

Taula de continguts

Exercici 1.....	2
1D.....	2
2D.....	3
Exercici 2.....	4
Canvis al kernel.....	4
Provar el kernel.....	5
Exercici 3.....	8
Metodologia utilitzada.....	8
Preparació de la sessió.....	8
Crear la imatge d'entrada a l'equip local.....	8
CUDA.....	9
Visualització de les imatges.....	12
Exercici 4.....	15
Obtenció de l'arranjament dels blocs de la graella i dels fils d'execució dels blocs.....	15
CUDA.....	17
Captures de pantalla.....	21

Exercici 1

Nom d'usuari capa08. En aquesta activitat s'ha d'especificar que la graella tingui només un bloc i que aquest tingui 9 fils d'execució. Els 9 fils d'exercici poden estar en 1D (9) o en 2D (3x3).

1D

Variables 1D:

- blockIdx.x identificador de bloc (el seu valor és 0)
- blockDim.x nombre total de fils d'execució del bloc (el seu valor és 9)
- threadIdx.x identificador del fil d'execució (el seu valor pot ser 0, 1, 2, 3, 4, 5, 6, 7, 8)

En 1D, podem obtenir l'índex general amb aquesta fórmula:

- Índex = (blockIdx.x * blockDim.x) + threadIdx.x

En una graella d'un sol bloc podem utilitzar directament threadIdx.x com a índex general. La fórmula continua sent vàlida perquè blockIdx.x és zero.

```
%%cuda
#include <stdio.h>

__global__ void kernel(){
    // express the collection of blocks, and the collection of threads within
    // a block, as a 1-D array
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int bdim = blockDim.x;
    int idx = bid * bdim + tid;
    printf("My Id is %d, I am the thread %d of %d in the block %d\n",
        idx, tid, bdim, bid);
}

int main(){
    int bnum = 1;
    int tnum = 9;
    printf("Username: capa08\n");
    printf("Blocks: %d\n", bnum);
    printf("Threads per block: %d\n", tnum);
    kernel<<<bnum,tnum>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Sortida

```
Username: capa08
Blocks: 1
Threads per block: 9
My Id is 0, I am the thread 0 of 9 in the block 0
My Id is 1, I am the thread 1 of 9 in the block 0
My Id is 2, I am the thread 2 of 9 in the block 0
My Id is 3, I am the thread 3 of 9 in the block 0
My Id is 4, I am the thread 4 of 9 in the block 0
My Id is 5, I am the thread 5 of 9 in the block 0
My Id is 6, I am the thread 6 of 9 in the block 0
My Id is 7, I am the thread 7 of 9 in the block 0
My Id is 8, I am the thread 8 of 9 in the block 0
```

2D

Codi d'una graella amb 1 bloc, on el bloc té 9 fils d'execució en 2D (3x3).

```
%%cuda
#include <stdio.h>

#define N 3

__global__ void kernel(){
    int tidx = threadIdx.x;
    int tidy = threadIdx.y;
    int bid = blockIdx.x;
    int bdimx = blockDim.x;
    int bdimy = blockDim.y;
    int tnum = bdimx * bdimy;
    int idx = bid * tnum + (tidy*bdimy) + tidy;
    printf("My Id is %d, I am the thread %d,%d of %dx%d in the block %d\n",
        idx, tidx, tidy, bdimy, bdimx, bid);
}

int main(){
    dim3 dimGrid(1, 1); // grid = 1 x 1 block
    dim3 dimBlock(N, N); // block = N x N threads

    printf("Username: capa08\n");
    printf("Blocks: %d\n", dimGrid.x);
    printf("Threads per block: %d\n", dimBlock.x * dimBlock.y);
    kernel<<<dimGrid, dimBlock>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Sortida:

```
Username: capa08
Blocks: 1
Threads per block: 9
My Id is 0, I am the thread 0,0 of 3x3 in the block 0
My Id is 1, I am the thread 1,0 of 3x3 in the block 0
My Id is 2, I am the thread 2,0 of 3x3 in the block 0
My Id is 3, I am the thread 0,1 of 3x3 in the block 0
My Id is 4, I am the thread 1,1 of 3x3 in the block 0
My Id is 5, I am the thread 2,1 of 3x3 in the block 0
My Id is 6, I am the thread 0,2 of 3x3 in the block 0
My Id is 7, I am the thread 1,2 of 3x3 in the block 0
My Id is 8, I am the thread 2,2 of 3x3 in the block 0
```

Variables 2D:

- blockIdx.x id de la columna del bloc al grid (el seu valor és 0)
- blockIdx.y id de la fila del bloc al grid (el seu valor és 0)
- blockDim.x nombre de columnes de fils d'execució del bloc (el seu valor és 3)
- blockDim.y nombre de files de fils d'execució del bloc (el seu valor és 3)
- threadIdx.x id de la columna del fil d'execució al bloc (el seu valor pot ser 0, 1, 2)
- threadIdx.y id de la fila del fil d'execució al bloc (el seu valor pot ser 0, 1, 2)

En 2D, si el grid té un bloc, podem obtenir l'índex general amb aquesta fórmula:

- Índex = (blockDim.y * threadIdx.y) + threadIdx.x

Exercici 2

Canvis al kernel

En la implementació de l'algoritme de multiplicació de dues matrius quadrades que es proporciona, no s'ha tingut en compte la paral·lelització a nivell de dades, i s'ha implementat com si s'utilitzés la CPU, amb diversos bucles imbricats i cost computacional d'ordre $O(N^3)$.

```
__global__ void multiplicationKernel(const int size,
                                     const double *src_matrix_1,
                                     const double *src_matrix_2,
                                     double *dst_matrix)
{
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
        {
            double sum = 0;
            for (int k = 0; k < size; k++)
```

```

        sum += src_matrix_1[i * size + k] * src_matrix_2[k * size + j];
        dst_matrix[i * size + j] = sum;
    }
}

```

S'hauria d'haver implementat d'aquesta manera. On per a l'algoritme de multiplicació d'una matriu quadrada, el nombre de fils d'execució és $O(N^2)$ i en cada fil d'execució es calcula una cel·la del resultat $dst_matrix_{row,col}$

```

__global__ void multiplicationKernel(const int size,
                                    const double *src_matrix_1,
                                    const double *src_matrix_2,
                                    double *dst_matrix)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    if (row < size && col < size)
    {
        double sum = 0;
        for (int k = 0; k < size; k++)
            sum += src_matrix_1[row * size + k] * src_matrix_2[k * size + col];
        dst_matrix[row * size + col] = sum;
    }
}

```

El condicional abans del codi `if (row < size && col < size)`, evita sortir dels límits del problema (les cel·les de la matriu quadrada del resultat).

Provar el kernel

En el següent codi es du a terme una prova del kernel:

```

%%cuda
#include <iostream>

#define N 10

using namespace std;

__global__ void multiplicationKernel(const int size,
                                    const double *src_matrix_1,
                                    const double *src_matrix_2,
                                    double *dst_matrix)
{
    // express the collection of blocks, and the collection of threads
    // within a block, as a 2-D array
    int row = blockIdx.x * blockDim.x + threadIdx.x;

```

```
int col = blockIdx.y * blockDim.y + threadIdx.y;
// Typical problems are not friendly multiples of blockDim.x
// Avoid accessing beyond the end of the arrays
if (row < size && col < size)
{
    double sum = 0;
    for (int k = 0; k < size; k++)
        sum += src_matrix_1[row * size + k] * src_matrix_2[k * size + col];
    dst_matrix[row * size + col] = sum;
}
}

void print_matrix(double m[N][N], string name) {
    std::cout << name << endl;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            std::cout << m[i][j] << ", ";
        }
        std::cout << endl;
    }
}

int main()
{
    double a[N][N], b[N][N], c[N][N];
    double *d_a, *d_b, *d_c;
    size_t size = N * N * sizeof(double);

    // alloc space for device copies of a, b, and c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // initialize matrices
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            a[i][j] = i;
            b[i][j] = j;
        }
    }

    // copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

```
dim3 dimGrid(1, 1); // grid = 1 x 1 block
dim3 dimBlock(N, N); // block = N x N threads

multiplicationKernel<<<dimGrid, dimBlock>>>(N, d_a, d_b, d_c);

cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

print_matrix(a, "A");
print_matrix(b, "B");
print_matrix(c, "C");

// clean up
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;
}
```

Sortida:

```
A
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
B
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
C
0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 10, 20, 30, 40, 50, 60, 70, 80, 90,
0, 20, 40, 60, 80, 100, 120, 140, 160, 180,
0, 30, 60, 90, 120, 150, 180, 210, 240, 270,
0, 40, 80, 120, 160, 200, 240, 280, 320, 360,
```

```
0, 50, 100, 150, 200, 250, 300, 350, 400, 450,  
0, 60, 120, 180, 240, 300, 360, 420, 480, 540,  
0, 70, 140, 210, 280, 350, 420, 490, 560, 630,  
0, 80, 160, 240, 320, 400, 480, 560, 640, 720,  
0, 90, 180, 270, 360, 450, 540, 630, 720, 810,
```

Exercici 3

Metodologia utilitzada

S'utilitza Google Drive per emmagatzemar els fitxers de les imatges. La imatge d'entrada es genera en l'equip local i es puja a Google Drive. Per accedir als fitxers de Google Drive es munta la unitat des de Google Collab. Una vegada generada la imatge de sortida, es visualitza i es fan proves per veure si les dades de la imatge són correctes des de Google Collab i des de l'equip local.

Preparació de la sessió

Canviar l'entorn d'execució a «T4 GPU», instal·lar i carregar l'extensió nvcc4jupyter, i muntar la carpeta de Google Drive:

```
!pip install nvcc4jupyter  
%load_ext nvcc4jupyter
```

Ordres per instal·lar i carregar l'extensió nvcc4jupyter

Crear la imatge d'entrada a l'equip local

Primer s'han instal·lat les dependències per executar el codi python de manera local:

```
$ brew install pil numpy python-matplotlib
```

Després s'ha generat el fitxer de la imatge i s'ha pujat a la carpeta de Google Drive:

```
from PIL import Image  
import numpy as np  
  
width, height = 1920, 1080  
data = np.zeros((height, width, 3), dtype=np.uint8)  
for x in range(width):  
    for y in range(height):  
        data[y, x] = [x % 256, (y % 256), (x * y) % 256] # Color pattern  
  
img = Image.fromarray(data, 'RGB')  
img.save('test_image.ppm')
```

CUDA

Codi:

```
%%cuda
#include <stdio.h>
#include <iostream>
#include <fstream>

#ifndef INPUT_FILE "test_image.ppm"
#ifndef OUTPUT_FILE "output_image.ppm"

#ifndef INPUT_FILE "/content/sample_data/test_image.ppm"
#ifndef OUTPUT_FILE "/content/sample_data/output_image.ppm"

#define INPUT_FILE "/content/drive/MyDrive/test_image.ppm"
#define OUTPUT_FILE "/content/drive/MyDrive/output_image.ppm"

using namespace std;

void savePPM(const char *filename,
             unsigned char *data,
             int width,
             int height) {
    ofstream file(filename, ios::binary);
    file << "P5\n" << width << " " << height << "\n255\n";

    // Write grayscale data
    file.write(reinterpret_cast<char *>(data), width * height);
    file.close();
}

bool loadPPM(const char *filename,
             unsigned char *data,
             int width,
             int height) {
    ifstream file(filename, ios::binary);
    if (!file) {
        cerr << "couldn't find file: " << filename << endl;
        return false;
    }

    string header;
    file >> header >> width >> height; // P6, width, height
    int maxVal;
    file >> maxVal;
    file.ignore(); // skip newline

    file.read(reinterpret_cast<char *>(data), width * height * 3);
    return true;
}
```

```

}

__global__ void kernel(unsigned char *d_A, unsigned char *d_B, int width, int height){
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
int p = y * width + x;
int idx = p * 3;

// Typical problems are not friendly multiples of blockDim.x
// Avoid accessing beyond the end of the arrays
if (x < width && y < height) {
    float gray = 0.3f * d_A[idx] +
        0.59f * d_A[idx + 1] +
        0.11f * d_A[idx + 2];
    d_B[p] = gray;
}
}

int main(){
const int width = 1920, height = 1080;
const int n_pixels = width * height;
const size_t bytes_A = n_pixels * sizeof(unsigned char) * 3;
const size_t bytes_B = n_pixels * sizeof(unsigned char);

// Allocate memory for arrays A, B on host
unsigned char *A, *B;
A = (unsigned char*)malloc(bytes_A);
B = (unsigned char*)malloc(bytes_B);

// Allocate memory for arrays d_A, d_B on device
unsigned char *d_A, *d_B;
cudaMalloc(&d_A, bytes_A);
cudaMalloc(&d_B, bytes_B);

if (!loadPPM(INPUT_FILE, A, width, height)) {
    // Cleanup
    free(A);
    free(B);
    cudaFree(d_A);
    cudaFree(d_B);
    // Raise SIGABRT
    abort();
}

// Copy data from host array A to device array d_A
cudaMemcpy(d_A, A, bytes_A, cudaMemcpyHostToDevice);

dim3 dimGrid(15, 135);
dim3 dimBlock(128, 8);
}

```

```
kernel<<<dimGrid,dimBlock>>>(d_A, d_B, width, height);

// Copy data from device array d_B to host array B
cudaMemcpy(B, d_B, bytes_B, cudaMemcpyDeviceToHost);

savePPM(OUTPUT_FILE, B, width, height);

// Cleanup
free(A);
free(B);
cudaFree(d_A);
cudaFree(d_B);

return 0;
}
```

Requisits:

- La GPU com a molt pot tenir 1024 fils d'execució per bloc:
 - dimBlock.x * dimBlock.y <= 1024
- Requisits del kernel amb arranjament 2D:
 - dimGrid.x * dimBlock.x >= 1920
 - dimGrid.y * dimBlock.y >= 1080

Descomposició en factors:

- $1920 = 2^7 \cdot 3 \cdot 5$
- $1020 = 2^3 \cdot 3^3 \cdot 5$

S'utilitza una graella de 15 x 135 blocs amb 128 x 8 fils d'execució per bloc.

Exemples d'altres configuracions:

- una graella de 192 x 108 blocs amb 10 x 10 fils (100 fils)
- una graella de 46 x 108 blocs amb 20 x 10 fils (200 fils)
- una graella de 23 x 108 blocs amb 40 x 10 fils (400 fils)
- una graella de 15 x 135 blocs amb 128 x 8 fils (1024 fils)

Explicació del codi de la funció main:

1. Primer es crea l'espai d'emmagatzematge per emmagatzemar les dades de les imatges. La imatge d'entada en format P6 requereix el triple de bytes que la imatge de sortida, ja que la imatge d'entrada té tres canals (roig, verd i blau), mentre que la imatge de sortida només un. Per crear i assignar l'espai de memòria l'espai a l'equip s'utilitza malloc mentre que per crear i assignar l'espai de memòria per als vectors al dispositiu s'utilitza cudaMalloc.

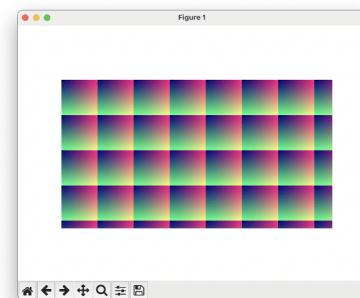
2. Després es llegeixen les dades binàries de la imatge, on de forma consecutiva hi ha el component roig, verd i blau per a cada píxel.
3. Les dades de la imatge d'entrada es transfereixen al dispositiu abans de cridar al kernel, amb la crida a la funció `cudaMemcpy(d_A, A, bytes_A, cudaMemcpyHostToDevice);`
4. Després s'especifiquen els blocs de la graella i els fils d'execució de cada blocben la crida al kernel.
5. S'executa el kernel.
6. Després d'executar el kernel, es copien les dades binàries de la imatge de sortida, del dispositiu a l'equip, amb la crida a la funció:
`cudaMemcpy(B, d_B, bytes, cudaMemcpyDeviceToHost);`
7. S'emmagatzema la imatge al disc, amb la crida a la funció:
`savePPM(OUTPUT_FILE, B, width, height);`
8. S'allibera la memòria de l'equip i del dispositiu, amb `free` i `cudaFree` respectivament.
9. La funció retorna zero, execució correcta.

Visualització de les imatges

Visualització de la imatge original a l'equip local:

```
from PIL import Image
import matplotlib.pyplot as plt

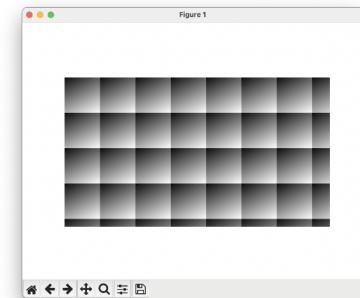
img = Image.open('test_image.ppm')
plt.imshow(img)
plt.axis('off')
plt.show()
```



Visualització de la imatge del resultat a l'equip local:

```
from PIL import Image
import matplotlib.pyplot as plt

img = Image.open('output_image.ppm')
plt.imshow(img, cmap="gray")
plt.axis('off')
plt.show()
$ python3 show_output.py
```

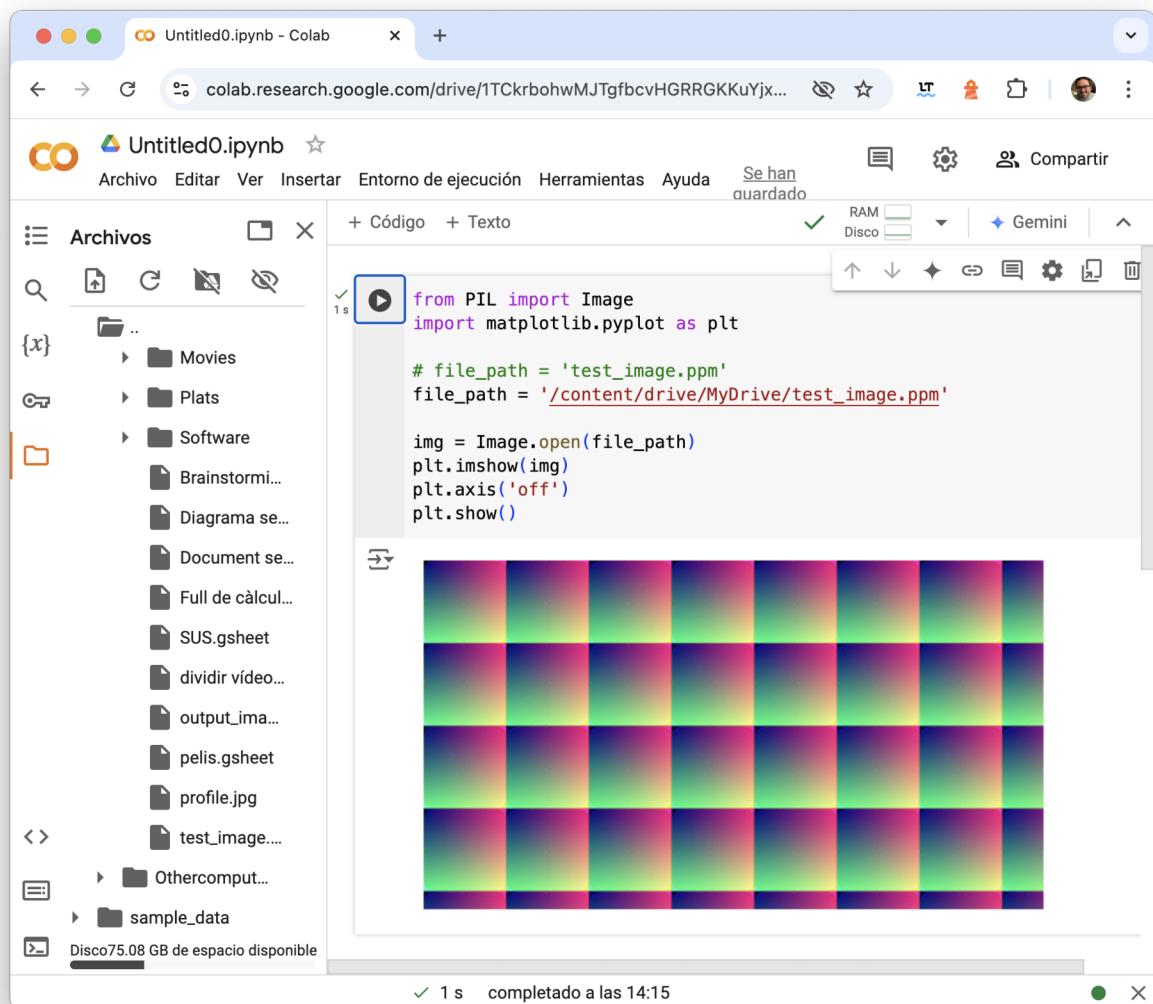


Visualització de la imatge original a Google Collab

