Calcul Numérique et accélération Matérielle (CNM)

Prof Marina Zapater
Assistants : Mehdi Akeddar

Session 1 – Introduction to CUDA

lab04 (28.11.2024)

Objectives of the laboratory

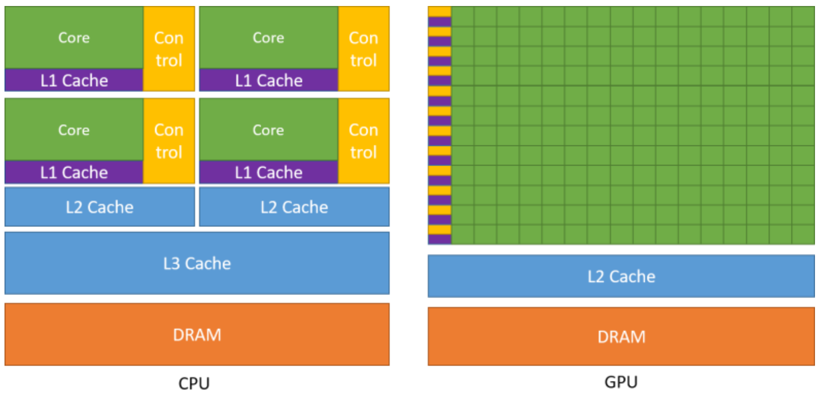
The objectives of Lab04 are to exercise the content regarding acceleration using CUDA. Lab04 is structured in four sessions. This manual refers to “Session 1” and is devoted as an introduction to CUDA programming and understand some basic concepts.

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) four different commits to the git repository, one per session, clearly indicating the session submission (for example: git commit -m “submission session 1 lab02”)
- This commit should be done at the latest the night before the next lab session.
- **Therefore, you have a full week to submit the lab.**

Stage 1 – GPU programming with CUDA

CPU and GPU are designed for different things. A CPU is designed to perform a sequence of operations (thread) as fast as possible and can execute a few of these threads in parallel. A GPU, originally designed to render graphics, is designed to execute thousands of these threads in parallel. CUDA, originally Compute Unified Device Architecture, is a computing architecture and API created by NVIDIA that enables NVIDIA GPUs to serve as platform for general purpose applications.



Stage 2 – CUDA development environment

The CUDA driver and the CUDA toolkit are already installed in our boards. The default installation location for the toolkit is `/usr/local/cuda-<version>`. The symbolic link `/usr/local/cuda` points to the location where the CUDA Toolkit was installed. It is recommended to use the symbolic link because it allows us to use different CUDA Toolkits without any configuration file update.

We need to configure the environment:

- The environment path variable needs to include the CUDA Toolkit binary folder:

```
export PATH=$PATH:/usr/local/cuda/bin
```

- The dynamic libraries path for the linker needs to include the CUDA Toolkit lib folder:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64:/usr/local/cuda/lib
```

- Once the environment is configured, we can use the CUDA Toolkit in our boards. Execute this command to check if the environment is correctly set:

```
nvcc -V
```

- To compile a CUDA C++ application we need to use the NVIDIA `nvcc` compiler driver:

```
nvcc input_file.cu -o output_file
```

This is the simplest command we need to execute to generate a CUDA application. Something to consider is the capabilities of our hardware. NVIDIA has different architectures with different features. We might need to specify the GPU architecture if we want to take advantage of the features in newer architectures.

- First of all, we need to replace the original Nsys installed with our distribution, because it's not suited for Orin boards. We will install the tegra alternative:

```
sudo update-alternatives --install /usr/local/bin/nsys nsys /opt/nvidia/nsight-
systems/2024.2.2/target-linux-tegra-armv8/nsys 0
```

```
sudo update-alternatives --set nsys /opt/nvidia/nsight-systems/2024.2.2/target-
linux-tegra-armv8/nsys
```

- You can check which version of Nsys you have using:

```
readlink -f "$(which nsys)"
```

- An easy way to profile a CUDA application is using the nsys profiling tool, supported by NVIDIA Nsight Systems:

```
 nsys profile --output=your_application_report ./your_application
```

```
nsys stats your_application.nsys-rep
```

- In Visual Studio Code we can add CUDA development and debugging support installing the **Nsight Visual Studio Code Edition.** It will detect when we are working with files containing CUDA source code and it will be enabled by default. If we want to debug inside Visual Studio Code we need to follow the instructions in the documentation of the extension.

To enable code auto completion, we need to add the `include` folder of the CUDA Toolkit (`/usr/local/cuda/include`) to the IntelliSense settings of the Microsoft C/C++ extension for Linux.

Stage 3 – Hardware characteristics and CUDA capabilities

We have provided a CUDA application that uses the CUDA Runtime to query the capabilities of hardware. We will use it to obtain the capabilities of our board and to understand some basic concepts relevant to develop in CUDA.

- To compile it and generate an application we can execute the following:

