Heterogeneous computing with Python: why we need it?

Javier Vegas Regidor

31/10/2018

ECMWF Python Workshop 2018

Heterogeneous computing

Heterogeneous computing refers to systems that use more than one kind of processor or cores.

These systems gain performance or energy efficiency not just by adding the same type of processors, but by adding dissimilar coprocessors, usually incorporating specialized processing capabilities to handle particular tasks.

Those special coprocessors can be Graphics Processing Units (GPU), Field-programmable gate arrays (FPGA) ...

Emergent technologies at BSC

MareNostrum 4 is our general-purpose cluster with a peak power is 11.15 Petaflops

In the same HPC compound there are also three smaller clusters for emergent technologies:

- A cluster consists of IBM POWER9 processors and NVIDIA Volta GPUs. Computing power over 1.5 Petaflop/s.
- A cluster made up of Intel Knights Hill (KNH) processors. Computing power in excess of 0.5 Petaflop/s.
- A cluster formed of 64 bit ARMv8 processors in a prototype machine. Computing power over 0.5 Petaflop/s.

NVIDIA GPU CENTER OF EXCELLENCE

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

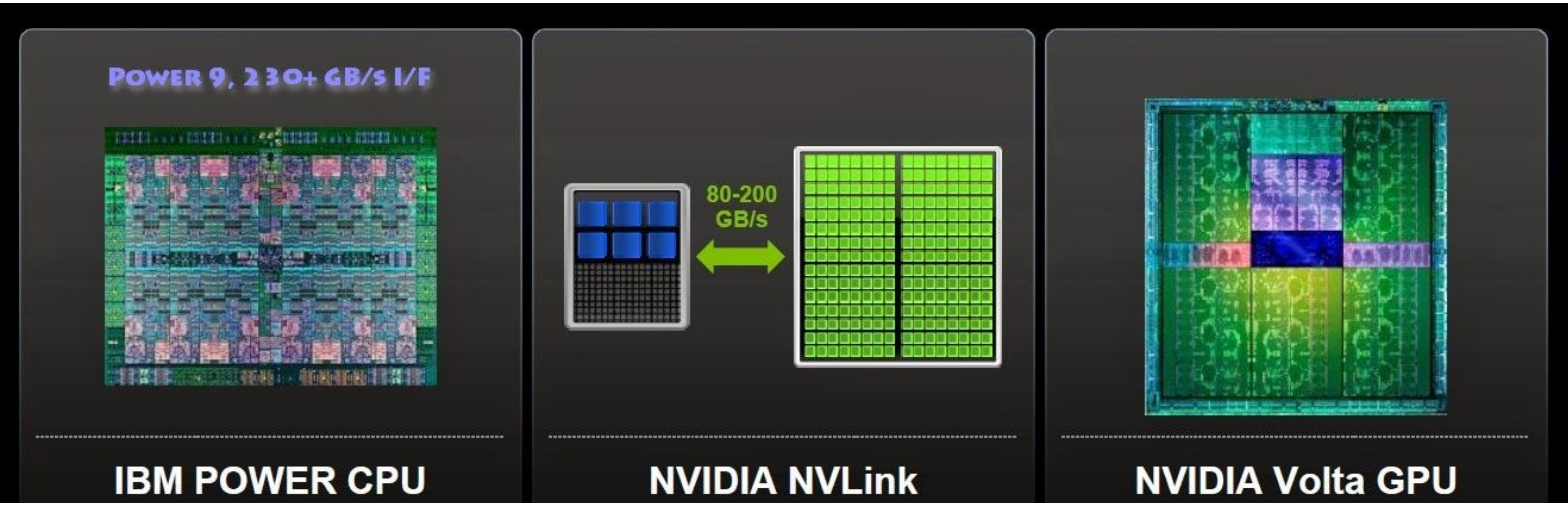
CTE-Power specifications

2 login nodes and 52 compute nodes, each of them:

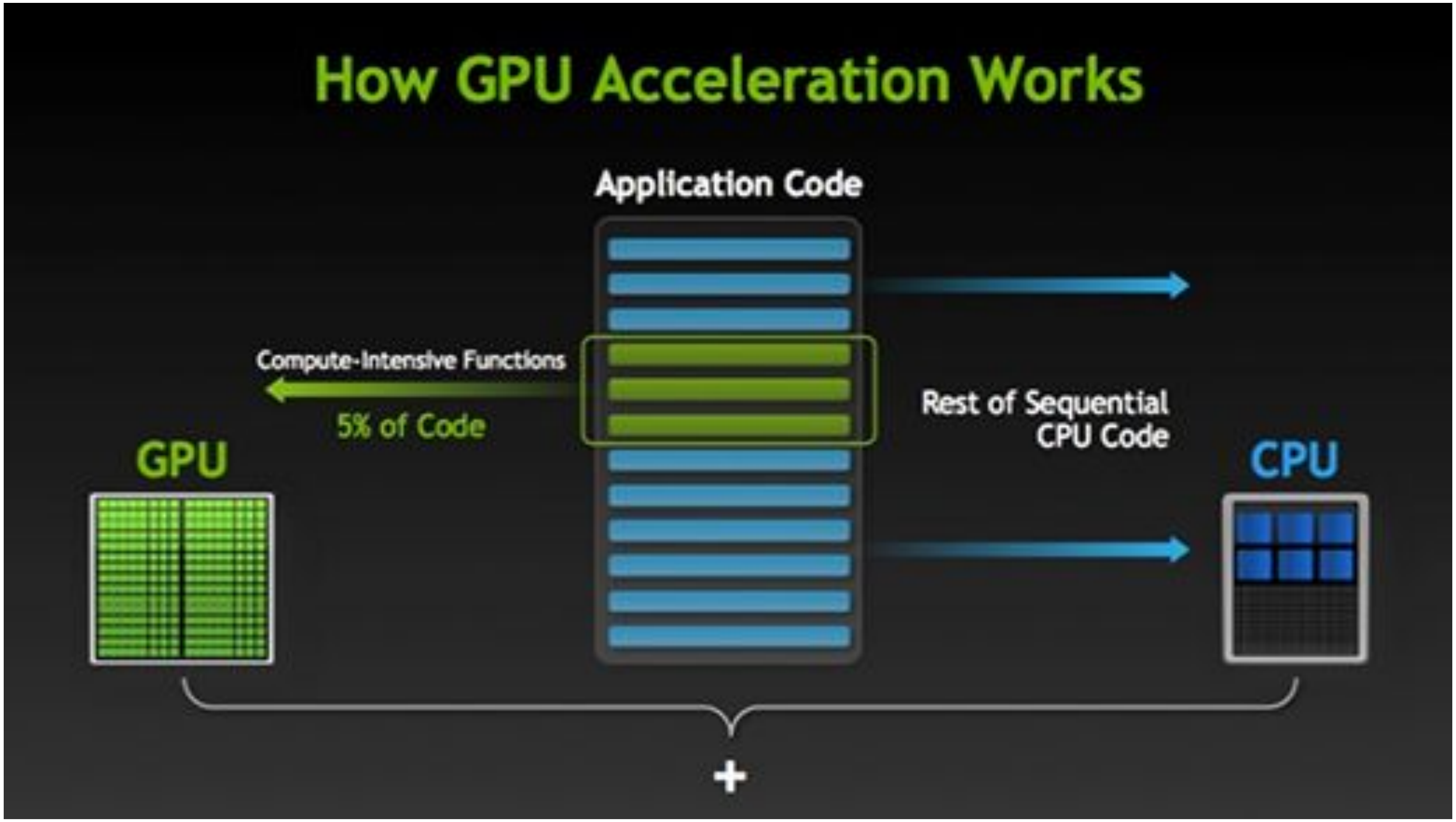
- 2 x IBM Power9 8335-GTG @ 3.00GHz (20 cores and 4 threads/core, total 160 threads per node)
- 512GB of main memory
- 2 x 3.2TB NVME scratch
- 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2.

Too powerful to ignore !!

Why we need to adapt?



