Kernel Image Processing with CUDA

[B024314] Parallel Computing Exam

Tommaso Botarelli

23/04/2025

UNIVERSITÀ
DEGLI STUDI
FIRENZE
Da un secolo, oltre.

Table of Contents

▶ Kernel Image Processing
  Introduction
  Problem Definition
  Convolution Operation
  Boundary Handling
  Types of Kernel Filters
  Computational Complexity

Introduction

1 Kernel Image Processing

Image processing plays a crucial role in various fields, including computer vision, medical imaging, artificial intelligence...

kernel-based filtering is one of the fundamental techniques in image processing.

It involves applying a small matrix (kernel) to modify pixel values in an image.

This operation is used for tasks such as edge detection, blurring, and sharpening.

Problem Definition

1 Kernel Image Processing

Given:

- a three-dimensional matrix I of size $H \times W \times C$ (for a color image with C channels)
- a **convolution kernel** (or filter) K of size $k \times k$

The goal is to compute a new image I', where each pixel value $I'(x, y)$ is determined by applying the kernel K over a local neighborhood of $I(x, y)$.

Convolution Operation

Formally, the output pixel value at position (x, y) is computed as:

$$I'(x, y) = \sum_{i=-\frac{k}{2}}^{\frac{k}{2}} \sum_{j=-\frac{k}{2}}^{\frac{k}{2}} K(i, j) \cdot I(x + i, y + j)$$

where:

- $K(i, j)$ is the kernel weight at position (i, j).
- $I(x + i, y + j)$ is the pixel value from the input image at the corresponding location.
- The summation iterates over all elements of the kernel, applying the weighted sum over the local neighborhood.

Boundary Handling

Since the kernel accesses neighboring pixels, boundary conditions must be addressed. Common strategies include:

- **Zero padding**: Setting out-of-bounds pixels to zero.
- **Replication padding**: Extending the edge pixels.
- **Reflection padding**: Mirroring the pixel values at the boundary.

Types of Kernel Filters

Some examples of filters (with size 3×3) are:

Edge Detection

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

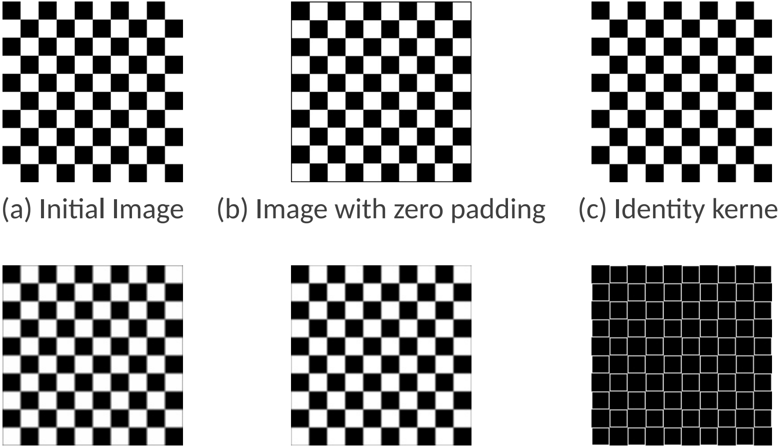
Gaussian Blur

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Box Blur

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Types of Kernel Filters: Examples



(a) Initial Image

(b) Image with zero padding

(c) Identity kernel

(d) Blur kernel

(e) Gaussian kernel

(f) Edge kernel

Computational Complexity

For an image of size $H \times W$ and a kernel of size $k \times k$, the computational complexity is:

$$O(H \cdot W \cdot k^2)$$

which becomes computationally expensive for large images and large kernels, especially in real-time applications...

Table of Contents

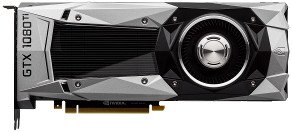
2 Parallelization

- Parallelization
  - Necessity for Parallelization
  - CUDA
  - GPU Data Transfer Overhead

Necessity for Parallelization

As image resolutions increase, the number of pixel operations grows significantly, making real-time processing infeasible on traditional CPU architectures.

