GPU-accelerated Co-design of Induced Dimension Reduction: Algorithmic Fusion and Kernel Overlap

Hartwig Anzt, Eduardo Ponce, Gregory D. Peterson, Jack Dongarra
University of Tennessee
Knoxville, Tennessee, USA
hanzt@icl.utk.edu, eponcemo@utk.edu, gdp@utk.edu, dongarra@icl.utk.edu

ABSTRACT

In this paper we present an optimized GPU co-design of the Induced Dimension Reduction (IDR) algorithm for solving linear systems. Starting from a baseline implementation based on the generic BLAS routines from the MAGMA software library, we apply optimizations that are based on kernel fusion and kernel overlap. Runtime experiments are used to investigate the benefit of the distinct optimization techniques for different variants of the IDR algorithm. A comparison to the reference implementation reveals that the interplay between them can succeed in cutting the overall runtime by up to about one third.

CCS Concepts

•**Mathematics of computing** → **Solvers;**

Keywords

Induced Dimension Reduction (IDR), GPU, co-design, kernel fusion, kernel overlap.

1. INTRODUCTION

Krylov subspace iterative methods [20] are among the most popular methods for solving large sparse linear systems of the form $Ax = b$ which arise in many scientific and engineering fields. Against the background of an increasing number of computer systems featuring hardware accelerators like GPUs [11, 2], significant research focuses on 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. A straightforward way to use accelerators in Krylov subspace solvers is to offload all matrix and vector computations to the device using library functions. In many cases, this results in significant acceleration of the algorithm [8]. However, even larger improvements are often available when replacing library-based functions with application-specific kernels that keep data in local memory as much as possible. This concept of "kernel fusion" in particular pays off as algorithms working on sparse matrices are typically memory-bound, and merging multiple linear algebra routines into a single kernel reduces the pressure on the memory bandwidth [6]. Aside from that, the acceleration of the sparse matrix-vector product needed to generate the Krylov subspace [20], which is often the computationally most expensive part of the algorithms, is often the subject of tuning. Finally, some algorithms allow for overlapping computation and communication by executing multiple kernels concurrently. This, however, not only requires the algorithm to contain computations that are not directly dependent and can be executed in a flexible order, but furthermore will only result in noticeable benefits if not all computations are memory-bound.

Although Krylov subspace solvers are usually memory-bound when applied to sparse problems, they may contain parts that benefit from using concurrent kernel execution. Identifying these parts can be challenging, especially as overlapping kernels can conflict with kernel fusion, e.g., in case two kernels use the same data to update two distinct vectors. In this paper we address the interplay of these two optimization techniques when porting the Induced Dimension Reduction (IDR [22]) algorithm to NVIDIA GPUs. We structure the rest of the paper as follows. First, in Sect. 2 we provide an overview of related work on strategies for accelerating Krylov subspace methods on GPUs. We then briefly review the IDR algorithm and its variants in Sect. 3. Section 4 and Sect. 5 contain the main contributions of this work:

- We discuss the GPU implementation of IDR and the possible optimization steps such as the fusion of multiple linear algebra operations into a single kernel, and overlapping different computations by running multiple GPU kernels in concurrent fashion.
- We investigate the trade-off between kernel fusion and kernel overlap for a memory-bound matrix addition for different levels of data reuse.
- We apply optimization steps expected to provide benefit to a reference implementation based on BLAS function calls.
- We analyze the performance benefit of the distinct optimization steps for the IDR implementation.

We conclude with a summary of the results and some ideas for future work in Sect. 6.

2. RELATED WORK

Several open-source software packages provide GPU-support for Krylov subspace solvers [15, 3, 17, 14]. These often provide significantly higher performance than native CPU implementations [8, 12, 13]. For the IDR algorithm, the benefits of GPU-acceleration were investigated in [10]. There is also literature available on the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Co-HPC2015 November 15-20, 2015, Austin, TX, USA

© 2015 ACM. ISBN 978-1-4503-3992-6/15/11. . . $15.00
DOI: http://dx.doi.org/10.1145/2834899.2834907

concept of merging multiple linear algebra operations into one single algorithm-specific kernel in order to reduce pressure on the device memory. In [9] the authors have shown that kernel fusion can be realized for BLAS1 and dense BLAS2 operations by using a source-to-source compiler. No automatic fusion of sparse linear algebra operations is addressed, however. In [23] the authors combine CUDA kernels in iterative sparse linear system solvers, but only consider kernels that provide the same functionality and have no dependencies among them. Explicit kernel coding was used in [4], 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 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 [25] 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 [5], where a precise characterization of the kernels and the possibility of merging them into one single kernel is presented.

