Riccardo Fontanini

Moltiplicazione tra matrici

# Consegna

Implementare un programma C che sfrutti la GPU per eseguire una moltiplicazione tra due matrici utilizzando la shared memory. Inoltre adattare il programma perché sfrutti più di una GPU per eseguire il calcolo.

# Moltiplicazione tra matrici

Date due matrici A e B, nel nostro caso quadrate con dimensione N, si definisce il prodotto matriciale in questo modo

Con

Per rendere immediato il risultato della moltiplicazione sono state moltiplicate una matrice generica A con la matrice identità, il risultato quindi dovrà ritornare la matrice originaria A. In caso contrario è facile riscontrare un errore perché la matrice di ritorno sarà differente dalla matrice A.

# Sistema per il test

I test sono stati eseguiti su una macchina server che sfrutta processori Intel Xeon CPU E5-2603, una scheda grafica NVIDIA K40 ed una GeForce GTX 750.

# Problematiche

La shared memory per blocco, per la GPU considerata (Tesla K40), è di 49152 byte. Tale quantità di memoria non provoca particolari problematiche se la dimensioni delle matrici da moltiplicare è piccola, per esempio, se N = 10:

Ma se la dimensione inizia a diventare più importante, per esempio N = 100:

Valore che eccede la quantità massima di memoria shared utilizzabile in un blocco. Quindi non sarebbe possibile fare la moltiplicazione tra due matrici con tale dimensione in un singolo blocco sfruttando la shared memory.

Soluzione:

Per sfruttare l’elevato parallelismo all’interno di uno stesso blocco, senza eccedere la shared memory, è stata impiegata la moltiplicazione per sottomatrici. Partendo da una matrice quadrata, è possibile suddividerla in varie sottomatrici:

Dove A,B,C,D sono loro stesse matrici, per esempio:

Con la matrice totale quindi:

Quando i blocchi hanno stessa dimensione e sono quadrati, allora si possono moltiplicare similmente alla convenzionale moltiplicazione tra matrici, cioè:

In questo caso se la matrice è multipla di 32 elementi:

C00 = A00 \* B00 + A01 \* B10 + A02 \* B20

C01 = A00 \* B01 + A01 \* B11 + A02 \* B21

C10 = A10 \* B00 + A11 \* B10 + A12 \* B20

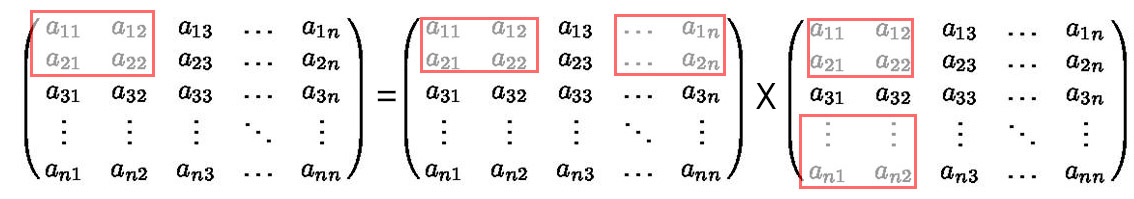
C11 = A10 \* B01 + A11 \* B11 + A12 \* B21

E così via..

Quindi è possibile spezzare l’operazione di moltiplicazione di una matrice con dimensione N (per semplicità si è assunto N multiplo di 32) nella somma di N/32 sottomatrici ricavate a loro volta da prodotti delle sottomatrici che compongono A e B, come descritto precedentemente. Quindi ogni blocco della griglia elabora una moltiplicazione tra due matrici di dimensione 32x32 e lo fa usando la shared memory, quindi necessita di:

Valore che è al di sotto della massima quantità di memoria disponibile per blocco.

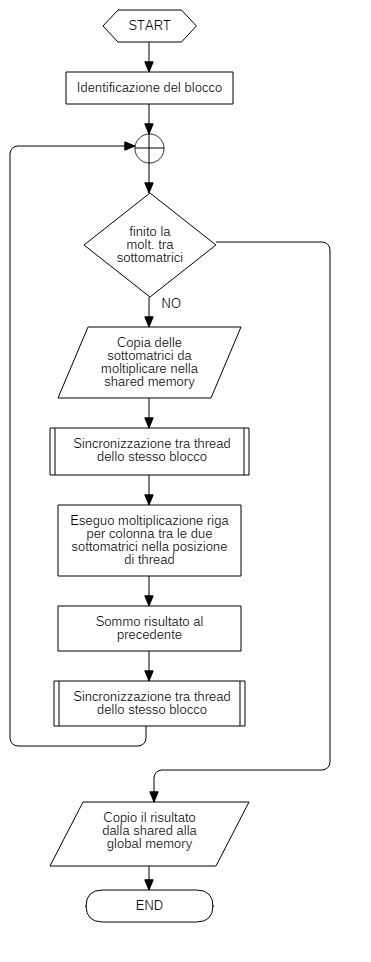
Essendo ogni blocco 32 x 32, cioè pari alla dimensione di una sottomatrice, posso mappare ogni thread del blocco ad ogni elemento della sottomatrice, quindi eseguire un prodotto riga per colonna per ogni coppia di sottomatrici e sommare tale risultato per tutte le coppie di sottomatrici che interessano quel blocco.



