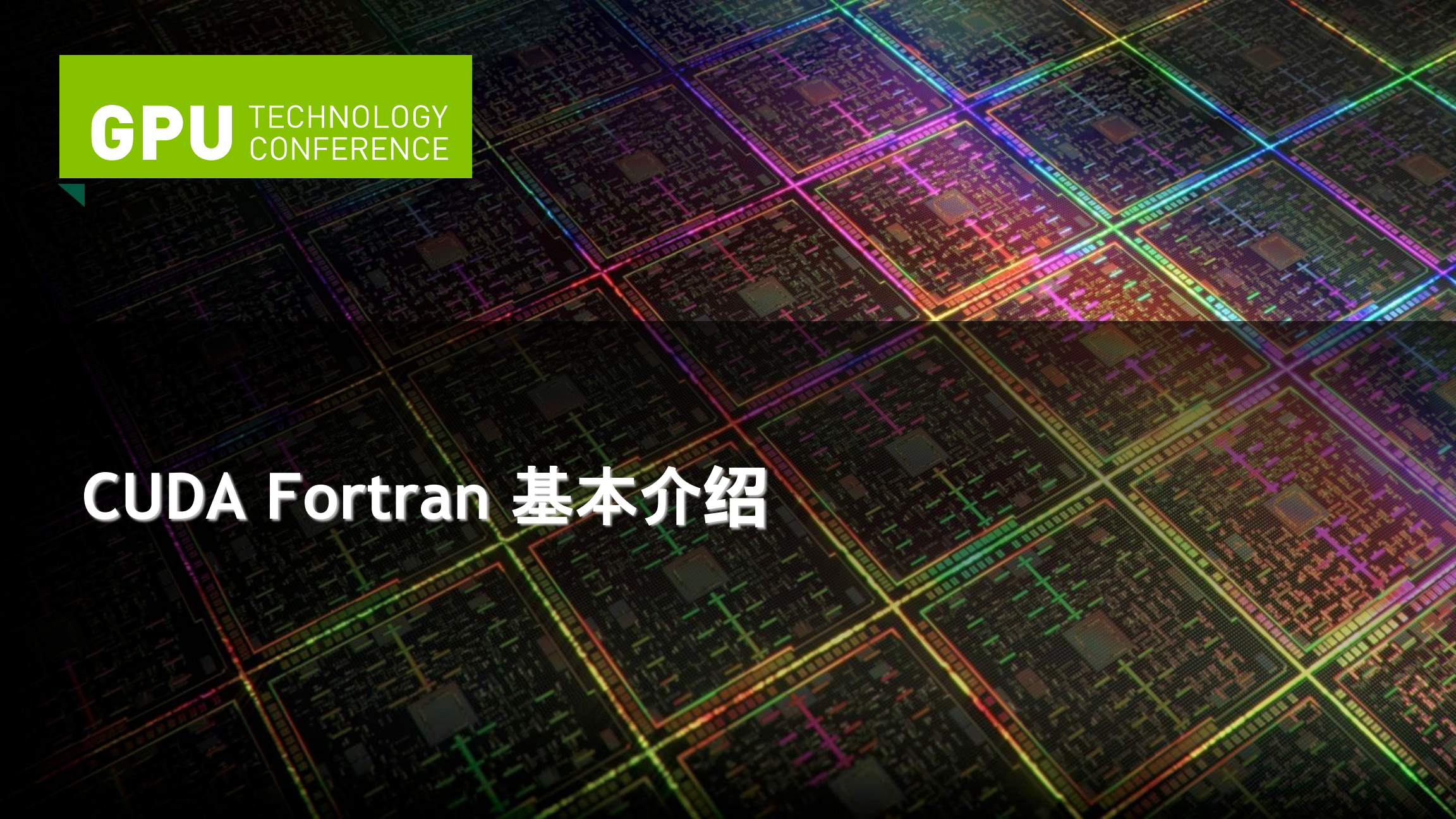
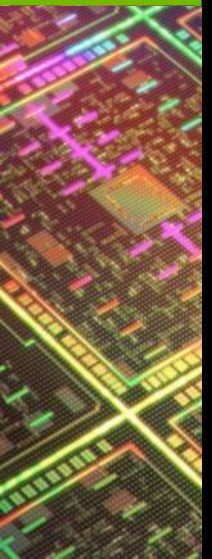


