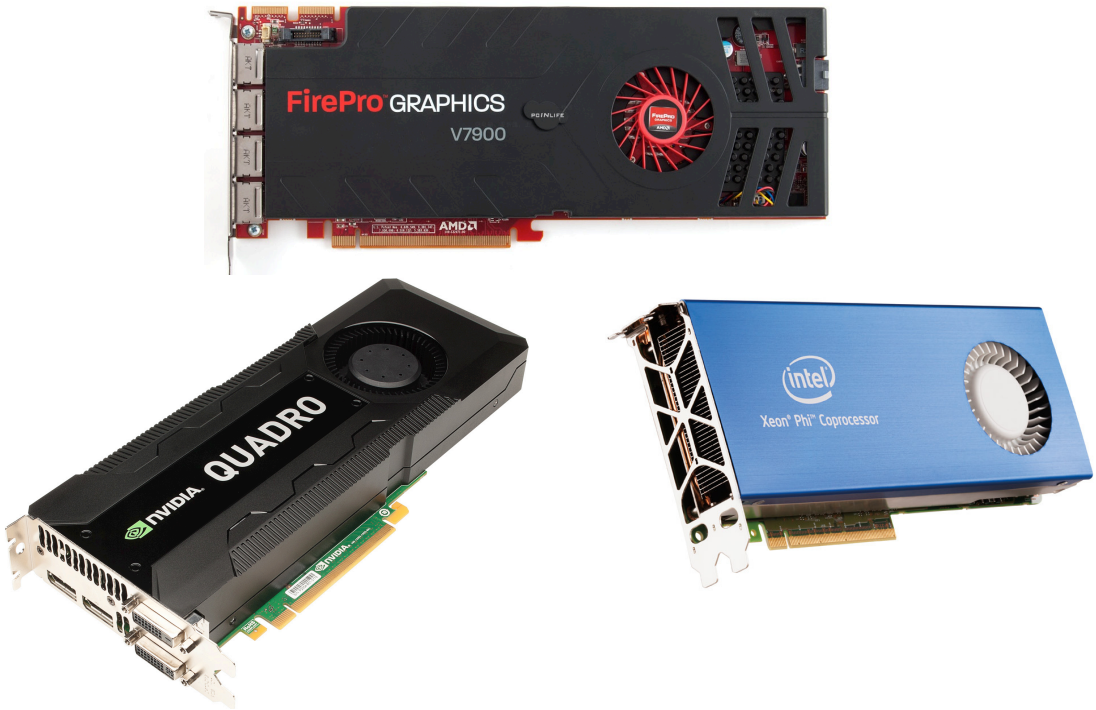
Port meshing tools and solvers that deal with unstructured meshes on GPUs

The GMlib library



Loïc MARÉCHAL / INRIA, Gamma Project

November 2018

Document v1.21

Table des matières

Cover pictures : various OpenCL compatible graphic cards and accelerators.

1 Introduction

Today's world of HPC offers many different kinds of architectures like clusters, monolithic supercomputers, GPUs, FPGAs or shared memory machines to name a few. All of them suffer from the same limitation : how to efficiently connect huge amount of computing power on one side, to an equally huge amount of memory on the other side ?

Up to the 1990s the main problem was to increase both memory and computing capabilities. Now, those two resources are aplenty and their sole limitation are the bus or network that connect them together. Consequently, a fast High Performance Computer is no more one that offers the most Terra-Bytes or Terra-Flops, but one with the highest memory and network bandwidth and the shortest latency.

Increasing this memory bandwidth has become increasingly difficult because of the necessity for any processor to access any location in memory. Such arbitrary access, called random access, requires to go through an interconnection matrix, known as crossbar or switch, whose complexity grows with the square of the number of connected elements, like chips and memory busses.

It is to circumvent this difficulty that the first vector supercomputers, like the IBM 3090 or Cray 1, were invented in the 1970s. Each of their compute unit could efficiently access to a limited part of memory and suffered a big-time penalty when accessing data outside these bounds. Practically speaking, if such a machine possesses 64 adding units and has to add two vectors `u[ 1->256 ]` and `v[ 1->256 ]`, the unit $n°1$ could only access the memory location efficiently that 0 `u[1]`, `u[65]`, `u[129]` and `u[193]`, conversely unit $n°2$ would access quickly `u[2]`, `u[66]`, `u[130]` and `u[194]` (`u[ 2 + i MODULO 64 ]`). This imposes a heavy constrains to the programmer but also greatly simplifies the hardware design : indeed, its complexity grows linearly with the number of computing units.

Today's GPUs borrow some vector characteristics from their predecessors, but the way they are implemented makes them much more convenient from the programmer's point of view. GPU programming is based on simple loops, called kernels, that are run in parallel by the numerous compute units. This way, porting software on GPU is more akin to multithreading than to old times vectorization. Nevertheless, the vector nature of GPUs shows through this apparent simplicity in two different ways :

The macroscopic level : maximum performance is obtained when memory is accessed via the main loop index, that is, when the code swipes linearly through memory. This introduces the most important concept in GPU computing : memory access predictability. These processors are much more efficient when the location where data is to be read or written at each step of the loop is known beforehand. Thus, the following loop is foreseeable : $\forall_i\ u(i) = v(i) * w(i)$, while this one is not :$\forall_i\ u(i) = v(w(i))$. Even if we know that $w(i)$ will go through memory linearly $v(w(i))$ will access it depending on the value stored in $w(i)$ which is unknown before execution. Such access is non-linear and unpredictable, and consequently will run an order of magnitude slower.

The microscopic level : many GPUs are made of single compute units that are working on small size vectors, typically 4 elements wide, that are very useful when performing geometric calculations. It is more efficient to define a set of 3D coordinates as a four floating points vector (`float4 coord` in OpenCL) than a table of four scalar values (`float coord[4]`). In the first case, adding two sets of coordinates would take one processor cycle, treating a 128 bits super word in one go, while it would take four scalar 32 bit operations in the second case.

