A large, faint, circular watermark of the University of Florence seal is visible on the left side of the slide, featuring a seated figure and the text "UNIVERSITAS FLORENTINA STUDIORUM".

Integral Image Generator

Parallel Programming for Machine Learning
Project Work in Artificial Intelligence Programming

Sofia Galante

Introduzione

Il progetto svolto `e un generatore di immagini integrali.

In un'immagine integrale, data un'immagine di partenza, ogni pixel `e ottenuto sommando se stesso con tutti i pixel precedenti (sia lungo l'asse x che lungo l'asse y).

Si sono creati diversi generatori di immagini integrali:
















- **un generatore sequenziale**: in questo caso si generano le immagini integrali con un algoritmo *sequenziale*;
- **due versioni di un generatore con CUDA**: in questo caso si utilizza la *GPU* per parallelizzare la creazione dell'immagine integrale;
- **tre versioni di un generatore con OpenMP**: in questo caso si compie una parallelizzazione a livello della *CPU* del codice tramite OpenMP (3 versioni).

Lo scopo del progetto è quello di osservare lo speedup ottenuto nelle due versioni parallele dell'algoritmo, osservando anche quale parallelizzazione (tra GPU e CPU) risulta più efficiente.

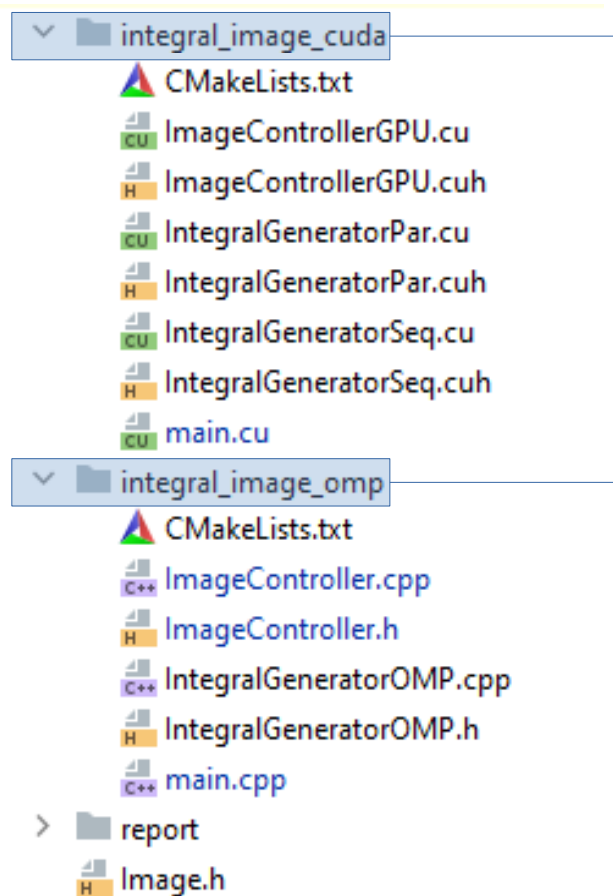
Tutti gli esperimenti sono stati svolti su un PC con sistema operativo Windows 10, una CPU Intel Core i5-11400 e una GPU RTX 3070Ti.

La versione sequenziale e quella parallelizzata a livello di GPU utilizzano Visual C++ come compilatore. Per quanto riguarda la versione con OpenMP, si è optato per l'utilizzo di MinGW.

Organizzazione del codice

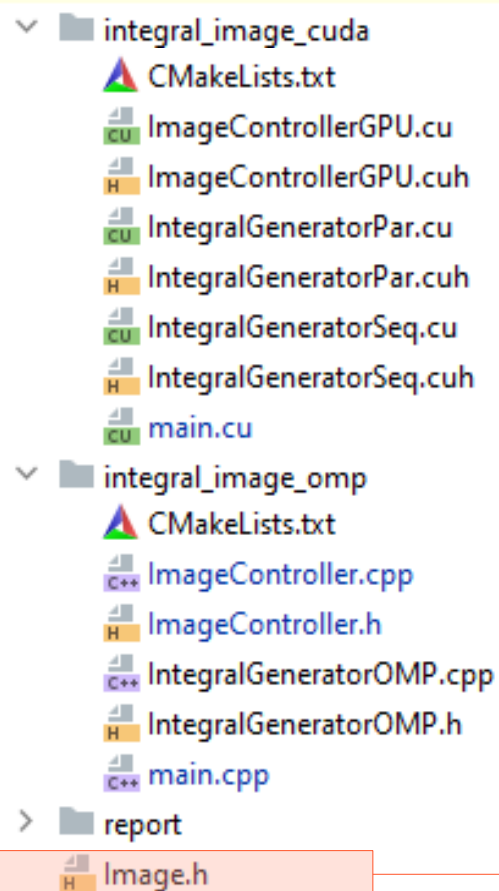
- ▼ integral_image_cuda
 -  CMakeLists.txt
 -  ImageControllerGPU.cu
 -  ImageControllerGPU.cuh
 -  IntegralGeneratorPar.cu
 -  IntegralGeneratorPar.cuh
 -  IntegralGeneratorSeq.cu
 -  IntegralGeneratorSeq.cuh
 -  main.cu
- ▼ integral_image_omp
 -  CMakeLists.txt
 -  ImageController.cpp
 -  ImageController.h
 -  IntegralGeneratorOMP.cpp
 -  IntegralGeneratorOMP.h
 -  main.cpp
- > report
 -  Image.h

