CUDA Lab

Many of today's computing systems contain powerful graphics cards that can be used as general purpose computing devices. There are several different programming platforms for using these graphics cards for general purpose computing, including:

1. **CUDA** is Nvidia's proprietary but free platform that works (only) with Nvidia devices. It is the oldest and most well-established framework, and it is the best documented.
2. **OpenCL** is a non-proprietary standard that can use any OpenCL compatible cores (CPU or GPU) in a system. It is supported by AMD/Radeon, Nvidia, Apple, Intel, ...
3. **OpenACC** is a (currently proprietary) standard that provides a higher level interface to the GPU, similar to what OpenMP does for multithreading. It is supported by the Portland Group International (PGI) and Cray, among others.

This week, we will explore how to use CUDA.

Part 0: Discover your GPU

Using `nvcc` try to compile `which-device.cu` and interpret the result: memory size, block and thread number, etc. CUDA programs are named using the *.cu* extension.

You can specify with the `-gencode` linking flag to specify the architecture (arch) and code base (code) for your particular graphics card:

- `compute_52` and `sm_52` settings, respectively, to specify the Maxwell architecture,
- `compute_61` and `sm_61` settings to specify the Pascal architecture,
- `compute_62` and `sm_62` to specify the Pascal with integrated GPU (Jetson TX2).

In the `timing.cu` file, you have some examples to measure execution time.

Part I: Vector Addition

To get started, you have at your disposal, the source file and `Makefile`.

This program is a tweaked version of a sample program that comes with Nvidia's CUDA Toolkit. Aside from cleaning up some error-handling and adding support for a command-line argument, the main tweak was to add a sequential loop that performs the same computation as the CUDA kernel, so that we can compare their performance. Use the provided `Makefile` to build the program and verify that it builds and runs without errors before continuing.

For this exercise, our research question is: *for what size problem is the CUDA computation faster than the sequential computation?*

Modify *vectorAdd.cu* so that it reports:

1. The time required by the CUDA computation, specifically:
    a. The time spent copying the A and B arrays from the host to the device.
    b. The time spent computing the sum of the A and B arrays into C.
    c. The time spent copying the C array from the device to the host.
    d. The total time of the CUDA computation (i.e., the sum of a-c).
2. The time required by the sequential computation.
3. Notez aussi le nombre de blocks et de threads utilisés.

Neither of these times should include any I/O, so be sure you comment out the printf() statements that are present in these sections. Likewise, we are not especially interested in memory allocation times, so do not time how long cudaMalloc() takes (unless you really want to).

Use the Makefile to build your modified version of the program. When it builds successfully, run it as follows: ./vectorAdd

By default, the program's array size is set to 50,000 elements.

In a spreadsheet, record and label your timings in a column with the heading 50000. Which is faster, the CUDA version or the sequential version? Are you seeing any speedup for the CUDA version?

Perhaps the problem size is the issue. Run it again, but increase the size of the array to 500,000 elements: ./vectorAdd 500000

As before, record your timings. How do these timings compare to those using 50,000 elements?

Run it again, using 5,000,000 elements, and record your timings. How do these times compare to your previous ones?

Run it again, using 50,000,000 elements, and record your timings. How do these times compare?

Run it again, using 500,000,000 elements, and record your timings. What happens this time?

Create a line chart, with one line for the sequential code's times and one line for the CUDA code's total times. Your X-axis should be labeled with 50,000; 500,000; 5,000,000; and 50,000,000; your Y-axis should be the time.

Then create a second chart, but make this one a "stacked" barchart of the CUDA times with the same X and Y axes as your first chart. For each X-axis value, this chart should "stack" the CUDA computation's

1. host-to-device transfer time
2. computation time
3. device-to-host transfer time

What observations can you make about the CUDA vs the sequential computations? How much time does the CUDA computation spend transferring data compared to computing? What is the answer to our research question?

Part II: Vector Multiplication

Let's revisit the same research question, but using a more "expensive" operation. Multiplication is a more expensive operation than addition, so let's try that.

Create a vectorMult.cu file and modify the Makefile to build vectorMult instead of vectorAdd.

Then edit vectorMult.cu and change it so that instead of storing the sum of A[i] and B[i] in C[i], the program stores the product of A[i] times B[i] in C[i]. Note that you will need to change:

- The CUDA kernel that added the vectors.
- The verification test for the CUDA kernel.
- The sequential code that added the vectors.
- The verification test for the sequential code.

Then build vectorMult and run it using 50,000; 500,000; 5,000,000; and 50,000,000 array elements. As in part I, record the timings for each of these in your spreadsheet, and create charts to help us visualize the results.

What is the answer to our research question? How do your results compare to those of Part I?

Part III: Vector Square Root

Let's revisit the same research question again, but using an even more "expensive" operation AND reducing the amount of data we're transferring. Square root is a more expensive operation than multiplication, so let's try that.

As in Part II, create a vectorRoot.cu and modify the Makefile to build vectorRoot. Then edit vectorRoot.cu and change it so that it computes the square root of A[i] in C[i].

Then build vectorRoot and run it using 50,000; 500,000; 5,000,000; and 50,000,000 array elements. As before, record the timings for each of these in your spreadsheet, and create charts to help us visualize the results.

What is the answer to our research question? How do these results compare to those of Parts I and II?

Part IV: Vector Square

Let's revisit the same research question one last time. You should have seen speedup in Part III, but it could have been either because (i) square root is an expensive operation, or (ii) we only transferred one array (instead of two) from the host to the device.

To try to see which of these two made the difference, we will use a less expensive operation than square root, but keep the amount of data we're transferring the same.

As in Part III, build vectorSquare.cu file and modify the Makefile to build vectorSquare. Then edit vectorSquare.cu and change it so that it computes the square of A[i] in C[i].

Then build vectorSquare and run it using 50,000; 500,000; 5,000,000; and 50,000,000 array elements. As before, record the timings for each of these in your spreadsheet, and create charts to help us visualize the results.

What is the answer to our research question? How do your results compare to those of the previous parts?