DOI: 10.15514/ISPRAS-2021-33(5)-10

Using the functional programming library for solving numerical problems on graphics accelerators with CUDA technology

[1,2]*M.M. Krasnov, ORCID: 0000-0001-7988-6323 <kmm@kiam.ru>*
[1]*O.B. Feodoritova, ORCID: 0000-0002-2792-9376 <feodor@kiam.ru>*
[1]*Keldysh Institute of Applied Mathematics of Russian Academy of Sciences,*
4, Miusskaya sq., Moscow, 125047, Russia
[2]*Moscow Institute of Physics and Technology,*
9, Insitutsky per., Dolgoprudny, 141701, Russia

Abstract. 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 the transfer of programs to such accelerators is carried out by almost completely rewriting them (for example, when using the OpenCL technology). This raises the daunting task of maintaining two independent source codes. However, CUDA graphics accelerators, thanks to technology developed by NVIDIA, allow you to have a single source code for both conventional processors (CPUs) and CUDA. The machine code generated when compiling this single text depends on which compiler it is compiled with (the usual one, such as gcc, icc and msvc, or the compiler for CUDA, nvcc). However, in this single source code, you need to somehow tell the compiler which parts of this code to parallelize on shared memory. For the CPU, this is usually done using OpenMP and special pragmas to the compiler. For CUDA, parallelization is done in a completely different way. The use of the functional programming library developed by the authors allows you to hide the use of one or another parallelization mechanism on shared memory within the library and make the user source code completely independent of the computing device used (CPU or CUDA). This article shows how this can be done.

Keywords: C ++; functional programming library; CUDA; OpenMP; OpenCL; OpenACC

For citation: Krasnov M.M., Feodoritova O.B. Using the functional programming library for solving numerical problems on graphics accelerators with CUDA technology. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 5, 2021, pp. 167-180. DOI: 10.15514/ISPRAS–2021–33(5)–10

Использование библиотеки функционального программирования для решения численных задач на графических ускорителях с технологией CUDA

[1,2]*М.М. Краснов, ORCID: 0000-0001-7988-6323 <kmm@kiam.ru>*
[1]*О.Б. Феодоритова, ORCID: 0000-0002-2792-9376 <feodor@kiam.ru>*
[1]*Институт прикладной математики им. М.В. Келдыша РАН,*
125047, Москва, Миусская пл., д.4
[2]*Московский физико-технический институт,*
141701, Россия, Долгопрудный, Институтский пер., д. 9

Аннотация. Современные графические ускорители (GPU) позволяют существенно ускорить выполнение численных задач. Однако перенос программ на графические ускорители является непростой задачей. Иногда перенос программ на такие ускорители осуществляется путём практически полного их переписывания (например, при использовании технологии OpenCL). При этом возникает

непростая задача поддержки двух независимых исходных кодов. Однако, графические ускорители CUDA, благодаря разработанной компанией NVIDIA технологии, позволяют иметь единый исходный код как для обычных процессоров (CPU), так и для CUDA. Машинный код, генерируемый при компиляции этого единого текста, зависит от того, каким компилятором он компилируется (обычным, таким, как gcc, icc и msvc, или компилятором для CUDA, nvcc). Однако, в этом едином исходном коде нужно каким-то образом указать компилятору, какие части этого кода нужно распараллеливать на общей памяти. Для CPU это обычно делается с помощью OpenMP и специальных прагм компилятору. Для CUDA распараллеливание делается совершенно по-другому. Применение разработанной авторами библиотеки функционального программирования позволяет скрыть использование того или иного механизма распараллеливания на общей памяти внутри библиотеки и сделать пользовательский исходный код полностью независимым от используемого вычислительного устройства (CPU или CUDA). В настоящей статье показывается, как это можно сделать.

Ключевые слова: C++; библиотека функционального программирования; CUDA; OpenMP; OpenCL

Для цитирования: Краснов М.М., Феодоритова О.Б. Использование библиотеки функционального программирования для решения численных задач на графических ускорителях с технологией CUDA. Труды ИСП РАН, том 33, вып. 5, 2021 г., стр. 167-180 (на английском языке). DOI: 10.15514/ISPRAS–2021–33(5)–10

1. Introduction

In recent years, graphics accelerators (GPUs), used as computing devices for numerical calculations, have become increasingly widespread. Such accelerators are installed on many computing clusters, in particular, in the TOP500 list of the most productive supercomputers from June 2021, in the top ten, six use graphics accelerators from NVIDIA [1]. The speed of numerical calculations on such accelerators can be many times higher than on a CPU (according to the authors' experience, the acceleration can reach 10-20 times). Therefore, the transfer of programs implementing numerical methods to graphics accelerators is an extremely urgent task.