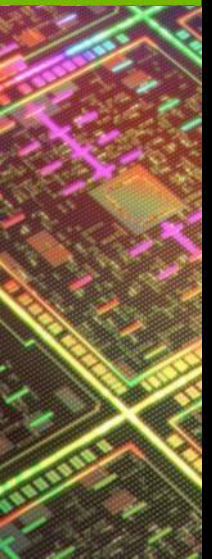
CUDA Fortran 基本介绍

CUDA Fortran

- CUDA：使用 GPU 硬件资源实现高性能计算的计算平台
  - 硬件：支持 CUDA 的 GPU https://developer.nvidia.com/cuda-gpus
  - 软件：PGI 的 CUDA Fortran 编译器

- CUDA Fortran VS CUDA C
  - 与 CUDA C 类似，CUDA Fortran 也分主机端代码和设备端代码
  - CUDA Fortran 主机端代码在 CPU 上执行
  - CUDA Fortran 设备端代码在 GPU 上执行
  - 与 CUDA C 相比，CUDA Fortran 中 CPU 与 GPU 之间内存管理更加简单





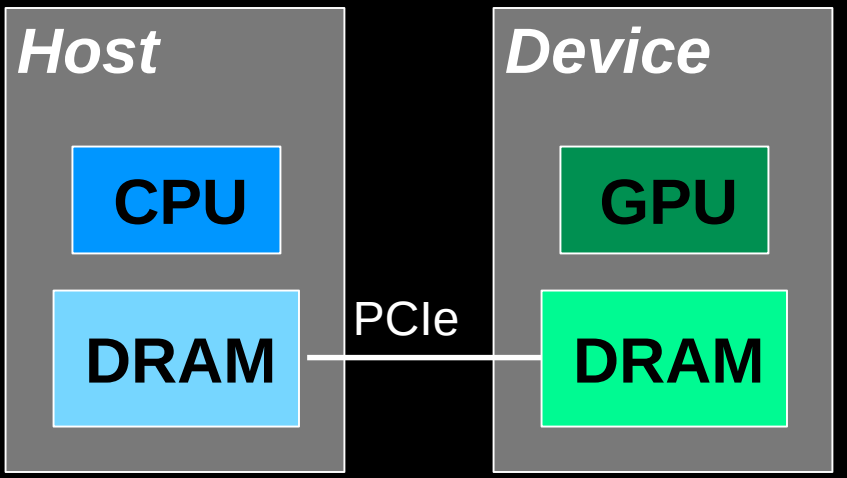
CUDA Fortran 编程

- 异构编程模型
  - CPU 和 GPU 是两个独立的硬件设备，有各自的物理内存空间。相应地
  - 在 CPU 上执行的代码，即主机端代码
    - CPU 与 GPU 之间的内存管理
    - 启动在 GPU 上执行的函数 (Kernel)
  - 在 GPU 上执行的代码，即设备端代码
    - GPU 上，会开启大量的线程，完成设备端的计算
    - GPU Kernel 的执行模式为异步执行。即 Kernel 在 CPU 端启动后，不必等待 Kernel 执行完成，控制立即返回 CPU
  - 充分利用 GPU 和 CPU 的资源

CUDA Fortran 编程

- Host （主机）= CPU 和它的内存
- Device （设备）= GPU 和它的内存

- CUDA Fortran 三步执行流程
  - 分别在 CPU 和 GPU 上申请内存，把 CPU 端准备好的数据，从 CPU 端拷贝到 GPU 端
  - 从 CPU 端启动在 GPU 端执行的 kernel，完成并行计算
  - 将计算结果从 GPU 拷贝回 CPU 端。最后记得释放内存空间



