Supercomputing at the Service of the Science

Programming Models

Rosa M. Badia
Alejandro Duran
Xavier Martorell

Barcelona Supercomputing Center

Aules d'Empresa
Facultat d'Informàtica de Barcelona
Barcelona, January 27th, 2011

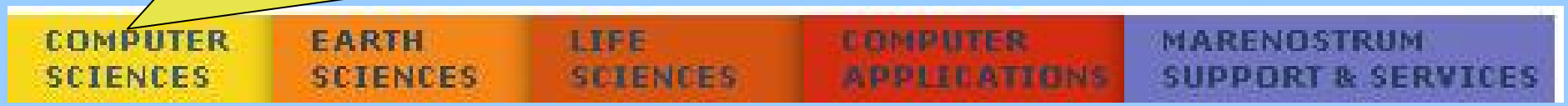
Outline

- Introduction to the BSC/Programming Models
- Motivation
- Proposal: OmpSs
- Examples
  - Matrix Multiply
  - BlackScholes
  - Perlin noise
  - Krist
  - Julia Set

BSC/Programming Models

- OmpSs, StarSs, OpenMP, MPI
- Mercurium compiler
- Nanox runtime system
- SMPs/GPUs/FPGAs/Cell B.E./Clusters

Computer Architecture Operating System Interface
Computer Architecture
Performance Tools
Programming Models
Grid Computing
Autonomic Systems and e-Business Platforms
Storage Systems



Barcelona Supercomputing Center

BSC/Programming Models

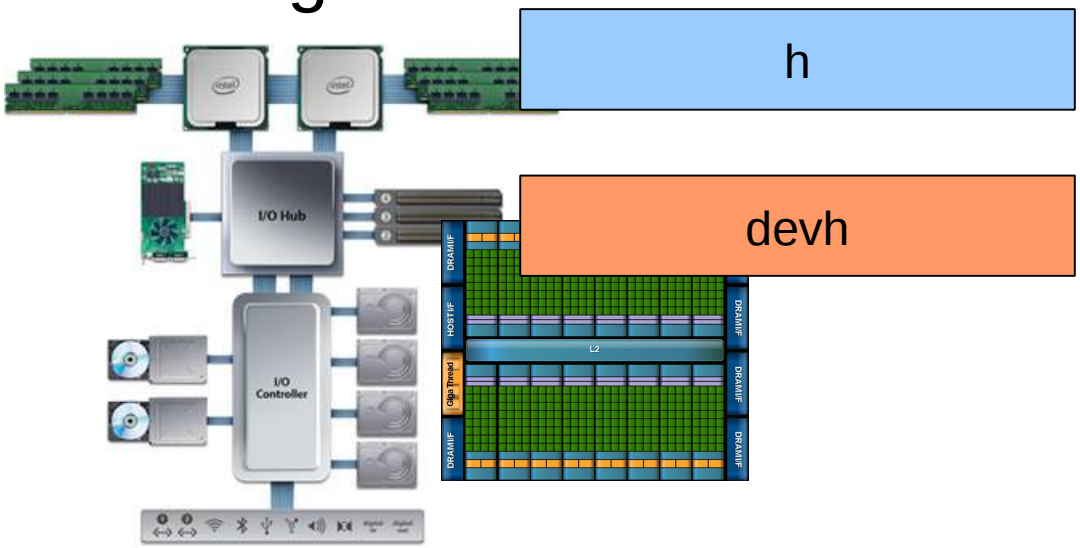
- Selected topic
  - OmpSs programming for SMPs and GPUs
  - Compare programming on OmpSs and CUDA/OpenCL
  - Examine some applications exploiting the model
  - Perform actual runs in real hardware

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

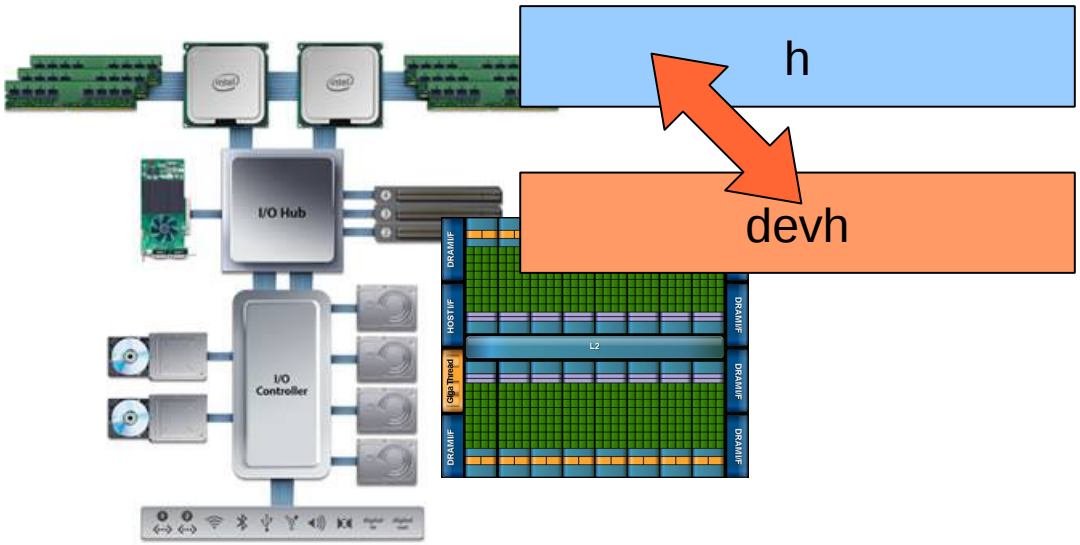


Motivation

- Data copies to/from device memory
  - copy_in/copy_out

cudaMemcpy(devh,h,sizeof(*h)*nr*DIM2_H, cudaMemcpyHostToDevice);

- Increase options for data overwrite



Motivation

- Manual work scheduling
  - Determine number of blocks/threads

```
dim3 threads(256);
int blocks = nr/threads.x;
if (blocks*threads.x < nr)
   blocks++;
dim3 grid(blocks);
```

  - Compute amount of shared memory to use

```
cudaGetDeviceProperties(&prop,0);
int sharedsize = prop.sharedMemPerBlock-1024;
int maxatoms = sharedsize/sizeof(TYPE_A);
```

  - Invoke the kernel on the iteration space

```
cstructfac<<<grid, threads, sharedsize>>>
                                    (na,nr,maxatoms,f2,deva,devh,devE,devv);
```

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
  - Parallelism, iteration scheduling & tasking
- StarSs expressiveness
  - Data direction (input/output) hints
  - Detection of dependences at runtime
  - Automatic data movement
- OpenCL SIMDization

Proposal: OmpSs

- Objective
  - Minimal set of annotations

```
    float16 a[N];     // we have a machine with long
    float16 b[N];     // 16-float vector registers
    float16 c[N];

    for (i=0; i<N; i++) {
#pragma omp target device(cuda)
#pragma omp task input (a, b) input (c)
     {
        c[i] = a[i] + b[i];
     }
    }
```

OpenMP expressiveness

- Parallelism in CUDA/OpenCL applications is easily expressed using OpenMP
  - Data parallelism

```
for (i=0; i < N; i++) {
   for (j=0; j < M; j++) {
      kernel (i, j);
   }
}
```

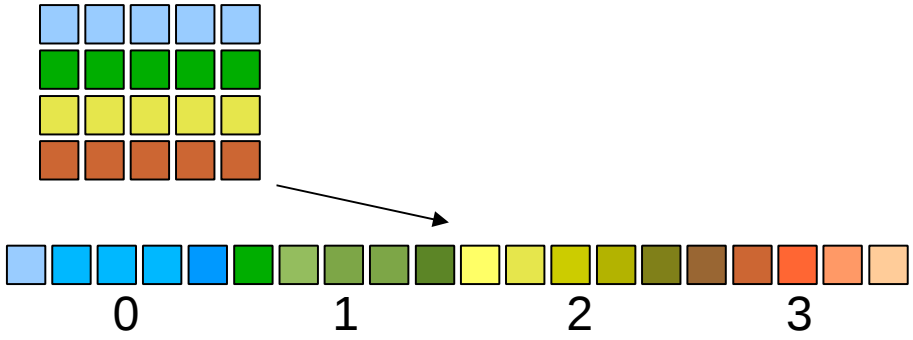
```
dim3 threads(16,16);
dim3 grid   (N/16, M/16);
kernel <<<grid, threads>>>(...);
```

```
for (I=0; I < N; I+=16) {
   for (J=0; J < M; J+=16) {

      for (i=I; i<I+16; i++) {
         for (j=J; j<J+16; j++) {
            kernel (i, j);
         }
      }
   }
}
```

