Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Optimizing non-coalesced memory access for irregular applications with GPU computing*

Ran ZHENG‡[1,2,3,4], Yuan-dong LIU[1,2,3,4], Hai JIN[1,2,3,4]

[1]*National Engineering Research Center for Big Data Technology and System, Huazhong University of Science and Technology, Wuhan 430074, China*

[2]*Services Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan 430074, China*

[3]*Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan 430074, China*

[4]*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*

E-mail: zhraner@hust.edu.cn; 1531364016@qq.com; hjin@hust.edu.cn

Received May 24, 2019; Revision accepted May 31, 2020; Crosschecked Aug. 10, 2020

Abstract: General purpose graphics processing units (GPGPUs) can be used to improve computing performance considerably for regular applications. However, irregular memory access exists in many applications, and the benefits of graphics processing units (GPUs) are less substantial for irregular applications. In recent years, several studies have presented some solutions to remove static irregular memory access. However, eliminating dynamic irregular memory access with software remains a serious challenge. A pure software solution without hardware extensions or offline profiling is proposed to eliminate dynamic irregular memory access, especially for indirect memory access. Data reordering and index redirection are suggested to reduce the number of memory transactions, thereby improving the performance of GPU kernels. To improve the efficiency of data reordering, an operation to reorder data is offloaded to a GPU to reduce overhead and thus transfer data. Through concurrently executing the compute unified device architecture (CUDA) streams of data reordering and the data processing kernel, the overhead of data reordering can be reduced. After these optimizations, the volume of memory transactions can be reduced by 16.7%–50% compared with CUSPARSE-based benchmarks, and the performance of irregular kernels can be improved by 9.64%–34.9% using an NVIDIA Tesla P4 GPU.

Key words: General purpose graphics processing units; Memory coalescing; Non-coalesced memory access; Data reordering

https://doi.org/10.1631/FITEE.1900262 **CLC number:** TP319

1 Introduction

In recent years, general purpose graphics processing units (GPGPUs) are becoming an integral part of modern system architectures. Given their massively parallel architecture, graphics processing units (GPUs) can significantly accelerate many regular, data-parallel applications, such as GEneral Matrix-matrix Multiplication (GEMM) and computational finance. However, the benefits of GPUs are not as useful for irregular applications, such as n-body simulation (Wang L et al., 2015), fluid dynamics (Pickering et al., 2015), social networks (Alandoli et al., 2016), data mining (Song et al., 2015), and graph processing (Fan et al., 2013; Wang Y et al., 2015).

The performance of a GPU application depends mainly on the data transfer time between a CPU and a GPU and the execution time of a GPU kernel. The

‡ Corresponding author

* Project supported by the National Key Research and Development Program of China (No. 2018YFB1003500)

ORCID: Ran ZHENG, https://orcid.org/0000-0002-3058-7581

data transfer time is limited by hardware resource (e.g., PCIe bandwidth), and the size of transferred data cannot be reduced in most cases. The kernel execution time is sensitive to memory access efficiency, which is sensitive to the presence of irregularities in an application. This phenomenon is due to the memory architecture and single instruction multiple threads (SIMT) execution model of GPUs. Table 1 lists the fractions of the GPU kernel execution time and CPU-GPU data transfer time of three irregular programs: conjugate gradient (CG), survey propagation (SP), and molecular dynamics (MD). The execution time of kernels with irregular memory access consumes a large fraction in all three irregular programs. Thus, improving performance by optimizing the irregular memory access in irregular applications has great potential benefit.

Table 1 Time proportions of irregular applications

Program	Time proportion		
	PCI-e	Irregular kernels	Others
CG	7%	85%	8%
SP	1%	94%	5%
MD	49%	51%	–

CG: conjugate gradient; SP: survey propagation; MD: molecular dynamics

In recent decades, several solutions have been presented to solve the irregularity problem. Fung et al. (2007) and Meng et al. (2010) have focused on hardware extensions to redesign the warp of a modern GPU, but it is difficult to make this a popular method. Baskaran et al. (2008), Lee et al. (2009), and Yang et al. (2010) have proposed certain software solutions that use compiler techniques to reorder or relocate data before running the program. These solutions are useful for static memory access but inapplicable to dynamic memory access, where memory access patterns are unknown until runtime. For dynamic memory irregularity, Zhang et al. (2011) and Wu et al. (2013) have proposed several data reordering algorithms on CPUs to eliminate irregularity. However, reordering data on CPUs is inefficient, and the overhead is high at times for large-scale data. Another shortcoming for reordering data on CPUs is an extra overhead for transferring the reordered data.

In this study, we propose a software solution with no hardware extensions to solve dynamic irregularities. In our solution, a data reordering approach is presented to eliminate indirect memory access (e.g., $A[B[tid]]$). To improve efficiency, data reordering is offloaded to a GPU, and the overhead is reduced by reordering data in parallel. Index indirection is a mechanism for ensuring accuracy, thereby redirecting the reference of each thread in irregular kernels to the reordered data. To reduce the overhead of reordering data and maximize the performance of the whole program, two optimization methods are proposed. For loop-carried applications, reordered data are cached for the next iteration or other kernels to eliminate possible redundant data reorganization. A pipeline mechanism is presented to reduce the overhead of data reordering.

2 Related work

Many real-world applications with GPU computing exhibit varying but consistent degrees of irregularity (Burtscher et al., 2012). To improve the performance of irregular applications further, an increasing number of studies have focused on irregularity when mapping to a GPU. Combining a software approach with hardware extension methods can achieve better optimization results. For example, it is possible to exchange memory access contents and branch processing between two threads by adjusting warp scheduling from hardware, so that more optimized data adjustment methods can be designed from the software level to deal with more complex irregular memory access. However, hardware extension is expensive, and optimizing efforts are highly dependent on and limited by hardware. Therefore, we focus mainly on software methods to optimize non-coalesced memory access for irregular applications.

Fung et al. (2007) proposed a hardware mechanism to recover the lost performance potential of the hardware for a shader program. They designed a hardware unit, namely, a thread scheduler, to form new warps and select one warp to execute every cycle. With a lane-aware mechanism, scalar threads, whose next program counter (PC) values are the same, are combined to form new warps dynamically. In addition, five scheduler policies are explored to select one warp to a single instruction multiple data (SIMD) pipeline every cycle. A dynamic warp subdivision, rather than warp formation, is also suggested to eliminate branch and memory divergence (Meng et al.,

2010). Warps in GPUs are selectively subdivided into warp-splits with fewer threads than the available SIMD width, but they can be individually regarded as an additional scheduling entity. Thus, stall cycles are reduced, thereby resulting in improved latency hiding and memory-level parallelism. However, regardless of warp formation or subdivision, it is necessary to design the hardware units and expand real GPU systems to add these hardware units.

Baskaran et al. (2008) presented a compiler framework to optimize global and shared memory access. After compilation, effective transformation codes were generated to access global memory efficiently, and the array copied into a shared memory was suitably padded to minimize bank conflicts in the shared memory access. Lee et al. (2009) presented a compiler framework for an automatic source-to-source translation of OpenMP applications into compute unified device architecture (CUDA) applications. During translation, two compiler-time techniques, namely, parallel loop-swap and loop-collapsing, were used to optimize global memory access. CUDA-Lite, introduced by Ueng et al. (2008), uses an automated tool to eliminate static memory access. This transforms non-coalesced global memory access to coalesced shared memory access. Li et al. (2015) presented a probabilistic modeling method of performance analysis and estimation for sparse matrix-vector multiplication (SpMV) on GPU, so as to choose the most appropriate format from coordinate (COO), compressed sparse row (CSR), ELLPACK (ELL), and hybrid (HYB). Yang et al. (2010) developed a compiler to use a GPU memory hierarchy and manage parallelism efficiently. Vectorization was used to coalesce global memory access, and tiling and unrolling were used to reuse data and manage parallelism. Li et al. (2016) presented a direct and iterative mixed method for the quasi-tridiagonal equation, in which a partition strategy was used to store intermediate data in shared memory to reduce the latency of memory access.

