J Supercomput
DOI 10.1007/s11227-014-1102-4

Performance evaluation of kernel fusion BLAS routines on the GPU: iterative solvers as case study

S. Tabik · G. Ortega · E. M. Garzón

© Springer Science+Business Media New York 2014

Abstract Programmers usually implement iterative methods that solve partial differential equations by expressing them using a sequence of basic kernels from libraries optimized for the graphics processing unit (GPU). The global runtime of the resulting combination is often penalized by the smallest and most inefficient vector operations. To improve the GPU exploitation, we identify and analyze the potential kernels to be fused according to the data dependence, data type and size, and GPU resources. This paper provides an extensive analysis of the impact of fusing vector operations [level 1 of Basic Linear Algebra Subprograms (BLAS)] on the performance of the GPU. The experimental evaluation shows that this optimization provides noticeable improvement especially for kernels with lower memory requirements and on more modern GPUs. It is worth noting that the fused BLAS operations can be very useful to help programmers efficiently code iterative methods to solve large linear systems of equations for the GPU. Iterative methods such as biconjugate gradient method (BCG)

The work was supported by the Spanish Ministry of Science and Innovation through projects TIN2010-16144, TIN2008-01117, TIN2012-37483-C03-03/01, Junta of Andalucía through project P10-TIC-6002 and CAPAP-H4 network (TIN2011-15734-E). G. Ortega is fellow of the Spanish FPU program. We thank NVIDIA for hardware donation under Professor Partnership 2008–2010, CUDA Teaching Center 2012–2013 and CUDA Research Center 2013 Awards.

S. Tabik
Department of Computer Architecture, University of Málaga, 29080 Málaga, Spain
e-mail: stabik@uma.es

G. Ortega (✉) · E. M. Garzón
Department of Computer Architecture and Electronics, University of Almería,
Agrifood Campus of Int. Excell., ceiA3, 04120 Almería, Spain
e-mail: gloriaortega@ual.es

E. M. Garzón
e-mail: gmartin@ual.es

Published online: 23 January 2014

are one of the examples that can benefit from this optimization strategy. Indeed, kernel fusion of vector routines makes the most efficient GPU implementation of BCG run between 1.09× and 1.27× faster on three GPUs of different characteristics.

Keywords Kernel fusion · Iterative solvers · BCG · BLAS 1 · GPU

1 Introduction

Most current computing systems from supercomputers to personal laptops include both multicore processors and accelerators such as graphics processing units (GPUs). Applications can take advantage of these heterogeneous resources by efficiently exploiting different levels of parallelism, i.e., the instruction-level parallelism (ILP), data-based parallelism and task-based parallelism. The performance improvement on these systems depends directly on the ability of the programmer to optimally exploit these three levels of parallelism.

Two main research directions have been taken to increase the programmer's productivity and enhance the performance of their applications. The first alternative consists of completely relying on the compiler to automatically generate optimized codes for the target architecture. Some relevant advances have been achieved in the last decades, but they showed to be applicable only for very simple loops. One of the examples in this context is the automatic parallelization of loop implemented under Intel C Compiler (icc). The second alternative to ease the development of parallel applications consists of using parallel libraries that efficiently implement the most frequent operations on a wide range of target high-performance architectures, so that programmers can express their codes in terms of a set of optimized routines without worrying about the specific details of each architecture. In this line, the performance of a given application does not depend only on the programmers' skill to appropriately express his code in terms of the available routines, but also on the quality of the design and implementation of the set of basic operations able to fully exploit the underlined architecture.

Different sets of routines have been developed in different fields. We focus the study provided in this work on numerical methods based on vector and matrix computations. The development of LINPACK library in the seventies was the starting point in this field [9]. Currently, Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) are the most extended libraries and used as baseline for the development of high-level libraries in this field.[1]

GPUs are becoming ubiquitous thanks to their high performance–cost ratio. They are characterized by having lots of cores, wide Single Instruction Multiple Threads (SIMT) units and several on-chip and off-chip memories. An efficient exploitation of all these resources represents a development effort. However, it is crucial for increasing the performance of scientific applications, particularly of those based on matrix operations [7].

