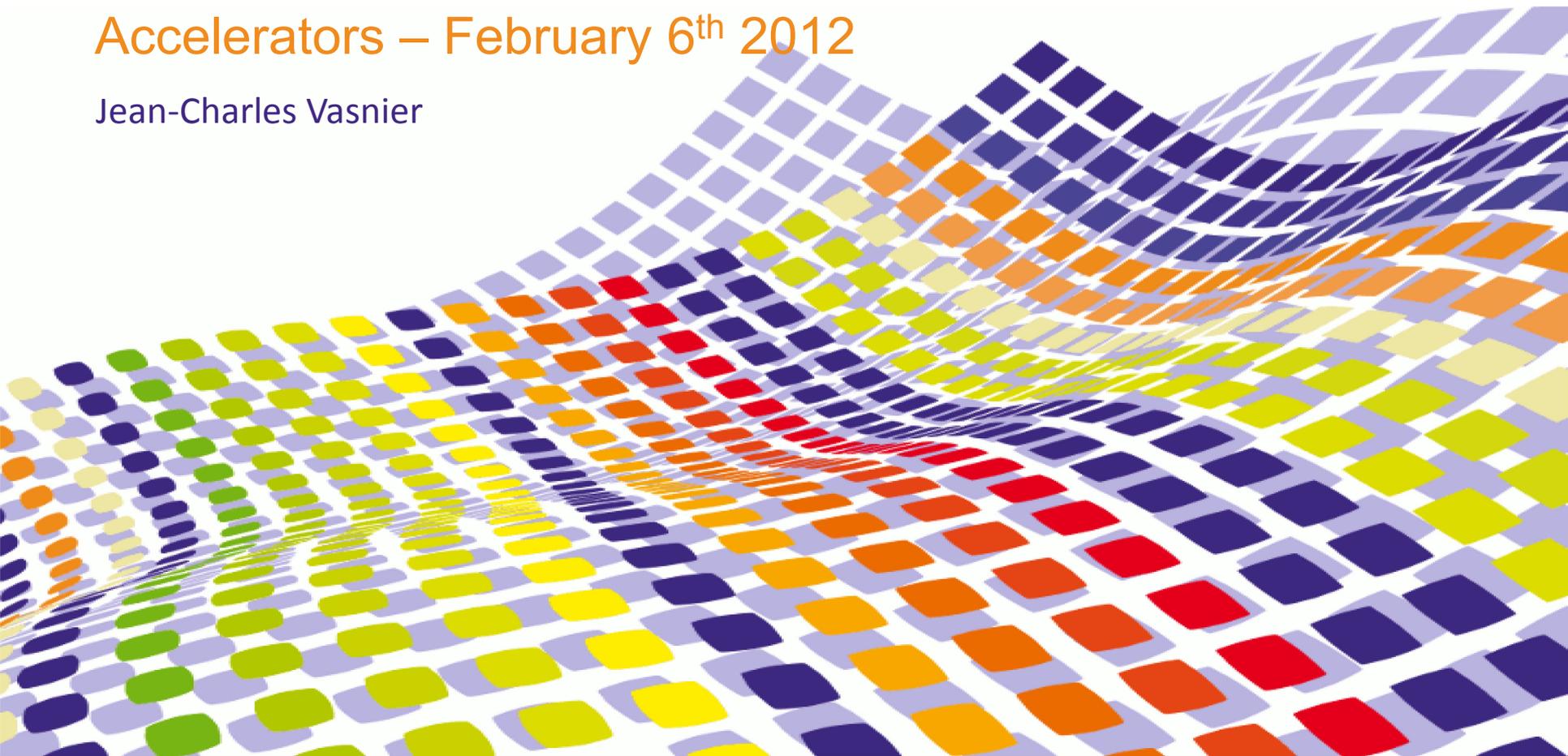


HMPP Methodology and Tools for Porting Legacy Code to Manycore

Electronic Structure Calculation Methods on Accelerators – February 6th 2012

Jean-Charles Vasnier



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?

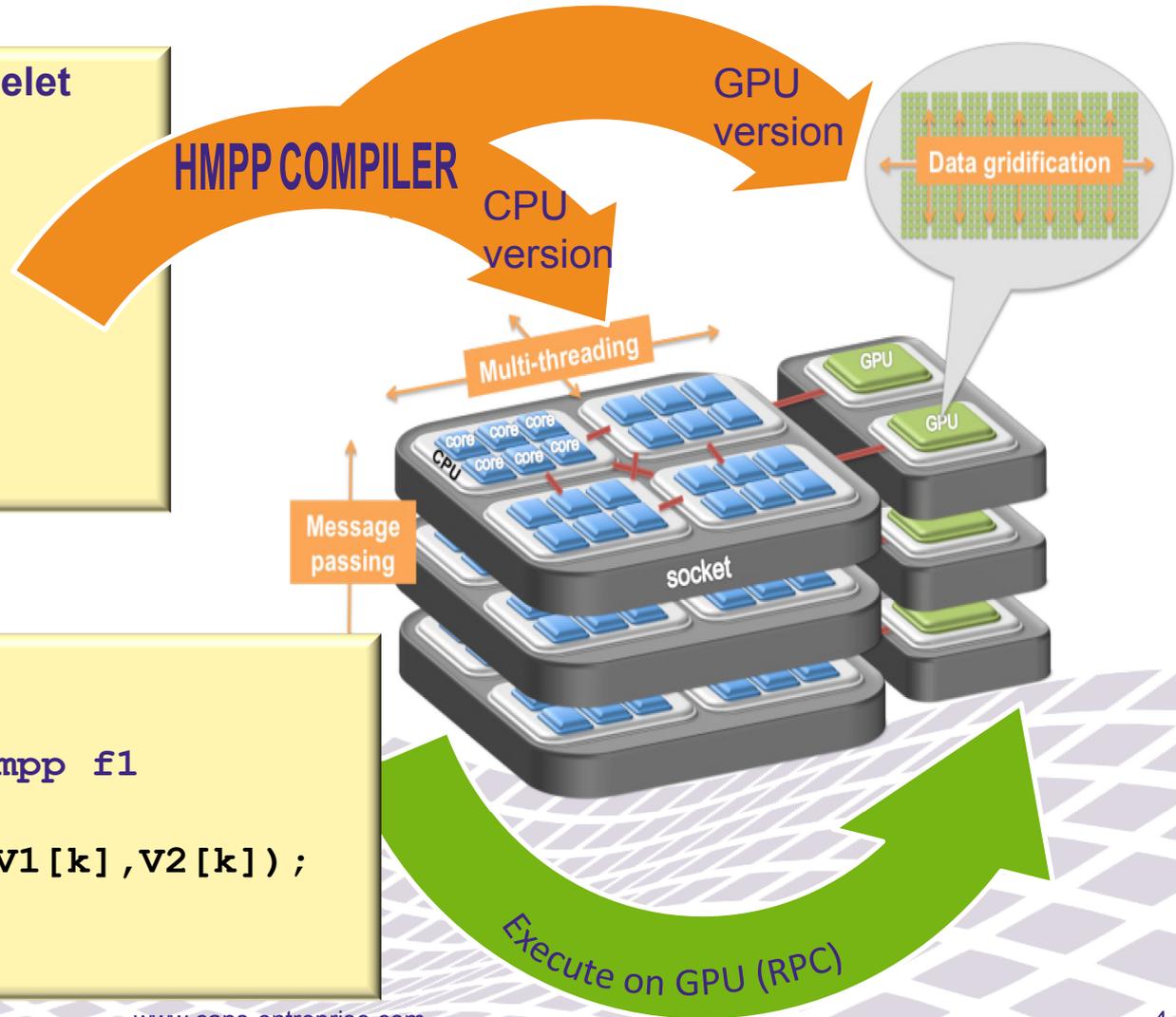
- **Dynamic data management mechanism**
 - Mirrors identified by their host address
 - Simplifies management of data with less directives
- **Multi-device programming**
 - Exploit multiple devices in one compute node
 - Distribute collections of data over multiple devices
- **New run-time API**
 - Three 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.

