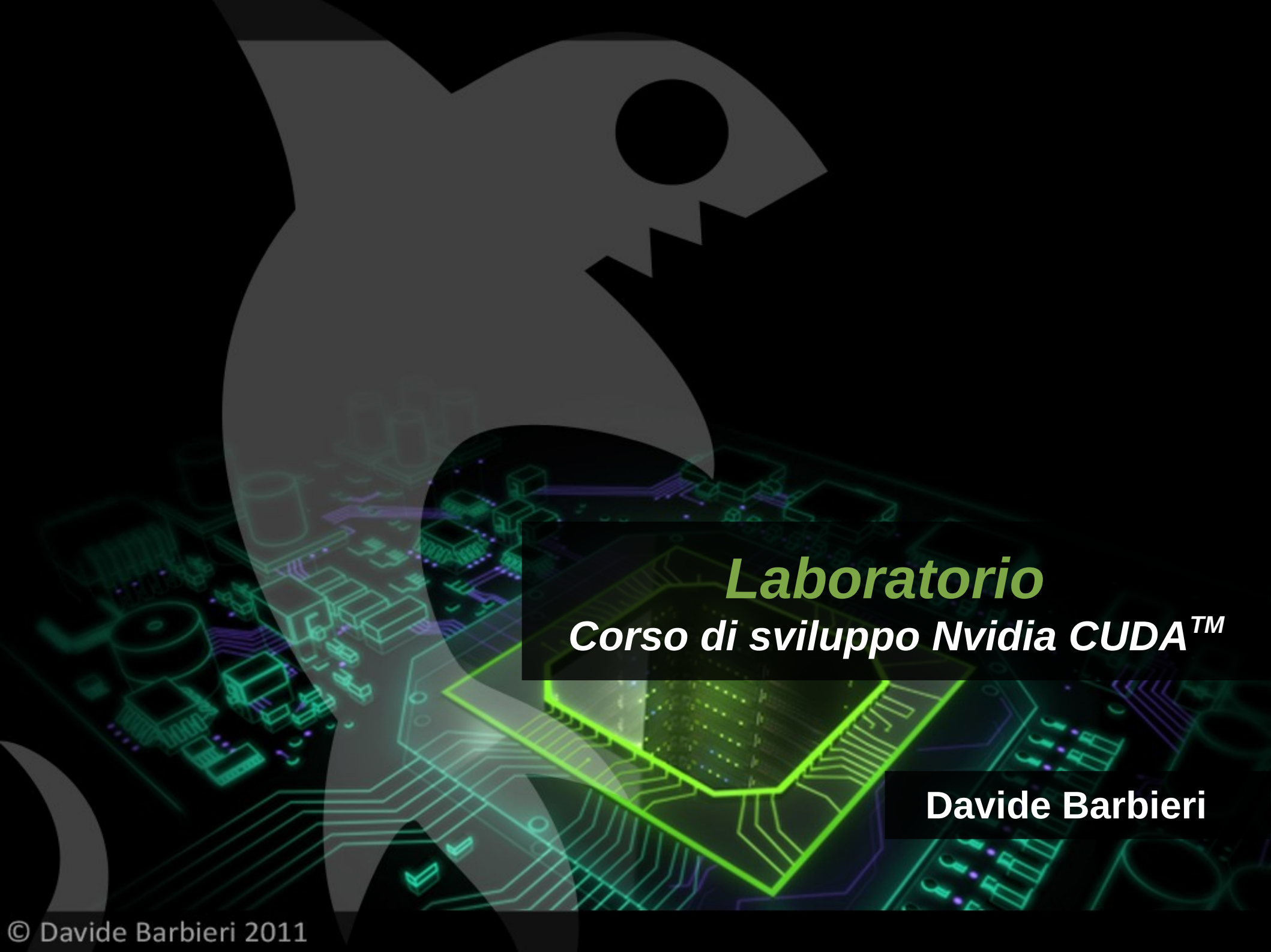


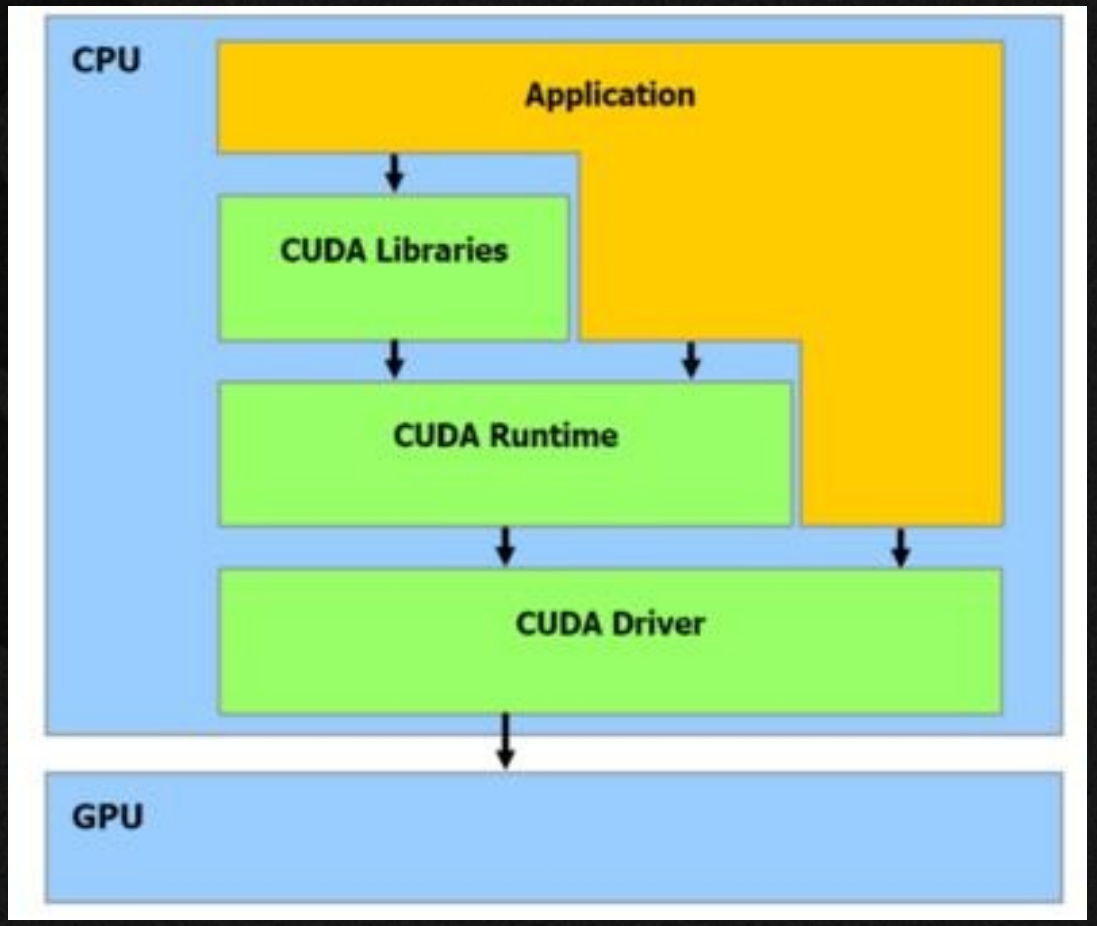
Laboratorio

Corso di sviluppo Nvidia CUDA™

Davide Barbieri

Applicazione CUDA

- **Librerie CUDA**
  - **CUBLAS, CUSPARSE**
- **CUDA Runtime**
  - ***Funzioni cuda*()***
- ***CUDA Driver API***
- **Libreria dinamica *nvcuda* installata con i driver della scheda video**
  - ***Funzioni cu*()***
- ***In questo corso utilizzeremo solo Runtime***

