CRSD: Application Specific Auto-tuning of SpMV for Diagonal Sparse Matrices*

Xiangzheng Sun, Yunquan Zhang, Ting Wang, Guoping Long, Xianyi Zhang, and Yan Li

Lab. of Parallel Software and Computational Science,
Institute of Software, Chinese Academy of Sciences.
Graduate University of Chinese Academy of Sciences
{xiangzheng08,yunquan,wangting,guoping,xianyi,liyan08}@iscas.ac.cn

Abstract. Sparse Matrix-Vector multiplication (SpMV) is an important computational kernel in scientific applications. Its performance highly depends on the nonzero distribution of sparse matrices. In this paper, we propose a new storage format for diagonal sparse matrices, defined as Compressed Row Segment with Diagonal-pattern (CRSD). We design diagonal patterns to represent the diagonal distribution. As the diagonal distributions are similar within matrices from one application, some diagonal patterns remain unchanged. First, we sample one matrix to obtain the unchanged diagonal patterns. Next, the optimal SpMV codelets are generated automatically for those diagonal patterns. Finally, we combine the generated codelets as the optimal SpMV implementation. In addition, the information collected during auto-tuning process is also utilized for parallel implementation to achieve load-balance. Experimental results demonstrate that the speedup reaches up to 2.37 (1.70 on average) in comparison with DIA and 4.60 (2.10 on average) in comparison with CSR under the same number of threads on two mainstream multi-core platforms.

Keywords: CRSD, Auto-tuning, SpMV, Diagonal-pattern, Application Specific Optimization.

1 Introduction

The Sparse Matrix-Vector multiplication (SpMV) is one of the most important computational kernels in sparse linear algebra. Algorithms based on Compressed Sparse Row(CSR) format often perform poorly on modern computer systems. The performance highly depends on nonzero distribution, which determines the memory access pattern and varies significantly among different applications.

In this paper, we study the optimization for diagonal sparse matrices, in which the nonzeros mainly distribute along diagonals. Diagonal sparse matrices are

* This paper is supported by the National 863 Plan of China (No.2006AA01A125, No. 2009AA01A129, No. 2009AA01A134), the China HGJ Significant Project (No. 2009ZX01036-001-002), the Knowledge Innovation Program of the Chinese Academy of Sciences (No.KGCX1-YW-13), the Ministry of Finance (No. ZDYZ2008-2).

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 316–327, 2011.
© Springer-Verlag Berlin Heidelberg 2011

universal. As far as we know, the Finite Difference Method(FDM) is widely used to solve the numerical problems. Once the FDM is used, the coefficient matrix of discrete Partial Differential Equations(PDEs) is usually the diagonal sparse matrix. The numerical solution to the PDEs is an approximation to its exact solution by using a discrete representation to the PDEs on the $m \times n \times l$ mesh points (x_i, y_j, z_k), where $1 \leq i \leq m, 1 \leq j \leq n, 1 \leq k \leq l$. In the finite difference scheme, the unknown's value $U_{i,j,k} = U(x_i, y_j, z_k)$ is related to $U_{i\pm t, j\pm p, k\pm q}$ (where t, p, q may normally be 1 or 2). As long as the difference scheme is fixed, t, p and q remain unchanged. When we modify m, n and l to change the problem size, the diagonal distribution remains similar.

The Diagonal format(DIA) [1] is designed to store the diagonal sparse matrix. All nonzeros on the same diagonal share the same index. However, a large number of zeros should be filled when there are many scatter points or the diagonal is broken by a long zero section. We define the long zero section as ***idle section***.

To address this problem, we propose a novel storage format CRSD. In order to represent the diagonal distribution, we design the diagonal pattern, which divides diagonals into different groups. Furthermore, the matrix is split into row segments. In each row segment, nonzeros on the diagonals of the same group are viewed as the unit of storage and operation. We store those nonzeros contiguously and organize the operation on them in one loop. Simultaneously, the scatter points are also detected in each row segment. The number of filled zero for idle section can be controlled according to the application.