Scope of HMPP 3.0 Programming

- Remote procedure calls (RPCs) on accelerator devices
 - Parallel loop nests to exploit multiple computing units

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

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

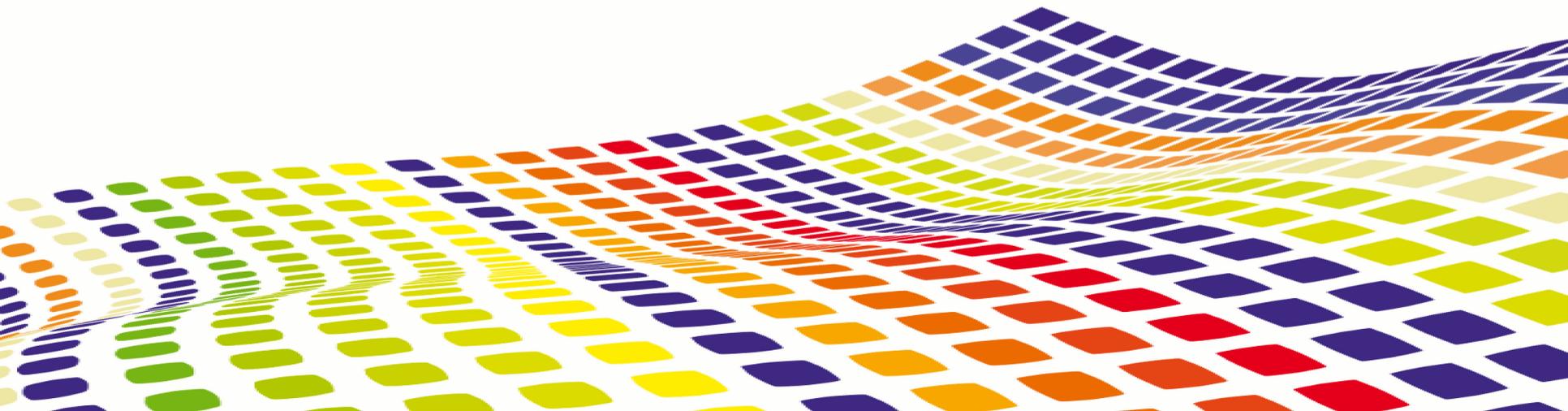


To give developers a high level abstraction for manycore programming

- Incrementally program manycore applications
- Preserve legacy code and keep application hardware and software independent
- Maintain single software source
- Ensure application portability and interoperability

- **C and Fortran GPU programming directives**
 - Define and execute GPU-accelerated versions of code
 - Optimize CPU-GPU data movement
 - Complementary to OpenMP and MPI
- **A source-to-source hybrid compiler**
 - Generate CUDA and OpenCL kernels
 - Works with standard compilers and target tools
 - Tuning directives to optimize GPU kernels
- **A runtime library**
 - Allocate and manage computing resources
 - Dispatch computations on CPU and GPU cores
 - Scale to multi-GPUs systems

HMPP Programming



HMPP Programming and Tuning

- GPU Programming directives in legacy code
 - Declare and generate GPU versions of computations: HMPP codelets
 - Optimize data movement
 - Use asynchronous capabilities

- GPU Tuning directives in codelets
 - Unleash performance
 - Fully exploit GPU stream architecture

```
#pragma hmppcg(CUDA) unroll(8), jam
```

```
!$hmppcg parallel
```

```
#pragma hmpp label command [ , attribute ... ]
```

```
!$hmpp label command [ , attribute ... ]
```

Label identifies directives that belong to one codelet declaration and execution

OpenACC[®]

