2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Optimizing Krylov Subspace Solvers on Graphics Processing Units

Hartwig Anzt, Stanimire Tomov, Piotr Luszczek,
Ichitaro Yamazaki, Jack Dongarra
Innovative Computing Lab,
University of Tennessee, Knoxville, USA.
Emails: {hanzt, tomov, luszczek, iyamazak, dongarra}@icl.utk.edu

William Sawyer
Swiss National Supercomputing Centre (CSCS),
Lugano, Switzerland
Email: william.sawyer@cscs.ch

***Abstract*—Krylov subspace solvers are often the method of choice when solving sparse linear systems iteratively. At the same time, hardware accelerators such as graphics processing units (GPUs) continue to offer significant floating point performance gains for matrix and vector computations through easy-to-use libraries of computational kernels. However, as these libraries are usually composed of a well optimized but limited set of linear algebra operations, applications that use them often fail to leverage the full potential of the accelerator. In this paper we target the acceleration of the BiCGSTAB solver for GPUs, showing that significant improvement can be achieved by reformulating the method and developing application-specific kernels instead of using the generic CUBLAS library provided by NVIDIA. We propose an implementation that benefits from a significantly reduced number of kernel launches and GPU-host communication events, by means of increased data locality and a simultaneous reduction of multiple scalar products. Using experimental data, we show that, depending on the dominance of the untouched sparse matrix vector products, significant performance improvements can be achieved compared to a reference implementation based on the CUBLAS library. We feel that such optimizations are crucial for the subsequent development of high-level sparse linear algebra libraries.**

I. Introduction

Modern research and development is often driven by virtual simulations on computer systems. Partial differential equations are utilized to generate a virtual model of the physical reality and their discretizations are used to obtain linear systems of equations that can then be solved numerically. Depending on the problem and the discretization method, the obtained matrices are often sparse and of large size, which makes iterative solvers attractive for their solution. Among the most efficient iterative methods are Krylov subspace solvers that search for an approximate solution in a subspace. Besides the well-known generalized minimum residual method (GMRES), the BiCG algorithm is another Krylov subspace solver able to handle non-symmetric indefinite systems. To improve the stability and convergence of the original algorithm, H. A. van der Vorst developed a stabilized version called biconjugate gradient stabilized method, often abbreviated as BiCGSTAB [15]. Depending on the problem characteristics, BiCGSTAB can outperform the GMRES method and be the method of choice when solving a system of linear equations.

The latest hardware developments promote the use of accelerators when solving large linear systems. A straight forward way to use accelerators in Krylov subspace solvers is to offload all matrix and vector computations to the device using library functions. These steps were taken in an implementation of the BiCGSTAB method in a tutorial for educational purposes that can be found at hpcforge.org [14], where CUBLAS functions provided by NVIDIA are used to handle the matrix and vector operations. Despite the appealing performance improvements compared to CPU implementations, we argue that these steps are not sufficient to leverage the full performance potential, but it is necessary to reformulate the algorithm and implement method-specific kernels to achieve higher performance on GPU-accelerated systems. Prior work [1] took a similar approach to improve runtime and energy footprint of a Conjugate Gradient method. This paper goes further by describing the reformulation of a Krylov subspace solver using techniques such as aggregation of multiple arithmetic operations into a single kernel to reduce GPU memory traffic and CPU-GPU communication. Moreover, simultaneous computation of multiple dot products is introduced and a model is provided to estimate the runtime improvements achieved by the modifications. The BiCGSTAB method is chosen as representative Krylov method because it strikes a balance between different execution patterns, that are typical for this class of solvers. The patterns include consecutive and isolated dot products, vector updates, and matrix vector multiplications. The rest of the paper is structured as follows: we begin in Section II with a brief review of the BiCGSTAB algorithm, and discuss the importance of the sparse matrix vector product in Krylov subspace methods in Section III. The core of the paper (Section IV) is the redesign of the BiCGSTAB algorithm, where we derive application-specific kernels by merging multiple arithmetic operations and we introduce a kernel capable of computing multiple dot products simultaneously. We also address the topic of data locality, GPU-host communication, and asynchronous stopping check. Based on these modifications, in Section V we derive a model quantifying the expected performance improvements and use experimental results to validate the model, which show the superior performance of the reformulated BiCGSTAB algorithm.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.107

1: $x_0 := 0$ or any other initial guess
2: $r_0 := b - Ax_0$
3: $\hat{r}_0 := r_0$ or any other initial guess s.t. $\hat{r}_0^{\mathsf{T}} r_0 \neq 0$
4: $\rho_0 := \omega_0 := \alpha_0 := 0$
5: $v_0 := 0\ p_0 := 0$
6: $k := 1$
7: **while** $(k < maxiter)$ && $(res > \tau)$
8: $k := k + 1$
9: $\rho_k := \hat{r}_0^{\mathsf{T}} r_{k-1}$ (dot)
10: $\beta_{k+1} := \frac{\rho_k}{\rho_{k-1}} \frac{\alpha_{k-1}}{\omega_{k-1}}$
11: $p_k := r_{k-1} + \beta\,(p_{k-1} - \omega_{k-1} v_{k-1})$ (scal + 2 axpy)
12: $v_k := A p_k$ (SpMV)
13: $\alpha_k := \frac{\rho_k}{\hat{r}_0^{\mathsf{T}} v}$ (dot)
14: $s_k := r_{k-1} - \alpha v_k$ (copy + axpy)
15: $t_k := A s_k$ (SpMV)
16: $\omega_k := \frac{s_k^{\mathsf{T}} t_k}{t_k^{\mathsf{T}} t_k}$ (2 dot)
17: $x_{k+1} := x_k + \alpha_k p_k + \omega_k s_k$ (2 axpy)
18: $r_k := s_k - \omega t_k$ (copy + axpy)
19: $res = r_k^{\mathsf{T}} r_k$ (dot)
20: **end**

Fig. 1. Algorithmic description of the BiCGSTAB method [2].

