ПРОГРАММИРОВАНИЕ, *2024*, *№ 1*, *с. 15–29*

УДК 519.6

ПРИМЕНЕНИЕ БИБЛИОТЕКИ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ ДЛЯ РАСПАРАЛЛЕЛИВАНИЯ ВЫЧИСЛЕНИЙ НА CUDA

М. М. Краснов[a],[*] (ORCID: 0000-0001-7988-6323),
О. Б. Феодоритова[a],[**] (ORCID: 0000-0002-2792-9376)

[a]*Институт прикладной математики им. М.В. Келдыша Российской академии наук, 125047 Москва, Миусская пл., д. 4*

**E-mail: kmm@kiam.ru*
***E-mail: feodor@kiam.ru*

Поступила в редакцию 15.02.2023
После доработки: 28.02.2023
Принята к публикации: 10.04.2023

Современные графические ускорители (GPU) позволяют существенно ускорить выполнение численных задач. Однако перенос программ на графические ускорители является непростой задачей, иногда требующей практически полного их переписывания. Графические ускорители CUDA, благодаря разработанной компанией NVIDIA технологии, позволяют иметь единый исходный код как для обычных процессоров (CPU), так и для CUDA. Однако распараллеливание на общей памяти все равно делается по-разному и его нужно указывать явно. 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++; функциональное программирование; численные методы; CUDA; OpenMP; OpenCL
DOI: 10.31857/S0132347424010027 **EDN:** HVJUIU

1. ВВЕДЕНИЕ

В последние годы все большее распространение получают графические ускорители (GPU), используемые в качестве вычислительных устройств для численных расчетов. Такие ускорители устанавливаются на многих вычислительных кластерах. Хотя в списке TOP500 самых производительных суперкомпьютеров от июня 2022 г. (JUNE 2022) [1] графические ускорители от компании NVIDIA [2] уступили пальму первенства технологиям от других производителей, в течение многих лет они были в числе первых. Что касается самых высокопроизводительных суперкомпьютеров России, по состоянию на март 2022 г. [3], практически все они оснащены ускорителями от NVIDIA. Скорость численных расчетов 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 главным встает вопрос — какую технологию работы на GPU использовать? В настоящий момент существует три основных технологии — это OpenCL (открытый стандарт для гетерогенных систем) [4], OpenACC [5] и CUDA [6] — разработка компании NVIDIA для своих графических ускорителей. 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 часто возникает в двух экземплярах: для CPU, который компилируется обычным компилятором и является частью основной программы, и текст для OpenCL в отдельных файлах. При изменениях в алгоритмах правки надо будет 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 технологии OpenACC в том, что она пока еще недостаточно распространена. Компилятор, поддерживающий эту технологию, установлен далеко не на всех кластерах с графическими ускорителями.

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, реально неактуальны. Поэтому недостаток CUDA недостатком для нас не является, а ее преимущество остается.

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, придется 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++ [7]. При использовании этой библиотеки всю специфику вычислительного устройства (CPU или CUDA) можно поместить внутрь библиотеки, и пользовательский исходный код останется платформонезависимым. С другими аналогичными работами авторов можно также ознакомиться в работах [8] и [9].

Настоящая работа состоит из трех основных частей: краткое введение в функциональное программирование (в объеме, 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 (иногда функции, принимающие в качестве параметров другие функции, называют функциями высшего порядка). 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 Haskell [10]) такие возможности встроены в язык, в то время как реализация композиции функций на языке C++ является нетривиальной задачей, требующей специальных ухищрений. Примеры будут приводиться 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 ее “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. В частности, в современном стандарте языка C++ (начиная с версии C++11) лямбда-выражения имеются. Остальные принципы функционального программирования не столь важны и часто отсутствуют, но их наличие суще-

ственно 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** (**int const** 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, то есть то, что ее результат зависит только от ее параметров, и при одном и том же наборе параметров она всегда возвращает одно и то же. Функция не меняет значение ни одной переменной (потому что не может), значит, она “чистая”.

Каррирование. Названо по фамилии американского математика и логика Хаскелла Карри. Принцип каррирования состоит в том, что при неполном указании параметров функции ошибки не происходит, а вместо этого генерируется функция с меньшим (равным числу недостающих) числом параметров. Реализовано во 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 “идет 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 передается в качестве аргумента. То есть фактически любые вычисления в языке Haskell — это вычисления функций.

