1 Setting up a CUDA Environment

This section will describe how to configure a development environment that's suitable for compiling and executing CUDA programs. By default, this assignment will assume the use of the SECS Linux server `yoko`, which is installed with an Nvidia graphics card. An additional subsection at the end will point to some resources on how to harness a CUDA-enabled GPU on a personal computer.

1.1 Accessing the SECS servers

- For the following steps, you will need an SECS account. If you do not already have one, an SECS account can be requested here:

  https://oakland.edu/secs/technology-office/

- The SECS servers can be accessed when off-campus with the use of a VPN. SECS provides VPN software that is compatible with Windows, MacOS, and Linux. The latest official SECS instructions on how to set up the VPN can be found at the above link under TECHNOLOGY RESOURCES -> Resources. A copy of the instruction is also posted with this assignment in case the link does not work.

- Once the VPN is installed, configured, and enabled, the SECS servers can be accessed via SSH from a terminal. Connect to the server `yoko.secs.oakland.edu` using your SECS username and password using the following command.

  ```
  ssh USERNAME@yoko.secs.oakland.edu
  ```

  You will be prompted for your SECS password before the connection is accepted.

- Further documentation on the SECS servers, including those which have CUDA-enabled GPUs, can be found at the above link under TECHNOLOGY RESOURCES -> Servers -> Linux Servers.

- If you already had files saved on the SECS network, you can run the `ls` command to view them after connecting to the server.

1.2 The CUDA environment

The primary tool when developing a CUDA program is the Nvidia CUDA compiler, `nvcc`. On the `yoko` server, `nvcc` should have already been made available by default, which you can verify by running `which nvcc`. If successful, this command will show the location of `nvcc` on the filesystem.

`nvcc` is a compiler for the language *CUDA C/C++*, which is a *superset* of both the standard C and C++ languages. That is, `nvcc` is still able to compile any C or C++ program that a normal compiler for those respective languages could. The remainder of this assignment will only use CUDA C.

- As a first test of `nvcc`, run the command `nano hello-world.c` to open the terminal's text editor and enter the following simple C program:

```
#include <stdio.h>

void main() {
  printf("Hello world!\n");
}
```

- Save and close the file. (Recall that in the `nano` text editor, Ctrl+X attempts to close the current file, after first prompting you to save the file if it has changed.)
- Compile the program with `nvcc hello-world.c -o hello-world`. The command-line argument "`-o hello-world`" tells the compiler to write the executable output to the file `hello-world`. If the `-o` flag isn't given, `nvcc` will write the executable to the file `a.out` by default.
- Run the executable with the command `./hello-world`. You should see the output `Hello world!`, just as you would with when using a normal C compiler.

1.3 Installing CUDA locally

If you have a relatively new Nvidia GPU installed in a personal computer, it is likely CUDA-enabled, meaning you would be able to follow these assignments without needing to use the remote SECS server. This will require substantially more configuration than is necessary on an SECS server, but may be worthwhile for interested individuals who plan to use CUDA beyond the scope of this class.

In order to follow the remainder of the assignment on a local environment, you will need to install the CUDA Toolkit. Instructions are provided by Nvidia and will depend on your operating system.

- A list of CUDA-enabled GPUs is provided by Nvidia here:

  https://developer.nvidia.com/cuda-gpus
- CUDA installation instructions for Windows:

  https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html
- CUDA installation instructions for Linux:

  https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html

Once the CUDA Toolkit is installed, add the `bin` subdirectory from the installation directory to your PATH variable. This will make the newly installed CUDA executables available from the terminal. If this has been done successfully, the terminal command `which nvcc` should correctly display the location of the CUDA compiler.

2 The CUDA Programming Model

The goal of the CUDA programming model is to provide a high-level interface to the highly parallelizable computational power of a GPU. This section will summarize the underlying structure of each CUDA program by example, as well as give a method of profiling CUDA programs that will be used to compare them to their serial counterparts.

2.1 Thread hierarchy

A function that is to be executed on the GPU is generally called a *kernel*. When a CUDA program calls a kernel function, the program must also allocate that kernel a certain number of threads. Each thread will then execute its own copy of that function, either in parallel or in sequence, according to the kernel's definition.

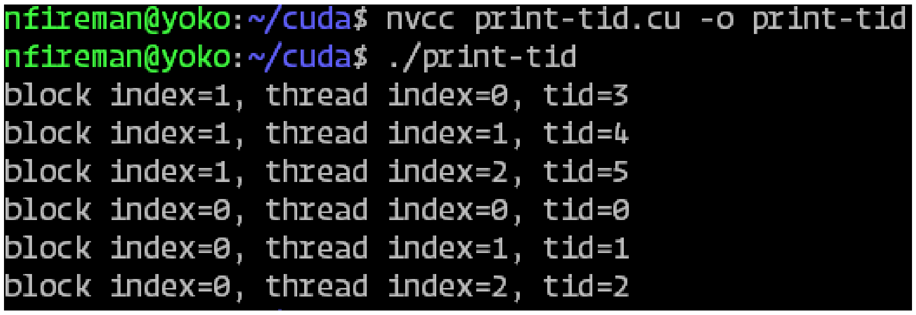
The threads assigned to a kernel are divided into blocks, where each block has the same number of threads. Thus, each thread is associated to both a block index, as well as a thread index within that block. This data can be accessed by device code using the following CUDA-specific global variables:

- `blockDim.x` contains the number of threads in each block.
- `blockIdx.x` contains the executing thread's block number.
- `threadIdx.x` contains the executing thread's index within its block.

