www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

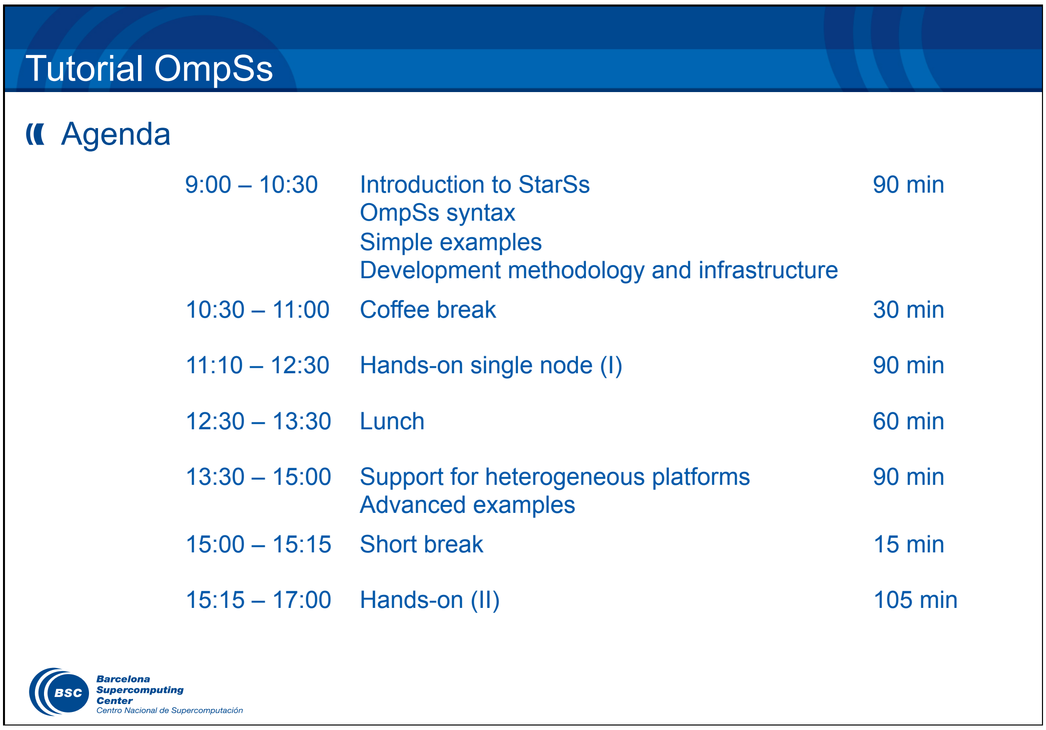
Tutorial OmpSs: Support for heterogeneous platforms

PUMPS 2013 tutorial
Hybrid and Heterogeneous Parallel Programming with MPI/ OmpSs for Exascale Syste

Rosa M Badia, Xavier Martorell

PUMPS 2013, 12 July 2013

EXCELENCIA SEVERO OCHOA



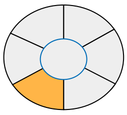
Tutorial OmpSs

Agenda

9:00 – 10:30	Introduction to StarSs OmpSs syntax Simple examples Development methodology and infrastructure	90 min
10:30 – 11:00	Coffee break	30 min
11:10 – 12:30	Hands-on single node (I)	90 min
12:30 – 13:30	Lunch	60 min
13:30 – 15:00	Support for heterogeneous platforms Advanced examples	90 min
15:00 – 15:15	Short break	15 min
15:15 – 17:00	Hands-on (II)	105 min

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs + heterogeneity

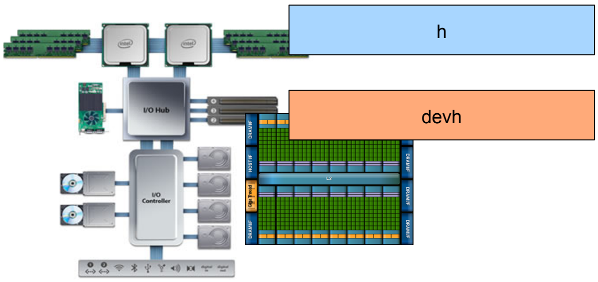


Heterogeneity

- Has come to actually mean three things …
  - ISA heterogeneity
  - Separated address spaces
  - Work definition/generation model
    - Task / Kernel
- …although it also refers to
  - Different performance between equally functional devices
- How does OmpSs support them
  - Attempt to do it separately/orthogonal

Motivation

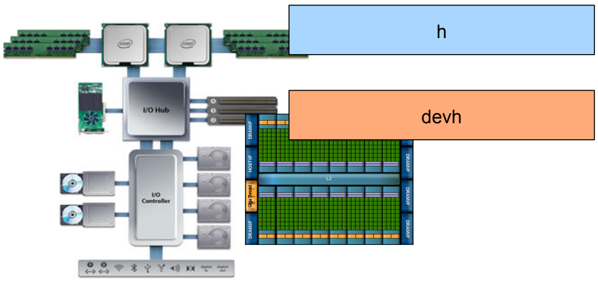
- OpenCL/CUDA coding, complex and error-prone
  - Memory allocation
  - Data copies to/from device memory
  - Manual work scheduling
  - Code and data management from the host



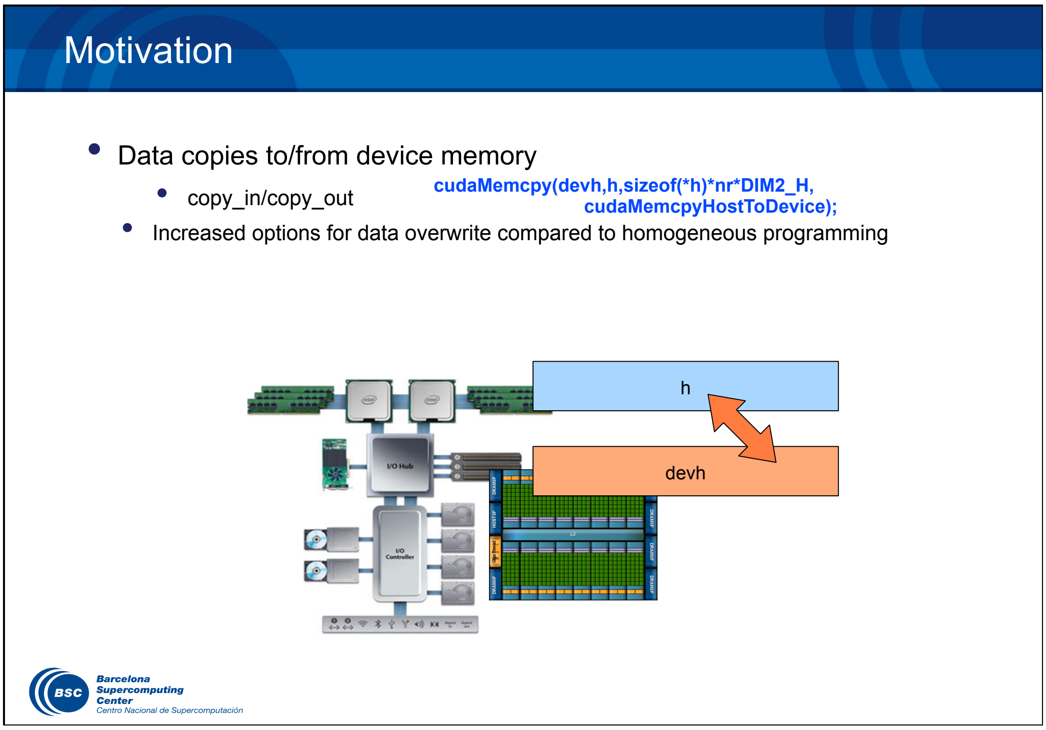
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Motivation

