An Automatic OpenCL Compute Kernel Generator for Basic Linear Algebra Operations

Philippe Tillet[1], Karl Rupp[1,2], Siegfried Selberherr[1]
[1] Institute for Microelectronics, TU Wien
Gußhausstraße 27-29/E360, 1040 Wien, Austria
[2] Institute for Analysis and Scientific Computing, TU Wien
Wiedner Hauptstraße 8-10/E101, 1040 Wien, Austria
`Phil.Tillet@gmail.com`, {`rupp|selberherr`}`@iue.tuwien.ac.at`

Keywords: Automatic Code Generation, Linear Algebra, High Performance Computing, OpenCL, ViennaCL

Abstract

An automatic OpenCL compute kernel generator framework for linear algebra operations is presented. It allows for specifying matrix and vector operations in high-level C++ code, while the low-level details of OpenCL compute kernel generation and handling are dealt with in the background. Our approach releases users from considerable additional effort required for learning the details of programming graphics processing units (GPUs), and we demonstrate that higher performance than for a fixed, predefined set of OpenCL compute kernels is obtained due to the minimization of launch overhead. The generator is made available in the Vienna Computing Library (ViennaCL) and is demonstrated here with the stabilized bi-conjugate gradient algorithm, for which performance gains up to a factor 1.6 are observed.

1. INTRODUCTION

The emergence of general purpose computations on graphics processing units (GPUs) has lead to new programming approaches, which require a reconsideration of algorithm design as well as program flow. While a purely CPU-based program on a single machine directly operates on data in random access memory (RAM), which is managed by the operating system, a GPU-based function (denoted as *compute kernel* in the following) first and foremost requires the data as well as the executable to be available on the respective device. The compute kernel is then launched and manipulates the data on the GPU, where typically hundreds of threads operate simultaneously. Further compute kernels can be loaded and launched on the GPU for additional independent data manipulation. Finally, the result data is copied back from GPU RAM to CPU RAM, where the CPU-based execution can continue to process the results. In order to obtain a performance gain over a purely CPU-based execution, all data transfer and program launch overhead plus the execution time on the GPU must be smaller than the time required in the CPU-based execution.

For programming GPUs, basically two approaches are in wide-spread use in the scientific community: The Compute Unified Device Architecture (CUDA) from NVIDIA [1], which is a vendor-specific environment for NVIDIA hardware, and the Open Computing Language (OpenCL) [2], which is an open industry standard for program execution across heterogeneous platforms. Even though both the CUDA and the OpenCL language are derived from the C programming language, they require to learn a new language and to get acquainted with the details of the underlying hardware. This is certainly not desirable from an abstraction point of view.

In many application scenarios, the time required for transferring data from CPU RAM to GPU RAM is negligible compared to the accumulated execution time of all compute kernels. This is mostly due to the high speed of the PCI-Express link, which typically provides a bandwidth of 8 gigabytes per second (GB/s) in Version 2, and a bandwidth of 16 GB/s in Version 3. For example, the whole GPU RAM can thus be written and read from the host within one second for most current GPU boards. In contrast, the overhead for launching a compute kernel can accumulate to significant values, if many compute kernels with very short execution times are involved. Our measurements confirm the kernel launch overhead in the range of 10 to 100 microseconds specified by the vendors, which is a long latency considering that clock frequencies are in the range of gigahertz. In practice, kernel launch overheads are about one order of magnitude smaller, because the launch of a kernel can already be prepared while another kernel is as yet active. Still, with an internal memory bandwidth of, say, 160 GB/s of current mid- to high-end GPU boards, the overhead is in the order of the execution time of a single vector accumulation using two vectors with around 10 000 entries. Consequently, the execution time of compute kernels should be sufficiently large in order to keep any compute kernel startup overhead negligible.

