ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs

Karl Rupp
CD Laboratory for Reliability Issues in Microelectronics
IuE, TU Wien, A-1040 Wien
rupp@iue.tuwien.ac.at

Florian Rudolf
Institute for Microelectronics
Gußhausstraße 27-29/E360
TU Wien, A-1040 Wien
rudolf@iue.tuwien.ac.at

Josef Weinbub
Institute for Microelectronics
Gußhausstraße 27-29/E360
TU Wien, A-1040 Wien
weinbub@iue.tuwien.ac.at

ABSTRACT

The vast computing resources in graphics processing units (GPUs) have become very attractive for general purpose scientific computing over the past years. Moreover, central processing units (CPUs) consist of an increasing number of individual cores. Most applications today still make use of a single core only, because standard data types and algorithms in wide-spread procedural languages such as C++ make use of a single core only. A customized adaption of existing algorithms to parallel architecture requires a considerable amount of effort both from algorithmic and programming point of view. Taking this additional amount of work hours required for an adaption to GPUs starting from scratch into account, the use of GPUs may not pay off on the overall.

The Vienna Computing Library (ViennaCL), which is presented in this work, aims at providing standard data types for linear algebra operations on GPUs and multi-core CPUs. It is based on OpenCL, which provides unified access to both GPUs and multi-core CPUs. The ViennaCL API following existing programming and interface conventions established with uBLAS, which is part of the peer-reviewed Boost library. Thus, the open source library can be easily integrated into existing C++ implementations and therefore reduces the necessary code changes in existing software to a minimum. In addition, algorithms provided with ViennaCL can directly be used with uBLAS types due to the common interface.

The algorithmic focus of ViennaCL is on iterative solvers, which are often used for the solution of large systems of linear equations typically encountered in the discretization of partial differential equations using e.g. finite element methods. Benchmark results given in this work show that the performance gain of ViennaCL over uBLAS is on both GPUs and multi-core CPUs up up to an order of magnitude. For small amounts of data, the use of ViennaCL may not pay off due to an OpenCL management overhead associated with the launch of compute kernels.

1. INTRODUCTION

General purpose scientific computing on GPUs has become very attractive over the past years [11–13, 18]. In the early days of such computations, the lack of double precision arithmetic was often considered a major drawback. However, recent GPUs such as a NVIDIA Geforce GTX 470 or an ATI Radeon HD 5850 used for the benchmarks in this work do not suffer from this restriction any longer, thus they push into the field of high performance computing (HPC). Simultaneously, CPUs consist of an increasing number of cores, for which many serial algorithms become less and less attractive.

Considerable performance gains have been reported [11–13, 18], but the adaption of existing algorithms to GPUs starting from scratch requires a considerable amount of change in existing codes to account for the highly parallel architecture of GPUs. Consequently, the effort required for porting an existing code to GPUs was often considered to be too large to have a considerable benefit on the overall. In particular, programmers are required to learn specialized programming languages like CUDA [14] or OpenCL [20], even if only standard linear algebra algorithms such as defined by the basic linear algebra subprograms (BLAS) [2] are to be executed on the GPU. It is thus desirable to have data types that provide parallel standard operations on the target machine, utilizing the available hardware in the best possible way.

There is a number of linear algebra libraries for GPUs available, for example ACML-GPU [1], CULA [4], MAGMA [6] or CUBLAS [14], focusing on computationally demanding operations such as dense matrix-matrix multiplications. However, sparse matrix arithmetic and iterative solvers are much less pronounced or not provided at all, even though this type of matrices is common for the discretization of partial differential equations. Cusp [5] provides two iterative solvers, but only matrix-vector products are computed on the GPU, leading to considerable memory transfer overhead. The CNC plugin in OpenNL [7] provides only a single iterative solver. The functionality provided by these libraries is available through function calls, which provide a certain set of basic operations. Thus, appropriate data setup and initialization is typically left to the user. The approach of the Vienna Computing Library (ViennaCL) [9] presented in this work is to wrap GPU data in high level C++ datatypes and provide an interface that adheres to established conventions. In the following, we refer to version 1.0.5 of the library.

