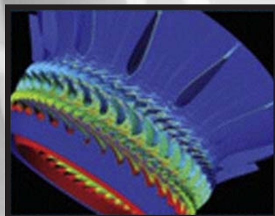
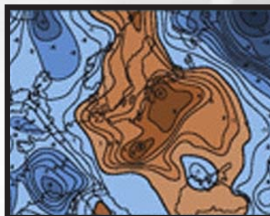
Г. Рутм, М. Фатика

CUDA Fortran

для инженеров и научных работников





Рекомендации по эффективному программированию

MK
MORGAN KAUFMANN
PGI®

ДМК
издательство

УДК 004.3'144:004.383.5CUDA
ББК 32.973.26-04
Р60

Р60 Рутш Г., Фатика М.

CUDA Fortran для инженеров и научных работников. Рекомендации по эффективному программированию на языке CUDA Fortran / пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2014. – 364 с.: ил.

ISBN 978-5-97060-065-8

Fortran – один из важнейших языков программирования для высокопроизводительных вычислений, для которого было разработано множество популярных пакетов программ для решения вычислительных задач. Корпорация NVIDIA совместно с The Portland Group (PGI) разработали набор расширений к языку Fortran, которые позволяют использовать технологию CUDA на графических картах NVIDIA для ускорения вычислений.

Книга демонстрирует всю мощь и гибкость этого расширенного языка для создания высокопроизводительных вычислений. Не требуя никаких предварительных познаний в области параллельного программирования авторы скрупулезно шаг за шагом раскрывают основы создания высокопроизводительных параллельных приложений, попутно поясняя важные архитектурные детали современного графического процессора – ускорителя вычислений.

Издание предназначено для инженеров, научных работников, программистов, в также будет полезно студентам вузов соответствующих специальностей.

Original English language edition published by Gregory Ruetsch/NVIDIA Corporation and Massimiliano Fatica/NVIDIA Corporation. Published by Elsevier Inc, 225 Wyman Street, Waltham, MA 02451, USA. Copyright © 2014 Gregory Ruetsch/NVIDIA Corporation and Massimiliano Fatica/NVIDIA Corporation. Published by Elsevier Inc.. Russian-language edition copyright © 2014 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0-12-416970-8 (англ.)
©2014 Gregory Ruetsch/NVIDIA Corporation and Massimiliano Fatica/NVIDIA Corporation. Published by Elsevier Inc. All rights reserved.

ISBN 978-5-97060-065-8 (рус.)
© Оформление, перевод на русский язык, издание, ДМК Пресс, 2014

ОГЛАВЛЕНИЕ

Благодарности 10

Предисловие 11

ЧАСТЬ I
Программирование на CUDA Fortran 13

Глава 1. Введение 14

1.1. Краткая история вычислений на GPU 14
1.2. Параллельные вычисления 16
1.3. Основные идеи 17
1.3.1. Первая программа на CUDA Fortran 18
1.3.2. Обобщение на большие массивы 22
1.3.3. Многомерные массивы 25
1.4. Определение возможностей и ограничений оборудования с поддержкой CUDA 27
1.5. Обработка ошибок 38
1.6. Компиляция программы на CUDA Fortran 39
1.6.1. Раздельная компиляция 43

Глава 2. Измерение производительности и метрики производительности 48

2.1. Измерение времени выполнения ядра 48
2.1.1. Синхронизация хоста и устройства и таймеры CPU 49
2.1.2. Хронометраж с помощью событий CUDA 50
2.1.3. Командный профилировщик 51
2.1.4. Профилировщик nvprof 53
2.2. Ядра, производительность которых, ограничена вычислениями, пропускной способностью памяти и задержкой 54
2.3. Пропускная способность памяти 58
2.3.1. Теоретически максимальная пропускная способность 58
2.3.2. Эффективная пропускная способность 60