For linear algebra operations, the OpenCL-based Vienna Computing Library (ViennaCL) [3, 4] provides a high-level application programming interface comparable to the purely CPU-based UBLAS library shipped with the Boost libraries [5]. A similar high-level approach for CUDA hardware is provided by Cusp [6]. The library MAGMA [7] is also based on

CUDA and aims at providing fast BLAS routines for dense matrices using multi-core CPUs and NVIDIA GPUs.

Unlike purely CPU-based BLAS implementations, where function calls have negligible costs, the overhead of OpenCL compute kernel launches becomes significant for most vector operations defined at BLAS Level 1, and for operations on BLAS Level 2, if the operands are of small dimensionality. For example, the assignment $\mathbf{x} \leftarrow \mathbf{a} + \mathbf{b} + \mathbf{c}$ for vectors $\mathbf{x}$, $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ can be computed in principle by the use of a single loop. Using BLAS Level 1 routines, three calls of e.g. the `*AXPY` subroutine are required, each looping over all elements of the operands. Similarly, ViennaCL requires the same number of compute kernel launches and may, depending on how the user writes the code, even create a temporary object, which is even more detrimental to performance than it is on CPU-based programs. As a remedy, the user could provide a custom compute kernel for the vector addition example, but this is not desirable from a usability point of view.

In this work we present an OpenCL compute kernel source generator for linear algebra operations defined at BLAS Level 1 and Level 2. Level 3 is not considered, because kernel launch overhead is negligible, when compared to execution times obtained already at small dimensions [8]. Our generator is based on a C++ domain specific embedded language for the generation of OpenCL kernels defined by high-level specifications provided by the user. Any kernel launch overhead for simple operations such as $\mathbf{x} \leftarrow \mathbf{a} + \mathbf{b} + \mathbf{c}$ is removed to the highest extent possible. The generator is presented in Sec. 2. and the benchmark results in Sec. 3. clearly show the efficiency of the generator at small to moderate problem sizes. An outlook to further applications is given and a conclusion is drawn in Sec. 4.

2. THE GENERATOR

Before going into the details of the generator, a glimpse of the operator overloads in ViennaCL is given. In order to instantiate three vectors in GPU RAM, to add two of them and to store the result in the third vector, it is sufficient to write

```
1   viennacl::vector<double> x, y, z;
2   /* Fill vectors with data here */
3   x = y + z;
```

Consequently, computations on the GPU can be carried out at a level of convenience comparable to purely CPU-based libraries such as Boost.UBLAS. Internally, operators are overloaded using so-called expression templates [9], by which the arithmetic expression on the right hand side is transformed into an expression tree at compile time. The benefit of such an approach is the possibility to minimize the demand of temporary objects, while the price to pay is increased compilation time, if expression templates and other metaprogramming techniques are used excessively. Due to the fixed set of OpenCL kernels shipped with ViennaCL, the expression tree is only generated up to the complexity covered by the OpenCL kernels. If the vector expression is more complicated, temporaries are introduced.

In the context of the generator, the expression template technique is used to encode the full calculation specification. Instead of actually performing the calculation, the respective OpenCL kernel is generated at runtime, which can then be used to launch the kernel with appropriate arguments. Similar functionality could in principle be achieved by pure runtime programming techniques, yet the functional flavor of template metaprogramming simplifies evaluations and manipulations of the expression trees significantly.

For the specification of the kernel operations, separate classes are provided for representing vector arguments:

```
1 symbolic_vector<double, 0> symb_x;
2 symbolic_vector<double, 1> symb_y;
3 symbolic_vector<double, 2> symb_z;
```

The objects `symb_x`, `symb_y`, `symb_z` refer to the first three vector operands with double-precision floating point entries of the resulting OpenCL compute kernel. Arithmetic operators for the `symbolic_vector` are then overloaded as usual in the expression template setting. Binary operators are encoded by the `compound_node<LHS, OP, RHS>` class, which takes the type of the left hand side operand as the first template argument, the operation encoded by a suitable tag as the second argument, and the type of the right hand side operand is the third template argument. For example, the type of the statement `symb_y + symb_z` evaluates to

```
1 compound_node< symbolic_vector<double,1>,
2                add_type,
3                symbolic_vector<double,2> >
```

where `add_type` is a tag class referring to the addition operation. The resulting nested type hierarchy of the statement `symb_x = symb_y + symb_z` is depicted in Fig. 1.

