# Programming Heterogeneous Many-cores Using Directives

## HMPP - OpenAcc

Romain Dolbeau, François Bodin, CAPS entreprise

# Introduction

- Programming many-core systems faces the following dilemma
  - The constraint of keeping a unique version of codes, preferably mono-language
    - Reduces maintenance cost
    - Preserves code assets
    - Less sensitive to fast moving hardware targets
    - Codes last several generations of hardware architecture
  - Achieve "portable" performance
    - Multiple forms of parallelism cohabiting
      - Multiple devices (e.g. GPUs) with their own address space
      - Multiple threads inside a device
      - Vector/SIMD parallelism inside a thread
    - Massive parallelism
      - Tens of thousands of threads needed
- For legacy codes, directive-based approach may be an alternative

# Profile of a Legacy Application

- Written in C/C++/Fortran
- Mix of user code and library calls
- Hotspots may or may not be parallel
- Lifetime in 10s of years
- Cannot be fully re-written
- Migration can be risky and mandatory

```
while(many){
   ...
   mylib1(A,B);
   ...
   myuserfunc1(B,A);
   ...
   mylib2(A,B);
   ...
   myuserfunc2(B,A);
   ...
}
```

# Heterogeneous Many-Cores

- Many general purposes cores coupled with a massively parallel accelerator (HWA)

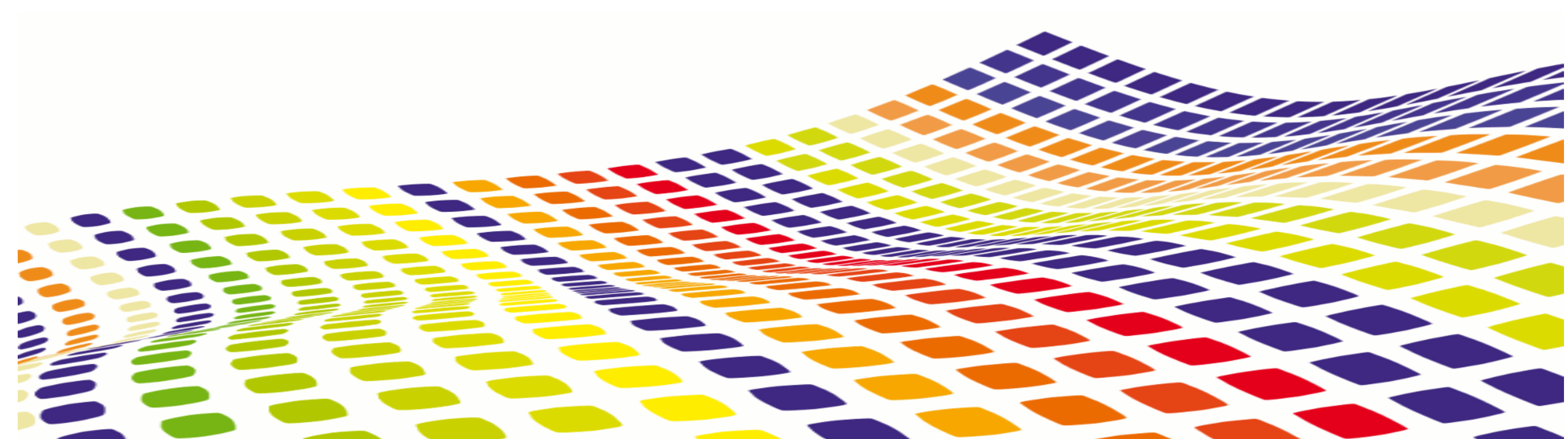
CPU and HWA linked with a PCIx bus

Data/stream/vector parallelism to be exploited by HWA
e.g. CUDA / OpenCL

# Outline of the Presentation

- Usual parallel programming techniques in HPC
- Many-core architecture space
- Example code description
- Directives-based programming
- Going further with OpenHMPP

# Usual Parallel Programming Techniques in HPC



# Styles of Parallel Programming

- Task Parallel
  - Thread based
  - Mostly shared memory

➔ OpenMP directive based API

- Data Parallelism

➔ SIMD / SIMT

  - Same computation applied to all elements of a collection
  - Data distribution over the processors
  - Frequently exploit loop-level parallelism

- Message Passing

➔ MPI Library

  - Mostly distributed memory
  - Frequently use data distribution techniques

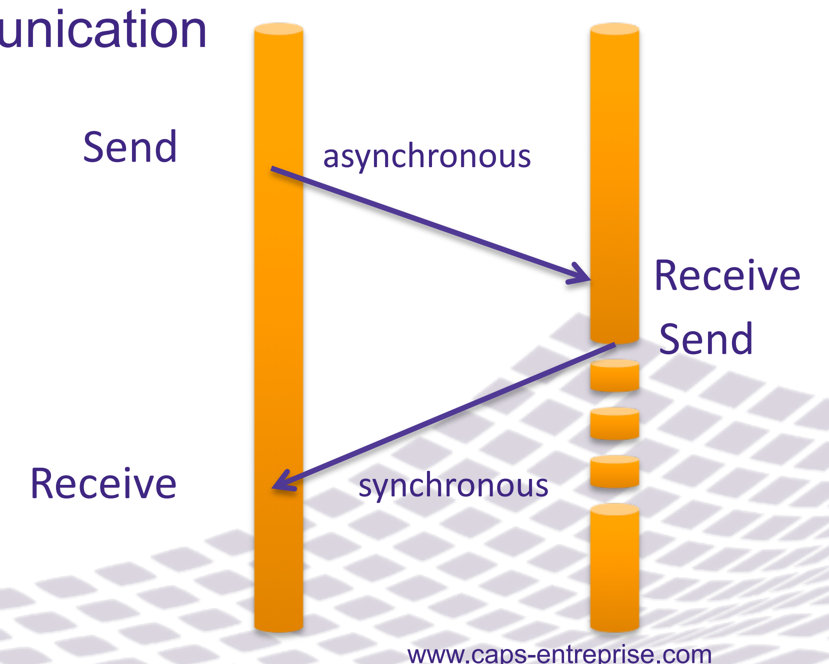
# Task Based Example: Threads Fork and Join

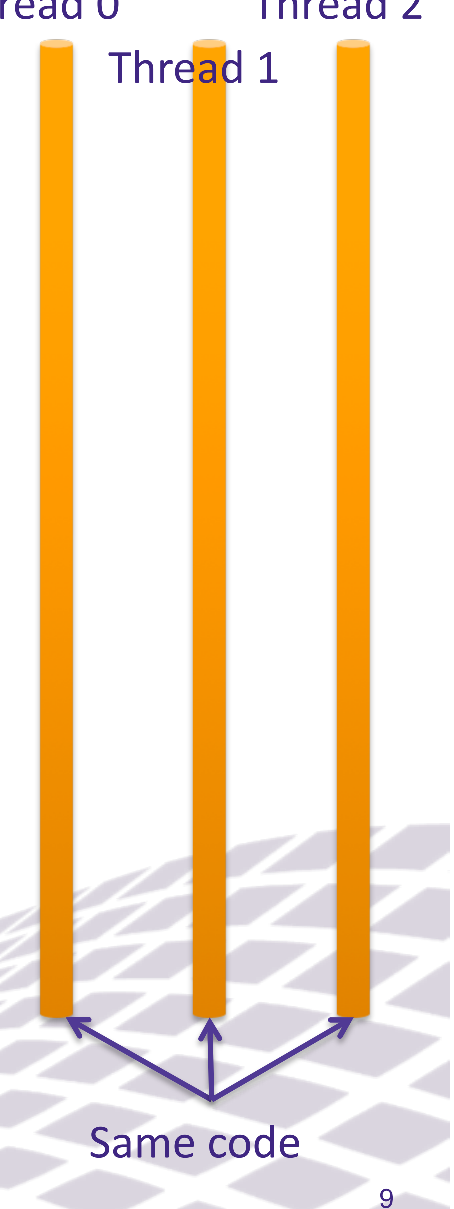
- The whole the programming environment assumes some level of thread managements
  - Gang scheduling
  - Thread / data affinity / migration
- For instance OpenMP Parallel Loops
  - Express distribution of loop iterations over the threads
  - Various scheduling clauses

```
#pragma omp parallel for schedule(static, 3)
  for (i=0; i<N; i++){
      dowork(i);
}
```

# Single Program Multiple Data (SPMD)

- This is the dominant form of parallelism in scientific computing
  - No thread creation overhead
  - Simple
  - Shared or distributed memory architecture
- MPI message passing based on this model
  - Explicit control of all inter-processor communication



