МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

А.А. Романенко

ОСОБЕННОСТИ АДАПТАЦИИ ПРОГРАММ ПОД GPU С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ OPENACC

Методическое пособие

Новосибирск
2016

УДК 519.684.6
ББК В185.12я73-1
Р691

Рецензент:
канд. физ.-мат. наук *А. В. Снытников*

Издание подготовлено в рамках реализации *Программы развития государственного образовательного учреждения высшего профессионального образования «Новосибирский государственный университет» на 2009–2018 годы.*

Р691 **Романенко, А. А.**
Особенности адаптации программ под GPU с использованием технологии OpenACC: учеб. пособие / А. А. Романенко; Новосиб. гос. ун-т. – Новосибирск : РИЦ НГУ, 2016. – 33 с.

ISBN 978-5-4437-0479-1

В пособии обсуждаются особенности адаптации программ с использованием директив OpenACC для выполнения части расчетов на графических процессорах. Затрагиваются такие вопросы, как профилирование кода, разметка кода директивами компилятора, отладка программы и пр. В качестве компилятора, поддерживающего данный стандарт, использовался компилятор PGI.

Пособие предназначено для студентов инженерных и технических факультетов университетов, тем, кто занимается математическими расчетами, а также всем самостоятельно изучающим программирование графических процессоров.

УДК 519.684.6
ББК В185.12я73-1

ISBN 978-5-4437-0479-1

Оглавление

Введение

Уже не одно десятилетие борьба за вычислительные ресурсы идет как между государствами, так и на уровне отдельных организаций. Чем больше производительность вычислительных систем, тем быстрее на рынок можно вывести новый продукт, обеспечить лучшие показатели и пр. С другой стороны, большие вычислительные ресурсы требуют больших затрат энергии. Так, на июнь 2015 года самый мощный вычислительный комплекс Tianhe-2 (MilkyWay-2) при пиковой производительности 54.9 PFlops и реальной[1] 33.86 PFlops потребляет 17.8 MW [www.top500.org]. Второй в списке комплекс Titan с пиковой производительностью 27.1 PFlops и реальной 17.59 PFlops потребляет 8.21 MW. Оба эти вычислительных комплекса являются гибридными вычислительными системами. В качестве сопроцессоров используются графические процессоры NVidia K20x и Intel XeonPhi. Использование сопроцессоров позволяет существенно поднять производительность комплекса, при этом незначительно увеличивая его энергопотребление. Так, на июнь 2015 года в первой десятке списка самых энергоэффективных компьютеров в мире семь построены на базе графических процессоров и шесть из них – на базе графических процессоров Nvidia [www.green500.org]. Вычислительные комплексы и компьютеры с графическими процессорами сейчас доступны практически всем.

Другой вопрос – эффективность использования доступных вычислительных ресурсов. Для того чтобы задействовать для вычислений графический процессор, необходим компилятор, который знает, как с этим графическим процессором работать, а также необходима специальная программа, которая будет исполняться на графическом процессоре. Большинство исследователей, занимающихся математическими расчетами, не являются профессионалами в высокопроизводительных вычислениях, и часто у них нет возможности вникать в особенности архитектуры новых процессоров и пр. Для такой группы людей в свое время был разработан стандарт OpenMP, позволяющий с помощью директив компилятора переносить код на многоядерные системы. Аналогичный директивный подход используется и в стандарте OpenACC, который позволяет часть вычислений переносить на графические процессоры.

В данном пособии обсуждаются особенности адаптации программ с использованием директив OpenACC для выполнения части расчетов на графических процессорах. В качестве компилятора, поддерживающего данный стандарт, использовался компилятор PGI. Данное пособие не претендует на полное и подробное описание стандарта OpenACC. Описание стандарта можно найти на официальном сайте

[1] Производительность полученная на тестах Linpack

[http://www.openacc.org/]. Информация о том, в каком объеме стандарт OpenACC поддерживается компилятором PGI, доступна на сайте разработчиков компилятора [http://www.pgroup.com/support/release.htm].

Стандарт OpenACC

OpenACC (**Open Acc**elerators) – программный стандарт для параллельного программирования. В разработке первой версии стандарта приняли участие такие компании, как NVidia, PGI, CAPS, Cray. Сейчас список участников существенно расширен. Создатели стандарта ставили себе цель упростить разработку параллельных программ для гетерогенных вычислительных систем. Здесь, как и в OpenMP, программист может разметить код на языках C/C++ или FORTRAN директивами, определяя те участки кода, которые должны быть перенесены на графический процессор. Сейчас разработчики стандарта OpenACC тесно работают в группе OpenMP, чтобы сблизить эти два стандарта, расширить стандарт OpenMP с целью поддержки графических ускорителей в будущих релизах. Так, в черновике стандарта OpenMP 4.1 можно найти часть директив по управлению данными, позаимствованных из стандарта OpenACC.

На текущий момент доступно несколько компиляторов, поддерживающих стандарт OpenACC. Это коммерческие компиляторы от компаний Cray, CAPS, PGI. Компиляторы с открытым кодом – OpenUH и OpenARC (полнота поддержки стандарта не изучалась). Экспериментальная поддержка реализована в компиляторе GCC 5 [https://gcc.gnu.org/wiki/OpenACC].

Для знакомства со стандартом OpenACC компания PGI предоставляет пробную версию своего компилятора на 30 дней. Для академических организаций и университетов можно получить урезанную версию пакета на больший срок (openACC ToolKit, 2015).

В стандарте предусмотрены как директивы, так и отдельные функции.

#pragma acc <directive> [clause [clause […]]] Структурный блок программы
#pragma acc <directive> [clause [clause […]]] операция
#pragma acc <directive> [clause [clause […]]]

Рис. 1. Синтаксис директив в C/C++

!$acc <directive> [clause [clause […]]] Структурный блок программы !$acc end <directive>
!$acc <directive> [clause [clause […]]] Операция
!$acc <directive> [clause [clause […]]]

Рис. 2. Синтаксис директив в FORTRAN

Все директивы OpenACC можно разбить на три группы:

1. Директивы параллельного исполнения: **parallel**, **kernels**, **loop**.
2. Директивы управления данными: **data**, **enter data**, **exit data**, **update**.
3. Прочие: **routine**, **atomic**, **wait**, **host_data**.

Директивы параллельного выполнения

Директива **parallel** позволяет пометить участок кода, содержащий параллелизм. Встретив эту директиву, компилятор создаст ядро для исполнения на графическом процессоре.

Директивой **kernels** указывается, что код может содержать параллелизм, и компилятор определяет, что из этого кода может быть безопасно распараллелено.

```
#pragma acc parallel
{
   for(int i=0; i<N; i++){
      y[i] = a*x[i]+y[i];
   }
   for(int i=0; i<N; i++){
      z[i] = a*x[i]+z[i];
   }
}
```

```
#pragma acc kernels
{
    for(int i=0; i<N; i++){
       x[i] = 1.0;
       y[i] = 2.0;
    }
    for(int i=0; i<N; i++){
       y[i] = a*x[i] + y[i];
    }
}
```

Рис. 3. Пример использования директив parallel и kernels

Внешне эти две директивы делают одно и тоже, хотя в их обработке есть существенные отличия. Так, в примерах на рис. 3 в случае директивы **parallel** компилятор создаст одно ядро, а в случае с директивой **kernels** – два. Другим существенным отличием является то, что в случае директивы **parallel** задача анализа кода на возможность распараллеливания в основном на разработчике.