F90 Array Increment 例子

```
module simpleOps_m
contains
  subroutine inc(a, b)
    implicit none
    integer :: a(:)
    integer :: b
    integer :: i, n

    n = size(a)
    do i = 1, n
       a(i) = a(i)+b
    enddo

  end subroutine inc
end module simpleOps_m
```

```
program incTest
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable:: a(:)

  allocate (a(n))
  a = 1   ! array assignment
  b = 3
  call inc(a, b)

  if (all(a == 4)) &
     write(*,*) 'Test Passed'

  deallocate(a)
end program incTest
```

F90 Array Increment 例子

```
module simpleOps_m
contains
  subroutine inc(a, b)
    implicit none
    integer :: a(:)
    integer :: b
    integer :: i, n

    n = size(a)
    do i = 1, n
       a(i) = a(i)+b
    enddo

  end subroutine inc
end module simpleOps_m
```

```
program incTest
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable:: a(:)

  allocate (a(n))
  a = 1   ! array assignment
  b = 3
  call inc(a, b)

  if (all(a == 4)) &
     write(*,*) 'Test Passed'

  deallocate(a)
end program incTest
```

F90 Array Increment 例子

```
module simpleOps_m
contains
  subroutine inc(a, b)
    implicit none
    integer :: a(:)
    integer :: b
    integer :: i, n

    n = size(a)
    do i = 1, n
       a(i) = a(i)+b
    enddo

  end subroutine inc
end module simpleOps_m
```

```
program incTest
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable:: a(:)

  allocate (a(n))
  a = 1   ! array assignment
  b = 3
  call inc(a, b)

  if (all(a == 4)) &
     write(*,*) 'Test Passed'

  deallocate(a)
end program incTest
```

CUDA Fortran - 主机端代码

F90

```
program incTest

  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable:: a(:)


  allocate (a(n))
  a = 1   ! array assignment
  b = 3


  call inc(a, b)


  if (all(a == 4)) &
     write(*,*) 'Test Passed'
  deallocate(a)

end program incTest
```

CUDA Fortran

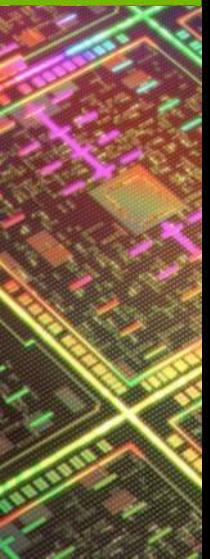
```
program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable :: a(:)
  integer, allocatable, device :: a_d(:)

  allocate (a(n),a_d(n))
  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) &
     write(*,*) 'Test Passed'
  deallocate (a,a_d)

end program incTest
```

CUDA Fortran - 主机端代码

CUDA Fortran

```
program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable :: a(:)
  integer, allocatable, device :: a_d(:)

  allocate (a(n),a_d(n))
  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) &
     write(*,*) 'Test Passed'
  deallocate (a,a_d)

end program incTest
```

cudafor 定义了 CUDA runtime API



CUDA Fortran - 主机端代码

CUDA Fortran

```
program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable :: a(:)
  integer, allocatable, device :: a_d(:)

  allocate (a(n),a_d(n))
  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) &
     write(*,*) 'Test Passed'
  deallocate (a,a_d)

end program incTest
```

device 修饰的变量：在设备端分配内存
变量的默认属性 host，如 a，在主机端分配内存

CUDA Fortran - 主机端代码

CUDA Fortran

```
program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable :: a(:)
  integer, allocatable, device :: a_d(:)

  allocate (a(n),a_d(n))
  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) &
     write(*,*) 'Test Passed'
  deallocate (a,a_d)

end program incTest
```

allocate 在分别在主机端和设备端为 a 和 a_d 申请内存

CUDA Fortran - 主机端代码

CUDA Fortran

```
program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable :: a(:)
  integer, allocatable, device :: a_d(:)

  allocate (a(n),a_d(n))
  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) &
     write(*,*) 'Test Passed'
  deallocate (a,a_d)

end program incTest
```

a_d = a 赋值语句，完成了数据从 CPU 到 GPU 的拷贝

CUDA Fortran - 主机端代码

CUDA Fortran

```
program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable :: a(:)
  integer, allocatable, device :: a_d(:)

  allocate (a(n),a_d(n))
  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) &
     write(*,*) 'Test Passed'
  deallocate (a,a_d)

end program incTest
```

call inc<<<1, n>>>: 在主机端启动设备端kernel.
注意，由于 kernel 是异步执行。所以 kernel 启动后，控制立即返回到主机端
<<<1, n>>>: 后面再详细的介绍

