Applied Mathematics & Information Sciences
An International Journal

Implementing Sparse Matrix-Vector Multiplication with QCSR on GPU

Jilin Zhang[1], *Enyi Liu*[1], *Jian Wan*[1], *Yongjian Ren*[1], *Miao Yue* [2] *and Jue Wang*[3]

[1] Department of Computer and Technology, Hangzhou Dianzi University, 310018, Hangzhou, Zhejiang, China
[2] Department of Architecture & Art, Zhejiang College of Construction, 311231, Hangzhou, Zhejiang, China
[3] Supercomputing Center of Computer Network Information Center, Chinese Academy of Sciences, Beijing, China

Received: Jul 8, 2012; Revised Oct. 4, 2012; Accepted Oct. 6, 2012
Published online: 1 Mar. 2013

Abstract: We are going through the computation from single core to multicore architecture in parallel programming. Graphics Processor Units (GPUs) have recently emerged as outstanding platforms for data parallel applications with regular data access patterns. However, it is still challenging to optimize computations with irregular data access patterns like sparse matrix-vector multiplication (SPMV). SPMV is one of the most important computational kernels in engineering practice and scientific computation. Various data formats to store the sparse matrix have been implemented on GPUs to maximize the performance. In this paper, we propose and evaluate a new implementation of SPMV on GPU based on $QCSR$ storage format which combines the $quadtree$ storage format and CSR format. We also outline some optimization strategies to improve performance. In comparison with previously published implementation, it achieves higher overall performance than $BCSR$ format. The results show that it achieves 1.15 speedup averagely than $BCSR$ format.

Keywords: GPU, Sparse Matrix-Vector Multiplication, QCSR, CUDA.

1. Introduction

Recently, we are witnessing the emergence of massive multicore architecture. There is no doubt that the multicore processors will compose the future supercomputers which will allow applications to support higher peak operation speed. GPU is one of powerful massively parallel systems and its highly parallel structure makes it more effective than CPU for algorithms which process large blocks of data in parallel [1]. GPUs are initially used to accelerate the memory-intensive work of texture mapping and rendering polygons for image processing. With the opening of its programmable interface and popularity of advanced language, GPU is extensively used for general computing because of its powerful parallel processing capabilities and high memory bandwidth. For example, NVIDIA GeForce GTX 285 peaks at 1063 GFLOPS in single precision and 159 GBytes/s memory bandwidth. Good results have been achieved in appropriate compute-intensive tasks like sorting [2], image processing, k-nearest neighbor search [3] and data mining [4].

The sparse matrix-vector multiplication is recognized as one of the most important numerical methods for science and engineering in the next decade [5] and is widely used in many scientific computing, such as Conjugate Gradient or GMRES, large linear systems and eigenvalues problems. For example, solving a partial differential equation using finite elements method boils down to solving a system of linear equation $Ax = b$, where A is sparse matrix. There are many zero elements in such matrices. It is inefficient as most calculations on zero elements are redundant and sometimes even impractical due to large dimensions of the matrix.

However, it is also well known that sparse matrix-vector multiplication yields only a small part of machine peak performance due to indirect and irregular memory accesses [6]. Optimization of sparse matrix-vector multiplication kernel has become increasingly significant and challenging for any architecture including GPU. Higher performance for sparse matrix-vector multiplication computation requires optimizations to best utilize the properties of the sparse matrix and system architecture. The storage for-

* Corresponding author: e-mail: wanjian@hdu.edu.cn

mat of sparse matrix is also very important in determining the performance. Traditional CSR storage format cannot take full use of the efficiency of GPU when the the nonzero values of a matrix. The transmission between CPU and GPU and storage cost are usually too much.

In this paper, we present a new storage structure for matrices called $QCSR$ storage format which which combines the $quadtree$ storage format and CSR format. We use recursive dividing method [7, 8, 9, 33] to generate $quadtree$ data structure, each leaf of which are consistent with the cache capacity. Using this structure can not only reduces the cache miss rate, but also efficiently improve the data locality during execution. Moreover, this structure makes it less nonsensitive to the distribution of nonzero elements. In this way, it will increase the efficiency in NVIDIA GPU. In order to achieve better performance, we outline some optimization strategies to maximize its performance according to the GPU architecture and its parallel programming model. Experiments show that it has an average of 1.15 speedup in sparse matrix-vector multiplication compared with traditional CSR format.

The rest of the paper is organized as follows. In section 2, related work on SPMV is described. In section 3, background on GPU architecture and parallel computing model is given. Details of our implementation about $QCSR$ storage format are present in section 4 and some optimizations are shown in section 5. Section 6 shows experimental results,while section 6 summarizes the conclusions and discusses future work.

$$A=\begin{pmatrix} 2 & 0 & 0 & 10.5 \\ 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 9 \\ 13 & 0 & 6 & 12 \end{pmatrix}$$

$$val=[2 \quad 10.5 \quad 5 \quad 7 \quad 9 \quad 13 \quad 6 \quad 12]$$

$$col=[0 \quad 3 \quad 1 \quad 2 \quad 3 \quad 0 \quad 2 \quad 3]$$

$$rowindex=[0 \quad 2 \quad 4 \quad 5 \quad 8]$$

Figure 1 CSR storage format representation of sparse matrix A. The nonzero values is stored in an array val. An array col stores the index of nonzero values and $rowindex$ stores the first value of each row in val.

```
// Basic SPMV implementation
// y = A*x, where A is in CSR.
  for(i = 0; i < m; i++)
{
    double y0 = y[i];
    for(k = rowindex[i]; k < rowindex[i+1]; k++)
      y0 += val[k]*x[col[k]];
    y[i] = y0;
}
```

Figure 2 A basic CSR-based SPMV implementation.

