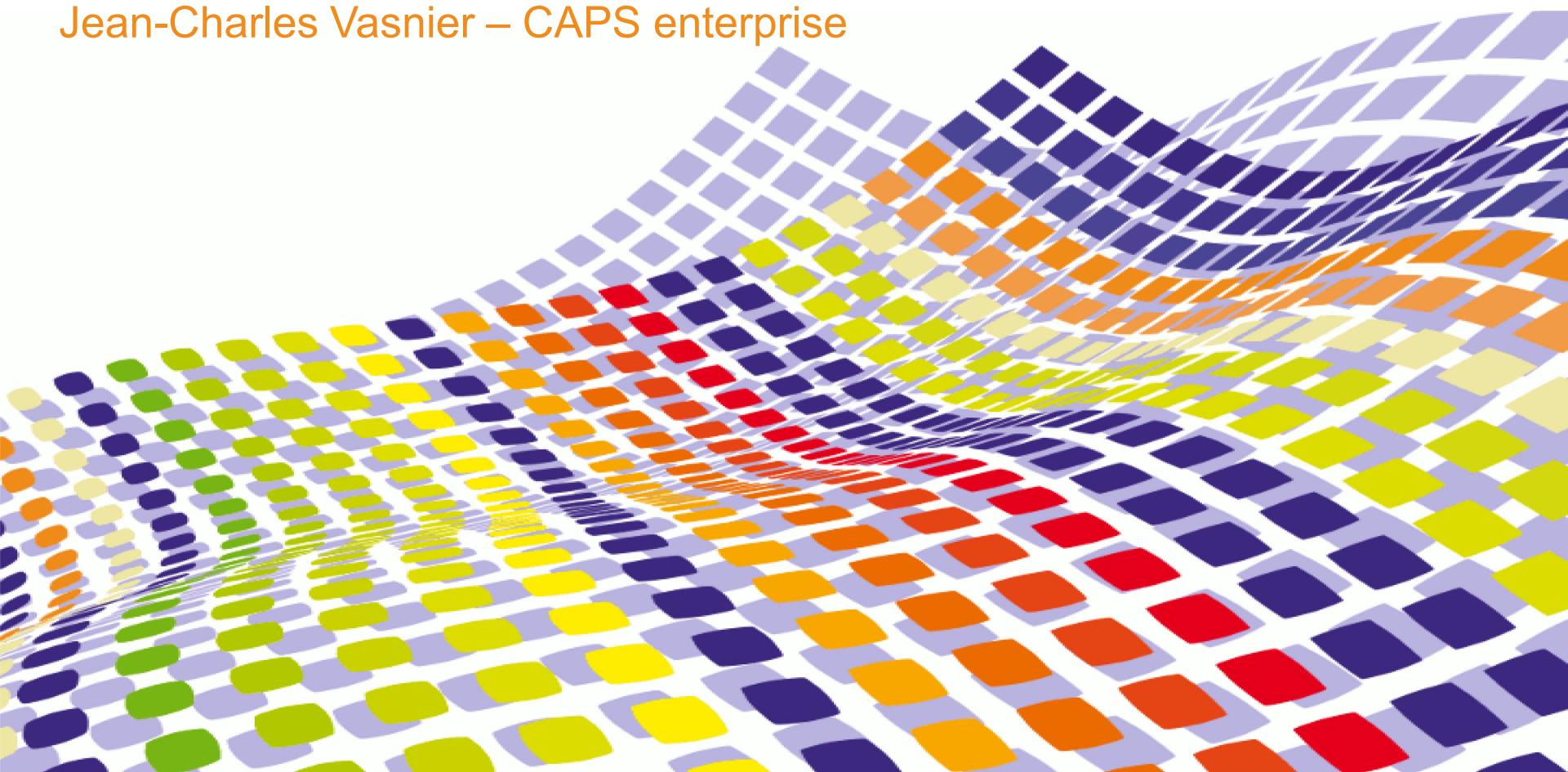


OpenACC Acceleration for Real Science

HMPP Compiler

OLCF Spring Training and Users' Meeting 2012

Jean-Charles Vasnier – CAPS enterprise



A Bit of History



- HMPP 1.0: pioneer in directive-based programming
 - Preserve legacy code using directives, offload computations onto remote devices
 - Abstract CUDA programming with CUDA generator directives
- HMPP 2.0: enrich set of directives to fully exploit GPU capabilities
 - Optimize data movement: data sharing, resident data, partial transfers
 - Enable developers to address CUDA features such as constant and shared memory, grid, etc.
 - New OpenCL generator

What's New in HMPP 3.0?



- Dynamic data management mechanism
 - Mirrors identified by their host address
 - Simplifies management of data with less directives
- Multi-device programming
 - Exploit multiple devices in one compute node
 - Distribute collections of data over multiple devices
- New run-time API
 - Three bindings for C, C++ and Fortran 90-2003
 - Low level OpenCL style programming with OpenCL/CUDA kernel generation
- Open library integration system
 - CPU and GPU libraries coexist in same binary (proxy mechanism)
 - Data sharing between HMPP codelets and libraries
 - User can write their own HMPP proxies
 - Proxies provided for cuBLAS, CULA, cuFFT, keeping CPU API.

Accelerators for Scientific Computing



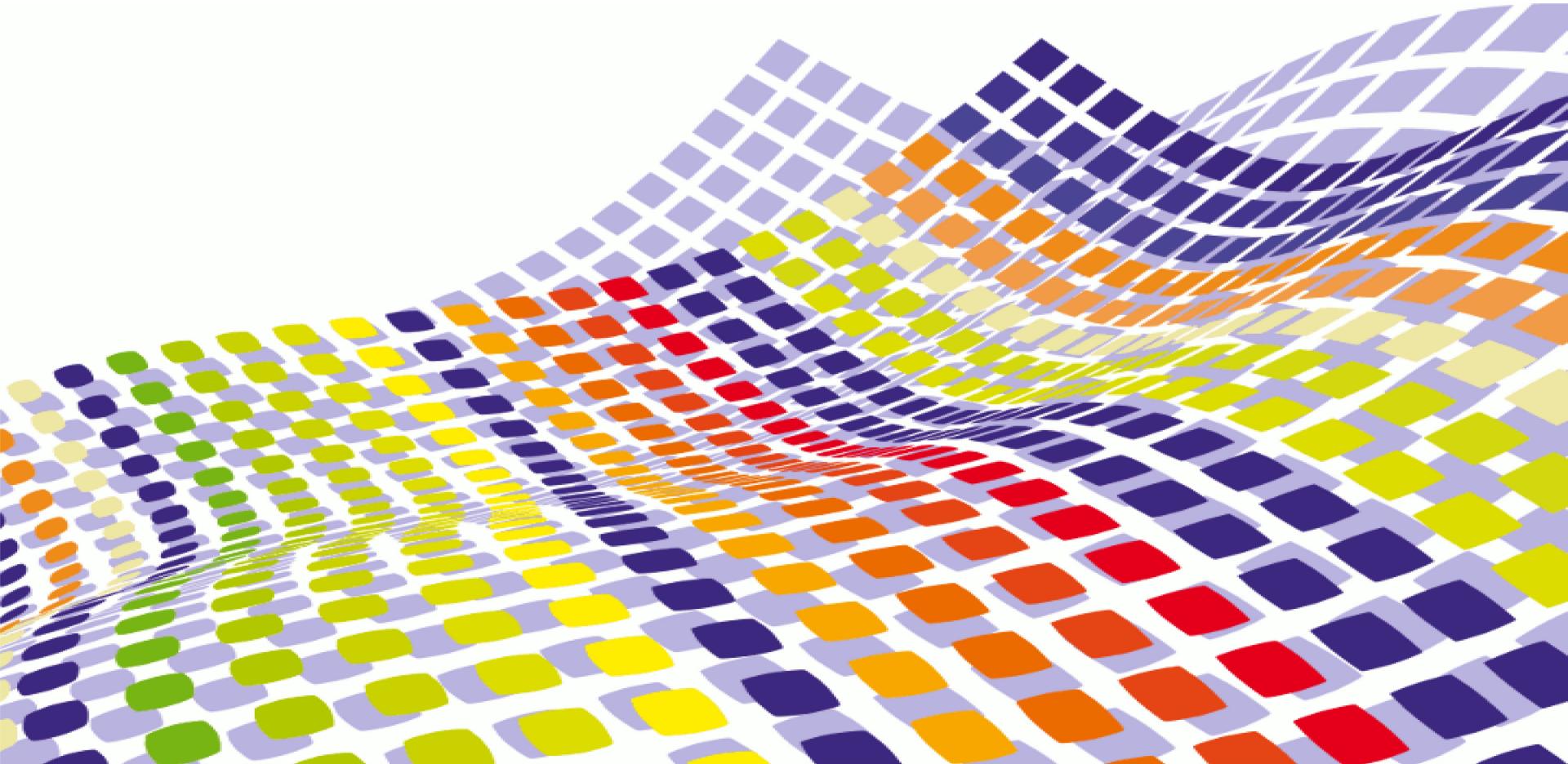
- CUDA with Nvidia GPUs has been a breakthrough providing unprecedented computing power for many computing fields
 - Oil and Gas exploration
 - Image processing
 - Bio informatics
 - ...
- Integration of CUDA code in scientific applications is not always possible
 - Application programmers unfamiliar with C/C++ language
 - 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 architectures
- Directive-based approaches have been design to supplement CUDA and OpenCL programming models for helping the use of accelerators

OpenACC for Leveraging CUDA Technology



- Spread the existing CUDA success to a wider community
 - About 150,000 CUDA developers
 - Huge existing scientific code bases cannot be rewritten, only changes can be made
- Fundamentally OpenACC is abstracting accelerator programming
 - Fast learning curve
 - Portable across hardware
- Incremental approach for migrating legacy codes to accelerator technology
 - Fast prototyping: quickly produce code that runs in the accelerator
 - Increases productivity: few code changes, one single source code

