SIAM J. SCI. COMPUT.
Vol. 38, No. 5, pp. S412–S439

VIENNACL—LINEAR ALGEBRA LIBRARY FOR MULTI- AND MANY-CORE ARCHITECTURES*

KARL RUPP†, PHILIPPE TILLET‡, FLORIAN RUDOLF†, JOSEF WEINBUB†, ANDREAS MORHAMMER†, TIBOR GRASSER†, ANSGAR JÜNGEL§, AND SIEGFRIED SELBERHERR†

Abstract. CUDA, OpenCL, and OpenMP are popular programming models for the multicore architectures of CPUs and many-core architectures of GPUs or Xeon Phis. At the same time, computational scientists face the question of which programming model to use to obtain their scientific results. We present the linear algebra library ViennaCL, which is built on top of all three programming models, thus enabling computational scientists to interface to a single library, yet obtain high performance for all three hardware types. Since the respective compute back end can be selected at runtime, one can seamlessly switch between different hardware types without the need for error-prone and time-consuming recompilation steps. We present new benchmark results for sparse linear algebra operations in ViennaCL, complementing results for the dense linear algebra operations in ViennaCL reported in earlier work. Comparisons with vendor libraries show that ViennaCL provides better overall performance for sparse matrix-vector and sparse matrix-matrix products. Additional benchmark results for pipelined iterative solvers with kernel fusion and preconditioners identify the respective sweet spots for CPUs, Xeon Phis, and GPUs.

Key words. ViennaCL, iterative solvers, CUDA, OpenCL, OpenMP, CPU, GPU, Xeon Phi

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

DOI. 10.1137/15M1026419

1. Introduction.

The availability of fast implementations of linear algebra operations is crucial for the efficient solution of many problems in scientific computing. Typical use cases are the inversion of dense or sparse matrices, but also application areas such as graph algorithms or text mining, where linear algebra operations are less apparent, yet are the limiting factor for overall performance. In order to maximize the numerical resolution possible for a given time budget, optimized software libraries for linear algebra operations are broadly available. In the era of sequential processing on single-core processors, reasonable performance can often be obtained with custom implementations. However, this is no longer true for modern multicore central processing units (CPUs) or graphics processing units (GPUs): First, additional knowledge of the underlying hardware must be acquired, which can quickly consume a substantial amount of time. Second, efficient parallel algorithms may differ substantially from efficient algorithms for sequential execution. For historical reasons, sequential algorithms are covered more prominently in curricula or text books, so

*Received by the editors June 16, 2015; accepted for publication (in revised form) July 1, 2016; published electronically October 27, 2016.

http://www.siam.org/journals/sisc/38-5/M102641.html

Funding: This work has been supported by the Austrian Science Fund (FWF), grants P23296 and P23598, by the European Research Council (ERC) through the grant 247056 MOSILSPIN, and by Google via the Google Summer of Codes 2011, 2012, 2013, and 2014.

†Institute for Microelectronics, TU Wien, Gußhausstraße 27-29/E360, A-1040 Wien, Austria and Institute for Analysis and Scientific Computing, TU Wien, Wiedner Hauptstraße 8-10/E101, A-1040 Wien, Austria (rupp@iue.tuwein.ac.at, rudolf@iue.tuwein.ac.at, weinbub@iue.tuwein.ac.at, morhammer@iue.tuwein.ac.at, grasser@iue.tuwein.ac.at, selberherr@iue.tuwein.ac.at).

‡School of Engineering and Applied Sciences, Harvard University, 29 Oxford Street, Cambridge, MA 02138 (ptillet@g.harvard.edu).

§Institute for Analysis and Scientific Computing, TU Wien, Wiedner Hauptstraße 8-10/E101, A-1040 Wien, Austria (juengel@asc.tuwien.ac.at).

additional time must be spent on finding proper algorithms in addition to the actual implementations.

Several programming approaches for modern hardware are available: OpenMP [17] relies on code annotations via compiler directives for generating multithreaded programs. Such an approach is particularly attractive for legacy applications, because—ideally—existing code does not need to be modified. At the same time, OpenMP is frequently used in new projects, because it is supported by all major compilers and platforms, thus making only a minimum of assumptions about the target system. Based on valuable experiences from OpenACC,[1] the OpenMP forum also standardized support for accelerators such as GPUs in OpenMP 4.0.

If full control over individual threads is desired, the POSIX standard for threads (pthreads) can be used on all POSIX-conformant operating systems as well as Microsoft Windows. In fact, compilers typically transform OpenMP-annotated code into executables which ultimately call the respective pthreads functionality. A direct use of pthreads instead of OpenMP-annotated code also prevents certain portability problems which can arise, if object files were compiled with different versions of OpenMP. Other multithreading approaches such as CILK [11] or the Threading Building Blocks library[2] [41] exist, but are used less frequently and therefore not considered further in this work.

The message passing interface (MPI) [21] is a language-independent communications protocol and the de facto standard for parallel systems. In contrast to OpenMP, where nonlocal data are per default shared across all threads, data are by default local to each process in the parallel run, and all data exchange is explicit. While MPI has primarily been used in distributed memory systems in the past, current benchmarks also demonstrate comparable or even better performance of purely MPI-based codes on shared memory machines than OpenMP-based codes [13]. However, the choice of a purely MPI-based execution model (*flat MPI*) including shared memory additions in the MPI 3.0 standard versus a hybrid model, where data exchange across distributed memory domains is handled with MPI and shared memory parallelism is handled with, e.g., OpenMP, is still debated in the scientific community [32]. For the remainder of this work we will focus on parallelism for shared memory machines, hence we will only comment occasionally on MPI.

The compute unified device architecture (CUDA) [39] and the open computing language (OpenCL)[3] [52] are the two main programming models for GPUs. CUDA is a proprietary programming model by NVIDIA and consequently only targets NVIDIA GPUs. Compute kernels are written in a way very similar to C or C++, so existing code may be ported with only small changes. Compilation requires the compiler wrapper NVCC, which separates the GPU-specific code from the host code. GPU-specific code is then either compiled into a native binary code, or a portable intermediate representation. After replacing the original GPU-specific code with the transformed code, the modified source is passed on to the host compiler. However, only some host compilers are supported by NVCC, which can cause problems, if the host compiler is updated, but the CUDA framework is not.

OpenCL is a royalty-free, cross-platform standard for parallel programming. Device code is written in a kernel language similar to C and passed to a just-in-time compiler for the respective target platform at runtime, which results in several key

[1] http://www.openacc.org/
[2] https://software.intel.com/en-us/intel-tbb
[3] http://www.khronos.org/opencl/

differences from CUDA: First, no compiler wrapper like NVCC is required, allowing for maximum flexibility in the choice of the host compiler. The resulting application only must be linked to the shared OpenCL library. Second, the just-in-time compilation step enables high flexibility and optimization potential for the target hardware, but entails additional overhead in each run. While a significant share of this overhead can be eliminated by buffering the binaries obtained from the just-in-time compilation on the local filesystem, this option is likely not available on all machines.

A computational scientist working in typical application areas such as materials science, climate modeling, or computational fluid dynamics has a high interest in using the computational resources of a given machine as efficiently as possible. In an ideal world, the computational scientist should not have to deal with hardware details of a given workstation or supercomputer, but instead be provided with a performance-portable, machine-independent set of libraries which take care of the machine's details. These performance-portable libraries must provide reasonable default values, but at the same time provide appropriate optimization switches for applications for which every percent of performance is important. Such switches must be available at runtime, otherwise each user of a particular application must be exposed to the build process—the build process becomes part of the application programming interface (API). However, a build-time configuration is unacceptable in nonscientific software, but somehow passes for the norm in scientific packages [12]. This situation may be a direct consequence of the low credit for writing and maintaining scientific software, which makes it very hard for authors of scientific software to pursue academic careers [6, 7].

Since CUDA, OpenCL, and OpenMP are in widespread use in the scientific community, a software library for current and future parallel architectures should support all three programming models. The Vienna Computing Library (ViennaCL)[4] [43] aims to provide individual building blocks as well as larger blocks of linear algebra functionality with CUDA, OpenCL, and OpenMP compute backends, so that functionality is available to users irrespective of their chosen programming model. Moreover, the API of ViennaCL enables users to build their own algorithms using a high-level C++ interface and run on either compute backend rather than dealing with low-level details of a single compute backend. As the comparison of other library approaches with similar functionality in section 2 shows, the ability to switch between compute backends at runtime is a unique feature of ViennaCL. In section 3 we discuss generic requirements on a linear algebra library, while in section 4 we discuss how ViennaCL seeks to fulfill the same. The library architecture is discussed in section 5, providing an overview of the API as well as the internals of ViennaCL. Benchmark results in section 6 demonstrate that ViennaCL provides performance comparable to or better than vendor-tuned libraries for sparse matrix-vector products and sparse matrix-matrix products. These results complement earlier work, which reported competitive performance of ViennaCL for dense linear algebra operations [44, 54]. In addition, benchmark results for pipelined iterative solvers with kernel fusion and two important types of preconditioners allow for a comparison of solver performance on different hardware platforms. We discuss lessons learnt from the development of ViennaCL and discuss future directions in section 7, where we do not shy away from a discussion of early design decision which ultimately turned out to be nonoptimal. In retrospect, these earlier design errors reflect general issues in designing scientific software libraries in the ecosystem of programming models available today.

[4] http://viennacl.sourceforge.net/

2. Related work. Although GPUs have been used for general purpose computations for several years already, the library ecosystem is still relatively small. Most libraries are provided by vendors directly: NVIDIA provides several optimized libraries including dense and sparse linear algebra for their GPUs, but some developers are discouraged by the implied vendor lock and its impact on computational science. One exception is the Thrust library[5] [31] included in the CUDA development kit: Thrust provides parallel primitives through a generic C++ API where an OpenMP backend is available as a fallback mechanism if a CUDA device is not available. AMD provides the OpenCL-based clBLAS library,[6] which is a priori also suitable for hardware from other vendors. However, compute kernels are optimized exclusively for AMD GPUs, resulting in the absence of any performance portability across vendors. INTEL offers optimized implementations for x86 CPUs as well as Xeon Phis via their Math Kernel Library (MKL).[7] While an OpenCL driver for targeting integrated GPUs is provided, no optimized linear algebra functionality for GPUs is offered by INTEL.

Nonvendor libraries with support for computations on GPUs are often focused on only a few operations using one particular programming model. For example, clSpMV [53] and yaSpMV [60] provide several sparse matrix-vector product kernels and their own sparse matrix vector storage formats in OpenCL. While such approaches perform well in isolated benchmarks, their use in practice often suffers from additional overhead such as data conversion at library boundaries, which can have a considerable hit on the overall application's performance. Consequently, we focus on packages with a sufficiently large set of linear algebra functionality allowing users to compose algorithms from a rich set of small building blocks within the same library. The emphasis is on outlining differences of the individual packages from ViennaCL from a user's perspective.

2.1. Boost.Compute. Boost.Compute[8] provides parallel OpenCL-based implementations of functionality implemented sequentially in the C++ standard template library. The inclusion into the popular Boost C++ libraries[9] [50] resulted in a broader availability of the library, but at the same time leads to a huge dependence for applications which aim to stay lightweight. Functionality very similar to NVIDIA's Thrust library is offered, for example sort and prefix sum operations. However, the just-in-time compilation of OpenCL kernels enforces a stronger separation between host code and device code when using Boost.Compute compared to using Thrust. Only very basic vector and matrix operations are provided.