2. Related Work

Since sparse matrix-vector multiplication is the core problem in many applications, there are a large number of researches on this related subject.

Memory access is the bottleneck of sparse matrix-vector multiplication and is more prominent in the multicore architecture. In the CPU architecture, the primary mean is to load local data into caches or registers [10]. One of the GPU architecture characteristics is multi-level memory which is different from common architectures based on cache. Therefore, fine grained thread parallel is more suitable for GPU and different optimization strategies are designed according to the characteristics of the problem to take full use of its high memory bandwidth. Dehnavi, et al.,used Prefetch CSR($PCSR$) to accelerate finite-element SPMV kernels on GT8800 [27].

Sparse matrix formats are strongly involved in achieving high performance because they define the matrix data structure in memory. A variety of formats including Diagonal Format (DIA), ELLPACK Format (ELL), Coordinate Format (COO), Compressed Sparse Row Format (CSR), Hybrid Format (HYB) and Packet Format (PKT) have been evaluated on NVIDIA GPUs. Bell and Garland implemented several compressed storage formats and algorithms [11]. They took the memory bound nature of SPMV into consideration, and successfully utilized large percentages of peak bandwidth. Matrices in our research on the whole are sparse but there may be small dense sub-matrices. These dense sub-matrices help to improve data reuse. One of the most used formats in sparse matrix applications is CSR, as is illustrated in figure 1. This format compresses each row of matrix A and stores the nonzero values in an array val. An array col is created and stores each column index of data. A last array $rowindex$ stores the index of the element in array val, which is the first nonzero value of each row. The array $rowindex$ size is $row+1$ in which the number of nonzero values is stored at the last element. Diagonal format and ELLPACK format are shown in figure 3 and figure 4.

Some of the first work on sparse matrix-vector multiplication on GPU architectures was by Bolz et al [26]. They implemented conjugate gradient and multigrid solvers on GPU by using the graphics pipeline.

Volkov and Demmel presented an experimental study of GPU memory subsystem and an efficient implementation of dense matrix-matrix multiplication [12]. F. Vazquez and Demmel proposed ELLPACK-R format based on the ELLPACK storage format, compared with CRS, ELL, HYB format, it reduces the computation and data access. But when the maximum number of nonzeros per row does substantially differ from the average, threads for the ELLPACK-R suffer from load imbalance [13].

$$B = \begin{pmatrix} 2 & 11 & 0 & 0 & 0 \\ 13 & 5 & 14 & 0 & 0 \\ 0 & 6 & 10.5 & 13 & 0 \\ 0 & 0 & 4 & 7 & 3 \\ 0 & 0 & 0 & 8 & 9 \end{pmatrix}$$

$$data: \begin{pmatrix} * & 2 & * \\ 13 & 5 & 11 \\ 6 & 10.5 & 14 \\ 4 & 7 & 13 \\ 8 & 9 & 3 \end{pmatrix}$$

$$diag: \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

Figure 3 DIA storage format representation of sparse matrix. The diagonals are laid out as columns in a dense matrix structure ($data$), starting with the farthest sub-diagonal and ending with the largest super diagonal. An additional vector ($diag$) is kept which contains the offset of the diagonal represented by column in data from the central diagonal. This storage format does not require column offset or row pointer vectors like CSR, leading to a lower storage overhead. The $*$ elements in the data matrix represent unused elements that are needed to pad the diagonals into full columns. Unlike CSR, the DIA matrix layout is more efficient for diagonal matrices and allows contiguous memory access when reading matrix elements along diagonals [30].

$$data: \begin{pmatrix} 2 & 10.5 & * \\ 5 & 7 & * \\ 9 & * & * \\ 13 & 6 & 12 \end{pmatrix}$$

$$col: \begin{pmatrix} 0 & 3 & * \\ 1 & 2 & * \\ 3 & * & * \\ 0 & 2 & 3 \end{pmatrix}$$

Figure 4 $ELLPACK$ storage format representation of sparse matrix A. If K is the largest number of non-zero elements per row of an $N * M$ matrix,the $ELLPACK$ format stores the matrix as an $N * K$ dense matrix ($data$), along with a column index matrix (col) that stores the column index of each element. For rows that contain less than K non-zero elements, the matrices data and col are padded with unused elements. The $ELLPACK$ representation offers an efficient storage format if the maximum number of non-zero elements in all rows is significantly less than the number of columns in the sparse matrix [31].