In order to create an OpenCL kernel from the expression tree, the nested types are unwrapped recursively. The OpenCL kernel name is deduced by a textual representation of the encoded operation, which is obtained from a recursive iteration over all nodes in the tree. For `symb_x = symb_y + symb_z`, the resulting kernel name is `vec0_eq_vec1_plus_vec2`. The second step is to deduce the function arguments, for which purpose all leaf nodes (i.e. all operands) of the expression tree are collected in a typelist [10] consisting of distinct types only. Each of the `symbolic_vector` classes then leads to one kernel argument referring to the OpenCL memory buffer of the vector and to an integer kernel argument referring to the number of entries of the vector. The generated function head for the vector addition thus is

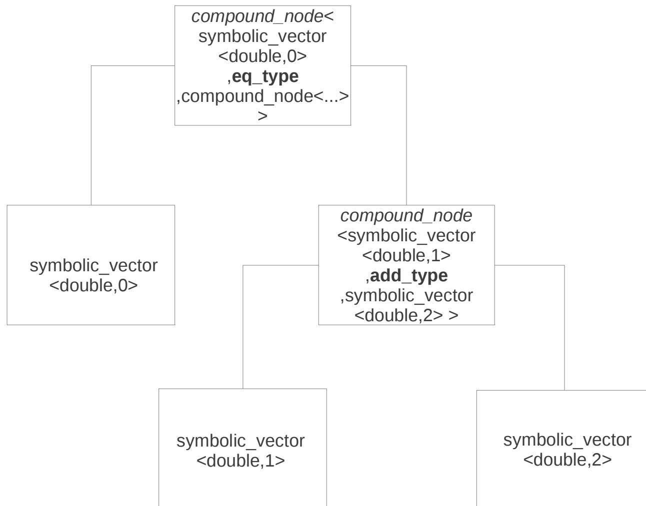

Figure 1. Expression tree for the calculation specification `symb_x = symb_y + symb_z`.

```
__kernel void vec0_eq_vec1_plus_vec2(
  __global double* vec0, uint size_vec0,
  __global double* vec1, uint size_vec1,
  __global double* vec2, uint size_vec2
)
```

In a similar manner, the kernel arguments for symbolic scalars and matrices are determined: A scalar leads to one argument, which is passed by value, if it originates from CPU RAM, or a pointer if passed from GPU RAM. A matrix leads to five arguments, namely a pointer to its first element, two arguments for the number of rows including possible alignment of the internal memory buffer to powers of two, and analogously two arguments for the number of columns.

In the OpenCL kernel body the worker threads are distributed over all entries in the result vector. Then, the expression tree is unwrapped and the code in the function body is generated. Overall, the resulting code in the OpenCL kernel body is

```
{
  for ( uint k  = get_global_id(0);
             k  < size_vec1;
             k += get_global_size(0) )
  { vec1[k] = vec2[k] + vec3[k]; }
}
```

Now as the kernel generation in the background is completed, we can now turn to the discussion of the user interface. The purpose of the kernel is to be launched for one or several sets of function arguments, hence it takes the role of a function object (functor). Consequently, the user interface is designed as follows: The symbolic expression is provided as constructor argument to the class `custom_operation`:

```
symbolic_vector<double, 0> symb_x;
symbolic_vector<double, 1> symb_y;
symbolic_vector<double, 2> symb_z;

custom_operation
 my_op(symb_x = symb_y + symb_z );
```

This triggers the kernel source code generation and compilation within the OpenCL just-in-time compiler while `my_op` is created. The kernel is executed by passing the kernel arguments to the parentheses-operator:

```
// Execute for vectors x, y, z:
viennacl::ocl::enqueue(my_op(x, y, z));
```

Here, `x`, `y` and `z` refer to the plain `viennacl::vector<>` objects introduced in the beginning of this section. The three vectors are internally mapped to two OpenCL kernel arguments each, namely the OpenCL memory handle and the vector size. The `enqueue()` function adds the operation to an OpenCL command queue. In the above case no queue is specified, hence the default queue is used.

