CUDA Fortran Dynamic Parallelism (1)

```
attributes(global) subroutine strassen( A, B, C, M, N, K )
real :: A(M,K), B(K,N), C(M,N)
integer, value :: N, M, K
. . .
if (ntimes == 0) then
  allocate(m1(1:m/2,1:k/2))
  allocate(m2(1:k/2,1:n/2))
  allocate(m3(1:m/2,1:k/2))
  allocate(m4(1:k/2,1:n/2))
  allocate(m5(1:m/2,1:k/2))
  allocate(m6(1:k/2,1:n/2))
  allocate(m7(1:m/2,1:k/2))
  flags =  cudaStreamNoBlocking
  do i = 1, 7
    istat = cudaStreamCreateWithFlags(istreams(i), flags)
  end do
end if
. . .
```

Support for Fortran allocate and deallocate in device code

CUDA Fortran Dynamic Parallelism (2)

```
. . .
! m1 = (A11 + A22) * (B11 + B22)
call dgemm16t1<<<devblocks,devthreads,0,istreams(1)>>>(a(1,1), &
                        a(1+m/2,1+k/2), m, &
                        b(1,1), b(1+k/2,1+n/2), k, &
                        m1(1,1), newn, newn, 1.0d0)
! m2 = (A21 + A22) * B11
call dgemm16t2<<<devblocks,devthreads,0,istreams(2)>>>(a(1+m/2,1), &
                        a(1+m/2,1+k/2), m, &
                        b(1,1), k, &
                        m2(1,1), newn, newn)

. . .

! m7 = (A12 - A22) * (B21 + B22)
call dgemm16t1<<<devblocks,devthreads,0,istreams(7)>>>(a(1,1+k/2), &
                        a(1+m/2,1+k/2), m, &
                        b(1+k/2,1), b(1+k/2,1+n/2), k, &
                        m7(1,1), newn, newn, -1.0d0)

istat = cudal

. . .
```

Support for kernel launch from device code

CUDA Fortran Dynamic Parallelism (3)

```
. . .
! C11 = m1 + m4 - m5 + m7
 call add16x4<<<1,devthreads,0,istreams(1)>>>(m1,m4,m5,m7,m/2,c(1,1),m,n/2)

! C12 = m3 + m5
call add16<<<1,devthreads,0,istreams(2)>>>(m3,m/2,m5,m/2,c(1,1+n/2),m,n/2)

! C21 = m2 + m4
call add16<<<1,devthreads,0,istreams(3)>>>(m2,m/2,m4,m/2,c(1+m/2,1),m,n/2)

! C22 = m1 + m3 - m2 + m6
call add16x4<<<1,devthreads,0,istreams(4)>>>(m1,m3,m2,m6,m/2,c(1+m/2,1+n/2),m,n/2)

. . .

end subroutine strassen
```

Compile using –Mcuda=rdc,cc35

CUDA Fortran Separate Compilation

Compile separate modules independently

```
% pgf90 -c -O2 -Mcuda=rdc ddfun90.cuf ddmod90.cuf
```

Object files can be put into a library

```
% ar rc ddfunc.a ddfun90.o ddmod90.o
```

Use the modules in device code in typical Fortran fashion

```
% cat main.cuf
        program main
        use ddmodule
        . . .
```

Link using pgf90 and the rdc option

```
% pgf90 -O2 -Mcuda=rdc main.cuf ddfunc.a
```

Performance Optimization

- Host-device data transfers
  - Page-locked transfers
  - Asynchronous transfers
- Device memory
  - Coalescing
  - Shared memory
  - Textures
- Execution Configuration
  - Thread-level parallelism
  - Instruction-level parallelism

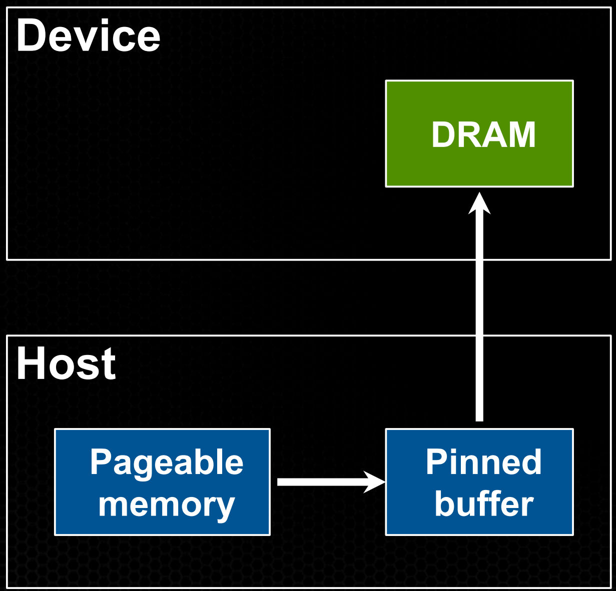
Host-Device Transfers

- Host-device bandwidth is much lower than bandwidth within device
  - 8 GB/s peak (PCIe x16 Gen 2) vs. 250 GB/s peak (Tesla K20X)

