Российская академия наук
Институт прикладной математики им. М.В. Келдыша РАН

На правах рукописи

Краснов Михаил Михайлович

СЕТОЧНО-ОПЕРАТОРНЫЙ ПОДХОД К ПРОГРАММИРОВАНИЮ ЗАДАЧ МАТЕМАТИЧЕСКОЙ ФИЗИКИ

05.13.11 - математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель –
доктор физико-математических наук
Жуков Виктор Тимофеевич

Москва – 2016

Оглавление

Введение ... 4
Актуальность темы. ... 4
Цели работы. ... 5
Постановка задачи. ... 6
Научная новизна работы. ... 7
Практическая значимость. ... 8
Положения, выносимые на защиту. ... 10
Публикации. ... 10
Краткое содержание работы. ... 11
Глава 1. Обзор работ, направленных на упрощение записи и облегчение переноса программ ... 12
1.1. Язык Норма ... 13
1.2. Система DVM ... 15
1.3. Язык Liszt ... 18
1.4. Библиотека Blitz++ ... 20
1.5. Библиотека PETSc ... 22
1.6. Пакет OpenFOAM ... 22
Глава 2. Общее описание подхода ... 24
2.1. Назначение сеточно-операторного подхода ... 24
2.2. Типы обрабатываемых данных ... 27
2.3. Поддержка автоматического дифференцирования ... 29
2.4. Сеточная функция ... 31
2.5. Вычисляемый объект ... 32
2.6. Сеточный оператор ... 32
2.7. Сеточный вычислитель ... 33
2.8. Исполнители ... 33
2.9. Индексаторы ... 34
2.10. Общий вид запуска вычислений ... 37
2.11. Примеры ... 39
Глава 3. Принципы реализации ... 40
3.1. Общая информация ... 40
3.2. Шаблонный полиморфизм ... 40
3.3. Метавычисления ... 42
3.4. Размерные величины ... 44
3.5. Объекты-заместители ... 47
3.6. Контекст исполнения ... 50
3.7. Исполнение на CUDA ... 52
3.8. Обмен данными между основным процессором и графическим ускорителем ... 54
3.9. Использование пула нитей. ... 55
Глава 4. Приложения и тесты, разработанные с использованием библиотеки gridmath ... 56

4.1. Многосеточный метод 57
4.2. Тест MG из набора тестов NAS Parallel Benchmarks 59
4.3. Система квазигазодинамических уравнений 60
4.4. Локально-адаптивная сетка 64
4.5. Задача теплопроводности 65
4.6. Разрывный метод Галёркина 70
4.7. Метод ENO 73
Заключение 73
Литература 74
Приложение 1. Описание программного интерфейса сеточно-операторной библиотеки 81
Общая информация. 81
Базовый класс math_object 82
Класс grid_evaluable_object (вычисляемый объект) 83
Класс grid_operator (сеточный оператор) 84
Класс negate (оператор отрицания) 88
Класс shift (оператор сдвига) 90
Класс grid_value_operator (единичный оператор) 91
Классы grad (оператор градиента) и div (оператор дивергенции) 93
Класс grid_function (сеточная функция) 96
Класс scalar_grid_function (скалярная сеточная функция) 97
Класс computable_grid_function (вычисляемая сеточная функция) 98
Плотные сеточные функции 102
Класс abstract_dense_grid_function 103
Класс dense_grid_function_base 103
Класс simple_dense_grid_function (простая плотная сеточная функция) 105
Класс dense_grid_function (плотная сеточная функция) 106
Присваивание выражений плотной сеточной функции 110

Введение

Актуальность темы.

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 математической литературе эти преобразования описываются операторами, такими, например, как оператор Лапласа, градиента, дивергенции, ротора. 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 (число применений оператора A_h), 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.

Ещё одной немаловажной проблемой является то, что в настоящее время большое распространение получили вычислительные комплексы с нетрадиционной (не фон-Неймановской) архитектурой, основу вычислительной мощности которых составляют графические ускорители NVIDIA CUDA (Compute Unified Device Architecture, см. [2]) или новейшие процессоры Intel Xeon Phi с 60 ядрами на одном кристалле (см.

[3]). Например, в списке top500 самых мощных суперкомпьютеров (см [4]) за ноябрь 2016 года на первом месте стоит суперкомпьютер Sunway TaihuLight (Китай – NRCPC), основанный на многоядерных процессорах Sunway SW26010 260C, на втором – Tianhe-2 (Китай - NUDT), основанный на процессорах Intel Xeon Phi 31S1P, а на третьем – Titan (США - Cray Inc), основанный на графических ускорителях NVIDIA K20x. Многие Российские суперкомпьютеры, например, самый мощный Российский суперкомпьютер «Ломоносов» (см. [5]) и вычислительный кластер К-100 в ИПМ им. М.В. Келдыша РАН (см. [6]) также содержат графические ускорители. Например, на К-100 на каждом из 64-х узлов, помимо двух основных процессоров Intel Xeon X5670 стоят по три платы NVIDIA Fermi C2050 (по 448 GPU и 2,8 Гбайт памяти на каждом). Эффективное же использование этих весьма внушительных вычислительных мощностей затруднено, т.к. требует освоения большого объёма новой и непривычной информации о методах программирования на них.

Цели работы.

Целями данной диссертационной работы являются:

- Разработка подхода к программированию, позволяющего:
    - Компактно записывать и эффективно программно реализовывать класс математических формул, в частности, введение в программы понятия «сеточного оператора», аналогичного математическому понятию оператора.
    - Единообразно реализовывать подход на разных типах сеток и для различных вычислительных архитектур;
- Разработка экспериментального программного комплекса, показывающего возможность реализации данного подхода для решения широкого класса задач математической физики.

Постановка задачи.

Обзор существующих на настоящее время работ (см. ниже) показывает, что на настоящее время нет готовых систем, реализующих в полной мере обе основные поставленные цели, а именно – упрощённую запись математических вычислений и поддержку новых вычислительных архитектур. Хотя работы в этом направлении ведутся, в частности, идёт разработка новой реализации языка НОРМА (см. [7-12]) с поддержкой графических ускорителей. Также поддержка графических ускорителей реализована в модели DVMH системы DVM (см. [13]).

Ставилась задача разработать подход к программированию, позволяющий реализовать его в виде библиотеки классов для языка C++, а не нового языка (как НОРМА) или препроцессора (как DVM). Это облегчает использование системы, так как не требует отдельного прохода при компиляции и не закрывает все остальные возможности языка C++. Вычисляться должны целиком величины, определённые на одно, двух или трёхмерных областях (в терминах языка НОРМА), а не отдельные их элементы. Основным оператором при использовании библиотеки должен быть оператор присваивания вида `f=e;` где `f` – сеточная функция, `e` – сеточное выражение (выражение с другими сеточными функциями). Выражение может быть любой сложности, при этом фактические вычисления в соответствие с этим выражением должны выполняться только в операторе присваивания, а до этого накапливаться. Обеспечить это можно методами шаблонного метапрограммирования языка C++ (см [14, 15]).

Так как в качестве целевой машины для использования сеточно-операторного подхода к программированию рассматривался кластер с гетерогенной архитектурой, то необходимо было обеспечить работу библиотеки на графических ускорителях NVidia CUDA. С появлением процессора Intel Xeon Phi была поставлена задача о переносе библиотеки и на этот процессор.

Научная новизна работы.

1. Разработан новый подход к программированию для класса математических вычислений на различных типах сеток, как регулярных, так и произвольных нерегулярных. Основой данного подхода является введённое понятие программного сеточного оператора, аналогичного математическому понятию оператора. Возможности программного оператора существенно превышают возможности традиционных шаблонов (таких, как, например, шаблон частной производной нужной степени и точности). По сути, программный оператор – это функция произвольной сложности с нужным числом аргументов.

2. В рамках разработанного подхода вводится новая система понятий, обеспечивающая эффективную реализацию подхода. Эта система понятий, помимо ключевого понятия программного сеточного оператора, также включает понятия вычисляемого объекта (evaluable object), вычислителя (evaluator), сеточной функции (grid function), исполнителя вычислений (executor), индексатора (index), заместителя (proxy). Введение в подход чётко определённых понятий позволяет единообразно реализовать его на разных типах сеток и для различных параллельных вычислительных архитектур с общей памятью. В частности, реализации подхода для локально-адаптивных сеток и для трёхмерных нерегулярных тетраэдральных сеток были написаны на основе реализации для регулярных трёхмерных сеток. Процесс переноса подхода на новый тип сетки, конечно, неформальный и творческий, но несложный.

3. Разработан комплекс программных библиотек, демонстрирующих возможность эффективной программной реализации предложенного подхода к программированию для различных типов сеток. Данный подход апробирован на регулярных прямоугольных

трёхмерных сетках на многосеточном методе, на локально-адаптивных сетках на задаче теплопроводности, на тетраэдральных трёхмерных сетках на разрывном методе Галёркина и на одномерных сетках на задаче ENO (Essentially Non-Oscillatory). Для каждого типа сетки была реализована своя программная сеточная библиотека, реализующая данный подход. Все версии библиотеки были перенесены на параллельную платформу NVIDIA CUDA, а также на OpenMP (см. [16]), что даёт возможность запуска программ на Intel Xeon Phi в «native» режиме. Путём простой перекомпиляции удаётся ускорить программы на CUDA в 4-8 раз по сравнению с последовательной версией (на суперкомпьютере К-100).

Выбранная архитектура системы позволила легко осуществить перенос библиотеки на нетрадиционные архитектуры, т.к. реализация упомянутого выше оператора присваивания, запускающего вычисления, скрыта от пользователя. Это значит, что реализовать его (а именно, обход всех точек сеточной функции в левой части оператора присваивания) можно по-разному для разных вычислительных архитектур. В последовательном случае это система вложенных циклов, а в параллельном – тот или иной вид (в зависимости от архитектуры системы) параллельного обхода точек.

Программы, написанные с использованием данного подхода к программированию, показали, во-первых, высокую переносимость исходных текстов на нетрадиционные архитектуры, включая NVidia CUDA и Intel Xeon Phi, и, во-вторых, неплохую производительность самой библиотеки.

Практическая значимость.

В соответствие с предлагаемым подходом написана сеточно-операторная библиотека gridmath для трёхмерных регулярных

прямоугольных сеток, с помощью которой перенесена на графические ускорители NVIDIA CUDA программа, реализующая параллельный многосеточный метод в рамках проекта для Института проблем безопасного развития атомной энергетики РАН. Программа запускалась на кластере К-100 в ИПМ им. М.В. Келдыша РАН, причём процессы обменивались между собой по протоколу обмена сообщениями MPI, а внутри процесса счёт вёлся на графическом ускорителе NVIDIA CUDA. Каждый узел К-100 содержит по три таких ускорителя, использовались все три ускорителя на каждом узле.

На библиотеку gridmath (версия для регулярных трёхмерных сеток) был также перенесён тест MG из набора тестов NAS Parallel Benchmarks и задача решения системы квазигидродинамических уравнений QGD3D. Обе эти программы показали высокую переносимость исходного текста, написанного с использованием сеточно-операторной библиотеки, на ускорители NVIDIA CUDA, а также на OpenMP с последующим запуском на процессорах Intel Xeon Phi.

Затем подход был успешно реализован на локально-адаптивных сетках внутри прямоугольного параллелепипеда, и с помощью неё решена задача диффузии, в которой внутрь параллелепипеда помещалось другое тело (например, шар или цилиндр), на границе которого был скачок коэффициента диффузии. На границе внутреннего тела сетка сгущалась.

Библиотека была также перенесена на тетраэдральные трёхмерные сетки и с её помощью было решено уравнение Эйлера разрывным методом Галёркина. Также был реализован метод ENO для одномерных сеток. Последние три примера показывают, что в соответствие с предлагаемым подходом можно относительно легко реализовать программную библиотеку для любых типов сеток.

На примере трёхмерной задачи теплопроводности реализованы несколько параллельных алгоритмов решения задач, включая методы Якоби, Гаусса-Зейделя, Чебышевские методы и многосеточный метод.

Положения, выносимые на защиту.

1. Разработан новый подход к программированию для класса математических вычислений на различных типах сеток.
2. В рамках разработанного подхода введена новая система понятий, обеспечивающая эффективную реализацию подхода.
3. Разработан комплекс программных библиотек, демонстрирующих возможность эффективной программной реализации предложенного подхода к программированию для различных типов сеток.

Публикации.

По теме диссертации были сделаны девять докладов на научных конференциях:

1. Международной научной конференции «Научный сервис в сети Интернет: экзафлопсное будущее», сентябрь 2011 г., г. Новороссийск.
2. Национальном суперкомпьютерном форуме (НСКФ-2013), ноябрь 2013 г., Переславль-Залесский, ИПС им. А.К. Айламазяна РАН.
3. XX Всероссийской конференции «Теоретические основы и конструирование численных алгоритмов решения задач математической физики», посвященная памяти К.И. Бабенко, сентябрь 2014 г., г. Новороссийск.
4. Международной конференции «Суперкомпьютерные дни в России», 28 - 29 сентября 2015 г., г. Москва.
5. XIV Международный семинар «Математические модели и моделирование в лазерно-плазменных процессах и передовых научных технологиях», 4 - 9 июля 2016 г., г. Москва.
6. VII всероссийская научная молодежная школа-семинар «Математическое моделирование, численные методы и комплексы программ» имени Е. В. Воскресенского с международным участием, 12 - 15 июля 2016 г., г. Саранск.

7. XXI Всероссийская конференция «Теоретические основы и конструирование численных алгоритмов для решения задач математической физики», посвященная памяти К.И.Бабенко, 5-11 сентября 2016 г., г. Новороссийск.
8. XV International seminar «Mathematical models & modeling in laser plasma processes & advanced science technologies», 26 - 30 September, 2016, Montenegro, Petrovac.
9. XVI Международная конференция «Супервычисления и математическое моделирование», 3-7 октября 2016, г. Саров.

Имеется 19 публикаций [17-35], из которых четыре [23-27] – в журналах из списка ВАК.

Краткое содержание работы.

В первой главе приводится обзор существующих работ, направленных, с одной стороны, на облегчение программирования на новых нетрадиционных компьютерных архитектурах, а с другой стороны, на сокращение способа записи сложных математических формул.

Вторая глава посвящена общему описанию подхода, его назначению и области применимости, типам обрабатываемых данных. Вводятся основные термины, описываются взаимосвязи основных объектов.

В третьей главе описываются принципы реализации подхода, использованные технологии. Более подробно описываются интерфейсы основных объектов.

Четвёртая глава содержит описание приложений, написанных с использованием сеточно-операторного подхода к программированию.

Работа выполнена под руководством доктора физико-математических наук, заведующего отделом Института прикладной математики им. М.В. Келдыша РАН Жукова Виктора Тимофеевича, которому автор выражает искреннюю признательность.

Автор также выражает благодарность Илюшину Александру Ивановичу, моему первому научному руководителю в далёкие уже

студенческие годы, Лацису Алексею Оттовичу за первые наставления в области параллельного программирования и полезные обсуждения и сотрудникам 8 отдела ИПМ им. М.В. Келдыша РАН Феодоритовой Ольге Борисовне и Новиковой Наталье Дмитриевне за помощь и поддержку.

Глава 1. Обзор работ, направленных на упрощение записи и облегчение переноса программ

Сеточно-операторный подход к программированию стоит на стыке двух направлений. Первое направление связано с тем, что освоение методов программирования на новых нетрадиционных архитектурах является непростой задачей, особенно для прикладных математиков. С другой стороны, освоение этих архитектур необходимо, т.к. новейшие суперкомпьютеры оснащаются такими вычислителями и обидно их не использовать. Одним из важных направлений развития системного программного обеспечения становится создание систем, упрощающих для прикладного программиста написание программ, использующих вычислительные возможности таких суперкомпьютеров с новейшими вычислителями. Основная цель при создании подобных систем – обеспечить переносимость исходного текста программ на эти новые архитектуры. При этом исходный текст может снабжаться псевдокомментариями или прагмами, с помощью которых можно управлять работой специализированных компиляторов. Или же один и тот же исходный текст можно компилировать разными компиляторами и получать код для разных архитектур. Такие работы активно ведутся в том числе и в ИПМ им. М.В. Келдыша РАН. Например, язык НОРМА (см. [7, 8]) и система DVM (см. [13]). Среди других работ можно упомянуть высокоуровневый язык Liszt (см. [36]) и OpenACC (см. [37]).

Второе направление связано с попыткой упростить написание сложных программ за счёт выноса часто повторяющегося кода в отдельные

модули или классы и затем многократного их использования. При этом исходный текст становится более структурированным и понятным. Так как выносимый код обычно используется внутри многократно повторяющегося тела цикла, важным становится вопрос эффективности. В частности, этот код нельзя выносить в отдельные обычные функции, т.к. вызов функции – дорогая по времени операция. При программировании на языке C++ функции могут быть объявленными инлайновыми (inline). В этом случае код этих функций вставляется компилятором в точке вызова. При использовании шаблонов функций и классов (использование шаблонного метапрограммирования - Template Metaprogramming) компилятор такие функции и методы таких классов делает инлайновыми автоматически. При хорошо работающем оптимизаторе полученный машинный код по своей производительности может не уступать коду, полученному с помощью компилятора с языка Фортран. Одной из лучших библиотек, эффективно использующих шаблонное метапрограммирование, является библиотека Blitz++ (см. [38, 39]).

Есть системы, упрощающие запись вычислений за счёт реализации матрично-векторной алгебры, в которых оперирование векторами и матрицами производится как с едиными объектами. К таким системам можно отнести, например, систему Matlab (см. [40]) и язык SIMIT (см. [41]). Запись сложных манипуляций с матрицами и векторами в этих системах очень компактна и не уступает по своей компактности записи в математической литературе.

Существуют также системы для решения задач математического моделирования с уже готовыми решателями различных задач. Как правило, пользователь может добавлять в них свои расширения. Среди таких систем можно упомянуть библиотеку PETSc (см. [42]) и пакет OpenFOAM (см. [43]).

1.1. Язык Норма

Язык Норма является средством, предназначенным для автоматизации решения задач математической физики на вычислительных системах с параллельной архитектурой.

Язык Норма позволяет исключить фазу программирования, которая необходима при переходе от расчетных формул, заданных прикладным специалистом, к программе. Между расчетными формулами и записью на Норме нет существенной разницы - эти формулы являются исходной информацией для транслирующей системы. Фактически, программа на Норме является непроцедурным описанием решаемой задачи. Математические проблемы, связанные с решением задачи синтеза выходной программы, в случае языка Норма разрешимы.

В программе на языке Норма задаются многомерные области и величины на них, а затем задаются связи между этими величинами. Связи между величинами можно задавать в любой последовательности, т.к. фактическая последовательность выполнения операторов определяется компилятором с учётом зависимостей между величинами. Например, если величина B зависит от величины A, а величина C зависит от величины B, то величина B будет вычислена раньше величины C независимо от последовательности операторов в программе. Компилятор с языка Норма в качестве результата своей работы выдаёт текст программы на языке C или Fortran, который затем нужно скомпилировать соответствующим компилятором. Компилятор может автоматически распределять данные между процессами и обмениваться данными по MPI. Планируется выход версии компилятора, проводящего вычисления на графических ускорителях CUDA. Приведём пример программы на языке Норма, решающую систему линейных уравнений методом Гаусса-Жордана:

```
! A solution of linear equations system by Gauss-Jourdan
method.
DOMAIN PARAMETERS n=100.
BEGIN
so:(ts:(t=0..n);ijs:(is:(i=1..n);js:(j=1..n))).
```

```
s1o:(ts;is). s2:((l=1);(k=1)).
s:so/ts-LEFT(1).
s1:s1o/ts-LEFT(1).
VARIABLE a DEFINED ON ijs.
VARIABLE m DEFINED ON so.
VARIABLE b,x DEFINED ON is.
VARIABLE r DEFINED ON s1o.
     INPUT a ON ijs.
     INPUT b ON is.
     DISTRIBUTION INDEX i=1..5,j=1..5.
     DOMAIN PARAMETERS n = 5.
     FOR s/t=0 ASSUME m = a.
     FOR s1/t=0 ASSUME r = b.
     sa,sb:s/i=t.
     sa1,sb1:s1/i=t.
     FOR sa ASSUME
          m = m[t-1,i=t]/m[t-1,i=t,j=t].
     FOR sa1 ASSUME
          r = r[t-1,i=t]/m[t-1,i=t,j=t].
     FOR sb ASSUME
          m = m[t-1]-m[t-1,j=t]*m[i=t].
     FOR sb1 ASSUME
          r = r[t-1]-m[t-1,j=t]*r[i=t].
     FOR is ASSUME
          x = r[t=n].
     OUTPUT x ON is.
END PART.
```

1.2. Система DVM

DVM-система предназначена для создания переносимых и эффективных вычислительных приложений на языках C-DVM и Fortran-DVM для многопроцессорных компьютеров с общей и распределенной памятью, включая и гибридные системы, в узлах которых вместе с универсальными многоядерными процессорами используются в качестве ускорителей и графические процессоры (модель DVMH).