2.2. CUSP. CUSP[10] [18] is a standalone, CUDA-based open source library for sparse linear algebra primarily developed by NVIDIA. It provides several sparse matrix formats [10], iterative solvers, and advanced preconditioners including algebraic multigrid methods [9]. An OpenMP-accelerated compute backend is also available as a fallback mechanism. NVIDIA also supplemented CUSP, which is a header-only C++ library, with the CUDA-only CUSPARSE library,[11] which provides a C interface, simplifying the use with other languages such as FORTRAN.

[5] http://thrust.github.io/
[6] https://github.com/clMathLibraries/clBLAS
[7] http://software.intel.com/en-us/intel-mkl
[8] https://github.com/boostorg/compute
[9] http://www.boost.org/
[10] http://cusplibrary.github.io/
[11] http://developer.nvidia.com/cusparse/

2.3. MAGMA, clMAGMA, MAGMA MIC. MAGMA,[12] clMAGMA, and MAGMA MIC [1] implement linear algebra functionality for CUDA, OpenCL, and OpenMP targeting Xeon Phis, respectively. All three libraries aim to provide optimized implementations of functionality from the BLAS and LAPACK libraries. The APIs of MAGMA, clMAGMA, and MAGMA MIC are similar, but mutually incompatible. The three libraries share no common code base and there is a broad diversity in functionality offered. For example, MAGMA now provides operations for sparse linear algebra, which is absent in clMAGMA and MAGMA MIC.

2.4. PARALUTION. Paralution[13] [55] is a C++ software library with a focus on providing iterative solvers and preconditioners for sparse systems of linear equations. It provides CUDA, OpenMP, and OpenCL compute backends, out of which one has to be selected at compile time. Many preconditioners, however, are not available with the OpenCL backend. Support for distributed memory machines is available via an MPI layer.

2.5. VexCL. VexCL[14] [20] is a vector expression library for OpenCL and CUDA written in C++11. The library provides a convenient syntax for vector operations, either on a single or multiple devices accessible from the same shared memory domain. Similar to PARALUTION, the compute backend must be selected at compile time. VexCL generates OpenCL or CUDA kernels for each vector expression encountered during the execution [20], which may lead to many invocations of the just-in-time compiler during the execution step. These overheads are reduced by a separate kernel cache, which stores the resulting binary on the file system after the first just-in-time compilation and then reloads the binary in subsequent runs. While the NVIDIA GPU driver offers such a kernel buffer mechanism out of the box, such a feature is missing for many other OpenCL implementations.

3. Requirements on linear algebra software libraries. Before discussing ViennaCL in greater depth, we discuss generic requirements of computational scientists on a linear algebra library in this section. The computational scientist may be either a developer using the API of the library, or a user of a larger application based on the respective linear algebra library. Since the importance of the individual aspects discussed in the following subsections depends on the particular use case, we discuss these aspects in lexicographical order.

3.1. Composability. The mathematical description of linear algebra algorithms makes a minimum of assumptions about the objects involved. Therefore, a software library should not impose additional restrictions, so that one can choose from the product space of all available options. For example, a library that provides M different storage formats for matrices, K different Krylov solvers, and P different preconditioners should support all $M \times K \times P$ combinations. Going one step further, a preconditioner for a 2×2-block matrix

$$\begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{pmatrix}$$

may be based on the Schur complement $\mathbf{A}-\mathbf{B}\mathbf{C}^{-1}\mathbf{D}$, which is commonly implemented using a solver involving $\mathbf{C}$. A library should also offer the full set of solver options for the inner block matrix $\mathbf{C}$, allowing for arbitrarily nested solvers.

[12] http://icl.cs.utk.edu/magma/
[13] http://www.paralution.com/
[14] https://github.com/ddemidov/vexcl/

3.2. Extensibility. In order to minimize the amount of code changes necessary in a linear algebra library, it needs to be extensible by anticipating growth in the future. For example, it should be possible to use a specialized routine, which may only be available in binary form from a third party, for computing matrix-vector products in a Krylov solver. A common implementation paradigm to achieve extensibility is the use of a plugin architecture, through which additional functionality is registered at runtime.

3.3. Portability. Supercomputers are in operation for about four to five years, while codes tend to have a lifetime spanning decades. Therefore, a linear algebra library should not be written for a particular machine, but instead aim to support a broad range of machines. Different targets may run on different operating systems, provide different software toolchains, or provide vastly different computational power.

Also, an application interfacing to the linear algebra library may be written in a different language. The linear algebra library should provide an API which makes it easy for any application to call the respective functionality.

3.4. Robustness. A linear algebra library is used to fulfill a set of tasks such as the solution of a system of linear equations or the computation of eigenvalues of a given matrix. If a task cannot be completed, the library must report the error with an appropriate error handling mechanism such that the enclosing application can select further steps. If a successful execution of a task is more important than the use of a particular algorithm, a hierarchy of fallback methods should be provided. For example, a library for the solution of a sparse system of equations may first use an iterative solver and in the case of failure use a different preconditioner or a direct solver.

Even though fallback routines are usually less efficient in a benchmark scenario, an automatic fallback mechanism can save a lot of computational time when compared to an error encountered at a very late state in the overall progress using a nonrobust algorithm. Also, the application developer may not possess the knowledge to select the best algorithm for a particular task, but only provide structural information about the problem.

3.5. Speed. Because the performance of linear algebra functionality within an application is indicative of total application performance, many computational scientists select the fastest linear algebra library they can find. Consequently, library performance in terms of floating point operations per second is a common target metric for comparisons with other packages.

Speed, however, manifests itself not only in terms of raw compute performance, but also in terms of the time it takes to set up the corresponding code. For a typical code-compile-debug development cycle, a good linear algebra library should assist the developer in all three stages: First, the time spent in the coding stage can be substantially reduced by a concise API, good documentation, tutorials, and responsiveness to user questions. Second, the time spent on the compilation stage consists of two contributions. On the one hand, there is a one-time effort of setting up the build environment, and on the other hand one has to account for the time spent on code recompilation after each change of the code. Ideally, both the time needed for setting up the build environment and the time needed for recompilations are negligible. Third, the time spent on the debugging stage can be reduced by integrated debugging features such as stack traces or array bound checks.

4. ViennaCL overview. Before going into the details of ViennaCL, this section will discuss our approach of fulfilling the requirements on linear algebra libraries from section 3. Where appropriate, we will discuss the relevant API functionality. A full description of the API is, however, part of the documentation and not the purpose of this work.

ViennaCL's aim is to provide portable performance for common parallel programming models in a way that the user can manipulate objects similar to existing packages for single-threaded execution. For example, a matrix-vector product, an inner product, and a vector update when using the C++ library uBLAS included in the Boost libraries is written as

```
1  matrix<double> A(42, 42);
2  vector<double> x(42);
3
4  /* Initialize A and x with data */
5
6  vector<double> y = prod(A, x);
7  double s = inner_prod(x, y);
8  y += s * x;
```

A denotes a dense matrix represented by the templated type matrix<> consisting of double precision floating point numbers. Similarly, x and y are vector objects consisting of double precision floating point numbers. prod() calls a matrix-vector product routine, while inner_prod() computes the inner product of two vectors. Some C++ linear algebra libraries even overload the multiplication operator for matrix-vector products. This, however, can cause some unexpected side effects when chaining products such as y = A ∗ B ∗ x, because the left-to-right operator evaluation would result in the costly computation of A∗B rather than in two computationally cheaper matrix-vector products. Certain expression template techniques have been proposed to overcome most of these issues [34], but also lead to a higher load on the compiler.

The translation of the above code snippet to lower-level CUDA or OpenCL code requires familiarity with the respective programming model. With ViennaCL the above code snippet remains unchanged when executed with either of the compute backends; the user merely has to include ViennaCL's header files and use the respective namespace instead of using the header files and namespaces from uBLAS. Therefore, ViennaCL uses expression template techniques at the API level similar to those used in conventional C++ linear algebra libraries. However, as will be discussed in section 5, the implementation details for providing such an API are distinctly different from conventional expression template implementations in order to be able to support not only pure host code, but also CUDA code and just-in-time compiled OpenCL code.

The functionality provided by ViennaCL 1.7.1 is listed in Table 1. An outlook of future functionality and changes to ViennaCL can be found in section 7. In the following we discuss how ViennaCL addresses the generic library requirements from section 3.

4.1. Composability. ViennaCL extensively uses C++ generic programming techniques for composability. The dispatches occur at compile time, which on the one hand reduce the ability to compose functionality at runtime, but on the other hand allow for additional optimizations by the compiler. Operations involving dense matrices and vectors may not only be called for the full objects, but may also be called for submatrices and subvectors, enabling almost unlimited nesting.

TABLE 1
Overview of features provided by ViennaCL 1.7.1.

BLAS levels	1, 2, 3
Dense solvers	LU factorization
Sparse matrix formats	Compressed sparse row (CSR), coordinate format (COO), ELLPACK (ELL), hybrid (HYB) [10], sliced ELLPACK [35]
Sparse solvers	CG, BiCGStab, GMRES (classic and pipelined)
Preconditioners	Incomplete Cholesky, incomplete LU (ILU0, ILU with threshold, block-ILU), algebraic multigrid, sparse approximate enverses, Jacobi
Eigenvalue routines	QR method, Lanczos, bisection, power iteration
Bandwidth reduction	Cuthill–McKee, Gibbs–Poole–Stockmeyer
Structured matrices	Circulant, Hankel, Töplitz, Vandermonde
Miscellaneous	Fast Fourier transform (Radix-2, Bluestein), singular value decomposition, QR factorization

4.2. Extensibility. The iterative solvers in ViennaCL cannot only be called for objects with a ViennaCL class type, but also for types from other libraries. Support for classes from Armadillo[15] [48], Eigen[16] [29], MTL4[17] [26], and uBLAS are provided with ViennaCL and serve as a template for extending the available functionality to other libraries.

4.3. Portability. ViennaCL is written according to the 2003 standard of C++. This ensures broad support even on machines with operating systems released many years ago.

A unique feature of ViennaCL is that it provides support for CUDA, OpenCL, and OpenMP with the ability to switch between any of the three compute backends at runtime. For the cases where either CUDA or OpenCL is not available at the target system, the respective backend can also be statically disabled in the build system. If OpenMP is not available or disabled, the host-based backend falls back to a single-threaded execution.

We will discuss aspects of portable performance below.

4.4. Robustness. ViennaCL uses C++ exceptions to report errors to higher entities. The algorithms used for running particular tasks such as computing eigenvectors are selected by the surrounding application. An automatic switch to a different algorithm in the case of failure of the selected algorithm is not provided.

4.5. Speed. The OpenCL backend is particularly attractive for a broad set of hardware from different vendors. However, the availability of a portable programming model does not automatically imply portable performance. To also ensure portable performance, ViennaCL contains a device database, which allows for fine tuning the respective compute kernels to the target devices at runtime. Performance higher or at least comparable to has already been presented for dense linear algebra in former work [54]. Section 6 presents additional results for sparse matrix-vector and sparse matrix-matrix multiplications as well as solver performance for preconditioners.