Block Compressed Sparse Row ($BCSR$) format which is a variant of CSR was investigated by using register and cache blocking to reuse sparse matrix elements [14]. Both SPARSITY and OSKI adopt a heuristic algorithm to determine the optimal block size of sparse matrix to improve the performance of SPMV [15, 16]. Some optimizations were proposed to effectively develop a high-performance

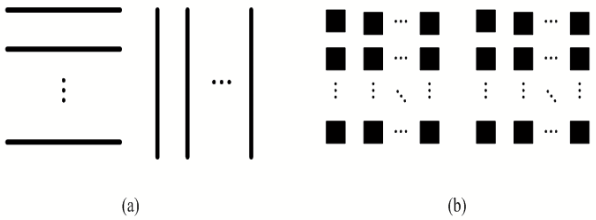


Figure 5 Two different ways of matrix reading, (a) is row or column oriented, (b) is block oriented.

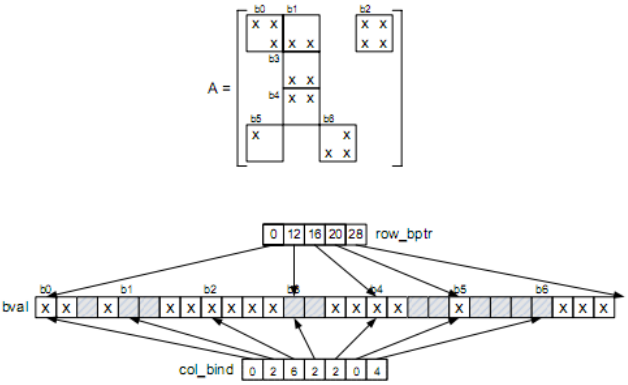


Figure 6 BCSR format, the shaded boxes are zero-pading. It stems from CSR format, but instead of storing and pointing to individual elements of the matrix [32].

```
for(i = 0; i < nr_block_rows; i++)
{
    double y0 = y1 = 0;
    for(k = row_bptr[i], j = k / (2*2), l = col_bind[j];
        k < row_bptr[i+1]; k += 2*2, l = col_bind[++j])
    {
        y0 += bval[k] * x[l] + bval[k+1] * x[l+1];
        y1 += bval[k+2] * x[l] + bval[k+3] * x[l+1];
    }
    y[2*i] = y0;
    y[2*i+1] = y1;
}
```

Figure 7 The standard BCSR SPMV implementation for a 2*2 block.

SPMV kernel on NVIDIA GPUs: synchronization-free parallelism, thread mapping based on the affinity towards optimal memory access pattern, optimized off-chip memory access to tolerate the high access latency, and exploiting data reuse [17]. Choi et al proposed an auto-tuning framework for sparse matrix-vector kernels using blocked CSR and blocked $ELLPACK$ format rearranges the rows of the sparse matrix in decreasing order of the number of non-zero elements [22]. The rows are then separated into blocks, where each block is stored in the ELLPACK format.

3. GPU and Programming Model

In this section, we provide background information on GPU architecture and programming model called Compute Unified Device Architecture (CUDA) for NVIDIA GPU.

In these years, GPUs are usually combined with CPUs to constitute heterogeneous system for supercomputers. A large quantity of researches have been investigated on this architecture to achieve speedup in compute-intensive area like medical imaging,molecular dynamics and financial simulation. GPUs are massively-threaded, many-core architecture with high computing power and memory bandwidth and compared with CPUs, furthermore, they can also promote energy efficiency. But they are not comprehensive like CPUs and are inferior in some processing methods especially in branch prediction. At present, GPUs will not take place of the CPUs. It is much worthwhile to take advantage of the CPUs and GPUs to accelerate existing computing-intensive tasks.

The GPU parallel computing architecture mainly consists of two important components,processor units and memory hierarchy [28].

GPU consists of multiprocessor units called streaming multiprocessors (SMs), each one of which contains a set of processor cores called streaming processors (SPs). As is shown in figure 8. There are various memories available including global memory, local memory, shared memory, constant memory, texture memory and registers on GPU. Computations are launched in a parallel manner, with all threads executing the same function called a kernel and the threads are partitioned into blocks. Threads within a block have access to shared memory. Threads within a block can also be synchronized by using barriers [18]. Figure 9 depicts the overall flow. The task partition will greatly affect the implementation performance. It is well-advised to choose the number of blocks and threads of each block based on the task characteristics and GPU itself hardware characteristics.

