Partnership for Advanced Computing in Europe

Introduction to HPC Programming

6. CUDA and OpenCL

Valentin Pavlov <vpavlov@rila.bg>

CAPACITIES

e-infrastructure

About these lectures

- This is the last of series of six introductory lectures discussing the field of High-Performance Computing;
- The intended audience of the lectures are high-school students with some programming experience (preferrably using the C programming language) having interests in scientific studies, e.g. physics, chemistry, biology, etc.
- This lecture provides an overview of the CUDA and OpenCL environments, used to program GPU accelerators.
- Some of the slides provide only pointers for the lecturer (and the readers) to elaborate on since its not feasible to describe all features here!

Graphical Processing Units (GPUs)

- The Graphical Processing Units (GPUs) are co-processor devices (they cannot work independently without a central processing unit), which in the past were used predominantly for modelling and visualization of 3D scenes;
- These tasks are solved with the help of certain mathematical constructs, for example scalar, vector and matrix algebra, trigonometric functions, etc., applied over floating point numbers;

GPUs

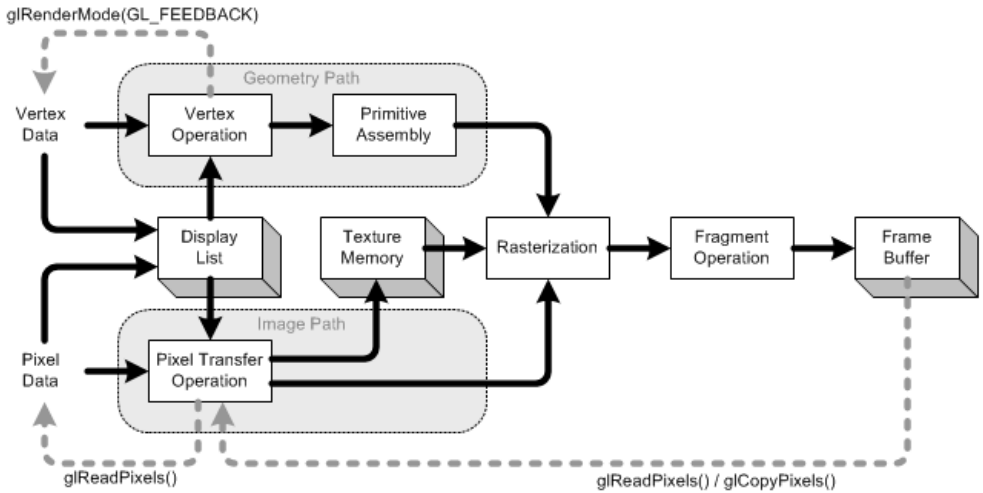
- The calculations involved in rendering a 3D scene are very specific and the device that carries them out is conceptually very simple;
- Moreover, the algorithm that is used to render the scene naturaly requires the same calculations to be performed on many different objects independently of each other, so they can be executed in parallel in SIMD fashion;
- This led to the creation of extremely powerful GPUs with a lot of processing cores, which can perform limited set of operations on many different data objects in parallel.

Fixed and programmable GPUs

- It is of no surprise that there is desire to use such power not only 3D rendering, but also for carrying out general purpose calculations.
- Initially, the GPUs did not provide any means for programatically controlling the sequence of calculations; they only provided means for parametrization of the scene — objects and camera positions, lightning, materials, etc. the calculations themselves follow a fixed algorithm.
- Later, GPU manifacturers started to incorporate programmable features, which enabled control over the rendering algorithm and provided the means to perform general purpose calculations using GPUs (GPGPU);

OpenGL Pipeline, to be elaborated

- A GPU is controlled by some API like OpenGL or DirectX;
- In order to visualize a scene in OpenGL, there are certain operations that need to be performed — the so called *rendering pipeline*.



GPGPU

- In order to be able to use the GPU for general purpose calculations (GPGPU) some of the above steps were made programmable.
- These are the steps that perform operations on *vertices* and *fragments*: the **vertex shader** and **fragment shader**;
- This ability was introduced in OpenGL 2.0, along with a special programming language in which programmers can write their shaders – GLSL (GL Shading Language);

How is GPGPU achieved using OpenGL/GLSL ?

