Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Programming with OmpSs

Support for heterogeneous platforms

Computer Sciences Research Dept.
BSC

Agenda

- Motivation
- Leveraging CUDA and GPUs
- Examples

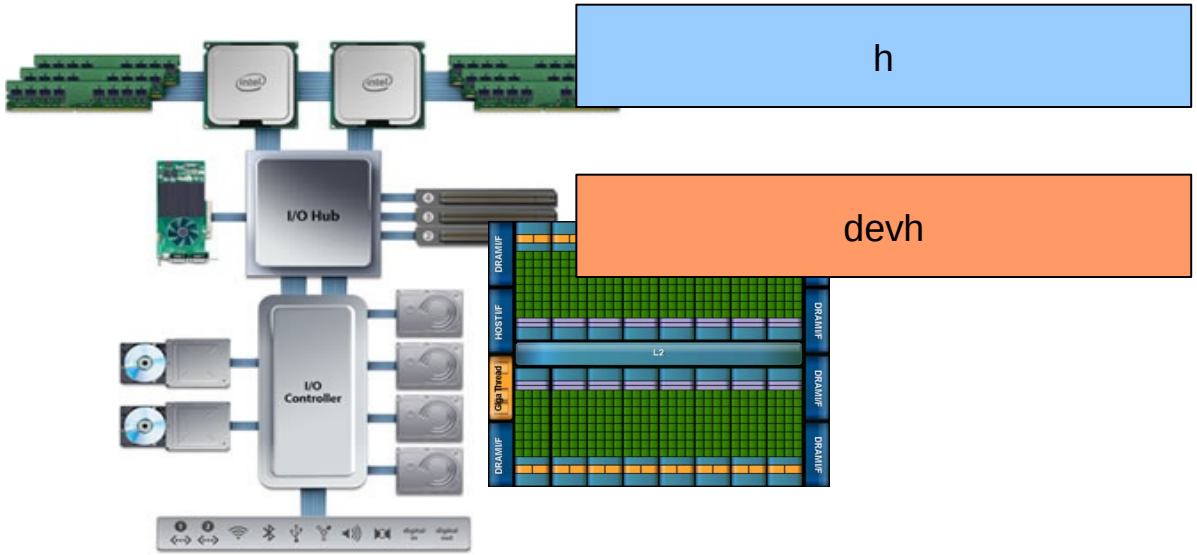
- Contact: pm-tools@bsc.es
- Source code available from http://pm.bsc.es/ompss/

