International Journal of High Performance Computing Applications
000(00):1–18
©The Author(s) 2010
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI:doi number
http://mms.sagepub.com

Acceleration of GPU-based Krylov solvers via Data Transfer Reduction

Hartwig Anzt[*]**, Stanimire Tomov, Piotr Luszczek**
Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA

William Sawyer
Swiss National Supercomputing Centre (CSCS), Lugano, Switzerland

Jack Dongarra
Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
Oak Ridge National Laboratory, USA University of Manchester, UK

Abstract
Krylov subspace iterative solvers are often the method of choice when solving large sparse linear systems. 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 reduce certain data communications, and hence fail to leverage the full potential of the accelerator. In this paper we target the acceleration of Krylov subspace iterative methods for GPUs, and in particular the BiCGSTAB solver, showing that significant improvement can be achieved by reformulating the method to reduce data-communications through application-specific kernels instead of using the generic BLAS kernels, e.g., as provided by NVIDIA's cuBLAS library, and by designing a GPU-specific sparse matrix-vector product (SpMV) kernel that is able to more efficiently use the GPU's computing power. Furthermore, we derive a model estimating the performance improvement, and use experimental data to validate the expected runtime savings. Considering that the derived implementation achieves significantly higher performance, we assert that similar optimizations addressing algorithm structure, as well as SpMV, are crucial for the subsequent development of high-performance GPU-accelerated Krylov subspace iterative methods.

Keywords
Krylov Subspace Methods, Iterative Solvers, Sparse Linear Systems, Graphics Processing Units, BiCGSTAB

[*] Corresponding author; e-mail: hanzt@icl.utk.edu

1. Introduction

Krylov subspace iterative methods (1) are among the most popular for solving large sparse linear systems. Against the background of an increasing number of computer systems featuring hardware accelerators like GPUs (2; 3), the efficient use of the available computing power requires their inclusion in the computation. Moreover, GPU's very high memory bandwidth — and current four to five times higher energy efficiency than multicore CPUs (4) — motivate the development of new software technologies for their use in sparse computations. A straightforward way to employ accelerators in Krylov subspace solvers is to offload all matrix and vector computations to the device using library functions. However, recent research has shown that this approach may provide appealing performance improvement against a CPU-restricted code, but fails to leverage the full potential of the accelerator (5; 6). Using generic kernels provided by libraries fails to benefit from data reuse, which is a key optimization strategy for memory-bound computations and algorithms. Custom-designed kernels can provide a work-around by merging multiple arithmetic operations, and keeping data in shared memory whenever possible. Aside from that, the sparse matrix-vector product needed to generate the Krylov subspace (1) is often the computationally most expensive part of the algorithms. Hence, accelerating this kernel, e.g., by accounting for the properties of the specific problem and integrating it into the implementation, is often seen as the major challenge when porting and optimizing Krylov solvers on GPUs.

1.1. Related Work

Although the memory-bound characteristics of sparse iterative solvers pose a challenge when porting them to throughput-processors like GPUs, significant research effort focusses on deriving efficient implementations. As the sparse matrix-vector product (SpMV) serves as the backbone of many iterative solvers, the acceleration of this stand-alone building block is often considered as the key to enhancing the performance of the complete class of iterative methods. Matrix storage formats and sophisticated kernels accounting for the specific hardware and matrix characteristics have been extensively studied in literature (7; 8; 9; 10). Performance comparisons reveal that leveraging the computational power of GPU accelerators can also be beneficial when considering the total iterative solver runtime (11; 12; 13). Aside from research implementations, several open-source software packages provide GPU-support for solving sparse linear systems (14; 15; 16; 17). The implementation of the BiCGSTAB and the custom-designed kernels we use for our analysis in this paper are taken from the MAGMA (17) numerical linear algebra library. The performance of the BLAS-1 and BLAS-2 CUDA kernels that are typically needed for all iterative methods, can be improved by reducing the communication of the memory-bound operations to GPU main memory. In (18) the authors have shown how this can be achieved by using a source-to-source compiler that merges vector and dense matrix-vector operations. Sparse iterative solvers have been addressed with similar algorithmic modifications in (5), where the authors have shown how custom-designed kernels improve performance and energy-efficiency of a GPU implementation for the Conjugate Gradient iterative solver. In (6), this idea was extended to a more generic approach using the BiCGSTAB (19) algorithm as a target application: aside from applying the aggregation of multiple arithmetic operations into a single kernel to reduce GPU memory traffic and CPU-GPU communication, the authors proposed a kernel computing multiple dot products simultaneously, and derived a model providing an a-priori approximation of the expected performance improvement.

1.2. Outline

In this paper we extend these results by combining the communication reduction ideas with the optimization of the sparse matrix-vector product. Also, we derive a theoretical communication bound for the GPU implementation of the BiCGSTAB solver, and include a model that predicts the performance savings. We structure the paper as follows: we begin in Section 2 with a brief characterization of the Krylov subspace solvers, and review the BiCGSTAB algorithm as a typical representative. For this, we also introduce a baseline reference implementation, based on the cuBLAS library which we use for further analysis. In Section 3 we propose modifications to the distinct building blocks typical for a Krylov subspace solver: In 3.1 we then discuss the importance of the sparse matrix-vector product (SpMV), and present a GPU kernel that computes the multiplication at significantly higher performance than the default CSR-based implementation. We derive application-specific kernels in Subsections 3.2, and 3.3 by merging multiple arithmetic operations, and provide a lower bound of the communication volume needed for a parallel implementation of BiCGSTAB. On the hardware platform we introduce in 4.1, we present the performance results for the optimized algorithm-specific kernels we developed in Sections 4.2, 4.3 and 4.4. Based on these modifications, in Subsection 4.5, we derive a model quantifying the expected performance improvements and use experimental results to validate the model, revealing the superior performance of the reformulated BiCGSTAB algorithm across the complete range of SpMV dominance.

2. Krylov subspace solvers

Krylov subspace solvers (1) are among the most widely used methods for solving sparse linear systems iteratively. The methods are based on the idea of approximating the solution in a Krylov subspace of increasing dimension. In the basic approach, the subspace of dimension d is generated by the system matrix A, and a vector s as the subspace spanned by the projections of s under the first d powers of A, that is:

$$\mathcal{K}_d(A,s) = \text{span}\{s, As, A^2 s \dots A^{d-1} s\}. \tag{1}$$

To solve a system $Ax = b$ of dimension n, Krylov methods use a starting vector x_0 and an initial residual $r_0 = b - Ax_0$ to generate a sequence of approximations x_i in $K_i(A, r_0)$. In the absence of roundoff error and a breakdown caused by division by zero, the exact solution is reached after a finite number of steps (1). However, in particular when enhancing the method with a sophisticated preconditioner (1), a good approximation is typically available after few iterations, much smaller than the theoretical limit, which makes Krylov methods attractive as iterative solvers. Among the most popular methods are the Conjugate Gradient (20) for solving symmetric positive definite systems, and the GMRES and BiCGSTAB algorithms for solving nonsymmetric linear systems of equations (1). While there exists a large variety of Krylov subspace solvers, they are all comprised of the following building blocks:

- **Sparse matrix-vector product.**
  To generate the Krylov subspace, every iteration contains at least one SpMV product[1]. Typically, this operation accounts for a significant part of the computational cost of the solver. Indeed, the SpMV is memory bound, as well as latency bound due to the irregular memory accesses to the vector's elements, making it notoriously slow and hard to optimized and in the end being able to achieve only a fraction of the peak performance of modern architectures.
- **Reductions as part of scalar products.**
  In the generation of the Krylov subspace, the pairwise orthogonalization of the vectors has to be ensured. This requires the computation of scalar products, which include global reductions. Despite parallelization approaches like the fan-in algorithm (22), reductions usually pose a bottleneck when running on parallel platforms, as they require synchronization and communication between the processors.
- **Local vector updates.**
  Even though local vector updates are inherently parallel and hence suitable for highly parallel architectures like GPUs, they are also memory bound. Therefore, optimizing the number of reads/writes during the updates is a major goal towards a significant performance improvement.

[1] Using communication-avoiding approaches like matrix powers kernel (21), the matrix-vector products may be grouped, but it is still possible to map them to the respective iterations.

In order to accelerate Krylov subspace solvers, in Section 3 we derive algorithm-specific kernels that reflect the aforementioned building blocks, but improve the performance by increasing the computational intensity. For better cache reuse, we gather multiple vector updates into a single kernel, and derive a parallel reduction kernel capable of handling multiple scalar products simultaneously. Furthermore, we break up the barriers between the building blocks by merging vector updates with the first part of a scalar product into a single kernel. Although we have shown (6) that an equivalent kernel merging is possible for the sparse matrix-vector product, we refrain from this modification to maintain the genericness of the matrix-vector kernel, but present an alternative to the basic CSR-based kernel in Section 3.1, which achieves significantly higher performance.

Although the focus of this paper is on the BiCGSTAB solver, our algorithm redesign techniques, new kernels, and performance improvements can be projected easily through the developed main building blocks to other Krylov subspace methods.

2.1. 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 (19). It belongs to the above introduced class of Krylov subspace solvers and can be used to solve linear systems of equations that are not necessarily symmetric and positive definite (1). 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 (23).

