

Introduction à OpenCL et applications

Philippe Helluy¹, Anaïs Crestetto¹

¹Université de Strasbourg - IRMA

Lyon, journées groupe calcul, 10 novembre 2010

Sommaire

- 1 OpenCL
- 2 Solving a transport equation with OpenCL
- 3 Local memory optimizations
- 4 Atomic operations

OpenCL

GPU architecture

A modern Graphics Processing Unit (GPU) is made of:

- Global memory (typically 1 Gb)
- Compute units (typically 27)

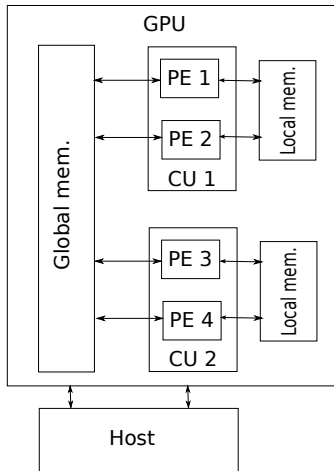
Each compute unit is made of:

- Processing elements (typically 8)
- Local memory (typically 16 kb)

The same program can be executed on all the processing elements at the same time.

- All the processing elements have access to the global memory
- The processing elements have only access to the local memory of their compute unit.
- The access to the global memory is slow while the access to the local memory is fast.

A (virtual) GPU with 2 Compute Units and 4 Processing Elements



OpenCL

- Host: the computer into which the GPU is plugged.
- Kernel: a program that is executed on the processing elements.
- OpenCL means “Open Computing Language”. It includes:
 - A library of C functions, called from the host, in order to drive the GPU.
 - A C-like language for writing the kernels that will be executed on the processing elements.
- Practically available since september 2009. The specification is managed by the Khronos Group (OpenGL).

Work-items and work-groups

In order to perform a complex task, a kernel has to be executed many times.

- Each execution of a kernel is called a work-item.
- Each work-item is identified globally by a global ID, i , $0 \leq i < N_{global}$.
- A work-group is a collection of work-items running on the processing elements of a given compute unit. They can access the local memory of their compute unit.
- Each work-item is identified locally, in its work-group, by a local ID, j , $0 \leq j < N_{local}$.
- Each work-group is identified by a group ID, k , $0 \leq k < N_{group}$.
- $i = k \times N_{local} + j$.

Work-items and work-groups

- If N_{global} is the total number of work-items, N_{local} the number of work-items in a work-group and N_{group} the number of work-groups, then

$$N_{global} = N_{group} \times N_{local}.$$

For efficiency reasons, it is advised that $N_{group} \gg 27$ and $N_{local} \gg 8$.

- The distribution of the work-groups on the compute units and the work-items on the processing elements is managed by the OpenCL implementation.
- The order of execution is completely arbitrary. The algorithm has to take it into account...

OpenCL/CUDA

| CUDA | OpenCL |
|----------------|--------------------|
| multiprocessor | compute unit |
| scalar core | processing element |
| global memory | global memory |
| shared memory | local memory |
| local memory | private memory |
| kernel | kernel |
| block | work-group |
| thread | work-item |

Solving a transport equation with OpenCL

Transport equation

We want to solve numerically the transport equation

$$\partial_t w + u \partial_x w = 0.$$

- The unknown is a function $w(x, t)$ that depends on the space variable $x \in [0, L]$ and the time variable $t \in [0, T]$.
- The constant velocity $u > 0$ is given.
- The initial condition at $t = 0$ is known $w(x, 0) = w_0(x)$.
- We also know the left boundary value $w(0, t) = 0$.

