

# High-Throughput Molecular Dynamics on GPUs using ACEMD

M J Harvey

Imperial College London  
CBBL, IMIM-UPF

Daresbury GPU Workshop Sept 2010

# Why GPUs for scientific computing?

Why should we care about GPUs?

# Why GPUs for scientific computing?

Why should we care about GPUs?

Continued relevance:

- ▶ **High performance** for FP & integer arithmetic *competitive market*.
- ▶ **Low cost** *subsidised by volume GPU market*.
- ▶ **Short Product cycle** *Follow Moore's Law trends/process technology*.

## Why GPUs for scientific computing?

Why should we care about GPUs?

Continued relevance:

- ▶ **High performance** for FP & integer arithmetic *competitive market.*
- ▶ **Low cost** *subsidised by volume GPU market.*
- ▶ **Short Product cycle** *Follow Moore's Law trends/process technology.*

Practically useful: (cf Clearspeed, FPGAs)

- ▶ **Low barrier to entry** *CUDA free, works on any NV GPU*
- ▶ **Not difficult to program** *cf FPGAs*
- ▶ **Able to do more than trivial things** *cf Clearspeed*

## Why GPUs for scientific computing?

Why should we care about GPUs?

Continued relevance:

- ▶ **High performance** for FP & integer arithmetic *competitive market*.
- ▶ **Low cost** *subsidised by volume GPU market*.
- ▶ **Short Product cycle** *Follow Moore's Law trends/process technology*.

Practically useful: (cf Clearspeed, FPGAs)

- ▶ **Low barrier to entry** *CUDA free, works on any NV GPU*
- ▶ **Not difficult to program** *cf FPGAs*
- ▶ **Able to do more than trivial things** *cf Clearspeed*

For Top500 watchers:

- ▶ The only thing likely to satisfy the Exascale people. *GFLOPS/Watt*

## GPUs in scientific computing

- ▶ Hot topic: Much activity in the literature
  - ▶ Many examples of implementations of key classes of algorithm (Berkeley's Seven Dwarves)
- ▶ Many headline examples on NVIDIA website -*some stretch credibility*
- ▶ *Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers* D. H. Bailey, Supercomp. Rev. 54-55 (1991)
- ▶ *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU* V. W. Lee, SIGARCH 38 (3) (2010)
- ▶ HMPP porting cases. (CAPS presentation)

For the computational scientist it's not clear whether GPU computing will be as useful as promised.

The screenshot displays the 'CUDA ZONE' website. At the top, there are navigation links: 'WHAT IS NEW', 'WHAT IS CUDA?', 'CUDA APIs', and 'DEVELOPERS'. Below this is the 'CUDA Community Showcase' section, which states: 'GPU computing applications developed on the CUDA architecture by programmers, scientists, and researchers around the world.' The main content area features a grid of 15 application showcases, each with a thumbnail image and a title. The titles include: 'Recursive Fast-Four Transform Algorithm on GPU', 'Implementation of a Lattice Boltzmann method on GPU', 'Fast and Exact Solution of 1D Schrödinger Equation', 'Fast Time Evolution for Composite Systems', 'Fast-Chem', 'GPU-Based Acceleration of the Genetic Algorithm', 'Computer Generated Images on GPU - Simple color', 'GPU Based Sparse Grids Technique for Solving Multi...', 'Accelerating numerical solution of Stochastic DE', 'Faster Algorithms for Solving Heavily Equations', 'Real Time Visualization of non-linearly interacting...', 'Parallel computing with graphics processing units', 'Optimization of FTLE calculation', 'Stochastic Differential Equations with CUDA', and 'AN APPROACH FOR THE EFFICIENT IMPLEMENTATION OF GPU'. At the bottom of the grid, there is a search bar and a filter button labeled 'Filter by Application Type'.

## When to use GPUs

What sort of applications benefit from GPU use?

## When to use GPUs

What sort of applications benefit from GPU use?

- ▶ (From the perspective of a computational scientist and HPC Service provider)

## When to use GPUs

### What sort of applications benefit from GPU use?

- ▶ (From the perspective of a computational scientist and HPC Service provider)
- ▶ Will address this question, using ACEMD and GPUGRID as a case study.
- ▶ Present current work on adapting ACEMD for maturing GPU sector.

# GPUs- When to they make sense?

Broadly, all computational science codes are either:

- ▶ Serial ensemble jobs
- ▶ Parallel (MPI) jobs

## GPUs- When to they make sense?

Broadly, all computational science codes are either:

- ▶ Serial ensemble jobs
- ▶ Parallel (MPI) jobs

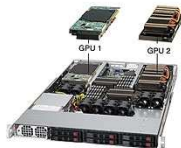
Snapshot from IC HPC cluster:

- ▶ 61 active users from across college
- ▶ 286 serial jobs (batches of 50-100) (16%)
- ▶ 110 single-node parallel jobs (44%)
- ▶ 42 multi-node MPI jobs (40%)

## GPUs- When to they make sense?

Broadly, all computational science codes are either:

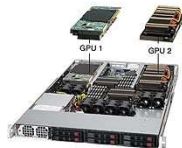
- ▶ Serial ensemble jobs
- ▶ Parallel (MPI) jobs
- ▶ Consider a hypothetical cluster where adding 2 GPUs costs 2 CPUs/12 cores, eg:
  - ▶  $N$  nodes with 2GPUs, 12 CPU cores in 1u.
  - ▶  $2N$  nodes with 24 CPU cores in 1u
- ▶  $\approx$  true whether capex, opex or space-constrained



## GPUs- When to they make sense?

Broadly, all computational science codes are either:

- ▶ Serial ensemble jobs
- ▶ Parallel (MPI) jobs
- ▶ Consider a hypothetical cluster where adding 2 GPUs costs 2 CPUs/12 cores, eg:
  - ▶  $N$  nodes with 2GPUs, 12 CPU cores in 1u.
  - ▶  $2N$  nodes with 24 CPU cores in 1u
- ▶  $\approx$  true whether capex, opex or space-constrained



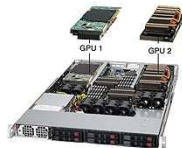
Ask the question:

When is there a net gain in changing existing CPU codes to use the GPUs?

