Appendix B

CUDA: *C Language Extensions*

B. Introduction

The most popular way to write code in CUDA is using the CUDA C/C++ language. It is minimal extension to the GNU C/C++ so that CUDA features and APIs are supported. It is designed in such a way that a programmed, who is experienced with C/C++, will not get any trouble to learn these new features. Some of these features are discussed here.

B.1. Function Type Qualifiers

Function type qualifiers specify whether a function executes on the host or on the device and whether it is callable from the host or from the device.

`__device__`

The `__device__` qualifier declares a function that is:

- Executed on the device,
- Callable from the device only.

`__global__`

The `__global__` qualifier declares a function as being a kernel. Such a function is:

- Executed on the device,
- Callable from the host,
- Callable from the device for devices of compute capability 3.x

`__global__` functions must have void return type. Any call to a `__global__` function must specify its execution.

A call to a `__global__` function is asynchronous, meaning it returns before the device has completed its execution.

`__host__`

The `__host__` qualifier declares a function that is:

- Executed on the host,
- Callable from the host only.

It is equivalent to declare a function with only the `__host__` qualifier or to declare it without any of the `__host__`, `__device__`, or `__global__` qualifier; in either case, the function is compiled for the host only.

The `__global__` and `__host__` qualifiers cannot be used together.

The __device__ and __host__ qualifiers can be used together however, in which case the function is compiled for both the host and the device. The `__CUDA_ARCH__` macro can be used to differentiate code paths between host and device.

`__noinline__` and `__forceinline__`

The compiler inlines any `__device__` function when deemed appropriate.

The `__noinline__` function qualifier can be used as a hint for the compiler not to inline the function if possible. The function body must still be in the same file where it is called.

The `__forceinline__` function qualifier can be used to force the compiler to inline the function.

B.2. Variable Type Qualifiers

Variable type qualifiers specify the memory location on the device of a variable.

An automatic variable declared in device code without any of the __device__, __shared__ and __constant__ qualifiers described in this section generally resides in a register. However in some cases the compiler might choose to place it in local memory, which can have adverse performance consequences as detailed in Device Memory Accesses.

`__device__`

The `__device__` qualifier declares a variable that resides on the device.

At most one of the other type qualifiers defined in the next two sections may be used together with `__device__` to further specify which memory space the variable belongs to. If none of them is present, the variable:

- Resides in global memory space.
- Has the lifetime of an application.
- Is accessible from all the threads within the grid and from the host through the runtime library (i.e., `cudaGetSymbolAddress()` / `cudaGetSymbolSize()` / `cudaMemcpyToSymbol()` / `cudaMemcpyFromSymbol()`).
- May be additionally qualified with the `__managed__` qualifier. Such a variable can be directly referenced from host code, e.g., its address can be taken or it can read or written directly from a host function. As a convenience, `__managed__` implies `__managed__ __device__` i.e., the `__device__` qualifier is implicit when the `__managed__` qualifier is specified.

`__constant__`

The `__constant__` qualifier, optionally used together with `__device__`, declares a variable that:

- Resides in constant memory space,
- Has the lifetime of an application,
- Is accessible from all the threads within the grid and from the host through the runtime library (i.e., `cudaGetSymbolAddress()` / `cudaGetSymbolSize()` / `cudaMemcpyToSymbol()` / `cudaMemcpyFromSymbol()`).

`__shared__`

The `__shared__` qualifier, optionally used together with `__device__`, 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.

When declaring a variable in shared memory as an external array such as

```
extern __shared__ float shared[];
```

The size of the array is determined at launch time (see Execution Configuration). All variables declared in this fashion, start at the same address in memory, so that the layout of the variables in the array must be explicitly managed through offsets.

`__restrict__`

`nvcc` supports restricted pointers via the `__restrict__` keyword.

Restricted pointers were introduced in C99 to alleviate the aliasing problem that exists in C-type languages, and which inhibits all kind of optimization from code re-ordering to common sub-expression elimination.

Since register pressure is a critical issue in many CUDA codes, use of restricted pointers can have negative performance impact on CUDA code, due to reduced occupancy.

B.3. Built-in Vector Types

B.3.1. `char`, `short`, `int`, `long`, `longlong`, `float`, `double`

These are vector types derived from the basic integer and floating-point types. They are structures and the 1st, 2nd, 3rd, and 4th components are accessible through the fields `x`, `y`, `z`, and `w`, respectively. They all come with a constructor function of the form `make_<type name>`; for example,

```
int2 make_int2(int x, int y);
```

This creates a vector of type `int2` with value `(x, y)`.

In host code, the alignment requirement of a vector type is equal to the alignment requirement of its base type. This is not always the case in device code as detailed in Table B.1.

B.3.2. `dim3`

This type is an integer vector type based on `uint3` that is used to specify dimensions. When defining a variable of type `dim3`, any component left unspecified is initialized to `1`.

Table 4.1 Alignment Requirements in Device Code

Type	**Alignment**
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4
short3, ushort3	2
short4, ushort4	8
int1, uint1	4
int2, uint2	8
int3, uint3	4
int4, uint4	16
long1, ulong1	4 if sizeof(long) is equal to sizeof(int) 8, otherwise
long2, ulong2	8 if sizeof(long) is equal to sizeof(int), 16, otherwise
long3, ulong3	4 if sizeof(long) is equal to sizeof(int), 8, otherwise
long4, ulong4	16
longlong1, ulonglong1	8
longlong2, ulonglong2	16
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16

B.4. Built-in Variables

Built-in variables specify the grid and block dimensions as well as the block and thread indices. They are only valid within functions that are executed on the device.

`gridDim`

This variable is of type `dim3` and contains the dimensions of the grid.

`blockIdx`

This variable is of type `uint3` and contains the block index within the grid.

`blockDim`

This variable is of type `dim3` and contains the dimensions of the block.

`threadIdx`

This variable is of type `uint3` and contains the thread index within the block.

`warpSize`

This variable is of type `int` and contains the warp size in threads.

Note: for complete details of CUDA C Extensions refer to CUDA C Programming Guide
http://docs.nvidia.com/cuda/cuda-c-programming-guide