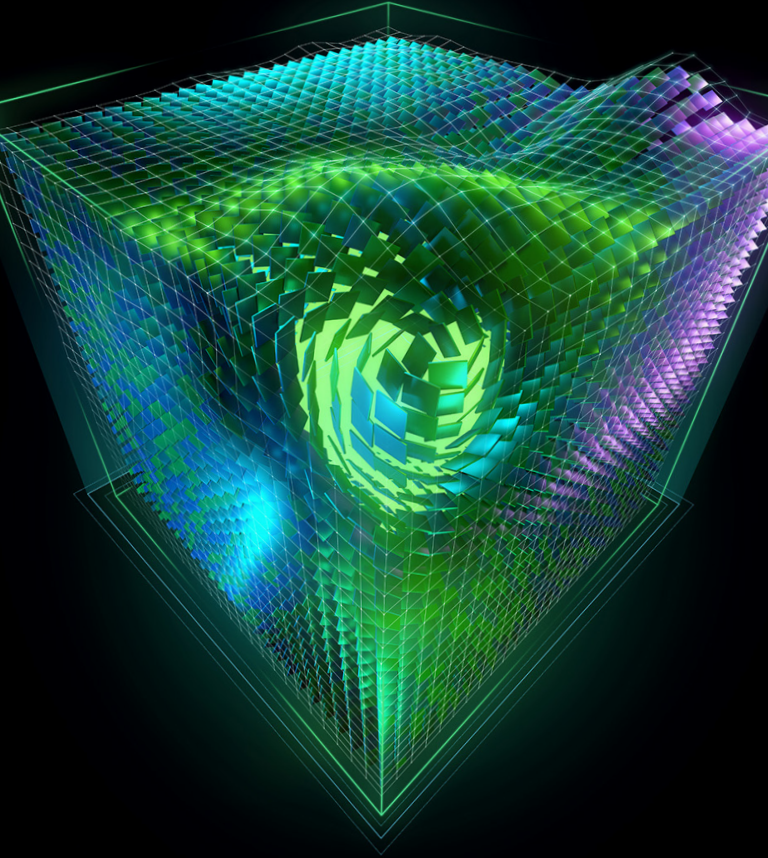


Quick Reference Guide

CUDA Fortran



PGI[®]

COMPILERS & TOOLS

CUDA Fortran Quick Reference Guide

CUDA Fortran is a Fortran analog to the NVIDIA CUDA C language for programming GPUs.

It includes language features, intrinsic functions and API routines for writing CUDA kernels and host control code in Fortran while remaining fully interoperable with CUDA C. Included in the language are subroutine attributes to define global (kernel) and device subroutines and functions, and variable attributes to declare and allocate device data and host pinned data.

Most Fortran numeric and mathematical intrinsic functions can be used in device code, as can many CUDA device built-in and libm functions. The CUDA Fortran language allows allocatable device arrays, and array assignments between host and device arrays using standard Fortran syntax to move data between host and device memory. A full set of CUDA runtime API routines is available for low-level control of device memory, streams, asynchronous operations, and events.

Subprogram Qualifiers

`attributes(global), attributes(grid_global)`

- declares a subroutine as being a kernel
- must be a **subroutine**
- may not be recursive
- may not contain variables with **save** attribute or data initialization
- may not have **optional** arguments
- **grid_global** allows grid-wise synchronization; introduced in CUDA 9.0; requires **cc60** or higher

`attributes(device)`

- declares a subprogram that is executed on and callable only from the device
- may not be recursive
- may not contain variables with the **save** attribute or data initialization
- may not have **optional** arguments

Variable Qualifiers

`device`

- allocated in device global memory

`constant`

- allocated in device constant memory space
- may be written by a host subprogram; is read-only in global and device subprograms

`shared`

- allocated in device shared memory
- may only appear in a **global** or **device** subprogram

