PIPELINED ITERATIVE SOLVERS WITH KERNEL FUSION FOR GRAPHICS PROCESSING UNITS

K. RUPP*†, J. WEINBUB*, A. JÜNGEL†, AND T. GRASSER*

Abstract. We revisit the implementation of iterative solvers on discrete graphics processing units and demonstrate the benefit of implementations using extensive kernel fusion for pipelined formulations over conventional implementations of classical formulations. The proposed implementations with both CUDA and OpenCL are freely available in ViennaCL and achieve up to three-fold performance gains when compared to other solver packages for graphics processing units. Highest performance gains are obtained for small to medium-sized systems, while our implementations remain competitive with vendor-tuned implementations for very large systems. Our results are especially beneficial for transient problems, where many small to medium-sized systems instead of a single big system need to be solved. [1]

Key words. Iterative Solvers, Conjugate Gradient Method, BiCGStab Method, GMRES Method, GPU, OpenCL, CUDA

AMS subject classifications. 65F10, 65F50, 65Y05, 65Y10

1. Introduction.

The need for the solution of a linear system of equations described by a sparse matrix A and a right hand side vector b is ubiquitous in computational science and engineering. Most prominently, discretizations of linear partial differential equations by means of the finite element or the finite volume method directly lead to such systems. Smaller-sized systems may be solved using sparse direct solvers, whereas iterative solvers are preferred or even necessary for large systems, eventually supplemented by preconditioning techniques of various degrees of sophistication.

The fine-grained parallelism of iterative solvers from the family of Krylov methods is particularly attractive for massively parallel hardware such as graphics processing units (GPUs), whereas much more effort is required to expose the parallelism in sparse direct solvers appropriately [19, 41]. Sparse matrix-vector products - essential parts of Krylov methods - have been studied in detail for GPUs [5, 7] and for INTEL's many-integrated core (MIC) architecture [24, 39], based on which a unified format also well-suited for multi-core processors has been proposed recently [21]. Similarly, vendor-tuned implementations of the vector operations required in addition to the sparse matrix-vector products for implementing sparse iterative solvers from the family of Krylov methods are available. A disadvantage of current accelerators is their connection to the host system via the PCI-Express bus, which is often a bottleneck both in terms of latency as well as bandwidth. This mandates a certain minimum system size to amortize the overhead of data transfer through the PCI-Express bus in order to obtain any performance gains over an execution on the host.

Two programming models are currently in widespread use for general purpose computations on GPUs: CUDA is a proprietary programming model for NVIDIA GPUs [31] providing its own compiler wrapper, whereas OpenCL is a royalty-free

*Institute for Microelectronics, TU Wien, Gußhausstraße 27-29/E360, A-1040 Wien, Austria

†Institute for Analysis and Scientific Computing, TU Wien, Wiedner Hauptstraße 8-10/E101, A-1040 Wien, Austria

[1]Editorial Policy Note: A preliminary investigation of a pipelined conjugate gradient formulation for graphics processing units was presented at the GAMM meeting in Erlangen, Germany, earlier this year and is about to appear as a two-page proceedings entry; this paper builds on top of these initial ideas and presents them in full depth.

open standard maintained by the Khronos Group [32] and is typically provided as a shared library. Although OpenCL can also be used for NVIDIA GPUs, the richer CUDA toolchain has resulted in a higher share of research on general purpose computations on GPUs using CUDA. Also, slight performance differences of CUDA and OpenCL, caused by different degrees of compiler optimizations or differences in the implementation rather than through differences in the programming model, have been reported [11, 18]. Automated translators such as Swan [15] or CU2CL [27] have been developed to reduce the maintenance effort of CUDA and OpenCL branches. However, only a subset of CUDA and OpenCL is supported by these translators, limiting their applicability particularly for highly optimized implementations. Consequently, portable software libraries targeting GPUs are currently driven into providing support for both CUDA and OpenCL, for example Paralution [33], VexCL [44], or ViennaCL [45].

A substantial amount of research has been conducted on various preconditioning techniques for iterative solvers on GPUs including algebraic multigrid [6, 13, 34, 46], incomplete factorizations [23, 30], or sparse approximate inverses [10, 25, 40]. Nevertheless, hardware-efficient and scalable black-box preconditioners for GPUs are not available, but instead the use of problem-specific information is required [50]. Taking preconditioner setup costs into account, iterative solvers using simple diagonal preconditioners or no preconditioner at all are often observed to be competitive in terms of time-to-solution for small to mid-sized systems, where e.g. the asymptotic optimality of multigrid preconditioners is not yet dominant [46]. Similarly, matrix-free methods cannot be used with complicated black-box preconditioners in general.

In this work we consider three popular iterative solvers: The conjugate gradient (CG) method for symmetric positive definite systems [16], the stabilized bi-conjugate gradient (BiCGStab) method for non-symmetric positive definite systems [43], and the generalized minimum residual (GMRES) method for general systems [38]. In contrast to previous work with a focus on the optimization of sparse matrix-vector products [5, 7, 21, 24, 39], we consider the optimization potential of the full solvers rather than restricting the optimization to a single kernel. After a careful evaluation of the limiting resources for different system sizes and different densities of nonzeros in the system matrix, pipelining and kernel fusion techniques are presented in Section 2 to resolve these bottlenecks to the extent possible. The key principle in pipelined techniques is to apply not only a single operation to a data word loaded from main memory, but to chain multiple operations together to reduce the overall number of loads and stores to global memory. Pipelining is typically achieved by fusing multiple compute kernels, but compute kernels may also be fused only to reduce the overall number of kernel launches, not exhibiting any pipelining effect. Pipelining and kernel fusion are then applied to the CG method, the BiCGStab method, and the GMRES method in Section 3, leading to more efficient solver implementations than those using a sequence of calls to the basic linear algebra subprograms (BLAS) in vendor-tuned libraries. Section 4 then compares the proposed solver implementations with existing solver implementations for GPUs available in the software libraries CUSP [9], MAGMA [26], and Paralution [33], demonstrating a substantial performance gain for small systems without sacrificing performance for large systems. Our benchmark results clearly show the benefit of kernel fusion and pipelining techniques and that these techniques have not been rigorously applied to the implementation of the CG method, the BiCGStab method, and the GMRES method in the context of GPU computing before.

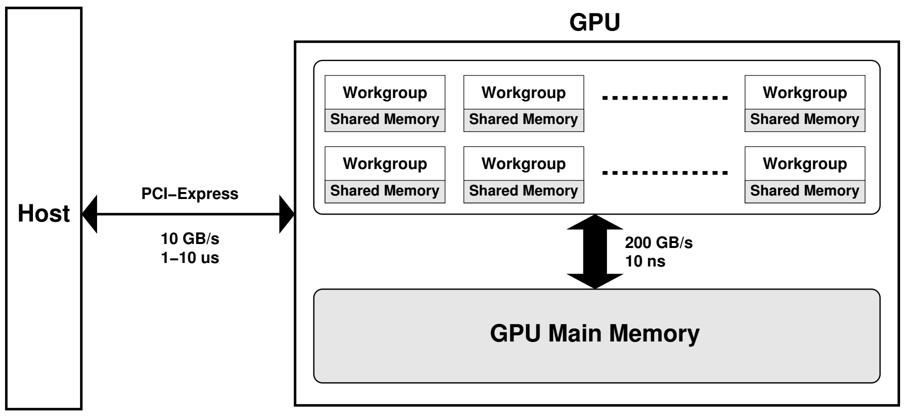


Fig. 1. *Schematic view of a GPU board connected to the host via PCI-Express at a bandwidth of about 10 GB/sec and a latency on the order of 10 microseconds. Each workgroup of threads can be synchronized through shared memory, but global synchronization is available only through separate kernel launches.*

The obtained execution times are also compared with those obtained from CPU-based implementations in the PETSc library [2, 3] to demonstrate that CPU-based implementations are superior for typical sparse systems below about 5 000 unknowns. Our results, similar to previous investigations [22], also falsify wide-spread misconceptions of extreme performance gains using GPUs. We show that performance gains of GPUs over power-equivalent dual-socket CPU machines are below an order of magnitude on average. This holds true also for large problem sizes and when initial data setup costs on GPUs are not taken into account. Finally, Section 5 discusses the implications of our findings to software design and the need for more tightly integrated future hardware generations.

2. Implementation Techniques for Fast Iterative Solvers. The purpose of this section is to identify the general bottlenecks of the typical building blocks of iterative solvers and to present techniques for mitigating their detrimental effects on performance. A schematic view of a machine (*host*) equipped with a discrete GPU connected via PCI-Express is given in Fig. 1, where the following key features are schematically depicted using a terminology similar to OpenCL:

- Threads are collected in *workgroups*, where each workgroup provides dedicated memory shared across threads in the workgroup. Thread synchronizations within a workgroup are possible inside a compute kernel, but a global synchronization of all workgroups is typically only possible by launching a new kernel. Although global synchronization primitives and spin locks through atomic operations are used occasionally, these techniques are not sufficiently portable across different hardware and thus not further considered.
- If a kernel launch is initiated on the host, it takes at least a few microseconds until the kernel will launch on a GPU. This is because a kernel launch on the GPU requires a message from the host to trigger the execution, entailing high latency for communication across PCI-Express. This latency of kernel launches can be hidden if another kernel is currently active on the GPU, in which case the PCI-Express message for launching the new kernel is received asynchronously.
- Memory access latency of GPU main memory is around three orders of mag-

nitude smaller than the latency of messages across the PCI-Express bus.
- The memory bandwidth between GPU main memory and the GPU compute units can be more than ten times higher than the bandwidth of the PCI-Express bus connecting host and GPU. Current high-end GPUs offer over 200 GB/sec memory bandwidth, whereas the current PCI-Express 3.0 offers up to 15.75 GB/sec for a 16-lane slot.

The remainder of this section quantifies the overhead of PCI-Express latency and presents techniques for reducing the number of kernel launches to reduce the detrimental latency effect.