CUDA is general parallel programming model to be used on NVIDA GPUs. As illustrated in figure 10, CUDA programming model regard CPU as host and GPU as device. Parallel computation function on GPU is called kernel which is executed on a set of threads. Firstly data will be copied from CPU to GPU and then from GPU to CPU after the kernel is over. CUDA model allows programmers

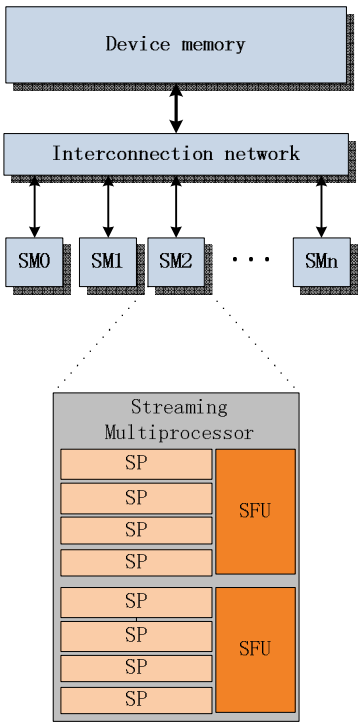


Figure 8 SM structure graph. Streaming multiprocessor consists of streaming processors (SP) and other resources. SP is basic processing unit. The number of SPs and SMs on different GPUs is not the same.

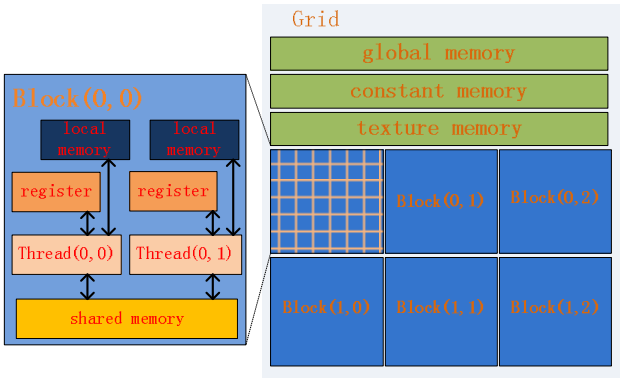


Figure 9 Memory model. Parallel computation function on GPU is called kernel and will be executed on a set of threads in the form of blocks to compose the grid. Each thread has access to global memory but the access latency is big. It also has a read-only constant memory and texture memory that are shared by all the threads. Only the threads in the same block can be synchronized and own the shared memory. The shared memory access speed is almost equal to the register memory access speed,which is the least access latency.

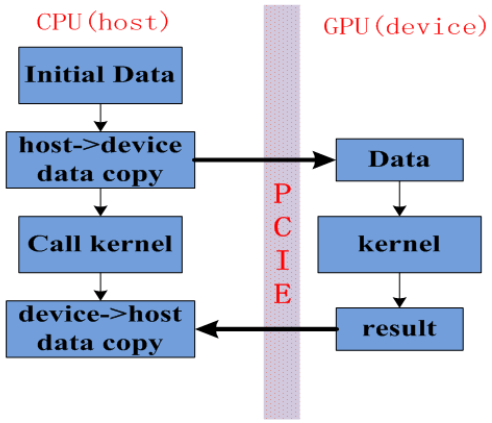


Figure 10 The general process of the GPU programming. CPU is taken as host and GPU is device. Parallel computation function called kernel is to be executed on GPU.

to better exploit the parallel power of the GPU for general purpose computing.

When programming kernels on GPU, we should take measures to optimize the kernels because there are several inefficiencies that are somewhat unique to GPU architectures. Although GPU devices provide very high memory bandwidth, it is only achieved with coalesced access. Memory requests are serviced for halves of a warp (32 threads in a block) at a time. To achieve highest memory throughput, memory access pattern must follow coalescing rules: accessed addresses must fit into a 64 or 128 byte window which must be aligned. To fully exploit the massive computing resources of the GPUs, the memory latency needs to be efficiently hidden. Additionally, the shared memory is a banked memory architecture. If concurrently executing threads in a block make request to shared memory locations in the same bank, a bank conflict will occur and the requests will be serialized. Therefore, to achieve efficient access to shared memory ,concurrently executing threads should access memory belonging to different banks. Finally, all concurrently executing threads in a block must issue the same instruction to the streaming processors to avoid divergent control flow [25].

4. QCSR Storage Format

4.1. The implementation of $QCSR$ storage format

The performance of sparse matrix-vector multiplication depends strongly on the used matrix storage format. Sparse matrices often contain dense sub-matrices, so various blocking formats were designed to accelerate matrix operations [19]. Compared to the CSR format, the aim of these formats is to consume less memory. Storing a matrix as a set of small dense blocks can significantly improve the performance. Algorithms for the multiplication of such blocks can be fine tuned for a specific architecture. But it still suffers from large transformation overhead.

We introduce a new sparse matrix storage format called $QCSR$ storage format which combines $quadtree$ storage format and CSR. To improve generation process of quadtree, traditional tree generation algorithms (such as depth-first search and breadth-first search) are not suitable here, because of large numbers of traversal, search and stack overheads in such algorithms [29].

