Lab assistant: Andreas Axelsson

Contemporary Computer Architecture
Lab 2 CUDA Getting Started

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 second lab we will start to explore the GPU of the Jetson Nano. Using CUDA (which is an acronym for Compute Unified Device Architecture) we can exploit the parallel power of the many cores found in the Nvidia GPU.

Learning Outcomes

By the end of this lab session:

- Understand the basics of CUDA.
- Setup and run a simple CUDA program on the Jetson Nano.
- Understand the importance of multi-threaded programs in high performance computing.
- See how parallel code can speed up calculations.
- Use profile tools to measure performance.

Setup

The Jetson Nano Developer Kit SD Card Image installed and setup last lab session already contains CUDA packages to compile and run GPU accelerated code. Nevertheless, there are some steps to perform before you can use it.

If you use a terminal and run:

$ nvcc –version

It will tell you *command not found*. We need to ensure your terminal finds the proper commands by altering the PATH environmental variable. To make the change permanent between reboots, the easiest way is to add a few lines to the ".bashrc" file found in your home directoty. NOTE: A file starting with a punctuation "." will not show up if you try to list a directory using the "ls" command as those files are considered to be hidden. By adding the argument -a also hidden files will be listed.

Try: *$ ls -a* in a terminal

Open the ".bashrc" in an editor and add the following lines at the end of it:

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

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

The first of these two lines makes sure that the commands included in the CUDA package can be found when you type it in the terminal (such as 'nvcc --version'), and the second line makes sure that the compiler/linker finds the libraries we use for our CUDA programs we build.

Note that you need to restart the terminal to ensure it re-reads the .bashrc (or run *source .bashrc*).

Now try: $ nvcc --version

We will use the command nvprof later. This command needs to be run as an elevated super user using sudo nvprof. The path you exported in .bashrc is not available when using sudo and thus we have to fix that problem. Below is two solutions:

1. The most straight forward is to use the absolute path when launching nvprof:
   $ sudo /usr/local/cuda/bin/nvprof
2. The second option is to add /usr/local/cuda/bin: in sudoers file.
   a. $ sudo visudo
   b. Add /usr/local/cuda/bin: first in secure_path="…"
   c. Press ctrl+s to save and ctrl+x to exit

Another tool we will use is jtop. It is a python tool and is not installed by default. Do the steps below to install it:

1. Install pip using: $ sudo apt install python-pip
2. Install jtop using: $ sudo -H pip install jetson-stats
3. Run: $ sudo systemctl restart jtop.service
4. Logout and login again to be able to run jtop from a terminal

Now we are ready to start our endeavor into the world of GPU parallel programming.

Exercise 1

Out first steps into the world of parallel programming might not be very fancy or exotic, but to be able to run, we first have to take some baby steps first!

Add the code below into a file called ex1.cu. We use extension cu to know that it is a cuda source file.

```
#include <iostream>
#include <math.h>

__global__ void multKernel(int n, float* a, float* b, float* c)
{
    for (int i = 0; i < n; i++) {
        c[i] = a[i] * b[i];
    }
}

int main() {
    int N = 1<<24;
    float *h_a, *h_b, *h_c;
    float *d_a, *d_b, *d_c;

    // Allocate host memory
    h_a = new float[N];
    h_b = new float[N];
    h_c = new float[N];

    // Allocate device memory
    cudaMalloc(&d_a, N * sizeof(float));
    cudaMalloc(&d_b, N * sizeof(float));
    cudaMalloc(&d_c, N * sizeof(float));

    // Initialize host data
    for (int i = 0; i < N; i++)
    {
        h_a[i] = 2.0f;
        h_b[i] = 3.0f;
    }

    // Copy data from host to device
    cudaMemcpy(d_a, h_a, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, N * sizeof(float), cudaMemcpyHostToDevice);

    // Launch the kernel
    multKernel<<<1, 1>>>(N, d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(h_c, d_c, N * sizeof(float), cudaMemcpyDeviceToHost);

    // Check result for errors (all values should be 6.0f)
```

```
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(h_c[i] - 6.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Clean up
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    delete[] h_a;
    delete[] h_b;
    delete[] h_c;

    return 0;
}
```

To compile it we use the cuda compiler nvcc instead of gcc:

$ nvcc ex1.cu -o ex1

If you run it using $ ./ex1 not much happends (only a small print out “Max error: 0”).

The program just multiply two vectors containing about 16 million values.

Let us examine what happends a bit more using the nvprof tool:

$ sudo nvprof --print-gpu-summary ./ex1

Question:

How much time is spent in running the kernel, copying data from host to device and from device to host?

Not so impressive to spend so long time to multiply 16 million values!!!

Question:

Why is the performance so poor?

Exercise 2

Let's see how we step by step can improve the performance. As you have suspected right now, we have only used one of all the lovely cores to do the calculations. That poor bastard is exhausted from all the work. Sometimes it is fairer to share the work among others.

With a few changes we can speed up things. Copy ex1.cu to file ex2.cu so we can improve. Let us change the kernel launch and increase the block size in the call:

```
// Launch the kernel
multKernel<<<1, 256>>>(N, d_a, d_b, d_c);
```

With this change more threads will be activated and run. Remember that threads are activated in a multiple of 32 (that is a Warp), so keep the block size a multiple of the warp size.