2.1. PCI-Express Latency. At the very least, iterative solvers executed on the GPU need to communicate information about the current residual norm to the host. In the typical case of a communication of the residual norm in each iteration for convergence checks, the time required for a data transfer from the device to the host represents a lower bound for the time required for an iterative solver iteration. An OpenCL benchmark for PCI-Express data communication shown for an NVIDIA Tesla C2050 in Fig. 2 exhibits a latency-dominated regime for message sizes below ten kilobytes, where the transfer time is around eight microseconds. Latency-dominated data transfer from the device to the host takes almost twice as long, because a transfer initiation from the host is required first. Similar timings and bandwidths are obtained on other GPUs both with PCI-Express 2.0 and 3.0. Our overall observation in Section 4 is that NVIDIA GPUs show slightly lower latency than AMD GPUs on the Linux-based machines used for the comparison.

To better understand the latency induced by PCI-Express transfer, consider a high-end GPU with 200 GB/s memory bandwidth. Within the PCI-Express latency of 8 microseconds, the GPU can load or store 1.6 megabytes of data from main memory assuming full saturation of the memory channel, which amounts to 200 000 values in double precision and which we will refer to as *latency barrier*. Consequently, GPUs suffer from inherent performance constraints for any kernel limited by memory bandwidth whenever the total amount of data processed in a kernel is significantly below the latency barrier. On the other hand, many practical applications induce systems with storage requirements for the unknowns close to or even below the latency barrier. In such case, iterative solver implementations for GPUs need to keep the latency-induced overhead as small as possible by packing multiple operations into each kernel.

2.2. Kernel Fusion. As a prototypical example for many iterative solvers, consider the sequence of operations

$$q = Ap \tag{2.1}$$

$$\alpha = \langle p, q \rangle \tag{2.2}$$

for a scalar value α, vectors p and q, and a sparse square matrix A. Conventional implementations based on BLAS routines involve the following steps:

1. Call the sparse matrix-vector product kernel for computing (2.1). For a standard compressed sparse row (CSR) representation of the sparse matrix, a typical OpenCL kernel body is as follows (cf. [5, 7]):

```
for (uint i = get_global_id(0); i < size; i += get_global_size(0)) {
  double q_at_i = 0;
  for (uint j = A_row[i]; j < A_row[i+1]; ++j)
    q_at_i += A_values[j] * p[A_col[j]];
  q[i] = q_at_i;
}
```

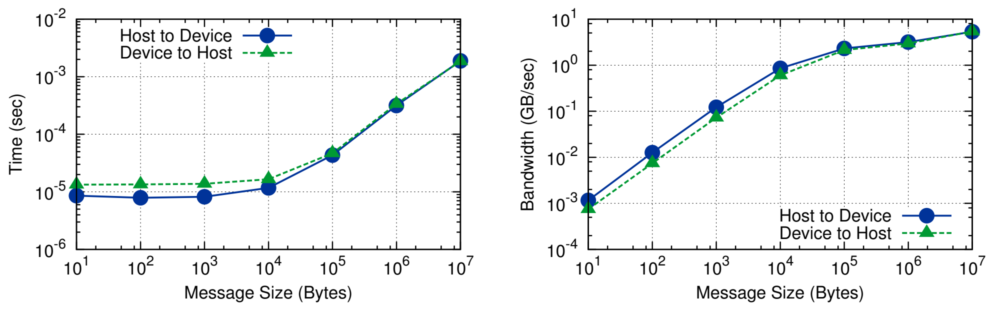


FIG. 2. *Plot of median values for execution time (left) and obtained bandwidth (right) from* 100 *host-device data transfers over PCI-Express* 2.0 *using an NVIDIA Tesla C2050. The benchmark uses the OpenCL functions clEnqueueWriteBuffer() and clEnqueueReadBuffer() in a blocking manner so that the respective function only returns after the data is sent or received. Message sizes below* 10^4 *bytes are limited by latency, not PCI-Express bandwidth.*

where A_row and A_col are the arrays holding the row and column indices in the CSR storage format, and A_values holds the nonzero entries.

2. Compute the partial results of $\langle p, q\rangle$ for the subvectors assigned to each of the thread workgroups.
3. If α is stored on the GPU, launch another kernel using a single thread workgroup to sum the partial results. If α is stored on the host, transfer the partial results to the host and perform the summation there.

Although this conventional implementation can reuse vendor-tuned routines, the multiple kernel launches are detrimental to performance for data sizes below the PCI-Express latency barrier.

On closer inspection, the operations (2.1) and (2.2) can be computed more efficiently by fusing compute kernels: Since the respective values in q and p are already available in the GPU processing elements when computing the matrix-vector product, they can be reused to compute the partial results for each thread workgroup of the inner product. The fused kernel body for the CSR format is as follows:

```
// Part 1: Matrix-vector product
double p_in_q = 0;
for (uint i = get_global_id(0); i < size; i += get_global_size(0)) {
  double q_at_i = 0;
  for (uint j = A_row[i]; j < A_row[i+1]; ++j)
    q_at_i += A_values[j] * p[A_col[j]];
  q[i]    = q_at_i;
  p_in_q += q_at_i * p[i];       // extra operation for <p, q>
}

// Part 2: Reduction to obtain contribution from thread workgroups:
__local double shared_buf[BUFFER_SIZE];
shared_buffer[get_local_id(0)] = p_in_q;
for (uint stride=get_local_size(0)/2; stride > 0; stride /= 2) {
  barrier(CLK_LOCAL_MEM_FENCE);
  if (get_local_id(0) < stride)
    shared_buf[get_local_id(0)] += shared_buf[get_local_id(0) + stride];
}

if (get_local_id(0) == 0)
  partial_result[get_group_id(0)] = shared_buf[0];
```

First, the matrix-vector kernel from the previous snippet is only slightly augmented to accumulate the partial results for each thread in p_in_q. Then, a reduction using a shared buffer (*local memory* in OpenCL terminology) shared_buf of appropriate size

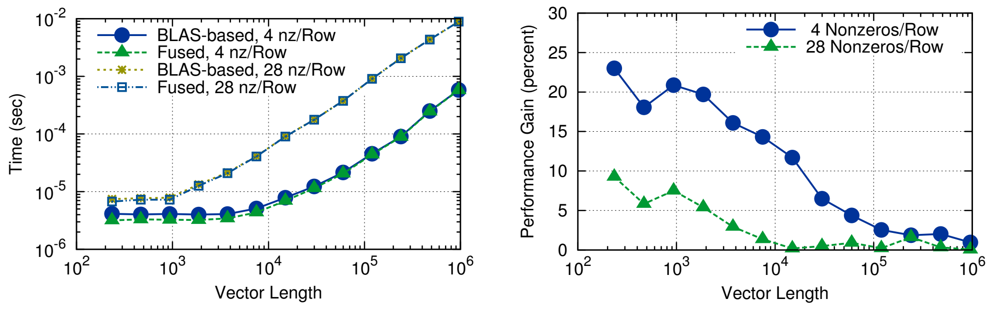


Fig. 3. *Total time required to run the operations* (2.1) *and* (2.2) *for different matrix and vector sizes on an NVIDIA Tesla C2050. If the vector size is below* 10 000 *entries, the total time is dominated by the latency for enqueuing the kernel, not the kernel execution time.*

BUFFER_SIZE is applied to obtain the sum over all threads within a thread workgroup. Finally, the first thread in each thread workgroup writes the partial result of the workgroup to a temporary buffer partial_result . The summation of the values in partial_result is carried out on the host as outlined in the third step above.

A comparison of execution times of the conventional implementation with the implementation using the fused kernel is given in Fig. 3. In both cases the final reduction step for the partial results from 128 thread workgroups has been computed on the host and is included in the timings. Two types of matrices have been compared: The first family of matrices with four randomly (with a uniform distribution over all column indices) distributed nonzeros per row is limited by latency for systems with up to 10^4 unknowns. A performance gain of about 20 percent is obtained from the use of a fused kernel, which reduces the number of kernels required from two to just one. At system sizes above 10^5 unknowns, a performance gain of a few percent is still obtained because the vector q does not need to be reloaded from memory again when computing the inner product (2.2). The second matrix type with 28 randomly distributed nonzeros per row is limited by the kernel execution time for system sizes below 10^3 unknowns. This is because each thread needs to process 28 nonzeros per row in A, which results in a larger execution time than the pure PCI-Express latency. Nevertheless, a performance gain of up to ten percent is obtained for smaller systems, yet there is no notable performance gain or loss at larger system sizes due to diminishing savings from reusing values q for computing the inner product.

It is not only possible to fuse the first stage in the inner product computation with the matrix-vector product, but one can also fuse the second stage (summation of partial results) with subsequent operations. Since the summation result is usually needed in each thread workgroup, the final summation has to be computed in each thread workgroup in such case. These redundant computations are usually well below the PCI-Express latency barrier and thus faster than the use of a separate host-device transfer or a dedicated summation kernel. While kernel fusion can in principle be applied to an arbitrary number of vector updates, the global synchronization points induced by matrix-vector products as well as inner products are natural boundaries for fusing compute kernels. However, not every inner product induces a separate synchronization point: The partial summation stage of several inner products may also be computed within the same kernel, which is then followed by a second kernel computing the final results of the inner products and possibly other vector operations.

3. Pipelined Iterative Methods for Graphics Processing Units. The implementation of the CG method, the BiCGStab method, and the GMRES method are investigated in depth in the following. Each of these solvers is analyzed for the number of kernel launches to evaluate latency. The kernel fusion techniques outlined in Section 2 are applied to reduce the number of kernel launches whenever appropriate. We restrict our investigations to the execution on a single GPU, as this is the most frequent use case, and leave optimizations for multi-GPU implementations for future work. Nevertheless, certain optimizations applied in this section can also be transferred to a multi-GPU setting, even though additional logic is required to exchange data between GPUs via the PCI-Express bus.

