www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

On the role of the programmer, the compiler and the runtime when facing accelerators in OpenMP 4.0

Guray Ozen
Eduard Ayguade and Jesus Labarta

Salvador/Brazil, IWOMP’10
2014, Sep 28-30th

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

1. Motivation
2. MACC: Mercurium ACCelerator Model
3. Evaluation
4. Conclusion

Motivation

- **GPUs** have become popular
  - Performance / WATT

- **GPGPU Challenges**
  - Productivity is low, due to the different programming languages
    - **Takes more time** to learn and program
  - A lot of new concepts to consider
    - **Thread divergence** (due to conditional statements)
    - Using efficient **Multi-GPU | Concurrency** of kernel computation
    - **Minimizing Data movement (**Slow bandwidth because of PCIe)
    - Appropriate **use of GPU's memory hierarchy** (**private vs. shared vs. global memory**) and memory access patterns (**coalesced memory accessing**)
  - Optimization is hard even for experts

- **Code generation** for GPGPU?

Outcome

- **Approach** → Analyzed Directive Based APIs
  1. **OpenACC** proposal based on directives and compiler to translate to GPU code
  2. **OpenMP 4.0** accelerator model included in the OpenMP standard
  3. **OmpSs** programming model at BSC
- **Outcome** → **MACC** = **M**ercurium **ACC**elerator compiler
  - **CUDA code generator** by OpenMP 4.0 Accelerator directives
    - Involves little GPU specific compiler optimization techniques
  - Trying to influence the evolution of the OpenMP 4.0
    - **Data transfer minimization automatically** (HOST-2-GPU)
    - **Extended OpenMP 4.0** with experimental **new clauses**
      - In order to use more team memory
    - Become available **Multi-GPU task scheduling** | **Device-to-Device** data transfer
  - Based on **OmpSs suite**
    - Already supports different memory address space such as GPU
    - Generated CUDA kernels by MACC can be OmpSs task
    - All kind of tasks (SMP | CUDA | ACC) **works Asynchronously**
    - Manages **CUDA Concurrency**

OmpSs Programming Model

- Extended OpenMP & Task based programming model
  - Mercurium Compiler
  - Nanos Runtime
- Forerunner for OpenMP
  - Tasking and tasks dependences are two examples of OmpSs influence
- OmpSs Current Accelerator Supports
  - Tasks to be executed on GPU programmed in CUDA or OpenCL
  - Runtime system takes care of data movement, overlapping and scheduling
  - Doesn't generate gpu code

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

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

```
#pragma omp target device ({ smp | cuda | opencl })      \
    { copy_deps | [ copy_in (...)] [ copy_out (...)] [ copy_inout (...)] } \
    [ ndrange (…) ]
```

Support kernel based programming

```
#pragma omp task [ in (...)] [ out (...)] [ inout (...)]
{
    <<.. function or code block ..>>
}
```

To compute dependences

```
#pragma omp taskwait
```

Wait for sons

VectorADD: MACC vs OmpSs vs OpenMP 4.0

OpenMP 4.0

```
void main()
{
    double a[N], b[N], c[N];

    #pragma omp target map(to:a,b) map(from:c)
    #pragma omp teams
    #pragma omp distribute parallel for
    for (int i=0; i<N; ++i)
       c[i] = a[i] + b[i];

}
```

OmpSs

```
#pragma omp target device(cuda) ndrange(1,N,N) copy_deps
#pragma omp task in([N]a,[N]b) out([N]c)
__global__ void vecadd(double* a, double* b, double* c, int N)
{
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

void main()
{
    double a[N], b[N], c[N];

    vecadd(a, b, c, N);
    #pragma omp taskwait
}
```

MACC

```
void main()
{
    double a[N], b[N], c[N];

    #pragma omp target device(acc) copy_deps
    #pragma omp task in(a,b) out(c)
      #pragma omp teams
      #pragma omp distribute parallel for
      for (int i=0; i<N; ++i)
        c[i] = a[i] + b[i];

    #pragma omp taskwait
}
```

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

MACC
Code Generation

- **Offload**
  - Starts after `#pragma omp target device(acc)`
  - Device clause extended to specify device type, not physical device number (much better support for multiple GPUs)

- **Besides task level parallelism for `target` directive**
  - Generated CUDA codes from task's code region will be OmpSs task
  - Works asynchronously

- **Kernel configuration**
  - `#pragma omp teams | num_threads(int) | thread_limits(int)`
  - If not specified MACC defaults to **one iteration per block/thread**

- **Work-sharing Directives**
  - `#pragma omp distribute` → Iterations of loop distributed among **CUDA blocks**
  - `#pragma omp parallel for` → Iterations of loop distributed among **CUDA threads**

    Nesting to enable multiple thread dimensions (2D/3D)

MACC
Code Generation

- **Cyclic distribution**
- **1 iteration → 1 CUDA Block / Thread**
- **If at all possible, remove redundant iteration**
  - Thread Divergence in CUDA
  - Assign one iteration to one thread/block