Each pixel in the output image can be computed *independently* from the others → **the problem is highly parallelizable**.
This makes kernel-based filtering an ideal candidate for **GPU acceleration**!

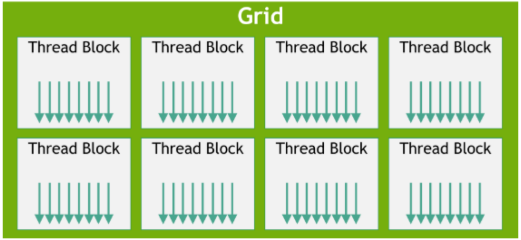


CUDA

2 Parallelization

CUDA (Compute Unified Device Architecture) is a parallel computing platform developed by NVIDIA that allows developers to leverage GPUs for general-purpose computations.

- It enables fine-grained control over thread execution, shared memory usage, and data transfers
- It allows to create hierarchical structure for maximum parallelization: **SM, Grid, Block of Threads** and **Threads**
- Data parallelism → SIMD and SIMT



GPU Data Transfer Overhead

The steps for executing a CUDA kernel typically involve:

- Transferring the input image from CPU to GPU (host to device).
- Executing the kernel (parallel computation on GPU).
- Retrieving the processed image from GPU to CPU (device to host).

The data transfer between the host and device occurs over the PCIe (Peripheral Component Interconnect Express) bus, which has a much **lower bandwidth** compared to the GPU's internal memory.

The cost of these memory transfers can significantly impact overall performance!

UNIVERSITÀ DEGLI STUDI FIRENZE

Da un secolo, oltre.

Table of Contents

3 Algorithms Implementation

- Algorithms Implementation
  - Image Struct
  - Sequential
  - Parallel Algorithms
  - $1channel_{noConst}$
  - `__constant__` memory
  - $1channel$
  - $3channel_{grid}$
  - $3channel$
  - $3channel_3$

Image Struct

- **Structure of arrays** (SoA) allows to reach better performances for the sequential and paralllel code
- `short int` type is chosen to optimize both sequential (vectorization) and parallel code (size of memory moved between device and host reduced). The type is ok because we know that the value of a pixel can be a number in [0, 256]

```
struct Image {
    long long int rows;
    long long int cols;

    short int* channels[3];
}
```

Sequential

- First the new image is initialized
- Then the loop calculates the new values

```
Image* processImage(Image* inputImage, Kernel* kernel) {
    int newRows = inputImage->rows - 2 * kernel->paddingSize;
    int newCols = inputImage->cols - 2 * kernel->paddingSize;
    Image* processedImage = initColoredImage(newRows, newCols);

    int kernelSize = 2 * kernel->paddingSize + 1;
    float* kernelValues = kernel->values;

    // loop (see next slide)

    return processedImage;
}
```

Sequential: Main Loop

3 Algorithms Implementation

The computational complexity can be seen in the main loop → for each channel 4 level nested loop!

```
for (int channelIndex = 0; channelIndex < 3; channelIndex++) {
    short int* inputChannel = inputImage->channels[channelIndex];
    short int* outputChannel = processedImage->channels[channelIndex];

    for (int i = kernel->paddingSize; i < inputImage->rows - kernel->paddingSize; ++i) {
        for (int j = kernel->paddingSize; j < inputImage->cols - kernel->paddingSize; ++j) {
            float sum = 0.0f;
            int outIndex = (i - kernel->paddingSize) * newCols + (j - kernel->paddingSize);

            for (int k = 0; k < kernelSize; ++k) {
                for (int l = 0; l < kernelSize; ++l) {
                    int imgIndex = (i + k - kernel->paddingSize) * inputImage->cols + (j + l - kernel->paddingSize);
                    int kernelIndex = k * kernelSize + l;
                    sum += kernelValues[kernelIndex] * inputChannel[imgIndex];
                }
            }

            outputChannel[outIndex] = std::max(0.0f, std::min(255.0f, sum));
        }
    }
}
```