Языки C-DVM и Fortran-DVM являются расширениями языков C и Fortran в соответствии с моделью DVM, разработанной в ИПМ им М.В. Келдыша РАН. Аббревиатура DVM отражает два названия модели: распределенная виртуальная память (Distributed Virtual Memory) и распределенная виртуальная машина (Distributed Virtual Machine). Эти два названия указывают на адаптацию модели DVM как для систем с общей памятью, так и для систем с распределенной памятью. Высокоуровневая модель DVM позволяет не только снизить трудоемкость разработки параллельных программ, но и определяет единую формализованную базу для систем поддержки выполнения, отладки, оценки и прогноза производительности.

В системе DVM не ставилась задача полной автоматизации распараллеливания вычислений и синхронизации работы с общими данными. С помощью высокоуровневых спецификаций программист полностью управляет эффективностью выполнения параллельной программы. Единая модель параллелизма встроена в языки Си и Фортран на базе конструкций, которые “невидимы” для стандартных компиляторов, что позволяет иметь один экземпляр программы для последовательного и параллельного выполнения. Компиляторы с языков C-DVM и Fortran DVM переводят DVM-программу в программу на соответствующем языке (Си или Фортран) с вызовами функций системы поддержки параллельного выполнения.

Язык C-DVM является расширением ANSI-C специальными прагмами вида #pragma dvm `directive`, которые в последовательной программе игнорируются компилятором. DVM-директивы могут быть условно разбиты на три класса:

- Распределение элементов массива между процессорами;
- Распределение витков цикла между процессорами;
- Спецификация параллельно выполняющихся секций программы (параллельных задач) и отображение их на процессоры;

- Организация эффективного доступа к удаленным (расположенным на других процессорах) данным;
- Организация эффективного выполнения редукционных операций - глобальных операций с расположенными на различных процессорах данными (таких, как их суммирование или нахождение их максимального или минимального значения).

Модель параллелизма DVM базируется на специальной форме параллелизма по данным: одна программа – множество потоков данных (SPMD). В этой модели одна и та же программа выполняется на каждом процессоре, но каждый процессор выполняет свое подмножество операторов в соответствии с распределением данных.

В модели DVM пользователь вначале определяет многомерный массив виртуальных процессоров, на секции которого будут распределяться данные и вычисления. При этом секция может варьироваться от полного массива процессоров до отдельного процессора.

На следующем этапе определяются массивы, которые должны быть распределены между процессорами (распределенные данные). Эти массивы специфицируются директивами отображения данных. Остальные переменные (распределяемые по умолчанию) отображаются по одному экземпляру на каждый процессор (размноженные данные). Размноженная переменная должна иметь одно и то же значение на каждом процессоре за исключением переменных в параллельном цикле.

Распределение данных определяет множество локальных или собственных переменных для каждого процессора. Множество собственных переменных определяет правило собственных вычислений: процессор присваивает значения только собственным переменным.

Модель DVM объединяет достоинства модели параллелизма по данным и модели параллелизма по управлению. Параллелизм по данным реализуется распределением витков тесно-гнездового цикла между процессорами массива (или секции массива) процессоров. При этом

каждый виток такого цикла полностью выполняется на одном процессоре. Операторы вне параллельного цикла выполняются по правилу собственных вычислений. Параллелизм задач реализуется распределением данных и вычислений на (разные) секции массива процессоров.

Приведём пример программы на языке C-DVM:

```
#define N 100
#pragma dvm array distribute[block][*]
float (*A)[N + 1];
#pragma dvm array align([i] with A[i][])
float (*X);
.   .   .
/* reverse substitution of Gauss algorithm */
/* own computations outside the loops */
X[N-1] = A[N-1][N] / A[N-1][N]
for(j = N - 2; j >= 0; j--){
  #pragma dvm parallel([k] on A[k][]) remote_access(X[j + 1])
  for(k = 0; k <= j; k++)
      A[k][N] = A[k][N] - A[k][j + 1] * X[j + 1];
  X[j] = A[j][N] / A[j][j];
}
```

1.3. Язык Liszt

Liszt – это специализированный язык (DSL - Domain Specific Language) для написания решателей (solvers) уравнений в частных производных на неструктурных сетках. Язык Liszt является расширением языка Scala. Компилятор с языка Liszt генерирует программу на языке C++, которую затем можно скомпилировать подходящим компилятором. Язык поддерживает распараллеливание вычислений с помощью MPI, pthreads или CUDA. При этом текст на языке Liszt переносим между различными платформами. Приведём в качестве примера задачу теплопроводности:

```
{ // Файл конфигурации liszt.cfg
"runtimes": ["single"],
```

```
"mainclass": "example",
"meshfile": "mesh.msh",  // В формате VTK
"redirecttolog": false,
"numprocs": 1,
"debug": true,
"log": "Progress"
}
import Liszt.Language._
@lisztcode
object HeatTransferExample {
     val rl = 1; val Kq = 0.20f;
     val Position = FieldWithLabel[Vertex,Float3]("position")
     val Temperature = FieldWithConst[Vertex,Float](0.f)
     val Flux = FieldWithConst[Vertex,Float](0.f)
     val Jacobi = FieldWithConst[Vertex,Float](0.f)
     def main() {
          // mesh – глобальная переменная,
          // сетка загружается из файла,
          // указанного в файле конфигурации
          // initialize a single point
          for (v <-vertices(mesh)) {
               if (ID(v) == 1){
                    Temperature(v) = 1000.0f;
               } else {
                    Temperature(v) = 0.f;
               }
          }
          //run Jacobi iteration
          var i = 0;
          while (i < 100) {
               for (e <- edges(mesh)) {
                    val v1 = head(e)
                    val v2 = tail(e)
                    val dP = Position(v2) - Position(v1)
```

```
                    val dT = Temperature(v2) - Temperature(v1)
                    val step = 1.0f/(length(dP))
                    Flux(v1) += dT * step
                    Flux(v2) += dT * step
                    Jacobi(v1) += step
                    Jacobi(v2) += step
               }
               for (p <- vertices(mesh)) {
                    Temperature(p) += 0.1f*Flux(p)/Jacobi(p)
               }
               for (p <- vertices(mesh)) {
                    Flux(p) = 0.f
                    Jacobi(p) = 0.f
               }
               i += 1
          }
          for (p <- vertices(mesh)) {
               Print("Temp ", Temperature(p))
          }
     }
}
```

1.4. Библиотека Blitz++

Библиотека Blitz++ была разработана для выполнения научных расчетов и обеспечивает производительность наравне с Fortran 77/90. Цитата с главной страницы проекта: «Blitz++ is a C++ class library for scientific computing which provides performance on par with Fortran 77/90»). В этой библиотеке для выноса вовне часто используемого кода вводятся шаблоны (stencils). При этом в самой библиотеке уже есть несколько десятков готовых шаблонов для вычисления производных разных степеней, а также таких математических операторов как градиент, дивергенция, лапласиан, якобиан. Пользователь библиотеки может также писать собственные шаблоны. Приведём пример небольшой программы:

```
#include <blitz/array.h>
```

```
BZ_USING_NAMESPACE(blitz)

/*
 * This example program illustrates the "stencil objects",
 * which provide a cleaner notation for finite differencing.
 * The example is solving the 3D acoustic wave propagation
 * problem.
 * Three arrays (P1,P2,P3) contain the pressure field at three
 * consecutive time steps.
 * The c array contains the conduction velocity.
 *
 * Without stencil objects,
 * the stencil would be implemented like this:
 *    Range I(1,N-2), J(1,N-2), K(1,N-2);
 *    P3(I,J,K) = (2-6*c(I,J,K)) * P2(I,J,K)
 *      + c(I,J,K)*
 *          (P2(I-1,J,K) + P2(I+1,J,K) + P2(I,J-1,K)
 *         + P2(I,J+1,K) + P2(I,J,K-1) + P2(I,J,K+1))
 *      - P1(I,J,K);
 */
BZ_DECLARE_STENCIL4(acoustic3D_stencil,P1,P2,P3,c)
  P3 = 2 * P2 + c * Laplacian3D(P2) - P1;
BZ_END_STENCIL

int main(){
    Array<float,3> P1, P2, P3, c;
    const int N = 64;
    allocateArrays(shape(N,N,N), P1, P2, P3, c);
    // Initial conditions: obviously in a real application
    // these wouldn't be zeroed...
    P1 = 0; P2 = 0; P3 = 0; c = 0;
    for(int i=0; i<10; ++i){
        // Apply the stencil object to the arrays
        applyStencil(acoustic3D_stencil(), P1, P2, P3, c);
        // Set [P1,P2,P3]<-[P2,P3,P1]
```

```
        // to set up for the next time step
        cycleArrays(P1,P2,P3);
    }
    return 0;
}
```

1.5. Библиотека PETSc

PETSc (Portable, Extensible Toolkit for Scientific Computation) – это библиотека структур данных и функций для масштабируемого (параллельного) решения научных задач, моделируемых уравнениями в частных производных. Она поддерживает MPI, pthreads, графические ускорители через библиотеки CUDA или OpenCL. Поддерживается также совместное использование GPU и MPI. Библиотеку можно использовать из языков C/C++, Fortran и Python. Библиотека допускает также прямое обращение из MATLAB (см. [40]).

1.6. Пакет OpenFOAM

OpenFOAM (Open Source Field Operation And Manipulation CFD ToolBox) – открытая интегрируемая платформа для численного моделирования задач механики сплошных сред.

В основе кода лежит набор библиотек, предоставляющих инструменты для решения систем дифференциальных уравнений в частных производных, как в пространстве, так и во времени. Рабочим языком кода является C++. В терминах данного языка большинство математических дифференциальных и тензорных операторов в программном коде (до трансляции в исполняемый файл) уравнений может быть представлено в удобочитаемой форме, а метод дискретизации и решения для каждого оператора может быть выбран уже пользователем в процессе расчёта. Таким образом, в коде полностью инкапсулируются и разделяются понятия расчетной сетки (метод дискретизации), дискретизации основных уравнений и методов решения алгебраических

уравнений. Например, уравнение сохранения количества движения для ньютоновской несжимаемой жидкости без действия массовых сил:

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U}\mathbf{U}) - \nabla \cdot (\mu \nabla \mathbf{U}) = -\nabla p$$

может быть представлено в виде:

```
solve
   (
       fvm::ddt(rho, U)
     + fvm::div(rho, U, U)
     - fvm::laplacian(mu, U)
       ==
     - fvc::grad(p)
   );
```

Вместе с кодом поставляется набор программ-решателей, в которых реализованы различные математические модели механики сплошных сред. OpenFOAM может быть запущен на кластере в параллельном режиме и обмениваться между процессами через MPI.

Глава 2. Общее описание подхода

2.1. Назначение сеточно-операторного подхода

Сеточно-операторный подход к программированию предназначен для упрощенной записи вычислений на произвольных сетках. Основными объектами, с которыми позволяет работать библиотека, написанная в соответствие с данным подходом, являются: вычисляемый объект (evaluable object), сеточная функция (grid function), сеточный оператор (grid operator) и сеточный вычислитель (grid evaluator), который получается в результате «применения» сеточного оператора к вычисляемому объекту. К вычисляемым объектам относятся сеточные функции и сеточные вычислители. Для сеточных операторов реализована своя арифметика, 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.

Вычислители можно присваивать плотным сеточным функциям (сеточным функциям, хранящим все свои значения во всех узлах сетки). Запуск вычислений производится именно при таком присваивании, не раньше. До момента присваивания вычислителя плотной сеточной функции цепочка вычислений просто запоминается. Для запоминания цепочки вычислений используется механизм шаблонов выражений (expression templates). При присваивании происходит запуск вычислений для всех требуемых точек сеточной функции в левой части оператора присваивания. Для запуска вычислений используется специальный объект – исполнитель (executor). Он может быть указан пользователем явно или браться по умолчанию. При компиляции для CUDA (компилятор nvcc) и

для обычного процессора по умолчанию подставляются разные исполнители. Для CUDA исполнитель вычисляет все точки параллельно путём вызова ядра (kernel) в графическом ускорителе. В последовательном случае исполнитель обходит все точки последовательно с помощью вложенных циклов.

Применение оператора к вычисляемому объекту реализуется 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);
```

Данная запись является сокращённой формой записи `h=A(f)+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 относительно 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 отложенных вычислений, позволяющая избавиться от промежуточных переменных и за счёт этого сэкономить оперативную память (что становится важным при больших размерах сеток).

Концепция данной библиотеки в чём-то перекликается с концепцией языка Норма (см. [7, 8]). Сеточные функции, с которыми работает библиотека, аналогичны величинам на 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 программистом явно, а в программе на языке Норма задаются только зависимости между величинами, последовательность вычислений компилятор определяет сам.

2.2. Типы обрабатываемых данных

Величины, определённые на трехмерных сетках, это, по сути, физические поля. В физике поля могут быть скалярными (поля температур, плотностей и давлений) или векторными (поля скоростей, ускорений, сил, импульсов и т.п.). Кроме того, все величины в физике имеют размерность. При этом величины разных размерностей нельзя складывать и вычитать, а при умножении и делении получаются величины новых размерностей.

Библиотека *gridmath* поддерживает работу как со скалярными величинами, так и с векторными, причём и те, и другие могут быть как безразмерными, так и иметь размерность. Обычно при решении задач вычислительной математики работают с безразмерными типами данных, такими, как double, float или std::complex. Однако работа с типами данных, имеющими размерность, имеет свои преимущества. Во-первых, программа становится более наглядной, так как размерность той или иной переменной видна уже из её определения. Во-вторых, часть возможных программистских ошибок (таких, как сложение и вычитание величин с разными размерностями) обнаруживаются уже на стадии компиляции. Что касается эффективности вычислений (а это чрезвычайно важный аспект), то информация о размерности той или иной величины (переменной или вычисленного значения) является информацией времени компиляции (метаданными) и не приводит ни к увеличению объёма занимаемой оперативной памяти, ни к замедлению выполнения программы. Возможно, использование размерных величин несколько замедлит время компиляции программы, но несущественно.

Иногда встречаются многомерные величины, в которых различные компоненты имеют разные размерности. Например, уравнение Эйлера в гидродинамике идеальной жидкости в консервативной векторной форме записывается следующим образом:

$$\partial \mathbf{m}/\partial t + \partial \mathbf{f}_x/\partial x + \partial \mathbf{f}_y/\partial y + \partial \mathbf{f}_z/\partial z = 0,$$

где

$\mathbf{m} = (\rho, \rho u, \rho v, \rho w, E)^T$,

$\mathbf{f}_x = (\rho u, p+\rho u^2, \rho uv, \rho uw, u(E+p))^T$,

$\mathbf{f}_y = (\rho v, \rho vu, p+\rho v^2, \rho vw, v(E+p))^T$,

$\mathbf{f}_z = (\rho w, \rho wu, \rho wv, p+\rho w^2, w(E+p))^T$.

Здесь ρ – это плотность жидкости, u, v, w – компоненты скорости, p – давление, E – полная энергия единицы объёма жидкости, $E=\rho e+\frac{1}{2}\rho(u^2+v^2+w^2)$, где e – внутренняя энергия единицы массы жидкости. Величины $\mathbf{f}_x$, $\mathbf{f}_y$, $\mathbf{f}_z$ задают потоки жидкости в направлениях трёх осей.

Каждая из этих векторных величин состоит из пяти компонент, имеющих разные размерности. Уравнение Эйлера в данной форме задаёт законы сохранения массы, трёх компонент момента импульса и энергии. Библиотека *gridmath* позволяет работать и с подобными величинами.

Поддержка различных типов обрабатываемых данных в библиотеке обеспечивается естественным образом за счёт параметризации основных классов библиотеки. Все основные классы библиотеки шаблонные и принимают в качестве параметра шаблона тип обрабатываемого (или хранимого для плотных сеточных функций) значения. Таким типом может быть или любой встроенный в язык простой арифметический тип (float, double, long double), или любой класс (в том числе созданный пользователем), в котором имеются переопределённые основные арифметические операции (например, стандартный библиотечный класс std::complex). Несколько таких классов имеются и в самой библиотеке для работы с размерными или безразмерными скалярными или векторными величинами. Прикладному программисту при необходимости не составит большого труда написать собственный класс, хранящий, например, пять чисел в каждой точке.

Скалярные и векторные величины образуют в библиотеке самостоятельные пространства объектов (сеточных функций и вычисляемых значений), но эти пространства незамкнуты. Можно написать операторы, преобразующие скалярные значения (размерные или

безразмерные) в векторные или наоборот. Например, в библиотеке имеются операторы grad и quantity_grad (градиент), преобразующие скалярную величину в векторную, и операторы div и quantity_div (дивергенция), преобразующие векторную величину в скалярную.

2.3. Поддержка автоматического дифференцирования

Автоматическое дифференцирование – популярное направление в вычислительной математике (см. [44]). Основная идея автоматического дифференцирования состоит в том, чтобы в каждой точке наряду со значением функции хранить и её производную (одну или несколько по разным независимым переменным). При преобразованиях вычисляется не только новое значение функции, но и её производная, причём делается это автоматически исходя из формул при преобразовании. Пусть, например, значение новой функции g вычисляется как $g(x)=\sin(f(x)^2)$, тогда производная равна $g'(x)=\cos(f(x)^2)*2f(x)*f'(x)$. Ясно, что если в некоторой точке известно значение функции f(x) и её производной f'(x), то вычислить значение функции g(x) и её производной g'(x) очень просто. Проблема в том, как это сделать автоматически для всех точек (например, плотной сеточной функции), причём независимо от сложности формулы. Это требует «запоминания» всей цепочки вычислений по формуле и «применения» этой цепочки ко всем точкам. Но «запоминание» цепочки вычислений – это именно то, чем занимается данная библиотека.

Для решения данной задачи в библиотеке имеется класс dual, хранящий два числа: значение функции и значение её производной. Для этого класса имеются переопределённые операторы для всех основных арифметических операций, правильно вычисляющие значение как операции, так и её производной. Например, произведение определено следующим образом:

```
template<typename T>
dual<T> operator*(const dual<T>& a, const dual<T>& b){
     return dual<T>(a.value()*b.value(),
```

```
     a.derivative()*b.value()+a.value()*b.derivative());
}
```

Имеются также переопределённые основные функции из стандартной математической библиотеки. Например, синус определён так:

```
template<typename T>
dual<T> sin(dual<T> a){
return dual<T>(sin(a.value()), a.derivative()*cos(a.value()));
}
```

Теперь достаточно подставить этот класс в качестве параметра шаблона плотной сеточной функции. Вот пример исходного кода:

```
dense_grid_function<dual<double> > f(100, 100, 100);
fill(f); // заполняем функцию f (вместе с производной)
// создаём новую сеточную функцию того же размера.
dense_grid_function<dual<double> > g = f.clone();
g = sin(f*f);
```

Функция g автоматически заполнится правильными значениями как самой функции, так и её производной. По поводу последнего оператора в примере нужно сделать одно замечание: в библиотеке переопределены основные математические функции из стандартной библиотеки, принимающие в качестве параметра вычисляемый объект и возвращающие сеточный вычислитель, вызывающий требуемую функцию при вычислении значений в точках.

В библиотеке также имеется класс dual3, хранящий частные производные функции по всем трём осям (класс хранит 4 значения). Для него также переопределены основные математические операции и основные математические функции из стандартной библиотеки. Использование данного класса в библиотеке аналогично использованию класса dual.

Далее рассмотрим более подробно основные классы и понятия библиотеки.

2.4. Сеточная функция

Под сеточной функцией понимается объект, принимающий определённые значения (одно или несколько) в каждом узле некоторой сетки. В качестве параметра принимается индекс узла сетки. Возможны разные реализации (классы) сеточных функций. В библиотеке 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 при компиляции для 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 суперкомпьютере К-100 объём оперативной памяти основного (host) процессора – 96 Гбайт на узел, а графического ускорителя – всего 2,8 Гбайт на одну графическую плату.

Сеточная функция является вычисляемым объектом (наряду с сеточным вычислителем, см. далее). 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.

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 значения в узле сетки не используются значения из других узлов той же сеточной функции.

Пользователь библиотеки для своих нужд может реализовать собственные сеточные функции. Например, можно реализовать разреженную сеточную функцию, хранящую значения только для определённых комбинаций координат узлов, а для остальных возвращающую ноль.

2.5. Вычисляемый объект

Вычисляемый объект (evaluable object) – 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.

2.6. 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).

2.7. Сеточный вычислитель

Сеточный вычислитель (grid evaluator) 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. Как правило, вычислитель возникает как промежуточный результат вычислений и явно нигде не встречается.

2.8. Исполнители

Как уже говорилось выше, исполнители (executor) используются для запуска вычислений при присваивании вычисляемого объекта плотной сеточной функции. При этом исполнитель можно указать явно или воспользоваться исполнителем по умолчанию. Именно исполнитель определяет, в какой последовательности будут производиться вычисления в разных точках сеточной функции. В библиотеке уже имеется несколько исполнителей, кроме того, пользователь имеет возможность написать свои. Два из уже имеющихся в библиотеке исполнителя используются по умолчанию, т.е. в том случае, если исполнитель не указан явно. Первый – для CUDA, осуществляющий параллельный запуск вычислений для всех точек сетки путём вызова ядра. Второй – для последовательного случая, обходящий точки в одном или нескольких вложенных циклах (в зависимости от версии библиотеки). Хотя этот исполнитель и называется последовательным, он также может распараллелить обход точек за счёт использования технологии OpenMP. Перед циклами стоит прагма

```
#pragma omp parallel for
```

и если при компиляции включить опцию -openmp, то исполнение циклов будет распараллелено.

Кроме того, есть несколько исполнителей, которые пользователь может указать явно. Например, есть исполнители, обрабатывающие точки в соответствие с определённой последовательностью, что позволяет

решать с помощью библиотеки задачи по неявной схеме типа Гаусса-Зейделя, когда часть точек берётся с предыдущего временного слоя, а часть с текущего. При этом все вычисляемые точки делятся на несколько групп и эти группы обрабатываются последовательно. Для точек в самой группе вычисления могут производиться параллельно. При использовании таких исполнителей, как правило, вычисления можно производить «на месте», т.е. в правой части оператора присваивания использовать ту же сеточную функцию, что и в левой части, так как при вычислениях в точке никогда не используются точки из той же самой группы.

Библиотека поддерживает две подобных схемы вычислений. Первая – это шахматная раскраска, при которой все точки делятся на две группы – «чёрные» с чётной суммой индексов и «белые» с нечётной. В начале вычисления производятся параллельно во всех «чёрных» точках, а затем, также параллельно – во всех «белых».

Вторая схема – это схема, при которой, например, значения из точек с меньшими индексами берутся с текущего временного слоя, а с большими – с предыдущего. Подобные схемы решаются методом гиперплоскости фронта вычислений.

2.9. Индексаторы

При работе на графических ускорителях чрезвычайно важным для производительности является то, что CUDA-потоки с последовательными номерами должны обращаться к последовательным ячейкам глобальной памяти. Если это условие не выполняется, то производительность может упасть в несколько раз (от 3 по 5). В случае регулярных сеток это требование обычно выполняется. Данные сеточной функции располагаются в памяти в начале по оси x, затем по оси y и в конце по оси z. Исполнитель по умолчанию для CUDA осуществляет трёхмерный запуск ядра, при этом соседние потоки имеют соседние индексы по оси x.

Ситуация меняется, когда используется нерегулярные сетки или используется гиперплоскость фронта вычислений. При стандартном

расположении данных плотной сеточной функции ни при каком порядке обхода точек гиперплоскости на очередном шаге движения этой гиперплоскости соседние потоки не будут обращаться к соседним ячейкам памяти. Выход был подсказан разработчиками системы DVM (см. [13]), столкнувшимися при реализации своей системы для CUDA с той же проблемой. В системе DVM данные в таком случае переупорядочиваются так, чтобы в соседних ячейках памяти лежали данные из ячеек трёхмерного массива, расположенных на одной горизонтальной диагонали (в горизонтальной плоскости xy). Для параллельного обхода точек гиперплоскости применяется алгоритм, при котором на очередном шаге соседние потоки обрабатывают соседние точки, лежащие на одном горизонтальном отрезке, принадлежащем гиперплоскости, и расположенном на диагонали в плоскости xy.

В данной библиотеке нет необходимости явно переупорядочивать данные сеточной функции. Вместо этого был применён другой подход – в библиотеку были введены новые объекты – индексаторы. Индексаторы в библиотеке не являются самостоятельными объектами, а служат для параметризации плотных сеточных функций. В библиотеке сеточная функция является не классом языка C++, а шаблоном класса. Параметрами шаблона являются тип хранимых данных и класс индексатора. При этом класс индексатора имеет значение по умолчанию, т.е. его можно не указывать. Прототип шаблона класса плотной сеточной функции следующий:

```
template<typename T, class I = default_grid_index>
struct dense_grid_function;
```

В классе индексатора должен быть определён функциональный оператор, принимающий три индекса (i, j, k) и возвращающий линейный индекс данной ячейки. Размеры сеточной функции, которые могут понадобиться индексатору, могут быть указаны или в конструкторе индексатора, или в методе resize:

```
void resize(size_t size_x, size_t size_y, size_t size_z);
```

Индексатор по умолчанию (класс default_grid_index) вычисляет линейный индекс для трёхмерной регулярной сетки следующим образом:

```
size_t operator()(size_t i, size_t j, size_t k) const {
    return (k * m_size_y + j) * m_size_x + i;
}
```

Таким образом, ячейки плотной сеточной функции, у которых индексы j и k совпадают, а индекс i отличается на единицу, будут иметь соседние линейные индексы.

Для работы с гиперплоскостью фронта вычислений в библиотеке имеется индексатор (класс diagonal_grid_index), в котором соседние линейные индексы имеют ячейки сеточной функции, лежащие в одной горизонтальной плоскости (с равными индексами k) на одной горизонтальной диагонали (с равными суммами индексов i+j) и у которых индексы i отличаются на единицу (чем больше i, тем больше линейный индекс).

В библиотеке имеется также «правильный» исполнитель, в котором соседние потоки обращаются к соседним ячейкам сеточной функции на горизонтальной диагонали. При использовании данного исполнителя замена индексатора на диагональный даёт на графических ускорителях ускорение примерно в 4 раза по сравнению с индексатором по умолчанию.

Эти «диагональные» исполнитель и индексатор работают только с гиперплоскостью, проходящей под углом 45° ко всем осям и движущейся от начала координат, т.е. с гиперплоскостью с параметрами (1, 1, 1). Гиперплоскость с такими параметрами встречается чаще всего. Если параметры гиперплоскости оказались другими, то можно использовать более общие универсальные исполнители для гиперплоскости (см, например, [19]) и индексатор по умолчанию. В этом случае всё будет работать правильно, но не столь эффективно.

2.10. Общий вид запуска вычислений

Как уже говорилось выше, вычислитель можно присвоить плотной сеточной функции, при этом будут запущены вычисления для всех точек сеточной функции в левой части оператора присваивания. В этом (простейшем) случае вычисления производятся во всех точках плотной сеточной функции. Однако, бывают случай, когда вычисления нужно провести не во всех точках. Например, оператор Лапласа не определён в граничных точках. Кроме того, в простейшем случае всегда подставляется исполнитель по умолчанию.

Общий вид операции запуска вычислений для трёхмерных регулярных сеток следующий:

```
grid_assign(assignment1, assignment2, …).exec(x0, xn, y0, yn, z0, zn, executor);
```

Для нерегулярных сеток операция запуска имеет следующий вид:

```
grid_assign(assignment1, assignment2, …).exec(i0, in, executor);
```

Здесь assignment1, assignment2 – это т.н. «присваиватели», выражения вида «func <<= expr», где func – это плотная сеточная функция, а expr – произвольный вычисляемый объект. Данное выражение очень похоже на оператор присваивания и отличается только тем, что вместо оператора «=» используется оператор «<<=». Это выражение, в отличие от оператора присваивания, не запускает вычисления, а создаёт объект (присваиватель), который запоминает, какой сеточной функции и что нужно присвоить. Функция grid_assign принимает в качестве параметров от 1 до 10 присваивателей. Если 10 присваивателей мало, то перед включением заголовочных файлов библиотеки можно задать константу MAX_ASSIGNENT_SIZE равной нужному максимальному числу присваивателей.

Смысл задания нескольких присваивателей в том, чтобы осуществить все эти вычисления в едином цикле (или, в случае CUDA, в одном вызове

ядра). Функция grid_assign возвращает объект класса grid_package, в котором запоминаются все присваиватели. Затем в этом объекте вызывается метод exec, который и осуществляет запуск вычислений. В качестве параметров этому методу передаются границы области, в которой нужно произвести вычисления и исполнитель. Исполнитель можно не указывать, в таком случае подставится исполнитель по умолчанию. Использовать эту более сложную форму запуска вычислений нужно в одном из трёх случаев (если выполнено хотя бы одно из трёх условий):

- Нужно выполнить несколько присваиваний в одном вызове;
- Вычисления нужно провести не на всей области, а на её части (например, только во внутренних точках);
- Нужно использовать исполнитель, отличный от исполнителя по умолчанию.

Оператор присваивания на самом деле реализован через вызов функции grid_assign следующим образом (для трёхмерных регулярных сеток):

```
template<typename T, class I = default_grid_index>
struct dense_grid_function {
     …
     template<class EO>
     void operator=( const grid_evaluable_object<EO, typename
EO::proxy_type> &eobj){
          grid_assign(*this <<= eobj).exec(
               0, size_x(), 0, size_y(), 0, size_z());
     }
};
```

Функции grid_assign передаётся единственный присваиватель (присваивающий данной сеточной функции указанный вычисляемый объект), а в качестве границ вычисляемой области в метод exec передаются полные границы сеточной функции. Исполнитель не указывается (используется исполнитель по умолчанию).

Приведём пример явного использования функции grid_assign из программы, реализующей метод верхней релаксации (SOR). В ней на каждом шаге итерации нужно вычислить невязку (разницу между старым и новым значениями). Основная сеточная функция – A, невязка хранится в сеточной функции S. Основной оператор итерации следующий:

```
hyperplane_executor executor;
grid_assign(
     S <<= A,            // Запоминаем старое значение
     A <<= sor(A),       // Вычисляем новое значение
     S <<= abs(A - S)    // Вычисляем невязку
).exec(1, size_x - 1, 1, size_y - 1, 1, size_z - 1, executor);
```

В данном примере функции grid_assign передаются три присваивателя, вычисления проводятся только во внутренних точках и используется исполнитель, обходящий область вычислений по гиперплоскостям.

2.11. Примеры

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 вычислителя A(f) 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)); // применение оператора A к вычислителю B(f) и
сложение скаляра 3.0 и вычислителя (A(B(f)).
h=B(f-A(g));   // вычитание вычислителя A(g) из сеточной
функции f и применение оператора B к полученному вычислителю
(f-A(g)).
```

Глава 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 большинством современных компиляторов с языка C++.

Библиотека *gridmath*, помимо стандартной библиотеки языка C++ (STL) использует библиотеку *boost* (см. [45]), а при компиляции для CUDA – ещё и библиотеку thrust из состава CUDA SDK (см. [2], [46]). Библиотека *thrust* содержит аналоги большинства необходимых компонентов как из STL, так и из *boost*.

3.2. Шаблонный полиморфизм

Как уже говорилось выше, библиотека шаблонная. С другой стороны, она объектно-ориентированная и классы выстроены в определённое иерархическое дерево классов. Одно из свойств объектно-ориентированного программирования – полиморфизм, когда, имея ссылку на объект базового класса, можно вызвать метод из класса-наследника. В классическом объектно-ориентированном программировании на C++ полиморфизм реализуется с помощью механизма виртуальных функций. Однако, этот механизм является чрезвычайно затратным по времени исполнения. Если тело функции небольшое, то может оказаться, что временные затраты на вызов виртуальной функции сравнимы с временем выполнения самой функции. Гораздо эффективнее работают «inline» функции, когда код тела функции подставляется вместо вызова, но для виртуальных функций такое невозможно, т.к. компилятору тело функции в конечном классе неизвестно. С появлением в языке C++ шаблонов ситуация радикальным образом изменилась. Хотя, возможно, первоначальной целью введения шаблонов в язык было обеспечить работу

с типизированными коллекциями и другие относительно простые примеры использования, выяснилось, что шаблоны являются мощным механизмом для реализации таких вещей, как, например, метавычисления (вычисления времени компиляции) и нового типа полиморфизма – шаблонного (см. [15, 46, 47, 48]). Шаблонный полиморфизм реализуется с помощью шаблона проектирования (design pattern), известного как CRTP - Curiously recurring template pattern (см. [49]). Идея шаблона CRTP следующая: в базовый класс в качестве шаблонного параметра передаётся имя конечного класса. При этом, если имеется ссылка на объект базового класса, то её всегда можно привести к ссылке на конечный класс и вызвать тот или иной метод конечного класса непосредственно. При этом компилятор знает всё про этот конечный класс и может вместо вызова метода подставить его тело, что и требуется. Покажем на простом примере, как это реализуется. Пусть Base – это базовый класс, а Derived – конечный класс и мы хотим, имея ссылку на объект базового класса, вызвать метод в конечном классе. Сделать это можно, например, так:

```
template<class D>
struct Base {
     D& self(){ return static_cast<D&>(*this); }
};
struct Derived : Base<Derived> {
     void m(){ … }  // Метод, который хотим вызвать
};
// Полиморфная функция
template<class D>
void f(Base<D>& obj){ // Ссылка на объект базового класса
     obj.self().m(); // Вызов метода конечного класса
}
```

В данной библиотеке все классы пронаследованы от единого базового класса math_object и передают ему два шаблонных параметра – себя и класс заместитель (см. ниже). Вот полный исходный текст этого класса:

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

```
struct math_object {
     typedef T final_type;
     typedef _Proxy proxy_type;

     final_type& self(){
          return static_cast<final_type&>(*this);
     }
     const final_type& self() const {
          return static_cast<const final_type&>(*this);
     }
     proxy_type get_proxy() const { return self(); }
};
```

Если имеется ссылка на объект класса math_object, то её можно с помощью метода self преобразовать в ссылку на конечный класс и вызвать требуемый метод в этом конечном классе.

3.3. Метавычисления

В библиотеке gridmath активно используются метавычисления. Метавычисления – это вычисления, производимые во время компиляции. В языке C++ во время компиляции можно вычислять типы и целочисленные константы. Для метавычислений можно писать т.н. метафункции. Параметрами метафункций также могут быть типы и целочисленные константы. Подробно о метавычислениях в C++ можно прочитать в [15].

Приведём в качестве примера две метафункции. Первая (is_numeric) принимает произвольный тип и возвращает в переменной value булевскую константу (true или false) в зависимости от того, является ли переданный тип числовым:

```
template<bool B>    // Вспомогательный класс
struct bool_ { static const bool value = B; };
template<typename T>
struct is_numeric : bool_<false>{}; // Общий случай, по
умолчанию тип не числовой
```

```
};
template<> struct is_numeric<int> : bool_<true>{};
template<> struct is_numeric<long> : bool_<true>{};
template<> struct is_numeric<short> : bool_<true>{};
template<> struct is_numeric<float> : bool_<true>{};
template<> struct is_numeric<double> : bool_<true>{};
template<> struct is_numeric<long double> : bool_<true>{};
template<typename T> struct is_numeric<std::complex<T> > :
bool_<true>{};
```

Теперь к этой метафункции можно обращаться как из других метафункций, так и из исполняемого кода. Например:

```
bool for_double = is_numeric<double>::value;     // true
bool for_string = is_numeric<std::string>::value; // false
```

Другой пример:

```
template<typename T>
class my_class {
  BOOST_STATIC_ASSERT(is_numeric<T>::value);
  … // остальной код класса
};
```

В этом примере попытка передать в качестве параметра класса my_class нечисловой тип приведёт к ошибке при компиляции.

Вторая метафункция (get_value_type) принимает тип коллекции или итератора и возвращает в типе type тип значений, которые хранятся в коллекции. В эту метафункцию в качестве параметра можно также передавать указатель на тип, в этом случае метафункция вернёт этот тип.

```
template<typename T>
struct get_value_type {  // Общий случай, берём из коллекции
     typedef typename T::value_type type;
};
template<typename T>
struct get_value_type<T*> {   // Специализация для указателей
     typedef T type;
};
template<typename T>
```

```
struct get_value_type<const T*> {  // Специализация для const
указателей
     typedef T type;
};
```

Пример использования этой метафункции из исполняемого кода:

```
template<typename I>
typename get_value_type<I>::type element_at(I it)
{ return *it; }
```

Данная функция принимает итератор или указатель и возвращает значение, на которое он указывает. Тип этого значения определяется с помощью метафункции get_value_type.

Библиотека gridmath активно использует оба типа метафункций. То, как это используется для работы с размерными величинами, описано в следующем параграфе.

