PATC Training

OmpSs and GPUs support

Xavier Martorell
Programming Models / Computer Science Dept.
BSC

May 23rd-24th, 2012

Outline

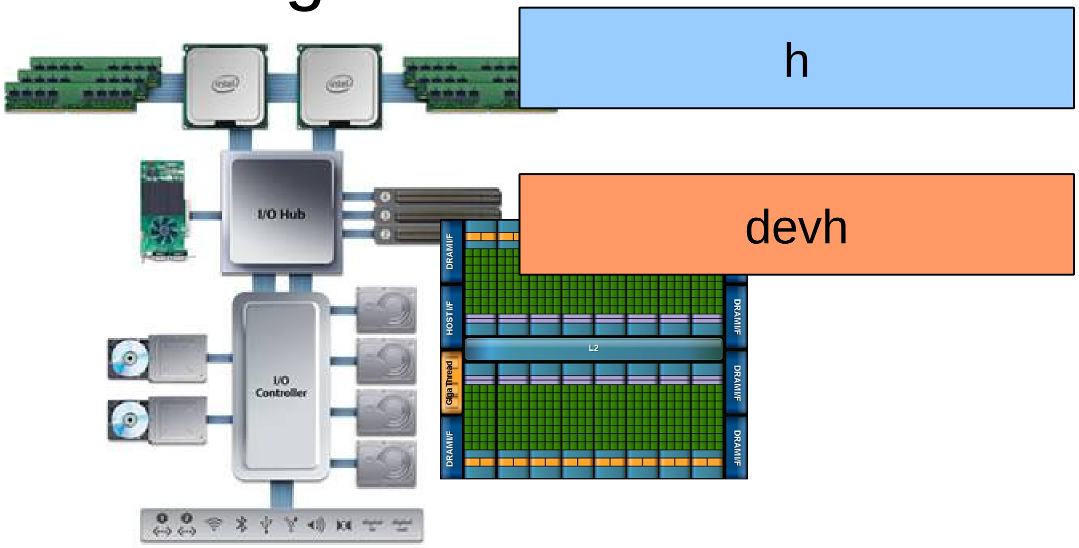
- Motivation
- OmpSs
- Examples
  - BlackScholes
  - Perlin noise
  - Julia Set
- Hands-on

