# Глава 6

# Архитектура GPU

В данной главе рассматриваются сущствующие архитектуры GPU и общие приемы оптимизации программ.

## 6.1. Архитектура GPU

GPU архитектур G80, Tesla, Fermi и Kepler построены как масштабируемый массив потоковых мультипроцессоров (Streaming Multiprocessor, SM – для архитектур до Fermi включительно, SMX – для Kepler). Когда на GPU запускается CUDA-ядро, то блоки его сетки выполняются на доступных мультипроцессорах. При этом на каждый мультипроцессор целиком помещается до 8 блоков. По мере того, как отдельные блоки завершают свое выполнение, их место занимают новые блоки. Это позволяет даже на небольшом числе потоковых мультипроцессоров запустить на выполнение сетку с очень большим числом блоков. При этом планировщик блоков «скрыт» от программиста – ядро просто запускается на выполнение, все остальное делает устройство.

На рис. 6.1 и рис. 6.2 представлены архитектуры мультипроцессора GPU Tesla C870 и Tesla C1060. Каждый потоковый мультипроцессор содержит восемь *скалярных ядер* (SP, Scalar Processor), каждая нить выполняется на одном из этих ядер. Кроме скалярных ядер, потоковый мультипроцессор также содержит два блока для вычисления специальных функций (SFU, Special Function Unit), блок управления командами (Instruction Unit) и собственную память. В потоковых мультипроцессорах GPU Tesla C1060 дополнительно реализован специальный блок для обработки 64-битных чисел с плавающей точкой (Double Precision Unit).

Непосредственно в мультипроцессоре находится память следующих типов:

- набор 32-битных регистров (register file);
- разделяемая память, доступная всем скалярным ядрам;
- кэш константной памяти, доступный на чтение всем скалярным ядрам;
- кэш текстурной памяти, доступный на чтение всем скалярным ядрам.

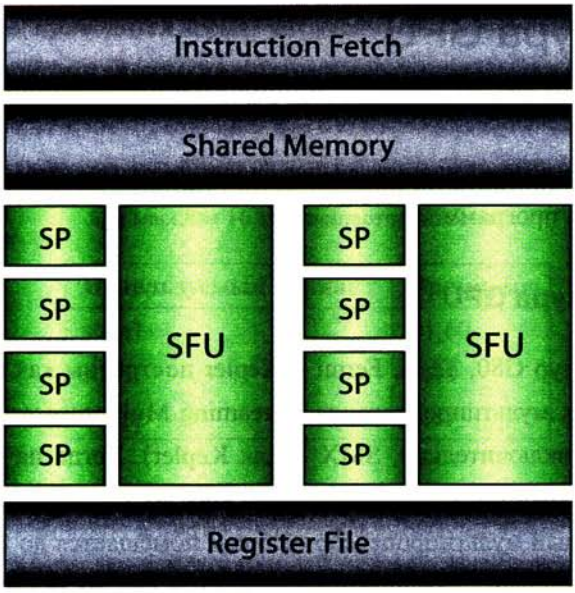


*Рис. 6.1.* Устройство потокового мультипроцессора для GPU Tesla C870/GeForce 8800

Для доступа к текстурной памяти предназначен специальный текстурный блок (TEX), совместно используемый сразу несколькими мультипроцессорами. Текстурный блок вместе со своей группой мультипроцессоров образует так называемый Texture Processing Cluster (TPC) (рис. 6.3).

На рис. 6.4 представлен мультипроцессор GPU Tesla M2090 архитектуры Fermi, который значительно увеличился в размерах по сравнению с предыдущими архитектурами. GPU Tesla M2090 содержит 16 таких мультипроцессоров (рис. 6.5). Каждый мультипроцессор имеет 32 скалярных ядра, четыре блока для вычисления специальных функций (SFU) и два текстурных блока (TEX). Кроме типов памяти, перечисленных для устройств с compute capability 1.x, в мультипроцессор архитектуры Fermi встроен кэш первого уровня, совмещенный с разделяемой памятью (см. секцию 3.2.1). Кэш-память второго уровня размером 768 Кбайт (L2-cache) является общей для всех мультипроцессоров.

Группу из четырех мультипроцессоров архитектуры Fermi принято называть GPC (Graphics Processing Cluster) вместо аббревиатуры TPC, использовавшейся в

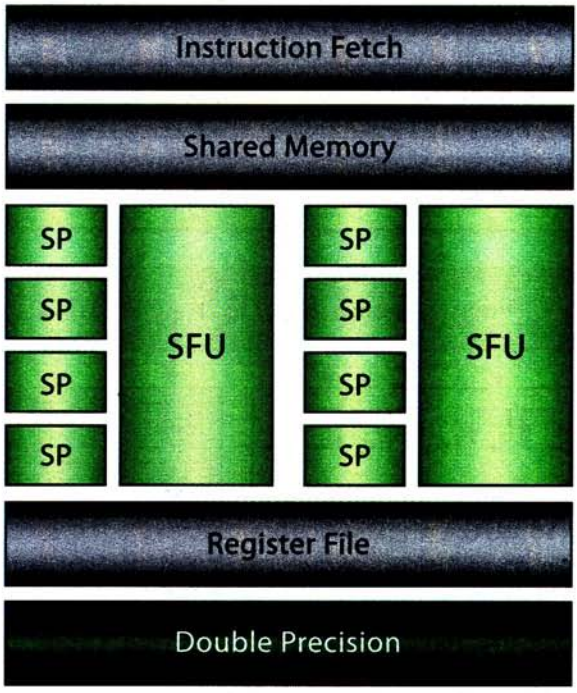


*Рис. 6.2.* Устройство потокового мультипроцессора для GPU Tesla C1060/GeForce GTX 260

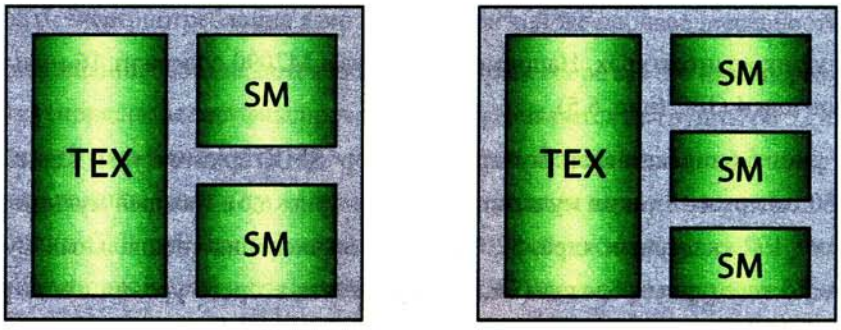


*Рис. 6.3.* Структура TPC для архитектур Tesla C860 (слева) и Tesla C1060 (справа)

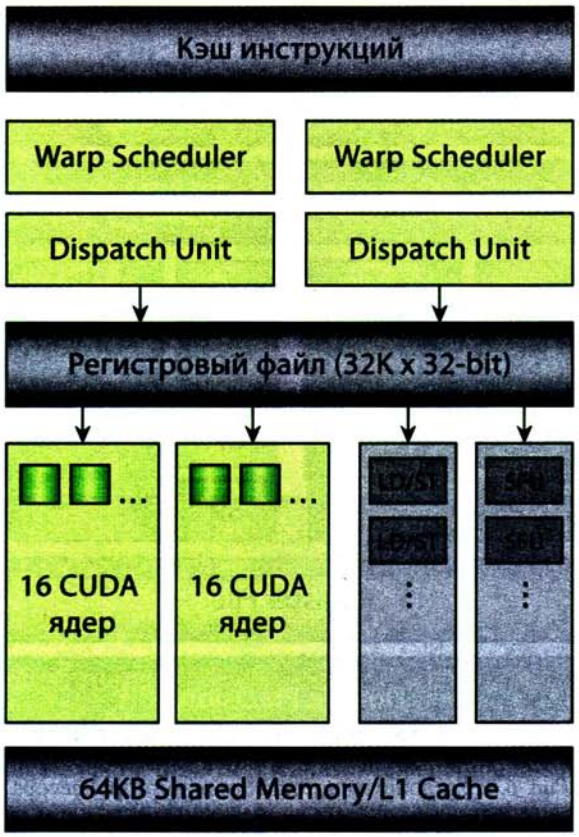


*Рис. 6.4.* Устройство потокового мультипроцессора для GPU Tesla M2090/GeForce GTX 580

предыдущих архитектурах. Например, GPU Tesla M2090 содержит 16 мультипроцессоров или 4 GPC (рис. 6.5).

На рис. 6.6 представлен мультипроцессор SMX архитектуры Kepler. GPU GTX 680 содержит 8 таких мультипроцессоров (рис. 6.6). Каждый мультипроцессор имеет 192 скалярных ядра, 32 блока для вычисления специальных функций (SFU) и 16 текстурных блоков (TEX). Кэш-память второго уровня имеет размер 512 Кбайт.