Because the diagonal distribution remains similar in one application of different problem sizes, most diagonal patterns remain unchanged. We define the unchanged diagonal pattern as application specific diagonal pattern(detailed in section 2.2). For any given application, we analyze one matrix, named the sample matrix, to obtain application specific diagonal patterns. Next, the optimal SpMV codelets for those diagonal patterns are generated automatically. Finally, we combine those codelets as the optimal SpMV implementation. In addition, the information collected during auto-tuning process can also be utilized for parallel implementation to achieve load-balance. As the unchanged diagonal patterns vary across diverse applications, the optimization is application specific.

The rest of this paper is organized as follows: section 2 describes the diagonal pattern and CRSD storage format; section 3 presents the process of automatic performance tuning; in section 4, the experiment results are provided and analyzed; the related works are given in section 5. At last, conclusion is summarized in section 6.

2 CRSD Storage Format

2.1 Diagonal Pattern

For any two diagonals in the matrix, if the absolute value of difference of their offset [1] is 1, they are adjacent. We can group a sequence of diagonals by the following steps: if two diagonals are adjacent, put them into an ***adjacent (AD) group***; after removing the diagonals within the adjacent groups, the original

diagonal sequence is broken up into pieces. We assign the diagonals of each piece into a ***nonadjacent(NAD) group***. The ***diagonal pattern*** is defined as the way that the AD group(s) and the NAD group(s) are organized.

When the group is represented by group type(AD or NAD) and the number of diagonals in it, then

$$\text{group} = (\text{group type, the number of diagonals})$$

According to the definition, the diagonal pattern is represented as follows:

$$\text{diagonal-pattern} = \{\text{group}_1, \text{group}_2, \ldots \text{group}_m\}$$

If the whole matrix contains several diagonal patterns, then

$$\text{matrix} = \{\text{dia-pattern}_1, \text{dia-pattern}_2, \ldots \text{dia-pattern}_n\}.$$

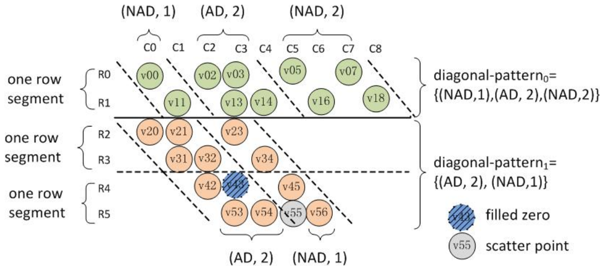


Fig. 1. Example of diagonal sparse matrix

For example, there are two diagonal patterns in the matrix shown in Fig. 1 except nonzero v55. The matrix is represented as follows:

$$\text{matrix} = \{\{(\text{NAD},1),(\text{AD},2),(\text{NAD},2)\}, \{(\text{AD},2), (\text{NAD},1)\}\}.$$

With diagonal pattern, we can process idle section: if there are few zeros in the idle section, we can fill zeros to maintain the diagonal structure; otherwise, if a large number of zeros are needed, we believe that the diagonal is broken and the diagonal pattern should be changed. For example, a zero is filled at v43 position to maintain the diagonal structure, while the main diagonal is broken. The application developer can set the maximum number of filled zeros according to the property of application and the problem size.

2.2 Application Specific Diagonal Pattern

The diagonal patterns that remain unchanged among different problem sizes are abstracted as **Application Specific Diagonal Pattern** and stored into one group with group type ***Application Specific*** (***AS***). As there are more than one application specific diagonal patterns, a tag is needed to identify them. In this way, the group is represented as (AS, tag). We analyze one matrix, named the sample matrix, from a given application to obtain application specific diagonal

patterns. For the matrix listed in Fig. 1, if the second diagonal pattern is viewed as application specific diagonal pattern with tag 1, then

$$(\text{AS}, 1) = \{(\text{AD}, 2), (\text{NAD}, 1)\}$$
$$\text{matrix} = \{\{(\text{NAD},1),(\text{AD},2),(\text{NAD},2)\}, \{(\text{AS}, 1)\}\}.$$

2.3 Storage Format

We have grouped the diagonals using diagonal pattern. Furthermore, the matrix is split into row segments. The number of rows in each row segment is defined as ***row segment size*** and represented by the token ***mrows***. In this way, the whole matrix is split in two dimensions, as the dotted lines show in Fig. 1. In each row segment, nonzeros on the diagonals of the same group are the ***storage unit*** of CRSD. Additionally, if only one nonzero is on a diagonal within one row segment, the nonzero is viewed as scatter point, such as v55 in Fig. 1.

