Riassunto 4° capitolo di “Professional CUDA C Programming”: Modello di memoria in CUDA, Memoria Globale e pattern di accesso.

1 - Gerarchia di memoria

Per rendere le nostre applicazioni CUDA efficienti, non basta giocare con gli indici dei thread: bisogna anche cercare di rendere efficienti gli accessi alla memoria, in quanto spesso possono comportare un bottleneck per le prestazioni dei kernel. In particolare, hanno effetto sulle nostre computazioni sia i trasferimenti di dati da CPU a GPU che le letture di dati dagli Streaming Multiprocessor (SM) alla Memoria Globale della GPU. In CUDA troviamo, come in qualsiasi calcolatore che si rispetti, una gerarchia di memoria, con memorie di ogni dimensione, velocità, bandwidth e scopo. Una prima classificazione delle memorie che è utile per i programmatori è la seguente:

- Memorie programmabili: si tratta delle memorie sotto il controllo del programmatore. Qui possiamo scegliere cosa mettere in memoria.
- Memorie NON programmabili: non abbiamo controllo di come i dati vengono messi in queste memorie, ci pensa l’hardware insomma. Tuttavia, se capiamo come l’hardware gestisce queste memorie, possiamo comunque trarne dei benefici.

Cerchiamo di capire, innanzitutto, la gerarchia di memorie presenti in CUDA. L’immagine che segue è una schematizzazione di tale gerarchia.

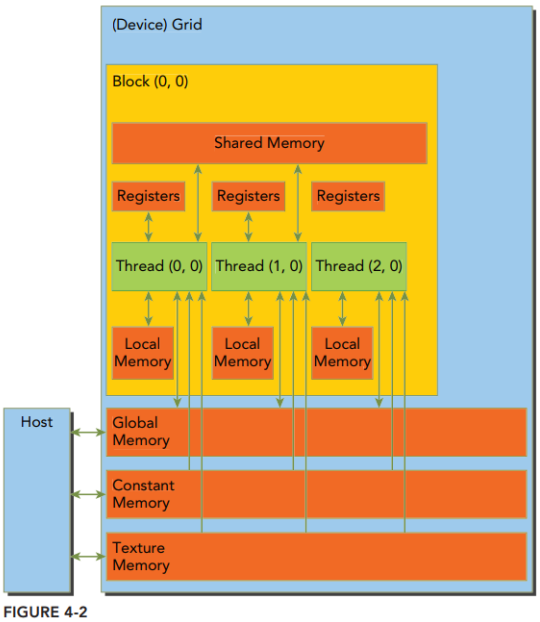


FIGURE 4-2

Le memorie di CUDA possono essere classificate sulla base del loro scope, ovvero sulla base di chi le può usare: alcune possono essere usate da un singolo thread, altre da un intero blocco, altre ancora dall’intera

grid (quindi dall'intera applicazione). A seconda della memoria considerata variano ovviamente latenza, capacità e badwidth. Partiamo da quelle più specifiche per poi scendere a quelle più generali.

Registri

Sono le memorie più piccole e più veloci che ci siano. Un registro può essere visto (leggi "visto" come "ha come scope") da un solo thread, e di solito nei registri di un thread ci vanno a finire variabili temporanee (automatiche) dichiarate nel kernel di riferimento e array riferiti da indici costanti (ovvero che possono essere determinati a tempo di compilazione). Quando un kernel termina la sua esecuzione, i registri associati ai suoi thread vengono liberati. In ogni SM ci sono pochi registri, che però vanno divisi equamente tra i thread degli warp attivi su quel SM (ricordiamo infatti che in un SM ci possono essere più warp/blocchi attivi in un istante (?)). Per esempio, sulle GPU Fermi, ci possono essere al massimo 63 registri per thread, da Kepler in poi invece fino a 255 registri. Tanti meno registri un thread usa, tanti più thread block possono risiedere contemporaneamente su un SM. Se però un thread usa tante variabili/array, i registri possono non bastare. Si verifica quindi il cosiddetto *register spiling*, ovvero ci sono variabili che dovrebbero risiedere in dei registri, ma non ce ne sono a sufficienza. Per conservare tali variabili si ripiega allora nella…

Local Memory

Tutte le variabili che un thread non riesce a mettere nei registri ad esso dedicati vanno a finire qui. Ci finiscono anche array locali al thread i cui valori degli indici non sono costanti e in generale strutture dati, array o meno che siano, troppo grandi per poter essere messi nei registri. C'è una fregatura però: la Local Memory non è locale per un corno al suo thread, nel senso che queste due entità sono molto distanti fra loro. **La locazione fisica delle Local Memory è la stessa della Global Memory, che è la memoria più distante dagli SM.** Questo implica che se un thread deve accedere alla sua memoria locale deve aspettarsi alta latenza e bassa bandwidth. Tutti i discorsi di ottimizzazione riguardanti l'accesso alla memoria globale che faremo (Memory Access Patterns), pertanto, sono validi anche per questa memoria. Nota: dalle GPU con Compute Capability 2.0 e superiore, i dati della Local Memory possono essere cachati sia nella cache L1 dello SM in cui risiede il thread, sia nella cache L2 globale invece all'intero device. Ci saranno più dettagli sulle cache a seguire.

