FERMI COMPATIBILITY GUIDE FOR CUDA APPLICATIONS

DA-05607-001_v1.5 | May 2011

Application Note

DOCUMENT CHANGE HISTORY

DA-05607-001_v1.5

Version	Date	Authors	Description of Change
1.0		CW,EY	Initial public release.
1.1 1.3		CW	See Appendix A.
1.4	December 4, 2010	TechPubs	New Template, minor edits.
1.5	May 3, 2011	CW	Updated cross-references.

TABLE OF CONTENTS

Chapter 1. Fermi Compatibility 1
1.1 About This Document 1
1.2 Application Compatibility on Fermi 1
1.3 Verifying Fermi Compatibility for Existing Applications 2
1.3.1 Applications Using Earlier CUDA Toolkit Versions 2
1.3.2 Applications Using CUDA Toolkit 3.0 or Later 2
1.3.3 Kernels 2
1.4 Building Applications with Fermi Support 4
1.4.1 CUDA Runtime API Applications 4
1.4.2 CUDA Driver API Application 6
APPENDIX A. Revision History 8
A.1 Version 1.1 8
A.2 Version 1.3 8

Chapter 1. FERMI COMPATIBILITY

1.1 ABOUT THIS DOCUMENT

This application note–*Fermi Compatibility Guide for CUDA Applications* is intended to help developers ensure that their NVIDIA® CUDA™ applications will run effectively on GPUs based on the NVIDIA® Fermi Architecture. This document provides guidance to developers who are already familiar with programming in CUDA C/C++ and want to make sure that their software applications are compatible with Fermi.

Important Note: Prior to the introduction of the Fermi architecture, all NVIDIA Tesla™-branded products were based on the Tesla architecture. For the purposes of this document, the term "Tesla" refers only to the GPU architecture and not to any particular NVIDIA product. Hereinafter, Tesla refers to devices of compute capability 1.x, and Fermi refers to devices of compute capability 2.x.

1.2 APPLICATION COMPATIBILITY ON FERMI

The NVIDIA CUDA C compiler, `nvcc`, can be used to generate both architecture-specific **cubin** files and forward-compatible PTX versions of each kernel.

Applications that already include PTX versions of their kernels should work as-is on Fermi-based GPUs. Applications that only support specific GPU architectures via **cubin** files, however, will either need to provide a PTX version of their kernels that can be just-in-time (JIT) compiled for Fermi and future GPUs or to be updated to include Fermi-specific **cubin** versions of their kernels. For this reason, to ensure forward compatibility with CUDA architectures introduced after the application has been released, it is recommended that all applications support launching PTX versions of their kernels.

Each **cubin** file targets a specific compute capability version and is forward-compatible only with CUDA architectures of the same major version number; e.g., **cubin** files that target compute capability 1.0 are supported on all compute-capability 1.x (Tesla) devices but are not supported on compute-capability 2.x (Fermi) devices.

1.3 VERIFYING FERMI COMPATIBILITY FOR EXISTING APPLICATIONS

The first thing to do is to check that Fermi-compatible device code is compiled in to the application. The following sections show how to deal with applications built with different CUDA Toolkit versions.

1.3.1 Applications Using Earlier CUDA Toolkit Versions

CUDA applications built using the CUDA Toolkit versions 2.1 through 2.3 are compatible with Fermi as long as they are built to include PTX versions of their kernels. NVIDIA driver versions 195.xx or newer allow the application to use the PTX JIT code path. To test that PTX JIT is working for your application, you can do the following:

- Download and install the latest driver from http://www.nvidia.com/drivers (use version 195.xx or later).
- Set the system environment variable `CUDA_FORCE_PTX_JIT=1`
- Launch your application.

When starting a CUDA application for the first time with the above environment flag, the CUDA driver will JIT compile the PTX for each CUDA kernel that is used into native **cubin** code. The generated **cubin** for the target GPU architecture is cached by the CUDA driver. This cache persists across system shutdown/restart events.

1.3.2 Applications Using CUDA Toolkit 3.0 or Later.

CUDA applications built using the CUDA Toolkit version 3.0 or later are compatible with Fermi as long as they are built to include kernels in either Fermi-native **cubin** format (see Section 1.4) or PTX format (see Section 1.3.1) or both.

1.3.3 Kernels

Check that kernels that communicate among threads in a warp use `volatile`.

When threads within a warp need to communicate values with each other via shared or global memory, a common optimization is to omit `__syncthreads()` after writing these values to memory (see Sections 5.4.3 and D.2.1.2 of the CUDA C Programming Guide).

In these cases, `__syncthreads()` can be omitted because of the synchronicity of execution of the threads in a warp.

A common application of this optimization is in parallel reduction, which is a common data-parallel operation covering a set of problems in which each output depends on all inputs (e.g., finding the sum of a large group of numbers, counting the instances of value *n* among a large set of values, etc.). Such applications often employ code similar to the example at the end of this section (which is a simplified excerpt from the `reduction` sample from the GPU Computing SDK).