В архитектуре Kepler были упразднены несколько компонентов, использовавшихся для определения времени выполнения инструкций. Вместо этого латент-

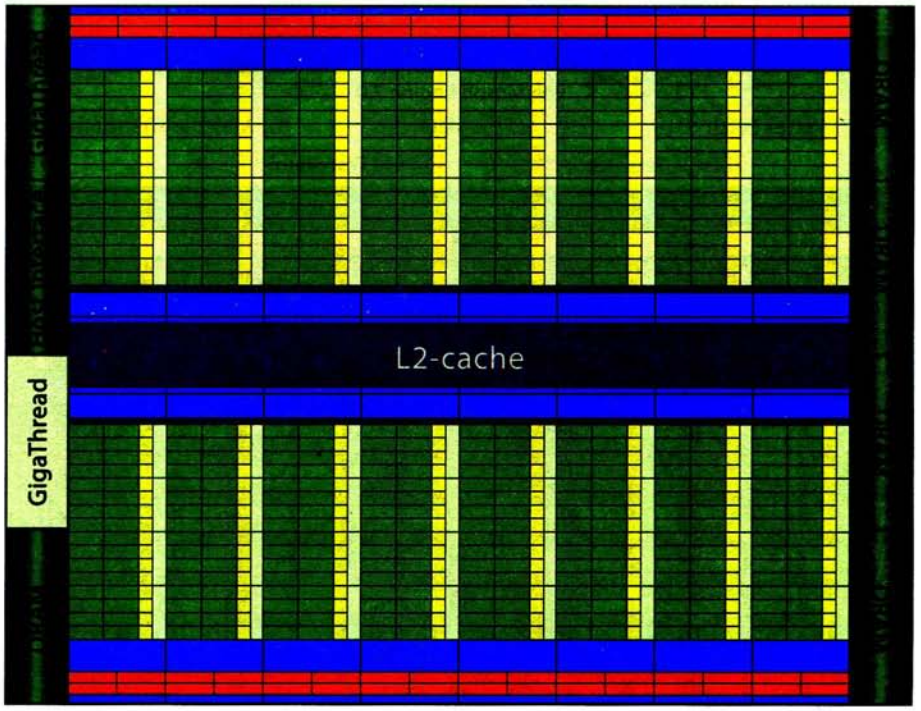


*Рис. 6.5.* Архитектура Fermi (Tesla M2090)

ности расчитываются на этапе компиляции и записываются в ассемблерный код (рис. 6.7). Скалярные ядра в Kepler работают на обычной частоте, в то время как в предыдущих архитетурах использовались ядра с удвоенной частотой. Это позволило снизить энергопотребление ядер в 2 раза при той же производительности.

При выполнении блока на потоковом мультипроцессоре все его нити перенумеровываются (порядок всегда фиксированный) и разбиваются на группы по 32 нити, каждая такая группа называется *варп* (warp). Нити в составе варпа выполняют физически параллельно одну и ту же команду (нити разных варпов могут выполнять разные команды). В таблице 6.1 указано, сколько требуется тактов одному варпу на выполнение одной арифметической операции на одном мультипроцессоре для устройств с compute capability 1.x и 2.0.

Если нити одного варпа должны идти по разным веткам кода (например, из-за условного оператора), то выполнятся все проходимые ветки. Даже если только


*Рис. 6.6.* Архитектура Kepler (GTX 680)

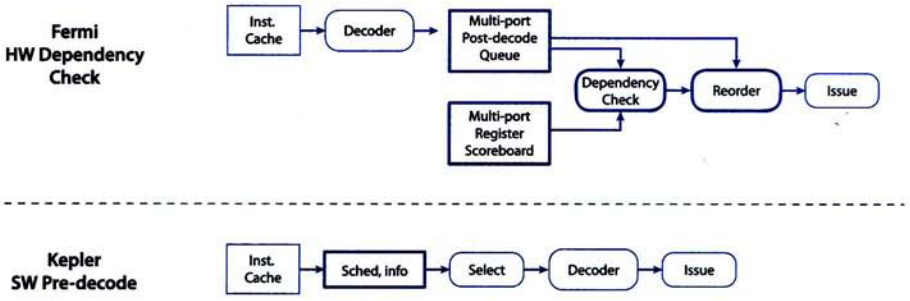


*Рис. 6.7.* Пример планирования работы SMX

*Таблица 6.1.* Количество тактов на мультипроцессор на варп

| Операция | Tesla C870/C1060 | Tesla C2050 |
|---|---|---|
| 32-битная вещественная add, multiply, multiply-add | 4 | 1 |
| 64-битная вещественная add, multiply, multiply-add | 32 | 2 |
| 32-битная целочисленная add, logical operation | 4 | 1 |
| 32-битная целочисленная shift, compare | 4 | 2 |
| 32-битная целочисленная multiply, multiply-add | 4 (24-битная) | 2 |
| 32-битная вещественная reciprocal, reciprocal square root | 16 | 8 |

одна из 32 нитей пойдет по другой ветке кода, то все 32 нити должны будут выполнить обе ветки. Этот эффект называется *дивергентным ветвлением* (divergent branching) и ведет к снижению быстродействия. При этом нити, принадлежащие другим варпам, не оказывают никакого влияния на данный варп. Рассмотрим, как будет выполняться на GPU следующий фрагмент кода:

```
if ( threadIdx.x == 0 )
    x = cos ( y );
```

```
else
    x = sin ( y * 2.0f );
```

Сразу видно, что для первой нити варпа будет выполнена одна ветка, а для всех остальных нитей – другая, т.е. имеет место ветвление внутри варпа. В результате все нити данного варпа выполнят обе ветви ($x = cos(y)$ и $x = sin(y * 2.0f)$), что приведет к снижению быстродействия ядра. Чем больше ветвление внутри варпа, тем медленнее он выполняется из-за необходимости пройти все встречающиеся ветви кода. Знание размера варпа и того, каким образом происходит распределение нитей по варпам, позволяет в ряде случаев снизить ветвление.

Потоковый мультипроцессор для каждого варпа отслеживает готовность данных и на каждой очередной команде выбирает готовый к выполнению варп и выполняет для него одну команду. Для него определяется, какие данные необходимы для следующей команды, после чего выбирается очередной готовый к выполнению варп, для него выполняется одна команда и т.д.

Наличие большого количества выполняемых потоковым мультипроцессором варпов позволяет эффективно покрывать латентность доступа к памяти – пока одни варпы ожидают готовности данных, другие выполняются. Поскольку, например, для устройств compute capability 1.x размер варпа в четыре раза больше числа скалярных процессоров, то на выполнение одной команды всеми нитями варпа нужно четыре такта. Для того чтобы полностью покрыть латентность в 200 тактов, достаточно всего 50 варпов – команда обращения каждого варпа к глобальной памяти займет 4 такта (без учета латентности). Таким образом, когда 50-й варп выполнит свое обращение, данные для первого варпа уже будут готовы. Если обращение к глобальной памяти происходит лишь в каждой четвертой команде, то для полного покрытия латентности в 200 тактов достаточно всего 13 варпов.

Таким образом, потоковый мультипроцессор фактически осуществляет «бесплатное» управление выполнением варпов.

## 6.2. Общие методы оптимизации CUDA-программ

Оптимизация производительности, как правило, сводятся к следующим шагам:

1. Максимальное использование параллелизма задачи.

2. Оптимизация доступа в память.

3. Оптимизация математики.

Для достижения наилучшей производительности, следует попытаться разложить задачу на такие подзадачи, чтобы полностью использовать ресурс параллелизма по данным. В тех участках алгоритма, где параллелизм нарушается (например, потокам необходимо синхронизироваться для разделения данных между собой) возможны два случая:

1. Если потоки принадлежат одному блоку, то они могут использовать разделяемую память для эффективного обмена данными и *__syncthreads()* для синхронизации внутри одного ядра.

2. Если потоки принадлежат разным блокам, то эффективнее использовать 2 разных ядра с промежуточной записью в глобальную память.

При условии, что алгоритм достаточно распараллелен, можно приступить к оптимизации работы с памятью:

1. Оптимизация начинается с минимизации обмена данными между хостом и GPU. В этом могут помочь следующие приемы:

   (а) Помимо функций вида *cudaMemcpy*, возвращающих управление только по завершении копирования, существуют механизмы асинхронного копирования данных *cudaMemcpyAsync*. Асинхронное копирование работает с pinned-памятью (память, которую запрещено выгружать из ОЗУ). Для управления асинхронными вызовами можно применить *cudaEvents*.

   (б) Устройства с compute capability 1.1 и выше имеют отдельный DMA-контроллер, который позволяет копировать данные по PCI-E во время исполнения ядра.

   (в) CUDA stream позволяет задавать различные очереди задач. Например, пусть существует два независимых ядра:

```
__global__ void ka(float *pDst, float *pSrc);
```

```
__global__ void kb(float *pDst, float *pSrc);
```

и их фактические аргументы:

```
float A[]; float B[];        // входные значения на CPU
float *pD_a; float *pD_b; // память на GPU для результата
float *pS_a; float *pS_b; // входные значения на GPU
```

и требуется выполнить две независимые цепочки операций:

$$(pS\_a) \rightarrow ka(pD\_a, pS\_a) \rightarrow (pD\_aA),$$
$$(BpS\_b) \rightarrow kb(pD\_b, pS\_b) \rightarrow (pD\_bB)$$

Эти цепочки необходимо поместить в разные очереди задач, так как это позволит производить копирование данных для ядра *B* параллельно с выполнением ядра *A* и одновременно копирование данных для *A* обратно на CPU будет происходить вместе с выполнением ядра *B*.

(г) Зачастую бывает эффективнее выполнить работу на GPU (даже если при этом прирост минимален, или ядро медленнее аналога на CPU), чем передавать данные по PCI-E и обратно.

2. Оптимизацию обращений в память из ядра необходимо начать с проверки выполнения правил коалесинга (были рассмотрены в гл. 3). Выполнение или невыполнение условий коалесинга может изменять производительность на порядок. Рассмотрим типичные ошибки:

   (а) При работе с двумерными и трехмерными типами данных массив оказывается не выровненным по ширине. Представим, что дана сетка *pDst* размера $1000 \times Height$ (значение *Height* не принципиально), каждый элемент которой есть вещественное число. В ядре CUDA вычисляется индекс для записи в pDst по формуле:

   ```
   tidx = threadIdx.x + blockIdx.x * blockDim.x;
   tidy = threadIdx.y + blockIdx.y * blockDim.y;
   pDst[tidx + tidy * 1000] = 0.0f.
   ```

   Такой простой код вызовет невыполнение условий коалесинга уже на второй строке (в данном случае нарушается правило о выравнивании

адреса по *16 * sizeof(*pDst)*), и производительность сразу упадет на порядок на архитектуре карт серии 8x и 9x.