The BiCGSTAB method for solving 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, is outlined in Figure 1 (left side). Here τ sets an upper bound on the relative residual for the computed solution approximation x_k, and *maxiter* is a bound for the maximum number of iterations. Besides the two sparse matrix-vector multiplications at lines 12 and 15 that usually dominate the computational effort, every BiCGSTAB iteration, as shown, can be expressed using the BLAS copy, axpy, dot product, and norm routines. The floating point operations (flops) for one iteration, including the residual computation, can be estimated $2 \cdot SpMV + 22n$. Each of the SpMVs counts for $4nnz + 2n$ additional flops if using a CSR-based multiplication (see Section 3.1), where *nnz* is the number of nonzero entries in A.

An implementation of BiCGSTAB for GPU-accelerated platforms (25) 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 of the iteration loop are given on the right side in Figure 1. As the intention of this implementation was to provide a high-performance BiCGSTAB through the highly optimized cuBLAS library, which is a common and highly efficient practice, we take it as a reference implementation to compare against the new developments.

3. 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 improving the sparse matrix-vector product, and designing application-specific routines, reducing the number of kernel calls, GPU memory transfers, and GPU-host communications (5). 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 enables a better GPU utilization by reducing the number of kernel calls and data communications. While Figure 2 provides a general overview of the original cuBLAS reference implementation and the new implementation featuring these improvements, we discuss the distinct optimizations and modifications we propose to the classical formulation of the algorithm in the

```
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: $\rho_0 := \omega_0 := \alpha_0 := 1$
5: $v_0 := p_0 := 0$
6: $k := 0$
7: while $(k < maxiter)$ && $(res > \tau)$
8:     $k := k+1$
9:     $\rho_k := \hat{r}_0^\mathsf{T} r_{k-1}$                                (dot)
10:    $\beta := \frac{\rho_k}{\rho_{k-1}} \frac{\alpha_{k-1}}{\omega_{k-1}}$
11:    $p_k := r_{k-1} + \beta\,(p_{k-1} - \omega_{k-1} v_{k-1})$      (scal + 2 axpy)
12:    $v_k := Ap_k$                                           (SpMV)
13:    $\alpha_k := \frac{\rho_k}{\hat{r}_0^\mathsf{T} v_k}$                                  (dot)
14:    $s_k := r_{k-1} - \alpha_k v_k$                               (copy + axpy)
15:    $t_k := As_k$                                           (SpMV)
16:    $\omega_k := \frac{s_k^\mathsf{T} t_k}{t_k^\mathsf{T} t_k}$                                  (2 dot)
17:    $x_k := x_{k-1} + \alpha_k p_k + \omega_k s_k$                   (2 axpy)
18:    $r_k := s_k - \omega_k t_k$                                  (copy + axpy)
19:    $res = r_k^\mathsf{T} r_k$                                    (dot)
20: end
```

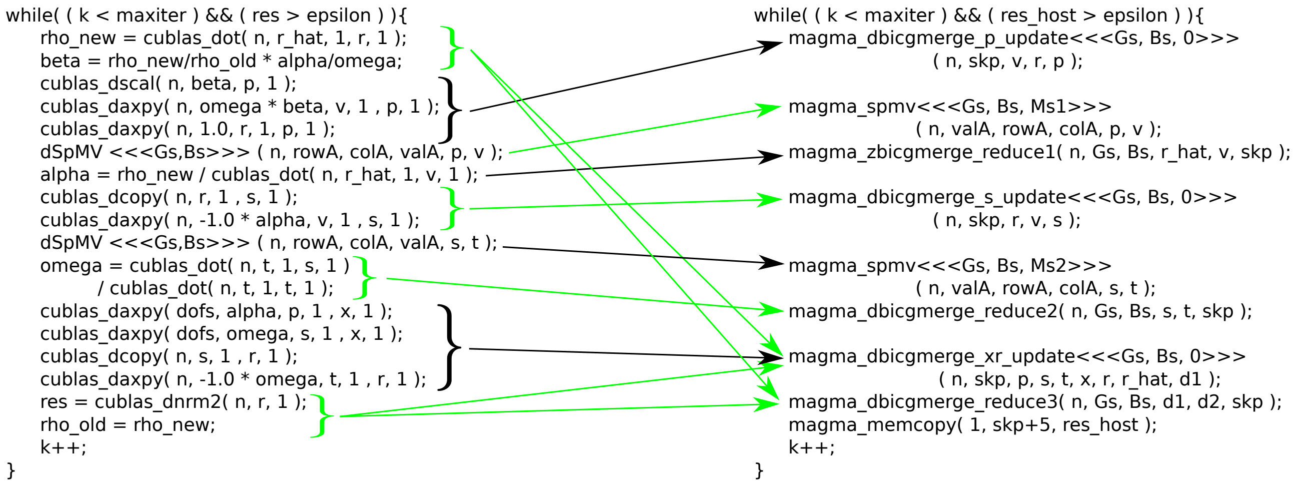
```
while( (k < maxiter) && (res > epsilon) ){
    rho_new = cublasDdot(n, r_hat, 1, r, 1);
    beta = rho_new/rho_old * alpha/omega;
    cublasDscal(n, beta, p, 1 );
    cublasDaxpy(n, omega*beta, v, 1 , p, 1);
    cublasDaxpy(n, 1.0, r, 1, p, 1);
    dSpMV <<<Gs,Bs>>>
          (n, rowA, colA, valA, p, v);
    alpha = rho_new
            / cublasDdot(n, r_hat, 1, v, 1);
    cublasDcopy(n, r, 1 , s, 1 );
    cublasDaxpy(n, -1.0*alpha, v, 1 , s, 1);
    dSpMV <<<Gs,Bs>>>
          (n, rowA, colA, valA, s, t);
    omega = cublasDdot(n, t,1, s,1)
                  / cublasDdot(n, t,1, t,1);
    cublasDaxpy(dofs, alpha, p, 1 , x, 1);
    cublasDaxpy(dofs, omega, s, 1 , x, 1);
    cublasDcopy(n, s, 1 , r, 1);
    cublasDaxpy(n, -1.0*omega, t, 1 , r, 1);
    res = cublasDnrm2(n, r, 1);
    rho_old = rho_new;
    k++;
}
```

Fig. 1. Algorithmic description of the BiCGSTAB method (24) (left), and a reference GPU implementation for the iteration loop using the cuBLAS library (right).

following sections. Considering the building blocks identified in Section 2 to be characteristic for Krylov methods, we address the optimization of the matrix-vector product in Section 3.1, the reductions related to scalar products in Section 3.2, and the reduction of memory transfers related to parallel vector updates in Section 3.3. Note that in contrast to the approach in (6), we refrain from combining the matrix-vector multiplications with other operations. Although this results in small performance penalties, we argue that the flexibility of choosing the SpMV kernel according to the target problem has a higher priority. Thus, we want to point out that our communication-avoiding (CA) optimizations are for the standard Krylov methods, and we do not investigate the potential of breaking up the data dependency between the sparse matrix-vector multiply and the dot products as in recent work on the *s*-step and communication-avoiding Krylov methods (21; 26).

3.1. Accelerating the sparse matrix-vector product

In the iterative solution of sparse linear systems with the Krylov subspace methods, the matrix-vector product — which generates the vector space — is usually the dominant contributor to the overall computational cost of each iteration. Hence, a significant effort is spent on developing storage formats and corresponding computational kernels that — together — are suitable for efficient execution on the target architecture. In the spirit of reducing data transfers, one key objective when targeting GPUs is to allow for coalescent memory access in the matrix-vector kernel. Aligning the data in memory does not decrease the number of useful transfers, but it decreases the number of “wasted” transfers, as every access touches a set of bytes contiguous in memory (in the case of NVIDIA GPUs it is 32 bytes (27)). If a request only uses a subset of these bytes, either by requesting only a single floating point number, or several non-contiguous entries scattered in memory, the remaining transfer capacity is lost. Hence, a coalescent memory access virtually decreases the data transfers by reducing the wasted transfer capacity. Unfortunately, no GPU kernel exists for the matrix-vector product, that is superior in terms of performance for all sparse matrices. Instead, the performance varies depending on the structure of the matrix. Typically, sparse matrices exhibit varying sparsity patterns, and when used inside iterative solvers, any given kernel can be outperformed by another kernel for at least one matrix with just the right structure. Hence, we do not claim that the sparse matrix-vector (SpMV) kernel that we present next is the best one available, but a performance analysis in (10) has



```
while( ( k < maxiter ) && ( res > epsilon ) ){
    rho_new = cublas_dot( n, r_hat, 1, r, 1 );
    beta = rho_new/rho_old * alpha/omega;
    cublas_dscal( n, beta, p, 1 );
    cublas_daxpy( n, omega * beta, v, 1 , p, 1 );
    cublas_daxpy( n, 1.0, r, 1, p, 1 );
    dSpMV <<<Gs,Bs>>> ( n, rowA, colA, valA, p, v );
    alpha = rho_new / cublas_dot( n, r_hat, 1, v, 1 );
    cublas_dcopy( n, r, 1 , s, 1 );
    cublas_daxpy( n, -1.0 * alpha, v, 1 , s, 1 );
    dSpMV <<<Gs,Bs>>> ( n, rowA, colA, valA, s, t );
    omega = cublas_dot( n, t, 1, s, 1 )
          / cublas_dot( n, t, 1, t, 1 );
    cublas_daxpy( dofs, alpha, p, 1 , x, 1 );
    cublas_daxpy( dofs, omega, s, 1 , x, 1 );
    cublas_dcopy( n, s, 1 , r, 1 );
    cublas_daxpy( n, -1.0 * omega, t, 1 , r, 1 );
    res = cublas_dnrm2( n, r, 1 );
    rho_old = rho_new;
    k++;
}
```

```
while( ( k < maxiter ) && ( res_host > epsilon ) ){
    magma_dbicgmerge_p_update<<<Gs, Bs, 0>>>
                ( n, skp, v, r, p );
    magma_spmv<<<Gs, Bs, Ms1>>>
                ( n, valA, rowA, colA, p, v );
    magma_zbicgmerge_reduce1( n, Gs, Bs, r_hat, v, skp );
    magma_dbicgmerge_s_update<<<Gs, Bs, 0>>>
                ( n, skp, r, v, s );
    magma_spmv<<<Gs, Bs, Ms2>>>
                ( n, valA, rowA, colA, s, t );
    magma_dbicgmerge_reduce2( n, Gs, Bs, s, t, skp );
    magma_dbicgmerge_xr_update<<<Gs, Bs, 0>>>
                ( n, skp, p, s, t, x, r, r_hat, d1 );
    magma_dbicgmerge_reduce3( n, Gs, Bs, d1, d2, skp );
    magma_memcopy( 1, skp+5, res_host );
    k++;
}
```

