Lab 5 - advanced features of CUDA

Files to be used in this lab (download):
- histshared.cu
- simpleMultiGPU.cu
- plm.cu
- single.cu

Lab material is also available on colab

After the last 2 labs, you should have a handle on basic CUDA features. This week we'll introduce a few of the more complex functionalities that you can use in your CUDA programs.

1) Atomics

If several threads need to modify the same location in memory at the same time, it's possible that there may be data races that produce incorrect results. To prevent that, we can use atomic instructions. CUDA provides several functions that can be used for both global and shared memory. The most fundamental one is `atomicAdd`:

```
int atomicAdd(int* address, int val);
```

The first argument is the address of the value that we want to increment, and the second is the increment amount (e.g. `atomicAdd(&var, 1)` increases the value of var by 1). To see all available atomic operation, consult the CUDA programming guide.

Example:

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo) {
        __shared__ unsigned int temp[256];
        temp[threadIdx.x] = 0;
        __syncthreads();

        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        int stride = blockDim.x * gridDim.x;

        while (tid < size) {
                atomicAdd(&temp[buffer[tid]], 1);
                tid += stride;
        }
        __syncthreads();

        atomicAdd(&(histo[threadIdx.x]), temp[threadIdx.x]);
}
```

This example shows a kernel that computes a histogram from a given buffer and utilises shared memory (and `atomicAdd`) to do so. First we atomically increase values in shared memory and then use the intermediate results from each block to atomically increase values in the output buffer.

You can see a full example in `lab5/histogram/histshared.cu`.

2) CUDA streams - asynchronous execution

a) Page-locked host memory

In all of our examples so far, we've allocated host memory with regular `malloc` calls. However, there is another way to do this - the `cudaHostAlloc` function. In contrast to `malloc` (which allocates standard pageable memory), `cudaHostAlloc` allocates page-locked (or pinned) memory. Such memory has one important property - it can never be paged to disk, it always resides in physical RAM. The CUDA driver uses DMA (Direct Memory Access) to transfer data between host and device, so in case of pageable memory it can hit a paged buffer, which needs to be copied twice (first from disk to a page-locked staging buffer and then to the GPU). Conversely, in case of page-locked memory, the copy takes place only once. To free pinned memory, you need to use the `cudaFreeHost` function instead of the standard `free` function.

Keep in mind that page-locked memory is never buffered, so you can quickly run out of physical memory if you're not careful.

You can test the performance benefits of `cudaHostAlloc` in exercise 1.

b) Streams

Page-locked memory plays an important role in CUDA streams. Before describing that in more details, let's first familiarise ourselves with streams themselves.

You may have noticed that we use `cudaEvent_t` structures to measure CUDA execution time. They need to be passed to the `cudaEventRecord` function - the first argument of that function is the event, but the second one has been a mysterious 0 so far. That 0 is the default stream id. Streams are a way of achieving task parallelism in CUDA - they can be used to perform several copy operations or kernel launches at once. Or in other words: a stream represents a queue of GPU operations that get executed in a given order.

The default stream (also known as the null stream) is a special case. It's a synchronising stream with respect to operations on the device - no operation in that stream will begin until all previously issued operations in any stream on the device have completed and no operation in other streams can begin until an operation in the default stream has finished.

Example:

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1, N>>>(d_1);
myCpuFunction(b);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

In the code above, all CUDA operations (copying and the kernel launch) are issued to the default stream. The copy functions are synchronous, but the kernel launch is asynchronous. This is in relation to CPU execution - as soon as the `increment` kernel is launched on the device, the CPU executes `myCpuFunction` - `increment` and `myCpuFunction` overlap.

Aside from the default stream, you can also use custom-defined streams. Let's look at an example with a single non-default stream.

Let's say we have a kernel that performs some operation on two vectors and returns a third one (like in our dot product example from lab 4). It's not really important what it does, because all stream-relevant code resides in `main`.

Example:

