CUDA Compatibility
# Table of Contents

Chapter 1. Overview

1.1. CUDA ................................................................. 1
1.2. Source Compatibility ............................................ 2
1.3. Binary Compatibility ............................................. 3
1.4. CUDA Binary (cubin) Compatibility ....................... 6

Chapter 2. CUDA Toolkit Versioning ............................... 7

Chapter 3. CUDA Enhanced Compatibility Across Minor Releases ........................................ 8

3.1. Existing CUDA Applications within Minor Versions ......................................................... 8
3.2. Handling New CUDA Features ................................... 9
3.3. Using PTX .......................................................... 10
3.4. Dynamic Code Generation ...................................... 11

Chapter 4. Forward-Compatible Upgrade Path ..................... 13

4.1. Installing Packages to Enable Forward Compatible Upgrade ............................................. 14

Chapter 5. Support .......................................................... 17

5.1. Hardware Support ................................................. 17
5.2. Supported GPUs .................................................. 17
5.3. CUDA Application Compatibility ............................. 17
5.4. Feature Support ................................................... 18

Chapter 6. Deployment Considerations ............................... 20

6.1. Disk Image ........................................................ 20
6.2. Modules System ............................................... 20
6.3. Miscellaneous ................................................... 21

Chapter 7. Frequently Asked Questions ............................... 22
List of Figures

Figure 1. Components of CUDA ........................................................................................................2
Figure 2. CUDA Upgrade Path ........................................................................................................13
Figure 3. Forward Compatibility Upgrade Path ............................................................................14
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>CUDA Toolkit and Compatible Driver Versions</td>
<td>5</td>
</tr>
<tr>
<td>Table 2</td>
<td>Compute Capability Support</td>
<td>17</td>
</tr>
<tr>
<td>Table 3</td>
<td>CUDA Application Compatibility Support Matrix</td>
<td>18</td>
</tr>
<tr>
<td>Table 4</td>
<td>Forward-Compatible Feature-Driver Support Matrix</td>
<td>18</td>
</tr>
</tbody>
</table>
Chapter 1. Overview

The CUDA Toolkit is transitioning to a faster release cadence to deliver new features, performance improvements, and critical bug fixes.

However, the tight coupling of the CUDA runtime with the display driver (for example libcuda.so—the CUDA driver on Linux systems), means that users are required to update the entire driver stack to use the latest CUDA software (including the compiler, libraries, and tools), which can be disruptive in some scenarios.

Enhancing the compatibility of the CUDA platform is thus intended to address a few scenarios:

1. NVIDIA driver upgrades to systems with GPUs running in production for enterprises or datacenters can be complex and may need advance planning. Delays in rolling out new NVIDIA drivers could mean that users of such systems may not have access to new features available in CUDA releases. Not requiring driver updates for new CUDA releases can mean that new versions of the software can be made available faster to users.

2. Many software libraries and applications built on top of CUDA (such as math libraries or deep learning frameworks) do not have a direct dependency on the CUDA runtime, compiler or driver. In such cases, users or developers can still benefit from not having to upgrade the entire CUDA Toolkit or driver to use these libraries or frameworks.

3. Upgrading dependencies is error-prone and time consuming, and in some corner cases, can even change the semantics of a program. Constantly recompiling with the latest CUDA Toolkit means forcing upgrades on the end-customers of an application product. Package managers facilitate this process but unexpected issues can still arise and if a bug is found, it necessitates a repeat of the above upgrade process.

In this document, we introduce two key features of CUDA compatibility:

‣ First introduced in CUDA 10, the **CUDA Forward Compatible Upgrade** is designed to allow users to get access to new CUDA features and run applications built with new CUDA releases on systems with older installations of the NVIDIA datacenter GPU driver.

‣ First introduced in CUDA 11.1, **CUDA Enhanced Compatibility** provides two benefits:
  
  ▶ By leveraging semantic versioning across components in the CUDA Toolkit, an application can be built for one CUDA minor release (such as 11.1) and work across all future minor releases within the major family (such as 11.x).
  
  ▶ CUDA has relaxed the minimum driver version check and thus no longer requires a driver upgrade with minor releases of the CUDA Toolkit.

Before we introduce compatibility, it is important to review the various parts of the CUDA software and some concepts that will be referred to in this document.
1.1. CUDA

The CUDA software environment consists of three parts:

- **CUDA Toolkit (libraries, CUDA runtime and developer tools) - User-mode SDK used to build CUDA applications**
- **CUDA driver - User-mode driver component used to run CUDA applications (such as libcuda.so on Linux systems)**
- **NVIDIA GPU device driver - Kernel-mode driver component for NVIDIA GPUs**

On Linux systems, the CUDA driver and kernel mode components are delivered together in the NVIDIA display driver package. This is shown in Figure 1.