5. Library architecture. We will discuss the current architecture of ViennaCL 1.7.1 in the following subsections. The starting point is the code line `u = x + y`

[15] http://arma.sourceforge.net/
[16] http://eigen.tuxfamily.org
[17] http://mtl4.org/

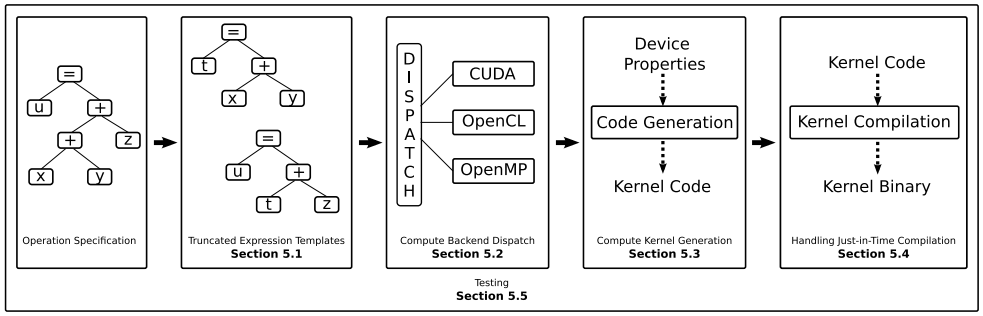

FIG. 1. *Overview of the individual steps required in the ViennaCL backend for executing the vector operation* $u = x + y + z$.

TABLE 2

The predefined set of operator-overloaded operations in ViennaCL. A, B, and C are either all vectors or matrices, while α *and* β *denote scalars.*

$A = B;$	$A \mathrel{+}= B;$	$A \mathrel{-}= B;$
$A = \pm\alpha B;$	$A \mathrel{+}= \pm\alpha B;$	$A \mathrel{-}= \pm\alpha B;$
$A = \pm\ B \pm\ C;$	$A \mathrel{+}= \pm\ B \pm\ C;$	$A \mathrel{-}= \pm\ B \pm\ C;$
$A = \pm\alpha B \pm\ C;$	$A \mathrel{+}= \pm\alpha B \pm\ C;$	$A \mathrel{-}= \pm\alpha B \pm\ C;$
$A = \pm\alpha B \pm \beta C;$	$A \mathrel{+}= \pm\alpha B \pm \beta C;$	$A \mathrel{-}= \pm\alpha B \pm \beta C;$
$A = \pm\ B \pm \beta C;$	$A \mathrel{+}= \pm\ B \pm \beta C;$	$A \mathrel{-}= \pm\ B \pm \beta C;$

+ z for vectors u, x, y, and z. We start with a discussion of how the expression template encoded operation is mapped to a set of predefined operations and how these operations dispatch into the respective compute backend. For the OpenCL backend we additionally discuss how we manage device-specific kernels and the implications of OpenCL just-in-time compilation on the OpenCL kernel management. A schematic overview of these steps is given in Figure 1, where the enclosing testing methodology is outlined in the last subsection.

5.1. Truncated expression templates. Expression templates are an established C++ technique for providing operator overloads without the performance hit of expensive intermediate temporary objects [57, 59]. For example, a naive operator overload applied to the vector operation $u = x + y + z$ results in temporary objects for $x + y$ and for the addition of z from the first temporary object. By replacing expensive temporary vector objects with lightweight objects encoding the operation via templated types, the compiler is able to optimize away any temporary objects, resulting in an implementation that is equivalent to an efficient C-like for-loop-based implementation. However, a direct application of expression templates to OpenCL is not attractive because of the just-in-time compilation overhead for more complex algorithms.

ViennaCL internally uses a technique we refer to as *truncated expression templates*. The expression is encoded via conventional expression templates, but the execution is mapped to a predefined set of operations. Table 2 lists the set of predefined operations for vectors and matrices, which resembles the set of operations at the basic linear algebra subprograms (BLAS) level 1. The scalars α and β may either be an integer or floating point type, or a scalar<T> object representing a single scalar on the compute device. Longer expressions are decomposed into these predefined operations. Reconsidering the vector expression $u = x + y + z$, it gets decomposed into

the operations $t = x + y$ for a temporary vector t, followed by $u = t + z$. Even though truncated expression templates do not eliminate all temporary objects in complex expressions, they have certain advantages: First, temporary objects are completely removed for the most common operations. Complex expressions result in less temporaries than naive operator overloading, so the overhead due to temporaries may still be lower than the overhead for an additional just-in-time compilation. Second, a predefined set of operations enforces a separation of the API from the computational backends. In other words, computational backends are fully decoupled from the use of expression templates for the API.

5.2. Multibackend. Since CUDA memory, OpenCL memory, and main memory each have distinct representations of memory buffers, memory buffers are represented by a mem_handle class in an abstract way. A mem_handle object may hold either a raw CUDA pointer, an OpenCL memory handle (of type cl_mem), or a host pointer. Since the memory handle in OpenCL is not type safe, mem_handle was also designed to be type agnostic of the underlying memory. The type-agnostic mem_handle was selected over a type-safe mem_handle<T> in order to avoid repeated instantiation of memory management functionality for different data types. These memory management functionalities include read and write operations from main memory, as well as raw buffer copies within each compute backend. Moreover, routines for the migration of a memory buffer from one type of memory (e.g., CUDA) to another (e.g., OpenCL) are provided. Extra care had to be taken with respect to binary data representations: Certain integer types may have a different binary representation for conventional host types such as **int** than on the OpenCL device, so appropriate conversion routines are necessary.

With the abstraction of memory handles via mem_handle and the associated routines for reading, writing, and copying memory handles, all other data containers are implemented in a backend-agnostic way using mem_handle. The computational backend consists of a dispatcher interface, which calls the respective operations from the correct compute backend. Let us consider the matrix-vector product

```
y = prod(A, x);
```

of a matrix A and vectors x and y. With the use of expression templates, a temporary vector is avoided and the computation is deferred to a compute backend dispatcher routine prod_impl(A, x, y). The compute dispatcher routine inspects the operands and calls the operations from the respective compute backend:

```
1  void prod_impl(matrix<T> const & A,
2                 vector<T> const & x, vector<T> & y)
3  {
4    switch (A.handle().get_active_handle_id()) {
5    case   MAIN_MEMORY: host_based::prod_impl(A, x, y); break;
6    case OPENCL_MEMORY:      opencl::prod_impl(A, x, y); break;
7    case   CUDA_MEMORY:        cuda::prod_impl(A, x, y); break;
8    default: /* error handling */
9    }}
```

A.handle() returns the memory object of type mem_handle, from which an identifier for the currently active memory domain is requested. Then, the implementation in the associated compute backend is called, where additional checks of the input arguments are performed.

The implementations in the three compute backends are distinct from each other. Although this results in the same operations being implemented multiple times, it allows for maximum flexibility at only moderately increased development time. The following workflow has been found to be most efficient: First, any new operation is implemented for the host (OpenMP) backend under the assumption of a high number of threads. This requires us to use algorithms similar to the ones implemented for the CUDA and OpenCL compute backends later. At the same time, the host backend can be easily debugged and tested with the full set of debugging tools available. Once the implementation for the host backend is completed, the new functionality is implemented in the CUDA backend. This is accomplished by copying and altering the code for the host backend to consider CUDA specifics. Significant differences only arise if reduction-like operations in on-chip shared memory are used. The CUDA implementation is then tested with the already existing, backend-agnostic test suite initially implemented for the host backend. Because a reference implementation is available on the host and because the CUDA toolchain provides a rich set of debugging tools, bugs can be located and fixed quickly. Finally, the OpenCL implementation is derived from the CUDA implementation by a substitution of keywords.

5.3. Device database. OpenCL allows for a device-specific kernel to be just-in-time compiled on the target machine with all optimizations for the target device enabled. However, there is no automatic performance portability: The optimization of a kernel for a particular target device does not imply good performance on a different device. Even though there is good correlation of performance across devices of the same type [44], target-specific characteristics of the device need to be taken into account for best performance. To generate such optimized kernels, a database holding the best kernel parameters for target devices is integrated into ViennaCL.

Kernel parameters are used to transform abstract kernel code templates into the final kernel code [54]. These kernel code templates must be general enough in order to be able to generate efficient kernel code, but at the same time be specialized enough to remain manageable. ViennaCL provides the following kernel code templates for manipulating full vectors and matrices as well as subvectors and submatrices described by index offsets and nonunit strides:

- *Vector operations without reductions*: This template covers element-by-element vector operations. Examples are vector additions, vector subtractions, and elementwise operations such as element-by-element multiplications.
- *Vector operations with reductions*: This template describes element-by-element vector operations followed by a reduction operation over the whole vector. Typical examples are inner products and vector norms.
- *Matrix operations without reductions*: This template extends vector operations without reductions to matrices.
- *Matrix operations with row- or column-wise reduction*: This template is used for kernels which involve a reduction step within rows or columns. Examples are matrix-vector products or the computation of row norms.
- *Matrix-matrix products*: Kernels for general dense matrix-matrix products are generated with this template.

The database is populated by dedicated autotuning runs on as many target devices as possible. We currently follow a static approach where the device database is integrated into the source code. This is the most robust approach, since it avoids possible errors when otherwise interacting with the system environment by, e.g., loading device parameters from the filesystem.

If a device is not yet available in the device database, the procedure for selecting parameters is as follows:

- If there is an explicit mapping of a device to another device available in the database, the parameters from the mapped device are used. This is common for devices within the same product generation, where products differ only by minor details such as clock frequency, but otherwise use the same hardware.
- If no explicit mapping for a device is available, default parameters for devices from the respective vendor are used.
- If no default parameters for devices of the vendor are available, a default for the device type (either CPU, GPU, or accelerator) is used.

The parameters in the device database are also useful for the CUDA compute backend, since it includes the best parameters found for the local and global workgroup sizes. Because architectural differences for GPUs from NVIDIA are smaller than across vendors, we found that only setting proper workgroup sizes at runtime is enough to obtain good performance for memory-bandwidth-limited kernels on NVIDIA GPUs [44].

An earlier study of the impact of different kernel parameters on device performance has shown that there is a strong correlation of best kernel parameters among different operations [44]. Thus, a good parameter set for vector operations (cf. the STREAM benchmark[18] [38]) is also a good choice for operations for which no full template is available. For example, parallel prefix sums benefit directly from good parameters for vector operations with reductions. This correlation enables us to provide good default values even if the respective target device is not available in the database, and helps to speed up the autotuning process used for populating the database [54].

5.4. OpenCL just-in-time compilation. One possible approach for OpenCL just-in-time kernel compilation, which is taken by VexCL, is to generate an OpenCL compute kernel each time a vector expression is encountered. To avoid unnecessary and time-consuming recompilation in each run, the resulting binary may be cached on the filesystem. Subsequent runs may thus skip the recompilation step and only load the respective binary from the file system. Such a caching option, however, is not available on systems with a read-only filesystem. Similarly, certain network filesystems may provide too high a latency to cache kernel binaries there. ViennaCL per default does not cache kernels to avoid additional sources of errors. However, caching can be enabled for optimization purposes by setting the cache path through an environment variable.

