www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs Support for Heterogeneous Platforms (CUDA)

Rosa M. Badia, Xavier Teruel and Xavier Martorell

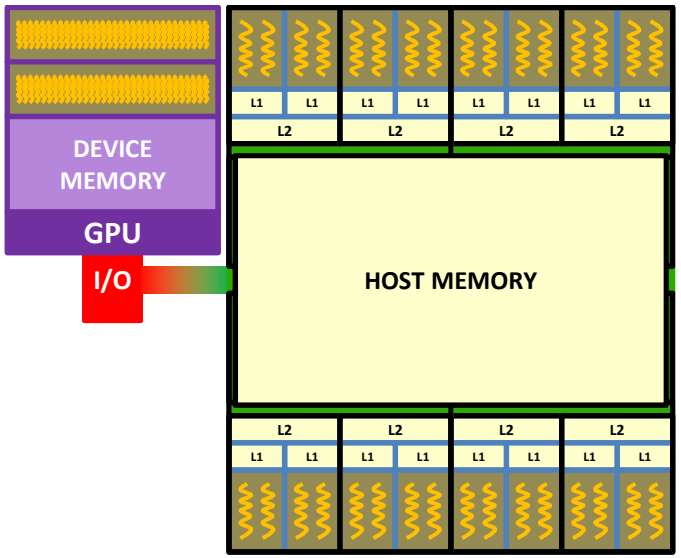
PACT2018 tutorial
Limassol, Nov 4th, 2018

Motivation (CUDA complexity)

Manual work scheduling

- Synchronize device's kernel execution
  - » Using explicit CUDA service's calls
  - » Using CUDA streams (and events)

```
void foo(void * args) {
     ...
     cudaDeviceSynchronize(); // OmpSs ~ taskwait
}
```



Memory allocation and copy to/from device

- Need to have a double memory allocation
- Different data sizes (due to blocking) makes the code confusing
- Explicit data copy operations → increase options for data overwrite

```
void foo(void * args) {
     h = (float*) malloc(sizeof(*h)*DIM2_H*nr);
     r = cudaMalloc((void**)&devh, sizeof(*h)*nr*DIM2_H);
     ...
     cudaMemcpy(devh,h,sizeof(*h)*nr*DIM2_H, cudaMemcpyHostToDevice);
     ...
}
```

Code comparison

```
#pragma omp target device(cuda) copy_deps ndrange(1,N,128)
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
int main(int argc, char *argv[])
{
  float a=5, *x, *y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));



  for (int i=0; i<N; ++i) x[i] = y[i] = (float) i;




  saxpy(N, a, x,  y);

  #pragma omp taskwait

  for (int i=0; i<N; ++i)
      if (y[i]!=a*i+i) printf("Error\n");
```

task declaration

synchronization

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
int main(int argc, char *argv[])
{
  float a=5, *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));
  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) x[i] = y[i] = (float) i;

  cudaMemcpy(d_x,x,N*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_y,y,N*sizeof(float),cudaMemcpyHostToDevice);

  saxpy<<<(N+127)/128, 128>>>(N, a, d_x, d_y);

  cudaMemcpy(y,d_y,N*sizeof(float),cudaMemcpyDeviceToHost);

  for (int i=0; i<N; ++i)
      if (y[i]!=a*i+i) printf("Error\n");

  cudaMemFree(d_x);
  cudaMemFree(d_y);
```

declaration

dev. allocation

copy to device

invokation

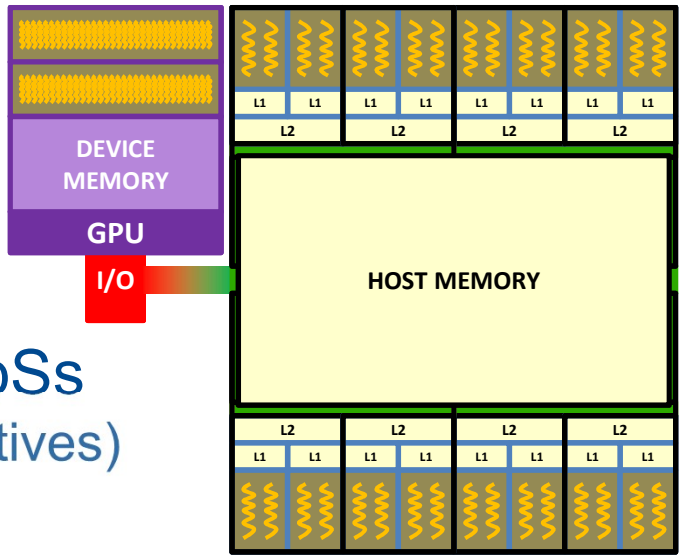
copy from device

dev. mem free

BSC accelerator's proposal

Standalone CUDA programming

- Manual work scheduling (kernel offloading)
- Memory allocation and copy to/from device
- Complex code and data management



BSC's proposal: combine CUDA and OmpSs

- OmpSs expressiveness (through compiler directives)
  - » Task based programming model
  - » Data directionality information (in/out/inout)
    - *Dependence detection at runtime (according with the provided information)*
    - *Automatic data movement among different Memory Address Spaces (device and host)*
- CUDA performance
  - » Leveraging existing CUDA kernels (including CUBLAS)
  - » CUDA kernels → OmpSs tasks

Outline: Heterogeneous Support (CUDA)

Introduction to heterogeneous support

- Execution model (device thread)
- Memory model (device memory)
- Task workflow using CUDA devices

Language support

- Target directive: device, copy, ndrange, shmem, implements
- Memory consistency description
- Porting standalone CUDA to OmpSs + CUDA

Compile and execute

- Mercurium compiler options
- Nanos++ runtime options

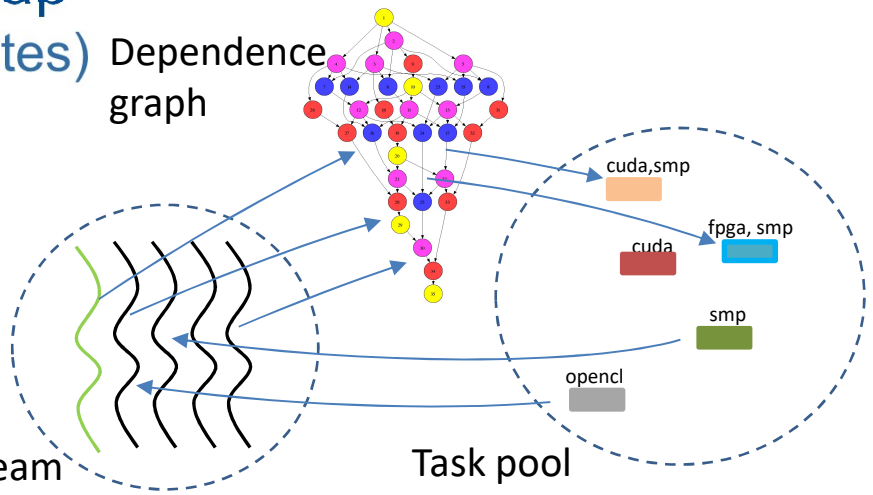
Hands-on session: exercises


www.bsc.es
BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
Introduction

Execution Model (OmpSs + CUDA)

Global thread team created on startup

- One worker starts main task (also executes)
- N-1 workers execute tasks
- One representative per device (CUDA)



All get work from a task pool

- CUDA kernels become tasks
- Task is labeled with (at least) one target device
- Scheduler decides which task to execute
- Tasks may have several targets (versioning)

Memory Model (device memory hierarchy)

Device local memory (also register)

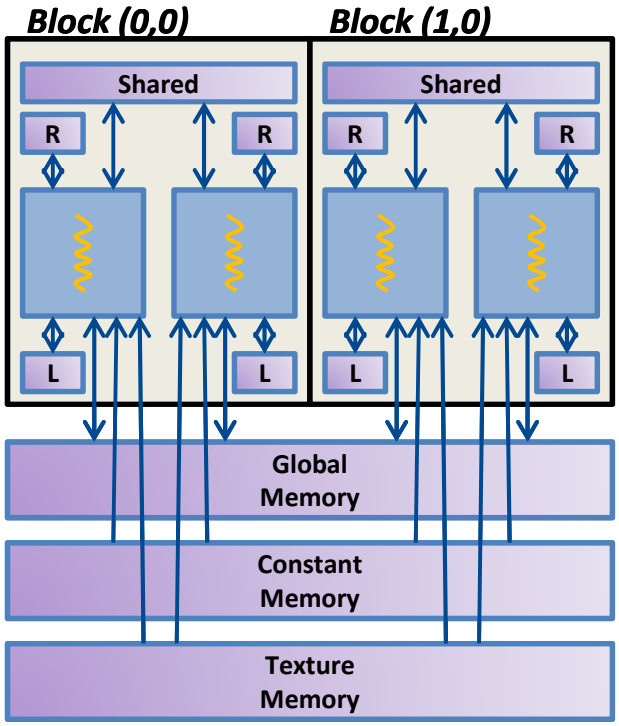
- Each thread has its own local storage
- Mostly registers (managed by the compiler)

Device shared memory

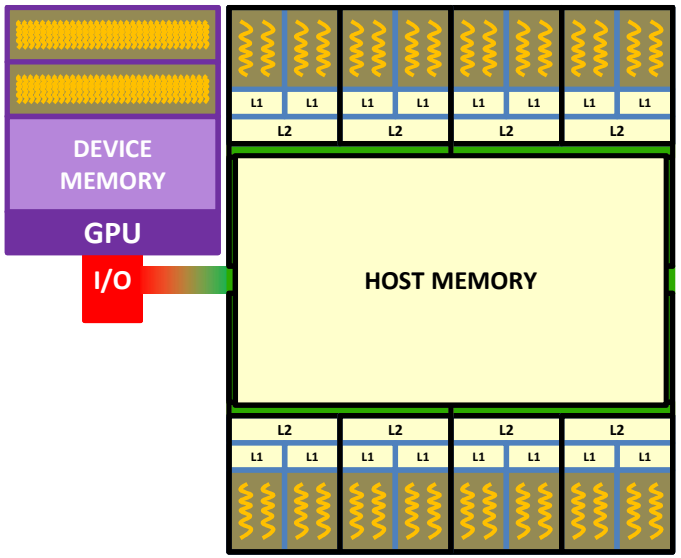
- Each thread block has its own shared memory
- Very low latency (a few cycles)
- Very high throughput: 38-44 GB/s per multiprocessor
  - » 30 multiprocessors per GPU → over 1.1-1.4 TB/s

Device global memory (+constant/texture)

- Accessible by all threads (as well as the host)
- Higher latency (400-800 cycles)
- Lower throughput
  - » 140 GB/s (1GB boards)
  - » 102 GB/s (4GB boards)



Memory Model (OmpSs + CUDA)



A global (logical) address space

Runtime handles device/host memories
- SMP machines → no 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

CUDA memory model (handled by OmpSs)
- Host memory → device global memory (automatically)
- Tasks may also allocate device shared memory (on demand)

Task Workflow (OmpSs + CUDA) [1/3]

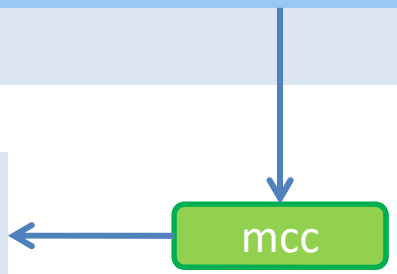
Compiler generates a stub function task that invokes the kernel

— Using the information at ndrange and shmem clauses