II. Biconjugate gradient stabilized method

The BiCGSTAB method was developed by H. A. van der Vorst with the objective to improve stability and convergence of the BiCG method [15]. It belongs to the class of Krylov subspace solvers and can be used to solve linear systems of equations that are not necessarily symmetric and positive definite [13], unlike the CG method. BICGSTAB's usually fast convergence makes it an attractive candidate when targeting the numerical solution of partial differential equations via finite element or finite difference methods [4].

For the linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$ is the sought-after solution, we outline the BiCGSTAB method in Figure 1 where τ and *maxiter* set upper bounds, respectively, on the relative residual for the computed approximation to the solution x_k, and the maximum number of iterations. Beside the two sparse matrix vector multiplications (line 12 and 15), usually dominating the computational effort, every BiCGSTAB iteration contains several vector operations such as copy, axpy or dot product. The cost for one iteration can be estimated by $4nnz + 18n$ where nnz is the number of nonzero entries in A and n is the number of unknowns.

An implementation of BiCGSTAB for GPU-accelerated platforms [14] was originally drafted as an example for a course on GPU-enabled libraries. In that implementation, all matrix and vector operations are handled by the accelerator using NVIDIA's CUBLAS library. The essential operations are given in Figure 2. Although the intention is to provide an example of how the CUBLAS library can be employed instead of providing highly tuned software, it shares the principles of most implementations for GPU-accelerated systems, and we will therefore take it as a reference CUBLAS implementation.

```
1  while( ( k < maxiter ) && ( res > epsilon ) ){
2      rho_new = cublas_dot( n, r_hat, 1, r, 1 );
3      beta = rho_new/rho_old * alpha/omega;
4      cublasDscal( n, beta, p, 1 );
5      cublasDaxpy( n, omega * beta, v, 1 , p, 1 );
6      cublasDaxpy( n, 1.0, r, 1, p, 1 );
7      dSpMV <<<Gs,Bs>>> ( n, rowA, colA, valA, p, v );
8      alpha = rho_new / cublasDdot( n, r_hat, 1, v, 1 );
9      cublasDcopy( n, r, 1 , s, 1 );
10     cublasDaxpy( n, -1.0 * alpha, v, 1 , s, 1 );
11     dSpMV <<<Gs,Bs>>> ( n, rowA, colA, valA, s, t );
12     omega = cublasDdot(n, t,1, s,1) / cublasDdot(n, t,1, t,1);
13     cublasDaxpy( dofs, alpha, p, 1 , x, 1 );
14     cublasDaxpy( dofs, omega, s, 1 , x, 1 );
15     cublasDcopy( n, s, 1 , r, 1 );
16     cublasDaxpy( n, -1.0 * omega, t, 1 , r, 1 );
17     res = cublasDnrm2( n, r, 1 );
18     rho_old = rho_new;
19     k++;
20 }
```

Fig. 2. CUBLAS implementation of the BiCGSTAB algorithm from Figure 1.

III. Sparse matrix vector product

In the iterative solution of sparse linear systems via Krylov methods, the matrix vector product generating the subspace is usually dominating the overall computational cost of every iteration. Hence, significant effort is spent on developing storage formats along with kernels that are suitable for efficient execution on the target architecture. While the widespread compressed sparse row (CSR [2]) format usually provides good performance on CPU architectures, the use of less compact formats like ELLPACK or packet format [3] may be beneficial when targeting accelerators like GPUs using streams for the execution [10]. The underlying reason is that introducing (storage and computational) overhead may allow for coalesced memory access which is key for leveraging performance on streaming processors. In addition to formats for specific matrix patterns, hybrid storage formats have also been developed, balancing fill for superior memory access patterns and computational overhead [3]. However, it is in general difficult to determine the optimal storage format a priori, and without knowledge about the matrix characteristics, only statistical information can be used. An extensive comparison between different storage formats can be found in [3]. As the focus of this paper is on the reformulation of the BiCGStab algorithm, we refrain from optimizing the sparse matrix vector kernel used in the algorithm, but stick to the CSR format. The motivation is that we want to quantify the effect of various optimization techniques and show how switching from the straight forward implementation to a reformulated version based on method-specific kernels improves the performance, while changes to the used matrix format and respective kernels would not be considered as algorithmic changes and sophisticate a fair comparison.

IV. Reformulation of the BiCGSTAB algorithm

The reference implementation of BiCGSTAB using CUBLAS functions as presented previously yields appealing performance improvement compared to CPU code, but it also misses some performance improvement opportunities. For example, a better resource utilization can be achieved by

designing application-specific routines, reducing the number of kernel calls, reducing the GPU-host communications, and applying power-saving mechanisms featured by the hardware [1]. To this end, a reformulation of the algorithm in Figure 1 is inevitable. Gathering similar operations (e.g., component-wise vector operations, dot products, and scalar operations) allows the programmer to design algorithm-specific kernels with higher computational intensity than the replaced CUBLAS functions. Merging several arithmetic operations into one kernel reduces the amount of kernel calls and memory access, and handling the residual stopping criterion asynchronously to the iteration process enables a better GPU utilization. While Figure 3 provides a general overview about the original CUBLAS reference implementation and the new implementation featuring these improvements, we discuss the distinct modifications we propose to the classical formulation of the algorithm in the following sections.

A. Experimental Setup

Our experiments were performed on a Tesla K20c GPU that belongs to the Kepler line of NVIDIA's hardware accelerators. The GPU consists of 2,496 CUDA cores @705 MHz, grouped in 13 Streaming Multiprocessors (SXM) with 192 cores per SXM. It provides 3,520 Gflop/s in single precision and 1,170 Gflop/s in double precision. The main memory is 5 GB of GDDR5 and has a peak bandwidth of 208 GB/s, which is sufficiently large to keep all the matrices and all the vectors needed in the iteration process. We limit our analysis to double precision, and to ensure the accuracy of the data, we usually run every experiment 1,000 times and either average the values or report the total time.

The host processor was an Intel Xeon E5 (codename: Sandy Bridge, model `0x2D`, family `0x06`) in a two-socket configuration featuring 8 cores in each socket with HyperThreading enabled and the nominal frequency was 2.6 GHz.