Figure 2 compares the time spent on the just-in-time compilation of 64 simple OpenCL kernels for different OpenCL vendor implementations, where the kernels are distributed over a varying number of OpenCL programs. Each OpenCL program represents a compilation unit containing one or several kernels, hence the OpenCL just-in-time compiler is invoked as many times as there are OpenCL programs. A linear increase in the overhead is observed for more than ten OpenCL programs, indicating that the total overhead is dominated by the number of invocations of the just-in-time compiler. Therefore, OpenCL kernels in ViennaCL are grouped into a moderate number of OpenCL programs based on functionality: For example, vector operations at BLAS level 1 are grouped into one OpenCL program, matrix-vector operations defined at BLAS level 2 into another OpenCL program, and so on. Typical functionality accessed by a user can thus be provided with about two to five OpenCL program compilations.

[18] https://www.cs.virginia.edu/stream/

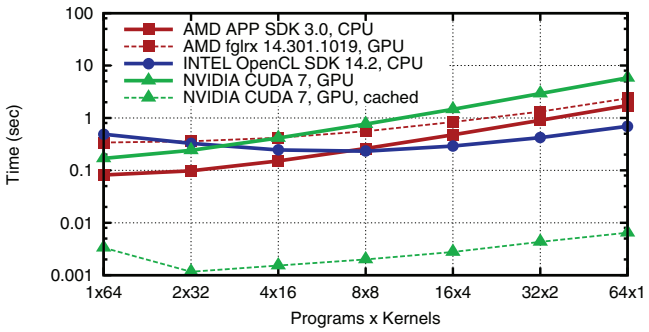

FIG. 2. *Evaluation of the just-in-time compilation overhead of different OpenCL vendor implementations for compiling* 64 *OpenCL kernels split over one to* 64 *OpenCL programs. Splitting the* 64 *OpenCL kernels into two or four OpenCL programs results in shortest compilation times overall. One OpenCL program per kernel consistently results in highest overheads. A caching mechanism reduces the overhead by two orders of magnitude.*

5.5. Testing. Continuous integration as well as nightly tests are considered best practices [33] and used in the development of ViennaCL. The development of a GPU-accelerated library, however, needs to consider one important difference to the development of purely CPU-based libraries: GPUs are not fully virtualized (yet). This implies that it is not sufficient to use a single physical machine with many virtual machines in which the different target operating systems are emulated. Instead, different physical test machines consisting of low- to high-end GPUs of different hardware generations are required to verify the correct operation of the library on a broad range of hardware. If we estimate three main operating systems (Windows, Linux, MacOS) and two GPU vendors with at least three different hardware architectures each, 18 different configurations are required for full test coverage. The combinatorial complexity explodes if one also takes different versions of operating systems, compilers, or drivers into account. Even after considerable effort, ViennaCL's set of nightly test machines covers only about half of the possible configurations outlined above. However, the focus on "extreme" configurations with old compilers and low-end hardware[19] helped a lot in improving the robustness of the library without sacrificing good performance on high-end hardware.

6. Benchmarks. We will demonstrate the portable high performance of ViennaCL by the examples of sparse matrix-vector multiplications (SpMVs), sparse matrix-matrix multiplications, pipelined iterative solvers with kernel fusion, algebraic multigrid preconditioners, and parallel incomplete LU (ILU) factorization preconditioners. For results underlining the portable high performance of ViennaCL for dense linear algebra operations including dense matrix-matrix multiplications we refer to earlier publications [44, 54]. The set of sparse matrices used for the performance evaluations is listed in Table 3 and is similar to the sets of matrices used for assessing performance in earlier publications [27, 28, 35]. All tests are run on Linux-based machines equipped with hardware from AMD, INTEL, and NVIDIA (using CUDA SDK 7) as listed in Table 4. Within a sparse matrix-vector multiplication benchmark we also compare with two additional NVIDIA GPUs (GeForce GTX 470, GeForce GTX 750 Ti) to evaluate performance portability. Since the four hardware platforms are

[19] One of our most useful test machines is equipped with an NVIDIA GeForce 8600 GT (released in 2007) running CentOS 5.11, where the default compiler is GCC 4.1.

TABLE 3

Properties of the square sparse matrices from the Florida Sparse Matrix Collection [19] *used for performance comparisons. The grouping into somewhat more regular (upper* 9*) and more irregular (lower* 11*) matrices is taken from the work by Gremse et al.* [28].

Name	Dimension	Nonzeros (NNZ)	Maximum NNZ/row	Average NNZ/row
Cantilever	62 451	4 007 383	78	64.2
Economics	206 500	1 273 389	44	6.2
Epidemiology	525 825	2 100 225	4	4.0
Harbor	46 835	2 374 001	145	50.7
Protein	36 417	4 344 765	204	119.3
qcd	49 152	1 916 928	39	39.0
Ship	140 874	7 813 404	102	55.5
Spheres	83 334	6 010 480	81	72.1
Windtunnel	217 918	11 634 424	180	53.4
Accelerator	121 192	2 624 331	81	21.7
Amazon0312	400 727	3 200 440	10	8.0
ca-CondMat	23 133	186 936	280	8.1
cit-Patents	3 774 768	16 518 948	770	4.4
circuit	170 998	958 936	353	5.6
email-Enron	36 692	367 662	1 383	10.0
p2p-Gnutella31	62 586	147 892	78	2.4
roadNet-CA	1 971 281	5 533 214	12	2.8
webbase1m	1 000 005	3 105 536	4 700	3.1
web-Google	916 428	5 105 039	456	5.6
wiki-Vote	8 297	103 689	893	12.5

TABLE 4

Overview of hardware used for the comparisons. All values are theoretical peaks. Practical peaks for memory bandwidth and floating point operations per seconds are around 70 *to* 80 *percent of the theoretical peaks* [44], *except for the Xeon Phi, where only* 50 *percent of the theoretical peak bandwidth can be obtained* [49].

	FirePro W9100	Dual Xeon E5-2670 v3	Xeon Phi 7120P	Tesla K20m
Vendor	AMD	INTEL	INTEL	NVIDIA
Memory bandwidth (GB/sec)	320	136	352	208
GFLOP/sec (float)	5238	1766	2416	4106
GFLOP/sec (double)	2619	884	1208	1173
TDP (watt)	275	240	300	244

comparable in their thermal design power (TDP), our benchmark results allow for a fair comparison also in terms of performance per watt.

6.1. SpMV. SpMVs are the essential building blocks of iterative solvers from the family of Krylov methods. Also, several preconditioners such as algebraic multigrid preconditioners or polynomial preconditioners rely on the availability of fast SpMVs. Consequently, many publications have focused on optimizing SpMVs to exploit fine-grained parallelism available in modern hardware by not only optimizing SpMVs for conventional sparse matrix formats such as the CSR format, but also by proposing new formats [5, 8, 10, 27, 35, 36, 49, 53, 60]. ViennaCL provides optimized SpMV implementations for several different sparse matrix formats, but recent improvements for SpMVs in the common CSR format [27] reduced resiliance on specialized sparse matrix formats: On the one hand, other operations such as sparse matrix-matrix

multiplications may be very inefficient for a sparse matrix format exclusively tailored to SpMVs, for example, if individual rows can no longer be accessed quickly. On the other hand, many applications already provide their data in CSR format, so any other sparse matrix format requires additional conversion overhead. Consequently, the higher flexibility and the widespread use of the CSR format may outweigh the, nowadays, only small performance gains of SpMVs using specialized sparse matrix formats.

ViennaCL implements the CSR-adaptive SpMV algorithm initially proposed for AMD GPUs [27]. Our evaluations have shown that CSR adaptive is also very efficient for very sparse matrices (less than six nonzeros per row on average) on NVIDIA GPUs. If the average number of nonzeros per row exceeds six, we assign 8, 16, or 32 threads per row on NVIDIA GPUs. We always use CSR adaptive on AMD GPUs, as it has been demonstrated to be the most efficient SpMV implementation compared to other SpMV implementations for CSR as well as specialized sparse matrix formats [27].

A performance comparison of ViennaCL's SpMV routine with SpMV routines in CUSP and CUSPARSE on an NVIDIA Geforce GTX 750 Ti GPU (Maxwell architecture), an NVIDIA Tesla K20m (Kepler architecture), and an NVIDIA GeForce GTX 470 (Fermi architecture) is given in Figure 3. These devices represent the three most recent architectures from NVIDIA in order to also assess performance across different architectures. ViennaCL outperforms CUSPARSE on the GTX 750 Ti by 36 percent and on the Tesla K20m by 24 percent in terms of geometric averages of giga floating point operations per second (GFLOP/sec). On the GTX 470 the performance of ViennaCL is 13 percent lower than CUSPARSE. Better performance of ViennaCL over CUSP of 31 percent, 32 percent, and 31 percent is observed for the three GPUs, respectively. Overall, ViennaCL provides higher overall performance than CUSPARSE and CUSP by reusing performance optimizations initially proposed for AMD GPUs.

6.2. Sparse matrix-matrix multiplication. Sparse matrix-matrix multiplications are important for computing coarse grid operators in algebraic multigrid methods [56], but are also the basis for graph processing operations such as subgraph extraction [14], breadth-first search [25], transitive closure [40], or graph clustering [58]. In contrast to SpMVs, which are usually computed many times for a given matrix, sparse matrix-matrix multiplications are often only computed once for a given pair of factors. Also, the performance in terms of GFLOP/sec is lower than for SpMVs because of the additional level of indirection in memory accesses required to compute the result.

We have recently extended the RMerge algorithm proposed by Gremse *et al.* [28]. RMerge is based on computing each of the result rows $\mathbf{c}_i$ of $\mathbf{C} = \mathbf{AB}$ by simultaneously merging multiple rows of $\mathbf{B}$ according to

$$\mathbf{c}_i = \sum_{j:a_{ij}\neq 0} a_{ij}\mathbf{b}_j,$$

where a_{ij} denote the entries of $\mathbf{A}$ and $\mathbf{b}_j$ denote the jth row of $\mathbf{B}$. We extend the idea of merging multiple rows to AMD GPUs by generalizing the idea of subwarps on NVIDIA GPUs to subwavefronts on AMD GPUs, and to Haswell CPUs as well as Xeon Phi by merging multiple rows using advanced vector extension intrinsics [42].