- Memory allocation
  - Need to have a double memory allocation
  - Host memory `h = (float*) malloc(sizeof(*h)*DIM2_H*nr);`
  - Device memory `r = cudaMalloc((void**)&devh,sizeof(*h)*nr*DIM2_H);`
- Different data sizes due to blocking may make the code confusing



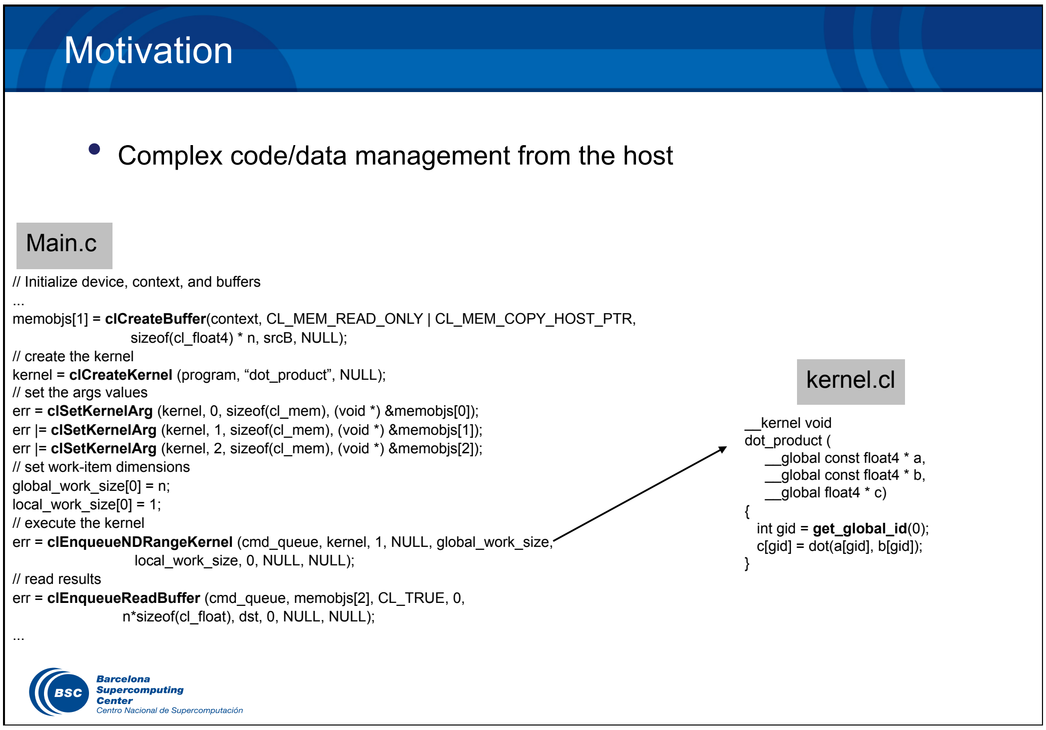
Barcelona Supercomputing Center
Centro Nacional de Supercomputación



Motivation

- Data copies to/from device memory
  - copy_in/copy_out **cudaMemcpy(devh,h,sizeof(*h)*nr*DIM2_H, cudaMemcpyHostToDevice);**
  - Increased options for data overwrite compared to homogeneous programming

Barcelona Supercomputing Center
Centro Nacional de Supercomputación



Motivation

- Complex code/data management from the host

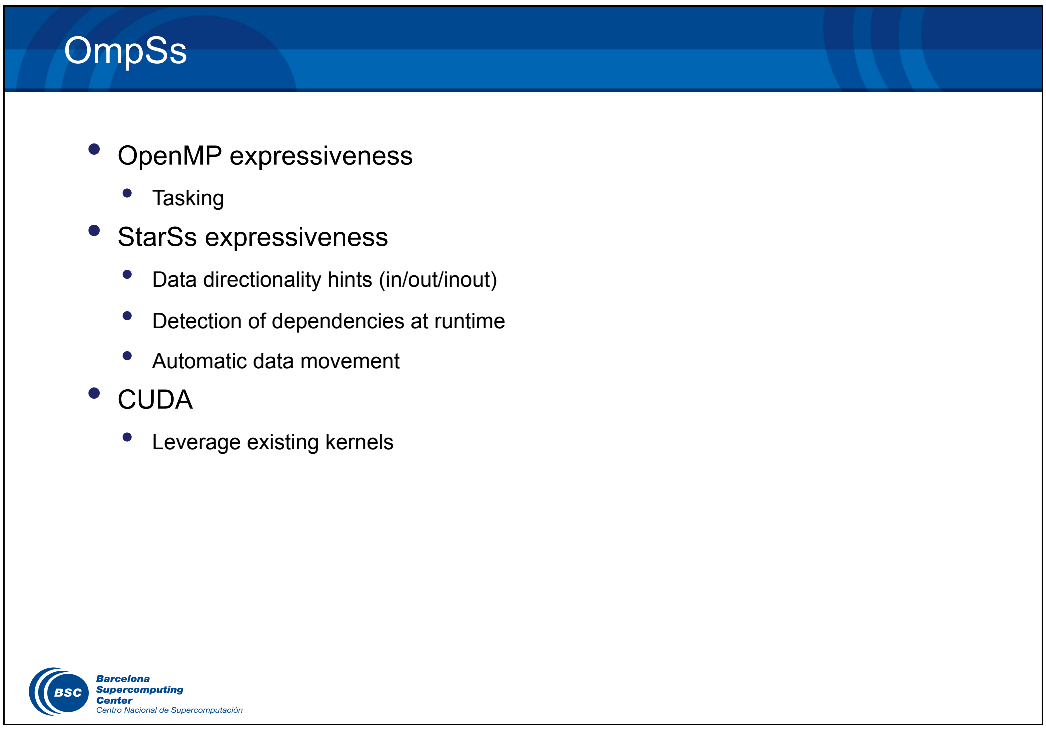
Main.c

```
// Initialize device, context, and buffers
...
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    sizeof(cl_float4) * n, srcB, NULL);
// create the kernel
kernel = clCreateKernel (program, "dot_product", NULL);
// set the args values
err = clSetKernelArg (kernel, 0, sizeof(cl_mem), (void *) &memobjs[0]);
err |= clSetKernelArg (kernel, 1, sizeof(cl_mem), (void *) &memobjs[1]);
err |= clSetKernelArg (kernel, 2, sizeof(cl_mem), (void *) &memobjs[2]);
// set work-item dimensions
global_work_size[0] = n;
local_work_size[0] = 1;
// execute the kernel
err = clEnqueueNDRangeKernel (cmd_queue, kernel, 1, NULL, global_work_size,
                    local_work_size, 0, NULL, NULL);
// read results
err = clEnqueueReadBuffer (cmd_queue, memobjs[2], CL_TRUE, 0,
                    n*sizeof(cl_float), dst, 0, NULL, NULL);
...
```

kernel.cl

```
__kernel void
dot_product (
    __global const float4 * a,
    __global const float4 * b,
    __global float4 * c)
{
  int gid = get_global_id(0);
  c[gid] = dot(a[gid], b[gid]);
}
```