If N threads are allocated, then each thread can use the above three values to compute its own globally unique *thread ID* (written `tid` in this assignment) between 0 and $N-1$. This pattern is used frequently in CUDA programs:

```
int tid = blockDim.x * blockIdx.x + threadIdx.x;
```

Program 1. Compile and execute `print-tid.cu` in your CUDA environment. You should get output similar to the following:



```
nfireman@yoko:~/cuda$ nvcc print-tid.cu -o print-tid
nfireman@yoko:~/cuda$ ./print-tid
block index=1, thread index=0, tid=3
block index=1, thread index=1, tid=4
block index=1, thread index=2, tid=5
block index=0, thread index=0, tid=0
block index=0, thread index=1, tid=1
block index=0, thread index=2, tid=2
```

There are several language features of CUDA C to note about this program:

- The `__global__` keyword indicates that the marked function executes on the device but is called from the host (in other words, the marked function is a kernel function). A similar keyword, `__device__`, means the marked function both executes on and must be called by the device.
- The triple angle bracket notation seen in `print_tid<<<2,3>>>()` is known as a *kernel launch*. Calling a kernel function in this way from the host means that 2 thread blocks that contain 3 threads will each call that kernel function (allocating 6 threads altogether).

- The `cudaDeviceSynchronize()` function blocks the program's process until each device thread has finished executing. Without this synchronization call, the host process may terminate before one or more threads has a chance to print its statement to the console.

- Finally, note that each `tid` value printed to the console is unique and between 0 and 5, as expected.

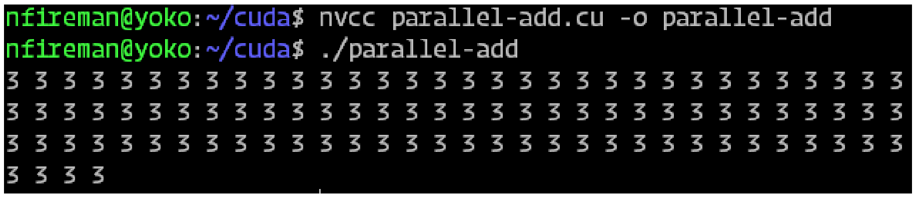
Exercise 1. Change the `print_tid` kernel launch to use 4 blocks of 3 threads each, recompile and execute the program, and verify that the thread ID values are again unique from 0 to 11.

2.2 Memory model

In the context of a CUDA program, the CPU is known as the *host* and the CUDA-enabled GPU is known as the *device*. The host cannot directly access the device's memory (and vice versa); the memory of one must be copied to the memory of the other by calling a CUDA library function. Therefore, a CUDA program will often broadly consist of the following steps:

1. Allocate sufficient memory on the device to store data needed for a computation.
2. Copy the data from host memory to device memory.
3. Perform a computation on the device using the data in device memory.
4. Copy the results back from device memory to host memory.

Program 2. The program `parallel-add.cu` illustrates these steps in a minimal CUDA program. Run and compile the `parallel-add.cu` program and compare your observed output to the following:



```
nfireman@yoko:~/cuda$ nvcc parallel-add.cu -o parallel-add
nfireman@yoko:~/cuda$ ./parallel-add
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3
```

The program allocates `N` threads on a single block, and performs one addition per thread in the kernel function `kernel_add`. To verify that the program is working correctly, it prints the resulting sums after copying them from the device back to the host.

This program introduces the following features of CUDA C:

- The function `cudaMalloc` is the standard way to allocate device memory. Consider the call:

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

  – The second argument `N * sizeof(int)` denotes the number of bytes of device memory to allocate.

  - The value `&dev_a` is the address of the pointer `dev_a`. Its address is passed instead of the value itself since `cudaMalloc` will write the pointer to allocated device memory into the address of `dev_a`, changing its value. This is known as a *pass by reference*.
  - Since `dev_a` is of type `int*`, `&dev_a` will be of type `int**`, as `&dev_a` is a pointer to `dev_a`. However, since `cudaMalloc` is written to allocate any amount (that is, any type) of memory, it expects its pointer argument to be of type `void**`, which signifies a pointer to *typeless* memory. To accommodate this, `&dev_a` must be cast to `void**`.
  - Once `dev_a` has been set to a device memory pointer by `cudaMalloc`, it is an error to attempt to read its value on the host. In order to read device memory from the host, a suitable function from the CUDA API must be used.

