ERO2.0

Xavier Sáez

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

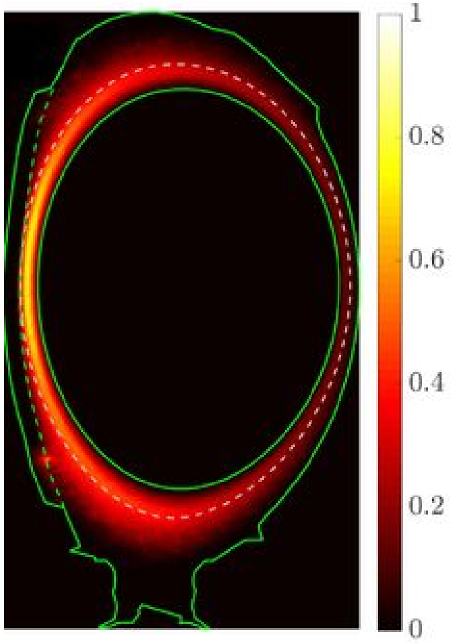
31/08/2023

People

- ~~In January and February, two new people arrived on our staff to increase our dedication to all ACH tasks, which includes ero2.~~
- In January, Federico Cipolletta joined or group → JOREK
- In August, Augusto Maidana has joined our group → ERO2.0

ERO2.0

- ERO2.0 is a code for modelling plasma-wall interaction and global material migration in fusion devices.
- The migration is simulated by following 3D trajectories of Monte-Carlo test particles.
- The 3D gyro-orbits are resolved instead of applying the guiding-center approximation.
- ERO2.0 is parallelized using MPI/OpenMP.
- Goal: porting to GPU



Parallelization with CUDA

- **The Octree construction** is carried out on the host CPU.
- It is hard to run efficiently on GPU due to their hierarchical and recursive nature.
- **Preparatory tasks:**
  - **Translate the recursive tree structure to a "flattened" octree to a linear structure**, where nodes and their children are stored in arrays. Technically, it allows more predictable traversals and is better suited for coalesced memory access on GPUs.
  - **Convert the recursive octree search to an iterative process**.
- Next, we can parallelize querying using CUDA.

