Semiautomatic Acceleration of Sparse Matrix-Vector Product Using OpenACC

Przemysław Stpiczyński(✉)

Institute of Mathematics, Maria Curie–Skłodowska University,
Pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin, Poland
przem@hektor.umcs.lublin.pl

Abstract. The aim of this paper is to show that well known SPARSKIT SpMV routines for *Ellpack-Itpack* and *Jagged Diagonal* formats can be easily and successfully adapted to a hybrid GPU-accelerated computer environment using OpenACC. We formulate general guidelines for simple steps that should be done to transform source codes with irregular data access into efficient OpenACC programs. We also advise how to improve the performance of such programs by tuning data structures to utilize hardware properties of GPUs. Numerical experiments show that our accelerated versions of SPARSKIT SpMV routines achieve the performance comparable with the performance of the corresponding CUSPARSE routines optimized by NVIDIA.

Keywords: Sparse matrices · SpMV · GPUs · OpenACC · CUSPARSE

1 Introduction

Recently, GPU-accelerated computer architectures have become very attractive for achieving high performance execution of scientific applications at low costs [1,2], especially for linear algebra computations [3,4]. Unfortunately, the process of adapting existing software to such new architectures can be difficult. Compute Unified Device Architecture (CUDA) programming interface can be used only for NVIDIA cards, while the use of OpenCL (Open Computing Language [5]) leads to a substantial increase of software complexity.

SPARSKIT is a well known package tool for manipulating and working with sparse matrices [6]. It is a very good example of widely used valuable software packages written in Fortran. Unfortunately, it does not utilize modern computer architectures, especially GPU-accelerated multicore machines. The new implementation of the most important SPARSKIT routines for NVIDIA GPUs has been presented in [7].

Sparse matrix-vector product (SpMV) is a central part of many numerical algorithms [6,8]. There are a lot of papers presenting rather sophisticated techniques for developing SpMV routines that utilize the underlying hardware of GPU-accelerated computers [9–13]. Unfortunately, these methods are rather complicated and usually machine-dependent. However, the results presented

© Springer International Publishing Switzerland 2016
R. Wyrzykowski et al. (Eds.): PPAM 2015, Part II, LNCS 9574, pp. 143–152, 2016.
DOI: 10.1007/978-3-319-32152-3_14

in [14] show that simple SPARSKIT SpMV routines using CSR (Compressed Sparse Row) format [6] can be easily and efficiently adapted to modern multicore CPU-based architectures. Loops in source codes can be easily parallelized using OpenMP directives [15,16], while the rest of the work can be done by a compiler. Such parallelized SpMV routines achieve the performance comparable with the performance of the SpMV routines available in libraries optimized by hardware vendors (i.e. Intel MKL).

OpenACC is a new standard for accelerated computing [17]. It offers compiler directives for offloading C/C++ and Fortran programs from host to attached accelerator devices. Such simple directives allow to mark regions of source code for automatic acceleration in a vendor-independent manner [18]. However, sometimes it is necessary to apply some high-level transformations of source codes to achieve reasonable performance [19–21]. Paper [22] shows attempts to apply OpenACC for accelerating SpMV. However, the authors consider only some modifications of the CSR format and apply other GPU-specific optimizations (just like communication hiding).

In this paper we show that well known SPARSKIT SpMV routines for *Ellpack-Itpack* (ELL) and *Jagged Diagonal* (JAD) formats [6] can be easily and successfully adapted to a hybrid GPU-accelerated computer environment using OpenACC. We also advise how to improve the performance of such programs by tuning data structures to utilize hardware properties of GPUs applying some high-level transformation of the source code. The paper is structured as follows. Section 2 describes ELL and JAD – two formats which are suitable for GPU-accelerated computations. We show how to apply some basic source code transformations to obtain accelerated versions of SpMV routines. In Sect. 3 we present pJAD - a new format, which allows to outperform SpMV routine for JAD. Section 4 discusses the results of experiments performed for a set of test matrices. We also compare the performance of our OpenACC-accelerated routines with the performance of SpMV for the HYB (ELL/COO) format [23]. Finally, in Sect. 5 we formulate general guidelines for simple steps that should be done to transform irregular source codes into OpenACC programs.

2 SPARSKIT and SpMV Routines

ELL format for sparse matrices assumes the fixed-length rows [24]. A sparse matrix with n rows and at most $ncol$ nonzero elements per row is stored columnwise in two dense arrays of dimension $n \times ncol$ (Fig. 1). The first array contains the values of the nonzero elements, while the second one contains the corresponding column indices.

