Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–11
© The Author(s) 2016
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342016646844
hpc.sagepub.com
SAGE

Optimization and performance evaluation of the IDR iterative Krylov solver on GPUs

Hartwig Anzt[1], Moritz Kreutzer[2], Eduardo Ponce[1], Gregory D Peterson[1], Gerhard Wellein[2] and Jack Dongarra[1,3,4]

Abstract
In this paper, we present an optimized GPU implementation for the induced dimension reduction algorithm. We improve data locality, combine it with an efficient sparse matrix vector kernel, and investigate the potential of overlapping computation with communication as well as the possibility of concurrent kernel execution. A comprehensive performance evaluation is conducted using a suitable performance model. The analysis reveals efficiency of up to 90%, which indicates that the implementation achieves performance close to the theoretically attainable bound.

Keywords
Induced dimension reduction (IDR), GPU, co-design, kernel fusion, kernel overlap, roofline performance model

1 Introduction

Krylov subspace solvers (Saad, 2003) are among the most popular methods for iteratively solving large sparse linear systems of the form $Ax = b$. Given that an increasing number of computer systems are equipped with hardware accelerators such as GPUs (Bergman et al., 2008; top), significant efforts are spent on investigating how these methods can be designed to benefit from the computing power of the accelerators. Possible paths range from outsourcing individual computations to the device, to porting the complete algorithm to the accelerator. The algorithms typically arise as combination of a sparse matrix vector product that is generating the Krylov subspace (Saad, 2003), and a set of basic linear algebra level 1 subprograms (BLAS1) operations. Hence, a straightforward way of using GPUs is to offload all matrix and vector computations to the device using the BLAS1 functions. In many cases, this results in significant acceleration of the algorithm (Dorostkar et al., 2014). However, even larger improvements are often available when replacing the standard BLAS operations with application-specific kernels that keep data in local memory as much as possible. This concept of "kernel fusion", in particular, pays off as the algorithms are typically memory-bound, and merging multiple linear algebra routines into a single kernel reduces the pressure on the memory bandwidth (Anzt et al., 2015b). Aside from that, significant research is also looking into the acceleration of the sparse matrix-vector product, as this is often the most time consuming part of the algorithms. Although the advantages from overlapping different operations are typically negligible for entirely memory-bound algorithms, some Krylov solvers allow for scheduling data-independent operations in parallel, and can, for settings where the memory bandwidth is not saturated, benefit from concurrent kernel execution. In Anzt et al. (2015a), the benefits of kernel fusion and concurrent kernel execution have been investigated for a GPU implementation of the induced dimension reduction (IDR, Sonneveld and van Gijzen, 2009) algorithm. In this paper, we extend the previous results by applying additional tuning steps and analyzing the runtime performance against a performance model. This allows us to identify performance bottlenecks, and to quantify the GPU utilization efficiency.

We structure the rest of the paper as follows. First, in Section 2, we provide an overview of related work on

[1]University of Tennessee, Knoxville, USA
[2]University of Erlangen-Nuremberg, Germany
[3]Oak Ridge National Laboratory, USA
[4]University of Manchester, UK

Corresponding author:
Hartwig Anzt, University of Tennessee, 1122 Volunteer Blvd, Suite 203, Claxton, Knoxville, TN 37996, USA.
Email: hanzt@icl.utk.edu

strategies for accelerating Krylov subspace methods on GPUs. We then briefly review the IDR algorithm and its variants in Section 3. In Section 4, we discuss the optimization steps we apply to the GPU implementation: that is, the fusion of multiple linear algebra operations into a single kernel, the design of a GeMV kernel for extremely tall-and-skinny matrices, and the possibility of overlapping different computations by running multiple GPU kernels in concurrent fashion. We then introduce, in Section 5, the experiment set-up and the test matrices along with some key characteristics that help us understand the performance results. A roofline model, serving as a performance guideline, is introduced in Section 6. It is based on a theoretical analysis of the IDR implementation, and the quantification of the performance-limiting GPU characteristics such as the memory bandwidth. In Section 7, we compare the execution times of the optimized IDR implementation with the projected runtimes, quantify the performance gap for different test cases, and evaluate the benefit of concurrent kernel execution. We conclude in Section 8.

2 Related Work

Several open-source software packages provide GPU-support for Krylov subspace solvers (MAGMA, b; vie, 2015; cus, 2015; MAGMA, a; Kreutzer et al., 2015). For this class of algorithms, one can often obtain a performance gain compared to CPU implementations which directly reflects the architectural benefits (Dorostkar et al., 2014; Li and Saad, 2013; Lukash et al., 2012). Specifically for the IDR algorithm, a first evaluation of GPU-acceleration was investigated in Knibbe et al. (2011). The evaluated implementation, however, did not yet contain optimizations such as the concept of merging multiple linear algebra operations into one single algorithm-specific kernel. This strategy and its performance impact has already been well investigated for other algorithms. In Filipovic et al. (2013), the authors have shown that kernel fusion can be realized for certain BLAS1 and dense BLAS2 operations by using a source-to-source compiler. No automatic fusion of sparse linear algebra operations is addressed, however. In Tabik et al. (2014), the authors combine CUDA kernels in iterative sparse linear system solvers. Explicit kernel coding was used in Aliaga et al. (2013), 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 Anzt et al. (2015b) this idea was transferred to the BiCGSTAB algorithm, and combined with the acceleration of the sparse matrix vector product. Also, a general model estimating the savings was introduced. In Wang et al. (2010), the authors take a structured approach to improve performance and energy efficiency by way of kernel fusion. Kernel fusions are categorized into the classes “inner thread”, “inner thread block”, and “inter thread block”, and their effects on performance and energy efficiency are investigated by using two general benchmarks. For sparse linear system solvers, a deeper analysis on the first category can be found in Aliaga et al. (2015), where a precise characterization of the kernels and the possibility of merging them into one single kernel is presented. Research quantifying the advantages of concurrent kernel execution has primarily focused on compute-intense operations. Larger improvement can be expected if other resources than the memory bandwidth bound the performance. A comprehensive analysis for different microbenchmarks can be found in Wang et al. (2011). Scientifically relevant operations like merge-sort and convolution kernels were addressed in Gregg et al. (2012). In Jiao et al. (2015), the authors investigate the interplay between concurrent kernel execution and dynamic voltage and frequency scaling (DVFS), and quantify the impact on energy efficiency. Finally, Anzt et al. (2015a) applied both optimization techniques to the IDR algorithm, and evaluated the performance of the resulting algorithm with respect to a baseline implementation consisting of BLAS library function calls.

