GPU Computing

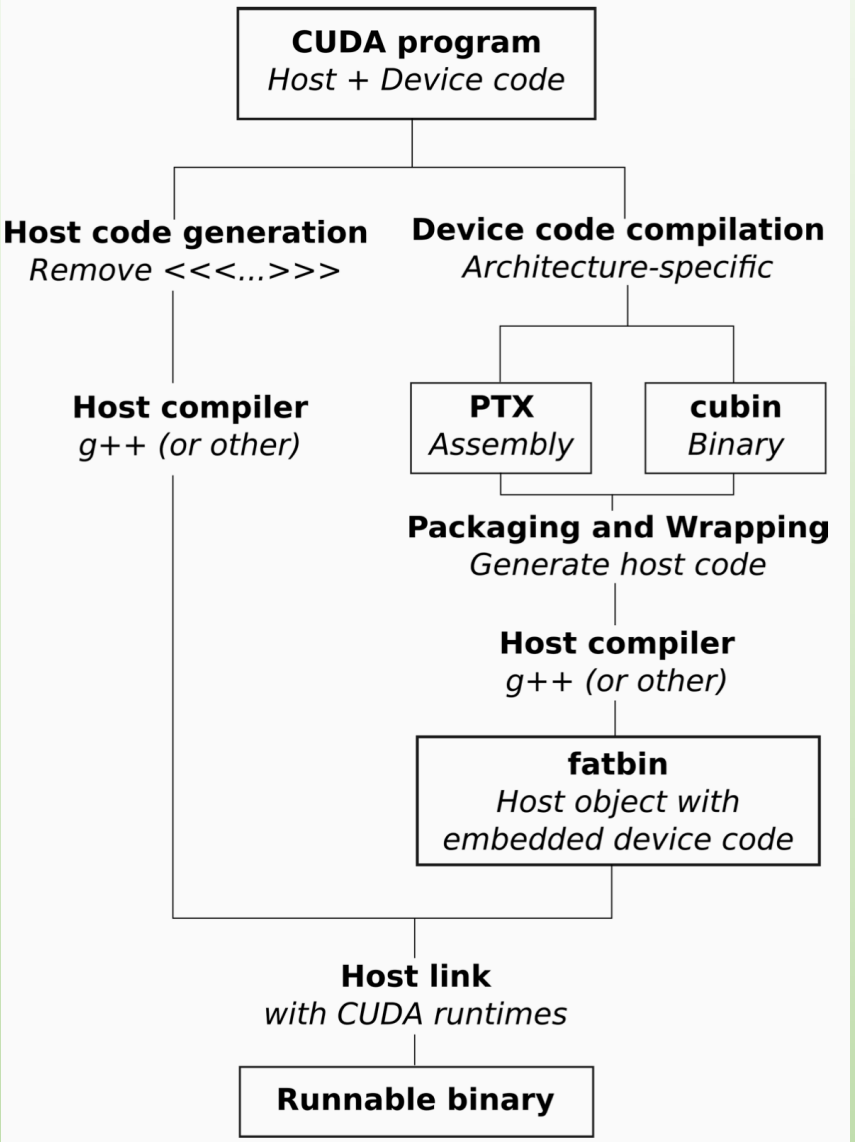
Laurea **Magistrale in Informatica** - AA 2019/20

