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

[1] *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.

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 1, вып. 2, 2019 г., стр. 15–19. DOI: 10.15514/ISPRAS–2019–1(2)–1

Благодарности: В этом блоке перечисляются организации, поддерживающие исследование, описанное в статье, гранты и т.д. (стиль ispAnotation)

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

[1] *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

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. 1, issue 2, 2019. pp. 15-19 (in Russian). DOI: 10.15514/ISPRAS-2019-1(2)-1

Acknowledgements. Блок «Благодарности» на английском языке.

1. Введение

В последние годы всё большее распространение получают графические ускорители (GPU), используемые в качестве вычислительных устройств для численных расчётов. Такие ускорители устанавливаются на многих вычислительных кластерах, в частности, в списке TOP500 самых производительных суперкомпьютеров от июня 2021 г. (JUNE 2021) в первой десятке шесть используют графические ускорители от компании NVIDIA [1]. Скорость численных расчётов на таких ускорителях может быть во много раз выше, чем на CPU (по опыту авторов, ускорение может достигать 10-20 раз), поэтому перенос на графические ускорители программ, реализующих численные методы, является чрезвычайно актуальной задачей.

Однако перенос существующей программы на GPU является непростой задачей. Возможно, идеальный вариант – сразу писать программу так, чтобы она могла считаться на любых вычислителях. В любом случае первым встаёт вопрос – какую технологию работы на GPU использовать? В настоящий момент существует три основных технологии – это OpenCL (открытый стандарт для гетерогенных систем) [2], OpenACC [3] и CUDA – разработка компании NVIDIA для своих графических ускорителей [4]. Каждая из этих технологий имеет свои преимущества и недостатки. Главное преимущество OpenCL – открытый стандарт. Программа, использующая OpenCL, будет работать на любом вычислительном устройстве, поддерживающем этот стандарт, в том числе на GPU от NVIDIA и от AMD, процессорах Intel Xeon Phi с технологией Intel MIC и даже на обычных CPU. Главным недостатком этой технологии является то, что исходный код программы возникает в двух экземплярах: для CPU, который компилируется обычным компилятором и является частью основной программы, и текст для OpenCL в отдельных файлах, и при изменениях в алгоритмах правки надо вносить в оба места. Преимущества и недостатки технологии CUDA являются зеркальным отражением недостатков и преимуществ OpenCL. CUDA работает только на GPU от NVIDIA. С другой стороны, в CUDA мы имеем единый исходный текст, который компилируется предварительно и является частью основной программы (в том числе и код, который будет исполняться на GPU). Главный недостаток технология OpenACC в том, что она пока ещё недостаточно распространена. Компилятор, поддерживающий эту технологию, установлен далеко не на всех кластерах с графическими ускорителями. Кроме того, OpenACC подразумевает, что имеются две копии данных: на основном (host) процессоре и на GPU, и OpenACC сама синхронизует эти данные (копирует их в ту или другую сторону). Мы считаем, что это не нужно. Все вычисления должны проводиться на GPU, и копирование данных должно быть только на старте (например, чтобы закачать начальные данные из файла) и в конце, чтобы сохранить их в файл. Причём, если данные инициализируются программно, то копирование возникает только в конце.

Мы выбираем технологию CUDA. Наш главный аргумент состоит в том, что в (нашей) реальной жизни мы сталкиваемся исключительно с устройствами от NVIDIA. GPU от AMD и процессоры Intel Xeon Phi достаточно экзотичны, и хотя нам и встречались, но реально неактуальны. Поэтому недостаток CUDA недостатком для нас не является, а её преимущество остаётся.

Следующая проблема состоит в том, что распараллеливание на общей памяти на CPU и на GPU делается совершенно по-разному. Если мы хотим получить единый текст, который должен компилироваться и для CPU и для CUDA, то в тех местах, где должно быть распараллеливание, придётся писать разный код (например, с помощью конструкции #ifdef), что неудобно. И тут возникла идея воспользоваться ранее написанной одним из авторов статьи библиотекой функционального программирования для языка C++ [5]. При использовании этой библиотеки всю специфику вычислительного устройства (CPU или CUDA) можно поместить внутрь библиотеки, и пользовательский исходный код останется платформонезависимым.

Настоящая статья состоит из трёх основных частей: краткое введение в функциональное программирование (в объёме, необходимом для понимания остального текста), краткое описание библиотеки функционального программирования funcprog и описание применения этой библиотеки для решения численных задач.

