# How to Query Device Properties and Handle Errors in CUDA C/C++

In this third post of the CUDA C/C++ series we discuss various characteristics of the wide range of CUDA-capable GPUs, how to query device properties from within a CUDA C/C++ program, and how to handle errors.

## Querying Device Properties

In our last post, [about performance metrics](https://developer.nvidia.com/blog/parallelforall/how-implement-performance-metrics-cuda-cc/), we discussed how to compute the theoretical peak bandwidth of a GPU. This calculation used the GPU’s memory clock rate and bus interface width, which we obtained from product literature. The following CUDA C++ code demonstrates a more general approach, calculating the theoretical peak bandwidth by querying the attached device (or devices) for the needed information.

#include <stdio.h>

int main() {

int nDevices;

cudaGetDeviceCount(&nDevices);

for (int i = 0; i < nDevices; i++) {

cudaDeviceProp prop;

cudaGetDeviceProperties(&prop, i);

printf("Device Number: %d\n", i);

printf(" Device name: %s\n", prop.name);

printf(" Memory Clock Rate (KHz): %d\n",

prop.memoryClockRate);

printf(" Memory Bus Width (bits): %d\n",

prop.memoryBusWidth);

printf(" Peak Memory Bandwidth (GB/s): %f\n\n",

2.0\*prop.memoryClockRate\*(prop.memoryBusWidth/8)/1.0e6);

}

}

This code uses the function [cudaGetDeviceCount()](http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#group__CUDART__DEVICE_1gd3f924ba23b94793b0c93b2addba0741) which returns in the argument nDevices the number of CUDA-capable devices attached to this system. Then in a loop we calculate the theoretical peak bandwidth for each device.The body of the loop uses [cudaGetDeviceProperties()](http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#group__CUDART__DEVICE_1g929dcd9a191e17b7498e7ccaa3d16350) to populate the fields of the variable prop, which is an instance of the struct [cudaDeviceProp](http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#structcudaDeviceProp). The program uses only three of cudaDeviceProp's many members: name, memoryClockRate, and memoryBusWidth.

When I compile (using any recent version of the CUDA nvcc compiler, e.g. 4.2 or 5.0rc) and run this code on a machine with a single NVIDIA Tesla C2050, I get the following result.

Device Number: 0

Device name: Tesla C2050

Memory Clock Rate (KHz): 1500000

Memory Bus Width (bits): 384

Peak Memory Bandwidth (GB/s): 144.00

This is the same value for theoretical peak bandwidth that we calculated in [the previous post](https://developer.nvidia.com/blog/parallelforall/how-implement-performance-metrics-cuda-cc/). When I compile and run the same code on my laptop computer, I get the following output.

Device Number: 0

Device name: NVS 4200M

Memory Clock Rate (KHz): 800000

Memory Bus Width (bits): 64

Peak Memory Bandwidth (GB/s): 12.800000

There are many other fields in the cudaDeviceProp struct which describe the amounts of various types of memory, limits on thread block sizes, and many other characteristics of the GPU. We could extend the above code to print out all such data, but the deviceQuery code sample provided with the NVIDIA CUDA Toolkit already does this.

## Compute Capability

We will discuss many of the device attributes contained in the cudaDeviceProp type in future posts of this series, but I want to mention two important fields here: major and minor. These describe the compute capability of the device, which is typically given in major.minor format and indicates the architecture generation. The first CUDA-capable device in the Tesla product line was the Tesla C870, which has a compute capability of 1.0. The first double-precision capable GPUs, such as Tesla C1060, have compute capability 1.3. GPUs of the Fermi architecture, such as the Tesla C2050 used above, have compute capabilities of 2.x, and GPUs of the Kepler architecture have compute capabilities of 3.x. Many limits related to the execution configuration vary with compute capability, as shown in the following table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Tesla C870 | Tesla C1060 | Tesla C2050 | Tesla K10 | Tesla K20 |
| Compute Capability | 1.0 | 1.3 | 2.0 | 3.0 | 3.5 |
| Max Threads per Thread Block | 512 | 512 | 1024 | 1024 | 1024 |
| Max Threads per SM | 768 | 1024 | 1536 | 2048 | 2048 |
| Max Thread Blocks per SM | 8 | 8 | 8 | 16 | 16 |

In the [first post of this series](https://developer.nvidia.com/blog/parallelforall/easy-introduction-cuda-c-and-c/) we mentioned that the grouping of threads into thread blocks mimics how thread processors are grouped on the GPU. This group of thread processors is called a streaming multiprocessor, denoted SM in the table above. The CUDA execution model issues thread blocks on multiprocessors, and once issued they do not migrate to other SMs. Multiple thread blocks can concurrently reside on a multiprocessor subject to available resources (on-chip registers and shared memory) and the limit shown in the last row of the table. The limits on threads and thread blocks in this table are associated with the compute capability and not just a particular device: all devices of the same compute capability have the same limits. There are other characteristics, however, such as the number of multiprocessors per device, that depend on the particular device and not the compute capability. All of these characteristics, whether defined by the particular device or its compute capability, can be obtained using the cudaDeviceProp type.