3.4. Размерные величины

Использование размерных величин повышает читаемость программы, т.к. уже из размерности переменных видно, что же они хранят (например, скорость, плотность или давление). При этом использование размерных величин вместо безразмерных не меняет объём памяти, занимаемой переменными и не влияет на скорость работы программы, т.к. вся информация о размерности хранится в метаданных. Библиотека gridmath поддерживает работу с размерными величинами. Для работы со скалярными размерными величинами (плотность, давление) предназначен класс quantity, а с векторными (скорость, ускорение, сила) – vector_quantity. Оба класса принимают два шаблонных параметра – тип хранимого значения (например, float или double) и собственно размерность:

```
template<typename T, class Dimensions>
struct quantity
{
… // Реализация класса
private:
```

```
    data_type m_value;
};
```

В качестве второго параметра следует передавать специальный тип quantity_dim, хранящий в метаданных (параметрах шаблона) семь целых чисел, соответствующий семи основным физическим величинам: масса, длина, время, сила тока, температура, сила света и количество вещества. Единицы физических величин (например, граммы или килограммы для массы, метры или сантиметры для длины) в библиотеке не определены, но подразумеваются одинаковыми:

```
template<int N1,int N2,int N3,int N4,int N5,int N6,int N7>
struct quantity_dim {
    static const int value1 = N1;
    static const int value2 = N2;
    static const int value3 = N3;
    static const int value4 = N4;
    static const int value5 = N5;
    static const int value6 = N6;
    static const int value7 = N7;
};
```

Имеются две метафункции для сложения (add_dimensions) и вычитания (sub_dimensions) размерностей. Первая используется при умножении размерных величин, а вторая – при делении:

```
template<class D1, class D2> struct add_dimensions;

template<
    int N1, int N2, int N3, int N4, int N5, int N6, int N7,
    int M1, int M2, int M3, int M4, int M5, int M6, int M7
>
struct add_dimensions
<
    quantity_dim<N1, N2, N3, N4, N5, N6, N7>,
    quantity_dim<M1, M2, M3, M4, M5, M6, M7>
>
{
```

```
    typedef quantity_dim<N1 + M1, N2 + M2, N3 + M3, N4 + M4,
N5 + M5, N6 + M6, N7 + M7> type;
};

template<class D1, class D2> struct sub_dimensions;

template<
    int N1, int N2, int N3, int N4, int N5, int N6, int N7,
    int M1, int M2, int M3, int M4, int M5, int M6, int M7
>
struct sub_dimensions
<
    quantity_dim<N1, N2, N3, N4, N5, N6, N7>,
    quantity_dim<M1, M2, M3, M4, M5, M6, M7>
>
{
    typedef quantity_dim<N1 - M1, N2 - M2, N3 - M3, N4 - M4,
N5 - M5, N6 - M6, N7 - M7> type;
};
```

Обе метафункции на вход принимают две размерности, а возвращают в типе type результирующую размерность. Вот как эти метафункции используются при умножении и делении размерных величин:

```
template<typename T, class D1, class D2>
quantity<T, typename add_dimensions<D1, D2>::type>
operator*(const quantity<T, D1>& x, const quantity<T, D2>& y)
{
    typedef typename add_dimensions<D1, D2>::type dim;
    return quantity<T, dim>(x.value() * y.value());
}

template<typename T, class D1, class D2>
quantity<T, typename sub_dimensions<D1, D2>::type>
operator/(const quantity<T, D1>& x, const quantity<T, D2>& y)
{
    typedef typename sub_dimensions<D1, D2>::type dim;
```

```
    return quantity<T, dim>(x.value() / y.value());
}
```

В классе quantity определены операции сложения и вычитания с величинами той же размерности, умножение и деление на безразмерные скаляры и на величины произвольных размерностей. При умножении и делении на размерные величины получаются величины новых размерностей.

Класс vector_quantity отличается от класса quantity, во-первых, тем, что он хранит три значения вместо одного (value_x, value_y, value_z), и, во-вторых, тем, что для него определён другой (более широкий) набор операций. Операция умножения двух векторных величин определена как векторное умножение, операция деления двух векторных величин не определена, определены операции умножения и деления векторной величины на скалярную размерную величину. Также определена операция скалярного умножения двух векторных величин, возвращающая скалярную размерную величину (через оператор &) и операция покомпонентного произведения двух векторных величин (через оператор ^).

Чтобы задать размерность, нужно указать степени каждой из семи основных величин. Например, размерность ускорения задаётся так:

```
typedef quantity_dim<0,1,-2,0,0,0,0> acceleration;
```

Для задания размерной величины нужно указать тип хранимого значения и размерность. Например, переменная, хранящая ускорение, задаётся так:

```
vector_quantity<double, acceleration> a;
```

3.5. Объекты-заместители

В библиотеке реализована концепция отложенных вычислений: до тех пор, пока не будет исполнен оператор присваивания вычисляемого объекта 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 что значит «запоминаются»? Ссылки на объекты сохранять нельзя, также, как нельзя 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, так как временный объект сразу после использования может быть удалён и затёрт. Вторая причина состоит в том, что при компиляции для CUDA 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, копировать же ссылки (или указатели) на объекты из памяти host-процессора в память CUDA не имеет смысла, т.к. это разные адресные пространства. Фактически в случае плотной сеточной функции нужно скопировать указатель на данные.

Для того чтобы решить указанную проблему, в библиотеке вводится понятие «заместителя» объекта (proxy). В качестве замещаемого объекта может быть практически любой объект библиотеки: 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 программистом, 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 *dense_grid_function*. Схематично это можно показать на следующем примере:

```
template<typename T>
struct dense_grid_function_proxy; // Предварительное
объявление
temlate<typename T>
struct dense_grid_function :
math_object<dense_grid_function<T>,
dense_grid_function_proxy<T> >, std::vector<T>
{
dense_grid_function(size_t size_x, size_t size_y, size_t
size_z) : m_size_x(size_x), m_size_y(size_y),
m_size_z(size_z){}
size_t size_x() const { return m_size_x; }
size_t size_y() const { return m_size_y; }
size_t size_z() const { return m_size_z; }
     … // Другие методы
private:
     size_t m_size_x, m_size_y, m_size_z;
};
temlate<typename T>
struct dense_grid_function_proxy
{
     dense_grid_function_proxy(dense_grid_function<T>& func) :
m_size_x(func.size_x()), m_size_y(func.size_y()),
m_size_z(func.size_z()), m_data(func.data_pointer()){}
     size_t size_x() const { return m_size_x; }
     size_t size_y() const { return m_size_y; }
```

```
    size_t size_z() const { return m_size_z; }
    T operator()(size_t i, size_t j, size_t k) const {
        return m_data[(k * m_size_y + j) * m_size_x + i];
    }
private:
    size_t m_size_x, m_size_y, m_size_z;
    T *m_data; // Указатель на данные вместо массива
};
```

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().

3.6. Контекст исполнения

В «обычной» программе (без использования данной библиотеки) при одновременном вычислении нескольких величин для ускорения счёта для каждой точки могут однократно вычисляться дополнительные вспомогательные переменные, которые затем многократно используются. Поясним это на примере. Пусть одновременно вычисляются две величины (плотных сеточных функции), a=sin(c*c) и b=cos(c*c), где c – некая третья сеточная функция. Тогда цикл в «обычной» программе мог бы выглядеть так:

```
for(size_t i = 0; i < n1; ++i){
for(size_t j = 0; j < n2; ++j){
    for(size_t k = 0; k < n3; ++k){
      double temp = sqr(c(i, j, k));
      a(i, j, k) = sin(temp);
      b(i, j, k) = cos(temp);
    }
  }
}
```

Ясно, что мы экономим вычисления за счёт того, что значение квадрата величины c в каждой точке вычисляется однократно. Если мы просто перепишем эту программу, то получим примерно такой код:

```
grid_assign(
     a <<= sin(sqr(c)),
     b <<= cos(sqr(c))
).exec(0, n1, 0, n2, 0, n3);
```

Этот код существенно лучше предыдущего тем, что он может быть исполнен параллельно для разных точек (например, при использовании CUDA или OpenMP), но, тем не менее, имеет тот недостаток, что одна и та же величина в нём вычисляется дважды. Для решения этой проблемы в библиотеке введено понятие контекста исполнения. Контекст исполнения – это структура, определяемая пользователем (программистом), которая создаётся для каждого набора индексов (каждой точки), и члены которой могут затем многократно использоваться. Для создания контекста используется вспомогательный объект – построитель контекста (context builder). Покажем на примере, как это работает.

```
struct my_context; // forward declaration
// Класс построителя контекста
struct my_context_builder :
grid_context_builder<my_context_builder>
{
     // В типе context_type содержится класс контекста
     typedef my_context context_type;
     // В конструктор построителя контекста передаются
     // всё, что нужно для вычисления переменных,
     // в данном случае передаётся сеточная функция c.
     my_context_builder(const dense_grid_function<double>& c) :
          c_proxy(c.get_proxy())
     const typename dense_grid_fuction<double>::proxy_type
c_proxy;  // Заместитель объекта (плотной сеточной функции)
};
// Класс контекста
```

```
struct my_context : grid_context<my_context>
{
// В конструктор контекста передаются координаты точки
// и построитель контекста. На основании этой информации
// вычисляются все вспомогательные переменные
     my_context(size_t i, size_t j, size_t k,
const my_context_builder& cb)
{
          double temp = cb.c_proxy(i, j, k);
sqrc = temp * temp;
}
     double sqrc;
};
// Ещё один класс для извлечения вспомогательной переменной
struct context_sqrc : grid_evaluable_object<context_sqrc>
{
     value_type operator()(size_t i, size_t j, size_t k,
const grid_context<my_context>& context) const {
          return context.self().sqrc;
     }
} _sqrc;  // И переменная этого класса
// Оператор присваивания
grid_assign(
     a <<= sin(_sqrc),
     b <<= cos(_sqrc)
).exec(0, n1, 0, n2, 0, n3, my_context_builder(c));
```

3.7. Исполнение на CUDA

Компилятор nvcc полноценно поддерживает работу с шаблонами функций и классов. В частности, глобальные функции (помеченные ключевым словом __global__), которые исполняются на графическом устройстве, а вызываются из основного (host) процессора, также могут быть шаблонными. Исходя из этого факта возникло решение: для любой задачи, которую нужно исполнить на устройстве, создать объект, содержащий в себе все необходимые данные для решения конкретной

задачи и имеющий очень простой внешний интерфейс (например, функциональный метод, в который передаются глобальные индексы нитей по всем трём направлениям. Решение было подсмотрено в библиотеке thrust. Там такой объект называется «замыканием» (closure). Итак, нужно написать одну глобальную шаблонную функцию с одним параметром, в которую передать в качестве параметра шаблона название класса замыкания. Параметром функции будет замыкание. Эта глобальная функция вычисляет глобальные индексы нитей и вызывает функциональный метод в замыкании, передав туда в качестве параметров эти глобальные индексы. Всю основную работу будет исполнять этот метод в замыкании. Исходный текст этой глобальной функции следующий:

```
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);
}
#define BLOCK1_SIZE 512
template<class Closure>
void launch_closure_3d(Closure& closure, size_t size_x, size_t
size_y, size_t size_z)
{
     size_t block_size_x = BLOCK1_SIZE;
     while(block_size_x / 2 >= size_x && block_size_x > 8)
          block_size_x /= 2;
     size_t block_size_y = BLOCK1_SIZE / block_size_x;
     while(block_size_y / 2 >= size_y)
          block_size_y /= 2;
```

```
    size_t block_size_z = BLOCK1_SIZE / block_size_x /
block_size_y;
    while(block_size_z / 2 >= size_z)
        block_size_z /= 2;
    dim3 dimGrid(
        (size_x + block_size_x - 1) / block_size_x,
        (size_y + block_size_y - 1) / block_size_y,
        (size_z + block_size_z - 1) / block_size_z);
    dim3 dimBlock(block_size_x, block_size_y, block_size_z);
    launch_3d<<<dimGrid, dimBlock>>>(closure);
}
```

На класс замыкания накладываются определённые требования (концепция замыкания). Так как объект этого класса передаётся по значению (копируется) из основного процессора на графическую плату, он должен содержать только переменные или простых типов (в том числе указателей), или классов, имеющих копирующий конструктор по умолчанию. В нём также должен быть функциональный метод с тремя параметрами – глобальными индексами нитей.

3.8. Обмен данными между основным процессором и графическим ускорителем

При использовании графического ускорителя очень важно иметь возможность пересылать данные из памяти основного процессора в память графического устройства и наоборот. Это может быть нужно, например, для загрузки исходных данных из файла и сохранения результатов счёта в файл, а также в процессе счёта для обмена данными между процессами по MPI. В библиотеке данные хранятся в векторах. Основной класс для хранения данных – это класс math_vector. Этот класс хранит данные в том устройстве, на котором ведётся счёт. Если это последовательная версия библиотеки, то данные хранятся в памяти основного процессора, а если это CUDA, то в памяти графического ускорителя. Есть также класс host_vector, который хранит данные всегда в памяти основного процессора. Оба класса находятся в пространстве имён kiam::math. Переменные этих двух классов

можно присваивать друг другу, при этом данные будут копироваться из памяти основного процессора в память графического ускорителя или наоборот.

Что касается реализации, то в последовательном варианте оба класса пронаследованы от класса std::vector из стандартной библиотеки языка C++ (то есть фактически math_vector и host_vector – это одно и то же). В случае CUDA класс math_vector пронаследован от класса thrust::device_vector, а класс host_vector – от класса thrust::host_vector. Копирование данных фактически осуществляется библиотекой thrust.

В частности, плотная сеточная функция хранит данные в переменной типа math_vector. В переменную того же типа сохраняется срез плотной сеточной функции (см. Приложение 1), которую затем можно скопировать в переменную типа host_vector и затем передать данные по MPI.

3.9. Использование пула нитей.

На современных компьютерах, как серверных, так и обычных рабочих станциях, стоящих у нас на рабочих столах или дома, часто стоят современные многоядерные процессоры, способные запускать по несколько нитей (от 2 до 8 на процессор) параллельно. А на серверах ещё и процессоров может быть несколько на одном узле. Если использовать полностью возможности таких процессов, то программа может считаться существенно быстрее.

Есть разные способы распараллеливания работы программы, например, если есть компилятор, поддерживающий OpenMP, то можно в последовательной программе перед циклом поставить соответствующую прагму и всё заработает. Кстати, в последовательной версии библиотеки эта прагма перед тремя вложенными циклами стоит. Это значит, что если указать опцию компилятору –openmp, то программа автоматически распараллелится. Именно так можно запустить параллельную программу на процессоре Intel Xeon Phi в «native» режиме.

Однако, не все компиляторы поддерживают эту опцию. Чтобы и в этом случае можно было распараллелить работу последовательной программы, библиотека использует возможности распараллеливания, предоставляемые библиотекой boost. Используется библиотека boost-threads, которая должна быть скомпилирована.

Для использования пула нитей предназначен специальный исполнитель – grid_threadpool_executor. В качестве параметра конструктора этого класса передаётся количество нитей. Чтобы использовать все имеющиеся в системе процессоры, нужно передать результат функции boost::thread::hardware_concurrency(). Этот исполнитель нужно передать в метод exec() объекта grid_package, возвращаемого функцией grid_assign (см. 2.8).

Глава 4. Приложения и тесты, разработанные с использованием библиотеки gridmath

В общем случае перенос программы на графические ускорители CUDA является непростой задачей, подразумевающей, в том числе, знакомство с архитектурой CUDA и методами программирования на ней. Библиотека gridmath позволяет «автоматически» переносить на CUDA написанные с её использованием программы. Программа может быть отлажена в последовательном режиме на «обычном» процессоре, а затем перенесена на CUDA практически без изменений путём перекомпиляции компилятором nvcc.

Библиотека была задумана при решении задачи по переносу программы, реализующей многосеточный метод (multigrid) на графические ускорители NVidia CUDA в рамках проекта для Института проблем безопасного развития атомной энергетики РАН. С использованием библиотеки задача была успешно решена.

Была переписана с использованием библиотеки последовательная версия теста MG из набора тестов NAS Parallel Benchmarks (см. [51]),

которая затем была перенесена на NVIDIA CUDA, а также задача о решении системы квазигидродинамических уравнений, любезно предоставленная автору Александром Давыдовым (см. [52, 53, 54]).

Была реализована версия данной библиотеки для нерегулярных сеток. В качестве такой нерегулярной сетки была взята локально-адаптивная сетка, в которой в первоначальную трёхмерную регулярную сетку помещается объект некоторой формы (например, шар или цилиндр) и на границах этого объекта сетка измельчается.

Было проведено сравнительное исследование на производительность различных методов решения задачи теплопроводности. Результаты опубликованы в статье [24].

4.1. Многосеточный метод

Сотрудниками ИПМ им. Келдыша РАН Жуковым Виктором Тимофеевичем, Феодоритовой Ольгой Борисовной и Новиковой Натальей Дмитриевной был разработан параллельный многосеточный метод для разностных эллиптических уравнений (см. [1]) и программа, реализующая этот метод. В рамках проекта для Института проблем безопасного развития атомной энергетики РАН (ИБРАЭ РАН) была поставлена задача о переносе этой программы на графические ускорители NVidia CUDA.

Исходная задача считалась на кластере, при этом вся трёхмерная сетка была распределена между узлами, обмен граничными условиями вёлся с помощью MPI, а на каждом узле счёт вёлся последовательно. Теперь предстояло счёт на каждом узле также распараллелить на графических ускорителях. Это потребовало весьма существенного переписывания алгоритмической части программы, т.к. в исходном коде цикл в операторе присваивания вёлся по сеточной функции не в левой части оператора, а в правой. Это не давало возможности распараллелить существующий цикл. Были написаны три сеточных оператора: интерполяции, проектирования и сглаживания. Особенностью данной задачи было то, что хотя сетка и прямоугольная, но не равномерная, шаг сетки в каждом направлении для

каждого индекса свой (хотя от индексов в других направлениях он не зависит). Оператор сглаживания реализовывал семиточечную схему со своим коэффициентом в каждом узле сетки и с переменным шагом сетки в каждом направлении.

Ещё одной особенностью задачи было то, что нужно было обмениваться по MPI данными, лежащими в памяти GPU. Это значит, что эти данные нужно было вначале копировать в память основной (host) системы, пересылать по MPI, а затем копировать обратно в память GPU. Для копирования данных использовалась возможность получать срезы (грани) плотных сеточных функций.

Также следует обратить внимание, что при работе на GPU библиотека хранит данные в памяти GPU, а её не так уж и много по сравнению с объёмом памяти в основной системе. Так, на кластере К-100 на основной системе имеется 96 Гбайт памяти, что в пересчёте на один процесс (при полной загрузке узла – 12 процессов) даёт по 8 Гбайт на процесс, в то время как на каждой графической плате имеется только 2,8 Гбайт памяти. Это накладывает ограничение на размер сетки. Максимальный размер сетки на один процесс при работе на GPU составляет 256 точек в каждом направлении.

Кроме CUDA версии программы есть также версия, работающая на обычном процессоре. CUDA версия программы в максимальном варианте запускалась на 128 процессах по 3 процесса на узел (по количеству плат CUDA на узле), процессы обменивались данными по MPI. При этом было задействовано 43 узла из имеющихся 64, общий размер сетки составлял 1024x1024x2048 (на каждый процесс – 256x256x256).

Для обычного процессора программа показала производительность, аналогичную исходной версии программы, как в однопроцессном варианте, так и в многопроцессном с обменом данными между процессами по MPI. CUDA версия программы в однопроцессном варианте ускорялась примерно в 8 раз по сравнению с версией для обычного процессора, в

многопроцессном варианте это ускорение составляло от 4 до 5 раз. Это замедление ускорения (с 8 до 4 раз) объясняется тем, что по сети можно пересылать только данные из памяти host-процессора. Соответственно, для варианта с CUDA дополнительно требуется пересылать данные из памяти графического ускорителя в память host-процессора и обратно. Один из путей возможного ускорения программы видится в том, чтобы вычисления на графическом ускорителе и пересылку данных по сети производить параллельно.

Результаты расчётов сведены в следующей таблице:

Таблица 1. Сравнение эффективности разных реализаций многосеточного метода.

Процессорная сетка	Расчётная сетка	t_{grid}, сек.	t_{CUDA}, сек.	Ускорение, t_{grid}/t_{CUDA}
1x1x1	256x256x256	30		
2x1x1	512x256x256	42		
2x2x1	512x512x256	40	28,1	
2x2x2	512x512x512	37/45	23	
4x2x2	1024x512x512	55/67	25,8	
4x4x2	1024x1024x512	64		
4x4x4	1024x1024x1024	83		

4.2. Тест MG из набора тестов NAS Parallel Benchmarks

Этот тест является реализацией стандартного варианта многосеточного метода для решения трехмерного уравнения Пуассона с периодическими краевыми условиями на равномерной декартовой сетке. Используются: линейный оператор интерполирования, точечный метод Якоби в качестве сглаживающей процедуры и набор вложенных сеток с коэффициентом 2 прореживания узлов по каждому направлению. Операторы на каждой сетке строятся по простейшим формулам 27-точечной разностной аппроксимации.

Для задачи определены классы задачи, определяющие размер сетки и число сглаживающих итераций. Есть классы S, W, A, B, C, D, E. класс S – самый маленький, а класс E – самый большой. Для класса B размер сетки – 256 точек в каждом направлении, а число итераций равно 20. Для класса C размер сетки – 512 точек в каждом направлении. При таком количестве точек требуемый размер оперативной памяти около 3,5 Гбайт и на K-100 на GPU памяти уже не хватает. Для класса D размер сетки – 1024 точек в каждом направлении. кроме того, для класса D делается 50 итераций. Приведём результаты сравнения:

Таблица 2. Сравнение эффективности разных реализаций теста MG.