Глава 3. Оптимизация.. 63

3.1. Передача данных между хостом и устройством................ 63

3.1.1. Зафиксированная область памяти 64

3.1.2. Объединение мелких операций передачи в один пакет 69

3.1.3. Асинхронная передача данных (дополнительная тема)..........72

3.2. Память устройства 83

3.2.1. Объявление данных в коде, выполняемом на устройстве 85

3.2.2. Объединенный доступ к глобальной памяти 85

3.2.3. Текстурная память...99

3.2.4. Локальная память.. 105

3.2.5. Константная память 108

3.3. Внутрикристалльная память... 112

3.3.1. L1-кэш.. 112

3.3.2. Регистры... 113

3.3.3. Разделяемая память 115

3.4. Пример оптимизации работы с памятью: транспонирование матрицы 122

3.4.1. Недогрузка разделов (дополнительная тема)...................... 128

3.5. Конфигурация выполнения 133

3.5.1. Параллелизм на уровне потоков.. 133

3.5.2. Параллелизм на уровне команд... 137

3.6. Оптимизация команд 140

3.6.1. Встроенные функции устройства 141

3.6.2. Флаги компилятора.. 142

3.6.3. Расходящиеся варпы 143

3.7. Директивы генерации ядра из цикла 144

3.7.1. Редукция в CUF-ядрах 147

3.7.2. Потоки CUDA в CUF-ядрах 147

3.7.3. Параллелизм на уровне команд в CUF-ядрах...................... 148

Глава 4. Программирование компьютера с несколькими GPU 150

4.1. Средства CUDA для работы с несколькими GPU 150

4.1.1. Связь между равноправными устройствами....................... 152

4.1.2. Прямая передача данных между равноправными устройствами.. 157

4.1.3. Транспонирование матрицы с применением равноправного доступа... 168

4.2. Программирование нескольких GPU с применением библиотеки MPI 177

4.2.1. Сопоставление устройств рангам MPI 179

4.2.2. Транспонирование матрицы с применением MPI 186

4.2.3. Транспонирование матрицы с применением MPI, поддерживающей GPU... 188

ЧАСТЬ II
Примеры задач 191

Глава 5. Метод Монте-Карло 192
5.1. Библиотека CURAND 193
5.2. Вычисление π с помощью CUF-ядер 198
5.2.1. Стандарт IEEE-754 (дополнительная тема) 202
5.3. Вычисление π с помощью ядер редукции 205
5.3.1. Редукция с атомарными блокировками (дополнительная тема) 211
5.4. Точность суммирования 213
5.5. Опционное ценообразование 220

Глава 6. Метод конечных разностей 229
6.1. Девятиточечный шаблон конечно-разностной схемы для вычисления первой производной 229
6.1.1. Повторное использование данных и разделяемая память 231
6.1.2. Ядро производной по *x* 232
6.1.3. Производные по *y* и *z* 237
6.1.4. Неравномерные сетки 242
6.2. Двумерное уравнение Лапласа 246

Глава 7. Приложения быстрого преобразования Фурье 254
7.1. Библиотека CUFFT 254
7.2. Спектральное дифференцирование 263
7.3. Свертка 267
7.4. Решение уравнения Пуассона 276

ЧАСТЬ III
Приложение 283

Приложение A. Технические характеристики Tesla 284

Приложение B. Управление системой и окружением 287
B.1. Переменные окружения 287
B.1.1. Общие переменные окружения 287
B.1.2. Командный профилировщик 288
B.1.3. JIT-компиляция 288

B.2. Интерфейс управления системой nvidia-smi 289
B.2.1. Включение и выключение режима ECC................................290
B.2.2. Режим вычислений 292
B.2.3. Инерционный режим..293

Приложение C. Вызов CUDA C из CUDA Fortran..... 295

C.1. Вызовы библиотеки, написанной на CUDA C................... 295
C.2. Вызов написанной пользователем функции на CUDA C... 298

Приложение D. Исходный код 300

D.1. Текстурная память 300
D.2. Транспонирование матрицы 304
D.3. Параллелизм на уровне потоков и команд 311
D.4. Программирование с использованием нескольких GPU 315
D.4.1. Транспонирование с применением равноправного доступа к памяти..316
D.4.2. Транспонирование с применением библиотеки MPI для передачи данных между хостами 322
D.4.3. Транспонирование с применением библиотеки MPI для передачи данных между устройствами 327
D.5. Программирование метода конечных разностей 332
D.6. Решение уравнения Пуассона спектральным методом ... 352

Литература... 357

Предметный указатель 359

ГЛАВА 3
Оптимизация

В предыдущей главе мы обсудили, как можно использовать данные хронометража для определения того, что именно является лимитирующим фактором при выполнении ядра. Производительность многих научных и инженерных программ лимитирована пропускной способностью памяти, именно поэтому мы посвятили большую часть этой, довольно длинной, главы оптимизации доступа к памяти. В устройствах с поддержкой CUDA есть много разных типов памяти, и, если мы хотим, чтобы программа работала эффективно, то должны научиться использовать эту память должным образом.

Операции передачи данных можно отнести к одной из двух широких категорий: передача данных между хостом и устройством и между различными видами памяти внутри устройства. Мы начнем с оптимизации передачи данных между хостом и устройством. Затем мы обсудим различные типы памяти внутри устройства и эффективную работу с ними. Различные приемы оптимизации мы будем иллюстрировать на примере ядра для транспонирования матрицы.

Помимо оптимизации доступа к памяти, мы в этой главе обсудим также факторы, влияющие на выбор той конфигурации выполнения, при которой оборудование используется максимально эффективно. Наконец, мы обсудим оптимизацию команд.

