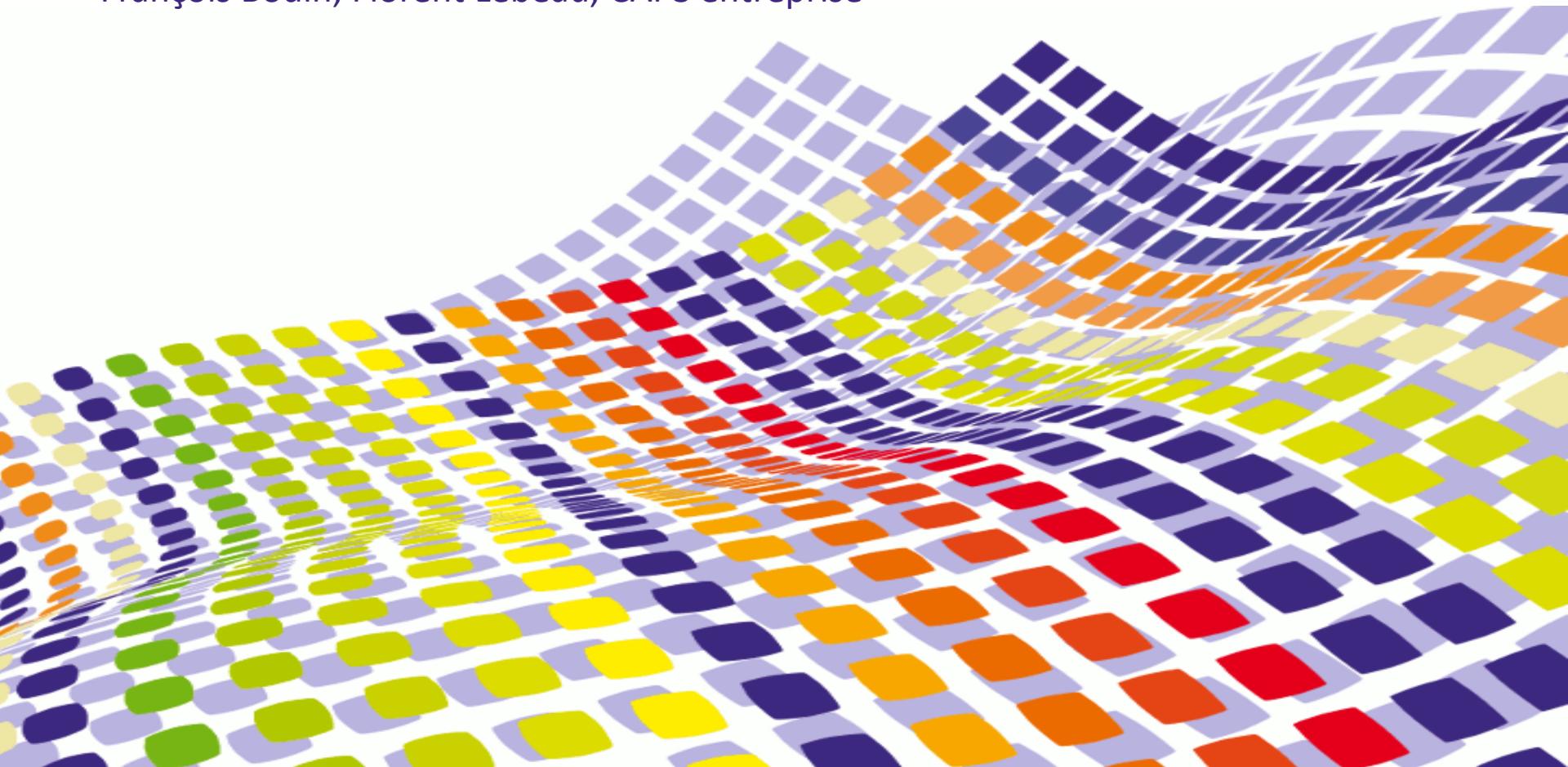


Programming Heterogeneous Many-cores Using Directives

HMPP - OpenAcc

François Bodin, Florent Lebeau, 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

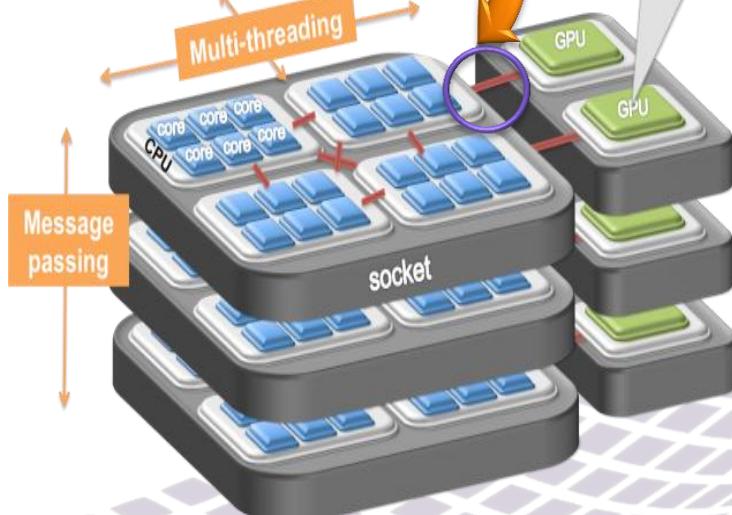
- Written in C/C++/Fortran
- Mix of user code and library calls
- Hotspots may or may not be parallel
- Lifetime in 10s of years
- Cannot be fully re-written
- Migration can be risky and mandatory

```
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 PCIe bus

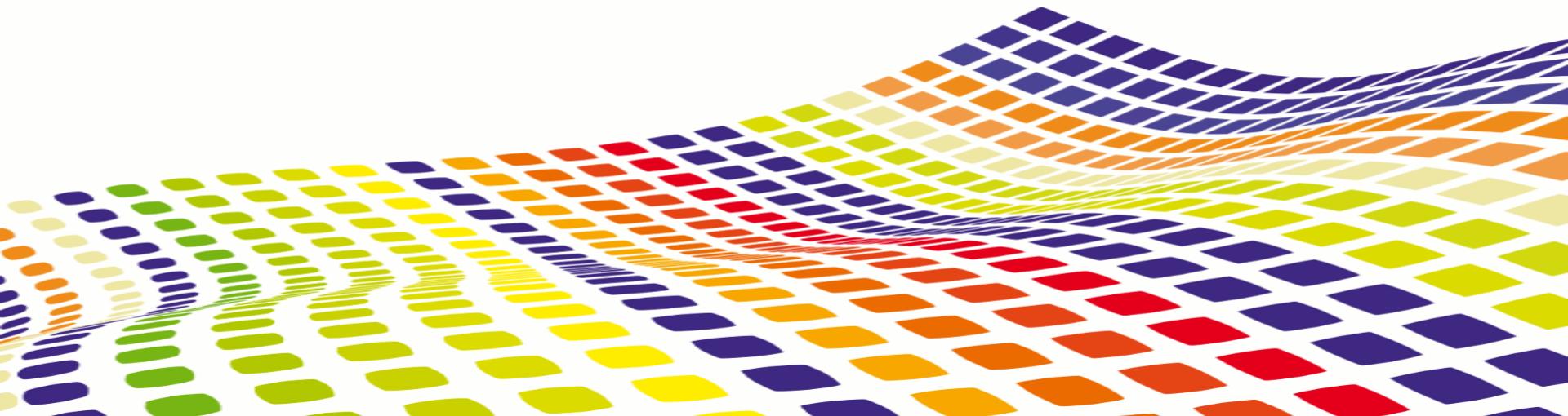


Data/stream/vector parallelism to be exploited by HWA
e.g. CUDA / OpenCL

Outline of the Presentation

- Usual parallel programming techniques in HPC
- Many-core architecture space
- Demonstration 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**
 - Same computation applied to all elements of a collection
 - Data distribution over the processors
 - Frequently exploit loop-level parallelism

→ SIMD / SIMT
- **Message Passing**
 - Mostly distributed memory
 - Frequently use data distribution techniques

→ MPI Library

Task Based Example: Threads Fork and Join

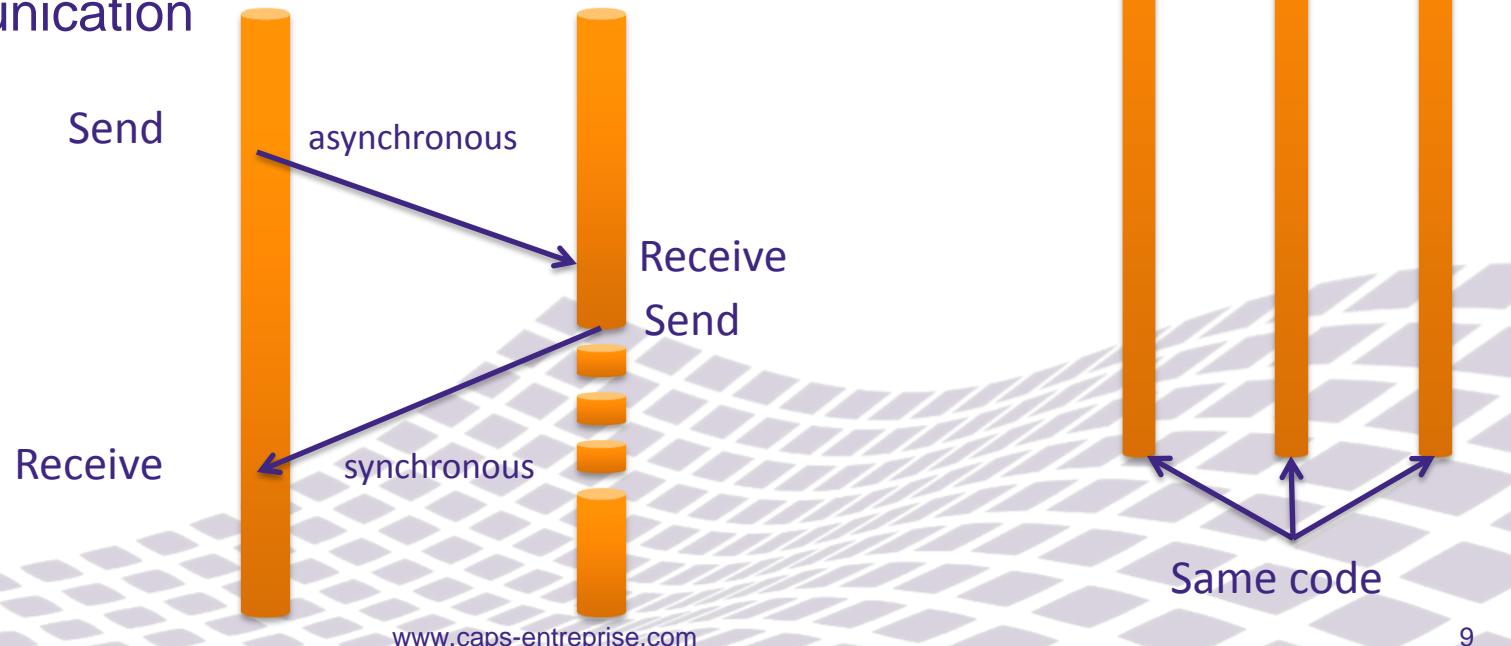
- The whole the programming environment assumes some level of thread managements
 - Gang scheduling
 - Thread / data affinity / migration
- For instance OpenMP Parallel Loops
 - Express distribution of loop iterations over the threads
 - Various scheduling clauses

```
#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

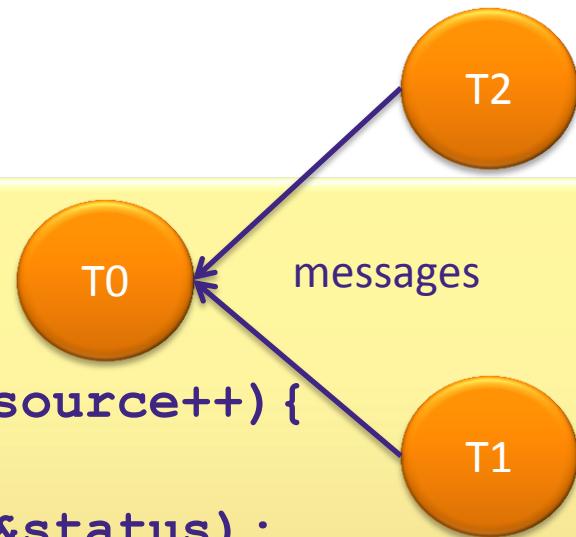


MPI Example

```
initialize
if (my_rank== 0) {
    sum = 0.0;
    for (source=1; source<num_procs; source++) {
        MPI_RECV(&value, 1, MPI_FLOAT,
                 source, tag,MPI_COMM_WORLD, &status);
        sum += value;
    }
} else {
    MPI_SEND(&value, 1, MPI_FLOAT, 0, tag,
             MPI_COMM_WORLD);
}
finalize
```

MPI_ANY_TAG

MPI_ANY_SOURCE

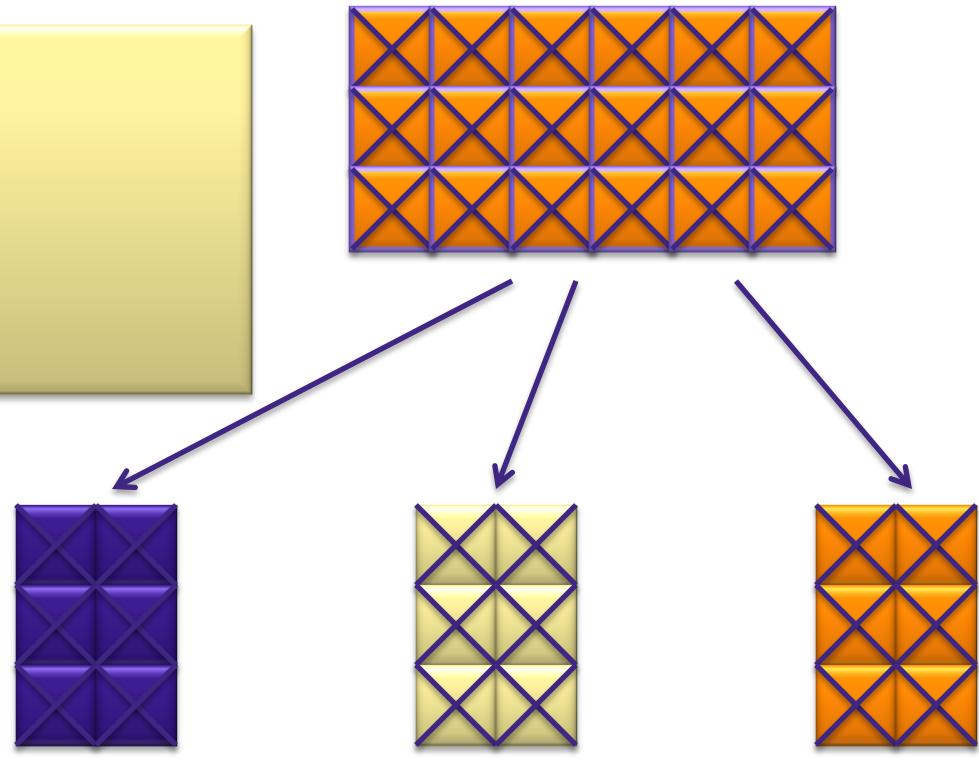


SPMD MPI code

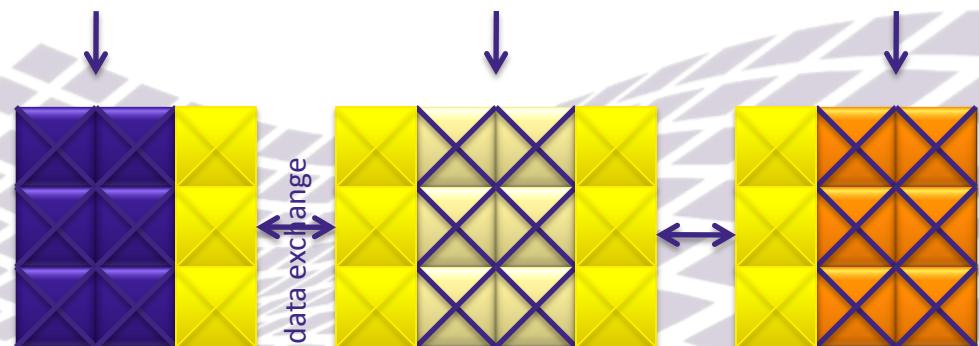
Data Distribution Example

```
forall cells of PE  
    read properties, state, neighbors;  
for time = 1 to end  
    forall cells of PE  
        compute next step  
        update neighbors ghost cells
```

