Expression Templates and OpenCL

Uwe Bawidamann and Marco Nehmeier

Institute of Computer Science, University of Würzburg
Am Hubland, D 97074 Würzburg, Germany
nehmeier@informatik.uni-wuerzburg.de

Abstract. In this paper we discuss the interaction of expression templates with OpenCL devices. We show how the expression tree of expression templates can be used to generate problem specific OpenCL kernels. In a second approach we use expression templates to optimize the data transfer between the host and the device which leads to a measurable performance increase in a domain specific language approach. We tested the functionality, correctness and performance for both implementations in a case study for vector and matrix operations.

Keywords: GPGPU, OpenCL, C++, Expression templates, Domain specific language, Code generation

1 Introduction

In the last years the computer architecture has changed from a single-core design to a multi-core architecture providing several processor cores on a single CPU. This new way of processor design has the intention to circumvent the physical constraints of increasing the performance of a single-core CPU by using symmetric multi-core processors to parallelize the computation [4].

Additionally the GPU has recently come into focus for general purpose computing by the introduction of CUDA (Compute Unified Device Architecture) [16] as well as the open standard OpenCL (Open Computing Language) [8] to exploit the tremendous performance of highly parallel graphic devices.

CUDA is NVIDIA's parallel computing architecture on GPU's which could be used for GPGPU (General Purpose Computing on Graphics Processing Units) through CUDA C [15], CUDA Fortran [20] or OpenCL [18].

OpenCL is an open standard for general purpose parallel programming across CPU's, GPU's and other processors [8]. It provides a C like programming language to address the parallel concept of heterogeneous systems. In contrast to CUDA C, which is the common programming language for CUDA devices, OpenCL is not limited onto a specific GPU architecture. OpenCL is rather available for NVIDIA CUDA GPU's [18], multi-core CPU's and the latest GPU's from AMD [2], Intel Core CPU's [7], DSP's [8] and many other architectures.

Preprint. The final publication is available at link.springer.com
http://dx.doi.org/10.1007/978-3-642-31500-8_8

Both technologies, CUDA as well as OpenCL, have a huge impact onto the world of scientific computing. At the moment the fastest super computer, the Chinese Tianhe-1A system at the National Supercomputer Center in Tianjin, reaches 2.57 petaflop/s by using a heterogeneous architecture of 7168 NVIDIA M2050 GPU's and 14,336 Intel Xeon CPU's [21]. But also scientists without an access onto a super computer as well as the normal user could benefit from the tremendous performance of GPU's in their workstation and desktop computers.

Alongside the use of standard applications like Mathematica [27], Matlab [11] or the operating system Mac OS X 10.6 [3] which support CUDA or OpenCL, users are able to write their own applications running on GPU's. But this is burdened with many new concepts and techniques which are necessary to address the parallel programming on GPU's or heterogeneous systems. The programmer has to take care about the data transfer between the host system and the CUDA or OpenCL device, he has to organize the thread local or shared memory as well as the global memory and he has to use specific parallel programming techniques [14, 17].

All these new concepts and techniques could be a real challenge for unexperienced developers in the area of GPGPU. In the worst case the program on a GPU has a inferior performance compared to a plain old sequential approach on a CPU caused by the bottleneck of inappropriate data transfers between host and device, misaligned memory access and bank conflicts, inappropriate load balancing, deadlocks and many more.

Hence, our intention was to investigate techniques and concepts to provide user-friendly libraries with an interface completely integrated into C++ hiding all the GPGPU specific paradigms from the user. Additionally the C++ concept of operator overloading offers the possibility to integrate the library interface as a domain specific language (DSL). Furthermore expression templates [23] are used to optimize the data transfer between the host and the device.

In this paper we used OpenCL as the hidden layer under the C++ interface using operator overloading and expression templates to utilize the benefits of GPU's as well as of multi-core CPU's. As an application area we chose matrix and vector operations which are particularly suitable to show the capability of a domain specific language embedded into C++ using expression templates and OpenCL.

2 Expression Templates

One of the outstanding features of C++ is the feasibility to overload several operators like the arithmetic operators, the assignment operator, the subscript operator or the function call operator. This offers a developer of particularly mathematical data types to implement the interface of this types as a domain specific language embedded into C++. But besides the possibility to define an easy and intuitively usable data type, the operator overloading in C++ has a drawback called pairwise evaluation problem [25]. This means that operators in C++ are defined as unary or binary functions. For expressions with more

than one operator like `r = a + b + c` this has the consequence that `a + b` are evaluated first and than their result, stored in a temporary variable, is used to perform the second operator to evaluate the addition with the variable `c`. Obviously this leads to at least one temporary variable for each `+` operator but if the data types are e.g. vectors it also performs several consecutive loops to perform the particular vector additions. Both the temporary variables as well as the consecutive loops are a performance penalty compared to a hand coded function computing the result with a single loop and no temporaries [25, 22], see Listing 1.

```
for (int i = 0; i < a.size(); ++i)
    r[i] = a[i] + b[i] + c[i];
```

Listing 1. Hand coded and optimal computation of `r = a + b + c`.