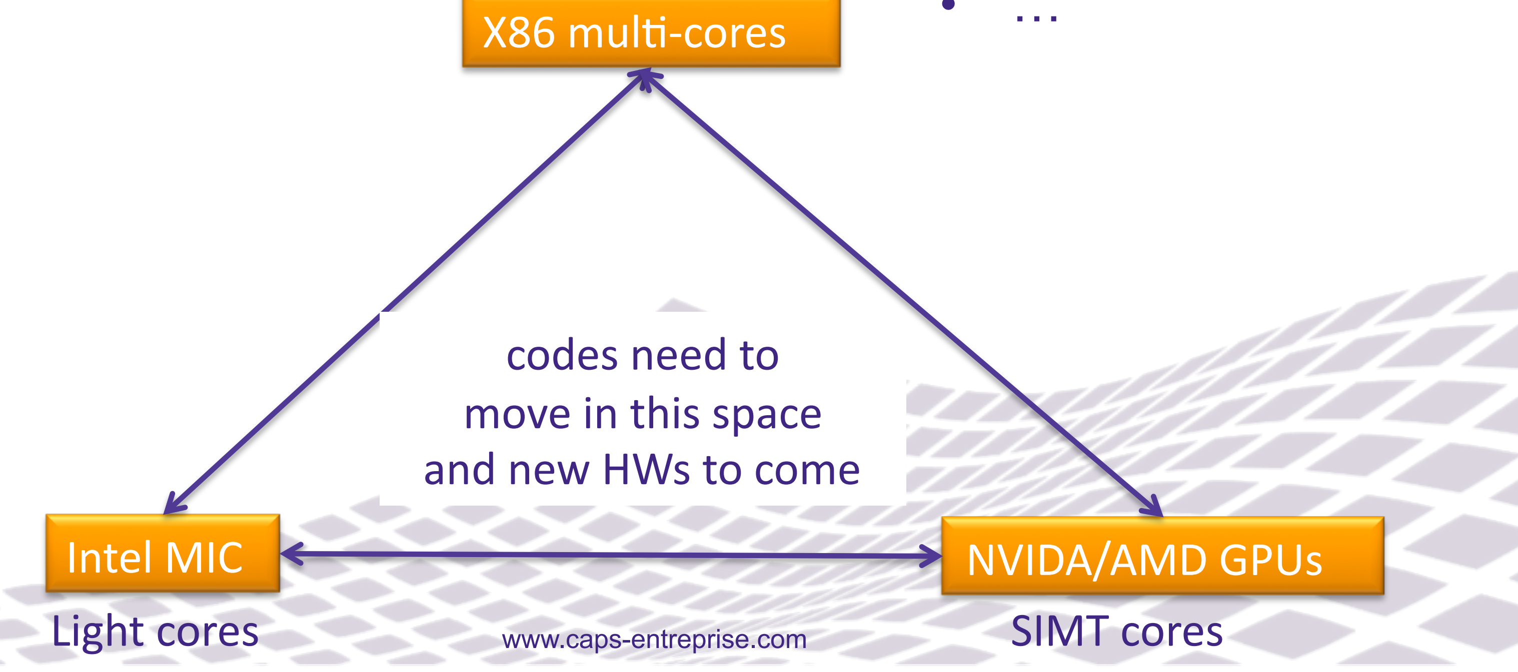
# Many-Core Architecture Space

# Heterogeneous Architecture Space

- Achieving "portable" performance across architecture space

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

- A code must be written for a set of hardware configurations
  - 6 CPU cores + MIC
  - 24 CPU cores + GPU
  - 12 cores + 2 GPUs
  - …



# Where Are We Going?

GHz: 4 *, 3,5, 3, 2,5, 2, 1,5, 1, 0,5, 0

Processing Units: 10000**, 1000, 100, 10, 1

2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015

CUDA/NVIDIA
Tesla release

forecast

Specialized
Manycores (GPGPU)

Frequency based
Performance
Improvement Era

Manycores processors
reaching the general
purpose market

Non migrated applications
do not scale up

Clock Frequency
CPU
GPU

* Frequency based on
Intel Processor (max.)

** Number of
Processing Units
( ~cores x threads)
Intel CPU, NVIDIA GPU

# Usual Parallel Programming Won't Work Per Se

- Exploiting heterogeneous many-core with MPI parallel processes
  - Extra latency compared to shared memory use
    - MPI implies some copying required by its semantics (even if efficient MPI implementations tend to reduce them)
    - Cache trashing between MPI processes
  - Excessive memory utilization
    - Partitioning for separate address spaces requires replication of parts of the data
    - When using domain decomposition, the sub-grid size may be so small that most points are replicated (i.e. ghost cells)
    - Memory replication implies more stress on the memory bandwidth which finally prevent scaling
- Exploiting heterogeneous many-core with thread based APIs
  - Data locality and affinity management non trivial
  - Reaching a tradeoff between vector parallelism (e.g. using the AVX instruction set), thread parallelism and MPI parallelism
  - Threads granularity has to be tuned depending on the core characteristics (e.g. SMT, heterogeneity)
  - Most APIs are shared memory oriented

# Domain Decomposition Parallelism

32x32x32 cell domain
ghost cells 2 ←→
ghost cells / domain cells = **0.42**

1 process → 8 processes

16x16x16 cell domain
ghost cells 2 ←→
ghost cells / domain cells = **0.95**

# Flexible Code Generation Required

- The parallel programming API must not assume too much about the HW targets

Cluster Level APIs
MPI, PGAS
…

Threads APIs
OpenMP
Intel TBB, Cilk, …

Accelerator Directives
HMPP, OpenACC

Accelerator Languages
CUDA, OpenCL

X86 multi-core

Intel MIC

NVIDIA/AMD GPU

# Auto-Tuning is Required to Achieve Some Performance *Portability*

- The more optimized a code is, the less portable it is
  - Optimized code tends to saturate some hardware resources
  - Parallelism ROI varies a lot
    - i.e. # threads and workload need to be tuned
  - Many HW resources not virtualized on HWA (e.g. registers, #threads)

performance

HW1

HW2

cores

Threads

1

0,8

0,6

0,4

0,2

0

Occupancy

Registers/threads

Mem. Throughput

L1 Hit Ratio

Run 1 norm

Run 2 norm

Example of an optimized versus a non optimized stencil code

# Example Code Description

# HydroC Code

- HydroC* is a *summary* from RAMSES
  - Used to study large scale structure and galaxy formation.
  - Includes classical algorithms we can find in many applications codes for Tier-0 systems
  - Solves compressible Euler equations of hydrodynamics, based on finite volume numerical method using a second order Godunov scheme for Euler equations
  - **The algorithms have not been modified**
  - ~1500 LoC, two versions, Fortran and C, MPI
- GNU Compiler 4.4.5, MPICH, NV SDK 4.1, CAPS OpenACC, compiler flag -O3
- More at
  - http://irfu.cea.fr/Phocea/Vie_des_labos/Ast/ast_sstechnique.php?id_ast=904
  - http://hipacc.ucsc.edu/html/HIPACCLectures/lecture_hydro.pdf
  - http://calcul.math.cnrs.fr/Documents/Manifestations/CIRA2011/IDRIS_io_lyon2011.pdf

*Pierre-François Lavallée[a], Guillaume Colin de Verdière[b], Philippe Wautelet[a], Dimitri Lecas[a], Jean-Michel Dupays[a]
[a]*IDRIS/CNRS*, [b]*CEA,Centre DAM*

# HydroC Migration Steps

**Hotspots**

- Understand your **performance goal** (analysis, definition and achievement)
- Know your **hotspots** (analysis, code reorganization, hotspot selection)
- Establish a **validation process**
- Set a **continuous integration process** with the validation

**Hours to Days**

**Parallelization**

- **Optimize** CPU code
- **Exhibit** application SIMT parallelism
- Parallelize for **Many-core**
- **Validate execution**

**Days to Weeks**

**Define your Parallel Project**

**Port your Application on Many-core**

**Phase 1**

**Many-core operational application with known potential**

**Phase 2**

**Optimize Your Many-core Application**

**Tuning**

- Reduce **data transfers**
- **Optimize** kernel execution
- Provide **feedback to application programmers** for improving algorithm data structures/…
- Consider **multiple devices**

**Weeks to Months**

**A corporate project**

- Purchasing Department
- Scientists
- IT Department

# Migration Process, Tool View

CPU Code Analysis → GPU Programming → GPU Code Analysis

- CPU Code Analysis → CPU Profiler
- GPU Programming → OpenHMPP/OpenACC → Accelerated Libraries → Proxy to CuFFT, CuBlas, Cula, ArrayFire, etc.
- GPU Code Analysis → Kernel performance → CPU-GPU data transfers → Tracing visualization tool

# HydroC Call-Graph

# Directive-based Programming

# Directives-based Approaches

- Supplement an existing serial language with directives to express parallelism and data management
  - Preserves code basis (e.g. C, Fortran) and serial semantic
  - Competitive with code written in the device dialect (e.g. CUDA)
  - Incremental approach to many-core programming
  - Mainly targets legacy codes

- Many variants
  - OpenHMPP
  - PGI Accelerator
  - OpenACC
  - OpenMP Accelerator extension
  - …

- OpenACC is a new initiative by CAPS, CRAY, PGI and NVidia
  - A first common subset presented at SC11

# How Does Directive Based Approach Differ from CUDA or OpenCL?

- HMPP/OpenACC parallel programming model is **parallel loop centric**
- CUDA and OpenCL parallel programming models are **thread centric**

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

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

int nblocks = (n + 255) / 256;
saxpy_cuda<<<nblocks,
256>>>(n, 2.0, x, y);
```

# Two Styles of Directives in CAPS Workbench

- Functions based, i.e. codelet (OpenHMPP)
- Code regions based (OpenACC)