Directive-based Programming



Directive-based Approaches

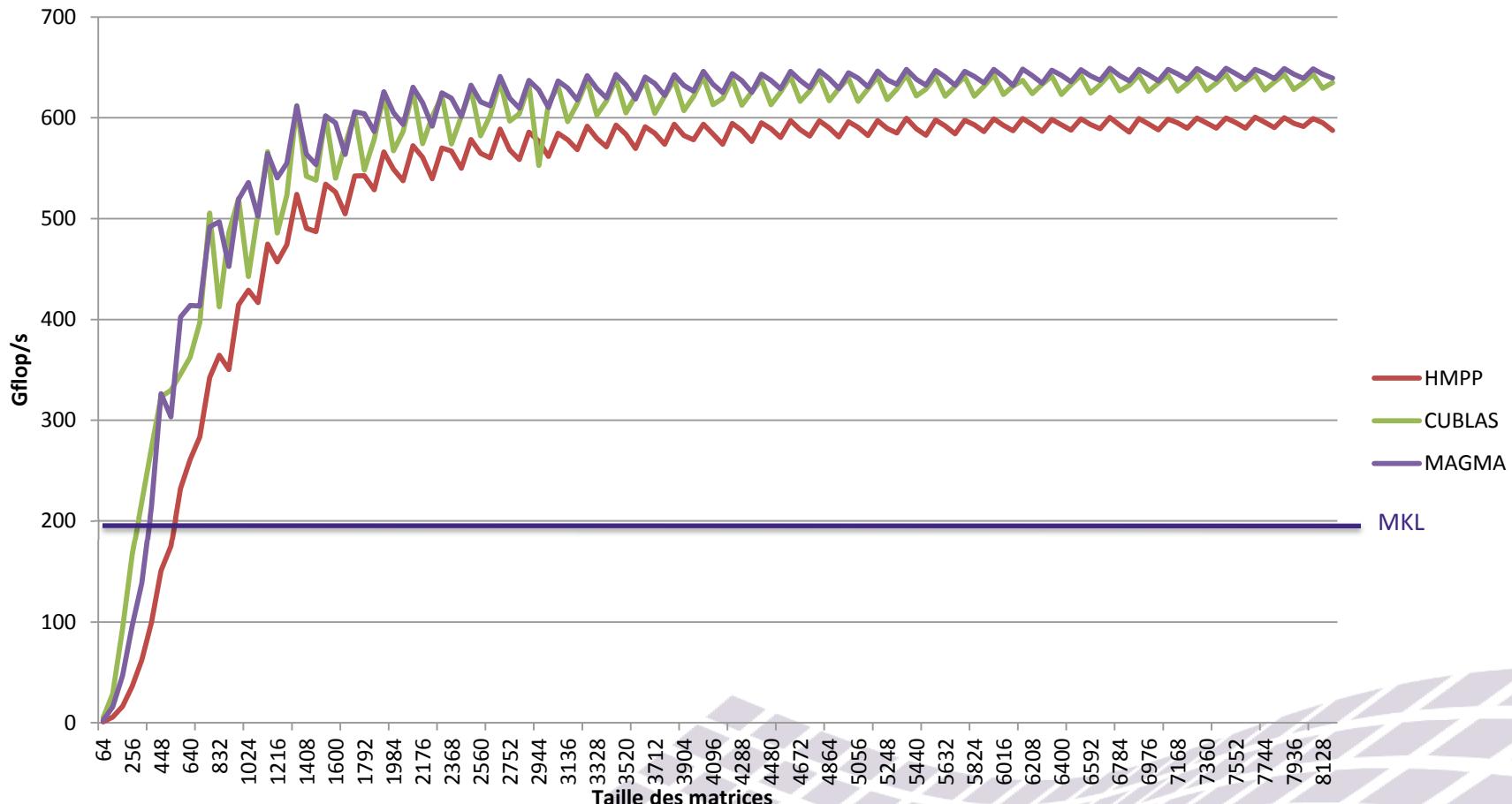


- Supplement an existing serial language with directives to express parallelism and data management
 - Preserves code basis (e.g. C, Fortran) and serial semantic
 - Competitive with code written in the device dialect (e.g. CUDA)
 - Incremental approach to many-core programming
 - Mainly targets legacy codes
- Many variants
 - HMPP
 - PGI Accelerator
 - OpenACC
 - OpenMP Accelerator extension
 - ...
- OpenACC is a new initiative by CAPS, CRAY, PGI and NVidia
 - A first common subset

HMPP BLAS Performance on NVIDIA Fermi



SGEMM Performance



Performances (max)

NO TRANSFERTS

énergie atomique + énergies alternatives

T in s

	Scalar	18457.52	3030.9	4668.67	7531.1	1054.4	15.68	1933.82
	OMP=8	5040.1	379.09	1479.91	1302.64	1248.35	6.76	348.32
	HMPP	820.34	56.88	388.23	156.61	81.58	2.04	29.7
	CUDA	1267.66	75.01	458.95	135.61	66.99	66.51	531.3
	ALL	NOISE	DIFFUS	KERSBS	SHIFT	BOUND	FLUX	
	OMP / SEQ	3.66	8.00	3.15	5.78	0.84	2.32	5.55
	HMPP / SEQ	22.50	53.29	12.03	48.09	12.92	7.69	65.11
	CUDA / SEQ	14.56	40.41	10.17	55.53	15.74	0.24	3.64

Speedup

Geom: 128 x 128 x 256

No I/O

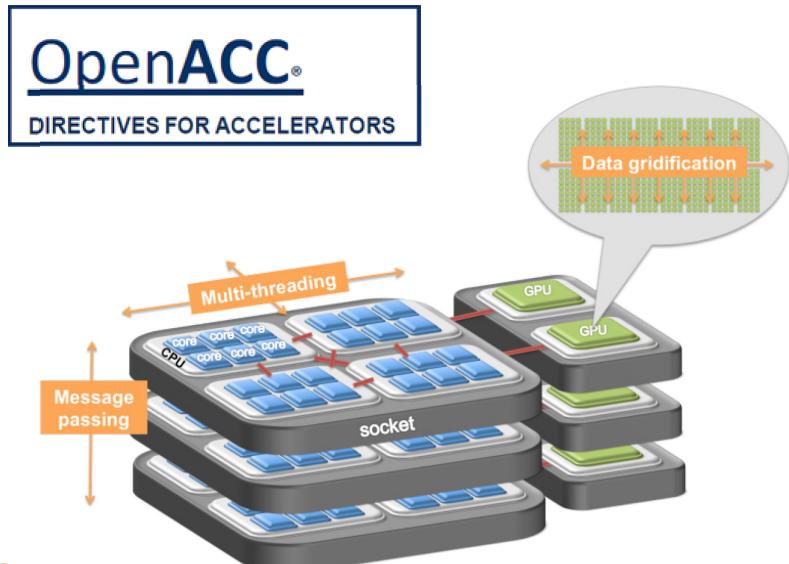
1 MPI

Diffus = FFT FW + diffrac + FFTBW

KERSBS = KER + SBS

OpenACC Initiative

- Express data and computations to be executed on an accelerator
 - Using marked code regions
- Main OpenACC constructs
 - Parallel and kernels regions
 - Parallel loops
 - Data regions
 - Runtime API
- Subset of HMPP supported features
 - OpenACC constructs interoperable with other HMPP directives
 - OpenACC support to be released in HMPP in April 2012 (beta available)
- Visit <http://www.openacc-standard.com> for more information



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

Parallel Regions

- Start parallel activity on the accelerator device
 - Gangs of workers are created to execute the accelerator parallel region
 - Exploit parallel loops
 - 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] + ...;
            }
        }
    }
}
```

Kernels 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];
  }
}
```