Expression templates [23] are a C++ technique to avoid this unnecessary temporaries as well as the consecutive loops by generating an expression tree composed of nested template types. Furthermore this expression tree is explicitly visible at compile time and could be evaluated as a whole resulting in a similar computation as shown in Listing 1. This could be achieved by defining a template class `BExpr<typename T, class OP, typename A, typename B>` as an abstract representation of an binary operation[1] specified by the template parameter `OP` as an exchangeable policy class [1] working on the arguments of the types `A` and `B` which are stored as reference member variables.

```
template<typename T, class OP, typename A, typename B>
class BExpr {
    A const& a_;
    B const& b_;
  public:
    BExpr(A const& a, B const& b) : a_(a), b_(b) { }

    T operator[] (size_t i) const {
        return OP.eval(a_[i], b_[i]);
    }
    ...
};
```

Listing 2. Class `BExpr<typename T, class OP, typename A, typename B>`.

Thereby the template type `T` specifies the type of the vector elements which is important to implement the element-wise evaluation using the subscript operator. In principle the subscript operator uses the policy class specified by the template parameter `OP` to compute the $\mathtt{i}^{\mathrm{th}}$ element of the result of the operation. Listing 3 shows an implementation of a policy class for a vector addition.

```
template<typename T> struct Add {
```

[1] Types for operations with a different order are defined in a similar manner. Furthermore we have shown in [12] how a general type for operations of a arbitrary order could be specified using a C++0x.

```
    static T eval(T a, T b) { return a + b; }
};
```

Listing 3. Policy class `ADD<typename T>`.

With these building blocks it is quite easy to overload the arithmetic operators for vector and scalar types to implement the operations like the vector sum or the multiplication with a scalar returning an expression tree. Figure 1 shows the composed tree structure for the expression `Scalar * (Vector + Vector)`. Therefore expression templates are a technique to compose expression trees with nested template classes at compile time.

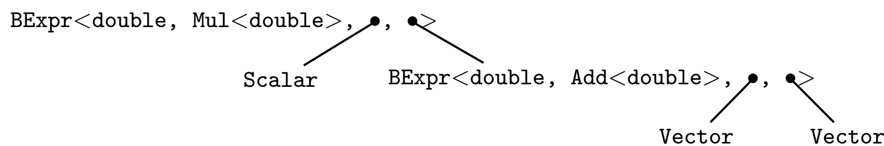

Fig. 1. Tree structure for the expression `Scalar * (Vector + Vector)`.

The evaluation of such an expression tree then could be performed as a whole by overloading the assignment operator. For the example with vector operations it is quite easy by using one loop over the size of the vectors computing the result element-wise by calling the overloaded subscript operator of the tree[2].

Obviously there are a lot of small objects and inline functions used for the expression template approach but due to modern compilers this evaluation leads almost to a similar machine code as Listing 1.

Using expression templates for vector operations to avoid unnecessary loops and temporaries is the classical example used in [23] to introduce these powerful technique. Besides loop fusion the expression templates are used in many other areas like the adaptation to grids in finite element methods [6] or to increase the accuracy in dot product expressions [9].

In addition we are working in the area of interval arithmetic and primarily used expression templates to reduce the necessary but time-consuming rounding mode switches of the floating point unit to speed up the computation [13]. Because this showed promising results we investigated an optimized expression template approach and combined it with additional functionality like the automatic differentiation [12] at compile time using template meta programming [24].

[2] Note that for this approach it is required that the `Vector` class as well as the `Scalar` class overload the subscript operator. For an `Scalar` it could easily return the value of the scalar.

3 Implementation

Recently we started to investigate the usage of GPU's for reliable computing and especially interval arithmetic. Hence, the tremendous computing power of GPU's motivated us to research the potential of a GPU as a back end for a C++ interval library which is based on expression templates and template meta programming. Our goal in this context is to offer the user a domain specific language for interval computations which is able to utilize GPU's for specific algorithms or problem solvers. The aim of this work was to have a case study for the combination of expression templates and GPU's.