(б) Использование невыровненных типов данных часто приводит к аналогичным проблемам. Например, если сравнить производительность двух ядер:

```
__global__
void kf3( float3 * pDst ){ pDst[threadIdx.x] = f3(); }

__global__
void kf4( float4 * pDst ){ pDst[threadIdx.x] = f4(); },
```

то окажется, что второе работает на порядок быстрее. Это связано с тем, что правила коалесинга работают с 32-битными, 64-битными и 128-битными структурами данных.

(в) Существуют задачи, в которых необходимо читать элементы по ширине, а записывать – по высоте. Такой задачей, например, является задача транспонирования матрицы. Прямолинейная запись по высоте сразу нарушает все правила коалесинга и вызывает падение производительности на любой архитектуре.

3. Задержка при доступе варпа к глобальной памяти занимает сотни тактов, поэтому необходимо, чтобы на мультипроцессоре исполнялось достаточное количество независимых от этой операции варпов для ее покрытия. Другими словами, во время ожидания одного варпа по инструкции доступа к памяти GPU должен иметь возможность исполнять другие независящие от этой инструкции. При этом нужно иметь в виду, что задержки, связанные с арифметикой, на порядок меньше, и одних этих операций зачастую может быть недостаточно. Таким образом, общим методом получения наилучшей пропускной способности памяти является покрытие задержек путем максимизации количества одновременно исполняемых на мультипроцессоре варпов.

4. При написании ядра важно следить за тем, использует ли компилятор локальную память. Использование локальной памяти происходит в следующих случаях:

(а) Если используется большое количество регистров. Компилятор ограничивает использование регистров (например, числом 32). Если компилятор не может вместить все вычисления в заданное количество регистров, то он выделяет массив глобальной памяти, в который при необходимости сбрасывает данные из регистров или восстанавливает их по мере надобности.

(б) Если в ядре объявить автоматический массив – переменную вида

```
float a[2];
```

и в дальнейшем обращаться к ней по индексу, который приходит из внешних данных (например, индекс можно прочитать из глобальной памяти или получить в результате вычислений, которые компилятор не сможет провести на этапе компиляции), то компилятор вынужден разместить данную переменную в локальной памяти, так как регистры не являются индексируемым видом памяти. В такой ситуации лучше использовать разделяемую память (если она не используется активно) или по возможности переосмыслить алгоритм.

(в) В CUDA нет способа обозначить на какую именно память ссылается передаваемый в __device__-функцию указатель, поэтому компилятор вынужден пытаться получить заключение о типе памяти на основании анализа контекста. В некоторых случаях передача указателя может привести к использованию локальной памяти.

(г) Некоторые математические функции могут приводить к использованию локальной памяти.

В устройствах архитектуры Fermi появилась кэш-память первого уровня, которая также используется для ускорения доступа к локальной памяти. В тех случаях, когда использование локальной памяти предотвратить нельзя, увеличение L1-кэша за счет разделяемой памяти (см. секцию 3.2.1) может улучшить производительность.

5. Разделяемая память требует правильного использования банков.

(а) Просто достигнуть максимальной производительности, если каждый поток обращается в свой банк или все потоки обращаются к одному элементу.

(б) Стоит избегать хаотичного обращения в разделяемую память.

(в) Не следует хранить в разделяемой памяти структуры, поскольку это приведет к банк-конфликтам. Например, в массиве структур float4 последовательные поля x, y, z, w займут 4 банка, что потенциально дает банк-конфликт четвертого порядка. От конфликтов можно избавиться, если добавить смещения:

```
__shared__ float a[512*4];
#define V_X  a[threadIdx.x+512*0]
#define V_Y  a[threadIdx.x+512*1]
#define V_Z  a[threadIdx.x+512*2]
#define V_W  a[threadIdx.x+512*3]
```

Однако, всегда стоит оценивать насколько это увеличит объем математики, ведь теперь на каждое обращение будет выполняться до 4-х MAD-инструкций.

6. При использовании текстурной памяти важно соблюдать локальность при обращении к ней. Это повышает эффективность текстурного кэша.

(а) Использование текстурной памяти в случае, если перекрытия по данным между соседними варпами нет, вряд ли позволит получить более высокую производительность, чем с использованием глобальной памяти. Это оправдано только в том случае, если выполнить условия коалесинга невозможно.

(б) Кроме того, не стоит забывать, что текстурный блок имеет свой конвейер (по отображению адресов, нормализации значений, фильтрации и проч.), а значит обладает большей латентностью по сравнению с прямым обращением в глобальную память.

(в) Использование текстуры и разделяемой памяти может как увеличить производительность, так и уменьшить. Для оценки возможного эффек-

та необходимо учесть роль различных факторов. Например, если попробовать ускорить таким образом алгоритм Non-Local Means, то:

- Загрузка из текстуры в разделяемую память возможна только при использовании *uchar4* элементов, если на *float4* не хватит памяти. Загрузка ничего не стоит по сравнению со временем вычислений.
- При чтении *uchar4* нормализацию значений придется проводить самостоятельно, что добавляет существенный объем инструкций.

7. Работая с константной памятью, важно помнить, что ее максимальная производительность достигается при обращении всеми потоками варпа по одному адресу.

Наконец, если выбрана самая эффективная параллельная реализация алгоритма и налажена оптимальная работа с памятью, то можно приступить к оптимизации арифметики. Необходимо иметь в виду следующее:

1. Разные операции имеют разную пропускную способность, соответственно, это нужно учитывать при анализе приложения. Например, операции *ADD* и *FMAD* – «быстрые» (на Fermi за 1 такт для всего варпа), тогда как операции *LOG* и *RCP* – в несколько раз медленнее. Также есть специальные встроенные функции, которые имеют более высокую пропускную способность, но меньшую точность, например, *__sinf* (точность на 2–3 бита меньше). Существуют также дополнительные опции компилятора, как, например, *-ftz=true*, *-prec-div=false*, *-prec-sqrt=false*. Поэтому можно выбирать между более точным и более быстрым вариантом, в зависимости от поставленных целей;

2. Пропускная способность GPU измеряется в IPC (число инструкций за один такт). Этот показатель доступен в CUDA profiler. Чем ближе он к пиковому значению, тем более эффективно работает устройство. Таким образом, можно определять имеет ли смысл оптимизировать инструкции. Стоит учитывать, что разные инструкции имеют разную пропускную способность и пик IPC нужно рассчитывать в зависимости от набора инструкций в приложении;

3. Одна из основных проблем, связанных с инструкциями, – это *сериализация*. В нормальном режиме каждая инструкция вызывается для всего варпа один

раз. Сериализация возникает, когда инструкция вызывается несколько раз, для каждой нити варпа – тем самым снижается производительность. Сериализация бывает 2 типов: разные пути исполнения и банк-конфликты.

- Любая условная операция (if, switch, do, for, while) может заметно повлиять на общую производительность, так как нити внутри варпа имеют дивергентные пути исполнения. Степень влияния можно оценить довольно просто. Для этого необходимо заменить все условные выражения так, чтобы они выполнялись или не выполнялись для всего варпа. Уменьшить число дивергентных варпов можно сгруппировав нити так, чтобы они шли по одному пути исполнения. При профилировании, счетчик "divergent branch" увеличивается на единицу при каждом ветвлении внутри варпа, и его следует сравнивать со счетчиком "branch", показывающим общее количество ветвлений.
- Конфликты банков были рассмотрены в секциях 3.5.2 и 3.5.3. Простой способ проверить, насколько сильно они влияют на общую производительность – заменить все индексы при обращении к разделяемой памяти на threadIdx.x. Таким образом, гарантируется, что обращения будет без конфликтов банков, при этом, если основная логика программы не поменяется, можно оценить потенциальное ускорение. Для индикации конфликтов банков в устройствах архитектуры Fermi при профилировании используется счетчик "l1 shared bank conflict". Если его величина сравнима со значениями счетчиков "shared load", "shared store" и "instructions issued", то можно ожидать влияния конфликтов банков на производительность. В секции 3.5.3 также был рассмотрен пример устранения конфликтов с помощью добавления к массиву в разделяемой памяти фиктивных данных.

# Глава 7

# Прикладные математические библиотеки

В основе методов решения большинства современных задач, требующих высокопроизводительных вычислений (HPC), лежит одна или несколько технологий из следующего списка:

1. Линейная алгебра для плотных матриц;
2. Линейная алгебра для разреженных матриц;
3. Операции над регулярными сетками;
4. Операции над нерегулярными сетками;
5. Спектральные методы;
6. Взаимодействие частиц;
7. Метод Монте-Карло.

Для того, чтобы GPU играли существенную роль в мире HPC, необходимы не только вычислительный потенциал и перспективная программная модель, но и готовая технологическая экзосистема. По этой причине были разработаны и включены в состав CUDA Toolkit следующие математические библиотеки:

- CUBLAS – линейная алгебра для плотных матриц (1);
- CUSPARSE – линейная алгебра для разреженных матриц (2);
- CUFFT – преобразования Фурье (5);
- CURAND – генераторы псевдо- и квазислучайных чисел (7).

В соответствии с лицензией, данные библиотеки могут быть включены в поставку сторонних приложений. Доступ к исходному коду могут получить участники программы зарегистрированных разработчиков.

Также существует множество качественных свободно распространяемых библиотек сторонних разработчиков, например:

- MAGMA – линейная алгебра для плотных матриц, открытый эквивалент библиотеки CUBLAS, расширенный дополнительными методами;
- CUSP – линейная алгебра и итерационные методы решения СЛАУ для разреженных матриц;
- OpenCurrent – динамика жидких сред.