Класс задачи	Объём памяти	$t_{Fortran}$, сек.	$t_{Fortran\ OMP}$, сек.	$t_{gridmath}$, сек.	$t_{gridmath\ OMP}$, сек.	t_{CUDA}, сек.
B	0,45 ГБ	6,53	1,56	26,8	3,71	1,1
C	3,57 ГБ	45,76	13,21	71,77	31,27	
D	28,4 ГБ	1009,9	263,59	1454,48	567,01	

Замедление последовательной версии на библиотеке gridmath по сравнению с исходной фортрановской версией можно объяснить тем, что эта исходная версия тщательно оптимизировались специалистами в течение многих лет. Операторная версия не может использовать методы оптимизации из исходной версии, но зато позволяет путём простой перекомпиляции перенести программу на CUDA и получить существенное ускорение.

4.3. Система квазигазодинамических уравнений

Известная задача о каверне с подвижной крышкой решается с помощью системы квазигазодинамическимх уравнений по явной схеме.

Решение этой задачи интересно в двух отношениях. Во-первых, в ней использовались не только скалярные трёхмерные поля (такие, как поля плотностей, давлений и температур), но и векторные (такие, как поля скоростей, ускорений и сил). Вторая особенность – использование размерных величин, т.е. для, например, давления, использовалась не переменная типа double, а переменная специального типа, хранящая в

метаданных (данные времени компиляции) ещё и размерность. Работа с такими размерными величинами (как скалярными, так и векторными) также поддерживается библиотекой gridmath. Использование размерных типов данных позволяет решить две задачи. Во-первых, исходный текст программы становится более наглядным, т.к. из размерности той или иной переменной сразу становится ясно, что в ней хранится (например, скорость или давление). Во-вторых, часть возможных ошибок в программе, связанных с неправильным использованием переменных (присваивать, складывать и вычитать можно только величины одной размерности) обнаруживаются уже на этапе компиляции. Использование размерных типов данных никак не увеличивает объём памяти, занимаемый переменными, и не замедляет скорость выполнения программы.

Библиотека gridmath позволяет писать операторы, принимающие скалярные величины, а возвращающие векторные (например, оператор градиента) или наоборот (оператор дивергенции), причём величины могут быть как безразмерными, так и размерными. Для работы с такими операторами в библиотеке активно используется метапрограммирование языка C++ (см., например, [14, 15]). В библиотеке есть готовые операторы градиента, дивергенции, ротора и Лапласа, причём как для безразмерных, так и для размерных величин.

Покажем на одном примере (метод calcNewVars) использование библиотеки gridmath. В оригинальной версии программы текст был примерно следующий (основной цикл):

```
for(i = 1; i <= nx; i++){
     for(j = 1; j <= ny; j++){
     for(k = 1; k <= nz; k++){
     dts = dt / (hx[i] * hy[j] * hz[k]);
     ro1 = ro[GDF(i,j,k)] + (Frox[FX(i,j-1,k-1)]-Frox[FX(i-1,j-
1,k-1)]+Froy[FY(i-1,j,k-1)]-Froy[FY(i-1,j-1,k-1)]+Froz[FZ(i-
1,j-1,k)]-Froz[FZ(i-1,j-1,k-1)])*dts;
     ro[GDF(i,j,k)] = ro1;
```

```
     ux[GDF(i,j,k)] = (ro[GDF(i,j,k)]*ux[GDF(i,j,k)] +
(Fuxx[FX(i,j-1,k-1)]-Fuxx[FX(i-1,j-1,k-1)]+Fuxy[FY(i-1,j,k-
1)]-Fuxy[FY(i-1,j-1,k-1)]+Fuxz[FZ(i-1,j-1,k)]-Fuxz[FZ(i-1,j-
1,k-1)])*dts)/ro1;
     uy[GDF(i,j,k)] = (ro[GDF(i,j,k)]*uy[GDF(i,j,k)] +
(Fuyx[FX(i,j-1,k-1)]-Fuyx[FX(i-1,j-1,k-1)]+Fuyy[FY(i-1,j,k-
1)]-Fuyy[FY(i-1,j-1,k-1)]+Fuyz[FZ(i-1,j-1,k)]-Fuyz[FZ(i-1,j-
1,k-1)])*dts)/ro1;
     uz[GDF(i,j,k)] = (ro[GDF(i,j,k)]*uz[GDF(i,j,k)] +
(Fuzx[FX(i,j-1,k-1)]-Fuzx[FX(i-1,j-1,k-1)]+Fuzy[FY(i-1,j,k-
1)]-Fuzy[FY(i-1,j-1,k-1)]+Fuzz[FZ(i-1,j-1,k)]-Fuzz[FZ(i-1,j-
1,k-1)])*dts)/ro1;
     E[GDF(i,j,k)] = E[GDF(i,j,k)] + (FEx[FX(i,j-1,k-1)]-
FEx[FX(i-1,j-1,k-1)]+FEy[FY(i-1,j,k-1)]-FEy[FY(i-1,j-1,k-
1)]+FEz[FZ(i-1,j-1,k)]-FEz[FZ(i-1,j-1,k-1)])*dts;
     p[GDF(i,j,k)] = (E[GDF(i,j,k)]-
ro[GDF(i,j,k)]*0.5*(ux[GDF(i,j,k)]*ux[GDF(i,j,k)] +
uy[GDF(i,j,k)]*uy[GDF(i,j,k)] +
uz[GDF(i,j,k)]*uz[GDF(i,j,k)]))*gm1;
} } }
```

Текст довольно длинный и сложный, но если в нём разобраться, то видно, что вычисления ведутся по одной схеме. Поэтому был введён оператор new_vars_op:

```
template<typename T>
struct new_vars_op : grid_operator<new_vars_op<T> >{
     typedef quantity<data_type, length> length_type;
     template<class D> struct get_dim {
          typedef typename sub_dimensions<D, volume>::type
type;
     };
     template<class Q> struct get_value_type;

     template<class D>
     struct get_value_type<vector_quantity<data_type, D> >{
```

```
      typedef quantity<data_type, typename get_dim<D>::type>
type;
     };

     new_vars_op(const length_type hx_[], const length_type
hy_[], const length_type hz_[]) : hx(hx_), hy(hy_), hz(hz_){}

     template<class EOP, class GC> __DEVICE
     typename get_value_type<typename EOP::value_type>::type
     operator()(size_t i, size_t j, size_t k, const EOP& eop,
const grid_context<GC>& context) const {
     return (
       eop(i, j - 1, k - 1, context).qvalue_x()
     - eop(i - 1, j - 1, k - 1, context).qvalue_x()
     + eop(i - 1, j,     k - 1, context).qvalue_y()
     - eop(i - 1, j - 1, k - 1, context).qvalue_y()
     + eop(i - 1, j - 1, k    , context).qvalue_z()
     - eop(i - 1, j - 1, k - 1, context).qvalue_z()
     ) / (hx[i] * hy[j] * hz[k]);
     }
     const length_type *hx, *hy, *hz;
};
```

С использованием этого оператора текст функции существенно упрощается и становится более понятным:

```
double gm = params[2], gm1 = gm - 1;
new_vars_op<double> op(hx, hy, hz);
grid_assign(
u <<= (ro * u + dt * get_make_vector_evaluator(op(Fux),
    op(Fuy), op(Fuz))) / (ro + dt * op(Fro)),
     ro <<= ro + dt * op(Fro),
     E <<= E + dt * op(FE),
     p <<= (E - ro * .5 * (u & u)) * gm1
).exec(1, nx + 1, 1, ny + 1, 1, nz + 1);
```

Размер исходного текста сильно сократился за счёт многократного использования оператора new_vars_op. Видно также использование скалярного произведения векторов (u & u). Текст стал гораздо понятнее.

4.4. Локально-адаптивная сетка

Данная библиотека может быть достаточно легко перенесена на произвольные нерегулярные сетки. Основная идея такого переноса следующая: сетка должна быть задана таким образом, чтобы все ячейки сетки, в которых нужно произвести вычисления, были каким-либо образом упорядочены, т.е. чтобы по порядковому номеру узла сетки можно было бы узнать всю информацию об этой ячейке.

В качестве первого примера реализации библиотеки для нерегулярных сеток была выбрана локально-адаптивная сетка. Строится она следующим образом: берётся регулярная трёхмерная сетка и в неё помещается некоторый объект (например, шар или цилиндр). Затем рёбра, грани и ячейки, которые лежат на границах этого объекта, делятся пополам. Такое разбиение сетки повторяется несколько раз (например, 5). В полученной сетке имеются объекты разных уровней измельчения и эта сетка существенно нерегулярна. Тем не менее все ячейки такой сетки – прямоугольные параллелепипеды, и это весьма существенно облегчает работу с такой сеткой. В частности, легко определяется объём ячеек. По разные стороны каждой грани могут быть ячейки с разных уровней, соответственно, разные ячейки могут иметь разное количество соседних ячеек.

При реализации библиотеки для локально-адаптивной сетки были определены два разных объекта – сетка и сеточная функция. Сетка хранит информацию обо всех ячейках (из размеры, список соседних ячеек и пр.), а сеточная функция – значения некоторой величины в каждой ячейке сетки. Для регулярных сеток такое разбиение было не нужно, т.к. структура сетки в этом случае настолько проста, что всё необходимое проще каждый раз сосчитать, чем где-то хранить.

На такой локально-адаптивной сетке была решена задача диффузии. При этом на границе объекта, помещённого в сетку, у коэффициента диффузии был скачок – внутри объекта этот коэффициент был на два порядка больше, чем снаружи.

Так же, как и для регулярных сеток, программа была легко перенесена на графические ускорители и решалась на кластере К-100 на нескольких узлах с обменом по MPI. Декомпозиция сетки на несколько узлов была сделана Головченко Евдокией Николаевной с помощью разработанного ею метода (см. [55]).

Опыт переноса библиотеки на локально-адаптивные сетки позволяет предположить, что перенос библиотеки на произвольные неструктурные сетки – вполне реальная задача, хотя, возможно, и непростая.

4.5. Задача теплопроводности

В качестве тестовой рассматривается трёхмерная задача теплопроводности. Находится стационарное решение уравнения теплопроводности

$$\frac{\partial u}{\partial t} = \Delta(u) + f(x, y, z).$$

Находится приближённое решение для заданного точного решения

$$u_0(x, y, z) = \sin(lx) \times \sin(my) \times \sin(nz).$$

В этом уравнении l, m, n – константы. В этом случае

$$f(x, y, z) = (l^2 + m^2 + n^2) \times \sin(lx) \times \sin(my) \times \sin(nz).$$

Точное решение используется для задания граничных условий (типа Дирихле). Задача решается в единичном кубе, каждое ребро которого делится на N частей, т.е. шаг по направлению h=1/N. Шаг по временит полагается равным $\tau = \frac{h^2}{12}$

Константы полагаются равными l=3, m=6, n=1. Число N в разных запусках полагалось равным 64, 128 или 256. Задача решается с точностью ε, которая меняется от 0.1 до 1e-6.

Задача решается разными методами и на разных процессорах, точнее на процессоре Intel Core i7 (последовательная версия программ) и на графическом ускорителе NVIDIA GeForce GTX 670 (параллельная CUDA версия). Задача решалась явным методом (метод Якоби) и двумя неявными методами (Гаусса-Зейделя) – шахматной раскраски и гиперплоскости фронта вычислений. Каждый из неявных методов также решался в двух вариантах – простом и последовательной верхней релаксации (successive over relaxation - SOR). Кроме того, для сравнения были реализованы двухслойный метод простой итерации, двухслойный и трёхслойный Чебышевские методы и многосеточный метод. Все методы были реализованы в последовательном и параллельном (для CUDA) вариантах.

Общее описание итерационных методов для линейных систем, включая метод Якоби, Гаусса-Зейделя, последовательной верхней релаксации и других, можно прочитать в [56], глава 6. Метод Гаусса-Зейделя, сформулированный в самом общем виде (как в [57]), подразумевает, что все точки области делятся на несколько подобластей, выстроенных в определённом порядке. В первой подобласти вычисления производятся на основе значений в точках на предыдущей итерации (как в методе Якоби), а вычисления в каждой следующей подобласти – на основе только что вычисленных значений в предыдущих подобластях на текущей итерации. При этом вычисления внутри каждой из подобластей можно производить параллельно. Делить все точки области на подобласти можно по-разному, это приводит к разным вариантам метода Гаусса-Зейделя, в частности, к методу шахматной раскраски (две подобласти) и методу гиперплоскости (много подобластей).

Идея метода шахматной раскраски проста. Вся область разбивается на чередующиеся между собой «чёрные» (с чётной суммой индексов) и «белые» (с нечётной суммой индексов) ячейки (трёхмерная шахматная доска). «Чёрные» ячейки считаются первыми и используют значения с

предыдущей итерации. Затем считаются «белые» ячейки, которые используют только что посчитанные «чёрные» ячейки с текущей итерации.

В методе гиперплоскости фронта вычислений эта гиперплоскость последовательно передвигается от некоторой начальной вершины под некоторыми углами к осям, пока не дойдёт до противоположной вершины. Начальная вершина и углы к осям определяются параметрами гиперплоскости. В данной работе гиперплоскость передвигается от вершины с координатами (1, 1, 1) под углом 45° ко всем осям, пока не дойдёт до вершины с координатами (N, N, N). Были реализованы несколько разных алгоритмов обхода точек гиперплоскости с тем, чтобы сравнить их эффективность.

Идея метода последовательной верхней релаксации (обозначается через SOR(ω), где ω – *параметр релаксации*) состоит в том, чтобы улучшить цикл Гаусса-Зейделя подходящим взвешенным усреднением значений с предыдущей и текущей итераций, при этом значения с текущей итерации берутся с коэффициентом ω, а с предыдущей – 1-ω. Значение параметра релаксации ω=1 соответствует методу Гаусса-Зейделя, при ω<1 говорят о *нижней релаксации*, при ω>1 – о *верхней релаксации*. В методе последовательной верхней релаксации параметр релаксации лежит в интервале от 1 до 2. Подробно об этом методе можно прочитать, например, в [55], раздел 6.5.3. Оптимальное значение параметра релаксации подбиралось экспериментально при малой точности (ε=1e-2). При этом для метода шахматной раскраски оптимальное значение оказалось равным ω=1,18, а для метода гиперплоскости ω=1,83.

Метод простой итерации, двухслойный и трёхслойный Чебышевские методы описаны в [56]. Многосеточный метод описан в [1]. Все эти методы явные, т.е. значения во всех точках вычисляются на основе значений с предыдущей итерации. В этом они похожи на явный метод Якоби. Так же, как и в методе Якоби, значения во всех точках области в одной итерации можно вычислять одновременно. Это существенно

упрощает (и, соответственно, ускоряет) алгоритм одного цикла и сильно ускоряет параллельную реализацию.

Общая идея двухслойных методов состоит в построении двухслойной итерационной последовательности вида:

$$u_{j+1} = u_j + \tau_{j+1} \cdot \left(\Delta u_j + f\right), j = 0,1, \ldots$$

с произвольным начальным приближением u_0. Здесь $\tau\{_j\}$ – последовательность итерационных параметров. В методе простой итерации все τ_j берутся одинаковыми и равными некоторому оптимальному значению. Это оптимальное значение зависит от минимального и максимального значений собственных чисел линейного оператора в решаемом уравнении. Для уравнения Пуассона (линейный оператор – оператор Лапласа) $\lambda_{min}=24$, $\lambda_{max}= 12/h^2$, где h – шаг сетки (см. [1]). При этом $\tau_j \equiv \tau_0=2/(\lambda_{min}+\lambda_{max})$. Используются также следующие обозначения (см. [56]):

$$\xi = \frac{\lambda_{min}}{\lambda_{max}}, \rho_0 = \frac{1-\xi}{1+\xi}, \rho_1 = \frac{1-\sqrt{\xi}}{1+\sqrt{\xi}} \ .$$

В двухслойном Чебышевском методе итерационные параметры вычисляются на основе многочленов Чебышева. Число Чебышевских итераций определяется по формуле (см. [50], [1]):

$$n_0(\varepsilon) = \frac{\ln\left(1/\varepsilon + \sqrt{1/\varepsilon^2 - 1}\right)}{\ln\left(1/\rho_0\right)} \approx \frac{\ln\left(\frac{2}{\varepsilon}\right)}{\ln\left(1/\rho_0\right)}.$$

Для устойчивости процесса вычислений итерационные параметры должны быть определённым образом упорядочены, алгоритм их упорядочения взят из [57], глава VI, §2, п.5, стр. 280-283. Особенностью этого метода является то, что количество итераций определяется заранее, исходя из размера сетки и требуемой точности вычислений. При этом для сходимости нужно выполнить все эти итерации. Во всех других методах, (кроме Чебышевских и многосеточного), в конце каждой итерации

вычисляется норма невязки и проверяется, стала ли она в нужное число раз меньше начальной невязки. Если нет, то выполняется следующая итерация. Таким образом, общее количество итераций становится известным только в конце вычислений.

В трёхслойном Чебышевском методе строится трёхслойная итерационная схема с произвольным начальным приближением u_0. Значение u_1 вычисляется как в методе простой итерации:

$$u_1 = u_0 + \tau \cdot (\Delta u_0 + f).$$

Следующие итерации вычисляются по формуле:

$$u_{j+1} = (1 - \alpha_{j+1}) \cdot u_{j-1} + \alpha_{j+1} \cdot \left(u_j + \tau \cdot (\Delta u_j + f)\right), j = 1,2, \ldots$$

Здесь итерационный параметр τ такой же, как в методе простой итерации. Из последней формулы видно, что при $\alpha_j \equiv 1$ трёхслойный метод превращается в метод простой итерации. Реальные итерационные параметры $\{\alpha_j\}$ вычисляются с использованием полиномов Чебышева первого рода и обеспечивают оптимальную сходимость метода. Оптимальные значения итерационных параметров вычисляются из рекуррентной формулы $\alpha_{j+1}=4/(4-\rho_0^2\alpha_j)$, $\alpha_1=2$ (см. [57]). Число итераций, требуемых для достижения заданной точности ε, определяется как

$$n_0(\varepsilon) = \frac{\ln 0{,}5\,\varepsilon}{\ln \rho_1}.$$

Возможен также стационарный трёхслойный метод, в котором все α_j равны между собой и равны оптимальному значению, равному $\alpha_j \equiv \alpha = 1+\rho_1^2$.

В многосеточном методе используется 5 сеточных уровней. Самый грубый уровень решается с помощью двухслойного Чебышевского метода. На других уровнях перед сборкой (переходом на грубый уровень) и после проектирования (перехода на подробный уровень) осуществляется сглаживание тем же двухслойным Чебышевским методом, но с малым числом итераций (уменьшение нормы невязки в два раза), как правило, это 2 шага.

Интересно посмотреть, с одной стороны, на то, за сколько шагов сойдётся с заданной точностью решение для разных алгоритмов, и, с другой стороны, измерить быстродействие разных алгоритмов. Для параллельной реализации того или иного алгоритма интересно посмотреть, насколько он ускорится по сравнению с последовательной версией того же алгоритма. Оценить сложность алгоритма можно по времени выполнения одной итерации. Простые алгоритмы (в которых время одной итерации мало) даже при большем числе итераций могут показать лучшее время.

4.6. Разрывный метод Галёркина

Для реализации разрывного метода Галёркина (см. [60-64]) операторный метод был перенесён на трёхмерные нерегулярные сетки. У этого метода относительно сложная математика, которая хорошо формулируется в терминах операторов (лимитирования, вычисления объёмных интегралов и потоков через грани), и на нём операторный метод может показать все свои преимущества. Как известно, разрывный метод Галёркина характеризуется высоким порядком точности решения, что влечет за собой и высокую вычислительную сложность, что является большим недостатком метода Галёркина. В данной работе активно применяется механизм метапрограммирования языке C++, позволяющий вынести часть вычислений на стадию компиляции.