```
#pragma omp target device(cuda) copy_deps ndrange(1,N,128)
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy(int n, float a, float *x, float *y);
```

```
__global__ void saxpy(int n, float a, float *x, float *y){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

```
#include <kernel.h>
int main(int argc, char *argv[]) {
  ...
  saxpy(N, a, x,  y); // create a task with saxpy_ol
  ...
}
```

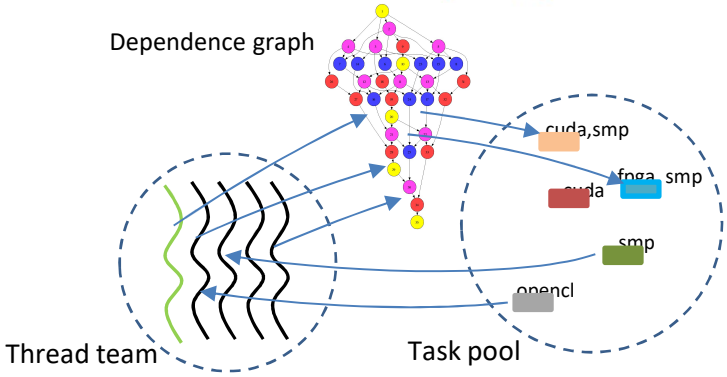


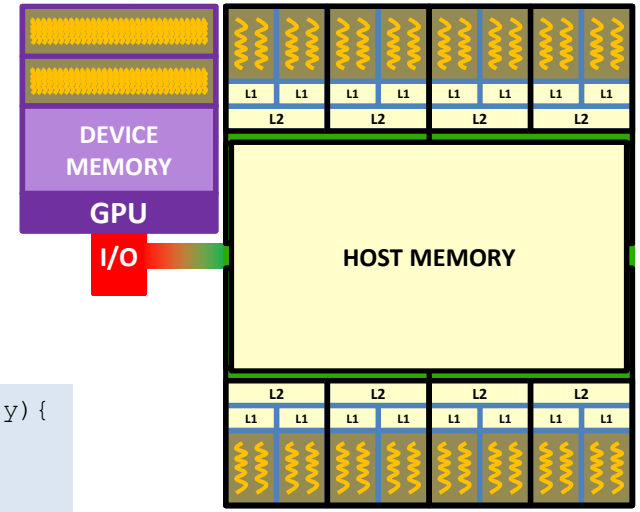

```
void saxpy_ol(int N, float a, float *x, float *y){
   ... // translates x -> d_x; y -> d_y
   saxpy<<<(N/128, 128>>>(N, a, d_x, d_y);
}
```

Task Workflow (OmpSs + CUDA) [2/3]

Helper thread: setup and execute phases

- On task setup/prefetch
  - » Determines allocation/copy requirements (invalidates if needed)
  - » Copy data to target device (corresponding stream and setting CUDA events)
- On task execution
  - » Determines execution stream and CUDA events
  - » Sets kernel arguments (device pointers)
  - » Invokes kernel (copy events → kernel → execution events)






```
void saxpy_ol(int N, float a, float *x, float *y){
   ... // translates x -> d_x; y -> d_y
   saxpy<<<(N/128, 128>>>(N, a, d_x, d_y);
}
```

Task Workflow (OmpSs + CUDA) [3/3]

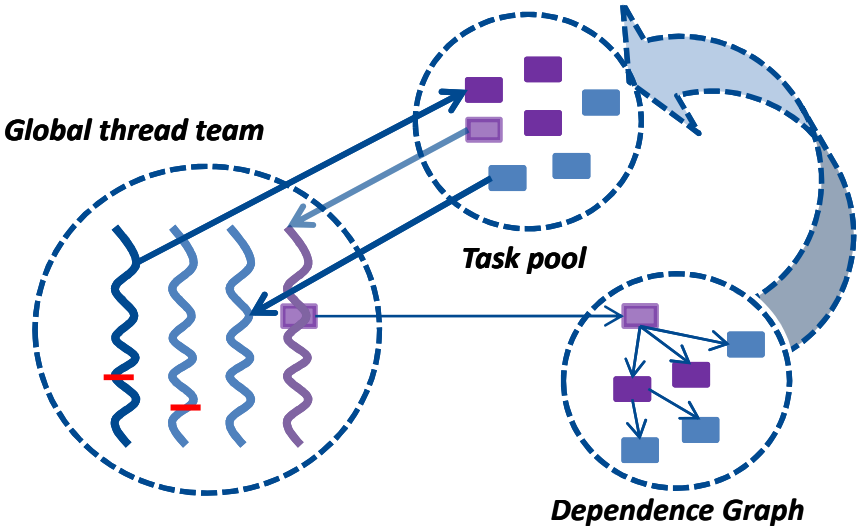
Helper thread: idle phase

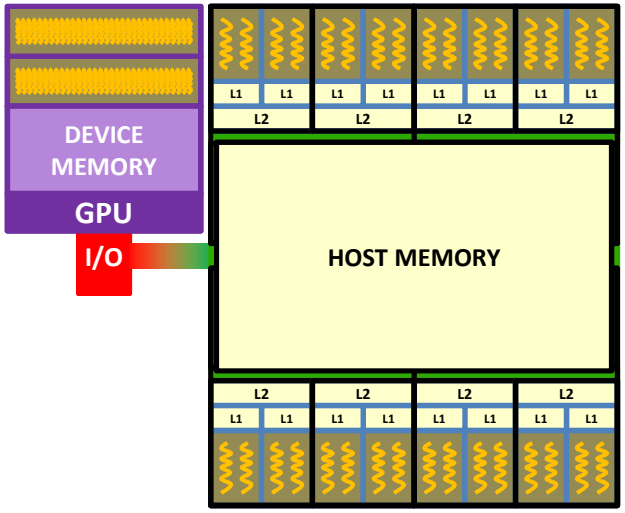
– On idle loop (spin)