However, porting an existing program to a GPU is not an easy task. Perhaps the ideal option is to write the program right away so that it can execute on any computer. In any case, the first question that arises is what kind of GPU technology to use? At the moment there are at least three main technologies - OpenCL (open standard for heterogeneous systems) [2], OpenACC [3] and CUDA - developed by NVIDIA for its graphics accelerators [4]. Each of these technologies has advantages and disadvantages. The main advantage of OpenCL is its open standard. A program using 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 appears in two copies: for the CPU, which is compiled by a regular compiler and is part of the main program, and the text for OpenCL is in separate files, and when changing algorithms, you must make changes in both places. The advantages and disadvantages of CUDA technology are a mirror image of the disadvantages and advantages of OpenCL. CUDA only runs on NVIDIA GPUs. On the other hand, in CUDA we have a single source code that is precompiled as a part of the main program (including the code that will be executed on the GPU). The main disadvantage of the OpenACC technology is that it is not yet widespread enough. Compilers that supports this technology are not installed on all clusters with graphics accelerators.

We choose CUDA technology. Our main argument is that in (our) real life we are faced exclusively with devices from NVIDIA. GPUs from AMD and Intel Xeon Phi processors are quite exotic, and although we have met them, they are really irrelevant. Therefore, the disadvantages of CUDA is not a disadvantage for us, but its advantages remains.

The next problem is that parallelization on shared memory on the CPU and on CUDA is done in completely different ways. If we want to get a single text that should be compiled for both CPU and CUDA, then in those places where parallelization should be, we will have to write different code (for example, using the #ifdef construction), which is inconvenient. Then the idea arose to use a library of functional programming for the C++ language previously written by one of the authors of the article [5]. When using this library, all the specifics of the computing device (CPU or CUDA) can be placed inside the library, and the user source code will remain platform independent.

This article consists of three main parts: a short introduction to functional programming (as long as needed to understand the rest of the text), a short description of the funcprog functional programming library, and a description of the use of this library for solving numerical problems. A short working example is also given.

2. A brief introduction to functional programming

In functional programming, the central object is (as the name suggests) a function. Functions are full participants in the computational process, the same as numbers are in ordinary calculations. This means that a function can be passed as a parameter to another function and can be returned as a result of a function. A function can be calculated, just as a number can be calculated in normal calculations. A simple example is a composition of two one-place functions, which returns a new one-place function that calls both functions in sequence. In specialized functional programming languages (such as Haskell), such capabilities are built into the language, while the implementation of function composition in C++ is a non-trivial task that requires special tweaks. Examples will be given in the Haskell language, since this language allows you to write many things as concisely and at the same time clearly.

Functional programming has a number of features compared to imperative programming that can be summarized in several principles. The first and main mandatory principle, already mentioned above, is that a function is a full-fledged participant in the computational process and can be either passed as a parameter or returned as a result of a function. Among others we can mention function purity, immutable variables and lazy calculations. Let us shortly describe some other principles (important for our discussion).

Currying. Named after the American mathematician and logician Haskell Curry (the programming language Haskell is also named after him). The principle of currying is that if the parameters of a function are not fully specified, an error does not occur, but instead a function is generated with a smaller number of parameters (equal to the number of missing parameters). It is implemented in many modern functional programming languages (in particular, in the Haskell language). Currying allows a function with multiple arguments to be treated as a collection of functions with one argument.

η-reduction (eta reduction, or η-transformation). Suppose we want to write a function with one parameter, a list of numbers, that returns a list of the sines of these numbers. The text of this function in Haskell is obvious:

```
mapsin lst = map sin lst
```

It follows from the currying principle that if we omit the second parameter when calling the map `sin lst` function, that is, we write just map sin, then we will get a function with one parameter that takes a list of numbers as this parameter and returns a list of sines of these numbers, that is, in fact `mapsin` function. That is, `mapsin` is equivalent to `map sin`. The principle of η-reduction says that in such cases, the last parameter (one or more) in the function definition can be omitted. The definition of the `mapsin` function can be written shorter: `mapsin = map sin`.