Docente **G. Grossi**

Lezione 11 – compilation and runtime

Compilation trajectories

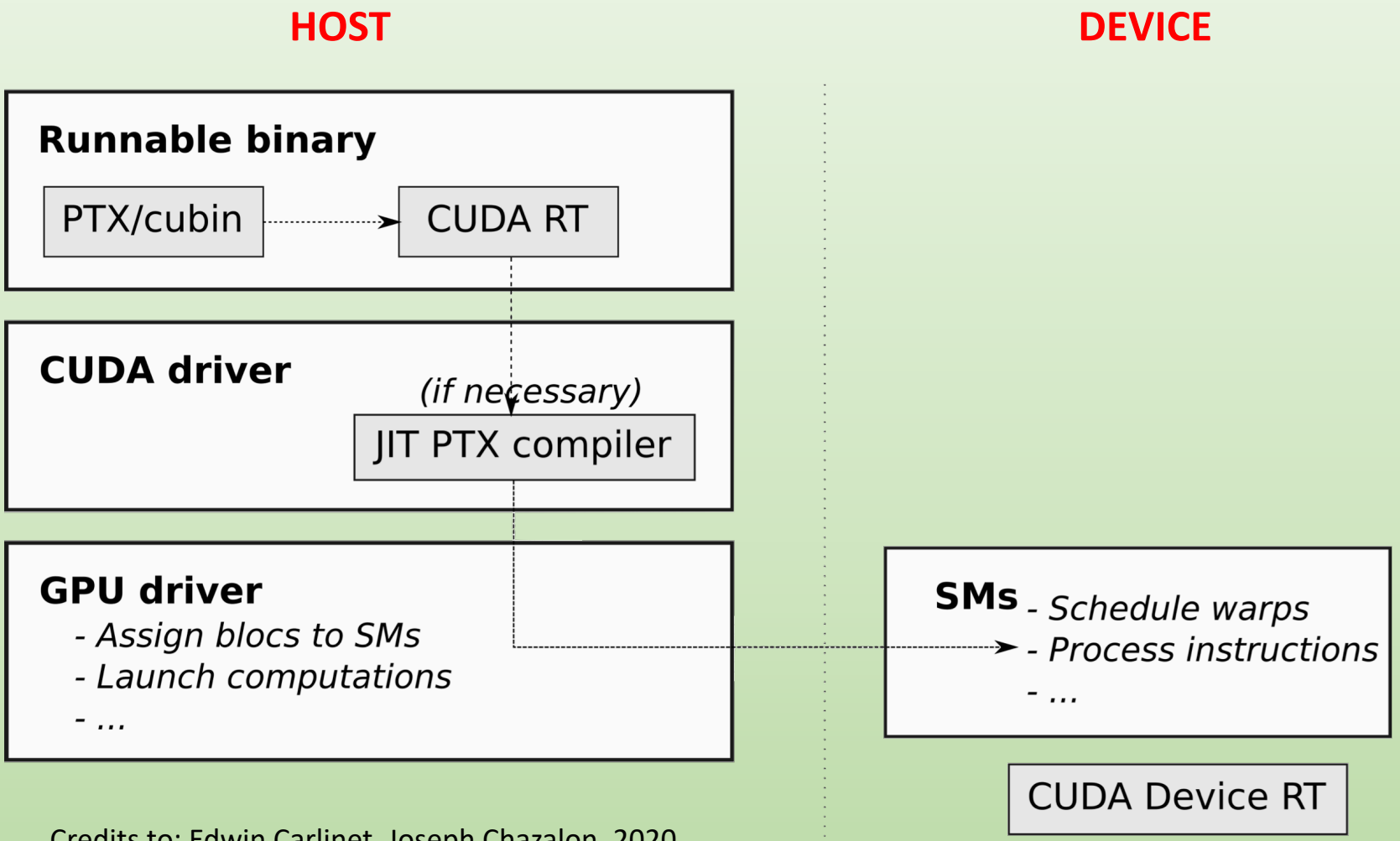
- ✓ Host and devices code follow two different compilation trajectories
- ✓ **Device code** is compiled into two **formats**:
  - **PTX** assembly tied to a virtual architecture specification
  - **cubin** binary code tied to a particular GPU product family — aka real architecture like Fermi, Kepler, Maxwell, Pascal, Volta, Turing and Ampere (soon)
- ✓ The final runnable binary contains both host and device code
- ✓ is linked with the CUDA s



