www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs-2 and Accelerators

Rosa M. Badia, Xavier Martorell, Xavier Teruel, and Orestis Korakitis

Barcelona, September 6th 2022

Outline: OmpSs-2 and Accelerators

OmpSs-2 (*driven* by **CUDA**)
- Motivation
- Introduction: execution model, memory model
- Language support for accelerators
- Use case: porting to CUDA

OmpSs-2 (*driven* by **OpenACC**)
- Use case: OpenACC motivation
- A trade-off between productivity and performance
- Compile support for accelerator

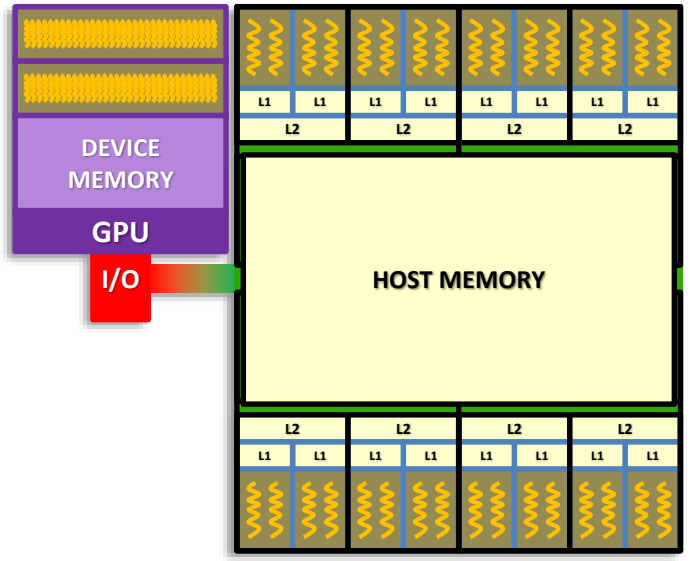
Hands-on exercises (description)