3 Induced dimension reduction

The induced dimension reduction (IDR) algorithm is a robust framework for deriving iterative solvers for general linear systems of equations. It is based on the Krylov subspace idea, and was first introduced by P Sonneveld and MB Gijzen in Sonneveld and van Gijzen (2009). Numerous variants exist to enhance the solver’s convergence, stability or parallelism level (Simoncini and Szyld, 2010; Van Gijzen and Sonneveld, 2011; Rendel et al., 2013). However, they all share the idea of exploiting the IDR central theorem (Sonneveld and van Gijzen, 2009) that allows the use of a finite series of nested linear subspaces G_j of decreasing dimension to obtain a solution in no more than N steps for a $N \times N$ matrix.

One popular variant is IDR(s), which can be constructed by considering s independent, standard normally distributed, shadow vectors $p_1, p_2, \ldots, p_s$ to solve a smaller system of equations based on the iterative residuals (Van Gijzen and Sonneveld, 2011). The smaller system represents a set of polynomials that force the generated residuals to be in subspaces G_j, which thus enforces the convergence of the solution after, at most N, dimension-reduction steps.

An improved variant is IDR(s)-biortho including smoothing (Van Gijzen and Sonneveld, 2011). The approach uses the iteration residuals with the assumption that each residual is included in the next reduced subspace G_{j+1}. Convergence can be accelerated by exploiting biorthogonality properties between subspaces and applying iterative refinements. Incorporating the residual smoothing using the technique developed by Hestenes and Stiefel Hestenes and Stiefel (1952) results in a monotonically decreasing residual norm (van Gijzen et al., 2015). Although smoothing does not accelerate the convergence, it is often attractive for production of code, which justifies the overhead of an additional dot product. The implementation we consider features residual smoothing as an option. We base our experimental analysis, in this paper, on a setting in which smoothing is enabled. A comprehensive collection of research efforts, and a more detailed derivation of the algorithm, can be found in idr.

4 Optimizing IDR(s) on GPUs

The pseudocode for a basic implementation of the IDR(s)-biortho algorithm using standard BLAS function calls can be found in Anzt et al. (2015a). In Figure 2, we outline the optimized version that we use in the performance analysis in Section 7. The most relevant modifications improving performance are the following.

- **Kernel fusion.** An important optimization step in memory-bound algorithms is the fusion of BLAS functions into algorithm-specific kernels. Based on the classification proposed in Aliaga et al. (2015), we identify kernels that can be mapped. We then ignore all kernels that have no data dependency and all kernels that do not share any data beside scalars. Also, we do not consider fusing the sparse matrix vector product generating the Krylov subspace with any other operations. The motivation is to maintain the algorithm's flexibility with respect to switching between different sparse matrix vector kernels. The remaining operations that allow and potentially benefit from kernel fusion are the smoothing operations (line 35, 40, 54, and 59 in Figure 2). For those, we design algorithm-specific kernels that operate on data local in the multiprocessor's cache.
- **Merged dot product.** The IDR(s) algorithm, as given in Figure 2, requires multiple gemv calls with a matrix of size $n \times s$, where n is the problem size and s is the shadow space dimension. Typically, n is much larger than s, and handling this operation with a special kernel based on the strategy of interleaving multiple dot products can be much faster Anzt et al. (2015b). In Figure 1, we compare the performance achieved for this operation using different kernels with the performance expected from a roofline performance analysis, as presented in Section 6. The tested scenario is a matrix of size $5,000,000 \times s$ with $s \leq 32$. The results show that the `magma_dmdotc` achieves, for all s, the best performance, which is about 60% of the theoretical bound. Given this observation, we replace, in the optimized IDR(s), all `gemvs` of this kind with the `magma_dmdotc` function (see lines 3, 29, 36, 49, and 55 in Figure 2). In Collignon and van Gijzen (2011), the authors propose to also merge all dot products involving the columns of P (corresponding to lines 18 and 29 in Figure 2). In the resulting algorithm, every cycle has $s + 1$ combined inner products involving the columns of P. Although not realized in the GPU implementation we consider, this strategy may provide some additional improvement.
- **Reducing host-device communication.** We reduce the communication between host and device to a level where it no longer has an impact on performance. Precisely, only some scalar values are communicated; all vector data and the majority of scalar values are kept in GPU memory only.

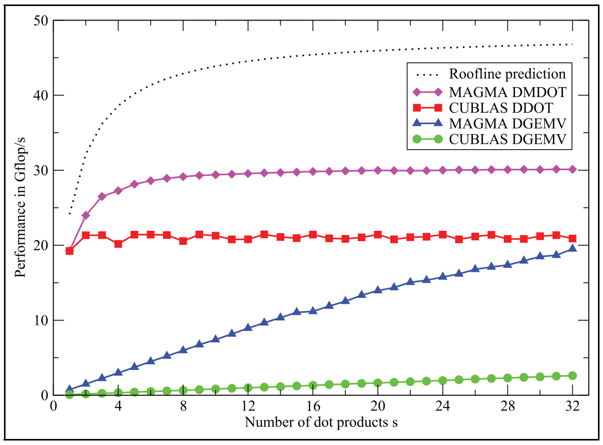


Figure 1. Performance comparison of different implementations for solving s dot products $x_i^T y$ with $i = 1, \ldots, s$ and x_i, y vectors of length 5,000,000.

