This cost will be dramatically increased if the program contains many kernel calls and sometimes the situation is getting severe when programmers are not sophisticated enough to make GPU programs. Therefore, we firstly defined and analyzed the data copy in (out) path for each data transfer operation utilizing compiler techniques. The data copy in or copy out operation can be migrated along with its data copy path. Multiple data transfer operations could be merged into one operation if they are of the same transfer direction and their data copy paths have overlap. Migrating and merging multiple data transfer operations could obviously reduce the number of data exchange times and the system resource consumption. The experimental results show that our approach has a significant effect on optimizing the parallel programs especially when the program has numerous kernel calls.**

Keywords—**GPU, data transfer, optimization, code migration, merging**

I. Introduction

Currently, even entry-level PCs are equipped with integrated Intel GPUs capable of hundreds of GFLOPS. Initially, GPUs were only used for graphics related computing, but due to their high processing power, now they are used as coprocessors to help CPU deal with high parallel and high computing operations. This helped a lot in performance gain and speed up the processes between 4x and 100x [1] [2] [3], but writing excellent parallel code is still a challenge for average programmers. For example, CUDA is a simple C-like interface proposed for programming NVIDIA GPUs, but it is difficult to port applications to CUDA which needs programmers to manage data transfer between the host and GPU memories explicitly [4]. How to manage data transfer operations to optimize parallel programs is the focus of this paper.

In recent years, researchers around the world have adopted a number of ways to optimize parallel programs. Some optimizations are based on the GPU physical properties, such as vectorization (An array is represented as a vector and modify the code logic to use the correct element of the vector) [5] to make full use of GPU cores, using read only data cache to store constants [3]. Other optimization methods are based on common compiler optimization, such as loop invariant code motion (moving the invariant part out of loop structure) [6] [7] and loop unrolling (reducing or eliminating instructions that control the loop) [8]. Table I shows some of the methods that have been applied to optimize parallel code.

The memory of CPU and GPU are isolated, so when the program switches between different processors, it is required to copy the data into the corresponding memory [11] [12]. Data transfer between CPU and GPU is based on PCIe bus which bandwidth is about 16GB/s [13] [14]. But the GPU memory bandwidth can be up to 100GB/s-250GB/s which is about 8 times of the CPU

Table I: parallel code optimizations

Type	Optimization
Based on GPU	Vectorization
	Parallel loop swap
	Texture fetching
	Coalesced global memory access
	Using read only data cache
General optimizations	Loop unrolling
	Common sub-expression elimination
	Loop invariant code motion

memory bandwidth. Therefore, the data transfer through PCIe bus must be the bottleneck of the system. Due to the huge gap of the bandwidth, there will be a demonstrable delay when programmers can't manage data transfer reasonably on PCIe bus [15]. Our previous work designed a CPU/GPU collaboration framework based on Hadoop MapReduce to process the big data [9]. We introduced a data flow analysis approach to optimize the data transfer between CPU and GPU [10]. The approach can significantly improve the performance of the original program. But we also found that the performance is not only affected by the data transmission via PCIe, but also some system preparation work for opening and closing data transfer, especially in a parallel program with many kernels functions. Consumption of data transfer operations consists of transmission consumption on PCIe and system consumption for initializing and freeing. High frequency data transfer becomes the major factor of some inefficient programs. So optimizing this type of program becomes intensely important, particularly when programmers using some high level GPU grogram languages like OpenAcc, which may generate many kernel function calls.

Using async data transfer can hide some cost, but it can not reduce actual cost of data transfer. This paper proposes a new approach to reduce system consumption for initializing and freeing by reducing the number of times of data transfer. We optimize the GPU source code by merging multiple data transfer operations after migrating them along the control flow based on compiler technique. Typically, a GPU kernel function call contains four steps: copy data into GPU memory, execute kernel function, copy result back to CPU memory, free data in GPU, and for most of time these four steps are called next to each other. Data copy operations of different kernel functions do not have any overlap, which makes the program easy to understand, but some times it is not efficient enough. In our work, we firstly analyse the data dependency of each data transfer operation, and define the data migration range for each of these operations. The data migration range is actually the positions in the source code where the data transfer operation can be moved to without changing the result of the program. The migration range of different data transfer operations may have some overlaps which means multiple data transfer operations can be migrated to the same position. After that, the best execution positions are calculated according to the algorithms defined by this paper with the aim of merging multiple data transfer operations into a single data transfer. It is reasonable to make two transfer operations of different kernels executed at the same time if they are of the same transfer direction and the merging will not affect the result. By merging data transfer operations, we can reduce the number of times of data copy between CPU and GPU, even the total amount of data to transfer.

