Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1857 – 1866

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

High-Level Manipulation of OpenCL-Based Subvectors and Submatrices

Karl Rupp

Institute for Analysis and Scientific Computing, TU Wien
Institute for Microelectronics, TU Wien

Abstract

High-level C++ proxies for the convenient manipulation of subvectors and submatrices on OpenCL-enabled devices are introduced. It is demonstrated that the programming convenience of standard host-based code can be retained using native C++ language features only, even if massively parallel computing architectures such as graphics processing units are employed. The required modifications of the underlying OpenCL kernels are discussed and a case study of an implementation of the QR-factorization is given. Benchmark results confirm that the convenience of purely CPU-based libraries can be preserved without sacrificing performance of OpenCL-enabled devices, particularly graphics processing units.

Keywords: C++, OpenCL, QR factorization, ViennaCL

1. Introduction

The continued miniaturization of semiconductor devices, which is commonly expressed by Moore's law [1], has shown validity for over four decades already. However, the clock frequency on central processing units (CPUs) has essentially saturated in the early 2000s due to otherwise excessive power consumption, thus limiting a further scaling of sequential processing speed. As a consequence, additional transistors have mostly been used for duplicating CPU cores since then. The implications of this shift are a major concern for software development, since the efficient utilization of all available computational resources on a given machine now requires to write parallel programs, while formerly serial implementations have been sufficient. In contrast to a purely sequential processing, the challenges are manifold: On the algorithmic level, sufficient parallelization needs to be ensured. In particular, well-established sequential algorithms may be replaced by parallel variants, which may require additional floating point operations, yet they lead to shorter execution times on the overall due to a better hardware utilization. From the programmer's point of view, additional effort and familiarity with the pitfalls of parallel programming are required in order to synchronize threads and processes, and to avoid race conditions. This, in turn, typically leads to a larger code base, which becomes harder to maintain. Furthermore, old serial implementations no longer automatically scale with the increased computational power of newer hardware generations and may require a reimplementation if they are crucial for the overall performance of an application.

Email address: rupp@iue.tuwien.ac.at (Karl Rupp)

1877-0509 © 2012 Published by Elsevier Ltd.
doi:10.1016/j.procs.2012.04.204

Classical parallel programming approaches on CPUs include the multi-process oriented message passing interface (MPI), POSIX Threads for multi-threaded applications [2], and compiler-based techniques such as OpenMP [3] or Cilk [4, 5]. Recently, general purpose computations on graphics processing units (GPUs) have gained a lot of attention. Two frameworks for general purpose computations on GPUs are nowadays widely available: The compute unified device architecture (CUDA) [6] is a proprietary toolkit provided by NVIDIA and targets GPUs only. OpenCL is a royalty-free standard maintained by the Khronos Group [7] and supports both multi-core CPUs, GPUs and even special-purpose hardware like field programmable gate arrays (FPGAs) from different vendors. Development kits and runtime libraries are freely available from all major vendors. It is crucial to note at this point that CUDA relies on a separate compilation step prior to the execution of the program, while OpenCL provides a just-in-time compiler at run time for that purpose. Thus, OpenCL source code can in principle be generated on-the-fly during program execution and tailored to the hardware of the target machine by the just-in-time compiler. However, since OpenCL is designed for a wider range of hardware, several GPU-specific features (e.g. *warps*, available with CUDA) are not natively provided, which may result in lower performance of OpenCL kernels when compared to CUDA kernels.

Since many algorithms rely, at least partially, on linear algebra operations, many libraries have been made available over the years in various programming languages for CPUs. The most prominent example, LAPACK [8], is well tuned, yet vendor-specific libraries such as ACML [9], ESSL [10] or MKL [11] may provide considerably better performance for selected routines. While these libraries provide basic linear algebra subroutines (BLAS), high-level interfaces using syntactic sugar such as operator overloads are often preferred. Examples of C++-based libraries with such a high-level interface include Eigen [12], MTL [13] and Boost.uBLAS [14], which all rely on so-called expression templates [15, 16] in order to avoid spurious temporary objects from operator overloads.

General purpose computations on GPUs have been introduced just recently, therefore a relatively small number of libraries providing linear algebra operations on GPUs is available. MAGMA [17] is based on CUDA and provides BLAS routines for NVIDIA GPUs. Commercial libraries provide functionality via predefined functions only. Similarly, vendor-specific libraries (CUBLAS [18], APPML [19]) offering basic BLAS functionality are also available. High-level programming interfaces are provided by the CUDA-based Cusp [20] and the OpenCL-based ViennaCL [21], for which an overview is given in Sec. 2.

Thanks to powerful abstraction facilities, the high-level interface of CPU-based libraries hides internals of the underlying data type. For example, a dense matrix type is able to abstract the underlying row- or column-major memory layout, such that entries can be conveniently manipulated using the parentheses operator without runtime penalty. For GPUs, however, compute kernel launch times are in the range of 10 to 100 microseconds due to PCI-Express communication, thus abstraction facilities can only be applied to sufficiently large aggregates of operations. Reconsidering the example of dense matrices, abstraction facilities need to be provided for operations such as matrix addition or matrix multiplication rather than for individual matrix element access whenever reasonable performance is required. Therefore, a reevaluation of established abstraction techniques for CPU-based libraries is required for OpenCL-based applications.