В качестве дополнительных условий к этим двум директивам чаще всего используются:

- **async[(n)]** – говорит о том, что ядро запустить асинхронно в очереди **n,** а по завершению секции не ставить принудительную синхронизацию;
- **if(логическое условие)** – создать две версии кода (для центрального и графического процессоров), и, если **условие** верно, то запустить код на графическом процессоре, в противном случае на – центральном.

```
#pragma acc kernels async          // асинхронный запуск ядра
for(int i=0; i<N; i++){
    …
}
Do_CPU_calculations();             // код на CPU исполняется
                                   // параллельно с кодом
                                   // ядра на графическом процессоре
#pragma acc wait                   // точка синхронизации
```

Рис. 4. Пример использования условия async и директивы wait

Директива loop описывает, какой тип параллелизма использовать для исполнения цикла. Наиболее часто используемые условия:

- **independent** – заверить компилятор, что в данном цикле нет зависимостей и все итерации могут исполняться параллельно;
- **collapse(n)** - превратить n вложенных циклов в один; может быть выгодно в случае, если сами циклы небольшие;
- **private(список переменных)** – указывает список переменных, приватных для данного цикла; используется, если компилятор не может сам распознать зависимости;
- **reduction(операция:список переменных)** – выполнить редукцию над переменными из списка;
- **gang[(выражение)]** – указывает, что итерации цикла будут исполняться параллельно; в терминах CUDA – указание размерности сети потоковых блоков;
- **vector[(выражение)]** – указывает, что итерации цикла исполняются в векторном или SIMD режиме; в терминах CUDA – указание размера потокового блока.
- **seq** – указывает, что цикл будет исполняться последовательно.

```
!$acc kernels
!$acc loop private(tmp) reduction(max:err)
      do I=1,M
         tmp = a(I-1) + 2.0*a(I)…
         err = max(err,tmp)
      enddo
!$acc end kernels
```

Рис. 5. Пример использования условий private и reduction

```
M = 64
N = 64
…
!$acc loop gang vector(16)
do I=2,M-1
!$acc loop gang vector(16)
    do J=2,N-1
      out(J,I) = coef*(a(J-1,I-1)+a(J,I-1)…
    enddo
enddo
```

Рис. 6. Пример использования условий gang и vector

В коде на рис. 6 в терминах CUDA ядро будет запущено на сети потоковых блоков размерностью 4x4, потоковый блок будет иметь размеры 16x16. Если явные размеры не указывать, то компилятор выберет оптимальные значения, основываясь на информации о том, какие ресурсы необходимы для запуска ядра и максимально эффективного использования графического процессора.

Директивы управления данными

Если расставить только директивы параллельного исполнения, то компилятор проанализирует использование данных внутри этих директив и самостоятельно вставит необходимые инструкции по передаче данных на графический процессор и обратно. Проблема здесь в том, что некоторые данные могут оказаться промежуточными между двумя запусками ядер, и их значение на стороне центрального процессора не нужно. В этом случае разработчик может сам определять, какие данные в какое время где находятся.

С помощью директивы **data** разработчик может указать, какие данные в нижеследующей секции присутствуют на графическом процессоре, что с ними надо сделать при в ходе в секцию и при выходе из нее.

Условия:

- **copy(список)** – для всех переменных из списка выделить память и скопировать их значения на графический процессор в начале секции и в память центрального процессора по выходу из секции с освобождением памяти;
- **copyin(список)** – для всех переменных из списка выделить память и скопировать их значения на графический процессор в начале секции; по выходу из секции память на графическом процессоре освободить;
- **copyout(список)** – для всех переменных из списка выделить память на графическом процессоре в начале секции и скопировать значения в память центрального процессора по выходу из секции;

по выходу из секции память на графическом процессоре освободить;

- **create(список)** – для всех переменных из списка выделить память на графическом процессоре в начале секции и по выходу из секции освободить память;
- **present(список)** – считать, что все переменные из списка уже присутствуют на графическом процессоре;
- **present_or_copy**, **present_or_copyin**, **present_or_copyout**, **present_or_create** – комбинация предыдущих условий.
- **if(условие)**;
- и др.

```
!$acc data copy(A) create(Anew)
iter=0
do while ( err > tol .and. iter < iter_max )
    iter = iter +1
    err=0._fp_kind
!$acc kernels
    do j=1,m
       do i=1,n
          Anew(i,j) = .25_fp_kind *( A(i+1,j ) + A(i-1,j ) &
                                    +A(i,j-1)  + A(i,j+1))
          err = max( err, Anew(i,j)-A(i,j))
       end do
    end do

    A = Anew
!$acc end kernels
    IF(mod(iter,100)==0 .or. iter == 1) print *, iter, err
end do
!$acc end data
```

Рис. 7. Пример разметки кода директивами для задачи решения уравнения Пуассона

На рис. 7 наличие дата-секции позволяет избавиться от лишних операций копирования данных из памяти центрального процессора на графический и обратно между запусками ядер на разных итерациях цикла **while**. В примере мы не прописали необходимость выполнения операции редукции для вычисления максимальной величины ошибки. Компилятор достаточно «умен», чтобы самому распознать необходимость данной операции и вставить соответствующие инструкции.

В случае, когда необходимо описать присутствие данных на графическом процессоре, но в то же время нет возможности использовать структурный блок, можно использовать следующие директивы:

- **`enter data`** – вход в неструктурированную дата-секцию. Дополнительные условия: **`create`**, **`copyin`**, **`present_or_copyin`**, и т. д.
- **`exit data`** – выход из неструктурированной дата-секции. Дополнительные условия: **`delete`**, **`copyout`** и пр.

Неструктурированые дата-секции могут использоваться, например, в конструкторе и деструкторе классов. В конструкторе используется директива **`enter data`** для выделения и/или инициализации памяти на графическом процессоре, а в деструкторе – **`exit data`** для освобождения памяти.

Порой возникает необходимость обновить данные в памяти центрального или графического процессора. Для этого можно использовать директиву update с одним из условий:

- **`update host(список)`** – для всех переменных из списка обновить их значения в памяти центрального процессора значениями на графическом процессоре;
- **`update device(список)`** – для всех переменных из списка обновить их значения в памяти графического процессора значениями из памяти центрального процессоре.

Директивам управления данными можно передавать не только имя массивов (в этом случае будет выделена память и скопирован массив целиком), но и указывать ту его часть, которую необходимо скопировать или обновить. Это может быть удобно, например, при блочной обработке данных и организации совмещения передачи данных с исполнением ядер. (Синтаксис см. в описании стандарта.)

```
for(k=0; k< nBlocks; k++){
#pragma acc update device(a[k*szBlock,szBlock])
    ...
}
```

Рис. 8. Пример на C/C++ обновления части массива на устройстве

```
      do k=1,nBlocks
!$acc update device(a((k-1)*szBlock+1:k*szBlock))
        ...
      enddo
```

Рис. 9. Пример на FORTRAN обновления части массива на устройстве

На рис. 8 и 9 даны примеры обновления блока массива **`a`** в начале каждой итерации цикла. Для C/C++ указывается первый элемент массива и количество элементов, а для FORTRAN – первый и последний элементы массива.

Прочие директивы

Часто в циклах, которые необходимо выполнять на графическом процессоре, присутствуют вызовы функций. Для того, чтобы компилятор смог перенести данные циклы на графический процессор, необходимо или включать тело функции в место вызова (руками или с помощью ключей компилятора), или помечать функцию как вызываемую на устройстве. В последнем случае следует использовать директиву routine. Описывая функцию, необходимо указать уровень параллелизма, который содержится в функции (**gang**, **vector**, **worker**, **seq**) и, возможно, ряд других параметров.

```
// mandelbrot.h
#pragma acc routine seq
unsigned char mandelbrot(int Px, int Py);


// Used in main()
#pragma acc parallel loop
   for(int y=0;y<HEIGHT;y++)
   {
     for(int x=0;x<WIDTH;x++)
     {
         image[y*WIDTH+x]=mandelbrot(x,y);
     }
   }
```

Рис. 10. Пример применения директивы routine

В примере на рис. 10 в заголовочном файле указывается, что необходимо создать функцию для графического процессора и что эта функция будет исполняться каждым потоком последовательно; в месте вызова компилятору необходимо знать, что функция доступна на графическом процессоре и это последовательная функция.

В стандарт OpenACC включена поддержка атомарных операций (директива **atomic**). Данной директиве можно указать одно из четырех дополнительных условий: **read**, **write**, **update**, **capture**. Несмотря на поддержку атомарных операций в OpenACC, следует организовывать алгоритм так, чтобы их количество было минимальным. (См. раздел «Оптимизация кода».)

Директивы OpenACC можно использовать совместно с CUDA, OpenCL и оптимизированными под графический процессор библиотеками. Можно, например, добавлять CUDA ядра в программу, размеченную директивам OpenACC, или, наоборот, в OpenACC программе использовать библиотеки, работающие с графическим процессором. Все взаимодействие происходит через передачу адреса переменных на графическом процессоре. Для этого в стандарте OpenACC предусмотрена

директива **host_data**. Эта же директива может использоваться, если разрабатываемая программа использует реализацию MPI, поддерживающую обмен напрямую из/в память графического процессора. На момент написания пособия такими реализациями MPI[2] являются OpenMPI и MVAPICH.

```
int N = 1<<20;
float *x, *y;
// Allocate & Initialize X & Y
...

cublasInit();
#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
    #pragma acc host_data use_device(x,y)
    {
        // Perform SAXPY on 1M elements
        cublasSaxpy(N, 2.0, x, 1, y, 1);
    }
}
cublasShutdown();
```

Рис. 11. Пример использования директивы host_data и вызова функции из библиотеки cuBLAS

Адаптация кода для исполнения на графическом процессоре

Большинство примеров в презентациях и статьях по применению OpenACC выглядит достаточно гладко и просто: расставляем директивы распараллеливания, затем директивы управления данными для обеспечения локальности данных и получаем великолепный результат. Однако такое не всегда возможно. Код должен быть предварительно подготовлен для переноса на графический процессор.