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

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

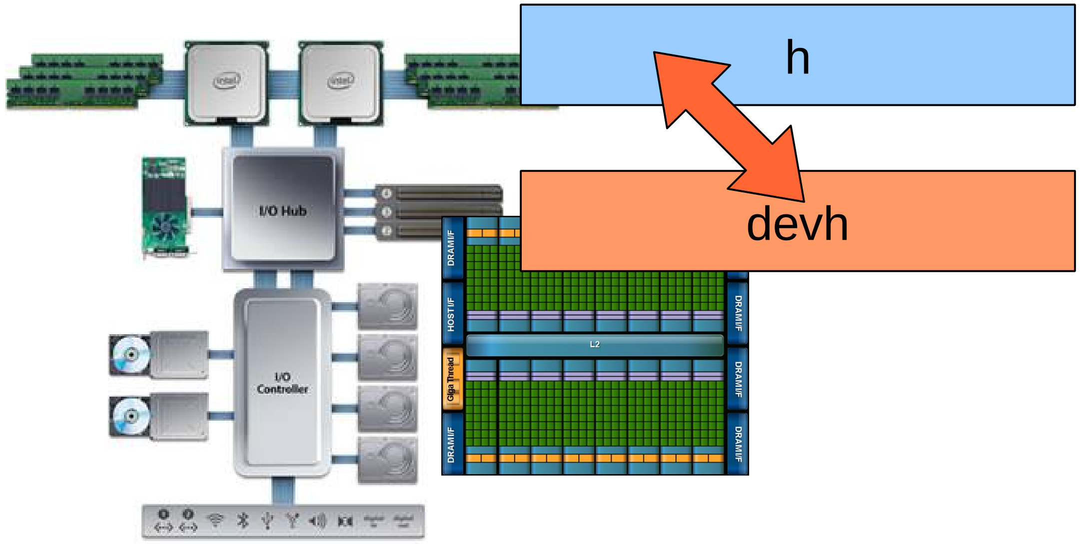


Motivation

- 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



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);
...
```

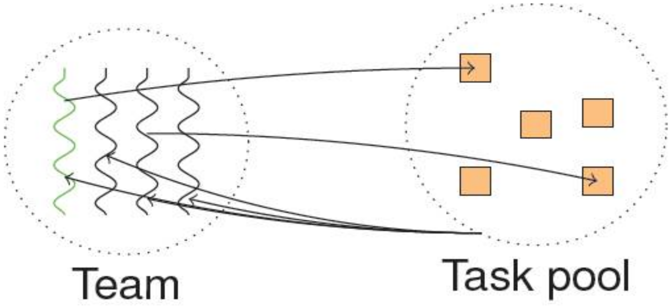
kernel.cl

```
__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]);
}
```

Proposal: OmpSs

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

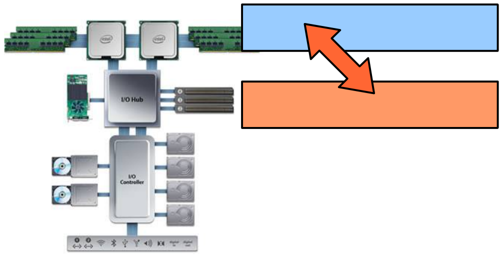
OmpSs: execution model



- Thread-pool model
  - OpenMP parallel “ignored”
- All threads created on startup
  - One of them (SMP) executes main... and tasks
  - P-1 workers (SMP) execute tasks
  - One representative (SMP to CUDA) per GPU
- All get work from a task pool
  - Work is labeled with possible “targets”

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 exist
      - Versions of the same data can reside on them
    - Data consistency ensured by the runtime system



OmpSs: the **target** directive

- Specify device specific information

  #pragma omp target [clauses]

  – Clauses

    - Device: which type of device to use (smp, gpu...)
    - copy_in, copy_out, copy_inout: data to be moved in and out of the device memory
    - copy_deps: copy the data specified on the dependency clauses (input/output/inout) of the task
    - implements (on development): specifies alternate implementations

OmpSs example

```
 float a[N];
 float b[N];
 float c[N];

 for (J=0; J<N; J+=BK) {
#pragma omp target device(cuda) copy_deps
#pragma omp task input (a[J;BK], b[J;BK]) output (c[J;BK])
     {
         for (j=J; j < J+BK; j++) c[j] = a[j] + b[j];
     }
#pragma omp target device(cuda) copy_deps
#pragma omp task input (c[J;BK])...
     {
         for (j=J; ...)      ...   = c[j];
     }
 }
```

OmpSs example

- Invoking a CUDA kernel from an OmpSs task

```
for (j = 0; j < img_height; j+=BS) {
  #pragma omp target device (cuda) copy_deps
  #pragma omp task output (out[ j ; rowstride ])
  {
    dim3 dimBlock;
    dim3 dimGrid;
    dimBlock.x = BS;
    dimBlock.y = BS;
    dimBlock.z = 1;
    dimGrid.x = img_width/dimBlock.x;
    dimGrid.y = img_height/dimBlock.y;
    dimGrid.z = 1;

    cuda_perlin <<<dimGrid, dimBlock>>> (&output[j*rowstride], time, j, rowstride);
  } }
```

Important restriction:
- No data accesses in this host code

We recommend:
- Set block/grid and invoke kernel

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs: the **target** directive

- Example of alternative implementations

```
#pragma omp target device (smp) copy_deps
#pragma omp task input ([size] c) output ([size] b)
void scale_task (double *b, double *c, double scalar, int size)
{
    int j;
    for (j=0; j < BSIZE; j++)
        b[j] = scalar*c[j];
}
```

