Getting started with CUDA
Part 4 - Compilation and Runtime

Edwin Carlinet, Joseph Chazalon {firstname.lastname@epita.fr}

Fall 2023

EPITA Research Laboratory (LRE)

Slides generated on September 8, 2023

Compilation and Runtime

Compilation simplified overview

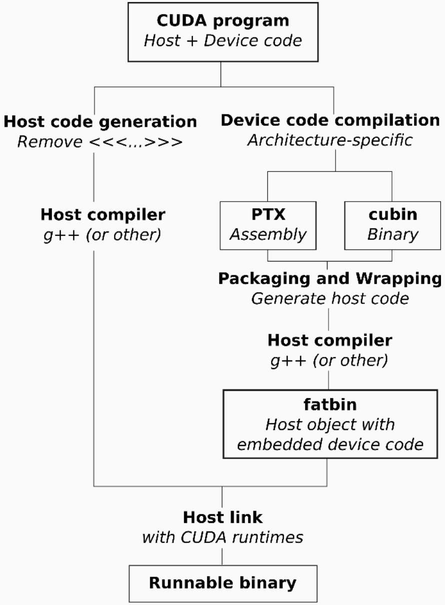

Figure 1: Separate compilation of host and device code

Host and devices code follow two different compilation trajectories.

Device code is compiled into two formats:

- **PTX assembly** tied to a *virtual architecture* specification
- **cubin binary code** tied to a particular GPU product family — aka *real architecture* like *Fermi*, *Kepler*, *Maxwell*, *Pascal*, *Volta*, *Turing* and *Ampere*

The final runnable binary

- contains both host and device code
- is linked with the CUDA runtime(s).