3.1. Conjugate Gradient Method. Several variations of the classical CG method [16] have been proposed in the past, cf. [4, 8, 14]. Also, techniques for merging multiple solver iterations have been proposed, but they do not find broad acceptance in practice because of numerical instabilities [36]. In the following, the classical CG method and a pipelined version are compared, where the latter has already been developed for vector machines [8], revisited for extreme-scale scalability [14], and implemented in field-programmable gate arrays [42]:

Algorithm 1: Classical CG

```
1  Choose $x_0$
2  $p_0 = r_0 = b - Ax_0$
3
4
5
6  for $i = 0$ to convergence do
7      Compute and store $Ap_i$
8      Compute $\langle p_i, Ap_i \rangle$
9      $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
10     $x_{i+1} = x_i + \alpha_i p_i$
11     $r_{i+1} = r_i - \alpha_i Ap_i$
12
13
14
15     Compute $\langle r_{i+1}, r_{i+1} \rangle$
16
17     $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
18     $p_{i+1} = r_{i+1} + \beta_i p_i$
19 end
```

Algorithm 2: Pipelined CG

```
1  Choose $x_0$
2  $p_0 = r_0 = b - Ax_0$
3  Compute and store $Ap_0$
4  $\alpha_0 = \langle r_0, r_0 \rangle / \langle p_0, Ap_0 \rangle$
5  $\beta_0 = \alpha_0^2 \langle Ap_0, Ap_0 \rangle / \langle r_0, r_0 \rangle - 1$
6  for $i = 1$ to convergence do
7
8
9
10     $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
11     $r_i = r_{i-1} - \alpha_{i-1} Ap_{i-1}$
12     $p_i = r_i + \beta_{i-1} p_{i-1}$
13     Compute and store $Ap_i$
14     Compute $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$
15     Compute $\langle r_i, r_i \rangle$
16     $\alpha_i = (r_i, r_i) / (p_i, Ap_i)$
17     $\beta_i = \alpha_i^2 \langle Ap_i, Ap_i \rangle / \langle r_i, r_i \rangle - 1$
18
19 end
```

A direct implementation of Algorithm 1 using one call to a matrix-vector product routine and five calls to BLAS routines per solver iteration is straightforward. Optimizations of the matrix-vector products on lines 2 and 7 in Algorithm 1 and lines 2, 3 and 13 in Algorithm 2 have been investigated in detail for different matrix formats on GPUs in the past [5, 7]. The inner products in lines 8 and 15 in the classical CG formulation impose synchronization by either splitting the operation into two kernels or by requiring a host-device transfer. In particular, the residual norm computed in line 15 is typically required on the host for convergence checks. The vectors r and p are loaded in lines 10 and 11, but have to be reloaded for the search vector update operation in line 18.

The pipelined version in Algorithm 2 is based on the relation

$$\langle r_{i+1}, r_{i+1} \rangle = \frac{\alpha_i^2 \langle Ap_i, Ap_i \rangle - \langle r_i, r_i \rangle}{\langle r_i, r_i \rangle} = \alpha_i^2 \langle Ap_i, Ap_i \rangle / \langle r_i, r_i \rangle - 1 \tag{3.1}$$

to compute p_i in line 18 of Algorithm 1 without having computed $\langle r_{i+1}, r_{i+1} \rangle$ yet. We note that it has been stated in the literature that precomputing inner products involving the vectors p_i and r_i by using recursion formulas based only on inner products of p_j and r_j with $j < i$ may lead to unstable algorithms [8, 36]. However, the computation of β_i involves Ap_i, resulting in a stable algorithm based on experiences from multiple groups in different application contexts [4, 8, 42].

Relation (3.1) allows for a rearrangement of Algorithm 1 such that all vector updates can be computed right after each other (lines 10, 11, and 12 in Algorithm 2). An application of kernel fusion not only allows for computing all three vector updates within a single kernel instead of three, but also for avoiding a reload of p_{i-1} and r_i (line 12) when using registers for intermediate values. Furthermore, all three inner products in Algorithm 2 can be computed simultaneously, allowing all intermediate results to be communicated to the host with a single data transfer. More precisely, the first reduction stage for the inner products $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$, and $\langle r_i, r_i \rangle$ can be computed within the same kernel sharing the same buffer for intermediate results. Then, the second reduction stage for obtaining the final results is either computed by only a single additional kernel launch, or by communicating all partial results with a single transfer to the host, which performs the summation. The data size of a single partial result is about one kilobyte, hence the data transfer time remains in the latency-dominated regime even if three of them are packed together, cf. Fig. 2.

To further enhance data reuse, we fuse the matrix-vector product in line 13 with the inner products in line 14, so that the result values of the matrix vector product can be processed right before they are written to GPU RAM. Thus, the computation of $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$ comes at reduced data transfer cost, because the j-entry of Ap_i has just been computed, and the j-th entry of p_i may still be available in cache. Similarly, the inner product $\langle r_i, r_i \rangle$ in line 15 is fused with the vector update kernel for lines 10, 11, and 12.

In summary, we propose the following implementation of Algorithm 2:

- Compute lines 10, 11, and 12 in one kernel and store the reduction results of each workgroup for the computation of $\langle r_i, r_i \rangle$ in line 15 in a temporary buffer.
- Compute lines 13 and 14 in one kernel and append the reduction results of each workgroup for the computation of the inner products in line 14 to the same temporary buffer.
- Communicate the temporary buffer to the host, where the final reduction is computed to obtain $\langle r_i, r_i \rangle$, α_i, and β_i from lines 15, 16 and 17.

Since $\langle r_i, r_i \rangle$ is available for monitoring the residual norm on the host, a convergence check can be applied in each iteration with no extra effort. The proposed implementation requires only two kernel launches per iteration and one host-device data transfer. In contrast, a direct translation of the classical CG algorithm into BLAS-routines requires at least six kernel launches (lines 7, 8, 10, 11, 15, and 18) and may involve a second host-device data transfer for $\langle p_i, Ap_i \rangle$. Consequently, we expect an up to three-fold performance gain for small systems in the latency-dominated regime. Because p_i and r_i do not need to be loaded from memory twice per iteration, a performance gain of a few percent may also be obtained for large systems with very few nonzeros per row.

3.2. BiCGStab. BiCGStab is an attractive solver for systems described by non-symmetric matrices, because the transposed operator A^{T} is not required. Based on the initial derivation [43], a pipelined method with only two global synchronizations has been proposed [17]. Later, a variant with only a single global synchronization has been proposed at the cost of an application of the transposed operator in the setup stage [49]. Also, a preconditioned BiCGStab method overlapping global communication with the application of the preconditioner has been developed [20]. A preliminary optimization study of the classical BiCGStab for GPUs is also available [1], for which we postpone a comparison to Section 4.

Similar to the classical BiCGStab algorithm, the pipelined BiCGStab implementation considered in this work does not require the transposed operator A^{T} to be available and is similar to the one proposed with two global synchronizations [17]. A comparison with the classical BiCGStab algorithm [37] is as follows:

Algorithm 3: Classical BiCGStab

```
1  Choose $x_0$
2  $p_0 = r_0 = b - Ax_0$
3  Choose $r_0^*$ arbitrary
4  Compute $\langle r_0, r_0^* \rangle$
5  for $i = 0$ to convergence do
6      Compute and store $Ap_i$
7      Compute $\langle Ap_i, r_0^* \rangle$
8      $\alpha_i = \langle r_i, r_0^* \rangle / \langle Ap_i, r_0^* \rangle$
9      $s_i = r_i - \alpha_i A p_i$
10     Compute and store $As_i$
11     Compute $\langle As_i, s_i \rangle$, $\langle As_i, As_i \rangle$
12
13
14     $\omega_i = \langle As_i, s_i \rangle / \langle As_i, As_i \rangle$
15     $x_{i+1} = x_i + \alpha_i p_i + \omega_i s_i$
16     $r_{i+1} = s_i - \omega_i A s_i$
17     Compute $\langle r_{i+1}, r_0^* \rangle$
18     $\beta_i = \frac{\langle r_{i+1}, r_0^* \rangle}{\langle r_i, r_0^* \rangle} \times \frac{\alpha_i}{\omega_i}$
19     $p_{i+1} = r_{i+1} + \beta_i (p_i - \omega_i A_i)$
20
21 end
```

Algorithm 4: Pipelined BiCGStab

```
1  Choose $x_0$
2  $p_0 = r_0 = b - Ax_0$
3  Choose $r_0^*$ arbitrary
4  Compute $\langle r_0, r_0^* \rangle$
5  for $i = 0$ to convergence do
6      Compute and store $Ap_i$
7      Compute $\langle Ap_i, r_0^* \rangle$
8      $\alpha_i = \langle r_i, r_0^* \rangle / \langle Ap_i, r_0^* \rangle$
9      $s_i = r_i - \alpha_i A p_i$
10     Compute and store $As_i$
11     Compute $\langle As_i, s_i \rangle$, $\langle As_i, As_i \rangle$
12     Compute $\langle As_i, r_0^* \rangle$
13     $\beta_i = -\frac{\langle As_i, r_0^* \rangle}{\langle Ap_i, r_0^* \rangle}$
14     $\omega_i = \langle As_i, s_i \rangle / \langle As_i, As_i \rangle$
15     $x_{i+1} = x_i + \alpha_i p_i + \omega_i s_i$
16     $r_{i+1} = s_i - \omega_i A s_i$
17
18
19     $p_{i+1} = r_{i+1} + \beta_i (p_i - \omega_i A_i)$
20     Compute $\langle r_{i+1}, r_0^* \rangle$
21 end
```

The classical BiCGStab method in Algorithm 3 requires a global synchronization after line 7 to compute α_i for use in line 8. Similarly, synchronizations are also required after line 11 to compute ω_i for use in line 14 and after line 17 to compute β for use in line 18. In analogy to the classical CG method, the search direction vector p_{i+1} (line 19) cannot be updated together with the approximated solution x_{i+1} (line 15) and the residual vector r_{i+1} (line 16). Consequently, additional loads from GPU main memory are required. Overall, two calls to routines for sparse matrix-vector products and at least eight calls to BLAS level 1 routines are needed in a conventional implementation of the classical BiCGStab method. Four host-device data transfers are required if each inner product induces a data transfer between host and device. An additional call to a BLAS level 1 routine and a host-device transfer are necessary if the residual norm is recomputed explicitly in each iteration.