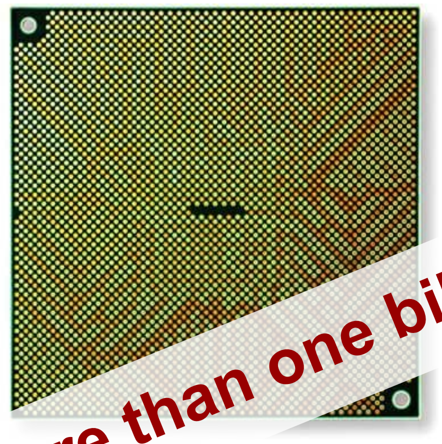
How we can adapt?

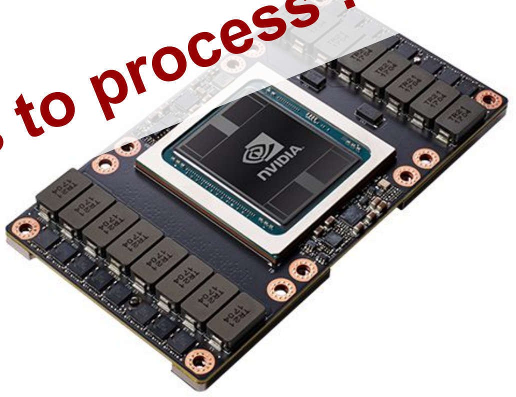

How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
GPU
Rest of Sequential CPU Code
CPU
+

http://la.nvidia.com/object/what-is-gpu-computing-la.html

It is worth it?

- Let's compare two implementations of ocean heat content
  - Based in numpy, vectorized, serial
  - GPU accelerated using custom code
- We compute it for one year of EC-Earth HR experiment (monthly data) from NVME folder
  - 12 months, 75 ocean levels, 1442 x 1050 irregular grid





More than one billion points to process !