2. Краткое введение в функциональное программирование

В функциональном программировании центральным объектом является (как это и следует из названия) функция. Функции являются полноправными участниками вычислительного процесса, такими же, какими при обычных вычислениях являются числа. Это значит, что функция может быть передана как параметр другой функции и может быть возвращена как результат работы функции. Функцию можно вычислить, так же, как при обычных вычислениях можно вычислить число. Простой пример – композиция двух одноместных функций, которая возвращает новую одноместную функцию, вызывающую последовательно обе функции. В специализированных функциональных языках программирования (таких, как Lisp или Haskell) такие возможности встроены в язык, в то время, как реализация композиции функций на языке C++ является нетривиальной задачей, требующей специальных ухищрений. Примеры будут приводится на языке Haskell, так как этот язык позволяет записывать многие вещи максимально кратко и в то же время понятно.

2.1 Принципы функционального программирования

Функциональное программирование имеет ряд особенностей по сравнению с императивным программированием, которые можно сформулировать в виде нескольких принципов. Некоторые из этих принципов являются обязательными для функционального программирования и поддерживаются всеми языками и библиотеками функционального программирования, а другие – опциональными, то есть в некоторых языках и библиотеках функционального программирования могут отсутствовать. По тому, насколько полно эти принципы реализованы в языке или библиотеке функционального программирования, можно судить о степени её «функциональности».

Первый и главный обязательный принцип, уже упоминавшийся выше – это то, что функция является полноценным участником вычислительного процесса и может быть как передана в качестве параметра, так и возвращена как результат работы некоторой функции.

Другой обязательный принцип – наличие лямбда-выражений. Лямбда-выражение – это такое выражение в языке, результатом которого является функция. Собственно, первый принцип без второго практически невозможен. Как правило, функция, возвращающая в качестве результата функцию, фактически возвращает лямбда-выражение.

Остальные принципы функционального программирования не столь важны и часто отсутствуют, но их наличие существенно повышает возможности языка или библиотеки. Опишем основные из них.

«Чистые» функции. Под «чистотой» функции в функциональном программировании подразумевается отсутствие у функции побочных эффектов. Это значит, что результат, возвращаемый функцией, зависит только от переданных аргументов и больше ни от чего.

Следующий принцип функционального программирования – **неизменяемые (immutable) переменные**. Это как если бы в Вашей программе на C++ все переменные имели бы модификатор const (`const int i = 5;`). Переменные есть, но им что-то присвоить можно только один раз при создании. Именно с такими переменными работают все функциональные языки программирования (Haskell, Lisp). Этот принцип позволяет гарантировать «чистоту» функции (функция не меняет значение ни одной переменной (потому что не может), значит, она «чистая»).

Каррирование. Названо по фамилии американского математика и логика Хаскелла Карри (а по его имени назван язык программирования Haskell). Принцип каррирования состоит в том, что при неполном указании параметров функции ошибки не происходит, а вместо этого генерируется функция с меньшим (равным числу недостающих параметров) числом параметров. Реализовано в многих современных функциональных языках программирования (в частности, в языке Haskell). Каррирование позволяет функцию с несколькими аргументами рассматривать как набор функций с одним аргументом.

Ленивые вычисления. Этот принцип в языке Haskell является прямым следствием принципа каррирования. В языке Haskell каррирование «идёт до конца». Это значит, что даже при передаче всех параметров функции фактического вызова не происходит. При применении каждого следующего параметра порождается функция с меньшим на единицу числом параметров, и после применения всех параметров получается функция без параметров. Именно эта функция без параметров передаётся в качестве аргумента. То есть фактически любые вычисления в языке Haskell – это вычисления функций.

η-редукция (эта редукция, или η-преобразование). Пусть мы хотим написать функцию с одним параметром – списком чисел, возвращающую список синусов этих чисел. Текст этой функции на языке Haskell очевидный:

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

Из принципа каррирования следует, что если мы опустим второй параметр при вызове функции `map sin lst`, то есть напишем просто `map sin`, то мы получим функцию с одним параметром, принимающим в качестве этого параметра список чисел и возвращающую список синусов этих чисел, то есть фактически функцию `mapsin`. То есть `mapsin` эквивалентно `map sin`. Принцип η-редукции гласит, что в подобных случаях последний параметр в определении функции можно опускать, то есть определение функции `mapsin` можно записать короче:

```
mapsin=map sin
```