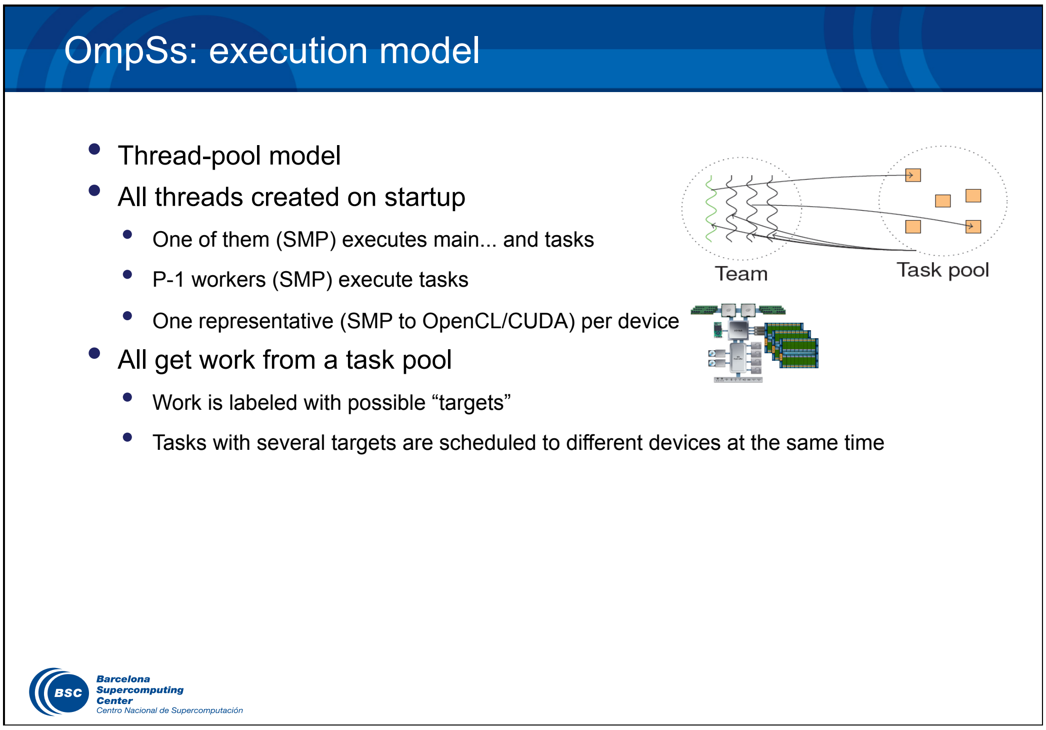
Barcelona Supercomputing Center
Centro Nacional de Supercomputación



OmpSs

- OpenMP expressiveness
  - Tasking
- StarSs expressiveness
  - Data directionality hints (in/out/inout)
  - Detection of dependencies at runtime
  - Automatic data movement
- CUDA
  - Leverage existing kernels

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación



OmpSs: execution model

- Thread-pool model
- All threads created on startup
  - One of them (SMP) executes main... and tasks
  - P-1 workers (SMP) execute tasks
  - One representative (SMP to OpenCL/CUDA) per device
- All get work from a task pool
  - Work is labeled with possible “targets”
  - Tasks with several targets are scheduled to different devices at the same time

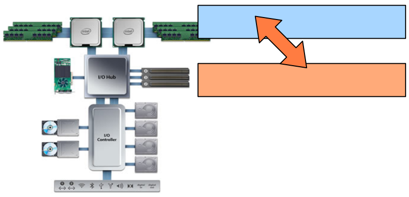
Team

Task pool

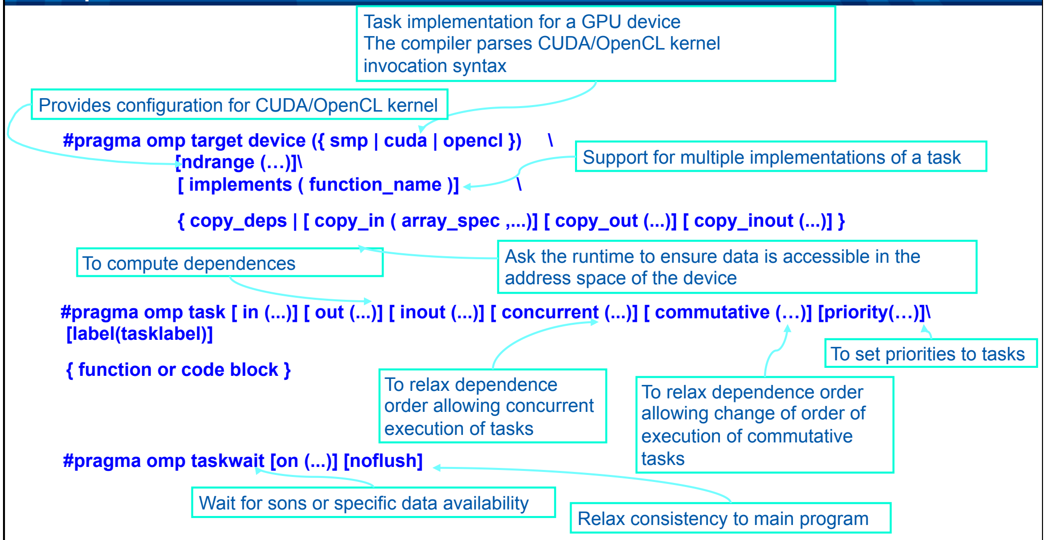
BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs: memory model

- A single global address space
- The runtime system takes care of the devices/local memories
  - SMP machines: no need for extra runtime support
  - Distributed/heterogeneous environments
    - Multiple physical address spaces
      - Possibility of multiple versions of the same data
    - Data consistency ensured by the runtime system



OmpSs: Directives



Task implementation for a GPU device
The compiler parses CUDA/OpenCL kernel invocation syntax

Provides configuration for CUDA/OpenCL kernel

**#pragma omp target device ({ smp | cuda | opencl }) \
[ndrange (…)]\
[implements (function_name)] **

Support for multiple implementations of a task

{ copy_deps | [copy_in (array_spec ,...)] [copy_out (...)] [copy_inout (...)] }

To compute dependences

Ask the runtime to ensure data is accessible in the address space of the device

**#pragma omp task [in (...)] [out (...)] [inout (...)] [concurrent (…)] [commutative (…)] [priority(…)]\
[label(tasklabel)]**