Для метода Галёркина было реализовано несколько операторов, в том числе операторы лимитирования, объёмного интегрирования и вычисления потоков через грани. Вычисления проводились на кластере К-100 с использованием MPI, вычисления на одном узле распараллеливались с помощью OpenMP или CUDA.

Проиллюстрируем реализацию на примере оператора объёмного интегрирования. В разрывном методе Галёркина для каждого тетраэдра требуется вычислить пять объёмных интегралов:

$$I_{kj}^{1ev} = \int\limits_{T_k} \left[(\rho u) \frac{\partial \varphi_j}{\partial x} + (\rho v) \frac{\partial \varphi_j}{\partial y} + (\rho w) \frac{\partial \varphi_j}{\partial z} \right] dV$$

$$I_{kj}^{2ev} = \int\limits_{T_k} \left[(\rho u^2 + p) \frac{\partial \varphi_j}{\partial x} + (\rho u v) \frac{\partial \varphi_j}{\partial y} + (\rho u w) \frac{\partial \varphi_j}{\partial z} \right] dV$$

$$I_{kj}^{3ev} = \int\limits_{T_k} \left[(\rho v u) \frac{\partial \varphi_j}{\partial x} + (\rho v^2 + p) \frac{\partial \varphi_j}{\partial y} + (\rho v w) \frac{\partial \varphi_j}{\partial z} \right] dV$$

$$I_{kj}^{4ev} = \int\limits_{T_k} \left[(\rho w u) \frac{\partial \varphi_j}{\partial x} + (\rho w v) \frac{\partial \varphi_j}{\partial y} + (\rho w^2 + p) \frac{\partial \varphi_j}{\partial z} \right] dV$$

$$I_{kj}^{5ev} = \int\limits_{T_k} \left[(E + p) u \frac{\partial \varphi_j}{\partial x} + (E + p) v \frac{\partial \varphi_j}{\partial y} + (E + p) w \frac{\partial \varphi_j}{\partial z} \right] dV$$

Все эти интегралы подходят под следующий единый шаблон:

$$I_{kj}^{nev} = \int\limits_{T_k} \left[f_x^n \frac{\partial \varphi_j}{\partial x} + f_y^n \frac{\partial \varphi_j}{\partial y} + f_z^n \frac{\partial \varphi_j}{\partial z} \right] dV,$$

где $f^n = f^n(\rho, u, v, w, p, E)$, а φ_j – базисные функции.

Интеграл по тетраэдру вычисляется методом Гаусса, для чего нужно найти значение выражения в квадратных скобках в Гауссовых точках. На входе в процедуру имеются коэффициенты разложения по базисным функциям величин ρ, ρu, ρv, ρw, E.

В начале по этим коэффициентам находятся сами величины в Гауссовых точках, затем вычисляются значения переменных $u=\rho u/\rho$, $v=\rho v/\rho$, $w=\rho w/\rho$. После этого вычисляется значение переменной e: $e=E/\rho-(u^2+v^2+w^2)/2$. Давление p вычисляется по уравнению состояния $p=p(\rho, e)$.

Поставим целью реализовать объёмное интегрирование один раз, а выражения для вычисления функций f^n передавать как параметры. По этим выражениям в процессе интегрирования будут вычисляться конкретные значения. Механизм, позволяющий передавать выражения для последующих вычислений по ним, называется «шаблоны выражений»

(expression templates). Создадим набор классов, реализующих следующую простую грамматику:

«выражение»::=«элементарное выражение» | (*«выражение»*) |

«выражение»+«выражение» | *«выражение»-«выражение»* |

«выражение»«выражение»* | *«выражение»/«выражение»;*

«элементарное выражение»::=ρ|ρu|ρv|ρw|E|u|v|w|e|p;

При вычислении значений каждое выражение будет получать в качестве параметра объект, хранящий все необходимые для вычисления величины:

```
struct integrate_data_type {
     data_type rho, rho_u, rho_v, rho_w, E, u, v, w, e, p;
};
```

Для реализации выражений использовался механизм шаблонов выражений. В результате вычисление объёмных интегралов удалось записать в весьма элегантной форме, в которой интегрируемые выражения задаются явно:

```
R = A * (join_polynom_operator<grid_type>(ugrid)(
 volume_integral<integrate_tetra_t>
  (ugrid, eos, _rho_u, _rho_v, _rho_w)(U),
 volume_integral<integrate_tetra_t>
  (ugrid, eos, _rho_u * _u + _p, _rho_u * _v, _rho_u * _w)(U),
 volume_integral<integrate_tetra_t>
  (ugrid, eos, _rho_v * _u, _rho_v * _v + _p, _rho_v * _w)(U),
 volume_integral<integrate_tetra_t>
  (ugrid, eos, _rho_w * _u, _rho_w * _v, _rho_w * _w + _p)(U),
 volume_integral<integrate_tetra_t>
  (ugrid, eos, (_E + _p) * _u, (_E + _p) * _v, (_E + _p) * _w)(U)) -
hflow_op(U));
```

Результаты работы опубликованы в виде препринтов (см. [21, 22]).

Приведём сравнительную таблицу эффективности использования одного узла на кластере К-100 при решении некоторой задачи на сетке среднего размера (1,5 млн. ячеек) методом Галёркина операторным

методом. Делалось 50 шагов по времени, вычисления производились на одном процессоре (последовательная программа), с использованием OpenMP и с использованием CUDA, причём использовалась как одна плата, так и все три имеющиеся на узле. В последнем случае делалась декомпозиция сетки на три части и запускалось три процесса. Кроме того, проверялся также вариант с использованием всех имеющихся на узле вычислительных мощностей. Сетка делилась на 4 части и запускалось 4 процесса (на одном узле). Три процесса считали на трёх платах CUDA, а четвёртый – на CPU с использованием OpenMP. Замерялось время непосредственно счёта (без обменов и сохранения результатов).

	CPU1	OpenMP	CUDA1	CUDA3	MIX
Время, сек.	854.39	92.01	117.37	39.08	

Видно, что использование OpenMP ускоряет счёт на 12 ядрах примерно в 9.3 раза. Одна CUDA более, чем в 7 раз (7.2) быстрее, чем последовательная программа и примерно в на 27% медленнее, чем OpenMP, но все три CUDA примерно вдвое быстрее, чем OpenMP. MIX не даёт существенного ускорения по сравнению с CUDA3, т.к. CPU становится узким местом и всё тормозит. OpenMP не даёт такого ускорения, т.к. приходится разделять ядра с процессами, работающими с CUDA.

4.7. Метод ENO

Метод ENO был реализован для одномерного случая. Данная реализация ещё раз показала широкую применимость операторного метода для различных задач и типов сеток.

Заключение

Основные результаты работы:

1. Предложен новый подход к разработке ПО для решения задач математической физики с использованием сеточных операторов, аналогичных математическим операторам.

2. Подход реализован в виде программных библиотек для разных типов сеток, что обеспечивает наглядную математическую нотацию, улучшающую структурированность исходных текстов программ и упрощает их перенос на параллельные архитектуры.
3. С помощью данного подхода был решён ряд задач на различных типах сеток: регулярных трёхмерных, трёхмерных локально-адаптивных, нерегулярных тетраэдральных.

В целом данный подход упрощает написание программ, решающих задачи на различных сетках. Текст программ становится более структурированным и понятным. Благодаря этому уменьшается количество возможных ошибок в программах и время на разработку и отладку. При этом программы работают достаточно эффективно и не требуют дополнительной памяти для промежуточных вычислений.

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

Литература

1. Жуков В.Т., Новикова Н.Д., Феодоритова О.Б. Параллельный многосеточный метод для разностных эллиптических уравнений. Часть I. Основные элементы алгоритма // Препринты ИПМ им. М.В.Келдыша. 2012. № 30. 32 с. URL: http://library.keldysh.ru/preprint.asp?id=2012-30
2. NVidia CUDA. URL: http://www.nvidia.com/object/cuda_home_new.html
3. Intel Xeon Phi. URL: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html
4. TOP500 Supercomputer Sites. URL: http://www.top500.org

5. Суперкомпьютер "Ломоносов"
   URL: http://parallel.ru/cluster/lomonosov.html
6. Гибридный вычислительный кластер К-100
   URL: http://www.kiam.ru/MVS/resourses/k100.html
7. Язык программирования НОРМА. URL: http://keldysh.ru/pages/norma/
8. А.Н. Андрианов, К.Н. Ефимкин, И.Б. Задыхайло, Н.В. Поддерюгина. Язык Норма.
   // Препринты ИПМ им. М.В. Келдыша. — 1985. — № 165. 34 с.
9. Задыхайло И.Б., Пименов С.П. Семантика языка Норма.
   // Препринты ИПМ им. М.В. Келдыша АН СССР, 1986 г., № 139
10. Андрианов А.Н. Организация циклических вычислений в языке Норма
    // Препринты ИПМ им. М.В. Келдыша АН СССР, 1986 г., № 171
11. Андрианов А.Н. Синтез параллельных и векторных программ по непроцедурной записи на языке Норма. Диссертация на соискание ученой степени кандидата физико-математических наук. М., 1990 г.
12. Андрианов А.Н. Система Норма. Разработка, реализация и использование для решения задач математической физики на параллельных ЭВМ. Диссертация на соискание ученой степени доктора физико-математических наук. М., 2001 г.
13. DVM система. URL: http://www.keldysh.ru/dvm/
14. Template metaprogramming.
    URL: http://en.wikipedia.org/wiki/Template_metaprogramming
15. David Abrahams, Aleksey Gurtovoy. C++ Template Metaprogramming. Addison-Wesley. — 2004.
16. OpenMP. URL: http://openmp.org
17. 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=2013-09

18. Жуков В.Т., Краснов М.М., Новикова Н.Д., Феодоритова О.Б. Параллельный многосеточный метод: сравнение эффективности на современных вычислительных архитектурах
// Препринты ИПМ им. М.В.Келдыша. 2014. № 31. 22 с.
URL: http://library.keldysh.ru/preprint.asp?id=2014-31
19. Краснов М.М. Оптимальный параллельный алгоритм обхода точек гиперплоскости фронта вычислений и его сравнение с другими итерационными методами решения сеточных уравнений
// Препринты ИПМ им. М.В.Келдыша. 2015. № 20. 20 с.
URL: http://library.keldysh.ru/preprint.asp?id=2015-20
20. Жуков В.Т., Краснов М.М., Новикова Н.Д., Феодоритова О.Б. Численное решение параболических уравнений на локально-адаптивных сетках чебышевским методом
// Препринты ИПМ им. М.В. Келдыша. 2015. № 87. 26 с.
URL: http://library.keldysh.ru/preprint.asp?id=2015-87
21. Краснов М.М., Ладонкина М.Е., Тишкин В.Ф. Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Использование операторного метода программирования.
// Препринты ИПМ им. М.В.Келдыша. 2016. № 23. 27 с.
URL: http://library.keldysh.ru/preprint.asp?id=2016-23
22. Краснов М.М., Ладонкина М.Е. Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++.
// Препринты ИПМ им. М.В.Келдыша. 2016. № 24. 23 с.
URL: http://library.keldysh.ru/preprint.asp?id=2016-24
23. Краснов М.М. Операторная библиотека для решения трёхмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA.
// Математическое моделирование, 2015, т. 27, с. № 3, 109-120.

24. Жуков В.Т., Краснов М.М., Новикова Н.Д., Феодоритова О.Б. Сравнение эффективности многосеточного метода на современных вычислительных архитектурах.
// Программирование, 2015, № 1, с. 21-31.
25. Краснов М.М. Параллельный алгоритм вычисления точек гиперплоскости фронта вычислений. // Журнал вычислительной математики и математической физики, 2015, том 55, № 1, с. 145-152.
26. Краснов М.М., Кучугов П.А., Ладонкина М.Е., Тишкин В.Ф. Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Использование операторного метода программирования.
// Математическое моделирование, 2017 г., т. 29, № 2, с. 3-22 (принято в печать).
27. Краснов М.М., Ладонкина М.Е. Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++. // Программирование, 2017 г., № 3 (принято в печать).
28. Краснов М.М. Операторная библиотека для решения трёхмерных сеточных задач математический физики с использованием графических плат с архитектурой CUDA. // Национальный суперкомпьютерный форум (НСКФ-2013), Переславль-Залесский, ИПС им. А.К. Айламазяна РАН, 26-29 ноября 2013 г. URL: http://2013.nscf.ru/TesisAll/Section%206/04_1000_KrasnovMM_S6.pdf
29. Краснов М.М. Несколько примеров переноса программ на графические ускорители CUDA с использованием библиотеки gridmath.
// Тезисы докладов XX Всероссийской конференции и Молодёжной школы-конференции «Теоретические основы и конструирование численных алгоритмов решения задач математической физики», посвященная памяти К.И. Бабенко (Дюрсо, 15-20 сентября 2014). – М: Институт прикладной математики им. М.В. Келдыша, 2014, с. 72-73.

30. Краснов М.М. Операторная библиотека для решения задач математической физики на трёхмерных локально-адаптивных сетках с использованием графических ускорителей CUDA. // Международная конференция «Суперкомпьютерные дни в России», 28 - 29 сентября 2015 г., г. Москва.
31. Краснов М.М., Кучугов П.А., Ладонкина М.Е., Тишкин В.Ф. Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Использование операторного метода программирования. // XIV Международный семинар «Математические модели и моделирование в лазерно-плазменных процессах и передовых научных технологиях», 4 - 9 июля 2016 г., г. Москва. URL: http://lppm3.ru/files/histofprog/LPpM3-2016-1-Programme.pdf
32. Краснов М.М., Кучугов П.А., Ладонкина М.Е., Тишкин В.Ф. Разрывный метод Галёркина на трёхмерных тетраэдральных сетках. Использование операторного метода программирования. // VII всероссийская научная молодежная школа-семинар «Математическое моделирование, численные методы и комплексы программ» имени Е. В. Воскресенского с международным участием, 12-15 июля 2016 г., г. Саранск.
33. Краснов М.М., Кучугов П.А., Ладонкина М.Е., Тишкин В.Ф. Метод Галеркина с разрывными базисными функциями в многомерном случае. // XXI Всероссийская конференция «Теоретические основы и конструирование численных алгоритмов для решения задач математической физики», посвященная памяти К.И.Бабенко, 5-11 сентября 2016г. г. Новороссийск, пос. Абрау-Дюрсо (Тезисы докладов)
34. Krasnov M.M., Kuchugov P.A., Ladonkina M.E., Tishkin V.F. Application of discontinuous Galerkin method for modeling of three-dimensional problems of flow past solid bodies. // XV International Seminar «Mathematical models & modeling in laser plasma processes & advanced science technologies»,

26-30 September, 2016, Montenegro, Petrovac.
URL: http://lppm3.ru/files/histofprog/lppm3-2016-2-programme.pdf
35. Краснов М.М., Кучугов П.А., Ладонкина М.Е., Тишкин В.Ф. Обобщение метода Годунова, использующее кусочно-полиномиальные аппроксимации. // XVI Международная конференция «Супервычисления и математическое моделирование», г. Саров, 3-7 октября 2016.
36. Liszt DSL. URL: http://graphics.stanford.edu/hackliszt/
37. OpenACC Toolkit. URL: https://developer.nvidia.com/openacc
38. The Blitz++ library. Official website. URL: http://blitz.sourceforge.net/
39. Blitz++ Library on SourceForge.net. URL: http://sourceforge.net/projects/blitz/
40. MATLAB – The Language of Technical Computing. URL: http://www.mathworks.com/products/matlab/
41. SIMIT Language. URL: http://simit-lang.org/
42. PETSc. URL: http://www.mcs.anl.gov/petsc/
43. OpenFOAM. URL: http://www.openfoam.com
44. Automatic Differentiation. URL: http://www.autodiff.org/
45. Boost C++ Libraries. URL: http://www.boost.org/
46. Thrust – Parallel Algorithms Library. URL: http://thrust.github.com/
47. Бьерн Страуструп. Язык программирования C++. Специальное издание/ Пер. с англ. – М.; СПб., «БИНОМ» – «Невский Диалект», 2002 г.
48. Бьерн Страуструп. Дизайн и эволюция языка C++. СПб., «Питер», 2006 г.
49. Björn Karlsson. Beyond the C++ Standard Library: An Introduction to Boost. Addison Wesley Professional, 2005
50. Curiously recurring template pattern URL: http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

51. NAS Parallel Benchmarks
URL: http://www.nas.nasa.gov/publications/npb.html
52. Елизарова Т.Г. Квазигазодинамические уравнения и методы расчета вязких течений. М.: Науч. мир, 2007. 352 с.
53. Davydov, Alexander A.; Shilnikov, Evgeny V. Numerical Simulation of the Low Compressible Viscous Gas Flows on GPU-based Hybrid Supercomputers. International Conference on Parallel Computing - ParCo2013, 10-13 September 2013, Munich, Germany.
54. B.N. Chetverushkin, E.V. Shilnikov, A.A. Davydov. Numerical Simulation of Continuous Media Problems on Hybrid Computer Systems", in P. Ivanyi, B.H.V. Topping, (Editors), "Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering", Civil-Comp Press, Stirlingshire, UK, Paper 25, 2012. doi:10.4203/ccp.95.25, 14 p
55. Е.Н. Головченко. Параллельный пакет декомпозиции больших сеток // Математическое моделирование. 2011. Т. 23. № 10. с. 3-18
56. Деммель Дж. Вычислительная линейная алгебра. Теория и приложения. М.: «Мир», 2001 г., 430 с.
57. Самарский А.А., Николаев Е.С. Методы решения сеточных уравнений. М.: «Наука», Главная редакция физико-математической литературы, 1978 г., 592 с.
58. Leslie Lamport. The Parallel Execution of DO Loops. Communications of the ACM, February 1974, Volume 17, Number 2.
59. Лацис А.О. Параллельная обработка данных. М., «Академия», 2010 г.
60. Bernardo Cockburn, An Introduction to the Discontinuous Galerkin Method for Convection - Dominated Problems, Advanced Numerical Approximation of Nonlinear Hyperbolic Equations (Lecture Notes in Mathematics), 1998, V. 1697, pp. 151-268.

61. Галанин М.П., Савенков Е.Б., Токарева С.А. Применение разрывного метода Галеркина для численного решения квазилинейного уравнения переноса, 2005, Препринт 105, ИПМ им. М.В. Келдыша РАН, Москва.
62. Ладонкина М.Е., Неклюдова О.А.,Тишкин В.Ф. Исследование влияния лимитера на порядок точности решения разрывным методом Галеркина // Препринты ИПМ им. М.В.Келдыша. 2012. № 34. 31 с. URL: http://library.keldysh.ru/preprint.asp?id=2012-34
63. Ладонкина М.Е., Неклюдова О.А., Тишкин В.Ф. Исследование влияния лимитера на порядок точности решения разрывным методом Галеркина // Математическое моделирование 2012 г., т. 24. № 12.
64. Ладонкина М.Е., Неклюдова О.А.,Тишкин В.Ф. Лимитер повышенного порядка точности для разрывного метода Галеркина на треугольных сетках // Препринты ИПМ им. М.В.Келдыша. 2013. № 53. 26 с. URL: http://library.keldysh.ru/preprint.asp?id=2013-53
65. Самарский А.А., Тишкин В.Ф., Фаворский А.П., Шашков М.Ю. О представлении разностных схем математический физики в операторной форме. – Докл. АН СССР, 1981, т. 258, № 5, с. 1092-1096.
66. Самарский А.А., Тишкин В.Ф., Фаворский А.П., Шашков М.Ю. Операторные разностные схемы. – Дифференциальные уравнения, 1981, т. 17, № 7, с. 1317-1327.