This paper is organized as follows: First, the design of ViennaCL is discussed in Sec. 2. Sec. 3 presents the library interface for linear algebra operations on BLAS levels 1 and 2. The iterative solvers provided with ViennaCL are explained in Sec. 4. The inclusion of custom compute kernels is discussed in Sec. 5 and benchmark results are given in Sec. 6. Finally, an outlook to future work is given in Sec. 7 and a conclusion is drawn in Sec. 8.

2. DESIGN OF VIENNACL

The roots of ViennaCL are in the need for fast iterative solvers for the solution of large sparse systems arising from the discretization of partial differential equations (PDEs) for use in our in-house simulators. To allow other researchers and engineers to benefit from our effort, ViennaCL is designed to be used with other modern software packages that serve a similar purpose, e.g. deal.ii [15], Getfem++ [17] or Sundance [25], which are all implemented in C++. Consequently, C++ is chosen for the implementation of ViennaCL.

For accessing GPUs, the two main branches are CUDA [14] and OpenCL [20]. While CUDA is tailored to the specific architecture of NVIDIA GPUs, the first royalty-free standard for cross-platform parallel programming, OpenCL, provides much higher flexibility with respect to the underlying hardware. Thus, OpenCL supports a superset of the hardware supported by CUDA and is not limited to GPUs. Moreover, CUDA kernels need to be precompiled by a separate compiler, while OpenCL allows just-in-time compilation of the source code on the target machine. The latter approach is especially attractive for developers, because this allows to create header-only libraries. For these reasons, OpenCL is chosen for low level hardware programming.

The major design goal of ViennaCL is to be convenient and easy to use. For simple integration into existing projects, ViennaCL is a header-only library, which simplifies the build process considerably. On the other hand, initialization and management of OpenCL is done completely in the background and is discussed in the following subsections.

2.1 Hardware Initialization

A common approach in parallel software such as PETSc [10] is to rely on dedicated initialization routines that have to be called by the library user prior to any use of other functionality. In ViennaCL, hardware initialization is automatically triggered when the first object of a type provided by ViennaCL such as `scalar` or `vector` is created. In the background, available devices are queried. If a suitable GPU is available, it is then used for all calculations, otherwise ViennaCL searches for a CPU supported by the OpenCL implementation. The simultaneous use of multiple devices is not included in version 1.0.5 of ViennaCL, because multi-device support was added to OpenCL only recently [20].

2.2 Source Code Compilation

The compilation of OpenCL source code at each run of ViennaCL leads to additional setup costs during the automatically triggered initialization phase. A full compilation of all OpenCL sources included in ViennaCL takes several seconds and may be too long for certain applications. Therefore, ViennaCL groups sources into smaller compilation units associated with the basic types and the underlying floating point precision. This allows a on-demand compilation: The first time an object of a particular type is created, all OpenCL kernels associated with that particular type are compiled. A more fine-grained compilation on a per kernel basis, which compiles a kernel at the first invocation, turned out to have larger overall setup costs in most cases. This just-in-time compilation reduces setup times to a bare minimum.

2.3 Transfer between Host and Device

Prior to any calculations on GPUs, the data needs to be transferred from the host memory to the OpenCL device memory (e.g. GPU RAM). Even if ViennaCL is used on multi-core CPUs, data also needs to be set up accordingly in the OpenCL layer.

Since every data transfer from host memory to device memory and back from the device memory to host memory can be seen as a copy operation, ViennaCL reuses the conventions introduces with the Standard Template Library (STL) (see e.g. [24]). In order to copy all entries of a vector `cpu_vec` from the host to a vector `gpu_vec` in the GPU memory, the call

```
1 copy(cpu_vec.begin(),
2      cpu_vec.end(),
3      gpu_vec.begin());
```

is sufficient. The member functions `begin()` and `end()` return iterators pointing to the beginning and the end of the vector respectively. Thus, programmers acquainted with the iterator concept and the STL can reuse their knowledge. Moreover, parts of a vector can be manipulated easily and also plain pointers to CPU data can be supplied. A shorthand notation for the above code line is

```
1 copy(cpu_vec, gpu_vec);
```

which only requires that the `begin()` and `end()` member functions are available for the respective type of `cpu_vec`.