The basic example of vector addition has been used to demonstrate the main steps during the OpenCL kernel generation process. Arbitrary additions, subtractions, and multiplications of vectors by scalars can be tackled in the same way. However, additional complications arise as soon as dot-products and matrix-vector multiplications are included. Consider a slight modification of the previous vector addition example, where here and in the following we do not distinguish between a symbolic specification, or an actual computation statement:

```
 x = inner_prod(y, z) * (y + z);
```

Due to the inherent serial nature of the inner product, it is for large vectors most efficient to split it into several chunks $\mathbf{x}_i$ and $\mathbf{z}_i$, $i = 1, \ldots, N$, where N denotes the number of work groups and is typically a small power of two, e.g. 32. Each chunk is computed using parallel reduction inside a work group and the intermediate results of each work group are summed in a second kernel to obtain the final result. Therefore, the resulting expression tree is first scanned for operations which need to be carried out in one or more separate kernels prior to the final assignment. It is important to mention that our generator is able to fuse the final summation from the inner product into the next kernel. More precisely, instead of three kernels for the operations:

- In work groups i, compute `inner_prod(y_i, z_i)`, $i = 1, \ldots, N$. Store the result in a temporary array `t`.
- Sum up the entries in `t` to obtain the final result `alpha` of `inner_prod(y, z)`.
- Compute `x = alpha * (y+z)`.

Our kernel generator fuses the second and the third kernel in order to avoid any kernel launch overhead resulting from the quite simple second kernel.

We proceed with an operation involving a dense matrix A, and now consider the slightly more complicated operation:

```
x = y - alpha*(prod(A,y) - z)
```

Here, prod() denotes the matrix-vector product. The straightforward approach to the computation of this expression using BLAS functionality is the computation of a temporary vector t = prod(A,y) − z, and then to compute z = y− alpha*t. However, as already mentioned, temporary objects on the GPU should be avoided whenever possible. In order to generate efficient kernels for the above operation, our generator decomposes the initial operation into an equivalent set of elementary tokens. In the above example two tokens are obtained which refer to the operations

```
x = y + alpha*z
x -= alpha*prod(A,y)
```

Note that no temporary vector is required in this case. The code for the kernel body is then generated by processing the tokens in the semantically correct order.

The first step of the tokenization process is to expand the encoded expression into tokens which are connected either by addition or subtraction. For vectors or matrices $\mathbf{x}, \mathbf{y}$, and a scalar value α, every sub-tree corresponding to an operation of the form $\alpha * (\mathbf{x} + \mathbf{y})$ (resp. $(\mathbf{x} + \mathbf{y}) * \alpha$) is transformed to $\alpha * \mathbf{x} + \alpha * \mathbf{y}$ (resp. $\mathbf{x} * \alpha + \mathbf{y} * \alpha$). This results in two tokens $\alpha * \mathbf{x}$ and $\alpha * \mathbf{y}$. In summary, the recursive expansion of the expression tree is accomplished by the following algorithm:

Procedure 1 Expand(): Expansion of an expression tree

Input: compound_node<L, OP, R>
Output: compound_node<L_out, OP_out, R_out>
  OP_MULT=scalar_mult
  if IsScalMul(OP) **and** IsAddOrSub(R) **then**
    L_out=Expand(compound_node<L, OP_MULT, R::L>)
    OP_out= R::OP
    R_out=Expand(compound_node<L, OP_MULT, R::R>)
  else if IsScalMul(OP) **and** IsAddOrSub(L) **then**
    L_out=Expand(compound_node<L::L, OP_MULT, R>)
    OP_out= L::OP
    R_out=Expand(compound_node<L::R, OP_MULT, R>)
  else
    L_out = Expand(L)
    OP_out = OP
    R_out = Expand(R)
  end if

Here, scalar_mult refers to multiplication by a scalar, ::L and ::R refer to the left and right hand side operands of the expression, ::OP refers to the operator token, IsScalMul() is true, if the supplied operator refers to multiplication by a scalar, and IsAddOrSub() returns true, if the supplied expression is a addition or subtraction.