OpenMP iteration scheduling

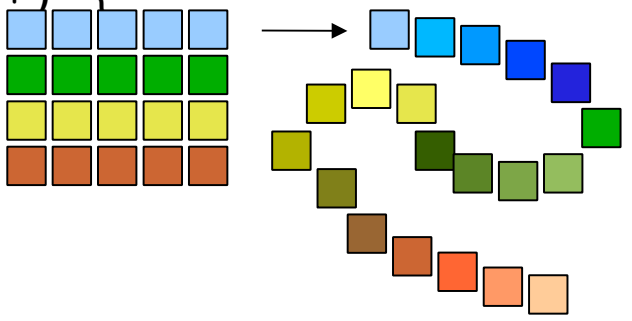
- Options
  - Parallel loops

```
#pragma omp parallel for collapse(2) \
                         private(I,J,i,j)
for (I=0; I < N; I+=16) {
   for (J=0; J < M; J+=16) {
      for (i=I; i<I+16; i++) {
         for (j=J; j<J+16; j++) {
            kernel (i, j);
         }
      }
   }
}
```

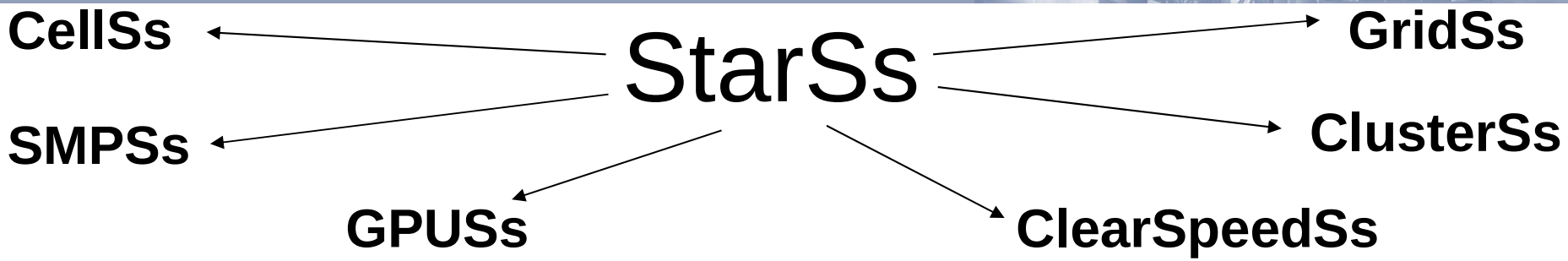


  - Tasks

```
for (I=0; I < N; I+=16) {
   for (J=0; J < M; J+=16) {
#pragma omp task
      for (i=I; i<I+16; i++) {
         for (j=J; j<J+16; j++) {
            kernel (i, j);
         }
      }
   }
}
#pragma omp taskwait
```



Dynamic/ static processor assignment



- Node level prog. model for C/C++/Fortran
  - Natural support for heterogeneity
  - Nicely integrates in hybrid MPI/StarSs
- Programmability
  - Incremental parallelization
  - Abstract/separate algorithm from resources
  - Disciplined programming
- Portability
  - “Same” source code runs on “any” machine
  - Optimized task implementations will result in better performance
  - Single source
- Performance
  - Asynchronous (data-flow) execution and locality awareness
  - Intelligent runtime, specific for each platform

A sequential program...

```
void vadd3 (float A[BS], float B[BS],
            float C[BS]);

void scale_add (float sum, float A[BS],
                float B[BS]);

void accum (float A[BS], float *sum);
```

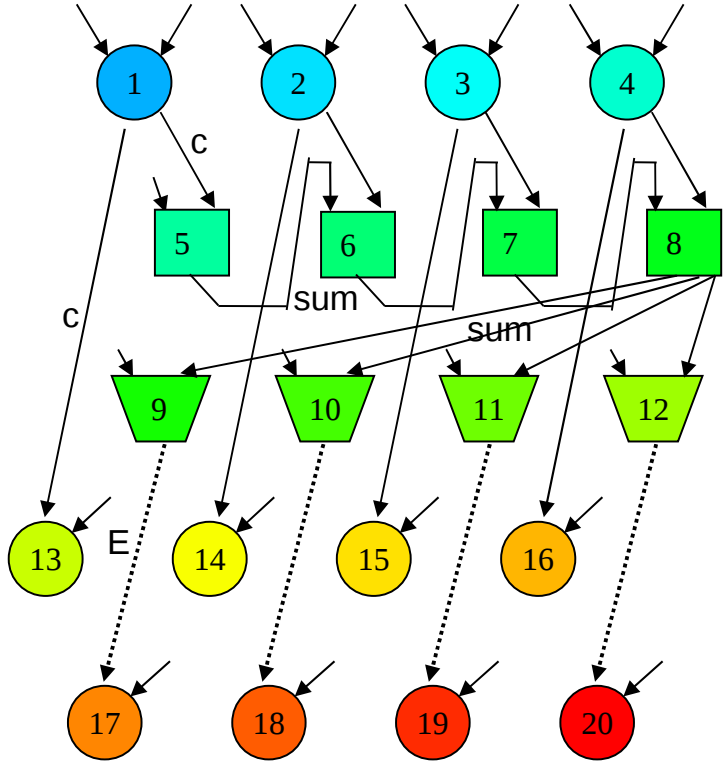
```
for (i=0; i<N; i+=BS)              // C=A+B
   vadd3 ( &A[i], &B[i], &C[i]);
...
for (i=0; i<N; i+=BS)            // sum(C[i])
   accum (&C[i], &sum);
...
for (i=0; i<N; i+=BS)              // B=sum*E
   scale_add (sum, &E[i], &B[i]);
...
for (i=0; i<N; i+=BS)              // A=C+D
   vadd3 (&C[i], &D[i], &A[i]);
...
for (i=0; i<N; i+=BS)              // E=G+F
   vadd3 (&G[i], &F[i], &E[i]);
```

... taskified ...

```
#pragma omp task input(A, B) output(C)
void vadd3 (float A[BS], float B[BS],
            float C[BS]);
#pragma omp task input(sum, A) output(B)
void scale_add (float sum, float A[BS],
                float B[BS]);
#pragma omp task input(A) inout(sum)
void accum (float A[BS], float *sum);
```

```
for (i=0; i<N; i+=BS)              // C=A+B
   vadd3 ( &A[i], &B[i], &C[i]);
...
for (i=0; i<N; i+=BS)            // sum(C[i])
   accum (&C[i], &sum);
...
for (i=0; i<N; i+=BS)              // B=sum*E
   scale_add (sum, &E[i], &B[i]);
...
for (i=0; i<N; i+=BS)              // A=C+D
   vadd3 (&C[i], &D[i], &A[i]);
...
for (i=0; i<N; i+=BS)              // E=G+F
   vadd3 (&G[i], &F[i], &E[i]);
```

Compute dependences @ task instantiation time



Color/number: order of task instantiation
Some antidependences covered by flow dependences not drawn

... and executed in a data-flow model

```
#pragma omp task input(A, B) output(C)
void vadd3 (float A[BS], float B[BS],
            float C[BS]);
#pragma omp task input(sum, A) inout(B)
void scale_add (float sum, float A[BS],
                float B[BS]);
#pragma omp task input(A) inout(sum)
void accum (float A[BS], float *sum);
```