**Figure 1. Components of CUDA**

The CUDA Driver API ([CUDA Driver API Documentation](#)) is a programming interface for applications to target NVIDIA hardware. On top of that sits a runtime [cudart] with its own set of APIs, simplifying management of devices, kernel execution, and other aspects. The CUDA
compiler (nvcc), provides a way to handle CUDA and non-CUDA code (by splitting and steering compilation), and along with the CUDA runtime, is part of a CUDA compiler toolchain. Built on top of these technologies are CUDA libraries, some of which are included in the CUDA Toolkit, while others such as cuDNN may be released independently of the CUDA Toolkit.

1.2. Source Compatibility

We define source compatibility as a set of guarantees provided by the library, where a well-formed application built against a specific version of the library (using the SDK) will continue to build and run without errors when a newer version of the SDK is installed.

Both the CUDA driver and the CUDA runtime are not source compatible across the different SDK releases. APIs can be deprecated and removed, requiring changes to the application. Developers are notified through deprecation and documentation mechanisms of any current or upcoming changes. Although the driver APIs can change, they are versioned, and their symbols persist across releases to maintain binary compatibility.

1.3. Binary Compatibility

We define binary compatibility as a set of guarantees provided by the library, where an application targeting the said library will continue to work when dynamically linked against a different version of the library.

The CUDA Driver API has a versioned C-style ABI, which guarantees that applications that were running against an older driver (for example CUDA 3.2) will still run and function correctly against a modern driver (for example one shipped with CUDA 11.0). This is a stronger contract than an API guarantee - an application might need to change its source when recompiling against a newer SDK, but replacing the driver with a newer version will always work.

The CUDA Driver API thus is binary-compatible (the OS loader can pick up a newer version and the application continues to work) but not source-compatible (rebuilding your application against a newer SDK might require source changes). In addition, the binary-compatibility is in one direction: backwards.

Each version of the CUDA Toolkit (and runtime) requires a minimum version of the NVIDIA driver. The CUDA driver (libcudart.so on Linux for example) included in the NVIDIA driver package, provides binary backward compatibility. For example, an application built against the CUDA 3.2 SDK will continue to function even on today’s driver stack. On the other hand, the CUDA runtime has not provided either source or binary compatibility guarantees. Newer major and minor versions of the CUDA runtime have frequently changed the exported symbols, including their version or even their availability, and the dynamic form of the library has its shared object name (.SONAME in Linux-based systems) change every minor version.

If your application dynamically links against the CUDA 10.2 runtime, it will only work in the presence of the dynamic 10.2 CUDA runtime on the target system. If the runtime was statically linked into the application, it will function on a minimum supported driver, and any driver beyond. This concept is shown in Figure 2.
When an application built with CUDA 11.0 is run on a system with the R418 driver installed, CUDA initialization will return an error as can be seen in the example below.

In this example, the deviceQuery sample is compiled with CUDA 11.1 and is run on a system with R418. In this scenario, CUDA initialization returns an error due to the minimum driver requirement.

```
ubuntu@:~/samples/1_Utilsities/deviceQuery
$ make
/usr/local/cuda-11.1/bin/nvcc -ccbin g++ -I../../common/inc -m64 -gencode arch=compute_35,code=sm_35 -gencode arch=compute_37,code=sm_37 -gencode arch=compute_50,code=sm_50 -gencode arch=compute_52,code=sm_52 -gencode arch=compute_60,code=sm_60 -gencode arch=compute_61,code=sm_61 -gencode arch=compute_70,code=sm_70 -gencode arch=compute_75,code=sm_75 -gencode arch=compute_80,code=sm_80 -gencode arch=compute_86,code=sm_86 -gencode arch=compute_86,code=compute_86 -o deviceQuery.o -c deviceQuery.cpp
```

```
/usr/local/cuda-11.1/bin/nvcc -ccbin g++ -m64 -gencode arch=compute_35,code=sm_35 -gencode arch=compute_37,code=sm_37 -gencode arch=compute_50,code=sm_50 -gencode arch=compute_52,code=sm_52 -gencode arch=compute_60,code=sm_60 -gencode arch=compute_61,code=sm_61 -gencode arch=compute_70,code=sm_70 -gencode arch=compute_75,code=sm_75 -gencode arch=compute_80,code=sm_80 -gencode arch=compute_86,code=sm_86 -gencode arch=compute_86,code=compute_86 -o deviceQuery deviceQuery.0 -c deviceQuery.cpp
```
The table below summarizes the minimum driver required for a specific version of the CUDA runtime/toolkit. For convenience, today the NVIDIA driver can be installed as part of the CUDA Toolkit installation.

