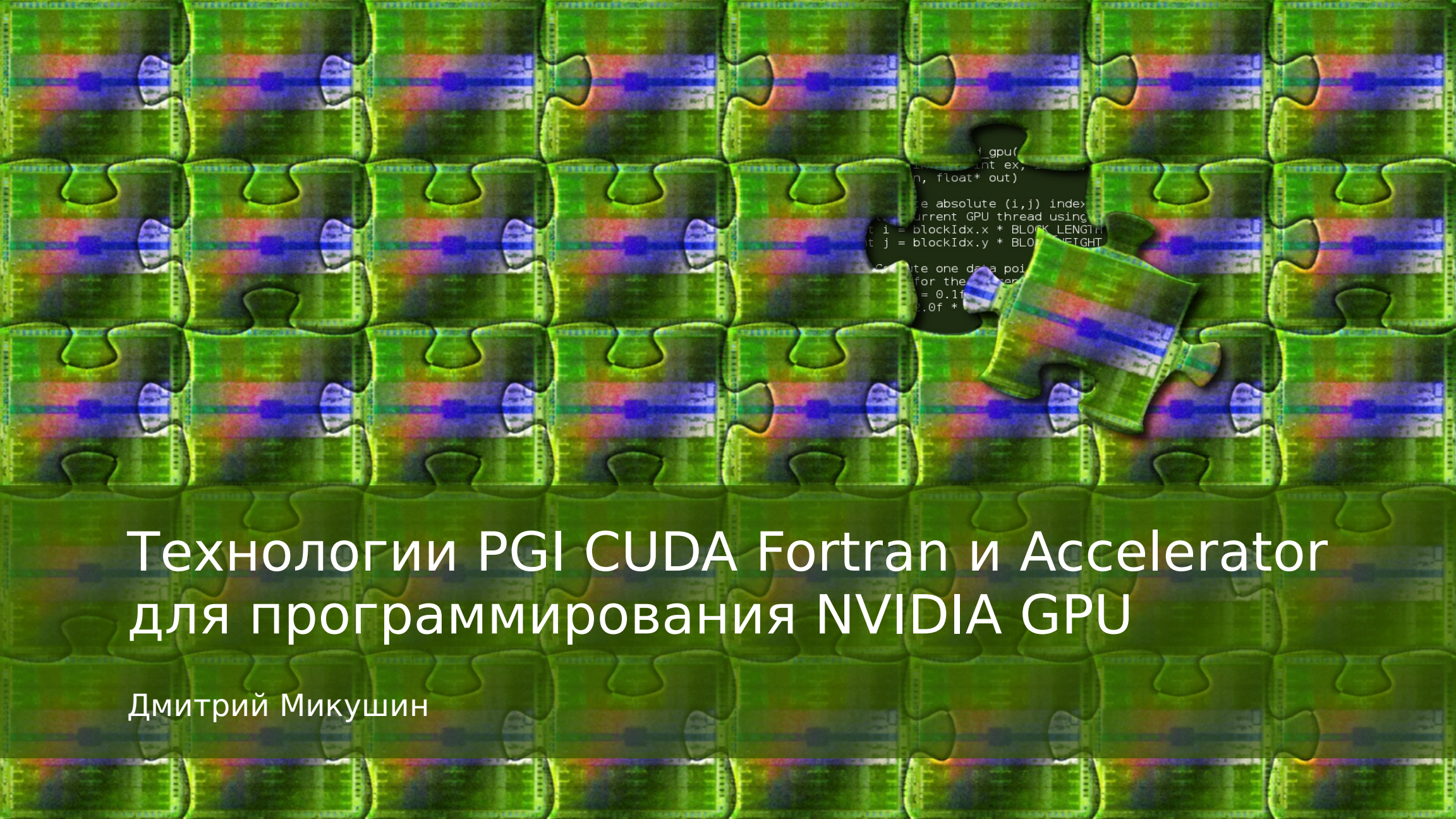

Технологии PGI CUDA Fortran и Accelerator для программирования NVIDIA GPU

Дмитрий Микушин

План

- **PGI CUDA Fortran**
  - Простейший пример программы
  - Программная модель CUDA Fortran
  - Low-level Programming with CUDA Fortran
  - Компиляция
- **PGI Accelerator**