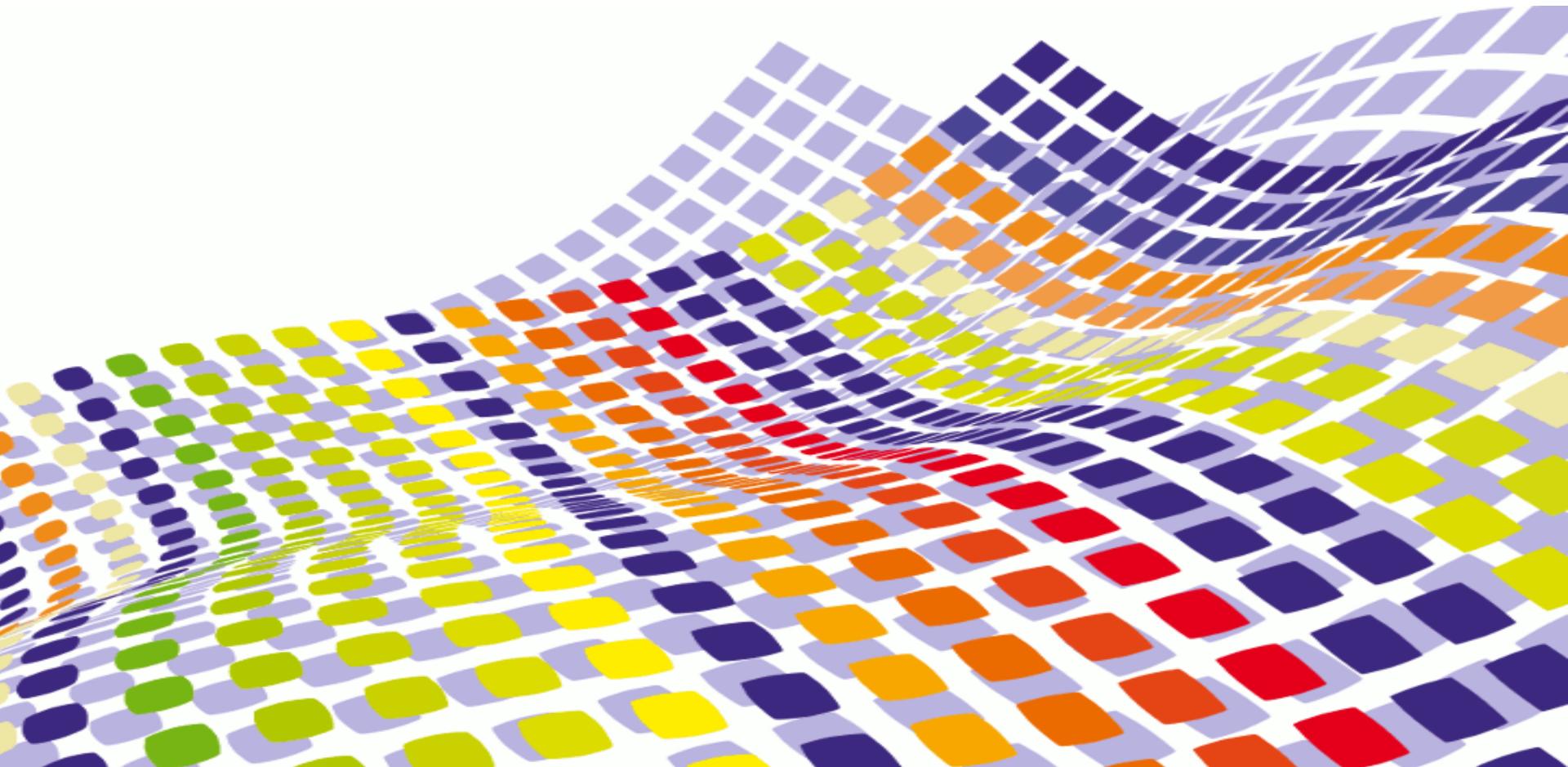
1. Domain decomposition



1. Add ghost cells

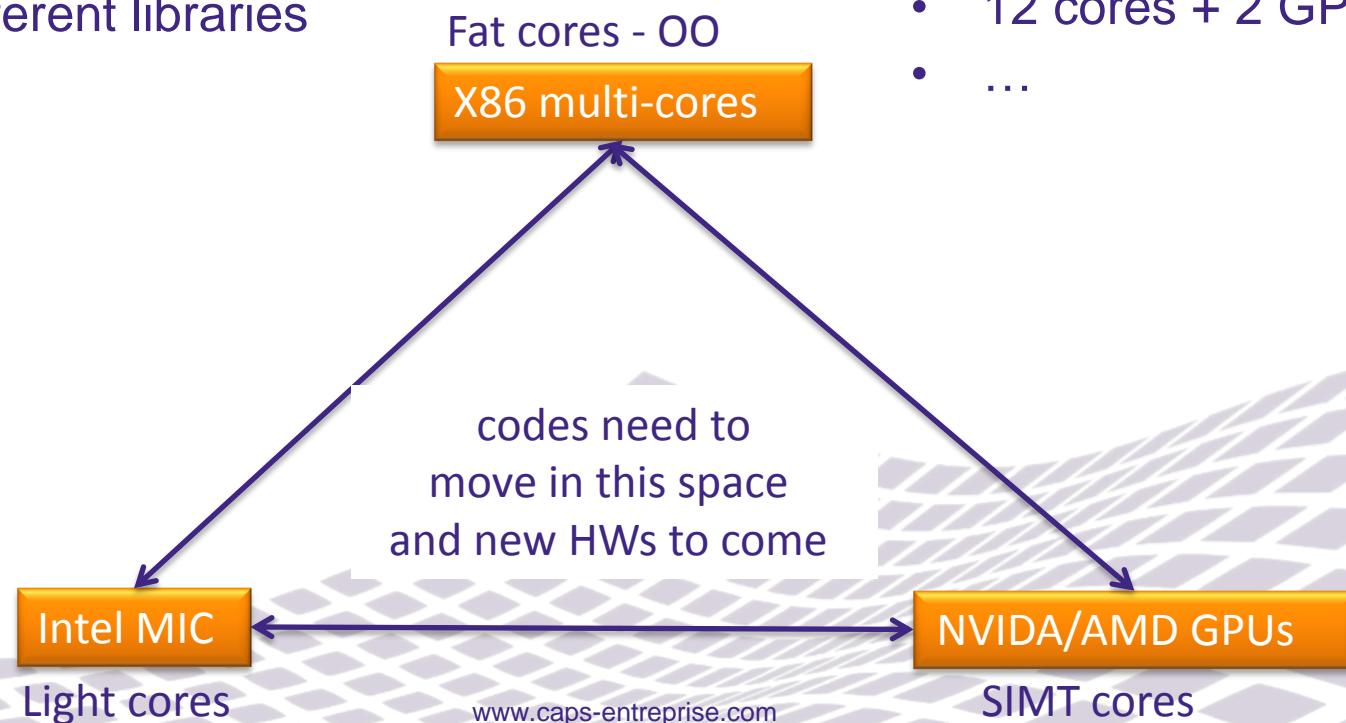


Many-Core Architecture Space

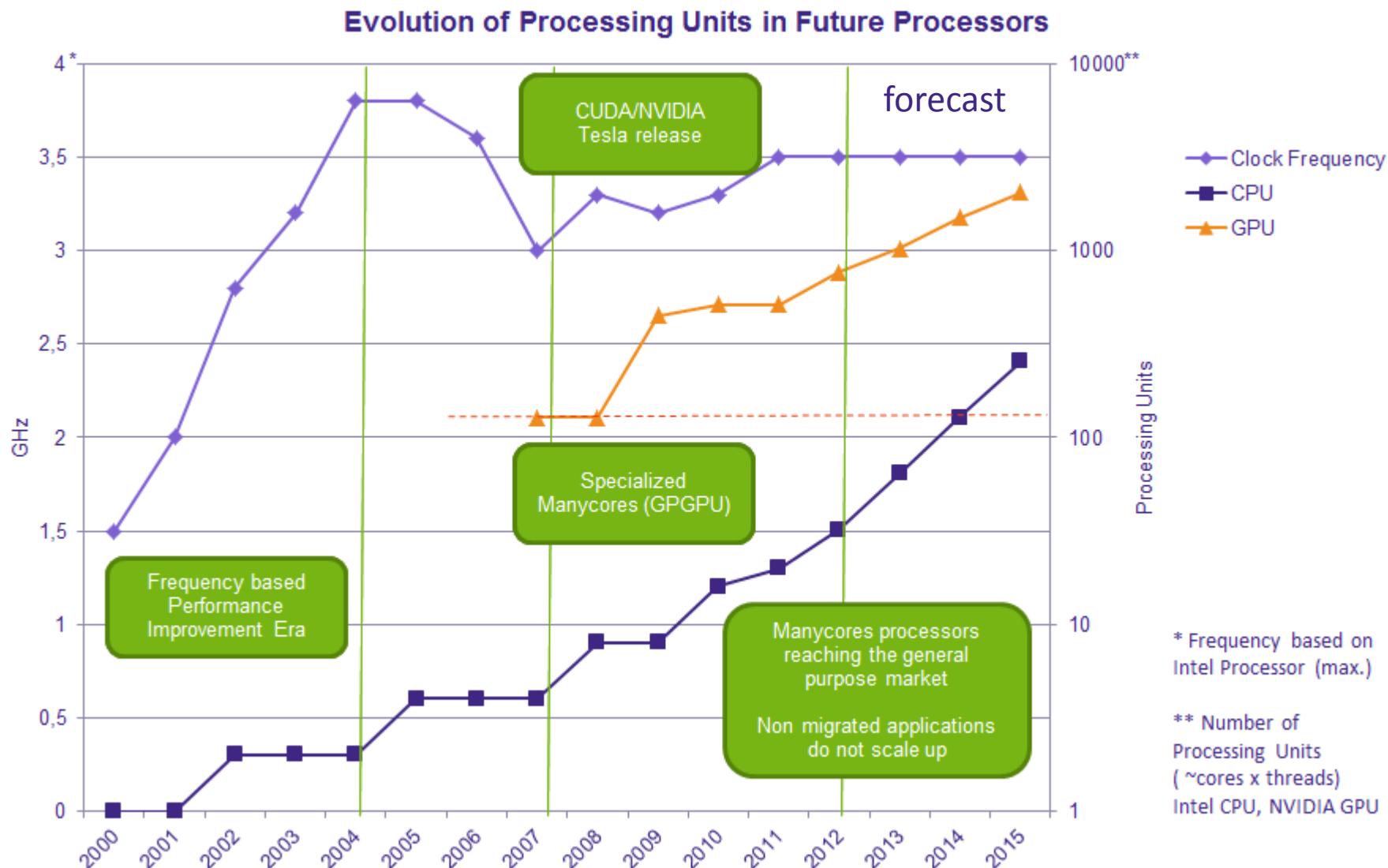


Heterogeneous Architecture Space

- 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?

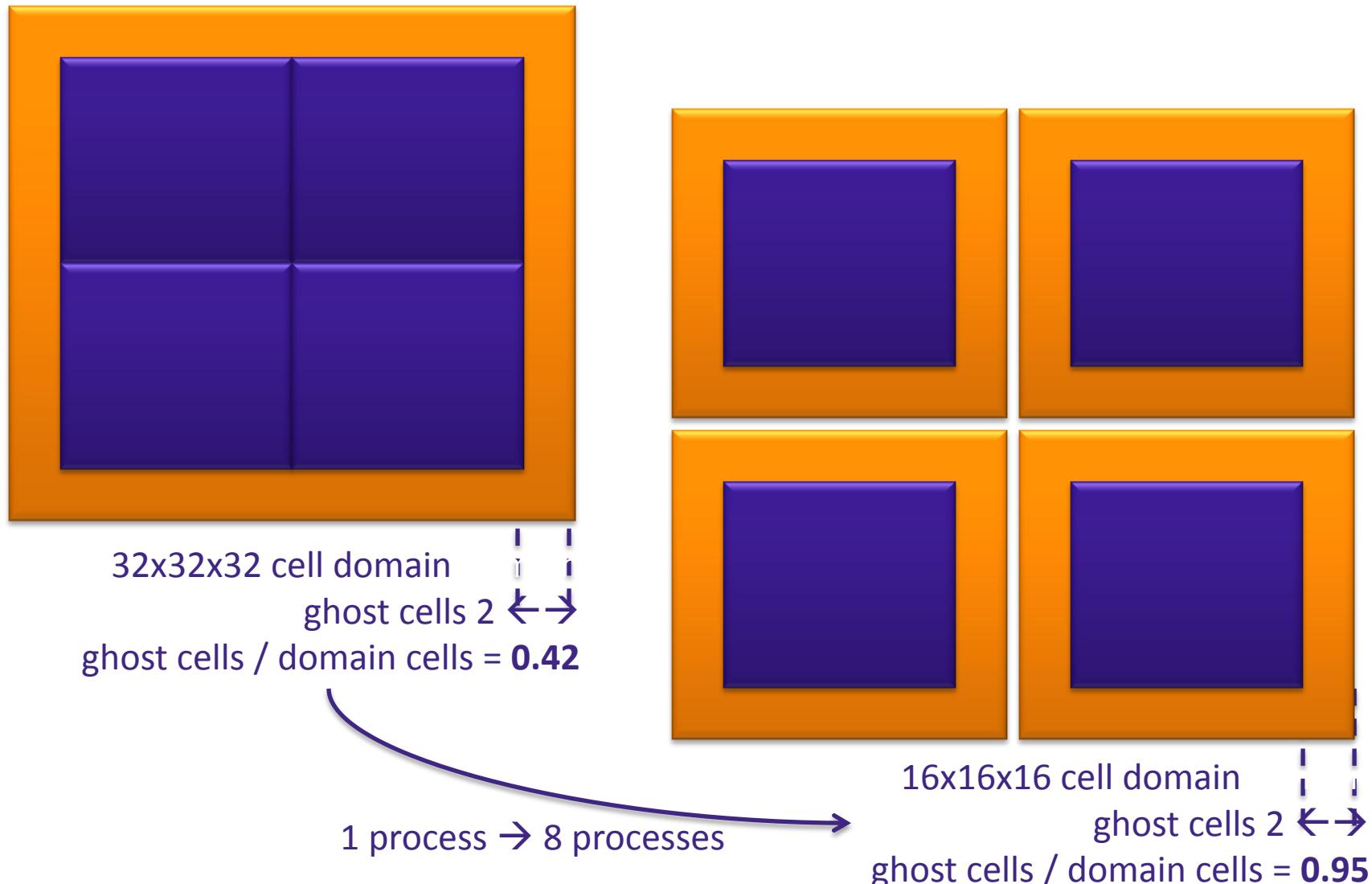


Usual Parallel Programming Won't Work Per Se



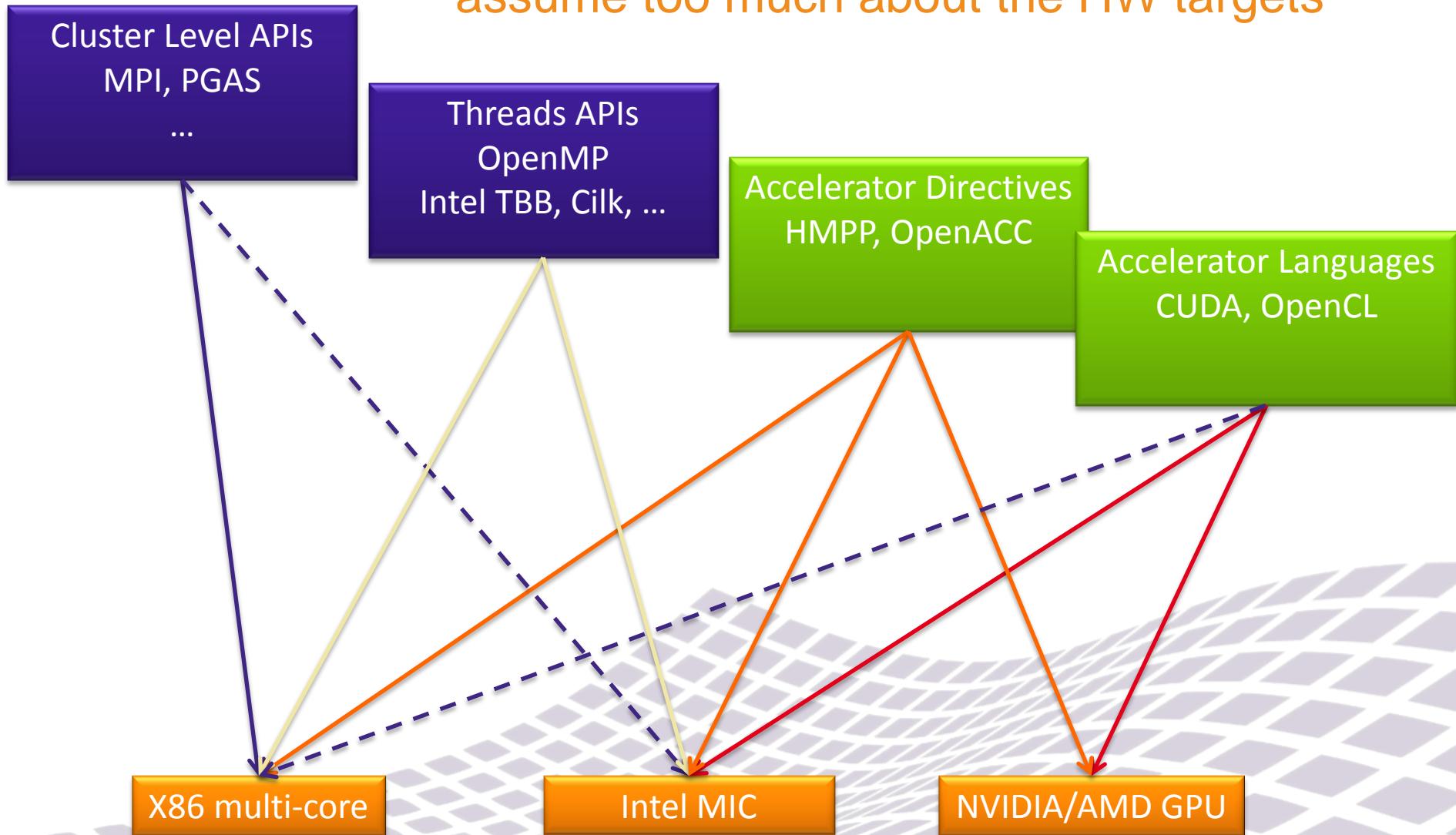
- Exploiting heterogeneous many-core with MPI parallel processes
 - Extra latency compared to shared memory use
 - MPI implies some copying required by its semantics (even if efficient MPI implementations tend to reduce them)
 - Cache trashing between MPI processes
 - Excessive memory utilization
 - 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



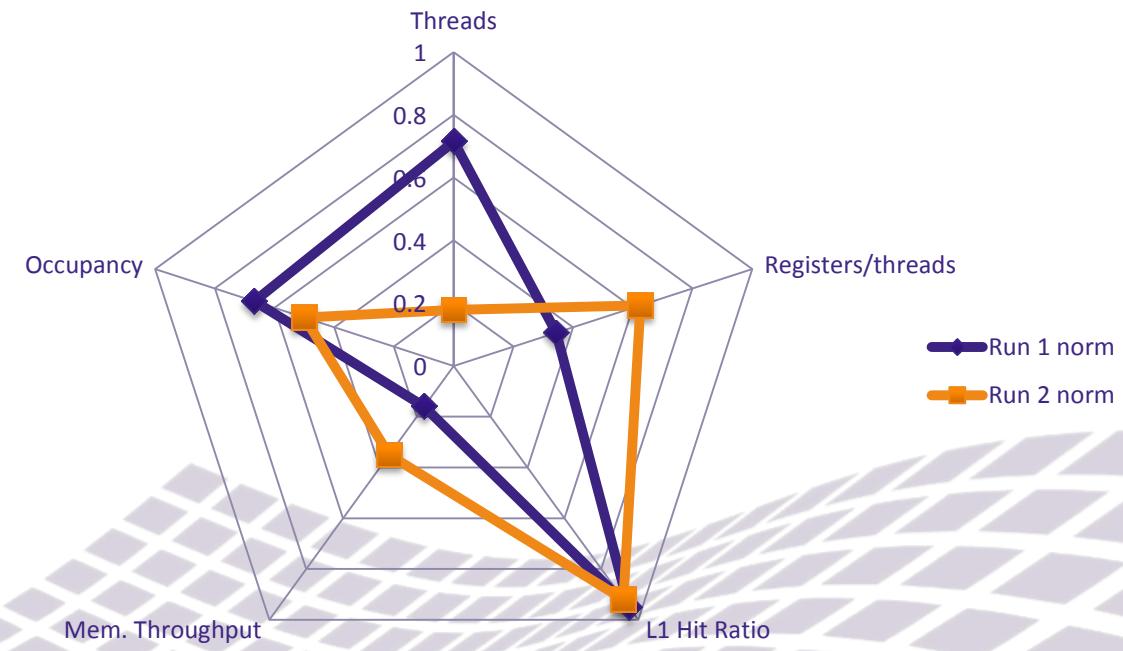
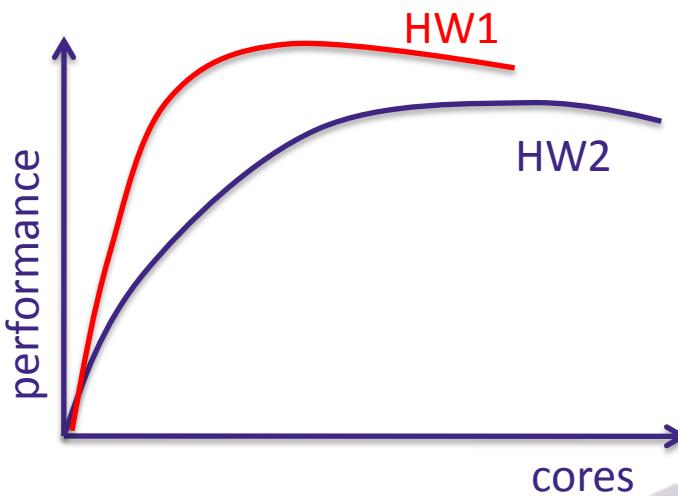
Flexible Code Generation Required

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



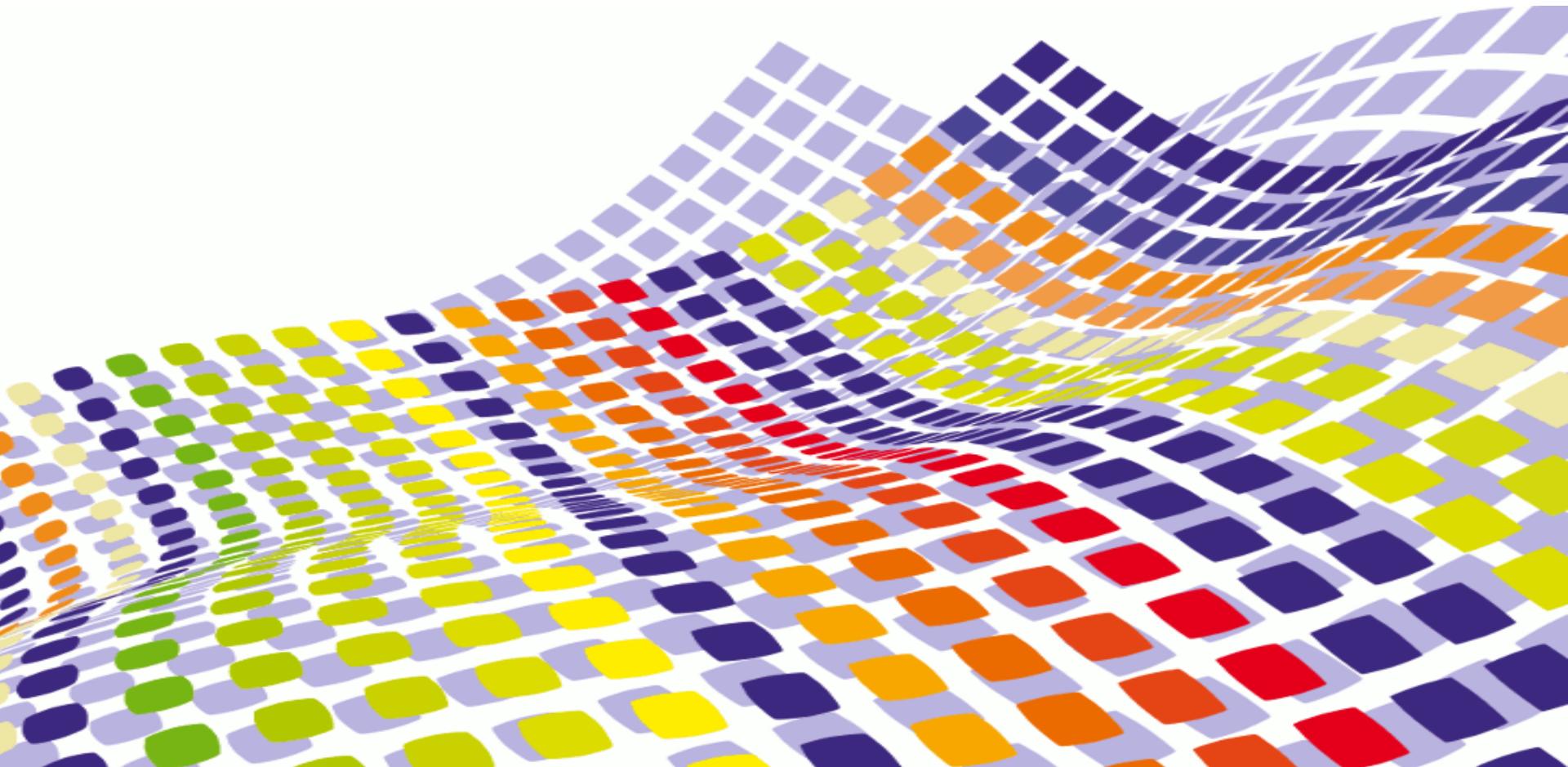
Auto-Tuning is Required to Achieve Some Performance Portability

- 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)



Example of an optimized versus a non optimized stencil code

Demonstration 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 5.1, 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_lyon2011.pdf

*Pierre-François Lavallée^a, Guillaume Colin de Verdière^b,
Philippe Wautelet^a, Dimitri Lecas^a, Jean-Michel Dupays^a
^aIDRIS/CNRS, ^bCEA, Centre DAM