Using $quadtree$ storage format, a matrix is divide into four same-size sub-regions from top to bottom, and each sub-region is divided into four regions recursively. The decomposition continues until the length of sub-region reaches a given limit d. Figure 11 details the process of recursively decomposing a sparse matrix. The sparse matrix can be recursively decomposed into a number of sub-regions. Obviously, in order to improve the cache data locality, the best case is that data loaded into the cache can complete all of its operations. It can not be fully achieved, but we can adjust the region length d to approximate this ideal. Therefore, an appropriate selection d can help to decompose the matrix into independent operations of various sub-regions, which is not bigger than the cache capacity. This process can be expressed as the recursive generation of a quadtree, as is shown in figure 12. There are two kinds of nodes in this quadtree. One is called empty-region (expressed as E), which doesn't contain nonzero elements. The other is called mixed-region (expressed as M), which contains nonzero elements and can be decomposed continually. In this way, quadtree stores all useful information in its leaf nodes and the nodes can be indexed without traversing all values of matrix while processing the matrix multiplication[24]. Algorithm 1 shows the quadtree generation algorithm. In the worst case, the total number of intermediate nodes is $(4^{log_2(n/d)}-1)/3$ and the decomposition complexity after the transformation is $O(2n/d)$. Obviously, the time complexity of transformation is much less than that of multiplication. When storing a leaf node, up to $d*d$ elements need to be located. These elements can be located by binary search since the elements in each sub-region are stored orderly.

Although the total complexity of this algorithm is higher than time complexity of a single matrix-vector multiplication, considering the matrix-vector multiplication usually takes more than thousands of iterations in practical applications, the performance of quadtree based algorithm is better than traditional multiplication algorithm due to amortization. Table 1 shows the performance ratio between transformation and matrix-vector multiplication. As it shows, the cost of transformation has a direct ratio with the dimension size, the larger dimension the matrix has, the larger cost the transformation uses, and the larger region size the transformation algorithm uses, the less time cost it will be.

This quadtree storage structure has these following characteristics:

Table 1 The performance ratio between transformation and SPMV

Region size	715176 × 715176	34920 × 34920
256	251.128	144.092
1024	210.342	112.227
8192	160.653	61.301
32768	120.84	23.1371
131072	75.345	6.793

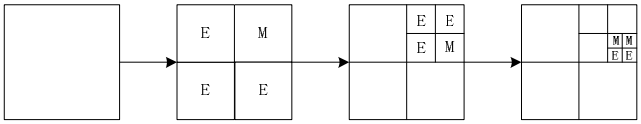


Figure 11 A sparse matrix in the quadtree storage format. Empty nodes with zero elements are marked E. Mix nodes with zero elements and nonzero elements are marked M.

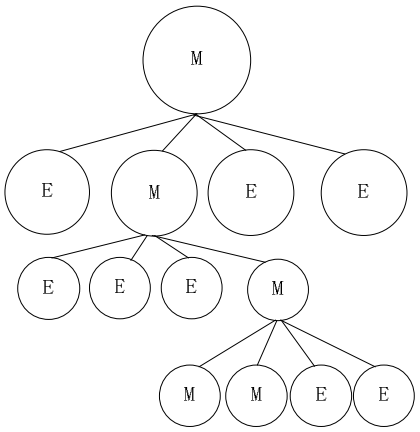


Figure 12 A schematic diagram of matrix division process into a quadtree for matrix storage format.

(1) If the size of matrix is $n \times n$ and the size of the target region is $d \times d$, the maximum depth of the tree is $Dep = log_d n$ and the maximum number of intermediate nodes is $N = O(4^{Dep-1})$;

(2) The overhead of storing block information and intermediate node. During the process of generating the quadtree structure, each intermediate node contains an index pointer (x, y) to a relate region (occupy a storage space S_I) and the region length d (occupy a storage space S_D). The maximum additional space overhead of quadtree is $C_S = (2S_I + S_D) \times O(4^{Dep-1})$. However, this part of overhead is in the process of transformation, organization and representation of the matrix, not in the multiplication process, and it does not increase the time complexity of computation.

(3) Lower compression ratio. Compared to the traditional CSR storage format, the $QCSR$ storage format requires less total memory to store the sparse matrix. This is particular important on GPU with limited memory. Meanwhile, lower sparsity representation also enables us to eliminate the useless computation that results from the zero values.

(4) Elimination of the possible impact of the distribution of nonzero elements in the multiplication. The original multiplication has been divided into independent operations for sub-regions. Most of these sub-regions are dense form, based on which the algorithm is more general.

(5) Easy to programming. This data structure reflects the process of recursive decomposition. It is suitable for the block matrix multiplication algorithm.

4.2. The effects of matrix feature for efficiency

The number of nonzero elements which is determined by the size and sparsity of a matrix, and the distribution of nonzero elements define the space complexity. For $n \times n$ matrices under the same distribution, a higher sparsity matrix can get a smaller amount of target regions, therefore, it contributes to less storage overhead and better performance. When the sparsity of a matrix decreases within a certain range, the number of target regions may not increase largely, which means the additional overhead will not increase significantly. Due to the preservation of data locality, the greater the density of a target region is, the better performance the program has. When the sparsity continues to decrease, the number of target regions will increase apparently, as well as the additional space overhead. Since target regions are independent within each other, the overall program performance will not decline significantly. When the size and sparsity of a matrix have been determined, the distribution of nonzero elements only affects the generation speed of a quadtree and the processing storage space, but has little influence on the efficiency of multiplication processing. Therefore, matrix multiplication algorithm based on quadtree storage format could works efficiently for all general cases.