JAD format removes the assumption on the fixed-length rows [7]. Rows of a matrix are sorted in non-increasing order of the number of nonzero elements per row (Fig. 2). The matrix is stored in three arrays. The first array `a` contains nonzero elements of the matrix (i.e. jagged diagonals), while the second one (i.e. `ja`) contains column indices of all nonzeros. Finally, the array `ia` contains the beginning position of each jagged diagonal. The number of jagged diagonals is

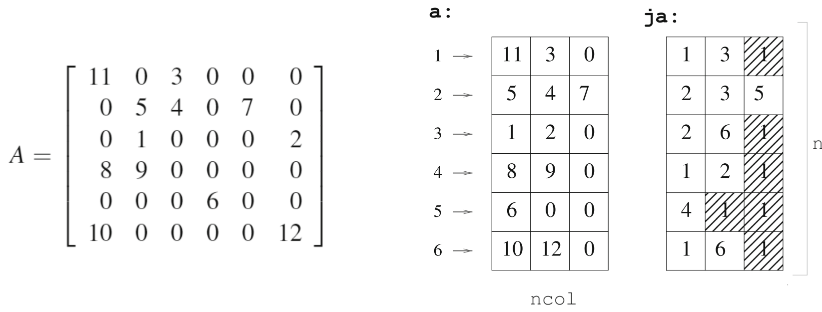

Fig. 1. ELL format for sparse matrices

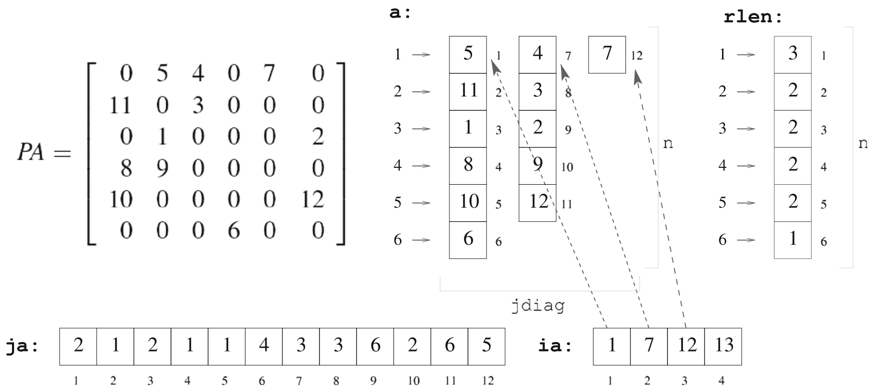

Fig. 2. JAD format for sparse matrices

stored in `jdiag`. Optionally, we can consider just another array `rlen` containing lengths of all rows [11]. Elements of this array can be easily calculated (even in parallel) using the following formula

$$\mathbf{rlen}(i) = |\{j : 1 \le j \le \mathbf{jdiag} \wedge \mathbf{ia}(j+1) - \mathbf{ia}(j) \ge i\}|, \quad i = 1, \ldots, n. \tag{1}$$

Figure 3 shows Fortran subroutines which implement SpMV for ELL and JAD. Note that SPARSKIT subroutines `amuxe` and `amuxj` were originally written in Fortran 77, but here we present their equivalents written in Fortran 90.

OpenACC provides the `parallel` construct that launches gangs that will execute in parallel. Gangs may support multiple workers that execute in vector or SIMD mode [17]. This standard also provides several constructs that can be used to specify the scope of data in accelerated parallel regions. It should be noticed that proper data placement and carefully planned data transfers can be crucial for achieving reasonable performance of accelerated programs [19].

In our OpenACC program, a GPU is responsible for performing SpMV while the host program has to read data and initialize computations. The accelerated subroutines `accamuxe` and `accamuxj` are presented in Fig. 4. From the

```fortran
subroutine amuxe (n,x,y,na,ncol,a,ja)
 real*8 x(n), y(n), a(na,*)
 integer  n, na, ncol, ja(na,*)
 integer i, j

 do 1 i=1, n
   y(i) = 0.0
 end do
 do j=1,ncol
   do i=1,n
     y(i)=y(i)+a(i,j)*x(ja(i,j))
   end do
 end do
end subroutine amuxe
```

```fortran
subroutine amuxj(n,x,y,jdiag,a,ja,ia)
  integer n, jdiag, ja(*), ia(*)
  real*8 x(n), y(n), a(*)
  integer i, ii, k1, len, j

  do i=1, n
    y(i) = 0.0d0
  end do
  do ii=1, jdiag
     k1 = ia(ii)-1
     len = ia(ii+1)-k1-1
     do j=1,len
        y(j)=y(j)+a(k1+j)*x(ja(k1+j))
     end do
  end do
end subroutine amuxj
```