Functions composition. Function composition is so important in functional programming that Haskell makes it as simple as possible. Usually, one considers the composition of one-place functions (let's call them `f` and `g`), in Haskell it is written like this:

```
(f . g) x = f (g x)
```

3. Functors, Applicatives, and Monads

Functors. Suppose we have some container that stores a number of values, for example, a list or an object or the Maybe class. Now let us set the task: apply a regular one-place function (for example, sin) to the values in the container. You know how to do this with lists – use the map function. However, how do you do this with the Maybe type, and in general, how do you do it with data in an arbitrary container? The universal approach is to entrust this responsible business to the container itself. For this, Haskell defines a special `Functor` class, in which the `fmap` function is declared:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$>) = fmap
```

The operator `<$>` is an infix synonym of the fmap function. This is a statement for applying a function to a functor. It is similar to the operator for applying a function to an ordinary value (`$`). The fmap function prototype can be written in another equivalent form (this follows from the right-associativity of the right arrow):

```
fmap :: (a -> b) -> (f a -> f b)
```

Thus, `fmap` can be thought of as a one-parameter function that takes a function that takes and returns normal values, and converts it into a function that takes and returns functors. Any data type can be declared a functor by implementing an instance of the `Functor` class and the `fmap` function for it. Any functor implementation must satisfy two functor laws:

```
1. fmap id = id                    -- 1st law
2. fmap (g . f) = fmap g . fmap f -- 2nd law
```

Here `id` is a function that returns its argument: `id x = x`. The first law says that applying the id function to a functor should not change the functor, just as applying this function to an ordinary value does not change it. The second law is the distribution law of the functor operation with respect to the composition of functions.

Applicatives. If the task is to apply a function with two arguments to two containers (for example, to sum two lists), then the functionality of the `Functor` class will not be enough. To solve this problem, another class is intended – an applicative functor (applicative). Here is the definition of the `Applicative` class:

```
class Functor f => Applicative f where
  pure  :: a->f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a->b->c)->f a->f b->f c
  liftA2 f x y = f <$> x <*> y
```

Thus, in each applicative, two main operations must be implemented: the pure function, which puts an ordinary value in the «pure» applicative, and the operator (`<*>`), which takes the function placed in the applicative as the first parameter, and the value, placed in the same applicative, as the second parameter and returning the result in the same applicative.

If we look at the prototype and implementation of the `liftA2` function, we can see that it passes a function with two parameters to the operator (`<$>`), which takes a function with one parameter. However, there is no contradiction here, since we can write a function with the prototype `a->b->c` as follows: `a->(b->c)`, that is, as a function with one parameter that returns a function. Then the operator (`<$>`) will just return to us the function `b->c`, placed in the functor, which is then passed to the operator (`<*>`). By analogy with the `fmap` function, we can write the prototype of the `liftA2` function like this:

```
liftA2 :: (a->b->c)->(f a->f b->f c)
```

that is, think of it as a function with one argument, taking a function that works with ordinary values and returns a function that works with functors, «lifting» a function with two arguments (hence its name) into the applicative. By analogy with the `liftA2` function, you can write a function that «lifts» a function with three (and any other number) arguments into the applicative:

```
liftA3 :: Applicative f =>
  (a->b->c->d)->f a->f b->f c->f d
liftA3 f x y z = f <$> x <*> y <*> z
```

Any implementation of the applicative must satisfy the applicative laws:

```
pure id <*> v = v                 -- Identity
pure f <*> pure x = pure (f x)    -- Homomorphism
u <*> pure y = pure ($ y) <*> u -- Interchange
-- Composition:
pure (.) <*> u <*> v <*> x = u <*> (v <*> x)
```

Monads. Let's write the «safe» functions `safe_sqrt` and `safe_log`:

```
safe_sqrt :: (Ord a, Floating a) => a -> Maybe a
safe_sqrt x=if x<0 then Nothing else Just(sqrt x)
safe_log :: (Ord a, Floating a) => a -> Maybe a
safe_log x=if x<=0 then Nothing else Just(log x)
```

These functions return the result of `Maybe` data type, which has two constructors, `Nothing` with no arguments and `Just` with exactly one argument. Returning `Nothing` means an error condition (the function's argument is out the scope of its definition) and `Just` – normal result. Now suppose we want to calculate the square root of the logarithm of a number. For both operations, we have «safe» functions. How do we now apply the `safe_sqrt` function to the result of the `safe_log` function? The functionality of functor and applicative is not enough for this. Monads serve this purpose. Monads can be viewed as a further continuation of the applicative; they are intended for building chains of monad calculations. Each monad has two main functions: `mreturn` (`return` in Haskell) and `mbind` (operator >>= in Haskell). The `mreturn` function is similar to the pure function for applicatives (in fact, for most monads, `mreturn` is defined as pure), and the `mbind` operation has the following definition:

```
(>>=) :: (Monad m) -> m a -> (a -> m b) -> m b
```

It takes as parameters a monad and a function that takes an ordinary (non-monad) value and returns a monad value (possibly of a different type, but in the same monad). We will call such functions monadic. The `safe_log` and `safe_sqrt` functions are examples of monadic functions.

The `mreturn` and `mbind` functions must obey three monad laws. In order to formulate them, we introduce the operation of monad composition (`mcompose` or the operator >=> («fish» operator) in the Haskell language). It is defined as follows:

```
  (>=>) :: f >=> g = \x -> (f x >>= g)
```

Both operands of a monad composition and its result are monadic functions. Consequently, monadic composition can be viewed as a group operation in the space of such functions. In terms of this group operation, monad laws are formulated as follows:

```
1.  mreturn >=> f == f
2.  f >=> mreturn == f
3.  (f >=> g) >=> h == f >=> (g >=> h)
```

In other words, the `mreturn` and `mbind` functions must be defined so that, firstly, the `mreturn` function is the unit element (left and right) of the monad composition (the first two laws), and, secondly, the monad composition must be associative (the third law). Now the square root of the logarithm can be calculated like this:

```
safe_log   5  >>= safe_sqrt -- Just 1.26863624
safe_log (-5) >>= safe_sqrt -- Nothing
safe_log 0.5  >>= safe_sqrt -- Nothing
```

If an error occurs somewhere in the chain of calculations, then a quick exit from the entire chain occurs, the rest of the functions are not actually calculated. Keep in mind that the `Maybe` data type is declared as a functor, an applicative and a monad.

4. Functional programming library

4.1 General description of the library

When implementing the functional programming library for the C++ language named funcprog, the task was to write a library with which one could write in C++ in a style close to the style of the Haskell language.

The important question is what is a function in terms of this library? In the original version of the library, a function was understood as an object of the `std::function` class. This option does not suit us now, since we want the function to be executed on a graphics accelerator, and an object of the `std::function` class can be executed only on the CPU (in particular, because its implementation uses virtual functions that are not portable to the GPU). It cannot be an ordinary function either, since it (its address) cannot be passed as a parameter from the CPU to CUDA. It was decided to consider any functional object (having a functional operator ()) as a function, in particular, it can be a C++ lambda expression. However, in order for this object to be passed to CUDA, this functional operator must be marked with the __device__ keyword. The disadvantage of this approach is that the metadata of the function includes not only its prototype (parameter types and return value), but also the implementation (object class) – this is the price for the possibility of porting to CUDA. This is how the function is implemented:

```
template<typename FuncType, typename FuncImpl> struct function2;

template<typename Ret, typename... Args, typename FuncImpl>
struct function2<Ret(Args...), FuncImpl> {
  using result_type = Ret;
  function2(FuncImpl const& impl) : impl(impl){}
  result_type operator()(Args... args) const
  { return impl(args...); }
private: FuncImpl const impl;2
};
```

You can see that, unlike the `std::function` class, two template parameters are passed to the `function2` class. To simplify the work with such functions, there is a helper function _ (underscore). Here's how it is defined:

```
template<typename FuncImpl, typename Ret,
  typename... Args>
struct function2_type_ {
  using type = function2<Ret(Args...), FuncImpl>;
};

template<class F>  // Common case for functors & lambdas
```

```
struct function2_type : function2_type<decltype(&F::operator())>{};

template<class Cls,typename Ret,typename... Args>
struct function2_type<Ret(Cls::*)(Args...)> :
  function2_type_<Cls, Ret, Args...>{};

template<class Cls,typename Ret,typename... Args>
struct function2_type<Ret(Cls::*)(Args...) const>
  : function2_type_<Cls, Ret, Args...>{};

template<typename F>
using function2_type_t =
  typename function2_type<F>::type;

template<typename F>
function2_type_t<F> _(F f){ return f; }
```

Let's give a short example of working with this library:

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

In this example, we create two functions (using the _ function), compose them (using the & operator), and call the resulting composite function with the parameter 5. As a result, we get the number 36.

The funcprog library implements function currying quite fully. To do this, the library implements an operator for applying an argument to a function using the left shift operator <<. This creates a new function with one less parameter. In particular, if the original function had a single parameter, then a function with no parameters will be created.

4.2 Implementation of functors, applicatives and monads

The implementation of functors, applicatives and monads in the funcprog library is somewhat similar to the implementation of these concepts in the Haskell language. Any class can be declared as a functor, an applicative, or a monad. For this, it is enough to implement the specialization of classes `Functor`, `Applicative` and `Monad`, respectively, for this class. No changes are required to the class itself.

Any functor or monad is a type with one parameter. In C++, parameterized types are implemented using class templates. Let us look at the definition of a functor using the Maybe class as an example. It is defined like this:

```
template<typename A> struct Maybe;
```

A class `template` is not a type and cannot be passed as a parameter to a template of another class. Therefore, another class (without a template) is defined with the name `_Maybe` (with an underscore in front). This is a real class, it can be passed as a template parameter:

```
struct _Maybe {};
```

The `Maybe` class template inherits from this class:

```
template<typename A>
struct Maybe : std::optional<A>, _Maybe, {
  ...
};
```

The specializations of the `Functor`, `Applicative` and `Monad` classes are written for this `_Maybe` class:

```
template<> struct Functor<_Maybe>{
  ...
};
```

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 the applicative, the methods are called pure and apply, and for the monad their names are `mreturn` and `mbind`. When implementing static methods of these classes, one should not forget about the functor, applicative and monad laws.

Note also that the funcprog library uses the division operator as a functor operator (`fmap`), and multiplication as an applicative operator (apply).

5. Using the library for numerical methods

When solving numerical problems (for example, problems of gas dynamics or heat conduction), a certain grid (regular or, more often, irregular) is constructed in the region to be solved. On some elements of this grid (for example, nodes or cells), a so-called grid function is created, in which some physical quantities in grid elements are stored. As a rule, there are two grid functions – at the previous time step and at the current one. The main cycle of the program runs over time; at each time step, the values at the current step are calculated based on the values at the previous step. This cycle is always performed sequentially. At the end of a step, the computed new values are copied from the grid function for the current step to the grid function for the previous step (sometimes these grid functions are simply swapped). Inside the body of the main loop, there is a loop (one or more) over the grid function, in which new values of physical variables are calculated for each value of the grid function index. If the method is explicit (in which the new values of the physical variables depend only on the old ones and do not depend on new values in the neighboring grid elements), then the values in different grid elements can be calculated independently of each other, in particular, these calculations can be carried out in parallel. Thus, inner loops can (and should) be parallelized on shared memory. When calculating on a CPU, parallelization of loops is done, as a rule, using OpenMP. When calculating on CUDA, the parallelization methods are different from OpenMP. To hide the parallelization method from the applied mathematician who implements the numerical method, it is proposed to use the funcprog functional programming library as described below.

5.1 Grid Expressions and Grid Functions

Let us introduce the concept of a grid expression. This is an object defined on all grid elements, that is, for any object that is a grid expression, you can find out what its value is for any grid index. A 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 template is defined for the `grid_expression` class, from which all grid expression classes must inherit (in particular, the `grid_function` class also inherits 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 the Curiously Recurring Template Pattern (CRTP) [6], in which the final class is passed to the base class as a template parameter. You can read about this pattern and other metaprogramming methods in books [7,8]. You can also read about expression templates in [9].

Any grid expression can be assigned to a grid function. This assignment operator iterates over all the indices of the grid function to which the grid expression is assigned. For each index it queries the grid expression for its value and assigns this value to the grid function at the given index. This assignment operator implies that values for different indices can be computed independently of each other, and therefore can be computed in parallel. It is in this assignment operator that the very inner loop over the elements of the grid function is executed. This assignment operator, depending on which compiler the program is compiled with, chooses the method of parallelization of this loop. If it is a compiler for CUDA (the preprocessor variable `__CUDACC__` is defined), then parallelization is carried out using CUDA, otherwise – using OpenMP. Thus, the parallelization method is hidden from the application programmer inside this assignment operator. Let us show how this assignment operator is implemented for the CPU:

```
template<class GEXP>
void operator=(grid_expression(GEXP, typename GEXP::proxy_type) const&
gexp0){
  GEXP const& gexp = gexp0();
#pragma omp parallel for
  for(size_t i = 0; i < size(); ++i)
    (*this)[i] = gexp[i];
}
```

There is one more aspect to mention when talking about grid functions. A GPU can work only with its own memory, which means that when working on a GPU, the grid function must request memory for its data in the CUDA memory. There are no problems with this either. Grid functions are designed in such a way that, when compiled on CUDA, they request memory in CUDA, otherwise, in CPU memory.

5.2 Proxy objects

The `grid_expression` class template is defined as follows:

```
1. template<class E, class _Proxy = E>
2. struct grid_expression;
```

Here `E` is the final class and `_Proxy` is the proxy class. By default (if not specified), the proxy class is the same as the final class. It is needed to create a copy of an object. The only way to transfer parameters from the main processor memory to the CUDA memory is by value transfer (that is, a copy of the parameter is made). Transfer by address and reference is not possible. Only variables of simple types (numbers and pointers) and class objects containing only simple types can be passed by value. In addition, these objects must not contain virtual methods, and the methods called on them must be accessible from CUDA. Not all objects meet all these requirements. If such an object still needs to be transferred from the CPU to CUDA, then a proxy object can be created for it that meets the listed requirements and stores all the basic data from the main object. There is another reason why it is not always possible to store references and pointers to objects, even on the CPU. The fact is that in complex expressions temporary unnamed variables may appear that do not have permanent memory, and references and pointers to which cannot be saved. Such objects must be copied. You cannot always copy objects either, since there are «large» objects (for example, data vectors) that, when copied, make a copy of this data. The general rule is as follows. If the object is «small» and does not have virtual methods, then it can be copied, and the proxy class is not needed, otherwise such a class is needed.

5.3 Grid Expressions as Functors, Applicatives, and Monads

Grid expressions can be thought of as containers (this is especially true for grid functions). In funcprog library, the containers (e.g. lists and the `Maybe` class) are functors, applicatives, and monads. This makes it possible to apply ordinary functions to the values stored in them (a property of a functor). We will also make the grid expression a functor, applicative, and 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 a new value in a cell from the old one. The old value in the cell is passed to it as a parameter. In the new approach, we want to be able to write 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 can write:

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

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

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

something like:

```
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 queried 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 to functors and applicatives. In addition, the sum and difference of two grid expressions, and the product and quotient of the grid expression and numbers, are also grid expressions.

Now let `f` be the function to be applied and `gexp` the grid expression.

Functor. For a functor, we give the following definition (in pseudo-Haskell):

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

Theorem 1 (on the functor). The `fmap` function defined above is functorial.

Proof. Let us rewrite the first functor law completely (without η-reduction)

```
fmap id gexp = id gexp
```

or (by the definition of the `id` function):

```
fmap id gexp = gexp
```

Further,

```
(fmap id gexp)[i] = -- defenition of fmap
  id gexp[i] =      -- definition of id
  gexp[i]
```

that is, indeed, `fmap id gexp = gexp`. The first law is proven. Let us rewrite the second functor law without η-reduction:

```
fmap (g . f) gexp = (fmap g . fmap f) gexp
```

Then

```
(fmap (g . f) gexp)[i] = -- definition of fmap
  (g . f) gexp[i] = -- definition of function composition
  g (f gexp[i])
```

On the other side:

```
((fmap g . fmap f) gexp)[i] = -- definition of function composition
  (fmap g (fmap f gexp))[i] = -- def. of fmap
  g (fmap f gexp)[i] =        -- def. of fmap
  g (f gexp[i])
```

That is, indeed,

```
fmap (g . f) gexp = (fmap g . fmap f) gexp
```

The theorem is proved. The definition of the Functor is correct. □

Applicative. The `pure` function takes on some value and «brings» it into the applicative. In our case, makes a grid expression out of it. Let us define its operator `[]` so that it returns the same value for any index:

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

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

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

Theorem 2 (on the applicative). The `pure` and `apply` functions defined above satisfy applicative laws.

