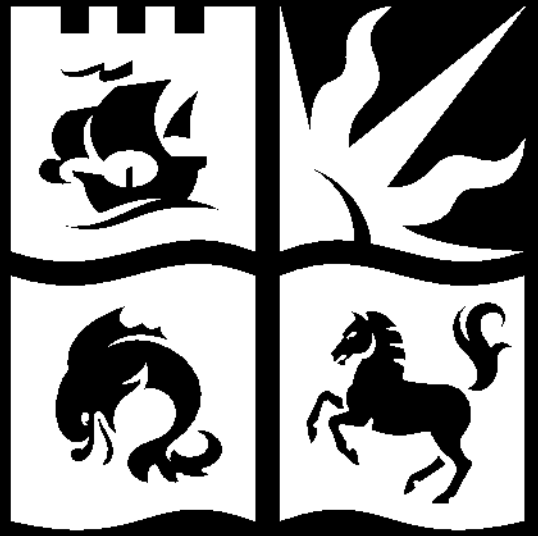
HETEROGENEOUS COMPUTING

Benedict Gaster, AMD
Lee Howes, AMD
Simon McIntosh-Smith, University of Bristol

BEYOND THE NODE

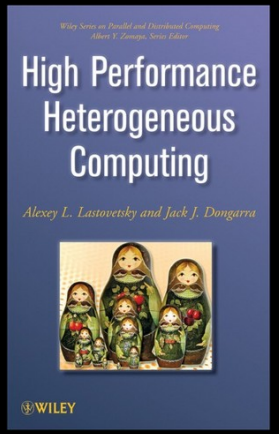


AMD

University of BRISTOL

BEYOND THE NODE

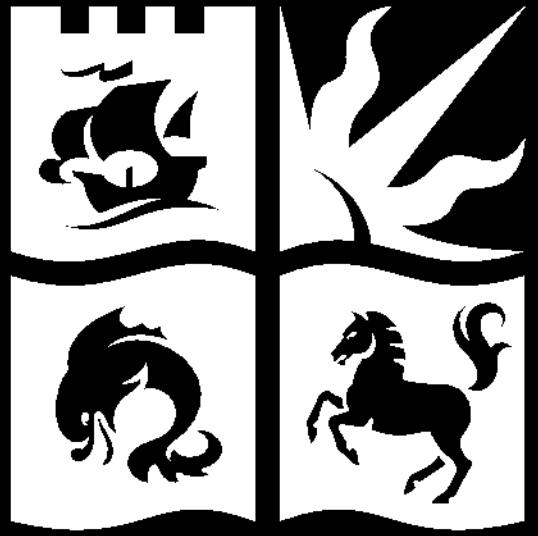
- So far have focused on heterogeneity within a node
- Many systems constructed from multiple nodes
- Easy for node types to diverge:
  - Different technologies become available over time
  - A mix of different nodes may be best to accommodate different applications
    - E.g. Compute-intensive vs. Data-intensive



- Even homogeneous hardware may behave heterogeneously
  - OS jitter, data-dependent application behavior, multi-user systems, …
- Thus heterogeneity extends right across a multi-node system

- See “High-performance heterogeneous computing” by A. Lastovetsky and J. Dongarra, 2009.

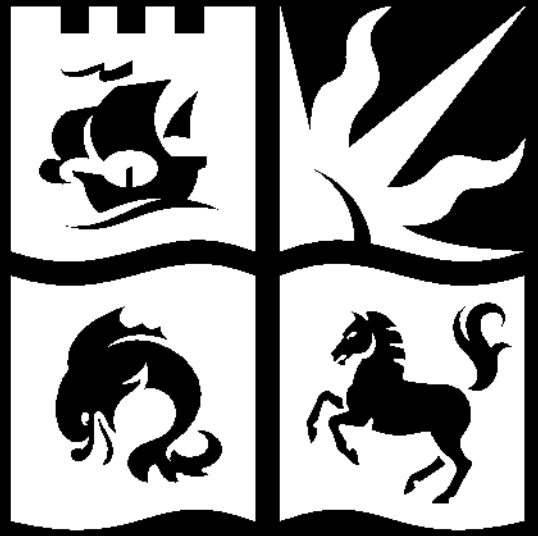
MESSAGE PASSING AND PARTITIONED GLOBAL ADDRESS SPACE PROGRAMMING



AMD

University of BRISTOL

MPI OVERVIEW



AMD

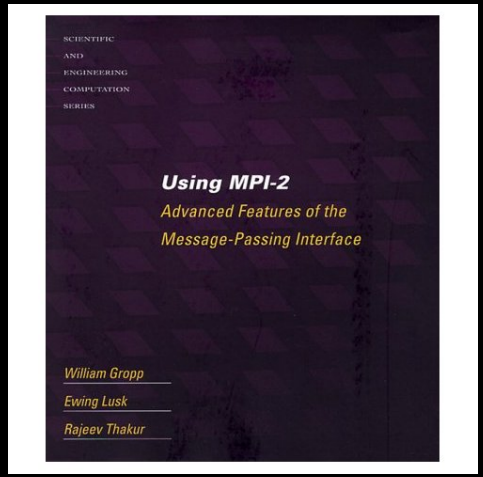
University of BRISTOL

MPI OVERVIEW

- The Message Passing Interface (MPI) has become the most widely used standard for distributed memory programming in HPC

MPI OVERVIEW

- Available for C and Fortran
- Library of functions & pre-processor Macros
- Standards:
  - MPI 1.0, June 1994 (now at MPI 1.3)
  - MPI 2.0 (now at MPI 2.2, Sep 2009)
  - MPICH (MVAPICH for Infiniband networks)
  - OpenMPI
  - Proprietary tuned... (Cray, SGI, Microsoft, ...)
- Designed to be portable (although with the usual performance caveats)