Fig. 3. SPARSKIT SpMV for ELL (left) and JAD (right)

developer's point of view, the OpenACC `parallel` construct together with `vector_length` should be used to vectorize loops. In case of `amuxe`, the simplest way to accelerate SpMV is to vectorize the loops 6–8 and 10–12. Then, the loop 9–13 would repeat generated kernel `ncol` times. However, it is better to apply the loop exchange. In `accamuxe`, the outermost loop 9–16 is vectorized. Similarly we obtain `accamuxj`. In case of this routine we can observe that the loop 11–19 works on rows, thus we have to provide the length of each row in `rlen`. Note that to avoid unnecessary transfers, we use the clause `present` to specify that the data already exist in the device memory.

```fortran
subroutine accamuxe(n,x,y,na,ncol,a,ja)
 real*8 x(n), y(n), a(na,*)
 integer  n, na, ncol, ja(na,*)
 integer i, j
 real*8 t

!$acc parallel loop vector_length(128)&
!$acc present(x,y,a,ja)
 do i = 1,n
   t=0
   !$acc loop seq
   do j=1,ncol
     t = t+a(i,j)*x(ja(i,j))
   end do
   y(i)=t
 end do
end subroutine accamuxe
```

```fortran
subroutine accamuxj(n,x,y,jdiag,a,ja,ia,
                             rlen,iperm)
 integer n,jdiag,ja(*),ia(*),rlen(*),
                               iperm(*)
 real*8 x(n), y(n), a(*)
 integer i, ii, k1, len, j, k
 real*8 t

!$acc parallel loop vector_length(128)&
!$acc present(a,ja,ia,y,x,rlen,iperm)
 do i=1, n
   t = 0.0d0
   !$acc loop seq
   do j=1,rlen(i) !for each within a row
      k=ia(j)-1+i
      t=t+a(k)*x(ja(k))
   end do
   y(iperm(i)) = t  !apply permutation
 end do
end subroutine accamuxj
```

Fig. 4. Accelerated versions of SpMV for ELL (left) and JAD (right)

3 Optimizing SpMV Using pJAD Format

Our version of SpMV for JAD can be further optimized. We can improve memory access by aligning (padding) columns of the arrays `a` and `ja`. Thus, in each column we add several zero elements and each column's length should be a multiple of a given *bsize*. Then, each block of threads will have to work on rows of the same length. The number of elements in `a` and `ja` will be increased to the size which is bounded by $n_{nz} + jdiag \cdot (bsize - 1)$, where n_{nz} is the number of nonzero elements (Fig. 5). This modified format can be called pJAD (i.e. padded JAD format). Similar modifications have been introduced in [25]. However, Kreutzer et al. consider *bsize* equal to the length of half-warp, what is specific for NVIDIA GPUs. They also assume that threads with a block can be responsible for processing various amount of data. Figure 6 shows the source code of `accpamuxj`. Note that the array `brlen` contains the length of each block of rows of a given size *bsize*.