II. Migration of Data transfer operations

Because of the different architecture, the CPU is good at performing sequential calculations, I/O, and program flow, and the GPU is well suited for parallel data processing. Therefore, program runs on a CPU/GPU environment always contains CPU code and GPU code. CPU code usually has complex control logic and lower parallelism, GPU as a coprocessor has the ability to quickly perform complex mathematical and geometric calculations. In the program flow, each kernel function requires two data transfers between CPU and GPU: the corresponding data need to be copied to GPU memory (hereafter referred to as the copyin) before kernel execution, and then copy the generated result back to CPU memory (hereafter referred to as the copyout) after the execution of kernel function. Afterwards, the data are freed(hereafter referred to as the data freeing) from GPU memory if it is not going to be used anymore.

A. Migration constraints of data transfer operations

A kernel execution process is divided into four separate phases: data copyin, execution, data copyout, and data freeing. Obviously, phases cannot be reversed, a kernel must be executed in the order of data copyin→execution→data copyout→data freeing. But the time that each phase start to execute is adjustable. Data copyin operation can move forward alone the code control flow until the code line changes the data to be copied in. Data copyout operation can move backward along the code control flow to delay the data transfer, but it must be done before CPU code using the corresponding data. This is actually the data dependancy of the variables in the program. The data dependancy can be represented as the *def-use chain* and *use-def chain* and calculated by the compiler technique *data flow analysis*. After data copyout, GPU will perform data freeing operation immediately. But, if the data will be used in follow-up kernels again, we could make GPU keep holding the data until the follow-up kernels executed. This will reduce the total amount of data exchanges.

B. Migration by code motion

Now we will introduce how to adjust four phases described previously and how to migrate data transfer operations by code motion. As shown in listing 1, the initial CUDA code structure describes that, data copyin, execution, data copyout, and data freeing operations are an integral whole. As shown in listing 2, data copyin operation is executed in advance while data copyout and data freeing operations are postponed after migration. Apparently these migrations of data transfer operations do not affect result of the parallel program. Data copyin operation can move up along the code control flow, and it is legal to copy data to GPU memory before kernel execution and after the data changed in CPU code. Similarly data copyout operation can move down along the code control flow, and it is legal to copy out data to CPU memory after kernel execution and before the data used in CPU code. We call the conversion from the code structure in Listing 1 to the code structure in Listing 2 migration of data transfer operations.

```
__global__ void kernel(int *device, int i);
int host[1000] = {0};
int tmp[1000] = {0};
for (int i = 0; i < 1000; i++)
    host[i] = host[i] + 1;
......
for (int i = 0; i < 1000; i++)
    tmp[i] = tmp[i]+1;
int *device;
CUDAMalloc(&device, sizeof(int) * 1000);
CUDAMemcpy(device, host, sizeof(int) * 1000,
   CUDAMemcpyHostToDevice); /*data copyin*/
for (int i = 0; i < 1000; i++) /*execution*/
    kernel<<<1,1>>>(device, i);
CUDAMemcpy(host, device, sizeof(int)* 1000,
   CUDAMemcpyDeviceToHost); /*data copyout*/
CUDAFree(device); /*data freeing*/
......
for (int i = 0; i < 1000; i++)
    tmp[i] = tmp[i]+1;
```

Listing 1: Initial CUDA code

```
__global__ void kernel(int *device, int i);
int host[1000] = {0};
int tmp[1000] = {0};
for (int i = 0; i < 1000; i++)
    host[i] = host[i]+1;
int *device;
CUDAMalloc(&device, sizeof(int) * 1000);
CUDAMemcpy(device, host, sizeof(int) * 1000,
   CUDAMemcpyHostToDevice); /*data copyin*/
......
for (int i = 0; i < 1000; i++)
    tmp[i] = tmp[i]+1;
for (int i = 0; i < 1000; i++) /*execution*/
    kernel<<<1,1>>>(device, i);
......
for (int i = 0; i < 1000; i++)
    tmp[i] = tmp[i]+1;
CUDAMemcpy(host, device, sizeof(int)* 1000,
   CUDAMemcpyDeviceToHost); /*data copyout*/
CUDAFree(device); /*data freeing*/
```