Shared Memory (SMEM)

Si tratta della prima memoria (tra quelle che stiamo analizzando) che è visibile da un intero blocco di thread, anziché da un singolo thread. È una memoria peggiore dei registri ma migliore della locale e globale (in termini di bandwidth e latenza: è chiaramente più grande dei registri ma più piccola della globale). Come coi registri, ogni SM ha un certo quantitativo di Shared Memory che può mettere a disposizione dei thread block che risiedono in esso: tanta meno SMEM un blocco richiede, tanti più blocchi possono stare attivi contemporaneamente nello stesso SM. La Shared Memory si dichiara, a livello di codice, *all'interno di un kernel*, ma la sua vita è lunga tanto quanto quella del blocco a cui viene assegnata. Quando un blocco finisce la sua esecuzione, la SMEM che gli era stata allocata viene liberata. Si possono dichiarare variabili o array che dovranno finire in questa memoria semplicemente apponendo il qualificatore

__shared__

a una variabile/array dichiarate nel kernel. L'idea è quella di usare la SMEM come mezzo di comunicazione tra i thread dello stesso blocco. Questo però può portare a situazioni di inconsistenza se non si fa attenzione. È pertanto buona prassi chiamare

__syncthreads();

tutte le volte che i thread devono sincronizzarsi sulla SMEM. Un caso in cui la SMEM torna comoda l'abbiamo visto a lezione, ad esempio con la convoluzione: se i thread di un blocco devono leggere tutti gli stessi dati (più o meno), anziché andarli a prendere ogni volta dalla memoria globale, tanto vale portarli tutti quanti in SMEM (dividendo il compito tra i vari thread dello stesso blocco) all'inizio dei tempi, per poi usarli. La SMEM è infatti più vicina agli SM della Global Memory.

Una cosa da notare è che La SMEM condivide con la cache L1 (entrambe sono infatti relative a un singolo SM) la stessa memoria on-chip da 64 KB, che viene staticamente partizionata per dare spazio a entrambe le memorie. Tuttavia, al programmatore viene data la possibilità di impostare dinamicamente questa partizione con la funzione

```
cudaError_t cudaFuncSetCacheConfig(const void* func, enum cudaFuncCache
   cacheConfig);
```

che permette di specificare che, per il kernel rappresentato dalla funzione func, vogliamo suddividere la memoria on-chip da 64 KB tra SMEM e L1 in un particolare modo, rappresentato dal secondo parametro (cudaFuncCachecacheConfig). I vari modi sono:

```
cudaFuncCachePreferNone:   no preference (default)
cudaFuncCachePreferShared: prefer 48KB shared memory and 16KB L1 cache
cudaFuncCachePreferL1:     prefer 48KB L1 cache and 16KB shared memory
cudaFuncCachePreferEqual:  Prefer equal size of L1 cache and shared memory,
   both 32KB
```

Constant Memory

Si tratta della prima memoria visibile all'intera grid, anche se ogni SM ha una cache dedicata specificatamente ad essa. Per dichiarare una variabile/array costante (cioè che deve risiedere in questa memoria) si usa il qualificatore

__constant__

al di fuori di un kernel (i dati sono costanti per l'applicazione intera, quindi per ogni kernel). Tra l'alatro questa memoria ha una dimensione fissa di 64 KB indipendentemente dalla Compute Capability. Si tratta di una memoria a sola lettura, e la Constant Memory del kernel va inizializzata dall'host usando la funzione

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, const void* src,
   size_t count);
```

che dice: copia count byte da src verso symbol, che è una variabile che risiede sul device. Si tratta di una funzione sincrona. Si usa questa memoria quando i thread di un warp vanno a leggere sempre dallo stesso indirizzo di memoria (es. un coefficiente di una formula). Infatti una singola read dalla Constant Memory viene mandata in broadcast a tutti i thread di un warp.

Texture Memory

Anch'essa visibile dall'intera grid, ma non ci interessa.

Global Memory

Visibile dall'intera grid, si tratta della memoria più grande e a più alta latenza della GPU, ma è anche quella più comunemente usata. Nel corso dell'esecuzione di un'applicazione, può essere acceduta da un qualunque SM.

Si può allocare una variabile/array in memoria globale sia staticamente che dinamicamente. Per farlo staticamente si dichiara una variabile col qualificatore

__device__

che deve essere poi copiata nel device con cudaMemcpyToSymbol() (e copiata di nuovo nell'host con cudaMemcpyFromSymbol()). Segue un esempio di dichiarazione statica di una variabile globale.

LISTING 4-1: Static declared global variable (globalVariable.cu)

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

__device__ float devData;

__global__ void checkGlobalVariable() {
   // display the original value
   printf("Device: the value of the global variable is %f\n",devData);

   // alter the value
   devData +=2.0f;
}

int main(void) {
   // initialize the global variable
   float value = 3.14f;
   cudaMemcpyToSymbol(devData, &value, sizeof(float));
   printf("Host:   copied %f to the global variable\n", value);

   // invoke the kernel
   checkGlobalVariable <<<1, 1>>>();

   // copy the global variable back to the host
   cudaMemcpyFromSymbol(&value, devData, sizeof(float));
   printf("Host:   the value changed by the kernel to %f\n", value);

   cudaDeviceReset();
   return EXIT_SUCCESS;
}
```

Per dichiarare dinamicamente queste variabili, invece, si usano le funzioni cudaMalloc() e cudaFree(). I puntatori passati come primi argomenti a tutte queste funzioni sono poi quelli che verranno usati dai kernel. Se più thread modificano in maniera non controllata dati presenti in memoria globale, si possono creare inconsistenze. Quindi occhio, anche perché lo scope di questa memoria è dell'intera grid.

La Global Memory può essere acceduta mediante transazioni di memoria da 32, 64 o 128 byte, e sono sempre allineate e coalescenti, ovvero, il primo indirizzo letto può essere solo ed esclusivamente un multiplo di 32, 64 o 128, e una volta letto quello verranno letti necessariamente anche i successivi 31, 63 o 127 byte adiacenti. Questo è molto importante per capire poi come rendere efficienti gli accessi alla memoria globale.

Quando i thread di un warp hanno bisogno di leggere dati, dovrebbero accedere, il più possibile, a 32, 64, o 128 byte allineati e coalescenti. L'unità di misura è infatti una transazione: non importa da quanti byte sia.

