GPU Computing

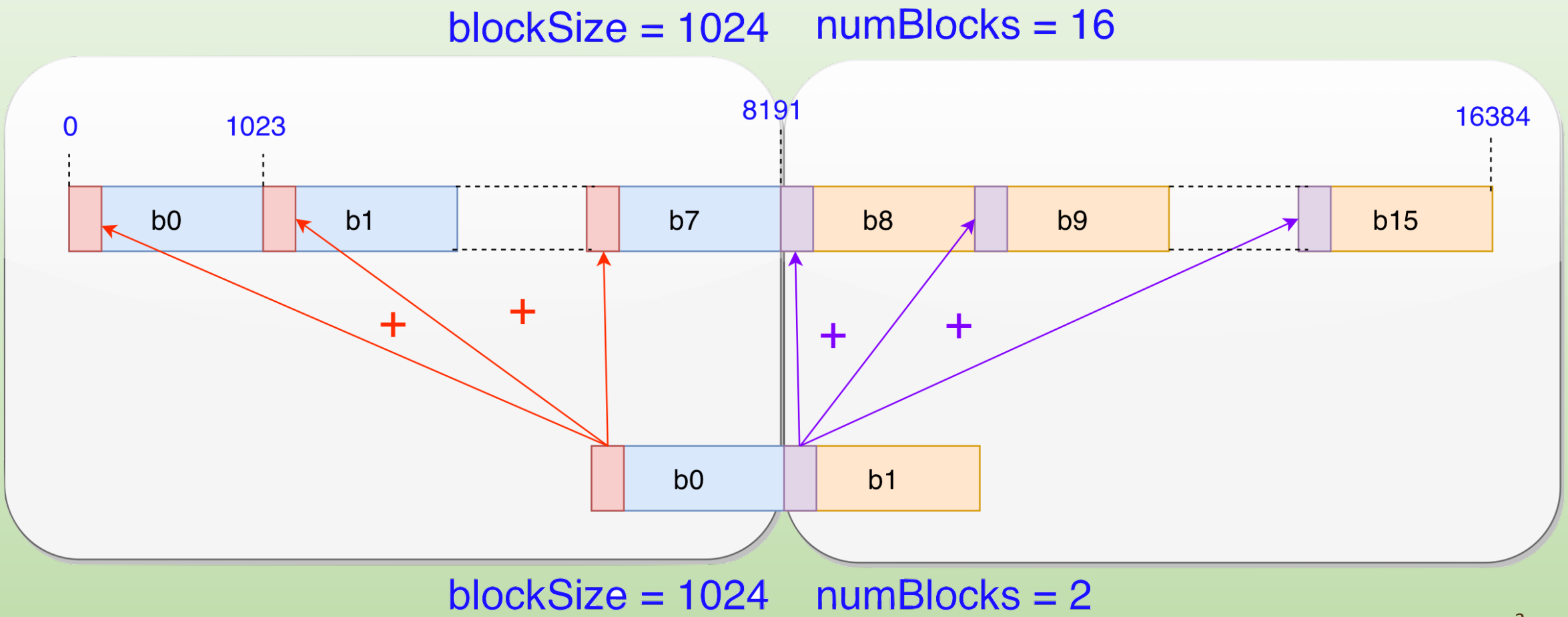
Lab 5

Loop unrolling

Ottimizzazione di loop

Block unrolling

unrolling factor = 8



Block unrolling

Indice globale elementi dell'array in

Offset puntatore elementi dell'array in nel gruppo di 8 block che devono essere ridotti a 1 block

```
__global__ void multBlockParReduceUroll8(int *in, int *out, ulong n) {
    uint tid = threadIdx.x;
    ulong idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;

    // boundary check
    if (idx >= n)
        return;

    // convert global data pointer to the local pointer of this block
    int *thisBlock = in + blockIdx.x * blockDim.x * 8;

    // unrolling 8 blocks
        . . .
```