## GPUs- When to they make sense?

Broadly, all computational science codes are either:

- ▶ Serial ensemble jobs
- ▶ Parallel (MPI) jobs
- ▶ Consider a hypothetical cluster where adding 2 GPUs costs 2 CPUs/12 cores, eg:
  - ▶  $N$  nodes with 2GPUs, 12 CPU cores in 1u.
  - ▶  $2N$  nodes with 24 CPU cores in 1u
- ▶  $\approx$  true whether capex, opex or space-constrained



Ask the question:

When is there a net gain in changing existing CPU codes to use the GPUs?

NB: Explicitly ignore cost of software engineering and algorithmic suitability

## GPUs - When do they make sense? Serial jobs

Consider Serial program that are run in an ensemble (eg Matlab, R):

- ▶ Throughput increases linearly with number of cores.
- ▶ Any optimisation will increase throughput of ensemble in proportion to speedup of single instance.

## GPUs - When do they make sense? Serial jobs

Consider Serial program that are run in an ensemble (eg Matlab, R):

- ▶ Throughput increases linearly with number of cores.
- ▶ Any optimisation will increase throughput of ensemble in proportion to speedup of single instance.
- ▶ Net increase in throughput when GPU code **> 7x faster** than CPU. (24 cores vs 10 cores + 2 GPUs)
- ▶ Even a code that runs completely on the GPU requires 1 CPU core to support it.
- ▶ Best system maximises GPUs/host system

Adding GPUs can lead to **quantitative** increase in capability  
(Probably modest on a mixed-workload system)

## GPUs - When do they make sense? MPI jobs

Consider an MPI program that runs on multiple nodes:

## GPUs - When do they make sense? MPI jobs

Consider an MPI program that runs on multiple nodes:

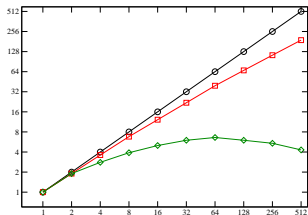
- ▶ Reduction in runtime proportionately less than CPU speedup factor (Amdahl's Law, MPI communications unaffected)
- ▶ Easiest mapping is 1 GPU per MPI rank, otherwise imbalanced/heterogeneous program
- ▶ Parallel performance on  $N$  GPUs should be  $> 12N$  cores running MPI code.

## GPUs - When do they make sense? MPI jobs

Consider an MPI program that runs on multiple nodes:

- ▶ Reduction in runtime proportionately less than CPU speedup factor (Amdahl's Law, MPI communications unaffected)
- ▶ Easiest mapping is 1 GPU per MPI rank, otherwise imbalanced/heterogeneous program
- ▶ Parallel performance on  $N$  GPUs should be  $> 12N$  cores running MPI code.

ectly.

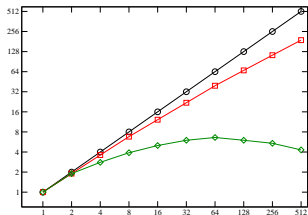


## GPUs - When do they make sense? MPI jobs

Consider an MPI program that runs on multiple nodes:

- ▶ Reduction in runtime proportionately less than CPU speedup factor (Amdahl's Law, MPI communications unaffected)
- ▶ Easiest mapping is 1 GPU per MPI rank, otherwise imbalanced/heterogeneous program
- ▶ Parallel performance on  $N$  GPUs should be  $> 12N$  cores running MPI code.

ectly.



GPUs useful when:

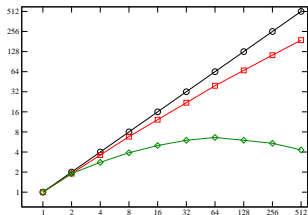
- ▶ Scaling characteristics of CPU parallel code make desired level of performance impossible/prohibitively expensive. *strong scaling*
- ▶ The serial/small-parallel GPU code is fast enough to replace a multi-node/CPU parallel job. Effective speedup much greater than vs serial code.
- ▶ Best system maximises GPUs/host system - maximise inter-GPU BW.

## GPUs - When do they make sense? MPI jobs

Consider an MPI program that runs on multiple nodes:

- ▶ Reduction in runtime proportionately less than CPU speedup factor (Amdahl's Law, MPI communications unaffected)
- ▶ Easiest mapping is 1 GPU per MPI rank, otherwise imbalanced/heterogeneous program
- ▶ Parallel performance on  $N$  GPUs should be  $> 12N$  cores running MPI code.

ectly.



GPUs useful when:

- ▶ Scaling characteristics of CPU parallel code make desired level of performance impossible/prohibitively expensive. *strong scaling*
- ▶ The serial/small-parallel GPU code is fast enough to replace a multi-node/CPU parallel job. Effective speedup much greater than vs serial code.
- ▶ Best system maximises GPUs/host system - maximise inter-GPU BW.

Adding GPUs can lead to **qualitative increase in capability**  
And can reduce need for high-performance interconnects

## Aside: GPU systems for HPC clusters

Use cases suggest that “fat” GPU nodes are most useful:

- ▶ Minimise overhead of host system
- ▶ Low CPU core: GPU ratio, good match for MPI.
- ▶ Maximise inter-GPU BW for parallel jobs



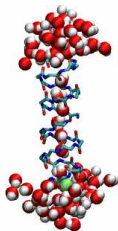
### Tyan B7015

- ▶ 8x GPUs (any type)
- ▶ 16x PCIe 2.0 2:1
- ▶ dual socket
- ▶ 1:1 core:GPU ratio
- ▶ no external cabling
- ▶ 2GPU/u density (4u box)

## Molecular Dynamics and ACEMD

# Molecular dynamics - algorithms 1

1. Give atoms initial positions  $\vec{r}(t = 0)$  and velocities  $\vec{v}(t = 0)$ . Choose short  $\Delta t$
2. Compute forces  $\vec{F} = -\nabla V(\vec{r}(t))$  and  $\vec{a} = \vec{F}/m$
3. Move atoms  $\vec{r}(t + \Delta t) \leftarrow \vec{r}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}\Delta t^2 + \dots$
4. Advance time  $t \leftarrow t + \Delta t$
5. Go to 2.



