Automatic Performance Optimization in ViennaCL for GPUs

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

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

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

ABSTRACT

Highly parallel computing architectures such as graphics processing units (GPUs) pose several new challenges for scientific computing, which have been absent on single core CPUs. However, a transition from existing serial code to parallel code for GPUs often requires a considerable amount of effort. The Vienna Computing Library (ViennaCL) presented in the beginning of this work is based on OpenCL to support a wide range of hardware and aims at providing a high-level C++ interface that is mostly compatible with the existing CPU linear algebra library uBLAS shipped with the Boost libraries. As a general purpose linear algebra library, ViennaCL runs on a variety of GPU boards from different vendors pursuing different hardware architectures. As a consequence, the optimal number of threads working on a problem in parallel depends on the available hardware and the algorithm executed thereon.

We present an optimization framework, which extracts suitable thread numbers and allows ViennaCL to automatically optimize itself to the underlying hardware. The performance enhancement of individually tuned kernels over default parameter choices range up to 25 percent for the kernels considered on high-end hardware, and up to a factor of seven on low-end hardware.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming —*Parallel programming*; D.2.2 [**SOFTWARE ENGINEERING**]: Design Tools and Techniques; D.3.2 [**PROGRAMMING LANGUAGES**]: Language Classifications—*C++*; G.4 [**MATHEMATICAL SOFTWARE**]: *Parallel and vector implementations*; I.3.1 [**COMPUTER GRAPHICS**]: Hardware Architecture—*Graphics processors*

General Terms

Scientific Software

Keywords

ViennaCL, OpenCL, GPU, C++

1. INTRODUCTION

The vast computing resources in graphics processing units (GPUs) became very attractive for general purpose scientific computing over the past years [9–11,13]. In the early days of general purpose computations on GPUs, the lack of double precision arithmetic was often considered a major drawback. However, recent GPUs such as NVIDIA Geforce GTX 470 or ATI Radeon HD 5850 do not suffer from this restriction any longer, thus they push into the field of high performance computing (HPC).

Considerable performance gains have been reported in former work [9–11,13], but the adaption of existing algorithms to GPUs requires a considerable amount of change in existing codes to account for the highly parallel architecture. Consequently, the effort required for porting an existing code to GPUs is sometimes 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 [4] or OpenCL [3], even if only standard linear algebra algorithms such as defined by the basic linear algebra subprograms (BLAS) [1] are to be executed on the GPU. To eliminate such an entry barrier, the Vienna Computing Library (ViennaCL) [6] was established, which provides BLAS routines using OpenCL on GPUs and multicore architectures with a high level C++ interface that hides parallelism details.

Unlike single core applications written in a high level programming language, where programmers do not have to care about hardware details, the proper choice of the best implementation for a particular GPU with optimal thread sizes adds another level of complexity for a high level abstraction library such as ViennaCL. Considering that GPU vendors offer a broad product range starting from a few up to several hundreds of processing units, a natural requirement on a general purpose linear algebra library for GPUs thus is to utilize the available units in the best possible way. The focus of this work is on how such an optimization environment can be provided with minimum effort for users of the library. Our approach is to perform separate optimization runs on the target machine, from which the best parameters are extracted and stored in a XML file. The optimized parameters can then be loaded into the main application and overwrite built-in default parameters. Such an optimization run is optional, so advanced users can get full performance, while first-time users do not have to care about this detail.

This paper is organized as follows: First, the basic philosophy of ViennaCL is presented in Sec. 2, where main features as well as basic design decisions are also discussed. Sec. 3 presents our approach for automated performance tuning. Unlike previous publications on general purpose calculations on GPUs, where kernels and kernel parameters have been tuned by hand for a particular target device, the optimization approach aims at optimizing each compute kernel in ViennaCL for the library user's target machine. Once the optimal parameter for the target machine are determined, they can be stored in a XML file and reused for subsequent uses of the library, which is discussed in Sec. 4. Sec. 5 quantifies the performance gain obtained from the presented optimization framework. Sec. 6 discusses possible future directions and a conclusion is drawn in Sec. 7.