Listing 2: After migration of data transfer operations

III. Migrating algorithms

A. Basic block descriptions

Because of the loop and conditional logic in the program, the source code are not executed sequentially, some parts of the code may be excluded for execution. Therefore, the whole structure of the program need to be modelled for analyzing the possible migration ranges. In this paper, we leverage the concept of Basic Block (BB) from complier theory [20]. In figure 1(a), the code structure is divided into four BBs, a BB is a straight-line code control flow with no branches in except entry and no branches out except exit. A BB has only one entry point and only one exit point, so it must be entered from its entry and must be exited from exit [22]. In figure 1(b), BB_I, BB_J, BB_K, BB_L and control blocks branch (B), merger (M) make up the code structure described in figure 1(a). BB_I is executed prior to BB_L, expressed as $BB_I < BB_L$. Each BB between BB_I and BB_L meets principle of $BB_I \leq BB \leq BB_L$. We have defined BBs which are in a pair of control blocks B and M as *BBs. BB_K and BB_J are *BBs in figure 1(b), with control blocks B, M making up a branch structure. B_n, M_n, and all *BBs which meet $B_n <$ *$BB < M_n$ make up a branch structure $BRANCH_n$. Apparently not all *BBs will be executed, in figure 1b, within both BB_K and BB_J, there's one and only one will be executed. If migrating a data transfer operation into a *BB, data exchange operation between CPU and GPU might not be executed, causing program to crash. The program does not know specific execution process in compile phase, unable to determine which *BB will be accessed. So each data exchange operation must be placed outside each branch in the code control flow to ensure implementation.

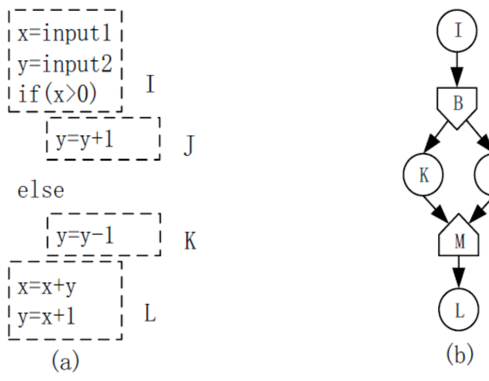

Figure 1: Control Flow Graph

B. Migration algorithms for data transfer operations

Entire program structure is composed of many code lines. Code at N-th line is expressed as $LINE_N$, and whole program structure is made up of $[LINE_1, LINE_{end}]$. Migration algorithms for data transfer operations are to obtain a largest set of code lines (copypath) to migrate data copyin or data copyout operations. This paper optimizes parallel programs from the perspective of data transfer, so we don't need to migrate kernel executions in parallel programs. Based on the BB, data copyin and data copyout operations are moved with code line unit. These algorithms are aimed at a single kernel, so we don't care whether the data after data copyout in GPU will be used again by subsequent kernels. After data copyout operation, GPU will free the data immediately in data freepath.

1) Obtain data copyinpaths: A data copyin operation can move upward along code control flow until current code line or current BRANCH changes (kills) the corresponding data. $LINE_i$ does not kill dataA expressed as $\neg kill$(dataA, $LINE_i$), and each code line in $BRANCH_i$ does not kill dataA expressed as $\neg kill$(dataA, $BRANCH_i$). Algorithm 1 takes moving upward dataA copyin operation as an example, initial line of dataA copyin is lower bound of upward movement. Current BB is initial BB, and initial copyinpath is empty. If current BB is not *BB, moving dataA copyin operation upward along code control flow until code line would kill dataA or exit current BB. The line begins to move and line to exit make up a moving interval, then add the interval of code lines to copyinpath. If meeting exit point in current BB to exit, continue moving up and if killing dataA to exit, quit. When current BB is *BB, if all *BBs in current BRANCH do not kill dataA, moving up to exit current BRANCH and continue to generate copyinpath, otherwise quit. Final copyinpath is a largest set of code lines for moving dataA copyin operation.

Definition 1: Given a dataA copyin operation at $LINE_n$, $\exists LINE_m$ with $\neg kill$(dataA, $[LINE_m, LINE_n]$) and $kill$(dataA, $LINE_{m-1}$), $[LINE_m, LINE_n]$ $-$ codelines in BRANCHs is dataA copyinpath.

2) Obtain data copyoutpath and freepath: Data copyout operations can move down along code control flow until current code line or current BRANCH changes or gets value of the corresponding data (*use data*). $LINE_i$ does not use dataA expressed as $\neg use$(dataA, $LINE_i$), and each code line in $BRANCH_i$ does not use dataA expressed as $\neg use$(dataA, $BRANCH_i$). Algorithm 2 describes how to obtain copyoutpath and freepath of dataA.

