ИСПОЛЬЗОВАНИЕ МЕТОДОВ И БИБЛИОТЕКИ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ ДЛЯ РЕШЕНИЯ ЧИСЛЕННЫХ ЗАДАЧ НА ГРАФИЧЕСКИХ УСКОРИТЕЛЯХ С ТЕХНОЛОГИЕЙ CUDA

М.М. Краснов, О.Б. Феодоритова

Институт прикладной математики им. М.В. Келдыша РАН, Москва, ktm@kiam.ru, feodor@kiam.ru

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 [1] 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. В частности, при реализации этого подхода активно использовалось метапрограммирование шаблонов языка C++ (см [2], [3]), в том числе шаблоны выражений [4], основан-

ные на идиоме языка C++ CRTP [5]. Более подробно про новые возможности языка C++ можно прочитать в книге автора языка [6]. Предлагаемый авторами подход применен для переноса на графические ускорители программного кода MCFL для численного моделирования течения вязкого теплопроводного многокомпонентного газа, взаимодействующего с твердым телом [7].

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 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.

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.

Функторы, аппликативы и монады

Функторы. Пусть у нас есть некоторый контейнер, хранящий какое-то количество значений, например, список. Теперь поставим задачу: применить обычную одноместную функцию (например, sin) к значениям в 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
```

Прототип функции fmap можно записать в другом эквивалентном виде (это следует из правоассоциативности стрелки вправо):

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

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
```

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
```

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
```

Монады позволяют строить цепочки вычислений из так называемых «монадных» функций, принимающих обычное значение (например, число), а возвращающих значения в контейнере (например, список значений).

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.

Сеточные выражения как функторы, аппликативы и монады.

Сеточные выражения можно рассматривать как контейнеры (особенно это справедливо для сеточных функций). Сделаем сеточное выражение функтором, аппликативом и монадой, чтобы и к ним можно было применять функции. Пусть 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.

Аппликатив. Аппликатиыные функции pure и apply для сеточных выражений определены следующим образом:

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

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

Монада. Монадные функции mreturn и mbind определены следующим образом:

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

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

Пример программы. Приведём пример программы, вычисляющий функцию axpy из библиотеки BLAS:

```
template<typename T>
void axpy(T a, math_vector<T> const& x, math_vector<T> &y){
    mv(y) = _([] (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
}
```

Литература

1. Краснов М.М. Библиотека функционального программирования для языка C++. Программирование, 2020 г., № 5, с. 47-59, DOI: 10.31875/S0132347420050040
2. David Abrahams, Aleksey Gurtovoy. C++ Template Metaprogramming. Addison-Wesley, 2004, 400 с. ISBN 978-0-321-22725-6.
3. 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
4. T. Veldhuizen, Expression Templates. C++ Report, Vol. 7 № 5, June 1995, pp. 26-31.
5. J.O. Coplien. Curiously recurring template patterns. C++ Report, February 1995, pp. 24–27.
6. Bjarne Stroustrup. The C++ Programming Language, Fourth Edition. Addison-Wesley, 2013. ISBN 978-0-321-56384-2, 1368 с.
7. Feodoritova O.B., Krasnov M.M., Zhukov V.T. A Numerical Method for Conju-gate Heat Transfer Problems in Multicomponent Flows. // J.Phys.Conf.Ser., 2021. Vol.2028 012024, DOI:10.1088/1742-6596/2028/012024