- Used on most (all?) Top500 supercomputers for multi-node applications

MPI EXAMPLE: HELLO, WORLD

```
#include "mpi.h"
int main(int argc, char* argv[])
{
  ...
  MPI_Init( &argc, &argv );
  MPI_Initialized(&flag);
  if ( flag != TRUE ) {
    MPI_Abort(MPI_COMM_WORLD,EXIT_FAILURE);
  }
  MPI_Get_processor_name(hostname,&strlen);
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  printf("Hello, world; from host %s: process %d of %d\n", \
         hostname, rank, size);
  MPI_Finalize();
}
```

COMPILING AND RUNNING MPI

Makefile:

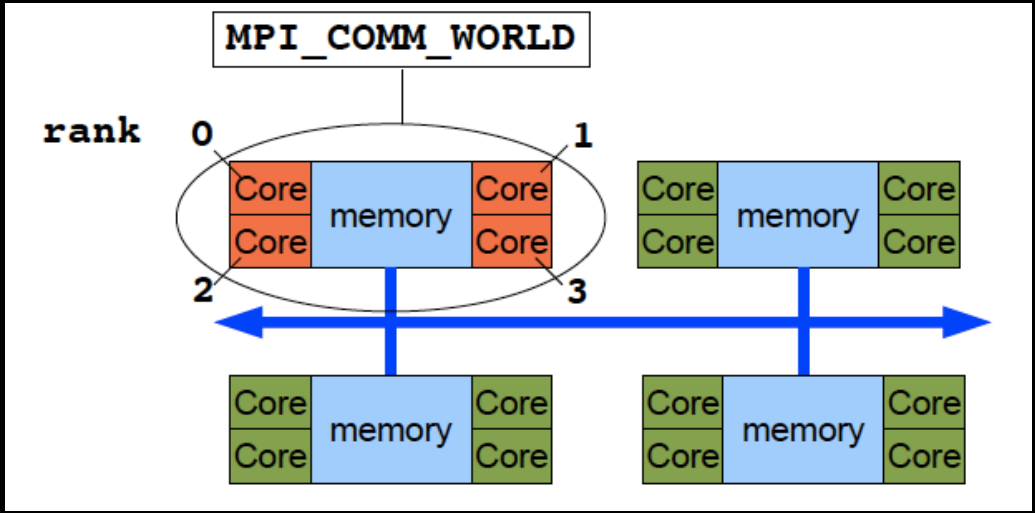
```
hello_world_c: hello_world.c
    mpicc -o $@ $^
```

mpi_submit:

```
#PBS -l nodes=1:ppn=4,walltime=00:05:00

#! Create a machine file for MPI
cat $PBS_NODEFILE > machine.file.$PBS_JOBID

numprocs=`wc $PBS_NODEFILE | awk '{ print $1 }'`

#! Run the parallel MPI executable (nodes*ppn)
mpirun -np $numprocs \
       -machinefile machine.file.$PBS_JOBID \
       $application $options
```

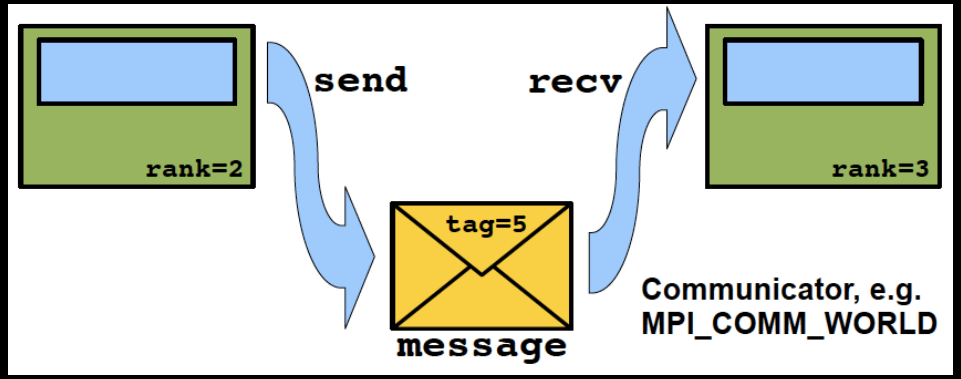
MPI COMMUNICATORS

- The process `mpirun` waits until all instances of `MPI_Init()` have acquired knowledge of the cohort



- Ranks: 0 (master), 1, 2, 3,..
- The *queuing system* decides how to distribute over nodes (servers)
- The *kernel* decides how to distribute over multi-core processors

MPI IS PRIMARILY POINT TO POINT



- Common pattern for MPI functions:
  - MPI_<function>(message, count, datatype, .., comm, flag)
- E.g.:
  - `MPI_Recv(message, BUFSIZE, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);`
- Supports both synchronous and asynchronous communication, buffered and unbuffered
- Later versions support one-sided communications
- However does support collective operations (broadcast, scatter, gather, reductions)

MPI IS LOW LEVEL

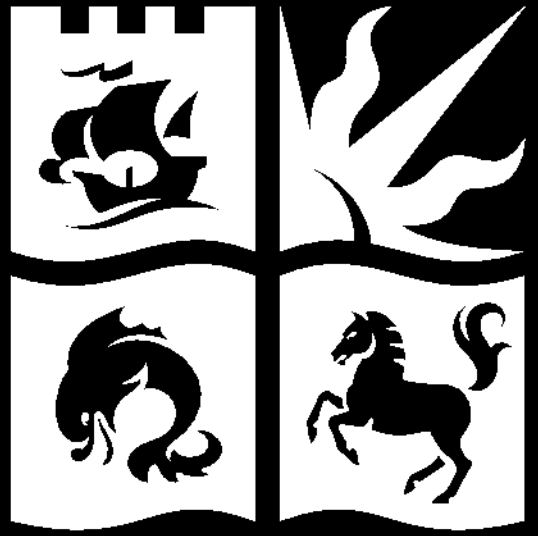
- Lets the programmer do anything they want
- Doesn't necessarily encourage good programming style
- Message passing programs are, in general, hard to design, optimise and debug
  - Challenges with deadlock, race conditions, et al.

