Arquiteturas de Alto Desempenho 2024/2025

Practical class AAD_P06 (2024-10-28/2024-11-04 and 2024-10-29/2024-11-05)

Tomás Oliveira e Silva

1 Objectives

Learn how to parallelize code using CUDA and OpenCL. List of things you will learn about:

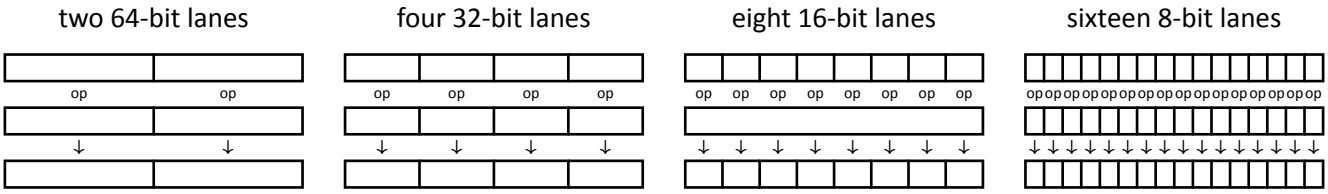
- how to compile and run a CUDA program using the (easy) runtime API,
- how to compile and run a CUDA program using the (hard) driver API, and
- how to compile and run an OpenCL program.

2 The architecture of a modern graphics card

The explanation that follows may not be exactly what it going on a graphics card. However, it should be close enough for you to be able to write efficient programs for it.

Modern processors have one of more execution pipelines (cores). Server processors can have a little more than a hundred cores. Each execution pipeline may allow more than one concurrent execution flow. This is called simultaneous multi-threading (SMT). Intel calls it hyperthreading, and in Intel's case, each core can accommodate up to two execution flows. This means that the pipeline has two complete sets of registers, one for each execution flow, and almost all the rest is shared between the two execution flows.

Modern processors also have wide registers. These are special purpose registers with more than the 64 bits of the general purpose registers. For example, Intel/AMD has 128-bit registers (used by SSE and AVX instructions), 256-bit registers (used by AVX2 instructions), and even 512-bit registers (used by AVX-512 instructions); ARM has 128-bit registers (used by NEON instructions). Each of these special purpose registers can be subdivided in lanes. There exist instructions, the so-called Single Instruction Multiple Data (SIMD) instructions, that perform the same operation, in parallel, on each lane. The following figure shows how these registers are used; op is usually an arithmetic or logical operation, and is applied in parallel to each lane.



On Intel processors, the destination register (bottom) is usually restrained to be one of the two source registers (top). Some of the SIMD instructions on recent processors allow conditional activation of the lanes. This is done with a mask register: a 0 bit in the mask register means that the result of the op operation will be replaced by either zeros (`maskz` instructions) or the contents of the corresponding lane of another source register (`mask` instructions). There exist instruction to move data between lanes and to load/store the entire contents of these registers from/to memory.

Modern NVidia graphics cards take what was written in the previous two paragraphs to extreme levels. There are many execution pipelines (the most powerful graphics cards can have hundreds of them). Each pipeline can accommodate 32 or even 48 flows of execution. Each register has 1024-bits, subdivided into 32 lanes of 32-bits each. So, each instruction acts in the 32 lanes in parallel. All instructions use masks (called predicates); a 0 bit in the mask means that the corresponding lane is inactive. In load/store instructions,

each lane has its own address; data addresses are not shared between lanes, as happens in most SIMD instruction in Intel/AMD processors.

The graphics card hardware is optimized to **manipulate massive amounts of information**, preferably in a repetitive way. For example, this is perfectly adequate to process a large image, where the same operation is applied to each pixel. In order to take advantage on the immense processing power of the graphics card, it is necessary to distribute work to its many execution pipelines. In the programming model used in NVidia's Compute Unified Device Architecture (CUDA) this is solved in an elegant way:

- Because the task is supposed to be repetitive, the programmer specifies what is happening in **a single lane** in one of the execution flows. CUDA calls it a "thread" (and the code is a "kernel"), but that is unfortunate because of the SIMD nature of the architecture: each instruction acts in parallel in the 32 lanes of the 1024-bit register, so what happens in one of these "threads" also happens in the 31 other "threads" that share the same instruction. In CUDA's lingo, the group of 32 "threads" that evolve in lockstep is called a warp. So, one warp is one instruction flow, using 32-way SIMD instructions.
- To take advantage of the extreme simultaneous multi-threading capabilities of each core (NVidia calls the core a Streaming Multiprocessor), several execution flows (i.e., warps) are grouped together and are assigned to the same core. CUDA calls this group of warps a block. So, if one of more warps are blocked due to a high latency operation, say, a memory access, it is likely that there exists at least another warp ready to be executed. This keeps the pipeline busy. Remember, we want to manipulate massive amounts of information, so we are fine as long as there is work ready to be done. Because the simultaneous multi-threading of one core can handle at most 32 warps (48 is some graphics cards, but from now on we will consider that it is only 32), a block can have at most 32 warps, i.e., 1024 CUDA "threads".
- To distinguish the "threads" of a given block, each one is given a set of coordinates: `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`. These are integer variables that we can use in our code. The dimension of the block is also available: `blockDim.x`, `blockDim.y`, and `blockDim.z`. The number of threads of the block is the product of these three values.
- Each core has a small amount of memory (64KiB or so), so the warps of a given block can share data between them in the shared memory area. There exist instructions that implement synchronization barriers at the block level. Warps of different blocks can only share data via global memory.[1]
- Doing things using a single block of "threads" does not take advantage of the many execution cores present on a graphics card (a block executes in a single core). So, in a computational task the programmer usually also specifies a grid of blocks. In the end she/he will have many blocks and many "threads" per block. All blocks have the same dimensions. To distinguish the blocks of a given grid, each one is given a set of coordinates: `blockIdx.x`, `blockIdx.y`, and `blockIdx.z`. Again, these are integer variables that we can use in our code. The dimension of the grid is also available: `gridDim.x`, `gridDim.y`, and `gridDim.z`.
- So, each "thread" is uniquely identified by its `threadIdx` and `blockIdx` coordinates. It is up to the programmer to specify how the entire work is to be subdivided. For example, to handle one element of a unidimensional array in a single "thread" one can use only the x coordinates. The unique identifier of each "thread" would then be `threadIdx.x+blockDim.x*blockIdx.x`.

Recommendations:

- When the 32 "threads" of a warp access memory, the address each one produces should be either the same for all "threads" or be consecutive. If so, the memory controller can handle the request in the smallest number of memory transactions. If not, many transactions that stress the memory subsystem are needed and that makes the program inefficient. Try to avoid it.

[1] All this means that there exit two distinct address spaces: one for local shared memory and one for global memory. There are others. For example, for constants and for textures. Consult the CUDA documentation if you want to know more about this.

- When a control flow statement appears in an execution flow, say, an `if` statement with an `else`, it may be necessary to execute both. This is handled by the hardware in the following way:

  1. The current execution mask, call it `m`, is saved (stack of execution masks). Recall that it specifies with lanes are currently active.
  2. The `if` expression is evaluated and a new mask, call it `t`, is produced: ones when the expression is non-zero (true), zeros when the expression is zero (false).
  3. The new execution mask becomes `m & t`; if is it all zeros skip immediately to the "else" part of the `if` statement.
  4. Execute the "then" part of the `if` statement, and after than, skip to the end of the `if` statement.
  5. To execute the "else" part of the `if` statement, the new execution mask becomes `m & ~t`; if it is all zeros skip immediately to the end of the `if` statement.
  6. Execute the "else" part of the `if` statement.
  7. Finally, restore the original execution mask (`m`).

  So, if there exists control flow divergence, meaning that the "threads" of a warp go different ways the program becomes less efficient, because some of the lanes will be inactive during part of the computation. A `switch` statement can be particularly bad. Likewise for a function that is called only for some of the "threads". Try to avoid that.
- A block is only retired when all its warps terminate. When that happens, a new block takes its place (if there are any yet unassigned in the grid of blocks). So, avoid situations where one warp of a block takes much more time to finish than the others. It is possible to assign up to 8 (small) blocks in the same pipeline. So, it may be better to have blocks of 128 "threads" (8 to fill the pipeline) instead of blocks of 1024 "threads" (only one to fill the pipeline). Blocks of less that 128 "threads" will not use the full capacity of the pipeline. (This applies to NVidia graphics cards.)
- Memory accesses have (collectively) high-bandwidth but also high latency. So, try to perform many arithmetic operations for each memory access. Also, try to reduce as much as possible data transfers between the host abnd the device.

CUDA programming was lots of quirks. To master it, it is really necessary to read the documentation. Here go some useful links:

- CUDA C Programming Guide (clickable link).

  URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`
- CUDA Toolkit Documentation (clickable link).

  URL: `https://docs.nvidia.com/cuda/index.html`
- CUDA runtime API (clickable link).

  URL: `(https://docs.nvidia.com/cuda/cuda-runtime-api/index.html`
- CUDA driver API (clickable link).

  URL: `https://docs.nvidia.com/cuda/cuda-driver-api/index.html`
- Even easier introduction to CUDA (clickable link).

  URL: `https://developer.nvidia.com/blog/even-easier-introduction-cuda/`
- Different models of NVidia graphics cards may have different so-called computing capabilities. Wikipedia does a good job of listing them (clickable link).

  URL: `https://en.wikipedia.org/wiki/CUDA`

Besides NVidia, there exists other brands of graphics cards (AMD, Intel, standalone or in the processor). As CUDA is specific for NVidia cards, to program for them something else is needed: either proprietary tools (AMD ROCm, Intel oneAPI), or OpenCL. OpenCL in principle can handle all graphics cards, and can even run on processors using SIMD instructions.

3 Our first CUDA program: “hello world”, using the easy runtime API

There exist two ways to program in CUDA. The easy way, using the so-called runtime API, with function names beginning with `cuda` followed by a capital letter. And the hard but more powerful way, using the so-called driver API, with function names beginning with `cu` followed by a capital letter.