3.1. Передача данных между хостом и устройством

Максимальная скорость передачи данных между памятью устройства и GPU гораздо выше (к примеру, на устройстве NVIDIA Tesla K20 она составляет 208 ГБ/с), чем максимальная скорость передачи между памятью хоста и памятью устройства (16 ГБ/с на шине PCIe x16 Gen3 и 8 ГБ/с на шине PCIe x16 Gen2). Поэтому для достижения наивысшей общей производительности приложения важно по возмож-

ности минимизировать количество операций передачи данных между хостом и устройством, а в тех случаях, когда без этого не обойтись, оптимизировать их.

При написании нового или переносе существующего приложения на CUDA Fortran обычно в ядра преобразуются несколько критических участков кода. Если эти участки изолированы, то потребуется передавать данные с хоста и обратно, а общая производительность будет ограничена этими операциями. На этой стадии полезно сделать оценку производительности при наличии операций передачи и без них. Общее время вместе с передачей данных можно считать точной оценкой текущей производительности программы, а время без учета передачи показывает, какой производительности можно добиться, если перенести на устройство больше кода. В этот момент не имеет смысла тратить время на оптимизацию передачи данных между хостом и устройством, потому что чем больше кода преобразуется в ядра, тем меньше останется операций передачи промежуточных данных. Разумеется, сколько-то данных передавать придется всегда, и делать это нужно максимально эффективно, но заниматься оптимизацией тех операций передачи данных, которые в конечном итоге будут устранены, – бессмысленная потеря времени.

Могут существовать операции, при выполнении которых на устройстве не наблюдается никакого ускорения. Если при выполнении операции на хосте необходимы дополнительные передачи данных между хостом и устройством, то, возможно, удастся получить выигрыш, выполнив операцию целиком на устройстве.

Бывают также случаи, когда передачи данных между хостом и устройством можно избежать. Можно создать промежуточные структуры данных в памяти устройства, обработать их там же и уничтожить, так ни разу и не скопировав в память хоста.

До сих пор мы говорили, что при любой возможности следует избегать передачи данных между хостом и устройством. А в оставшейся части главы расскажем, как эффективно осуществить такую передачу, если без нее не обойтись. Мы обсудим зафиксированную область памяти хоста, объединение мелких операций передачи в один пакет и асинхронную передачу данных.

3.1.1. Зафиксированная область памяти

Для размещения переменных на хосте по умолчанию выделяется выгружаемая память. Такую память можно выгрузить на диск и тем самым позволить программе использовать больше памяти, чем фи-

зически имеется в системе. При передаче данных между хостом и устройством подсистема прямого доступа к памяти (ПДП) на GPU должна обращаться к блокированной, или *зафиксированной* памяти хоста. Закрепленную память нельзя выгрузить, поэтому она всегда доступна для операций передачи. Чтобы передать данные из выгружаемой памяти хоста на GPU, операционная система хоста сначала выделяет временный закрепленный буфер, копирует в него данные, а затем уже передает их на устройство, как показано на рис. 3.1. Размер буфера в закрепленной памяти может быть меньше, чем размер области данных в выгружаемой памяти, и в этом случае передача производится в несколько этапов. Точно так же буферы в закрепленной памяти используются при передаче от устройства хосту. Накладных расходов на копирование между выгружаемой и закрепленной памятью можно избежать, если сразу объявить, что массив на хосте должен быть размещен в закрепленной памяти.

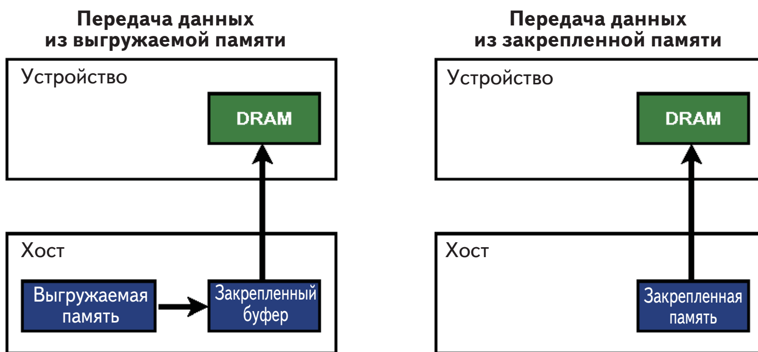


Рис. 3.1. Передача данных от хоста устройству из выгружаемой памяти (слева) и из закрепленной памяти (справа). В случае выгружаемой памяти данные предварительно копируются во временный буфер в закрепленной памяти хоста. Если изначально использовать закрепленную память, как на рисунке справа, то эта дополнительная операция копирования исключается.

