

GPU Computing

Laurea Magistrale in Informatica - AA 2019/20

Docente **G. Grossi**

Lezione 7 – Stream e concorrenza

Sommario

- ✓ Gradi di concorrenza in CUDA
- ✓ CUDA stream: creazione, uso e sincronizzazione
- ✓ Uso tipico: sovrapporre trasferimento dati ed esecuzione kernel
- ✓ CUDA eventi: creazione, uso e sincronizzazione
- ✓ Misura del tempo mediante eventi
- ✓ Esercitazione su stream ed eventi

Gradi di concorrenza in CUDA

- ✓ **CPU/GPU concurrency:** poiché sono dispositivi distinti, la CPU e la GPU possono operare indipendentemente una dall'altra
- ✓ **Memcpy/kernel processing concurrency:** grazie al DMA il trasferimento dati tra host e device può avere luogo mentre gli SMs processano i kernel
- ✓ **Kernel concurrency:** dalla classe SM 3.x l'hardware è in grado di eseguire fino a 32 kernel in parallelo anche da thread di CPU distinti
- ✓ **Grid-level-concurrency:** uso di stream multipli per operazioni indipendenti
- ✓ **Multi-GPU concurrency:** problemi con elevata richiesta di risorse computazionali possono ripartire il carico tra diverse GPU che operano in parallelo (multi-GPU programming)

Asynchronous Commands in CUDA

As described by the CUDA C Programming Guide, **asynchronous commands** return control to the calling host thread before the device has finished the requested task (they are **non-blocking**)

These commands are :

- **Kernel launches**
- **Memory copies** between two addresses to the same **device** memory
- **Memory copies** from host to device of a memory block of **64 KB** or less
- **Memory copies** performed by functions with the **Async** suffix
- **Memory set function** calls with the **Async** suffix

Sincronismo nel codice GPU vs CPU

In GPU:

- ✓ A livello di **warp** l'esecuzione è effettivamente **sincrona**
- ✓ **Warp diversi** eseguono con arbitraria sovrapposizione (con **syncthreads** si può assicurare il corretto comportamento all'interno di un blocco di thread)
- ✓ **Blocchi di thread distinti** eseguono in modo arbitrario e sovrapposto

In CPU:

- ✓ Molte CUDA call sono **sincrone** (bloccanti)
- ✓ Il lancio di kernel è **asincrono** (non bloccante) così come le copie asincrone (**cudaMemcpyAsync**)
- ✓ In entrambi i casi si forza la sincronizzazione dell'host con **cudaDeviceSynchronize**
- ✓ Vantaggio è la sovrapposizione di operazioni pur preservando la correttezza computazionale

CUDA Stream

- ✓ Uno stream CUDA è riferito a **sequenze di operazioni CUDA asincrone** che vengono eseguite sul device nell'ordine stabilito dal codice host
- ✓ lo stream **incapsula** le operazioni, ne mantiene **l'ordine FIFO**, e ne **indaga** lo **status**
- ✓ Le operazioni tipiche includono: **trasferimento** dati host-device, **lancio** di kernel, gestione di **eventi**, assieme ad altre operazioni messe in atto dall'host verso il device
- ✓ L'esecuzione di operazioni in uno stream è sempre **asincrona** rispetto all'host
- ✓ Le operazioni appartenenti a **stream distinti** non hanno restrizioni vicendevoli sull'ordine di esecuzione, sono **indipendenti**
- ✓ **Grid-level parallelism:** dal punto di vista CUDA le operazioni di stream distinti vengono eseguite in **parallelo**

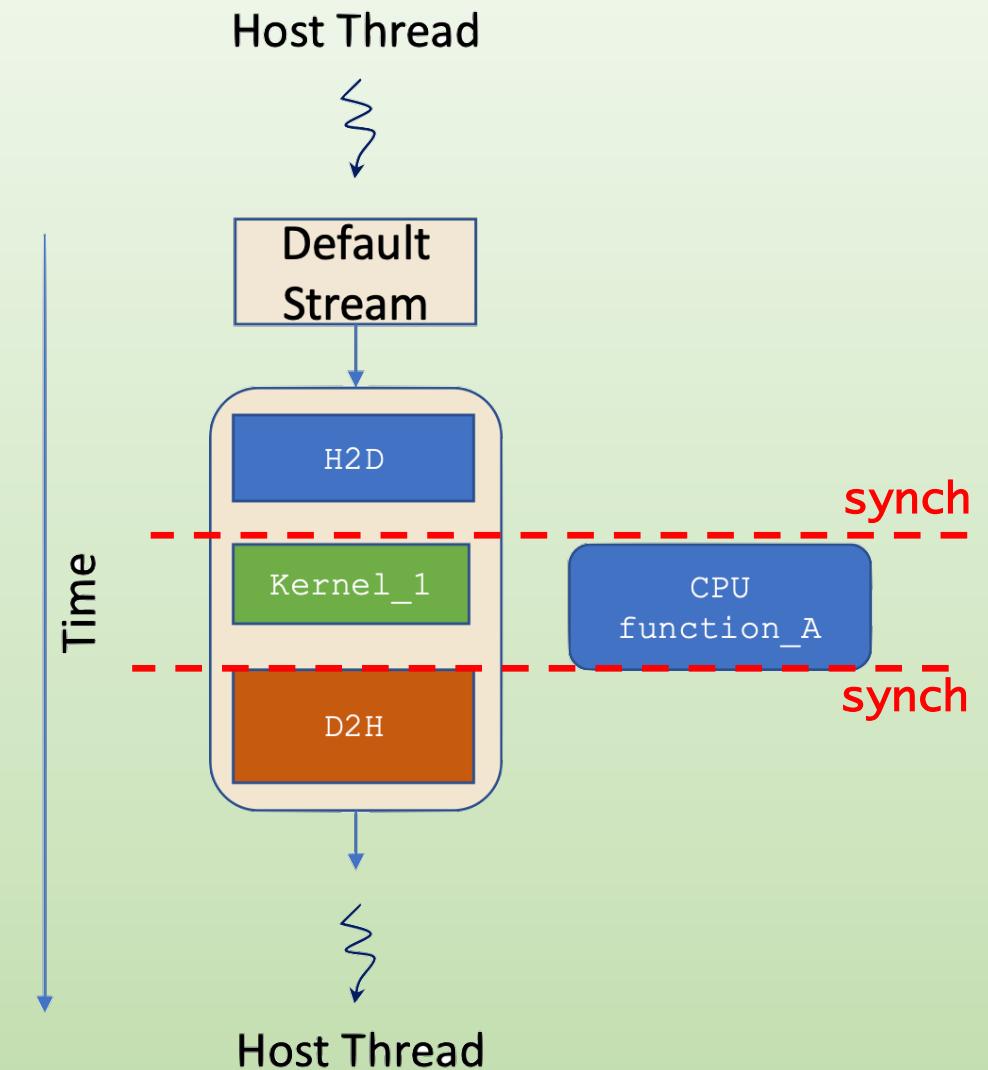
Tipi di stream

- ✓ Le operazioni CUDA vengono eseguite **esplicitamente o implicitamente** in uno stream
- ✓ Esistono **2 tipi** di stream:
 - Dichiarato **implicitamente** (**NULL stream o default stream**)
 - Dichiarato **esplicitamente** (**non-NUL stream**)
- ✓ Il **default** stream interviene quando **non** si usa **esplicitamente** uno stream
- ✓ Meglio **usare non-NUL stream** (stream asincroni concorrenti) per:
 - Sovrapporre articolate computazioni host e device
 - Sovrapporre computazioni host e trasferimento dati host-device
 - Sovrapporre trasferimento dati host-device e computazioni device
 - Computazioni concorrenti su device

Default stream

Sovrapposizione: **computazione host e device**:

```
cudaMemcpy(d_a, a, numBytes, H2D);
kernel_1<<<blocks,threads>>>(d_a)
function_A(b)
cudaMemcpy(a, d_a, numBytes, D2H);
```



Default stream (NULL stream)

Esempio d'uso del default stream: **prospettiva host e device**

GPU:
Vede le tre OP
in sequenza e
le esegue

1. `cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);`
2. `GPU_kernel<<<blocks,threads>>>(d_a)`
3. `cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);`

CPU:
OP 1. è bloccante,
quando terminata
passa al kernel (non
bloccante) e si mette
in attesa della
terminazione prima di
eseguire OP 3.