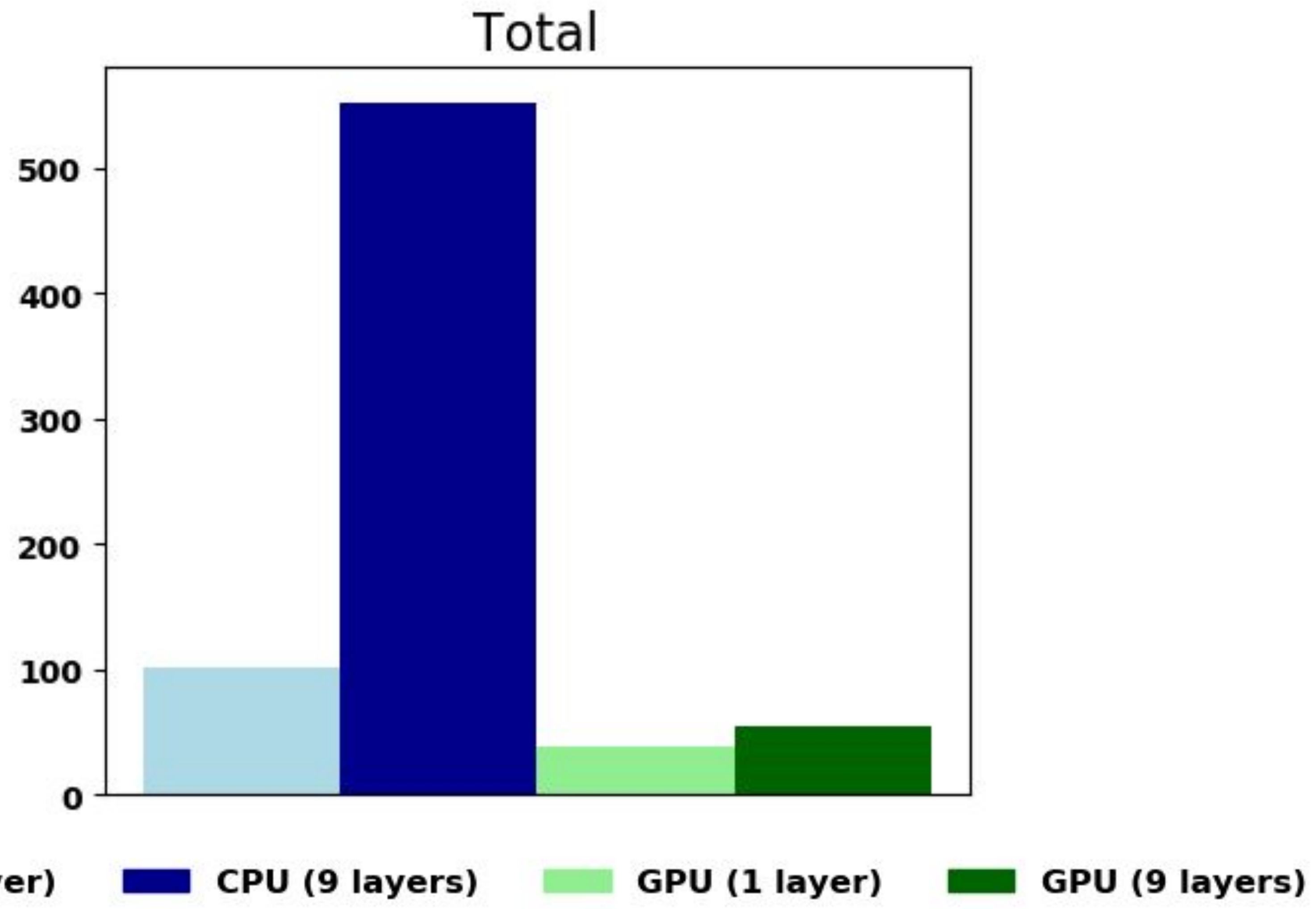
The algorithm

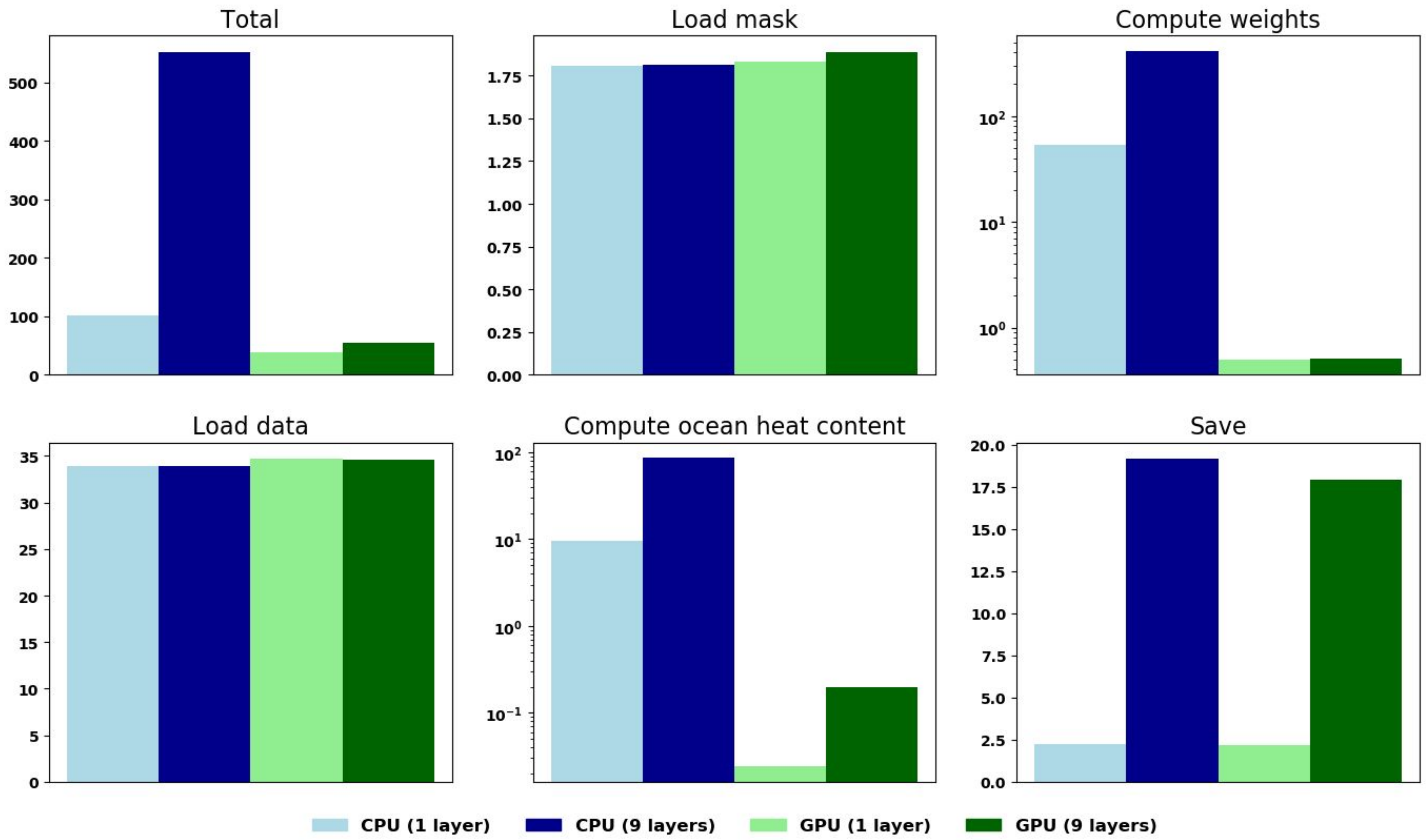
1. **Load mask**
   Load grid information from disk
2. **Compute weights**
   How much of the cell is in the specified layer?
3. **Load thetao data**
   Load sea water potential temperature data
4. **Compute OHC**
   Multiply each thetao value by cell weight and aggregate over levels
5. **Save results**
   Save computed OHC to a netCDF file