```
#define N (1024 * 1024)
#define FULL_DATA_SIZE (N * 20)

__global__ void kernel(int *a, int *b, int *c) {
    //code that writes to c
}

int main(void) {
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    int *host_a, *host_b, *host_c;
    int *dev_a, *dev_b, *dev_c;

    cudaMalloc((void**)&dev_a, N * sizeof(int));
    cudaMalloc((void**)&dev_b, N * sizeof(int));
    cudaMalloc((void**)&dev_c, N * sizeof(int));

    cudaHostAlloc((void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault);
    cudaHostAlloc((void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault);
    cudaHostAlloc((void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault);

    //fill host vectors

    for (int i = 0; i < FULL_DATA_SIZE; i += N) {
        cudaMemcpyAsync(dev_a, host_a + i, N * sizeof(int), cudaMemcpyHostToDevice, stream);
        cudaMemcpyAsync(dev_b, host_b + i, N * sizeof(int), cudaMemcpyHostToDevice, stream);

        kernel<<<N / 256, 256, 0, stream>>>(dev_a, dev_b, dev_c);

        cudaMemcpyAsync(host_c + i, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost, stream);
    }

    cudaStreamSynchronize(stream);

    cudaFreeHost(host_a);
    cudaFreeHost(host_b);
    cudaFreeHost(host_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    cudaStreamDestroy(stream);
}
```

There are several new elements here. First of all, to create a stream you need to declare a variable of type `cudaStream_t` and pass its address to the `cudaStreamCreate` function. Secondly, we don't copy whole buffers at once but split them into smaller chunks. For a single stream it's not that important, but you will see its significance when we move to several streams. Next, we use `cudaMemcpyAsync` instead of `cudaMemcpy` to copy data from host to device (and vice versa). The `cudaMemcpyAsync` function is (like its name suggests) asynchronous - it doesn't block the host thread. With `cudaMemcpy` the host has to wait until the copy is complete, whereas `cudaMemcpyAsync` simply places a copy request into a given stream. When the call from the latter function returns, there is no guarantee that the copy has finished (or even that it's started). The only guarantee is that the copy will be finished before the next operation placed into the same stream.