PGI CUDA Fortran –

расширение языка Fortran для поддержки Compute Unified Device Architecture (CUDA), органично встраиваемое в существующие элементы языка, подобно тому как CUDA C/C++ расширяет язык C/C++

Пример vecadd (cpu)

```
subroutine cpu_vecadd(A,B,C,N)
  real(4) :: A(N), B(N), C(N)
  integer :: N
  integer :: i
  do i = 1,N
    C(i) = A(i) + B(i)
  enddo
end subroutine
```

Пример vecadd (gpu:device)

```
module gpu_vecadd_module
  use cudafor

  contains

  attributes(global) subroutine gpu_vecadd_kernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*32 + threadidx%x
    if( i <= N ) C(i) = A(i) + B(I)
  end subroutine

end module
```

Пример vecadd (gpu:host)

```
subroutine gpu_vecadd_host( A, B, C )
  use gpu_vecadd_module
  real(4), dimension(:) :: A, B, C
  real(4), device, allocatable:: Ad(:), Bd(:), Cd(:)
  integer :: N
  N = size( A, 1 )
  allocate( Ad(N), Bd(N), Cd(N) )
  Ad = A(1:N)
  Bd = B(1:N)
  call gpu_vecadd_kernel<<<(N + 31) / 32, 32>>>( Ad, Bd, Cd, N )
  C(1:N) = Cd
  deallocate( Ad, Bd, Cd )
end subroutine
```

Схема типовой хост-программы

- Выбор рабочего GPU (для multi-GPU систем)
- Выделение памяти на устройстве
- Копирование данных в память устройства
- Запуск ядер
- Копирование данных из памяти устройства
- Высвобождение памяти на устройстве

Пример vecadd (gpu:host)

```
subroutine gpu_vecadd_host( A, B, C )
  use gpu_vecadd_module
  real(4), dimension(:) :: A, B, C
  real(4), devic
  integer :: N
  N = size( A, 1 )
  allocate( Ad(N), Bd(N), Cd(N) )
  Ad = A(1:N)
  Bd = B(1:N)
  call gpu_vecadd_kernel<<<(N + 31) / 32, 32>>>( Ad, Bd, Cd, N )
  C(1:N) = Cd
  deallocate( Ad, Bd, Cd )
end subroutine
```

Выделение памяти на устройстве

Пример vecadd (gpu:host)

```
subroutine gpu_vecadd_host( A, B, C )
  use gpu_vecadd_module
  real(4), dimension(:) :: A, B, C
  real(4), device, allocatable:: Ad(:), Bd(:), Cd(:)
  integer :: N
  N = si
  allocat
  Ad = A(1:N)
  Bd = B(1:N)
  call gpu_vecadd_kernel<<<(N + 31) / 32, 32>>>( Ad, Bd, Cd, N )
  C(1:N) = Cd
  deallocate( Ad, Bd, Cd )
end subroutine
```

Копирование данных в память устройства

Пример vecadd (gpu:host)

```
subroutine gpu_vecadd_host( A, B, C )
  use gpu_vecadd_module
  real(4), dimension(:) :: A, B, C
  real(4), device, allocatable:: Ad(:), Bd(:), Cd(:)
  integer :: N
  N = size( A, 1 )
  allocate( Ad(N), Bd(N), Cd(N) )
  Ad = A(1:N)
  Bd = B
  call gpu_vecadd_kernel<<<(N + 31) / 32, 32>>>( Ad, Bd, Cd, N )
  C(1:N) = Cd
  deallocate( Ad, Bd, Cd )
end subroutine
```

Запуск вычислительного ядра на GPU

Пример vecadd (gpu:host)

```
subroutine gpu_vecadd_host( A, B, C )
  use gpu_vecadd_module
  real(4), dimension(:) :: A, B, C
  real(4), device, allocatable:: Ad(:), Bd(:), Cd(:)
  integer :: N
  N = size( A, 1 )
  allocate( Ad(N), Bd(N), Cd(N) )
  Ad = A(1:N)
  Bd = B(1:N)
  call gpu_vecadd_device( Ad, Bd, Cd, N )
  C(1:N) = Cd
  deallocate( Ad, Bd, Cd )
end subroutine
```