3. INDUCED DIMENSION REDUCTION

The Induced Dimension Reduction (IDR) algorithm is a robust framework for deriving iterative solvers for large nonsymmetric linear systems of equations. It is based on the Krylov subspace idea, and was first introduced by P. Sonneveld and M. B. Gijzen in [22]. Numerous variants exist to enhance the solver's convergence, stability, or parallelism level [21, 24, 19]. However, they all share the idea of exploiting the following central theorem [22]:

THEOREM 1. *(IDR) Let A be any matrix in $C^{N\times N}$, let v_0 be any nonzero vector in C^N, and let G_0 be the complete Krylov space $K^N(A,v_0)$. Let P denote any (proper) subspace of C^N such that P and G_0 do not share a nontrivial invariant subspace of A, and define the sequence $G_j, j=1,2,...$ as*

$$G_j = (I-\omega_j A)(G_{j-1}\cap P)$$

where the ω_j are nonzero scalars. Then

1. *$G_j \subset G_{j-1}$ for all $j>0$.*
2. *$G_j=\{0\}$ for some $j\leq N$.*

One popular variant of Theorem 1 is IDR(s) which can be constructed by considering s independent, standard normally distributed, shadow vectors $p_1,p_2,...,p_s$ to solve a smaller system of equations based on the iterative residuals [24]. The smaller system represents a set of polynomials that force the generated residuals to be in subspaces G_j, thus enforcing the convergence of x_i after, at most N, dimension-reduction steps.

An improved variant of the IDR algorithm is the biortho-variant including smoothing [24]. The approach uses the iteration residuals with the assumption that each residual is the first vector in the next reduced subspace G_{j+1}. Faster convergence is achieved by taking advantage of the biorthogonality property

$$g_{n+k}\perp p_i, i=1,...,k-1, k=2,...,s$$

$$r_{n+k+1}\perp p_i, i=1,...,k, k=1,...,s$$

where p_i are orthonormal shadow vectors. A comprehensive collection of research efforts and a more detailed derivation of the algorithm can be found at [1].

4. OPTIMIZING IDR ON GPUS

A GPU implementation for main loop of the IDR(s)-biortho algorithm enhanced with smoothing is provided in pseudocode in Fig. 1. For convenience, we extracted the smoothing option into Fig. 2. This implementation uses, exclusively, BLAS1 and BLAS2 functions as part of the MAGMA software library to express the algorithm. This BLAS-based implementation will also serve as a baseline to quantify the architecture-specific tuning in Sect. 5. Our co-design supports heterogeneous execution via a single interface of the MAGMA library which allows mapping the IDR algorithm to both CPU and GPU systems effectively.

The optimization steps we apply to this baseline implementation are the fusion of BLAS functions into algorithm-specific kernels, and the concurrent execution of GPU kernels. In general, these optimization steps are not independent, as overlapped kernels cannot be fused, and fusing two kernels into one removes the possibility of overlapping. Depending on the amount of data reused, the one or other may provide larger benefits. For a memory-bound algorithm, we use a small experiment to analyze this issue. More precisely, we compute two matrix sums, where the second sum partly reuses addends from the first sum, see Figure 3. A parameter κ is used to control how much of the data gets reused: For $\kappa=0$ the two sums are identical, for increasing $\kappa=0$ the amount of data reused decreases. Using two distinct kernels allows for concurrent execution using different streams. Also, in Figure 3, both additions are merged into one single kernel, such that data gets reused.

Figure 3 reports runtime and achieved memory bandwidth of both options for increasing κ. Especially for the smaller problem size, we see some benefit of launching the two kernels in concurrent fashion (left-hand side of Figure 3). The fused kernel requires reading from distant memory locations, resulting in a lower memory bandwidth. As long as the addends do not overlap, executing two independent kernels (in concurrent fashion) gives better performance. If data is shared by the two sums, the fused kernel catches up. For 10% overlap (corresponding to reducing the memory transfers by 6%), the runtime of the two independent and overlapped kernels is matched. Completely overlapping the addends reduces the memory transfers by one third, resulting in 33% runtime improvement for this memory-bound operation. We conclude that for memory-bound algorithms, larger performance benefits can be expected from reusing data already read into local memory (kernel fusion) than from overlapping data reload with a previous computation (concurrent execution). Kernels sharing only scalar values, or being completely data-independent, benefit more from overlapping than from fusing, as the overhead of reading a scalar value multiple times is small compared to the benefits of more consecutive memory reads. We apply the two optimization techniques with respect to these premises.

