ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 9 за 2013 г.

Краснов М.М., Феодоритова О.Б.

Операторная библиотека для решения трёхмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA

Рекомендуемая форма библиографической ссылки: Краснов М.М., Феодоритова О.Б. Операторная библиотека для решения трёхмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA // Препринты ИПМ им. М.В.Келдыша. 2013. № 9. 32 с. URL: http://library.keldysh.ru/preprint.asp?id=2013-9

Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В.Келдыша
Российской академии наук

М.М. Краснов, О.Б. Феодоритова

Операторная библиотека для решения вычислительных задач с использованием графических плат с архитектурой CUDA

Москва — 2013

УДК 519.6

М.М. Краснов, О.Б. Феодоритова

Операторная библиотека для решения вычислительных задач с использованием графических плат с архитектурой CUDA.

В данной работе описывается библиотека операторов для работы с сеточными функциями, предназначенная для упрощенной записи вычислений на трехмерных индексных сетках. Библиотека спроектирована таким образом, чтобы детали ее реализации были скрыты от пользователя, что позволяет эффективно реализовать ее на машинах с различной архитектурой (параллельных, гибридных и т. п.). При создании библиотеки ставилось несколько целей: приближение внешнего вида программ к формулам в теоретических работах, относительная простота использования, в том числе при работе на графических платах с архитектурой CUDA, вычислительная эффективность.

Ключевые слова: сеточные функции, сеточные операторы, сеточные вычислители, гетерогенные системы, CUDA

M.M. Krasnov, O.B. Feodoritova

Operator library for solving of numerical problems using GPUs with CUDA architecture

This paper describes a library of operators to work with grid functions, aimed to simplify the calculations on three-dimensional index grid. The library is designed so that the details of its implementation are hidden from the user, allowing effective implementation on machines with different architectures (parallel, hybrid, etc.). The library creation raises several goals: the approximation of the appearance of the program to the theoretical formulas, the relative ease of use, including the work on the graphics card with CUDA architecture, computational efficiency.

Key words: grid functions, grid operators, grid evaluators, heterogeneous systems, CUDA

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проекты 11-01-12082-офи-м, 11-01-12045-офи-м

1. Введение

В задачах математического моделирования широко используются сеточные функции – величины, определённые в каждом узле некоторой трёхмерной прямоугольной сетки (см. [1]). Для численного решения задач математического моделирования с этими сеточными функциями делаются определённые преобразования. В теоретической литературе эти преобразования описываются операторами, например, оператором Лапласа. Кроме того, в уравнениях могут встречаться линейные комбинации и композиции операторов. Например, при решении эллиптического уравнения многосеточным методом итерирующий оператор для двух уровней имеет вид (см. [1]):

$$Q = S_p (I - P\ A_H^{-1}\ R\ A_h)\ S_p$$

где A_H и A_h – операторы (матрицы) на подробной и грубой сетках соответственно, P – оператор интерполяции (продолжения), R – оператор сборки (проектирования), S_p – сглаживающий оператор, p – число пре– и пост– сглаживающих шагов, I – единичный оператор.

В теоретических работах описание алгоритмов выглядит очень компактно и элегантно. При реальном программировании текст программы выглядит гораздо более громоздко и требует дополнительной памяти для сохранения промежуточных результатов вычислений. Целью данной работы было приблизить, насколько это возможно, внешний вид программы к формулам в теоретических работах. Кроме того, важнейшей задачей ставилась эффективность вычислений. Ещё одной немаловажной проблемой является то, что в настоящее время большое распространение получили гибридные суперкомпьютеры с вычислительными графическими платами, а их эффективное использование затруднено, т.к. требует освоения большого объёма новой и непривычной информации о методах программирования на них.

В данной библиотеке «оператор» из абстрактного теоретического понятия превращён в реальный программный объект. Основными объектами, с которыми позволяет работать библиотека, являются: вычисляемый объект (evaluatable object), сеточная функция (grid function), сеточный оператор (grid operator) и сеточный вычислитель (grid evaluator). Для сеточных операторов реализована своя арифметика, позволяющая получать из простых операторов новые составные операторы. Сеточные операторы можно «применять» к вычисляемым объектам, к которым относятся сеточные функции и сеточные вычислители. Результатом такого применения является сеточный вычислитель. Для сеточных вычислителей также реализована своя арифметика, позволяющая получать из простых вычислителей новые составные вычислители. Сеточные вычислители можно присваивать плотным сеточным функциям. Запуск

вычислений производится именно при таком присваивании, не раньше. До момента присваивания вычислителя плотной сеточной функции цепочка вычислений просто запоминается. Таким образом, в библиотеке реализуется концепция «ленивых» вычислений. Для реализации «ленивых» вычислений в библиотеке активно используется механизм шаблонных метавычислений языка C++ (Template metaprogramming – см. [2]).

Хотя в данной работе описывается одна библиотека, фактически их две – *gridmath* и *gridmath_cuda*. Обе библиотеки содержат аналогичные классы, отличающиеся названием (например, `grid_function` в библиотеке *gridmath* и `cuda_grid_function` в библиотеке *gridmath_cuda*) и реализацией. Основной операцией в библиотеке является присваивание вычислителя плотной сеточной функции. В библиотеке *gridmath* эта операция реализована последовательно (три вложенных цикла), а в библиотеке *gridmath_cuda* – параллельно.

Исходный текст библиотеки может показаться на первый взгляд довольно сложным, но авторы надеются, что использование библиотеки будет достаточно простым.

Библиотека *gridmath* была написана и отлажена в ИПМ им. М.В. Келдыша на гибридной супер ЭВМ К–100.

2. Общее описание библиотеки

2.1 Назначение библиотеки

Библиотека *gridmath* предназначена для упрощенной записи вычислений на трехмерных индексных сетках. Основными объектами, с которыми позволяет работать библиотека, являются: вычисляемый объект (evaluatable object), сеточная функция (grid function), сеточный оператор (grid operator) и сеточный вычислитель (grid evaluator). К вычисляемым объектам относятся сеточные функции и сеточные вычислители. К ним можно применять сеточные операторы, при таком применении получается сеточный вычислитель. Для сеточных операторов реализована арифметика с другими сеточными операторами и скалярными величинами, при этом порождаются новые составные сеточные операторы. Для сеточных вычислителей также реализована своя арифметика с другими сеточными вычислителями и скалярными величинами, при этом порождаются новые составные сеточные вычислители. Сеточные вычислители можно присваивать плотным сеточным функциям. В дальнейшем изложении слово «сеточный» в словосочетаниях «сеточный оператор» и «сеточный вычислитель» будет опускаться, чтобы не загромождать текст.

Применение оператора к вычисляемому объекту реализовано с помощью функционального оператора `()`. Приведём примеры. Пусть `f`, `g` – сеточные функции, `h` – плотная сеточная функция, `A`, `B` – сеточные операторы. Тогда мы можем написать такие операторы:

```
h=A(f);
```

В этом примере оператор `A` применяется к сеточной функции `f`, при этом получается вычислитель, который присваивается плотной сеточной функции `h`.

```
h=(A+B)(f);
```

В этом примере складываются два оператора `A` и `B`, при этом получается новый составной оператор. Этот новый оператор применяется к сеточной функции `f`, при этом получается вычислитель, который присваивается плотной сеточной функции `h`.

```
h=A(f+g);
```

В этом примере складываются две сеточные функции `f` и `g`, при этом получается вычислитель (как сумма двух вычисляемых объектов, которыми являются сеточные функции). К этому вычислителю применяется оператор `A`, при этом получается ещё один вычислитель, который присваивается плотной сеточной функции `h`.

```
h=B(A(f)+g);
```

В этом примере вначале оператор `A` применяется к сеточной функции `f`, при этом создаётся вычислитель. Затем этот вычислитель складывается с сеточной функцией `g`, в результате чего создаётся новый составной вычислитель. К этому составному вычислителю применяется сеточный оператор `B`, в результате создаётся ещё один вычислитель, который присваивается плотной сеточной функции `h`. При исполнении данного оператора создаются три вычислителя.

