Parallel Programming

CUDA C Extensions and Basic APIs

Overview

- CUDA C Extensions
  - Function type qualifiers
  - Variable qualifiers
  - Built-in types
  - Built-in variables
- CUDA Basic APIs
  - Memory management
  - Execution configuration & thread synchronization
  - Event management & error handling

Function Type Qualifiers

- Specify (1) whether a function executes on the host or on the device and (2) whether it is callable from the host or from the device.
- *__global__*
- *__device__*
- *__host__*

The __*global*__ qualifier

- Declares a function as being a kernel:
  - Executed on the device
  - Callable from the host
  - Callable from the device for devices of compute capability 3.x and up
- __*global*__ functions must have *void* return type.
- Any call to a __global__ function must specify its execution configuration <<<….>>>.
- A call to a __global__ function is asynchronous:
  - Returns before the device has completed its execution

The __*device*__ qualifier

- Executed on the device
- Callable from the device only
- No execution configuration
- No restriction on function types
- Can call another __*device*__ function
- Synchronous

The __*host*__ qualifier

- Executed on the host
- Callable from the host only
- Is optional:
  - Equivalent to without any of the three qualifiers
- Can be used together with __*device*__
  - compiled for both the host and the device
  - Inside the function the __*CUDA_ARCH*__ value tells whether it is for the host or the device

Example of __host__ __device__

```
__host__ __device__ func()
{
#if __CUDA_ARCH__ >= 300
   // Device code path for compute capability 3.x
#elif __CUDA_ARCH__ >= 200
   // Device code path for compute capability 2.x
#elif __CUDA_ARCH__ >= 100
  // Device code path for compute capability 1.x
#elif !defined(__CUDA_ARCH__)
  // Host code path
#endif
}
```

Variable Type Qualifiers

- A variable type qualifier specifies the memory location of the variable **on the device**.
- Three type qualifiers for variables on the device:
  - *__device__*
  - *__constant__*
  - *__shared__*
- A variable declared in **device code** without a type qualifier typically resides in the register.

The __*device*__ qualifier

- Declares a variable that resides on the device
  - Resides in global memory space
  - Has the lifetime of an application
  - Is accessible from all the threads within the grid
  - Is accessible from the host through the runtime library
    - *cudaGetSymbolAddress(), cudaGetSymbolSize(),*
    - *cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()*

Example of __device__ variable

```
__device__ int d_value;
__global__ void test_Kernel()
{
	int threadID = threadIdx.x;
	d_value = 1;
	printf("threadID %-3d d_value%3d\n",threadID,d_value);
}
int main()
{
	int h_value = 0;
	test_Kernel<<<1,2>>>();
	cudaMemcpyFromSymbol(&h_value,d_value,
		sizeof(int),0,cudaMemcpyDeviceToHost);

	printf("Output from host: %d\n",h_value);
	return 0;
}
```

The __*constant*__ qualifier

- Declares a variable that
  - Resides in constant memory space (read-only)
  - Has the lifetime of an application
  - Is accessible from all the threads within the grid and from the host through the runtime library
  - Optionally used together with __device__

The __*shared*__ qualifier

- Declares a variable that
  - Resides in the shared memory space of a thread block
  - Has the lifetime of the block
  - Is only accessible from all the threads within the block
  - Optionally used together with __device__

Built-In Vector Types

- Structures of *<basic_type><i> (i=1,2,3,4)*
  - *char, uchar, short, ushort, int, uint*
  - *long, ulong, longlong, ulonglong*
  - *float, double* (*i=1,2* for *double* vectors)
- 1st, 2nd, 3rd, and 4th components (if any) are accessible through the fields *x*, *y*, *z*, and *w*, respectively
- Constructor in the form: *make_<type name>*
  - E.g., *int2 make_int2(int x, int y);*

Built-In Variables

- *gridDim*, *blockDim*
  - Both are of type *dim3* (based on *uint3*)
- *blockIdx*, *threadIdx*
  - Both are of type *uint3*
- *warpSize*
  - Type *int*; size of warp in number of threads

Frequently Used CUDA Types

- CUDA stream type
  - *typedef CUstream_st * cudaStream_t*
- CUDA event type
  - *typedef CUevent_st * cudaEvent_t*
- CUDA Error type
  - *typedef enumcudaError cudaError_t*

Memory Allocation and Deallocation

*cudaError_t cudaMalloc (void** devPtr,*

size_t size)

Allocate memory on the device.

cudaError_t cudaFree (void devPtr)*

Free memory on the device.

