1.1



1.2 When cudaDeviceReset is removed, none of the prints from the GPU are displayed:

$ ./hello

Hello World from CPU!

$

While printf is still called on the GPU, cudaDeviceReset forces those prints to be flushed from the GPU, to the host, and then output in the user visible console. Without calling cudaDeviceReset (or other functions that force flushing of GPU output), there are no guarantees that these prints will be displayed. Chapter 10 will provide more details on the use and behavior of CUDA’s printf.

1.3 When cudaDeviceReset is replaced with cudaDeviceSynchronize, the prints from the GPU are displayed on the console:

$ ./hello

Hello World from CPU!

Hello World from GPU!

Hello World from GPU!

Hello World from GPU!

Hello World from GPU!

Hello World from GPU!

Hello World from GPU!

Hello World from GPU!

Hello World from GPU!

Hello World from GPU!

Hello World from GPU!

$

cudaDeviceSynchronize is another function (in addition to cudaDeviceReset) that can be used to force GPU prints to be flushed to the user-visible console.

1.4 Removing the device architecture switch (-arch=sm\_20) produces the following output, and fails to produce an executable:

$ nvcc hello.cu -o hello

nvcc warning : The 'compute\_10' and 'sm\_10' architectures are deprecated, and may be removed in a future release.

hello.cu(11): error: calling a \_\_host\_\_ function("printf") from a \_\_global\_\_ function("helloFromGPU") is not allowed

1 error detected in the compilation of "tmpxft\_00005521\_00000000-6\_hello.cpp1.ii".

When no architecture flag is specified, nvcc will default to using compute capability 1.0. However, calling printf from the GPU is not supported by compute capability 1.0. The compiler therefore prints an error and exits before the executable is generated. This exercise illustrates the different capabilities that progressively higher compute capabilities can add to a CUDA application.

1.5 nvcc supports the following input file suffixes: .cu, .cup, .c, .cc, .cxx, .cpp, .gpu, .ptx, .o, .obj, .a, .lib, .res, and .so. It can be used as a general-purpose compiler to build more than just CUDA source code.

1.6 For this solution, use a conditional to ensure that only thread 5 executes the printf, and add the thread ID to the print statement:

#include <stdio.h>

\_\_global\_\_ void helloFromGPU(void)

{

if (threadIdx.x == 5)

{

printf("Hello World from GPU thread %d!\n", threadIdx.x);

}

}

int main(void)

{

printf("Hello World from CPU!\n");

helloFromGPU<<<1, 10>>>();

cudaDeviceReset();

return 0;

}

2.1 With block.x set to 1023, sumArraysOnGPU-timer produces the following output on an M2050:

$ ./sumArraysOnGPU-timer

./sumArraysOnGPU-timer Starting...

Using Device 0: Tesla M2050

Vector size 16777216

initialData Time elapsed 0.356134 sec

sumArraysOnHost Time elapsed 0.023177 sec

sumArraysOnGPU <<< 16401, 1023 >>> Time elapsed 0.003164 sec

Arrays match.

With block.x set to 1024, the following output is produced:

$ ./sumArraysOnGPU-timer

./sumArraysOnGPU-timer Starting...

Using Device 0: Tesla M2050

Vector size 16777216

initialData Time elapsed 0.356447 sec

sumArraysOnHost Time elapsed 0.023294 sec

sumArraysOnGPU <<< 16384, 1024 >>> Time elapsed 0.002416 sec

Arrays match.

Note the performance difference between the kernels with different block configurations. With block.x set to 1024 the kernel runs 1.31 times faster. Because threads are scheduled in groups of 32, running blocks with 1023 threads leads to two suboptimal application characteristics. First, the last warp in every thread block will have a disabled thread that performs no work because 1024 is not evenly divisible into warps. Second, because there are fewer threads per block the application will require more blocks to fully process the input. Running more thread blocks causes longer execution times as only a fixed number of blocks can run concurrently.

2.2