B. Merging multiple arithmetic operations into one kernel

When implementing complex arithmetic operations using only CUBLAS functions, there are often several of the CUBLAS calls that have to be combined together and the set of calls to choose from is very limited. In fact, the entire set attempts to establish close correspondence with the Basic Linear Algebra Subprograms (BLAS) [11], [9], [8], [6], [7], a software interface originally designed for linear algebra operations on the CPU. But while on a CPU-based system the memory hierarchy usually offers sufficient performance when executing a series of these functions, this is far from true on the GPU. The GPU architecture provides small caches that are used by the CUBLAS routines, but consecutive CUBLAS operations would not keep reusable data locally. Also, the data streaming scheme, the key to achieving high throughput and GPU performance, suffers when multiple individual CUBLAS functions are invoked when the compiler fails to detect dependencies. An example is the computation $p_k := r_{k-1} + \beta\,(p_{k-1} - \omega_{k-1} v_{k-1})$ in line 11 of Figure 1 that results in three CUBLAS function calls (line 4-6 in Figure 2).

```
1 __global__ void
2 magma_dbicgmerge_p_update( int n,
3                            double *skp, double *v,
4                            double *r,   double *p ){
5     int i = blockIdx.x * blockDim.x + threadIdx.x;
6     double beta=skp[1], omega=skp[2];
7     if( i<n )
8         p[i] = r[i] + beta * ( p[i]-omega*v[i] );
9 }
```

Fig. 4. Algorithm-specific kernel for the operation in line 11 in Figure 1.

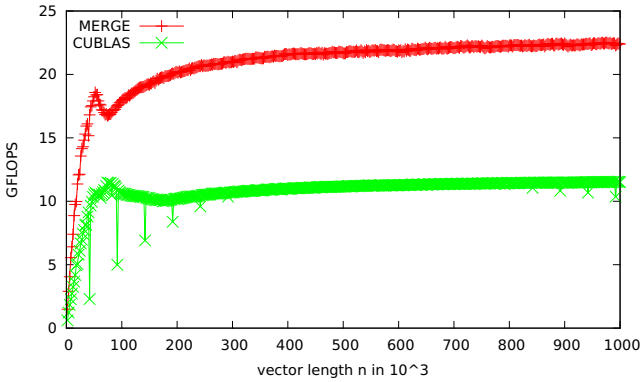

Fig. 5. Performance comparison between the merged BiCGSTAB and the reference implementation for the update of vector p (see line 11 of Figure 1).

Every function reads the data from main memory, processes the operation, and writes data back. As vector operations (Level 1 BLAS) are bandwidth-bound, the $8n+4$ memory transfers ($5n+4$ reads and $3n$ writes) limit the performance. Significant improvements can be achieved by replacing the CUBLAS functions with the kernel given in Figure 4, where data movement is reduced down to $3n+4$ reads and n writes. Due to the 50% memory traffic reduction, we expect an asymptotic speedup of 2, which is reflected in Figure 5 where the update of p using CUBLAS functions reaches, for large vector sizes, only 11.5 Gflop/s compared to 22.4 Gflop/s of the `magma_dbicgmerge_p_update` kernel given in Figure 4.

Comparing the total GPU memory access in one BiCGSTAB iteration, we conclude from Table I that the savings in global memory reads are $13n$ reads and $5n$ writes. Neglecting the sparse matrix vector multiplications ($4nnz+6n$ memory transfers in total), we remain with $34n$ memory transfers in the CUBLAS reference implementation. In the accelerated BiCGSTAB, we have merged the arithmetically untouched matrix vector products with other vector operations. Hence, as the vectors have to be transferred anyway, we remain with $22n$ memory transactions when omitting the impact of the matrix vector products. Depending on the dominance of the aforementioned matrix vector product, we expect a performance improvement of up to 35%.

C. Reduce GPU-host communication

Merging multiple arithmetic operations into a single GPU kernel not only improves the computational intensity, but also

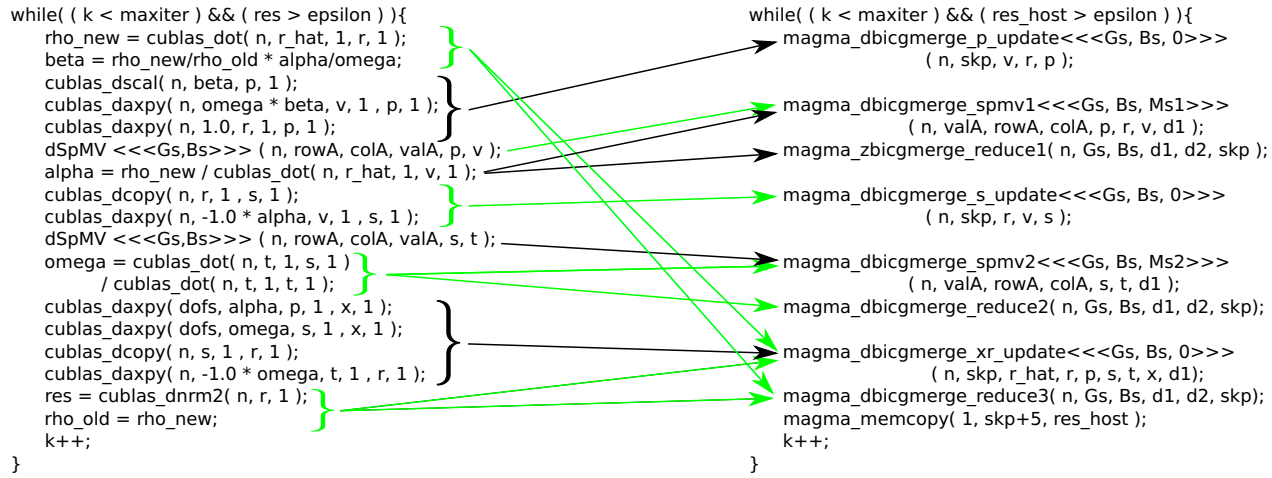

Fig. 3. Visualizing the reformulation of the reference BiCGSTAB implementation (left) to the optimized version (right). While all parameters remain in GPU memory, note the explicit transfer of the residual back to the host in the last line.