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 (OmpSs-2+CUDA vs CUDA)

```
#pragma oss task in([n]x) inout([n]y) device(cuda) \
            ndrange(1,N,128)
__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 oss 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

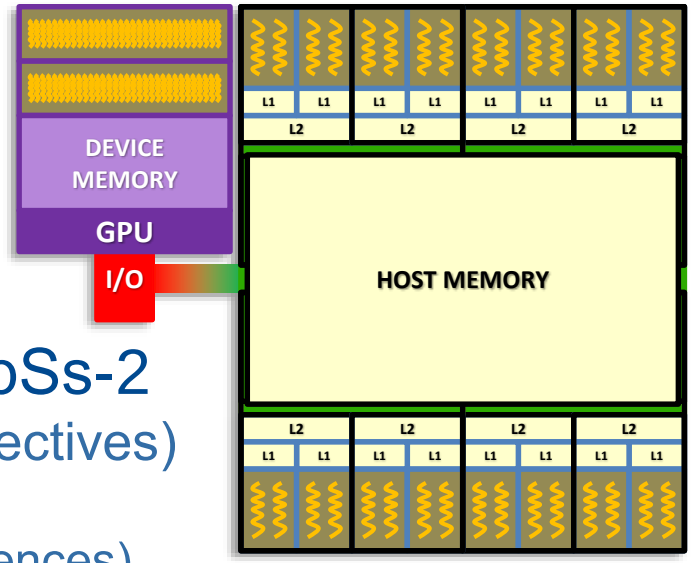
copy from device

dev. mem free

BSC accelerator's 1st proposal (preliminar)

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-2

- OmpSs-2 expressiveness (through compiler directives)
  - » Task based programming model
  - » Data directionality information (in/out/inout dependences)
    - *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

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Introduction

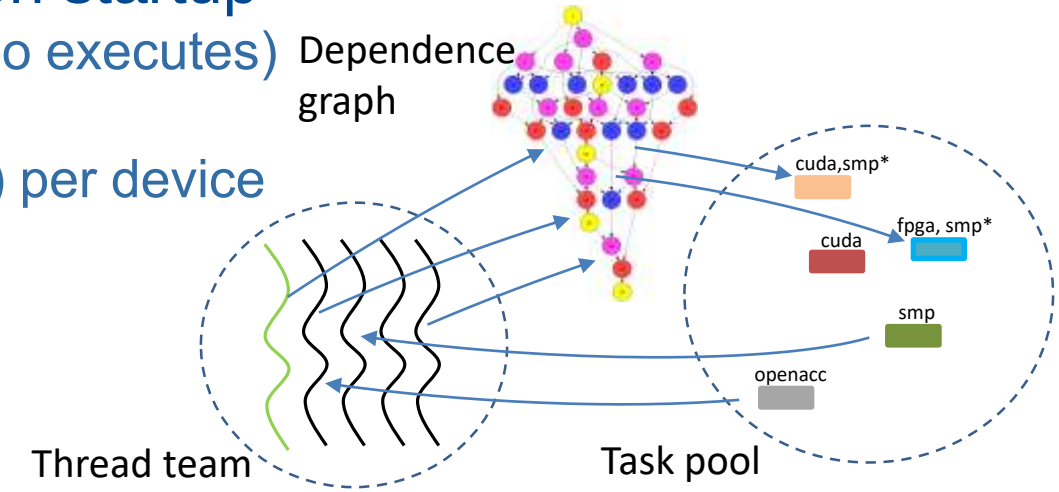
PUMPS: OmpSs-2 + Accelerators

Barcelona, September 6th 2022

Execution Model (OmpSs-2 + Accelerators)

Global thread team created on startup

- One worker starts main task (also executes)
- N-1 workers execute tasks
- One representative (*aka*. helper) per device



All get work from a task pool

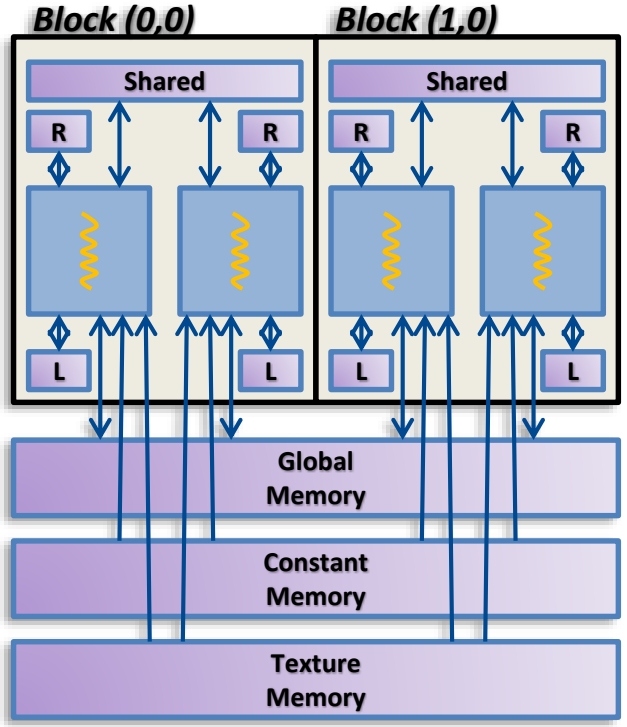
- Device kernels become OmpSs-2 tasks
- Task is labeled with (at least) one target device
- Scheduler decides which task to execute
- Tasks may have several targets (versioning)

Memory Model (device memory hierarchy)

Device local memory (+ registers)
- 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

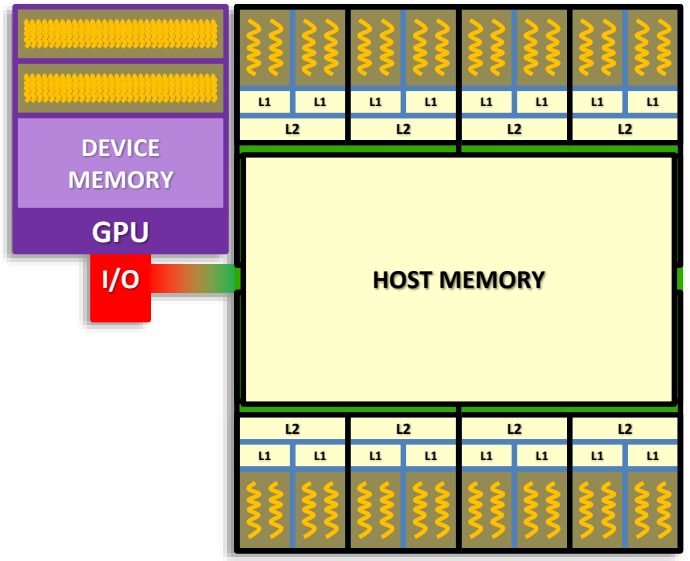
Device global memory (+ constant/texture)
- Accessible by all threads (as well as the host)
- Higher latency (hundreds of cycles) →
- Lower throughput



Memory Model (OmpSs-2 + Accelerators)

A global (logical) memory address space

Runtime handles device/host memories
- Pure SMP systems → no extra runtime support
- Distributed/heterogeneous environments
  - » Multiple physical memory address spaces exist
  - » Versions of the same data can reside on them
  - » Data consistency ensured by the runtime system



Device memory model (handled by device native or OmpSs-2)
- Host memory ←→ device global memory (automatically)
- Tasks may also allocate device shared memory (on demand)

The Task Workflow (OmpSs-2 + CUDA) [1/3]

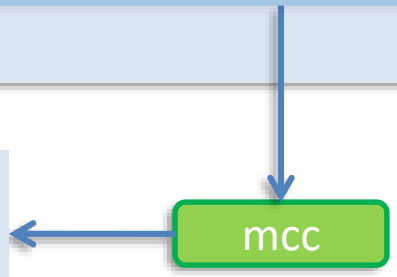
Compiler generates a stub function task that invokes the kernel

— Using the information at ndrange and shmem clauses