Старый код

Достаточно часто OpenACC используется для адаптации уже готового кода и это, как правило, код, который разрабатывается в течение нескольких десятков лет, возможно, даже разными людьми. Такое характерно для кода различных программ и библиотек, которые исследователи разрабатывали для своих целей. Обычно это код на языке FORTRAN. Одно время на защитах дипломов часто можно было услышать, что студенты перед развитием этого кода предварительно переносили его с FORTRAN на C/C++, а затем добавляли функциональность и/или переносили на параллельные архитектуры.

[2] В англоязычной литературе такие реализации называются CUDA-aware MPI.

Каковы же особенности кода, которые могут препятствовать эффективному переносу на графические процессоры? Необходимо помнить, что графический процессор – это спецпроцессор с SIMT (single instruction, multiple threads) архитектурой. Для его эффективной загрузки должна быть возможность создания сотен и тысяч потоков. Программы же, которые разрабатывались в 80-90х гг., создавались в условиях ограниченного объема памяти, регистров, для процессоров SISD архитектуры, что внесло дополнительные особенности в код.

Так, например, если алгоритм допускал обработку двумерных или трехмерных расчетных областей независимо по линиям вдоль какого-то измерения, то заводились одномерные массивы, в которые в цикле копировалась часть данных, вызывалась функция обработки этих одномерных массивов, и результат возвращался назад в исходные многомерные массивы. С точки зрения разметки кода директивами OpenACC проблема здесь может быть в следующем:

- одномерные массивы могут содержать недостаточное количество элементов для загрузки графического процессора;
- структура функции ввиду ограничений компилятора не может быть помечена директивой «acc routine» или встроена в код (inline);
- одномерные массивы, скалярные переменные для внешнего цикла должны быть объявлены как приватные, что может существенно повышать требования к памяти на графическом процессоре.

В попытках вручную развернуть циклы и повысить параллелизм на уровне инструкций (Instruction Level Parallelism) разработчики заводили массивы из двух-четырех элементов под промежуточные значения. Для того чтобы распараллелить цикл для графического процессора, такие массивы должны быть помечены как приватные, что опять же приводит к дополнительным, порой существенным расходом по памяти. Решением тут может быть замена коротких массивов на отдельные переменные, которые компилятором будут положены на регистры.

```
 1         PROGRAM priv
 2         REAL:: a(128,128), b(128,128)
 3         REAL :: tmp(2)
 4         INTEGER :: i,j
 5 !$acc data copyout(a,b)
 6 !$acc kernels
 7         do j=1,128
 8 !$acc loop private(tmp)
 9         do i=1,128
10            tmp(1) = sin(i*3.1415/180.0)
11            tmp(2) = cos(i*3.1415/180.0)
12            a(i,j) = tmp(1)+tmp(2)
13            b(i,j) = tmp(1)-tmp(2)
14         end do
15         end do
16 !$acc end kernels
17 !$acc end data
18         print *, sum(a), sum(b)
19         END PROGRAM priv
```

Рис. 12. Пример программы, в которой есть цикл с приватным массивом

```
$ PGI_ACC_NOTIFY=61 ./a.out
Enter enter data construct file=/../priv.f90 function=priv line=5 ..
create CUDA data  bytes=65536 file=/../priv.f90 function=priv line=5 .. <= array "a"
alloc  CUDA data  devaddr=0x2304540000 bytes=65536 file=/../priv.f90 function=priv
                                                                  line=5 ..
create CUDA data  bytes=65536 file=/../priv.f90 function=priv line=5 .. <= array "b"
alloc  CUDA data  devaddr=0x2304550000 bytes=65536 file=/../priv.f90 function=priv line=5
Leave enter data construct file=/../priv.f90 function=priv line=5 ..
Enter compute region file=/../priv.f90 function=priv line=6 ..
create CUDA data  bytes=131072 file=/../priv.f90 function=priv
                                                  line=6 .. <= tmp, 128*128*2*4
alloc  CUDA data  devaddr=0x2304640000 bytes=131072 file=/../priv.f90 function=priv
                                                                  line=6 ..
launch CUDA kernel  file=/../priv.f90 function=priv line=9 num_gangs=128
                    num_workers=1 vector_length=128 grid=1x128 block=128
delete CUDA data  devaddr=0x2304640000 bytes=131072 file=/../priv.f90 function=priv
                                                                  line=9 ..
Implicit wait  file=/../priv.f90 function=priv line=9 ..
Leave compute region file=/../priv.f90 function=priv line=9 ..
Enter exit data construct file=/../priv.f90 function=priv line=17 .. queue=0
delete CUDA data  devaddr=0x2304550000 bytes=65536 file=/../priv.f90 function=priv
                                                                  line=17 ..
delete CUDA data  devaddr=0x2304540000 bytes=65536 file=/../priv.f90 function=priv
                                                                  line=17 ..
Implicit wait  file=/../priv.f90 function=priv line=17 ..
Leave exit data construct file=/../priv.f90 function=priv line=17 ..
    17575.32        6223.129
```

Рис. 13. Вывод информации об исполнении директив OpenACC программы, представленной на рисунке выше

На рис. 13 видно, что на пятой строке программы, представленной на рис. 12, происходит выделение памяти под переменные **a** и **b** (по 64КБ под каждую переменную). На шестой строке программы выделяется еще 128КБ. Этот участок памяти будет использован для приватной переменной **tmp**.

Еще одно затруднение при переносе кода на графический процессор вызывает использование глобальных переменных в теле функции. Это не плохо, но препятствует, например, использованию директивы **routine**. Ограничения на функции, которые могут быть помечены директивой **routine**, см. в документации к компилятору. Если такая функция находится в рамках цикла, который необходимо распараллелить, то единственный вариант – включить ее тело в место вызова (inline). Но и тут может возникнуть проблема, если тело функции находится в другом файле или модуле. Сюда же можно добавить наличие реализаций элементарных операций в виде отдельных функций, которые реализованы в отдельном модуле.

Стоит также отметить «информативное» название переменных и функций. Так, например, в коде пакета WRF можно встретить несколько функции GAMMA, при этом некоторые из них возвращают $\ln(\Gamma(x))$.

Еще одной особенностью реализации «старых» программ можно назвать реализацию алгоритмов, при которой у разработчиков возникала необходимость экономить память. Нередко, как в примере ниже, это приводит к невозможности исполнять итерации внешнего цикла независимо. Здесь препятствием к распараллеливанию являются: а) зависимость по последнему индексу переменной **fqy**; б) обмен значениями между переменными **jp0** и **jp1**. В приведенном примере решением было увеличение размерности временной переменной **fqy**, разбиение внешнего цикла на два, в первом из которых происходило заполнение массива **fqy**, а во втором – его использование.

```
jp1 = 2
jp0 = 1
j_loop_y_flux_6 : DO j = j_start, j_end+1
 IF((j>=j_start_f ).and.(j<=j_end_f)) THEN  ! use full stencil
   DO k=kts,ktf
   DO i = i_start, i_end
     vel = 0.5*(rv(i,k,j)+rv(i-1,k,j))
     fqy( i, k, jp1 ) = vel*flux6(               &
              u(i,k,j-3), u(i,k,j-2), u(i,k,j-1), &
              u(i,k,j  ), u(i,k,j+1), u(i,k,j+2),  vel )
   ENDDO
   ENDDO
 ELSE IF …
 END IF
   IF(j > j_start) THEN
     DO k=kts,ktf
     DO i = i_start, i_end
       mrdy=msfux(i,j-1)*rdy   ! ADT eqn 44, 2nd term on RHS
       tendency(i,k,j-1) = tendency(i,k,j-1) -   &
                           mrdy*(fqy(i,k,jp1)-fqy(i,k,jp0))
     ENDDO
```

```
          ENDDO
        ENDIF
        jtmp = jp1
        jp1 = jp0
        jp0 = jtmp
      ENDDO j_loop_y_flux_6
```

Рис. 14. Пример участка кода для расчета перемещения воздушных масс из пакета WRF