Копирование данных из памяти устройства

Пример vecadd (gpu:host)

```
subroutine gpu_vecadd_host( A, B, C )
  use gpu_vecadd_module
  real(4), dimension(:) :: A, B, C
  real(4), device, allocatable:: Ad(:), Bd(:), Cd(:)
  integer :: N
  N = size( A, 1 )
  allocate( Ad(N), Bd(N), Cd(N) )
  Ad = A(1:N)
  Bd = B(1:N)
  call gp
  C(1:N)
  deallocate( Ad, Bd, Cd )
end subroutine
```

Высвобождение памяти на устройстве

Особенности gpu-программы

- Все потоки исполняют один и тот же код
- Потоки объединены в сетку из блоков
- Сетка может быть 1-мерной или 2-мерной (максимум 65535 x 65535)
- Блок может быть 1-мерным, 2-мерным или трёхмерным (максимальный размер – 512 или 1024)
- Встроенная переменная blockidx – индекс блока (%x,%y)
- Встроенная переменная threadidx – индекс потока в блоке (%x,%y,%z)

Особенности gpu-программы

- За параллельное исполнение потоков отвечает драйвер и устройство
- Блоки потоков распределяются для исполнения внутри GPU между мультипроцессорами

Пример vecadd (gpu:device)

```
module gpu_vecadd_module
  use cudafor

  cont
  attributes(global) subroutine gpu_vecadd_kernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*32 + threadidx%x
    if( i <= N ) C(i) = A(i) + B(I)
  end subroutine

end module
```

Аттрибут “global” означает функция для GPU

Пример vecadd (gpu:device)

```
module gpu_vecadd_module
  use cudafor

  contains

  attributes(global) subroutine gpu_vecadd_kernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, val
    integer :: i
    i = (blockidx%x-1)*32 + threadidx%x
    if( i <= N ) C(i) = A(i) + B(I)
  end subroutine

end module
```

Массивы в памяти GPU

Пример vecadd (gpu:device)

```
module gpu_vecadd_module
  use cudafor

  contains

  attributes(global) subroutine gpu_vecadd_kernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    if( i <= N ) C(i) = A(i) + B(I)
  end subroutine

end module
```

Скаляр передан как значение (а не как адрес)

Пример vecadd (gpu:device)

```
module gpu_vecadd_module
  use cudafor

  contains

  attributes(global) subroutine gpu_vecadd_kernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*32 + threadidx%x
    if( i <= N
  end subroutine

end module
```

blockidx в диапазоне 1 .. (N + 31) / 32

Пример vecadd (gpu:device)

```
module gpu_vecadd_module
  use cudafor

  contains

  attributes(global) subroutine gpu_vecadd_kernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*32 + threadidx%x
    if( i <= N ) C(i) = A
  end subroutine

end module
```

threadidx в диапазоне 1 .. 32

Элементы языка CUDA Fortran

- **Хост-код**
  - Объявление и выделение памяти на GPU
  - Обмен данными между хостом и GPU
  - Pinned-память
  - Запуск GPU-ядер
- **GPU-код**
  - Аттрибуты
  - Процедуры-ядра и device-подпрограммы
  - Общая память
  - CUDA Runtime API

Объявление данных в памяти GPU

- Переменные и массивы с аттрибутом “device” выделяются в памяти GPU:

```
real, device, allocatable :: a(:)
real, allocatable :: a(:)
attributes(device) :: a
```

- В коде хост-программы:
  - Могут быть объявленны allocatable и автоматические переменные в памяти GPU
  - Переменные и массивы в GPU памяти могут быть переданы в другие подпрограммы хост-кода и процедуры-ядра

Данные в памяти GPU и модули

- Данные для GPU в модулях должны иметь фиксированный размер или быть allocatable:

```
module mm
  real, device, allocatable :: a(:)
  real, device :: x, y(10)
  real, constant :: c1, c2(10)
  integer, device :: n
  contains

    attributes(global) subroutine s( b )

...
```

- Переменная или массив может иметь аттрибут “constant”, обозначающий размещение в константной памяти GPU

Выделение памяти на GPU

- Оператор allocate / deallocate

```
real, device, allocatable :: a(:,:), b
allocate( a(1:n,1:m), b )
...
deallocate( a, b )
```