η-редукция (эта редукция, или η-преобразование). Рассмотрим пример. 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
```

В этом примере последний параметр функции mapsin передается как последний параметр в правой части. 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 (назовем их f и g), в языке Haskell она записывается так:

```
(f . g) x = f (g x)
map (exp . sin) [1,2,3] — пример
```

2.2. Математические основы функционального программирования

В основе функционального программирования (в частности языка Haskell) лежит современная математическая теория — теория категорий. Подробно с этой теорией можно ознакомиться, например, по источникам [11] и [12]. **Категорией** называется совокупность объектов, снабженных стрелками (морфизмами) между ними (некоторыми из них). Между двумя заданными объектами может быть много стрелок. Совокупность стрелок из объекта A в объект B в категории **C** обозначается $Hom_{\mathbf{C}}(A, B)$ или просто $Hom(A, B)$, если конкретная категория подразумевается.

Для стрелок имеются два обязательных условия. Во-первых, из каждого объекта должна существовать стрелка в него самого (“единичная” стрелка, или тождественный морфизм). Тождественный морфизм объекта A обозначается id_A. Таким образом, для любого объекта A $Hom(A, A)$ непусто (всегда имеется тождественный морфизм). Во-вторых, для любых двух стрелок $f \in Hom(A,B)$ и $g \in Hom(B,C)$ должна существовать их композиция $g \circ f \in Hom(A,C)$. Для существования композиции морфизмов важно, чтобы конец первой (правой) стрелки f совпадал с началом второй (левой) стрелки g. Обратите внимание, что первая стрелка указывается справа от оператора композиции, а вторая — слева. Композицию стрелок $g \circ f$ можно читать как “g после f”.

Морфизмы, в которых начало и конец совпадают (морфизмы из некоторого объекта в него самого), называются **эндоморфизмами**. Множество эндоморфизмов объекта A: $End(A) = Hom(A, A)$ является **моноидом** относительно операции композиции с еди-

ничным элементом id_A. Напомним, что моноидом называется множество с определенной на нем бинарной ассоциативной операцией и нейтральным (единичным) элементом относительно этой операции.

Функторами называются отображения категорий, сохраняющие структуру исходной категории. Точнее, функтор $F : \mathbf{C} \to \mathbf{D}$ ставит в соответствие каждому объекту в категории **C** объект в категории **D** и каждому морфизму $F: A \to B$ в категории **C** морфизм $F(f): F(A) \to F(B)$ в категории **D** так, что

- $F(id_A) = id_{F(A)}$

и для двух морфизмов $f: A \to B$ и $g: B \to C$ в категории **C**

- $F(g) \circ F(f) = F(g \circ f)$.

Функторы из категории **C** в категорию **D** также образуют категорию (категорию функторов), морфизмами в которой являются так называемые **естественные преобразования**. Естественное преобразование предоставляет способ перевести один функтор в другой, сохраняя внутреннюю структуру (например, композиции морфизмов).

Пусть F и G — функторы из категории **C** в категорию **D**. Тогда естественное преобразование η: $F \Rightarrow G$ сопоставляет каждому объекту X в категории **C** морфизм η_X: $F(X) \to G(X)$ в категории **D**, называемый компонентой η в X, так, что для любого морфизма $f: X \to Y$ в категории **C** диаграмма (в категории **D**), изображенная на рисунке ниже, коммутативна.

$$\begin{array}{ccc} X & F(X) & \xrightarrow{\eta_X} G(X) \\ {\scriptstyle f}\downarrow & {\scriptstyle F(f)}\downarrow & \downarrow{\scriptstyle G(f)} \\ Y & F(Y) & \xrightarrow[\eta_Y]{} G(Y) \end{array}$$

Коммутативность данной диаграммы означает, что из $F(X)$ в $G(Y)$ можно прийти двумя разными путями, или что выполнено следующее равенство:

$$\eta_Y \circ F(f) = G(f) \circ \eta_X.$$

Можно определить внешнюю (external) бинарную операцию между функторами и естественными преобразованиями (в английской терминологии **whiskering**). Пусть **B**, **C**, **D**, **E** — категории, K: $\mathbf{B} \to \mathbf{C}$, F, G: $\mathbf{C} \to \mathbf{D}$, H: $\mathbf{D} \to \mathbf{E}$ — функторы, η: $F \Rightarrow G$ — естественное преобразование. Тогда мы можем определить естественное преобразование $H\eta$: $H \circ F \Rightarrow H \circ G$, задав

$$(H\eta)_X = H(\eta_X), X \in \mathbf{C}.$$

Здесь задается равенство морфизмов в категории **E**. Аналогично мы можем определить естественное преобразование ηK: $F \circ K \Rightarrow G \circ K$, задав

$$(\eta K)_X = \eta_{K(X)}, X \in \mathbf{B}.$$

Здесь задается равенство морфизмов в категории **D**.

Функторы из категории в нее саму называются **эндофункторами**. Эндофункторы играют важную роль в теории категорий. Между любыми двумя эндофункторами в некоторой категории **C** определена композиция (так как начало и конец совпадают), причем эта композиция ассоциативна, а также существует единичный эндофунктор (обозначим его как $1_{\mathbf{C}}$), оставляющий все объекты и морфизмы в категории **C** без изменения. Эндофункторы в некоторой категории **C** образуют свою категорию с естественными преобразованиями в качестве морфизмов.

Определим теперь теоретико-категорное понятие монады. **Монадой** в теории категорий называется тройка (T, η, μ), где

- T — эндофунктор в некоторой категории **K** (T: $\mathbf{K} \to \mathbf{K}$);
- η: $1_{\mathbf{K}} \to T$ – естественное преобразование;
- μ: $T^2 \to T$ — естественное преобразование;
- следующая диаграмма коммутативна (ассоциативность):

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ {\scriptstyle \mu T}\downarrow & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow[\mu]{} & T \end{array}$$

- следующая диаграмма коммутативна (двухсторонняя единица):

$$\begin{array}{ccc} T & \xrightarrow{\eta T} & T^2 \\ {\scriptstyle T\eta}\downarrow & & \mu\downarrow \\ T^2 & \xrightarrow[\mu]{} & T \end{array}$$

Из определения моноида (см. выше) видно, что монада является моноидом в категории эндофункторов **End**(**K**).

2.3. Функторы и монады в программировании

Функторы. Пусть у нас есть некоторый контейнер, хранящий какое-то количество значений, например, список или объект класса Maybe (хранящий одно значение указанного типа или не хранящий

ничего, в стандартной библиотеке C++ этому классу соответствует класс std::optional). Теперь поставим задачу: применить обычную одноместную функцию (например, sin) к значениям в контейнере. Как это сделать в языке Haskell со списками известно — применить функцию map. Но как это сделать с типом Maybe, и в общем случае — как это сделать данными в произвольном контейнере? Универсальный подход состоит в том, чтобы доверить это ответственное дело самому контейнеру. Для этого в языке Haskell определен 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)
```