We chose OpenCL in place of CUDA as the framework for our case study because it offers the possibility to generate and compile the OpenCL code at run time and additionally it is possible to run it on different GPU's as well as on CPU's.

As a field of application we chose vector and matrix operations for the case study and we implemented two different approaches. The first one is to use expression templates to generate specific OpenCL code for the expressions. The second one uses precompiled OpenCL kernels which are utilized by the expression templates. Both approaches are compared against an implementation using standard operator overloading to realize the domain specific language in C++ and computing the result on the GPU.

3.1 OpenCL code generation

The approach of generating specific OpenCL kernels with expression templates needs to address three different problems. The transfer of the data between the host and the device, the unique mapping between the data and the kernel parameters and the generation of the kernel itself.

In a domain specific language approach there are two different possibilities to realize the transfer of the data onto the device. The first one is to allocate and fill the `cl::Buffer` on the device at the creation of the vector or matrix. For this, the two OpenCL memory flags `CL_MEM_READ_WRITE` and `CL_MEM_COPY_HOST_PTR` are used to initialize a read-write buffer with data from the host. The second one is to allocate the buffer and copy the data within the assignment operator prior to the execution of the kernel. Both of them have their advantages and disadvantages. The first one has to copy the data only once but the required memory, which is limited on a GPU, is used all the time and the access of the data from the host is more costly. The second one has to copy the data for each expression but it only uses the required memory for the ongoing expression and the access from the host is easy.

For the unique mapping of the data and the kernel parameters we used the singleton pattern [5] to implement a class to generate unique id's which are requested at the construction of the vectors and matrices and stored as the member `id_`. Additionally the vector and matrix classes provide two methods `getIdent` and `getCode` which return the string `‘‘v’’ + id_` as an identifier and the string `‘‘v’’ + id_ + ‘‘[index]’’` as the code snippet to access the data.

Both methods are required to generate the specific OpenCL kernels out of the expression tree inside the assignment operator and to access the required data. For scalar types it is not required to return the identifier. In this case the method `getCode` returns the value of the scalar itself which is included as a constant into the kernel code.

The generation of the kernel itself is then subdivided into three parts. The first one is the computation of all required parameters. For this task an instance `paramSet` of the STL [19] container type `std::set` is recursively filled with all vectors or matrices of the expression[3].

Then afterwards this `paramSet` could be used to generate the header of the kernel while iterating over the set declaring the parameters of the expression, see Listing 4. Additionally a constant for the index of the work item is declared.

```
std::string code = "__kernel void CLETGenVectorKernel( ";
for ( it = paramSet.begin() ; it != paramSet.end(); it++ ) {
        code += "__global float* " +
              (*it).getObject().getIdent() += ", ";
}
code += " const unsigned int size )" +
        "\n{ \n" +
        " const int index = get_global_id(0); \n";
```

Listing 4. Generation of the kernel header and the constants.

The last but most important part is to generate the body of the kernel itself. This is done by calling the method `getCode` for the result type as well as for the expression tree, see Listing 5.

```
code += " " + result.getCode() + " = " +
        expression.getCode() + ";\n};\n \n";
```

Listing 5. Generation of the kernel body.

Obviously the method `getCode` is a replacement of the subscript operator of the expression template implementation introduced in Section 2. Hence the nodes of the expression tree used for the code generation, which are almost similar to the class `BExpr` in Listing 2, have to implement the method `getCode`. But this is quite easy by using specific policy classes to concatenate the strings of the recursive call of the child nodes with the required arithmetic operator, see Listing 6.

```
std::string getCode() const {
        return std::string("(" + a_.getCode() +
                " + " + b_.getCode() + ")" );
 }
```

Listing 6. Method `getCode` of a tree node for the addition.

Subsequently the generated kernel string could be compiled and executed using the OpenCL API functions inside the assignment operator.

[3] The vectors or matrices are stored as a constant reference in a wrapper class to afford a sorted organization of the `paramSet`. The method `getObject` offers access to the stored object.

3.2 Utilize precompiled OpenCL kernels

In addition to the implementation in Section 3.1 to generate specific kernels out of the expression tree we inspected the use of expression templates in place of operator overloading to minimize the data transfer of a domain specific language using precompiled kernels on a GPU back end. The aim of this case study is to realize a DSL where some parts of the computation, e.g. vector or matrix operations, are executed on a GPU without the requirement to transfer the data between the host and the device manually by the user. A classical approach using operator overloading suffer from the pairwise evaluation problem, see Section 2, where data is transferred from the host to the device, then the result is computed on the GPU and transferred back onto the host for every single operation. With expression templates the unnecessary data transfers can be eliminated.