After the expansion procedure, the expression tree is transformed into a sequence of tokens using typelists [10], for which correct signs must be preserved. For example, the expression $\mathbf{x} - (\mathbf{y} - \mathbf{z})$ becomes $\mathbf{x} - \mathbf{y} + \mathbf{z}$, while $(\mathbf{x} - \mathbf{y}) - \mathbf{z}$ does not require sign changes. The resulting list of tokens is then scanned for dependencies among the tokens. If there are no dependencies, such as in the first example $x = y + z$, the list of tokens directly leads to the body of the **for**-loop over all entries in the result vector or matrix. Inner products are always computed in a separate kernel, while matrix-vector products only lead to a temporary object, if one of the operands depends on the left hand side of the assignment statement.

Another frequent requirement for linear algebra algorithms are element-wise modifications. This requirement is addressed by our generator using the meta-object `elementwise_modifier`. In order to release the user from some implementation details required for setting up the meta-object by hand, a convenience macro is provided. For instance, an elementwise modifier F for modifying all entries x_i of a scalar, vector or matrix to $1/(1+x_i)$ can be declared as follows:

```
VIENNACL_EW_MODIFIER(F, "1/(1+X)")
```

It is also possible to combine `elementwise_modifiers` with each other.

A schematic of the final generator is given in Fig. 2. It should be emphasized that basically all complexity is hidden from the user, only the mnemonic specification of the kernel action at a high level of abstraction is required.

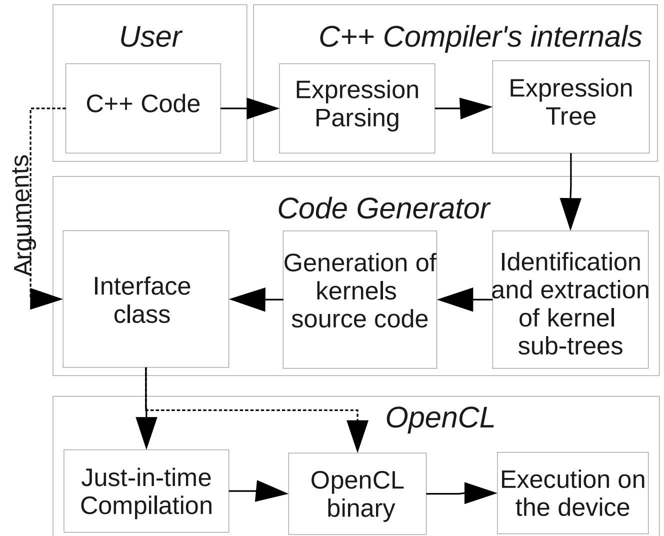

Figure 2. General execution model of the generator.

3. EXAMPLES/RESULTS

In this section we consider a synthetic BLAS Level 1 example demonstrating the impact of kernel launch times for BLAS Level 1 functions, a BLAS Level 2 example consisting of a matrix-vector product with nested vector operations, and the application of our generator to the iterative solver BiCGStab. The benchmarks are carried out in single precision on a Linux machine equipped with an NVIDIA Geforce GTX 470, driver version 285.05.09. We present execution times rather than the typically employed Gigaflops, because the kernel launch overhead cannot be suitably reflected in the latter.