Warning !!!

- Comparing exact numbers is worthless with the amount of time I spent on both implementations
  - CPU implementation is fairly optimized compared with my first attempt.
  - No effort in GPU optimization. Some of my code may horrify actual experts in GPU computing
- The aim of this work is evaluating if going for CPU + GPU is worth the effort, not how much gain can we have from it

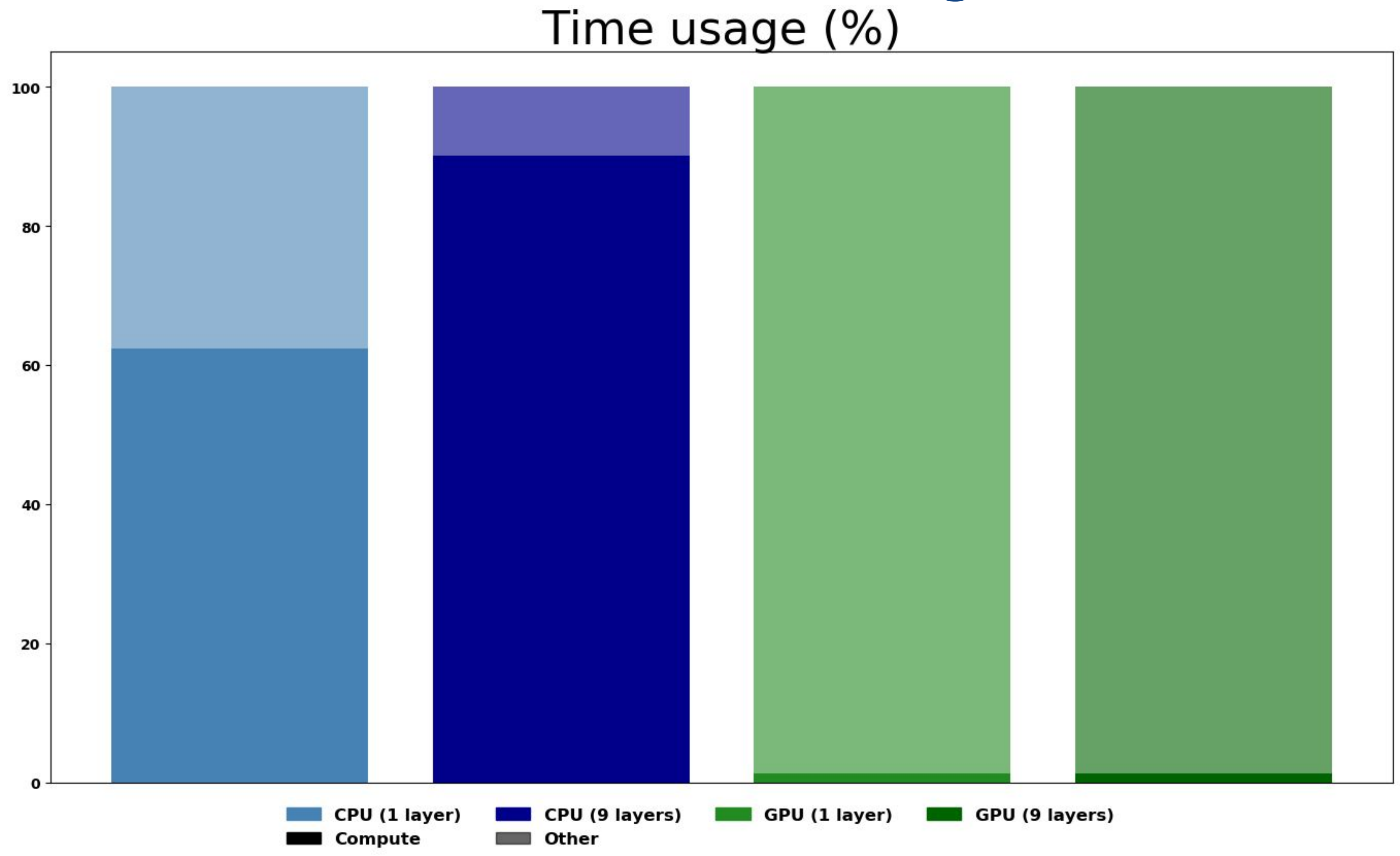
Performance





Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Relative time usage



It is really that difficult to use GPUs?

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

How can we implement it?

Two computing steps:

- Compute weights
- Compute final ohc

Both should be done in the GPU: we need to develop specific code for them (it is called a ‘kernel’) using CUDA API (using C or Fortran)

I used CUDA through PyCUDA Python package

Moving data from RAM to VRAM and back

```
# Load mesh info
gpu e3t = cuda.gpuarray.to gpu async(e3t.astype(np.float32))
gpu mask = cuda.gpuarray.to gpu async(mask.astype(np.float32))
gpu depth = cuda.gpuarray.to gpu async(depth.astype(np.float32))
gpu_weight = cuda.gpuarray.empty_like(gpu_mask)

## Compute weight
del gpu e3t
del gpu depth
del gpu_mask

# Load thetao
gpu thetao = cuda.gpuarray.to gpu async(thetao)
gpu ohc = cuda.gpuarray.empty((shape[0], shape[2], shape[3]),
np.float32)

## Compute ohc
del gpu thetao
ohc = gpu ohc.get()
del gpu_ohc
```

First kernel: compute weights

```
weight declaration = "float *e3t, float
*depth, float *mask, float *weight"

compute weight = ElementwiseKernel(
weight declaration,  weight_kernel,
"compute_weight")
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

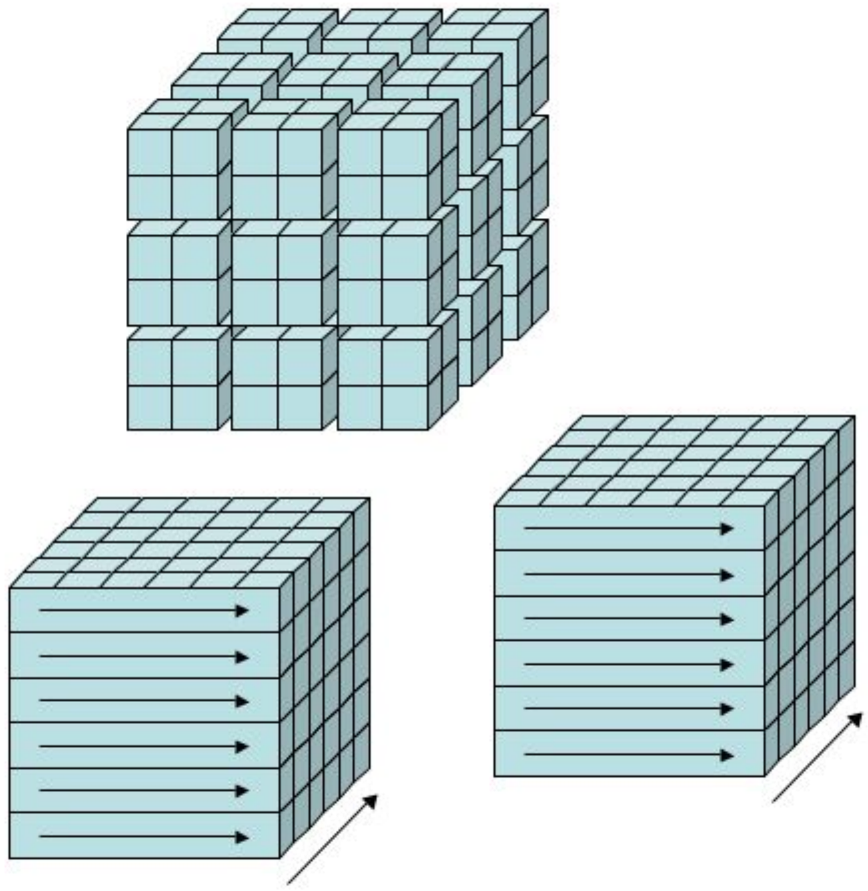
First kernel: compute weights