» Check CUDA events → fulfil OmpSs task dependences

Any thread: on task synchronization (taskwait)

– Performs data transfers, deallocation and invalidation (when needed)





OmpSs GPU Terminology

OmpSs Device	CUDA Device	OpenCL Device
Local memory (thread)	Local memory	Private Memory
Shared memory (block)	Shared memory	Local memory
Global memory (device)	Global memory	Global memory
NDRange*	Grid	NDRange
Thread	Thread	Work item
Block	Block	Work group

*also **thread hierarchy** or **index space**


www.bsc.es
BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
Language Support

Target Directive

Device information: the target directive

– Always attached to the task directive (outlined functions)

```
#pragma omp target device (type) [clause[[,] clause]...]
{outlined-task-construct}
```

Where clause:

– device(type): specific device to run the following task (smp, cuda, …)
– ndrange(dimensions)
– shmem(size)
– implements(function-name)
– Copy Clauses
  » Explicit copies: copy_in(list), copy_out(list), copy_inout(list)
  » Dependences: copy_deps (default), no_copy_deps

Target Directive (C/C++)

Always attached to the task directive (outlined functions)

```
#pragma omp target device (type) [clause[[,] clause]...]
{outlined-task-construct}
```

Preferably at function prototypes (header)

```
#pragma omp target device(cuda) ndrange(1,nr,128)
#pragma omp task in([NA] a, [NH] h) out([NE] E)
void cstructfac(int na, int nr, int nc, float f2,
     int NA, TYPE_A *a, int NH, TYPE_H *h, int NE, TYPE_E *E );
```

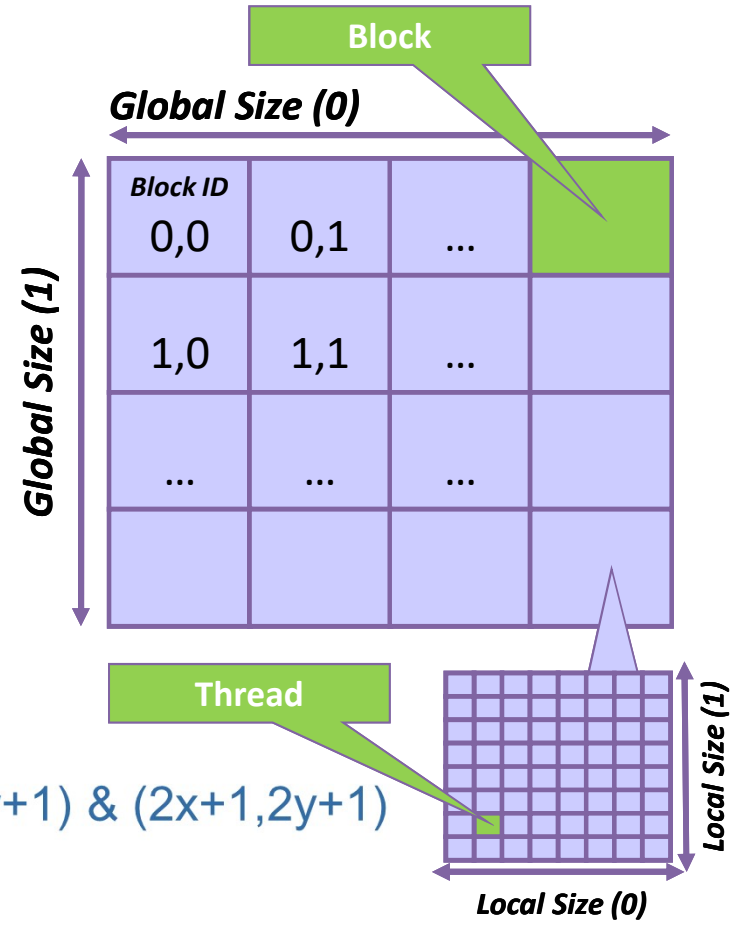
Task creation (invocation)

```
...
cstructfac(na, nr_2,maxatoms, f2, NA/DIM2_A, a,
           NH/DIM2_H/tasks, &h[DIM2_H*ii],
           NE/DIM2_E/tasks, &E[DIM2_E*ii]);
...
```

Target Directive: ndrange clause [1/2]

Thread hierarchy description (grid)

- Kernel execution / Data layout
  - » Same kernel / different data
  - » Instances executed in parallel (thread)
  - » Threads grouped into Blocks
    - *Synchronization within the block*
    - *Shared memory within the block*
    - *Maximum number of thread per block*
  - » Blocks grouped into a single Grid
- Thread / kernel instance identifier
  - » 2-level hierarchy → (block & threads)
  - » 1-, 2- or 3- dimensions per level
- Mapping thread hierarchy to data (e.g.)
  - » thread (x,y) computes (x,y)
  - » thread (x,y) computes (2x,2y),(2x+1,2y),(2x,2y+1) & (2x+1,2y+1)



Target Directive: ndrange clause [2/2]

```
#pragma omp target device(type) [ ndrange(…) ]
#pragma omp task [clause[[,] clause]...]
```

ndrange(n, global_array, local_array)

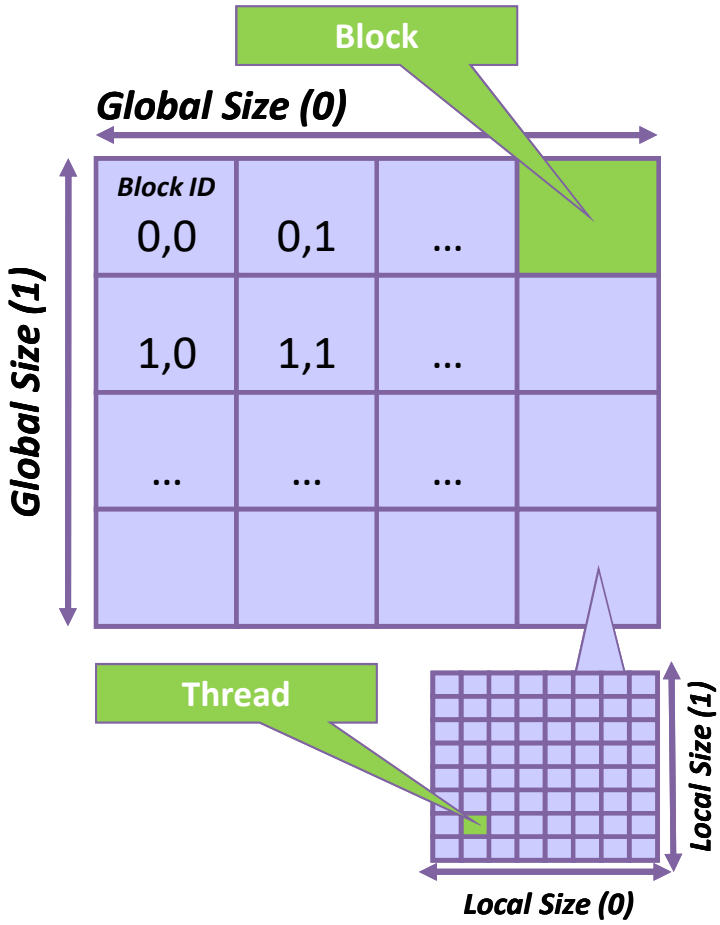
- » n → 1, 2 or 3: number of dimensions
- » global_array → size array of n elements
- » local_array → size array of n elements

ndrange(n, G1[,…Gn], L1[,…Ln])