```
#pragma oss task in([n]x) inout([n]y) device(cuda) \
                 ndrange(1,N,128)
__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
  ...
}
```

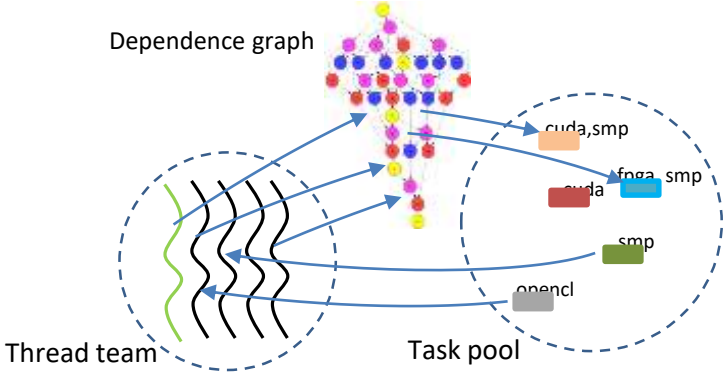


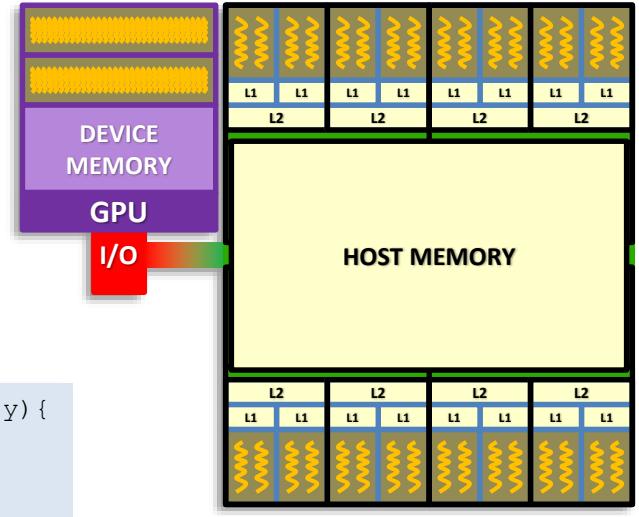

```
void saxpy_ol(int N, float a, float *x, float *y){
   ... // translates x -> d_x; y -> d_y
   saxpy<<<(N/128, 128>>>(N, a, d_x, d_y);
}
```

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

Helper thread: setup and execute phases

- On task setup/pre-fetch
  - » Determines allocation/copy requirements (*invalidates if needed*)
  - » Determines data operations execution stream and CUDA events (synchronize)
  - » Determines kernel execution stream and CUDA events (synchronize)
- On task execution
  - » 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);
}
```

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

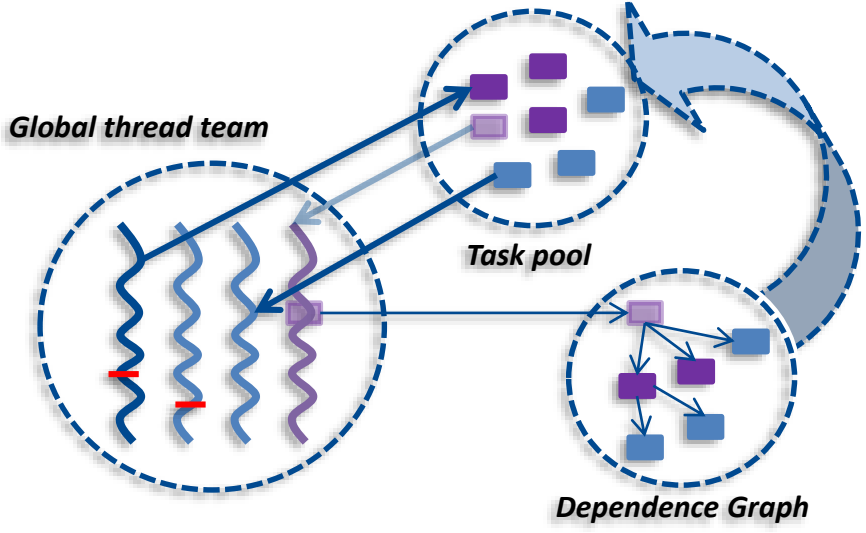
Helper thread: idle phase

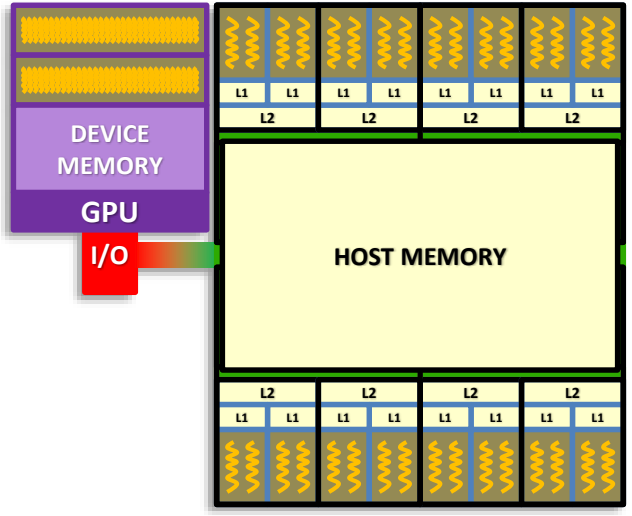
— On idle loop (spin)

» Check CUDA events → fulfil OmpSs task dependences

Any thread: on task synchronization (taskwait)