Другой особенностью «старых» программ является плохое документирование, на которое иногда накладывается специфика именования переменных и функций. Так, например, в FORTRAN 77 переменная должна была содержать не более восьми символов[3]. Именно поэтому функции из библиотеки BLAS имеют на первый взгляд нечитаемый вид. Иногда к плохому документированию кода прибавляется то, что его разработчик может быть уже недоступен по тем или иным причинам.

```
      SUBROUTINE MATRIX(N,T,V,B,N0)
      IMPLICIT REAL*8 (A-Z)
      INTEGER*2 LD,N,N0,I,J
      DIMENSION V(N+1),B(N0,N0)
      COMMON/ADD/ LD,Kcoaacet,KISBacet,KMSBacet,
     *KISBCoA ,KMSBCoA ,KISBeduc,KMSBeduc,KISBme5P,KISBm5PP,
     *KISBpePP,KdimetPP,KgeraPP1,KgeraPP2,K1GeFPPS,K2GeFPPS,
     *K1IsFPPS,K2IsFPPS,
     *KISBynth,Kpresqu1,Kpresqu2,KISBsqua,KMSBsqua,KISBin  ,
     *Kclaw1  ,Kclaw2  ,Kfagocyt,KISBfree,Kfre____,KISBio  ,
     *KMSBio  ,KISBprot,KMSBprot,Ksre1   ,Ksre2   ,KISBhmgs,
....
*****************************************************************
      DO I = 1,N0
         DO J = 1,N0
            B(I, J) = 0.
         END DO
      END DO
      A1 = KMSBCoA **2+v(1)*v(2)+v(1)*v(3)+v(2)*v(3)
      A2 = KMSBCoA **2+v(1)*v(2)+v(1)*v(3)+v(2)*v(3)
      A3 = KMSBCoA **2+v(1)*v(2)+v(1)*v(3)+v(2)*v(3)
      B(1,1) = -4*KISBacet*v(1)*CoAtiola*(KMSBacet**2+v(1)*CoAtiola)/
     *(KMSBacet**2+v(1)**2+2*v(1)*CoAtiola)**2
     *-KISBCoA *v(2)*v(3)*(KMSBCoA **2+v(2)*v(3))/
     *A1**2-KutiACoA
      B(1,2) = -KISBCoA *v(1)*v(3)*(KMSBCoA **2+v(1)*v(3))/
     *A1**2
      B(1,3) = -KISBCoA *v(1)*v(2)*(KMSBCoA **2+v(1)*v(2))/
     *A1**2
      B(2,1) = 2*KISBacet*v(1)*CoAtiola*(KMSBacet**2+v(1)*CoAtiola)/
     *(KMSBacet**2+v(1)**2+2*v(1)*CoAtiola)**2
     *-KISBCoA *v(2)*v(3)*(KMSBCoA **2+v(2)*v(3))/
...
```

Рис. 15. Пример малопонятного кода

[3] Переменная может содержать и больше восьми символов, но все символы после восьмого будут проигнорированы.

Особенности разметки кода на C/C++

При написании программ на C/C++ программисты, как правило, эффективно используют указатели. Указатель в C/C++ не несет информации о размерности массива или структуры, на которую этот указатель ссылается. Поэтому в C/C++ обязательно надо указывать размерности массивов при их выделении на графическом процессоре и при их обновлении на стороне графического и центрального процессоров.

На рис. 16 представлен фрагмент кода решения уравнения Пуассона, аналогичный тому, что был представлен на рис. 7. В отличае от кода на языке Фортран, здесь добавлена дополнительная data-секция вокруг ядра. Дело в том, что вместо копирования вновь вычисленных значений из массива **anew** в исходный массив, как это было реализовано на Фортране, используется перекидывание указателей (см. строки 49-51). Связывание имени массива на стороне центрального процессора и графического происходит только при в ходе в data-секцию и в случае, если мы опустим вложенную data-секцию на строке 36, мы поменяем указатели только на центральном процессоре, но не на стороне графического процессора.

```
30      #pragma acc data copy(a[0:M*N]) create(anew[0:M*N])
31      {
32      int iter=0;
33      while(err > tol && iter < iter_max) {
34          iter += 1;
35          err = 0;
36          #pragma acc data present(a,anew)
37          #pragma acc parallel reduction(max:err)
38          {
39          #pragma acc loop independent
40          for(int i = 1; i < M-1; i++) {
41              #pragma acc loop independent
42              for(int j = 1; j < N-1; j++) {
43                  anew[i*N+j] = 0.25*(a[(i+1)*N+j] + a[(i-1)*N+j]
44                                    + a[i*N+ j-1]  + a[i*N+j+1]);
45                  err = fmax(err, anew[i*N+j]-a[i*N+j]);
46              }
47          }
48          }
49          tmp = anew;
50          anew = a;
51          a = tmp;
52      } // while
53      } // acc data
```

Рис. 16. Пример разметки кода директивами для задачи решения уравнения Пуассона

Использование в коде указателей порой не дает возможности компилятору разобраться с наличием или отсутствием зависимости по

данным между итерациями цикла. Разработчику рекомендуется использовать при необходимости ключевое слово **__restrict__** при объявлении переменных и условием **independent** директивы **loop**.

Стоит отметить особенности передачи на графичекий процессор массивов, членов класса[4]. На рисунке ниже представлен представлен пример кода в котором на графическом процессоре происходит заполнение массива **a**, который является членом класса **foo**. В коде, представленном слева, в data-секцию передается непосредственно имя массива. Будучи запущенной, данная программа завершается с ошибкой, что некоторые переменные не присутствуют в памяти графического процессора. Для решения данной проблемы рекомендуется завести дополнительную локальную переменную и присвоить ей адрес массива **a**, как это было сделано в коде, представленном справа.

```
class foo{
  float * a;
  public:

  void run(){
#pragma acc data \
        copyout(a[0:1024])
     {
#pragma acc parallel loop \
        independent
     for(int i=0; i<1024; i++)
        a[i] = i;
     }
     for(int i=0; i<10; i++)
        cout<< a[i] << " ";
     cout << endl;
 }
```

```
class foo{
  float * a;
  public:

  void run(){
    float * a_ptr = a;
#pragma acc data \
        copyout(a_ptr[0:1024])
    {
#pragma acc parallel loop \
        independent
    for(int i=0; i<1024; i++)
       a_ptr[i] = i;
    }
    for(int i=0; i<10; i++)
       cout<< a[i] << " ";
    cout << endl;
  }
```

Рис. 17. Пример работающего (справа) и неработающего (слева) варианта программы.

Проверка корректности расчетов

При переносе программы на параллельную архитектуру, в том числе и на графический процессор, возникает вопрос проверки корректности кода, поскольку не всегда результат получается бинарно идентичным. Так происходит по нескольким причинам:

1. На разных процессорах одни и те же вычисления могут производиться по разному. Так, графическим процессором поддерживается операция FMA (Fused Multiply Add)

[4] На момент написания методического пособия описанное поведение проявлялось в программах, собранных компилятором PGI 16.3

[https://en.wikipedia.org/wiki/FMA_instruction_set], которая вычисляет выражение **`A*B+C`** без промежуточной денормализации. Некоторыми центральными процессорами эта операция не используется и вычисление данного выражения распадается на две операции. Понятно, что при этом результат может несколько отличаться. Intel представила FMA инструкции только в процессорах с ядром Haswell (2013), AMD – в процессорах с Piledriver cores в 2012. Поддержка этих инструкций компилятором – отдельный вопрос. Для кода, исполняемого на графическом процессоре, поддержку FMA инструкций можно отключить, передав соответствующий ключ компилятору.

2. Ряд специальных функций, таких как sin(), cos(), exp(), rsqrt() и другие на разных системах могут быть реализованы по разному. Стандарт IEEE 754-2008 не накладывает никаких ограничений на их реализацию и точность.
3. Параллельное исполнение может приводить к изменению порядка выполнения некоторых операций. Так, например, результат параллельного суммирования элементов массива может отличаться от результата, полученного последовательным суммированием.

Учитывая вышесказанное, при сравнении результатов расчетов на разных архитектурах разработчику часто приходится вычислять погрешность и анализировать, находится ли данная погрешность в допустимых пределах.

Проверяя корректность расчетов, стоит также помнить, что компилятор не лишён ошибок, отличие в результатах могут быть обусловлены именно этим фактором.