The pipelined BiCGStab version in Algorithm 4 allows for improved data reuse by shifting the calculation of β_i to line 13 through

$$\begin{aligned}\beta_i &= \frac{\langle r_{i+1}, r_0^* \rangle}{\langle r_i, r_0^* \rangle} \times \frac{\alpha_i}{\omega_i} \\ &= \frac{\langle s_i - \omega_i A s_i, r_0^* \rangle}{\langle r_i, r_0^* \rangle} \times \frac{\langle r_i, r_0^* \rangle}{\omega_i \langle A p_i, r_0^* \rangle} \\ &= \frac{\langle s_i, r_0^* \rangle}{\omega_i \langle A p_i, r_0^* \rangle} - \frac{\langle A s_i, r_0^* \rangle}{\langle A p_i, r_0^* \rangle} \ .\end{aligned}$$

Using the orthogonality

$$\langle s_i, r_0^* \rangle = \langle r_i - \alpha_i A p_i, r_0^* \rangle = 0 \ .$$

one arrives at

$$\beta_i = -\frac{\langle A s_i, r_0^* \rangle}{\langle A p_i, r_0^* \rangle} \ ,$$

which we found to be numerically stable based on our experiments. This derivation of a pipelined BiCGStab version is similar to the modification of the classical CG method in Algorithm 1 to obtain the pipelined Algorithm 2. The minor price to pay for this rearrangement is the calculation of $\langle A s_i, r_0^* \rangle$ in line 12.

The next step is to apply kernel fusion extensively to the pipelined BiCGStab version in Algorithm 4. The calculation of $\langle A s_i, r_0^* \rangle$ can be fused with the sparse matrix product in line 10 together with the calculation of $\langle A s_i, s_i \rangle$ and $\langle A s_i, A s_i \rangle$ in line 11. Similarly, lines 6 and 7 are fused to a single kernel computing a matrix-vector product and the first reduction stage of the inner product. The vector update in line 9 is fused with the second reduction stages for the inner products needed to compute α_i in line 8. Since the residual norm is obtained via

$$\langle r_{i+1}, r_{i+1} \rangle = \langle s_i, s_i \rangle - 2\omega_i \langle s_i, A s_i \rangle + \omega_i^2 \langle A s_i, A s_i \rangle$$

for which $\langle s_i, A s_i \rangle$ and $\langle A s_i, A s_i \rangle$ are computed in line 11 and needed for the calculation of ω_i in line 14, we augment the update kernel for the computation of s_i in line 9 with the first reduction stage for $\langle s_i, s_i \rangle$. The partial results are transferred to the host together with the partial results for all other inner products after line 12, where β_i and ω_i are computed. Finally, the vector updates in lines 15, 16 and 19 as well as the first reduction stage for the inner product in line 20 are fused into another kernel.

Overall, the proposed pipelined BiCGStab implementation of Algorithm 4 consists of four kernel launches and one host-device transfer of the partial results from the four inner products $\langle A s_i, s_i \rangle$, $\langle A s_i, A s_i \rangle$, $\langle A s_i, r_0^* \rangle$, and $\langle s_i, s_i \rangle$:

- Compute the matrix-vector product in line 6 and the partial results for the two inner products required for α_i in line 8.
- Compute s_i in line 9 by redundantly computing α_i in each thread workgroup from the partial results of the inner products $\langle r_i, r_0^* \rangle$ and $\langle A p_i, r_0^* \rangle$.
- Compute and store $A s_i$ (line 10) and the partial results for the inner products in lines 11 and 12.
- Communicate all partial results for the inner products to the host, sum them there and perform a convergence check.
- Compute the vector updates in lines 15, 16, and 19 as well as the partial results for the inner product in line 20.

In comparison, the BiCGStab implementation proposed in [1] requires five kernel launches and three reductions, while a BLAS-based implementation of the classical method requires at least eight kernel launches and four additional kernel launches or host-device transfers for the second reduction stage in the computation of the inner products. Therefore, a roughly 60 percent performance improvement over the implementation in [1] and a two- to three-fold performance gain over purely BLAS-based implementations in the latency-dominated regime is expected, assuming that kernel launches and host-device transfers entail comparable latency.

3.3. GMRES. In contrast to the CG and BiCGStab methods, the GMRES method requires to store the full Krylov basis rather than only the current search direction vector, leading to an increase in the number of operations with each iteration [38]. To limit the computational expense, the GMRES method is typically restarted after m iterations, which is denoted by GMRES(m). Typical values for m are in the range of 20 to 50. Smaller values tend to slow down the overall convergence, whereas higher values increase the computational cost and may lead to more time spent in the orthogonalization rather than the matrix-vector product, making GMRES less attractive when compared to other methods.

In the following we consider the simpler GMRES method [47], which allows for a simpler solution of the minimization problem than the original formulation, but is otherwise comparable in terms of computational expense. Three methods for the computation of an orthonormal Krylov basis from a set of linearly independent vectors $\{v_k\}_{k=1}^{m}$ are common [37]:

- *Classical Gram-Schmidt*: The k-th vector of the basis is obtained as

$$v_k \leftarrow v_k - w_k \ , \ \text{where } w_k \leftarrow \sum_{i=1}^{k-1} \langle v_i, v_k \rangle v_i$$

  followed by a normalization of v_k.
- *Modified Gram-Schmidt*: An accumulation of round-off errors in the basis vectors v_k may lead to a loss of orthogonality. Better robustness has been observed when computing

$$v_k \leftarrow v_k - \langle v_i, v_k \rangle v_i$$

  for i from 1 to $k-1$ rather than forming a single update vector w_k. Although equivalent to the modified Gram-Schmidt method in exact arithmetic, the repeated computation of inner products $\langle v_i, v_k \rangle$ reduces the influence of round-off errors in finite precision arithmetic. The disadvantage of the modified Gram-Schmidt method is the reduced parallelism: Rather than computing all inner products $\langle v_i, v_k \rangle$ concurrently, only one inner product can be computed at a time, followed by a vector update.
- *Householder reflections*: The Krylov basis may also be obtained through Householder reflections $P_k = (I - \beta_k u_k u_k^{\mathrm{T}})$ with identity matrix I, suitably chosen scalars β_k, and Householder vectors u_k. Similar to the modified Gram-Schmidt method, the Householder reflections have to be applied sequentially to obtain the Krylov basis. Although the method allows for the computation of an orthonormal basis up to machine precision, the method is less regularly used in practice due its sequential nature and higher computational expense when compared to the Gram-Schmidt method.

Further algorithms employed for the orthogonalization in a multi-GPU setting with significantly different constraints in terms of communication can be found in [48].

A comparison of the classical restarted GMRES(m) method in simplified form [47] and a pipelined formulation is as follows:

Algorithm 5: Classical GMRES(m)

```
1  Choose $x_0$
2  $r_0 = b - Ax_0$
3  $\rho_0 = \|r_0\|_2$
4  $v_0 = r_0 = r_0/\rho_0$
5  $R_{i,j} = 0$ for $i, j \in \{1, \ldots, m\}$
6  for $i = 1$ to $m$ do
7      $v_i = Av_{i-1}$
8      for $j = 1$ to $i - 1$ do
9          $R_{j,i} = \langle v_j, v_i \rangle$
10     end
11     for $j = 1$ to $i - 1$ do
12         $v_i = v_i - R_{j,i} v_j$
13     end
14     $v_i = v_i / \|v_i\|$
15     $\xi_i = \langle r, v_i \rangle$
16     $r = r - \xi_i v_i$
17 end
18
19
20
21 Solve $R\eta = (\xi_1, \ldots, \xi_m)$
22 Update $x_m = \eta_1 r + \sum_{i=2}^{m} \tilde{\eta}_i v_{i-1}$
```

Algorithm 6: Pipelined GMRES(m)

```
1  Choose $x_0$
2  $r_0 = b - Ax_0$
3  $\rho_0 = \|r_0\|_2$
4  $v_0 = r_0 = r_0/\rho_0$
5  $R_{i,j} = 0$ for $i, j \in \{1, \ldots, m\}$
6  for $i = 1$ to $m$ do
7      $v_i = Av_{i-1}$
8      for $j = 1$ to $i - 1$ do
9          $R_{j,i} = \langle v_j, v_i \rangle$
10     end
11     for $j = 1$ to $i - 1$ do
12         $v_i = v_i - R_{j,i} v_j$
13     end
14     $v_i = v_i / \|v_i\|$
15     $\xi_i = \langle r, v_i \rangle$ (first stage)
16
17 end
18 for $i = 1$ to $m$ do
19     $\xi_i = \langle r, v_i \rangle$ (second stage)
20 end
21 Solve $R\eta = (\xi_1, \ldots, \xi_m)$
22 Update $x_m = \eta_1 r + \sum_{i=2}^{m} \eta_i v_{i-1}$
```

with $\tilde{\eta}_i = \eta_i + \eta_1 \xi_{i-1}$ to account for the updates of the residual r.

Both formulations use the classical Gram-Schmidt method for higher efficiency on massively parallel architectures such as GPUs. The main difference between the classical formulation in Algorithm 5 and the pipelined formulation in Algorithm 6 involves the update of the residual vector in line 16 of Algorithm 5. Because of the orthonormality of $\{v_k\}_{k=1}^{m}$, the inner product in line 15 remains unchanged when using exact arithmetic. Similarly, since the values ξ_i do not enter the Gram-Schmidt process, the values in the matrix R remain unchanged so that round-off errors only affect the right hand side vector in line 21. Our numerical experiments indicate that round-off errors in ξ_i are dominated by round-off errors in the classical Gram-Schmidt process and therefore do not affect the overall numerical stability of the solver. Also, the convergence monitors proposed in [47] do not require updates of the residual and are based on the values ξ_i only. Therefore, the full convergence history is still accessible before solving the minimization problem in line 21 and premature breakdown of the solver can be detected. Nevertheless, $m - 1$ unnecessary steps of the Gram-Schmidt process will be carried out if convergence is obtained right at the first iteration.

The benefit of removing the residual update from the Gram-Schmidt orthogonalization is that extensive kernel fusion can be applied to obtain an implementation of Algorithm 6 with almost no host-device communication. To begin, the reduction stage of the inner products in line 9 can be computed in two ways: The first option is a specialized matrix-vector routine for tall matrices if all Krylov vectors are stored as either the rows or the columns of a matrix. The second option is to fuse multiple inner products into the same kernel if all Krylov vectors reside in distinct buffers [35]. With both options the second reduction stage for computing $R_{j,i}$ in line 9 is fused with the vector updates in line 12 and also with the first reduction stage for computing $\|v_i\|$ needed in line 14. The normalization of v_i in line 14 is carried out by a kernel first computing the second reduction stage for $\|v_i\|$, then scaling v_i and directly computing the first reduction stage for the obtaining ξ_i in line 15. Consequently, no data transfer between host and device is required during the Gram-Schmidt orthogonalization. If desired, an asynchronous transfer of the intermediate values for ξ_i at the end of each orthogonalization step enables a better monitoring of the convergence process.