MACC: Input

```
#pragma omp target device(acc)
#pragma omp task
#pragma omp teams
#pragma omp distribute
for (i = 0; i < 48; ++i)
{
    <..Computation Code..>

    #pragma omp parallel for
    for (j = 0; j < 64; ++j)
    {
        <..Computation Code..>

        #pragma omp parallel for
        for (k = 0; k < 32; ++j)
           <..Computation Code..>
    }
}
```

MACC

MACC: Generated Kerneler

```
void macc_kerneler(...)
{
  /*Mercurium ACCelerator Compiler - KERNELER*/
  dim3 gridDim, blockDim;

  gridDim.x = MIN(_CUDA_MAX_TEAM, 48);
  blockDim.x = MIN(_CUDA_MAX_THREAD, 64);
  blockDim.y = MIN(_CUDA_MAX_THREAD, 32);

  macc_generated_kernel <<< gridDim, blockDim,…>>> (...);
}
```

MACC

MACC: Generated CUDA Kernel

```
__global__ void macc_generated_kernel(...)
{
 int _macc_i = macc_blkidx();
 ~~for(int _macc_i = macc_blkidx(); _macc_i < 48; _macc_i+=macc_grdnumx())~~
 {
    <..Computation Code in CUDA..>
    int _macc_j = macc_tidx();
    ~~for (_macc_j = macc_tidx(); _macc_j < 64; _macc_j += macc_blknumx())~~
    {
       <..Computation Code in CUDA..>

       int _macc_k = macc_tidy();
       ~~for (_macc_k = macc_tidy(); _macc_k < 32; _macc_k += macc_blknumy())~~
           <..Computation Code in CUDA..>
    }
  }
}
```

MACC
Code Generation

Data Transfer Minimized Automatically (GPU-HOST)

- OpenMP 4.0
  - Need to specify `target data` in order to stay data on device
  - Sometimes download / upload is performed with `target update` by hand
- MACC
  - **Ignored** `target data & target update`
  - Programmer only specifies **directionality of task data**, not the actual data movement
    - *#pragma omp* ***task*** *in(list) out(list) inout(list)*
  - Doesn't download data from GPU until *taskwait*

Task scheduling with Multi-GPU

- OpenMP 4.0
  - *device_id* is given by hand → *device(**int**)*
    - Multi-Gpu scheduling is managed by user!
  - Device-to-device data transfer is unavailable!
    - target data device(*device_id*)
- MACC
  - No *device_id*
  - Runtime can schedule **Multi-GPU | Concurrent Kernel**
  - Became available **device-2-device transfer**

MACC

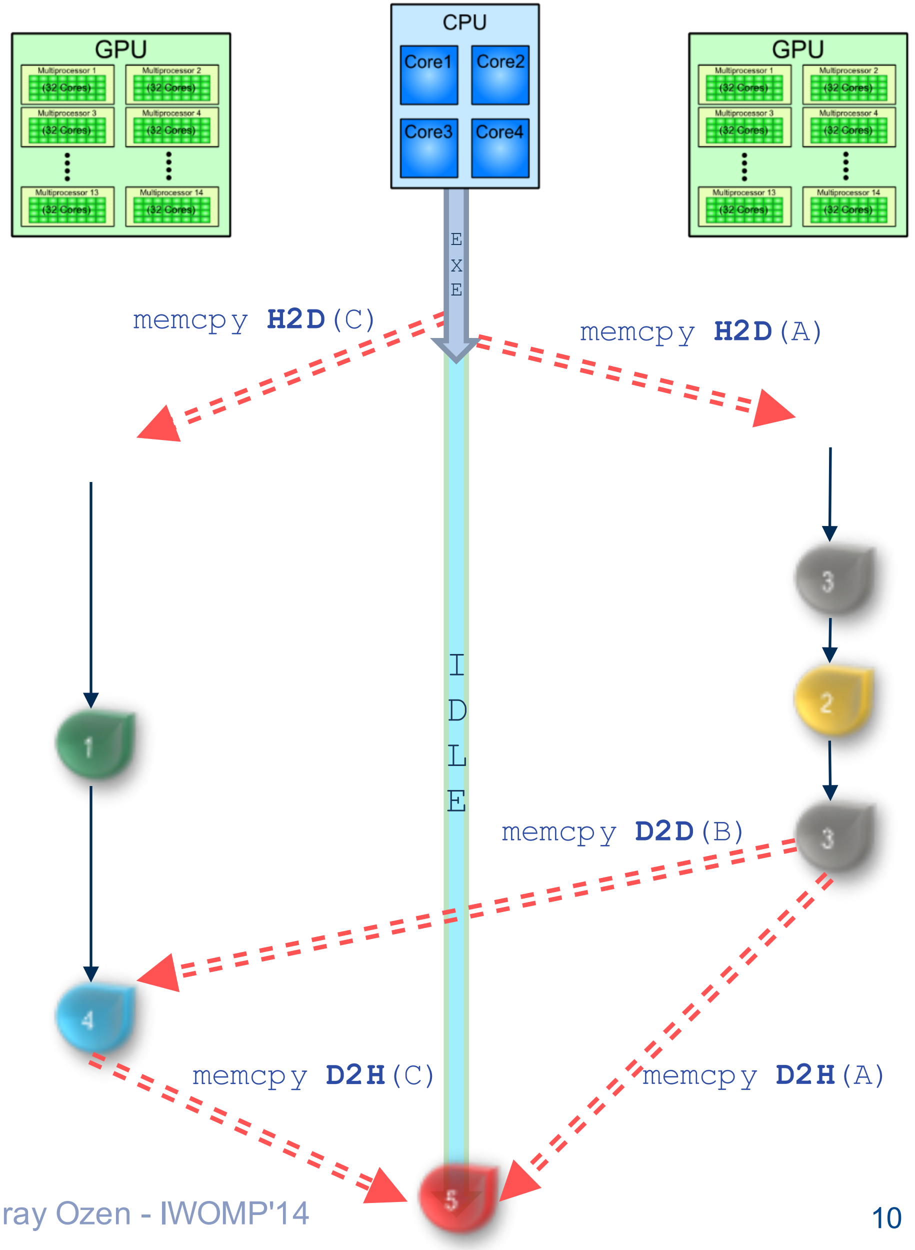
```
for (...)
{
  #pragma omp target device(acc) copy_deps
  #pragma omp task inout(x[beg:end])
  #pragma omp teams distribute parallel for
  for (i = 0; i < SIZE; ++i)
     if( cond1() )
        << ..Takes long time.. >>
     else
        << ..Sometimes takes long time.. >>
}
```