Runtime

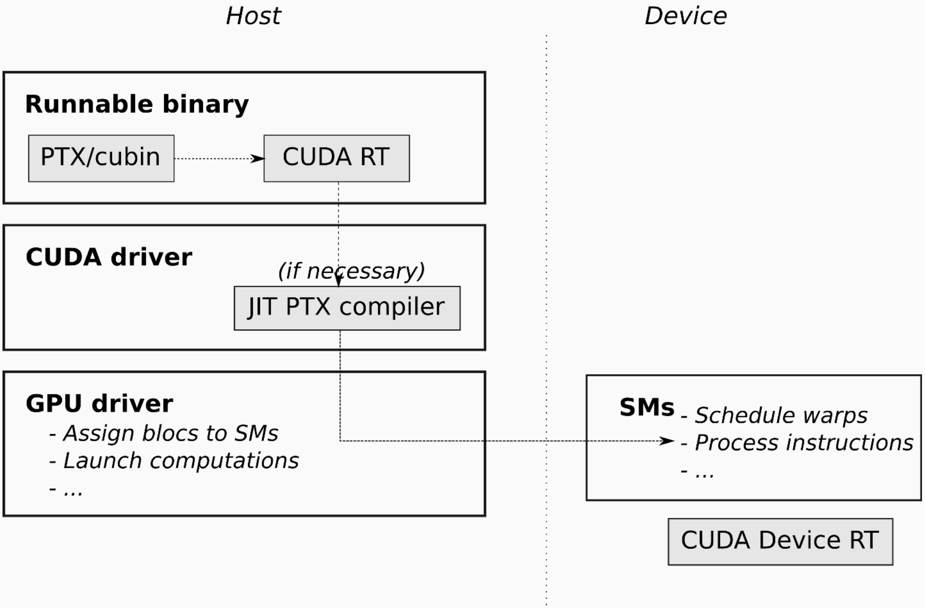

Figure 2: Transfer of code to device with optional JIT compilation

Two-stage compilation

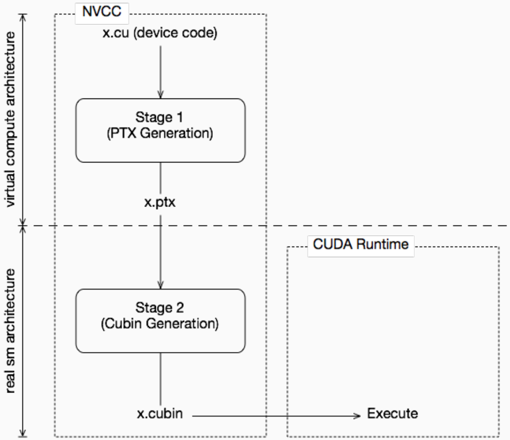

Figure 3: Two-Staged (offline) Compilation with Virtual and Real Architectures

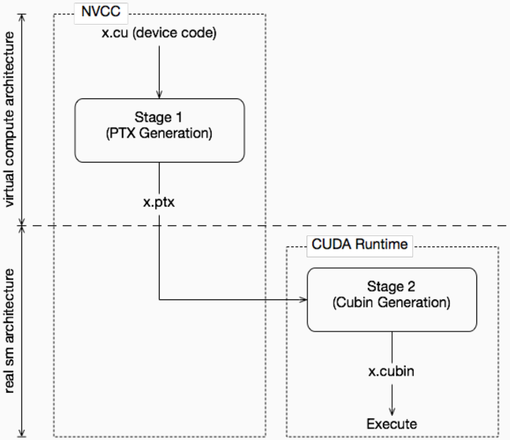

Figure 4: Just-in-Time Compilation of Device Code

PTX, cubin, fatbinary... Why?

Because NVidia wants to be able to push innovations on their hardware as soon as possible, they **do not ensure forward compatibility of binaries**, unlike CPU vendors.

They break forward compatibility at each major GPU release, ie when they release a new GPU family.

Binary code compatibility (cubin)

GPU (device) binary code is **not forward (nor backward) compatible**:
it is architecture-specific and can be run only by hardware with the same major version.

Example:
Binary code compiled and optimized for sm_30 cards

- can be run by sm_32 and sm_35 cards (Kepler family),
- but cannot be run by sm_5x cards (Maxwell family).

PTX code compatibility

Assembly code, however, is based on an always-increasing set of instructions (much like SSE extensions).
This implies two things:

- **PTX assembly is forward compatible with newer architectures**,
- it is **not backward compatible** though,
- it is always possible to compile
  the PTX assembly of an earlier version (like `compute_30`)
  to a binary for the most recent architecture (like `sm_75`).

This is how NVidia ensures that old code will still run on newer hardware.
New code, however, will not run on old hardware unless special care is taken (more on that later).

CUDA Driver and PTX compilation

The **CUDA driver** (`libcuda.so`) contains the **JIT PTX compiler** and is **always backward compatible** (this is what actually makes PTX forward compatible).

This means that it can take assembly code from an older version anc compile it for the current version of the device on the current machine.

However, it is **not forward compatible**: code compiled with newer PTX assembly cannot be understood.

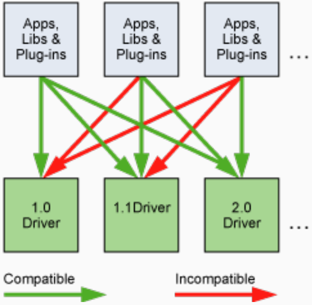

Figure 5: Compatibility of CUDA Versions

It may be necessary to ask the user to install a newer version of the CUDA driver on its system, or to add some compatibility code for older architectures / CUDA drivers.

As of Nov. 2019, what is safe to use?

Maximum compatibility

```
/usr/local/cuda/bin/nvcc
  -gencode=arch=compute_30,code=sm_30
  -gencode=arch=compute_35,code=sm_35
  -gencode=arch=compute_50,code=sm_50
  -gencode=arch=compute_60,code=sm_60
  -gencode=arch=compute_70,code=sm_70
  -gencode=arch=compute_75,code=sm_75
  -gencode=arch=compute_75,code=compute_75
  -O2 -o mykernel.o -c mykernel.cu
```

Distribute the cudart lib (static or dynamic link) with your application.

`__CUDA_ARCH__`

Use different code paths to support previous architectures.

```
__device__ func()
{
#if __CUDA_ARCH__ < 350
  /* Do something special for
  architectures without dynamic
  parallelism. */
#else
  /* Do something else. */
#endif
}
```

Deprecations

Kepler and Maxwell hardware are being deprecated (sm_3x, sm_5x).
2022 update: sm_3x, sm_5x and sm_6x **ARE** deprecated now.

Compilation and Runtime Summary

Host code and device code are compiled separately.

- Device code is packaged with host code to be launched.
- A host compiler (ex g++) is required.

You can select which features you want to activate in your code,
hence which compatibility you offer.

- Using `__CUDA_ARCH__` macro in your code to support multiple architectures.
- Using `nvcc`'s `-arch compute_xx` flag.
- This controls the PTX assembly which is generated.
- PTX assembly is forward compatible thanks to JIT compilation.

You can select the hardware you want to build a precompiled binary (cubin) for.

- Accelerates application startup (do not care about it for now).
- Using `nvcc`'s `-code sm_xx` flag.