- The function `cudaMemcpy` is used both to copy memory from device to host and vice versa. Consider the two calls:

  ```
  cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);
  ```

  - `cudaMemcpy` always copies from its second argument into its first argument.
  - The third argument `N * sizeof(int)` states the number of bytes of memory to copy.
  - The fourth argument describes whether the source and destination memory are on the device or host using a value from the enum `cudaMemcpyKind`. In particular, note that `cudaMemcpyHostToHost` and `cudaMemcpyDeviceToDevice` also exist, though their use is less common.
  - Thus, the first call above copies `N int` values from `a` on the host to `dev_a` on the device. Likewise, the second call copies `N int` values from `dev_c` on the device to `c` on the host.

- Finally, note the usage of the thread ID (`tid`) in the kernel function to assign each thread a unique piece of the overall program's computation.

Exercise 2. Modify `parallel-add.cu` so that the size of the arrays `a`, `b`, and `c` is twice the number of threads. Then, modify the kernel function so that each thread performs two additions on its input arrays instead of just one. Make sure that no two threads perform an addition on the same array index.

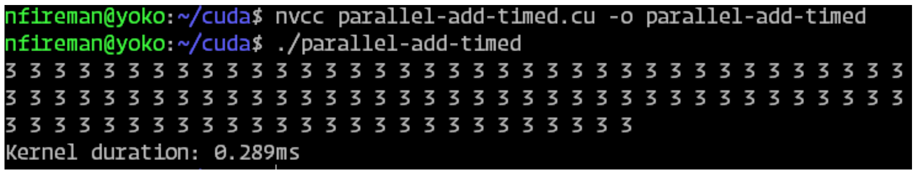
2.3 Events and timing

A common theme in this assignment will be to compare the performance of a regular C program with a parallelized CUDA program that performs the same task. This will be accomplished by comparing the runtimes of the C program to that of the equivalent CUDA program.

Timing CUDA program execution is best done using a language feature known as *CUDA events*. For our purposes, a CUDA event is a timestamp at a certain point in the device's execution.

Program 3. The program `parallel-add-timed.cu` builds on the previous `parallel-add.cu` program by using CUDA events to time the execution of the `kernel_add` kernel. In addition to the output of the original `parallel-add` program, this program also prints the duration of the kernel execution.

Run and compile the `parallel-add-timed.cu` program and compare your observed output to the following:



```
nfireman@yoko:~/cuda$ nvcc parallel-add-timed.cu -o parallel-add-timed
nfireman@yoko:~/cuda$ ./parallel-add-timed
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
Kernel duration: 0.289ms
```

The following features of CUDA events are introduced by this program:

- The type of a CUDA event is `cudaEvent_t`.
- Each CUDA event must be *created* by passing it to `cudaEventCreate` before it can be used.
- Passing an event to the function `cudaEventRecord` saves a timestamp in that event. This is used both before and after the kernel is executed to save timestamps in the events `start` and `stop`, respectively.
- The call `cudaEventSynchronize(stop)` blocks the host until the `stop` event has been recorded (that is, passed to `cudaEventRecord`). This serves the same purpose as `cudaDeviceSynchronize()`, which was used previously.
- The duration between the two recorded events is computed with the call `cudaEventElapsedTime(&milliseconds, start, stop)`. Note that the computed duration is returned by reference using the first `float` argument, rather than as the return value of the function call itself.
- The function `cudaEventDestroy` is used to free an event's memory after the event isn't needed anymore. While this is an important step in large, real-world programs to avoid memory leaks, in simple example programs like this one it's ultimately unnecessary.

Exercise 3. Note that the addition in the kernel function of `parallel-add-timed` is nested in a for loop, where the number of loop iterations is equal to `TRIALS`. By default, `TRIALS` is set to 1. Recompile and execute the program again where `TRIALS` is set to 10, 1,000, and 100,000, and record the resulting duration in each case. Does the duration increase at the same rate as the number of trials?

Exercise 4. The program `serial-add-timed.c` has been provided to perform a comparable task as `parallel-add-timed.cu`, except without the use of CUDA. Use the command

```
gcc serial-add-timed.c -o serial-add-timed
```

to compiler this program with the `gcc` C compiler. Compare its execution time to that of `parallel-add-timed.cu` where `TRIALS` is set to 10, 1,000, and 100,000.

Exercise 5. The constant `N` in both `parallel-add-timed.cu` and `serial-add-timed.c` determines the length of the arrays being added. Compare the execution times after changing the value of `N` in both programs to 10,000, 100,000, and 1,000,000 (while `TRIALS` is still set to `1`, as in the original programs).

Note: You should also remove the `for` loop at the end of the `main` function that prints every element of the result array for large values of `N`.