Из последней записи следует, что функцию fmap можно рассматривать как функцию с одним параметрам (“обычной” функцией, то есть функцией, принимающей и возвращающей простые значения), возвращающую функцию, принимающую и возвращающую значения в контейнере.

Любой тип данных можно объявить функтором, реализовав для него экземпляр класса Functor и функцию fmap. Любая реализация функтора должна удовлетворять двум функторным законам:

```
1. fmap id = id
2. fmap (g . f) = fmap g . fmap f
```

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 функториой операции относительно композиции функций.

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
```

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, помещенную в аппликатив, и второго параметра — значение, помещенное в тот же аппликатив, и возвращающий результат в том же аппликативе.

Любая реализация аппликатива должна удовлетворять аппликативным законам:

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

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, причем если аргумент функции принадлежит области ее определения, то возвращается значение Just value, а иначе — Nothing (признак ошибки). 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.

Определение класса Monad в языке Haskell выглядит так:

```
class Applicative m => Monad m where
  return :: a -> m a
  return = pure
  (>>=) :: m a -> (a -> m b) -> m b
```

Каждая монада имеет две основные функции: **return** и bind (в языке Haskell оператор (>>=). Функция **return** аналогична функции pure для аппликативов (фактически для большинства монад **return** определяется как pure). Операция bind 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 (а также функция **return**) — примеры монадных функций.

Функции **return** и bind должны удовлетворять трем монадным законам. Для того чтобы их сформулировать, введем операцию монадной композиции (оператор (>=>) в языке Haskell). Она определяется следующим образом:

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

Оба операнда монадной композиции и ее 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. return >=> f == f
2. f >=> return == f
3. (f >=> g) >=> h == f >=> (g >=> h)
```

Другими словами, функции **return** и bind должны быть определены так, чтобы, во-первых, функция **return** являлась единичным элементом (левым и правым) монадной композиции (первые два закона) и, во-вторых, монадная композиция должна быть ассоциативной (третий закон).