OpenMP

```
for (...)
{
  int dev_id = i % omp_get_num_devices();

  #pragma omp task
  #pragma omp target device(dev_id) \
                     map(tofrom: x[beg:SIZE])
  #pragma omp teams distribute parallel for
  for (i = 0; i < SIZE; ++i)
     if( cond1() )
        << ..Takes long time.. >>
     else
        << ..Sometimes takes long time.. >>
}
```

MACC
Minimized Data Transfers & MultiGPU

```
int main(…) {
    double A[N], B[N], C[N] , D[N];

    while (0-> 2)
    {
       #pragma omp target device(cuda) ndrange(...) copy_deps
       #pragma omp task inout(C) out(D)
         <..Optimized CUDA Kernel Invocation..>

       #pragma omp target device(acc) copy_deps
       #pragma omp task in(A) out(B)
       #pragma omp teams distribute parallel for
        for(i=0 ; i< N; ++i)
         <..Sequential Codes to generate CUDA..>

       #pragma omp target device(acc) copy_deps
       #pragma omp task inout(A,B)
       #pragma omp teams distribute parallel for
        for(i=0 ; i< N; ++i)
          <..Sequential Codes to generate CUDA..>
     }
    #pragma omp target device(acc) copy_deps
    #pragma omp task inout(C,B) in(D)
    #pragma omp teams distribute parallel for
     for(i=0 ; i< N; ++i)
      <..Sequential Codes to generate CUDA..>

    #pragma omp target device(smp) copy_deps
    #pragma omp task in(A, C)
      <..Sequential codes / Result Test..>
    #pragma omp taskwait
}
```



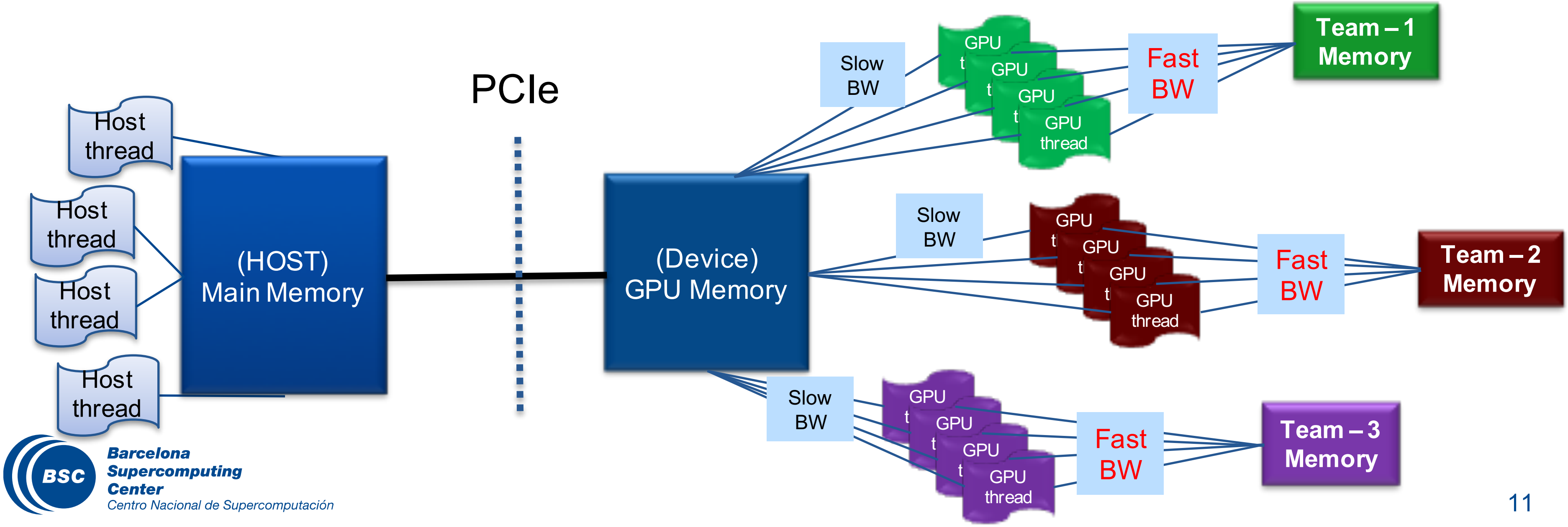
MACC
Code Generation

GPU Global Memory

- Slow & Big !

