DOI: 10.14529/jsfi220403

Functional Programming Libraries for Graphics Accelerators

Mikhail M. Krasnov[1], *Olga B. Feodoritova*[1]

© The Authors 2022. This paper is published with open access at SuperFri.org

Modern graphics accelerators (GPUs) can significantly speed up the execution of numerical tasks. However, porting programs to graphics accelerators is not an easy task, sometimes requiring their almost complete rewriting. CUDA graphics accelerators, thanks to the technology developed by NVIDIA, allow you to have a single source code for both conventional processors (CPUs) and graphics accelerators (CUDA). However, parallelization on shared memory is done differently and still must be specified explicitly. The use of the functional programming library developed by the authors makes it possible to hide the use of one or another parallelization mechanism on shared memory inside the library and make the user source code independent of the computing device used (CPU or CUDA). Functional programming is based on the modern mathematical theory, namely the Category Theory, in which the notions of Functors and Monads are widely used. Our work intensively utilizes these notions and extends them to grid expressions used in solving numerical problems.

Keywords: C++, functional programming library, CUDA, OpenMP, OpenCL, OpenACC.

Introduction

In recent years, graphics accelerators (GPUs) have become increasingly popular as computing devices for numerical calculations. Such accelerators are installed on many computing clusters. In the TOP500 list of the most productive supercomputers of November 2022 [1], graphics accelerators from NVIDIA [2] and AMD [3] occupy leading places, including the first place, which has overcome the long-desired exascale performance. NVIDIA, which for many years was among the first, has lost the palm to AMD with its AMD Instinct™ MI250X Accelerator. As for the most high-performance supercomputers in Russia, as of March 2022 [4], almost all of them are equipped with accelerators from NVIDIA. The speed of numerical calculations on such accelerators can be many times higher than on the CPU (according to the experience of the authors, the acceleration can reach 10–20 times), so the transfer of programs that implement numerical methods to graphics accelerators is an extremely urgent task.

However, porting an existing program to a GPU is not an easy task. Of course, the ideal option is to immediately write a program so that it can work on any computer. In any case, the main question arises – what technology to use for the GPU? Currently, there are three main technologies – OpenCL (an open standard for heterogeneous systems) [5], OpenACC [6] and CUDA, developed by NVIDIA for its graphics accelerators [7]. Each of these technologies has its own advantages and disadvantages. The main advantage of OpenCL is that it is an open standard. A program that uses OpenCL will run on any computing device that supports this standard, including NVIDIA and AMD GPUs, Intel Xeon Phi processors with Intel MIC technology, and even conventional CPUs. The main disadvantage of this technology is that the source code of the program often appears in two copies: for the CPU, which is compiled by a conventional compiler and is part of the main program, and the text for OpenCL in separate files. With changes in the algorithms, changes will need to be made in both places. The advantages and disadvantages of the CUDA technology are a mirror image of the advantages and disadvantages of OpenCL. CUDA works only on NVIDIA GPUs. On the other hand, in CUDA we have a single source code that is precompiled and is part of the main program (including the code that will be

[1]Keldysh Institute of Applied Mathematics, Moscow, Russian Federation

executed on the GPU). Unfortunately the OpenACC technology is not accessible to us, as the compiler that supports this technology is not installed on our clusters with graphics accelerators.

Nowadays, many calculations are carried out on heterogeneous systems using graphics accelerators. Examples include recent publications [9, 12, 13]. Some of works use OpenCL technology and as a result have two versions of source code, others use NVIDIA CUDA and have common code for CPU and GPU, but they cannot run on e.g. AMD GPUs. We choose CUDA technology. Our main argument is that in (our) real life we only deal with devices from NVIDIA. AMD GPUs and Intel Xeon Phi processors are still quite exotic for us. Therefore, the disadvantage of CUDA is not a disadvantage for us, but its advantage remains.

The next problem is that shared memory parallelization is done quite differently on the CPU and on the GPU. If we want to get a single text that should be compiled for both the CPU and CUDA, then in those places where there should be parallelization, we will have to write different code (for example, using the **#ifdef** construct), which is inconvenient. We emphasize once again that we are talking about the parallelization of the loops on shared memory. Although it is possible to write a single source code for the CPU and GPU for the loop body, the loop organization itself is written differently.