For the kernel fusion, we use the classification proposed in [5] to sort the operations into groups depending on whether the input and output vectors can be mapped to the GPU thread block alignment. We then ignore all kernels that have no data dependency nor 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 that MAGMA supports multiple storage formats for sparse matrices, and we want to keep the flexibility of switching between the respective sparse matrix vector kernels. Although replacing the sequence of axpys to update U in the inner biorthogonalization (line 27 in Fig. 1) with a matrix vector product outside this loop does not require the implementation of an algorithm-specific kernel, it may be considered as kernel fusion at it combines a set of BLAS1 functions into one BLAS2 function.

```
do {
  numiter++;
  // f = P' r
  magma_dgemv(CT,P.nrows,P.ncols,1.,P.val,P.nrows,r.val,1,0.,f.val,1);
  // shadow space loop
  for (k = 0; k < s; ++k) {  sk = s - k;
    magma_dcopy(sk,&f.val[k],1,c.val[k],1);          // f(k:s) = c(k:s)
    magma_dcopy(M.nrows*M.ncols,M.val,1,M1.val,1);  // M = M1
    // f(k:s) = M(k:s,k:s) c(k:s)
    magma_dgesv_gpu(sk,c.ncols,M1.val[k*M1.ld+k],M1.nrows,piv,c.val[k],c.nrows,&info);
    magma_dcopy(dof,r.val,1,v.val,1);                // r = v
    // v = r - G(:,k:s) c(k:s)
    magma_dgemv(N,G.nrows,sk,-1.,G.val[k*G.ld],G.nrows,c.val[k],1,1.,v.val,1);
    // U(:,k) = om * v + U(:,k:s) c(k:s)
    magma_dgemv(N,U.nrows,sk,1.,U.val[k*U.ld],U.nrows,c.val[k],1,om,v.val,1);
    magma_dcopy(U.nrows,v.val,1,U.val[k*U.ld],1);   // v = U(:,k)
    magma_d_spmv(1.,A,v,0.,v,queue);                 // G(:,k) = A v
    magma_dcopy(G.nrows,z.val,1,G.val[k*G.ld],1);
    for (i = 0; i < k; ++i) {
      // alpha = <P(:,i),G(:,k)> / M(i,i)
      alpha = magma_ddotc(P.nrows,P.val[i*P.ld],1,G.val[k*G.ld],1);
      magma_dgetvector(1,M.val[i*M.ld+i],1,&mkk,1);
      alpha = alpha / mkk;
      // G(:,k) = G(:,k) - alpha * G(:,i)
      magma_daxpy(G.nrows,-alpha,G.val[i*G.ld],1,G.val[k*G.ld],1);
      // U(:,k) = U(:,k) - alpha * U(:,i)
      magma_daxpy(U.nrows,-alpha,U.val[i*U.ld],1,U.val[k*U.ld],1);
    }
    // M(k:s,k) = P(:,k:s)' G(:,k)
    magma_dgemv(CT,P.nrows,sk,1.,P.val[k*P.ld],P.nrows,
                          G.val[k*G.ld],1,0.,M.val[k*M.ld+k],1);
    magma_dgetvector(1,&f.val[k],1,&fk,1);
    beta = fk / mkk;                                 // beta = f(k) / M(k,k)
    magma_daxpy(dof,beta,U.val[k*U.ld],1,x->val,1); // x = x + beta * U(:,k)
    magma_daxpy(dof,-beta,G.val[k*G.ld],1,r.val,1); // r = r - beta * G(:,k)
    magma_dsmoothing(...);                           // smoothing operation
    if (nrmr <= tolb) return;                        // check convergence
    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)],1,&f.val[k+1],1);
    }
  numiter++;
  }
  magma_dcopy(dof,r.val,1,z.val,1);
  magma_d_spmv(1.,A,v,0.,t,queue);                // t = A v
  nrmt = magma_dznrm2(dof,t.val,1);               // ||t||
  tr = magma_ddotc(dof,t.val,1,r.val,1);          // tr = <t,r>
  rho = fabs(MAGMA_Z_REAL(tr) / (nrmt * nrmr); // rho = tr / (||t|| * ||r||)|
  om = tr / (nrmt * nrmt);                        // om = tr / (||t|| * ||t||)
  magma_daxpy(dof,om,z.val,1,x->val,1);           // x = x + om * v
  magma_daxpy(dof,-om,t.val,1,r.val,1);           // r = r - om * t
  magma_dsmoothing(...);                          // smoothing operation
  if (nrmr <= tolb) return;                       // check convergence
} while (numiter < maxiter);
```