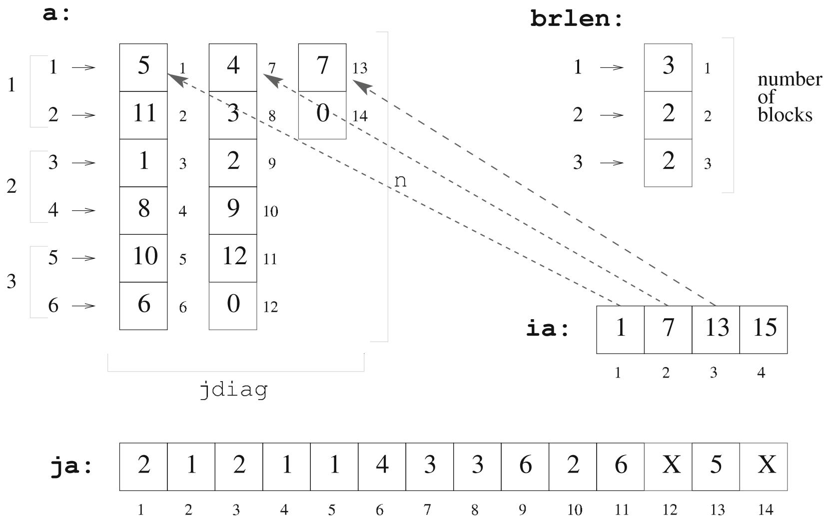

Fig. 5. pJAD format and its data structures

4 Results of Experiments

Our OpenACC implementation of the SpMV routines has been tested on a computer with two Intel Xeon X5650 (6 cores each with hyper- threading, 2.67 GHz, 48 GB RAM) and two NVIDIA Tesla M2050 (448 cores, 3 GB GDDR5 RAM with ECC off), running under Linux under with NVIDIA CUDA Toolkit version 6.5 and PGI Accelerated Server version 15.4, which supports OpenACC [26].

```
1   subroutine accpamuxj(n,x,y,jdiag,a,ja,ia,brlen,iperm,bsize)
2    integer n,jdiag,ja(*),ia(*),brlen(*),iperm(*),bsize
3    real*8 x(n), y(n), a(*)
4    integer i,ii,k1,len,j,k
5    real*8 t
6   !
7    !$acc parallel loop vector_length(128) &
8    !$acc present(a,ja,ia,y,x,brlen,iperm)
9    do i=1,n
10     t = 0.0d0
11     !$acc loop seq
12     do j=1,brlen((i-1)/bsize+1)
13       k=ia(j)-1+i
14       t=t+a(k)*x(ja(k))
15     end do
16     y(iperm(i)) = t
17   end do
18  end subroutine accpamuxj
```

Fig. 6. Accelerated SpMV routine for pJAD format

Table 1 summarizes the results obtained for a set of test matrices chosen from *Matrix Market* [27] and *University of Florida sparse matrix collection* [28].

The set contains various matrices with different number of rows and nonzero elements. The largest *cage15* has over $5 \cdot 10^6$ rows and almost 10^8 nonzero elements. For each matrix we provide the number of rows (columns), number of nonzero entries, average number of nonzero entries per row, maximum number of nonzero elements within a row. We also show the performance (in GFLOPS) of accelerated versions of SpMV for ELL, JAD and pJAD. The last column shows the performance of CUSPARSE SpMV routine using HYB (i.e. hybrid format [23]). In Table 1, the best performance for each matrix is underlined.

The HYB sparse storage format is composed of a regular part stored in ELL and an irregular part stored in COO. CUSPARSE conversion operation from CSR to HYB partitions a given sparse matrix into the regular and irregular parts automatically or according to developer-specified criteria [23]. For our tests, we have chosen the first option.

We can observe that for almost all matrices pJAD format achieves better performance than ELL and JAD. ELL outperforms pJAD only for matrices *cry1000*, *af23560*, *majorbasis*, *ecology2*, *atmosmodl*, where all rows have almost the same number of nonzero elements (i.e. $n_{nz}/n \approx \max_{nz}$) or where the number of nonzero elements is rather big in comparison with the number of rows (i.e. $n \ll n_{nz}$ for *nd24k*). It should be noticed that for some matrices ELL exceeds the memory capacity of Tesla M2050 (*pre2*, *torso1*, *inline_1*). The performance of pJAD is a little bit worse than the performance of HYB, because pJAD format requires re-permutation of the result's entries. For some matrices with $n_{nz}/n \ll \max_{nz}$, pJAD outperforms HYB (i.e. *af23560*, *bcsstk36*, *bbmat*, *cfd1*, *torso1*, *ldoor*). Note that for *cage15*, CUSPARSE routine for conversion from CSR to HYB has failed because memory capacity has been exceeded.