Es: un warp che richiede l'accesso a 128 byte consecutivi risulta in una solta transazione, che è più efficiente di un warp che, nel complesso, richiede di accedere a due blocchi da 32 byte non contigui (per esempio dall'indirizzo 32 al 63 e dal 92 al 127), richiesta che deve essere soddisfatta con due transazioni da 32 byte ciascuna. *Sono le transazioni che contano, non i byte*.

Es: allo stesso modo, per citare invece un caso di allineamento, se il nostro warp vuole accedere a 128 byte partendo dall'indirizzo 1 in un caso e dall'indirizzo 0 nel secondo, avremo due transazioni nel primo caso e una sola nel secondo. Infatti, anche se sono sempre 128 byte quelli da leggere, le transazioni possono partire

sempre e solo da un multiplo di 32, 64 o 128. Quindi nel primo caso ci possiamo aspettare una transazione da 128 byte che parte dall'indirizzo 0, e un'altra da 32, 64 o 128 (non lo so lol) che parte dall'indirizzo 128, *solo per leggere un byte tra l'altro!* Nel secondo caso invece ce la caviamo con una sola transazione da 128 byte che parte dall'indirizzo 0.

Quindi, per misurare l'efficienza di un'operazione di memoria compiuta da un warp (leggi come: compiuta da i thread di un warp ad un certo istante), bisogna contare le transazioni necessarie e dare ai thread i dati che richiedono (nello stesso istante, essendo i thread eseguiti in modo SIMT, abbiamo che due thread diversi o fanno la stessa cosa, in questo caso una richiesta di load/store, o uno fa questa richiesta e l'altro è idle, o sono entrambi idle). Se tutti i thread di un warp, nello stesso momento (alla stessa riga di codice literally), richiedono indirizzi allineati e coalescenti, avremo una richiesta che può essere soddisfatta nel numero minimo di transazioni, spesso una sola (tipo se tutti devono accedere a interi consecutivi di un array).

In generale, tante più sono le transazioni richieste, tanto più il rischio di trasferire byte non necessari, cosa che causa una riduzione nell'efficienza di throughput.

Spostiamoci ora al mondo delle cache della GPU, anch'esse con scope diversi.

Le Cache di CUDA risiedono nel mondo delle memorie NON programmabili, ma non per questo non dobbiamo evitare di studiarne il comportamento. Vedremo anzi che comprendere il ruolo soprattutto delle cache L1 e L2 ci aiuterà a migliorare le nostre applicazioni. Ci sono 4 cache in tutto:

- L1, una cache on-chip presente su ogni SM.
- L2, una cache più grande condivisa da tutte le SM.
- Read-only constant.
- Read-only texture.

Sia la L1 che la L2 (che sono anche le uniche cache che ci interessano davvero) conservano dati prelevati dalla memoria locale e globale, compresi i register spills visti prima. A partire dalle GPU più recenti (Fermi e Kepler K40 o successive), è possibile abilitare e disabilitare la cache L1. Ora, qui il libro dice che queste cache intervengono solo nelle operazioni di load, ma sono abbastanza sicuro che più avanti invece si corregga e ammetta store sulla L2.

Le ultime due cache si trovano su ogni SM, e sono usate per migliorare le performance.

Segue un breve riassunto dei qualificatori e delle memorie analizzate fino ad ora.

Qualificatori:

TABLE 4-1: CUDA Variable and Type Qualifier

QUALIFIER	VARIABLE NAME	MEMORY	SCOPE	LIFESPAN
	`float var`	Register	Thread	Thread
	`float var[100]`	Local	Thread	Thread
`__shared__`	`float var†`	Shared	Block	Block
`__device__`	`float var†`	Global	Global	Application
`__constant__`	`float var†`	Constant	Global	Application

† Can be either scalar variable or array variable

Memorie:

TABLE 4-2: Salient Features of Device Memory

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached only on devices of compute capability 2.x

2 – Gestione della memoria

Generalmente, anche se sia il codice host che il codice device di un'applicazione CUDA risiedono nello stesso file, sappiamo bene che questi esistono in due mondi paralleli. Il codice host viene eseguito dalla CPU, quello device dalla GPU. La maggiore "fregatura", nel senso che si tratta di una cosa controintuitiva per un programmatore, è che, per poter accedere a quelli che sono logicamente gli stessi dati, spesso ci servono due puntatori: uno per l'host e uno per il device. Ad esempio, quando abbiamo visto il codice in cui veniva usata una variabile statica globale nella sezione dedicata alla Global Memory, abbiamo dovuto usare CudaMemcpyTo/FromSymbol, passando ogni volta coppie di puntatori e dimensione in byte, per poter copiare i dati da host a device e da device a host. Questo perché l'host non può accedere alla memoria del device e viceversa, almeno normalmente. L'unica eccezione a questa regola, che quindi permette all'host di riferire direttamente la memoria della GPU, sta nella Pinned Memory.

Dopo questa introduzione, atta a "stimolare" il pensiero dello studente lettore verso la gestione della memoria in CUDA, possiamo dire che ci occuperemo, in questa sezione, di:

- Capire come allocare e deallocare memoria sul device.
- Capire i vari modi che si hanno per poter trasferire data tra host e device.