Ускорение

Как правило, код директивами OpenACC не размечается сразу целиком. Сначала отдельные циклы обрамляются секциями **`kernels`** и **`parallel loop`**, затем вводятся секции данных, происходит оптимизация. Не стоит удивляться, если на первом этапе программа начнет работать медленнее. Как правило, это происходит из-за того, что слишком много данных перемещается между графическим и центральным процессором. В процессе минимизации количества пересылок может возникнуть необходимость часть циклов с небольшим уровнем параллелизма также помечать для исполнения на графическом процессоре.

На рис. 18 представлен график того, как может вести себя ускорение программы на разных этапах адаптации. Стоит отметить, что в реальной жизни ускорение будет меняться скачкообразно.

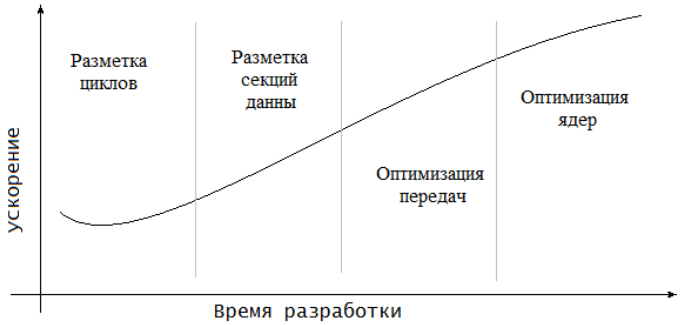

Рис. 18. Наиболее вероятное ускорение на разных этапах адаптации приложения

Оптимизация кода

Прежде чем заниматься оптимизацией кода, необходимо понять, насколько эффективно работает текущая версия кода. Для этого нужно провести профилирование кода – собрать некоторые характеристики о работе программы, которые позволят сделать выводы об эффективности ее работы, выявить узкие места в реализации.

Профилирование

Сразу стоит сказать, что профилирование программы необходимо проводить на реальных данных – данных, наиболее близких по параметрам к тем, с которыми программу будет использовать пользователь. В противном случае могут возникнуть ситуации, когда реальная причина просадки по производительности и/или неработоспособности программы останется незамеченной. Также перед профилированием необходимо убедиться, что результат, полученный с помощью адаптированной программы, является корректным и с приемлемой точностью совпадает с результатом, который выдает исходная программа.

Профилирование программы стоит начать с измерения времени исполнения исходной версии программы на центральном процессоре и программы, полученной в результате адаптации. Сравнивая время работы двух программ, необходимо учитывать, какое количество ядер центрального процессора используется исходной программой (OpenMP, MPI, пр.), используется ли адаптированной программой одна или несколько графических карт, какое используется оборудование. Если программа допускает масштабирование на множество узлов кластера, то стоит проверить как меняется ускорение в зависимости от количества используемых процессоров (CPU и GPU). В таблице ниже приводится

пиковая производительность некоторых графических и центральных процессоров. Эти значения стоит учитывать при оценке качества адаптации программы. Поскольку все современные центральные процессоры являются многоядерными, то интерес представляет ускорение, полученное при использовании графического процессора по сравнению с сокетом центрального процессора.

Таблица 1. Пиковая производительности некоторых CPU и GPU. Одинарная точность

CPU	E3-1280 v5	i7 3770	i7 4770	E5-2699 v3	E7-8890 v3
GFlop/s	~250	~250	~500	~600	~950
GPU	GT 340	GTX 580	GTX 680	GTX 780 Ti	GTX 980 Ti
TFlop/s	0.386	1.5811	3.090	5.046	5.630
GPU	K10 (2 чипа)	K20X	K40	K80 (2 чипа)	M40
TFlop/s	4.58	3.95	5	8.74	7

Если полученное ускорение или результаты масштабирования являются неудовлетворительными, то необходимо проводить более глубокий анализ работы программы.

Самый простой вариант – воспользоваться встроенными в библиотеку OpenACC средствами. Для этого необходимо установить переменную окружения PGI_ACC_TIME.

```
$ PGI_ACC_TIME=1 ./a_acc.out
…
/home-2/…/transport_single_component.for
 moveallparticles  NVIDIA  devicenum=0
    time(us): 105,444
    754: data region reached 157 times
    755: compute region reached 157 times
        755: kernel launched 157 times
           grid: [1]  block: [128]
             device time(us): total=644 max=10 min=4 avg=4
           elapsed time(us): total=6,490 max=526 min=34 avg=41
    780: data region reached 157 times
        780: data copyin transfers: 2826
             device time(us): total=33,368 max=92 min=5 avg=11
    788: compute region reached 157 times
        790: kernel launched 157 times
           grid: [2-625]  block: [128]
             device time(us): total=61,442 max=941 min=87 avg=391
           elapsed time(us): total=66,242 max=971 min=112 avg=421
    811: data region reached 157 times
        811: data copyout transfers: 314
             device time(us): total=9,990 max=79 min=8 avg=31
    812: data region reached 157 times
```

Рис. 19. Вывод информации о профилировании программы

Из рис. 19 видно, что

1) в функции moveparticles() в файле transport_single_component.for секция данных, которая начинается на строке 754, вызывалась 157 раз. Столько же раз вызывались два ядра, которые начинаются на строках 755 и 788;
2) второе ядро (на строке 788) обрамлено передачей данных на графический процессор перед вызовом ядра (на строке 780) и возвратом данных на центральный процессор после (на строке 811);
3) количество передач на строке 780 – 2826 шт., что составляет 18 раз на один вызов ядра. Малое время (в среднем 11 микросекунд) говорит о том, что передаются очень маленькие объемы данных, скорее всего, отдельные переменные. Убедиться в этом можно, просмотрев отчет о сборке программы, если включить опцию **`-Minfo=accel`**, или запустить программу еще раз, предварительно установив переменную окружения PGI_ACC_NOTIFY=65. Возможно, стоит объединить данные в структуру и передать на графический процессор за один раз, или, если эти данные доступны только на чтение и переиспользуются многократно, перенести их на графический процессор один раз в начале программы, а тут указать как уже присутствующие на устройстве;
4) первое ядро (на строке 755) запускается на сети потоковых блоков размерности 1. Время исполнения ядра на графическом процессоре (device time) на порядок меньше времени, которое затрачивает центральный процессор на запуск этих ядер и ожидания их завершения (elapsed time). Это говорит о том, что в блоке кода не достаточно параллелизма для графического процессора и, возможно, этот код лучше оставить на центральном процессоре.

Аналогичным образом можно проанализировать все участки кода, размеченные с помощью инструкций OpenACC. Если полученная таким образом информация не является достаточной для понимания, то следующий шаг – это воспользоваться утилитой nvprof из пакета NVidia CUDA ToolKit [https://developer.nvidia.com/cuda-zone]. В OpenACC работа с графической картой Nvidia идет через API драйвера и поэтому все передачи данных и запуск ядер будет запротоколирован утилитой nvprof.

По умолчанию результат профилирования утилита nvprof выводит в консоль в текстовом виде. Его можно перенаправить в файл для дальнейшего анализа. Результат профилирования можно сохранить в бинарном файле, который затем визуализировать в программе NVVP (NVidia Visual Profiler) из CUDA ToolKit.

```
$ nvprof -o log.nvprof ./a_acc.out
==20095== NVPROF is profiling process 20095, command: ./a_acc.out
==20095== Generated result file: /home-2/.../log.nvprof
$ nvprof -i log.nvprof
======== Profiling result:
Time(%)      Time  Calls       Avg       Min       Max  Name
 45.52%  115.50ms    157  735.64us  115.13us  1.9226ms
                                    moveallparticles_779_gpu
 43.06%  109.25ms   7856  13.906us  1.4720us  212.86us
                                    [CUDA memcpy HtoD]
  9.03%  22.915ms    471  48.652us  3.4240us  206.04us
                                    [CUDA memcpy DtoH]
  1.81%  4.5916ms    157  29.245us  8.0640us  72.638us
                                    reseed_component_138_gpu
  0.36%  914.19us    157  5.8220us  4.3840us  8.6720us
                                    reseed_component_138_gpu_red
  0.22%  546.22us    157  3.4790us  2.8790us  10.399us
                                    moveallparticles_755_gpu
```

Рис. 20. Вывод информации о профилировании

На рис. 20 видно, что передача данных занимает столько же времени, что и время исполнения ядер. Если таковое не используется, то в целях оптимизации стоит подумать об асинхронной передаче данных и запуске ядер. Также стоит обратить внимание на ядра, которые находятся в верхней части списка и занимают основное время от времени использования графического процессора. Кстати, наличие ядра с суффиксом **_red** говорит о том, что после соответствующей функции вызывается еще одно ядро для окончательного вычисления результата операции редукции.

