Confronto tra versione sequenziale e versioni parallelizzate con CUDA e OpenMP di convoluzioni

Silvia Dani e Niccolò Niccoli

Introduzione

- L’obiettivo è eseguire delle convoluzioni (utilizzando sia kernel separabili che non) su di un’immagine.
- Il codice viene eseguito in modo sequenziale, parallelo con OpenMP e parallelo con CUDA.

Progettazione

La funzione che applica il filtro esamina in sequenza tutti i pixel dell'immagine.

Il filtro che viene applicato è un box blur.

I pixel di bordo sono stati gestiti considerando quelli fuori dall'immagine come pixel neri.

Parallelizzazione con OpenMP

L’immagine è stata divisa in chunk in base al numero di thread che si sceglie di utilizzare.

Ciascun thread applica il filtro sui pixel appartenenti alla porzione di immagine che gli è stata assegnata.

Esempio di come sono stati divise le immagini per permettere la parallelizzazione.

Parallelizzazione con OpenMP

```
#pragma omp parallel default(none) shared (src, dst, kernel, tileHeight, numProcs, kernelSize, offset)
#pragma omp for
    for (int threadIdx = 0; threadIdx < numProcs; threadIdx++)
        filter(src, dst, kernel, kernelSize, offset, tileHeight * threadIdx, tileHeight * (threadIdx + 1));
```

```
void filter(cv::Mat* src, cv::Mat* dst, std::vector<double> kernel, int kernelSize, int offset, int y0, int y1){
    for(int y = y0; y < y1; y++){
        for(int x = 0; x < src->cols; x++){
            for(int channel = 0; channel < src->channels(); channel++) {
                double convolutedValue = 0;
                for(int i = 0; i<kernelSize; i++){
                    for(int j = 0; j<kernelSize; j++){
                        if(x + j >= offset && x + j < src->cols + offset && y + i >= offset && y + i < src->rows +
offset){
                            convolutedValue += src->data[(y+i - offset) * src->step + (x+j - offset) * src->channels()
+ channel] * kernel[i*kernelSize + j];
                        }}}
                dst->data[y * dst->step + x * dst->channels() + channel] = static_cast<uchar>(convolutedValue);
            }}}};
```

Parallelizzazione con OpenMP - filtri separabili

Nel caso della parallelizzazione che sfrutta filtri separabili l'immagine viene sempre divisa in chunk e poi viene eseguita una convoluzione con un filtro colonna e poi una con uno riga.

```
#pragma omp parallel default(none) shared (src, dst,
kernel_col, kernelSize, tileHeight, numProcs, offset,
intermediate)
#pragma omp for
    for (int threadIdx = 0; threadIdx < numProcs;
threadIdx++) {
        applyColumnFilter(src, &intermediate,
kernel_col, kernelSize, offset, tileHeight *
threadIdx, tileHeight * (threadIdx + 1));
    }
#pragma omp for
    for (int threadIdx = 0; threadIdx < numProcs;
threadIdx++) {
        applyRowFilter(&intermediate, dst, kernel_row,
kernelSize, offset, tileHeight * threadIdx, tileHeight
* (threadIdx + 1));
    }
```