And then the idea arose to use the funcprog functional programming library for the C++ language [14], previously written by one of the authors of the article. An appropriate modification of this library will allow all the specifics of a computing device (CPU or CUDA) to be placed inside the library, and the user source code will turn out to be completely platform independent.

The article is organized as follows. Section 1 is devoted to a brief introduction to functional programming (to the extent necessary to understand the rest of the text). In Section 2 a brief description of the funcprog2 functional programming library is given and Section 3 contains examples of using this library to solve numerical problems. Conclusion summarizes the study and points directions for further work.

1. Functional Programming Library

1.1. Scope of the Funcprog2 Library

Let us describe a family of algorithms for which the developed funcprog2 library is applicable.

Technologically, any task of numerical simulation begins with the construction of a grid in the calculation area. Moreover, in areas of complex shape, this grid is, as a rule, non-structural. The grid functions of interest to the researcher can be specified both at the nodes of the cells and at their centers.

At this stage of the study, we consider only non-stationary problems and believe that various explicit schemes are used for time integration, for example, in the software package that we used to transfer to the GPU, this is an explicit classical Runge–Kutta scheme of the fourth order. Implicit schemes involving the solution of linear systems of equations have not been considered at the moment. Using the described approach to solve settling problems using implicit schemes requires additional developments that expand the capabilities of the presented library.

If the method is explicit, then the values of the grid functions at different points of the grid can be calculated independently of each other, and, therefore, these calculations can be carried out in parallel. Thus, each explicit step (loop over the grid function index) can be parallelized on shared memory. Note that MPI parallelization is also possible, but has not yet been considered.

The developed functional programming library funcprog2 allows an applied mathematician to implement the described numerical algorithms without delving into the details and features of parallelization.

1.2. General Description of the Funcprog2 Library

When implementing the functional programming library funcprog2 for the C++ language, the task was to write a library with which one could write in the C++ language in a style close to the style of the Haskell language [8]. We cite the Haskell language as a role model, as it seems to be one of the most advanced modern functional programming languages based on modern mathematical theory (category theory), widely used and actively supported by the world scientific community. Details about category theory can be found, for example, in the books [15, 16].

An important question is what is a function from the point of view of this library? In the original version of the library, a function meant an object of the std::function class. This option does not suit us now, since we want the function to be executed on the graphics accelerator, and the std::function class object can only be executed on the CPU (mainly because its implementation uses virtual functions that are not portable on the GPU). It cannot be an ordinary function either, since it cannot be passed as a parameter from the CPU to CUDA, because an ordinary function can only be passed by address, and addresses cannot be passed from the CPU to CUDA. It was decided to create a function2 class within the funcprog2 library and consider any object of this class to be a function. Any function object (having a functional operator ()) can be converted into an object of the function2 class, in particular, it can be a lambda expression. Recall that in modern C++ (since C++11), a lambda expression begins with a pair of square brackets, inside of which variables accessible from the lambda expression can be placed. In order for an object to be passed to CUDA, the functional operator must be marked with the __device__ keyword. To make user source code platform-independent, the funcprog2 library has a __DEVICE macro, which expands to the __device__ keyword when compiled for CUDA, and to an empty string when compiled for CPU. Thus, a functional operator must be marked with the word __DEVICE. To convert a functional object into an object of the function2 class, the library has a special function _ (underscore). Here is an example of working with the funcprog2 library:

```
double d=(_([](double x){ return x * x; }) &
  _([](double x){ return x + 1; }))(5); //36
```

In this example, we created two functions, composed them (using the & operator), and invoked the resulting compound function with a parameter of 5. The result is 36.

1.3. Implementation of Functors, Applicatives and Monads

The funcprog2 library essentially relies on the concepts of functor, applicative, and monad. The implementation of functors, applicatives, and monads in the library is similar to the implementation of these concepts in Haskell language. Any class can declare itself a functor, an applicative, or a monad. To do this, it is enough to implement a specialization of the Functor, Applicative and Monad classes, respectively, for this class. You don't need to make any changes to the class itself.

Inside the specialization of the Functor class, you need to define a static function fmap. The specializations of the Applicative and Monad classes are defined similarly. For an applicative,

the methods are called pure and apply, and for a monad, mreturn and bind. When implementing the static methods of these classes, one should not forget about the implementation of functor, applicative and monadic laws. Note also that in the funcprog2 library, the division operator is used as a functor operator, and multiplication is used as an applicative operator.