Sovrapposizione: **computazione host e device:**

GPU:
Come prima...
non "vede"
differenze

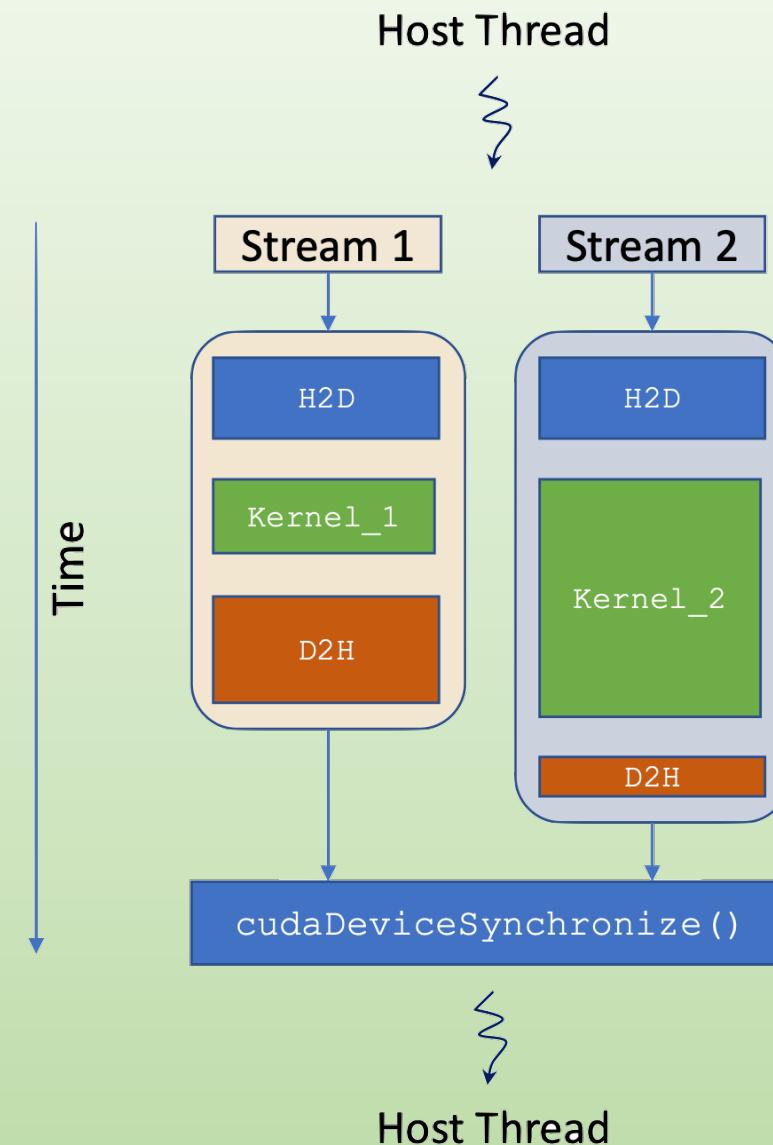
1. `cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);`
2. `GPU_kernel<<<blocks,threads>>>(d_a)`
3. `CPU_function(b)`
4. `cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);`

CPU:
Terminata OP 1. e non
appena lanciato il kernel
esegue OP 3
simultaneamente.
OP 4 viene eseguita
quando CPU e GPU
hanno terminato
entrambi il loro lavoro

CUDA stream

CUDA Programming Guide:

- ✓ Applications manage **concurrency** through streams
- ✓ A **stream** is a **sequence** of commands (possibly issued by different host threads) that execute in order
- ✓ **Different streams**, on the other hand, may execute their commands out of order with respect to one another or **concurrently**



CUDA API stream create

- ✓ **Creazione** di uno stream non nullo

Prototipo

```
cudaError_t cudaStreamCreate(cudaStream_t* pStream);
```

- Crea esplicitamente uno stream

- ✓ **Lancio** del kernel

Prototipo

```
kernel_name<<< grid, block, sharedMemSize, pStream >>>(argument list);
```

- Lancio del kernel sullo stream specifico (4° argomento della tripla parentesi)

- ✓ **Eliminazione** di stream:

Prototipo

```
cudaError_t cudaStreamDestroy(cudaStream_t pStream);
```

- se eseguito prima della terminazione delle op nello stream, ritorna il controllo ma il lavoro dello stream viene completato (sincronizzazione con host)

CUDA API stream Async-copy

- ✓ **Allocazione** spazio su pinned memory

Prototipo

```
cudaError_t cudaMallocHost(void **ptr, size_t size);  
cudaError_t cudaHostAlloc(void **pHost, size_t size, unsigned int flags);
```

- Alloca su host memoria non paginabile (pinned memory), se flag = 0 il comportamento delle due API è uguale, altrimenti...

- ✓ **Trasferimento** asincrono basato su pinned memory

Prototipo

```
cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count,  
                           cudaMemcpyKind kind, cudaStream_t stream);
```

- Copia asincrona dei dati host->device

CUDA API stream synchronize

- ✓ Poiché tutte le operazioni sono asincrone le seguenti funzioni permettono di **controllare** se tutte le **operazioni** in uno **stream** sono state completate oppure no
- ✓ **Blocco** dell'host sullo stream

Prototipo

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
```

- forza il blocco dell'host fino a che tutte le operazioni nello stream sono state completate

Nota: `cudaDeviceSynchronize()` blocca l'host fino a che **tutti i comandi su tutti gli stream** sono stati completati

- ✓ **Controllo** stream completato

Prototipo

```
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

- controlla se op sono completate ma non forza blocco dell'host in caso negativo.
- Ritorna `cudaSuccess` se op complete o `cudaErrorNotReady` altrimenti

Uso in pratica: gestione stream

- ✓ Una operazione è inviata allo **stream di default** senza specificarlo (o specificando lo **stream 0**) :

Definizione
equivalente
di **default stream**

```
kernel<<< blocks, threads, SMEM_bytes >>>(); // default stream
kernel<<< blocks, threads, SMEM_bytes, 0 >>>(); // 0 = default stream
```

Creazione di una batteria
stream non-NUL

```
// creazione degli stream asincroni non-NUL
cudaStream_t stream[nStreams];
for (int i = 0; i < nStreams; ++i) {
    checkCuda(cudaStreamCreate(&stream[i]));
}
```