Композиция функций. Композиция функций настолько важна в функциональном программировании, что в языке Haskell эта операция максимально упрощена. Обычно рассматривают композицию одноместных функций (назовём их f и g), в языке Haskell она записывается так:

```
(f . g) x = f (g x)
map (exp . sin) [1,2,3] -- Пример использования
```

2.2 Функторы, аппликативы и монады

Функторы. Пусть у нас есть некоторый контейнер, хранящий какое-то количество значений, например, список или объект класса `Maybe`. Теперь поставим задачу: применить обычную одноместную функцию (например, sin) к значениям в контейнере. Как это сделать со списками, известно – применить функцию `map`. Но как это сделать с типом `Maybe`, и в общем случае – как это сделать с данными в произвольном контейнере? Универсальный подход состоит в том, чтобы доверить это ответственное дело самому контейнеру. Для этого в языке Haskell определён специальный класс `Functor`, в котором продекларирована функция `fmap`:

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

Оператор `(<$>)` – синоним функции `fmap`. Это оператор применения функции к функтору. Он похож на оператор применения функции к обычному значению `($)`. Прототип функции `fmap` можно записать в другом эквивалентном виде (это следует из правоассоциативности стрелки вправо):

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

Функтором называется тип, реализующий класс `Functor`. Таким образом, функцию `fmap` можно рассматривать как функцию с одним параметром, принимающим функцию, принимающую и возвращающую обычные значения, и преобразующую её в функцию, принимающую и возвращающую функторы. Любой тип данных может объявить себя функтором, реализовав для себя экземпляр класса `Functor` и функцию `fmap`. Любая реализация функтора должна удовлетворять двум функторным законам:

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

Здесь `id` – функция, возвращающая свой аргумент: `id x=x`. Первый закон гласит: применение функции id к функтору не должно менять функтор, так же, как применение этой функции к обычному значению его не меняет. Второй закон – это распределительный закон функторной операции относительно композиции функций.

Для списков и типа данных `Maybe` функтор реализован так:

```
instance Functor [] where
  fmap = map -- Это логично

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

Аппликативы. Если стоит задача применить функцию с двумя аргументами к двум контейнерам (например, просуммировать два списка), то функционала класса `Functor` будет недостаточно. Для решения этой задачи предназначен другой класс – аппликативный функтор (аппликатив). Вот определение класса `Applicative`:

```
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
```

Таким образом, в каждом аппликативе должны быть реализованы две основные операции: функция `pure`, помещающая обычное значение в «чистый» аппликатив, и оператор `(<*>)`,

принимающий в качестве первого параметра функцию, помещённую в аппликатив, и второго параметра – значение, помещённое в тот же аппликатив, и возвращающий результат в том же аппликативе.

Если мы посмотрим на прототип и реализацию функции `liftA2`, то мы увидим, что она передаёт функцию с двумя параметрами в оператор `(<$>)`, который принимает функцию с одним параметром. Но противоречия тут нет, так как функцию с прототипом `a->b->c` мы можем записать так: `a->(b->c)`, то есть как функцию с одним параметром, возвращающую функцию. Тогда оператор `(<$>)` нам как раз и вернёт функцию `b->c`, помещённую в функтор, которая затем передаётся в оператор `(<*>)`. По аналогии с функцией fmap, прототип функции `liftA2` мы можем записать так:

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

то есть рассматривать её как функцию с одним аргументом, принимающую функцию, работающую с обычными значениями и возвращающую функцию, работающую с функторами, «поднимающую» функцию с двумя аргументами (отсюда и её название) в аппликатив. По аналогии с функцией `liftA2` можно написать функцию, «поднимающую» в аппликатив функцию с тремя (и любым другим числом) аргументами:

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

Любая реализация аппликатива должна удовлетворять аппликативным законам:

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

Вот как реализованы аппликативны для списков и `Maybe`:

```
instance Applicative [] where
  pure x    = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]

instance Applicative Maybe where
  pure = Just -- η-редукция
  Just f  <*> m = fmap f m
  Nothing <*> _ = Nothing
```

Монады. Напишем «безопасные» функции safe_sqrt и safe_log:

```
safe_sqrt x=if x<0 then Nothing else Just(sqrt x)
safe_log x=if x<=0 then Nothing else Just(log x)
```

Эти функции возвращают результат типа `Maybe`, причём если аргумент функции принадлежит области её определения, то возвращается значение `Just value`, а иначе – `Nothing`. Пусть теперь мы хотим вычислить квадратный корень от логарифма числа. Для обеих операций у нас есть «безопасные» функции, возвращающие результат типа `Maybe` и проверяющие, что значение аргумента принадлежит области определения функции. Как нам теперь применить функцию `safe_sqrt` к результату функции `safe_log`? Функционала функтора и аппликатива для этого недостаточно. Для этой цели служат монады. Монады можно рассматривать как дальнейшее продолжение аппликатива, они предназначены для построения цепочек монадных вычислений. Каждая монада имеет две основных функции: `mreturn` (в языке Haskell `return`) и `mbind` (в языке Haskell оператор `(>>=)`. Функция `mreturn` аналогична функции `pure` для аппликативов (фактически для большинства монад `mreturn` определяется как `pure`), а операция `mbind` имеет следующее определение:

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

Она принимает в качестве параметров монаду и функцию, принимающую обычное (не монадное) значение и возвращающую монадное значение (возможно, другого типа, но в той же монаде). Будем называть такие функции монадными. Функции `safe_log` и `safe_sqrt` – примеры монадных функций.

Функции `mreturn` и `mbind` должны удовлетворять трём монадным законам. Для того чтобы их сформулировать, введём операцию монадной композиции (`mcompose` или оператор `(>=>)` в языке Haskell). Она определяется следующим образом:

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

Оба операнда монадной композиции и её результат – монадные функции. Следовательно, монадную композицию можно рассматривать как групповую операцию в пространстве таких функций. В терминах этой групповой операции монадные законы формулируются так:

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

Другими словами, функции `mreturn` и `mbind` должны быть определены так, чтобы, во-первых, функция `mreturn` являлась единичным элементом (левым и правым) монадной композиции (первые два закона), и, во-вторых, монадная композиция должна быть ассоциативной (третий закон). Покажем, как определена монадная операция для списков и `Maybe`:

```
instance Monad [] where
  xs >>= f = [y | x <- xs, y <- f x]

instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= _ = Nothing
```

Теперь квадратный корень от логарифма можно вычислить так:

```
safe_log   5  >>= safe_sqrt -- Just 1.2686362411795196
safe_log (-5) >>= safe_sqrt -- Nothing
safe_log 0.5  >>= safe_sqrt -- Nothing
```

Если где-то в цепочке вычислений возникает ошибка, то происходит быстрый выход из всей цепочки, остальные функции фактически не вычисляются.

3. Библиотека функционального программирования

3.1 Общее описание библиотеки

При реализации библиотеки функционального программирования для языка C++ funcprog ставилась задача написать библиотеку, с помощью которой на языке C++ можно было бы писать в стиле, близком к стилю языка Haskell.

Важный вопрос – что такое функция с точки зрения этой библиотеки? В первоначальной версии библиотеки под функцией подразумевался объект класса `std::function`. Этот вариант нас теперь не устраивает, так как мы хотим, чтобы функция исполнялась на графическом ускорителе, а объект класса `std::function` может исполняться только на CPU (в частности, потому, что в её реализации используются виртуальные функции, которые на GPU непереносимы). Это не может быть и обычная функция, так как её невозможно передать как параметр из CPU в CUDA, так как обычную функцию можно передать только по адресу, а адреса из CPU в CUDA передавать нельзя. Было принято решение считать функцией любой

функциональный объект (имеющий функциональный оператор `()`), в частности, это может быть лямбда-выражение. При этом для того, чтобы этот объект можно было передавать в CUDA, этот функциональный оператор должен быть помечен ключевым словом `__device__`. Недостатком такого подхода является то, что в метаданные функции попадает не только её прототип (типы параметров и возвращаемое значение), но и реализация (класс объекта) – такова плата за возможность переноса на CUDA. Вот как реализована функция:

```
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;
};
```

Видно, что, в отличие от класса `std::function`, в класс `function2` передаются два параметра шаблона. Для упрощения работы с такими функциями имеется вспомогательная функция `_` (подчерк). Вот как она определена:

```
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; }
```

Приведём пример работы с этой библиотекой:

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

В этом примере мы создали две функции (с помощью функции `_`), сделали их композицию (с помощью оператора `&`) и вызвали получившуюся составную функцию с параметром 5. В результате получили число 36.

В библиотеке funcprog достаточно полно реализовано каррирование функций. Для этого в библиотеке реализован оператор применения аргумента к функции с помощью оператора сдвига влево `<<`. При этом создаётся новая функция, имеющая на один параметр меньше. В частности, если исходная функция имела единственный параметр, то создастся функция без параметров. В библиотеке имеется также функция `invoke_f0`, которой можно передать любую функцию. Если переданная функция без параметров, то она будет вызвана и вернётся результат её исполнения. Если же переданная функция имеет параметры (один или больше), то будет просто возвращена эта функция. В библиотеке имеется также метафункция

`remove_f0`, принимающая в качестве параметра шаблона тип функции. Если функция без параметров, то в переменной `type` вернётся тип возвращаемого функцией значения, а если параметры есть, то тип самой функции.

3.2 Реализация функторов, аппликативов и монад

Реализация функторов, аппликативов и монад в библиотеке `funcprog` в чём-то похожа на реализацию этих понятий в языке Haskell. Любой класс может объявить себя функтором, аппликативом или монадой. Для этого достаточно для этого класса реализовать специализацию классов соответственно `Functor`, `Applicative` и `Monad`. В сам класс никаких изменений вносить не требуется.

Любой функтор или монада являются типом с одним параметром. В языке C++ типы с параметром реализуются с помощью шаблонов классов. Рассмотрим определение функтора на примере класса `Maybe`. Класс `Maybe` в библиотеке `funcprog` определён так:

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

Шаблон класса типом не является, и его нельзя передать как параметр шаблона другого класса. Поэтому определяется ещё один класс (без шаблона) с именем `_Maybe` (с подчерком впереди). Это уже настоящий класс, его можно передавать как параметр шаблона:

```
struct _Maybe {};
```

Шаблон класса `Maybe` наследуется от этого класса:

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

Специализации классов `Functor`, `Applicative` и `Monad` пишутся именно от этого класса `_Maybe`:

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

Внутри специализации класса `Functor` нужно определить статическую функцию `fmap`. Специализации классов `Applicative` и `Monad` определяются аналогично. Для аппликатива методы называются `pure` и `apply`, а для монады – `mreturn` и `mbind`. При реализации статических методов этих классов нужно не забывать про выполнение функторных, аппликативных и монадных законов.

Заметим также, что в библиотеке `funcprog` в качестве функторного оператора используется оператор деления, а в качестве аппликативного – умножение.

4. Применение библиотеки для численных методов.

При решении численных задач (например, задач газовой динамики или задачи теплопроводности) в решаемой области строится некоторая сетка (регулярная или чаще нерегулярная). На некоторых элементах этой сетки (например, узлах или ячейках) создаётся так называемая сеточная функция, в которой хранятся некоторые физические величины в элементах сетки. Сеточных функций, как правило, две – на предыдущем шаге по времени и на текущем. Основной цикл работы программы идёт по времени, на каждом шаге по времени по значениям на предыдущем шаге вычисляются значения на текущем шаге. Этот цикл всегда выполняется последовательно. В конце шага вычисленные новые значения копируются из сеточной функции для текущего шага в сеточную функцию на предыдущем шаге (иногда эти

сеточные функции просто меняются местами). Внутри тела основного цикла имеется цикл (один или несколько) по сеточной функции, в котором для каждого значения индекса сеточной функции вычисляются новые значения физических переменных. Если метод явный (при котором новые значения физических переменных зависят только от старых и не зависят от новых значений в соседних элементах сетки), то вычислять значения в разных элементах сетки можно независимо друг от друга, в частности, эти вычисления можно вести параллельно. Таким образом, внутренние циклы можно (и нужно) распараллеливать на общей памяти. При расчётах на CPU распараллеливание циклов делается, как правило, с помощью OpenMP. При расчётах на CUDA методы распараллеливания свои, и они сильно отличаются от OpenMP. Чтобы скрыть метод распараллеливания от прикладного программиста-математика, реализующего численный метод, предлагается использовать библиотеку функционального программирования `funcprog` так, как это описывается далее.

4.1 Сеточные выражения и сеточные функции

Введём понятие сеточного выражения. Это объект, определённый на всех элементах сетки, то есть у любого объекта, являющегося сеточным выражением, можно узнать, чему равно его значение для любого индекса сетки. Частным случаем сеточного выражения является сеточная функция, которая свои значения просто хранит в памяти и их, если нужно, возвращает. Для сеточных выражений определяется шаблон класса `grid_expression`, от которого должны наследоваться все классы объектов, являющихся сеточными выражениями (в частности, класс `grid_function` также пронаследован от класса `grid_expression`). Таким образом, фраза «объект является сеточным выражением» означает, что класс этого объекта пронаследован от класса `grid_expression`. При таком наследовании используется шаблон проектирования CRTP (Curiously Recurring Template Pattern) [6], при котором в базовый класс в качестве параметра шаблона передаётся конечный класс. Про этот шаблон и другие методы метапрограммирования можно прочитать в книгах [7,8]. Про шаблоны выражений можно также прочитать в [9].

Любое сеточное выражение можно присвоить сеточной функции. Этот оператор присваивания проходит по всем индексам сеточной функции, которой присваивается сеточное выражение, для каждого индекса запрашивает у сеточного выражения его значение и присваивает это значение сеточной функции по данному индексу. Этот оператор присваивания подразумевает, что значения для разных индексов можно вычислять независимо друг от друга, и значит их можно вычислять параллельно. Именно в этом операторе присваивания выполняется тот самый внутренний цикл по элементам сеточной функции. Метод распараллеливания этого цикла выбирается оператором присваивания в зависимости от того, каким компилятором компилируется программа. Если это компилятор для CUDA (определена переменная препроцессора `__CUDACC__`), то распараллеливание осуществляется с помощью CUDA, иначе – с помощью OpenMP. Таким образом, метод распараллеливания скрыт от прикладного программиста внутри этого оператора присваивания. Покажем, как примерно реализован этот оператор присваивания для 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_i I = 0; i < size(); ++i)
    (*this)[i] = gexp[i];
}
```