After the Gram-Schmidt process, the intermediate results for computing ξ_i are transferred to the host if not already transferred asynchronously, where the final values ξ_i are computed. Similarly, the triangular matrix R is transferred to the host. After the triangular system R is inverted, the result vector containing the values η_i is transferred to the device and the update of the result vector x_m is computed in line 22 using in a single kernel similar to the vector update in line 12.

Overall, the proposed implementation of the pipelined GMRES(m) method in Algorithm 6 requires two kernel launches in the first iteration and four kernel launches in subsequent iterations:

- Compute the matrix-vector product in line 7 and the first reduction stage for $\langle v_{i-1}, v_i\rangle$.
- Compute the first reduction stage for the inner products $\langle v_j, v_i\rangle$ in line 9 with j ranging from 1 to $i-2$.
- Compute the second reduction stage for the inner products $\langle v_j, v_i\rangle$ in line 9 for j from 1 to $i-1$, use the results directly for computing the vector update in 12, and compute the first reduction stage for $|v_i\|$.
- Compute the second reduction stage for $|v_i\|$. Use the result to normalize v_i and compute ξ_i.

A conventional implementation of the classical GMRES(m) method in Algorithm 5 requires at least seven kernel launches and may involve several host-device data exchanges per iteration. Thus, an up to two-fold performance gain in the latency-dominated regime is expected.

4. Benchmark Results.

The implementations proposed in this work are implemented in the upcoming 1.6.0 release of the free open-source linear algebra library ViennaCL [45] and are compared in the following with the implementations in the free open-source libraries CUSP 0.4.0 [9], MAGMA 1.5.0 [26] with INTEL MKL 11.0, and Paralution 0.7.0 [33]. Since both CUSP and MAGMA are based on CUDA, benchmark data for AMD GPUs could only be obtained with ViennaCL and Paralution. All four libraries are used in an out-of-the-box manner without additional target-specific tuning in order to reflect typical use cases.

All tests were carried out on Linux-based machines running the CUDA 6.0 SDK on NVIDIA GPUs with GPU driver version 331.20 and the AMD APP SDK 2.9 with GPU driver version 13.352.1014 on AMD GPUs. An NVIDIA Tesla C2050, an NVIDIA Tesla K20m, an AMD FirePro W9000, and an AMD FirePro W9100 were used for a comparison, representing the latest two generations of high-end workstation models from each vendor. Since all operations are limited by the available memory bandwidth, the obtained results are also representative for a broader range of high-end consumer GPUs with comparable memory bandwidth, yet with less main memory and at a fraction of the price.

In addition to GPU-benchmarks, we also compare with the execution times obtained with the CPU-based PETSc library [2, 3] on a dual-socket system equipped with INTEL Xeon E5-2620 CPUs, where parallel execution is based on the Message Passing Interface (MPI) [28] using MPICH 3.1 [29]. The fastest execution time from runs with 1, 2, 4, and 8 MPI ranks for each system size are taken for comparison. However, it should be noted that a comparison with a CPU-based library needs to be interpreted with care, because our benchmarks only compare the time taken per solver iteration, not the time required for copying the data to the GPU or for obtaining the result vector.

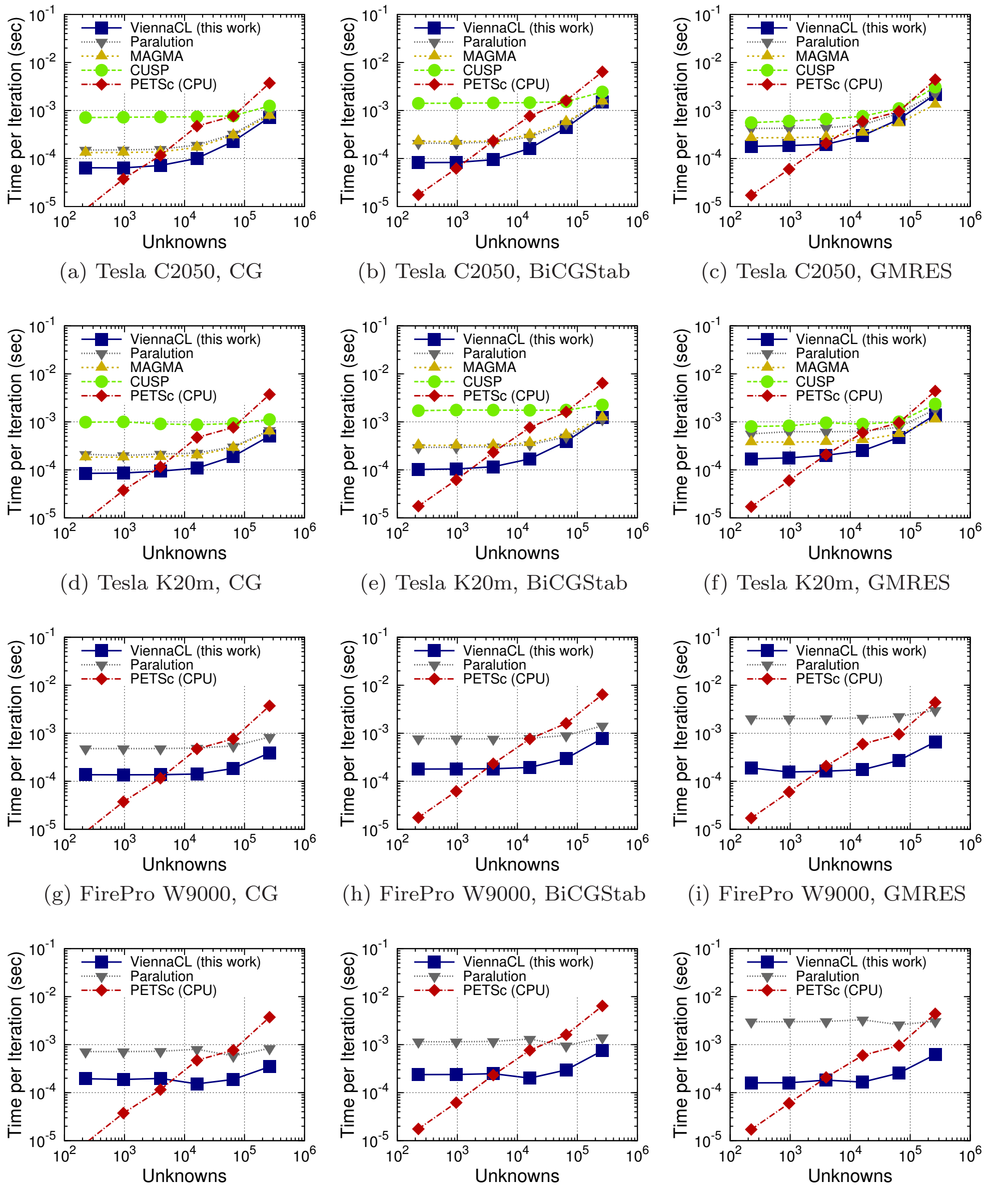
Execution times per iterative solver iteration are computed from the median value of ten solver runs with a fixed number of 30 iterations for each solver. In our experiments we have not observed any significant differences in the number of solver iterations required for convergence of the classical implementation and the pipelined implementation, hence the execution time per solver iteration is a suitable metric for comparison.

4.1. Finite Elements. We consider the execution time obtained with linear finite elements applied to the solution of the Poisson equation on the unit rectangle on a hierarchy of uniformly refined unstructured triangular meshes as a first benchmark. The resulting systems consist of 225, 961, 3 969, 16 129, 65 025 and 261 121 equations, respectively, and cover a broad range of typical system sizes solved on a single workstation. Results for CG, BiCGStab and GMRES using the ELLPACK sparse matrix format (cf. [7] for a description) are given in Fig. 4 for the four GPUs considered in our comparison. Similar results are obtained for other matrix formats, because the execution times are primarily dominated by latency effects. The case of large system matrices, where the various matrix formats become important, is considered in Section 4.2.

The performance gain of the proposed implementations is about three-fold for small systems for the CG and BiCGStab and therefore as projected based on the reduced number of kernels called. The pipelined GMRES method allows for up to two-fold performance gains on NVIDIA GPUs. The six-fold performance increase over Paralution for the GMRES method on AMD GPUs is not only due to pipelining and kernel fusion, but also an indication that there is potential for further optimizations of the implementation in Paralution.

Execution times for each solver iteration at system sizes below 10^4 are practically constant for both NVIDIA and AMD GPUs. Because this constant is about a factor of two larger for AMD GPUs and because AMD GPUs offer higher memory bandwidth, essentially constant execution times are obtained for systems with up to 10^5 unknowns for AMD GPUs. Only at system sizes above 10^5 unknowns, PCI-Express communication becomes negligible compared to kernel execution times, hence the performance of all libraries becomes similar and varies only mildly.

When comparing the execution times of GPU-based solvers with the execution times obtained with the CPU-based PETSc implementations, it is observed that the proposed pipelined implementations on GPUs are faster if systems carry more than about 3 000 unknowns on average. In contrast, at least 10 000 unknowns are needed with the conventional implementations in Paralution, MAGMA, and CUSP to outperform the CPU-based implementations in PETSc. Consequently, the roughly three-fold performance gains due to pipelining and kernel fusion are also reflected in the minimum system sizes required for better performance than the CPU-based implementations in each solver cycle.