Так как любая монада также является функтором, то для любой монады операцию bind можно определить через операцию fmap следующим образом:

```
x >>= f = join (f <$> x)
```

Функция fmap в данном случае вернет значение “монады в монаде” (например, список списков). Функция join убирает этот один лишний уровень вложенности. Функции fmap и join можно определить через монадные операции:

```
fmap f m = m >>= (return . f)
join n = n >>= id
```

Таким образом, по-умолчанию, bind, join и fmap циклически определяются друг через друга. Чтобы разорвать этот замкнутый круг, нужно какие-то из этих операций определить явно. Как правило, явно определяются fmap и bind.

Если сравнить математическое и “программистское” определения монады, то можно заметить, что естественному преобразованию соответствует функция **return**, а естественному преобразованию μ — функция join.

Возвращаясь к нашим “безопасным” функциям, заметим, что 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 вычислений, остальные функции фактически не вычисляются. Это чем-то напоминает исключения (exceptions) в императивных языках (таких, как C++).

3. БИБЛИОТЕКА ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ

3.1. Общее описание библиотеки

При реализации библиотеки функционального программирования funcprog для языка C++ 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 (главным образом потому, что в ее реализации используются виртуальные функции, которые на GPU непереносимы). Это не может быть и обычная функция. Адрес обычной функции передать в CUDA, конечно же, можно, но это не имеет смысла, так как у CPU и у CUDA несовместимые наборы инструкций. Было принято решение в рамках библиотеки funcprog2 создать класс function2 и считать функцией любой объект этого класса. В объект класса function2 можно преобразовать любой функциональный объект (имеющий функциональный оператор ()), в частности, это может быть лямбда-выражение. Напомним, что в современном языке C++ (начиная с версии C++11) лямбда-выражение

начинается с пары квадратных скобок, внутрь которых могут быть помещены переменные, доступные из лямбда-выражения. Для того чтобы объект можно было передавать в CUDA, функциональный оператор должен быть помечен ключевым словом __device__. Чтобы пользовательский исходный код был платформо-независим, в библиотеке funcprog2 имеется макрос __DEVICE, который при компиляции для CUDA расширяется в ключевое слово __device__, а при компиляции для CPU — в пустую строку. Таким образом, функциональный оператор нужно пометить словом __DEVICE. Для преобразования функционального объекта в объект класса function2 в библиотеке имеется специальная функция _ (подчерк). Недостатком класса function2 по сравнению с классом std::function является то, что в метаданные функции попадает не только ее прототип (типы параметров и возвращаемое значение), но и реализация (класс объекта) — такова плата за возможность переноса на CUDA. Вот как реализован класс function2:

```
template<typename FuncType,
  typename FuncImpl> struct function2;
  template<typename Ret, typename... Args,
    typename FuncImpl>
  struct function2<Ret(Args...) ,FuncImpl>{
    function2(FuncImpl const& impl)  :
      impl(impl){}
    Ret 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>; };

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

Приведем 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 funcprog2 достаточно полно реализовано каррирование функций. Для этого в библиотеке реализован оператор применения аргумента к функции с помощью оператора сдвига влево <<. При этом создается 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 вернется результат ее 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 вернется 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 чем-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 funcprog2 определен так:

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

Шаблон класса типом не является, и его нельзя передать как параметр шаблона другого класса. Поэтому определяется еще 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 — return_ и bind. При реализации статических методов этих классов нужно не забывать про выполнение функторных, аппликативных и монадных законов.

Заметим также, что в библиотеке funcprog2 в качестве функторного оператора используется оператор деления, а в качестве аппликативного — умножение.

4. ПРИМЕНЕНИЕ БИБЛИОТЕКИ ДЛЯ ЧИСЛЕННЫХ МЕТОДОВ

Технологически любая задача численного моделирования начинается с построения сетки в области расчета. Причем в областях сложной формы эта сетка, как правило, неструктурная. Интересующие исследователя сеточные функции могут быть заданы как в узлах ячеек, так и в их центрах.

На данном этапе исследования мы рассматриваем только нестационарные задачи и считаем, что для интегрирования по времени используются различные явные схемы, например, в программном комплексе, который мы использовали для перевода на GPU, это явная классическая схема Рунге–Кутты четвертого порядка. Неявные схемы, предполагающие решение линейных систем уравнений, в настоящий момент не рассматривались.

Если метод явный, то вычислять значения сеточных функций в разных точках сетки можно независимо друг от друга, и, следовательно, эти вычисления можно вести параллельно. Таким образом, каждый явный шаг (цикл по индексу сеточной функции) можно распараллеливать на общей памяти. Заметим, что MPI-параллелизация также возможна, но пока не рассматриваются. При расчетах на CPU распараллеливание циклов делается, как правило, с помощью OpenMP. При расчетах 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 funcprog2 так, как это описывается далее.

4.1. Сеточные выражения и сеточные функции

Введем понятие сеточного выражения. Это объект, определенный для всех индексов сетки, то есть для любого объекта, являющегося сеточным выражением, можно узнать, чему равно его значение для любого индекса сетки. Простейшим 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 используются шаблоны выражений [13] и шаблон проектирования CRTP (Curiously Recurring Template Pattern) [14], при котором в базовый класс в качестве параметра шаблона передается конечный класс. Про этот шаблон и другие методы метапрограммирования можно прочитать в книгах [15], [16], [18].

Любое сеточное выражение можно присвоить сеточной функции. Этот оператор присваивания проходит по всем индексам сеточной функции, которой присваивается сеточное выражение, для каж-

дого 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 реализован этот оператор присваивания.

Вначале введем понятие “исполнителя” (executor). Исполнитель — это объект, параллельно выполняющий цикл с указанным телом цикла. Тело цикла задается объектом-замыканием (closure), который все нужное для исполнения, кроме индекса цикла, хранит в своих переменных-членах, а индекс цикла передается ему в функциональном операторе. При этом этот функциональный оператор с разными значениями индекса цикла может быть вызван параллельно. Вот как реализован исполнитель для обычного процессора (CPU):