Memory Allocation and Deallocation

In generale, una qualunque applicazione CUDA prevede di allocare memoria sul device, trasferire memoria dall'host al device, fare calcoli usando quei dati, e infine copiare il risultato del kernel dalla memoria device a quella host. Per allocare memoria device, si usa la funzione

```
cudaError_t cudaMalloc(void **devPtr, size_t count);
```

che alloca count byte sul device, dove devPtr è un puntatore a quei byte. Nota che il contenuto della memoria allocata può essere qualsiasi cosa: magari lì dentro ci sono ancora i dati di una vecchia esecuzione, chi più dirlo. Solitamente questo non è un problema, dato che dopo aver allocato della memoria device solitamente la si vuole poi inizializzare. Ma, in generale, se si vuole essere sicuri del contenuto iniziale della porzione di memoria allocata, si può usare la funzione

```
cudaError_t cudaMemset(void *devPtr, int value, size_t count);
```

che riempie count byte di dati, che partono da devPtr, col valore value. Infine, per liberare la memoria device allocata, si usa:

```
cudaError_t cudaFree(void *devPtr);
```

Memory Transfer

Una volta allocata la memoria (globale) sul device, è possibile trasferire dati da host a device e viceversa usando la funzione

```
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count,
                       enum cudaMemcpyKind kind);
```

Dove kind può assumere uno dei seguenti valori, con l'ovvio significato:

```
cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
```

L'idea è: copia count byte da src in dst, e il kind deve essere appropriato rispetto alla natura dei due puntatori. Questa funzione è sincrona.

Ora, guardiamo un attimo l'immagine 4-3:

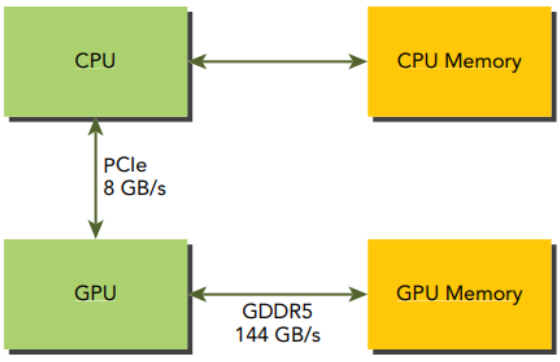


FIGURE 4-3

Essa ci mostra come, in termini di bandwidth, sia molto più efficiente trasferire dati dalla memoria della GPU ai suoi SM (144 GB/s) rispetto a trasferire dati dalla CPU alla GPU (8 GB/s). In poche parole:

per alcune applicazioni, il bottleneck potrebbe essere costituito dalla velocità con cui i dati del problema vengono trasferiti dall'host al device, e non dalla complessità del problema stesso.

Questo implica che dovremmo cercare di minimizzare i trasferimenti di dati da host a device e viceversa. Proprio per questo motivo, studiamo ora meccanismi alternativi per la gestione della memoria in CUDA.

Pinned Memory

Nell'host, la memoria della RAM è normalmente paginabile, ovvero soggetta a page fault: si fa credere a ogni processo (i cui dati e codice sono divisi in pagine, ovvero chunk di byte di dimensione fissata) di avere una quantità di memoria volatile illimitata. "Under the hood", però, il sistema operativo prende pagine inutilizzate e le salva sul disco quando c'è bisogno di spazio in memoria centrale e allo stesso tempo può recuperare pagine messe precedentemente su disco e riportarle in memoria centrale all'occorrenza. Quando però il SO fa queste cose, a noi, non è dato saperlo. Ne consegue che la GPU non può semplicemente copiare dati presenti in memoria centrale, perché se questi vengono buttati sul disco dal SO mentre c'è un trasferimento in corso, i dati trasferiti divengono corrotti. Infatti, dietro le quinte, tutte le copie da host a device sono gestite come segue: il driver CUDA alloca temporaneamente, sulla RAM dell'host, una memoria *page-locked* o *pinned*, ovvero memoria non soggetta a paginazione: così stiamo tranquilli! A questo punto i

dati dell'host vengono copiati in questa memoria, e poi da questa a quella del device. La cosa figa è che, sapendo questo meccanismo, possiamo ora "svelare" il fatto che, a noi sviluppatori, viene data la possibilità, con un'apposita API, di allocare esplicitamente della memoria pinned:

```
cudaError_t cudaMallocHost(void **devPtr, size_t count);
```

Con questa funzione si allocano count byte di memoria page-locked sull'host a partire dall'indirizzo devPtr. Allo stesso tempo, questa memoria può essere liberata con la corrispondente Free:

```
cudaError_t cudaFreeHost(void *ptr);
```

Questa memoria può essere letta e scritta con bandwidth maggiore rispetto alla memoria paginabile, e la modalità di accesso è asincrona. Ci sono esempi (vedi libro e slide) che dimostrano come con questa memoria i trasferimenti da host a device e viceversa accelerino notevolmente. Quindi perché non allocare sempre pinned memory, quando bisogna passare dati al device? Perché stiamo comunque allocando memoria page-locked nella RAM dell'host, di fatto sottraendo quella memoria a tutte le altre applicazioni dell'host (degradiamo le performance dell'host insomma). Mettiamo infine a confronto due pezzi di codice, dove nel primo la memoria viene allocata normalmente, mentre nel secondo si sfrutta la Pinned Memory:

```
// allocate the host memory
float *h_a = (float*)malloc(nbytes);
// allocate the device memory
float *d_a;
cudaMalloc((void**)&d_a, nbytes);
// transfer data from the host to the device
cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice);
// transfer data from the device to the host
cudaMemcpy(h_a, d_a, nbytes, cudaMemcpyDeviceToHost);
// free memory
cudaFree(d_a);
free(h_a);
```

```
// allocate pinned host memory
float *h_a;
cudaMallocHost((void**)&h_a, nbytes);
// allocate device memory
float *d_a;
cudaMalloc((float **)&d_a, nbytes);
// transfer data from the host to the device
cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice);
// transfer data from the device to the host
cudaMemcpy(h_a, d_a, nbytes, cudaMemcpyDeviceToHost);
// free memory
cudaFree(d_a));
cudaFreeHost(h_a);
```

Sì, lo so che da questo codice sembra sia cambiata una sola riga. Il fatto è questo: mentre a sinistra te prima allochi della memoria paginabile per poi dietro le quinte trasferire i dati in una memoria pinned (allocata dall'API) e infine trasferire i dati da questa memoria a quella del device, a destra te allochi i tuoi dati DIRETTAMENTE nella pinned, e poi copi da essa nel device. Salti un passaggio insomma, come mostra la figura che segue.