Usually, when developing or porting a scientific code to the GPU, programmers use basic libraries such as NVIDIA CUDA Basic Linear Algebra Subroutines

[1] http://www.netlib.org/lapack/#_related_projects.

(CUBLAS[2]) [11] and NVIDIA CUDA Sparse Matrix library (CUSPARSE) [12], both implemented using NVIDIA CUDA functionalities. Indeed, programmers implement iterative methods for the GPU using these libraries as baseline, but in most cases the achieved performance is far from the optimum. Based on our experience, the use of several isolated kernels of different complexities degrades the performance on GPU architectures [13]. A typical set of applications where this degradation can be very important comprises the iterative methods that solve linear systems of equations since they include several vector–vector operations (level 1 BLAS) and a small number of sparse matrix–vector products (level 2 BLAS). This is mainly due to the fact that these operations are memory bound and have a very low arithmetic intensity which reduce their opportunities to obtain a balanced overlapping between memory accesses and computation [5]. Recently, some sparse matrix–vector product implementations are able to exploit the GPU architecture more efficiently [15]. Consequently, level 1 BLAS became the operations that dominate the overall runtime of iterative solvers in linear algebra. Our solution to this problem is to design new kernels by combining or fusing multiple BLAS operations to alleviate the memory bottleneck and increase the arithmetic intensity on the GPU.

This paper provides an experimental study of the impact of applying kernel fusion technique on multiple BLAS operations. The obtained results demonstrate the high interest of defining new BLAS kernels to improve programmers' productivity and enhance the performance of sparse solvers on the GPU. Moreover, the impact evaluation of using fused BLAS kernels in the well-known biconjugate gradient method (BCG) shows that the optimized BCG runs between $1.09\times$ and $1.27\times$ faster on Fermi and Kepler architectures, respectively.

The contributions of this work can be summarized as follows: (1) Analyzing and comparing the impact of fusing multiple BLAS operations on the performance of different GPU architectures and for all data types from single precision float to complex double. (2) Evaluating the use of fused BLAS kernels in iterative solvers; the BCG method has been selected as a representative example of these methods.

The rest of this document is organized as follows. Section 2 describes related works. Section 3 studies and analyzes the effect of kernel fusion on the performance of the GPU. Section 4 provides an experimental evaluation of using new BLAS kernels in the BCG solver and finally Sect. 5 gives conclusions and future works.

2 Related works

The optimization technique of fusing, merging or coalescing several code segments, e.g., loop fusion, was shown to be very useful for a number of specific applications on specific architectures. For instance, coalescing multiple functions that work on the same data tile in the bulge chasing algorithm enhances cache memory reuse and reduces the overheads due to the used runtime system on multicore processors [6]. Merging a number of pipeline stages in streaming applications improves their locality and reduces contention in the work queues on multicore systems [10]. In [17], the

[2] http://docs.nvidia.com/cuda/cublas/index.html.

authors state that using both fusion and fission, i.e., splitting up kernels, its orthogonal analog, for a number of kernels in data warehousing applications also enhances the performance on GPUs, especially for large volumes of data. In [1], the authors showed that reformulating a seven-kernel preconditioned conjugate gradient method into the four-kernel version that fuses multiple kernels of different types improves the performance on a GPU. However, they did not analyze under which specific conditions this was true. The most related work to ours is described in [2], which explores the specific fusion of heterogeneous sequences of BLAS routines that share at least one array.

Differently to all previous works, the present work provides a complete analysis of the possibilities and limitations of the kernel fusion technique for BLAS routines, which are used in a much broader field of applications. Given the importance of BLAS library in scientific, engineering and commercial applications together with the extensive use of GPUs, this work focuses on defining new useful BLAS kernels that merge multiple individual BLAS kernels of the same type to further improve the performance of iterative methods on the GPU.