Во всех этих примерах важно обратить внимание на следующее. Каким бы сложным ни было выражение в правой части оператора присваивания, процесс вычисления запускается один раз при присваивании. Это особенно важно при работе на графических ускорителях CUDA, когда вызов ядра является дорогостоящей операцией. Независимо от сложности выражения в правой части при присваивании делается один вызов одного ядра, и все вычисления производятся в этом ядре.

Главное назначение вычислителей – «запомнить» последовательность и параметры вычислений. Таким образом, в библиотеке реализуется концепция «ленивых» вычислений, позволяющая избавиться от промежуточных переменных, за счёт этого сэкономить оперативную память (что становится важным при больших размерах сеток) и существенно ускорить вычисления.

В составе библиотеки есть примеры использования. Примеры расположены в папке teplo библиотеки *gridmath* и teplo_cuda библиотеки *gridmath_cuda*. В примерах решается трёхмерная задача теплопроводности: $\partial U/\partial t = k\Delta U + f$. Подробное описание примера см. ниже.

Концепция данной библиотеки в чём-то перекликается с концепцией языка Норма (см. [3]). Трёхмерные сеточные функции, с которыми работает библиотека, аналогичны величинам на трёхмерных областях языка Норма. Реализация операторов и в библиотеке, и в языке скрыта от пользователя и может выполняться как последовательно, так и параллельно. Существенным отличием является то, что Норма – это самостоятельный язык программирования со своей идеологией, а *gridmath* – библиотека для языка

C++. При использовании библиотеки последовательность вычислений задаётся явно, а в программе на языке Норма задаются только зависимости между величинами, последовательность вычислений компилятор определяет сам. Кроме того, в библиотеке *gridmath* более развита концепция «ленивых» вычислений.

Рассмотрим основные классы и понятия библиотеки более подробно.

2.2 Вычисляемый объект

Вычисляемый объект – это объект, к которому может быть применён сеточный оператор. К вычисляемым объектам относятся сеточные функции и сеточные вычислители.

Для вычисляемых объектов определены операции сложения и вычитания с другими вычисляемыми объектами и все четыре арифметических операции со скалярными величинами, причём скалярная величина может быть как в левой, так и в правой части арифметической операции. Результатом арифметической операции (с другим вычисляемым объектом или со скалярной величиной) является сеточный вычислитель, который, в свою очередь, также является вычисляемым объектом.

Можно сказать, что вычисляемые объекты образуют алгебраическую группу, в которой в качестве алгебраической операции выступает операция сложения с обратной к ней операцией вычитания. Есть и свой «ноль», в качестве которого может выступать, например, скалярная сеточная функция (см. ниже), возвращающая значение ноль для всех узлов сетки.

2.3 Сеточная функция

Под сеточной функцией понимается объект, принимающий определённые значения (одно или несколько) в каждом узле некоторой трёхмерной прямоугольной сетки. Физический шаг сетки не обязан быть равномерным. Каждый узел сетки имеет свои целочисленные координаты по всем трём осям (оси будем обозначать как x, y и z, а целочисленные координаты сетки по этим осям – соответственно как i, j и k). Именно эти целочисленные координаты используются при определении значения сеточной функции в узлах сетки (а не физические координаты узлов). Возможны разные реализации (классы) сеточных функций. В библиотеке на сегодня реализованы три таких класса. Плотная сеточная функция (*dense_grid_function*) принимает, вообще говоря, различные значения в различных узлах сетки и хранит все значения в оперативной памяти. Скалярная сеточная функция (*scalar_grid_function*) принимает во всех узлах сетки одно и то же заданное значение. Вычисляемая сеточная функция (*computable_grid_function*) не хранит свои значения, а каждый раз их вычисляет на основе заданного функционального объекта. Размер, занимаемый в оперативной памяти скалярной и вычисляемой сеточными функциями, пренебрежимо мал. Для оценки размера требуемой оперативной памяти достаточно учесть только плотные сеточные функции.

Следует обратить внимание на то, что в библиотеке *gridmath_cuda* плотная сеточная функция хранит свои данные в памяти графического процессора, поэтому то, какого размера сетку удастся разместить в памяти, определяется не объёмом оперативной памяти основного процессора (каким бы большим он ни был), а объёмом оперативной памяти графического процессора, который может быть существенно меньше. Например, на супер–ЭВМ К–100 объём оперативной памяти обычного процессора – 96 Гбайт на узел, а графического процессора – 2,8 Гбайт на одну графическую плату. Даже с учётом того, что на одном узле стоят три графических платы, объём оперативной памяти оказывается более чем на порядок меньше.

Сеточная функция является вычисляемым объектом. Это означает, что к сеточным функциям можно применять сеточные операторы, и для сеточных функций определены арифметические операции с другими вычисляемыми объектами и со скалярными величинами. Напомним, что другим вычисляемым объектом в библиотеке является сеточный вычислитель.

Для плотной сеточной функции реализован оператор присваивания, принимающий в качестве параметра сеточный вычислитель. Именно этот оператор запускает отложенные вычисления. Порядок вычисления значений в узлах сетки для оператора присваивания не определён и, вообще говоря, может осуществляться параллельно для разных узлов сетки. В связи с этим может возникнуть вопрос: может ли в левой части оператора присваивания стоять плотная сеточная функция, участвующая в вычислениях в правой части? Ответ неоднозначный. Это, как правило, можно делать в том случае, если для вычисления значений в узлах сетки не используются значения из соседних узлов.

Пользователь библиотеки для своих нужд может реализовать собственные сеточные функции. Например, при реализации многосеточного метода для описания правой части автором была написана специальная сеточная функция, принимающая фиксированное значение во всех внутренних узлах сетки (как библиотечная скалярная сеточная функция), а в граничных точках – заданные в каждой точке значения. Такая сеточная функция занимает существенно меньше оперативной памяти, чем плотная сеточная функция (объём занимаемой памяти растёт как квадрат от размера сетки, а не как куб у плотной сеточной функции).

Важный вопрос, который может возникнуть – как сделать первоначальное заполнение плотной сеточной функции на основе имеющейся функции, вычисляющей значение в узле по его координатам? Заполнять её по точкам – не самое лучшее решение, особенно если используется CUDA и данные плотной сеточной функции расположены в памяти графического ускорителя. Пересылка нескольких гигабайт данных (в случае большой сетки) по 8 байт за раз (размер double) может затянуться на часы, т.к. на каждое присваивание будет запускаться ядро. Правильное решение – написать функциональный объект для вычисляемой сеточной функции и присвоить вычисляемую сеточную функцию плотной сеточной функции. При этом все вычисления будут производиться в

одном ядре и займут доли секунды (или считанные секунды) даже для большой сетки, так как будут производиться параллельно для разных узлов сетки. То, как это делается, можно посмотреть в примере teplo_cuda.

2.4 Сеточный оператор

Сеточный оператор (grid operator) – это объект, главное назначение которого – применение к вычисляемому объекту (сеточной функции или сеточному вычислителю). Результатом этого применения является сеточный вычислитель (grid evaluator).

Применение сеточного оператора к вычисляемому объекту реализовано с помощью функционального оператора `()`. Например, если `f` – сеточная функция, `g` – плотная сеточная функция, а `A` – сеточный оператор, то мы можем написать:

```
g=A(f);
```

При этом оператор `A` применяется к сеточной функции `f`, в результате создаётся сеточный вычислитель, который присваивается плотной сеточной функции `g`.