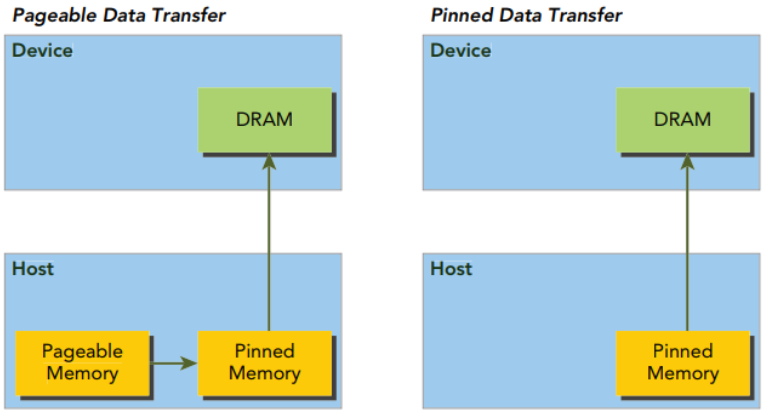


FIGURE 4-4

Ho avuto un'epifania. Il libro dice che con la Pinned Memory il device ha accesso diretto alla memoria dell'host, ma dal codice sopra sembra che dobbiamo comunque fare una copia dall'host al device. In realtà, con "accesso diretto", intendeva dire che non c'è bisogno di copiare dati dalla memoria paginabile dell'host a quella pinned. Non c'è insomma alcun livello di indirezione, mettiamola così.

Zero-Copy Memory

Si tratta di una memoria accessibile sia dall'host che dal device, e in quanto tale bisogna sincronizzarne gli accessi. Può essere utile quando non c'è sufficiente memoria nel device o quando non si vogliono trasferire esplicitamente dati dall'host al device. La Zero-Copy memory è come la Pinned nel senso che entrambe sono memorie non paginabili allocate sull'host, e la Zero-Copy viene mappata nello spazio di indirizzamento del device (appunto, è accessibile dal device). Per allocare questo tipo di memoria si usa la funzione:

```
cudaError_t cudaHostAlloc(void **pHost, size_t count, unsigned int flags);
```

Che alloca count byte di memoria host page-locked, a partire da pHost, accessibile dal device. Per liberare questa memoria, si usa cudaFreeHost(), proprio come con la Pinned. L'ultimo argomento, flags, permette di configurare ulteriormente la memoria allocata.

- cudaHostAllocDefault: con questo parametro questa funzione fa la stessa cosa di cudaMallocHost() (allochi della vera e propria Pinned Memory insomma).
- cudaHostAllocPortable: la memoria pinned allocata può essere usata da tutti i contesti CUDA, non solo quello che l'ha allocata (non chiedermi cosa voglia dire).
- cudaHostAllocWriteCombined: la memoria restituita è di tipo write-combined, e può essere trasferita più velocemente sul bus PCI express su alcune configurazioni di sistema ma non può essere letta efficientemente da un sacco di host.
- cudaHostAllocMapped, che restituisce host memory che viene mappata nello spazio di indirizzamento del device. Di fatto, se vuoi della Zero-Copy Memory vera e propria, devi usare questo flag.

Anche se si tratta di una memoria accessibile sia dall'host che dal device, è necessario "istanziare" due puntatori diversi: uno usato dall'host per accedere a questa memoria, l'altro usato dal device. Quello dell'host l'abbiamo visto sopra in cudaHostAlloc, mentre quello del device può essere acceduto usando:

```
cudaError_t cudaHostGetDevicePointer(void **pDevice, void *pHost, unsigned int flags);
```

Che restituisce, in pDevice, un puntatore alla Zero-Copy memory che il device può usare.

Ora, attenzione: stiamo allocando memoria sull'host che verrà acceduta (anche) dal device. Per farlo, il device deve passare dal PCI express, che abbiamo visto avere una latenza più bassa di quella tra device e memoria device, quindi, non bisogna progettare un'applicazione affinché faccia utilizzo spasmodico di questa memoria. **La latenza per leggere o scrivere dati in questa memoria è perfino superiore a quella necessaria per accedere alla memoria globale**. Ma se questa memoria ha come svantaggi il fatto che viene allocata sull'host come memoria non paginabile e che è più lenta da accedere rispetto a quella device, ha se non altro come vantaggio che non c'è bisogno di copiare i dati da host a device quando si scrive un'applicazione CUDA:

```
// allocate zerocpy memory
CHECK(cudaHostAlloc((void **) &h_A, nBytes, cudaHostAllocMapped));
CHECK(cudaHostAlloc((void **) &h_B, nBytes, cudaHostAllocMapped));
// pass the pointer to the device
cudaHostGetDevicePointer((void **)&d_A, (void *)h_A, 0);
cudaHostGetDevicePointer((void **)&d_B, (void *)h_B, 0);
// execute kernel with zero copy memory
sumArraysZeroCopy<<<grid, block>>>(d_A, d_B, d_C, nElem);
```

(in questa immagine, i dati sull'host vengono inizializzati chiamando una generica funzione initialData(h_, nElem), prima con h_A e poi con h_B, tra cudaHostAlloc e (…)GetDevicePointer)

E se lo chiedi a me, questo è un vantaggio del cavolo! Ti risparmi la scrittura di sì e no 5 -10 righe di codice ma hai un degradamento delle performance MOSTRUOSO quando accedi a questa memoria. Inoltre, devi coordinare gli accessi a questa memoria, dato che ci mettono le mani sia l'host che il device. Per me, fa schifo.