**Elaborazione multi
stream...**

```
...  
// elaborazione asincrona inviata ai diversi stream
```

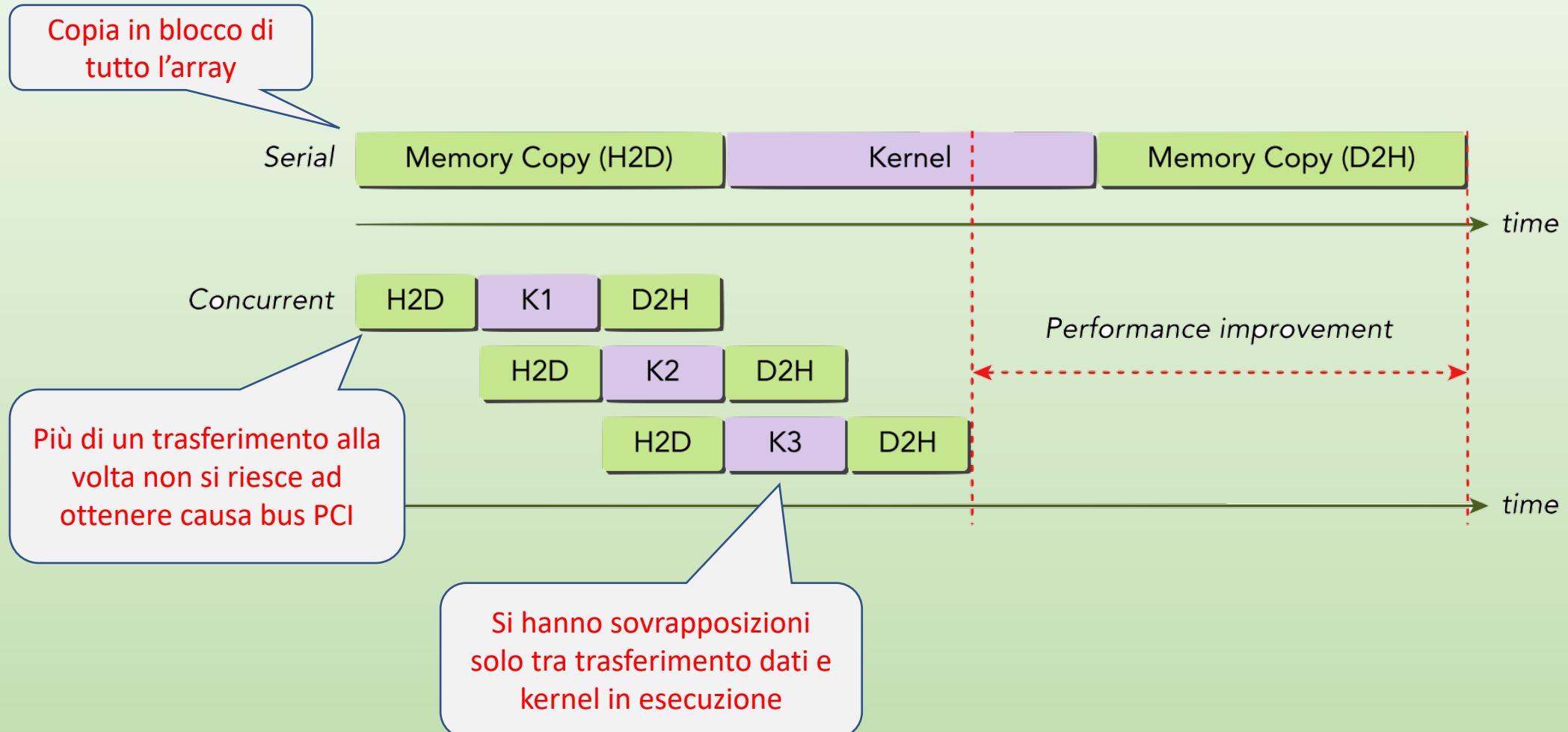
Eliminazione di stream
non-NUL

```
...  
// cleanup finale
for (int i = 0; i < nStreams; ++i)
    checkCuda(cudaStreamDestroy(stream[i]));
```

Sovrapporre kernel e trasferimento dati

- ✓ Devono essere verificati diversi requisiti perché si possa effettuare questa sovrapposizione
 1. Il device deve essere capace di “**concurrent copy and execution**”
 2. questo può essere indagato con il campo **deviceOverlap** della struct **cudaDeviceProp**
 3. Tutti i device con **compute capability (>= 1.1)** hanno questa capacità
 4. Il kernel e trasferimento dati devono appartenere a **differenti non-default stream**
 5. La host memory coinvolta nel trasferimento deve essere **pinned memory**
- ✓ **ES. di uso tipico:**
 1. se abbiamo un array di **N** (grande) elementi lo spezziamo in gruppi da **M** elementi
 2. Se il kernel opera indipendentemente su tutti gli elementi ogni gruppo può essere elaborato a parte
 3. Il numero di stream (non-NULL) usati è: **nStreams = N/M**

Uso in pratica: sovrapposizione



Uso in pratica: codice per sovrapposizione

Copia un blocco alla volta

esegue kernel alla fine della copia H-D

esegue copia D->H alla fine della kernel

sincronizza su fine degli stream

```
for (int i = 0; i < nStreams; i++) {  
    int offset = i * bytesPerStream;  
    cudaMemcpyAsync(&d_a[offset], &a[offset], bytePerStream,  
                   cudaMemcpyHostToDevice, streams[i]);  
    kernel<<< M/blockSize, blockSize, 0, stream[i] >>>(d_a,offset);  
    cudaMemcpyAsync(&a[offset], &d_a[offset], bytesPerStream,  
                   cudaMemcpyDeviceToHost, streams[i]);  
}  
for (int i = 0; i < nStreams; i++) {  
    cudaStreamSynchronize(streams[i]);  
}
```

Somma di vettori con stream

kernel

allocazione di pinned memory

```
__global__ void sumArrays(float *A, float *B, float *C, const int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N) {
        for (int i = 0; i < N; ++i) {
            C[idx] = A[idx] + B[idx];
        }
    }
}

...
// malloc pinned host memory for async memcpy
float *h_A, *h_B, *hostRef, *gpuRef;
CHECK(cudaHostAlloc((void**) &h_A, nBytes, cudaHostAllocDefault));
CHECK(cudaHostAlloc((void**) &h_B, nBytes, cudaHostAllocDefault));
CHECK(cudaHostAlloc((void**) &gpuRef, nBytes, cudaHostAllocDefault));
CHECK(cudaHostAlloc((void**) &hostRef, nBytes, cudaHostAllocDefault));
```

Somma di vettori con stream

allocazione di devicememory

un solo kernel

suddivisione in chunk

creazione di NSTREAM

```
// malloc device global memory
float *d_A, *d_B, *d_C;
CHECK(cudaMalloc((float**) &d_A, nBytes));
CHECK(cudaMalloc((float**) &d_B, nBytes));
CHECK(cudaMalloc((float**) &d_C, nBytes));

. . .

// kernel
sumArrays<<<grid, block>>>(d_A, d_B, d_C, nElem);

. . .

// grid parallel operation
int iElem = nElem / NSTREAM;
size_t iBytes = iElem * sizeof(float);
grid.x = (iElem + block.x - 1) / block.x;
cudaStream_t stream[NSTREAM];
for (int i = 0; i < NSTREAM; ++i) {
    CHECK(cudaStreamCreate(&stream[i]));
}
```

Somma di vettori con stream

```
// initiate all work on the device asynchronously in depth-first order

for (int i = 0; i < NSTREAM; ++i) {

    int ioffset = i * iElem;

    CHECK(cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes, cudaMemcpyHostToDevice,
                         stream[i]));

    CHECK(cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes, cudaMemcpyHostToDevice,
                         stream[i]));

    sumArrays<<<grid, block, 0, stream[i]>>>(&d_A[ioffset], &d_B[ioffset], &d_C[ioffset],
                                                iElem);

    CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d_C[ioffset], iBytes, cudaMemcpyDeviceToHost,
                         stream[i]));

}
```

Risultati

```
> Using Device 0: Tesla P100-PCIE-16GB
> Compute Capability 6.0 hardware with 56 multi-processors
> vector size = 262144
> grid (2048, 1) block (128, 1)
```

Measured timings (throughput):

Memcpy host to device	:	0.220384 ms (4.757950 GB/s)
Memcpy device to host	:	0.093664 ms (11.195081 GB/s)
Kernel	:	428.958008 ms (0.004889 GB/s)
Total	:	429.272064 ms (0.004885 GB/s)

Actual results from overlapped data transfers:

overlap with 4 streams	:	91.028259 ms (0.023038 GB/s)
speedup	:	78.794739

Default Stream prima di CUDA 7

- ✓ The default stream is useful where concurrency is not crucial to performance
- ✓ **Before CUDA 7:** each device has a **single default stream used for all host threads**, which causes **implicit synchronization**
- ✓ “**Implicit Synchronization**”: (As the section in the CUDA C Programming Guide explains) two commands from different streams cannot run concurrently if the host thread issues any CUDA command to the default stream between them

Sincronizzazione rispetto NULL-stream

- ✓ Un **NULL-stream** blocca tutte le precedenti operazioni dell'host con la sola eccezione del lancio kernel
- ✓ Anche se i **non-NUL stream** sono non-bloccanti rispetto all'host, possono essere **sincroni** o **asincroni** rispetto al **NULL-stream**
- ✓ Per questo gli stream **Non-NULL** possono essere di due tipi:
 - ✓ **Blocking** stream: lo stream NULL è bloccante
 - ✓ **Non-blocking** stream: lo stream NULL non è bloccante
- ✓ Gli stream creati usando **cudaStreamCreate** sono bloccanti: l'esecuzione di operazioni in questi stream vengono bloccate in attesa del completamento di operazioni dello stream NULL
- ✓ Il NULL stream è implicitamente definito e sincronizza con tutti gli altri stream bloccanti nello stesso contesto CUDA
- ✓ In generale il **NULL stream non si sovrappone** con nessun altro stream bloccante

Esempio

```
kernel_1<<<1, 1, 0, stream_1>>>();  
kernel_2<<<1, 1>>>();  
kernel_3<<<1, 1, 0, stream_2>>>();
```