5. Optimization Strategies

Although modern processors integrated with many processing cores enable the high performance of programs, it highlights the memory bottleneck, especially for SPMV in which data loaded from the memory mainly are computed only once. Moreover, multilevel memory system and access and size limitation make it much more difficult for the Single Instruction Multiply Thread (SIMT) architecture of GPU. In this paper, we investigate some optimizations on thread mappingdata reuse, data access and data transform to achieve better performance [20, 21].

Input:
pseudo code.
matrix A in CSR for the transformation;
$n0$=the number of nonzero elements in matrix A;
$n1$=the order of matrix A;
Output:
the pointer for the root of the quadtree;

```
1: if(n0==0)
2: {
3: return NULL;
4: }
5:
6: if(n1 > tile_size)
7: {
8: create M, the leaf of type "Mixed";
9: M is the parent node of M1, M2, M3, M4;
10: divide matrix M into four submatrix A1, A2, A3, A4;
11: M1=Transf(A1);
12: M2=Transf(A2);
13: M3=Transf(A3);
14: M4=Transf(A4);
15: return M;
16: }
17:
18: else
19: {
20: density=n0/(n1 * n1)
21: if (density > fill_in_ratio)
22: {
23: transform the matrix A to the leaf F of type "Full";
24: return F;
25: }
26: else
27: {
28: transform the matrix A to the leaf S of type "Sparse";
29: return S;
30: }
31: }
```

Algorithm 1
Trans(startRow, endRow, startColumn, endColumn)

Input:
The first row of sub-matrix, startRow;
The last row of sub-matrix, endRow;
The first column of sub-matrix, startColumn;
The last column of sub-matrix, endColumn;
The size of target region, d;
Output:
The quadtree structure;

```
midRow = (endRow - startRow) / 2;
2: midColumn = (endColumn - startColumn) / 2;
detRow = endRow - startRow;
4: detColumn = endColumn - startColumn;
if(detRow > d and detColumn > d)
6: {
Trans(startRow, midRow, startColumn, midColumn);
8: Trans(midRow, endRow, startColumn, midColumn);
Trans(startRow, midRow, midColumn, endColumn);
10: Trans(midRow, endRow, midColumn, endColumn);
}
12:
else if(detColumn > d)
14: {
Trans(startRow, endRow, startColumn, midColumn)
16: Trans(startRow, endRow, midColumn, endColumn)
}
18:
else if(detRow > d)
20: {
Trans(startRow, midRow, startColumn, endColumn)
22: Trans(midRow, endRow, startColumn, endColumn)
}
24:
else
26: {
TREE.SAVE(startRow, endRow, startColumn, endColumn)
28: }
```

5.1. Thread mapping

The most natural idea is that each element of the result vector is calculated by a thread. Each thread is responsible for each row of the input sparse matrix and vector multiplication. However, this way is more suitable for relatively small processor cores and the threads scheduling independently in parallel computing platform but not for GPU. In GPU architecture, thread blocks are divided into warps in SM and the threads in the same warp are naturally synchronized. If the number of nonzero differs quite for each thread in the same warp, it would cause conditional branch which will not take full use of computing resources of many threads to hide the access latency of global memory. According to the characteristic that threads in the block can be parallelized in GPU, each block is responsible for each row is a suitable method. But, each block also possess many threads, when the number of nonzero elements in a row cannot be exactly divided by the total number of threads in the block, there will be high probabilities of resulting in a number of threads have nothing to do and thread synchronization overhead.

Based on the above strategy and considering the coalescing access, we calculate a row of sparse matrix and vector multiplication in half-warp to further reduce the number of threads for each row. Obviously, the defects will arise again when the number of nonzero elements in adjacent rows differs largely. Although the thread idling cannot be absolutely avoided, it can reduce the synchronization waiting in warp.

5.2. Data Access

The starting nonzero of a row is always in non-aligned position in the value array that stores the nonzero of the sparse matrix. If the alignment is not adjusted, it might result in the entire row being accessed in a non-optimal man-

ner and eventually increasing memory access cost. So, the starting address alignment and coalescing access are the keys to optimize GPU data access. Coalescing access is that threads in half-warp visit altogether memory and adjacent threads access the adjacent data. Therefore, data is stored by row priority to meet the requirement of coalescing access. Therefore, we pick up but not compute the surplus part of a row to next row and compute each row in the number of a multiple of 16. At the last, zeros are padded to ensure this optimization.

5.3. Data Reuse

In GPU architecture, the global memory access latency is very high. In this paper, data will be loaded into shared memory while the input vector will be put into the texture memory to reduce the high latency of global memory.

5.4. Data Transform

GPU offers high memory bandwidth, but maximizing the achievable bandwidth is usually a hard problem. Since the sparse matrix-vector multiplication kernels have little reuse across the data arrays, it is significant to optimize the kernel to maximize the bandwidth while reading matrix elements. The transformation between CPU and GPU is overhead for general computing based on GPU. CUDA enables us to allocate and use zero-copy memory to reduce the data transform time. But when the matrix is too large to put into this memory, the performance will not be so excellent. So we should selectively use this paged-lock transform for different matrices.