- Память для массивов и переменных с аттрибутом “device” выделяется на GPU:
  - Выделение памяти производит хост-подпрограмма
  - Виртуальной памяти нет, т.е. свободная память GPU может закончиться
  - Необязательный аргумент STAT=ivar может быть использован для контроля ошибок

Копирование данных между хостом и GPU (Fortran-стиль)

- Копирование с помощью оператора присваивания:

```
real, device, allocatable :: a(:,:), b
allocate( a(1:n,1:m), b )
a(1:n,1:m) = x(1:n,1:m)
! copies to device
b = 99.0
...
x(1:n,1:m) = a(1:n,1:m)
! copies from device
y = b
deallocate( a, b )
```

- Копирование разрывных диапазонов может быть медленее, т.к. приводит к копированию отдельно каждой строки

Копирование данных между хостом и GPU (CUDA API)

- В модуле cudafor определены функции, идентичные CUDA API:

```
use cudafor
real, allocatable, device :: a(:)
real :: b(10), b2(2), c(10)
...
istat = cudaMalloc( a, 10 )
istat = cudaMemcpy( a, b, 10 )
istat = cudaMemcpy( a(2), b2, 2 )
istat = cudaMemcpy( c, a, 10 )
istat = cudaFree( a )
```

Запуск GPU-ядер

- Вызов процедуры-ядра с конфигурацией вычислительной сети в тройных угловых скобках:

```
call gpu_vecadd_kernel <<< (N+31)/32, 32 >>> ( A, B, C, N )
type(dim3) :: g, b
g = dim3((N+31)/32, 1, 1)
b = dim3( 32, 1, 1 )
call gpu_vecadd_kernel <<< g, b >>> ( A, B, C, N )
```

- Процедура-ядро должна иметь явный интерфейс или находиться в используемом модуле или в том же модуле что и вызывающая функция
- Конфигурация вычислительной сети: размеры сетки и блока могут быть целыми числами или значениями типа dim3

GPU-ядра на CUDA Fortran

- Аттрибут “global” для процедур-ядер:

```
attributes(global) subroutine kernel ( A, B, C, N )
```

- В ядре можно объявлять скаляры и массивы фиксированной длины
- В ядре можно объявлять массивы в shared-памяти (памяти, разделяемой потоками одного блока):

```
real, shared :: sm(16,16)
```

- В ядре поддеживается использование основных встроенных типов данных, а также производных типов (структур)

```
integer(1,2,4,8), logical(1,2,4,8), real(4,8), complex(4,8),
character(len=1)
```

GPU-ядра на CUDA Fortran

- Встроенные переменные: *blockidx, threadidx, griddim, blockdim, warpsize*
- Поддержка в ядре следующего подмножества конструкций языка:
  - Оператор присваивания
  - *do, if, goto, case*
  - *call* (подпрограмм с аттрибутом “device”)
  - intrinsics
  - *where, forall*

Области видимости

- Процедура-ядро с аттрибутом “global” в модуле:
  - Может напрямую обращаться к данным в памяти GPU, объявленным в этом же модуле
  - Может вызывать функции и процедуры с аттрибутом “device”, определённые в этом же модуле

- Процедура или функция с аттрибутом “device” в модуле:
  - Может напрямую обращаться к данным в памяти GPU, объявленным в этом же модуле
  - Может вызывать функции и процедуры с аттрибутом “device”, определённые в этом же модуле
  - Неявно опередедена как приватная

Области видимости

- Процедура-ядро с аттрибутом “global” вне модуля:
  - Не может обращаться к данным в памяти GPU, за исключением аргументов
- Процедура или функция, выполняемая на хосте:
  - Может вызывать процедуры-ядра, определённые как в модулях, так и вне модулей
  - Может обращаться к любым данным в модулях
  - Может вызывать ядра на CUDA C при наличии интерфейса

Взаимодействие с CUDA C

- C → Fortran: интерфейс

```
interface
  attributes(global) subroutine saxpy(a,x,y,n) bind(c)
    real, device :: x(*), y(*)
    real, value :: a
    integer, value :: n
  end subroutine
end interface
...
call saxpy<<<grid,block>>>( aa, xx, yy, nn )
```

