Lab assistant: Andreas Axelsson

Contemporary Computer Architecture
Lab 1 Introduction to Jetson Nano

Introduction

The NVIDIA Jetson Nano is a small, powerful computer that enables the development of millions of new small, low-power AI systems. It's suitable for edge AI applications and research purposes. In this lab, we will focus on using the Jetson Nano for high-performance computing, particularly using threads and some vectorized instructions to achieve high performance. Pthreads provide a method to create multi-threaded applications, a key to harnessing the power of multi-core CPUs. The vectorized instruction set (NEON) gives the possibility to run SIMD instructions and thus increase the computation.

Learning Outcomes

By the end of this lab session:

- Understand the basics of Jetson Nano.
- Setup and run a simple Pthreads program on the Jetson Nano.
- Understand the importance of multi-threaded programs in high performance computing.
- See how vectorized code can speed up calculations.
- Looked at generated assembly code and see the impact of compiler optimization settings.

Setup

Below is the setup procedure to prepare a Jetson Nano for the following labs. First lab might take some time due to updates etc.

1. **Hardware Requirements:**
    - NVIDIA Jetson Nano Developer Kit
    - MicroSD Card (16GB minimum recommended)
    - Power supply
    - Monitor, keyboard, and mouse
    - Ethernet cable for internet connection

2. **Software Setup:**
    - Download the Jetson Nano Developer Kit SD Card Image from NVIDIA's official site. (NOTE: Already done!!)
    - Flash the SD card using a tool like Etcher. (NOTE: Already done!!)
    - Insert the SD card into the Jetson Nano, connect the peripherals and power it up.
    - Follow the on-screen instructions to set up the device.
3. **Setting Up Development Environment:**
    - Open terminal and update the package list:

      ```
      sudo apt update
      sudo apt upgrade
      ```

    - Install the required development tools:

      ```
      sudo apt install gcc g++ make
      ```

The update/upgrade can take some time to perform. So you can take a coffee and relax a while!

If you are new to use Linux ask the lab assistant or any of your Linux savvy class mates.

Exercise 1

Below is a small program demonstrating a multi-threaded program. In main it launches 5 threads and then waits until all the threads have exited using *pthread_join*, before main itself returns. As you can see one has to include pthread.h to access the standard POSIX functions to create threads. In other programming languages there are other methods to launch threads. For instance in C++11 you have std::thread, and in C++20 you have the improved std::jthread. Please feel free to look into these if you intend to develop in C++.

For this exercise we use C-code.

Add the code snippet into a file called 'threads.c' using an editor. There is *gedit* installed in the linux distribution you are running right now.

To run your code, start a terminal and run:

```
$ gcc threads.c -o threads -lpthread
```

```
$ ./threads
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Number of threads to create
#define NUM_THREADS 5

// Thread function
void* printHello(void* threadId) {
    long tid = (long)threadId;
    printf("Hello from thread #%ld\n", tid);
    pthread_exit(NULL);
    return NULL;  // This line is typically not reached due to
pthread_exit above.
}

int main() {
    pthread_t threads[NUM_THREADS];
    int rc;
    for(long t = 0; t < NUM_THREADS; t++) {
        printf("Creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, printHello, (void*)t);
        if(rc) {
            printf("Error: Unable to create thread, %d\n", rc);
            exit(-1);
        }
    }

    // Join the threads
    for(long t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }

    printf("Main thread completing\n");
    return 0;
}
```

pthread_create takes as first argument a pointer to a pthread_t object, second argument is thread attribute (unused, thus NULL), third argument is out thread function, and fourth argument is a parameter sent to the thread. This parameter can for instance be used to give each thread a unique id.

Quiz Questions:

1. **Conceptual Understanding**:

   - What is the primary purpose of **pthread_create()**?
     a) To terminate a thread.
     b) To pause the execution of a thread.
     c) To create a new thread.
     d) To join a thread back to the main thread.

2. **Code Interpretation**:
    - How many threads, including the main thread, are running in the program?
      a) 1
      b) 5
      c) 6
      d) 10
3. **Execution Sequence**:
    - Which of the following best describes the order of thread completion in the program?
      a) The main thread always finishes last.
      b) The main thread always finishes first.
      c) The main thread can finish before some threads but not others.
      d) The order is completely unpredictable.
4. **Function Purpose**:
    - What is the role of **pthread_join()** in the code?
      a) It ensures that the main thread waits for the worker threads to finish.
      b) It terminates the worker threads immediately.
      c) It creates new threads.
      d) It prints out messages from the worker threads.
5. **Thread Termination**:
    - In the **printHello** function, what does **pthread_exit(NULL)** do?
      a) Creates a new thread.
      b) Pauses the execution of the current thread.
      c) Terminates the current thread.
      d) Joins the current thread back to the main thread.

Mini exercise: run the following command: $ lscpu

Determine how many cores the cpu has, maximum clock frequency model name, architecture, as well as if it is big or little endian. Last find how much cache memory it has.

Exercise 2

The next exercise is to test a small program that do some calculations, both using standard code, and to test to do the same using vectorized code.
The arm processor has a normal instruction set that uses SISD as per Flynn's taxonomy (Single Instruction, Single Data) run by its instruction execution pipeline. It has 31 registers of 64-bit width, that can be used either at its full width or smaller sizes depending on the

calculations to be done.
Besides these standard instructions, it also has a vectorized instruction set that can use 128-bit wide registers to store data in, and then perform computations on this vector. The vector could as an example hold 4 parallel 32-bit wide floating numbers that can be used multiplying another set of 4 parallel 32-bit floats, thus giving a 4-fold performance boost.