Esecuzione su pascal P100

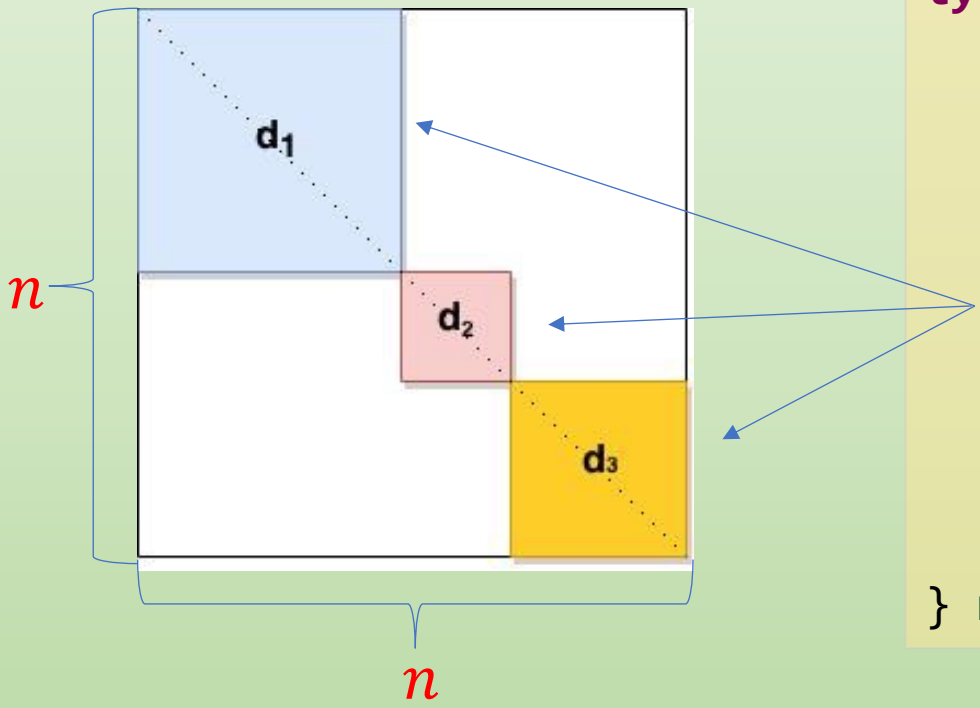
```
****  test on parallel reduction  ****
  Vector length: 3072.00 MB  sum: 3221225472

  CPU procedure...
    Elapsed time: 1.578515 (sec)

  GPU kernels (mem required 12884901888 bytes)

  Launch kernel: blockParReduce1...
    Elapsed time: 0.395506 (sec) - speedup 4.0

  Launch kernel: blockParReduce2...
    Elapsed time: 0.154076 (sec) - speedup 10.2

  Launch kernel: blockParReduceUroll...
    Elapsed time: 0.119489 (sec) - speedup 13.2

  Launch kernel: multBlockParReduceUroll8...
    Elapsed time: 0.032330 (sec) - speedup 48.8

  Launch kernel: multBlockParReduceUroll16...
    Elapsed time: 0.026311 (sec) - speedup 60.0
```

Dynamic Parallelism

Applicazione al prodotto MQDB

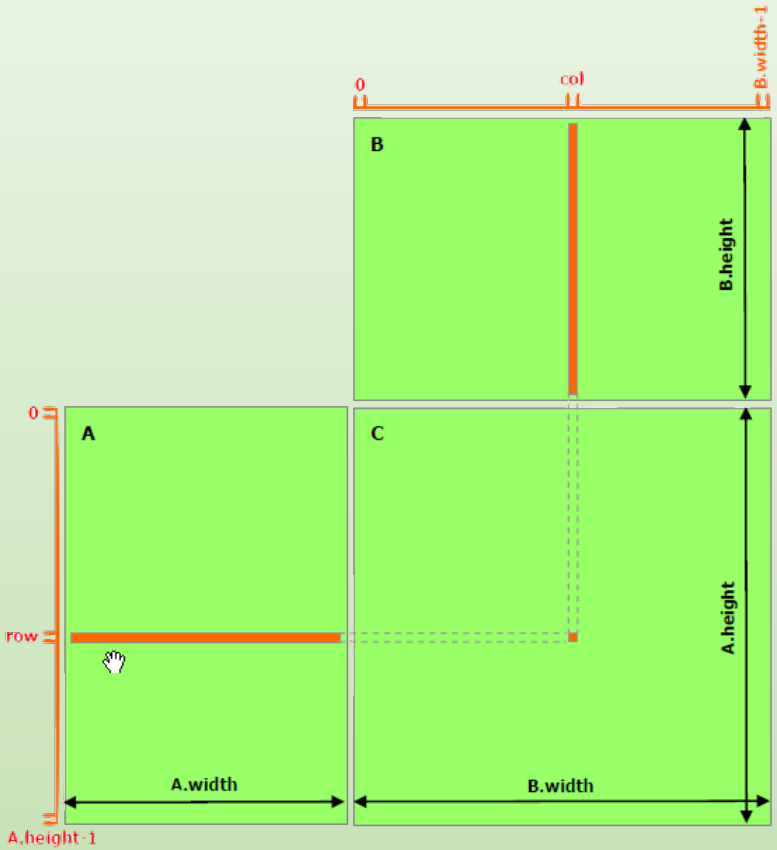
Tipo MQDB

- ✓ k blocchi sulla diagonale: $B_1, \ldots, Bk\ (0 \leq k \leq n)$
- ✓ Elementi della diagonale: $D = \{d_1, \ldots, d_k\} : n = \sum_{j=1}^{k} d_j$




```
typedef struct MQDB {
   char desc[100];    // description
   int nBlocks;       // num. of blocks k
   int *blkSize;      // block dimensions: d_1,…,d_k
   float *elem;       // elements in row-major order
   ulong nElems;      // actual number of elements
} mqdb;
```