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





OmpSs-2 GPU Terminology

OmpSs-2 Device	CUDA Device	OpenCL Device	OpenACC
Local memory (thread)	Local memory	Private Memory	Local memory (vector)
Shared memory (block)	Shared memory	Local memory	Shared memory (gang)
Global memory (device)	Global memory	Global memory	Global memory (device)
NDRange*	Grid dimensions	NDRange	By means of directives
Thread	Thread	Work item	Vectors
Block	Block	Work group	Gangs

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

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Language Support

PUMPS: OmpSs-2 + Accelerators

Barcelona, September 6th 2022

Extended task directive

Device information: the task directive

```
#pragma oss task [clause[[,] clause]...]
{structured block}
```

Where clause:

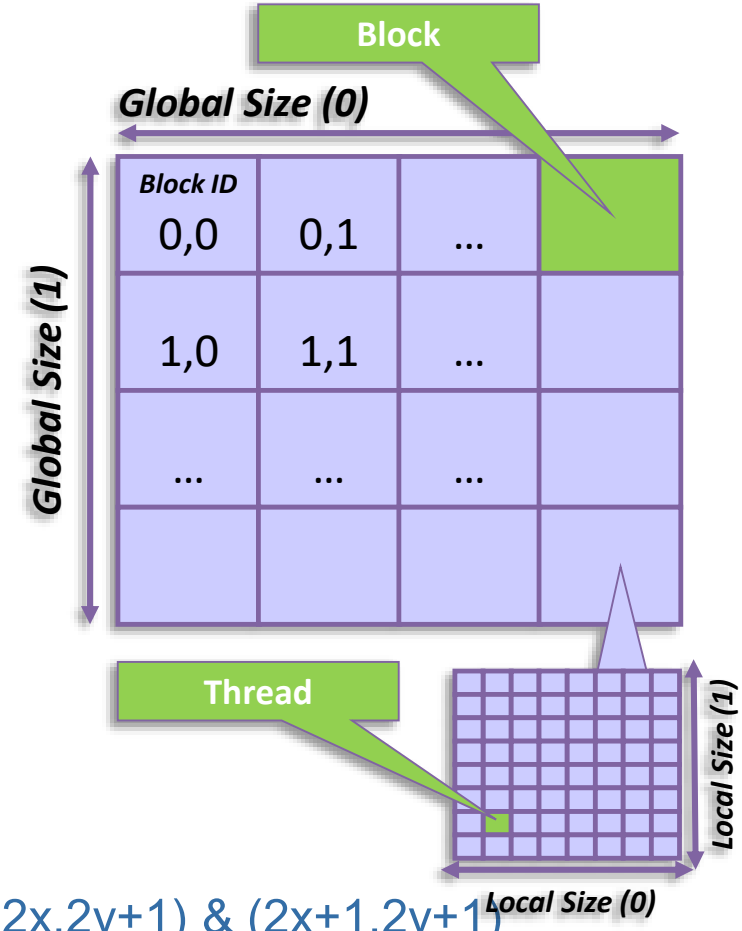
- **device**(type): specific device to run the following task (smp, cuda, openacc)
- **ndrange**(dimensions) → *not needed in OpenACC*
- **shmem**(size) → *not needed in OpenACC*
- ***implements****(function-name)**
- the **in**, **out**, and **inout** dependence clauses extend their semantics

Preferably annotating function prototypes (at header files)

Device clauses: the “ndrange” philosophy

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 data (x,y) → M[x][y]
  - » thread (x,y) computes data (2x,2y),(2x+1,2y),(2x,2y+1) & (2x+1,2y+1)

Device clauses: the “ndrange” clause

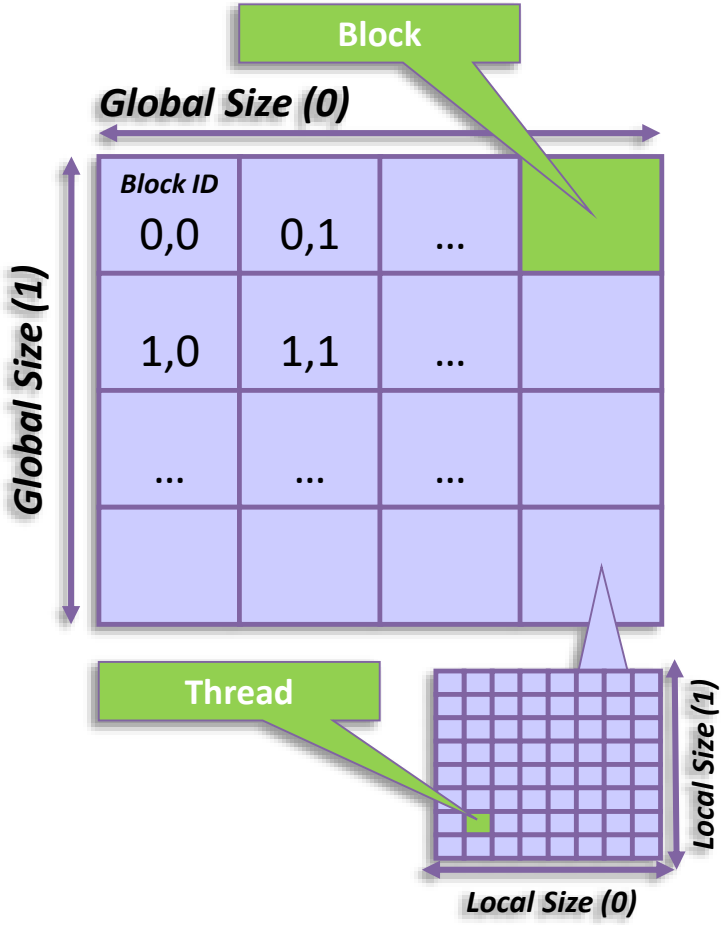
```
#pragma oss task device(type) [ ndrange(…) ]
```