- Fortran → C: прототип

```
extern __global__ void saxpy_(
  float a, float* x, float* y, int n );
...
saxpy ( a, x, y, n );
```

Компиляция

- Сборка объектов с ядрами на CUDA Fortran:

```
pgfortran -Mcuda[=[emu|cc10|cc11|cc12|cc13|cc20]] a.cuf
```

  - Суффикс .cuf ~ CUDA Fortran (free form)
  - Суффикс .CUF ~ + препроцессор
  - -Mfixed ~ fixed form (Fortran 77)

- При линковке объектов также необходима опция -Mcuda
- Необходим установленный CUDA Toolkit (nvcc, ...)

CUDA C vs CUDA Fortran

CUDA C

- Текстуры, текстурная память
- Поддержка CUDA Runtime API
- Поддержка Driver API
- cudaMalloc, cudaFree
- cudaMemcpy
- OpenGL, Direct3D
- Индексация элементов массивов и блоков/потоков – с нуля
- Девайс-функции и специализированные библиотеки

CUDA Fortran

- Текстурная память недоступна
- Поддержка CUDA Runtime API
- Driver API недоступен
- allocate, deallocate
- Оператор присваивания
- OpenGL и Direct3D недоступны
- Индексация элементов массивов и блоков/потоков – с единицы
- Модули с коллекциями функций, например use cublas

Краткая форма

```
!$cuf kernel do[(n)] <<< grid, block >>>
      <блок кода>
```

Fortran

```
!$cuf kernel do(2) <<< (*,*), (32,4) >>>
do j = 1, m
  do i = 1, n
    a(i,j) = b(i,j) + c(i,j)
  end do
end do
```

- Двумерный цикл отображается на двумерную сетку блоков 32x4
- Размерность сетки потоков (*, *) вычисляется во время исполнения делением размерностей циклов на размерности сетки блоков

GPU-ядра в явном виде или генерация ядер компилятором?

- Программирование GPU-ядер вручную:
  - \+ Высокая производительность при тонкой настройке
  - – Работает только с CUDA, фрагментация версий исходного кода

GPU-ядра в явном виде или генерация ядер компилятором?

Компиляторы могут генерировать GPU-ядра из оригинального хост-кода **неявно**, полностью автоматически или по аннотациям/директивам

GPU-ядра в явном виде или генерация ядер компилятором?

- Неявная генерация GPU-ядер:
  - \+ Хорошая производительность возможна
  - \+ Код остаётся полностью совместимым с версией для CPU
  - – Сложно контролировать работу

PGI Accelerator –

набор директив компиляции для языков Fortran и C, позволяющих автоматически транслировать в CUDA-код и выполнять на NVIDIA GPU выделенные фрагменты хост-кода

Идея

Фрагменты обычного хост-кода программы помещаются в окружение директивы *acc region*:

!$acc region

<код для исполнения на GPU>

!$acc end region

Код компилируется с флагом использования ускорителя:

```
pgfortran -fast -Minfo=accel -ta=nvidia
```

Пример программы

```
subroutine sincos(nx, ny, nz)

implicit none

integer :: nx, ny, nz
real, allocatable, dimension(:,:,:) :: x, y, xy1, xy2

integer :: i, j, k
real :: start, finish

allocate(x(nx, ny, nz))
allocate(y(nx, ny, nz))
allocate(xy1(nx, ny, nz))
allocate(xy2(nx, ny, nz))
```

Пример программы

```
! Fill input arrays with random values
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      call random_number(x(i,j,k))
      call random_number(y(i,j,k))
    enddo
  enddo
enddo

call cpu_time(start)

! PGI Accelerator region
!$acc region
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy1(i,j,k) = sin(x(i,j,k)) + cos(y(i,j,k))
    enddo
  enddo
enddo
!$acc end region

call cpu_time(finish)
```

Пример программы

```
print *, 'acc time = ', finish - start

call cpu_time(start)

! Control CPU implementation
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy2(i,j,k) = sin(x(i,j,k)) + cos(y(i,j,k))
    enddo
  enddo
enddo

call cpu_time(finish)

print *, 'cpu time = ', finish - start

! Compare results
print *, 'diff = ', maxval(xy1 - xy2)

deallocate(x)
deallocate(y)
deallocate(xy1)
deallocate(xy2)

end subroutine sincos
```