Use Team Memory

- Correspond **shared memory** in CUDA
- Each thread groups (CUDA blocks) have **one shared memory**.
- Shared memory **faster** than global memory
- **Very limited** in size (e.g. 48 KB compared to 6 GB of global memory)
- In some **hand-written CUDA** codes we observed the use of shared memory for shared data, using blocking to overcome limited size



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MACC
Code Generation

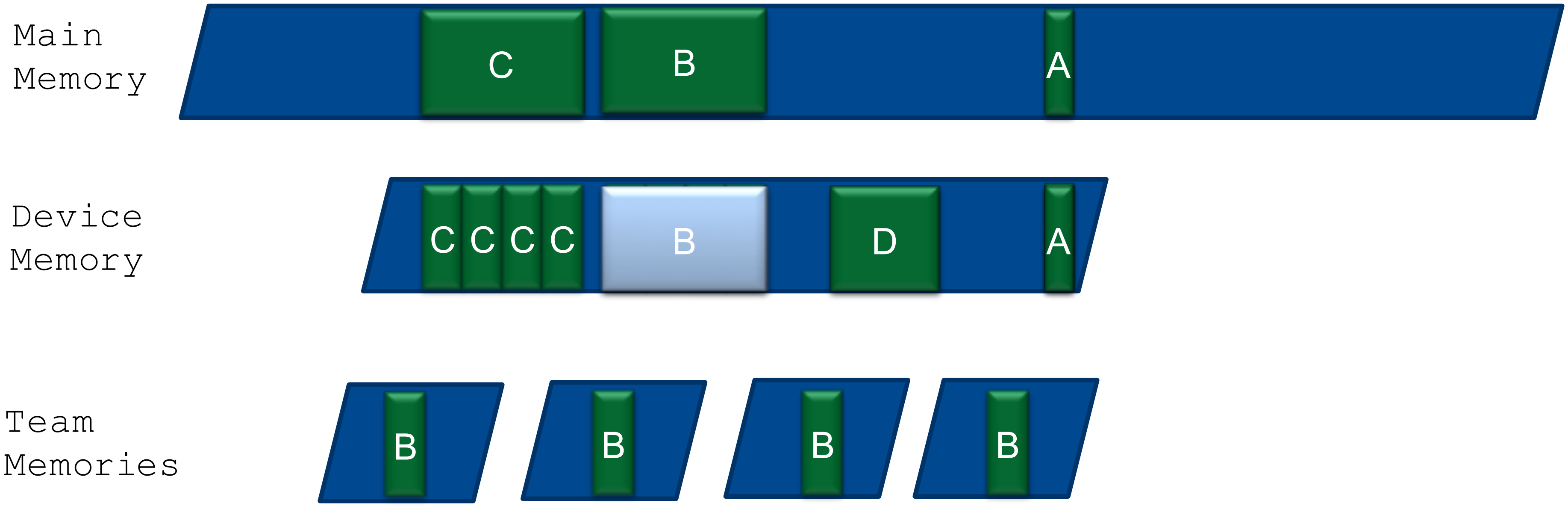
- Data sharing clauses with `teams` | `private` | `first_private`
- Offers experimental 3 new clauses for `distribute` directive
  - `dist_private([CHUNK]data1, [CHUNK]data2 …)`
  - `dist_firstprivate([CHUNK]data1, [CHUNK]data2 …)`
  - `dist_lastprivate([CHUNK]data1, [CHUNK]data2 …)`

```
#pragma omp target device(acc) copy_deps
#pragma omp task in(A[0:SMALL],C[0:HUGE]) inout(B[0:HUGE]) out(0:D[BIG])
#pragma omp teams first_private(A)
#pragma omp distribute parallel for dist_first_private([CHUNK]C) dist_first_last_private([CHUNK]B)
for(...)
   <<..Computation..>>
```

Data movement to **Device Memory**

Using **TeamMem** with **Small DATA**

Using **TeamMem** with **Big DATA**



Jacobi (A*x=B)

- **Transparent management** of data movement in MACC
- **No need for data scoping** directives in OpenMP 4.0 / OpenACC

OpenACC Baseline

```
while ( cond1() )
{

 #pragma acc kernels copyin(u) copyout(uold)
 #pragma acc loop
 for (i = 0; i<n; i++)
     <..computation with (u & uold)..>

 #pragma acc kernels copyin(uold) \
                    copyout(u) copy(err)
 #pragma acc loop reduction(+:err)
 for (i = 1; i< (n - 1); i++)
     <..computation with (u & uold)..>

  <..serial computation for cond1 ..>
}
```