- ***Design patterns*** can help (e.g. Mattson et al)

- Higher-level parallel programming models may use MPI underneath for optimised message passing

- Often used for homogeneous parallel structures (2D/3D grids etc), but can also be used to support heterogeneous computation, e.g. task farms with dynamic load balancing
  - MPI 2 added support for dynamic task creation

PGAS LANGUAGES OVERVIEW: UPC, CAF, CHAPEL, X10



AMD

University of BRISTOL

PARTITIONED GLOBAL ADDRESS SPACE (PGAS) LANGUAGES

- Provide shared memory-style higher-level programming on top of distributed memory computers

- Several examples:
  - Unified Parallel C
  - Co-Array Fortran
  - Titanium
  - X-10
  - Chapel

UNIFIED PARALLEL C (UPC) INTRODUCTION

- Extension of C/C++
- First released 1999, one of the more widely used PGAS languages
  - UPC support in GCC 4.5.1.2 (Oct 2010)
  - Berkeley UPC compiler 2.12 released Nov 2010
- Supported by Berkeley, George Washington University, Michigan Tech University
- Supported by vendors including Cray, IBM
- User can express data locality via "*shared*" and "*private*" address space qualifiers
- Fixed number of threads spawned across the system (no spawning)
- Lightweight coordination between threads (user responsibility)
- `upc_forall()` operator for parallelism
- Provides a hybrid, user-controlled consistency model for the interaction of memory accesses in shared memory space. Each memory reference in the program may be annotated to be either "strict" or "relaxed".

UPC EXAMPLE – MATRIX MULTIPLY

```
#include<upc.h>
#include<upc_strict.h>

shared [N*P /THREADS] int a[N][P] , c[N][M];
shared int b[P][M] ;

void main(void) {
  int i,j,l;

  upc_forall (i=0; i<N; i++; &a[i][0])
    // &a[i][0] specifies that this iteration will be executed by the thread
    // that has affinity to element a[i][0]
    for (j=0; j<M; j++) {
       c[i][j] = 0;
       for(l=0; l< P; l++) c[i][j] +=a[i][l]*b[l][j];
    }
```

CO-ARRAY FORTRAN

- An SPMD extension to Fortran 95
- Defined in 1998
- Adds a simple, explicit notation for data decomposition, similar to that used in message-passing models
- May be implemented on both shared- and distributed memory machines
- The ISO Fortran Committee include coarrays in Fortran the 2008 standard
- Adds two concepts to Fortran 95:
  - Data distribution
  - Work distribution
- Used in some important codes
  - E.g. the UK Met Office's Unified Model

AMD

University of BRISTOL

CO-ARRAY FORTRAN PROGRAMMING MODEL

- Single-Program-Multiple-Data (SPMD)

- Fixed number of processes/threads/images
  - Explicit data decomposition
  - All data is local
  - All computation is local
  - One-sided communication through co-dimensions
- Explicit synchronization

- See “An Introduction to Co-Array Fortran” by Robert W. Numrich
  - http://www2.hpcl.gwu.edu/pgas09/tutorials/caf_tut.pdf

CO-ARRAY FORTRAN WORK DISTRIBUTION

- A single Co-Array Fortran program is replicated a fixed number of times
- Each replication, called an "*image*", has its own set of data objects
- Each image executes asynchronously
- The execution path may differ from image to image
- The programmer determines the actual control flow path for the image with the help of a unique image index, using normal Fortran control constructs, and by explicit synchronizations
- For code between synchronizations, the compiler is free to use all its normal optimisation techniques, as if only one image were present

CO-ARRAY FORTRAN DATA DISTRIBUTION

- One new entity, the ***co-array***, is added to the language:

```
REAL, DIMENSION(N)[*] :: X,Y
X(:) = Y(:)[Q]
```

- Declares that each image has two real arrays of size N
- If Q has the same value on each image, the effect of this assignment statement is that each image copies the array Y from image Q and makes a local copy in array X (a broadcast)
- Array indices in parentheses follow the normal Fortran rules within one memory image
- Array indices in square brackets enable accessing objects across images and follow similar rules
- Bounds in square brackets in co-array declarations follow the rules of assumed-size arrays since co-arrays are always spread over all the images

MORE CO-ARRAY FORTRAN EXAMPLES

```
X       = Y[PE]  ! get from Y[PE]
Y[PE]   = X      ! put into Y[PE]
Y[:]    = X      ! broadcast X
Y[LIST] = X      ! broadcast X over subset of PE's in array LIST
Z(:)    = Y[:]   ! collect all Y
S = MINVAL(Y[:]) ! min (reduce) all Y
B(1:M)[1:N] = S  ! S scalar, promoted to array of shape (1:M,1:N)
```