- This is more of a legacy subject, since its quite hard and error-prone, so is given just as an illustration:
  - Arrays of data are represented as textures;
  - Calculations are encoded in fragment shader kernels;
  - The calculation is triggered by drawing a plain rectangle on the whole screen;
  - In order to get the results of the calculation, the result is not rendered on the screen, but on either off-screen buffer or a texture (so called “feedback” mechanism);

OpenCL

- The above method for performing calculations is quite awkward and the constructs used are not typical and well-known to programmers in general;
- It is virtually impossible to debug such a program and is not standardized, so it will probably not run on a device made by other vendor;
- In order to create a more programmer-friendly interface and to provide for portability, the OpenCL standard was created;
- OpenCL, OpenGL, GLSL and other related standards are maintained by the Khronos Group (`http://khronos.org`);

OpenCL

- OpenCL provides a standardized interface for using the computational power of the GPU;
- In OpenCL we no longer deal with stuff like shaders, textures, rendering, etc., but instead we have arrays of data, computational procedures (kernels) and an easy way to exchange data between the CPU and GPU;
- The computational procedures (kernels) are compiled and linked into a program, much like the way a C program is compiled and linked. The compiler and linker are provided by OpenCL and are accessed by the main program.
- OpenCL is implemented in many different platforms and works on AMD GPUs, NVIDIA GPUs, CPUs, etc.

OpenCL terminology

- **Platform** — a single machine can have several different OpenCL implementations, provided by different vendors;
- Context — an house-keeping object that encompasses a given calculation;
- **Device** — a single machine can have multiple OpenCL devices installed, e.g. several CPUs and several GPUs;
- Host — the CPU on which the program that calls the OpenCL functions works;

OpenCL terminology

- **Kernel** — an entry point to a certain calculation routine; it is part of the code that is compiled for the given OpenCL device;
- A kernel can be scheduled on a 1D, 2D or 3D domain and a separate kernel instance is executed for each “point” (index) in that space. Several of these instances can be executed in parallel, depending on the available resources.
- The total number of indices for which the kernel is being executed is called **the global work size**;
- Individual indices are known as **work-items**;

OpenCL terminology

- If there is a need for communication between different work-items (so they are not independent), they can be grouped into **work groups**. The size of the group is known as **the local work size**.
- **Queue** — all commands are send to a command queue for asynchronous execution;
- Everything in OpenCL is asynchronous, so the program needs to organize an event loop in order to have feedback from the OpenCL devices;

OpenCL example to be elaborated upon

- One of the best OpenCL implementations is the one by AMD, called AMDAPP, freely available from AMD's web-site;
- The AMDAPP SDK includes many examples of OpenCL usage, like the NBody simulation:
- `/opt/AMDAPP/samples/opencl/cl/app/NBody`
- `/opt/AMDAPP/samples/opencl/bin/x86_64/NBody`

NVIDIA CUDA

- Altough OpenCL is a **big** advance compared to GLSL, it is still quite hard to program;
- Thus NVIDIA created CUDA — a **closed** platform (not standartized) for programming GPGPU applications easily, but it has the big disadvantage that it only works on NVIDIA GPUs;
- In CUDA the level of abstraction is considerably higher than in OpenCL and many of the tiresome details of initialization and maintenance of the calculations are hidden;
- CUDA calls are generally not asynchronous and thus a CUDA program is much easier to implement.

nvcc

- In an OpenCL source code the kernels need to be stored in strings and given to API calls to compile/link, etc.
- In CUDA, the kernels source code is written along with the rest of the C code. For that purpose, some new reserved words and synatactic constructs are added to the language.
- Since the ordinary C compiler does not understand this new syntax, CUDA programs are compiled with the NVIDIA compiler wrapper `nvcc`, which pre-processes the source code and then feeds the result to the standard C compiler;
- The new syntax include `__global__`, `__host__`, `__device__`, `__shared__`, `<<<n, k>>>`, etc.

Simplest example

```
__global__ void kern(void)
{
}

int main()
{
    kern<<<1, 1>>>();
    printf("Hello, world!\n");
    return 0;
}
```

Explanation of the example

- `__global__` marks a function that is called from the host, but is executed on the device (kernel);
- The text between <<< and >>> is the kernel execution; inside the brackets we given the number of **blocks** and the number of threads inside each block. In this case we have 1 block of 1 thread each.
- Blocks are the equivalent of OpenCL work-groups and threads inside the block are the same as the work-items in OpenCL;
- The “number” of blocks and the “number” of threads inside them can be 1D, 2D or 3D vector, as in OpenCL;

Memory

- Except the kernel, which is defined with `__global__`, we can also define any number of functions that are called by the kernel; they are defined with `__device__`
- The memory of the CUDA device and the host are physically different spaces;
- Kernels and functions that it calls can only access device memory;
- This means that the data that the host works with cannot be accessed by the device; any data that the kernel needs must be transferred from the host to the device; any results that the kernel produces must be transferred from the device to the host after the end of the calculations.

