# Parallel Calibration


## Marzia Rivi (Oxford)


Hilado F2F meeting

Oxford, 11-12 Apr 2013


Contract No: 283393

# Calibration problem and data size

- From a mathematical point of view: least square minimisation

  $G_s = \mathit{argmin}\ ||M-GDG^H||_F$

  M,D visibilities matrices of order n (non-polarised) or 2n (polarised)

  G diagonal (non-polarised) or 2×2 block diagonal (polarised) complex gains

  n = number of antennas

- Typical LOFAR data

  96 antennas

  512 channels

  i.e. calibration to be solved for 512 matrices of order 96 per data cube (of size 288 MB)

Independent problems for each frequency and for each data cube

Small matrices

Contract No: 283393

# Parallelisation

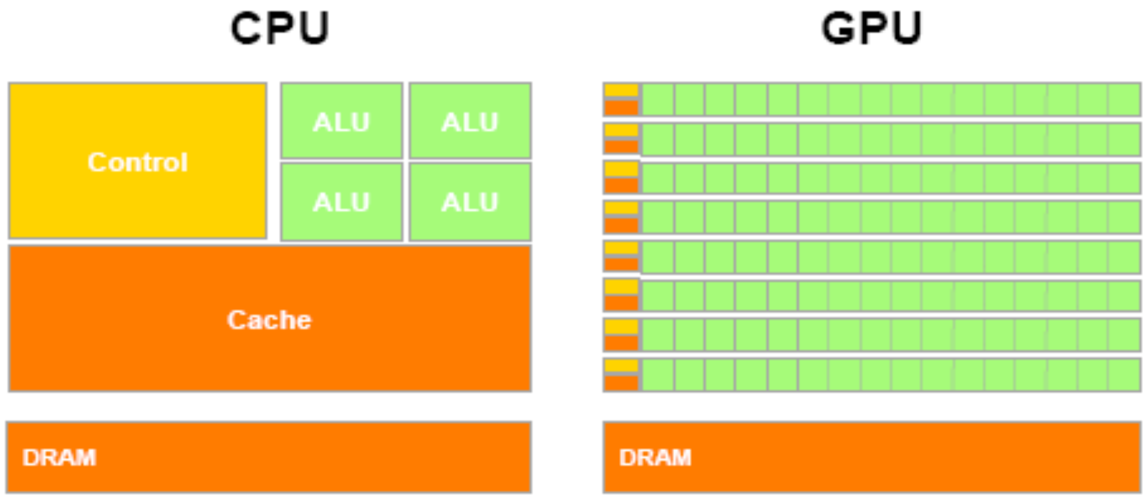
Given a data cube:

- calibrate each frequency in parallel (1st level of parallelism – embarrassingly parallelism)
- solve each frequency by distributing time-consuming sections of the minimisation algorithm among threads sharing the same memory (2st level of parallelism)
  - cooperation
  - synchronisation

Possible programming paradigms:

- OpenMP ⟶ multi-core processor, Intel MIC
- CUDA ⟶ GPU

# Different worlds: CPU and GPU

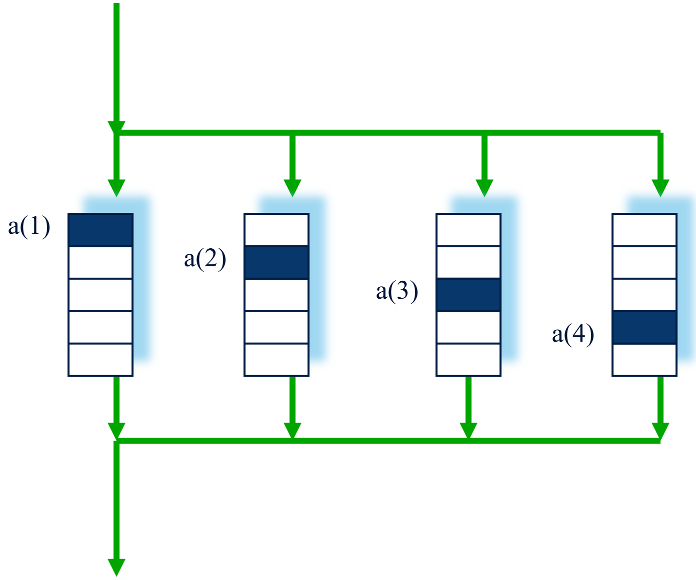



| | | |
|---|---|---|
| *Threading resources* | multi-core (few)<br>sophisticated control logic unit<br>tens of threads | many-cores (several hundreds)<br>simple control logic unit<br>thousands of threads |
| *Threads* | «Heavy» entities | Extremely lightweight, managed grouped into warps |
| *Memory* | large cache memories to reduce access latencies | long-latency memory accesses<br>large bandwidth<br>small memory size (6GB) |

# OpenMP

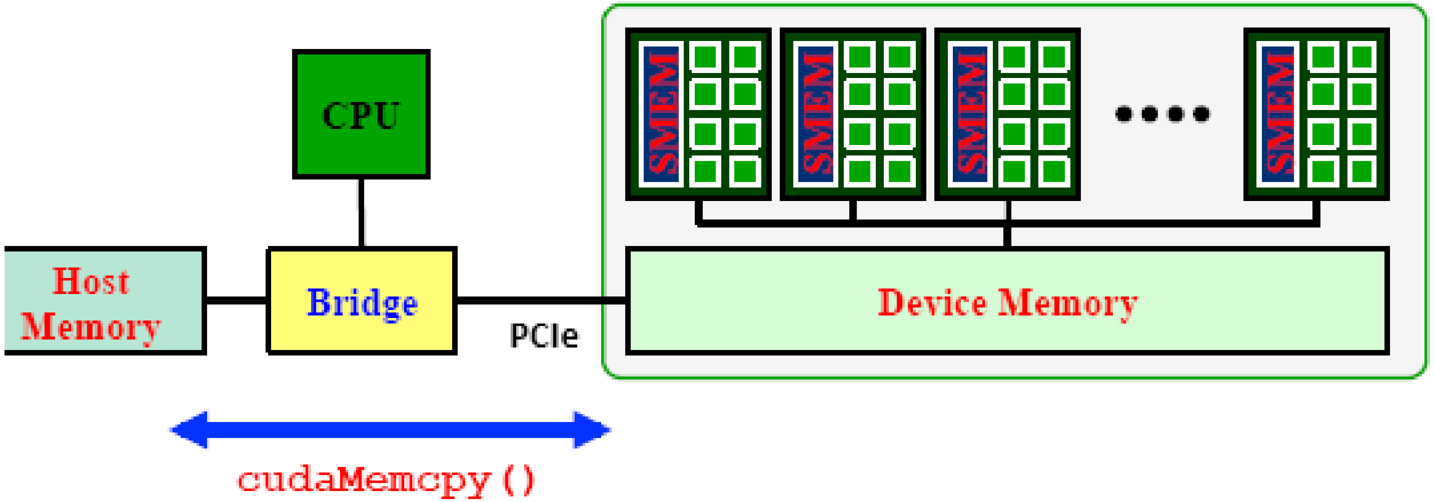
- API for *shared-memory* parallelism in C, C++ and Fortran
- compiler directives to define parallel regions of the code
- library routines
- environment variables

```
!$omp parallel

   .........

!$omp do
    do i = 1, n
       a(i) = xxx
    end do
!$omp end do

   .........
!$omp end parallel
```