Говоря про сеточные функции, нужно упомянуть ещё один аспект. GPU может работать только со своей памятью, это значит, что при работе на GPU сеточная функция должна память под свои данные запрашивать в памяти CUDA. С этим также проблем нет. Сеточные

функции устроены так, что при компиляции на CUDA они запрашивают память в CUDA, иначе – в памяти CPU.

4.2 Объекты-заместители (proxy)

Шаблон класса сеточного выражения `grid_expression` определён следующим образом:

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

Здесь `E` – конечный класс, а `_Proxy` – класс заместителя. По умолчанию (если он не указан), класс заместителя совпадает с конечным классом. Он нужен для создания копии объекта. Дело в том, что единственный способ передачи параметров из памяти основного процессора в память CUDA – это передача по значению (то есть делается копия параметра). Передача по адресу и ссылке невозможна. По значению можно передавать только переменные простых типов (числа и указатели) и объекты классов, содержащие только простые типы. Кроме того, эти объекты не должны содержать виртуальных методов и вызываемые в них методы должны быть доступны из CUDA. Далеко не все объекты удовлетворяют всем этим требованиям. Если же такой объект всё-таки нужно передать из CPU в CUDA, то для него можно создать объект-заместитель, удовлетворяющий перечисленным требованиям и хранящим все основные данные из основного объекта. Есть и другая причина того, почему не всегда можно хранить ссылки и указатели на объекты даже на CPU. Дело в том, что в сложных выражениях могут появляться временные безымянные переменные, у которых нет постоянной памяти, и ссылки и указатели на которые сохранять нельзя. Такие объекты необходимо копировать. Всегда копировать объекты тоже нельзя, так как бывают «большие» объекты (например, вектора данных), которые при копировании делают копию этих данных. Общее правило следующее. Если объект «маленький» и не имеет виртуальных методов, то его, как правило, можно копировать, и класс-заместитель не нужен, иначе такой класс необходим.

4.3 Сеточные выражения как функторы, аппликативы и монады

Сеточные выражения можно рассматривать как контейнеры (особенно это справедливо для сеточных функций). В библиотеке `funcprog` контейнеры (списки и класс `Maybe`) являются функторами, аппликативами и монадами. Это даёт возможности применять к значениям, хранящимся в них, обычные функции (свойство функторов). Сделаем сеточное выражение также функтором, аппликативом и монадой, чтобы и к сеточным выражениям можно было применять функции. Чтобы понять, как это можно сделать, рассмотрим типичный цикл, вычисляющий новое значение сеточной функции по старой:

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

здесь `calculate` – функция, вычисляющая новое значение в ячейке по старому. Ей в качестве параметра передаётся старое значение в ячейке. В новом подходе мы хотим, чтобы в этом случае можно было написать:

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

Если же для вычислений требуются ещё несколько сеточных функций (назовём их `f2` и `f3`), то вместо

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

мы могли бы написать:

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

то есть для первой сеточной функции мы применили свойство функтора, а для последующих – аппликатива. Если мы хотим в функцию передать ещё дополнительно некоторое постоянное значение (не зависящее от индекса цикла), то чтобы можно было бы написать вместо:

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

что-то типа:

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

Таким образом, результатом применения функции к сеточному выражению (или к нескольким сеточным выражениям в случае аппликатива) должно быть также сеточное выражение, то есть у него можно запросить значение по индексу (должен быть реализован оператор `[]`). Сеточными выражениями, помимо сеточных функций, являются также результаты применения функций к сеточным выражениям как к функторам и монадам. Кроме того, сумма и разность двух сеточных выражений, а также произведение и частное сеточного выражения и числа, также являются сеточными выражениями.