- **kernel_2** non parte finché **kernel_1** sia stato completato
- **kernel_3** non parte finché **kernel_2** sia stato completato
- Dal punto di vista dell'host ogni kernel è asincrono e non bloccante

- ✓ CUDA runtime permette di definire il **comportamento** di uno **stream non-NULL** in relazione al NULL stream:

Prototipo

```
cudaError_t cudaStreamCreateWithFlags(cudaStream_t* pStream,  
                                      unsigned int flags)
```

- Flag specifica se null stream è bloccante o no rispetto agli altri

cudaStreamDefault

default stream creation flag (**blocking**)

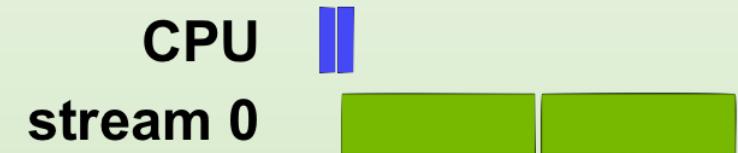
cudaStreamNonBlocking

asynchronous stream creation flag (**non-blocking**)

Concorrenza tra stream

- ✓ Assumendo che kernel usi meno del 50% della GPU...
- ✓ Comportamento dello stream NULL di default

```
kernel<<<blocks,threads>>>();  
kernel<<<blocks,threads>>>();
```



- ✓ Comportamento dello stream NULL e non-NULL

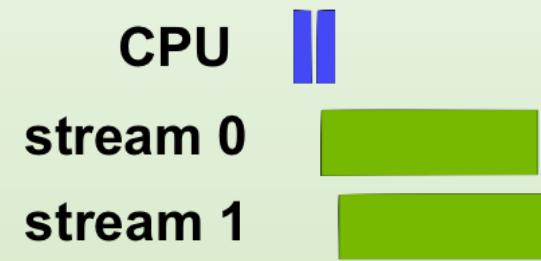
```
cudaStream_t stream1;  
cudaStreamCreate(&stream1);  
kernel<<<blocks,threads>>>();  
kernel<<<blocks,threads,0,stream1>>>();  
cudaStreamDestroy(stream1);
```



Concorrenza tra stream

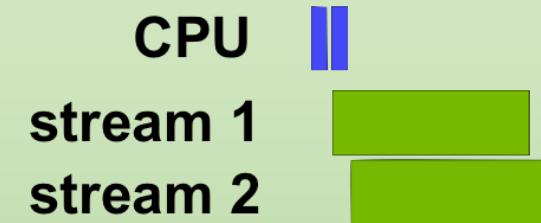
- ✓ Comportamento dello stream NULL di default

```
cudaStream_t stream1;  
cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);  
kernel<<<blocks, threads>>>();  
kernel<<<blocks, threads, 0, stream1>>>();  
cudaStreamDestroy(stream1);
```



- ✓ Comportamento di stream non-NULL

```
cudaStream_t stream1, stream2;  
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
kernel<<<blocks, threads, 0, stream1>>>();  
kernel<<<blocks, threads, 0, stream2>>>();  
cudaStreamDestroy(stream1);  
cudaStreamDestroy(stream2);
```



... e dopo CUDA 7

- ✓ Introduces the new option the **per-thread** default stream that has two effects:
 - it gives **each host thread** its own **default stream**
 - commands issued to the default stream by different host threads can run concurrently
 - these **default streams** are **regular streams**
 - commands in the default stream may run concurrently with commands in non-default streams
- ✓ To enable per-thread default streams in CUDA 7 and later, you can either
 - compile with the nvcc command-line option **--default-stream per-thread**
 - **#define CUDA_API_PER_THREAD_DEFAULT_STREAM** preprocessor macro before including CUDA headers (cuda.h or cuda_runtime.h).
- ✓ It is important to note: you cannot use **#define CUDA_API_PER_THREAD_DEFAULT_STREAM** to enable this behavior in a .cu file when the code is compiled by nvcc because nvcc implicitly includes cuda_runtime.h at the top of the translation unit.

Pre e post CUDA 7

- ✓ Lo stream di default è utile quando la concorrenza non è cruciale per le performance
- ✓ Prima di **CUDA 7**, ogni device ha un **singolo** default stream usato per tutti i thread dell'host che causano sincronizzazione implicita
- ✓ CUDA 7 introduce la nuova opzione ***per-thread default stream***, che ha due effetti:
 1. Assegna a ogni **thread** dell'host il **proprio** default stream (comandi inviati al default stream da diversi thread dell'host possono eseguire concorrentemente)
 2. I default stream sono **stream regolari** (comandi nel default stream possono eseguire in concorrenza con quelli in un non-default stream)
- ✓ Per abilitare per-thread default stream compilare con **nvcc** command-line option **--default-stream per-thread**, o definire la macro per il preprocessore
#define CUDA_API_PER_THREAD_DEFAULT_STREAM

Esempio multi-stream

kernel

```
__global__ void kernel(float *x, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    for (int i = tid; i < n; i += blockDim.x * gridDim.x) {
        x[i] = sqrt(pow(2,i));
    }
}
```

main

```
...
cudaStream_t streams[num_streams];
float *data[num_streams];

for (int i = 0; i < num_streams; i++) {
    cudaStreamCreate(&streams[i]);

    cudaMalloc(&data[i], N * sizeof(float));

    // launch one worker kernel per stream
    kernel<<<1, 64, 0, streams[i]>>>(data[i], N);

    // launch a dummy kernel on the default stream
    kernel<<<1, 1>>>(0, 0);
}
```

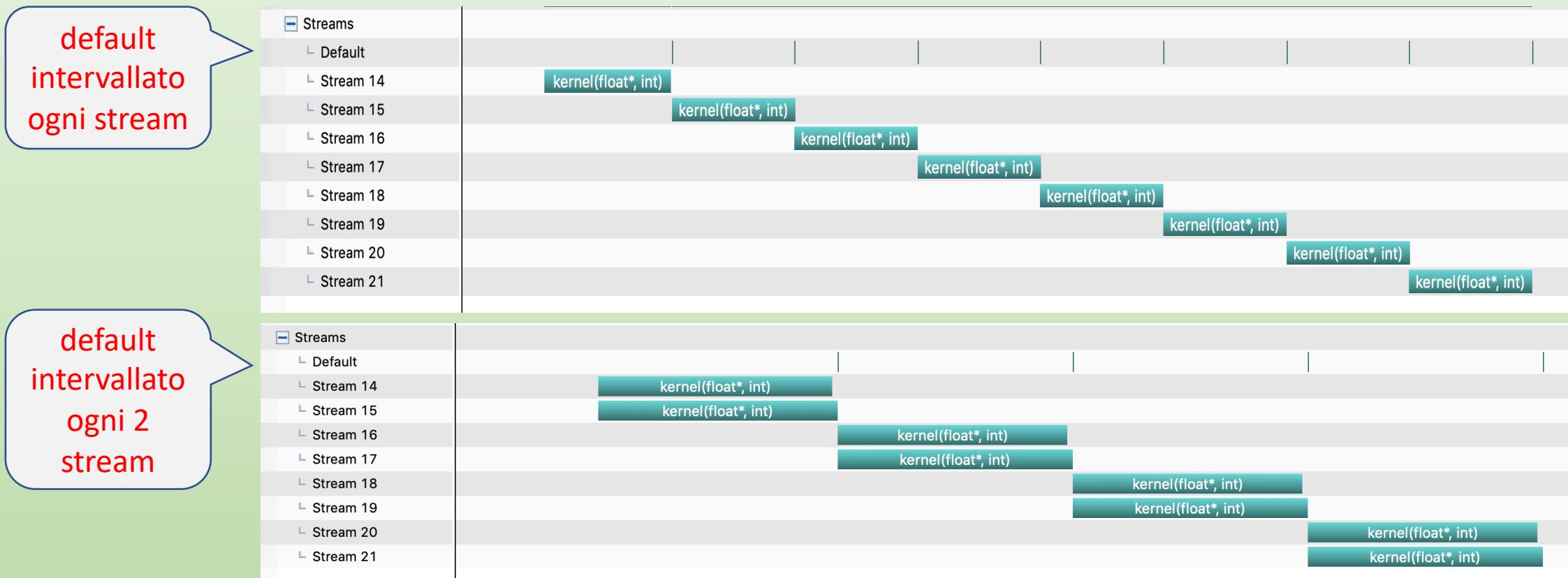
lancio del kernel
su ogni stream

Dummy kernel
sul default
stream

Esempio multi-stream (profile)