The code will not work as expected, or at least we will not get the speed up.

Change the CUDA kernel as well:

```
__global__ void multKernel(int n, float* a, float* b, float* c)
{
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride) {
        c[i] = a[i] * b[i];
    }
}
```

Now when we have multiple threads that will run we need to figure out which thread is running in each instance. That is where threadIdx.x, and blockDim.x comes into play. blockDim.x contains the size of the block, thus 256 in our case (remember the change in launch of the kernel?), and the threadIdx.x contains the actual thread id, a value between 0 and 255 in our case.

Using these two variables we can calculate a unique index where we can fetch and store data in each thread.

Now compile ex2.cu and run it in the same way using nvprof as before.

$ sudo nvprof --print-gpu-summary ./ex2

Question:

How much time is spent in running the kernel, copying data from host to device and from device to host?

Question:

How much has the performance improved?

Quiz: CUDA Array Multiplication

Basic Understanding:

Given two arrays A and B, both of size N, you launch a kernel with one thread per array element to multiply them. If you launch the kernel with <<<N, 1>>>, what does this configuration imply?

a) There are N blocks and each block has 1 thread.

b) There is 1 block and it has N threads.

c) There are N blocks and each block has N threads.

d) There is 1 block and it has 1 thread.

Thread Indexing:

If you're using a 1D grid and 1D block configuration for the kernel launch and you wish to compute the global index of a thread in the grid, which of the following formulas would you use?

a) threadIdx.x + blockDim.x

b) blockIdx.x + threadIdx.x

c) blockIdx.x * blockDim.x + threadIdx.x

d) blockDim.x / threadIdx.x

Race Conditions:

In a kernel where each thread writes its result to an output array, how can threadIdx be used to avoid race conditions?

a) By using threadIdx.x to ensure each thread writes to a different memory location.

b) By synchronizing all threads using threadIdx.x after every write operation.

c) By using atomic operations based on threadIdx.x.

d) threadIdx.x cannot be used to avoid race conditions.

Memory Access:

If threads within a block access contiguous memory locations in the arrays A and B during multiplication, this is termed as:

a) Scattered memory access.

b) Strided memory access.

c) Coalesced memory access.

d) Randomized memory access.

Shared Memory:

If you want to load elements of arrays A and B into a faster on-chip memory before performing the multiplication, which memory space in CUDA would you use?

a) Global memory

b) Texture memory

c) Shared memory

d) Constant memory

Exercise 3

Sofar, we have only used one block. Let's try to run multiple blocks. Copy ex2.cu to ex3.cu

Modify the kernel launch to:

```
    // Launch the kernel
    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;
    multKernel<<<numBlocks, blockSize>>>(N, d_a, d_b, d_c);
```

and the kernel to:

```
__global__ void multKernel(int n, float* a, float* b, float* c)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride) {
        c[i] = a[i] * b[i];
    }
}
```

$ sudo nvprof --print-gpu-summary ./ex3

Question:

How much time is spent in running the kernel, copying data from host to device and from device to host? Better or worse than before? Why the difference?

Exercise 4

In the quiz before you answered a question regarding memory access. Let us investigate how important this memory access pattern is in terms of performance.

To simulate that we have reordered the data and how it is fetched from memory we will introduce another vector into the kernel called perm. With this vector we can reorder the access order by using it as a lookup table.

Modify the kernel to:

```
__global__ void multKernel(int n, float* a, float* b, float* c, int* perm)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride) {
        int idx = perm[i];
        c[idx] = a[idx] * b[idx];
    }
}
```

In the main add were appropriate:

```
int *h_perm, *d_perm;

h_perm = new int[N];
```

Now we need to fill h_perm with something. If we fill it from 0 to N-1 we will not permeate the order of the memory access at all. Easiest is to add a line in the initialize host data loop:

```
h_perm[i] = i;
```

Copy the h_perm to d_perm in the exact same manner as h_a and h_b is copied.

In the kernel launch, add the perm argument at the end:

```
multKernel<<<numBlocks, blockSize>>>(N, d_a, d_b, d_c, d_perm);
```

If you compile and run it you should get similar results as before.

$ sudo nvprof --print-gpu-summary ./ex4

Here comes the interesting part. Let us shuffle the perm array. That will result that we will read non-consecutive memory addresses for each thread index.

This will result in what is referred to as non-coalesced memory access.

See the shuffle code next page.

After the initialize host data loop (but before copy to device), add another shuffle loop:

```
// Shuffle perm
for (int i = N-1; i > 0; i--)
{
    int j = rand() % (i + 1);
    std::swap(h_perm[i], h_perm[j]);
}
```

The code above just reorders the perm array so that it still contains the numbers from 0 to N-1 but in random order. When we use perm as a lookup-table in our kernel we will then read from memory in a random order.

$ sudo nvprof --print-gpu-summary ./ex4

Question:

How much time is spent in running the kernel, copying data from host to device and from device to host?

Question:

Why is the performance so poor now?

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

Lab OK: _____