HydroC Application – CPU Version

D

CAPS

- To compile:

```
$ make
```

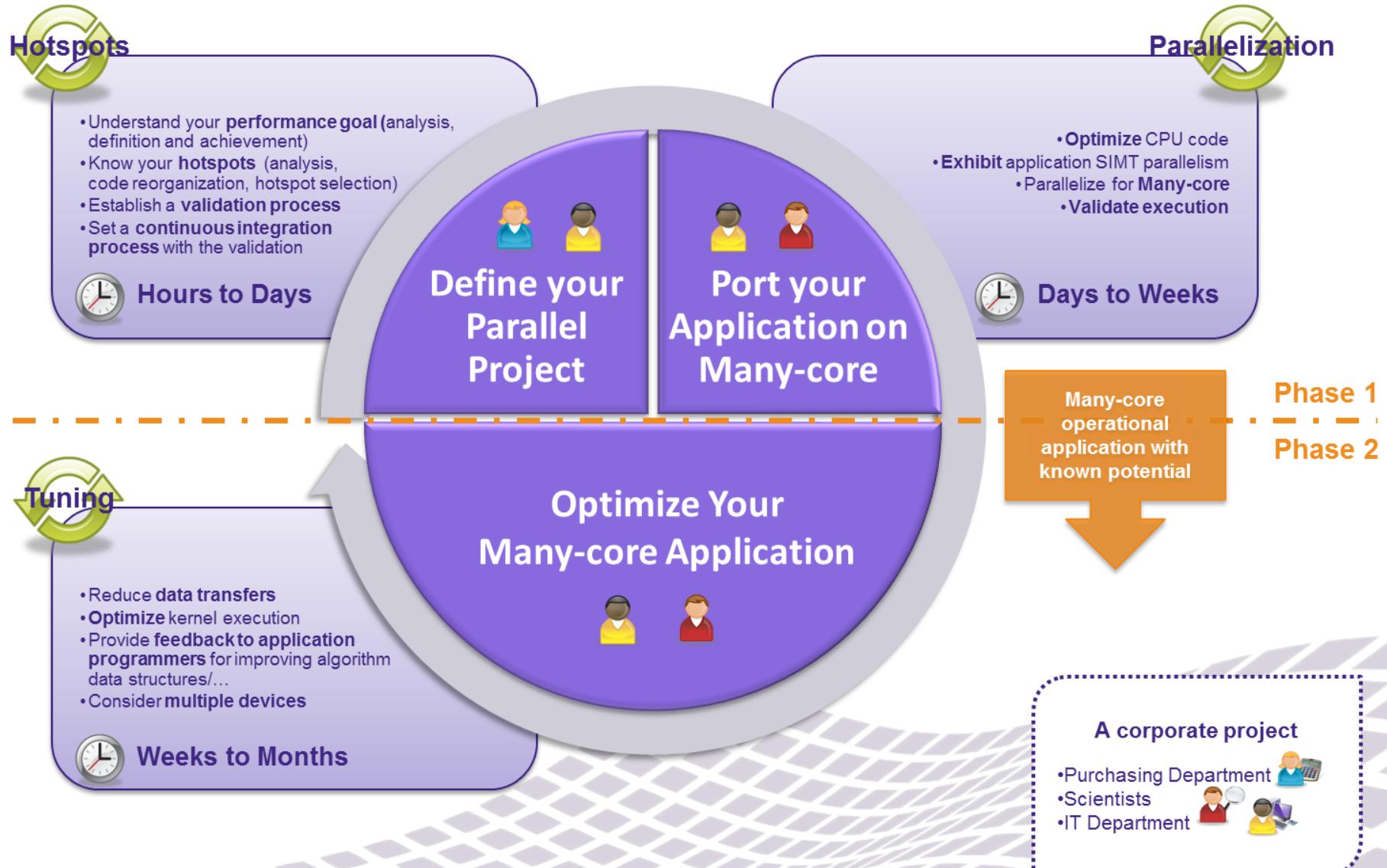
- To execute:

```
$ ./HydroC -i input.base
```

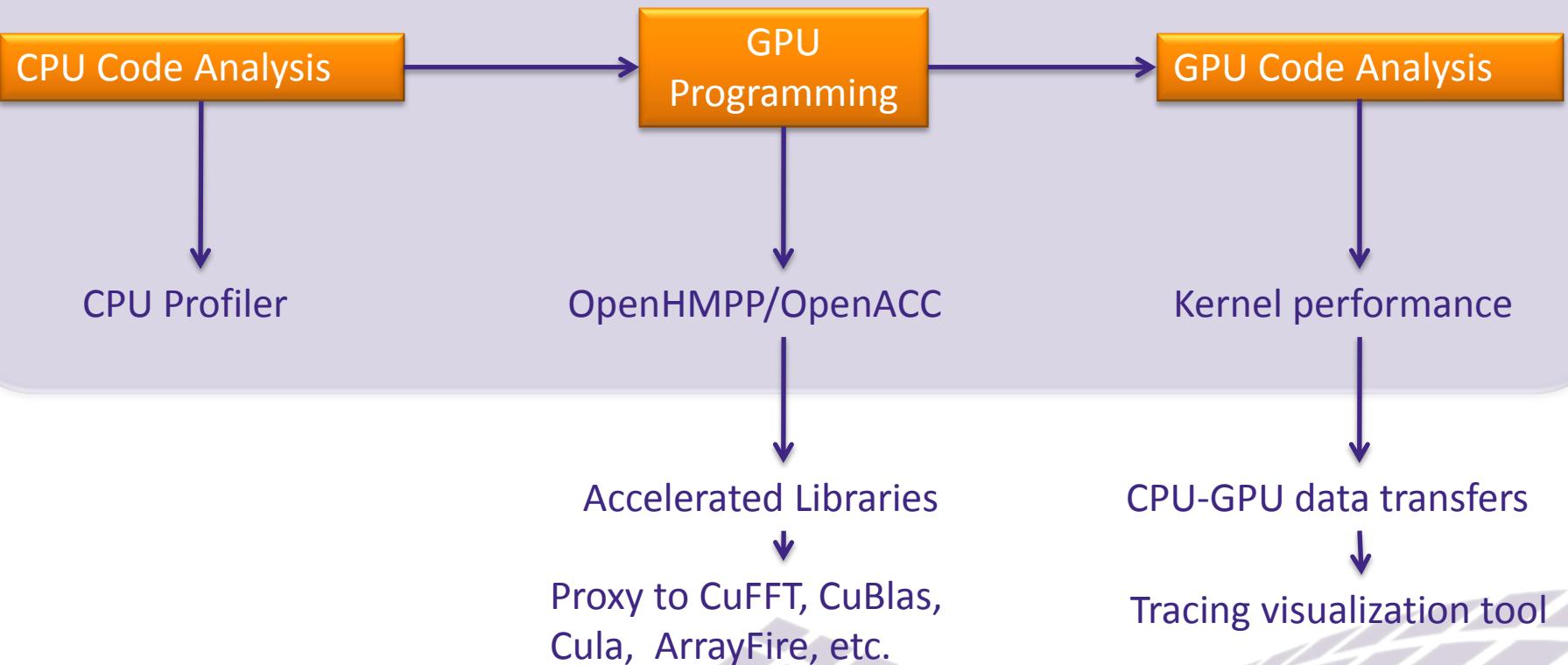
- Output:

```
+-----+
| nx=400
| ny=500
| nxystep=500
| tend=40.000
| nstepmax=81
| noutput=1000000
| dtoutput=0.000
+-----+
202 252
--> Step=    1,    1.33631e-03,  1.33631e-03 {296.098 Mflops 39290168} (0.133s)
--> Step=    2,    2.67261e-03,  1.33631e-03 {332.653 Mflops 39293940} (0.118s)
...
--> Step=   80,    4.56073e-01,  6.74684e-03 {329.866 Mflops 39398490} (0.119s)
--> Step=   81,    4.62848e-01,  6.77480e-03 {318.822 Mflops 39399392} (0.124s)
Hydro ends in 00:00:10.307s (10.307).
```

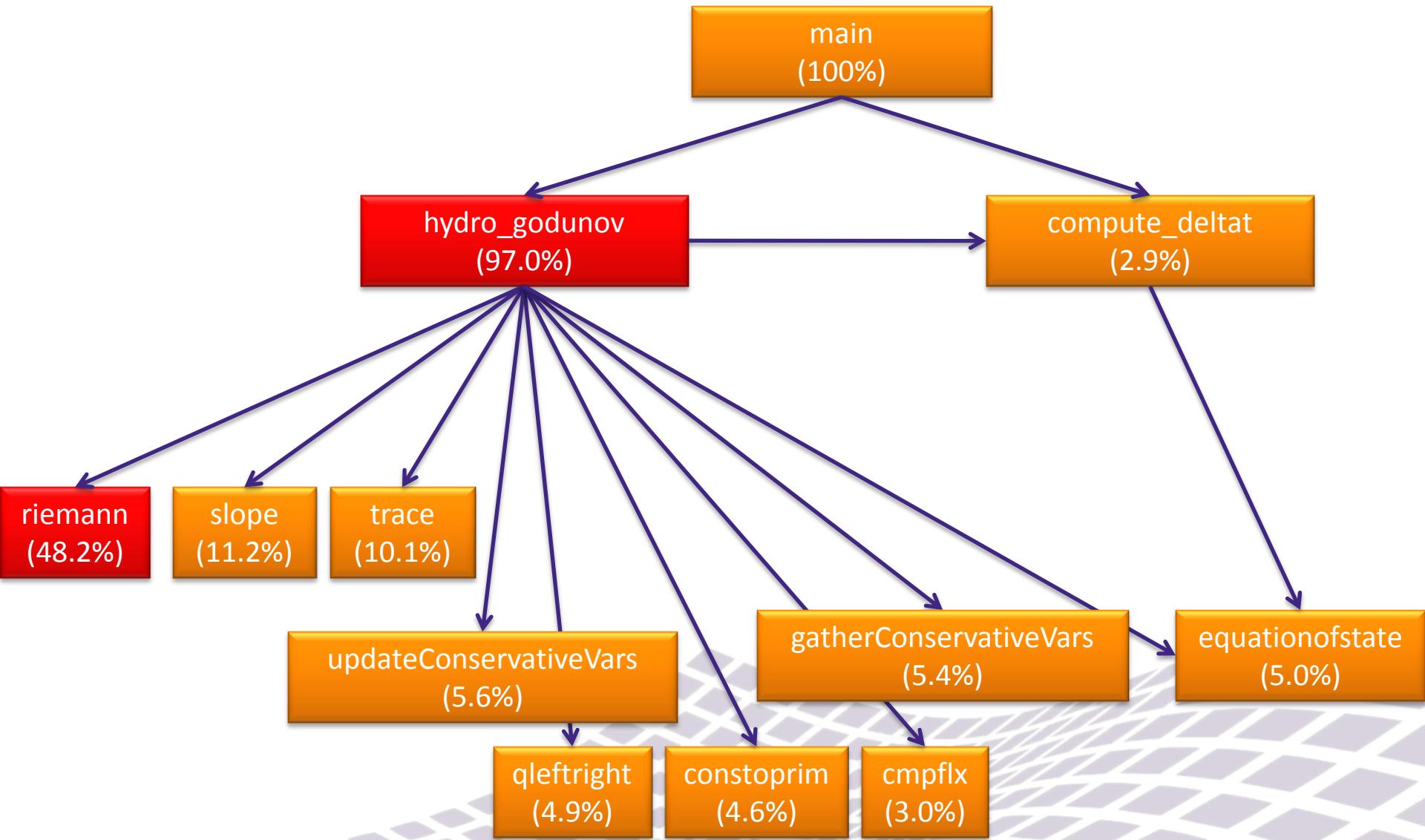
HydroC Migration Steps



Migration Process, Tool View



HydroC Call-Graph



Profiling



- Profile of the CPU version application using gprof

Flat profile:

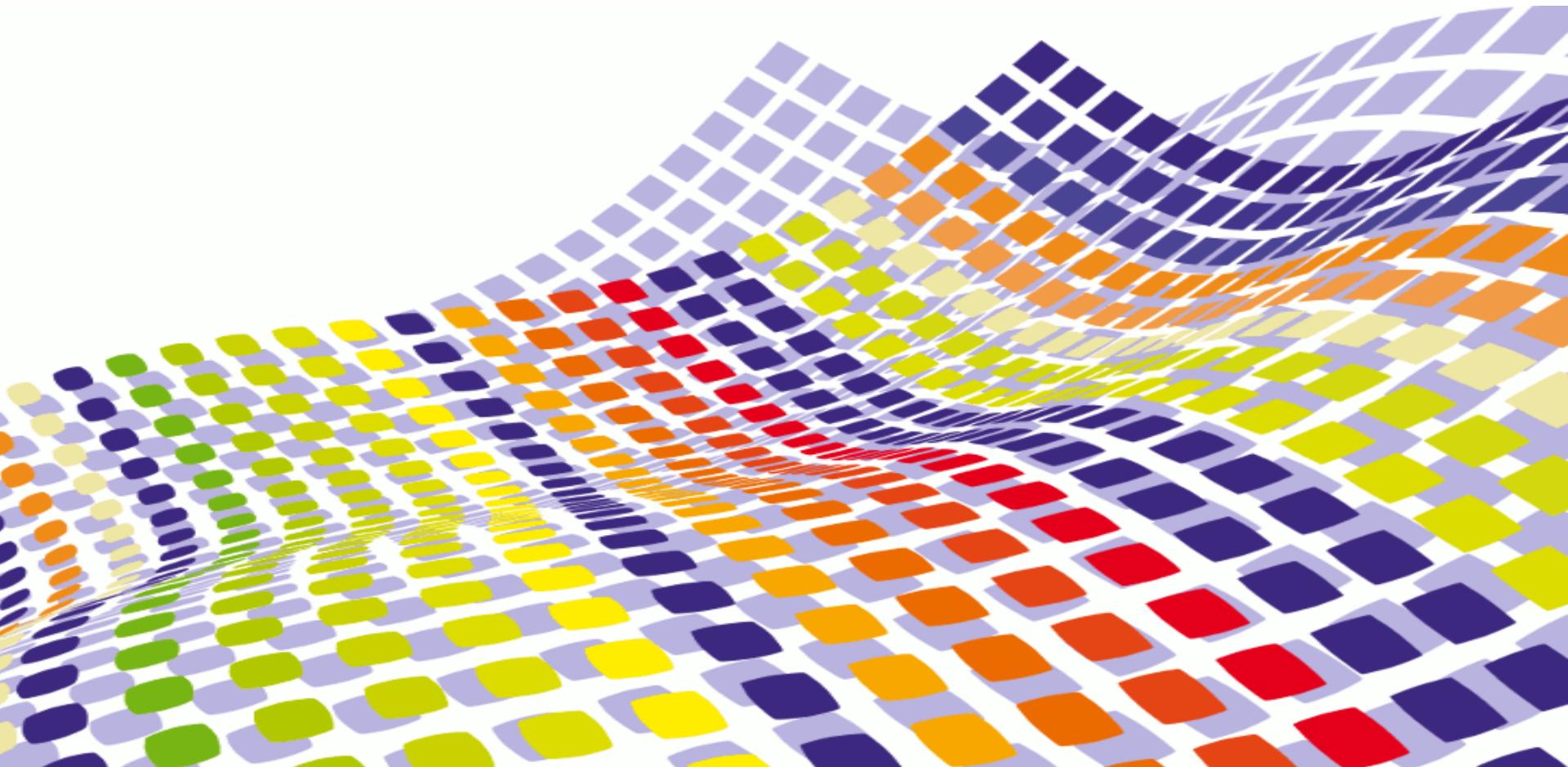
We start here for code migration

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
48.24	4.77	4.77	162	29.45	29.45	riemann
11.23	5.88	1.11	162	6.85	6.85	slope
10.11	6.88	1.00	162	6.17	6.17	trace
5.56	7.43	0.55	162	3.40	3.40	updateConservativeVars
5.36	7.96	0.53	162	3.27	3.27	gatherConservativeVars
4.96	8.45	0.49	203	2.41	2.41	equation_of_state
4.85	8.93	0.48	162	2.96	2.96	qleftright
4.55	9.38	0.45	162	2.78	2.78	constoprim
3.03	9.68	0.30	162	1.85	1.85	cmpflx
1.92	9.87	0.19	41	4.64	7.05	compute_deltat
0.10	9.88	0.01	162	0.06	0.06	make_boundary
0.10	9.89	0.01	8	1.25	1.25	ToBase64
0.00	9.89	0.00	891	0.00	0.00	Dmalloc
0.00	9.89	0.00	164	0.00	0.00	cclock
0.00	9.89	0.00	81	0.00	0.00	Imalloc
0.00	9.89	0.00	81	0.00	0.00	allocate_work_space

...

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

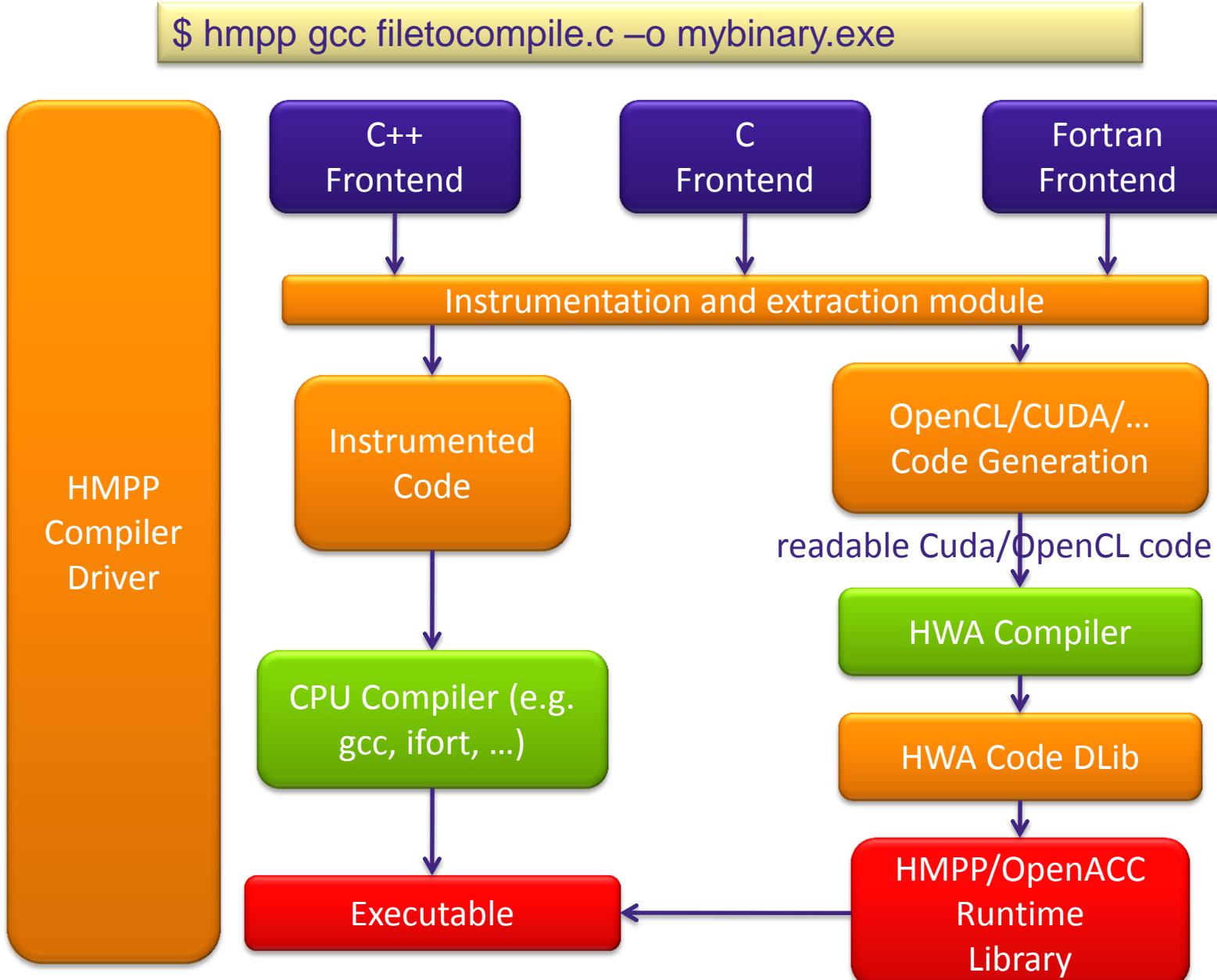


- Functions based, i.e. codelet (OpenHMPP)
- Code regions based (OpenACC)

```
#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];
}
```