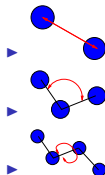
## Molecular dynamics - algorithms 2

- ▶ Force field parameterised against higher-level theory (QC).
- ▶ Many force fields. Charmm, Amber, parameterised for biomolecular simulation. Similar functional form:

## Molecular dynamics - algorithms 2

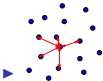
- ▶ Force field parameterised against higher-level theory (QC).
- ▶ Many force fields. Charmm, Amber, parameterised for biomolecular simulation. Similar functional form:

$$\begin{aligned}
 V(\vec{r}(t)) = & \sum_{\text{bonds}} \frac{1}{2} (r - r_0)^2 \\
 & + \sum_{\text{angles}} \frac{1}{2} k_a (\theta - \theta_0)^2 \\
 & + \sum_{\text{torsions}} \frac{1}{2} V_n [1 + \cos(\eta\omega - \gamma)] \\
 & + \sum_{j=1}^{N-1} \sum_{i=j+1}^N \left( \epsilon_{ij} \left[ \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{12} - 2 \left( \frac{\sigma_{ij}}{r_{ij}} \right)^6 \right] + \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}} \right)
 \end{aligned}$$



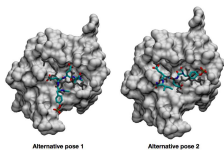
## Molecular dynamics - algorithms 2

- ▶ Force field parameterised against higher-level theory (QC).
- ▶ Many force fields. Charmm, Amber, parameterised for biomolecular simulation. Similar functional form:

$$\begin{aligned}
 V(\vec{r}(t)) = & \sum_{\text{bonds}} \frac{1}{2} (r - r_0)^2 \\
 & + \sum_{\text{angles}} \frac{1}{2} k_a (\theta - \theta_0)^2 \\
 & + \sum_{\text{torsions}} \frac{1}{2} V_n [1 + \cos(\eta\omega - \gamma)] \\
 & + \sum_{j=1}^{N-1} \sum_{i=j+1}^N \left( \epsilon_{ij} \left[ \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{12} - 2 \left( \frac{\sigma_{ij}}{r_{ij}} \right)^6 \right] + \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}} \right)
 \end{aligned}$$


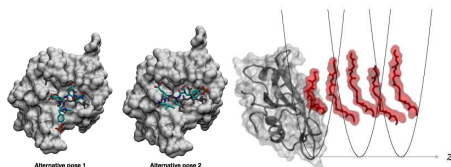
## Hot topics in protein-ligand modelling

- ▶ Prediction of binding sites and modes



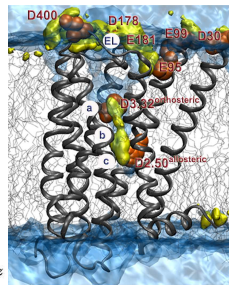
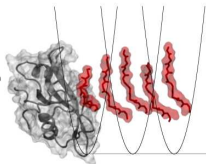
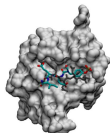
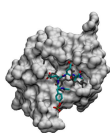
## Hot topics in protein-ligand modelling

- ▶ Prediction of binding sites and modes
- ▶ Calculation of binding affinities and kinetics



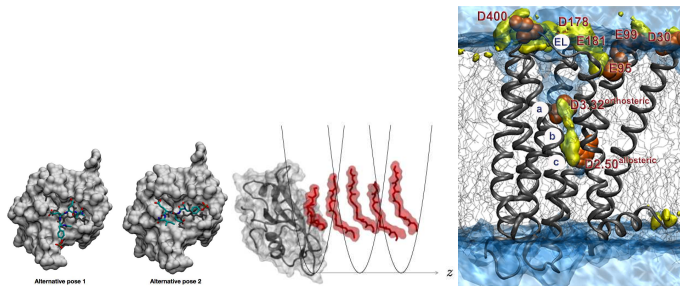
## Hot topics in protein-ligand modelling

- ▶ Prediction of binding sites and modes
- ▶ Calculation of binding affinities and kinetics
- ▶ Understanding allosteric modulation



## Hot topics in protein-ligand modelling

- ▶ Prediction of binding sites and modes
- ▶ Calculation of binding affinities and kinetics
- ▶ Understanding allosteric modulation



- ▶ All require modelling solvated single-protein (<100k atoms) systems → *strongly-scaling problem*
- ▶ All require extensive  $O(10 - 100\mu s)$  sampling for statistical significance → *requires ensembles of very long simulations*

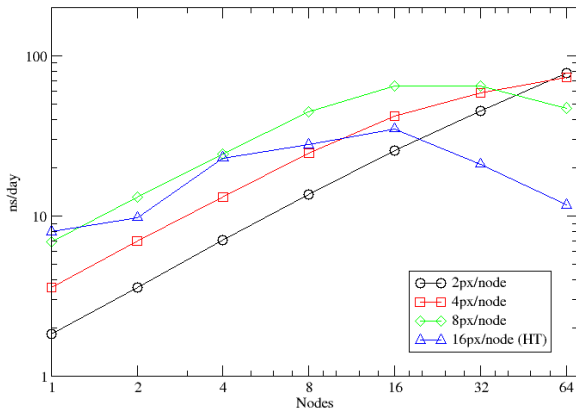
## Molecular dynamics - computational cost

- ▶  $\Delta t \approx 1\text{fs}$ , biologically relevant timescales  $> 1\mu\text{s}$
- ▶  $\rightarrow$  simulations require  $10^9$  iterations.
- ▶  $\rightarrow$  to compute a single  $1\mu\text{s}$  trajectory in 1 week requires an iteration time of  $\approx 3\text{ms}$  ( $\approx 140\text{ns/day}$ ) .
- ▶ Many parallel MD applications (NAMD, Gromacs, LAMMPS). How do they perform?

## NAMD performance

### NAMD on SGI ICE (Xeon 5570)

DHFR 23k atoms (dt=2fs)



- ▶ State-of-the-art machine, scaling only achievable by undersubscribing nodes → maximise IO/process. **>128 nodes (256px)** reqd for target rate.
- ▶ At peak efficiency, could do  $1\mu\text{s}$  in 2.5 weeks on 16 nodes (128px).
- ▶ Or  $1\mu\text{s}$  in 1 week in 16 single-node sims.

## Molecular dynamics - computational cost

- ▶ From NAMD performance, would need 15x single-node performance to reach target rate.
- ▶ Possible to port parts of existing code to GPU?

## Molecular dynamics - computational cost

- ▶ From NAMD performance, would need 15x single-node performance to reach target rate.
- ▶ Possible to port parts of existing code to GPU?
- ▶ Nonbonded force term computation 85% of total time/iteration → only off-loading NB would limit speed-up to 6x
- ▶ System state  $O(MB)$  - would take  $O(ms)$  to copy to/from GPU. Total time budget for iteration is 3ms.