Definition 2: Given a dataA copyout operation at $LINE_n$, $\exists LINE_m$ with $\neg use$(dataA, $[LINE_n, LINE_m]$) and use(dataA, $LINE_{m+1}$), $[LINE_n, LINE_m]$ $-$ codelines in BRANCHs is dataA copyoutpath.

IV. Merging of data transfer operations

After migration, it is legal to make data copyin operations and data copyout operations executed at any code line in the corresponding copypaths. In a highly parallel program with multiple kernels, each kernel is no longer an independent entity, copyinpaths and copyoutpaths of different kernels will overlap. As shown in figure 2 right, dataA copyinpath overlaps dataB copyinpath, merging two data copyin operations into one. Two things happened here, first we define a new data structure dataAB which size is size of dataA plus size of dataB in CPU memory, and copy dataA and dataB to dataAB. DataAB is a contiguous area of CPU memory, so we make dataAB copyin operation replaces dataA copyin and dataB copyin. Obviously, copying dataA and dataB to dataAB requires extra time and extra space in CPU memory. Assume that the memory space is sufficient,

Algorithm 1: Obtain dataA copyinpath

```
Input: copyinpath = $\emptyset$, $LINE_i$ = $LINE_j$ = initial
       line of dataA copyin operation
1  if current BB is not *BB then
2      while $LINE_i$ in current BB and $\neg$kill(dataA, $LINE_i$) do
3          $LINE_i = LINE_i - 1$;
4      end
5      update current BB based on $LINE_i$;
6      copyinpath = copyinpath +($LINE_i$, $LINE_j$];
7      if kill(dataA, $LINE_i$) then
8          end algorithm;
9      end
10 end
11 if $\neg$kill(dataA, current BRANCH) then
12     current BB = the BB before current BRANCH;
13     $LINE_i$ = $LINE_j$ = the last code line in current BB;
14     goto 1;
15 else
16     end algorithm;
17 end
```

Algorithm 2: Obtain dataA copyoutpath and freepath

```
Input: copyoutpath = $\emptyset$, $LINE_i$ = $LINE_j$ = initial
       line of dataA copyout operation
1  if current BB is not *BB then
2      while $LINE_j$ in current BB and $\neg$use(dataA, $LINE_j$) do
3          $LINE_j = LINE_j + 1$;
4      end
5      update current BB based on $LINE_j$;
6      copyoutpath = copyoutpath +[$LINE_i$, $LINE_j$);
7      freepath = [$LINE_j$, $LINE_j$];
8      if use(dataA, $LINE_j$) then
9          end algorithm;
10     end
11 end
12 if $\neg$use(dataA, current BRANCH) then
13     current BB = the BB after current BRANCH;
14     $LINE_i$ = $LINE_j$ = the first code line in current BB;
15     goto 1;
16 else
17     end algorithm;
18 end
```