## 7.1. CUBLAS

CUBLAS реализует программный интерфейс BLAS (Basic Linear Algebra Subprograms – основные операции линейной алгебры над векторами и матрицами) на CUDA для одного GPU.

В соответствии с классическим BLAS на языке Fortran, в CUBLAS многомерные массивы располагаются в памяти *по столбцам* (column-major order), индексирование ведется от 1. Функции BLAS разделены на три уровня (таблица 7.1).

*Таблица 7.1.* Уровни функций BLAS

| **Уровень** | **Вычислительная сложность** | **Примеры функций** |
|---|---|---|
| 1 (векторные операции) | $O(N)$ | AXPY: y := alpha x + y, DOT: dot := (x, y) |
| 2 (матрица-вектор) | $O(N^2)$ | GEMV – умножение матрицы общего вида на вектор |
| 3 (матрица-матрица) | $O(N^3)$ | GEMM – умножение двух матриц общего вида |

Имя любой функции CUBLAS образуется по правилу *cublas<T><func>*, где *T* – литера, определяющая тип данных (*S* – вещественный, одинарная точность, *D* – вещественный, двойная точность, *C* – комплексный, одинарная точность, *Z* – комплексный, двойная точность), *func* – 3–4 литеры классического имени BLAS-функции, например, *cublasDgemm*. Как и вызовы CUDA-ядер, функции CUBLAS являются *асинхронными*.

Начиная с версии CUDA 4.0, в библиотеке CUBLAS появился новый API "v2" (старый API по-прежнему поддерживается). В новом API каждый вызов CUBLAS-функции использует *дескриптор* (handle), связанный с текущим контекстом устройства и потоком исполнения (CUDA Stream). Такой дизайн призван упростить разработку приложений, использующих несколько потоков исполнения или несколько GPU. Все результаты вычислений возвращаются через указатели в аргументах. В новом API функции библиотеки возвращают статус ошибки типа *cublasStatus_t*, тогда как в старом текущий статус ошибки доступен через общую вспомогательную функцию *cublasGetError()*.

Типовая схема использования CUBLAS в приложении имеет вид:

1. Инициализировать дескриптор CUBLAS функцией *cublasCreate*;
2. Выделить необходимую память на GPU для матриц и векторов, загрузить входные данные;
3. Вызвать необходимую последовательность функций CUBLAS;
4. Выгрузить результаты вычислений из памяти GPU в память основной системы, освободить память;
5. Освободить дескриптор CUBLAS функцией *cublasDestroy*.

### 7.1.1. Пример: *matmul*

Для проверки результата и сравнения производительности в примерах *matmul* главы 3 используется функция *cublasSgemm* библиотеки CUBLAS:

```
#include <cublas_v2.h>
#include <stdio.h>
//...
// Создать дескриптор CUBLAS.
cublasHandle_t handle;
cublasStatus_t cberr = cublasCreate_v2(&handle);
if (cberr != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "Cannot create cublas handle: %d\n", cberr);
    return 1;
}
```

```
// Выполнить умножение матриц cdev := adev * bdev на GPU.
float alpha = 1.0, beta = 0.0;
cberr = cublasSgemm_v2(
    handle, CUBLAS_OP_T, CUBLAS_OP_T, n, n, n,
    &alpha, adev, n, bdev, n, &beta, cdev, n);
if (cberr != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "Error launching cublasSgemm_v2: %d\n", cberr);
    return 1;
}

// Ожидать завершения операции.
cuerr = cudaDeviceSynchronize();
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot synchronize kernel: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}

// Удалить дескриптор CUBLAS.
cberr = cublasDestroy_v2(handle);
if (cberr != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "Cannot destroy cublas handle: %d\n", cberr);
    return 1;
}
```

### 7.1.2. Пример: степенной метод

В вычислительной математике часто возникает необходимость оценить спектр (собственные значения) линейного оператора. Это связанно с тем, что многие априорные оценки ошибок алгоритмов опираются на оценки спектров. Более того, нередко подобные оценки в алгоритме приходится выполнять постоянно. Например, такая необходимость возникает при решении нелинейных гиперболических задач, где шаг по времени обратно пропорционален величине максимального по модулю собственного значения матрицы системы, которая меняется на каждом временном слое.

Рассмотрим матрицу $A$. Как известно, вектор $a$ называется собственным вектором матрицы $A$, если выполнено равенство $Aa = \lambda a$. При этом $\lambda$ называется

собственным числом [5]. Если матрица $A$ имеет размеры $N \times N$, то у нее есть $N$ собственных чисел (возможно, комплексных). При этом данные собственные числа не обязательно различны.

В общем случае, задача нахождения максимального по модулю собственного числа не очень проста. Поэтому мы введем ряд предположений, которые позволят использовать компактный алгоритм.

Предположим, что матрица $A$ имеет $N$ собственных числел и полную систему из собственных векторов. Это означает, что любой вектор можно разложить в линейную комбинацию собственных векторов:

$$x = C_1 a_1 + C_2 a_2 + ... + C_N a_N = \sum_{i=0}^{N} C_i a_i, \tag{7.1}$$

где $Aa_i = \lambda_i a_i, i = 1..N$ - собственные вектора. Также предположим, что максимальное по модулю собственное число единственно: $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq ... \geq |\lambda_N|$. Данные предположения позволяют построить более простой и компактный алгоритм, чем для матрицы общего вида.

Умножим уравнение 7.1 на матрицу $A$ слева:

$$Ax = \sum_{i=0}^{N} C_i A a_i = \sum_{i=0}^{N} C_i \lambda_i a_i. \tag{7.2}$$

Умножая $k$ раз, получаем:

$$A^k x = \sum_{i=0}^{N} C_i A^k a_i = \sum_{i=0}^{N} C_i \lambda_i^k a_i, \tag{7.3}$$

где $A^k = \underbrace{A \cdot A \cdot ... \cdot A}_{k \text{ раз}}$. Преобразовав 7.3, получаем:

$$\begin{aligned} A^k x = \lambda_1^k \left( C_1 a_1 + C_2 \left( \frac{\lambda_2}{\lambda_1} \right)^k a_2 + \ldots + C_N \left( \frac{\lambda_N}{\lambda_1} \right)^k a_N \right) = \\ = \lambda_1^k \sum_{i=0}^{N} C_i \left( \frac{\lambda_i}{\lambda_1} \right)^k a_i. \end{aligned} \tag{7.4}$$

Поскольку $\lambda_1$ – самое большое по модулю число, то все члены суммы в 7.4, кроме первого, стремятся к нулю при $k \to \infty$. Следовательно, по сравнению с первым членом суммы остальные уменьшаются при росте $k$, и вектор $A^k x$ близок к собственному вектору $C_1 \lambda_1^k a_1$.

Таким образом, для нахождения максимального по модулю собственного значения может быть использован следующий алгоритм:

1. выбрать вектор начального приближения $x_0$;
2. построить следующее приближение по формулам: $y_{n+1} = Ax_n$; $x_{n+1} = \alpha_{n+1} y_{n+1}$, где $\alpha_{n+1}$ выбирается из условия $|x_{n+1}| = 1$;
3. продолжать процесс, пока не будет выполнено заданное условие сходимости, например:
$$\|x_{n+1} - x_n\| < \varepsilon;$$
4. последнее приближение $x_{n+1} = x$ является собственным вектором, а собственное значение вычисляется по формуле:
$$x^T Ax \approx \lambda_1 x^T x = \lambda_1,$$
поскольку $\|x\| = 1$.

Данный алгоритм называется *степенным методом* нахождения максимального собственного значения [6]. Можно показать, что степенной метод применим и при отсутствии предположения о полноте системы собственных векторов. Однако, для корректной работы алгоритма в любом случае требуется наличие единственного максимального по модую собственного числа. Согласно теореме Фробениуса – Перрона [7], для этого достаточно, чтобы матрица имела только положительные элементы: $A : a_{ij} > 0$.

Степенной метод легко реализовать с помощью прикладных библиотек CUDA: CUBLAS – для операций с матрицами и векторами и CURAND – для генерации случайной положительной матрицы и начального приближения.

```
#include <cuda.h>
#include <curand.h>
```

```
#include <cublas_v2.h>
#include <stdio.h>
#define CUDA_CALL(x) \
    do { cudaError_t err = x; if (( err ) != cudaSuccess ) { \
        printf ("Error \"%s\" at %s :%d \n" , cudaGetErrorString(err), \
        __FILE__ , __LINE__ ) ; return -1; \
    }} while (0);

#define CURAND_CALL(x) do { if (( x ) != CURAND_STATUS_SUCCESS ) { \
    printf ("Error at %s :%d \n" , __FILE__ , __LINE__ ) ; \
    return -1; }} while (0);

#define CUBLAS_CALL(x) do { if (( x ) != CUBLAS_STATUS_SUCCESS ) { \
    printf ("Error at %s :%d \n" , __FILE__ , __LINE__ ) ; \
    return -1; }} while (0);

// Функция вывода матрицы на экран
// ddata - указатель на матрицу, расположенную в памяти GPU
int PrintDenseMatrix(char *name, float *ddata, int n, int m)
{
    float *data;
    data = (float*) malloc(n*m*sizeof(float));
    CUDA_CALL(cudaMemcpy(data, ddata, n*m*sizeof(float),
        cudaMemcpyDeviceToHost));
    printf("Dense matrix %s:", name);
    int ln, lm;
    ln = (n > 10) ? 10 : n;
    lm = (m > 10) ? 10 : m;
    for (int i=0; i<ln; i++)
    {
        for (int j=0; j<lm; j++)
            printf("%.4f ", data[j*n + i]);
        if (m != lm)
            printf("... ");
        printf("\n");
    }
    if (ln != n)
        printf("...\n");
    free(data);
    return 0;
}
```