- ✓ Eempio multi-stream in cui non si ha concorrenza tra qualsiasi kernel inviato a stream non default con quello mandato sul default stream

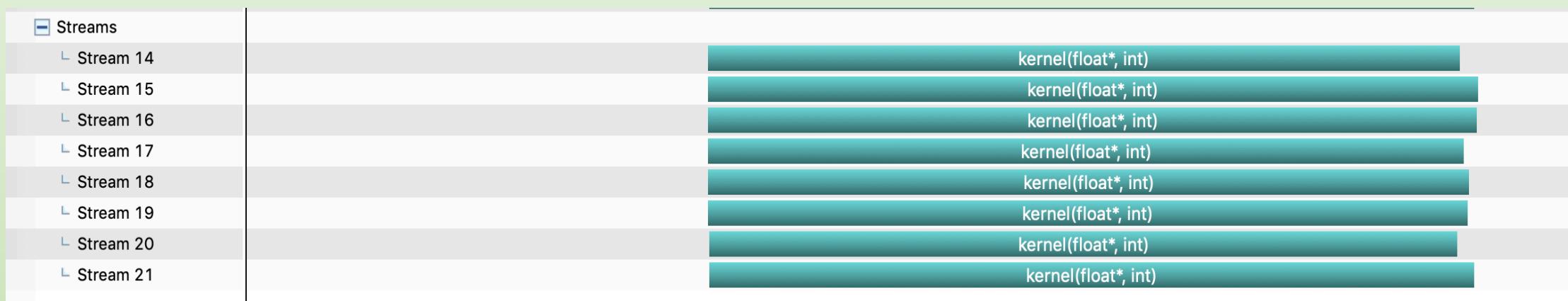
```
nvcc ./ese_stream.cu -o ese_stream
```



Senza default stream (profile)

Without the dummy kernel on the default stream

```
nvcc ./ ese_stream.cu -o ese_stream
```

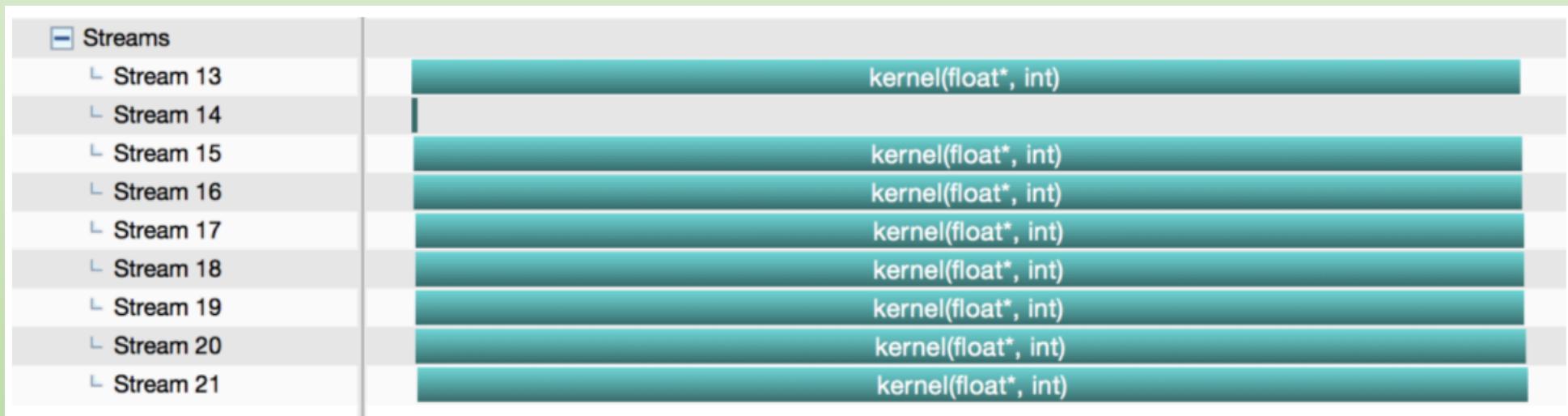


New per-thread default stream

Here you can see full concurrency between nine streams:

- the default stream, which in this case maps to Stream 14,
- eight other streams we created
- Note that the dummy kernels run so quickly that it's hard to see that there are eight calls on the default stream in this image.

```
nvcc --default-stream per-thread./ ese_stream.cu -o ese_stream
```



Esempio multi-threading (CPU)

```
#include <pthread.h>
#include <stdio.h>

const int N = 1 << 20;

__global__ void kernel(float *x, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    for (int i = tid; i < n; i += blockDim.x * gridDim.x)
        x[i] = sqrt(pow(2,i));
}

void *launch(void *dummy) {
    float *data;
    cudaMalloc(&data, N * sizeof(float));
    kernel<<<1, 64>>>(data, N);
    cudaStreamSynchronize(0);
    return NULL;
}
```

main lancia 8 pthread e
ognuno 1 kernel... 8
kernel in totale

main con
pthread

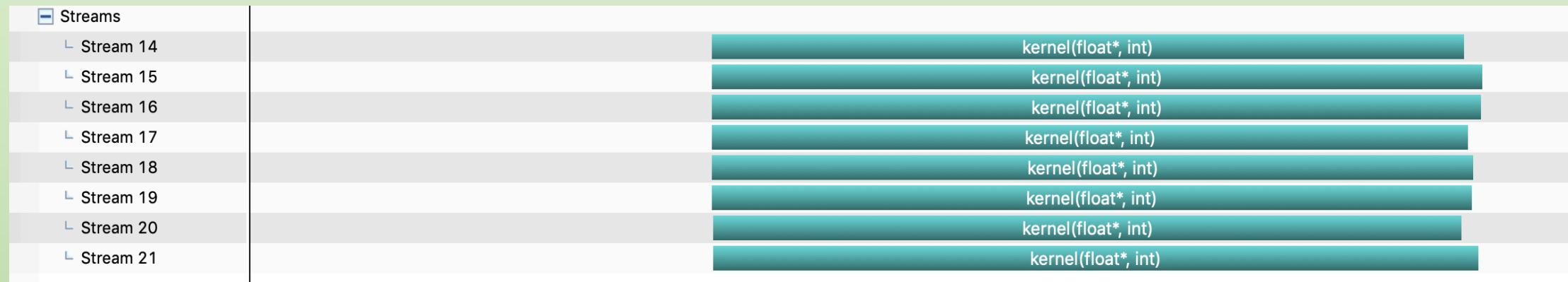
```
int main() {
    const int num_threads = 8;
    pthread_t threads[num_threads];
    for (int i = 0; i < num_threads; i++) {
        if (pthread_create(&threads[i], NULL, launch, 0)) {
            fprintf(stderr, "Error creating thread\n");
            return 1;
        }
    }

    for (int i = 0; i < num_threads; i++) {
        if(pthread_join(threads[i], NULL)) {
            fprintf(stderr, "Error joining thread\n");
            return 2;
        }
    }
    . .
}
```

Per-thread default stream option

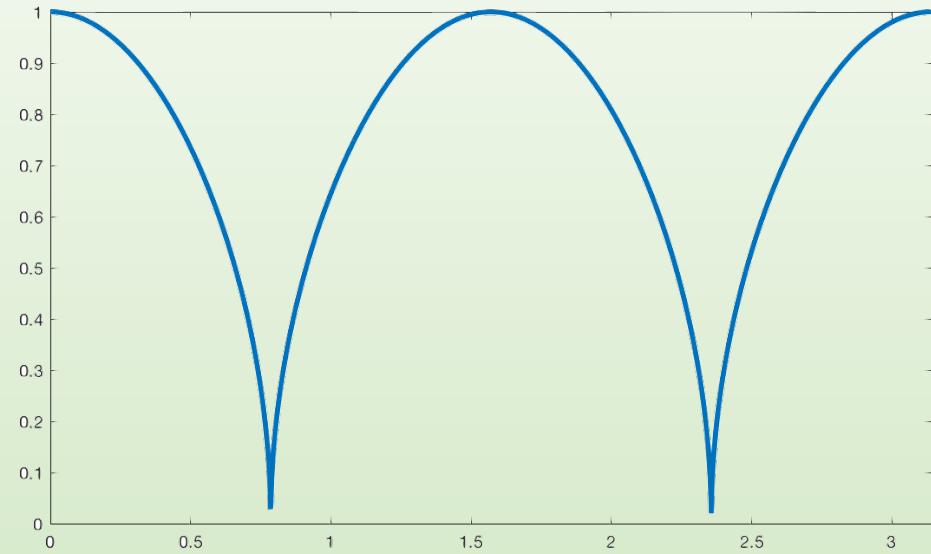
- ✓ Multi-stream example using the new per-thread default stream option, which enables fully concurrent execution

```
nvcc --default-stream per-thread ./ ese_stream.cu -o ese_stream
```