	CUBLAS BiCGSTAB		merged BiCGSTAB		
line[1]	read	write	merged into	read	write
4	$n+\mathcal{O}(1)$	n	p_update	$3n+\mathcal{O}(1)$	n
5	$2n+\mathcal{O}(1)$	n			
6	$2n+\mathcal{O}(1)$	n			
7	$2nnz+2n+\mathcal{O}(1)$	n	spmv1+reduce1	$2nnz+3n+\mathcal{O}(1)$	$n+\mathcal{O}(1)$
8	$2n+\mathcal{O}(1)$	$\mathcal{O}(1)$			
9	n	n	s_update	$2n+\mathcal{O}(1)$	n
10	$2n+\mathcal{O}(1)$	n			
11	$2nnz+2n+\mathcal{O}(1)$	n	spmv2+reduce2	$2nnz+2n+\mathcal{O}(1)$	$n+\mathcal{O}(1)$
12	$2n+\mathcal{O}(1)$	$\mathcal{O}(1)$			
13	$2n+\mathcal{O}(1)$	$\mathcal{O}(1)$			
14	$2n+\mathcal{O}(1)$	n	xr_update+reduce3	$6n+\mathcal{O}(1)$	$2n+\mathcal{O}(1)$
15	$2n+\mathcal{O}(1)$	n			
16	n	n			
17	$2n+\mathcal{O}(1)$	n			
18	$2n+\mathcal{O}(1)$	$\mathcal{O}(1)$			
2	$2n+\mathcal{O}(1)$	$\mathcal{O}(1)$			
sum	$4nnz+29n+\mathcal{O}(1)$	$11n+\mathcal{O}(1)$	sum	$4nnz+16n+\mathcal{O}(1)$	$6n+\mathcal{O}(1)$

TABLE I
COMPARISON OF GPU MEMORY ACCESS FOR THE CUBLAS REFERENCE IMPLEMENTATION (SEE FIGURE 2) AND THE MERGED VARIANT.

reduces the communication between GPU and host as less kernels are launched. This is particularly important when running CUDA in blocking mode to improve the energy efficiency [1]. Comparing the number of kernels launched in one iteration, we realize that the number of kernels in the BiCGSTAB-merge implementation is case-dependent, as the kernel count in the iterative reduction procedure of the scalar products depends on the system's dimension (see `magma_dbicgmerge_reduce2`, Figure 12 in the Appendix). In fact, for the applied block size of 256, each of the three reduction operations launches $k = \lceil \log_{512}\left(\frac{n}{256}\right)\rceil$ kernels. However, this number is usually small, and k does not exceed 2 in any of the test matrices we consider in Section V.

A prerequisite for reducing the number of kernel launches by merging multiple arithmetic operations into one kernel is to keep parameters like scalar products in the device memory. We achieve this by using an additional array `skp` of the form `[alpha|beta|omega|rho_o|rho_n|nom|t1|t2]` in GPU memory that contains the parameters and two entries for temporary storage. This step comes with reduced memory transfer between GPU and host. The CUBLAS reference implementation transfers 13 double precision values between host and GPU in every iteration, while the new algorithm only needs the transfer of the residual to check the error stopping criterion.

D. Handling the residual stopping process asynchronously

The developed algorithm should now be able to run a complete iteration on the GPU while the only task remaining for the CPU is to administer the kernel launch order. The sticking point becomes checking the residual stopping criterion that is handled by the CPU. While a classical implementation would interrupt the iteration process on the GPU until the CPU has validated the stopping criterion and instructed the continuation

of the execution, this may have severe performance impact in the case of slow GPU-host communication. A workaround is given by an asynchronous check of the stopping criterion. The residual is copied to the host asynchronously, while the GPU continues the iteration process. The CPU receives residuals with some delay, and only interrupts once the residual stopping criterion is met. Although the algorithm may at this time already have started the next iteration, these additional computations are not detrimental to the performance. Only in the case of unstable convergence and a delay larger than the time for one iteration may this scheme become dangerous, as an oscillating residual may result in a "bad breakdown" of the iteration process. However, we never experienced this case in our experiments.

E. From dot product to matrix vector multiply

NVIDIA's CUBLAS library provides an efficient routine to process dot products on the GPU. However, as soon as multiple dot products need to be computed consecutively, performance obviously suffers from memory access and the fact that the reduction for each vector is handled one after another. This motivates us to come up with a kernel able to compute multiple dot products at once and reduce the vectors simultaneously. Although the BiCGSTAB only requires the parallel computation of two dot products, we aim for a general analysis on this topic, as the computation of a set of dot products with one vector being part in all of them can also be seen as a matrix-vector multiplication $A^T x$, where A is a tall and skinny matrix.

While parallel matrix vector multiplications usually assign a set of rows/columns to each processing unit for the A non-transpose/transpose cases respectively, the approach becomes inefficient if A consists of less rows/columns than processing units available. Especially when targeting GPUs, handling columns by threads is not suitable for the $A^T x$ computation, as the typically used thread number will exceed the number of columns by orders of magnitude. The MAGMA library [12] developed at the Innovative Computing Lab (ICL) at the University of Tennessee overcomes this by assigning one SXM to each column, and splitting each column into chunks that are then handled by different threads. To have all 13 SXMs of the K20c working, the matrix needs at least 13 columns [5]. Each column is then split into parts according to the block size, and each thread strides over the complete column, handling one element in every part. In the end, the partial sums computed by the distinct threads are collected using fan in.

Using the ideas based on the dot product where computing units process in a tree-reduction fashion, we extend the implementation proposed in [1] to process multiple vector products simultaneously. The advantage of this algorithm is that instead of only one, all SXMs are utilized to compute the reduction of a single column, with the drawback of additional memory usage. Like in the MAGMA implementation, each thread of a thread block (handled by one SXM) strides over the complete column, but the usage of all SMXs reduces the number of column chunks and the computations of each thread

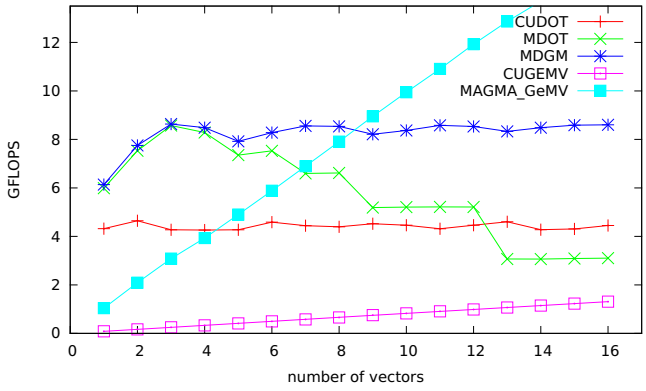

Fig. 6. Performance comparison between CUBLAS dot product, the developed simultaneous dot product implementations MDOT and MDGM, and the matrix vector products from NVIDIA and MAGMA, respectively for a vector length of 100,000.

considerably. The price for this is that every multiprocessor, once the reduction for the thread block is completed, has to write data to the global memory and synchronize with the other multiprocessors after every reduction step, as the partial sums computed by the different thread blocks are used in the next reduction step.