Finite volume approximation

- We consider a space step $\Delta x = L/N$, a time step $\Delta t = \beta \Delta x / u$, the instants $t_n = n \Delta t$ and the points $x_i = (i + \frac{1}{2}) \Delta x$ (x_i is the middle of the cell $C_i =]i \Delta x, (i + 1) \Delta x[$, $i = 0 \dots N - 1$).
- N is the number of approximations points in the x direction.
- The CFL number β is such that $0 < \beta < 1$.
- We want to compute an approximation $w_i^n \simeq w(x_i, t_n)$.
- The approximation is given by an upwind finite volume scheme

$$\frac{w_i^{n+1} - w_i^n}{\Delta t} + u \frac{w_i^n - w_{i-1}^n}{\Delta x} = 0, \quad n = 0, 1, 2 \dots$$

A simple kernel

At a given time-step n the values of w_i^n for $i = 0 \dots N - 1$ are stored in the global memory of the GPU in an array `wn[]`. The work-item of global ID i , will compute the new value w_i^{n+1} . The kernel is the following

```
__kernel void transport(__global float* wn) {
    int i = get_global_id(0);
    int N = get_global_size(0);
    float dx = 1.f / N;
    float dt = dx * 0.8f;
    if (i > 0 && i < N) wn[i] = wn[i] - dt / dx * (wn[i] - wn[i - 1]);
}
```

The boring part

Now, we have to plug all the wires between this kernel, the GPU and the host, initialize the variables and the OpenCL framework, *etc.*

The “boring part” can be largely simplified with recently developed tools (PyOpenCL, for instance)

1) Create an OpenCL context

```
// context creation
Context = clCreateContext(
    0, // optional
    1, // number of detected devices
    &Devices[0], // chosen device
    NULL, // optional
    NULL, // optional
    &status); // error code
assert (status == CL_SUCCESS);
```

The devices list is obtained from other OpenCL API calls...

2) Create a command queue

```
CommandQueue = clCreateCommandQueue  
  (Context, // the context  
   Devices[0], // the chosen device  
   0, // optional  
   &status); // error code  
assert (status == CL_SUCCESS);
```


3) Create a program

```
Program = clCreateProgramWithSource
(Context, // the context
 1, // number of source strings
 (const char **) & prog, // string with
 // kernel source
 NULL, // optional
 &err);
assert(Program);
```

The kernel source is read from a file and put into a C++ string. Note that at this point, the kernel is still not build. Compilation is made at runtime.

4) Build the program

```
err = clBuildProgram(Program, 0, NULL, NULL,  
                    NULL, NULL);  
assert(err == CL_SUCCESS);
```

The OpenCL kernel compiler is invoked at runtime. If the build is not successful, it is of course possible to obtain the compiler errors with the function `clGetProgramBuildInfo(...)`.

5) Create a kernel

```
Kernel = clCreateKernel(  
    Program,          // the program  
    "transport",     // name of the function  
                    // that defines the kernel  
    &err);
```

A program source may contain several kernel functions. This instruction is needed to select a particular function in the program source.

6) Create a buffer for the initial data in the GPU

```
wa_gpu = clCreateBuffer(  
    Context, // the context  
    CL_MEM_READ_WRITE, // the buffer will be r/w  
    sizeof(cl_float) * _N, // size of the buffer  
    NULL, // optional  
    NULL); // optional
```

7) Copy the initial data in the GPU

```
err = clEnqueueWriteBuffer
(CommandQueue, // the command queue
 wa_gpu, // the buffer that has
           // to be filled in the gpu
 CL_TRUE, // indicates a blocking write
 0, // optional
 sizeof(float) * _N, // size of the buffer
 wa, // pointer to the host
      // memory to copy in the gpu
 0, // optional
 NULL, // optional
 NULL); // optional
assert(err == CL_SUCCESS);
```

8) Link the arguments of the kernel to the right buffer

```
err = clSetKernelArg(  
    Kernel,      // the kernel  
    0,          // number of the argument (0,1,2,...)  
    sizeof(cl_mem), // size of the argument value  
    &wa_gpu); // pointer to the gpu buffer  
              //in global memory
```

The same function has to be called for defining each argument of the kernel. We present here only the case of an argument pointing to global memory. For local or constant memory arguments, the call to `clSetKernelArg(...)` is slightly different. See [OCL] and below.

9) Compute the time-steps on the GPU

```
while(t<0.25){  
    t=t+dt;  
    err = clEnqueueNDRangeKernel(  
        CommandQueue, // the command queue  
        Kernel, // the kernel to execute  
        1,NULL,  
        &NbGlobal, // total number of work-items  
                // (= N the number cells)  
        &NbWorks, // number of work-items  
                // inside a work-group  
        0,NULL,NULL);  
}
```

In our simple example, the value of NbWorks is not very important because we do not use the local memory.

