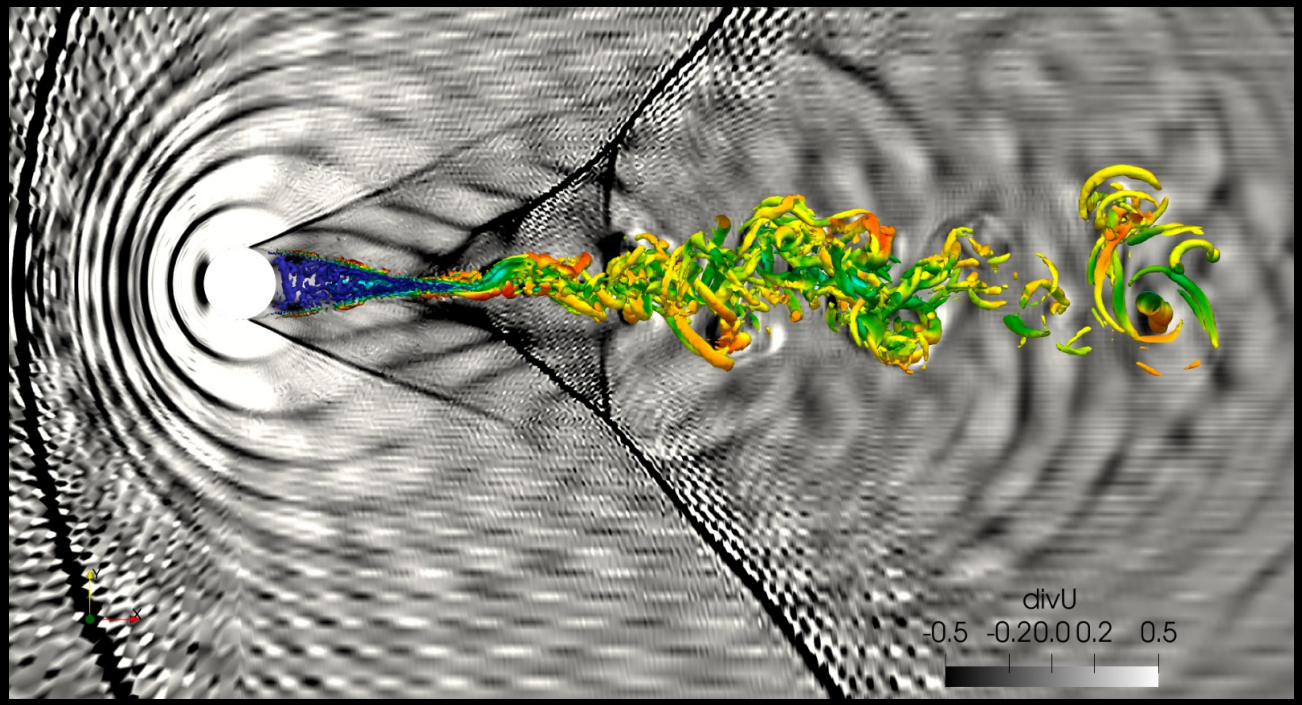
Introduction to GPU programming

Lucas Gasparino

Objectives

- Overview of GPU architecture
  - What IS a GPU
  - SM structure and execution model
  - Memory layout
- Accessing the hardware
  - Developer’s POV: grids and threads
  - Programming models
  - Overview of CUDA
  - OpenACC basics
- Example 1: matrix multiplication
- Example 2: FEM1D