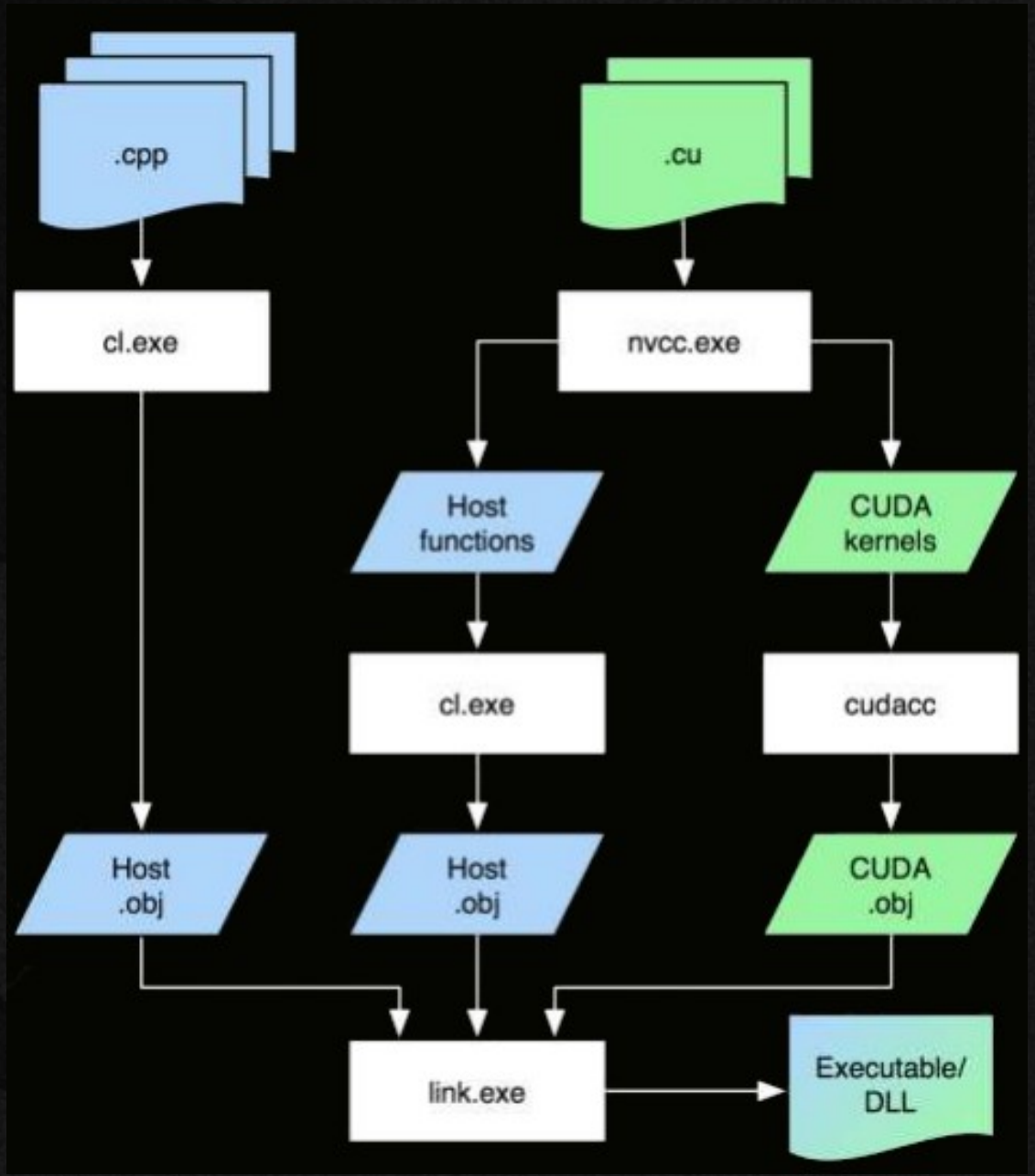


Toolkit

- **nvcc**
  - **Compiler Driver**
- **nvopen64**
  - **Compilatore da CUDA a *ptx***
    - **Versione modificata di AMD open64 (da CUDA 4.1 sarà basato su LLVM)**
- **ptxas**
  - Compilatore da ***ptx*** a ***cubin*** (binario eseguibile da GPU)
- **cuobjdump**
  - Stessa filosofia di *objdump* per il codice CUDA
- **cuda-memcheck**
  - Informazioni a run-time su errori di accesso a memoria
  - Generalmente cudaError ci da “unspecified launch error”