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

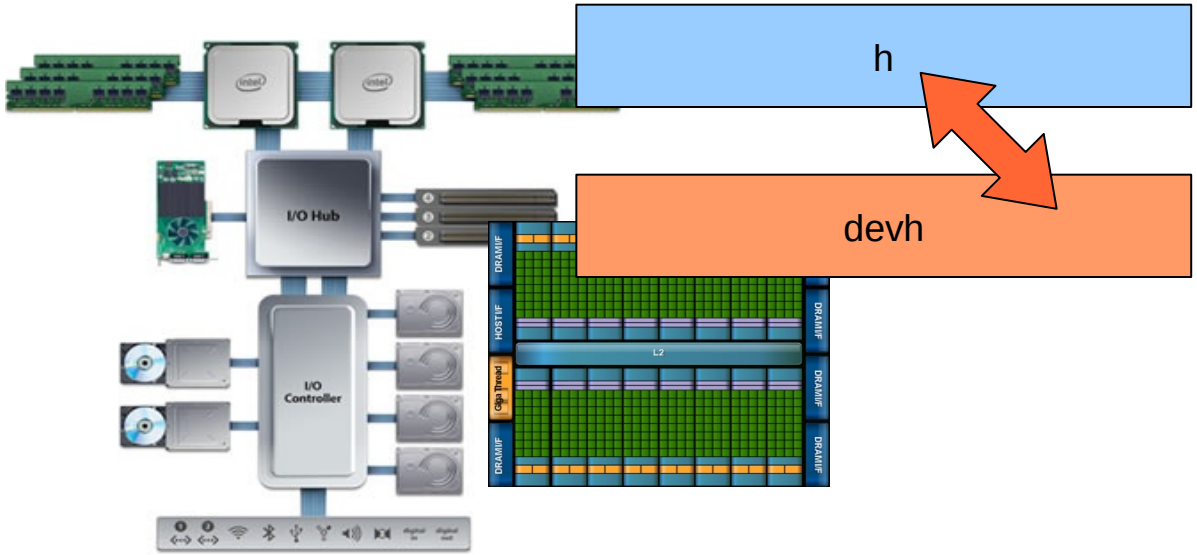


Motivation

- 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]);
}
```

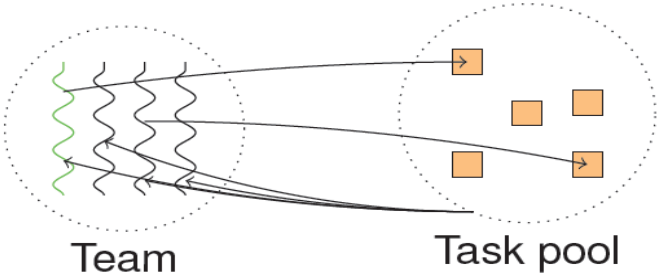

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Proposal: OmpSs

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

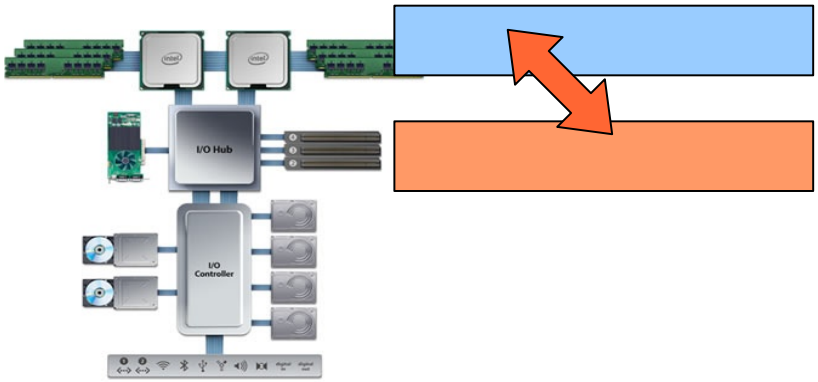
OmpSs: execution model

- 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



Heterogeneity: the target directive

#pragma omp target [clauses]

- Specifies that the code after it is for a specific device

- The compiler parses the specific syntax of that device and hands the code over to the appropriate back end compiler

- Currently supported devices:
  - smp: default device. Back end compiler to generate code can be gcc, icc, xlc,….
  - cuda: cuda code is separated to a temporary file and handed over to nvcc for code generation.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Heterogeneity: the copy clauses

#pragma omp target [clauses]

- If the code is a task (target pragma immediately followed by a task pragma)

- Some devices (cuda) have their private physical address space.
- The **copy_in, copy_out,** an **copy_inout** clauses have to be used to specify what data has to be maintained consistent between the original address space of the program and the address space of the device.
- The **copy_deps** is a shorthand to specify that for each input/output/inout declaration, an equivalent copy_in/out/inout is used.
- A copy clause has to be provided for every data required by the task for devices with private address space
    - Relaxed consistency results otherwise
- Tasks on the original program device (smp) also have to specify directionality to ensure consistency for those arguments referenced in some other device.
- The default **taskwait** semantic is to ensure consistency of all the data in the original program address space.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Heterogeneity: the copy clauses

- **copy_in**: requests a consistent copy of the data to be available in the device address space when the task is executed
  - It may require a data transfer
    - Automatically performed by the runtime
    - Scheduled at its convenience
  - Data may be already in the device
    - The runtime keeps track of all data replication and ensures consistency
  - Address translation may be needed
- **copy_out**: specifies that the task produces that data
  - The runtime has to use this information to ensure data consistency
  - May require a transfer sometime after the task completes
    - Automatically performed by the runtime, at its convenience
- **copy_inout**: specifies that a consistent copy of the data is required on input and a new value is produced for which consistency has to be maintained

Heterogeneous 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])
    {
        <<< CUDA kernel_1_invocation >>>
    }
#pragma omp target device(cuda) copy_deps
#pragma omp task input (c[J;BK])...
    {
        <<< CUDA kernel_2_invocation >>>
    }
 }
```


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

CUDA device specifics

- The **task body** CUDA code is actually **executed** by one thread in the **host**
- This thread can **launch kernels**, passing them arguments including pointers
- The **runtime does the memory allocation and deallocation on the device** as well as the data transfers for **copy variables**
- The **task body** in CUDA **can not reference the pointers it is passed** as they have been translated by the runtime to point to the copy of the object in the device physical address space