DIRECTIVES FOR ACCELERATORS

Support in Q2 2012

- 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

Accelerate Regions

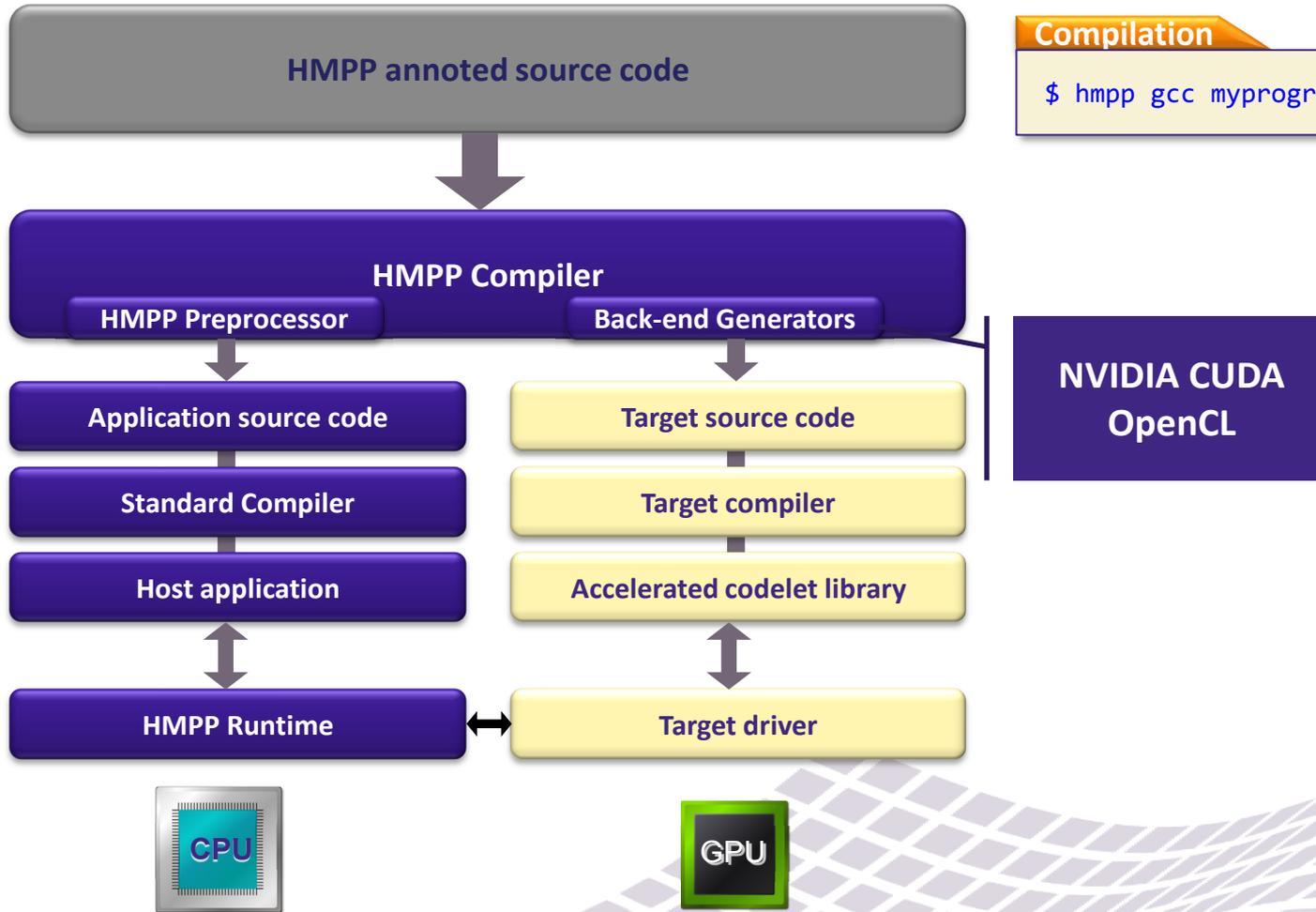
- Codelets are built from declaration of accelerated regions

```
!$hmp MyRegion region, target=CUDA, args[A].io=inout, args[B;C].io=in,  
private=[i,j]  
  
DO i=1,N  
  DO j=1,N  
    A(i,j) = A(i,j) + B(i,j) * C(i,j) ;  
  ENDDO  
ENDDO  
  
!$hmp MyRegion endregion
```



