CS 623: Compiling CUDA Programs

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2026-27-I

NVCC Compilation Workflow

Nvcc is a driver program based on LLVM

- Compiles and links all input CUDA files
- Requires a general-purpose C/C++ host compiler for the host code
  - Uses GCC/G++ by default on Linux platforms

High-level overview of the compilation steps

(i) Input program is preprocessed for device compilation

(ii) The device code is compiled to a CUDA binary (.cubin) and/or PTX (Parallel Thread Execution) assembly code, which are encoded in a fatbinary

(iii) Input program is processed for compilation of the host code

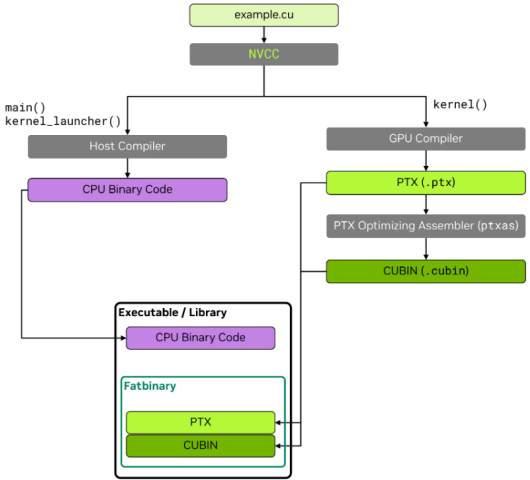
(iv) Host code and the embedded fatbinary are linked together to generate the executable

NVIDIA CUDA Compiler Driver Release 12.9

NVCC Compilation Workflow

```
#include <iostream>

__global__ void kernel() {
  printf("Hello from kernel\n");
}
void kernel_launcher() {
  kernel<<<1, 1>>>();
  cudaDeviceSynchronize();
}
int main() {
  kernel_launcher();
  return 0;
}
```

```
$ nvcc hello-kernel.cu
```



Understanding PTX, the Assembly Language of CUDA GPU Computing

Disassemble the Binaries

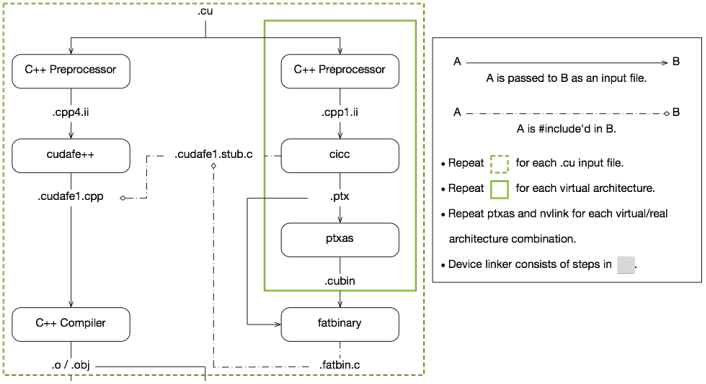
```
$ nvcc -arch=compute_86 hello-kernel.cu
$ cuobjdump a.out
```

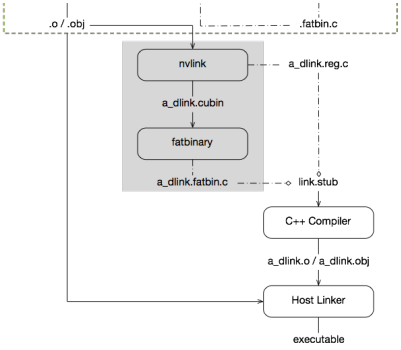
```
swarnendu@gpu3:~/cs623-2026-27-1/compilation$ cuobjdump bin/hello-kernel-compute.out

Fatbin ptx code:
================
arch = sm_86
code version = [8,5]
host = linux
compile_size = 64bit
compressed
ptxasOptions =
swarnendu@gpu3:~/cs623-2026-27-1/compilation$
```

```
$ nvcc -arch=sm_86 hello-kernel.cu
$ cuobjdump a.out
```

```
swarnendu@gpu3:~/cs623-2026-27-1/compilation$ cuobjdump bin/hello-kernel-sm.out

Fatbin elf code:
================
arch = sm_86
code version = [1,7]
host = linux
compile_size = 64bit

Fatbin elf code:
================
arch = sm_86
code version = [1,7]
host = linux
compile_size = 64bit

Fatbin ptx code:
================
arch = sm_86
code version = [8,5]
host = linux
compile_size = 64bit
compressed
ptxasOptions =
swarnendu@gpu3:~/cs623-2026-27-1/compilation$
```

CUDA Compilation Trajectory





High-Level Steps in Compilation

Preprocessor The input CUDA file containing host and device functions is preprocessed the host C/C++ preprocessor

Host/device split cudafe++ splits the input to host and device code. For the host, it extracts all CPU code, replaces CUDA-specific syntax (i.e., <<<>>>) with C function calls, and outputs a clean .cpp file that is compiled by the host compiler. For the device path, it extracts all `__global__` and `__device__` code, producing a device-only C++ file.