Motivating example



What IS a GPU

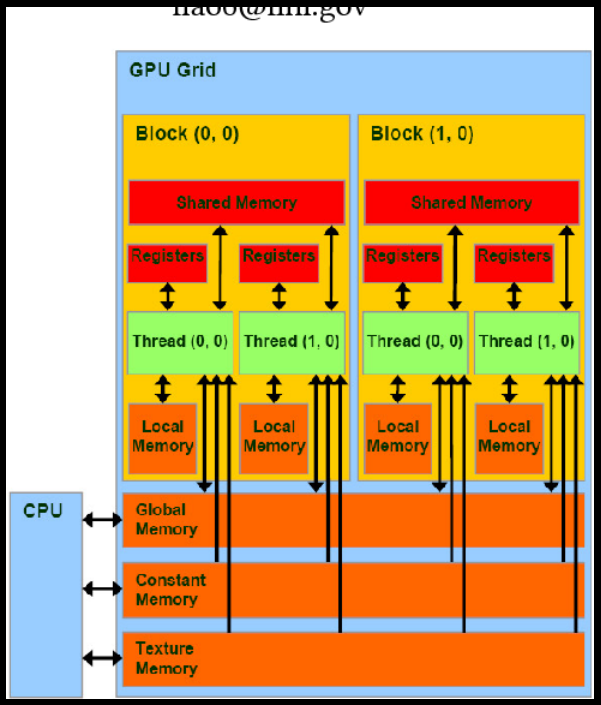
- GPGPU: General Purpose Graphics Processing Unit;
- Massively parallel hardware, created to handle millions of pixels simultaneously;
- On HPC: expected to be more efficient than multicore CPU architectures for certain types of computations;

The computing guts: SMs

- GPU “core” composed of many SMs (CUDA cores for NVIDIA);
- Each SM contains a multitude of threads;
- Instructions in each cycle are issued to “warps” containing 32 threads each, per SM;

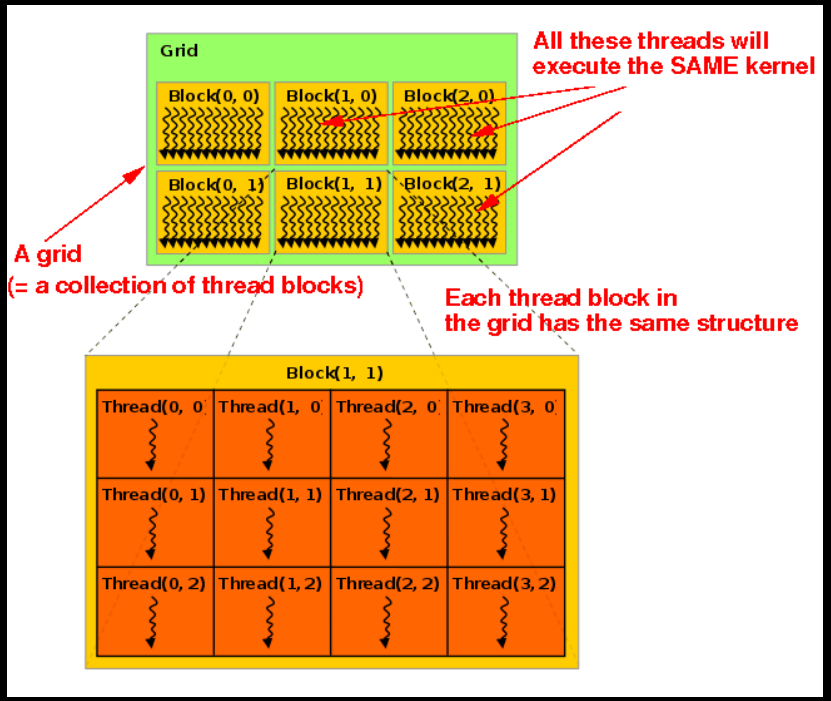
Memory layout

- GPU memory is detached from system memory;
- Multiple caches and registers that allow better data access rates;
- Hierarchy:
  - Global memory
    - Shared memory
    - L1 cache
    - Constant memory
    - Texture memory
      - Thread register



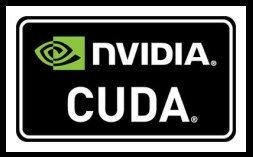
Developer's POV: grids and threads

- GPU kernel execution must happen in a *grid*, composed of *thread blocks*;
- Thread blocks are groupings of threads that access the same region of data;
- Developer's task: ensure that the algorithm can be efficiently parallelized using this structure;



GPU programming models

- Low level models:
  - CUDA, OpenCL, NVPTX, SYCL
- Pragma-based high level models:
  - OpenACC, OpenMP
- Graphics engines:
  - Unreal, DirectX, Vulkan, OpenGL
- Language standard parallelization:
  - Stdpar for Fortran and C/C++
- Library-based:
  - PyTorch, Thrust



