GPU Computing

Lab 9

cuBLAS

Linear Algebra

Column-major order

A = | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 |

array **A** = matrice linearizzata

encodes matrix B,

$$\begin{bmatrix} 1 & 5 & 32 \\ 1 & 8 & 55 \\ 2 & 13 & 89 \\ 3 & 21 & 144 \end{bmatrix}$$

ld: lead-dimension che in questo caso è il num di righe

Index by

$$B[\text{row } i,\ \text{col } j] = A[j \cdot ld + i]$$

Es: `i = j = 1, ld = 4` → `B[1,1] = 8` ↔ `A[1·4 + 1] = 8`

Generazione dati

matrice random
m x n
m righe, n colonne

Memorizzazione in **column-major** order

```c
/*
 * Generate a matrix with M rows and N columns in column-major order. The matrix
 * will be filled with random single-precision floating-point values between 0
 * and 1.
 */
void generate_random_dense_matrix(int m, int n, float **A) {
   float *a = (float *) malloc(sizeof(float) * m * n);

   // For each column
   for (int j = 0; j < n; j++)
      // For each row
      for (int i = 0; i < m; i++) {
         a[j * m + i] = ((float)rand() / RAND_MAX);
      }
   *A = a;
}
```

Allocazione matrici

Prototipo

```
cublasSetMatrix(int rows, int cols, int elementSize, const void *A, int lda, void *B, int ldb)
```

- **lda** and **ldb** specify the leading dimension of the source matrix A and destination matrix B
- The leading dimension is the **total number of rows** in the respective matrix

allocazioni contigua di matrici e vettori

trasferimento 'formattato' di matrici e vettori

```
// Allocate device memory
CHECK(cudaMalloc((void ** )&d_A, sizeof(float) * m * n));
CHECK(cudaMalloc((void ** )&d_B, sizeof(float) * m * p));
CHECK(cudaMalloc((void ** )&d_C, sizeof(float) * m * p));
CHECK(cudaMalloc((void ** )&d_x, sizeof(float) * n));
CHECK(cudaMalloc((void ** )&d_y, sizeof(float) * m));


// Transfer inputs to the device
CHECK_CUBLAS(cublasSetMatrix(m, n, sizeof(float), A, m, d_A, m));
CHECK_CUBLAS(cublasSetMatrix(n, p, sizeof(float), B, n, d_B, n));
CHECK_CUBLAS(cublasSetMatrix(m, p, sizeof(float), C, m, d_C, m));
CHECK_CUBLAS(cublasSetVector(n, sizeof(float), x, 1, d_x, 1));
CHECK_CUBLAS(cublasSetVector(m, sizeof(float), y, 1, d_y, 1));
```

CUBLAS context

CUBLAS: creazione del CUBLAS context

handle che può essere passato a ogni funzione nel codice

```
// Create the cuBLAS handle
CHECK_CUBLAS(cublasCreate(&handle));
int version;
CHECK_CUBLAS(cublasGetVersion(handle, &version));
printf("\nUsing CUBLAS Version: %d\n", version);

// your code

CHECK_CUBLAS(cublasDestroy(handle));
```

Nota: handle può essere usato da vari **host thread** e su **tutte** le **GPU** del nodo

…Code

C = A*B
A mat M x N
B mat N x P
C mat M x P

```
// mat-vect product
alpha = 1.0f;
beta = 1.0f;

// Retrieve the output vector from the device
CHECK_CUBLAS(cublasSgemv(handle, CUBLAS_OP_N, m, n, &alpha, d_A, m, d_x, 1, &beta, d_y, 1));
// Retrieve the output vector from the device
CHECK_CUBLAS(cublasGetVector(m, sizeof(float), d_y, 1, y, 1));


. . .
// mat-mat product
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, p, n, &alpha, d_A, m, d_B, n, &beta, d_C, m);
// Retrieve the output vector from the device
CHECK_CUBLAS(cublasGetMatrix(m, m, sizeof(float), d_C, m, C, m));

. . .

// free memory
cublasDestroy(handle);
```

Esercitazione

Effettuare moltiplicazioni tra matrici MQDB:

Utilizzare cuBLAS e verificare efficienza:

- Indicizzare sottomatrici
- Effettuare un loop su host senza kernel
- Provare approccio complessivo o blocco-a-blocco

MQDB

possibile sostituire qui la libreria cublas?
Come selezionare le sottomatrici?

```
/*****************************************************************/
/*                    GPU MQDB product                            */
/*****************************************************************/
printf("Kernel MQDB product...\n");
uint sdim = 0;
start = seconds();
for (uint i = 0; i < k; i++ ) {
   uint d = A.blkSize[i];
   mqdbBlockProd<<<grid, block>>>(d_A, d_B, d_C, sdim, d, n);
   sdim += d;
}
CHECK(cudaDeviceSynchronize());

. . .

// copy the array 'C' back from the GPU to the CPU
CHECK(cudaMemcpy(C1.elem, d_C.elem, nBytes,
cudaMemcpyDeviceToHost));
CHECK(cudaMemset(d_C.elem, 0.0, nBytes));
```

Conjugate Gradient

- x_0 = *vettore iniziale a caso*
- $r_0 = b - A * x_0$
- $p_0 = r_0$
- For $k = 0,1,\dots,n$

  1. $\alpha_k = \dfrac{p_k^T * r_k}{p_k^T * A * p_k}$
  2. $x_{k+1} = x_k + \alpha_k p_k$
  3. $r_{k+1} = b - A * x_{k+1}$
  4. $\beta_k = \dfrac{p_k^T * A * r_{k+1}}{p_k^T * A * p_k}$
  5. $p_{k+1} = r_{k+1} - \beta_k p_k$
  6. $k = k + 1$