However, these optimization techniques are applicable to only static memory access or specific algorithms, but inapplicable to dynamic memory access. For applications with dynamic memory access, the memory access pattern is unknown until runtime and it varies with execution of the programs, whereas the compiler techniques can optimize only the memory access pattern in compile time.

A few explorations have focused on dynamic irregularities. A duplication algorithm (Zhang et al., 2011) creates duplicate data on a CPU and reorders the original data with every irregular memory reference; in addition, a novel technique, namely, job swapping, was proposed to eliminate an irregular control flow. Wu et al. (2013) presented two improved versions of the duplication algorithm, that is, padding and sharing algorithms, to relieve the space overhead. The basic improvement of the padding algorithm is that only one data copy is created when two threads within one warp access the same data, and the key insight into the sharing algorithm is that a shared memory is used to enlarge the scope of the duplication algorithm. However, for large-scale data, reordering data with one CPU thread is inefficient, and the overhead is high at times. Moreover, each time there is the reordering of data on a CPU, one extra overhead is available to transfer the reordered data from a CPU to a GPU. For loop-carried applications, data must be reordered many times, and the transfer overhead may be high.

Our approach uses a data reordering technique to eliminate irregular references, but original data, rather than an immediate data copy, will be divided into multiple chunks and reordered chunk by chunk in accordance with the access patterns. After each data reordering, a data copy will be cached for reuse in the next iteration or other kernel. To reduce the overhead of data reordering and caching, multiple CUDA streams and pipeline mechanisms will be used to overlap reordering with computations.

3 System analysis and design

In this section, the background of the GPU memory hierarchy and SIMT execution model, which causes non-coalesced memory access, is introduced. Then, the common memory access pattern of irregular programs is analyzed through a view of source code. Finally, the solution to the problem is suggested.

3.1 Analysis of memory access pattern

Modern GPUs are equipped with more than 2000 cores (e.g., 2880 cores in Tesla K40c) as the underlying execution unit of a GPU, and thousands of threads execute concurrently in the SIMT model on a

single GPU. In CUDA programming, 32 threads with consecutive identifiers are grouped into one warp. All threads in one warp execute in a lock step, indicating that the same memory load/store instruction is executed in 32 threads simultaneously. To maximize the global memory bandwidth, contiguous 32 or 128 bytes are combined into a segment on a GPU. If all threads in a warp access words that lie in the same segment, then only one memory transaction is required to transfer this segment, called memory coalescing. Therefore, the total number of memory transactions equals the number of segments where the requested data fall. By contrast, if the memory region that each thread accesses is non-coalesced and data fall onto multiple segments, then the number of memory transactions is greater than the possible minimum and the memory transactions are called non-coalesced memory access.

Non-coalesced memory access is common in many irregular programs. For programs that issue a sparse matrix, graph, or molecule, their memory access patterns are unknown until runtime. The main reason is the indirect data access, denoted as $A[B[tid]]$, in which tid means the CUDA thread identifier. The memory access pattern depends on the values of array B.

Listing 1 illustrates a code snippet in an SpMV kernel in a CG solver:

Listing 1 Simplified code snippet of the SpMV kernel in a CG solver

```
//tid: the global id of a thread
a = rowPtr[tid];
b = rowPtr[tid + 1];
for (int k = a; k < b; k++) {
   dest[tid] += val[k] * vec[column[k]];
}
```

In this implementation, each GPU thread computes one element. A sparse matrix is constantly stored in a CSR format. To determine the values of matrix elements, each thread must obtain the row offset first to index the array, which stores the values of elements. Multiple threads, such as the statement $k = rowPtr[tid]$, access contiguous memory regions simultaneously, and the access pattern of *rowPtr* is a coalesced memory access. By contrast, for statement $val[k]$, the memory access pattern of *val* is determined by the value of *rowPtr*. Fig. 1 depicts that the memory reference to *val* is consistently non-coalesced. If warp and segment sizes are both four, with $rowPtr[] = 0, 3, 6, 9$, then three memory transactions are required for one memory access, which is three times the minimum. These extra memory transactions exist in each loop to access the array *val*, which will cause a serious damage to the performance of the whole kernel. This phenomenon is common in most irregular programs, such as graph processing and MD simulation.

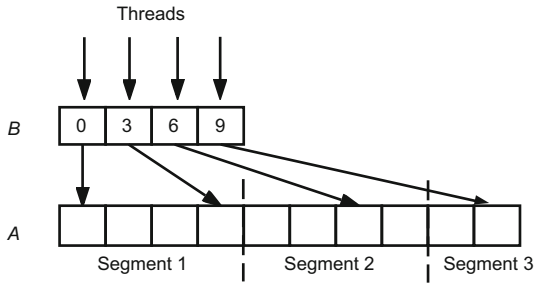

Fig. 1 An example for an indirect memory access (warp size=4, segment size=4)

In a GPU memory system, there are six types of memory: register, shared, local, constant, texture, and global memory. The first three kinds of memory are on-chip with high access speed but small capacity. The last three kinds of memory are off-chip with slow access speed but large capacity. Registers and local memories are unique to each thread, while shared memory is unique to thread blocks. In the Kepler architecture, all threads in a thread block share 64-KB capacity with L1 cache. Constant memory is a kind of memory optimized for broadcasting operations. It is often used to store constants. Texture memory is a specific memory optimized to improve the locality of a two-dimensional (2D) data space. In GPU applications, global memory is the most widely used, so research focuses on global memory.

3.2 Data reordering with CPU and GPU

The basic strategy for eliminating non-coalesced memory access is to reorder irregular data accessed by GPU kernels in a coalesced way. Zhang et al. (2011) and Wu et al. (2013) have conducted data reordering studies on CPUs, but these studies are inefficient in considering the serial processing of data and extra transfer of duplicate data.

In our solution, data reordering is offloaded to the GPU to exploit the highly parallel computing power. Fig. 2 demonstrates the schematic of reordering data with a CPU thread and with a GPU kernel. In Fig. 2a, one CPU thread is responsible for

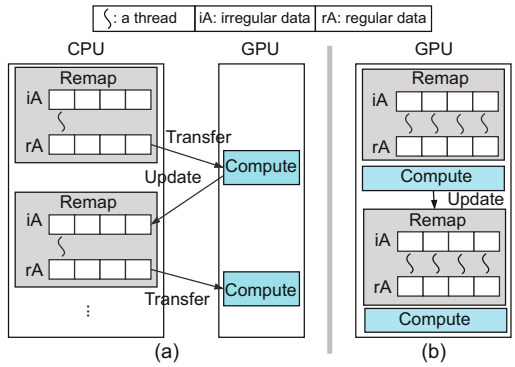

Fig. 2 Comparison of data reordering on CPU (a) and GPU (b)

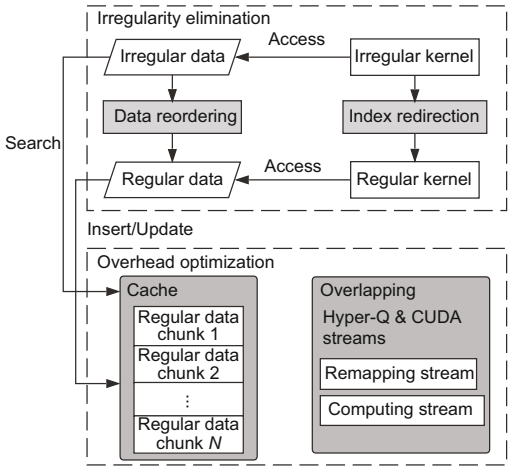

Fig. 3 System scheme of eliminating irregular memory access

reordering data. This is inefficient in comparison with the GPU kernel in Fig. 2b. When applying data reordering, all data are independent; thus, all data can be reordered in parallel, which is well suited to a GPU. Experimental results show that the time to reorder data with a GPU kernel is much less than that to reorder data with one CPU thread (described in Section 6). Another problem found in Fig. 2a is the extra overhead to transfer reordered data. In several loop-carried applications, the values of data are changed in each iteration; thus, the data must be reordered. If the data are reordered on a CPU, each time after reordering data, a transfer is necessary from the CPU to GPU for reordered data and back from the GPU to CPU for data after being updated by the GPU kernel. The multiple expensive transfers of large-scale data result in a high overhead. If data reordering is offloaded to a GPU, then no extra transfer is required because the reordered data are already in a GPU global memory.