The limited cache size in GPUs poses restrictions when aiming for the simultaneous reduction of multiple vectors, as the shared memory is the key to the efficiency of the implementation. We overcome this bottleneck by processing the data in chunks of vectors allowing for efficient cache usage. Note that the number of vectors in every chunk is dependent on the hardware characteristics, the block size, and the precision format used, but independent of the vector length.

According to the performance shown in Figure 6 (see line labelled MDOT for the multi-dot-product kernel) a chunk size of 4 seems reasonable for our implementation. The obtained kernel using the chunk-size of 4 and labelled as MDGM in Figure 6 shows minor performance loss when hitting the reload barrier, but then stabilizes around 8 Gflop/s, outperforming the sequence of CUBLAS dot products by a factor of two. The difference becomes smaller for larger vectors, as the kernels approach their asymptotic performance peaks of about 18 and 14 Gflop/s respectively (see Figure 8). Interesting is the comparison with the matrix vector product kernels. While NVIDIA's implementation (CUGeMV) is not at all able to keep up with the MDGM for tall and skinny matrices, the matrix vector product provided by MAGMA (MAGMA_GeMV), where one SXM handles one vector, catches up with the CUBLAS dot product as soon as four dot products are needed, and thus, 4 SXMs are used. In Figure 7 we show the superiority areas of MDGM and MAGMA_GeMV as a function of the vector length. As expected, the more column-dominated a matrix is, the more reasonable the usage of MDGM is where all SXMs are used for every row. A direct comparison of MDOT against NVIDIA's dot product for different vector sizes is given in Figure 8. The first observation is that, when computing only one dot product, the MDGM outperforms the

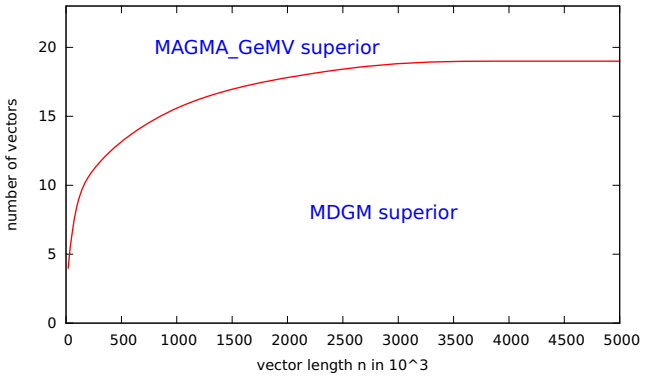

Fig. 7. Superiority areas of MDGM and MAGMA's matrix vector kernel.

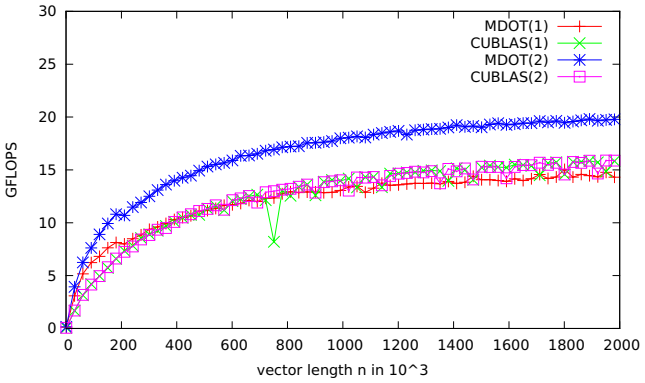

Fig. 8. Performance comparison (double precision) between CUBLAS and MDOT executing 1 and 2 vector products.

CUBLAS implementation for small lengths while CUBLAS yields slightly higher performance as soon as the vector length exceeds 10^6. Close inspection reveals that the performance of MDOT decreases slightly around 200,000. This stems from the iterative reduction procedure: for larger vectors one reduction step is not sufficient, rather a second kernel is needed (see line 62–72 in `magma_dbicgmerge_reduce2`, Figure 12 in the Appendix). When adding a second dot product with one vector shared by both operations (see data labelled MDOT(2) and CUBLAS(2) in Figure 8) the performance of MDOT increases by about one third due to reuse of one vector and the simultaneous reduction of two vectors. For CUBLAS we do not observe any performance improvement when executing two consecutive dot products. Improvement would only become possible by a compiler detecting the reuse of one vector. By reordering the operations in the BiCGSTAB method, we can gather two sets containing two consecutive dot products using the same vector in both computations. For efficient processing, the merged implementation uses the vector `q = [r_hat|r|p|v|s|t]` containing the distinct vectors of the reference implementation. In the accelerated version of BiCGSTAB, we merge the computation of (multiple) dot products with other arithmetic operations where possible. As an example, Figure 12 in the Appendix provides the code for the `magma_dbicgmerge_spmv2` and `magma_dbicgmerge_reduce2` kernels.

V. Experimental Comparison with Performance Model Guiding the Optimizations

In the previous sections, we have proposed different modifications to the BiCGSTAB algorithm structure and its implementation on a GPU-accelerated system. In this section, we aim for a theoretical model quantifying the improvements the modifications are expected to achieve in experiments using a set of test matrices taken from the University of Florida matrix collection (UFMC)[2]. While the matrices were selected to cover a broad spectrum with respect to dimension and sparsity, some key characteristics are summarized in Table II. In Table III, we profile the CUBLAS reference implementation for the different test matrices.