1.1 Motivation

The two main motivations to programming on GPUs are their performance / price ratio that is much higher than that of traditional CPUs, and their room for evolution that is also greater than CPUs whose number of cores is now reaching some plateau.

Nonetheless, one should remain realistic when it comes to pricing, as high-end HPC dedicated GPU cards often cost as much as a server. And when it comes to performance, the numbers advertised by manufacturers or published in many scientific papers that surf on the GPU trend are only theoretical. The "cheating" comes in two ways :

First, they compare a GPU implementation to a sequential CPU one, which is unfair as all CPUs now have many cores.

Second, in order to best demonstrates the GPU's computing power, they run the academic codes on simplified data that are not representative of real industrial codes and test cases.

In real life, when porting an industrial code that is well multithreaded to a 4000 compute units GPU, the performance increase may not be higher than two to four compared with a good 32-core server. Anyway, such a limited speed increase is interesting in terms of price, electric consumption and space (an extension card takes less space a rack unit).

Since the expected practical speedup is not so great, it is critical not to waste too much time trying to port any code to GPUs. And that is precisely the GPU programming drawbacks, it is very tedious and time consuming !

It comes out those two aspects are very important to achieve a successful port :

-Shaping your data so that they're accessed efficiently by the GPU and it minimizes their transfer to the graphic card.

-Optimizing and debugging the code on the GPU, which is cumbersome because of the lack of interactions between kernel launches (there are no such things as"printf") and the absence of memory protection.

Henceforth, it appeared to me that providing developers with a library that could help deal with unstructured meshes data in a transparent way could lighten the programmer's burden. The GPU Meshing Library, *GMlib*, offers meshing data structures that are very easy to set up and transfer, as well as sample codes that deal efficiently with them.

The operating mode is the following :

-Choose the right data type among the proposed list and copy your data to the *GMlib*'s own structures. -Start from one of the sample codes whose data access matches yours and fill your code inside the template kernel.

This leads to talking about the way to connect your code with the library through its API.

1.2 API

Data Types

The foundation of the *GMlib* is its data types, either pre-defined ones like GmlVertices, GmlEdges, GmlTriangles, GmlQuadrialerals, GmlTetrahedra and GmlHexahedra, or the free user defined GmlRawData. Allocation, storage and transfer of those data are taken care of by the library. The programmer interacts with the *GMlib* via simple GmlGet and GmlSet commands. As an example is more telling than a long explanation, here is a basic code that sets up the coordinates of 100 vertices.

```
VerIdx = GmlNewData(GmlVertices, 100, 0, GmlInput);
for(i=0;i<100;i++)
  GmlSetVertex(VerIdx, i, coords[i][0], coords[i][1], coords[i][2]);
GmlUploadData(VerIdx);
```

The procedure `GmlNewData` allocates a hundred vertices and returns a unique identifier that should be provided to other *GMlib*'s commands that will work on this vertex table. Internally, two tables are allocated, one in main memory accessible by the CPU only and the other one on the GPU card's dedicated memory. None of those buffers being directly accessible by the programmer.

Instead, filling the table is to be done by looping over the 100 vertices and calling the `GmlSetVertex` command that writes a single vertex set of coordinates into the library's internal buffer in main memory.

At this stage, the library owns a copy of users' data in its own internal buffer, but this data is still located in main memory and thus is inaccessible from the GPU. Transferring the whole table from the main memory buffer to the GPU buffer is done in one step with the `GmlUploadData` command. The section 2 will present every single API commands.

Kernels

The other key entity is the kernel, it is a procedure written in OpenCL language that will be executed by the GPU. Integration between a C program and an OpenCL procedure go the following way :

1. transform the source file my_code.cl into a C header file my_code.h with the help of the command cl2h provided by the library.
2. simply include this header file from the C file that is responsible for launching the GPU procedure with a simple `#include "my_code.h";`
3. compile the OpenCL kernel at the start of the execution of the C code with the *GMlib* command `KrnIdx = GmlNewKernel(my_code, "my_procedure");`. This command

will compile the code, send it to the graphic card and return a unique index that will be used to launch this kernel afterwards.

4. launch the kernel on the GPU, for example to do some work on the previously allocate vertex coordinates, with the `LaunchKernel(KrnIdx, 100, 1, VerIdx)` command. The four arguments being : the index of the kernel to execute, the number of loop iterations (here the number of vertices), the number of different data types this kernel should access to (only one in this case) and finally comes the list of the data types indices (here only the vertex data type is given).

Programming in OpenCL and doing those back and forth transfer between the CPU and the GPU is bewildering in the beginning. That is why a collection of 10 template kernels covering direct and indirect memory writes for each of the five kinds of elements are provided to start from. The easiest way to make your first steps in the GPU world is to start from a sample code and modify it to suit your needs.

1.3 Kernel types

Two memory access patterns

When it comes to writing data in memory, most codes dealing with meshes fall into two distinct categories :

— Direct memory writes loops, which means that every time a data is stored to a table in memory, this table is indexed through the main loop index. Consequently, when several iterations of the loop are processed concurrently, they will write at separate memory location.

— Indirect memory writes loops. This happens when we are looping over a certain kind of mesh entity and writing data to another kind of related entity. For example, a code may loop over triangles and write data associated with those triangles' vertices. In such situation, if two different triangles are processed at the same time, they may share a common vertex and write concurrently at the same memory location, leading to a calculation error. One of the compute units may write its result right after the over, overriding the first value, or both may write at the same time producing the much dreaded "Nan" (Not a Number) error !

Sections 2.3 and 2.4 illustrate those two situations.

Direct access kernels

Complete code is provided (both C and OpenCL parts) for each of the five kinds of elements supported by the library.

Each sample code reads a mesh made of vertices and a particular kind of element, initialisms the *GMlib*, allocates and transfers the mesh on the GPU and compiles and launches an OpenCL kernel that computes each element's barycenter. The result is then written into a table that is retrieved from the GPU to the main system's memory to be further processed by the C code.

These kinds of loops that read memory indirectly (it loops over elements but accesses their vertex data), and write memory directly (it stores the barycenter element-wise), are very common in finite elements analysis.