10) Read back the results from the GPU

```
clEnqueueReadBuffer(  
    CommandQueue,  
    wa_gpu,  
    CL_TRUE, 0,  
    sizeof(float) * _N,  
    wa,  
    0, NULL, NULL);
```

This call to `clEnqueueReadBuffer(...)` copy the buffer pointed by `wa_gpu` to the buffer pointed by `wa`. Then, we can compare to the results obtained on the host CPU.

11) Enjoy your work !

Execution on a MacBook GPU (NVidia GeForce 9400M)

```
Données calcul: Mem GPU=512Gb  
Nb Procs=4  
Nb Works Max=512  
Mem locale=16kb  
Plateformes:1  
Devices:2  
copie dans le gpu  
NbGlobal=51200 _N=51200  
début du calcul...  
temps gpu=3 s  
temps cpu=14 s  
Computed 48205/51200correct values!  
speedup=4.66667
```

Other important notions

- Optimizations in local memory;
- Atomic operations;
- Sharing objects with OpenGL (not tested);
- Using multiple devices, including CPU (not tested).

Local memory optimizations

Sod's shock tube

- We consider a model for an inviscid compressible gas

$$\partial_t w + \partial_x f(w) = 0.$$

- $w(x, t)$ is now a vector $\in \mathbb{R}^3$. $w = (\rho, \rho u, \rho E)^T$. The density is ρ , the velocity u and the total energy E . The flux is given by $f(w) = (\rho u, \rho u^2 + p, (\rho E + p)u)^T$.
- The pressure is given by $p = (\gamma - 1)(\rho E - \frac{1}{2}\rho u^2)$ where $\gamma > 1$ is the polytropic constant.
- The initial condition is piecewise constant (Riemann's problem)

$$w(x, 0) = \begin{cases} w_L & \text{if } x < 0, \\ w_R & \text{if } x > 0. \end{cases}$$

Finite volume scheme

The approximation is given by a finite volume scheme

$$\frac{w_i^{n+1} - w_i^n}{\Delta t} + \frac{f_{i+1/2}^n - f_{i-1/2}^n}{\Delta x} = 0.$$

The numerical flux at the cell boundaries

$$f_{i+1/2}^n = f(w_i^n, w_{i+1}^n)$$

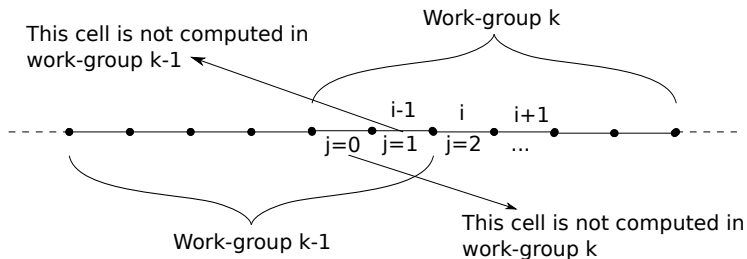
has a rather complex expression (we use the VFRoe scheme [MFG99] with an entropy correction at sonic points [HHMM09]).

Main kernel

- At a given time-step n the values of w_i^n and w_i^{n+1} for $i = 0 \cdots N - 1$ are stored in the global memory of the GPU in two arrays of `cl_float4` (a value is unused).
- In order to avoid too much access to global memory, we first copy for each work-group a part of the array containing w_i^n in local memory. The cache size imposes the maximal size of the work-group.
- The first and the last cell in a work-group are not computed, which implies a two-cell overlap between the work-groups.
- At the end of the time step the updated values are written back in the other array in global memory.
- A pointer exchange permits to avoid another transfer in global memory before the next time step.