OpenACC Optimised

```
#pragma acc data copy(u) copyout(err) \
                 create(uold)
while ( cond1() )
{
   #pragma acc kernels loop
   for (i = 0; i < n; i++)
      <..computation with (u & uold)..>

   #pragma acc kernels loop reduction(+:err)
   for (i = 1; i < (n - 1); i++)
      <..computation with (u & uold)..>

   <..serial computation for cond1 ..>
}
```

MACC

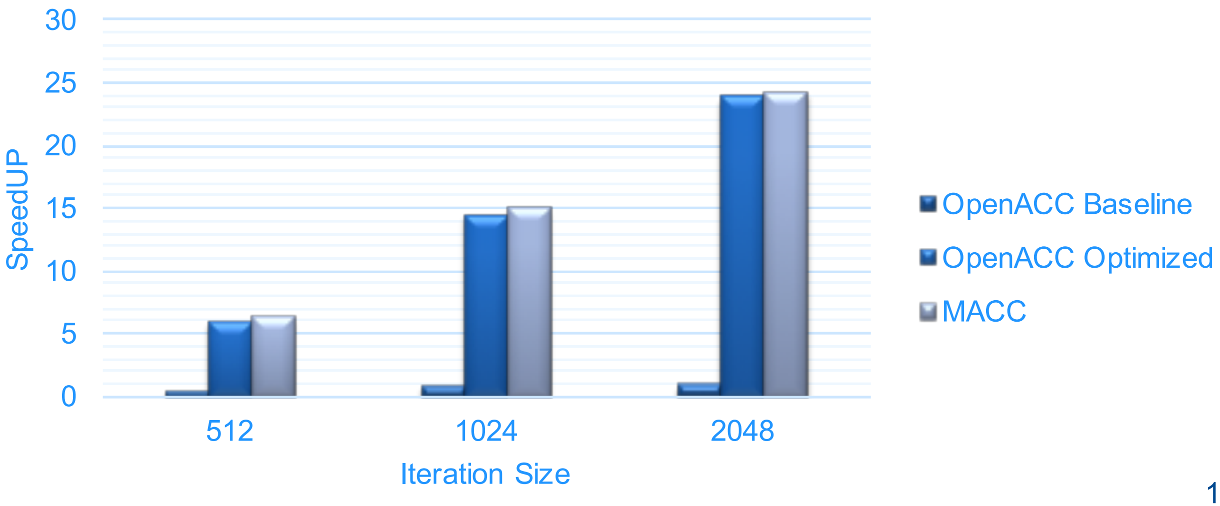
```
while ( cond1() ) {

  #pragma omp target device(acc) copy_deps
  #pragma omp task in(u) out(uold)
  #pragma omp teams distribute parallel for
  for (i = 0; i < n; i++)
     <..computation with (u & uold)..>

  #pragma omp target device(acc) copy_deps
  #pragma omp task in(uold) out(u) inout(err)
  #pragma omp teams distribute parallel for reduction(+:err)
  for (i = 1; i < (n - 1); i++)
     <..computation with (u & uold)..>

  <..serial computation for cond1 ..>
}
#pragma omp taskwait
```

Hardware

1. 2 x Xeon E5649
2. 2 x NVidia Tesla M2090
3. 24GB Main Memory

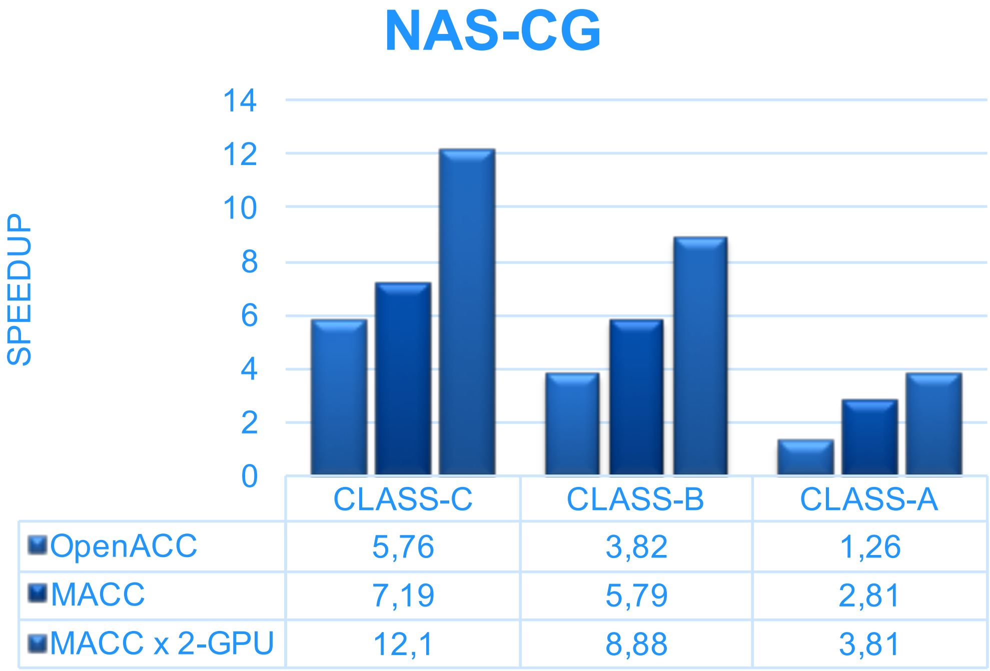
Software

1. OpenACC → HMPP
2. NVCC 5.0
3. GCC 4.6



NAS Parallel Benchmark CG

- NAS-CG Solves an unstructured sparse linear system by the conjugate gradient method
- 3 Problem Set C > B > A