The given examples cover all five types of elements : edges, triangles, quadrilaterals, tetrahedra and hexahedra.

Indirect access kernels

Handling concurrent memory writes is more complex and requires to split the loop into a scatter/gather pair as will be explained in the section 2.4.

Each code does the same as the direct access kernels but instead of computing the element's barycenter, it tries to optimize the element's geometrical shape by smoothing the vertices positions. Doing so, two triangles processed concurrently may write to the same vertex coordinates. The sample codes deal with these problems by splitting the loop into three independent ones.

Similarly the examples cover all five types of elements.

1.4 OpenCL programming

This document does not aim at giving a full course on OpenCL programming. The reference document in this matter is the official specification from the Khronos Group [2], a gathering of manufacturers and software editors that champion this open development platform.

It may also be of interest to read some OpenCL optimization guides ([3] or [5]), or experience report on porting existing software to GPU ([4]).

2 Usage

2.1 Installation and compilation

The library being very compact, it is made of a single .c file and two headers .h files, the easiest way is to include a copy of the *GMlib* and recompile it altogether with your own project.

All you need to add to any software using the *GMlib* is an include to *gmlib2.h* and to provide the path to your system's OpenCL installation at compile time. During linking, you also need to provide the path to the OpenCL runtime library.

Under Linux and Windows you need to install the OpenCL development kit provided by the graphic card's manufacturer. Then you need to tell the compiler where to find the includes ("-I /usr/local/SDK_CUDA" with an NVIDIA card) and add *-lOpenCL* at compile time.

Under MacOs X you only need to add *–framework=OpenCL* when compiling, this will add the correct paths to includes and libraries as OpenCL is part of the system since version 10.6.

2.2 Initialization

The library is initialized with the command `GmlInit( DeviceIndex )` whose single argument is the index of any OpenCL capable device available on your system. The list of such devices can be obtained with the command `GmlListGPU()` that prints them on the screen.

As for now, only one device can be used at the same time and the *GMlib* is not thread safe and reentrant. This will change over time and multithreading and hybrid computing capabilities will be added to the next version.

Note that you can run an OpenCL code on the CPU itself if your system does not have any GPU, which is very useful for portability purpose.

2.3 Example 1 : computing a mean value for each triangle of a mesh

2.3.1 Problem

We are provided with a mesh made of NV vertices, NT triangles and we would like to compute the barycenter of each triangle on the GPU and get the result back in a main memory table.

The process unfolds this way :

1. initialize the library
2. allocate a vertex data type
3. fill the vertex coordinates to the data type's table
4. transfer the coordinates to the GPU's memory
5. allocate a triangle data type
6. fill the triangle vertex indices in the data type's table
7. transfer the triangles' data to the GPU memory
8. allocate a raw data type to store the barycenter coordinates
9. compile the kernel source code at runtime
10. launch the kernel on the GPU giving as an argument the indices of the three allocated data types
11. wait for completion and transfer back the barycenters table from the GPU
12. finally, loop over the barycenter to retrieve the coordinates from the internal library's buffer

2.3.2 CPU part of the code

We suppose that the mesh was previously read from a file and that the vertices and triangles are numbered from 0 to $n - 1$.

```
#include "CalMidKernel.h"
int triangles[nt][3];
float coords[nv][3], centers[nt][3];

GmlInit(GmlGpu);

VerIdx = GmlNewData(GmlVertices, nv, 0, GmlInput);
for(i=0;i<nv;i++)
  GmlSetVertex(VerIdx, i, coords[i][0], coords[i][1], coords[i][2]);
GmlUploadData(VerIdx);

TriIdx = GmlNewData(GmlTriangles, nt, 0, GmlInput);
for(i=0;i<nt;i++)
  GmlSetTriangle(TriIdx, i, triangles[i][0], triangles[i][1], triangles[i][2]);
GmlUploadData(TriIdx);

MidIdx = GmlNewData(GmlRawData, nt, sizeof(cl_float3), GmlOutput);
CalMid = GmlNewKernel(calmidkernel, "CalMid");
GmlLaunchKernel(CalMid, nt, 3, VerIdx, TriIdx, MidIdx);

DownloadData(MidIdx);
for(i=0;i<nt;i++)
  GetRawData(MidIdx, i, &centers[i][0], &centers[i][1], &centers[i][2]);
```

2.3.3 GPU part of the code

We start by getting the loop iteration we are processing with the `get_global_id` command. Remember that we are only focusing on loop's kernel and that increasing the loop counter is taken care of by the OpenCL library. Then we read the triangle's three vertex indices that are stored in a 3D integer vector. Since the variable `idx` is declared as a `int3`, a vector of three integers, the indices are read from the table in one vector instruction. Each separate index can be accessed by adding suffixes to the vector variable : `idx.s0`, `idx.s1` and `idx.s2`.

Likewise, the vertex coordinates are stored in 3D floating point vectors (`float3`) and will be retrieved and computed in a vector way. Accessing to `coords[ idx.s0 ]` reads the x,y,z coordinates from the first triangle's vertex and adding (`+ coords[ idx.s1 ]`) will add the second vertex coordinate element wise with one single instruction.

```
CalMid(float3 *coords, int3 *triangles, float3 *centers)
{
  int i = get_global_id(0);
  int3 idx = triangles[i];
  centers[i] = (coords[ idx.s0 ] + coords[ idx.s1 ] + coords[ idx.s2 ]) / 3;
}
```

2.4 Example 2 : indirect memory access loop with memory dependency issues

2.4.1 Problem

We are provided with a mesh made of NV vertices, NT triangles and we would like to loop over the triangles and modify their vertex coordinates in order to improve their geometrical shape. In such a loop, to different triangles may be processed at the same time and may share a common vertex. Consequently they may write to the same vertex coordinates which will lead to the wrong result due to memory write inconsistency.

To avoid such problem, an indirect memory write loop can be split into two direct write loops, one that writes an intermediary result in a triangle based buffer (also called a scatter loop), and the other one that will loop over vertices and compute the mean value of each vertex surrounding triangles (called a gather loop).