For dense matrix types, the iterator concept could also be used in principal, but matrix dimensions would have to be supplied in addition. Instead, data transfer from a matrix `cpu_matrix` on the host, no matter if dense or sparse, to a matrix `gpu_matrix` on the device is accomplished with

```
1 copy(cpu_matrix, gpu_matrix);
```

For this generic interface a number of type requirements needs to be imposed on the type of the dense `cpu_matrix`, which are as follows:

- A member function `size1()` provides the number of rows
- A member function `size2()` provides the number of columns
- Entries are accessed using the parenthesis operator with index range starting at zero.

These conventions are fulfilled by uBLAS types, so data set up in a dense uBLAS matrix can exchanged with ViennaCL with a single line of code. Library users willing to use a dense matrix type not fulfilling these requirements have to provide a wrapper class.

For sparse matrix types, instead of overloaded parenthesis operators, data must be accessible via iterators as in uBLAS [8]. As an alternative using only STL types, a sparse matrix can also be supplied in a vector of maps, i.e.

```
1 vector< map<unsigned int, NumericT> >
```

where `NumericT` is either `float` or `double`.

To modify individual entries of a vector or a dense matrix located on the OpenCL device, overloaded operators are provided. Sparse matrix types cannot be manipulated directly in OpenCL memory in ViennaCL 1.0.5. For example, setting the fifth element of a vector `gpu_vec` to seven, the line

```
1 gpu_vec(4) = 7;
```

is sufficient. Note that the indices start with zero. Under the hood, the parenthesis operator in `gpu_vec(4)` returns a proxy class, for which the assignment operator is overloaded and the transfer from host to device is initiated. However, direct initialization of all entries on the GPU as in

```
1 // one possible initialization of device
2 // memory (not recommended!)
3 for (int i=0; i<100000; ++i)
4   gpu_vec(i) = i;
```

is not recommended, because each update initiates a separate transfer with a significant overhead. Thus, the loop above takes four to five orders of magnitude longer than for pure host types. A much faster alternative is

```
1 std::vector<NumericT> cpu_vec(100000);
2 for (int i=0; i<100000; ++i)
3   cpu_vec(i) = i;
4 copy(cpu_vec, gpu_vec);
```

which has only small overhead due to creation of the temporary vector `cpu_vec` and the copy operation at the end of the for-loop. Consequently, it is recommended to fully set up the data (i.e. vectors, matrices) on the CPU host, then copy to the device and start processing the data with ViennaCL.

2.4 Kernel Execution

To start an OpenCL kernel, arguments need to be set and several parameters need to be supplied using the C interface. In ViennaCL, however, operator overloads and other abstraction mechanisms in C++ allow an encapsulation of all these details. For example, the addition of two vectors `vec2` and `vec3`, typically written in C++ using operator overloads as

```
1 vec1 = vec2 + vec3;
```

requires the launch of the appropriate OpenCL kernel with the memory locations and the vector lengths as kernel arguments. All these details are encapsulated, so that users of ViennaCL do not have to deal with OpenCL internals.

3. BASIC LINEAR ALGEBRA

There are many linear algebra libraries available in C++, one of the most commonly used is uBLAS [8] included in the peer-reviewed Boost libraries [3]. In contrast to early implementations of BLAS functionality in FORTRAN, overloaded operators are used in uBLAS whenever appropriate. ViennaCL accounts for the broad acceptance of the approach by uBLAS and provides an interface that is to a large extent a subset of that of uBLAS. More precisely, any code for algorithms using linear algebra operations from ViennaCL is required to be also valid when using uBLAS objects. This simplifies testing and verification on the one hand and is a benefit for uBLAS library users due to reusable algorithms on the other hand.