Fig. 2. 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.

shown that it achieves very good performance on the wide range of tested matrices and it also compares favorably to the highly-tuned implementations provided in NVIDIA's cuSPARSE library (16).

For dense matrices it is usually reasonable to store all matrix entries in consecutive position in the computer memory. For sparse matrices, which are typical targets for Krylov-subspace solvers and may be characterized by a large number of zero elements, storing these zeros is not only unnecessary for numerical properties, but would clearly result in significant storage overhead. Various storage layouts exist that aim to reduce the memory footprint of the sparse matrix by storing only a fraction of the elements explicitly (mostly the non-zero ones), and do not store all other (zero-) elements (24; 28; 29). We note that numerous ideas also exist on how to benefit from additional matrix characteristics like symmetry, tridiagonal form, or a special (and thus predictable) sparsity pattern. The CSR format (24) is based on the straightforward idea that only non-zero entries of the matrix are stored. In addition to the array of **values** containing the non-zero elements, two integer arrays **colind** and **rowptr** are used to locate the elements in the matrix, see Figure 3. This storage format is often suitable when computing a sparse matrix-vector product on processors with a deep cache hierarchy, because it reduces the memory bandwidth requirements to a minimum. However, CSR does not allow for coalesced memory access which is important when computing the same product on streaming-processors like GPUs. Instead, the ELLPACK format is preferable and provides padding of the rows with zeros to achieve a uniform row-length, which directly allows for coalesced memory access, which we already identified as a key to optimizing data transfer. The ELLPACK format requires no **rowptr** array, because every row now contains the same number of elements in memory. The storage overhead of ELLPACK is determined by the "longest row," which is the maximum number of nonzero elements in one row of the matrix, see Figure 3. While this is usually compensated for by the more efficient hardware usage when targeting streaming processors, it is unlikely to be suitable for cache-based processor architectures, as the explicitly stored zero elements all have to be processed, which consumes scarce resources – primarily the memory bandwidth. One improvement that reduces the storage overhead is the introduction of the sliced ELLPACK format, which splits the original matrix into blocks of rows, and each slice is then stored using the ELLPACK format, see Figure 3. The resulting format is usually abbreviated as SELL or SELL-C, where C denotes the blocksize (8; 9). As the number of explicitly stored elements in each row is no longer determined by the maximum of the non-zero elements in one row of the matrix but by the "longest row" in this block of rows, some of the slices may have less storage overhead compared to the ELLPACK format. The blocksize becomes the parameter that controls the fill-in, in other words, using the blocksize of 1 (SELL-1) results in the CSR format, using a blocksize of (the matrix dimension) n (SELL-n) results in the original ELLPACK format. However, an additional integer array pointing to

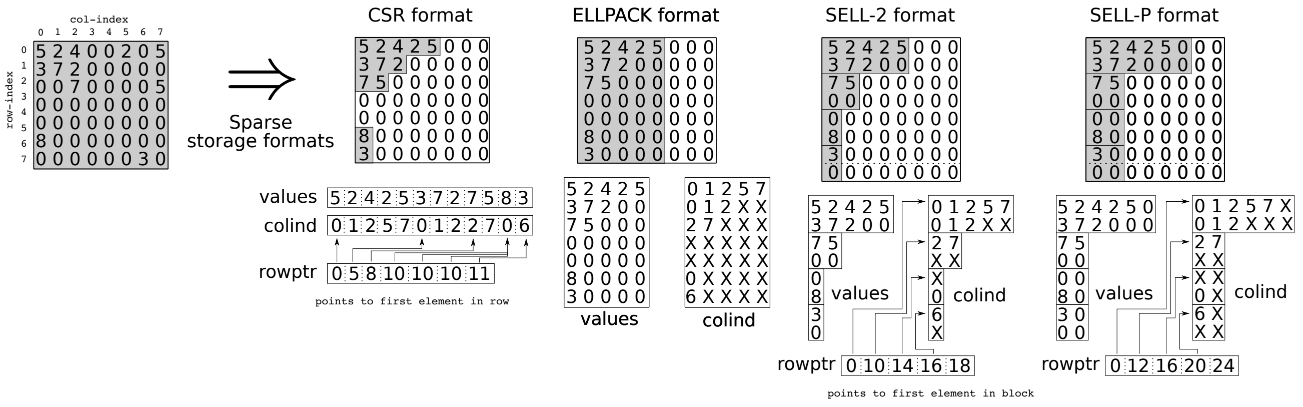


Fig. 3. Dense and sparse matrix storage format representation (10). The memory requirement is visualized with the grey areas. Notice that using SELL-C with the blocksize $b = 2$ (SELL-2), requires adding one row to the original matrix. Furthermore, padding the SELL-P format to a rowlength divisible by 2 ($t = 2$), requires explicit storage of some additional zeros.

the start of each slice is needed. The idea of a blocking ELLPACK has been applied to block matrices (30) where structured matrices are stored by using a grid of small ELLPACK blocks of auto-tuned size. Also, sliced ELLPACK can be enhanced with a-priori row sorting (9) such that the rows with a similar number of non-zero elements are gathered in one block. This obviously reduces the fill-in further, but it is important to trade-off between the cost of sorting and the acceleration of the sparse matrix-vector product. In (9), the authors also propose a trimmed sliced ELLPACK format that arises as a hybrid combination of SELL-C-σ and CSR, but refrain from showing performance results. In the remainder of their paper they argue that the SELL-C-σ format is suitable for cross-platform usage and show in benchmark experiments that it is capable of competing with other storage layouts on different architectures.

Another approach to reducing the computational overhead – without lowering the storage overhead – involves storing the number of non-zero elements in each row in an additional array and computing the partial products only for the non-zero elements. This approach was proposed in combination with assigning multiple threads to one row of the matrix in (10), and the corresponding benchmark results showed the superiority of this idea over the plain ELLPACK format on GPUs. It is beneficial to assign multiple threads to one row and integrate it into an optimized hardware-aware implementation of the sparse matrix-vector product for the SELL-C/SELL-C-σ matrix format (10). For this purpose the SELL-P format (P for "padded") was introduced as a natural extension to the SELL-C/SELL-C-σ with the hardware capabilities in mind. The padding of the rows with zeros is such that the rowlength of each block becomes a multiple of the number of threads assigned to each row, see Figure 3 for the $t = 2$.

In the sparse matrix-vector product kernel for the SELL-P format, the threads of the GPU thread-block handle one slice of the partitioned ELLPACK. The threads are arranged in a $b \times t$ 2D thread grid, where b is the number of rows in one slice, and t the number of threads assigned to each row, see Figure 4. For each slice, the kernel computes the number, *max_*, of necessary multiply-add's that each thread has to handle, and the threads proceed with this information. Once all the data is processed, the partial products are written into shared memory, and a fan-in algorithm with an increment of the thread count computes the sum for each row in shared memory. Accounting for the parameters α and β in the SpMV operation $y = \alpha \cdot Ax + \beta y$, the result is written back into the global memory. The grid necessary to launch the thread blocks has to cover the complete matrix, i.e., the number of blocks is equal to s – the number of slices the matrix is partitioned into. As this number is typically large, a 2D grid of thread blocks, with both grid dimensions close to $\sqrt{s}$, is suitable for efficient processing. The pseudo-code for the kernel is provided on the left in Figure 11, in the Appendix. The underlying data layout, the memory access pattern, and the reduction step is visualized in Figure 4.

