README for cuPrintf

Introduction

cuPrintf allows you to to add printf-equivalent cuPrintf calls to your CUDA C code.

In addition to the readme and the license agreement, the cuPrintf zip file package includes two source files: cuPrintf.cuh and cuPrintf.cu. Drop these files into your source directory or your include path and start making calls to cuPrintf in your CUDA C code.

This sample code works on all CUDA-enabled GPUs.

This cuPrintf code works significantly better on GPUs with SM arch 1.1 and later, so always build your code with "-arch=sm_11" or higher if possible.

This cuPrintf code should work with CUDA 2.3 or newer, and is supported on all platforms on which the CUDA Toolkit is supported.

Use

The cuPrintf package consists of two device functions (i.e. called from within a CUDA kernel) and three host functions (i.e. called from within the host application). These are packaged in a single *cuPrintf.cu* file, along with declarations included in a separate *cuPrintf.cuh* header file. To use cuPrintf in your application, you must do one of the following:

a) Either: Include the header-file *cuPrintf.cuh* at the top of your device code, and add *cuPrintf.cu* to your makefile or build command-line so that the file is included in your program.
b) Or: Directly "*#include cuPrintf.cu"* at the top of your device code. In this case you should not add this file to your makefile/build-command, and you should take care to only include it once in your entire project.

It is strongly recommended that if you have a GPU capable of architecture 1.1 or 1.3, you build targeting your GPU's preferred architecture. Lack of atomic support in architecture 1.0 leads to very inefficient use of the buffer and loss of ordering of the printf output.

Incorporating cuPrintf in your application requires explicit initialisation and display calls in your host-side code. A simple example program would be:

```
#include "cuPrintf.cu"

__global__ void testKernel(int val)
{
        cuPrintf("Value is: %d\n", val);
}

int main()
{
        cudaPrintfInit();

        testKernel<<< 2, 3 >>>(10);
        cudaPrintfDisplay(stdout, true);

        cudaPrintfEnd();
        return 0;
}
```

Calls to *cudaPrintfInit* and *cudaPrintfEnd* are needed only once per application. It is recommended that *cudaPrintfDisplay* is called after each synchronization point to avoid buffer overflow. Note that *cudaPrintfDisplay* implicitly forces context synchronization.

Limitations / Known Issues

Currently, the following limitations and restrictions apply to cuPrintf:

1. Buffer size is rounded up to the nearest factor of 256
2. Arguments associated with "%s" string format specifiers must be of type (const char *)
3. To print the value of a (const char *) pointer, it must first be converted to (char *). All (const char *) arguments are interpreted as strings
4. Non-zero return code does not match standard C printf()
5. Cannot asynchronously output the printf buffer (i.e. while kernel is running)
6. Calling *cudaPrintfDisplay* implicitly issues a *cudaThreadSynchronize()*
7. Restrictions applied by *cuPrintfRestrict* persist between launches. To clear these from the host-side, you must call *cudaPrintfEnd()* then *cudaPrintfInit()* again
8. cuPrintf output is undefined if multiple modules are loaded into a single context
9. Compile with *"-arch=sm_11"* or better when possible. Buffer usage is far more efficient and register use is lower
10. Supported format specifiers are: "cdiouxXeEfgGaAs"
11. Behaviour of format specifiers, especially justification/size specifiers, are dependent on the host machine's implementation of printf
12. cuPrintf requires applications to be built using the CUDA runtime API

Function Descriptions

All functions and information on their usage is also included in the *cuPrintf.cuh* header file.

cuPrintf

Synopsis:

```
    __device__ int cuPrintf(const char *fmt, ...);
```

Arguments:

fmt – Format string, as per normal printf() function

... – Between 0 and 10 arguments of any type, as per normal printf() function

Return:

0 on failure

>0 on success

Description:

This has equivalent functionality to the well-known C *printf()* function, taking a format string which contains format specifiers and outputting a corresponding string. Format specifiers supported are "cdiouxXeEfgGaAs", with all size and justification modifiers permitted by your host compiler. Please see your host compiler documentation for a complete description of printf() functionality.