- return x_{k+1}

```
cublasSdot(cublasHandle_t handle, int n, const float *x,
               int incx, const float *y, int incy, float *result)
```

```
cublasSaxpy(cublasHandle_t handle, int n, const float *alpha,
                const float *x, int incx, float *y, int incy)
```

cuRAND

Random Numbers

Esercitazione

MONTE CARLO SU GPU:

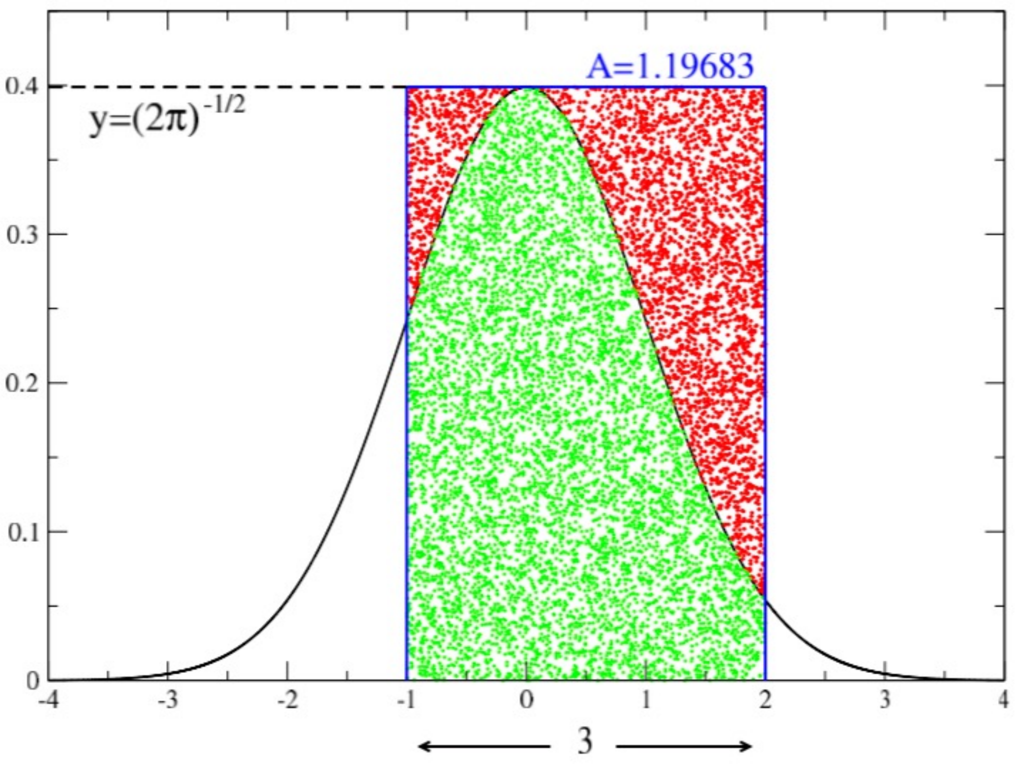
Disegnare un kernel per la stima dell'area della campana di gauss con il metodo Monte Carlo usando la libreria curand

$$P = \int_{-1}^{2} \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}} dx$$

$$A = 1.196826841$$

$$P = 0.8185946141$$

$$\frac{s}{n} \approx \frac{P}{A} = 0.6839780$$



Stima probabilità MC: host

```
// GPU procedure
#include <curand_kernel.h>

#define TRIALS_PER_THREAD 10000
#define BLOCKS  264
#define THREADS 264
#define PI 3.1415926535 // known value of pi
. . .
float host[BLOCKS * THREADS];
float *dev;
float a = -1;
float b = 2;
float max = 1.0f/sqrt(2*PI);
float A = (b-a)*max;
float P_true = 0.818594;
```

Stima probabilità MC

```
// GPU procedure
curandState *devStates;
cudaMalloc((void **) &dev, BLOCKS * THREADS * sizeof(float));
cudaMalloc((void **) &devStates, BLOCKS * THREADS * sizeof(curandState));
cudaEventRecord(start);
Gauss_GPU<<<BLOCKS, THREADS>>>(dev, devStates, a, b, max);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaMemcpy(host, dev, BLOCKS * THREADS * sizeof(float), cudaMemcpyDeviceToHost);
float P = 0.0;
for (int i = 0; i < BLOCKS * THREADS; i++) {
        P += host[i];
}
P = P/(BLOCKS * THREADS)*A;
```

Stima probabilità MC: kernel

```
__global__ void Gauss_GPU(float *estimate, curandState *states, float a, float b, float max) {
        unsigned int tid = threadIdx.x + blockDim.x * blockIdx.x;
        int s = 0;
        curand_init(tid, 0, 0, &states[tid]);
        for (int i = 0; i < TRIALS_PER_THREAD; i++) {
                float x = (b-a)*curand_uniform(&states[tid])+a;
                float y = curand_uniform(&states[tid]);  // max* dropped
                s += (y <= expf(-x*x/2));
        }
        estimate[tid] = s / (float) TRIALS_PER_THREAD;
```

Risultati e profiling

```
Device : Tesla P100-PCIE-16GB
CPU elapsed time: 27.96 (sec)
CPU estimate of P = 0.818583 [error of 0.000011]
GPU elapsed time: 0.00823 (sec)
GPU estimate of P = 0.818559 [error of 0.000035 ]
Speedup = 3397
```

```
>> nvprof Release/lab8-Gauss-MC
```

```
                   Type   Time(%) Calls Name
GPU activities: 99.70%  8.2086ms   1.  Gauss_GPU(float*, curandStateXORWOW*, float, float, float)
                 0.30%  24.319us   1   [CUDA memcpy DtoH]
    API calls:  95.68%  230.12ms   2   cudaEventCreate
```