{ function or code block }

To set priorities to tasks

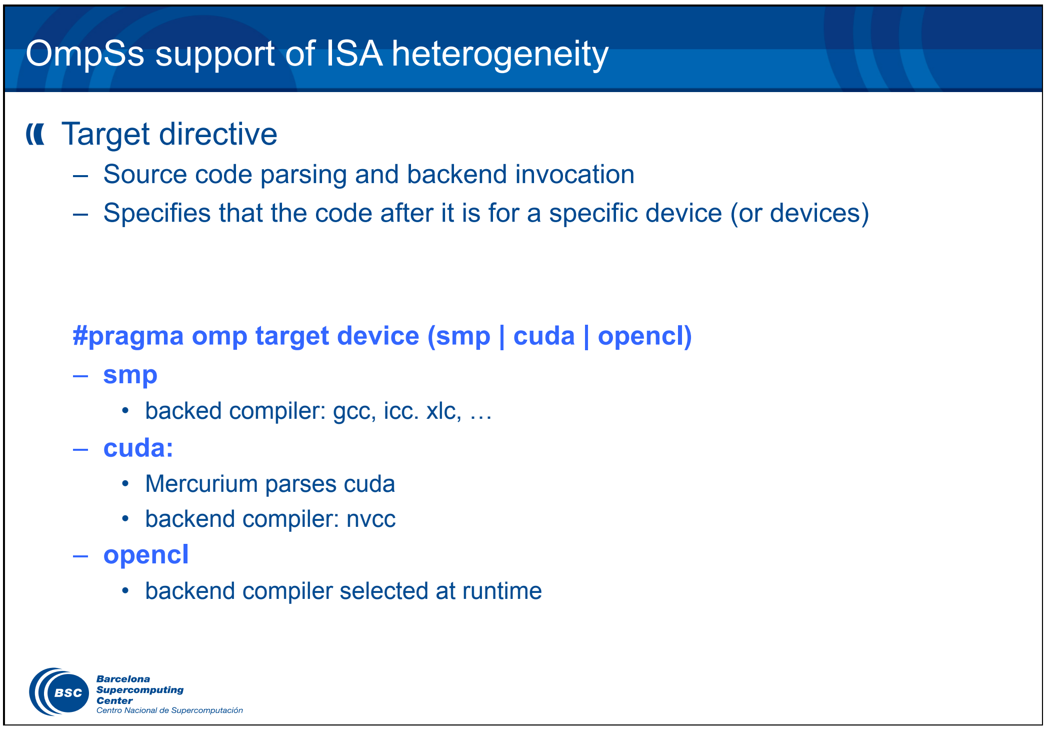
To relax dependence order allowing concurrent execution of tasks

To relax dependence order allowing change of order of execution of commutative tasks

#pragma omp taskwait [on (...)] [noflush]

Wait for sons or specific data availability

Relax consistency to main program



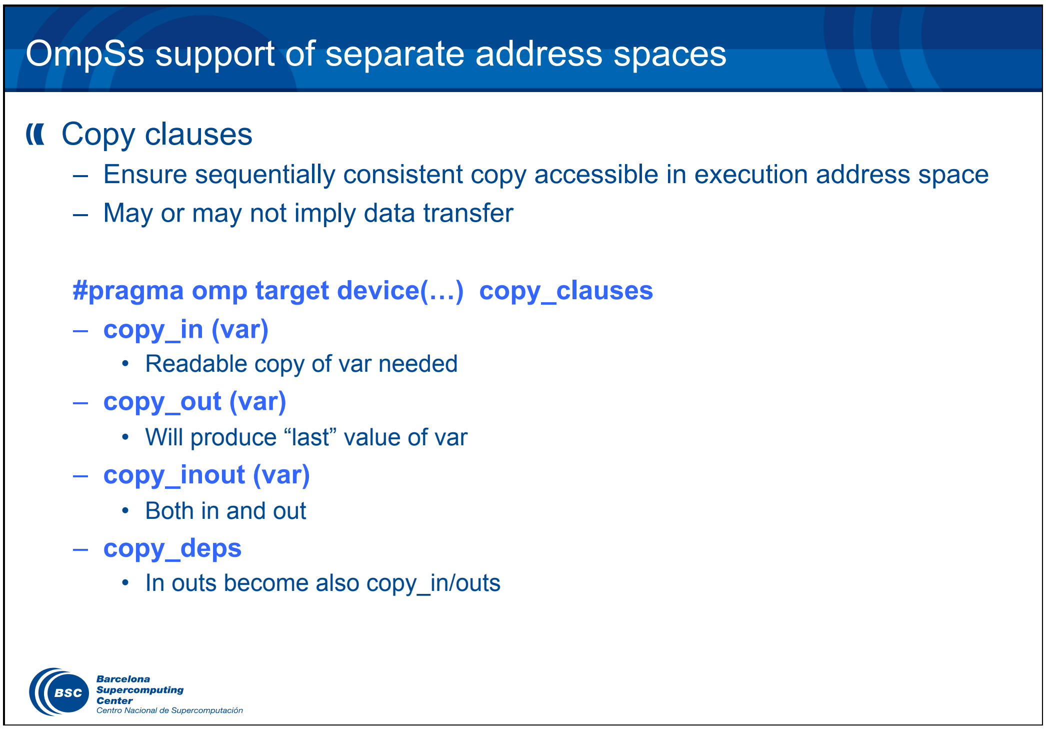
OmpSs support of ISA heterogeneity

- Target directive
  - Source code parsing and backend invocation
  - Specifies that the code after it is for a specific device (or devices)

#pragma omp target device (smp | cuda | opencl)

- **smp**
  - backed compiler: gcc, icc. xlc, …
- **cuda:**
  - Mercurium parses cuda
  - backend compiler: nvcc
- **opencl**
  - backend compiler selected at runtime

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación



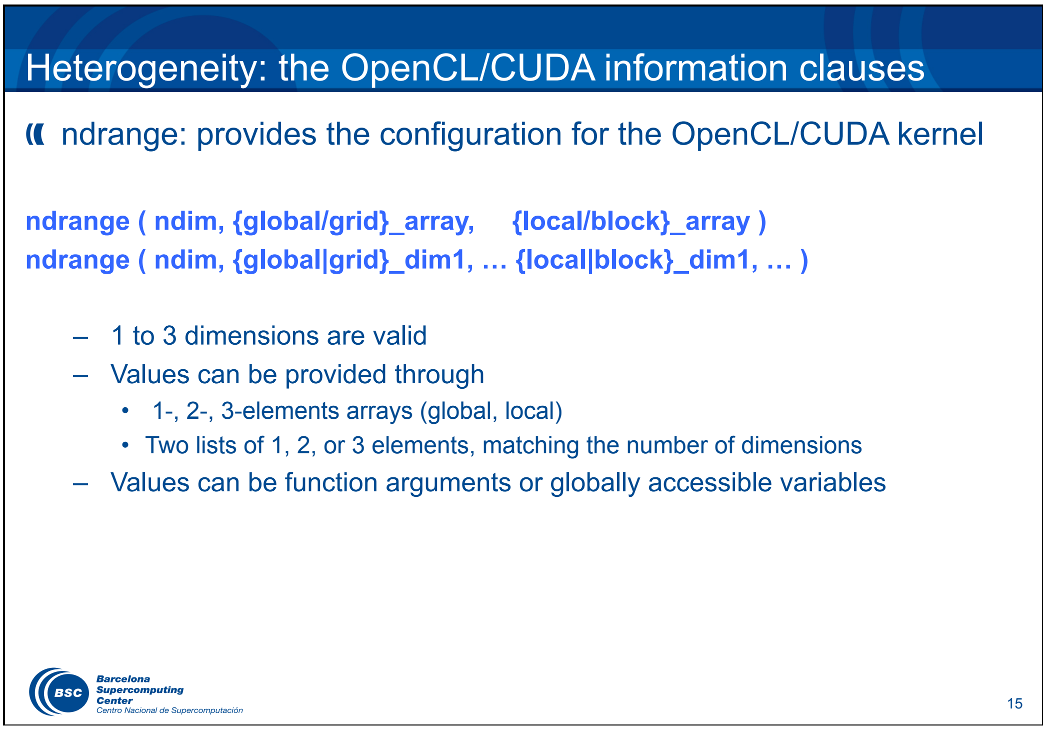
OmpSs support of separate address spaces

