# Directives Based Approaches in an Exascale Perspective

F. Bodin / Irisa

June 2013

Lyon

# Introduction

- HPC and embedded software going for dramatic changes to adapt to massive parallelism
  - Huge market/programmers issue
  - Many codes and users not ready → ***directives based approaches***
  - Key economical competitive topic

- Performance and energy consumption intimately coupled
  - Looking for code execution time and energy consumption minimization
  - Specialized solutions based on accelerators and co-processors

- Exascale driving the next generation of technologies (and vice versa)
  - Embedded systems
  - HPC
  - Big data

# *Extract from CRAY-1 Fortran Reference Manual, 1978.*

IVDEP directives

The IVDEP directive is specified in advance of a DO statement to cause the compiler's attempts to vectorize the corresponding DO-loop to ignore any vector dependencies encountered. The IVDEP directive affects only the single DO-loop it precedes. Note that conditions other than vector dependencies may cause the inhibiting of vectorization whether or not an IVDEP directive is specified.

5.4.4 INTEGER CONTROL DIRECTIVE

The form of the single integer control directive, INT24, is:

```
INT24 v [,v] ...
```

where INT24 specifies a 24-bit integer data type and

*v* is the symbolic name of a variable or array.

# OpenMP Directives

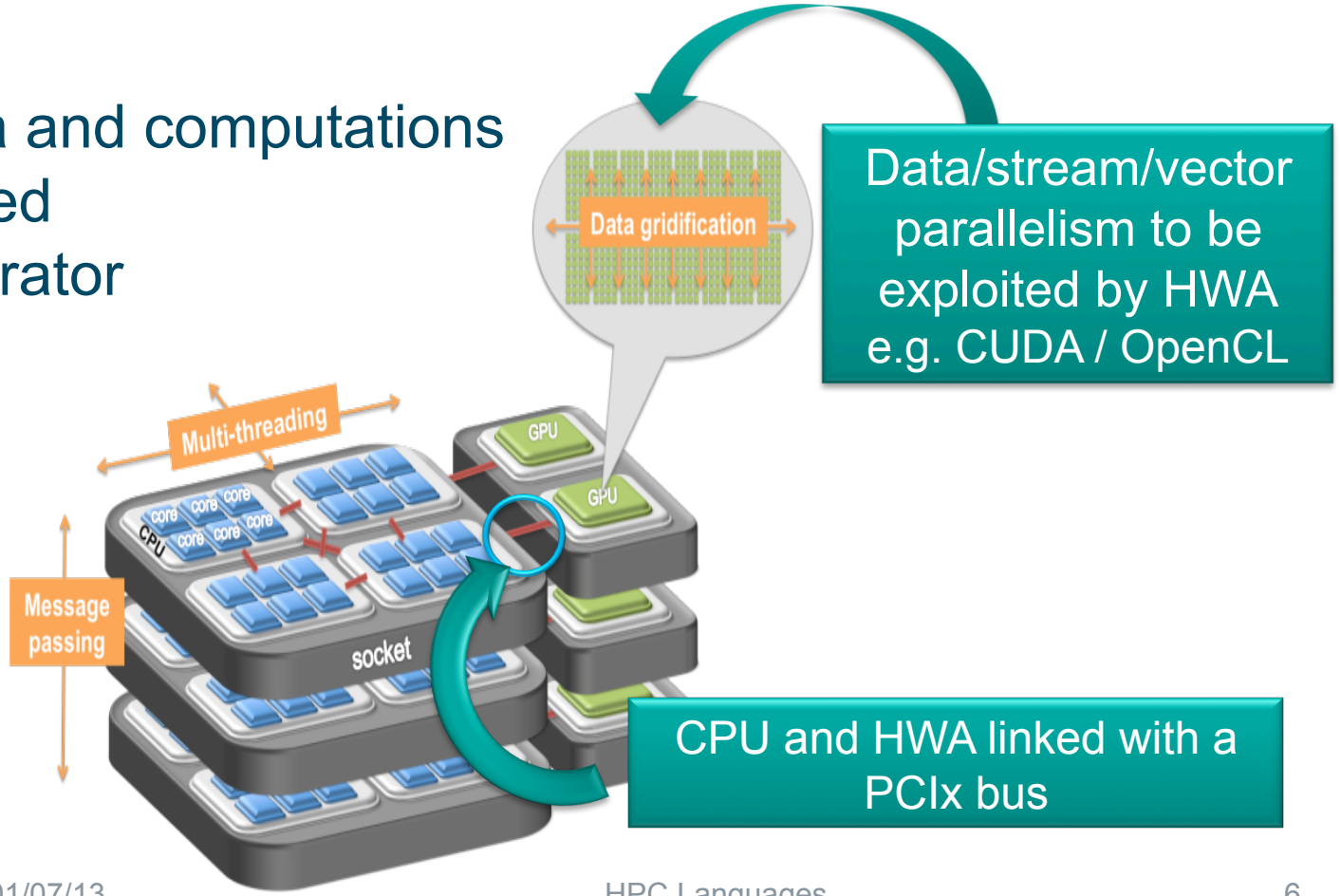
- 1980-1990
  - Alliant, Convex, Cray, Encore, IBM, Sequent, and others
  - Constructor specific  directives
  - About parallel loops
  - Failed to reach a common agreement
- 1990 → …
  - OpenMP committee
  - Parallel loops and shared memory
  - ANSI X3H5 adopted in 1997

# The Origin of Directives for Accelerators (HWA) @ CAPS

- A proof of concept in 2007
  - targeted toward Fortran users that were not OpenGL fans
  - OpenMP directives has been a previous success
  - Was also targeting FPGA
- Code maintenance was identified as a main issue
  - Only one source code to maintain
- We were looking for a solution that preserves serial code
  - CPU regular code → compiled and used as usual
  - Simplifying debugging
  - Incremental development approach
  - Avoid exit cost
- Needed to be complementary to MPI and OpenMP
  - All targeted applications used one or both of these

# Programming Model for Accelerators

- Remote Procedure Call
- Express data and computations to be executed on an accelerator



# A Few Directive Based Approaches

- F2CACC
  - Directives from NOAA → very close to OpenACC parallel regions
- HiCuda
  - Directives from University of Toronto
- OpenACC
  - New initiative from a group of companies
- OpenHMPP
  - CAPS and Pathscale directives
- OpenMP accelerator extension
  - Being defined by OpenMP ARB
- OMPC
  - OpenMP compilation to Cuda (Purdue)
- OpenMP stream-computing extension
  - Directives from Inria (A. Cohen, A. Pop)
- OMPSs
  - Directives from BSC → task graph oriented
- PGI Accelerator
  - PGI proprietary directives style
- R-Stream
  - Reservoir Lab proprietary  directives
- …

# Main Design Considerations for CAPS

- Focus on the main bottleneck
  - Communication between GPUs and CPUs
- Allow incremental application development
  - Up to full access to the hardware features
- Work with other parallel API (OpenMP, MPI)
  - Do not oppose GPU to CPU,
- Consider multiple languages
  - Avoid asking users to learn a new language
- Consider resource management
  - Generate robust software
- Exploit HWA constructors programming tools
  - Do not replace, complement
- Take into accounts compilers best capabilities

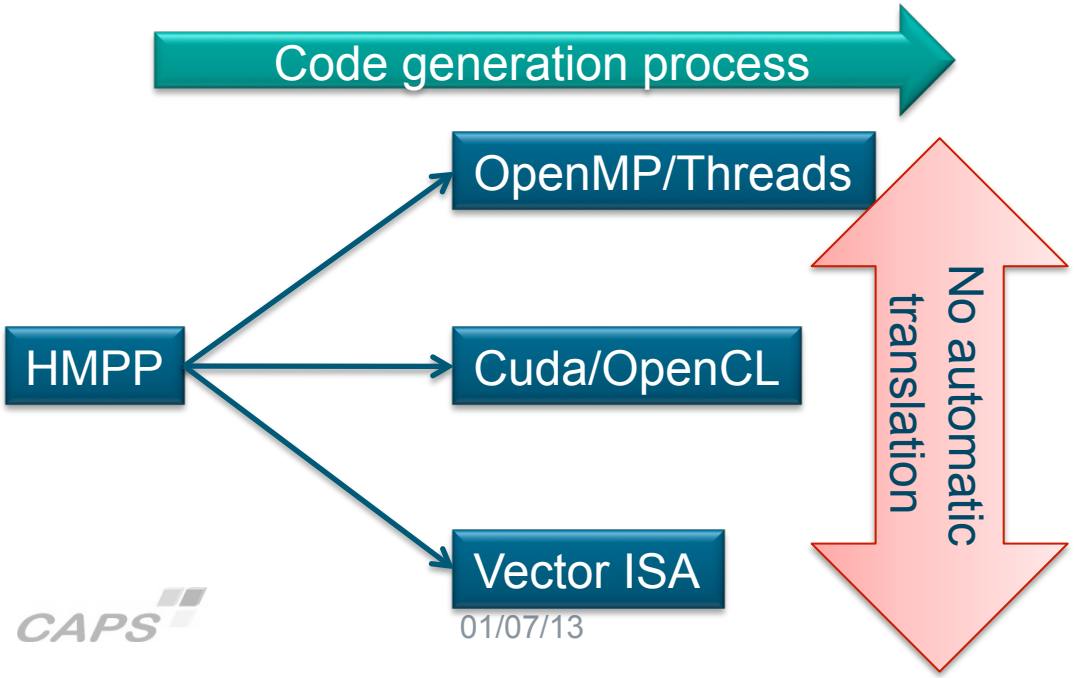
# Limits of Compilers

- Excellent at transforming codes, poor at understanding semantic and making decisions
  - Lack many data anyway
  - Code execution more sensitive to optimization on heterogeneous hardware
- Experts invent strategies, not compilers
  - Look at "3D Finite Difference Computation on GPUs using CUDA" from Paulius Micikevicius, NVIDIA
    - Known code transformations but specific strategy
- Need to provide extra semantic and optimization strategies
  - Specific to each target system and application