Certain restrictions apply, along with some behavioural differences compared to standard *printf*:

1. String formats, "%s", *must* be accompanied by a (const char *) argument. Strings declared as (char *) must be cast to (const char *) when matching "%s".
2. Corollary to (1), all (const char *) arguments are interpreted as strings; therefore to output the address of any (const char *), it must first be cast to (char *).
3. No more than 10 arguments are supported after the format string.
4. The only format specifiers supported are "cdiouxXeEfgGaAs". All others are output according to the host-compiler's printf rules (typically the format specifier is output directly).
5. The return value does not mimic standard C printf() (which returns the number of characters output). The only meaningful return is 0, indicating a failure to output, or non-zero, indicating success.

cuPrintfRestrict

Synopsis:

```
    __device__ void cuPrintfRestrict(int threadid, int blockid);
```

Arguments:

threadid – Thread ID for which output is permitted. Pass the constant *CUPRINTF_UNRESTRICTED* to enable all threads.

blockid – Block ID for which output is permitted. Pass the constant *CUPRINTF_UNRESTRICTED* to enable all blocks.

Description:

This is a utility function permitting run-time control over filtering for cuPrintf output. Typically, a cuPrintf() call will be executed by all threads in a warp or even a block, resulting in a large quantity of output from each function call. This is not always desirable, so the *cuPrintfRestrict* mechanism allows restriction of the output to a specified thread, block or both.

Thread ID is calculated as the linear expansion of the block dimension. Block ID is likewise calculated as the linear expansion of the grid dimension. Therefore:

*threadid = threadIdx.x + (threadIdx.y * blockDim.x) + (threadIdx.x * threadIdx.y * blockDim.y)*
*blockid = blockIdx.x + (blockIdx.y * gridDim.x)*

For output from a given thread to appear, the thread must match both *threadid* and *blockid*. Setting either to the constant CUPRINTF_UNRESTRICTED automatically satisfies all threads for that type. Note that all output can be disabled by selecting a non-existent *threadid* or *blockid*.

cudaPrintfInit

Synopsis:

```
__host__ cudaError_t cudaPrintfInit(size_t bufferLen=1048576);
```

Arguments:

(optional) bufferLen – Specify the size in bytes for the buffer used for receiving cuPrintf output. Default is 1 megabyte. Note that the value passed here is rounded up to a factor of 256.

Return:

cudaSuccess if all is well
Errors arise from improper initialisation of either CUDA or cuPrintf

Description:

cuPrintf does not automatically copy data from the GPU to the screen – this must be done explicitly. All output, therefore, is buffered until *cudaPrintfDisplay()* is called. The buffer is circular: overflow will overwrite the oldest data first.

The size passed here will cause *bufferLen* bytes to be allocated on both the host and the device. Buffer size is rounded up to the nearest factor of 256.

Note that for architecture 1.0 builds, this buffer is divided equally between all threads whether or not a given thread actually uses it. For architecture 1.1 and above, the buffer is accessed linearly so all threads share the one space (which is more efficient).

cudaPrintfEnd

Synopsis:

```
	__host__ void cudaPrintfEnd();
```

Arguments:

None

Description:

Call this to free up the memory allocated by *cudaPrintfInit*. If you need to change the size of the output buffer, you must call *cudaPrintfEnd* and then call *cudaPrintfInit* again.

cudaPrintfDisplay

Synopsis:

```
	__host__ cudaError_t cudaPrintfDisplay(FILE *outputFP=NULL, bool
showThreadID=false);
```

Arguments:

(optional) outputFP – File descriptor to which the cuPrintf log should be sent. Pass NULL to select *stdout*

(optional) showThreadID – If this is *true*, output will automatically be prefixed by an indicator of "[blockid,threadid]"

Return:

cudaSuccess if all is well

Description:

This dumps the current contents of the output buffer to the requested file descriptor. Multiple launches may be made before calling *cudaPrintfDisplay*, and provided that sufficient space exists in the buffer all output will be recorded. The exception here is for architecture 1.0 GPUs: for these, you must dump the buffer between each launch or the output behaviour is unspecified.

Output will appear in the order in which it was issued by the threads in the kernel, therefore thread-execution ordering is visible. The exception to this is on architecture 1.0 GPUs: lack of atomics prevents this, and output is issued in-order within each thread.

For convenience, the *showThreadID* flag enables display of the origin of each line of output. The default output is to *stdout* (the screen) and with ID display turned off.

Note: Calling *cudaPrintfDisplay* implicitly synchronizes the CUDA context (as if *cudaThreadSynchronize()* had been called).