3.3 System design

To improve the performance of the GPU kernel, a software solution without hardware extensions or offline profiling is proposed to eliminate the dynamic non-coalesced memory access. Based on the above analysis, the nature of an irregular memory reference is the inferior mapping between GPU threads and data. Thus, the basic strategy for eliminating an irregular memory reference is to enhance the mapping between GPU threads and data, and two components, namely, irregularity elimination and overhead optimization, are proposed to enable this basic strategy to work efficiently (Fig. 3).

Irregularity elimination is a component for eliminating non-coalesced memory access by creating a new thread-data mapping. Two main parts of this component are data reordering and index redirection. The former creates new data layouts and copies the original data to a new memory location; the latter transforms an original index to a new index that refers to new data layouts. Collectively, the non-coalesced memory access in the kernel is removed. There are two main aspects to eliminate non-coalesced memory access accurately and effectively, i.e., the desirable data layouts and mapping rules between threads and data, and the overhead of hiding and minimizing data remapping.

The other component, overhead optimization, hides the overhead of irregularity elimination and includes several other optimizations. To remove the dynamic irregular memory access, the desirable data layouts must be determined, and irregular data elements are copied to the new memory locations. This must occur during runtime. These additional operations will incur a high overhead to the whole program. To address this problem, hyper-Q features are used to divide remapping work and kernel execution into different CUDA streams. Hyper-Q increases the total number of hardware-managed connections between CPU and GPU reaching 32, which enables 32 CUDA streams to run simultaneously. Thus, the overhead of remapping is hidden with data transfer and kernel execution. By contrast, in certain applications, the same memory access pattern exists in multiple kernels or these kernels are involved many times. To avoid redundant data reordering, all remapping traces are cached for reuse.

4 Irregularity elimination

The essence of irregularity elimination is to find the optimal data layouts between GPU threads and data accessed, indicating that all data accessed by a warp must fall into a minimal segment for memory reference, and that all threads in a warp must proceed to one branch for a control flow. However, minimizing the number of non-coalesced memory accesses through only data repositioning is an NP-hard problem (Wu et al., 2013). To reduce complexity, several tradeoffs are performed between space and time, that is, creating data copies to reduce time for determining the optimal data layouts. Therefore, the problem is transformed into determining the rule of creating data copies.

4.1 Data reordering

In many cases, indirect data access is the reason for dynamic irregular memory references. Data duplication is used to eliminate the indirect index. A new memory region is allocated to the GPU global memory and accessed with a coalesced pattern. It is necessary to copy correct data from the original memory region to the new memory region. At an indirect memory access, that is, $A[B[tid]]$, the values of B are used only to index array A and determine the number of iterations to access the elements in A in most irregular programs. The values used by kernels are only elements in A. The idea of removing the indirect memory access is to extract elements in A indexed by B and fill these elements into the new memory region individually.

Fig. 4 presents an example of removing an indirect memory access ($A[B[tid]]$). In this example, elements in A are accessed iteratively, and the number of iterations is the maximum of the interval between two adjacent elements of B. In the i^{th} iteration, the elements in A are indexed by the sum of $B[tid]$ and i, that is, $A[B[tid] + i]$. Overall, thread i accesses elements $A[B[i]]$, $A[B[i] + 1]$, until $A[B[i+1]]$ in an iterative way. Before reordering the data, a new memory region must be allocated first to store data copies of A, denoted as new_A. The number of iterations, denoted as $maxIterNum$, is the maximum of the numbers of non-zero elements in each line. The size of one row is denoted as $rowSize$; thus, the size of new_A, denoted as $newSize$, is determined by

$$newSize = maxIterNum \cdot rowSize. \quad (1)$$

In each iteration, the elements of A accessed by the kernel will be filled in new_A consequently. In Fig. 4, the memory locations of new_A are filled by the values of $A[0]$, $A[3]$, $A[7]$, and $A[9]$. Then, the next row is filled by $A[1]$, $A[4]$, $A[8]$, and $A[10]$, until all elements of A are copied to new_A. Generally, the intervals between two adjacent elements of B are unequal, indicating that, in an irregular kernel, several threads will be idle in certain iterations. In this case, to reduce the complexity of modifying the kernel, several trivial values will be filled in new_A based on the computing work of the kernel (e.g., 0 in the $SpMV$ kernel), and their indices are equal to the identifier of idle threads. Therefore, the elements of new_A are expressed as

$$new\_A[i + iter \cdot n] = \begin{cases} A[B[i] + iter], & 0 \le iter < B[i+1] - B[i], \\ trivial\ values, & \text{otherwise}, \end{cases} \quad (2)$$

where i means thread ID, $iter$ the iteration index, and n the number of elements contained in each row of new_A. Then, all references of $A[B[tid]]$ are replaced by $new\_A[tid]$, and the references to elements of B are no longer required in the kernel, thereby possibly reducing the number of memory transactions further. Thus, the optimal data reference is achieved, and the number of memory transactions is minimized. This approach is also appropriate for the access pattern $A[B[C[tid]]]$. In this case, two-level remapping is required to create accurate and optimal data layouts.

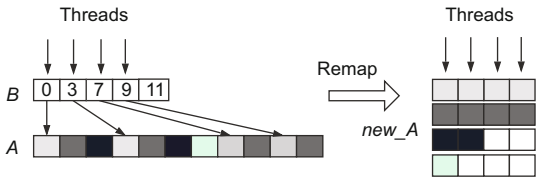

Fig. 4 An example of removing an indirect memory access (warp size=4, segment size=4)

Data reordering can achieve the optimal data layout to reduce the execution time of the CUDA kernel, but it also causes space and time overheads, in which space overhead is the main drawback of this approach. The extra space overhead is from the space costs of the copy of old data and the padding elements. The overhead is related to the maximum number of non-zero elements and number of rows of

the matrix to be processed. However, with the rapid development of GPU, the capacity of GPU global memory is becoming larger and larger. For most applications, the memory consumption of data replicas will not exceed the limitation of GPU memory capacity.

Two time overheads are caused during reordering data with the CPU thread, and similar methods were proposed in Zhang et al. (2011) and Wu et al. (2013). One overhead is the inherent cost to reorder the data and remap the references, and the other is the extra cost in transferring reordered data to the GPU. Several experiments show that, for many applications, the time of reordering data with a CPU thread is consistently longer than the execution time of an irregular kernel. Thus, hiding the overhead by overlapping the time of data reordering with the execution time of the kernel is difficult. When applying data reordering, each element is independent of each other. Thus, a novel solution is to offload the reordering task from a CPU to a GPU.

For each data chunk, a remapping kernel is launched, and one kernel thread is responsible for copying or filling one element into a new memory region. The execution time of the kernel is related to the number of rows of the matrix and the sparsity of non-zero elements per row, and the number of kernel startup times is equal to the maximum number of non-zero elements per row. With the size of the matrix or the number of non-zero elements per row increasing, the overhead of reordering data will become larger. However, by exploiting the deep multithreading characteristics of the GPU, the overhead of data reordering with the GPU is less than the overhead of data reordering with the CPU. Furthermore, this overhead can be hidden by overlapping the remapping kernel with a computing kernel, because these two kernels can be processed concurrently with the hyper-Q feature (described in detail in Section 5.1). However, the cost in transferring reordered data is eliminated because data are copied from the original GPU global memory to the new GPU global memory to save a lot of transfer time.

4.2 Index redirection

After reordering data, all computing kernels with a non-coalesced memory access must be modified to access the reordered data. The start address of the new memory region that stores each reordered data chunk will be cached, and the address is then passed to the new computing kernel with a coalesced memory access pattern. For example, the indirect data access $A[B[tid]]$ must be replaced by $new\_A[tid]$. To improve parallelism, the iterations that process all data chunks in one loop are moved outside the computing kernel, and the computing kernel is responsible for only one data chunk but is called multiple times. Accordingly, the pointer of the new memory region that passes to the computing kernel is changed each time, and the other change to the original kernel is that the access to B is eliminated.