### Table 1. CUDA Toolkit and Compatible Driver Versions

<table>
<thead>
<tr>
<th>CUDA Toolkit</th>
<th>Linux x86_64 Driver Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA 11.2</td>
<td>&gt;= 460.80.02</td>
</tr>
<tr>
<td>CUDA 11.1 (11.1.0)</td>
<td>&gt;= 450.80.02</td>
</tr>
<tr>
<td>CUDA 11.0 (11.0.3)</td>
<td>&gt;= 450.36.06</td>
</tr>
<tr>
<td>CUDA 10.2 (10.2.89)</td>
<td>&gt;= 440.33</td>
</tr>
<tr>
<td>CUDA 10.1 (10.1.105)</td>
<td>&gt;= 418.39</td>
</tr>
<tr>
<td>CUDA 10.0 (10.0.130)</td>
<td>&gt;= 410.48</td>
</tr>
<tr>
<td>CUDA 9.2 [9.2.88]</td>
<td>&gt;= 396.26</td>
</tr>
<tr>
<td>CUDA 9.1 [9.1.85]</td>
<td>&gt;= 390.46</td>
</tr>
<tr>
<td>CUDA 9.0 [9.0.76]</td>
<td>&gt;= 384.81</td>
</tr>
<tr>
<td>CUDA 8.0 [8.0.61 GA2]</td>
<td>&gt;= 375.26</td>
</tr>
<tr>
<td>CUDA 8.0 [8.0.44]</td>
<td>&gt;= 367.48</td>
</tr>
<tr>
<td>CUDA 7.5 [7.5.16]</td>
<td>&gt;= 352.31</td>
</tr>
<tr>
<td>CUDA 7.0 [7.0.28]</td>
<td>&gt;= 346.46</td>
</tr>
</tbody>
</table>
1.4. **CUDA Binary (cubin) Compatibility**

A slightly related but important topic is one of application binary compatibility across GPU architectures in CUDA.

CUDA C++ provides a simple path for users familiar with the C++ programming language to easily write programs for execution by the device. Kernels can be written using the CUDA instruction set architecture, called PTX, which is described in the PTX reference manual. It is however usually more effective to use a high-level programming language such as C++. In both cases, kernels must be compiled into binary code by nvcc (called cubins) to execute on the device.

Cubins are architecture-specific. Binary compatibility for cubins is guaranteed from one compute capability minor revision to the next one, but not from one compute capability minor revision to the previous one or across major compute capability revisions. In other words, a cubin object generated for compute capability $X.y$ will only execute on devices of compute capability $X.z$ where $z \geq y$.

To execute code on devices of specific compute capability, an application must load binary or PTX code that is compatible with this compute capability. For portability, that is, to be able to execute code on future GPU architectures with higher compute capability (for which no binary code can be generated yet), an application must load PTX code that will be just-in-time compiled by the NVIDIA driver for these future devices.

More information on cubins, PTX and application compatibility can be found in the [CUDA C++ Programming Guide](#).
Chapter 2. CUDA Toolkit Versioning

Starting with CUDA 11, the toolkit versions are based on an industry-standard semantic versioning scheme: .X.Y.Z, where

- .X stands for the major version - APIs have changed and binary compatibility is broken.
- .Y stands for the minor version - Introduction of new APIs, deprecation of old APIs, and source compatibility might be broken but binary compatibility is maintained.
- .Z stands for the release/patch version - new updates and patches will increment this.

Each component in the toolkit is recommended to be semantically versioned, but you will find certain ones deliberately have slight deviations (such as NVRTC). We will note some of them later on in the document. The versions of the components in the toolkit are available in this [table](#).
Chapter 3. CUDA Enhanced Compatibility Across Minor Releases

By leveraging the semantic versioning starting with CUDA 11, components in the CUDA Toolkit will remain binary compatible across the minor versions of the toolkit. In order to maintain binary compatibility across minor versions, the CUDA runtime no longer bumps up the minimum driver version required for every minor release - this only happens when a major release is shipped. This feature is called CUDA Enhanced Compatibility.

One of the main reasons a new toolchain requires a new minimum driver is to handle the JIT compilation of PTX code and the JIT linking of binary code.

In this section, we will review the usage patterns that may require new user workflows when taking advantage of the enhanced compatibility features of the CUDA platform.

3.1. Existing CUDA Applications within Minor Versions

With CUDA enhanced compatibility, applications that were compiled with CUDA 11.1 can be run on the driver associated with CUDA 11.0 (i.e. R450).

Going back to our CUDA sample in the previous sections, we can see that the application compiled with CUDA 11.1 will work on the R450 CUDA 11.0 driver installations:

```
$ nvidia-smi
```

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.80.02    Driver Version: 450.80.02    CUDA Version: 11.0     |
+-------------------------------+----------------------+----------------------+
| GPU  Name Persistence-M| Bus-Id Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0 Tesla T4 On   | 00000000:00:1E.0 Off | 0 |
| N/A   39C    P8     9W / 70W |      0MiB / 15109MiB |      0%      Default |
| | | N/A |
+-------------------------------+----------------------+----------------------+

Processes:

```
<table>
<thead>
<tr>
<th>GPU</th>
<th>GI</th>
<th>CI</th>
<th>PID</th>
<th>Type</th>
<th>Process name</th>
<th>GPU Memory</th>
</tr>
</thead>
</table>
```

CUDA Compatibility
When our CUDA 11.1 application (i.e. cudart 11.1 is statically linked) is run on the system, we see that it runs successfully even when the driver reports a 11.0 version - i.e. without requiring the driver or other toolkit components to be updated on the system.

```
$ samples/bin/x86_64/linux/release/deviceQuery
samples/bin/x86_64/linux/release/deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)
Device 0: "Tesla T4"
CUDA Driver Version / Runtime Version 11.0 / 11.1
CUDA Capability Major/Minor version number: 7.5
...<snip>...

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.0, CUDA Runtime Version = 11.1, NumDevs = 1
Result = PASS
```