`texture`

- accessible read-only in device subprograms
- explicit: declared with **texture, pointer** attributes at module scope; real or integers
- implicit: declare kernel arguments with **intent(in)**; all intrinsic types (**cc35** or higher)

`pinned`

- allocated in host page-locked memory
- must be an **allocatable** array; valid only in host subprograms

`managed`

- declared on the host
- can be accessed from either host or device

Predefined Types, Constants and Variables

(Parenthesized names used in Host Control API descriptions)

Parameters

cuda_count_kind (count)
cuda_stream_kind (stream)

Types

C_DEVPTR (devptr)	cudaArrayPtr (carray)
DIM3 (dim3)	cudaChannelFormatDesc (cdesc)
cudaDeviceProp (prop)	cudaFuncAttributes (attr)
cudaEvent (event)	cudaMemcpy3DParms (p)
cudaExtent (cext)	cudaMemcpy3DPeerParms (p)
cudaPos	cudaPitchedPtr (pitchptr)
cudaSymbol (symbol)	cudaPointerAttributes (ptr)

Variables

type(dim3) :: threadIdx, blockDim, blockIdx, griddim
integer(kind=4) :: warpsize

Device Code

Data Types

character(len= 1)	integer(kind= 1 2 4 8)
complex(kind= 4 8)	logical(kind= 1 2 4 8)
double precision	real(kind= 4 8)

Numeric and Logical Intrinsic Functions

abs(integer | real | complex)
int(integer | real | complex)

aimag(complex)	logical(logical)
aint(real)	max(integer real)
aint(real)	min(integer real)
ceiling(real)	mod(integer real)
cmplx(real)	modulo(integer real)
cmplx(real,real)	nint(real)
conjg(complex)	real(integer real complex)
dim(integer real)	sign(integer real)
floor(real)	

Mathematical Intrinsic Functions

acos(real)	log(real complex)
asin(real)	log10(real)
atan(real)	sin(real complex)
atan2(real,real)	sinh(real) cos(real complex)
sqrt(real complex)	cosh(real)
tan(real)	tanh(real)
exp(real complex)	

Numeric Inquiry Intrinsic Functions

bit_size(integer)	precision(real complex)
digits(integer real)	radix(integer real)
epsilon(real)	range(integer real complex)
huge(integer real)	selected_int_kind(integer)
maxexponent(real)	minexponent(real)
tiny(real)	
selected_real_kind(integer,integer)	

Bit Manipulation Intrinsic Functions

btest(integer)	ishft(integer)
iand(integer)	ishftc(integer)
ibclr(integer)	leadz(integer)
ibits(integer)	mvbits(integer)
ibset(integer)	not(integer)
ieor(integer)	popcnt(integer)
ior(integer)	poppar(integer)
transfer(integer real complex)	

CUDA Synchronization and Fence Functions

syncthreads()
syncthreads_count(integer | logical)
syncthreads_and(integer | logical)
syncthreads_or(integer | logical)
threadfence()
threadfence_block()
threadfence_system()

Reduction Intrinsic Functions

all(logical)	minloc(integer real)
any(logical)	minval(integer real)
count(logical)	maxloc(integer real)
maxval(integer real)	sum(integer real complex)
product(integer real complex)	

CUDA Warp Vote Functions

allthreads(logical)
anythread(logical)
ballot(integer | logical)

CUDA Warp Shuffle Functions

__shfl(integer | real, [size])
__shfl_up(integer | real, [size])
__shfl_down(integer | real, [size])
__shfl_xor(integer | real, [size])

CUDA Atomic Functions

atomicadd(integer | real) atomicmax(integer | real)
atomicand(integer) atomicmin(integer | real)
atomiccas(integer | real) atomicor(integer)
atomicdec(integer) atomicsub(integer | real)
atomicexch(integer | real) atomicxor(integer)
atomicinc(integer)

CUDA Device libm Routines

use libm
acosh[f](x) llrint[f](x)
asinh[f](x) lrint[f](x)
asinf(x) llround[f](x)
acosf(x) lround[f](x)
atan2f(x,y) logb[f](x)
atanh[f](x) log10f(x)