Listings 2 and 3 illustrate an example of index redirection to the SP computing kernel. Two-level indirect memory access exists in *var_col*, and two remapping kernels are launched. One is responsible for reordering *var_rowOff* to *new_var_rowOff*, and the other is responsible for reordering *var_col* to *new_var_col* based on the result of the former remapping kernel. Array *var_col* is used to store the values that the kernel really requires, and array *var_rowOff* is used only to index *var_col*. Thus, in the reordered version, the pointer which points to *new_var_rowOff* is not transmitted to the kernel because *new_var_rowOff* no longer accesses in this kernel. One basic change in the reordered version is that the access to *var_col* is replaced by *new_var_col*; for example, $var\_col[v_j + i]$ is replaced by $new\_var\_col[tid]$. The other change is the processing of iterations. To reduce the overhead of data reordering, the loop is moved outside the kernel, which is called iteratively, and the kernel is responsible for one line of *new_var_col*. Based on the data reordering rule, each line of *new_var_col* contains all elements required in each iteration. Each time the kernel is called, *new_var_col* is changed to point to the start address of the corresponding line. For example, in the i^{th} time of calling the kernel, the pointer, which is equal to $new\_var\_col + i \cdot n$, is passed to the kernel, in which n represents the number of elements in each line of *new_var_col*.

Listing 2 A GPU kernel code example for the original version

```
__global__ void calc_pi_values(int * ed_dst,
        int* var_rowOff, int* var_col, ...)
{
    int tid = threadIdx.x + blockIdx.x
              + blockDim.x;
    int j = ed_dst[tid];
```

```
    int v_j = var_rowOff[j];
    int v_j_len = var_rowOff[j+1] - v_j;
    for (int i = 0; i < v_j_len; i++) {
        int ed_btoj = var_col[v_j + i];
    }
}
```

Listing 3 A GPU kernel code example for the reordered version

```
__global__ void calc_pi_values(int * ed_dst,
        int* new_var_col, ...) {
    int tid = threadIdx.x + blockIdx.x
            + blockDim.x;
    int j = ed_dst[tid];
    /* new_var_rowOff is no longer accessed
       in this kernel. Loop is moved outside
       the kernel */
    int ed_btoj = new_var_col[tid];
}
```

5 Overhead optimization

Dynamic irregularity elimination must occur at runtime, and this will cause a considerable overhead. For data reordering, the overhead originates from copying elements from the original data to the new memory region to generate a coalesced memory layout. However, the overhead is too high at times. Thus, an optimization mechanism for overhead minimization must be designed. The basic idea is to overlap the overhead with a computation time of the CUDA kernel through CUDA streams. By contrast, in several irregular programs, the same memory access of similar data exists in multiple kernels or iterations. Thus, each reordered data chunk can be cached to avoid redundant data reordering.

5.1 Overlapping remapping with computation

Additional overhead will be introduced when dealing with irregular memory access. Most studies have proposed some methods to overlap data transmission and kernel computation to reduce the overhead. Shredder (Bhatotia et al., 2012) is a pipelined double buffering technology to overlap communication and computation, so as to reduce the overhead of serialized GPU data transmission. BigKernel (Mokhtari and Stumm, 2014) is a CPU-GPU data transmission optimization scheme to process large datasets. A four-level pipeline data prefetching technology was used to optimize CPU-GPU data transmission. Sabne et al. (2013) proposed an automated pipeline technology to overlap computation and data transmission time. Sourouri et al. (2014) proposed a scheme to optimize data transmission within a single node with multiple GPUs. All focus on overlapping data transmission and kernel computation. In our work, the data reordering operation is completed on the GPU without additional data transmission. The technologies mentioned above cannot be directly applied in our scheme. We must make some improvement to the overlapping between data reordering and kernel computation to hide the overhead.

The essence of hiding the overhead is to perform data ordering with a computing kernel asynchronously. In many data-intensive applications, the kernel processes data in a loop, and one data chunk is issued in each iteration. The data can be divided into multiple chunks, and the loop can be moved outside the kernel. Thus, each loop of a remapping kernel can reorder one data chunk, and computing a kernel requires only processing a reordered data chunk; that is, each data chunk is processed asynchronously. When data reordering is offloaded to the GPU, the CUDA streams are used to achieve asynchronous executions. Thus, the overhead of data reordering can be hidden. The other benefit of offloading data reordering to the GPU is that device and host codes, which update the cache, are executed asynchronously. Therefore, the overhead of updating a cache is hidden by overlapping with the execution time of GPU kernels.

A CUDA stream refers to a sequence of asynchronous CUDA operations to achieve grid-level concurrency. The parallelism of CUDA streams refers to the number of CUDA streams running simultaneously, which is limited by the GPU hardware architecture and GPU resources. Different GPU architectures support different maximum numbers of runtime flows. Kernels in different streams are executed simultaneously on a single device. With hyper-Q, the new feature of the Kepler architecture, the stream number, can be set up to 32 on Kepler GPUs. At most 32 kernels execute simultaneously if a GPU resource is sufficient. In our design, for each data chunk, each remapping kernel is positioned at a different stream, followed by a computing kernel to process this data chunk. Without exceeding hardware constraints, the number of streams is equal to the number of iterations in a single process, meaning that it depends on the number of data chunks.

If the number of data chunks is too large, the size of each data chunk will be too small such that the overhead of launching a kernel will weigh on the execution time of the kernel. Given a small number of data chunks, the execution time of kernels will be so long that the overlapping time will be too small. Because GPU global memory is used here, for simplicity, the number of registers supported by a single GPU thread library is considered as a limited GPU resource. When the registers are sufficient, the size of the data chunk processed by each stream is the row data of the matrix. Otherwise, the row data will be subdivided into several blocks according to the register resources and processed several times.

A sample code to overlap remapping and computing kernels with CUDA streams is shown in Listing 4.

Listing 4 Code snippet of overlapping remapping and computing kernels

```
//Initialize the number of streams
int streams_num = DATA_CHUNK;

//Initialize the size of the data chunk
int chunk_size = size / DATA_CHUNK;

//Create one computing stream
cudaStream_t comp_stream;
cudaStreamCreate(comp_stream);

//Create multiple reordering streams
cudaStream_t *streams = (cudaStream_t *)
malloc (streams_num*sizeof(cudaStream_t));
for (int i = 0; i < stream_num; i++)
        cudaStreamCreate(&streams[i]);

//Overlap remapping and computing kernels
for (int i =0; i < stream_num; i++) {
  //Reorder data chunks concurrently
  data_remap<<<grid,block,0,streams[i]>>>
        (original_data + i * chunk_size,
  new_data+i*chunk_size,index_array);

  if (check_Done(new_data,i)) {
  //Execute the computing stream
  // asynchronously
   data_comp<<<grid,block,0,comp_stream>>>
        (new_data+i*chunk_size);
  }
}
```

Only the CPU part is presented, while the details of the CUDA kernel are ignored. In the sample code, the *data_ remap* kernel performs data reordering, and the *data_ comp* kernel processes new data. The index i, which means the serial number of the data chunk, is passed to the *data_ remap* kernel to identify the data to be reordered. Considering that the computing kernel is data-dependent, only one stream is created for the computing kernel. Multiple streams are created for the remapping kernel because the reordering of each data chunk is independent. In each iteration, one data chunk is processed in one stream. Considering the processing of the i^{th} iteration, a remapping kernel is first invoked to reorder the i^{th} data chunk in a CUDA stream, that is, *streams*[i]. Then, the CPU checks whether this data chunk has been reordered in a polling mode. If accomplished, then a device pointer is returned from the record and passed to an optimized *data_ comp* kernel, which is executed in another CUDA stream, *comp_ stream*.

Given that the device code is executed asynchronously with a host code while the *data_ comp* kernel is running, the CPU moves onto the next iteration and the *data_ remap* kernel in which reordering the $(i+1)^{\text{th}}$ data chunk is invoked in *streams*[i+1]. Running on different streams simultaneously, the overhead of reordering the $(i+1)^{\text{th}}$ data chunk is hidden by the execution time of processing the i^{th} chunk of reordered data. With a careful selection of the number of data chunks, the overhead of reordering multiple data chunks can be concealed, and the performance of the whole program is improved. A schematic diagram of the execution flow is shown in Fig. 5. However, the overhead cannot be completely hidden. Because different data execute on GPU resources, the execution time of a single stream is longer than expected, and multiple streams cannot be fully parallelized. So, we can only optimize the execution to reduce (instead of completely eliminating) the overhead.