	CLASS-C	CLASS-B	CLASS-A
■ OpenACC	5,76	3,82	1,26
■ MACC	7,19	5,79	2,81
■ MACC x 2-GPU	12,1	8,88	3,81

- **Effects of Runtime**
  - **How important Task-Scheduling**
  - **Multiple-GPU**
    - **Device-to-Device transfer**
- With 2-GPU
  - Easy to develop with MACC
- MACC is better even with one GPU
  - Supports CUDA concurrency by streaming
  - Optimized task scheduling by Nanos runtime

DG Kernel

- To calculate climate benchmark developed by NCAR (National Center for Atmospheric Research)

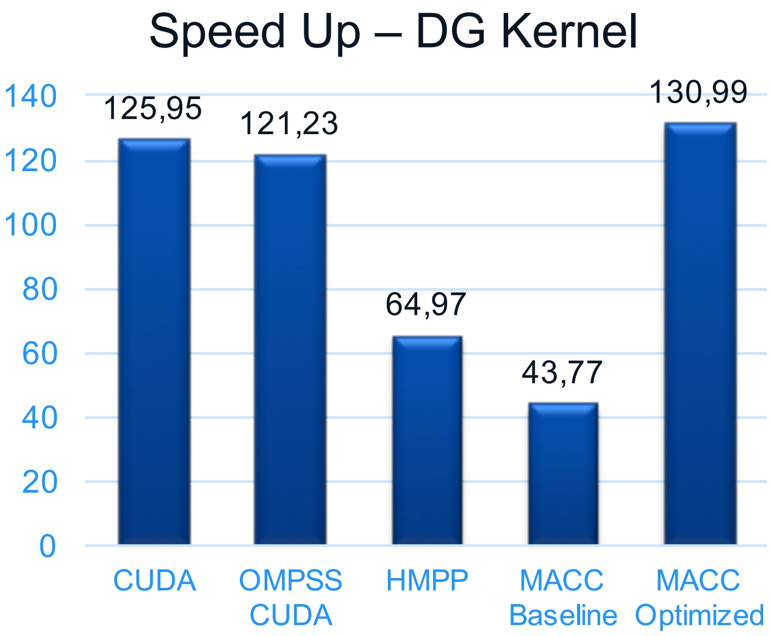
- 4 versions of DG-Kernel
  1. **CUDA hand optimized** code developed at NCAR
  2. **OmpSs + CUDA** kernel
  3. **OpenACC** code developed NCAR
  4. **MACC**

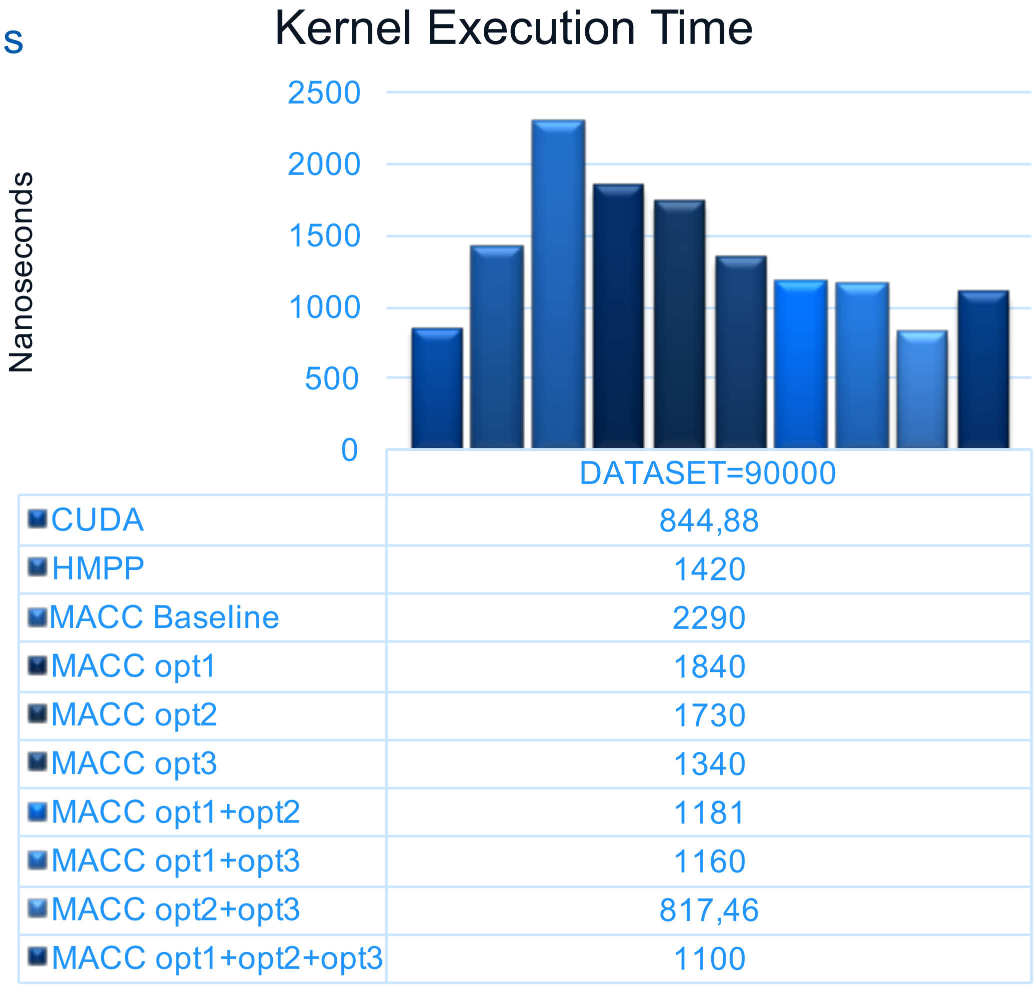
- Used to demonstrate:
  - MACC can have better results than hand-optimized CUDA
  - MACC optimization techniques
  - Compare MACC with hand optimized CUDA program