- » n → 1, 2 or 3: number of dimensions
- » Gi → global size for dimension i (1 ≤ i ≤ n)
- » Li → local size for dimension i (1 ≤ i ≤ n)

… if not used, explicit call is needed

```
#pragma omp target device(cuda)
#pragma omp task [clauses]
void kernel_bridge ( arg_type1 arg1,…){
   dim3 dB(…), dT(…);
   kernel_code<<<dB,dT>>>( arg1,…);
}
```



Target Directive: shmem clause

The shmem clause

```
#pragma omp target device(type) [ shmem(…) ]
#pragma omp task [clause[[,] clause]...]
```

- shmem(size): allocate shared memory at kernel invocation
  - » Shared memory will be used by the kernel code
  - » The runtime system does not use this buffer for any optimization purpose

```
#pragma omp target device(cuda) shmem(1024)
#pragma omp task in(X[N]) out(Y[N])
__global__ void vector_op(int N, float* X, float *Y);
```

```
__global__ void vector_op(int N, float* X, float *Y){
   __shared__ float X_sh[];
   int bx = blockIdx.x;
   int tx = threadIdx.x;
   for (...)
      X_sh[i] = ... ;
   ...
}
```

```
vector_op_ol (int N, float a, float *x, float *y)
{
   ... // translates x -> d_x; y -> d_y
   saxpy<<<(N, BS, 1024>>>(N, a, d_x, d_y);
}
```

Target Directive: implements clause [1/2]

The implements clause

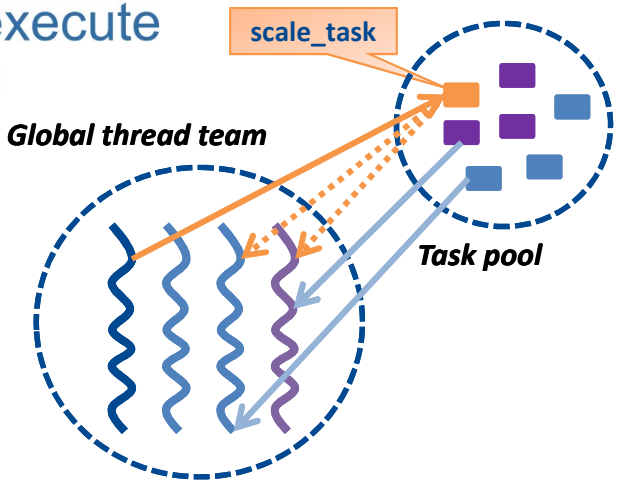
```
#pragma omp target device(type) [ implements(…) ]
#pragma omp task [clause[[,] clause]...]
```

- implements(function-name): annotate equivalent functions
  - » Can be skipped when outlined function == function-name
  - » Calls to “function-name” will become a single task creation/instantiation with as many implementation as the number of tasks annotated with implements(function-name)

Target Directive: implements clause [2/2]

Example: One single task → two different implementations

- Scheduler decides (at runtime) which version to execute
  - » On resource availability (first thread requesting work)
  - » Smart planification (shortest execution time)



scale.h

```
#pragma omp target device (smp) implements(scale_task)
#pragma omp task in([N] c) out([N] b)
void scale_task(double *b, double *c, double a, int N);
```

scale.c

```
void scale_task(double *b, double *c, double a, int N){
   for (int j=0; j < N; j++) b[j] = a*c[j];
}
```

scale.cuh

```
#pragma omp target device(cuda) implements(scale_task) ndrange(1,N,128)
#pragma omp task in([N] c) out([N] b)
__global__ void scale_task_cu(double *b, double *c, double a, int size);
```

scale.cu

```
__global__ void scale_task_cu(double *b, double *c, double a, int N) {
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   if (j < N) b[j] = a * c[j];
}
```

main.c

```
#include <scale.h>
#include <scale.cuh>

#define SIZE 100
int main (int argc, char *argv[])
{
   . . .
   scale_task(B,C,alpha,SIZE);
   . . .
}
```

Target Directive: copy clauses [1/3]

Explicit copy clauses

- copy_in(var-list): requests a consistent copy of variables before execution
- copy_out(var-list): after execution produces next “version” of variable
- copy_inout(var-list): combination of “in” and “out”

```
#pragma omp target device(cuda) copy_in([n]x) copy_inout([n]y)
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy(int n, float a, float* x, float* y);
```

Target Directive: copy clauses [2/3]

Copy data using tasks dependence clauses

- copy_deps (default behavior)
  - » in(var-list) → copy_in(var-list)
  - » out(var-list) → copy_out(var-list)
  - » inout(var-list) → copy_inout(var-list)
- no_copy_deps

99% of cases

```
#pragma omp target device(cuda) copy_in([n]x) copy_inout([n]y)
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy(int n, float a, float* x, float* y);
```

⇔

```
#pragma omp target device(cuda) copy_deps
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy(int n, float a, float* x, float* y);
```

Target Directive: copy clauses [3/3]

Other considerations

– All tasks (including **host**) must also specify copy clauses

```
#pragma omp target device(cuda) copy_deps
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy_cuda(int n, float a, float* x, float* y);
```

```
#pragma omp target device(smp) copy_deps
#pragma omp task in([n]x) inout([n]y)
void saxpy_smp(int n, float a, float* x, float* y);
```

– Taskwait ensures centralized data consistency

Memory consistency (getting consistent copies)

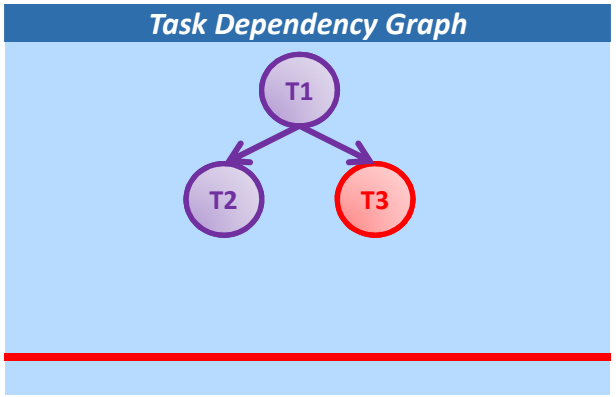
Relaxed-consistency "shared-memory" model (OpenMP-like)

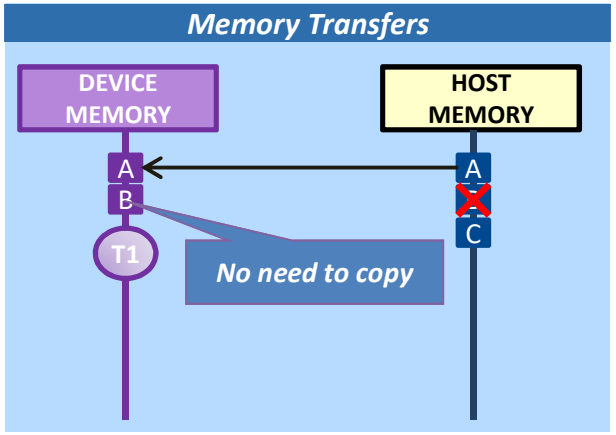
```
#pragma omp target device (cuda)
#pragma omp task out([N] b) in([N] c)
void scale_task_cuda(double *b, double *c, double a, int N)
{
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   if (j < N) b[j] = a * c[j];
}
#pragma omp target device (smp)
#pragma omp task out([N] b) in([N] c)
void scale_task_host(double *b, double *c, double a, int N)
{
   for (int j=0; j < N; j++) b[j] = a*c[j];
}
void main(int argc, char *argv[]) {
   ...
   scale_task_cuda (B, A, 10.0, 1024); //T1
   scale_task_cuda (A, B, 0.01, 1024); //T2
   scale_task_host (C, B, 2.00, 1024); //T3

   #pragma omp taskwait
   // can access any of A,B,C
   ...
```

T1 needs a valid copy of array A in the device

Also it allocates array B in the device (no copy needed), and invalidates other B's





Memory consistency (reusing data in place)