(a) Tesla C2050, CG (b) Tesla C2050, BiCGStab (c) Tesla C2050, GMRES

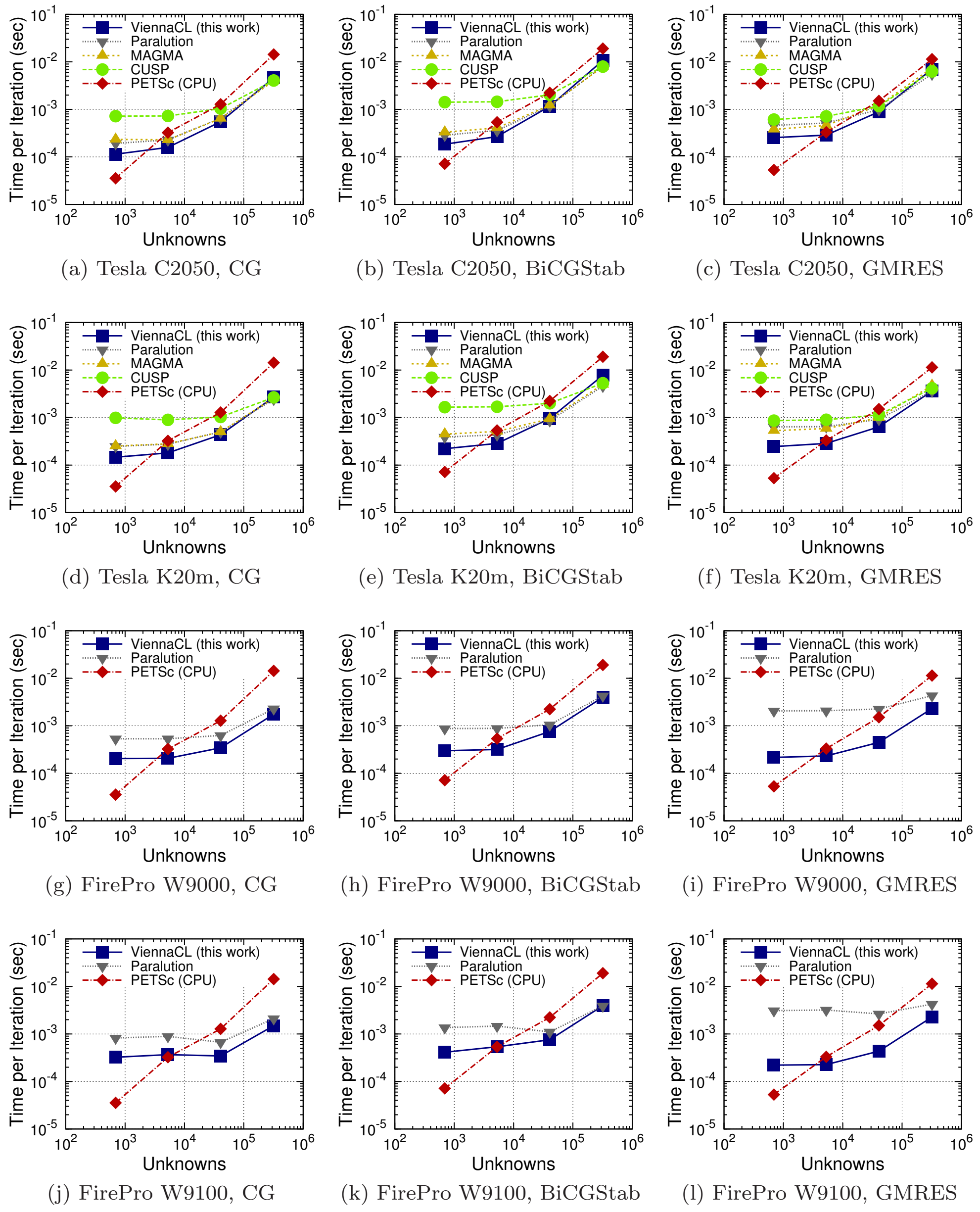
(d) Tesla K20m, CG (e) Tesla K20m, BiCGStab (f) Tesla K20m, GMRES

(g) FirePro W9000, CG (h) FirePro W9000, BiCGStab (i) FirePro W9000, GMRES

(j) FirePro W9100, CG (k) FirePro W9100, BiCGStab (l) FirePro W9100, GMRES

FIG. 4. *Comparison of the solver time required per iteration for solving the Poisson equation using finite elements on triangular grids in two spatial dimensions. The proposed pipelined implementations in ViennaCL significantly outperform other GPU-accelerated solver libraries for system sizes below* 10^5 *thanks to a smaller number of kernel launches and better data reuse.*

The second benchmark compares the execution time obtained with linear finite elements for numerical solutions of the linear elasticity model in three spatial dimensions. A hierarchy of uniformly refined tetrahedral meshes of the unit cube was used, resulting in system sizes of 693, 5 265, 40 725, and 319 725, respectively. Compared to the first benchmark, the average number of unknowns per row increases from about



(a) Tesla C2050, CG (b) Tesla C2050, BiCGStab (c) Tesla C2050, GMRES

(d) Tesla K20m, CG (e) Tesla K20m, BiCGStab (f) Tesla K20m, GMRES

(g) FirePro W9000, CG (h) FirePro W9000, BiCGStab (i) FirePro W9000, GMRES

(j) FirePro W9100, CG (k) FirePro W9100, BiCGStab (l) FirePro W9100, GMRES

FIG. 5. *Comparison of the solver time required per iteration for solving the linear elasticity model using finite elements in three spatial dimensions. The proposed pipelined implementations in ViennaCL outperform other libraries for system sizes beyond* 10^5 *thanks to a smaller number of kernel launches and better data reuse.*

7 to 60 for the largest system, resulting in a higher share of the execution time being spent on the sparse matrix-vector product. Nevertheless, the results in Fig. 5 show a similar trend as the results in Fig. 4: For small matrix sizes, two- to three-fold performance gains of the pipelined implementations are obtained. Although the system matrix carries more nonzeros than in the first benchmark, about 10^4 unknowns

Name	Rows	Nonzeros	Nonzeros/Row	Symmetric
pdb1HYS	36 417	4 344 765	119.31	yes
cant	62 451	4 007 383	64.17	yes
consph	83 334	6 010 480	72.13	yes
shipsec1	140 874	7 813 404	55.46	yes
pwtk	217 918	11 643 424	53.39	yes
rma10	46 835	2 374 001	50.69	no
cop20k_A	121 192	2 624 331	21.65	no
scircuit	170 998	958 936	5.61	no
mac_econ_fwd500	206 500	1 273 389	6.17	no
RM07R	381 689	37 464 962	98.16	no
Hamrle3	1 447 360	5 514 242	3.81	no
kkt_power	2 063 494	13 612 663	7.08	no

Table 1

Summary of symmetric and non-symmetric matrices taken from the Florida Sparse Matrix Collection [12] for comparison. These matrices represent the set of real-valued, floating point square matrices used in earlier contributions on optimizing sparse matrix-vector products [7, 21].

on NVIDIA GPUs and 10^5 unknowns on AMD GPUs are required such that kernel execution times hide performance penalties due to PCI-Express communication. Performance differences for CG and BiCGStab between ViennaCL and Paralution on the AMD GPUs are less pronounced than for the first benchmark, but still significant.

4.2. Florida Sparse Matrix Collection. The performance of the proposed pipelined implementations is compared in the following for matrices from the Florida Sparse Matrix Collection [12] used for the evaluation of sparse matrix-vector products in the past [7, 21]. While the focus in the previous section was on demonstrating the benefit of the proposed implementations for small to medium-sized systems, the purpose of this section is the show that the proposed implementations are also competitive for large systems. Thus, Paralution and MAGMA are a-priori expected to provide the best performance, since they use the vendor-tuned sparse matrix-vector product kernels from NVIDIA's CUSPARSE library. In contrast, our implementations in ViennaCL rely fused kernels, while CUSP implements its own set of sparse matrix-vector product kernels [7].

Since OpenCL does not support complex arithmetic natively, we restrict our benchmark to real-valued matrices listed in Tab. 1. The symmetric, positive definite matrices are used for benchmarking the implementations of the CG method, while the non-symmetric matrices are used for benchmarking the implementations of the BiCGStab and GMRES methods. All sparse matrix formats available in the respective library are compared using implementations in CUDA and, if available, OpenCL. The fastest combination is then taken for the comparison, since such a procedure resembles the typical user who picks the fastest sparse matrix format and the programming model with best performance for a particular application.

The benchmark results for the CG method in Fig. 6 show that the proposed solver implementation outperforms all other solver libraries for three out of five matrices. The difference is particularly pronounced on the AMD GPUs, where the performance of our proposed implementation is up to twice as high as the performance of Paralution. A comparison of absolute execution times also shows that the AMD GPUs provide a better overall performance due to their higher memory bandwidth.

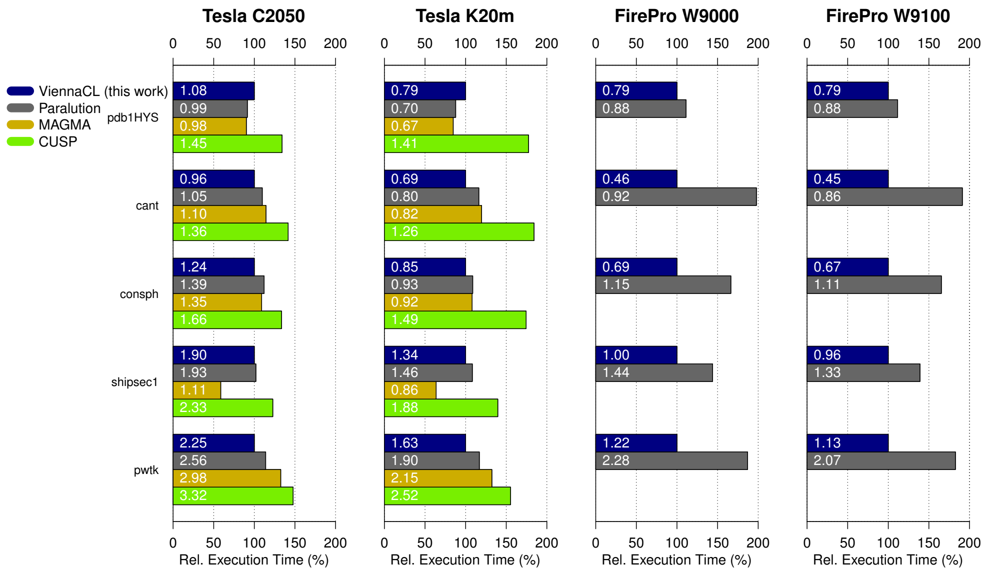


FIG. 6. *Comparison of execution times per CG solver iteration for different systems from the Florida Sparse Matrix Collection relative to the proposed pipelined implementations. Absolute execution times in milliseconds are given inside each bar.*

The comparison of execution times for the BiCGStab method in Fig. 7 shows similar performance of ViennaCL, Paralution, and MAGMA on average: Depending on the device and the matrix considered, either of the three is the best choice. Since the proposed implementations do not contain any device-specific optimizations, further tuning may provide further performance gains, whereas implementations using vendor-tuned kernels do not have this option. In contrast, the custom sparse matrix-vector product kernels in CUSP result in about 50 percent higher execution times on average, hence we conclude that the higher performance of our custom implementations stems from the extensive use of kernel fusion and pipelining. On AMD GPUs, the performance gain over Paralution is again up to two-fold. Similar to the results of the benchmark of the CG method, AMD GPUs provide slightly higher overall performacne than the NVIDIA GPUs because of to their higher memory bandwidth.