3 Impact of fusing BLAS operations on GPU performance

Fusing two or more kernels consists of merging all the kernels together into one single kernel. This section analyzes the impact of fusing multiple operations of Level 1 BLAS of the same type, namely saxpy and dot, which are the most frequently used operations to solve large linear systems of equations, on the GPU performance. Figure 1 illustrates how the kernel fusion optimization is applied to multiple saxpy operations. The parameter *FL* refers to the fusion level. The fusion transformation has advantages or disadvantages depending on the volume and type of the involved data, volume of the arithmetic operations and synchronization points. All the possible advantages can be summarized as follows:

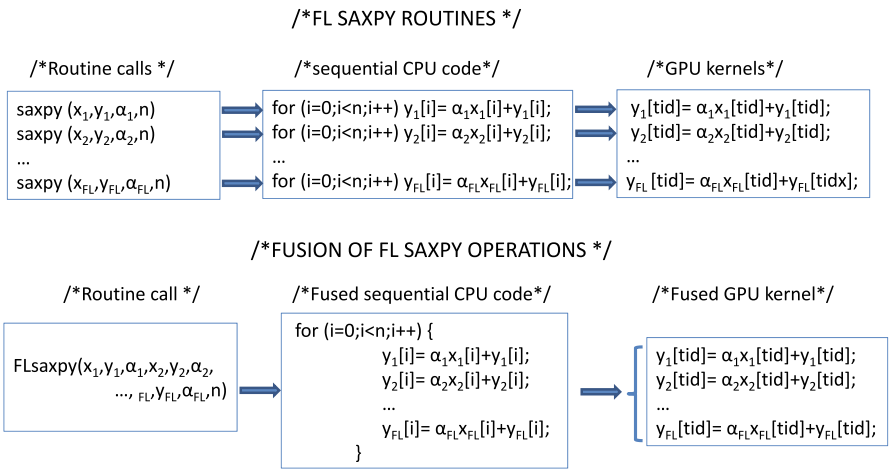

Fig. 1 An example of fusing FL simple saxpy operations into one single kernel for the CPU and GPU. The index *tid* refers to thread index

(a) *Reduces the number of global barriers*: Merging two or more kernels that include explicit barriers, e.g., dot kernels, allows eliminating a number of these synchronization points.
(b) *Reduces GPU global memory accesses*: Fusing multiple kernels may reduce data movements between on-chip and off-chip memories of the GPU. That is, fused kernels store intermediate data in on-chip memories (i.e., registers and shared memory) of the GPU, which can be accessed much faster than global memory.
(c) *Enhances locality at GPU memories level*: This benefit occurs only when one or more arrays are shared by multiple fused kernels. This fact eliminates multiple accesses to the same data.
(d) *Improves ILP and increases possibilities for compiler optimizations*: Fusing multiple kernels increases the volume of the work unit carried out by each thread. In addition, a kernel of larger scope gives more opportunities for optimization to the compiler.
(e) *Reduces the number of data movements between CPU and GPU memories*: Fusing multiple kernels may reduce the number of movements of intermediate data through the Peripheral Component Interconnect express (PCIe). For example, merging multiple dot operations allows copying the final partial sums from GPU to CPU memories in one movement, whereas non-fused dot kernels make as many movements as the number of dot kernels.
(f) *Increases concurrency*: Increasing the number of independent instructions by threads increases the workload volume per thread and consequently improves the overall management of the threads by the scheduler which can be translated into an enhancement of the concurrency, i.e., a higher number of active threads per active cycle.

However, it is necessary to underline that the fusion has also a disadvantage. The fact that threads have to access a larger volume of data located in different address ranges may penalize the locality of memory accesses. This means that operations with higher memory requirements may take less advantage of the fusion. Next, two representative operations of level 1 of BLAS (saxpy and dot) are considered to evaluate the performance of fused BLAS kernels for different kinds of data (float, double, complex and complex double). This analysis will allow identifying the factors that make the fusion optimization interesting to improve the runtime on different GPU cards.