This way, the first one loops over and write data to triangles and the second one loops over and write data to vertices, avoiding any memory race.

The process unfolds this way :

1. initialize the library
2. allocate a vertex data type
3. fill the vertex coordinates to the data type's table
4. transfer the coordinates to the GPU's memory
5. allocate a triangle data type
6. fill the triangle vertex indices in the data type's table
7. transfer the triangles' data to the GPU memory
8. allocate a raw data type to store the temporary coordinates for each triangle's three vertices
9. build the ball of triangles sharing a common vertex
10. upload the ball to the GPU's memory
11. compile the first scatter kernel that loops over triangles
12. launch this kernel with the triangles, vertices and temporary coordinates as arguments
13. compile the first vector part of the gather kernel
14. compile the second extension part of the gather kernel
15. launch the gather kernel with the triangles, vertices and temporary coordinates and the ball as arguments
16. wait for completion and transfer back the optimal coordinates table from the GPU
17. finally, loop over the vertices to update their coordinates with the new ones from the library's buffer

2.4.2 Vectorized vertex balls

This example differs from the first one in two main aspects :

The first one is the splitting of the loop into a pair of scatter/gather loops. Indeed, memory access can be categorized as reading or writing and direct or indirect, which leads to four different combinations :

- direct read : (s = u[i]), are fast because the table is accessed through the loop index and safe because no writing occurs,
- indirect read : (s = u[v[i]]) are slow because table u[] is accessed through an unpredictable index but safe because no writing occurs,
- direct write : (u[i] = s), are fast because the table is accessed through the loop index and safe because two different iterations of the loop cannot write at the same memory location,
- indirect write : (u[v[i]] = s), are slow because table u[] is accessed through an unpredictable index and unsafe because two different iterations of the loop could write at the same memory location.

That is why the original CPU loop that presents indirect reads and writes had to be split into two loops that read indirectly but write directly. There are still slow because of the memory indirection, but safe thanks to direct memory writes.

The second aspect is the vertex ball, that is the list of elements (triangles in our case) that share a common vertex. The problem with this kind of data is that its size is different for each vertex because of the unstructured nature of the mesh. The average value being six, but it may go up sharply in case of a vertex connected to many triangles. And such a case frequently occurs with adapted or anisotropic meshes.

In order to store this kind of information, two tables are needed : one called the ball table that is the concatenation of the lists of all triangles sharing the same vertex, and the other one called the registry that gives for each vertex the position of its own list of triangles in the first table and how many there are (known as the vertex degree). Getting the ball of a vertex consists in accessing the second table to get the position and then jumping to the first table at this given position to read the list of triangles.

Such data structure and algorithm is very compact and efficient on a CPU but has poor performance on a GPU because of the double indirect memory read. Indeed, we first need to read a position in a table, then indirectly access to the list of triangles, before indirectly accessing the triangle's data itself ! Ideally, we should have constant vertex degrees so that we may directly store the ball of each vertex in a fixed size vector. This is the case with structured grids where the vertex degree is always four for 2D quadrilateral meshes and eight for 3D hexahedral ones. That is why such kinds of meshes are very often used to demonstrate the high performance of GPUs. Reaching those performances with unstructured meshes is much more challenging and that is the purpose of this library.

It is indeed possible to get close to the theoretical performance with unstructured data using ball vectorization. The trick is to consider every vertex as having a fixed degree, let's say eight for a triangulated mesh, store the first eight triangles in a vector table and save the eventual remaining ones in an overflow table. If a vertex has a degree lesser than eight,

the empty slots are set with -1. One the other side, if the degree is greater than eight, those extra triangles are stored in a "classical" ball table as used by CPU algorithms. But this indirect, and slow, overflow table would contain only a very limited number of extra elements if the base table's vector size is carefully chosen.

The following table gives some statistics about how many elements fall in the direct access table and the overflow indirect table as well as the wasted space.

element kind	vector	direct table	overflow table	extra memory
edges	16	98,72 %	1,28 %	34,60 %
triangles	8	99,98 %	0,02 %	33,35 %
quadrilaterals	4	99,99 %	0,01 %	0,02 %
tetrahedra	32	99,71 %	0,29 %	46,75 %
hexahedra	8	96,87 %	3,13 %	9,79 %

Consequently, the speed penalty for using an unstructured mesh versus a structured one is always less than 50%.

2.4.3 CPU part of the code

We suppose that the mesh was previously read from a file and that the vertices and triangles are numbered from 0 to $n-1$.

```
#include "sources_opencl.h"
int triangles[nt][3];
float coords[nv][3], optimal[nt][3][3];

GmlInit(GmlGpu);

VerIdx = GmlNewData(GmlVertices, nv, 0, GmlInout);
for(i=0;i<nv;i++)
  GmlSetVertex(VerIdx, i, coords[i][0], coords[i][1], coords[i][2]);
GmlUploadData(VerIdx);

TriIdx = GmlNewData(GmlTriangles, nt, 0, GmlInput);
for(i=0;i<nt;i++)
  GmlSetTriangle(TriIdx, i, triangles[i][0], triangles[i][1], triangles[i][2]);
GmlUploadData(TriIdx);

OptIdx = GmlNewData(GmlRawData, nt, 3 * sizeof(cl_float3), GmlInternal);
BalIdx = GmlNewBall(VerIdx, TriIdx);
PosTri = GmlNewKernel(sources_opencl, "scatter_tri");
VerBal1 = GmlNewKernel(sources_opencl, "gather1");
VerBal2 = GmlNewKernel(sources_opencl, "gather2");
GmlLaunchKernel(PosTri, nt, 3, VerIdx, TriIdx, OptIdx);
GmlLaunchBallKernel(VerBal1, VerBal2, BalIdx, 3, VerIdx, OptIdx);
```

```
DownloadData(MidVerIdx);
for(i=0;i<nv;i++)
  GmlSetVertex(VerIdx, i, &coords[i][0], &coords[i][1], &coords[i][2]);
```

2.4.4 GPU part of the code

This process could have been done in one single loop on a CPU but requires three on a GPU!