Дополнительную информацию о результатах профилирования можно получить, если передать следующие опции:

- **--print-gpu-trace** – вывод подробной информации о запуске каждого ядра и операции передачи данных;
- **--print-api-trace** – вывод подробной информации о каждом обращении к драйверу: запуск ядра, инициализация передачи данных, выделение памяти, ожидание завершения операции и пр.

Подобная информация о том, какие события и информация могут собираться в процессе профилирования доступна в документации.

При работе на кластерах может потребоваться сбор информации о каждом запущенном MPI процессе. Чтобы отделять результаты профилирования одного MPI процесса от другого, предусмотрена возможность записи результатов профилирования в разные файлы.

```
// %p  будет заменено на PID процесса
$ mpirun -np <n> nvprof -o log.%p.nvprof ./a.out

// %q{<ENV>}  будет заменено на значение переменной окружения ENV
$mpirun -np <n> nvprof -o log.%q{OMPI_COMM_WORLD_RANK}.nvprof
./a.out
```

Рис. 21. Примеры запуска MPI-программы для профилирования

Если есть возможность, то профилирование и анализ результатов рекомендуется проводить в NVidia Visual Profiler (NVVP). Среди особенностей NVVP стоит отметить автоматический анализ возможных проблем с производительностью и их причин, а также возможность удаленного профилирования.

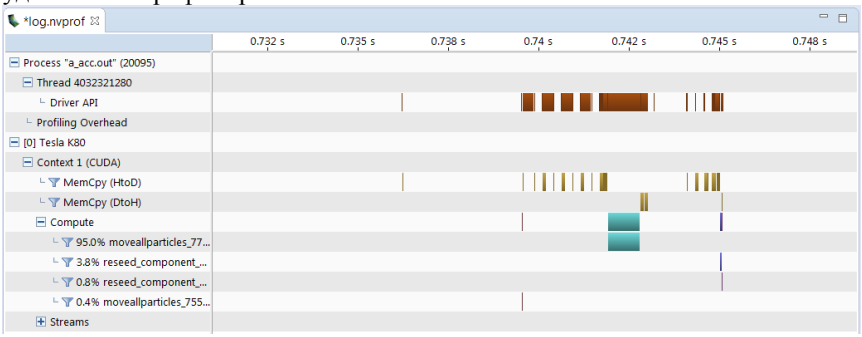

Рис. 22. Результат визуализации по использованию графического процессора

Визуализировав информацию, можно лучше понять, как ведет себя программа, есть ли перекрытие передачи данных с исполнением ядер, параллельное исполнение ядер, параллельное исполнение кода на центральном и графическом процессорах и пр.

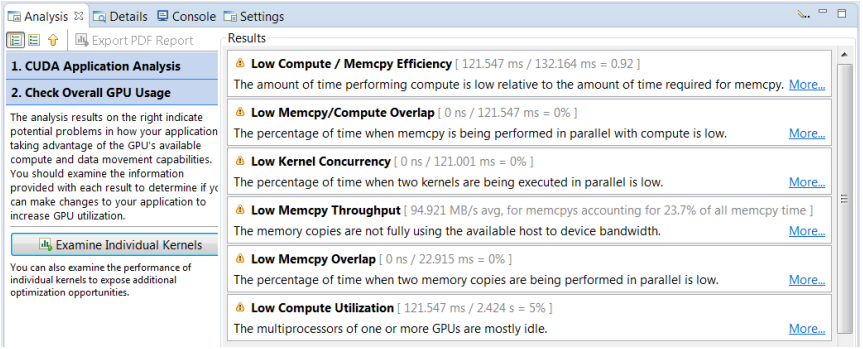

Рис. 23. Один из вариантов автоматического анализа

При профилировании кода дополнительную информацию можно получить, если вызвать специальные функции в начале и в конце участков кода, которые интересуют. Такие функции позволяют замерять время, протоколировать операции, которые были выполнены на стороне как драйвера, так и графического процессора в рамках данного диапазона. Работать с такими функциями позволяет библиотека NVidia Tools Extension (NVTX) [http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvtx].

```
#include <nvToolsExtCuda.h>
#include <nvToolsExtCudaRt.h>
...
void init_data( int n, double * x ) {
    nvtxRangePushA("init");
    //initialize x
    ...
    nvtxRangePop();
}
...
```

Рис. 24. Пример того, как можно отметить диапазон при профилировании

Помимо имени диапазона («init» на рисунке выше) можно задать цвет, которым данный диапазон будт отмечен в NVVP.

```
========      NVTX range "w_damp" (36 times, total time: 26.389ms)
Time(%)      Time     Calls       Avg       Min       Max  Name
 86.49%  4.4068ms        36  122.41us  120.93us  125.57us
                                                 w_damp_2741_gpu
  9.40%  478.83us        36  13.300us  13.152us  13.536us
                                                 w_damp_2741_gpu_red
  3.01%  153.38us        36  4.2600us  3.7760us  4.6080us
                                                 [CUDA memcpy DtoH]
  1.10%  56.033us        36  1.5560us  1.4720us  2.0160us
                                                 [CUDA memcpy HtoD]
```

```
========      API calls:
Time(%)      Time      Calls       Avg       Min       Max  Name
 94.91%  24.402ms         36  677.84us  359.28us  1.7062ms
                                                  cuMemcpyDtoHAsync
  3.45%  886.74us         72  12.315us  10.059us  17.235us
                                                  cuLaunchKernel
  1.64%  421.90us         36  11.719us  9.6200us  18.034us
                                                  cuMemcpyHtoDAsync
```

Рис. 25. Вывод программы nvprof. Информация о диапазоне "w_damp"

Различные примеры исползования библиотеки NVTX и ее интеграции в FORTRAN представлены по ссылкам ниже:

- http://devblogs.nvidia.com/parallelforall/customize-cuda-fortran-profiling-nvtx/;
- http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/;
- http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvtx.

Диапазоны, отмеченные маркерами через функции nvtxRangePushA() и nvtxRangePop(), могут быть вложенными, но не перекрываться. Если по каким-то причинам нужны перекрывающиеся диапазоны, то необходимо использовать функции nvtxRangeStartA() и nvtxRangeEnd(). На рис. 25 отмеченные в коде диапазоны представлены зелеными и красным прямоугольниками в секции «Markers and Ranges» (Маркеры и диапазоны).

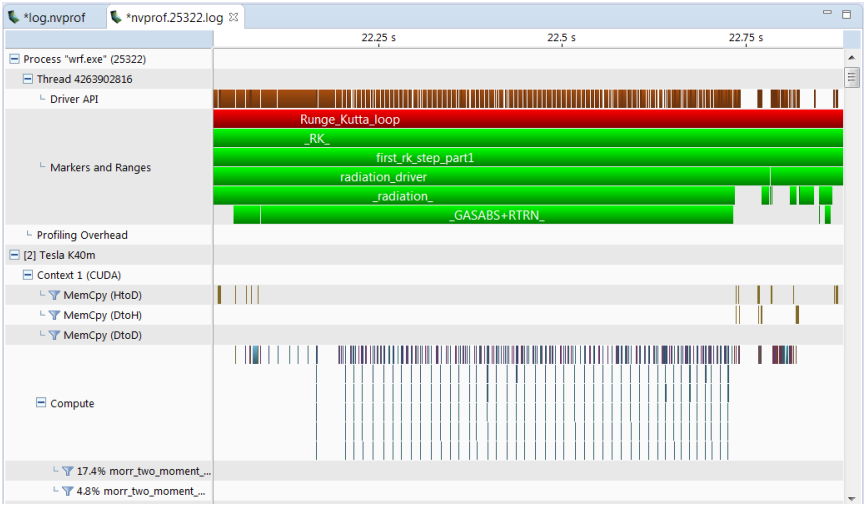

Рис. 26. Результат расстановки маркеров в коде с использованием библиотеки NVTX

Примеры анализа кода с помощью NVidia Visual Profiler и его оптимизации можно найти в презентациях на GPU Technology conference:

- Jeff Larkin, “Advanced OpenACC Programming”, URL: http://on-demand.gputechconf.com/gtc/2015/presentation/S5195-Jeff-Larkin.pdf;
- Jeff Larkin, “Introduction to Compiler Directives with OpenACC”, URL: http://on-demand.gputechconf.com/gtc/2015/presentation/S5192-Jeff-Larkin.pdf .

Оптимизация кода

Оптимизация кода – это итеративный процесс, в рамках которого выполняется профилирование и оптимизация проблемных участков. После того, как были выявлены проблемные места в коде, можно приступать к их оптимизации. Оптимизация включает оптимизацию передачи данных и оптимизацию ядер.