В CUDA Fortran для обозначения закрепленной памяти используется квалификатор переменной `pinned`, причем такая память должна быть объявлена также выделяемой с помощью квалификатора `allocatable`. Не исключено, что оператор `allocate` не сможет выделить закрепленную память, и в этом случае будет предпринята попытка получить выгружаемую память. В следующем примере продемонстрировано выделение закрепленной памяти с проверкой оши-

бок и показано, какого ускорения можно ожидать от закрепленной памяти.

```
 1 program BandwidthTest
 2
 3   use cudafor
 4   implicit none
 5
 6   integer, parameter :: nElements = 4*1024*1024
 7
 8   ! массивы в памяти хоста
 9   real(4) :: a_pageable(nElements), b_pageable(nElements)
10   real(4), allocatable, pinned :: a_pinned (:), b_pinned (:)
11
12   ! массивы в памяти устройства
13   real(4), device :: a_d(nElements)
14
15   ! события для хронометража
16   type (cudaEvent) :: startEvent, stopEvent
17
18   ! разное
19   type (cudaDeviceProp) :: prop
20   real(4) :: time
21   integer :: istat, i
22   logical :: pinnedFlag
23
24   ! выделить и инициализировать
25   do i = 1, nElements
26      a_pageable(i) = i
27   end do
28   b_pageable = 0.0
29
30   allocate(a_pinned(nElements), b_pinned(nElements), &
31         STAT=istat, PINNED= pinnedFlag)
32   if (istat /= 0) then
33      write (*,*) 'Ошибка при выделении a_pinned/b_pinned'
34      pinnedFlag = .false.
35   else
36     if (.not. pinnedFlag) write (*,*) 'Не удалось выделить закрепленную память'
37   end if
38
39   if (pinnedFlag) then
40      a_pinned = a_pageable
41      b_pinned = 0.0
42   endif
43
44   istat = cudaEventCreate(startEvent)
45   istat = cudaEventCreate(stopEvent)
46
47   ! вывести сведения об устройстве и размер передачи
48   istat = cudaGetDeviceProperties(prop, 0)
```

```
49
50   write (*,*)
51   write (*,*) 'Устройство: ', trim(prop%name)
52   write (*,*) 'Размер передачи (МБ): ', 4* nElements /1024./1024.
53
54   ! передача данных из выгружаемой памяти
55   write (*,*)
56   write (*,*) 'Передача для выгружаемой памяти'
57
58   istat = cudaEventRecord(startEvent, 0)
59   a_d = a_pageable
60   istat = cudaEventRecord(stopEvent, 0)
61   istat = cudaEventSynchronize(stopEvent)
62
63   istat = cudaEventElapsedTime(time, startEvent, stopEvent)
64   write (*,*) ' Пропускная способность хост-устройство (ГБ/с): ', &
65         nElements*4/time/1.e+6
66
67   istat = cudaEventRecord(startEvent, 0)
68   b_pageable = a_d
69   istat = cudaEventRecord(stopEvent, 0)
70   istat = cudaEventSynchronize(stopEvent)
71
72   istat = cudaEventElapsedTime(time, startEvent, stopEvent)
73   write (*,*) ' Пропускная способность устройство-хост (ГБ/с): ', &
74         nElements*4/time/1.e+6
75
76   if (any(a_pageable /= b_pageable )) &
77      write (*,*) '*** Ошибка при передаче из выгружаемой памяти ***'
78
79    ! передача данных из закрепленной памяти
80   if (pinnedFlag) then
81      write (*,*)
82      write (*,*) 'Передача для закрепленной памяти'
83
84   istat = cudaEventRecord(startEvent, 0)
85   a_d = a_pinned
86   istat = cudaEventRecord(stopEvent, 0)
87   istat = cudaEventSynchronize(stopEvent)
88
89   istat = cudaEventElapsedTime(time, startEvent, stopEvent)
90   write (*,*) ' Пропускная способность хост-устройство (ГБ/с): ', &
91         nElements*4/time/1.e+6
92
93   istat = cudaEventRecord(startEvent, 0)
94   b_pinned = a_d
95   istat = cudaEventRecord(stopEvent, 0)
96   istat = cudaEventSynchronize(stopEvent)
97
98   istat = cudaEventElapsedTime(time, startEvent, stopEvent)
99   write (*,*) ' Пропускная способность устройство-хост (ГБ/с): ', &
```

```
100         nElements*4/time/1.e+6
101
102   if (any(a_pinned /= b_pinned )) &
103      write (*,*) '*** Ошибка при передаче из закрепленной памяти ***'
104   end if
105
106   write (*,*)
107
108   ! очистка
109   if (allocated(a_pinned )) deallocate(a_pinned)
110   if (allocated(b_pinned )) deallocate(b_pinned)
111   istat = cudaEventDestroy(startEvent)
112   istat = cudaEventDestroy(stopEvent)
113
114 end program BandwidthTest
```

Выделение закрепленной памяти производится в строке 30 путем задания необязательных аргументов с ключевыми словами `STAT` и `PINNED`, которые можно проверить и узнать, была ли операция выделения успешной и, если да, то какая память выделена: закрепленная или выгружаемая. Это делается в строках 32–37.