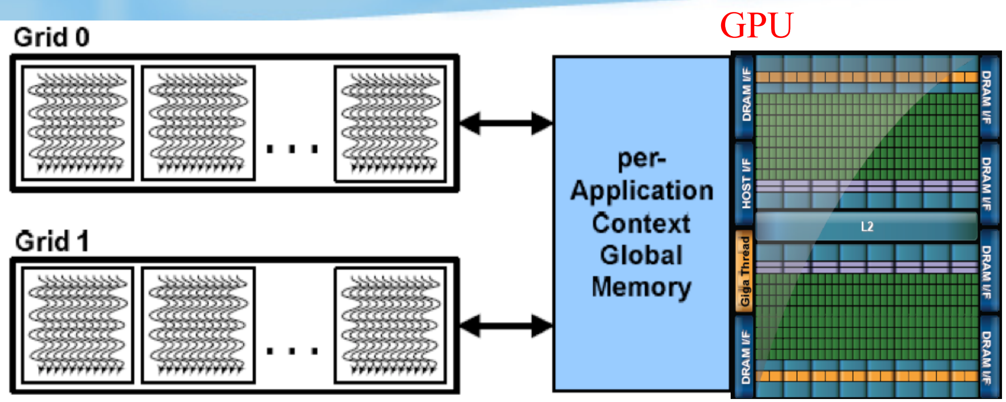


# CUDA

- Provides API to manage joint CPU/GPU execution of an application
- Extension of the C/C++ (Fortran) language
- Serial sections of the code are performed by CPU (host)
- The parallel ones (that exhibit rich amount of *data parallelism*) are performed by GPU (device) in the SPMD mode as CUDA kernels.
- Host and device have separate memory spaces: programmers need to transfer data between CPU and GPU via PCIe.

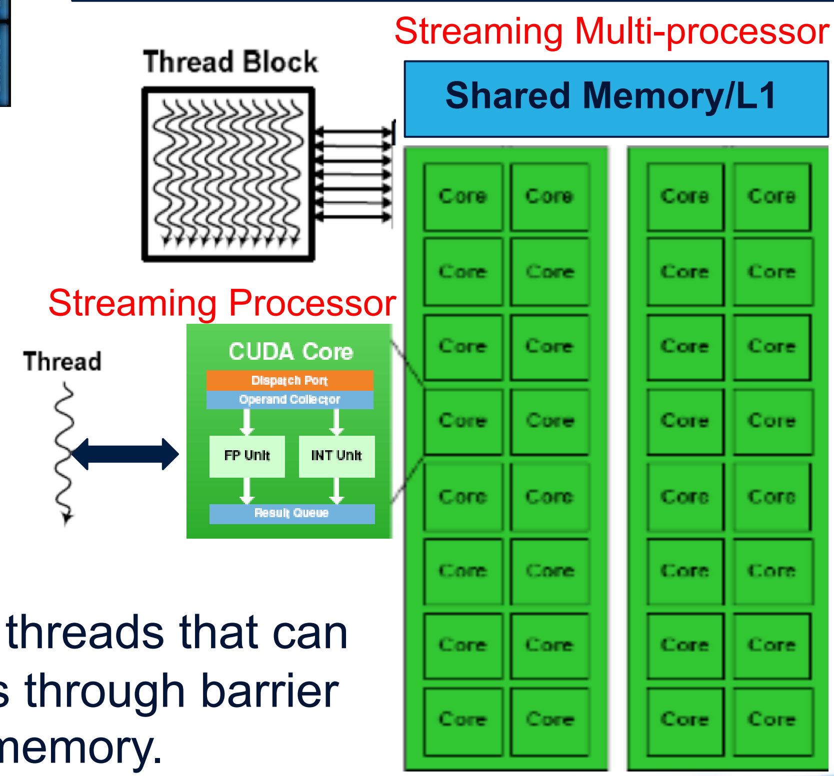


# CUDA threads organization




A kernel is executed as a *grid* of many (thousands) parallel threads organized into *blocks* of the same size.
Block size and number of blocks are parameters defined in the code.

**Kepler Tesla K20**:
6GB global memory
48kB shared memory
192 single-precision cores per SMX
15 SMX




**Block of threads:**
set of concurrently executing threads that can *cooperate* among themselves through barrier synchronization and shared memory.

# GPU non-polarised StefCal

- Transfer a fixed number *k* of data cubes on the device memory
- Perform calibration kernel:
  - CUDA blocks of size n=number of antennas, each solving one frequency
  - Within the block
    - each thread solve one antenna gain $g_i$
    - for each iteration, $g_i$ computations are independent
    - threads synchronisation at the end of each iteration
- Copy back results to the host

Example: LOFAR data

```
__global__ void KernelCal(…);
dim3 gridDim(k•512);  // number of blocks
dim3 blockDim(96);    // threads per block

//call the kernel
KernelCal<<< gridDim, blockDim >>>(<arguments>);
```