GMlib v3

A framework to develop numerical simulation software on GPUs

Loïc Maréchal / INRIA, November 2023

Context and motivation

- CPUs sequential and multicore speed is staling
- The only way to keep up increasing the speed: specialized compute units (GPU, FPGA, vector coprocessors,...)
- Simplicity, efficiency and general purpose: you cannot have them all !
- GMlib: easy and efficient, but dedicated to mesh data structures (hybrid and unstructured)
- Based on OpenCL, an open and multiplatform library (CPU and GPU, AMD and NVIDIA, Linux, macOS and Windows)
- It handles the most complex tasks on a GPU: vectorized and efficient storage and memory access to unstructured data
- Programming is straightforward: very close to ANSI C or Fortran77

Using GPU shaders in a meshing context

How to repurpose a well-known and tried graphic technic

- All data types, their storage and transfer are taken care of by the GMlib, you cannot use your own data structures
- All meshing data types are supported: vertices, triangles, quads, tets, pyramids, prisms and hexes
- Any kind of solution field can be associated with a mesh data type: scalar, vector or tables made of integers, single or double precision types
- All possible topological links between any pair of mesh entities are automatically deducted and generated by the library: neighbours, shells, balls, ...
- A KERNEL loops over a single kind of element and can access indirectly to any other mesh entity or solution field requested by the programer

How does it work ?

- GmlInit(GPU_device_index)
- GmlImportMesh("tets.meshb", GmfVertices, GmfTetrahedra)
- TetQal = GmlNewSolutionData(GmlTetrahedra, 1, GmlFlt, "quality")
- krn = GmlCompileKernel("get_quality", GmlTetrahedra, 2, VerIdx, GmlReadMode, NULL, TetQal, GmlWriteMode, NULL)
- GmlLaunchKernel(krn)
- GmlExportSolution("qualities.solb", TetQal)

Challenges

A program that generates another program…

- The loop is handled by the GMlib, you only write the inner (kernel) part
- You can only access the data related to the item index being processed
- All variables are defined and filled automatically
- This is the same for memory writes that are handled by the library
- The same mesh entity changes name depending on the loop's context
- Memory can be read directly or indirectly but writing can only be direct
- You need to split indirect memory writes into a pair of scatter / gather loops

Writing and compiling a kernel

- Define your mesh data, and solution fields associated, by choosing the right kind (scalar, vector, integer, single, double,etc.)
- Initialize the GMlib's data structures with your own data (or import them from a file)
- From the main C code, you need to provide the loop's OpenCL source code to compile, the mesh kind to loop over, and the list of data structures to be read or written by the loop
- In each kernel, the names of variables are defined based on the storage type, the loop main entity and the entity being accessed: for example, a loop over triangles that needs to read the vertices coordinates will trigger the following definition, **float3 TriCrd[3]**, because a triangle (Tri) is made of three sets of coordinates (Crd[3]) that are stored in a 3D vector (float3)