Credits to: Edwin Carlinet, Joseph Chazalon, 2020

Runtime

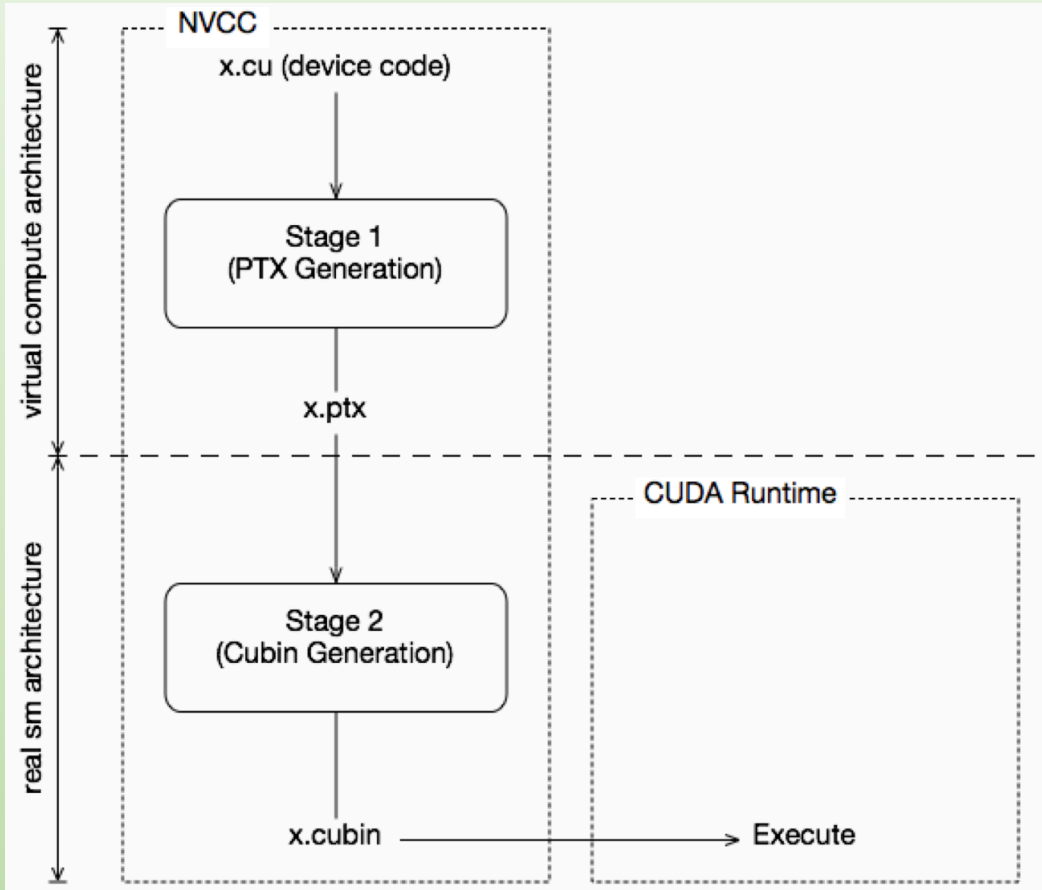
✓ Transfer of code to device with optional JIT compilation