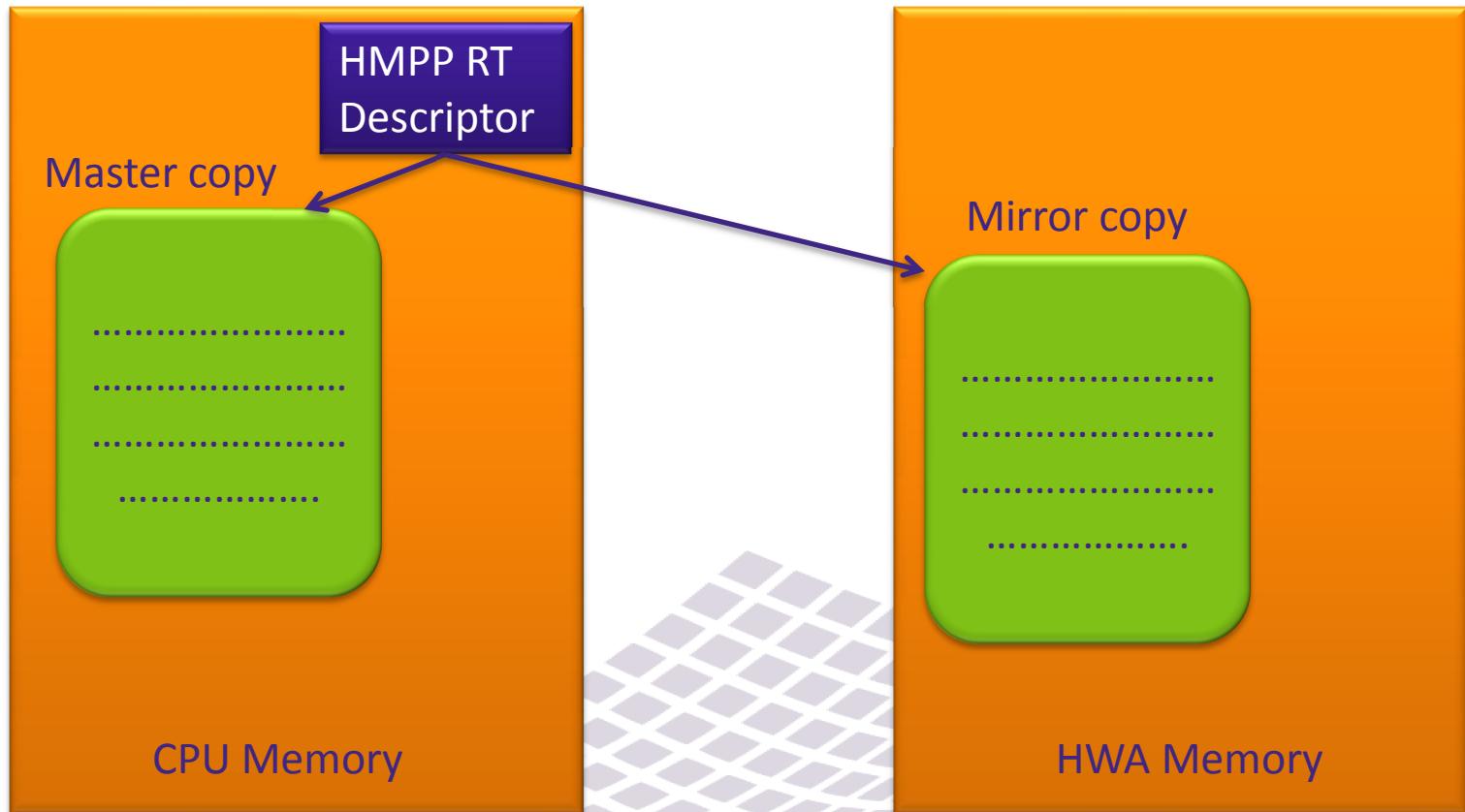
Data Management Directives

- 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] = ...;
    }
}
```

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
 - User ensures consistency of copies via directives

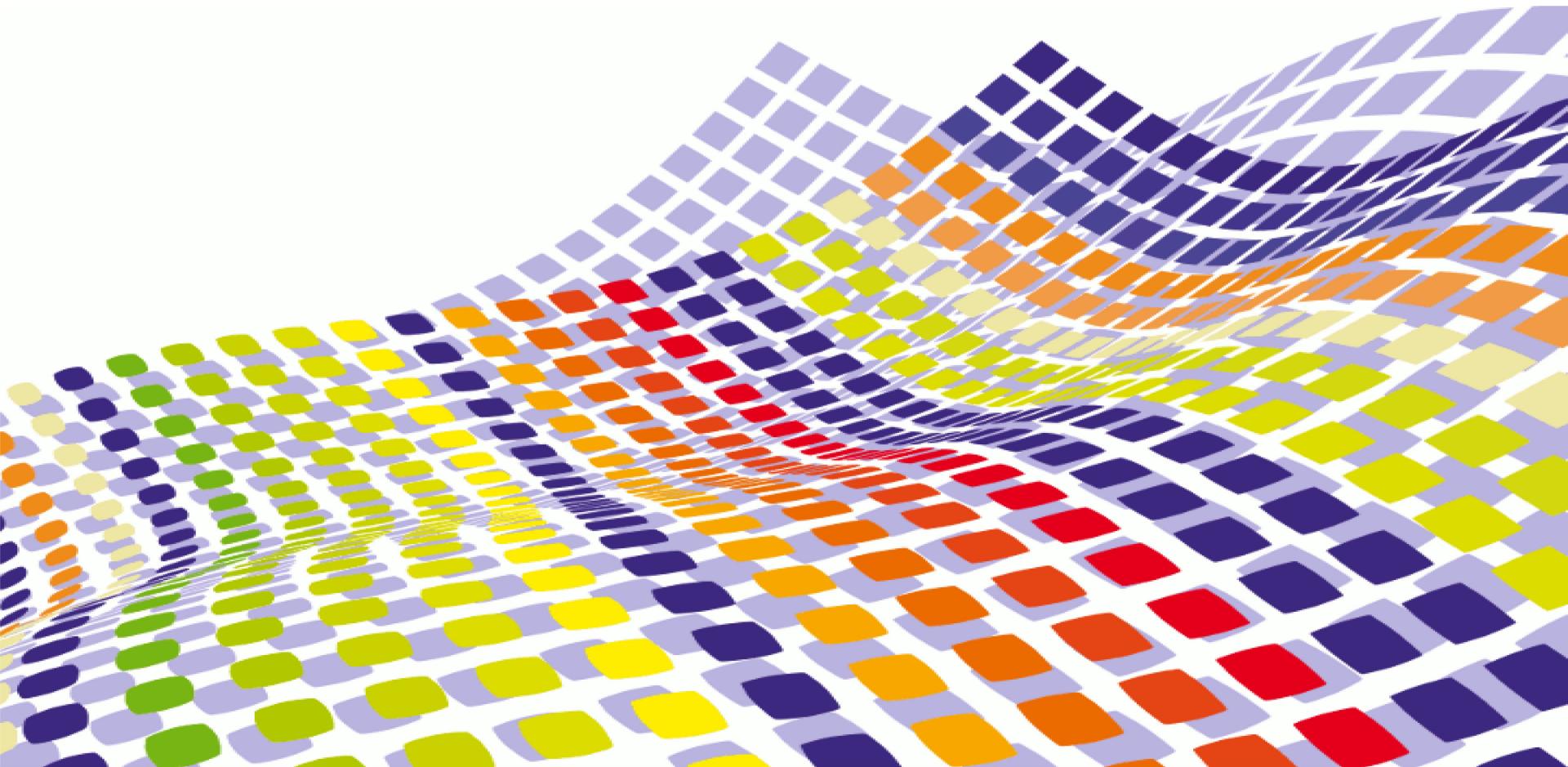


Runtime API

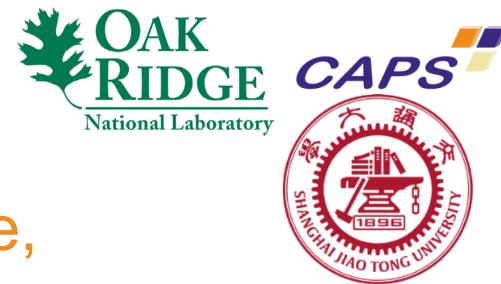
- Set of functions for managing device allocation (C version)

- `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*)`
- ...

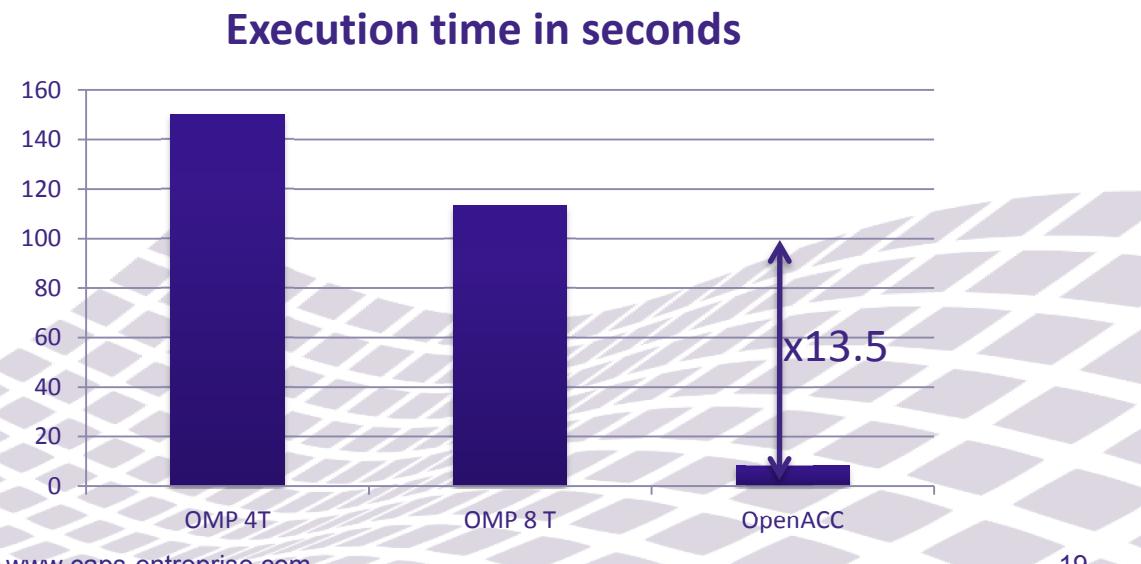
A Few Examples



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/)

OpenACC Applications with HMPP



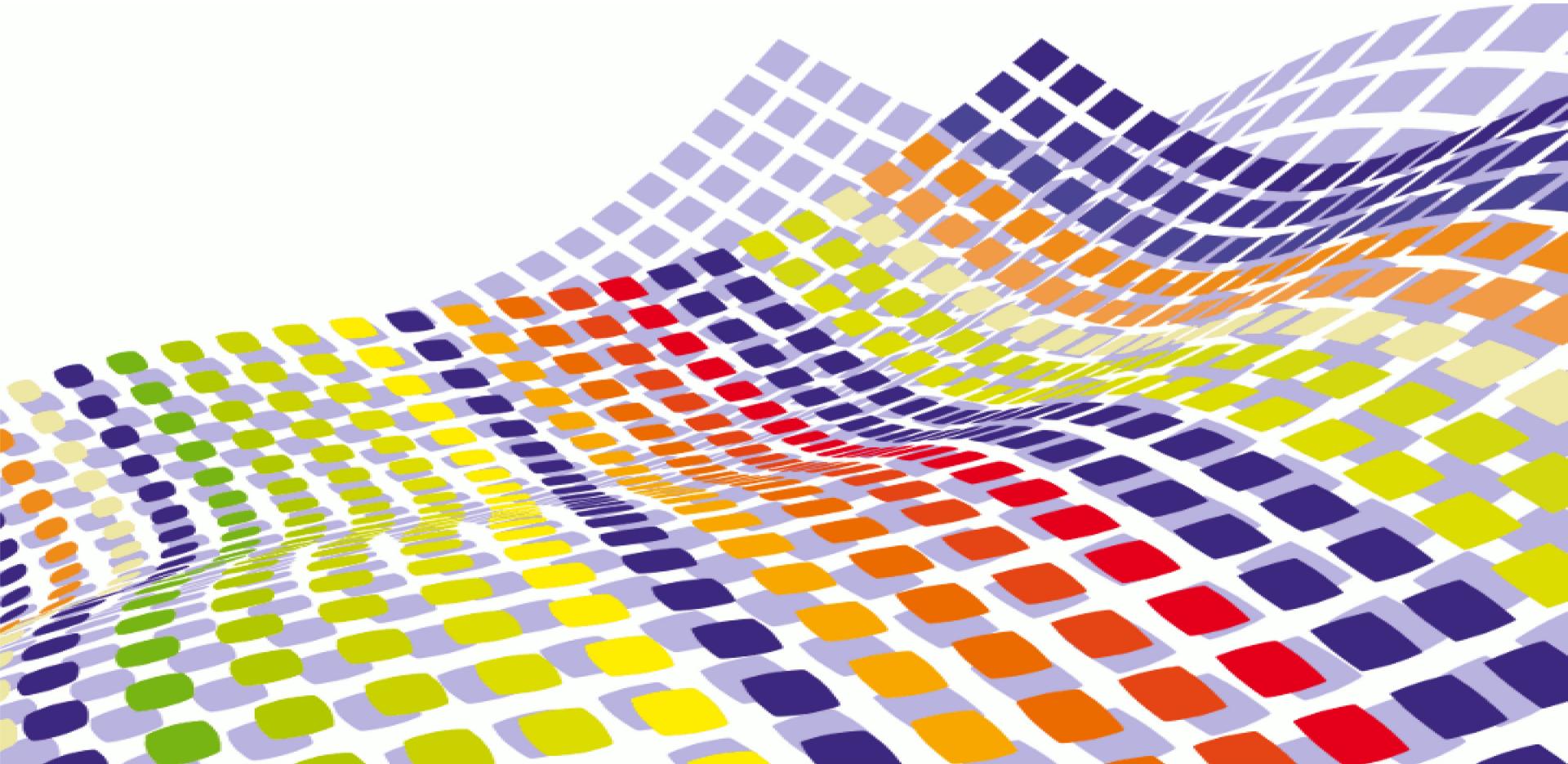
- BlackSholes: partial differential equations used in finance
 - 250 LoC, C language
 - 7 OpenACC directives used
- SobelFilter: discrete differentiation operator in image processing
 - 189 LoC, C language
 - 6 OpenACC directives
- Convolution kernel
 - 500 LoC in C
 - 3 OpenACC directives
- HydroC: simulation of the structure and formation of the galaxy used in the PRACE European project
 - ~4,000 LoC, C language
 - 50 OpenACC directives

Performance Results

- Tesla C2070
- Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz (4 cores)
- CUDA 4.1
- HMPP 3.0.7 (beta OpenACC support)

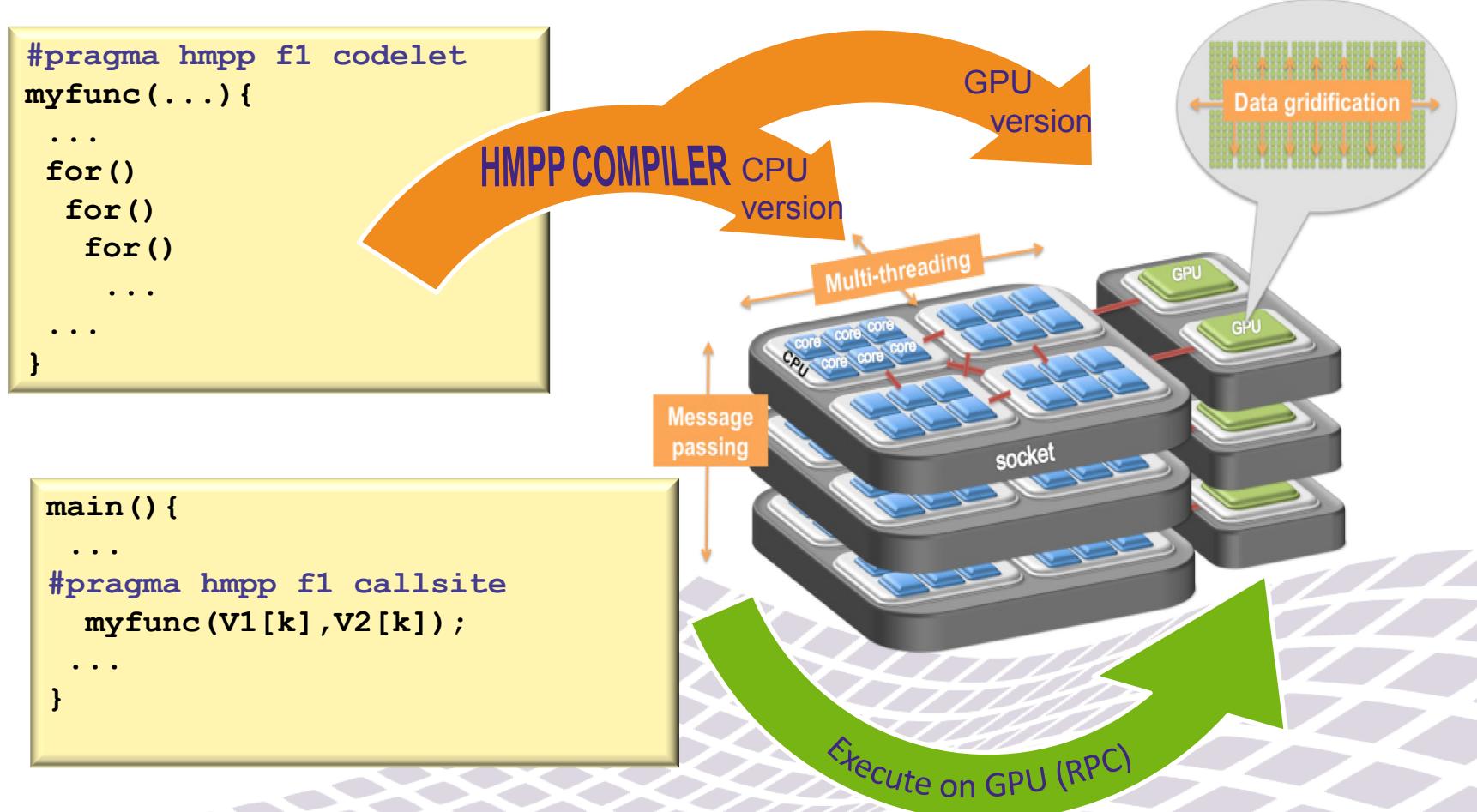
Application	OpenACC execution time (seconds)	CPU execution time (seconds)	Speedup
BlackScholes_C	5,88E-03	1,77E-02	3,0
SobelFilter_C	3,29E-03	7,79E-02	23,7
HydroC	5,53E+00	1,29E+01	2,3
Convolution	2,00E+00	2,68E+00	1,3

HMPP Programming with OpenACC



HMPP Heterogeneous Multicore Parallel Programming

- Codelet and region based directives for many-cores
 - CUDA, OpenCL code generation, soon Intel MIC, x86



Directives Overview