```
#pragma hmpp myfunc codelet, …
void saxpy(int n, float alpha, float x[n], float y[n])
{
  #pragma hmppcg gridify(i)
  for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```

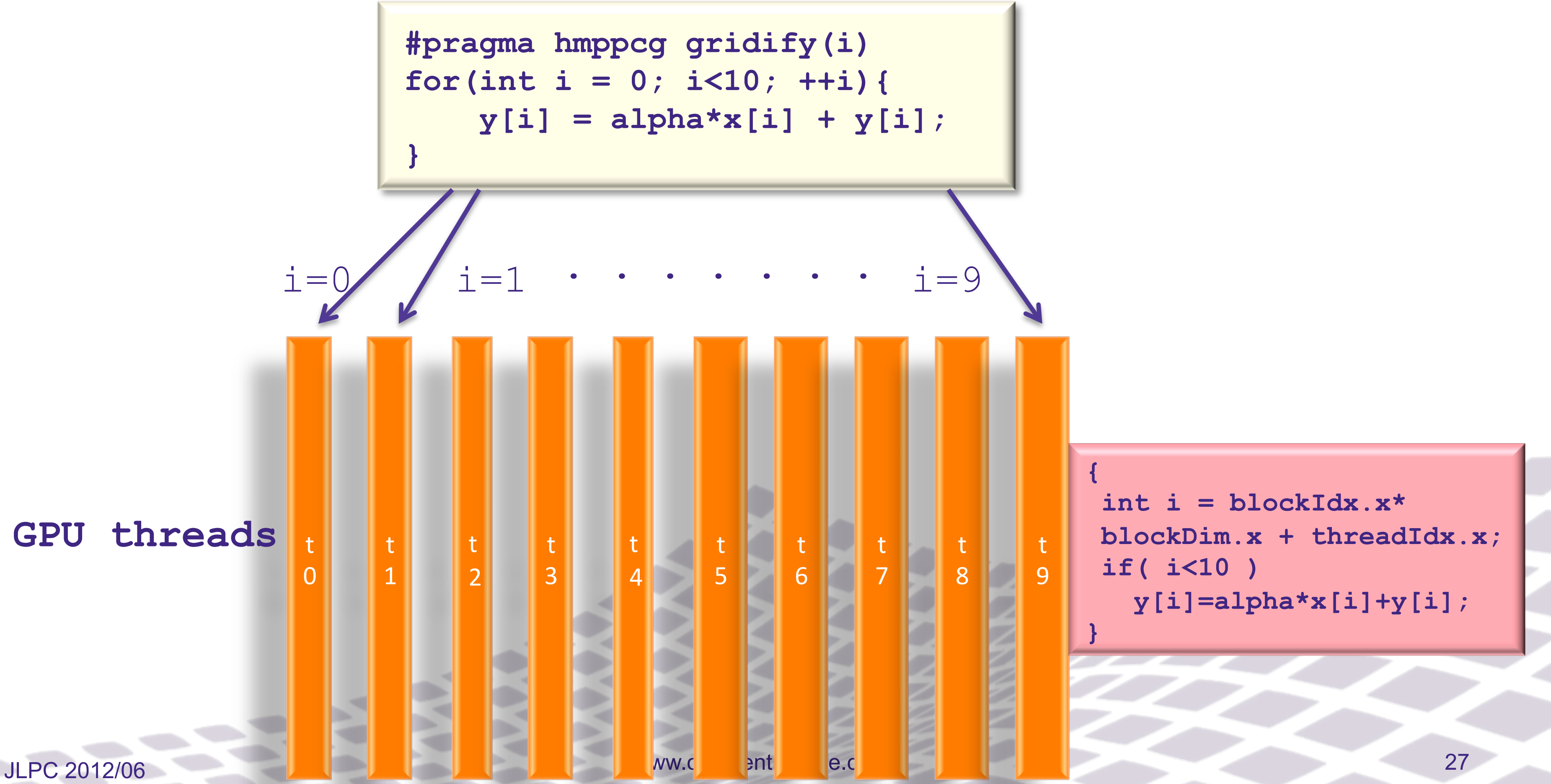
```
#pragma acc kernels …
{
  for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```

# HMPP/OpenACC Compiler flow (Src to Src Cpler)

```
$ hmpp gcc filetocompile.c –o mybinary.exe
```

# Code Generation based on "Loop Nest *Gridification*"

- The loop nest *gridification* process converts parallel loop nests in a grid of GPU/HWA threads
  - Use the parallel loop nest iteration space to produce the threads



# Remote Procedure Call on an Accelerator

- Running on the accelerator is implemented using some form of RPC (Remote Procedure Call)

```
#pragma hmpp f1 codelet
myfunc(...){
 ...
 for()
  for()
   for()
    ...
 ...
}
```

```
main(){
 ...
#pragma hmpp f1 callsite
  myfunc(V1[k],V2[k]);
 ...
}
```

HMPP COMPILER

CPU version

GPU version

Data gridification

Multi-threading

Message passing

core core core

core core core

CPU

socket

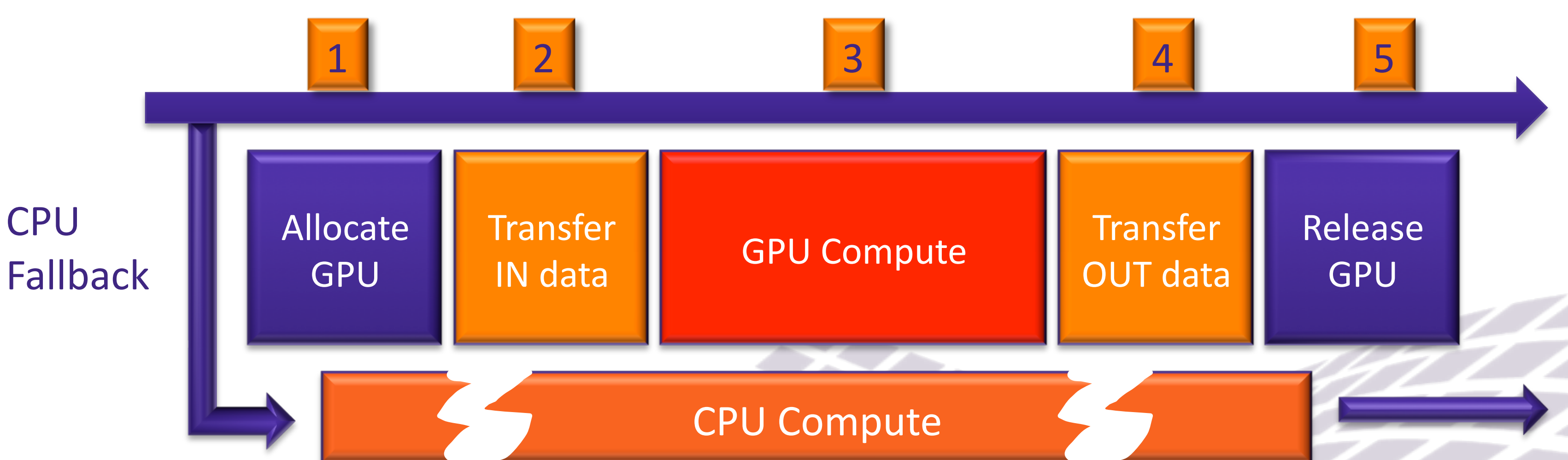
GPU

GPU

Execute on GPU (RPC)

# Remote Procedure Call Basic Operations

- A RPC sequence consists in 5 basic steps:
  (1) Allocate the GPU and the memory
  (2) Transfer the input data: CPU => GPU
  (3) Compute
  (4) Transfer the output data: CPU <= GPU
  (5) Release the GPU and the memory



# RPC Example (Codelet Style)

Upload GPU data

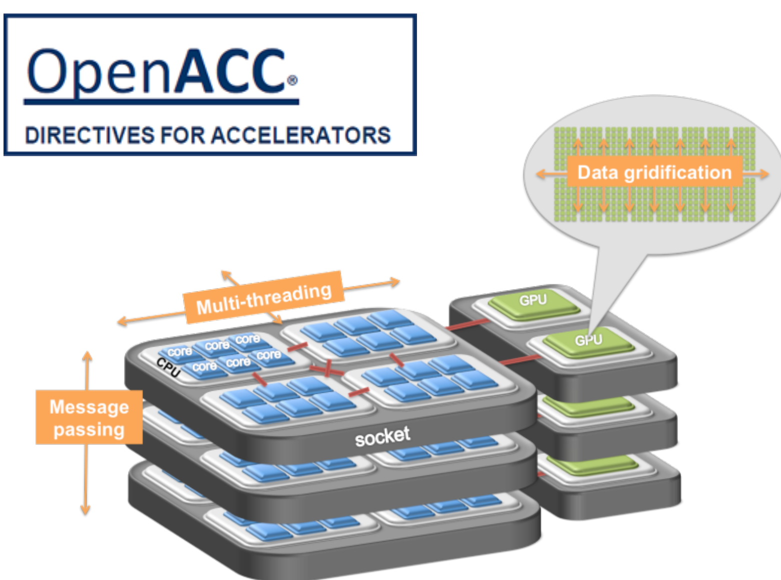
Execution on the GPU