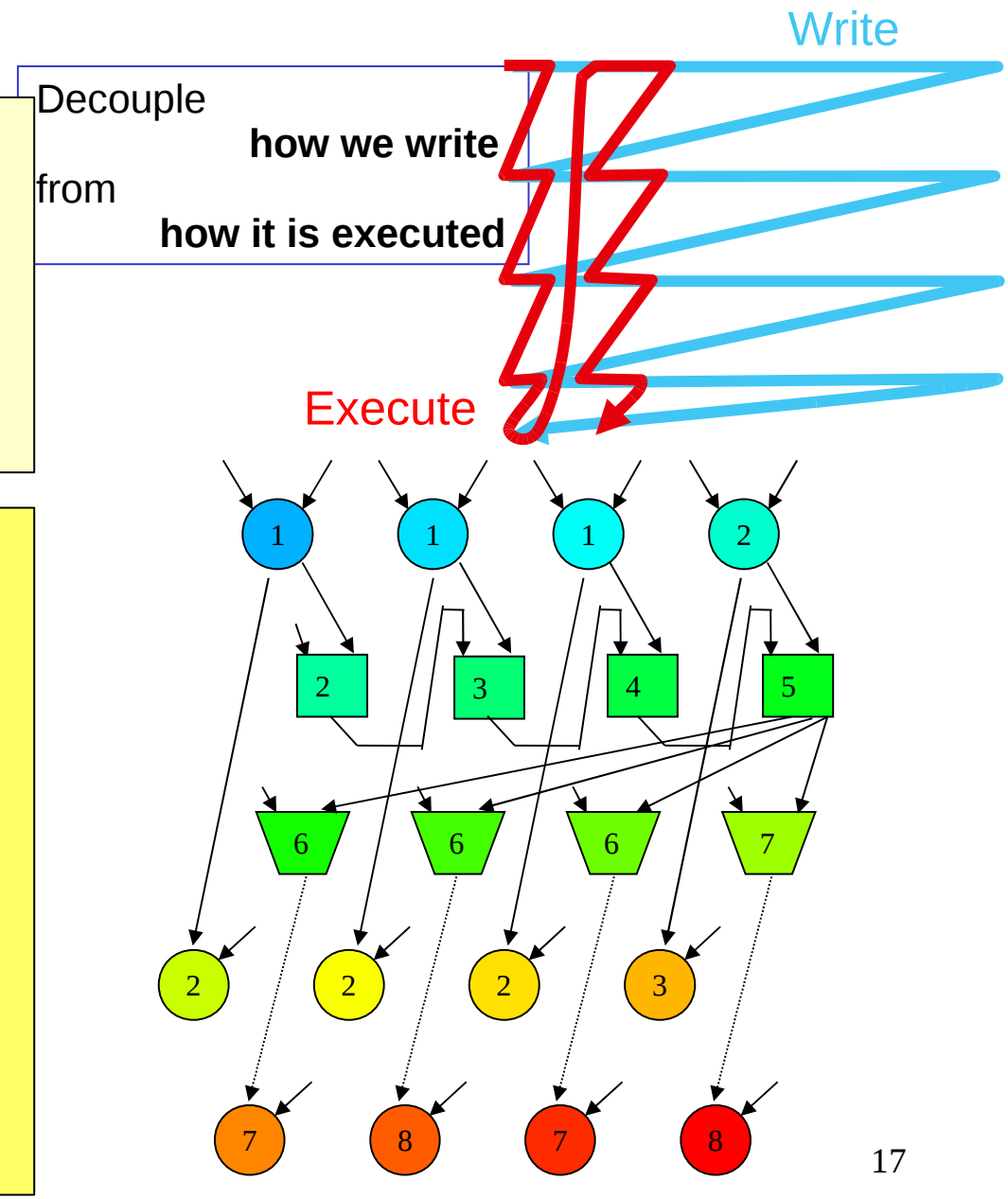
```
for (i=0; i<N; i+=BS)                // C=A+B
   vadd3 ( &A[i], &B[i], &C[i]);
...
for (i=0; i<N; i+=BS)              // sum(C[i])
   accum (&C[i], &sum);
...
for (i=0; i<N; i+=BS)                // B=sum*E
   scale_add (sum, &E[i], &B[i]);
...
for (i=0; i<N; i+=BS)                // A=C+D
   vadd3 (&C[i], &D[i], &A[i]);
...
for (i=0; i<N; i+=BS)                // E=G+F
   vadd3 (&G[i], &F[i], &E[i]);
```

Decouple

how we write

from

how it is executed



Color/number: a possible order of task execution

The potential of data access information

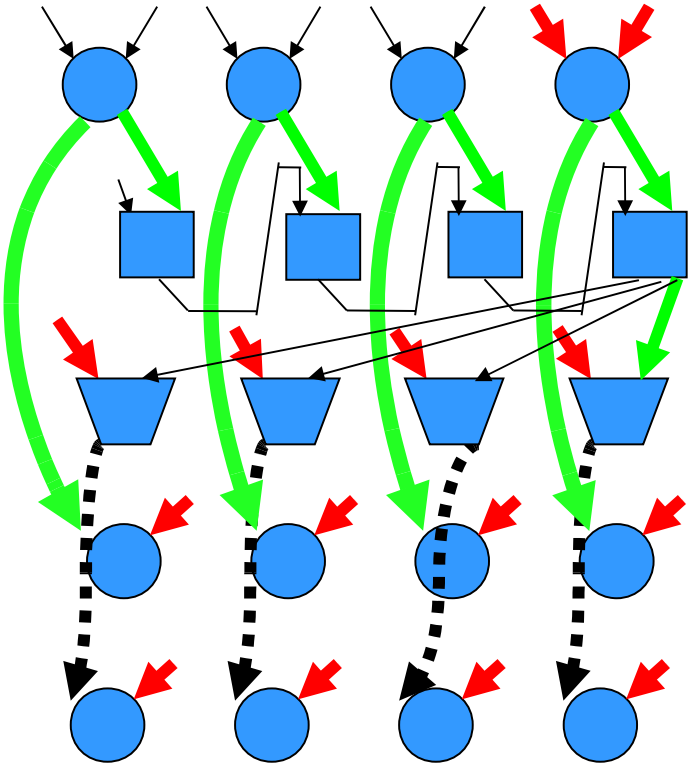
Flat global address space seen by programmer

Flexibility to dynamically traverse dataflow graph “optimizing”

- Concurrency. Critical path
- Memory access: data transfers performed by run time

Opportunities for

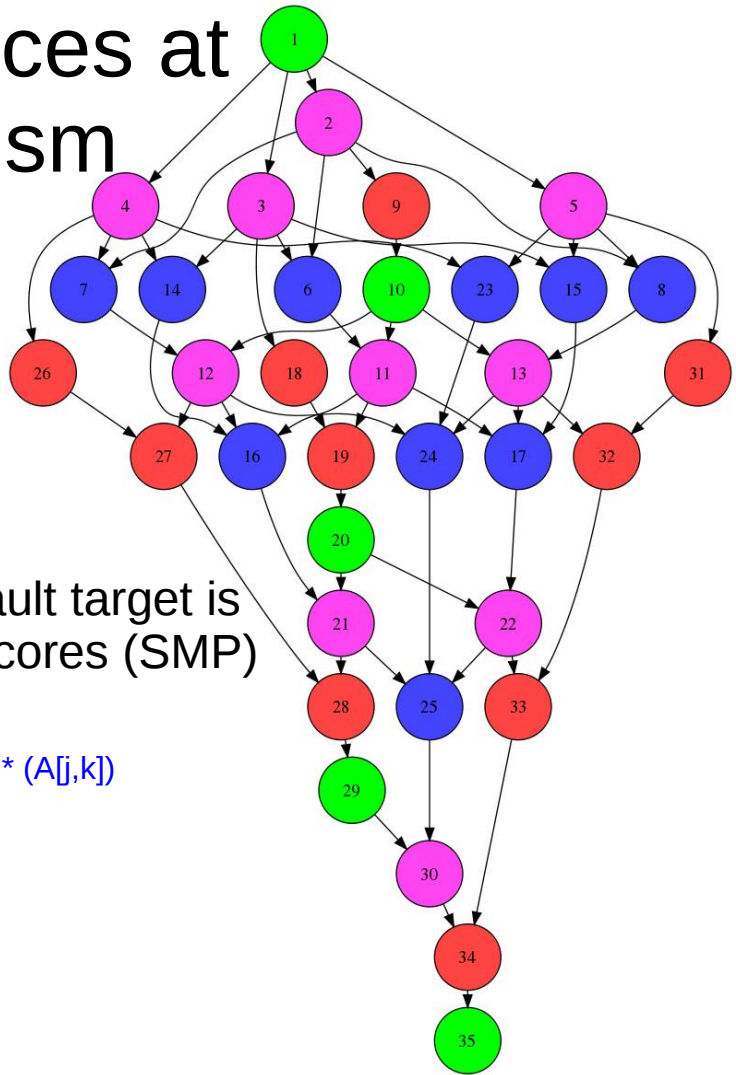
- Prefetch
- Reuse
- Eliminate antidependences (rename)
- Replication management
  - Coherency/consistency handled by the runtime