### 3.2. Handling New CUDA Features

When working with a feature exposed in a minor version of the toolkit, the feature might not be available when at runtime the application is running against an older CUDA driver. Users wishing to take advantage of such a feature, they should query its availability with a dynamic check in the code:

```
static bool hostRegisterFeatureSupported = false;
static bool hostRegisterIsDeviceAddress = false;

static error_t cuFooFunction(int *ptr)
{
    int *dptr = null;
    if (hostRegisterFeatureSupported) {
        cudaHostRegister(ptr, size, flags);
        if (hostRegisterIsDeviceAddress) {
            qptr = ptr;
        } else {
            cudaHostGetDevicePointer(&qptr, ptr, 0);
        }
    } else {
        // cudaMalloc();
        // cudaMemcpy();
    }

    gemm<<<1,1>>>(dptr);
    cudaMemcpy(
}

int main()
{
    // rest of code here
    cudaDeviceGetAttribute(
        &hostRegisterFeatureSupported,
        cudaDevAttrHostRegisterSupported, 0);
    cudaDeviceGetAttribute(
```
CUDA Enhanced Compatibility Across Minor Releases

Alternatively the application’s interface might not work at all without a new CUDA driver and then its best to return an error right away:

```c
#define MIN_VERSION 11010

cudaError_t foo()
{
  int version = 0;
  cudaGetDriverVersion(&version);
  if (version < MIN_VERSION) {
    return CUDA_ERROR_INSUFFICIENT_DRIVER;
  }
  // proceed as normal
}
```

A new error code is added to indicate that the functionality is missing from the driver you are running against: `cudaErrorCallRequiresNewerDriver`

### 3.3. Using PTX

PTX defines a virtual machine and ISA for general purpose parallel thread execution. PTX programs are translated at load time to the target hardware instruction set via the JIT Compiler which is part of the CUDA driver. As PTX is compiled by the CUDA driver, new toolchains will generate PTX that is not compatible with the older CUDA driver. This is not a problem when PTX is used for future device compatibility (the most common case), but can lead to issues when used for runtime compilation.

For codes continuing to make use of PTX, in order to support compiling on an older driver, your code must be first transformed into device code via the static ptxjitcompiler library or NVRTC with the option of generating code for a specific architecture (e.g. sm_80) rather than a virtual architecture (e.g. compute_80). For this workflow, a new nvptxcompiler_static library is shipped with the CUDA Toolkit.

We can see this usage in the following example:

```c
char* compilePTXToNVElf()
{
  nvPTXCompilerHandle compiler = NULL;
  nvPTXCompileResult status;

  size_t elfSize, infoSize, errorSize;
  char *elf, *infoLog, *errorLog;
  int minorVer, majorVer;

  const char* compile_options[] = { "--gpu-name=sm_80",
                                      "--device-debug"
                          };

  nvPTXCompilerGetVersion(&majorVer, &minorVer);
  nvPTXCompilerCreate(&compiler, (size_t)strlen(ptxCode), ptxCode);
  status = nvPTXCompilerCompile(compiler, 2, compile_options);
  if (status != NVPTXCOMPILE_SUCCESS) {
    nvPTXCompilerGetErrorLogSize(compiler, (void *)&errorSize);
    if (errorSize != 0) {
      errorLog = (char*)malloc(errorSize+1);
```
CUDA Enhanced Compatibility Across Minor Releases

3.4. Dynamic Code Generation

NVRTC is a runtime compilation library for CUDA C++. It accepts CUDA C++ source code in character string form and creates handles that can be used to obtain the PTX. The PTX string generated by NVRTC can be loaded by cuModuleLoadData and cuModuleLoadDataEx, and linked with other modules by cuLinkAddData of the CUDA Driver API.

As mentioned in the PTX section, the compilation of PTX to device code lives along with the CUDA driver, hence the generated PTX might be newer than what is supported by the driver on the deployment system. When using NVRTC, it is recommended that the resulting PTX code is first transformed to the final device code via the steps outlined by the PTX user workflow. This ensures your code is compatible. Alternatively, NVRTC can generate cubins directly starting with CUDA 11.1. Applications using the new API can load SASS directly using the driver API.

NVRTC used to support only virtual architectures through the option -arch, since it was only emitting PTX. It now supports actual architectures as well to emit SASS. The interface is augmented to retrieve either the PTX or cubin if an actual architecture is specified.