Apart from these optimizations, we evaluate also, in Section 7, the potential of concurrent kernel execution and overlapping computation with device-host communication. Given the background that the IDR(s) algorithm arises as a combination of memory-bound operations, the benefit of this optimization is expected to be small: the memory bandwidth is the performance-limiting factor for each kernel, and thus makes it impossible to execute operations on the unused GPU resources. Performance improvements can only be expected if the parallelism, reflected in the number of active GPU threads, is too small to saturate the memory bandwidth. As we communicate only a few scalars between host and device, overlapping communication and computation is expected to bring relevant effects only for computationally cheap problems. In Figure 3, we visualize the data dependencies in one outer IDR(s)

```
do {
  numiter++;
  magma_dmdotc(n,s,P.val,r.val,f.val);    // f = P’ r
  for (k = 0, sk = s; k < s; ++k, --sk) { // shadow space loop
    // solve small system: f(k:s) = M(k:s,k:s) c(k:s)
    magma_dcopy(sk,&f.val[k],&c.val[k]);
    magma_dtrsv(’L’,’N’,’N’,sk,&M.val[k*M.ld+k],M.ld,&c.val[k]);
    magma_dcopy(n,r.val,v.val);
    // v = r - G(:,k:s) c(k:s)
    magma_dgemv(’N’,n,sk,-1.,&G.val[k*G.ld],&c.val[k],1.,v.val);
    // U(:,k) = ω * v + U(:,k:s) c(k:s)
    magma_dgemv(’N’,n,sk,1.,&U.val[k*U.ld],n,&c.val[k],om,v.val);
    magma_dcopy(n,v.val,&U.val[k*U.ld]);
    magma_d_spmv(1.,A,v.val,0.,&G.val[k*G.ld]); // G(:,k) = A * U(:,k)
    // bi-orthogonalize the new basis vectors
    for (i = 0; i < k; ++i) {
      // α = P(:,i)’ G(:,k) / M(i,i)
      halpha.val[i] = magma_ddotc(n,&P.val[i*P.ld],&G.val[k*G.ld]);
      halpha.val[i] = halpha.val[i] / Mdiag.val[i];
      // G(:,k) = G(:,k) - α * G(:,i)
      magma_daxpy(n,-halpha.val[i],&G.val[i*G.ld],1,&G.val[k*G.ld],1);
    }
    if (k > 0) {
      magma_dsetvector(k,halpha.val,1,alpha.val,1); // upload α
      // U(:,k+1) = U(:,k+1) - U(:,1:k) * α(1:k)
      magma_dgemv(’N’,n,k,-1.,U.val,U.ld,alpha.val,1,1.,&U.val[k*U.ld],1);
    }
    // M(k:s,k) = (G(:,k)’ P(:,k:s))’ = P(:,k:s)’ G(:,k)
    magma_dmdotc(n,sk,1.,&P.val[k*P.ld],&G.val[k*G.ld],&M.val[k*M.ld+k]);
    magma_dgetvector(1,&M.val[k*M.ld+k],1,&Mdiag.val[k],1); // download M(k,k)
    magma_dgetvector(1,&f.val[k],1,&fk,1);                  // download f(k)
    beta.val[k] = fk / Mdiag.val[k];                 // β = f(k) / M(k,k)
    magma_daxpy(n,beta.val[k],&U.val[k*U.ld],x.val); // x = x + β * U(:,k)
    magma_daxpy(n,-beta.val[k],&G.val[k*G.ld],r.val); // r = r - β * G(:,k)
    magma_dsmoo1(n,rs.val,r.val,tt.val);             // t = rs - r
    magma_dmdotc(n,2,tt.val,tt.val,z.val);           // z = {<t,t>, <t,rs>}
    magma_dgetvector(2,&z.val[2],&hz.val[2]);        // download z
    gamma = hz.val[1] / hz.val[0];                   // γ = <t,rs>/<t,t>
    magma_daxpy(n,-gamma,tt.val,rs.val);             // rs = rs - γ * (rs-r)
    magma_dsmoo2(n,-gamma,x.val,xs.val);             // xs = xs - γ * (xs-x)
    nrmr = magma_dnrm2(n,rs.val);                    // |r|
    if (nrmr <= tolb) return;                        // convergence check
    if ((k + 1) < s) {
      // f(k+1:s) = f(k+1:s) - beta * M(k+1:s,k)
      magma_daxpy(sk-1,-beta,&M.val[k*M.ld+(k+1)],&f.val[k+1]);
    }
  }
  magma_d_spmv(1.,A,r.val,0.,t.val);     // t = A * r
  magma_dmdotc(n,2,t.val,t.val,z.val);   // z = {<t,t>, <t,r>}
  magma_dgetvector(2,z.val,hz.val);      // download z
  om = hz.val[1] / hz.val[0];            // ω = tr / (|t| * |t|)
  magma_daxpy(n,om,r.val,x.val);         // x = x + ω * r
  magma_daxpy(n,-om,t.val,r.val);        // r = r - ω * t
  magma_dsmoo1(n,rs.val,r.val,tt.val);   // t = rs - r
  magma_dmdotc(n,2,tt.val,tt.val,z.val); // z = {<t,t>, <t,rs>}
  magma_dgetvector(2,z.val,hz.val);      // download z
  gamma = hz.val[1] / hz.val[0];         // γ = <t,rs>/<t,t>
  magma_daxpy(n,-gamma,tt.val,rs.val);   // rs = rs - γ * (rs - r)
  magma_dsmoo2(n,-gamma,x.val,xs.val);   // xs = xs - γ * (xs - x)
  nrmr = magma_dnrm2(n,rs.val);          // |r|
  if (nrmr <= tolb) return;              // convergence check
} while (numiter < maxiter);
```

Figure 2. GPU implementation of IDR(*s*) in pseudocode using the MAGMA library.