```
struct cpu_executor {
  template<class Closure>
  void operator()(size_t size,
    Closure const& closure) const {
#pragma omp parallel for
  for(size_t i = 0; i < size; ++i)
    closure(i);
  }
};
using default_executor = cpu_executor;
```

А вот как исполнитель реализован для CUDA (мы используем тот факт, что ядро (kernel) может быть полиморфной функцией):

```
#ifndef BLOCK1_SIZE
#define BLOCK1_SIZE 512
#endif

template<class Closure>
__global__ void cuda_exec(
  size_t size, Closure closure
){
  size_t const i = blockIdx.x *
    blockDim.x + threadIdx.x;
  if(i < size) closure(i);
}

struct cuda_executor {
  template<class Closure>
  void operator()(size_t size,
    Closure const& closure) const
{
  unsigned const
    dimGrid = (size +
      BLOCK1_SIZE - 1) / BLOCK1_SIZE,
    dimBlock = size < BLOCK1_SIZE ?
      size : BLOCK1_SIZE;
    cuda_exec<<<dimGrid, dimBlock>>>
      (size, closure)
  }
};
using default_executor = cuda_executor;
```

Теперь мы можем определить оператор присваивания сеточного выражения сеточной функции:

```
template<typename T>
template<class GE>
void vector_grid_function<T>::operator=
  (grid_expression<GE,
    typename GE::proxy_type> const& gexp)
{
  proxy_type gf_p = get_proxy();
  auto const gexp_p = gexp.get_proxy();
  default_executor()(m_local_size ,
    [gf_p, gexp_p] __DEVICE (size_t i) {
      const_cast<proxy_type&>
        (gf_p)[i] = gexp_p[i];
});
}
```

Говоря про сеточные функции, нужно упомянуть еще 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 определен 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 все-таки нужно передать из CPU в CUDA, то для него можно создать объект-заместитель, удовлетворяющий перечисленным требованиям и хранящий 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, векторы 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 (например, списки) являются функторами, аппликативами и монадами. Это дает возможность 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 передается старое значение в ячейке. В новом подходе мы хотим, чтобы в этом случае можно было написать что-то типа:

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

Если же для вычислений требуются еще несколько сеточных функций (назовем 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 еще дополнительно некоторое постоянное значение (не зависящее от индекса цикла), то вместо

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

можно было бы написать

```
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.

Покажем, как реализованы функторы, аппликативы и монады для сеточных выражений и докажем, что эти реализации корректны (удовлетворяют требуемым законам). Доказывать теоремы будем методом эквационального рассуждения (equational reasoning). Этот метод доказательства основан на приведении левой и правой частей доказываемого равенства путем эквивалентных преобразований к одинаковому выражению. Пусть теперь f — применяемая функция, а gexp — сеточное выражение.