CO-ARRAY FORTRAN MATRIX MULTIPLY

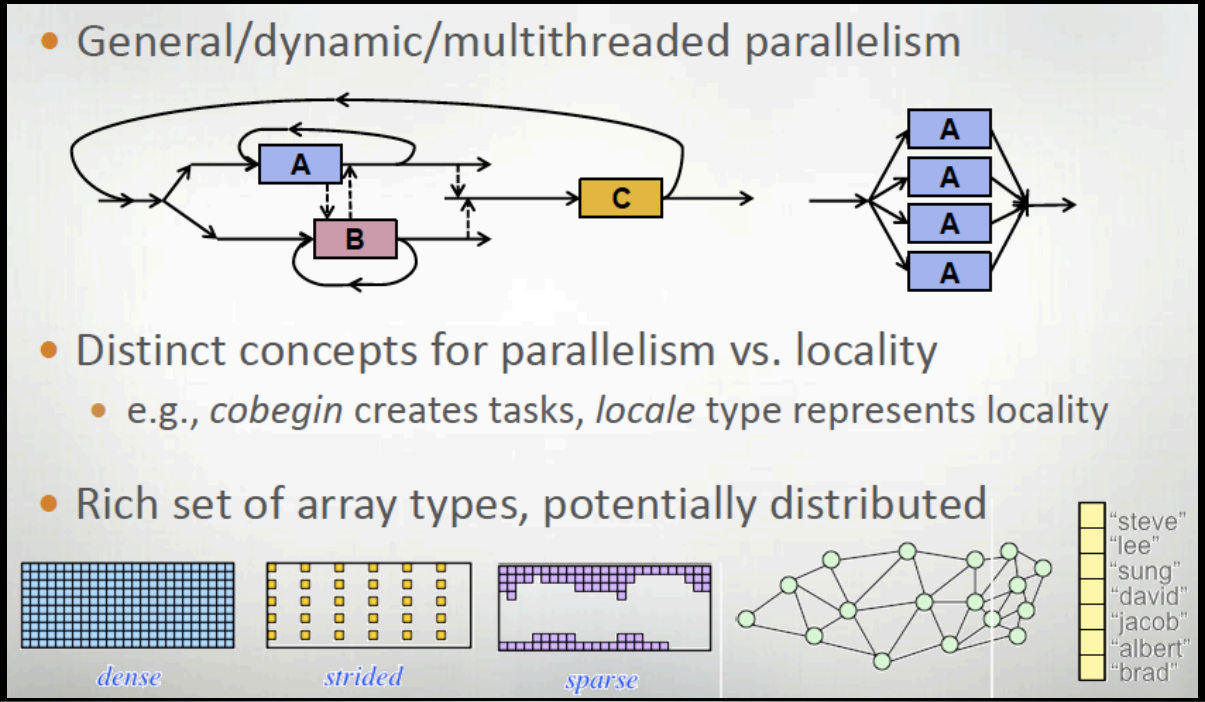
```
real,dimension(n,n)[p,*] :: a,b,c

do k=1,n
  do q=1,p
    c(i,j) = c(i,j) + a(i,k)[myP,q]*b(k,j)[q,myQ]
  enddo
enddo
```

CHAPEL

- Cray development funded by DARPA as part of the HPCS program
  - Available on Cray, SGI, Power, as well as for Linux clusters, GPU port underway
- “Chapel strives to vastly improve the programmability of large-scale parallel computers while matching or beating the performance and portability of current programming models like MPI.”
- Chapel is a clean sheet design but based on parallelism features from ZPL, High-Performance Fortran (HPF), and the Cray MTA™/Cray XMT™ extensions to C and Fortran
- Supports a multithreaded execution model with high-level abstractions for:
  - data parallelism
  - task parallelism
  - concurrency, and
  - nested parallelism.
- The `locale` type enables users to specify and reason about the placement of data and tasks on a target architecture in order to tune for locality
- Supports global-view data aggregates with user-defined implementations

CHAPEL CONCEPTS

- See: “Chapel: striving for productivity at Petascale, sanity at Exascale” by Brad Chamberlain, Dec 2011:
  - http://chapel.cray.com/presentations/ChapelForLLNL2011-presented.pdf



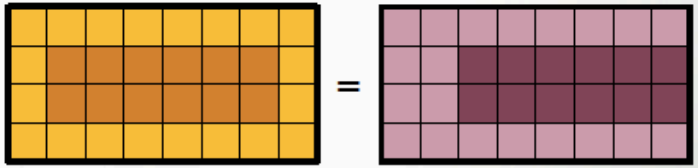
- Parallel and Serial Iteration

```
A = forall (i,j) in D do (i + j/10.0);
```

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8

- Array Slicing; Domain Algebra

```
A[InnerD] = B[InnerD+(0,1)];
```



- Promotion of Scalar Functions and Operators

```
A = B + alpha * C;
```

```
A = exp(B, C);
```

- And several other operations: indexing, reallocation, set operations, reindexing, aliasing, queries, …

X10

- Open source development by IBM, again funded by DARPA as part of HPCS
- An asynchronous PGAS (APGAS) loosely based on Java and functional languages
- Four basic principles:
  - Asynchrony
  - Locality
  - Atomicity
  - Order
- Developed on a type-safe, class-based, object-oriented foundation
- X10 implementations are available on Power, x86 clusters, on Linux, AIX, MacOS, Cygwin and Windows

X10 HELLO WORLD EXAMPLE

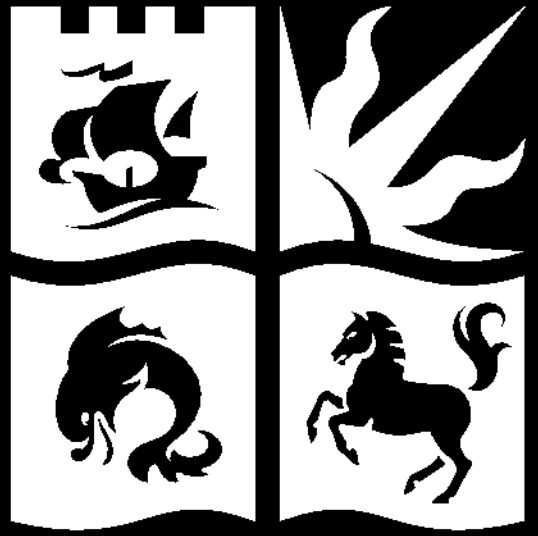
```
class HelloWholeWorld {
  public static def main(args:Array[String](1)):void {
    for (var i:Int=0; i<Place.MAX_PLACES; i++) {
      val iVal = i;
      async at (Place.places(iVal)) {
        Console.OUT.println("Hello World from place "+here.id);
      }
    }
  }
}
```

X10 FUTURE

- Looking to add support for:
  - Multiple levels of parallelism (hierarchy)
  - Fault tolerance

- Actively being supported on multiple platforms
- One of the more promissing (A)PGAS languages