```
#pragma omp target device (cuda) copy_deps implements (scale_task)
#pragma omp task input ([size] c) output ([size] b)
void scale_task_cuda(double *b, double *c, double scalar, int size)
{
    dim3 dimBlock;
   dimBlock.x = threadsPerBlock;

   dim3 dimGrid;
   dimGrid.x = size/threadsPerBlock+1;

   scale_kernel<<<dimGrid,dimBlock>>>(size, 1, b, c, scalar);
}
```

OmpSs: Summary of directives

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

Support for multiple implementations of a task

**#pragma omp target device ({ smp | cuda }) **
**[implements (function_name)] **
{ copy_deps | [copy_in (array_spec ,...)] [copy_out (...)] [copy_inout (...)] }

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

#pragma omp task [input (...)] [output (...)] [inout (...)] [concurrent (...)]
{ function or code block }

To compute dependences

To allow concurrent execution of commutative tasks

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

Wait for sons or specific data availability

Relax consistency to main program

OmpSs: reducing data transfers

- Sometimes there is a need to synchronize...
  - but no need for data output at that point

```
void compute_perlin_noise_device(pixel * output, float time, unsigned int
rowstride, int img_height, int img_width)
{
    unsigned int i, j;
    float vy, vt;
    const int BSy = 1;
    const int BSx = 512;
    const int BS = img_height/16;

    for (j = 0; j < img_height; j+=BS) {
#pragma omp target device(cuda) copy_out(output[j*rowstride;BS*rowstride])
#pragma omp task
        {
            dim3 dimBlock, dimGrid;
            dimBlock.x = (img_width < BSx) ? img_width : BSx;
            dimBlock.y = (BS < BSy) ? BS : BSy;
            dimBlock.z = 1;
            dimGrid.x = img_width/dimBlock.x;
            dimGrid.y = BS/dimBlock.y;
            dimGrid.z = 1;
                cuda_perlin <<<dimGrid, dimBlock>>> (&output[j*rowstride],
                                time, j, rowstride);
        }
    }
#pragma omp taskwait noflush     // a later taskwait will force
}                               // the data to be consistent
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

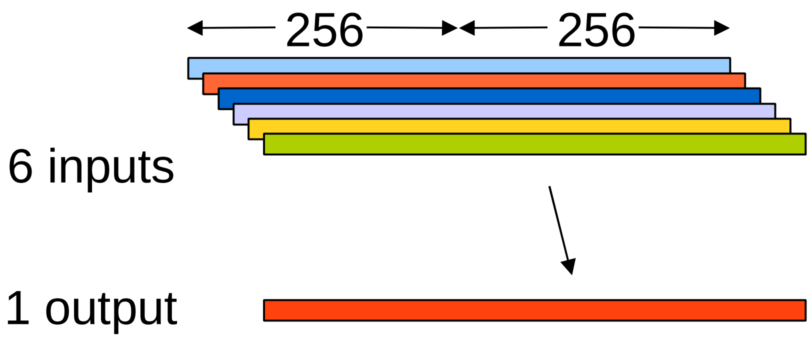
Outline

- Motivation
- OmpSs
- Examples
  - BlackScholes
  - Perlin noise
  - Julia Set
- Hands-on

OmpSs BlackScholes

- Use of input/output/inout

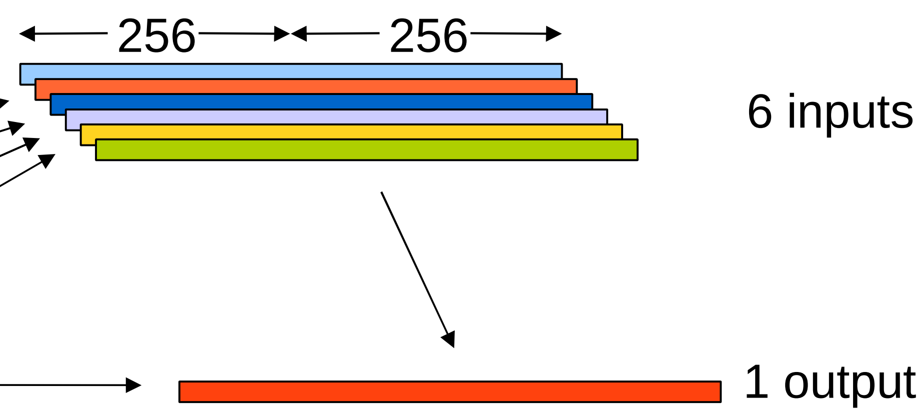
```
#pragma omp target device(cuda) copy_deps
#pragma omp task input ( \
                       [global_work_group_size]   cpflag_fptr, \
                       [global_work_group_size]   S0_fptr, \
                       [global_work_group_size]   K_fptr, \
                       [global_work_group_size]   r_fptr, \
                       [global_work_group_size]   sigma_fptr, \
                       [global_work_group_size]   T_fptr) \
        output ([global_work_group_size]   answer_fptr)
void bsop_ref_float (
                     unsigned int cpflag_fptr[],
                     float S0_fptr[],
                     float K_fptr[],
                     float r_fptr[],
                     float sigma_fptr[],
                     float T_fptr[],
                     float answer_fptr[])
                 {
                     // kernel code
                 }
```