Declare region intent
parameters

HMPP Compilation



Compilation

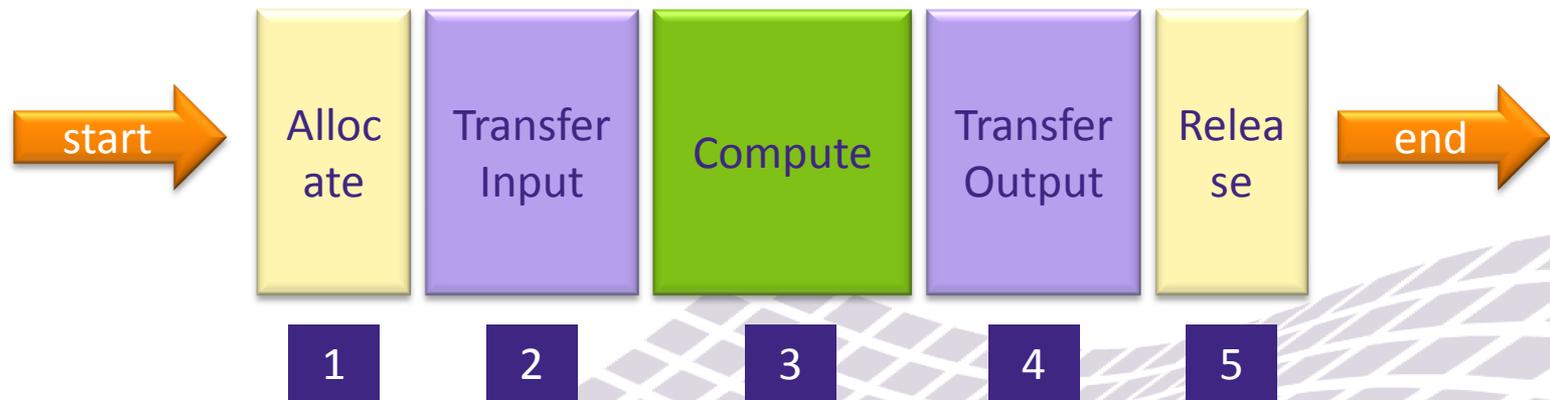
```
$ hmpp gcc myprogram.c
```

NVIDIA CUDA
OpenCL

RPC Sequence

- **RPC = Remote Procedure Call**

1. Allocate HWA and memory
2. Download input data
3. Call
4. Upload result output
5. Release HWA and memory



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

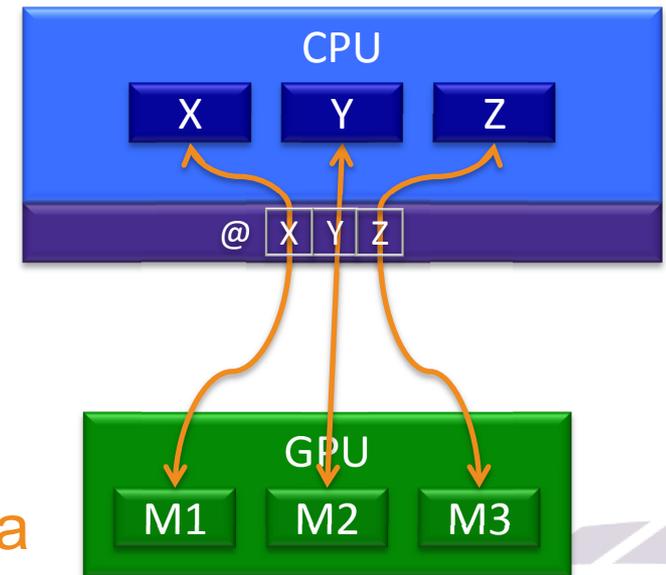
```
!$hmpp <label> <directive type> [, <directive parameter>]* [&]
```

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

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

Storage Policy

- **Mirrored data or simply mirror**
 - An area of memory on the host is mirrored on the accelerator
 - The HMPP runtime dynamically makes the link between the host address and the device address
- **Simple data management**
 - Few directives to manage mirrored data
- **Easy to dynamically allocate and free a mirror**
 - Use the ALLOCATE and FREE directives



Data Mirroring Example

- A simple mirror example with multiple callsites

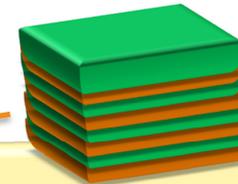
```
...
float vin1[size][size], vin2[size][size], vout[size][size];
...
//Allocate the mirrors for vin1, vin2 and vout
#pragma hmpp allocate, data[vin1, ...], size={size,size}

//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]
...
```

Multiple Devices and Data Distribution

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



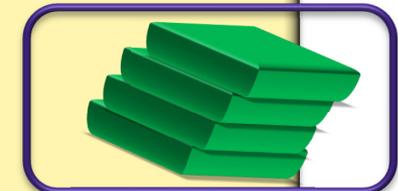
3D

```
...  
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 sgemv callsite  
      sgemv( vin1[i], vin2[i], vout[i] );  
    ...  
    //Get back the result  
    #pragma hmpp delegatedstore, data["vout[i]"]  
  }  
...
```

GPU 0

2D



GPU 1

- Inlining of functions defined in same source file
- External functions
 - C/Fortran functions
 - CUDA/OpenCL version generated by HMPP
 - Can be defined in another compilation unit (avoid code duplication)
- Native pure CUDA/OpenCL functions
 - Hand-written CUDA/OpenCL functions
 - Can be seen as CUDA `__device` function calls in HMPP-generated codelets
 - There are not CUDA kernels nor library functions

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 */
```

Declare a CUDA version

sum.c

```
#include "sum.h"

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

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

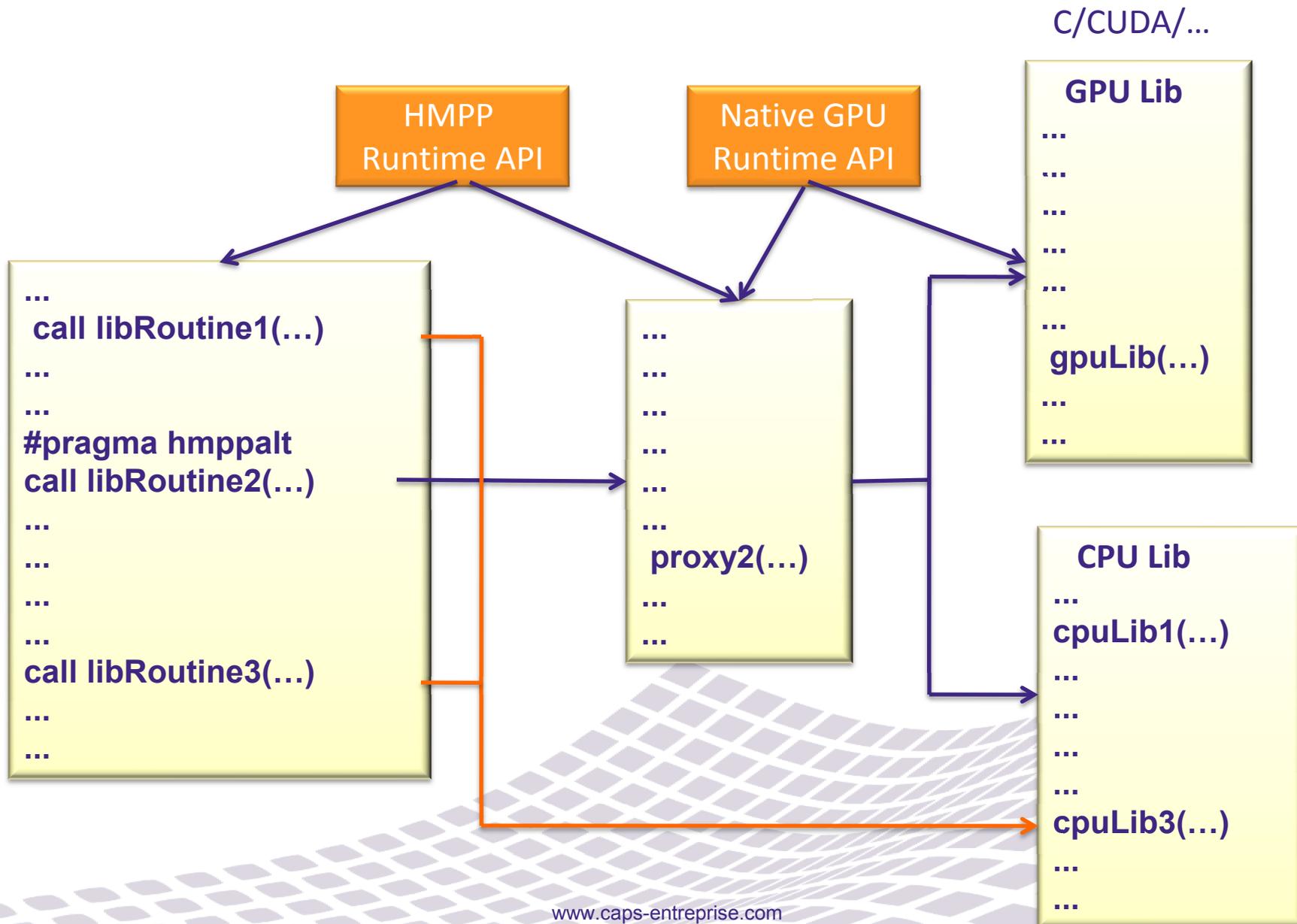
Compilation

```
$ 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
```