```
from PIL import Image
import matplotlib.pyplot as plt

# file_path = 'test_image.ppm'
file_path = '/content/drive/MyDrive/test_image.ppm'

img = Image.open(file_path)
plt.imshow(img)
plt.axis('off')
plt.show()
```

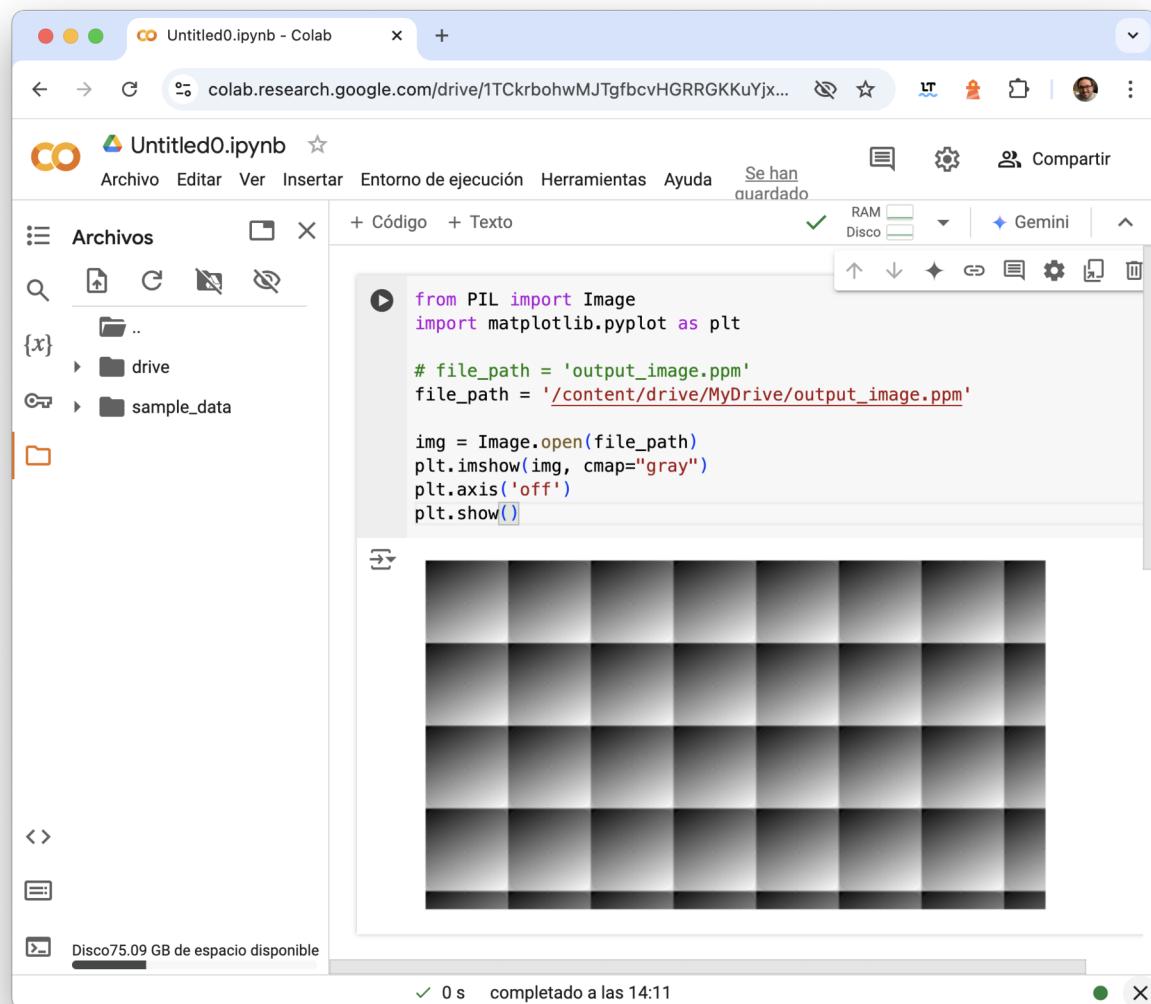


Visualització de la imatge del resultat a Google Collab