Cusp and ViennaCL provide matrix types for which operators are overloaded suitably. However, objects are so far limited to the representation of a whole vector or a whole matrix respectively. For algorithms requiring the manipulation of subvectors and submatrices, representations of a whole vector or a whole matrix are inadequate. Prominent examples of such algorithms are Cholesky-, LU- and QR-factorizations for the solution of systems of linear equations, but also compression algorithms in e.g. image processing relying on singular value decompositions by keeping only the largest singular values and their associated singular vectors [22]. The lack of a suitable addressing of subvectors and submatrices consequently leads to unnecessary copies on the OpenCL device and may involve spurious host-device communication, thus degrading performance. To overcome these limitations, this work presents a high-level application programming interface (API) for manipulating subvectors and submatrices directly. The functionality is released with ViennaCL 1.3.0 and introduced in Sec. 3, where implications on the underlying OpenCL kernel sources and their performance are discussed. The new proxy objects are used in a case-study of QR-factorization using panel factorizations in Sec. 4. Benchmark results quantifying the obtained performance are given in Sec. 5. For the remainder of this work, OpenCL-enabled devices are identified with GPUs, even though also multi-core CPUs and other accelerators can be used with OpenCL. This simplifies a distinction between code and resources for the host and for OpenCL device.

2. Abstraction Techniques in ViennaCL

Because proxy objects presented in Sec. 3 are based on the abstraction techniques employed in ViennaCL, a brief overview of the library with focus on the abstraction techniques is given. Additional information including a full list of features can be found on the project webpage [21].

Consider a simple program which creates two vectors v_1 and v_2 of size `DIM`, fills v_1 with values, assigns $v_2 \leftarrow 2v_1$ and then prints both vectors. With typical CPU-based high-level linear algebra libraries, the necessary C++ code is in most cases similar to the following:

```
1 vector<double> v1(DIM), v2(DIM);              // instantiate
2 for (size_type i=0; i<DIM; ++i) v1[i] = i;    // fill
3
4 v2 = 2.0 * v1;                                // compute
5
6 std::cout << "v1: " << v1 << std::endl << ", v2: " << v2 << std::endl; //print
```

Note that each code line fully reflects one of these steps; no bloat with initialization or memory allocation routines occurs. From a user's perspective, it is desirable to keep this level of abstraction even if GPUs are used. Consequently, the API of ViennaCL is designed such that the code above is valid if directly placed inside the `main()` routine. Many details are dealt with in the background and are discussed next.

As soon as the first vector is instantiated in line 1, the OpenCL backend is configured automatically using a singleton pattern. Per default, a context consisting of the first device is created on the first platform returned by the OpenCL library. These default selections can be customized by suitable API calls prior to the first instantiation of a linear algebra object. Moreover, all OpenCL compute kernels related to pure vector operations (BLAS level 1) are compiled in a single program, because in all practical cases a vector object requests some of them at a later point. In principle, each kernel could also be compiled separately at the first use. However, it turned out that repeatedly launching the OpenCL just-in-time compiler introduces a lot of overhead, hence compiling all vector operation kernels at once is still significantly faster than compiling a relevant subset (e.g. one quarter of these kernels) individually.

The vector `v1` is filled with data in line 2 in the snippet above . The bracket operator is overloaded accordingly, such that for each entry in `v1` a separate data transfer is initiated. Compared to setting up a vector on the CPU, access to individual entries using OpenCL is by several orders of magnitude slower, cf. Sec. 5. Nevertheless, such an overload is handy for prototyping purposes or for the convenient manipulation of a few entries only.

Line 4 specifies the actual vector operation using operator overloads. Note that the high-level specification is not only shorter than a corresponding `for`-loop over all entries, but at the same time also aggregates the manipulation of all vector entries, thus one OpenCL kernel can be launched for the full operation rather than launching a kernel for each entry of `v1` and `v2` respectively. One subtlety requires additional attention: Overloading the multiplication operator for a scalar and a vector in a naive way, e.g.

```
1 vector<double> operator*(double val, vector<double> const & v) { ... }
```

recasts the statement $v_2 \leftarrow 2.0 * v_1$ into two operations $v_{\text{temp}} \leftarrow 2.0 * v_1$ and $v_2 \leftarrow v_{\text{temp}}$, where v_{temp} denotes a temporary object. While temporary objects can be expensive on CPU-based programs, they are extremely detrimental to performance on GPUs, since additional PCI-Express communication for the creation and the deletion of the temporary object as well as one additional compute kernel launch are required. For this reason, `operator*` is overloaded such that only a proxy object encoding the operation $2.0 * v_1$ is returned, but no calculation is carried out. The assignment operator for v_2 is in addition overloaded with respect to these proxy objects, where the operation is unwrapped and an OpenCL kernel for $v_2 \leftarrow 2.0 * v_1$ without temporary vector objects is launched. The technique follows the idea of *expression templates* [15], but requires some modifications due to the restricted set of available compute kernels. For example, the operation $v_1 \leftarrow v_2 + v_3 + v_4$ is not directly mapped to a single compute kernel, because only OpenCL kernels up to three vector operands including the result vector are provided. Thus, the expression template technique is modified such that a temporary is introduced in order to map the full operation onto a sequence of the functionality provided with the predefined OpenCL kernels. For the considered example, the operations $v_{\text{temp}} \leftarrow v_2 + v_3$ and $v_1 \leftarrow v_{\text{temp}} + v_4$ with temporary v_{temp} are carried out. In principle, it is also possible to create OpenCL compute kernels on-the-fly for arbitrarily complex vector operations [23], but this is beyond the scope of this discussion.