The BLAS Level 1 example we consider in our first test case is

$$\mathbf{x} += (\alpha + \beta) * \mathbf{x} - (\mathbf{y} - F(G(\mathbf{z}))) ,$$

where the element-wise modifiers F and G are given by $(F(\mathbf{x}))_i = (1+x_i)^{-1}$ and $(G(\mathbf{x}))_i = x_i^2$. Note that **x**, **y** and **z** can either be all vectors, or all matrices. The required C++ code is:

```
1  //The elementwise modifiers
2  VIENNACL_EW_MODIFIER(F, "1/(1+X)")
3  VIENNACL_EW_MODIFIER(G, "X*X")
4
5  //Instantiation of the symbolic variables
6  symbolic_vector<NumericT,0> sX;
7  gpu_symbolic_scalar<NumericT,1> sAlpha;
8  gpu_symbolic_scalar<NumericT,2> sBeta;
9  symbolic_vector<NumericT,3> sY;
10 symbolic_vector<NumericT,4> sZ;
11
12 //Creation of the custom operation
13 custom_operation example
14   (sX += (sAlpha + sBeta)*sX
15          - (sY - F(G(sZ))));
16
17 //Execution of the custom operation
18 enqueue(example(x,alpha,beta,y,z));
```

NumericT denotes the underlying floating point type (**float** or **double**).

Benchmark results are depicted in Fig.3. An implementation using three BLAS Level 1 calls and one separate kernel for the element-wise modification is taken as reference. Since the operation is memory bandwidth limited, the reference implementation using four kernels requires four times the execution time than the single kernel generated by our kernel generator. Execution times start to saturate for vectors below a size of about 100.000, and a kernel launch overhead of about 30 microseconds is identified.

As our second example, we consider the operation

$$\mathbf{x} = \mathbf{A} \times \left[(\mathbf{y} \cdot (\mathbf{y} + \mathbf{z}))\mathbf{y} + \mathbf{z} \right] ,$$

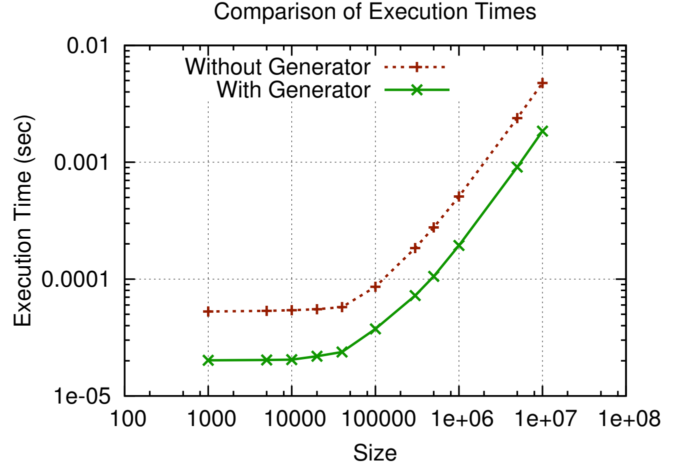

Figure 3. Performances of the generator on BLAS Level 1.

where **x**, **y** and **z** denote vectors, **A** is a dense matrix and the dot denotes the inner vector product. With the proposed generator it is sufficient to write the following C++ code:

```
1  // Instantiation of the symbolic variables
2  symbolic_vector<NumericT,0> sX;
3  symbolic_matrix<NumericT,1> sA;
4  symbolic_vector<NumericT,2> sY;
5  symbolic_vector<NumericT,3> sZ;
6
7  //Creation of the custom operation
8  custom_operation example2
9   (sX = prod(sA, inner_prod(sY, sY+sZ) * sY
10                + sZ));
11
12 //Execution of the custom operation
13 enqueue(example2(x,A,y,z));
```

Only two kernels are generated: The first is responsible for computing the inner product, while the second sums the intermediate results from the inner product calculation chunks and computes the matrix-vector product.

The benchmark results in Fig. 4 clearly show that the generator is superior for matrix sizes below 1000×1000. The difference of a factor of about 1.5 at a matrix size of 100×100 can make a considerable difference in real-time scenarios. As the dimensions of **A** increase to higher values, the BLAS Level 2 operation becomes dominant and the savings at BLAS Level 1 become negligible.