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 < size; 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: a reasonable merge/evolution of OpenMP

blue already in OpenMP
red extensions based on StarSs

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 **consistent** data is **accessible** in the address space of the device

```
#pragma omp task [ input (...)] [ output (...)] [ inout (...)] [ concurrent (...)]

{code block or function}
```

To compute dependences

To allow concurrent execution of commutative tasks

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

Master waits for sons or specific data availability

Relax consistency to main program

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Heterogeneity: the target directive

```
#pragma omp target device (cuda) copy_deps
#pragma omp task input ([size] c) output ([size] b)
void scale_task_cuda(double *b, double *c, double scalar, int size)
{
   dim3 dimBlock, dimGrid;
   dimBlock.x = 128; dimBlock.y = dimBlock.z = 1;
   dimGrid.x = size/128+1;
   scale_kernel<<<dimGrid,dimBlock>>>(size, 1, b, c, scalar);
}
```

Executed in the host

Kernel launched to the device.
b and c are pointers to the memory allocated by the runtime on the device

```
#pragma 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)
{
for (int j=0; j < BSIZE; j++) b[j] = scalar*c[j];
}
```

```
        double A[1024], B[1024], C[1024]
            double D[1024], E[1024];

main(){
   …
   scale_task_cuda(A, B, 10.0, 1024); //T1
   scale_task_cuda(B, A, 0.01, 1024); //T2
   scale_task (C, A, 2.0, 1024);      //T3
   scale_task_cuda (D, E, 5.0, 1024); //T4
   scale_task_cuda(B, C, 3.0, 1024);  //T5
   #pragma omp taskwait
   // can access any of A,B,C,D,E
}
```

A, B have to be transferred to device before task execution

No data transfer. Will execute after T1

A, has to be transferred to host.
Can be done in parallel with T2

D, E, have to be transferred to GPU.
Can be done at the very beginning

C has to be transferred to GPU.
Can be done when T3 finishes

Copy D, E back to host

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

CUDA examples

```
void cuRiemann(int narray, double Hsmallr, double Hsmallc, double Hgamma,   //
        int Hniter_riemann, int Hnvar, int Hnxyt, int slices, int Hnxystep,  //
        double *qleft, double *qright, double *qgdnv, long *sgnm) {
   #pragma omp target device (cuda) copy_deps
   #pragma omp task input ([Hnvar][Hnxystep][Hnxyt]qleft, \
                           [Hnvar][Hnxystep][Hnxyt]qright, \
                           [Hnxystep][narray]sgnm) \
                  output([Hnvar][Hnxystep][Hnxyt]qgdnv) \
                  firstprivate(narray, Hsmallr, Hsmallc, Hgamma, Hniter_riemann, \
                               Hnvar, Hnxyt, slices, Hnxystep)
   {
      dim3 block, grid;
      SetBlockDims(Hnxyt * slices, 192, block, grid);
      Loop1KcuRiemann <<< grid, block >>> (narray, Hsmallr, Hsmallc, Hgamma,   //
          Hniter_riemann, Hnvar, Hnxyt, slices, Hnxystep,  //
          qleft, qright, qgdnv, sgnm );
      cudaDeviceSynchronize();
      if (Hnvar > IP + 1) {
          Loop10KcuRiemann <<< grid, block >>> (narray, Hnvar, Hnxyt, slices, //
             Hnxystep, qleft, qright, qgdnv, sgnm );
    cudaDeviceSynchronize();
}
#pragma omp taskwait
}
```

Inlined pragma needs explicit privatization of read scalars for CUDA tasks

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Heterogeneity: relaxing consistency

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

No copy of consistent instance of input to smp.
Definition of new value for output

```
double A[1024], B[1024], C[1024]
double D[1024], E[1024];

main(){
   …
   scale_task_cuda (A, B, 1.0, 1024);   //T1
   scale_task_cuda (B, A, 0.1, 1024);   //T2
   scale_task_nocpin (C, A, 2.0, 1024); //T3
   scale_task_cuda (D, E, 5.0, 1024);   //T4
   scale_task_cuda (C, B, 3.0, 1024);   //T5

   #pragma omp taskwait noflush

   printf ("%f, %f\n", D[1],E[512]
   scale_task (E, B, 0.5, 1024);        //T6
   scale_task_nocpin (D, C, 0.5, 1024); //T7

   #pragma omp taskwait noflush
}
```

A, B have to be transferred to device before task execution

No data transfer. Will execute after T1

A, is not transferred to host.
T3 will execute after T1 finishes but with old value of A

D, E, have to be transferred to GPU.
Can be done at the very beginning

C value updated by T3 has to be transferred to GPU.
Can be done when T3 finishes.