2. THE VIENNA COMPUTING LIBRARY

In the following a brief overview of ViennaCL is given, which was first released in the end of May 2010. To account for the youth of ViennaCL, we give a short overview in this section. More details can be found in [15].

The programming language of choice for ViennaCL on the host is C++ due its wide availability and high abstraction capabilities. OpenCL [3] is chosen for low level hardware programming, because it supports a much larger variety of devices compared to for example CUDA [4].

A 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, so the user does not have to know details about the underlying hardware. Thus, ViennaCL does not require dedicated initialization routines such as for example the solver library PETSc [8].

There are many linear algebra libraries available in C++, one of the most commonly used is uBLAS included in the peer-reviewed Boost libraries [2]. 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.

The basic types used for linear algebra operations are the following in both uBLAS and ViennaCL:

```
scalar<NumericT> s; //scalar
vector<NumericT> v; //vector
matrix<NumericT> m; //dense matrix
compressed_matrix<NumericT> c1; //CSR
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. [16]), 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 is provided just as in uBLAS by overloaded operators in order to maximize readability of the source code. The following snippet gives a brief impression on how to use basic operations as defined on BLAS level 1:

```
// BLAS level 1
// x, y and z are vectors
y = 2.0 * x;
z = x + y;
x += 3.1415 * z;
NumericT n1 = norm_1(x);
NumericT n2 = norm_2(y);
NumericT ninf = norm_inf(z);
plane_rotation(x, y, n1, n2);
```

Temporary objects are eliminated to a large extent by the use of expression templates [20, 21] for operations such as (possibly scaled) additions and subtractions of two vectors, either in place or when assigned to a third vector. This avoids costly temporary memory allocations on the respective computing device.

On BLAS level 2, ViennaCL is also uBLAS compatible:

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

For dense matrices, the type of the variable `tag` is either `upper_tag`, `lower_tag`, `unit_upper_tag` or `unit_lower_tag` and is used to choose the dense, triangular linear solver.

BLAS level 3 as well as the BLAS levels defined for sparse matrices [12] are not fully implemented in the first release of ViennaCL, but are in preparation. An important sparse operation, namely sparse matrix vector products, is provided and serves as a building block for iterative solvers, of which a conjugate gradient solver (CG) [14] for symmetric positive definite systems, a stabilized bi-conjugate gradient solver (BiCGStab) [19] and a generalized minimal residual solver(GMRES) [17, 22] for indefinite systems are provided. An optional CPU-based ILUT preconditioner is also included. Since the ViennaCL API is essentially a subset of that of uBLAS, the iterative solvers can be used with objects from both libraries thanks to templates in C++.

Initialization of data on the device memory is a copy operation, thus ViennaCL reuses the conventions introduces with the Standard Template Library (STL) (see e.g. [18]). In order to copy the entries of a vector `cpu_vec` from the host to a vector `gpu_vec` in the GPU memory, the call