The benchmark results obtained for the GMRES method are depicted in Fig. 8 and show the same trend as the results obtained when comparing the implementations of the BiCGStab method. Paralution exceeds the available memory for the matrices `cop20k_A` and `kkt_power`. MAGMA, on the other hand, fails for the matrix `RM07R` because a check for positive definiteness of the system matrix fails. The OpenCL-based implementation in Paralution for AMD GPUs is not yet optimized.

Finally, execution times for the proposed implementations of the three iterative solvers using CUDA and OpenCL are compared in Fig. 9. In most cases, the obtained execution times of CUDA and OpenCL are within ten percent. Only in the case of BiCGStab on the Tesla K20m, the OpenCL implementation shows better performance than the CUDA implementation. We explain this with better device-specific optimizations of the OpenCL just-in-time compiler on the Tesla K20m, while the CUDA code - even though compiled with with the `sm_35` architecture flag - was not taking full potential of the newer Kepler architecture in the Tesla K20m.

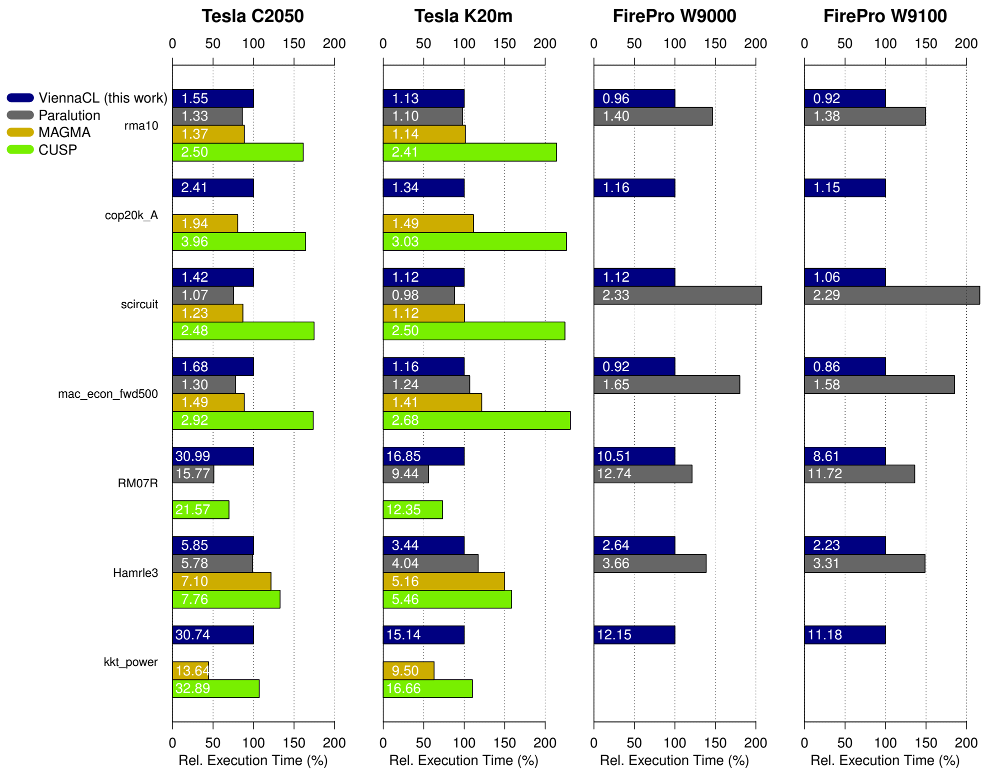


Fig. 7. *Comparison of execution times per BiCGStab solver iteration for different systems from the Florida Sparse Matrix Collection relative to the proposed pipelined implementations. Absolute execution times in milliseconds are given inside each bar. The* `cop20k_A` *and the* `kkt_power` *matrices could not be tested with Paralution due to segmentation faults. The* `RM07R` *matrix could not be run with MAGMA since it did not pass a check for positive definiteness.*

5. Conclusion. The proposed pipelined implementations of the CG, BiCGStab and GMRES methods address the latency-induced performance penalties of GPU-accelerated implementations for sparse systems with less than about 10^5 unknowns. Our comparison with other solver packages shows up to three-fold performance gains for practically relevant problem sizes between 10^4 and 10^5 unknowns. A comparison for larger systems shows that the proposed implementations using fused kernels shows also performance competitive with implementations built on top of vendor-tuned kernels. As a consequence, our results suggest that future efforts on the optimization of compute kernels should not be restricted to standard BLAS-like kernels, but additional performance can be obtained when taking fused kernels into account. For example, not only the sparse matrix-vector product kernel, but also a kernel computing the sparse matrix-vector product plus the first reduction stage of inner products involving the result vector may offer superior performance for iterative solvers from the family of Krylov methods.

While an extensive use of pipelining and kernel fusion addresses latency issues and limited memory bandwidth, it also brings new challenges for the design of scientific software. To leverage the full potential of modern hardware, it is no longer sufficient to only use a fairly small set of vendor-tuned BLAS-kernels, but instead provide modular building blocks for minimizing communication of data.

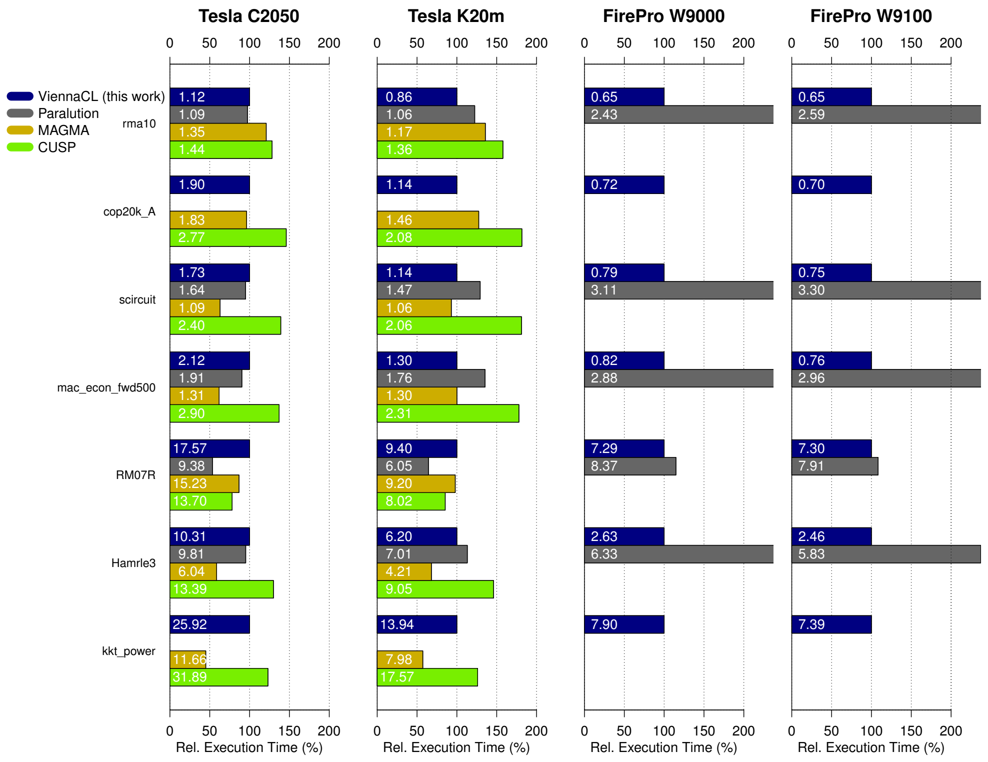


Fig. 8. *Comparison of execution times per GMRES solver iteration for different systems from the Florida Sparse Matrix Collection relative to the proposed pipelined implementations. Absolute execution times in milliseconds are given inside each bar. The* `cop20k_A` *and the* `kkt_power` *matrices could not be tested with Paralution due to segmentation faults.*

Future GPUs as well as CPUs will see gains in memory bandwidth, but the latency induced by the PCI-Express bus will not change substantially. Therefore, the minimum system size required to get any performance gains on GPUs over CPUs will continue to grow. As a consequence, the replacement of the PCI-Express bus with a interconnect technology of lower latency is essential for making accelerators more attractive. Integrations of GPU units on the CPU die are one possible path to achieve lower latency. However, no benefit over a well-optimized, purely CPU-based implementations can be expected for the memory-bandwidth limited operations in iterative solvers, if both the accelerator and the CPU core share the same memory link.

The techniques applied in this work can also be extended to preconditioned iterative solvers. Not only can the application of the preconditioner be possibly fused with vector updates, but also the setup stage can benefit from fusing as many operations as possible into the same kernel. A rigorous application of these techniques to preconditioners is left for future work.

Acknowledgment. This work has been supported by the Austrian Science Fund (FWF) through the grants P23296 and P23598. The authors also thank AMD and NVIDIA for hardware donations and Joachim Schöberl for providing access to a system equipped with two NVIDIA K20m for benchmarking purposes.