CUDA Fortran - 主机端代码

CUDA Fortran

```
program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable :: a(:)
  integer, allocatable, device :: a_d(:)

  allocate (a(n),a_d(n))
  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) &
     write(*,*) 'Test Passed'
  deallocate (a,a_d)

end program incTest
```

a= a_d 赋值语句，计算结果从 GPU 拷贝到 CPU

CUDA Fortran - 主机端代码

CUDA Fortran

```
program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable :: a(:)
  integer, allocatable, device :: a_d(:)

  allocate (a(n),a_d(n))
  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) &
     write(*,*) 'Test Passed'
  deallocate (a,a_d)

end program incTest
```

释放 CPU 和 GPU 内存

CUDA Fortran - 设备端代码

F90

```
module simpleOps_m
contains
  subroutine inc(a, b)
    implicit none
    integer :: a(:)
    integer :: b
    integer :: i, n

    n = size(a)
    do i = 1, n
       a(i) = a(i)+b
    enddo

  end subroutine inc
end module simpleOps_m
```

CUDA Fortran

```
module simpleOps_m
contains
  attributes(global) subroutine inc(a, b)
    implicit none
    integer :: a(:)
    integer, value :: b
    integer :: i

    i = threadIdx%x
    a(i) = a(i)+b


  end subroutine inc
end module simpleOps_m
```

CUDA Fortran - 设备端代码

CUDA Fortran

```
module simpleOps_m
contains
  attributes(global) subroutine inc(a, b)
    implicit none
    integer :: a(:)
    integer, value :: b
    integer :: i

    i = threadIdx%x
    a(i) = a(i)+b



  end subroutine inc
end module simpleOps_m
```

global 关键字修饰的函数 inc:

该函数在 CPU 启动，在 GPU 上执行

通常把它称为 kernel. Kernel 执行为异步模式

其它的函数修饰符有

device: 在设备端启动，在设备上执行

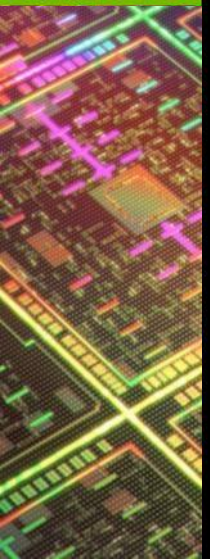
host ：在主机端启动，在主机上执行

*** global 函数必须在 module 中定义

编译运行 CUDA Fortran 程序

- 使用 pgi 的 fortran 编译器：pgf90/pgfortran
- CUDA Fortran 程序的后缀 .cuf

```
$ pgf90 increment.cuf
$ ./a.out
 Program Passed
$
```



<<<X1, X2>>> (eg. <<<1, n>>>)

- 用于 kernel 运行的并行配置

```
call inc<<<1,n>>>(a_d,b)
```

  - <<<1, n>>>: 发启了1个 线程块 block, 每个 block 有 n 个线程 thread
  - 线程块是线程的集合;线程块内的线程用线程 ID threadIdx%x 标记, 从 1 开始编号

```
i = threadIdx%x
a(i) = a(i)+b
```

thread 1	thread 2	thread 3	thread 4
a[1] = a[1] + b;	a[2] = a[2] + b;	a[3] = a[3] + b;	a[4] = a[4] + b;

<<<X1, X2>>>

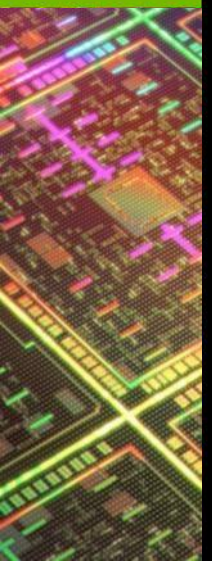
- 用于 kernel 运行的并行配置

```
call inc<<<n, 1>>>(a_d,b)
```