You can generate multiples PTX and cubins using the following `nvcc`'s flags repeatidly:
`-gencode arch=compute_xx,code=sm_yy`

More details

Real architectures vs Virtual architectures

Real architectures

Hardware version	Features
sm_30 and sm_32	Basic features + Kepler support + Unified memory programming
sm_35	+ Dynamic parallelism support
sm_50, sm_52 and sm_53	+ Maxwell support
sm_60, sm_61 and sm_62	+ Pascal support
sm_70 and sm_72	+ Volta support
sm_75	+ Turing support

Virtual architectures

Compute capability	Features
compute_30 and compute_32	Basic features + Kepler support + Unified memory programming
compute_35	+ Dynamic parallelism support
compute_50, compute_52, and compute_53	+ Maxwell support
compute_60, compute_61, and compute_62	+ Pascal support
compute_70 and compute_72	+ Volta support
compute_75	+ Turing support

Real architectures ("code")

- Run compiled binary code (cubin)
- Instantiate a virtual architecture to a particular number of SMs per GPU
- Specifies a particular SM model
- Noted `sm_xx`
- Selected using the `-code` parameter of `nvcc`

What's the point?

- Pre-compile your kernels for a particular hardware and accelerate program startup.

Virtual architectures ("arch")

- Specifies an instruction set for PTX assembly (ptx) (much like SSE extensions)
- Specifies features available
- Noted `compute_xx`
- Selected using the `-arch` parameter of `nvcc`

What's the point?

- Limit the features you want to use to maximize compatibility
- Migrate code progressively as some behavior may change (like Independent Thread Scheduling in `compute_70`)
- The `__CUDA_ARCH__` macro will be set accordingly in your code so you can have different code paths for different compute capabilities

More on compute capabilities

Excellent summaries:

- Appendix H on Compute Capabilities of CUDA C programming guide
- CUDA page on Wikipedia
- List of GPUs and their compute capability version available here: developer.nvidia.com/cuda-gpus

<table>
<tr><th>Feature Support</th><th colspan="6">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>3.0</th><th>3.2</th><th>3.5, 3.7, 5.0, 5.2</th><th>5.3</th><th>6.x</th><th>7.x</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td><td colspan="6">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td><td colspan="6">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())</td><td colspan="4">No</td><td colspan="2">Yes</td></tr>
<tr><td>Warp vote and ballot functions (Warp Vote Functions)</td><td colspan="6" rowspan="6">Yes</td></tr>
<tr><td>__threadfence_system() (Memory Fence Functions)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Synchronization Functions)</td></tr>
<tr><td>Surface functions (Surface Functions)</td></tr>
<tr><td>3D grid of thread blocks</td></tr>
<tr><td>Unified Memory Programming</td></tr>
<tr><td>Funnel shift (see reference manual)</td><td>No</td><td colspan="5">Yes</td></tr>
<tr><td>Dynamic Parallelism</td><td colspan="2">No</td><td colspan="4">Yes</td></tr>
<tr><td>Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion</td><td colspan="3">No</td><td colspan="3">Yes</td></tr>
<tr><td>Tensor Core</td><td colspan="5">No</td><td>Yes</td></tr>
</table>