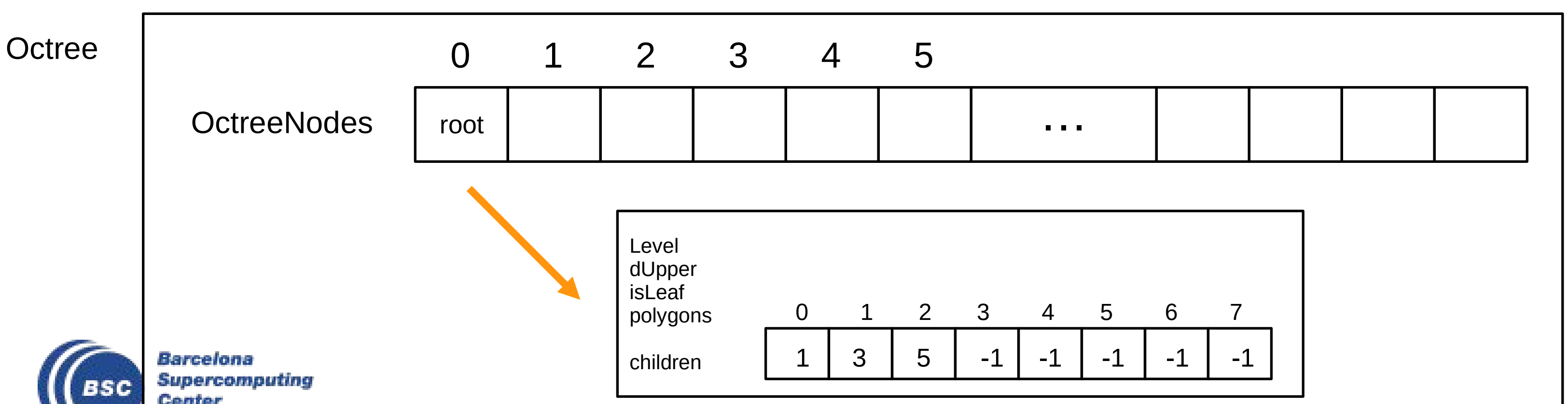
“Flattened” octree

OctreeNode.h

```
class Octree;

/*
 *
 */
class OctreeNode: public CartesianBox
{
public:
        OctreeNode (Octree* octree);
        OctreeNode (Octree* octree, const AxisAlignedBox& box);

        void subdivide (
                const std::vector<Polygon*>& polygons,
                size_t maxPolygons,
                size_t maxLevel,
                size_t level=0
        );

        void getDistance (
                const Vector& p,
                Vector& proj,
                double& dMin,
                const Polygon*& poly
        ) const;

        ...

        Octree* tree;
        size_t level = 0;
        double dUpper = std::numeric_limits<double>::max();
        int children[8];
        bool isLeaf;
        std::vector<Polygon*> polygons;
};
```

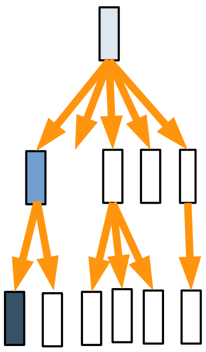
Octree.h

```
class Octree
{

friend class OctreeNode;

public:
        Octree ();
        Octree (const AxisAlignedBox& box);
        ~Octree();
        void subdivide (
                const std::vector<Polygon*>& polygons,
                size_t maxPolygons,
                size_t maxLevel,
                size_t level=0
        );
        void getDistance (
                const Vector& p,
                Vector& proj,
                double& dMin,
                const Polygon*& poly
        ) const;

        ...

        const OctreeNode* getLeaf (const Vector& p) const;
        size_t size () const;
        size_t getSubIndex (const Vector& p) const;

        Octree& operator= (const Octree& oct);

        std::vector<OctreeNode> octreeNodes;
};
```

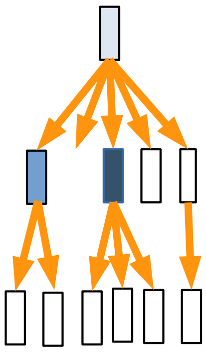


Iterative Octree Traverse

- The current traversal of an octree is a **Depth-First Search (DFS)** => one explores as deeply as possible along a branch before backtracking
- To take advantage of the massively parallel nature of GPUs we will use **Breadth-First Search (BFS)** => all nodes at a given depth are visited before visiting the nodes at the next depth.