The basic types used for linear algebra operations on BLAS level 1 and 2 are the following:

```
1 scalar<NumericT> s; //scalar
2 vector<NumericT> v; //vector
3 matrix<NumericT> m; //dense matrix
4 compressed_matrix<NumericT> c1; //CSR
5 coordinate_matrix<NumericT> c2; //(i,j,aij)
```

Here, `NumericT` denotes the underlying floating point type (either `float` or `double`). The `compressed_matrix` type stores a sparse matrix in a compressed sparse rows format (see e.g. [21]), while `coordinate_matrix` stores all matrix entries as triplets (i, j, a_{ij}), where i is the row index, j is the column index and a_{ij} is the corresponding entry.

BLAS functionality in ViennaCL can be invoked similarly to uBLAS using overloaded operators:

```
1 // BLAS level 1
2 // x, y and z are vectors
3 y = 2.0 * x;
4 z = x + y;
5 x += 3.1415 * z;
6 NumericT n1 = norm_1(x);
7 NumericT n2 = norm_2(y);
8 NumericT ninf = norm_inf(z);
9 plane_rotation(x, y, n1, n2);
```

The first three code lines manipulate vectors using overloaded operators. Unlike in naive C++, where expressions like `x += 3.1415 * z;` would lead to a temporary object for `3.1415 * z`, none of the expressions above leads to a temporary object due to the use of expression templates [27, 28]. Internally, only a single multiply-add kernel is called for this example with vectors arguments `x`, `z` and scalar argument `3.1415`. Temporary objects on GPUs are much more detrimental for performance and should thus be avoided, since allocation has to be done via the OpenCL layer. Lines 6 to 8 in the above snippet compute the l^1-, l^2- and l^∞-norm of the respective function argument. The last line performs a plane rotation of the argument vectors as required by BLAS level 1.

On BLAS level 2, ViennaCL and uBLAS are also fully compatible:

```
1 // BLAS level 2
2 // x, y are vectors, A is a matrix
3 y = prod(A, x);       //matrix-vector product
4 x = prod(trans(A), x);  //transposed product
5 y = alpha * prod(A, x) + beta * y
6 y = solve(A, x, tag);   //triangular solver
7 inplace_solve(A, x, tag);
8 A += alpha * outer_prod(x,y); //rank1 update
```

Lines 3 to 5 show matrix vector products are handled. Lines 6 and 7 call a triangular solver for dense matrices, where the variable `tag` is either `upper_tag`, `lower_tag`, `unit_upper_tag` or `unit_lower_tag` and is used to choose the dense linear solver.

4. ITERATIVE SOLVERS

In many applications such as the discretization of partial differential equations using finite element or finite difference methods, large sparse systems of linear equations need to be solved. While direct methods can be used for moderate problem sizes, iterative solvers are necessary for large systems of equations. The BLAS levels defined for sparse matrices [16] are not fully implemented in ViennaCL 1.0.5 yet, but the most important sparse operation, namely sparse matrix vector products, is provided and serves as a building block for iterative solvers.

The choice of a suitable iterative solver strongly depends on the properties of the system of linear equations. ViennaCL 1.0.5 provides the following three iterative solvers, which cover most application areas:

- Conjugate Gradient (CG) [19] for the solution of symmetric, positive definite systems.
- Stabilized Bi-Conjugate Gradient (BiCGStab) [26] for positive definite systems.
- Generalized Minimum Residual (GMRES) [22, 29] for general systems.

Since no iterative solvers are provided by uBLAS, the interface for the iterative solvers was designed such that it naturally extends the existing solver interface for the triangular solvers. In ViennaCL, the BLAS level 2 call for dense matrices

```
y = solve(A, x, tag);
```

is extended to support the additional tags `cg_tag`, `bicgstab_tag` and `gmres_tag`, hence the solvers can be called using

```
// CG solver:
result = solve(matrix, rhs, cg_tag());
// BiCGStab solver:
result = solve(matrix, rhs, bicgstab_tag());
// GMRES solver:
result = solve(matrix, rhs, gmres_tag());
```