```
#pragma hmpp sobelfilter advancedload,data["original_image"...

  for( i=0; i<NB_RUNS; i++ ){
    #pragma hmpp sobelfilter callsite
    sobelFilter( original_image, edge_image, size, size );
  }
#pragma hmpp sobelfilter delegatedstore, data ["edge_image"]
```

Download GPU data

# OpenACC Initiative

- Express data and computations to be executed on an accelerator
  - Using marked code regions




- Main OpenACC constructs
  - Parallel and kernel regions
  - Parallel loops
  - Data regions
  - Runtime API

- OpenACC support released in April 2012 (HMPP Workbench 3.1)
  - OpenACC Test Suite provided by University of Houston

- Visit http://www.openacc-standard.com for more information

# OpenACC Data Management

- Mirroring duplicates a CPU memory block into the HWA memory
  - Mirror identifier is a CPU memory block address
  - Only one mirror per CPU block
  - Users ensure consistency of copies via directives

HMPP RT Descriptor

Master copy

Mirror copy

CPU Memory

HWA Memory

# OpenACC Execution Model

- Host-controlled execution
- Based on three parallelism levels
  - Gangs – coarse grain
  - Workers – fine grain
  - Vectors – finest grain

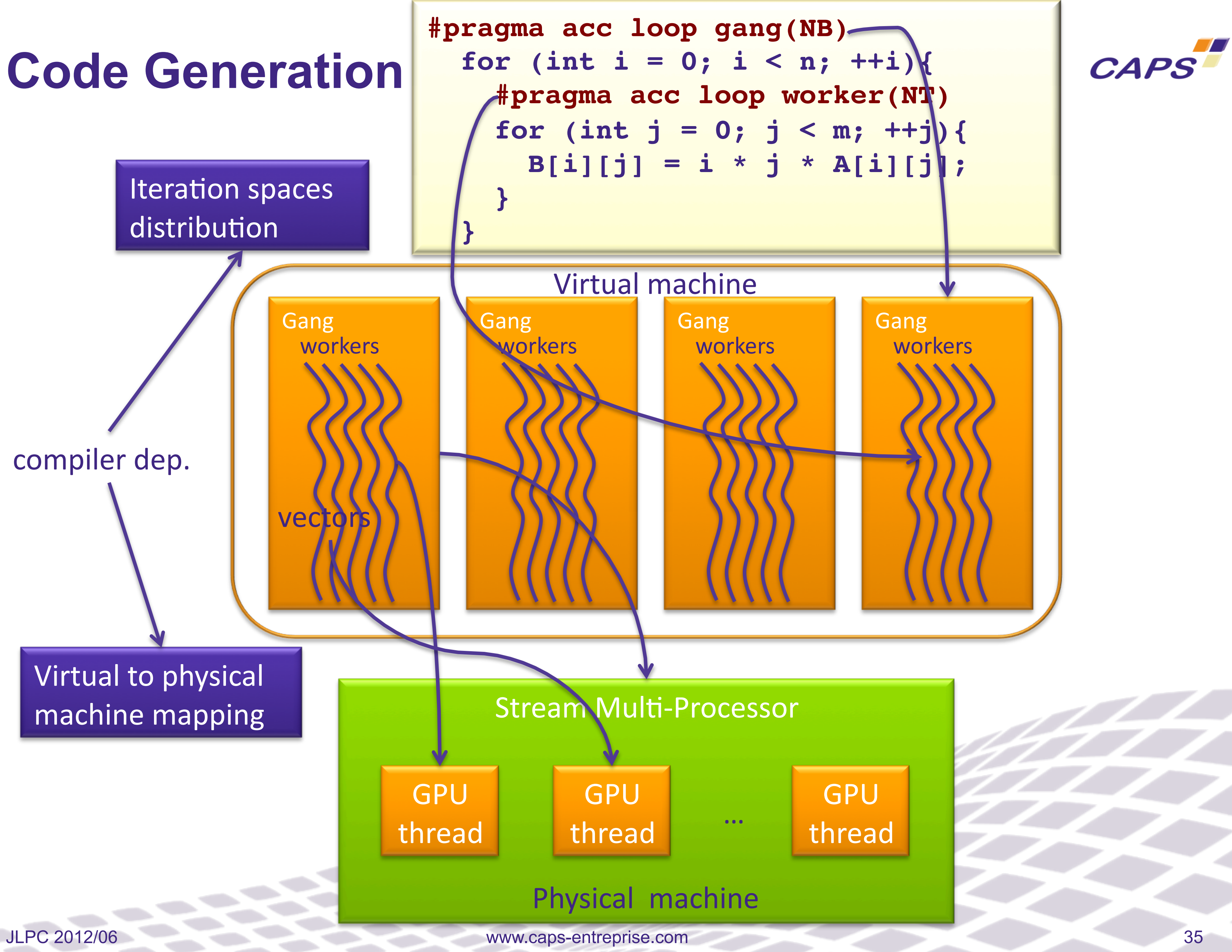
# Parallel Loops

- The loop directive describes iteration space partitioning to execute the loop; declares loop-private variables and arrays, and reduction operations
- Clauses
  - `gang [(scalar-integer-expression)]`
  - `worker [(scalar-integer-expression)]`
  - `vector [(scalar-integer-expression)]`
  - `collapse(n)`
  - `seq`
  - `independent`
  - `private(list)`
  - `reduction(operator:list )`

```
#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];
    }
  }
```

Iteration space distributed over NB gangs

Iteration space distributed over NT workers


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];
}
}
CAPS
Iteration spaces distribution
Virtual machine
Gang workers
Gang workers
Gang workers
Gang workers
vectors
compiler dep.
Virtual to physical machine mapping
Stream Multi-Processor
GPU thread
GPU thread
...
GPU thread
Physical machine

# Kernel Regions

- Parallel loops inside a region are transformed into accelerator kernels (e.g. CUDA kernels)
  - Each loop nest can have different values for gang and worker numbers
- Clauses
  - if(condition)
  - async[(scalar-integer-ex
  - copy(list)
  - copyin(list)
  - copyout(list)
  - create(list)
  - present(list)
  - present_or_copy(list)
  - present_or_copyin(list)
  - present_or_copyout(list)
  - present_or_create(list)
  - deviceptr(list)

```
#pragma acc kernels
{
#pragma acc loop independent
  for (int i = 0; i < n; ++i){
    for (int j = 0; j < n; ++j){
      for (int k = 0; k < n; ++k){
        B[i][j*k%n] = A[i][j*k%n];
      }
    }
  }
#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);
   }
}
```

Gang 0

```
for (i=0; i<n/2; i++){
   for (j=0;j<m;j++){
     iter(i,j);
   }
}
```

Gang 1

```
for (i=n/2+1; i<n; i++){
   for (j=0;j<m;j++){
     iter(i,j);
   }
}
```

... ...

Gang 0, Worker 0

```
for (i=0; i<n/2; i++){
   for (j=0;j<m;j+=2){
     iter(i,j);
   }
}
```

Gang 0, Worker 1

```
for (i=0; i<n/2; i++){
   for (j=1;j<m;j+=2){
     iter(i,j);
   }
}
```

Distribution scheme is compiler dependant (here simplified scheme)

# HydroC OpenACC Migration Step

**Hotspots**

- Understand your **performance goal** (analysis, definition and achievement)
- Know your **hotspots** (analysis, code reorganization, hotspot selection)
- Establish a **validation process**
- Set a **continuous integration process** with the validation

Hours to Days

**Define your Parallel Project**

**Port your Application on Many-core**

**Parallelization**

- **Optimize** CPU code
- **Exhibit** application SIMT parallelism
- Parallelize for **Many-core**
- **Validate execution**

Days to Weeks

**Many-core operational application with known potential**

Phase 1

Phase 2

**Optimize Your Many-core Application**

**Tuning**

- Reduce **data transfers**
- **Optimize** kernel execution
- Provide **feedback to application programmers** for improving algorithm data structures/…
- Consider **multiple devices**

Weeks to Months

**A corporate project**

- Purchasing Department
- Scientists
- IT Department

# Porting the `Riemann` Hotspot

```
void riemann ()
{
  #pragma acc kernels
    copy( qleft[0:Hnvar*Hstep*Hnxyt], \
          qright[0:Hnvar*Hstep*Hnxyt], \
          qgdnv[0:Hnvar*Hstep*Hnxyt], \
          sgnm[0:Hstep*Hnxyt] )
  {
    #pragma acc loop independent
    for (int s = 0; s < slices; s++){
      for (int i = 0; i < narray; i++){
        …
  }
  …
  #pragma acc kernels
    copy( qleft[0:Hnvar*Hstep*Hnxyt], \
          qright[0:Hnvar*Hstep*Hnxyt], \
          sgnm[0:Hstep*Hnxyt], \
          qgdnv[0:Hnvar*Hstep*Hnxyt] )
  {
    #pragma acc loop independent
    for (int invar = IP + 1; invar < Hnvar; invar++){
      for (int s = 0; s < slices; s++){