StarSs Cholesky

- Input/output direction hints
- Enable detection of dependences at runtime, and increase parallelism

```
#pragma omp task inout([NB*NB] A)
void spotrf_tile(float *A,long NB);

#pragma omp task input([NB*NB] A, [NB*NB] B) inout([NB*NB] C)
void sgemm_tile(float  *A, float *B, float *C, unsigned long NB);

#pragma omp task input([NB*NB] T) inout([NB*NB] B)
void strsm_tile(float *T, float *B, unsigned long NB)

#pragma omp task input([NB*NB] A) inout([NB*NB] C)
void syrk_tile( float *A, float *C, long NB)
```



Default target is host cores (SMP)

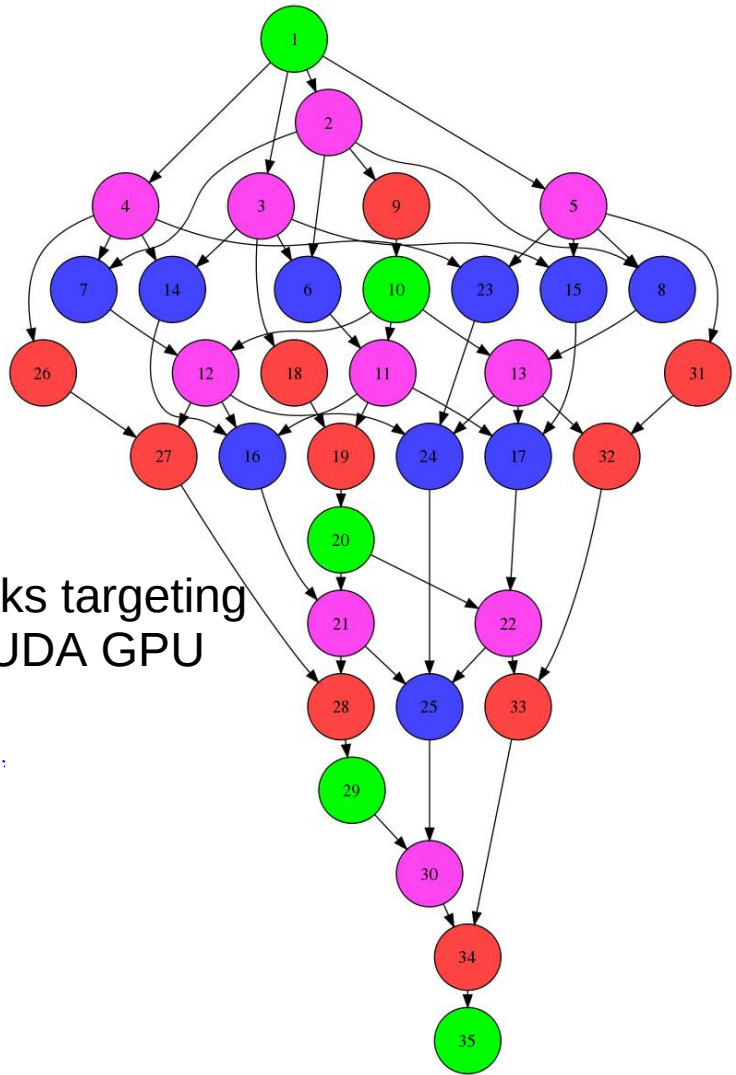
```
for (int j = 0; j < N; j+=BS) {
    for (int k= 0; k< j; k+=BS)
      for (int i = j+BS; i < N; i+=BS)
        sgemm_tile(BS, N, &A[k][i], &A[k][j], &A[j][i]); // A[i,j] = A[i,j] - A[i,k] * (A[j,k])
    for (int i = 0; i < j; i+=BS)
      ssyrk_tile(BS, N, &A[i][j], &A[j][j]); // A[j,j] = A[j,j] - A[j,i] * (A[j,i])
    spotrf_tile(BS, N, &A[j][j]); // Cholesky Factorization of A[j,j]
    for (int i = j+BS; i < N; i+=BS)
      strsm_tile(BS, N, &A[j][j], &A[j][i]);
}
```

StarSs Cholesky

- Specific kernels for different target arquitectures/devices

```
#pragma omp target device (cuda) copy_deps
#pragma omp task inout([NB*NB] A)
void spotrf_tile_cuda (float *A,long NB);

#pragma omp target device (cuda) copy_deps
#pragma omp task input([NB*NB] A, [NB*NB] B) inout([NB*NB] C)
void sgemm_tile_cuda (float  *A, float *B, float *C, unsigned long NB);

#pragma omp target device (cuda) copy_deps
#pragma omp task input([NB*NB] T) inout([NB*NB] B)
void strsm_tile_cuda (float *T, float *B, unsigned long NB)

#pragma omp target device (cuda) copy_deps
#pragma omp task input([NB*NB] A) inout([NB*NB] C)
void syrk_tile_cuda( float *A, float *C, long NB)
```

All tasks targeting a CUDA GPU




```
for (int j = 0; j < N; j+=BS) {
    for (int k= 0; k< j; k+=BS)
      for (int i = j+BS; i < N; i+=BS)
        sgemm_tile(BS, N, &A[k][i], &A[k][j], &A[j][i]); // A[i,j] = A[i,j] - A[i,k]
    for (int i = 0; i < j; i+=BS)
      ssyrk_tile(BS, N, &A[i][j], &A[j][j]); // A[j,j] = A[j,j] - A[j,i] * (A[j,i])
    spotrf_tile(BS, N, &A[j][j]); // Cholesky Factorization of A[j,j]
    for (int i = j+BS; i < N; i+=BS)
      strsm_tile(BS, N, &A[j][j], &A[j][i]);
}
```

OmpSs: Integration into OpenMP

- Kernels will need to access data structures
  - input/output hints used to copy data to local memories
  - copy_in/copy_out hints

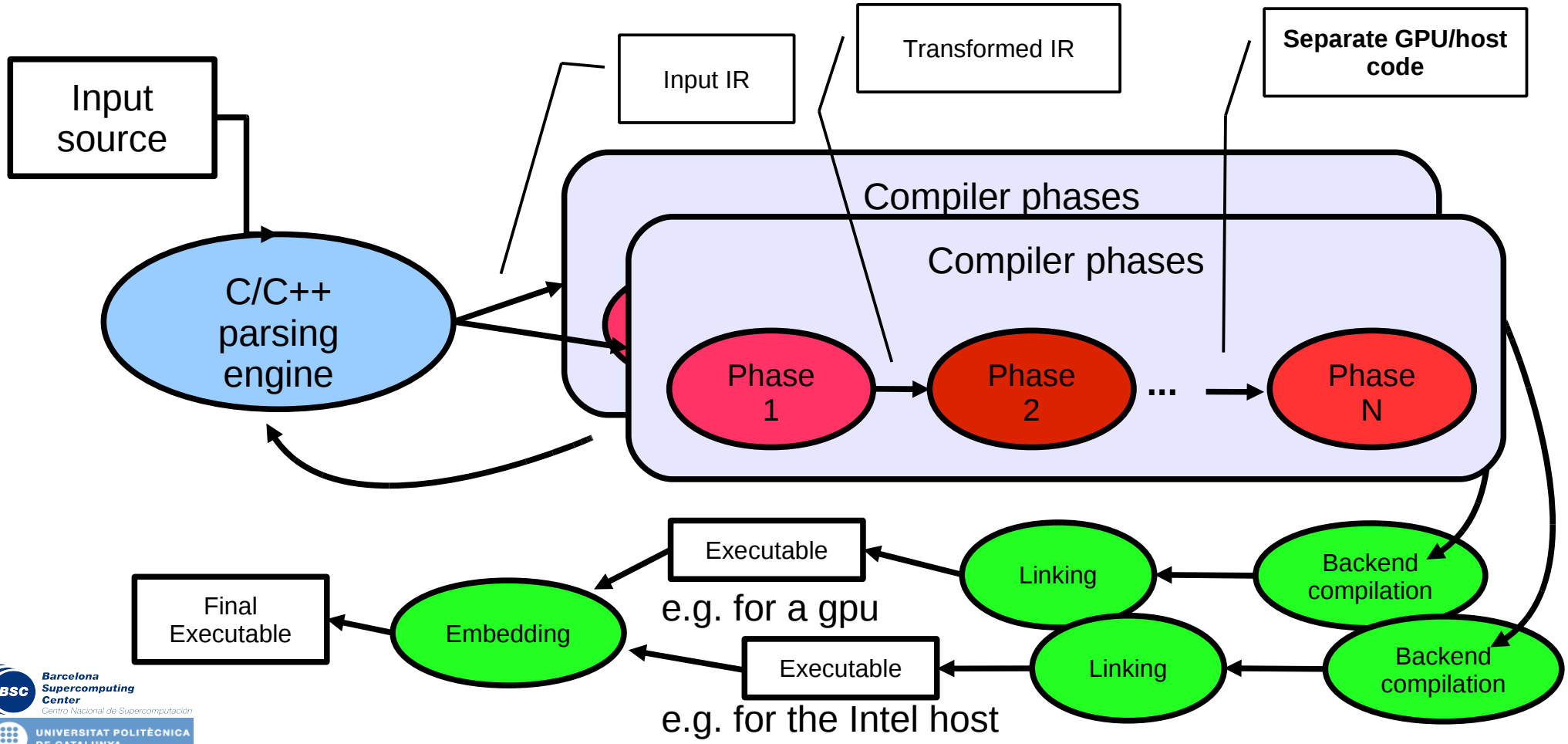
```
for (i=0; i < N; i++) {
  for (j=0; j < M; j++) {
#pragma omp target device(cuda)  copy_deps
#pragma omp task input([BSx+1][BSy]   data_in) output ([BSx][BSy]   data_out)
    {
        ...
        data_out[ii][jj] = w0*data_in[ii][jj] + ... data_in[ii+1][j]...
    }
  }
}
```