3.1 Evaluation of fusing BLAS kernels

This section provides the experimental evaluation of fusing multiple BLAS operations of the same type, saxpy and dot, for all data types and for different data sizes, on three GPUs. We have explored only realistic ranges of fusion level (FL) which are $FL = 1, 2, 3$ for saxpy and $FL = 1, 2$ for dot, since in iterative methods the possible FL of BLAS operations is limited by data dependence constraints. We used different vectors in each operation to discard the benefits from data reutilization. We did not include the evaluation of fusing multiple SpMV operations because of space restrictions and because it does not provide noticeable enhancement.

Table 1 Brief architectural summary of the GPUs used in the experiments

	GeForce GTX 480	Tesla C2050	Tesla K20C
Peak GFlop/s (single precision)	1345	1030	3520
Bandwidth (GB/s)	177.4	144	208
# SM(X)s	15	14	13
# Cores/SM(X)	32	32	192
Clock speed (GHz)	1.00	1.15	0.71
shmem/block (Kb)	48	48	48
# Reg/block	32768	32768	65536
Global memory (Gb)	1.5	2.62	4.68

Error-Checking Codes (ECC) is disabled in all the experiments

Table 2 shows the speedup of the fused kernel with respect to the kernels without fusion calculated as $T_{without\ fusion}/T_{with\ fusion}$, achieved when fusing 2 and 3 saxpy operations (see $2saxpy$ and $3saxpy$ columns) and 2 dot operations (see $2dots$ column), for all data types, float (f), double (d), complex float (cf) and complex double (cd), on three GPUs, GTX 480, Tesla C2050 and K20C (whose main characteristics are shown in Table 1) and considering four array dimensions, $n = 1.024 \times (10^4, 10^5, 10^6$ and $10^7)$. We used these specific vector sizes to avoid load unbalance issues and to ensure coalescence of memory accesses requested by each thread block. These routines have been compiled with NVIDIA CUDA C 5.0 and gcc compilers with -O2 as optimization option.

In general, from columns $2saxpy$ in Table 2, one can see that fusing 2 saxpy operations always outperforms the non-fused kernels for array sizes of the order of 1.024×10^4 on all the used GPUs. In particular, fusing two saxpy kernels on Kepler reduces the runtime for all the data sizes and types. In fact, it achieves speedups of up to $53\times$ faster and up to $17\times$ faster for array dimensions of the order of 10^4 and 10^5, respectively. One can also notice that on all the GPUs the enhancement of the runtime is higher for float data types and lower for complex double. Fusing is interesting for larger arrays of type complex only on K20C. The benefit from fusing 2 saxpy is higher for GPUs with more resources, in general. On the other hand, fusing 3 saxpy operations reaches better enhancement for array dimensions smaller than 10^5. In general, this improvement is lower than when fusing 2 saxpy. Similarly to fusing 2 saxpy, fusing 3 saxpy becomes more interesting for the most modern GPUs. In conclusion, fusing multiple saxpy operations, which implies an increase of the workload unit volume carried out by each thread, is beneficial on the GPU for data sizes and types that do not penalize the memory management.