Memory

- There is a certain pattern that is observed around a kernel call:
  - allocate memory on the device;
  - copy data from host memory to device memory;
  - call the kernel;
  - copy results from device memory to host memory;
  - free device memory;
- Allocation, deallocation and copying between host and device memory (and vice versa) is done through the `cudaMalloc`, `cudaFree` and `cudaMemcpy` functions.

```
__global__ void multk(int *a, int *b, int *c)
{
    *c = *a * *b;
}

int main()
{
    int a = 6, b = 7, c;
    int *da, *db, *dc;
    cudaMalloc((int **)&da, sizeof(int));
    cudaMalloc((int **)&db, sizeof(int));
    cudaMalloc((int **)&dc, sizeof(int));

    cudaMemcpy(da, &a, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(db, &b, sizeof(int), cudaMemcpyHostToDevice);

    multk<<<1, 1>>>(da, db, dc);

    cudaMemcpy(&c, dc, sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(da);
    cudaFree(db);
    cudaFree(dc);
    printf("c = %d\n", c);
    return 0;
}
```

Block parallel execution

- If we specify more than one block (the first “number” in the triangle brackets), they will be executed in parallel, as long as there are enough resources;
- Inside the kernel we can identify the block we’re currently in by using the pre-defined variables `blockIdx.x`, `blockIdx.y` and `blockIdx.z`;
- For example,
  `c[blockIdx.x] = a[blockIdx.x] * b[blockIdx.x];`
  (which starts to look like a scalar multiplication of vectors, apart the final reduction)

Thread parallel execution

- We could also specify more than one thread in a block (the second “number” in the triangle brackets). Again, they will be executed in parallel, but in contrast to blocks they are not necessarily independant and can share memory, thus need to synchronize, etc.
- Inside the kernel we can identify the thread we’re currently executing by using the pre-defined variables `threadIdx.x`, `threadIdx.y` and `threadIdx.z`
- For example,
  `c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];`

Block *and* Thread parallel execution

- We can use blocks and threads at the same time. Indexing then becomes a little more compicated:
- `index = blockIdx.x * blockDim.x + threadIdx.x`
- Why do we need this complication?
- As said above, blocks cannot communicate with each other, while threads can.
- Threads inside a block can use shared memory, which is exceptionally fast.
- Shared memory is declared in the kernel with `__shared__`;

Dot product using threads only

```
#define N 512

__global__ void dot(int *a, int *b, int *c)
{
    // Shared memory for the partial results
    __shared__ int tmp[N];
    tmp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    __syncthreads();  // Wait for all threads to finish the above

    // One of the treads (e.g. 0) sums the results in tmp[]
    if(threadIdx.x == 0) {
        int sum = 0;
        for(int i = 0; i < N; i++ ) {
            sum += tmp[i];
        }
        *c = sum;
    }
}
```

What about many blocks?

- The above example works with a single block, which is enormous waste of resources;
- We can easily change the code so as to use more blocks;
- We only need to change two things:
  - The indexing becomes:
    ```
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    tmp[index] = a[index] * b[index];
    ```
  - And since now `sum` is only the partial result out of each block, we need to *add* it to the final result `*c`, instead of assigning it:
  - `*c += sum;`
- **However!** There is a hidden **race condition** in this line!

Race condition

- The statement `*c += sum;` actually contains 3 separate operations:
  - **Read** the current value of `*c`;
  - **Modify** the value by adding `sum` to it;
  - **Write** back the modified value to `*c`;
- We can then end up in a situation in which one of the blocks has read the value, but while it did modify, another block had already written a new value (consider the sequence Block 1 Reads, Block 2 Reads, Block 1 Modifies, Block 2 Modifies, Block 2 Writes, Block 1 Writes)
- This leads to loss of data and as consequence — to wrong result!

Atomic operations

- In order to avoid that situation, we need to use **atomic operations**, which are read/modify/write operations that guarantee that no other block has access to the same memory cell involved in the operation.
- CUDA offers several such atomic operations: `atomicAdd()`, `atomicSub()`, `atomicMin()`, `atomicMax()`, etc.
- Thus, the last line should actually read `atomicAdd(c, sum);`

Conclusion

- That wraps up the contents of these introductory lectures.
- The field of HPC programming is enormous and cannot be encomassed by any amount of lectures alone; thus we have tried to provide useful starting points and some hooks to help grasp some of its more important concepts.
- Parallel programming is a very exciting discipline and we hope that more young people will take keen interest in it, since HPC is the only feasable way into the future, with Moore's law starting to fail.
- Godspeed!