  - Similar to PGI approach

```
#pragma acc region local(b[1:n-2][1:m-2]) copy(a[0:n-1][0:m-1])
 {
  for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
      b[i][j] = w0*a[i][j] + w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);
  for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
      a[i][j] = b[i][j];
 }
```

OmpSs compilation environment

- Automatic offloading of code and data onto the GPUs



Code generation

- Code generated by Mercurium
  - Encapsulates arguments in a structure
  - Outlines GPU code into a function when needed
  - Passes the correct parameters to the runtime to invoke the function
    - **How many times we have lost hou of work because of**
      - **Incorrect parameter c**
      - **Incorrect param**
      - **Incorrect ty ions?**
      - **Incorr ions of runtime services?**
      - **I or checking?**
      - **ect...**

We need Productivity!!!

Code generation for the GPU

```
typedef struct _nx_data_env_0_t_tag
{
        float * * a_0;
        float * * h_0;
        float * * E_0;
        int __tmp_0_0;
        int __tmp_1_0;
        float __tmp_2_0;
       ....
} _nx_data_env_0_t;

void _gpu__ol_structfac_gpuss_0(_nx_data_env_0_t * _args)
{
    nanos_err_t cp_err;
    float * _cp___tmp_4;
    cp_err = nanos_get_addr(0, (void **) &_cp___tmp_4);
    if (cp_err != NANOS_OK)
        nanos_handle_error(cp_err);
    float * _cp___tmp_6;
    cp_err = nanos_get_addr(1, (void **) &_cp___tmp_6);
    if (cp_err != NANOS_OK)
        nanos_handle_error(cp_err);
    float * _cp___tmp_8;
    cp_err = nanos_get_addr(2, (void **) &_cp___tmp_8);
    if (cp_err != NANOS_OK)
        nanos_handle_error(cp_err);
    {
        structfac_kernel(_args->__tmp_0_0, _args->__tmp_1_0, _args->__tmp_2_0, _
args->__tmp_3_0, _cp___tmp_4, _args->__tmp_5_0, _cp___tmp_6, _args->__tmp_7_0, _
cp___tmp_8);
    }
}
```

```
void structfac_kernel(int na, int nr, float f2,
                      int NA, float * a, int NH,
                      float * h, int NE, float * E)
{
  dim3 threads(128,1,1);
  int blocks = nr/threads.x;
  if (blocks*threads.x < nr)
    blocks++;
  dim3 grid(blocks,1,1);
  int sharedsize = 16384-2048;
  int maxatoms = sharedsize/sizeof(TYPE_A);



  cstructfac<<<grid, threads, sharedsize>>>
          (na,nr,maxatoms,f2, (TYPE_A *) a,
          (TYPE_H *) h, (TYPE_E *) E);
  cudaError_t err = cudaGetLastError();
  if (err!=cudaSuccess) {
    fprintf (stderr,"error %d: %s\n", err,
          cudaGetErrorString(err));
    exit(1);
  }
}
```

Outline

- Motivation
- Proposal: OmpSs
- Examples
  - Matrix Multiply
  - BlackScholes
  - Perlin noise
  - Krist
  - Julia Set
- Hands-on

OmpSs Matrix Multiply

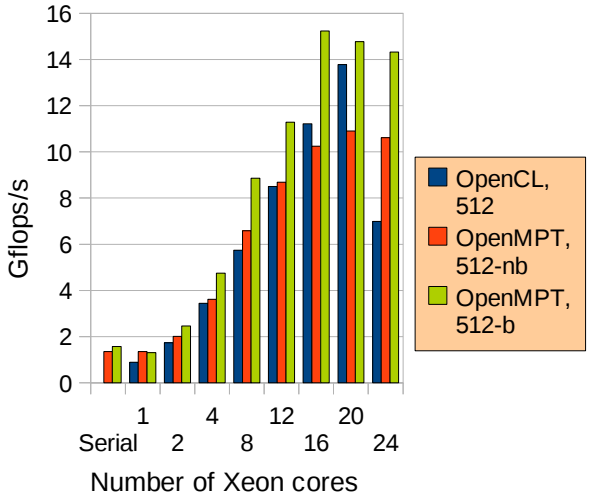
- Coding

```
#pragma omp target device (cuda) copy_deps
#pragma omp task inout( [NB*NB] C ) input([NB*NB] A, [NB*NB] B)
void matmul_tile (float *A, float *B, float *C)
{
        int hA, wA, wB;
        hA = NB;
        wA = NB;
        wB = NB;

        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
        dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

        Muld <<<dimGrid, dimBlock>>> ( A, B, wA, wB, C );
        cudaError_t err = cudaGetLastError();
        if (err != cudaSuccess) {
                printf("ERROR: %s\n", cudaGetErrorString(err));
                exit(1);
        }
}
```

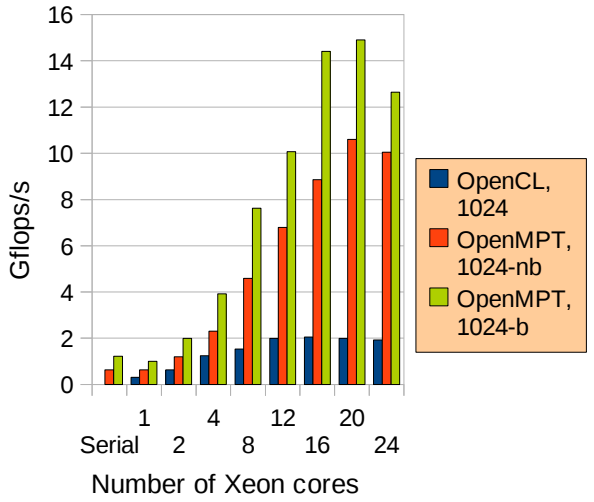
```
__global__ void Muld(float* A,
                     float* B,
                     int wA, int wB,
                     float* C) {
    // the Nvidia CUDA SDK kernel
}
```