- Copy clauses
  - Ensure sequentially consistent copy accessible in execution address space
  - May or may not imply data transfer

#pragma omp target device(…) copy_clauses

- **copy_in (var)**
  - Readable copy of var needed
- **copy_out (var)**
  - Will produce “last” value of var
- **copy_inout (var)**
  - Both in and out
- **copy_deps**
  - In outs become also copy_in/outs

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación



Heterogeneity: the OpenCL/CUDA information clauses

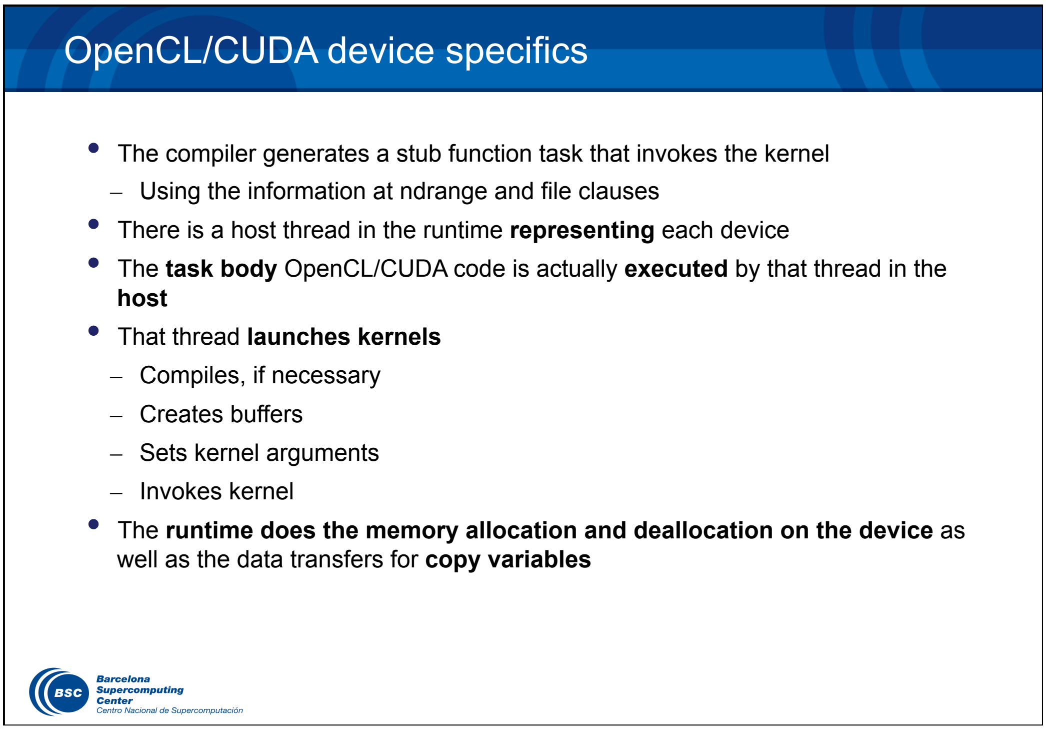
- ndrange: provides the configuration for the OpenCL/CUDA kernel

ndrange (ndim, {global/grid}_array, {local/block}_array)
ndrange (ndim, {global|grid}_dim1, … {local|block}_dim1, …)

- 1 to 3 dimensions are valid
- Values can be provided through
  - 1-, 2-, 3-elements arrays (global, local)
  - Two lists of 1, 2, or 3 elements, matching the number of dimensions
- Values can be function arguments or globally accessible variables

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

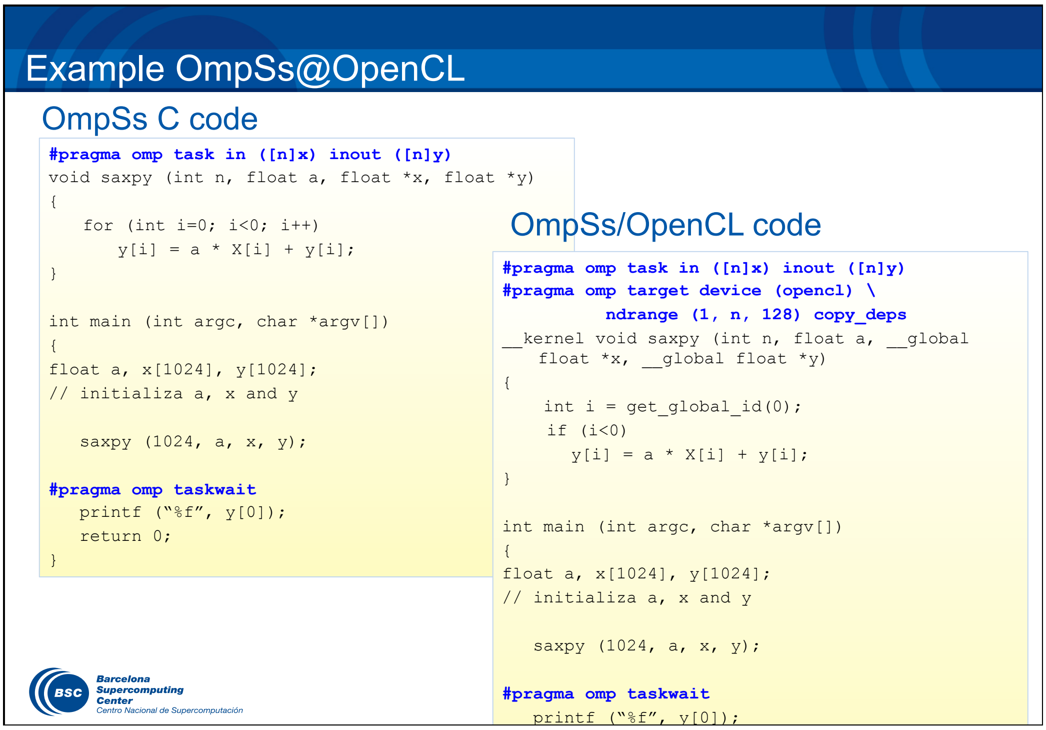
15



OpenCL/CUDA device specifics

- The compiler generates a stub function task that invokes the kernel
  - Using the information at ndrange and file clauses
- There is a host thread in the runtime **representing** each device
- The **task body** OpenCL/CUDA code is actually **executed** by that thread in the **host**
- That thread **launches kernels**
  - Compiles, if necessary
  - Creates buffers
  - Sets kernel arguments
  - Invokes kernel
- The **runtime does the memory allocation and deallocation on the device** as well as the data transfers for **copy variables**