Fusing two dot kernels as shown in Table 2, columns $2dots$ is always beneficial to the performance for all the used array dimensions, from sizes of the order of 10^4 elements to 10^7 elements. This improvement is due to the fact that fusing 2 dot operations not only increases the number of arithmetic operations carried out per thread, but also reduces the number of global synchronizations and the number of data movements from GPU to CPU (i.e., only one small array is sent back to the host). Notice that

Table 2 Runtime improvement ($T_{without\ fusion}/T_{with\ fusion}$) when fusing 2 saxpy, 3 saxpy and 2 dots, for all data types, float (f), double (d), complex float (cf) and complex double (cd), for four array dimension, $n = 1.024 \times (10^4, 10^5, 10^6$ and $10^7)$, on GTX 480, Tesla C2050 and Tesla K20C

	$2saxpy$			$3saxpy$			$2dots$		
	GTX 480	C2050	K20C	GTX 480	C2050	K20C	GTX 480	C2050	K20C
1.024×10^4									
f	3.53	6.54	30.12	2.03	2.12	2.47	2.51	2.42	17.76
d	4.78	4.83	47.71	2.25	2.12	2.40	2.17	2.12	17.01
cf	4.38	4.15	38.67	1.89	1.88	2.02	2.02	2.17	9.86
cd	2.09	1.98	53.05	1.62	1.53	1.48	1.77	1.75	9.05
1.024×10^5									
f	1.35	1.53	17.19	1.42	1.36	1.65	1.58	1.59	8.13
d	1.37	1.24	11.15	1.24	1.13	1.36	1.49	1.41	6.64
cf	1.34	1.23	9.57	1.21	1.13	1.31	1.53	1.67	9.95
cd	1.05	0.88	6.11	0.96	0.83	1.09	1.60	1.50	7.59
1.024×10^6									
f	1.06	1.03	4.05	1.05	0.97	1.09	1.31	1.29	5.50
d	1.03	0.98	2.55	1.00	0.92	1.01	1.20	1.15	2.64
cf	0.99	0.95	2.39	0.96	0.92	0.98	1.20	1.50	3.90
cd	0.87	0.81	2.46	0.81	0.74	0.96	1.27	1.38	1.72
1.024×10^7									
f	1.02	0.98	1.30	1.00	0.94	1.00	1.12	1.11	1.41
d	0.99	0.95	1.13	0.97	0.89	0.96	1.12	1.09	1.20
cf	0.95	0.92	1.10	0.82	0.88	0.92	1.14	1.38	1.31
cd	0.85	0.80	1.01	0.80	0.73	0.94	1.17	1.35	1.03

fusing 2 dot kernels achieves better performance than saxpy, because saxpy has more accesses to global memory (reads X and Y and updates Y).

4 Evaluation of using fused BLAS kernels in iterative solvers: BCG

Commonly, implementing iterative solvers on GPUs consists of expressing the main iteration of the method using a series of elemental kernels, namely multiple saxpy operations, dot operations and matrix–vector product in a natural order according to the given data dependencies.

Generally, these methods perform at least four data movements between CPU and GPU. The first and the last data movements are needed for the initialization phase before the starting of the method and at the end of the iterative process. In each single iteration, two lighter data movements are needed to check the stop criteria and verify whether the system has already converged to the right solution.

Iterative solvers are expressed as a sequence of vector and matrix operations which are performed in a specific order. In most cases, a number of operations of the same

