HEIG-VD | REDS Institut Reconfigurable and Embedded Digital Systems

Calcul Numérique et accélération Matérielle (CNM)

Prof Marina Zapater
Assistants : Mehdi Akeddar

Stage 2 – Matrix multiplication optimization

lab01 (10.10.2024)

Objectives of the laboratory

The objectives of Lab01 are to exercise the content regarding acceleration using CPU and SIMD. Lab01 is structured in three sessions. This manual refers to "Session 2" and is devoted to optimizing the performance of matrix multiplication by taking into consideration the cache memory hierarchy.

Laboratory validation and grading

- We will grade this laboratory based on the work we ask to be delivered.
  The parts in *light blue* are what you are expected to reply in the readme file.
- We expect (at least) three different commits to the git repository, one per session, clearly indicating the session submission (for example: git commit -m "submission session 2 lab01")
- This last commit, with all work for all sessions completed, should be done at the latest the night before the next lab (lab2 starts). This means by October 30th 2024 at 23h59.

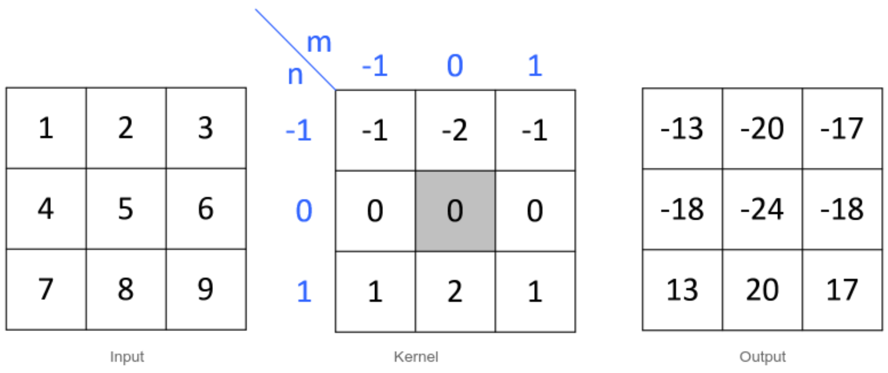
Stage 1 – Matrix convolution and edge detection

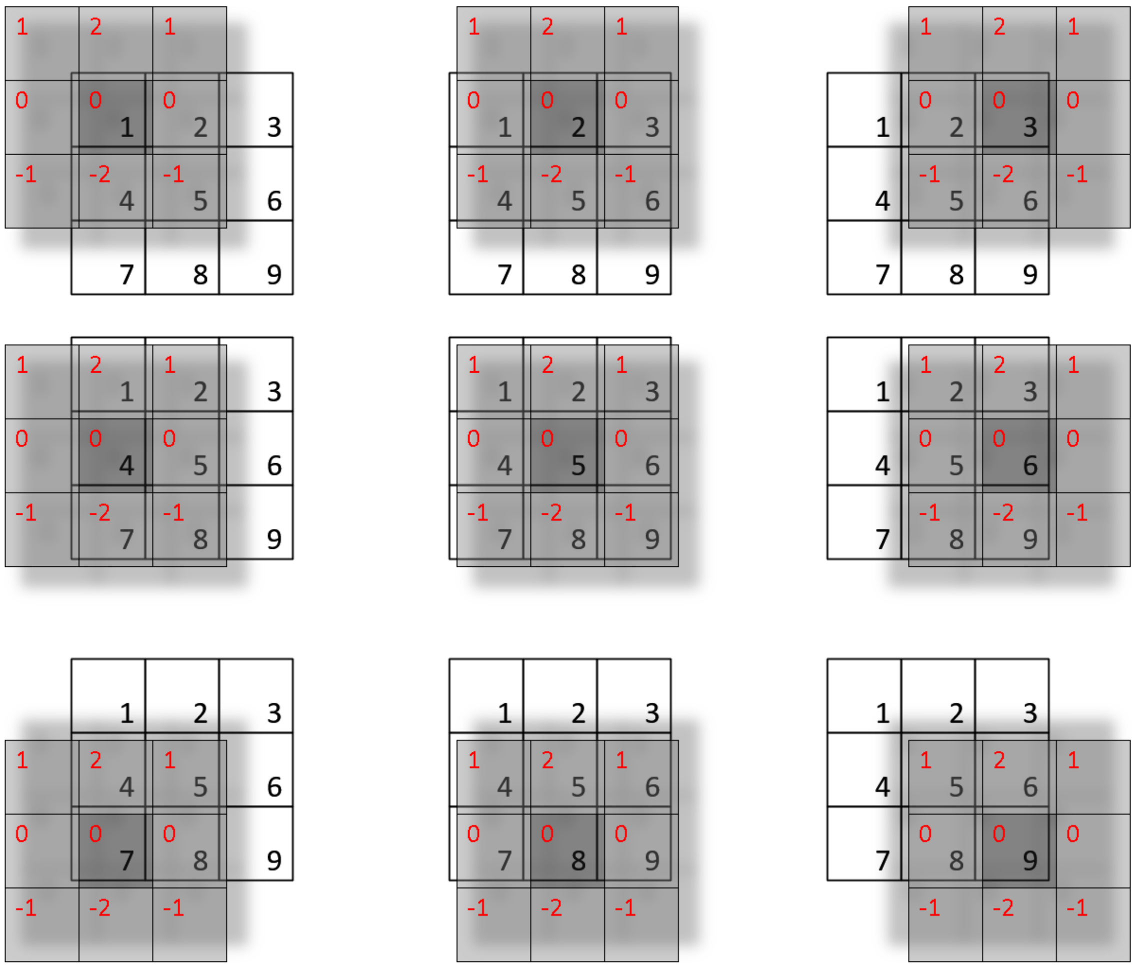
Matrix convolution is one of the most used operations in image processing and deep learning. Convolution is the process of adding each element of the matrix to its local neighbours weighted by the kernel matrix.