# Express Parallelism, not Implementation

- Rely on code generation for implementation details
  - Usually not easy to go from a low level API to another low level one
  - Tuning easier starting from the high level (if not too high)

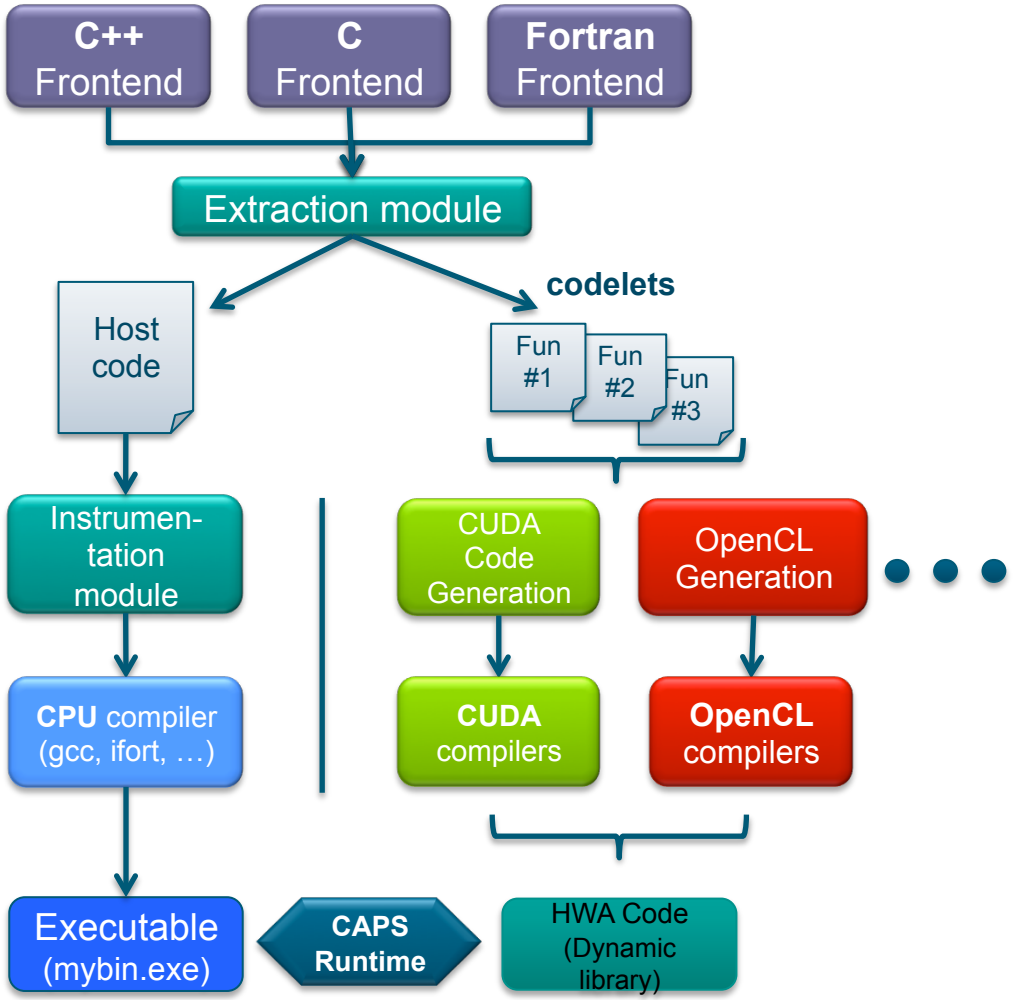
## An example with OpenHMPP




```
#pragma hmppcg gridify(j,i)
#pragma hmppcg unroll(4), jam(2)
  for( j = 0 ; j < p ; j++ ) {
    for( i = 0 ; i < m ; i++ ) {
     for (k = ...) { ...}
      vout[j][i] = alpha * ...;
    }
  }
```

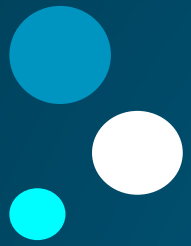
# CAPS Compilers – Source-to-Source

- CAPS Compilers drives all compilation passes
- Host application compilation
  - Calls traditional CPU compilers
  - CAPS Runtime is linked to the host part of the application
- Device code production
  - According to the specified target
  - A dynamic library is built



# Talk Overview

- Accelerator / Co-processor Technology
- OpenACC 1.x and 2.x Directives
- OpenMP 4.0 Accelerator
- OpenACC (and OpenCL) in an Exascale Perspective



# Accelerator / Co-processor Technology

# Accelerator/Coprocessor Architectures

- Many architectures
  - GPU based systems: Nvidia Kepler, AMD APU, ARM Mali, …
  - CPU core based systems: Intel Xeon Phi, Kalray MPPA, …
- SIMT based architecture
  - Performance from vector accesses and plenty of threads
- Cache based architecture
  - Performance from caching and vector instructions
- Different address spaces
  - Distributed or shared (APU and embedded systems)

# Heterogeneous Architectures

- Heterogeneity is
  - Different parallel models
  - Different ISAs
  - Different compilers
  - Different memory systems
  - Different libraries

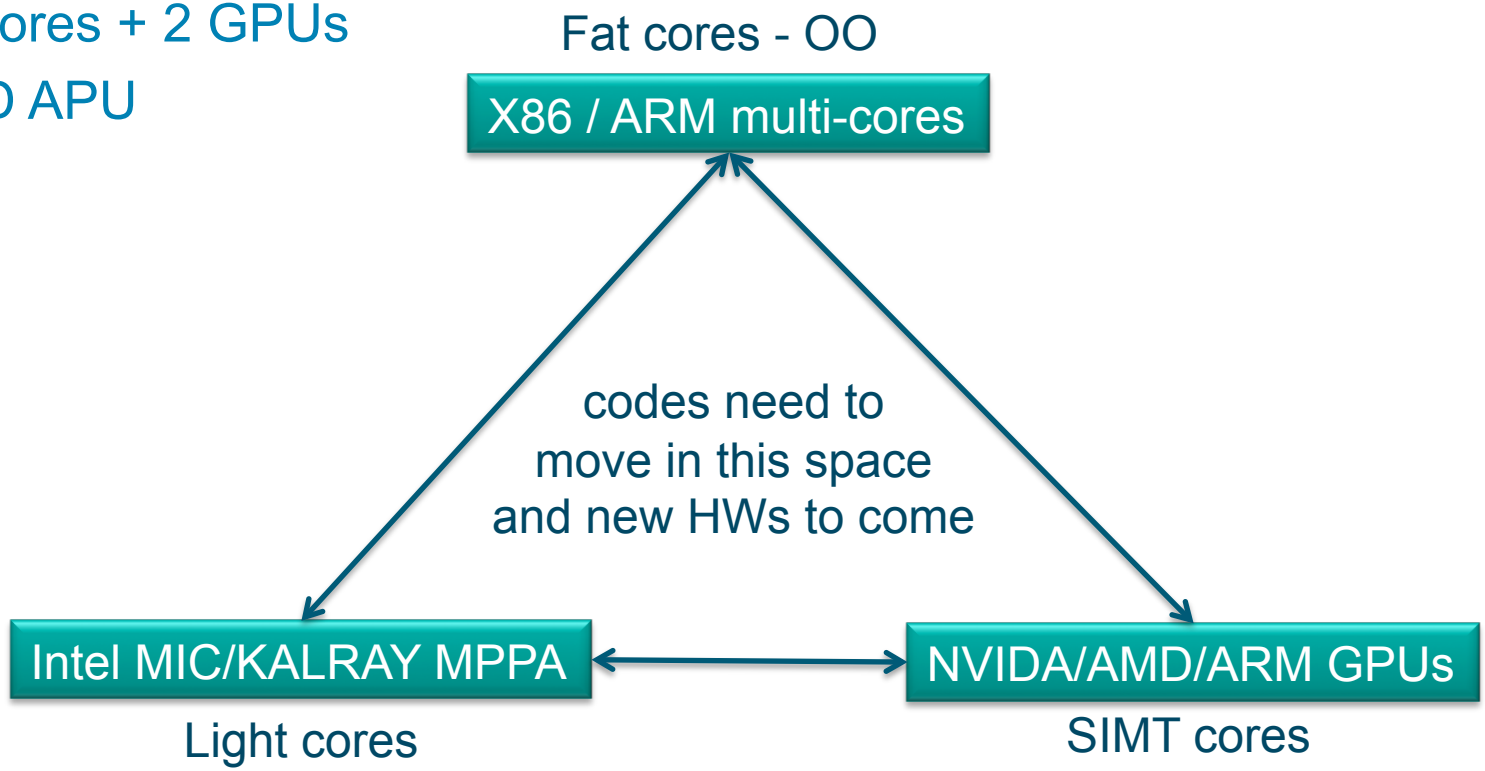
- Performance and code migration very dependant on hardware idiosyncrasies
  - Hardware landscape still very chaotic

# Programming Heterogeneous Model

- Native programming languages
  - CUDA / OpenCL
  - OpenCL available almost everywhere
- Directive based API
  - OpenACC, OpenHMPP, PGI Acc, …
    - Intersection of accelerators capabilities
  - OpenMP accelerator extension in two flavors
    - GPU execution model oriented
    - OpenMP execution model oriented