Organizzazione del codice



due progetti diversi

Organizzazione del codice



contiene la definizione
di immagine e il valore
del SEED

Organizzazione del codice

- ▼ integral_image_cuda
 - ▲ CMakeLists.txt
 - CU ImageControllerGPU.cu
 - H ImageControllerGPU.cuh
 - CU IntegralGeneratorPar.cu
 - H IntegralGeneratorPar.cuh
 - CU IntegralGeneratorSeq.cu
 - H IntegralGeneratorSeq.cuh
 - CU main.cu
- ▼ integral_image_omp
 - ▲ CMakeLists.txt
 - C++ ImageController.cpp
 - H ImageController.h
 - C++ IntegralGeneratorOMP.cpp
 - H IntegralGeneratorOMP.h
 - C++ main.cpp
- > report
 - H Image.h

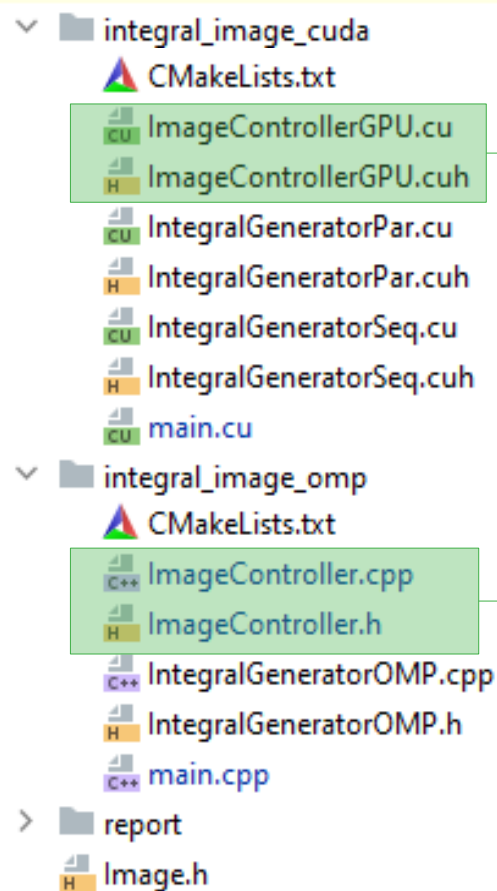
```
#ifndef SEED
#define SEED 111
#endif //SEED

#ifndef INTEGRAL_IMAGEOMP_IMAGE_H
#define INTEGRAL_IMAGEOMP_IMAGE_H

struct Image {
    int * pixels = nullptr;
    int width = 0;
    int height = 0;
};

#endif //INTEGRAL_IMAGEOMP_IMAGE_H
```

Organizzazione del codice



file che contengono
le funzioni per gestire
le immagini

Organizzazione del codice

- integral_image_cuda
 - CMakeLists.txt
 - ImageControllerGPU.cu
 - ImageControllerGPU.cuh
 - IntegralGeneratorPar.cu
 - IntegralGeneratorPar.cuh
 - IntegralGeneratorSeq.cu
 - IntegralGeneratorSeq.cuh
 - main.cu
 - integral_image_omp
 - CMakeLists.txt
 - ImageController.cpp
 - ImageController.h
 - IntegralGeneratorOMP.cpp
 - IntegralGeneratorOMP.h
 - main.cpp
 - report
 - Image.h

```
__host__ Image generateImage(int width, int height);
__host__ Image copyImage(Image const &image);

__host__ Image allocateOnDevice(Image const &hostImage);

__host__ void freeImageHost(Image &hostImage);
__host__ void freeImageDev(Image &devImage);

__host__ void copyFromHostToDevice(Image const &hostImage, Image &devImage);
__host__ void copyFromDeviceToHost(Image const &devImage, Image &hostImage);

__host__ void printImage(Image const &image);

__host__ bool areTheSame(Image const &image1, Image const &image2);
```

```
Image generateImage(int width, int height);
Image copyImage(Image const &image);

void freeImage(Image &hostImage);

void printImage(Image const &image);
```


Organizzazione del codice

- integral_image_cuda
 - CMakeLists.txt
 - ImageControllerGPU.cu
 - ImageControllerGPU.cuh
 - IntegralGeneratorPar.cu
 - IntegralGeneratorPar.cuh
 - IntegralGeneratorSeq.cu
 - IntegralGeneratorSeq.cuh
 - main.cu
 - integral_image_omp
 - CMakeLists.txt
 - ImageController.cpp
 - ImageController.h
 - IntegralGeneratorOMP.cpp
 - IntegralGeneratorOMP.h
 - main.cpp
 - report
 - Image.h

```

__host__ Image generateImage(int width, int height);
__host__ Image copyImage(Image const &image);

__host__ Image allocateOnDevice(Image const &hostImage);

__host__ void freeImageHost(Image &hostImage);
__host__ void freeImageDev(Image &devImage);

__host__ void copyFromHostToDevice(Image const &hostImage, Image &devImage);
__host__ void copyFromDeviceToHost(Image const &devImage, Image &hostImage);

__host__ void printImage(Image const &image);

__host__ bool areTheSame(Image const &image1, Image const &image2);

```

```

Image generateImage(int width, int height);
Image copyImage(Image const &image);

void freeImage(Image &hostImage);

void printImage(Image const &image);

```

Organizzazione del codice

- ▼ integral_image_cuda
 - ▲ CMakeLists.txt
 - CU ImageControllerGPU.cu
 - H ImageControllerGPU.cuh

- CU IntegralGeneratorPar.cu
 - H IntegralGeneratorPar.cuh
 - CU IntegralGeneratorSeq.cu
 - H IntegralGeneratorSeq.cuh