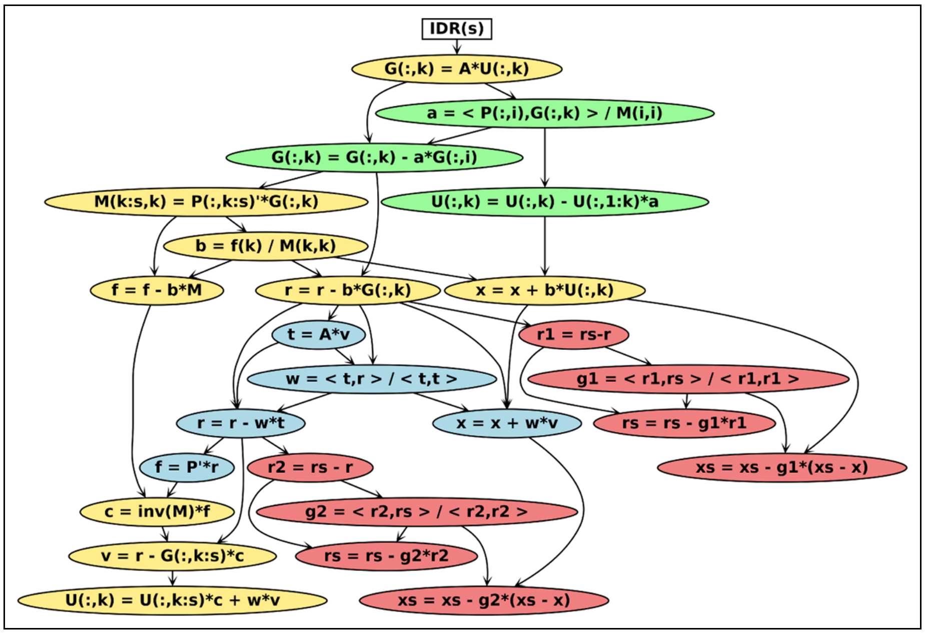


Figure 3. Dependency graph for a single iteration of IDR(s) biortho-variant enhanced with smoothing. The colors correspond to specific regions of the algorithm: blue → main loop, yellow → shadow space loop, green → biorthogonalization loop, and red → smoothing steps.

iteration. The graphs are colored to represent the different regions of the algorithm (e.g. loops and smoothing), this helps identify which steps are suitable for overlapping. In the experiments assessing the benefits of overlap, operations in the same row are scheduled for concurrent execution. This requires the rearrangement of some operations in the pseudocode provided in Figure 2. Precisely, the gemv (lines 3) and triangular solve (line 7) are taken out of the loop, and overlapped with the final smoothing operation of a previous iteration. The solution approximation of the next inner iteration (line 33) can be handled independently and therefore parallel to the bi-orthogonalization loop. For shadow space dimensions $s>1$, this applies also to the update of f (line 5 ff in Figure 2). A 3-way overlap occurs after the bi-orthogonalization loop between the gemv (line 29), scalar transfer (line 31), and the update of U (line 26) using a gemv kernel. Inside the smoothing operations, the residual and solution updates can be scheduled in parallel. Finally, some computations of the next iteration (lines 3, 7, 10, 12) can be overlapped with the current residual update (line 51).

5 Testbed

The GPU results for this paper are obtained from a Tesla K40 GPU which belongs to the Kepler line of NVIDIA's hardware accelerators and has a theoretical peak performance of 1682 Gflop/s (double precision). On the GPU, the 12 GB of device memory are sufficiently large to keep all the matrices and all the vectors needed in the iteration process. The theoretical memory bandwidth is listed as 288 GB/s. The practical bandwidth that can be achieved is determined in Section 6. The GPU hosts a shared L2 cache of 1.5 MB size. The IDR(s) implementation is based on NVIDIA's CUDA version 7.5 cud (2015). For the experiments, we use a set of test matrices taken from the University of Florida matrix collection. (UFMC)[1] Also, the efficiency of the sparse matrix vector product is guiding the performance of any Krylov subspace solver. Optimizing the sparse matrix vector product is outside the focus of this paper, but we always use the sliced Ellpack format with blocksize 32 (SELL-32 to follow the notation of Monakov et al. (2010)) for storing the matrices and handling the SpMV. Depending on the matrix characteristics, this can result in malicious vector access and significant storage overhead. For the latter, we report, in Table 1, the overhead of nonzeros explicitly stored in the used SELL-32 format. We use double precision arithmetic throughout all experiments, and 32-bit integers to store indexes.

6 Performance model

To assess the efficiency of the optimized IDR(s) implementation, we derive a roofline performance model Williams et al. (2009) that provides an upper performance bound. In general, the execution performance P of any algorithm is bound by

$$P = \min(P^{\text{peak}}; Ib)\ \text{Gflop/s} \tag{1}$$

with P^{peak} being the theoretical peak performance of the processing units, I the computational intensity of

Table 1. Key characteristics of the test matrices ordered with increasing size.

abbrev	matrix	size *n*	nonzeros *nz*	*nz*/*n*	*nz* stored in SELL-32	SELL-32 overhead [%]
SCR	scircuit	170,998	958,936	5.61	2,410,304	151.35
TDK	thermomech_dk	204,316	2,846,228	13.93	3,288,352	15.53
WEB	webbase-1M	1,000,005	3,105,536	3.11	9,828,736	216.49
NLP	nlpkkt80	1,062,400	28,704,672	27.02	29,074,432	1.29
DIE	dielFilterV2real	1,157,456	48,538,952	41.94	87,200,640	79.65
THM	thermal2	1,228,045	8,580,313	6.99	10,586,976	23.39
AFS	af_shell10	1,508,065	52,672,325	34.93	52,749,920	0.15
MLG	ML_Geer	1,504,002	110,879,972	73.72	111,261,248	0.35
G3	G3_circuit	1,585,478	7,660,826	4.83	7,794,048	1.74
TRA	Transport	1,602,111	23,500,731	14.67	23,582,656	0.35

Table 2. Maximum computational intensities in flops/byte of operations executed in our IDR(s) implementation.

`mdotc`, `gemv`	`copy`	`spmv`	`dotc`, `smoo2`	`axpy`	`smoo1`	`nrm2`
$\frac{1}{4(1+1/s)}$	0	$\frac{1}{6+8n/nz}$	$\frac{1}{8}$	$\frac{1}{12}$	$\frac{1}{24}$	$\frac{1}{4}$

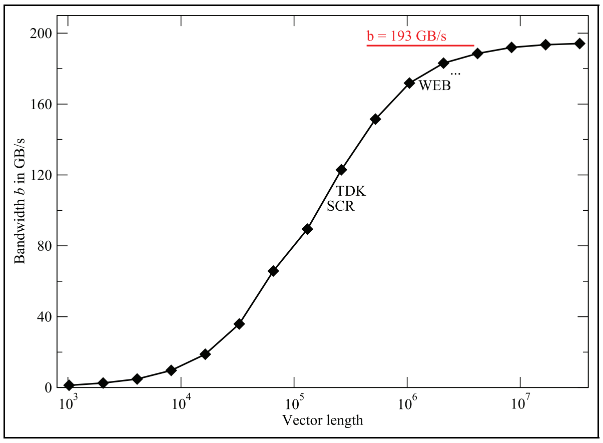


Figure 4. Attainable main memory bandwidth as obtained by a copy benchmark on the K40 GPU. Each thread copies one data element. A thread block contains 1024 threads and the number of thread blocks is doubled for each data point. Vector lengths for a selection of test problems (cf. Table 1) are marked.

the algorithm (that is, the number of executed floating point operations per transferred byte), and b the maximum attainable memory bandwidth.