```
copy(cpu_vec.begin(),
     cpu_vec.end(),
     gpu_vec.begin());
```

is sufficient. In this way, also parts of a vector can be manipulated. Thus, programmers acquainted with the iterator concept and the STL can reuse their knowledge.

For data types where the iterator concept cannot be applied easily, a modified copy function is provided. For example, a matrix `cpu_matrix` on the host, no matter if dense or sparse, is copied to a matrix `gpu_matrix` on the device using

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

For this generic interface a number of type requirements such as iterator retrieval and entry access have to be posed on the `cpu_matrix` type in order to access the data. These requirements are chosen such that they are automatically fulfilled by the concepts modelled by uBLAS types.

3. KERNEL PARAMETER TUNING

The OpenCL specification decomposes each compute device into compute units and processing elements [3]. Each compute unit consists of at least one processing element. Processing elements within a compute unit can be synchronized, while compute units execute independently from each other. Compute units are an abstraction of a collection of processing elements in hardware and may, but do not have to, refer to a collection of processing units in hardware.

Threads executing a particular kernel on a device have to be grouped in compute units at kernel launch. The maximum number of processing elements per compute unit depends on the device and is typically 256, 512 or 1024, hence not equal to the physical number of processing elements. This allows the scheduler on the device to process a group of threads, say A, while another group of threads, say B, waits for data from the global device memory. As a consequence, the general advice of hardware vendors is to use large numbers of threads in order to get best performance. More precisely, the two parameters associated with an OpenCL kernel launch are the number of compute units and the number of processing elements per compute unit.

The simplest possibility for the determination of an optimal parameter set is to test all parameters from the discrete set $[1, \dots, N] \times [1, \dots, M]$, where N refers to a suitable upper limit for the number of compute units and M denotes the maximum number of processing elements per compute unit. Typical values of N are in the range 100 to 1000. Current GPUs report 256, 512 or 1024 processing elements per compute unit. Therefore, a brute force approach testing $N \times M$ parameter sets is impractical.

To account for the optimality of parallel reduction schemes for data that is a power of two, a rough overview can be obtained from benchmarking the set $[1, \dots, 2^n] \times [1, \dots, 2^m]$, where $n = \lfloor \mathrm{ld} N \rfloor$ and $m = \lfloor \mathrm{ld} M \rfloor$ with $\lfloor \cdot \rfloor$ denoting the round to next smallest integer operation. For reasons of uniform hardware utilization, the parameter for the number of compute units can also be chosen as a multiple of the physically available compute units. Hence, instead of testing powers of two, one may also test with first parameter $u, 2u, 4u, 8u, \dots$, where u denotes the number of compute units on the device.

Typically small numbers of processing elements per compute unit show poor performance and can be skipped. In principle one can selectively search for a better parameter set after the coarse sampling step, but the results in Sec. 5 suggest that an eventual performance gain is negligible and does not justify the additional benchmarking effort.

Such a search for the best parameter set can be carried out for each compute kernel, which is currently accomplished in dedicated programs. After moderate execution times in the range of several seconds for each optimization program, the best parameters obtained are written to a file. Other programs using ViennaCL can then read these optimized parameters. This file-based approach also allows to distribute the results of one or more optimization runs to other machines with similar hardware.

4. STORING AND LOADING PARAMETERS

ViennaCL internally associates compute kernels with the main data type for the computation. Thus, compute kernels are grouped as follows:

- *Scalars*: Compute kernels that operate on scalar arguments only
- *Vectors*: Compute kernels for vector operations
- *Matrix*: Compute kernels for dense matrix operations
- *Sparse*: Compute kernels for sparse matrix operations grouped by sparse matrix type.

Each group consists of 5 to 25 kernels and new kernels may be added in future releases. For a discussion of the just-in-time compilation of these kernels, the reader is referred to [15].

Good parameters for the number of compute units and the number of processing elements not only depend on the kernel, they also depend on the device and the numerical data type. To be able to store this hierarchy of dependencies, a XML data structure is used. A specialized XML tree setup is applied, which satisfies the aforementioned need to map the hierarchy of dependencies for the parameters on the data structure. Furthermore, additional meta-information can be added, for example, the total number of available compute units. For creating and accessing XML files by XPath [7], PugiXML has been used [5]. In the following, the device part is depicted, which contains information about the device to which the parameters are related to.

```
1  <parameters>
2    <devices>
3      <device>
4        <name>GeForce GTX 470</name>
5        <driver>195.36.31</driver>
6        <computeunits>40</computeunits>
7        <workgroupsize>512</workgroupsize>
8         ...
9      </device>
10     <device>
11       <name>Intel Core i7</name>
12       <computeunits>8</computeunits>
13        ...
14     </device>
15   </devices>
16 </parameters>
```

The presented XML format allows to store parameters for different devices (Lines 3-9, 10-14). Note that the parameters are stored in subtrees of the device nodes (Lines 8, 13). An example is given in the next snippet.