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación



Example OmpSs@OpenCL

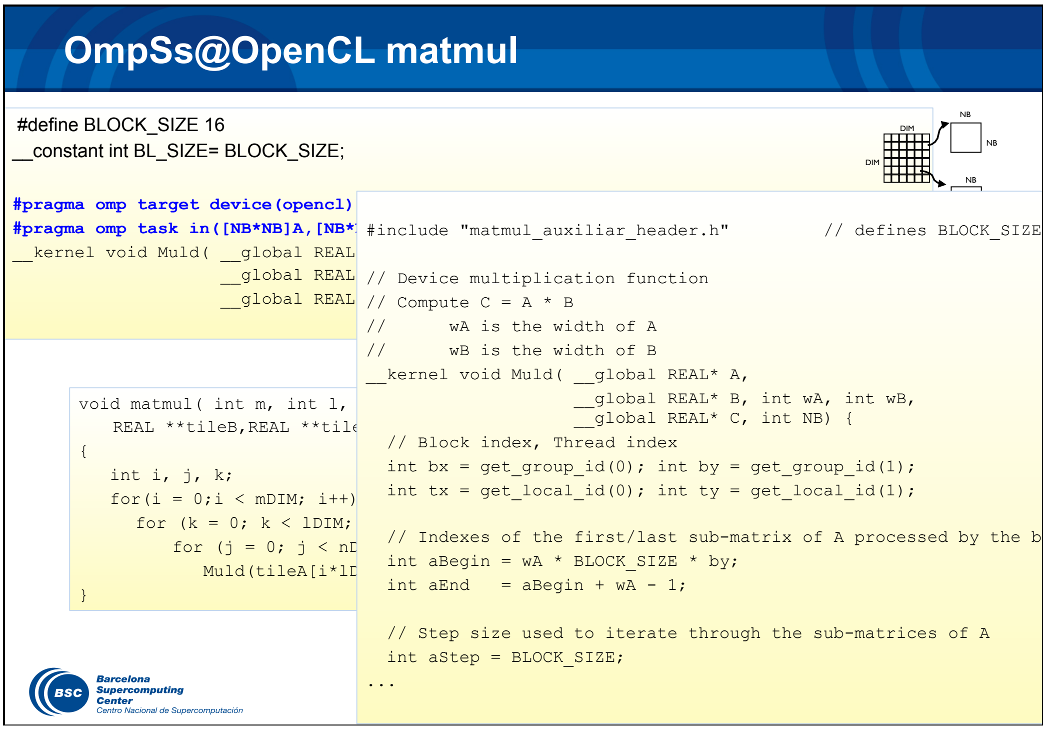
OmpSs C code

```
#pragma omp task in ([n]x) inout ([n]y)
void saxpy (int n, float a, float *x, float *y)
{
   for (int i=0; i<0; i++)
      y[i] = a * X[i] + y[i];
}

int main (int argc, char *argv[])
{
float a, x[1024], y[1024];
// initializa a, x and y

   saxpy (1024, a, x, y);

#pragma omp taskwait
   printf (“%f”, y[0]);
   return 0;
}
```

OmpSs/OpenCL code

```
#pragma omp task in ([n]x) inout ([n]y)
#pragma omp target device (opencl) \
         ndrange (1, n, 128) copy_deps
__kernel void saxpy (int n, float a, __global
   float *x, __global float *y)
{
   int i = get_global_id(0);
   if (i<0)
      y[i] = a * X[i] + y[i];
}

int main (int argc, char *argv[])
{
float a, x[1024], y[1024];
// initializa a, x and y

   saxpy (1024, a, x, y);

#pragma omp taskwait
   printf (“%f”, y[0]);
```

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación



OmpSs@OpenCL matmul

#define BLOCK_SIZE 16
__constant int BL_SIZE= BLOCK_SIZE;

DIM
DIM
NB
NB
NB

```
#pragma omp target device(opencl)
#pragma omp task in([NB*NB]A,[NB*
__kernel void Muld( __global REAL
                    __global REAL
                    __global REAL
```

```
void matmul( int m, int l,
   REAL **tileB,REAL **tile
{
   int i, j, k;
   for(i = 0;i < mDIM; i++)
      for (k = 0; k < lDIM;
         for (j = 0; j < nD
            Muld(tileA[i*lD
}
```

```
#include "matmul_auxiliar_header.h"         // defines BLOCK_SIZE

// Device multiplication function
// Compute C = A * B
//       wA is the width of A
//       wB is the width of B
__kernel void Muld( __global REAL* A,
                    __global REAL* B, int wA, int wB,
                    __global REAL* C, int NB) {
  // Block index, Thread index
  int bx = get_group_id(0); int by = get_group_id(1);
  int tx = get_local_id(0); int ty = get_local_id(1);

  // Indexes of the first/last sub-matrix of A processed by the b
  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd   = aBegin + wA - 1;

  // Step size used to iterate through the sub-matrices of A
  int aStep = BLOCK_SIZE;
...
```

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

OmpSs@CUDA matmul

```
#pragma omp target d
#pragma omp task ino
__global__ void Muld
```

```
void matmul( int m,
    REAL **tileB, REA
{
   int i, j, k;
   for(i = 0;i < mDI
      for (k = 0; k <
         for (j = 0;
            Muld(til
}
```

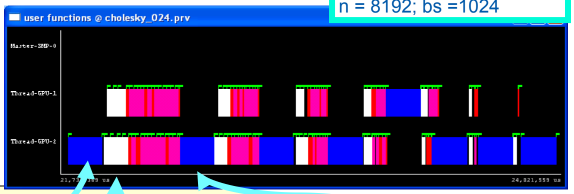
```
#include "matmul_auxiliar_header.h"

// Thread block size
#define BLOCK_SIZE 16


// Device multiplication function called by Mul()
// Compute C = A * B
//      wA is the width of A
//      wB is the width of B
__global__ void Muld(REAL* A, REAL* B, int wA, int wB, REAL* C, int NB)
    {
  // Block index
  int bx = blockIdx.x; int by = blockIdx.y;
  // Thread index
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Index of the first sub-matrix of A processed by the block
  int aBegin = wA * BLOCK_SIZE * by;
  // Index of the last sub-matrix of A processed by the block
  int aEnd   = aBegin + wA - 1;

  // Step size used to iterate through the sub-matrices of A
  int aStep = BLOCK_SIZE;
…
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Heterogeneity support: the target clause

- Source code independent of # devices

n = 8192; bs =1024



Spotrf:
Slow task @ GPU
In critical path (scheduling problem)

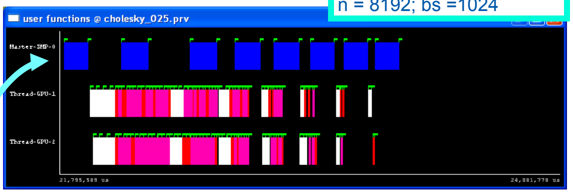
```
void blocked_cholesky( int NT, float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT;
         for (j=k+1; j<
            sgemm( A[k*
         ssyrk (A[k*NT+
   }
}
```

```
#pragma target device (cuda)
#pragma omp task in([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void sgemm(float  *A, float *B, float *C, unsigned long NB)
{
  int hA, wA, wB;
  hA = NB; wA = NB; wB = NB;
  cudaStream_t stream = nanos_get_kernel_execution_stream();
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  int gx = wB / dimBlock.x;
  int gy = hA / dimBlock.y;
  dim3 dimGrid(gx, gy);
  Muld <<<dimGrid, dimBlock, 0, stream>>> ( A, B, wA, wB, C );
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Heterogeneity support: the target clause

- Spotrf more efficient at CPU
- Overlap between CPU and GPU



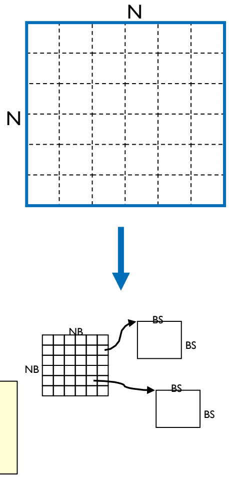
n = 8192; bs =1024

```
#pragma target device (smp)
#pragma omp task inout([NB][NB]A)
void spotrf_tile(float *A, int NB)
{
    long INFO;
    char L = 'L';

    spotrf_( &L, &NB, A, &NB, &INFO );
}
```