- CU main.cu

- ▼ integral_image_omp
 - ▲ CMakeLists.txt
 - C++ ImageController.cpp
 - H ImageController.h

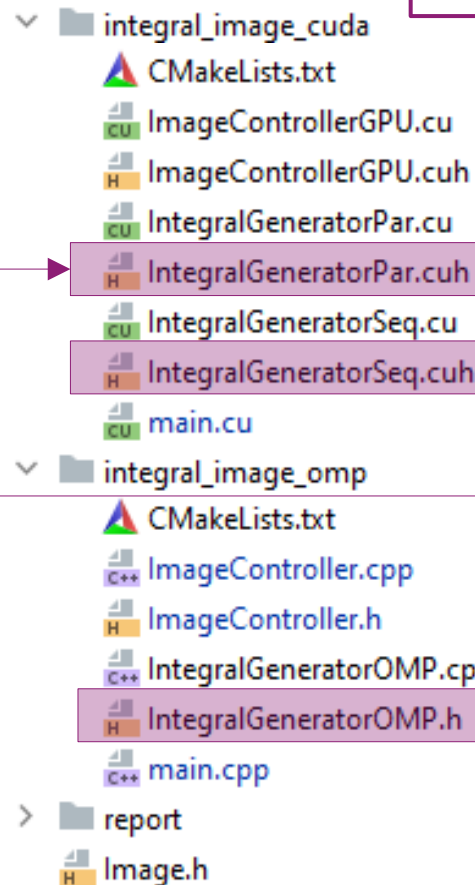
- C++ IntegralGeneratorOMP.cpp
 - H IntegralGeneratorOMP.h

- C++ main.cpp

- > report
 - H Image.h

file che contengono
le funzioni per generare
le immagini integrali

Organizzazione del codice



```
__host__ void generateIntegralCPUseq(int width, int height, int const * original, int * result);
```

```
__global__ void generateIntegralGPUglobalMem(int width, int height, int const * original, int * result);
__global__ void generateIntegralGPUsharedMem(int width, int height, int const * original, int * result);

__host__ void setUp(Image const &original, Image const &result, Image &dev_original, Image &dev_result);
__host__ void finish(Image &dev_original, Image &dev_result, Image &result);
```

```
void generateIntegralCPUompFirstLevel(int width, int height, int const * original, int * result, int threads);
void generateIntegralCPUompSecondLevel(int width, int height, int const * original, int * result, int threads);
void generateIntegralCPUompBothLevels(int width, int height, int const * original, int * result, int threads1, int threads2);
```

Algoritmo sequenziale

```
__host__ void generateIntegralCPUseq(int width, int height, int const * original, int * result){  
    for(int row = 0; row < height; row++){  
        for(int col = 0; col < width; col++){  
            int value = 0;  
            for(int y = row; y >= 0; y--){  
                for(int x = col; x >= 0; x--){  
                    value += original[y * width + x];  
                }  
            }  
            result[row * width + col] = value;  
        }  
    }  
}
```

Algoritmo parallelo - GPU

Utilizzo solo della global memory

```
__global__ void generateIntegralGPUglobalMem(int width, int height, int const * original, int * result){  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
  
    int row = by * blockDim.y + ty;  
    int col = bx * blockDim.x + tx;  
  
    int value = 0;  
    if(col < width && row < height){  
        for(int y = row; y >= 0; y--){  
            for(int x = col; x >= 0; x--){  
                value += original[y * width + x];  
            }  
        }  
        result[row * width + col] = value;  
    }  
}
```

Algoritmo parallelo - GPU

Utilizzo solo della global memory

```
__global__ void generateIntegralGPUglobalMem(int width, int height, int const * original, int * result){  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
  
    int row = by * blockDim.y + ty;  
    int col = bx * blockDim.x + tx;  
  
    int value = 0;  
    if(col < width && row < height){  
        for(int y = row; y >= 0; y--){  
            for(int x = col; x >= 0; x--){  
                value += original[y * width + x];  
            }  
        }  
        result[row * width + col] = value;  
    }  
}
```

sostituisce i primi
due for

Algoritmo parallelo - GPU

Utilizzo solo della global memory

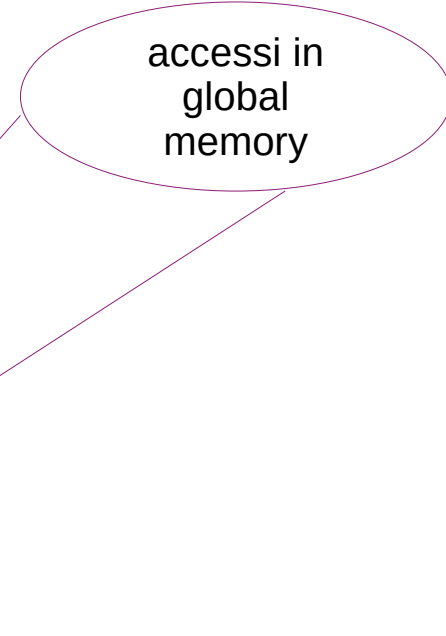
```
__global__ void generateIntegralGPUglobalMem(int width, int height, int const * original, int * result){  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
  
    int row = by * blockDim.y + ty;  
    int col = bx * blockDim.x + tx;  
  
    int value = 0;  
    if(col < width && row < height){  
        for(int y = row; y >= 0; y--){  
            for(int x = col; x >= 0; x--){  
                value += original[y * width + x];  
            }  
        }  
        result[row * width + col] = value;  
    }  
}
```

controllo per non
andare fuori
dall'immagine

Algoritmo parallelo - GPU

Utilizzo solo della global memory

```
__global__ void generateIntegralGPUglobalMem(int width, int height, int const * original, int * result){  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
  
    int row = by * blockDim.y + ty;  
    int col = bx * blockDim.x + tx;  
  
    int value = 0;  
    if(col < width && row < height){  
        for(int y = row; y >= 0; y--){  
            for(int x = col; x >= 0; x--){  
                value += original[y * width + x];  
            }  
        }  
        result[row * width + col] = value;  
    }  
}
```