```
#pragma hmpp <label> <directive type> [, <directive parameter>]* [&]  
!$hmpp <label> <directive type> [, <directive parameter>]* [&]
```

- Label identifies a group of directives belonging to a codelet definition and execution

HMPP directives	Meaning
codelet callsite	Codelet declaration Codelet execution
advancedload delegatedstore	Host to Device data transfer Device to Host data transfer
synchronize	Synchronization barrier
acquire release	Device acquisition Release the device
allocate free	Data allocation on the device Free allocated data on the device
parallel	Parallel execution on multiple devices

Accelerate Codelet Function

- Declare and call a GPU-accelerated version of a function

```
#pragma hmpp sgemm codelet, target=CUDA:OPENCL, &
#pragma hmpp & transfer=atcall
extern void sgemm( int m, int n, int k, float alpha,
                    const float vin1[n][n], const float vin2[n][n],
                    float beta, float vout[n][n] );
int main(int argc, char **argv) {
    /* . . . */
    for( j = 0 ; j < 2 ; j++ ) {
        #pragma hmpp sgemm callsite
        sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    }
    /* . . . */
}
```

Declare CUDA and
OPENCL codelets

Synchronous codelet call

Data Mirroring

- A simple mirror example with multiple callsites

```

...
float vin1[size][size], vin2[size][size], vout[size][size];
...

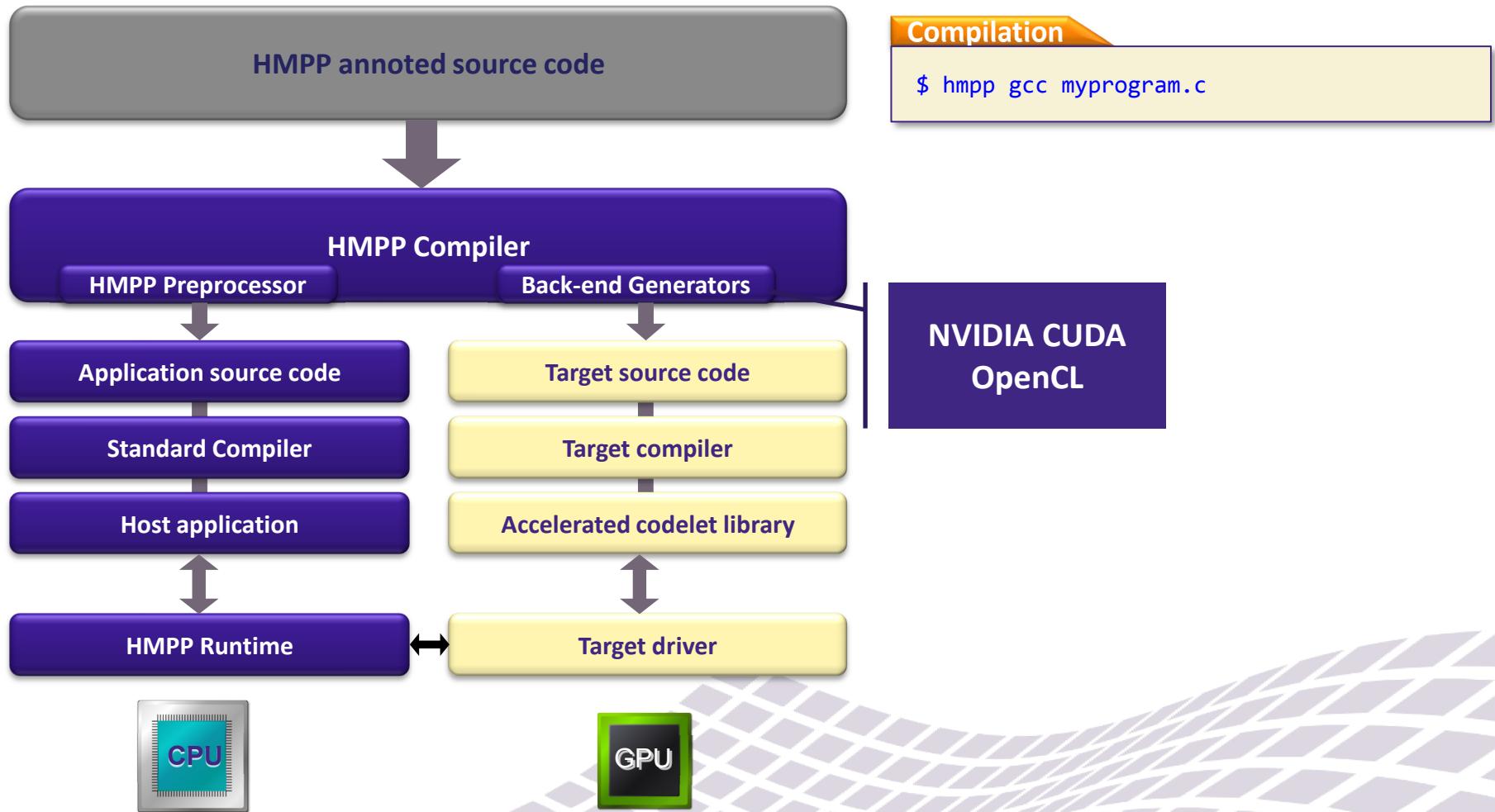
//Allocate the mirrors for vin1, vin2 and vout
#pragma hmpp allocate, data[vin1, ...], size={size,size}

//Transfer data to the GPU from the mirrors
#pragma hmpp advancedload, data[vin1,vin2,vout]