CUDA Device libm Routines

use libm
acosh[f](x) llrint[f](x)
asinh[f](x) lrint[f](x)
asinf(x) llround[f](x)
acosf(x) lround[f](x)
atan2f(x,y) logb[f](x)
atanh[f](x) log10f(x)
cbrt[f](x) logf(x)
ceil[f](x) log1p[f](x)
copysign[f](x,y) log2[f](x)
cosf(x) modf[f](x,y)
coshf(x) nearbyint[f](x)
erf[f](x) nextafter[f](x,y)
erfc[f](x) pow[f](x,y)
expm1[f](x) remainder[f](x,y)
expf(x) remquo[f](x,y,i)
exp10[f](x) rint[f](x)
exp2[f](x) scalbn[f](x,n)
fabs[f](x) scalbln[f](x,n)
floor[f](x) sinf(x)
fma[f](x,y,z) sinhf(x)
fmax[f](x,y) sqrtf(x)
fmin[f](x,y) tanf(x)
frexp[f](x,y) tanhf(x)
ilogb[f](x) tgamma[f](x)
ldexp[f](x,y) trunc[f](x)
lgamma[f](x)

CUDA Device Built-in Routines

```
use cudadevice
__[u]mulhi(i, j)          __double2[u]int_r[n|z|u|d](r)
__[u]mul64hi(i, j)      __[u]int2double_r[n|z|u|d](i)
__int_as_float(i)       __double2[u]ll_r[n|z|u|d](r)
__float_as_int(i)       __[u]ll2double_r[n|z|u|d](i)
__saturatef(r)          __fadd_r[n|z|u|d](a, b)
__[u]sad(i, j, k)       __fmul_r[n|z|u|d](a, b)
__fdividef(r, d)        __fmaf_r[n|z|u|d](a, b, c)
fdividef[r, d]          __frcp_r[n|z|u|d](a)
__sinf(r)               __fsqrt_r[n|z|u|d](a)
__cosf(r)               __fdiv_r[n|z|u|d](a,b)
__tanf(r)               __dadd_r[n|z|u|d](x, y)
__expf(r)               __dmul_r[n|z|u|d](x, y)
__exp10f(r)             __dfma_r[n|z|u|d](x, y, z)
__log2f(r)              __drcp_r[n|z|u|d](x)
__log10f(r)             __dsqrt_r[n|z|u|d](x)
__logf(r)               __ddiv_r[n|z|u|d](x, y)
__powf(r,d)             __float2[u]int_r[n|z|u|d](r)
__clz[ll](i)            __[u]int2float_r[n|z|u|d](i)
__ffs[ll](i)            __float2[u]ll_r[n|z|u|d](r)
__popc[ll](i)           __[u]ll2float_r[n|z|u|d](i)
__brev[ll](i)           __float2half_r[n|z|u|d](r)
__half2float(i)
```

Host Control Code

Kernel Launch

```
call kernel<<<grid,block[, [nbytes]*]
  [,streamid]>>> ( arguments )
```

- **grid** and **block** are integer expressions, or **type(dim3)** variables

For **grid_global** launches the wildcard "*" can be used for the **grid** parameter, where the compiler will determine the number of blocks to use.

- **nbytes** specifies how many bytes of memory to allocate for dynamic shared memory
- **streamid** is a stream identifier

CUF Kernel

```
!$cuf kernel do(n) [ <<< grid, block >>> ]
  do i_n ...
  ...
  do i_1 ...
```

- Create a kernel from the following **n** nested loops
- **grid** and **block** are lists of **n** integer expressions, corresponding to the **n** loops, starting with the innermost
- If any expression has the value 1, that loop will not correspond to a block or thread index
- If any expression is *, the compiler will choose a size to use for that dimension