Figure 1: GPU implementation of IDR in pseudocode using the MAGMA library.

```
if (smoothing == 1) {
  magma_dcopy(dof,rs.dval,1,t.dval,1);            // rs = t
  magma_daxpy(dof,-1.,r.dval,1,t.dval,1);         // t = rs - r
  nrmt = magma_dznrm2(dof,t.dval,1);              // ||t||
  gamma = magma_ddotc(dof,t.dval,1,rs.dval,1); // gamma = <t,rs>
  gamma = gamma / (nrmt * nrmt);                  // gamma = gamma / (||t|| * ||t||)
  magma_daxpy(dof,-gamma,t.dval,1,rs.dval,1);     // rs = rs - gamma * t
  magma_dcopy(dof,xs.dval,1,t.dval,1);            // xs = t
  magma_daxpy(dof,-1.,x->dval,1,t.dval,1);        // t = xs - x
  magma_daxpy(dof,-gamma,t.dval,1,xs.dval,1);     // xs = xs - gamma * t
  nrmr = magma_dznrm2(dof,rs.dval,1);             // nrmr = ||rs||
} else {
  nrmr = magma_dznrm2(dof,r.dval,1); }            // nrmr = ||r||
```

Figure 2: Pseudocode of the smoothing option in IDR implemented using the MAGMA software library.

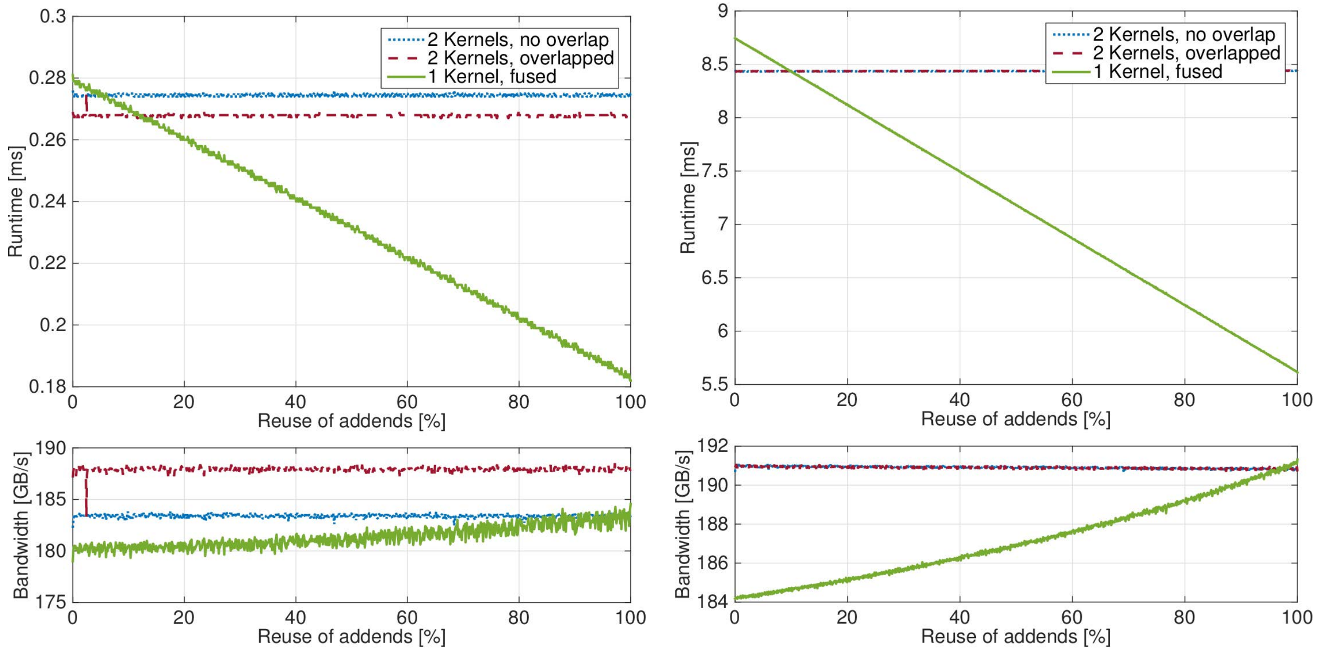

Figure 3: Comparing runtime (top) and achieved memory bandwidth (bottom) for running two independent (overlapped) kernels or the merged option for computing the two matrix sums. The left-hand side results are for a small problem ($32,768 \times 32$; $1,024$ GPU thread blocks of size 32×32), the right-hand side results for a large problem (matrix size $1,048,576 \times 32$; $32,768$ GPU thread blocks of size 32×32).

Similarly, the solution approximation update in line 34 (Fig. 1) can be handled by a matrix vector product outside the respective loop if no smoothing is used. Smoothing is essential for a monotone residual decay as can be seen by comparing left- and right-hand side in Fig. 5 in the experimental evaluation in Sect. 5. Therefore, we keep smoothing as an option in our algorithm. Custom-designed kernels are required in the smoothing option (Fig. 2). The computation of $t = rs - r$ (line 2,3) as well as the computation of $xs = xs - \gamma(xs - x)$ (line 8-10) can be realized with higher efficiency in one kernel, respectively. The baseline implementation also contains, in several places, a set of two consecutive dot products sharing one vector (line 46,47 in Fig. 1, and line 4,5 in Fig. 2).