```
from PIL import Image
import matplotlib.pyplot as plt

# file_path = 'output_image.ppm'
file_path = '/content/drive/MyDrive/output_image.ppm'

img = Image.open(file_path)
plt.imshow(img, cmap="gray")
plt.axis('off')
plt.show()
```



Exercici 4

Requisits del problema:

- La GPU com a molt pot tenir 1024 fils d'execució per bloc:
 - $\text{dimBlock.x} * \text{dimBlock.y} \leq 1024$
- Requisits del kernel amb arranjament 2D:
 - $\text{dimGrid.x} * \text{dimBlock.x} \geq 1000$
 - $\text{dimGrid.y} * \text{dimBlock.y} \geq 1000$

Obtenció de l'arranjament dels blocs de la graella i dels fils d'execució dels blocs

```
#include <stdio.h>
#include <math.h>
#define SIZE 500

int value_in_vector(int *vector,
                    int value,
                    int lenght) {
    for(int i=0;i<lenght;i++) {
        if (vector[i] == -1) return 0;
        if (vector[i] == value) return 1;
    }
    return 0;
}

int main(void)
{
    int values[60];
    int sizes[SIZE];
    int config_thrd[SIZE][2];
    int sizes_idx;

    // init vector values
    for (int i=0; i<SIZE; i++) sizes[i] = -1;

    // at least 2D (i=2), and max value is the square root of 1000
    // values[i-2] * values[59+2-i] >= 1000
    for (int i=2; i<32; i++) {
        values[i-2] = i;
        values[59+2-i] = 1 + ceil(1000/i);
    }

    sizes_idx = 0;
    for (int i=0;i<60;i++) {
```

```

for (int j=0;j<30;j++) {
    int product = values[i] * values[j];
    if (product < 1000 && !value_in_vector(sizes, product, sizes_idx)) {
        sizes[sizes_idx] = product;
        config_thrd[sizes_idx][0] = i;
        config_thrd[sizes_idx][1] = j;
        sizes_idx++;
    }
}
}

printf("dimBlock.x * dimBlock.x, dimBlock.x, dimBlock.y, dimGrid.x, dimGrid.y, dimBlock.x
* dimGrid.x, dimBlock.y * dimGrid.y\n");
for(int i=0; i<sizes_idx; i++)
printf("%d, %d, %d, %d, %d, %d, %d\n",
    sizes[i],
    values[config_thrd[i][0]],
    values[config_thrd[i][1]],
    values[59-config_thrd[i][0]],
    values[59-config_thrd[i][1]],
    values[config_thrd[i][0]] * values[59-config_thrd[i][0]],
    values[config_thrd[i][1]] * values[59-config_thrd[i][1]]
);
return 0;
}

```

Exemple de sortida:

```

dimBlock.x * dimBlock.x, dimBlock.x, dimBlock.y, dimGrid.x, dimGrid.y, dimBlock.x * dimGrid.x, dimBlock.y * dimGrid.y
4, 2, 2, 501, 501, 1002, 1002
6, 2, 3, 501, 334, 1002, 1002
8, 2, 4, 501, 251, 1002, 1004
10, 2, 5, 501, 201, 1002, 1005
12, 2, 6, 501, 167, 1002, 1002
14, 2, 7, 501, 143, 1002, 1001
16, 2, 8, 501, 126, 1002, 1008
18, 2, 9, 501, 112, 1002, 1008
20, 2, 10, 501, 101, 1002, 1010
22, 2, 11, 501, 91, 1002, 1001
24, 2, 12, 501, 84, 1002, 1008
26, 2, 13, 501, 77, 1002, 1001
28, 2, 14, 501, 72, 1002, 1008
30, 2, 15, 501, 67, 1002, 1005
32, 2, 16, 501, 63, 1002, 1008

```

CUDA

```
%%cuda
#include <iostream>
#include <random>
#include <cmath>
#include <ctime>
#include <iomanip>

#define N 1000
#define SIZE 500

using namespace std;

int value_in_vector(int *vector,
                     int value,
                     int lenght) {
    for(int i=0;i<lenght;i++) {
        if (vector[i] == -1) return 0;
        if (vector[i] == value) return 1;
    }
    return 0;
}

__global__ void multiplicationKernel(const int size,
                                     const double *src_matrix_1,
                                     const double *src_matrix_2,
                                     double *dst_matrix)
{
    // express the collection of blocks, and the collection of threads within a
    // block, as a 2-D array
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    // Typical problems are not friendly multiples of blockDim.x
    // Avoid accessing beyond the end of the arrays
    if (row < size && col < size)
    {
        double sum = 0;
        for (int k = 0; k < size; k++)
            sum += src_matrix_1[row * size + k] * src_matrix_2[k * size + col];
        dst_matrix[row * size + col] = sum;
    }
}

int main()
{
    double *a, *b, *c;
    double *d_a, *d_b, *d_c;
    size_t size = N * N * sizeof(double);
```

```

struct timespec t0, t1;

int values[60];
int sizes[SIZE];
int config_thrd[SIZE][2];
int sizes_idx;

// alloc space for host copies of a, b, and c
a = (double *) malloc(size);
b = (double *) malloc(size);
c = (double *) malloc(size);

// alloc space for device copies of a, b, and c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

// initialize matrices
double lower_bound = 0;
double upper_bound = 10;
uniform_real_distribution<double> unif(lower_bound, upper_bound);
default_random_engine re;
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        int idx = (i*N) + j;
        a[idx] = unif(re);
        b[idx] = unif(re);
    }
}