API Routines for Device Management

```

cudaChooseDevice( devnum, prop )
*cudaDeviceGetCacheConfig( cacheconfig )
*cudaDeviceGetLimit( val, limit )
cudaDeviceGetSharedMemConfig( config )
cudaDeviceGetStreamPriorityRange( least, greatest )
cudaDeviceReset( )
cudaDeviceSetCacheConfig( cacheconfig )
cudaDeviceSetLimit( limit, val )
cudaDeviceSetSharedMemConfig( config )
*cudaDeviceSynchronize( )
*cudaGetDevice( devnum )
*cudaGetDeviceCount( numdev )
*cudaGetDeviceProperties( prop, devnum )
cudaSetDevice( devnum )
cudaSetDeviceFlags( flags )
cudaSetValidDevices( devices, numdev )

```

API Routines for Memory Management and Data Transfer

```

*cudaFree( devptr )
cudaFreeArray( arrayptr )
cudaFreeHost( hostptr )
cudaHostAlloc( hostptr, size, flags )
cudaHostGetDevicePointer( devptr, hostptr, flags )
cudaHostGetFlags( flags, hostptr )
cudaHostRegister( hostptr, count, flags )
cudaHostUnregister( hostptr )
*cudaMalloc( devptr, count )
cudaMallocManaged( devptr, count, flags )
cudaMemcpy( dst, src, count )
cudaMemcpyAsync( dst, src, count, stream )
cudaMemcpyPeer( dst, dstdev, src, srcdev, count )
cudaMemcpyPeerAsync( dst, dstdev, src, srcdev,
count, stream )
cudaMemcpy2D( dst, dpitch, src, spitch, width, height,
kdir )
*cudaMemcpy2DAsync( dst, dpitch, src, spitch, width,
height, kdir, stream )
cudaMemcpy3D( p )
*cudaMemcpy3DAsync( p, stream )
cudaMemcpy3DPeer( p )
cudaMemcpy3DPeerAsync( p, stream )
cudaMemGetInfo( free, total )

```


cudaMemset(devptr, value, count)
cudaMemset2D(devptr, pitch, value, width, height)
cudaMemset3D(pitchptr, value, cext)
cudaMemsetAsync(devptr, value, count, stream)
cudaMemset2DAsync(devptr, pitch, value, width,
height, stream)
cudaMemset3DAsync(pitchptr, value, cext, stream)
cudaPointerGetAttributes(attr, ptr)

API Routines for Execution Control

*cudaFuncGetAttributes(attr, func)
cudaFuncSetAttributes(func, attr, value)
cudaFuncSetCacheConfig(func, cacheconfig)
cudaFuncSetSharedMemConfig(func, config)
cudaSetDoubleForDevice(d)
cudaSetDoubleForHost(d)

API Routines for Stream Management

cudaforSetDefaultStream(stream)
cudaforGetDefaultStream(stream)
cudaStreamAttachMemAsync(stream, devptr, length,
flags)
cudaStreamCreate(stream)
*cudaStreamCreateWithFlags(stream, flags)
cudaStreamCreateWithPriority(stream, flags,
priority)
*cudaStreamDestroy(stream)
cudaStreamGetPriority(stream, priority)
cudaStreamQuery(stream)
cudaStreamSynchronize(stream)
*cudaStreamWaitEvent(stream, events, flags)

API Routines for Event Management

cudaEventCreate(event)
*cudaEventCreateWithFlags(event, flags)
*cudaEventDestroy(event)
cudaEventElapsedTime(time, start, end)
cudaEventQuery(event)
*cudaEventRecord(event, stream)
cudaEventSynchronize(event)

API Routines for Unified Addressing and Peer Device Memory Access

cudaDeviceCanAccessPeer(canAccessPeer, device,
peerDevice)
cudaDeviceDisablePeerAccess(peerDevice)
cudaDeviceEnablePeerAccess(peerDevice, flags)
cudaMemAdvise(devptr, count, advice, device)
cudaMemPrefetchAsync(devptr, count, device,
stream)