Для сеточных операторов определена своя арифметика. В алгебраических терминах можно сказать, что операторы образуют кольцо. Есть две основные алгебраические операции над операторами. Первая – это групповая операция сложения (с обратной к ней операцией вычитания). Результатом операции сложения (соответственно вычитания) двух операторов является новый оператор, прибавляющий (соответственно, вычитающий) к результату вычисления первого оператора результат вычисления второго оператора. Формально у операции сложения есть свой «ноль» – оператор, возвращающий значение 0 для каждого узла независимо от значения сеточной функции в данном узле.

Вторая алгебраическая операция – это композиция операторов. Она реализована через переопределённую операцию умножения языка C++. Результатом композиции двух операторов является новый оператор, применяющий первый (левый) оператор к результату вычисления второго (правого) оператора. Операция композиции не является групповой, т.к. для неё нет обратной операции. «Единица» для операции композиции есть – это т.н. «единичный» оператор, не меняющий передаваемое ему (из сеточной функции или из другого оператора) значение. Единичный оператор – единственный «конечный» оператор, реализованный в библиотеке. Он называется *value_operator* в библиотеке *gridmath* и, соответственно, *cuda_value_operator* в библиотеке *gridmath_cuda*. Кроме единичного оператора, в библиотеке реализованы только операторы, осуществляющие операции сложения и вычитания операторов и оператор композиции операторов.

Для операторов реализована также арифметика со скалярными величинами. Например, к оператору можно прибавить скалярную величину, результатом будет новый оператор, прибавляющий к результату вычисления оператора эту

скалярную величину. Реализованы все четыре арифметических операции, причём скалярная величина может стоять как в левой части арифметической операции, так и в правой.

Написание «конечных» операторов – главная задача программиста, использующего данную библиотеку. Например, в упоминавшейся выше реализации многосеточного метода был написан оператор, реализующий дискретный оператор Лапласа.

2.5 Сеточный вычислитель

Сеточный вычислитель (grid evaluator) – это объект, который, в отличие от сеточных функций и сеточных операторов, пользователю самому, как правило, писать не придётся. Вычислитель создаётся автоматически при применении сеточного оператора к вычисляемому объекту (сеточной функции или другому вычислителю) или как результат арифметической операции между вычисляемыми объектами (сеточными функциями и вычислителями), и его можно присвоить плотной сеточной функции. Текст программы может содержать код такого вида:

```
g=A(f);
```

Здесь `f` – сеточная функция, `g` – плотная сеточная функция, `A` – сеточный оператор. `A(f)` – оператор применения сеточного оператора `A` к сеточной функции `f`. Этот оператор возвращает вычислитель, который затем присваивается плотной сеточной функции `g`. Из приведённого примера видно, что вычислитель возникает как промежуточный результат вычислений и явно нигде не встречается.

Приведём примеры исходного кода, который может встречаться в программе, использующей данную библиотеку. Пусть `f`, `g` – сеточные функции, `h` – плотная сеточная функция, `A`, `B` – сеточные операторы. Тогда можно написать такой код:

```
    h=A(f)+B(g);    // сложение двух вычислителей.
    h=f+B(g);  // сложение сеточной функции f и вычислителя B(g).
    h=2.0*(A+B)(f); // сложение операторов A и B и умножение
скаляра на вычислитель (A+B)(f).
    h=2.0*(3+A)(f); // сложение скаляра 3 и оператора A и умножение
скаляра 2.0 на вычислитель (3+A)(f).
    h=2.0*(f-B(g)); // вычитание вычислителя B(g) из сеточной
функции f и умножение скаляра 2.0 на вычислитель (f-B(g)).
    h=3.0+(A*B)(f); // композиция операторов A и B и сложение
скаляра 3.0 и вычислителя (A*B)(f).
    h=(3.0+A*B)(f); // композиция операторов A и B и сложение
скаляра 3.0 и оператора A*B.
    h=B(f-A(g));    // вычитание вычислителя A(g) из сеточной
функции f и применение оператора B к полученному вычислителю f-
A(g).
```

2.6 Сеточный диапазон

Не все операторы определены на всей сетке. Например, дискретный оператор Лапласа определён только на внутренних точках сетки, а на границе не определён. Поэтому иногда требуется ограничить область, на которой будет вычислена плотная сеточная функция. Именно для этой цели предназначен объект сеточный диапазон (grid range). Он позволяет задать отступы от правой и левой границ по всем трём направлениям. Вот пример использования сеточного диапазона. В этом примере все отступы равны единице.

```
grid_range_info ginfo;
ginfo.init(1, 1, 1, 1, 1, 1);
ginfo(h)=A(f);
```

В этом примере `f` – сеточная функция, `h` – плотная сеточная функция, `A` – оператор. Значения плотной сеточной функции `h` будут вычислены только во внутренних точках сетки. На границе сетки значения плотной сеточной функции `h` не изменятся.

Сеточный диапазон не привязан к конкретной плотной сеточной функции, поэтому его можно один раз проинициализировать и затем многократно использовать с разными плотными сеточными функциями. Например:

```
ginfo(h1)=A(f);
ginfo(h2)=B(f);
```

В этом примере `f` – сеточная функция, `h1` и `h2` – плотные сеточные функции, `A` и `B` – операторы.

3. Принципы реализации

3.1 Общая информация

Библиотека *gridmath* является шаблонной (template). Большинство её классов и функций шаблонные. Таким образом, вся библиотека поставляется в исходных текстах в виде набора .h и .hpp файлов. Скомпилировать исходные тексты, использующие данную библиотеку, можно любым современным компилятором C++. Автор компилировал библиотеку компиляторами Microsoft Visual C++ 2008 под Windows, GNU C++ и Intel C++ под UNIX, а также компилятором nvcc для CUDA.

Библиотека *gridmath*, помимо стандартной библиотеки языка C++ (STL) использует библиотеку *boost* (см. [4]), при разработке библиотеки это была версия 1.47, а библиотека *gridmath_cuda* – библиотеку *thrust* из состава CUDA SDK (см. [5], [6]). Библиотека *thrust* содержит аналоги большинства необходимых компонентов как из STL, так и из *boost*. Таким образом, для

компиляции библиотеки *gridmath* необходимо, чтобы была установлена библиотека *boost*, а для библиотеки *gridmath_cuda* ничего, кроме CUDA SDK, дополнительно устанавливать не нужно.

Все классы и функции расположены в пространстве имён *kiam::math* для библиотеки *gridmath*, и *kiam::math::cuda* для библиотеки *gridmath_cuda*.

Далее рассмотрим существенные для пользователя особенности реализации основных объектов библиотеки.

3.2 Вычисляемые объекты

Вычисляемые объекты должны удовлетворять следующим условиям:

1. В качестве базового класса должен быть указан следующий библиотечный шаблонный класс:

   для библиотеки *gridmath*:

```
template<class EO, class _Proxy = EO>
struct grid_evaluatable_object;
```

   для библиотеки *gridmath_cuda*:

```
template<class EO, class _Proxy = EO>
struct cuda_grid_evaluatable_object;
```

   В обоих случаях в качестве параметра шаблона `EO` следует указать класс вычисляемого объекта, или, другими словами, класс должен передать базовому классу самого себя. Такое определение базового класса позволяет реализовать т.н. «шаблонный полиморфизм», т.е. имея ссылку на объект базового класса, можно получить ссылку на объект конечного класса и доступ ко всем его методам.

   Параметр `_Proxy` задаёт класс–заместитель, про который будет рассказано ниже.

