MOLTIPLICAZIONE TRA MATRICI

Riccardo Fontanini

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 $C = A \times B$ in questo modo

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ \dots$$

Con $c_{ij} \in C, a_{ik} \in A \ \ e \ b_{kj} \in B$

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:

$$n_{byte} = N \cdot N \cdot size_{int} \cdot n_{matrix} = 10 \cdot 10 \cdot 4 \cdot 2 = 800 \ byte$$

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

$$n_{byte} = N \cdot N \cdot size_{int} \cdot n_{matrix} = 100 \cdot 100 \cdot 4 \cdot 2 = 80000 \ byte$$

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:

$$T = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

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

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

$$C = \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}$$

$$D = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

Con la matrice totale quindi:

$$T = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 4 & 3 & 4 & 4 \end{bmatrix}$$

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

$$\begin{bmatrix} A_1 & B_1 \\ C_1 & D_1 \end{bmatrix} \times \begin{bmatrix} A_2 & B_2 \\ C_2 & D_2 \end{bmatrix} = \begin{bmatrix} A_1A_2 + B_1C_2 & A_1B_2 + B_1D_2 \\ C_1A_2 + D_1C_2 & C_1B_2 + D_1D_2 \end{bmatrix}$$

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

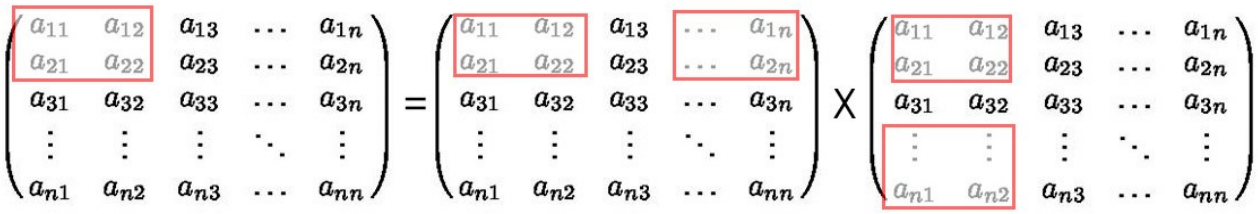
E così via..

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:

$$n_{byte} = N \cdot N \cdot size_{int} \cdot n_{matrix} = 32 \cdot 32 \cdot 4 \cdot 3 = 12288\ byte$$

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

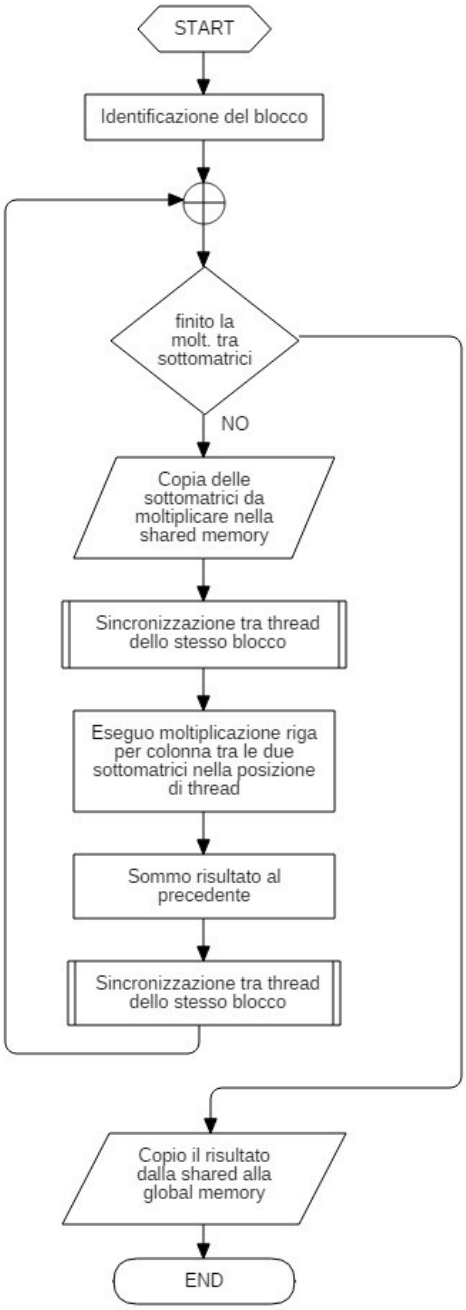
Essendo ogni blocco 32 x 32, cioè pari alla dimensione di una sottomatrice, posso mappare ogni thread del blocco ad ogni elemento della sottomatrice, quindi eseguire un prodotto riga per colonna per ogni coppia di sottomatrici e sommare tale risultato per tutte le coppie di sottomatrici che interessano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