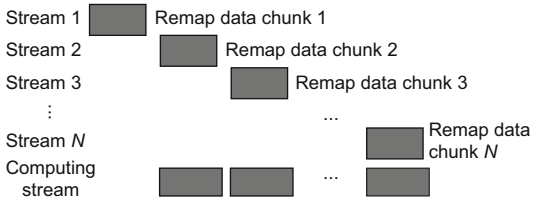

Fig. 5 Execution flow of the remapping and computing kernels

5.2 Cache

In several applications, the same memory access patterns exist in multiple kernels or iterations.

If data are reordered before invoking each kernel in each iteration, certain redundant reordering occurs, thus increasing the overhead considerably. We must identify whether the data chunk to be processed has been reordered before invoking the remapping kernel. However, determining manually whether the memory access of data in each kernel or each iteration is the same, especially for large-scale applications, is a tedious task. Thus, a software cache mechanism is designed to record every remapping. Before launching the remapping kernel, the cache is first looked up to find whether the data have been reordered. If found, then a pointer of new memory locations is returned to avoid redundant data reordering.

The basis of the cache mechanism is to record the mapping between the original and the corresponding reordered data. For each data chunk, a cache line is responsible for recording the mapping relations. To represent the relationships, each cache line consists of three fields as follows: the start address of the original data chunk (*addr_ old*), the start address of the corresponding chunk of reordered data (*addr_ new*), and a valid field (*valid*). The cache entries are indexed by the field *addr_ old*. When performing lookups, a CPU thread compares the address of processed data with *addr_ old* of each entry. If they are equal, then the value of field *addr_ new* of this entry is returned and passed to the computing kernel. Otherwise, a remapping kernel will be launched to reorder this chunk of data, and a new entry with the field *addr_ new* filled by the results of the remapping kernel will be inserted into the cache. The size of each chunk of reordered data is the same as the padding trivial value. Thus, the size of the granularity of a cache is fixed, equal to the size of one data chunk.

In several applications, the values of the original data are changed in different iterations. Thus, the values of the corresponding reordered data must also be changed. That is, the entry of this data chunk in the cache is invalid. The field *valid* is in charge of indicating whether the values of this data chunk are changed. Two states of field *valid* exist, where "1" means that this data chunk is valid and can be passed to the computing kernel directly, while "0" indicates that this data chunk is changed, and a remapping kernel must be launched to reorder this data chunk again. When the entry is first inserted into the cache, the field *valid* is set to 1. If several computing kernels modify the values of reordered data, then the field *valid* in the corresponding entry will be set to 0. The next time this invalid data chunk is processed, a remapping kernel will be launched to update the corresponding memory region. Thus, no extra data copies are generated, and the space of global memory is saved considering the in-place update.

6 Experiments and evaluation

In this section, the experimental setup and three test programs with dynamic irregular memory access are described, and the experiments and analysis of the three applications are presented.

6.1 Experimental setup

Two GPU architectures, Kepler and Pascal, are used to build two computing environments for performance evaluation. The configurations of the environments are shown in Table 2.

Table 2 Different computing configurations for experimental evaluation

Configuration	Envi_Kepler	Envi_Pascal
CPU	Intel Xeon E5 2620	Intel Xeon E5 2682
GPU	NVIDIA Tesla K40c with 2880 cores & 12-GB VRAM	NVIDIA Tesla P4 with 2560 cores & 8-GB VRAM
OS	Ubuntu 14.04.1	Ubuntu 14.04.5
Software	CUDA 7.5	CUDA 8.0

The computing architecture of K40c GPU is Kepler with computing capability 3.5, and the computing architecture of P4 GPU is Pascal with computing capability 6.1. To compare the numbers of memory transactions of irregular kernels before and after data relocation, the NVIDIA profile tool, nvprof, is used to measure the number of global memory loads/stores and the execution time of each kernel. The execution time of the application contains the total data transfer time and total execution time of kernels, whereas the preprocessing (such as loading/storing data) time is ignored. In the experiments, the size of global memory occupied by the original data and data duplications is smaller than the capacity of the GPU global memory.

Three benchmarks, namely, CG, SP, and MD, are selected to evaluate the performance benefits of our solution (Table 3). Each benchmark is the representative application of the corresponding

domains. CG originates from the real application of SpMV (Baskaran and Bordawekar, 2008) and is implemented using the CUSPARSE library. CUSPARSE (the NVIDIA CUDA Sparse Matrix library) provides GPU-accelerated basic linear algebra subroutines for sparse matrices, and is widely used by engineers and scientists working on applications such as machine learning, natural language processing, and computational fluid dynamics. SP originates from the released GPU benchmark collection of LonestarGPU v2.0 (Burtscher et al., 2012), and MD originates from the benchmark collection of SHOC v1.1.4 (Danalis et al., 2010). The three benchmarks contain the same type of irregularities and the same memory access pattern.

Table 3 Benchmarks for evaluating the efficiency of our solution

Program	Source	Description	Input
CG	SpMV	Conjugate gradient	Matrix
SP	LonestarGPU	Survey propagation	Graph
MD	SHOC	Molecular dynamics	Array

6.2 Benefits of data reordering on GPU

In many cases, indirect data access is the culprit of dynamic irregular memory references. Our proposed reordering method is executed on GPU, coming from the CPU algorithm in Wu et al. (2013), a rare dynamic irregularity elimination. Two experiments are carried out to evaluate the overload of data reordering, i.e., the number of global memory transactions and the execution time of reordering.

6.2.1 Reduction of the number of memory transactions

Fewer memory transactions mean less computing overhead. In the first experiment, we compare our proposed reordering approach with the three benchmarks with CUSPARSE, LonestarGPU, and SHOC in terms of the number of memory transactions of the GPU kernel. The results are summarized in Tables 4–6.

All results are collected by NVIDIA's CUDA profiler. Given that the access pattern of writing a global memory is coalesced, and that the codes of writing are unchanged, the write operations of the benchmarks are regular on the GPU, so the experimental results focus mainly on the optimization effects of read operations, rather than on the write operations. Therefore, the numbers of global memory transactions of CG, SP, and MD are reduced considerably; i.e., the reduction ratios are 1.9:1, 1.2:1, and 2.0:1 on average, respectively. The intuitive benefit of the decrease in the number of memory transactions is the decrease in the execution time of irregular kernels.

Table 4 Total load memory transactions of CG with different input sizes

Input size	CUSPARSE	Proposed reorder	Reduction ratio
$259\,789^2$	12 586 844	10 436 777	1.2:1
$525\,825^2$	9 737 331	4 786 588	2.0:1
$715\,176^2$	10 929 912	5 189 755	2.1:1
$999\,999^2$	11 351 417	4 857 436	2.3:1

The input size is the size of a sparse matrix's row and column

Table 5 Total load memory transactions of SP with different input sizes

Input size	LonestarGPU	Proposed reorder	Reduction ratio
16 800-4000-3	1 959 774	1 709 631	1.2:1
42 000-10 000-3	5 028 018	4 386 694	1.2:1

The input size is the numbers of clauses-literals-literals per clause

Table 6 Total load memory transactions of MD with different input sizes

Input size	SHOC	Proposed reorder	Reduction ratio
12 288	7 097 802	3 584 748	2.0:1
24 567	14 204 070	7 173 772	2.0:1
36 864	21 306 534	10 760 875	2.0:1
73 728	42 613 954	21 522 198	2.0:1

The input size is the number of molecules

6.2.2 Data reordering varying from CPU to GPU

Data reordering and transferring are two operations for data remapping in dynamic irregularity elimination. Compared with data reordering on CPU, the operations on GPU can obtain higher efficiencies in these two respects, which is the premise to overlapping data remapping with GPU kernel computation with the hyper-Q technology.