In CRSD storage format, the scatter points and the nonzeros in diagonal are stored separately. In order not to change the order of floating point operations, the whole row where the scatter point locates is stored together. The row number, the number of nonzeros in this row and the column index of each nonzero are used as the indices and stored in array ***scatter_index***. The nonzero values are stored in array ***scatter_val***.

Except scatter points, the whole matrix is represented by diagonal patterns. All nonzeros in the same diagonal pattern share the same index: the diagonal pattern, the start row number of the diagonal pattern, the number of row segments, and the column indices of diagonals. The column index of each diagonal is needed for nonadjacent group, while only the column index of first diagonal in adjacent group needs to be recorded. The diagonal pattern is stored in array ***matrix*** and the remaining of index value is stored in array ***crsd_dia_index***. The nonzero values in each storage unit are stored contiguously in array ***crsd_dia_val***, such as v20, v31, v21 and v32.

The number of diagonal patterns and rows that contain the scatter point are assigned to ***num_dia_patterns*** and ***num_scatter_rows*** respectively. An example is shown in Fig. 2 for the matrix in Fig. 1, when row segment size is 2.

num_scatter_rows=1
num_dia_patterns=2

matrix={{(NAD,1),(AD,2),(NAD,2)}, {(AS, 1)} }

crsd_dia_index = {R0, 1, C0, C2, C5, C7, | R2, 2, C0, C4}
crsd_dia_val = {{(v00,v11),(v02,v13,v03,v14),(v05,v16,v07,v18)}, {(v20,v31,v21,v32),(v23,v24)}, {(v42,v53,***0***,v54),(v45,v56)} }

scatter_index = {R5, 4, C3, C4, C5, C6}
scatter_val = {v53, v54, v55, v56}

Fig. 2. The CRSD storage format for matrix shown in Fig. 1 when mrows=2

2.4 SpMV Implementation for CRSD

In the SpMV implementation for CRSD, the storage unit is also the *operation unit*, for the reason that all SpMV operations on the elements in each storage unit are organized together. When we set the upper limit of the number of diagonals in (non)adjacent group, it is practical to enumerate the SpMV operations for all kinds of groups. Once the number of diagonals in one group exceeds the upper limit, it will be split into many sub-groups until the number of diagonals is less than the upper limit.

```
for each row-segment in diagonal pattern
    for each group of the diagonal pattern in one row segment
        switch group type
        case (NAD, 1): operation for group (NAD, 1)
        case (AD, 2) etc: operation for the enumerated groups
        case (AS, 1): // for application specific diagonal pattern
            for each row-segment in diagonal pattern // Loop inside
                // the generated SpMV codelet for (AS, 1)={(AD,2),(NAD,1)}
```

C1
x vector ... ...
V1
V2

...
load x[c1] to register x1
load v1, v2
y2 += v2 * x1;
y1 += v1 * x1; // x1 is reused
Store y2 into destination vector y
...

LAT_RD for x1
LAT_OP for y2

```
            done
        end switch
    done
done
```

Fig. 3. SpMV code fragment for matrix shown in Fig. 1 when mrows=2

As application specific diagonal patterns are only available after sampling the sample matrix, a code generator is designed to generate codelets for those diagonal patterns. Because application specific diagonal patterns are inherent to an application, the SpMV for CRSD becomes application specific. In addition, it is not reasonable to generate all application specific diagonal patterns, since some patterns cover few nonzero values. A *threshold* is set to determine the least number of nonzeros that an application specific diagonal pattern should cover.

