EBook Gratuito

APPRENDIMENTO cuda

Free unaffiliated eBook created from
Stack Overflow contributors.

#cuda

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cuda

It is an unofficial and free cuda ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cuda.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con cuda

Osservazioni

CUDA è una tecnologia di elaborazione parallela proprietaria NVIDIA e linguaggio di programmazione per le loro GPU.

Le GPU sono macchine altamente parallele in grado di eseguire migliaia di thread leggeri in parallelo. Ogni thread GPU di solito è più lento in esecuzione e il loro contesto è più piccolo. D'altra parte, GPU è in grado di eseguire diverse migliaia di thread in parallelo e anche più simultaneamente (i numeri precisi dipendono dal modello GPU effettivo). CUDA è un dialetto C ++ progettato specificamente per l'architettura della GPU NVIDIA. Tuttavia, a causa delle differenze di architettura, la maggior parte degli algoritmi non può essere semplicemente copiata da un semplice C ++ - essi verrebbero eseguiti, ma sarebbero molto lenti.

Terminologia

- *host* - fa riferimento al normale hardware basato su CPU e ai normali programmi eseguiti in quell'ambiente
- *dispositivo* - fa riferimento a una GPU specifica in cui vengono eseguiti i programmi CUDA. Un singolo host può supportare più dispositivi.
- *kernel* - una funzione che risiede sul dispositivo che può essere richiamata dal codice host.

Struttura del processore fisico

Il processore GPU abilitato CUDA ha la seguente struttura fisica:

- *il chip* - l'intero processore della GPU. Alcune GPU ne hanno due.
- *streamming multiprocessor* (SM) - ogni chip contiene fino a ~ 100 SM, a seconda del modello. Ogni SM opera in modo quasi indipendente da un'altra, utilizzando solo la memoria globale per comunicare tra loro.
- *Core CUDA* : una singola unità di calcolo scalare di un SM. Il loro numero preciso dipende dall'architettura. Ogni core può gestire alcuni thread eseguiti contemporaneamente in una rapida successione (simile all'hyperthreading nella CPU).

Inoltre, ogni SM presenta uno o più *schedulatori di warp* . Ogni schedulatore invia una singola istruzione a diversi core CUDA. Ciò causa efficacemente l'SM per operare in modalità SIMD 32-wide.

Modello di esecuzione CUDA

La struttura fisica della GPU ha un'influenza diretta su come i kernel vengono eseguiti sul dispositivo e su come li programma in CUDA. Il kernel viene invocato con una *configurazione di chiamata* che specifica quanti thread paralleli vengono generati.

- *la griglia* - rappresenta tutti i thread che vengono generati in seguito alla chiamata del kernel. È specificato come uno o due set di *blocchi* dimensionali
- *il blocco* - è un insieme semi-indipendente di *thread* . Ogni blocco è assegnato a un singolo SM. In quanto tale, i blocchi possono comunicare solo attraverso la memoria globale. I blocchi non sono sincronizzati in alcun modo. Se ci sono troppi blocchi, alcuni possono eseguire in sequenza dopo altri. D'altra parte, se le risorse lo consentono, più di un blocco può essere eseguito sullo stesso SM, ma il programmatore non può trarre vantaggio da ciò che sta accadendo (tranne che per l'evidente aumento delle prestazioni).
- *il thread* : una sequenza scalare di istruzioni eseguite da un singolo core CUDA. I thread sono "leggeri" con un contesto minimo, consentendo all'hardware di scambiarli e inserirli rapidamente. A causa del loro numero, i thread CUDA operano con pochi registri a loro assegnati e uno stack molto breve (preferibilmente nessuno!). Per questo motivo, il compilatore CUDA preferisce incorporare tutte le chiamate di funzione per appiattire il kernel in modo che contenga solo salti e loop statici. Le chiamate pon pon di funzioni e le chiamate a metodi virtuali, mentre sono supportate nella maggior parte dei dispositivi più recenti, di solito comportano una maggiore penalità delle prestazioni.

Ogni thread è identificato da un blocco `blockIdx` e indice del thread all'interno del blocco `threadIdx` . Questi numeri possono essere controllati in qualsiasi momento da qualsiasi thread in esecuzione ed è l'unico modo per distinguere un thread da un altro.

Inoltre, i thread sono organizzati in *orditi* , ciascuno contenente esattamente 32 thread. I thread all'interno di un singolo warp vengono eseguiti in una sincronizzazione perfetta, in SIMD fahsion. Thread da diversi orditi, ma all'interno dello stesso blocco possono essere eseguiti in qualsiasi ordine, ma possono essere forzati a sincronizzarsi dal programmatore. Thread da blocchi diversi non possono essere sincronizzati o interagire direttamente in alcun modo.

Organizzazione della memoria

Nella normale programmazione della CPU, l'organizzazione della memoria è solitamente nascosta al programmatore. I programmi tipici agiscono come se ci fosse solo RAM. Tutte le operazioni di memoria, come la gestione dei registri, l'utilizzo di L1-L2-L3-caching, lo scambio su disco, ecc. Sono gestite dal compilatore, dal sistema operativo o dall'hardware stesso.

Questo non è il caso di CUDA. Mentre i modelli di GPU più recenti nascondono parzialmente l'onere, ad esempio attraverso la memoria unificata in CUDA 6, vale comunque la pena di comprendere l'organizzazione per motivi di prestazioni. La struttura di base della memoria CUDA è la seguente:

- *Memoria host* : la RAM normale. Utilizzato principalmente dal codice host, ma anche i nuovi modelli di GPU possono accedervi. Quando un kernel accede alla memoria host, la GPU deve comunicare con la scheda madre, di solito attraverso il connettore PCIe e come tale è relativamente lento.
- *Memoria del dispositivo / memoria globale* : la principale *memoria* off-chip della GPU, disponibile per tutti i thread.
- *La memoria condivisa* , situata in ogni SM, consente un accesso molto più rapido rispetto a quello globale. La memoria condivisa è privata per ogni blocco. I thread all'interno di un

singolo blocco possono usarlo per la comunicazione.
- *Registri* : memoria più veloce, privata e non indirizzabile di ogni thread. In generale, questi non possono essere usati per la comunicazione, ma alcune funzioni intrinseche permettono di mescolare il loro contenuto all'interno di una distorsione.
- *Memoria locale* - la memoria privata di ogni filo che è indirizzabile. Questo è usato per le perdite di registro e gli array locali con indicizzazione variabile. Fisicamente, risiedono nella memoria globale.
- *Memoria texture, memoria costante* - una parte della memoria globale contrassegnata come immutabile per il kernel. Ciò consente alla GPU di utilizzare cache speciali.
- *Cache L2* - on-chip, disponibile per tutti i thread. Data la quantità di thread, la durata prevista di ciascuna riga della cache è molto inferiore rispetto alla CPU. Viene utilizzato principalmente con schemi di accesso di memoria disallineati e parzialmente casuali.
- *Cache L1* - si trova nello stesso spazio della memoria condivisa. Di nuovo, l'importo è piuttosto piccolo, dato il numero di thread che lo utilizzano, quindi non aspettatevi che i dati rimangano a lungo. La cache L1 può essere disabilitata.