The attainable bandwidth b has to be measured with a suitable micro-benchmark. For the K40 GPU, we use a copy benchmark from a GPU implementation of the STREAM McCalpin (1995) benchmark suite to quantify the attainable memory bandwidth b. The data reported in Figure 4 shows that we need large data sets and thread counts exceeding one million for saturating the main memory bandwidth. For some of the problems listed in Table 1, this requirement is not fulfilled, and the performance for those systems may be limited by a significantly lower bandwidth (see the respective matrix names indicated in Figure 4). To keep the performance model as generic as possible, we do not take the dependency between bandwidth and problem size into account, but base the roofline model on the maximum bandwidth of $b = 193$ GB/s.

Table 2 lists the computational intensities of the different operations involved in the IDR(s) algorithm. We assume perfect data reuse within operations and no data reuse across operations. The first assumption means that the input vector of `spmv` has to be read only once. This corresponds to the best case scenario for this operation and this assumption can be made in the light of giving an absolute upper performance limit.

The latter assumption is valid against the background that we consider data sets that are large compared to the L2 cache size of about 1.5 MB. To determine whether equation (1) is bound by P^{peak} or Ib, we consider the largest computational intensity of any involved operation. Naturally, the computational intensity of the entire algorithm can never be larger than this value.

In the search space of any values of n, nz, and s, the maximum computational intensity for a single operation is 1/4 flops/byte. This intensity is achieved for `nrm2` and, for very large values of s, also for `mdotc/gemv`. In this case, we get

$$\begin{aligned} P_{\texttt{nrm2,mdotc,gemv}} &= \min(P^{\text{peak}}; Ib) \\ &= \min(1682; 193/4) \text{ Gflop/s} \end{aligned}$$

Hence, even the computationally most intense operations are bound by the bandwidth. We deduce that the entire algorithm, as well as all individual kernels, are memory bound. Thus, we can limit our performance analysis to a pure bandwidth analysis.

Quantifying the performance of the algorithm in Gflop/s would be nonintuitive, and we reformulate the

Table 3. Minimum amount of vector and matrix transfers for each operation used in our IDR(s) implementation with line numbers as given in listing 2. Only operations which copy at least n data elements are considered. The last line summarizes the transfers for the full IDR(s) solver.

Line	Operation	Vector transfers	Matrix transfers
3	`mdotc`	$s+1$	
8	`copy`	$2s$	
10	`gemv`	$\sum_{i=2}^{s+1} i$	
12	`gemv`	$\sum_{i=2}^{s+1} i$	
13	`copy`	$2s$	
14	`spmv`	$2s$	s
18	`dotc`	$\sum_{i=1}^{s-1} 2i$	
21	`axpy`	$\sum_{i=1}^{s-1} 3i$	
26	`gemv`	$\sum_{i=2}^{s} i$	
29	`mdotc`	$\sum_{i=2}^{s+1} i$	
33	`axpy`	$3s$	
34	`axpy`	$3s$	
35	`smoo1`	$2s$	
36	`mdotc`	$3s$	
39	`axpy`	$3s$	
40	`smoo2`	$3s$	
41	`nrm2`	s	
48	`spmv`	2	1
49	`mdotc`	2	
54	`axpy`	3	
55	`axpy`	3	
56	`smoo1`	3	
57	`mdotc`	2	
60	`axpy`	3	
61	`smoo2`	3	
62	`nrm2`	1	
	`mdotc`	$s^2/2+9s/2+5$	
	`copy`	$4s$	
	`gemv`	$3s^2/2+7s/2-1$	
	`spmv`	$2s+2$	$s+1$
	`dotc`	s^2-s	
	`axpy`	$3s^2/2+15s/2+9$	
	`smoo1`	$3s+3$	
	`smoo2`	$3s+3$	
	`nrm2`	$s+1$	
	`IDR(s)`	$9s^2/2+55s/2+22$	$s+1$

roofline model for predicting execution times. For V being the minimum data volume which has to be transferred to or from main memory, the following formula defines a minimum runtime estimate for bandwidth-bound algorithms

$$t^{\min} = \frac{V}{b}\ \text{sec}$$

Table 3 lists all relevant operations in the IDR(s) implementation for different shadow space dimensions s. In the end, one outer iteration of IDR(s) requires $22 + 9s^2/2 + 55s/2$ vector transfers and $s+1$ matrix reads. Given the testbed setting described in Section 5, a vector transfer contains $8n$ bytes and a matrix transfer contains at least $12nz$ bytes (8 bytes for the value, 4 bytes for the column index). Therefore, the roofline performance model predicts the minimum execution runtimes for a single outer IDR(s) iteration as

$$t^{\min}_{\mathrm{IDR}(s)} = \frac{8n(9s^2/2 + 55s/2 + 22) + 12nz(s+1)}{b}\ \text{s} \quad (2)$$

7 Performance evaluation

For the matrices listed in Section 5, we measure the execution time for 100 outer iterations, and compare with the minimum runtime estimate as given in equation (2). The ratio between actual runtime and predicted runtime quantifies the efficiency of the optimized IDR(s) implementation.

In the left-hand panel of Figure 5, we visualize the efficiency for different shadow space dimensions s. As previously elaborated, the runtime of the `SpMV` kernel can significantly differ from the projected performance due to matrix storage overhead, malicious access patterns and undersized data sets. In order to differentiate between those influences, we present adjusted efficiency numbers under the assumption that all stored matrix entries contain useful information in the right-hand panel of Figure 5. This compensates for the storage overhead (which, in some sense, corresponds to a SELL-C-σKreutzer et al. (2014) matrix with optimally chosen σ). However, it ignores the effects of malicious vector access and small data sets not saturating the memory bandwidth.