Figure 6: Feature Support per Compute Capability

<table>
<tr><th></th><th colspan="12">Compute Capability</th></tr>
<tr><th>Technical Specifications</th><th>3.0</th><th>3.2</th><th>3.5</th><th>3.7</th><th>5.0</th><th>5.2</th><th>5.3</th><th>6.0</th><th>6.1</th><th>6.2</th><th>7.0</th><th>7.5</th></tr>
<tr><td>Maximum number of resident grids per device (Concurrent Kernel Execution)</td><td>16</td><td>4</td><td colspan="4">32</td><td>16</td><td>128</td><td>32</td><td>16</td><td colspan="2">128</td></tr>
<tr><td>Maximum dimensionality of grid of thread blocks</td><td colspan="12">3</td></tr>
<tr><td>Maximum x-dimension of a grid of thread blocks</td><td colspan="12">2^{31}-1</td></tr>
<tr><td>Maximum y- or z-dimension of a grid of thread blocks</td><td colspan="12">65535</td></tr>
<tr><td>Maximum dimensionality of thread block</td><td colspan="12">3</td></tr>
<tr><td>Maximum x- or y-dimension of a block</td><td colspan="12">1024</td></tr>
<tr><td>Maximum z-dimension of a block</td><td colspan="12">64</td></tr>
<tr><td>Maximum number of threads per block</td><td colspan="12">1024</td></tr>
<tr><td>Warp size</td><td colspan="12">32</td></tr>
<tr><td>Maximum number of resident blocks per multiprocessor</td><td colspan="4">16</td><td colspan="7">32</td><td>16</td></tr>
<tr><td>Maximum number of resident warps per multiprocessor</td><td colspan="11">64</td><td>32</td></tr>
<tr><td>Maximum number of resident threads per multiprocessor</td><td colspan="11">2048</td><td>1024</td></tr>
<tr><td>Number of 32-bit registers per multiprocessor</td><td colspan="3">64 K</td><td>128 K</td><td colspan="8">64 K</td></tr>
<tr><td>Maximum number of 32-bit registers per thread block</td><td>64 K</td><td>32 K</td><td colspan="4">64 K</td><td>32 K</td><td colspan="2">64 K</td><td>32 K</td><td colspan="2">64 K</td></tr>
<tr><td>Maximum number of 32-bit registers per thread</td><td>63</td><td colspan="11">255</td></tr>
<tr><td>Maximum amount of shared memory per multiprocessor</td><td colspan="3">48 KB</td><td>112 KB</td><td>64 KB</td><td>96 KB</td><td colspan="2">64 KB</td><td>96 KB</td><td>64 KB</td><td>96 KB</td><td>64 KB</td></tr>
<tr><td>Maximum amount of shared memory per thread block [26]</td><td colspan="10">48 KB</td><td>96 KB</td><td>64 KB</td></tr>
<tr><td>Number of shared memory banks</td><td colspan="12">32</td></tr>
<tr><td>Amount of local memory per thread</td><td colspan="12">512 KB</td></tr>
<tr><td>Constant memory size</td><td colspan="12">64 KB</td></tr>
<tr><td>Cache working set per multiprocessor for constant memory</td><td colspan="7">8 KB</td><td>4 KB</td><td colspan="4">8 KB</td></tr>
<tr><td>Cache working set per multiprocessor for texture memory</td><td colspan="7">Between 12 KB and 48 KB</td><td colspan="3">Between 24 KB and 48 KB</td><td>32 ~ 128 KB</td><td>32 or 64 KB</td></tr>
<tr><td>Maximum width for a 1D texture reference bound to a CUDA array</td><td colspan="12">65536</td></tr>
<tr><td>Maximum width for a 1D texture reference bound to linear memory</td><td colspan="12">2^{27}</td></tr>
<tr><td>Maximum width and number of layers for a 1D layered texture reference</td><td colspan="12">16384 x 2048</td></tr>
<tr><td>Maximum width and height for a 2D texture reference bound to a CUDA array</td><td colspan="12">65536 x 65535</td></tr>
<tr><td>Maximum width and height for a 2D texture reference bound to linear memory</td><td colspan="12">65000 x 65000</td></tr>
<tr><td>Maximum width and height for a 2D texture reference bound to a CUDA array supporting texture gather</td><td colspan="12">16384 x 16384</td></tr>
<tr><td>Maximum width, height, and number of layers for a 2D layered texture reference</td><td colspan="12">16384 x 16384 x 2048</td></tr>
<tr><td>Maximum width, height, and depth for a 3D texture reference bound to a CUDA array</td><td colspan="12">4096 x 4096 x 4096</td></tr>
<tr><td>Maximum width (and height) for a cubemap texture reference</td><td colspan="12">16384</td></tr>
<tr><td>Maximum width (and height) and number of layers for a cubemap layered texture reference</td><td colspan="12">16384 x 2046</td></tr>
<tr><td>Maximum number of textures that can be bound to a kernel</td><td colspan="12">256</td></tr>
<tr><td>Maximum width for a 1D surface reference bound to a CUDA array</td><td colspan="12">65536</td></tr>
<tr><td>Maximum width and number of layers for a 1D layered surface reference</td><td colspan="12">65536 x 2048</td></tr>
<tr><td>Maximum width and height for a 2D surface reference bound to a CUDA array</td><td colspan="12">65536 x 32768</td></tr>
<tr><td>Maximum width, height, and number of layers for a 2D layered surface reference</td><td colspan="12">65536 x 32768 x 2048</td></tr>
<tr><td>Maximum width, height, and depth for a 3D surface reference bound to a CUDA array</td><td colspan="12">65536 x 32768 x 2048</td></tr>
<tr><td>Maximum width (and height) for a cubemap surface reference bound to a CUDA array</td><td colspan="12">32768</td></tr>
<tr><td>Maximum width (and height) and number of layers for a cubemap layered surface reference</td><td colspan="12">32768 x 2046</td></tr>
<tr><td>Maximum number of surfaces that can be bound to a kernel</td><td colspan="12">16</td></tr>
<tr><td>Maximum number of instructions per kernel</td><td colspan="12">512 million</td></tr>
</table>