Aiming for a general model providing estimations for the savings rendered by the modifications we proposed in the previous sections, we may consider the existence of two factors determining the improvement when switching from the reference implementation to the merged BiCGSTAB, where we utilize the custom-designed kernels. One is the dominance of the matrix-vector product that we did not change in the optimization process (need to subtract $6n$ from the operation count). According to the data given in Table I, we may expect performance improvements of up to $\eta_{\text{memory}} = \frac{13n+5n-6n}{29n+11n-6n} \approx 35\%$ due to reduced memory transfers in the memory-bound algorithm, however linearly decreasing with the dominance of the SpMV. This already allows for the derivation of a very simple model estimating the expected savings as:

$$\begin{aligned} P_{\text{memory}} &= 1 - \frac{T_{MERGE}}{T_{CUBLAS}} \\ &= (1 - SpMV) \times \eta_{\text{memory}}. \end{aligned} \tag{1}$$

The second factor is the dimension of the system. In Section IV-E, we proposed MDOT, capable of computing multiple dot products simultaneously. But as the improvements when switching from CUBLAS to MDOT depends on the vector size, we have to quantify these as well. For this purpose, we run experiments on the sequence of dot products that occur in the BiCGSTAB algorithm: two sets of consecutive dot products sharing one of the vectors and one separate dot product (see Figure 9). For the remainder of the paper we use the following function (produced with a regression fit of the experimental results) to approximate the runtime savings (shown in Figure 10):

$$\mathscr{F}(n) = \frac{1}{100}\left(\frac{1}{2\times 10^{-7}\times n + 0.021} + 12.5\right). \tag{2}$$

The impact of data transfer and kernel launch overhead is application specific, which makes the reduced GPU-host communication and the asynchronous check of the stopping criterion difficult to integrate into a general model. Instead, we now combine the improvements due to data locality of

[2]UFMC; see http://www.cise.ufl.edu/research/sparse/matrices/

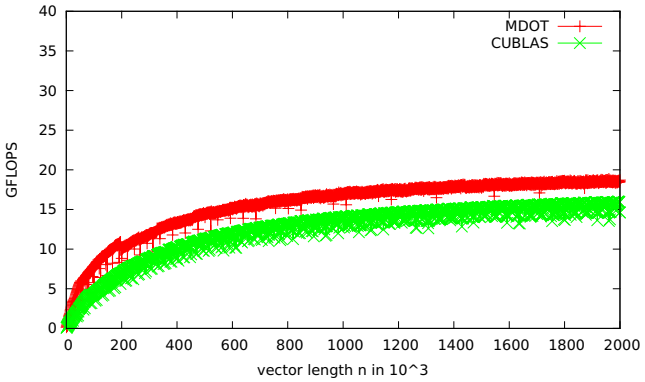

Fig. 9. Performance comparison between CUBLAS and MDOT for the sequence of dot products in one BiCGSTAB iteration.

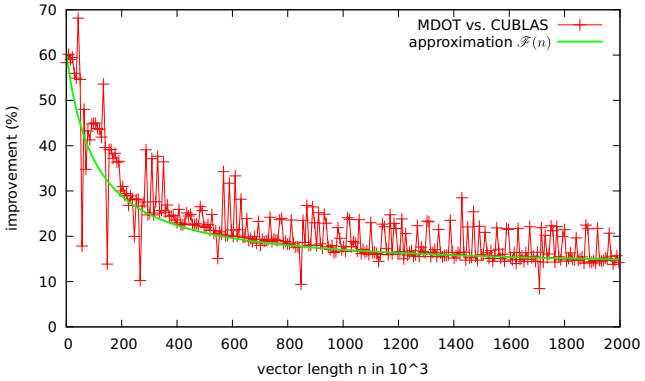

Fig. 10. Size-dependent runtime improvement obtained by replacing CUBLAS with MDOT in the sequence of dot products in BiCGSTAB.

MDOT in Eq. (3), which models the expected improvement that depends on the matrix size n, and the relative portion of the sparse matrix-vector kernel ($SpMV$), and the dot product (dot) in one iteration, respectively.

$$P_{\text{memory+dot}} = \eta_{\text{memory}} \times (1 - \text{SpMV}) + (1 - \mathscr{F}(n)) \times \text{dot}. \quad (3)$$

Using the data from Table II and III, we visualize the savings predicted by $P_{memory+dot}$ in Figure 11 as memory+dot along with experimental data. We observe that in most cases we are able, in general, to provide acceptable estimations, better than the linear model based exclusively on the memory improvement. However, in some cases, we still underestimate the performance improvement. In particular, for the test matrices AUDIKW_1 and BMW3_2 where the matrix-vector kernel dominance should allow only negligible improvement, about 20% of runtime reduction cannot be explained by any of the aforementioned effects. Detailed analysis revealed a phenomenon that needs further research to provide a satisfying explanation: the matrix vector kernel using CSR format can be accelerated on the GPU architecture used by adding additional operations writing data to global memory. It may be assumed

Matrix	#nonzeros (nnz)	Size (n)	nnz/n
AIRFOIL_2D	259,688	14,214	18.27
APACHE_2	4,817,870	715,176	6.74
AUDIKW_1	77,651,847	943,645	82.28
BLOWEYBQ	49,999	10,001	5.00
BMW3_2	11,288,630	227,362	49.65
CAGE_10	150,645	11,397	13.22
ECOLOGY_2	4,995,991	999,999	5.0
FV1	85,264	9,604	8.88
G3_CIRCUIT	7,660,826	1,585,478	4.83
POISSON_3DA	352,762	13,514	26.10
PRES_POISSON	715,804	14,822	48.29
TREFETHEN_2000	41,906	2,000	20.95
TREFETHEN_20000	554,466	20,000	27.72

TABLE II
DESCRIPTION AND PROPERTIES OF THE TEST MATRICES.