```
...
<optimizations>
  <optimization>
    <name>vector</name>
    <numeric>float</numeric>
    <kernels>
      <kernel>
        <name>add</name>
        <params>
          <param>
            <name>compute units</name>
            <value>128</value>
          </param>
            ...
        </params>
      </kernel>
    </kernels>
  </optimization>
</optimizations>
...
```

A set of nodes is wrapped by a corresponding set node. For example, the individual parameter subtrees are placed within an enveloping parameters node (Lines 9-15). This approach imposes a structuring of the data, which not only supports convenient data traversal but also maps the natural structure of the data on the XML data structure. Although this XML setup seems overloaded, it supports unique distinction of different parameter sets and especially the extendibility for new, not yet recognized, nodes or subtrees. Moreover, a collection of parameters for frequently used devices can be shipped with ViennaCL.

The XML data can be accessed by using XPath. For example, the following XPath expression returns the parameter subtree for a specific device.

```
"/parameters/devices/device[name='GeForce
    GTX 470']";
```

If an ViennaCL application loads a parameter file, it can be tested if a parameter set is available which fits the present device.

Another more specific access can be realized by accessing a certain numerical data type

```
"/parameters/devices/device[name='GeForce
    GTX 470']/optimizations/optimization[
    numeric='float']";
```

And finally, to access the values for a specific environment, the following expression may be used.

```
"/parameters/devices/device[name='GeForce
    GTX 470']/optimizations/optimization[
    numeric='float']/kernels/kernel[name='
    add']/params/param[name='work_groups']/
    value/text()";
```

With this line, the number of work groups is accessed for:

- a GeForce GTX 470
- a numerical data type float
- an *add* kernel

This XPath expression would, for example, return the value:

```
"128"
```

Apparently, applying XPath expressions allows to conveniently access the stored data within an extensively formulated XML data structure.

5. PERFORMANCE GAINS

The performance of the compute kernels in ViennaCL, version 1.0.5, has been compared on a Radeon HD 5850 from ATI and a GTX 470 from NVIDIA. Comments are also given on a low-end NVIDIA Geforce 8500 GT, but no detailed execution times are given for the latter. The test platform was a Intel Core i7 960 with 6 Gigabytes of random access memory, running a 64-bit Linux kernel. For the ATI GPU, the kernel version was 2.6.33 with GPU driver version 10.7 using Stream SDK 2.2. For NVIDIA GPUs, a kernel with version 2.6.34 and a GPU driver, version 195.36.24, was used.

In ViennaCL 1.0.x, a single parameter pair of 128 compute units and 128 processing elements per compute unit was chosen uniformly for all compute kernels, because no automated optimization environment for the individual optimization of parameters for each of the about 40 kernels was available. Compute kernels bound to a single compute unit due to synchronization requirements also use 128 processing elements. On less powerful GPUs we have observed that a number of processing elements per compute unit larger than 128 may fail, so the default parameter set is not only a tradeoff among the performance of compute kernels, but also a tradeoff with support for weaker hardware.

Even for operations such as the additions of two vectors, which can be parallelized trivially, the choice of parameters has notable impact on performance, see Tab. 1 and Tab. 2. On NVIDIA hardware, the best parameter pair leads to a performance gain of five percent over the default parameters $(128, 128)$. Taking the best parameter set obtained from NVIDIA hardware Tab. 1 for comparable ATI hardware fails, because on a Radeon HD 5850 only up to 256 threads per compute unit are possible.

The best parameter for the Radeon HD 5850 in Tab. 2 is $(64, 128)$, which is by twelve percent faster than the default parameter set. However, taking the best parameter set on the ATI hardware for the NVIDIA GPU, a runtime slightly below the default parameter set is obtained. Thus, even a simple operation such as vector addition shows a considerable dependence on the choice of the number of compute units and processing elements per compute unit.

	32	64	128	256	512
32	0.769	0.541	0.439	0.398	0.380
64	0.538	0.439	0.397	0.378	0.379
128	0.544	0.444	**0.396**	0.378	0.384
256	0.485	0.407	0.385	0.386	0.379
512	0.452	0.399	0.381	0.377	**0.376**

Table 1: Execution times (in milliseconds) for the addition of two vectors with 1 000 000 entries assigned to a third vector on a NVIDIA GTX 470 in single precision. The chosen number of compute units is given at the beginning of each row, the number of processing units per compute unit is given at the beginning of each column. The execution time for the fastest and the default parameter set are printed in bold.

	32	64	128	256
32	1.346	0.722	0.478	0.420
64	0.776	0.478	**0.419**	0.420
128	0.858	0.538	**0.475**	0.438
256	0.732	0.479	0.446	0.427
512	0.674	0.448	0.432	0.421

Table 2: Execution times (in milliseconds) for the addition of two vectors with 1 000 000 entries assigned to a third vector on a ATI Radeon HD 5850 in single precision. The chosen number of compute units is given at the beginning of each row, the number of processing units per compute unit is given at the beginning of each column. The execution time for the fastest and the default parameter set are printed in bold.

	32	64	128	256	512
32	0.564	0.317	0.198	0.149	0.153
64	0.320	0.199	0.146	**0.144**	0.149
128	0.322	0.206	**0.161**	0.149	0.153
256	0.264	0.179	0.152	0.150	0.165
512	0.246	0.171	0.151	0.160	0.201

Table 3: Execution times (in milliseconds) for the computation of inner products of two vectors with 1 000 000 entries assigned to a third vector on a NVIDIA GTX 470 in single precision. The chosen number of compute units is given at the beginning of each row, the number of processing units per compute unit is given at the beginning of each column. The execution time for the fastest and the default parameter set are printed in bold.

	32	64	128	256
32	0.632	0.584	0.410	0.328
64	0.584	0.410	0.329	**0.314**
128	0.595	0.380	**0.360**	0.333
256	0.514	0.388	0.343	0.332
512	0.481	0.373	0.343	0.341

Table 4: Execution times (in milliseconds) for the computation of inner products of two vectors with 1 000 000 entries assigned to a third vector on a ATI Radeon HD 5850 in single precision. The chosen number of compute units is given at the beginning of each row, the number of processing units per compute unit is given at the beginning of each column. The execution time for the fastest and the default parameter set are printed in bold.

	32	64	128	256	512
32	0.303	0.182	0.176	0.185	0.210
64	0.191	0.151	0.174	0.193	0.201
128	0.194	0.177	**0.195**	0.214	0.221
256	0.161	0.164	0.195	0.214	0.218
512	**0.145**	0.157	0.198	0.211	0.222

Table 5: Execution times (in milliseconds) for sparse matrix-vector multiplication using a sparse matrix with 65 025 rows and columns and seven entries per row on a NVIDIA GTX 470 in single precision. The chosen number of compute units is given at the beginning of each row, the number of processing units per compute unit is given at the beginning of each column. The execution time for the fastest and the default parameter set are printed in bold.

	32	64	128	256
32	0.508	0.325	0.259	0.252
64	0.324	0.262	0.256	0.249
128	0.357	0.290	**0.272**	**0.247**
256	0.307	0.264	0.256	0.248
512	0.282	0.255	0.258	0.253

Table 6: Execution times (in milliseconds) for sparse matrix-vector multiplication using a sparse matrix with 65 025 rows and columns and seven entries per row on a ATI Radeon HD 5850 in single precision. The chosen number of compute units is given at the beginning of each row, the number of processing units per compute unit is given at the beginning of each column. The execution time for the fastest and the default parameter set are printed in bold.

The best common parameter set in Tab. 1 and Tab. 2 is to be $(64, 256)$, which is almost optimal for both GPUs and also gives best performance on a NVIDIA Geforce 8500 GT. On this low-end board, execution times for vector addition show a similar picture and only four percent of performance can be gained by optimized parameters. Thus, taking vector addition alone, an optimization framework may not pay off compared to setting a common parameter set $(64, 256)$ on current hardware.