# Code Writing Constraints

- A code must be written for a set of hardware configurations
  - 6 CPU cores + Intel Xeon Phi
  - 24 CPU cores + AMD GPU / Nvidia GPU / …
  - 12 cores + 2 GPUs
  - AMD APU
  - …



# Compilers and Heterogeneous Hardware

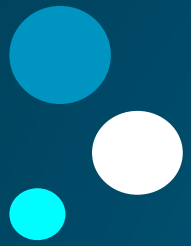
- Compilers are heterogeneous themselves
  - Not one technology fits all
- Want to mix the best compilers to address heterogeneity

CPU compilers
- •Intel compilers
- •IBM compilers
- •ABSoft
- •Pathscale
- •PGI
- •Gcc
- •LLVM
- •Open64
- •…

x86
ARM
MIPS
PowerPC
…

Accelerator compilers
- •Nvidia Cuda compiler
- •Intel OpenCL
- •AMD OpenCL
- •ARM OpenCL
- •Kalray compilers
- •…

x86
PTX
HSA
Kalray MPPA Isa
…



# OpenACC Directives

# OpenACC Initiative


- A CAPS, CRAY, Nvidia and PGI initiative
- Open Standard
- A directive-based approach for programming heterogeneous many-core hardware for C and FORTRAN applications

- http://www.openacc-standard.com

# Parallel Construct

- Starts parallel execution on the accelerator
  - All the region is one accelerator kernel
- Creates gangs/workers/vectors
  - Their numbers remain constant for the parallel region
  - One worker in each gang begins executing the code in the region

```
#pragma acc parallel […]
{
  …
  for(i=0; i < n; i++) {
    for(j=0; j < n; j++) {
      …
    }
  }
  …
}
```

Code executed on the hardware accelerator

# Kernels Construct

- Defines a region of code to be compiled into a sequence of accelerator kernels
  - Typically, each loop nest will be a distinct kernel
  - The number of gangs and workers can be different for each kernel

```
#pragma acc kernels […]
{
  for(i=0; i < n; i++) {
    …
  }
  …
  for(j=0; j < n; j++) {
    …
  }
}
```

```
$!acc kernels […]

  DO i=1,n
    …
  END DO
  …
  DO j=1,n
    …
  END DO

$!acc end kernels
```

1st Kernel

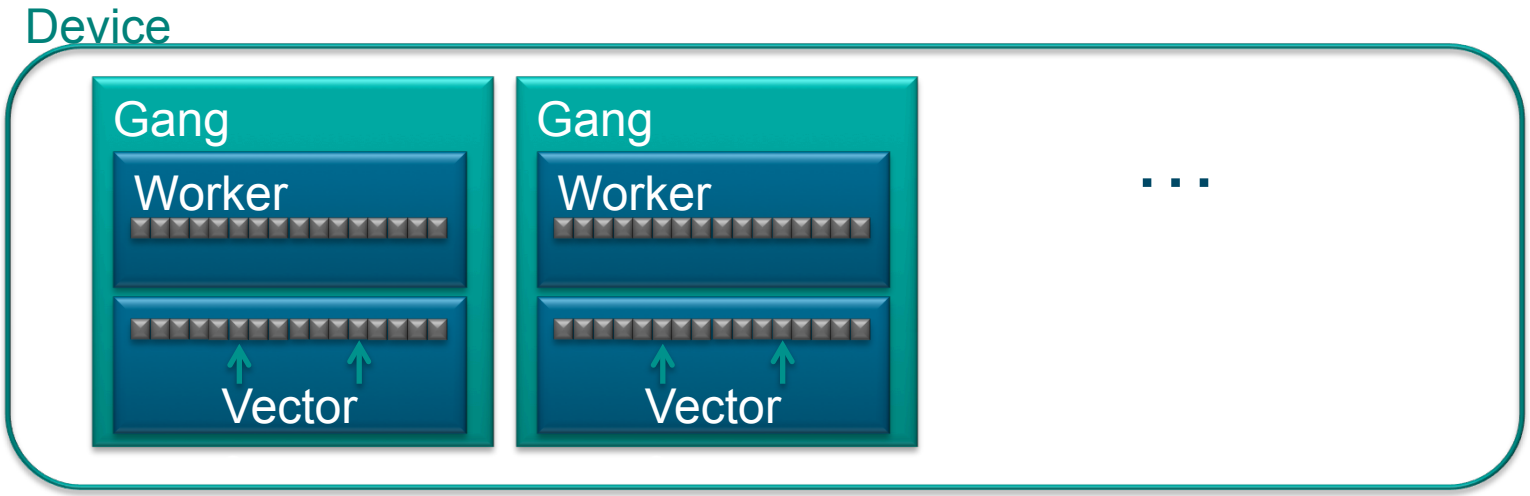
2nd Kernel

# Execution Model

- Among a bulk of computations executed by the CPU, some regions can be offloaded to hardware accelerators
  - Parallel regions
  - Kernels regions
- Host is responsible for
  - Allocating memory space on accelerator
  - Initiating data transfers
  - Launching computations
  - Waiting for completion
  - Deallocating memory space
- Accelerators execute parallel regions
  - Use work-sharing directives
  - Specify level of parallelization

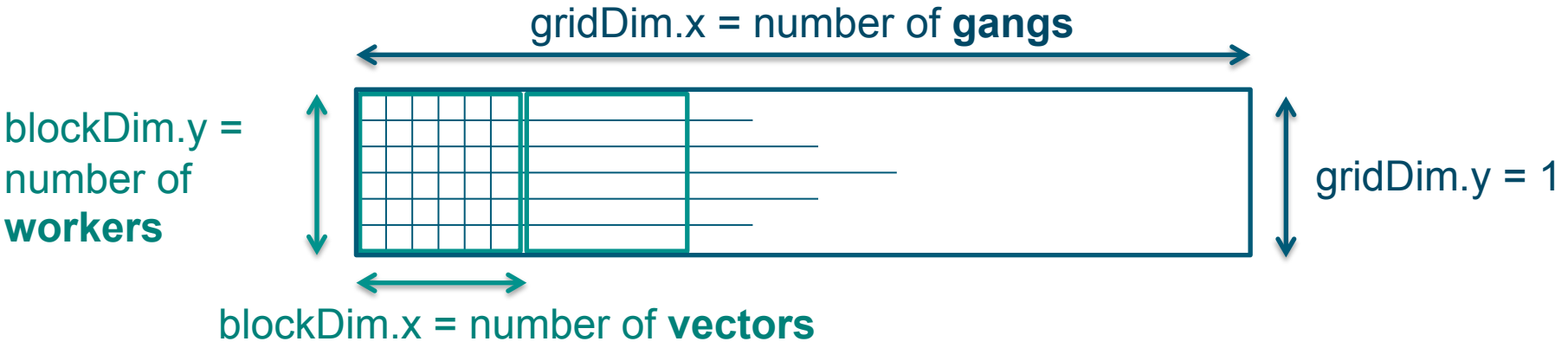
# OpenACC Execution Model

- Host-controlled execution
- Based on three parallelism levels
  - Gangs – coarse grain (e.g. distribution on multiprocessors)
  - Workers – fine grain (e.g. inside a multiprocessor)
  - Vectors – finest grain



# Gangs, Workers, Vectors

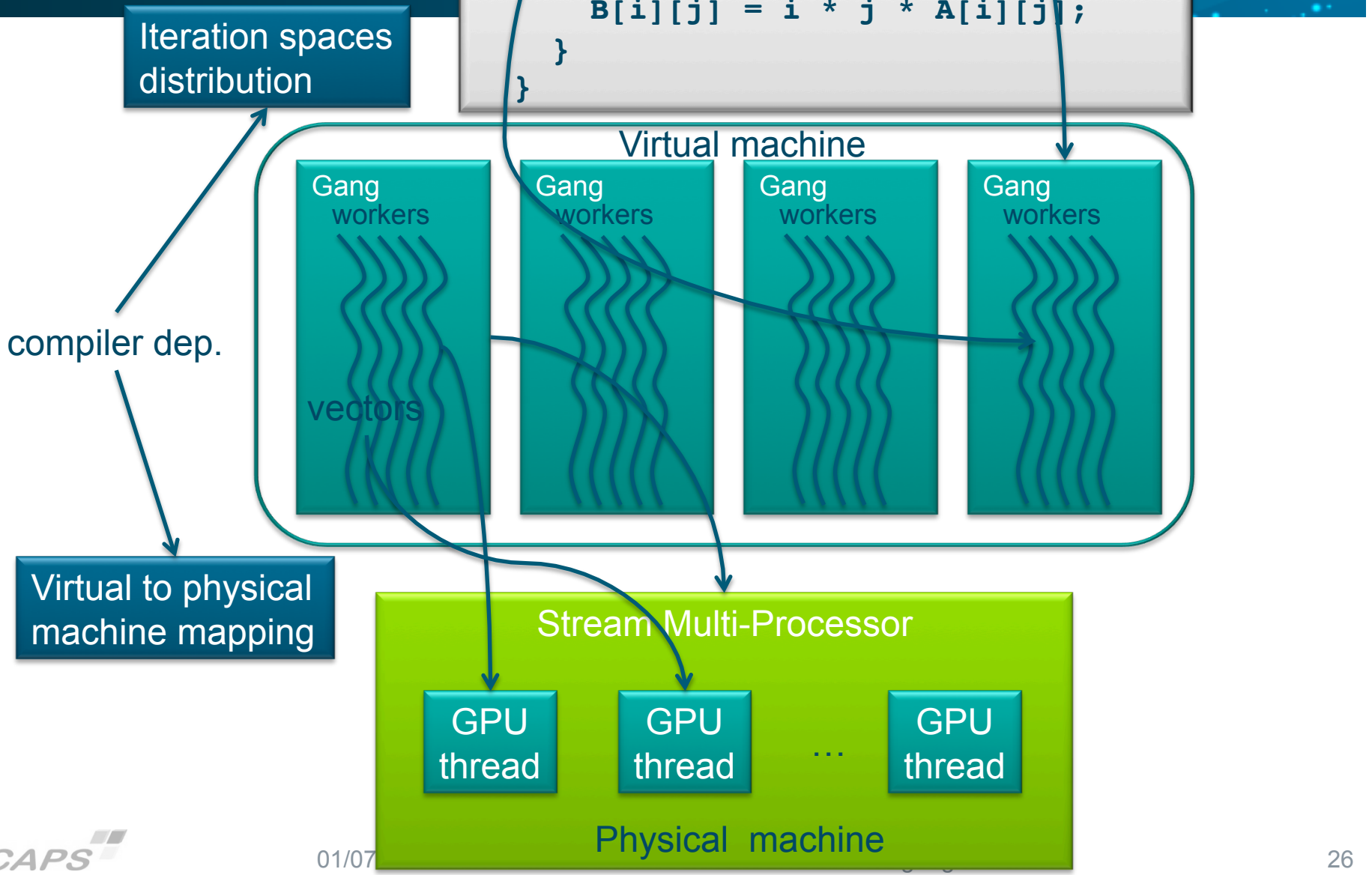
- In CAPS Compilers, gangs, workers and vectors correspond to the following in a CUDA grid