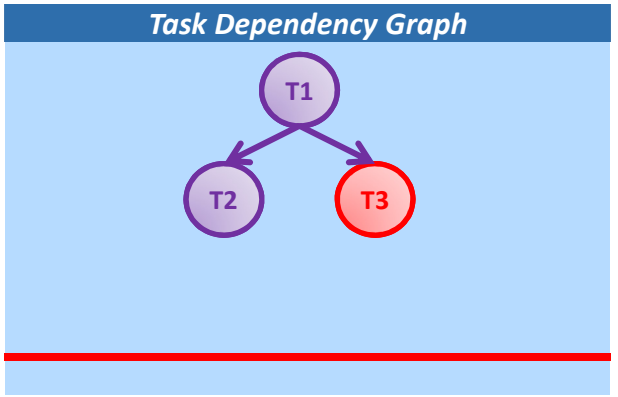
Relaxed-consistency "shared-memory" model (OpenMP-like)

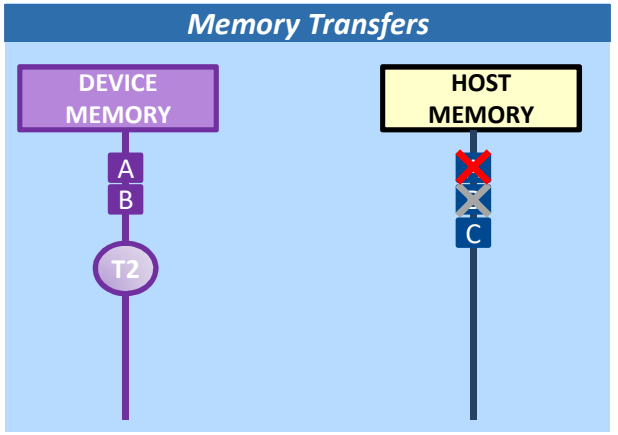
```
#pragma omp target device (cuda)
#pragma omp task out([N] b) in([N] c)
void scale_task_cuda(double *b, double *c, double a, int N)
{
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   if (j < N) b[j] = a * c[j];
}
#pragma omp target device (smp)
#pragma omp task out([N] b) in([N] c)
void scale_task_host(double *b, double *c, double a, int N)
{
   for (int j=0; j < N; j++) b[j] = a*c[j];
}
void main(int argc, char *argv[]) {
   ...
   scale_task_cuda (B, A, 10.0, 1024); //T1
   scale_task_cuda (A, B, 0.01, 1024); //T2
   scale_task_host (C, B, 2.00, 1024); //T3

   #pragma omp taskwait
   // can access any of A,B,C
   ...
```

T2 can reuse arrays A and B, due they have been used by previous task (T1)

Additionally it also invalidates others A's





Memory consistency (on demand copy back)

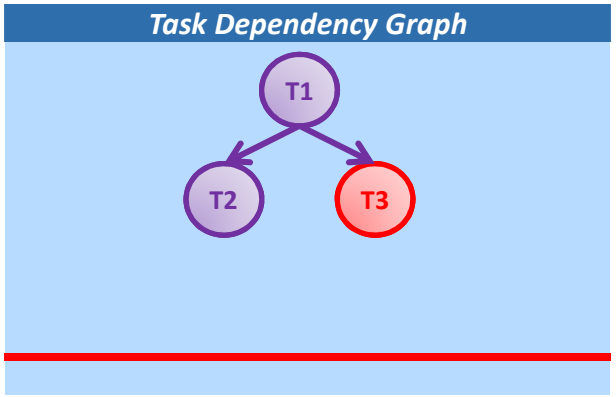
Relaxed-consistency "shared-memory" model (OpenMP-like)

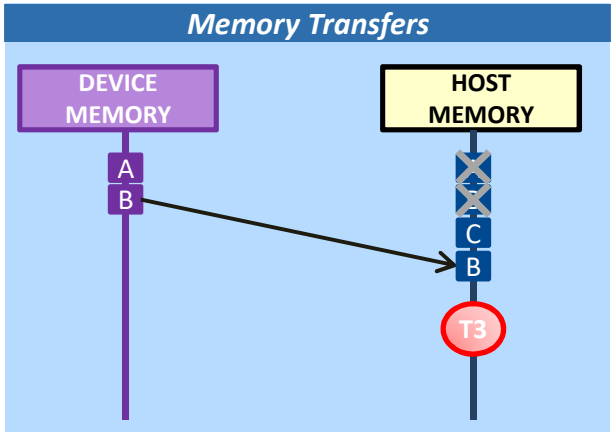
```
#pragma omp target device (cuda)
#pragma omp task out([N] b) in([N] c)
void scale_task_cuda(double *b, double *c, double a, int N)
{
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   if (j < N) b[j] = a * c[j];
}
#pragma omp target device (smp)
#pragma omp task out([N] b) in([N] c)
void scale_task_host(double *b, double *c, double a, int N)
{
   for (int j=0; j < N; j++) b[j] = a*c[j];
}
void main(int argc, char *argv[]) {
   ...
   scale_task_cuda (B, A, 10.0, 1024); //T1
   scale_task_cuda (A, B, 0.01, 1024); //T2
   scale_task_host (C, B, 2.00, 1024); //T3

   #pragma omp taskwait
   // can access any of A,B,C
   ...
```

T3 needs to copy back to the host array B

Does not invalidate the existent copy in device





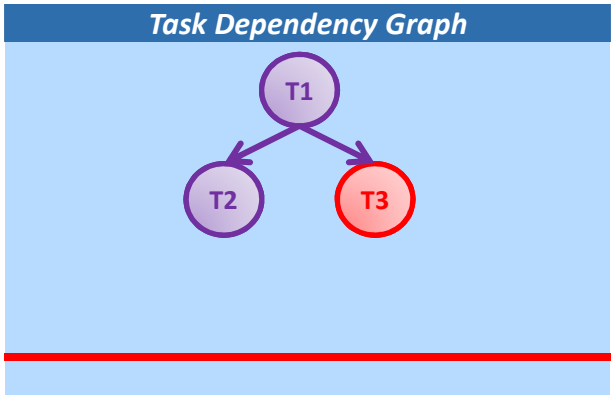
Memory consistency (centralized consistency)

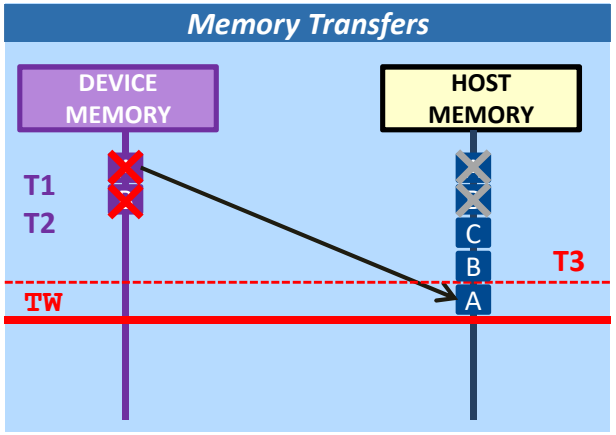
Relaxed-consistency "shared-memory" model (OpenMP-like)

```
#pragma omp target device (cuda)
#pragma omp task out([N] b) in([N] c)
void scale_task_cuda(double *b, double *c, double a, int N)
{
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   if (j < N) b[j] = a * c[j];
}
#pragma omp target device (smp)
#pragma omp task out([N] b) in([N] c)
void scale_task_host(double *b, double *c, double a, int N)
{
   for (int j=0; j < N; j++) b[j] = a*c[j];
}
void main(int argc, char *argv[]) {
   ...
   scale_task_cuda (B, A, 10.0, 1024); //T1
   scale_task_cuda (A, B, 0.01, 1024); //T2
   scale_task_host (C, B, 2.00, 1024); //T3

   #pragma omp taskwait
   // can access and modify any of A,B,C
   ...
```

Taskwait requires a full memory consistency in the host





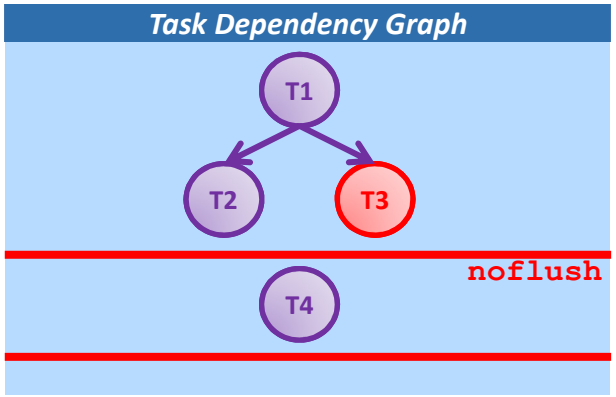
Memory consistency (avoid tw consistency)

