Moltiplicazione tra matrici

Marco Bellizzi 223966

Il seguente programma cuda ha come scopo effettuare la moltiplicazione tra due matrici generiche. Sono presentate 3 versioni: una versione sequenziale, una versione in parallelo lineare e una versione in parallelo più efficiente. L'obbiettivo è quello di mostrare come limitando gli accessi alla memoria globale le prestazioni migliorino.

Sono state dichiarate e allocate tre matrici (m, n, r) di dimensione differente (il numero di colonne di m è uguale al numero di righe di n). Le prime due sono state riempite con valore 1 ad ogni cella. Infine viene chiamato il kernel che ne effettua la moltiplicazione e riempie la matrice r con il risultato. Il programma usa blocchi di thread di dimenzione 32x32, in quantità sufficiente e ricoprire le dimensioni delle matrici.

```
int main() {
   int mHeight = 1000;
   int commonDim = 500;
   int nWidth = 1000;

   int* m;
   int* n;
   int* r;

   cudaMallocManaged(&m, mHeight * commonDim * sizeof(int));
   cudaMallocManaged(&n, commonDim * nWidth * sizeof(int));
   cudaMallocManaged(&r, mHeight * nWidth * sizeof(int));

   for(int i=0; i<mHeight; i++){
      for(int j=0; j<commonDim; j++) {
         m[commonDim*i + j] = 1;
      }
   }
```

```
    for(int i=0; i<commonDim; i++){
        for(int j=0; j<nWidth; j++) {
            n[nWidth*i + j] = 1;
        }
    }

    dim3 block_size(32, 32, 1);
    dim3 number_of_blocks(ceil(mHeight/32.0), ceil(commonDim/32.0), 1);

//    multiplySequential(m, n ,r, mHeight, commonDim, nWidth);
//    multiplyStraightforward<<<number_of_blocks, block_size>>>(m, n, r, mHeight, commonDim, nWidth);
    multiplyTiled<<<number_of_blocks, block_size>>>(m, n, r, mHeight, commonDim, nWidth);

    cudaDeviceSynchronize();

    return 0;
}
```

La versione sequenziale e quella parallela lineare sono molto simili, la differenza è che la versione sequenziale usa 2 cicli per scorrere le celle della matrice ed effettuare il calcolo, mentre nella versione parallela lineare per ogni cella è assegnato un thread che effettua il calcolo della corrispondente cella parallelamente agli altri thread. Nella versione più efficiente invece sono state usate due matrici condivise con ogni thread appartenente allo stesso blocco, in cui sono memorizzati i dati provenienti dalla memoria globale. In questa maniera si limita l'accesso alla memoria globale e si riduce il tempo di elaborazione. Il calcolo viene effettuato attraverso una variabile pValue in cui sono memorizzate le somme parziali derivanti dai calcoli sulle matrici condivise. I dati nelle matrici condivise vengono poi aggiornati fin quando non si completa l'operazione.

```
void multiplySequential(int* m, int* n, int* r, int mHeight, int commonDim, int nWidth) {
    for(int i=0; i<mHeight; i++) {
        for(int j=0; j<nWidth; j++) {

            int sum = 0;
            for(int k=0; k<commonDim; k++) {
```

```
            sum += m[i*commonDim + k] * n[k*nWidth + j];
          }

           r[i*nWidth + j] = sum;
       }
   }
}

__global__
void multiplyStraightforward(int* m, int* n, int* r, int mHeight, int
commonDim, int nWidth) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   if(i<mHeight && j<nWidth) {
      int sum = 0;
      for(int k=0; k<commonDim; k++) {
         sum += m[i*commonDim + k] * n[k*nWidth + j];
      }

      r[i*nWidth + j] = sum;
   }
}

__global__
void multiplyTiled(int * m, int* n, int* r, int mHeight, int commonDim, int
nWidth) {
       int tileWidth = 32;

        __shared__ int mShared[32][32];
        __shared__ int nShared[32][32];

       int row = blockIdx.x * blockDim.x + threadIdx.x;
       int col = blockIdx.y * blockDim.y + threadIdx.y;

       int pValue = 0;

       for(int p=0; p<=commonDim/tileWidth; p++) {
             if(p*tileWidth + threadIdx.y < commonDim && row<mHeight)
                   mShared[threadIdx.x][threadIdx.y] =
m[row*commonDim + p*tileWidth + threadIdx.y];
             else mShared[threadIdx.x][threadIdx.y] = 0;
```

```
            if(p*tileWidth + threadIdx.x < commonDim && col<nWidth)
                nShared[threadIdx.x][threadIdx.y] = n[(p*tileWidth +
threadIdx.x)*nWidth + col];
            else nShared[threadIdx.x][threadIdx.y] = 0;

            __syncthreads();

            for(int k=0; k<tileWidth; k++) {
                pValue += mShared[threadIdx.x][k] * nShared[k]
[threadIdx.y];
            }

            __syncthreads();

        }

        if(row<mHeight && col<nWidth)
            r[row*nWidth + col] = pValue;
}
```

Riporto in seguito i tempi di esecuzioni delle differenti versioni, usando differenti grandezze per le matrici.

M: 100x200
N: 200x300

tempo in sequenziale = 0,029 s
tempo parallelo lineare = 0,257 s
speed-up = 0,11
tempo parallelo più efficiente = 0,256 s
speed-up = 0,11

M: 500x500
N: 500x800

tempo in sequenziale = 0,672 s
tempo in parallelo lineare = 0,274 s
speed-up = 2,45
tempo in parallelo più efficiente = 0,263s
speed-up = 2,55

M: 1000x500
N: 500x1000

tempo in sequenziale = 1,542 s
tempo in parallelo lineare = 0,296 s
speed-up = 5,21
tempo in parallelo più efficiente = 0,266 s
speed-up = 5,80

M: 2000x500
N: 500x2000

tempo in sequenziale = 6,175 s
tempo in parallelo lineare = 0,338 s
speed-up = 18,27
tempo in parallelo più efficiente = 0,282 s
speed-up = 21,90