In addition to the `vector` type, a dense matrix type `matrix`, several sparse matrix types, and structured matrix types are provided by ViennaCL. Setting the entries of these types individually using the overloaded parenthesis operator is too costly unless only a few entries are affected, therefore the transfer from CPU to GPU is accomplished by a generic `copy()` function. The interface for the vector type mimics that of the C++ standard template library (STL) using the iterator concept:

```
viennacl::copy(input.begin(), input.end(), output.begin());
```

where `input` and `output` are the source and the destination vector objects and `begin()` and `end()` return iterators or pointers to the begin and the end of the underlying data container, respectively. For example, if `input` is a CPU-based type and `output` is a ViennaCL vector, an OpenCL transfer to the GPU is initiated. It is worthwhile to mention that the iterator-interface also allows for the transfer of partial data. For example, the line

```
viennacl::copy(input.begin() + 2, input.end() + 6, output.begin() + 1);
```

copies four entries with an offset of two from the source vector to the destination vector with an offset of one. For other ViennaCL types, an iterator-based `copy()` function is inadequate, because CPU-based types often do not provide appropriate functionality. Therefore, overloads of `copy()` using two arguments are provided:

```
viennacl::copy(input, output);
```

Since `copy()` is a free function, it can be overloaded with respect to its arguments, thus enabling generic wrappers for linear algebra types from other libraries. Moreover, the approach is non-intrusive, because there are no changes to the respective ViennaCL classes required, if wrappers for other libraries are to be added. In contrast to the iterator approach used for vectors, partial updates to e.g. a dense matrix using the `copy()` function are not directly possible, unless additional proxy types are introduced.

3. Proxy Objects

As has been discussed in the introduction, the manipulation of subvectors and submatrices is a frequent requirement in many linear algebra algorithms. With the functionality of ViennaCL presented in Sec. 2, only calculations using full matrices and vectors can be carried out. In this section the necessary extensions to the user API as well as the underlying OpenCL kernels are presented in order to enable calculations on subvectors and submatrices.

A first approach is to just provide conversion routines, which store subvectors and submatrices in new vector and matrix objects respectively. Calculations are then performed using these temporary objects, and results are written back to the original objects. As example, given matrices $\mathbf{A} = (\mathbf{A}_1, \mathbf{A}_2)$ and $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$, adding $\mathbf{B}_1$ to $\mathbf{A}_1$ would thus require temporary matrices `m_A_1` and `m_B_1`. These two temporary objects are then processed, and the updated `m_A_1` is written to the block A_1.

The advantage of this first approach is that only little functionality needs to be added, thus the effort for the library developers is small. However, library users are in such case required to first extract the subvectors and submatrices and store them in new objects, then carry out the actual operation, and finally ensure that the computed results are written back to the initial objects. Thus, instead of one single statement for the computation, up to three statements are required. Besides usability concerns, the temporary objects involved result in memory overhead. The implications on execution times can be considerable, if only simple operations such as matrix additions are performed, while the overhead is negligible for computationally demanding operations such as matrix-matrix products.

The disadvantage of temporary objects holding copies of the subvectors and submatrices can be avoided with the use of proxy objects. Here, proxy objects are objects which do not hold the actual data, but reference the respective containers and carry additional selection information. In the C++ community such proxy objects are also referred to as *view objects* [24]. First, two mechanisms for the specification of a subset of possible row- or column indices are provided similarly to Boost.uBLAS:

1. A `range(a,b)` refers to the set of integers in the half-open interval $[a, b)$ in ascending order, where the lower bound a is included, but the upper bound b is not.
2. A `slice(a,c,d)` refers to a set of integers starting at a, with nonzero increment c, and consisting of d integers.

Two examples demonstrate the use of these two types:

```
range r(3, 7);       // indices 3, 4, 5, 6
slice s(4, -2, 3)    // indices 4, 2, 0
```

Since every `range` can equivalently be written as a `slice`, it is sufficient to restrict the subsequent discussion to the `slice` type. In addition to `range` and `slice`, one may also allow for any arbitrary set of admissible indices. For this purpose a `vector` of integers can be used, therefore no additional type needs to be introduced. However, additional care is required for write operations, which are not well defined if an index appears more than once in such an index vector. As a consequence, the remaining discussion is mainly devoted to the handling of subvectors and submatrices specified by the `range` and `slice` objects.