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

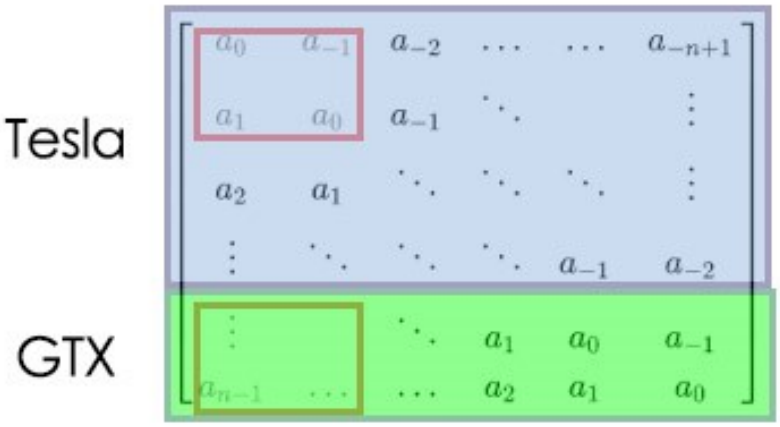
1. Un primo ciclo necessario per iterare le sottomatrici da moltiplicare che interessano a tale blocco
2. Un secondo ciclo necessario a svolgere l'operazione di moltiplicazione riga per colonna delle due sottomatrici identificate dall'iteratore del ciclo precedente.

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.

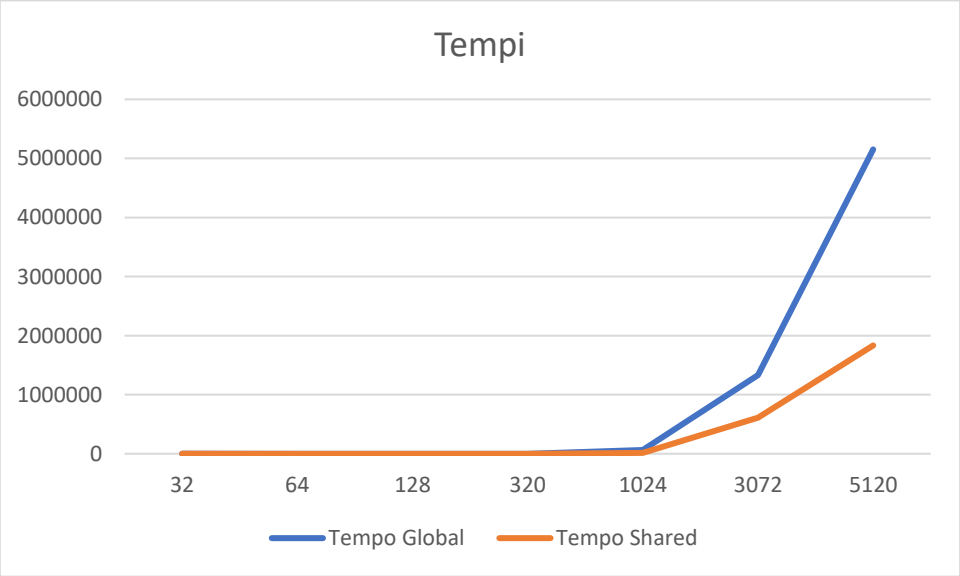
```
==10607== Profiling application: ./mmult
==10607== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   80.18%  107.04us         1  107.04us  107.04us  107.04us  simple_mmult(int*, int*, int*)
                   19.82%  26.464us         1  26.464us  26.464us  26.464us  shared_mmult(int*, int*, int*)
      API calls:   98.80%  470.65ms         3  156.88ms  10.610us  470.61ms  cudaMallocManaged
                    0.71%  3.3599ms         4  839.97us  12.035us  1.8344ms  cudaDeviceSynchronize
                    0.30%  1.4320ms       188  7.6170us     278ns  380.15us  cuDeviceGetAttribute
                    0.13%  614.97us         2  307.49us  91.794us  523.18us  cudaLaunch
                    0.04%  189.70us         2  94.849us  80.899us  108.80us  cuDeviceTotalMem
                    0.03%  120.90us         2  60.451us  57.309us  63.594us  cuDeviceGetName
                    0.00%  6.8070us         6  1.1340us     310ns  4.0320us  cudaSetupArgument
                    0.00%  3.4330us         3  1.1440us     328ns  2.4700us  cuDeviceGetCount
                    0.00%  3.2020us         2  1.6010us  1.3970us  1.8050us  cudaConfigureCall
                    0.00%  3.0520us         4     763ns     302ns  1.4100us  cuDeviceGet
```