A SpMV code fragment is given in Fig. 3. In processing the adjacent group, the elements of x can be reused. For this reason, we can load the element into a register and use it repeatedly, such as register x1. The group with type AS represents application specific diagonal pattern and describes diagonal distribution of entire row segment. Then operations for entire row segment are organized in one loop.

3 Application Specific Automatic Performance Tuning

In order to improve the performance as well as portability of the generated codelets, we apply auto-tuning to select the optimal implementation. For the

reason that the SpMV for CRSD is application specific, the auto-tuning process is also application specific.

We optimize the generated codelets by applying SSE intrinsics and explicit prefetching. The SSE intrinsics allow simultaneous operations on a vector of two double precision values. Explicit prefetching is implemented via compiler intrinsic *__builtin_prefetch*. It sets *prefetch distance* to determine which elements to be preloaded. Meanwhile, it can change the temporal locality of the preloaded elements by modifying the *prefetch locality*, ranging from 0 to 3. The bigger the *prefetch locality* is, the higher the temporal locality is. Furthermore, we reschedule the SpMV operation via modifying the latency between data read and data available(LAT_RD) as well as the latency between data operation and result available(LAT_OP)(as shown in Fig. 3).

For different diagonal patterns, the performance is affected by the following parameters on different hardware platforms and the value ranges are determined according to the experimental statistics results:

- *mrows*. It determines the number of nonzeros to be processed in one loop, ranging from 2 to 8;
- *prefetch distance* and *prefetch locality*. The prefetch function is applied to nonzeros and vector x. The range of prefetch distance is from 30 to 300.
- LAT_RD and LAT_OP. If the number of SSE registers is defined as num_SSE_regs. Their range is from 1 to $num_SSE_regs/3$;

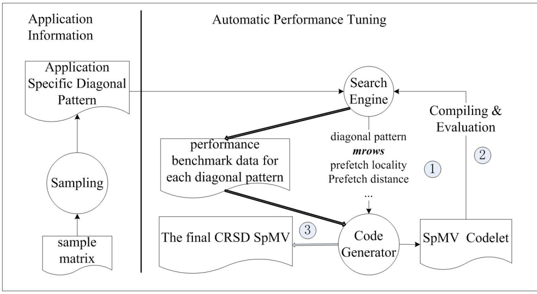


Fig. 4. Application specific auto-tuning

Table 1. Experimental Platforms

platform #	AMD	Intel
CPU	AMD PhenomTM II X4 940, 3.0 GHz	Intel Xeon X5550, 2.67GHz
MEM	8GB	8GB
Sockets	1	2
Compiler	GCC 4.3.3	GCC 4.4.3
Compiler option	-msse2 -O3 -fopenmp	

After obtaining the application specific diagonal patterns, the whole automatic performance tuning process is described as follows (shown in Fig 4):

Step 1. Search engine reads each application specific diagonal pattern, determines value set of parameters and passes them to code generator.

Step 2. The SpMV Codelet is generated, compiled and executed. The performance information is sent back to search engine and recorded until all application specific diagonal patterns are measured.

Step 3. The optimal performance and the corresponding parameter values for all application specific diagonal patterns are sent to the code generator to produce the final SpMV implementation.

The matrix, which is used to evaluate the generated codelet in step 2, is extended according the indices of the corresponding diagonal pattern, since the performance of SpMV is affected by the input sparse matrix.

The search engine uses orthogonal search method[6] to determine the parameter value set. The search order is *prefetch distance, prefetch locality, LAT_RD, LAT_OP* and *mrows*. Moreover, the entire process is completed during the building phase rather than at runtime.

3.1 The Final CRSD SpMV Implementation

As the auto-tuning records show, the row segment size is not same for different diagonal patterns when the performance of generated codelet is optimal. When we split the matrix in row direction, we chose different row segment size for different diagonal patterns. The SpMV codelet for each diagonal pattern should be generated according the corresponding row segment size. Then we combine those generated codelets to produce the final CRSD SpMV implementation.