ndrange(2, G, L) // G[2] = {32,32} L[2] = {8,8}
- » n → 1, 2 or 3: number of dimensions
- » global_array → size array of n elements
- » local_array → size array of n elements

ndrange(2, 32, 32, 8, 8)
- » 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 oss task device(cuda) [clauses]
void kernel_bridge ( arg_type1 arg1,…){
   dim3 dB(…), dT(…);
   kernel_code<<<dB,dT>>>( arg1,…);
}
```



Device clauses: the shmem clause

The shmem clause

```
#pragma oss task device(type) [ shmem(…) ]
```

- 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 oss task in(X[N]) out(Y[N]) shmem(1024) \
                 device(cuda) ndrange(1, N, BS)
__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, BS, 1024>>>(N, a, d_x, d_y);
}
```

Device clauses: the implements clause [1/2]*

The implements clause

```
#pragma oss task device(type) [ implements(function-name) ]
```

— implements(function-name): annotate “equivalent” functions

» 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)

» Runtime will decide which one to use

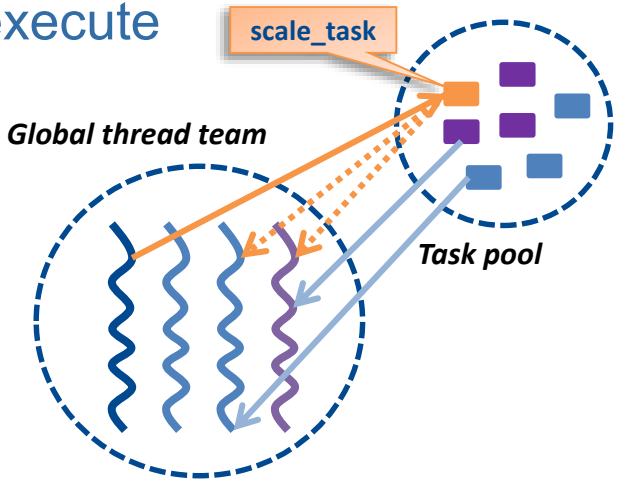
» [clause] Can be skipped when outlined function == function-name

**** The implements clause is not included in OmpSs-2 yet***

Device clauses: the 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 oss task in([N]c) out([N]b) device (smp) implements(scale_task)
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 oss task in([N]c) out([N]b) device(cuda) implements(scale_task) \
                ndrange(1,N,128)
__global__ void scale_task_cu(double *b, double *c, double a, int N);
```

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);
   . . .
}
```

Device clauses: dependence clauses

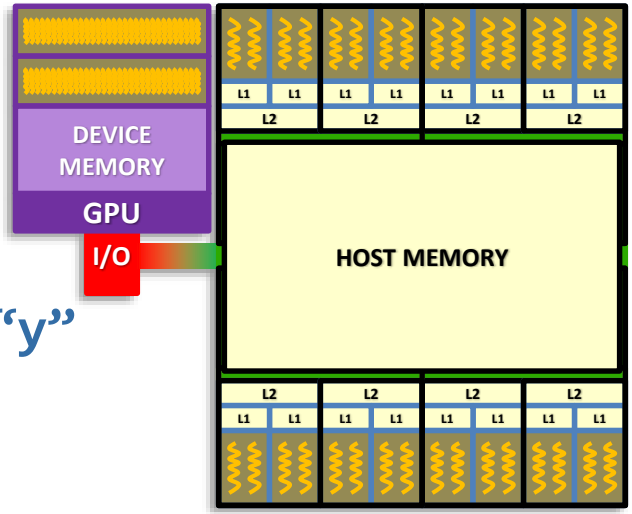
Dependence clauses extends their semantics to:

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

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

Execution of saxpy:

1. Request a copy of **“x”** and **“y”** in GPU device
2. Execute the CUDA kernel (**`saxpy`**)
3. Device now have the “last” updated version of **“y”**



www.bsc.es

Use Case

PUMPS: OmpSs-2 + Accelerators

Barcelona, September 6th 2022

Use case: AXPY Algorithm

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. Verify correctness of the SMP version
?. Use these tasks (specific calls)