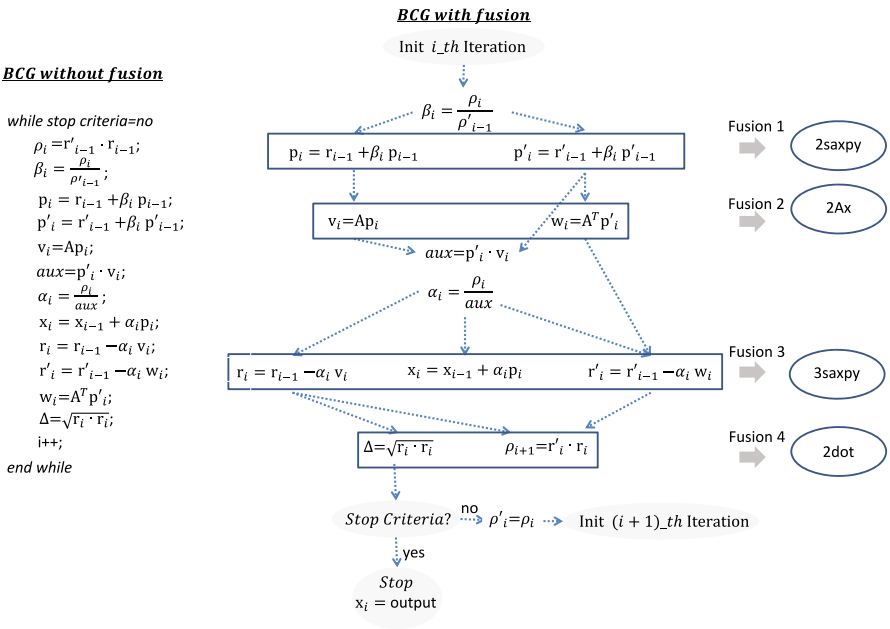

Fig. 2 Candidate BLAS kernels for fusion in the BCG method. Applying the kernel fusion optimization to BCG consists of performing Fusion 1–4 that correspond to fusing two saxpy operations, two matrix vector products, three saxpy operations and two dot operations, respectively

type are not dependent at all and can be merged together into one single routine. Therefore, a simple preprocessing analysis is needed to reorder independent BLAS operations, so that they can benefit from kernel fusion optimization evaluated in this paper.

After an extended analysis of a large set of sparse solvers, e.g., CG, BCG and all their variants, we found that there is a number of vector and matrix operations that occur frequently in this kind of method. In particular, multiple saxpy operations, multiple dot operations and multiple matrix vector operations are usually performed in a specific order and can be merged together into one larger kernel.

In this work we use the BCG, a representative sample in linear algebra, to evaluate the benefits of the kernel fusion optimization on GPUs. The BCG method is a non-stationary iterative method to solve linear systems of equations $Ax = b$, where the matrix $A \in C^{n \times n}$ can be non-symmetric [8]. A denotes the sparse matrix, b indicates the independent term and x is the unknown vector. Let us focus on the particular case of the BCG where $A \in R^{n \times n} \subset C^{n \times n}$ and $x, b \in R^n$.

As can be observed in Fig. 2, in each iteration of the method, eight vector–vector operations (three dot operations and five saxpy operations) and two SpMVs are computed. The two SpMV operations are computed with the matrices A and A^T. Note that the SpMV for the transpose matrix (A^T) consumes more runtime due to the penalties related to the locality loss in the access to the elements of A^T. To overcome these penalties, our approach is to store both, A and A^T, as two different sparse matrices. The computational cost associated with SpMV products increases with respect to the

set of vector–vector operations as the ratio nz/n increases, where nz and n denote the number of non-zero elements and the number of rows and columns of A, respectively.

The potential candidate vector–vector operations that can benefit from kernel fusion in BCG taking into account the data dependencies in the algorithm are shown in Fig. 2. This figure also represents the data dependence graph of the main iteration of BCG after a reordering process, where the candidate BLAS kernels for fusion are shown in rectangles. The BLAS operations that can be fused in BCG are:

- two saxpy operations,
- two matrix products,
- two dot operations,
- three saxpy operations.

Experimentally, using several matrices from Matrix Market for the fusion of two matrix products, we found that this fusion does not provide noticeable improvement on the performance especially when there is no shared vector between the SpMV kernels.

Each iteration of BCG is expressed using a set of four fused operations. It is worth noting that the analysis described to fuse baseline operations in BCG illustrates a simple methodology which can be applied to other iterative methods. It is also relevant to underline that there are several GPU implementations of BCG [3,4,16]. To evaluate kernel fusion in this work we used the fastest BCG implementation over all the existing GPU implementations. In particular, the used non-fused BCG version utilizes CUBLAS routines for saxpy and dot operations and the ELLR-T kernel for SpMV [14,15], where ELLR-T is one of the most efficient Sparse matrix–vector products, outperforming the one supplied by NVIDIA CUSPARSE [12].