// init block and thread dim vectors
for (int i=0; i<SIZE; i++)
    sizes[i] = -1;
for (int i=2; i<32; i++) {
    values[i-2] = i;
    values[59+2-i] = 1 + ceil(1000/i);
}
sizes_idx = 0;
for (int i=0;i<60;i++) {
    for (int j=0;j<30;j++) {
        int product = values[i] * values[j];
        if (product < 1000 && !value_in_vector(sizes, product, sizes_idx)) {
            sizes[sizes_idx] = product;
            config_thrd[sizes_idx][0] = i;
            config_thrd[sizes_idx][1] = j;
            sizes_idx++;
        }
    }
}

```

```

        }

}

// copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

cout << "# dimGrid.x, dimGrid.y, dimBlock.x, dimBlock.y, execution time" << endl;
for (int i=0; i<sizes_idx; i++) {
    dim3 dimGrid(values[59-config_thrd[i][0]],
                  values[59-config_thrd[i][1]]);
    dim3 dimBlock(values[config_thrd[i][0]],
                  values[config_thrd[i][1]]);

    timespec_get(&t0, TIME_UTC);
    multiplicationKernel<<<dimGrid, dimBlock>>>(N, d_a, d_b, d_c);
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    timespec_get(&t1, TIME_UTC);
    double diff = (double)(t1.tv_sec - t0.tv_sec) +
                  ((double)(t1.tv_nsec - t0.tv_nsec)/1000000000.0);
    cout << values[59-config_thrd[i][0]] << ", "
        << values[59-config_thrd[i][1]] << ", "
        << values[config_thrd[i][0]] << ", "
        << values[config_thrd[i][1]] << ", "
        << fixed << setprecision(9) << diff << endl;
}

// clean up
free(a);
free(b);
free(c);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;
}

```

Sortida en format CSV

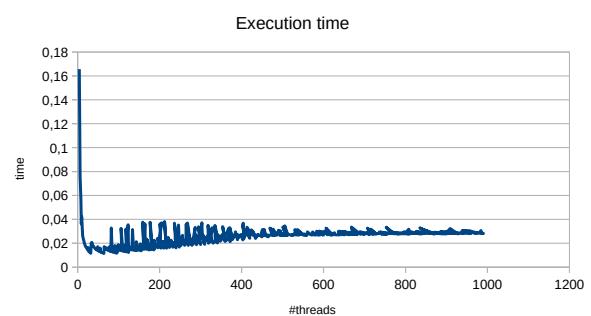
```

# dimGrid.x, dimGrid.y, dimBlock.x, dimBlock.y, execution time
501, 501, 2, 2, 0.165654459
501, 334, 2, 3, 0.075818144
501, 251, 2, 4, 0.056779180
501, 201, 2, 5, 0.042136417
501, 167, 2, 6, 0.026378391
501, 143, 2, 7, 0.023147287
501, 126, 2, 8, 0.020388349

```

```
501, 112, 2, 9, 0.018504774
501, 101, 2, 10, 0.016920034
501, 91, 2, 11, 0.015873657
501, 84, 2, 12, 0.014922423
501, 77, 2, 13, 0.014018112
501, 72, 2, 14, 0.012843326
501, 67, 2, 15, 0.012219531
501, 63, 2, 16, 0.011650651
...
...
```

Utilitzar menys blocs i més fils d'execució per bloc té un millor rendiment. A partir de 400 fils d'execució el rendiment és estable.



Captures de pantalla

The screenshot shows a Google Colab notebook titled "exercici1.ipynb". The code in the cell is a CUDA program. It includes a kernel function that prints thread and block IDs, and a main function that initializes variables and calls the kernel. The output of the cell shows the printed messages for 9 threads in one block.

```
%%cuda
#include <stdio.h>

__global__ void kernel(){
    // express the collection of blocks, and the collection of threads within
    // a block, as a 1-D array
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int bdim = blockDim.x;
    int idx = bid * bdim + tid;
    printf("My Id is %d, I am the thread %d of %d in the block %d\n",
           idx, tid, bdim, bid);
}

int main(){
    int bnum = 1;
    int tnum = 9;
    printf("Username: capa08\n");
    printf("Blocks: %d\n", bnum);
    printf("Threads per block: %d\n", tnum);
    kernel<<<bnum,tnum>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Output:

```
Username: capa08
Blocks: 1
Threads per block: 9
My Id is 0, I am the thread 0 of 9 in the block 0
My Id is 1, I am the thread 1 of 9 in the block 0
My Id is 2, I am the thread 2 of 9 in the block 0
My Id is 3, I am the thread 3 of 9 in the block 0
My Id is 4, I am the thread 4 of 9 in the block 0
My Id is 5, I am the thread 5 of 9 in the block 0
My Id is 6, I am the thread 6 of 9 in the block 0
My Id is 7, I am the thread 7 of 9 in the block 0
My Id is 8, I am the thread 8 of 9 in the block 0
```

The screenshot shows a Google Colab notebook titled "exercici2.ipynb". The code cell contains the following CUDA C code:

```
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;
```

The code is annotated with labels A, B, and C below the corresponding matrix elements:

- A**:
0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 1, 1, 1, 1, 1, 1, 1,
2, 2, 2, 2, 2, 2, 2, 2, 2,
3, 3, 3, 3, 3, 3, 3, 3, 3,
4, 4, 4, 4, 4, 4, 4, 4, 4,
5, 5, 5, 5, 5, 5, 5, 5, 5,
6, 6, 6, 6, 6, 6, 6, 6, 6,
7, 7, 7, 7, 7, 7, 7, 7, 7,
8, 8, 8, 8, 8, 8, 8, 8, 8,
9, 9, 9, 9, 9, 9, 9, 9, 9,
B
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
C
0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 10, 20, 30, 40, 50, 60, 70, 80, 90,
0, 20, 40, 60, 80, 100, 120, 140, 160, 180,
0, 30, 60, 90, 120, 150, 180, 210, 240, 270,
0, 40, 80, 120, 160, 200, 240, 280, 320, 360,
0, 50, 100, 150, 200, 250, 300, 350, 400, 450,
0, 60, 120, 180, 240, 300, 360, 420, 480, 540,
0, 70, 140, 210, 280, 350, 420, 490, 560, 630,
0, 80, 160, 240, 320, 400, 480, 560, 640, 720,
0, 90, 180, 270, 360, 450, 540, 630, 720, 810,

The status bar at the bottom indicates "1 s" and "completado a las 11:57".

The screenshot shows a Google Colab notebook titled "exercici3.ipynb". The code in the cell is a CUDA C++ program. It includes #include statements for <iostream> and <fstream>. It defines INPUT_FILE and OUTPUT_FILE using preprocessor directives. It also defines INPUT_FILE and OUTPUT_FILE using absolute paths from Google Drive. The code then uses namespaces std and defines functions savePPM and loadPPM. The savePPM function writes grayscale data to a file in P5 format. The loadPPM function reads data from a file and returns a boolean value indicating success or failure.

```
%%cuda
#include <iostream>
#include <fstream>

//#define INPUT_FILE "test_image.ppm"
//#define OUTPUT_FILE "output_image.ppm"

//#define INPUT_FILE "/content/sample_data/test_image.ppm"
//#define OUTPUT_FILE "/content/sample_data/output_image.ppm"

#define INPUT_FILE "/content/drive/MyDrive/test_image.ppm"
#define OUTPUT_FILE "/content/drive/MyDrive/output_image.ppm"

using namespace std;

void savePPM(const char *filename,
             unsigned char *data,
             int width,
             int height) {
    ofstream file(filename, std::ios::binary);
    file << "P5\n" << width << " " << height << "\n255\n";

    // Write grayscale data
    file.write(reinterpret_cast<char*>(data), width * height);
    file.close();
}

bool loadPPM(const char *filename,
             unsigned char *data,
             int width,
             int height) {
    ifstream file(filename, std::ios::binary);
    if (!file) {
        cerr << "couldn't find file: " << filename << endl;
        return false;
    }

    string header...
```

The screenshot shows a Google Colab notebook titled "activitat4.ipynb". The notebook interface includes a toolbar with file operations, a sidebar with various icons, and a main workspace containing a list of numerical values. The values are displayed in a code cell, each consisting of five integers followed by a decimal point and a long floating-point number. The floating-point numbers all start with "0.03".

Value
17, 167, 59, 6, 0.033910575
17, 143, 59, 7, 0.033364718
17, 126, 59, 8, 0.032046294
17, 112, 59, 9, 0.031081368
17, 101, 59, 10, 0.030148177
17, 91, 59, 11, 0.029800915
17, 84, 59, 12, 0.029640127
17, 77, 59, 13, 0.029212537
17, 72, 59, 14, 0.030007159
17, 67, 59, 15, 0.029645426
17, 63, 59, 16, 0.030063122
15, 501, 67, 2, 0.031399992
15, 334, 67, 3, 0.036170825
15, 251, 67, 4, 0.036890587
15, 201, 67, 5, 0.034864910
15, 167, 67, 6, 0.033983692
15, 143, 67, 7, 0.032647581
15, 126, 67, 8, 0.031435341
15, 112, 67, 9, 0.030210613
15, 101, 67, 10, 0.029961406
15, 91, 67, 11, 0.029405082
15, 84, 67, 12, 0.030195771
15, 77, 67, 13, 0.029191311
15, 72, 67, 14, 0.030507220
13, 143, 77, 7, 0.029877891
13, 91, 77, 11, 0.030963385
11, 143, 91, 7, 0.030023892
10, 501, 101, 2, 0.032627387
10, 334, 101, 3, 0.037141338
10, 251, 101, 4, 0.036840455
10, 201, 101, 5, 0.034578795
10, 167, 101, 6, 0.032618381
10, 143, 101, 7, 0.032586690
10, 126, 101, 8, 0.031719146
10, 112, 101, 9, 0.032292935
7, 201, 143, 5, 0.031721458
6, 501, 167, 2, 0.031935367
6, 334, 167, 3, 0.033607599
6, 251, 167, 4, 0.033572151
6, 201, 167, 5, 0.032867922
4, 501, 251, 2, 0.032185681
4, 334, 251, 3, 0.033468541

At the bottom of the notebook, a status bar indicates "16 s" and "completado a las 16:14".