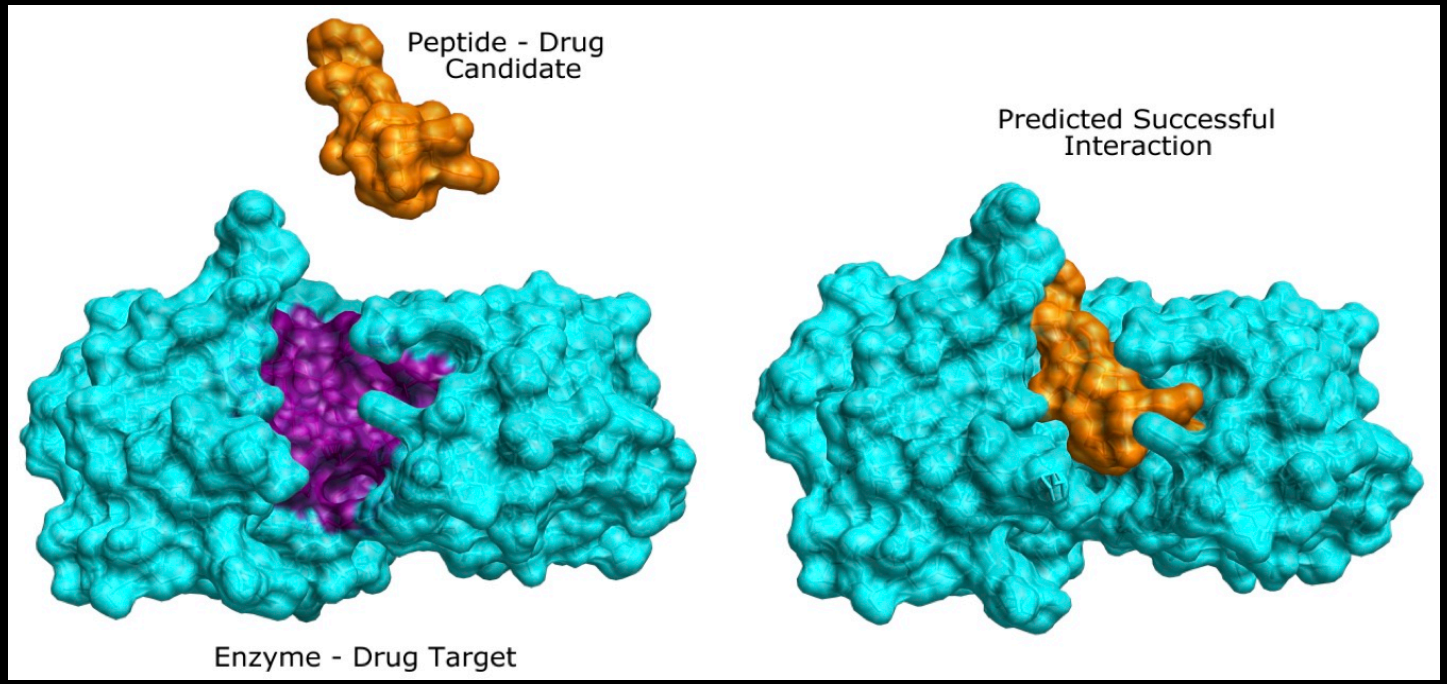
A HETEROGENEOUS EXAMPLE: MOLECULAR DOCKING USING OPENCL AND MPI



AMD

University of BRISTOL

MOLECULAR DOCKING



Proteins typically O(1000) atoms
Ligands typically O(100) atoms

EMPIRICAL FREE ENERGY FUNCTION (ATOM-ATOM)

$$\Delta G_{\text{ligand binding}} = \sum_{i=1}^{N_{\text{protein}}} \sum_{j=1}^{N_{\text{ligand}}} f(x_i, x_j)$$

Parameterised using experimental data†

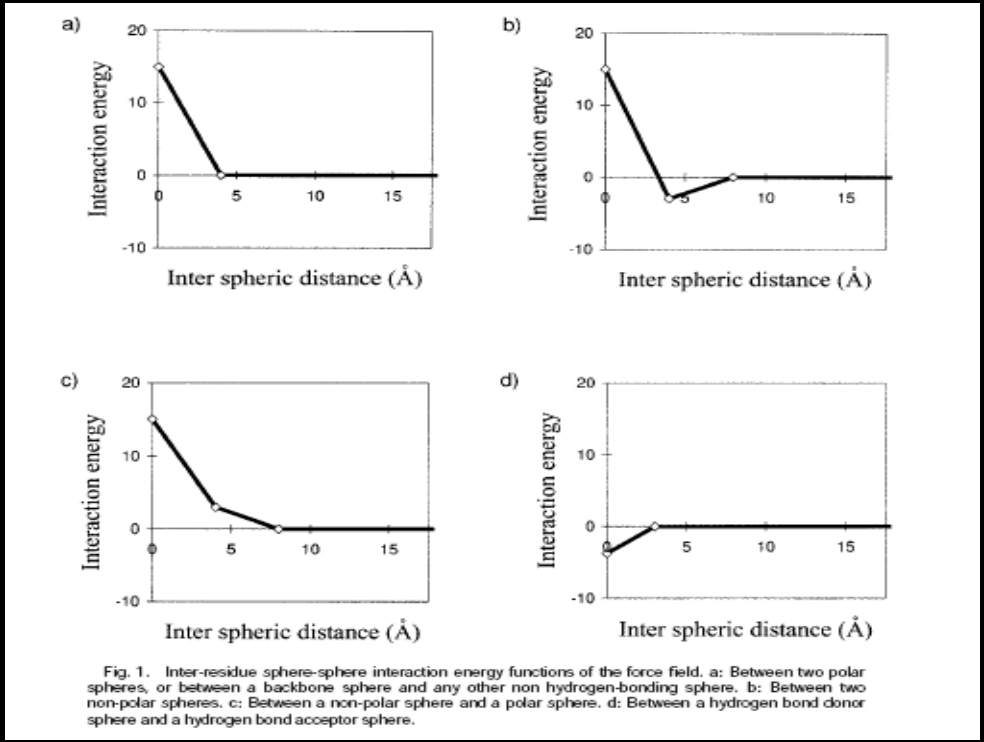


Fig. 1. Inter-residue sphere-sphere interaction energy functions of the force field. a: Between two polar spheres, or between a backbone sphere and any other non hydrogen-bonding sphere. b: Between two non-polar spheres. c: Between a non-polar sphere and a polar sphere. d: Between a hydrogen bond donor sphere and a hydrogen bond acceptor sphere.