The implementation of the expression templates are almost similar to the approaches in Section 2 and Section 3.1 respectively. The most important difference is that in addition to the leaf nodes (vectors, matrices ...) of the expression tree all inner nodes of the tree are annotated with unique id's. These id's are required to map the several operations of the expression, performed by operation specific precompiled kernels, onto the temporaries used for the results of the operations.

The evaluation of the expression tree is as well performed inside the assignment operator. But in contrast to the implementation in Section 3.1 there is no OpenCL code generated. Rather two traversals of the expression tree are required.

The first traversal is used to allocate all the necessary memory for the leaf nodes (vectors, matrices ...) as well as for the temporary results of the operations of the inner nodes on the OpenCL device. For each leaf node with a new id a `cl::Buffer` is allocated and initialized from the host. Additionally the id of the leaf node as well as the associated `cl::Buffer` are stored in a instance `paramMap` of the STL container `std::map` [19]. This approach has the benefit that the data of a leaf node is transferred only once even if the same associated variable/parameter is used multiple times in an expression. For inner nodes of the tree the required memory is only allocated for the temporary results of the related kernel of the operation. Certainly these allocated memory areas are stored in the `paramMap` together with their id's.

The second traversal of tree performs the computation of the expression on the OpenCL device by using the method `compute`[4] of the nodes recursively. This method could be subdivided into two parts. First of all the recursive calls of the method `compute` of the child nodes take place to evaluate the subexpressions. Afterwards the operation specific parts are performed, which could be also implemented with policy classes to generalize the code of the implementation, see Section 2. These parts are the determination of the required memory areas for

[4] This is the correspondent method to the subscript operator or the method `getCode` of the implementations in Section 2 or Section 3.1, respectively.

the parameters[5] and the result of the operation using the `paramMap` and the id's of the nodes as well as the execution of the operation specific precompiled kernel.

After the second traversal the evaluation of the expression is finished and the result is stored in the associated memory area of the root node of the tree. These data is transferred back to the host and the allocated device memory is freed.

4 Experimental Results

We have tested our two implementations on a AMD Radeon HD 6950 GPU.

For the code generation approach (Section 3.1) it has been shown that for standard vector operations the compile time for the generated kernel is out of scale for most of the problems. For example the time required for the expression `v1 + v2 + v1 * 5` is shown in Table 1. In this comparison the code generation approach as well as the operator overloading approach use device memory which is allocated and initialized during the construction of the vectors. Hence only the time for the compilation as well as for the execution of the kernel is measured. In contrast the operator overloading approach uses precompiled kernels.

Table 1. Performance comparison of the code generation approach (Section 3.1).

Vector size	Code generation	Operator overloading
4096	63 ms	1,2 ms
1048576	63,2 ms	1,3 ms
16777216	64,8 ms	5,7 ms

However, if we don't regard the compile time, we have measured an execution time of 3 ms for a vector size of 16777216 which is almost the half of the execution time with operator overloading. Hence, it could be a good choice for harder problems. Since expression templates are explicit types only one compilation of identical expression trees is required. Another application is to mix in the problem specific code into preassembled problem solvers, e.g. mix in the computation of a function and their derivative, using automatic differentiation, into a preassembled interval Newton method.

On the other hand we have inspected the approach using precompiled kernels against an implementation using operator overloading for matrix operations. For this case study the data of both approaches are kept on the host and are only transferred for the execution of the expression template or for the particular operation, respectively. Table 2 shows the required time for the execution of three different expressions for a matrix size 2048×2048. Note that for the second expression where the matrix `M1` occurs two times the expression template approach

[5] These are the memory areas which are initialized by the evaluation of the child nodes or which are defined by the leaf nodes during the first traversal.

Table 2. Performance comparison of the precompiled kernel approach (Section 3.2).

Expression	Precompiled kernels	Operator overloading
`(M1 + M2) + (M3 + M4)`	26 ms	51 ms
`(M1 + M2) + (M3 + M1)`	21 ms	51 ms
`(M1 * M2) * (M3 * M4)`	183 ms	218 ms

is 5 ms faster than the first expression which also adds 4 matrices whereas the operator overloading approach is identical. This speed up was reached because the matrix `M1` is in this approach transferred to the device memory only once.

For the third expression the speed up of the expression template approach is not such significant because the execution of the matrix multiplication on the GPU predominate the required latency time for the data transfer from the host to the device.

