CUDA complete | Complete reference on CUDA

By **Nitin Gupta**, cuda-programming.blogspot.com (http://getpocket.com/redirect?url=http%3A%2F%2Fcuda-programming.blogspot.com%2F2013%2F03%2Fcuda-complete-complete-reference-on-cuda.html)
View Original (http://getpocket.com/redirect?url=http%3A%2F%2Fcuda-programming.blogspot.com%2F2013%2F03%2Fcuda-complete-complete-reference-on-cuda.html)

CUDA syntax

Source code is in .cu files, which contain mixture of host (CPU) and device (GPU) code.

Declaring functions

Declaring variables

__device__	declares device variable in global memory, accessible from all threads, with lifetime of application
__constant__	declares device variable in constant memory, accessible from all threads, with lifetime of application
__shared__	declares device varibale in block's shared memory, accessible from all threads within a block, with lifetime of block
__restrict__	standard C definition that pointers are not aliased

Types

Most routines return an error code of type **`cudaError_t`**.

Vector types

```
char1, uchar1, short1, ushort1, int1, uint1,
long1, ulong1, float1
char2, uchar2, short2, ushort2, int2, uint2,
long2, ulong2, float2
char3, uchar3, short3, ushort3, int3, uint3,
long3, ulong3, float3
char4, uchar4, short4, ushort4, int4, uint4,
long4, ulong4, float4

longlong1, ulonglong1, double1
longlong2, ulonglong2, double2

dim3
```

Components are accessible as **`variable.x`**, **`variable.y`**, **`variable.z`**, **`variable.w`**.

Constructor is **`make_<type>( x, ... )`**, for example:

```
float2 xx = make_float2( 1., 2. );
```

dim3 can take 1, 2, or 3 argumetns:

```
dim3 blocks1D( 5       );
dim3 blocks2D( 5, 5    );
dim3 blocks3D( 5, 5, 5 );
```

Pre-defined variables

Kernel invocation

```
__global__ void kernel( ... ) { ... }

dim3 blocks( nx, ny, nz );            // cuda
1.x has 1D and 2D grids, cuda 2.x adds 3D grids
dim3 threadsPerBlock( mx, my, mz );  // cuda
1.x has 1D, 2D, and 3D blocks

kernel<<< blocks, threadsPerBlock >>>( ... );
```

Thread management

Memory management

```
__device__ float* pointer;
cudaMalloc( &pointer, size );
cudaFree( pointer );

// direction is one of cudaMemcpyHostToDevice
or cudaMemcpyDeviceToHost
cudaMemcpy( dst_pointer, src_pointer, size,
direction );

__constant__ float dev_data[n];
float host_data[n];
cudaMemcpyToSymbol  ( dev_data,  host_data,
sizeof(host_data) );  // dev_data  = host_data
cudaMemcpyFromSymbol( host_data, dev_data,
sizeof(host_data) );  // host_data = dev_data
```

Also, **`malloc`** and **`free`** work inside a kernel (2.x), but memory allocated in a kernel must be deallocated in a kernel (not the host). It can be freed in a different kernel, though.

Atomic functions

```
old = atomicAdd ( &addr, value );  // old =
*addr;  *addr += value
old = atomicSub ( &addr, value );  // old =
*addr;  *addr -= value
old = atomicExch( &addr, value );  // old =
*addr;  *addr  = value

old = atomicMin ( &addr, value );  // old =
*addr;  *addr = min( old, value )
old = atomicMax ( &addr, value );  // old =
*addr;  *addr = max( old, value )

// increment up to value, then reset to 0
// decrement down to 0, then reset to value
old = atomicInc ( &addr, value );  // old =
*addr;  *addr = ((old >= value) ? 0 : old+1 )
old = atomicDec ( &addr, value );  // old =
*addr;  *addr = ((old == 0) or (old > val) ?
val : old-1 )

old = atomicAnd ( &addr, value );  // old =
*addr;  *addr &= value
old = atomicOr  ( &addr, value );  // old =
*addr;  *addr |= value
old = atomicXor ( &addr, value );  // old =
*addr;  *addr ^= value

// compare-and-store
old = atomicCAS ( &addr, compare, value );  //
old = *addr;  *addr = ((old == compare) ? value
: old)
```