The next step is to define suitable proxy objects for encoding subvectors and submatrices. Again, types are named in similarity to Boost.uBLAS: `vector_slice<V>` refers to a vector of type `V` (typically a `viennacl::vector<>`) consisting of the entries identified by the indices in the provided slice. Similarly, `matrix_slice<M>` represents the submatrix of a matrix of type `M` given by the entries obtained with a slice for the row indices and a slice for the column indices. As an example, the following code prints the first and the third entry of a vector and the first and the third row and column entries of a matrix:

```
slice s(0, 2, 2);                                 // instantiate the slice

vector_slice< vector<double> > vs(v, s);          // proxy for entries v_0 and v_2
std::cout << vs << std::endl;                     // print

matrix_slice< matrix<double> > ms(m, s, s); // proxy: m_00, m_02; m_20, m_22
std::cout << ms << std::endl;                     // print
```

The proxies `vs` and `ms` do not store their own entries, they rather hold references to the original objects `v` and `m`. Consequently, the proxy objects are only created on the CPU and no additional communication with the GPU is required at each instantiation.

Proxies can be manipulated in the same way as vectors and matrices. For example, the matrix-vector product of the above proxies can be computed using the straight-forward function overloads including expression templates for the elimination of spurious temporaries as

```
vector<double> result = prod(ms, vs);
```

The proxies can also serve as a left hand side operand in an assignment statement (*lvalue*):

```
vs = prod(ms, b);  //b is another vector or vector_slice
```

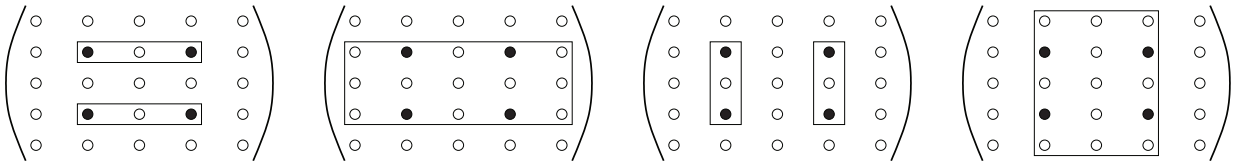
From a usability point of view, the separate instantiation of the proxy objects `vs` and `ms` can be tedious. A more compact notation is enabled by the free function `project()`, which is also included in Boost.uBLAS and takes the vector (or matrix) object as first argument and the proxy object(s) as second (and third) argument. This allows for a more compact representation of the previous snippets:

```
slice s(0, 2, 2);
vector<double> result = prod( project(m, s, s), project(v, s) );
project(v, s) = prod( project(m, s, s), b);
```

In addition to encoding linear algebra operations, which is analogous to the purely CPU-based Boost.uBLAS library, proxy objects can also be used for transfers between CPU host and OpenCL device. For example, copying the submatrix represented by the matrix slice `ms` back to CPU then reads

```
copy(ms, cpu_m);
```

where `cpu_m` is any CPU matrix of suitable dimensions and providing access to its entries using the parenthesis operator. In a similar fashion, selected entries of a matrix on the GPU can be filled with new values from a CPU-based matrix.



(a) Row major, one r/w transfer per affected row.

(b) Row major, one blocked r/w transfer.

(c) Column major, one r/w transfer per affected column.

(d) Column major, one blocked r/w transfer.

Figure 1: Four different memory transfer patterns for writing four entries (filled circles) in a 5×5-matrix with row or column-major memory layout.

In general, a slice does not cover all row and column indices of a matrix, therefore it is not possible to write all entries using a single memory transfer only. As already mentioned in Sec. 2, the transfer of individual entries between CPU and GPU is orders of magnitude slower than setting a single entry on the CPU, cf. Sec. 5. Consequently, it can be more appropriate to first copy a full row or column of a matrix from GPU to the CPU, update the entries there, and then copy the full row or column back to GPU. Moreover, for a slice with indices between a and b, where $|a - b|$ is small compared to the matrix dimensions, it can be even appropriate to use a single read transfer for all rows or columns with index between a and b, update the entries on the CPU and copy all data back to the GPU in one write transfer. The contiguous block of memory for the transfer is chosen with minimum size such that the first and the last entry are modified. An overview of these update patterns for row- and column-major matrices is given in Fig. 1, while benchmark results are postponed to Sec. 5.

Next, the implications of introducing proxy objects for the manipulation of subvectors and submatrices on OpenCL kernels are investigated. As an introductory example, one possible OpenCL kernel for the operation $v_1 \leftarrow v_1 + v_2$ is considered:

```
__kernel void add(__global       double * v1,
                  __global const double * v2, uint size){
  for (uint i = get_global_id(0); i < size; i += get_global_size(0))
    v1[i] += v2[i];                                                  }
```