4.1 Experimental study

In this section, we show and analyze the performance results of using the four new fused kernels in the most efficient GPU implementation of BCG method (described in the previous paragraph). These fused kernels are enumerated as Fusion 1–4 in Fig. 2. Fusion 1, encapsulates two saxpy kernels; Fusion 2 merges two sparse matrix vector products, Fusion 3 fuses three saxpy operations and Fusion 4 fuses two dot operations. We evaluated the BCG method with and without using the four fused kernels for float data types on three GPUs, GTX 480, C2050 and K20C (whose characteristics are summarized in Table 1) considering five different matrices of different characteristics as shown in Table 3.

Figure 3a expresses the performance in terms of GFlop/s (calculated as $(4nz+16n)\cdot total_num_iter/time$) of the optimized BCG, with fusion, versus the non-optimized version, without fusion. In addition, Fig. 3b quantifies the speedup when using the four fused kernels in BCG with respect to the BCG code that uses CUBLAS for vector operations and ELLR-T kernel for SpMV.

As it can be seen from these figures, using the four new kernels improves the GFlops of BCG on all the considered GPUs and matrices. The overall runtimes are from 1.09× to 1.27× faster on both Fermi and Kepler architectures. These results are in accordance with the results obtained in the previous section. Kernel fusion of

Table 3 Characteristics of the sparse matrices used to evaluate the new fused kernels in BCG, where n is the number of rows, nz is the total number of nonzero elements Av is the average number of entries per row

Matrix	n (10^3)	nz (10^6)	Av
pdb1HYS	36.4	4.3	119.3
cant	62.4	4.1	64.2
consph	83.3	6.0	72.1
cop20k_A	121.2	2.6	21.7
shipsec1	140.8	3.6	25.3

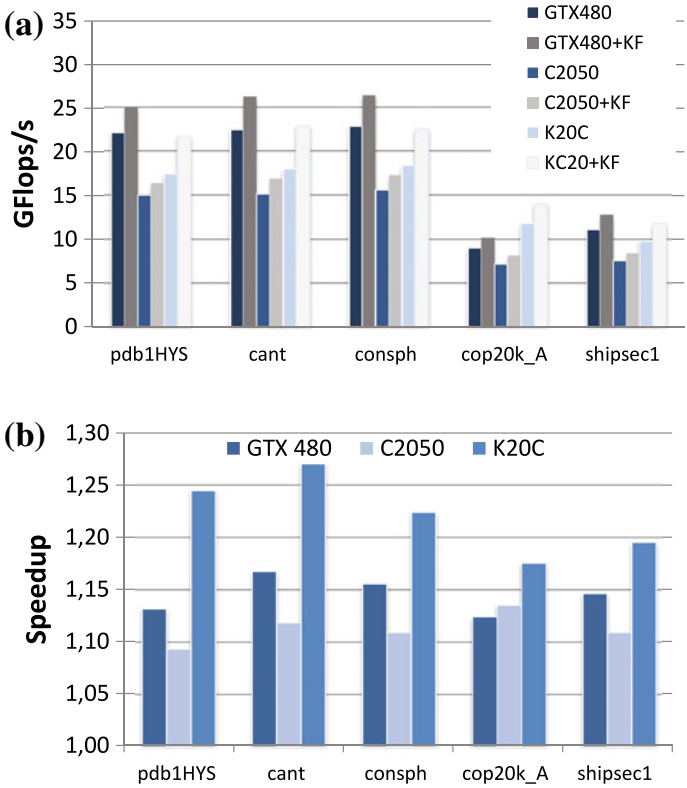

Fig. 3 **a** The GFlop/s of BCG with kernel fusion technique (+KF) and without kernel fusion on three GPUs, GTX 480, C2050 and K20C for five matrices from the matrix market. **b** The runtime improvement, calculated as $T_{without\ fusion}/T_{with\ fusion}$, for the BCG when using our fused kernels with respect to the version based on CUBLAS kernels and a very fast SpMV-GPU implementation

two successive saxpies and two successive dots improves the GPU performance of iterative methods for problem sizes of the order of 10^6.