Taking the four sparse matrices as examples, the efficiency of reordering data on the GPU is compared with that of reordering data on the CPU, and the CPU of E5-2682 with 16 cores is used to support multiple reordering threads. Clearly, data

reordering on GPU is more efficient than reordering on CPU because of the highly parallel characteristics of the GPU. The testing sparse matrices are collected from the sparse matrix collection in Davis and Hu (2011), listed in Table 7. Fig. 6 shows the time to reorder data on the CPU (E5-2682 with 16 cores) and the GPU (NVIDIA Tesla P4 with 2560 cores), correspondingly. The reordering speedups of the four sparse matrices are 2.6×, 6.4×, 7.9×, and 8.6×, respectively, increasing significantly with the increase of sparseness (Table 7). However, it is not the key aspect of improving the reordering performance.

Table 7 Information of sparse matrices

Name	Dimension	NNZ	Sparseness
offshore	259 789	2 251 231	0.999 967
parabolic_fem	525 825	3 674 625	0.999 987
apache2	715 176	4 817 870	0.999 991
ecology2	999 999	4 995 991	0.999 995

NNZ: number of non-zero elements. The four different sparse matrices are selected from electromagnetics, computational fluid dynamics (CFD), structural, and 2D/3D applications, separately

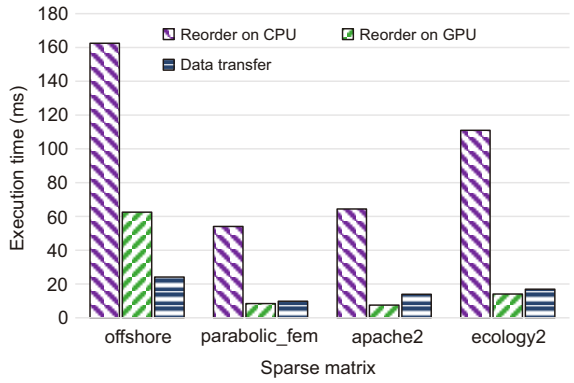

Fig. 6 Time comparison of data reordering on CPU and GPU

The additional overhead of transferring reordered data between the CPU and GPU also affects the reordering performance of the CPU-based reordering method. The overheads to transfer the four sparse matrices once are shown in Fig. 6. Only one data transfer is needed for GPU-based reordering for data initialization, but several transfers may be needed in the CPU-based method. Reordered data on the CPU must be transferred to the GPU for parallel computation, and after they are updated during the iterations, they will be transferred back to the CPU for the next reordering.

The CPU- and GPU-based data reordering methods are quite different in execution time, and their influence on different applications may also be different. This is because the influence is related to the number of data reorderings, the execution proportion compared with the GPU kernel, etc. For example, although the overhead gap between CPU- and GPU-based methods is in milliseconds to reorder and transfer data once, it can be omitted in a CG application but not in an MD application. In a CG program, the time of data reordering can be negligible because reordering is conducted only once for static data and occupies a lower proportion of time of the whole execution. On the contrary, the difference will become even larger, in seconds, with data reordering per iteration. The detailed analysis will be given in the next subsection.

6.3 Performance optimization with overlap and cache

CG, SP, and MD have the same memory access pattern, that is, indirect memory access. With the hyper-Q feature of the Kepler architecture, the GPU-based data reordering kernel and the most important computing kernel can be executed simultaneously on the GPU. This means that the reordering time can be partially or even completely hidden into the computation time. The execution time of the main computing kernel and the whole program is evaluated and analyzed, and the impact of CPU- or GPU-based data reordering on the whole execution is also analyzed.

6.3.1 Evaluation of the CG program

In most cases, the sparse matrix A in the CG program is stored in CSR format with three vectors: *row_ptr*, *col_ind*, and *val*. To determine the values of elements of each row in a matrix, *row_ptr* is first accessed, and *val* is indexed by the results of *row_ptr*. In the irregular kernel *SpMV*, each thread computes the multiplication $val[row\_ptr[tid]]\times vec[col\_ind[row\_ptr[tid]]]$ and stores the results in $dest[tid]$. With data reordering, the remapping kernel is first called to reorder *val* and *col_ind*, and this is then followed by invoking the new *SpMV* kernel. All data of *val* and *col_ind* are processed chunk by chunk. In the

new *SpMV* kernel, the access of *row_ptr* is removed, and each thread accesses a new vector directly. The *SpMV* kernel is called iteratively, but the data order is unchanged in each iteration. After caching, the remapping for the *SpMV* kernel must be executed only once. Therefore, there is no difference between data reordering on CPU or GPU. The reason is that the only performance difference between the two methods is the reordering time difference of tens of milliseconds, and such a small time difference is shielded and overlapped into the *SpMV* kernel execution with thousands of iterations.

A sparse matrix is the input data of the CG program (Table 7). The irregular kernel *SpMV* consumes a large fraction of the whole program. To verify the effectiveness of data reordering, the execution times of the *SpMV* kernel after data reordering and the original *SpMV* kernel in the CUSPARSE library are compared. The results are shown in Fig. 7a. We can see that the performance after data reordering is better than that of the kernel in CUSPARSE. Whether reordering on the CPU or GPU, the execution of the *SpMV* kernel is the same, because the data reordering kernel breaks away from the *SpMV* kernel and is executed asynchronously. The reordered kernel achieves a 9.64% performance improvement on average. This means that data reordering can remove the non-coalesced memory access to speed up the computing.

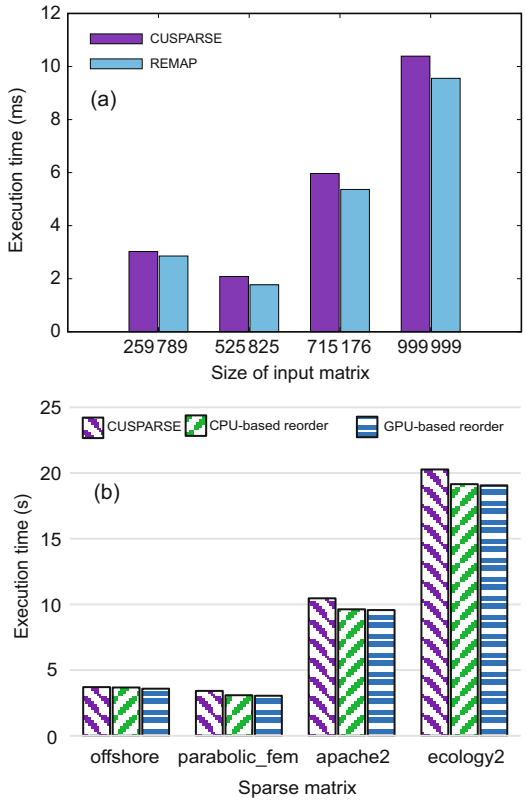

Fig. 7 Comparisons of the kernel and whole performance of the CG benchmark: (a) execution time of the *SpMV* kernel in each iteration; (b) execution time of the whole CG program

Reordering overhead is inevitable, including the reordering operation and data transferring, so the execution time of the whole program is tested (Fig. 7b), which includes the PCIe transfer and overhead of data reordering except for the *SpMV* kernel. Here the CPU- and GPU-based reordering methods are tested with their different overheads of data reordering. Compared with CUSPARSE execution, the whole execution time of the two reordering methods is reduced by 7.17% on average, while the difference between the two reordering methods is not visible to the naked eye. The reason is that the values of matrix elements are unchanged in each iteration, and thus the reordering operation is launched only once and its execution time is small (only tens of milliseconds) and can be omitted. Compared with the performance improvement of the *SpMV* kernel, the performance loss comes from the initial data loading with a small amount of non-overlapping reordering overhead.

6.3.2 Evaluation of the SP program

The SP program is designed to solve the Boolean SATisfiability (SAT) problem, which is a factor graph. The following structures are used to express the SAT problem: one *vars* graph to represent all variables, one *clauses* graph to represent all clauses, and one structure *edge* to represent all edges. The two graphs are both expressed in the form of an adjacency matrix. The structure *edge* consists mainly of two vectors, namely, *src* to store source nodes and *dst* to store destination nodes. The irregular kernel *calc_pi_values* has a two-level indirect memory access. Each thread first accesses with array *dst* (i.e., *edge.dst*[*tid*]) to obtain the destination node j and then accesses array *row_off* (i.e., *row_off*[j]) to obtain the start index v_j of *col_ind*. Finally, each thread accesses *col_ind* to find all nodes that point to node j (i.e., *col_ind*[v_j]). Correspondingly, two-level data reordering is applied to remove the non-coalesced memory access. The first-level remapping kernel is launched to reorder *row_off*

to *new_row_off*. The second-level remapping kernel uses the results of *new_row_off* as an input to reorder *col_ind* to *new_col_ind*. The SP algorithm updates each variable iteratively, and kernel *calc_pi_values* is called multiple times. By observing the number of launches of the remapping kernel, we find that the data are unchanged. Thus, the remapping kernel for *calc_pi_values* is called once as the caching.