Percorso di compilazione

- *nvcc* divide il codice C/C++ dal codice CUDA
- Il codice C/C++ viene compilato da *cl* (visual studio) o *gcc*
- Il codice CUDA da *nvopen64*



- **cudafe (CUDA front end)**
  - **Opera una serie di preprocessamenti utili ad i passi di compilazione successivi**

- **fatbin**
  - **Produce un eseguibile contenente:**
  - ***cubin***
    - **Uno per architettura specificata in compilazione**
  - ***ptx***
    - **Compatibilità con architetture future**

- **In esecuzione, se non è presente un cubin compatibile**

→ Libreria dinamica contiene un compilatore **jit** per il ***ptx***

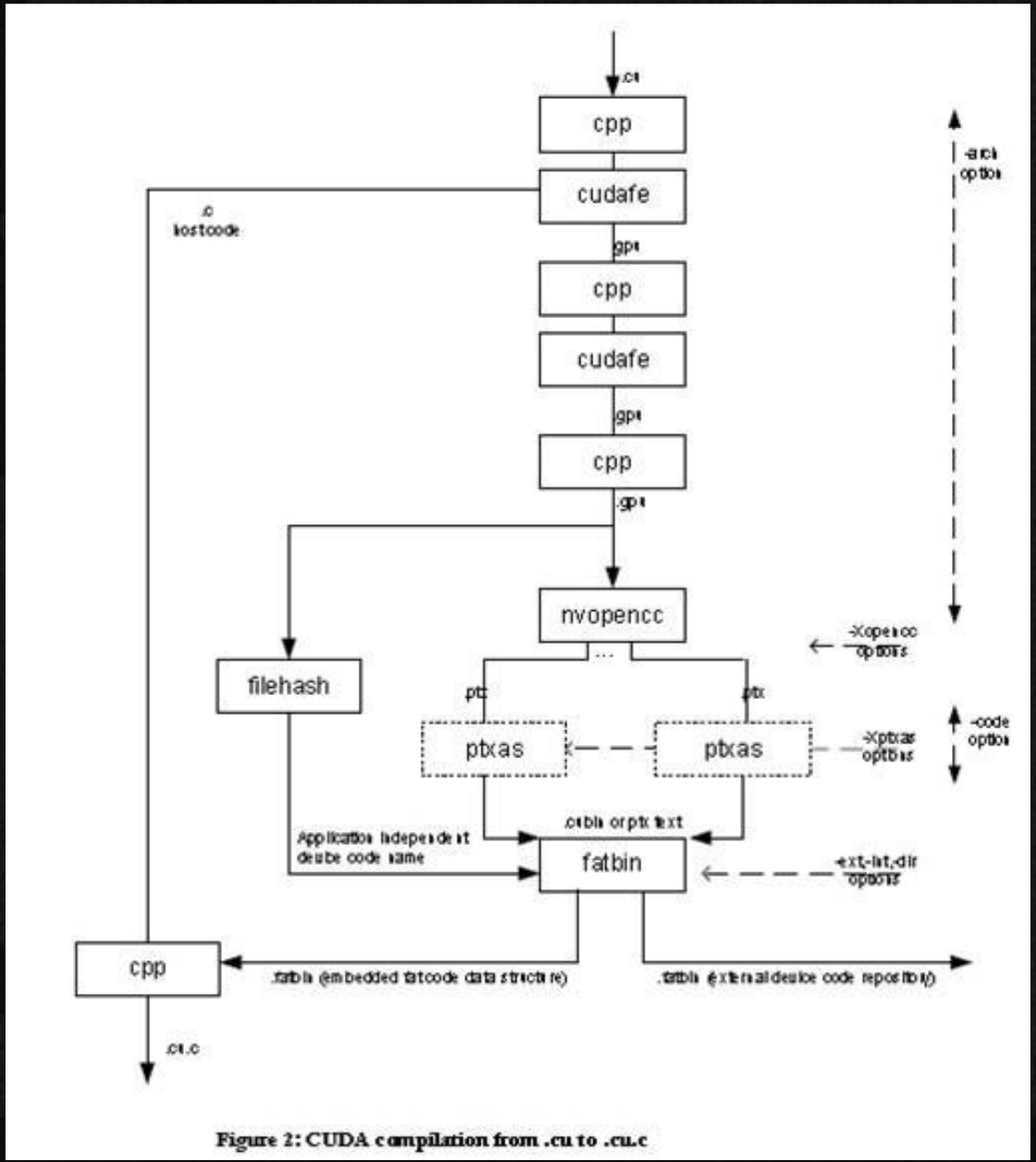


Figure 2: CUDA compilation from .cu to .cu.c