No copy of A, B, C, D, E back to host

Will print original unmodified value of D and E

Copy B and E back to host before execution

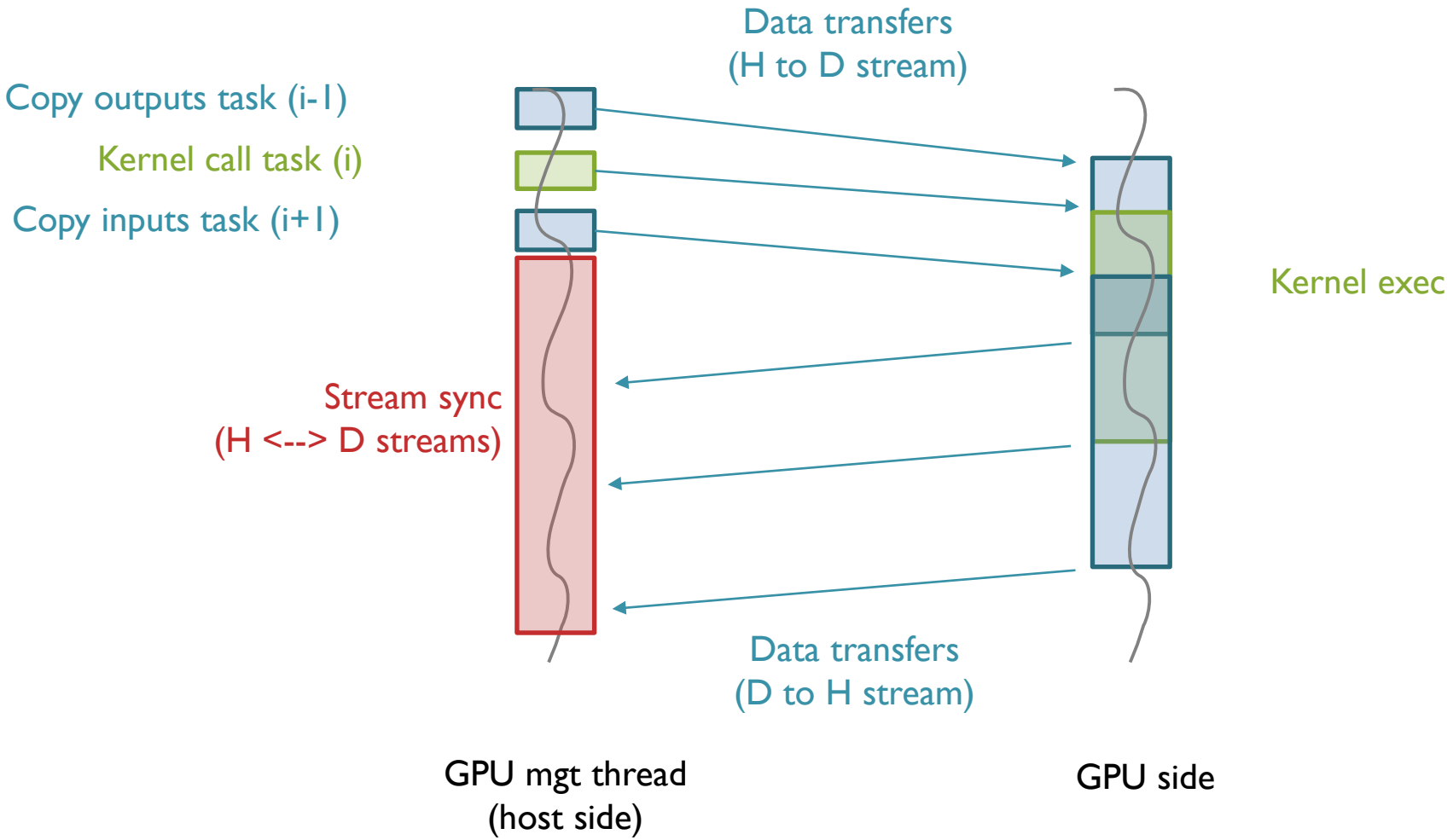
No copy of C back to host
No copy of D, back to host but new value defined by T7