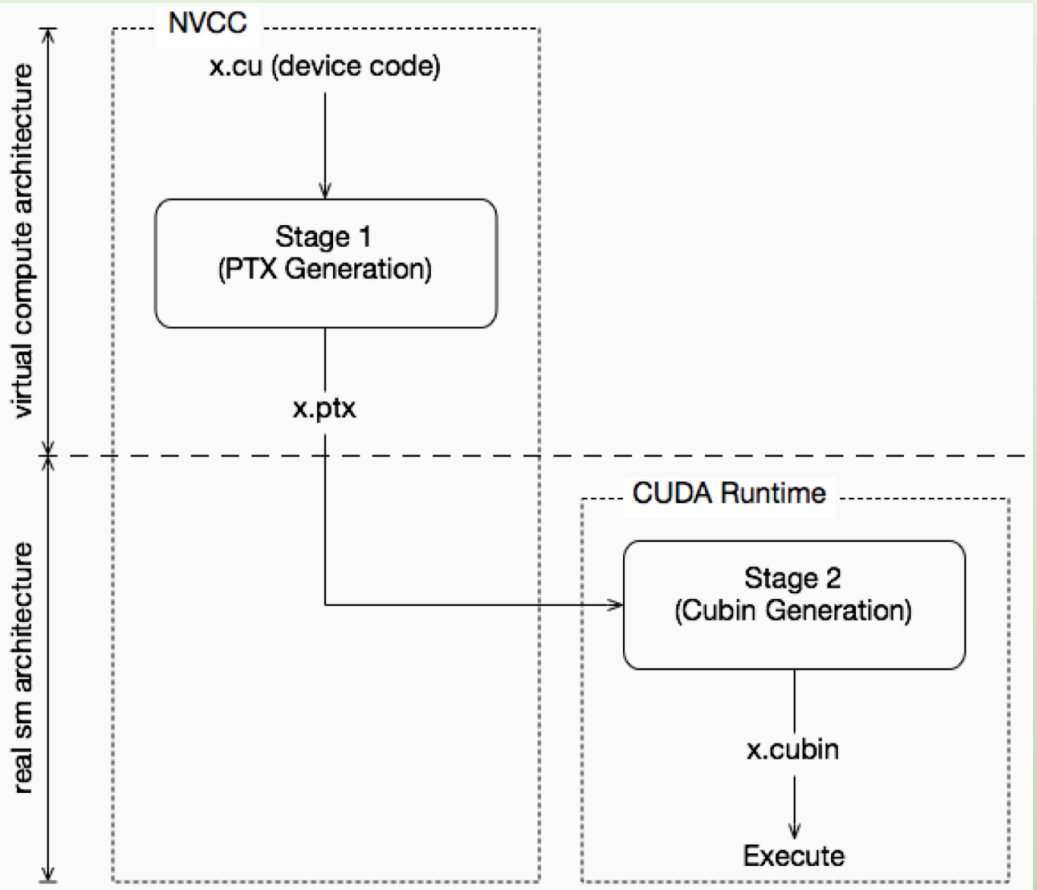


Credits to: Edwin Carlinet, Joseph Chazalon, 2020

Two-stage compilation

- ✓ **Two-Staged (offline) Compilation with Virtual and Real Architectures**
- ✓ **Just-in-Time Compilation of Device Code**





Credits to: Edwin Carlinet, Joseph Chazalon, 2020

Rationale

- ✓ 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
- ✓ 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:

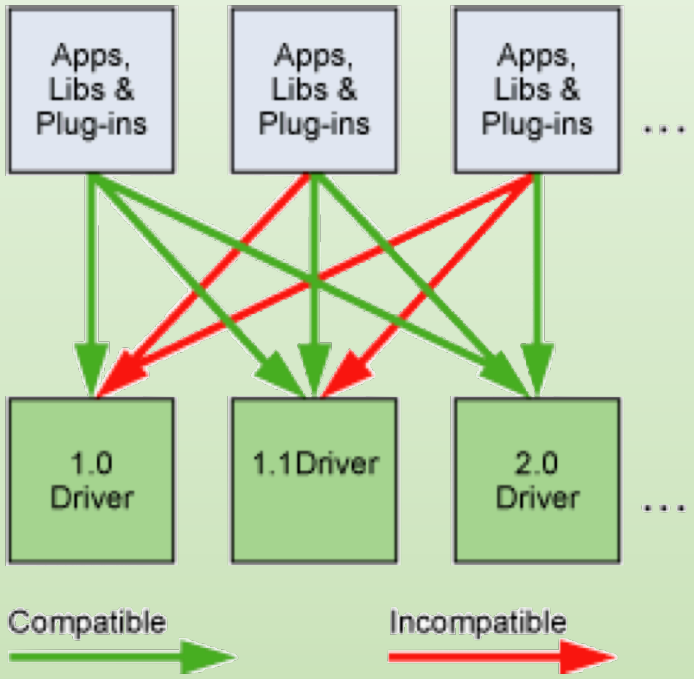
- Binary code compiled and optimized for sm_30 cards
  - can be run by sm_32 and sm_35 cards (Kepler family)
  - but cannot be run by sm_5x cards (Maxwell family).