  …
  }
}
```

Allocate and copy data from host to device and device to host and deallocate at the end of the block.

1D *gridification*

Copy data from device to host and deallocate

Potential Speedup
$S_p = 1 / (1 - 0.4824) = 1.93$

# Parallel Regions

- Start parallel activity on the accelerator device
  - Gangs of workers are created to execute the accelerator parallel region
  - Can share HWA resources across loops nests → one region = one CUDA kernel
  - SPMD style code without barrier
- Clauses
  - `if(condition)`
  - **`async[(scalar-integer-expres`**
  - **`num_gangs(scalar-integer-exp`**
  - **`num_workers(scalar-integer-e`**
  - **`vector_length(scalar-integer`**
  - **`reduction(operator:list)`**
  - `copy(list)`
  - `copyin(list)`
  - `copyout(list)`
  - `create(list)`
  - `present(list)`
  - `present_or_copy(list)`
  - `present_or_copyin(list)`
  - `present_or_copyout(list)`
  - `present_or_create(list)`
  - `deviceptr(list)`
  - `private(list)`
  - `firstprivate(list)`

```
#pragma acc parallel num_gangs(BG),
   num_workers(BW)
{
  #pragma acc loop gang
  for (int i = 0; i < n; ++i){
    #pragma acc loop worker
    for (int j = 0; j < n; ++j){
      B[i][j] = A[i][j];
    }
  }
  for(int k=0; k < n; k++){
    #pragma acc loop gang
    for (int i = 0; i < n; ++i){
      #pragma acc loop worker
      for (int j = 0; j < n; ++j){
        C[k][i][j] = B[k-1][i+1][j] + …;
      }
    }
  }
}
```

# Iterations Mapping

```
#pragma acc parallel num_gangs(2)
{
  #pragma acc loop gang
  for (i=0; i<n; i++){
   for (j=0;j<m;j++){
     iter1(j,j);
   }
  }
 #pragma acc loop gang
  for (i=0; i<n; i++){
     iter2(i);
  }
}
```

Gang 0, Worker 0

```
for (i=0; i<n/2; i++){
   for (j=0;j<m;j++){
     iter1(i,j);
   }
}
for (i=0; i<n; i++){
     iter2(i);
}
```

Gang 1, Worker 0

```
for (i=n/2+1; i<n; i++){
   for (j=0;j<m;j++){
     iter1(i,j);
   }
}
for (i=n/2+1; i<n; i++){
     iter2(i);
}
```

one kernel

# Data Management Directives

- Data regions define scalars, arrays and sub-arrays to be allocated in the device memory for the duration of the region
  - Explicit management of data transfers using clauses or directives
- Many clauses
  - `if(condition)`
  - `copy(list)`
  - `copyin(list )`
  - `copyout(list)`
  - `create(list)`
  - `present(list)`
  - `present_or_copy(list)`
  - `present_or_copyin(list)`
  - `present_or_copyout(list)`
  - `present_or_create(list)`
  - `deviceptr(list)`

```
#pragma acc data copyin(A[1:N-2]),
  copyout(B[N])
{
  #pragma acc kernels
  {
    #pragma acc loop independant
    for (int i = 0; i < N; ++i){
      A[i][0] = ...;
      A[i][M - 1] = 0.0f;
    }
    ...
  }
  #pragma acc update host(A)
    ...
  #pragma acc kernels
  for (int i = 0; i < n; ++i){
      B[i] = ...;
  }
}
```

# Runtime API

- Set of functions for managing device allocation (C version)

  - `int acc_get_num_devices( acc_device_t )`
  - `void acc_set_device_type ( acc_device_t )`
  - `acc_device_t acc_get_device_type ( void )`
  - `void acc_set_device_num( int, acc_device_t )`
  - `int acc_get_device_num( acc_device_t )`
  - `int acc_async_test( int )`
  - `int acc_async_test_all( )`
  - `void acc_async_wait( int )`
  - `void acc_async_wait_all( )`
  - `void acc_init ( acc_device_t )`
  - `void acc_shutdown ( acc_device_t )`
  - `void* acc_malloc ( size_t )`
  - `void acc_free ( void* )`
  - ...

# HydroC Migration Step

**Hotspots**

- Understand your **performance goal** (analysis, definition and achievement)
- Know your **hotspots** (analysis, code reorganization, hotspot selection)
- Establish a **validation process**
- Set a **continuous integration process** with the validation

**Hours to Days**

**Parallelization**

- **Optimize** CPU code
- **Exhibit** application SIMT parallelism
- Parallelize for **Many-core**
- **Validate execution**

**Days to Weeks**

**Define your Parallel Project**

**Port your Application on Many-core**

**Phase 1**

**Many-core operational application with known potential**

**Phase 2**

**Optimize Your Many-core Application**

**Tuning**

- Reduce **data transfers**
- **Optimize** kernel execution
- Provide **feedback to application programmers** for improving algorithm data structures/…
- Consider **multiple devices**

**Weeks to Months**

**A corporate project**

- Purchasing Department
- Scientists
- IT Department

# Transfer Optimizations in Riemann

```
void riemann ()
{
  #pragma acc kernels
    pcopyin(qleft[0:Hnvar*Hstep*Hnxyt], qright[0:Hnvar*Hstep*Hnxyt]) \
    pcopyout(qgdnv[0:Hnvar*Hstep*Hnxyt], sgnm[0:Hstep*Hnxyt]) \
    pcopyin( Hnvar,Hstep,Hnxyt,smallp_, smallpp_, gamma6_,slices,
             narray, Hsmallr, Hsmallc, Hniter_riemann, Hgamma)
  {
    #pragma acc loop independent
    for (int s = 0; s < slices; s++){

      for (int i = 0; i < narray; i++){
        …
  }
  …
  #pragma acc kernels
    pcopyin(qleft[0:Hnvar*Hstep*Hnxyt], qright[0:Hnvar*Hstep*Hnxyt],\
    sgnm[0:Hstep*Hnxyt]), \
    pcopyout(qgdnv[0:Hnvar*Hstep*Hnxyt]) \
    pcopyin(Hnvar, Hstep, Hnxyt, slices, narray)
  {
    #pragma acc loop independent
    for (int invar = IP + 1; invar < Hnvar; invar++){

      for (int s = 0; s < slices; s++){
        …
      }
    }
  …
  }
}
```

If not present on the device, allocate and copy data from host to device

Here the optimization is local and optimize in and out status

If not present on the device at region entry, copy data from device to host

# Riemann Kernel Optimizations

```
void riemann (){
  #pragma acc kernels
    pcopyin(qleft[0:Hnvar*Hstep*Hnxyt], qright[0:Hnvar*Hstep*Hnxyt]) \
    pcopyout(qgdnv[0:Hnvar*Hstep*Hnxyt], sgnm[0:Hstep*Hnxyt]) \
    pcopyin( Hnvar,Hstep,Hnxyt,smallp_, smallpp_, gamma6_,slices,
             narray, Hsmallr, Hsmallc, Hniter_riemann, Hgamma)
  {
    #pragma acc loop independent
     for (int s = 0; s < slices; s++){
       #pragma acc loop independent
       for (int i = 0; i < narray; i++){
       …
  }
  …
  #pragma acc kernels
    pcopyin(qleft[0:Hnvar*Hstep*Hnxyt], qright[0:Hnvar*Hstep*Hnxyt],
    sgnm[0:Hstep*Hnxyt]) \
    pcopyout(qgdnv[0:Hnvar*Hstep*Hnxyt]) \
    pcopyin(Hnvar, Hstep, Hnxyt, slices, narray)
  {
    #pragma acc loop independent
    for (int invar = IP + 1; invar < Hnvar; invar++){
      #pragma acc loop independent
      for (int s = 0; s < slices; s++){
       …
      }
    }
  …
  }
}
```

Exploit more parallelism
2D gridification
more threads are created

# Porting the `Riemann` Hotspot

- Total execution time (CPU: i5-2400, 1 GPU: GTX 465)
  - Original CPU application: 10.3 sec. (serial)
  - Naive OpenACC porting: 11.1 sec.
  - Optimized transfers: 9.8 sec. ($S_p$ = 1.05)
  - Optimized kernels: 7.9 sec. ($S_p$ = 1.31 over a potential of 1.9)

# HydroC Migration Step

## Hotspots

- Understand your **performance goal** (analysis, definition and achievement)
- Know your **hotspots** (analysis, code reorganization, hotspot selection)
- Establish a **validation process**
- Set a **continuous integration process** with the validation

Hours to Days

## Parallelization

- **Optimize** CPU code
- **Exhibit** application SIMT parallelism
- Parallelize for **Many-core**
- **Validate execution**

Days to Weeks

**Define your Parallel Project**

**Port your Application on Many-core**

Many-core operational application with known potential

Phase 1

Phase 2

**Optimize Your Many-core Application**

## Tuning

- Reduce **data transfers**
- **Optimize** kernel execution
- Provide **feedback to application programmers** for improving algorithm data structures/…
- Consider **multiple devices**

Weeks to Months

**A corporate project**

- Purchasing Department
- Scientists
- IT Department

# Further Optimizations

Reuse of data on GPU across multiple compute regions with a data region

main (100%)

hydro_godunov (97.0%)

compute_deltat (2.9%)

riemann (48.2%)

slope (11.2%)

trace (10.1%)

updateConservativeVars (5.6%)

gatherConservativeVars (5.4%)

equationofstate (5.0%)

qleftright (4.9%)

constoprim (4.6%)

cmpflx (3.0%)

# Adding Data Region