- **Beware:** this implementation is compiler-dependent

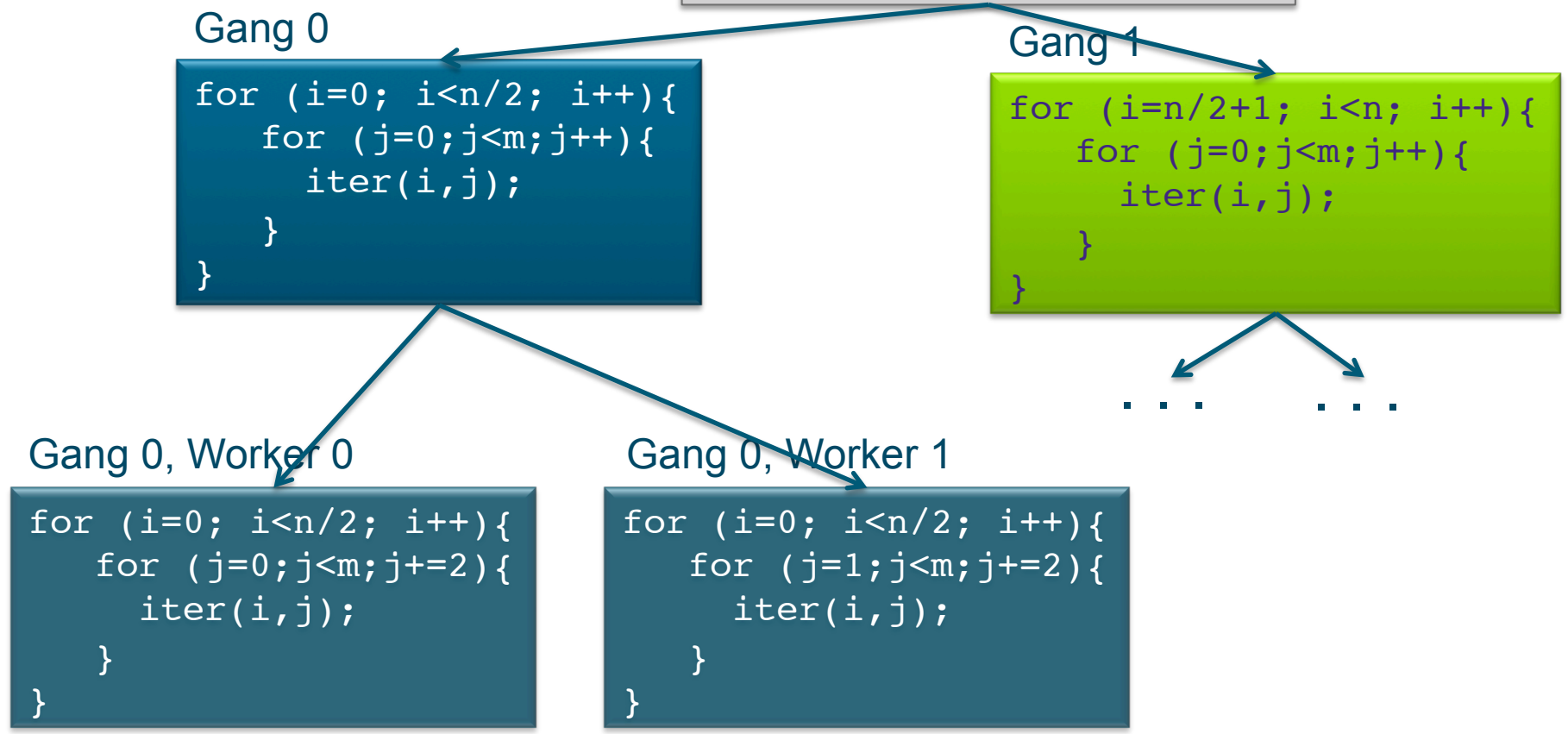
# Code Generation

```
#pragma acc loop gang(NB)
  for (int i = 0; i < n; ++i){
    #pragma acc loop worker(NT)
    for (int j = 0; j < m; ++j){
      B[i][j] = i * j * A[i][j];
    }
}
```



# Iterations Mapping

```
#pragma acc loop gang(2)
for (i=0; i<n; i++){
#pragma acc loop worker(2)
   for (j=0;j<m;j++){
     iter(j,j);
   }
}
```




Distribution scheme is compiler dependant (here simplified scheme)

# Device Memory Reuse

- In this example
  - A and B are allocated and transferred for the first *kernels* region
  - A and C are allocated and transferred for the second *kernels* region
- How to reuse A between the two *kernels* regions?
  - And save transfer and allocation time

```
float A[n];

#pragma acc kernels
{
  for(i=0; i < n; i++) {
    A[i] = B[n - i];
  }
}
…
init(C)
…
#pragma acc kernels
{
  for(i=0; i < n; i++) {
    C[i] += A[i] * alpha;
  }
}
```

# OpenACC Data Regions

- OpenACC data are basically equivalent to HMPP mirrors
  - But managed using data regions instead of standalone directives

```
REAL :: X(100), Y(100)

!$acc data create(Y) copyin(X)
  …

  !$acc parallel

    …  work on X and Y

  !$acc end parallel

  !$acc update host(Y)

!$acc end data
```

- Create Y on the device
- Create and send X to the device
- Implicit DATA region with
  → implied present(X,Y)
- Work on X and Y on the device
- Explicitly receive Y from the device
- Free X and Y on the device

# Memory Allocations

- Avoid data reallocation using the *create* clause
  - It declares variables, arrays or subarrays to be allocated in the device memory
  - No data specified in this clause will be copied between host and device
- The scope of such a clause corresponds to a *data* region
  - *Data* regions are used to define such scopes (as is, they have no effect)
  - They define scalars, arrays and subarrays to be allocated on the device memory for the duration of the region
- *Kernels* and *Parallel* regions implicitly define *data* regions

# Data Presence

- How to tell the compiler that data has already been allocated?
- The *present* clause declares data that are already present on the device
  - Thanks to data region that contains this region of code
- Runtime will find and use the data on device

# Data Construct: Create and Present Clause

Allocation of A of size n on the

Reuse of A already allocated on the device

Reuse of A already allocated on the device

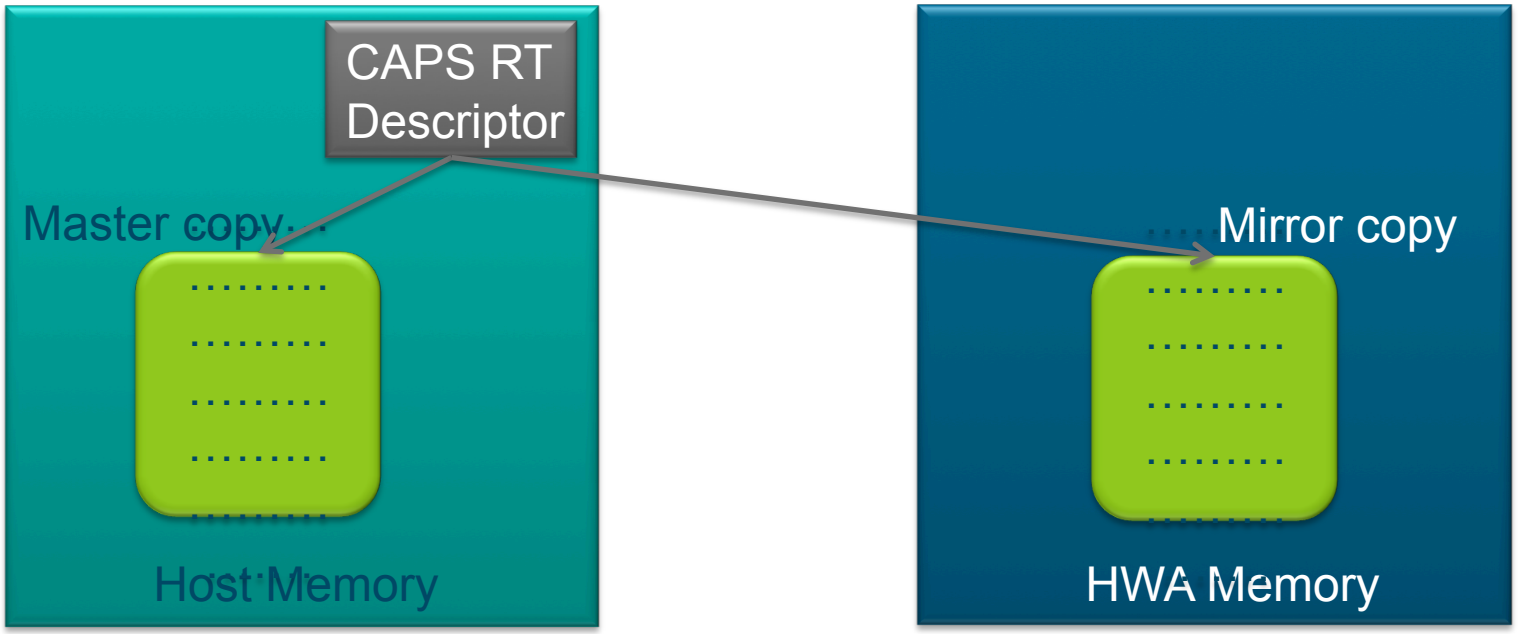
Deallocation of A on the device

```
float A[n];

#pragma acc data create(A)
{
  #pragma acc kernels present(A)
  {
    for(i=0; i < n; i++) {
      A[i] = B[n - i];
    }
  }
  …
  init(C)
  …
  #pragma acc kernels present(A)
  {
    for(i=0; i < n; i++) {
      C[i] += A[i] * alpha;
    }
  }
}
```