Пусть теперь f – применяемая функция, а gexp – сеточное выражение.

Функтор. Для функтора мы даём следующее определение (на псевдо-Haskell-е):

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

Теорема 1 (о функторе). Определённая выше функция `fmap` удовлетворяет функторным законам.

Доказательство. Перепишем первый функторный закон полностью (без η-редукции):

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

или (по определению функции id):

```
fmap id gexp = gexp
```

Далее,

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

то есть, действительно, `fmap id` gexp=gexp. Первый закон доказан. Перепишем второй функторный закон без η-редукции:

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

Тогда

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

С другой стороны:

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

то есть, действительно, `fmap (g . f) gexp = (fmap g . fmap f) gexp`. Теорема доказана. Определение функтора корректно. □

Аппликатив. Функция `pure` принимает некоторое значение и «вносит» его в аппликатив. В нашем случае делает из него сеточное выражение. Определим его оператор [] так, чтобы он для любого индекса возвращал одинаковое значение val:

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

Функция `apply` (аналог оператора `<*>` в языке Haskell и оператора * в библиотеке `funcprog`) в нашем случае принимает два сеточных выражения: первый (назовём его `gexp_f`) возвращает функции, а второй (назовём его `gexp`) – некоторые значения (параметры этих функций). Определим сеточное выражение функции `apply` следующим образом:

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

Теорема 2 (об аппликативе). Определённые выше функции `pure` и `apply` удовлетворяют аппликативным законам.

Доказательство. Перепишем аппликативные законы с использованием функции `apply`:

```
apply (pure id) gexp = gexp                  -- 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
```

Первый закон (Identity):

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

то есть `apply (pure id) gexp = gexp`. Первый закон доказан.

Второй закон (Homomorphism):

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

С другой стороны:

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

Второй закон доказан. Третий закон (Interchange):

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

С другой стороны:

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

Третий закон доказан. Четвёртый закон (Composition):

```
(apply (apply (apply (pure (.)) u) v) x)[i] = -- definition of apply
  (apply (apply (pure (.)) u) v[i] x[i] =     -- definition of apply
  (apply (pure (.)) u)[i] v[i] x[i] =         -- definition of apply
  (pure (.))[i] u[i] v[i] x[i] =              -- definition 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])
```

С другой стороны:

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

Четвёртый закон доказан. Теорема доказана. Определение аппликатива корректно. □

Монада. Монадная функция `mreturn` определена так же, как и функция `pure`:

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

Монадная функция `mbind` (в языке Haskell и в библиотеке funcprog оператор `>>=`) принимает монаду (в нашем случае сеточное выражение) и функцию, принимающую обычное (не монадное) значение и возвращающую монаду (сеточное выражение). Определим функцию `mbind` следующим образом:

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

Теорема 3 (о монаде). Определённые выше функции `mreturn` и `mbind` удовлетворяют монадным законам.

Доказательство. Перепишем монадные законы в терминах функций mreturn и mbind:

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

Первый закон:

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

Первый закон доказан. Второй закон:

```
(mbind gexp mreturn)[i] = -- definition of mbind
  (mreturn gexp[i])[i]  = -- definition of mreturn
  gexp[i]
```

Второй закон доказан. Третий закон:

```
(mbind (mbind gexp f) g)[i] = -- definition of mbind
  (g (mbind gexp f)[i])[i]  = -- definition of mbind
  (g (f gexp[i])[i])[i]
```

С другой стороны:

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

Результат получился одинаковый. Третий закон доказан. Теорема доказана. □

4.4 Пример программы

Приведём пример программы, вычисляющий функцию axpy из библиотеки BLAS:

```
#include <iostream>

#include <funcprog_data.hpp>

using namespace _KIAM_MATH;
using namespace _FUNCPROG2;

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;
}
```

Эта программа компилируется без каких-либо изменений для CPU и для CUDA. Вначале создаются два вектора данных длиной 10 (x и y) и инициализируются начальными значениями (2 для x и 3 для y). Для CPU класс `math_vector` эквивалентен классу `std::vector` (данные размещены в памяти основного процессора), а для CUDA эквивалентен классу `thrust::device_vector` (данные размещены в памяти CUDA). Функция mv превращает вектор в сеточную функцию, а функция p вызывает функцию `pure` из аппликатива сеточного выражения. Оператор присваивания в функции `axpy` запускает цикл по сеточной функции y, присваивая каждому её элементу соответствующий элемент сеточного выражения в правой части оператора присваивания. Этот цикл для CPU цикл распараллеливается с помощью OpenMP, а для CUDA – с помощью CUDA. Вся специфика этого распараллеливания спрятана от прикладного программиста в этом операторе присваивания внутри библиотеки.