Оптимизация передачи данных

Проводя профилирование промежуточной версии программы, стоит учитывать, что некоторые передачи, которые, например, использовались для отладки, могут исчезнуть в окончательной версии программы. Для оптимизации передачи данных с графического процессора на хост и обратно можно дать следующие рекомендации:

- **Выполняйте меньше передач данных**. Если это возможно, постарайтесь передавать данные за пределами основного цикла. Попробуйте организовать вашу программу так, чтобы все необходимые данные загружались в начале программы и всегда находились на графическом процессоре, а в память центрального процессора передавалось только то, что надо сохранить в качестве результата. Порой от некоторых передач удается избавиться, если пометить переменные как приватные. Уменьшить количество передач в MPI программах позволит использование CUDA-aware MPI.
- **Объединяйте маленькие передачи в большие**. Для передачи малых объемов данных инициализация передачи может занимать больше времени, чем сама передача данных.
- **Используйте pinned память**. Часть памяти центрального процессора может быть помечена как pinned память. Эта память не выгружается в файл подкачки и именно она используется для выполнения асинхронных операций по передаче данных. Чтобы ваш код, размеченный директивами OpenACC, использовал эту память, необходимо добавить опцию **`pin`** к ключу компиляции:

`-ta=nvidia,pin`. Без использования данного ключа процесс передачи данных выглядит следующим образом:

- при запуске программы OpenACC библиотека выделяет область памяти и помечает ее как pinned память. Размер этой памяти может контролироваться через переменную окружения **`PGI_ACC_BUFFERSIZE;`**
- при инициализации передачи данных на графический процессор массив (часть массива, если он не помещается в отведенный участок pinned памяти) копируется в pinned память;
- запускается асинхронная передача данных;
- процесс повторяется до тех пор, пока весь массив не будет передан на графический процессор.

Процесс копирования данных с графического процессора выглядит аналогично. На рис. 26 представлена временная развертка процесса копирования большого массива данных с графического процессора в память центрального процессора. Копирование происходит по частям, и зазоры между операциями копирования – перемещения частей массивов из pinned-памяти в результирующий массив.

- **Совмещайте передачу данных с другими операциями**. Совмещение передачи данных возможно с исполнением ядер и вычислениями на центральном процессоре[5]. Для этого необходимо использовать условие **`async`** директив **`update device`**, **`update host`** и др.

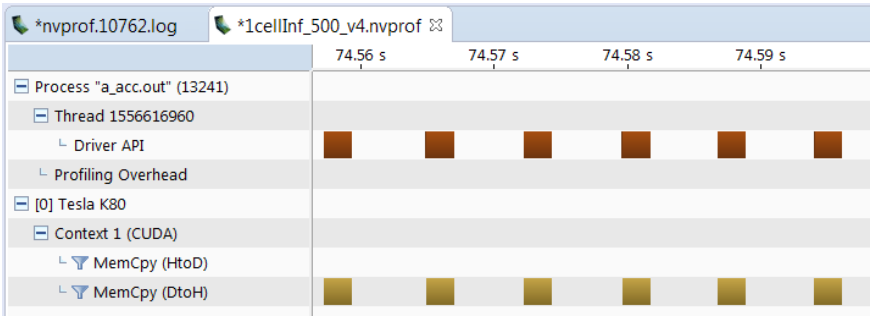

Рис. 27. Процесс копирование данных

[5] На момент написания пособия компилятор PGI 15.9 копирование данных с графического процессора производил в синхронном режиме. http://www.pgroup.com/userforum/viewtopic.php?p=18936#18936

Оптимизация ядер

Разработчики компилятора PGI заявляют, что компилятор сам должен выбирать оптимальные параметры запуска ядер. Однако у пользователя остается ряд возможностей по настройке параметров запуска ядра (**`gang`**, **`vector`**, **`tile`**, **`cache`**, пр.).

На странице https://www.pgroup.com/resources/openacc_tips_fortran.htm разработчики компилятора PGI дают ряд рекомендаций, как правильно реализовывать некоторые циклы с тем, чтобы они эффективно исполнялись на графическом процессоре.

Атомарные операции часто являются узким местом в параллельных программах. Справедливо это и для OpenACC программ. Однако, изменив алгоритм, можно существенно уменьшить количество атомарных операций, которые могут происходить одновременно над одной и той же переменной и тем самым повысить скорость работы программы.

Например, для сохранения промежуточных значений можно завести временный массив в несколько раз больше, чем результирующий. В зависимости от какого-то параметра каждый поток заполняет свою часть временного массива, а после его заполнения в еще одном цикле производится вычисление финального результата. Ввиду того, что количество элементов массива, к которым происходит обращение, больше, чем в первом варианте, то и количество атомарных операций, блокирующих друг друга, будет существенно меньше. Пример того, как это реализуется на CUDA, представлен тут [http://devblogs.nvidia.com/parallelforall/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/].

Отладка OpenACC программ

Отладка любой программы, в том числе и программы, размеченной директивами OpenACC, – это итеративный процесс, в рамках которого строятся предположения об источнике ошибки, а на их основе составляются тесты, и по результатам тестов проверяется сделанная гипотеза. Далее будем исходить из предположения, что та же самая программа, но без директив OpenACC (или с ними, но без соответствующих опций компилятора), работает корректно.

В программах, размеченных с помощью OpenACC директив, могут встречаться следующие ошибки:

- ***Ложное выявление параллелизма***. Блок кода, который пользователь пометил директивой **`acc parallel`** или **`loop independent`**, не является на самом деле параллельным циклом.
- ***Неактуальность данных***. Пользователь забыл обновить данные на устройстве или в памяти центрального процессора.

- ***Ошибки граничных данных***. Недостаточно данных перемещено на устройство.
- ***Ошибка "acc data present"***. Разработчик считает, что данные находятся на устройстве, где они реально не находятся. По крайней мере в данном участке кода.

Ошибки округления. **Не являются ошибкой как таковой. Стоит говорить об особенностях реализации вычислений. Возникают по причине разности арифметики с плавающей точкой на графическом и графическом и центральном процессорах. Возникновение отклонения отклонения возможно при параллельном суммировании, умножении. умножении. См. раздел «Особенности разметки кода на C/C++**

При написании программ на C/C++ программисты, как правило, эффективно используют указатели. Указатель в C/C++ не несет информации о размерности массива или структуры, на которую этот указатель ссылается. Поэтому в C/C++ обязательно надо указывать размерности массивов при их выделении на графическом процессоре и при их обновлении на стороне графического и центрального процессоров.

На рис. 16 представлен фрагмент кода решения уравнения Пуассона, аналогичный тому, что был представлен на рис. 7. В отличае от кода на языке Фортран, здесь добавлена дополнительная data-секция вокруг ядра. Дело в том, что вместо копирования вновь вычисленных значений из массива **anew** в исходный массив, как это было реализовано на Фортране, используется перекидывание указателей (см. строки 49-51). Связывание имени массива на стороне центрального процессора и графического происходит только при в ходе в data-секцию и в случае, если мы опустим вложенную data-секцию на строке 36, мы поменяем указатели только на центральном процессоре, но не на стороне графического процессора.

```
30      #pragma acc data copy(a[0:M*N]) create(anew[0:M*N])
31      {
32      int iter=0;
33      while(err > tol && iter < iter_max) {
34          iter += 1;
35          err = 0;
36          #pragma acc data present(a,anew)
37          #pragma acc parallel reduction(max:err)
38          {
39          #pragma acc loop independent
40          for(int i = 1; i < M-1; i++) {
41              #pragma acc loop independent
42              for(int j = 1; j < N-1; j++) {
43                  anew[i*N+j] = 0.25*(a[(i+1)*N+j] + a[(i-1)*N+j]
44                                    + a[i*N+ j-1]  + a[i*N+j+1]);
45                  err = fmax(err, anew[i*N+j]-a[i*N+j]);
46              }
```

```
47          }
48          }
49          tmp = anew;
50          anew = a;
51          a = tmp;
52      } // while
53      } // acc data
```

Рис. 16. Пример разметки кода директивами для задачи решения уравнения Пуассона

Использование в коде указателей порой не дает возможности компилятору разобраться с наличием или отсутствием зависимости по данным между итерациями цикла. Разработчику рекомендуется использовать при необходимости ключевое слово **__restrict__** при объявлении переменных и условием **independent** директивы **loop**.