Parallel Algorithms

3 Algorithms Implementation

Multiple parallel implementations have been developed:

- $1channel_{noConst}$: parallel implementation for a single channel $\rightarrow$ multiple channel can be concatenated;
- $1channel$: same as $1channel_{noConst}$ but `__constant__` is used;
- $3channel$: 3 channel parallelization $\rightarrow$ reduced data movement between host and device;
- $3channel_{grid}$: different data organization;
- $3channel_3$: 3 channel based with 3 arrays as input;

$1channel_{noConst}$: Main function

```
...
int pixelsPerChannel = resultImageRows * resultImageCols;
int threadsPerBlock = 1024;
int numBlocks = (pixelsPerChannel + threadsPerBlock - 1) / threadsPerBlock;
...
for (int channelIndex = 0; channelIndex < 3; channelIndex++){
        cudaMemcpy(d_image, h_image->channels[channelIndex], rows*cols*sizeof(short int),
              cudaMemcpyHostToDevice);
        kernelProcessing<<<numBlocks, threadsPerBlock>>>(d_image, d_kernel, d_resultImage, resultImageCols,
              paddingSize, cols, kernelCols, pixelsPerChannel);
        cudaMemcpy(h_resultImage->channels[channelIndex], d_resultImage, h_resultImage->cols*h_resultImage->
              rows*sizeof(short int), cudaMemcpyDeviceToHost);
    }
...
```

$1channel_{noConst}$: Kernel function

```
__global__ void kernelProcessing_oneChannel(short int* inputImage, float* kernel, short int* resultImage, int resultCols, int paddingSize, int inputImageCols, int
        kernelCols, int N) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    int i = index / resultCols;
    int j = index % resultCols;

    if (index < N){
        float sum = 0.0f;
        for (int k = −paddingSize; k <= paddingSize; ++k) {
            for (int l = −paddingSize; l <= paddingSize; ++l) {
                int kernelIndex = (k+paddingSize) * kernelCols + (l+paddingSize);
                int imageIndex = (i+paddingSize+k) * inputImageCols + (j+paddingSize+l);
                sum += kernel[kernelIndex] * inputImage[imageIndex];
            }
        }
        if (sum < 0) {
            sum = 0;
        }
        if (sum > 255) {
            sum = 255;
        }
        resultImage[i * resultCols + j] = sum;
    }
}
```

$1channel_{noConst}$

Idea:

- The method implements a parallelization for a single channel
- Then the total image is built by concatenate the results of each channel

Advantages:

- It uses the standard data structure (`struct image`) of the sequential algorithm
- Simple implementation

Disadvantages:

- Repeated moving of data between host and device;
- The memory organization of the GPU isn't used for maximum performance → ”constant” variable (like kernel values and array sizes) are copied between the threads;

`__constant__`

- Special, read-only memory region accessible by the GPU kernel;
- Declared statically then used with `cudaMemcpyToSymbol()`;
- Efficiently provide identical data to multiple threads within a warp;
- `__constant__` variables can be broadcasted to all the threads in a warp (32) simultaneously;
- It utilizes a dedicated constant cache (separate from L1/L2) optimized for this broadcast pattern;

1channel

Idea:

Use `__constant__` for the variables shared between the threads that are ”read only”

Advantages:

- **Reduced memory bandwidth usage**: broadcasting a single value to an entire warp requires substantially less bandwidth than potentially 32 separate reads from global memory;
- **Dedicated cache**: using the constant cache avoids polluting the L1 or L2 caches

Disadvantages:

- **Limited size**: the total size of memory is 64KB;
- **Read only memory**: data cannot be modified;

