РЕАЛИЗАЦИЯ ЯВНОЙ РАЗНОСТНОЙ СХЕМЫ ДЛЯ РЕШЕНИЯ ДВУМЕРНОГО УРАВНЕНИЯ ТЕПЛОПРОВОДНОСТИ НА ГРАФИЧЕСКОМ ПРОЦЕССОРНОМ УСТРОЙСТВЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ CUDA

Д.В. Деги, А.В. Старченко, А.А. Трунов

1. **Введение**

В целом ряде недавних работ было показано высокое ускорение (1-2 порядка) вычислений общего назначения (относительно центральных процессорных устройств (ЦПУ) традиционной архитектуры) при использовании графических процессорных устройств (ГПУ), см., например, [1, 2].

Одним из наиболее совершенных программно-аппаратных комплексов в настоящее время является платформа NVidia CUDA (Compute Unified Device Architecture). Стоит отметить, что существует альтернативная платформа, обладающая (в отличие от CUDA) свойством переносимости в классе ГПУ — OpenCL [3], поддержанная многими производителями аппаратного обеспечения (AMD, NVidia и др.). Однако на данный момент OpenCL-программы требуют значительных усилий по оптимизации для каждого типа ГПУ. Поэтому в данной работе авторы ограничились рассмотрением технологии CUDA.

Целью настоящей работы является исследование возможностей CUDA для решения нестационарных задач математического моделирования с применением явных разностных схем. В данной работе поставленная проблема изучается на модельной задаче — решении двумерного уравнения теплопроводности с применением явной разностной схемы (пятиточечный шаблон типа «крест»).

2. **Модельная задача**

Рассмотрим уравнение теплопроводности с граничными условиями 1-го рода в следующей постановке:

$$\frac{\partial T}{\partial t}=\alpha\left(\frac{\partial^2 T}{\partial^2 x}+\frac{\partial^2 T}{\partial^2 y}\right);$$

$$T|_G=T_b(x,y);\quad T_{t=0}=100;$$

$$0\leqslant x,y\leqslant 1;\quad 0\leqslant t\leqslant \tau.$$

Для решения уравнения будем использовать явную разностную схему «крест»:

$$T^{n+1}_{i,j}=T^{n}_{i,j}+\alpha\Delta t\left(\frac{T^{n}_{i+1,j}-2T^{n}_{i,j}+T^{n}_{i-1,j}}{\Delta x^2}+\frac{T^{n}_{i,j+1}-2T^{n}_{i,j}+T^{n}_{i,j-1}}{\Delta y^2}\right);$$

$$i=\overline{1,m_x};\quad j=\overline{1,m_y};$$

$$T^{n}_{0,j}=T_b(x_0,y_j);\quad T^{n}_{m_x+1,j}=T_b(x_{m_x+1},y_j);\quad j=\overline{0,m_y};$$

$$T^{n}_{i,0}=T_b(x_i,y_0);\quad T^{n}_{i,m_y+1}=T_b(x_i,y_{m_y+1});\quad i=\overline{0,m_x};$$

$$T^{0}_{i,j}=100;\quad i=\overline{0,m_x};\quad j=\overline{0,m_y}.$$

Данная задача имеет большой ресурс параллелизма — вычисление значения для каждого узла двумерной сетки на новом временном слое не зависит от других узлов сетки на новом временном слое (при использовании двух двумерных массивов). Необходима только синхронизация при переходе от предыдущего временного слоя к следующему.

3. Вычислительные технологии

ГПУ NVidia с упрощённой аппаратной точки зрения представляют собой набор мультипроцессоров, слабо друг с другом связанных, и имеющих медленную общую оперативную память размером порядка 1 Гбайт. Физически оперативная память (global memory) находится на плате ГПУ и имеет минимум на порядок большую пропускную способность, чем оперативная память ЦПУ. Каждый мультипроцессор состоит из нескольких (порядка 10) простых вычислительных ядер, имеющих быструю разделяемую (shared) память небольшого размера (~10 кбайт). У ГПУ распространённых поколений (compute capability 1.x) фактически отсутствует кэш-память в традиционном смысле. Предполагается, что разработчик будет использовать быструю разделяемую память мультипроцессора в качестве управляемой вручную кэш-памяти.