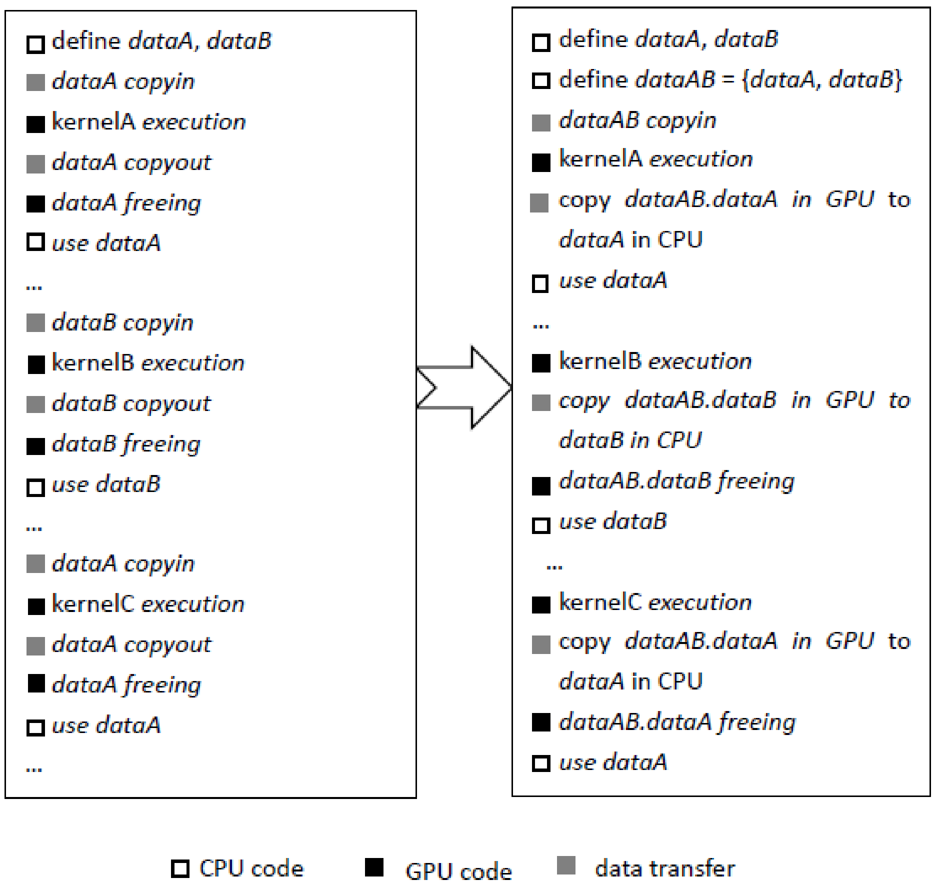

Figure 2: Data transfer operations merging

due to high CPU memory bandwidth (Intel Xeon Phi x200 processor, a single Haswell chip can deliver about 60 GB/s sustained memory bandwidth [16] [17]), extra time spent on CPU is very small. The total amount of data exchange does not reduce after merging, but the number of times to copy data to GPU operations decreased from two to one, reducing system consumption for opening and closing data transfer operations, making the best use of transmission bandwidth.In figure 2 left, GPU frees dataA immediately after dataA copyout operation. CPU must copy dataA into GPU again before executing kernelC. Actually, dataA copyin operation of kernelC is not necessary, because this data is same as the data which kernelA frees. In figure 2 right, dataA freeing operation of kernelA is cancelled and GPU memory will keep dataA, so kernerlC reuse dataA of kernelA and dataA copyin operation of kernelC is cancelled too. The total amount of data exchange is reduced after merging, hence speeded up program processing.

A. Merging algorithms for data transfer operations

Merging algorithms for data transfer operations consist of two part, one is merging data freepaths and copyin paths of different kernels to reduce the total amount of data exchange by data reuse, the other is merging data copyths to reduce the frequency of data transfer. Data copyinpaths, copyoutpaths and freepaths are obtained by Algorithm 1 and Algorithm 2, now we will describe how to merge data copypaths and freepaths of different kernels optimally.

1) Merging data freepaths and copyinpaths: GPU memory is not always enough, and it's useless for GPU to keep the data which will not be used in follow-up kernels. Based on four properties (data, copyinpath, copyoutpath, freepath) in each kernel, Algorithm 3 presents a smart solution.

Definition 3: Given the kernels K_i and K_j with K_i.data==K_j.data and K_i.freepath $\cap$ K_j.copyinpath $\neq \emptyset$, the data freeing of K_i and data copyin operation of K_j can be cancelled, and K_j will reuse the data of K_i.

2) Merging data copypaths: This algorithm will make each data transfer operation executed at a fixed code line in corresponding data copypath, then we will merge data copyin or copyout operations if they are at the same code line. Algorithm 4 proposes a greedy strategy to merge data copypaths to minimize the number of times to copy data between CPU and GPU memories. Merging data copyoutpaths is the same as merging data copyinpaths,

Algorithm 3: Merge data freepaths and copyinpaths

```
1 foreach kernel $K_i$ in kernel set do
2     foreach kernel $K_j$ in kernel set do
3         if $K_i$.data == $K_j$.data and $K_i$.freepath $\cap$ $K_j$.copyinpath $\neq \emptyset$ then
4             $K_i$.freepath = $\emptyset$;
5             $K_j$.copyinpath = $\emptyset$;
6         end
7     end
8 end
```

so we just describe how to merge copyinpaths. Initial kernel set contains all kernels. Define an array W, W[i] is the number of times LINE$_i$ is covered by all copyinpaths in current kernel set.

Definition 4: Given the kernels K_i and K_j with K_i.copyinpath $\cap$ K_j.copyinpath $\neq \emptyset$, the data copyin operations of K_i and K_j can be merged into one data copyin operation.