1*channel*: Static declaration

```
__constant__ float d_kernel_const[4096];
__constant__ int const_resultCols;
__constant__ int const_resultRows;
__constant__ int const_paddingSize;
__constant__ int const_inputImageCols;
__constant__ int const_kernelCols;
__constant__ int const_N;
__constant__ int const_channelN;
__constant__ int const_channelNInput;
```

1*channel*: Main function

```
Image* processImage_CUDA_oneChannel_constant(Image* h_image, Kernel* h_kernel) {
    // ...

    cudaMalloc(&d_resultImage, (size_t)h_resultImage->rows*h_resultImage->cols*sizeof(short int));
    cudaMalloc(&d_image, (size_t)h_image->rows*h_image->cols*sizeof(short int));

    cudaMemcpyToSymbol(d_kernel_const, h_kernel->values, h_kernel->rows * h_kernel->cols * sizeof(float));

    for (int channelIndex = 0; channelIndex < 3; channelIndex++){
        cudaMemcpy(d_image, h_image->channels[channelIndex], h_image->rows*h_image->cols*sizeof(short int), cudaMemcpyHostToDevice);
        kernelProcessing_constant_noGrid<<<numBlocks, threadsPerBlock>>>(d_image, d_resultImage);
        cudaMemcpy(h_resultImage->channels[channelIndex], d_resultImage, h_resultImage->cols*h_resultImage->rows*sizeof(short int),
                cudaMemcpyDeviceToHost);
    }

    cudaFree(d_image);
    cudaFree(d_resultImage);

    return h_resultImage;
}
```

1*channel*: Kernel function

```
__global__ void kernelProcessing_constant_noGrid(short int* inputImage, short int* resultImage) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    int i = index / const_resultCols;
    int j = index % const_resultCols;
    int channelIndex = blockIdx.z;

    if (index < const_N){
        float sum = 0.0f;
        for (int k = -const_paddingSize; k <= const_paddingSize; ++k) {
            for (int l = -const_paddingSize; l <= const_paddingSize; ++l) {
                int kernelIndex = (k+const_paddingSize) * const_kernelCols + (l+const_paddingSize);
                int imageIndex = channelIndex * const_channelNInput + (i+const_paddingSize+k) * const_inputImageCols + (j+const_paddingSize+l);
                sum += d_kernel_const[kernelIndex] * inputImage[imageIndex];
            }
        }
        if (sum < 0) {
            sum = 0;
        }
        if (sum > 255) {
            sum = 255;
        }
        resultImage[channelIndex * const_channelN + i * const_resultCols + j] = sum;
    }
}
```

$3channel_{grid}$

Idea:

Maximum parallelization and minimum number of data movements between the host and device memory

Advantages:

- The number of data movements between host and device is reduced → less time consumed in moving data;
- Optimized parallel execution: sequential → parallel → sequential;
- Thread organization reflects the data organization → `Dim3` data type used;

Disadvantages:

- The data structure need to be transformed → linearization and delinearize functions;
- `struct image` cannot be used directly;

$3channel_{grid}$: Support functions

```
short int* linearizeImage(Image* image) {
    long long int channelSize = image->rows * image->cols;
    long long int N = channelSize * 3;
    short int* pixels = new short int[N];

    for (long long int channelIndex = 0; channelIndex < 3; channelIndex++) {
        for (long long int i = 0; i < image->rows; i++) {
            for (long long int j = 0; j < image->cols; j++) {
                pixels[channelIndex*channelSize + i*(long long int)image->cols + j] = image->channels[channelIndex][i*image->cols + j];
            }
        }
    }

    return pixels;
}

void delinearizeImage(short int* pixels, Image* image) {
    long long int channelSize = image->rows * image->cols;

    for (long long int channelIndex = 0; channelIndex < 3; channelIndex++) {
        for (long long int i = 0; i < image->rows; i++) {
            for (long int j = 0; j < image->cols; j++) {
                image->channels[channelIndex][i*image->cols + j] = pixels[channelIndex*channelSize + i*image->cols + j];
            }
        }
    }
}
```