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 oss 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 oss task in([n]x) inout([n]y) device(smp)
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 oss task in([n]x) inout([n]y) device(cuda)\
            ndrange(1,n,128)
__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];
}
```

Accelerator productivity and performance

Approach	Prod	Perf
Stand-alone CUDA		
OmpSs-2 + CUDA		

Slide disclaimers

***CUDA** delivers awesome coding productivity w.r.t., e.g., OpenGL, but we only want to use 3 (easy) colours here. Please interpret colours as relative to each other*

***CUDA** productivity and performance*

- *Stand-alone CUDA is hard to program for non GPU oriented programmers*
- *OmpSs-2 + CUDA increase the productivity, alleviating the burden of memory transfers and synchronization*
- *A very good approach for already parallelized kernels or libraries (e.g., cuBLAS)*
- *On incremental parallelisation there is still room for [productivity] improvement*

What is OpenACC?

The OpenACC API provides a set of:

- compiler directives (pragmas)
- library routines and
- environment variables

that can be used to write data-parallel C, C++ or Fortran programs, that run on accelerator devices, including GPUs and CPUs

Directives provide the compiler with information that is not specified in the language standard

- C/C++ have the form of a pragma: `#pragma acc directive-name`
- Fortran have the form of a comment: `!$acc directive-name`

~ OmpSs-2 (but accelerator oriented)

Why OpenACC?

As a directive-based programming model

- reuse existing (CPU) code, no need to rewrite/redesign
- same source code may run on different systems
  - » OpenACC code can also be compiled by non-OpenACC compilers by ignoring the pragmas
- simpler workflow than CUDA, i.e., increased productivity:
  - » easiest way to make an application heterogeneous
  - » start with a sequential version; incremental parallelization.
  - » performance gain with minimal effort
- when absolute performance is key, it can still be combined with CUDA!

OpenACC asynchronous execution

Asynchronous execution is supported, but:

- requires a special clause in the construct directive: `async`
- all synchronization is responsibility of the programmer
  - » by `wait` (directive or API call)
  - » by *querying* (`acc_async_test()`) the runtime for pending async operations

For independent *async* regions:

- use of *device queues*, equivalent to CUDA streams (FIFOs)
- multiple queue management is responsibility of the programmer

Code comparison (OmpSs-2+ACC vs OpenACC)

```
#pragma oss task in([n]x) inout([n]y) device(openacc)
void saxpy(int n, float a, float *x, float *y);

void saxpy(int n, float a, float *x, float *y)
{
#pragma acc kernels
   for (int i = 0; i < n; i++)
       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;
  // invoke task
  saxpy(N, a, x, y);

  #pragma oss taskwait

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

task declaration

synchronization

```
void saxpy(int n, float a, float *x, float *y, int queue);

void saxpy(int n, float a, float *x, float *y, int queue)
{
#pragma acc kernels async(queue)
   for (int i = 0; i < n; i++)
       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;
  int queue = 42; // Programmer responsibility!!
  saxpy(N, a, x, y, queue);

  #pragma acc wait(queue)
  // multiple async queues could require multiple waits!!
  for (int i=0; i<N; ++i)
      if (y[i]!=a*i+i) printf("Error\n");
```

definition

declarations

invokation

manual synch

Motivation (OpenACC limitations)

- Manual asynchronous execution
- Manual data management (although higher-level than CUDA)
- No support for both host & device parallelism in the same executable:
  - » OpenACC does host-side multithreading similar to OpenMP work-sharing
  - » Implementations do not support device parallelism + host multithreading
  - » The only way to achieve it is through:
    - *Manual programming (e.g. pthreads)*
    - *Another programming model! (e.g., **OmpSs** or OpenMP)*

Could it be simpler?

1. Port kernel to CUDA
2. Annotate device (CUDA)
3. Verify correctness of the SMP version
?. Use these tasks (specific calls)

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 oss 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 oss task in([n]x) inout([n]y) device(smp)
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 oss task in([n]x) inout([n]y) device(cuda)\
            ndrange(1,n,128)
__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];
}
```

Could it be simpler? Yes; using OpenACC

1. Port kernel to **OpenACC**
2. Annotate device (**OpenACC**)
3. Verify correctness of the SMP version
?. Use these tasks (specific calls)

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 oss 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 oss task in([n]x) inout([n]y) device(smp)
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.h 2

```
#pragma oss task in([n]x) inout([n]y) device(openacc)
void saxpy_acc(int n, float a, float* x, float* y);
```

kernel.c 1