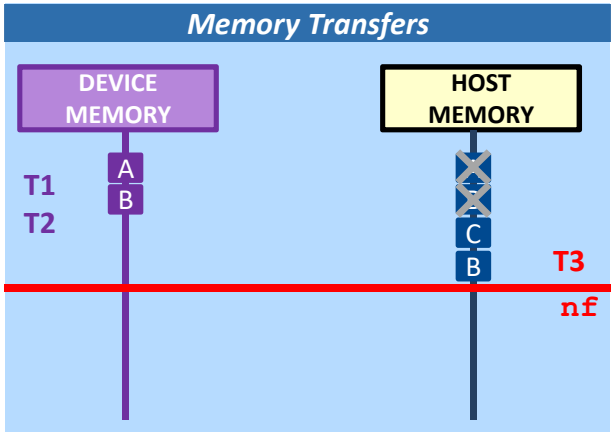
Relaxed-consistency “shared-memory” model (OpenMP-like)

```
#pragma omp target device (cuda)
#pragma omp task out([N] b) in([N] c)
void scale_task_cuda(double *b, double *c, double a, int N)
{
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   if (j < N) b[j] = a * c[j];
}
#pragma omp target device (smp)
#pragma omp task out([N] b) in([N] c)
void scale_task_host(double *b, double *c, double a, int N)
{
   for (int j=0; j < N; j++) b[j] = a*c[j];
}
void main(int argc, char *argv[]) {
   ...
   scale_task_cuda (B, A, 10.0, 1024); //T1
   scale_task_cuda (A, B, 0.01, 1024); //T2
   scale_task_host (C, B, 2.00, 1024); //T3
   #pragma omp taskwait noflush
   // does not flush data dev -> host

   scale_task_cuda (B, C, 3.00, 1024); //T4
   #pragma omp taskwait
   // can access any of A,B,C
   ...
```

Taskwait is waiting for task finalization, but does not copy memory back to the host (neither invalidate it)





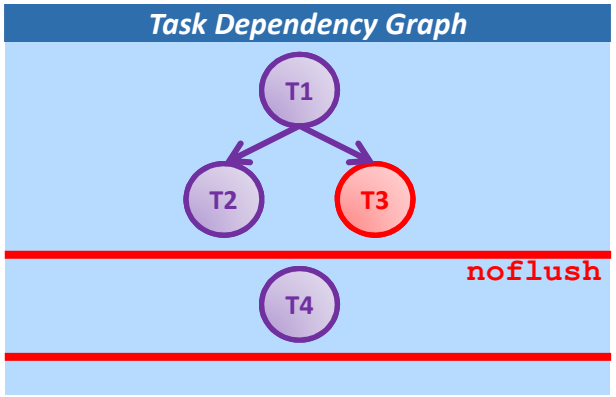
Memory consistency (multiple invalidations)

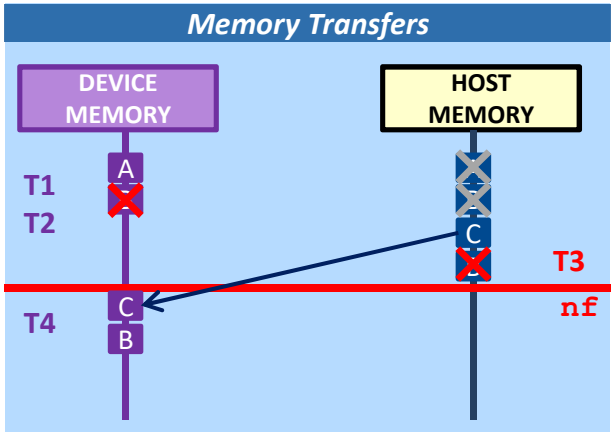
Relaxed-consistency "shared-memory" model (OpenMP-like)

```
#pragma omp target device (cuda)
#pragma omp task out([N] b) in([N] c)
void scale_task_cuda(double *b, double *c, double a, int N)
{
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   if (j < N) b[j] = a * c[j];
}
#pragma omp target device (smp)
#pragma omp task out([N] b) in([N] c)
void scale_task_host(double *b, double *c, double a, int N)
{
   for (int j=0; j < N; j++) b[j] = a*c[j];
}
void main(int argc, char *argv[]) {
   ...
   scale_task_cuda (B, A, 10.0, 1024); //T1
   scale_task_cuda (A, B, 0.01, 1024); //T2
   scale_task_host (C, B, 2.00, 1024); //T3
   #pragma omp taskwait noflush
   // does not flush data dev -> host

   scale_task_cuda (B, C, 3.00, 1024); //T4
   #pragma omp taskwait
   // can access any of A,B,C
   ...
```

Before execute T4 it will need a consistent copy of C and it will also invalidates all previous versions of B





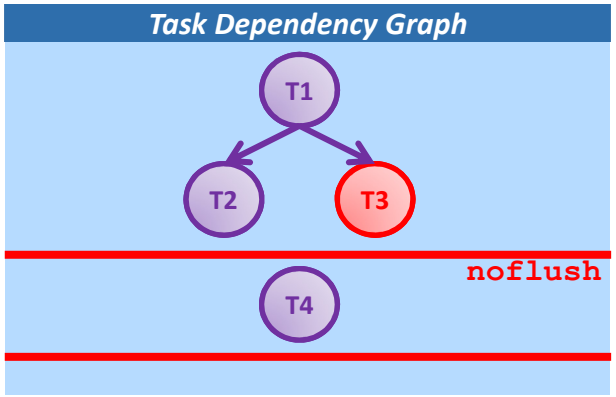
Memory consistency (centralized consistency)

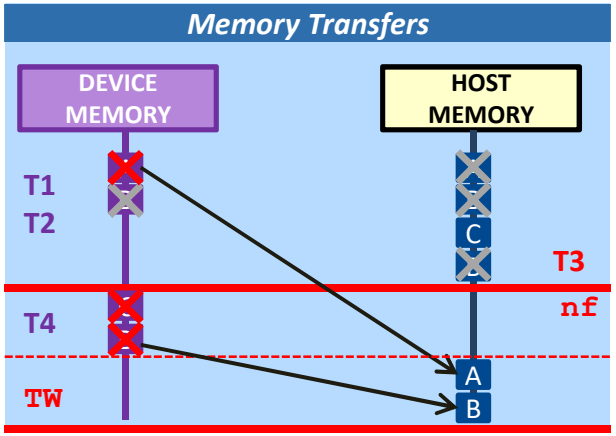
Relaxed-consistency “shared-memory” model (OpenMP-like)

```
#pragma omp target device (cuda)
#pragma omp task out([N] b) in([N] c)
void scale_task_cuda(double *b, double *c, double a, int N)
{
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   if (j < N) b[j] = a * c[j];
}
#pragma omp target device (smp)
#pragma omp task out([N] b) in([N] c)
void scale_task_host(double *b, double *c, double a, int N)
{
   for (int j=0; j < N; j++) b[j] = a*c[j];
}
void main(int argc, char *argv[]) {
   ...
   scale_task_cuda (B, A, 10.0, 1024); //T1
   scale_task_cuda (A, B, 0.01, 1024); //T2
   scale_task_host (C, B, 2.00, 1024); //T3
   #pragma omp taskwait noflush
   // does not flush data dev -> host

   scale_task_cuda (B, C, 3.00, 1024); //T4
   #pragma omp taskwait
   // can access any of A,B,C
   ...
```

Taskwait waits for tasks finalization, it will invalidates all data versions and force memory consistency





AXPY Algorithm (example of porting to CUDA)

AXPY computes alpha times a vector X plus a vector Y

$$Z = \propto X + Y$$

```
// Single precision axpy algorithm (Y = aX+Y)
void saxpy ( int n, float a, float *X, float *Y );
```

Where:

- n (scalar) number of elements in the vectors
- a (scalar) alpha factor (α)
- X (array of dimension n), vector to be scaled before summation (X)
- Y (array of dimension n), vector to be summed (Y), after the routine ends contains the result of the summation (Z)

AXPY Algorithm (OmpSs → OmpSs + CUDA)