Функтор. Функция *fmap* принимает функцию с одним параметром и функтор (в нашем случае сеточное выражение) и возвращает тот же функтор (новое сеточное выражение). Это новое сеточное выражение запоминает параметры функции *fmap* в своих переменных-членах класса (назовем их f и gexp) и реализует оператор [] следующим образом:

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

Теорема 1 (Теорема о функторе) *Определенная 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] = -- def of fmap
  id gexp[i] = -- def 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] = -- def of fmap
  (g . f) gexp[i] = -- def of
    function composition
  g (f gexp[i])
```

С другой стороны:

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

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) в нашем случае принимает два сеточных выражения: первый (назовем его gexp_f) возвращает функции, а второй (назовем 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 (Теорема об аппликативе) *Определенные выше функции pure и apply удовлетворяют аппликативным законам.*

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

```
1. apply (pure id) gexp = gexp
2. apply (pure f) (pure x) = pure (f x)
3. apply u (pure y) = apply(pure($ y)) u
4. apply (apply (apply (pure (.) ) u) v) x =
   apply u (apply v x)
```

Первый закон (Identity):

```
(apply (pure id) gexp ) | i | =          -- apply
  (pure id) | i | gexp | i | id gexp|i| = -- pure
  id gexp|i| =                           -- id
  gexp [i]
```

т.е. apply (pure id) gexp = gexp. Первый закон доказан.

Второй закон (Homomorphism):

```
(apply (pure f) (pure x))[i] = -- apply
  (pure f)[i] (pure x)[i] =    -- pure
  f x
```

С другой стороны:

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

Второй закон доказан. Третий закон (Interchange):

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

С другой стороны:

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

Третий закон доказан. Четвертый закон (Composition):

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

С другой стороны:

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

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

Монада. Монадная функция *return* определена так же, как и аппликативная функция *pure*:

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

Монадная операция *bind* (в языке Haskell и в библиотеке funcprog оператор >>=) принимает монаду (в нашем случае сеточное выражение, обозначим его переменной gexp) и функцию, принимающую обычное (не монадное) значение и возвращающую монаду (сеточное выражение). Определим операцию *bind* следующим образом:

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

Теорема 3 (Теорема о монаде) *Определенные выше функции return и bind удовлетворяют монадным законам.*

Доказательство. Перепишем монадные законы в терминах функций return и bind:

```
1. bind (return x) f = f x
2. bind gexp return = gexp
3. bind (bind gexp f) g =
   bind gexp (\x->bind (f x) g)
```

Первый закон:

```
(bind (return x) f)[i] =          -- bind
  (f (return x)[i])[i] =          -- return
  (f x)[i]
```

Первый закон доказан. Второй закон:

```
(bind gexp return)[i] =           -- bind
  (return gexp[i])[i] =           -- return
  gexp[i]
```

Второй закон доказан. Третий закон:

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

С другой стороны:

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

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

Итак, сеточные выражения являются функторами, аппликативами и монадами. Это значит, что любую обычную унарную функцию можно “применить” к сеточному выражению (с помощью функции *fmap*). Такое применение вернет новое сеточное выражение. Любую бинарную функцию можно “применить” к двум сеточным выражениям (с помощью функции *apply*), а любую функцию с n аргументами можно “применить” к n сеточным выражениям. Это можно сделать одной строчкой. Например, путь есть две сеточных функции f и g. Тогда можно написать так:

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

Так как сеточные выражения являются монадами, то из них можно строить цепочки монадных вычислений вида:

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

Здесь f1, f2, f3 — некоторые функции, принимающие обычные (не монадные) значения и возвращающие сеточные выражения.

5. ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ

5.1. Простейший пример

Рассмотрим функцию axpy из библиотеки BLAS. Эта функция принимает константу a и два вектора (x и y) и модифицирует вектор y по формуле $y[i] += a * x[i]$. Ее реализацию для обычного процессора можно записать так:

```
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];
}
```

Эта функция прекрасно распараллеливается, но способ распараллеливания в данной реализации указан явно и для графических ускорителей не подходит. С использованием функционального программирования эту функцию можно было бы переписать следующим образом:

```
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;
}
```

Лямбда-выражение в теле этой функции принимает в качестве параметров нашу константу a, i-й элемент сеточной функции x, по ссылке i-й элемент сеточной функции y и текущий индекс цикла i (который мы игнорируем). Каждому входному параметру (а их у нас два) соответствует один параметр лямбда-выражения, а сеточной функции, которой делается присваивание выражения, соответствует выходной параметр (передаваемый по ссылке) и индекс цикла. Лямбда-выражение передается функции _ (подчерк), которая преобразует это лямбда-выражение в объект класса function2 (функцию в понятиях библиотеки). Этому объекту передаются входные параметры, первый — как для функтора, а следующие — как для аппликатива. Вот пример обращения к функции axpy:

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

5.2. Более сложный пример

Теперь мы будем вычислять выражение вида z[i]=a*x[i]+b*y[ind[i]] (назовем функцию axpby). Ее особенностью является то, что индекс сеточной функции y содержится в дополнительной сеточной функции 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;
}
```