Скорость передачи данных может зависеть как от типа хоста, так и от GPU. Например, в системе Intel Xeon E5540 с картой Tesla K20 эта программа дает такие результаты:

```
Устройство: Tesla K20
Размер передачи (МБ): 16.00000

Передача для выгружаемой памяти
  Пропускная способность хост-устройство (ГБ/с): 1.659565
  Пропускная способность устройство-хост (ГБ/с): 1.593377

Передача для закрепленной памяти
  Пропускная способность хост-устройство (ГБ/с): 5.745055
  Пропускная способность устройство-хост (ГБ/с): 6.566322
```

тогда как в системе Intel Xeon E5-2667, также с картой Tesla K20, имеем:

```
Устройство: Tesla K20
Размер передачи (МБ): 16.00000

Передача для выгружаемой памяти
  Пропускная способность хост-устройство (ГБ/с): 3.251782
  Пропускная способность устройство-хост (ГБ/с): 3.301395

Передача для закрепленной памяти
  Пропускная способность хост-устройство (ГБ/с): 6.213710
  Пропускная способность устройство-хост (ГБ/с): 6.608200
```

Скорость передачи для закрепленной памяти в обеих системах примерно одинакова. Что же касается скорости передачи в случае выгружаемой памяти на хосте, то она сильно зависит от хост-системы из-за неявного копирования из выгружаемой памяти в закрепленный буфер на стороне хоста.

Чтобы проверить, используется ли при передаче данных между хостом и устройством закрепленная память, мы можем воспользоваться командным профилировщиком, задав в его конфигурационном файле параметр `memtransferhostmemtype`. Например, профилирование программы `BandwidthTest` дает:

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla K20
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff69b0066e8b8
method,gputime, cputime, occupancy, memtransferhostmemtype
method=[ memcpyHtoD ] gputime=[ 9018.912 ] cputime=[ 9937.000 ]
       memtransferhostmemtype=[ 0 ]
method=[ memcpyDtoH ] gputime=[ 9216.160 ] cputime=[ 10160.000 ]
       memtransferhostmemtype=[ 0 ]
method=[ memcpyHtoD ] gputime=[ 2786.464 ] cputime=[ 3127.991 ]
       memtransferhostmemtype=[ 1 ]
method=[ memcpyDtoH ] gputime=[ 2501.312 ] cputime=[ 2555.000 ]
       memtransferhostmemtype=[ 1 ]
```

где значение 0 параметра `memtransferhostmemtype` означает, что память выгружаемая, а значение 1 – что закрепленная.

Злоупотреблять закрепленной памятью не следует, так как ее чрезмерное использование можно привести к снижению общей производительности системы. Что считать злоупотреблением, заранее сказать трудно, поэтому, как и в случае любой оптимизации, следует тестировать комбинацию приложения и системы, в которой оно работает, чтобы определить оптимальные параметры.

3.1.2. Объединение мелких операций передачи в один пакет

Накладные расходы сопровождают любую операцию передачи данных между хостом и устройством – неважно, используется при этом выгружаемая или закрепленная память. Если передается немного данных, то доля накладных расходом может оказаться велика, поэтому объединение мелких операций передачи в одну может дать выигрыш.

Получить представление о том, как объединять несколько операций передачи данных, поможет запуск программы из раздела 3.1.1 для массивов разных размеров. На рис. 3.2 и 3.3 показаны скорости передачи данных в случае выгружаемой и закрепленной памяти на двух системах, описанных в разделе 3.1.1, при размере передачи варьирующейся от нескольких килобайтов до почти гигабайта. Если выполняется несколько операций передачи с размером в крутой части кривой, то их пакетирование может привести к существенному снижению общего времени передачи.

3.1.2.1. Явная передача с помощью cudaMemcpy()

CUDA Fortran может неявно разбивать операции передачи данных, записанные с помощью операторов присваивания, на несколько операций. В последних версиях компиляторов вероятность такого развития событий заметно снижена, но возможность все-таки осталась. (Сколько операций передачи было произведено для одного оператора присваивания, можно узнать с помощью командного профилировщика.) Чтобы избежать этого, мы можем явно определить единственную операцию передачи данных в непрерывной области с помощью функции `cudaMemcpy()`. Например, неявную передачу в строке 59 можно было бы заменить таким кодом:

```
istat = cudaMemcpy(a_d, a_pageable, nElements)
```

Аргументами функции `cudaMemcpy()` являются конечный массив, исходный массив и количество передаваемых элементов[1]. Поскольку CUDA Fortran – строго типизированный язык, нет нужды задавать направление передачи. Компилятор способен самостоятельно определить, где находятся данные, указанные в первых двух аргументах, по наличию или отсутствию квалификатора `device` в объявлении и выполнить соответствующую операцию передачи. Но при желании можно задать и четвертый необязательный аргумент, определяющий направление передачи; он может принимать значения `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`. При задании четвертого аргумента компилятор игнорирует типы переменных, указанных в первых двух аргументах. В таком случае под количеством элементов понимается количество элементов в исходном массиве.

---

[1] В третьем аргументе задается количество элементов массива, а не количество передаваемых байтов, как в функции `cudaMemcpy()` для CUDA C.

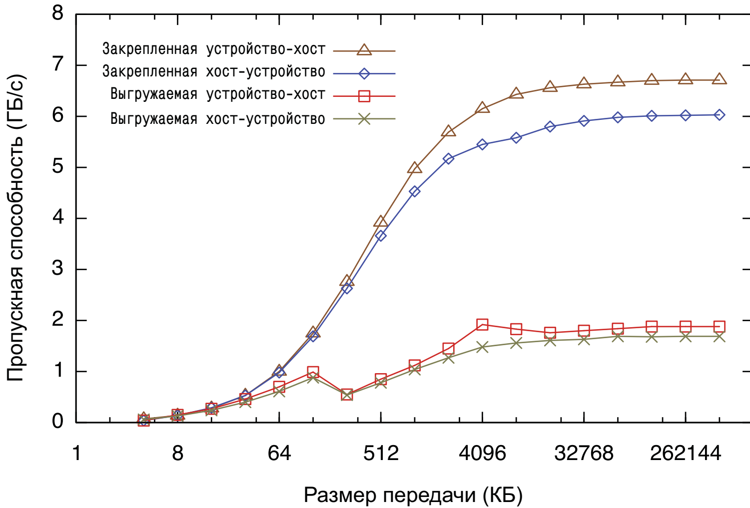


Рис. 3.2. Зависимость пропускной способности передачи хост–устройство и устройство–хост для выгружаемой и закрепленной памяти в системе Intel Xeon E5440 с картой Tesla K20

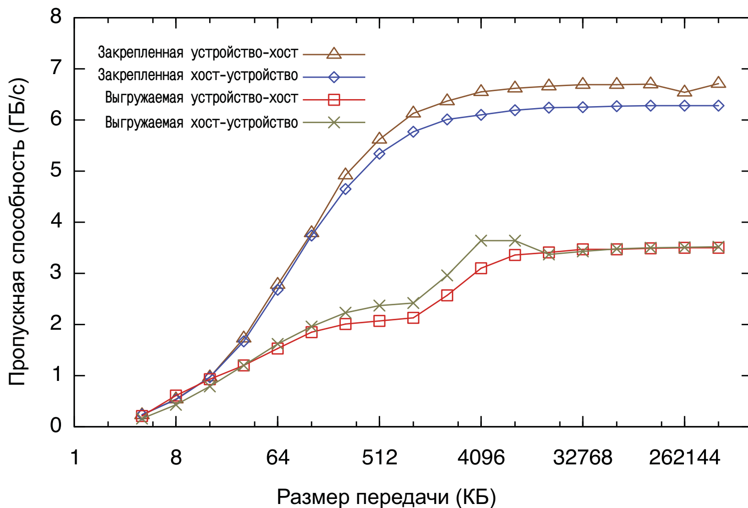


Рис. 3.3. Зависимость пропускной способности передачи хост–устройство и устройство–хост для выгружаемой и закрепленной памяти в системе Intel Xeon E5-2667 с картой Tesla K20

Операторы присваивания в CUDA Fortran можно использовать для передачи секций массива между устройством и хостом, например:

```
a_d(n1_l:n1_u, n2_l:n2_u) = a(n1_l:n1_u, n2_l:n2_u)
```

В общем случае такие операции разбиваются на несколько отдельных передач. Более эффективно выполнять их с помощью функции `cudaMemcpy2D()`. В следующем примере показано, как ту же самую операцию передачи секции массива выполнить посредством `cudaMemcpy2D()`:

```
istat = cudaMemcpy2D(a_d(n1_l, n2_l), n, &
                     a(n1_l, n2_l), n, &
                     n1_u-n1_l+1, n2_u-n2_l+1)
```

Первый и третий аргументы – это первые элементы конечного и исходного массива соответственно. Второй и четвертый аргументы – первые размерности этих массивов, которые мы считаем равными `n`, а последние два аргумента – размер подмассива, заданный в виде количества элементов по каждому измерению. Существует аналогичная функция `cudaMemcpy3D()` для передачи секций трехмерных массивов.

3.1.3. Асинхронная передача данных (дополнительная тема)

Операция передачи данных в любом направлении, будь то с помощью операторов присваивания или функции `cudaMemcpy()`, является блокирующей, то есть управление возвращается потоку хоста только после завершения передачи. Функция `cudaMemcpyAsync()` – неблокирующий вариант передачи, при котором управление возвращается потоку хоста немедленно. В отличие от операторов присваивания и функции `cudaMemcpy()`, *для асинхронной передачи данных участие закрепленной памяти на хосте обязательно*, и, кроме того, должен быть задан дополнительный аргумент – идентификатор потока (stream) CUDA. Поток CUDA – это просто последовательность операций, выполняемых на устройстве в определенном порядке. Операции в разных потоках CUDA могут чередоваться, а иногда даже перекрываться – это свойство можно использовать для маскирования передачи данных между хостом и устройством.

Асинхронная передача позволяет двумя разными способами организовать перекрытие операций передачи данных и вычислений во

времени. На всех устройствах, поддерживающих CUDA, существует возможность перекрывать вычисления на хосте с асинхронной передачей данных и с вычислениями на устройстве. Так, в следующем фрагменте показана подпрограмма `cpuRoutine()`, которая выполняется на хосте одновременно с передачей данных на устройство и выполнением ядра.

```
istat = cudaMemcpyAsync(a_d, a_h, nElements, 0)
call kernel<<<gridSize,blockSize>>>(a_d)
call cpuRoutine(b)
```

Первые три аргумента функции `cudaMemcpyAsync` – такие же, как аргументы `cudaMemcpy`. Последний же аргумент – это идентификатор потока CUDA; в данном случае он равен 0, то есть определяет поток по умолчанию. Ядро также пользуется потоком по умолчанию. Поскольку ядро находится в том же потоке CUDA, что асинхронная передача данных, оно не начнет работать, пока не завершится копирование, поэтому явная синхронизация не нужна. Поскольку и операция копирования памяти, и ядро возвращают управление хосту немедленно, подпрограмма `cpuRoutine()`, работающая на хосте, может перекрываться с ними во времени.

В этом примере копирование памяти и выполнение ядра происходят последовательно. Но на устройствах, поддерживающих «параллельное копирование и выполнение», выполнение ядра на устройстве может перекрываться с передачей данных между хостом и устройством. Обладает ли конкретное устройство такой возможностью, можно узнать из поля `deviceOverlap` переменной `cudaDeviceProp`, которое выводится также утилитой `pgaccelinfo`. Для устройств, где эта возможность поддерживается, перекрытие все равно требует закрепленной памяти на хосте, и, кроме того, операция передачи данных и ядро должны использовать разные потоки CUDA, не совпадающие с потоком по умолчанию (то есть имеющие ненулевые идентификаторы). Последнее необходимо, потому что операции копирования памяти, операции заполнения памяти и вызовы ядра, использующие поток по умолчанию, начинаются только после того, как все предыдущие операции на устройстве (в любом потоке CUDA) завершатся, и, наоборот, никакая операция на устройстве (в любом потоке CUDA) не начнется, пока не завершатся все такие операции. В следующем фрагменте

```
istat = cudaStreamCreate(stream1)
istat = cudaStreamCreate(stream2)
```

```
istat = cudaMemcpyAsync(a_d, a, n, stream1)
call kernel<<<gridSize,blockSize,0,stream2>>>(b_d)
```

создаются два потока CUDA, которые затем используются в операции передачи данных и вызове ядра. Идентификатор потока задается в последнем аргументе функции `cudaMemcpyAsync()` и в последнем аргументе конфигурации выполнения ядра[2].

В тех случаях, когда операции над данными в ядре поточечные, то есть не зависят от других данных, имеет смысл организовать *конвейер*, состоящий из операций передачи данных и выполнения ядер: данные можно разбить на порции и передавать в несколько этапов, при этом запускается несколько ядер, обрабатывающих каждую порцию по мере поступления, а, когда ядро закончит работу, результаты отправляются назад хосту. В следующем листинге приведен полный код программы, в которой эта техника перекрытия передачи данных и выполнения ядер позволяет замаскировать время, затрачиваемое на передачу.

```
 1 ! Эта программа демонстрирует стратегию маскирования времени
 2 ! данных за счет асинхронного копирования данных в нескольких потоках
 3
 4 module kernels_m
 5 contains
 6   attributes( global) subroutine kernel(a, offset)
 7     implicit none
 8     real :: a(*)
 9     integer, value :: offset
10     integer :: i
11     real :: c, s, x
12
13     i = offset + threadIdx%x + (blockIdx%x-1)* blockDim%x
14     x = i; s = sin(x); c = cos(x)
15     a(i) = a(i) + sqrt(s**2+c**2)
16   end subroutine kernel
17 end module kernels_m
18
19 program testAsync
20   use cudafor
21   use kernels_m
22   implicit none
23   integer, parameter :: blockSize = 256, nStreams = 4
24   integer, parameter :: n = 4*1024* blockSize*nStreams
25   real, pinned, allocatable :: a(:)
26   real, device :: a_d(n)
```

[2] Последние два аргумента конфигурации выполнения необязательны. Третий аргумент относится к использованию в ядре разделяемой памяти, о чем пойдет речь ниже в этой главе.