The general expression of matrix convolution is

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x-dx, y-dy),$$

Where g is the result, f is the original matrix and ω is the kernel.





In matrix convolution with m × n kernel, each sample requires m × n multiplications and accumulations. Thus, it is a very expensive operation.

Matrix convolution can be used for image processing. Doing convolution between an image and a kernel you can blur the image, sharp it, detect its edges and more. It is also used as the main component of the convolutional layer in convolutional neural networks.

Stage 2 – Edge detection and compilation flags

We provide the source code that implements image edge detection using the Sobel operation. It uses convolution with two 3 × 3 kernels to approximate the gradient of the image intensity (vertical and horizontal). Then, it combines them to obtain the gradient magnitude.

We are using the OpenCV C++ API to read images in different formats. We have generated a *Makefile* to ease the compilation with the right dependencies.

Compile the source code using the following command.

```
make
```

Inside the *Makefile* you can observe the compilation flags. One relevant flag is the optimization level (in this case -O0 that disables most optimizations). Check out the official documentation to learn more about optimizations.

The generated program accepts one argument, an image file.

```
# image edge detection

./edge_detection image_file
```

The program will output in the terminal the image size (rows and columns) and the convolution duration (in microseconds). It also writes the result images in the same folder as the input with the same name as the original but with a different suffix (*x* and *y* for partial gradient results and *edges* for result)

Stage 3 – Measuring edge detection

We will measure the performance of the naïve convolution implementation provided in the source code. We need to run the program with different images with different size.

We ask you to measure the time it takes to perform the convolution. Call the generated program with the images provided.

Stage 4 – Implement edge detection with loop unrolling

We have implemented edge detection using a very simple convolution method. We iterate over each element in the original image and for each element, we iterate over each element in the kernel. Knowing the size of the kernel beforehand, we can use **loop unrolling** to improve the convolution method.

Loop unrolling is an optimization technique used to improve the performance of loops, particularly in cases where the loop's body is relatively simple and can be executed multiple times without relying on the previous iteration's results. It works by reducing loop control overhead (control and branching instructions) and leveraging the pipelining capabilities of modern processors.

The basic idea of loop unrolling is to execute multiple iterations of a loop in a single iteration. Instead of running the loop for every individual value, the loop is executed for a group of values, effectively reducing the number of loop control instructions, and improving the instruction-level parallelism. This can result in improved performance, especially on processors with deep pipelines.

You can find more information and examples on loop unrolling on the additional slides material in pdf for today’s session.

We ask you to improve the convolution method using loop unrolling. Unroll the loops that iterates over the kernel.

Stage 5 – Measuring edge detection with loop unrolling

Once you have implemented edge detection with loop unrolling, we will measure its performance.

We ask you to measure the time it takes to perform the convolution with loop unrolling. Call the generated program with the same images used in the previous measurement stage. Analyze the result comparing it with the implementation without loop unrolling.

Loop unrolling is one of the most typical optimizations performed by the compiler. You can trigger loop unrolling by using the flag `-funroll-loops`, either over the non-optimized code (`-O0` optimization level, or over optimized code (`-O1` or -O2` optimization levels).

Change the makefile provided (file Makefile) to add the loop unrolling optimization, and compare the performance with respect to your loop unrolling technique.

We ask you to measure the time it takes to perform the convolution by performing loop unrolling directly via the compiler, by adding the `-funroll-loops` on the makefile for -O0, -O1 and -O2 optimization levels.

Before you leave…

Make sure you have all the requests *in light blue* of this manual, that you have implemented all the functions required and filled-in the answers in the *README* file in the repo. Then commit all files to your own “cnm24-yourname” repository.

We expect (at least) one commit for this session, clearly indicating the session submission (for example: git commit -m “submission session 2 lab01”)