Table 1. Results of experiments for a set of test matrices

Matrix	n	n_{nz}	n_{nz}/n	$\max_{nz}$	ELL	JAD	pJAD	HYB
cry10000	10000	49699	5.0	5	3.99	3.79	3.37	2.11
poisson3Da	13514	352762	26.1	110	2.14	3.40	3.54	4.41
af23560	23560	484256	20.6	21	12.03	11.88	11.94	9.05
g7jac140	41490	565956	13.6	153	1.21	3.23	3.39	4.90
fidapm37	9152	765944	83.7	255	4.97	6.19	6.78	6.84
bcsstk36	23052	1143140	49.6	178	4.34	11.64	13.52	9.21
majorbasis	160000	1750416	10.9	11	15.20	13.78	13.80	14.01
bbmat	38744	1771722	45.7	126	6.03	7.73	8.88	7.34
cfd1	70656	1828364	25.9	33	12.84	12.77	13.21	12.83
ASIC_680ks	682712	2329176	3.4	210	0.28	4.32	4.46	7.54
FEM_3D_thermal2	147900	3489300	23.6	27	13.09	13.13	14.95	14.98
parabolic_fem	525825	3674625	7.0	7	9.80	9.33	9.83	11.58
ecology2	999999	4995991	5.0	5	13.83	12.00	12.84	15.76
pre2	659033	5959282	9.0	628	—	6.59	7.32	8.74
boneS01	127224	6715152	52.8	81	9.96	9.51	11.43	12.58
torso1	116158	8516500	73.3	3263	—	10.70	11.73	6.96
thermal2	1228045	8580313	7.0	11	5.92	4.80	5.03	8.65
atmosmodl	1489752	10319760	6.9	7	13.67	12.42	12.68	15.55
bmw3_2	227362	11288630	49.7	336	2.16	11.58	14.17	16.08
af_shell8	504855	17588875	34.8	40	13.56	14.69	17.59	19.48
cage14	1505785	27130349	18.0	41	6.40	9.48	11.05	12.79
nd24k	72000	28715634	398.8	520	11.46	4.23	4.52	12.52
inline_1	503712	36816342	73.1	843	—	9.89	11.74	12.26
ldoor	952203	46522475	48.9	77	9.05	12.52	15.43	14.68
cage15	5154859	99199551	19.2	47	5.86	9.12	10.49	—

5 Conclusions and Future Work

We have shown that well known SPARSKIT SpMV routines for ELL and JAD formats can be easily and successfully adapted to a hybrid GPU-accelerated computer environment using OpenACC. Such routines achieve reasonable performance. Further improvements can be obtained by introducing the new data formats for sparse matrices to utilize specific GPU hardware properties. Numerical experiments have justified that the performance of our optimized SpMV routines is comparable with the performance of the routine provided by the vendor. We have also discussed when the use of considered formats would be profitable. We believe the use of OpenACC and accelerated Fortran routines

can be attractive for people who prefer to develop applications using high-level directive programming techniques instead of complicated CUSPARSE API.

The general guidelines for semiautomatic acceleration of irregular codes using OpenACC can be summarized as follows:

1. Define regions where data should exist on accelerators. Try to reduce transfers between host and accelerators.
2. Try to vectorize outermost loops within your code. Vectorized loops should have sufficient computational intensity, namely the ratio of the number of computational operations to the number of memory operations should be greater than one.
3. If necessary, apply loop exchange and inform the compiler that loops are safe to parallelize using the `independent` clause in OpenACC `loop` constructs.
4. Try to keep threads within gangs (or *blocks* in terms of CUDA) working on the same amount of data.
5. The best performance occurs when coalesced memory access takes place [29,30]. Threads within gangs should operate on contiguous data blocks.
6. Tune your data structures by aligning data in arrays. It can be done by data structure padding.

In the future, we plan to implement some other important routines from SPARSKIT, especially well-known solvers for sparse systems of linear equations. We also plan to implement multi-GPU support using OpenACC and OpenMP [31]. The full package with the software will soon be available for the community.

References

1. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue **6**, 40–53 (2008)
2. Leist, A., Playne, D.P., Hawick, K.A.: Exploiting graphical processing units for data-parallel scientific applications. Concurrency Comput. Pract. Experience **21**, 2400–2437 (2009)
3. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. J. Phys. Conf. Ser. **180**, 012037 (2009)
4. Nath, R., Tomov, S., Dongarra, J.: Accelerating GPU kernels for dense linear algebra. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 83–92. Springer, Heidelberg (2011)
5. Kowalik, J.S., Puzniakowski, T.: Using OpenCL - Programming Massively Parallel Computers. Advances in Parallel Computing, vol. 21. IOS Press, Amsterdam (2012)
6. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
7. Li, R., Saad, Y.: GPU-accelerated preconditioned iterative linear solvers. J. Supercomputing **63**, 443–466 (2013)
8. Helfenstein, R., Koko, J.: Parallel preconditioned conjugate gradient algorithm on GPU. J. Comput. Appl. Math. **236**, 3584–3590 (2012)
9. Feng, X., Jin, H., Zheng, R., Shao, Z., Zhu, L.: A segment-based sparse matrix-vector multiplication on CUDA. Concurrency Comput. Pract. Experience **26**, 271–286 (2014)