BlackScholes

- Use of copy_in/copy_out/copy_inout

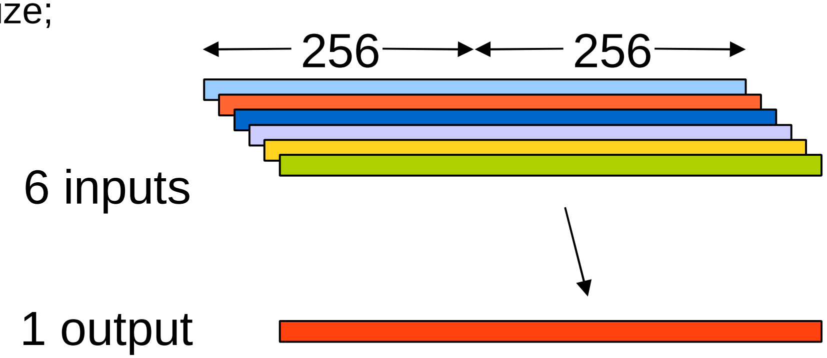
```
   chunksize = 256;
   for (i=0; i<array_size; i+= chunk_size ) {
      ...
      elements = min(i+chunk_size, array_size ) - i;
#pragma omp target device(cuda) copy_in( \
                    cpf    [i;elements], \
                    S0     [i;elements], \
                    K      [i;elements], \
                    r      [i;elements], \
                    sigma [i;elements], \
                    T      [i;elements]) \
               copy_out (answer[i;elements])
#pragma omp task firstprivate(local_work_group_size, i)
      {
         dim3 dimBlock(local_work_group_size, 1 , 1);
         dim3 dimGrid(elements / local_work_group_size, 1 , 1 );
         cuda_bsop <<<dimGrid, dimBlock>>>
                    (&cpf[i], &S0[i], &K[i], &r[i], &sigma[i], &T[i], &answer[i]);
      }
   }
#pragma omp taskwait
```



OmpSs BlackScholes

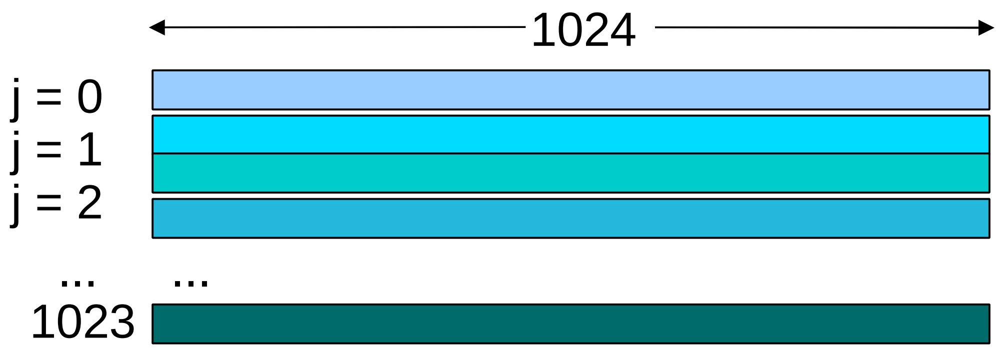
- Use of copy_in/copy_out when no need for dependence checking

```
for (i=0; i<array_size; i+=local_work_group_size*vector_width) {
  int limit = ((i+local_work_group_size)>array_size) ?
                        array_size - i : local_work_group_size;
  uint * cpflag_f = &cpflag_fptr[i];
  float * S0_f   = &S0_fptr[i];
  float * K_f    = &K_fptr[i];
  float * r_f    = &r_fptr[i];
  float * sigma_f = &sigma_fptr[i];
  float * T_f     = &T_fptr[i];
  float * answer_f = &answer_fptr[i];
```




```
#pragma omp target device(cuda) copy_in( \
                              [global_work_group_size]  cpflag_f, \
                              [global_work_group_size]  S0_f, \
                              [global_work_group_size]  K_f, \
                              [global_work_group_size]  r_f, \
                              [global_work_group_size]  sigma_f, \
                              [global_work_group_size]  T_f) \
                    copy_out ([global_work_group_size]  answer_f)
#pragma omp task shared(cpflag_f,S0_f,K_f,r_f,sigma_f,T_f,answer_f)
 {
   // kernel code
 }
```