The right-hand side of Figure 5 shows that we achieve very good efficiency for problems that are suitable for use of the SELL-32 format and large enough to saturate the memory bandwidth. We have around 70% efficiency for the THM problem and between 75% and 90% efficiency for the problems G3, AFS, NLP, TRA and MLG. For those systems, the efficiency is mostly consistent across the different shadow space dimensions.

Efficiency is much lower for WEB and DIE. For both problems, the efficiency grows with larger shadow space dimensions s. As larger shadow space dimensions decrease the impact of the `SpMV` on the overall runtime, this indicates that the performance is lost in the `SpMV`. This is consistent with the SELL-32 matrix storage format incurring significant overhead for these systems (see Table 1). Looking at the right-hand side of Figure 5 reveals that a performance model that accounts for the additionally stored zero elements reduces the performance gap for both systems. For the DIE problem, the memory-adjusted efficiency is about 80%, and is constant across the shadow space dimensions. The efficiency for the WEB problem stays below 50%, which still shows that the efficiency increases for larger shadow space dimensions. Detailed analysis reveals that the `SpMV` performance for the WEB problem suffers from the random vector access pattern which is not

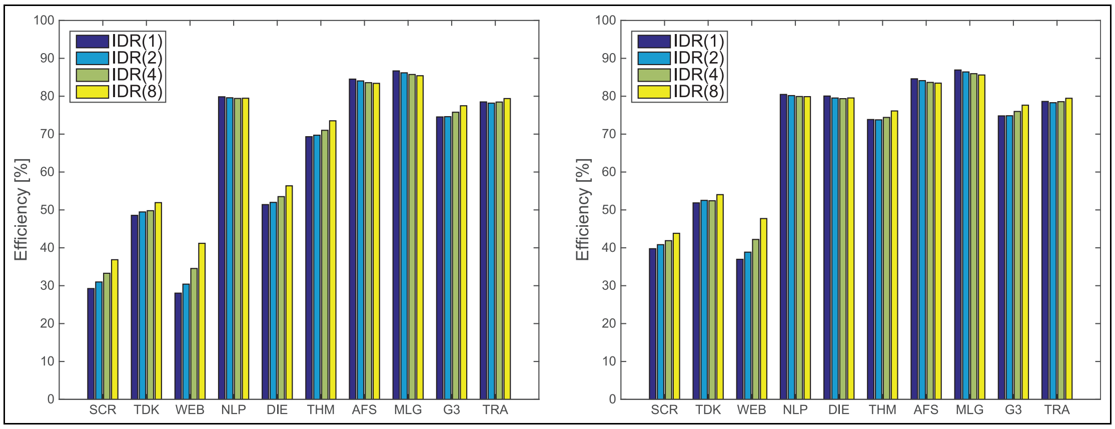


Figure 5. Efficiency of the optimized IDR(s) implementation based on MAGMA library function calls with respect to the roofline performance model. On the left-hand side the performance of the roofline model is based on an optimal sparse matrix vector product (`SpMV`) implementation. The roofline model on the right hand-side accounts for the matrix storage overhead to reflect the nonoptimality of the `SpMV`.

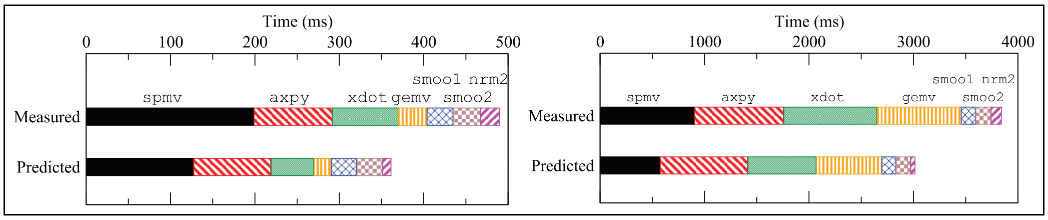


Figure 6. Runtime contributions for 100 outer iterations of IDR(1) (left panel) and IDR(8) (right panel) for the THM test case. The `mdotc` and `dotc` runtimes have been combined because the latter function gets called from within `mdotc` if appropriate and the profiler output does not reflect this fact.

accounted for in the memory-adjusted roofline model. Similar findings regarding this test case have also been made in Kreutzer et al. (2014).

For the small problems SCR and TDK, we achieve only 30%–45% and 50% efficiency, respectively. For SCR, the increase in efficiency with the shadow space dimension s is again an indicator for the nonoptimality of the `SpMV`. Also, when accounting for the memory overhead, we do not exceed 45% efficiency for the SCR case and 55% for the TDK case.

Given these efficiency numbers, we want to investigate whether we missed relevant optimization steps, or whether the remaining performance gaps originate from the specific problems and the nonoptimality of GPU kernel execution. For this purpose, we focus on the THM problem, and use a detailed performance analysis to identify missed optimization steps that help to reduce the performance gap to the roofline model.

In Figure 6, we compare the runtime for the distinct solver routines reported by NVIDIA's profiler with the execution time values projected by the roofline performance model. The upper bars are for a shadow space dimension $s = 1$ (IDR(1)). The entirely vector-parallel operations `axpy`, `smoo1` and `smoo2`, show only negligible differences from predictions. The roofline model does not account for the reduction phase included in `nrm2`, `xdot` and in `mdotc`. This explains why the runtimes for these routines are larger than predicted. Also, the `gemv` kernel shows performance lower than the model's predictions. The largest deviation, both relative and absolute, is, however, reported for the `SpMV` routine. Optimizing the sparse matrix product is outside the focus of this work, but its complexity is well known in the community (for GPU-related work on `SpMV` performance see, for example, Bell and Garland (2009); Monakov et al. (2010); A. Dziekonski and Mrozowski (2011); Vázquez et al. (2011); Kreutzer et al. (2014)).

The lower bar plot in Figure 6 is for IDR(8), using a shadow space dimension of $s = 8$. For larger shadow space dimensions, the `SpMV` becomes less important, and the performance impact of other operations is enhanced. As a result, the overall runtime gets closer to the roofline model prediction. Performance is still lost in the operations including a reduction phase and the `SpMV`. As previously elaborated, the roofline model is for those operations based on optimality premises that are unrealistic for an actual implementation.