that some scheduling internal to the GPU, and maybe a superior cache use, is responsible for this effect, and we will further investigate this issue. While we were not able to reproduce similar behavior on NVIDIA's previous GPU architecture – the Fermi line – we observed savings of about $\eta_{\text{SpMV}} = 15\%$ for the SpMV using large matrices on the Kepler K20c. Tests on the Kepler K20x, to which we had limited access, indicated that the effect remains, however it became smaller. Motivated by this observation, we enhanced the model given in (3) by a correction term reflecting the acceleration of the SpMV due to unknown scheduling advantages when merging it with the other operations. The resulting improvement estimation, that was given in Equation (4) and is labelled `memory+dot+SpMV` in Figure 11, predicts the observed improvements in experiments with high accuracy.

$$P_{\text{memory+dot+SpMV}} = P_{memory+dot} + \text{SpMV}(n, nnz) \times \eta_{\text{SpMV}}. \quad (4)$$

Beyond the successful validation of the derived model, we observe that the main goal of our work was achieved as well: the new BiCGSTAB implementation outperforms the CUBLAS reference implementation for all test cases. Depending on the dominance of the matrix-vector kernel, which we refrained from accelerating, the merged version achieves an average runtime reduction close to 40%. For specific matrices, where the impact of the matrix vector-kernel is small, we achieve speedups as large as 3×.

VI. Summary and Conclusion

Taking the BiCGSTAB method as representative for a Krylov subspace solver, we have investigated how to leverage the performance potential of graphics processing units. The optimized implementation reformulates the algorithm, merges multiple arithmetic into algorithm-specific kernels to reduce the memory traffic, keeps all data in GPU memory to remove pressure from the PCI connection, checks the stopping criterion asynchronously to avoid performance-detrimental synchronization points between CPU and GPU, and uses new

Matrix	total [s]	SpMV [s]	dot [s]	p update [s]	s update [s]	x+r update [s]
AIRFOIL_2D	0.79	0.43 (54%)	0.26 (33%)	0.03 (4%)	0.03 (3%)	0.04 (5%)
APACHE_2	3.94	2.37 (60%)	0.57 (15%)	0.33 (8%)	0.21 (5%)	0.45 (11%)
AUDIKW_1	190.92	188.94 (99%)	0.70 (0%)	0.42 (0%)	0.27 (0%)	0.58 (0%)
BLOWEYBQ	0.41	0.06 (15%)	0.25 (61%)	0.03 (7%)	0.02 (6%)	0.04 (9%)
BMW3_2	25.26	24.50 (97%)	0.38 (01%)	0.12 (0%)	0.08 (0%)	0.16 (1%)
CAGE_10	0.60	0.21 (36%)	0.28 (47%)	0.03 (5%)	0.02 (4%)	0.04 (6%)
ECOLOGY_2	4.30	2.24 (52%)	0.71 (16%)	0.45 (10%)	0.28 (7%)	0.61 (14%)
FV1	0.48	0.10 (21%)	0.28 (59%)	0.03 (6%)	0.02 (5%)	0.04 (9%)
G3_CIRCUIT	6.77	3.69 (55%)	0.98 (15%)	0.69 (10%)	0.44 (6%)	0.95 (14%)
POISSON_3DA	1.24	0.85 (69%)	0.28 (23%)	0.03 (2%)	0.03 (2%)	0.04 (3%)
PRES_POISSON	1.63	1.27 (78%)	0.26 (16%)	0.03 (2%)	0.03 (2%)	0.04 (2%)
TREFETHEN_2000	0.51	0.15 (30%)	0.26 (52%)	0.03 (5%)	0.02 (4%)	0.03 (7%)
TREFETHEN_20000	1.45	1.03 (71%)	0.31 (21%)	0.03 (2%)	0.03 (2%)	0.04 (3%)

TABLE III
PROFILING OF THE CUBLAS REFERENCE IMPLEMENTATION, ALL TIMINGS ARE FOR 1000 BICGSTAB ITERATIONS.

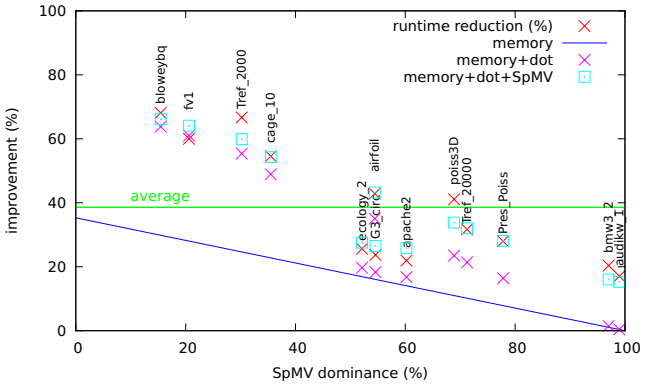

Fig. 11. Performance improvement by replacing the CUBLAS reference implementation with the reformulated version depending on the matrix vector kernel dominance in the original code.

highly-efficient dot product kernels able to reduce multiple dot products simultaneously. Compared to a reference implementation where the arithmetic operations of the mathematical formulation are directly translated into CUBLAS function calls, the new implementation yields appealing performance improvement for matrices taken from the University of Florida Matrix Collection. Furthermore, we have derived a model, that succeeds in predicting the performance improvements. This model is based on the reduced memory accesses and the faster execution due to our new and optimized dot product. While we focused on BiCGSTAB, the necessity of method-specific kernels to achieve high performance on GPUs also applied to other Krylov subspace methods, and deriving models similar to ours may provide a-priori insight into whether a specific solver is suitable for custom-designed GPU implementation. Future research in this direction should focus on including preconditioner techniques, as preconditioning is often the key to efficiency when solving sparse linear systems via Krylov subspace methods.

ACKNOWLEDGMENTS

This research was supported in part by DOE grant #DE-SC0010042, NVIDIA, and the National Science Foundation.