The OpenCL function `get_global_id(0)` returns the global identifier of each thread, while `get_global_size(0)` returns the total number of threads [7]. In this example, only a one-dimensional execution model is chosen, to which the function arguments `0` for the two OpenCL functions refer. Despite the fact that the kernel fits its intended purpose, it is not sufficiently general to cover the case of subvectors v_1 and v_2. Therefore, the additional parameters required for the specification of a slice are now passed for each memory buffer to the kernel:

```
__kernel void add(__global double * v1, uint start1, uint stride1, uint size1,
            __global const double * v2, uint start2, uint stride2, uint size2){
  for (uint i = get_global_id(0); i < size1; i += get_global_size(0))
    v1[start1 + stride1 * i] += v2[start2 + stride2 * i];                  }
```

The additional parameters `start1`, `stride1` and `size1` refer to the slice specification for v_1, and similarly for the other parameters on v_2. Note that in this case `size1` and `size2` hold the same value, since v_1 and v_2 need to be of the same size. Even though one of the two parameters could thus be dropped, the use of a unified parameter set is beneficial for the maintainability of the kernels and simplifies an automatic creation of OpenCL kernels considerably [23]. A fifth parameter is used for a `vector` in ViennaCL in addition, namely the internal buffer length. It denotes the actual memory buffer size, which may be larger than the vector size specified by the user due to an optional internal padding with zeros in order to enable the use of vector data types in compute kernels. For example, a vector of length 30 may reside in a buffer of length 32 in order to enable the use of vector data types representing four entries. The following mapping is thus suggested: A `vector` is always mapped to a set of five parameters: The pointer to the raw memory, a start index, the stride, the vector length and the internal buffer size. Similarly, a `matrix` is mapped to nine parameters: The raw memory and four integers for the row and the column index specification, respectively. These parameters are again the start index, the stride, the number of elements, and the internal buffer length per row or column.

If arbitrary index vectors are used for addressing subvectors or submatrices, these vectors also need to be passed to OpenCL kernels. Since each of the indices of an index vector needs to be loaded from the GPU RAM, additional memory bandwidth is required. Thus, simple operations such as vector additions, which are commonly memory bandwidth limited, suffer from reduced performance. This is in contrast to the use of slices, where all index calculations are performed on-chip and no additional memory bandwidth is required. Similar reasoning applies to BLAS level 2 operations such as matrix-vector operations, which are typically memory bandwidth limited as well. Consequently, the use of slices does not have a notable impact on performance, while the use of index vectors leads to reduced performance due to the higher memory bandwidth requirements. For the compute-bound BLAS level 3 operations such as matrix-matrix multiplications, additional memory bandwidth requirements are less a concern. Instead, the increased set of OpenCL kernel parameters occupies additional shared memory, hence crucial cache block sizes may need to be reduced [25].

4. Case Study: QR Factorization

A study of the QR factorization using block Householder factorizations [26] is given in this section. The QR factorization is the key ingredient for the computation of eigenvalues using the QR algorithm and is usually the method of choice for the solution of least-squares problems [26]. For the implementations considered in this section, proxy objects as introduced in the previous section are extensively used in order to obtain a fast, yet easy to maintain, implementation. The following discussion can also be applied to Cholesky- and LU-factorizations, which are commonly implemented using similar delayed update strategies.

The QR factorization computes an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$, such that $\mathbf{A} = \mathbf{QR}$ for given $\mathbf{A} \in \mathbb{R}^{n \times n}$. For the sake of clarity it is assumed that $\mathbf{A}$ is square and has full rank. The repeated application of suitable Householder transformations $\mathbf{H}_i = \mathbf{I} - 2\mathbf{v}_i\mathbf{v}_i^T/(\mathbf{v}_i^T\mathbf{v}_i)$ with identity matrix $\mathbf{I}$ and Householder vector $\mathbf{v}_i$ zeroing all elements below the diagonal in column i leads to $\mathbf{H}_n\mathbf{H}_{n-1}\cdots\mathbf{H}_1\mathbf{A} = \mathbf{R}$. Since Householder matrices are orthogonal and symmetric, the orthogonal matrix $\mathbf{Q}$ is thus obtained as $\mathbf{Q} = \mathbf{H}_1\cdots\mathbf{H}_n$.

The Householder matrices $\mathbf{H}_1, \ldots, \mathbf{H}_n$ and thus $\mathbf{Q}$ are usually kept in an implicit form, thus only the Householder vectors $\mathbf{v}_i$ and the coefficients $\beta_i = 2/(\mathbf{v}_i^T\mathbf{v}_i)$ are stored. A direct implementation of the QR factorization using Householder reflections is rich in BLAS level 2 operations such as matrix-vector multiplications and outer-product updates. Using delayed updates, one can accumulate several Householder reflections $\mathbf{H}_r\cdots\mathbf{H}_s$, $r < s$, as $(\mathbf{I} + \mathbf{W}_{r,s}\mathbf{Y}_{r,s}^T) = \mathbf{H}_r\cdots\mathbf{H}_s$. The matrices $\mathbf{W}_{r,s}$ and $\mathbf{Y}_{r,s}$ are determined from the Householder vectors $\mathbf{v}_r, \ldots, \mathbf{v}_s$ with coefficients $\beta_r, \ldots, \beta_s$ by