5 Conclusions

This paper has shown the importance of defining new BLAS kernels that fuse multiple frequent level 1 BLAS operations to code more efficient iterative methods on GPUs. The new defined kernels are based on the fusion optimization technique. The experi-

mental evaluation showed that fusing vector operations (level 1 of BLAS) on the GPU provides noticeable enhancement, up to 53× faster, for kernels with lower memory requirements on GPU with more resources. Moreover, this optimization enhances the overall performance of the BCG method, a representative sample in linear algebra, to up to 1.27× using reasonable problem sizes. The results obtained in this work validates the importance of extending the CUBLAS library with new versions of multiple fused vector operations for the GPU.

The fused BLAS kernels for dots and saxpy analyzed in this work are freely available through the following web site: https://sites.google.com/site/mcfastsparse/.

References

1. Dehnavi MM, Fernandez DM, Giannacopoulos D (2011) Enhancing the performance of conjugate gradient solvers on graphic processing units. IEEE Trans Magn 47(5):1162–1165
2. Filipovič J, Madzin M, Fousek J, Matyska L (2013) Optimizing cuda code by kernel fusion—application on BLAS. CoRR abs/1305.1183
3. Gaikwad A, Toke IM (2010) Parallel iterative linear solvers on GPU: a financial engineering case. In: Proceediongs of PDP, pp 607–614
4. Garcia N (2010) Parallel power flow solutions using a biconjugate gradient algorithm and a newton method: a GPU-based approach. In: IEEE Power and Energy Society general meeting, pp 1–4
5. Golub GH, van Van Loan CF (1996) Matrix computations (Johns Hopkins studies in mathematical sciences), 3rd edn. The Johns Hopkins University Press. Baltimore, MD
6. Haidar A, Ltaief H, Luszczek P, Dongarra J (2012) A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In: Proceedings of of IEEE IPDPS, pp 25–35
7. Hwu W (2011) Computing Gems Jade Edition. Applications of GPU computing series, Jade edn. Elsevier Science, Amsterdam
8. Lanczos C (1952) Solution of systems of linear equations by minimized iterations. J Res Natl Bur Stand 49:33–53
9. Lawson CL, Hanson RJ, Kincaid DR, Krogh FT (1979) Basic linear algebra subprograms for fortran usage. ACM Trans Math Softw 5(3):308–323
10. Navarro AG, Asenjo R, Tabik S, Cascaval C (2009) Analytical modeling of pipeline parallelism. In: Proceedings of PACT, pp 281–290. IEEE Computer Society
11. NVIDIA (2013) Du-06702-001_v5.5 CUBLAS user guide. Technical report. http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf
12. NVIDIA (2013) Du-06709-001_v5.5 CUSPARSE library. Technical report. http://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf
13. Ortega G, Garzón EM, Vázquez F, García I (2013) The biconjugate gradient method on GPUs. J Supercomput 64:49–58
14. Vázquez F, Fernández JJ, Garzón EM (2012) Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach. Parallel Comput 38:408–420
15. Vázquez F, Ortega G, Fernández JJ, Garzón EM (2010) Improving the performance of the sparse matrix vector product with GPUs. In: Proceedings of IEEE CIT, pp 1146–1151. IEEE Computer Society
16. Wozniak M, Olas T, Wyrzykowski R (2010) Parallel implementation of conjugate gradient method on graphics processors. In: Parallel processing and applied mathematics, LNCS vol 6067, pp 125–135
17. Wu H, Diamos G, Wang J, Cadambi S, Yalamanchili S, Chakradhar S (2012) Optimizing data warehousing applications for GPUs using kernel fusion/fission. In: Proceedings of IEEE IPDPSW, pp 2433–2442