Prodotto matriciale (naive) MQDB




```
/*
 * Kernel for standard (naive) matrix product
 */
__global__ void matProd(mqdb A, mqdb B, mqdb C, int n) {
   // row & col indexes
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;

   // each thread computes an entry of the product matrix
   if ((row < n) && (col < n)) {
      float val = 0;
      for (int k = 0; k < n; k++) {
         val += A.elem[row * n + k] * B.elem[k * n + col];
      }
      C.elem[row * n + col] = val;
  }
}
```

Kernel per prodotto MQDB

- ✓ Introdurre il **loop con CPU** per il prodotto sui singoli blocchi
- ✓ Valutare il **numero** di **grid** adeguate (adattamento ai dati?)
- ✓ Mostrarne **vantaggi** e **svantaggi sperimentalmente**

Kernel per prod MQDB

sub-block$_i$: salto dei blocchi (**`n * sdim`**) precedenti 1,2,…,i-1 più num cols blocco attuale (**`sdim`**)

loop standard per prodotto matriciale con controllo sui limiti pari alla dimensione **`d`** del blocco

```
/*
 * Kernel for block sub-matrix product of mqdb
 */
__global__ void mqdbBlockProd(mqdb A, mqdb B, mqdb C, int sdim, int d, int n) {
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;

   // jump to the right block sub-matrix
   int  offset = (n+1)*sdim;


   // each thread computes an entry of the product matrix
   if ((row < d) && (col < d)) {
      float val = 0;
      for (int k = 0; k < d; k++)
         val += A.elem[row * n + k + offset] * B.elem[k * n + col + offset];
      C.elem[row * n + col + offset] = val;
   }
}
```

Invocazione del kernel mqdbBlockProd

ciclo sui blocchi di sotto-matrici 1,2 …,k

riga dove inizia il prossimo blocco

```
// MAIN
. . .
printf("Kernel MQDB product...\n");
uint sdim = 0;
start = seconds();
for (uint i = 0; i < k; i++ ) {
   uint d = A.blkSize[i];
   mqdbBlockProd<<<grid, block>>>(d_A, d_B, d_C, sdim, d, n);
   sdim += d;
}
CHECK(cudaDeviceSynchronize());
```

Parall dinamico per prodotto MQDB

- ✓ Introdurre il **parallelismo dinamico** per il prodotto
- ✓ Valutare il **numero** di **grid** adeguate (quante grid figlio?)
- ✓ Mostrarne **vantaggi** e **svantaggi sperimentalmente**

Risultati MQDB

Matrice MQDB: n= 16*1024 (size = 1GB), k = 10 blocchi sulla diagonal (avg block size = 1638)

Dati su GPU Tesla P100-PCIE-16GB

```
*****   k = 10 --- (avg block size = 1638.4)
Memory size required = 1024.0 (MB)

CPU MQDB product...
   CPU elapsed time: 646.9 (sec)

Kernel (naive) mat product...
   elapsed time:                34.31 (sec)
   speedup vs CPU MQDB product: 18.85
   Arrays match


Kernel MQDB product...
   elapsed time:                   0.62 (sec)
   speedup vs CPU MQDB product:    1042.53
   speedup vs GPU std mat product: 55.29
   Arrays match

. . .
```

```
. . .


Kernel MQDB product with dynamic paral GRID(1)...
   elapsed time:                          0.62 (sec)
   speedup vs CPU MQDB product:           1037.28
   speedup vs GPU std mat product:        55.02
   speedup vs GPU MQDB product:           0.99
   Arrays match


Kernel MQDB product with dynamic paral GRID(k)...
   elapsed time:                          2.12 (sec)
   speedup vs CPU MQDB product:           305.25
   speedup vs GPU std mat product:        16.19
   speedup vs GPU MQDB product:           0.29
   speedup vs GPU MQDB product GRID(1): 0.29
   Arrays match
```

Risultati MQDB

Matrice MQDB: n= 16*1024 (size = 1GB), k = 30 blocchi sulla diagonal (avg block size = 546)