The recently proposed merged dot product [6] allows for fusing these. For this case, the performance benefits not only come from reusing one vector, but also from combining the computationally expensive parallel reduction phases into one. We must emphasize that we do not apply all fusions that are possible, only those that turned out to be beneficial for performance, and do not conflict with kernel overlaps providing larger improvement. Similarly, we do not push concurrency of kernels to the limit. Running concurrent kernels requires switching between the GPU stream and careful synchronization. For runtime improvement, this overhead has to be compensated for by the kernel overlap.

Figure 4 shows a dependency graph for one iteration of the IDR(s) algorithm with smoothing enabled. 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. Additional overlapping can be attained if the graph is constructed for $s > 1$ and multiple iterations. We now outline several key concepts of CUDA-enabled GPUs and the MAGMA library that enable effective 2-way and 3-way concurrency with few explicit synchronization points.

- The default stream is synchronous with respect to CPU and GPU. Any execution in this stream synchronizes all other streams, unless a stream is created with the *cudaStreamNonBlocking* flag.
- Asynchronous transfers require CPU page-locked memory.
- Concurrency requires operations to be handled by different streams — given that sufficient resources are available.
- Operations get added to streams in issue order; this enforces a synchronization signal between streams but not within them.
- MAGMA reduction operations that return a scalar value (e.g., dot and norm) are synchronous with respect to the CPU.

Based on these concepts, our optimized version makes use of three streams: one using *cudaStreamNonBlocking* flag, and two non-default streams. All scalar values are allocated in page-locked memory, and the leading dimension of the matrices are aligned to 32 bytes. Reduction operations are used as implicit synchronization points, kernel-kernel operations are issued in breadth first order, and transfer-kernel-transfer operations are issued in depth first order. Another essential modification to permit maximum concurrency is to rearrange the structure of the loops by unrolling the loop until a dependency that requires a synchronization is hit. Both loop levels in the IDR(s) algorithm were unrolled using this strategy.

In the optimized version, we overlap the following routines and refer to the code shown in Fig. 1 in case smoothing is enabled. The first gemv (lines 4) and general solver (line 10) are taken out of the loop and overlapped with the final smoothing operation. This general solver was transformed into a triangular solver since the matrix M is lower triangular, and its transfer (line 8) is unrolled and overlapped with the first smoothing operation. The solution approximation of the next inner iteration (line 34) is handled concurrently with the biorthogonalization loop, and for cases with $s > 1$, the update of f as well. A 3-way overlap occurs after the biorthogonalization loop between the gemv (line 30), scalar transfer (line 32), and the

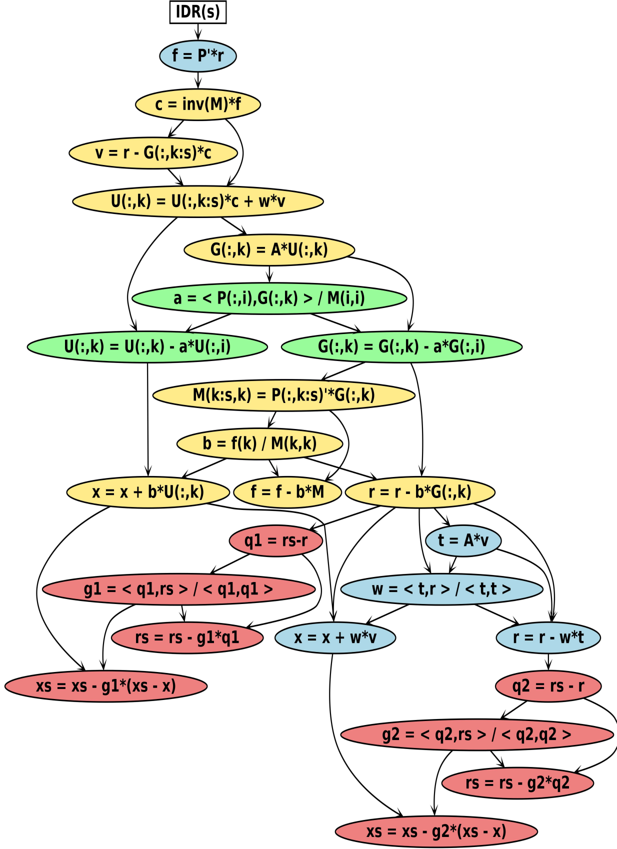

Figure 4: Dependency graph for a single iteration of IDR(s) biortho-variant enhanced with smoothing. The colors correspond to specific regions of the algorithm: blue → main loop, yellow → shadow space loop, green → biorthogonalization loop, and red → smoothing steps.