Proof. Let us rewrite the applicative laws using the apply function:

```
apply (pure id) eobj = eobj          -- Identity
apply (pure f) (pure x) = pure (f x) -- Homomorphism
apply u (pure y) = apply (pure ($ y)) u -- Interchange
apply (apply (apply (pure (.)) u) v) x = apply u (apply v x) --
Composition
```

The first law (Identity):

```
(apply (pure id) eobj)[i] = -- def of apply
  (pure id)[i] eobj[i] =    -- def of pure
  id eobj[i] =              -- def of id
  eobj[i]
```

i.e. `apply (pure id) eobj = eobj`. The first law is proved. The second (Homomorphism):

```
(apply (pure f) (pure x))[i] = -- def of apply
  (pure f)[i] (pure x)[i] =    -- def of pure (2 times)
  f x
```

On the other side:

```
(pure (f x))[i] = -- definition of pure
  f x
```

The second law is proved. The third law (Interchange):

```
(apply u (pure y))[i] = -- definition of apply
  u[i] (pure y)[i] =    -- definition of pure
  u[i] y
```

On the other side:

```
(apply (pure ($ y)) u)[i] = -- def of apply
  (pure ($ y))[i] u[i] =    -- def of pure
  ($ y) u[i] =              -- def of function application
  u[i] y
```

The third law is proved. The forth law (Composition):

```
(apply (apply (apply (pure (.)) u) v) x)[i] = -- definition of apply
  (apply (apply (pure (.)) u) v)[i] x[i] = -- definitiono apply
  (apply (pure (.)) u)[i] v[i] x[i] = -- definition of apply
  (pure (.))[i] u[i] v[i] x[i] = -- def of pure
  (.) u[i] v[i] x[i] = -- rewrite function composition in infix form
  (u[i] . v[i]) x[i] = -- definition of function composition
  u[i] (v[i] x[i])
```

On the other side:

```
(apply u (apply v x))[i] = -- def of apply
  u[i] (apply v x)[i] =    -- def of apply
  u[i] (v[i] x[i])
```

The forth law is proved. The theorem is proved. The definition of the Applicative is correct. □

Monad. The monad function `mreturn` is defined in the same way as the `pure` function:

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

The monad function `mbind` takes a monad and a function that takes an ordinary (non-monad) value and returns a monad. Let us define the `mbind` function for grid expressions as follows:

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

Theorem 3 (on the monad). The `mreturn` and `mbind` functions defined above obey monadic laws.

Proof. Let us rewrite the monad laws in terms of the `mreturn` and `mbind` functions:

```
1. mbind (mreturn x) f = f x
2. mbind eobj mreturn = eobj
3. mbind (mbind eobj f) g =
  mbind eobj (\x -> mbind (f x) g)
```

The first law:

```
(mbind (mreturn x) f)[i] = -- def of mbind
  (f (mreturn x)[i])[i] =  -- def of mreturn
  (f x)[i]
```

The first law is proved. The second law:

```
(mbind eobj mreturn)[i] = -- def of mbind
  (mreturn eobj[i])[i] =  -- def of mreturn
  eobj[i]
```

The second law is proved. The third law:

```
(mbind (mbind eobj f) g)[i] = -- def of mbind
  (g (mbind eobj f)[i])[i] =  -- def of mbind
  (g (f eobj[i])[i])[i]
```

On the other side:

```
(mbind eobj (\x -> mbind (f x) g))[i] = -- definition of mbind
  ((\x -> mbind (f x) g) eobj[i])[i] = -- beta-reduction, substitute
eobj[i] instead of x
  (mbind (f eobj[i]) g)[i] = -- def of mbind
  (g (f eobj[i])[i])[i]
```

The result is the same. The third law is proved. The theorem is proved. The definition of the Monad is correct. □

5.4 Grid Expressions as Functors, Applicatives, and Monads

Here is an example program that calculates the `axpy` function from the BLAS library:

```
#include <iostream>
#include <funcprog_data.hpp>

template<typename T>
void axpy(T a, math_vector<T> const& x, math_vector<T> &y){
  mv(y) = _([] __DEVICE __HOST
    (T a, T xi, T &yi, size_t /*i*/){
    yi += a * xi;
  }) / p(a) * mv(x);
}
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
  return 0;
}
```