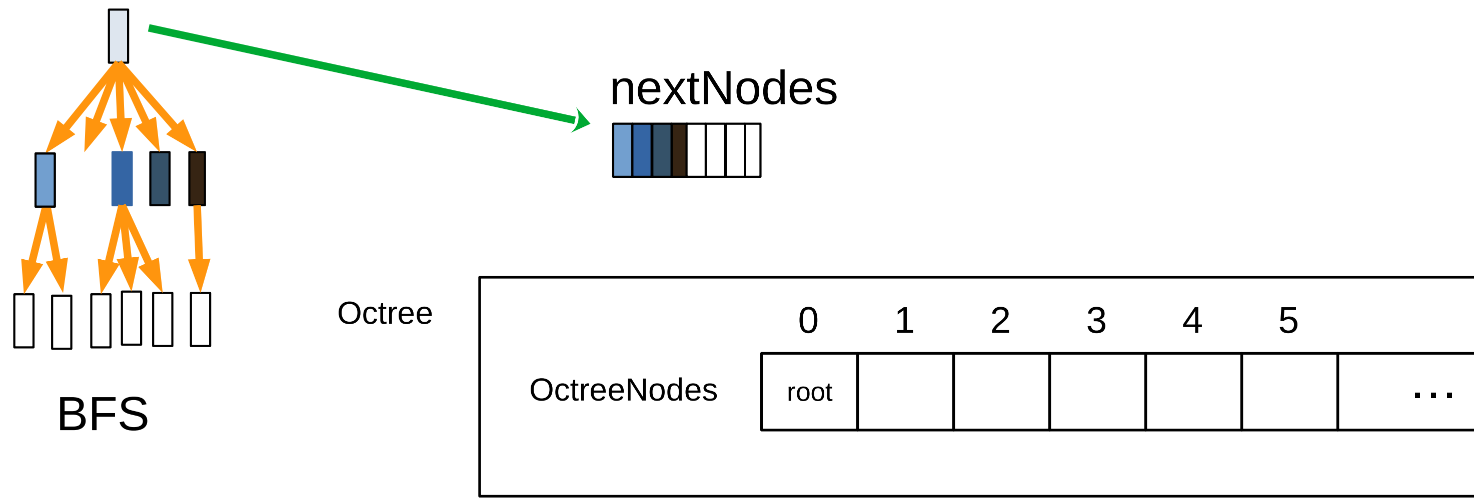
DFS



BFS

Octree Breadth-First Search (BFS)

- use a two-array approach: one list for the current set of nodes to be processed and another for the next set of nodes. After each iteration, the roles of the two lists are swapped.
- If a node contains the polygon, we push its child nodes onto the next nodes to process for further exploration.



Octree Breadth-First Search (BFS)

```cpp
void Octree::getDistance (
        const Vector& p,
        Vector& proj,
        double& dMin,
        const Polygon*& poly
) const
{
        std::vector<int> idNodes;
        std::vector<int> nextIdNodes;

        idNodes.push_back(0); //root

        std::set<std::pair<size_t,size_t>> checkedPolyIds;
        double dMinSq = dMin*dMin;

        while (!idNodes.empty())
        {
                for (auto id : idNodes)
                {
                        const OctreeNode& node = octreeNodes[id];

                        double dBoxSq = node.getDistanceBox (p);
                        if (!(dBoxSq>dMinSq))
                        {
                                if (node.isLeaf)
                                {
                                        for (const auto& polygon : node.polygons)
                                        {
                                                std::pair<size_t,size_t> key = std::make_pair(polygon->face_id, polygon->mesh_id);
                                                if (checkedPolyIds.find(key) == checkedPolyIds.end())
                                                        checkedPolyIds.insert(key);
                                                else
                                                        continue;

                                                double dBoxSq = polygon->bbox.getDistanceSq (p);
                                                if (dBoxSq > dMinSq)
                                                        continue;

                                                double dTmpSq = polygon->getDistanceSq(p);
                                                if (dTmpSq <= dMinSq)
                                                {
                                                        dMinSq = dTmpSq;
                                                        poly = polygon;
                                                }
                                        }
                                }
                                else
                                {
                                        for (const auto& childId : node.children)
                                        {
                                                if (childId != -1)
                                                        nextIdNodes.push_back(childId);
                                        }
                                }
                        }
                }

                idNodes.swap(nextIdNodes);
                nextIdNodes.resize(0);
        }

        if (poly)
        {
                dMin = poly->getDistance (p, proj);
        }
}
```

CUDA

- **std::vecto**r is a C++ data structure can not be used on CUDA. Additionally, std::vector has dynamic operations (eg: memory allocation and deallocation) that aren't GPU-compatible
- **Thrust** is a parallelism library similar to C++'s STL, but not recommended inside CUDA kernels due to performance considerations and the lack of support for certain operations
- **Solution**: use std::vector on the host part (CPU) and convert it *manually* to simple arrays on the GPU

```
int idNodes[MAX_NODES];
int nextIdNodes[MAX_NODES];
int nNodes = 1;
int nNextNodes = 0;

idNodes[0] = 0; //root

double dMinSq = dMin*dMin;

while (nNodes > 0)
{
        for (int i = 0; i < nNodes; i++)
        {
                const OctreeNode& node = octreeNodes[idNodes[i]];

                double dBoxSq = node.getDistanceBox (p);
                if (!(dBoxSq>dMinSq))
                {
                        if (node.isLeaf)
                        {
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Implementation

- **Memory Management**