If your kernels implement this sort of optimization when passing values among threads in a warp using shared or global memory, it is essential that the pointer into that memory is declared with the `volatile` qualifier (shown in red below) to force the compiler to write the intermediate values out to memory after each step rather than holding the values (`smem[tid]` in the example below) in registers.

Code such as this that omits the `volatile` qualifier will not work correctly on Fermi due to enhanced compiler optimizations. In the example below, the volatile qualifier tells the compiler that it must store `smem[tid]` back to shared memory after every assignment rather than simply allocating a register and optimizing away the stores to shared memory.

```
__device__ void reduce(float *g_idata, float *g_odata)
{
    unsigned int tid = threadIdx.x;
    extern __shared__ float sdata[];

    sdata[tid] = g_idata[...]; // assign initial value

    __syncthreads();

    // do reduction in shared mem.  this example assumes
    // that the block size is 256; see the "reduction"
    // sample in the GPU Computing SDK for a complete and
    // general implementation

    if (tid < 128) {sdata[tid]+=sdata[tid+128];} __syncthreads();
    if (tid <  64) {sdata[tid]+=sdata[tid+ 64];} __syncthreads();
    if (tid <  32) {
        // no __syncthreads() necessary after each of the
        // following lines as long as we access the data via
        // a pointer declared as volatile because the 32 threads
        // in each warp execute in lock-step with each other
        volatile float *smem = sdata;
        smem[tid] += smem[tid + 32];
        smem[tid] += smem[tid + 16];
        smem[tid] += smem[tid +  8];
        smem[tid] += smem[tid +  4];
```

```
        smem[tid] += smem[tid +  2];
        smem[tid] += smem[tid +  1];
    }

    // write result for this block to global mem
    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];
}
```

1.4 BUILDING APPLICATIONS WITH FERMI SUPPORT

1.4.1 CUDA Runtime API Applications

The compilers included in the CUDA Toolkit 2.1, 2.2, and 2.3 generate **cubin** files native to the Tesla architecture but cannot generate **cubin** files native to the Fermi architecture (this requires CUDA Toolkit 3.0 or later). To allow support for Fermi and future architectures when using the 2.x versions of the CUDA Toolkit, the compiler can generate a PTX version of each kernel. By default, the PTX version is included in the executable and is available to be run on Fermi devices via just-in-time (JIT) compilation.

Beginning with version 3.0 of the CUDA Toolkit, `nvcc` can generate **cubin** files native to the Fermi architecture as well. When using the CUDA Toolkit 3.0 or later, to ensure that `nvcc` will generate **cubin** files for all released GPU architectures as well as a PTX version for future GPU architectures, specify the appropriate "`-arch=sm_xx`" parameter on the `nvcc` command line as shown below.

When a CUDA application launches a kernel, the CUDA Runtime library (CUDART) determines the compute capability of each GPU in the system and uses this information to find the best matching **cubin** or PTX version of the kernel. If a **cubin** file supporting the architecture of the GPU on which the application is launching the kernel is available, it is used; otherwise, the CUDA Runtime will load the PTX and JIT compile the PTX to the **cubin** format before launching it on the GPU.

Below are the compiler settings to build `cuda_kernel.cu` to run on Tesla devices natively and Fermi devices via PTX. The main advantage of providing the native code is to save the end user the time it takes to PTX JIT a CUDA kernel that has been compiled to PTX. However, since the CUDA driver will cache the native ISA generated as a result of the PTX JIT, this is mostly a one-time cost. There will still be some additional per-invocation overhead, as the CUDA Runtime will need to check the architecture of the current GPU and explicitly call the best-available version of the CUDA kernel.

Windows:

```
nvcc.exe -ccbin "C:\vs2008\VC\bin" -I"C:\CUDA\include"
-Xcompiler "/EHsc /W3 /nologo /O2 /Zi /MT"
-arch=sm_10
--compile -o "Release\cuda_kernel.cu.obj" "cuda_kernel.cu"
```

Mac/Linux:

```
/usr/local/cuda/bin/nvcc
-arch=sm_10
--compiler-options -fno-strict-aliasing -I.
-I/usr/local/cuda/include -DUNIX -O2
-o release/cuda_kernel.cu.o -c cuda_kernel.cu
```

Note: the `nvcc` command-line option "`-arch=sm_xx`" is a shorthand equivalent to the following more explicit `-gencode` command-line options.

```
-gencode=arch=compute_xx,code=sm_xx
-gencode=arch=compute_xx,code=compute_xx
```

The `-gencode` options must be used instead of `-arch` if you want to compile **cubin** or PTX code for multiple target architectures, as shown below.

Alternatively, with version 3.0 of the CUDA Toolkit, the compiler can build `cuda_kernel.cu` to run on both Tesla devices and Fermi devices natively as shown below. This example also builds in forward-compatible PTX code.