The test cases for the SP program originate from the dataset of the benchmark suite LonestarGPU. One test case consists of 4000 variables and 16 800 clauses, and the other consists of 10 000 variables and 42 000 clauses. In the SP program, indirect memory access exists in multiple GPU cores, in which the *calc_pi_values* kernel is the most time-consuming so that only its execution time is analyzed because of the highly parallel performance with the hyper-Q feature. The performances of the *calc_pi_values* kernel and the whole program are examined and the results are demonstrated in Fig. 8. After removing the indirect memory access, the performance is better in the remapping kernel than in the original kernel for all test cases, and the REMAP method achieves a 17.1% performance improvement on average. As with the SP program, the performance improvement of the overall runtime has a slight impact due to limited data reordering overhead, which is reduced by 7.03% against the original program on average. Given the caching, the remapping kernel consumes only 0.1% of the execution time of the whole program, which is of great significance.

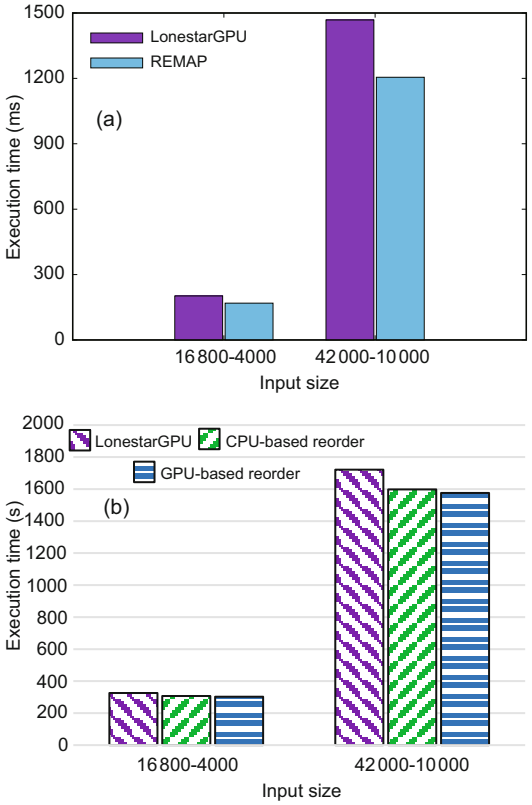

Fig. 8 Comparisons of the kernel and whole performance of the SP benchmark: (a) execution time of the *calc_pi_values* kernel in each iteration; (b) execution time of the whole SP program

6.3.3 Evaluation of the MD program

In the MD program, the information of all molecules is stored in array *pos*, and array *neighbors* stores the position of the adjacent molecules of a specific molecule. To determine the information of molecule i's neighbors, threads first access the array *neighbors* and use the results to index array *pos*. Given the movements of molecules, their neighborhoods are changed dynamically, and the indirect memory access (i.e., *pos*[*neighbors*[*tid*]]) causes the non-coalesced memory access of the kernel *compute_lj_force*. With data reordering, the data of *pos* are divided into multiple chunks, and the loop is moved outside the kernel. The data are reordered chunk by chunk and the new kernel processes one data chunk once. Benefiting from multiple CUDA streams, the remapping from the i^{th} chunk of *pos* to *new_pos* overlaps with the computation of the $(i-1)^{\text{th}}$ chunk of *new_pos*. In the experiments, the iteration number of the MD algorithm is set to be one such that the *compute_lj_force* kernel is executed only once.

Fig. 9 displays the execution time of the *compute_lj_force* kernel and the whole program. The test cases are randomly generated in the program. In Fig. 9a, the number of neighbors of each molecule is set to 128. For all inputs, the performance is better in the reordering kernel than in the original kernel, thus achieving 34.9% performance improvement. In Fig. 9b, the performance of the whole program also outperforms that of the original version, thereby achieving a 7.0% performance improvement. Considering that data are updated in each iteration, data reordering will be done per iteration. The overhead of data reordering is slightly high in that it consumes 23.7% of the whole execution time, although its execution time overlaps with the time of kernel computing.

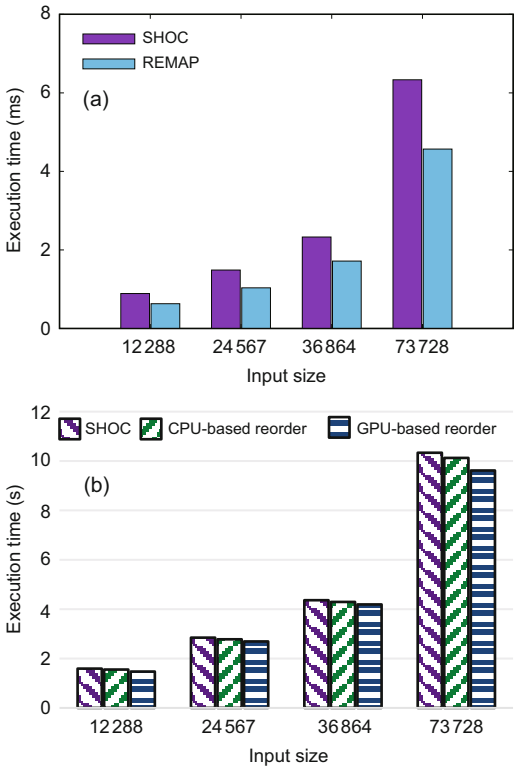

Fig. 9 Comparisons of the kernel and whole performance of the MD benchmark: (a) execution time of the *compute_lj_force* kernel in each iteration; (b) execution time of the whole MD program

With the increase of the number of computing iterations, the advantage of GPU-based reordering becomes more and more obvious compared with the CPU-based method. The runtime of GPU-based reordering is far less than that of CPU-based one, and with the help of hyper-Q, it can run simultaneously with the computing kernel, so as to minimize the overhead of data reordering. On the other hand, it can avoid the overhead of multiple data transferring between CPU and GPU to store them into the GPU cache. Fig. 9b shows that the whole execution time with GPU-based reordering is significantly lower than that of the CPU-based reordering method.

6.4 Overview of results

The performance improvement falls in three aspects: reducing the number of memory transactions, improving the execution time of irregular kernels, and enhancing the execution time of the whole application. For the CG, SP, and MD programs, the number of memory transactions is reduced; the reduction radios are between 1.2:1 and 2.0:1. Table 8 summarizes the speedup of our optimized kernel and the whole application against the implementation in the benchmarks. Overall, the reduced number of memory transactions and data reordering yield a speedup of the main computing kernel between 1.17× and 1.43× of the irregular kernel, thus confirming the effectiveness of the proposed solution in eliminating non-coalesced memory access. With effective optimizations to hide additional data reordering overhead, the performance of the whole application remains improved, achieving a speedup between 1.08× and 1.12×.

Table 8 Speedup of the optimized kernel and the whole program

Program	Kernel time (ms)		Speedup
	Base	Ours	
CG	2083	1771	1.17×
SP	1468	1205	1.22×
MD	1.488	1.036	1.43×
Program	**Application time (s)**		**Speedup**
	Base	Ours	
CG	3417	3045	1.12×
SP	1692	1554	1.09×
MD	2.913	2.697	1.08×

CG: conjugate gradient; SP: survey propagation; MD: molecular dynamics

The proposed method can solve irregular memory access caused by multiple indirect memory access similar to $A[B[tid]]$. For irregular applications of the non-$A[B[tid]]$ model, this method is not applicable. For irregular applications that need only to be processed once or a limited number of times, the optimization is not obvious. It should also be noted that data reordering occurs on the GPU, and it will lead to a lot of unnecessary transmissions for those irregular applications with instant data interaction between CPU and GPU, so that the optimization cannot be achieved.

