CUDA TOOLKIT 3.2 READINESS FOR CUDA APPLICATIONS

August 20, 2010

Technical Brief

TECHNICAL BRIEF

INTRODUCTION

In NVIDIA® CUDA™ Toolkit version 3.2 and the accompanying 260.xx release of the CUDA driver, changes are being made to the CUDA driver to support 64-bit addressing. This enables Fermi-based GPUs to address more than four gigabytes of system memory using zero-copy, and it enables support for GPUs with greater than four gigabytes of device memory such as the 6GB NVIDIA Tesla™ C2070.

In addition, the syscall linking mechanism introduced in CUDA Toolkit 3.1 to support in-kernel **printf()** is changing to allow more general syscall support. The new linking mechanism allows support for in-kernel **malloc()** and **free()** beginning in CUDA Toolkit 3.2, and it will allow a future version of the CUDA Toolkit to support additional syscalls.

CUDA applications built against earlier CUDA Toolkits that need neither large memory support nor in-kernel **printf()** will continue to work unchanged with new CUDA drivers. Applications requiring these features must be compiled against CUDA Toolkit 3.2. In most cases, recompilation is all that is required. This technical brief describes the situations where minor application-level changes might be necessary and how they can be handled with minimal effort.

Note that CUDA C Runtime API applications generally require no changes other than recompilation to take advantage of the features described in this document.[1]

[1] Applications using both the CUDA C Runtime API *and* the CUDA Driver API via driver/runtime interoperability are considered equivalent to Driver API applications for the purposes of this document unless otherwise noted.

64-BIT ADDRESSING

The API-level changes to support 64-bit addressing are completely transparent to applications that use the CUDA C Runtime API, requiring only a recompile against the CUDA Toolkit 3.2. As such, this section focuses on the changes to the Driver API and the corresponding changes that might be needed in Driver API applications.

Note that applications using CUDA Toolkit 3.1 or earlier can make full use of large-memory (greater than 4GB) devices by sharing the memory across multiple CUDA contexts, each of which gets its own 4GB virtual address space.

MOTIVATION

The CUDA Driver API defines a type called **CUdeviceptr** that is used to store pointers into device memory. However, in CUDA Toolkit 3.1 and prior, this type is equivalent to the type **unsigned int**, which limits it to a length of 32 bits, thereby limiting the addressable memory to 4GB. To support larger memories, 64-bit pointers are required, thus requiring a new version of the CUDA Driver API with a new definition of **CUdeviceptr**.

BACKWARD COMPATIBILITY

Every effort has been made to ensure the maximum degree of backward compatibility for existing applications with this new CUDA Driver API. First and foremost, applications compiled against CUDA Toolkit 3.1 or prior will continue to work with newer CUDA drivers by way of legacy entry points into the CUDA driver. Such applications, however, will continue to be limited to 4GB of device memory even on devices with greater than 4GB.

To support larger device memories, applications will need to be recompiled against the new CUDA Driver API, which is defined in the **cuda.h** header to be included with version 3.2 of the CUDA Toolkit. This new API is name-compatible with the legacy API, so very few (if any) changes should be necessary to port applications to compile against the new API, and any bugs should be readily evident at compile time and easy to fix. The only exception to this is that applications compiled against the new API will no longer be able to use the “mixed-bitness mode” that was allowed in the legacy Driver API, whereby a 64-bit host application was allowed to load 32-bit device code: with the new API, the CUDA context inherits the bitness of the host application, meaning device code of a bitness different than the host application cannot be loaded.

In the unlikely event that porting an existing application to the new API is impractical (e.g., if the application absolutely requires the use of mixed-bitness mode for some reason), CUDA Driver API applications can continue using the legacy Driver API using the compiler flag **-DCUDA_FORCE_API_VERSION=3010** . This method should be avoided if at all possible, however, since applications using the legacy API will always be limited to a maximum of 4GB of accessible device memory per context and since any new features added to the CUDA Driver API at a future date will not be available via the legacy API.[2]

API CHANGES

CUdeviceptr is now defined as **uintptr_t**, meaning that its size is now equivalent to **sizeof(void*)**. Additionally, Driver API functions that accept or return memory sizes, such as **cuMemAlloc()** and **cuMemGetInfo()**, now use **size_t** for this purpose rather than **unsigned int**.

LIKELY APPLICATION CHANGES REQUIRED

Based on our experience with porting over 100 programs to the new API, the following are the most likely compile-time issues that might be encountered when porting applications:

Problem: The application passes **CUdeviceptr** values to kernels using **cuParamSeti()**, but **cuParamSeti** only supports 32-bit values.

Solution: Use **cuParamSetv()** instead.

Example: Incorrect code:

```
CUdeviceptr d_ptr;
cuMemAlloc(&d_ptr, nbytes);
ALIGN_UP(offset, __alignof(d_ptr));
cuParamSeti(func, offset, d_ptr);
```

Correct code:

```
CUdeviceptr d_ptr;
cuMemAlloc(&d_ptr, nbytes);
ALIGN_UP(offset, __alignof(d_ptr));
cuParamSetv(func, offset, &d_ptr, sizeof(d_ptr));
```

[2] Applications using driver/runtime interoperability that require the legacy API must build against the CUDA Toolkit 3.1 rather than using the CUDA_FORCE_API_VERSION compiler flag, since version 3.2 of the CUDA C Runtime library is compiled against the new Driver API rather than the legacy Driver API.

Problem: The application passes a pointer to a variable of type **unsigned int** to **cuMemGetInfo()**, **cuMemGetAddressRange()**, **cuModuleGetGlobal()**, etc.

Solution: Change the type of the variable from **unsigned int** to **size_t**. (Important note: simply casting the pointer passed to these functions from **(unsigned int*)** to **(size_t*)** is unsafe and can result in stack corruption. Change the data type of the underlying variable rather than casting the pointer to that variable.)

Example: Incorrect code:

```
unsigned int free, total;
cuMemGetInfo(&free, &total);
```

Incorrect code:

```
unsigned int free, total;
cuMemGetInfo((size_t*)&free, (size_t*)&total);
```

Correct code:

```
size_t free, total;
cuMemGetInfo(&free, &total);
```

SYSCALL LINKING FOR IN-KERNEL PRINTF()

The syscall linking mechanism used to implement in-kernel **printf()** for Fermi-based` GPUs is being changed in CUDA Toolkit 3.2 to a more general mechanism that allows support for additional in-kernel syscalls such as **malloc()** and **free()**.

Since the only in-kernel syscall supported prior to CUDA Toolkit 3.2 was **printf()** and since **printf()** is typically used only for debugging purposes rather than in production code, support for the old-style linking mechanism has been removed in CUDA Toolkit 3.2. This means that CUBINs that call **printf()** that were compiled with CUDA Toolkit 3.1 will fail to load with CUDA drivers of version 260.xx or higher, returning the driver error **CUDA_ERROR_INVALID_IMAGE**. Recompiling the CUBINs and the applications that load them with CUDA Toolkit 3.2 will automatically enable the new linking mechanism and allow the CUBINs to load successfully.

Notice

Trademarks

Copyright