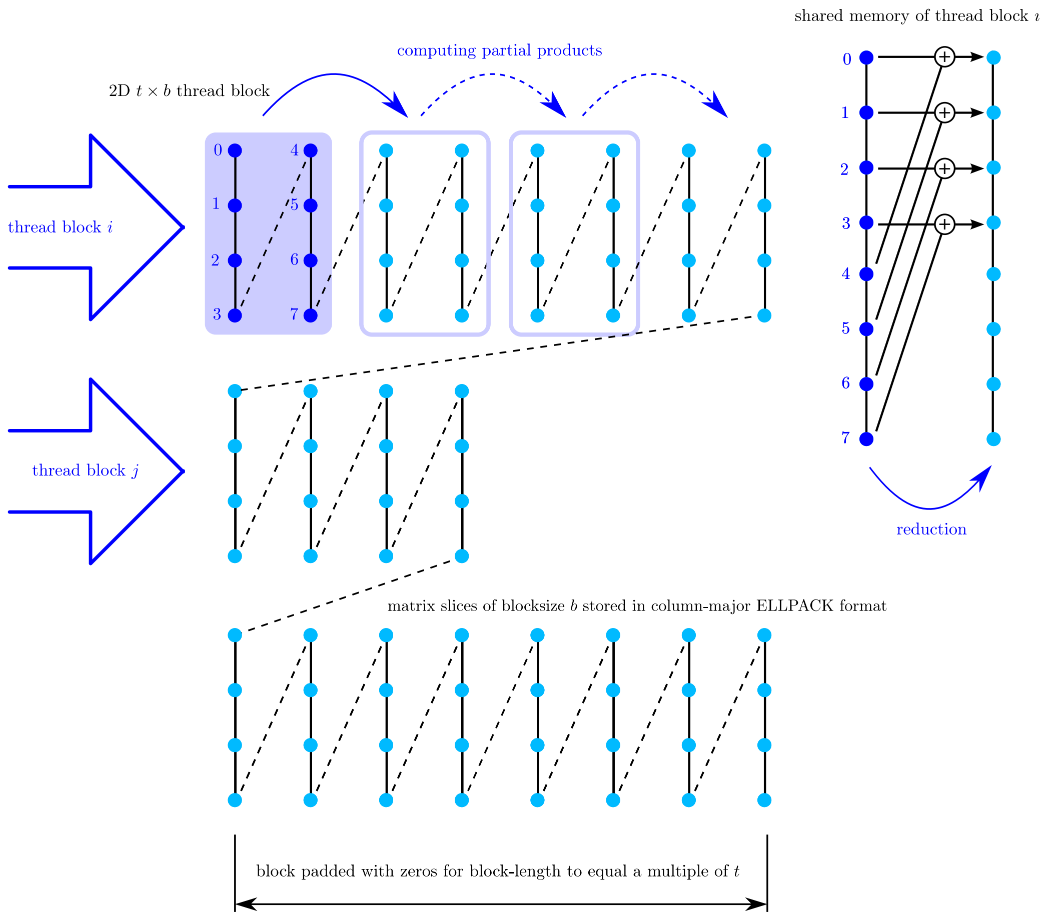

shared memory of thread block $\imath$
computing partial products
2D $t \times b$ thread block
thread block i
thread block j
0
1
2
3
4
5
6
7
reduction
matrix slices of blocksize b stored in column-major ELLPACK format
block padded with zeros for block-length to equal a multiple of t

Fig. 4. Visualization of the SELL-P memory layout and the SELL-P SpMV kernel including the reduction step using the blocksize $b = 4$ (corresponding to SELL-4), and $t = 2$ (10).

3.2. From dot product to matrix-vector multiply

NVIDIA's cuBLAS library provides an efficient routine to compute dot products on the GPU. However, as soon as multiple dot products need to be computed consecutively, performance suffers from memory access and the fact that the reduction for each vector is handled consecutively and independently from one another. This motivated us to come up with a kernel capable of computing multiple dot products at once and reducing the vectors simultaneously. Although the most common form of BiCGSTAB only requires the parallel computation of two dot products, we aim for a general approach here. As an additional use case, we consider the fact that 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 $A^{\mathsf{T}}x$, where A is a tall and skinny matrix (number of rows of A far excceeds the number of columns).

The common practice in parallel matrix-vector multiplication is to assign a fixed set of rows/columns to each processing unit for the A non-transpose/transpose cases, respectively. But this approach becomes inefficient if A consists of less rows/columns than the number of the processing units available. Especially when targeting GPUs, handling columns by threads is not suitable for the $A^{\mathsf{T}}x$ computation, as the typically used thread number will exceed the number of columns by orders of magnitude. The MAGMA library (17), developed at the Innovative Computing Lab (ICL) at the University of Tennessee, overcomes this by assigning one SMX processor to each column, and splitting each column into chunks that are then handled by different threads. To have all 13 SMXs of the K20c GPU working, the matrix needs at least 13 columns (31). 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 the fan in summation.

Using the ideas based on the dot product where computing units process in a tree-reduction fashion, we extend the implementation proposed in (5) 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 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 each reduction step, as the partial sums computed by the different thread blocks are used in the next reduction step.

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, the right side of Figure 11 in the Appendix provides the code for the `magma_dbicgmerge_reduce2` kernel.

3.3. Merging multiple arithmetic operations into one kernel

Optimizing communications in memory-bound algorithms is often impossible by using only BLAS functions. For example, the BLAS-based computation of

$$p_k := r_{k-1} + \beta\,(p_{k-1} - \omega_{k-1} v_{k-1})$$

from line 11 of Figure 1 would result in three BLAS calls (lines 4-6 in Figure 1, right). Every routine reads the data from the main memory, computes, and writes the result back. As vector operations (Level 1 BLAS) are memory-bound, the $8n+4$ memory transfers ($5n+4$ reads and $3n$ writes) limit the performance. Significant improvements can be achieved by "merging" the BLAS functions into a new kernel (see Section 4.4, Figure 8), where data movement is reduced down to the optimal for this particular case $3n+4$ reads and n writes.

Deriving a model reflecting the total data communication volume of one BiCGSTAB iteration is difficult as the memory footprint of the sparse matrix-vector products depends on the used storage format, the kernel implementation, and the matrix characteristics. While the memory volume for the matrix in a certain format is usually available (nnz floating point numbers and $nnz+n+1$ integers for CSR, see Section 3.1), the computation of the product with a vector x requires (for matrices containing off-diagonal entries) typically more than n additional memory reads, as entries of x are needed multiple times and may not all be kept in cache. Against this background, and with the motivation to keep the modularity of the SpMV, we introduce the constant C_{SpMV} reflecting the data volume associated with the matrix-vector products. Note, for the CSR-based SpMV we use in the cuBLAS reference implementation, this constant has a lower bound of $2nnz+2n$ reads and n writes assuming a well-defined linear system and the same data volume for integers and floating point numbers.

For the optimization of data transfers, we introduce the following proposition, providing a lower bound of the necessary communication volume:

Theorem 1 (Proposition)**.** *The optimal communication for a parallel implementation of the BiCGSTAB iteration from Fig. 1, which preserves the modularity of the matrix vector multiplication, that is* ***not*** *accounted for in the communication cost, is* $18n$ *where* $n \geq 2$ *is the matrix size.*

Proof. Let us assume the parallel computing platform is equipped with n ($n \geq 2$) independent processors, where n is the size of the linear system to which the iteration method is applied. Furthermore, each processor may be equipped with cache sufficiently large to keep all scalars and one component of each vector used in the algorithm in local memory. We refer to this setup as "data-optimal layout," as all vector operations can be executed in parallel without partitioning the vectors. Communication between these n processors is possible via broadcast (one-to-all messages). This allows the system to compute dot products by the processors consecutively, broadcasting their local sums, and adding any incoming partial sums to form the dot product. In this scenario, all vector updates and local sums can be computed locally without accessing the global memory. The only data read/write phases are at the beginning and end of a processing phase. These phases are defined by the sparse matrix vector products: the modularity of the SpMV building block may require a rearrangement of the processors and flushing all local vector memory (the scalars may be kept). Hence, before each SpMV, all data that will be needed in the future has to be written to the global memory, and after completion of the SpMV, the vectors for the next phase have to be read from global memory. As a result, the iteration loop of the BiCGSTAB has two vector-parallel computation phases interspersed by the SPMV kernels:

vec1**:** the computation of α and updating s (line 13-14 of Figure 1, left side),
vec2**:** the remaining computations, neglecting the SpMVs (line 16-19 + 8-11 of Figure 1, left side).

Obviously, this data-optimal layout catches the lower bound of data transfers, as any setup with more processors results in idle processors, and any setup with less processors requires one processor to load multiple entries for each vector, perform multiple broadcasts of partial sums for a dot product, and write multiple entries of each vector to global memory. For this reason, we now use the data-optimal setup to proof the proposition by mathematical induction.

•Induction Hypothesis
For a data-optimal setup of dimension n (n unknowns in the linear system, n processors $\mathscr{P}_1 \dots \mathscr{P}_n$), the minimum data transfer volume for one BiCGSTAB iteration preserving SpMV modularity is $18n$.
•Initial Step
Let $n = 2$. In the vector-parallel computation phase vec1, both processors need to read their local value of r_0, r, and v, resulting in 6 data transfers. The computation of the dot product $\alpha = \langle r_0, v\rangle$ requires two broadcasts as it gets formed out of two partial sums. Adding these, the processors form α and compute $s[i] = r[i] - \alpha \cdot v[i]$, ($i = 1,2$) in local memory. As the SpMV in line 15 requires flushing all local vector memory, 2 data transfers are necessary to write the generated s to global memory. This results in 10 data transfer units for the vec1 vector computation phase.