† N. Gibbs, A.R. Clarke & R.B. Sessions, "Ab-initio Protein Folding using Physicochemical Potentials and a Simplified Off-Lattice Model", Proteins 43:186-202,2001

MULTIPLE LEVELS OF PARALLELISM

- $O(10^8)$ conformers from $O(10^7)$ ligands, all independent
- $O(10^5)$ poses per conformer (ligand), all independent
- $O(10^3)$ atoms per protein
- $O(10^2)$ atoms per ligand (drug molecule)

- Parallelism across nodes:
  - Distribute ligands across nodes using MPI – 10^7-way parallelism
  - Nodes request more work as needed – load balancing across nodes of different speeds
- Parallelism within a node:
  - All the poses of one conformer distributed across all the OpenCL devices in a node – 10^3-way parallelism
- Parallelism within an OpenCL device (e.g. a GPU, CPUs)
  - Each Work-Item (thread) performs an entire conformer-protein docking – 10^5-way parallelism
  - →10^5 atom-atom force calculations per Work-Item

BUDE'S OPENCL CHARACTERISTICS

- Single precision
- Compute intensive, not bandwidth intensive
- Very little data needs to be moved around
  - KBytes rather than GBytes!
- Very little host compute required
  - Can scale to many OpenCL devices per host

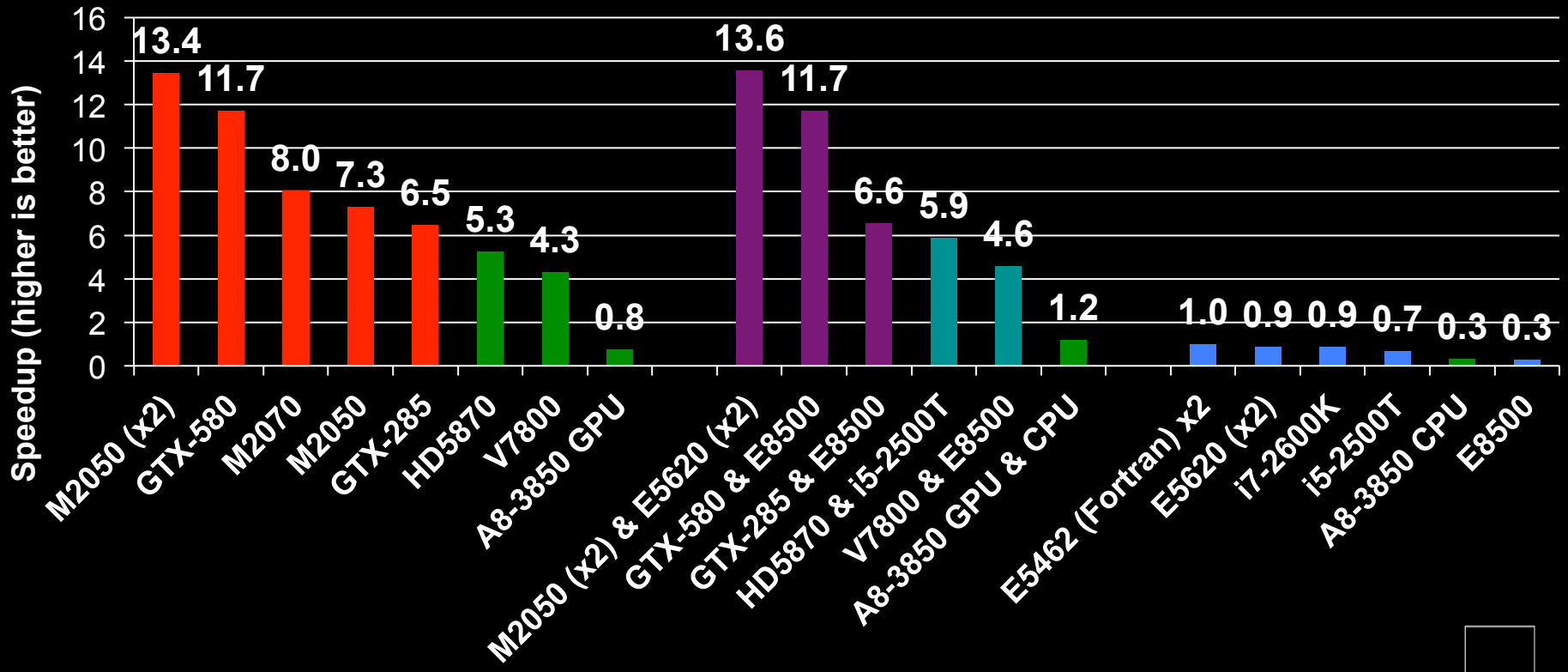
BUDE'S HETEROGENEOUS APPROACH

1. Distribute ligands across nodes, nodes request more work when ready
   - Copes with nodes of different performance and nodes dropping out
   - Can use fault tolerant MPI for this
2. Within each node, discover all OpenCL platforms/devices, including CPUs *and* GPUs
3. Run a *micro benchmark* on each OpenCL device, ideally a short piece of real work
   - Ideally use some real work so you're not wasting resource
   - Keep the microbenchmark very short otherwise slower devices penalize faster ones too much
4. Load balance across OpenCL devices using micro benchmark results
5. Re-run micro benchmark at regular intervals in case load changes within the node
   - The behavior of the workload may change
   - CPUs may become busy (or quiet)
6. Most important to keep the fastest devices busy
   - Less important if slower devices finish slightly earlier than faster ones
7. Avoid using the CPU for both OpenCL host code and OpenCL device code at the same time