2. В классе вычисляемого объекта и в классе его заместителя должен быть определён вложенный тип `value_type`, определяющий тип хранимого значения, например, `double` или `float`.
3. В классе заместителя (proxy) вычисляемого объекта должен быть реализован следующий оператор:

   для библиотеки *gridmath*:

```
value_type operator()(size_t i, size_t j, size_t k) const;
```

   для библиотеки *gridmath_cuda*:

```
__device__
value_type operator()(size_t i, size_t j, size_t k) const;
```

3.3 Сеточные функции

Сеточные функции должны удовлетворять следующим условиям:

1. В качестве базового класса должен быть указан следующий библиотечный шаблонный класс:

   для библиотеки *gridmath*:

```
template<class GF, class _Proxy = GF>
```

```
struct grid_function;
```

для библиотеки *gridmath_cuda*:

```
template<class GF, class _Proxy = GF>
struct cuda_grid_function;
```

В обоих случаях в качестве параметра шаблона `GF` следует указать класс сеточной функции, или, другими словами, класс должен передать базовому классу самого себя.

Параметр `_Proxy` задаёт класс-заместитель, про который будет рассказано ниже.

2. В классе сеточной функции и в классе её заместителя должен быть определён вложенный тип `value_type`, определяющий тип хранимого значения, например, `double` или `float`.
3. Может быть определён вложенный тип `const_reference`, определяющий константную ссылку на хранимое значение. Обычно он определяется следующим образом:

```
typedef const value_type& const_reference;
```

4. В классе заместителя (proxy) сеточной функции должен быть реализован следующий оператор:

для библиотеки *gridmath*:

```
value_type operator()(size_t i, size_t j, size_t k) const;
```

для библиотеки *gridmath_cuda*:

```
__device__
value_type operator()(size_t i, size_t j, size_t k) const;
```

Этот оператор может вместо `value_type` возвращать `const_reference`.

В библиотеке *gridmath* класс *grid_function* пронаследован от класса *grid_evaluatable_object*, а в библиотеке *gridmath_cuda* класс *cuda_grid_function* пронаследован от класса *cuda_grid_evaluatable_object*.

3.4 Сеточные операторы

Все сеточные операторы должны удовлетворять следующим условиям:

1. В качестве базового класса должен быть указан следующий библиотечный шаблонный класс:

для библиотеки *gridmath*:

```
template<class GO, class _Proxy = GO>
struct grid_operator;
```

для библиотеки *gridmath_cuda*:

```
template<class GO, class _Proxy = GO>
struct cuda_grid_operator;
```

В обоих случаях в качестве параметра шаблона `GO` следует указать класс сеточного оператора, или, другими словами, класс должен передать базовому классу самого себя.

Параметр `_Proxy` задаёт класс-заместитель, про который будет рассказано ниже.

2. В классе сеточного оператора и в классе его заместителя должен быть определён вложенный тип `value_type`, определяющий тип хранимого значения, например, `double` или `float`.
3. В классе заместителя (proxy) сеточного оператора должен быть реализован следующий оператор:

   для библиотеки *gridmath*:

```
template<class EOP, class OI>
value_type operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy, const OI& oi) const;
```

   для библиотеки *gridmath_cuda*:

```
template<class EOP, class OI>
__device__
value_type operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy, const OI& oi) const;
```

   Здесь `EOP` – класс заместителя вычисляемого объекта, `OI` – класс вызова сеточного оператора (operator invocation).

   `i, j, k` – целочисленные координаты узла вычисляемой сеточной функции.

   `eobj_proxy` – заместитель вычисляемого объекта, к которому применяется оператор.

   Параметр `oi` был введён для реализации композиции операторов. Смысл его следующий. При композиции операторов может оказаться, что данный оператор будет вызван для вычисления значения не от самого вычисляемого объекта, а от значения, вычисленного другим оператором. Поэтому для определения текущего значения вместо вызова

```
eobj_proxy(i, j, k)
```

   нужно писать

```
oi(i, j, k, eobj_proxy)
```

   Например, если пишется оператор, возвращающий синус от текущего значения, то описываемый оператор для библиотеки *gridmath* может выглядеть следующим образом:

```
template<class EP, class OI>
value_type operator()(size_t i, size_t j, size_t k, const EP&
eobj_proxy, const OI& oi) const {
        return std::sin(oi(i, j, k, eobj_proxy));
}
```

Для библиотеки *gridmath_cuda* данный оператор будет выглядеть аналогично с соответствующими поправками на то, что он исполняется в CUDA.

4. Из–за особенностей языка C++ (компилятор скрывает операторы, описанные в базовом классе) требуется переопределение оператора из класса *grid_operator*:

для библиотеки *gridmath:*

```
template<class EO>
operator_evaluator<C, EO>
operator()(const grid_evaluatable_object<EO, typename
EO::proxy_type>& eobj) const {
        return grid_operator<C, typename
C::proxy_type>::operator()(eobj);
    }
```

для библиотеки *gridmath_cuda:*

```
template<class EO>
cuda_operator_evaluator<C, EO>
 operator()(const grid_evaluatable_object<EO, typename
EO::proxy_type>& eobj){
        return cuda_grid_operator<C, typename
C::proxy_type>::operator()(eobj);
    }
```

Здесь C – имя текущего класса. Таким образом, мы заново определяем оператор, имеющийся в базовом классе, путём вызова этого самого оператора из базового класса. Если этого не сделать, то оператор из базового класса не будет виден извне.

3.5 Объекты–заместители

В библиотеке реализована концепция «ленивых» вычислений, т.е. пока не будет исполнен оператор присваивания вычислителя плотной сеточной функции, вычисления не производятся. Вместо этого последовательность вычислений запоминается. При этом запоминаются сеточные функции, операторы, вычислители и операции с ними и со скалярами. Но ссылки на объекты сохранять нельзя, и приходится создавать и сохранять копии объектов. Ссылки на объекты нельзя сохранять по двум причинам. Первая – это то, что объекты (операторы и вычислители) часто создаются «на лету» как результат арифметических операций. Ссылки на такие объекты (созданные «на лету») сохранять нельзя и необходимо делать их копии. Вторая причина актуальна только для библиотеки *gridmath_cuda*. Состоит она в том, что объекты создаются в памяти host–процессора, а исполняются в памяти графического вычислителя CUDA. Чтобы вычисления можно было выполнить, нужно вычисляемый объект скопировать из памяти host–процессора в память графического вычислителя. При этом помимо сеточных операторов и сеточных вычислителей приходится копировать и сеточные функции. Плотная сеточная функция может хранить гигабайты информации и делать копию этих данных

нет смысла, да и памяти может не хватить. Фактически достаточно сохранить указатель на массив данных.

Для того чтобы решить указанную проблему, в библиотеке вводится понятие «заместителя» (proxy) объекта. В качестве замещаемого объекта может быть сеточная функция, оператор или вычислитель. Тот или иной объект может быть «лёгким» или «тяжёлым» для копирования. Лёгкими для копирования будем называть объекты, не содержащие векторы данных, а тяжёлыми – содержащие их. Если объект лёгкий, то при копировании будет сохраняться копия самого объекта, а если тяжёлый – то его лёгкий заместитель. Заместитель объекта похож на сам объект за тем исключением, что в заместителе объекта все вектора данных заменяются на указатели на эти данные.

По умолчанию все объекты (сеточные функции, операторы и вычислители) считаются «лёгкими», т.е. в качестве класса–заместителя по умолчанию передаётся сам класс объекта, а в качестве объекта–заместителя создаётся копия самого объекта. Если сеточный оператор или сеточная функция, написанные Вами, является «тяжёлым» (содержит один или несколько векторов данных), то следует создать класс заместителя и в базовый класс передать в качестве второго шаблонного параметра этот класс заместителя. В качестве примера при реализации класса заместителя можно взять класс *cuda_dense_grid_function*.

Если нужно сохранить сеточную функцию, оператор или вычислитель для последующих отложенных вычислений, то библиотека всегда сохраняет заместитель объекта, а не сам объект путём вызова в сохраняемом объекте метода `get_proxy()`.

Концепция класса–заместителя очень похожа на концепцию замещаемого класса. Основное отличие – класс–заместитель не должен ни от кого наследоваться.