Esercitazione

```
__global__ void tabular(float *a, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) {
        float x = a[i];
        float s = sinf(x);
        float c = cosf(x);
        a[i] = sqrtf(abs(s * s - c * c));
    }
}
```



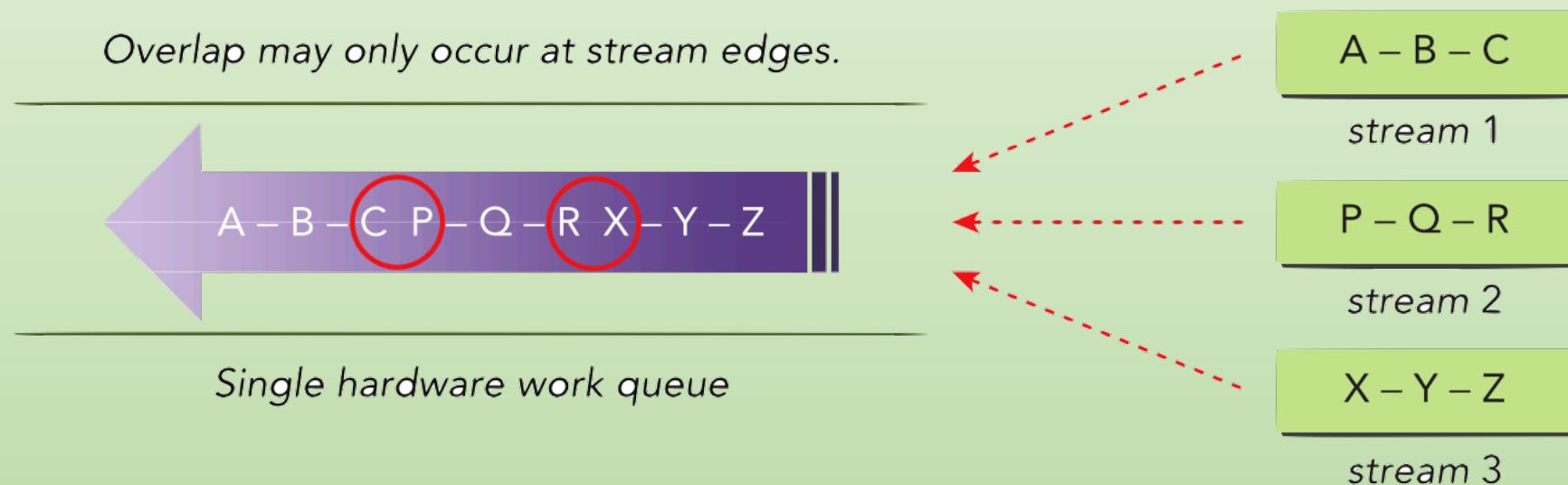
Tabulazione funzione trigonometrica:

Disegnare un kernel con stream per la tabulazione della funzione $f(x) = \sqrt{|\sin(x)^2 - \cos(x)^2|}$

- Estendere l'analisi all'intervallo $[0, \pi]$
- Usare stream + memoria pinned per sovrapporre kernel che lavorano su parti dell'array
- Misurare in tempo le differenze rispetto alla versione su GPU sequenziale (cudaMemcpy e un solo kernel)
- Variare il numero di stream e riportare le migliori prestazioni

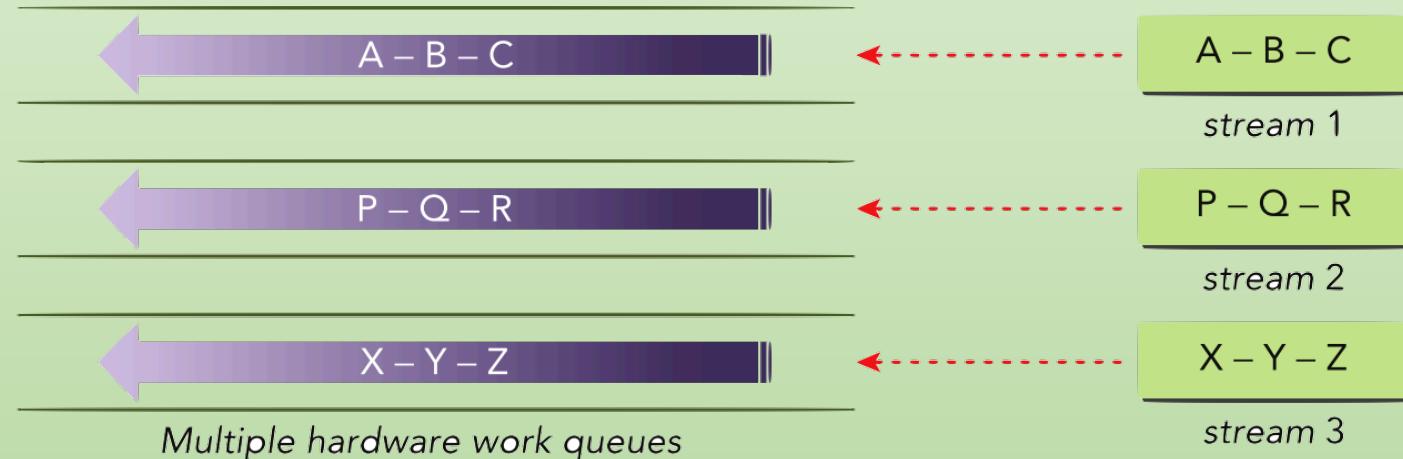
Scheduling di stream su FERMI

- ✓ Fermi supporta fino a **16 grid** in esecuzione simultaneamente e
- ✓ Tutti gli stream sono multiplexati su **una sola coda hardware**
- ✓ Solo i task di **stream evidenziati in figura** vengono eseguiti in concorrenza
- ✓ Questo crea delle **false dipendenze** tra quelli “al centro” delle code



HYPER Q

- ✓ Il problema delle false dipendenze in FERMI sono risolti con la tecnologia **Hyper Q**: code di lavoro hardware multiple
- ✓ Distinti **processi** (thread) in **CPU** possono lanciare lavoro su una **singola GPU** simultaneamente mantenendo aperte le connessioni tra host e device fino al completamento del flusso
- ✓ Kepler usa fino a **32 code hardware** e alloca una coda per ogni stream (eliminazione delle false dipendenze inter-stream)
- ✓ Se si hanno più di 32 stream, più stream vengono allocati nella stessa coda



Con e senza HyperQ

```
for (int i = 0 ; i < nstreams ; ++i) {  
    kernel_A<<<1,1,0,streams[i]>>>(&d_a[2*i], time_clocks);  
    total_clocks += time_clocks;  
    kernel_B<<<1,1,0,streams[i]>>>(&d_a[2*i+1], time_clocks);  
    total_clocks += time_clocks;  
}
```

Senza
HyperQ

Streams	Stream 6	kernel_A(long*, lo...)	kernel_B(long*, lo...)	Stream 7	kernel_A(long*, lo...)	kernel_B(long*, lo...)	Stream 8	kernel_A(long*, lo...)	kernel_B(long*, lo...)	Stream 9	kernel_A(long*, lo...)	kernel_B(long*, lo...)	Stream 10	kernel_A(long*, lo...)	kernel_B(long*, lo...)	Stream 11	kernel_A(long*, lo...)	kernel_B(long*, lo...)	Stream 12	kernel_A(long*, lo...)	kernel_B(long*, lo...)	Stream 13	kernel_A(long*, lo...)	kernel_B(long*, lo...)	Stream 14	kernel_A(long*, lo...)	kernel_B(long*, lo...)	Stream 15	kernel_A(lo