  - <<<1, n>>>: 发启了 n 个 线程块 block, 每个 block 有 1 个线程 thread
  - 所有线程块的集合称为 Grid;
  - Grid 内不同线程块用线程块 ID blockIdx%x 标记，从 1 开始编号

```
i = blockIdx%x
a(i) = a(i)+b
```

线程在 GPU 上执行模型



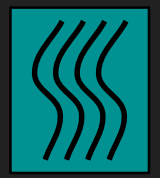
Software

Thread

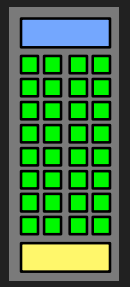
Hardware

Thread Processor

线程在 CUDA 计算核心上执行



Thread Block

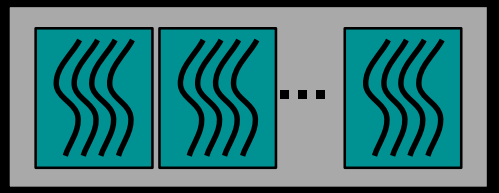


Multiprocessor

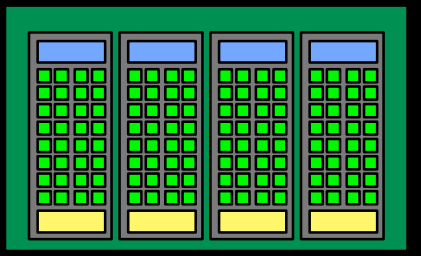
线程块 block 在 SM 上执行

一般情况下，线程块执行过程中，不会在不同的 SM 间跳转

多个并发的线程块可以在同一个 SM 执行，即线程块与SM 是多对一的关系



Grid



Device

在 CPU 端启动 kernel，会在设备端启动一个 Grid.

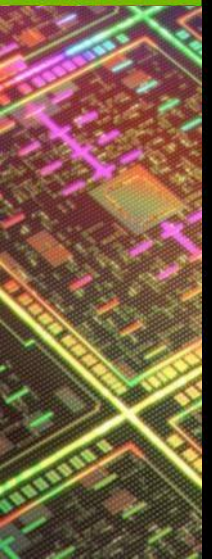
事实上，Grid 和 Block 是 GPU 上线程的两层组织结构

线程块 Block 和 栅格 Grid

- 线程块 Block
  - 线程块是线程的集合，线程块中不同线程用线程 ID threadIdx 区分
  - 同一个 block 最多能包含的线程数有限制（Kepler，1024）
  - 内建变量 threadIdx 是包含三个成员的结构体，即支持 1D，2D 和 3D 的线程块配置，以方便线程与任务的映射
  - 同一个线程块内，不同的线程间可以通过共享内存（shared）共享数据
  - 同一个线程块内，所有线程可以通过 syncthreads() 实现线程同步

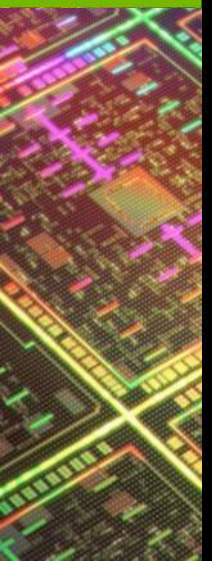
线程块 Block 和 栅格 Grid

- 栅格 Grid
  - Grid 是线程块的集合；Grid 的不同线程块之间用线程块 ID blockIdx 区分
  - Grid 最多能包含的线程块数目有限制，取决于具体的 GPU 硬件
  - 内建变量 blockIdx 也是包含三个成员的结构体，即支持 1D，2D 和 3D 的线程块配置，以方便线程与任务的映射
  - 不同的 block 之间，不能通过共享内存共享数据。因为它们可能在不同的 SM 上执行
  - 不同 block 的线程，不能同步
  - 正是由于 block 线程之间有这样的限制。通常我们在设计 kernel 时，尽量把 block 开得比较大（<<<1, n>>>，而非 <<<n, 1>>>），然后再设计 Grid 的大小