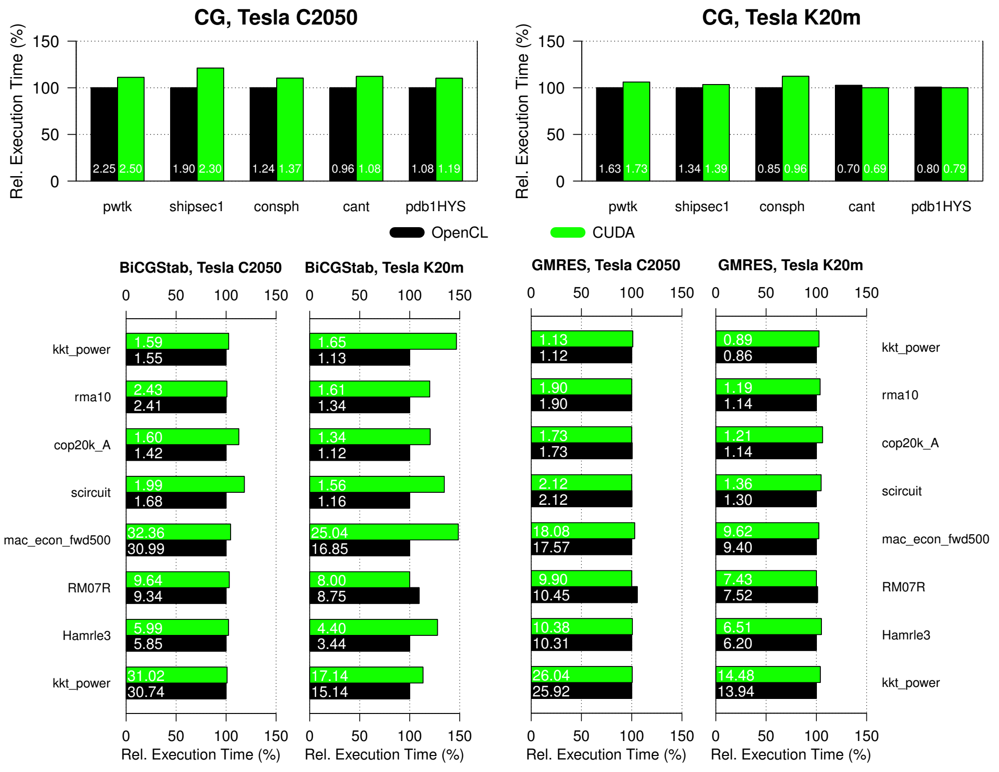


FIG. 9. *Comparison of execution times obtained with CUDA and OpenCL for the CG method (top), the BiCGStab method (left), and the GMRES method (right). Relative execution times are with respect to the faster framework. Absolute execution times in milliseconds are given inside each bar. Overall, the performance differences of CUDA and OpenCL are negligible in practice, even though OpenCL shows slightly better performance overall.*

REFERENCES

[1] H. Anzt, S. Tomov, P Luszczek, I. Yamazaki, J. Dongarra, and W. Sawyer, *Optimizing Krylov Subspace Solvers on Graphics Processing Units*, Tech. Report 583, University of Tennessee, 2014. `http://www.eecs.utk.edu/resources/library/583`.

[2] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman-McInnes, K. Rupp, B. F. Smith, and H. Zhang, *PETSc Users Manual*, Tech. Report ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[3] ———, *PETSc Web page*. `http://www.mcs.anl.gov/petsc`, 2014.

[4] D. Barkai, K. J. M. Moriarty, and C. Rebbi, *A Modified Conjugate Gradient Solver for Very Large Systems*, Comp. Phys. Comm., 36 (1985), pp. 1 – 8.

[5] M. M. Baskaran and R. Bordawekar, *Optimizing Sparse Matrix-Vector Multiplication on GPUs*, IBM RC24704, (2008).

[6] N. Bell, S. Dalton, and L. Olson, *Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods*, SIAM J. Sci. Comp., 34 (2012), pp. C123–C152.

[7] N. Bell and M. Garland, *Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors*, in Proc. HPC Netw., Stor. Anal., SC '09, ACM, 2009, pp. 18:1–18:11.

[8] A.T. Chronopoulos and C.W. Gear, *s-step Iterative Methods for Symmetric Linear Systems*, Journal Comp. Appl. Math., 25 (1989), pp. 153–168.

[9] *CUSP Library*. `http://cusplibrary.github.io/`.

[10] M.M. Dehnavi, D.M. Fernandez, J. Gaudiot, and D.D. Giannacopoulos, *Parallel Sparse Approximate Inverse Preconditioning on Graphic Processing Units*, IEEE Trans. Par. Dist. Sys., 24 (2013), pp. 1852–1862.

[11] J. Fang, A. L. Varbanescu, and H. Sips, *A Comprehensive Performance Comparison of CUDA and OpenCL*, in Proc. Intl. Conf. Par. Proc., 2011, pp. 216–225.

[12] *Florida Sparse Matrix Collection.* `http://www.cise.ufl.edu/research/sparse/matrices/`.

[13] R. Gandham, K. Esler, and Y. Zhang, *A GPU Accelerated Aggregation Algebraic Multigrid Method*, arXiv e-Print 1403.1649, (2014).

[14] P. Ghysels and W. Vanroose, *Hiding Global Synchronization Latency in the Preconditioned Conjugate Gradient Algorithm*, Par. Comp., 40 (2014), pp. 224–238.

[15] M. J. Harvey and G. De Fabritiis, *Swan: A Tool for Porting CUDA Programs to OpenCL*, Comp. Phys. Comm., 182 (2011), pp. 1093–1099.

[16] M. R. Hestenes and E. Stiefel, *Methods of Conjugate Gradients for Solving Linear Systems*, J. Res. Natl. Bureau of Standards, 49 (1952), pp. 409–436.

[17] T. Jacques, L. Nicolas, and C. Vollaire, *Electromagnetic Scattering with the Boundary Integral Method on MIMD Systems*, in High-Performance Computing and Networking, vol. 1593 of LNCS, Springer, 1999, pp. 1025–1031.

[18] K. Karimi, N. G. Dickson, and F. Hamze, *A Performance Comparison of CUDA and OpenCL*, arXiv e-Print 1005.2581, (2010).

[19] K. Kim and V. Eijkhout, *Scheduling a Parallel Sparse Direct Solver to Multiple GPUs*, in Proc. IEEE IPDPS, 2013, pp. 1401–1408.

[20] B. Krasnopolsky, *The Reordered BiCGStab Method for Distributed Memory Computer Systems*, Procedia Comp. Sci., 1 (2010), pp. 213–218.

[21] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, *A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiply on Modern Processors with Wide SIMD Units*, arXiv e-Print 1307.6209, (2013).

[22] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, *Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU*, in Proc. Intl Symp. Comp. Arch., ACM, 2010, pp. 451–460.

[23] R. Li and Y. Saad, *GPU-Accelerated Preconditioned Iterative Linear Solvers*, J. Supercomp., 63 (2013), pp. 443–466.

[24] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, *Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors*, in Proc. Supercomp., ACM, 2013, pp. 273–282.

[25] M. Lukash, K. Rupp, and S. Selberherr, *Sparse Approximate Inverse Preconditioners for Iterative Solvers on GPUs*, in Proc. HPC Symp., SCS, 2012, pp. 13:1–13:8.

[26] *MAGMA Library.* `http://icl.cs.utk.edu/magma/`.

[27] G. Martinez, M. Gardner, and Wu chun Feng, *CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures*, in IEEE Intl. Conf. Par. Dist. Sys., 2011, pp. 300–307.

[28] MPI Forum, *Message Passing Interface Forum.* `http://www.mpi-forum.org/`.

[29] *MPICH Library.* `http://www.mpich.org/`.

[30] M. Naumov, *Preconditioned Block-Iterative Methods on GPUs*, PAMM, 12 (2012), pp. 11–14.

[31] J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Scalable Parallel Programming with CUDA*, Queue, 6 (2008), pp. 40–53.

[32] *OpenCL.* `http://www.khronos.org/opencl/`.

[33] *Paralution Library.* `http://www.paralution.com/`.

[34] C. Richter, S. Schops, and M. Clemens, *GPU Acceleration of Algebraic Multigrid Preconditioners for Discrete Elliptic Field Problems*, IEEE Trans. Magn., 50 (2014), pp. 461–464.

[35] K. Rupp, Ph. Tillet, B. Smith, K.-T. Grasser, and A. Jüngel, *A Note on the GPU Acceleration of Eigenvalue Computations*, in AIP Proc., volume 1558, 2013, pp. 1536–1539.

[36] Y. Saad, *Practical Use of Polynomial Preconditionings for the Conjugate Gradient Method*, SIAM J. Sci. Stat. Comp., 6 (1985), pp. 865–881.

[37] ———, *Iterative Methods for Sparse Linear Systems, Second Edition*, SIAM, 2003.

[38] Y. Saad and M. H. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comp., 7 (1986), pp. 856–869.

[39] E. Saule, K. Kaya, and Ü. Catalyürek, *Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi*, in Parallel Processing and Applied Mathematics, LNCS, Springer Berlin Heidelberg, 2014, pp. 559–570.

[40] W. Sawyer, C. Vanini, G. Fourestey, and R. Popescu, *SPAI Preconditioners for HPC Applications*, Proc. Appl. Math. Mech., 12 (2012), pp. 651–652.

[41] O. Schenk, M. Christen, and H. Burkhart, *Algorithmic Performance Studies on Graphics Processing Units*, J. Par. Dist. Comp., 68 (2008), pp. 1360–1369.

[42] R. Strzodka and D. Göddeke, *Pipelined Mixed Precision Algorithms on FPGAs for Fast*

and Accurate PDE Solvers from Low Precision Components, in Proc. IEEE FCCM, IEEE Computer Society, 2006, pp. 259–270.

[43] H. van der Vorst, *Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comp., 13 (1992), pp. 631–644.

[44] *VexCL Library*. `https://github.com/ddemidov/vexcl/`.

[45] *Vienna Computing Library (ViennaCL)*. `http://viennacl.sourceforge.net/`.

[46] M. Wagner, K. Rupp, and J. Weinbub, *A Comparison of Algebraic Multigrid Preconditioners Using Graphics Processing Units and Multi-core Central Processing Units*, in Proc. HPC Symp., SCS, 2012, pp. 2:1–2:8.

[47] H. F. Walker and L. Zhou, *A Simpler GMRES*, Num. Lin. Alg. Appl., 1 (1994), pp. 571–581.

[48] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra, *Improving the Performance of CA-GMRES on Multicores with Multiple GPUs*, in Proc. IEEE IPDPS, IEEE Computer Society, 2014, pp. 382–391.

[49] L.T. Yang and Richard P. Brent, *The Improved BiCGStab Method for Large and Sparse Unsymmetric Linear Systems on Parallel Distributed Memory Architectures*, in Proc. Alg. Arch. Par. Proc., 2002, pp. 324–328.

[50] R. Yokota, J. P. Bardhan, M. G. Knepley, L. A. Barba, and T. Hamada, *Biomolecular Electrostatics using a Fast Multipole BEM on up to 512 GPUs and a Billion Unknowns*, Comp. Phys. Comm., 182 (2011), pp. 1272 – 1283.