//Main loop
for( i=0;i<Nb_iter;i++) {
    ...
    //Compute the sgemm and process on the GPU
#pragma hmpp sgemm callsite
    sgemm( vin1, vin2, vout );
#pragma hmpp outprocessing callsite
    process( vout );
    ...
}
//Get back the result
#pragma hmpp delegatedstore, data[vout]
...

```

HMPP Compilation



What is in HMPP and not in OpenACC



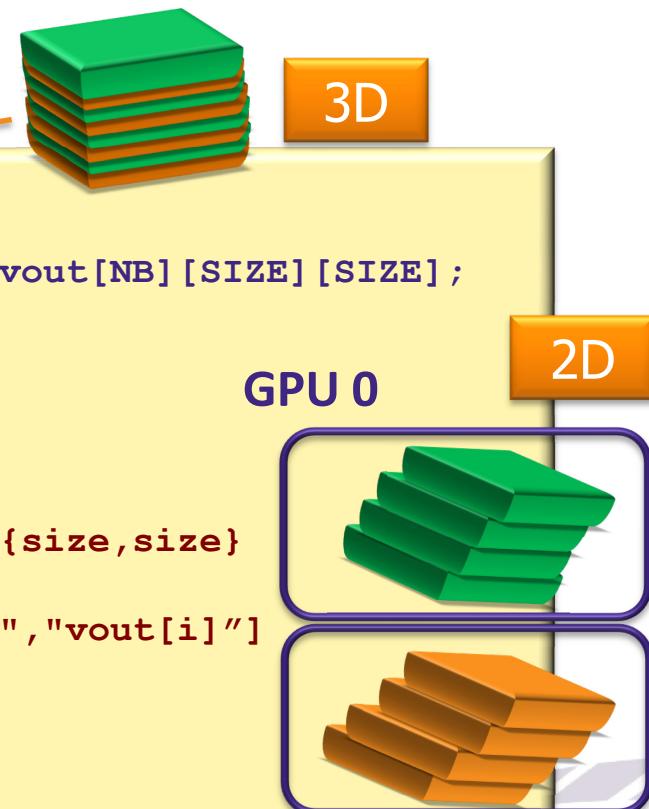
- Multiple devices management
 - Data collection / map operation
- Library integration directives
 - Needed for a “single source many-core code” approach
- Loop transformations directives for kernel tuning
 - Tuning is very target machine dependent
- Open performance APIs
 - Tracing
 - Auto-tuning (H2 2012)
- And many more features
 - Native functions, buffer mode, UVA support, codelets, ...

Multiple Devices and Data Distribution

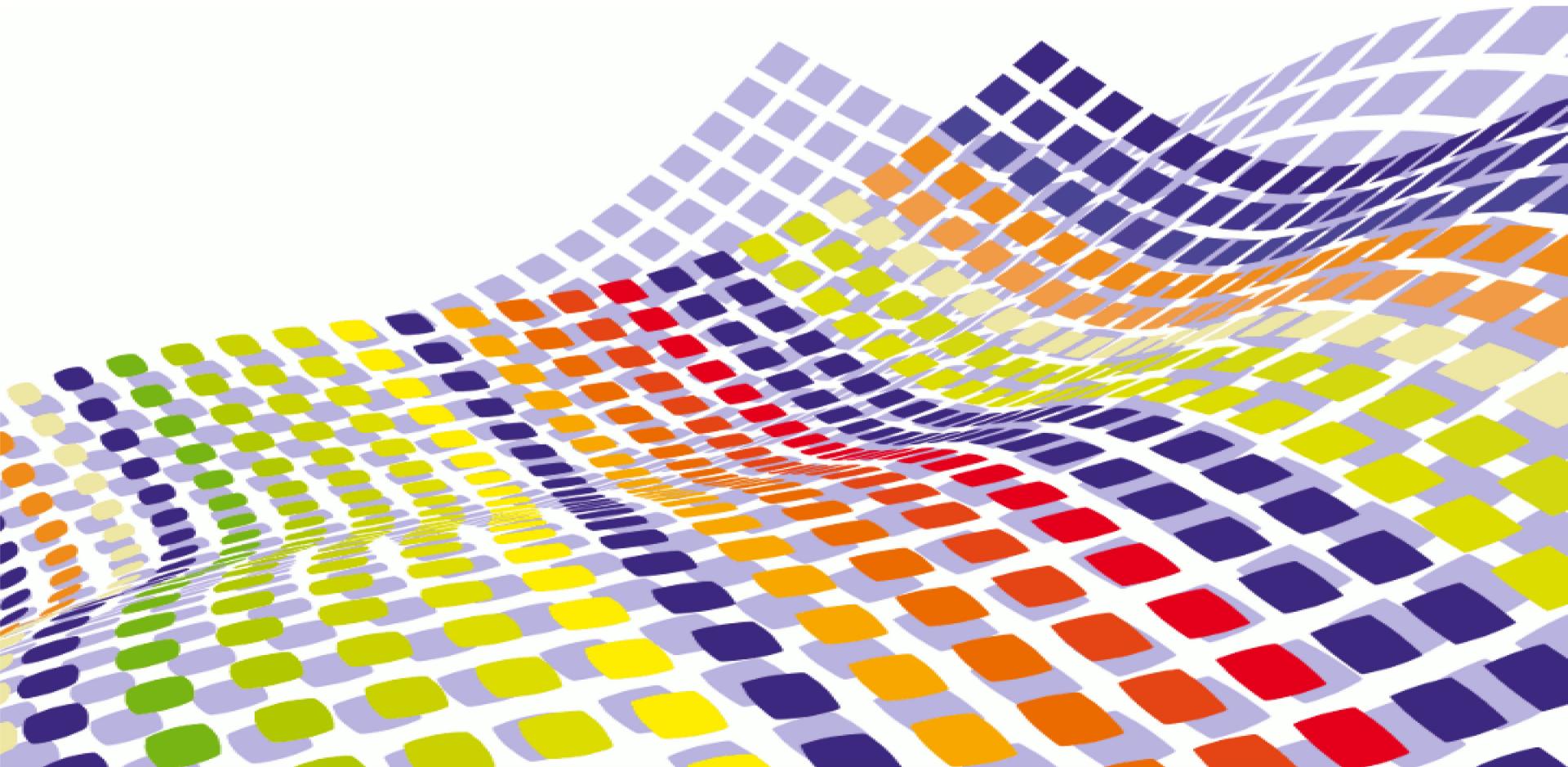
- Define a multi-GPU data distribution scheme and let HMPP execute over multiple devices

```
...
float vin1[NB][SIZE][SIZE], vin2[NB][SIZE][SIZE], vout[NB][SIZE][SIZE];
...

#pragma hmpp parallel, device="i%2"
    for( i=0;i<NB;i++ ) {
        //Allocate the mirrors for vin1, vin2 and vout
#pragma hmpp allocate, data["vin1[i]", ...], size={size,size}
        //Transfer data to the GPU from the mirrors
#pragma hmpp advancedload, data["vin1[i]","vin2[i]","vout[i]"]
        ...
#pragma hmpp sgemm callsite
        sgemm( vin1[i], vin2[i], vout[i] );
        ...
//Get back the result
#pragma hmpp delegatedstore, data["vout[i]"]
    }
...
```



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 storage 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, HMPP, etc.
- Mostly an engineering issue

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 native library function by a call to a proxy that handles GPUs and allows to mix user GPU code with libraries

```
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)
```

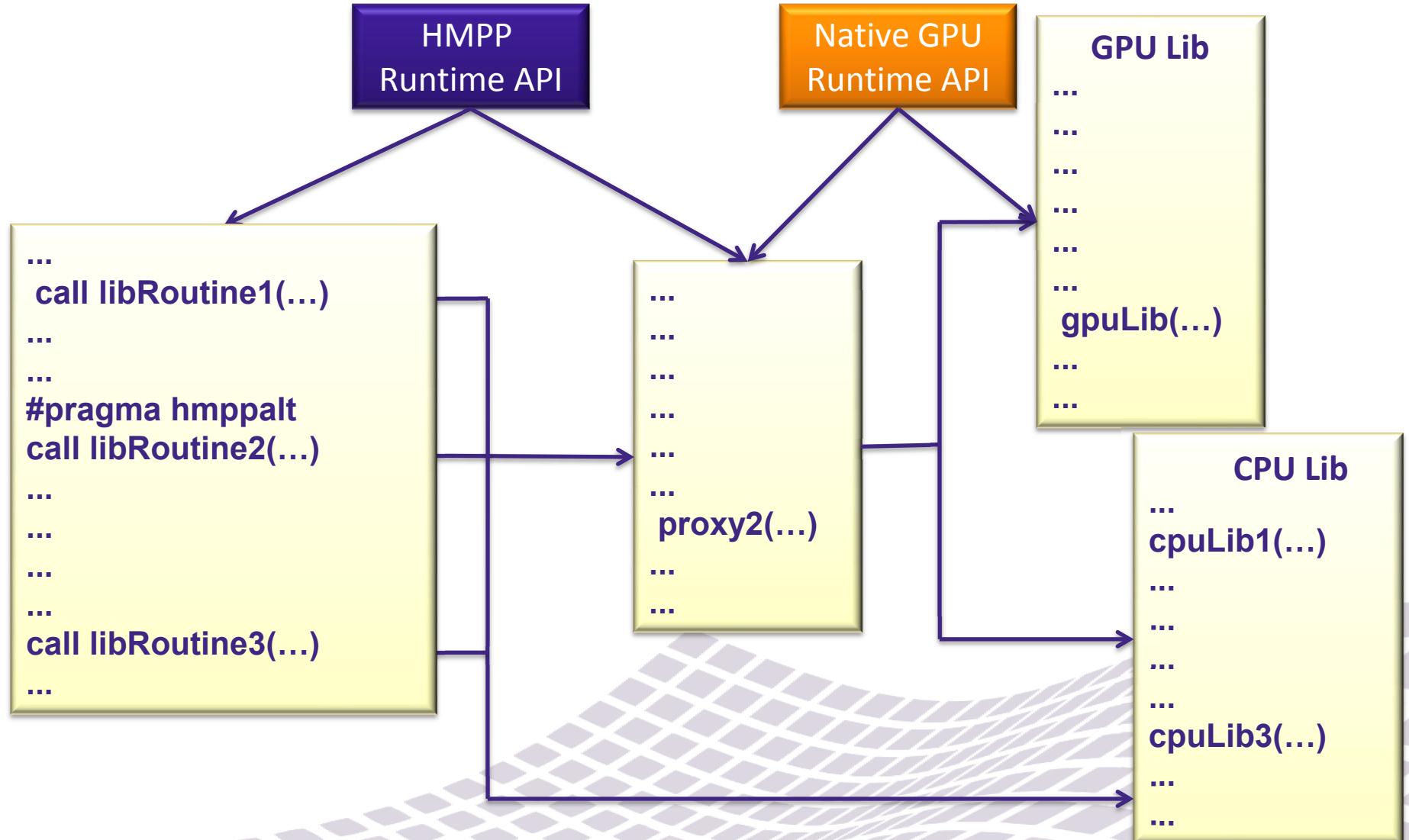
C

C SAME HERE

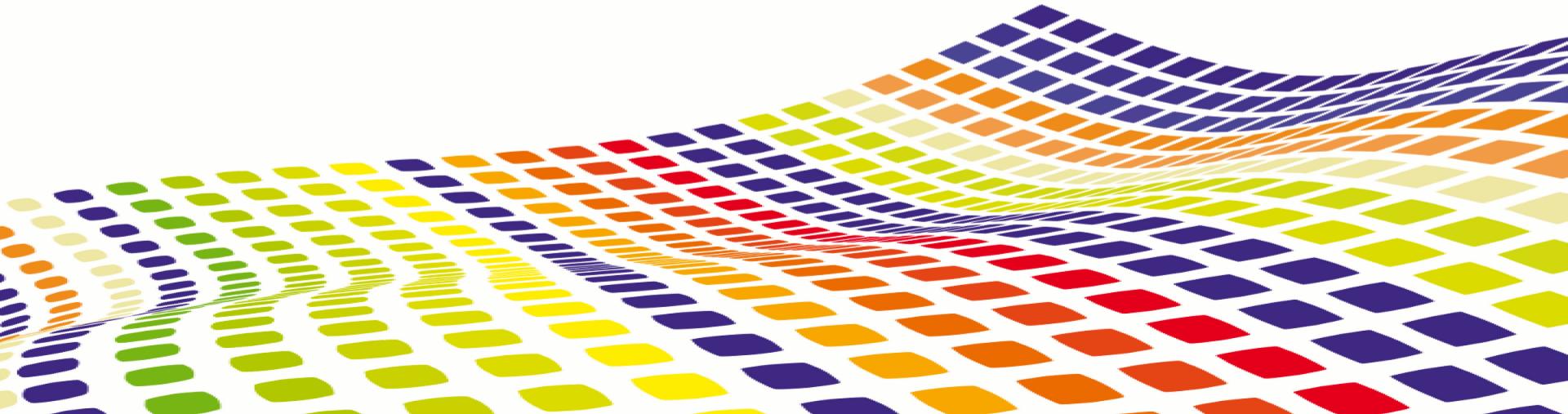
```
!$hmppalt ffte call , name="zfft1d" , error="proxy_err"
CALL ZFFT1D(A,N,1,B)
CALL DUMP(A,N)
```

Library Interoperability in HMPP 3.0

C/CUDA/...



HMPP Tuning



Fine tune kernel performance

- Add code properties
 - Force loop parallelization
 - Indicate parameter aliasing
- Apply code transformation
 - Loop unrolling, blocking, tiling, permute, ...
- Control mapping of computations
 - Gridify
 - Use of GPU constant/shared memory
 - GPU threads synchronization barriers

Tuning Directive Example