$3channel_{grid}$: Main function

```
Image* processImage_CUDA_threeChannel_constant(Image* h_image, Kernel* h_kernel) {
    Image* h_resultImage = initColoredImage(h_image->rows - 2 * h_kernel->paddingSize, h_image->cols - 2 * h_kernel->paddingSize);
    long long int totalResultPixels = h_resultImage->rows * h_resultImage->cols * (long long int)3;

    short int* h_resultImagePixels = new short int[totalResultPixels];
    short int* h_imagePixels = linearizeImage(h_image);

    dim3 threadsPerBlock(32, 32);
    dim3 numBlocks((h_resultImage->rows + threadsPerBlock.x - 1) / threadsPerBlock.x,
                (h_resultImage->cols + threadsPerBlock.y - 1) / threadsPerBlock.y,
                3);

    short int* d_resultImagePixels;
    short int* d_imagePixels;

    long long int totalInputPixels = h_image->rows * h_image->cols * 3;

    cudaMalloc(&d_resultImagePixels, (size_t)totalResultPixels*sizeof(short int));
    cudaMalloc(&d_imagePixels, (size_t)totalInputPixels*sizeof(short int));

    cudaMemcpyToSymbol(d_kernel_const, h_kernel->values, h_kernel->rows * h_kernel->cols * sizeof(float));
    // more cudaMemcpyToSymbol...

    ...
```

$3channel_{grid}$: Main function

```
    ...

    int channelN = h_resultImage->rows * h_resultImage->cols;
    cudaMemcpyToSymbol(const_channelN, &channelN, sizeof(int));
    int channelNInput = h_image->rows * h_image->cols;
    cudaMemcpyToSymbol(const_channelNInput, &channelNInput, sizeof(int));

    cudaMemcpy(d_imagePixels, h_imagePixels, totalInputPixels*sizeof(short int), cudaMemcpyHostToDevice);

    kernelProcessing_threeChannel_constant<<<numBlocks, threadsPerBlock>>>(d_imagePixels, d_resultImagePixels);

    cudaMemcpy(h_resultImagePixels, d_resultImagePixels, totalResultPixels*sizeof(short int), cudaMemcpyDeviceToHost);

    delinearizeImage(h_resultImagePixels, h_resultImage);

    delete[] h_resultImagePixels;
    delete[] h_imagePixels;

    cudaFree(d_imagePixels);
    cudaFree(d_resultImagePixels);

    return h_resultImage;
}
```

$3channel_{grid}$: Kernel function

```
__global__ void kernelProcessing_threeChannel_constant(short int* inputImage, short int* resultImage) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int channelIndex = blockIdx.z;

    if (i < const_resultRows && j < const_resultCols){
        float sum = 0.0f;
        for (int k = -const_paddingSize; k <= const_paddingSize; ++k) {
            for (int l = -const_paddingSize; l <= const_paddingSize; ++l) {
                int kernelIndex = (k+const_paddingSize) * const_kernelCols + (l+const_paddingSize);
                int imageIndex = channelIndex * const_channelNInput + (i+const_paddingSize+k) * const_inputImageCols + (j+const_paddingSize+l);
                sum += d_kernel_const[kernelIndex] * inputImage[imageIndex];
            }
        }
        if (sum < 0) {
            sum = 0;
        }
        if (sum > 255) {
            sum = 255;
        }
        resultImage[channelIndex * const_channelN + i * const_resultCols + j] = sum;
    }
}
```

3channel

Idea:

A different organization of the distribution of the data to the threads to see the impacts in performances

Instead of calculate the index using multidimensional indexes a 2 dimension coordinate (one for the pixel and one for the channel) is used:

```
int threadsPerBlock = 1024;
dim3 numBlocks((h_resultImage->cols * h_resultImage->rows + threadsPerBlock - 1) / threadsPerBlock, 1, 3);

// instead of

dim3 threadsPerBlock(32, 32);
dim3 numBlocks((h_resultImage->rows + threadsPerBlock.x - 1) / threadsPerBlock.x,
               (h_resultImage->cols + threadsPerBlock.y - 1) / threadsPerBlock.y,
               3);
```

The core difference lies in the shape of the thread blocks and the shape of the grid of blocks, which affects how threads are indexed and how they might map to the image data.