The data transfer cost for vec2 can be obtained similarly: While the respective components of r can be generated locally, s, t, x, p, r_0, and v have to be read from global memory, resulting in 12 memory reads for the case of $n = 2$. Four dot products ($\langle s,t\rangle$, $\langle t,t\rangle$, $\langle r,r\rangle$, $\langle r_0,r\rangle$) require 8 data transfer units, and writing x, r, and p to global memory requires another 6. By adding the 26 data transfers of vec2 to the 10 data transfers of vec1 we obtain 36 ($= 18 \cdot 2$) data transfers all together.

•Induction Step

Consider a system of $n+1$ unknowns and a subset of n processors $\mathscr{P}_1 \ldots \mathscr{P}_n$ that handle the vector entries from 1 to n. This requires $18n$ data transfers, but to compute the correct scalars, the left processor $\mathscr{P}_{n+1}$ has to broadcast the $n+1$-th partial sum of the dot products. This requires $\mathscr{P}_{n+1}$ to read the necessary data from global memory (3 reads for vec1, 6 reads for vec2), performing the broadcasts (5 in total) and writing the $n+1$-th component of s, r, x and p to global memory. Hence, 18 additional data transfers are required, resulting in a total data communication volume of $18n + 18 = 18(n+1)$.

□

Remark 1 We assumed the flushing of the local memory does not affect the scalars. This assumption may not be satisfied under some circumstances, but the additional cost of one processor writing the scalars to global memory before starting the SpMV kernel, and broadcasting the scalars after SpMV completion, results in $\mathscr{O}(1)$ additional data transfer units that may be neglected for large n.

Remark 2 For a linear system of dimension 1, the lower bound do not hold, as then the dot products does not require a reduction phase.

Subsequently, we show that the optimization introduced in Figure 2 reaches this lower bound for the given GPU hardware consisting of fewer than n processors. Using this aforementioned notation, we compare in Table 1 the total GPU memory access in one BiCGSTAB iteration. To account for the additional reduction step that may be necessary when computing dot products, we included the term $[\mathscr{O}(\log_{bs}(n))]$, which is only relevant for large vectors. Not considering the sparse matrix-vector multiplications ($2 \cdot C_{SpMV}$ per iteration), we still have $25n + 9n$ memory transfers in the cuBLAS reference implementation and $14n + 4n$ memory transactions in the accelerated version (left and right side of Table 1, respectively). Hence, we succeed in realizing the theoretical optimization potential, and under the assumption that memory reads and writes are similarly expensive, we may, depending on the dominance of the matrix-vector products and the reduction phase of the dot products, expect a performance improvement of up to 47% when replacing the cuBLAS reference implementation with the optimized code.

4. Numerical results and experiments

4.1. Experimental Setup

In this paper we pursue an incremental approach, where we support all algorithmic and conceptual modifications to the reference implementation with experimental results. We obtain those results from a Tesla K40 GPU that belongs to the Kepler line of NVIDIA's hardware accelerators with a theoretical peak performance of 1,682 GFlop/s (double precision). The host system has a theoretical peak of 333 GFlop/s, main memory size is 64 GB, and theoretical bandwidth is up to 51 GB/s. On the K40 GPU, the 12 GB of main memory, accessed at a theoretical bandwidth of 288 GB/s, 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. The implementation of all GPU kernels is realized in CUDA (32), version 6.0 (33), using a thread-block size of 256. For

BLAS-based BiCGSTAB			merged BiCGSTAB		
line[2]	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	C_{SpMV}		SpMV	C_{SpMV}	
8	$2n+\mathcal{O}(1)$	$\mathcal{O}(1)$	reduce1	$2n+[\mathcal{O}(\log_{bs}(n))]$	$1+[\mathcal{O}(\log_{bs}(n))]$
9	n	n	s_update	$2n+\mathcal{O}(1)$	n
10	$2n+\mathcal{O}(1)$	n			
11	C_{SpMV}		SpMV	C_{SpMV}	
12	$2n+\mathcal{O}(1)$	$\mathcal{O}(1)$	reduce2	$2n+[\mathcal{O}(\log_{bs}(n))]$	$1+[\mathcal{O}(\log_{bs}(n))]$
13	$2n+\mathcal{O}(1)$	$\mathcal{O}(1)$			
14	$2n+\mathcal{O}(1)$	n	xr_update+reduce3	$5n+[\mathcal{O}(\log_{bs}(n))]$	$2n+[\mathcal{O}(\log_{bs}(n))]$
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[3]	$25n+\mathcal{O}(1)$	$9n+\mathcal{O}(1)$	sum[3]	$14n+[\mathcal{O}(\log_{bs}(n))]$	$4n+[\mathcal{O}(\log_{bs}(n))]$

[2]see right of Figure 1; [3]not accounting for the SpMV kernels;

Table 1. Comparison of GPU memory access for the BLAS-based BiCGSTAB (see right of Figure 1) and the merged variant. The term $[\mathcal{O}(\log_{bs}(n))]$ reflects the additional reduction step necessary in the computation of scalar products for large problem sizes.

the experiments, we use 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 (see Figure 4.1), some key characteristics are summarized in Table 2.

4.2. Performance of the SELL-P SpMV

In (10), a comprehensive performance comparison reveals the competitiveness of the SELL-P SpMV against other highly-tuned implementations, including NVIDIA's in-house developments available via the cuSPARSE library (16). Comparison to multicore CPU approaches is available in (4). We refrain from including all options in this paper, but instead limit

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

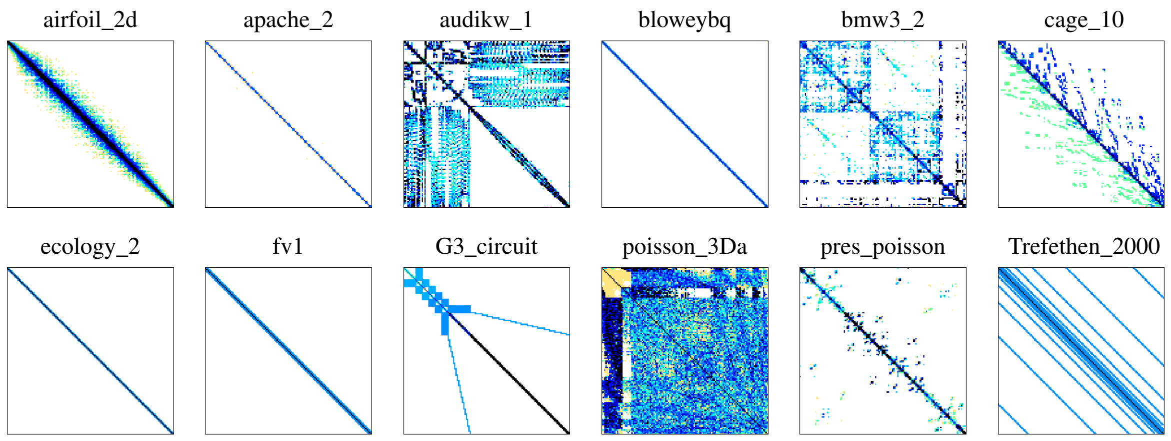


Fig. 5. Sparsity structure of the selected test matrices.

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 2. Description and properties of the test matrices.

matrix	CSR SpMV [GFlop/s]	SELL-P SpMV [GFlop/s]	speedup s	improvement ζ_M
airfoil_2d	3.52	9.93	2.81	64 %
apache_2	9.81	8.08	–	–
audikw_1	1.89	22.12	11.68	91%
bloweybq	5.16	4.70	–	–
bmw3_2	2.06	22.69	11.03	91%
cage_10	4.04	8.88	2.20	54%
ecology_2	10.04	6.02	–	–
fv1	5.47	6.20	1.13	12%
G3_circuit	6.49	5.68	–	–
poisson_3Da	2.22	8.47	3.81	74%
pres_poisson	2.84	18.35	6.45	84%
Trefethen_2000	2.17	6.45	2.98	66%
Trefethen_20000	1.45	16.75	11.52	91%

Table 3. Performance comparison between CSR-SpMV and the SELL-P SpMV, speedup and improvement obtained by replacing the CSR SpMV where beneficial.

our focus on the potential improvement obtained by replacing the standard CSR-based kernel with the SELL-P format. In Table 3 we list the performance achieved by either of the implementations, and, in the case of superior SELL-P performance, the speedup s is obtained by replacing the standard CSR SpMV. The results reveal that no overall-superior SpMV format exists, and particularly for very sparse systems, the basic CSR may provide higher performance, see results for apache_2, bloweybq, ecology_2, and G3_circuit. In a realistic scenario, comparing the SpMV performance for the target matrix allows the user to choose a suitable format.