Стоит отметить особенности передачи на графичекий процессор массивов, членов класса. На рисунке ниже представлен представлен пример кода в котором на графическом процессоре происходит заполнение массива **a**, который является членом класса **foo**. В коде, представленном слева, в data-секцию передается непосредственно имя массива. Будучи запущенной, данная программа завершается с ошибкой, что некоторые переменные не присутствуют в памяти графического процессора. Для решения данной проблемы рекомендуется завести дополнительную локальную переменную и присвоить ей адрес массива **a**, как это было сделано в коде, представленном справа.

```
class foo{
  float * a;
  public:

  void run(){
#pragma acc data \
        copyout(a[0:1024])
     {
#pragma acc parallel loop \
        independent
     for(int i=0; i<1024; i++)
        a[i] = i;
     }
     for(int i=0; i<10; i++)
        cout<< a[i] << " ";
     cout << endl;
 }
```

```
class foo{
  float * a;
  public:

  void run(){
    float * a_ptr = a;
#pragma acc data \
        copyout(a_ptr[0:1024])
    {
#pragma acc parallel loop \
        independent
    for(int i=0; i<1024; i++)
       a_ptr[i] = i;
    }
    for(int i=0; i<10; i++)
       cout<< a[i] << " ";
    cout << endl;
  }
```

Рис. 17. Пример работающего (справа) и неработающего (слева) варианта программы.

- Проверка корректности расчетов».

- ***Ошибки синхронизации***. При использовании асинхронных операций разработчик может забыть поставить директиву **`wait`**.
- ***Ошибки компилятора***.
- ***Прочие ошибки***. Например, в сторонних библиотеках, программах и пр.

Часть из представленных ошибок выявить и локализовать довольно просто. Так, если каких-то данных нет на графическом процессоре, а пользователь указал соответствующие переменные в условии **`present`** директивы **`data`**, то программа во время исполнения завершится с распечаткой сообщения о том, какая переменная отсутствует, где данная ошибка случилась и какие переменные на текущий момент на графическом процессоре присутствуют. Далее остается проверить, откуда данный участок кода вызывается и почему переменная там отсутствует. Чаще всего данная ошибка возникает в случае, когда в рамках какой-либо функции программист помечает переменные как присутствующие на графическом процессоре, а в том месте, откуда эта функция вызывается, эти переменные реально выделяет или копирует на графический процессор. При этом нередко программист упускает из виду, что функция может вызываться откуда-либо еще, а там указанные переменные на графическом процессоре действительно отсутствуют. Подобное упущение может приводить к тому, что на графическом или центральном процессоре окажутся неактуальные данные, но это уже другая ошибка.

Ошибки синхронизации характеризуются тем, что: а) программа от запуска к запуску может выдавать разный результат, в том числе и корректный; б) до некоторого момента подобные ошибки могут в программе не проявляться. Если используются асинхронные операции для передачи данных или запуска ядер, то программу стоит проверить на данный тип ошибки. Для этого можно запустить программу, предварительно установив переменную окружения **`PGI_ACC_SYNCHRONOUS`**. Эта переменная заставляет все операции на графическом процессоре исполняться последовательно. Если при этом был получен корректный результат (и при многократных запусках он повторяется), то, скорее всего, это та самая ошибка. Далее необходимо аккуратно проанализировать все места, где используются асинхронные операции и где должна происходить синхронизация.

Ошибки, связанные с **ложным выявлением параллелизма** (ошибочное указание директивы **`parallel`** или условия **`loop independent`**), как правило, также локализуются достаточно быстро. Процесс переноса вычислений на графические процессоры является итеративным. Поэтому если программа стала выдавать неверный результат после внесения дополнительных директив, то, с большой вероятностью,

именно эти директивы и заставили компилятор создать некорректный код и на них стоит посмотреть более пристально. Общей рекомендацией здесь может быть следующее: сначала используйте директиву **kernels**, что заставит компилятор провести всесторонний анализ кода на наличие зависимостей, проанализируйте отчет о распараллеливании и лишь затем занимайтесь оптимизацией и помощью компилятору в распараллеливании кода.

Более сложными являются ошибки, связанные с **использованием неактуальных данных** на стороне графического или центрального процессора. Поиск подобных ошибок несколько усложняется еще и тем, что в компиляторе PGI начиная с версии от 2015 года условия **copy**, **copyin**, **copyout** директивы **acc data** интерпретируются как **present_or_copy**, **present_or_copyin**, **present_or_copyout** соответственно. Можно порекомендовать использовать условия **create** и **present** директивы **acc data** и явное обновление массивов с помощью директивы **acc update.** Выяснение, является ли некорректный результат работы программы результатом использования неактуальных данных, может вестись по следующим направлениям: а) анализ последних изменений в коде, б) отключение всех директив выделения и копирования данных. Во втором случае необходимо быть осторожным в случае наличия в функциях, в которых вызываются ядра, опциональных переменных. Для анализа перемещения данных может, в частности, использоваться результат вывода при запуске программы с установленной переменной окружения **PGI_ACC_DEBUG.** Второй вариант может быть наиболее трудоемким, однако, если исключение **data** секций и операций обновления не дало результата, то ошибка с большой вероятностью в чем-то другом.

Бывает, что некоторые ошибки не проявляются в последовательной версии программы на центральном процессоре, но всплывают в реализации OpenACC. Как правило, эти ошибки связаны с выходом за границы массива. Для частичного исключения этого типа ошибок рекомендуется выполнить сборку программы для центрального процессора с проверкой индекса при обращении к массивам (для компилятора PGI это опция **-C**). Следующим шагом можно рекомендовать запуск программы под управлением утилиты cuda-memcheck. Эта утилита для ядер на графическом процессоре проверяет корректность обращения к памяти.

Еще одним шагом к поиску места ошибки в программе может быть разбиение больших **kernel** секций на более мелкие.

Ошибки компилятора являются, пожалуй, самыми неприятными при разработке программ. Тем не менее, такие ошибки встречаются, и о них

стоит упомянуть. Здесь мы не рассматриваем ошибки, которые приводят к аварийному завершению компилятора, поскольку они легко детектируются.

Приведенный выше анализ кода должен позволить вам сузить поиски проблемного участка до одного ядра. Если цикл в **`kernel`** секции корректно выполняется на центральном процессоре, но при этом дает некорректный результат на графическом, у вас нет явного указания независимости итераций циклов, и вы уверены в актуальности данных, то, скорее всего, это ошибка компилятора.

Можно привести следующий алгоритм работы с такими ошибками:

1. Попробовать собрать программу последней версией компилятора.
2. Попробовать найти способ заставить код работать корректно.
   a. Заменить секцию **`kernels`** на **`parallel`** или наоборот. Поскольку компилятор немного по разному обрабатывает данные директивы, то код ядра может отличаться.
   b. Если сборка программы проводилась с ключом **`-g`** (включение отладочной информации), то можно попробовать убрать этот ключ, а если его не было, то добавить.
3. Выделить некорректно работающий код в отдельный файл и запустить его. В этом случае у вас готов тестовый пример, которым вы можете поделиться с разработчиками через форум (http://www.pgroup.com/userforum) и уточнить причину, по которой код на графическом и центральном процессорах дает разный результат.
4. Энтузиасты могут предложить попробовать собрать ключ с опцией **`keep`** к ключу **`-ta`** и изучить промежуточный код ядра, который был создан компилятором. В версии 2014 года и ранее компилятор создавал промежуточный код ядра на CUDA C.

Заключение

В пособии дан обзор технологии OpenACC и подходов к переносу вычислений на графические процессоры с использованием данной технологии. Для эффективного исполнения программы на графическом процессоре порой недостаточно снабдить компилятор указаниями. Требуется изменить реализацию алгоритма и/или выбрать алгоритм, который больше подходит для параллельных архитектур.

В заключение стоит добавить, что, используя компилятор PGI, можно скомпилировать программу для многоядерных систем, как если бы она была размечена директивами OpenMP.

Учебное издание

Романенко Алексей Анатольевич

ОСОБЕННОСТИ АДАПТАЦИИ ПРОГРАММ ПОД GPU С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ OPENACC

Методическое пособие

Редактор *Д.М. Валова*
Оригинал-макет *А. А.Романенко*
Обложка *Е. В. Неклюдовой*

Подписано в печать 24.02.2016 г.
Формат 60 x 84 1/16. Уч.-изд. л. 1,9. Усл. печ. л. 1,8.
Тираж 100 экз. Заказ № 15
Редакционно-издательский центр НГУ
630090, Новосибирск-90, ул. Пирогова, 2.