10. Pichel, J.C., Lorenzo, J.A., Rivera, F.F., Heras, D.B., Pena, T.F.: Using sampled information: is it enough for the sparse matrix-vector product locality optimization? Concurrency Comput. Practi. Experience **26**, 98–117 (2014)
11. Vázquez, F., López, G.O., Fernández, J., Garzón, E.M.: Improving the performance of the sparse matrix vector product with GPUs. In: 10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, 29 June-1 July 2010, pp. 1146–1151 (2010)
12. Williams, S., Oliker, L., Vuduc, R.W., Shalf, J., Yelick, K.A., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Parallel Comput. **35**, 178–194 (2009)
13. Matam, K.K., Kothapalli, K.: Accelerating sparse matrix vector multiplication in iterative methods using GPU. In: International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, 13–16 September 2011, pp. 612–621 (2011)
14. Bylina, B., Bylina, J., Stpiczyński, P., Szałkowski, D.: Performance analysis of multicore and multinodal implementation of SpMV operation. In: Proceedings of the Federated Conference on Computer Science and Information Systems, 7–10 September 2014, Warsaw, Poland, pp. 575–582. IEEE Computer Society Press (2014)
15. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers, San Francisco (2001)
16. Marowka, A.: Parallel computing on any desktop. Commun. ACM **50**, 74–78 (2007)
17. OpenACC: The OpenACC Application Programming Interface (2013). http://www.openacc.org
18. Sabne, A., Sakdhnagool, P., Lee, S., Vetter, J.S.: Evaluating performance portability of OpenACC. In: Brodman, J., Tu, P. (eds.) LCPC 2014. LNCS, vol. 8967, pp. 51–66. Springer, Heidelberg (2015)
19. Wang, C., Xu, R., Chandrasekaran, S., Chapman, B.M., Hernandez, O.R.: A validation testsuite for OpenACC 1.0. In: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, 19–23 May 2014, pp. 1407–1416 (2014)
20. Reyes, R., López-Rodríguez, I., Fumero, J.J., de Sande, F.: A preliminary evaluation of OpenACC implementations. J. Supercomputing **65**, 1063–1075 (2013)
21. Eberl, H.J., Sudarsan, R.: OpenACC parallelisation for diffusion problems, applied to temperature distribution on a honeycomb around the bee brood: a worked example using BiCGSTAB. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013, Part II. LNCS, vol. 8385, pp. 311–321. Springer, Heidelberg (2014)
22. Fegerlund, O.A., Kitayama, T., Hashimoto, G., Okuda, H.: Effect of GPU communication-hiding for SpMV using OpenACC. In: Proceedings of the 5th International Conference on Computational Methods (ICCM 2014) (2014)
23. NVIDIA: CUDA CUSPARSE Library. NVIDIA Corporation (2015). http://www.nvidia.com/
24. Grimes, R., Kincaid, D., Young, D.: ITPACK 2.0 users guide. Technical report CNA-150, Center for Numerical Analysis, University of Texas (1979)
25. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Basermann, A., Bishop, A.R.: Sparse matrix-vector multiplication on GPGPU clusters: a new storage format and a scalable implementation. In: 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDpPS 2012, Shanghai, China, 21–25 May 2012, pp. 1696–1702 (2012)

26. Wolfe, M.: Implementing the PGI accelerator model. In: Kaeli, D.R., Leeser, M. (eds.) Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGpPU 2010, Pittsburgh, Pennsylvania, USA, 14 March 2010. ACM International Conference Proceeding Series, vol. 425, pp. 43–50. ACM (2010)
27. Boisvert, R.F., Pozo, R., Remington, K.A., Barrett, R.F., Dongarra, J.: Matrix market: a web resource for test matrix collections. In: Boisvert, R.F. (ed.) Quality of Numerical Software - Assessment and Enhancement, Proceedings of the IFIP TC2/WG2.5 Working Conference on the Quality of Numerical Software, Assessment and Enhancement, Oxford, UK, 8–12 July 1996. IFIP Conference Proceedings, vol. 76, pp. 125–137. Chapman & Hall (1997)
28. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**, 1–25 (2011)
29. NVIDIA Corporation: CUDA Programming Guide. NVIDIA Corporation (2015). http://www.nvidia.com/
30. NVIDIA: CUDA C Best Practices Guide. NVIDIA Corporation (2015). http://www.nvidia.com/
31. Xu, R., Chandrasekaran, S., Chapman, B.M.: Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model. In: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, 20–24 May 2013, pp. 1169–1176 (2013)