Versioni

Capacità di calcolo	Architettura	Nome in codice GPU	Data di rilascio
1.0	Tesla	G80	2006-11-08
1.1	Tesla	G84, G86, G92, G94, G96, G98,	2007-04-17
1.2	Tesla	GT218, GT216, GT215	2009-04-01
1.3	Tesla	GT200, GT200b	2009-04-09
2.0	Fermi	GF100, GF110	2010-03-26
2.1	Fermi	GF104, GF106 GF108, GF114, GF116, GF117, GF119	2010-07-12
3.0	Kepler	GK104, GK106, GK107	2012-03-22
3.2	Kepler	GK20A	2014/04/01
3.5	Kepler	GK110, GK208	2013/02/19
3.7	Kepler	GK210	2014/11/17
5.0	Maxwell	GM107, GM108	2014/02/18
5.2	Maxwell	GM200, GM204, GM206	2014/09/18
5.3	Maxwell	GM20B	2015/04/01
6.0	Pascal	GP100	2016/10/01

Capacità di calcolo	Architettura	Nome in codice GPU	Data di rilascio
6.1	Pascal	GP102, GP104, GP106	2016/05/27

La data di rilascio segna il rilascio della prima GPU che supporta la capacità di calcolo fornita. Alcune date sono approssimative, ad esempio la carta 3.2 è stata rilasciata nel Q2 2014.

Examples

Prerequisiti

Per iniziare a programmare con CUDA, scarica e installa CUDA Toolkit e il driver dello sviluppatore . Il toolkit include `nvcc` , il compilatore NVIDIA CUDA e altro software necessario per sviluppare applicazioni CUDA. Il driver assicura che i programmi GPU vengano eseguiti correttamente su hardware compatibile CUDA , di cui avrete anche bisogno.

È possibile verificare che CUDA Toolkit sia installato correttamente sulla macchina eseguendo `nvcc --version` da una riga di comando. Ad esempio, su una macchina Linux,

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2016 NVIDIA Corporation
Built on Tue_Jul_12_18:28:38_CDT_2016
Cuda compilation tools, release 8.0, V8.0.32
```

restituisce le informazioni del compilatore. Se il comando precedente non ha avuto esito positivo, probabilmente il CUDA Toolkit non è installato, oppure il percorso di `nvcc` (`C:\CUDA\bin` su macchine Windows, `/usr/local/cuda/bin` su sistemi POSIX) non fa parte del tuo Variabile d'ambiente `PATH` .

Inoltre, avrai anche bisogno di un compilatore host che lavori con `nvcc` per compilare e creare programmi CUDA. Su Windows, questo è `cl.exe` , il compilatore Microsoft, fornito con Microsoft Visual Studio. Nei sistemi operativi POSIX sono disponibili altri compilatori, inclusi `gcc` o `g++` . La Guida rapida ufficiale CUDA può dirti quali versioni del compilatore sono supportate sulla tua piattaforma specifica.

Per assicurarci che tutto sia impostato correttamente, compila ed esegui un banale programma CUDA per assicurarti che tutti gli strumenti funzionino correttamente.

```
__global__ void foo() {}

int main()
{
  foo<<<1,1>>>();

  cudaDeviceSynchronize();
  printf("CUDA error: %s\n", cudaGetErrorString(cudaGetLastError()));

  return 0;
```

```
}
```

Per compilare questo programma, copialo in un file chiamato test.cu e compilarlo dalla riga di comando. Ad esempio, su un sistema Linux, il seguente dovrebbe funzionare:

```
$ nvcc -o test
$ ./test
CUDA error: no error
```

Se il programma ha esito positivo senza errori, allora iniziamo a programmare!

Sommare due array con CUDA

Questo esempio illustra come creare un semplice programma che sommerà due array `int` con CUDA.

Un programma CUDA è eterogeneo e consiste di parti eseguite sia su CPU che su GPU.

Le parti principali di un programma che utilizzano CUDA sono simili ai programmi della CPU e consistono in

- Allocazione di memoria per i dati che verranno utilizzati sulla GPU
- Copia dei dati dalla memoria dell'host alla memoria delle GPU
- Richiamo della funzione del kernel per elaborare i dati
- Copia il risultato nella memoria della CPU

Per allocare la memoria dei dispositivi `cudaMalloc` funzione `cudaMalloc` . Per copiare i dati tra dispositivo e host è possibile utilizzare la funzione `cudaMemcpy` . L'ultimo argomento di `cudaMemcpy` specifica la direzione dell'operazione di copia. Ci sono 5 tipi possibili:

- `cudaMemcpyHostToHost` - Host -> Host
- `cudaMemcpyHostToDevice` - Host -> Dispositivo
- `cudaMemcpyDeviceToHost` - Dispositivo -> Host
- `cudaMemcpyDeviceToDevice` - Dispositivo -> Dispositivo
- `cudaMemcpyDefault` : spazio di indirizzi virtuali unificato basato su predefinito

Successivamente viene richiamata la funzione del kernel. Le informazioni tra i triple chevrons sono la configurazione di esecuzione, che stabilisce quanti thread di dispositivo eseguono il kernel in parallelo. Il primo numero (`2` nell'esempio) specifica il numero di blocchi e il secondo (`(size + 1) / 2` nell'esempio): numero di thread in un blocco. Nota che in questo esempio aggiungiamo 1 alla dimensione, in modo da richiedere un thread in più anziché avere un thread responsabile per due elementi.

Poiché la chiamata al kernel è una funzione asincrona, `cudaDeviceSynchronize` viene chiamato ad attendere fino al completamento dell'esecuzione. Gli array di risultati vengono copiati nella memoria dell'host e tutta la memoria allocata sul dispositivo viene liberata con `cudaFree` .

Per definire la funzione viene usato lo specificatore della dichiarazione `__global__` kernel. Questa funzione sarà invocata da ogni thread. Se vogliamo che ogni thread elabori un elemento dell'array

risultante, abbiamo bisogno di un mezzo per distinguere e identificare ciascun thread. CUDA definisce le variabili `blockDim` , `blockIdx` e `threadIdx` . Il `blockDim` variabile predefinitoDim contiene le dimensioni di ciascun blocco di thread come specificato nel secondo parametro di configurazione dell'esecuzione per l'avvio del kernel. Le variabili predefinite `threadIdx` e `blockIdx` contengono l'indice del thread all'interno del suo blocco di thread e il blocco di thread all'interno della griglia, rispettivamente. Nota che, poiché potenzialmente richiediamo un thread in più rispetto agli elementi negli array, dobbiamo passare in `size` per assicurarci di non accedere oltre la fine dell'array.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

__global__ void addKernel(int* c, const int* a, const int* b, int size) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size) {
        c[i] = a[i] + b[i];
    }
}

// Helper function for using CUDA to add vectors in parallel.
void addWithCuda(int* c, const int* a, const int* b, int size) {
    int* dev_a = nullptr;
    int* dev_b = nullptr;
    int* dev_c = nullptr;

    // Allocate GPU buffers for three vectors (two input, one output)
    cudaMalloc((void**)&dev_c, size * sizeof(int));
    cudaMalloc((void**)&dev_a, size * sizeof(int));
    cudaMalloc((void**)&dev_b, size * sizeof(int));

    // Copy input vectors from host memory to GPU buffers.
    cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

    // Launch a kernel on the GPU with one thread for each element.
    // 2 is number of computational blocks and (size + 1) / 2 is a number of threads in a
block
    addKernel<<<2, (size + 1) / 2>>>(dev_c, dev_a, dev_b, size);

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaDeviceSynchronize();

    // Copy output vector from GPU buffer to host memory.
    cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
}

int main(int argc, char** argv) {
    const int arraySize = 5;
    const int a[arraySize] = {  1,  2,  3,  4,  5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };
```

```
    addWithCuda(c, a, b, arraySize);

    printf("{1, 2, 3, 4, 5} + {10, 20, 30, 40, 50} = {%d, %d, %d, %d, %d}\n", c[0], c[1],
c[2], c[3], c[4]);

    cudaDeviceReset();

    return 0;
}
```

Lanciamo un singolo thread CUDA per dire ciao

Questo semplice programma CUDA dimostra come scrivere una funzione che verrà eseguita sulla GPU (ovvero "dispositivo"). La CPU, o "host", crea i thread CUDA chiamando funzioni speciali chiamate "kernel". I programmi CUDA sono programmi C ++ con sintassi aggiuntiva.