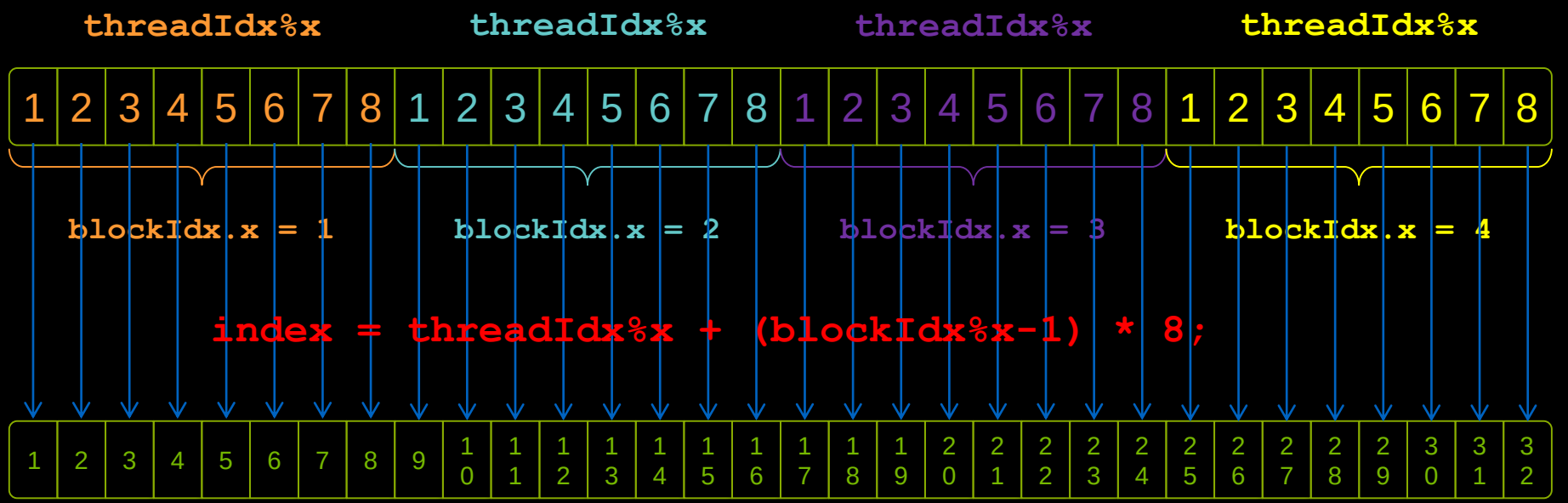
N 非常大，主机端代码

```
program incrementTest
  use cudafor
  use simpleOps_m
  integer, parameter :: n = 1024*1024
  integer, allocatable :: a(:)
  integer, device, allocatable :: a_d(:)
  integer :: b, tPB = 256

  allocate(a(n), a_d(n))

  a = 1; b = 3
  a_d = a
  call increment<<<ceiling(real(n)/tPB),tPB>>>(a_d, b)
  a = a_d

  if (any(a /= 4)) then
     write(*,*) '**** Program Failed ****'
  else
     write(*,*) 'Program Passed'
  endif
  deallocate(a, a_d)
end program incrementTest
```

