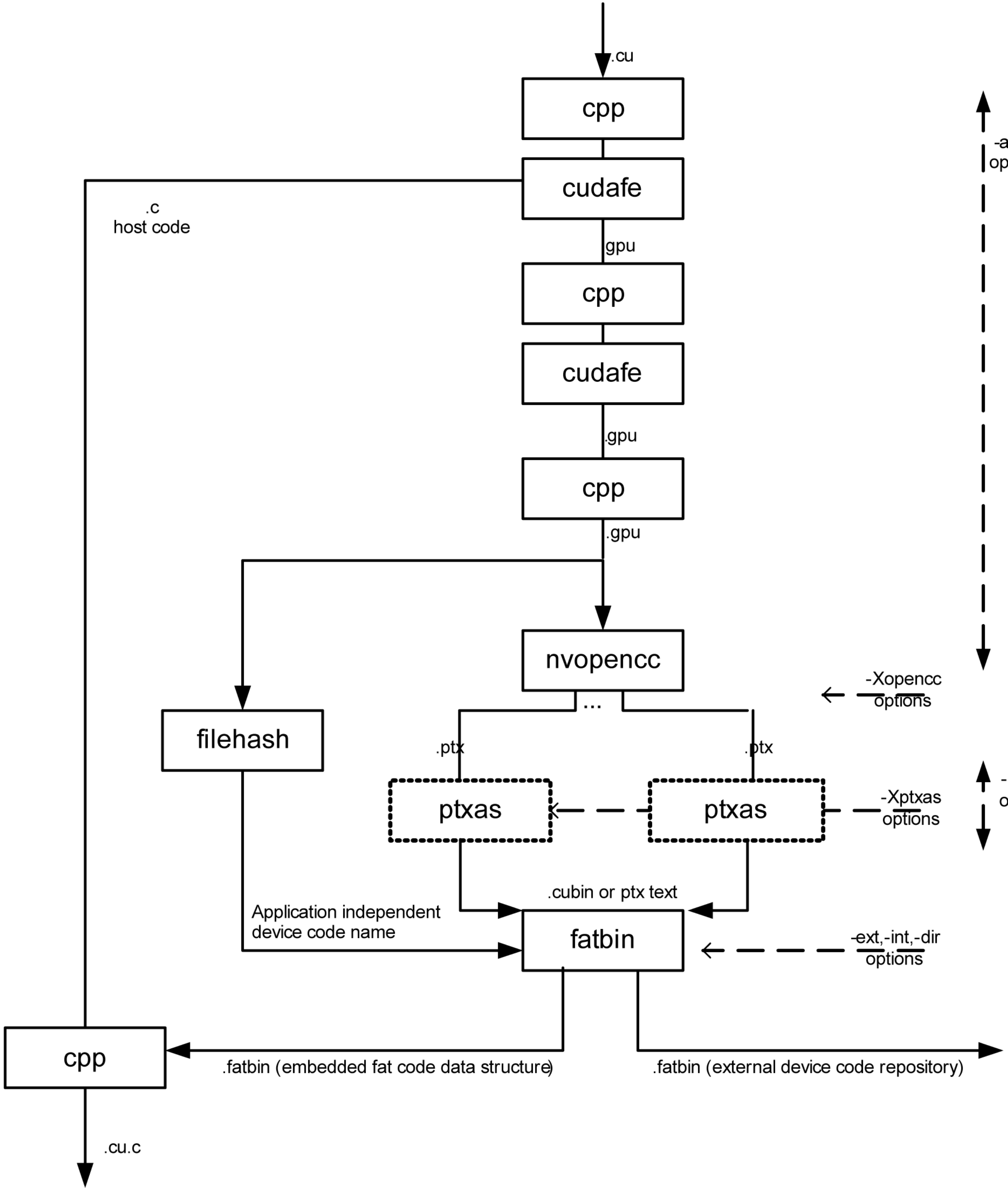
Compiler et pipeline

- Compilateur nvidia : nvcc
    - Calls host compiler c/c++ for c/c++ code et ptx for cuda code

Compilation phases overview (*trajectory*)


.cu
cpp
cudafe
.c
host code
gpu
cpp
cudafe
.gpu
cpp
.gpu
-arch
option
nvopencc
...
-Xopencc
options
filehash
.ptx
.ptx
ptxas
ptxas
-Xptxas
options
-code
option
.cubin or ptx text
Application independent
device code name
fatbin
-ext,-int,-dir
options
cpp
.fatbin (embedded fat code data structure)
.fatbin (external device code repository)
.cu.c

- cudafe (front end) : splits the code in.gpu (device code) and c code
- After : .gpu -> .cubin and / or .ptxas
- Flows through a *descriptor*
- Finally, descriptor included into *host code*
- When program is running => descriptor inspected by CUDA runtime.