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs runtime features

- Automatic and configurable overlapping of data transfers and computation
  - Input / output / inout
    - with the kernel executions
- Data prefetching
  - Transfer input data to GPU memory
    - well before tasks need the data

OmpSs runtime features



OmpSs runtime features

- Configurable data cache policy
  - Write-through
    - after the execution of each task, output data is updated on the host
  - Write-back
    - only updates host data if needed
  - Nocache
    - cache is disabled
      - not recommended, only for testing
- Configurable limit on GPU memory usage

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs runtime features

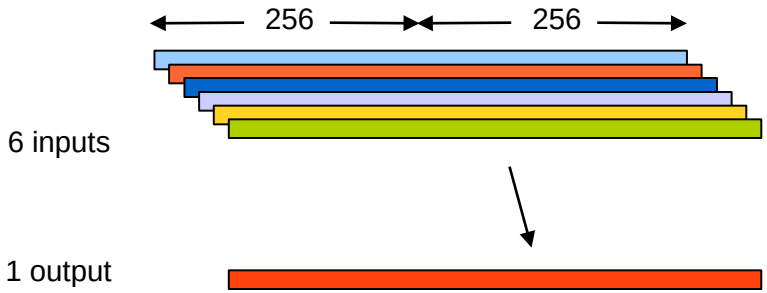
- CUBLAS support and context management
  - No need to call cublasInit / cublasShutdown
    - NX_GPUCUBLASINIT
  - CUBLAS functions can also be scheduled to specific streams to enable overlapping
    - Runtime takes care of properly allocation of streams/handles

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Use of input/output/inout

- OmpSs BlackScholes

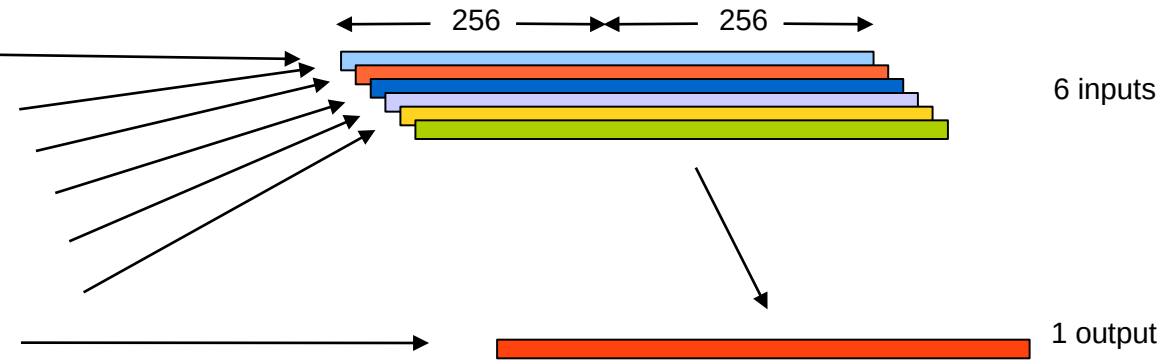
```
#pragma omp target device(cuda) copy_deps
#pragma omp task input ( \
                [global_work_group_size]   cpflag_fptr, \
                [global_work_group_size]   S0_fptr, \
                [global_work_group_size]   K_fptr, \
                [global_work_group_size]   r_fptr, \
                [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
            }
```



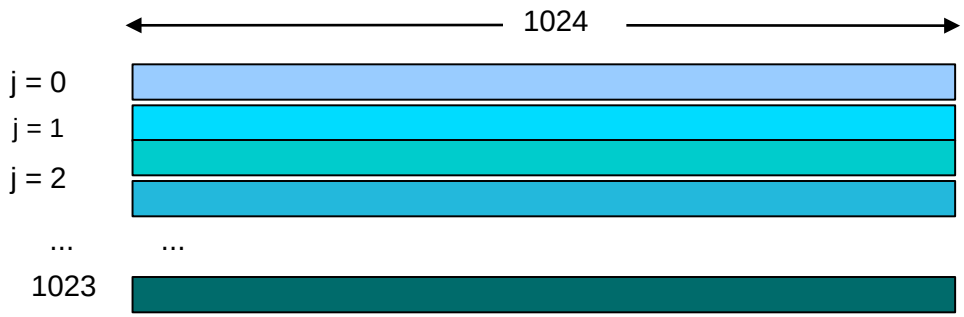
BlackScholes

- 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 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”