Put the code below in a file called multiply.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <arm_neon.h>

void mult_std(float* a, float* b, float* r, int num)
{
        for (int i = 0; i < num; i++)
        {
                r[i] = a[i] * b[i];
        }
}

void mult_vect(float* a, float* b, float* r, int num)
{
        float32x4_t va, vb, vr;

        for (int i = 0; i < num; i +=4)
        {
                va = vld1q_f32(&a[i]);
                vb = vld1q_f32(&b[i]);

                vr = vmulq_f32(va, vb);

                vst1q_f32(&r[i], vr);
        }
}

int main(int argc, char *argv[]) {

        int num = 100000000;

        float *a = (float*)aligned_alloc(16, num*sizeof(float));
        float *b = (float*)aligned_alloc(16, num*sizeof(float));
        float *r = (float*)aligned_alloc(16, num*sizeof(float));

        for (int i = 0; i < num; i++)
        {
                a[i] = (i % 127)*0.1457f;
                b[i] = (i % 331)*0.1231f;
        }
        struct timespec ts_start;
        struct timespec ts_end_1;
        struct timespec ts_end_2;

        clock_gettime(CLOCK_MONOTONIC, &ts_start);
        mult_std(a, b, r, num);
        clock_gettime(CLOCK_MONOTONIC, &ts_end_1);
        mult_vect(a, b, r, num);
        clock_gettime(CLOCK_MONOTONIC, &ts_end_2);

        double duration_std = (ts_end_1.tv_sec  - ts_start.tv_sec) +
                        (ts_end_1.tv_nsec - ts_start.tv_nsec) * 1e-9;
        double duration_vec = (ts_end_2.tv_sec  - ts_end_1.tv_sec) +
                        (ts_end_2.tv_nsec - ts_end_1.tv_nsec) * 1e-9;

        printf("Elapsed time std: %f\n", duration_std);
        printf("Elapsed time vec: %f\n", duration_vec);

        free(a);
        free(b);
        free(r);

        return 0;
}
```

The code generates two arrays a and b of floats of length 100,000,000 and a result array r of the same length. The purpose of the program is just to calculate *r[i] = a[i]*b[i]* for all indices in the arrays. There are two functions, *mult_std* that uses standard C-code and *mult_vect* that uses an intrinsic function that uses the SIMD instruction inside the NEON vectorized instruction set. The intrinsic functions are just wrappers around the corresponding assembly instructions that makes it easier to use from C or C++.

Using the linux function clock_gettime found in time.h, we can time the two functions and see how they perform relative to each other. Note that clock_gettime fills in a struct containing two variables, tv_sec and tv_nsec, thus the number of seconds, as well as nanoseconds that has elapsed since the clock started.

To calculate the duration of our two function calls we can subtract values from two readings and calculate a floating point value with the number of elapsed seconds.

To compile the code and try it run:

```
$ gcc multiply.c -o multiply
```

```
$ ./multiply
```

NOTE:

Out of the box the compiler generates quite slow code code. By adding a compiler directive -O0, -O1, -O2 and -O3 we can control the level of optimization. The -O0 is the default.

Exercise a):

Generate a table with the elapsed durations for the two functions in the columns and the four different optimization levels in the rows.

How do the different optimization levels compare? Note that our program is quite small and not especially complicated. So if you don't see any dramatic change from one level to the other then "Don't worry – be happy"!

Exercise b):

It is interesting to see what machine code instructions the compiler generates from our C-code. The normal process of generating an executable file from source code is:

1) Compiler compiles the C-code and generates assembly code
2) The assembly code is assembled by the assembler into an object code file
3) The object code file is linked by the linker into the executable

We want to look at the assembly code. But we don't see and file called multiply.s in the file directory. This is because the compiler as default removes any intermediate files and only keeps the executable. To keep the *.s file we can add yet another directive to the gcc line, that is -save-temps after which we will see that the multiply.s is there.

Now do this for optimization level -O1 and -O3 and compare the files using a text editor or doing more in the terminal. Look especially after the labels mult_std: and mult_vect:. Those labels indicate the function entry points of the two functions we have timed.

TODO: Try to at least at a basic level understand the code, and explain it to the lab assistant. Especially the differences between -O1 and -O3 optimization.

To learn about the instruction sets for the CPU inside the Jetson Nano, you can look into the document "Armv8-A Instruction Set Architecture.pdf".

Exercise 3

We tested running multiple threads in Exercise 1, and in Exercise 2 we did some computation that took fairly long time to perform. Could we improve the running time even more? Perhaps if we can combine the two exercises and run the computations on multiple threads.

Create a worker thread function that calls mult_std, but with a subset of the array. Perhaps you can divide the work so that the first thread calculates the first part of the array, the second thread the second part of the array etc using the thread_id to determine which part of the array to do work on. Launch the threads and time how long time it takes to run. Make sure you use the pthread_join before you measure the end time, otherwise some of the worker threads might still running.

How do we pass the information to the worker threads? We used a single thread id as a parameter in the first exercise. Can we do better? What about passing a custom struct containing pointers to the work each thread should do?

Question: How many threads should you optimally launch? Why?

Do the same with a worker thread function calling mult_vect instead.

You can use -O3 as optimizer directive for this exercise. How does this compare with the case using only one core.

Question: Do you get the speed-up you expected? If not, why?

Please demonstrate your code and present your results to the lab assistant.

Lab OK: ____________________