Dati su GPU Tesla P100-PCIE-16GB

```
*****   k = 30 --- (avg block size = 546.133362)
Memory size required = 1024.0 (MB)

CPU MQDB product...
   CPU elapsed time: 34.4 (sec)

Kernel (naive) mat product...
   elapsed time:                 34.30 (sec)
   speedup vs CPU MQDB product: 1.01
   Arrays match


Kernel MQDB product...
   elapsed time:                    0.12 (sec)
   speedup vs CPU MQDB product:     284.68
   speedup vs GPU std mat product:  283.13
   Arrays match

. . .
```

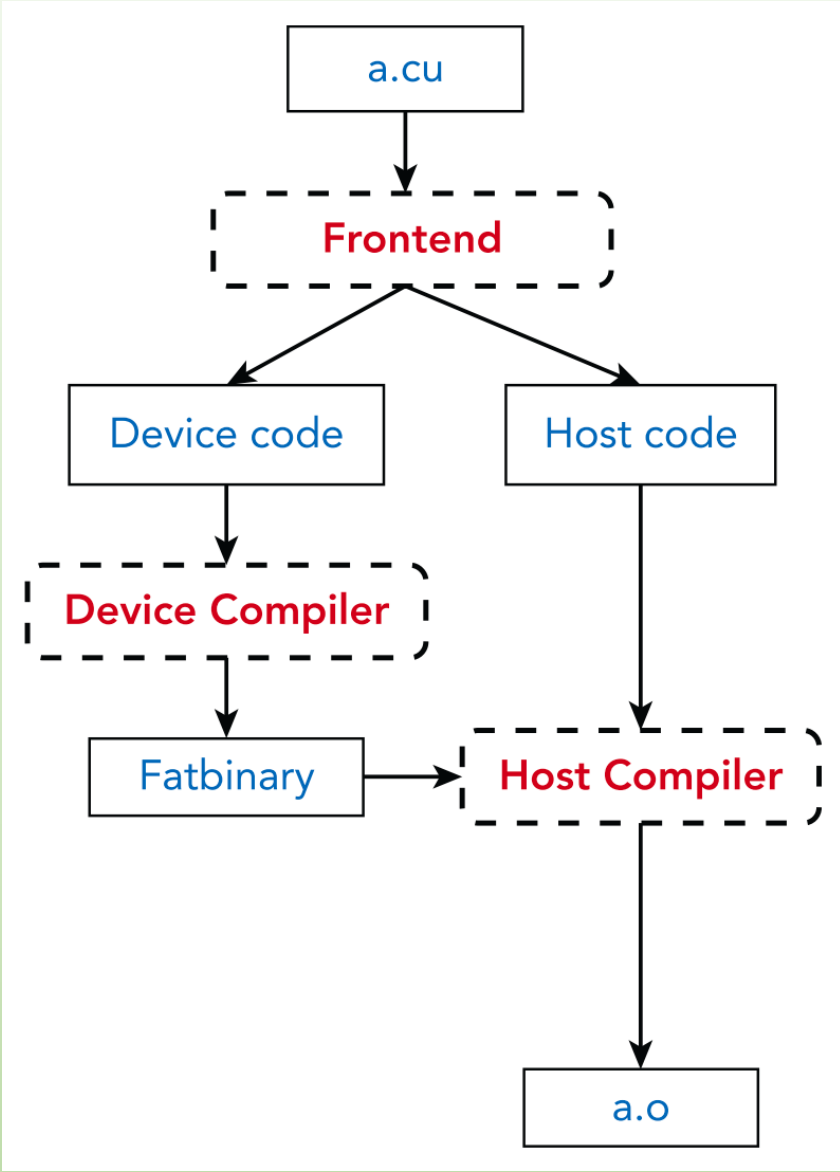
```
. . .


Kernel MQDB product with dynamic paral GRID(1)...
   elapsed time:                          0.12 (sec)
   speedup vs CPU MQDB product:           281.44
   speedup vs GPU std mat product:        279.91
   speedup vs GPU MQDB product:           0.99
   Arrays match


Kernel MQDB product with dynamic paral GRID(k)...
   elapsed time:                          0.08 (sec)
   speedup vs CPU MQDB product:           424.23
   speedup vs GPU std mat product:        421.92
   speedup vs GPU MQDB product:           1.49
   speedup vs GPU MQDB product GRID(1): 1.51
   Arrays match
```

CUDA code compilation

CUDA compilation trajectory

CUDA Code Compilation

- ✓ Within a device code file there are **device functions** and **host functions** that call device functions or manage device resources
- ✓ CUDA provides **two methods** for compiling
  - **Whole-program compilation** (prior to CUDA 5.0)
  - **Separate compilation**
- ✓ Separate compilation improves CUDA project management with the following benefits:
  - **Eases porting** of legacy C code to CUDA
  - **Facilitates** code **reuse** and **reduces** compile time
  - **link** and **call** external device code