Пример программы

```
[marcusmae@noisy pgiacc]$ make
pgfortran -fast -Minfo=accel -ta=nvidia sincos.f90 -o sincos
sincos:
     29, Generating copyin(y(1:nx,1:ny,1:nz))
         Generating copyin(x(1:nx,1:ny,1:nz))
         Generating copyout(xy1(1:nx,1:ny,1:nz))
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
     30, Loop is parallelizable
     31, Loop is parallelizable
     32, Loop is parallelizable
```

Пример программы

```
[marcusmae@noisy pgiacc]$ make
pgfortran -fast -Minfo=accel -ta=nvidia sincos.f90 -o sincos
sincos:
     29, Generating copyin(y(1:nx,1:ny,1:nz))
         Generating copyin(x(1:nx,1:ny,1:nz))
         Generating copyout(xy1(1:nx,1:ny,1:nz))
         Generating compute capability 2.0 binary
     30, Loop is parallelizable
     31, Loop is parallelizable
     32, Loop is parallelizable
```

Компилятор определяет, как нужно копировать данные

Пример программы

```
[marcusmae@noisy pgiacc]$ make
pgfortran -fast -Minfo=accel -ta=nvidia sincos.f90 -o sincos
sincos:
     29, Generating copyin(y(1:nx,1:ny,1:nz))
         Generating copyin(x(1:nx,1:ny,1:nz))
         Generating copyout(xy1(1:nx,1:ny,1:nz))
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
     30, Loop is parallelizable
     31, Loop is parallelizable
     32, Loop is parallelizable
```

Компилятор определяет параллельные циклы

Пример программы

```
          Accelerator kernel generated
          30, !$acc do parallel, vector(4) ! blockidx%y threadidx%z
          31, !$acc do parallel, vector(4) ! blockidx%x threadidx%y
          32, !$acc do vector(16) ! threadidx%x
              CC 1.0 : 17 registers; 76 shared, 136 constant, 56
local memory bytes; 33% occupancy
              CC 1.3 : 17 registers; 76 shared, 136 constant, 56
local memory bytes; 75% occupancy
              CC 2.0 : 25 registers; 8 shared, 152 constant, 4 local
memory bytes; 66% occupancy
pgcc -c main.c -o main.o
pgfortran -Mnomain -Minfo=accel -ta=nvidia sincos.o main.o -o
sincos

[marcusmae@noisy pgiacc]$ ./sincos
 acc time =    0.1568580
 cpu time =    0.5031471
 diff =    2.3841858E-07
```

Пример программы

```
          Accelerator kernel generated
          30,  !$acc do parallel, vector(4) ! blockidx%y threadidx%z
          31,  !$acc do parallel, vector(4) ! blockidx%x threadidx%y
```

Два внешних цикла отображаются на сетку блоков и потоков

```
local memory bytes; 33% occupancy
              CC 1.3 : 17 registers; 76 shared, 136 constant, 56
local memory bytes; 75% occupancy
              CC 2.0 : 25 registers; 8 shared, 152 constant, 4 local
memory bytes; 66% occupancy
pgcc -c main.c -o main.o
pgfortran -Mnomain -Minfo=accel -ta=nvidia sincos.o main.o -o
sincos

[marcusmae@noisy pgiacc]$ ./sincos
 acc time =    0.1568580
 cpu time =    0.5031471
 diff =    2.3841858E-07
```

Пример программы

```
          Accelerator kernel generated
          30, !$acc do parallel, vector(4) ! blockidx%y threadidx%z
          31, !$acc do parallel, vector(4) ! blockidx%x threadidx%y
          32, !$acc do vector(16) ! threadidx%x
```

Внутренний цикл отображается на сетку потоков

```
local m
              CC 1.3 : 17 registers; 76 shared, 136 constant, 56
local memory bytes; 75% occupancy
              CC 2.0 : 25 registers; 8 shared, 152 constant, 4 local
memory bytes; 66% occupancy
pgcc -c main.c -o main.o
pgfortran -Mnomain -Minfo=accel -ta=nvidia sincos.o main.o -o
sincos

[marcusmae@noisy pgiacc]$ ./sincos
 acc time =    0.1568580
 cpu time =    0.5031471
 diff =    2.3841858E-07
```