В памяти GPU необходимо выделить массивы для хранения матрицы и двух векторов. Матрица и один из векторов заполняются случайными числами, с помощью библиотеки CURAND:

```
// Функция генерации матрицы и начального условия
int Init (float **dA, float **dx, float **dy, int n)
{
   printf("Data generation: ");
   // Выделение памяти для A, x_0 и вектора y (необходим в алгоритме)
   CUDA_CALL(cudaMalloc((void**) dA, n*n*sizeof(float)));
   CUDA_CALL(cudaMalloc((void**) dx, n*sizeof(float)));
   CUDA_CALL(cudaMalloc((void**) dy, n*sizeof(float)));

   // Генерация матрицы и вектора
   curandGenerator_t gen;
   CURAND_CALL(curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT));
   CURAND_CALL(
   curandSetPseudoRandomGeneratorSeed(gen, (unsigned long long)time(NULL)));

   CURAND_CALL(curandGenerateUniform(gen, dA[0], n*n));
   CURAND_CALL(curandGenerateUniform(gen, dx[0], n));
   printf("Done\n");
   return 0;
}
```

**Основная часть кода содержит цикл итераций степенного метода: вычисление нормы вектора, умножение вектора на число, сложение векторов (с коэффициентами), а также умножение матрицы на вектор. Время работы алгоритма измеряется с использованием *cudaEvent*:**

```
// Функция, выполняющая итерации алгоритма
int CublasIterations (cublasHandle_t cublasHandle,
                      float *A, float *x, float *y, int n,
                      float eps, int DEBUG)
{
    // Используем cudaEvent для замеров времени
    cudaEvent_t start, stop;
    CUDA_CALL(cudaEventCreate(&start));
    CUDA_CALL(cudaEventCreate(&stop));
    float a;
    float norm;
    int loop = 0;
    // Приводим начальное приближени к виду norm2(x) = 1
    CUBLAS_CALL(cublasSnrm2(cublasHandle, n, x, 1, &norm));
    a = 1.0f / norm;
    CUBLAS_CALL(cublasSscal(cublasHandle, n, &a, x, 1));
    float alpha = 1.0f;
    float beta = 0.0f;
    float diff = 100.0f;
```

```
    CUDA_CALL(cudaEventRecord(start, 0));
    while ((diff > eps) && (loop < 1000))
    {
        loop++;
        CUBLAS_CALL(cublasSgemv(cublasHandle, CUBLAS_OP_N,
            n, n, &alpha, A, n, x, 1, &beta, y, 1));
        CUBLAS_CALL(cublasSnrm2(cublasHandle, n, y, 1, &norm));
        a = 1.0f / norm;
        CUBLAS_CALL(cublasSscal(cublasHandle, n, &a, y, 1));

        // На данном этапе в y находится (n+1) итерация процесса,
        // а в x - n-ая
        // Вычислим норму разности приближений
        // x[i] = x[i] - y[i]:
        a = -1.0f;
        CUBLAS_CALL(cublasSaxpy(cublasHandle, n, &a, y, 1, x, 1));
        CUBLAS_CALL(cublasSnrm2(cublasHandle, n, x, 1, &diff));
        if (DEBUG)
        {
            printf("%d-th iteration. Difference norm %f\n", loop, diff);
        }
        // Копируем (n+1)-ую итерацию в x
        CUBLAS_CALL(cublasScopy(cublasHandle, n, y, 1, x, 1));
    }
    CUDA_CALL(cudaEventRecord(stop, 0));
    CUDA_CALL(cudaEventSynchronize(stop));

    if (loop < 1000)
    {
        printf("Process converged in %d iterations.\n", loop);
        printf("Residuals norm is %f\n", diff);
    }
    else
    {
        printf("Process didn't converge in 1000 iterations.\n");
        printf("Residuals norm is %f\n", diff);
    }

    // Вычисление затраченного времени на итерации
    float elapsedTime;
    CUDA_CALL(cudaEventElapsedTime(&elapsedTime, start, stop));
    CUDA_CALL(cudaEventDestroy(start));
    CUDA_CALL(cudaEventDestroy(stop));
    printf("Elapsed time: %.3f ms\n", elapsedTime);
    return 0;
}
```

Несмотря на то, что алгоритм должен сходиться, в программе установлено дополнительное ограничение на число итераций для защиты от зацикливания, в случае если ошибка округления превысит $\varepsilon$.

В последней части кода определена функция, вычисляющая собственное число, соответствующее найденному собственному вектору $x$ по формуле $\lambda \approx \frac{x^T Ax}{x^t x}$:

```
// Функция, вычисляющая по приближенному значению собственного векотора
// собственное значение
int EigValEstimate(cublasHandle_t cublasHandle,
                   float *A, float *x, float *y, int n, int DEBUG)
{
    // y = Ax
    float alpha = 1.0f;
    float beta = 0.0f;
    CUBLAS_CALL(cublasSgemv(cublasHandle, CUBLAS_OP_N, n, n,
        &alpha, A, n, x, 1, &beta, y, 1));
    float num, denom;
    CUBLAS_CALL(cublasSdot(cublasHandle, n, x, 1, y, 1, &num));
    CUBLAS_CALL(cublasSdot(cublasHandle, n, x, 1, x, 1, &denom));
    printf("Approximate eigenvalue is %f", num/denom);
    PrintDenseMatrix("eigenvector", x, n, 1);
    return 0;
}

int main (int argc, char** argv)
{
    float *dA, *dx, *dy;

    // Обработка входных параметров
    if (argc != 4)
    {
        printf("Error. Eig usage: ./eig n, eps, DEBUG\n");
        return -1;
    }
    int n = atoi(argv[1]);
    float eps = atof(argv[2]);
    int DEBUG = atoi(argv[3]);
    int res = 0;

    // Вызов функции инициализации
    res = Init(&dA, &dx, &dy, n);
    if (res < 0)
        return res;

    // Вывод матрицы
```

```
    res = PrintDenseMatrix("A", dA, n, n);
    if (res < 0)
        return res;

    // Вывод вектора
    res = PrintDenseMatrix("x_0", dx, n, 1);
    if (res < 0)
        return res;
    printf("\n");

    // Создание контекста CUBLAS
    cublasHandle_t cublasHandle;
    CUBLAS_CALL(cublasCreate(&cublasHandle));

    // Вызов функции, реализующей алгоритм
    res = CublasIterations(cublasHandle, dA, dx, dy, n, eps, DEBUG);
    if (res < 0)
        return res;

    // Оценка собственного значения
    res = EigValEstimate(cublasHandle, dA, dx, dy, n, DEBUG);
    if (res < 0)
        return res;

    // Освобождение дескриптора
    CUBLAS_CALL(cublasDestroy(cublasHandle));

    return 0;
}
```

## 7.2. CUSPARSE

CUSPARSE реализует основные операции линейной алгебры для разреженных векторов и матриц (sparse matrices and vectors). Функции библиотеки имеют интерфейс для C и C++, поддерживается индексация элементов как с нуля, так и с единицы.

*Разреженными* называются матрицы или векторы с преимущественно[1] нулевыми элементами. Принцип, в соответствии с которым хранится разреженный вектор или матрица, прост: хранить только ненулевые значения и информацию об их

[1] Есть разные понимания «преимущественно». Например: «имеет малый процент ненулевых элементов», или «для матрицы $N \times N$ количество ненулевых элементов есть $O(N)$».

положении в матрице. Разреженный вектор представляется парой массивов. В первом массиве находятся все ненулевые значения из соответствующего плотного массива. Второй целочисленный массив содержит индексы этих элементов. Существует множество форматов для хранения матриц; среди них поддерживаются только:

1. Плотный формат (Dense format);
2. Координатный формат (Coordinate format, COO);
3. Строчный разреженный формат (Compressed Sparse Row Format, CSR);
4. Столбцовый разреженный формат (Compressed Sparse Column Format, CSC).

Все функции библиотеки реализованы для вещественных и комплексных типов одинарной и двойной точности и делятся на 4 группы:

1. Операции над разреженными векторами и плотными векторами: сложение векторов, скалярное произведение, поворот Гивенса, сборка (gather) и разборка (scatter) элементов вектора.
2. Операции над разреженными матрицами и плотными векторами:
   - *csrmv*: $y = \alpha op(A) \cdot x + \beta \cdot y$, где $op(A) = A$, либо $op(A) = A^T$, либо $op(A) = A^H$. $A$ – матрица размером $m \times n$ в формате упакованных разреженных строк (CSR). $\alpha$ и $\beta$ – скаляры. $x$ и $y$ – вектора в плотном формате.
   - *csrsv_analysis* – выполняет анализ при решении разреженной треугольной системы: $op(A) \cdot y = \alpha \cdot x$, где $op(A) = A$, либо $op(A) = A^T$, либо $op(A) = A^H$. $A$ – матрица размером $m \times n$ в формате упакованных разреженных строк (CSR).
   - *csrsv_solve* – выполняет решение разреженной треугольной системы, описанной в предыдущем примере.
3. Операции над разреженными векторами и множествами плотных векторов:

$$C = \alpha \cdot op(A) \cdot B + \beta \cdot C,$$

где $op(A) = A$, либо $op(A) = A^T$, либо $op(A) = A^H$. $\alpha$ и $\beta$ – скаляры. $B$ и $C$ – матрицы, хранящиеся в памяти *по столбцам* (column-major order). $A$ – матрица размером $m \times k$ в формате упакованных разреженных строк (CSR).

4. Функции для преобразования одного формата матриц в другой.

Имя любой функции CUSPARSE образуется по правилу *cusparse<T><func>*, где *T* – литера, определяющая тип данных (*S* – вещественный, одинарная точность, *D* – вещественный, двойная точность, *C* – комплексный, одинарная точность, *Z* – комплексный, двойная точность), *func* – литеры имени функции; например *cusparseScsrmv*.