With what to feed it ?

- Source files
  - .cu : CUDA source
  - .cpp / .cc / .cxx
- Des objets :
  - .o / .obj / .so
- Des librairies :
  - .lib / .a

But also…

- Intermediate assembly : .ptx
- Ressource : .res
- Gpu intermediate file : .gpu
- Pre-processed : .cup

Note : only compile .c files with nvcc if only C API is present ! (no kernels)

Step 0 : warming up

- Write a dummy C main function in a file with .c extension (eg : hello world)

Case 1 : compile a c file with only API functions

- Write any of the API functions. Example :
  ```
  float *ptr;
  cudaMalloc(&ptr, 1 * sizeof(float));
  ```

- Compile with nvcc. Works ?

Case 2 : compile a C file with a kernel inside

- Now place this block visible by the main (just at the top of main function or in a header file)
  __global__ void foo();

- Compile. Works?

Case 2 : compile a C file with a kernel inside

- Try again by changing the extension from .c to .cu

Case 3 : you want to be able compiling with gcc (no gpu)

- Remove the API function form inside the main (i.e. you can leave an empty main) but let the __global__ function.

- Compile with gcc (or any c compiler).

Case 3 : you want to be able compiling with gcc (no gpu)

- You want to keep your source code, that can run on non gpu machines

- Try wrapping the __global__ function by checking the definition of macro __NVCC__ :

  ```
  #ifdef __NVCC__
  __global__ void foo();
  #endif
  ```

Case 4 : you want to have your GPU functions in a separate file from the c file but keep the .c extension for the main

- You CANNOT have a GPU function (__global__, __device__…) inside a c / cpp file !
- Solution : compile files separately and link with a host c compiler. Don't forget to link with cuda runtime library (-L/usr/local/cuda/lib -lcudart)
- You have to use a C function (wrapper) to call your global function with the command foo<<<1,1>>>(void);
- Solution at next slide

Case 4 : you want to have your GPU functions in a separate file from the c file but keep the .c extension for the main

- Consider the files main.c and side.cu in COMPILATION folder.

```
1.gcc -c main.c # generates a object file

2.nvcc -c side.cu # also

3.gcc main.o side.o -L/usr/local/cuda/lib -lcudart
```

Case 5 : you want to dispatch your device functions into multiple files

- Function executing from the *device*, i.e. from a __global__ function, example :
  __device__ int dev();

- Place this declaration in side.cu and call it (like you would with any function) from the __global__ function in side.cu

