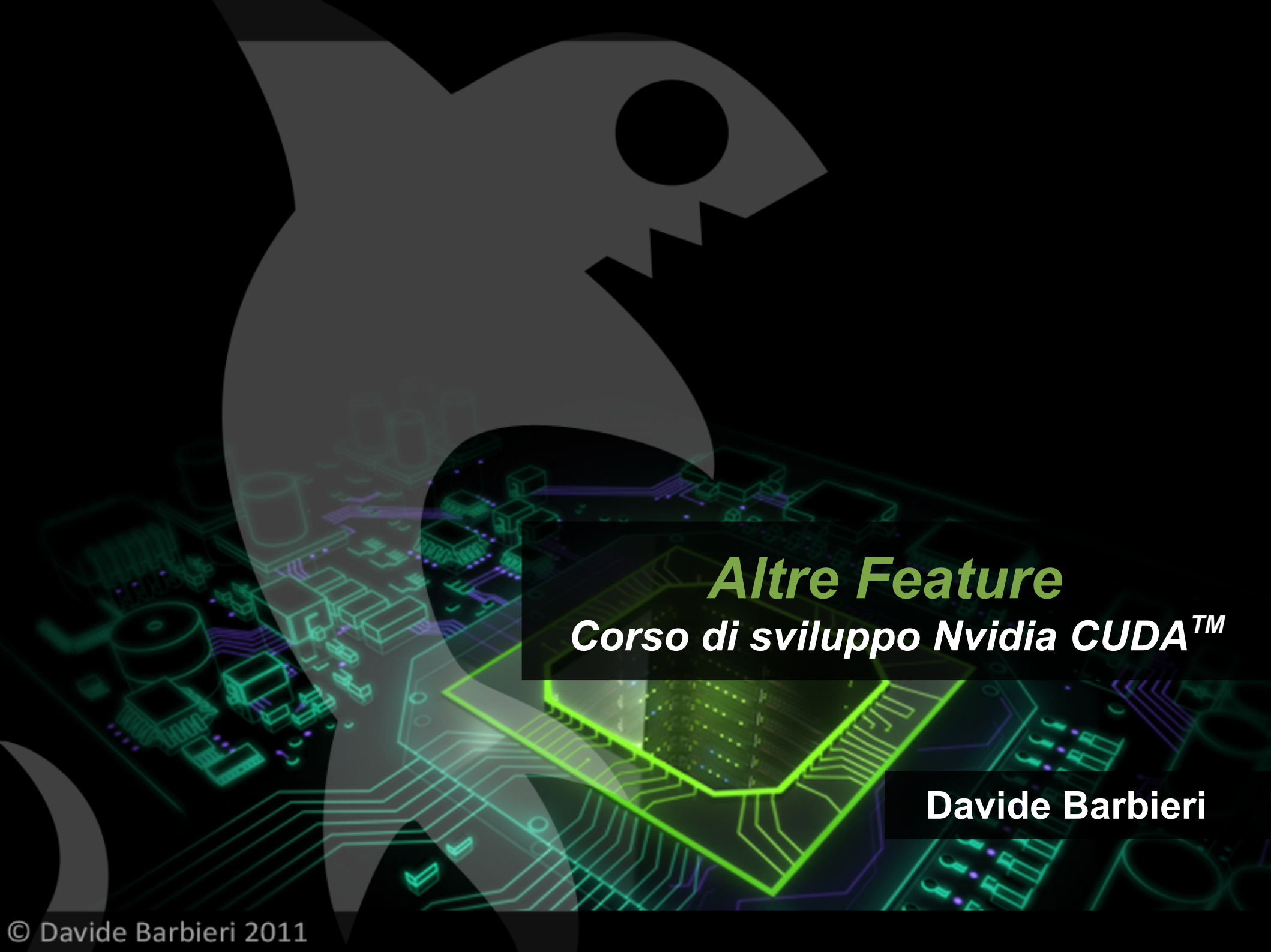

Altre Feature

Corso di sviluppo Nvidia CUDA™

Davide Barbieri

© Davide Barbieri 2011

Panoramica Lezione

- Allocazione dinamica Memoria Condivisa
- Qualificatore *volatile*
- Memoria Texture
- Funzioni di Warp Vote, di Memory Fence e varianti di syncthreads
- Capacità aggiuntive dell'architettura Fermi
- PTX inline

Memoria Condivisa Dinamica

- Abbiamo visto come allocare staticamente memoria condivisa

  __shared__ float shared;

  __shared__ float sMem[SHARED_SIZE];

- È possibile stabilire a *runtime* la dimensione della memoria condivisa da allocare:

```
nomeKernel<<<gridSize,blockSize,sharedMemSize>>>()
```

- Indirizzabile da kernel tramite:

```
extern __shared__ float sMem[];
```

- Ogni dichiarazione di questo tipo parte dallo stesso indirizzo
  - Se volete utilizzare più variabili lo dovete fare “a mano”

```
extern __shared__ float sMem[];

__global__ void nomeKernel()
{
  char* array1 = &sMem[0];
  float* array2 = &sMem[16];
  int* array3 = &sMem[64];
}
```