Standard row-wise matrix association ?

```
void flat_cholesky( int N, float *A ) {
   float **Ah;
   int nt = n/BS;
   Ah = allocate_block_matrix();
   convert_to_blocks(n, nt, A, Ah);
   blocked_cholesky (nt, Ah);
   convert_to_linear(n, bs, Ah, A);
   #pragma omp taskwait
   free_block_matrix(Ah)
}
```

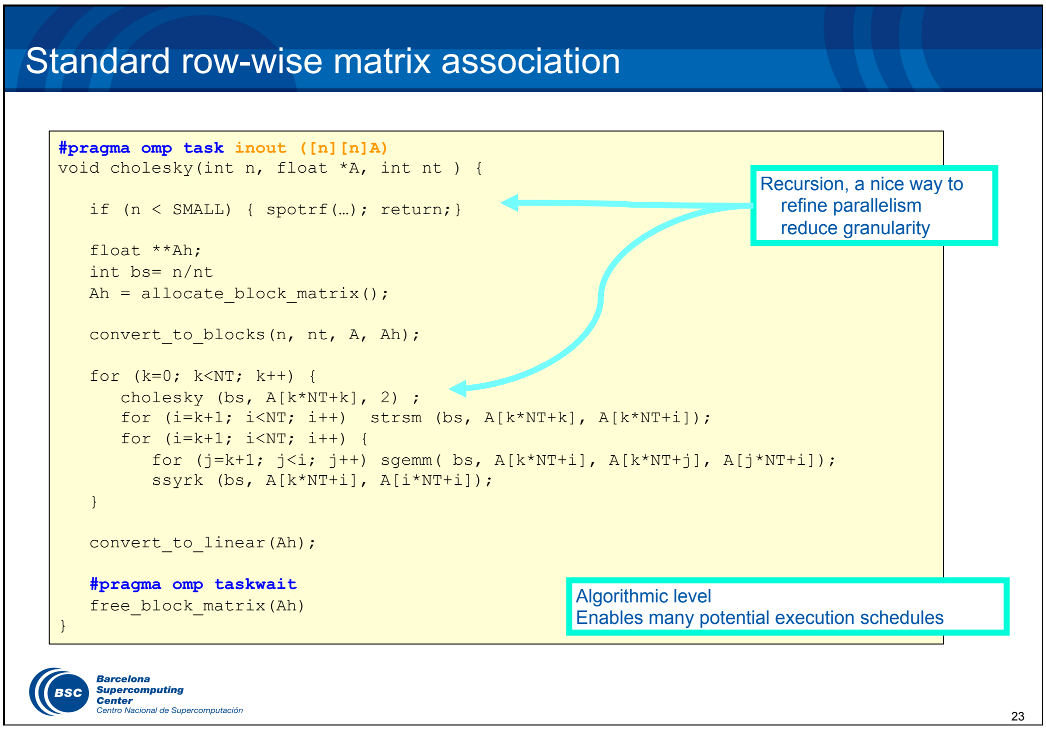
Local memory management
Temporary work arrays



```
void convert_to_block( int n, int nt, float * A , float **Ah) {
  for (i=0; i<nt; i++)
    for (j=0; j<nt; j++) gather_block (n, A, i, j, Ah[i*nt+j]]);
}
```

```
void convert_to_linear(int n, int bs, float **Ah, float * A ) {
 for (i=0; i<nt; i++)
    for (j=0; j<nt; j++) scatter_block (n, bs, A, Ah[i*nt+j], I, j);
}
```

```
#pragma omp task in ([n][n]A) out ([bs][bs]bA)
void gather_block (int n, float *A, int i, int j, float *bA);
#pragma omp task in ([bs][bs]bA) inout ([n][n]) concurrent(A)
void scatter_block (int n, bs, float *bA, float *A, i, j);
```



Standard row-wise matrix association

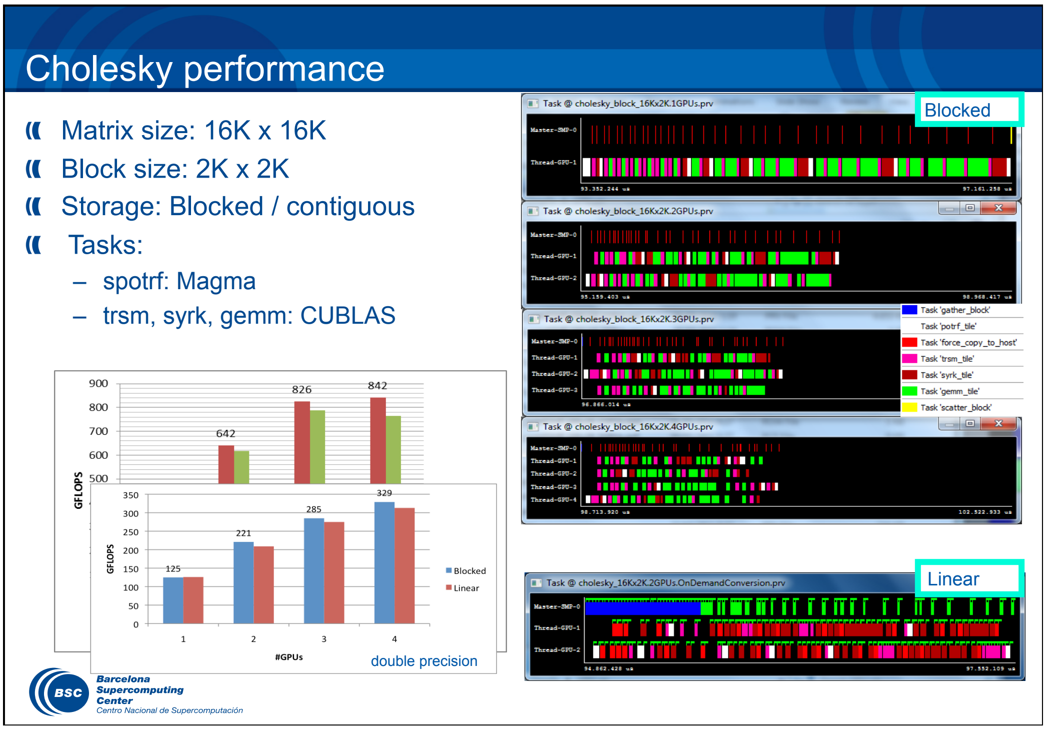
```
#pragma omp task inout ([n][n]A)
void cholesky(int n, float *A, int nt ) {

   if (n < SMALL) { spotrf(…); return;}

   float **Ah;
   int bs= n/nt
   Ah = allocate_block_matrix();

   convert_to_blocks(n, nt, A, Ah);

   for (k=0; k<NT; k++) {
      cholesky (bs, A[k*NT+k], 2) ;
      for (i=k+1; i<NT; i++)  strsm (bs, A[k*NT+k], A[k*NT+i]);
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++) sgemm( bs, A[k*NT+i], A[k*NT+j], A[j*NT+i]);
         ssyrk (bs, A[k*NT+i], A[i*NT+i]);
   }

   convert_to_linear(Ah);

   #pragma omp taskwait
   free_block_matrix(Ah)
}
```