Algorithm 4: Merge data copyinpaths

```
1  while kernel set is not empty do
2      W[1, end] = {0};
3      foreach kernel $K_i$ in kernel set do
4          foreach $LINE_j$ in $K_i$.copyinpath do
5              W[j] = W[j] + 1;
6          end
7      end
8      n = W.findMaxIndex();
9      foreach kernel $K_i$ in kernel set do
10         if $LINE_n$ in $K_i$.copyinpath then
11             $K_i$.copyinpath = [$LINE_n$, $LINE_n$];
12             Remove $K_i$ from kernel set;
13         end
14     end
15     goto 1;
16 end
```

V. Evaluation

Since our target is to optimize data transfer between CPU and GPU, we don't care what GPU code does and we can't reduce GPU execution time by the algorithms proposed in this paper. What we focus on is how to optimize parallel program by reducing the total amount and frequency of data exchange. So we take data transfer volume and data transfer frequency as experimental parameters, and take run-time of all data transfer operations which consist of all transmission consumption on PCIe and all system consumption for initializing and freeing as evaluating indicator.

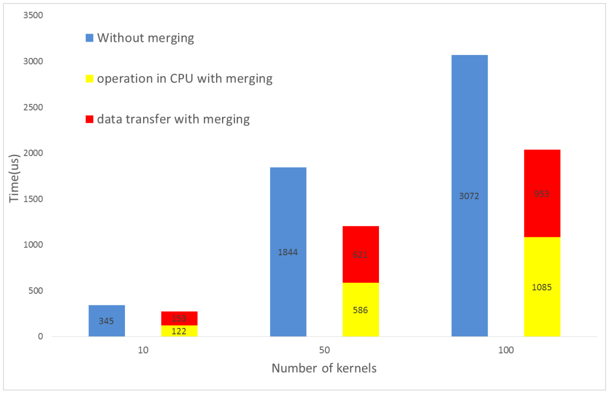

Figure 3: Matrix size per kernel = 10 KB

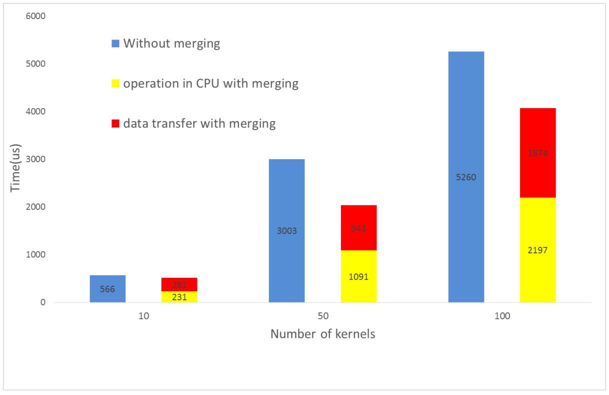

Figure 4: Matrix size per kernel = 20 KB

A. Experimental platform

We use an Intel(R) Xeon(R) E5-2620 v2 @2.10GHZ CPU with 1.5MB of L2 cache to be the host CPU for the GPU. All GPU codes were executed on an ASPEED Graphics Family (rev 21) video card, a CUDA device with 4,799 MB of global memory. The CUDA driver version is release 5.5. The nvcc compiler release 5.5, V5.5.0 compiled all CUDA programs using default optimization level.

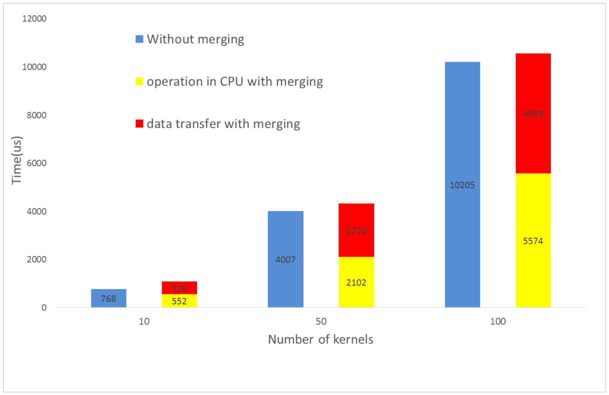

Figure 5: Matrix size per kernel = 50 KB

Table II: Comparing parallel programs with and without merging data transfer operations

amount of data transfer per kernel	number of kernels	without (us)	with(us)	
			copying operations in CPU	data transfer
1 KB	10	257	16	61
	50	1139	57	102
	100	2378	141	170
	500	11867	678	633
10 KB	10	345	122	153
	50	1844	586	621
	100	3072	1085	953
	500	18942	5699	5020
20 KB	10	566	231	281
	50	3003	1091	943
	100	5260	2197	1874
	500	26301	8835	9835
50 KB	10	768	552	526
	50	4007	2102	2220
	100	10205	5574	4989
	500	51428	21654	22298