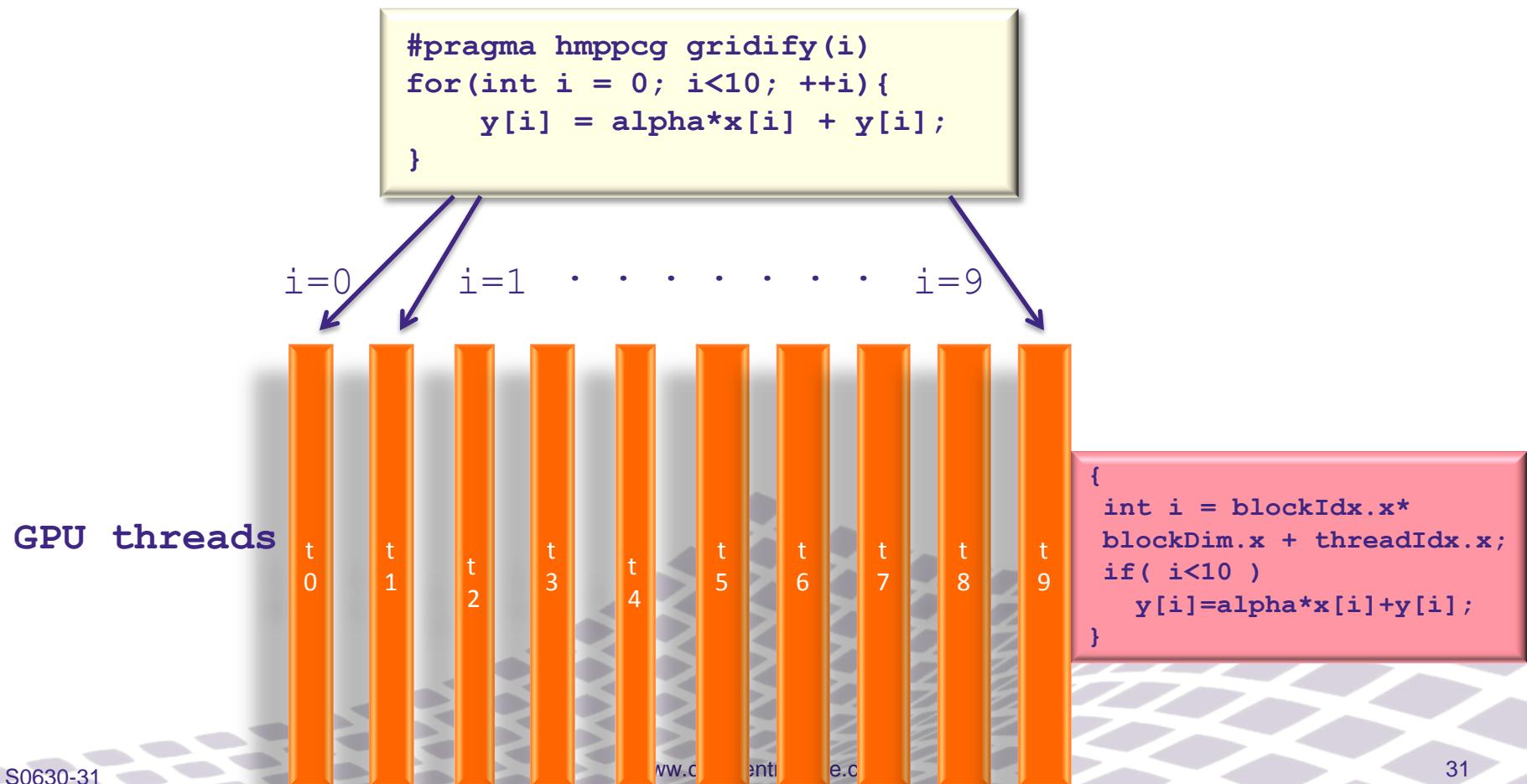
```
#pragma acc kernels ...
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
```

HMPP/OpenACC Compiler flow (Src to Src Cpler)



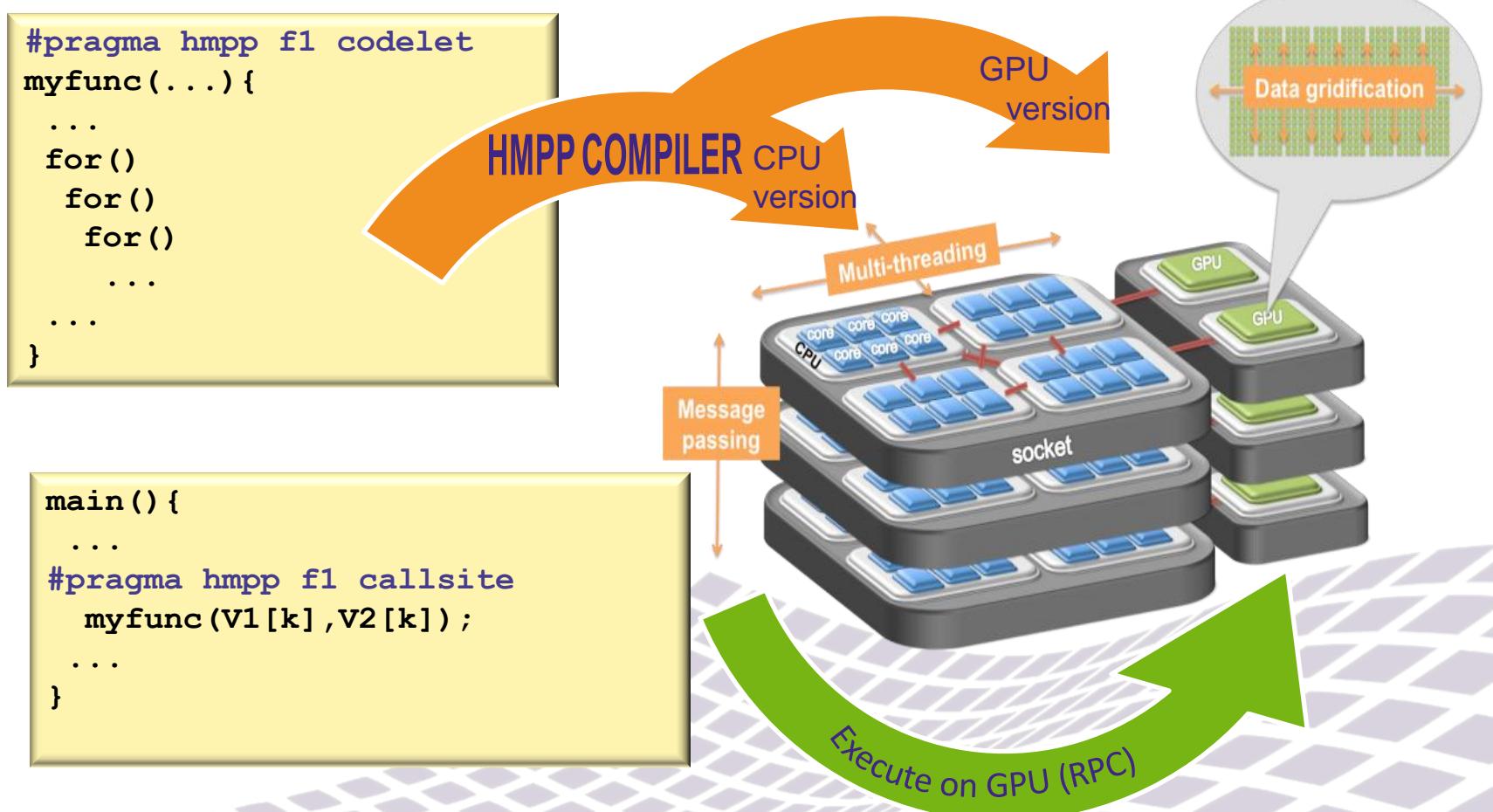
Code Generation based on "Loop Nest Gridification" CAPS

- 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



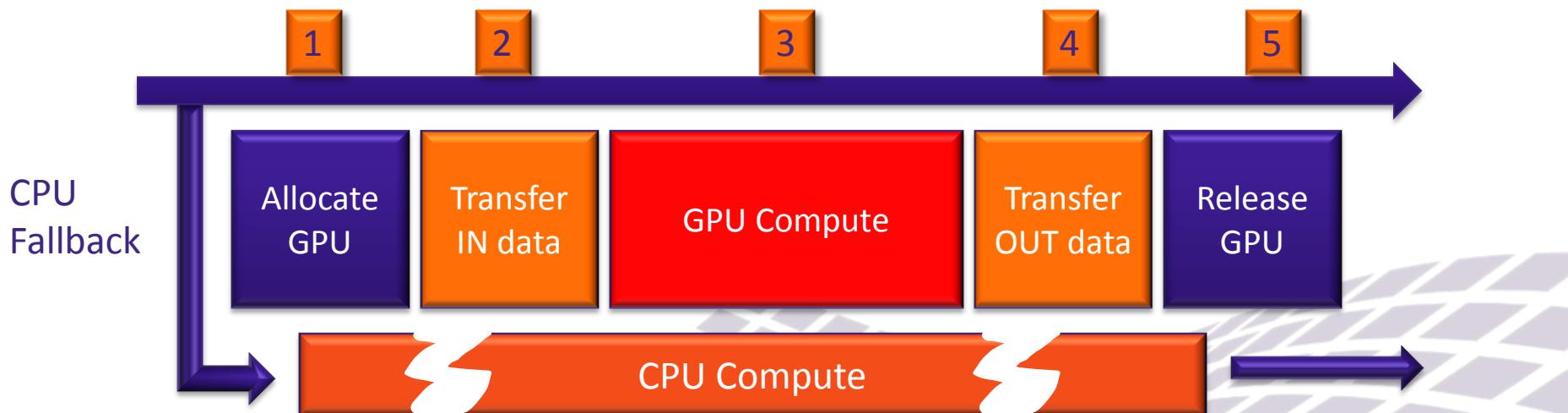
Remote Procedure Call on an Accelerator

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

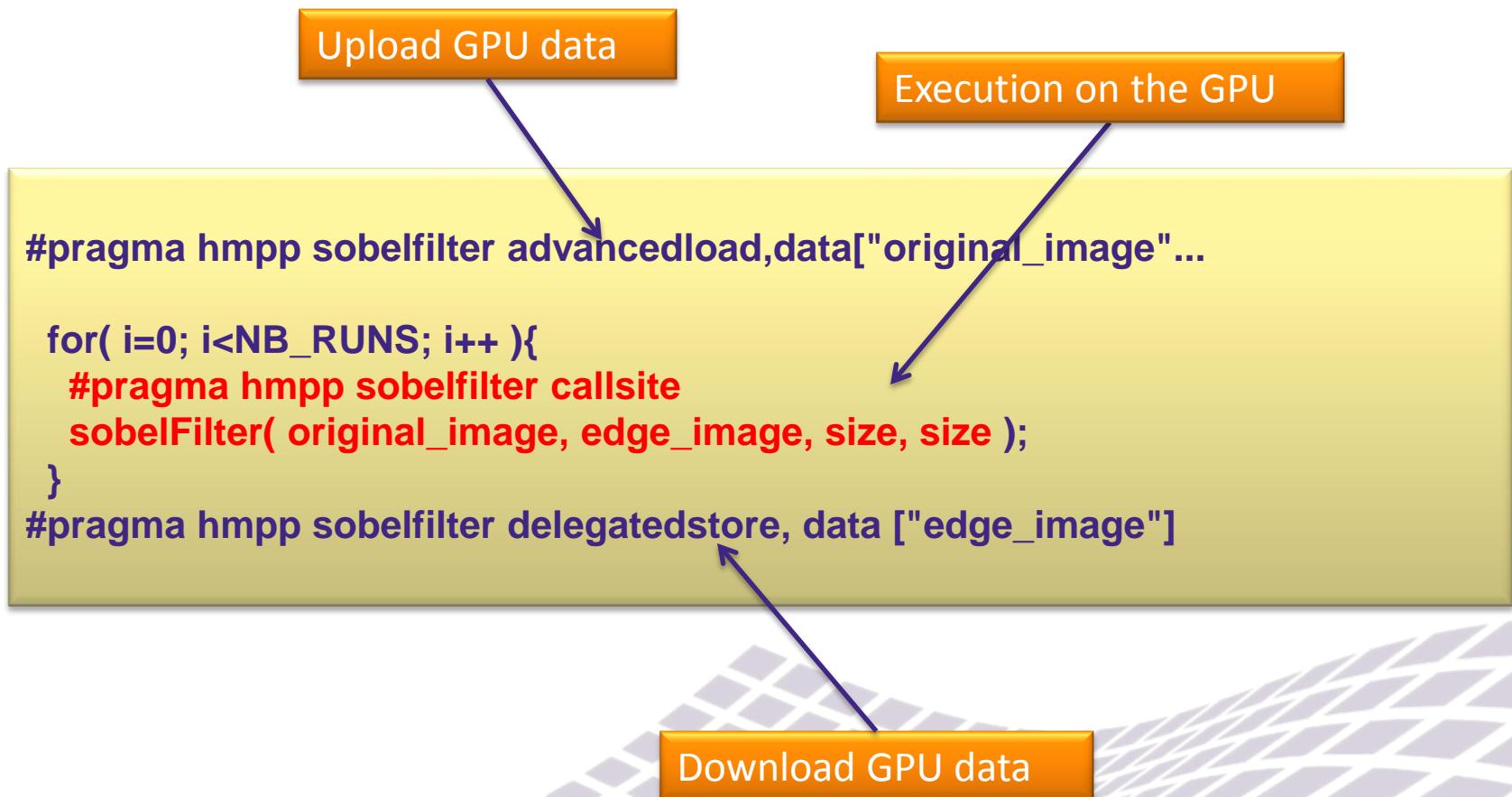


Remote Procedure Call Basic Operations

- 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

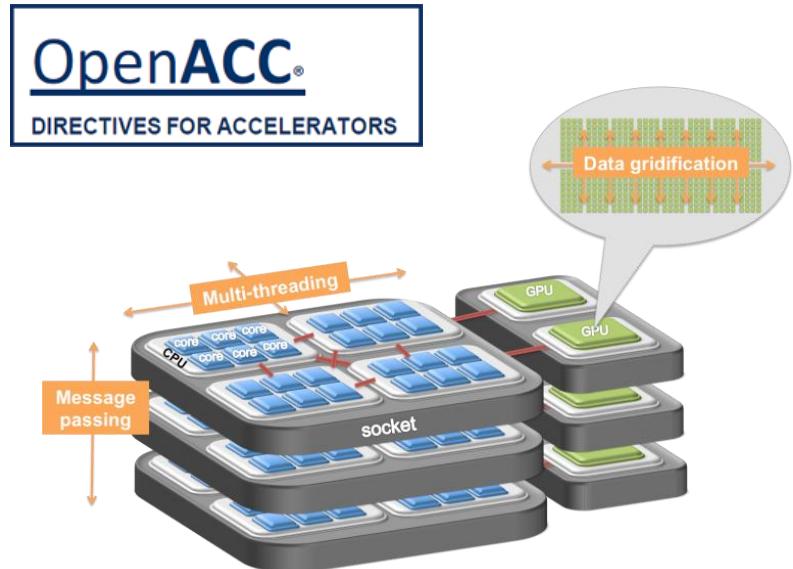


RPC Example (Codelet Style)



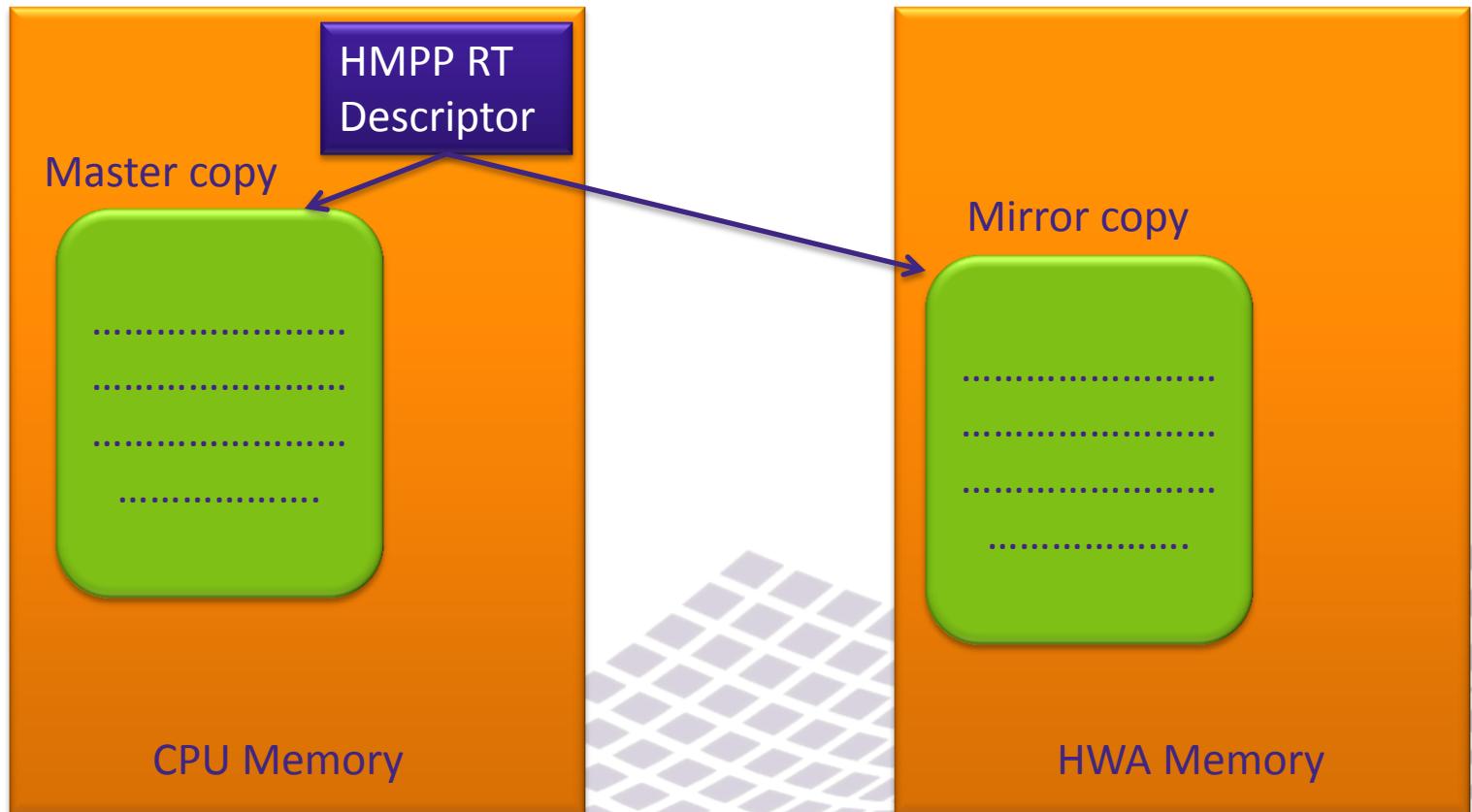
OpenACC Initiative

- 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



OpenACC Execution Model

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



Parallel Loops

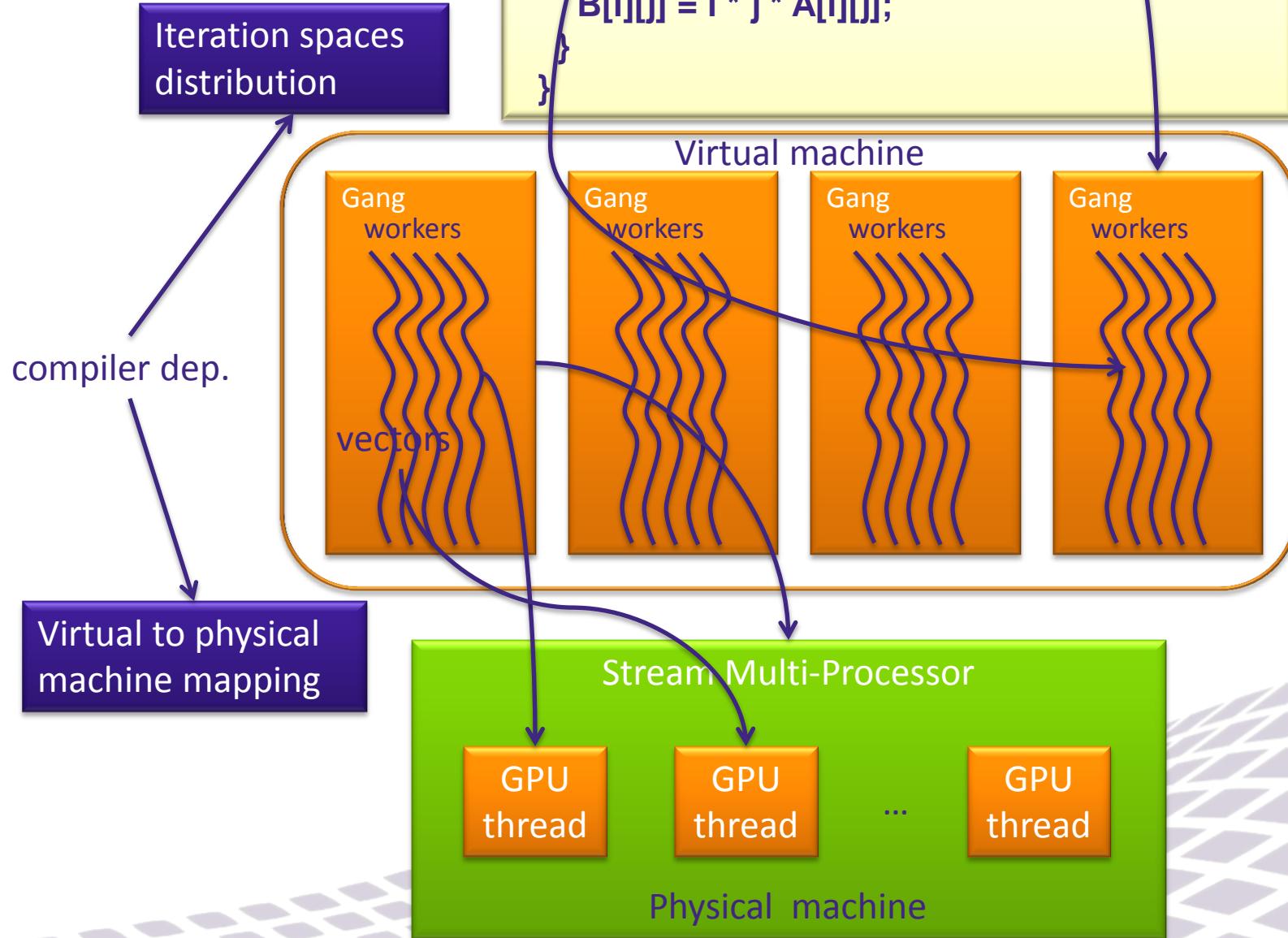
- The loop directive describes iteration space partitioning to execute the loop; declares loop-private variables and arrays, and reduction operations
- Clauses
 - gang [(scalar-integer-expression)]
 - worker [(scalar-integer-expression)]
 - vector [(scalar-integer-expression)]
 - collapse(*n*)
 - seq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