It's necessary to use pinned memory in asynchronous copying - passing pageable memory to `cudaMemcpyAsync` would not cause any errors, but it would make the copy synchronous. So in order to perform overlapping kernel execution and data transfers, you have to use page-locked host memory.

The stream also has to be passed as a kernel argument - this adds the kernel to that stream.

Since all operations in non-default streams are non-blocking with respect to the host code, you will run across situations where you need to synchronize the host code with operations in a stream. There are several ways to do this. In our example we use the `cudaStreamSynchronize` function - it blocks the host thread until all previously issued operations in the specified stream have completed. Another approach is the `cudaDeviceSynchronize` function - this blocks the host code until all operations on the device have finished. You can also use events - first you can record an event in a specific stream and then call the `cudaEventSynchronize` function to wait until that event has occured.

For an example with two overlapping streams, look at exercise 2.

c) NVIDIA Nsight Systems

You may be wondering how exactly will the system behave using one versus more streams. Luckily, we may observe it on a nice, graphical timeline. To do this we will introduce our first performance analysis tool - NVIDIA Nsight Systems. This tool allow to visualise both host and the GPU operations on a single timeline.

Installation

Nsight Systems is available on entropy cluster the same way as nvcc. Sometimes pathing is broken, but it is accessible via `srun --partition=common --gres=gpu:1 /usr/local/cuda/bin/nsys.` It may also be downloaded from https://developer.nvidia.com/gameworksdownload. The tool should be used via `nsys` and `nsys-ui` binaries. You may also find them on students at /tmp/nvidia/ (Note: you must be on students, not specific machine!)

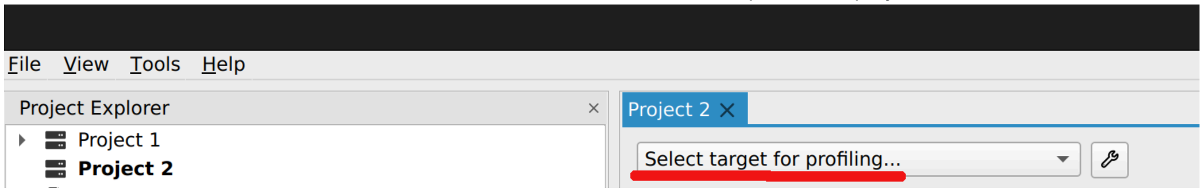
Report generation