OmpSs Matrix Multiply

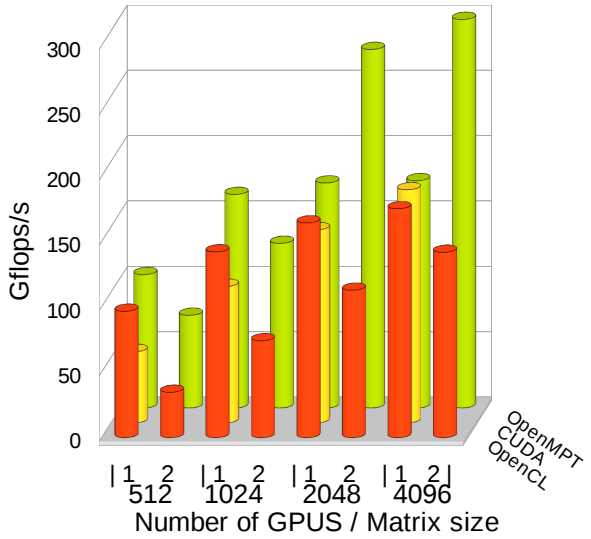
- Matrix multiply evaluation



Intel Xeon server



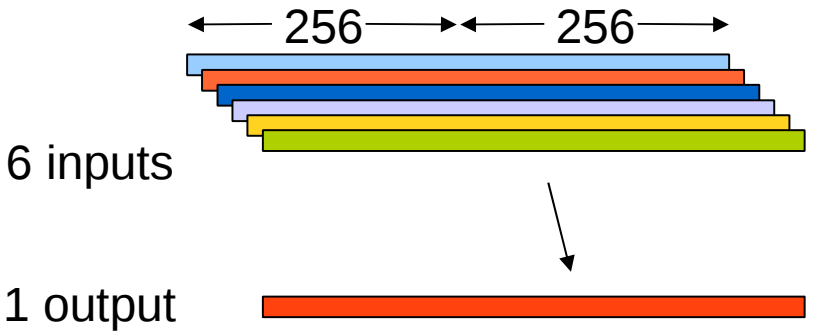
Nvidia GPU GTX 285



OmpSs BlackScholes

- Use of input/output/inout

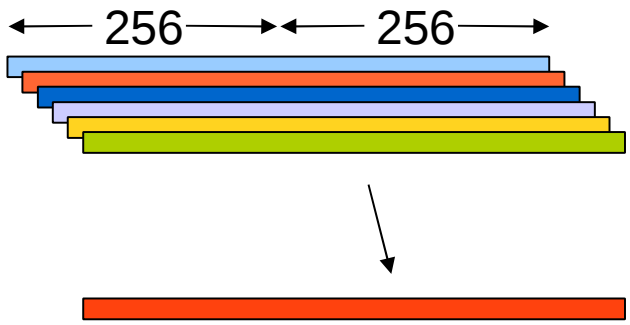
```
#pragma omp target device(cuda) copy_deps
#pragma omp task input ( \
                [global_work_group_size]   cpflag_fptr, \
                [global_work_group_size]   S0_fptr, \
                [global_work_group_size]   K_fptr, \
                [global_work_group_size]   r_fptr, \
         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



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];
```

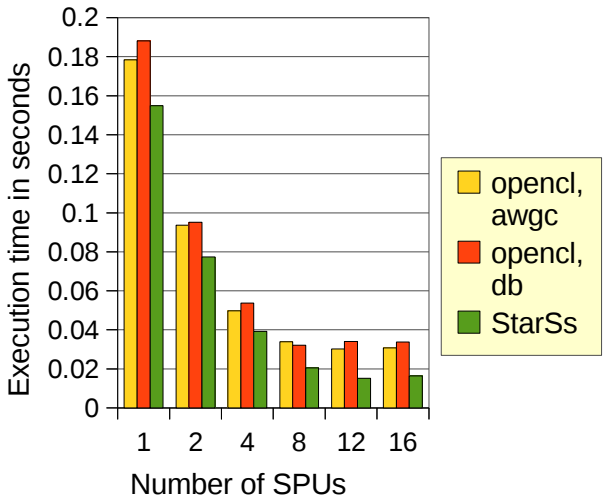



```
#pragma omp target device(cuda) copy_in( \
                                [global_work_group_size]   cpflag_f, \
                                [global_work_group_size]   S0_f, \
                                [global_work_group_size]   K_f, \
                                [global_work_group_size]   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
 }
```