Finally, we investigate whether concurrent kernel execution, as suggested in Anzt et al. (2015a), can improve the overall IDR(s) performance. Concurrent execution is only possible for data-independent kernels and communication instances. Also, benefits are only

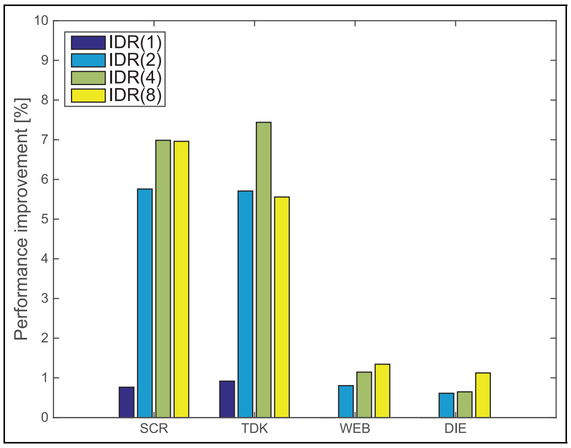


Figure 7. Performance improvement obtained by concurrent kernel execution for the test matrices SCR, TDK, WEB and DIE.

available in case operations listed side by side in Figure 3 and do not fully utilize the GPU resources. The analysis in Section 6 has revealed that the IDR(s) implementation is entirely bandwidth bound. Therefore performance improvements can only be expected if the parallelism, reflected in the number of active GPU threads, is too small to saturate the memory bandwidth. The bandwidth test in Section 6 indicates that this situation may occur for the small test matrices. At the same time, the cost of communicating scalar values between host and device becomes more relevant for decreasing problem size. Hence, also overlapping the communication with kernel execution may bring larger benefit for small problems. In Figure 7, we report the performance improvements obtained from concurrent kernel execution for the test cases with the lowest efficiency: SCR, TDK, WEB and DIE.

Noticeable benefits can only be observed for the small systems. For SCR and TDK, runtime can be reduced by 5%–7% when using shadow space dimensions 2, 4 or 8. The improvement benefits cases where $s>1$ for two main reasons. First, the transition between iterations in both inner and outer loops overlap; therefore concurrency gains will be more noticeable as the number of iterations increases. Second, operations in the inner loop use matrices and vectors with sizes based on parameter s, while operations in the outer loop are based on the actual problem size. These factors are more likely to allow inner loop operations to run in a concurrent fashion.

8 Summary

In this paper, we have proposed a GPU implementation of the IDR(s) algorithm based on algorithm-specific and data-optimized kernels. A roofline performance model was used to evaluate the efficiency for different test matrices. The analysis revealed that the IDR(s) performance is close to the maximum that can be expected for this algorithm. We also evaluated the benefits of overlapping computation with communication, and the possibility of concurrent kernel execution. As expected for memory-bound algorithms, the potential of these techniques is very limited, and only relevant when targeting small problems.

Acknowledgements

This material is based upon work supported in part by the US Department of Energy (grant number DE-SC-0010042), the German Research Foundation (DFG) through the Priority Program 1648 (SPPEXA) under project ESSEX, the Air Force Office of Scientific Research (grant number FA9550-12-1-0476), and NVIDIA.

Note

1. We selected a mix of symmetric and nonsymmetric matrices to cover a broad spectrum with respect to dimension and sparsity (see Table 1 for some key characteristics). The matrix characteristics can have significant impact on the IDR(s) performance. Larger problems provide more parallelism, which brings the achieved bandwidth closer to the maximum bandwidth the roofline performance model is based on (see Section 6).

References

(2015) *CUDA Toolkit v7.5*. NVIDIA Corporation.

(2015) *cuSPARSE Toolkit v7.0*. NVIDIA Corporation, v7.0 edition.

Aliaga J, Perez J, Quintana-Orti E, et al. (2013) Reformulated conjugate gradient for the energy-aware solution of linear systems on GPUs. In: *Proceedings of the 2013 42nd International Conference on Parallel Processing, IEEE Computer Society*, Washington, DC, USA 2013, pp.320–329.

Aliaga JI, Pérez J and Quintana-Ortí ES (2015) Systematic fusion of CUDA kernels for iterative sparse linear system solvers. In: *Euro-Par 2015: Parallel processing: 21st international conference on parallel and distributed computing*, Vienna, Austria, 24–28 August 2015, pp.675–686. Berlin, Heidelberg: Springer.

Anzt H, Ponce E, Peterson GD, et al. (2015a) GPU-accelerated co-design of induced dimension reduction: algorithmic fusion and kernel overlap. In: *Proceedings of the 2nd international workshop on hardware-software co-design for high performance computing*, Co-HPC 2015, pp.5:1–5:8. New York, NY: ACM. ISBN 978-1-4503-3992-6

Anzt H, Sawyer W, Tomov S, et al. (2015b) Acceleration of GPU-based Krylov solvers via data transfer reduction. *International Journal of High Performance Computing* 29: 366–383.

Bell N and Garland M (2009) Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: *Proceedings of the conference on high performance computing networking, storage and analysis*, SC 2009, pp.18:1–18:11. New York, NY: ACM.

Bergman K, Borkar S, Campbell D, et al. (2008) ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead. DARPA/IPTO Program.

Blackford LS, Demmel J, Dongarra J, et al. (2002) An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software* 28(2): 135–151.

Collignon TP and van Gijzen MB (2011) Minimizing synchronization in IDR (s). *Numerical Linear Algebra with Applications* 18(5): 805–825.

Dorostkar A, Lukarski D, Lund B, et al. (2014) CPU and GPU performance of large scale numerical simulations in geophysics. In: *Euro-Par 2014: parallel processing workshops, Lecture Notes in Computer Science*, volume 8805. Switzerland: Springer International Publishing, pp.12–23.

Dziekonski AL and Mrozowski M (2011) A memory efficient and fast sparse matrix vector product on a GPU. *Progress In Electromagnetics Research* 116: 49–63.