Indeed, a first split into a pair of scatter/gather loop is required to get rid of the memory indirect writes. And this second gather loop that needs to access to the vertices balls also requires to be split into two loops, one to deal with the first eight triangles of each vertex ball and a second one to deal with the remaining overflow triangles.

Here is the first kernel that loops over the triangles, computes the optimal positions of their three vertices and store those positions in a temporary table associated with each triangle.

```
scatter_tri(float3 *coords, int3 *triangles, float3 (*optimal)[3])
{
  int i = get_global_id(0);
  int3 idx = triangles[i];
  float3 v0, v1, v2;

  v0 = coords[ idx.s0 ];
  v1 = coords[ idx.s1 ];
  v2 = coords[ idx.s2 ];

  optimal[i][0] = (2*v0 + v1 + v2) / 4;
  optimal[i][1] = (2*v1 + v0 + v2) / 4;
  optimal[i][2] = (2*v2 + v1 + v0) / 4;
}
```

The second kernel loops over vertices reads the number of incident triangles and adds all their optimal positions, and finally computes the average value that will replace the original vertex coordinates. Note that in this example the ball table provides an encoded value that corresponds to the triangle index multiplied by eight plus the local vertex position in this triangle. To separate back the two numbers from the code we need to perform a logical AND with a mask to get the vertex index and divide the code by eight, or shift it to right by three bits, to get the triangle number. Such tricks are very common and powerful on GPUs.

```
gather1(char *degrees, int *balls, float3 *coords, float3 (*optim)[3])
{
  int i, deg;
```

```
  int8 code, TriIdx, VerIdx;
  float4 NewCrd = (float4){0,0,0,0}, NulCrd = (float4){0,0,0,0};

  deg = degrees[i];
  code = balle[i];
  TriIdx = code >> 3;
  VerIdx = code & (int8){7,7,7,7,7,7,7,7};

  NewCrd += (deg >  0) ? optim[ TriIdx.s0 ][ VerIdx.s0 ] : NulCrd;
  NewCrd += (deg >  1) ? optim[ TriIdx.s1 ][ VerIdx.s1 ] : NulCrd;
  NewCrd += (deg >  2) ? optim[ TriIdx.s2 ][ VerIdx.s2 ] : NulCrd;
  NewCrd += (deg >  3) ? optim[ TriIdx.s3 ][ VerIdx.s3 ] : NulCrd;
  NewCrd += (deg >  4) ? optim[ TriIdx.s4 ][ VerIdx.s4 ] : NulCrd;
  NewCrd += (deg >  5) ? optim[ TriIdx.s5 ][ VerIdx.s5 ] : NulCrd;
  NewCrd += (deg >  6) ? optim[ TriIdx.s6 ][ VerIdx.s6 ] : NulCrd;
  NewCrd += (deg >  7) ? optim[ TriIdx.s7 ][ VerIdx.s7 ] : NulCrd;

  coords[i] = NewCrd / (float)deg;
}
```

Finally, the third and most complicated kernel, will loop over some vertices whose degree is greater than eight and complete the rest of the calculation. It continues the calculation where it was left by the previous loop, add contributions from the remaining triangles and finally computes the final average value. Caution : the loop index is not the total number of vertices but the number of vertices whose degree is greater than eight ! Hence, we need to retrieve the global vertex index from the registry table.

```
gather2(int3 *infos, int *balls, float3 *coords, float3 (*optim)[3])
{
  int i, j, deg, VerIdx, code, BalAdr;
  float4 NewCrd;

  VerIdx = infos[i].s0;
  BalAdr = infos[i].s1;
  deg = infos[i].s2;
  NewCrd = VerCrd[ VerIdx ] * (float4){8,8,8,0};

  for(j=BalAdr; j<BalAdr + deg; j++)
  {
    code = balls[j];
    NewCrd += optim[ code >> 3 ][ code & 7 ];
  }
```

```
  coords[ VerIdx ] = NewCrd / (float)(8 + deg);
}
```

3 List of commands

3.1 GmlDownloadData

Syntax

```
error = GmlDownloadData(data);
```

Notes

See GmlUploadData, but on the other way.

3.2 GmlFreeBall

Syntax

```
error = GmlFreeBall(ball);
```

Notes

Free a ball data type as well as its four internal tables.

3.3 GmlFreeData

Syntax

```
error = GmlFreeData(IndexOfData);
```

Notes

Free this data type's information as well as the two buffers in main's and device's memories. Note the number IndexOfData could be reused by any further call to NewData.

3.4 GmlGetArguments

Syntax

```
arguments = GmlGetArguments();
```

Notes

Return a pointer on the *GMlib*'s parameters structure that was returned when the library was first initialized.

3.5 GmlGetEdge

Syntax

```
error = GmlGetEdge(data, line, pi1, pi2);
```

Arguments

Argument	type	description
data	int	index of the GmlEdge data type to read from
line	int	index of the edge to read
pi1,pi2	int	pointers on integers that will be filled with the two edges' vertex indices

3.6 GmlGetHexahedron

Syntax

```
error = GmlGetHexahedron(data, line, pi1, pi2, pi3, pi4, pi5, pi6, pi7, pi8);
```

Arguments

Argument	type	description
data	int	index of the GmlHexahedron data type to read from
line	int	index of the hexahedron to read
pi1...pi8	int	pointers on integers that will be filled with the eight hexahedron's vertex indices

3.7 GmlGetMemoryTransfer

Syntax

```
size = GmlGetMemoryTransfer();
```

Notes

Return the number of bytes transferred to and from the GPU since initialization of the *GMlib*.

3.8 GmlGetQuadrilateral

Syntax

```
error = GmlGetQuadrilateral(data, line, pi1, pi2, pi3, pi4);
```

Arguments

Argument	type	description
data	int	index of the GmlQuadrilateral data type to read from
line	int	index of the quadrilateral to read
pi1...pi4	int	pointers on integers that will be filled with the four quadrilaterals' vertex indices

3.9 GmlGetMemoryUsage

Syntax

```
size = GmlGetMemoryUsage();
```

Notes

Return the total size in bytes of allocated data type's internal buffers on the GPU memory.