Quindi all’interno del kernel di blocco CUDA sarà presente:

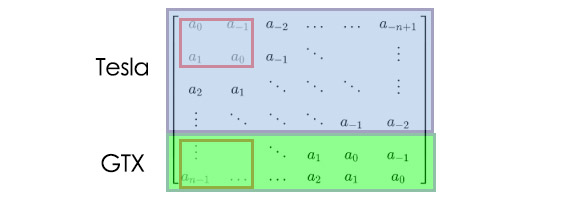
1. Un primo ciclo necessario per iterare le sottomatrici da moltiplicare che interessano a tale blocco
2. Un secondo ciclo necessario a svolgere l’operazione di moltiplicazione riga per colonna delle due sottomatrici identificate dall’iteratore del ciclo precedente.

Di seguito viene illustrato il flowchart del kernel per l’algoritmo che sfrutta dal shared memory.



# Multi-device

Nel programma multi-device l’elaborazione viene ripartita su entrambe GPU presenti nel sistema di test. L’algoritmo per eseguire la moltiplicazione è simile a quello utilizzato per il mono-device. L’unica differenza è che dovendo ripartire il carico su due GPU è stato necessario indicare come eseguire la moltiplicazione (in che porzione su una, rispetto all’altra). Sfruttando quindi il sistema di moltiplicazione per sottomatrici è stato possibile suddividere agevolmente il carico e in base ai parametri di compilazione inseriti.



# Compilazione

Per la compilazione (in sistemi linux) è stato creato un makefile, una volta invocato il comando:

*make mmult*

permette la compilazione di tutte le librerie utilizzate e la creazione dell’eseguibile nella cartella:

*./build/linux*

La compilazione avviene sfruttando lo standard C99.

## Single Device

È possibile specificare alcuni parametri aggiuntivi in fase di compilazione utilizzando la regola -D di gcc:

*make mmult D=SIMPLE,N=10240*

Esegue, oltre alla moltiplicazione in shared memory anche la moltiplicazione sfruttando la global memory per eseguire un confronto. Con il parametro N viene modificata la dimensione della matrice da moltiplicare.

## Multi Device

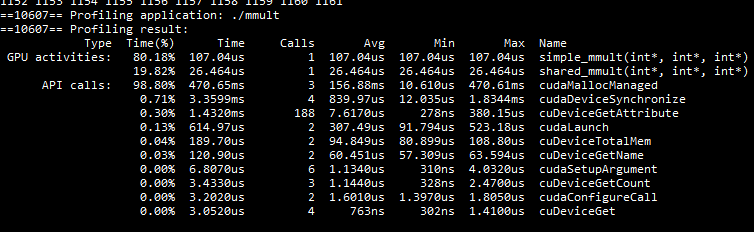
Per l’algoritmo multi device è possibile ripartire il carico di lavoro sulle due GPU durante la compilazione come segue:

*make mmult D=N=3072,NROWSTESLA=2048*

# Analisi del sistema

## Shared memory

Utilizzando il tool NVPROF è stato possibile ottenere i tempi di elaborazione della GPU nel caso del solo utilizzo della global memory e nel caso di utilizzo della shared memory.



Di seguito sono raccolti i dati relativi all’elaborazione con i due algoritmi:

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **Tempo Global** | **Tempo Shared** | **Rapporto** |
| **32** | 15 | 5 | 3 |
| **64** | 31 | 8 | 3,875 |
| **128** | 107 | 26 | 4,115384615 |
| **320** | 871 | 196 | 4,443877551 |
| **1024** | 62909 | 13459 | 4,674121406 |
| **3072** | 1329000 | 609000 | 2,18226601 |
| **5120** | 5152880 | 1833200 | 2,810866245 |

I tempi sono espressi in microsecondi

Come si può notare aumentando la dimensione N delle due matrici, aumentano anche i tempi di elaborazione del prodotto; ciò nonostante possiamo notare come i tempi dell’algoritmo che sfrutta la shared memory siano molto inferiori rispetto a quelli della Global.