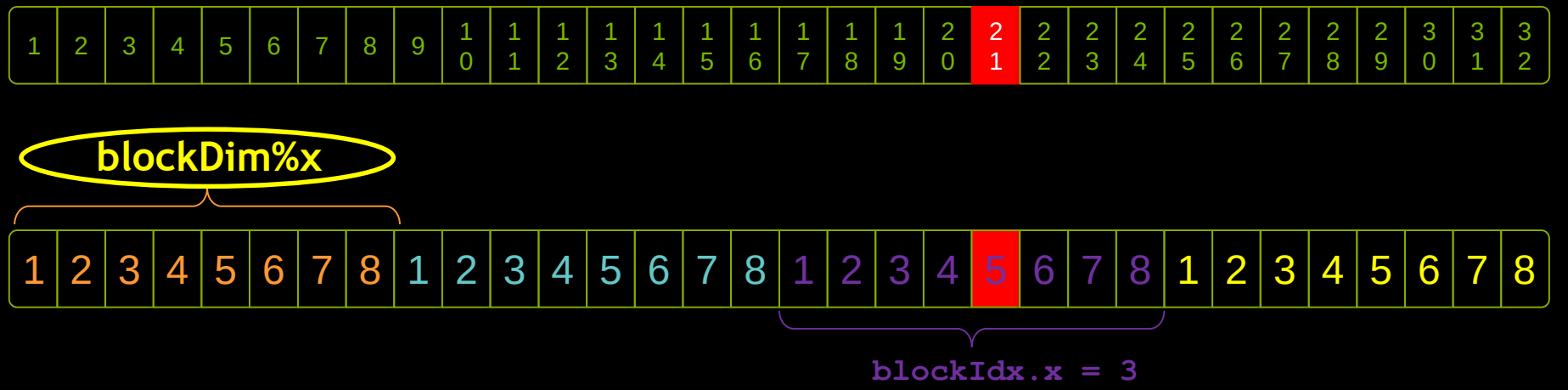
多个线程块、每个线程块有多个线程协同处理

- 考虑此时的索引关系
  - <<<1, n>>> : threadIdx
  - <<<n, 1>>> : blockIdx
  - <<<nG, nB>>> eg nG = 4, nB = 8

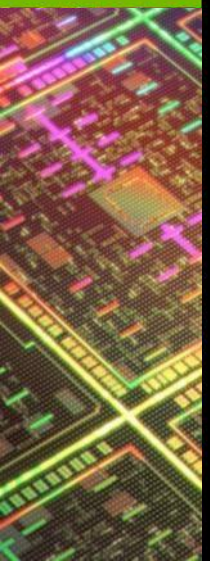


多个线程块、每个线程块有多个线程协同处理

- 考虑此时的索引关系




```
index = threadIdx%x + (blockIdx%x-1) * 8;
      =       5     +      2       * 8;
      = 21;
```



多个线程块、每个线程块有多个线程协同处理

- 考虑此时的索引关系
  - 除线程块ID blockIdx%x 和 线程 ID threadIdx%x 外，还要用到线程块的维度信息 blockDim%x 确定线程与待处理数据之间的对应关系

```
index = threadIdx%x + (blockIdx%x-1) * blockDim%x
```

  - 线程块的维度信息 `blockDim` **也是有三个成员的内建变量，即对应线程块每个维度的大小**

N 非常大，设备端代码

```
module simpleOps_m
contains
  attributes(global) subroutine increment(a, b)
    implicit none
    integer, intent(inout) :: a(:)
    integer, value :: b
    integer :: i, n

    i = blockDim%x*(blockIdx%x-1) + threadIdx%x
    n = size(a)
    if (i <= n) a(i) = a(i)+b

  end subroutine increment
end module simpleOps_m
```

Kernel Loop Directives

- 对主机端循环代码自动生成在 GPU 执行的 kernel
- 注意，用这种方式生成 kernel，主机端循环中的数组必须在设备端分配

```
program incTest
  use cudafor
  implicit none
  integer, parameter :: n = 256
  integer :: a(n), b
  integer, device :: a_d(n)

  a = 1; b = 3; a_d = a

  !$cuf kernel do <<<*,*>>>
  do i = 1, n
     a_d(i) = a_d(i)+b
  enddo

  a = a_d
  if (all(a == 4)) write(*,*) 'Test Passed'
end program incTest
```

Kernel Loop Directives (CUF Kernels)