- Attenzione:
  - all'allineamento di ogni tipo di dato
  - ai conflitti di banco

Qualificatore *volatile*

- Per ordinare letture e scritture in memoria condivisa o globale, thread del blocco si sincronizzano con __syncthreads()
- Se i thread appartengono allo **stesso** warp, però, le loro istruzioni sono già implicitamente sincronizzate (a meno di warp divergenti)

```
__global__ void reductionKrn()
{
  __shared__ float sMem[64];
  int tId = threadIdx.x;
  float res;
  if (tId < 32) sMem[tId] += sMem[tId + 32];
  __syncthreads();
  if (tId < 16) sMem[tId] += sMem[tId + 16];
  __syncthreads();
  if (tId < 8) sMem[tId] += sMem[tId + 8];
  __syncthreads();
  if (tId < 4) sMem[tId] += sMem[tId + 4];
  __syncthreads();
  if (tId < 2) sMem[tId] += sMem[tId + 2];
  __syncthreads();
  if (tId == 0) res = sMem[0] + sMem[1];
}
```

Qualificatore *volatile*

- Purtroppo il compilatore da per *scontato* che se intendiamo scambiare dati tra thread diversi utilizzeremo una primitiva di sincronizzazione (e.g. __syncthreads())

- Altrimenti si sente libero di ottimizzare il codice
  - Evita le scritture ripetute nella stessa allocazione utilizzando un registro temporaneo
  - L'ultima scrittura va in memoria

- Per evitare ciò si usa il qualificatore ***volatile***
  - Diciamo al compilatore che quella allocazione *può essere scritta/letta da altri thread in ogni momento*
  - Ogni modifica dovrà essere effettuata sull'allocazione

Qualificatore *volatile*

- Il nuovo codice sarà:

```
__global__ void reductionKrn()
{
  __shared__ float sMem[64];
  int tId = threadIdx.x;
…
  volatile float* vsMem = sMem;
  float res;

  if (tId < 32) vsMem[tId] += vsMem[tId + 32];
  if (tId < 16) vsMem[tId] += vsMem[tId + 16];
  if (tId < 8) vsMem[tId] += vsMem[tId + 8];
  if (tId < 4) vsMem[tId] += vsMem[tId + 4];
  if (tId < 2) vsMem[tId] += vsMem[tId + 2];
  if (tId == 0) res = vsMem[0] + vsMem[1];
}
```

Memoria Texture

- CUDA conserva ancora alcune proprietà legate alla grafica
  - Texture Cache, da 6 a 8 KB per multiprocessore
- Una *texture* è un'immagine contenente i dati relativi ai pixel da stampare su ogni triangolo di un modello 3D
- Cache ottimizzata per località spaziale 2D
  - I cache hit non riducono la latenza
  - Riducono solamente la richiesta di banda alla RAM GPU
- Potete interfacciare la texture cache ad un'allocazione in RAM GPU
  - Memoria globale
  - CUDA array
    - Allocazione speciale che l'host non può leggere direttamente
    - Interoperabilità con librerie grafiche

Warp Vote Functions

- Supportate solo da c.c. 1.2 in poi:
  - int __all(int predicate);
    - Ritorna 1 se tutti i thread del warp valutano *predicate* diverso da 0
  - int __any(int predicate);
    - Ritorna 1 se almeno un thread del warp valuta *predicate* diverso da 0
- Supportata solo da c.c. 2.0 in poi
  - unsigned int __ballot(int predicate);
    - Ritorna un intero il cui *k-esimo* bit è 1 se il k-esimo thread ha valutato *predicate* diverso da 0

Varianti di __syncthreads()

- Simili ad un __*syncthreads()* unito ad un *vote*
  - Le condizioni vengono valutate su tutto il blocco
  - Non solo sul warp

- Supportate solo da c.c. 2.0 in poi:

- int __syncthreads_count(int predicate);
  - Sincronizza e ritorna il numero di thread con *predicate* diverso da zero

- int __syncthreads_or(int predicate);
  - Sincronizza e ritorna non-zero se almeno un thread valuta *predicate* diverso da zero

- int __syncthreads_and(int predicate);
  - Sincronizza e ritorna non-zero se tutti i thread valutano *predicate* diverso da zero

Memory Fence Functions

- *__threadfence_block()*
  - Il thread aspetta che tutte le scritture in memoria globale o condivisa siano visibili a tutti i thread del blocco
- *__threadfence()*
  - come __threadfence_block(), in più le scritture in memoria globale sono visibili da tutti i thread della griglia
- *__threadfence_system()* (solo Fermi)
  - come __threadfence(), in più le scritture in memoria host mappata sul device è visibile da tutti gli host thread
- Non dev'essere eseguita da tutti i thread
  - Il thread aspetta il completamento di transazioni di memoria
  - Non aspetta altri thread

printf() in Fermi

- Fermi supporta *printf()* su device
- Ogni thread che la incontra invia all'host la richiesta di una printf()
- Utile in Debug per inserire degli assert nei kernel

```
#ifdef _DEBUG

#define deviceAssert(x) \
  if (!(x)) { \
    printf( "Thread (%i,%i,%i) from block (%i,%i,%i) ” \
            “failed assert at %s:%i\n", threadIdx.x, threadIdx.y, threadIdx.z, \
            blockIdx.x, blockIdx.y, blockIdx.z, __FILE__, __LINE__ ); \
    return; }

#else
#define deviceAssert(x)
#endif
```