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

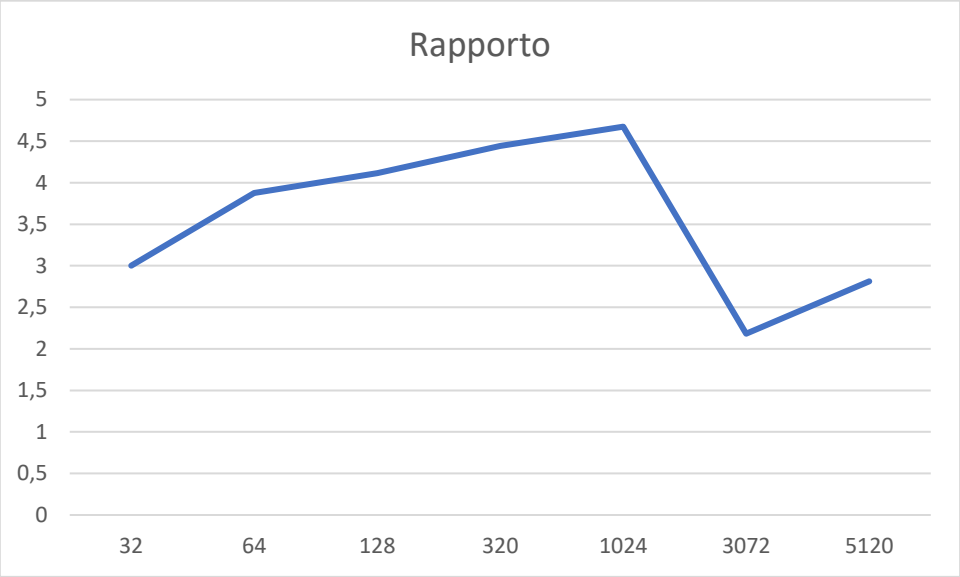
I tempi sono espressi in microsecondi

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.

```
==23793== Profiling application: ./mmult
==23793== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   44.37%  219.61ms         1  219.61ms  219.61ms  219.61ms  shared_mmult_tesla(int*, int*, int*, int, int)
                   40.72%  201.51ms         1  201.51ms  201.51ms  201.51ms  shared_mmult_gtx(int*, int*, int*, int, int)
                    8.73%  43.202ms         2  21.601ms  21.002ms  22.200ms  [CUDA memcpy DtoH]
                    6.18%  30.576ms         4  7.6439ms  6.5348ms  8.7058ms  [CUDA memcpy HtoD]
      API calls:   62.85%  793.80ms         5  158.76ms  21.550us  572.94ms  cudaDeviceSynchronize
                   29.26%  369.57ms         6  61.595ms  457.05us  366.62ms  cudaMalloc
                    6.52%  82.300ms         6  13.717ms  6.7145ms  26.918ms  cudaMemcpy
                    1.19%  15.051ms         6  2.5084ms  380.59us  4.0213ms  cudaFree
                    0.11%  1.3824ms       188  7.3520us     275ns  367.24us  cuDeviceGetAttribute
                    0.05%  576.83us         2  288.41us  41.054us  535.77us  cudaLaunch
                    0.01%  182.31us         2  91.155us  75.849us  106.46us  cuDeviceTotalMem
                    0.01%  109.25us         2  54.623us  51.848us  57.399us  cuDeviceGetName
                    0.00%  57.451us        10  5.7450us  1.6700us  17.508us  cudaSetDevice
                    0.00%  15.534us        10  1.5530us     572ns  9.1430us  cudaSetupArgument
                    0.00%  5.0220us         1  5.0220us  5.0220us  5.0220us  cudaGetDeviceCount
                    0.00%  3.4820us         2  1.7410us     907ns  2.5750us  cudaConfigureCall
                    0.00%  3.0670us         3  1.0220us     267ns  2.2550us  cuDeviceGetCount
                    0.00%  2.7250us         4     681ns     280ns  1.2150us  cuDeviceGet
smicro%
```

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