$\mathbf{Y}_{r,s} := \mathbf{v}_r$; $\mathbf{W}_{r,s} := -\beta_r\mathbf{v}_r$
for $j = r+1 : s$
  $\mathbf{z} := -\beta_j(\mathbf{I} + \mathbf{W}_{r,s}\mathbf{Y}_{r,s}^T)\mathbf{v}_j$
  $\mathbf{W}_{r,s} := [\mathbf{W}_{r,s}\ \mathbf{z}]$; $\mathbf{Y}_{r,s} := [\mathbf{Y}_{r,s}\ \mathbf{v}_j]$
end

where the brackets denote a concatenation of the two arguments. Updates to $\mathbf{A}$ can now be computed as

$$(\mathbf{I} + \mathbf{W}_{r,s}\mathbf{Y}_{r,s}^T)^T\mathbf{A}(r:n, s+1:n) = \mathbf{A}(r:n, s+1:n) + \mathbf{Y}_{r,s}\mathbf{W}_{r,s}^T\mathbf{A}(r:n, s+1:n) \quad (1)$$

using BLAS level 3 operations, where $\mathbf{A}(r:n, s+1:n)$ denotes the submatrix of $\mathbf{A}$ starting at row r and column $s+1$.

Transferring the above algorithm to code is rather immediate using the proxies presented in Sec. 3. The required code for applying Householder reflections to columns r to s of a matrix object `A` consists of four steps.

1. Householder reflections are applied to the panel consisting of columns r to s of $\mathbf{A}$. For the sake of brevity, this rather technical (yet not performance critical) part is not shown here and may even be put in a separate compute kernel for a high-performance implementation. The computed Householder vectors are stored memory-efficiently below the diagonal in the respective columns of `A` and the coefficients β_i are assumed to be available in a vector `betas`.
2. The matrix $\mathbf{Y} \in \mathbb{R}^{n \times (s-r+1)}$ is set up. Note that the Householder vector v_i is normalized such that the i-th entry is equal to one. Moreover, since the Householder vectors are already stored in $\mathbf{A}$, only copy operations are required. Thus, the k-th column of $\mathbf{Y}$ is set up using

```
Y(r+k, k) = 1.0;
project(Y, range(r+k+1, n), range(k, k+1)) =
  project(A, range(r+k+1, n), range(r+k, r+k+1));
```

3. The columns of $\mathbf{W} \in \mathbb{R}^{n\times(s-r+1)}$ are computed one after another. While the first column is up to a factor $-\beta_1$ identical to that of Y, the k-th column of $\mathbf{W}$ is computed using the proxy objects along the lines

```
MatrixRange Y_old = project(Y, range(r, n), range(0, k));
MatrixRange v_k   = project(Y, range(r, n), range(k, k+1));
MatrixRange W_old = project(W, range(r, n), range(0, k));
MatrixRange z     = project(W, range(r, n), range(k, k+1));

YT_prod_v = prod(trans(Y_old), v_k);
z = - betas(r+k) * (v_k + prod(W_old, YT_prod_v));
```

   Here, `MatrixRange` refers to the type of the matrix range proxy, `trans()` denotes the transpose operation and `YT_prod_v` is a temporary vector used for storing $\mathbf{Y}_k^T\mathbf{v}_k$.

4. Finally, the update to the remaining columns of $\mathbf{A}$ with column index larger than s as in (1) is applied.

```
MatrixRange A_part(A, range(r, n), range(s + 1, n));
MatrixRange W_part(W, range(r, n), range(0, block_size));
MatrixRange Y_part(Y, range(r, n), range(0, block_size));

MatrixType WT_prod_A = prod(trans(W_part), A_part);
A_part += prod(Y_part, WT_prod_A);
```

   The variable `block_size` refers to the number $s - r + 1$ of columns processed within the delayed update.

In summary, the presented code lines cover the main operations of the QR algorithm at a high level of abstraction. Particularly, up to the namespaces of the the involved types, the code for the GPU-accelerated ViennaCL code is identical to that obtained when using the purely CPU-based Boost.uBLAS library. Conversely, the implementation of the QR algorithm can be used for both ViennaCL types and Boost.uBLAS types using suitable template arguments and type deduction facilities. Thus, it is demonstrated that low-level programming using OpenCL and high-level abstraction facilities provided by C++ blend well. The high-level interface of ViennaCL thus counters productivity concerns for GPU computing raised in the past [27].

5. Benchmark Results

A comparison of different copy strategies for submatrices discussed in Sec. 3 as well as execution times for the QR factorization from Sec. 4 are given. All benchmark results have been collected using `double` data types on a Linux machine equipped with an Intel Core i7 960 CPU and an NVIDIA Geforce GTX 470 graphics adapter with driver version 290.10. No device- or problem-size specific optimizations have been applied in order to reflect the out-of-the-box performance for a ViennaCL library user.