```
void hydro_godunov (…)
{
#pragma acc data \
  create(qleft[0:H.nvar], qright[0:H.nvar], \
         q[0:H.nvar], qgdnv[0:H.nvar], \
         flux[0:H.nvar], u[0:H.nvar], \
         dq[0:H.nvar], e[0:Hstep], c[0:Hstep], \
         sgnm[0:Hstep], qxm[0:H.nvar], qxp[0:H.nvar]) \
  copy(uold[0:H.nvar*H.nxt*H.nyt]) \
  copyin(Hstep)
 {
  for (j = Hmin; j < Hmax; j += Hstep){
       // compute many slices each pass
         int jend = j + Hstep;
         if (jend >= Hmax)
           jend = Hmax;
         . . .// the work here
   } // for j
  }//end of data region
...
```

Data are left on the GPU during the step loop. pcopy clauses are used into called routines

# Full Application

- With the same strategy, the full application have been ported with OpenACC

- The following hotspots have been accelerated
  - cmplx
  - updateConservativeVar
  - gatherConservativeVar
  - constoprim
  - equationofstate
  - qleftright
  - riemann
  - slope
  - trace

1 week of development

60 directives, 4% of the LoC

Achieved speedup = 3x
and still room for improvement

# A Few Rules When Using OpenACC

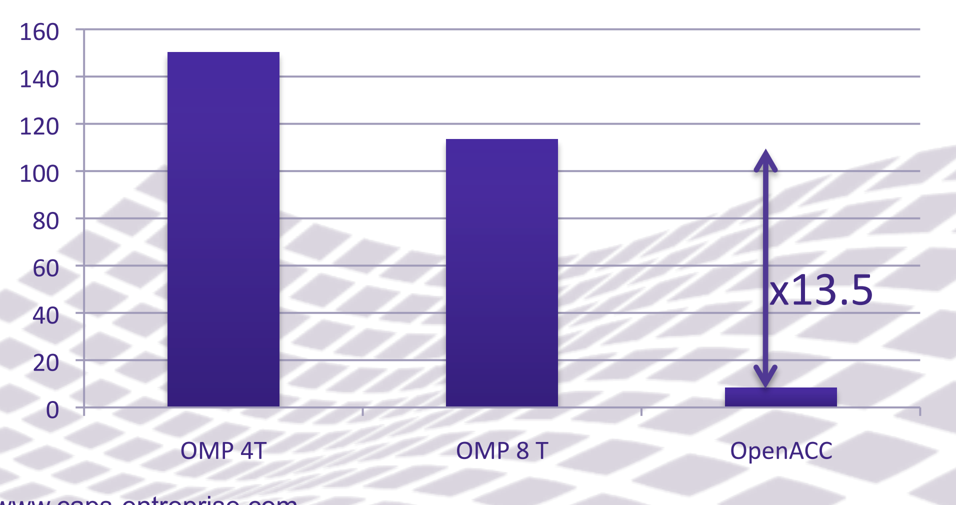
- Use kernels regions as a default
  - Parallel region semantic less trivial
- Add the copy-in and copy-out data clauses
  - Automatic variables scoping is compiler dependant
- Use gangs and workers before using vectors
  - Meanings is more uniform across compilers
  - Vector is more a hints, gang is not (especially in parallel section)
  - Use them in the order gang/worker/vector with one level of each
- Check different values of gangs and workers for performance
  - Similar to CUDA thread block tuning
- Avoid calls to routine in kernels/regions that are not in the same file
  - Inter-procedural inlining may not always work
  - This is ongoing discussion in the consortium

# Playing with Gangs and Workers

```
void convSM_N(typeToUse A[M][N], typeToUse B[M][N]){
 int i, j, k; int m=M, n=N;
 #pragma acc kernels pcopyin(A[0:m]) pcopy(B[0:m])
 {
   typeToUse c11, c12, c13, c21, c22, c23, c31, c32, c33;

   c11 = +2.0f;  c21 = +5.0f;  c31 = -8.0f;
   c12 = -3.0f;  c22 = +6.0f;  c32 = -9.0f;
   c13 = +4.0f;  c23 = +7.0f;  c33 = +10.0f;

   #pragma acc loop gang(16)
   for (int i = 1; i < M - 1; ++i) {
       #pragma acc loop worker(16)
       for (int j = 1; j < N - 1; ++j) {
           B[i][j] =c11*A[i-1][j-1]+c12*A[i+0][j-1]+c13*A[i+1][j-1]
              +c21*A[i-1][j+0]+c22*A[i+0][j+0]+c23*A[i+1][j+0]
              +c31*A[i-1][j+1]+c32*A[i+0][j+1]+c33*A[i+1][j+1];
         }
     }

  }//kernels region
}
```

# DNA Distance Application with OpenACC

- Biomedical application part of Phylip package,
  - Main computation kernel takes as input a list of DNA sequences for each species
    - Code is based on an approximation using Newton-Raphson method (SP)
    - Produces a 2-dimension matrix of distances
  - Experiments performed in the context of the HMPP APAC CoC*
- Performance
  - OpenMP version, 4 & 8 threads, Intel(R) i7 CPU 920 @ 2.67GHz
  - 1 GPU Tesla C2070

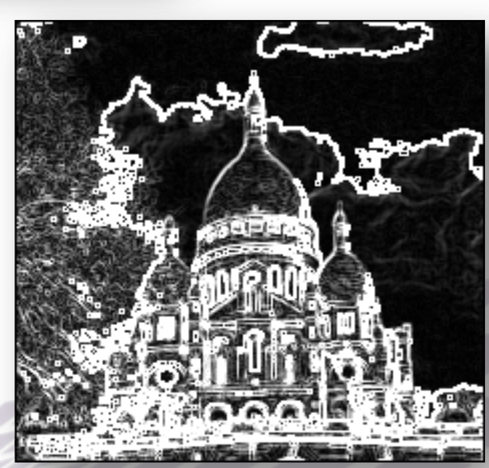



*http://competencecenter.hmpp.org/category/hmpp-coc-asia/

# Sobel Filter Performance Example

**Edge detection algorithm**

- Sobel Filter benchmark
- Size
  - ~ 200 lines of C code
- GPU C2070 improvement
  - x 24 over serial code on Intel i7 CPU 920 @ 2.67GHz
- Main porting operation
  - Inserting 6 OpenACC directives



Accelerator Programming Model Parallelization

Directive-based programming GPGPU Manycore programming

Hybrid Manycore Programming HPC community OpenACC

Petaflops Parallel computing HPC open standard

Multicore programming Exaflops NVIDIA Cuda

Code speedup Hardware accelerators programming

High Performance Computing OpenHMPP

Parallel programming interface

Massively parallel

Open CL

CAPS

OpenACC. DIRECTIVES FOR ACCELERATORS

open hmpp

http://www.caps-entreprise.com
http://twitter.com/CAPSentreprise
http://www.openacc-standard.org/
http://www.openhmpp.org

# Going Further with OpenHMPP Compiler

# What is in OpenHMPP and not in OpenACC

- Library integration directives
  - Needed for a “single source many-core code” approach
- Tracing Open performance APIs
  - Allows to use tracing third party tools
- Tuning directives
  - Loops nest transformations
- External and native functions
  - Allows to include CUDA code into kernels
- Multiple devices management
  - Data collection / map operation
- And many more features
  - Loop transformations directives for kernel tuning
  - buffer mode, UVA support, …

# Library Integration

# Dealing with Libraries

- Library calls can usually only be partially replaced
  - No one-to-one mapping between libraries (e.g. BLAS, FFTW, CuFFT, CULA, ArrayFire)
  - No access to all application codes (i.e. avoid side effects)
  - **Want a unique source code**

- 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)

- Libraries can be written in many different languages
  - CUDA, OpenCL, OpenHMPP, etc.

- OpenACC uses device pointers (`deviceptr` clause)
  - Accelerator specific, not a portable technique

# Library Mapping Example

FFTW

```
fftw_plan  fftwf_plan_dft_r2c_3d(
           sz,  sy,  sx,
           work1,  work2,
           FFTW_ESTIMATE);

fftwf_execute(p);

fftwf_destroy_plan(p);
```

NVIDIA cuFFT

```
cufftHandle  plan;
cufftPlan3d(&plan,sz,sy,sx,CUFFT_R2C);

cufftExecR2C(plan,(cufftReal*) work1,
                  (cufftComplex  *) work2);

cufftDestroy(plan);
```

# Proxy Directives "hmppalt" in HMPP3.0

- A proxy indicated by a directive is in charge of calling the accelerated library
- Proxies get the execution context from the HMPP runtime
- Proxies are used only to selected calls to the library

Replaces the call to a proxy that handles GPUs and allows to mix user GPU code with library ones

```
C
      CALL INIT(A,N)
      CALL ZFFT1D(A,N,0,B)  ! This call is needed to initialize FFTE
      CALL DUMP(A,N)