Work-group overlap

- k : work-group number
- i : global work-item number
- j : local work-item number in work-group k



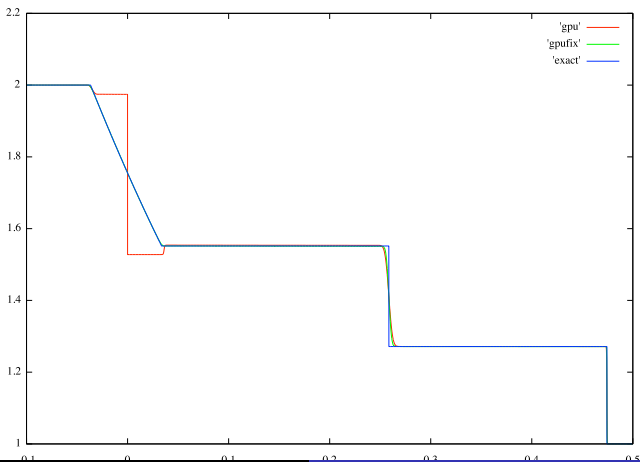
Perfs

We compare several GPUs for a 100,000 cells computations.

| | time (s) | proc. | CPU/GPU |
|-------------------------------------|----------|-------|---------|
| NVIDIA GeForce GTX 260 | 75 | 216 | 63 |
| ATI Radeon HD 5750 | 102 | 720 | 46 |
| NVIDIA GeForce 9400M | 572 | 16 | 8 |
| NVIDIA GeForce 9600M GT | 281 | 32 | 17 |
| AMD Phenom II x4 i810 (OpenCL) | 2057 | 4 | 2.3 |
| AMD Phenom II x4 i810 (on one core) | 4722 | 1 | 1 |

Results

Comparison between the exact and the numerical solution with or without entropy correction for 12,600 cells.



Atomic operations

The PIC method

- Vlasov-Poisson system:

$$\begin{cases} \frac{\partial f}{\partial t}(x, v, t) + v \frac{\partial f}{\partial x}(x, v, t) - E(x, t) \frac{\partial f}{\partial v}(x, v, t) = 0 \\ \frac{\partial E}{\partial x}(x, t) = \rho(x, t) = 1 - \int_{-\infty}^{+\infty} f(x, v, t) dv \end{cases}$$

- The Particle-In-Cell (PIC) method:

we consider N macro-particles: position x_k , velocity v_k and weight $\omega_k = \omega$,

we approach f by: $f_N(x, v, t) = \sum_{k=1}^N \omega_k \delta(x - x_k(t)) \delta(v - v_k(t))$.

- We solve the electric field equation at grid points $x_j = i\Delta x$, $\Delta x = L/N_x$.
- We move the macro-particles with the Newton's law: $\frac{dx_k}{dt} = v_k$ and $\frac{dv_k}{dt} = -E$.

Algorithm

We know x_k^n , v_k^n for each particle $k = 0, \dots, N-1$, E_i^n at each grid point $i = 0, \dots, N_x - 1$.

- Explicit Euler's scheme (for example) for moving the particles:

$$v_k^{n+1} = v_k^n - \Delta t E_{i_k}^n, \text{ with } i_k \text{ such that } x_k^n \in [x_{i_k}, x_{i_k+1}]$$

$$x_k^{n+1} = x_k^n + \Delta t v_k^n,$$

- the charges ρ_i^{n+1} calculated by linear interpolation from the position of the particles located in $[x_{i-1}, x_i]$ and $[x_i, x_{i+1}]$ at time $n+1$,
- electric field E_i^{n+1} : we need the charges ρ_j^{n+1} , for $j < i$.

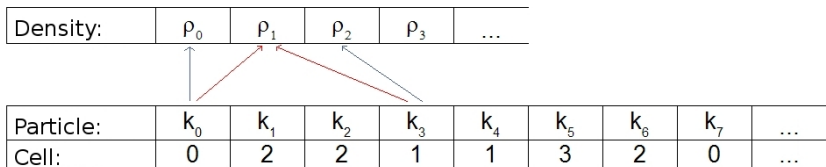
GPU programming

We work with particles ($N \sim 1\,000\,000$) and on a grid ($N_x \sim 128$).

- Particle move: one work-item per particle,
- E : one work-item for each grid point.

charge computation

First idea and first problem: move the particles and compute their contribution to ρ in the same Kernel \rightarrow problem when two particles add at the same time their contribution to the same grid point:



k_0 reads ρ_1^n ,

k_3 reads ρ_1^n ,

k_0 adds its contribution to ρ_1^n and writes ρ_1 : $\rho_1 = \rho_1^n + \text{contr}(k_0)$,

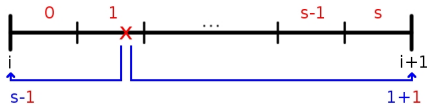
k_3 adds its contribution to ρ_1^n and overwrites ρ_1 : $\rho_1 = \rho_1^n + \text{contr}(k_3)$.