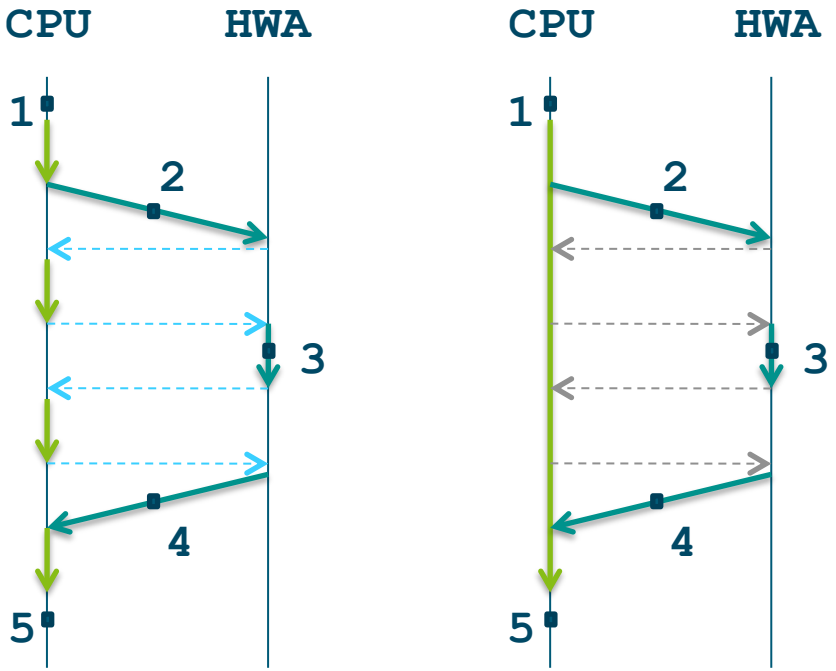
# Data Storage: Mirroring

- How is the data stored in a *data* region?
- A *data* construct defines a section of code where data are mirrored between host and device
- Mirroring duplicates a CPU memory block into the HWA memory
  - The mirror identifier is a CPU memory block address
  - Only one mirror per CPU block
  - Users ensure consistency of copies via directives



# Asynchronism

- By default, the code on the accelerator is synchronous
  - The host waits for completion of the parallel or kernels region
- The *async* clause enables to use the device while the host process continues with the code following the region
- Can be used on *parallel* and *kernels* regions and *update* directives



# Wait Directive

- Causes the program to wait for an asynchronous activity
  - Parallel, kernels regions or update directives
- An identifier can be added to the async clause and wait directive:
  - Host thread will wait for the asynchronous activities with the same ID
- Without any identifier, the host process waits for all asynchronous activities

```
#pragma acc kernels, async
{
  …
}
#pragma acc kernels, async
{
  …
}
#pragma acc wait
```

```
$!acc kernels, async 1
  …
$!acc end kernels
  …
$!acc kernels, async 2
  …
$!acc end kernels
  …
$!acc wait 1
```

# OpenACC 2.0

- OpenACC 2.0 is not officially available
  - A public draft can be downloaded from the OpenACC web site
    - http://www.openacc-standard.org/
  - This is still a work in progress.
  - The features described here show the current state as of April'13
    - Could be slightly different from the latest draft
  - Final version within a few months

# Summary of new features

- **Clarifications of the 1.0 specification & new terminology**
- **New routine directive**
- New device_type clause
- Better asynchronous behavior
- New enter data and exit data directives
- New link clause for the declare directive
- **Loop Tiling**
- Nested parallelism
- Several new API calls
- ...

# Clarifications (1)

- Gang, Worker and Vector shall appear in that order and at most once!
  - Parallel resources are created by the PARALLEL directive
    - worksharing is theoretically possible in all orders
  - But that was confusing for most users (even for advanced ones)
  - Some levels may still be omitted (e.g. gang & vector is still legal)

```
!$acc parallel
 !$acc loop worker
 DO i=1,n
   !$acc loop gang
   DO j=1,m
     A(i,j) = 0
   ENDDO
 ENDDO
!$acc end parallel
```

This code is not legal anymore!

# Clarifications (2)

- Reductions at gang level
  - The reduction clauses on PARALLEL or on LOOP GANG directives are equivalent
  - Each gang computes one partial value.
  - The final reduction occurs after the whole parallel region

```
!$acc parallel
 s = 0
 !$acc loop gang reduction(+:sum)
 DO i=1,n
   s = s + A(i)
 ENDDO
 !$acc loop gang
 DO i=1,n
   B(i) = B(i) + s
 ENDDO
!$acc end parallel
```

This code is not what it seems!

The reduction variable **s** does not contain the whole sum after the first loop

No global synchronization in a parallel region

# New terminology (1)

- The execution model is quite complex with its 3 optional levels of worksharing (gang, worker & vector)
- A new terminology was needed to describe the behavior at all levels of worksharing
  - The program starts in **gang-redundant** mode (GR mode) but enters **gang-partitioned** mode (GP mode) within a loop gang
    - In **GR** mode, all gangs execute the same code
    - In **GP** mode, each gang executes a private subset of the loop iterations
  - The program starts in **worker-single** mode (WS mode) but enters **worker-partitioned** mode (WP mode) within a loop worker
    - In **WS** mode, only one worker is active per gang
    - In **WP** mode, each worker executes a private subset of the loop iterations
  - The program start in **vector-single** mode (VS mode) but enters **vector-partitioned** mode (VP mode) within a loop vector
    - In **VS** mode, only one vector lane is active per gang
    - In **VP** mode, each vector lane executes a private subset of the loop iterations

# Gang-Worker-Vector Terminology Example

```
!$acc parallel private(tmp) num_gangs(16), num_workers(8),
vector_length(32)

    tmp=42                          ! GR+WS+VS

    !$acc loop gang
    DO i=1,n
      A(i) = A(i)+tmp               ! GP+WS+VS
      !$acc loop worker
      DO j=1,m
        B(i,j)=B(i,j)+tmp           ! GP+WP+VS
        !$acc loop vector
        DO k=1,p
          C(i,j,k)=C(i,j,k)         ! GP+WP+VP
        ENDDO
      ENDDO
    ENDDO

!$acc end parallel
```

**GR=Gang-Redundant**
**WS=Worker-Single**
**VS=Vector-Single**

**GP=Gang-Partitioned**
**WP=Worker-Partitioned**
**VP=Vector-Partitioned**

# Example of a Complex Loop Nest Parallelization

- Extract from NOAA Nonhydrostatic Icosahedral Model (NIM)

```
!$acc parallel present(nprox,prox,u,...) vector_length(1) num_workers(64) num_gangs(512)
!$acc loop gang private (rhsu,...) private(ipn,k,isn,...)
do ipn=ips,ipe
  n    = nprox(ipn)
  ipp1 = prox(1,ipn)
 ...
!$acc loop worker vector
  do k=1,nz-1
    rhsu(k,1) = cs(1,ipn)*u(k   ,ipp1)...
    ...
  enddo !k-loop
    k=nz-1
    rhsu(k+1,1) = cs(1,ipn)*u(k   ,ipp1)...
    ...
!$acc loop worker vector   private(wk)
   do k=1,nz
    Lots of statements
   enddo !k-loop
!$acc loop seq
  do isn = 1,nprox(ipn)
!$acc loop worker vector
    do k=1,nz-1
      Tgtu(k,isn) = ...
    enddo !k-loop
    Tgtu(nz,isn) =  2.*Tgtu(nz-1,isn) - ...
  end do  ! isn-loop
(continued on next page)
```

```
(continued from previous page)
!$acc loop seq
  do isn = 1,nprox(ipn)
    isp=mod(isn,nprox(ipn))+1
!$acc loop worker vector
    do k = 2,nz-1
      ...
    end do !  k -loop
    sedgvar( 1,isn,ipn,1)=(zm(1,ipn)...
    ...
   end do   ! isn-loop
!$acc loop worker vector
  do k=1,nz
    kp1=min(nz,k+1)
    ...
  end do
  bedgvar(0,ipn,1)=...
enddo !ipn-loop
!$acc end parallel
```

# The ROUTINE Directive (1)

- Users want to make procedure calls from within ACC regions
- Not officially supported by OpenACC 1.0
  - But implemented by vendors with some constraints (e.g. using inlining)
- The *caller* and the *callee* should agree on the worksharing

```
!$acc parallel
  CALL foo(A)
  !$acc loop gang
  DO i=1,n
    CALL foo(A)
    !$acc loop worker
    DO j=1,m
      CALL foo(A)
    ENDDO
  ENDDO
!$acc end parallel
```

```
SUBROUTINE foo(A)
  REAL    ::
A(1000)
  INTEGER :: k
  !$acc loop
  DO k=1,1000
    A(k) = 0
  ENDDO
END SUBROUTINE foo
```

- Determining worksharing the k-loop?
  - If foo is inlined: easy
  - If foo is not inlined: ?