Esercizio 1: hello world

- Somma di due vettori su GPU, C[i] = A[i] + B[i]
- Ogni elemento somma una coppia di elementi
- Funzioni Host da utilizzare:
  - cudaMalloc()
  - cudaMemcpy()
  - cudaFree()
- Funzione Device da realizzare:
  - sumVec<<<dimGriglia, dimBlocco>>>
    - Primo tentativo: 1 thread per blocco
    - Secondo tentativo: 32 thread per blocco

Esercizio 2: id a 2 dimensioni

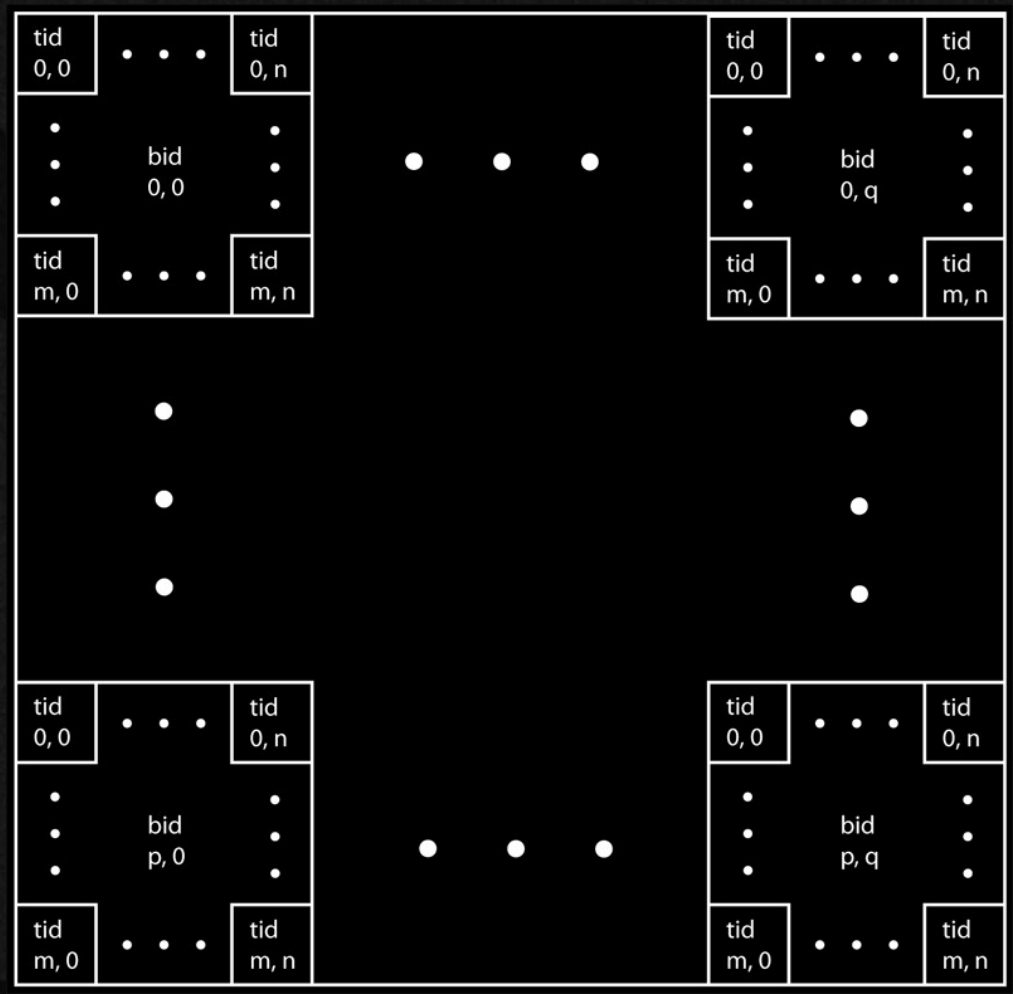
- Somma di due matrici su GPU, C = A + B
- Ogni elemento somma una coppia di elementi
- Utilizzare blocco e griglia a 2 dimensioni:
  - dim3 blockSize(BLOCK_W, BLOCK_H);
    - es. BLOCK_W = BLOCK_H = 16
  - dim3 gridSize(GRID_W, GRID_H);