You can generate code for a specific compute capability by using the nvcc compiler option -arch=sm\_xx, where xx indicates the compute capability (without the decimal point). To see a list of compute capabilities for which a particular version of nvcc can generate code, along with other CUDA-related compiler options, issue the command nvcc --help and refer to the -arch entry.

When you specify an execution configuration for a kernel, keep in mind (and query at run time) the limits in the table above. This is especially important for the second execution configuration parameter: the number of threads per thread block. If you specify too few threads per block, then the limit on thread blocks per multiprocessor will limit the amount of parallelism that can be achieved. If you specify too many threads per thread block, well, that brings us to the next section.

## Handling CUDA Errors

All CUDA C Runtime API functions have a return value which can be used to check for errors that occurr during their execution.  In the example above we can check for successful completion of cudaGetDeviceCount() like this:

cudaError\_t err = cudaGetDeviceCount(&nDevices);

if (err != cudaSuccess) printf("%s\n", cudaGetErrorString(err));

We check to make sure cudaGetDeviceCount() returns the value [cudaSuccess](http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#group__CUDART__TYPES_1g3f51e3575c2178246db0a94a430e0038). If there is an error, then we call the function [cudaGetErrorString()](http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#group__CUDART__ERROR_1g53be96d1df85202fd48c69043b659a79) to get a character string describing the error.

Handling kernel errors is a bit more complicated because kernels execute asynchronously with respect to the host. To aid in error checking kernel execution, as well as other asynchronous operations, the CUDA runtime maintains an error variable that is overwritten each time an error occurs. The function [cudaPeekAtLastError()](http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#group__CUDART__ERROR_1gd46da25e4ff94015fd069f54819d6b4f) returns the value of this variable, and the function [cudaGetLastError()](http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#group__CUDART__ERROR_1g1a56a496eabe915752bf72d9574d1b69) returns the value of this variable and also resets it to cudaSuccess.

We can check for errors in the saxpy kernel used [in the first post of this series](https://developer.nvidia.com/blog/parallelforall/easy-introduction-cuda-c-and-c/) as follows.

saxpy<<<(N+255)/256, 256>>>(N, 2.0, d\_x, d\_y);

cudaError\_t errSync = cudaGetLastError();

cudaError\_t errAsync = cudaDeviceSynchronize();

if (errSync != cudaSuccess)

printf("Sync kernel error: %s\n", cudaGetErrorString(errSync);

if (errAsync != cudaSuccess)

printf("Async kernel error: %s\n", cudaGetErrorString(errAsync);

This code checks for both synchronous and asynchronous errors. Invalid execution configuration parameters, e.g. too many threads per thread block, are reflected in the value of errSync returned by cudaGetLastError(). Asynchronous errors which occur on the device after control is returned to the host, such as out-of-bounds memory accesses, require a synchronization mechanism such as [cudaDeviceSynchronize()](http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#group__CUDART__DEVICE_1g32bdc6229081137acd3cba5da2897779), which blocks the host thread until all previously issued commands have completed. Any asynchronous error is returned by cudaDeviceSynchronize(). We can also check for asynchronous errors and reset the runtime error state by modifying the last statement to call cudaGetLastError().

if (errAsync != cudaSuccess)

printf("Async kernel error: %s\n", cudaGetErrorString(cudaGetLastError());

Device synchronization is expensive, because it causes the entire device to wait, destroying any potential for concurrency at that point in your program. So use it with care. Typically, I use preprocessor macros to insert asynchronous error checking only in debug builds of my code, and not in release builds.

Summary

Now you know how to query CUDA device properties and handle errors in CUDA C and C++ programs. These are very important concepts for writing robust CUDA applications.

In the first three posts of this series we have covered some of the basics of writing CUDA C/C++ programs, focusing on the basic programming model and the syntax of writing simple examples. We discussed [timing code and performance metrics in the second post](https://developer.nvidia.com/blog/parallelforall/how-implement-performance-metrics-cuda-cc/), but we have yet to use these tools in optimizing our code. That will change in the next post, where we will look at optimizing data transfers between the host and device.