CUDA API

- Excellent low-level control capabilities:
  - Micromanaged usage of threads and thread blocks;
  - Allow access to all memory caches in the GPU;
  - Allows configuration of streams to asynchronously execute data movements and kernel executions;
  - Offer peer-to-peer GPU communication tools;
- Hard to learn, hard to master:
  - Data transfer micromanagement can lead to issues;
  - Using hierarchical memory is not simple;
  - Data partition within the grid depends on developer skill;

CUDA C: Host/Device memory ops

- Basic device memory ops:
  - cudaMalloc: allocates memory on global device memory pool;
  - cudaMemcpy: copies a host or device array into it's counterpart;

```
// Define array size
const int n = 512*512;

// Create host arrays
float *a = (float*)malloc(n*sizeof(float));
float *b = (float*)malloc(n*sizeof(float));
float *c = (float*)malloc(n*sizeof(float));

// Fill host arrays
//...

// Create device arrays
float *d_a, *d_b, *d_c;

// Use cudaMalloc to allocate device arrays
cudaMalloc((void**)&d_a, n*sizeof(float));
cudaMalloc((void**)&d_b, n*sizeof(float));
cudaMalloc((void**)&d_c, n*sizeof(float));

// Copy host arrays to device arrays
cudaMemcpy(d_a, a, n*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, n*sizeof(float), cudaMemcpyHostToDevice);

// Execute kernel
vecAdd<<<n/256, 256>>>(n, d_a, d_b, d_c);

// Copy device arrays to host arrays
cudaMemcpy(c, d_c, n*sizeof(float), cudaMemcpyDeviceToHost);
return 0;
```

CUDA C: grid configuration

- Kernels must be launched on a grid of “thread blocks”;
- “dim3” type variables can be created to describe the configuration of the set of thread blocks, and the thread block sizes itself;
- Both the grid and the thread blocks can be 3 dimensional. Hardware sets limits n max. sizes;

```
//...

// Configure the thread block and the grid of thread blocks
dim3 block(16,16,1)
dim3 grid (256,256,1)

// Execute kernel
matrixAdd<<<grid, block>>>(...);

//...

return 0;
```

CUDA C: kernel structure

- CUDA kernels must have an attribute descriptor. Basic ones are:
  - Global: device code launched by host code;
  - Device: device code launched from device code;
- Indexes to access data structures are build using a combination of the following structs:
  - blockDim: size of thread block;
  - blockIdx: index of thread block on grid;
  - threadIdx: index of thread within the block;
- All these include x, y and z fields to account for their three-dimensionality;

```
#include <cuda.h>

__global__ kernel(...)
{
    // Configure how data is distributed to the grid and thread blocks
    // using blockIdx, blockDim and threadIdx variables. Remember that
    // those are structures containing .x, .y and .z fields.
    int i = f(blockIdx.x,blockDim.x,threadIdx.x)

    // Your kernel here
}
```

Example: Vector Addition

- Host kernel

```
void cpuVecAdd(int n, float *a, float *b, float *c) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
```

- GPU kernel

```
__global__ void gpuVecAdd(int n, float *a, float *b, float *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
```

OpenACC

- High level alternative to CUDA;
- Directive-based approach;
- Provides both “automagic” and user-controlled options;
- Good balance between efficiency and ease of use;
- Usage: “#pragma acc [directive] [options]” over section of code to be parallelized. Fortran pragma is “!$”;

OpenACC: data movement

- Structured data regions: data remains on device until the end of the scope. Allocates data if 1st time usage;
- Unstructured data regions: one-time transfers between host and device. More flexibility on usage, data will persist on device until manually destroyed;

OpenACC: data movement snips

```
// Arrays a, b and c are created on the device;
// a and b are copied to device before k=0 loop;
// c is copied back to host after k=99 loop.
// Data is destroyed on the device after the loop.
#pragma acc data copyin(a[0:n], b[0:n]) copyout(c[0:n])
{
    for (int k = 0; k < 100; k++)
    {
        #pragma acc parallel loop
        for (int i = 0; i < n; i++) {
            c[i] = a[i] + b[i];
        }
    }
}
```

```
// Arrays a, b and c are created on the device at k=0 loop;
// a and b are copied to device before every k loop;
// c is copied back to host after every k loop;
for (int k = 0; k < 100; k++)
{
    #pragma acc data copyin(a[0:n], b[0:n]) copyout(c[0:n])
    {
        #pragma acc parallel loop
        for (int i = 0; i < n; i++) {
            c[i] = a[i] + b[i];
        }
    }
}
```

```
// Arrays a, b and c are created on the device;
// a and b are copied to device before k=0 loop;
// c is copied back to host after k=99 loop;
// Data is destroyed on the device at the exit data clause;
#pragma acc enter data copyin(a[0:n], b[0:n]) create(c[0:n])
for (int k = 0; k < 100; k++)
{
    #pragma acc parallel loop
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
#pragma acc exit data copyout(c[0:n])
```