3.2 Parallelization

When one matrix is stored in CRSD storage format, the diagonal pattern and corresponding number of row segments are obtained. Given the performance for processing each diagonal pattern, we can estimate the execution time. Then we can split the matrix into sub-matrices and keep the estimated time for processing each sub-matrix equal. The diagonal patterns may be split in the process when necessary. The parallelization is implemented using OpenMP. specifically, we distribute the scatter points according to the row range of each sub-matrix to avoid write confliction of destination vector y.

4 Evaluation

In this section, we present the performance improvement of CRSD on two platforms(Table 1) and 13 matrices(Table 2). Those matrices are categorized in five classes: the first three classes come from [14]; the last two classes are from an astrophysics application [15]. The coverage represents the percentage of number of nonzeros in this diagonal pattern and threshold 15%, mentioned in section 2.4, is used to identify the application specific diagonal pattern.

We choose CSR and DIA storage formats to compare with our CRSD storage format. Owing to that SpMV is not available in ACML, we select Intel MKL, with version 10.2.6.038, on the two x86-based architectures.

We also select OSKI-1.0.1h[7] for the comparison. When we set hint that all the matrices are diagonal sparse matrices, it fails to tune the matrices and return the input matrices.The same situations occur in earlier work[4][5].Thus the performance of CSR can be viewed as the result of OSKI.

Table 2. Matrix Set and Application Specific Diagonal patterns

#	Matrix Information				Application Specific Diagonal pattern		
	name	row	nnz	picture	#	content	Coverage(%)
1	atmosmodd*	1270423	8814880				
2	atmosmodj	1270423	8814880		Dp1	{(NAD, 2), (AD, 3), (NAD, 2)}	95.3
3	atmosmodm	1270423	8814880				
4	cell1*	7055	30082				
5	cell2	7055	30082		Dp2	{(AD, 2), (NAD, 1), (AD, 2)}	79.9
6	kim1*	38415	933195				
7	kim2	456976	11330020		Dp3	{(AD,5),(AD,5), (AD, 5), (AD, 5) (AD, 5)}	98.0
8	A1	620000	4917600				
9	A2	1080000	8578800		Dp4	{(NAD,2), (AD,3), (NAD,3)}	44.0
10	A3*	320000	2532800		Dp5	{(NAD,3), (AD,3), (NAD,2)}	45.9
11	B1	620000	4917600				
12	B2	1080000	8578800		Dp6	{(NAD,2), (AD,5),(NAD,2)}	97.5
13	B3*	320000	2532800				

* the sample matrix

4.1 The Auto-Tuning Records

The auto-tuning records on two platforms are given in Table 3. From the records we can conclude that the parameter values are different when the performance is optimal for the same diagonal pattern on different platforms.

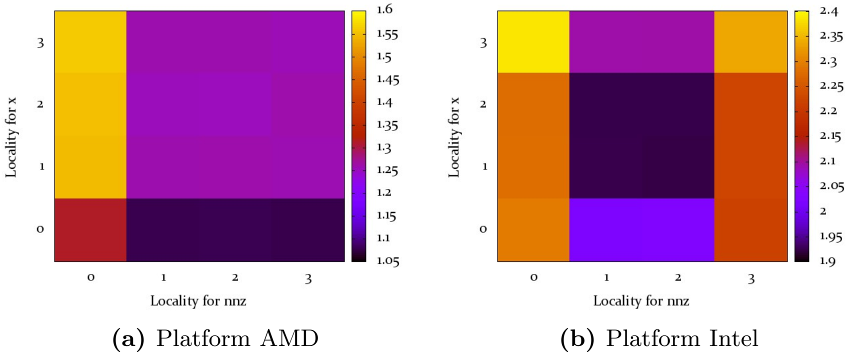
Table 3. The Auto-tuning Records for Application Specific Diagonal patterns

Dp#	Segment size		Locality for x		Locality for nonzeros		Prefetch distance		*LAT_RD*		*LAT_OP*	
	AMD	Intel	AMD	Intel	AMD	Intel	AMD	Intel	AMD	Intel	AMD	Intel
Dp1	3	8	2	3	0	0	50	30	2	3	4	2
Dp2	8	8	2	3	1	0	150	30	2	4	3	4
Dp3	5	4	2	3	1	0	150	90	3	4	3	2
Dp4	3	6	3	3	0	0	50	150	2	2	2	3
Dp5	2	6	2	3	0	0	150	90	3	3	3	3
Dp6	4	4	3	3	0	0	40	90	3	3	3	3