Submatrices of square matrices $\mathbf{A} \in \mathbb{R}^{n\times n}$ specified by the `slice s(n/4, 2, n/4)` are considered for different values of n in order to evaluate the overhead of copy operations. Without loss of generality, the matrices are row-major. The time required to copy the entries of `project(A, s, s)` from GPU memory to host memory, setting the respective entries to zero, and then to copy the affected entries back to GPU memory is recorded. Four different methods are compared in the benchmark: First, elements are set to zero one after another using the overloaded parenthesis operator. Second, rows addressed by the slice are transferred one after another to the CPU, where the respective entries of each row are manipulated, and then transferred back to the GPU. The third method considered in the benchmark is to copy all affected entries contained in the smallest contiguous piece of memory possible using a single transfer to the CPU. Updates are then applied on the CPU and the buffer is copied back to GPU RAM using again a single OpenCL transfer. The last method copies the whole matrix, updates the respective entries and then pushes the new data back to GPU.

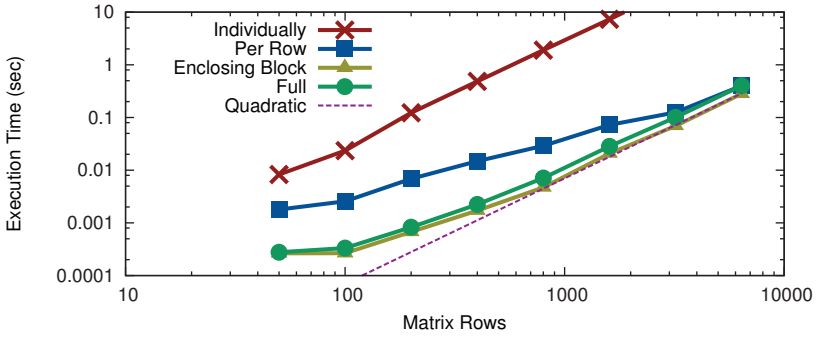
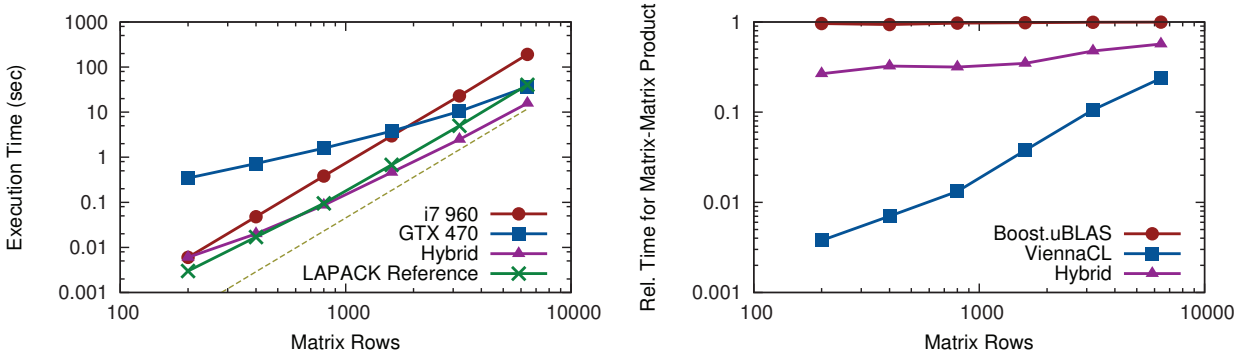


Figure 2: Comparison of execution times for the transfer of sliced submatrices. For comparison, quadratic complexity with respect to the matrix dimension is depicted with a dashed line.



(a) Total execution times.

(b) Relative amount of time spent on matrix-matrix multiplications.

Figure 3: Comparison of execution times for a QR-factorization using types from Boost.uBLAS only (single-threaded, CPU-based execution), types from ViennaCL only (GPU-based execution using OpenCL), a hybrid implementation using types from both libraries, and a single-threaded LAPACK reference. The same generic code base is used for the benchmarks.

Benchmark results are depicted in Fig. 2. Already at $n = 50$, copying entries individually is by a factor of five slower than copying each affected row one after another, which is again by a factor of six slower than copying the whole matrix in a single transfer. A saturation of execution times is observed for matrices matrices below 100×100, where OpenCL overhead dominates. In particular, from the total transfer time for the full matrix one can deduce a transfer overhead of slightly above $100\mu s$, which is independent of the transferred data. For matrix sizes above 100×100, setting entries individually is orders of magnitude slower than the other methods and thus not considered further. Copying individual rows is still by up to one order of magnitude slower than transferring all entries at once for matrix sizes up to 1000×1000. Beyond 1000×1000, row-based, block-based and full matrix transfer methods are comparable in execution times. It is worthwhile to note that a full transfer of the matrix requires twice as many data than a block-based transfer, still only a difference in execution times of a factor of 1.4 is obtained. From the execution time for the row-based method it is concluded that a single block transfer is preferred in general, unless only very local updates of the matrix are applied or the additional CPU RAM requirements are a concern.

An implementation of the QR-factorization considered in Sec. 4 is used for the second benchmark. The same generic code is used for the comparison of the single-threaded Boost.uBLAS implementation and a purely GPU-based implementation in ViennaCL. A hybrid implementation using Boost.uBLAS for the panel factorization and ViennaCL for BLAS level 3 operations is directly obtained by adding calls of `copy()` for the transfer of the panels of $\mathbf{A}$ as well as the matrices $\mathbf{Y}_{r,s}$ and $\mathbf{W}_{r,s}$ between host and device memory to the generic implementation. Suitable block sizes in the QR factorization of 20 for Boost.uBLAS and 10 for the GPU-assisted implementations have been determined by numerical experiments. No GPU-specific optimizations [28, 29] have been applied, because the focus of this benchmark is on a comparison of execution times obtainable from identical high-level implementations.