This program compiles without any modifications for CPU and for CUDA. First, two data vectors of length 10 (`x` and `y`) are created and initialized with initial values. On the CPU, the `math_vector` class is equivalent to the `std::vector` class, and for CUDA it is equivalent to the `thrust::device_vector` class. The `mv` function turns the `math_vector` into a grid function, and the p function calls the `pure` function from the grid expression applicative. The assignment operator in the `axpy` function starts a loop over the grid function `y`, assigning to each of its elements the corresponding element of the grid expression on the right side of the assignment operator. This loop for the CPU is parallelized using OpenMP, and for CUDA – using CUDA. All the specifics of this parallelization are hidden from the application programmer in this assignment operator inside the library.

7. Conclusion

Declarative programming languages, which include functional languages, allow, in contrast to imperative languages (such as C++), to concisely and at the same time clearly enough to write down the desired result without going into implementation details. The specific implementation can be hidden in the language and depend on the current software and hardware environment. The C++ language turned out to be powerful enough to allow the implementation of a functional programming library in it, allowing 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 transferring numerical problems to CUDA graphics accelerators. Grid expressions were defined as functors, applicatives, and monads, allowing functions to be applied to the values stored in them. More information about the C++ language can be found in the sources [10-14].

We are currently using the proposed approach for creating generic code that simulates compressible multicomponent viscous heat-conducting medium. The program is under testing.

Список литературы / References

[1]. TOP500. URL: https://www.top500.org/

[2]. OpenCL. URL: https://www.khronos.org/opencl/

[3]. OpenACC. URL: https://www.openacc.org/

[4]. NVIDIA CUDA. URL: https://developer.nvidia.com/language-solutions

[5]. М.М. Краснов. Библиотека функционального программирования для языка C++. Программирование, том 46, no. 5, 2020 г., стр. 47-59 / M.M. Krasnov. Functional Programming Library for C++. Programming and Computer Software, vol. 46, no. 5, 2020, pp. 330-340.

[6]. J.O. Coplien. Curiously recurring template patterns. C++ Report, vol. 7, issue 2, 1995, pp. 24–27.

[7]. D. Abrahams, A. Gurtovoy. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, 2004, 408 p.

[8]. M.M. Krasnov. C++ templates metaprogramming in problems of mathematical physics. KIAM RAS, 2017, 84 p. DOI: 10.20948/mono-2017-krasnov.

[9]. T. Veldhuizen. Expression Templates. C++ Report, vol. 7, issue 5, 1995, pp. 26-31.

[10]. B. Stroustrup. The C++ Programming Language. Fourth Edition. Addison-Wesley, 2013, 1376 p.

[11]. B. Stroustrup. Programming: Principles and Practice Using C++. Second Edition. Addison-Wesley, 2014, 1312 p.

[12]. B. Stroustrup. A Tour of C++. Addison-Wesley, 2014, 192 p.

[13]. B. Stroustrup. The Design and Evolution of C++. Addison-Wesley, 1994, 480 p.

[14]. The C++ Resources Network. URL: http://www.cplusplus.com/.

Информация об авторах / Information about authors

Михаил Михайлович КРАСНОВ – кандидат физико-математических наук, старший научный сотрудник ИПМ им. М.В. Келдыша РАН, доцент кафедры информатики МФТИ. Сфера научных интересов: языки программирования, метапрограммирование, функциональное программирование, применение технологий программирования к численным методам.

Mikhail Mikhailovich KRASNOV – PhD in Physics and Mathematics, Senior Researcher at KIAM RAS, Associate Professor of the Department of Informatics of MIPT. Research interests: programming languages, metaprogramming, functional programming and application of programming technologies to numerical methods.

Ольга Борисовна ФЕОДОРИТОВА – старший научный сотрудник ИПМ им. М.В. Келдыша РАН. Сфера научных интересов: разработка моделей, методов, алгоритмов и программ для численного решения задач механики сплошной среды, численные моделирование ударно-волновых процессов в неоднородных многофазных средах.

Olga Borisovna FEODORITOVA – Senior Researcher at KIAM RAS. Research interests: development of models, methods, algorithms and programs for the numerical solution of problems in continuum mechanics, numerical simulation of shock-wave processes in inhomogeneous multiphase media.