update of U (line 27) using a gemv kernel. It is worth mentioning that residual and solution updates in the smoothing operation are overlapped as well as memory transfers of internal work arrays. Finally, the next iteration computations of lines 4,7,10 are executed in concurrent fashion with the current residual updates and convergence check (lines 51,53). After these modifications, the optimized IDR(s) variant requires explicit synchronization calls for the non-blocked stream at the end of both the shadow space and main loops. The remaining streams make use of implicit synchronization points provided by scalar reduction computations.

5. EXPERIMENTAL RESULTS

5.1 Experiment setup

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 1,682 GFlop/s (double precision). On the 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. The baseline GPU results were obtained from using the iterative solver package of the MAGMA open source software library [14] linked to NVIDIA's cuBLAS [18] and cuSPARSE library [17] in version 7.0. For the optimized version, algorithm-specific kernels were implemented in the CUDA language, version

matrix	abbrev	#nonzeros (nnz)	Size (n)	nnz/n
AIRFOIL_2D	AIR	259,688	14,214	18.27
PRES_POISSON	PRE	715,804	14,822	48.29
TREFETHEN_20000	TRE	554,466	20,000	27.72
BMWCRA1	BMW	10,641,602	148,770	71.53
INLINE_1	INL	38,816,170	503,712	77.06
APACHE_2	APA	4,817,870	715,176	6.74
LDOOR	LDO	42,493,817	952,203	44.63
BONE010	BONE	47,851,783	986,703	48.50
ECOLOGY_2	ECO	4,995,991	999,999	5.0
G3_CIRCUIT	G3	7,660,826	1,585,478	4.83

Table 1: Key characteristics of the test matrices ordered according to their dimension.

7.0 [16]. For the experiments, we use a set of test matrices taken from the University of Florida matrix collection (UFMC, [7]). The IDR algorithm does not require the system matrix to be symmetric. 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 specific sparsity pattern and symmetry characteristics are not relevant for this performance evaluation as the implementation of the sparse matrix vector product remains outside the scope of the optimization process.

5.2 Convergence and Performance

To ensure the algorithm's correctness, in Fig. 5 we compare the convergence of the basic GPU implementation with the convergence of a MATLAB reference implementation taken from [1]. We analyze shadow space dimensions 1, 2, 4, and 8. The left-hand side of Fig. 5 shows the jagged convergence pattern for both implementations, which is characteristic for the basic IDR(s) algorithm. While already small rounding differences in the computation of dot products result in significant differences concerning the residual for a specific iteration count, the implementations have very similar average convergence rates. This becomes more obvious when enabling the smoothing option, see the right-hand side of Fig. 5.

To quantify the overhead of using larger shadow space dimensions, in Table 2 we list the actual runtimes (and normalized to the shadow space dimension 1) needed by the baseline GPU implementation to execute 100 iterations. The solution of the small dense systems required for the minimization process results in a superlinear runtime increase for larger shadow space dimensions. We notice that the relative runtime increase is smaller when smoothing is enabled.

Next, we apply the optimization steps outlined in Sect. 4. Fig. 6 reports the runtime improvement obtained from of the distinct optimization levels of the GPU implementation over the CPU code. Again, we address both, the basic IDR(s) (left-hand side plots) and the IDR(s) using smoothing (right-hand side plots). The graphs in the first line illustrate the runtime improvement obtained from kernel fusion: we replaced the generic BLAS functions with the algorithm-specific kernels introduced in Sect. 4. Kernel fusion provides larger improvements for the smoothed IDR(s), as algorithm-specific kernels significantly reduce the memory transfers in the smoothing operation. Furthermore, we observe that the algorithm-specific kernels are in particular beneficial when working with small shadow space dimensions s. The bottom of Fig. 6 combines the kernel fusion with kernel overlap. As expected, concurrent kernel execution is in particular attractive when overlapping the loops for the distinct shadow spaces.