## Molecular dynamics - computational cost

- ▶ From NAMD performance, would need 15x single-node performance to reach target rate.
- ▶ Possible to port parts of existing code to GPU?
- ▶ Nonbonded force term computation 85% of total time/iteration → only off-loading NB would limit speed-up to 6x
- ▶ System state  $O(MB)$  - would take  $O(ms)$  to copy to/from GPU. Total time budget for iteration is 3ms.
- ▶ Clear that whole simulation must be done on the GPU.

## MD on GPUs – ACEMD

ACEMD molecular dynamics package

- ▶ Written from scratch for NVIDIA GPUs using CUDA.

## MD on GPUs – ACEMD

ACEMD molecular dynamics package

- ▶ Written from scratch for NVIDIA GPUs using CUDA.
- ▶ Designed for maximum performance on a single GPU.
- ▶ Optimised for  $< 100k$  atom systems. *Hand-optimised for NV G200/GF100*

## MD on GPUs – ACEMD

### ACEMD molecular dynamics package

- ▶ Written from scratch for NVIDIA GPUs using CUDA.
- ▶ Designed for maximum performance on a single GPU.
- ▶ Optimised for  $< 100k$  atom systems. *Hand-optimised for NV G200/GF100*
- ▶ Implementation details in:
  - ▶ M. J. Harvey, G. Giupponi, G. De Fabritiis, *ACEMD: Accelerated molecular dynamics simulation in the microsecond timescale*, J. Chem. Theor. & Comput. 5, 1632 (2009)
  - ▶ M. J. Harvey, G. De Fabritiis, *An implementation of the smooth particle-mesh Ewald (PME) method on GPU hardware*, J. Chem. Theor. & Comput., 5, 2371-2377 (2009)

## MD on GPUs – ACEMD

### ACEMD molecular dynamics package

- ▶ Written from scratch for NVIDIA GPUs using CUDA.
- ▶ Designed for maximum performance on a single GPU.
- ▶ Optimised for  $< 100k$  atom systems. *Hand-optimised for NV G200/GF100*
- ▶ Implementation details in:
  - ▶ M. J. Harvey, G. Giupponi, G. De Fabritiis, *ACEMD: Accelerated molecular dynamics simulation in the microsecond timescale*, J. Chem. Theor. & Comput. 5, 1632 (2009)
  - ▶ M. J. Harvey, G. De Fabritiis, *An implementation of the smooth particle-mesh Ewald (PME) method on GPU hardware*, J. Chem. Theor. & Comput., 5, 2371-2377 (2009)
- ▶ Feature complete for production MD simulations
  - ▶ NVT, NPT ensembles
  - ▶ Energy minimisation
  - ▶ PME, GRF electrostatics
  - ▶ Shake constraints
  - ▶ H-mass repartitioning ( $dt = 4fs$ ) Feenstra et al, J. Comp. Chem. (1999)
  - ▶ Charmm, Amber force fields
  - ▶ Scriptable + plugin interface
  - ▶ PDB, DCD files
- ▶ In production since 2008.

## ACEMD performance

| Device          | Num | ms/step | ns/day | Relative performance |
|-----------------|-----|---------|--------|----------------------|
| GTX 480         | 1   | 4.45    | 74     | 1.00                 |
|                 | 3   | 2.46    | 140    | 1.89                 |
| M2050 (ECC on)  | 3   | 4.85    | 71     | 0.98                 |
| M2050 (ECC off) | 3   | 3.3     | 105    | 1.35                 |

Dihydrofolate Reductase benchmark. 23k atoms, NVT,  $dt = 4fs$  using H-mass repartitioning

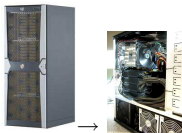
## ACEMD performance

| Device          | Num | ms/step | ns/day | Relative performance |
|-----------------|-----|---------|--------|----------------------|
| GTX 480         | 1   | 4.45    | 74     | 1.00                 |
|                 | 3   | 2.46    | 140    | 1.89                 |
| M2050 (ECC on)  | 3   | 4.85    | 71     | 0.98                 |
| M2050 (ECC off) | 3   | 3.3     | 105    | 1.35                 |

Dihydrofolate Reductase benchmark. 23k atoms, NVT,  $dt = 4fs$  using H-mass repartitioning

- ▶ GPUs so fast that strong scaling effect is savage. (Parallel scheme is simple, req good GPU bandwidth, fast CPU)
- ▶ A 3-GPU workstation can produce  $1\mu s/week$ , roughly equivalent to NAMD on **128 nodes** (256px) of SGI ICE.

Simulations that would previously have been a major undertaking on expensive HPC machines can now be performed routinely on a workstation.



# GPUGRID

- ▶ Now we have a fast MD code, where can we run it?
- ▶ Ca 2008 academic interest in GPUs just starting. No GPU resources in HPC centres, all out in desktops and workstations.

→ developed GPUGRID distributed computing project.

# GPUGRID

- ▶ Now we have a fast MD code, where can we run it?
- ▶ Ca 2008 academic interest in GPUs just starting. No GPU resources in HPC centres, all out in desktops and workstations.

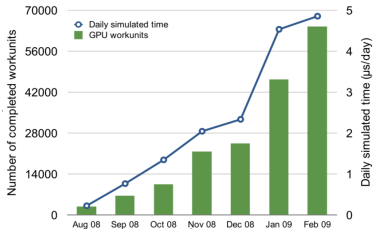
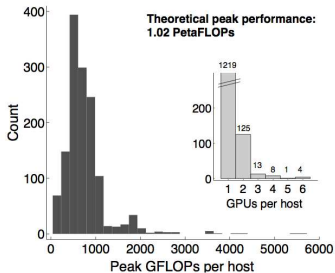
→ developed GPUGRID distributed computing project.