!$hmppalt ffte call , name="zfft1d", error="proxy_err"
      CALL ZFFT1D(A,N,-1,B)
      CALL DUMP(A,N)
C
C SAME HERE
!$hmppalt ffte call , name="zfft1d" , error="proxy_err"
      CALL ZFFT1D(A,N,1,B)
      CALL DUMP(A,N)
```

# Library Interoperability in HMPP 3.0

C/CUDA/…

HMPP Runtime API

Native GPU Runtime API

```
...
 call libRoutine1(…)
...
...
#pragma hmppalt
call libRoutine2(…)
...
...
...
...
call libRoutine3(…)
...
```

```
...
...
...
...
...
...
 proxy2(…)
...
...
```

**GPU Lib**

```
...
...
...
...
...
...
 gpuLib(…)
...
...
```

**CPU Lib**

```
...
cpuLib1(…)
...
...
...
...
cpuLib3(…)
...
...
```

# Library Integration with HMPPALT – Example

- The proxy set **blascuda** to translate BLAS calls into cuBLAS calls is composed of
  - A Fortran include file `blascuda.inc`
  - A shared library `libblascuda.so`
  - Some documentation
- A typical BLAS code

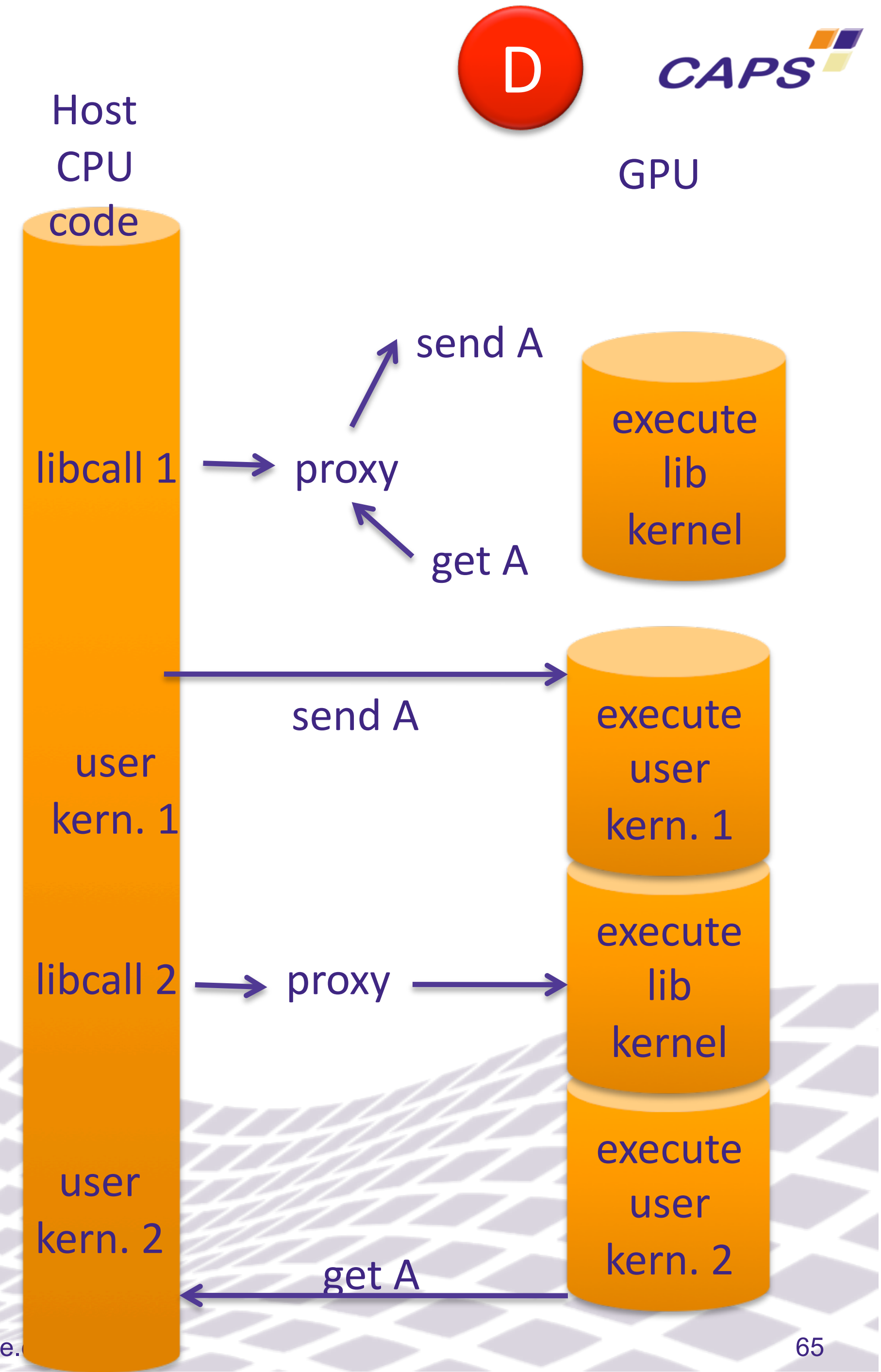
```
SUBROUTINE doit(n,X,Y)
        REAL :: X(n), Y(n)
        REAL prod
        CALL sscal(n, 2.0, 1, X)
        prod = sdot(n, 1, X, 1, Y)
END SUBROUTINE
```

```
SUBROUTINE doit(n,X,Y)
        INCLUDE "blascuda.inc"
        REAL :: X(n), Y(n)
        REAL prod
        !$HMPPALT blascuda call, name="sscal"
        CALL sscal(n, 2.0, 1, X)
        !$HMPPALT blascuda call, name="sdot"
        prod = sdot(n, 1, X, 1, Y)
END SUBROUTINE
```

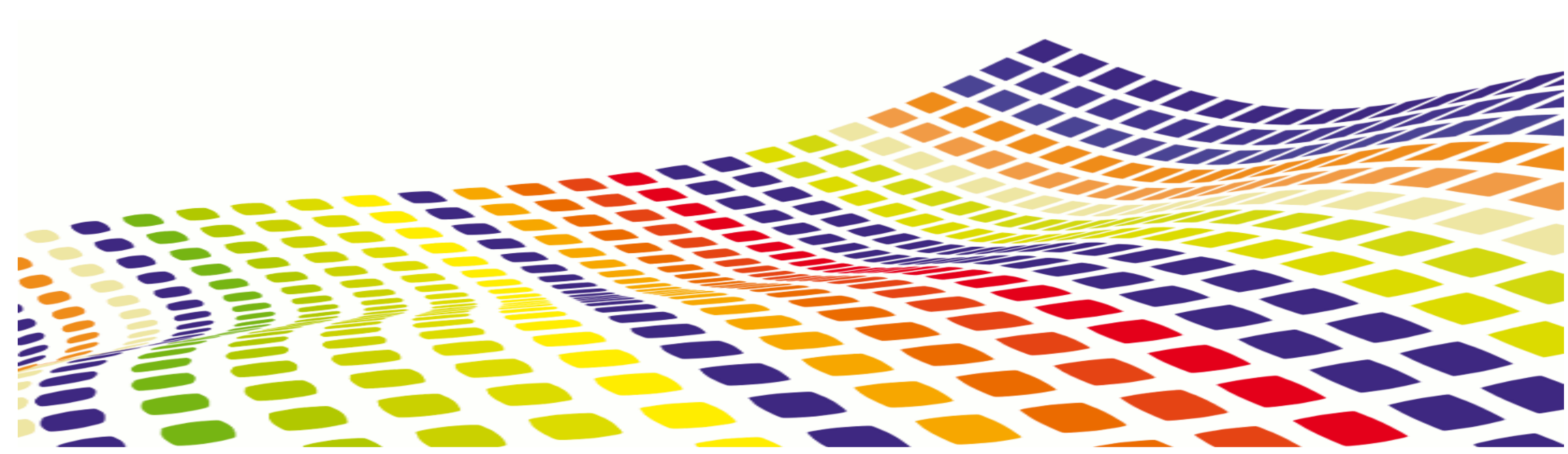
- Must be compiled with HMPP and linked with **`libmyblas.so`**
  - **`ifort -O3 -shared -o libmyblas.so  myblas.f90  -lcublas`**
  - **`hmpp ifort  myApplication.f90 -o myApplication.exe -lmyblas`**

# Different Proxies Settings

- Library level CPU-GPU data transfer management
  - Data updates are performed inside the proxies
  - Can be redundant with accelerated users code
- Users level CPU-GPU data transfer management
  - No data updates are performed inside the proxies
  - Allow to optimize the data transfers between users and library kernels



# Tuning Directives



# Code Tuning Directives

- Directive-based HWA kernel code transformations
- Directives preserve original CPU code

```
#pragma hmpp <mygroup> sgemm codelet, args[tout].io=inout, &
#pragma hmpp &          args[*].mirror args[*].trans
void sgemm( float alphav[1], float betav[1], const float t1[SIZE][SIZE],
const float t2[SIZE][SIZE], float tout[SIZE][SIZE] ) {

  int j, i;
  const float alpha = alphav[0], beta = betav[0];