```
27   integer( kind= cuda_stream_kind) :: stream(nStreams)
28   type (cudaEvent) :: startEvent, stopEvent, dummyEvent
29   real :: time
30   integer :: i, istat, offset, streamSize = n/nStreams
31   logical :: pinnedFlag
32   type (cudaDeviceProp) :: prop
33
34   istat = cudaGetDeviceProperties(prop, 0)
35   write(*,"(' Устройство: ', a,/) ") trim(prop%name)
36
37   ! выделить закрепленную память на хосте
38   allocate(a(n), STAT=istat, PINNED= pinnedFlag)
39   if (istat /= 0) then
40      write (*,*) 'Ошибка при выделении памяти'
41      stop
42   else
43      if (.not. pinnedFlag) &
44         write (*,*) 'Не удалось выделить закрепленную память '
45   end if
46
47   ! создать события и потоки
48   istat = cudaEventCreate(startEvent)
49   istat = cudaEventCreate(stopEvent)
50   istat = cudaEventCreate(dummyEvent)
51   do i = 1, nStreams
52      istat = cudaStreamCreate(stream(i))
53   enddo
54
55   ! базовый случай - последовательная передача и выполнение
56   a = 0
57   istat = cudaEventRecord(startEvent,0)
58   a_d = a
59   call kernel<<<n/blockSize, blockSize>>>(a_d, 0)
60   a = a_d
61   istat = cudaEventRecord(stopEvent, 0)
62   istat = cudaEventSynchronize(stopEvent)
63   istat = cudaEventElapsedTime(time, startEvent, stopEvent)
64   write (*,*) 'Время последовательной ', &
65          'передачи и выполнения (мс): ', time
66   write (*,*) ' макс погрешность: ', maxval(abs(a-1.0))
67
68   ! асинхронная версия 1: цикл {копирование, ядро, копирование}
69   a = 0
70   istat = cudaEventRecord(startEvent,0)
71   do i = 1, nStreams
72      offset = (i -1)*streamSize
73      istat = cudaMemcpyAsync( &
74           a_d(offset +1),a(offset +1), streamSize,stream(i))
75      call kernel<<<streamSize/blockSize, blockSize, &
76                   0, stream(i)>>>(a_d, offset)
```

```
 77      istat = cudaMemcpyAsync( &
 78           a(offset +1),a_d(offset +1),streamSize,stream(i))
 79   enddo
 80   istat = cudaEventRecord(stopEvent, 0)
 81   istat = cudaEventSynchronize(stopEvent)
 82   istat = cudaEventElapsedTime(time, startEvent, stopEvent)
 83   write (*,*) 'Время передачи и выполнения ', &
 84         'в асинхронной V1 (мс): ', time
 85   write (*,*) ' макс погрешность: ', maxval(abs(a-1.0))
 86
 87   ! асинхронная версия 2:
 88   ! цикл копирования, цикл выполнения ядра, цикл копирования
 89   a = 0
 90   istat = cudaEventRecord(startEvent,0)
 91   do i = 1, nStreams
 92      offset = (i -1)*streamSize
 93      istat = cudaMemcpyAsync( &
 94           a_d(offset +1),a(offset +1),streamSize,stream(i))
 95   enddo
 96   do i = 1, nStreams
 97      offset = (i -1)* streamSize
 98      call kernel<<<streamSize/ blockSize, blockSize, &
 99                   0, stream(i)>>>(a_d, offset)
100   enddo
101   do i = 1, nStreams
102      offset = (i -1)*streamSize
103      istat = cudaMemcpyAsync (&
104           a(offset +1),a_d(offset +1),streamSize,stream(i))
105   enddo
106   istat = cudaEventRecord(stopEvent, 0)
107   istat = cudaEventSynchronize(stopEvent)
108   istat = cudaEventElapsedTime(time, startEvent, stopEvent)
109   write (*,*) 'Время передачи и выполнения  ', &
110         'в асинхронной V2 (мс): ', time
111   write (*,*) ' макс погрешность: ', maxval(abs(a-1.0))
112
113   ! асинхронная версия 3:
114   ! цикл копирования, цикл {ядро, событие},
115   ! цикл копирования
116   a = 0
117   istat = cudaEventRecord(startEvent,0)
118   do i = 1, nStreams
119      offset = (i -1)*streamSize
120      istat = cudaMemcpyAsync( &
121           a_d(offset +1),a(offset +1),streamSize,stream(i))
122   enddo
123   do i = 1, nStreams
124      offset = (i -1)*streamSize
125      call kernel<<<streamSize/blockSize, blockSize, &
126                   0, stream(i)>>>(a_d, offset)
```