

Introduction to GPU programming

Lucas Gasparino



Instructions

- `git clone repo`
- `rsync -avzrP fold "[user]@plogin1.bsc.es:/"`
- `ssh [user]@plogin1.bsc.es`
- `Module load cuda/10.2 nvhpc/22.9`
- Alternative:

Objectives

- Overview of GPU architecture
- Accessing the hardware
 - Overview of CUDA
 - OpenACC basics
- Example 1: vector addition
- Example 2: Conjugate Gradient

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;



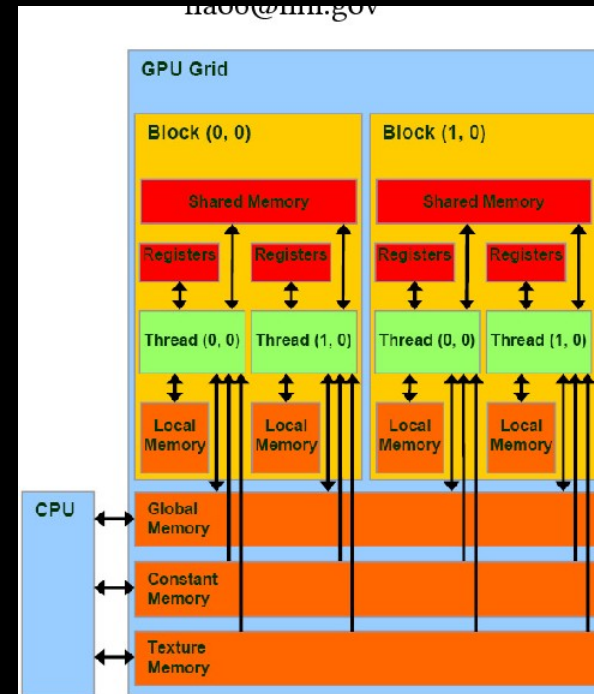
GPU arch: compute units

- Core composed of many SMs;
- Each SM contains a multitude of threads;
- Similar to having many multicore CPU packages bundled together;



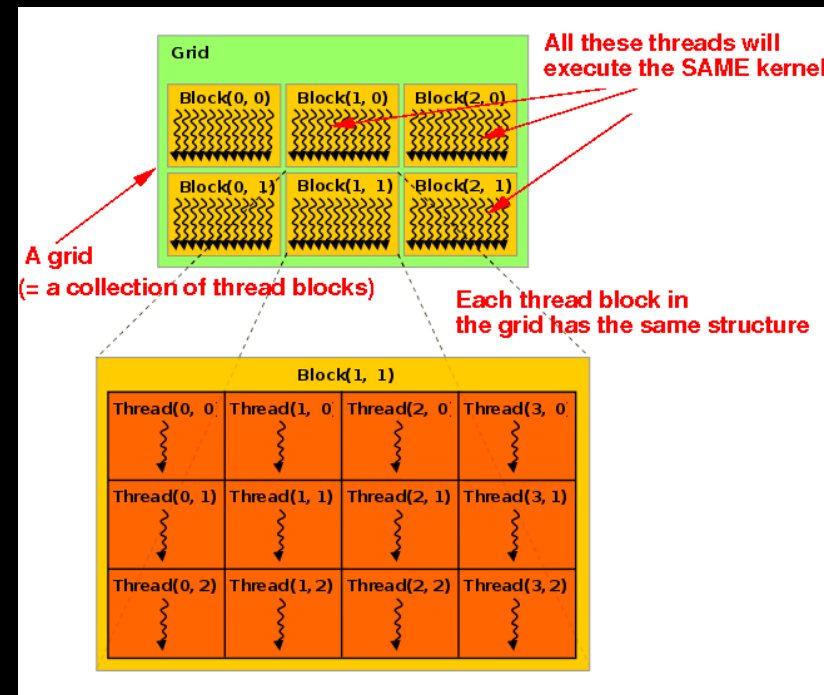
GPU arch: internal memory

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



GPU arch: thread usage

- Kernel execution must happen in a *grid*, composed of *thread blocks*;
- Thread blocks are groupings of threads that access the same region of data;
- Data in a block is not visible to another (privatized to one of the caches);
- Developer's task: ensure good data partition within the kernel grid launch;



GPU programming

- Low level models allow greater control of the GPU, but are significantly harder to use:
 - CUDA, OpenCL, SASS, NVPTX
- Pragma-based high level models direct the compiler into generating GPU-capable code, but offer less control options;
 - OpenACC, OpenMP
- Language standard parallelization, with compiler marking certain intrinsics and keywords for parallel execution on a device:
 - Fortran AutoPar, C++ stdpar