Compilation cicc translates device C++ code to PTX code. It uses NVVM IR for optimizations.

Assemble ptxas compiles the PTX assembly to a GPU-specific cubin file.

Linking nvcc embeds the CUBIN and PTX code in a fatbinary file, alongside the host-side compiled C++ object code.

Important NVCC Options

Options	Description
`-std {c++17|...}`	Select a particular C++ dialect
`-G`	Generate debug information for device code, turns off device optimizations
`-ccbin`	Specify the directory of the host compiler executable
`-cuda`	Generate host file that can be compiled by a host compiler
`-ptx`	Generate PTX code CUDA source
`-cubin`	Generate Cubin from CUDA or PTX source
`-fatbin`	Generate fat binary from CUDA source, PTX, or cubin files
`-arch ARCH`	Specify the **virtual** GPU architecture for generating assembly
`-code CODE`	Specify the **real** GPU architecture to assemble and optimize for
`-Xcompiler`	Pass options to the host compiler
`-Xptxas`	Pass options to the PTX assembler

NVIDIA CUDA Compiler Driver Release 12.9.2

Binary Application Compatibility

- Binaries compiled with nvcc are compatible within a major CC version if the minor version is the same or higher
  - A cubin compiled for sm_86 can be loaded on any GPU with CC sm_8x where $x \geq 6$
  - The cubin compiled for sm_86 cannot be loaded on a device with CC 8.0
- NVIDIA does not guarantee **binary** compatibility across GPU generations
  - A cubin compiled for sm_86 will not load and run on Hopper or later architectures (i.e., CC 9.0 onward)
  - Instruction set and instruction encodings of a generation may differ from other generations

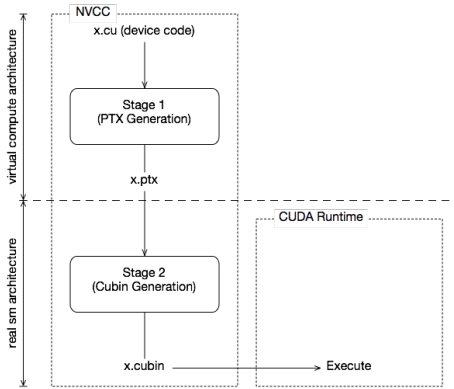
Philip Fabianek. A Gentle Introduction to CUDA PTX.

Two-Staged Compilation with Virtual and Real Architectures

nvcc relies on a two stage compilation model for ensuring application compatibility across GPU generations

(i) Code is compiled to a virtual assembly called PTX

(ii) PTX code is assembled for a real GPU architecture



Improving Application Portability with JIT Compilation

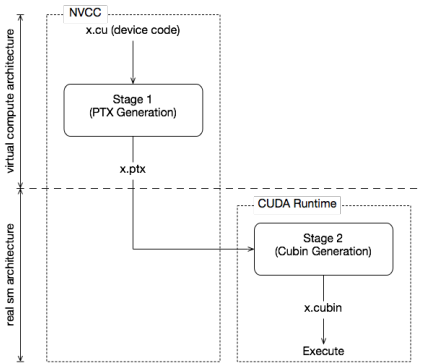
Two-stage compilation does not help improve application portability

- Generated code is bound to one GPU generation (e.g., `sm_75`)

Strategies to improve portability

Embed PTX Postpone assembly of PTX code until application run time if only a virtual GPU architecture is specified

Embed multiple cubins Generate multiple cubins for possible target architectures and embed them in a fat binary



PTX Assembly

- GPU compilation is performed via an intermediate representation called PTX*
  - PTX is an acronym for Parallel Thread eXecution
  - Goal is to provide an architecture-indent ISA for compilers
  - Can be considered as assembly for a virtual GPU architecture
- PTX code can also be inlined into CUDA with `asm`†
- Understanding PTX is important for rigorous performance analysis of kernels

> …DeepSeek actually had an excess of computing …This is actually impossible to do in CUDA. DeepSeek engineers had to drop down to PTX, a low-level instruction set for NVIDIA GPUs that is basically like assembly language. …
>
> – Ben Thompson, Stratechery, 2025.

*CUDA PTX ISA

†Inline PTX Assembly in CUDA

Understanding PTX Assembly

```
opcode[.modifier].type destination, source1, source2, ...;
```

Generate PTX code with `-ptx` option

- Memory regions in PTX are explicit
- PTX also allows cache policy hints for memory operations

%tid.x	thread index within block
%ntid.x	block dimension
%ctaid.x	block index
%nctaid.x	grid dimension
%laneid	lane index within warp
%warpid	warp index within SM context

Predication avoids changing control flow for small conditions

```
setp.lt.s32 %p1, %r1, %r2;
@%p1 add.s32 %r3, %r3, 1;
```