Get Memory Address and Size

cudaError_t cudaGetSymbolAddress

*(void** devPtr, const void* symbol)*

Find the address associated with a CUDA symbol.

cudaError_t cudaGetSymbolSize

(size_t size, const void* symbol)*

Find the size of the object associated with a CUDA symbol.

Memory Copy between Host Variables

cudaError_t cudaMemcpy
(void dst, const void* src, size_t count,*
cudaMemcpyKind kind)
Copy data between host and device.

enum cudaMemcpyKind
cudaMemcpyHostToHost (= 0: Host -> Host)
cudaMemcpyHostToDevice (= 1: Host -> Device)
cudaMemcpyDeviceToHost (= 2: Device -> Host
cudaMemcpyDeviceToDevice (= 3: Device -> Device)
cudaMemcpyDefault (= 4: Default unified virtual address space)

Memory Copy for Device Variable

cudaError_t cudaMemcpyToSymbol
(const void symbol, const void* src,*
size_t count, size_t offset = 0,
cudaMemcpyKind kind = cudaMemcpyHostToDevice)

Copy data to the given symbol on the device.

cudaError_t cudaMemcpyFromSymbol (void dst,*
const void symbol, size_t count, size_t offset = 0,*
cudaMemcpyKind kind = cudaMemcpyDeviceToHost)

Copy data from the given symbol on the device.

Execution Configuration

<<< Dg, Db, Ns, S >>>

- Dg is of type dim3 and specifies the dimension and size of the grid, such that Dg.x * Dg.y * Dg.z equals the number of blocks being launched;
- Db is of type dim3 and specifies the dimension and size of each block, such that Db.x * Db.y * Db.z equals the number of threads per block;
- Ns is of type size_t and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory. Ns is an optional argument which defaults to 0;
- S is of type cudaStream_t and specifies the associated stream; S is an optional argument which defaults to 0.

Thread Synchronization

[DEPRECATED]:

cudaError_t cudaThreadSynchronize (void)

Wait for compute device to finish.

Should use:

cudaError_t cudaDeviceSynchronize (void)

Wait for compute device to finish.

Within a block of threads:

void __syncthreads();

waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to *__syncthreads()* are visible to all threads in the block.

Event Management

cudaError_t cudaEventCreate (cudaEvent_t* event)
   Creates an event object.
cudaError_t cudaEventCreateWithFlags (cudaEvent_t* event, unsigned int flags)
   Creates an event object with the specified flags.
cudaError_t cudaEventDestroy (cudaEvent_t event)
   Destroys an event object.
cudaError_t cudaEventElapsedTime (float* ms, cudaEvent_t start, cudaEvent_t end)
   Computes the elapsed time between events.
cudaError_t cudaEventQuery (cudaEvent_t event)
   Queries an event's status.
cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream = 0)
   Records an event.
cudaError_t cudaEventSynchronize (cudaEvent_t event)
   Waits for an event to complete.

Error Handling

const __cudart_builtin__ char cudaGetErrorName (cudaError_t error)*

Returns the string representation of an error code.

const __cudart_builtin__ char cudaGetErrorString (cudaError_t error)*

Returns the description string for an error code.

cudaError_t cudaGetLastError (void)

Returns the last error from a runtime call and resets it to *cudaSuccess*.

cudaError_t cudaPeekAtLastError (void)

Returns the last error from a runtime call.

Summary

- CUDA function type qualifiers specify where a function to be executed and to be called.
- CUDA variable type qualifiers specify where a device variable resides.
- CUDA has its own data types extended from C.
- CUDA has common memory management, event management, thread synchronization, and error handling functions.