Benchmark results in Fig. 3 show that the generic implementation using Boost.uBLAS types is by a factor of 4.5 slower than the reference LAPACK implementation, which can be entirely attributed to the lower performance of matrix-matrix multiplications in Boost.uBLAS. The same implementation using ViennaCL objects suffers from excessive kernel launch overheads for problem sizes below 2000×2000, with less than five percent of the time spent on BLAS level 3 calls for smaller matrices. Above 2000×2000, kernel launch overheads diminish and smaller execution times than for Boost.uBLAS types are obtained. The performance of a LAPACK reference on the CPU is reached at a matrix size of 6400×6400, even though the GPU is shipped with reduced performance for `double` types. The hybrid implementation is by a factor of up to 4.5 faster than the LAPACK reference. Approximately half of the total execution time for the hybrid implementation is spent almost uniformly on BLAS level 3 operations. The high-level hybrid implementation is faster than the LAPACK reference already at a matrix size of 1000×1000.

6. Conclusion

The introduction of proxy objects for the manipulation of subvectors and submatrices essentially enables the convenience of scripting languages in C++ without sacrificing performance of the OpenCL device. This allows for user code to become considerably more compact and easier to maintain as compared to dealing with all low-level details of the OpenCL API and kernel language. The interface compatibility of ViennaCL and Boost.uBLAS allows for a careful prototyping of algorithms in a purely CPU-based environment using established debugging tools, and then to seamlessly switch to an OpenCL-based execution using ViennaCL. The hybrid implementation of the QR factorization case study shows an up to 4.5-times higher performance than a LAPACK reference even without applying any low-level tuning or sophisticated scheduling.

References

[1] G. Moore, Cramming more components onto integrated circuits, Electronics 38 (8) (1965) 56–59.
[2] POSIX.1c, Threads extensions, IEEE Std 1003.1c (1995).
[3] OpenMP, URL `http://openmp.org/`.
[4] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, in: Proceedings of the ACM SIGPLAN'95 Conference on Principles and Practice of Parallel Programming, 1995, pp. 207–216.
[5] Intel Cilk Plus, URL `http://cilk.com/`.
[6] NVIDIA CUDA, URL `http://www.nvidia.com/`.
[7] Khronos Group, OpenCL, URL `http://www.khronos.org/opencl/`.
[8] LAPACK library, URL `http://www.netlib.org/lapack/`.
[9] AMD core math library, URL `http://developer.amd.com/libraries/acml/`.
[10] IBM engineering and scientific subroutine library, URL `http://www.ibm.com/systems/software/essl/`.
[11] Intel math kernel library, URL `http://www.intel.com/software/products/mkl/`.
[12] Eigen library, URL `http://eigen.tuxfamily.org/`.
[13] MTL 4 library, URL `http://www.mtl4.org/`.
[14] Boost C++ libraries, URL `http://www.boost.org/`.
[15] T. Veldhuizen, Expression templates, C++ Report 7 (5) (1995) 26–31.
[16] D. Vandevoorde, N. Josuttis, C++ Templates, Addison-Wesley, 2002.
[17] MAGMA library, URL `http://icl.cs.utk.edu/magma/`.
[18] NVIDIA CUBLAS library, URL `http://developer.nvidia.com/cublas/`.
[19] AMD accelerated parallel processing math libraries, URL `http://developer.amd.com/libraries/appmathlibs/`.
[20] Cusp library, URL `http://code.google.com/p/cusp-library/`.
[21] ViennaCL library, URL `http://viennacl.sourceforge.net/`.
[22] H. Andrews, C. I. Patterson, Singular value decomposition (SVD) image coding, IEEE Transactions on Communications 24 (4) (1976) 425 – 432.
[23] P. Tillet, K. Rupp, S. Selberherr, An automatic OpenCL compute kernel generator for basic linear algebra operations, 2012, in press.
[24] G. Berry, C++ view objects, in: Dr. Dobb's, 2006, URL `http://drdobbs.com/cpp/196513737`.
[25] NVIDIA OpenCL best practices guide, URL `http://www.developer.nvidia.com/opencl/`.
[26] G. H. Golub, C. F. Van Loan, Matrix Computations, John Hopkins University Press, 1996.
[27] R. Bordawekar, U. Bondhugula, R. Rao, Can CPUs match GPUs on performance with productivity? Experiences with optimizing a FLOP-intensive application on CPUs and GPU, Technical report, IBM T. J. Watson Research Center (2010).
[28] Nath, R. and Tomov, S. and Dongarra, J., An improved MAGMA GEMM for Fermi graphics processing units, International Journal of High Performance Computing Applications 24 (4) (2010) 511–515.
[29] A. Kerr, D. Campbell, M. Richards, QR decomposition on GPUs, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009, pp. 71–78.