OmpSs Perlin Noise




```
  for (j = 0; j < img_height; j+=BS) {

#pragma omp target device(cuda)  copy_deps
#pragma omp task output (output[j*rowstride:(j*BS)*rowstride-1])
    {
        dim3 ...
        ...
        cuda_perlin <<<dimGrid, dimBlock>>> (&output[j*rowstride], time, j, rowstride);
    }
  }
#pragma omp taskwait
```

OmpSs Perlin Noise

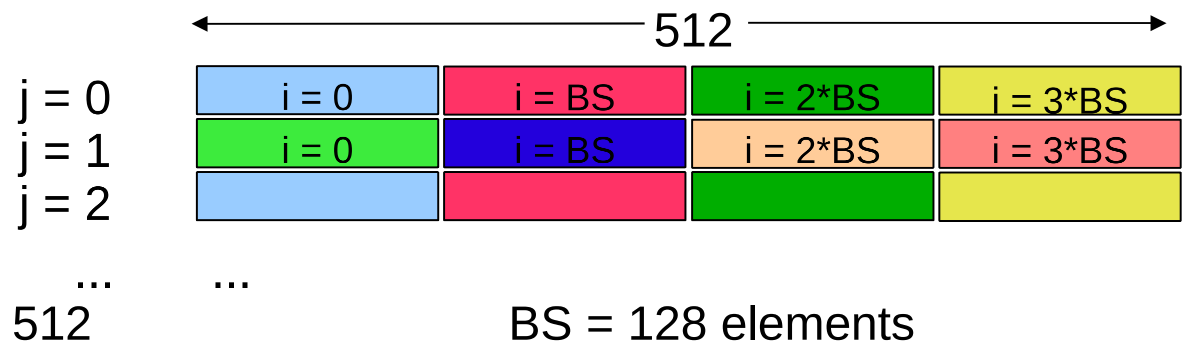
- Variables and functions can also be "targeted"

```
#pragma omp target device(smp,cuda)
int perm[512] = {
   151, 160, 137, 91, 90, 15, 131, 13, 201, 95, 96, 53, 194, 233,
   7, 225, 140, 36, 103, 30, 69, 142, 8, 99, 37, 240, 21, 10, 23,
  ...
};
```

```
#pragma omp target device(smp,cuda)
float grad(int hash, float x, float y, float z)
{
   int h = hash & 15;              // Convert low 4 bits of hash code
   float u = (h < 8) ? x : y;      // into 12 gradient directions.
   float v = (h < 4) ? y : (h == 12 || h == 14) ? x : z;

   u = (h & 1) == 0 ? u : -u;
   v = (h & 2) == 0 ? v : -v;
   return u + v;
}
```