Per vedere come funziona, inserisci il seguente codice in un file chiamato `hello.cu` :

```
#include <stdio.h>

// __global__ functions, or "kernels", execute on the device
__global__ void hello_kernel(void)
{
  printf("Hello, world from the device!\n");
}

int main(void)
{
  // greet from the host
  printf("Hello, world from the host!\n");

  // launch a kernel with a single thread to greet from the device
  hello_kernel<<<1,1>>>();

  // wait for the device to finish so that we see the message
  cudaDeviceSynchronize();

  return 0;
}
```

(Si noti che per utilizzare la funzione `printf` sul dispositivo, è necessario un dispositivo con una capacità di calcolo di almeno 2.0. Vedere la panoramica delle versioni per i dettagli.)

Ora compiliamo il programma usando il compilatore NVIDIA ed eseguiamolo:

```
$ nvcc hello.cu -o hello
$ ./hello
Hello, world from the host!
Hello, world from the device!
```

Alcune informazioni aggiuntive sull'esempio precedente:

- `nvcc` sta per "NVIDIA CUDA Compiler". Separa il codice sorgente in componenti host e dispositivo.

- `__global__` è una parola chiave CUDA utilizzata nelle dichiarazioni di funzione che indica che la funzione viene eseguita sul dispositivo GPU e viene chiamata dall'host.
- Le parentesi angolari triple (`<<<` , `>>>`) contrassegnano una chiamata dal codice host al codice dispositivo (chiamato anche "avvio kernel"). I numeri all'interno di queste parentesi quadre indicano il numero di volte da eseguire in parallelo e il numero di thread.

Compilazione ed esecuzione dei programmi di esempio

La guida all'installazione di NVIDIA termina con l'esecuzione dei programmi di esempio per verificare l'installazione di CUDA Toolkit, ma non indica esplicitamente come. Innanzitutto controlla tutti i prerequisiti. Controllare la directory CUDA predefinita per i programmi di esempio. Se non è presente, può essere scaricato dal sito web ufficiale di CUDA. Passare alla directory in cui sono presenti gli esempi.

```
$ cd /path/to/samples/
$ ls
```

Dovresti vedere un risultato simile a:

```
0_Simple     2_Graphics  4_Finance      6_Advanced       bin     EULA.txt
1_Utilities  3_Imaging   5_Simulations  7_CUDALibraries  common  Makefile
```

Assicurarsi che il `Makefile` sia presente in questa directory. Il comando `make` nei sistemi basati su UNIX costruirà tutti i programmi di esempio. In alternativa, accedere a una sottodirectory in cui è presente un altro `Makefile` ed eseguire il comando `make` da lì per creare solo quel campione.

Esegui i due programmi di esempio suggeriti: `deviceQuery` e `bandwidthTest` :

```
$ cd 1_Utilities/deviceQuery/
$ ./deviceQuery
```

L'output sarà simile a quello mostrato di seguito:

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 950M"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 4096 MBytes (4294836224 bytes)
  ( 5) Multiprocessors, (128) CUDA Cores/MP:     640 CUDA Cores
  GPU Max Clock rate:                            1124 MHz (1.12 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096,
4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
```

```
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.5, CUDA Runtime Version = 7.5,
NumDevs = 1, Device0 = GeForce GTX 950M
Result = PASS
```

L'istruzione `Result = PASS` alla fine indica che tutto funziona correttamente. Ora, esegui l'altro programma di esempio sample `bandwidthTest` in modo simile. L'output sarà simile a:

```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: GeForce GTX 950M
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33554432            10604.5

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33554432            10202.0

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(MB/s)
   33554432            23389.7

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU
Boost is enabled.
```

Anche in questo caso, la dichiarazione `Result = PASS` indica che tutto è stato eseguito correttamente. Tutti gli altri programmi di esempio possono essere eseguiti in modo simile.

Leggi Iniziare con cuda online: https://riptutorial.com/it/cuda/topic/1860/iniziare-con-cuda

Capitolo 2: Comunicazione tra blocchi

Osservazioni

I blocchi in CUDA operano in modo semi indipendente. Non esiste un modo sicuro per sincronizzarli tutti. Tuttavia, ciò non significa che non possano interagire tra loro in alcun modo.

Examples

Last block guard

Considera una griglia che lavori su qualche compito, ad esempio una riduzione parallela. Inizialmente, ogni blocco può fare il suo lavoro in modo indipendente, producendo alcuni risultati parziali. Alla fine, tuttavia, i risultati parziali devono essere combinati e riuniti. Un tipico esempio è un algoritmo di riduzione su un big data.

Un approccio tipico consiste nel richiamare due kernel, uno per il calcolo parziale e l'altro per l'unione. Tuttavia, se la fusione può essere eseguita in modo efficiente da un singolo blocco, è necessaria solo una chiamata del kernel. Ciò è ottenuto da una guardia `lastBlock` definita come:

2.0

```
__device__ bool lastBlock(int* counter) {
  __threadfence(); //ensure that partial result is visible by all blocks
  int last = 0;
  if (threadIdx.x == 0)
    last = atomicAdd(counter, 1);
  return __syncthreads_or(last == gridDim.x-1);
}
```

1.1

```
__device__ bool lastBlock(int* counter) {
  __shared__ int last;
  __threadfence(); //ensure that partial result is visible by all blocks
  if (threadIdx.x == 0) {
    last = atomicAdd(counter, 1);
  }
  __syncthreads();
  return last == gridDim.x-1;
}
```

Con una tale guardia l'ultimo blocco è garantito per vedere tutti i risultati prodotti da tutti gli altri blocchi e può eseguire la fusione.

```
__device__ void computePartial(T* out) { ... }
__device__ void merge(T* partialResults, T* out) { ... }

__global__ void kernel(int* counter, T* partialResults, T* finalResult) {
    computePartial(&partialResults[blockIdx.x]);
```

```
    if (lastBlock(counter)) {
      //this is executed by all threads of the last block only
      merge(partialResults,finalResult);
    }
}
```

ipotesi:

- Il contatore deve essere un puntatore di memoria globale, inizializzato a 0 *prima che* il kernel venga richiamato.
- La funzione `lastBlock` è invocata in modo uniforme da tutti i thread in tutti i blocchi
- Il kernel è invocato in una griglia monodimensionale (per semplicità dell'esempio)
- `T` identifica qualsiasi tipo che ti piace, ma l'esempio non intende essere un modello in senso C ++

Coda di lavoro globale

Considera una serie di oggetti di lavoro. Il tempo necessario per il completamento di ciascun elemento di lavoro varia notevolmente. Per bilanciare la distribuzione del lavoro tra i blocchi, può essere prudente per ogni blocco recuperare l'elemento successivo solo quando il precedente è completo. Ciò è in contrasto con l'attribuzione a priori di elementi a blocchi.