# The ROUTINE Directive (2)

- Annotate the procedure interface or implementation

```
!$acc routine [clause]
SUBROUTINE foo(A)
  ...
END SUBROUTINE foo
```

- Use one of the clauses gang, worker, vector or seq to control the valid level of worksharing
  - That information is used by both the caller and the caller (should be consistant)
  - If gang then the procedure may contain gang, worker or vector worksharing and is callable from gang-redundant mode (GR)
  - If worker then the procedure may contain worker or vector worksharing and is callable from worker-single mode (WS)
  - If vector then the procedure may contain vector worksharing and is callable from vector-single mode (VS)
  - If seq then the procedure contains no worksharing and is callable from anywhere (i.e. pure sequential)

# The ROUTINE Directive (3)

```
!$acc routine worker
SUBROUTINE foo(A)
  REAL    :: A(1000)
  INTEGER :: k
  !$acc loop worker
  DO k=1,1000
    A(k) = 0
  ENDDO
END SUBROUTINE foo
```

```
!$acc parallel
  CALL foo(A)                  ! valid
  !$acc loop gang
  DO i=1,n
    CALL foo(A)                ! valid
    !$acc loop worker
    DO j=1,m
      CALL foo(A)              ! invalid
    ENDDO
  ENDDO
!$acc end parallel
```

# The ROUTINE Directive (4)

- The BIND clause
  - Change the physical name of the procedure
  - Work as BIND in Fortran but takes a string or an identifier as argument.
  - Can be used in conjunction with the DEVICE_TYPE clause to call hand-written specialized versions (e.g. in CUDA)

```
INTERFACE
  !$acc routine worker dtype(cuda) bind("foo_cuda_worker")
  SUBROUTINE foo(A)
    REAL A(*)
  END SUBROUTINE foo
END INTERFACE
```

# Loop Tiling (1)

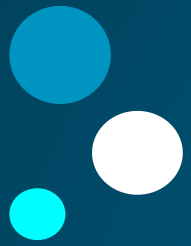
- The new TILE clause on the LOOP directive allows to tile the loop nest before applying worksharing
- Each loop in a tightly nested loop nest is decomposed into
  - An outer tile loop
  - An inner element loop
- If requested, gang worksharing is applied to the collapsed outer tile loops
- If requested, vector worksharing is typically applied to the collapsed inner element loops
- If requested, worker worksharing is applied to
  - the outer tile loops if vector worksharing is also requested
  - or to the inner element loops otherwise

# LOOP Tiling Example (2)

- For simplicity, let's assume that *m* is a multiple of 8 and *n* is a multiple of 12

```
!$acc loop gang worker tile(8,12)
DO i=1,m
 DO j=1,n
  B(i,j)=A(i+1,j)+A(i,j+1)+...
 ENDDO
ENDDO
```

```
!$acc loop gang collapse(2)
DO i0=1,m,8
 DO j0=1,n,12
   !$acc loop worker collapse(2)
   DO i=i0,i0+7
    DO j=j0,j0+11
      B(i,j)=A(i+1,j)+A(i,j
+1)+...
    ENDDO
   ENDDO
 ENDDO
ENDDO
```



# OpenMP Accelerator Directives

# OpenMP Views: Two kinds of architectures

## #1 - The accelerator is just another computer

- e.g. Intel MIC, TI DSPs , …
- It runs a fairly complete Operating System (e.g. Linux, ...)
  - Applications, Threads, Simple Memory Layout, SIMD instructions, …
- Full OpenMP can be or is already implemented on that device

## #2 - The accelerator is designed for performance

- e.g. NVIDIA, AMD, ARM GPUs
- No real operating system but a programming API (e.g. CUDA, OpenCL, …)
  - Kernels, Complex Memory Layout, Coalescing, ...
- OpenMP cannot be fully implemented on the device
  - at least not efficiently

# The TARGET directive

- Specify that a piece of code is executed on the device
- All OpenMP directives shall be allowed within the target code

```
!$omp target
  !$omp parallel num_threads(100)
     !$omp do
     DO i=1,n
       A(i) = compute(i)
     ENDDO
     !$omp barrier
     !$omp do
     DO i=1,n
       B(i) = A(i) + A(n-i-1)
     ENDDO
  !$omp end parallel
!$omp end target
```

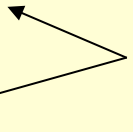
# TARGET - TEAMS - DISTRIBUTE (1)

- The *target* clause does not work well on GPUs
  - OpenMP cannot be fully implemented on the device
- A new level of parallelism was introduced: *team*
  - A team is basically equivalent to a CUDA thread-block or an OpenCL workgroup.
  - A new directive TEAMS  to create a group of teams
    - A team allocates its resources (i.e. max number of threads) when it is created
  - No barriers, atomics, critical sections, … across teams
- A new directive DISTRIBUTE to distribute loop iterations over the current group of teams
  - Similar to the DO and FOR directive

# TARGET - TEAMS - DISTRIBUTE (2)

- TARGET, TEAMS and DISTRIBUTE shall be perfectly nested
  - … in the current proposal! that may change in a later version
  - There is also a combined directive
- Inside the TARGET-TEAM-DISTRIBUTE use OpenMP directives to operate on the threads allocated for the current team.

```
!$omp target …
   !$omp teams num_teams(100) num_threads(16)
     !$omp distribute
     DO i=1,n
       !$omp parallel do num_threads(20)
       DO j=1,m
         A(i,j) = compute(i,j)
       ENDDO
     ENDDO
   !$omp end teams
!$omp end target
```



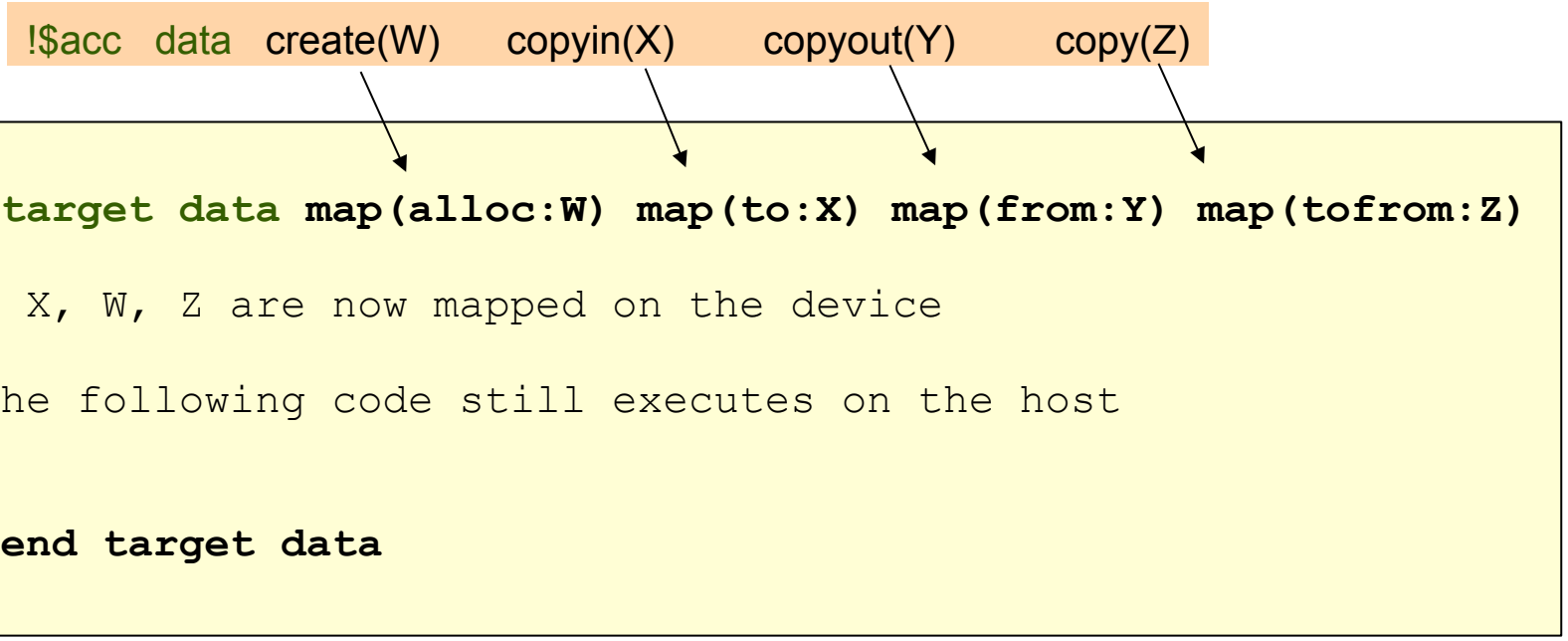
Hoops! There are only 16 threads in the current team

# TEAMS or not TEAMS?

- The TARGET-TEAMS-DISTRIBUTE model can theoretically be implemented on any target with OpenCL support.
  - Will it be implemented on Intel-MIC?
    - Teams are part of the spec so yes … in theory
- The TARGET model cannot be implemented on GPU
  - Could use a single team (i.e. one CUDA block) but that would be inefficient
- If you want portability
  - Use the TARGET-TEAMS-DISTRIBUTE model
- If you want maximum performances on some devices such as the Intel-MIC
  - Use the TARGET model