Ah tra l'altro nota bene: se il tuo kernel calcola dei valori in, che so, un array, poi quello va copiato da device a host. Boh, per me non ha senso (cioè, lo capisco il senso ma mi fa schifo, io programmo sulla GPU perché voglio le performance caspita! Anche se, per applicazioni piccole, ci possono essere speedup: però di nuovo, se devo fare un'applicazione piccola la faccio sulla CPU e bona).

```
// execute kernel with zero copy memory
sumArraysZeroCopy <<<grid, block>>>(d_A, d_B, d_C, nElem);

// copy kernel result back to host side
cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost);

// check device results
checkResult(hostRef, gpuRef, nElem);

// free  memory
cudaFree(d_C);
cudaFreeHost(h_A);
cudaFreeHost(h_B);
```

TABLE 4-3: Comparison of Zero-copy Memory vs Device Memory

SIZE	DEVICE MEMORY (ELAPSED TIME)	ZERO-COPY MEMORY (ELAPSED TIME)	SLOWDOWN
1 K	1.5820 us	2.9150 us	1.84
4 K	1.6640 us	3.7900 us	2.28
16 K	1.6740 us	7.4570 us	4.45
64 K	2.3910 us	22.586 us	9.45
256 K	7.2890 us	82.733 us	11.35
1 M	28.267 us	321.57 us	11.38
4 M	104.17 us	1.2741 ms	12.23
16 M	408.03 us	5.0903 ms	12.47
64 M	1.6276 ms	20.347 ms	12.50

Tesla M2090 ECC on and L1 cache enabled
Slowdown = Elapsed Time of Zero-copy Reads / Elapsed Time of Device Memory Reads

From the results, you can see that if you share a small amount of data between the host and device, zero-copy memory may be a good choice because it simplifies programming and offers reasonable performance. For larger datasets with discrete GPUs connected via the PCIe bus, zero-copy memory is a poor choice and causes significant performance degradation.

Unified Virtual Addressing

In poche parole: la UVA è una Zero-Copy Memory che ci ha creduto un po' di più, dato che permette di eliminare un altro paio di righe di codice. Tuttavia, è inefficiente come la Zero-Copy.

In più parole, invece: la UVA permette alla memoria sia dell'host che del device di condividere un singolo spazio di indirizzamento virtuale. Si tratta di un modello di indirizzamento introdotto in Cuda 4.0, e viene supportato da architetture linux 64-bit.

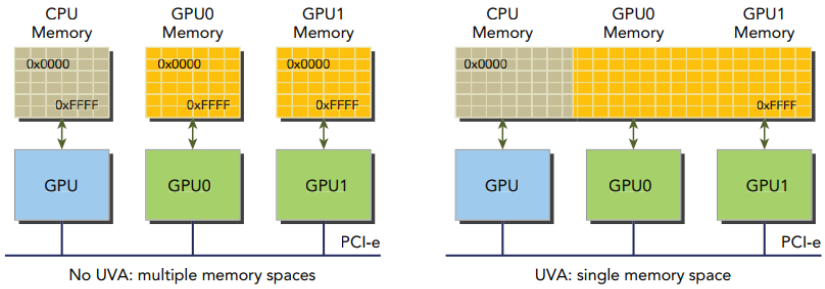


FIGURE 4-5

Si tratta letteralmente di un'evoluzione della Zero-Copy Memory, in quanto per usare l'UVA devi fare le stesse cose che facevi con la Zero-Copy, TRANNE prendere il puntatore del device (elimini insomma cudaHostGetDevicePointer).

```
// allocate zerocpy memory
CHECK(cudaHostAlloc((void **) &h_A, nBytes, cudaHostAllocMapped));
CHECK(cudaHostAlloc((void **) &h_B, nBytes, cudaHostAllocMapped));
// initialize data at host side
initialData(h_A, nElem);
initialData(h_B, nElem);
// execute kernel with zero copy memory
sumArraysZeroCopy<<<grid, block>>>(h_A, h_B, d_C, nElem);
```

Unified Memory

Da CUDA 6.0 in poi, è stata introdotta la Unified Memory con lo scopo di semplificare la gestione della memoria nel modello di programmazione CUDA. Crea di fatto una pool di memoria gestita (*managed memory*), dove tutto quello che ci viene allocato è accessibile sia dalla CPU che dalla GPU, e tramite *un solo* puntatore. Di fatto, la Unified Memory si propone di offrire un modello **single-pointer-to-data**: "eccoti il puntatore, accedi pure alla memoria, che tu sia CPU o GPU!" (in questo assomiglia all'UVA, ma ci sono differenze ovviamente, e a noi interessano soprattutto quelle di leggibilità). Possiamo creare anche un'applicazione ibrida, dove parte della memoria è gestita dal sistema, appunto la managed memory, mentre un'altra parte è gestita dal programmatore, mediante cudaMalloc e cudaFree. L'unico vantaggio significativo insomma è che possiamo usare un solo puntatore ed evitare di fare copie da host a device e viceversa per accedere ai nostri dati. Si possono dichiarare staticamente variabili/array di tipo managed con i qualificatori:

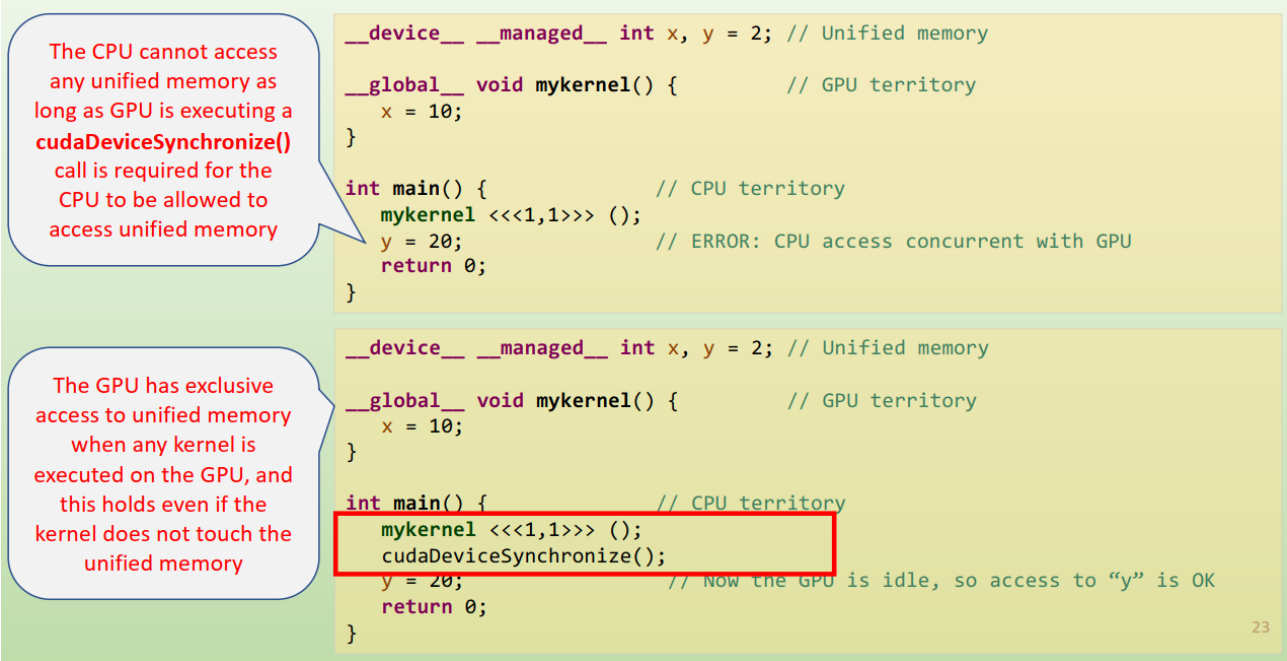
__device__ __managed__ int y; //servono entrambi, sia __device__ che __managed__

E allo stesso tempo si possono dichiarare dinamicamente con la funzione:

```
cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0);
```

Che alloca size byte di memoria managed puntati da devPtr (puntatore che possono usare sia l'host che il device). L'ultimo parametro è al momento inutilizzato ed è riservato per usi futuri. Nota bene che solo l'host può chiamare questa funzione: un kernel non può allocare della memoria managed, ma solo l'host. Una volta allocata, però, ci possono mettere mano tutti.

(dalle slide del Professore) essendo una memoria condivisa, bisogna prestare attenzione nel suo utilizzo, sincronizzando gli accessi ai dati in memoria managed:



3 – Pattern di accesso alla memoria

La memoria globale è il principale punto d'accesso ai dati per tutte quante le applicazioni CUDA, pertanto ottimizzarne l'accesso è un aspetto fondamentale di cui tenere di conto quando si vuole ottimizzare i propri kernel. In CUDA, sappiamo che gli SM ragionano a livello di **warp**: ad un certo istante, tutti i thread dello stesso warp devono fare la stessa cosa (o al massimo qualcuno rimane idle e gli altri fanno la stessa cosa). Quindi, allo stesso modo, ad un certo punto può accadere che tutti i thread dello stesso warp facciano una richiesta di accesso alla memoria. Se i thread si coordinano nell'effettuare questa richiesta, l'effetto complessivo è un più efficiente accesso alla memoria globale, che può essere portato a termine magari anche con una sola transazione. Vediamo ora alcuni patter di accesso alla memoria.

Aligned and Coalesced Access

Tra le SM e la Global Memory possiamo trovare due cache: la L1 (ce n'è una per SM) e la L2 (condivisa da tutto l'hardware). Nel caso della L1, i dati sono organizzati in righe da 128 byte. Ciò significa che, quando un thread vuole leggere anche solo un byte da questa cache, deve per forza leggere l'intera riga da 128 byte che contiene il byte cercato (transazione di memoria da 128 byte). Questo è un male? No, perché se tutti i thread di un warp devono leggere 128 byte adiacenti tra loro (tipo ciascuno deve leggere un intero) che si trovano sulla stessa linea di cache della L1, con una sola transazione io riesco a portare tutti i dati a tutti i thread. Nel caso della L2, invece, le transazioni sono da 32 byte (si possono leggere insomma blocchi da 32 byte alla volta dalla L2). Su alcune architetture, viene data la possibilità al programmatore di abilitare o disabilitare, per una data esecuzione di un'applicazione, la cache L1.

Ora, ricorda che avevamo detto, quando abbiamo parlato della Global Memory, che questa può essere acceduta mediante transazioni da 32, 64 o 128 byte: questi numeri tornano con le transazioni possibili per la L1 e la L2. La L1, quando dovrà leggere valori dalla Global Memory, effettuerà una transazione da 128 byte. La L2, invece, da 32. **Ma sempre transazioni sono: che siano da 32, 64 o 128 byte non importa, si conta il numero di transazioni.**

Quando si accede alla memoria device si dovrebbe sempre cercare di avere accessi alla memoria che siano:

- Aligned (allineati): un accesso è allineato quando il primo indirizzo di una transazione di memoria richiesta è multiplo della granularità della cache usata per soddisfare la richiesta, quindi 32 (se si accede alla L2) o 128 (se si accede alla L1). Se 32 thread in un warp accedono alla memoria, il minimo indirizzo richiesto (e non quello del thread con indice minore (?)) dovrebbe essere multiplo di 32 o 128.
- Coalesced (coalescenti/contigui): un accesso è coalescente quando i 32 thread dello stesso warp vogliono accedere a un chunk di dati contigui, tipo 32 elementi consecutivi di un array.