Пример программы

```
          Accelerator kernel generated
          30, !$acc do parallel, vector(4) ! blockidx%y threadidx%z
          31, !$acc do parallel, vector(4) ! blockidx%x threadidx%y
          32, !$acc do vector(16) ! threadidx%x
              CC 1.0 : 17 registers; 76 shared, 136 constant, 56
local memory bytes; 33% occupancy
              CC 1.3 : 17 registers; 76 shared, 136 constant, 56
local memory bytes; 75% occupancy
              CC 2.0 : 25 registers; 8 shared, 152 constant, 4 local
memory bytes; 66% occupancy
```

Генерируются ядра для GPU различных compute capabilities

```
sincos

[marcusmae@noisy pgiacc]$ ./sincos
 acc time =    0.1568580
 cpu time =    0.5031471
 diff =    2.3841858E-07
```

Пример программы

```
          Accelerator kernel generated
          30, !$acc do parallel, vector(4) ! blockidx%y threadidx%z
          31, !$acc do parallel, vector(4) ! blockidx%x threadidx%y
          32, !$acc do vector(16) ! threadidx%x
              CC 1.0 : 17 registers; 76 shared, 136 constant, 56
local memory bytes; 33% occupancy
              CC 1.3 : 17 registers; 76 shared, 136 constant, 56
local memory bytes; 75% occupancy
              CC 2.0 : 25 registers; 8 shared, 152 constant, 4 local
memory bytes; 66% occupancy
pgcc -c main.c -o main.o
pgfortran -Mnomain -Minfo=accel -ta=nvidia sincos.o main.o -o
sinc
```

Сравнение результатов: Tesla C2070 и AMD 1055T

```
[marcusmae@noisy pgiacc]$ ./sincos
 acc time =    0.1568580
 cpu time =    0.5031471
 diff =    2.3841858E-07
```

Пример программы

- PGI Accelerator автоматически генерирует ядро для NVIDIA GPU
- Компилятор выбирает размеры блоков потоков
- Каждый поток в блоке вычисляет одно значение выходного массива
- При доступе к массивам x и y shared-память не используется
- Один бинарный файл может содержать реализации этого фрагмента как для многоядерных CPU, так и ядро для GPU

Модель исполнения

- Хост
  - По-прежнему исполняет большую часть кода
  - Выделяет память на GPU
  - Управляет передачей данных и кода на GPU
  - Управляет запуском ядер и синхронизацией
  - Загружает результаты из памяти GPU
  - Высвобождает память на GPU
- GPU
  - Выполняет ядра одно за другим
  - Возможна асинхронная передача данных (zero-copy)

Процесс компиляции

```
void saxpy (float a,
    float *restrict x,
    float *restrict y, int n) {
    #pragma acc region
    {
        for (int i=1; i<n; i++)
            x[i] = a*x[i] + y[i];
    }
}
```

Host - код

```
extern "C" __global__ void pgi_kernel_2() {
    int i1, i1s, ibx, itx;
    ibx = blockIdx.x;
    itx = threadIdx.x;
    for (i1s = ibx*256; i1s < a2.tc1;
        i1s += gridDim.x*256 ){
        i1 = itx + i1s;
        if( i1 < a2.tc1 ){
            a2._x[i1] = (a2._y[i1] +
                (a2._x[i1]*a2._a));
        }
    }
}
```

Код для GPU (pgcc –ta=nvidia)

Единый объектный файл

acc data region

```
#pragma acc data region [clause [, clause]...]
    <блок кода>
```

C

```
!$acc data region [clause [, clause]...]
    <блок кода>
!$acc end data region
```

Fortran

clause:

copy(list)	local(list)	update device(list)
copyin(list)	mirror (list)	update host(list)
copyout(list)		

Ресурсы

- Michael Wolfe:
  GPU Programming with CUDA (C and PGI CUDA Fortran)
- Dave Norton:
  PGI Accelerator Compilers for x64+GPU Systems
- PGI Fortran & C Accelerator Programming Model