Advantages

Programing is a lazy guy's job

- All I/O are performed with a single function call to ImportMesh() or ExportSolution()
- The whole complexity related to topological aspects of unstructured meshes is handled by the library: Hilbert renumbering, balls of points, shells of edges, hashing, neighbours, etc.
- On the GPU side, many geometrical functions written in OpenCL are provided: area, distance, volume, element quality, projections, intersections, etc.
- The GMlib is working on many different architectures: CPUs (it uses the multithreading and vector acceleration), integrated GPUs (Intel HD graphics) and discrete GPUs from smartphones, tablets, PC and Mac running macOS, Linux or Windows
- Code and data are automatically vectorized: this is the most complex aspect of GPUs

```c
int main(){
   GmlIdx = GmlInit(GpuIdx);

   GmlImportMesh(GmlIdx, "tetrahedra.meshb", GmfVertices, GmfTetrahedra, 0);
   GetMeshInfo(GmlIdx, GmlVertices,   &NmbVer, &VerIdx);
   GetMeshInfo(GmlIdx, GmlTetrahedra, &NmbTet, &TetIdx);
   QalIdx = GmlNewSolutionData(GmlIdx, GmlTetrahedra, 1, GmlFlt, "qal");

   QalKrn = GmlCompileKernel( GmlIdx, mesh_quality, "mesh_quality", GmlTetrahedra, 2,
                              VerIdx, GmlReadMode,  NULL, QalIdx, GmlWriteMode, NULL );

   GmlLaunchKernel(GmlIdx, QalKrn);
   GmlReduceVector(GmlIdx, QalIdx, GmlSum, &AvgQal);
   GmlReduceVector(GmlIdx, QalIdx, GmlMin, &MinQal);

   QalTim = GmlGetKernelRunTime(GmlIdx, QalKrn);
   AvgTim = GmlGetReduceRunTime(GmlIdx, GmlSum);
   MinTim = GmlGetReduceRunTime(GmlIdx, GmlMin);

   GmlFreeData(GmlIdx, VerIdx);
   GmlFreeData(GmlIdx, TetIdx);
   GmlFreeData(GmlIdx, QalIdx);
   GmlStop(GmlIdx);
}
```

```
float len, srf, vol;

len =               dot(VerCrd[0] - VerCrd[1], VerCrd[0] - VerCrd[1]);
len = max(len, dot(VerCrd[0] - VerCrd[2], VerCrd[0] - VerCrd[2]));
len = max(len, dot(VerCrd[0] - VerCrd[3], VerCrd[0] - VerCrd[3]));
len = max(len, dot(VerCrd[1] - VerCrd[2], VerCrd[1] - VerCrd[2]));
len = max(len, dot(VerCrd[1] - VerCrd[3], VerCrd[1] - VerCrd[3]));
len = max(len, dot(VerCrd[2] - VerCrd[3], VerCrd[2] - VerCrd[3]));

srf = CalTriSrf(VerCrd[0], VerCrd[1], VerCrd[2])
    + CalTriSrf(VerCrd[0], VerCrd[1], VerCrd[3]);
    + CalTriSrf(VerCrd[1], VerCrd[2], VerCrd[3]);
    + CalTriSrf(VerCrd[2], VerCrd[0], VerCrd[3]);

vol = CalTetVol(VerCrd[0], VerCrd[1], VerCrd[2], VerCrd[3]);

qal = 7.348469 * vol / (srf * sqrt(len));
```

```
~/code/inria/GMlib/examples>./MQ_mesh_quality

MeshQuality GPU_index
 Choose GPU_index from the following list:
      0       : Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
      1       : Intel(R) HD Graphics 530
      2       : AMD Radeon Pro 460 Compute Engine
~/code/inria/GMlib/examples>
~/code/inria/GMlib/examples>
~/code/inria/GMlib/examples>./MQ_mesh_quality 0
Imported 202875 vertices and 1148326 tets from the mesh file
1148326 tets processed in 0.0120309 seconds, quality=0.00660762 s, min=0.00299225 s, mean=0.002431 s
34 MB used, 60 MB transfered
QalTet checksum = min quality=0.015592, mean quality=0.689101
~/code/inria/GMlib/examples>
~/code/inria/GMlib/examples>
~/code/inria/GMlib/examples>./MQ_mesh_quality 1
Imported 202875 vertices and 1148326 tets from the mesh file
1148326 tets processed in 0.00871185 seconds, quality=0.00411622 s, min=0.00254926 s, mean=0.00204637 s
34 MB used, 51 MB transfered
QalTet checksum = min quality=0.015592, mean quality=0.688992
~/code/inria/GMlib/examples>
~/code/inria/GMlib/examples>
~/code/inria/GMlib/examples>./MQ_mesh_quality 2
Imported 202875 vertices and 1148326 tets from the mesh file
1148326 tets processed in 0.00205216 seconds, quality=0.00132288 s, min=0.00038464 s, mean=0.00034464 s
34 MB used, 51 MB transfered
QalTet checksum = min quality=0.015592, mean quality=0.688992
```

Heat Cell Centered

A basic first order solver developed par Julien Vanharen

- Solves the heat equation
- Dirichlet and Neumann boundary conditions based on triangle references that are automatically sorted and grouped to avoid the GPU's severe dynamic branching penalty
- Scatter-gather pair: it first loops over the triangles while accessing the two neighbouring tets (uplink) to compute the temperature gradient, then loops over the tets to assemble the triangles' contributions (downlink)
- Advances the time step
- Computes a residual value by reducing the flux gradient vector (GmlReduce)

Runtime statistics

- The mesh is made of 200.000 nodes, 33.000 boundary triangles and 1.150.000 tetrahedra
- The source code is made of 230 lines of C and 52 lines of OpenCL
- Library overhead is 1.8s: mesh reading, extraction of inner faces, building the neighbourhoud relations between triangles and tets, kernels compiling, transferring all data to the GPU, getting back and writing the data to the disk
- 11,000 iterations needed to converge the residual down to 1E-6 -> 10 billion tetra-iterations !
- Laptop 4-core i7 @2.6 GHz : 360s and Radeon Pro 460, 1024 units @0.9 GHz: 65s
- Desktop 16-core Xeon @3.2 GHz : 111s and Radeon Pro Vega II, 4096 units @1.7 GHz: 8.8s
- The speed ratio between the CPU (all cores) and the GPU does not change much regarding the machine, laptop, desktop or server, because the components' speed is growing with the host machine electric power

Future developments

- Distribute the workload across multiple GPUs: this capacity is needed because of memory constrains rather than lack of speed. A GPU is an order of magnitude faster than a CPU but its memory is one or two orders of magnitude smaller!
- The distribution of calculations and the data exchange could be handled transparently by the library, thanks to the GMlib complete management of all data
- We need a comprehensive distributed framework: collaboration INRIA/ONERA
- We also need more use cases: as for now, only Wolf (CFD), Hexotic (node smoothing), HeatCellCentered, a high-order mesh quality evaluation code and a few other small demonstrators use the GMlib
- Offer a complete geometrical and topological toolkit: the current V3 features only the minimum needed to support the codes aforementioned

Conclusion

- Today there is no more "on size fits all" to keep up with the Moore's law, the programer needs a toolbox covering all available technologies: CPU, multithreading, MPI, GPU, FPGA, vectors, and he needs to know which to use in the right situation!
- The GPUs offer a better perspective in terms of scaling compared to the multicore CPUs that are stalling because of the memory access bottleneck (CPU with HBM ?)
- Abstracting away the data structures in a parallel framework is quite a hindrance from the programer's point of view but allows for much higher performance scalability and portability across different hardware
- The Jean ZAY supercomputer with its 1024 GPUs is waiting for us ;-)