```
class WorkQueue {
private:
  WorkItem* gItems;
  size_t totalSize;
  size_t current;
public:
  __device__ WorkItem& fetch() {
    __shared__ WorkItem item;
    if (threadIdx.x == 0) {
      size_t itemIdx = atomicAdd(current,1);
      if (itemIdx<totalSize)
        item = gItems[itemIdx];
      else
        item = WorkItem::none();
    }
    __syncthreads();
    return item; //returning reference to smem - ok
  }
}
```

ipotesi:

- L'oggetto WorkQueue e l'array gItem risiedono nella memoria globale
- Nessun nuovo oggetto di lavoro viene aggiunto all'oggetto WorkQueue nel kernel che sta recuperando da esso
- `WorkItem` è una piccola rappresentazione dell'assegnazione del lavoro, ad esempio un puntatore a un altro oggetto
- `WorkItem::none()` funzione membro statico `WorkItem::none()` crea un oggetto `WorkItem` che non rappresenta affatto lavoro
- `WorkQueue::fetch()` deve essere chiamato uniformemente da tutti i thread nel blocco

- Non ci sono 2 invocazioni di `WorkQueue::fetch()` senza un altro `__syncthreads()` in mezzo. Altrimenti apparirà una condizione di gara!

L'esempio non include come inizializzare `WorkQueue` o popolarlo. È fatto da un altro kernel o codice CPU e dovrebbe essere abbastanza semplice.

Leggi Comunicazione tra blocchi online: https://riptutorial.com/it/cuda/topic/4978/comunicazione-tra-blocchi

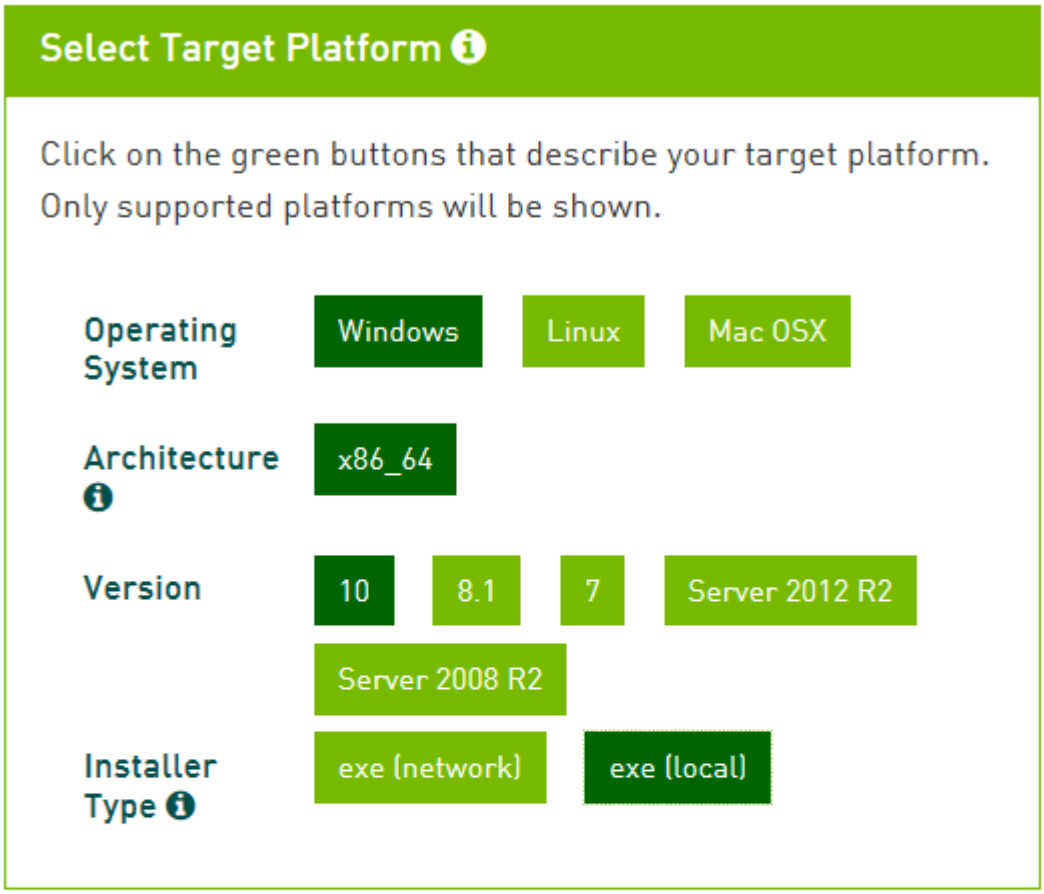
Capitolo 3: Installare cuda

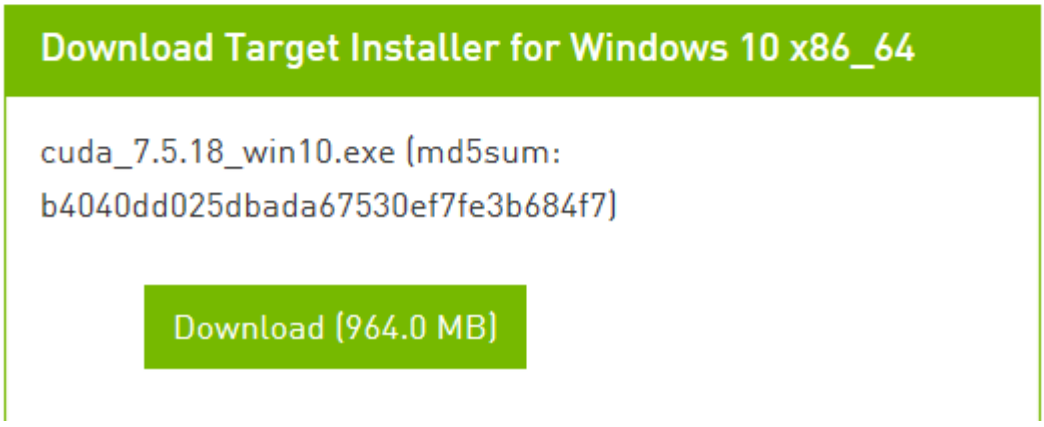
Osservazioni

Per installare CUDA toolkit su Windows, pugno è necessario installare una versione corretta di Visual Studio. Visual Studio 2013 dovrebbe essere installato se hai intenzione di installare CUDA 7.0 o 7.5. Visual Studio 2015 è supportato per CUDA 8.0 e successivi.

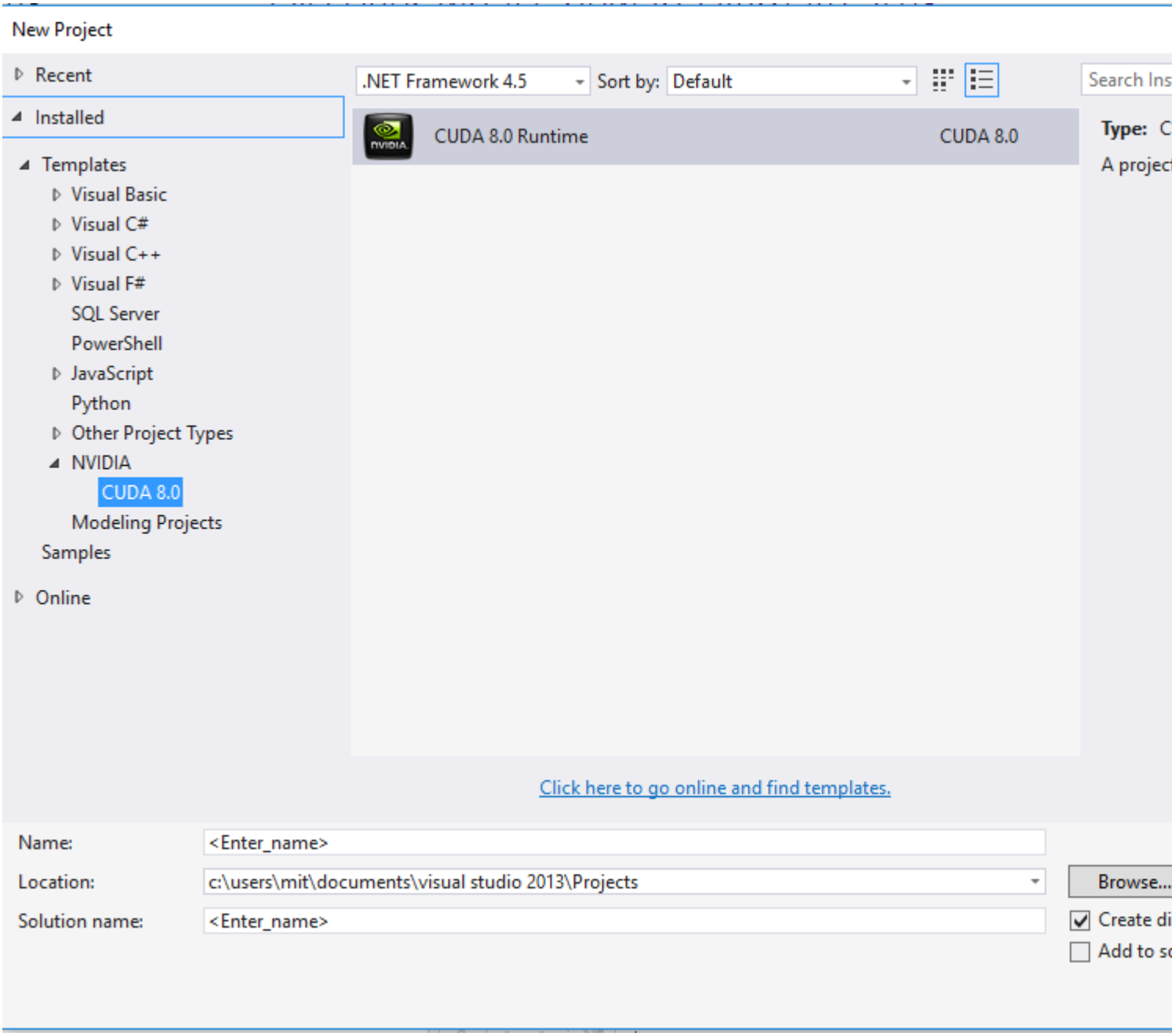
Quando hai una versione corretta di VS sul tuo sistema, è ora di scaricare e installare il toolkit CUDA. Segui questo link per trovare una versione del toolkit CUDA che stai cercando: archivio del toolkit CUDA

Nella pagina di download è necessario scegliere la versione di Windows sul computer di destinazione e il tipo di programma di installazione (selezionare locale).





Dopo aver scaricato il file exe, devi estrarlo ed eseguire `setup.exe` . Al termine dell'installazione, aprire un nuovo progetto e scegliere NVIDIA> CUDAX.X dai modelli.



Ricorda che l'estensione dei file di origine CUDA è `.cu` . È possibile scrivere codici host e dispositivo sulla stessa origine.

Examples

Codice CUDA molto semplice

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "cuda.h"
#include <device_functions.h>
#include <cuda_runtime_api.h>

#include<stdio.h>
#include <cmath>
#include<stdlib.h>
```

```
#include<iostream>
#include <iomanip>



using namespace std;
typedef unsigned int uint;


const uint N = 1e6;

__device__ uint Val2[N];

__global__ void set0()
{
    uint index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if (index < N)
    {
        Val2[index] = 0;
    }

}


int main()
{
    int numThreads = 512;
    uint numBlocks = (uint)ceil(N / (double)numThreads);
    set0 << < numBlocks, numThreads >> >();

    return 0;
}
```