DG Kernel

- 3 Optimization Techniques of MACC are used
  - **Opt1** New team memory techniques
  - **Opt2** Removing redundant For iterations
  - **Opt3** Start assign with 2 dim of thread

- **MACC has better result!**


Speed Up – DG Kernel


Kernel Execution Time

	DATASET=90000
■ CUDA	844,88
■ HMPP	1420
■ MACC Baseline	2290
■ MACC opt1	1840
■ MACC opt2	1730
■ MACC opt3	1340
■ MACC opt1+opt2	1181
■ MACC opt1+opt3	1160
■ MACC opt2+opt3	817,46
■ MACC opt1+opt2+opt3	1100

Conclusion

- Presented a **MACC** research compiler to include new accelerator directives in the OmpSs programming model
  - Avoid the use of kernel programming using CUDA/OpenCL
  - Programming productivity and performance
  - New extensions proposed to OpenMP 4.0

- **Compilers** plays key factor
  - Code generation
  - Applying GPU specific optimizations

- Effects of **runtime & programmer** are also important
  - Managing many kernels with many GPU?
  - Ability to use multi GPU
  - Using different pragma directives

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
guray.ozen@bsc.es

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

MACC

MERCUIRUM ACCELERATOR COMPILER

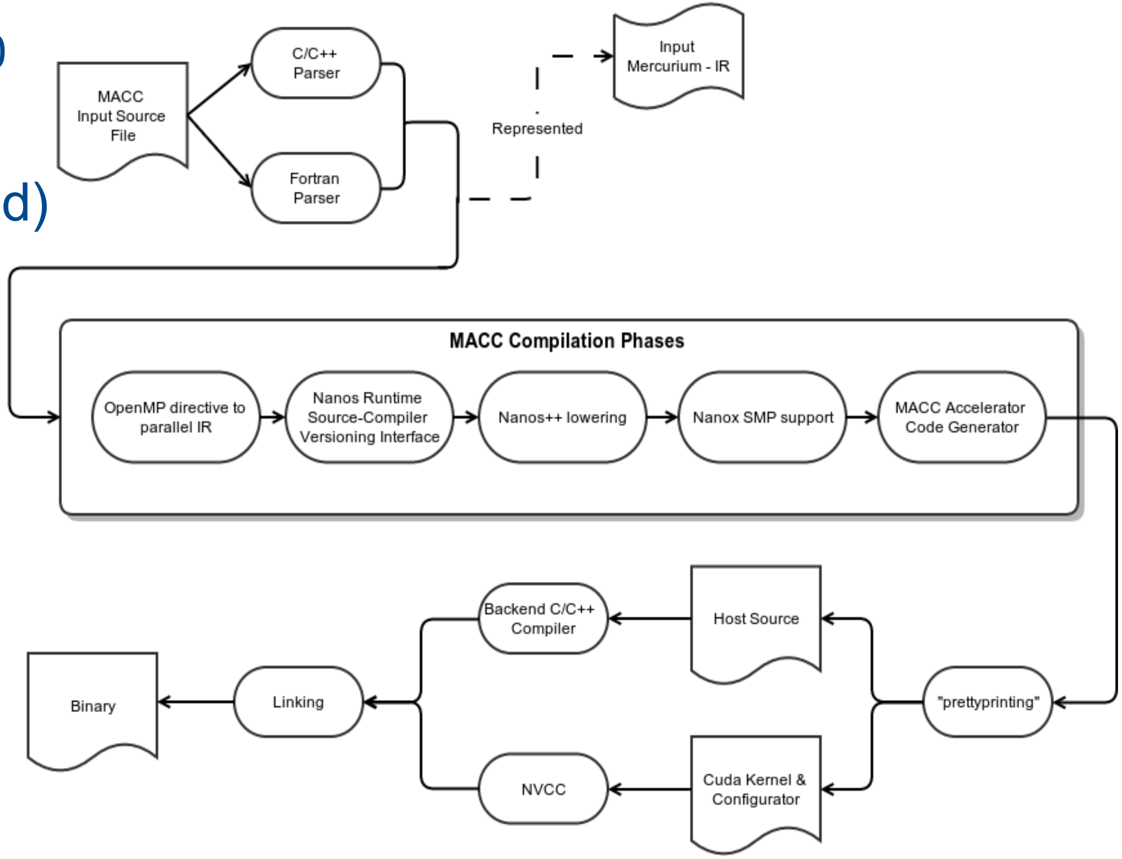
MACC → Mercurium ACCelerator compiler

- **Approach**
  - Start from OmpSs
    - Developed at BSC
    - Already providing support for task dependencies and offloading CUDA/OpenCL kernels to accelerators
  - Add the minimum set of OpenMP 4.0 accelerator model directives into the OmpSs programming in order to avoid kernel programming
  - Add extra directives for additional programming productivity and performance, if necessary

- **OmpSs** programming model implemented with
  - **Mercurium Compiler**
    - Source-2-Source compiler
    - Easy to prototype new code transformations and generation
    - MACC required some changes in existing compilation phases and a new phase
  - **Nanos++ runtime system**
    - Extremely good task parallelism
    - Supports Heterogeneous task (CUDA, OpenCL, SMP)
    - No changes required to support MACC code generation

Compiler phases in Mercurium

1. **Parser** (modified)
   - To parse new OpenMP 4.0 directives
   - Added new IR for OpenMP 4.0
2. **Nanos++ Lowering** (modified)
   - It lowers OpenMP directives
   - Some semantics are changed
3. **MACC lowering** (new)
   - CUDA code generation
4. **Compilation Driver**
   - Backend compiling & linking



MACC