```
#pragma hmpp dgemm codelet, target=CUDA, args[C].io=inout
void dgemm( int n, double alpha, const double *A, const double *B,
            double beta, double *C ) {
    int i;

#pragma hmppcg(CUDA) gridify(j,i), blocksize "64x1"
#pragma hmppcg(CUDA) unroll(8), jam, split, noremainder
    for( i = 0 ; i < n; i++ ) {
        int j;
#pragma hmppcg(CUDA) unroll(4), jam(i), noremainder
        for( j = 0 ; j < n; j++ ) {
            int k; double prod = 0.0f;
            for( k = 0 ; k < n; k++ ) {
                prod += VA(k,i) * VB(j,k);
            }
            VC(j,i) = alpha * prod + beta * VC(j,i);
        }
    }
}
```

1D gridification
Using 64 threads

Loop transformations

Using CUDA Shared Memory

```
void conv1(int A[N], int B[N])
{
    int i,k ;
    int buf[DIST+256+DIST] ;
    int grid = 0 ;
#pragma hmppcg set grid = GridSupport()
    if (grid) {
#pragma hmppcg gridify(i), blocksize 256x1, shared(buf)
        for (i=DIST; i<N-DIST ; i++) {
            int t ;
#pragma hmppcg set t = RankInBlock(i)
            // Load the first 256 elements
            buf[t] = A[i-DIST] ;
            // Load the remaining elements
            if (t < 2*DIST )
                buf[t+256] = A[i-DIST+256] ;
        }
#pragma hmppcg grid barrier
```

Set buffer size according to grid size

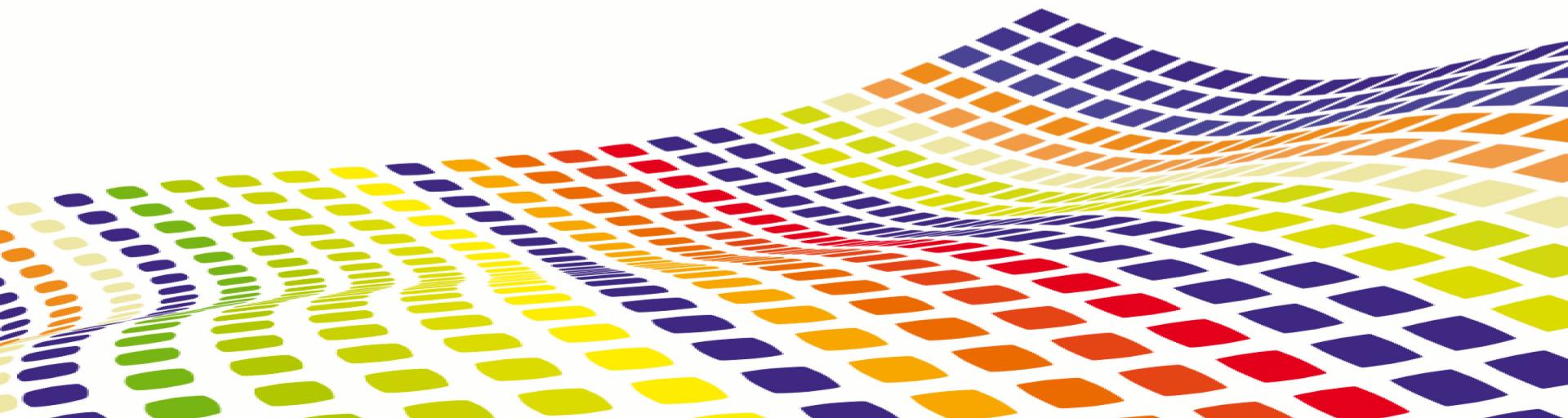
Detect grid support

Declare buf in shared memory

Parallel load in shared memory

Wait for end loading before use

HMPP For C++



HMPP++ Version With Template Support

```
int main(int argc, const char ** argv){  
    init(N, X, Y);  
  
    hmpprt::Device * device = 0;  
    bool fallback = false;  
    try{  
        device = hmpprt::DeviceManager::getInstance()->getFirstCUDADevice();}  
    catch (hmpperr::DeviceError &)  
    { fallback = true; }  
  
    if (fallback) {  
        #pragma hmppcg entrypoint as mycodelet, target=CUDA  
        Kernel<float>(N).myFunc(X,Y);}  
    else {  
        device->acquire();  
  
        hmpprt::Grouplet * grouplet = new hmpprt::Grouplet::getCurrentFileGrouplet();  
        hmpprt::Codelet * codelet = grouplet->getCodeletByName("mycodelet");  
  
        hmpprt::Data * datax = new hmpprt::Data(device, N * sizeof(float));  
        hmpprt::Data * datay = new hmpprt::Data(device, N * sizeof(float));  
  
        datax->allocate();  
        datay->allocate();  
        datay->upload(Y);  
    }  
  
    hmpprt::ArgumentList arguments(* codelet->getSignature());  
    arguments.addArgument(N);  
    arguments.addArgument(datax);  
    arguments.addArgument(datay);  
    device->call(codelet, &arguments);  
  