6. Experiment Result

Our experiments are carried out on Intel i5-2400 CPU@3.1 GHz with 4GB memory and G100. We choose CUDA 4.0 in Windows 7 with 64bit as compiler. The experimental data are from realistic scientific computing and engineering fields. The data can be obtained from the University of Florida sparse matrix collection [23]. The properties of these matrices are shown in table 2. Using the above optimization strategies and $QCSR$ storage format, SPMV can get better performance compared with $BCSR$ storage format, as illustrated in figure 13. The results show that averagely 1.15 speedup can be achieved. But we also can see that the speedup is affected by the matrix structures. Matrix $2D_54019_highK$ possess high dimension and much more nonzero than matrix $add32$, but high speedup is achieved because it has obvious appropriate matrix structure for $QCSR$. Comparing these matrix structures, we can see higher speedup can be achieved for a matrix consisted of more blocks marked E. Generally, good results can be achieved compared to $BCSR$.

Table 2 Matrix benchmark suites which are selected from different matrix structure.

matrix	id	dimension	nonzeros
/2D$_5$4019$_h$$ighK$	1	54,019	486,129
Hamm/add32	2	4,960	19,848
QY/case9	3	14,454	147,972
GHS$_i$$ndef/aug3d$	4	24,300	69,984
Hollinger/g7jac060	5	17,730	183,325
TKK/g3rmt3m3	6	5,357	207,695
Pajek/foldoc	7	13,356	120,238

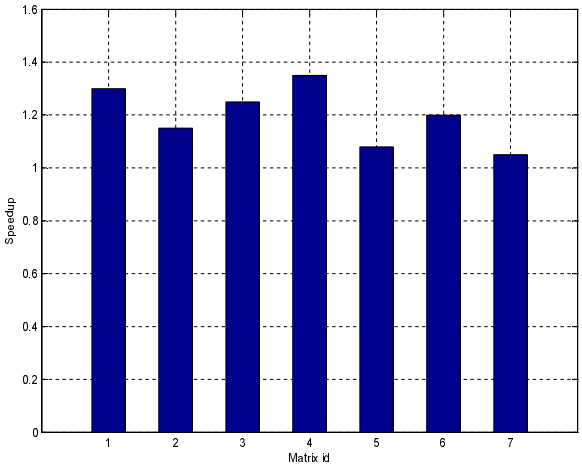


Figure 13 Speedup of QCSR to BCSR. Averagely 1.15 speedup can be achieved.

7. Conclusion and Future Work

In this paper, we propose a new storage format called $QCSR$ based on $quadtree$ storage format and CSR and discuss some optimization strategies of sparse matrix-vector multiplication on NVIDIA GPU using CUDA programming model. Although outstanding performance improvements are obtained over $BCSR$ storage format, the best case is to load data into the cache to complete all of its operations which can significantly improve the cache data locality. Therefore, an appropriate selection of target region length d can help to decompose the matrix into independent operations of various sub-regions. Moreover, compared with other implementations, high speedup for $QCSR$ is achieved for specialized matrices with appropriate distribution of nonzero elements. It is also affected by selection d. Therefore the next step for tuning d is very important.

Acknowledgement

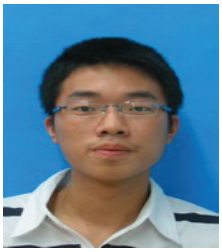
This paper is supported by State Key Development Program of Basic Research of China under Grant No.2007CB-310900, the Hi-Tech Research and Development Program

(863) of China under Grant No.2011AA01A205, Natural Science Fund of China under Grant No.61202094, No.6100-3077, No.60873023, No.60973029, the key grant project of Chinese Ministry of Education about ChinaGrid, the science and technology major project of Zhejiang Province (Grant No.2011-C11038), Zhejiang Provincial Natural Science Foundation under grant No.Y1101104, Y1101092, Y1090940. Zhejiang Provincial Education Department Scientific Research Project (No. Y2-01016492), Key Projects in the National Science & Technology Pillar Program (No. 2012BAH24B04).