Con HyperQ

Streams	Stream 6	kernel_A(long*, long)	kernel_B(long*, long)	Stream 7	kernel_A(long*, long)	kernel_B(long*, long)	Stream 8	kernel_A(long*, long)	kernel_B(long*, long)	Stream 9	kernel_A(long*, long)	kernel_B(long*, long)	Stream 10	kernel_A(long*, long)	kernel_B(long*, long)	Stream 11	kernel_A(long*, long)	kernel_B(long*, long)	Stream 12	kernel_A(long*, long)	kernel_B(long*, long)	Stream 13	kernel_A(long*, long)	kernel_B(long*, long)	Stream 14	kernel_A(long*, long)	kernel_B(long*, long)	Stream 15	kernel_A(long*, long)	kernel_B(long*, long)	Stream 16	kernel_A(long*, long)	kernel_B(long*, long)	Stream 17	kernel_A(long*, long)	kernel_B(long*, long)	Stream 18	kernel_A(long*, long)	kernel_B(long*, long)	Stream 19	kernel_A(long*, long)	kernel_B(long*, long)	Stream 20	kernel_A(long*, long)	kernel_B(long*, long)

Priorità negli stream

- ✓ Nei device con **compute capability** almeno **3.5** possono essere assegnate priorità agli streams
- ✓ Una grid con più alta priorità può **prelazionare** il lavoro già in esecuzione con più bassa priorità
- ✓ Le priorità hanno effetto **solo** su **kernel** e **non** su **data transfer**
- ✓ Priorità al di **fuori** del **range** vengono riportate **automaticamente** nel **range**
- ✓ Per creare e gestire uno stream con priorità si usano le funzioni:

Prototipo

```
cudaError_t cudaStreamCreateWithPriority(cudaStream_t* pStream, unsigned int flags, int priority);
```

➤ Crea un nuovo stream con priorità intera e ritorna l'handle in pStream

Prototipo

```
cudaError_t cudaDeviceGetStreamPriorityRange(int *leastPriority, int *greatestPriority);
```

➤ restituisce la minima e massima priorità del device (la più alta è la minima)

CUDA event

- ✓ Un evento è un **marker** all'interno di uno stream associato a un punto del flusso delle operazioni
- ✓ Serve per controllare se l'**esecuzione** di uno stream ha **raggiunto** un dato **punto** o anche per la **sincronizzazione inter-stream**
- ✓ Può essere usato per due scopi base:
 - **Sincronizzare** l'esecuzione di stream
 - **Monitorare** il progresso del device
- ✓ Le API CUDA forniscono funzioni che consentono di **inserire** eventi in qualsiasi punto dello stream
- ✓ Oppure effettuare delle **query** per sapere se lo stream è stato completato
- ✓ **Eventi** sullo **stream di default** sincronizzano con **tutte le precedenti operazioni** su tutti gli stream

Creazione di CUDA event

Prototipo

```
cudaEvent_t event;  
cudaError_t cudaEventCreate(cudaEvent_t* event);
```

➤ Crea un nuovo evento di nome event

Nota:

- **eventi nello stream zero** vengono completati dopo che tutti i precedenti comandi in tutti gli stream sono stati completati.
- Gli eventi hanno uno **stato booleano**: occorso/non-occorso (default = occorso)

Prototipo

```
cudaError_t cudaEventDestroy(cudaEvent_t event);
```

➤ Comporta rilascio di risorse

Sincronizzazione via CUDA event

Prototipo

```
cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream);
```

- Registrazione di un evento su uno stream (0 per quello di default)

Nota: Set the event **state** to **not occurred**, state is set to **occurred** when it reaches the **front** of the **stream**

Prototipo

```
cudaError_t cudaEventSynchronize(cudaEvent_t event);
```

- Blocca l'host fino a che non si verifica evento

Prototipo

```
cudaError_t cudaEventQuery(cudaEvent_t event);
```

- Check di un evento senza bloccare

Prototipo

```
cudaError_t cudaStreamWaitEvent(cudaStream_t stream , cudaEvent_t event);
```

- Lo stream attende fino a che l'evento non occorre

Sincronizzazione esplicita

- ✓ **Synchronize everything**

`cudaDeviceSynchronize ()`

Blocks host until all issued CUDA calls are complete

- ✓ **Synchronize w.r.t. a specific stream**

`cudaStreamSynchronize (streamid)`

Blocks host until all CUDA calls in streamid are complete

- ✓ **Synchronize using Events**

Create specific 'Events', within streams, to use for synchronization

`cudaEventRecord (event, streamid)`

`cudaEventSynchronize (event)`

`cudaStreamWaitEvent (stream, event)`

`cudaEventQuery (event)`

Esempio: Sincronizzazione esplicita con eventi

```
cudaEvent_t event;

cudaEventCreate (&event);                                // create event

cudaMemcpyAsync ( d_in, in, size, H2D, stream1 );    // 1) H2D copy of new input

cudaEventRecord (event, stream1);                      // record event on stream1

cudaMemcpyAsync ( out, d_out, size, D2H, stream2 ); // 2) D2H copy of previous result

cudaStreamWaitEvent ( stream2, event );                // wait for event in stream1

kernel <<< , , , stream2 >>> ( d_in, d_out );    // 3) must wait for 1 and 2

asynchronousCPUMethod ( ... )                         // Async GPU method
```

Sincronizzazione implicita

- ✓ These operations implicitly synchronize all other CUDA operations
- ✓ Page-locked memory allocation

`cudaMallocHost`

`cudaHostAlloc`

- ✓ Device memory allocation

`cudaMalloc`

- ✓ Non-Async version of memory operations

`cudaMemcpy*` (no Async suffix)

`cudaMemset*` (no Async suffix)

- ✓ Change to L1/shared memory configuration

`cudaDeviceSetCacheConfig`

Misura del tempo con CUDA event

Prototipo

```
cudaError_t cudaEventElapsedTime(float* ms, cudaEvent_t start, cudaEvent_t stop);
```

- Misura il tempo intercorso tra due eventi in millisecondi
- ✓ Esempio con più eventi (trasferimento dati ed esec. kernel):

```
. . .
cudaEventSynchronize(time1);
    // trasferimento dati CPU → GPU
cudaEventSynchronize(time2);
    // esecuzione del kernel
cudaEventSynchronize(time3);
    // trasferimento dati GPU → CPU
cudaEventSynchronize(time4);

cudaEventElapsedTime(&totalTime, time1, time4);
cudaEventElapsedTime(&tfrCPUtoGPU, time1, time2);
cudaEventElapsedTime(&kernelExecutionTime, time2, time3);
cudaEventElapsedTime(&tfrGPUtoCPU, time3, time4);
```

Esempio misura del tempo

- ✓ Gli eventi **start** and **stop** non devono necessariamente essere associati allo stesso stesso stream (questo caso default)
- ✓ Viene registrato un **timestamp** per l'evento **start** all'inizio del NULL stream
- ✓ Viene registrato un **timestamp** per l'evento **stop** alla fine del NULL stream
- ✓ Il tempo intercorso tra **start** e **stop** è quello ottenuto usando **cudaEventElapsedTime**
- ✓ Si noti che se associati a uno stream non-NUL il valore di tempo restituito potrebbe essere maggiore di quanto atteso
- ✓ La funzione **cudaEventRecord** è asincrona e non c'è garanzia che la latenza misurata sia solo legata agli eventi dati come argomenti

```
// create two events
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// record 'start' event on the default stream
cudaEventRecord(start);

// execute kernel
kernel<<<grid, block>>>(arguments);

// record 'stop' event on the default stream
cudaEventRecord(stop);

// wait until the stop event completes
cudaEventSynchronize(stop);

// calculate the elapsed time between two events
float time;
cudaEventElapsedTime(&time, start, stop);

// clean up the two events
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Misura del tempo con NULL stream

- ✓ **CUDA programming:**
- ✓ The runtime also provides a way to closely monitor the device's progress, as well as perform accurate timing, by **letting the application asynchronously record *events*** at any point in the program, and query when these events are completed.
- ✓ An **event has completed** when **all tasks** - or optionally, all commands in a given stream - **preceding the event have completed**.
- ✓ **Events in stream zero** are completed **after all** preceding tasks and commands in all streams are completed.

```
// create two events
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

. . .