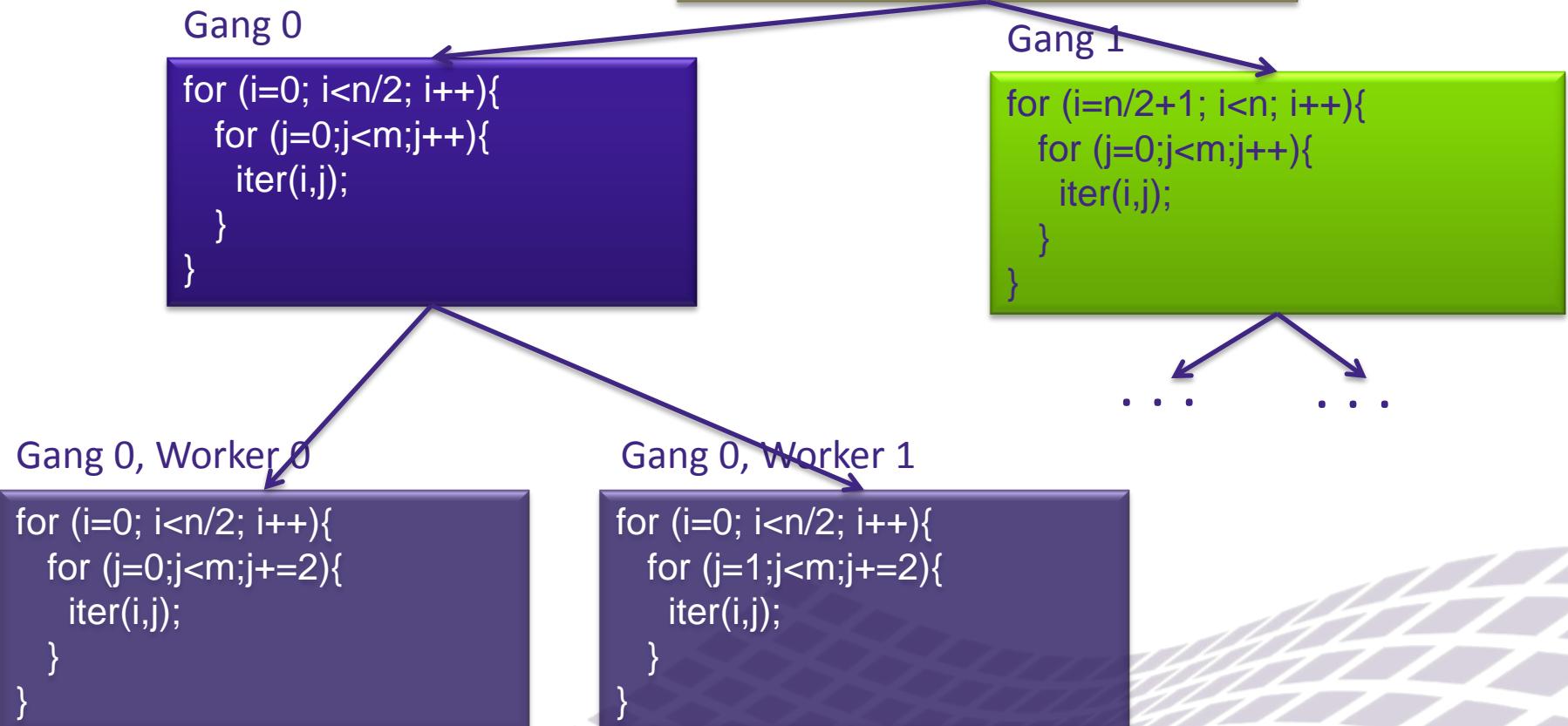
Kernel Regions

- 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-expression)]
 - 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];
    }
}
```

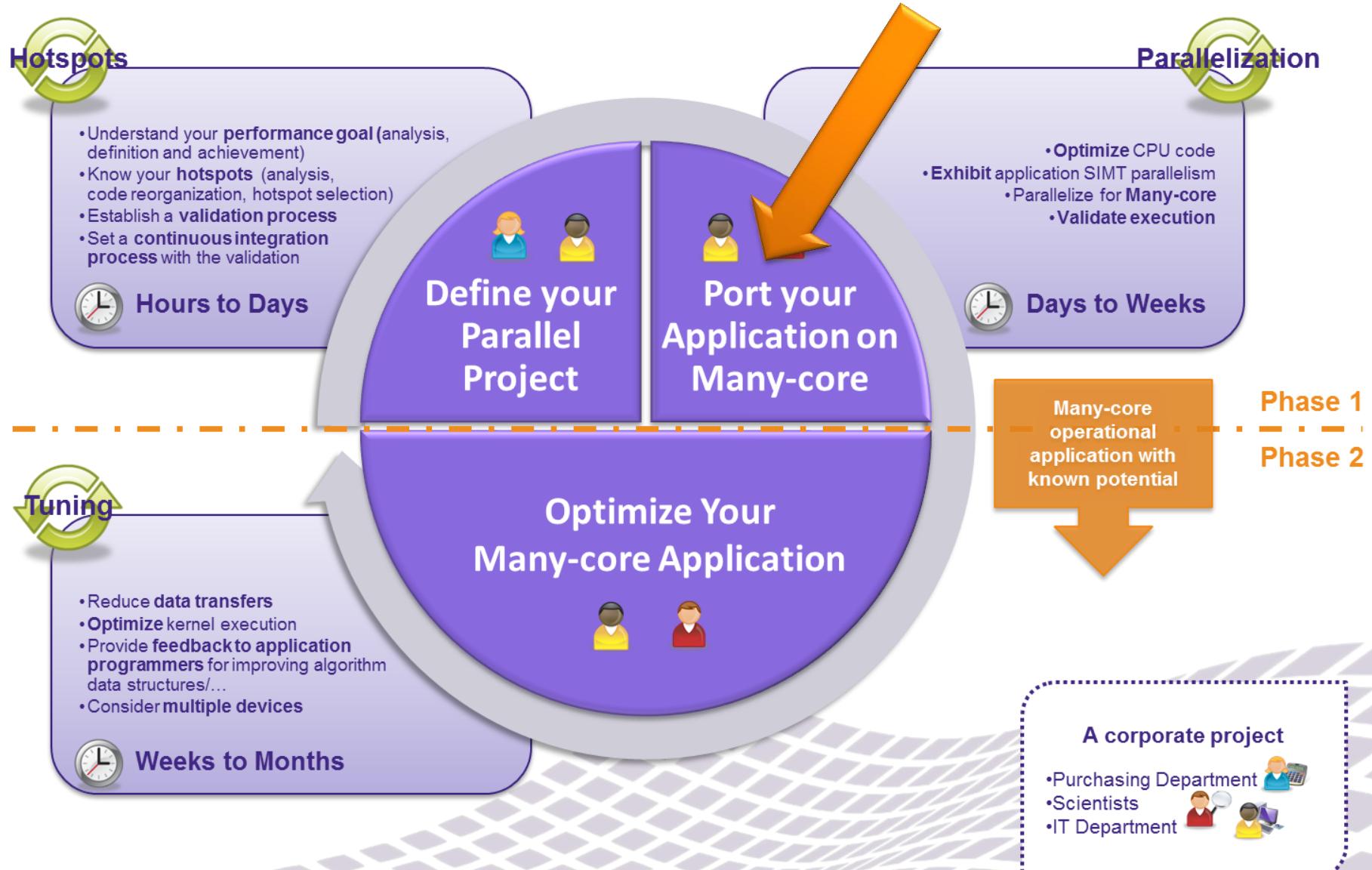
Iterations Mapping

```
#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);
    }
}
```



Distribution scheme is compiler dependant (here simplified scheme)

HydroC OpenACC Migration Step



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

Porting the Riemann Hotspot

- CAPS OpenACC compiler feedback

...

hmppcg: [Message DPL3000] riemann.c:86: Loop 's' was *gridified (1D)*

...

hmppcg: [Message DPL3000] riemann.c:266: Loop 'invar' was *gridified (1D)*

...

- Indicating successful parallel code generation

Porting the Riemann Hotspot

- Code execution feedback

```
[ 0.275872] ( 0) INFO : Enter kernels (queue=none, location=riemann.c:79)
[ 0.275886] ( 0) INFO : Allocate qleft[0:1008000] (element_size=8, host_address=0x7f5239905010, memory_space=cudaglob, queue=none, location=riemann.c:79)
[ 0.275962] ( 0) INFO : Upload qleft[0:1008000] (element_size=8, host_address=0x7f5239905010, queue=none, location=riemann.c:79)
[ 0.277318] ( 0) INFO : Allocate qright[0:1008000] (element_size=8, host_address=0x7f523914c010, memory_space=cudaglob, queue=none, location=riemann.c:79)
[ 0.277465] ( 0) INFO : Upload qright[0:1008000] (element_size=8, host_address=0x7f523914c010, queue=none, location=riemann.c:79)
[ 0.278861] ( 0) INFO : Allocate qgdnv[0:1008000] (element_size=8, host_address=0x7f5238993010, memory_space=cudaglob, queue=none, location=riemann.c:79)
[ 0.278997] ( 0) INFO : Upload qgdnv[0:1008000] (element_size=8, host_address=0x7f5238993010, queue=none)
[ 0.280249] ( 0) INFO : Allocate sgnm[0:252000] (element_size=4, host_address=0x197bd90, memory_space=cudaglob, queue=none, location=riemann.c:79)
[ 0.280324] ( 0) INFO : Upload sgnm[0:252000] (element_size=4, host_address=0x197bd90, queue=none, location=riemann.c:79)
...
[ 0.281202] ( 0) INFO : Call __hmpp_acc_region__54seznto (queue=none, location=riemann.c:79)
...
[ 0.285177] ( 0) INFO : Download sgnm[0:252000] (element_size=4, host_address=0x197bd90, queue=none, location=riemann.c:79)
[ 0.285775] ( 0) INFO : Free sgnm[0:252000] (element_size=4, host_address=0x197bd90, queue=none, location=riemann.c:79)
[ 0.285853] ( 0) INFO : Download qgdnv[0:1008000] (element_size=8, host_address=0x7f5238993010, queue=none, location=riemann.c:79)
[ 0.289225] ( 0) INFO : Free qgdnv[0:1008000] (element_size=8, host_address=0x7f5238993010, queue=none, location=riemann.c:79)
[ 0.289390] ( 0) INFO : Download qright[0:1008000] (element_size=8, host_address=0x7f523914c010, queue=none, location=riemann.c:79)
[ 0.290886] ( 0) INFO : Free qright[0:1008000] (element_size=8, host_address=0x7f523914c010, queue=none, location=riemann.c:79)
[ 0.291045] ( 0) INFO : Download qleft[0:1008000] (element_size=8, host_address=0x7f5239905010, queue=none, location=riemann.c:79)
[ 0.292503] ( 0) INFO : Free qleft[0:1008000] (element_size=8, host_address=0x7f5239905010, queue=none)
[ 0.292659] ( 0) INFO : Leave kernels (queue=none, location=riemann.c:79)
```

Allocate and copy *qleft* from host to device

Computation

Copy *qleft* from device to host and free buffer

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-expression)]
 - **num_gangs**(scalar-integer-expression)
 - **num_workers**(scalar-integer-expression)
 - **vector_length**(scalar-integer-expression)
 - **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)

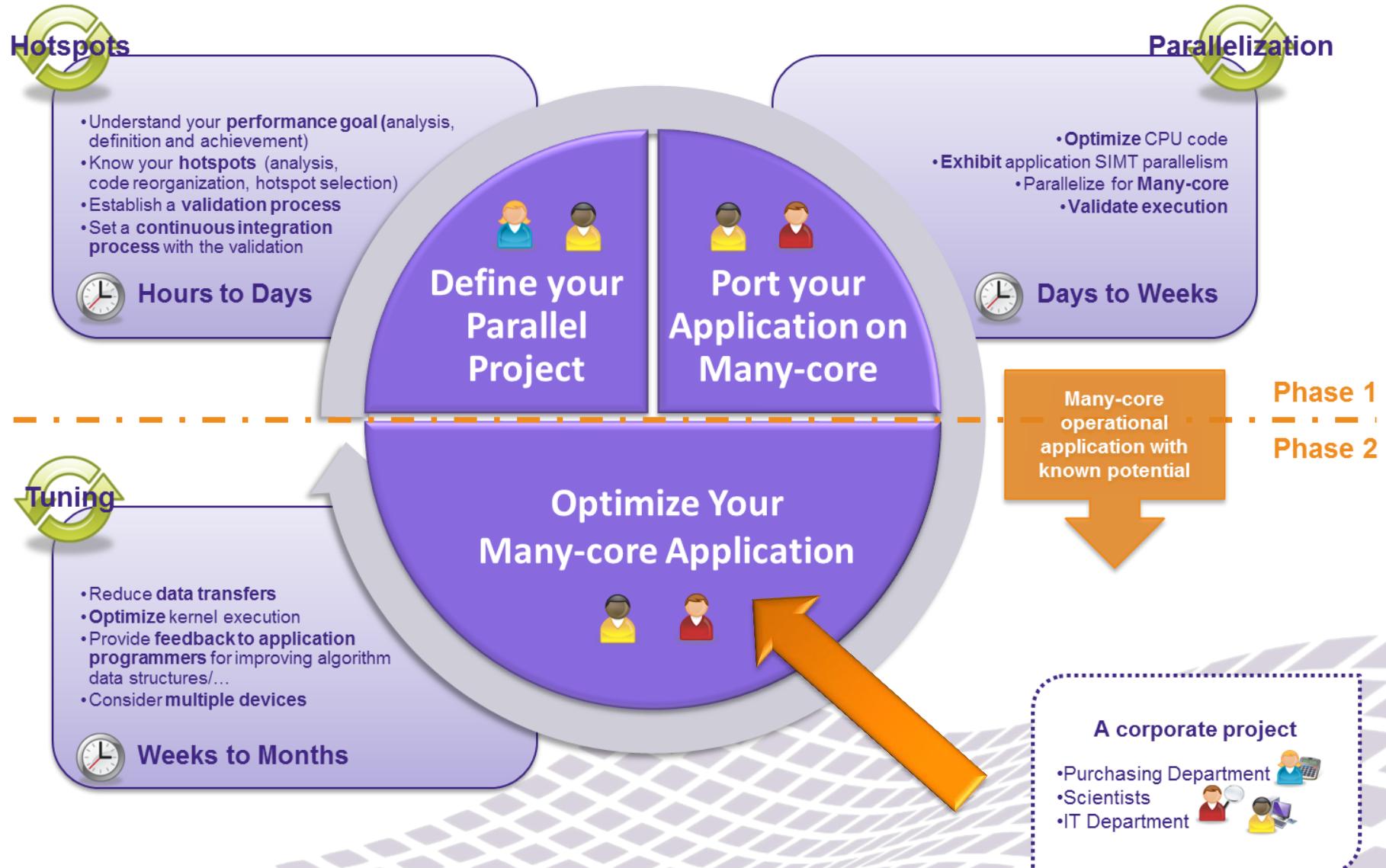
```
#pragma acc data copyin(A[1:N-2]),
    copyout(B[N])
{
    #pragma acc kernels
    {
        #pragma acc loop independant
        for (int i = 0; i < N; ++i){
            A[i][0] = ...;
            A[i][M - 1] = 0.0f;
        }
        ...
    }
    #pragma acc update host(A)
    ...
    #pragma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



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, Hsmal1r, Hsmal1c, 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

Riemann Kernel Optimizations

- CAPS OpenACC compiler feedback

...

hmppcg: [Message DPL3001] riemann.c:87: Loops 'i' and 's' were gridified (**2D**)

...

hmppcg: [Message DPL3001] riemann.c:267: Loops 's' and 'invar' were gridified (**2D**)

...

- Indicating effective parallelization

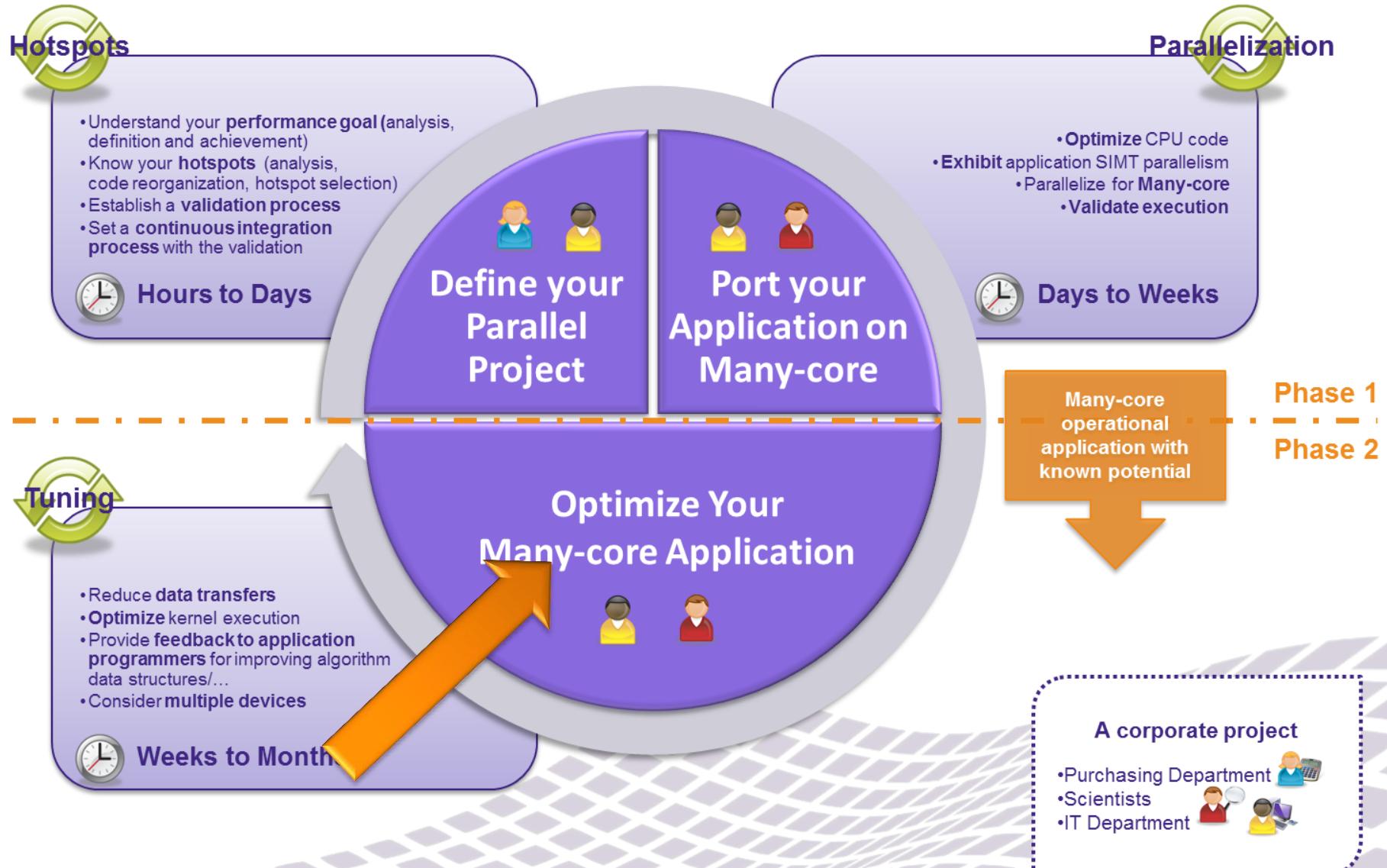
Porting the Riemann Hotspot



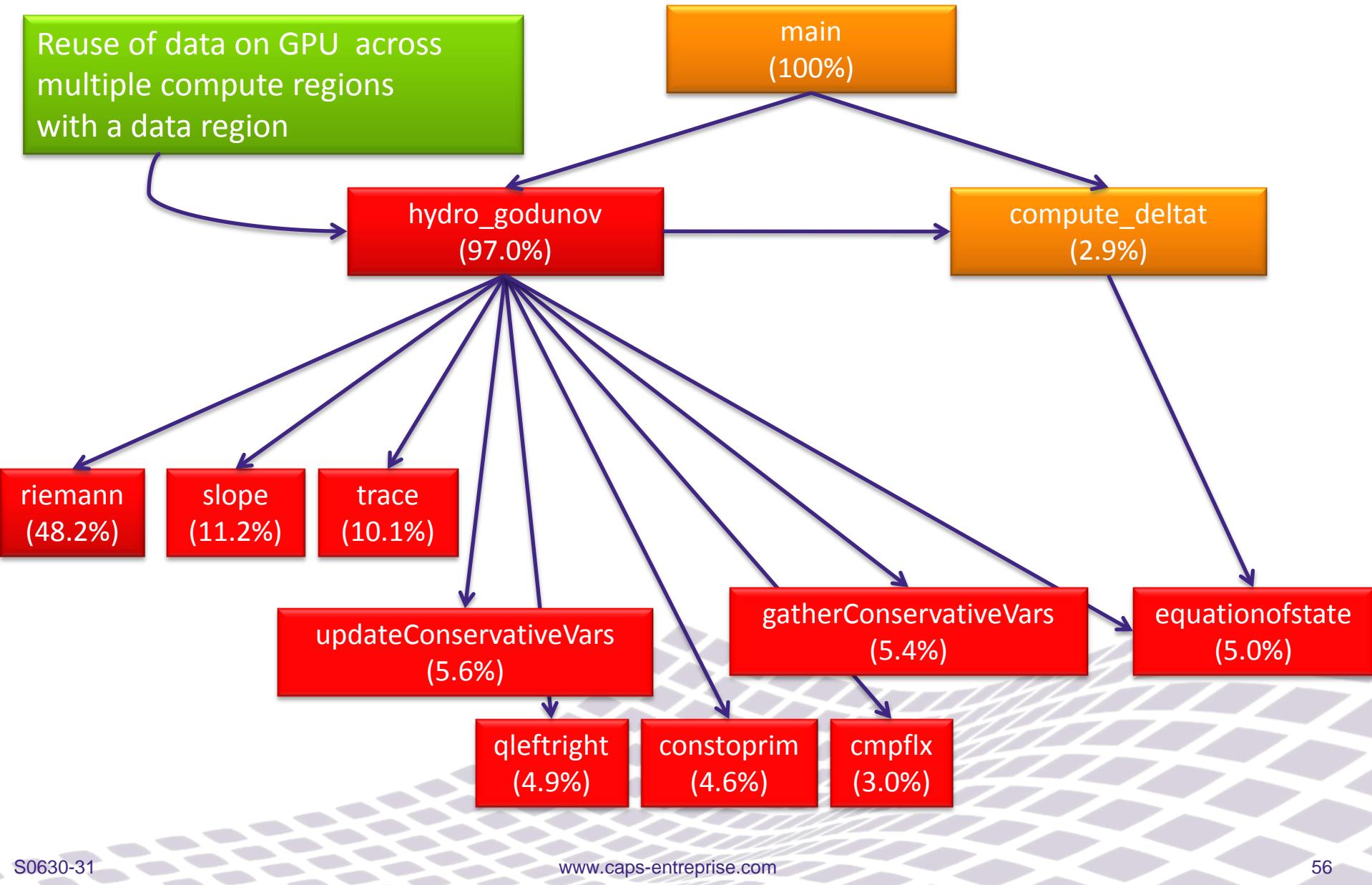
CAPS

- 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



Further Optimizations



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

D

CAPS

- With the same strategy, the full application have been ported with OpenACC
- The following hotspots have been accelerated
 - cmplx
 - updateConservativeVar
 - gatherConservativeVar
 - constoprime
 - 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