GPU generations

- ✓ In order to allow for **architectural evolution**, NVIDIA GPUs are released in **different generations**
- ✓ **New generations** introduce major **improvements** in **functionality** and/or chip **architecture**, while GPU models within the same generation show minor configuration differences that moderately affect functionality, performance, or both
- ✓ **Binary compatibility** of GPU applications is **not guaranteed** across **different generations**. For example, a CUDA application that has been compiled for a Fermi GPU will very likely not run on a Kepler GPU (and vice versa). This is the **instruction set** and **instruction encodings** of a generation is different from those of of other generations.
- ✓ **Binary compatibility within** one GPU **generation** can be guaranteed under **certain conditions** because they share the basic instruction set.

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda

CUDA naming scheme

GPUs are named `sm_xy`, where `x` denotes the **GPU generation** number, and `y` the **version** in that generation

Names of the current GPU architectures, annotated with the functional capabilities that they provide

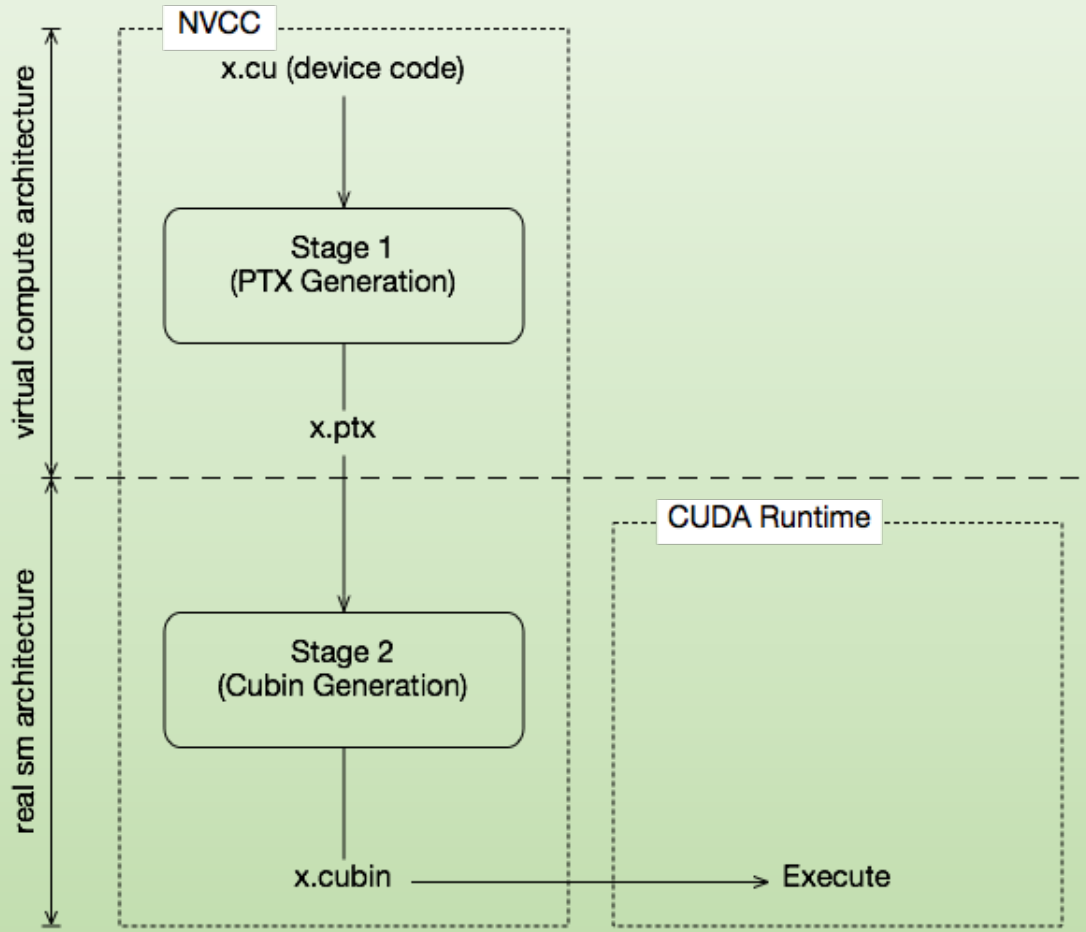
sm_35	Basic features + Kepler support + Unified memory programming + Dynamic parallelism support
sm_50, and sm_53	+ Maxwell support
sm_60, sm_61, and sm_62	+ Pascal support
sm_70 and sm_72	+ Volta support
sm_75	+ Turing support