REFERENCES

[1] H. Aliaga, J.and Anzt, J. Pérez, and E. Quintana-Ortí. Reformulated Conjugate Gradient for the energy-aware solution of linear systems on GPUs. In *41st International Conference on Parallel Processing*, 2013.
[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
[3] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.
[4] Dietrich Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*, volume 3. Cambridge University Press, 2007.
[5] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Kepler GK110. Whitepaper, 2012.
[6] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990.
[7] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:18–28, March 1990.
[8] Jack J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, March 1988.
[9] Jack J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, March 1988.
[10] Dominik Göddeke. *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*. PhD thesis, Technische Universität Dortmund, Fakultät für Mathematik, May 2010.
[11] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.
[12] MAGMA 1.4.1. http://icl.cs.utk.edu/magma/, 2013.
[13] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.
[14] William Sawyer. CUSPARSE/CUBLAS example: BiCGStab iterative solver for non-symmetric linear systems, May 2011. https://hpcforge.org/plugins/mediawiki/wiki/gpu-training/index.php/Main_Page.
[15] H. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

```
__global__ void
magma_zbicgmerge_spmv2( int n,
                        double *val,
                        int *rowptr,
                        int *colind,
                        double *s,
                        double *t,
                        double *vtmp ){

  extern __shared__ double tmp[];
  int BD = blockDim.x;
  int Idx = threadIdx.x;
  int i   = blockIdx.x * blockDim.x + Idx;
  if( i<n ){
    double dot = MAGMA_Z_ZERO;
    int start = rowptr[ i ];
    int end = rowptr[ i+1 ];
    for( int j=start; j<end; j++)
      dot += val[ j ] * s[ colind[ j ] ];
    t[ i ] =  dot;
  }
  __syncthreads();
  // 2 dot products: <s,t> and <t,t>
  if (i<n){
    double tmp2 = t[i];
    tmp[Idx] = s[i] * tmp2;
    tmp[Idx+BD] = tmp2 * tmp2;
  }
  else
    for( int j=0; j<2; j++)
      tmp[Idx+j*BD] = 0.0;
  __syncthreads();
  if( Idx < 128 )
    for( int j=0; j<2; j++)
      tmp[ Idx+j*BD ] += tmp[ Idx+j*BD+128 ];
  __syncthreads();
  if( Idx < 64 )
    for( int j=0; j<2; j++)
      tmp[ Idx+j*BD ] += tmp[ Idx+j*BD+64 ];
  __syncthreads();
  if( Idx < 32 ){
    volatile double *tmp2 = tmp;
    for( int j=0; j<2; j++){
      tmp2[ Idx+j*BD ] += tmp2[ Idx+j*BD+32 ];
      tmp2[ Idx+j*BD ] += tmp2[ Idx+j*BD+16 ];
      tmp2[ Idx+j*BD ] += tmp2[ Idx+j*BD+8 ];
      tmp2[ Idx+j*BD ] += tmp2[ Idx+j*BD+4 ];
      tmp2[ Idx+j*BD ] += tmp2[ Idx+j*BD+2 ];
      tmp2[ Idx+j*BD ] += tmp2[ Idx+j*BD+1 ];
    }
  }
  if( Idx == 0 )
    for( j=0; j<2; j++)
      vtmp[ blockIdx.x+j*n ] = tmp[ j*BD ];
}
```

```
// block reduction for k vectors
__global__ void
magma_reduce( int Gs, int n, int k,
              double *vtmp, double *vtmp2 ){
  extern __shared__ double tmp[];
  int Idx = threadIdx.x, int BS = 128;
  int gridSize = BS  * 2 * gridDim.x;
  for( int j=0; j<k; j++){
    int i = blockIdx.x * ( BS * 2 ) + Idx;
    tmp[Idx+j*BS] = 0.0;
    while (i < Gs ) {
      tmp[ Idx+j*BS  ] += vtmp[ i+j*n ];
      tmp[ Idx+j*BS  ] += ( i+BS< Gs ) ?
                          vtmp[ i+j*n+BS ] : 0.0;
      i += gridSize;
    }
  }
  __syncthreads();
  if ( Idx < 64 )
    for( int j=0; j<k; j++)
      tmp[ Idx+j*BS ] += tmp[ Idx+j*BS+64 ];

  if( Idx < 32 ){
    volatile double *tmp2 = tmp;
    for( int j=0; j<k; j++){
      tmp2[ Idx+j*BS ] += tmp2[ Idx+j*BS+32 ];
      tmp2[ Idx+j*BS ] += tmp2[ Idx+j*BS+16 ];
      tmp2[ Idx+j*BS ] += tmp2[ Idx+j*BS+8 ];
      tmp2[ Idx+j*BS ] += tmp2[ Idx+j*BS+4 ];
      tmp2[ Idx+j*BS ] += tmp2[ Idx+j*BS+2 ];
      tmp2[ Idx+j*BS ] += tmp2[ Idx+j*BS+1 ];
    }
  }
  if ( Idx == 0 )
    for( int j=0; j<k; j++)
      vtmp2[ blockIdx.x+j*n ] = tmp[ j*BS ];
}

// kernel computing omega
__global__ void
magma_zbicgstab_omega( double *skp ){
  int i = blockIdx.x * blockDim.x+threadIdx.x;
  if( i==0 ){
    skp[2] = skp[6]/skp[7];
    skp[3] = skp[4];
  }
}

// reduction routine
void
magma_zbicgmerge_reduce2( int n, int Gs, int Bs,
                double *d1, double *d2, double *skp ){
  int k=2; // number of dot products
  dim3 Gs_next;
  int Ms =  k * Bs * sizeof( double );
  double *aux1 = d1, *aux2 = d2;
  int b = 1;
  while( Gs > 1 ){
    Gs_next = ( Gs+Bs-1 )/ Bs ;
    if( Gs_next == 1 ) Gs_next = 2;
    magma_reduce<<< Gs_next/2, Bs/2, Ms/2 >>>
                        ( Gs, n, k, aux1, aux2 );
    Gs_next = Gs_next/2; Gs = Gs_next;
    b = 1 - b;
    if( b ){ aux1 = d1; aux2 = d2; }
    else   { aux2 = d1; aux1 = d2; }
  }
  cudaMemcpy( skp+6, aux1, sizeof(double),
               cudaMemcpyDeviceToDevice );
  cudaMemcpy( skp+7, aux1+n, sizeof(double),
               cudaMemcpyDeviceToDevice );
  dim3 Bs2( 1 );
  dim3 Gs2( 1 );
  magma_zbicgstab_omega<<< Gs2, Bs2, 0 >>>( skp );
}
```

Fig. 12. Algroithm-specific kernel implementation for `magma_dbicgmerge_spmv2` and `magma_dbicgmerge_reduce2` using a block size of 256.