Leggi Installare cuda online: https://riptutorial.com/it/cuda/topic/10949/installare-cuda

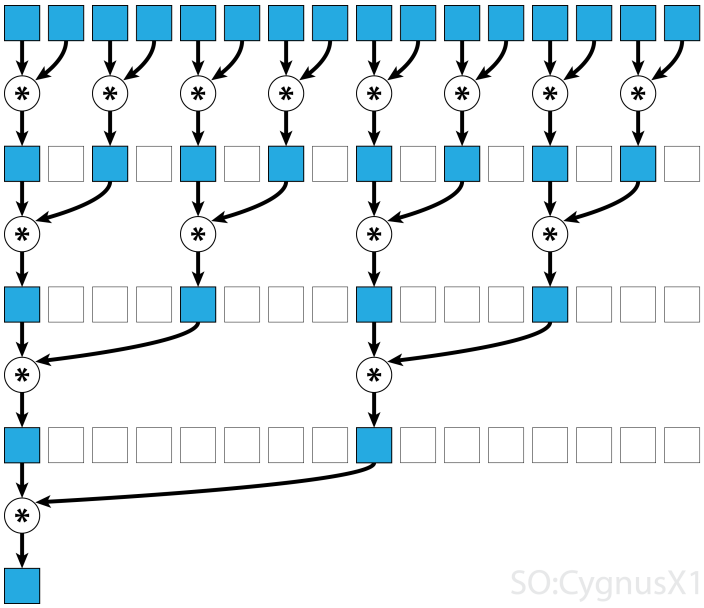
Capitolo 4: Riduzione parallela (es. Come sommare una matrice)

Osservazioni

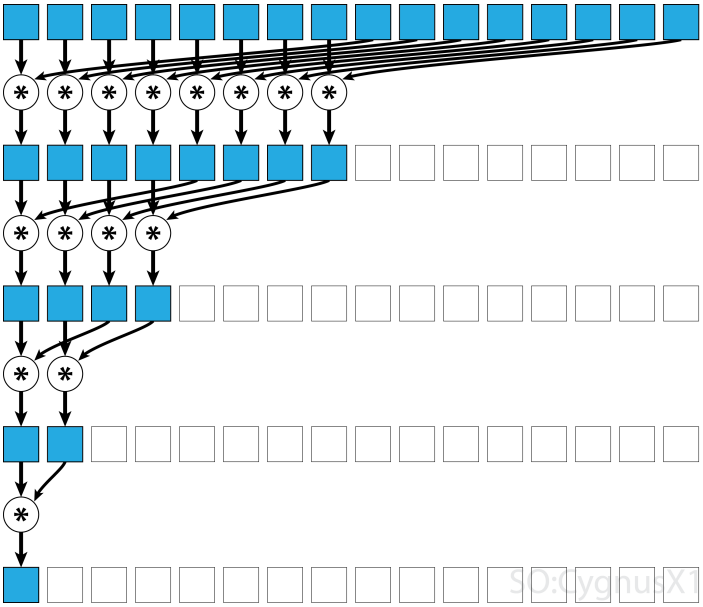
L'algoritmo di riduzione parallela si riferisce in genere a un algoritmo che combina una serie di elementi, producendo un singolo risultato. I problemi tipici che rientrano in questa categoria sono:

- riassumere tutti gli elementi in un array
- trovare un massimo in un array

In generale, la riduzione parallela può essere applicata per qualsiasi operatore associativo binario, cioè `(A*B)*C = A*(B*C)` . Con tale operatore *, l'algoritmo di riduzione parallela raggruppa ripetutamente gli argomenti dell'array in coppie. Ogni coppia è calcolata in parallelo con altre, dimezzando la dimensione complessiva dell'array in un solo passaggio. Il processo viene ripetuto fino a quando esiste un solo elemento.