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda

Virtual architectures

- ✓ GPU **compilation** is performed via an **intermediate representation**, **PTX**, which can be considered as assembly for a **virtual GPU architecture**
- ✓ A **virtual GPU architecture** provides a (largely) **generic instruction set**, and binary **instruction encoding** is a non-issue because PTX programs are always represented in text format

Two-Staged Compilation with Virtual and Real Architectures



https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda

Virtual architecture features

The virtual architecture naming scheme is the same as the real architecture naming scheme

compute_35, and compute_37	Kepler support Unified memory programming 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
compute_80	+ Ampere support

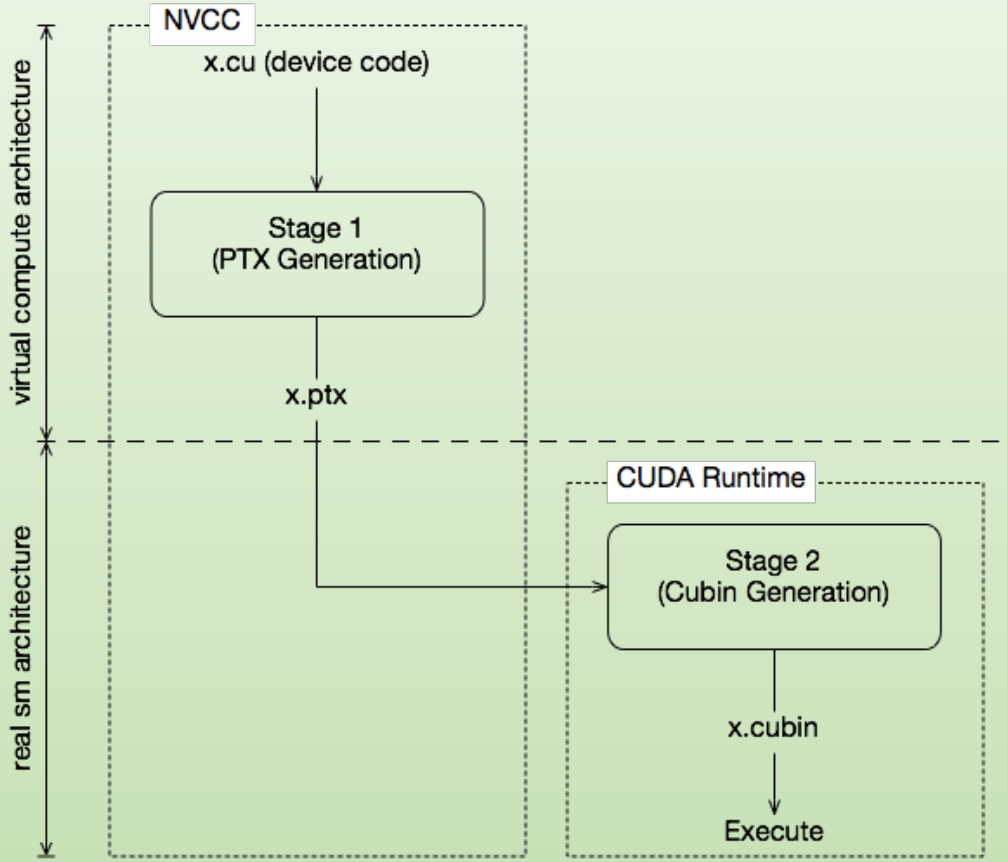
https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda

PTX code compatibility

- ✓ Assembly code 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 will not run on old hardware unless special care is taken

JIT compilation of device code

- ✓ The **compilation step** to an actual **GPU binds** the **code** to **one generation** of GPUs. Within that generation, it involves a choice between GPU coverage and possible performance
- ✓ By specifying a **virtual code architecture** instead of a real GPU, nvcc **postpones** the assembly of PTX code until application runtime at which the **target GPU** is exactly known



https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda

Example

- ✓ The command below allows generation of **exactly matching GPU** binary code when the application is launched on an **sm_50** or **later architecture**

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50
```

- ✓ **Fatbinary**: A different solution to **overcome startup delay** by **JIT** while still allowing execution on newer GPUs is **to specify multiple code** instances

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50,sm_50,sm_52
```

  - ✓ This command generates exact code for **two Maxwell variants**, plus **PTX code** for use by **JIT** in case a **next-generation** GPU is encountered. nvcc organizes its device code in fatbinaries, which are able to hold multiple translations of the same GPU source code. At runtime, the CUDA driver will select the most appropriate translation when the device function is launched

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda

Options for Steering GPU Code Generation

Specify the name of the class of NVIDIA virtual GPU architecture for which the CUDA input files must be compiled

```
--gpu-architecture arch (-arch)
```