The *prefetch locality* for x is almost 3, the highest temporal locality, whereas that for nonzeros is almost 0. The effect of *prefetch locality* for matrix B3 on two platforms is shown in Fig 5. The maximum difference of performance reaches up to 44.8% and 24.0% on platform AMD and Intel respectively. The major reason is the different access behavior of nonzeros and vector x: the elements of x may be accessed repeatedly, which is determined by nonzero distribution, whereas the elements in nonzeros are only accessed once.

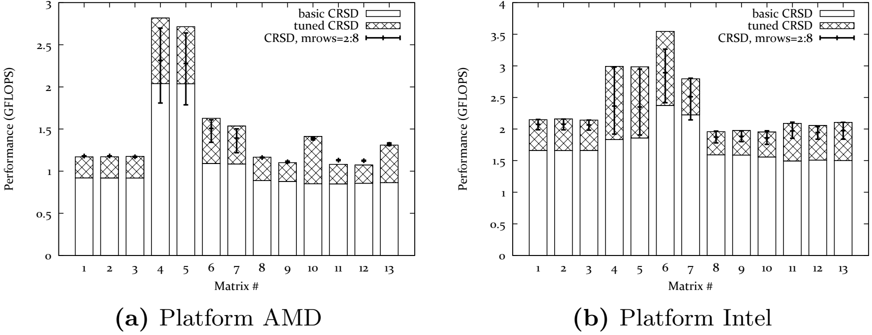
We also collect the performance data affected by *prefetch distance* for matrix B3. The maximum difference of performance is only 11.8% and 2.2% on platform AMD and Intel respectively. This phenomenon exists for other matrices. We can conclude that prefetch locality plays a more important role than prefetch distance on SpMV performance for diagonal sparse matrices.

The performance improvement for the automatic performance tuning is given in Fig 6. The final CRSD SpMV uses the variable row segment and other optimization methods, such as SSE intrinsic. The performance using only variable row segment size is viewed as the basic CRSD implementation. The performance



(a) Platform AMD (b) Platform Intel

Fig. 5. The performance effect of prefetch locality on AMD and Intel platforms



(a) Platform AMD (b) Platform Intel

Fig. 6. The perfomance improvement of auto-tuning

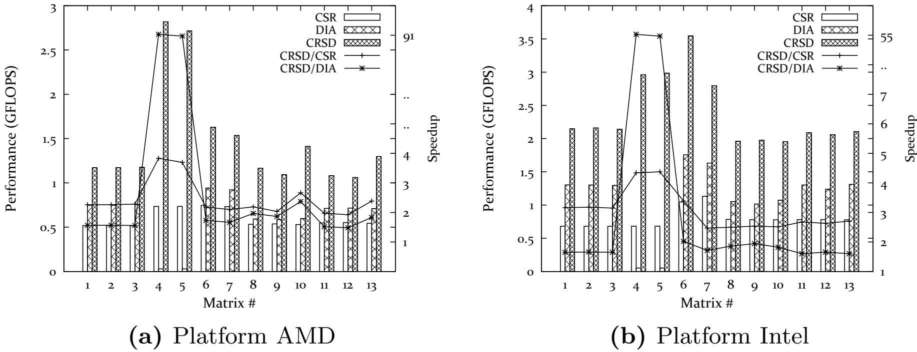
gain differs among the matrices and the platforms. The average performance improvement is 36.3% and 36.6% and maximum is 65.9% and 63.1% on platform AMD and Intel respectively.

In Fig 6, the performance range of CRSD with different identical ***mrows*** is also given. The final CRSD SpMV outperforms those implementations for all matrices on platform Intel. On platform AMD, the performance difference between the final CRSD SpMV and upper range bound is less than 4%. This verifies that the final CRSD SpMV is efficient.