1.4. Grid Expressions and Grid Functions

The notion of a grid expression plays a significant role in the funcprog2 library. This is an object defined for all grid indices, that is, for any object that is a grid expression, you can find out what its value is for a given index. The simplest special case of a grid expression is a grid function, which simply stores its values in memory and returns them, if necessary. For grid expressions, a grid_expression class template is defined, from which all classes of objects that are grid expressions must inherit (in particular, the grid_function class is also inherited from the grid_expression class). Thus, the phrase "an object is a grid expression" means that the class of this object is inherited from the grid_expression class. This inheritance uses expression templates [18] and the CRTP (Curiously Recurring Template Pattern) [10] design pattern, in which the final class is passed to the base class as a template parameter.

Any grid expression can be assigned to a grid function. This assignment operator iterates over all indices of the grid function to which the grid expression is assigned, for each index queries the grid expression for its value, and assigns that value to the grid function at the given index. The assignment operator implies that values for different indices can be calculated independently of each other, and therefore they can be calculated in parallel. It is in the assignment operator that the inner loop over the elements of the grid function is performed. The method of parallelization of this loop is chosen by the assignment operator, depending on which compiler the program is compiled with. If this is a compiler for CUDA (the __CUDACC__ preprocessor variable is defined), then parallelization is performed using CUDA, otherwise, using OpenMP. Thus, the parallelization method is hidden from the application programmer within this assignment statement.

Speaking about grid functions, one more aspect should be mentioned. The GPU can only work with its own memory, which means that when working on the GPU, the grid function must request memory for its data in CUDA memory. There are no problems with this either. Grid functions are arranged in such a way that when compiled on CUDA they request memory from CUDA, otherwise they request memory from the CPU.

1.5. Grid Expressions as Functors, Applicatives and Monads

Grid expressions can be thought of as containers (this is especially true for grid functions). In the funcprog2 library, containers (such as lists) are functors, applicatives, and monads. This makes it possible to apply ordinary functions to the values stored in them (a property of functors). Let's make the grid expression also a functor, an applicative, and a monad so that functions can be applied to grid expressions as well. To understand how this can be done, consider a typical loop that calculates the new value of the grid function from the old one:

```
for(size_t i = 0; i < N; ++i)
  f_new[i] = calculate(f_old[i]);
```

Here *calculate* is a function that calculates the new value in the cell according to the old one. It is passed the old value in the cell as a parameter. In the new approach, we want to be able to write something like this in this case:

```
f_new = _(calculate) / f_old;
```

If the calculations require several more grid functions (let's call them f2 and f3), then instead of

```
for(size_t i = 0; i < N; ++i)
  f_new[i] = calculate(f_old[i], f2[i], f3[i]);
```

we could write:

```
f_new = _(calculate) / f_old * f2 * f3;
```

that is, for the first grid function, we applied the functor property, and for the subsequent ones, we applied the applicative. If we want to pass some additional constant value to the function (independent of the cycle index), then instead of

```
for(size_t i = 0; i < N; ++i)
  f_new[i] = calculate(f_old[i], some_value);
```

we could write

```
f_new = _(calculate) / f_old * pure(some_value);
```

Thus, the result of applying a function to a grid expression (or to several grid expressions in the case of an applicative) must also be a grid expression, that is, it can be asked for a value by index (the [] operator must be implemented). Grid expressions, in addition to grid functions, are also the results of applying functions to grid expressions as functors and monads. In addition, the sum and difference of two grid expressions, as well as the product and quotient of a grid expression and a number, are also grid expressions.

Let's show how functors, applicatives and monads for grid expressions are implemented.