# Data Management (1)

- Inspired from OpenACC
  - But with a different terminology
- The OMP TARGET DATA construct
  - Allocate and copy data to or from the device
  - Comparable to the ACC DATA construct



# Data Management (2)

- The TARGET UPDATE directive
  - Copy already mapped data to and from the device

```
!$omp target data map(alloc:W,X,Y,Z)
  …
  !$omp target update to(X,Z)
  …
  !$omp target update from(Y)
  …
!$omp end target data
```

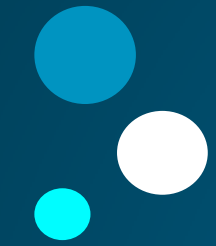
- Partial transfers and mapping are possible:
  - Data must be contiguous as in OpenACC and OpenHMPP

# Vectorization – SIMD

- The SIMD directive
  - Applied to a loop
  - Control the vectorization (SSE, AVX, … )
  - Not specific to accelerators
  - Provide vector worksharing of OpenACC
  - Several clauses:
    - safelen(length)
    - linear(list[:linear-step])
    - aligned(list[:alignment])
    - private(list)
    - lastprivate(list)
    - reduction(operator:list)
    - collapse(n)

# For the rest …

- Use OpenMP constructs and APIs calls  inside the TARGET constructs
  - OMP CRITICAL
  - OMP BARRIER
  - OMP PARALLEL
  - OMP DO
  - OMP TASKS
  - ...
- Is it realistic to implement the whole OpenMP specification on the accelerator?
  - Intel, TI,  and a few other accelerator vendors seem to think so
  - NVIDIA? AMD? …

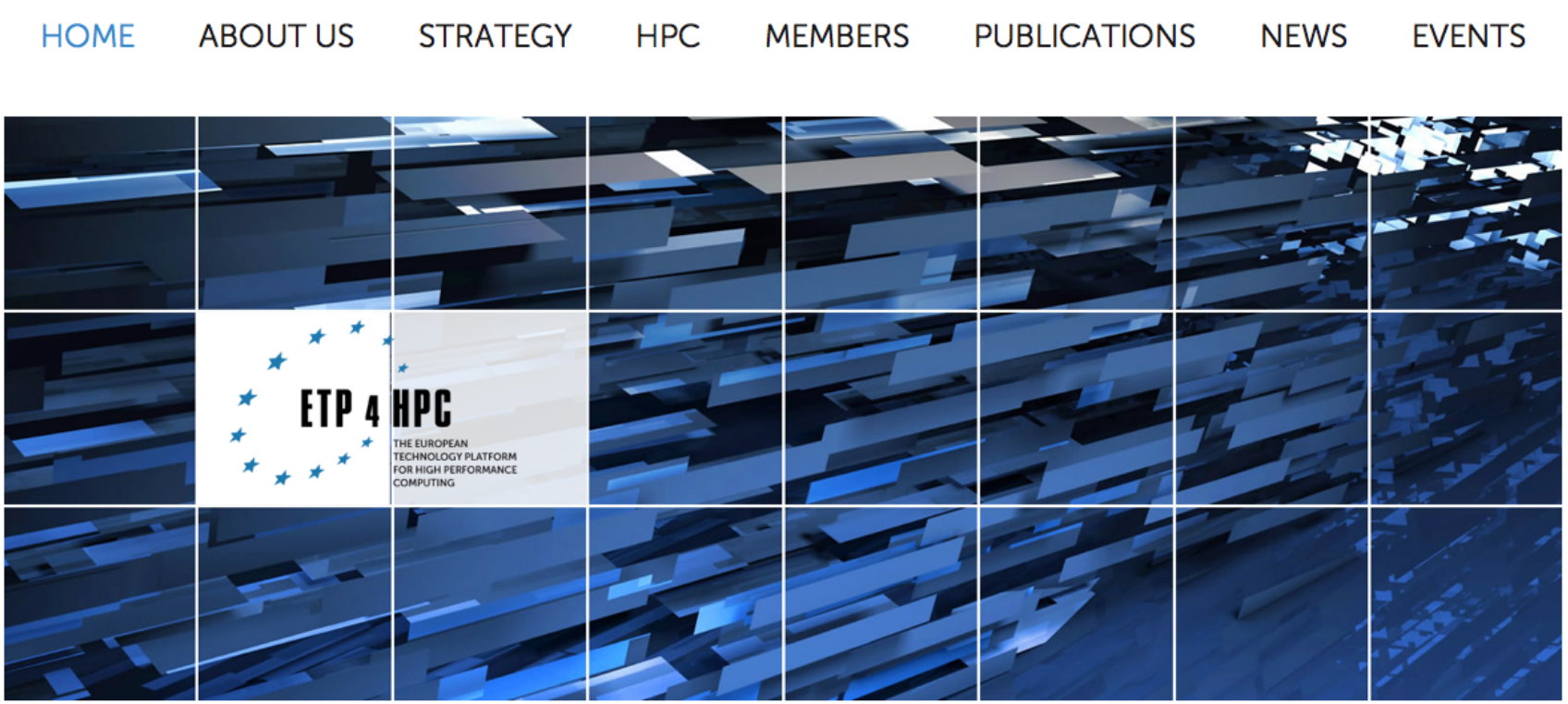


# OpenACC and Extreme Computing

# OpenACC (and OpenCL) in an Exascale Perspective

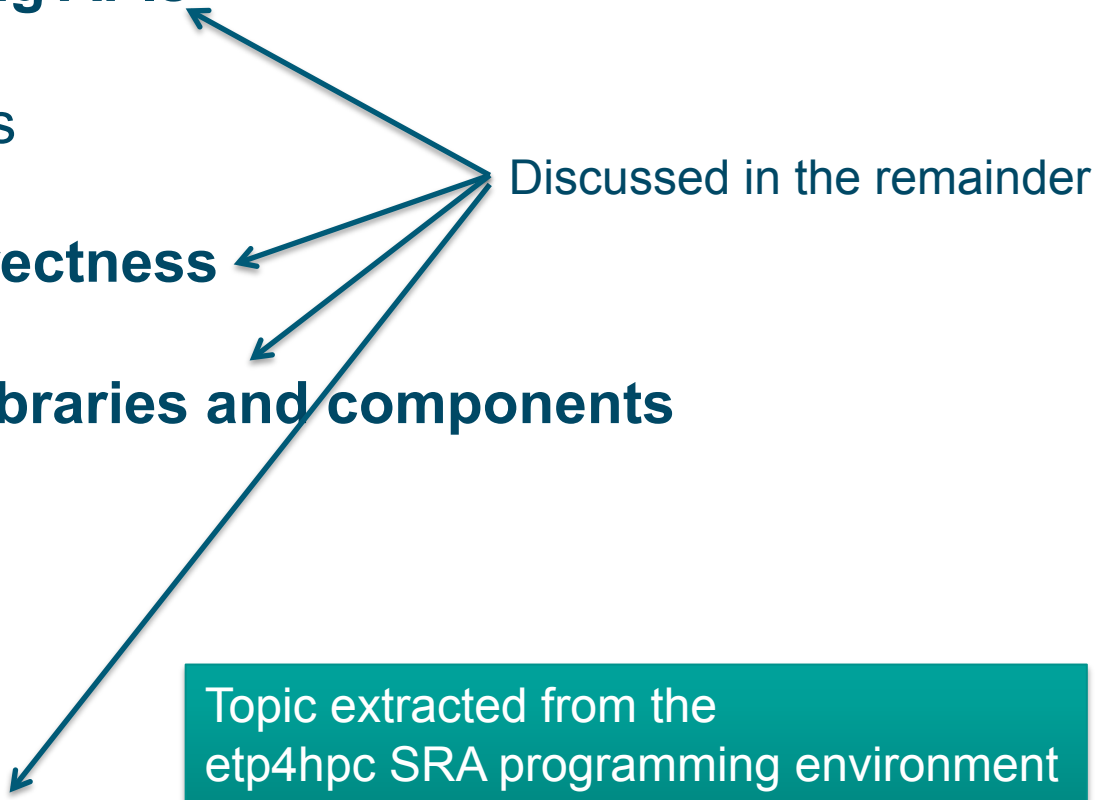
- Exascale architectures may be
  - Massively parallel
  - Heterogeneous compute units
  - Hierarchical memory systems
  - Unreliable
  - Asynchronous
  - Very energy saving oriented
  - …
- Exascale roadmap needs to be build on programming standards
  - Nobody can afford re-writing applications again and again
  - Exascale roadmap, HPC, mass market many-core and embedded systems are sharing many common issues
  - Exascale is not about an heroic technology development
  - Exascale project must provide technology for a large industry base/uses
- OpenACC and OpenCL may be candidates
  - Dealing with inside the node
  - Part of a standardization initiative
  - OpenACC complementary to OpenCL

# http://www.etp4hpc.eu



# Exascale Programming Environment Technological Challenges

- **(1) Parallel programming APIs**
- Runtime support/systems
- **(2) Debugging and correctness**
- **(3) High performance libraries and components**
- Performance tools
- Tools infrastructure
- **(4) Cross cutting issues**




Topic extracted from the
etp4hpc SRA programming environment
http://www.etp4hpc.eu

# (1) Parallel Programming APIs Topics

1. Domain specific languages
2. *API for legacy codes*
3. *MPI + X approaches*
4. Partitioned Global Address Space (PGAS) languages and APIs
5. *Dealing with hardware heterogeneity*
6. Data oriented approaches and languages
7. *Auto-tuning API*
8. *Asynchronous programming models and languages*

# 1-2 API for Legacy Codes Topics

- OpenACC
  - Directive based approaches particularly suited to legacy codes
  - Focused on heterogeneous node
  - Not C only targets also Fortran and C++