OpenACC: managed memory

- When “-gpu=managed” is used to compile, the code can make use of Nvidia’s Unified Virtual Memory (UVM) space;
- Now, compiler both generates automatic data transfers and tracks whether data is already present on the device (no need for manual transfer);
- Completely safe for pointers and values, not usable with objects!

OpenACC: automagic

- OpenACC provides the “kernels” directive for compute constructs;
- Instructs the compiler to generate a gpu-capable, “thread-safe” code;
- Good starting point, but rarely optimal;

```
// Basic FEM operations to compute a 1d convective term;
// Multiple nested loops can benefit from multi-level parallelism;
// Kernels parallelism provides a good starting point for complex operations;
#pragma acc kernels
{
    // Elemental loop
    for (int ielem = 0; ielem < nelem; ielem++)
    {
        // Form local arrays to each element
        for (int inode = 0; inode < nnode; inode++)
        {
            uloc[inode] = u[connec[ielem][inode]];
            Rloc[inode] = 0.0f;
        }
        // Gaussian integration loop
        for (int igaus = 0; igaus < ngaus; igaus++)
        {
            // grad(u) at Gauss point
            aux = 0.0f
            for (int inode = 0; inode < nnode; inode++)
            {
                aux += dNgp[igaus][inode]*uloc[inode];
            }
            // Update element residual
            for (int inode = 0; inode < nnode; inode++)
            {
                Rloc[inode] += gpvol[ielem][igaus]*Ngp[igaus][inode]*aux;
            }
        }
        // Update global residual
        for (int inode = 0; inode < nnode; inode++)
        {
            R[connec[ielem][inode]] += Rloc[inode];
        }
    }
}
```

OpenACC: kernel example

```
void matrixVectorProduct(int n, float *M, float *u, float *v)
{
    #pragma acc kernels
    {
        for (int i = 0; i < n; i++)
        {
            v[i] = 0.0f;
            for (int j = 0; j < n; j++)
            {
                v[i] += M[i*n+j]*u[j];
            }
        }
    }
}
```

```
matrixVectorProduct:
      7, Generating implicit copyin(u[:n]) [if not already present]
         Generating implicit copyout(v[:n]) [if not already present]
      8, Accelerator restriction: size of the GPU copy of M is unknown
         Complex loop carried dependence of v->,u-> prevents parallelization
         Loop carried dependence of M->,u-> prevents parallelization
         Loop carried backward dependence of u->,M-> prevents vectorization
         Complex loop carried dependence of M-> prevents parallelization
         Generating NVIDIA GPU code
          8, #pragma acc loop seq
         11, #pragma acc loop seq
      8, Generating implicit copyin(M[:]) [if not already present]
     11, Complex loop carried dependence of M->,v->,u-> prevents parallelization
         Loop carried dependence due to exposed use of v prevents parallelization
```

OpenACC: DIY

- Kernels are easy, parallel constructs are faster*;
- The “parallel” construct is the DIY of ACC usage, dev must ensure loops are indeed parallelizable;
- Trade-off: usually requires that the kernel is refactored to make proper use of the construct (a.k.a., back to the drawing board);
- Parallel allows usage of more advanced tools, such as shared memory caching;
- 3 levels of parallelism to exploit: gang, worker and vector => 1D grid of 2D thread blocks;

```
#pragma acc parallel loop gang // thread block partition
for (int i = 0; i < n; i++) {
    ...
    #pragma acc loop worker // Direction y of the thread block
    for (int j = 0; j < n; j++) {
        ...
        #pragma acc loop vector // Direction x of the thread block
        for (int k = 0; k < n; k++) {
            ...
        }
    }
}
```