3.10 GmlGetRawData

Syntax

```
error = GmlGetRawData(data, LineNumber, buffer);
```

Arguments

Argument	type	description
data	int	index of the data type to fetch from the library's internal buffer
line	int	index of the line of data to copy to the user's local buffer
buffer	void *	pointer to a buffer to which the line of data will be copied from the library's buffer

Return	type	description
error	int	0 for success, 1 otherwise

Notes

See GmlSetRawData.

3.11 GmlGetTetrahedron

Syntax

```
error = GmlGetTetrahedron(data, line, pi1, pi2, pi3, pi4);
```

Arguments

Argument	type	description
data	int	index of the GmlTetrahedron data type to read from
line	int	index of the tetrahedron to read
pi1...pi4	int	pointers on integers that will be filled with the four tetrahedra's vertex indices

3.12 GmlGetTriangle

Syntax

```
error = GmlGetTriangle(data, line, pi1, pi2, pi3);
```

Arguments

Argument	type	description
data	int	index of the GmlTriangle data type to read from
line	int	index of the triangle to read
pi1...pi3	int	pointers on integers that will be filled with the three triangles' vertex indices

3.13 GmlGetVertex

Syntax

```
error = GmlGetVertex(data, line, px, py, pz);
```

Arguments

Argument	type	description
data	int	index of the GmlVertex data type to read from
line	int	index of the vertex to read
px,py,pz	float*	pointers to three floating points values that will be filled with this vertex's coordinates

return	type	description
error	int	0 for success, 1 otherwise

Notes

Make sure the values pointed to are of single precision floating point type as the library does not yet support double precision numbers.

3.14 GmlInit

Syntax

```
LibIndex = GmlInit(DeviceIndex);
```

Description

Initializing the library is mandatory before launching any other *GMlib*'s commands.

It takes a single parameter that is the index of the compute device that will execute all OpenCL kernels. The list of such compute capable devices is provided by the `GmlListGPU` command. Those can be any kind of CPUs, in which case all the cores will be used and data will stay in central memory, or GPUs, is which case the data will automatically be transferred to the graphic card's dedicated memory before executing the code.

The command returns a pointer on a C structure that contains various arguments that will be copied to and from the GPU before and after each kernel execution. This structure is user modifiable and is very useful to transfer small quantities of data without explicitly creating a dedicated data type. It is mainly used for debugging purposes.

3.15 GmlLaunchBallKernel

Syntax

```
time GmlLaunchBallKernel(kernel1, kernel2, ball, arguments, ...);
```

Arguments

Argument	type	description
kernel1	int	index of the first kernel called the "base kernel" to launch on the GPU
kernel2	int	index of the second kernel called the "extension kernel" to launch on the GPU
ball	int	index of a ball data type on which the two kernels will loop over
arguments	int	number of entities to pass on to the kernels
...	int	list of comma separated data type indices

return	type	description
time	double	kernel execution time in seconds or an error code if the value is negative

Notes

A ball data type can only be passed as the first special argument to a `GmlLaunchBallKernel` command and cannot be part of the other arguments list. Likewise, it cannot be passed as a regular `GmlLaunchKernel` command.

3.16 GmlLaunchKernel

Syntax

```
time = GmlLaunchKernel(kernel, count, arguments, ...);
```

Arguments

Argument	type	description
kernel	int	index of the OpenCL kernel to launch on the GPU
count	int	ending loop counter, the kernel will loop from 0 to count-1. On the OpenCL side this is the value returned by the call to `get_global_index(0)`
arguments	int	number of subsequent arguments to pass to the kernel
...	int	list of comma separated data type indices

return	type	description
time	double	kernel execution time in seconds or an error code if the value is negative

Notes

The order of arguments as seen by the kernel procedure will the same as the indices given as arguments. Two additional arguments will be added at the end : the whole GmlParameters structure, to easily transfer small data, and an integer that contains the number of loop iterations to perform.

3.17 GmlListGPU

Syntax

```
GmlListGPU();
```

Description

Prints on the screen the list of OpenCL compatible hardware available on the system. Each line prints the peripheral's index number, to provide when initializing the library, followed by a short description. Here is a sample output on a 2013 Apple MacBook pro :

```
0       : Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz
1       : GeForce GT 650M
2       : HD Graphics 4000
```

The first device is the quad-core Intel CPU itself which can access the global 16 GB of main memory. The second one is the discrete Nvidia GeForce with 384 compute units and 1 GB of dedicated memory. Finally comes the Intel integrated GPU with 64 compute units that share up to 1.75 GB with the CPU's main memory.

3.18 GmlNewBall

Syntax

```
ball = GmlNewBall(VertexType, ElementType);
```

Arguments

Argument	type	description
VertexType	int	index of the GmlVertex data type whose ball of elements is to be built
ElementType	int	index of an element type that points to the vertex data type above

return	type	description
ball	int	index of a ball data type built by the library

Notes

It automatically creates each vertex's ball stored in a vector way.

The ball of a vertex is the list of elements that share this vertex. Contrary to the list of vertices for an element which has a constant size (four for a tetrahedron or eight for a hexahedron), there is an arbitrary number of elements sharing the same vertex in an unstructured mesh.

As GPUs are efficient only when dealing with regularly spaced data types and preferably whose size is a power of 2, one has to find a way to store the ball of a vertex in an OpenCL vector.

This is done via two different tables : a base table whose size is constant and that stores the first 2^n elements of each vertex, and a second irregular, and slow, one that stores the remaining elements for the vertices that have a degree greater than 2^n.

The width of each ball table is calculated depending on the element's type so that the base table is big enough to hold more than 90% of the balls, but not too big since it would spoil the scarce GPU's memory and require additional transfer. The width is 4 four quadrilaterals, 8 for triangles and hexahedra, 16 for 3D edges and 32 for tetrahedra.