Используемая модель параллелизма — одна инструкция, много потоков (SIMT). Параллельные процессы (нити, threads) объединяются в блоки потоков, для которых возможно выполнение операций синхронизации и обмена данными. Все нити блока выполняются на одном и том же мультипроцессоре и, поэтому, имеют возможность обмена данными через доступ к разделяемой памяти. Блоки потоков

группируются в сетку блоков (grid). Блоки потоков могут обмениваться данными только через медленную оперативную память ГПУ. Явные операции синхронизации для сетки блоков не предусмотрены. Синхронизация может быть осуществлена посредством ЦПУ, либо с использованием атомарных(atomic) инструкций.

Технология CUDA предоставляет разработчику набор расширений языка C/C++ для описания параллелизма и запуска параллельной программы на ГПУ.

Согласно рекомендациям производителя [4, 5] количество потоков, запускаемых на исполнение должно быть много больше, чем количество вычислительных ядер, что позволяет скрыть задержки при доступе к медленной оперативной памяти: если исполняемая группа потоков (варп, warp) ожидает данные из оперативной памяти, то планировщик передаёт управление следующему варпу.

Рассмотрим несколько подходов к реализации вычислений на ГПУ.

Способ 1

Каждый узел сетки обрабатывает один поток, копируя необходимые для этого данные из медленной оперативной памяти независимо от других потоков. Недостаток этого подхода очевиден: отсутствует повторное использование уже принятых данных. Почти все внутренние узлы сетки будут скопированы из памяти 4 раза (вследствие использования шаблона «крест»).

Количество читаемых из оперативной памяти элементов массива:

$4(n_x-2)(n_y-2) + 2\cdot3(n_x-2) + 2\cdot3(n_y-2) + 2\cdot(n_x-2) + 2\cdot(n_y-2) + 4\cdot2 = 4(n_x-2)(n_y-2) + 8(n_x-2) + 8(n_y-2) + 8 \approx 4n_xn_y$, при больших n_x и n_y.

Количество записываемых в оперативную память элементов массива: n_xn_y. Таким образом, количество пересылок приблизительно равно $5n_xn_y$.

В идеальном случае (при однократном копировании исходного массива) количество пересылок составит $2n_xn_y$. Поскольку копирование данных занимает время на 2 порядка большее, чем выполнение операции умножить-сложить (MAD), то максимальный ресурс ускорения вычислений составляет около $5n_xn_y/(2n_xn_y) = 2{,}5$ раз.

Способы 2 и 3

Доработаем первый способ следующим образом. Организуем потоки блока в двумерную структуру (для индексации потока внутри блока используется два целых числа). Блоки потоков организуем в двумерную сетку, так что каждому узлу расчётной области по-прежнему соответствует один поток. При этом каждому блоку потоков будет соответствовать своя двумерная подобласть.

Каждый блок потоков предварительно копирует вычисляемую двумерную подобласть в быструю разделяемую (shared) память, выполняет синхронизацию, гарантирующую окончание процесса копирования, затем каждый поток внутри блока вычисляет свой узел сетки и сохраняет рассчитанное значение в медленную память. Таким образом, данный способ предусматривает ручную имитацию кэш-памяти, позволяющей повторно использовать полученные из медленной памяти данные.

Для расчёта каждой подобласти необходимо использовать граничные значения соседних подобластей. Здесь возникает два варианта реализации.

Способ 2. Все потоки блока копируют из глобальной памяти подобласть с граничными узлами, а затем потоки, соответствующие внутренним ячейкам подобласти выполняют расчёт. При использовании размера подобласти 32x16, конфигурация блока потоков следующая - 34x18. Дальнейшее увеличение размеров подобласти наталкивается на аппаратное ограничение количества потоков в блоке.

Способ 3. Все потоки блока копируют из глобальной памяти в разделяемую память внутренние узлы подобласти, затем часть потоков копирует граничные узлы из глобальной памяти. При использовании размера подобласти 32x16, конфигурация блока потоков следующая — 32x16.

В случае способа 2 запросы на копирование из глобальной памяти получаются несгруппированными (uncoalesced) в силу того, что в пределах некоторых полуварпов (halfwarp) адреса копируемых элементов не обладают свойством близости.

У способа 3 также есть недостаток: наличие программных ветвлений для потоков в пределах варпа, замедляющих работу вычислителя (ГПУ приходится выполнять каждую ветвь ветвления, поскольку все потоки варпа выполняют одну и ту же инструкцию).