- 对于多维数组

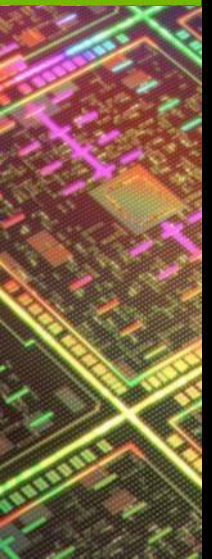
```
!$cuf kernel do(2) <<< *,* >>>
do j = 1, ny
  do i = 1, nx
    a_d(i,j) = b_d(i,j) + c_d(i,j)
  enddo
enddo
```

- 指定并行配置，即 block 和 Grid

```
!$cuf kernel do(2) <<<(*,*),(32,4)>>>
```

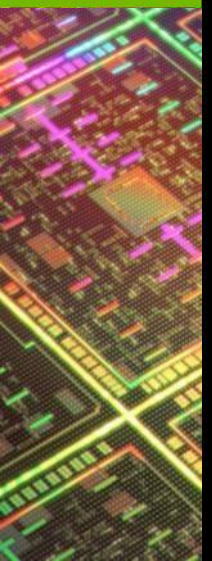
CUDA Fortran 编译选项

- PGI's Fortran 编译器
  - CUDA Fortran 默认文件后缀 .cuf 或 .CUF
  - `-Mcuda=cc35` 指定设备架构
  - `-Mcuda=cuda5.5` 指定 CUDA 版本号，即使用哪个版本的 CUDA
  - `-Mcuda=fastmath` 使用快速数学函数版本 (`__sinf()`)
  - `-Mcuda=maxregcount:<n>` 指定每个线程使用的最大线程数目
  - `-Mcuda=ptxinfo` 打印每个 kernel 的内存使用情况
- `pgf90 -Mcuda -help` 查看更多编译选项



GPU/CPU 同步

- `cudaDeviceSynchronize()`
  - 阻塞 CPU，直到 GPU 上的操作全部完成
- `cudaMemcpy/(a_d = a)`
  - 阻塞拷贝模式。阻塞 CPU，直到内存拷贝完成
- `cudaMemcpyAsync`
  - 非阻塞模式。拷贝任务启动后，控制返回给 CPU. 继续执行主机端代码
  - 为了保护数据的安全，在 CPU 端访问拷贝的数据前，需要做 CPU/GPU 之间同步



CUDA Fortran 总结(一)

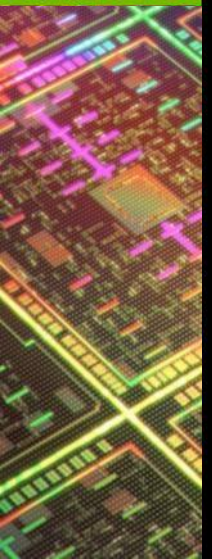
- 主机和设备
  - 主机：CPU 和它的内存；
  - 设备：GPU 和它的内存
  - 异构计算：CPU + GPU

- global, device, host
  - global 主机端调用，设备端执行；global 函数要在 module 中定义
  - device 修饰函数，或修饰变量
    - 修饰函数：设备端调用，设备端执行
    - 修饰变量：变量在设备上分配内存
  - host 主机端调用，主机端执行；函数默认行为

CUDA Fortran 总结(二)

- CUDA 编程的三个步骤
  - 设备端申请内存，并将数据从主机内存拷贝到设备内存
  - 调用 kernel，完成设备端的并发计算
  - 将计算结果从设备内存拷贝到主机内存，并释放内存

- 线程的两层组织结构
  - 线程块是线程的集合；Grid 是线程块的集合
  - 同一个线程块内，不同的线程间可以通过共享内存（shared）共享数据；通过 syncthreads() 实现线程同步
  - 不同的线程块之间，不能有效地实现数据共享；线程间不能同步



CUDA Fortran 总结(三)

- CUDA Fortran 内建变量
  - threadIdx : 线程 ID
  - blockIdx : 线程块 ID
  - blockDim : block的维度
  - gridDim : grid 的维度

- CUDA 执行模型
  - Kernel 都是异步执行
  - 内存拷贝默认为阻塞模式
  - 使用非阻塞模式拷贝内存时。主机端在读取数据时，需要做 CPU/GPU 同步