CUDA API

- Excellent low-level control capabilities:
 - Micromanaged usage of threads and thread blocks;
 - Allow access to L1 and Texture memory, as well as Shared memory;
 - 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 Example

```
// CUDA kernel for vector addition
__global__ void vecAdd(int n, float *a, float *b, float *c) {
    // Associate array entry to grid index
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    // If array entry is within bounds, add components
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
```

```
// Allocate memory on device
float *d_a, *d_b, *d_c;
cudaMalloc(&d_a, n * sizeof(float));
cudaMalloc(&d_b, n * sizeof(float));
cudaMalloc(&d_c, n * sizeof(float));
// Initialize array
for (int i = 0; i < n; i++) {
    a[i] = 1.0f;
    b[i] = 2.0f;
}
// Copy array to device
cudaMemcpy(d_a, a, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, n * sizeof(float), cudaMemcpyHostToDevice);
// Launch kernel
int blockSize = 256; // Threads per block
int numBlocks = (n + blockSize - 1) / blockSize; // Number of blocks to launch
vecAdd<<<numBlocks, blockSize>>>>(n, d_a, d_b, d_c);
// Copy result back to host
cudaMemcpy(c, d_c, n * sizeof(float), cudaMemcpyDeviceToHost);
// End of main
```

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 “`!$`”;
- GPU compilation: “`[nvcomp] -gpu=ccXY,[options] -cuda -acc`”

OpenACC: the “magic” way

- “**#pragma acc kernels**”: automatic compute construct directive;
- Compiler will try to generate “safe” parallel code when this directive is encountered;
- If any acc directive exists, compiler tries to generate necessary data transfers. **NOT SAFE TO ASSUME THIS WORKS!**

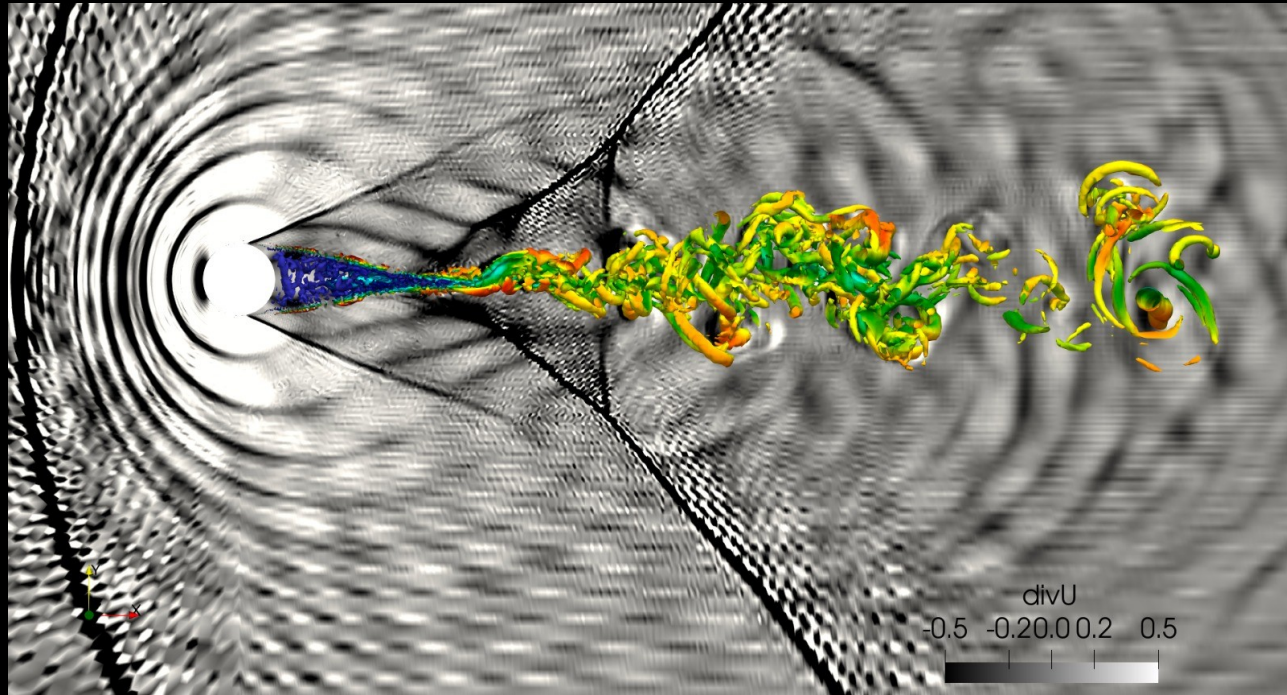
OpenACC: managed memory

- 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;
- Completely safe for pointer and values, not usable with objects!

OpenACC: the DIY way

- Automagically generated code might be slow/not efficient;
- The “`#pragma acc parallel`” compute construct allows the user to more explicitly guide the compiler;
- Compiler will **NOT** check for single thread correctness!
- Used on nested loops, every loop must be marked with an appropriate compute construct to guide parallelization/serialization;

Motivating example



Example 1: Vector Addition

- The “hello world” of OpenACC;
- Great for exploring compilation options and data transfer tricks!
- Objective: add 2 vectors and compare times against OpenMP kernel;

Example 2: Conjugate Gradient

- Solver operations are mostly dot products and matrix-vector products, prime candidates for fine-grained parallelism
- Objective: Naive implementation of OpenACC on a dense matrix CG solver;

Example 3: FEM example

- 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 brrrrrr!
- CUDA is not practical, OpenACC can deliver excellent performance without making you rage-quit;
- Micro-managing host/device data transfers is painful, and you're probably doing it wrong; use managed memory to avoid performance and hair loss;