Under the hood, a ball data type hides four different tables :

— MainDegree[iVertex] : s table that stores the partial degree of each vertex, that is $max(d_i, 2^n)$, where d_i is the real vertex degree and 2^n is the vector size of the main ball table.

— MainBall[iVertex][iElement] : a table that stores the 2^n first elements of each vertex ball.

— ExtraDegree[xVertex][3] : this table has an entry only for each vertex whose real degree is greater than 2^n. For each of those extra vertices it stores three values : the vertex index in the main table, the address of the list of extra elements in the ExtraBall table and the number of those extra elements ($= d_i - 2^n$).

— ExtraBall_ext[xElement] : a concatenation of all consecutive extra balls of elements. This table should not be accessed directly but via the previous table's information.

3.19 GmlNewData

Syntax

```
index = GmlNewData(type, NumberOfEntities, SizeInBytes, AccessMode);
```

Arguments

Argument	type	description
type	int	tag that specify whether it is a predefined *GMlib*'s data type (GmlVertices, GmlEdges, GmlTriangles, GmlQuads, GmlTetrahedra or GmlHexahedra) or an arbitrary user defined data (GmlRawData)
Number	int	how many entities of this kind should be allocated
Size	int	size to allocate in bytes is to be provided only for raw data (GmlRawData)
Access	int	indicate in which way the data memory could be moved between the CPU's memory and the device's memory. GmlInput : enable only copy from CPU to GPU. GmlOutput : enable only copy from GPU to CPU. GmlInout : both ways are allowed. GmlInternal : no transfer allowed, this data is used as an internal temporary buffer on the GPU.

Return	type	description
index	int	a unique index that should be provided to any kernel that needs to access this data type's memory.

Notes

This function allocates two same-sized buffers : one in the main CPU's memory and the other one in the GPU's memory. The first one is to be initialized with successive calls to the command `GmlSet` and the whole table should be copied to its GPU counterpart afterwards with the help of the `GmlUploadData` command.

After performing some calculations, the reverse process can be used : `GmlDownloadData` to copy the whole table from the GPU's memory to the CPU's image in main memory. Afterwards, the data can be accessed sequentially by the CPU with the `GmlGet` command.

3.20 GmlNewKernel

Syntax

```
kernel = GmlNewKernel(source, procedure);
```

Arguments

Argument	type	description
source	char *	pointer to a character string that holds the OpenCL source code to compile
procedure	char *	pointer to the position of the procedure to compile in the source code since it may contain several procedures

return	type	description
kernel	int	index of the compiled kernel

Notes

The command line utility *cl2h* is provided with the library to help integrate OpenCL sources within your C code without resorting to opening and reading external files at runtime. These command reads a source file *foo.cl* and converts it into a C header file, *foo.h*, that contains a single string initialized with the whole OpenCL source code.

3.21 GmlReduceVector

Syntax

```
time = GmlReduceVector(data, operation, residual);
```

Arguments

Argument	type	description
data	int	index of a raw data type made of one single floating point value per line
operation	int	GmlMin = search for the minimum value, GmlSum = compute the sum, GmlMax = search for the maximum value
residual	double*	pointer to a double precision scalar that will receive the residual value

return	type	description
time	double	kernel execution time in seconds or an error code if the value is negative

Notes

If the data you would like to reduce is more complex than a simple vector, like a 2D table or a table of structures for example, then you need an additional step before calling the reduction. Create a kernel that reads your complex data set, apply a normalizing function that reduces each data into a scalar value and finally writes it as a scalar in simple reduction vector. Afterward, you may reduce this final vector with `GmlReduceVector`.

Example

We would like to compute the residual of a table of 3D vectors ($tab[n][3]$) as the smallest L2 norm of the n vectors. We first allocate a temporary raw data type table $res[n]$. Then we launch a kernel that loops over each 3D vector, computes its L2 norm and stores it in the $res[n]$ table. Finally, a call to `GmlReduceVector` on this table with GmlMin as an operation computes the global residual value.

C part of the code :

```
VecIdx = NewData(GmlRawData, n, sizeof(float3), GmlInternal);
ResIdx = NewData(GmlRawData, n, sizeof(float), GmlOutput);
LaunchKernel(ComputeNorm, n, 2, VecIdx, ResIdx);
GmlReduceVector(ResIdx, GmlMin, &MinNorm);
```

OpenCL kernel :

```
__kernel void ComputeNorm(__global float3 *vec, __global float *res)
{
  int i = get_global_id(0);
  res[i] = length(vec[i]);
}
```

3.22 GmlSetEdge

Syntax

```
error = GmlSetEdge(data, line, i1, i2);
```

Arguments

Argument	type	description
data	int	index of the GmlEdge data type to modify
line	int	index of the edge to modify
i1,i2	int	index of the two vertices this edge is made of

return	type	description
error	int	0 for success, 1 otherwise

Notes

Do not forget that vertex index range from 0 to n-1 not from 1 to n.

3.23 GmlSetHexahedron

Syntax

```
error = GmlSetHexahedron(data, line, i1, i2, i3, i4, i5, i6, i7, i8);
```

Arguments

Argument	type	description
data	int	index of the GmlHexahedron data type to modify
line	int	index of the hexahedron to modify
i1,...,i8	int	index of the eight vertices this hexahedron is made of

return	type	description
error	int	0 for success, 1 otherwise

Notes

Do not forget that vertex index range from 0 to n-1 not from 1 to n.

3.24 GmlSetQuadrilateral

Syntax

```
error GmlSetQuadrilateral(data, line, i1, i2, i3, i4);
```

Arguments

Argument	type	description
data	int	index of the GmlQuadrilateral data type to modify
line	int	index of the quadrilateral to modify
i1,...,i4	int	index of the four vertices this quadrilateral is made of

return	type	description
error	int	0 for success, 1 otherwise

Notes

Do not forget that vertex index range from 0 to n-1 not from 1 to n.