Functor. The *fmap* function takes a function with one parameter and a functor (a grid expression in our case) and returns the same functor (a new grid expression). This new grid expression stores the parameters of the fmap function in its class member variables (let's call them *f* and *gexp*) and implements the [] operator as follows (in pseudo-Haskell):

```
(fmap f gexp)[i] = f gexp[i]
```

Applicative. The *pure* function takes some value and "introduces" it into the applicative. In our case, it makes a grid expression out of it. Let's define its operator [] so that it returns the same value val for any index:

```
(pure val)[i] = val
```

The *apply* function (an analogue of the (<∗>) operator in Haskell) in our case takes two grid expressions: the first (let's call it gexp_f) returns functions, and the second (let's call it gexp) returns some values (the parameters of these functions). Let's define the grid expression of the *apply* function as follows:

```
(apply gexp_f gexp)[i] = gexp_f[i] gexp[i]
```

Monad. The *return* monad function is defined in the same way as the *pure* applicative function:

```
return = pure
```

The *bind* monad operation (in the Haskell language and in the funcprog2 library, the (>>=) operator) takes a monad (in our case, a grid expression, let's denote it by the variable *gexp*) and a function that takes a regular (non-monadic) value and returns a monad (in our case, a grid expression). Let's define the *bind* operation as follows:

```
(bind gexp f)[i] = (f gexp[i])[i]
```

We have proved three theorems that these definitions of the functor, applicative, and monad for grid expressions are correct, that is, they satisfy the functor, applicative, and monad laws, respectively. This article does not present this evidence.

So grid expressions are functors, applicatives, and monads. This means that any ordinary unary function can be "applied" to a grid expression (using the *fmap* function). This application will return a new grid expression. Any binary function can be "applied" to two grid expressions (using the *apply* function), and any function with n arguments can be "applied" to n grid expressions. This can be done in one line. For example, suppose there are two grid functions f and g. Then you can write like this:

```
g = _([](double x){ return sin(x); }) / f;
```

Since grid expressions are monads, it is possible to build chains of monad calculations of the form from them:

```
g = f >>= f1 >>= f2 >>= f3;
```

Here f1, f2, f3 are some functions that take ordinary (non-monadic) values and return grid expressions.

2. Examples

2.1. The Simplest Example

Consider the *axpy* function from the BLAS library. This function takes a constant a and two vectors (x and y) and modifies the vector y as y[i] += a ∗ x[i]. Its implementation for a conventional processor can be written as follows:

```
template<typename T>
void axpy(T a, vector<T> const& x, vector &y){
#pragma omp parallel for
  for(int i = 0; i < y.size(); ++i)
    y[i] += a*x[i];
}
```

This function is perfectly parallelized, but the method of parallelization in this implementation is specified explicitly and is not suitable for graphics accelerators. Using functional programming, this function could be rewritten as follows:

```
template<typename T>
void axpy(T a, grid_function<T> const& x, grid_function<T> &y){
  y = _([](T a, T xi, T &yi, size_t /*i*/){
    yi += a * xi;
  }) / pure(a) * x;
}
```

The lambda expression in the body of this function takes as parameters our constant a, the i-th element of the grid function x, by reference the i-th element of the grid function y, and the current loop index i (which we ignore). Each input parameter (and we have two of them) corresponds to one parameter of the lambda expression, and the grid function, to which the expression is assigned, corresponds to the output parameter (passed by reference) and the loop index. The first parameter (in our case it is pure(a)) is passed as for the functor (via the / operator), and the next as for the applicative (via the * operator). Here is an example of calling the *axpy* function:

```
int main(){
  size_t const N = 10;
  math_vector<double> x(N, 2), y(N, 3);
  axpy(5., x, y);
  std::cout << y[0] << std::endl; // 13
}
```

2.2. More Complex Example

Now we will calculate an expression like z[i]=a*x[i]+b*y[ind[i]] (let's call the function *axpby*). Its peculiarity is that the index of the grid function y is contained in the additional grid function *ind*:

```
template<typename T>
void axpby(T a, grid_function<T> const& x,
T b, grid_function<T> const& y,
grid_function<size_t> const& ind, grid_function<T> &z)
{
  z = _([](T a, T xi, T b, grid_function_proxy<T> const y_p,
    size_t indi, T &zi, size_t /*i*/)
  {
    zi = a * xi + b * y_p[indi];
  }) / pure(a) * x * pure(b) * pr(y) * ind;
}
```

What is essentially new compared to the first example is that we can no longer pass the grid function y through *apply*. Instead, we need to create a proxy object for it and pass it through pure. This is exactly what the *pr* library function does:

```
template<class F>
auto pr(F const& f){
  return pure(get_proxy(f));
}
```

A few words about proxy objects. They are needed to transfer entire grid functions to the GPU. The fact is that in the GPU, parameters can only be passed by value, that is, their copy will be made. But a copy of the grid functions cannot be made, as this will lead to the creation of a copy of the data. Instead, a "light" placeholder object is created for the grid function, storing a pointer to the data and the size of the grid function. It is this object that is passed by value as a whole (using the pure function).