Algoritmo parallelo - GPU

Utilizzo della shared memory - 1

```
__global__ void generateIntegralGPUsharedMem(int width, int height, int const * original, int * result){  
    //Utilizzo della shared memory  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
    int w = blockDim.x;  
    int h = blockDim.y;  
  
    int row = by * h + ty;  
    int col = bx * w + tx;  
    extern __shared__ int sharedOriginal[];  
    int value = 0;  
    int _row, _col, _x, _y;  
    for(int _by = 0; _by <= by; _by++){  
        for(int _bx = 0; _bx <= bx; _bx++){ //itera tra i blocchi  
            _row = _by * h + ty;  
            _col = _bx * w + tx;  
            if(_col < width && _row < height)  
                sharedOriginal[ty * w + tx] = original[_row * width + _col];  
            else  
                sharedOriginal[ty * w + tx] = 0;  
            __syncthreads(); //ogni thread scrive nella shared memory e poi aspetta
```

Algoritmo parallelo - GPU

Utilizzo della shared memory - 1

```
__global__ void generateIntegralGPUsharedMem(int width, int height, int const * original, int * result){
    //Utilizzo della shared memory
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int w = blockDim.x;
    int h = blockDim.y;

    int row = by * h + ty;
    int col = bx * w + tx;
    extern __shared__ int sharedOriginal[];
    int value = 0;
    int _row, _col, _x, _y;
    for(int _by = 0; _by <= by; _by++){
        for(int _bx = 0; _bx <= bx; _bx++){ //itera tra i blocchi
            _row = _by * h + ty;
            _col = _bx * w + tx;
            if(_col < width && _row < height)
                sharedOriginal[ty * w + tx] = original[_row * width + _col];
            else
                sharedOriginal[ty * w + tx] = 0;
        }
    }
    __syncthreads(); //ogni thread scrive nella shared memory e poi aspetta
}
```

alloco dinamicamente
la shared memory

Algoritmo parallelo - GPU

Utilizzo della shared memory - 1

```
__global__ void generateIntegralGPUsharedMem(int width, int height, int const * original, int * result){  
    //Utilizzo della shared memory  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
    int w = blockDim.x;  
    int h = blockDim.y;  
  
    int row = by * h + ty;  
    int col = bx * w + tx;  
    extern __shared__ int sharedOriginal[];  
    int value = 0;  
    int _row, _col, _x, _y;  
    for(int _by = 0; _by <= by; _by++){  
        for(int _bx = 0; _bx <= bx; _bx++){ //itera tra i blocchi  
            _row = _by * h + ty;  
            _col = _bx * w + tx;  
            if(_col < width && _row < height)  
                sharedOriginal[ty * w + tx] = original[_row * width + _col];  
            else  
                sharedOriginal[ty * w + tx] = 0;  
            __syncthreads(); //ogni thread scrive nella shared memory e poi aspetta
```

divisione in fasi
dell'algoritmo

Algoritmo parallelo - GPU

Utilizzo della shared memory - 1

```
__global__ void generateIntegralGPUsharedMem(int width, int height, int const * original, int * result){  
    //Utilizzo della shared memory  
    int bx = blockIdx.x;  
    int by = blockIdx.y;  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
    int w = blockDim.x;  
    int h = blockDim.y;  
  
    int row = by * h + ty;  
    int col = bx * w + tx;  
    extern __shared__ int sharedOriginal[];  
    int value = 0;  
    int _row, _col, _x, _y;  
    for(int _by = 0; _by <= by; _by++){  
        for(int _bx = 0; _bx <= bx; _bx++){ //itera tra i blocchi  
            _row = _by * h + ty;  
            _col = _bx * w + tx;  
  
            if(_col < width && _row < height)  
                sharedOriginal[ty * w + tx] = original[_row * width + _col];  
            else  
                sharedOriginal[ty * w + tx] = 0;  
            __syncthreads(); //ogni thread scrive nella shared memory e poi aspetta
```

copia i valori
nella shared memory
(tiling)



Algoritmo parallelo - GPU

Utilizzo della shared memory - 2

```

    if(_bx < bx){
        _x = w-1;
    }
    else{
        _x = tx;
    }

    if(_by < by){
        _y = h-1;
    }
    else{
        _y = ty;
    }

    if(col < width && row < height){
        for(int y = _y; y >= 0; y--){
            for(int x = _x; x >= 0; x--){
                value += sharedOriginal[y * w + x];
            }
        }
    }

    __syncthreads();
}

if(col < width && row < height)
    result[row * width + col] = value;
}
```