While entry-wise arithmetic operations on vectors all show a picture similar to that of vector addition, the situation is different for more complicated compute kernels such as inner products. Here, the input vectors are split into small blocks, where products are summed up by parallel reduction in on-chip shared memory and the intermediate results are then summed up to obtain the final result.

The results in Tab. 3 and Tab. 4 again identify a best parameter set $(64, 256)$, which is the same as for vector addition. The performance benefit over the default parameter set is eleven and thirteen percent respectively.

However, for the low-end NVIDIA Geforce 8500 GT, the best parameter set is $(2, 512)$ with an execution time of 1.062 milliseconds. The default parameter set takes 6.960, the set $(64, 256)$ takes 6.946 milliseconds. Thus, while the optimization run improved execution speed on high-end hardware by only slightly above ten percent, it reduced execution time on low-end hardware down to one seventh. It has to be noted that the physical number of compute units on the Geforce 8500 GT is two, which is just the best parameter for the compute units found in the optimization run.

Execution times for sparse matrix-vector products are compared in Tab. 5 and Tab. 6. The sparse matrix used for the benchmark has 65 025 rows and stems from a finite element discretization of the Laplace operator. On average, seven nonzero values in each row are present. The matrix is stored in a CSR-format [16], which is known not to give optimal performance on GPUs [9, 10], but is the fastest matrix type available in ViennaCL 1.0.x.

On a NVIDIA GTX 470, the best parameter set yields an improvement of 26 percent over the default parameter set. The set $(64, 256)$, which gave best performance on the previous benchmarks, gives an improvement of only one percent. Thus, even on a single high-end GPU board, one static parameter set does not yield good performance for all kernels.

The ATI GPU performance could be improved by nine percent compared to the default parameters. The parameter set $(64, 256)$ from the previous benchmark is again close to best performance, which is obtained for the set $(128, 256)$.

On the low-end NVIDIA Geforce 8500 GT, most parameter sets are within three percent of the best performance, indicating that the operation is severely memory bandwidth-limited on this GPU. Thus, a parameter optimization does not pay off in this case.

6. OUTLOOK

This work investigates the automated tuning of the number of compute units and processing elements per compute unit. Further performance gains can be obtained by comparing different implementations that e.g. make use of vector data types within the OpenCL kernels. The benefit of such vectorized types comes at the cost of additional effort needed for memory management and it strongly depends on the underlying hardware whether there is a performance gain at all.

The presented optimization functionality can also be applied directly to future many-core CPUs. Current CPUs with four to six cores lead to a small of reasonable parameter sets, but future CPUs are expected to have a much higher degree of parallelism.

Furthermore, it is likely that the optimal parameters also show a dependence on the provider of the OpenCL implementation. While it is unlikely that OpenCL implementations for GPUs from non-vendors will be available, several OpenCL implementations for mainstream CPUs could emerge.

7. CONCLUSIONS

Even though good performance can be achieved with the default parameter pair for the number of compute units and the number of processing elements per compute unit [15], it is shown in this work that the automated optimization framework scheduled for ViennaCL 1.1.0 leads to significantly improved execution performance ranging from a few percent to up to a factor of seven on a low-end GPU. Thus, even though ViennaCL is designed to be a general purpose linear algebra library, it now automatically adopts to the available target hardware to obtain the best possible performance for a bearable tuning time frame.

The parameter set $(64, 256)$ was found to yield better performance by around ten percent than the default parameter set $(128, 128)$. This is remarkable, since the total number of threads, given by the products of the two parameters, is the same for both sets.

The possibility to save the best parameter set for the device to an XML file allows a quick and simple adaption to the target hardware. Even if users decide not run the optimizer, the insights gained from the investigations presented in this work allow us to ship with better default parameters that are deduced from certain device characteristics.

Acknowledgements

Karl Rupp gratefully acknowledges support by the Graduate School PDEtech at the Vienna University of Technology. Moreover, the authors wish to thank Peter Lagger for his work-in-progress on a profiling application that greatly simplified the comparison of different parameters, and 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.

8. REFERENCES

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

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

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

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

[5] PugiXML. http://code.google.com/p/pugixml/.

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

[7] XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath/.

[8] 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.

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

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

[11] 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.

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

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

[14] 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.

[15] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *Proc. GPUScA*, pages 51–56, 2010.

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

[17] 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.

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

[19] 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.

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

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

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