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

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 and 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
- ✓ It may be necessary to ask the user to install a newer version of the CUDA driver on its system.



Compilation and Runtime

- ✓ 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
- ✓ 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`

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

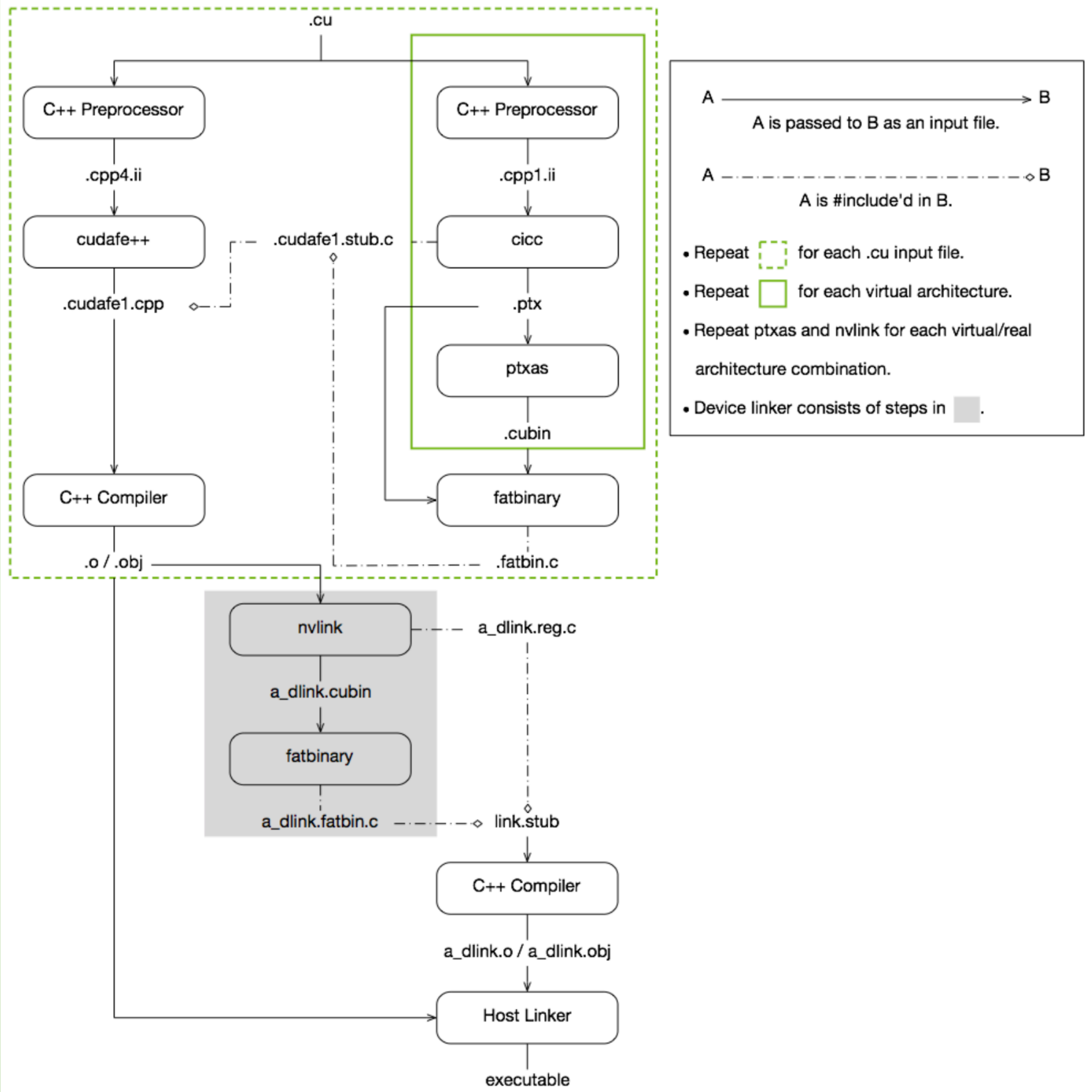
Virtual architectures

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

The CUDA Compilation Trajectory

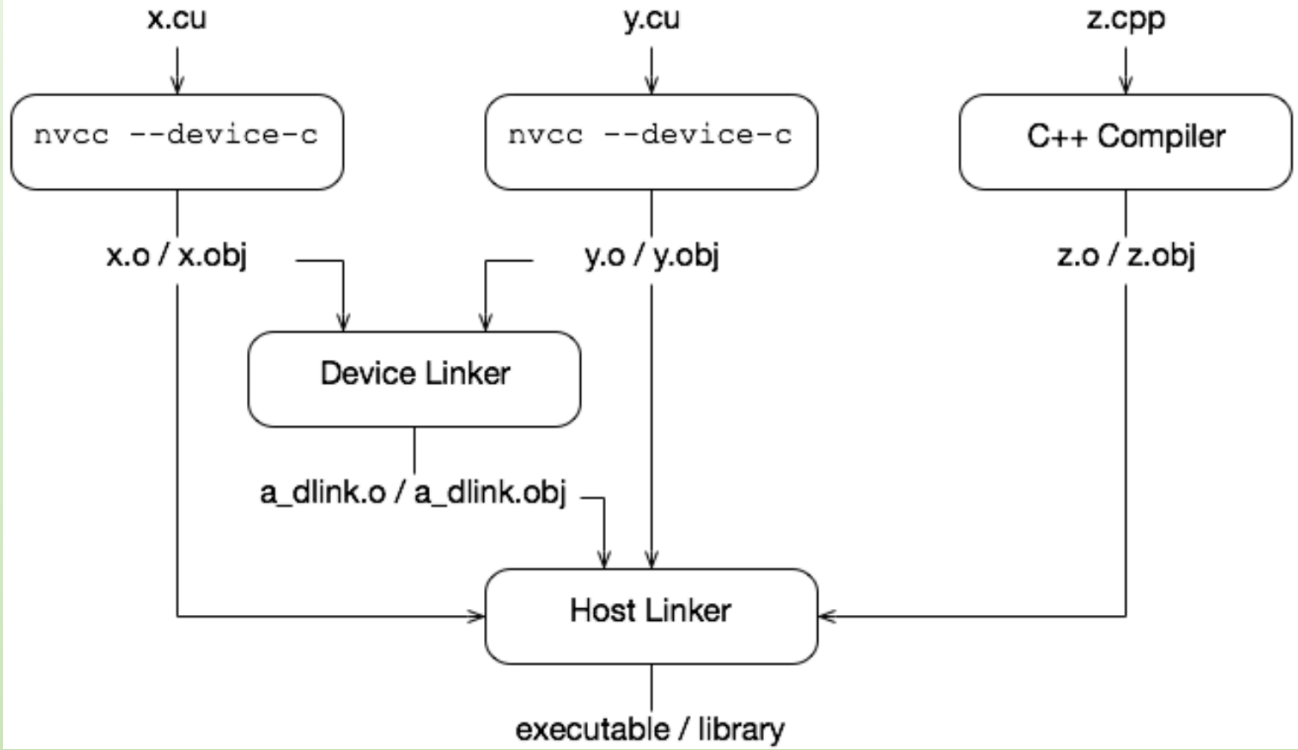
CUDA compilation works as follows:

- the input program is preprocessed for device compilation compilation and is compiled to CUDA binary (cubin) and/or PTX intermediate code, which are placed in a fatbinary.
- The input program is preprocessed once again for host compilation and is synthesized to embed the fatbinary and transform CUDA specific C++ extensions into standard C++ constructs.
- Then the C++ host compiler compiles the synthesized host code with the embedded fatbinary into a host object



NVCC Options for Separate Compilation

- ✓ CUDA works by embedding device code into host objects. In whole program compilation, it embeds executable device code into the host object
- ✓ In separate compilation, we embed relocatable device code into the host object, and run nvlink, the device linker, to link all the device code together
- ✓ The output of nvlink is then linked together with all the host objects by the host linker to form the final executable.