The performance comparison in Figure 4 for matrix squaring ($\mathbf{C} = \mathbf{AA}$) shows that ViennaCL outperforms CUSPARSE and CUSP by a factor of 2.5 and a factor of 2.0, respectively. A 30 percent performance gain over MKL is obtained on geometric average on the CPU. The situation is reversed on the Xeon Phi, where ViennaCL is on

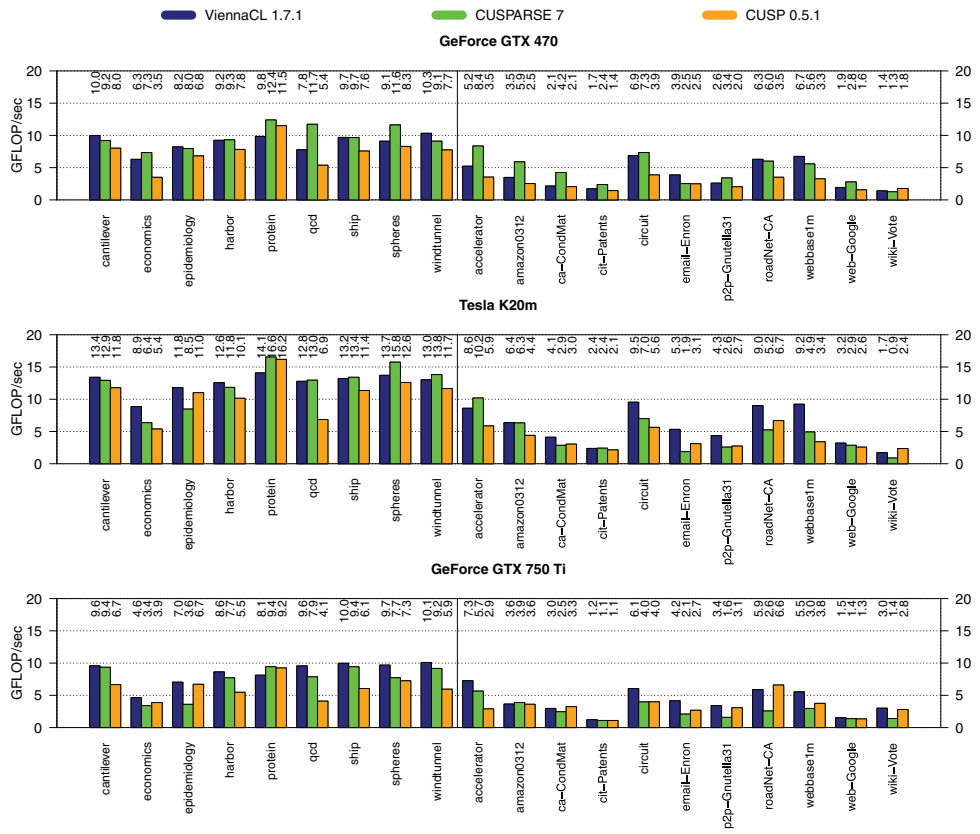

FIG. 3. *Performance comparison of SpMV routines in ViennaCL* 1.7.1, *CUSPARSE* 7, *and CUSP* 0.5.1 *on an NVIDIA Geforce GTX* 470 *(Fermi architecture), an NVIDIA Tesla K*20 *(Kepler architecture), and an NVIDIA GTX* 750 *Ti (Maxwell architecture). The geometric averages over all test runs on all three devices are* 5.9 *GFLOP/sec for ViennaCL,* 5.2 *GFLOP/sec for CUSPARSE, and* 4.5 *GFLOP/sec for CUSP.*

average 65 percent slower, which can be explained by a lower degree of optimization for the Xeon Phi in ViennaCL 1.7.1. We had no other implementation available for a comparison on AMD GPUs when running the benchmarks. A comparison of the geometric averages in terms of GFLOP/sec of ViennaCL's implementation on the NVIDIA Tesla K20m with ViennaCL's implementation on the AMD FirePro W9100 shows a 53 percent higher performance on the AMD GPU. While the results for SpMVs in the previous section showed a better performance on NVIDIA GPUs when leveraging optimizations for AMD GPUs, our results for sparse matrix-matrix multiplications in this section extend optimizations for NVIDIA GPUs to AMD GPUs as well as CPUs.

Overall, our benchmark results suggest that the large caches of CPUs result in significantly better sparse matrix-matrix multiplication performance of CPUs as compared to GPUs, because repeatedly accessed rows in $\mathbf{B}$ may already be available in cache. This, however, also implies that the sparsity pattern of the matrix $\mathbf{A}$ is especially important for cache hit rates. The throughput-oriented architectures of GPUs and Xeon Phis have much smaller caches, resulting in many inefficient reloads of data from global memory.

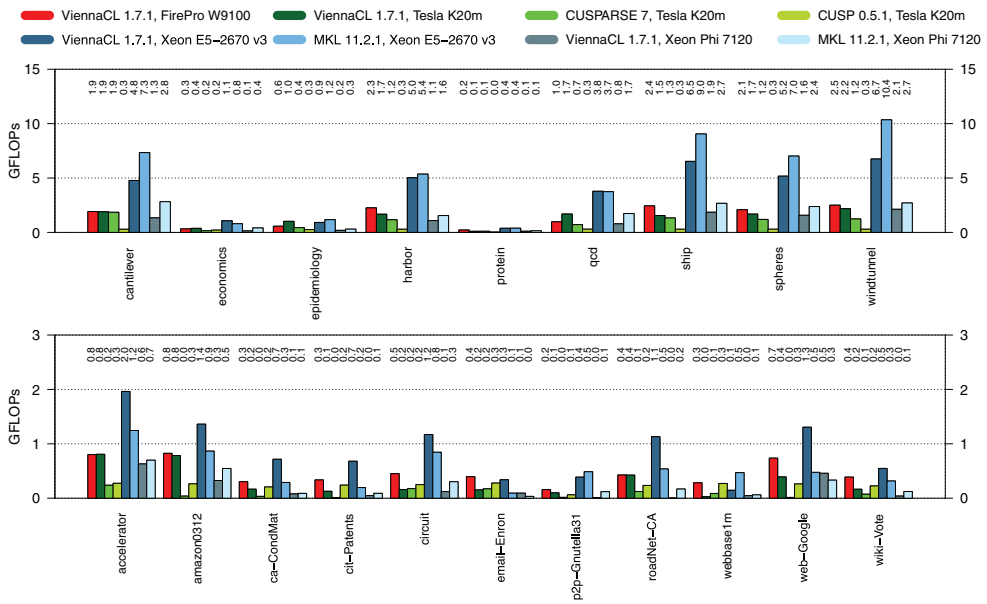

FIG. 4. *Performance comparison of sparse matrix-matrix multiplication routines in ViennaCL* 1.7.1*, CUSPARSE* 7*, CUSP* 0.5.1*, and MKL* 11.2.1 *for matrix squaring on the hardware listed in Table* 4*. Best overall performance is obtained on the CPU-based system, with ViennaCL outperforming MKL by* 30 *percent on geometric average. ViennaCL outperforms CUSPARSE by a factor of* 2.5 *and CUSP by a factor of* 2.0 *in geometric mean.*

6.3. Pipelined iterative solvers with kernel fusion. Iterative solvers from the family of Krylov methods are the methods of choice whenever sparse direct solvers are not appropriate. Many applications require preconditioners for fast convergence, but a fine-grained acceleration of efficient preconditioners is difficult. In contrast, iterative methods with no preconditioner or diagonal preconditioners provide the fine-grained parallelism required to run efficiently on GPUs. The only global synchronization points are sparse matrix-vector products and inner products, from which estimates for the residual reduction are obtained. However, global synchronization points translate into new kernel launches, which have a certain latency due to the communication of the host with the device through PCI-Express in the case of discrete GPUs. Such latencies are also observed for parallel runs on CPUs or Xeon Phis for process or thread synchronization, but these latencies are smaller and can be avoided by switching to a single-threaded execution.

Pipelined iterative solvers with kernel fusion use recurrence relations to rewrite the original algorithm. In the context of this work, the term *pipelined* refers to a reduction of synchronization points and a reduction of load and store operations in global memory by reusing intermediate results. The term is also used for overlapping computation with communication (e.g., [23, 24]), which, however, is not the case here. *Kernel fusion*, on the other hand, refers to merging the operations in subsequent kernels into a single kernel. The combination of pipelining and kernel fusion allows for a reduction of the number of synchronizations per solver iteration. As a result, better performance is obtained particularly for small systems, where synchronization overheads become important.

We have recently developed and implemented a pipelined conjugate gradient (CG) method with kernel fusion, a pipelined stabilized biconjugate gradient (BiCGStab)

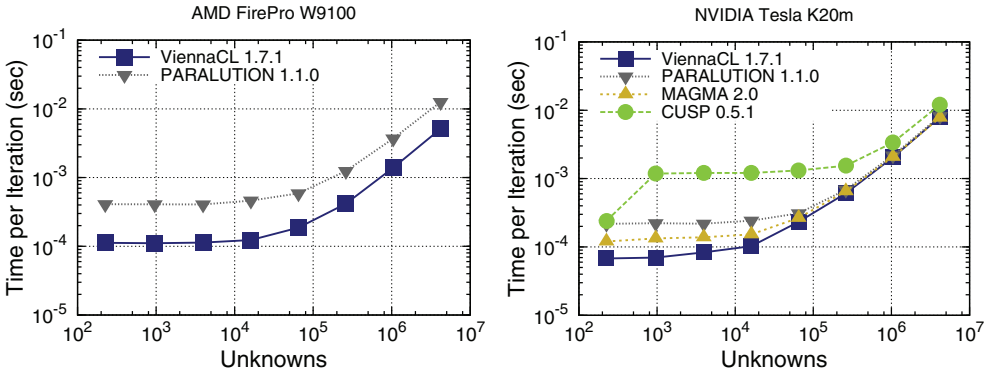

FIG. 5. *Comparison of execution times per solver iteration for the CG algorithm applied to a first-order finite element method on unstructured triangular meshes for the Poisson equation on the unit square on an AMD FirePro W9100 (left) and an NVIDIA Tesla K20m (right). The pipelined implementation in ViennaCL (and MAGMA for the Tesla K20m) provides up to three-fold better performance for small system sizes due to a reduced number of kernel launches and synchronization points than the classical implementations in PARALUTION and CUSP. Nevertheless, execution times below* 10^4 *unknowns are limited by kernel launch overheads on either GPU, because latency effects due to PCI-Express communications can no longer be hidden by computations.*

method with kernel fusion, and a pipelined general minimum residual method with kernel fusion, each tailored to GPUs [45]. Similar ideas have been used by other authors for reduced energy consumption [3, 2] or improved performance [4]. Results obtained for a linear finite element discretization of the Poisson equation on triangular meshes of the unit square in two spatial dimensions for the CG method are shown in Figure 5. Our pipelined implementation with kernel fusion reduces the number of kernel launches per solver iteration from six for the classical CG method to only two, thus providing a threefold performance gain for small systems when compared to classical implementations (PARALUTION, CUSP). In other words, pipelined implementations enable GPU-based solvers to scale to three-times smaller systems, which is particularly important if many small systems such as in time-stepping schemes need to be solved instead of only a single big system. Figure 6 depicts performance results for the sparse matrices in Table 3, investigating the performance for larger systems where kernel launch overhead is negligible. Overall, these results show that ViennaCL's implementation based on fused kernels is on a par with MAGMA also for large matrices, and outperforms the implementations based on dedicated SpMV kernels in PARALUTION and CUSP.

6.4. Algebraic multigrid preconditioners. Algebraic multigrid preconditioners are attractive because they are entirely based on entries in the system matrix and thus can be used in a black-box manner [56]. The algebraic multigrid preconditioners in ViennaCL 1.7.1 are based on earlier work by other authors on exposing fine-grained parallelism in algebraic multigrid methods for NVIDIA GPUs [9, 22]. We extended these techniques to support AMD GPUs, CPUs, and Xeon Phis [46]. In this work we will only summarize the key results for an aggregation-based and a smoothed aggregation algebraic multigrid method.