4. Справочник

4.1 Класс grid_object

Класс *grid_object* (*cuda_grid_object* в библиотеке *gridmath_cuda*) является базовым классом для всех основных объектов библиотеки (сеточных функций, операторов и вычислителей). Главное назначение данного класса – обеспечение шаблонного полиморфизма.

Описание класса *grid_object*:

```
template<class T, class _Proxy = T>
struct grid_object
{
 typedef _Proxy proxy_type;

 T& self(){ return static_cast<T&>(*this); }
 const T& self() const { return static_cast<const T&>(*this); }
```

```
    proxy_type get_proxy() const { return self(); }
};
```

Класс *cuda_grid_object* отличается от класса *grid_object* только названием.

В качестве параметра шаблона `T` класс *grid_object* получает имя конечного класса в иерархии классов и с помощью метода `self()` всегда может привести ссылку на себя в ссылку на конечный класс.

Параметр шаблона `_Proxy` задаёт класс заместителя объекта. По умолчанию (если этот параметр не указан), класс заместителя совпадает с классом самого объекта. Если класс заместителя отличается от класса объекта, то в этом классе–заместителе должен быть конструктор, принимающий константную ссылку на основной объект. Этого требует метод `get_proxy()`.

4.2 Класс grid_evaluatable_object

Класс *grid_evaluatable_object* (*cuda_grid_evaluatable_object* в библиотеке *gridmath_cuda*) является базовым классом для всех вычисляемых объектов. К вычисляемым объектам относятся сеточные функции и сеточные вычислители.

Описание класса *grid_evaluatable_object*:

```
template<class EO, class _Proxy = EO>
struct grid_evaluatable_object : grid_object<EO, _Proxy>{};
```

В качестве параметра шаблона `EO` класс *grid_evaluatable_object* принимает имя конечного класса вычисляемого объекта. Параметр шаблона `_Proxy` задаёт класс заместителя вычисляемого объекта. По умолчанию (если этот параметр не указан), класс заместителя совпадает с классом самого вычисляемого объекта.

Класс *grid_evaluatable_object* является маркерным классом и не содержит в себе никакого функционала. Если какой-либо класс пронаследован от данного класса, то он тем самым объявляет себя вычисляемым объектом. К любому вычисляемому объекту может быть применён сеточный оператор. Это накладывает определённые требования на класс вычисляемого объекта. В нём и в его классе заместителя должен быть определён вложенный тип `value_type`, определяющий тип хранимого значения и в классе заместителя должен быть реализован следующий оператор:

для библиотеки *gridmath*:

```
value_type operator()(size_t i, size_t j, size_t k) const;
```

для библиотеки *gridmath_cuda*:

```
__device__
value_type operator()(size_t i, size_t j, size_t k) const;
```

Для вычисляемых объектов реализована арифметика. Любые два вычисляемых объекта можно складывать и вычитать. При этом получается сеточный вычислитель, сам, в свою очередь, являющийся вычисляемым объектом. Кроме того, для вычисляемых объектов реализованы все четыре

арифметических операции со скалярными величинами, причём скалярная величина может быть как в правой, так и в левой частях арифметической операции. Результатом также получается сеточный вычислитель.

Класс *cuda_grid_evaluatable_object* в библиотеке *gridmath_cuda* отличается от класса *grid_evaluatable_object* только названием и тем, что он пронаследован от класса *cuda_grid_object*.

4.3 Класс grid_function

Класс *grid_function* (*cuda_grid_function* в библиотеке *gridmath_cuda*) является базовым классом для всех сеточных функций, как библиотечных, так и определённых пользователем.

Описание класса *grid_function*:

```
template<class GF, class _Proxy = GF>
struct grid_function : grid_evaluatable_object<GF, _Proxy>;
```

В качестве параметра шаблона `GF` класс *grid_function* принимает имя конечного класса сеточной функции. Параметр шаблона `_Proxy` задаёт класс заместителя сеточной функции. По умолчанию (если этот параметр не указан), класс заместителя совпадает с классом самой сеточной функции.

Любая сеточная функция является вычисляемым объектом и к ней может быть применён сеточный оператор. Кроме того, как для любых вычисляемых объектов, для сеточных функций определена арифметика с другими вычисляемыми объектами и со скалярными величинами.

Функционал класса *grid_function* минимальный – он хранит размеры сеточной функции. Хранением данных он не занимается. Это является задачей классов–наследников.

Класс *cuda_grid_function* в библиотеке *gridmath_cuda* отличается от класса *grid_function* только названием и тем, что он пронаследован от класса *cuda_grid_evaluatable_object*.

4.4 Класс scalar_grid_function

Класс *scalar_grid_function* (*cuda_scalar_grid_function* в библиотеке *gridmath_cuda*) – это скалярная сеточная функция, принимающая во всех точках сетки одно и то же заданное числовое значение.

Описание класса *scalar_grid_function*:

```
    template<typename T>
    struct scalar_grid_function :
grid_function<scalar_grid_function<T> >;
```

В качестве параметра шаблона `T` класс *scalar_grid_function* принимает тип хранимых данных. Этот тип должен быть числовым, т.е. для него должны быть определены арифметические операции. Конструктор без параметров создаёт скалярную сеточную функцию с нулевыми размерами и скалярным значением, равным нулю. Второй конструктор позволяет задать размер и значение скаляра. Позже размер можно изменить с помощью метода `resize()` базового класса, а значение скаляра поменять оператором присваивания, принимающим в качестве параметра скаляр. Определены арифметические операции со скалярной величиной.

Для класса *scalar_grid_function* класс заместителя совпадает с самим классом, поэтому в нём реализован функциональный оператор:

```
value_type operator()(size_t i, size_t j, size_t k) const;
```

Класс *cuda_scalar_grid_function* в библиотеке *gridmath_cuda* отличается от класса *scalar_grid_function* только названием и тем, что он пронаследован от класса *cuda_grid_function*. Кроме того, функциональный оператор имеет префикс `__device__`.

4.5 Класс computable_grid_function

Класс *computable_grid_function* (*cuda_computable_grid_function* в библиотеке *gridmath_cuda*) – это вычисляемая сеточная функция, не хранящая значения в узлах сетки, а каждый раз вычисляющая эти значения.

Описание класса *computable_grid_function*:

```
template<class F, class X, class Y, class Z>
struct computable_grid_function_proxy;

template<class F, class X, class Y, class Z>
struct computable_grid_function :
 grid_function<
       computable_grid_function<F, X, Y, Z>,
       computable_grid_function_proxy<F, X, Y, Z>
     >
{
 typedef grid_function<
       computable_grid_function<F, X, Y, Z>,
       computable_grid_function_proxy<F, X, Y, Z>
     > base;
 typedef typename F::value_type value_type;

 computable_grid_function(){}
 computable_grid_function(size_t size_x, size_t size_y, size_t
size_z) : base(size_x, size_y, size_z){}

 F& get_f(){ return f; }
```

```
    X& get_x(){ return x; }
    Y& get_y(){ return y; }
    Z& get_z(){ return z; }

    const F& get_f() const { return f; }
    const X& get_x() const { return x; }
    const Y& get_y() const { return y; }
    const Z& get_z() const { return z; }

    value_type operator()(size_t i, size_t j, size_t k) const {
      return f(x(i), y(j), z(k));
    }

    void get_slice_x(size_t i, std::vector<value_type>& s, size_t
width = 1) const;
    void get_slice_y(size_t j, std::vector<value_type>& s, size_t
width = 1) const;
    void get_slice_z(size_t k, std::vector<value_type>& s, size_t
width = 1) const;

  private:
    F f;
    X x;
    Y y;
    Z z;
  };
```

В этом классе `F` – класс функционального объекта, вычисляющего значение по физическим координатам x, y, z. `X`, `Y`, `Z` – классы функциональных объектов, вычисляющих физические координаты на основе сеточных целочисленных координат соответственно по осям x, y и z.