We will start the easy way. The file `hello.cu` contains the entire code of the “Hello world” program written using the runtime API. CUDA programs that use the runtime API are C++ programs with a small number of extensions to the C++ programming language. The file extension of the source code is usually `.cu`. Open the file with a text editor and examine its contents. Functions that are to be run on the graphics card are identified with the `__global__` key word. They are executed by inserting `<<< >>>` between the function name and its arguments. Inside `<<< >>>` you specify the launch grid parameters. This is done in two different ways in the code; study them.

Finally, compile (study the `makefile`!) and run the program:

```
$ make hello
$ ./hello
```

For this to work you will need to install CUDA on your computer (if it has a NVIdia graphics card). You can also transfer the program do `banana.ua.pt` and work from there. (**Please:** do not leave any Visual studio Code server running there. I will kill it with prejudice.)

4 A different “hello world”, using the harder driver API

The CUDA driver is closer to what you will need to do when switching from CUDA to OpenCL. To program with it there is much more things you will have to take care yourself; the runtime API does automatically some of them for you. First, you have to split your code. The part that is to be run on the graphics card in placed in a `.cu` file, with functions declared with `__global__` as before. You then compile it into an object file understood by the CUDA driver. The rest of your program, meant to be run on the processor — the host — will be a plain C or C++ program. You compile it as usual and link it with the CUDA library.

Because all CUDA driver API functions return an error code, it is good practice to test it. In this way, any error is caught and reported as soon as possible. The file `cuda_driver_api_util.h` contains a macro than is used for this. Study it.

The file `hello_cuda_driver_api.cu` contains the graphics card code (the CUDA kernel). Study it.

The file `hello_cuda_driver_api.c` contains the host code. It is by far the more complicated one. It initializes the CUDA driver API interface, selects a graphics card (there may exist more than one at it is possible to make them all work with the same program), loads the `.cubin` file in the graphics card, allocated memory on it to store results, calls the CUDA kernel, transfers the results back to host memory, prints them, and finally cleanly terminates the program. Study it with care.

Finally, study the `makefile` to see how it is compiled, compile it and run it.

```
$ make hello_cuda_driver_api
$ ./hello_cuda_driver_api
```

5 The different “hello world”, but now using OpenCL

The OpenCL “hello world” program does the same as the CUDA driver API program. The main coding differences are in the terminology used: `__kernel__` instead of `__global__`, and the global coordinates are computed using `get_global_id()`. Also, the device code, placed in a `.cl` file, is read and passed to

the driver; it could even have been stored in a string in the C code. A Just In Time compiler is then invoked to compile it. The rest is similar, but with functions with different but related names. Finally, the OpenCL code also reports some details about the OpenCL environment. Similar functionality exists in the CUDA driver API (which to make the program as short as possible we did not use in the `hello_cuda_driver_api` program).

Study the files `open_cl_util.h`, `hello_open_cl.cl`, `hello_open_cl.c`, and, of course, the `makefile`. Compile and run it.

```
$ make hello_open_cl
$ ./hello_open_cl
```

(The CUDA toolkit also allows compilation of OpenCL programs.)

6 Median filter using the runtime API

Our final demonstration program applies a median filter to an image. We will use the image produced by the `MandelbrotImage` program of our first practical class. so, before continuing, do a

```
$ make MandelbrotSet.png
```

in the directory holding the files of the fist practical class and move the file `MandelbrotSet.pgm` to the directory holding the files of this class. Note that that's the `.pgm` file and not the `.png` file.

In a median filter each intensity of each pixel of the image is replaced by the median of the intensities of the pixels of its neighborhood. We used a 3×3 neighborhood, and so for each pixel it is necessary to sort 9 numbers. The CUDA kernel `median` does this using a so-called sorting network. The sorting network uses minimum and maximum instructions, which the graphics card has, to avoid conditional jumps. Without conditional jumps where is no control flow divergence, so the kernel runs optimally.

The code provided in `median_filter_cuda.cu` also allocated "managed" memory. That is a memory region that is directly addressable from both the host and the device (done at the hardware level using the PCIe interface). This avoids the need to transfer data between the host and the device.

Study the program, compile and run it:

```
$ make median_filter_cuda
$ ./median_filter_cuda
```

Profile it using `nvprof`:

```
$ nvprof --unified-memory-profiling off ./median_filter_cuda
```

Run it again with the `cudaMemPrefetchAsync()` code segment active, just replace `#if 0` with `#if 1`. Try also other block sizes: instead of 32×32 blocks, why not 16×16, 8×8, 32×2 blocks, or even other sizes? Which one is the best?

Finally, clean up.

```
$ make clean
$ rm MandelbrotSet.pgm
```

7 Once again: parallelization of `MandelbrotCount.c`

It is now time to put all that you learned in this class in action. Modify the `MandelbrotCount.c` to do the computation in parallel using CUDA. But the double precision performance of almost all graphics cards is abismal, so use single precision instead (`float`).