Furthermore, concurrent kernel execution provides larger benefits to the unsmoothed IDR(s). This is expected as combining the

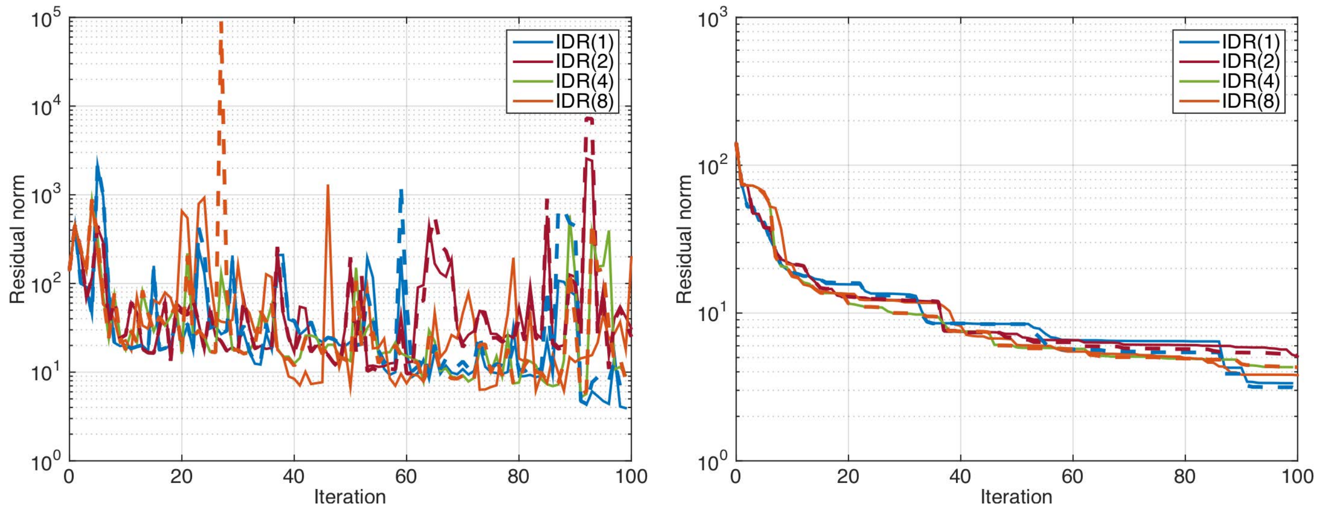

Figure 5: Convergence of the MATLAB IDR(s) code (solid lines) and the GPU implementation available in the MAGMA software library (dashed lines) for the TRE test matrix. The left-hand side is the basic algorithm, the right-hand side uses smoothing.

Figure 6: Runtime improvement obtained from kernel fusion (top) and the combination with kernel overlap (bottom) for the basic (left) and smoothed (right) IDR(s).

	No smoothing								Smoothing							
Matrix	IDR(1)		IDR(2)		IDR(4)		IDR(8)		IDR(1)		IDR(2)		IDR(4)		IDR(8)	
AIR	0.0340	(1.)	0.0361	(1.06)	0.0414	(1.22)	0.0521	(1.53)	0.0501	(1.)	0.0511	(1.02)	0.0573	(1.14)	0.0684	(1.37)
PRE	0.0409	(1.)	0.0427	(1.04)	0.0475	(1.16)	0.0585	(1.43)	0.0587	(1.)	0.0590	(1.01)	0.0630	(1.07)	0.0742	(1.26)
TRE	0.0409	(1.)	0.0414	(1.01)	0.0485	(1.19)	0.0576	(1.41)	0.0544	(1.)	0.0571	(1.05)	0.0622	(1.14)	0.0733	(1.35)
BMW	0.1714	(1.)	0.1764	(1.03)	0.1875	(1.09)	0.2115	(1.23)	0.2010	(1.)	0.2063	(1.03)	0.2171	(1.08)	0.2410	(1.20)
INL	0.5444	(1.)	0.5573	(1.02)	0.5817	(1.07)	0.6364	(1.17)	0.6045	(1.)	0.6179	(1.02)	0.6412	(1.06)	0.6951	(1.15)
APA	0.3192	(1.)	0.3367	(1.05)	0.3715	(1.16)	0.4465	(1.40)	0.3952	(1.)	0.4132	(1.05)	0.4473	(1.13)	0.5223	(1.32)
LDO	0.9050	(1.)	0.9293	(1.03)	0.9738	(1.08)	1.0740	(1.19)	1.0003	(1.)	1.0254	(1.03)	1.0687	(1.07)	1.1690	(1.17)
BONE	0.9628	(1.)	0.9870	(1.03)	1.0333	(1.07)	1.1373	(1.18)	1.0610	(1.)	1.0852	(1.02)	1.1316	(1.07)	1.2368	(1.17)
ECO	0.4214	(1.)	0.4438	(1.05)	0.4898	(1.16)	0.5954	(1.41)	0.5186	(1.)	0.5433	(1.05)	0.5895	(1.14)	0.6960	(1.34)
G3	0.6540	(1.)	0.6918	(1.06)	0.7631	(1.17)	0.9242	(1.41)	0.8011	(1.)	0.8393	(1.05)	0.9095	(1.14)	1.0701	(1.34)

Table 2: Absolute runtimes in seconds (and normalized to the runtime for shadow space dimension 1) for 100 iterations of the basic GPU implementation of IDR(s).

x-updates (line 34 in Figure 1) into a matrix vector product outside the shadow space loop allows to completely overlap this operation. Looking at the interplay of the optimization steps, larger runtime reduction (up to 33%) can be achieved for smoothing enabled. As expected, the benefits are larger when targeting smaller test matrices, as for those the cost of the sparse matrix vector product is not necessarily dominating the algorithm.