    datax->download(X);  
}  
  
check(N, X, Y);  
return 0;  
}
```

Indicate CUDA version of kernel to be generated as mycodelet function

Retrieve codelet in default .so generated library

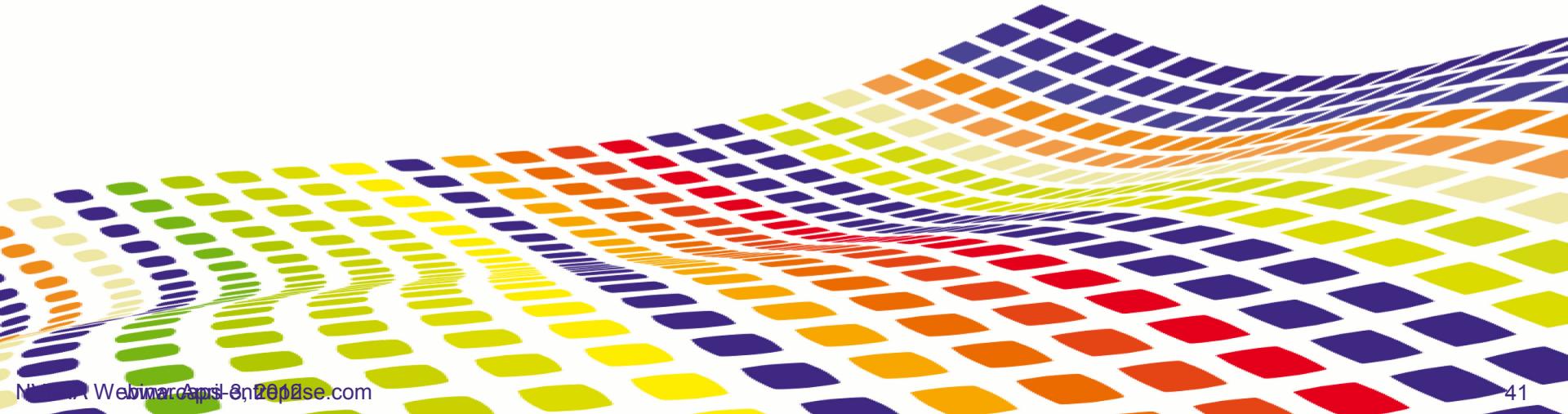
Allocate and upload data in the device

Create ArgumentList and pass each codelet argument

Execute codelet

Download result

Conclusion



Conclusion

- Software has to expose massive parallelism
 - **The key to success is the algorithm!**
 - The implementation has “just” to keep parallelism flexible and easy to exploit
- Directive-based approaches are currently one of the most promising track for heterogeneous many-cores
 - Preserve code assets
 - At node level help separating parallelism aspect from the implementation
- Official support for OpenACC in HMPP April release
- Many new targets to come this year
 - x86 CPU
 - Intel MIC
 - AMD/ATI

Many-core programming Parallelization GPGPU NVIDIA Cuda OpenHMPP

Directive-based programming Code Porting Methodology

OpenACC Hybrid Many-core Programming HPC community Petaflops

Parallel computing HPC open standard Exaflops Open CL

High Performance Computing Code speedup Multi-core programming

Massively parallel Hardware accelerators programming GPGPU

HMPP Competence Center Parallel programming interface DevDeck



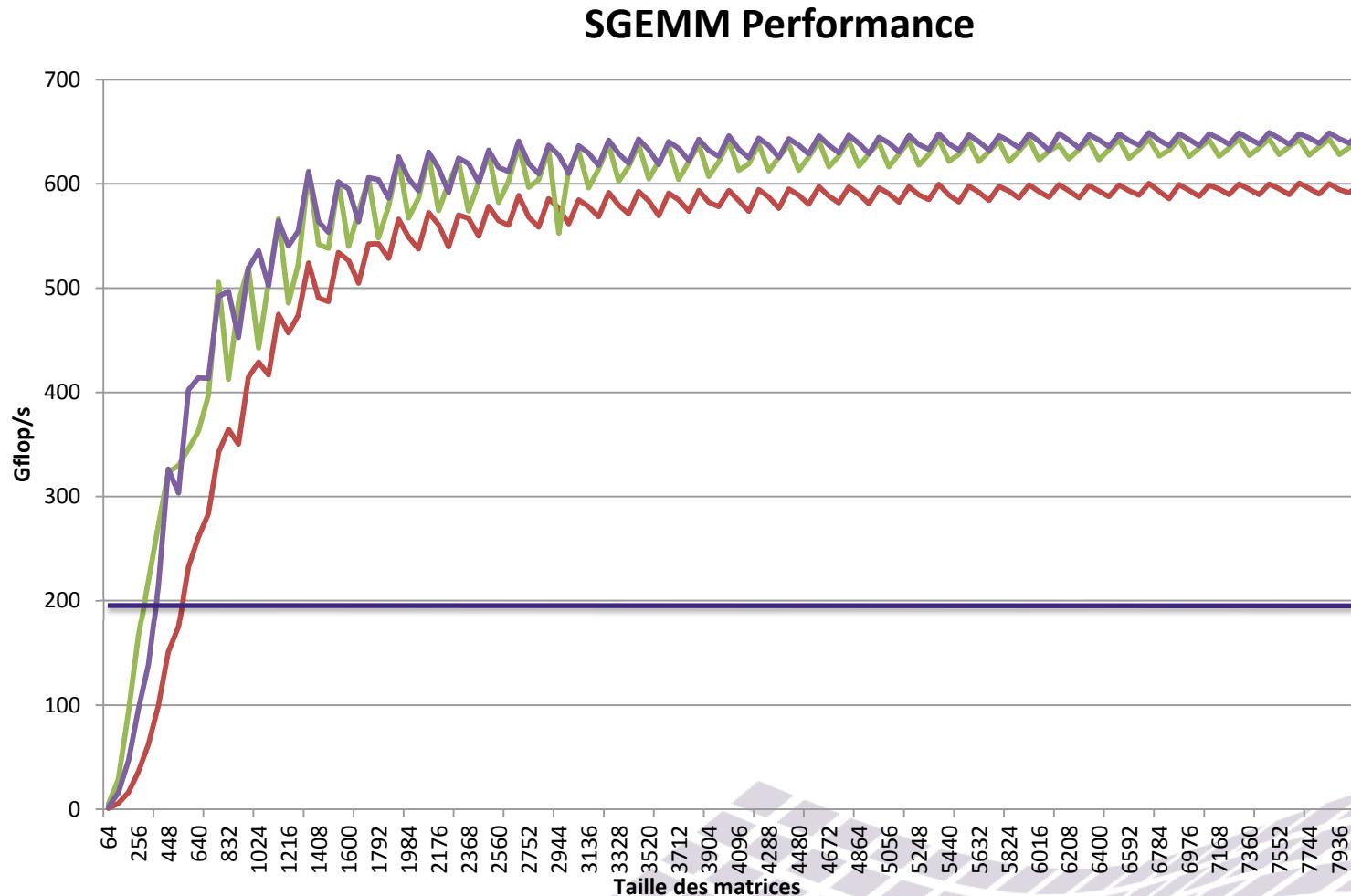
Global Solutions for
Many-Core Programming

vasnierj@ornl.gov

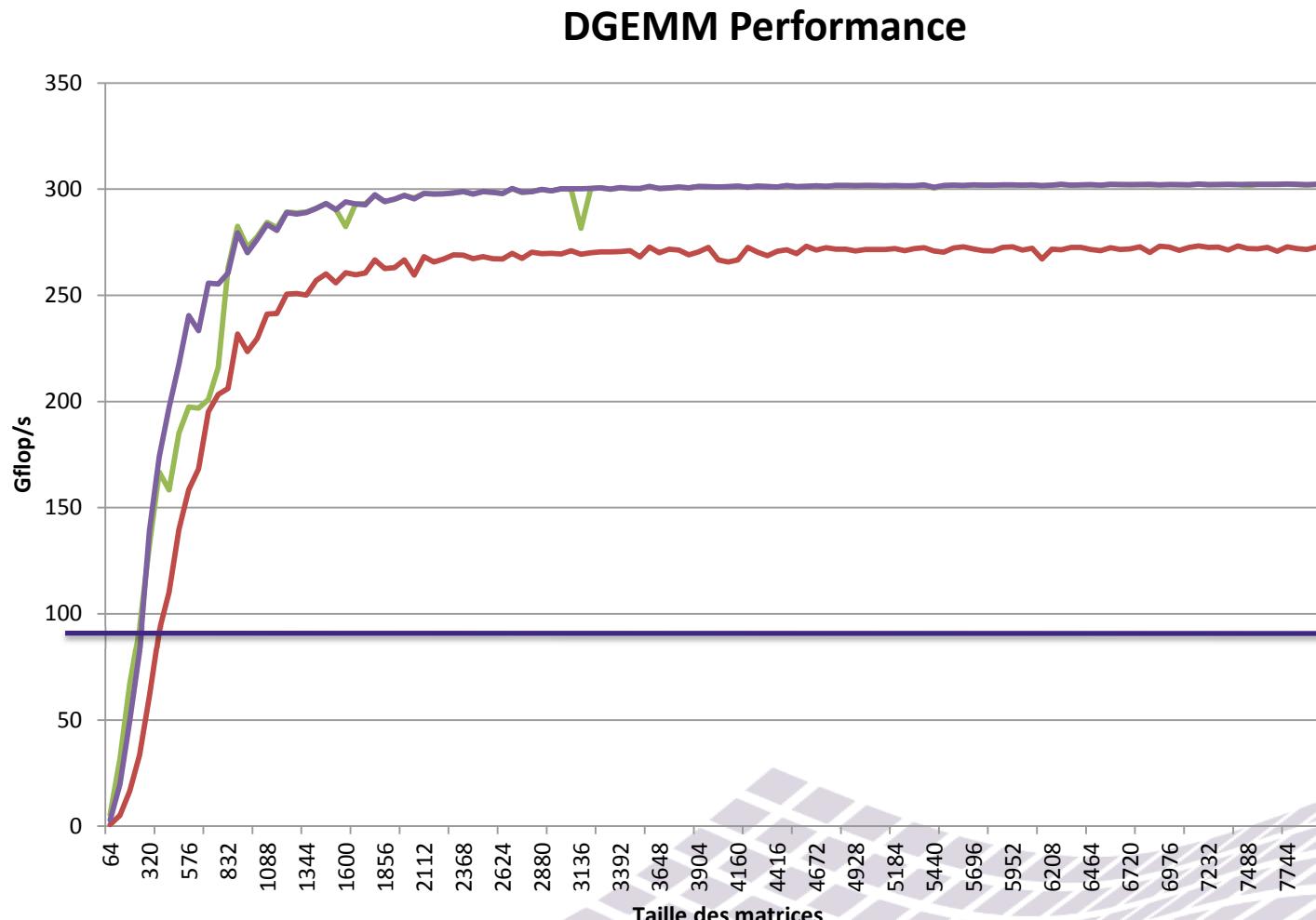
ORNL Bldg 5700 G220

<http://www.caps-enterprise.com>

HMPP BLAS Performance on NVIDIA Fermi



HMPP BLAS Performance on NVIDIA Fermi

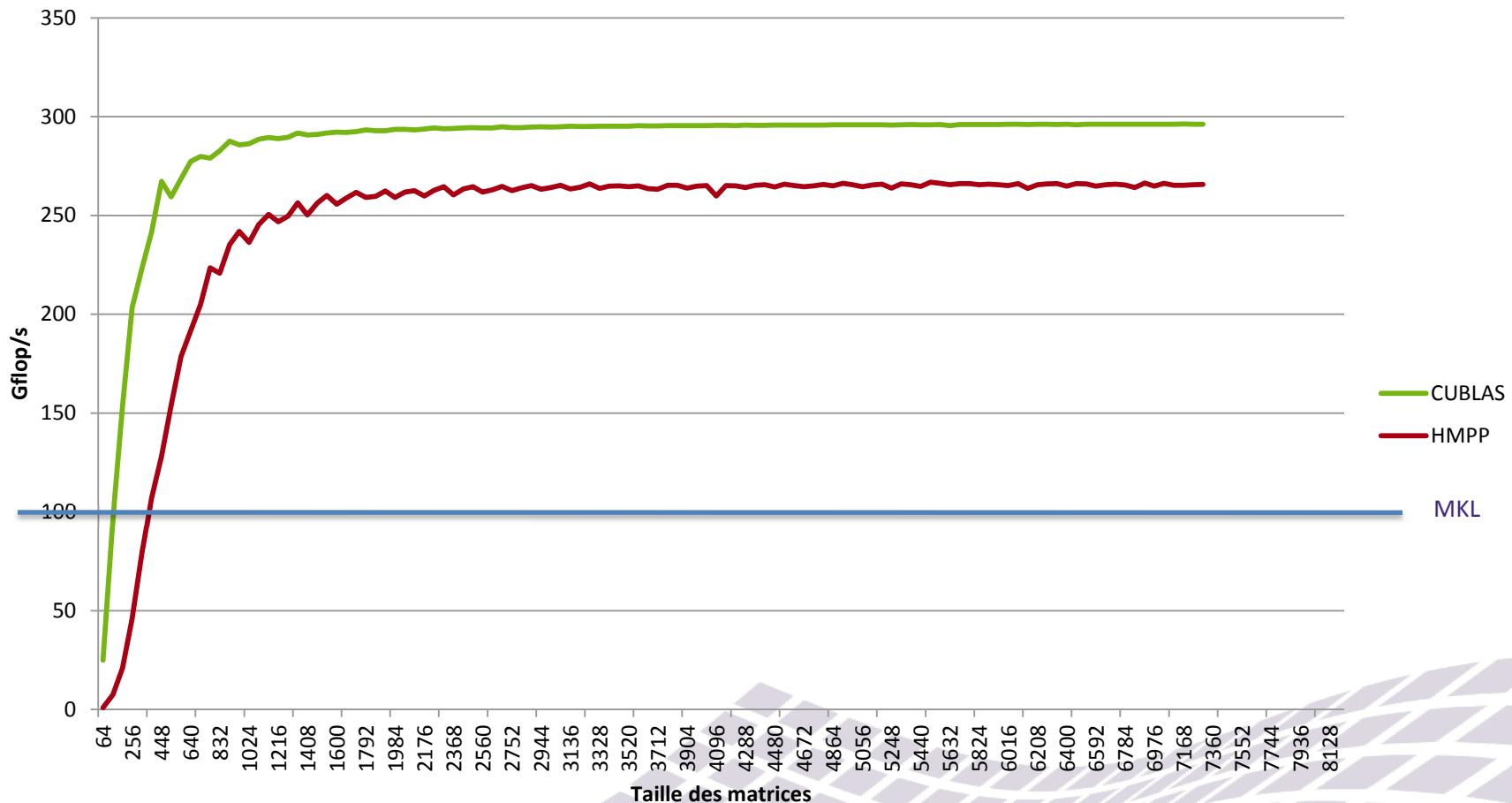


2 x Intel(R) Xeon(R) X5560 @ 2.80GHz (8 cores) - MKL
NVIDIA Tesla C2050, ECC activated – HMPP, CUBLAS, MAGMA

HMPP BLAS Performance on NVIDIA Fermi



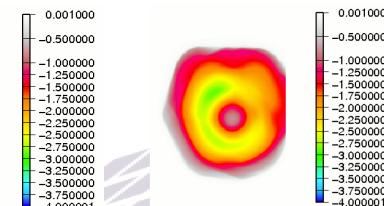
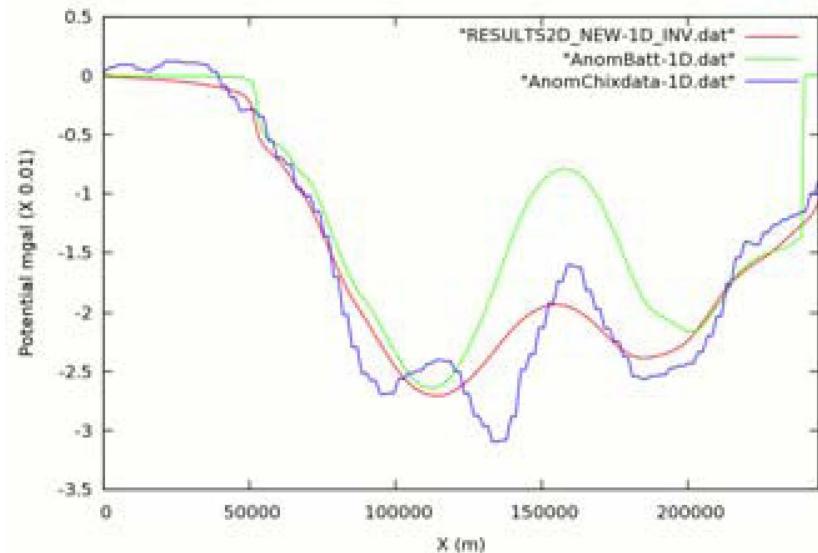
ZGEMM Performance



Geophysics

3D Poisson equation conjugate gradient

- Resource spent
 - 2 man-month
- Size
 - 2kLoC of F90 (DP)
- CPU improvement
 - X1,73
- GPU C1060 improvement