We consider linear finite element discretizations of the Poisson equation $\Delta u = 1$ with homogeneous Dirichlet boundary conditions at the x and y boundaries on unstructured simplex meshes of the unit square and unit cube, respectively. Since the resulting system matrices are symmetric positive definite, we use the CG method for

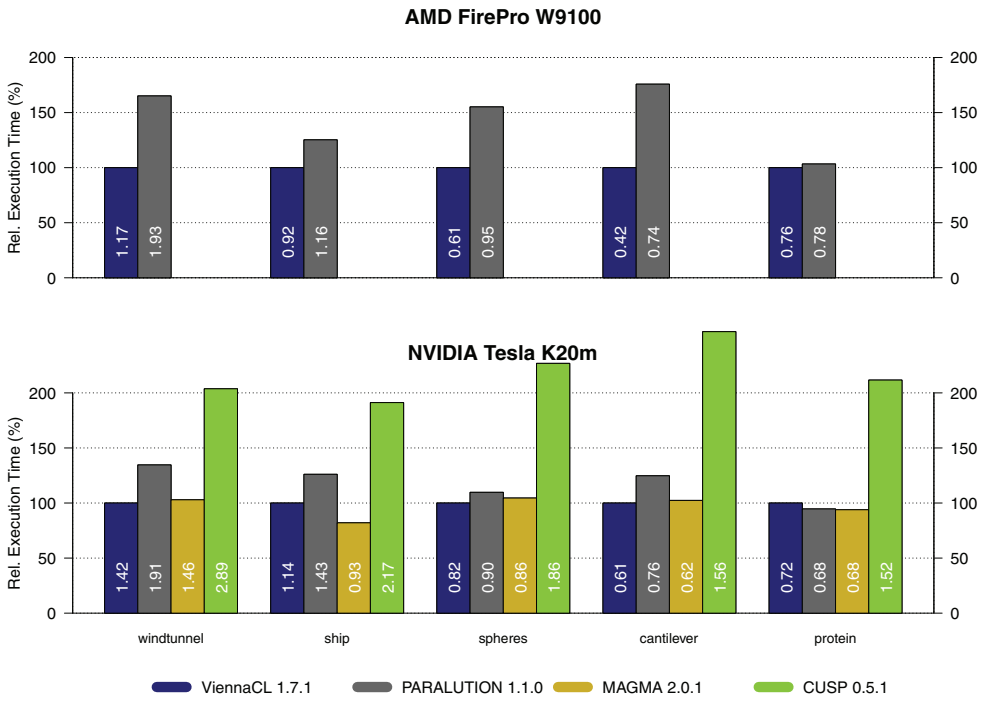

FIG. 6. *Comparison of execution times in milliseconds per CG solver iteration for symmetric positive definite matrices from Table 3. Even though kernel launch overheads are negligible here, the reduced data transfers of the pipelined implementation in ViennaCL result in better overall performance than for other libraries.*

the Krylov solver. For better comparison with other available solver alternatives we also include the pipelined CG solver from section 6.3 as well as classical AMG implementations using single-threaded one-pass Ruge–Stüben coarsening supplemented with direct interpolation [56]. The number of pre- and post-smoothing steps was chosen such that the smallest time to solution was obtained: One pre- and one post-smoothing step at each level in the multigrid hierarchy are applied for the classical AMG implementation, whereas the aggregation AMG preconditioner as well as the smoothed aggregation AMG preconditioner use two pre- and post-smoothing steps. The coarsest unstructured meshes were generated using Netgen[20] [51] and then subsequently refined uniformly to obtain higher resolution. Note that this simple setting ensures good convergence of the unpreconditioned solvers relative to solvers with AMG preconditioners. Thus, AMG preconditioners are likely to perform better relative to unpreconditioned solvers in practical situations involving, e.g., inhomogeneous or anisotropic diffusion.

A comparison of solver iterations required to reach a relative reduction of the initial residual by a factor 10^{-8} and the number of levels in the AMG hierarchy is given in Figure 7. Aggregation-based coarsening is more aggressive than classical coarsening and thus results in a smaller number of levels, but is not asymptotically optimal. Smoothed aggregation coarsening is even more aggressive and requires only four levels, but nevertheless shows better asymptotic behavior: The number of iterations grows

[20] http://sourceforge.net/projects/netgen-mesher/

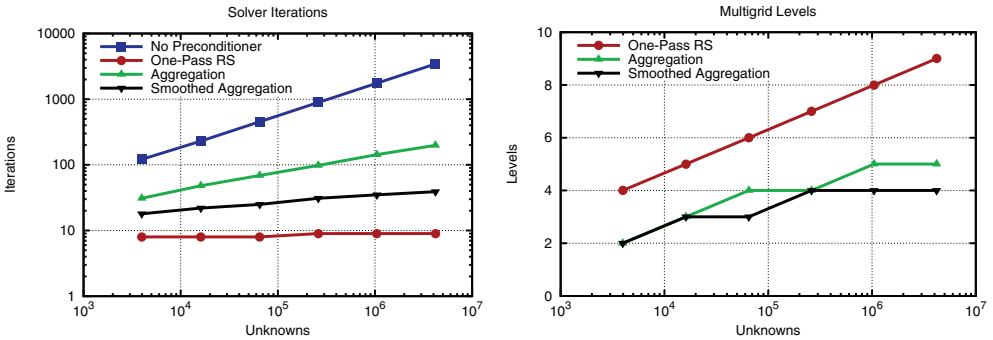

FIG. 7. *Comparison of the number of solver iterations (left) and levels in the AMG hierarchy (right) required for solving the two-dimensional Poisson equation when using a classical one-pass Ruge–Stüben AMG preconditioner (One-Pass RS), an aggregation-based AMG preconditioner, and a smoothed aggregation AMG preconditioner. Asymptotic optimality is only achieved with classical one-pass Ruge–Stüben AMG, at the expense of a higher number of levels.*

only by a factor of two for system sizes ranging over three decades. Looking at only the number of solver iterations required, the unpreconditioned solver is not an attractive choice at all.

Solver setup times, solver cycle times, and total solver execution times (i.e., the sum of the former two) for each of the four hardware platforms are depicted in Figure 8. The sequential one-pass Ruge–Stüben AMG preconditioner takes about one order of magnitude longer to set up than the two parallel AMG preconditioners. Kernel launch overheads on GPUs as well as OpenMP synchronization costs become visible for problem sizes below 10^5 unknowns. The highest overheads are observed for the AMD FirePro W9100, where problem sizes above 10^6 are needed for negligible overheads.

The solver cycle times in the center column of Figure 8 reflect the small number of iterations required for the classical one-pass Ruge–Stüben AMG preconditioner. Similarly, the smoothed aggregation AMG preconditioner outperforms the aggregation-based AMG preconditioner because of the smaller number of iterations needed. The better asymptotic behavior of the aggregation-based AMG preconditioner over the unpreconditioned pipelined CG solver only starts to show at very large problem sizes above 10^6 unknowns, but the gains are very mild.

Total solver execution times in the right column of Figure 8 show that the smoothed aggregation preconditioner is the best choice for problem sizes above 2×10^5 on all four hardware platforms. The unpreconditioned solver is a better choice for smaller problem sizes in the benchmark setting considered here. In contrast to the classical one-pass Ruge–Stüben AMG preconditioner, which spends most of its time in the setup stage, and in contrast to the aggregation-based AMG preconditioner, which spends most of its time in the solver cycle stage, the smoothed aggregation AMG preconditioner spends about the same amount of time in each of the two stages. This balance is visible on all four hardware platforms.

The results from Figure 8 are collected in Figure 9 for a better comparison of the individual hardware platforms. Overall, the performance differences are relatively small for the largest problem sizes: The dual INTEL Xeon E5-2670 v3 system is on a par with the AMD FirePro W9100, while the Tesla K20m is about 50 percent slower and the Xeon Phi is about a factor of two slower. The CPU-based execution is the

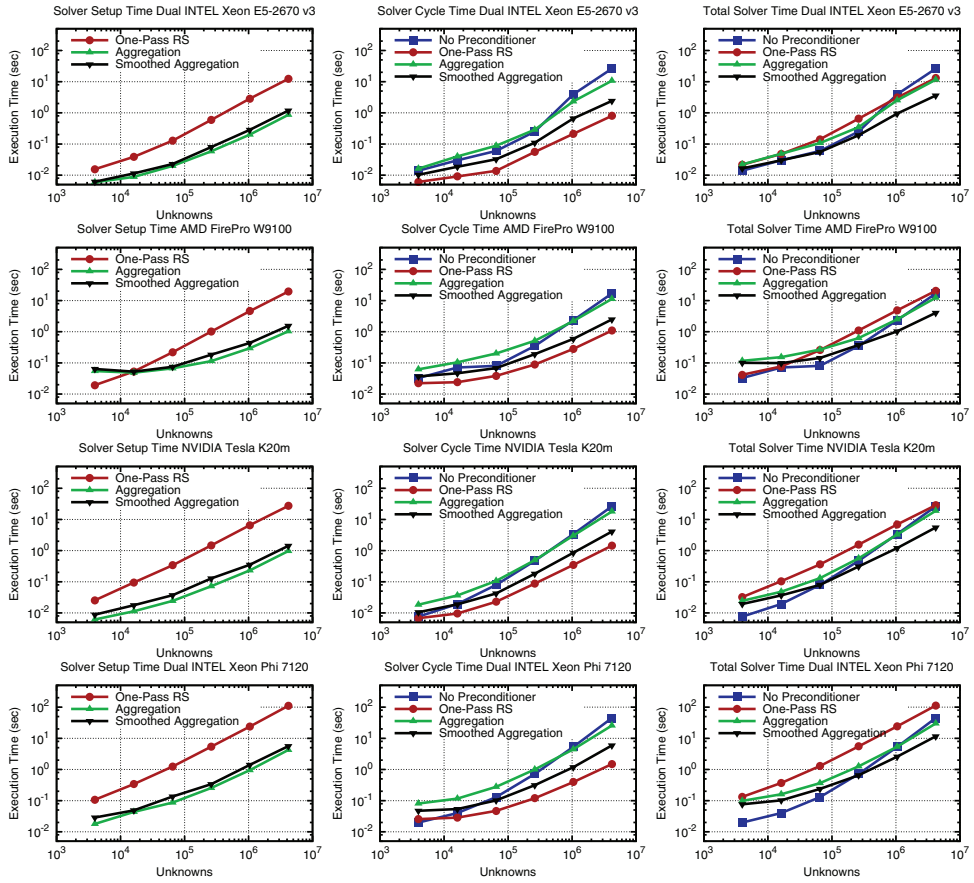

FIG. 8. *Comparison of solver setup times (left), solver cycle times (center), and total solver times (right) of classical one-pass Ruge–Stüeben coarsening (One-Pass RS), aggregation-based AMG, and smoothed aggregation AMG for the two-dimensional Poisson equation on each of the four hardware platforms considered. The setup for the sequential classical one-pass coarsening has been run on a single CPU core for the AMD and NVIDIA GPUs. Overall, performance differences among the AMG methods are fairly small despite their different algorithmic natures.*

best choice between 10^5 and 10^6 unknowns. Optimizations of the hardware or the driver stack for small system sizes are desirable for the AMD FirePro W9100 and the INTEL Xeon Phi, which do not take less than 0.1 seconds overall even for very small problem sizes.

6.5. Fine-grained parallel ILU factorization preconditioners. ILU factorization [47] is a popular family of preconditioners because of its black-box nature. However, in classical formulations fine-grained parallelism is neither available for the preconditioner setup nor for the preconditioner application. Techniques such as level scheduling [37] and multicoloring [30] have been developed to overcome some of the deficiencies, yet the efficiency of these techniques depends highly on the sparsity pattern of the system matrix.