Типовая схема использования CUSPARSE в приложении выглядит следующим образом:

1. Выделить память на GPU. Инициализировать память данными для вектора или матрицы в одном из поддерживаемых форматов.
2. Инициализировать библиотеку.
3. Выполнить преобразования над данными.
4. Освободить память на GPU и само преобразование.

Функции CUSPARSE можно вызывать из программ на языке Fortran.

### 7.2.1. Пример: решение треугольной линейной системы уравнений

*Треугольная матрица* – квадратная матрица, в которой все элементы ниже или выше главной диагонали равны нулю. *Треугольная линейная система уравнений* – линейная система уравнений $Ax = b$, где $A$ – треугольная матрица.

Ключевые шаги программы:

1. Создание разреженной матрицы в формате COO:

```
/* |1.0            |
   |2.0 3.0        |
   |4.0 5.0 6.0    |
   |7.0 8.0 9.0 1.0| */
```

```
int n=4, nnz=10;
cooRowIndexHostPtr = (int *)   malloc(
    nnz*sizeof(cooRowIndexHostPtr[0]));
cooColIndexHostPtr = (int *)   malloc(
    nnz*sizeof(cooColIndexHostPtr[0]));
cooValHostPtr      = (double *)malloc(
    nnz*sizeof(cooValHostPtr[0]));

cooRowIndexHostPtr[0]=0; cooColIndexHostPtr[0]=0;
cooValHostPtr[0]=1.0;

cooRowIndexHostPtr[1]=1; cooColIndexHostPtr[1]=0;
cooValHostPtr[1]=2.0;
cooRowIndexHostPtr[2]=1; cooColIndexHostPtr[2]=1;
cooValHostPtr[2]=3.0;

cooRowIndexHostPtr[3]=2; cooColIndexHostPtr[3]=0;
cooValHostPtr[3]=4.0;
cooRowIndexHostPtr[4]=2; cooColIndexHostPtr[4]=1;
cooValHostPtr[4]=5.0;
cooRowIndexHostPtr[5]=2; cooColIndexHostPtr[5]=2;
cooValHostPtr[5]=6.0;

cooRowIndexHostPtr[6]=3; cooColIndexHostPtr[6]=0;
cooValHostPtr[6]=7.0;
cooRowIndexHostPtr[7]=3; cooColIndexHostPtr[7]=1;
cooValHostPtr[7]=8.0;
cooRowIndexHostPtr[8]=3; cooColIndexHostPtr[8]=2;
cooValHostPtr[8]=9.0;
cooRowIndexHostPtr[9]=3; cooColIndexHostPtr[9]=3;
cooValHostPtr[9]=1.0;
```

2. Создание плотного вектора:

```
/* y   = [10.0 20.0 30.0 40.0 | 0.0 0.0 0.0 0.0] */
yHostPtr    = (double *)malloc(2*n       *sizeof(yHostPtr[0]));

yHostPtr[0] = 10.0;
yHostPtr[1] = 20.0;
yHostPtr[2] = 30.0;
yHostPtr[3] = 40.0;
yHostPtr[4] = 0.0;
yHostPtr[5] = 0.0;
yHostPtr[6] = 0.0;
yHostPtr[7] = 0.0;
```

3. Копирование вектора и матрицы в память на GPU:

```
cudaStat1 = cudaMalloc((void**)&cooRowIndex,
                       nnz*sizeof(cooRowIndex[0]));
cudaStat2 = cudaMalloc((void**)&cooColIndex,
                       nnz*sizeof(cooColIndex[0]));
cudaStat3 = cudaMalloc((void**)&cooVal,
                       nnz*sizeof(cooVal[0]));
cudaStat4 = cudaMalloc((void**)&y,
                       2*n*sizeof(y[0]));

cudaStat1 = cudaMemcpy(cooRowIndex, cooRowIndexHostPtr,
                       (size_t)(nnz*sizeof(cooRowIndex[0])),
                       cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(cooColIndex, cooColIndexHostPtr,
                       (size_t)(nnz*sizeof(cooColIndex[0])),
                       cudaMemcpyHostToDevice);
cudaStat3 = cudaMemcpy(cooVal,      cooValHostPtr,
                       (size_t)(nnz*sizeof(cooVal[0])),
                       cudaMemcpyHostToDevice);
cudaStat4 = cudaMemcpy(y,           yHostPtr,
                       (size_t)(2*n*sizeof(y[0])),
                       cudaMemcpyHostToDevice);
```

4. Инициализация CUSPARSE:

```
status = cusparseCreate(&handle);
```

5. Создание дескриптора матрицы (где указывается необходимое свойство *CUSPARSE_MATRIX_TYPE_TRIANGULAR*):

```
status = cusparseCreateMatDescr(&descra);
cusparseSetMatType(descra,CUSPARSE_MATRIX_TYPE_TRIANGULAR);
cusparseSetMatIndexBase(descra,CUSPARSE_INDEX_BASE_ZERO);
```

6. Преобразование матрицы из формата COO в CSR:

```
cudaStat1 = cudaMalloc((void**)&csrRowPtr,
                       (n+1)*sizeof(csrRowPtr[0]));
status= cusparseXcoo2csr(handle, cooRowIndex, nnz, n,
                         csrRowPtr, CUSPARSE_INDEX_BASE_ZERO);
```

7. Анализ и решение треугольной системы уравнений:

```
cusparseSolveAnalysisInfo_t info;
status = cusparseCreateSolveAnalysisInfo(&info);
status = cusparseDcsrsv_analysis(
    handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n,
    descra, cooVal, csrRowPtr, cooColIndex, info);
status = cusparseDcsrsv_solve(
    handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, 7.0,
    descra, cooVal, csrRowPtr, cooColIndex, info, &y[0], &y[n]);
status = cudaMemcpy(yHostPtr, y, (size_t)(2*n*sizeof(y[0])),
                    cudaMemcpyDeviceToHost);
cusparseDestroySolveAnalysisInfo(info);
```

8. Проверка результатов:

```
if (FLOATS_EQ(yHostPtr[0], 10.0) &&
    FLOATS_EQ(yHostPtr[1], 20.0) &&
    FLOATS_EQ(yHostPtr[2], 30.0) &&
    FLOATS_EQ(yHostPtr[3], 40.0) &&
    FLOATS_EQ(yHostPtr[4], 70.0) &&
    FLOATS_EQ(yHostPtr[5], 0.0) &&
    FLOATS_EQ(yHostPtr[6], -11.6667) &&
    FLOATS_EQ(yHostPtr[7], -105.0))
{
    CLEANUP("example test PASSED");
    return EXIT_SUCCESS;
} else {
    CLEANUP("example test FAILED");
    return EXIT_FAILURE;
}
```

В качестве упражнения читатель может переписать код для проверки результатов: умножить исходную разреженную матрицу $A$ на плотный вектор решения $x$ и убедиться, что результат не отличается от исходного плотного вектора $b$ в правой части.

## 7.3. CUFFT

Библиотека CUFFT реализует прямое и обратное быстрое дискретное преобразование Фурье (ДПФ):

$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi\left(x\frac{n}{N}\right)}$$

$$f(n) = \frac{1}{N}\sum_{n=0}^{N-1} F(x)e^{j2\pi\left(x\frac{n}{N}\right)}$$

Простейшая реализация ДПФ в виде произведения матрицы на вектор будет иметь алгоритмическую сложность $O(N^2)$. Для уменьшения сложности используется алгоритм Кули-Тьюки (Cooley-Tukey). (В популярной open-source библиотеке FFTW сложность используемых алгоритмов – $O(NlogN)$). Параллельная реализация преобразований основана на принципе «разделяй и властвуй» (divide and conquer)[8]. CUFFT оптимизирован для преобразований, размерности которых выражаются как $2^a \cdot 3^b \cdot 5^c \cdot 7^d$. Для преобразований других размеров используется алгоритм Блуштейна (Bluestein), реализованный на основе алгоритма Кули-Тьюки [9].

Интерфейс и формат данных CUFFT во многом схож с FFTW, но отличается в некоторых деталях. Для исключения отличий в CUFFT предусмотрен режим совместимости (FFTW Compatibility Mode). Ниже перечислены основные свойства реализации CUFFT:

- одно-, дву- и трехмерные вещественные и комплексные преобразования;
- одинарная и двойная точность;
- одномерное преобразование до 128 млн элементов одинарной точности и до 64 млн элементов двойной точности (точное ограничение на конкретном GPU определяется размером доступной глобальной памяти);
- поддержка CUDA Streams (Streamed CUFFT Transforms);
- in-place и out-of-place преобразования для вещественных и комплексных данных;
- преобразования с двойной точностью можно выполнить только на GPU с поддержкой двойной точности (GT200 и более поздние версии);
- ненормализованный вывод: $IFFT(FFT(A)) = len(A) * A$ (результат последовательного применения прямого и обратного преобразования – исходный вектор, умноженный на длину).

Для того, чтобы использовать библиотеку, необходимо включить заголовочный файл "cufft.h". Библиотека состоит из объявлений, типов данных и функции обработки данных этих типов.

Основные типы данных библиотеки CUFFT:

1. *cufftHandle* – план CUFFT (аналог *fftw_plan*), используется для адаптивного выбора наилучшего алгоритма и многократного использования известной оптимальной настройки;
2. *cufftResult* – результат вызова функции;
3. *cufftType* – тип преобразования (поддерживаются комплексные и вещественные преобразования).

Основные функции библиотеки CUFFT:

1. *cufftPlan** – функции создания планов, принимают на вход дескриптор плана, размерности и тип преобразования; *cufftDestroy* – освобождает план;
2. *cufftExec** – функции для выполнения преобразования, принимают на вход дескриптор плана, входные и выходные данные, возвращают *cufftResult*;
3. *cufftSetStream* связывает CUDA stream с CUFFT-планом. На вход принимает план и stream, а возвращает *cufftResult*.