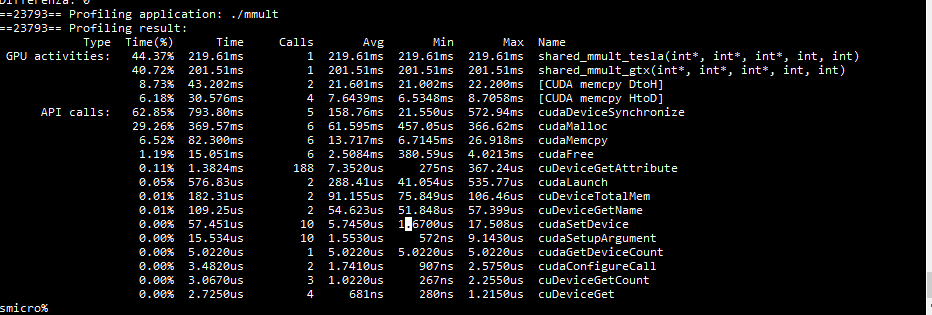
Se analizziamo il rapporto tra i tempi, si nota come per dimensioni piccole delle matrici l’efficienza leggermente più bassa che a dimensioni più elevate. Per dimensioni elevate possiamo notare come il valore del rapporto tra i tempi si tenda a stabilizzare attorno al 4.5, cioè l’algoritmo che sfrutta solamente la global memory ci metterà 4.5 volte di più rispetto all’algoritmo della shared memory, per avere poi una ricaduta in efficienza superando la dimensione di 1024 elementi.

## Multi-device

Per quanto riguarda l’algoritmo che sfrutta due GPU per eseguire la moltiplicazione di matrice è possibile analizzare i tempi di esecuzione dei due algoritmi in parallelo tramite il tool NVPROF. In particolare, per eseguire un confronto diretto, si è deciso di copiare la funzione di moltiplicazione e rinominandole. Quindi avremo:

* shared\_mmult\_tesla
* shared\_mmult\_gtx

Che lavoreranno in parallelo tra di loro una sulla scheda TESLA mentre l’altra sulla scheda GTX.



Di seguito sono riportati i tempi per una matrice con dimensione 3072 x 3072 con diversi carichi sulle due GPU:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N Tesla | N GTX | %TESLA | T tesla | T GTX | Rapporto tempi |
| 32 | 3040 | 0,010416667 | 3000 | 407000 | 0,007371007 |
| 64 | 3008 | 0,020833333 | 8000 | 395000 | 0,020253165 |
| 256 | 2816 | 0,083333333 | 37000 | 380000 | 0,097368421 |
| 1024 | 2048 | 0,333333333 | 226000 | 249000 | 0,907630522 |
| 1600 | 1472 | 0,520833333 | 219000 | 201000 | 1,089552239 |
| 2048 | 1024 | 0,666666667 | 340000 | 142000 | 2,394366197 |
| 3040 | 32 | 0,989583333 | 574000 | 5000 | 114,8 |

Tempi espressi in microsecondi

Vediamo come diminuendo il carico sulla tesla, i tempi generali si abbassano. Si ricorda che entrambi gli algoritmi sono eseguiti in parallelo, quindi una ripartizione equa su entrambe le GPU porta ad abbassamento sostanziale dei tempi. Per esempio, se prendiamo la stessa dimensione di matrice (3072), l’algoritmo che sfrutta la global memory ci impiega 1.32 secondi ad elaborare la matrice, mentre quello che sfrutta la shared memory ci impiega circa 0.6 secondi. Possiamo notare che ripartendo l’elaborazione su entrambe le schede abbiamo ancora un incremento di prestazione portando il tempo di elaborazione a circa 0.2 secondi. Se rapportiamo questi dati notiamo che sfruttando la shared memory e due schede grafiche abbiamo un incremento di prestazione di circa il 660% rispetto all’elaborazione con l’algoritmo che sfrutta la global memory.

Sommario

[Consegna 1](#_Toc518569206)

[Moltiplicazione tra matrici 1](#_Toc518569207)

[Sistema per il test 1](#_Toc518569208)

[Problematiche 1](#_Toc518569209)

[Multi-device 3](#_Toc518569210)

[Compilazione 4](#_Toc518569211)

[Single Device 4](#_Toc518569212)

[Multi Device 4](#_Toc518569213)

[Analisi del sistema 4](#_Toc518569214)

[Shared memory 4](#_Toc518569215)

[Multi-device 6](#_Toc518569216)