#pragma hmppcg(CUDA) unroll i:4, j:4, split(i), noremainder(i,j), jam
#pragma hmppcg gridify (j,i)
  for( j = 0 ; j < SIZE ; j++ ) {
    for( i = 0 ; i < SIZE ; i++ ) {
      int k;
      float prod = 0.0f;
      for( k = 0 ; k < SIZE ; k++ ) {
        prod += t1[k][i] * t2[j][k];
      }
      tout[j][i] = alpha * prod + beta * tout[j][i];
    }
  }
}
```

Loop transformations

Apply only when compiling for CUDA

# Tuning Example – 1*

- **HMPP-transformed PolyBench codes using CUDA and OpenCL:**
  - Given in terms of speedup over default (non-transformed) HMPP code
  - Compared with results of manually-written CUDA/OpenCL implementation
  - HMPP transformations gives speedup over default in 8 of the 14 transformed codes using CUDA and 6 of the 14 codes using OpenCL
    - **CUDA Results:**

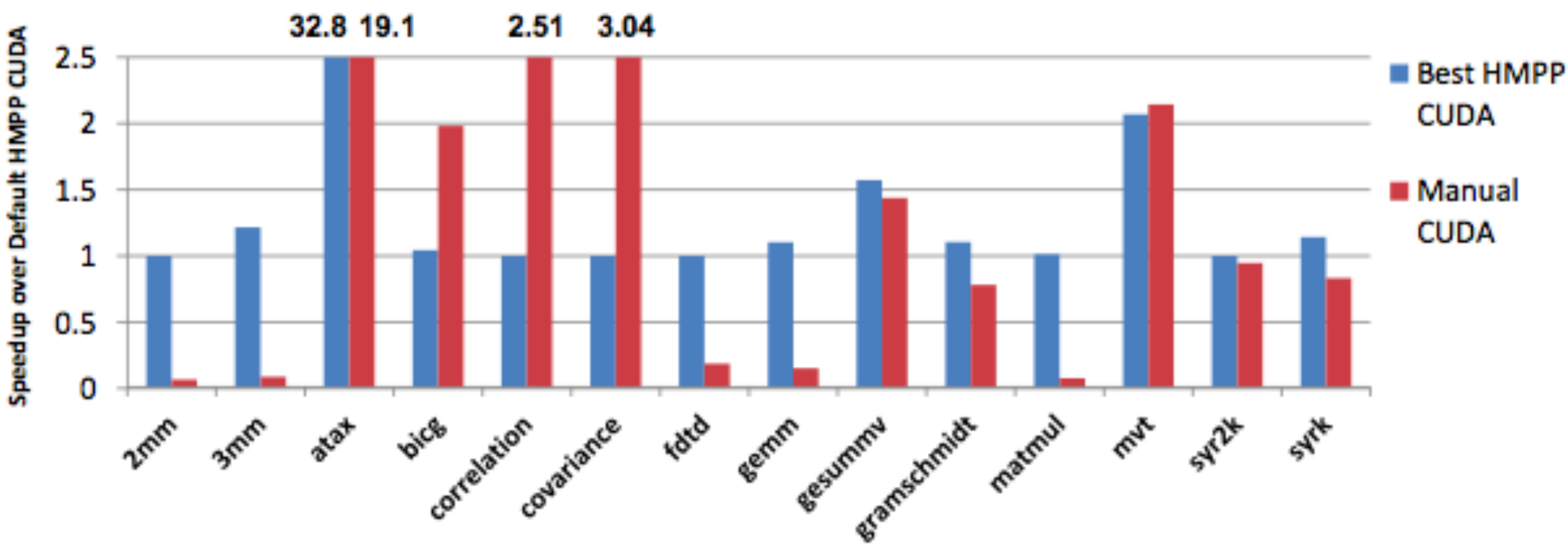



**Figure 2**: Speedup of best HMPP-transformed and manually-written PolyBench CUDA codes over default HMPP CUDA configuration

*From "Autotuning a High-Level Language Targeted to GPU Kernels", S. Grauer-Gray, R. Searles, L. Xu, S. Ayalasomayajula, J. Cavazos Supercomputing 2011, University of Delaware

# Tuning Example – 2*

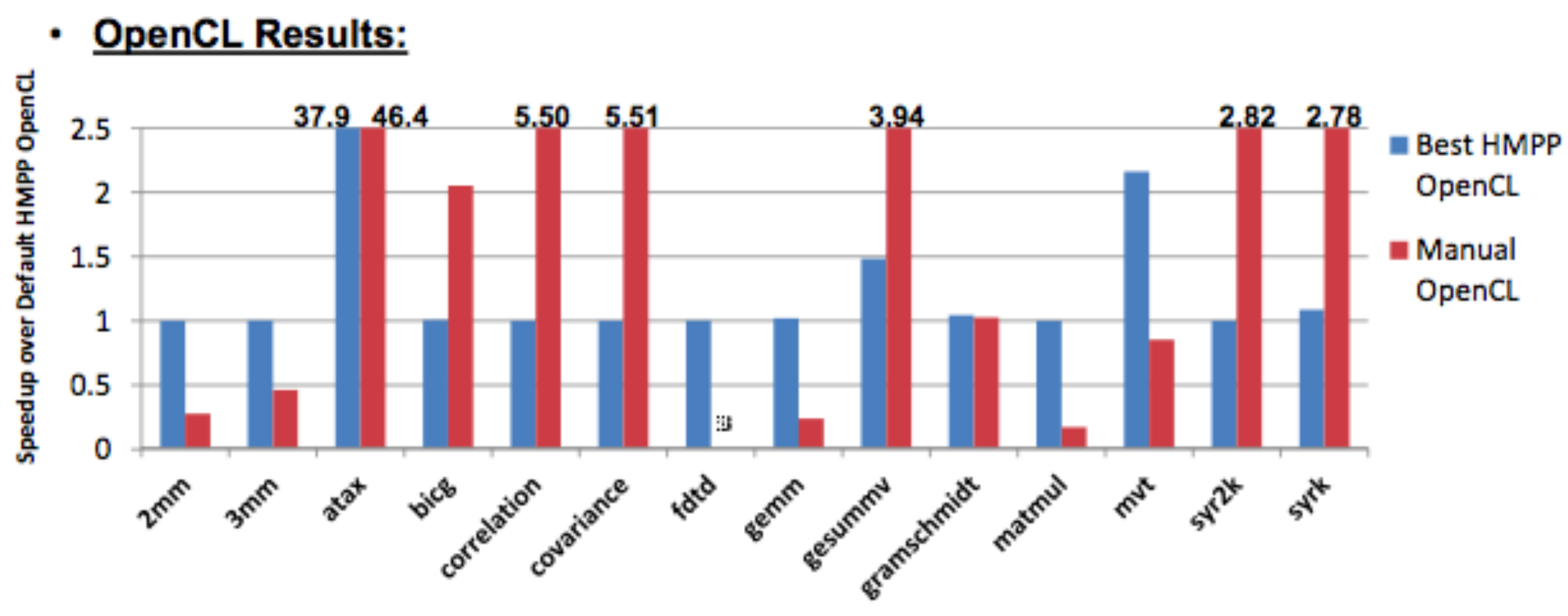



**Figure 3**: Speedup of best HMPP-transformed and manually-written PolyBench OpenCL codes over default HMPP OpenCL configuration

*From "Autotuning a High-Level Language Targeted to GPU Kernels",
S. Grauer-Gray, R. Searles, L. Xu, S. Ayalasomayajula, J. Cavazos
Supercomputing 2011, University of Delaware

# Multi-Accelerator Programming with OpenHMPP

# Multi-Accelerator with OpenHMPP – Device Clause

- Most HMPP directives are extended with a device clause

```
#pragma hmpp <MyLabel> MyCodelet command, …, device="expr"
```

- The device clause requires an index within the allocated devices (0,1,...)
- All arguments in a multi-device group must use the Mirror Storage Policy, each mirror is allocated on a single device

```
#pragma hmpp <MyLabel> allocate, data[x] , device="..."
```

- All mirrored arguments of a callsite must be allocated on the same device
  - The callsite is automatically performed on that device thanks to the "owner computes rule"

# Collections of Data in OpenHMPP 3.x

```
struct {
float * data ;
int size ;
} Vec;
```

# OpenHMPP 3.0 Map Operation on Data Collections

D

```
#pragma hmpp <mgrp> parallel
for(k=0;k<n;k++) {
      #pragma hmpp <mgrp> f1 callsite
          myparallelfunc(d[k],n);
    }
```

CPU 0

CPU 1

GPU 0

GPU 1

GPU 2

Main memory

d0 d1 d2 d3

Device memory

d1

Device memory

d2

Device memory

d3

Accelerator Programming Model Parallelization

Directive-based programming GPGPU Manycore programming

Hybrid Manycore Programming HPC community OpenACC

Petaflops Parallel computing HPC open standard

Multicore programming Exaflops NVIDIA Cuda

Code speedup Hardware accelerators programming

High Performance Computing OpenHMPP

Parallel programming interface

Massively parallel

Open CL

CAPS

OpenACC
DIRECTIVES FOR ACCELERATORS

open hmpp

http://www.caps-entreprise.com
http://twitter.com/CAPSentreprise
http://www.openacc-standard.org/
http://www.openhmpp.org