- ▶ Uses BOINC middleware (SETI@Home)
- ▶ Volunteers contribute time on their computers. Client common to other projects.
- ▶ Additional workflow tools make it as easy to use as conventional cluster (cf PBS, SGE commands)

T. Giorgino, M. J. Harvey and G. De Fabritiis, *Distributed computing as a virtual supercomputer: Tools to run and manage large-scale BOINC simulations*, Comp. Phys. Commun. 181, 1402 (2010)

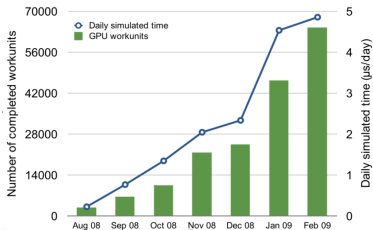
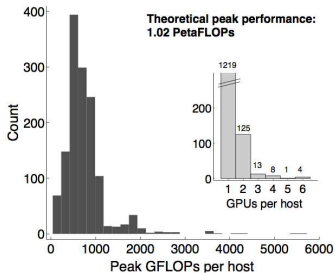


## GPUGRID resources



- ▶ The typical GPUGRID client is a single-GPU machine running Windows XP with a mid- to high-end G200 or GF100-class GPU and a quad core CPU.
- ▶ There are 1250 GPU-worth of clients attached at any one time.
- ▶ (As of Feb 09), roughly equivalent to having a **500 node** SGI ICE dedicated to NAMD.

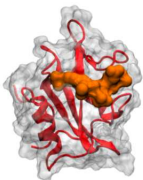
## GPUGRID resources



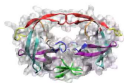
- ▶ The typical GPUGRID client is a single-GPU machine running Windows XP with a mid- to high-end G200 or GF100-class GPU and a quad core CPU.
- ▶ There are 1250 GPU-worth of clients attached at any one time.
- ▶ (As of Feb 09), roughly equivalent to having a **500 node** SGI ICE dedicated to NAMD.
- ▶ If these were dedicated GPUs, sampling rate would be 10x greater.
  - ▶ Reduce failures, data transfer, optimise for best GPUs.
  - ▶ Could run parallel jobs.

## GPUGRID results

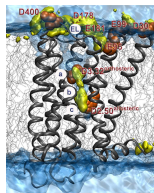
| System                  | Atoms  | Data (2009-10) | Publication                              |
|-------------------------|--------|----------------|--|
| SH2 domain/ligand       | 39,000 | 905 $\mu$ s    | I. Buch, J. Chem. Inf. Mod 50 397 (2010) |
| HIV-1 protease          | 56,000 | 230 $\mu$ s    | K. Sadiq, Proteins 78 2873 (2010)        |
| GPCR D2 receptor        | 60,000 | 69 $\mu$ s     | J. Selent, PLOS Comp. Biol. 6 (2010)     |
| $\beta$ -trypsin/ligand | 35,000 | 30 $\mu$ s     |  |



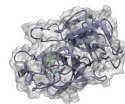
SH2-ligand binding affinities



Novel HIV Protease dynamics



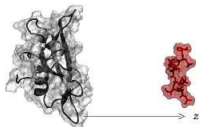
D2 GPCR allosteric modulation



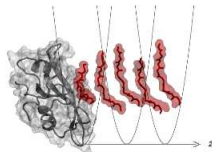
Trypsin-benzamidinium complex

Effective throughput of GPUGRID:  $8\mu$ s/day

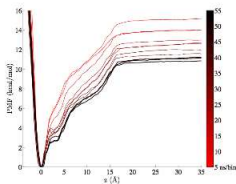
## Example - Binding affinity calculation



1. Generation of initial configurations



2. Umbrella sampling



3. Potential of mean force reconstruction\*

$$\Delta G^o = \Delta G_{PMF} - k_B T \ln\left(\frac{V_b A_{u,R}}{V_o}\right) + \Delta G_R$$

4. Standard free energy of binding\*\*

\*Roux B, *Comput Phys Commun* (1995)

\*\*Doudou S et al, *J Chem Theory and Comput* (2009)

## Next steps

- ▶ ACEMD is a very efficient MD code.
- ▶ GPUGRID is a very inefficient way of using it.

## Next steps

- ▶ ACEMD is a very efficient MD code.
- ▶ GPUGRID is a very inefficient way of using it.
- ▶ Increase GPUGRID capacity
  
- ▶ Deploy ACEMD on HPC centre GPU resources.

## Next steps

- ▶ ACEMD is a very efficient MD code.
- ▶ GPUGRID is a very inefficient way of using it.
- ▶ Increase GPUGRID capacity
  - ▶ Volunteers self-selecting, numbers stable.
  - ▶ Use AMD(ATI) GPUs. Radeon HD 5000 series. *On paper just as powerful as NVIDIA cards*
  - ▶ Multicore CPUs *We could these for short simulations*
- ▶ Deploy ACEMD on HPC centre GPU resources.

## Next steps

- ▶ ACEMD is a very efficient MD code.
- ▶ GPUGRID is a very inefficient way of using it.
- ▶ Increase GPUGRID capacity
  - ▶ Volunteers self-selecting, numbers stable.
  - ▶ Use AMD(ATI) GPUs. Radeon HD 5000 series. *On paper just as powerful as NVIDIA cards*
  - ▶ Multicore CPUs *We could these for short simulations*
- ▶ Deploy ACEMD on HPC centre GPU resources.
  - ▶ Still don't really exist! But centres starting to ask what makes a good GPU offering.
  - ▶ 8+ core nodes now (almost) ubiquitous.

## Next steps

- ▶ ACEMD is a very efficient MD code.
- ▶ GPUGRID is a very inefficient way of using it.
- ▶ Increase GPUGRID capacity
  - ▶ Volunteers self-selecting, numbers stable.
  - ▶ Use AMD(ATI) GPUs. Radeon HD 5000 series. *On paper just as powerful as NVIDIA cards*
  - ▶ Multicore CPUs *We could these for short simulations*
- ▶ Deploy ACEMD on HPC centre GPU resources.
  - ▶ Still don't really exist! But centres starting to ask what makes a good GPU offering.
  - ▶ 8+ core nodes now (almost) ubiquitous.
- ▶ Look for a way to use different GPUs and multicore CPUs without rewriting ACEMD code → **Improve capacity**
- ▶ Improve parallel performance of ACEMD → **Improve capability**

## ACEMD Porting Options

For porting a CUDA code to other platforms, there are 2 options:

- ▶ MCUDA <http://impact.crhc.illinois.edu/mcuda.php>
  - ▶ Translates CUDA for multicore x86 CPUs.
  - ▶ No good for other GPUs.
  - ▶ Only recently available.
  - ▶ *Stop press*: PGI have announced similar
- ▶ OpenCL
  - ▶ Multivendor-support, including NVIDIA, AMD, Intel (the important ones)
  - ▶ Applicable to GPUs and CPUs
  - ▶ Similar to CUDA but not API-compatible. Requires recoding.
- ▶ CAPS HMPP
  - ▶ CUDA or OpenCL targets.
  - ▶ Preprocessor directive based.
  - ▶ Assumes starting from C or Fortran code.