Из существенно нового по сравнению с первым примером здесь то, что сеточную функцию y мы уже не можем передавать через apply. Вместо этого нам нужно для нее создать объект-заместитель (proxy) и передать его через pure. Именно это и делает библиотечная функция pr:

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

Обратим еще внимание на то, что из GPU недоступны глобальные переменные, которые хранятся в памяти CPU (современные CUDA-платформы могут это позволять за счет механизма CUDA Unified Memory, но в любом случае это менее эффективно и, главное, доступно далеко не всегда), поэтому их приходится передавать явно с помощью функции pure. Если нужен буфер для промежуточных вычислений, то его размер должен быть равен числу узлов сетки, так как в случае CUDA мы не знаем абсолютный номер нити исполнения (thread).

Еще одна важная часто возникающая задача — это редукция (например, поиск максимального значения или сумма всех значений). Редукцию также очень важно выполнять параллельно. В исходной версии программы редукция выполнялась средствами OpenMP, но теперь мы этот механизм использовать не можем. К счастью, в современном языке C++ (начиная со стандарта языка C++17) у функций редукции (таких как accumulate и reduce) есть параллельная версия. При работе на CUDA мы используем библиотеку thrust, входящую в состав CUDA Computing Toolkit. В этой библиотеке также имеются параллельные функции редукции, работающие на GPU. Таким образом, при работе на CUDA мы используем параллельные функции редукции из библиотеки thrust, при работе на CPU, если имеются параллельные версии функций редукции (если компилятор их поддерживает), то используются они, иначе редукуция делается последовательно.

6. СРАВНЕНИЕ ЭФФЕКТИВНОСТИ

Мы решали некоторую задачу моделирования течения набегающего потока воздуха в трубе с препятствием с помощью гибридного суперкомпьютера K60, установленного в Центре коллективного пользования ИПМ им. М.В. Келдыша РАН [19], на одном узле в трех режимах: на обычном процессоре (2 x Intel Xeon Gold 6142 v4, 16 ядер) с использованием OpenMP и на GPU (nVidia Volta GV100GL) с использованием OpenCL и с использованием описанного подхода. Ускорение по сравнению с CPU при использовании OpenCL составило около 20 раз, а при использовании нашего подхода — около 12 раз. Ускорение получилось не таким большим, как при использовании OpenCL, но с учетом указанных преимуществ (прежде всего единый исходный код) его можно считать приемлемым.

7. 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 нем 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.

СПИСОК ЛИТЕРАТУРЫ

1. TOP 500. URL: https://www.top500.org
2. NVIDIA. URL: https://www.nvidia.com
3. TOP 50. URL: http://top50.supercomputers.ru
4. OpenCL. URL: https://www.khronos.org/opencl/
5. OpenACC. URL: https://www.openacc.org
6. CUDA Zone. URL: https://developer.nvidia.com/cuda-zone
7. *Краснов М.М.* Библиотека функционального программирования для языка C++ // Программирование. 2020. № 5. С. 47-59. DOI: 10.31857/S0132347420050040
8. *Краснов М.М.* Операторная библиотека для решения трехмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA // Математическое моделирование. 2015. Т. 27. № 3. С. 109-120. URL: http://www.mathnet.ru/links/38633e7a627ab2ce1527ae4a092be72f/mm3585.pdf
9. *Краснов М.М.* Канд. дисс. “Сеточно-операторный подход к программированию задач математической физики”. Автореф. URL: http://keldysh.ru/council/1/2017-krasnov/avtoref.pdf
10. Haskell language. URL: https://www.haskell.org/
11. *Маклейн С.* Категории для работающего математика / перевод с англ. под ред. В.А. Артамонова. М.: ФИЗМАТЛИТ, 2004. 352 с. ISBN 5-9221-0400-4.
12. *Bartosz Milewski.* Category Theory for Programmers. URL: https://github.com/hmemcpy/milewski-ctfp-pdf/releases/download/v1.3.0/category-theory-for-programmers.pdf
13. *Veldhuizen T.* Expression Templates. C++ Report. Vol. 7. № 5. June 1995. pp. 26-31.
14. *Coplien J.O.* Curiously recurring template patterns. C++ Report, February 1995, pp. 24–27.
15. *David Abrahams, Aleksey Gurtovoy.* C++ Template Metaprogramming. Addison-Wesley. 2004. 400 с. ISBN 978-0-321-22725-6.
16. *Краснов М.М.* Метапрограммирование шаблонов C++ в задачах математической физики. М.: ИПМ им. М.В. Келдыша, 2017. 84 с. http://dx.doi.org/10.20948/mono-2017-krasnov10.20948/mono-2017-krasnov.
17. *Краснов М.М.* Применение символьного дифференцирования для решения ряда вычислительных задач // Препринты ИПМ им. М.В. Келдыша. 2017. № 4. 24 с. http://dx.doi.org/10.20948/prepr-2017-410.20948/prepr-2017-4.
18. *Краснов М.М.* Применение функционального программирования при решении численных задач // Препринты ИПМ им. М.В. Келдыша. 2019. № 114. 36 с. http://dx.doi.org/10.20948/prepr-2019-11410.20948/prepr-2019-114.
19. Вычислительный комплекс К-60. https://www.kiam.ru/MVS/resourses/k60.html.