Let's also pay attention to the fact that global variables that are stored in CPU memory are not available from the GPU, so they have to be passed explicitly using the pure function. If you need a buffer for intermediate calculations, then its size should be equal to the number of grid nodes, since in the case of CUDA we do not know the absolute number of the execution thread.

Another important problem that often arises is reduction (for example, finding the maximum value or the sum of all values). It is also very important to carry out the reduction in parallel. In the original version of the program, the reduction was performed by means of OpenMP, but now we cannot use this mechanism. Fortunately, in modern C++ (since the C++17 language standard), reduction functions (such as std::accumulate and std::reduce) have parallel versions. When working on CUDA, we use the thrust library, which is part of the CUDA Computing Toolkit. This library also has parallel reduction functions running on GPU. Thus, when working on CUDA, we use the parallel reduction functions from the thrust library, when working on the CPU, if there are parallel versions of the reduction functions (if the compiler supports them), then they are used, otherwise the reduction is done sequentially.

3. Performance Comparison

To evaluate the effectiveness of the proposed approach, a problem is proposed that simulates an experiment carried out in the L2K pipe (Germany, [17]). An axisymmetric body consisting of two conical and cylindrical surfaces is placed in an oncoming air flow with the following characteristics: $M_\infty = 4.7$, $p_\infty = 272Pa$, $T_\infty = 764K$, $Y_{O2} = 0.245$, $Y_{N2} = 0.755$. The geometry of the problem is shown in Fig. 1. The solid body is combined from two parts with different thermodynamic characteristics: the head part is made of UHTC (Ultra High Temperature Material) and the rear cylindrical part is copper.

The initial temperature of the body is T=300 K. The adjoint problem is solved in a two-dimensional formulation. Modeling is based on the solution of the Navier–Stokes equations in a multicomponent gas and the heat equation in a solid using an original technique based on explicit Chebyshev iterations [11]. The mesh size is 297192 nodes.

We solved this problem using a K60gpu hybrid supercomputer installed at the Keldysh Institute of Applied Mathematics of RAS. One node of this supercomputer has two host processors Intel Xeon Gold 6142 v4, 16 cores (total 32 threads) and four GPUs nVidia Volta GV100GL. We used the program complex NOISETTE [12], which originally can work in two modes of parallelization: on CPU using OpenMP and on using OpenCL (with separate sources for kernels). We have rewritten it using our approach. For testing on CPU mode we used one host (32 threads) and for OpenCL and CUDA modes – one GPU. The acceleration compared to the CPU when using OpenCL was about 20 times, and when using our approach – about 12 times. The acceleration is not as great as when using OpenCL, but given the above advantages (above all, a single source code), it can be considered acceptable.