4.2 Serial Performance Improvement

The performance comparison with CSR and DIA is given in Fig 7. The maximum speedup compared with CSR reaches 3.83 and 4.38 on platform AMD and Intel respectively. And the average of speedup reaches 2.44 and 3.05.

In comparison with DIA, we find that the performance of DIA for cell1 and cell2 is very poor. For the reason that the number of filled zeros is almost 33 times larger than the number of nonzeros. The nonzeros distributes on 169 distinct diagonals. Therefore large number of idle sections exist. Even CSR is faster



(a) Platform AMD **(b)** Platform Intel

Fig. 7. Serial performance comparison results

than DIA by the factor of 23.50 and 11.87 on AMD and Intel respectively. However, using diagonal pattern CRSD is suitable for the two matrices, especially application specific diagonal pattern covers 79.9% of the nonzeros. In comparison with CSR, the speedups reach 3.69 and 4.34 on platform AMD and Intel respectively. Except cell1 and cell2, the maximum speedups reach 2.37 and 2.02 on platform AMD and Intel respectively. The average reaches 1.73 and 1.74.

4.3 Parallel Performance Improvement

Since the implementation based on the DIA is not parallelized in Intel MKL, only the CSR format is used for the comparison. The comparison results are shown in Fig 8. CRSD outperforms CSR under different number of available threads on two platforms. The maximum and average speedups are listed in Table 4.

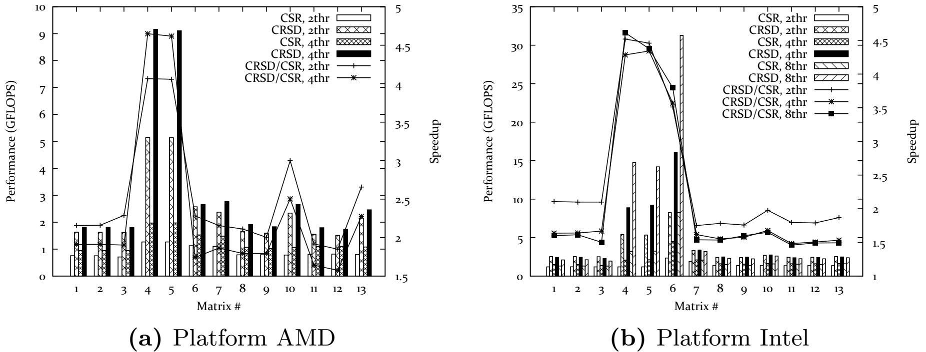
Table 4. Speedup of parallel CRSD compared with parallel CSR

# of threads		2	4	8
AMD	Max	4.07	4.64	X
	Average	2.51	2.32	X
Intel	Max	4.51	4.34	4.61
	Average	2.42	2.16	2.17

The performance of CRSD for matrices with prefix cell and kim are relatively high, especially for matrices whose sizes are small enough to be fitted into cache. In those matrices, a large percent of diagonals are adjacent. Hence the elements of x are reused frequently.

5 Related Work

Im and Yelick et al. propose register blocking, cache blocking and reordering techniques. Register blocking[3][7][8] is based on BCSR format. BCSR is suitable for matrices, in which nonzeros primarily distribute in dense blocks. This



(a) Platform AMD **(b)** Platform Intel

Fig. 8. Performance comparison results between parallel CSRD and parallel CSR

property is universal for the matrices produced by Finite Element Method(FEM, another major method for PDEs). Vuduc et al. estimate the performance bounds for the register blocking and propose a new approach to choose the register block size[9]. However, excessive zeros have to be filled to maintain the block format in BCSR, which wastes the computation and memory resources. To reduce the number of filled zero, Vuduc et al. in [2][10] exploit variable block structure rather than identical block size; Belgin et al. explore the distribution pattern of nonzeros in dense block and propose PBR to store matrices without zero filling[4]. Cache blocking[3] is used to increase the temporal locality by reordering the memory access, Nishtala et al. present a new performance models, which takes TLB misses into account, and a criteria to determine when to apply the cache blocking [11]. Samuel Williams [5] sums up all those optimization methods on the emerging multi-core platforms. To mitigate the memory bandwidth pressure, Willcock[12] and Kourtis et al. [13] utilize data compression to reduce the index. Furthermore Kourtis also introduce CSR-VI to compress the nonzero value when most of nonzero values are identical.