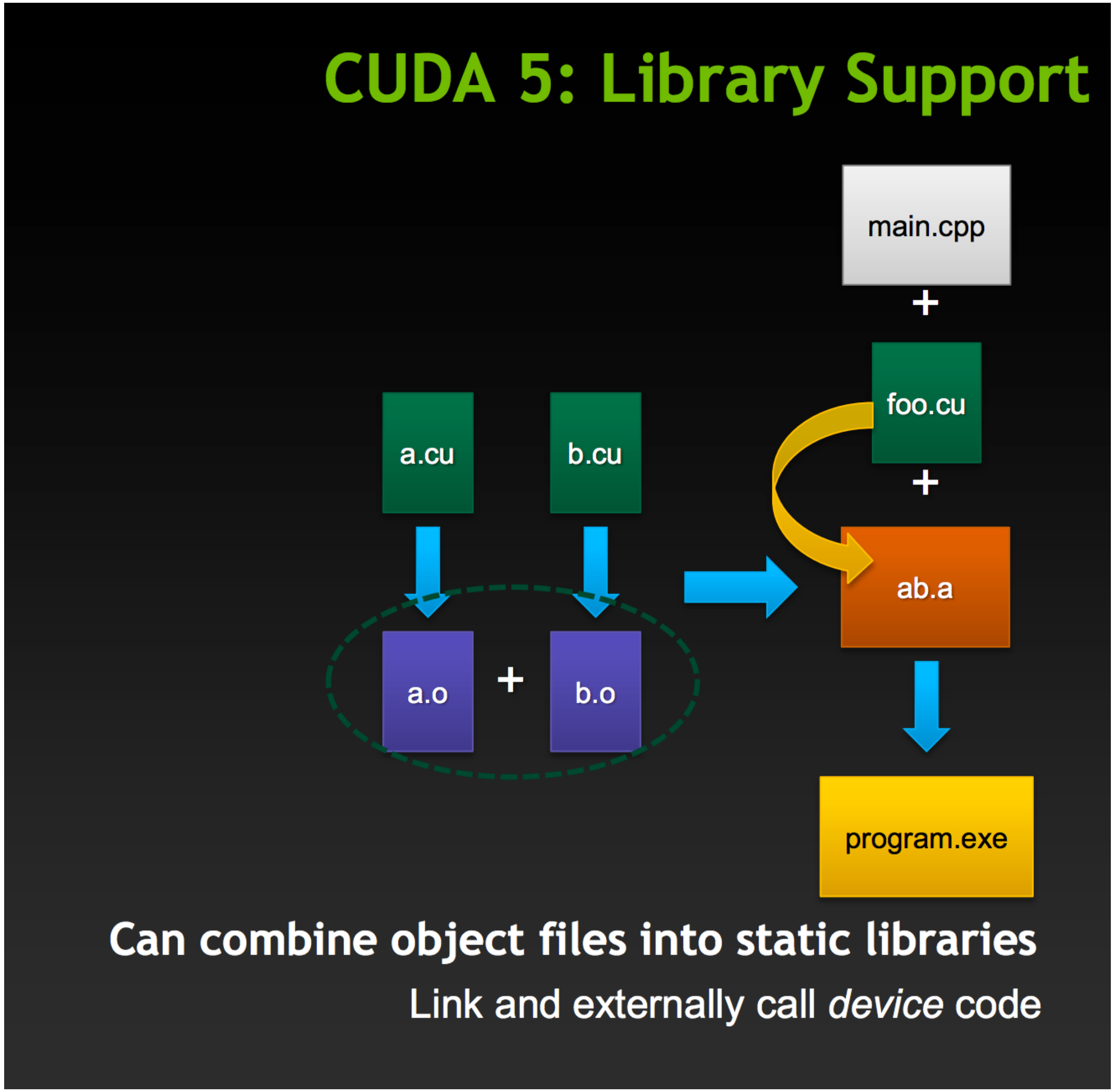
- Compile it. What happens?

« Separate compilation»

aka « relocatable device code » (RDC)

Starting from 5.0 !