B. Experimental results

Our experiments are performed with and without merging data transfer operations under a set of parallel programs for matrix addition with different matrix sizes and different numbers of kernels. In order to facilitate the experiment, in each parallel program, we let the matrix size per kernel is same and we make all data copyin operations and all data copyout operations can be merged into one data copyin operation and one data copyout operation. Figure 3, 4 and 5 show the data transfer time in microseconds for parallel programs without data transfer merging in comparison to with data transfer operations merging. The three histograms show the experimental results of parallel programs with different matrix sizes per kernel 10KB, 20KB, 50KB and different numbers of kernels 10, 50,100. The data transfer time of parallel program with data transfer operations merging is composed of the time to copy the data of different kernels to a new continuous CPU memory (yellow part in figure 3) and the time to transfer the continuous data to GPU memory (red part). From the experimental results we can see that the data transfer time of program with data transfer merging (red part) is obviously shorter than the data transfer time of program without data transfer merging (blue part). Because the red part contains little system consumption for initializing and closing data transfer operations. But when there is increase of the amount of data transferred per kernel, the proportion of the time spent copying operations in CPU memory (yellow part) is greater and greater, and even makes data transfer operations merging algorithm have side effect on parallel program. So, the algorithms proposed in this paper have significant optimization effect on data transfer of parallel programs when the data transfer volume is relatively small and data exchange is frequent. Table II shows more experimental results.

VI. Related Work

Although there has been prior work on optimizing parallel computing for CPU-GPU architecture, these implementations have not addressed how to migrate and merge data transfer operations of different kernels between CPU and GPU memories. Many scholars, according to the memory structure and multi-core characteristic of GPU have proposed some optimization methods to optimize parallel computing on CPU-GPU architecture.

First we can optimize parallel programs to make full use of GPU computing resources and GPU memory resources through manual code analysis and modifications. Vectorization can be applied by using vector data types provided in CUDA as data structure and modify the code logic to use the correct element of the vector in other expressions [4] [18]. Vectorization can make full use of multi-core characteristic of GPU and open more threads to run parallel computing. Texture memory can be used in kernels by invoking texture intrinsic provided in CUDA [21]. It needs to be bound explicitly to CUDA array allocated in device memory. Rational use of texture memory can significantly optimize parallel programs.

Then, there are some optimizations on parallel programs in the compilation phase. Loop Collapsing is performed by merging double or more nested loops into a single loop which can be done by loop analysis and transformation within the compiler [4] [19].

Loop Collapsing can increase the degree of parallelism compared with initial nested loops, and GPU will open more threads to run the loop. Thread and Thread-Block Merging can be achieved by duplicating the global memory array expressions identified by the programmer with incrementing the array indices and modifying the main loop partitioning based on the thread granularity defined by the programmer [4]. And using read only data cache to store constants in GPU can speedup GPU memory access [3]. And researchers found that optimizing programs by analyzing the structure of source code and reorganizing source code has a significant optimization effect. For example, at any point in time of program process, the program can't make full use of all system computing resources [22]. Because computing resources have some different types, program generally takes up one at a code line in program. Some researchers have proposed a smart algorithm that compiler migrates the subsequent code to current point legally to use free computing resourse. Data dependency and program logic are the most important criterions of code migration for programmers. A reasonably strategy of code migration can speed up program process by making program take up more system computing resources.

VII. Conclusion

We propose a new method to optimize parallel computing on CPU-GPU architecture by just optimizing data transfer between CPU and GPU memories. We don't reduce the transmission consumption on PCIe, but we reduce the system consumption for initializing and freeing by reducing the number of times of data transfer. Our algorithms consist of two parts, one is migrating each data transfer operation by code motion in parallel program, and the other is merging data transfer operations of different kernels with greedy strategy after migrating. Through a series of experimental results, our algorithms have been proved to have a significant optimization effect on parallel computing when the data transfer volume is relatively small and data exchange is frequent. In a parallel program which has more than 10 kernels and the average amount of data transfer each kernel less than 20KB, we can reduce the total data transfer time by about 20% by the algorithms proposed in this paper.