```
void saxpy_acc(int n, float a, float* x, float* y)
{
#pragma acc kernels
 for (int i=0; i<n; ++i)
      Y[i] = X[i] * a + Y[i];
}
```

Accelerator productivity and performance

Approach	Prod	Perf
Stand-alone CUDA		
Stand-alone OpenACC		
OmpSs-2 + CUDA		
OmpSS-2 + OpenACC		

Slide disclaimers

CUDA *delivers awesome coding productivity w.r.t., e.g., OpenGL, but we only want to use 3 (easy) colours here. Please interpret colours as relative to each other*

OpenACC *may well deliver more than the performance you *need*. However, we have the lowest control on performance w.r.t. the discussed alternatives*

OpenACC productivity and performance

- *Stand-alone OpenACC is simpler to program (high productivity); but achieves less performance than CUDA*
- *OmpSs-2 + OpenACC increase the performance, allowing to improve overlapping asynchronous execution, host+device parallelism, and smarter data movements*
- *A very good approach for incremental device parallelization*

Accelerator productivity and performance

Approach	Prod	Perf
Stand-alone CUDA		
Stand-alone OpenACC		
OmpSs-2 + CUDA		
OmpSS-2 + OpenACC		
Any other option ?		

Slide disclaimers

***CUDA** delivers awesome coding productivity w.r.t., e.g., OpenGL, but we only want to use 3 (easy) colours here. Please interpret colours as relative to each other*

***OpenACC** may well deliver more than the performance you *need*. However, we have the lowest control on performance w.r.t. the discussed alternatives*

?

BSC accelerator's proposal (reviewed)

Standalone OpenACC

- Manual *async*. work scheduling
- Manual *eff*. host-device data *mng*.
- Difficult host-side parallelism
- Trade-off prod vs. perf

Standalone CUDA

- Manual work scheduling
- Manual host-device data *mng*.
- Complex code and data *mng*.
- Very good performance

BSC's proposal: combine OmpSs-2 + OpenACC + CUDA

- **OmpSs-2** expressiveness (through compiler directives)
  - » Task based programming model
  - » Data directionality information (in/out/inout dependences)
    - *Dependence detection at runtime (according with the provided information)*
    - *Automatic data movement among different Memory Address Spaces (device and host)*
- **OpenACC** ease of use + **CUDA** performance
  - » Instant porting of CPU compute kernels for GPU execution
  - » Leveraging existing CUDA kernels (including cuBLAS)
  - » After the OpenACC PoC, consider only critical kernels to be ported to CUDA

BSC accelerator's proposal (reviewed)

Standalone OpenACC
- Manual *async*. work scheduling
- Manual *eff*. host-device data *mng*
- Difficult host-
- Trade-off pro

Standalone CUDA
- Manual work scheduling
- Manual host-device data *mng*.
- e and data *mng*.
- formance

Approach	Prod	Perf
Stand-alone CUDA	🔴	🟢
Stand-alone OpenACC	🟢	🔴
OmpSs-2 + CUDA	🟠	🟢
OmpSS-2 + OpenACC	🟢	🟠
OmpSs-2 + OpenACC + CUDA	🟢	🟢

BSC's propo C + CUDA
- **OmpSs-2** ex)
  - » Task based p
  - » Data directio
    - *Dependence nformation)*
    - *Automatic d aces (device and host)*
- **OpenACC** ease of use + **CUDA** performance
  - » Instant porting of CPU compute kernels for GPU execution
  - » Leveraging existing CUDA kernels (including cuBLAS)
  - » After the OpenACC PoC, consider only critical kernels to be ported to CUDA

Compiler Support (for accelerators)

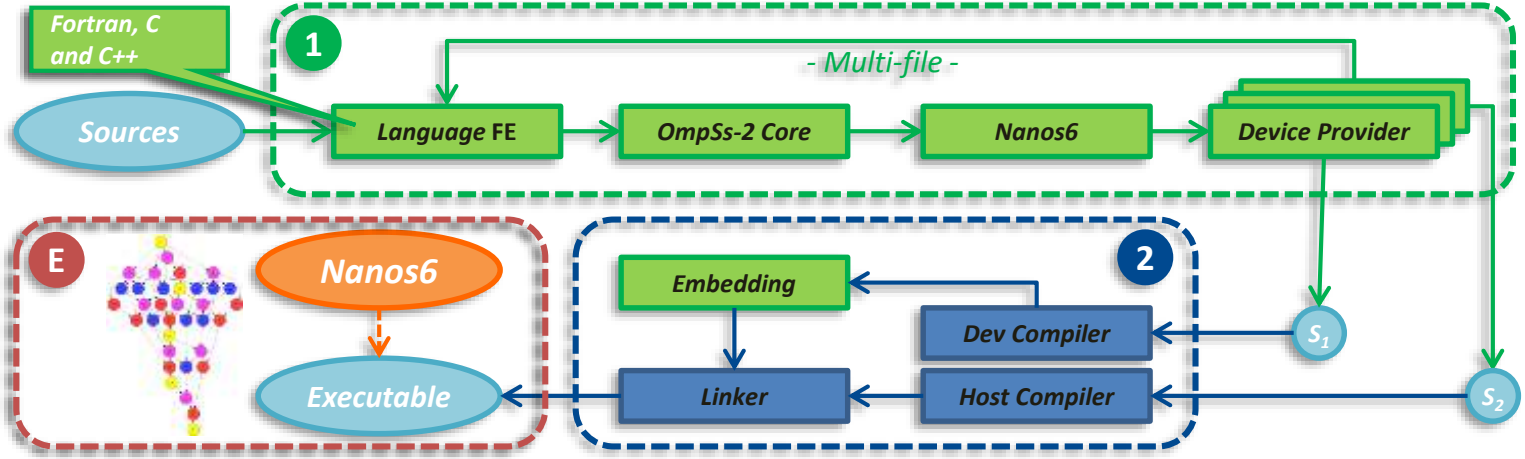
(1) Source-to-source transformation

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

Mercurium compiler
mcc → mnvcc
mcxx → mnvcxx
mfc → mnvfc

(2) Native compilation

- Kernel compiled with the NVIDIA/PGI compiler
- Kernel call files compiled with the NVIDIA/PGI compiler
- Cleansed host code compiled with the host compiler
- All object files (including CUDA) embedded on final executable



Compiler Support (CUDA example)

kernel.h