Filipovic J, Madzin M, Fousek J, et al. (2013) Optimizing CUDA code by kernel fusion—application on BLAS. *CoRR* abs/1305.1183.

Gregg C, Dorn J, Hazelwood K, et al. (2012) Fine-grained resource sharing for concurrent GPGPU kernels. In: *4th USENIX workshop on hot topics in parallelism*, Berkeley, CA: USENIX.

Hestenes MR and Stiefel E (1952) Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards* 49: 409–436.

Jiao Q, Lu M, Huynh HP, et al. (2015) Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS. In: *Proceedings of the 13th annual IEEE/ACM international symposium on code generation and optimization*, CGO 2015, pp.1–11. Washington, DC: IEEE Computer Society.

Knibbe H, Oosterlee C and Vuik C (2011) GPU implementation of a Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method. *Journal of Computational and Applied Mathematics* 236(3): 281–293.

Kreutzer M, Hager G, Wellein G, et al. (2014) A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36(5): C401–C423.

Kreutzer M, Thies J, Röhrig-Zöllner M, et al. (2015) GHOST: building blocks for high performance sparse linear algebra on heterogeneous systems. *CoRR* abs/1507.08101.

Li R and Saad Y (2013) GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing* 63(2): 443–466.

Lukash M, Rupp K and Selberherr S (2012) Sparse approximate inverse preconditioners for iterative solvers on GPUs. In: *HPC 2012: Proceedings of the 2012 symposium on high performance computing*, pp.1–8. San Diego, CA: Society for Computer Simulation International.

MAGMA (2015) MAGMA 1.6.2. Available at: http://icl.cs.utk.edu/magma/ (accessed November 2015).

Lukarski D and Trost N (2015) PARALUTION. Available at: http://www.paralution.com/PARALUTION (accessed November 2015).

McCalpin JD (1995) Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* 1995, 19–25.

Monakov A, Lokhmotov A and Avetisyan A (2010) Automatically tuning sparse matrix-vector multiplication for GPU architectures. In: *Proceedings of the 5th international conference on high performance embedded architectures and compilers*, HiPEAC 2010, pp.111–125. Berlin, Heidelberg: Springer-Verlag.

Rendel O, Rizvanolli A and Zemke JPM (2013) IDR: a new generation of Krylov subspace methods? *Linear Algebra and its Applications* 439(4): 1040–1061.

Rupp K (2015) ViennaCL. Available at: http://viennacl.sourceforge.net/ (accessed November 2015).

Saad Y (2003) *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Simoncini V and Szyld DB (2010) Interpreting IDR as a Petrov-Galerkin method. *SIAM Journal on Scientific Computing* 32(4), 1898–1912.

Sonneveld P and van Gijzen MB (2009) IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM Journal on Scientific Computing* 31(2): 1035–1062.

Strohmaier E, Dongarra J, Simon H and Meuer M (2015) The TOP500 list. Available at: http://www.top.org/ (accessed November 2015).

Tabik S, López GO and Garzón EM (2014) Performance evaluation of kernel fusion BLAS routines on the GPU: iterative solvers as case study. *The Journal of Supercomputing* 70(2): 577–587.

van Gijzen MB (2015) The induced dimension reduction method. Available at: http://ta.twi.tudelft.nl/nw/users/gijzen/IDR.html (accessed November 2015).

van Gijzen MB, Sleijpen GLG and Zemke JPM (2015) Flexible and multi-shift induced dimension reduction algorithms for solving large sparse linear systems. *Numerical Linear Algebra with Applications* 22(1): 1–25.

van Gijzen MB and Sonneveld P (2011) Algorithm 913: an elegant IDR(s) variant that efficiently exploits biorthogonality properties. *ACM Transactions on Mathematical Software* 38(1): 5:1–5:19.

Vázquez F, Fernández JJ and Garzón EM (2011) A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 23(8): 815–826.

Wang G, Lin Y and Yi W (2010) Kernel fusion: an effective method for better power efficiency on multithreaded GPU. In: *Proceedings of the 2010 IEEE/ACM international conference on green computing and communications & international conference on cyber, physical and social computing*, GREENCOM-CPSCOM 2010, pp.344–350. Washington, DC: IEEE Computer Society.

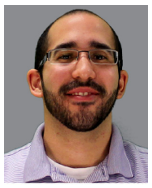
Wang L, Huang M and El-Ghazawi T (2011) Exploiting concurrent kernel execution on graphic processing units. In: *High performance computing and simulation (HPCS), 2011 international conference on*, Washington, DC: IEEE, pp.24–32.

Williams S, Waterman A and Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. *Communications of the Association for Computing Machinery* 52(4): 65–76.

Author biographies

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

Moritz Kreutzer completed his BSc in computational engineering in 2009 at the Friedrich-Alexander University of Erlangen-Nuremberg, Germany. Afterwards, he pursued his MSc studies in computational engineering (scientific computing) at the same university and the Royal Institute of Technology in Stockholm, Sweden. After finishing his MSc studies in 2011, he started working as a PhD student and research assistant under the supervision of Prof. Gerhard Wellein at the Erlangen Regional Computing Center (RRZE).



Eduardo Ponce is a PhD graduate research and teaching assistant in electrical engineering and computer science at The University of Tennessee in Knoxville. He holds a BS in computer engineering and MS in electrical engineering from Polytechnic University of Puerto Rico. His research interests include computational science applications, optimizations of numerical algorithms for computing accelerators, and performance modeling for distributed systems.

Gregory D. Peterson is Professor in the Department of Electrical Engineering and Computer Science and Director of the National Institute of Computational Sciences at The University of Tennessee, Knoxville, Tennessee, He received the BS, MS and DSc degrees in electrical engineering and the BS and MS degrees in computer science from Washington University, St Louis. His research interests include computational science, parallel processing, electronic design automation, performance evaluation, and high performance reconfigurable computing.

Gerhard Wellein holds a PhD in solid state physics from the University of Bayreuth and is a regular Professor at the Department for Computer Science at University of Erlangen-Nuremberg. He heads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than ten years of experience in teaching HPC techniques to students and scientists from computational science and engineering. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.

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 in 2004; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; in 2011 he was the recipient of the IEEE IPDPS Charles Babbage Award; and in 2013 he received the ACM/IEEE Ken Kennedy Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.