- Ogni thread deve posizionarsi in modo da effettuare una parte del lavoro:

- int idx = threadIdx.x + blockIdx.x*blockDim.x;
- int idy = threadidx.y + blockIdx.y*blockDim.y;
- int id = idx + idy*pitch;

- Con **pitch** comunemente si identifica la distanza (in byte o #elementi) tra:
  - il primo elemento di una riga
  - il primo elemento della riga successiva

- Spesso pitch != width



Esercizio 3: id che cambia ruolo

- Trasposta di una matrice
  - Per semplicità: matrice quadrata

- Ogni thread:
  - legge un elemento [i,j]
  - scrive nella posizione [j,i]

- Primo tentativo:
  Un elemento per blocco

- Secondo tentativo:
  Blocco di 16*16

Esercizio 4: memoria condivisa

- Trasposta di una matrice
- Nella versione precedente
  - Letture coalizzate
  - Scritture non coalizzate

- Dimensione blocco:
  16*16

- Usiamo memoria condivisa per ottenere anche scritture coalizzate
  __syncthreads()

Esercizio 5: Parallel Reduction

- Somma di elementi di un array: $\sum x[i]$
- Problema detto *parallel reduction*

- È un problema che prende in input un array di valori e da in output un solo valore

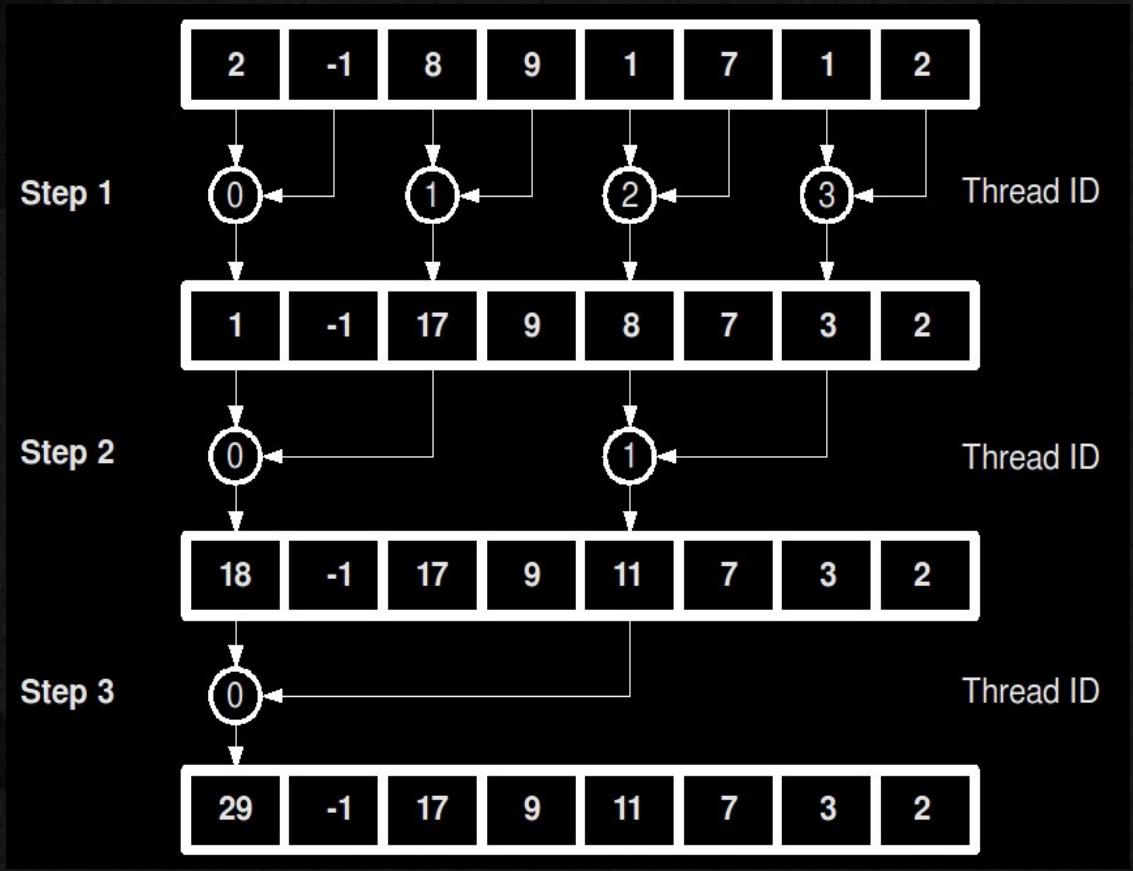
- Ha senso realizzarlo in CUDA?
  - Posso sfruttare la grande banda di memoria della GPU!