Separate compilation



CUDA 5: Library Support

Can combine object files into static libraries

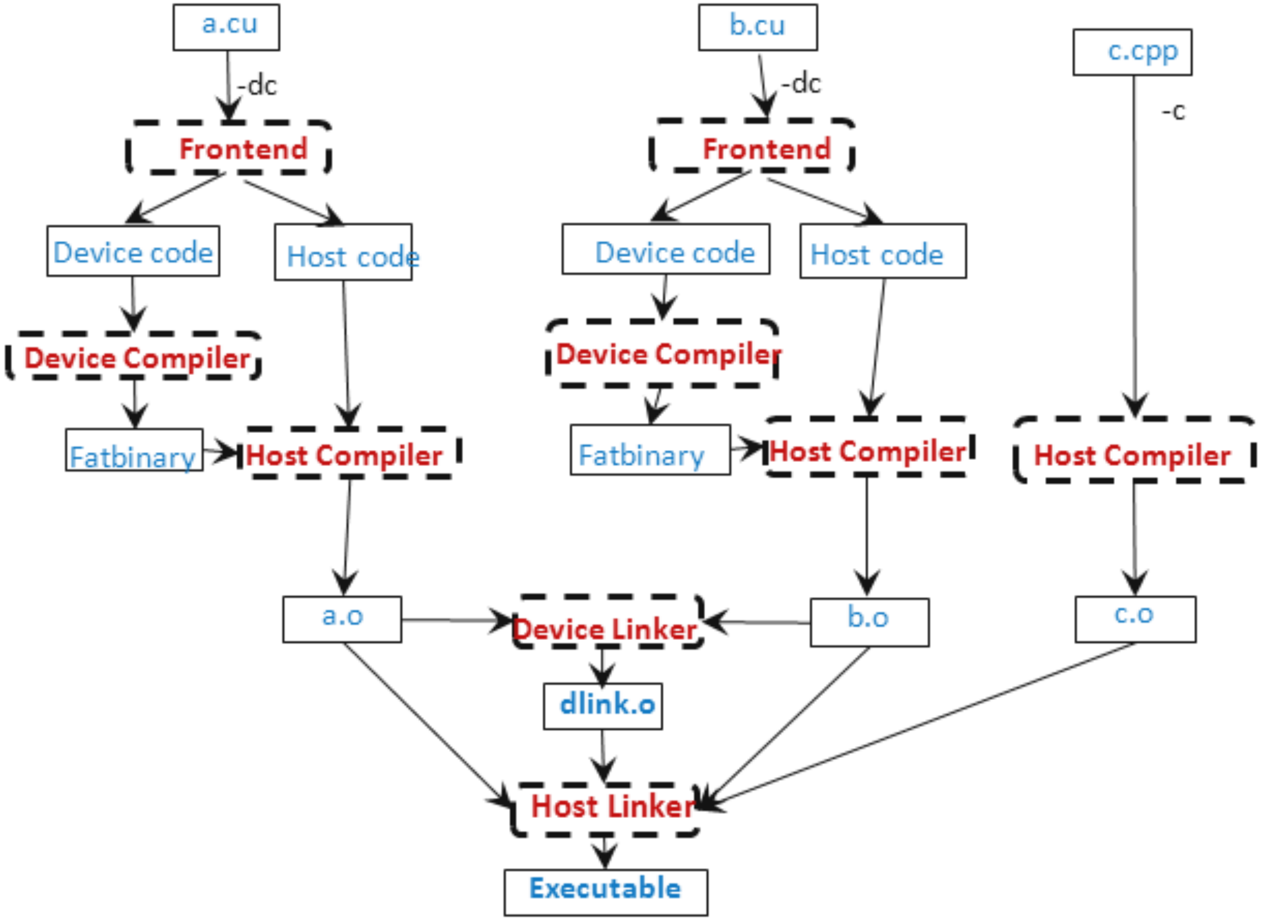
Link and externally call *device* code

- Avant : tout le *device code* dans un seul fichier !

- -dc : embed du *relocatable device code* dans du *host code* qui sera lié par *nvlink* (device linker) lors du linkage.

- nvlink peut être appelé explicitement par -dlink

- utilisation possible de *extern*



Case 6 : example of architecture code selection

- Write a .cu file with a __global__ function doing a printf(« hello world! \n»);
- Call it from the main with foo<<<1, 1>>>();
- Can you compile ?

Case 6 : example of architecture code selection

- Solution in hello_world.cu in the COMPILATION folder.
- printf is only available for architecture >= 2.0.
- Wrap the function with a #if (__CUDA_ARCH__ >= 200)

Compiler flags / options

Les plus communs

- -o *file* : le nom du output file
- -c *file* : compile le fichier en fichier objet
- -I *path* : inclut un chemin de recherche pour les headers
- -L *path* : inclut un chemin de recherche pour les librairies (phase de linkage)
- -O *level* : niveau d'optimisation (1, 2, 3…)
- -clean : supprime les fichiers temporaires qui auraient du être créés

Suite

- -m *arch* : machine (32 ou 64)
- -g : génère du code débuggable
- -G : génère du code device débugable
- -shared : génère une librairie partagée
- -M *file* : générer les dépendances pour utiliser en Makefile
- -D *macro* : définit une macro
- -U *macro* : dédéfinit une macro (undefine)

Parlons architecture

Architecture hardware / real (ex : sm_20) + architecture virtuelle / compute (ex : compute_20)

Architectures GPUs

- Tesla
  - sm_11
  - sm_12
  - sm_13
- Fermi
  - sm_20
  - sm_21
- Kepler
  - sm_30
  - sm_32
  - sm_35
- Maxwell
  - sm_50
  - sm_52

Rien d’assuré pour la compatibilité des fichiers binaires entre versions!