Algoritmo parallelo - GPU

Utilizzo della shared memory - 2

```
if(_bx < bx){
    _x = w-1;
}
else{
    _x = tx;
}

if(_by < by){
    _y = h-1;
}
else{
    _y = ty;
}

if(col < width && row < height){
    for(int y = _y; y >= 0; y--){
        for(int x = _x; x >= 0; x--){
            value += sharedOriginal[y * w + x];
        }
    }
}

__syncthreads();

if(col < width && row < height)
    result[row * width + col] = value;
```

determina quali
valori sono da
utilizzare

Algoritmo parallelo - GPU

Utilizzo della shared memory - 2

```
if(_bx < bx){
    _x = w-1;
}
else{
    _x = tx;
}

if(_by < by){
    _y = h-1;
}
else{
    _y = ty;
}

if(col < width && row < height){
    for(int y = _y; y >= 0; y--){
        for(int x = _x; x >= 0; x--){
            value += sharedOriginal[y * w + x];
        }
    }
}
__syncthreads();

if(col < width && row < height)
    result[row * width + col] = value;
```

calcolo parziale
del valore
cercato

Algoritmo parallelo - GPU

Utilizzo della shared memory - 2

```
if(_bx < bx){
    _x = w-1;
}
else{
    _x = tx;
}

if(_by < by){
    _y = h-1;
}
else{
    _y = ty;
}

if(col < width && row < height){
    for(int y = _y; y >= 0; y--){
        for(int x = _x; x >= 0; x--){
            value += sharedOriginal[y * w + x];
        }
    }
}

__syncthreads();

if(col < width && row < height)
    result[row * width + col] = value;
```

copia del valore
definitivo nella
global memory

Algoritmo parallelo - CPU

Parallelizzazione al “primo livello”

```
void generateIntegralCPUompFirstLevel(int width, int height, int const * original, int * result, int threads){  
    #pragma omp parallel for collapse(2) default(none) shared(original, result) firstprivate(width, height) \  
    num_threads(threads)  
    for(int row = 0; row < height; row++){  
        for(int col = 0; col < width; col++){  
            int value = 0;  
            for(int y = row; y >= 0; y--){  
                for(int x = col; x >= 0; x--){  
                    value += original[y * width + x];  
                }  
            }  
            result[row * width + col] = value;  
        }  
    }  
}
```



Algoritmo parallelo - CPU

Parallelizzazione al “primo livello”

```
void generateIntegralCPUompFirstLevel(int width, int height, int const * original, int * result, int threads){  
#pragma omp parallel for collapse(2) default(none) shared(original, result) firstprivate(width, height) \  
num_threads(threads)  
{  
    for(int row = 0; row < height; row++){  
        for(int col = 0; col < width; col++){  
            int value = 0;  
            for(int y = row; y >= 0; y--){  
                for(int x = col; x >= 0; x--){  
                    value += original[y * width + x];  
                }  
            }  
            result[row * width + col] = value;  
        }  
    }  
}
```

parallel for

Algoritmo parallelo - CPU

Parallelizzazione al “secondo livello”

```
void generateIntegralCPUompSecondLevel(int width, int height, int const * original, int * result, int threads){  
    for(int row = 0; row < height; row++){  
        for(int col = 0; col < width; col++){  
            int value = 0;  
#pragma omp parallel for collapse(2) default(none) shared(original) firstprivate(row, col, width) \  
num_threads(threads) reduction(+: value)  
            for(int y = row; y >= 0; y--){  
                for(int x = col; x >= 0; x--){  
                    value += original[y * width + x];  
                }  
            }  
            result[row * width + col] = value;  
        }  
    }  
}
```

Algoritmo parallelo - CPU

Parallelizzazione al “secondo livello”

parallel for
con reduction

```
void generateIntegralCPUompSecondLevel(int width, int height, int const * original, int * result, int threads){  
    for(int row = 0; row < height; row++){  
        for(int col = 0; col < width; col++){  
            int value = 0;  
            #pragma omp parallel for collapse(2) default(none) shared(original) firstprivate(row, col, width) \  
            num_threads(threads) reduction(+: value)  
                for(int y = row; y >= 0; y--){  
                    for(int x = col; x >= 0; x--){  
                        value += original[y * width + x];  
                    }  
                }  
            result[row * width + col] = value;  
        }  
    }  
}
```

Algoritmo parallelo - CPU

Parallelizzazione ad “entrambi i livelli”

```
void generateIntegralCPUompBothLevels(int width, int height, int const * original, int * result, int threads1, int threads2){  
    #pragma omp parallel for collapse(2) default(none) shared(original, result) firstprivate(width, height, threads2) \  
    num_threads(threads1)  
    for(int row = 0; row < height; row++){  
        for(int col = 0; col < width; col++){  
            int value = 0;  
            #pragma omp parallel for collapse(2) default(none) shared(original) firstprivate(row, col, width) \  
            num_threads(threads2) reduction(+: value)  
            for(int y = row; y >= 0; y--){  
                for(int x = col; x >= 0; x--){  
                    value += original[y * width + x];  
                }  
            }  
            result[row * width + col] = value;  
        }  
    }  
}
```