- Come lo parallelizziamo?

- Ogni blocco esegue una somma parziale

- Passo 1: Ogni thread esegue una somma
  - Risultato in memoria condivisa
- Passo 2: Metà dei thread del blocco esegue la somma sui risultati precedenti
- Passo 3: Un quarto dei thread esegue la somma sui risultati precedenti

  …
- Un thread salva in memoria globale il risultato parziale
- Rieseguiamo il kernel sul vettore dei risultati parziali
  - oppure eseguiamo le ultime somme su CPU
  - oppure *operazione atomica*



Esercizio 6: Sincronizzazione Warp

- Ha gli stessi identici problemi dell'esercizio precedente

- Riscriviamo l'algoritmo
  - La reduction è la parte più pesante del kernel

- Riduzione a livello di warp, anziché del blocco
  - Un warp è implicitamente sincronizzato (niente __syncthreads())

- Grandezza sia della griglia e del blocco fissi
  - Numero blocchi grande almeno quanto il numero di multiprocessori

Esercizio 7: Funzioni atomiche

- *__ballot()* è una funzione *warp vote* disponibile solo su Fermi
- *unsigned int __ballot(int predicate)*
- Ritorna una bitmask riferita ai thread *attivi* del warp corrente
  - se predicate è diverso da zero → bit uguale ad 1

- Implementare una versione compatibile con c.c. 1.2 e 1.3
  - Utilizzare la funzione

    unsigned int atomicOr(unsigned int address, unsigned int val);*

Esercizio 8: funzioni di vote

- *__syncthreads_and*, *__syncthreads_or*
  sono disponibili solo su Fermi

- Implementiamo delle funzioni equivalenti
  su c.c. 1.2 e 1.3

- Come?
- Utilizzando le funzioni di warp vote
  - *int __all(int predicate)*
  - *int __any(int predicate)*

Esercizio 9: inline ptx

- Esempio di inline ptx per leggere i registri speciali:
  - lane id
    - id del thread nel warp
  - warp id
    - id del warp nel blocco
  - sm id
    - id del multiprocessore attuale
  - grid id
    - id del kernel cui appartiene il thread corrente

Esercizio 10: sezioni critiche e approfondimento compilazione

- Implementare lock() e unlock() per sezioni critiche:
  - solo a livello di griglia di blocchi
- Non è possibile implementare lock a livello di thread (nel blocco)
  - Deadlock sicuro
  - Scopriamo perché disassemblando il codice finale
    - cuobjdump -sass

Esercizio 11: Classi device

- Esempio di programmazione C++ lato device
- Solo su Fermi
- Classe Operation
  - Polimorfismo
  - Add → Somma
  - Sub → Sottrazione