Se l'operatore è commutativo (cioè `A*B = B*A`) oltre ad essere associativo, l'algoritmo può accoppiarsi in un modello diverso. Dal punto di vista teorico non fa alcuna differenza, ma in pratica fornisce un modello di accesso alla memoria migliore:



Non tutti gli operatori associativi sono commutativi, ad esempio, prendere la moltiplicazione della matrice.

Examples

Riduzione parallela a blocco singolo per operatore commutativo

L'approccio più semplice alla riduzione parallela di CUDA consiste nell'assegnare un singolo blocco per eseguire l'operazione:

```
static const int arraySize = 10000;
static const int blockSize = 1024;

__global__ void sumCommSingleBlock(const int *a, int *out) {
    int idx = threadIdx.x;
    int sum = 0;
    for (int i = idx; i < arraySize; i += blockSize)
        sum += a[i];
    __shared__ int r[blockSize];
    r[idx] = sum;
    __syncthreads();
    for (int size = blockSize/2; size>0; size/=2) { //uniform
        if (idx<size)
            r[idx] += r[idx+size];
        __syncthreads();
    }
    if (idx == 0)
        *out = r[0];
}

...

sumCommSingleBlock<<<1, blockSize>>>(dev_a, dev_out);
```

Questo è più fattibile quando la dimensione dei dati non è molto grande (attorno a qualche elemento di migliaia). Questo di solito accade quando la riduzione fa parte di un programma CUDA più grande. Se l'input corrisponde a `blockSize` sin dall'inizio, il primo ciclo `for` può essere

completamente rimosso.

Nota che, in primo luogo, quando ci sono più elementi dei thread, aggiungiamo le cose in modo completamente indipendente. Solo quando il problema viene ridotto a `blockSize` , i trigger di riduzione paralleli effettivi. Lo stesso codice può essere applicato a qualsiasi altro operatore commutativo, associativo, come la moltiplicazione, il minimo, il massimo, ecc.

Si noti che l'algoritmo può essere reso più veloce, ad esempio utilizzando una riduzione parallela a livello di curvatura.

Riduzione parallela a blocco singolo per operatore non commutativo

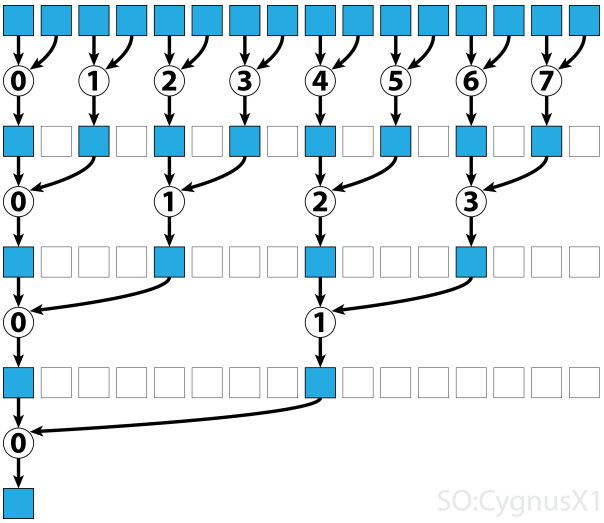
La riduzione parallela per un operatore non commutativo è un po 'più complicata, rispetto alla versione commutativa. Nell'esempio usiamo ancora un'aggiunta su interi per la semplicità. Potrebbe essere sostituito, ad esempio, con la moltiplicazione della matrice che è realmente non commutativa. Nota, nel fare ciò, 0 dovrebbe essere sostituito da un elemento neutro della moltiplicazione, cioè una matrice di identità.

```
static const int arraySize = 1000000;
static const int blockSize = 1024;

__global__ void sumNoncommSingleBlock(const int *gArr, int *out) {
    int thIdx = threadIdx.x;
    __shared__ int shArr[blockSize*2];
    __shared__ int offset;
    shArr[thIdx] = thIdx<arraySize ? gArr[thIdx] : 0;
    if (thIdx == 0)
        offset = blockSize;
    __syncthreads();
    while (offset < arraySize) { //uniform
        shArr[thIdx + blockSize] = thIdx+offset<arraySize ? gArr[thIdx+offset] : 0;
        __syncthreads();
        if (thIdx == 0)
            offset += blockSize;
        int sum = shArr[2*thIdx] + shArr[2*thIdx+1];
        __syncthreads();
        shArr[thIdx] = sum;
    }
    __syncthreads();
    for (int stride = 1; stride<blockSize; stride*=2) { //uniform
        int arrIdx = thIdx*stride*2;
        if (arrIdx+stride<blockSize)
            shArr[arrIdx] += shArr[arrIdx+stride];
        __syncthreads();
    }
    if (thIdx == 0)
        *out = shArr[0];
}

...

sumNoncommSingleBlock<<<1, blockSize>>>(dev_a, dev_out);
```

Nel primo ciclo while viene eseguito finché ci sono più elementi di input dei thread. In ogni iterazione viene eseguita una singola riduzione e il risultato viene compresso nella prima metà

dell'array `shArr` . La seconda metà viene quindi riempita con nuovi dati.

Una volta caricati tutti i dati da `gArr` , viene eseguito il secondo ciclo. Ora non comprimiamo più il risultato (che costa un `__syncthreads()`) extra. In ogni passo il thread n accede al `2*n` -esimo elemento attivo e lo aggiunge con `2*n+1` -esimo elemento:



Esistono molti modi per ottimizzare ulteriormente questo semplice esempio, ad esempio attraverso la riduzione del livello di curvatura e rimuovendo i conflitti di memoria del banco condiviso.

Riduzione parallela multiblocco per operatore commutativo

L'approccio a blocchi multipli per la riduzione parallela di CUDA rappresenta una sfida aggiuntiva rispetto all'approccio a blocco singolo, poiché i blocchi sono limitati nella comunicazione. L'idea è di lasciare che ogni blocco calcoli una parte dell'array di input e quindi un blocco finale per unire tutti i risultati parziali. Per fare ciò è possibile avviare due kernel, creando implicitamente un punto di sincronizzazione a livello di griglia.

```
static const int wholeArraySize = 100000000;
static const int blockSize = 1024;
static const int gridSize = 24; //this number is hardware-dependent; usually #SM*2 is a good
number.

__global__ void sumCommMultiBlock(const int *gArr, int arraySize, int *gOut) {
    int thIdx = threadIdx.x;
    int gthIdx = thIdx + blockIdx.x*blockSize;
    const int gridSize = blockSize*gridDim.x;
    int sum = 0;
    for (int i = gthIdx; i < arraySize; i += gridSize)
        sum += gArr[i];
    __shared__ int shArr[blockSize];
    shArr[thIdx] = sum;
    __syncthreads();
    for (int size = blockSize/2; size>0; size/=2) { //uniform
        if (thIdx<size)
            shArr[thIdx] += shArr[thIdx+size];
        __syncthreads();
    }
    if (thIdx == 0)
```

```
        gOut[blockIdx.x] = shArr[0];
}

__host__ int sumArray(int* arr) {
    int* dev_arr;
    cudaMalloc((void**)&dev_arr, wholeArraySize * sizeof(int));
    cudaMemcpy(dev_arr, arr, wholeArraySize * sizeof(int), cudaMemcpyHostToDevice);

    int out;
    int* dev_out;
    cudaMalloc((void**)&dev_out, sizeof(int)*gridSize);

    sumCommMultiBlock<<<gridSize, blockSize>>>(dev_arr, wholeArraySize, dev_out);
    //dev_out now holds the partial result
    sumCommMultiBlock<<<1, blockSize>>>(dev_out, gridSize, dev_out);
    //dev_out[0] now holds the final result
    cudaDeviceSynchronize();

    cudaMemcpy(&out, dev_out, sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dev_arr);
    cudaFree(dev_out);
    return out;
}
```

Uno idealmente vuole lanciare abbastanza blocchi per saturare tutti i multiprocessori sulla GPU in piena occupazione. Il superamento di questo numero, in particolare il lancio di tutti i thread quanti sono gli elementi nell'array, è controproducente. In questo modo non aumenta più la potenza di calcolo raw, ma impedisce l'utilizzo del primo loop molto efficiente.