4.3. Tuning of merged dot products

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 on the left in Figure 6 (see line labeled 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 labeled as

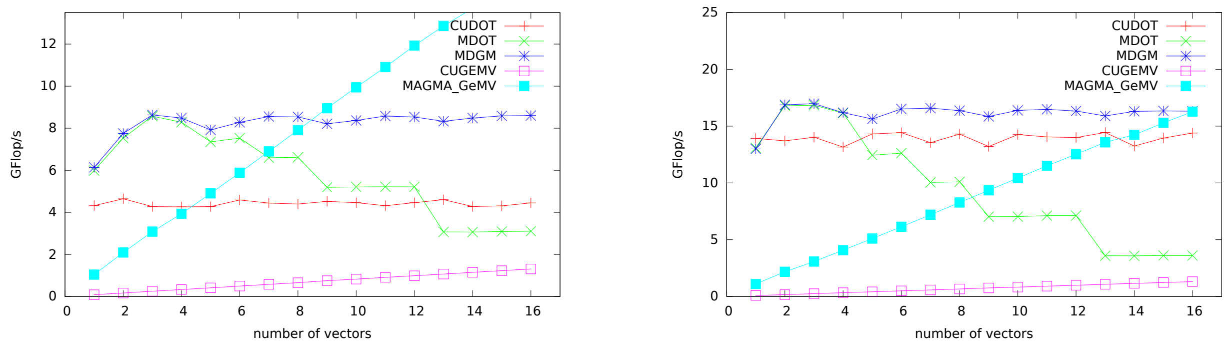


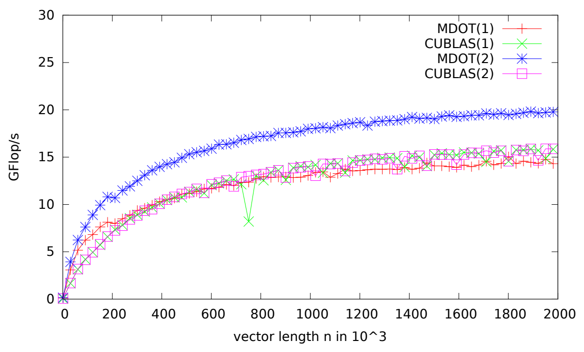
Fig. 6. Performance comparison between cuBLAS dot product, the developed simultaneous dot product implementations MDOT and MDGM, and the matrix-vector products from NVIDIA and MAGMA, respectively for a vector length of 100,000 (left) and 1,000,000 (right).

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 left of Figure 7). The comparison with the matrix-vector product kernels is interesting. 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. On the right of 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 on the left in Figure 7. The first observation is that, when computing only one dot product, the MDGM outperforms the 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` on the right of Figure 11 in the Appendix). This also impacts the memory traffic, with the memory reads rising form $2n$ to $2n+\mathcal{O}(\log_{bs}(n))$ and the writes from 1 to $\log_{bs}(n)$ for large vectors (bs denotes the block-size used in the GPU kernel implementation).

When adding a second dot product with one vector shared by both operations (see data labeled MDOT(2) and cuBLAS(2) in Figure 7, left) 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 using a compiler capable of detecting the reuse of one vector.

4.4. Reducing communication through merged kernels

Figure 8 shows the new "merged" kernel updating the vector p that we discussed in Section 3.3. As the data movement is reduced down to $3n+4$ reads and n writes (50% reduction against the set of cuBLAS routines), we expect an asymptotic speedup of 2, which is reflected on the right side in Figure 8 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 on the right in Figure 8.



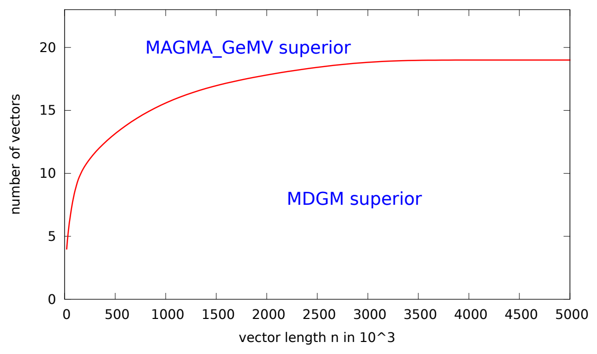


Fig. 7. Left: Performance comparison (double precision) between cuBLAS and MDOT executing 1 and 2 vector products. Right: Superiority areas of MDGM and MAGMA's matrix-vector kernel.

```
// cuBLAS
cublasDscal(n, beta, p, 1 );
cublasDaxpy(n, omega*beta, v, 1 , p, 1);
cublasDaxpy(n, 1.0, r, 1, p, 1);


// MAGMA
__global__ void
magma_dbicgmerge_p_update
         ( int n, double *skp, double *v,
                  double *r, double *p ){
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    double beta=skp[1], omega=skp[2];
    if( i<n )
        p[i] = r[i] + beta*(p[i]-omega*v[i]);
}
```

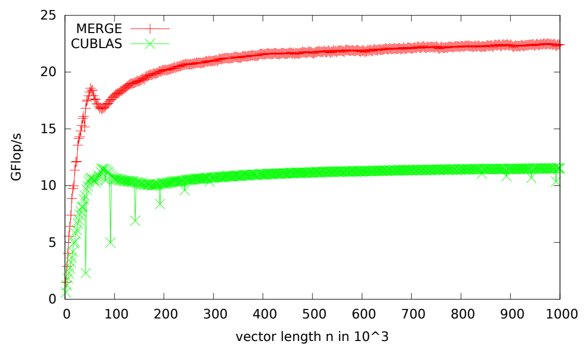


Fig. 8. Left: cuBLAS library calls and an algorithm-specific kernel (labeled *MAGMA*) for the update of vector p (see line 11 in Figure 1). Right: Comparison of the respective performance.

4.5. Experimental comparison with performance model guiding the optimizations

In the previous sections, we have proposed different optimizations and 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 that the modifications are expected to achieve in experiments using the set of test matrices introduced in Section 4.1. In Table 4, we profile the cuBLAS reference implementation for the different test matrices. Step by step we now develop a model providing estimations for the savings rendered by the modifications we proposed in the previous sections.

An almost problem-independent improvement is the reduction of memory transfers we realized in Section 3.3, where we merged multiple arithmetic operations into one kernel. While we left the sparse matrix-vector product untouched, we aggregated vector updates, and, whenever possible, included the first part of a scalar product computation. Hence, an accurate model would require distinguishing between the "reading" and the "reduction" phase of dot products. However, we argue that the "reduction" part is usually dominating the computational cost, and exclude the dot products from the memory-improved parts.

Against the background of estimating the memory savings for the remaining parts as $\eta_{\text{memory}} = \frac{25n+9n}{14n+4n} \approx 47\%$ due to reduced memory transfers in the memory-bound algorithm, a very simple model estimating the expected savings is given by:

$$P_{\text{memory}} = 1 - \frac{T_{MERGE}}{T_{cuBLAS}} = (1 - \text{SpMV} - \text{dot}) \times \eta_{\text{memory}}. \tag{2}$$

Ordering the matrices according to the combined SpMV and dot dominance (see Table 4), we visualize the savings obtained by the reduced memory transfers in Figure 10 (see 'memory'). This model is independent of the characteristics of the target matrix and the applied sparse matrix-vector kernel, as we did not merge the SpMV with any other operation to maintain the genericness of the algorithm. The size of the problem, already impacting the dominance of (SpMV+dot) operations, becomes even more important as soon as we extend the model by also accounting for the improvements rendered by the aggregated dot product (MDOT) we proposed in Section 3.2, which is capable of computing multiple dot products simultaneously. The improvements when switching from cuBLAS to MDOT are dependent on the vector size, and in order to quantify them, 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 left of 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 on the right of Figure 9):

$$\mathscr{F}(n) = \frac{1}{100}\left(\frac{1}{10^{-7} \times n + 0.018} + 8.0\right). \tag{3}$$

We now combine the improvements due to data locality of MDOT in (4), 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. Using this equation, it is possible to predict the runtime savings when switching from the cuBLAS to the merged implementation without changing the sparse matrix-vector kernel, when having knowledge about the matrix size exclusively (visualized in Figure 10 as 'memory+dot').

$$P_{\text{memory+dot}} = \eta_{\text{memory}} \times (1 - \text{SpMV} - \text{dot}) + \mathscr{F}(n) \times \text{dot}. \tag{4}$$

Predicting the improvements obtained by also replacing the CSR-SpMV with a more sophisticated matrix-vector kernel requires detailed knowledge of the characteristics of the sparse matrix. Although approaches exist to classify sparse matrices

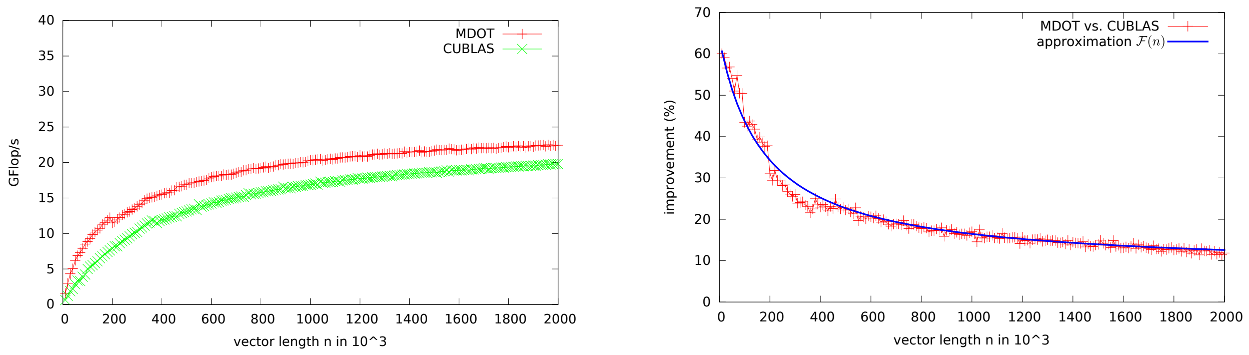


Fig. 9. Performance comparison (left) and size-dependent runtime improvement (right) between cuBLAS and MDOT in the sequence of dot products in BiCGSTAB.

for SpMV performance prediction (see e.g. (34)), we limit our model to the experimental SpMV performance analysis providing the data given in Table 3. Combining the listed improvements obtained by switching to the SELLP-SpMV with the time fraction spent on the sparse matrix-vector kernel (see Table 4), we can extend the model to account for the reduced memory transfers, the aggregated dot products, and the improved SpMV:

$$P_{\text{memory+dot+SpMV}} = (1 - \text{SpMV} - \text{dot}) \times \eta_{\text{memory}} + \mathscr{F}(n) \times \text{dot} + \text{SpMV} \times \zeta_M. \tag{5}$$

Comparing the predictions $P_{\text{memory+dot+SpMV}}$ visualized as 'memory+dot+SpMV' in Figure 10 with data obtained from runtime experiments based on 1000 BiCGSTAB iterations (labeled as 'experimental'), we observe that the model is in most cases able to provide very good estimations, better than the linear model based exclusively on the memory improvement, and the model $P_{\text{memory+dot}}$.