D

CAPS

```
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;

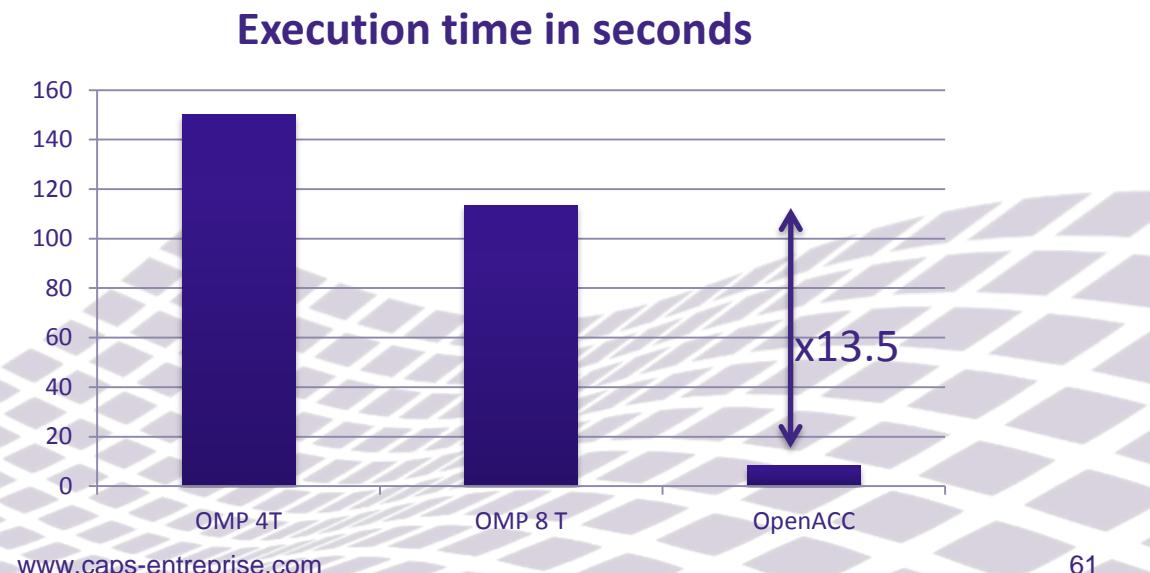
        #pragma acc loop gang(16)
        for (int i = 1; i < M - 1; ++i) {
            #pragma acc loop worker(16)
            for (int j = 1; j < N - 1; ++j) {
                B[i][j] =c11*A[i-1][j-1]+c12*A[i+0][j-1]+c13*A[i+1][j-1]
                           +c21*A[i-1][j+0]+c22*A[i+0][j+0]+c23*A[i+1][j+0]
                           +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/](http://competencecenter.hmpp.org/category/hmpp-coc-asia/)



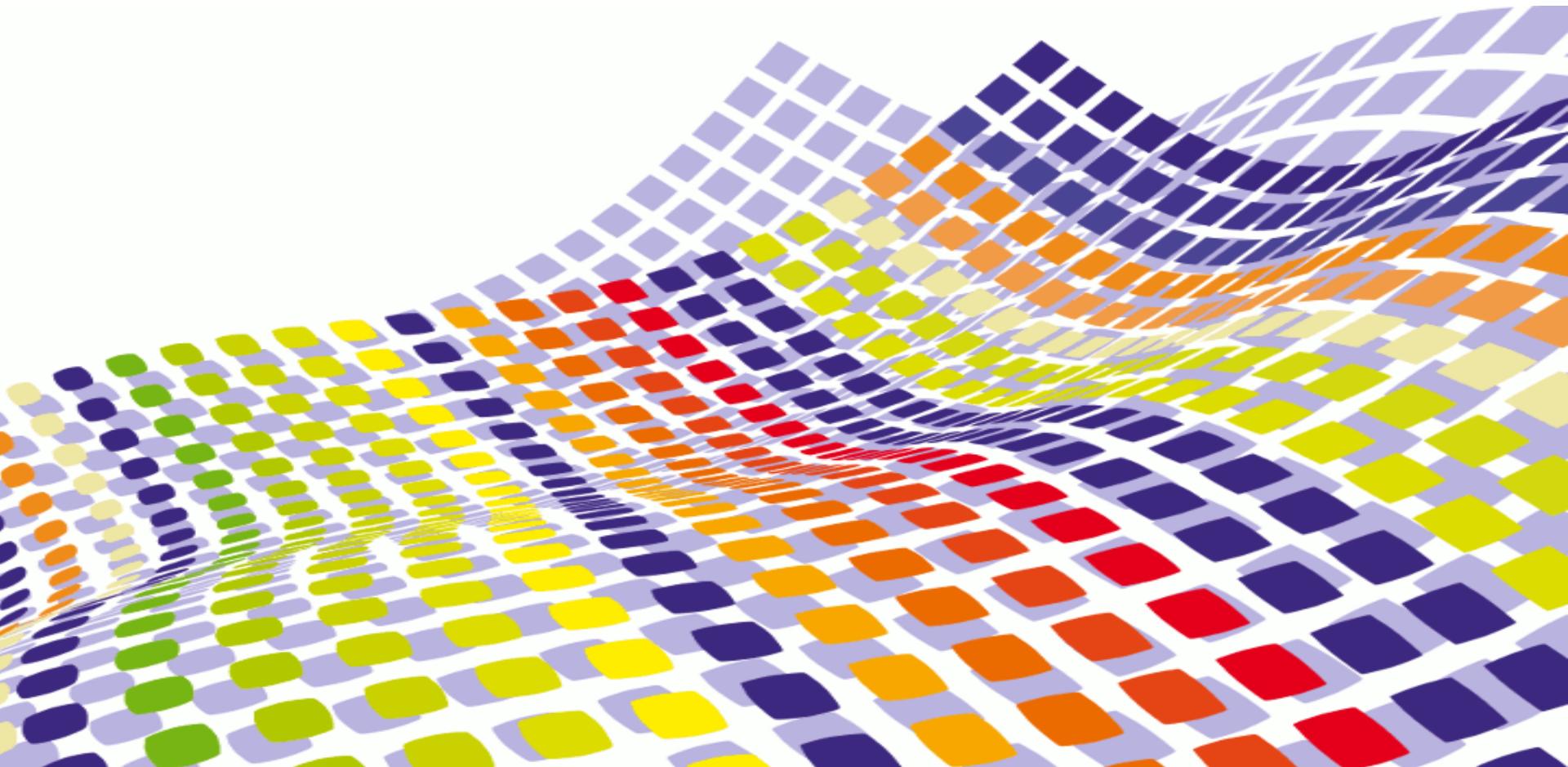
Sobel Filter Performance Example

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



Going Further with OpenHMPP Compiler

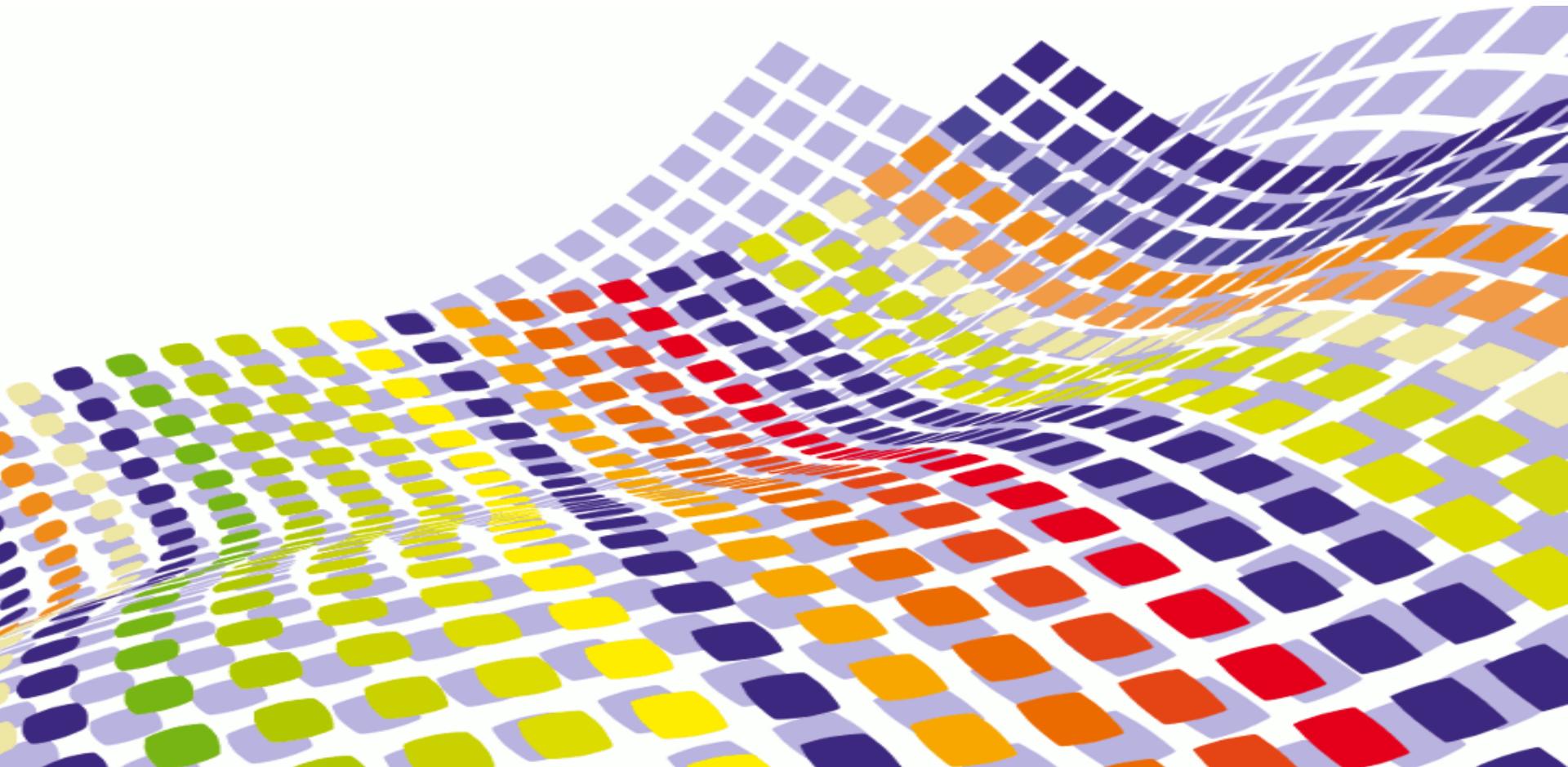


What is in OpenHMPP and not in OpenACC



- 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)
```

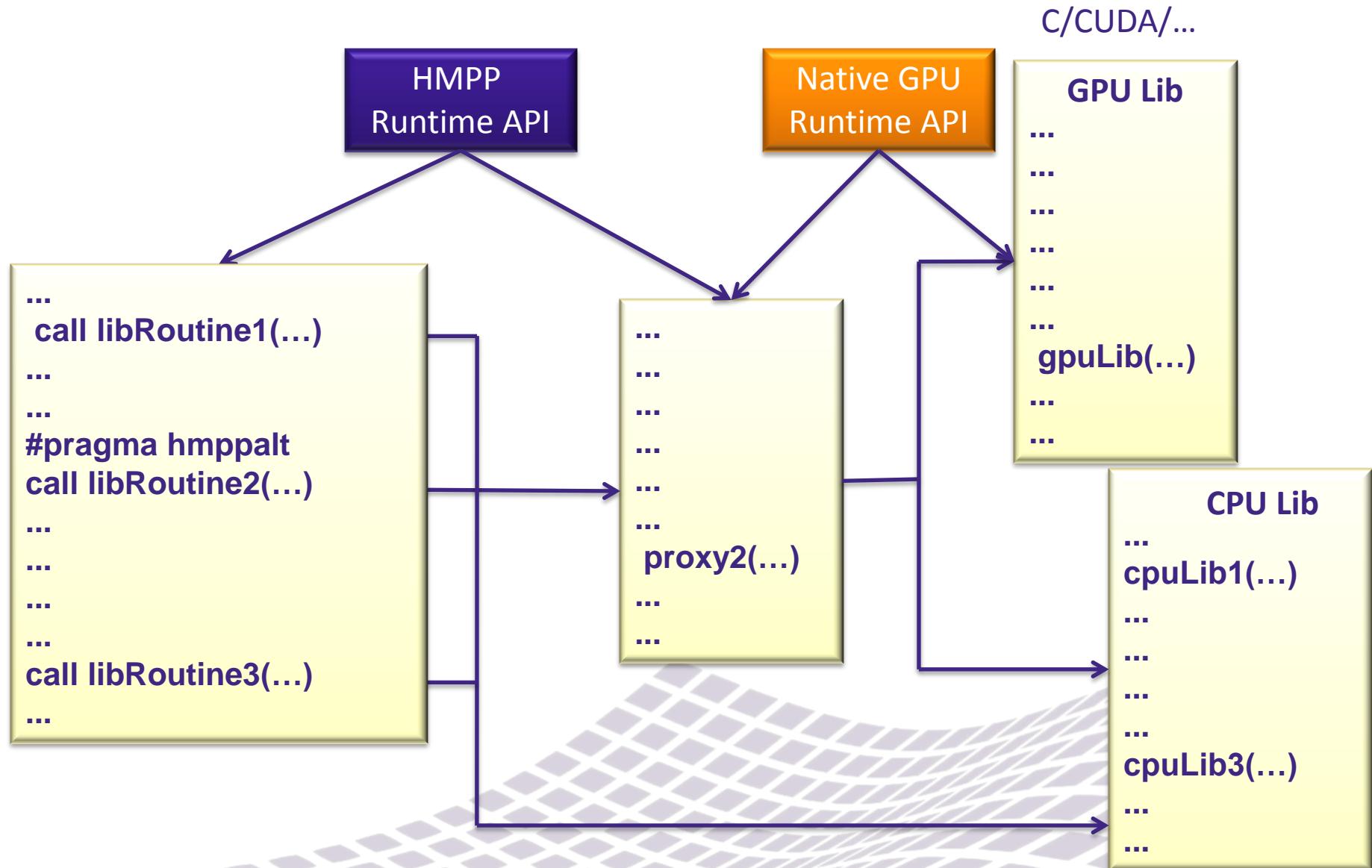
```
!$hmppalt ffte call , name="zfft1d", error="proxy_err"
CALL ZFFT1D(A,N,-1,B)
CALL DUMP(A,N)
```

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



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

D

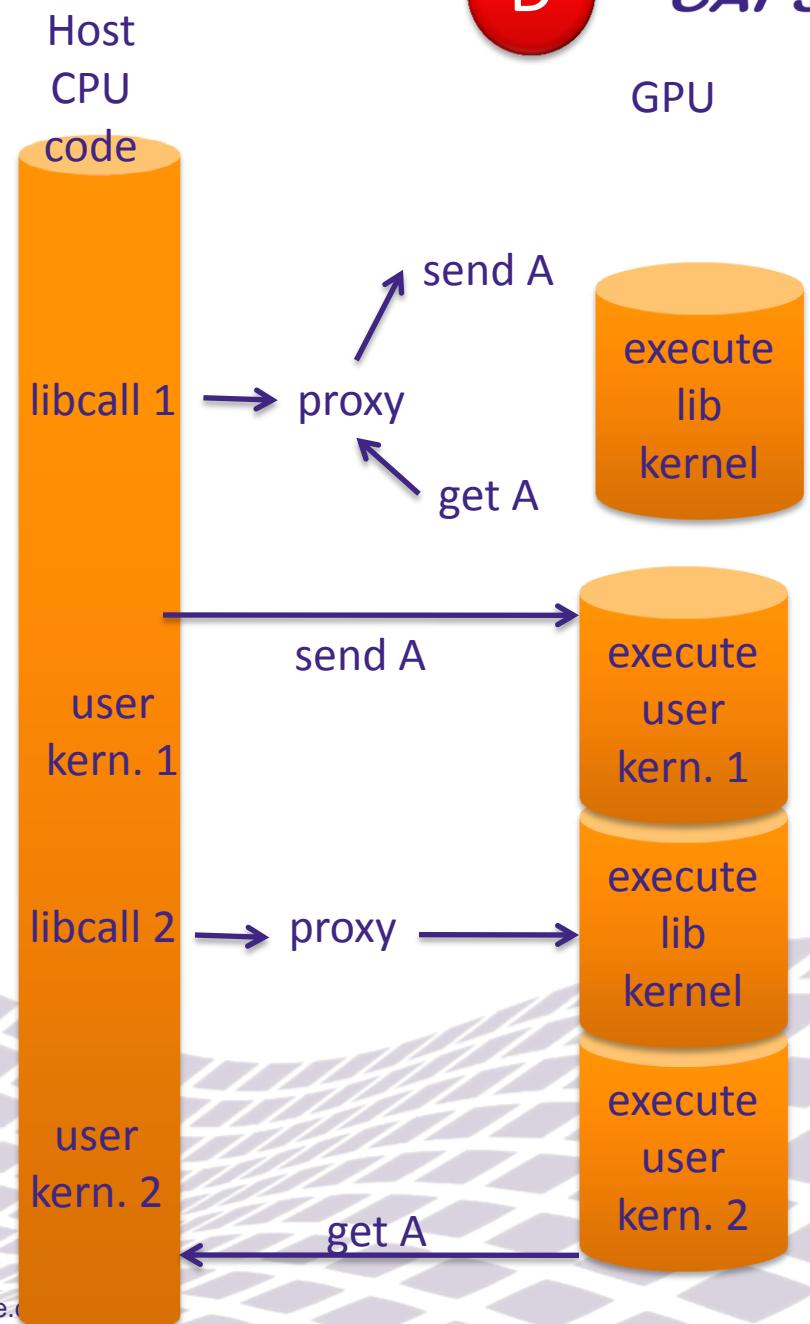
CAPS

- Library level CPU-GPU data transfer management

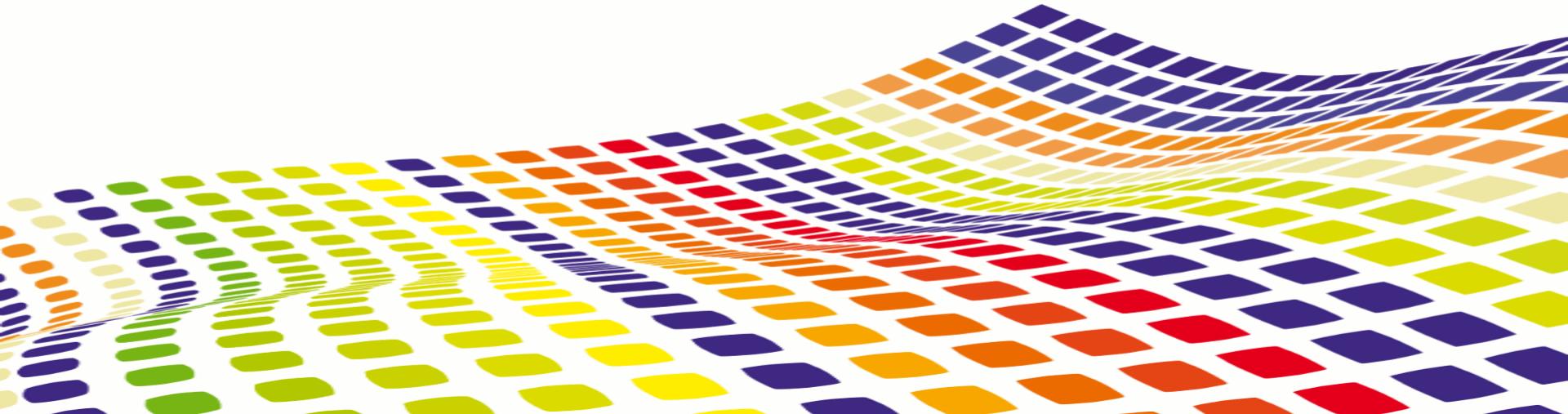
- Data updates are performed inside the proxies
 - Can be redundant with accelerated users code