6 Conclusion

In this paper, we propose CRSD for the diagonal sparse matrix. We design diagonal pattern to describe the diagonal distribution, making CRSD more suitable than DIA. Furthermore, as diagonal distributions are similar for different problem sizes, we introduce the idea of application specific diagonal pattern to optimize SpMV implementation. During the building phase, the optimal codelets for application specific diagonal patterns are generated automatically. It differs from OSKI, for OSKI chooses the optimal implementation at runtime. The auto-tuning records are also utilized to achieve load-balance for parallelization.

The results from our experiments demonstrate that CRSD is efficient for processing the diagonal sparse matrices from one application, when there are several major diagonal patterns and diagonal distribution remains similar among the matrices. We are transplanting the CRSD to the GPU. Our preliminary tests on

GPU indicate a strong potential for better performance. In the future, we will study more types of nonzero distributions and optimize the SpMV specifically on distinct architectures.

References

1. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput oriented processors. In: Supercomputing (2009)
2. Vuduc, R.W.: Automatic Performance of Sparse Matrix Kernels. The dissertation of Ph.D, Computer Science Division, U.C. Berkeley (2003)
3. Im, E.: Optimizing the performance of sparse matrix-vector multiplication. PhD thesis, University of California, Berkeley (2000)
4. Belgin, M., Back, G., Ribbens, C.J.: Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In: International Conference on Supercomputing, NY, USA (2009)
5. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, Reno, Nevada, November 10-16 (2007)
6. Kulkarni, M., Pingali, K.: An experimental study of self-optimizing dense linear algebra software. Proceedings of the IEEE 96(5), 832–848 (2008)
7. Vuduc, R., Demmel, J., Yelick, K.: OSKI: A library of automatically tuned sparse matrix kernels. In: Proceedings of SciDAC 2005, Journal of Physics: Conference Series (2005)
8. Im, E.-J., Yelick, K.A.: Optimizing sparse matrix computations for register reuse in SPARSITY. In: Alexandrov, V.N., Dongarra, J., Juliano, B.A., Renner, R.S., Tan, C.J.K. (eds.) ICCS-ComputSci 2001. LNCS, vol. 2073, pp. 127–136. Springer, Heidelberg (2001)
9. Vuduc, R., Demmel, J., Yelick, K., Kamil, S., Nishtala, R., Lee, B.: Performance optimizations and bounds for sparse matrix-vector multiply. In: Supercomputing, Baltimore, MD (2002)
10. Vuduc, R.W., Moon, H.-J.: Fast sparse matrix-vector multiplication by exploiting variable block structure. In: Yang, L.T., Rana, O.F., Di Martino, B., Dongarra, J. (eds.) HPCC 2005. LNCS, vol. 3726, pp. 807–816. Springer, Heidelberg (2005)
11. Nishtala, R., Vuduc, R., Demmel, J.W., Yelick, K.A.: When cache blocking sparse matrix vector multiply works and why. Applicable Algebra in Engineering, Communication, and Computing (2007)
12. Willcock, J., Lumsdaine, A.: Accelerating sparse matrix computations via data compression. In: ICS 2006: Proceedings of the 20th Annual International Conference on Supercomputing, pp. 307–316. ACM Press, New York (2006)
13. Kourtis, K., Goumas, G., Koziris, N.: Optimizing sparse matrix-vector multiplication using index and value compression. In: Proceedings of the 5th Conference on Computing Frontiers, Ischia, Italy, May 5-7 (2008)
14. Boisvert, R., Pozo, R., Remington, K., Miller, B., Lipman, R.: NISTMatrixMarket, `http://math.nist.gov/MatrixMarket/index.html`
15. Chana, K.H., Li, L., Liao, X.: Modelling the core convection using finite element and finite difference methods. Physics of the Earth and Planetary Interiors 157(2), 124–138 (2006)