## ACEMD Porting Options

For porting a CUDA code to other platforms, there are 2 options:

- ▶ MCUDA <http://impact.crhc.illinois.edu/mcuda.php>
  - ▶ Translates CUDA for multicore x86 CPUs.
  - ▶ No good for other GPUs.
  - ▶ Only recently available.
  - ▶ *Stop press*: PGI have announced similar
- ▶ OpenCL
  - ▶ Multivendor-support, including NVIDIA, AMD, Intel (the important ones)
  - ▶ Applicable to GPUs and CPUs
  - ▶ Similar to CUDA but not API-compatible. Requires recoding.
- ▶ CAPS HMPP
  - ▶ CUDA or OpenCL targets.
  - ▶ Preprocessor directive based.
  - ▶ Assumes starting from C or Fortran code.

OpenCL seems the best option (for now).

## Porting ACEMD to OpenCL

## OpenCL - What is it?

OpenCL is a standard promoted by Khronos Group (of OpenGL fame), supported by ATI, NVidia

- ▶ Host C API for controlling and interacting with GPU/Accelerator devices
- ▶ A C language for writing device kernels
- ▶ An abstract device model that maps very well to NVIDIA and AMD hardware (surprise!) can also maps on to CPUs too.
- ▶ Implementations available for NVIDIA, AMD GPUs, x86 CPUs (AMD, Fixstars, Intel *tba*), Cell/Power (IBM)

## CUDA vs OpenCL models

| CUDA term  | OpenCL term        |
|--|--------------------|
| GPU  | Device             |
| Multiprocessor                                       | Compute Unit       |
| Scalar core  | Processing element |
| Global memory  | Global memory      |
| Shared (per-block) memory                            | Local memory       |
| Local memory (automatic, or <code>__local__</code> ) | Private memory     |
| kernel   | program            |
| block  | work-group         |
| thread   | work item          |

- ▶ Syntactic differences in kernel code
- ▶ C host-side API like CUDA C API
- ▶ Nothing like the CUDA language extensions!

## OpenCL GPU devices

- ▶ Only two that matter. NVIDIA Fermi and AMD(ATI) Evergreen:

|                         | NVIDIA (Fermi)          | AMD (Evergreen) |
|-------------------------|-------------------------|-----------------|
| Compute Units           | 15                      | 20              |
| PE/CU (warp size)       | 32 (32)                 | 16 (64)         |
| PE arch                 | GF100: In-order, scalar | 5-way VLIW      |
| Clock                   | 1.4GHz                  | 0.85GHz         |
| Peak sp FMA GFLOPS (dp) | 1344 (672)              | 2720 (544)      |
| Peak memory BW (GB/s)   | 177                     | 155             |

- ▶ On paper, Evergreen looks like it should outperform Fermi.
- ▶ Also: Intel Many Integrated Cores (MIC/Knight's Ferry, etc).

## OpenCL CPU devices

OpenCL device model also maps on to CPUs:

|                  | NVIDIA<br>Fermi  | AMD<br>Evergreen | Intel<br>Westmere | Intel<br>Sandy Bridge+ |
|------------------|------------------|------------------|-------------------|------------------------|
| CU               | 15               | 20               | 6(12)             | 8+                     |
| PE/CU            | 32               | 16               | 1/2/4             | 1/2/4/8                |
| 'warp'           | (32)             | (64)             | 1/SSE width       | 1/AVX width            |
| PE arch          | In-order, scalar | 5-way VLIW       | OoO vector        | OoO vector             |
| Clock (GHz)      | 1.4              | 0.85             | 2.93              | 2.3-3.8                |
| GFLOPS           | FMA              | FMA              | MUL+ADD           | MUL+ADD                |
| sp               | 1344             | 2720             | 140               | <256                   |
| dp               | 672              | 544              | 70                | <128                   |
| Memory BW (GB/s) | 177              | 155              | 16                | >16                    |

- ▶ CPUs often deployed in 2-4 socket NUMA systems → common memory makes it easy to use multiple CPUs, unlike GPUs.
- ▶ CPU cores much more sophisticated (Out-of-order execution) – higher ILP
- ▶ No scratchpad/local memory, but big caches. More versatile memory subsystem, even if lower peak.
- ▶ Not unreasonable to see convergence with GPUs in near future. (AMD Fusion, Sandy Bridge GPU, Is MIC a “GPU”?)
- ▶ *OpenCL could be a unifying programming model.*

## From CUDA to OpenCL

Given the similarities it is possible to:

- ▶ Abstract the differences between CUDA (C API) and OpenCL API
- ▶ Use simple source-to-source translation to convert kernels
- ▶ Automatically generate source code for kernel entry point functions – Need to make a tool that does explicitly what `nvcc` does during a compilation

## From CUDA to OpenCL

Given the similarities it is possible to:

- ▶ Abstract the differences between CUDA (C API) and OpenCL API
- ▶ Use simple source-to-source translation to convert kernels
- ▶ Automatically generate source code for kernel entry point functions – Need to make a tool that does explicitly what `nvcc` does during a compilation

**Swan** - runtime library (CUDA or OpenCL) and compile-time translator for kernels.  
Available from <http://www.multiscalelab.org/swan> (GPL).

## SWAN CUDA to OpenCL translation

- ▶ Perl Regular expression based, eg: `s/threadIdx.x/get_local_id(0)/g`
- ▶ CUDA or OpenCL selectable at compile time. Compiles the kernel with `nvcc` (no translation needed) or OpenCL compiler.
- ▶ Generates entry points from code analysis.

### Pros:

- ▶ Removes all CUDA-specific host-side code.
- ▶ Supports optimised CUDA kernels for different device types.
- ▶ OpenCL code (re)compiled at runtime. Requires no knowledge of device at program compilation time.
- ▶ (almost) no changes required to kernel code.

### Cons:

- ▶ Requires some re-writing some host side code.
- ▶ `kernel<<< grid, block>>( ... )`  
becomes `k_kernel( grid, block, ... )`
- ▶ replace CUDA API calls with Swan API equivalents
- ▶ Doesn't support CUDA templates.