Приложение 1. Описание программного интерфейса сеточно-операторной библиотеки

Общая информация.

Сеточно-операторная библиотека gridmath является шаблонной библиотекой классов, это, в частности, означает, что она поставляется в виде набора заголовочных файлов (с расширениями .h и .hpp) и отдельной компиляции не требует. Заголовочные файлы лежат в дереве каталогов, начинающемся с каталога kiam/math. В этом каталоге есть подкаталоги grid, matrix, ugrid, linear и др. для разных типов сеток. Как правило, тот или

иной класс расположен в файле с именем <имя_класса.hpp>. Файлов и классов довольно много, поэтому, чтобы упростить жизнь программиста, добавлены заголовочные файлы, которые в себе включают всё, что нужно. К примеру, для использования версии библиотеки для трёхмерных регулярных сеток достаточно включить файл grid_all.h:

```
#include <kiam/math/grid/grid_all.h>
```

В этих файлах включаются не только все нужные файлы из самой библиотеки, но и все сторонние файлы, требуемые для работы библиотеки, включая нужные стандартные заголовочные файлы языка C++, а также нужные файлы из библиотеки boost и, при компиляции для CUDA (компилятором nvcc), из библиотеки thrust. Библиотека thrust входит в стандартную поставку CUDA и доступна по умолчанию, а путь к библиотеке boost при компиляции должен быть указан явно. Из библиотеки boost используются только заголовочные файлы, компилировать boost не обязательно.

Общие для обеих версий библиотеки классы и функции лежат в пространстве имён `kiam::math`. Классы и функции трёхмерной версии библиотеки лежат в пространстве имён `kiam::math::grid`, а двумерной – в пространстве имён `kiam::math::matrix`.

Опишем классы и функции трёхмерной версии библиотеки. В двумерной версии всё очень похоже. Там, где имя класса в трёхмерной версии библиотеки начинается с grid_, в двумерной версии это будет matrix_. Там, где в трёхмерной версии библиотеки передаются три индекса, в двумерной версии их будет два.

Базовый класс math_object

Класс math_object является базовым классом для всех остальных объектов всех библиотек. Шаблон этого класса принимает два параметра: название конечного класса и название класса заместителя. Если название класса заместителя не указано, то оно полагается равным названию конечного класса. Имея ссылку на класс math_object, можно получить

ссылку на конечный класс (метод self()) и создать объект-заместитель (метод get_proxy()). Приведём полный исходный текст этого класса:

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

     final_type& self(){
          return static_cast<final_type&>(*this);
     }

     const final_type& self() const {
          return static_cast<const final_type&>(*this);
     }
     proxy_type get_proxy() const {
          return self();
     }
};
```

Класс grid_evaluable_object (вычисляемый объект)

Класс grid_evaluable_object является чисто маркерным классом, от которого должны быть пронаследованы все классы вычисляемых объектов. К ним относятся, в частности, сеточные функции и сеточные вычислители, которые получаются как результат «применения» сеточных операторов к вычисляемым объектам (см. ниже). Исходный текст этого класса предельно простой:

```
template<class EO, class _Proxy = EO>
struct grid_evaluable_object : math_object<EO, _Proxy>{};
```

Здесь EO – это название конечного класса вычисляемого объекта, а _Proxy – название класса-заместителя (см. раздел 3.5).

В самом классе нет никакого функционала. Но от классов-наследников определённый функционал требуется. Во всех классах заместителях (про объекты-заместители см. раздел 3.5) классов

наследников должны быть определен хотя бы один из двух функциональных методов:

```
value_type operator()(size_t i, size_t j, size_t k) const;
template<class GC>
value_type operator()(size_t i, size_t j, size_t k, const
   grid_context<GC>& context) const;
```

Второй метод отличается наличием контекста исполнения (см. раздел 3.6). Любой вычисляемый объект должен по запросу возвращать значение, лежащее по определённым индексным координатам. Тип возвращаемого значения (value_type) определяется самим вычисляемым объектом. Если класс заместитель для вычисляемого объекта не задан, то этот класс заместителя совпадает с классом самого вычисляемого объекта. В этом случае эти два функциональных метода должны быть реализованы в самом классе вычисляемого объекта.

В языке C++ есть такое понятие – концепции (concepts), см. например, URL: http://en.wikipedia.org/wiki/Concepts_%28C%2B%2B%29. В сам язык как синтаксическая конструкция концепции войти не успели (в текущий стандарт языка C++11), но предложения есть, и в следующий стандарт языка они с большой вероятностью войдут. На понятийном уровне концепции – это требования к классу, который передан в другой класс или в функцию как параметр шаблона. В этом смысле про эти два функциональных метода можно говорить, что их обязательное наличие – это концепция класса вычисляемого объекта.

Для вычисляемых объектов определена своя арифметика. Вычисляемые объекты можно складывать и вычитать, их можно умножать и делить на константы. К вычисляемому объекту можно прибавлять или вычитать константу, а также применять сеточные операторы.

Класс grid_operator (сеточный оператор)

Класс grid_operator является базовым классом для всех сеточных операторов. Покажем исходный текст этого класса:

```
template<class GO, class _Proxy>
```

```
struct grid_operator : math_object<GO, _Proxy>
{
     template<typename T>
     struct get_value_type
     {
          typedef T type;
     };
     template<class EO>
     grid_operator_evaluator<GO, EO>
     operator()(const grid_evaluable_object<EO, typename
EO::proxy_type>& eobj) const {
          return grid_operator_evaluator<GO, EO>(*this, eobj);
     }
};
#define REIMPLEMENT_GRID_EVAL_OPERATOR() \
     template<class EO> \
     kiam::math::grid::grid_operator_evaluator<type, EO> \
     operator()(const
kiam::math::grid::grid_evaluable_object<EO, typename
EO::proxy_type>& eobj) const { \
          return base_type::operator()(eobj); \
     }
```

Здесь GO – название конечного класса сеточного оператора, а _Proxy – название класса заместителя (см. раздел 3.5).

В базовом классе есть метафункция get_value_type (про метафункции можно прочитать, например, в [15]) и функциональный метод, принимающий вычисляемый объект.

Для сеточных операторов реализована своя арифметика. Сеточные операторы можно складывать и вычитать, при этом образуются новые составные сеточные операторы. Если A и B – некоторые сеточные операторы, а e – некоторый вычисляемый объект, то (A+B)(e)=A(e)+B(e). По сути, операторная арифметика – это способ ещё больше сократить запись.

Для сеточных операторов также определена композиция операторов. Она задаётся через переопределённую операцию умножения. Если A и B – некоторые сеточные операторы, а e – некоторый вычисляемый объект, то (A*B)(e)=A(B(e)).

Как уже говорилось выше, главное назначение сеточных операторов – применение этих операторов к вычисляемым объектам. Функциональный метод как раз и реализует это применение оператора к вычисляемому объекту. Этот метод возвращает сеточный вычислитель - объект класса grid_operator_evaluator, который в свою очередь является вычисляемым объектом. Как уже говорилось выше, любой вычисляемый объект реализует функциональный метод, возвращающий результат некоторого типа (определяемого самим вычисляемым объектом). При применении оператора к вычисляемому объекту тип возвращаемого значения может измениться. Например, оператор градиента преобразует скалярный тип в векторный, а оператор дивергенции – наоборот, преобразует вектор в скаляр. Метафункция get_value_type как раз и описывает это преобразование типа. Реализация этой метафункции в базовом классе тип, возвращаемый вычисляемым объектом, не изменяет. В конечном классе реального оператора эта метафункция может быть переопределена. Именно так и сделано в классах div (дивергенция) и grad (градиент). Приведём исходный текст класса grid_operator_evaluator:

```
template<class GO, class EO>
struct grid_operator_evaluator : grid_evaluable_object<
grid_operator_evaluator<GO, EO> >
{
     typedef grid_operator_evaluator type;
     typedef grid_evaluable_object<type> base_type;
     typedef typename GO::template get_value_type<typename
EO::value_type>::type value_type;
     typedef grid_operator<GO, typename GO::proxy_type>
op_type;
```

```
     typedef grid_evaluable_object<EO, typename EO::proxy_type>
eobj_type;

     grid_operator_evaluator(const op_type& op, const
eobj_type& eobj) : op_proxy(op.get_proxy()),
eobj_proxy(eobj.get_proxy()){}

     template<class GC>
     value_type operator()(size_t i, size_t j, size_t k, const
grid_context<GC>& context) const {
          return op_proxy(i, j, k, eobj_proxy, context);
     }

     value_type operator()(size_t i, size_t j, size_t k) const
{
          return op_proxy(i, j, k, eobj_proxy);
     }

private:
     const typename GO::proxy_type op_proxy;
     const typename EO::proxy_type eobj_proxy;
};
```

Здесь GO – название конечного класса сеточного оператора, а EO – название конечного класса вычисляемого объекта, к которому этот оператор был применён.

Из этого исходного текста видно, что класс grid_operator_evaluator пронаследован от класса grid_evaluable_object, т.е. он является вычисляемым объектом. Тип значения (value_type), возвращаемого классом grid_operator_evaluator в функциональных операторах, определяется путём вызова метафункции get_value_type в сеточном операторе. В качестве параметра этой метафункции передаётся тип, возвращаемый вычисляемым объектом, к которому сеточный оператор был применён.

Два функциональных оператора, которые необходимы в каждом вычисляемом объекте, реализованы путём вызова соответствующих функциональных операторов в сеточном операторе, точнее, в его заместителе. Из этого следует, что в классе заместителе каждого сеточного оператора должны быть определены следующие два функциональных метода, один с контекстом исполнения, другой – без:

```
template<class EOP>
typename get_value_type<typename EOP::value_type>::type
     operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy) const;

template<class EOP, class GC>
typename get_value_type<typename EOP::value_type>::type
     operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy, const grid_context<GC>& context) const;
```

Здесь EOP – название класса заместителя вычисляемого объекта, к которому был применён сеточный оператор.

Эти два функциональных метода похожи на функциональные методы в вычисляемом объекте, отличаются они наличием дополнительного параметра eobj_proxy – ссылка на объект-заместитель вычисляемого объекта, к которому был применён сеточный оператор. Обязательное наличие этих двух операторов – это концепция класса сеточного оператора (про концепции языка C++ см. выше описание класса grid_evaluable_object).

Следует обратить также внимание на то, что в классе grid_operator_evaluator запоминаются сеточный оператор и вычисляемый объект, к которому он был применён. Но так как запоминать ссылки на объекты нельзя, и копии объектов тоже делать нельзя, то создаются объекты-заместители сеточного оператора и вычисляемого объекта.

Класс negate (оператор отрицания)

В качестве простого примера реализации сеточного оператора приведём класс negate. Будучи применённым к любому вычисляемому объекту, он возвращает отрицание того значения, которое возвращает этот вычисляемый объект. Вот исходный текст этого оператора:

```
struct negate : grid_operator<negate>
{
     typedef negate type;
     typedef grid_operator<type> base_type;

     template<class EOP, class GC>
     typename EOP::value_type
     operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy, const grid_context<GC>& context) const {
          return -eobj_proxy(i, j, k, context);
     }

     template<class EOP>
     typename EOP::value_type
     operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy) const {
          return -eobj_proxy(i, j, k);
     }

     REIMPLEMENT_GRID_EVAL_OPERATOR()
};

template<class EO>
grid_operator_evaluator<negate, EO> operator-(
     const grid_evaluable_object<EO, typename EO::proxy_type>&
eobj
){
     return negate()(eobj);
}
```

В классе negate реализованы два функциональных метода, входящих в концепцию сеточного оператора, а также повторно реализован метод, реализующий применение оператора к вычисляемому объекту путём вызова аналогичного метода в базовом классе. Необходимость этой повторной реализации вызвана тем, что компилятор скрывает (делает невидимым) функциональный оператор в базовом классе.

Далее, для вычисляемых объектов переопределён унарный оператор «-», что даёт возможность писать, например, такие конструкции: a = -(b+c); где a – плотная сеточная функция, а b и c – произвольные вычисляемые объекты, например, также плотные сеточные функции.

Класс shift (оператор сдвига)

Оператор сдвига позволяет получить значение, возвращаемое вычисляемым объектом, к которому применяется этот оператор, со сдвигом по индексам. Сдвиг задаётся по всем индексам, и может быть как положительным, так и отрицательным. Вот исходный текст этого класса:

```
struct _s
{
     _s(int sx_, int sy_, int sz_) : sx(sx_), sy(sy_),
sz(sz_){}
     int sx, sy, sz;
};

struct shift : grid_operator<shift>
{
     typedef shift type;
     typedef grid_operator<type> base_type;

     shift(int sx_, int sy_, int sz_) : sx(sx_), sy(sy_),
sz(sz_){}
     shift(const _s& s) : sx(s.sx), sy(s.sy), sz(s.sz){}

     template<class EOP, class GC>
     typename EOP::value_type
```

```
    operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy, const grid_context<GC>& context) const {
        return eobj_proxy(i + sx, j + sy, k + sz, context);
    }

    template<class EOP>
    typename EOP::value_type
    operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy) const {
        return eobj_proxy(i + sx, j + sy, k + sz);
    }

    REIMPLEMENT_GRID_EVAL_OPERATOR()

private:
    int sx, sy, sz;
};

template<class EO>
grid_operator_evaluator<shift, EO> operator<<(
    const grid_evaluable_object<EO, typename EO::proxy_type>&
eobj, const _s& s){
    return shift(s)(eobj);
}
```

Для удобства введён вспомогательный класс _s, в частности, он используется в переопределённом для вычисляемого объекта операторе сдвига. Использовать оператор сдвига можно, например, так:

```
grid_assign(a <<= b << _s(-1, 0, 0)).exec(1, n1 - 1, 1, n2 -
1, 1, n3 - 1);
```

Обратим внимание, что писать просто `a=b<<_s(-1, 0, 0);` нельзя, так как при вычислении значений в точках, где первый индекс равен нулю, произойдёт выход за границы допустимых значений индекса (он станет равным -1).

Класс grid_value_operator (единичный оператор)

Единичный оператор – это, видимо, один из простейших операторов. Он просто возвращает то значение, которое возвращает вычисляемый объект, к которому он применён. Вот его исходный текст:

```
struct grid_value_operator : grid_operator<value_operator>
{
     typedef value_operator type;
     typedef grid_grid_operator<type> base_type;

     template<class EOP, class GC>
     typename EOP::value_type
     operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy, const grid_context<GC>& context) const {
          return eobj_proxy(i, j, k, context);
     }
     template<class EOP>
     typename EOP::value_type
     operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy) const {
          return eobj_proxy(i, j, k);
     }
     REIMPLEMENT_GRID_EVAL_OPERATOR()
};
```

Таким образом, написать a=b; и a=I(b); где I – единичный оператор, а b – вычисляемый объект, это одно и то же. Смысл единичного оператора становится понятным при использовании операторной арифметики. Например, в выражении a=(I+L)(b+c); где L – некоторый сеточный оператор, а b и c – вычисляемые объекты (например, плотные сеточные функции). Без единичного оператора это выражение будет выглядеть гораздо сложнее. Если ввести ещё нулевой оператор, т.е. оператор, возвращающий нулевое значение для всех индексов, то можно будет сказать, что сеточные операторы образуют алгебраическое кольцо, где нулевой оператор является аддитивным нулём, а единичный – мультипликативной единицей. Аддитивной групповой операцией является

операция сложения сеточных операторов с обратной ей операцией вычитания, а мультипликативной операцией является операция композиции сеточных операторов.

Классы grad (оператор градиента) и div (оператор дивергенции)

Эти два класса примечательны тем, что в них делается преобразование типа, возвращаемого вычисляемым объектом, к которому применяется оператор. Оператор градиента преобразует скалярное значение в векторное, а оператор дивергенции – наоборот, преобразует вектор в скаляр.

Поясним, как работать со значениями векторного типа. Как известно, тип данных, с которым работает библиотека, задаётся как параметр шаблона (например, плотной сеточной функции). Это может быть, например, float, double или std::complex<double>. Все эти типы – скалярные, т.е. служат для описания скалярных полей (например, давления, плотности или температуры), но не годятся для описания векторных полей (например, скоростей, ускорений или сил). Для работы с векторными полями в библиотеке имеется специальный тип данных – класс vector_value. Этот класс шаблонный и принимает в качестве параметра шаблона тип хранимых значений (как тип std::complex). Внутри он хранит три значения указанного типа – проекции на три оси. Класс vector_value также, как и любой другой арифметический тип (для которого есть переопределенные арифметические операции) может быть передан как параметр шаблона для класса dense_grid_function – плотной сеточной функции. Приведём пример использования этого типа данных.

```
dense_grid_function<double> p(100, 100, 100);
dense_grid_function<vector_value<double> > a(100, 100, 100);
… // заполняем как-то эти сеточные функции
double dx = 0.1, dy = 0.1, dz = 0.1
grad<double> g(dx, dy, dz);
div<double> d(dx, dy, dz);
grid_assign(
```

```
     a <<= g(p)
).exec(1, n1 – 1, 1, n2 – 1, 1, n3 – 1);
grid_assign(
     p <<= d(a)
).exec(1, n1 – 1, 1, n2 – 1, 1, n3 – 1);
```

Также, как и для класса shift, нужно ограничивать область исполнения оператора присваивания с помощью функции grid_assign, т.к. оба оператора берут значения вычисляемого объекта, к которому были применены, из соседних точек.

Приведём исходный текст обоих классов:

```
template<typename AT>
struct grad : grid_operator<grad<AT> >
{
     typedef grad type;
     typedef grid_operator<type> base_type;
     typedef AT arg_type;

     template<class T>
     struct get_value_type
     {
          typedef vector_value<T> type;
     };

     grad(arg_type dx, arg_type dy, arg_type dz) : dx2(dx * 2),
dy2(dy * 2), dz2(dz * 2){}

     template<class EOP, class GC>
     vector_value<typename EOP::value_type>
     operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy, const grid_context<GC>& context) const;

     template<class EOP>
     vector_value<typename EOP::value_type>
```

```
     operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy) const;

     REIMPLEMENT_GRID_EVAL_OPERATOR()

private:
     arg_type dx2, dy2, dz2;
};
template<typename AT>
struct div : grid_operator<div<AT> >
{
     typedef div type;
     typedef grid_operator<type> base_type;
     typedef AT arg_type;

     template<class T>
     struct get_value_type;

     template<class T>
     struct get_value_type<vector_value<T> >
     {
          typedef T type;
     };

     div(arg_type dx, arg_type dy, arg_type dz) : dx2(dx * 2),
dy2(dy * 2), dz2(dz * 2){}

     template<class EOP, class GC>
     typename get_value_type<typename EOP::value_type>::type
     operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy, const grid_context<GC>& context) const;

     template<class EOP>
     typename get_value_type<typename EOP::value_type>::type
```

```
    operator()(size_t i, size_t j, size_t k, const EOP&
eobj_proxy) const;

    REIMPLEMENT_GRID_EVAL_OPERATOR()

private:
    arg_type dx2, dy2, dz2;
};
```

Обратите внимание на то, что в обоих классах переопределена метафункция get_value_type. В классе grad она возвращает вектор из значений указанного типа, а в классе div с помощью механизма частичной специализации из класса vector_value извлекается тип хранимых значений, который и возвращается из метафункции.

Класс grid_function (сеточная функция)

Класс grid_function является базовым классом для всех сеточных функций. Этот класс пронаследован от класса grid_evaluable_object, это значит, что все сеточные функции являются вычисляемыми объектами, и, значит, должны удовлетворять концепции вычисляемого объекта. Сам класс grid_function несёт в себе минимальный функционал – он хранит размеры сеточной функции по всем трём осям. Приведём исходный текст этого класса:

```
template<class GF, class _Proxy = GF>
struct grid_function : grid_evaluable_object<GF, _Proxy>
{
    grid_function() : m_size_x(0), m_size_y(0), m_size_z(0){}
    grid_function(size_t size_x, size_t size_y, size_t size_z)
:
        m_size_x(size_x), m_size_y(size_y),
m_size_z(size_z){}

    size_t size_x() const { return m_size_x; }
    size_t size_y() const { return m_size_y; }
    size_t size_z() const { return m_size_z; }
```