Additional solver parameters can be passed to the constructors of these tags to specify tolerances and maximum iteration counts. For example, a relative tolerance of 10^{-8} and at most 200 iterations for a CG solver can be set with the line

```
result=solve(matrix,rhs, cg_tag(1e-8,200) );
```

Since uBLAS and ViennaCL are mostly interface compatible, the generic implementation of the iterative solvers allows to directly reuse them with uBLAS types. Thus, the same iterative solver code allows to run the iterative solver either on GPUs or multi-core CPUs using ViennaCL or on a single CPU core using uBLAS. For other matrix and vector types, a wrapper facility allows library users to customize free functions such as `prod()` for matrix-vector products, `norm_2()` for computing the l^2-norm or `inner_prod()` for computing inner products to fit other matrix and vector types from external libraries.

The convergence of iterative solvers can be greatly improved by the use of preconditioners. ViennaCL 1.0.5 provides an optional incomplete LU factorization (ILUT) preconditioner with threshold [21], other preconditioners are in preparation. The ILUT preconditioner is due to its inherent serial structure always computed and applied on the CPU, thus the preconditioner is likely to serve as a bottleneck for an otherwise GPU accelerated iterative solver.

Preconditioners are supplied as an optional fourth argument to the function `solve()`. For example, an ILUT preconditioner can be used within a conjugate gradient solver by writing

```
// Set up ILUT
ilut_precond< compressed_matrix<NumericT> >
  ilut(matrix, ilut_tag());

// CG solver with ILUT preconditioner:
result = solve(matrix, rhs, cg_tag(), ilut);
```

Additional parameters for ILUT can be provided to the constructor of `ilut_tag` similar to the specification of parameters in solver tags. Again, the preconditioner can be used both for uBLAS types and for ViennaCL types. The generic solver interface also allows to provide custom preconditioners, the only requirement is that the parenthesis operator is defined for a vector argument.

5. CUSTOM COMPUTE KERNELS

Unlike other libraries, ViennaCL directly supports user-defined compute kernels written in OpenCL. The user can fully focus on the kernel, since details of the underlying OpenCL implementation are handled internally by ViennaCL.

For example, a kernel for elementwise products of two vectors is the following:

```
__kernel void elementwise_prod(
          __global const float * vec1,
          __global const float * vec2,
          __global float * result,
          unsigned int size)
{
  for (int i = get_global_id(0);
           i < size;
           i += get_global_size(0))
    result[i] = vec1[i] * vec2[i];
}
```

`vec1` and `vec2` denote the operands, `result` is the result vector and `size` the length of the vectors. Details on the OpenCL programming language, which is a subset of C with some extensions for parallelism, can be found in the specification [20], where in particular the keywords `__kernel`, `__global` and the functions `get_global_id()` and `get_global_size()` are explained. With a few additional code lines, the above kernel can be launched for three ViennaCL vectors of type `vector<float>`.

The possibility to easily include custom compute kernels in ViennaCL allows to run a long chain of possibly custom operations on the GPU without the overhead of copying data between host and device. For example, a custom matrix-vector multiplication kernel could be required for a specialized matrix of type, say, `A`. After writing the custom OpenCL kernel and overloading

Compute Device	float	double
Intel i7 960, single core	0.33	0.32
Intel i7 960, ViennaCL	**1.98**	0.85
NVIDIA Geforce GTX 470	1.88	**1.66**
ATI Radeon HD 5850	0.86	0.89

Table 1: Computational speed (in GFLOPs) for inner products of vectors with 3 000 000 entries. Multiply-add operations are counted as single floating point operations.

Compute Device	float	double
Intel i7 960, single core	0.17	0.16
Intel i7 960, ViennaCL	1.06	0.81
NVIDIA Geforce GTX 470	**1.71**	**1.10**
ATI Radeon HD 5850	1.30	0.93

Table 2: Computational speed (in GFLOPs) for sparse matrix-vector multiplication using `compressed_matrix`. The 65 025 matrix rows have seven nonzero entries on average. Multiply-add operations are counted as single floating point operations.

```
1 prod(A & a, vector<T> & b);
```

for matrix-vector products, objects of type `A` can directly be passed to the iterative solvers provided. Thus, the possibility to provide custom compute kernels and the generic implementation of the algorithms in ViennaCL result in high flexibility for the library user.