Practical example: matrix multiplication

- Very time-consuming for dense matrices;
- c_ij = a_ik * b_kj implies many simultaneous dot products;
- Parallel version using MPI quite tricky, and still requires many resources;
- Easy to write a naive GPU version, much harder to optimize;

Matmul: CPU vs GPU vs GPU

```cpp
void matmulCPU(float *A, float *B, float *C, int N) {
    printf("Launching CPU matmul\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0;
            for (int k = 0; k < N; k++) {
                sum += A[i * N + k] * B[k * N + j];
            }
            C[i * N + j] = sum;
        }
    }
}
```

```cpp
__global__ void matmul0(float *A, float *B, float *C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) {
        float sum = 0;
        for (int k = 0; k < N; k++) {
            sum += A[i * N + k] * B[k * N + j];
        }
        C[i * N + j] = sum;
    }
}
```

```cpp
__global__ void tiledMatmul(float *A, float *B, float *C, int N)
{
    // Create shared memory blocks
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];
    // Get block indices
    int ib = blockIdx.x;
    int jb = blockIdx.y;
    // Get thread indices
    int it = threadIdx.x;
    int jt = threadIdx.y;
    // Get global indices
    int i = ib * 16 + it;
    int j = jb * 16 + jt;
    // Check if indices are valid
    if (i < N && j < N) {
        float sum = 0;
        // Loop over tiles
        for (int t = 0; t < N / 16; t++) {
            // Load tiles into shared memory
            As[it][jt] = A[i * N + t * 16 + jt];   // Load a chunk of A
            Bs[it][jt] = B[(t * 16 + it) * N + j]; // Load a chunk of B^T
            // Synchronize threads
            __syncthreads();
            // Compute partial sum
            for (int k = 0; k < 16; k++) {
                sum += As[it][k] * Bs[k][jt];
            }
            // Synchronize threads
            __syncthreads();
        }
        // Write result
        C[i * N + j] = sum;
    }
}
```

- CPU
- Naive GPU
- Tiled GPU

Matmul: OpenACC versions

```
void matmulKernels(int n, float *a, float *b, float *c) {
    int i, j, k;
    float sum;
    #pragma acc kernels
    {
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                sum = 0.0;
                for (k = 0; k < n; k++) {
                    sum += a[i*n+k] * b[k*n+j];
                }
                c[i*n+j] = sum;
            }
        }
    }
}
```

```
void matmulParallel_V0(int n, float *a, float *b, float *c) {
    int i, j, k;
    float sum;
    #pragma acc parallel loop gang worker collapse(2)
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            sum = 0.0;
            #pragma acc loop vector reduction(+:sum)
            for (k = 0; k < n; k++) {
                sum += a[i*n+k] * b[k*n+j];
            }
            c[i*n+j] = sum;
        }
    }
}
```

Matmul: ACC vs CUDA timings

- All test done using nrows=ncols=2000
  - Acc-kernels: 8.7s
  - Acc-parallel: 422.126ms
  - CUDA-naive: 19.225ms
  - CUDA-tiled: 11.873ms

Matmul: analysis

- Naive version requires many global memory access requests, which is “slow”;
- Shared memory version loads data into cache, making data access much faster;
- In practice, performance gains only for very large datasets, and dependent on tiling size;
- ACC kernels slowed down as compiler thinks there are data dependencies;
- CUDA is faster as it can more easily exploit multi-level parallelism (2D grid of 2D blocks);

Example: FEM1D

- FEM: backbone of BSC-CASE research;
- Alya, SOD2D are FEM-based approaches;
- Elemental operations are costly, and can benefit greatly from GPU usage;
- Objective: introduce OpenACC into a 1D scalar convection kernel;

Conclusions

- GPUs are not magic boxes: just because code is on the device, doesn’t mean it go brrrrr!
- CUDA is not practical, OpenACC can deliver excellent performance without making you rage-quit;
- However, CUDA CAN deliver fantastic performance if multidimensional grids can be used in the kernel;
- ACC kernels is like James May driving. Parallel is like Hammond;
- Micro-managing host/device data transfers is painful, and you’re probably doing it wrong; use managed memory to avoid performance and hair loss;