Good to know : compilation macros

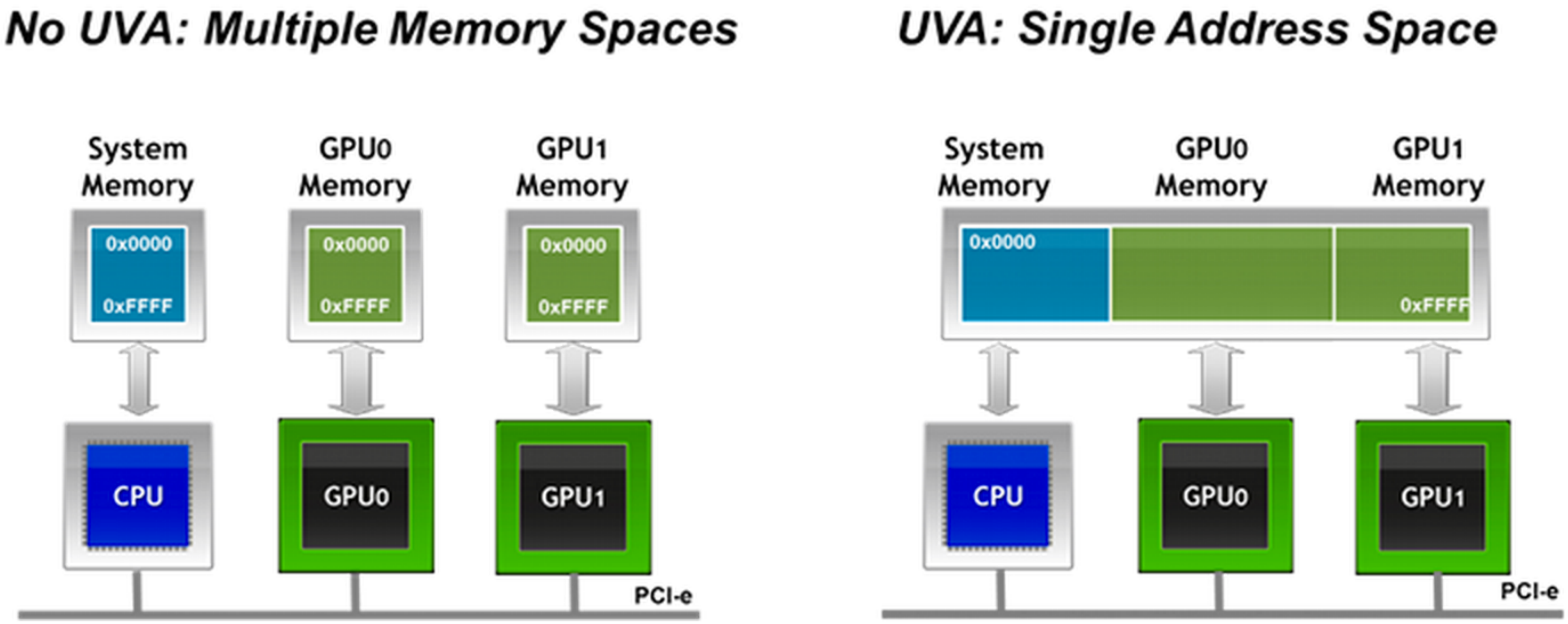
- 3 preprocessor macros are defined by nvcc :
  - __NVCC__ : When nvcc is used to compile the file
  - __CUDACC__ : When you compile cuda code (but not c / cpp)
  - __CUDACC_RDC__ : When you compile cuda code with « relocatable device code » (RDC)

Versions : What they brought to us

Release notes

- 4.0 (may 2011) unified virtual memory addressing
- 5.0 (oct. 2012)
  - dynamic parallelism (with Capability 3.5)
  - separate compilation (http://on-demand.gputechconf.com/g express/2012/presentations/gpu-object-linking.pdf)
- 6.0 (april 2014)
  - transparent memory copies handling (utlisation d’un pointe unique)
  - XT libraries (cublas & cufft) for automatic scaling to multiple GPUs (!)
  - Support for Maxwell architecture (sm_5.0)
- 7.0 (march 2015)
  - C++11 support !
  - cuSOLVER library (linear operations for eigen problems, de and sparse solvers)

Unified virtual memory addressing (> 4.0)



No UVA: Multiple Memory Spaces

UVA: Single Address Space

- +
- added routines to CUBLAS & CUSPARSE
- added math functions (sincos, sincospi, cospi, sinpi…) and performance improvements
- added distributions and generators to CURAND

Gcc/G++ compatible versions

5.0 : up to gcc 4.7
6.0 : up to gcc 4.8
7.0 : up to gcc 4.9

Note : C++ 2011 only supported starting from 7.0 !