È anche possibile ottenere lo stesso risultato usando un singolo kernel, con l'aiuto di last-block guard :

```
static const int wholeArraySize = 100000000;
static const int blockSize = 1024;
static const int gridSize = 24;

__device__ bool lastBlock(int* counter) {
    __threadfence(); //ensure that partial result is visible by all blocks
    int last = 0;
    if (threadIdx.x == 0)
        last = atomicAdd(counter, 1);
    return __syncthreads_or(last == gridDim.x-1);
}

__global__ void sumCommMultiBlock(const int *gArr, int arraySize, int *gOut, int*
lastBlockCounter) {
    int thIdx = threadIdx.x;
    int gthIdx = thIdx + blockIdx.x*blockSize;
    const int gridSize = blockSize*gridDim.x;
    int sum = 0;
    for (int i = gthIdx; i < arraySize; i += gridSize)
        sum += gArr[i];
    __shared__ int shArr[blockSize];
    shArr[thIdx] = sum;
    __syncthreads();
    for (int size = blockSize/2; size>0; size/=2) { //uniform
        if (thIdx<size)
            shArr[thIdx] += shArr[thIdx+size];
```

```
        __syncthreads();
    }
    if (thIdx == 0)
        gOut[blockIdx.x] = shArr[0];
    if (lastBlock(lastBlockCounter)) {
        shArr[thIdx] = thIdx<gridSize ? gOut[thIdx] : 0;
        __syncthreads();
        for (int size = blockSize/2; size>0; size/=2) { //uniform
            if (thIdx<size)
                shArr[thIdx] += shArr[thIdx+size];
            __syncthreads();
        }
        if (thIdx == 0)
            gOut[0] = shArr[0];
    }
}

__host__ int sumArray(int* arr) {
    int* dev_arr;
    cudaMalloc((void**)&dev_arr, wholeArraySize * sizeof(int));
    cudaMemcpy(dev_arr, arr, wholeArraySize * sizeof(int), cudaMemcpyHostToDevice);

    int out;
    int* dev_out;
    cudaMalloc((void**)&dev_out, sizeof(int)*gridSize);

    int* dev_lastBlockCounter;
    cudaMalloc((void**)&dev_lastBlockCounter, sizeof(int));
    cudaMemset(dev_lastBlockCounter, 0, sizeof(int));

    sumCommMultiBlock<<<gridSize, blockSize>>>(dev_arr, wholeArraySize, dev_out,
dev_lastBlockCounter);
    cudaDeviceSynchronize();

    cudaMemcpy(&out, dev_out, sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dev_arr);
    cudaFree(dev_out);
    return out;
}
```

Si noti che il kernel può essere reso più veloce, ad esempio usando una riduzione parallela a livello di curvatura.

Riduzione parallela multiblocco per operatore non commutabile

L'approccio a blocchi multipli per la riduzione parallela è molto simile all'approccio a blocco singolo. L'array di input globale deve essere suddiviso in sezioni, ciascuna ridotta da un singolo blocco. Quando si ottiene un risultato parziale da ciascun blocco, un blocco finale li riduce per ottenere il risultato finale.

- `sumNoncommSingleBlock` è spiegato più dettagliatamente nell'esempio di riduzione a blocco singolo.
- `lastBlock` accetta solo l'ultimo blocco che lo raggiunge. Se vuoi evitare questo, puoi dividere il kernel in due chiamate separate.

```
static const int wholeArraySize = 100000000;
```

```
static const int blockSize = 1024;
static const int gridSize = 24; //this number is hardware-dependent; usually #SM*2 is a good
number.

__device__ bool lastBlock(int* counter) {
    __threadfence(); //ensure that partial result is visible by all blocks
    int last = 0;
    if (threadIdx.x == 0)
        last = atomicAdd(counter, 1);
    return __syncthreads_or(last == gridDim.x-1);
}

__device__ void sumNoncommSingleBlock(const int* gArr, int arraySize, int* out) {
    int thIdx = threadIdx.x;
    __shared__ int shArr[blockSize*2];
    __shared__ int offset;
    shArr[thIdx] = thIdx<arraySize ? gArr[thIdx] : 0;
    if (thIdx == 0)
        offset = blockSize;
    __syncthreads();
    while (offset < arraySize) { //uniform
        shArr[thIdx + blockSize] = thIdx+offset<arraySize ? gArr[thIdx+offset] : 0;
        __syncthreads();
        if (thIdx == 0)
            offset += blockSize;
        int sum = shArr[2*thIdx] + shArr[2*thIdx+1];
        __syncthreads();
        shArr[thIdx] = sum;
    }
    __syncthreads();
    for (int stride = 1; stride<blockSize; stride*=2) { //uniform
        int arrIdx = thIdx*stride*2;
        if (arrIdx+stride<blockSize)
            shArr[arrIdx] += shArr[arrIdx+stride];
        __syncthreads();
    }
    if (thIdx == 0)
        *out = shArr[0];
}

__global__ void sumNoncommMultiBlock(const int* gArr, int* out, int* lastBlockCounter) {
    int arraySizePerBlock = wholeArraySize/gridSize;
    const int* gArrForBlock = gArr+blockIdx.x*arraySizePerBlock;
    int arraySize = arraySizePerBlock;
    if (blockIdx.x == gridSize-1)
        arraySize = wholeArraySize - blockIdx.x*arraySizePerBlock;
    sumNoncommSingleBlock(gArrForBlock, arraySize, &out[blockIdx.x]);
    if (lastBlock(lastBlockCounter))
        sumNoncommSingleBlock(out, gridSize, out);
}
```

Uno idealmente vuole lanciare abbastanza blocchi per saturare tutti i multiprocessori sulla GPU in piena occupazione. Il superamento di questo numero, in particolare il lancio di tutti i thread quanti sono gli elementi nell'array, è controproducente. In questo modo non aumenta più la potenza di calcolo raw, ma impedisce l'utilizzo del primo loop molto efficiente.

Riduzione parallela a singolo ordito per operatore commutativo

A volte la riduzione deve essere eseguita su una scala molto piccola, come parte di un kernel CUDA più grande. Supponiamo ad esempio che i dati di input contengano esattamente 32 elementi: il numero di fili in un ordito. In tale scenario può essere assegnato un singolo ordito per eseguire la riduzione. Dato che il warp viene eseguito in una perfetta sincronizzazione, molte istruzioni `__syncthreads()` possono essere rimosse - se confrontate con una riduzione a livello di blocco.

```
static const int warpSize = 32;

__device__ int sumCommSingleWarp(volatile int* shArr) {
    int idx = threadIdx.x % warpSize; //the lane index in the warp
    if (idx<16) shArr[idx] += shArr[idx+16];
    if (idx<8) shArr[idx] += shArr[idx+8];
    if (idx<4) shArr[idx] += shArr[idx+4];
    if (idx<2) shArr[idx] += shArr[idx+2];
    if (idx==0) shArr[idx] += shArr[idx+1];
    return shArr[0];
}
```

`shArr` è preferibilmente un array nella memoria condivisa. Il valore dovrebbe essere lo stesso per tutti i thread nel warp. Se `sumCommSingleWarp` è chiamato da più `shArr` , `shArr` dovrebbe essere diverso tra i warp (lo stesso all'interno di ogni warp).