Stack su Fermi

- Solo Fermi permette la ricorsione
  - Solo su funzioni __device__
- Normalmente una funzione __device__ viene applicata tramite *inlining*
- Lo stack sulla GPU viene utilizzato per la ricorsione
  - Di default, 1KB per thread
  - Allocato in memoria locale
  - È possibile stabilire un'altra dimensione
    - *cudaDeviceGetLimit(size_t* size, cudaLimitStackSize)*
    - *cudaDeviceSetLimit(cudaLimitStackSize, size_t size)*

Heap su Fermi

- malloc() e free() all'interno dei kernel
  - tempo di vita del contesto
- Queste funzioni non sono interscambiabili con cudaMalloc() e cudaFree() lato host
- La grandezza dell'heap di default è 8 MByte
- È possibile stabilire un'altra grandezza prima di ogni chiamata kernel che utilizza malloc()/free()
  - *cudaDeviceGetLimit(size_t* size, cudaLimitMallocHeapSize)*
  - *cudaDeviceSetLimit(cudaLimitMallocHeapSize, size_t size)*
- Il limite non è altro che la quantità di memoria che l'heap "*ruba*" a cudaMalloc()

CUDA C++

- CUDA supporta un sottoinsieme di funzionalità C++ su codice ***device***
  - ***classi***
  - ***template***

- Non supporta:
  - *RTTI*
  - *try{} / catch{}*
  - Libreria Standard C++ (STL & co.)

- Solo Fermi però supporta:
- new/delete (essendo l'unico a supportare un *heap*)
- Polimorfismo (funzioni *virtual*)
  - L'unica architettura a supportare puntatori a funzione

Esempio C++ su Fermi

```
class Operation {
private:
        int opCounter_;
protected:
  __device__ virtual int _op(int* x, int* y) = 0;
public:
  __device__ Operation() : opCounter_(0) { }
  __device__ int operator() (int* a, int* b) {
                ++opCounter_;
                return _op(a,b);
        }

};

class Add : public Operation {
protected:
 __device__ int _op(int* x, int* y) { return *x + *y; }
}

class Sub : public Operation {
protected:
 __device__ int _op(int* x, int* y) { return *x - *y; }
}
```

```
__device__ int res;
__global__ void kernel(int* a, int b)
{
        Add* op1 = new Add();
        Sub op2;

        int t = op1->operator()(a,&b);
        res = op2(&res,&t);
        delete op1;
}
```

PTX inlining

- È possibile inserire all'interno di codice CUDA del codice *PTX*
  - *La sintassi è uguale a quella dell'inline assembly:*

    asm(codeString : input : output : clobbered);

- Trovate la lista delle istruzioni PTX nel documento: **PTX_ISA_X.Y.pdf**

- Anche per l'inline PTX utilizzare *volatile* per dire al compilatore di non rimuovere/spostare il nostro codice

- PTX è un linguaggio intermedio, cui segue un processo di compilazione e ottimizzazione (*ptxas*)
  - Dipende dall'architettura target

PTX inlining

- Il codice PTX viene copiato ed incollato nel codice destinazione PTX
  - è *ptxas* che segnala eventuali errori
- Se inserite codice ptx in una funzione __device__ il codice viene replicato per ogni inlining
  → vengono riscritte anche le dichiarazioni dei registri utilizzati
- Per ovviare al problema:
  - __noinline__
  - Utilizzate le parentesi { } per tutto il codice inline
    - Come dichiarare variabili in uno scope privato in C

```
__device__ int testPtx(int a)
{
    int ret;
    asm("{"
    ".reg .s32 %myInt;\n"
    "mov.s32 %myInt, %1;\n"
    "mov.s32 %0, %myInt;\n"
    "}"
    : "=r"(ret) : "r"(a) : );
    return ret;
}
```

Alternative a CUDA: OpenCL

- OpenCL: Open Computing Language
  - Libreria C99 e linguaggio C con estensioni
- Compilazione solo JIT
  - Niente toolkit
- Molto simile a CUDA
  - thread → work item
  - thread block → work group
  - __global__ → __kernel
  - __shared__ → __local
  - __global per puntatori a memoria globale
- Khronos Group rilascia le specifiche
- Ogni produttore le implementa
  - Nvidia, AMD (Cpu e Gpu), Intel, ARM, VIA, IBM
  - I diritti appartengono ad Apple Inc.

Alternative a CUDA: Ocelot