References

[1] T. Brandvik, G. Pullan, Journal of Mechanical Engineering Science **221**, 1745 (2007).
[2] Sintorn Erik, Assarsson Ulf, Journal of Parallel and Distributed Computing **68**, 1381 (2008).
[3] Garcia Vincent, Debreuve Eric, Barlaud Michel, CVPR Workshop on Computer Vision on GPU, 1 (2008).
[4] Wenbin Fang, Lau Ka Keung, Mian Lu, Technical Report HKUST-CS08-07, Hong Kong University of Science and Technology, (2008).
[5] K. Asanovic, R. Bodik, B. C. Catanzaro, et al, Technical Report No. UCB/EECS-2006-183, (University of California at Berkeley, 2006).
[6] R. Vuduc, J. W. Demmel, K. A. Yelick, Journal of Physics: Conference Series, (SanFrancisco, 2005).
[7] I. Simecek, Proceedings of 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 168 (2010).
[8] Matam, Kiran Kumar, Kothapalli, Kishore, International Conference on Parallel Processing, 612 (2011).
[9] M. Michele, F. Salvatore, T. Salvatore, G. Pawel, Paprzycki, Marcin, Proceedings of the International Multiconference on Computer Science and Information Technology, 327 (2010).
[10] Richard Vuduc, Automatic Performance Tuning of Sparse Matrix Kernels, Ph.D.thesis, (2003).
[11] N. Bell, M. Garland, SC 2009: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 1 (New York, 2009).
[12] V. Volkov, J.W. Demmel, SC 2008: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 1 (2008).
[13] F. Vazquez, E. M. Garzon, J. A. Martnez, J. J. Fernandez, Techinical Report, (2009).
[14] Alexander Monakov, Arutyun Avetisyan, LNCS **5657**, 289 (2009).
[15] E. J. Im, K. A. Yelick, LNCS **2073**, 127 (2001).
[16] Optimized Sparse Kernel Interfac. http://bebop.cs.berkeley.edu/oski/.
[17] M. M. BASKARAN, R. BORDA WEKAR, IBM Research Report RC24704, (2009).
[18] NVIDIA Corporation: NVIDIA CUDA Programming Guide 4.2 (2012).
[19] V. Karakasis, G. Goumas, N. Koziris, Proceedings of the 2009 International Conference on Parallel Processing, 356 (2009).
[20] Feng Xiaowen, Jin Hai, Zheng Ran, Hu Kan, Zeng Jingxiang, Shao Zhiyuan, Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS, 165 (2011).
[21] Pichel, C. Juan, F. Francisco, Fernndez, Marcos, Rodrłguez, Aurelio, Microprocessors and Microsystems 36, **65** (2012).
[22] J. W. Choi, A. Singh, R. W. Vuduc, Proceedings of the 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 115 (Bangalore, 2010).
[23] T. Davis, The University of Florida sparse matrix collection [EB/OL], http://www.cise.ufl.edu/research/sparse/matrices, (1997).
[24] Zhang Ji Lin, Wan Jian, Xu Xiang Hua, Jiang Cong Feng, Ren Yong Jian, Proceedings 2011 6th Annual ChinaGrid Conference, 124 (DaLian, 2011).
[25] Jeswin Godwin, Justin Holewinski, P.Sadayappan, Proceedings of the 5th Annual Workshop on General Purpose Proceeding with Graphics Processing Units, 47 (New York, 2012).
[26] J. Bolz, I. Farmer , E. Grinspun, Schrooder, ACM SIGGRAPH 2003, 917 (New York, 2003).
[27] D. M. Mehri, F. M. David, I. Farmer , G. Dennis, IEEE Transactions Magnetics **46**, 2982 (2003).
[28] Feng Xiaowen, Jin Hai, Zheng Ran, Hu Kan, Zeng Jingxiang, Shao Zhiyuan, Proceedings of the International Conference on Parallel and Distributed Systems, 165 (2011).
[29] Guy E. Blelloch, Ioannis Koutis, Gary L. Miller, Kanat Tangwongsan, SuperComputing, SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 1 (2010).
[30] Saad, Y. SPARSKIT: A basic toolkit for sparse matrix computations - version 2.
[31] Grimes, R., Kincaid, D., Young, D. ITPACK 2.0 user's guide, August 1979.
[32] K. Vasileious, G. Georgios, K. Nectarios, Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium, (2009).
[33] Simecek Ivan, SYNASC 2009-11th International Symposium on symbolic and Numeric Algorithms for Scientific Computing, 168 (2009).

Jilin Zhang received the PhD degree in Computer Application Technology from University of Science Technology Beijing, Beijing, China, in 2009. He is currently a lecture in software engineering in Hangzhou Dianzi University, China. His research interests include High Performance Computing and Cloud Computing.



Enyi Liu received B.S. from College of Automatic Control Engineering and Information Technology, Beihang University. He is now M.S. in Computer Science and Technology in Hangzhou Dianzi University. His research interests include High Performance Computing, Cloud Computing, and Pattern Recognition.

Jian Wan eceived the PhD degree in Computer Application Technology from Zhejiang University, Zhejiang, China, in 1989. He is currently a professor in software engineering in Hangzhou Dianzi University, China. His research interests include Grid Computing, Service Computing and Cloud Computing.

Yongjian Ren received the PhD degree in Computing Application Technology from Florida Atlantic University, in 1998. He is currently a professor in software engineering in Hangzhou Dianzi University, China. His research interests include Cloud Computing and Mass Storage.

Miao Yue received the Master degree in Architecture Technology and Science from Zhejiang University, Hangzhou, China, in 2012. She is currently working as an assistant lecture in architecture in Zhejiang College of Construction, China. Her research interests include Architecture Digitized and GPU render computing.

Jue Wang is currently working as a associate professor in the supercomputing center of Chinese Academy of Science. The motivation behind his work is to improve soft systems by increasing the productivity of programmers and by increasing software performance on modern architectures including many cores clusters and GPU.