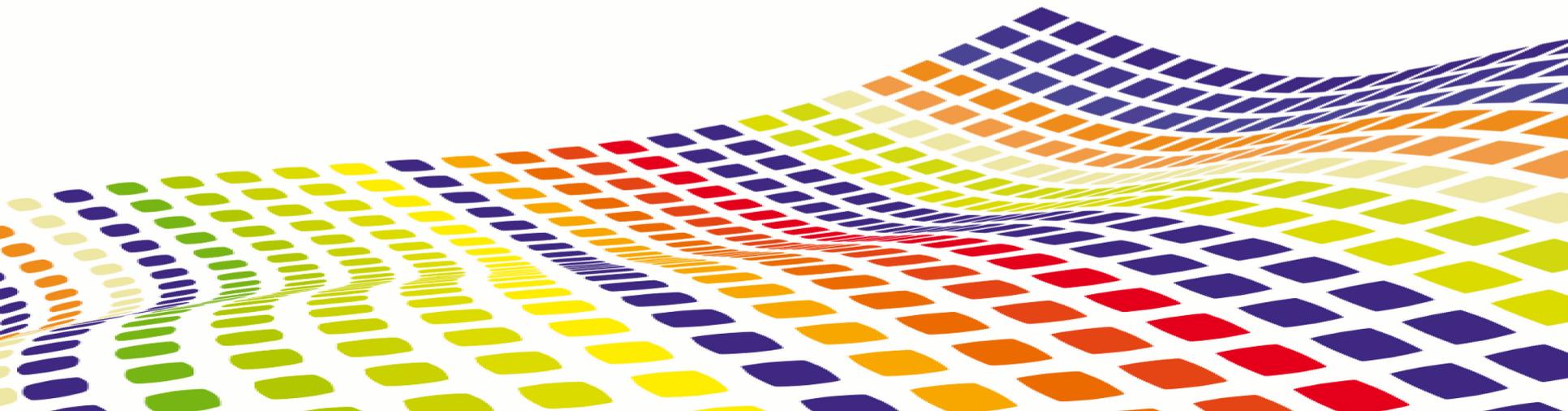
Library Integration in HMPP 3.0

- Legacy codes are full of 'standard' library calls
 - BLAS, LAPACK, FFTW, ...
- Those libraries were not designed for GPU
 - But still want to keep them for CPU execution and debugging purpose
- Similar libraries are developed
 - cuBLAS, cuFFT, ...
 - But APIS are usually different so libraries are not interchangeable
- Need for non-intrusive mechanisms to translate library calls
 - Directive based
 - To keep the CPU version operational
 - Compatible with HMPP
 - So HMPP codelets can share data with cuBLAS, cuFFT, ...

Library Integration in HMPP 3.0



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

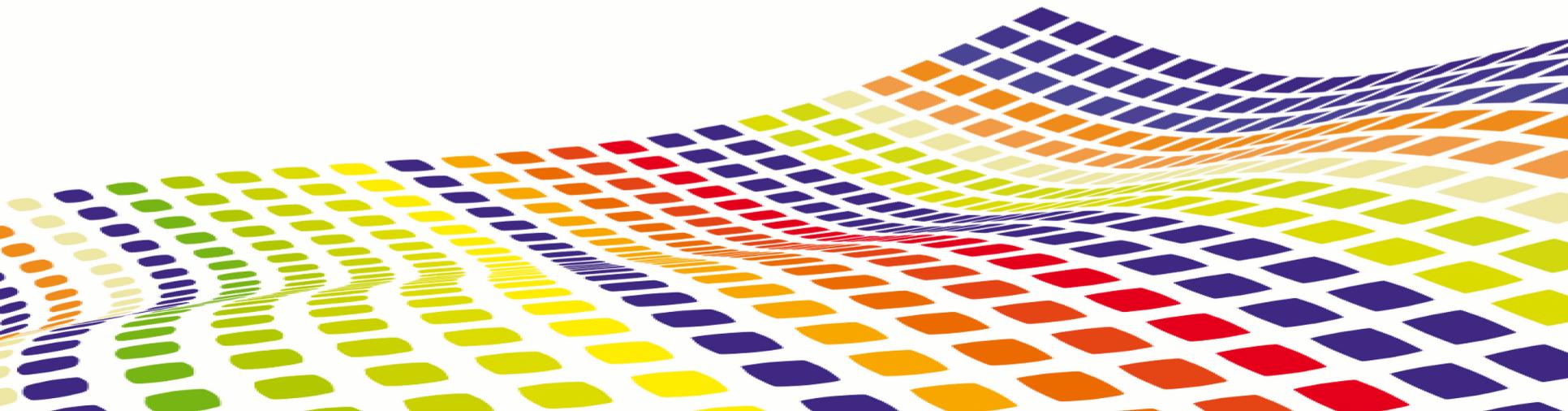
Detect grid support

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 (hmpprt::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