- OpenCL
  - Not that convenient for legacy codes
  - Complex to mix with OpenMP
  - Can be used to unify multithreading

# 1-3 MPI + X Approaches

- OpenACC
  - Complementary to MPI
  - Complex to mix with OpenMP, i.e. balancing the load over the CPUs and accelerators
  - OpenACC to deal with threads and accelerator parallelism → but parallelism expression not for all applications

- OpenCL
  - idem

# 1-5 Dealing with Hardware Heterogeneity

- OpenACC
  - Designed for this
  - May simplify code tuning
  - No automatic load balancing over the heterogeneous units, need to be extended
  - Better understanding of the code by the compiler (e.g. exposed data management, parallel nested loops)
    - Provide restructuring capabilities
  - May be extended to consider non volatile memories (NVM)
  - Does not consider multiple accelerators
    - Extension to come
- OpenCL
  - Designed for this
  - Code tuning exposes many low level details
  - Detailed API for resources management
    - Gives many control to users
    - Programming may be complex
  - Interesting parallel model to help vectorization

# 1-7 Auto-tuning API

- Targeting performance portability issues
- What would provide an auto-tuning API?
  - Decision point description
    - e.g. callsite
  - Variants description
    - Abstract syntax trees
    - Execution constraints (e.g. specialized codelets)
  - Execution context
    - Parameter values
    - Hardware target description and allocation
  - Runtime control to select variants or drive runtime code generation
- OpenACC
  - OpenACC gives more opportunity to compilers/automatic tools
  - Can be extended to provide a standard API
  - Many tuning techniques over parallel loops
  - Orthogonal to programming
- OpenCL
  - Can integrate auto-tuning but may be limited in scope
  - OpenCL is low level, guessing high level properties difficult

# 1-8 Asynchronous Programming Models and Languages

- OpenACC
  - Limited asynchronous capabilities, constraints by the host-accelerator model
  - Not suited for data flow approaches, need to be extended (OpenHMPP codelet concept more suitable for this)
- OpenCL
  - idem

# (2) Debugging and Correctness Topics

1. *Debugging heterogeneous/hybrid codes*
2. Static debugging
3. Dynamic debugging
4. Debugging highly asynchronous parallel code at full (Peta-,Exa-)scale
5. Runtime and debugger integration
6. Model aware debugging
7. Automatic techniques

# 2-1 Debugging Heterogeneous/Hybrid Codes

- OpenACC
  - Preserve most of the serial semantic, helps a lot to design debugging tools
  - Helps to distinguish serial bugs from parallel ones
  - Programming can be very incremental, simplifying debugging

- OpenCL
  - Complex debugging due to many low level details and parallel / memory model

# (3) High Performance Libraries and Components Topics

1. Application componentization
2. *Templates/skeleton/component based approaches and languages*
3. *Components / library interoperability*
4. *Self- / auto-tuning libraries and components*
5. New parallel algorithms / parallelization paradigms; e.g. resilient algorithms

# 3-2 Templates/skeleton/component Based Approaches and Languages

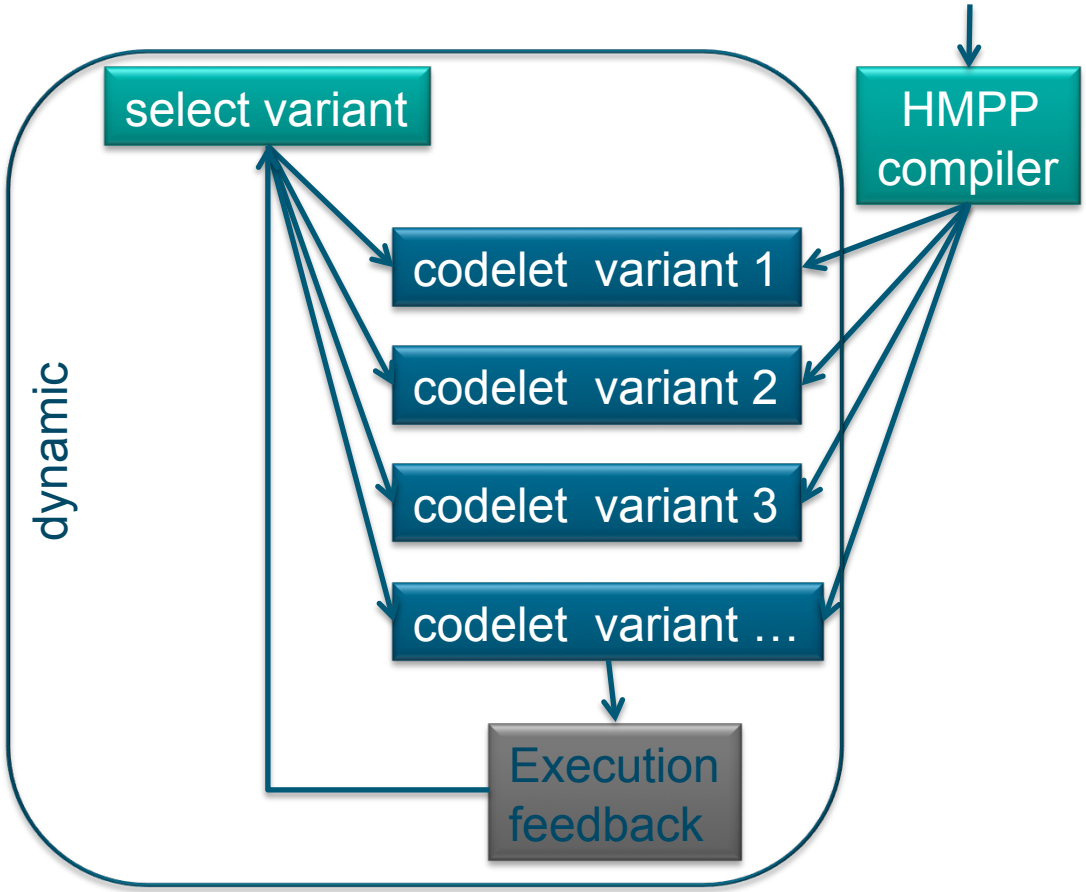
- OpenACC
  - Can be used to write libraries, can exploit already allocated data/HW (pcopy clause)
  - If extended with tuning directives such as hmppcg (e.g. loop transformations) can be used to express templates:
    - Templates to express static code transformations
    - Use runtime technique to tune dynamic parameters such as the number of gangs, workers and vector sizes
- OpenCL
  - Used a lot to write libraries
  - Fits well with C++ components development

# 3-3 Components / Library Interoperability

- Library calls can usually only be partially replaced
  - **Want a unique source code even when using accelerated libraries, CPU version is the reference point**
  - No one-to-one mapping between libraries (e.g.BLAS → Cublas, FFTW → CuFFT)
  - No access to all application codes (i.e. need to keep the CPU library)
- Deal with multiple address spaces / multi-HWA
  - Data location may not be unique (copies, mirrors)
  - Usual library calls assume shared memory
  - Library efficiency depends on updated data location (long term effect)
- OpenACC
  - Needs to interact with users codes, currently limited to sharing the device data ptr
  - Missing automatic data management allocation (e.g. StarPU) to deal with computation migrations (needed to adapt to hardware resources and compute load)
- OpenCL
  - OpenACC and OpenCL have to interact efficiently
  - API can easily be normalized thanks to standardization initiative

# 3-4 Self- / Auto-tuning Libraries and Components

- OpenACC
  - Already provided dynamic parameters for code tuning (e.g. #workers)
  - Need to be extended to allow code templates/ skeletons descriptions
- OpenCL
  - Maybe not the right level, a bit too low level
  - Except for vectorization techniques



# (4) [Cross cutting issues]

1. Standardization initiative
2. *Fault tolerance at programming level*
3. Programming energy consumption control
4. Tools interfaces and public APIs
5. Intellectual property issues
6. Performance portability issues
7. Software engineering, applications and users expectations
8. Tools development strategy
9. *Validation: Benchmarks and other mini-apps*
10. Co-design (hardware - software; applications -programming environment)

# 4-2 Fault Tolerance at Programming Level

- OpenACC
  - OpenACC data region can be extended to mark structures for specific fault tolerance management
  - Extension of the memory model for NVM, etc.

- OpenCL
  - Data management via the API makes it difficult for static tools (e.g. compiler, analyzer)

# 4-9 Validation: Benchmarks and other mini-apps

- OpenACC
  - Extremely important to have exascale good representative measurements
  - Kernels are not enough
  - Tools are usually designed to match benchmark requirement
    - Very influential of the output
  - Mini-apps (e.g. Hydro/Prace, Mantevo) pragmatic and efficient approach
    - But extremely expensive to design
    - Must be production quality
    - Need to exhibit extremely scalable algorithms
  - On the critical path for the foundation of an exascale platform
- OpenCL
  - Idem
  - Limited to C

# Conclusion

- OpenACC/OpenMP provide interesting frameworks for designing an Exascale, non revolutionary, programming environment for heterogeneous systems
  - Leverage existing academic and industrial initiative
  - May be used as a basic infrastructure for higher level approach
  - Mixable with MPI, PGAS, …
  - Available on many hardware targets
  - OpenCL very complementary as a device basic programming layer
- OpenACC and OpenMP technologies are still to evolve a lot as the architecture landscape stabilizes

Accelerator Programming model

Parallelization

**CAPS Workbench**

Directive-based programming

HPC

Portability

OpenCL

Parallel Computing

**CAPS Compilers**

Many-Core programming

NVIDIA CUDA

Code speedup

**OpenHMPP**

GPGPU

*OpenACC*

High Performance Computing

Performance


Visit CAPS Website:
www.caps-entreprise.com