Способ 4

Поскольку блок потоков, в силу ограничений архитектуры, не может скопировать данные из своей разделяемой памяти в разделяемую память другого блока потоков, то неизбежно повторное копирование данных из глобальной памяти на границах подобластей. В случае способов 2 и 3 повторному копированию подвергаются столбцы и строки на границах подобластей. В силу аппаратных ограничений, для сеток, имеющих порядка 10^8 узлов, количество активных блоков потоков для предлагаемой выше схемы распараллеливания много меньше общего количества блоков в сетке. Поэтому, изменив схему декомпозиции расчётной области можно сократить количество повторно копируемых элементов, не ухудшив параллелизма. Особенно актуально исключение повторного копирования элементов столбцов двумерного массива (в используемом языке программирования Си элементы матриц хранятся в памяти по строкам), поскольку уменьшается доля несгруппированных запросов к глобальной памяти ГПУ (относительно способа 3, который взят за основу для способа 4; поскольку способ 3 быстрее - см. ниже в п. 4).

Проведём одномерную декомпозицию расчётной области. При этом каждому блоку потоков достаётся полоса расчётной области, состоящая из нескольких строк массива, т.е. формируется одномерная сетка блоков. Блок потоков остаётся двумерным, его работа сводится к последовательному расчёту прямоугольных частей (размерностью блока) доставшейся ему подобласти. Наглядно это можно представить как перемещение рассчитываемой «плитки» вдоль столбцов выделенной полосы массива. При этом устраняется повторное копирование столбцов массива за счёт того, что на каждой итерации внутри подобласти (перемещение «плитки» на один шаг) граничные узлы передаются из последнего столбца массива, расположенного в разделяемой памяти, в первый столбец.

4. **Результаты**

При тестировании программной реализации алгоритмов использовалась ЭВМ следующей конфигурации: четырёхядерный ЦПУ Intel Core2 Quad 9550 2,83 ГГц с поддержкой HyperThreading, 12 Гбайт ОЗУ, ГПУ NVidia GTX 260 (24 восьмиядерных мультипроцессоров, общее количество потоковых ядер 192; 1,3 ГГц; 2 Гб ОЗУ).

Размерность области 14400x14400, расчёты с одинарной точностью (двойная точность на данной версии ГПУ поддерживается недостаточно), количество шагов по времени — 100 (синхронизация между итерациями посредством ЦПУ (функция cudaThreadSynchronize)), размерность блока потоков 32x16.

Результаты расчётов приведены в таблице 1, ускорение вычислено относительно способа 1. Время передачи данных между ЦПУ и ГПУ — 0,6 с.

Таблица 1

	Способ 1	Способ 2	Способ 3	Способ 4
Время вычислений на ГПУ (без учёта обмена данными ЦПУ-ГПУ), с	7,41	6,26	5,96	4,03
Ускорение	1	1,18	1,24	1,84

Для сравнения производительности ГПУ с современными многоядерными ЦПУ была написана OpenMP-программа, реализующая аналогичные вычисления. Время её работы при одинаковых параметрах с CUDA-программой составляет 29,6 с. Таким образом, ускорение при использовании ГПУ составляет 6,4 раза (с учётом времени обмена данными между ЦПУ и ГПУ).

5. **Выводы**

1. Получена хорошая эффективность при оптимизации параллельной программы для ГПУ (1,84/2,5 = 0,74 от идеальной — см. п.3, способ 1).
2. Получено хорошее ускорение относительно OpenMP-программы, выполняемой на четырёхядерном ЦПУс поддержкой HyperThreading — 6,4 раза (ускорение относительно одноядерной версии 48 раз).
3. Дальнейшее совершенствование ГПУ позволит преодолеть текущие недостатки (малый объём памяти, трудоёмкость программирования) и расширит область их применения.

ЛИТЕРАТУРА:

1. Ю.Ю. Клосс, Ф.Г. Черемисин, П.В. Шувалов Решение уравнения Больцмана на графических процессорах // Вычислительные методы и программирование, т. 11, 2010.
2. G. Stantchev, W. Dorland, N. Gumerov Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU // Journal of Parallel and Distributed Computing 68 (2008) 1339-1349.
3. www.khronos.org/opencl
4. NVIDIA CUDA Programming Guide 3.0 // www.nvidia.com
5. NVIDIA CUDA Best Practices Guide 3.0 // www.nvidia.com