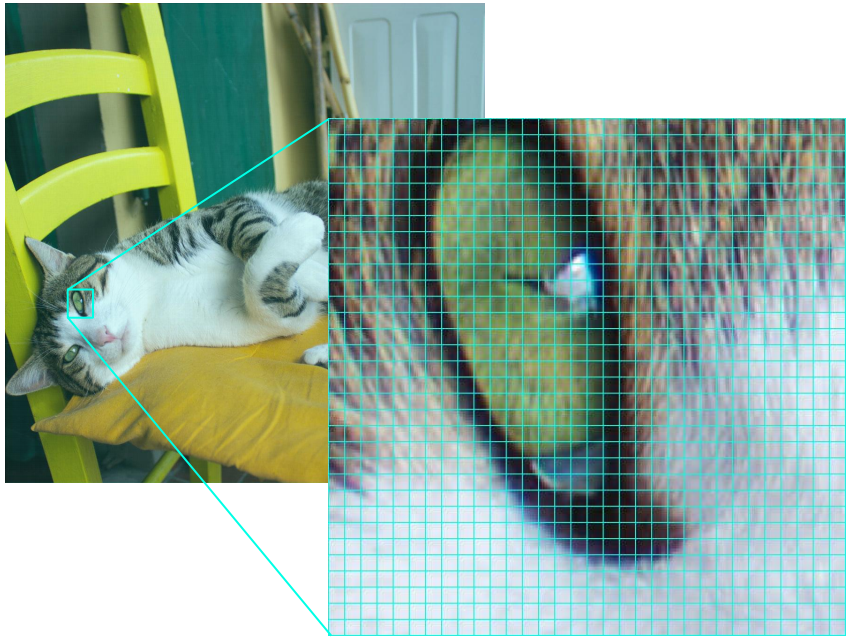
Parallelizzazione con CUDA

L'immagine è stata divisa in tile.

È stato scelto di utilizzare l'approccio in cui tutti i thread di un blocco caricano in memoria condivisa ma solo alcuni eseguono la convoluzione.

Le dimensioni del blocco e della griglia dipendono sia dall'immagine su cui deve essere applicato il filtro, sia dal tipo di filtro che viene applicato.

```
#define BLOCK_WIDTH (TILE_WIDTH + MASK_WIDTH - 1)
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);
dim3 dimGrid((src.cols - 1) / TILE_WIDTH + 1,
(src.rows - 1) / TILE_WIDTH + 1, src.channels());
```



Esempio di come la tassellatura è molto più fitta rispetto al caso precedente.

Parallelizzazione con CUDA

Per migliorare i tempi di accesso alla memoria vengono utilizzate le keyword `const` e `__restrict__` così da dire al compilatore che la locazione a cui punta un certo puntatore viene solo letta e mai sovrascritta.

```
__global__ void convolution(const uchar* __restrict__ src, int srcWidth, int srcHeight, int
srcChannels,const float* __restrict__ convKernel, int kernelWidth, int kernelHeight, uchar* dst)
```

Parallelizzazione con CUDA

```
__global__ void convolution(const uchar* __restrict__ src, int srcWidth, int srcHeight, int srcChannels,const float*
__restrict__ convKernel, int kernelWidth, int kernelHeight, uchar* dst){
    __shared__ uchar Ns[BLOCK_WIDTH][BLOCK_WIDTH];
    int mask_radius_w = kernelWidth/2, mask_radius_h = kernelHeight/2;
    int tx = threadIdx.x, ty = threadIdx.y;
    int row_o = blockIdx.y * TILE_WIDTH + ty, col_o = blockIdx.x * TILE_WIDTH + tx;
    int row_i = row_o - mask_radius_h, col_i = col_o - mask_radius_w;
    if((row_i >= 0) && (row_i < srcHeight) && (col_i >= 0) && (col_i < srcWidth)){
        Ns[ty][tx] = src[(row_i * srcWidth + col_i) * srcChannels + blockIdx.z];
    }else{
        Ns[ty][tx]=0.0f;
    }
    __syncthreads();
    float output = 0.0f;
    if(ty < TILE_WIDTH && tx < TILE_WIDTH){
        for(int i = 0; i < kernelHeight; i++)
            for(int j = 0; j < kernelWidth; j++)
                output += convKernel[i * kernelWidth + j] * (float)Ns[i+ty][j+tx];
        if(row_o < srcHeight && col_o < srcWidth)
            dst[(row_o * srcWidth + col_o) * srcChannels + blockIdx.z] = static_cast<uchar>(output);
    }
    __syncthreads();
}
```

Parallelizzazione con CUDA - filtri separabili

Nel caso della parallelizzazione che sfrutta filtri separabili vengono chiamati due kernel, il primo che utilizza un filtro colonna e il secondo che utilizza un filtro riga.

```
sepColConvolution<<<dimGrid, dimBlock>>>(d_src, src.cols, src.rows, src.channels(), d_filterCol, MASK_WIDTH, d_mid_dst);
sepRowConvolution<<<dimGrid, dimBlock>>>(d_mid_dst, src.cols, src.rows, src.channels(), d_filterRow, MASK_WIDTH, d_dst);
```