Il caso ideale è quando l'accesso è allineato e coalescente: il primo indirizzo richiesto è multiplo di 32 o 128, e i dati richiesti dai thread del warp sono tutti consecutivi. Questo tipo di accesso massimizza il throughput di dati acceduti alla memoria globale. In generale, quindi, tante meno sono le transazioni necessarie ad accedere ai dati richiesti dai thread di un warp nello stesso istante, tanto più sfruttiamo la bandwidth disponibile (e tanto più tempo risparmiamo).

Global Memory Reads

Quando vogliamo caricare dati, sui nostri thread, dalla memoria globale, questi vengono cachati nelel cache L1 e L2. Tuttavia, è possibile disabilitare la L1 dando delle indicazioni al compilatore, basta usare il flag:

-Xptxas –dlcm=cg

In questo modo, le richieste di load da parte dei thread saltano la L1 e vanno direttamente in L2. Se c'è un miss, i dati vengono letti dalla globale e portati in L2. Ricordiamo che i dati letti dalla globale vengono letti tramite transazioni da uno, due o quattro segmenti, dove un segmento è lungo 32 byte (quindi, ancora una volta, transazioni da 32, 64 o 128 byte). La L1 può essere esplicitamente abilitata col flag:

-Xptxas –dlcm=ca

Se la L1 è abilitata, le richieste di load dalla Global Memory vengono prima indirizzate alla L1, poi alla L2 e infine alla globale (se necessario ovviamente). Se la L1 è abilitata, le transazioni sono sempre da 128 byte.

A seconda del modello di GPU usata, la L1 potrebbe essere, di default, abilitata o disabilitata. Per esempio, sui dispositivi Fermi è abilitata, mentre sulla K40 e successive è disabilitata. Inoltre, per alcune GPU, come la Kepler K10, K20 e K20x, la L1 non viene usata per cachare dati caricati dalla memoria globale. Si usa invece solo per cachare i register spill della memoria locale.

MEMORY LOAD ACCESS PATTERNS

There are two types of memory loads:

- Cached load (L1 cache enabled)
- Uncached load (L1 cache disabled)

The access pattern for memory loads can be characterized by the following combinations:

- Cached versus uncached: The load is cached if L1 cache is enabled
- Aligned versus misaligned: The load is aligned if the first address of a memory access is a multiple of 32 bytes
- Coalesced versus uncoalesced: The load is coalesced if a warp accesses a contiguous chunk of data

Tip: in generale, se già sappiamo che la nostra applicazione avrà per sua natura un sacco di accessi o misaligned o uncoalesced, o entrambe le cose, può essere saggio disabilitare la L1, in quanto ciò permette di evitare di caricare troppi dati inutili. Se metà thread di un warp vogliono il dato all'indirizzo 27 e l'altra metà all'indirizzo 189, con la L1 dobbiamo effettuare due transazioni da 128 byte ciascuna per portare i due dati

in cache L1, con un'efficienza di 2/256 (carichiamo molti più dati di quanti ce ne servano). Con la L1 disabilitata, invece, abbiamo sempre bisogno di due transazioni, ma carichiamo solo 64 byte, con un'efficienza di 2/64 (abbiamo caricato meno dati inutili).

In generale, per calcolare l'efficienza delle operazioni di load, possiamo usare la seguente metrica:

$$\text{gld\_efficiency} = \frac{\text{Requested Global Memory Load Throughput}}{\text{Required Global Memory Load Throughput}}$$

Che può essere letta come:

efficienza di load da Global Memory = dati che effettivamente mi era necessario caricare / dati che ho dovuto caricare perché le transazioni sono da 128 o 32 byte a partire da multipli di 128 o 32.

Nvprof permette di analizzare un programma proprio sotto questo punto di vista:

```
$ nvprof --devices 0 --metrics gld_transactions ./readSegment 0

Offset    0:  gld_efficiency  100.00%
```

Allo stesso modo, si può anche conoscere il numero di transazioni in Memoria Globale che sono state necessarie:

```
$ nvprof --devices 0 --metrics gld_transactions ./readSegment $OFFSET

Offset    0: gld_transactions 65184
```

Tanto più è alto il valore di gld_transactions, tanti più dati sono stati trasferiti, e tanta più è alto il valore della load throughput.

Abilitare e disabilitare la cache L1 può modificare, anche pesantemente, questi valori.

Global Memory Writes

Abbiamo visto che sia la L1 che la L2 vengono usate durante la load di dati da memoria globale. Tuttavia, per le store, solo la L2 interviene, conservando scritture dirette verso la Global Memory (ad una granularità di 32 byte, essendo la L2). Anche le scritture vengono soddisfatte tramite transazioni, siano queste da uno, due o quattro segmenti alla volta. Se due thread accedono alla stessa regione di memoria da 128 byte, ma gli indirizzi non sono allineati all'interno di una più piccola regione da 64 byte, questa richiesta viene soddisfatta tramite una transazione da 4 segmenti, anziché da due transazioni da 1 segmento ciascuna. Ancora una volta, il numero di transazioni vince.

Esiste un comando nvprof duale del precedente, che usa gst_efficiency anziché gld_efficiency, per misurare le prestazioni delle scritture.

```
$ nvprof --devices 0 --metrics gld_efficiency --metrics gst_efficiency \
    ./writeSegment $OFFSET
```