Offers New Clauses in order to use Team Memory!

- IF DATA SMALL? (existing OpenMP clauses)
  1. Existing Clauses for `#pragma omp teams`
     1. `private(`*`list`*`)`
     2. `firstprivate(`*`list`*`)`
- IF DATA BIG? (new MACC clauses)
  1. **New Clauses for** `#pragma omp distribute`
     1. `dist_private(`*`[CHUNK]data1, [CHUNK]data2 …`*`)`
     2. `dist_firstprivate(`*`[CHUNK]data1, [CHUNK]data2`*` )`
     3. `dist_lastprivate(`*`[CHUNK]data1, [CHUNK]data2`*` )`

MACC

MACC: Generated Kerneler

```
void macc_kerneler(...)
{
  /*Mercurium ACCelerator Compiler - KERNELER*/
  dim3 gridDim, blockDim;
  gridDim.x = MIN(_CUDA_MAX_TEAM, 32);
  blockDim.x = MIN(_CUDA_MAX_THREAD, 8);

  int _macc_dyn_sh_mem_size =
       SMALL * sizeof(double) +  //Allocation for A[SMALL]
       CHUNK * sizeof(double) +  //Allocation for [CHUNK] C
       CHUNK * sizeof(double);   //Allocation for [CHUNK] B

  macc_gen_kernel << <gridDim, blockDim, _macc_dyn_sh_mem_size >> >(...);
}
```

MACC: Input

```
double A[SMALL], D[BIG];
double C[HUGE], B[HUGE];

#pragma omp target device(acc) copy_deps
#pragma omp task in(A[SMALL],C[HUGE]) inout(B[H
#pragma omp teams first_private(A) num_teams(32
#pragma omp distribute dist_first_private([CHUN
for (...)
<< ..Computation.. >>
```

MACC

MACC: Generated CUDA Kernel

```
__global__ void macc_gen_kernel(...)
{
  /*----[START]- Allocation & Filling for DataShared Variables on SharedMem */
    int _macc_sh_offset = 0;
    double *_macc_a = get_shared_memory(_macc_sh_offset);
    _macc_sh_offset += ((SMALL)+1);
    double *_macc_B = get_shared_memory(_macc_sh_offset);
    _macc_sh_offset += ((CHUNK)+1);
    double *_macc_C = get_shared_memory(_macc_sh_offset);
    _macc_sh_offset += ((CHUNK)+1);

    for (int _macc_sh_iter=macc_idx1d();_macc_sh_iter<CHUNK;_macc_sh_iter+ macc_blknum())
    {
      _macc_B[_macc_sh_iter] = B[_macc_sh_iter + CHUNK * macc_blkidx()];
      _macc_C[_macc_sh_iter] = C[_macc_sh_iter + CHUNK * macc_blkidx()];
    }
    macc_sync();
  /*----[END]--- Allocation & Filling for DataShared Variables on SharedMem */

  {    < ..CUDA Kernel Computation .. >    }

  /*----[START]- LastPrivate Variables Refill from SharedMem to GlobalMem */
    for (int _macc_sh_iter=macc_idx1d();_macc_sh_iter<CHUNK;_macc_sh_iter+=macc_blknum())
      C[_macc_sh_iter + CHUNK * macc_blkidx()] = _macc_C[_macc_sh_iter];
  /*----[END]--- LastPrivate Variables Refill from SharedMem to GlobalMem */
}
```