```
#pragma oss task in([n]x) inout([n]y) device(cuda)\
            ndrange(1,N,128)
__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

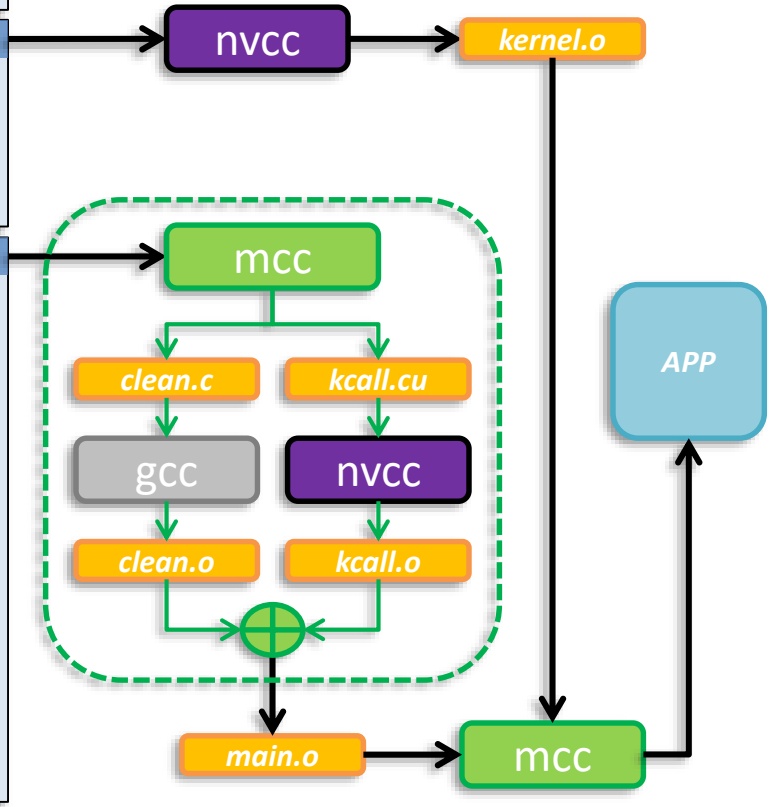
```
#include <kernel.h>
int main(int argc, char *argv[]) {
   float a=5, *x, *y;
   x = (float*)malloc(N*sizeof(float));
   y = (float*)malloc(N*sizeof(float));

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

   saxpy(N, a, x,  y);

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

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



Compiler Support (OpenACC example)

kernel.h

```
#pragma oss task in([n]x) inout([n]y) device(openacc)
void saxpy(int n, float a, float *x, float *y);
```

kernel.c

```
void saxpy(int n, float a, float *x, float *y) {
#pragma acc kernels        #pragma acc kernels async(oss_queue)
   for (int i = 0; i < n; i++)
       y[i] = a*x[i] + y[i];
}
```

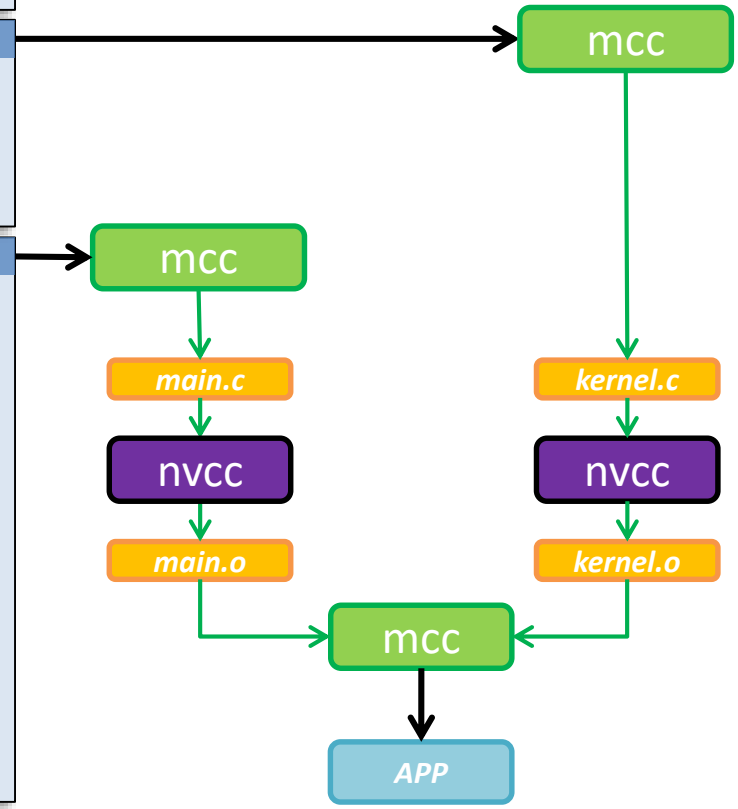
main.c

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



www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact

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

Barcelona, September 6th 2022