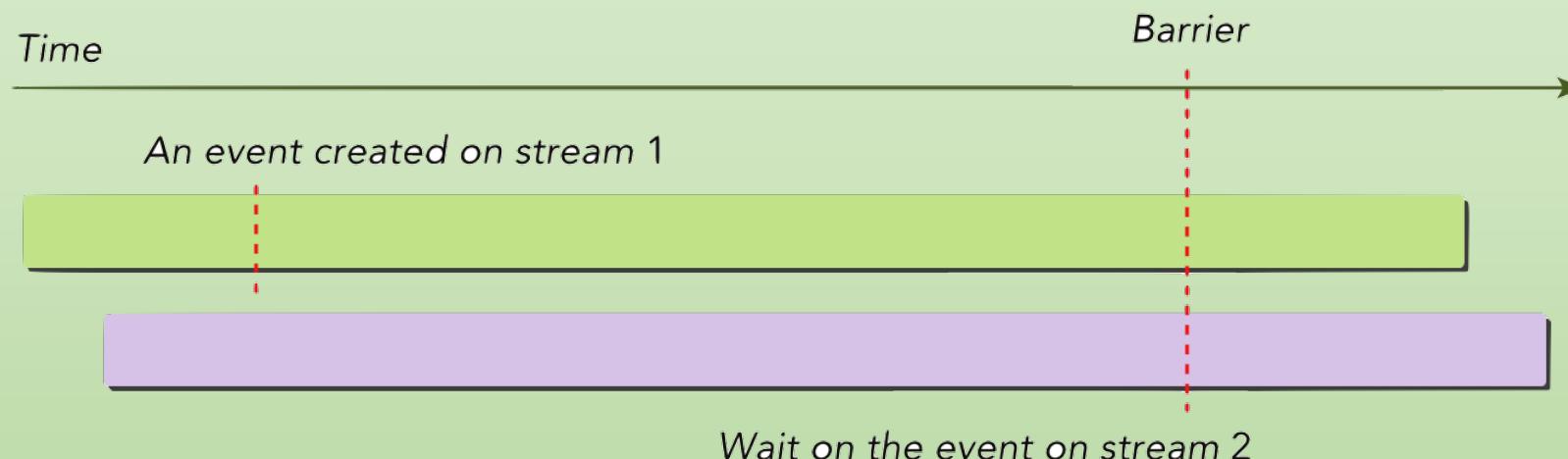
cudaEventRecord(start, 0);
for (int i = 0; i < k; ++i) {
    cudaMemcpyAsync(..., cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<n, m, 0, stream[i]>>>(a,b);
    cudaMemcpyAsync(..., cudaMemcpyDeviceToHost, stream[i]);
}
cudaEventRecord(stop, 0);
cudaEventSynchronize (stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
. . .
```

Sincronizzazione esplicita

La funzione **cudaStreamWaitEvent** offre un modo flessibile per introdurre dipendenza inter-stream usando CUDA event:

```
cudaError_t cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event);
```

- ✓ Impone allo stream passato per argomento di attendere lo specifico evento prima di eseguire ogni altra operazione in coda nello stream successivo alla chiamata **cudaStreamWaitEvent**
- ✓ L'evento può essere associato allo **stesso** oppure a uno **differente (cross-stream synchronization)**



Sincronizzazione esplicita

CUDA runtime supporta diversi modi di sincronizzazione esplicita a livello di grid in un programma CUDA :

- ✓ **Synchronizing the device**
- ✓ **Synchronizing a stream**
- ✓ **Synchronizing an event in a stream**
- ✓ **Synchronizing across streams using an event**

Si può bloccare il thread dell'host fino a che il device non abbia completato i task precedenti con al funzione:

```
cudaError_t cudaDeviceSynchronize(void);
```

Si può bloccare il thread dell'host fino a che tutte le operazioni in uno stream siano state completate (**cudaStreamSynchronize**) oppure eseguire un test non-bloccante (**cudaStreamQuery**):

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

CUDA event può anche essere usato per bloccare e sincronizzare:

```
cudaError_t cudaEventSynchronize(cudaEvent_t event);
cudaError_t cudaEventQuery(cudaEvent_t event);
```

Esempio con 1 e 2 stream ...

Creazione di eventi per il timing

Uso della memoria pinned

```
cudaEvent_t start, stop;
float elapsedTime;

cudaStream_t stream;
int *host_a, *host_b, *host_c;
int *dev_a, *dev_b, *dev_c;

// start the timers
cudaEventCreate(&start);
cudaEventCreate(&stop);

// initialize the stream
cudaStreamCreate(&stream);

// allocate the memory on the GPU
cudaMalloc((void**) &dev_a, N * sizeof(int));
cudaMalloc((void**) &dev_b, N * sizeof(int));
cudaMalloc((void**) &dev_c, N * sizeof(int));

// allocate host locked memory, used to stream
cudaHostAlloc((void**) &host_a, FULL_DATA_SIZE *
    sizeof(int), cudaHostAllocDefault);
cudaHostAlloc((void**) &host_b, FULL_DATA_SIZE *
    sizeof(int), cudaHostAllocDefault);
cudaHostAlloc((void**) &host_c, FULL_DATA_SIZE *
    sizeof(int), cudaHostAllocDefault);
```

Esempio con 1 e 2 stream

Molti
kernel su
1 stream

```
#define N    (1024*1024)
#define FULL_DATA_SIZE    (N*1000)
. . .
cudaEventRecord(start, 0);
// now loop over full data, in bite-sized chunks
for (int i = 0; i < FULL_DATA_SIZE; i += N) {
    // copy the locked memory to the device, async
    cudaMemcpyAsync(dev_a, host_a + i, N * sizeof(int), cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync(dev_b, host_b + i, N * sizeof(int), cudaMemcpyHostToDevice, stream);
    kernel<<<N / 256, 256, 0, stream>>>(dev_a, dev_b, dev_c);
    // copy the data from device to locked memory
    cudaMemcpyAsync(host_c + i, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost, stream);
}
```

Molti
kernel su
2 stream

```
for (int i = 0; i < FULL_DATA_SIZE; i += N * 2) {

    // enqueue copies of a in stream0 and stream1
    cudaMemcpyAsync(dev_a0, host_a + i, N * sizeof(int), cudaMemcpyHostToDevice, stream0);
    cudaMemcpyAsync(dev_a1, host_a + i + N, N * sizeof(int), cudaMemcpyHostToDevice, stream1);

    // enqueue copies of b in stream0 and stream1
    cudaMemcpyAsync(dev_b0, host_b + i, N * sizeof(int), cudaMemcpyHostToDevice, stream0);
    cudaMemcpyAsync(dev_b1, host_b + i + N, N * sizeof(int), cudaMemcpyHostToDevice, stream1);

    // enqueue kernels in stream0 and stream1
    kernel<<<N / 256, 256, 0, stream0>>>(dev_a0, dev_b0, dev_c0);
    kernel<<<N / 256, 256, 0, stream1>>>(dev_a1, dev_b1, dev_c1);

    // enqueue copies of c from device to locked memory
    cudaMemcpyAsync(host_c + i, dev_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0);
    cudaMemcpyAsync(host_c + i + N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1);
}
```

Esempio con 2 stream (tempi)

sincronizzazione

Tempi di esecuzione

```
cudaStreamSynchronize(stream0);
cudaStreamSynchronize(stream1);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);
printf("Time taken: %3.1f ms\n", elapsedTime);
```

```
$ basic_single_stream
Time taken: 2960.4 ms
```

Time(%)	Time	Calls	Avg	Min	Max	Name
59.98%	1.79772s	2000	898.86us	890.60us	913.54us	[CUDA memcpy HtoD]
32.48%	973.28ms	1000	973.28us	675.60us	1.0995ms	[CUDA memcpy DtoH]
7.54%	225.98ms	1000	225.98us	225.15us	226.90us	kernel(int*, int*, int*)

```
$ basic_double_stream
Time taken: 2659.9 ms
```

Time(%)	Time	Calls	Avg	Min	Max	Name
57.07%	1.93680s	2000	968.40us	883.68us	1.3361ms	[CUDA memcpy HtoD]
36.26%	1.23066s	1000	1.2307ms	675.69us	1.6416ms	[CUDA memcpy DtoH]
6.67%	226.20ms	1000	226.20us	225.40us	227.68us	kernel(int*, int*, int*)