Классы `F`, `X`, `Y`, `Z` должны иметь классы–заместители (возможно, совпадающие с самими классами). Проще всего пронаследовать эти классы от класса *grid_object* (*cuda_grid_object* в библиотеке *gridmath_cuda*).

Класс заместителя класса `F` должен содержать вложенный тип `value_type`, задающий тип значений, например, `double` или `float` и функциональный оператор, вычисляющий значение по физическим координатам:

```
  value_type operator()(value_type x, value_type y, value_type z)
const;
```

Экземпляры функциональных объектов `f`, `x`, `y`, `z` создаются внутри экземпляра самой вычисляемой сеточной функции конструкторами без параметров. При необходимости настроить объекты можно, получив ссылки на них с помощью методов `get_f`, `get_x`, `get_y`, `get_z`.

Вычисляемая сеточная функция позволяет получить двумерный срез по любой оси с помощью методов `get_slice_x`, `get_slice_y`, `get_slice_z`. По

умолчанию ширина среза равна единице, но может быть любой. Значения в срезе вычисляются «на лету».

Пример работы с вычисляемой сеточной функцией можно посмотреть в примерах teplo и teplo_cuda, распространяемых вместе с библиотеками.

Класс *cuda_computable_grid_function* в библиотеке *gridmath_cuda* отличается от класса *computable_grid_function* только названием и тем, что он пронаследован от класса *cuda_grid_function*.

4.6 Класс dense_grid_function_base

Класс *dense_grid_function_base* (*cuda_dense_grid_function_base* в библиотеке *gridmath_cuda*) является базовым классом для всех плотных сеточных функций.

Описание класса *dense_grid_function_base*:

```
    template<typename T, class GF, class _Proxy>
    struct dense_grid_function_base : grid_function<GF, _Proxy>,
std::vector<T>;
```

В качестве параметра шаблона `T` класс *dense_grid_function_base* принимает тип хранимых данных, `GF` – имя конечного класса плотной сеточной функции, `_Proxy` – имя класса-заместителя. Параметр шаблона `_Proxy` обязательный. Класс *dense_grid_function_base* пронаследован и от класса *grid_function*, и от класса `std::vector` из стандартной библиотеки C++, в котором реализовано хранение данных. Тип хранимых данных может быть любым, не обязательно числовым. При хранении данных в векторе наиболее быстро меняющийся индекс первый, затем второй и, в конце концов, третий.

Класс *cuda_dense_grid_function_base* из библиотеки *gridmath_cuda* отличается главным образом тем, что для хранения данных использует класс `thrust::device_vector` из библиотеки *thrust*, поставляемой вместе с CUDA SDK. Из названия класса `device_vector` видно, что данные располагаются в памяти графического вычислителя, т.е. для хранения данных память host-процессора не используется. Все вычисления также производятся непосредственно на графическом вычислителе. Если какие–либо данные нужно получить на host–процессоре (например, для сохранения в файле или для обмена по сети), то данные нужно явно копировать в host–процессор.

Конструктор по умолчанию создаёт плотную сеточную функцию нулевого размера по всем трём осям, нужный размер можно задать позже с помощью метода `resize()`. Имеется также конструктор с явным указанием размеров. Эти размеры позже также можно переопределить.

Функциональный оператор `()` возвращают ссылку на хранимые данные. Для библиотеки *gridmath_cuda* каждое обращение к функциональному оператору

приводит к пересылке данных из памяти графического вычислителя в память host–процессора. Данная операция удобна, но чрезвычайно затратная по времени. Для копирования большого объёма данных она не годится. Если нужно скопировать весь массив данных из памяти графического вычислителя в память host–процессора, то наиболее эффективный способ следующий. Нужно создать переменную класса `thrust::host_vector<T>` и просто присвоить ей сеточную функцию. После этого получить указатель на массив данных можно с помощью функции *thrust::raw_pointer_cast*. Вот пример кода:

```
cuda_dense_grid_function<double> f(k, m, n);
… // заполняем f
f = 5.0;    // Например, так.
thrust::host_vector<double> hf = f;
double *p = thrust::raw_pointer_cast(&hf[0]);
```

Теперь переменная `p` хранит указатель на массив данных в host–процессоре, которые можно сохранить в файл или переслать по сети.

Если же нужны не все данные, а, например, один или несколько слоёв по какой–либо оси, то указанный способ также плох, т.к. он в этом случае копирует очень много лишних данных, что тоже крайне неэффективно. Для работы со слоями в классе *cuda_dense_grid_function* (описан ниже) есть специальные методы, если же нужен какой-то специфический набор данных, то правильно написать специальную функцию. При реализации такой функции в качестве примера можно взять реализацию работы со слоями в классе *cuda_dense_grid_function*.

Класс *cuda_dense_grid_function_base* в библиотеке *gridmath_cuda* отличается от класса *dense_grid_function_base* только названием, тем, что он пронаследован от класса *cuda_grid_function* и классом вектора для хранения данных. Вместо класса `std::vector` используется класс `thrust::device_vector`.

4.7 Класс dense_grid_function

Класс *dense_grid_function* (*cuda_dense_grid_function* в библиотеке *gridmath_cuda*) является классом плотной сеточной функцией, хранящей числовые данные.

Описание класса *dense_grid_function*:

```
template<typename T>
struct dense_grid_function_proxy;

template<typename T>
struct dense_grid_function : dense_grid_function_base<
    T, dense_grid_function<T>, dense_grid_function_proxy<T> >;
```

Класс *cuda_dense_grid_function* в библиотеке *gridmath_cuda* отличается от класса *dense_grid_function* названием и родительским классом (*cuda_dense_grid_function_base* вместо *dense_grid_function_base*).

Описание класса *cuda_dense_grid_function*:

```
template<typename T>
struct cuda_dense_grid_function_proxy;


template<typename T>
struct cuda_dense_grid_function : cuda_dense_grid_function_base<
    T, cuda_dense_grid_function<T>,
cuda_dense_grid_function_proxy<T> >;
```

В качестве параметра шаблона `T` класс *dense_grid_function* принимает тип хранимых данных. Этот тип должен быть числовым, т.е. для него должны быть определены арифметические операции.

Конструктор без параметров создаёт плотную сеточную функцию с нулевыми размерами по всем осям. Конструктор с параметрами создаёт плотную сеточную функцию заданного размера.

Заполнить плотную сеточную функцию можно несколькими способами. Во-первых, есть оператор присваивания, принимающий в качестве параметра скалярную величину. Данный оператор заполняет всю плотную сеточную функцию одним и тем же заданным значением. Есть оператор присваивания, принимающий другую плотную сеточную функцию. Это, по сути, оператор копирования плотных сеточных функций. Третий оператор присваивания принимает на вход вычисляемый объект. Это может быть вычислитель или произвольная (но не плотная) сеточная функция. Сеточная функция заполняется вычисленными данными. Этот третий оператор присваивания запускает отложенные «ленивые» вычисления. Последовательность вычислений значений в узлах сетки не определена и может производиться параллельно для различных узлов сетки.

Так как сеточная функция является вычисляемым объектом, то имеются арифметические операции со скалярными величинами и с другими вычисляемыми объектами, возвращающие вычислители.

Функция клонирования (*clone*) возвращает незаполненную (точнее, заполненную нулями) сеточную функцию того же размера.