```
"""
   float w=0;
   float top = depth[i]; float bottom = top + e3t[i];
   int masked = mask[i];
   if (masked > 0 && bottom >= {0} && top <= {1})
   {{
     float up = top;
     float down = bottom;
     if (top < {0})
       up = {0};
     if (bottom > {1})
       down = {1};
     w = (down - up) * 1020 * 4000;
   }}
   weight[i] = w;
""".format(min_depth, max_depth)
```

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Second kernel: array dimensions

- thetao:
  - time
  - lev
  - i
  - j
- weight:
  - lev
  - i
  - j
- ohc:
  - time
  - i
  - j



Second kernel: compute ohc

```
"""
__global__ void ohc(float *thetao, float *weight, float* ohc)
  {{
    int iohc = blockIdx.y * {total_cells} + blockIdx.x *
{block_size} + threadIdx.x;
    float temp=0;
    if (threadIdx.x + blockIdx.x * {block_size} >= {total_cells})
      return;
    for(int lev=0; lev < {levels}; lev++) {{
      int idw = lev * ({total_cells}) + blockIdx.x * {block_size}
+ threadIdx.x;
      int idt = blockIdx.y * ({levels}*{total_cells}) + idw;
      temp += thetao[idt] * weight[idw];
    }}
    hc[iohc] = temp;
  }}""".format(levels=thetao.shape[1],
               block_size=block_size,
               grid_size=grid_size,
               total_cells=thetao.shape[2] * thetao.shape[3])
```

Center
Centro Nacional de Supercomputación

Second kernel: compute ohc

```
""""
__global__ void ohc(float *thetao, float *weight, float* ohc)
  {{
    int iohc = whatever
    float temp=0;
    if (lat-lon does not exists)
      return;
    for(int lev=0; lev < {levels}; lev++) {{
      int idw = whatever 2;
      int idt = whatever 3: revenge of the sithdex;
      temp += thetao[idt] * weight[idw];
    }}
    hc[iohc] = temp;
  }}
""""
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Conclusions

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Evolving with HPCs

- HPCs are becoming heterogeneous and we need to make an effort to adapt
- In particular, HPCs based on CPU + GPU are becoming more popular
- GPUs provide an opportunity to speed up great parts of our current tasks:
  - Time and regional means
  - Anomalies
  - Masking
  - Diagnostics
- In general, any computation that is applied to all points of a matrix or a small subset of it is suitable for straightforward GPU acceleration.

Change your mind

- We can find us in an scenario where data loading and writing is more time consuming than computation ***by several orders of magnitude***
- Our tools will need to evolve to focus on efficient data management and workflow planning
- Teaching users to plan ahead and request at once everything they ***may*** need for their analysis will be crucial to the best use of the resources

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Thank you

javier.vegas@bsc.es