3channel: Kernel function

- A modulo operation is needed to calculate the right indexes;
- Image is seen as a linear array;

```
__global__ void kernelProcessing_constant_noGrid(short int* inputImage, short int* resultImage) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    int i = index / const_resultCols;
    int j = index % const_resultCols;
    int channelIndex = blockIdx.z;

    // main calculation loop
}
```

$3channel$: Consideration

Advantages of thread organization:

- $3channel$: for image stored in linear memory can lead to better performances;
- $3channel_{grid}$: Directly maps the grid/block structure to the 2D image layout. Cleaner than the previous;
- $3channel_{grid}$: can be better for images due to locality;

Disadvantages of thread organization:

- $3channel$: no natural mapping between the image structure and the threads structure in the GPU;
- $3channel$: it requires explicit calculation of the indexes;
- $3channel_{grid}$: a good thread block number need to be chosen → more experimentation is required;

$3channel_3$

Idea:

Instead of transform the image into a linear array pass the channels as argument to the kernel function.

```
// main function
...
int numThreads = h_resultImage->rows * h_resultImage->cols;
int threadsPerBlock = 1024;
int numBlocks = (numThreads + threadsPerBlock - 1) / threadsPerBlock;
// call to the kernel function
kernelProcessing_threeChannelTogether<<<numBlocks, threadsPerBlock>>>(
                  d_imagePixels_channel0,
                  d_imagePixels_channel1,
                  d_imagePixels_channel2,
                  d_resultImagePixels_channel0,
                  d_resultImagePixels_channel1,
                  d_resultImagePixels_channel2
               );
...
```

$3channel_3$: Kernel function

```
__global__ void kernelProcessing_threeChannelTogether(short int* inputImage0, short int* inputImage1, short int* inputImage2, short int* resultImage0, short int* resultImage1, short int* resultImage2) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    int i = index / const_resultCols;
    int j = index % const_resultCols;

    if (index < const_N){
        float sum0 = 0.0f, sum1 = 0.0f, sum2 = 0.0f;
        for (int k = -const_paddingSize; k <= const_paddingSize; ++k) {
            for (int l = -const_paddingSize; l <= const_paddingSize; ++l) {
                int kernelIndex = (k+const_paddingSize) * const_kernelCols + (l+const_paddingSize);
                int imageIndex = (i+const_paddingSize+k) * const_inputImageCols + (j+const_paddingSize+l);
                sum0 += d_kernel_const[kernelIndex] * inputImage0[imageIndex];
                sum1 += d_kernel_const[kernelIndex] * inputImage1[imageIndex];
                sum2 += d_kernel_const[kernelIndex] * inputImage2[imageIndex];
            }
        }
        resultImage0[i * const_resultCols + j] = min(max(sum0, 0.0f), 255.0f);
        resultImage1[i * const_resultCols + j] = min(max(sum1, 0.0f), 255.0f);
        resultImage2[i * const_resultCols + j] = min(max(sum2, 0.0f), 255.0f);
    }
}
```

$3channel_3$: Considerations

Advantages:

- No transformation is needed $\rightarrow$ 0 time wasted in this operation;
- Each thread calculate the value given by the indexes for each channel;

Disadvantages:

- The generalization can be a problem (imagine update the method to 6 channel images...);
- Process three different channel in the same thread can lead to bad cache optimization;

Table of Contents

▶ Experimentation
Setup
System Overview
Experimentation Details

Setup

Each implementation has been compared to the sequential code performance in two tests:

- Fixed image size and kernel size increasing
- Fixed kernel size and image size increasing

The two tests are a sort of **weak scaling**.