Finally $\rho_1 = \rho_1^n + \text{contr}(k_3)$ instead of $\rho_1 = \rho_1^n + \text{contr}(k_0) + \text{contr}(k_3)$.

Atomic addition

Solution: use the atomic addition to add the contributions of particles.

- When a particle reads ρ and adds its contribution, the other work-items are stopped until this addition is finished. Can be complex [SDG08].
- Simple solution: add integer values.
- Cut each cell and transform the contributions into integer values:



if $x_k \in [x_i, x_{i+1}]$ and if k is in the subcell j , this particle adds $s - j$ to $contr_i$ and $1 + j$ to $contr_{i+1}$.

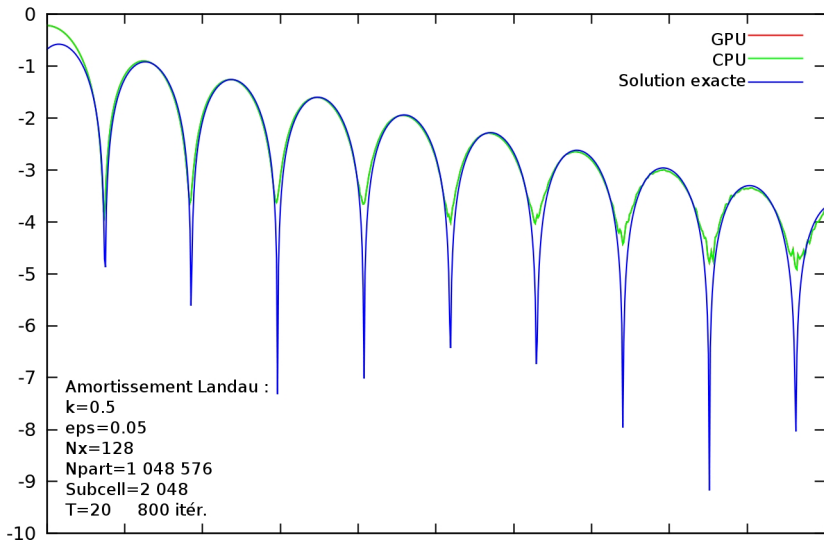
Speedups

| 524 288 particles | CPU | NVIDIA GeForce GTX 260 | ATI Radeon HD 5750 |
|-------------------|------|------------------------|--------------------|
| Time | 37s. | 15s. | 5s. |
| Speedup | | 2.47 | 7.40 |

| 1 048 576 particles | CPU | NVIDIA GeForce GTX 260 | ATI Radeon HD 5750 |
|---------------------|------|------------------------|--------------------|
| Time | 75s. | 30s. | 9s. |
| Speedup | | 2.50 | 8.33 |






| 2 097 152 particles | CPU | NVIDIA GeForce GTX 260 | ATI Radeon HD 5750 |
|---------------------|-------|------------------------|--------------------|
| Time | 151s. | 61s. | 19s. |
| Speedup | | 2.48 | 7.95 |

Graph



Comments

- 2D version coming soon...
- flaws in the current OpenCL implementations: memory (ATI), crashes (ATI/NVIDIA), computations on the CPU device (NVIDIA), documentation (ATI), double precision (ATI/NVIDIA), *etc.*
- lack of libraries for numerical algorithms.
- But OpenCL is already portable and efficient. It looks very promising.

-  Helluy, Hérard, Mathis, Müller. A simple parameter-free entropy correction for approximate Riemann solvers, 2009.
-  Masella, Faille, Gallouët. On an approximate Godunov scheme. Int. J. Comput. Fluid Dyn. 12 (1999), no. 2, 133–149.
-  NVIDIA. OpenCL Best Practices Guide. August 2009.
-  Khronos Group. The OpenCL specification, version 1.0. June 2009.
-  G. Stantchev, W. Dorland, N. Gumerov. Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU, J. Parallel Distrib. Comput. 68 (2008).