- ✓ With the exception as described for the shorthand below, the architecture specified with this option must be a virtual architecture (such as compute_50). Normally, this option alone does not trigger assembly of the generated PTX for a real architecture (that is the role of nvcc option --gpu-code, see below)
- ✓ rather, its purpose is to control preprocessing and compilation of the input to PTX
- ✓ For convenience, in case of simple nvcc compilations, the following shorthand is supported. If no value for option --gpu-code is specified, then the value of this option defaults to the value of --gpu-architecture
- ✓ In this situation, as only exception to the description above, the value specified for --gpu-architecture may be a real architecture (such as a sm_50), in which case nvcc uses the specified real architecture and its closest virtual architecture as effective architecture values
- ✓ For example, **`nvcc --gpu-architecture=sm_50`** is equivalent to

  `nvcc --gpu-architecture=compute_50 --gpu-code=sm_50,compute_50`

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda

Options for Steering GPU Code Generation

Specify the name of the NVIDIA GPU to assemble and optimize PTX for

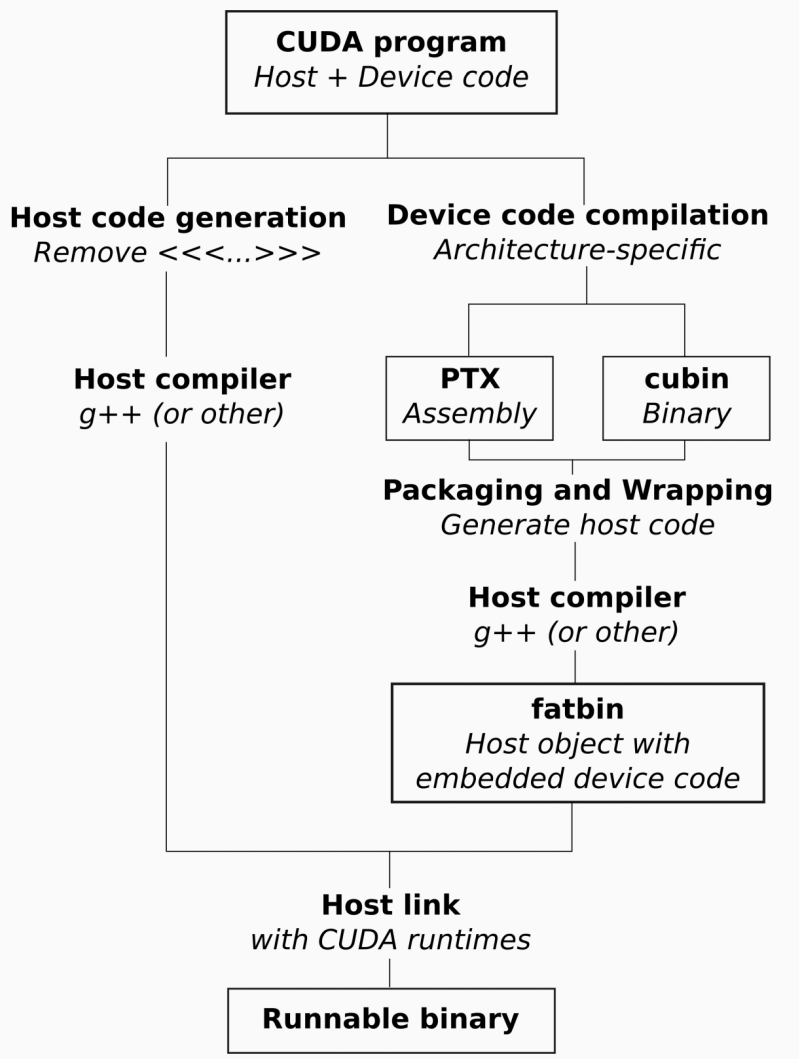
```
--gpu-code code,... (-code)
```

nvcc embeds a compiled code image in the resulting executable for each specified code architecture, which is a true binary load image for each real architecture (such as sm_50), and PTX code for the virtual architecture (such as compute_50).
During runtime, such embedded PTX code is dynamically compiled by the CUDA runtime system if no binary load image is found for the current GPU.
Architectures specified for options --gpu-architecture and --gpu-code may be virtual as well as real, but the code architectures must be compatible with the arch architecture. When the --gpu-code option is used, the value for the --gpu-architecture option must be a virtual PTX architecture.
For instance, --gpu-architecture=compute_60 is not compatible with --gpu-code=sm_52, because the earlier compilation stages will assume the availability of compute_60 features that are not present on sm_52