Finally, the generator is applied to the implementation of the non-preconditioned stabilized bi-conjugate gradient algorithm (BiCGStab) [11]. The generic implementation included in ViennaCL 1.1.2 is taken as a reference, which uses BLAS Level 1 functions for the vector updates and does not create hidden temporary objects. Tests have been carried out using matrices from the discretization of the Poisson equation in two spatial dimensions using piecewise linear finite elements

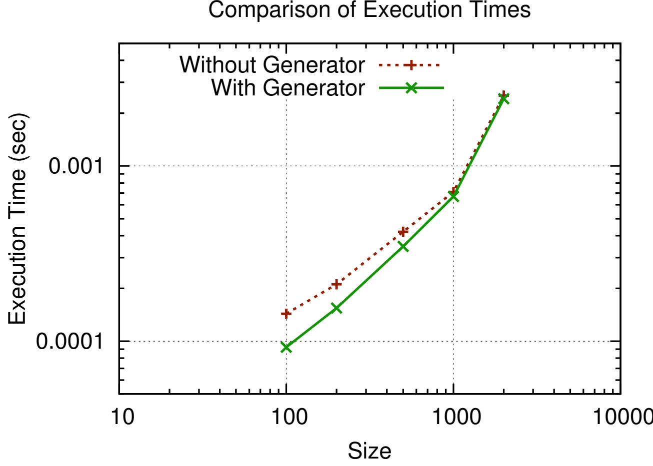

Figure 4. Performances of the generator on BLAS Level 2.

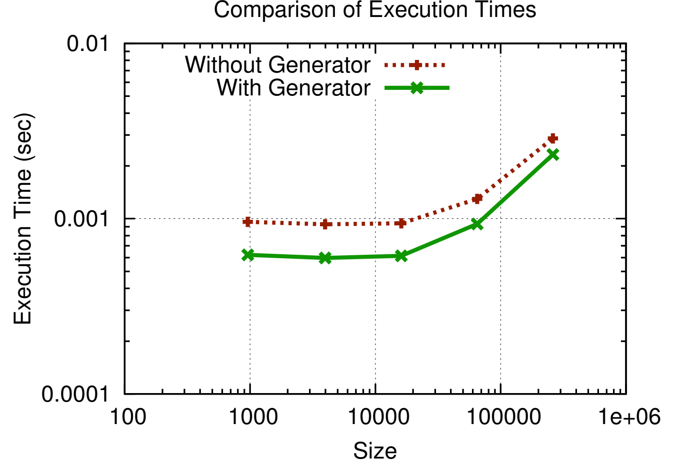

Figure 5. Execution time for a single BiCGStab iteration.

on a triangular grid. On average, there are five nonzero entries per row of the system matrix. Note that the performance of BiCGStab in terms of execution time heavily depends on the sparsity pattern of the system matrix and may thus be the limiting factor already at low matrix dimensions. The results have therefore to be taken with a grain of salt, yet they provide a realistic benchmark for many two-dimensional finite element simulations in practice.

The iteration loop of BiCGStab can be written as follows, where checks for convergence are omitted for brevity:

Procedure 2 BiCGStab loop

Input: Matrix $\mathbf{A}$, vectors $\mathbf{b}$, $\mathbf{r}$, $\mathbf{p}$, scalar ρ_0, int n
Output: Result vector $\mathbf{x}$