```
    void resize(size_t size_x, size_t size_y, size_t size_z)
    {
        m_size_x = size_x;
        m_size_y = size_y;
        m_size_z = size_z;
    }

private:
    size_t m_size_x, m_size_y, m_size_z;
};
```

Здесь параметр шаблона GF – это название конечного класса сеточной функции, а параметр _Proxy – название класса заместителя.

Класс scalar_grid_function (скалярная сеточная функция)

Скалярная сеточная функция – это такая сеточная функция, которая во всех точках принимает одно и то же значение. Это значение можно задать и запросить. Скалярная сеточная функция при любом размере сетки занимает в памяти фиксированный очень маленький размер. Приведём исходный текст класса:

```
template<typename T>
struct scalar_grid_function :
grid_function<scalar_grid_function<T> >
{
    typedef scalar_grid_function type;
    typedef grid_function<type> base_type;
    typedef T value_type;

    scalar_grid_function() : s(value_type()){}
    scalar_grid_function(size_t size_x, size_t size_y, size_t
size_z, value_type s_ = value_type()) :
        base_type(size_x, size_y, size_z), s(s_){}

    value_type get_scalar() const { return s; }
```

```
    value_type operator()(size_t i, size_t j, size_t k) const
{ return s; }

    template<class GC>
    value_type operator()(size_t i, size_t j, size_t k, const
grid_context<GC>& context) const { return s; }

    void operator=(value_type s_){ s = s_; }

    void operator+=(const value_type& value){ s += value; }
    void operator-=(const value_type& value){ s -= value; }
    void operator*=(const value_type& value){ s *= value; }
    void operator/=(const value_type& value){ s /= value; }

private:
    value_type s;
};
```

Скалярная сеточная функция удовлетворяет концепции вычисляемого объекта, но при этом никак не использует передаваемые (в функциональные методы) параметры. Если про сеточные операторы можно сказать, что они образуют алгебраическое кольцо, то про вычисляемые объекты можно сказать, что они образуют аддитивную группу, в которой групповой операцией служит операция сложения с обратной к ней операцией вычитания. Нулём этой группы может служить, например, скалярная сеточная функция, которой в качестве значения задан ноль.

Класс computable_grid_function (вычисляемая сеточная функция)

Вычисляемая сеточная функция совсем не хранит своих значений. Вместо этого она каждый раз их вычисляет. Это может быть удобно, например, для первоначального заполнения плотной сеточной функции. Приведём исходный текст этого класса:

```
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 computable_grid_function type;
     typedef grid_function<
          type,
          computable_grid_function_proxy<F, X, Y, Z>
     > base_type;
     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_type(size_x, size_y, size_z){}
     computable_grid_function(size_t size_x, size_t size_y,
size_t size_z,
          const F& f_, const X& x_, const Y& y_, const Z& z_) :
          base_type(size_x, size_y, size_z), f(f_), x(x_),
y(y_), z(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, math_vector<value_type>& s,
size_t width = 1) const;
    void get_slice_y(size_t j, math_vector<value_type>& s,
size_t width = 1) const;
    void get_slice_z(size_t k, math_vector<value_type>& s,
size_t width = 1) const;

private:
    void get_slice_x(size_t i, size_t y_begin, size_t y_end,
size_t z_begin, size_t z_end, math_vector<value_type>& s,
size_t width) const;
    void get_slice_y(size_t j, size_t x_begin, size_t x_end,
size_t z_begin, size_t z_end, math_vector<value_type>& s,
size_t width) const;
    void get_slice_z(size_t k, size_t x_begin, size_t x_end,
size_t y_begin, size_t y_end, math_vector<value_type>& s,
size_t width) const;

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

Класс computable_grid_function принимает четыре параметра шаблона: функциональный объект F, возвращающий значение по трём координатам точки (не индексам, и реальным координатам), и три функциональных объекта X, Y и Z, вычисляющие реальные координаты по индексу. Все четыре функциональных объекта должны быть пронаследованы от класса math_object. В классах X, Y и Z должен быть реализован функциональный

метод, возвращающий значение координаты по индексу, а в классе F должен быть реализован функциональный метод, возвращающий значение по трём координатам. Кроме того, в классе F должен быть определён тип value_type, указывающий тип возвращаемого значения.

В вычисляемой сеточной функции есть также методы, позволяющие получить срез значений по плоскости, пересекающий одну из осей при фиксированном значении индекса по этой оси. Эти значения затем можно, например, присвоить срезу плотной сеточной функции. Значения записываются в вектор math_vector, который для CUDA хранит данные в памяти графического процессора, а для последовательного варианта – в памяти обычного процессора.

Приведём пример использования вычисляемой сеточной функции:

```
template<typename T>
struct myF : kiam::math::math_object<myF<T> >
{
     typedef T value_type;

     value_type operator()(value_type x, value_type y,
value_type z) const {
          return x * x + y * y;
     }
};

template<typename T>
struct myCoord : kiam::math::math_object<myCoord<T> >
{
     typedef T value_type;
     myCoord() : m_dx(value_type()){}

     value_type get() const { return m_dx; }
     void set(value_type dx){ m_dx = dx; }
     value_type operator()(size_t i) const {
          return i * m_dx;
```

```
    }

private:
    value_type m_dx;
};

int main(int argc, char* argv[])
{
    kiam::math::grid::computable_grid_function<myF<double>,
myCoord<double>, myCoord<double>, myCoord<double> > cf(100,
100, 100);
    cf.get_x().set(0.1);
    cf.get_y().set(0.2);
    cf.get_z().set(0.05);
    kiam::math::grid::dense_grid_function<double> df(100, 100,
100);
    df = cf;
    return 0;
}
```

В этом примере плотная сеточная функция df заполняется на основе вычисляемой сеточной функции cf. Объект myF вычисляет функцию от трёх координат, равную x*x + y*y. Объект myCoord переводит индексные координаты в реальные в соответствии с заданным фиксированным шагом.

Также следует обратить внимание на то, что для того, чтобы задать параметры объектам F, X, Y и Z, нужно эти объекты запросить у вычисляемой сеточной функции.

Плотные сеточные функции

Плотные сеточные функции – это сеточные функции, хранящие свои значения во всех точках. Существует несколько видов плотных сеточных функций: это обычная плотная сеточная функция, сама занимающаяся хранением данных, и пользовательская плотная сеточная функция, для которой указатель на данные даёт пользователь. Для доступа к данным применяется функциональный оператор с тремя параметрами – индексами

по трём направлениям. Данные в памяти располагаются вначале по первому индексу, а затем по второму и третьему (как в Фортране). Все плотные сеточные функции должны наследоваться от класса **abstract_dense_grid_function**. Что касается концепции плотной сеточной функции (требуемый набор методов), то обязательными являются методы begin() и end(), возвращающие итераторы начала и конца массива данных. Это должен быть т.н. «forward» итератор, у которого должен быть определён метод преинкремента (++).

Класс abstract_dense_grid_function

Класс abstract_dense_grid_function является бызовым классом для всех плотных сеточных функций. Это чисто маркерный класс, не несущий в себе никакой дополнительной функциональности.

Класс dense_grid_function_base

Класс dense_grid_function_base пронаследован от класса abstract_dense_grid_function. Класс dense_grid_function_base организует хранение данных, для чего используется класс math_vector, который хранит данные на том устройстве, где производятся вычисления: в памяти обычного процессора для последовательной версии и в памяти графического ускорителя в версии для CUDA. Приведём исходный текст этого класса:

```
template<typename V>
struct dense_grid_function_base_proxy;

template<class DGF, typename T>
struct dense_grid_function_base :
     abstract_dense_grid_function<DGF,
dense_grid_function_base_proxy<T> >,
     math_vector<T>
{
     typedef abstract_dense_grid_function<DGF,
dense_grid_function_base_proxy<T> > base_type;
     typedef math_vector<T> baseV;
```

```
     typedef typename baseV::value_type value_type;
     typedef typename baseV::reference reference;
     typedef typename baseV::const_reference const_reference;

protected:
     dense_grid_function_base(){}
     dense_grid_function_base(size_t size_x, size_t size_y,
size_t size_z) :
          base_type(size_x, size_y, size_z), baseV(size_x *
size_y * size_z){}

public:
     reference operator()(size_t i, size_t j, size_t k){
          return baseV::operator[]((k * base_type::size_y() +
j) * base_type::size_x() + i);
     }

     const_reference operator()(size_t i, size_t j, size_t k)
const {
          return baseV::operator[]((k * base_type::size_y() +
j) * base_type::size_x() + i);
     }

     void resize(size_t size_x, size_t size_y, size_t size_z)
     {
          base_type::resize(size_x, size_y, size_z);
          baseV::resize(size_x * size_y * size_z);
     }

     void operator=(const value_type& value){
          MATH_FILL(baseV::begin(), baseV::end(), value);
     }

     void operator=(const dense_grid_function_base& func)
```

```
    {
        assert(func.size() == baseV::size());
        MATH_COPY(func.begin(), func.end(), baseV::begin());
    }
};
```

У шаблона класса два параметра – название конечного класса и тип хранимых значений. В этом классе определён функциональный метод (с тремя параметрами - индексами) для доступа к данным и задаётся класс объекта-заместителя. Есть также два оператора присваивания, один из которых заполняет всё одним и тем же значением, а второй копирует значения из другой плотной сеточной функции. Эти два оператора присваивания работают максимально быстро.

Класс simple_dense_grid_function (простая плотная сеточная функция)

Класс simple_dense_grid_function пронаследован от класса dense_grid_function_base, ничего в него не добавляя. Это очень простая плотная сеточная функция с минимальным функционалом. Приведём её исходный текст:

```
template<typename T>
struct simple_dense_grid_function :
dense_grid_function_base<simple_dense_grid_function<T>, T>
{
    typedef simple_dense_grid_function type;
    typedef dense_grid_function_base<type, T> base_type;

    simple_dense_grid_function(){}
    simple_dense_grid_function(size_t size_x, size_t size_y,
size_t size_z) : base_type(size_x, size_y, size_z){}
};
```

Класс принимает один параметр шаблона – тип хранимых значений. Хотя этот класс не добавляет никакой функциональности к классу dense_grid_function_base, он включён в библиотеку, так как задаёт

требуемый в классе dense_grid_function_base первый параметр шаблона – имя конечного класса.

Класс dense_grid_function (плотная сеточная функция)

Класс dense_grid_function – основной класс плотной сеточной функции. Этот класс реализует следующий функционал:

- Клонирование (метод clone). В результате создаётся плотная сеточная функция того же размера, заполненная нулевыми значениями.
- Оператор присваивания вычисляемого объекта. В правой части этого оператора присваивания может быть сложное выражение, которое запоминается в вычисляемом объекте. Этот метод запускает отложенные вычисления. Одна и та же запомненная цепочка вычислений выполняется для всех точек плотной сеточной функции. В последовательной версии выполняются три вложенных цикла по трём измерениям, а в версии для CUDA осуществляется запуск одного ядра.
- Прибавление и вычитание вычисляемого объекта. Эти методы аналогичны оператору присваивания, но вместо присваивания вычисленного в каждой точке значения делается его прибавление или вычитание к текущему значению.
- Оператор присваивания произвольной плотной сеточной функции (принимает ссылку на класс abstract_dense_grid_function). Оптимизирует присваивание по сравнению с произвольным вычисляемым объектом, используя тот факт, что у любой плотной сеточной функции должен быть метод begin(), возвращающий итератор начала массива данных. Этот метод просто их копирует.
- Операторы прибавления и вычитания произвольной плотной сеточной функции. Аналогичны предыдущему оператору, но не присваивают значения, а прибавляют или вычитают их.

- Операторы прибавления, вычитания, умножения и деления константы. Ко всем хранящимся значениям прибавляется, вычитается, умножается или делится одно и то же значение.
- Считывание и запись среза (slice) данных перпендикулярно одной из осей при заданном индексе по этой оси. Данные считываются в объект класса math_vector. Есть также методы, заполняющие срез указанным значением.
- Копирование среза данных перпендикулярно одной из осей в другую плотную сеточную функцию (abstract_dense_grid_function).
- Считывание и запись среза данных перпендикулярно двум осям (вдоль третьей оси).
- Оператор присваивания, принимающий объект класса host_dense_grid_function. Этот оператор присваивания особенно актуален для CUDA. host_dense_grid_function – это плотная сеточная функция, хранящая данные в памяти основного (host) процессора. Этот оператор служит для пересылки данных из памяти основного процессора в память на графическом устройстве. Пересылку данных в обратную сторону можно сделать аналогичным образом, присвоив объекту класса host_dense_grid_function плотную сеточную функцию.

Вот исходный текст (определение без реализации) этого класса:

```
template<typename T>
struct host_dense_grid_function;

template<typename T>
struct dense_grid_function :
dense_grid_function_base<dense_grid_function<T>, T>
{
     typedef dense_grid_function type;
     typedef dense_grid_function_base<type, T> base_type;
```

```
    typedef typename base_type::baseV baseV;

    dense_grid_function(){}
    dense_grid_function(size_t size_x, size_t size_y, size_t
size_z) : base_type(size_x, size_y, size_z){}

    void operator=(const typename base_type::value_type&
value){ base_type::operator=(value); }
    void operator=(const base_type& func){
base_type::operator=(func); }

    dense_grid_function clone() const;

    template<class EO>
    void operator=(const grid_evaluable_object<EO, typename
EO::proxy_type>& eobj);

    template<class EO>
    void operator+=(const grid_evaluable_object<EO, typename
EO::proxy_type>& eobj);

    template<class EO>
    void operator-=(const grid_evaluable_object<EO, typename
EO::proxy_type>& eobj);

    template<class ADGF>
    void operator+=(const abstract_dense_grid_function<ADGF,
typename ADGF::proxy_type>& rhs);

    template<class ADGF>
    void operator-=(const abstract_dense_grid_function<ADGF,
typename ADGF::proxy_type>& rhs);

    void operator+=(const T& value);
    void operator-=(const T& value);
```

```
    void operator*=(const typename get_scalar_type<T>::type&
value);
    void operator/=(const typename get_scalar_type<T>::type&
value);

    void get_slice_x(size_t i, math_vector<T>& s, size_t width
= 1) const;
    void set_slice_x(size_t i, const math_vector<T>& s, size_t
width = 1);
    void set_slice_x(size_t i, const T& value, size_t width =
1);

    void get_slice_y(size_t j, math_vector<T>& s, size_t width
= 1) const;
    void set_slice_y(size_t j, const math_vector<T>& s, size_t
width = 1);
    void set_slice_y(size_t j, const T& value, size_t width =
1);

    void get_slice_z(size_t k, math_vector<T>& s, size_t width
= 1) const;
    void set_slice_z(size_t k, const math_vector<T>& s, size_t
width = 1);
    void set_slice_z(size_t k, const T& value, size_t width =
1);

    template<class ADGF>
    void copy_slice_x(size_t i,
abstract_dense_grid_function<ADGF, typename ADGF::proxy_type>&
dest, size_t width = 1) const;

    template<class ADGF>
```

```
    void copy_slice_y(size_t j,
abstract_dense_grid_function<ADGF, typename ADGF::proxy_type>&
dest, size_t width = 1) const;

    template<class ADGF>
    void copy_slice_z(size_t k,
abstract_dense_grid_function<ADGF, typename ADGF::proxy_type>&
dest, size_t width = 1) const;

    void get_slice_xy(size_t i, size_t j, math_vector<T>& s)
const;
    void set_slice_xy(size_t i, size_t j, const
math_vector<T>& s);
    void get_slice_xz(size_t i, size_t k, math_vector<T>& s)
const;
    void set_slice_xz(size_t i, size_t k, const
math_vector<T>& s);
    void get_slice_yz(size_t j, size_t k, math_vector<T>& s)
const;
    void set_slice_yz(size_t j, size_t k, const
math_vector<T>& s);

    void operator=(const host_dense_grid_function<T>& hf){
        baseV::base_type::operator=(hf);
    }
};
```

Присваивание выражений плотной сеточной функции

У плотной сеточной функции есть оператор присваивания, принимающий произвольный вычисляемый объект. Недостатком этого способа присваивания является то, что присваивание осуществляется для всех точек плотной сеточной функции. Однако это не всегда допустимо или желательно. Пусть, например, a и b – это плотные сеточные функции, a l – оператор Лапласа и мы хотим вычислить a=l(b). Но оператор Лапласа для вычисления значения в точке берёт значения из соседних точек, а из

этого следует, что вычисления можно провести только во внутренних точках сетки. Могут быть и другие причины для ограничения области исполнения оператора присваивания.

У указанного оператора присваивания есть ещё один недостаток. Как уже было сказано выше, в случае CUDA присваивание делается в одном вызове ядра (kernel). Но допустим, что нужно присвоить значения нескольким переменным, имеющим одинаковый размер. Тогда для присваивания каждой переменной будет вызвано своё ядро, хотя, казалось бы, это можно было бы сделать в одном ядре.

Наконец, третий недостаток – это использование исполнителя по умолчанию, который не всегда наиболее оптимален.

В библиотеке есть способ присваивания выражений плотным сеточным функциям, лишённый всех указанных недостатков. Он может присваивать в одном вызове ядра до 10 выражений с явным указанием области исполнения и исполнителя. Кроме того, этот метод поддерживает контекст исполнения (см. 3.6). Для этого предназначена функция grid_assign. Эта функция принимает от 1 до 10 т.н. «присваивателей» - выражений вида a <<= e, где a – плотная сеточная функция, а e – вычисляемый объект. Максимальное количество параметров функции можно при необходимости увеличить, задав перед включением заголовочных файлов библиотеки макропеременную MAX_ASSIGNENT_SIZE. Если эта переменная не задана, то она полагается равной 10. Функция возвращает объект класса grid_package, в котором сохранены все указанные в параметрах функции выражения. Приведём часть исходного текста класса grid_package:

```
struct grid_package
{
public:
     template<class GCB, class E>
     grid_package& exec(size_t x_begin, size_t x_end, size_t
y_begin, size_t y_end, size_t z_begin, size_t z_end,
```

```
    const grid_context_builder<GCB, typename GCB::proxy_type>&
context_builder,
    grid_executor<E, typename E::proxy_type> &executor);

    template<class GCB>
    grid_package &exec(size_t x_begin, size_t x_end, size_t
y_begin, size_t y_end, size_t z_begin, size_t z_end,
    const grid_context_builder<GCB, typename GCB::proxy_type>
&context_builder
){
        default_grid_executor executor;
        return exec(x_begin, x_end, y_begin, y_end, z_begin,
z_end, context_builder, executor);
    }

    template<class E>
    grid_package& exec(size_t x_begin, size_t x_end, size_t
y_begin, size_t y_end, size_t z_begin, size_t z_end,
   grid_executor<E, typename E::proxy_type> &executor);

   grid_package &exec(size_t x_begin, size_t x_end, size_t
y_begin, size_t y_end, size_t z_begin, size_t z_end
   ){
        default_grid_executor executor;
        return exec(x_begin, x_end, y_begin, y_end, z_begin,
z_end, executor);
    }
```

Все методы exec принимают первые 6 параметров, задающие область исполнения, есть методы, принимающие, кроме того, построитель контекста исполнения и исполнитель. Каждый метод exec исполняет всё в одном вызове одного ядра. Кроме того, все эти методы возвращают ссылку на сам объект, что даёт возможность написать цепочку вызовов. Например:

```
grid_assign(
    a <<= e1,
```

```
      b <<= e2,
      c <<= e3)
.exec(0, 1, 0, n2, 0, n3)
.exec(n1 - 1, n, 0, n2, 0, n3);
```

Здесь a, b и c – плотные сеточные функции, а e1, e2, e3 – выражения, возвращающие вычисляемые объекты.