Recursion, a nice way to
refine parallelism
reduce granularity

Algorithmic level
Enables many potential execution schedules

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

23



Cholesky performance

- Matrix size: 16K x 16K
- Block size: 2K x 2K
- Storage: Blocked / contiguous
- Tasks:
  - spotrf: Magma
  - trsm, syrk, gemm: CUBLAS

double precision

Blocked

Linear

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

OmpSs + CUDA: multiple GPUs and nodes

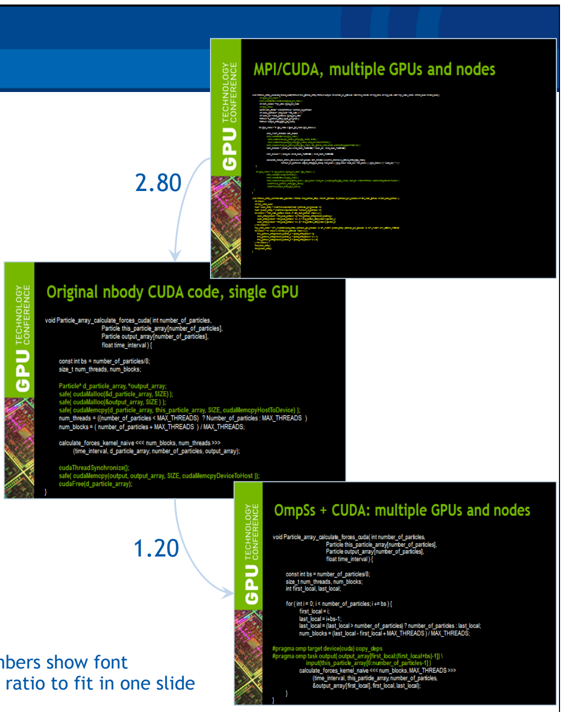
```
void Particle_array_calculate_forces_cuda( int no_particles,
        Particle this_particle_array[no_particles],
        Particle output_array[no_particles],
        float time_interval ) {

   const int bs = no_particles/8;
   size_t no_threads, no_blocks;
   int first_local, last_local;

   for ( int i = 0; i < no_particles; i += bs ) {
      first_local = i;
      last_local = (i+bs-1 > no_particles) ? no_particles : i+bs-1;
      no_blocks = (last_local - first_local + MAX_THREADS ) / MAX_THREADS;

#pragma omp target device(cuda) copy_deps
#pragma omp task in( this_particle_array[0:no_particles-1] ) \
                 out( output_array[first_local:(first_local+bs)-1])

      calculate_forces_kernel_naive <<< no_blocks, MAX_THREADS >>>
                    (time_interval, this_particle_array, no_particles,
                     &output_array[first_local], first_local, last_local);
   }
}
```

Block algorithms
Multiple tasks

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Programming productivity

- Development time
- Maintainability
- Structural malleability
- Number of concepts
- LoCs



MPI/CUDA, multiple GPUs and nodes

Original nbody CUDA code, single GPU

```
void Particle_array_calculate_forces_cuda( int number_of_particles,
                        Particle this_particle_array[number_of_particles],
                        Particle output_array[number_of_particles],
                        float time_interval ) {

      const int bs = number_of_particles/8;
      size_t num_threads, num_blocks;

      Particle* d_particle_array, *output_array;
      safe( cudaMalloc(&d_particle_array, SIZE) );
      safe( cudaMalloc(&output_array, SIZE ) );
      safe( cudaMemcpy(d_particle_array, this_particle_array, SIZE, cudaMemcpyHostToDevice) );
      num_threads = ((number_of_particles < MAX_THREADS)  ? Number_of_particles : MAX_THREADS  )
      num_blocks = ( number_of_particles + MAX_THREADS  ) / MAX_THREADS;

      calculate_forces_kernel_naive <<< num_blocks, num_threads >>>
            (time_interval, d_particle_array, number_of_particles, output_array);

      cudaThreadSynchronize();
      safe( cudaMemcpy(output, output_array, SIZE, cudaMemcpyDeviceToHost ));
      cudaFree(d_particle_array);
}
```

OmpSs + CUDA: multiple GPUs and nodes

```
void Particle_array_calculate_forces_cuda( int number_of_particles,
                        Particle this_particle_array[number_of_particles],
                        Particle output_array[number_of_particles],
                        float time_interval ) {

      const int bs = number_of_particles/8;
      size_t num_threads, num_blocks;
      int first_local, last_local;

      for ( int i = 0; i < number_of_particles; i += bs ) {
            first_local = i;
            last_local = i+bs-1;
            last_local = (last_local > number_of_particles) ? number_of_particles : last_local;
            num_blocks = (last_local - first_local + MAX_THREADS ) / MAX_THREADS;

#pragma omp target device(cuda) copy_deps
#pragma omp task output( output_array[first_local:(first_local+bs)-1]) \
                 input(this_particle_array[0:number_of_particles-1] )
            calculate_forces_kernel_naive <<< num_blocks, MAX_THREADS >>>
                  (time_interval, this_particle_array, number_of_particles,
                  &output_array[first_local], first_local, last_local);
      }
}
```

Numbers show font
size ratio to fit in one slide

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

One source, different platforms

Nbody code

1GPU/node

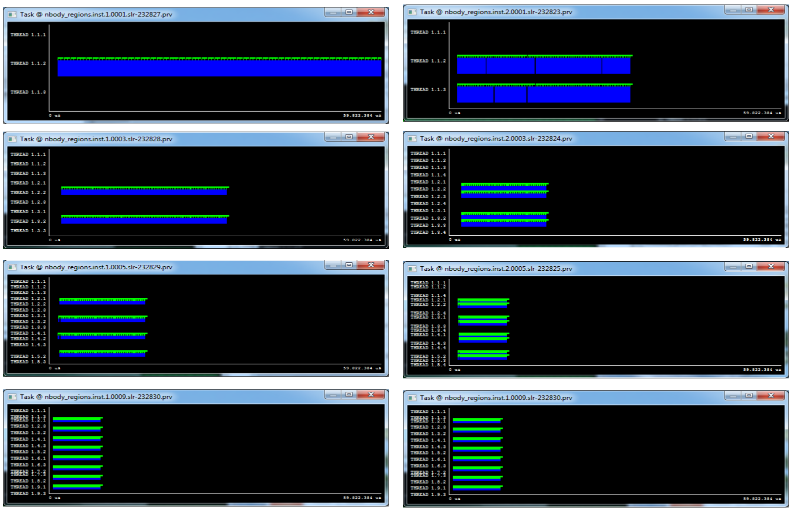
2 GPUs/node

1 node

2 nodes

4 nodes

8 nodes


Task @ nbody_regions.inst.1.0001.slr-232827.prv
Task @ nbody_regions.inst.2.0001.slr-232823.prv
Task @ nbody_regions.inst.1.0003.slr-232828.prv
Task @ nbody_regions.inst.2.0003.slr-232824.prv
Task @ nbody_regions.inst.1.0005.slr-232829.prv
Task @ nbody_regions.inst.2.0005.slr-232825.prv
Task @ nbody_regions.inst.1.0009.slr-232830.prv
Task @ nbody_regions.inst.1.0009.slr-232830.prv
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
0 us
59.822.384 us

J. Bueno et al, "Productive Programming of GPU Clusters with OmpSs", IPDPS2012

Barcelona Supercomputing Center
Centro Nacional de Supercomputación