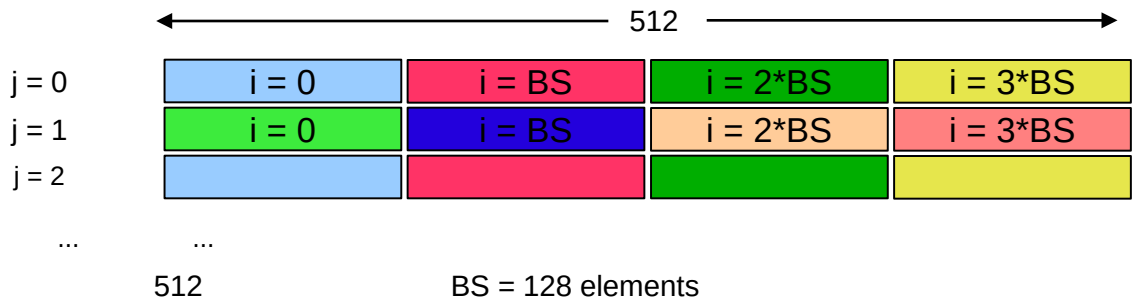
```
#pragma omp target device(smp,cuda)
int perm[512] = {
   151, 160, 137, 91, 90, 15, 131, 13, 201, 95, 96, 53, 194, 233,
   7, 225, 140, 36, 103, 30, 69, 142, 8, 99, 37, 240, 21, 10, 23,
  ...
};
```

```
#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;
}
```

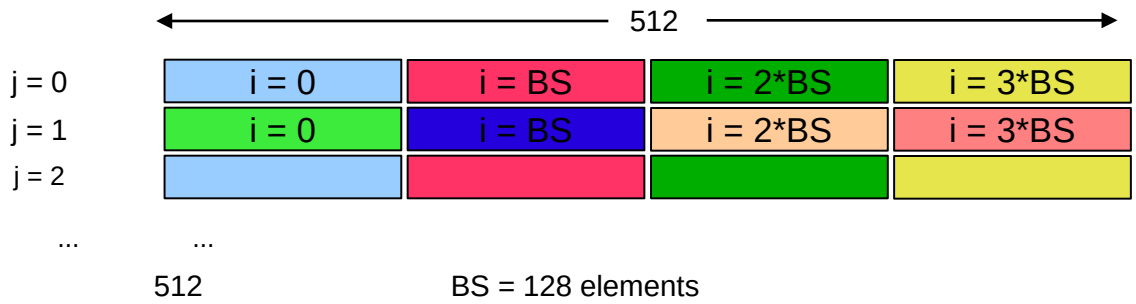

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Coding Julia Set




```
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
```