THE USE OF FUNCTIONAL PROGRAMMING LIBRARY TO PARALLELIZE ON GRAPHICS ACCELERATORS WITH CUDA TECHNOLOGY

M. M. Krasnov[a], O. B. Feodoritova[a]

[a] *Keldysh Institute of Applied Mathematics of Russian Academy of Sciences, Miusskaya sq., 4 Moscow, 125047, Russia*

Modern graphics accelerators (GPUs) can signi cantly 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 technology developed by NVIDIA, allow you to have a single source code for both conventional processors (CPUs) and CUDA. However, in this single source code, you need to somehow tell the compiler which parts of this code to parallelize on shared memory. 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

REFERENCES

1. TOP 500. URL: https://www.top500.org
2. NVIDIA. URL: https://www.nvidia.com
3. TOP 50. URL: http://top50.supercomputers.ru
4. OpenCL. URL: https://www.khronos.org/opencl/
5. OpenACC. URL: https://www.openacc.org
6. CUDA Zone. URL: https://developer.nvidia.com/cuda-zone
7. *Krasnov M.M.* Functional Programming Library for C++ // Programming and Computer Software, 2020, v. 46, no. 5, pp. 330–340. http://dx.doi.org/10.1134/S0361768820050047
8. *Krasnov M. M.* Operator library for solving three-dimensional grid problems of mathematical physics using graphics cards with CUDA architecture // Mathematical Modeling, 2015, v. 27, no. 3, pp. 109-120. URL: http://www.mathnet.ru/links/38633e7a627ab2ce1527ae4a092be72f/mm3585.pdf
9. *Krasnov M. M.* Candidate's thesis "Grid-operator approach to programming problems of mathematical physics". Abstract. URL: http://keldysh.ru/council/1/2017-krasnov/avtoref.pdf
10. Haskell language. URL: https://www.haskell.org/
11. *McLane S.* Categories for the working mathematician / Translation from English edited by V.A. Artamonova. M.: FIZMATLIT, 2004. 352 p. ISBN 5-9221-0400-4.
12. *Milewski B.* Category Theory for Programmers. URL: https://github.com/hmemcpy/milewski-ctfp-pdf/releases/download/v1.3.0/category-theory-for-programmers.pdf
13. *Veldhuizen T.* Expression Templates. C++ Report, Vol. 7. 5, June 1995, pp. 26-31.
14. *Coplien J.O.* Curiously recurring template patterns. C++ Report, February 1995, pp. 24-27.
15. *Abrahams D., Aleksey Gurtovoy.* C++ Template Metaprogramming. Addison-Wesley. — 2004. 400 c. ISBN 978-0-321-22725-6.
16. *Krasnov M. M.* Metaprogramming of C++ templates in problems of mathematical physics. M.: IPM im. M.V. Keldysh, 2017. 84 p. http://dx.doi.org/10.20948/mono-2017-krasnov10.20948/mono-2017-krasnov.
17. *Krasnov M. M.* Application of symbolic differentiation to solve a number of computational problems // Preprints of IAM im. M.V. Keldysh. 2017. No. 4. 24 p. http://dx.doi.org/10.20948/prepr-2017-410.20948/prepr-2017-4.
18. *Krasnov M. M.* Application of functional programming in solving numerical problems // Preprints of IPM im. M.V. Keldysh. 2019. No. 114. 36 p. http://dx.doi.org/10.20948/prepr-2019-11410.20948/prepr-2019-114.
19. Computer complex K-60. URL: https://www.kiam.ru/MVS/resourses/k60.html.