Функции по работе с двумерными срезами по осям принимают вектор данных (`std::vector` в библиотеке *gridmath* и `thrust::device_vector` в библиотеке *gridmath_cuda*) или скалярную величину. По умолчанию ширина среза равна единице, но может быть любой. Работа со срезами может быть нужна или для копирования данных из одной сеточной функции в другую, или для обмена данными по сети. Если используется библиотека *gridmath_cuda* и

данные среза нужны для пересылки по сети, то данные среза нужно передать из памяти графического процессора в память host-процессора, для этого удобно использовать класс `thrust::host_vector`. Пересылка данных между этими двумя типами векторов (`thrust::device_vector` и `thrust::host_vector`) делается простым присваиванием. Вот пример кода, отправляющего данные по сети.

```
cuda_dense_grid_function<double> f;
… // Инициализация и заполнение сеточной функции f.
thrust::device_vector<double> slice;
// Считываем срез.
f.get_slice_x(0, slice);
// Копируем вектор из памяти CUDA в память host-процессора.
thrust::host_vector<double> hslice = slice;
// Получаем указатель массив данных в host-процессоре.
const double *p = thrust::raw_pointer_cast(&hslice.front());
// Размер данных – hslice.size()
… // Отправляем данные по сети.
```

Пример кода для обратного преобразования:

```
// Плотная сеточная функция
cuda_dense_grid_function<double> f;
… // Инициализация и заполнение сеточной функции f.
size_t slice_size = f.size_y() * f.size_z(); // размер среза
// Вектор нужного размера в памяти host-процессора.
thrust::host_vector<double> hslice(slice_size);
// Получаем указатель массив данных в host-процессоре.
double *p = thrust::raw_pointer_cast(&hslice.front());
… // Считываем данные из сети в вектор hslice.
// Копируем срез из памяти host-процессора в память CUDA.
thrust::device_vector<double> slice = hslice;
f.set_slice_x(0, slice);
```

4.8 Класс grid_operator

Класс *grid_operator* (*cuda_grid_operator* в библиотеке *gridmath_cuda*) – это базовый класс для всех сеточных операторов.

Описание класса *grid_operator*:

```
template<class GO, class _Proxy = GO>
struct grid_operator : grid_object<GO, _Proxy>
{
   typedef grid_object<GO, _Proxy> base;

   template<class EO>
   operator_evaluator<GO, EO>
   operator()(const grid_evaluatable_object<EO, typename
EO::proxy_type>& eobj) const;
```

```
};
```

В качестве параметра шаблона `GO` класс *grid_operator* принимает имя конечного класса оператора. Параметр шаблона `_Proxy` задаёт класс заместителя оператора. По умолчанию (если этот параметр не указан), класс заместителя совпадает с классом самого оператора.

Для операторов реализована арифметика. Любые два сеточных можно складывать и вычитать. При этом получается новый оператор. Кроме того, для операторов реализованы все четыре арифметических операции со скалярными величинами, причём скалярная величина может быть как в правой, так и в левой частях арифметической операции. Результатом также получается оператор.

Для операторов реализована композиция операторов. В качестве операции композиции используется переопределённая операция умножения языка C++. Результатом композиции операторов является оператор композиции, применяющий первый (левый) оператор к результату вычисления второго (правого) оператора.

Для реализации композиции операторов был введен новый объект – вызов оператора (operator invocation). Его реализует в библиотеке *gridmath* реализует класс *operator_invocation*, а в библиотеке *gridmath_cuda* – класс *cuda_operator_invocation*. С помощью этого объекта фактически осуществляются рекурсивные вызовы операторов. Класс *operator_invocation* в своём функциональном методе вызывает функциональный метод в операторе (точнее, в его заместителе), а тот, в свою очередь, вызывает функциональный метод в другом объекте класса *operator_invocation*. Для завершения этой рекурсии существует специализация класса *operator_invocation*, не использующая операторов, а обращающаяся напрямую к вычисляемому объекту. Этому специализированному классу соответствует класс (определённый через `typedef`) *value_invocation*.

Класс *cuda_grid_operator* в библиотеке *gridmath_cuda* отличается от класса *grid_operator* только названием и тем, что он пронаследован от класса *cuda_grid_object*.

4.9 Класс value_operator

Класс *value_operator* (*cuda_value_operator* в библиотеке *gridmath_cuda*) – это сеточный оператор, не меняющий значения того, что он вычисляет, или единичный оператор.

Этот класс может быть полезен при формировании составных операторов. Приведём пример. Путь `f` – сеточная функция, `g` – плотная сеточная функция, `A` – сеточный оператор. Тогда можно написать такой код:

```
  value_operator<double> I;   // Заводим переменную под единичный
оператор.
  g = (I + A)(f);   // Аналогично g=f+A(f), только сложение
делается на уровне арифметики операторов, а не вычислителей.
```

Класс *cuda_value_operator* в библиотеке *gridmath_cuda* отличается от класса *value_operator* только названием и тем, что он пронаследован от класса *cuda_grid_operator*.

5. Пример teplo

В данном примере ищется стационарное решение трёхмерной задачи теплопроводности:

$$\frac{\partial U}{\partial t} = k\Delta U + f$$

Ищется приближённое решение для заданного точного решения:

$$U_0 = \sin lx \ \sin my \ \sin nz .$$

В этом случае $\frac{\partial U}{\partial t} = 0$ и $f = k\ (l^2 + m^2 + n^2)\ U_0$.

Задача решается в кубе $\Omega = [0,1]\times[0,1]\times[0,1]$, поделённом на N частей в каждом направлении. Таким образом, число расчетных узлов в каждом направлении равно $N+1$. Итерации ведутся по времени на интервале $t_{\max} = 0.1$ с шагом $\tau = \frac{h^2}{24k}$. Простейший итерационный метод взят из соображений удобства иллюстрации действия операторов. Предполагается и уже частично реализуется практическое применение операторной библиотеки для решения содержательных стационарных и эволюционных задач.

В файле discrete_laplas_operator.hpp реализован дискретный оператор Лапласа по семиточечной явной схеме:

$$u_{i,j,k}^{t+\tau} = \frac{u_{i-1,j,k}^{t} + u_{i+1,j,k}^{t} + u_{i,j-1,k}^{t} + u_{i,j+1,k}^{t} + u_{i,j,k-1}^{t} + u_{i,j,k+1}^{t} - 6u_{i,j,k}^{t}}{h^2}$$

При этом для точек на границе полагаем $u_{i,j,k}^{t+\tau} = u_{i,j,k}^{t}$.

Точное решение U_0 и функция f реализованы с помощью вычисляемой сеточной функции. При этом для точного решения в качестве класса F используется следующий класс `my_U`:

```
  template<typename T, class C>
  struct my_U_base : grid_object<C>
  {
     typedef grid_object<C> base;
     typedef T value_type;

     void init(value_type l_, value_type m_, value_type n_)
     {
       l = l_;
       m = m_;
       n = n_;
     }

     value_type operator()(value_type x, value_type y, value_type
z) const {
       return std::sin(l * x) * std::sin(m * y) * std::sin(n * z);
     }

  protected:
     value_type l, m, n;
  };

  template<typename T>
  struct my_U : my_U_base<T, my_U<T> >{};
```

Для функции *f* в качестве класса F используется следующий класс:

```
  template<typename T>
  struct my_f : my_U_base<T, my_f<T> >
  {
     typedef my_U_base<T, my_f<T> > base;
     typedef typename base::value_type value_type;

     void init(value_type k, value_type l, value_type m, value_type
n)
     {
       base::init(l, m, n);
       beta = k * (l * l + m * m + n * n);
     }

     value_type operator()(value_type x, value_type y, value_type
z) const {
       return beta * base::operator()(x, y, z);
     }

  private:
     value_type beta;
  };
```

Для обеих функций (U_0 и f) в качестве классов `X`, `Y` и `Z` используется следующий класс:

```
  template<typename T>
  struct i_to_x : grid_object<i_to_x<T> >
  {
     typedef grid_object<i_to_x<T> > base;
     typedef T value_type;

     void init(value_type h_)
     {
       h = h_;
     }
     value_type operator()(size_t i) const {
       return i * h;
     }

  private:
     value_type h;
  };
```