Warp vote

```
int __all   ( predicate );
int __any   ( predicate );
int __ballot( predicate );  // nth thread sets
nth bit to predicate
```

Timer

wall clock cycle counter

```
clock_t clock();
```

Texture

can also return float2 or float4, depending on texRef.

```
// integer index
float tex1Dfetch( texRef, ix );

// float index
float tex1D( texRef, x       );
float tex2D( texRef, x, y    );
float tex3D( texRef, x, y, z );

float tex1DLayered( texRef, x    );
float tex2DLayered( texRef, x, y );
```

Low-level Driver API

```
#include <cuda.h>

CUdevice dev;
CUdevprop properties;
char name[n];
int major, minor;
size_t bytes;

cuInit( 0 );  // takes flags for future use
cuDeviceGetCount          ( &cnt );
cuDeviceGet               ( &dev, index );
cuDeviceGetName           ( name, sizeof(name),
dev );
cuDeviceComputeCapability( &major, &minor,
dev );
cuDeviceTotalMem          ( &bytes,
dev );
cuDeviceGetProperties     ( &properties,
dev );  // max threads, etc.
```

cuBLAS

Matrices are column-major. Indices are 1-based; this affects result of i<t>amax and i<t>amin.

```
#include <cublas_v2.h>

cublasHandle_t handle;
cudaStream_t   stream;

cublasCreate( &handle );
cublasDestroy( handle );
cublasGetVersion( handle, &version );
cublasSetStream( handle,  stream );
cublasGetStream( handle, &stream );
cublasSetPointerMode( handle,  mode );
cublasGetPointerMode( handle, &mode );

// copy x => y
cublasSetVector     ( n, elemSize, x_src_host,
incx, y_dst_dev,  incy );
cublasGetVector     ( n, elemSize, x_src_dev,
incx, y_dst_host, incy );
cublasSetVectorAsync( n, elemSize, x_src_host,
incx, y_dst_dev,  incy, stream );
cublasGetVectorAsync( n, elemSize, x_src_dev,
incx, y_dst_host, incy, stream );

// copy A => B
cublasSetMatrix     ( rows, cols, elemSize,
A_src_host, lda, B_dst_dev,  ldb );
cublasGetMatrix     ( rows, cols, elemSize,
A_src_dev,  lda, B_dst_host, ldb );
cublasSetMatrixAsync( rows, cols, elemSize,
A_src_host, lda, B_dst_dev,  ldb, stream );
cublasGetMatrixAsync( rows, cols, elemSize,
A_src_dev,  lda, B_dst_host, ldb, stream );
```

Constants

BLAS functions have **`cublas`** prefix and first letter of usual BLAS function name is capitalized. Arguments are the same as standard BLAS, with these exceptions:

- All functions add handle as first argument.
- All functions return cublasStatus_t error code.
- Constants alpha and beta are passed by pointer. All other scalars (n, incx, etc.) are bassed by value.
- Functions that return a value, such as ddot, add result as last argument, and save value to result.
- Constants are given in table above, instead of using characters.

Examples:

```
cublasDdot ( handle, n, x, incx, y, incy,
&result );  // result = ddot( n, x, incx, y,
incy );
cublasDaxpy( handle, n, &alpha, x, incx, y,
incy );   // daxpy( n, alpha, x, incx, y, incy
);
```

Compiler

`nvcc`, often found in **`/usr/local/cuda/bin`**
Defines **`__CUDACC__`**

Flags common with cc

Flags specific to nvcc

`-v`	list compilation commands as they are executed
`-dryrun`	list compilation commands, without executing
`-keep`	saves intermediate files (e.g., pre-processed) for debugging
`-clean`	removes output files (with same exact compiler options)
`-arch=<compute_xy>`	generate PTX for capability x.y
`-code=<sm_xy>`	generate binary for capability x.y, by default same as `-arch`
`-gencode arch=...,code=...`	same as `-arch` and `-code`, but may be repeated

Argumenents for `-arch` and `-code`

It makes most sense (to me) to give `-arch` a virtual architecture and `-code` a real architecture, though both flags accept both virtual and real architectures (at times).