FFTW реализует «расширенный интерфейс» (advanced interface), который дает возможность преобразовать несколько массивов одновременно. Его аналогом в CUFFT является функция *cufftPlanMany*.

Типовая схема использования CUFFT в приложении выглядит следующим образом:

1. Выделить память на GPU;
2. Создать и настроить преобразование (размер, тип, ...) и план;
3. Выполнить преобразование необходимое число раз, используя план и данные;

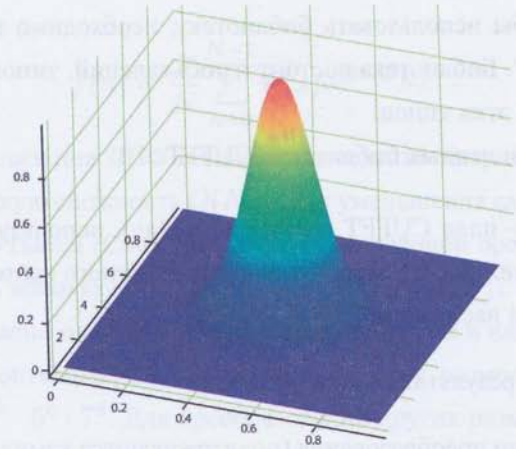


*Рис. 7.1.* Точное решение уравнения Пуассона

4. Освободить память на GPU и преобразование.

CUFFT может быть также использован в программах на языке Fortran.

### 7.3.1. Пример: решение уравнение Пуассона

Уравнение Пуассона описывает многие явления в физике. Оно встречается в задачах магнитостатики, электростатики, гидро- и газодинамики, описывает стационарное поле температуры. Поставим задачу с уравнением Пуассона следующего вида:

$$\begin{cases} \Delta u(p) = \rho(p), p \in \Omega = (x,y), 0 \le x,y \le 1, \\ \rho(x,y) = \frac{s(x,y)-2\sigma^2}{\sigma^4} \exp^{-\frac{s(x,y)}{2\sigma^2}}. \end{cases} \tag{7.5}$$

Пусть на границе области $\Omega$ заданы периодические граничные условия, тогда уравнение имеет точное решение (рис. 7.1):

$$u_0(x,y) = \exp^{\frac{s(x,y)}{2\sigma^2}}. \tag{7.6}$$

Решение уравнение Пуассона можно получить с помощью преобразования Фурье. Пусть $N$ – число разбиений отрезка $[0..1]$, $h = \frac{1}{N}$ – шаг сетки одинаковый по обоим осям X и Y. Тогда

$$\overline{\rho} = \frac{1}{N^2} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \rho(x_k, y_j) W^{nk+mj}, \tag{7.7}$$

где $W = \exp^{i\frac{2\pi}{N}}$. Можно убедиться, что при незначительных допущениях:

$$\overline{u}(n,m) = \overline{\rho}(n,m) h^2 (W^{-n} + W^n + W^{-m} + W^m - 4)^{-1}), \tag{7.8}$$

$$u(x_k, y_j) = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \overline{u}(n,m) W^{nk+mj}. \tag{7.9}$$

В данном примере прямое (7.7) и обратное преобразование (7.9) Фурье будет выполнено с помощью библиотек FFTW и CUFFT, а вычисления в Фурье-пространстве (7.8) – с помощью отдельного CUDA-ядра.

Сначала решим задачу с помощью FFTW. Читатель может сравнить эту реализацию с аналогичной для CUFFT и убедиться в схожести интерфейсов двух библиотек.

```
double s(double x, double y)
{
    return (x - 0.5) * (x - 0.5) + (y - 0.5) * (y - 0.5);
}

double rho(double x, double y)
{
    double ss = s(x, y);
    return (ss - 2 * sigma2) * exp(-ss / (2 * sigma2)) / sigma4;
}

double wave_num2(int i, int j, int n)
{
    if (!i && !j) return 1.0;

    double wn1 = i < n / 2 ? i : i - n;
    double wn2 = j < n / 2 ? j : j - n;

    return -4 * M_PI * M_PI * (wn1 * wn1 + wn2 * wn2);
}

int fft_cpu(int n, double* u)
{
```

```
    const double h = 1.0 / n;

    fftw_complex* v = (fftw_complex*)fftw_malloc(
        sizeof(fftw_complex) * (n / 2 + 1) * n);

    fftw_plan forward = fftw_plan_dft_r2c_2d(n, n, u, v, FFTW_ESTIMATE);
    fftw_plan inverse = fftw_plan_dft_c2r_2d(n, n, v, u, FFTW_ESTIMATE);

    fftw_execute(forward);

    for (int j = 0; j < n; j++)
        for (int i = 0; i < (n / 2 + 1); i++)
            v[j * (n / 2 + 1) + i][0] /= wave_num2(i, j, n);

    fftw_execute(inverse);

    double shift = u[0];
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            u[j * n + i] -= shift;
            u[j * n + i] /= n * n;
        }
    }

    fftw_destroy_plan(forward);
    fftw_destroy_plan(inverse);

    fftw_free(v);

    return 0;
}
```

Массив $u$ в памяти CPU содержит значения функции $\rho$ в точках сетки. Память для массивов $u$ и $v$ эффективнее выделить с помощью выравнивающего аллокатора fftw_malloc – это делает возможным использование инструкций SIMD. Число элементов в результате прямого преобразования из вещественного пространства вдвое меньше, чем в исходном массиве, поскольку вторая половина будет содержать комплексно-сопряженные значения. Для прямого и обратного преобразования создаются в начале и освобождаются в конце соответствующие планы. Важно, что планы настроены для преобразования из вещественных чисел в комплексные, без лишних промежуточных преобразований. Между вызовом прямого и обратного преобразования Фурье, согласно (7.8), производятся вычисления в спектральном

пространстве. Поскольку решение уравнение Пуассона определено с точностью до константы, то для сравнения результата работы FFTW и CUFFT удобно эту константу в конце привести к 0.

Теперь рассмотрим реализацию с помощью CUFFT:

```
__global__ void solve_transformed(int n,
    const cufftDoubleComplex* rhs, cufftDoubleComplex* u)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;

    int m = n / 2 + 1;
    if (i < m && j < n)
    {
        cufftDoubleComplex t = rhs[j * m + i];

        cufftDoubleReal w = cufftDoubleReal(M_PI) * (i < n/2 ? i : i - n);
        cufftDoubleReal v = cufftDoubleReal(M_PI) * (j < n/2 ? j : j - n);
        cufftDoubleReal s = (!i && !j) ? 1 : -4 * (w * w + v * v);

        t.x /= s;
        t.y /= s;

        u[j * m + i] = t;
    }
}

__global__ void scale_and_shift(int n, cufftDoubleReal* u, double shift)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;

    if (i < n && j < n)
        u[j * n + i] = (u[j * n + i] - shift) / (n * n);
}

int fft_gpu(int n, double* u)
{
    dim3 blk (32, 2);
    dim3 rgrd((n + blk.x - 1) / blk.x, (n + blk.y - 1) / blk.y);
    dim3 cgrd((n/2 + blk.x) / blk.x, (n + blk.y - 1) / blk.y);
    double shift;

    cufftDoubleComplex* v = NULL;
    cudaError_t cuerr = cudaMalloc((void**)&v, n * (n / 2 + 1) *
```

```
        sizeof(cufftDoubleComplex));
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot create gpu memory buffer for v: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }

    set_rhs<<<rgrd, blk>>>(n, u);

    cufftHandle forward, inverse;
    cufftPlan2d(&forward, n, n, CUFFT_D2Z);
    cufftPlan2d(&inverse, n, n, CUFFT_Z2D);
    cufftExecD2Z(forward, u, v);

    solve_transformed<<<cgrd, blk>>>(n, v, v);
    cufftExecZ2D(inverse, v, u);
    cudaMemcpy(&shift, u, sizeof(double), cudaMemcpyDeviceToHost);
    scale_and_shift<<<rgrd, blk>>>(n, u, shift);
    cudaFree(v);

    cufftDestroy(forward);
    cufftDestroy(inverse);

    return 0;
}
```

Здесь массив $u$ расположен в памяти GPU. Так же, как и для FFTW, создаются и освобождаются планы. Важное отличие состоит в том, что вычисления в спектральном пространстве (7.8) необходимо выполнить над данными в памяти GPU – для этого запускается отдельное CUDA-ядро. Еще одним CUDA-ядром выполняется приведение константы к 0.