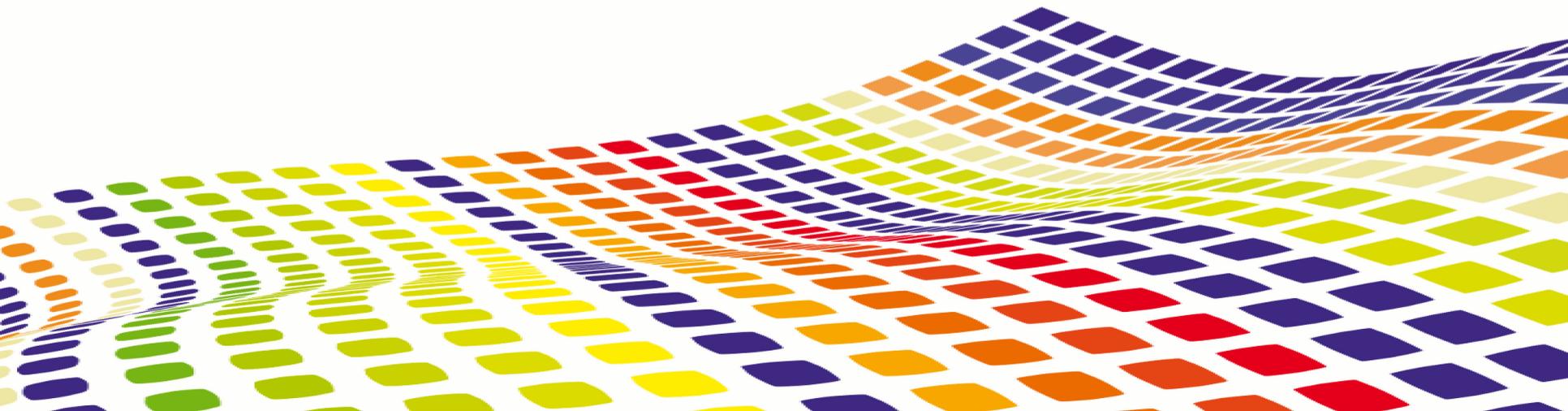
Allocate and upload data in the device

Create ArgumentList and pass each codelet argument

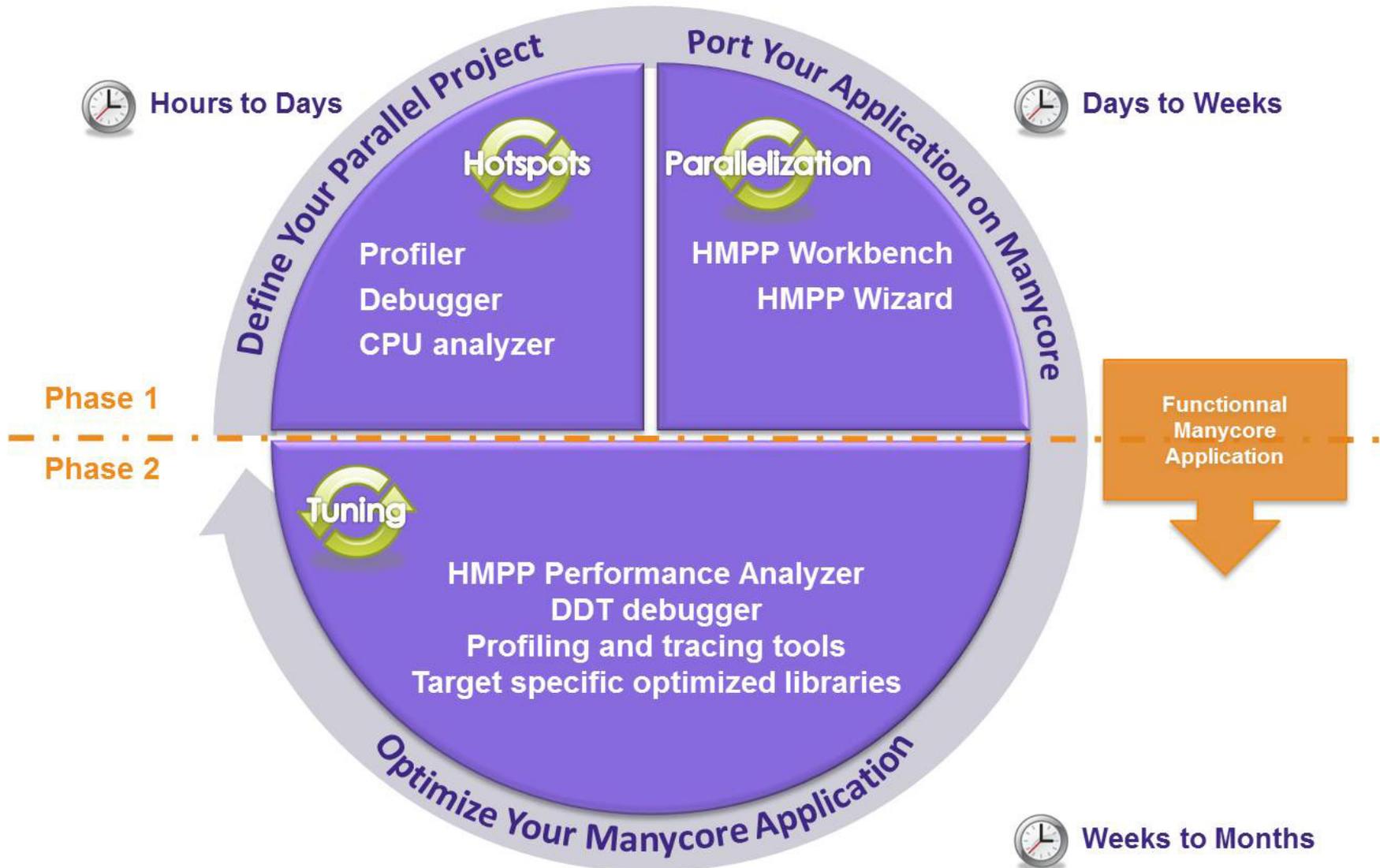
Execute codelet

Download result

Porting Methodology and Development Tools

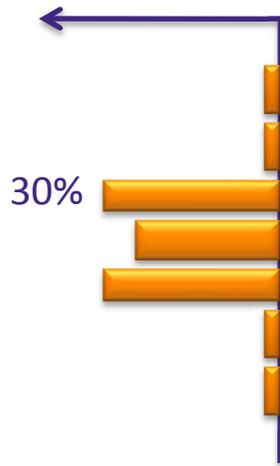
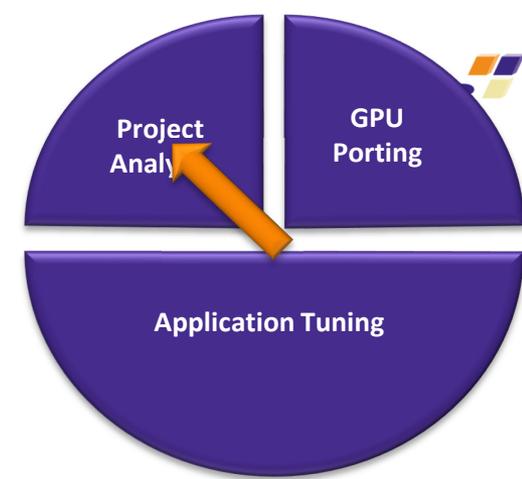