The acceleration shown in Figure 10 breaks down into improvements coming from applying the kernel fusion technique to reduce the communication, and, for some matrices, the improvements coming from the faster SpMV kernel. A detailed analysis on these contributions can be found in Table 5. Although double precision accuracy is usually considered mandatory for scientific computing, we also include single precision results, as techniques like iterative refinement (35; 36) allow us to leverage the single-precision performance of GPUs while maintaining double-precision accuracy of the solution (37). As the CSR and SELL-P data structure for the matrices consist of integer and floating point arrays, the SpMV acceleration for single precision can be quite different than the double precision acceleration. At the same time, the consistent communication-related performance improvements can be used to verify the model's accuracy.

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 matrix characteristics, the merged version based on either the CSR or the SELL-P matrix-vector kernel achieves runtime reductions between 20 and 90%. The fact that these improvements are distributed over the spectrum of the SpMV dominance shows that the modifications have to go hand-in-hand to ensure problem-independent performance improvement.

5. 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

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 4. Profiling of the cuBLAS reference implementation, all timings are for 1000 BiCGSTAB iterations.

	Double precision			Single precision		
Matrix	fusion	SpMV	total			
airfoil_2d	25.54	33.52	59.066	22.83	38.20	62.03
apache_2	14.63	-	14.62	16.61	-	16.61
audikw_1	0.00	90.94	90.94	0.03	89.09	88.63
bloweybq	61.35	-	61.35	56.23	-	56.23
bmw3_2	0.48	89.05	89.53	0.01	81.93	81.71
cage_10	35.03	19.63	54.66	31.18	30.86	62.04
ecology_2	17.49	-	17.49	13.73	-	13.73
fv1	45.16	3.38	48.54	40.55	3.21	43.77
G3_circuit	16.44	-	16.44	13.43	-	13.43
poisson_3Da	16.77	51.21	67.98	18.38	46.68	65.07
pres_poisson	10.98	66.82	77.80	9.00	68.91	77.91
Trefethen_2000	42.88	17.52	60.40	42.86	28.17	71.04
Trefethen_20000	12.83	66.47	79.31	11.22	74.79	86.01

Table 5. Performance improvement breakdown [%] for the kernel fusion and the faster SpMV kernel using single or double precision.

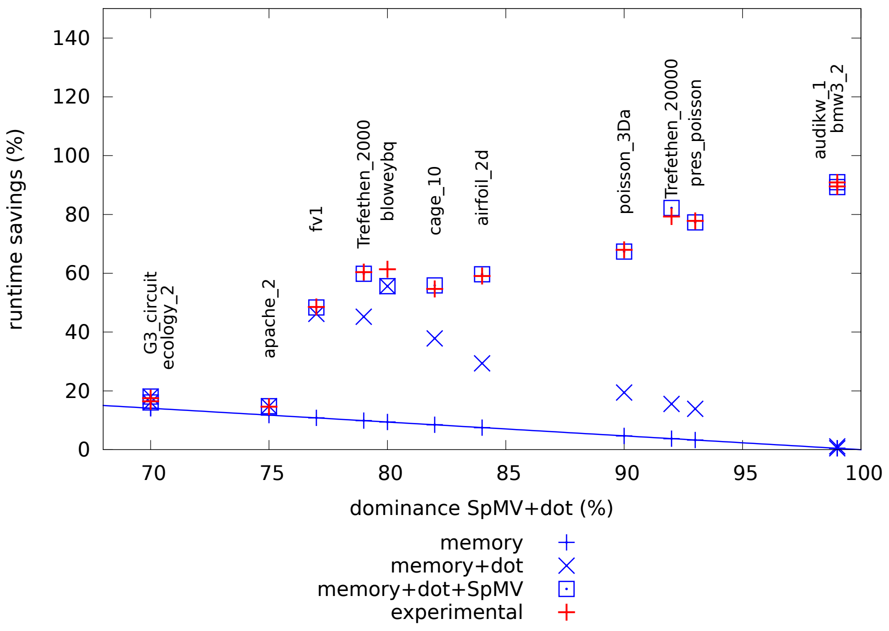


Fig. 10. Estimated (blue) and experimental (red) performance improvement obtained by replacing the cuBLAS reference implementation with the reformulated version depending on the matrix-vector kernel dominance in the original code.

pressure from the PCI connection, uses new highly-efficient dot product kernels able to reduce multiple dot products simultaneously, and replaces the standard CSR-based matrix-vector product by the SELL-P kernel where beneficial. 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 improvements from 20% to 90% 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, the faster execution due to our new and optimized dot product, and the accelerated SpMV. While we focused on BiCGSTAB, the necessity of method-specific kernels to achieve high performance on GPUs also applied to other Krylov subspace methods. Deriving models similar to ours may provide a-priori insight into whether a specific solver is suitable for a custom-designed GPU implementation when considering the achieved performance improvements. Future research should focus on including preconditioner techniques, as preconditioning is often the key to efficiency when solving sparse linear systems via Krylov subspace methods. Also, the integration of a matrix powers kernel may yield additional benefits due to reduced communication and synchronization.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, Department of Energy grant No. DE-SC0010042, and the Russian Scientific Fund, Agreement N14-11-00190.

References

[1] Saad Y. Iterative Methods for Sparse Linear Systems. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics; 2003.

[2] Kogge *et al* P. ExaScale Computing Study: Technology Challenges in Achieving ExaScale Systems; 2008.

[3] The Top 500 List, `http://www.top.org/`;.

[4] Anzt H, Tomov S, Dongarra J. Energy Efficiency and Performance Frontiers for Sparse Computations on GPU Supercomputers. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores. PMAM

’15. New York, NY, USA: ACM; 2015. p. 1–10. Available from: http://doi.acm.org/10.1145/2712386.2712387.

[5] Aliaga JI, Perez J, Quintana-Orti ES, Anzt H. Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs. In: Parallel Processing (ICPP), 2013 42nd International Conference on; 2013. p. 320–329.

[6] Anzt H, Sawyer W, Tomov S, Luszczek P, Yamazaki I, Dongarra J. Optimizing Krylov Subspace Solvers on Graphics Processing Units. In: 28th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2014); 2014. .

[7] Bell N, Garland M. Efficient Sparse Matrix-Vector Multiplication on CUDA [NVIDIA Technical Report]; 2008.

[8] Monakov A, Lokhmotov A, Avetisyan A. Automatically Tuning Sparse Matrix-vector Multiplication for GPU Architectures. In: Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers. HiPEAC’10. Berlin, Heidelberg: Springer-Verlag; 2010. p. 111–125. Available from: http://dx.doi.org/10.1007/978-3-642-11515-8_10.

[9] Kreutzer M, Hager G, Wellein G, Fehske H, Bishop AR. A unified sparse matrix data format for modern processors with wide SIMD units. CoRR. 2013;abs/1307.6209.

[10] Anzt H, Tomov S, Dongarra J. Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs. University of Tennessee; 2014. ut-eecs-14-727.

[11] Dorostkar A, Lukarski D, Lund B, Neytcheva M, Notay Y, Schmidt P. CPU and GPU Performance of Large Scale Numerical Simulations in Geophysics. In: Lopes L, Źilinskas J, Costan A, Cascella R, Kecskemeti G, Jeannot E, et al., editors. Euro-Par 2014: Parallel Processing Workshops. vol. 8805 of Lecture Notes in Computer Science. Springer International Publishing; 2014. p. 12–23. Available from: http://dx.doi.org/10.1007/978-3-319-14325-5_2.

[12] Li R, Saad Y. GPU-accelerated preconditioned iterative linear solvers. The Journal of Supercomputing. 2013;63(2):443–466. Available from: http://dx.doi.org/10.1007/s11227-012-0825-3.

[13] Lukash M, Rupp K, Selberherr S. Sparse Approximate Inverse Preconditioners for Iterative Solvers on GPUs. In: HPC ’12: Proceedings of the 2012 Symposium on High Performance Computing. San Diego, CA, USA: Society for Computer Simulation International; 2012. p. 1–8.

[14] PARALUTION;. http://www.paralution.com/.

[15] ViennaCL;. http://viennacl.sourceforge.net/.

[16] NVIDIA. CUSPARSE LIBRARY; 2013.

[17] MAGMA 1.6.1; 2015. http://icl.cs.utk.edu/magma/.

[18] Filipovic J, Madzin M, Fousek J, Matyska L. Optimizing CUDA Code By Kernel Fusion—Application on BLAS. CoRR. 2013;abs/1305.1183. Available from: http://arxiv.org/abs/1305.1183.