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda

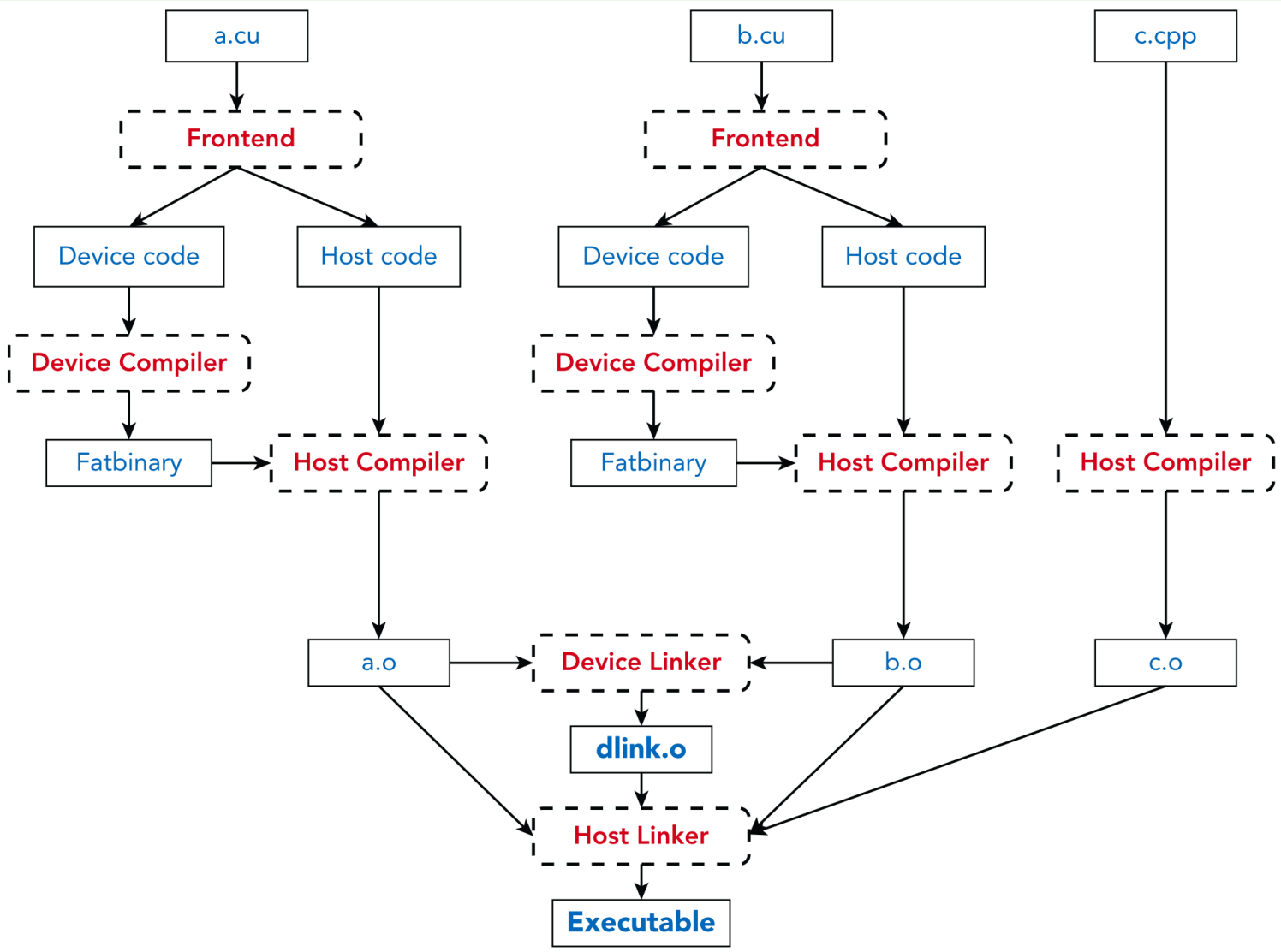
Compilation overview

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)

Example


a.cu
b.cu
c.cpp
Frontend
Frontend
Device code
Host code
Device code
Host code
Device Compiler
Device Compiler
Fatbinary
Host Compiler
Fatbinary
Host Compiler
Host Compiler
a.o
Device Linker
b.o
c.o
dlink.o
Host Linker
Executable

Compilation using nvcc

- ✓ Consider a simple example that contains three files: **a.cu**, **b.cu**, and **c.cpp**
- ✓ Suppose that some kernel functions in file **a.cu** refer to some functions or variables in file **b.cu**
- ✓ Because you reference across two files, you must use separate compilation to generate the executable

```
nvcc –arch=sm_20 –dc *.cu *.cpp
nvcc –arch=sm_20 *.o –o test
```

Compilation using host linker

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

```
nvcc -arch=sm_50 –dc a.cu b.cu
nvcc -arch=sm_50 –dc a.o b.o –o link.o
g++ a.o b.o link.o --library-path=<path> --library=cudart
```