Figure 7: Technical Specifications per Compute Capability

Documentation excerpt

Compute Capability 3.x:

- **Architecture**

  A multiprocessor consists of:
  - 192 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
  - 32 special function units for single-precision floating-point transcendental functions,
  - 4 warp schedulers.
- **Global Memory**
  - Global memory accesses for devices of compute capability 3.x are cached in L2...
  - A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory...
- **Shared Memory**
  - Shared memory has 32 banks...

Compute Capability 5.x:

- **Architecture**

  A multiprocessor consists of:
  - 128 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
  - 32 special function units for single-precision floating-point transcendental functions,
  - 4 warp schedulers.
- ...

CUDA Runtime and SDK support

The **CUDA runtime** (`libcudart.so`) is bundled with your SDK an provides high-level functionnality.

- You should distribute the CUDA runtime with your application.
- It is compatible with a certain range of GPU driver versions.
- It supports a certain range of hardware (GPU families):
  - ...
  - CUDA SDK 6.5 support for compute capability 1.1 - 5.x (Tesla, Fermi, Kepler, Maxwell). Last version with support for compute capability 1.x (Tesla)
  - CUDA SDK 7.0 - 7.5 support for compute capability 2.0 - 5.x (Fermi, Kepler, Maxwell)
  - CUDA SDK 8.0 support for compute capability 2.0 - 6.x (Fermi, Kepler, Maxwell, Pascal). Last version with support for compute capability 2.x (Fermi)
  - CUDA SDK 9.0 - 9.2 support for compute capability 3.0 - 7.2 (Kepler, Maxwell, Pascal, Volta)
  - CUDA SDK 10.0 - 10.2 support for compute capability 3.0 - 7.5 (Kepler, Maxwell, Pascal, Volta, Turing). Last version with support for compute capability 3.x (Kepler)

The complete compilation trajectory

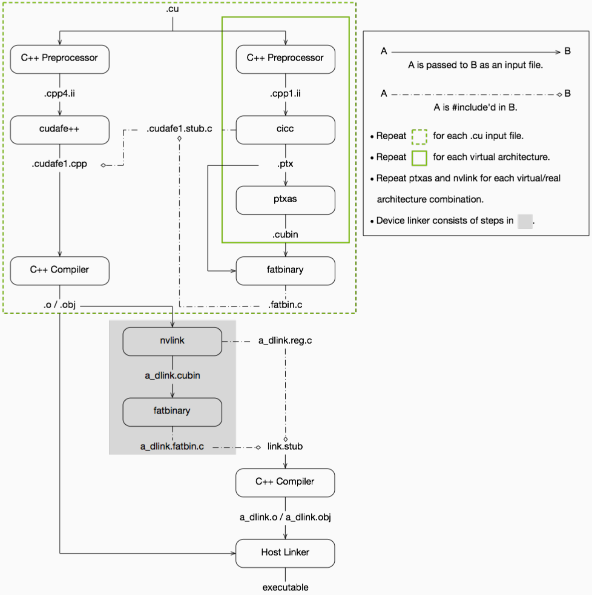

Figure 8: CUDA compilation trajectory

Whole program compilation vs Separate compilation (of device code)

Separate compilation of source code is possible.

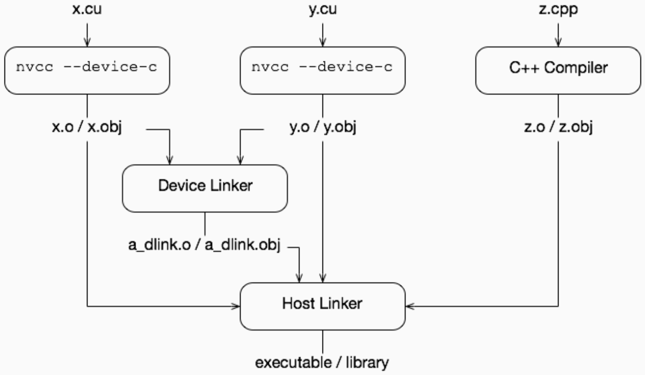

Figure 9: CUDA Separate Compilation Trajectory