Algoritmo parallelo - CPU

Parallelizzazione ad “entrambi i livelli”

```
void generateIntegralCPUompBothLevels(int width, int height, int const * original, int * result, int threads1, int threads2){  
#pragma omp parallel for collapse(2) default(none) shared(original, result) firstprivate(width, height, threads2) \  
num_threads(threads1)  
    for(int row = 0; row < height; row++){  
        for(int col = 0; col < width; col++){  
            int value = 0;  
#pragma omp parallel for collapse(2) default(none) shared(original) firstprivate(row, col, width) \  
num_threads(threads2) reduction(+: value)  
                for(int y = row; y >= 0; y--){  
                    for(int x = col; x >= 0; x--){  
                        value += original[y * width + x];  
                    }  
                }  
            result[row * width + col] = value;  
        }  
    }  
}
```

entrambi i
livelli di
parallelizzazione

Test

- ▼ integral_image_cuda
 - ▲ CMakeLists.txt
 - CU ImageControllerGPU.cu
 - H ImageControllerGPU.cuh
 - CU IntegralGeneratorPar.cu
 - H IntegralGeneratorPar.cuh
 - CU IntegralGeneratorSeq.cu
 - H IntegralGeneratorSeq.cuh
 - CU main.cu
- ▼ integral_image_omp
 - ▲ CMakeLists.txt
 - C++ ImageController.cpp
 - H ImageController.h
 - C++ IntegralGeneratorOMP.cpp
 - H IntegralGeneratorOMP.h
 - C++ main.cpp
- > report
 - H Image.h

i test si trovano
nei due file main

Test

- integral_image_cuda
 - CMakeLists.txt
 - ImageControllerGPU.cu
 - ImageControllerGPU.cuh
 - IntegralGeneratorPar.cu
 - IntegralGeneratorPar.cuh
 - IntegralGeneratorSeq.cu
 - IntegralGeneratorSeq.cuh
 - main.cu
 - integral_image_omp
 - CMakeLists.txt
 - ImageController.cpp
 - ImageController.h
 - IntegralGeneratorOMP.cpp
 - IntegralGeneratorOMP.h
 - main.cpp
 - report
 - Image.h

```
__host__ double CPU_sequential(Image const &original, Image &result){...}
__host__ double GPU_globalMem(Image const &original, Image &result, dim3 grid, dim3 block){...}
__host__ double GPU_sharedMem(Image const &original, Image &result, dim3 grid, dim3 block){...}

__host__ void dimTest(std::string const &testName){...}
__host__ void gridTest(std::string const &fileName, int d, int GB[4][4]){...}
__host__ void gridTest1(std::string const &testName){...}
__host__ void gridTest2(std::string const &testName){...}

int main(int argc, char *argv[]) {
    if(argc != 2){
        printf(Format: "Manca il parametro\n");
        return 1;
    }
    std::string testName = argv[1];
    dimTest(testName);
    gridTest1(testName);
    gridTest2(testName);
    return 0;
}
```

```
double CPU_omp(Image const &original, Image &result, int * threads, int level){...}
void dimTest(std::string const &testName, double * time){...}
void threadsTest(std::string const &testName){...}
void threadsTestV2(std::string const &testName){...}

int main(int argc, char *argv[]) {
    if(argc != 2){
        printf(format: "Manca il parametro\n");
        return 1;
    }
    omp_set_nested(4);
    std::string testName = argv[1];
    threadsTest(testName);
    threadsTestV2(testName);
    return 0;
}
```


Test 1 v1 - CUDA

Studio sulle dimensioni di grid e block (casi limite)

- dimensione immagine = {10x10, 100x100, 1000x1000}
- dimensione grid e block come nella tabella

		dim 10x10	dim 100x100	dim 1000x1000
righe	grid	1x10	1x100	1x1000
	block	10x1	100x1	1000x1
colonne	grid	10x1	100x1	1000x1
	block	1x10	1x100	1x1000
max n° blocks	grid	10x10	100x100	255x255
	block	1x1	1x1	4x4
max n° threads	grid	1x1	4x4	32x32
	block	10x10	32x32	32x32

Test 1 v1 - CUDA

Studio sulle dimensioni di grid e block (casi limite)

	GPU - global	GPU - shared
1 x 10 - 10 x 1	0.047 ms	0.039 ms
10 x 1 - 1 x 10	0.041 ms	0.046 ms
10 x 10 - 1 x 1	0.056 ms	0.061 ms
1 x 1 - 10 x 10	0.042 ms	0.038 ms

Immagine
10 x 10

Immagine
100 x 100

	GPU - global	GPU - shared
1 x 100 - 100 x 1	0.236 ms	0.131 ms
100 x 1 - 1 x 100	1.005 ms	0.808 ms
100 x 100 - 1 x 1	1.054 ms	11.78 ms
4 x 4 - 32 x 32	0.715 ms	0.386 ms

	GPU - global	GPU - shared
1 x 1000 - 1000 x 1	829.907 ms	202.801 ms
1000 x 1 - 1 x 1000	5500.471 ms	912.196 ms
255 x 255 - 4 x 4	1783.244 ms	969.084 ms
32 x 32 - 32 x 32	807.314 ms	229.303 ms