The example below shows how an existing example can be adapted to use the new features, guarded by the USE_CUBIN macro in this case:

```c
#include <nvrtc.h>
#include <cuda.h>
#include <iostream>

void NVRTC_SAFE_CALL(nvrtcResult result) {
    if (result != NVRTC_SUCCESS) {
        std::cerr << "nvrtc error: " << nvrtcGetErrorString(result) << '\n';
        std::exit(1);
    }
}

void CUDA_SAFE_CALL(CUresult result) {
    if (result != CUDA_SUCCESS) {
        const char *msg;
        cuGetErrorName(result, &msg);
        std::cerr << "cuda error: " << msg << '\n';
        std::exit(1);
    }
}
```
extern "C" __global__ void hello() {
    printf("hello world\n");
}

int main() {
    nvrtcProgram prog;
    nvrtcCreateProgram(&prog, hello, "hello.cu", 0, NULL, NULL);
    #ifdef USE_CUBIN
        const char *opts[] = {"-arch=sm_70"};
    #else
        const char *opts[] = {"-arch=compute_70"};
    #endif
    nvrtcCompileProgram(prog, 1, opts);
    size_t logSize;
    nvrtcGetProgramLog(prog, &logSize);
    char *log = new char[logSize];
    nvrtcGetProgramLog(prog, log);
    std::cout << log << '
';
    delete[] log;
    if (nvrtcResult != NVRTC_SUCCESS)
        exit(1);
    size_t codeSize;
    #ifdef USE_CUBIN
        nvrtcGetCUBINSize(prog, &codeSize);
        char *code = new char[codeSize];
        nvrtcGetCUBIN(prog, code);
    #else
        nvrtcGetPTXSize(prog, &codeSize);
        char *code = new char[codeSize];
        nvrtcGetPTX(prog, code);
    #endif
    nvrtcDestroyProgram(&prog);
    CUdevice cuDevice;
    CUcontext context;
    CUmodule module;
    CUfunction kernel;
    CUDA_SAFE_CALL(cuInit(0));
    CUDA_SAFE_CALL(cuDeviceGet(&cuDevice, 0));
    CUDA_SAFE_CALL(cuCtxCreate(&context, 0, cuDevice));
    CUDA_SAFE_CALL(cuModuleLoadDataEx(&module, code, 0, 0, 0));
    CUDA_SAFE_CALL(cuModuleGetFunction(&kernel, module, "hello");
    CUDA_SAFE_CALL(cuLaunchKernel(kernel, 1, 1, 1, 1, 1, 1, 0, NULL, NULL, 0));
    CUDA_SAFE_CALL(cuCtxSynchronize());
    CUDA_SAFE_CALL(cuModuleUnload(module));
    CUDA_SAFE_CALL(cuCtxDestroy(context));
    delete[] code;
}
Chapter 4. Forward- Compatible Upgrade Path

To meet the minimum requirements mentioned in Section 1.3, the upgrade path for CUDA usually involves upgrades to both the CUDA Toolkit and NVIDIA driver as shown in the figure below.

Figure 2. CUDA Upgrade Path

Starting with CUDA 10.0, NVIDIA introduced a new forward-compatible upgrade path that allows the kernel mode components on the system to remain untouched, while the CUDA
driver is upgraded. See Figure 3. This allows the use of newer toolkits on existing system installations, providing improvements and features of the latest CUDA while minimizing the risks associated with new driver deployments. This upgrade path is achieved through new packages provided by CUDA.

**Figure 3. Forward Compatibility Upgrade Path**

*Upgrade only CUDA user-mode components*

**4.1. Installing Packages to Enable Forward Compatible Upgrade**

The compatible upgrade files are meant as additions to the existing system installation and not replacements for those files. For ease of deployment, a new package is available in the local installers or the CUDA network repositories provided by NVIDIA:

- `cuda-compat-11.1`

This package provides the CUDA Compatibility Platform files necessary to run 11.1 CUDA APIs. The package can be installed using Linux package managers such as `apt` or `yum`. For
example, on an Ubuntu 16.04 system, run the following command when using the `apt` package manager:

```bash
$ sudo apt-get install -y cuda-compat-11-1
```

The package is installed to the versioned toolkit location typically found in the `/usr/local/cuda-11.1/` directory (or replace 11.1 with the appropriate version). The package consists of:

- `libcuda.so.*` - the CUDA Driver
- `libnvidia-ptxjitcompiler.so.*` - the JIT (just-in-time) compiler for PTX files

**Note:**

This package only provides the files, and does not configure the system. These files should be kept together as the CUDA driver depends on the PTX JIT loader (`libnvidia-ptxjitcompiler`) that is of the same version.

These files can also be extracted from the appropriate datacenter driver `runfile` installers (.run) available in NVIDIA driver downloads. To do this:

1. Download the latest NVIDIA datacenter (Tesla) driver, and extract the .run file using option `-x`.
2. Copy the three CUDA Compatibility Platform files, listed at the start of this section, into a user- or root-created directory.
3. Then follow your system’s guidelines for making sure that the system linker picks up the new libraries.