Windows:

```
nvcc.exe -ccbin "C:\vs2008\VC\bin" -I"C:\CUDA\include"
-Xcompiler "/EHsc /W3 /nologo /O2 /Zi /MT"
-gencode=arch=compute_10,code=sm_10
-gencode=arch=compute_10,code=compute_10
-gencode=arch=compute_20,code=sm_20
-gencode=arch=compute_20,code=compute_20
--compile -o "Release\cuda_kernel.cu.obj" "cuda_kernel.cu"
```

Mac/Linux:

```
/usr/local/cuda/bin/nvcc
-gencode=arch=compute_10,code=sm_10
-gencode=arch=compute_10,code=compute_10
-gencode=arch=compute_20,code=sm_20
-gencode=arch=compute_20,code=compute_20
--compiler-options -fno-strict-aliasing -I.
-I/usr/local/cuda/include -DUNIX -O2
-o release/cuda_kernel.cu.o -c cuda_kernel.cu
```

Note the distinction in these command lines between the "**code=sm_10**" argument to –gencode, which generates **cubin** files for the specified compute capability, and the "**code=compute_10**" argument, which generates PTX for that compute capability.

1.4.2 CUDA Driver API Application

What steps do I need to take to support Fermi?

Answer: You have several options:

- Compile CUDA kernel files to PTX. While **cubin** files can be generated using the compilers in the CUDA Toolkit 2.1 through 2.3, those **cubin** files are compatible only with Tesla devices, not Fermi devices.

  Refer to the following GPU Computing SDK code samples for examples showing how to use the CUDA Driver API to launch PTX kernels:

  - matrixMulDrv
  - simpleTextureDrv
  - ptxjit

  Use the the compiler settings below to create PTX output files from your CUDA source files:

Windows:

```
nvcc.exe -ccbin "C:\vs2008\VC\bin" -I"C:\CUDA\include"
-Xcompiler "/EHsc /W3 /nologo /O2 /Zi /MT"
-ptx
-o "cuda_kernel.ptx" "cuda_kernel.cu"
```

Mac/Linux:

```
/usr/local/cuda/bin/nvcc
-ptx
--compiler-options -fno-strict-aliasing -I.
-I/usr/local/cuda/include -DUNIX -O2
-o cuda_kernel.ptx cuda_kernel.cu
```

- Compile your CUDA kernels to both **cubin** and PTX output files. This must be specified explicitly at compile time, since nvcc must be called once for each generated output file of either type.

  At runtime, your application will need to explicitly check the compute capability of the current GPU with the following CUDA Driver API function. Refer to the deviceQueryDrv code sample in the GPU Computing SDK for a detailed example of how to use this function.

```
cuDeviceComputeCapability(&major, &minor, dev)
```

Based on the major and minor version returned by this function, your application can choose the appropriate **cubin** or PTX version of each kernel.

To load kernels that were compiled to PTX using the CUDA Driver API, you can use code as in the following example. Calling `cuModuleLoadDataEx` will JIT compile your PTX source files. (Note that there are a few JIT options that developers need to be aware of to properly compile their kernels.) The GPU Computing SDK samples `matrixMulDrv` and `simpleTextureDrv` further illustrate this process.

```
CUmodule cuModule;
CUfunction cuFunction = 0;
string ptx_source;

// Helper function load PTX source to a string
findModulePath ("matrixMul_kernel.ptx",
                module_path, argv, ptx_source));

// We specify PTXJIT compilation with parameters
const unsigned int jitNumOptions = 3;
CUjit_option *jitOptions = new CUjit_option[jitNumOptions];
void **jitOptVals = new void*[jitNumOptions];

// set up size of compilation log buffer
jitOptions[0] = CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES;
int jitLogBufferSize = 1024;
jitOptVals[0] = (void *)jitLogBufferSize;

// set up pointer to the compilation log buffer
jitOptions[1] = CU_JIT_INFO_LOG_BUFFER;
char *jitLogBuffer = new char[jitLogBufferSize];
jitOptVals[1] = jitLogBuffer;

// set up pointer for Maximum # of registers
jitOptions[2] = CU_JIT_MAX_REGISTERS;
int jitRegCount = 32;
jitOptVals[2] = (void *)jitRegCount;

// Loading a module will force a PTX to be JIT
status = cuModuleLoadDataEx(&cuModule, ptx_source.c_str(),
                            jitNumOptions, jitOptions,
                            (void **)jitOptVals);

printf("> PTX JIT log:\n%s\n", jitLogBuffer);
```

APPENDIX A. Revision History

A.1 VERSION 1.1

- Corrected discussion of `CUDA_FORCE_PTX_JIT=1` to indicate that this test can be used with CUDA Driver API applications as well as CUDA Runtime API applications.
- Added section discussing the use of `volatile` for warp-synchronous code omitting `__syncthreads()`.
- Added section discussing the passing of `CUdeviceptr` arguments to kernels through the CUDA Driver API when running device code compiled in 64-bit mode on Fermi.
- Minor clarifications in Section 1.4.1.

A.2 VERSION 1.3

- Removed section discussing the passing of `CUdeviceptr` arguments to kernels through the CUDA Driver API, since the Driver API is changing in CUDA Toolkit 3.2 to be 64-bit-clean. `CUdeviceptr` type-casting of the form `(void*)(size_t)` is no longer necessary.
- Clarified discussion of the use of `volatile`.
- Updated cross-references to CUDA C Programming Guide version 3.2.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2011 NVIDIA Corporation. All rights reserved.