1: **for** $i = 1 \to n$ **do**
2: $\mathbf{t} = \mathbf{A} \times \mathbf{p}$
3: $\alpha = \rho_{i-1}/(\mathbf{t} \cdot \mathbf{b})$
4: $\mathbf{s} = \mathbf{r} - \alpha \mathbf{t}$
5: $\mathbf{v} = \mathbf{A} \times \mathbf{s}$
6: $\omega = (\mathbf{v} \cdot \mathbf{s})/(\mathbf{v} \cdot \mathbf{v})$
7: $\mathbf{x} = \mathbf{x} + \alpha \mathbf{p} + \omega \mathbf{s}$
8: $\mathbf{r} = \mathbf{s} - \omega v$
9: $\rho_i = \mathbf{r} \cdot \mathbf{b}$
10: $\beta = \frac{\rho_i}{\rho_{i-1}} \frac{\alpha}{\omega}$
11: $\mathbf{p} = \mathbf{r} + \beta * (\mathbf{p} - \omega * \mathbf{t})$
12: **end for**

Since sparse matrix-vector products are not included in the generator, we apply the generator to the vector operations only. One custom operation was applied to each of the Lines on 3, 4, 6, 7, 9 and 11, with Line 8 reusing the kernel from Line 4. The simple operation in Line 10 is evaluated on the CPU after the convergence check.

In Fig. 5 one can clearly see that in our test case BiCGStab is limited by OpenCL kernel launch overheads up to systems of about 100 000 unknowns. A performance gain of up to 40 percent is obtained with our kernel generator for smaller matrices. We expect that similar results hold true for other Krylov methods such as the conjugate gradient algorithm or the generalized minimal residual method [12].

4. OUTLOOK AND CONCLUSION

The new C++11 standard provides additional support for metaprogramming, which gives additional convenience to the user. Most notably, the **auto**-keyword hides the template hierarchy from the user:

```
1 auto op1 = inner_prod(sX + sZ, sX - sZ);
2 auto op2 = inner_prod(sY + sZ, sY - sZ);
3 custom_operation example
4  (sX = (op1 + op2)*sY)
5 enqueue(example(x,y,z))
```

Apart from higher convenience, the generator could also be extended for multiple assignment statements. Consider the plane rotation as defined at BLAS Level 1

$$\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \alpha & \beta \\ -\beta & \alpha \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} .$$

In order to avoid a temporary vector, the OpenCL kernel has to process both vector updates at the same time. Setting for simplicity $\alpha = \beta = 1$, a natural extension of the existing user interface is:

```
1 custom_operation rot
2  (sX = sX + sY,
3   sY = -sX + sY);
```

However, severe complications arise in the case where inner products and matrix products are considered, because the handling of data dependencies becomes much more involved than for the single operation case.

In summary, the role of OpenCL compute kernel launch overheads was studied in this work and a generator for linear algebra operations at BLAS Level 1 and Level 2 was presented. We have further shown that the use of modern C++ techniques provides a convenient user front-end, which abstracts the details of the underlying hardware without sacrificing performance and which invalidates productivity concerns related to GPU computing [13]. In addition, our benchmark results show that BLAS Level 1 routines on GPUs are limited by kernel launch overheads up to vector sizes of about 100 000, hence BLAS Level 1 operations should be fused with other kernels in order to reduce this overhead. The only drawback is the increased compilation time if the generator is used excessively. However, in most application scenarios it is reasonable to expect that only a few custom kernels are in use, thus compilation times are not a concern.

REFERENCES

[1] NVIDIA CUDA. `http://www.nvidia.com/`.

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

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

[4] K. Rupp *et al.*, ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. *Proceedings of GPUScA*, p. 51-56, (2010).

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

[6] Cusp Library. `http://code.google.com/p/cusp-library/`

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

[8] R. Nath *et al.*, An Improved Magma Gemm For Fermi Graphics Processing Units. *Intl. J. HPC Appl.*, vol. 24 no. 4, p. 511-515, (2010).

[9] T. Veldhuizen. “Expression Templates”. *C++ Report*, vol. 7, p.26-31 (1995).

[10] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Longman Publishing Co., Inc. (2001)

[11] 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 J. Sci. Stat. Comput.*, vol. 12, p. 631–644 (1992).

[12] Y. Saad, *Iterative Methods for Sparse Linear Systems, Second Edition*, SIAM (2003).

[13] R. Bordawekar *et. al.*, *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).