## Swan usage example

```
__kernel__ void func(...) {  
}  
void host_func(...) {  
func <<< grid, block, shmem >>>( ... );  
}
```

```
#include <kernel.kh>  
void host_func(...) {  
k_func( grid, block, shmem, ... );  
}
```

▶ *or*

## Swan usage example

```
__kernel__ void func(...) {  
}  
void host_func(...) {  
func <<< grid, block, shmem >>>( ... );  
}
```

- ▶ `nvcc -o a.out program.cu`

```
#include <kernel.kh>  
void host_func(...) {  
k_func( grid, block, shmem, ... );  
}
```

- ▶ `swan -cuda kernel.kh kernel.cu`
- ▶ *or*
- ▶ `swan -opencl kernel.kh kernel.cu`
- ▶ `gcc -o a.out program.c -I. -lswan`

## Performance of OpenCL ACEMD – NVIDIA

| Device      | CUDA ms/step | OpenCL ms/step | Relative Perf |
|-------------|--------------|----------------|---------------|
| GTX 480     | 4.6          | 11.4           | 0.4           |
| Tesla C1060 | 9.5          | 21.9           | 0.4           |

DHFR 23k atoms, generic FFT, Cuda 3.1, Centos 5.5

- ▶ Nvidia OpenCL performance  $\approx$  0.4 of CUDA. Worst on performance-tuned kernels.
- ▶ The CUDA and OpenCL compilers have different back-ends.
- ▶ OpenCL compiler produces less efficient code:
  - ▶ Each CU has a (small) register file shared by all threads. The registers/thread limits the usage (occupancy) of the device.
  - ▶ Forcing a reg/thread limit improves occupancy, but causes spills to device memory
- ▶ Compiler bug – constant memory can't be used ( $\approx$  10% penalty)

## Performance of OpenCL ACEMD – AMD

| Device      | CUDA ms/step | OpenCL ms/step | Relative Perf                                      |
|-------------|--------------|----------------|--|
| GTX 480     | 4.6          | 11.4           | 0.4  |
| Tesla C1060 | 9.5          | 21.9           | 0.4  |
| Radeon 5850 |              | 24.3           | 0.4 (Tesla C1060 CUDA)<br>0.9 (Tesla C1060 OpenCL) |

DHFR 23k atoms, generic FFT, AMD OpenCL 2.2, Centos 5.5

- ▶ AMD Radeon performance comparable with Tesla C1060 OpenCL.
- ▶ **ILP**. The Evergreen is a 5-issue VLIW. Needs code with plenty of instruction-level parallelism to fill it up. Can't interleave ALU and memory operations.
- ▶ **Compiler optimisation** Inspection of the assembly emitted by the compiler (R800 ISA) reveals inefficient code generation.
- ▶ **Compiler immaturity** SDK 2.2 is the first of 3 public releases to correctly compile and run ACEMD (> 10 bugs)
- ▶ **Implementation immaturity** Global memory accesses perform poorly (caches not used), high kernel launch overhead (> 120 $\mu$ s vs < 10 $\mu$ s CUDA, cf Markall)
- ▶ **No profiling** CUDA runtime profiling (CUDA\_PROFILE) immensely useful in optimising ACEMD. No runtime profiling/performance counters accessible on AMD.
  - ▶ But: unlike Nvidia, can see the actual device assembly - essential for checking VLIW efficiency.

## Performance of OpenCL ACEMD – Intel

| Device                   | CUDA ms/step | OpenCL ms/step | Relative Perf   |
|--------------------------|--------------|----------------|-----------------|
| GTX 480                  | 4.6          | 11.4           | 0.4             |
| Tesla C1060              | 9.5          | 21.9           | 0.4             |
| Xeon X5670 (HT:24 cores) |              | 123.8          | 0.04            |
|                          |              | 40             | 0.1 (tuned PME) |

DHFR 23k atoms, generic FFT, AMD OpenCL 2.2, Xeon X5670, HT enabled, Turbo disabled, SLES 11.1

- ▶ For reference: NAMD on all cores:  $17\text{ms}/\text{step}$  → OpenCL rel perf 0.4.
- ▶ **Poor runtime** 24 logical cores in the system, process load  $\approx 16$  whilst ACEMD running, should have been higher. Not clear why (on larger bm, would use only 1 cpu)
- ▶ **PME** One particular kernel highly optimised for NV hardware, fine-gained block synchronisation. Replace with *simpler* kernel, improves perf by 10x.
- ▶ **ILP** AMD OpenCL CPU compiler treats each CPU as a separate CU. Uses SSE but almost nothing to vectorise in ACEMD kernels. (all loops turned into work groups).

There's a better way to use CPUs..

## Treating a CPU core as an OpenCL CU

- ▶ OpenCL CUs contain  $\geq 1$  PE, with the expectation that all PEs execute same program (SIMD)
- ▶ CPU contains multiple cores, each can execute different program
- ▶ **Option 1** Consider a CPU as a CU and a core as a vector PE (SSE 4sp/2dp, AVX 8sp/4dp width)
  - ▶ Kernel code should be explicitly vector or be easily vectorisable (eg loop-unrolling)

## Treating a CPU core as an OpenCL CU

- ▶ OpenCL CUs contain  $\geq 1$  PE, with the expectation that all PEs execute same program (SIMD)
- ▶ CPU contains multiple cores, each can execute different program
- ▶ **Option 1** Consider a CPU as a CU and a core as a vector PE (SSE 4sp/2dp, AVX 8sp/4dp width)
  - ▶ Kernel code should be explicitly vector or be easily vectorisable (eg loop-unrolling)
- ▶ **Option 2** Consider each vector lane of SSE/AVX a separate scalar PE.
  - ▶ Run 1/2/4/8 work-items simultaneously, one in each vector lane
  - ▶ Use vector type natural to algorithm.
  - ▶ Conceptually similar to NVIDIA compute units
  - ▶ Requires smarter compiler to perform horizontal vectorisation/implicit parallelisation; no implementations do this (yet).