- 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

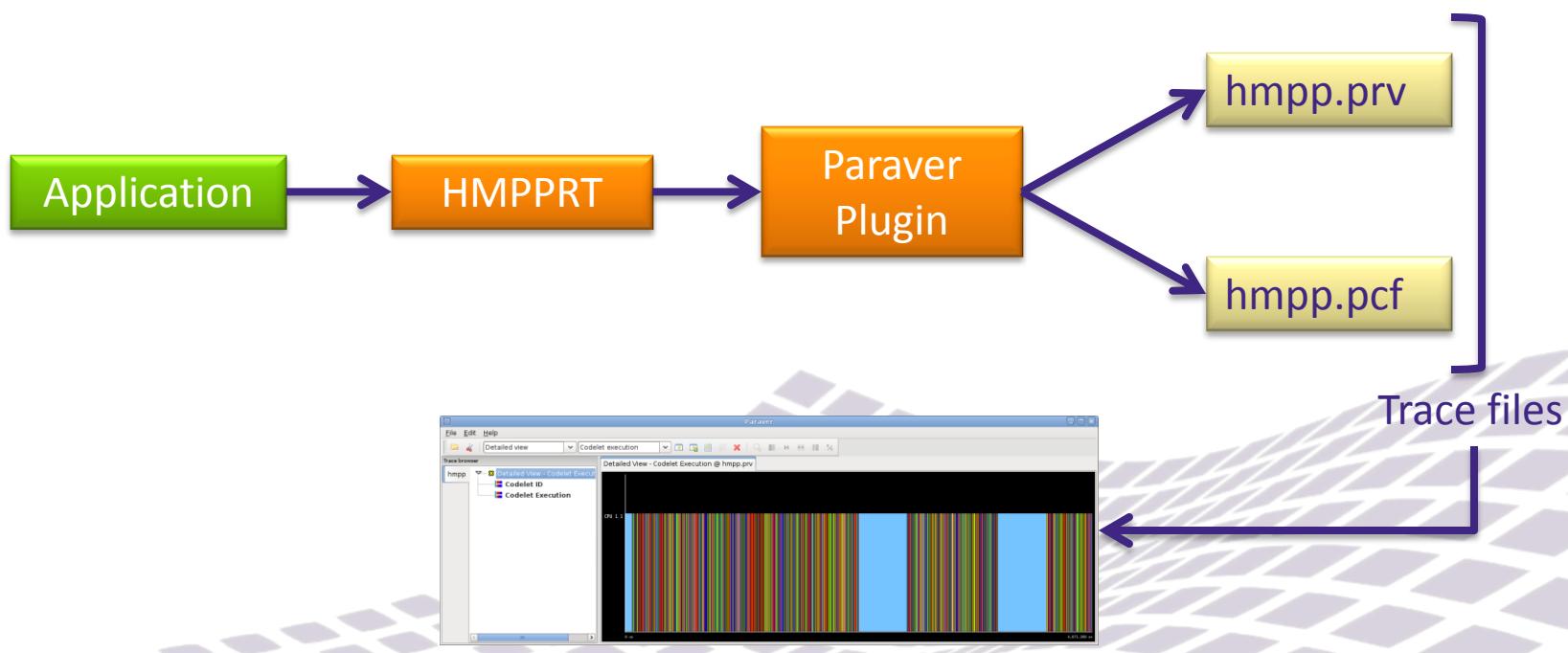


Profiling with Paraver



Profiling with Paraver – Overview

- Paraver is a trace visualization tool developed by the *Barcelona Supercomputing Center*
- HMPP 3.x supports application profiling through additional plugins



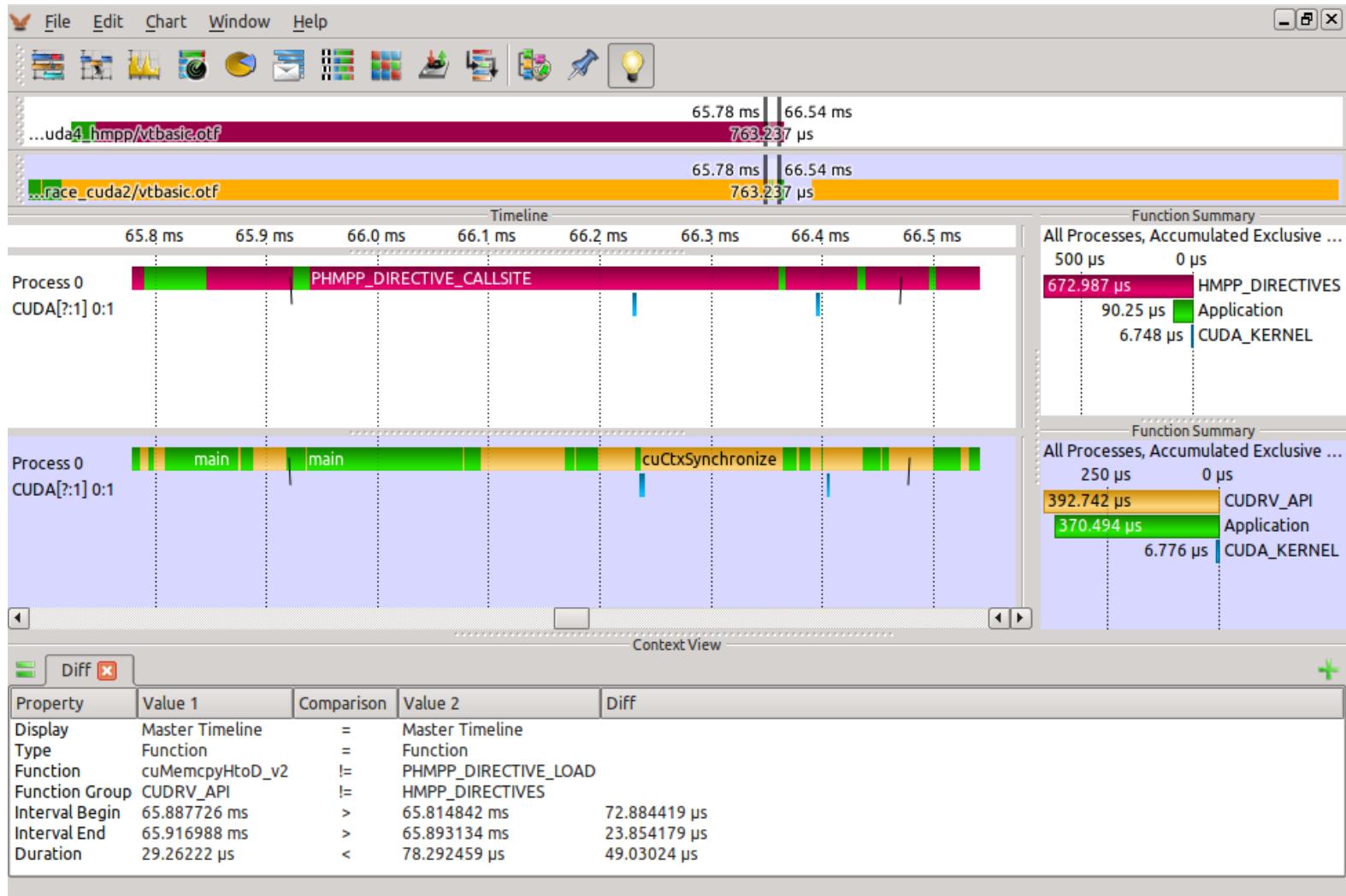
Profiling with Paraver – Filters



- Paraver is able to apply filters on the trace files to retrieve specific information
- Dedicated filters ease GPGPU porting for HMPP applications

Global View		
Intermediate View		
<i>Codelet Execution</i>	<i>Data Transfer</i>	<i>Device Allocation</i>
Detailed View		
<i>Codelet Execution</i>	<i>Data Transfer</i>	<i>Device Allocation</i>

Tracing Interface with Vampir



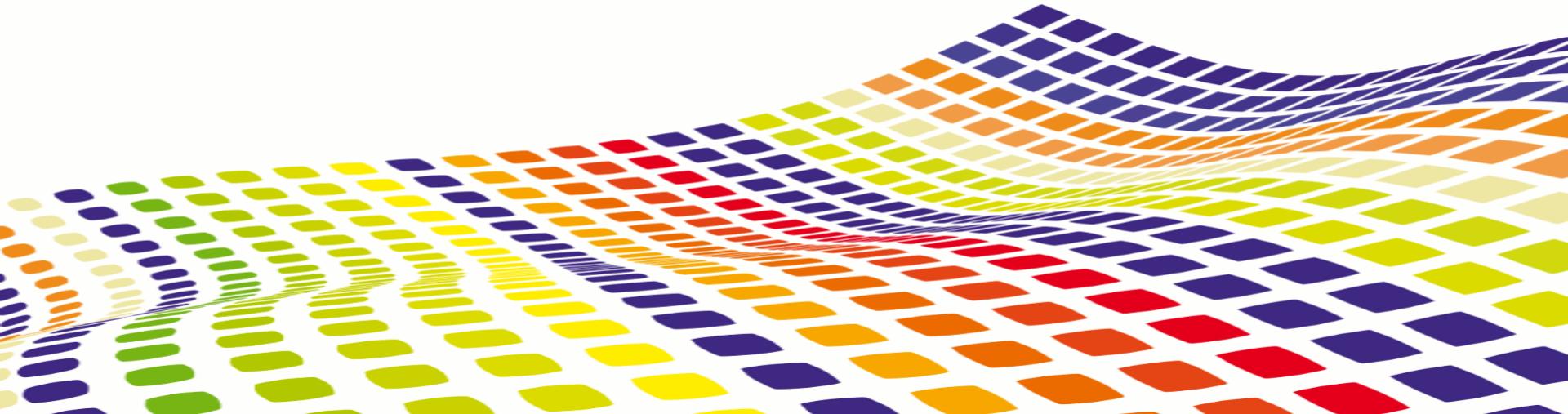
OpenHMPP Tracing Interface



- API specifying callbacks to trace OpenHMPP directives events
- Examples of event query functions
 - `phmpp_event_kind_t phmpp_get_event_kind(phmpp_event_t *event, ...);`
 - `phmpp_event_state_t phmpp_get_event_state(phmpp_event_t *event,...);`
 - `unsigned int phmpp_get_event_instance_id(phmpp_event_t *event, ...);`
 - `unsigned int phmpp_get_event_semantic_id(phmpp_event_t *event,...);`
 - `phmpp_code_t * phmpp_get_event_code(phmpp_event_t * event);`
 - `phmpp_hardware_t *phmpp_get_event_hardware(phmpp_event_t * event);`
 - ...

```
static void phmpp_init(void){  
    phmpp_register_prologue_callback(init);  
    phmpp_register_event_callback(PHMPP_EK_CODELET_REQUEST, codelet_event);  
    phmpp_register_event_callback(PHMPP_EK_TRANSFER_REQUEST, transfer_event);  
    phmpp_register_epilogue_callback(finish);  
}
```

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:**

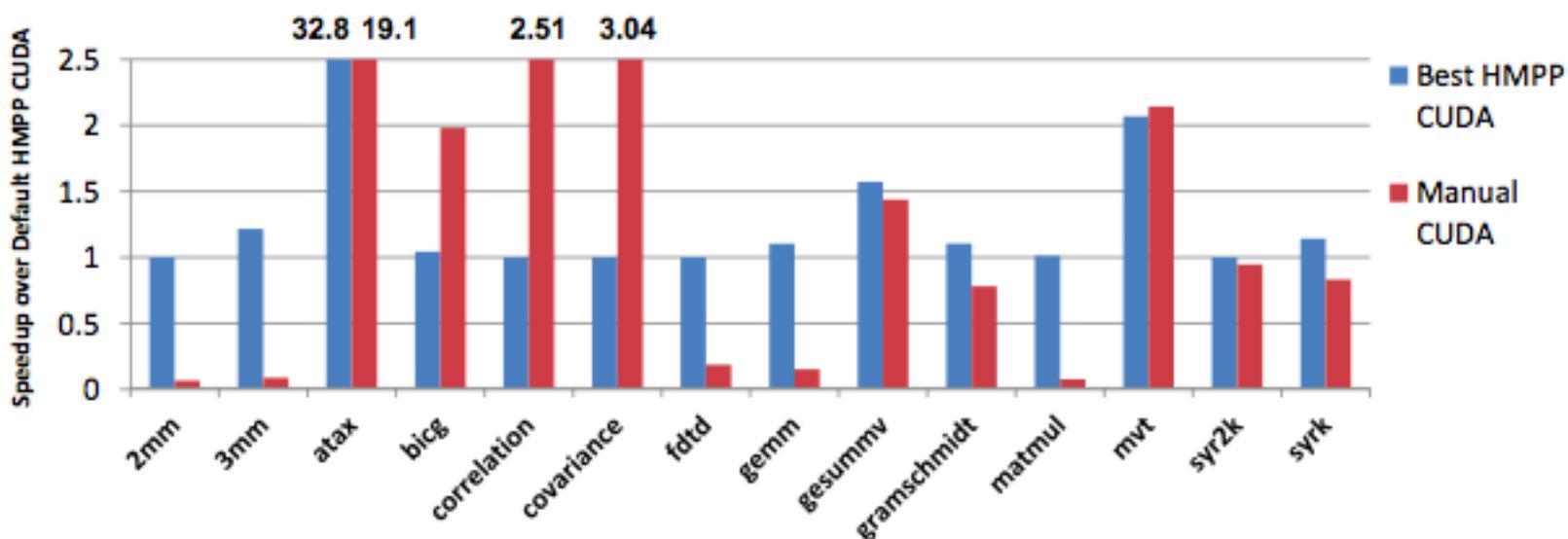


Figure 2: Speedup of best HMPP-transformed and manually-written PolyBench CUDA codes over default HMPP CUDA configuration

*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*

- **OpenCL Results:**

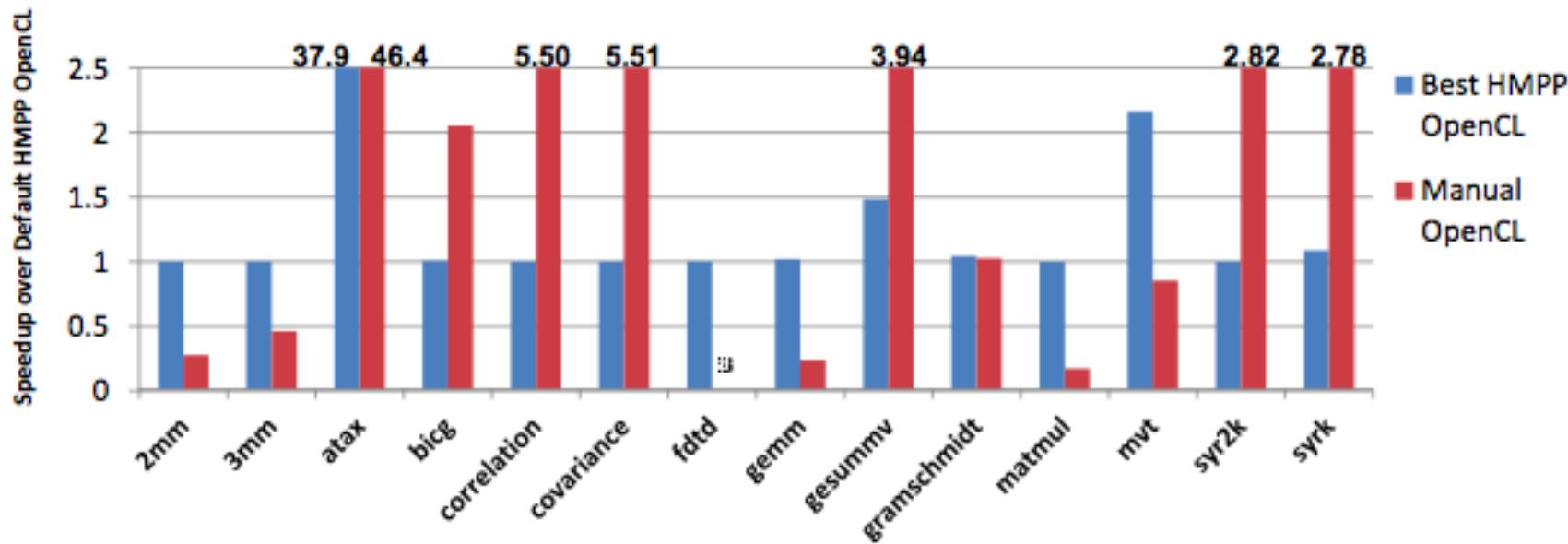
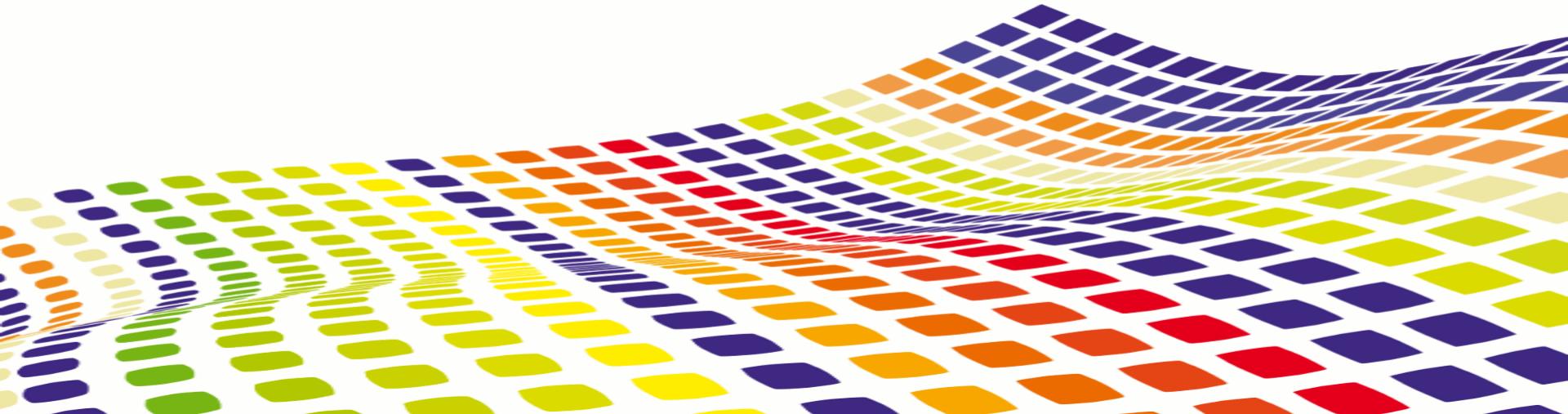


Figure 3: Speedup of best HMPP-transformed and manually-written PolyBench OpenCL codes over default HMPP OpenCL configuration

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

Codelet External Functions



External Functions Directives

sum.h

```
#ifndef SUM_H
#define SUM_H

float sum( float x, float y );

#endif /* SUM_H */
```

sum.c

```
#include "sum.h"

#pragma hmpp
function,target=CUDA
float sum( float x, float y ) {
    return x+y;
}
```

Declare 'sum' as a function
to be called in a codelet

Import of external definition

extern.c

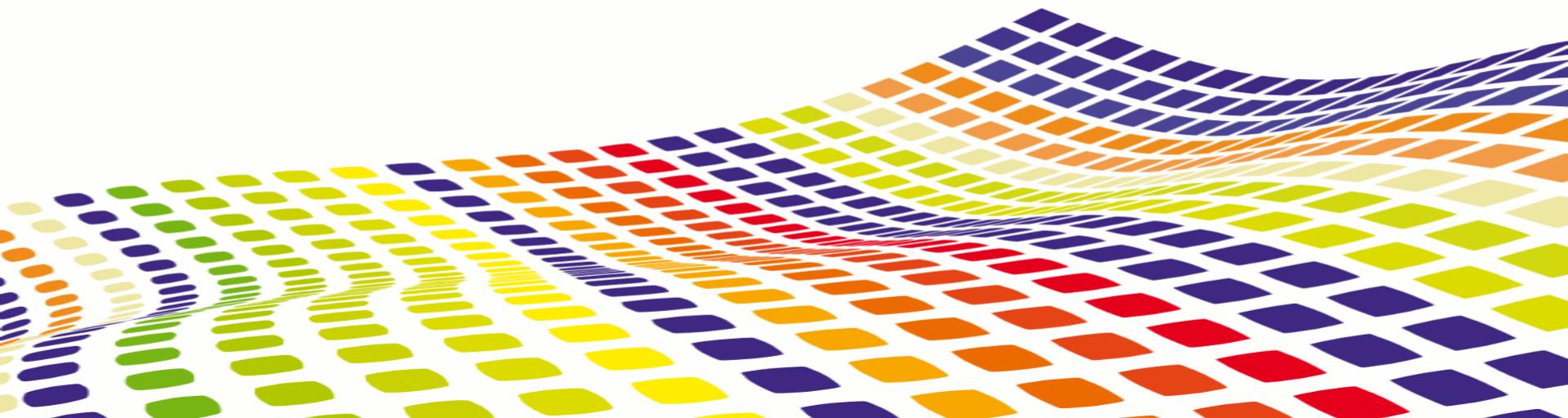
```
#include "sum.h"

int main(int argc, char **argv) {
    int i, N = 64;
    float A[N], B[N];
    ...
    #pragma hmpp cdlt region, args[B].io=inout,
target=CUDA
    {
        for( int i = 0 ; i < N ; i++ )
            B[i] = sum( A[i], B[i] );
    }
    ...
}
```