Going back to the previous, the deviceQuery sample was compiled with CUDA 11.1. The CUDA Compatibility is installed and the application can now run successfully as shown below. In this example, the user sets `LD_LIBRARY_PATH` to include the files installed by the cuda-compat-11-1 package.

```bash
$ sudo apt-get install -y cuda-compat-11-1
```

Checking the files installed under the `/usr/local/cuda/compat`:

```
$ ls -l /usr/local/cuda/compat
total 30808
lrwxrwxrwx 1 root root 12 Sep 19 01:50 libcuda.so -> libcuda.so.1
lrwxrwxrwx 1 root root 20 Sep 19 01:50 libcuda.so.1 -> libcuda.so.455.23.05
-rw-r--r-- 1 root root 21066104 Sep 18 19:49 libcuda.so.455.23.05
lrwxrwxrwx 1 root root 37 Sep 19 01:50 libnvidia-ptxjitcompiler.so.1 -> libnvidia-ptxjitcompiler.so.455.23.05
-rw-r--r-- 1 root root 10475688 Sep 18 19:35 libnvidia-ptxjitcompiler.so.455.23.05
```

The user can set `LD_LIBRARY_PATH` to include the files installed before running the CUDA 11.1 application:

```
$ LD_LIBRARY_PATH=/usr/local/cuda/compat:$LD_LIBRARY_PATH samples/bin/x86_64/linux/release/deviceQuery
```

```
CUDA Device Query (Runtime API) version (CUDART static linking)
```
Detected 1 CUDA Capable device(s)

Device 0: "Tesla T4"
  CUDA Driver Version / Runtime Version       11.1 / 11.1
  CUDA Capability Major/Minor version number: 7.5

...<snip>...

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.1, CUDA Runtime Version = 11.1, NumDevs = 1
Result = PASS
Chapter 5. Support

5.1. Hardware Support

The current hardware support is shown in Table 2.

Table 2. Compute Capability Support

<table>
<thead>
<tr>
<th>Hardware Generation</th>
<th>Compute Capability</th>
<th>Driver 384.111+</th>
<th>Driver 410.48+</th>
<th>Driver 418.40.04+</th>
<th>Driver 440.33.01+</th>
<th>Driver 450.36.06+</th>
<th>Driver 460.27.04+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ampere</td>
<td>8.0</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Turing</td>
<td>7.5</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Volta</td>
<td>7.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Pascal</td>
<td>6.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Maxwell</td>
<td>5.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Kepler</td>
<td>3.x</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fermi</td>
<td>2.x</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

5.2. Supported GPUs

The CUDA compatible upgrade is meant to ease the management of large production systems for enterprise customers. As such, the supported HW (hardware) for this new upgrade path is limited to NVIDIA Datacenter [Tesla] GPU products. It’s important to note that HW support is defined by the kernel mode driver and as such, newer CUDA drivers on their own will not enable new HW support. Refer to Hardware Support for which hardware is supported by your system.

5.3. CUDA Application Compatibility

With CUDA compatible upgrade, applications built with newer CUDA toolkits can be supported on NVIDIA datacenter driver branches. The Table 3 below shows the support matrix when using the CUDA compatible upgrade.
The entries in the table below indicate whether or not CUDA compatible upgrade is supported.

**Table 3. CUDA Application Compatibility Support Matrix**

<table>
<thead>
<tr>
<th>CUDA Forward Compatible Upgrade</th>
<th>NVIDIA Kernel Mode Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>418.40.04+ (CUDA 10.1)</td>
</tr>
<tr>
<td>11-2</td>
<td>Compatible</td>
</tr>
<tr>
<td>11-1</td>
<td>Compatible</td>
</tr>
<tr>
<td>11-0</td>
<td>Compatible</td>
</tr>
<tr>
<td>10-2</td>
<td>Compatible</td>
</tr>
<tr>
<td>10-1</td>
<td>Compatible</td>
</tr>
<tr>
<td>10-0</td>
<td>Not Compatible</td>
</tr>
</tbody>
</table>

5.4. Feature Support

There are specific features in the CUDA driver that require kernel-mode support and will only work with a newer kernel mode driver. A few features depend on other user-mode components and are therefore also unsupported. See Table 4.

**Table 4. Forward-Compatible Feature-Driver Support Matrix**

<table>
<thead>
<tr>
<th>CUDA Forward Compatible Upgrade</th>
<th>CUDA – OpenGL/Vulkan Interop</th>
<th>POWER9 ATS</th>
<th>cuMemMap* set of functionalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Base Installation: 450 (&gt;=.02) Driver</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11-1</td>
<td>No</td>
<td>Yes [2]</td>
<td>Yes [1]</td>
</tr>
<tr>
<td>11-0</td>
<td>No</td>
<td>Yes [2]</td>
<td>Yes [1]</td>
</tr>
</tbody>
</table>

| System Base Installation: 418 (>=.XX) Driver |
| 11-1                            | No                           | Yes [2]    | Yes [1]                         |
| 11-0                            | No                           | Yes [2]    | Yes [1]                         |
| 10-2                            | No                           | No         | Yes [1]                         |
[1] This relies on 
CU_DEVICE_ATTRIBUTE_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR_SUPPORTED and 
CU_DEVICE_ATTRIBUTE_VIRTUAL_ADDRESS_MANAGEMENT_SUPPORTED, which should be 
queried if you intend to use the full range of this functionality.