As a last remark to the execution time of OpenCL kernels we have compared the kernels on a AMD Radeon HD 6950 GPU against a Intel Core i7-950 CPU. Due to the latency time for the data transfer from the host to a GPU the execution of an expression is only worth for a big enough problem. For example the execution time including the data transfer for a matrix multiplication of two square matrices is almost similar for 256×256 matrices. For smaller matrices the CPU is significantly faster than the GPU. To avoid this runtime penalty we can use template meta programming[6] at compile time or normal branches at run time to decide, if we would run the kernels on the GPU or CPU, respectively.

5 Related Work

In [26] a analogical approach as in Section 3.1 is used to generate CUDA C for vector operations but they have only measured the pure execution time of the kernels neither with a regard for the time of the compilation nor the data transfer.

6 Conclusion and Further Research

In this paper we have shown that it is possible to use expression templates to build a bridge between a domain specific language in C++ and OpenCL. We have presented two different approaches.

The first one is to generate and compile the problem specific OpenCL kernels out of the expression trees. A drawback of this approach is the time required to compile the kernel, which is generated at the evaluation of the expression template. On the other hand this specifically generated kernels showed a good performance for the pure execution time on the GPU. Hence, the code generation approach could be a good choice for hard problems where the compile time is almost irrelevant.

[6] If the problem size is available at compile time.

Our second implementation is used to reduce the necessary data transfers between the domain specific language and the OpenCL device by using precompiled kernels which are called in an optimized way. Thanks to the expression templates it has the benefit that the necessary data transfer is reduced to minimum.

Further investigations are planned to improve the interaction between domain specific languages, expression templates and template meta programming on the host and OpenCL and CUDA on the device to offer libraries for heterogeneous systems which are fast and easy to use.

References

1. Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
2. AMD: AMD accelerated parallel processing OpenCL programming guide, version 1.2c (April 2011)
3. Apple: OpenCL technology brief - Taking the graphics processor beyond graphics (August 2009)
4. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M., Storaasli, O.O.: State-of-the-art in heterogeneous computing. Sci. Program. 18, 1–33 (January 2010)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
6. Härdtlein, J.: Moderne Expression Templates Programmierung. Ph.D. thesis, Universität Erlangen-Nürnberg (2007), in German
7. Intel: Intel OpenCL SDK user's guide, document number 323626-001US (2011)
8. Khronos OpenCL Working Group: The OpenCL Specification, version 1.1.44 (June 2011)
9. Lerch, M., von Gudenberg, J.W.: Expression templates for dot product expressions. Reliable Computing 5(1), 69–80 (1999)
10. Lippman, S.B. (ed.): C++ Gems. SIGS Publications, Inc., New York, NY, USA (1996)
11. MathWorks: MATLAB GPU Computing (April 20 2011), `http://www.mathworks.com/help/toolbox/distcomp/bsic3by.html`
12. Nehmeier, M.: Interval Arithmetic using Expression Templates, Template Meta Programming and the upcoming C++ Standard. SCAN 2010, submitted
13. Nehmeier, M., Wolff v. Gudenberg, J.: filib++, Expression Templates and the Coming Interval Standard. SCAN 2008, to appear
14. NVIDIA: NVIDIA CUDA C best practices guide, version 3.2 (August 2010)
15. NVIDIA: NVIDIA CUDA C programming guide, version 3.2 (November 2010)
16. NVIDIA: NVIDIA CUDA reference manual, version 3.2 Beta (August 2010)
17. NVIDIA: OpenCL Best Practices Guide (May 2010)
18. NVIDIA: OpenCL programming guide for the CUDA architecture, version 3.2 (August 2010)
19. SGI: Standard Template Library Programmer's Guide (April 20 2011), `http://www.sgi.com/tech/stl/`
20. The Portland Group: CUDA Fortran programming guide and reference, version 11.0 (November 2010)

21. TOP500.Org: TOP500 lists November 2010 (April 20 2011), http://www.top500.org/lists/2010/11
22. Vandevoorde, D., Josuttis, N.M.: C++ Templates – the Complete Guide. Addison-Wesley (2003)
23. Veldhuizen, T.: Expression templates. C++ Report 7(5), 26–31 (Jun 1995), reprinted in [10]
24. Veldhuizen, T.: Using C++ template metaprograms. C++ Report 7(4), 36–43 (May 1995), reprinted in [10]
25. Veldhuizen, T.: Techniques for scientific C++. Tech. Rep. 542, Indiana University Computer Science (August 2000), version 0.4
26. Wiemann, P., Wenger, S., Magnor, M.: CUDA expression templates. In: WSCG Communication Papers Proceedings 2011. pp. 185–192 (2011)
27. Wolfram: OpenCLLink user guide (April 20 2011), http://reference.wolfram.com/mathematica/OpenCLLink/tutorial/Overview.html