```
nvcc cuda_device_properties.cu -o cuda_device_properties
```

If we have configured our environment correctly there should not be any error and we can execute the generated application. It prints to the `stdout` a list of relevant capabilities of our board.

We ask you to complete the README.md about the capabilities and answer the questions. You will need to search in the official CUDA documentation.

***Stage 4* – Basic CUDA program and profiling**

We have provided a very simple CUDA application from the official CUDA samples that performs element by element vector addition inside the GPU: cuda_vector_add.cu .

Compile the application and profile it, using nsys command:

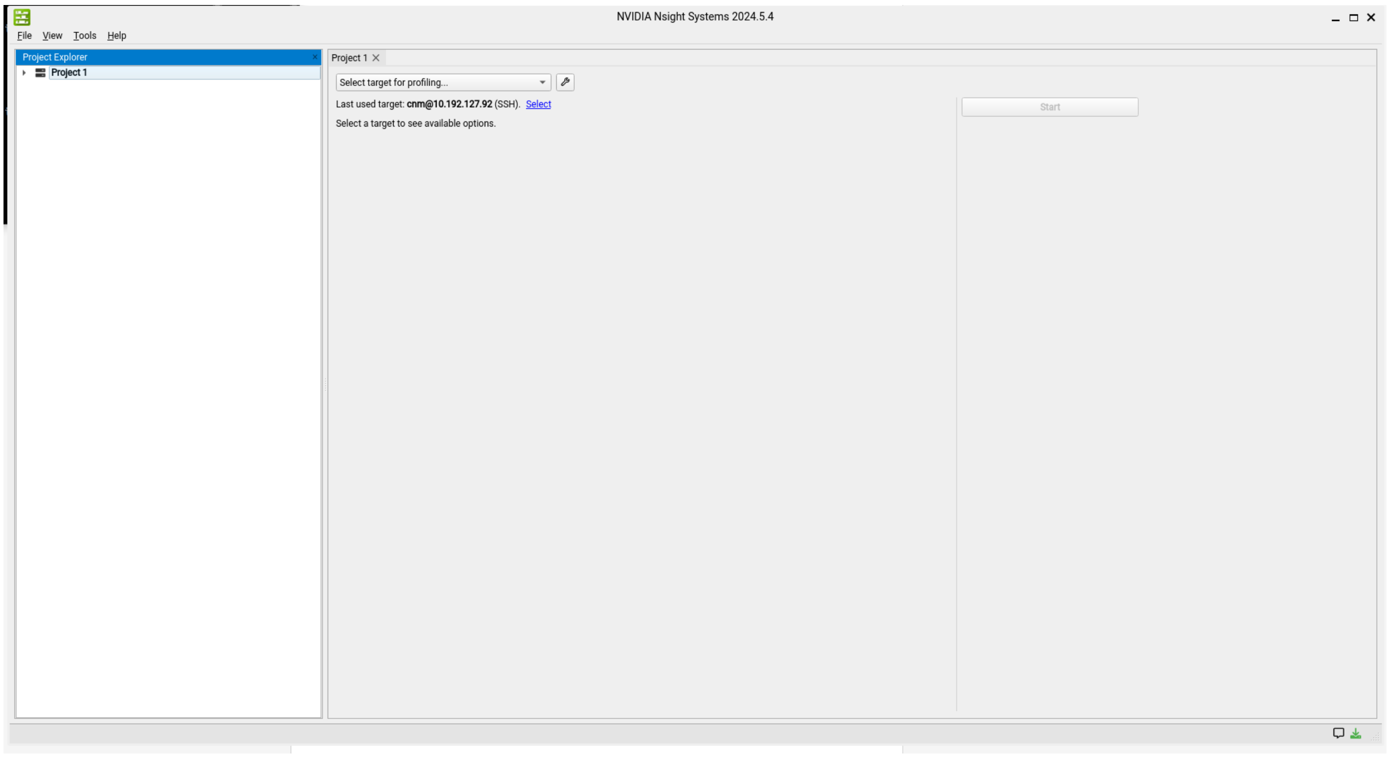
We ask you to answer the questions about the source code and the questions about the profiling of the execution in the README.md

***Stage 5* – Nsys UI**

To get a better view of your profiling, you can use the official Nsight UI to view your program. **On your local machine,** make sure that nsys is installed and run the following command:

```
nsys-ui &
```

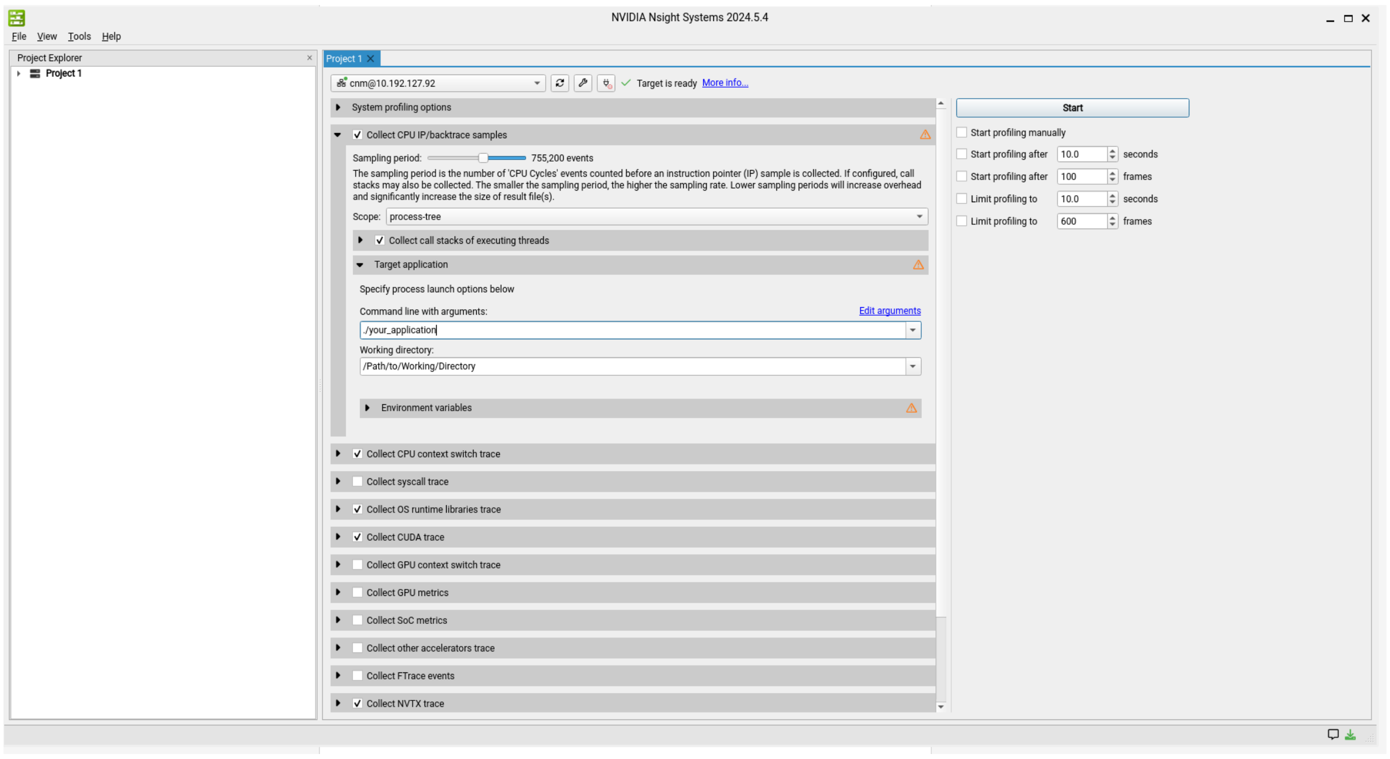
You should see a window like this:



Select target for profiling...
Last used target: cnm@10.192.127.92 (SSH). Select
Select a target to see available options.

Click on the little key, then Create a new connection. Add the IP address of the cnm and the username. Click on connect, you should be asked about the password. From here the connection is set.

The window should look like this:



cnm@10.192.127.92 ✓ Target is ready More info...

- System profiling options
- ✓ Collect CPU IP/backtrace samples
  - Sampling period: 755,200 events
  - The sampling period is the number of 'CPU Cycles' events counted before an instruction pointer (IP) sample is collected. If configured, call stacks may also be collected. The smaller the sampling period, the higher the sampling rate. Lower sampling periods will increase overhead and significantly increase the size of result file(s).
  - Scope: process-tree
  - ✓ Collect call stacks of executing threads
  - Target application
    - Specify process launch options below
    - Command line with arguments: ./your_application
    - Working directory: /Path/to/Working/Directory
    - Environment variables
- ✓ Collect CPU context switch trace
- Collect syscall trace
- ✓ Collect OS runtime libraries trace
- ✓ Collect CUDA trace
- Collect GPU context switch trace
- Collect GPU metrics
- Collect SoC metrics
- Collect other accelerators trace
- Collect FTrace events
- ✓ Collect NVTX trace

Start profiling manually		
Start profiling after	10.0	seconds
Start profiling after	100	frames
Limit profiling to	10.0	seconds
Limit profiling to	600	frames

Now enter the path to your working directory and the command to run your application, then click on Start.

We ask you to provide the timeline of your application with some analysis on the running process.

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 “cnm23-yourname” repository. We expect (at least) one commit for this session, clearly indicating the session submission (for example: git commit -m “submission session 2 lab03”)