Example

```
//---------- b.h ----------

#define N 8

extern __device__ int g[N];

extern __device__ void bar(void);
```

```
//---------- b.cu ----------


#include "b.h"
__device__ int g[N];
__device__ void bar(void) {
     g[threadIdx.x]++;
}
```

```
//---------- a.cu ----------
#include <stdio.h>
#include "b.h"
__global__ void foo(void) {
      __shared__ int a[N];
      a[threadIdx.x] = threadIdx.x;
      __syncthreads();
      g[threadIdx.x] = a[blockDim.x - threadIdx.x - 1];
      bar();
}
int main(void) {
      unsigned int i;
      int *dg, hg[N];
      int sum = 0;
      foo<<<1, N>>>();
      if (cudaGetSymbolAddress((void**) &dg, g)) {
            printf("couldn't get the symbol addr\n");
            return 1;
      }
      if (cudaMemcpy(hg, dg, N * sizeof(int), cudaMemcpyDeviceToHost)) {
            printf("couldn't memcpy\n");
            return 1;
      }
      for (i = 0; i < N; i++)
            sum += hg[i];
      if (sum == 36)
            printf("PASSED\n");
      else
            printf("FAILED (%d)\n", sum);
      return 0;
}
```

Separate Compilation

- ✓ These can be compiled with the following commands (these examples are for Linux):

```
nvcc --gpu-architecture=sm_50 --device-c a.cu b.cu
nvcc --gpu-architecture=sm_50 a.o b.o -o exe
```

- ✓ If you want to invoke the device and host linker separately, you can do:

```
nvcc --gpu-architecture=sm_50 --device-c a.cu b.cu
nvcc --gpu-architecture=sm_50 --device-link a.o b.o --output-file link.o
g++ a.o b.o link.o --library-path=<path> --library=cudart
```

Compilazione BFS con Cmake

```
//---------- graph_d.cu ----------

#include <stdio.h>
#include <iostream>
#include "graph.h"
using namespace std;

#define cudaCheck(cuSts,file,line) {

. . .
```

```
//---------- graph.cpp ----------

. . .

void Graph::setup(node_sz nn) {
     if (GPUEnabled)
          memsetGPU(nn, std::string("nodes"));
     else {
          str = new GraphStruct();
          str->cumDegs = new node[nn + 1]{};
     }
     str->nodeSize = nn;
}
. . .
```

```
//---------- testBFS.cu ----------

. . .
__global__ void initBFS(GraphStruct *str, bool *Fa, bool *Xa, unsigned int n) {

     int nodeID = threadIdx.x + blockIdx.x * blockDim.x;

     if (nodeID > n)
          return;
     else {
          Fa[nodeID] = false;
          Xa[nodeID] = false;
     }
}
. . .
```

Makefile con cmake

- ✓ Per compilare file multipli con cmake sotto linux:
  - definire il file cmake **`CMakeLists.txt`** e inserire l'elenco dei file ***.cu** e ***.cpp** (senza aggiungere gli header ***.h**) come elenco nel comando **`add_executable(…)`**
  - porre tutti i file nella stessa dir e lanciare il commando **`'cmake .'`**
  - viene prodotto un eseguibile di nome **BFS** (nome del progetto) nella stessa dir

file:CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(BFS LANGUAGES CUDA CXX)
add_executable(bfs graph.cpp graph_d.cu testBFS.cu)
```

```
>cmake -DCMAKE_CUDA_FLAGS="-arch=sm_60" .
>make
```