We will analyze reports in the `nsys-ui` tool.

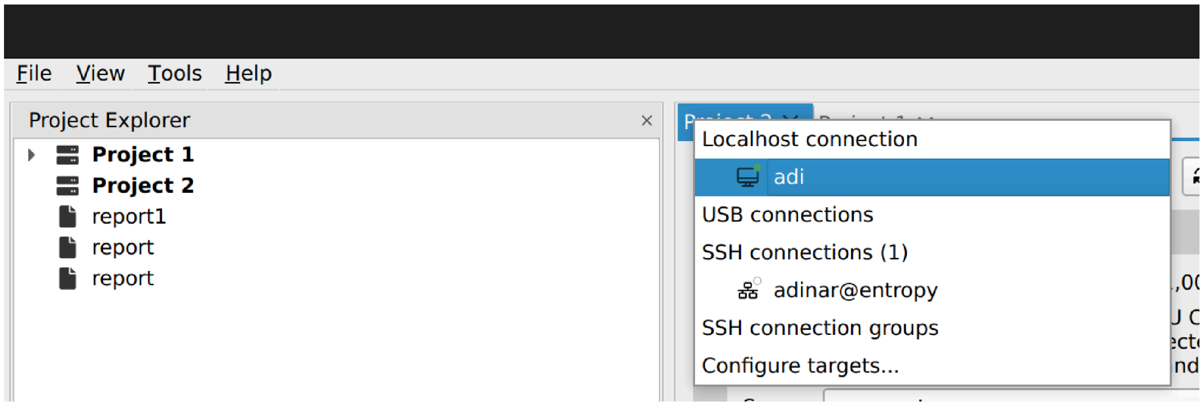
- You may generate profiler report:
  - Using the command line
    - with `nsys` directly: `nsys profile --trace=cuda -o <output_report_file> <binary_to_profile>`
    - ~~or with our `run_batch.sh`: `run_batch.sh <path/to/binary_to_profile> nsys`~~

    In order to open generated report in the `nsys-ui`, use `File -> Open...`
- Directly in the `nsys-ui` tool - **if** you are running on a computer with CUDA capable GPU

  First click on the `Select target for profiling...`. If there no such field, open a new project with `File -> New Project`



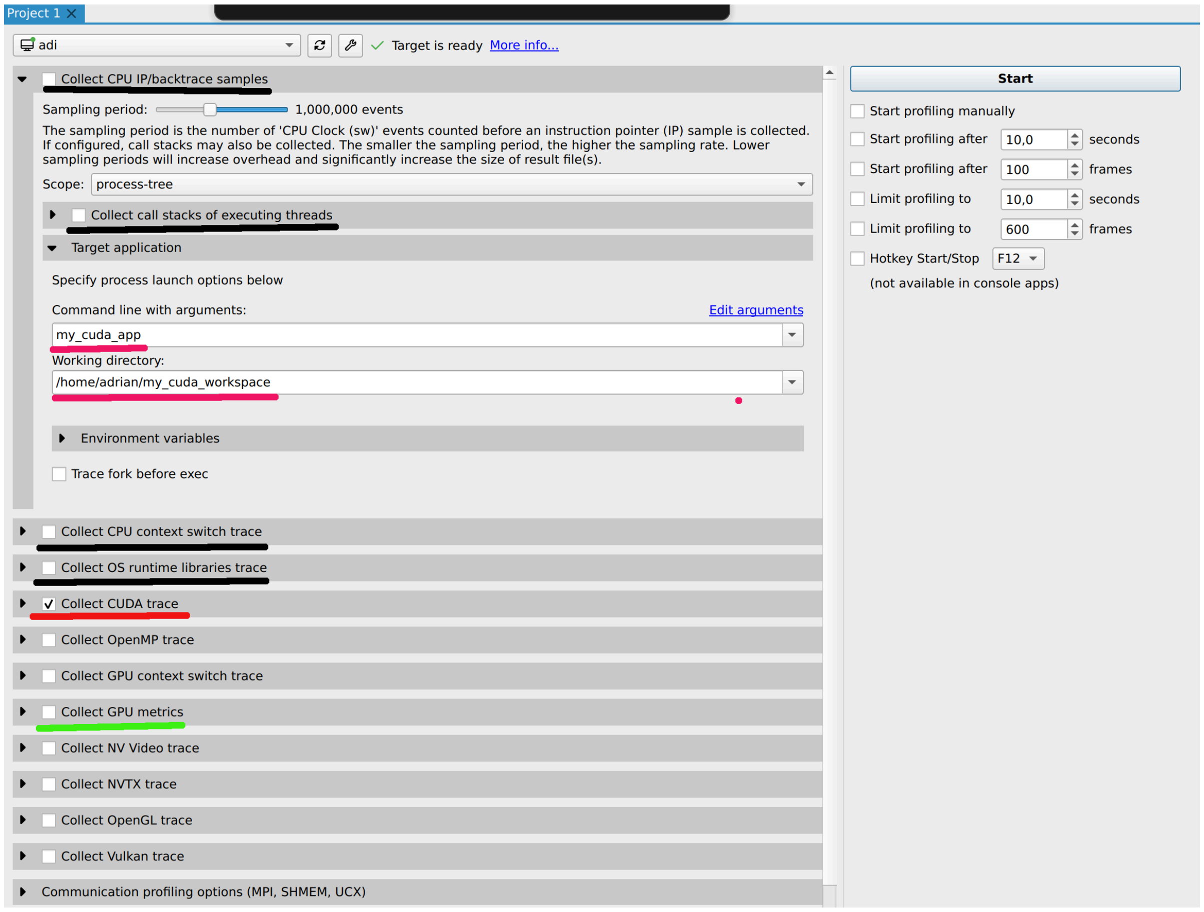
  and choose value from `Localhost connection`



  It might be tempting to use ssh connection to the `entropy` cluster. However, the GPU on the cluster is available via queueing system and thus cannot be used here directly.

  On the next pane fill-in required **red** fields and checkboxes. **Green** checkbox may come in handy later. **Black** checkboxes provide additional CPU and OS data. However, in order to collect them you must set appriopriate "kernel paranoid level" with command
  `sudo sh -c 'echo 2 >/proc/sys/kernel/perf_event_paranoid'`

For the first run, let's keep only the required ones.



Project 1

adi

Target is ready More info...

- [ ] Collect CPU IP/backtrace samples

Sampling period: 1,000,000 events

The sampling period is the number of 'CPU Clock (sw)' 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: Edit arguments

my_cuda_app

Working directory:

/home/adrian/my_cuda_workspace

Environment variables

- [ ] Trace fork before exec
- [ ] Collect CPU context switch trace
- [ ] Collect OS runtime libraries trace
- [✓] Collect CUDA trace
- [ ] Collect OpenMP trace
- [ ] Collect GPU context switch trace
- [ ] Collect GPU metrics
- [ ] Collect NV Video trace
- [ ] Collect NVTX trace
- [ ] Collect OpenGL trace
- [ ] Collect Vulkan trace

Communication profiling options (MPI, SHMEM, UCX)

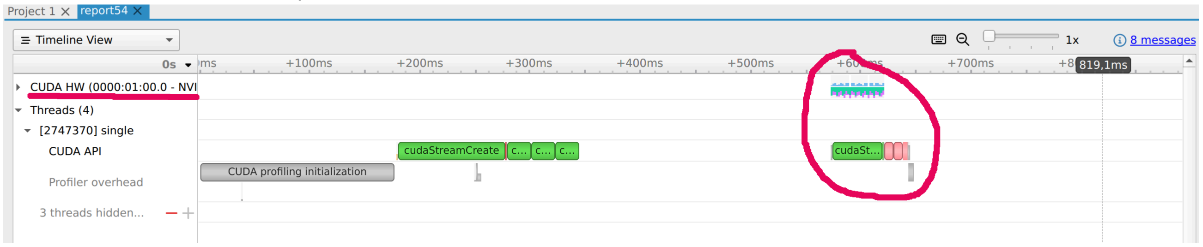
Start

- [ ] 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
- [ ] Hotkey Start/Stop F12 (not available in console apps)