OmpSs Julia Set

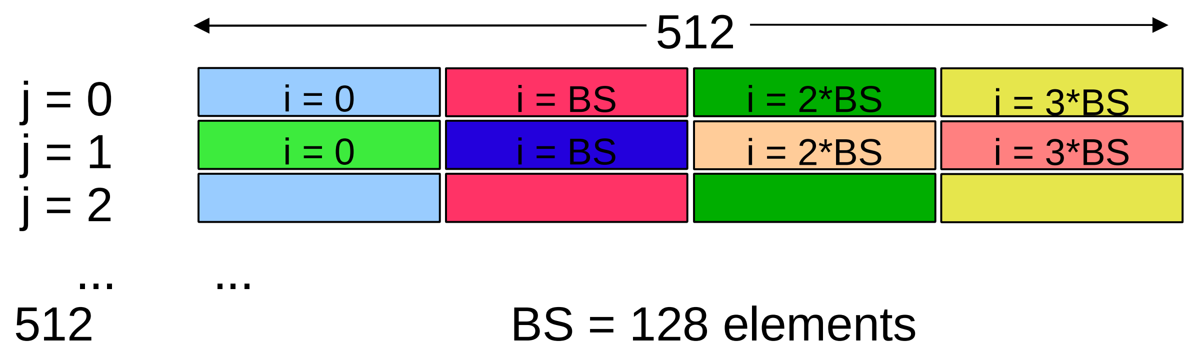
- Coding




```
for (j = 0; j < img_height; j+=BSY) {
    int offset = j;
    out = (struct pixel_group_s *) &outBuff[compute_frame][j*rowstride*4];
#pragma omp target device(cuda) copy_in (jc) \
                                                       copy_out([output_size]   out)
#pragma omp task shared(out, jc) \
                  firstprivate(currMu, rowstride, BSx, BSY, offset, ntasks)
    {
       // kernel
    }
}
#pragma omp taskwait
```

OmpSs Julia Set

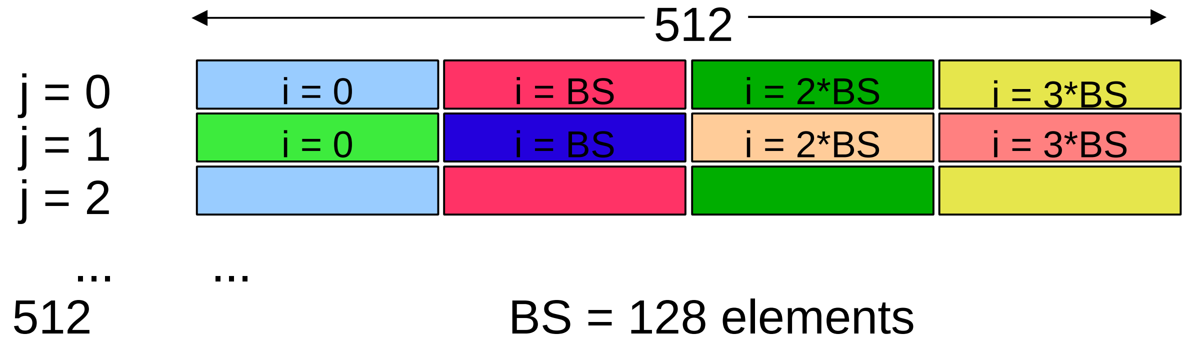
- Julia Set CUDA kernel task



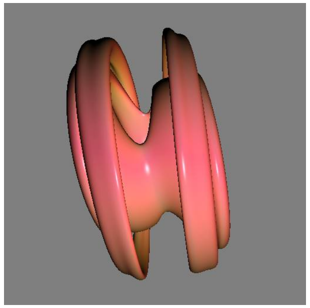

```
#pragma omp task shared(out, jc) \
            firstprivate(currMu, rowstride, BSx, BSY, offset, ntasks)
  {
    dim3 dimBlock (BSx, 1);
    dim3 dimGrid (rowstride/4/dimBlock.x,       // /4 due to vector size
                                BSY/dimBlock.y);

    compute_julia_kernel <<<dimGrid, dimBlock>>> (
          currMu, (uchar16 *) out, rowstride, jc, offset
    );
  }
```