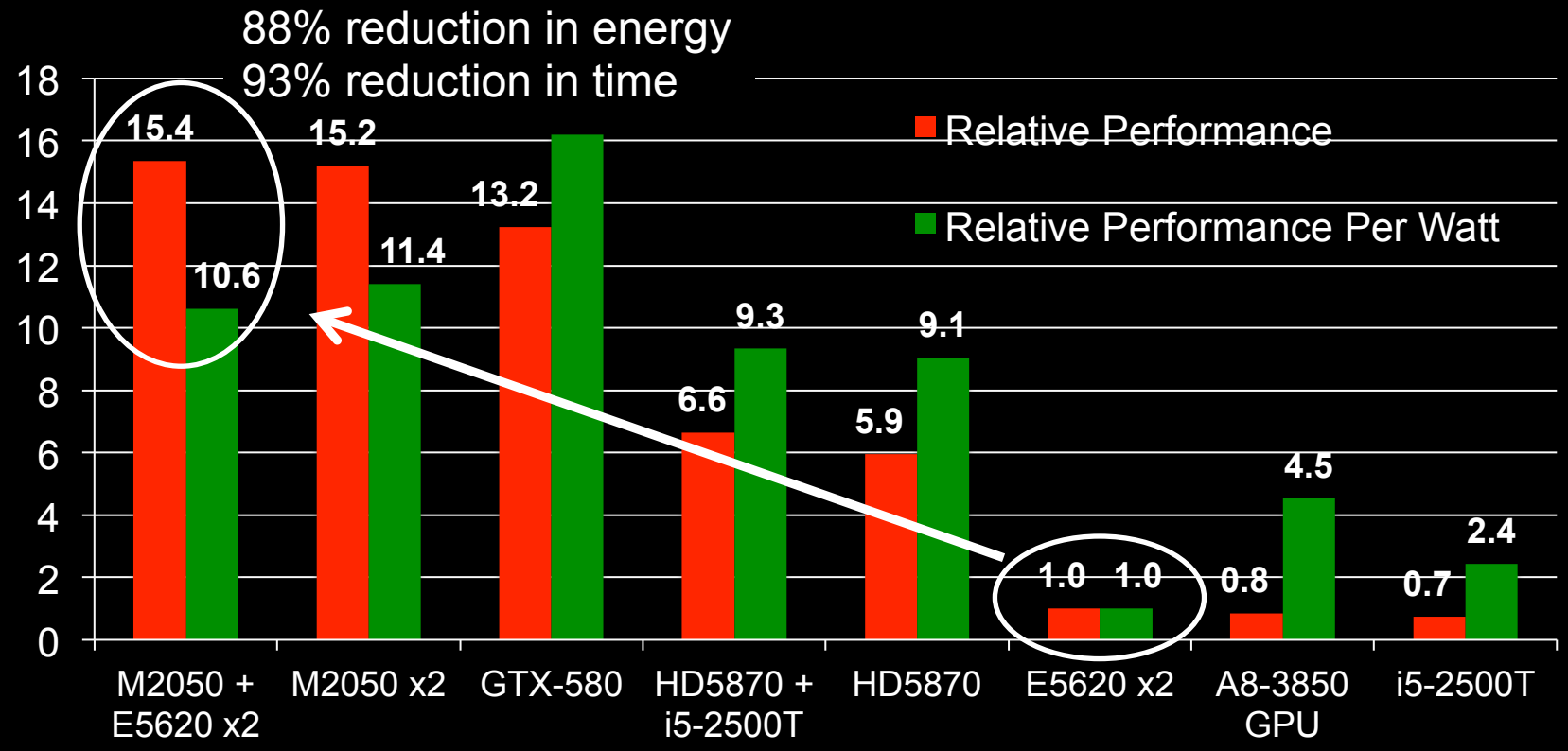
DISCOVERING OPENCL DEVICES AT RUN-TIME

```
// Get available platforms
cl_uint nPlatforms;
cl_platform_id platforms[MAX_PLATFORMS];
int ret = clGetPlatformIDs(MAX_PLATFORMS, platforms, &nPlatforms);

// Loop over all platforms
for (int p = 0; p < nPlatforms; p++) {
  // Get available devices
  cl_uint nDevices = 0;
  cl_device_id devices[MAX_DEVICES];
  clGetDeviceIDs(platforms[p], deviceType, MAX_DEVICES, devices, &nDevices);

  // Loop over all devices in this platform
  for (int d = 0; d < nDevices; d++)
    getDeviceInformation(devices[d]);
}
```

