ОПЕРАТОРНАЯ БИБЛИОТЕКА ДЛЯ РЕШЕНИЯ ТРЁХМЕРНЫХ СЕТОЧНЫХ ЗАДАЧ МАТЕМАТИЧЕСКОЙ ФИЗИКИ С ИСПОЛЬЗОВАНИЕМ ГРАФИЧЕСКИХ ПЛАТ С АРХИТЕКТУРОЙ CUDA

© *2013 г.* ***М.М. Краснов***

Институт прикладной математики им. М.В. Келдыша РАН, Москва, Россия

Email: kmm@kiam.ru

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]):

$$Q = S_p(I - PA_H^{-1}RA_h)\,S_p,$$

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 часто 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.

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. В библиотеке *gridmath* все основные операции реализованы последовательно, а в библиотеке *gridmath_cuda* – параллельно.

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.

В дальнейшем изложении слово «сеточный» в словосочетаниях «сеточный оператор» и «сеточный вычислитель» часто будет опускаться, чтобы не загромождать текст.

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]). Основное отличие между библиотеками *gridmath* и *gridmath_cuda* заключается именно в реализации данного оператора присваивания. В библиотеке *gridmath* он реализован последовательно с помощью трёх вложенных циклов (по трём осям), а в библиотеке *gridmath_cuda* – параллельно путём вызова одного ядра в графическом ускорителе CUDA.

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.

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. В качестве «нуля» 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.

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.

Важный вопрос, который может возникнуть – как сделать первоначальное заполнение плотной сеточной функции на основе имеющейся функции, вычисляющей значение в узле по его координатам? Одно из возможных решений – завести массив нужного размера в памяти обычного процессора, заполнить его, а затем скопировать в память графического ускорителя. Однако наиболее оптимальное 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.

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` – оператор, то мы можем написать:

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

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

Для операторов определена своя арифметика. В алгебраических терминах можно сказать, что операторы образуют группу с групповой операцией 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. «Нулём» операции сложения может служить оператор, возвращающий значение 0 для каждого узла независимо от значения сеточной функции в данном узле.

Написание собственных «конечных» операторов – главная задача программиста, использующего данную библиотеку.

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 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 2.0 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
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` – сеточный 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` – сеточные 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.

3.2. 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 считать 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 векторы 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 плотной сеточной функции *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()`.

3.3. Реализация оператора присваивания для CUDA

Как уже говорилось выше, основным отличием между библиотеками *gridmath* и *gridmath_cuda* является реализация оператора присваивания вычислителя плотной сеточной функции. Так как конкретный класс вычислителя до компиляции неизвестен, то данный оператор должен быть шаблонным, и конкретный класс вычислителя должен передаваться как параметр шаблона. То же самое относится и к запуску ядра CUDA. Функция, реализующая ядро, также должна быть шаблонной. Идея того, как это может быть реализовано, была взята автором из библиотеки thrust из состава CUDA SDK, которая решает аналогичные проблемы (см. [4]).

Для запуска ядра требуется создать т.н. «замыкание» (closure). Это объект, содержащий в себе всё, что нужно для исполнения тела ядра. Все необходимые параметры передаются в конструктор объекта и сохраняются в member-переменных. Для исполнения ядра в классе замыкания должен быть следующий функциональный оператор:

```
__device__
```

```
void operator()(size_t i, size_t j, size_t k);
```

В этом операторе i, j, k – целочисленные координаты узла сетки, в котором нужно произвести вычисления.

Для исполнения ядра нужно создать объект-замыкание и передать его в функцию `launch_closure_3d`. Помимо замыкания, в эту функцию передаются размеры сеточной функции. Вот прототип этой функции:

```
template<class Closure>
void launch_closure_3d(Closure& closure,
  size_t size_x, size_t size_y, size_t size_z);
```

Функция `launch_closure_3d` запускает следующее ядро:

```
template<class Closure>
__global__
void launch_3d(Closure closure){
  closure(blockIdx.x * blockDim.x + threadIdx.x, blockIdx.y * blockDim.y
  + threadIdx.y, blockIdx.z * blockDim.z + threadIdx.z);
}
```

Таким образом, в замыкании вызывается функциональный оператор, которому передаются координаты узла сетки. Замыкание копируется из адресного пространства host-процессора в адресное пространство графического устройства. Из этого, в частности, следует, что объект-замыкание должен быть «лёгким» для копирования, т.е. должно содержать переменные только простых типов (в т.ч. указатели).

4. Заключение

Была разработана операторная библиотека для решения трёхмерных задач математической физики. Библиотека была реализована в двух вариантах – последовательном и для графических ускорителей CUDA. Оба варианта библиотеки показали высокую эффективность. Последовательный вариант библиотеки работает примерно с той же скоростью, что и аналогичная программа, написанная на языке Фортран, при этом исходный текст программы существенно короче и понятнее за счёт использования сеточных операторов. При использовании варианта библиотеки для графических ускорителей CUDA программа ускоряется приблизительно до 8 раз по сравнению с последовательным вариантом (на супер-ЭВМ К–100). Ускорение зависит от размера задачи. Для маленьких размеров сетки ускорение менее существенно. Чем больше сетка, тем больше ускорение. Это объясняется тем, что вызов ядра – дорогостоящая операция и имеет смысл производить как можно больше вычислений на каждый вызов ядра.

В целом библиотека существенно упрощает написание программ, решающих задачи на трёхмерных сетках. Текст программ становится существенно короче и понятнее. Благодаря этому уменьшается количество возможных ошибок в программах и время на разработку и отладку. При этом программы работают очень эффективно и не требуют дополнительной памяти для промежуточных вычислений.

Особенно эффективно использование библиотеки для программирования на графических ускорителях CUDA. От прикладного программиста не требуется глубоких знаний о программировании для графических ускорителей, всю работу берёт на себя библиотека.

В 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? Автором 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.

СПИСОК ЛИТЕРАТУРЫ

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. http://www.keldysh.ru/papers/2012/prep2012_30.pdf
2. Template metaprogramming. URL: http://en.wikipedia.org/wiki/Template_metaprogramming
3. Boost. URL: http://www.boost.org/
4. Nvidia CUDA. URL: http://www.nvidia.com/object/cuda_home_new.html
5. Thrust. URL: http://thrust.github.com/