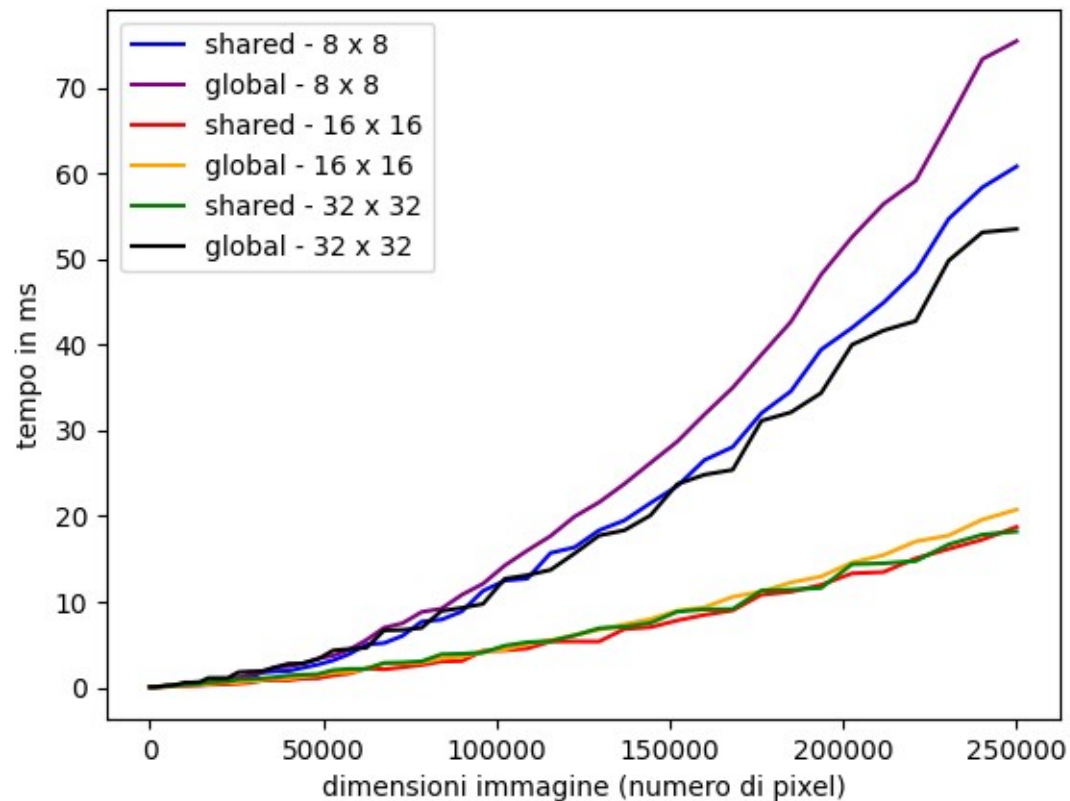
Immagine
1000 x 1000

Test 1 v2 - CUDA

Studio sulle dimensioni di grid e block (numero di thread multiplo di 32)

global memory & shared memory

- dimensione immagine che varia da 10x10 a 500x500 (aumentando il lato di 10 in 10)
- dimensione block = {8x8, 16x16, 32x32}

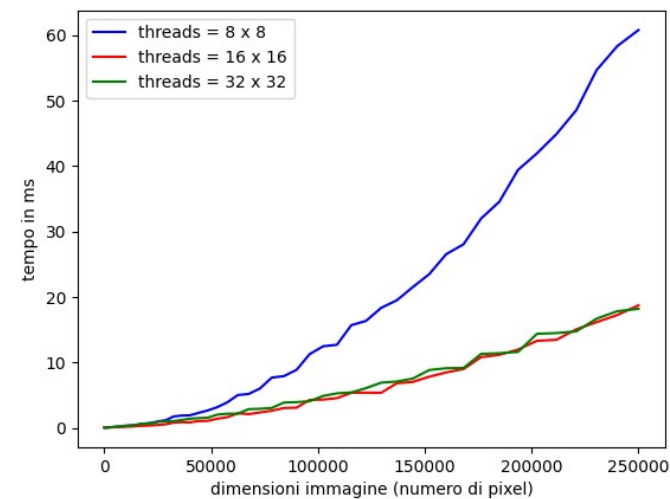
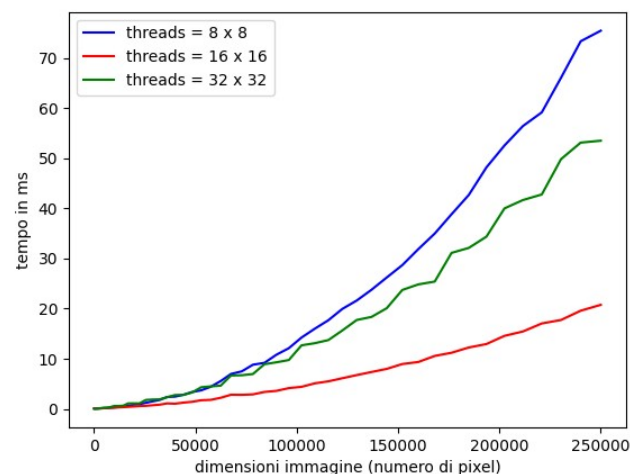


Test 1 v2 - CUDA

Studio sulle dimensioni di grid e block (numero di thread multiplo di 32)