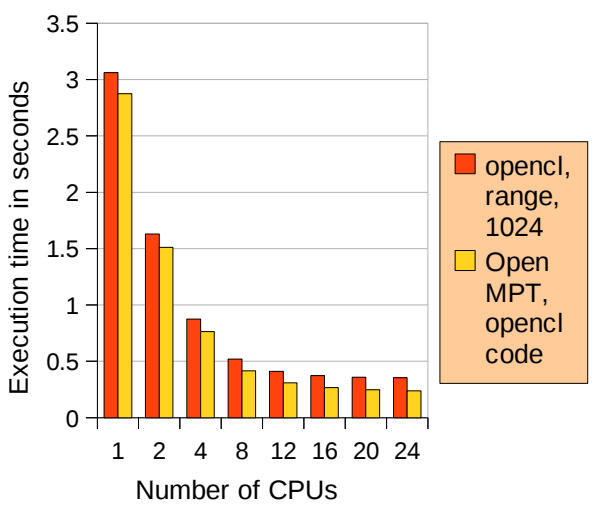
OmpSs BlackScholes

BlackScholes evaluation

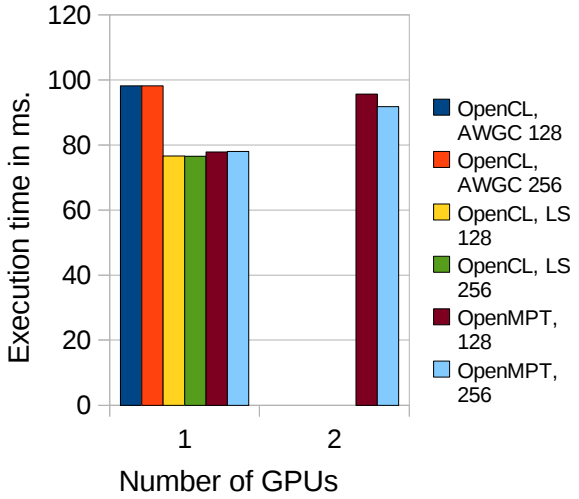
Cell processor



Intel Xeon server

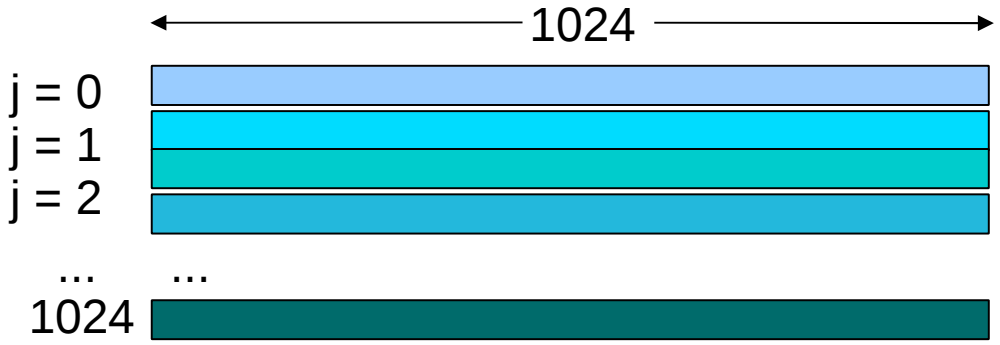


Nvidia GPU GTX 285



OmpSs Perlin Noise

- Perlin noise coding



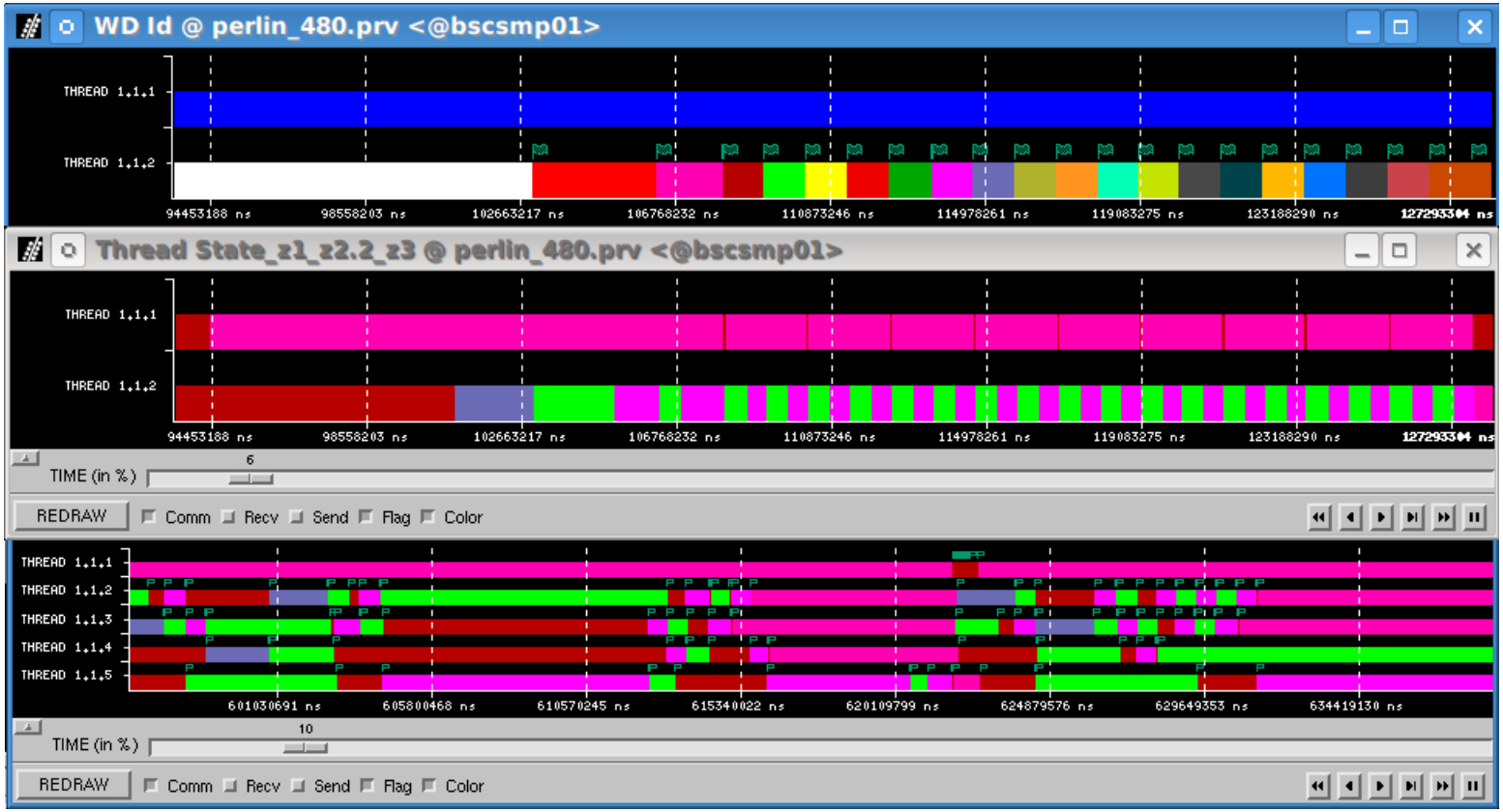

```
  for (j = 0; j < img_height; j+=BS) {
     pixel * optr = &output[j*rowstride];
#pragma omp target device(cuda)  copy_out( [BS*rowstride]  optr)
#pragma omp task shared(optr) firstprivate(tim, j, rowstride, img_width, BSx, BSy)
    {
        // kernel
    }
  }
#pragma omp taskwait
```