```
#include <g3d/OctreeGPU.h>
...
void Octree::initGPUData()
{
        // Allocate GPU Memory
        cudaMalloc(&d_octreeNodes, octreeNodes.size() * sizeof(OctreeNodeDevice));

        // Copy data to GPU Memora
        OctreeNodeDevice octreeNodesDevice[octreeNodes.size()];
        for (size_t i = 0; i < octreeNodes.size(); ++i)
        {
                octreeNodesDevice[i].level = octreeNodes[i].level;
                octreeNodesDevice[i].dUpper = octreeNodes[i].dUpper;
                octreeNodesDevice[i].isLeaf = octreeNodes[i].isLeaf;
                ...
        }

        cudaMemcpy(d_octreeNodes, octreeNodesDevice, sizeof(OctreeNodeDevice) * octreeNodes.size(), cudaMemcpyHostToDevice);
}
```

Octree.cpp

```
#ifndef G3D_OCTREE_CUDA_H_
#define G3D_OCTREE_CUDA_H_

typedef struct
{
        size_t level;
        double dUpper;
        int    children[8];
        bool   isLeaf;
        const  Polygon** polygons;
        int    nPolygons;
} OctreeNodeDevice;


void launch_getDistance (double* d_dMin, int totalNodes);

#endif
```

OctreeGPU.h

- **Kernel Definition & Launch**

```
#include <g3d/OctreeGPU.h>

__global__ void getDistanceKernel (OctreeNodeDevice* node, double* dMinSq, int totalNodes)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx >= totalNodes) return;

    const OctreeNodeDevice& node = d_octreeNodes[idx];

    ....

}

void launch_getDistance (double* d_dMinSq, int totalNodes)
{
        // Launch the kernel
        int threadsPerBlock = 256;
        int numBlocks = (totalNodes + threadsPerBlock - 1) / threadsPerBlock;
        getDistanceKernel<<<numBlocks, threadsPerBlock>>>(d_dMinSq, totalNodes);
}
```

OctreeGPU.cu

Implementation

- **Memory Transfer**

```
void Octree::getDistance (
        const Vector& p,
        Vector& proj,
        double& dMin,
        const Polygon*& poly
) const
{

        double dMinSq = dMin*dMin;

        // Allocate GPU memory
        double* d_dMinSq;
        Polygon** d_poly;

        // Allocate device memory
        cudaMalloc((void**)&d_dMinSq, sizeof(double));
        cudaMalloc((void**)&d_poly, sizeof(Polygon*));

        // Copy data to device
        cudaMemcpy(d_dMinSq, &dMinSq, sizeof(double), cudaMemcpyHostToDevice);

        // Launch the kernel
        launch_getDistance (d_dMinSq, 512);

        // Copy results back to host
        cudaMemcpy(&dMinSq, d_dMinSq, sizeof(double), cudaMemcpyDeviceToHost);
        cudaMemcpy(&poly, d_poly, sizeof(Polygon*), cudaMemcpyDeviceToHost);

        // Free GPU memory
        cudaFree(d_dMinSq);
        cudaFree(d_poly);
}
```

Summary

- Flatten ‘octreeNode’ vector
- Iterative Breadth-First Search (BFS) implemented (also in OpenACC)
- Partial CUDA implementation, where each octree node is a structure with multiple fields (AoS). It is simpler to manage but it could be inefficeint for memory access patterns.

Next steps

- Flatten the ‘polygon’ vector
- Finish CUDA implementation & evaluate
- Evaluate SoA for octree node.
- Augusto Maidana will continue CUDA work.