Methodology to Port Applications to Manycore



Step One: Find Hot Spots

```
void derive(int nx, double _Complex ...) {  
    int i;  
    for (i=1; i<nx/2; ++i) {  
        wrkq[i] = (0+i-1) * wrkq[i] * cf;  
    }  
    wrkq[ 0] = 0.0+i*0;  
    wrkq[nx/2] = 0.0+i*0;  
}
```

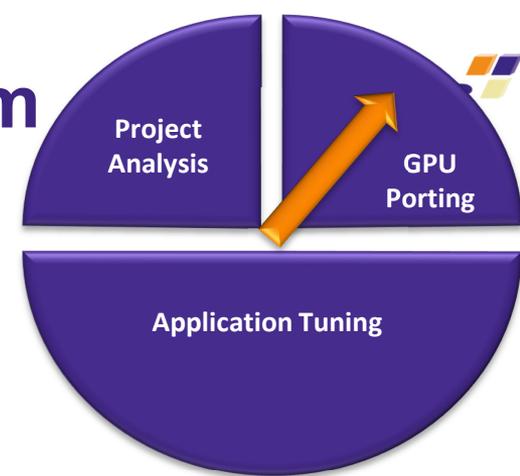


```
...  
pr2c = fftw_plan_dft_r2c_1d(n, idata_real, ...  
pc2r = fftw_plan_dft_c2r_1d(n, odata_intermediate, ...  
fftw_execute(pr2c);  
derive(n, odata_intermediate, cf);  
fftw_execute(pc2r);  
fftw_destroy_plan(pr2c);  
fftw_destroy_plan(pc2r);  
...
```

- Find hotspots, estimate potential (e.g. Amdahls' Law)
- Check CPU performance, optimize CPU execution
- Setup a validation process
- Estimate parallelism, complexity, ...

Initial Porting, Highlighting Parallelism

- Exhibit parallelism
- Push the code onto the GPU
- Validate execution



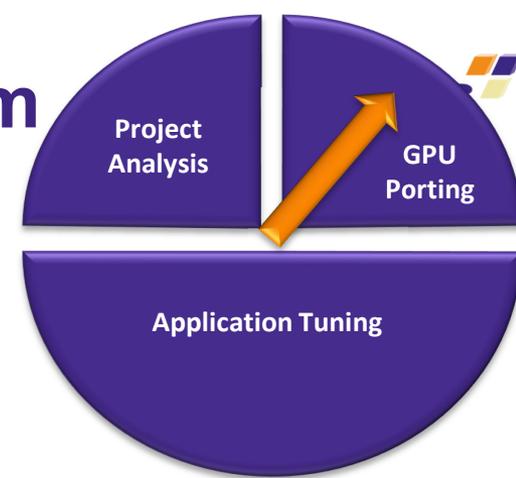
```
#pragma hmpp <g> group, target=CUDA[/OpenCL]  
#pragma hmpp <g> derive codelet, args[*]transfer=atcall
```

```
#pragma hmpp <g> group, target=CUDA[/OpenCL]  
#pragma hmpp <g> derive codelet, args[*].transfer=atcall  
void derive(int nx, double _Complex ...) {  
    int i;  
    for (i=1; i<nx/2; ++i) {  
        wrkq[i] = (0+i-1) * wrkq[i] * cf;  
    }  
    wrkq[ 0] = 0.0+i*0;  
    wrkq[nx/2] = 0.0+i*0;  
}
```

Build a GPU version
of the function

Initial Porting, Highlighting Parallelism

- Select implementation for library calls and hotspots
- Insert calls to execute on GPU



Call GPU version of *derive*

```
#pragma hmpp <g> derive callsite  
derive(n, odata_intermediate, cf);
```

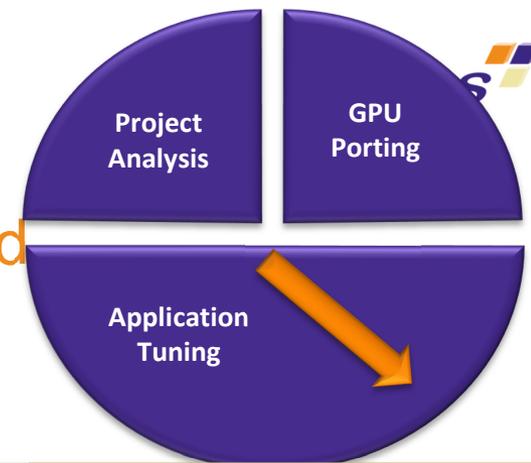
```
#pragma hmppalt cufft call, name="fftw_plan_dft_r2c_1d"  
pr2c = fftw_plan_dft_r2c_1d(n, idata_real, ...);  
#pragma hmppalt cufft call, name="fftw_plan_dft_c2r_1d"  
pc2r = fftw_plan_dft_c2r_1d(n, odata_intermediate, ...);  
#pragma hmppalt cufft call, name="fftw_execute"  
fftw_execute(pr2c);  
#pragma hmpp <g> d  
derive(n, odata_intermediate, cf);  
...  
#pragma hmpp  
...
```

Call GPU version of library call

```
#pragma hmppalt cufft call, name="fftw_execute"  
fftw_execute(pr2c);
```

Transfer Optimizations

- Reduce CPU-GPU communication overhead
- Exploit reuse of data on the GPU



Preload data

```
int main(int argc, char **argv) {  
#pragma hmpp sgemm acquire  
#pragma hmpp sgemm allocate, data[vin1;vin2;vout], size={size,size}
```

```
    . . .  
#pragma hmpp sgemm advancedload, data[vin1;vin2;vout]
```

```
    for( j = 0 ; j < 1000 ; j++ ) {  
#pragma hmpp sgemm  
        sgemm( size, size, size, alpha, vin1, vin2, beta, vout );  
    }
```