7 Conclusions

The benefits of GPUs are minimal for irregular applications. To improve the performance of these irregular applications further, a pure software solution to remove the indirect memory access has been proposed. Data reordering has been used to create a desirable data layout and relocate data to

the new memory region. This has been offloaded to the GPU to improve the efficiency. After data reordering, index redirection has been used to redirect threads to access the new memory region. Reordered data have been cached for reuse in multiple iterations and other kernels. With hyper-Q, multiple CUDA streams have been used to overlap the overhead of reordering data with the execution time of the computing kernel. Experimental results showed that our solution can improve the performance of irregular applications, achieving a 7.0%–7.17% overall performance improvement on an NVIDIA Tesla P4 GPU.

A major extension has been planned for the future. Our solution focuses on non-coalesced memory access, while warp divergence also exists in several irregular applications. The extension aims to expand our solution to eliminate warp divergence. In recent years, an increasing number of GPUs have been equipped with one node, and data can be divided into multiple GPUs, thus posing a new challenge to eliminate the irregularity of kernels on all GPUs. Furthermore, some hardware extensions have been performed to partially solve the irregularity problem, and we can develop some software solutions based on the hardware extension to improve performance.

Contributors

Ran ZHENG and Yuan-dong LIU designed the research. Yuan-dong LIU processed the data. Ran ZHENG and Yuan-dong LIU drafted the manuscript. Hai JIN helped organize the manuscript. Ran ZHENG and Hai JIN revised and finalized the paper.

Compliance with ethics guidelines

Ran ZHENG, Yuan-dong LIU, and Hai JIN declare that they have no conflict of interest.

References

Alandoli M, Shehab M, Al-Ayyoub M, et al., 2016. Using GPUs to speed-up FCM-based community detection in social networks. Proc 7th Int Conf on Computer Science and Information Technology, p.1-6. https://doi.org/10.1109/CSIT.2016.7549467

Baskaran MM, Bordawekar R, 2008. Optimizing Sparse Matrix-Vector Multiplication on GPUs Using Compile-Time and Run-Time Strategies. IBM Research Report, RC24704 (W0812-047).

Baskaran MM, Bondhugula U, Krishnamoorthy S, et al., 2008. A compiler framework for optimization of affine loop nests for GPGPUs. Proc 22nd Annual Int Conf on Supercomputing, p.225-234. https://doi.org/10.1145/1375527.1375562

Bhatotia P, Rodrigues R, Verma A, 2012. Shredder: GPU-accelerated incremental storage and computation. Proc 10th USENIX Conf on File and Storage Technologies, p.14.

Burtscher M, Nasre R, Pingali K, 2012. A quantitative study of irregular programs on GPUs. Proc IEEE Int Symp on Workload Characterization, p.141-151. https://doi.org/10.1109/IISWC.2012.6402918

Danalis A, Marin G, McCurdy C, et al., 2010. The scalable heterogeneous computing (SHOC) benchmark suite. Proc 3rd Workshop on General-Purpose Computation on Graphics Processing Units, p.63-74. https://doi.org/10.1145/1735688.1735702

Davis TA, Hu YF, 2011. The university of Florida sparse matrix collection. *ACM Trans Math Softw*, 38(1):1. https://doi.org/10.1145/2049662.2049663

Fan WS, Wang B, Paul JC, et al., 2013. An octree-based proxy for collision detection in large-scale particle systems. *Sci China Inform Sci*, 56(1):1-10. https://doi.org/10.1007/s11432-012-4616-5

Fung WW, Sham I, Yuan G, et al., 2007. Dynamic warp formation and scheduling for efficient GPU control flow. Proc 40th Annual IEEE/ACM Int Symp on Microarchitecture, p.407-420. https://doi.org/10.1109/MICRO.2007.12

Lee S, Min S, Eigenmann R, 2009. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *ACM SIGPLAN Not*, 44(4):101-110. https://doi.org/10.1145/1594835.1504194

Li KL, Yang WD, Li KQ, 2015. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. *IEEE Trans Parall Distrib Syst*, 26(1):196-205. https://doi.org/10.1109/TPDS.2014.2308221

Li KL, Yang WD, Li KQ, 2016. A hybrid parallel solving algorithm on GPU for quasi-tridiagonal system of linear equations. *IEEE Trans Parall Distrib Syst*, 27(10):2795-2808. https://doi.org/10.1109/TPDS.2016.2516988

Meng JY, Tarjan D, Skadron K, 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *ACM SIGARCH Comput Arch News*, 38(3):235-246. https://doi.org/10.1145/1816038.1815992

Mokhtari R, Stumm M, 2014. BigKernel—high performance CPU-GPU communication pipelining for big data-style applications. Proc IEEE 28th Int Parallel and Distributed Processing Symp, p.819-828. https://doi.org/10.1109/IPDPS.2014.89

Pickering BP, Jackson CW, Scogland TRW, et al., 2015. Directive-based GPU programming for computational fluid dynamics. *Comput Fluids*, 114:242-253. https://doi.org/10.1016/j.compfluid.2015.03.008

Sabne A, Sakdhnagool P, Eigenmann R, 2013. Scaling large-data computations on multi-GPU accelerators. Proc 27th Int ACM Conf on Supercomputing, p.443-454. https://doi.org/10.1145/2464996.2465023

Song W, Wu D, Wong R, et al., 2015. A real-time interactive data mining and visualization system using parallel computing. Proc 10th Int Conf on Digital Information Management, p.10-13. https://doi.org/10.1109/ICDIM.2015.7381890

Sourouri M, Gillberg T, Baden SB, et al., 2014. Effective multi-GPU communication using multiple CUDA streams and threads. Proc 20th IEEE Int Conf on Parallel and Distributed Systems, p.981-986. https://doi.org/10.1109/PADSW.2014.7097919

Ueng SZ, Lathara M, Baghsorkhi SS, et al., 2008. CUDA-Lite: reducing GPU programming complexity. Proc 21st Int Workshop on Languages and Compilers for Parallel Computing, p.1-15. https://doi.org/10.1007/978-3-540-89740-8_1

Wang L, Spurzem R, Aarseth S, et al., 2015. NBODY6++GPU: ready for the gravitational million-body problem. *Mon Not R Astron Soc*, 450(4):4070-4080. https://doi.org/10.1093/mnras/stv817

Wang Y, Davidson A, Pan YC, et al., 2015. Gunrock: a high-performance graph processing library on the GPU. *ACM SIGPLAN Not*, 50(8):265-266. https://doi.org/10.1145/2858788.2688538

Wu B, Zhao ZJ, Zhang EZ, et al., 2013. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. *ACM SIGPLAN Not*, 48(8):57-68. https://doi.org/10.1145/2517327.2442523

Yang Y, Xiang P, Kong JF, et al., 2010. A GPGPU compiler for memory optimization and parallelism management. *ACM SIGPLAN Not*, 45(6):86-97. https://doi.org/10.1145/1809028.1806606

Zhang EZ, Jiang YL, Guo ZY, et al., 2011. On-the-fly elimination of dynamic irregularities for GPU computing. *ACM SIGARCH Comput Arch News*, 39(1):369-380. https://doi.org/10.1145/1961295.1950408

Ran ZHENG received her MS and PhD degrees from Huazhong University of Science and Technology (HUST), China in 2002 and 2006, respectively. She is currently an Associate Professor of Computer Science and Engineering at HUST. Her research interests include distributed computing, cloud computing, high-performance computing, and their applications.

Hai JIN is a Cheung Kung Scholars Chair Professor of Computer Science and Engineering at Huazhong University of Science and Technology (HUST) in China. He received his PhD degree in Computer Engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked at the University of Hong Kong between 1998 and 2000, and was a visiting scholar at the University of Southern California between 1999 and 2000. He was supported by the National Science Fund for Distinguished Young Scholars in 2001. He is the Chief Scientist of ChinaGrid, the largest grid computing project in China, and the Chief Scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is an editorial board member of *Frontiers of Information Technology & Electronic Engineering*. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.