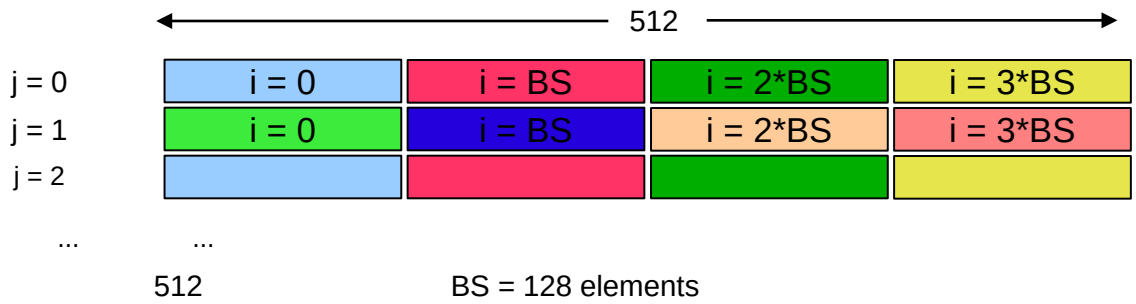
Julia Set CUDA kernel task



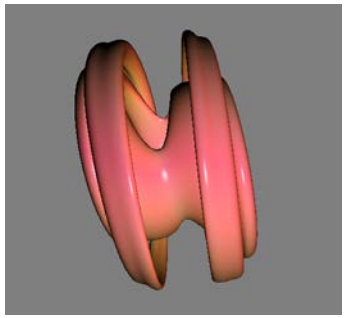

```
#pragma omp task shared(out, jc) \
                firstprivate(currMu, rowstride, BSx, BSY, offset, ntasks)
  {
    dim3 dimBlock (BSx, 1);
    dim3 dimGrid (rowstride/4/dimBlock.x,      // /4 due to vector size
                  BSY/dimBlock.y);

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

Julia Set CUDA kernel



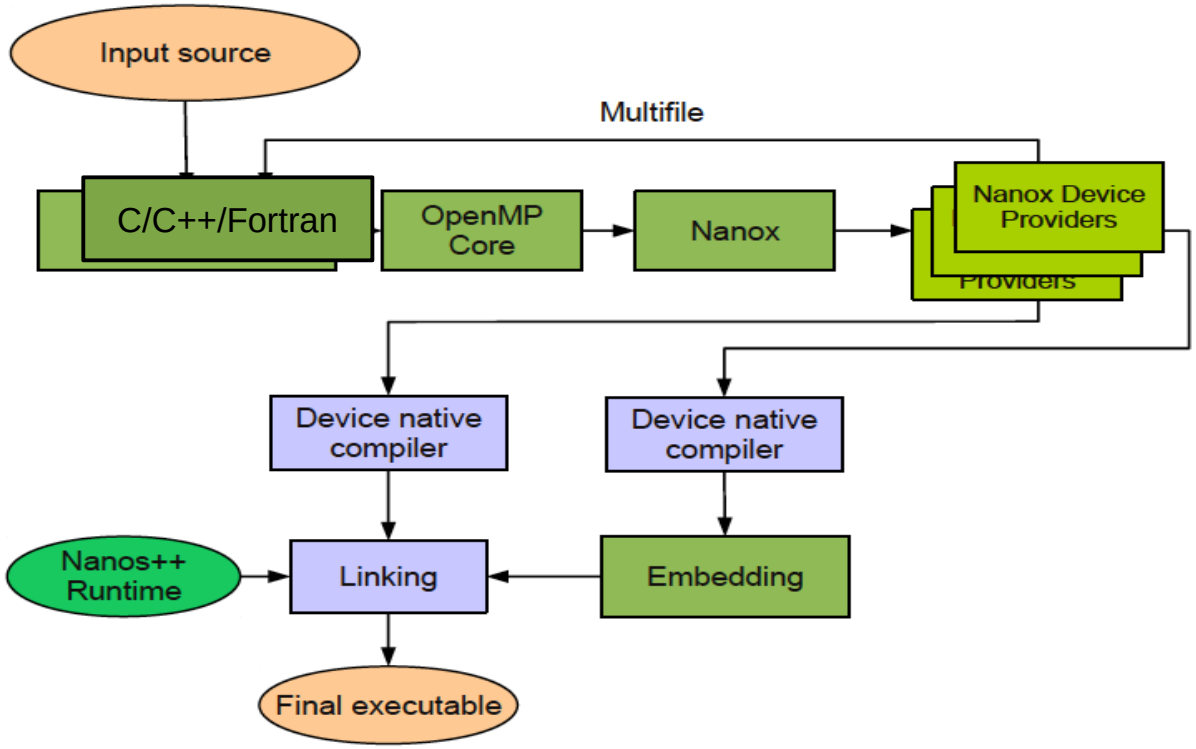

```
#pragma omp target (cuda)
__global__ void compute_julia_kernel (const float4 muP,
                                      uchar16 * framebuffer,
                                      int rowstride,
                                      const struct julia_context jc,
                                      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;
}
```



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Mercurium compiler

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



Conclusions

- Future programming models should:
  - Enable productivity and portability
  - Support for heterogeneous/hierarchical architectures
  - Support asynchrony → global synchronization in systems with large number of nodes is not an answer anymore
  - Be aware of data locality
- OmpSs is a proposal that enables:
  - Incremental parallelization from existing sequential codes
  - Data-flow execution model that naturally supports asynchrony
  - Nicely integrates heterogeneity and hierarchy
  - Support for locality scheduling
  - Active and open source project:

    http://pm.bsc.es/ompss

Barcelona Supercomputing Center
Centro Nacional de Supercomputación