6. PERFORMANCE

The performance of ViennaCL, version 1.0.5, is compared on GPUs from ATI and NVIDIA and a CPU from Intel. uBLAS is used to measure the performance on a single CPU core. The test platform was a Intel Core i7 960 with 4 physical cores, 8 logical cores, and 6 Gigabytes of random access memory, running a 64-bit Linux kernel. Stream SDK 2.2 was used with kernel of version 2.6.33 and GPU driver version 10.6. The Stream SDK was also used for running ViennaCL in parallel on the CPU. We observed that benchmark results for ViennaCL using Stream SDK under Windows 7 are by up to 30 percent better, especially when using double precision, hence the performance of ViennaCL is likely to improve with better OpenCL support in the future. For NVIDIA GPUs, a kernel with version 2.6.34 and a GPU driver, version 195.36.24, was used. When evaluating the following benchmark results in computational speed per money, it has to be considered that the CPU is by a factor of around two more expensive than each of the GPUs. All compute kernels are launched with the default settings in ViennaCL, namely 128 work groups with 128 work items each.

In Tab. 1 benchmark results for inner products are shown. Performance gains on GPUs and a fully loaded multi-core CPU of a factor of up to six compared to a single CPU core are observed. In double precision, the parallel execution on the CPU still results in a performance gain of a factor 2.6. A curiosity is that the OpenCL implementation of the Stream SDK provides better performance in double precision than in single precision on the GPU. We assume that this is due to the early stage of OpenCL support by ATI.

Execution times for matrix-vector products in Tab. 2 depict that the performance benefit over a single CPU core is around a factor of ten in single precision and about a factor of seven in double precision. Running ViennaCL on the CPU results in about 60 to 90 percent of the performance of the GPUs. We note that additional notable performance gains on GPUs can be obtained by the use of hybrid formats [11, 12], which are not included in ViennaCL yet. Additionally, we observed that the use of vector data types in the OpenCL kernels doubles performance on the GTX 470 in this case.

Tab. 3 lists the execution times for two iterative solvers. The CG solver is accelerated by a factor of five on the NVIDIA GPU, and only slightly on the ATI GPU. The performance gain for BiCGStab is comparable to that of CG. Using the ATI Stream SDK, a significant overhead of OpenCL kernel launches becomes apparent: While the performance of sparse matrix vector products, inner products and vector additions is comparable on the two GPUs, a call of several different kernels has a much larger overhead using the Stream SDK than for the NVIDIA implementation of OpenCL.

The observed performance gains of GPUs over CPUs for iterative solvers are essentially determined by the available memory bandwidth, because the iterative solvers use BLAS level 1 and 2 functions only. Further speedups can possibly be obtained if parts of the assembly algorithm for the linear system of equations are also ported to OpenCL. Higher performance gaps are usually observed for BLAS level 3 functions, e.g. matrix-matrix products, which are computationally more demanding than lower BLAS levels. However, no BLAS level 3 functionality is provided in version 1.0.5 of ViennaCL, but planned for future versions.

7. OUTLOOK

With the possibility of using ViennaCL on many different platforms, a global number of work groups and work items is not sufficient to yield reasonable performance on all target devices. While the choice is easier on CPUs due to the smaller number of cores on a die, it has a much stronger influence on GPUs. In particular, a higher number of work items or work groups does not necessarily result in better performance due to synchronization overhead. Thus, work on an automated tuning environment is in progress, which aims at finding the best set of parameters for each compute kernel. Performance gains of about 25 percent have already been observed for the operations compared in Sec. 6.