L'argomento `shArr` è contrassegnato come `volatile` per garantire che le operazioni sull'array vengano effettivamente eseguite dove indicato. Altrimenti, l'assegnazione ripetitiva a `shArr[idx]` può essere ottimizzata come assegnazione a un registro, poiché solo l'asserzione finale è un vero e proprio negozio di `shArr` . Quando ciò accade, i compiti immediati non sono visibili ad altri thread, producendo risultati errati. Nota che puoi passare un normale array non volatile come argomento di uno volatile, come quando passi non-const come parametro const.

Se non ci si preoccupa del contenuto di `shArr[1..31]` dopo la riduzione, si può semplificare ulteriormente il codice:

```
static const int warpSize = 32;

__device__ int sumCommSingleWarp(volatile int* shArr) {
    int idx = threadIdx.x % warpSize; //the lane index in the warp
    if (idx<16) {
      shArr[idx] += shArr[idx+16];
      shArr[idx] += shArr[idx+8];
      shArr[idx] += shArr[idx+4];
      shArr[idx] += shArr[idx+2];
      shArr[idx] += shArr[idx+1];
    }
    return shArr[0];
}
```

In questa configurazione abbiamo rimosso molti `if` le condizioni. I thread aggiuntivi eseguono alcune aggiunte non necessarie, ma non ci interessa più i contenuti che producono. Poiché gli orditi vengono eseguiti in modalità SIMD, in realtà non risparmiamo in tempo facendo in modo che quei thread non facciano nulla. D'altra parte, la valutazione delle condizioni richiede tempi relativamente lunghi, poiché il corpo di queste affermazioni `if` è così piccolo. L'istruzione `if`

iniziale può essere rimossa anche se `shArr[32..47]` è riempito con 0.

La riduzione del livello di curvatura può essere utilizzata anche per aumentare la riduzione a livello di blocco:

```
__global__ void sumCommSingleBlockWithWarps(const int *a, int *out) {
    int idx = threadIdx.x;
    int sum = 0;
    for (int i = idx; i < arraySize; i += blockSize)
        sum += a[i];
    __shared__ int r[blockSize];
    r[idx] = sum;
    sumCommSingleWarp(&r[idx & ~(warpSize-1)]);
    __syncthreads();
    if (idx<warpSize) { //first warp only
        r[idx] = idx*warpSize<blockSize ? r[idx*warpSize] : 0;
        sumCommSingleWarp(r);
        if (idx == 0)
            *out = r[0];
    }
}
```

L'argomento `&r[idx & ~(warpSize-1)]` è fondamentalmente `r + warpIdx*32` . Questo suddivide efficacemente l'array `r` in blocchi di 32 elementi, e ciascun blocco viene assegnato a distorsione separata.

Riduzione parallela a singolo ordito per operatore non commutabile

A volte la riduzione deve essere eseguita su una scala molto piccola, come parte di un kernel CUDA più grande. Supponiamo ad esempio che i dati di input contengano esattamente 32 elementi: il numero di fili in un ordito. In tale scenario può essere assegnato un singolo ordito per eseguire la riduzione. Dato che il warp viene eseguito in una perfetta sincronizzazione, molte istruzioni `__syncthreads()` possono essere rimosse - se confrontate con una riduzione a livello di blocco.

```
static const int warpSize = 32;

__device__ int sumNoncommSingleWarp(volatile int* shArr) {
    int idx = threadIdx.x % warpSize; //the lane index in the warp
    if (idx%2 == 0) shArr[idx] += shArr[idx+1];
    if (idx%4 == 0) shArr[idx] += shArr[idx+2];
    if (idx%8 == 0) shArr[idx] += shArr[idx+4];
    if (idx%16 == 0) shArr[idx] += shArr[idx+8];
    if (idx == 0) shArr[idx] += shArr[idx+16];
    return shArr[0];
}
```

`shArr` è preferibilmente un array nella memoria condivisa. Il valore dovrebbe essere lo stesso per tutti i thread nel warp. Se `sumCommSingleWarp` è chiamato da più `shArr` , `shArr` dovrebbe essere diverso tra i warp (lo stesso all'interno di ogni warp).

L'argomento `shArr` è contrassegnato come `volatile` per garantire che le operazioni sull'array vengano effettivamente eseguite dove indicato. Altrimenti, l'assegnazione ripetitiva a `shArr[idx]`

può essere ottimizzata come assegnazione a un registro, poiché solo l'asserzione finale è un vero e proprio negozio di `shArr` . Quando ciò accade, i compiti immediati non sono visibili ad altri thread, producendo risultati errati. Nota che puoi passare un normale array non volatile come argomento di uno volatile, come quando passi non-const come parametro const.

Se uno non si preoccupa del contenuto finale di `shArr[1..31]` e può pad `shArr[32..47]` con zeri, si può semplificare il codice di cui sopra:

```
static const int warpSize = 32;

__device__ int sumNoncommSingleWarpPadded(volatile int* shArr) {
    //shArr[32..47] == 0
    int idx = threadIdx.x % warpSize; //the lane index in the warp
    shArr[idx] += shArr[idx+1];
    shArr[idx] += shArr[idx+2];
    shArr[idx] += shArr[idx+4];
    shArr[idx] += shArr[idx+8];
    shArr[idx] += shArr[idx+16];
    return shArr[0];
}
```

In questa configurazione abbiamo rimosso tutto `if` le condizioni, che costituiscono circa la metà delle istruzioni. I thread aggiuntivi eseguono alcune aggiunte non necessarie, memorizzando il risultato in celle di `shArr` che alla fine non hanno alcun impatto sul risultato finale. Poiché gli orditi vengono eseguiti in modalità SIMD, in realtà non risparmiamo in tempo facendo in modo che quei thread non facciano nulla.

Riduzione parallela a singolo ordito utilizzando solo registri

In genere, la riduzione viene eseguita su un array globale o condiviso. Tuttavia, quando la riduzione viene eseguita su una scala molto piccola, come parte di un kernel CUDA più grande, può essere eseguita con un singolo warp. Quando ciò accade, su Keppler o architetture superiori (CC> = 3.0), è possibile utilizzare le funzioni warp-shuffle per evitare di utilizzare la memoria condivisa.

Supponiamo ad esempio che ogni filo in un curvino abbia un unico valore di dati di input. Tutti i thread hanno 32 elementi, che dobbiamo riassumere (o eseguire altre operazioni associative)

```
__device__ int sumSingleWarpReg(int value) {
    value += __shfl_down(value, 1);
    value += __shfl_down(value, 2);
    value += __shfl_down(value, 4);
    value += __shfl_down(value, 8);
    value += __shfl_down(value, 16);
    return __shfl(value,0);
}
```

Questa versione funziona sia per operatori commutativi che non commutativi.

Leggi Riduzione parallela (es. Come sommare una matrice) online:
https://riptutorial.com/it/cuda/topic/6566/riduzione-parallela--es--come-sommare-una-matrice-

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con cuda	Community, CygnusX1, Dev-iL, harrism, havogt, infinite.potential, Jared Hoberock, Ken Y-N, Marco13, NikolayKondratyev, Tejus Prasad
2	Comunicazione tra blocchi	CygnusX1, tera
3	Installare cuda	Mo Sani
4	Riduzione parallela (es. Come sommare una matrice)	CygnusX1