OmpSs Perlin Noise

- Internals

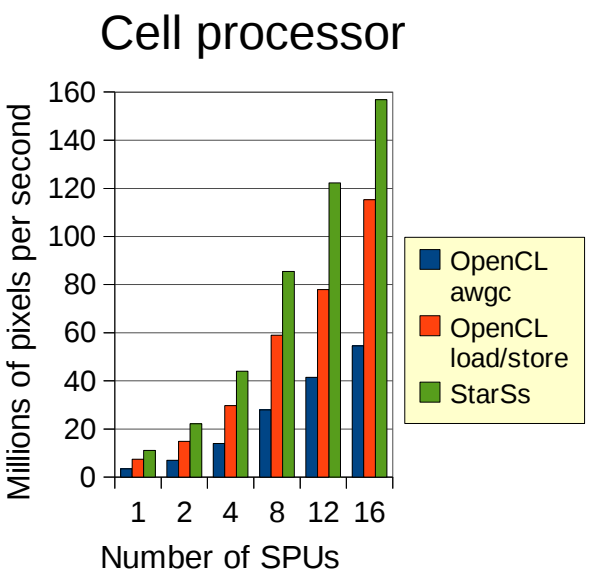
GTX 480

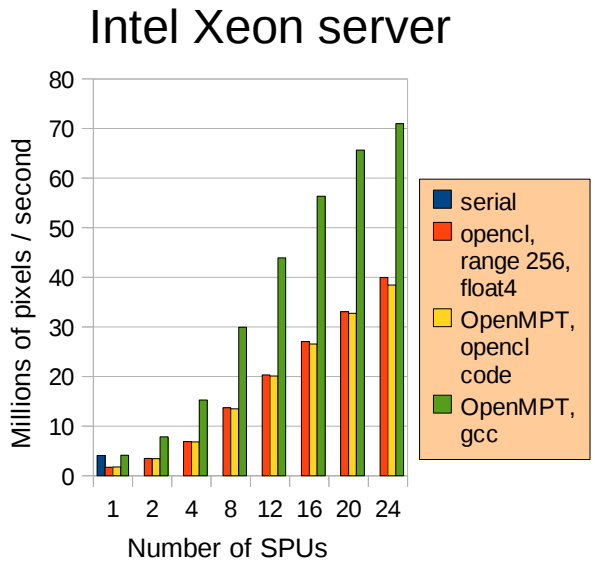
4 x GT200-T

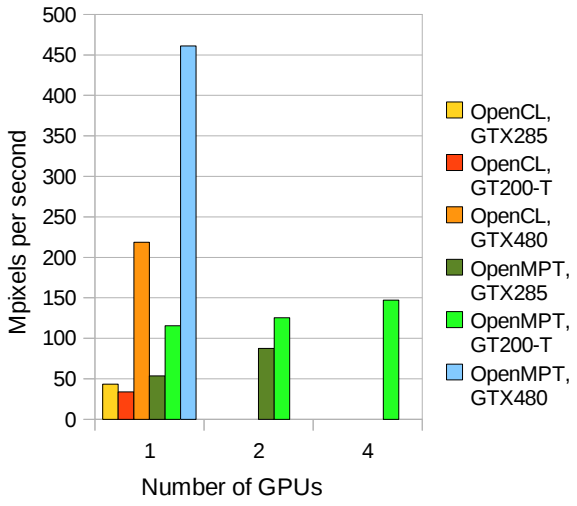


OmpSs Perlin Noise

- Perlin noise evaluation





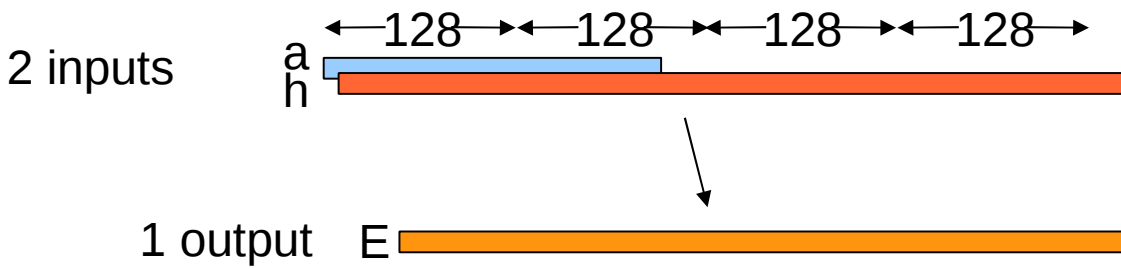


OmpSs Krist

- Coding

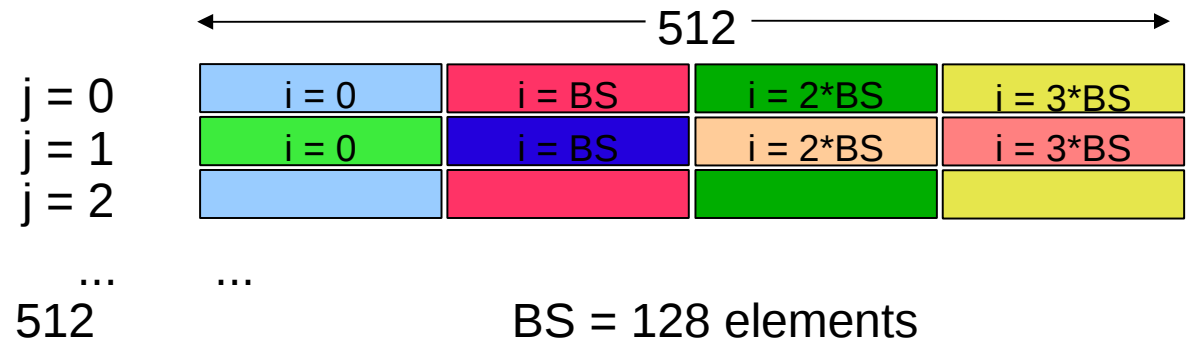
```
#pragma omp target device (cuda) copy_deps
#pragma omp task input ([NA] a, [NH] h) output ([NE] E)
void structfac_kernel(int na, int nr, float f2,
                      int NA, float*a, int NH, float*h, int NE, float*E);
```

```
void structfac_kernel(int na, int nr, float f2,int NA, float*a, int NH, float*h, int NE, float*E)
{
#pragma mcc verbatim start
   dim3 threads(128,1,1);
   int blocks = nr/threads.x;
   if (blocks*threads.x < nr)
      blocks++;
   dim3 grid(blocks,1,1);
   int sharedsize = 16384-2048;
   int maxatoms = sharedsize/sizeof(TYPE_A);

   cstructfac<<<grid, threads, sharedsize>>> (na,nr,maxatoms,f2,
                                              (TYPE_A *) a, (TYPE_H *) h, (TYPE_E *) E);
#pragma mcc verbatim end
}
```



OmpSs Julia Set

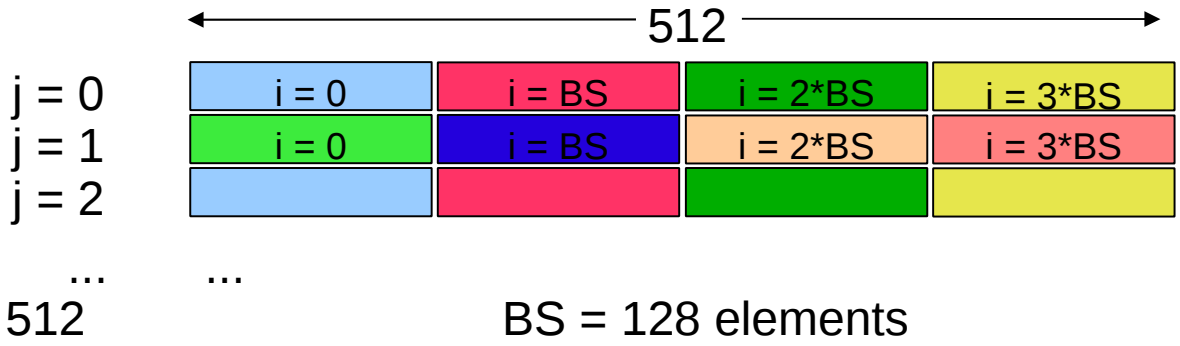
- Coding




```
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

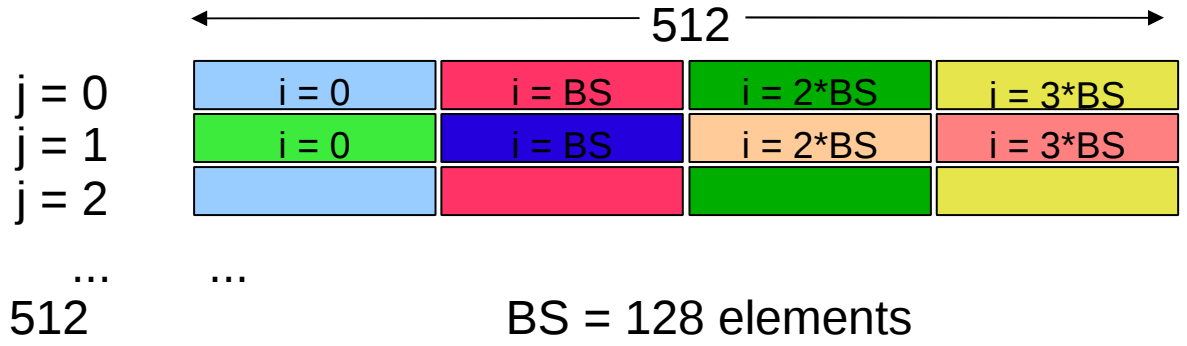
- Julia Set CUDA kernel task




```
#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 - adapted from OpenCL

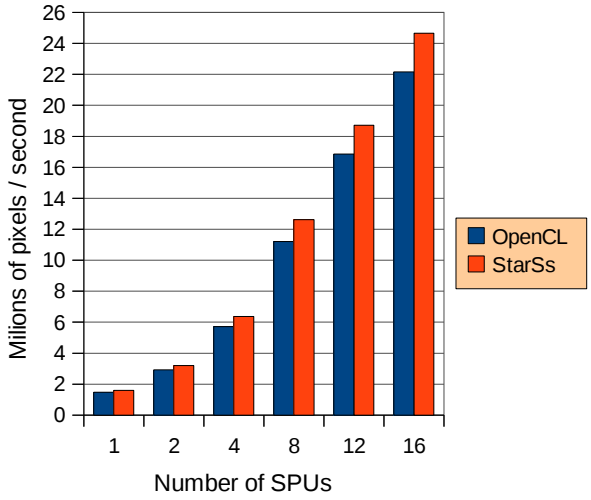



```
__global__ void compute_julia_kernel (const float4 muP,
                                      uchar16 * framebuffer,
                                      int rowstride,
                                      const struct julia_context jc,
                                      int offset)
{ ...
  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;
}
```

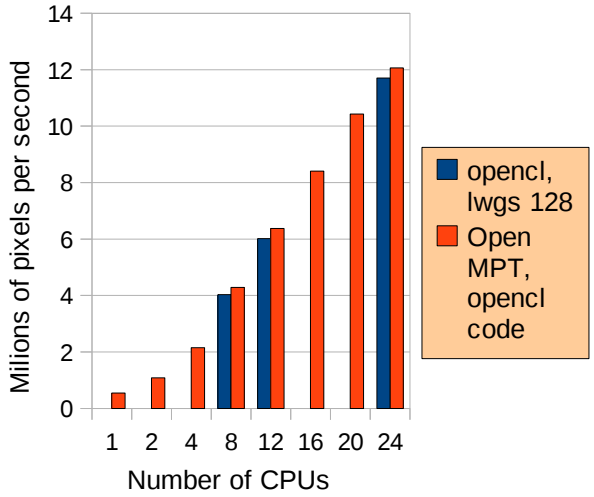
OmpSs Julia Set

- Evaluation

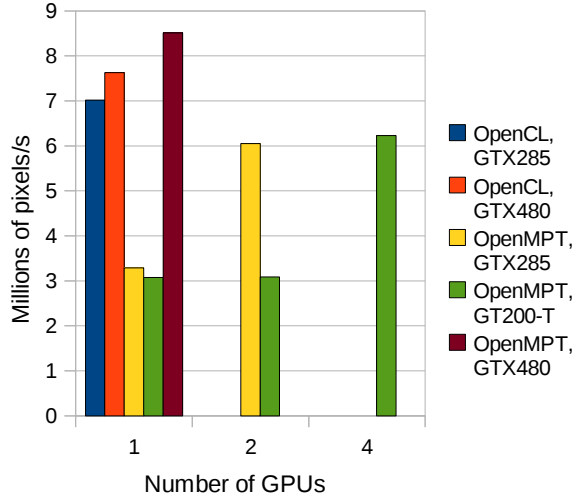
Cell processor



Intel Xeon server



Nvidia GPU GTX 285/480



Hands-on

- Log-on to your machine (alumne/sistemes)
- OmpSs/CUDA/OpenCL are already installed
- 'cd' into the 'apps' directory
- Each application directory has a Makefile
  - Compile using “make”
    - Use **mcc --ompss ...** # inside Makefiles
  - Execute using the “exec.sh” script
    - **NX_PES=P** to specify number of CPUS
    - **NX_GPUS=N** to specify the number of GPUS

Want to use OmpSs? please come to nanos.ac.upc.edu