API Routines for Occupancy

`cudaOccupancyMaxActiveBlocksPerMultiprocessor(numBlocks, func, blockSize, dynamicSMemSize)`

`cudaOccupancyMaxActiveBlocksPerMultiprocessor- WithFlags(numBlocks, func, blockSize, dynamicSMemSize, flag)`

API Routines for Unified Addressing and Peer Device Memory Access

`cudaDeviceCanAccessPeer(canAccessPeer, device, peerDevice)`

`cudaDeviceDisablePeerAccess(peerDevice)`

`cudaDeviceEnablePeerAccess(peerDevice, flags)`

`cudaMemAdvise(devptr, count, advice, device)`

`cudaMemPrefetchAsync(devptr, count, device, stream)`

`cudaPointerGetAttributes(attr, ptr)`

API Routines for Error Handling

`*cudaGetErrorString(errorcode)`

`*cudaGetLastError()`

`*cudaPeekAtLastError()`

API Routines for Version Management

`cudaDriverGetVersion(version)`

`*cudaRuntimeGetVersion(version)`

* These API routines are supported in device code as well as host code.

Compiler Options

`pgfortran a.cuf -Mcuda[=options]`

Where **options** are one or more of the following separated by commas:

- `cc30 | cc35 | cc3x | cc50 | cc60 | cc70`
- `cuda9.0 | cuda9.1 | cuda9.2 | cuda10.0`
- `fastmath`
- `maxregcount:n`
- `nofma`
- `ptxinfo`

Enter `pgfortran -Mcuda -help` for a complete list of compiler options.

Pre-compiled Host Modules

- **cudafor** contains Fortran interfaces for the CUDA API routines listed in this guide, the predefined types, and Fortran interfaces for device versions of the reduction intrinsics `sum`, `maxval`, and `minval`.
- **cublas** contains Fortran interfaces for the CUBLAS library. The standard Fortran BLAS interfaces are also overloaded to accept device data. Link with **-McuDaliB=cublas**.
See <http://docs.nvidia.com/cuda/cublas/>
- **cufft** contains Fortran interfaces for the CUFFT library. Link with **-McuDaliB=cufft**.
See <http://docs.nvidia.com/cuda/cufft/>
- **curand** contains Fortran interfaces for the CURAND library. Link with **-McuDaliB=curand**.
See <http://docs.nvidia.com/cuda/curand/>
- **cusolver** contains Fortran interfaces for LAPACK-like routines Link with **-McuDaliB=cusolver**.
See <http://docs.nvidia.com/cuda/cusolver/>
- **cusparse** contains Fortran interfaces for the CUSPARSE library. Link with **-McuDaliB=cusparse**.
See <http://docs.nvidia.com/cuda/cusparse/>

Pre-compiled Device Modules

- **cudaDevice** allows access and interfaces to many of the CUDA device built-in routines. Used by default.
- **libm** provides interfaces to standard libm functions which are not in the Fortran intrinsic library.
- **cublas_device** provides interfaces to the device CUBLAS library. Requires **cc35** or newer.
- **cooperative_groups** provides interfaces to cooperative group functionality. Requires CUDA 9.0 or newer.

Types: **grid_group**, **coalesced_group**, **thread_group** (**grid_group** requires **cc60** or higher)

Functions: **this_warp()**, **this_grid()**, **this_thread_block()** (**this_grid()** requires **cc60** or higher)

syncthreads() is overloaded to synchronize across specified groups:

call **syncthreads(gg | cg | tg)**

shfl_sync() shfl functions (requires CUDA 9.0 or newer)

CUDA Fortran

See also the PGI Fortran CUDA Library Interfaces document:

<http://www.pgicompilers.com/docs/pgicudaint>

PGI[®]
COMPILERS & TOOLS

www.pgicompilers.com/cudafortran

© 2018 NVIDIA Corporation. All rights reserved.