We extended the recently developed fine-grained parallel ILU factorization preconditioners of Chow and Patel [16] to CUDA and OpenCL and present a performance comparison for the different hardware platforms from Table 4 in Figure 10. An al-

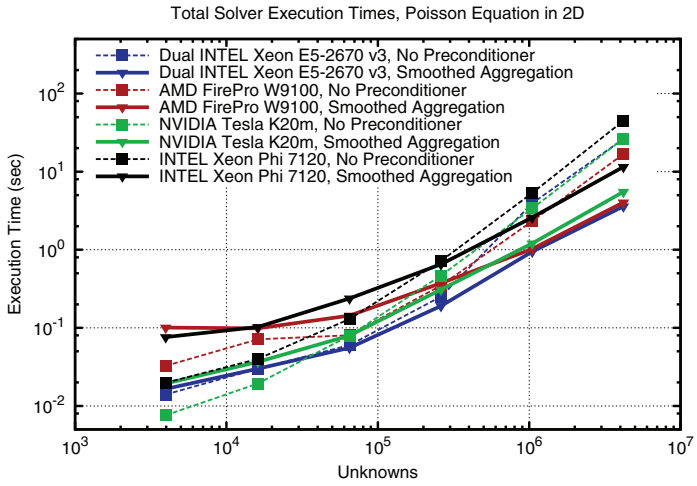

FIG. 9. *Comparison of total solver time (i.e., AMG setup time plus solver cycle time) for solving the two-dimensional Poisson equation using an unpreconditioned iterative solver, and a smoothed aggregation AMG preconditioner on each of the the four hardware platforms.*

ternative implementation based on CUDA was also developed concurrently by Chow, Anzt, and Dongarra [15]. Three nonlinear sweeps are carried out in the setup phase, as this has been identified to be a good default value [16]. Two Jacobiiterations are applied to the truncated Neumann series for the triangular solves in each preconditioner application.

Performance results were obtained for two- and three-dimensional solutions of the Poisson equation using linear finite elements on unstructured triangular and tetrahedral meshes of the unit square and unit cube, respectively. A BiCGStab solver was used to account for the asymmetric ILU preconditioner (static pattern, ILU0) and to better reflect practical applications where the system matrix is no longer symmetric. We observed that the setup time for the fine-grained ILU preconditioner is negligible compared to the solver cycle time. Therefore, any savings in execution time are obtained through reduced overall work in the solver cycle stage. Better performance than an unpreconditioned, pipelined BiCGStab solver was obtained if the reduction in solver iterations outweighs the additional cost of preconditioner applications in each iteration. Similar to the results for AMG, the OpenMP backend is the fastest for smaller sized problems, while GPUs are ultimately the fastest platform for larger problems. While the preconditioned solver results in shorter execution time overall in the two-dimensional case, the unpreconditioned, pipelined solver is faster in the three-dimensional case.

Overall, the fine-grained parallel ILU preconditioner is an attractive option whenever AMG cannot be applied and unpreconditioned solvers do not converge or converge only slowly. The overheads due to the sequential setup of the conventional ILU preconditioner are effectively removed, allowing for a full acceleration through multithreading and through GPUs.

7. Lessons learned and future directions. In the five years of continuously improving ViennaCL, a significant amount of experience and feedback from users has been accumulated. The purpose of this section is to share our experiences and provide concrete examples for software engineering questions which are often discussed in either a much more abstract setting, or using overly simplified examples.

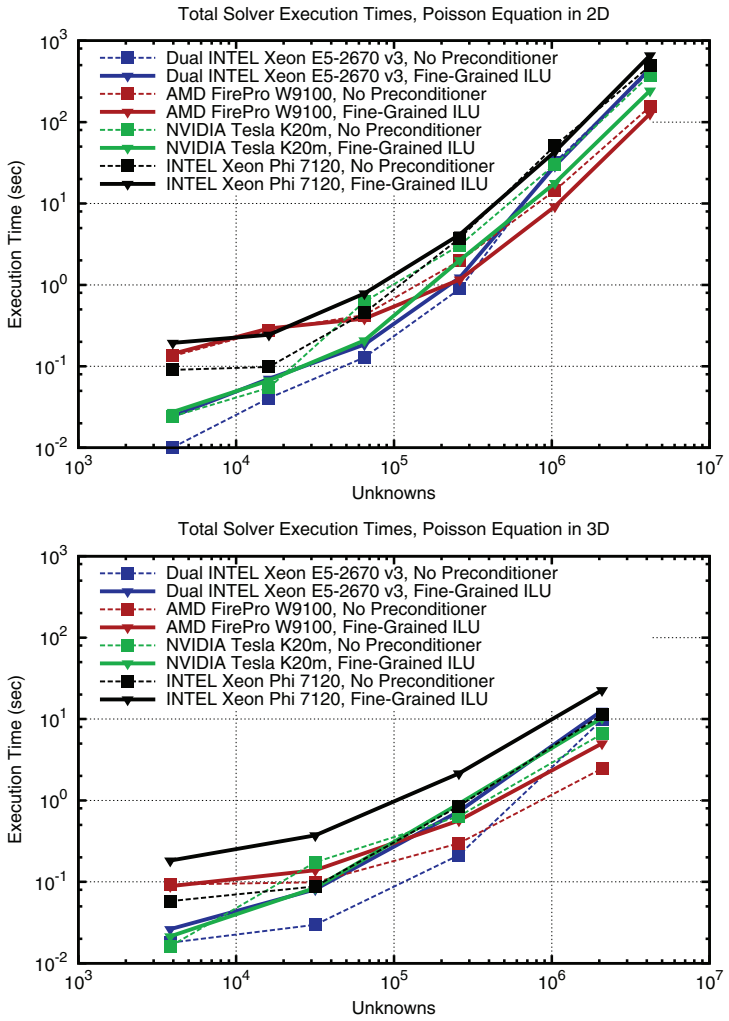

FIG. 10. *Comparison of total solver time (i.e., ILU setup time plus solver cycle time) for solving the two- (top) and three-dimensional (bottom) Poisson equation using an unpreconditioned iterative BiCGStab solver, and the fine-grained ILU preconditioner proposed by Chow and Patel* [16] *on each of the the four hardware platforms.*

7.1. Static versus dynamic dispatch. The design goal for the first ViennaCL release was to mimic the API of uBLAS as closely as possible in order to make a transition for C++ developers as easy as possible. As a consequence, the API relies heavily on static compile-time dispatches. For example, the numeric type of a vector <T> is statically determined by the template parameter T, or a sparse matrix type in compressed sparse row format is implemented in a class compressed_matrix<T>. If a user wants to switch between single and double precision arithmetic as well as different sparse matrix-vector formats at runtime, several disadvantages arise: First, the runtime dispatch needs to be written by the user. Second, compilation times increase in proportion to the number of combinations. With two different precisions and five sparse matrix formats, a tenfold increase in compilation times has to be expected. Third, unnecessary duplicated code ends up in the binary, potentially reducing performance due to increased instruction cache misses.

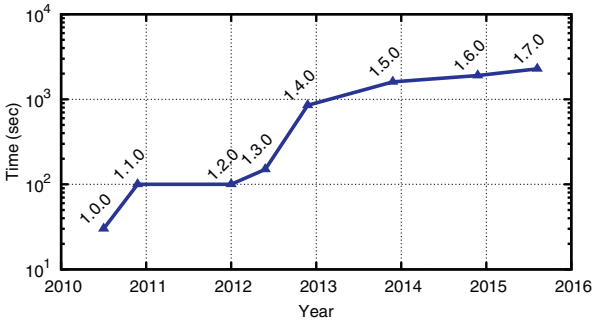

FIG. 11. *Total build time required for a full sequential build of all available targets for different versions of ViennaCL on a Core* 2 *Quad Q*9550 *using GCC* 4.6 *on Linux Mint Maya. Compiletime dispatches have increased compilation times for the test suite significantly.*

Figure 11 plots execution times for sequential builds of all targets in each new minor version upgrade of ViennaCL. Most of the build time is due to the statically dispatched test suite, which reflects the overall combinatorial explosion of choices. For example, the introduction of the CUDA and OpenMP backends in addition to the already existing OpenCL backend in release 1.4.0 increased compilation times for the test suite by a factor of three. Support for integer types in release 1.5.0 would result in another fivefold increase in build times, but a few test combinations are disabled such that overall compilation times increased only twofold. Some internal static dispatches have been converted to run time dispatches in the latest 1.7.0 release in order to reduce overall compilation times to more productive values.

7.2. Header only versus shared library. Because of the initial design goal to provide an API similar to uBLAS, ViennaCL uses a header-only approach, which means that the library is entirely implemented in header files. An advantage of this model is that users only need to include the source tree, but do not need to worry about linking static or shared libraries. The disadvantages, which we have found to outweigh the advantages as ViennaCL evolved, are as follows:

- *Leaky abstraction.* A header-only model fails to abstract the API from the internals. Tremendous efforts are required to ensure that the library compiles without warnings on all major compilers. This includes harmless warnings or even false positives, which only need to be eliminated because it would otherwise limit the possible set of compiler warnings the user could use. While we acknowledge that it is considered good practice to compile cleanly at high warning levels, a header-only approach forces one to go to even greater lengths without any additional benefit.
- *Compilation times.* Each compilation unit needs to parse not only the API, but also some of the implementations. As a result, compilation times are increased, which slow down the code-compile-run development cycle, reducing overall productivity. On machines with small amounts of main memory such as mobile devices, the compilation may then even fail because it runs out of memory.
- *CUDA compiler enforced on user code.* If the CUDA backend of ViennaCL is enabled, a compilation with the CUDA compiler NVCC is required. Unless the user isolates ViennaCL in a separate compilation unit, effectively creating a static or shared library, the user code also needs to be compiled with NVCC.

However, this severely restricts the user's compiler choice, particularly if the user code does not compile with NVCC for reasons beyond the user's control. In contrast, a shared library with a clear binary interface hides the use of NVCC induced by the CUDA backend from the user.

As a consequence, ViennaCL 2.0.0 will no longer be provided using a header-only approach. Instead, all functionality will be compiled into a static or shared C-library, through which all functionality is exposed. A C++ layer will then be a lightweight layer on top, generating a much smaller load for compilers.

7.3. Play nicely with others. ViennaCL provides a convenient C++-API—which is a problem. C++ lacks a standardized binary interface, which makes it extremely difficult to interface from other languages such as C, FORTRAN, or Python. Therefore, in addition to what has been discussed earlier, a shared library with a well-defined binary interface is better suited to the needs of the scientfic computing community.

Similarly, several users requested that the library be able to work with user-provided buffers directly. For example, a user may have assembled a sparse matrix using CUDA already, in which case it is essential that iterative solver implementations provided by a library are able to work directly with these buffers. Earlier versions of ViennaCL were unable to directly work on such buffers and required an expensive data conversion step. The latest version supports such a direct reuse and future work aims to further simplify such a use case. Conversely, a user may want to run custom operations on, e.g., the OpenCL buffer of a ViennaCL vector. This again requires access to the low-level buffer handles.