Acknowledgment

We thank the entire Beijing Institute of Control Engineering for their support and feedback during this work. We also thank Wenjun Tao for generously proposing their useful amendments and addendum to this paper. Additionally, We thank the anonymous reviewers for their insightful comments.

This work is supported by NSFC (National Natural Science Foundation of China) grants 61602037.

References

[1] D. M. Dang, C. Christara, and K. Jackson, *GPU pricing of exotic cross-currency interest rate derivatives with a foreign exchange olatility skew model.* SSRN eLibrary, 2010.

[2] D. R. Horn, M. Houston, and P. Hanrahan, *A streaming hmmer-search implementation.* Proceedings of the Conference on supercomputing (SC), 2005.

[3] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. Hwu, *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA.* In proceedings of the Thirteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2008.

[4] AH Khan, M Al-Mouhamed, M Al-Mulhem, and Adel F. Ahmed, *RT-CUDA: A Software Tool for CUDA Code Restructuring.* International Journal of Parallel Programming, 2016.

[5] A Kawai, K Yasuoka, K Yoshikawa, and T Narumi, *Distributed-shared CUDA: Virtualization of large-scale GPU systems for programmability and reliability.* The Fourth International Conference on Future Computational Technologies and Applications (FUTURE COMPUTING), 2012.

[6] P Colea, F Luporini, and PHJ Kelly, *Generalizing loop-invariant code motion in a real-world compiler.* MEng Computing Individual Project, 2015.

[7] Diogo Sampaio, Alain Ketterlin, Louis-Noel Pouchet, and Fabrice Rastello, *Hybrid data dependence analysis for loop transformations.* Parallel Architecture and Compilation Techniques (PACT), 2016.

[8] Manato Hirabayashi, Shinpei Kato, and Masato Edahiro, *Accelerated Deformable Part Models on GPUs.* IEEE Transactions on Parallel and Distributed Systems , 2016.

[9] Yanlong Zhai, Emmanuel Mbarushimana, Wei Li, Jing Zhang, and Ying Guo, *Lit: A High Performance Massive Data Computing.* international conference on cluster computing, 2013

[10] Yanlong Zhai, Ying Guo, Qiurui Chen, Kai Yang and Emmanuel Mbarushimana, *Design and Optimization of a Big Data Computing Framework based on CPU/GPU Cluster*. IEEE International Conference on High Performance Computing and Communications,2013

[11] B. Van Werkhoven, J. Maassen, F. J. Seinstra, *Performance Models for CPU-GPU Data Transfers.* Cluster, Cloud and Grid Computing (CCGrid), 2014.

[12] Janghaeng Lee, Mehrzad Samadi, and Yongjun Park, *Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems.* Parallel Architectures and Compilation Techniques (PACT), 2013.

[13] Fernando Martinez Vallina and Spenser Gilliland, *Performance optimization for a SHA-1 cryptographic workload expressed in OpenCL for FPGA execution.* IWOCL '15 Proceedings of the 3rd International Workshop on OpenCL, 2015.

[14] D. Cesini, A. Ferraro, and A. Falabella, *High throughput data acquisition with InfiniBand on x86 low-power architectures for the LHCb upgrade.* Real Time Conference (RT), 2016.

[15] Akira Nukada, Yasuhiko Ogata, Toshio Endo, and Satoshi Matsuoka, *Bandwidth intensive 3-D FFT kernel for GPUs using CUDA.* Proceedings of ACM/IEEE conference on Supercomputing, 2008.

[16] Jason Howard, Saurabh Dighe, and Yatin Hoskote, *A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS.* Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010.

[17] A. Jain, W. Anderson, and T. Benninghoff, *A 1.2 GHz Alpha microprocessor with 44.8 GB/s chip pin bandwidth.* Solid-State Circuits Conference, 2001.

[18] David F Bacon, Susan L Graham, and Oliver Sharp, *Compiler transformations for high-performance computing.* ACM Computing Surveys, 1944.

[19] Mikhail Plotnikov, Andrey Naraikin, and Elmoustapha Ouldahmedvall, *Collapsing of multiple nested loops, methods and instructions.*

[20] Ying Chen, *Fundamentals of Compiling.* Beijing Institute of Technology press, 2001.

[21] J M Schneider and Rudiger Westermann, *Compression domain volume rendering.* ieee visualization, 2003.

[22] Luiz C. V. dos Santos and Jochen A. G. Jess, *A Reordering Technique for Efficient Code Motion.* ACM, 1999.