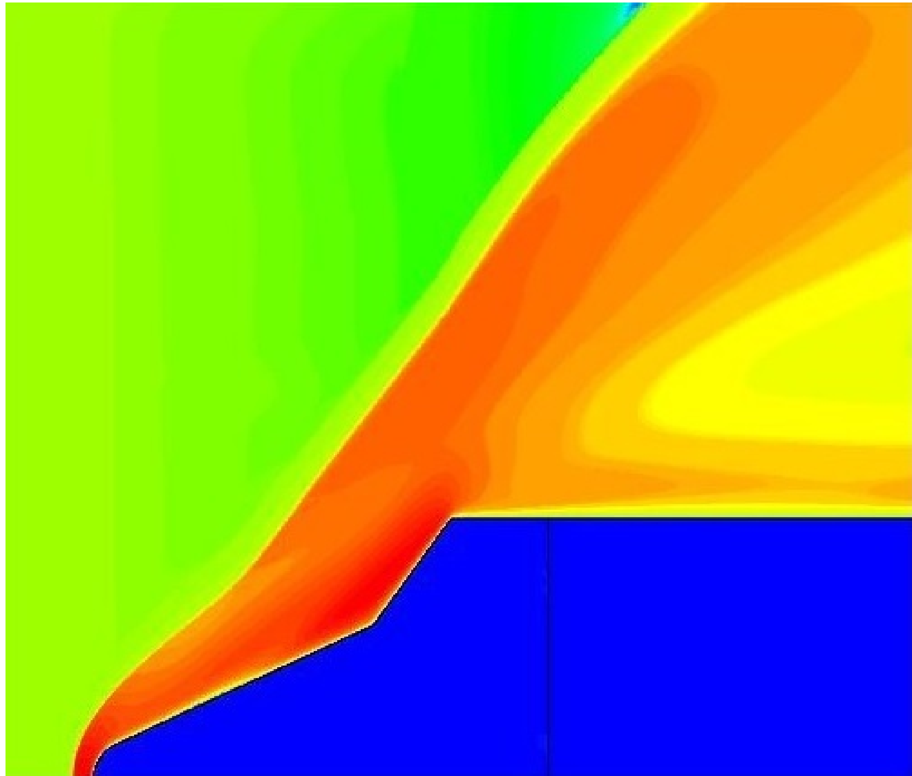

Figure 1. Geometry of the problem with the shock wave structure

Conclusion

Declarative programming languages, which include functional languages, allow, unlike imperative languages, which include most programming languages in which numerical methods are implemented, to briefly and at the same time quite clearly write down the desired result without going into implementation details. The specific implementation may be hidden in the language and depend on the current hardware and software environment. The C++ language proved to be powerful enough to allow the implementation of a functional programming library on it, which allows you to write programs in a style close to the style of purely functional languages such as Haskell. Such concepts from the world of functional programming as functors and monads, implemented in the functional programming library, turned out to be a very convenient tool for porting numerical problems to CUDA graphics accelerators. Grid expressions have been defined as functors, applicatives, and monads, allowing functions to be applied to the values they store. These functions themselves can be built by combining complex functions from simple ones, which is also the strength of functional programming.

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References

1. TOP 500. `https://www.top500.org`

2. NVIDIA. `https://www.nvidia.com`

3. AMD. https://www.amd.com

4. TOP 50. http://top50.supercomputers.ru

5. OpenCL. https://www.khronos.org/opencl

6. OpenACC. https://www.openacc.org

7. CUDA Zone. https://developer.nvidia.com/cuda-zone

8. Haskell language. https://www.haskell.org

9. Chaplygin, A., Gusev, A., Diansky, N.: High-performance shallow water model for use on massively parallel and heterogeneous computing systems. Supercomputing Frontiers and Innovations 8(4), 74–93 (2021). https://doi.org/10.14529/jsfi210407

10. Coplien, J.O.: Curiously recurring template patterns. C++ Report pp. 24–27 (February 1995)

11. Feodoritova, O., Krasnov, M., Zhukov, V.: A numerical method for conjugate heat transfer problems in multicomponent flows. J. Phys.: Conf. Ser 2028 012024 (2021). https://doi.org/10.1088/1742-6596/2028/1/012024

12. Gorobets, A., Bakhvalov, P.: Heterogeneous CPU+GPU parallelization for high-accuracy scale-resolving simulations of compressible turbulent flows on hybrid supercomputers. Computer Physics Communications 271(108231) (February 2022). https://doi.org/10.1016/j.cpc.2021.108231

13. Gorobets, A., Duben, A.: Technology for supercomputer simulation of turbulent flowsin the good new days of exascale computing. Supercomputing Frontiers and Innovations 8(4), 4–10 (2021). https://doi.org/10.14529/jsfi210401

14. Krasnov, M.M.: Functional programming library for C++. Programming and Computer Software 46, 330–340 (2020). https://doi.org/10.1134/S0361768820050047

15. MacLane, S.: Categories for the Working Matematitian. Springer (1998)

16. Milewski, B.: Category Theory for Programmers (2019), https://github.com/hmemcpy/milewski-ctfp-pdf/releases/download/v1.3.0/category-theory-for-programmers.pdf

17. Murty, C., Manna, P., Chakraborty, D.: Conjugate heat transfer analysis in high speed flows. Proceedings of Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering 227(10), 1672–1681 (2013). https://doi.org/10.1177/0954410012464920

18. Veldhuizen, T.: Expression templates. C++ Report 7(5), 26–31 (June 1995)