BENCHMARK RESULTS

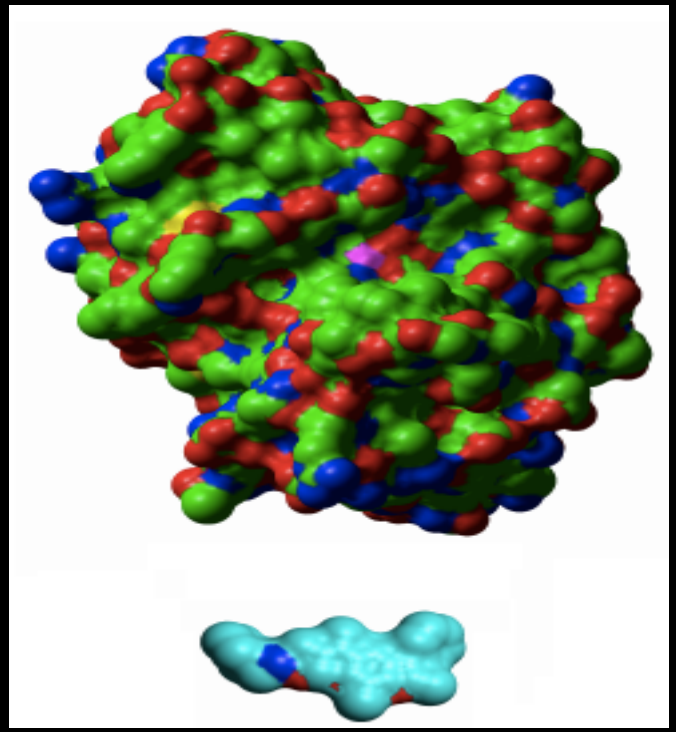


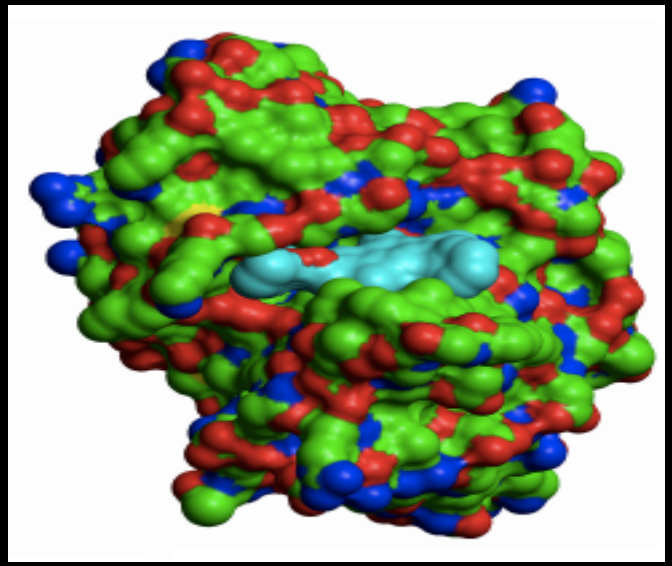
RELATIVE ENERGY AND RUN-TIME



Measurements are for a constant amount of work.
Energy measurements are “at the wall” and include any idle components.

NDM-1 AS A DOCKING TARGET





NDM-1 protein made up of 939 atoms

GPU-SYSTEM DEGIMA

- Used 222 GPUs in parallel for drug docking simulations
  - ATI Radeon HD5870 (2.72 TFLOPS) & Intel i5-2500T
- ~600 TFLOPS single precision
- Courtesy of Tsuyoshi Hamada and Felipe Cruz, Nagasaki

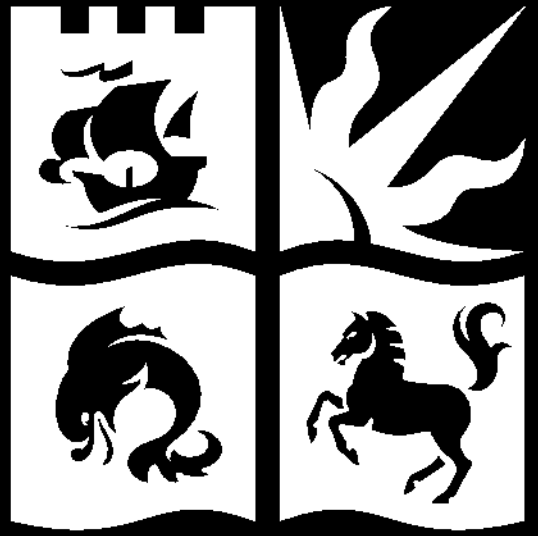
NDM-1 EXPERIMENT

- 7.65 million candidate drug molecules, 21.8 conformers each → 166.7×10^6 dockings
- 4.168×10^{12} poses calculated
- ~98 hours actual wall-time
- One of the largest collections of molecular docking simulations ever made
- Top 300 “hits” being analysed, down selecting to 10 compounds for wetlab trials soon

AMD

University of BRISTOL

PORTABLE PERFORMANCE WITH OPENCL



AMD

University of BRISTOL

PORTABLE PERFORMANCE IN OPENCL

- Portable performance is always a challenge, more so when OpenCL devices can be so varied (CPUs, GPUs, …)

- The following slides are general advice on writing code that should work well on most OpenCL devices

AMD

University of BRISTOL

PORTABLE PERFORMANCE IN OPENCL

- Don’t optimize too much for any one platform, e.g.
  - Don’t write specifically for certain warp/wavefront sizes etc
  - Be careful not to max out specific sizes of local/global memory
  - OpenCL’s vector data types have varying degrees of support – faster on some devices, slower on others
  - Some devices have caches in their memory hierarchies, some don’t, and it can make a big difference to your performance without you realizing
  - Need careful selection of Work-Group sizes and dimensions for your kernels
  - Performance differences between unified vs. disjoint host/global memories
  - Double precision performance varies considerably from device to device

- Recommend trying your code on several different platforms to see what happens (profiling is good!)
  - Try at least two different GPUs (ideally different vendors!) and at least one CPU

TIMING MICROBENCHMARKS

```
for (int i = 0; i < numDevices; i++) {
   // Wait for the kernel to finish
   ret = clFinish(oclDevices[i].queue);
   // Update timers
   cl_ulong start, end;
   ret = clGetEventProfilingInfo(oclDevices[i].kernelEvent,
            CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
   ret |= clGetEventProfilingInfo(oclDevices[i].kernelEvent,
            CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
   long timeTaken = (end - start);
   speeds[i] = timeTaken / oclDevices[i].load;
}
```

AMD

University of BRISTOL

ADVICE FOR PERFORMANCE PORTABILITY

- Assigning Work-Items to Work-Groups will need different treatment for different devices
  - E.g. CPUs tend to prefer 1 Work-Item per Work-Group, while GPUs prefer lots of Work-Items per Work-Group (usually a multiple of the number of PEs per Compute Unit, i.e. 32, 64 etc)
- In OpenCL v1.1 you can discover the preferred Work-Group size multiple for a kernel once it's been built for a specific device
  - Important to pad the total number of Work-Items to an exact multiple of this
  - Again, will be different per device
- The OpenCL run-time will have a go at choosing good EnqueueNDRangeKernel dimensions for you
  - With very variable results
    - For Bristol codes we could only do 5-10% better with manual tuning
    - For other codes it can make a ***much*** bigger difference
  - This is harder to do efficiently in a run-time, adaptive way!
- Your mileage will vary, the best strategy is to write *adaptive* code that makes decisions at run-time
- ***Assume heterogeneity!***

AMD

University of BRISTOL