[19] van der Vorst H. 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. 1992;13(2):631–644. Available from: http://epubs.siam.org/doi/abs/10.1137/0913035.

[20] Hestenes MR, Stiefel E. Methods of Conjugate Gradients for Solving Linear Systems. Journal of Research of the National Bureau of Standards. 1952 Dec;49:409–436.

[21] Hoemmen MF. Communication-avoiding Krylov subspace methods. EECS Department, University of California, Berkeley; 2010. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-37.html.

[22] Ashcraft C, Eisenstat SC, Liu JWH. A Fan-in Algorithm for Distributed Sparse Numerical Factorization. SIAM J Sci Stat Comput. 1990 May;11(3):593–599. Available from: http://dx.doi.org/10.1137/0911033.

[23] Braess D. Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics. vol. 3. Cambridge University Press; 2007.

[24] Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition. Philadelphia, PA: SIAM; 1994.

[25] Sawyer W. CUSPARSE/CUBLAS Example: BiCGStab Iterative Solver for Non-symmetric Linear Systems; 2011. https://hpcforge.org/plugins/mediawiki/wiki/gpu-training/index.php/Main_Page.

[26] Yamazaki I, Anzt H, Tomov S, Hoemmen M, Dongarra J. Improving the Performance of CA-GMRES on Multicores with Multiple

GPUs. In: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium. IPDPS '14. Washington, DC, USA: IEEE Computer Society; 2014. p. 382–391. Available from: `http://dx.doi.org/10.1109/IPDPS.2014.48`.

[27] NVIDIA. CUDA C Best Practices Guide;. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[28] Williams S, Bell N, Choi J, Garland M, Oliker L, Vuduc R. Sparse matrix vector multiplication on multicore and accelerator systems. In: Kurzak J, Bader DA, Dongarra J, editors. Scientific Computing with Multicore Processors and Accelerators. CRC Press; 2010. .

[29] Buluç A, Williams S, Oliker L, Demmel J. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In: Proc. IPDPS; 2011. p. 721–733.

[30] Choi JW, Singh A, Vuduc RW. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '10. New York, NY, USA: ACM; 2010. p. 115–126. Available from: `http://doi.acm.org/10.1145/1693453.1693471`.

[31] Corporation N. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110; 2012. Whitepaper.

[32] NVIDIA CUDA Compute Unified Device Architecture Programming Guide; 2009.

[33] NVIDIA. NVIDIA CUDA TOOLKIT V6.0; 2013.

[34] Malossi CI, Ineichen Y, Bekas C, Curioni A, Quintana-Ortí ES. Performance and energy-aware characterization of the sparse matrix-vector multiplication on multithreaded architectures. In: 3rd Int. Workshop on Power-aware Algorithms, Systems, and Architectures–ICPP PASA 2014; 2014. .

[35] Buttari A, Dongarra JJ, Langou J, Langou J, Luszczek P, Kurzak J. Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. Int J of High Perf Comp & Appl. 2007;21(4):457–486.

[36] Baboulin M, Buttari A, Dongarra JJ, Langou J, Langou J, Luszczek P, et al. Accelerating Scientific Computations with Mixed Precision Algorithms. Computer Physics Communications. 2009;180(12):2526–2533.

[37] Anzt H, Heuveline V, Rocker B. Mixed precision error correction methods for linear systems Convergence analysis based on Krylov subspace methods. In: Jonasson K, editor. PARA 2010, Part II, LNCS 7134. Springer, Heidelberg; 2010. p. 237–248.

Author biographies

Hartwig Anzt is a PostDoctoral researcher in Jack Dongarra's Innovative Computing Lab (ICL) at the University of Tennessee. He received his Ph.D. in mathematics from the Karlsruhe Institute of Technology (KIT) in 2012. Dr. Anzt's research interests include simulation algorithms, sparse linear algebra, hardware-optimized numerics for GPU-accelerated platforms, communication-avoiding and asynchronous methods, and power-aware computing.

William Sawyer is a computational scientist at the Swiss National Supercomputing Centre (CSCS), in Lugano, Switzerland, a branch of the Swiss Federal Institute of Technology, Zurich (ETH). , and supports CSCS's customers from the Geosciences. He completed his dissertation on "Efficient Numerical Methods for the Shallow Water Equations on the Sphere" in 2006 at the ETH Zurich. He has an extensive research record in the field of parallel applications and algorithms for HPC platforms, in particular for numerical weather prediction, climate models and data assimilation systems.

Stanimire Tomov is a Research Director in the Innovative Computing Laboratory (ICL) at the University of Tennessee. Tomov's research interests are in parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). He has been involved in the development of numerical algorithms and software tools in a variety of fields ranging from scientific visualization and data mining to accurate and efficient numerical solution of PDEs. Currently, his work is concentrated on the development of numerical linear algebra libraries for emerging architectures for HPC, such as heterogeneous multicore processors, graphics processing units (GPUs), and Many Integrated Core (MIC) architectures.

Piotr Luszczek is a Research Director at the University of Tennessee. His research interests include large scale parallel algorithms, numerical analysis, and high-performance computing (HPC). He has been involved in the development and maintenance of widely used software libraries for numerical linear algebra. In addition, he specializes in computer benchmarking of supercomputers using codes based on linear algebra, signal processing, and PDE solvers.

Jack Dongarra holds appointments at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award (2004), is the recipient of the first IEEE Medal of Excellence in Scalable Computing (2008), the first SIAM Special Interest Group on Supercomputing's award for Career Achievement (2010), and the IEEE IPDPS 2011 Charles Babbage Award. He is a fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.

Appendix

```
void sellp_spmv( int n, int bs,
                 int t, double alpha,
                 int *rowptr, int *colind,
                 double *values, double *x,
                 double beta, double *y ) {

    // t threads assigned to each row
    // thread in row:
  int idx = threadIdx.x;
    // local row:
  int idy = threadIdx.y;
    // first element to be accessed:
  int ldx = idy * bs + idx;
    // global block index:
  int bdx = blockIdx.y*gridDim.x+blockIdx.x;
    // global row index:
  int row = bdx * bs + idx;
    // define shared memory:
  extern __shared__ double sm[ ];

  if(row < n ){
    double dot = 0.0;
    int offset = rowptr[ bdx ];
      // total number of threads:
    int block = bs * t;
      // elements each thread handles:
    int max_ = (rowptr[ bdx+1 ]-offset )
                                    / block;

        // partial product loop unrolled:
    int kk, i1, i2;
    double x1, x2, v1, v2;
    d_colind += offset + ldx ;
    d_val += offset + ldx;
    for (kk = 0; kk < max_-1 ; kk+=2){
      i1 = colind[ block*kk ];
      i2 = colind[ block*kk + block ];
      x1 = x[ i1 ];
      x2 = x[ i2 ];
      v1 = values[ block*kk ];
      v2 = values[ block*kk + block ];

      dot += v1 * x1;
      dot += v2 * x2;
    }
      // maybe one additional step:
    if (kk<max_){
       x1 = d_x[ d_colind[ block*kk ] ];
       v1 = d_val[ block*kk ];
       dot += v1 * x1;
    }
      // write result to shared memory:
    sm[ ldx ]  = dot;
    __syncthreads();

      // reduction in shared memory:
    if(idy < 4 ){
      sm[ ldx ] += sm[ ldx+bs*4 ];
      __syncthreads();
      if(idy < 2 )
        sm[ ldx ] += sm[ ldx+bs*2 ];
      __syncthreads();
      if(idy == 0 ) {
        y[ row ] = alpha *
          (sm[ ldx ]+sm[ ldx+bs*1 ])
                         + beta*y[ row ];
      }
    }
  }
}
```

```
// block reduction for k vectors
__global__ void
magma_reduce( int Gs, int n, int k,
        double *vsm, double *vsm2 ){

  // define shared memory:
  extern __shared__ double sm[];
  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;
  sm[idx+j*BS] = 0.0;
  while (i < Gs ) {
    sm[ idx+j*BS  ] += vsm[ i+j*n ];
    sm[ idx+j*BS  ] += ( i+BS< Gs ) ?
             vsm[ i+j*n+BS ] : 0.0;
    i += gridSize;
  }
  }
  __syncthreads();
  if ( idx < 64 )
  for( int j=0; j<k; j++)
    sm[ idx+j*BS ] += sm[ idx+j*BS+64 ];

  if( idx < 32 ){
  volatile double *sm2 = sm;
  for( int j=0; j<k; j++){
    sm2[ idx+j*BS ] += sm2[ idx+j*BS+32 ];
    sm2[ idx+j*BS ] += sm2[ idx+j*BS+16 ];
    sm2[ idx+j*BS ] += sm2[ idx+j*BS+8 ];
    sm2[ idx+j*BS ] += sm2[ idx+j*BS+4 ];
    sm2[ idx+j*BS ] += sm2[ idx+j*BS+2 ];
    sm2[ idx+j*BS ] += sm2[ idx+j*BS+1 ];
  }
  }
  if ( idx == 0 )
  for( int j=0; j<k; j++)
    vsm2[ blockidx.x+j*n ] = sm[ 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. 11. Left: SpMV kernel implementation for the SELL-P sparse matrix format for $t = 8$, see (10). Right: Algorithm-specific kernel implementation for `magma_dbicgmerge_reduce2` using a block size of 256, see (6).