6. SUMMARY

In this paper, we have shown how a GPU implementation of the IDR(s) algorithm and an enhanced variant featuring a smoothing step for better convergence properties can be accelerated by applying kernel fusion and the concept of overlapping kernels. A runtime analysis revealed that custom-designed kernels are particularly attractive for small shadow space dimensions, while kernel overlap provides significant benefits when running the smoothed variant in combination with large shadow space dimensions. Combining the optimization steps we succeed in cutting the overall runtime by up to about one third.

In future, we will look into operations that are not memory bound for a more comprehensive study on whether kernel fusion or kernel overlap should be preferred. The overall goal is a theoretical model that allows to identify the optimal choice depending on the algorithm and hardware characteristics.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy (Award Number DE-SC-0010042), and NVIDIA.

7. REFERENCES

[1] The induced dimension reduction method; http://ta.twi.tudelft.nl/nw/users/gijzen/IDR.html.

[2] The top 500 list, http://www.top.org/.

[3] ViennaCL. http://viennacl.sourceforge.net/, 2015.

[4] J. Aliaga, J. Perez, E. Quintana-Orti, and H. Anzt. Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 320–329, Oct 2013.

[5] J. I. Aliaga, J. Pérez, and E. S. Quintana-Ortí. Systematic fusion of cuda kernels for iterative sparse linear system solvers. In *Lecture Notes in Computer Science, Euro-Par 2015*, accepted.

[6] H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, and J. Dongarra. Acceleration of GPU-based Krylov Solvers via Data Transfer Reduction. *International Journal of High Performance Computing*, 2015.

[7] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1 – 25, 2011.

[8] A. Dorostkar, D. Lukarski, B. Lund, M. Neytcheva, Y. Notay, and P. Schmidt. CPU and GPU performance of large scale numerical simulations in geophysics. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 12–23. Springer International Publishing, 2014.

[9] J. Filipovic, M. Madzin, J. Fousek, and L. Matyska. Optimizing CUDA code by kernel fusion—application on BLAS. *CoRR*, abs/1305.1183, 2013.

[10] H. Knibbe, C. Oosterlee, and C. Vuik. GPU implementation of a helmholtz krylov solver preconditioned by a shifted laplace multigrid method. *Journal of Computational and Applied Mathematics*, 236(3):281 – 293, 2011. Aspects of Numerical Algorithms, Parallelization and Applications.

[11] P. Kogge *et al*. ExaScale computing study: Technology challenges in achieving ExaScale systems, 2008.

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

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

[14] MAGMA 1.6.2. http://icl.cs.utk.edu/magma/, 2015.

[15] PARALUTION. http://www.paralution.com/, 2015.

[16] NVIDIA Corporation. *CUDA Toolkit v7.0*, March 2015.

[17] NVIDIA Corporation. *cuSPARSE Toolkit v7.0*, v7.0 edition, March 2015.

[18] NVIDIA Corporation v7.0. *CUDA cuBLAS Toolkit*, March 2015.

[19] O. Rendel, A. Rizvanolli, and J.-P. M. Zemke. IDR: A new generation of Krylov subspace methods? *Linear Algebra and its Applications*, 439(4):1040 – 1061, 2013. 17th Conference of the International Linear Algebra Society, Braunschweig, Germany, August 2011.

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

[21] V. Simoncini and D. B. Szyld. Interpreting IDR as a Petrov-Galerkin method. *SIAM Journal on Scientific*

Computing, pages 1898–1912, 2010.

[22] P. Sonneveld and M. B. van Gijzen. 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, 2009.

[23] S. Tabik, G. O. López, and E. M. Garzón. Performance evaluation of kernel fusion BLAS routines on the GPU: Iterative solvers as case study. *The Journal of Supercomputing*, 70(2):577–587, 2014.

[24] M. B. Van Gijzen and P. Sonneveld. Algorithm 913: An elegant IDR(s) variant that efficiently exploits biorthogonality properties. *ACM Trans. Math. Softw.*, 38(1):5:1–5:19, Dec. 2011.

[25] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 344–350, Washington, DC, USA, 2010. IEEE Computer Society.