1. Port kernel to CUDA
2. Annotate device (CUDA)
3. Complete devices (SMP)
4. Use these tasks (specific calls, implements,…)

main.c

```
#include <kernel.h>

int main(int argc, char *argv[])
{
   float a=5, x[N], y[N];

   // Initialize values
   for (int i=0; i<N; ++i)
      x[i] = y[i] = i;

   // Compute saxpy algorithm (1 task)
(?)saxpy(N, a, x,  y);
   #pragma omp taskwait

   //Check results
   for (int i=0; i<N; ++i){
      if (y[i]!=a*i+i) perror("Error\n")
   }

   message("Results are correct\n");
}
```

kernel.h (3)

```
#pragma omp target device(smp) copy_deps
#pragma omp task in([n]x) inout([n]y)
void saxpy(int n, float a, float* x, float* y);
```

kernel.c

```
void saxpy(int n, float a, float *X, float *Y)
{
   for (int i=0; i<n; ++i)
      Y[i] = X[i] * a + Y[i];
}
```

kernel.cuh (2)

```
#pragma omp target device(cuda) copy_deps ndrange(1,n,128)
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy_cu(int n, float a, float* x, float* y);
```

kernel.cu (1)

```
__global__ void saxpy_cu(int n, float a, float* x, float* y)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if(i < n) y[i] = a * x[i] + y[i];
}
```

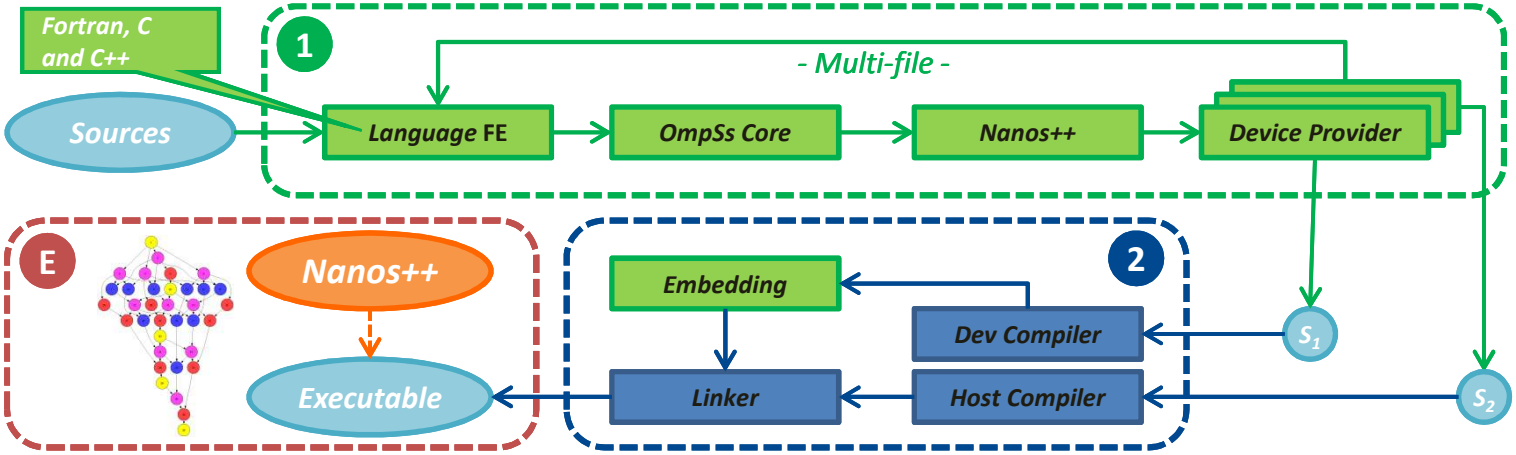

www.bsc.es
BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
Compile and Execute
OmpSs + CUDA

Compiler Support

(1) Source-to-source transformation
- Transforming directives to runtime calls (n-phases)
- It may generate additional files (e.g. kernel call files)

(2) Native compilation
- Kernel (.cu) compiled with the NVIDIA compiler
- Kernel call files compiled with the NVIDIA compiler
- Cleansed host code compiled with the host compiler
- All object files (including CUDA) embedded on final executable

Mercurium compiler
mcc → mnvcc
mcxx → mnvcxx
mfc → mnvfc



Compiler Support (CUDA)

kernel.h

```
#pragma omp target device(cuda) copy_deps ndrange(1,N,128)
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy(int n, float a, float *x, float *y);
```

kernel.cu

```
__global__ void saxpy(int n, float a, float *x, float *y) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

main.c

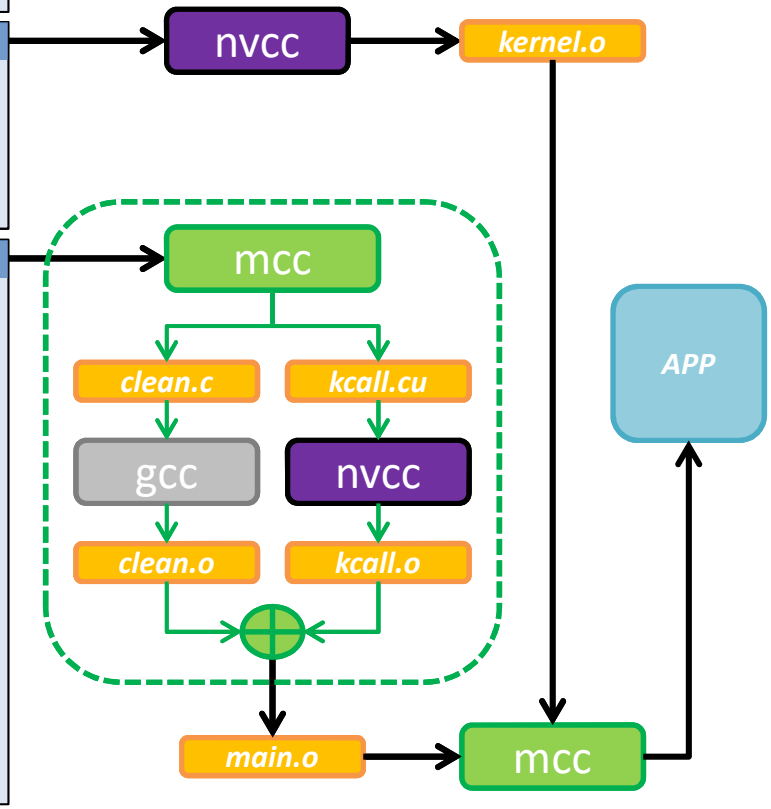
```
#include <kernel.h>
int main(int argc, char *argv[]) {
   float a=5, *x, *y;
   x = (float*)malloc(N*sizeof(float));
   y = (float*)malloc(N*sizeof(float));

   for (int i=0; i<N; ++i) x[i] = y[i] = (float) i;

   saxpy(N, a, x,  y);

   #pragma omp taskwait

   for (int i=0; i<N; ++i) if (y[i]!=a*i+i) printf("Error\n")
}
```



Compiler Support (getting additional help)

Getting Mercurium compiler version (useful reporting a bug)

```
$ mcc --version
mcxx 1.99.7 (git 48715a1 2015-02-23 08:52:32 +0100 developer version)
Configured with: /home/user/dev/mcxx/configure --prefix=/home/user/apps/ompss
--with-nanox=/home/user/apps/ompss PKG_CONFIG_PATH=/home/user/soft/lib/pkgconfig
--enable-ompss --with-cuda=/opt/cuda/5.0/
```

Getting Mercurium compiler options (explore additional flags)

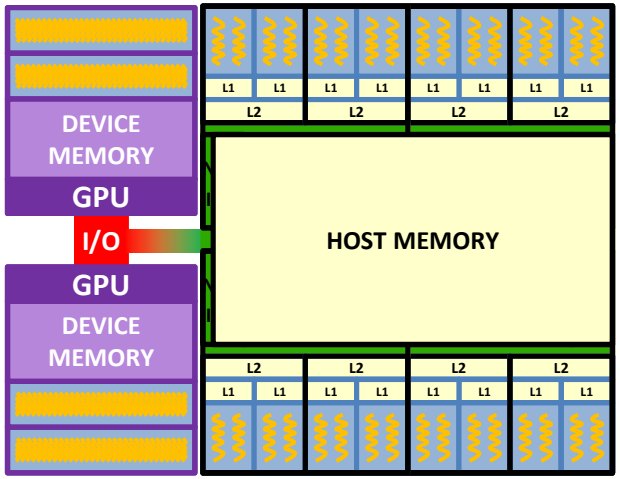
```
$ mcc --help