Experimentation configuration:

- Average times are collected to have a more reliable measure;
- The execution time doesn't include the image creation;
- The execution time include linearization and de-linearization of the images;
- To reduce randomness the images have been processed only with `blur` kernel;

System Overview

The experimentation has been conducted in:

- Dualsocket Intel Xeon Silver 4314 system with 32 physical cores;
- **NVIDIA RTX A2000 12GB**

Specification	Details
Memory Interface	192-bit
Memory Bandwidth	288 GB/s
CUDA Cores (Ampere)	3,328
Tensor Cores (3rd Gen)	104
RT Cores (2nd Gen)	26
Single-Precision Performance	8.0 TFLOPS
RT Core Performance	15.6 TFLOPS
Tensor Performance	63.9 TFLOPS
Power Consumption	70 W (Total Board Power)

Table: Some infos about NVIDIA RTX A2000 12GB GPU used for the experimentation

Experiments Details

Fixed Image Experiment

- Image sized fixed to a resolution of $1000x1000$ ($1'000'000$ values);
- Kernel size values: $(3, 9, 17, 33, 63, 121)$;
- CUDA implementations times are the result of an average over 1000 executions;
- The average time for Seq is given from an average of $1000, 100, 50, 10, 5, 2$ executions respectively;

Fixed Kernel Experiment

- Kernel is fixed to a resolution of 11×11 (121 values);
- CUDA implementations times are the result of an average over $1000, 1000, 1000, 1000, 1000, 100$ executions respectively;
- The average time for Seq is given from an average of $1000, 100, 50, 10, 5, 2$ executions respectively;

Table of Contents

5 Results

- ▶ Results
  - Fixed Image
  - Fixed Kernel

Fixed Image: Results Table

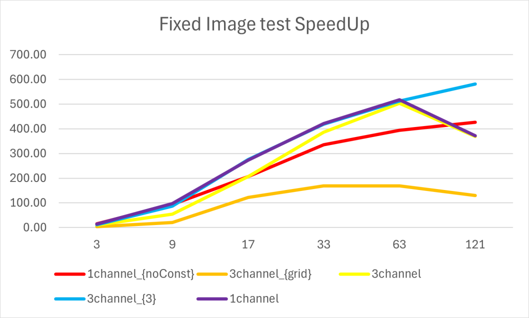
5 Results

Average Times [ms]						
Kernel Size	$1channel_{noConst}$	$3channel_{grid}$	$3channel$	$3channel_3$	$1channel$	Seq
3	1.89	8.73	2.83	2.48	2.13	28.88
9	2.48	11.22	4.27	2.66	2.38	232.30
17	4.87	8.27	4.86	3.65	3.67	1004.94
33	11.76	23.23	10.19	9.42	9.36	3940.20
63	39.07	90.89	30.55	30.00	29.71	15379.80
121	142.25	467.18	164.62	104.56	163.21	60764.00
Speedup						
3	15.25	3.31	10.21	11.63	13.57	-
9	93.86	20.71	54.45	87.21	97.48	-
17	206.46	121.53	206.63	275.17	273.48	-
33	334.99	169.63	386.60	418.24	420.88	-
63	393.62	169.21	503.37	512.63	517.71	-
121	427.17	130.07	369.11	581.16	372.30	-

Fixed Image: Considerations (1)

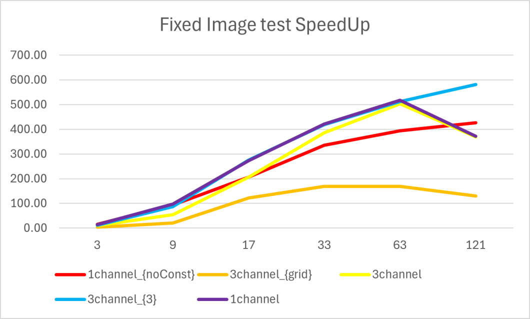
5 Results

- The overall speedup increases with the kernel size → remember k^2 in the computational complexity formula;
- `__constant__` memory gives a good performance improvement → experimentation confirms the theory;
- $3channel_{grid}$ gives the worst performances → no data locality? Data contention?;