Далее рассмотрим вариант решения, демонстрирующий взаимную совместимость CUFFT и FFTW:

```
int fft_gpu_compatibility(int n, double* u)
{
    const double h = 1.0 / n;

    dim3 blk (32, 2);
    dim3 rgrd((n + blk.x - 1) / blk.x, (n + blk.y - 1) / blk.y);
    dim3 cgrd((n/2 + blk.x) / blk.x, (n + blk.y - 1) / blk.y);

    fftw_complex* v = (fftw_complex*)fftw_malloc(
```

```
        sizeof(fftw_complex) * (n / 2 + 1) * n);
    fftw_plan forward = fftw_plan_dft_r2c_2d(n, n, u, v, FFTW_ESTIMATE);

    fftw_execute(forward);

    for (int j = 0; j < n; j++)
        for (int i = 0; i < (n / 2 + 1); i++)
            v[j * (n / 2 + 1) + i][0] /= wave_num2(i, j, n);

    double* u_gpu_dev = NULL;
    cufftDoubleComplex* v_gpu_dev = NULL;
    cufftHandle inverse;
    cufftPlan2d(&inverse, n, n, CUFFT_Z2D);
    cufftSetCompatibilityMode(inverse, CUFFT_COMPATIBILITY_FFTW_ALL);
    size_t size_of_u = sizeof(double) * n * n;
    size_t size_of_v = sizeof(cufftDoubleComplex) * (n / 2 + 1) * n;
    cudaMalloc((void**)&u_gpu_dev, size_of_u);
    cudaMalloc((void**)&v_gpu_dev, size_of_v);
    cudaMemcpy(v_gpu_dev, v, size_of_v, cudaMemcpyHostToDevice);
    cufftExecZ2D(inverse, v_gpu_dev, u_gpu_dev);

    double shift;
    cudaMemcpy(&shift, u_gpu_dev, sizeof(double), cudaMemcpyDeviceToHost);
    scale_and_shift<<<rgrd, blk>>>(n, u_gpu_dev, shift);

    cudaMemcpy(u, u_gpu_dev, size_of_u, cudaMemcpyDeviceToHost);

    cudaFree(u_gpu_dev);
    cudaFree(v_gpu_dev);

    cufftDestroy(inverse);

    fftw_free(v);

    fftw_destroy_plan(forward);

    return 0;
}
```

Массив $u$ располагается в памяти CPU. Для него с помощью FFTW производится прямое преобразование Фурье и (7.8). Затем создается план CUFFT со свойством *CUFFT_COMPATIBILITY_FFTW_ALL*. Далее результат прямого преобразования и (7.8) копируется из массива $v$ в память GPU, обратное преобразование выполняется с помощью CUFFT, и возвращается в массив $u$.

В качестве упражнения читатель может реализовать прямое преобразования с помощью CUFFT, а обратное – функциями FFTW.

## 7.4. CURAND

CURAND – это библиотека генераторов (псевдо)случайных чисел. На GPU поддержание приемлемого статистического качества и периода представляет некоторую проблему, поскольку генераторы, работающие в отдельных нитях, должны быть достаточно независимы. Поэтому для GPU необходимы специальные генераторы.

Последовательность неслучайных чисел называется *псевдослучайной*, если она удовлетворяет большинству свойств случайной последовательности, но генерируется детерминированным алгоритмом. Последовательность неслучайных чисел называется *квазислучайной*, если она генерируется детерминированным алгоритмом и равномерно заполняет $n$-мерное пространство; таким образом, ее можно использовать в методах Монте-Карло вместо случайной (при этом метод может работать лучше, чем со случайной последовательностью).

С помощью CURAND можно генерировать последовательности псевдо- и квазислучайных чисел:

- *CURAND_RNG_PSEUDO_XORWOW* – генератор псевдослучайных чисел на основе алгоритма XORWOW[10]. Генератор выдает последовательности из $2^{31} - 1$ целых чисел, или $2^{64} - 1$ пар целых чисел, или $2^{96} - 1$ троек целых чисел.

- 4 типа генераторов на основе алгоритма Соболя[11] для генерации квазислучайных чисел (каждый тип генерирует последовательности с измерениями до 20000):
  - *CURAND_RNG_QUASI_SOBOL32*
  - *CURAND_RNG_QUASI_SCRAMBLED_SOBOL32*
  - *CURAND_RNG_QUASI_SOBOL64*
  - *CURAND_RNG_QUASI_SCRAMBLED_SOBOL64*

Библиотека CURAND имеет два интерфейса: для хоста ("curand.h") и для GPU-ядер ("curand_kernel.h"). В первом случае вызовы функций CURAND должны являться частью кода хост-программы, и случайные числа могут генерироваться как на хосте, так и на GPU. Если генератор работает на GPU, то все необходимые вычисления производятся на GPU, результат будет помещен в глобальную память GPU и доступен для использования в CUDA-ядрах или для копирования в память хоста. Если генератор работает на хосте, то все вычисления производятся на хосте, и результат помещается в память хоста.

Типовая схема использования хост-интерфейса CURAND выглядит следующим образом:

1. Создать генератор с помощью *curandCreateGenerator()*.
2. Задать необходимые свойства генератора, например, начальное состояние: *curandSetPseudoRandomGeneratorSeed()*.
3. Выделить память на GPU с помощью *cudaMalloc*.
4. Запустить генерацию случайных чисел с необходимым распределением:
   - равномерное: *curandGenerateUniform()*;
   - нормальное: *curandGenerateNormal()*;
   - log-нормальное: *curandGenerateLogNormal()*.
5. Использовать результаты работы генератора в приложении.
6. При необходимости сгенерировать дополнительные данные новыми вызовами генератора.
7. *curandDestroyGenerator()* – освободить генератор.

Вызовы функций CURAND возвращают статус ошибки типа *curandStatus_t*.

Второй интерфейс позволяет генерировать случайные числа непосредственно в CUDA-ядрах, в месте их использования. Ниже приведена одна из возможных схем:

1. Выделить место в глобальной памяти GPU для массива состояний генераторов каждой нити *curandState*.

2. Создать и запустить ядро, инициализирующее *curandState* для каждой нити.

3. Создать и запустить ядро, использующее случайные числа, генерируемые функцией *curand(curandState)*.

4. Освободить память.

Инициализацию и использование генератора может быть более эффективно разместить в разных ядрах, поскольку инициализация требует больше регистров и локальной памяти.

### 7.4.1. Пример: генерация показаний распределенных датчиков

Пусть имеется некоторое приложение, обрабатывающее показания распределенных на местности датчиков, например, решающее задачу из пункта 8.1.7 Если необходимо проверить работу приложения на различных входных данных, поступающих от нескольких миллионов датчиков, то целесообразно использовать генератор из GPU API библиотеки CURAND:

```
const uint number_to_print = 15;
const uint num_days = 1024, num_sites = 16;
const uint thread_dim = 256, block_dim = num_days * num_sites / thread_dim;
__global__ void setup_random_states(curandState *state, int n)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    if (id < n)
        curand_init(1234, id, 0, &state[id]);
}
__global__ void generate_numbers(
    curandState *state, int* numbers, int numbers_size, uint range_max) {
    int id = threadIdx.x + blockIdx.x * blockDim.x;

    if (id >= numbers_size) return;

    curandState localState = state[id];
    numbers[id] = curand(&localState) % (2 * range_max) - range_max;
    state[id] = localState;
}
struct measurement_to_triple
{
    int m_num_days, m_num_sites;
    measurement_to_triple(int num_days, int num_sites) :
```

```
        m_num_days(num_days), m_num_sites(num_sites) { }
    __host__ __device__
    thrust::tuple<int, int, int> operator()(thrust::tuple<int, int> x)
    {
        int index, measurement;
        thrust::tie(index, measurement) = x;
        int day = index / m_num_sites, site = index % m_num_sites;
        return thrust::make_tuple(day, site, measurement);
    }
};
struct measurement_is_negative
{
    __host__ __device__
    bool operator()(thrust::tuple<int, int, int> x)
    {
        return thrust::get<2>(x) < 0; // i.e. measurement < 0
    }
};
struct is_positive
{
    __host__ __device__
    bool operator()(int v) { return v > 0; }
};

void collect_data_curand(
    const uint num_days, const uint num_sites,
    device_vector<int>& day, device_vector<int>& site,
    device_vector<int>& measurement)
{
    int total_size = num_days * num_sites;
    device_vector<int> measurement_n(total_size),
        day_n(total_size), site_n(total_size);

    curandState *devStates;
    cudaMalloc((void**)&devStates, total_size * sizeof(curandState));
    setup_random_states<<<block_dim, thread_dim>>>(devStates, total_size);
    int *measurement_n_ptr = thrust::raw_pointer_cast(&measurement_n[0]);
    generate_numbers<<<block_dim, thread_dim>>>(
        devStates, measurement_n_ptr, measurement_n.size(), 50);

    thrust::counting_iterator<int> cnt_iter(0);
    thrust::transform(
        thrust::make_zip_iterator(
            thrust::make_tuple(cnt_iter, measurement_n.begin())),
        thrust::make_zip_iterator(
            thrust::make_tuple(cnt_iter + measurement_n.size(),
```

```
            measurement_n.end())),
        thrust::make_zip_iterator(
            thrust::make_tuple(day_n.begin(), site_n.begin(),
            measurement_n.begin())),
        measurement_to_triple(num_days, num_sites));

    int size_n = thrust::count_if(measurement_n.begin(),
        measurement_n.end(), is_positive());
    day.resize(size_n); site.resize(size_n); measurement.resize(size_n);
    thrust::remove_copy_if(
        thrust::make_zip_iterator(
            thrust::make_tuple(day_n.begin(), site_n.begin(),
            measurement_n.begin())),
        thrust::make_zip_iterator(
            thrust::make_tuple(day_n.end(), site_n.end(),
            measurement_n.end())),
        thrust::make_zip_iterator(
            thrust::make_tuple(day.begin(), site.begin(),
            measurement.begin())),
        measurement_is_negative());

    cudaFree(devStates);
}
```

Промежуточные данные хранятся в векторах *measurement_n*, *day_n* и *site_n* в памяти GPU. Работа начинается с инициализации состояния генераторов *devStates* в ядре *setup_random_states*. С помощью ядра *generate_numbers* генерируется последовательность измерений. Значения не отсортированы и принимают значения на отрезке $[-50, 50]$. Для передачи указателя на память вектора *measurement_n* используется *thrust::raw_pointer_cast*. С помощью функтора *measurement_to_triple* из каждого значения датчика образуется тройка день – участок – измерение. Для большего значения индекса соответствующая пара день – участок оказывается больше. Из получившейся последовательности трансформацией *thrust::remove_copy_if* удаляются все тройки, значения измерений которых отрицательны. Оставшиеся тройки копируются в три выходных последовательности.

В качестве упражнения читатель может реализовать генерацию данных, используя хост-API библиотеки CURAND.