8. Summary and conclusion. The different parallel programming models make it hard for the computational scientist to leverage the computational resources of the various machines available. Ideally, a computational scientist only interfaces with a single library in order to use multi- and many-core architectures. Unlike most other libraries available, ViennaCL enables the computational scientist to do exactly that, whereas the use of vendor libraries requires application codes to manually manage different interfaces.

We have demonstrated that good portable performance can be obtained for all three computing backends and that target-specific optimizations can still be applied. Portable performance for GPUs is obtained via a device database coupled with a kernel code generator, which transforms device-specific code templates into valid code to be passed to the just-in-time compiler. Because just-in-time compilation involves additional overhead, we discussed the OpenCL kernel organization in ViennaCL such that the overall overhead is minimized.

Our benchmarks have shown that the use of GPUs does not automatically imply higher performance, because the large caches of CPUs and their broader applicability to general purpose workloads may result in better performance overall. Performance differences are seldom larger than a factor of two for typical workloads from sparse matrix linear algebra when comparing similar hardware in terms of cost and performance per watt.

Acknowledgments. The authors are grateful to Joachim Schöberl for providing access to some of the hardware used for performance comparisons in this work, and to AMD and NVIDIA for hardware donations.

REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, *Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects*, J. Phys. Conf. Ser., 180 (2009), 012037.

[2] J. I. Aliaga, J. Pérez, E. S. Quintana-Ortí, and H. Anzt, *Reformulated conjugate gradient for the energy-aware solution of linear systems on GPUs*, in Proceedings of the Internatioal Conference on Parallel Processing, IEEE, Piscataway, NJ, 2013, pp. 320–329.

[3] J. I. Aliaga, J. Pérez, and E. S. Quintana-Ortí, *Systematic fusion of CUDA kernels for iterative sparse linear system solvers*, in Euro-Par 2015: Parallel Processing, Lecture Notes in Comput. Sci. 9233, Springer, Heidelberg, 2015, pp. 675–686.

[4] H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, I. Yamazaki, and J. Dongarra, *Optimizing Krylov subspace solvers on graphics processing units*, in IEEE International Parallel and Distributed Processing Symposium Workshops, IEEE, Piscataway, NJ, 2014, pp. 941–949.

[5] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, *Fast sparse matrix-vector multiplication on GPUs for graph applications*, in International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, Piscataway, NJ, 2014, pp. 781–792.

[6] W. Bangerth and T. Heister, *What makes computational open source software libraries successful?*, Comput. Sci. Discov., 6 (2013), 015010/1.

[7] W. Bangerth and T. Heister, *Quo Vadis, Scientific Software?*, SIAM News, 47 (2014), 1.

[8] M. M. Baskaran and R. Bordawekar, *Optimizing sparse matrix-vector multiplication on GPUs*, Technical report RC24704, IBM 2008.

[9] N. Bell, S. Dalton, and L. N. Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, SIAM J. Sci. Comput., 34 (2012), pp. C123–C152.

[10] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, New York, 2009, 18.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, New York, 1995, pp. 207–216.

[12] J. Brown, M. G. Knepley, and B. F. Smith, *Run-time extensibility and librarization of simulation software*, Comput. Sci. Eng., 17 (2015), pp. 38–45.

[13] J. Brown, *HPGMG: Benchmarking computers using multigrid*, Copper Mountain Multigrid Conference 2015, https://jedbrown.org/files/20150324-HPGMG.pdf (2015).

[14] A. Buluç and J. R. Gilbert, *Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments*, SIAM J. Sci. Comput., 34 (2012), pp. C170–C191.

[15] E. Chow, H. Anzt, and J. Dongarra, *Asynchronous iterative algorithm for computing incomplete factorizations on GPUs*, in High Performance Computing, Lecture Notes in Comput. Sci. 9137, Springer, Cham, 2015, pp. 1–16.

[16] E. Chow and A. Patel, *Fine-grained parallel incomplete LU factorization*, SIAM J. Sci. Comput., 37 (2015), pp. C169–C193.

[17] L. Dagum and R. Menon, *OpenMP: An industry standard API for shared-memory programming*, IEEE Comput. Sci. Eng., 5 (1998), pp. 46–55.

[18] S. Dalton, N. Bell, L. Olson, and M. Garland, *Cusp: Generic parallel algorithms for sparse matrix and graph computations*, Version 0.5.1, http://cusplibrary.github.io/ (2014).

[19] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1:1–1:25.

[20] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling, *Programming CUDA and OpenCL: A case study using modern C++ libraries*, SIAM J. Sci. Comput., 35 (2013), pp. C453–C472.

[21] Message Passing Forum, *MPI: A Message-Passing Interface Standard*, Technical report, University of Tennessee, Knoxville, TN, 1994.

[22] R. Gandham, K. Esler, and Y. Zhang, *A GPU accelerated aggregation algebraic multigrid method*, Comp. Math. Appl., 68 (2014), pp. 1151–1160.

[23] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose, *Hiding global communication latency in the GMRES algorithm on massively parallel machines*, SIAM J. Sci. Comput., 35 (2013), pp. C48–C71.

[24] P. Ghysels and W. Vanroose, *Hiding global synchronization latency in the preconditioned conjugate gradient algorithm*, Parallel Comput., 40 (2014), pp. 224–238.

[25] J. R. Gilbert, V. B. Shah, and S. Reinhardt, *A unified framework for numerical and combinatorial computing*, Comput. Sci. Eng., 10 (2008), pp. 20–25.

[26] P. Gottschling and C. Steinhardt, *Meta-Tuning in MTL4*, International Conference of Numerical Analysis and Applied Mathematics, AIP Conf. Proc. 1281 (2010), pp. 778–782.
[27] J. L. Greathouse and M. Daga, *Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format*, in International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, Piscataway, NJ, 2014, pp. 769–780.
[28] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling, and U. Naumann, *GPU-accelerated sparse matrix-matrix multiplication by iterative row merging*, SIAM J. Sci. Comput., 37 (2015), pp. C54–C71.
[29] *Eigen v3*, http://eigen.tuxfamily.org (2010).
[30] V. Heuveline, D. Lukarski, and J.-Ph. Weiss, *Enhanced parallel ILU(p)-based preconditioners for multi-core CPUs and GPUs – The power(q)-pattern method*, Preprint Series of the Engineering Mathematics and Computing Lab, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2011.
[31] J. Hoberock and N. Bell, *Thrust: A Parallel Template Library*, http://eigen.tuxfamily.org (2010).
[32] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, *MPI + MPI: A new hybrid approach to parallel programming with MPI plus shared memory*, Computing, 95 (2013), pp. 1121–1136.
[33] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, Addison-Wesley, Upper Saddle River, NJ, 2010.
[34] K. Iglberger, G. Hager, J. Treibig, and U. Rüde, *Expression templates revisited: A performance analysis of current methodologies*, SIAM J. Sci. Comput., 34 (2012), pp. C42–C69.
[35] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units*, SIAM J. Sci. Comput., 36 (2014), pp. C401–C423.
[36] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, *Efficient sparse matrix-vector multiplication on x86-based many-core processors*, in International Conference on Supercomputing, ACM, New York, 2013, pp. 273–282.
[37] R. Li and Y. Saad, *GPU-accelerated preconditioned iterative linear solvers*, J. Supercomput., 63 (2013), pp. 443–466.
[38] J. D. McCalpin, *Memory bandwidth and machine balance in current high performance computers*, Computer Society Technical Committee on Computer Architecture Newsletter, (1995), pp. 19–25.
[39] J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Scalable parallel programming with CUDA*, Queue, 6 (2008), pp. 40–53.
[40] G. Penn, *Efficient transitive closure of sparse matrices over closed semirings*, Theoret. Comput. Sci., 354 (2006), pp. 72–81.
[41] J. Reinders, *Intel Threading Building Blocks*, O'Reilly, Beijing, 2007.
[42] K. Rupp, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, and A. Jüngel, *Optimized Sparse Matrix-Matrix Multiplication for Multi-Core CPUs, GPUs, and Xeon Phi*, manuscript.
[43] K. Rupp, F. Rudolf, and J. Weinbub, *ViennaCL - A high level linear algebra library for GPUs and multi-core CPUs*, in International Workshop on GPUs and Scientific Applications, 2010, pp. 51–56.
[44] K. Rupp, Ph. Tillet, F. Rudolf, J. Weinbub, T. Grasser, and A. Jüngel, *Performance portability study of linear algebra kernels in OpenCL*, in Proceedings of the International Workshop on OpenCL 2013-2014, IWOCL '14, ACM, New York, 2014, p. 8.
[45] K. Rupp, J. Weinbub, T. Grasser, and A. Jüngel, *Pipelined iterative solvers with kernel fusion for graphics processing units*. ACM Trans. Math. Software, 43 (2016), 11.
[46] K. Rupp, J. Weinbub, F. Rudolf, A. Morhammer, T. Grasser, and A. Jüngel, *A Performance Comparison of Algebraic Multigrid Preconditioners on CPUs, GPUs, and Xeon Phis*, manuscript.
[47] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, Philadelphia, 2003.
[48] C. Sanderson, *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*, Technical report, NICTA, 2010.
[49] E. Saule, K. Kaya, and Ü. Catalyürek, *Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi*, in Parallel Processing and Applied Mathematics, Lecture Notes in Comput. Sci. 8384, Springer, Heidelberg, 2014, pp. 559–570.
[50] B. Schäling, *The Boost C++ Libraries*, XML Press, Laguna Hills, CA, 2011.
[51] J. Schöberl, *NETGEN an advancing front 2D/3D-mesh generator based on abstract rules*, Comput. Vis. Sci., 1 (1997), pp. 41–52.
[52] J. E. Stone, D. Gohara, and G. Shi, *OpenCL: A parallel programming standard for heterogeneous computing systems*, IEEE Des. Test, 12 (2010), pp. 66–73.

[53] B.-Y. Su and K. Keutzer, *clSpMV: A cross-platform OpenCL SpMV framework on GPUs*, in Proceedings of the ACM International Conference on Supercomputing, ICS '12, ACM, New York, 2012, pp. 353–364.
[54] Ph. Tillet, K. Rupp, S. Selberherr, and C.-T. Lin, *Towards performance-portable, scalable, and convenient linear algebra*, in 5th USENIX Workshop on Hot Topics in Parallelism (HotPar'13), USENIX, Berkeley, CA 2013.
[55] N. Trost, J. Jiménez, D. Lukarski, and V. Sanchez, *Accelerating COBAYA3 on multi-core CPU and GPU systems using PARALUTION*, Ann. Nucl. Energy, 82 (2015), pp. 252–259.
[56] U. Trottenberg, C. W. Oosterlee, and Anton Schüller, *Multigrid*, Academic, San Diego, CA, 2001.
[57] D. Vandevoorde and N. M. Josuttis, *C++ Templates*, Addison-Wesley, Berlin, 2002.
[58] S. Van Dongen, *Graph clustering via a discrete uncoupling process*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 121–141.
[59] T. Veldhuizen, *Expression Templates*, C++ Rep., 7 (1995), pp. 26–31.
[60] S. Yan, Ch. Li, Y. Zhang, and H. Zhou, *yaSpMV: Yet another SpMV framework on GPUs*, in Proc. ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, PPoPP '14, ACM, New York, 2014, pp. 107–118.