HEIG-VD | REDS Institut Reconfigurable and Embedded Digital Systems

Calcul Numérique et accélération Matérielle (CNM)

Prof Marina Zapater
Assistants : Mehdi Akeddar

SIMD and NEON intrinsics

lab02 (31.10.2024)

Objectives of the laboratory

The objectives of Lab02 are to exercise the content regarding acceleration using CPU and SIMD. Lab02 only comprises **one manual** and is devoted to optimizing the performance of convolution by using SIMD instructions.

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 to 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 lab02")
- **This commit should be done at the latest the night before the next lab session (07.11.2024) .**
- **Therefore, you have a full week to submit the lab.**

Stage 1 – SIMD and NEON intrinsic

NEON intrinsic types and functions (intrinsics) provide functionality like inline assembly code at a higher abstraction level. Additionally, intrinsics provide features like type checking and automatic register allocation.

NEON vector data types. Vector data types represent vectors of varying type and size that can be used with NEON intrinsics. They have the following format:

<type><size>x<num_lanes>_t

For example:

- **uint8x8_t** is a vector of 8 unsigned 8bits integers.
- **int16x4_t** is a vector of 4 signed 16bits integers.
- **float32x4_t** is a vector of 4 32bits float numbers.

These are some examples of the intrinsics that corresponds with the vectorial instructions described in the slides:

- NEON intrinsics for addition (VADD). Each lane in the result is the consequence of performing the addition on the corresponding lanes in each operand vector. They have the following format:

  vadd[q]_<type>

  For example:
  - **vadd_s16** adds two vectors of 4 signed 16bits integers. It is equivalent to `VADD.I16 D0, D1, D2`
  - **vaddq_s32** adds two vectors of 4 signed 32bits integers. It is equivalent to `VADD.I32 Q0, Q1, Q2`

- NEON intrinsics for multiplication (VMUL). Same as with addition, each lane in the result is the consequence of the multiplication on the corresponding lanes in each operand vector. They have the following format:

  vmul[q]_<type>

  For example:
  - **vmul_s32** multiplies two vectors of 2 signed 32bits integers. It is equivalent to `VMUL.I32 D0, D1, D2`
  - **vmulq_u8** multiplies two vectors of 16 unsigned 8bits integers. It is equivalent to `VMUL.I8 Q0, Q1, Q2`

- NEON intrinsics for loading (VLD) a single vector or lane. They have the following format:

  vld1[q]_<type>

  For example:
  - **vld1_s16** loads 4 signed 16bits integers into an **int16x4_t** vector. It is equivalent to `VLD1.16 {D0}, [r0]`
  - **vld1q_u8** loads16 unsigned 8bits integers into an **uint8x16_t** vector. It is equivalent to `VLD1.8 {D0, D1}, [r0]`

- NEON intrinsics for storing (VST) all the lanes of a vector. They have the following format:

  vst1[q]_<type>

  For example:
  - **vstd1q_s16** stores 8 signed 16bits integers in the memory pointed by an **int16_t** pointer. It is equivalent to `VST1.16 {D0, D1}, [r0]`
  - **vst1q_u32** stores 2 unsigned 32bits integers in the memory pointed by an **uint32_t** pointer. It is equivalent to `VST1.32 {D0}, [r0]`

There are many more intrinsics available that could be useful to solve our problem. Check out the online documentation (NEON Programmers guide and the Neon Intrinsics reference) for useful examples and the full set of available NEON intrinsics.

Stage 2 – Edge detection with intrinsics

Now that we are familiar with the convolution operation and how to use the Sobel filter for edge detection, we will try to optimize it even further using NEON intrinsics. The convolution operation is a good candidate to be solved with SIMD.

We must pay special attention to **data types** and **sizes** (possible overflows). The **number of operations** required and how to **adapt** them to the **intrinsics data types** and **functions**. Online documentation can help us understand how to solve certain problems that might arise like what to do when your data does not fit perfectly into the intrinsic data types (leftovers)

We provide a source code template very similar to the previous session that you must use to implement it.

Compile the source code using the following command.

```
make
```

Inside the *Makefile* you can observe the compilation flags. Check out the documentation to learn more about GCC command line options for ARM processors.

The generated program accepts one argument, an image file.

```
# image edge detection

 ./edge_detection_simd 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)

We ask you to implement the convolution operation used for edge detection using NEON intrinsics. Observe the generated images to see if your implementation is correct.

Stage 3 – Measuring edge detection with NEON intrinsics

We will measure the performance of the convolution implementation with NEON. We need to run the program with different images of different sizes.

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

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 1 lab02”)

This commit should be done at the latest the day before the next lab session at 23:59h.