'sum' is an external function

\$ hmpp --function=hmppcg_functions.dpil gcc source_files

Integrating CUDA Code in Codelets



Native Cuda/OpenCL Functions Support



- Support of handwritten CUDA and OpenCL codes
 - Enables manual optimizations for advanced users

kernel.c

```
float f1 ( float x, float y,
           float alpha, float pow)
{
    float res;
    res = alpha * cosf(x);
    res += powf(y, pow);
    return sqrtf(res);
}
```

```
#pragma hmp mycdlt codelet, ...
void myfunc( int size, float alpha,
             float pow, float *v1,
             float *v2, float *res)
{
    int i;
    float t[size],temp;
    #pragma hmpccg include (<native.cu>)
    #pragma hmpccg native (f1)
    for(i = 0 ; i < size ; i++ )
        res[i] = f1( v1[i],v2[i],
                     alpha, pow);
}
```

native.cu

```
__device__ float f1 ( float x, float y,
                      float alpha, float pow)
{
    float res;
    res = alpha * __cosf(x);
    res += __powf(y, pow);
    return __fsqrt_rn(res);
}
```

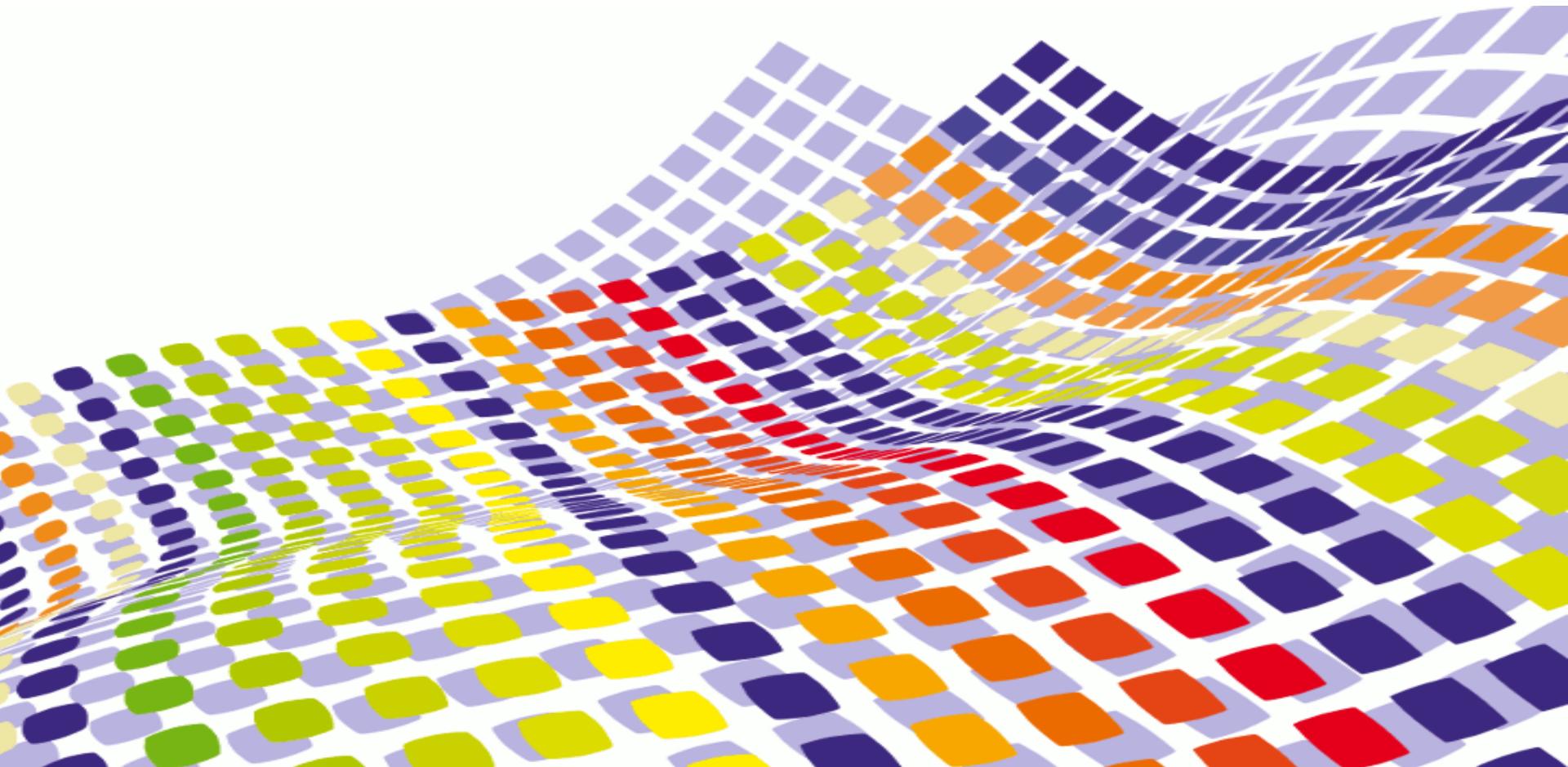
Native function in CUDA

Include file containing native function definition

'f1' is a native function

Use different GPU sqrt

Multi-Accelerator Programming with OpenHMPP



Multi-Accelerator with OpenHMPP – Device Clause



- Most HMPP directives are extended with a device clause

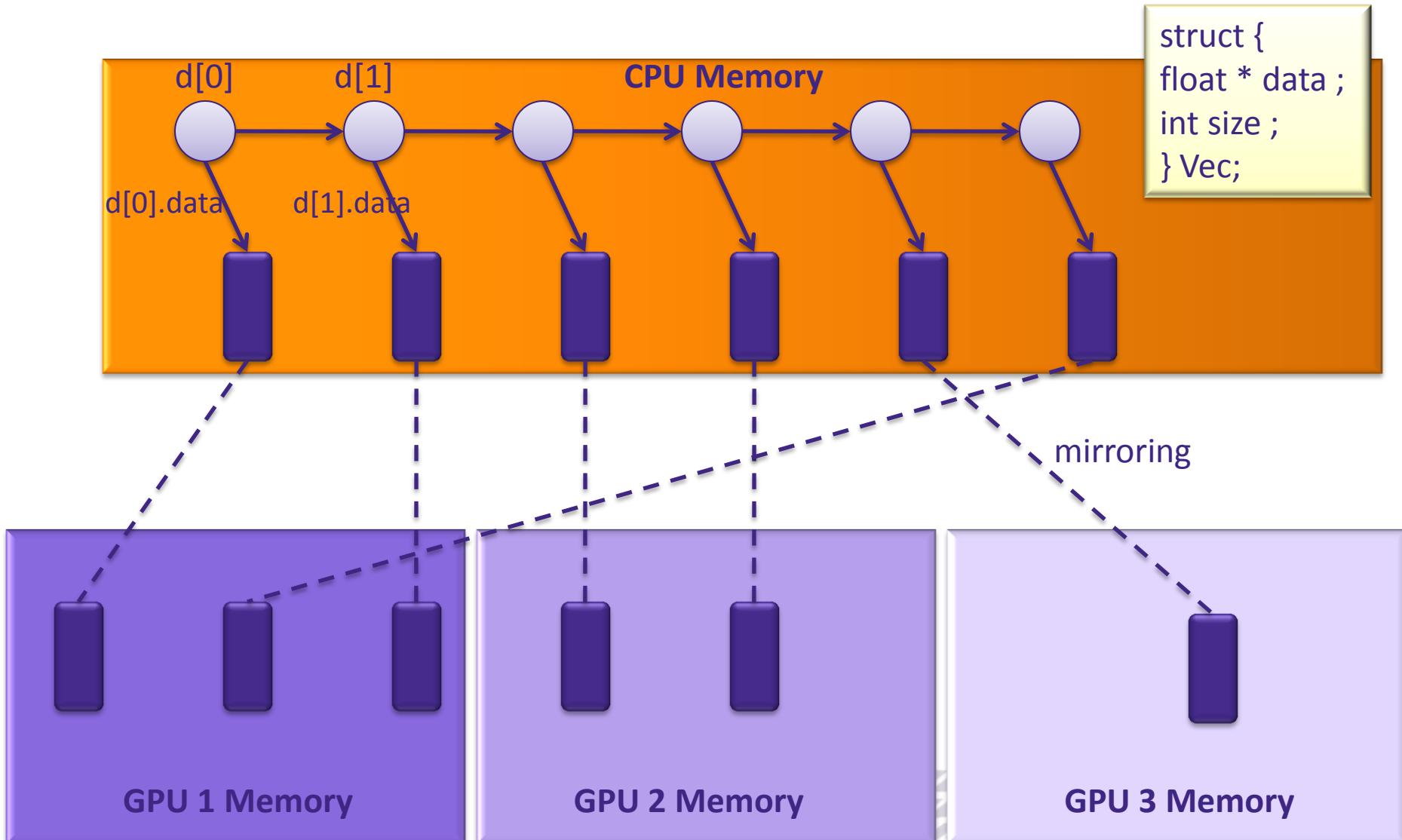
```
#pragma hmpp <MyLabel> MyCodelet command, ..., device="expr"
```

- 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"

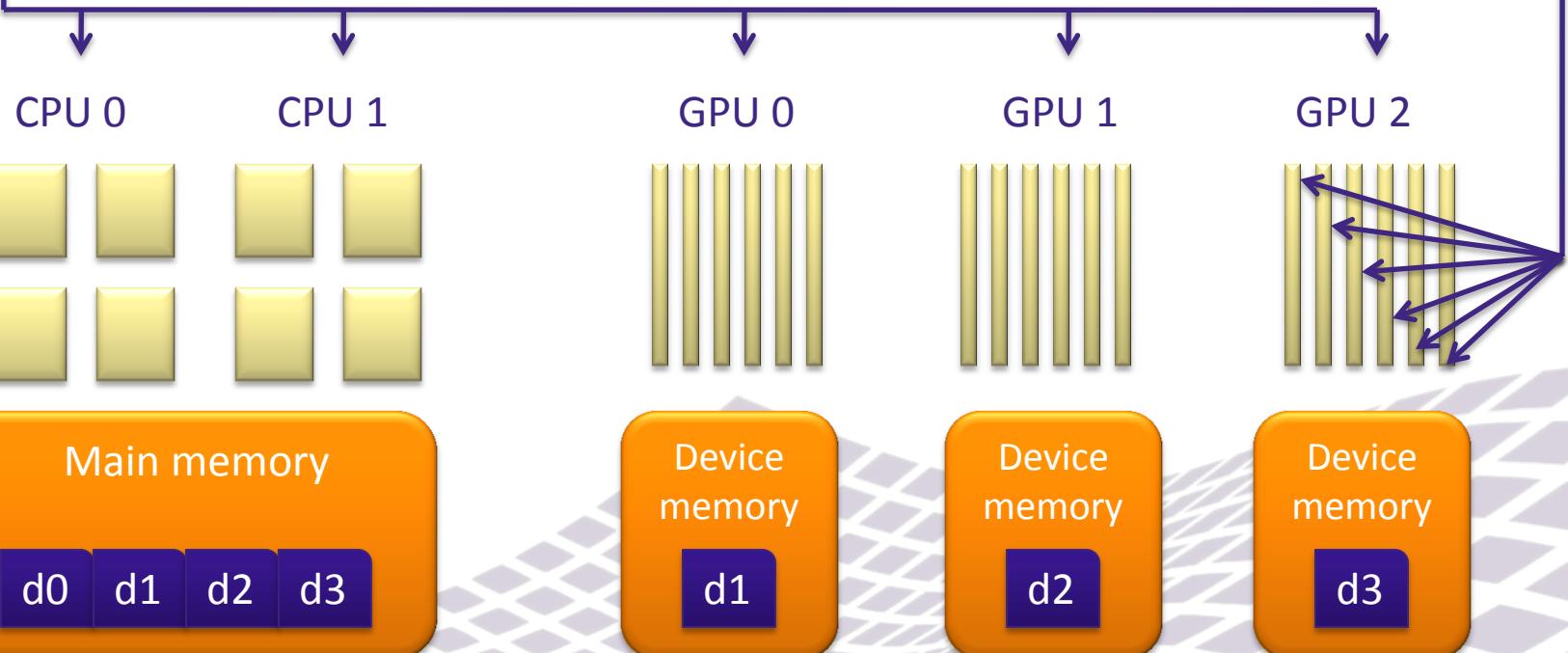
Collections of Data in OpenHMPP 3.x



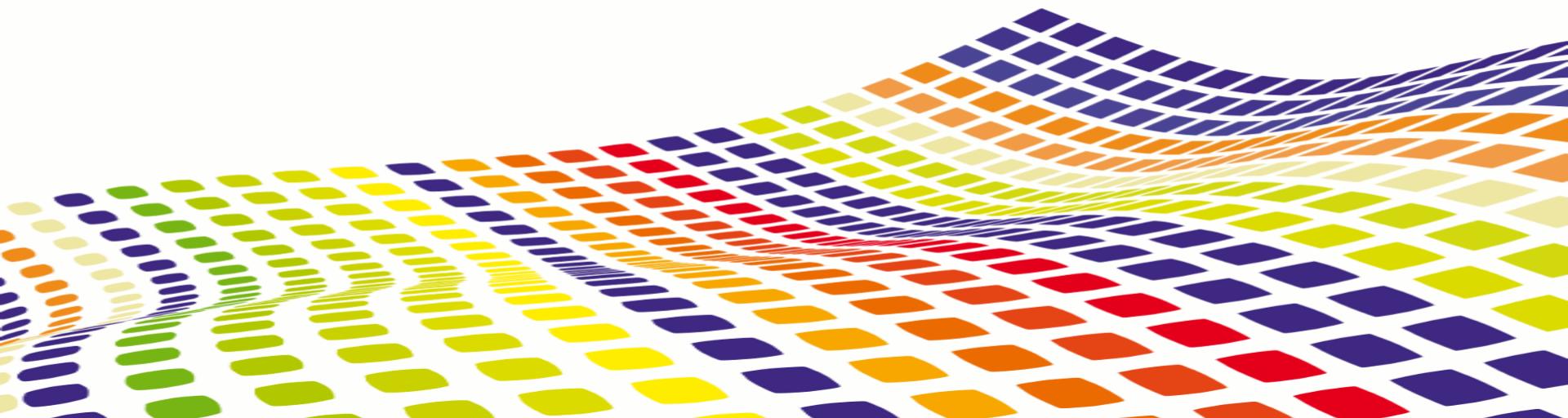
OpenHMPP 3.0 Map Operation on Data Collections

D

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



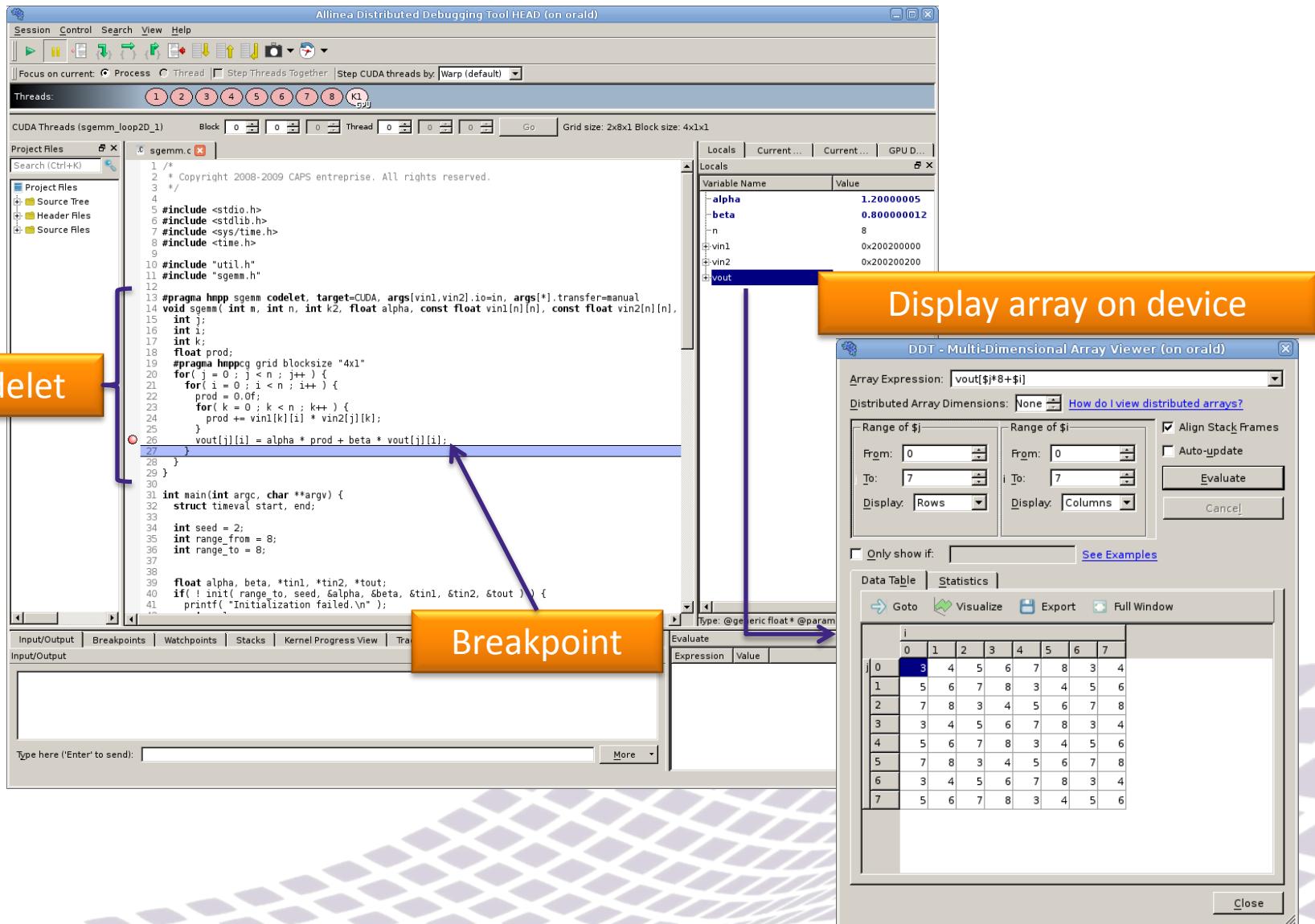
Debugging with Allinea DDT



Debugging with Allinea DDT (1)

- Debugging tool for parallel computing developed by Allinea
- Supports
 - MPI
 - OpenMP
 - CUDA
 - OpenACC, OpenHMPP CUDA applications
- Features
 - Host debugging
 - Device debugging
 - CAPS error handler

Debugging with Allinea DDT (2)



The screenshot illustrates the Allinea Distributed Debugging Tool (DDT) interface for debugging CUDA code. A blue arrow points from the 'Breakpoint' button in the bottom-left window to the line of code where the breakpoint is set. An orange box labeled 'Codelet' highlights the code editor window, which contains the following CUDA code:

```
1 /*  
2 * Copyright 2008-2009 CAPS entreprise. All rights reserved.  
3 */  
4  
5 #include <stdio.h>  
6 #include <stdlib.h>  
7 #include <sys/time.h>  
8 #include <time.h>  
9  
10 #include "util.h"  
11 #include "sgemm.h"  
12  
13 #pragma hpp sgemm codelet, target=CUDA, args[vin1,vin2].io=in, args[*].transfer=manual  
14 void sgemm( int m, int n, int k, float alpha, const float vin1[n][n], const float vin2[n][n],  
15     int j;  
16     int i;  
17     int k;  
18     float prod;  
19     #pragma hppcg grid blocksize "4x1"  
20     for( j = 0 ; j < n ; j++ ) {  
21         for( i = 0 ; i < n ; i++ ) {  
22             prod = 0.0f;  
23             for( k = 0 ; k < n ; k++ ) {  
24                 prod += vin1[k][i] * vin2[j][k];  
25             }  
26             vout[j][i] = alpha * prod + beta * vout[j][i];  
27         }  
28     }  
29 }  
30  
31 int main(int argc, char **argv) {  
32     struct timeval start, end;  
33  
34     int seed = 2;  
35     int range_from = 8;  
36     int range_to = 8;  
37  
38     float alpha, beta, *tin1, *tin2, *tout;  
39     if( ! init( range_to, seed, &alpha, &beta, &tin1, &tin2, &tout ) ) {  
40         printf( "Initialization failed.\n" );  
41     }
```

An orange box labeled 'Breakpoint' highlights the line of code where the breakpoint is set (line 26). Another orange box labeled 'Display array on device' highlights the 'DDT - Multi-Dimensional Array Viewer' window, which shows the value of the 'vout' array:

i	0	1	2	3	4	5	6	7
j	0	3	4	5	6	7	8	3
1	5	6	7	8	3	4	5	6
2	7	8	3	4	5	6	7	8
3	3	4	5	6	7	8	3	4
4	5	6	7	8	3	4	5	6
5	7	8	3	4	5	6	7	8
6	3	4	5	6	7	8	3	4
7	5	6	7	8	3	4	5	6

Conclusion

- Directive-based approaches are currently one of the most promising track for heterogeneous many-cores
 - Preserve code assets
 - At node level help separating parallelism description from implementation
- Need to integrate libraries and user codes
 - Requires interoperability between runtimes
- Auto-tuning is one of the future keys to efficient portability
 - Has to be part of the programming API

GTC Special Offering

- CAPS OpenACC Compiler for \$199
 - OpenACC 1.0 directives
 - C and Fortran to CUDA - for Linux
 - One-year maintenance and support
 - 1 perpetual license
 - Ends June 15th
- See us on Booth #28 or on Exxact booth #36

Accelerator Programming Model

Parallelization

CAPS

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>