[2] Supported on Red Hat Enterprise Linux operating system version 8.1 or higher.

In addition to the CUDA driver and certain compiler components, there are other drivers 
in the system installation stack (e.g. OpenCL) that remain on the old version. The forward-
compatible upgrade path is for CUDA only.
Chapter 6. Deployment Considerations

Consider a cluster of 500+ multi-GPU servers running bare-metal in support of 50-1500 users, running a variety of DL and HPC workloads. This system is scheduled in a classical manner (for example, using Slurm or LSF) with resources being allocated within a cgroup, sometimes in exclusive mode. The cluster can be homogeneous with respect to node configurations, or heterogeneous, including some nodes of each of a few generations of NVIDIA datacenter [Tesla] GPUs - V100 16GB or 32GB, P100, P40, K80, and/or K40.

With the introduction of CUDA compatible upgrade, [regardless of how the files are packaged - run/deb/rpm/etc.], the site operator must pay close attention to where these libraries are placed. It could potentially be part of the disk image [i.e., more closely tied to the kernel and driver installations], or part of one of the modules [i.e., more closely tied to the toolkit and the libraries].

6.1. Disk Image

In this case the compatibility files are located somewhere on the boot image alongside the existing system files. The exact path is not important, but the files should remain together, and be resolvable by the dynamic loader. This could be accomplished by having the system’s loader automatically pick them up [e.g. ld.so.conf], through the use of RPATH, or through a more manual process of documentation [users of the new toolkit must manually set LD_LIBRARY_PATH].

6.2. Modules System

It is common for the users to request any of the several CUDA Toolkit versions in the same way they might request any of several versions of numerous other system libraries or compiler toolchains.

A common mechanism to enable this is the Modules System, wherein the admin, or the user, sets up ‘module’ scripts for each version of each package, and then the user can execute a command like ‘module load cuda/11.0’.

Often the loading of various module versions will be scripted with the application such that each application picks up exactly the versions of its dependencies that it needs, even if other versions would have been available for other applications to choose from.
If the components from the CUDA compatible upgrade are placed such that they are chosen by the module load system, it is important to note the limitations of this new path – namely, only certain major versions of the system driver stack, only NVIDIA datacenter (Tesla) GPUs are supported, and only in a forward compatible manner (i.e. an older libcuda.so will not work on newer base systems).

It is therefore recommended that the module load script be aware of these limitations, and proactively query the system for whether the compatibility platform can be used. In the cases where it cannot use the CUDA compatible upgrade, a fallback path to the default system’s installed CUDA driver can provide a more consistent experience. See Miscellaneous for performing this task with RPATH.

6.3. Miscellaneous

After the system is fully upgraded (the display driver and the CUDA driver) to a newer base installation, the CUDA compatible upgrade files should be removed as they are no longer necessary and will not function.

A common way of ensuring that the new CUDA compatible upgrade files are picked up, is by using RPATH. This ensures that during the compilation process of the application, the runtime search path is hard-coded into the executable.

For example, the RPATH can be set to /usr/compat and this setting is used throughout the cluster. Each individual system then configures /usr/compat to the appropriate directory – if forward compatibility is desired/allowed, this can contain the (latest) CUDA compatible upgrade files.

In order to workaround the lack of automatic fallback when the system driver leapfrogs the CUDA compatible upgrade files, or in case of a non-supported HW configuration, the /usr/compat can point to the NVIDIA datacenter driver for those systems instead. This way a single, consistent, path is used throughout the entire cluster.
Chapter 7. Frequently Asked Questions

This section includes some FAQs related to CUDA compatibility.

1. **What is the difference between CUDA forward compatible upgrade and CUDA enhanced compatibility? When should users use these features?**

<table>
<thead>
<tr>
<th>Area</th>
<th>CUDA Forward Compatible Upgrade</th>
<th>CUDA Enhanced Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUs supported</td>
<td>NVIDIA datacenter GPUs only (previously “Tesla”)</td>
<td>All GPU products supported</td>
</tr>
<tr>
<td>OS distributions supported</td>
<td>Linux only</td>
<td>Windows, Linux</td>
</tr>
<tr>
<td>Mobile platforms supported</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Features Supported</td>
<td>Some features such as [CUDA-GL interop, Power 9 ATS, cuMemMap APIs] are not supported. These features depend on a new kernel mode driver and thus are not supported. These are explicitly called out in the documentation.</td>
<td>All existing CUDA features (from older minor releases) work. Users may have to incorporate checks in their application when using new features in the minor release (that require a new driver) to ensure graceful errors. More information can be found in the “Handling New CUDA Features” section.</td>
</tr>
<tr>
<td>CUDA Releases Supported</td>
<td>All CUDA releases supported through the lifetime of the datacenter driver branch. For example, R418 (CUDA 10.1) EOLs in March 2022 - so all CUDA versions released (including major releases) during this timeframe are supported.</td>
<td>Only works within a ‘major’ release family (e.g. 11.0 through 11.x). Compatibility is not supported across major CUDA releases.</td>
</tr>
<tr>
<td>Application includes PTX or uses NVRTC</td>
<td>No additional workflow required.</td>
<td>Users should use the new PTX static library to rebuild binaries. Refer to the workflow section for more details.</td>
</tr>
<tr>
<td>Requires administrator involvement</td>
<td>Depends on the deployment. Users can also set up $LD_LIBRARY_PATH$ with the</td>
<td>Not required.</td>
</tr>
</tbody>
</table>
### FAQ: CUDA Compatibility