Parallelizzazione con CUDA - filtri separabili

```
__global__ void sepColConvolution(const uchar* __restrict__
src, int srcWidth, int srcHeight, int srcChannels,const
float* __restrict__ convKernel_col, int kernelHeight, uchar*
dst){
    __shared__ uchar Ns[BLOCK_WIDTH * BLOCK_WIDTH];
    int mask_radius_h = kernelHeight/2;
    int tx = threadIdx.x, ty = threadIdx.y;
    int row_o = blockIdx.y * TILE_WIDTH + ty;
    int col_o = blockIdx.x * TILE_WIDTH + tx;
    int row_i = row_o - mask_radius_h, col_i = col_o;
    if((row_i >= 0) && (row_i < srcHeight)){
        Ns[tx + ty * BLOCK_WIDTH] = src[(row_i * srcWidth +
col_i) * srcChannels + blockIdx.z];
    }else
        Ns[tx + ty * BLOCK_WIDTH]=0.0f;
    __syncthreads();
    float output = 0.0f;
    if(ty < TILE_WIDTH && tx < TILE_WIDTH){
        for(int i = 0; i < kernelHeight; i++)
                output += convKernel_col[i] * (float)Ns[tx +
(ty+i) * BLOCK_WIDTH];
        if(row_o < srcHeight && col_o < srcWidth){
            dst[(row_o * srcWidth + col_o) * srcChannels +
blockIdx.z] = static_cast<uchar>(output);
        }}
    __syncthreads();
}
```

```
__global__ void sepRowConvolution(const uchar* __restrict__
src, int srcWidth, int srcHeight, int srcChannels,const
float* __restrict__ convKernel_row, int kernelWidth, uchar*
dst){
    __shared__ uchar Ns[BLOCK_WIDTH * BLOCK_WIDTH];
    int tx = threadIdx.x, ty = threadIdx.y;
    int row_o = blockIdx.y * TILE_WIDTH + ty;
    int col_o = blockIdx.x * TILE_WIDTH + tx;
    int row_i = row_o, col_i = col_o - MASK_WIDTH/2;
    if((col_i >= 0) && (col_i < srcWidth)){
        Ns[tx + ty * BLOCK_WIDTH] = src[(row_i * srcWidth +
col_i) * srcChannels + blockIdx.z];
    }else
        Ns[tx + ty * BLOCK_WIDTH]=0.0f;
    __syncthreads();
    float output = 0.0f;
    if(ty < TILE_WIDTH && tx < TILE_WIDTH){
            for(int j = 0; j < kernelWidth; j++)
                output += convKernel_row[j] * (float)Ns[j +
tx + ty * BLOCK_WIDTH];
        if(row_o < srcHeight && col_o < srcWidth){
            dst[(row_o * srcWidth + col_o) * srcChannels +
blockIdx.z] = static_cast<uchar>(output);
        }}}
```

Esperimenti

Gli esperimenti che sono stati svolti consistono in:

- variare la dimensione del filtro (3x3, 5x5, 9x9, 15x15, 31x31)
- variare la dimensione dell'immagine (512x512, 1024x1024, 2048x2048, 4096x4096, 8192x8192)
- variare il numero di thread (2, 4, 8, 16, 32) (OpenMP)

L'hardware su cui sono stati svolti gli esperimenti è composto da un processore con 8 core logici e una scheda video Nvidia GeForce GTX 1650 (CC 7.5).

Il codice dove viene utilizzata CUDA è stato compilato con nvcc e msvc, quello dove viene utilizzato OpenMP invece con gcc.

Risultati

Utilizzando OpenMP l’accelerazione massima che è stata ottenuta è 5.493.

Non ci sono grosse differenze di accelerazione dovute all’aumento delle dimensioni dell’immagine.

Nel caso dell’utilizzo di filtri separabili l’accelerazione massima è 2.990.

L’accelerazione aumenta fintanto che il numero di thread utilizzati è minore o uguale del numero di core logici disponibili, dopodiché resta costante.

Immagine originale

Risultato della convoluzione

Risultati

Lo speed up ottenuto con CUDA è sensibilmente maggiore rispetto a quello ottenuto con OpenMP infatti il massimo che è stato registrato è 162.74.

Tuttavia non sempre si hanno delle performance così buone: nel caso del filtro 31x31 è stato necessario ridurre la dimensione del tile (da 16x16 a 2x2) e quindi l'accelerazione è stata molto minore.

Nel caso dell'utilizzo di filtri separabili, lo speed up massimo è stato 69.97.

Conclusioni

L’operazione di parallelizzazione comporta vantaggi sensibili dal punto di vista del tempo di esecuzione.

I vantaggi si notano di più se la parallelizzazione avviene utilizzando una scheda video o se si utilizzano dei filtri separabili.