Branches are required for large control flow regions and loops

Makefile

Examples of PTX Instructions

```
add.s32 %r1, %r2, %r3;
mul.lo.s32 %r1, %r2, %r3;          // Integer multiplication, keep low 32 bits
mad.lo.s32 %r1, %r2, %r3, %r4;     // Multiply, keep low 32 bits, add
fma.rn.f32 %f1, %f2, %f3, %f4;     // Fused multiply-add
setp.eq.s32 %p1, %r1, %r2;
selp.s32 %r1, %r2, %r3, %p1;       // r1 = p1 ? r2 : r3
and.b32 %r1, %r2, %r3;
shr.u32 %r1, %r2, 3;               // Unsigned shift right
ld.global.f32 %f1, [%rd1];         // Other memory spaces: shared, local, constant
st.shared.f32 [%rd1], %f1;         // Other memory spaces: global, local, constant
mov.f32 %f1, %f2;
ld.global.ca.f32 %f1, [%rd1];      // Cache at both L1 and L2
ld.global.cg.f32 %f1, [%rd1];      // Cache only at L2
ld.global.cs.f32 %f1, [%rd1];      // Streaming, mark for early eviction
ld.global.cv.f32 %f1, [%rd1];      // Do not cache, fetch from global memory
```

Understanding -arch and -code

```
nvcc -arch=compute_86 hello-world.cu
```

Generates only PTX for CC 8.6 and embeds in the binary

```
nvcc -arch=compute_86 -code=sm_86 hello-world.cu
```

Generates SASS code compliant with CC 8.6

```
nvcc -arch=sm_86 hello-world.cu
```

- Implies `nvcc -arch=compute_86 -code=sm_86,compute_86 hello-world.cu`
- Embeds both the PTX and the SASS code for CC 8.6 in the binary

```
nvcc -arch=compute_75 -code=sm_86 hello-world.cu
```

Generates SASS code compliant with CC 8.6

```
nvcc -arch=compute_75 -code=sm_75,sm_86 hello-world.cu
```

Generates SASS for the two GPU architectures and embeds the cubin files in the executable

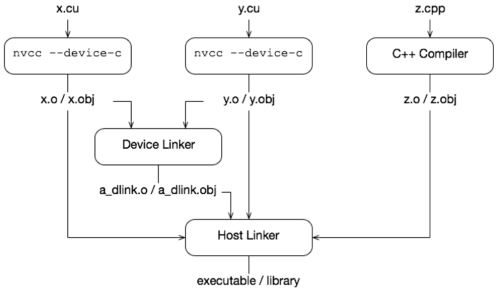
Separate Compilation in CUDA

- CUDA releases prior to version 5 did not support separate compilation
  - CUDA code could not call device functions or access variables across files
  - nvcc expected all GPU code and symbols to be present in the compilation unit that uses them
  - Such compilation is referred to as whole program compilation
- Newer CUDA releases support separate compilation, but whole program compilation is still the default
  - Use `extern` and `static` to control the visibility of symbols
- Separate compilation allows more flexible code structure, reduces compile time, and can lead to smaller binaries

Separate Compilation in CUDA

Choice 1: Compile source files into standalone intermediate object files containing relocatable device code without linking, and then perform device linking with nvlink and host linking

```
$ nvcc -arch=sm_86 -dc a.cu -o a.o
$ nvcc -arch=sm_86 -dc b.cu -o b.o
$ nvcc -arch=sm_86 a.o b.o -o a-dc.out
$ ./a-dc.out
```



Choice 2: Generate relocatable code and perform linking

```
$ nvcc -arch=sm_86 -rdc=true a.cu b.cu -o a-rdc.out
$ ./a-rdc.out
```

a.cu b.h b.cu

SASS

- SASS is the low-level instruction representation corresponding to the machine code executed on NVIDIA GPUs
  - SASS is most possibly an acronym for Source and ASSembly
  - SASS is poorly documented, and there have been numerous attempts to reverse engineer
  - Instruction scheduling, register allocation, scoreboard management, loop fusion and unrolling happens on SASS code
- CUDA includes two binary utilities for examining cubin files

  `cuobjdump` Parses both cubin files and host binaries

  `nvdisasm` Only parses cubin files but provides richer output

```
$ cuobjdump -sass vector-addition.cubin
$ nvdisasm vector-addition.cubin
```

Florian Mattana. SASS King, Part 1: Reading NVIDIA SASS from First Principles.

References

- NVIDIA Corporation. CUDA C++ Programming Guide, Release 12.9.2.
- NVIDIA Corporation. NVIDIA CUDA Compiler Driver, Release 12.9.2.
- NVIDIA Corporation. CUDA PTX ISA, Release 9.3.
- T. Scudiero and J. Bentz. Understanding PTX, the Assembly Language of CUDA GPU Computing.
- Viswa Ravichandran. CUDA PTX: Learning to Read NVIDIA's Virtual ISA.
- Viswa Ravichandran. CUDA Register Mapping: From PTX to SASS.
- Dheemanth. Intro to PTX Optimization: Part 1.