Usage: mcc options file [file..]
Options:
  -h, --help               Shows this help and quits
  --version                Shows version and quits
  --v, --verbose           Runs verbosely, displaying the programs
                           invoked by the compiler
  -o, --output=<file>      Sets <file> as the output file
  -c                       Does not link, just compile
  -E                       Does not compile, just preprocess
  -I <dir>                 Adds <dir> into the searched include
                           directories
  -L <dir>                 Adds <dir> into the searched library
                           directories
```

Runtime support (execution summary)

When executing a OmpSs CUDA program…

```
...
MSG: [1] GPU 0 TRANSFER STATISTICS
MSG: [1]     Total input transfers: 31.25 KB
MSG: [1]     Total output transfers: 7.8125 KB
MSG: [1]     Total device transfers: 0 B
MSG: [2] GPU 1 TRANSFER STATISTICS
MSG: [2]     Total input transfers: 31.25 KB
MSG: [2]     Total output transfers: 7.8125 KB
MSG: [2]     Total device transfers: 0 B
```



… we get a summary of memory transferences

- Total input transfers (from host to device)
- Total output transfers (from device to host)
- Total device transfers (from one device to the other device)

Runtime support (device configuration)

Enable or disable CUDA devices (debug)

```
$ export NX_ARGS="--disable-cuda=<yes/no>"
$ export NX_DISABLECUDA=<yes/no>
```

Define the maximum number of GPUs to use

```
$ export NX_ARGS="--gpus=<integer>"
$ export NX_GPUS=<integer>
```

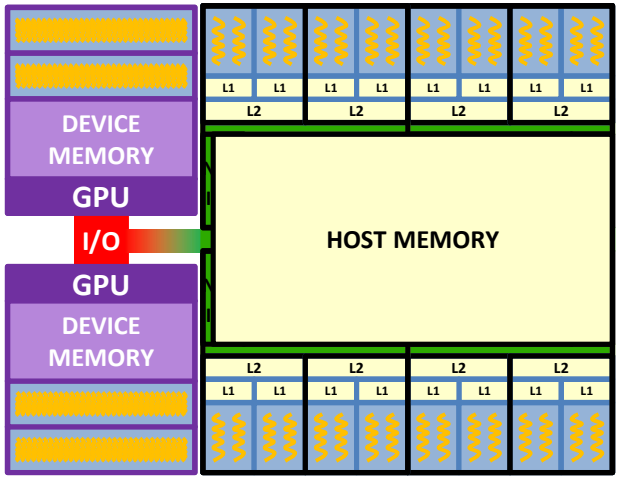
CUBLAS support and context management

- CUBLAS functions → execution stream (overlap)
- RT takes care of properly allocation of streams/handles

```
$ export NX_ARGS="--gpu-cublas-init=<yes/no>"
$ export NX_GPUCUBLASINIT=<yes/no>
```

Configurable limit on GPU memory usage (%)

```
$ export NX_ARGS="--gpu-max-memory=<positive>"
$ export NX_GPUMAXMEM=<positive>
```



Runtime support (device memory mechanisms)

Overlap data transfers and computation

```
$ export NX_ARGS="--gpu-overlap=<yes/no>"
$ export NX_GPUOVERLAP=<yes/no>
```

Data prefetching

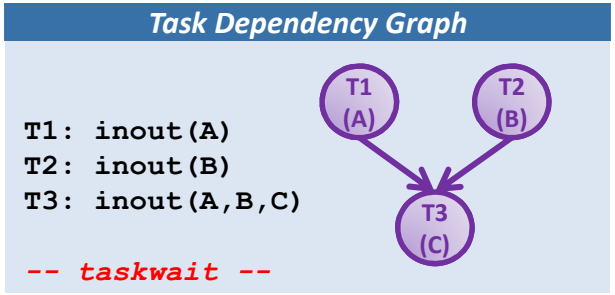
```
$ export NX_ARGS="--gpu-prefetch=<yes/no>"
$ export NX_GPUPREFETCH=<yes/no>
```

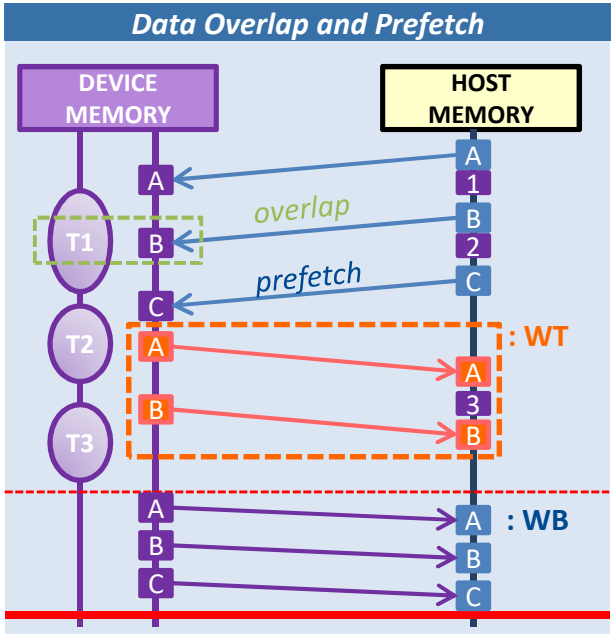
— Transfer input data to GPU memory well before tasks need the data

Configurable data cache policy

```
$ export NX_ARGS="--gpu-cache-policy=<wb|wt|nocache>"
$ export NX_GPU_CACHE_POLICY=<wb|wt|nocache>
```

— No Cache: cache disabled (not recommended)
— Write-back (WB) updates host data if needed
— Write-through (WT) after the execution of each task, output data is updated on the host





Runtime Support (debugging device operations)

Tracing device operations (--verbose-devops)

```
$ export NX_ARGS="--verbose-devops"
$ ./program
```

Getting a sequence of…

- … Device memory allocation
- … Device memory copies
- … Device memory free

Output example

```
[1] tryGetAddress _device(GPU)._memAllocate( memspc=1, allocSize=4096, wd=4 [..], copyIdx=0 ); ret 0x90..00
[1] tryGetAddress _device(GPU)._memAllocate( memspc=1, allocSize=4096, wd=4 [..], copyIdx=1 ); ret 0x90..00
[1] _device(GPU)._copyIn( copyTo=1, hAddr=0x7f..b0 [0.007], dAddr=0x90..00, len=4096, _pe, ops, wd=4 [..]);
[1] _device(GPU)._copyIn( copyTo=1, htAddr=0x7f..b0 [32], dAddr=0x90..00, len=4096, _pe, ops, wd=4 [..]);
[0] _device(GPU)._copyOut( copyFrom=1, hAddr=0x7f..b0, dAddr=0x90..00, len=4096, _pe, ops, wd=-1 [..]);
[0] _device(GPU).memFree(  memspace=1, devAddr=0x90..00);
[0] _device(GPU).memFree(  memspace=1, devAddr=0x90..00);
```

Runtime Support (getting additional help)

Getting Nanos++ runtime version (useful reporting a bug)

```
$ nanox --version
nanox 0.9a (git master 0765872 2015-06-08 14:03:54 +0200 developer version)
Configured with: /home/bsc56/bsc56678/dev/nanox/configure --prefix=/home/user/apps/ompss
--disable-instrumentation --disable-debug --enable-gpu-arch --with-cuda=/opt/cuda/5.0/
--enable-opencl-arch --with-opencl-include=/opt/cuda/5.0/include --disable-resiliency
--with-hwloc=/apps/HWLOC/1.5.1
```

Getting Nanos++ runtime options

```
$ nanox --help

Task depth throttle:
   Throttle policy based on tasks depth
   NX_ARGS options
      --throttle-limit [=]<positive integer>
        Defines maximum depth of tasks

Core [Scheduler]:
   Policy independent scheduler options
   NX_ARGS options
      --checks [=]<positive integer>
        Set number of checks before Idle (default = 1)
      --spins [=]<positive integer>
        Set number of spins on Idle before yield (default = 1)
```

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact

https://www.linkedin.com/in/xteruel