Fixed Image: Considerations (2)

- $3channel$ gives good performance (near the best) → best generalization but performances capped by the sequential transformation;
- $3channel_3$ and $1channel$ methods reach the best performance → no (sequential) transformation is needed;
- Linearization and de-linearization can be optimized to run in parallel;



Fixed Kernel: Results Table

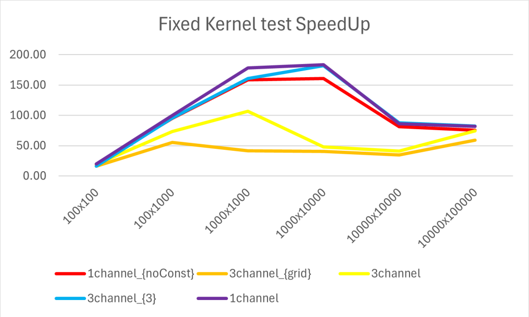
5 Results

Average Times [ms]						
Image Size	$1channel_{noConst}$	$3channel_{grid}$	$3channel$	$3channel_3$	$1channel$	Seq
100x100	0.21	0.25	0.21	0.25	0.20	4.00
100x1000	0.47	0.80	0.60	0.46	0.45	44.45
1000x1000	2.91	11.10	4.34	2.88	2.60	462.52
1000x10000	26.88	106.37	90.43	23.71	23.56	4317.20
10000x10000	521.08	1225.91	1027.72	481.76	489.77	42243.00
10000x100000	5828.72	7413.29	5904.32	5323.00	5354.60	438437.00
Speedup						
100x100	19.41	16.13	19.12	16.13	20.07	-
100x1000	95.11	55.32	73.53	95.69	99.65	-
1000x1000	158.81	41.68	106.67	160.50	178.11	-
1000x10000	160.60	40.59	47.74	182.11	183.26	-
10000x10000	81.07	34.46	41.10	87.68	86.25	-
10000x100000	75.22	59.14	74.26	82.37	81.88	-

Fixed Kernel: Considerations (1)

5 Results

- In a first phase the speedup increases with the image size than we reach a limit for almost all the methods → memory bandwidth becomes more and more important for the performances;
- Methods without mechanism of linearization and de-linearization have better results than the others → image becomes bigger and at the same time the impact of the sequential code on the performances increase;



Fixed Kernel: Considerations (2)

- $1channel$ vs $1channel_{noConst}$ confirms the results of the previous experiment;
- $3channel_{grid}$ worst speedup $\rightarrow$ due combination of not optimal data organization in memory and sequential transformations;
- $1channel$ and $3channel_3$ similar (good) performances $\rightarrow$ no big differences among the methods... The most impactful thing is the memory transfers and sequential code!

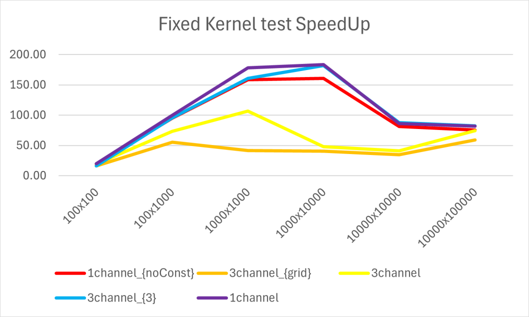


Table of Contents

6 Conclusions

▶ Conclusions

- CUDA provides very good speedup;
- The use of `__constant__` memory gives a visible speedup;
- Sequential transformations can be optimized to a better scaling;
- For increasing image size the performances are limited;

Future developments:

- Test the code with different thread block size;
- Test the code with parallel transformation instead of sequential;
- Use a system clean from other computations that might interfere with the computation time measurement;

Kernel Image Processing with CUDA

Thank you for listening!
Any questions?