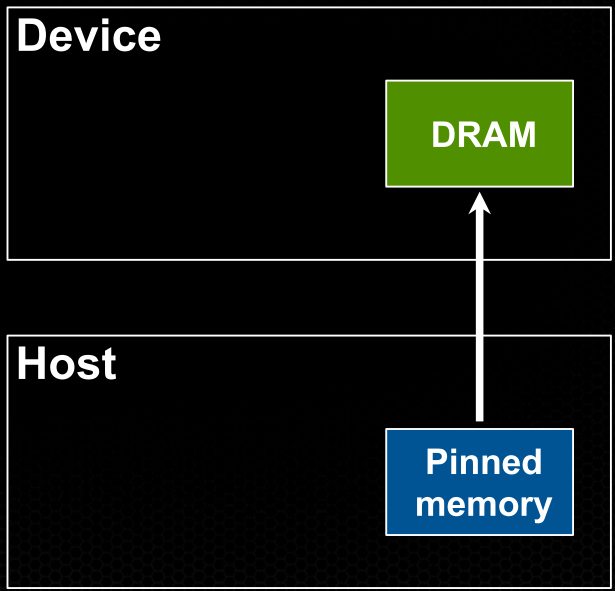
- Minimize number of transfers
  - Intermediate data can be allocated, used, and deallocated without copying to host memory
  - Sometimes better to do low parallelism operations on the GPU if it avoids transfers to and from host

Page-Locked Data Transfers

Pageable Data Transfer



Page-locked Data Transfer



Page-Locked Data Transfers

- Page-locked or pinned host memory by declaration
  - Designated by `pinned` variable attribute
  - Must be `allocatable`

```
real, device :: a_d(N)
real, pinned, allocatable :: a(:)
allocate(a(N), STAT=istat, PINNED=pinnedFlag)
...
a_d = a
```

  - Tesla K20/Sandy Bridge
    - Pageable: ~3.3 GB/s
    - Pinned: ~6 GB/s

Overlapping Transfers and Computation

- Kernel launches are asynchronous, normal memory copies are blocking

```
a_d = a   ! blocks on host until transfer completes
call inc<<<g,b>>>(a_d, b) ! Control returns immediately to CPU
a = a_d   ! starts only after kernel completes
```

- Asynchronous and Stream APIs allow overlap of transfers with computation
- A stream is a sequence of operations that execute in order on the GPU
  - Operations in different (*non-default*) streams can be interleaved
  - Stream ID used as arguments to async transfers and kernel launches

Asynchronous Data Transfers

- Asynchronous host-device transfers return control immediately to CPU
  - `cudaMemcpyAsync(dst, src, nElements, stream)`
  - Requires pinned host memory

- Overlapping data transfer with CPU computation
  - default stream = 0

```
istat = cudaMemcpyAsync(a_d, a_h, N, 0)
call kernel<<<grid, block>>>(a_d)
call cpuFunction(b)
```

overlapped

Overlapping Transfers and Kernels

- Requires:
  - Pinned host memory
  - Kernel and transfer to use different *non-zero* streams

```
integer (kind=cuda_stream_kind) :: stream1, stream2
...
istat = cudaStreamCreate(stream1)
istat = cudaStreamCreate(stream2)
istat = cudaMemcpyAsync(a_d, a_h, N, stream1)
call kernel<<<grid, block, 0, stream2>>>(b_d)
```

overlapped

GPU/CPU Synchronization

- `cudaDeviceSynchronize()`
  - Blocks until all previously issued operations on the GPU complete
- `cudaStreamSynchronize(stream)`
  - Blocks until all previously issued operations to `stream` complete
- `cudaStreamQuery(stream)`
  - Indicates whether `stream` is idle
  - Does not block CPU code

GPU/CPU Synchronization

- Stream-based using CUDA events
  - Events can be inserted into streams

```
type (cudaEvent) :: event
...
istat = cudaEventRecord(event, stream1)
```

  - Event is recorded when the GPU reaches it in the stream
    - Recorded = assigned a time stamp
    - Useful for timing code
  - `cudaEventSynchronize(event)`
    - Blocks CPU until event is recorded

Asynchronous Example

- async.cuf