Дискретный оператор Лапласа реализован следующим классом:

```
  template<typename T>
  struct discrete_laplas_operator :
grid_operator<discrete_laplas_operator<T> >
  {
     typedef grid_operator<discrete_laplas_operator<T> > base;
     typedef T value_type;

     void init(value_type h, size_t size_x_, size_t size_y_, size_t
size_z_)
     {
       h2 = 1 / (h * h);
       size_x = size_x_;
       size_y = size_y_;
       size_z = size_z_;
     }

     template<class EOP, class OI>
     value_type operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy, const OI& oi) const
     {
       return (
            oi(i-1, j, k, eobj_proxy) + oi(i + 1, j, k, eobj_proxy)
          + oi(i, j-1, k, eobj_proxy) + oi(i, j + 1, k, eobj_proxy)
          + oi(i, j, k-1, eobj_proxy) + oi(i, j, k + 1, eobj_proxy)
          - 6 * oi(i, j, k, eobj_proxy)
        ) * h2;
     }
```

```
    template<class EO>
    operator_evaluator<discrete_laplas_operator, EO>
    operator()(const grid_evaluatable_object<EO, typename
EO::proxy_type>& eobj) const {
      return base::operator()(eobj);
    }

  private:
    value_type h2;
    size_t size_x, size_y, size_z;
  };
```

Вот как создаётся и инициализируется вычисляемая сеточная функция для точного решения:

```
  computable_grid_function<
    my_U<data_type>,
    i_to_x<data_type>,
    i_to_x<data_type>,
    i_to_x<data_type>
  > cUe(N + 1, N + 1, N + 1); // Точное решение
  cUe.get_f().init(l, m, n);
  cUe.get_x().init(h);
  cUe.get_y().init(h);
  cUe.get_z().init(h);
```

Вот как создаётся и инициализируется функция *f*:

```
  computable_grid_function<
    my_f<data_type>,
    i_to_x<data_type>,
    i_to_x<data_type>,
    i_to_x<data_type>
   > cf(N + 1, N + 1, N + 1);     // Правая часть.
   cf.get_f().init(k, l, m, n);
   cf.get_x().init(h);
   cf.get_y().init(h);
   cf.get_z().init(h);
```

Основная плотная сеточная функция и её инициализация граничными значениями:

```
   dense_grid_function<data_type> u1(N + 1, N + 1, N + 1);
   std::vector<data_type> slice;
   cUe.get_slice_x(0, slice);
   u1.set_slice_x(0, slice);
```

```
cUe.get_slice_x(cUe.size_x() - 1, slice);
u1.set_slice_x(u1.size_x() - 1, slice);
cUe.get_slice_y(0, slice);
u1.set_slice_y(0, slice);
cUe.get_slice_y(cUe.size_y() - 1, slice);
u1.set_slice_y(u1.size_y() - 1, slice);
cUe.get_slice_z(0, slice);
u1.set_slice_z(0, slice);
cUe.get_slice_z(cUe.size_z() - 1, slice);
u1.set_slice_z(u1.size_z() - 1, slice);
```

Вспомогательная плотная сеточная функция:

```
dense_grid_function<data_type> u2 = u1.clone();
u2 = u1;

dense_grid_function<data_type> *u = &u1, *v = &u2;
```

Дискретный оператор Лапласа и его инициализация:

```
discrete_laplas_operator<data_type> delta;
delta.init(h);
```

Сеточный диапазон:

```
 grid_range_info ginfo;
 ginfo.init(1, 1, 1, 1, 1, 1);
```

Норму ошибки вычисляет функция norm2:

```
ginfo(*v) = k * delta(*u) + cf;     // Ошибка в переменной v
data_type norm = norm2(*v);
```

Основной цикл программы выглядит так:

```
size_t niter = 0;
for(data_type t = 0; t < tmax; t += tau, ++niter){
 ginfo(*v) = *u + tau * (k * delta(*u) + cf);
 std::swap(u, v);    // Меняем местами указатели
 // Считаем текущую норму ошибки.
 ginfo(*v) = k * delta(*u) + cf;    // Ошибка в переменной v
 // Норма ошибки.
 data_type norm1 = norm2(*v);
 if(norm1 > norm)break; //Норма ошибки увеличивается, выходим.
 norm = norm1;
}
```

Выход из цикла происходит в одном из двух случаев: или если мы дошли до максимального времени, или если значение нормы ошибки стало увеличиваться.

В конце основного цикла вычисленное значение находится в плотной сеточной функции, на которую указывает переменная `u`.

Функция norm2 вычисляет норму внутренних точек сеточной функции:

```
data_type norm2(const dense_grid_function<data_type>& u)
{
  data_type result = data_type();
  for(size_t k = 1, k_last = u.size_z() - 1; k < k_last; ++k)
     for(size_t j = 1, j_last = u.size_y() - 1; j < j_last; ++j)
       result = std::inner_product(&u(1, j, k), &u(u.size_x() - 1,
j, k), &u(1, j, k), result);
  return std::sqrt(result / ((u.size_x() - 2) * (u.size_y() - 2) *
(u.size_z() - 2)));
}
```

Пример для библиотеки *gridmath_cuda* отличается главным образом тем, что классы, используемые для вычисляемой сеточной функции, должны быть пронаследованы от класса cuda_grid_object. Наиболее существенное отличие – в функции norm2. Вычисление нормы производится параллельно. Функция отличается довольно сильно и занимает существенно больше места, поэтому здесь не приводится.

6. Заключение

Рассмотрим вопрос об эффективности композиции сеточных операторов. Пусть `A`, `B` – сеточные операторы, `f`, `g`, `h` – плотные сеточные функции. Рассмотрим следующий оператор: `g=B(A(f))`. В этом операторе сеточный оператор `B` применяется к результату применения сеточного оператора `A`. Эти вычисления можно провести и по-другому. Можно вычислить `A(f)`, сохранив результат в промежуточной плотной сеточной функции, и затем к этой промежуточной сеточной функции применить оператор `B`, т.е. написать: `h=A(f); g=B(h)`. Хотя этот второй способ и требует больше памяти, но из общих соображений ясно, что он будет работать быстрее. Вопрос: насколько быстрее? Был проведён следующий вычислительный эксперимент: вычислялось выражение: `h=lambda(f+g);` Здесь `f`, `g`, `h` – плотные сеточные функции, `lambda` – дискретный оператор Лапласа, вычисляемый по семиточечной явной схеме. В этом выражении сумма в каждом узле сетки будет вычисляться семь раз. Альтернатива состоит в том, чтобы вначале вычислить сумму, а затем к ней применить оператор Лапласа: `tmp=f+g; h=lambda(tmp);` Вычисления проводились на сетке размером 256 точек в

каждом направлении последовательно и на CUDA. На супер-ЭВМ К-100 первый вариант оказался медленнее второго в последовательном варианте на 25%, а в варианте CUDA – на 35%. Таким образом, у прикладного программиста, использующего данную библиотеку, есть выбор: использовать дополнительную оперативную память для промежуточных переменных и за счёт этого несколько уменьшить время исполнения программы или не использовать. Всё должно определяться тем, достаточно ли оперативной памяти.

7. Литература

1. В.Т. Жуков, Н.Д. Новикова, О.Б. Феодоритова. Параллельный многосеточный метод для разностных эллиптических уравнений. Часть I. Основные элементы алгоритма // Препринты ИПМ им. М.В. Келдыша. — 2012. – № 30. – 32 с. –
   URL: http://www.keldysh.ru/papers/2012/prep2012_30.pdf
2. Template metaprogramming.
   URL: http://en.wikipedia.org/wiki/Template_metaprogramming
3. Язык программирования НОРМА. URL: http://keldysh.ru/pages/norma/
4. Boost. URL: http://www.boost.org/
5. Nvidia CUDA. URL: http://www.nvidia.com/object/cuda_home_new.html
6. Thrust. URL: http://thrust.github.com/

Оглавление