Заключение.

Декларативные языки программирования, к которым относятся и функциональные языки, позволяют, в отличие от императивных языков, к которым относятся большинство языков программирования, на которых реализуются численные методы, кратко и в то же время достаточно ясно записывать желаемый результат, не вдаваясь в подробности реализации. Конкретная реализация может быть скрыта в языке и зависеть от текущего программно-аппаратного окружения. Язык C++ оказался достаточно мощным, чтобы позволить реализовать на нём библиотеку функционального программирования, позволяющую писать

программы в стиле, близком к стилю чисто функциональных языков, таких, как Haskell. Такие понятия из мира функционального программирования, как функторы и монады, реализованные в библиотеке функционального программирования, оказались очень удобным средством для переноса численных задач на графические ускорители CUDA. Сеточные выражения были определены как функторы, аппликативы и монады, что позволило применять функции к хранящимся в них значениям. Сами эти функции можно строить, комбинируя сложные функции из простых, что является также сильной стороной функционального программирования. Авторам представляется, что за функциональным программированием будущее технологии программирования вообще и решения численных задач в частности. Мы надеемся, что данная работа будем нашим вкладом в этом направлении. С дополнительной информацией о языке C++ можно ознакомиться в источниках [10-16].

Литература:

[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]. Краснов М.М. Библиотека функционального программирования для языка C++. Программирование, 2020 г., № 5, с. 47-59. DOI: 10.31875/S0132347420050040
[6]. J.O. Coplien. Curiously recurring template patterns. C++ Report, February 1995, pp. 24–27.
[7]. David Abrahams, Aleksey Gurtovoy. *C++ Template Metaprogramming*. Addison-Wesley, 2004, 400 с. ISBN 978-0-321-22725-6.
[8]. Краснов М.М. Метапрограммирование шаблонов C++ в задачах математической физики. М.: ИПМ им. М.В. Келдыша, 2017. 84 с. DOI: 10.20948/mono-2017-krasnov
[9]. T. Veldhuizen, Expression Templates. C++ Report, Vol. 7 № 5, June 1995, pp. 26-31.
[10]. Bjarne Stroustrup. The C++ Programming Language, Fourth Edition. Addison-Wesley, 2013. ISBN 978-0-321-56384-2, 1368 с.
[11]. Bjarne Stroustrup. Programming: Principles and Practice Using C++, Second Edition. Addison-Wesley, 2013, 1312 с. ISBN 978-0-321-99278-9.
[12]. Bjarne Stroustrup. A Tour of C++. Addison-Wesley, 2014, 192 с. ISBN 978-0-321-95831-0.
[13]. Bjarne Stroustrup. The Design and Evolution of C++. Addison-Wesley, 1994, 480 с. ISBN 978-0-201-54330-8.36
[14]. Бьерн Страуструп. Программирование: Принципы и практика с использованием C++, второе издание. пер. с англ., Вильямс, 2016, 1328 с. ISBN 978-5-8459-1949-6, 978-0-321-99278-9.
[15]. Бьерн Страуструп. Дизайн и эволюция языка C++. ДМК Пресс, 2016, 446 с. ISBN 978-5-97060-419-9, 978-0-201-54330-8.
[16]. 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 Института прикладной математики им. М.В. Келдыша РАН с 2014 года, доцент кафедры информатики Московского физико-технического института с 2017 года. 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 the Keldysh Institute of Applied Mathematics of the RAS since 2014, Associate Professor of the Department of Informatics of Moscow Institute of Physics and Technology. Research interests: programming languages, metaprogramming, functional programming and application of programming technologies to numerical methods.

Ольга Борисовна ФЕОДОРИТОВА – старший научный сотрудник Института прикладной математики им. М.В. Келдыша РАН с 2008 года. 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 the Keldysh Institute of Applied Mathematics of the RAS since 2008. 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.