Having fast sparse matrix vector product available, an implementation of eigenvalue computations using either Lanczos' or Arnoldi's method is in progress. Simple implementations often suffer from severe round-off errors that introduce so-called ghost eigenvalues, therefore orthogonality of the Krylov basis has to be ensured by e.g. partial reorthogonalization [23]. The time consuming matrix-vector and inner products can then be carried out on the GPU or the CPU in parallel.

Compute Device	CG, float	CG, double	BiCGStab, float	BiCGStab, double
Intel i7 960, single core	0.23	0.21	0.25	0.22
Intel i7 960, ViennaCL	0.73	0.44	0.52	0.33
NVIDIA Geforce GTX 470	**1.15**	**0.87**	**1.16**	**0.73**
ATI Radeon HD 5850	0.40	0.35	0.20	0.22

Table 3: Computational speed (in GFLOPs) for the CG and BiCGStab solvers without preconditioner. The 65 025 matrix rows have seven nonzero entries on average. Multiply-add operations are counted as single floating point operations.

8. CONCLUSIONS

The newly released open source library ViennaCL is presented in this work. It allows to use the huge computational resources of both GPUs and multi-core CPUs without going into the details of the underlying hardware. Thanks to a common programming interface with uBLAS, ViennaCL library users benefit on the one hand from the reuse of a widely accepted programming interface and on the other hand from the implementation of the three iterative solvers CG, BiCGStab and GMRES provided by ViennaCL, which can also directly be used with uBLAS types as well as with linear algebra types from other libraries using the generic wrappers provided. Benchmarks show that the library provides good performance on both GPUs and multi-core CPUs for large amounts of data. Performance gains of up to a factor of ten compared to a single CPU core can be observed for common linear algebra operations. Due to the use of OpenCL, ViennaCL can be run on many different parallel architectures.

9. ACKNOWLEDGEMENTS

Karl Rupp gratefully acknowledges support by the Graduate School PDEtech at the Vienna University of Technology. The authors wish to thank Prof. Siegfried Selberherr for providing a test platform for benchmarking and regression tests. This work has been supported by the European Research Council through the grant #247056 MOSILSPIN.

10. REFERENCES

[1] AMD Core Math Library for GPUs. http://developer.amd.com/gpu/acmlgpu/pages/default.aspx.

[2] BLAS homepage. http://www.netlib.org/blas/.

[3] Boost C++ Libraries. http://www.boost.org/.

[4] CULA. http://www.culatools.com/.

[5] Cusp. http://code.google.com/p/cusp-library/.

[6] MAGMA - Matrix Algebra on GPU and Multicore Architectures. http://icl.cs.utk.edu/magma/.

[7] Open Numerical Library. alice.loria.fr/index.php/software/4-library/23-opennl.html.

[8] uBLAS Library. http://www.boost.org/doc/libs/release/libs/numeric/ublas/.

[9] ViennaCL. http://viennacl.sourceforge.net/.

[10] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc Web Page. http://www.mcs.anl.gov/petsc/.

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

[12] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. *NVIDIA Technical Report NVR-2008-004*, 12, 2008.

[13] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.*, 22:917–924, July 2003.

[14] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[15] deal.II . http://www.dealii.org/.

[16] I. S. Duff, M. A. Heroux, and R. Pozo. The Sparse BLAS. *Technical Report TR/PA/01/24*, Sept. 2001.

[17] Getfem++. http://home.gna.org/getfem/.

[18] D. Göddeke, R. Strzodka, and S. Turek. Accelerating Double Precision FEM Simulations with GPUs. *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, 2005.

[19] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49, 1952.

[20] Khronos OpenCL. http://www.khronos.org/opencl/.

[21] Y. Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, April 2003.

[22] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

[23] H. D. Simon. The Lanczos Algorithm with Partial Reorthogonalization. *Mathematics of Computation*, 42(165):115–142, January 1984.

[24] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[25] Sundance. http://www.math.ttu.edu/ klong/Sundance/html/.

[26] H. A. van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Non-Symmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 12:631–644, 1992.

[27] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[28] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995.

[29] H. F. Walker and L. Zhou. A Simpler GMRES. *Numer. Linear Algebra Appl.*, 1(6):571–581, 1994.