3.25 GmlSetRawData

Syntax

```
error = GmlSetRawData(data, LineNumber, buffer);
```

Arguments

Argument	type	description
data	int	index of the data type to modify
line	int	index of the line of data to modify
buffer	void *	pointer to a table containing the whole line of data that is to be copied to the main memory's buffer

Return	type	description
error	int	0 for success, 1 otherwise

Comment

When filling a table of RawData kind, the easiest way is to use a small local table that contains only one line of data and to reuse it on each call to GmlSetRawData as is illustrated in the following example :

```
struct
{
  int type;
  float quality;
  float normal[3];
}my_data;

for(i=0;i<NmbTriangles;i++)
{
  my_data.type = 1;
  my_data.qualite = qualities[i];
  my_data.normal[0] = normals[i][0];
  my_data.normal[1] = normals[i][1];
  my_data.normal[2] = normals[i][2];
  GmlSetRawData(IdxData, i, &my_data, sizeof(my_data));
}
```

3.26 GmlSetTetrahedron

Syntax

```
error = GmlSetTetrahedron(data, line, i1, i2, i3, i4);
```

Arguments

Argument	type	description
data	int	index of the GmlTetrahedron data type to modify
line	int	index of the tetrahedron to modify
i1,...,i4	int	index of the four vertices this tetrahedron is made of

return	type	description
error	int	0 for success, 1 otherwise

Notes

Do not forget that vertex index range from 0 to n-1 not from 1 to n.

3.27 GmlSetTriangle

Syntax

```
error GmlSetTriangle(data, line, i1, i2, i3);
```

Arguments

Argument	type	description
data	int	index of the GmlTriangle data type to modify
line	int	index of the triangle to modify
i1,i2,i3	int	index of the three vertices this triangle is made of

return	type	description
error	int	0 for success, 1 otherwise

Notes

Do not forget that vertex index range from 0 to n-1 not from 1 to n.

3.28 GmlSetVertex

Syntax

```
error GmlSetVertex(data, line, x, y, z);
```

Arguments

Argument	type	description
data	int	index of the vertex data type to modify
line	int	index of the vertex to modify
x,y,z	float	set the vertex with these coordinates

return	type	description
error	int	0 for success, 1 otherwise

Notes

As for now, the *GMlib* only supports single precision floating-point values.

3.29 GmlStop

Syntax

```
GmlStop();
```

Description

Free all allocated data types and kernels and close the OpenCL session.

3.30 GmlUploadBall

Syntax

```
error = GmlUploadBall(ball);
```

Notes

Upload the four internal tables associated to a ball to the GPU's memory.

3.31 GmlUploadData

Syntax

```
error GmlUploadData(data);
```

Notes

Triggers the copy from the main memory to graphic cards. This operation is extremely slow as it goes through the PCI Express bus whose practical speed is around 2 to 6 GB/s while discrete GPUs with dedicated memory range from 50 to 300 GB/s. That is why those memory transfers should be limited to the strict minimum, otherwise they may spoil the whole GPU speedup.

4 Glossary

4.1 API

Application programming interface, it is the list arguments to provide to each procedure of a program or a library.

4.2 Compute Unit

A GPU chip contains a great number of so called "compute units" that work in parallel pretty much in the same way as CPU cores. Manufacturers tend to overstate their compute power by multiplying the number of computing units by the size of the vector each unit can handle. To put things in perspective, a four-core Intel Core i7-3770, as each core is capable of handling 8 element-wide vectors in one single cycle should be considered as a $4 * 8 = 32$ compute units device in GPU marketing language.

It is also important to note that GPU cores are much simpler than their CPU counterparts and the number of instructions they can execute per cycle (IPC) is typically three or four times lower than that of CPUs. Hence the brute comparison of core gigahertz (a chip's total number of computing units multiplied by its frequency) between CPU and GPU is unrealistic.

As an example, my laptop's CPU with it four cores, handling 8-wide vectors running at 3 GHz ($4 * 8 * 3 = 96$ GHz-Core), is almost as fast as its GPU with 384 compute units running at 1.15 GHz ($384 * 1.15 = 441$ GHz-Core).

4.3 GPU

Graphic Processing Unit, also called GPGPU (for General Purpose), is a kind of processor well suited for geometric and vector calculations. As its name does not imply, it is far from being "general purpose" ! This kind of chip can be found in all common discrete graphic cards from AMD and NVIDIA, but a lot of GPUs are now embedded inside CPU chips for cost, size and power reduction, like in all smartphones and tablets (ARM Mali and Imagination Technologies SGX) or gaming consoles (the Sony PS4 is made of a single AMD chip that contains 8 x86 CPU cores and 1152 Radeon GPU compute units).

4.4 Kernel

Name given to a loop executed in parallel by a GPU. It is called kernel because the program does not write the loop controlling part of the code, as it is handled by the GPU itself, but only the part that lies inside the DO/LOOP statements, hence the name kernel.

4.5 OpenCL

Open Compute Language, it is a language derived from ANSI C with some borrowings from C++. It has been conceived to be independent from any underlying hardware so that an OpenCL code can be executed efficiently on any kind of architecture, either CPU, GPU, FPGA or any concurrent computing device. Its objective is to expose the inherent concurrency clearly of calculation and data in order to better distribute them across all compute units.

4.6 Workgroup

It is a subset of a loop's range that will really be processed concurrently by several GPU compute units. That is why one should never assume that iteration i of a loop will be executed before iteration $i+1$ since they may belong to the same workgroup (typically 64 entries) and thus will be executed exactly at the same time.

Références

[1] SAGAN, Space-Filling Curves, *Springer Verlag, New York, 1994.*

[2] Aaftab Munshi, The OpenCL Specification, *Khronos Group, http ://www.khronos.org/opencl/, 2012.*

[3] NVIDIA Corporation, OpenCL Optimization, *NVIDIA Corporation, NVIDIA_GPU_Computing_Webinars_Best_Practices_For_OpenCL_Programming.pdf, 2009.*

[4] Andrew Corrigan & Rainald Löhner, Porting of FEFLO to Multi-GPU Clusters, *49th AIAA Aerospace Sciences Meeting, Orlando, January 2011.*

[5] Apple Corp, OpenCL Programming Guide for Mac, *http ://developer.apple.com, 2012.*