OmpSs Julia Set

- Julia Set CUDA kernel

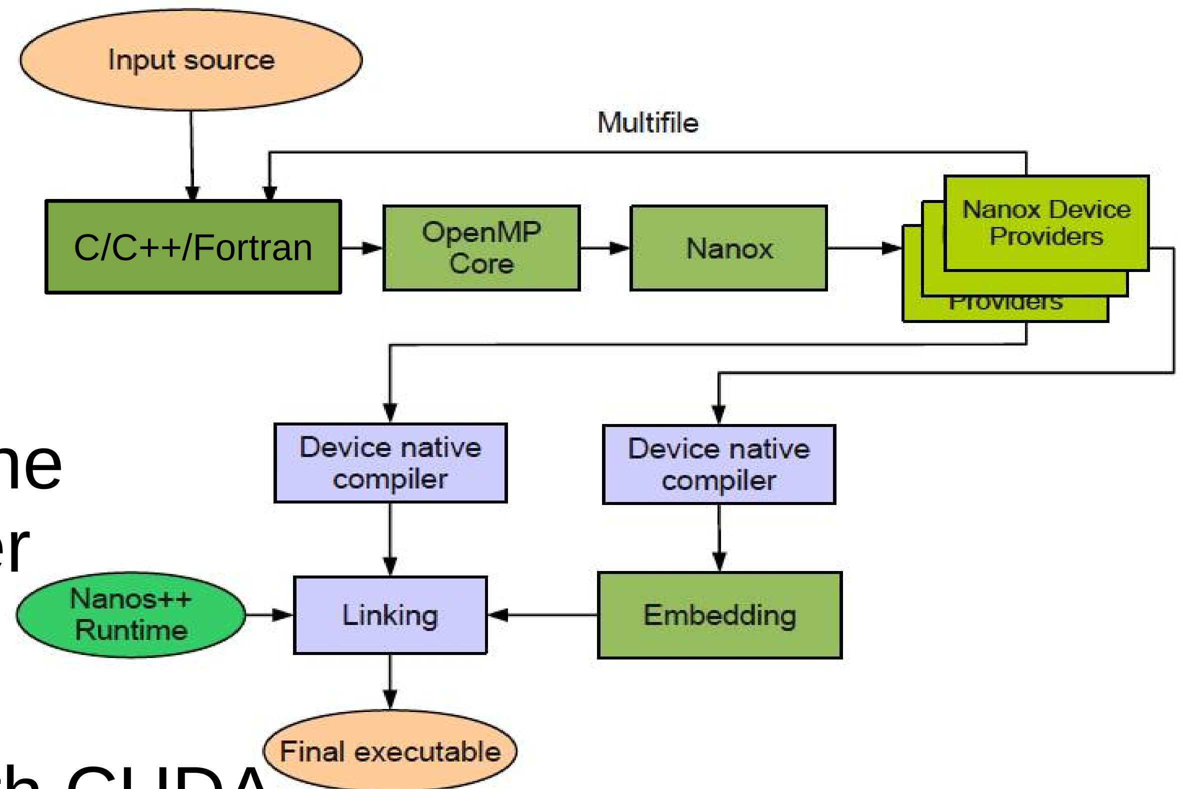



```
#pragma omp target (cuda)
__global__ void compute_julia_kernel (const float4 muP,
                                       uchar16 * framebuffer,
                                       int rowstride,
                                       const struct julia_context jc,
                                       int offset)
{ ...
  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;
  ...
  fragmentShader4(rO, curr_dir, mu, epv,
                  light, jc->maxIterations, renderShadows, &pcolors);
  framebuffer[(j*rowstride)/4 + i] = pcolors;
}
```



Mercurium compiler

- Transforming
  - Directives to runtime calls
- Compiling
  - Natively with the Nvidia compiler
- Embedding
  - Object files with CUDA embedded on host files



Hands-on

- Enter **Minotauro {mt1, mt2}.bsc.es**
- Copy/unpack the file
  - tar zxf /tmp/tutorial_PATC_ompss+gpus.tar.gz
- Enter the “tutorial_PATC_ompss+gpus” directory
- There is a “env.sh” file to setup environment for compiling and executing: source ./env.sh
- Each application directory has a Makefile
  - Compile using “make”
    - It uses **mcc / mcxx / mnvcc / mnvcxx --ompss**
- Execute using the provided “job.sh” script
    - It uses **NX_GPUS=N** to specify the number of GPUS
    - **NX_INSTRUMENTATION=extrae** to get traces

Hands-on

- Look at the application and the Makefile
- Search for places to set directives
  - We've set some hints... or even correct directives on some places!!!
- Taskify + annotate data and code needs
- Compile and run
  - Each directory has...
    - job.sh to be submitted to the queuing system for execution
- Get traces and compare execution with one and two GPUs
- Suggested: **bsop**, **perlin_noise**, **stream**, **nbody** and **multisort**