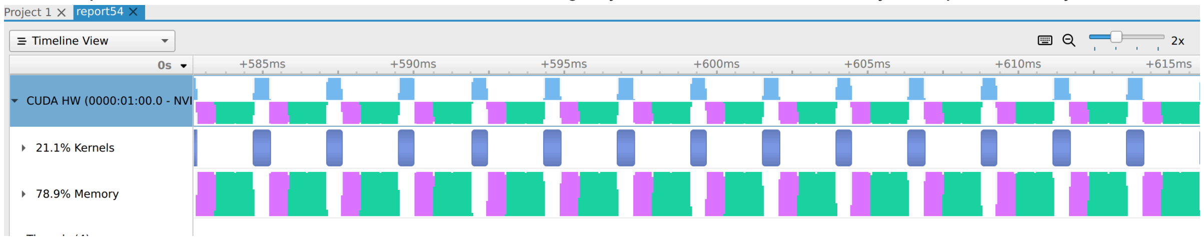
When everything is set up, click on the `Start` button on the right. Profiler will now do its work.

Analysis

You should now see the timeline view. It's often useful to open the `CUDA HW` dropdown on the left. What really interest us is in the red circle. Let's zoom in to this part - use `Ctrl + <mouse scroll>` or select desired area and double click it.



In close-up we see that most of the time GPU is not running any kernel - 80% of the time it just copies memory back and forth.



3) Multi-GPU support (Note - depending on our reservation this section may not work)

In GPGPU computations it's typical to have several GPUs in a single host system. In fact, even our *arnold* machine has 4 discrete GPUs installed. It's relatively easy to divide computations between GPUs in a multi-GPU system. All you need to do is call the following function whenever you want to switch the GPU:

```
cudaSetDevice(int deviceId);
```

This function switches the execution context to the GPU identified by deviceId. An example of using several GPUs for reduction can be seen in `lab5/multigpu/simpleMultiGPU.cu`.

4) Exercises

a) Page-locked memory

Compile the `lab5/plm/plm.cu` file. Execute it and write down the bandwidth - it's the result with pageable memory. Now edit the plm.cu file and replace malloc calls with cudaHostAlloc. Write down the bandwidth again. What's the difference?

b) Streams (1 point)

Let's see how overlapping memory transfers and kernel executions can improve the performance of CUDA applications. In `lab5/streams/single.cu` you can find the code described in section 2 (it uses a single non-default stream). Modify it so it uses 2 streams instead - each stream should handle half of all memory transfers and kernel executions. Compare the results from both versions using NVIDIA Nsight Systems.

5) Sources

- CUDA Programming Guide
- *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Sanders J., Kandrot E.
- *Programming Massively Parallel Processors: A Hands-on Approach (Third Edition)*, Kirk B., Hwu W.

Ostatnia modyfikacja: piątek, 28 marca 2025, 09:48