<table>
<thead>
<tr>
<th>Area</th>
<th>CUDA Forward Compatible Upgrade</th>
<th>CUDA Enhanced Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>new libraries from the cuda-compat-* package.</td>
<td></td>
</tr>
</tbody>
</table>

2. **If we build our CUDA application using CUDA 11.0, can it continue to be used with newer NVIDIA drivers (e.g. CUDA 11.1/R455, 11.x etc.)? Or is it only the other way around?**

Drivers have always been backwards compatible with CUDA. This means that a CUDA 11.0 application will be compatible with R450 (11.0), R455 (11.1) and beyond. CUDA applications typically statically include all the libraries (for example cudart, CUDA math libraries such as cuBLAS, cuFFT) they need, so they should work on new drivers or CUDA Toolkit installations.

In other words, since CUDA is backward compatible, existing CUDA applications can continue to be used with newer CUDA versions.

3. **What is the minimum CUDA 11.0 driver that supports the CUDA enhanced compatibility?**

The minimum driver version required is 450.80.02.

4. **With CUDA enhanced compatibility, can components from CUDA 11.1 be mixed with CUDA 11.x installations? E.g. if an application dynamically links against cuBLAS from 11.1, can the new cuBLAS be dropped into a 11.0 installation or 11.x installation?**

- Toolkit libraries are independently and semantically versioned. With the exception of NVRTC, the SONAME for all the other libraries will remain the same across 11.x releases. If you build an application with CUDA 11.1 and dynamically link with some 11.1 libraries and want it to work on a 11.0 installation, those 11.1 libraries might have some symbols that may not present in a 11.0 installation. In this case, you should ensure that the correct libraries are present on the system - this can be done through package managers for example. If your application package has the right dependencies specified, then package managers on Linux will download the right 11.1 libraries to satisfy your application’s dependencies.

- If your application was built with CUDA 11.1 and in future is used in a 11.1+ installation, then the application should continue to work.

5. **What about new features introduced in minor releases of CUDA? How does a developer build an application using newer CUDA Toolkits (e.g. 11.x) work on a system with a CUDA 11.0 driver (R450)?**

By using new CUDA versions, users can benefit from new CUDA programming model APIs, compiler optimizations and math library features. There are some caveats:

- A subset of CUDA APIs don’t need a new driver and they can all be used without any driver dependencies. For example, `async copy` APIs introduced in 11.1 do not need a new driver.

- To use other CUDA APIs introduced in a minor release (that require a new driver), one would have to implement fallbacks or fail gracefully. This situation is not different from what is available today where developers use macros to compile out features based on CUDA versions. Users should refer to the CUDA headers and documentation for new CUDA APIs introduced in a release.
6. **Does CUDA compatibility work with containers?**

   Yes. CUDA enhanced compatibility and CUDA forward compatible upgrade both work when using either NGC Deep Learning Framework containers or using containers that are based on the official CUDA base images. The images include the CUDA compatible upgrade libraries and the NVIDIA Container Toolkit [nvidia-docker2](https://docs.nvidia.com/nvidia-docker2) has logic to correctly load the required libraries.

7. **I’m running an NGC container and see this error: “This container was built for NVIDIA Driver Release 450.51 or later, but version 418.126.02 was detected and compatibility mode is UNAVAILABLE.”. What could be wrong?**

   It is possible you are either running a wrong version of the NVIDIA driver on the system or your system does not have an NVIDIA datacenter (Tesla) GPU.

8. **What are the errors returned by CUDA that an application can use to determine if CUDA Forward Compatible Upgrade is supported?**

   - Two error codes have been added:
     - **CUDA_ERROR_SYSTEM_DRIVER_MISMATCH = 803.** This error indicates that there is a mismatch between the versions of the display driver and the CUDA driver.
     - **CUDA_ERROR_COMPAT_NOT_SUPPORTED_ON_DEVICE = 804.** This error indicates that the system was upgraded to run with forward compatibility but the visible hardware detected by CUDA does not support this configuration.
   
   - More information can be found in the CUDA driver API reference: [https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__TYPES.html](https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__TYPES.html)

9. **Does CUDA forward compatible upgrade work intra-branch?**

   This is the scenario where the user is running an NVIDIA kernel mode driver and CUDA user mode driver (from the cuda-compat-* package) from the same driver branch. This use-case is supported only for NVIDIA datacenter driver branches. Refer to the documentation on the supported datacenter drivers.
Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND 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. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

OpenCL

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

Trademarks

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

Copyright

© 2007-2021 NVIDIA Corporation. All rights reserved.