global memory & shared memory

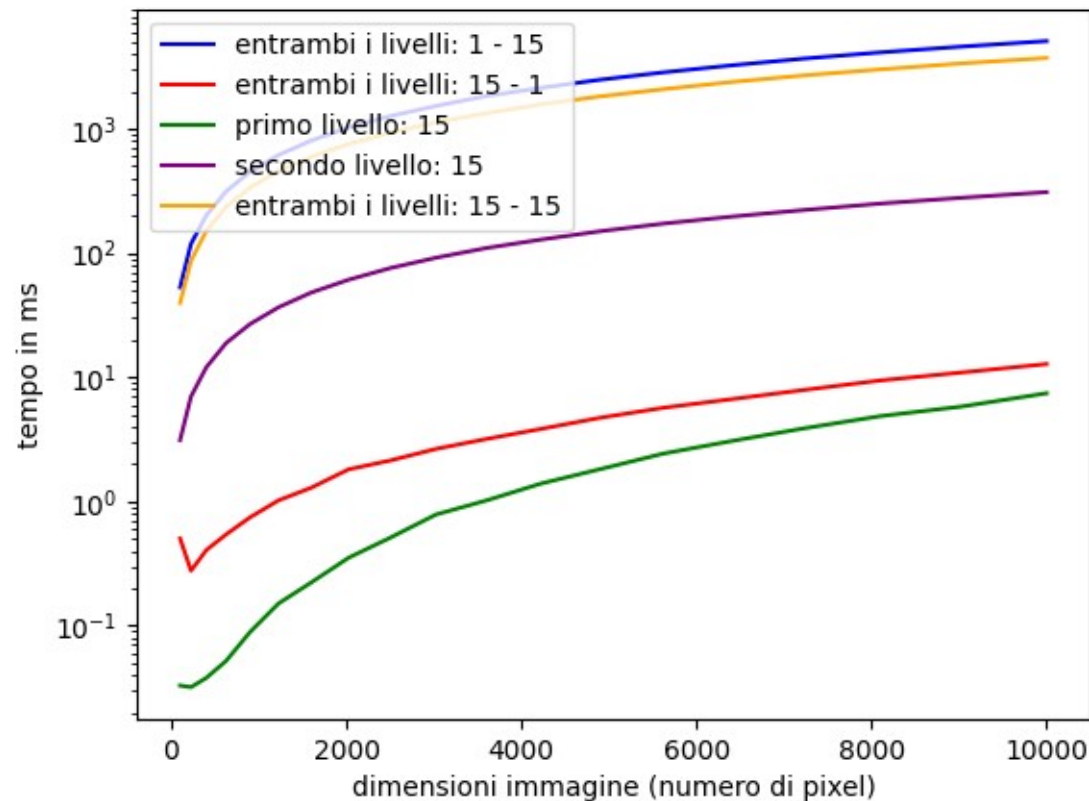
- dimensione immagine che varia da 10x10 a 500x500 (aumentando il lato di 10 in 10)
- dimensione block = {8x8, 16x16, 32x32}



Test 2 v1 - OpenMP

Studio sul tipo di parallelizzazione migliore

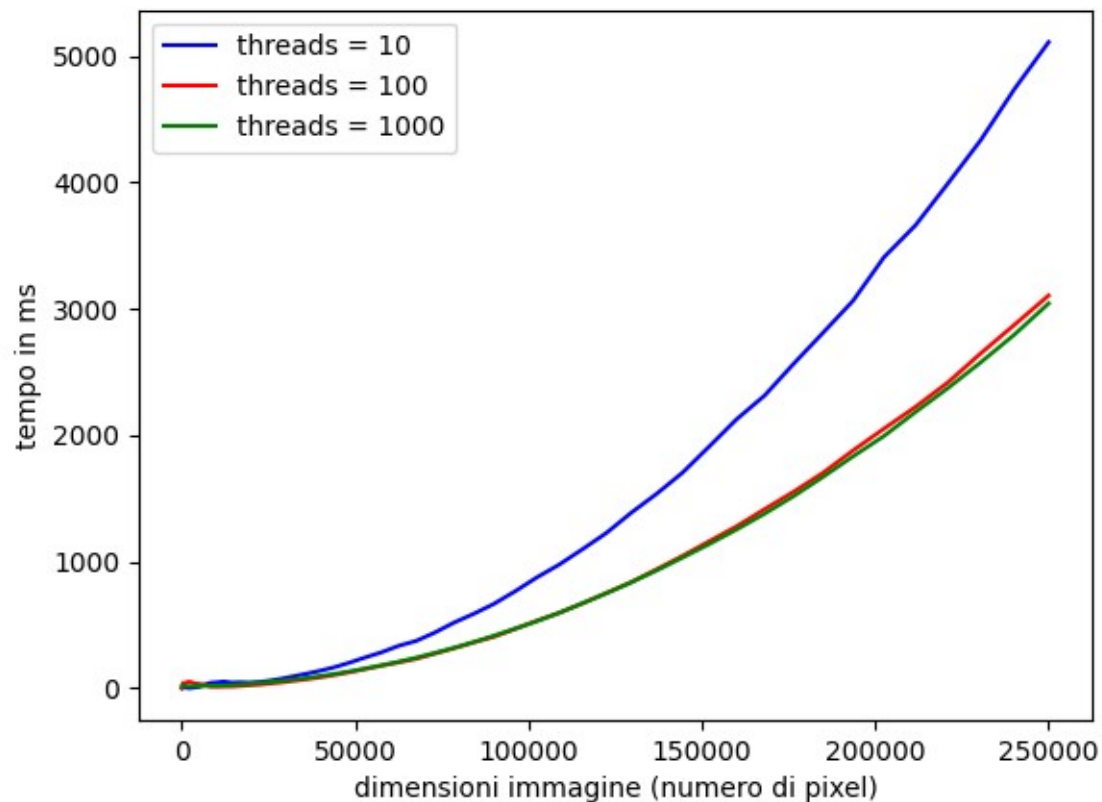
- dimensione dell'immagine che varia da 10x10 a 100x100 (aumentando il lato di 5 in 5)
- livelli e thread:
 - entrambi i livelli, 1 e 15 thread
 - entrambi i livelli, 15 e 1 thread
 - primo livello, 15 thread
 - secondo livello, 15 thread
 - entrambi i livelli, 15 e 15 thread



Test 2 v2 - OpenMP

Aumento del numero di thread (primo livello)

- dimensione dell'immagine che varia da 10x10 a 500x500 (aumentando il lato di 10 in 10)
- numero di thread = {10, 100, 1000}



Test 3

Aumento della dimensione dell'immagine

- dimensione dell'immagine che varia da 10x10 a 500x500 (aumentando il lato di 10 in 10)
- OpenMP – primo livello con 100 thread
- CUDA – shared memory con dimensione dei block = 16x16