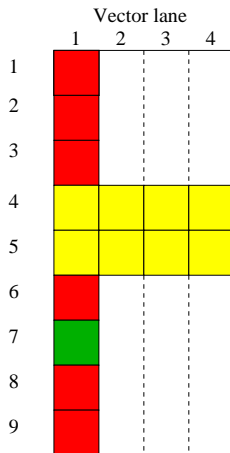
## Implicit parallelisation/Horizontal vectorisation -1

Consider the following kernel, with a mix of **scalar**, **vector** and **transcendental** functions:

```
__kernel len_scale( float4 *p1, float4 *p2, float fac, float *dot ) {  
    int i = threadIdx;  
    float4 p_1 = p1[ i ];  
    float4 p_2 = p2[ i ];  
    float4 r = p_2 - p_1;  
    float4 len2 = dot4( r, r );  
    float len = ( len2.x + len2.y +len2.z + len2.w );  
    len = sqrt( len );  
    len *= fac;  
    dot[ i ] = len;  
}
```

## OpenCL on CPUs – naïve mapping

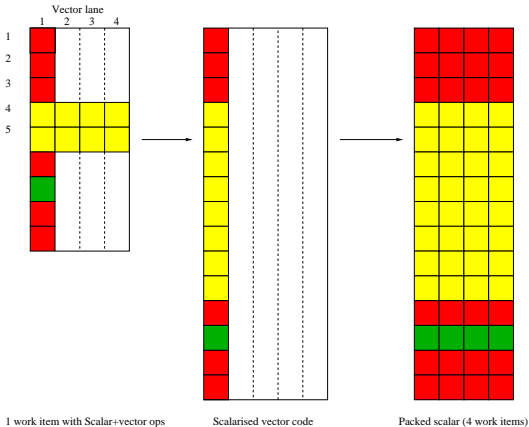
This can be mapped on to SSE instructions, with the scalar ops using a single lane:  
→ inefficient use of resources – vector unit mostly idle.  
Can we do better?



1 work item with Scalar+vector ops

## OpenCL on CPUs - Implicit parallelisation - 1

Scalarise the kernel operations and pack one WI in each vector lane  
Increase throughput by (up to) vector unit width (4x SSE, 8x AVX)  
→ A CPU is treated as a CU with 4 scalar PEs.

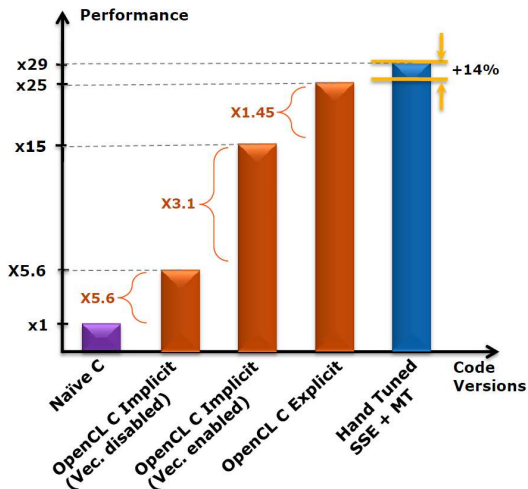


## OpenCL on CPUs - Implicit parallelisation - 2

### Problems:

- ▶ SSE fussy about alignment, destructive use of registers, limited masking (for branches, etc)
  - AVX alignment requirements relaxed, additional instructions added (*blending/conditional selection*, etc) and doubled width (256bit)
- ▶ Additional compiler complexity. Neither AMD's nor Fixstars' does this. Intel's (unreleased) OpenCL does ...

## Intel OpenCL implicit vs explicit vectorisation



- ▶ NBody kernel
- ▶ 3.1x performance increase (cf 4x max) over vec. *Inline with expectations - 4x from vectorisation, minus some for scalarisation*
- ▶ explicit vec. still faster (almost as good as hand-tuned), but of more complex source.

Test on i7 975. *Optimizing OpenCL on CPUs*, Ofer Rosenberg, OpenCL BOF, SIGGRAPH 2010

## OpenCL Summary

- ▶ On real-world code, OpenCL much (60%) slower than CUDA.
- ▶ NVIDIA and AMD performance with OpenCL code almost at parity for ACEMD (C1060 vs 5850)
- ▶ OpenCL programs can be run on multicore CPUs with AMD OpenCL
  - ▶ Runtime management of work-groups has overhead.
  - ▶ Kernels with fine-grained synchronisation treated very poorly.
  - ▶ Doesn't do horizontal vectorisation. Would yield  $\leq 4\times$  improvement. *OpenCL ACEMD on the Xeon system could be as on Tesla C1060.*

## OpenCL Summary

- ▶ On real-world code, OpenCL much (60%) slower than CUDA.
- ▶ NVIDIA and AMD performance with OpenCL code almost at parity for ACEMD (C1060 vs 5850)
- ▶ OpenCL programs can be run on multicore CPUs with AMD OpenCL
  - ▶ Runtime management of work-groups has overhead.
  - ▶ Kernels with fine-grained synchronisation treated very poorly.
  - ▶ Doesn't do horizontal vectorisation. Would yield  $\leq 4\times$  improvement. *OpenCL ACEMD on the Xeon system could be as on Tesla C1060.*
- ▶ OpenCL could be a useful unifying programming model for GPUs and CPUs
- ▶ OpenCL Implementations immature
- ▶ Best GPU performance is still NVIDIA + CUDA
- ▶ Tools like Swan enable portability

## Conclusions

- ▶ GPUs are most useful when used to reduce a poorly scaling parallel application to serial/small- $N$  parallel program.
- ▶ NVIDIA+CUDA still gives the best performance
- ▶ OpenCL is shaping up as a credible alternative for device-agnostic (GPU/CPU) programming
  - ▶ Easy to change away from CUDA to OpenCL
  - ▶ Implementations immature
  - ▶ AMD's OpenCL performance close to parity with NVIDIA's on real-world code
- ▶ Improved implementations on AVX-supporting CPUs, CPU-GPU convergence
- ▶ For HPC services, fat nodes that maximise GPUS/host (keep GPU:CPU core ratio close to 1:1) are best for serial and parallel GPU workloads.

## Acknowledgements

### Collaborators:

- ▶ G. De Fabritiis Group, IMIM-UPF, Barcelona, Spain

### Funding:

- ▶ HPC Europa and HPC Europa2 programmes.
- ▶ Virtual Physiological Human Network of Excellence.
- ▶ Plan Nacional de España.
- ▶ Sony.
- ▶ NVIDIA.
- ▶ Acellera.

### Thanks:

- ▶ Imperial College London.
- ▶ All the GPUGRID volunteers.

Any Questions?