 - x 5,15 over serial code on Nehalem
- Main porting operation
 - highly optimizing kernels

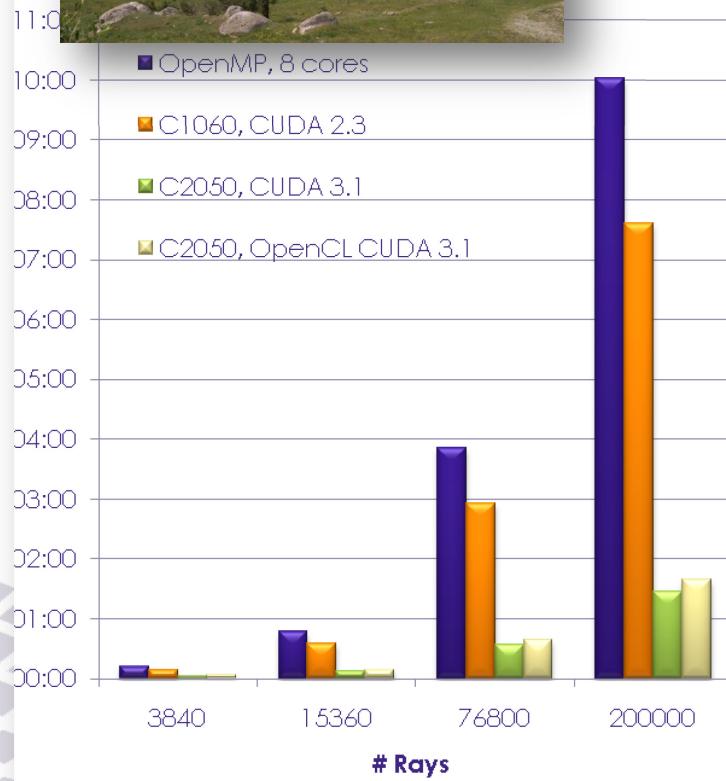


The finite volumes method (left) is more accurate than the analytic solution (right)
which over estimates the central peak

Alternative Energy

Monte Carlo simulation for thermal radiation

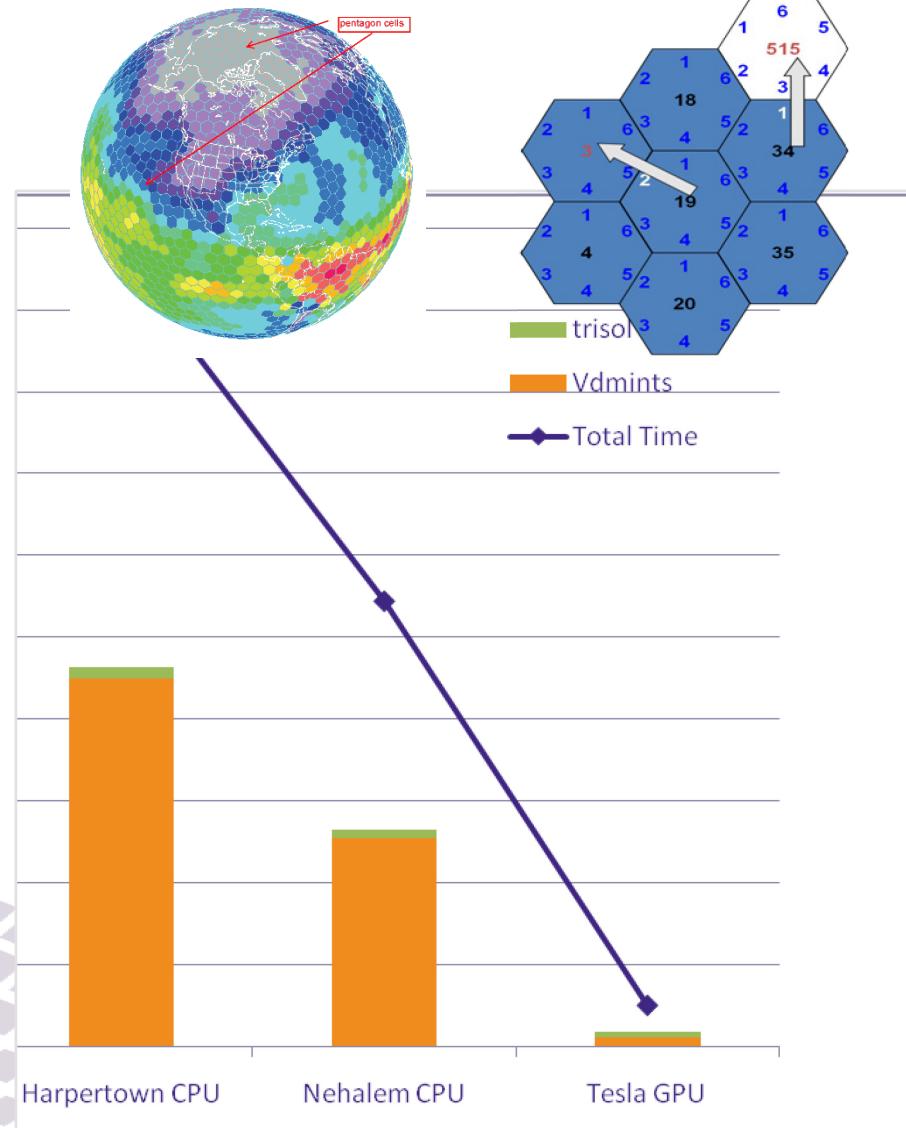
- Resource spent
 - 1,5 man-month
- Size
 - ~1kLoC of C d (DP)
- GPU C2050 improvement
 - x6 over 8 Nehalem cores
- Full hybrid version
 - **x23 with 8 nodes compare to the 8 core machine**
- Main porting operation
 - adding a few HMPP directives



Weather Forecasting

A global cloud resolving model

- Resource spent
 - 1 man-month (part of the code already ported)
- GPU C1060 improvement
 - 11x over serial code on Nehalem
- Main porting operation
 - reduction of CPU-GPU transfers
- Main difficulty
 - GPU memory size is the limiting factor



Computer vision & Medical imaging

MultiView Stereo

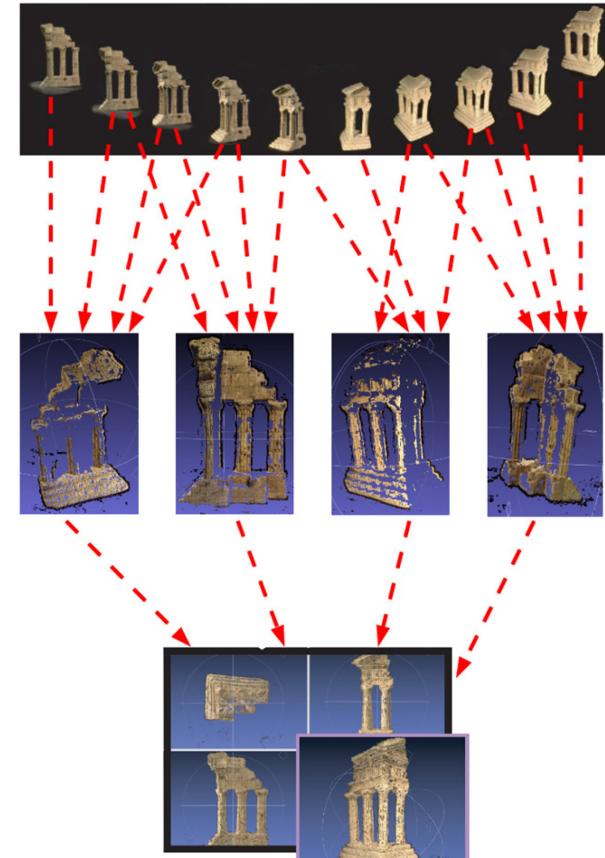
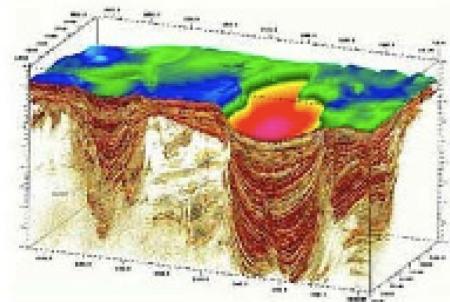


Image processing

Edge detection algorithm



GPU-accelerated seismic depth imaging



GPU accelerated Rack

Rack



4.4 CPU Racks



performance



Native Function Call in Codelets

- User can call hand-written CUDA or OpenCL kernels
 - They are seen as `__device` functions

native.c

```
float sum( float x, float y ) {
    return x+y;
}

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
{
    #pragma hmppcg native, sum
    for( int i = 0 ; i < N ; i++ )
        B[i] = sum( A[i], B[i] );
    ...
}
```

Compilation

```
$ hmpp --native=native.xml gcc native.c -o native.exe
```

External Function Call in Codelets

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 a CUDA version

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
{
    #pragma hmppcg extern, sum
    for( int i = 0 ; i < N ; i++ )
        B[i] = sum( A[i], B[i] );
    }
    ...
}
```

Make a call in the region

Compilation

```
$ hmpp gcc -c sum.c -o sum.o
$ hmpp gcc -c extern.c -o extern.o
$ hmpp gcc sum.o extern.o -o extern.exe
```