Iterate 1000 times
without data transfer

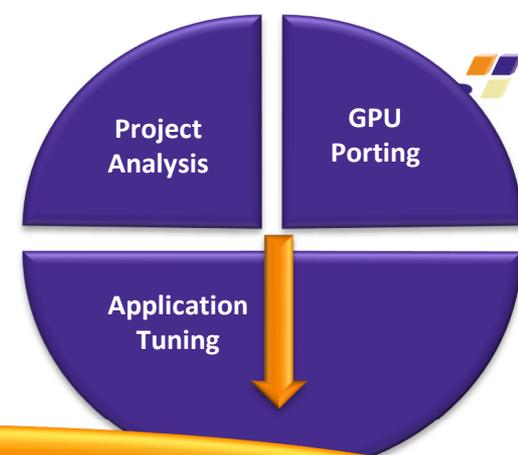
```
    . . .  
#pragma hmpp sgemm delegatedstore, data[vout]
```

```
#pragma hmpp sgemm free  
#pragma hmpp sgemm release
```

Download results

Improving Code Generation

- Directive-based GPU kernel code transformations



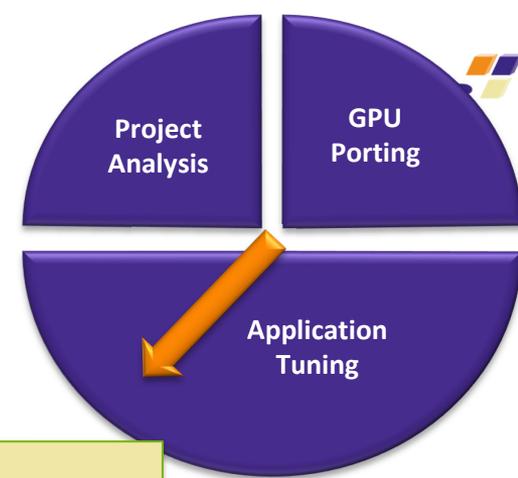
#pragma hmppcg unroll(4),split, noremainder

```
#pragma hmppcg unroll(4), jam(2), noremainder
for( j = 0 ; j < p ; j++ ) {
  #pragma hmppcg unroll(4), split, noremainder
  for( i = 0 ; i < m ; i++ ) {
    double prod = 0.0;
    double v1a,v2a ;
    k=0 ;
    v1a = vin1[k][i] ;
    v2a = vin2[j][k] ;
    for( k = 1 ; k < n ; k++ ) {
      prod += v1a * v2a;
      v1a = vin1[k][i] ;
      v2a = vin2[j][k] ;
    }
    prod += v1a * v2a;
    vout[j][i] = alpha * prod + beta * vout[j][i];
  }
}
```

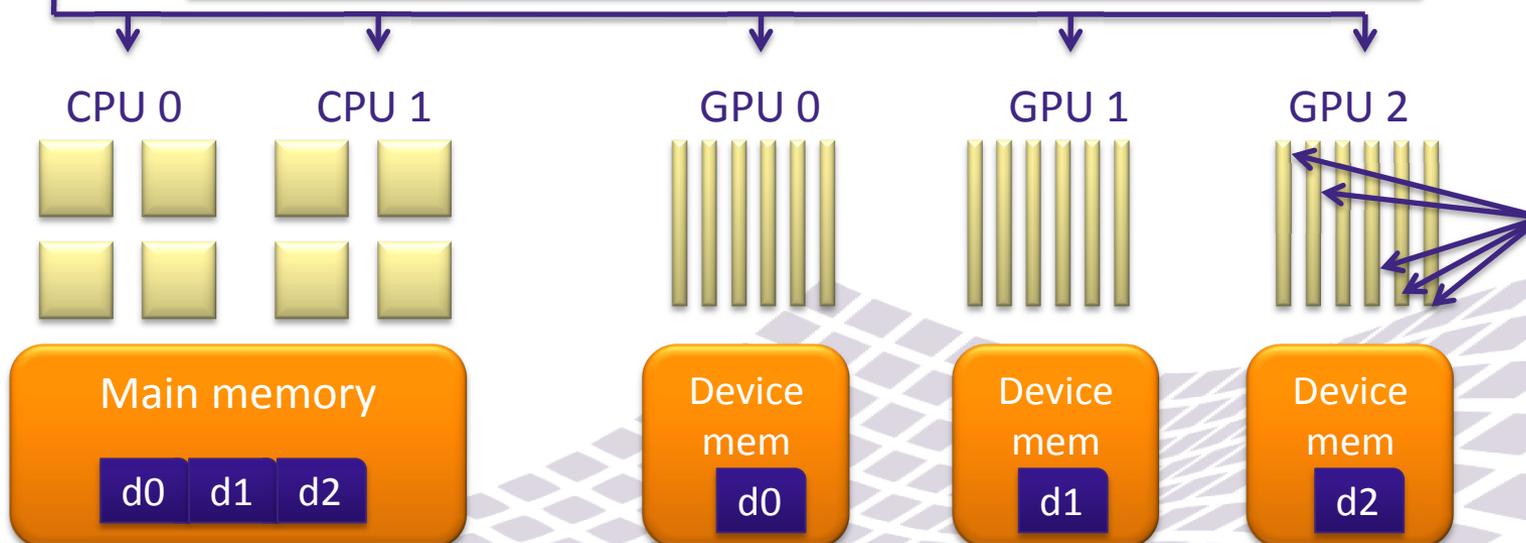
Use pragma to preserve CPU code

Scaling to Many-many cores

- Spread computations on available devices
- Manage data over several memory spaces



```
float data[n][x][y];  
#pragma hmp parallel, device="k%3"  
for(k=0;k<n;k++) {  
    #pragma hmp <MyGroup> f1 callsite  
    myparallelfunc(&data[k],n);  
}
```



HMPP Wizard and Performance Analyzer

Welcome to HMPP Wizard » Advice Results

GPU library usage detection

Filters Results Advice54

Close this tab

```
88 double t_create1 = ctkRealTimer();
89 pr2c = fftw_plan_dft_r2c_1d(n, idata_real, odata_intermed
FTW_ESTIMATE);
90 pc2r = fftw_plan_dft_c2r_1d(n, odata_intermediate, odata_real_CPU
FTW_ESTIMATE);
91 double t_create2 = ctkRealTimer();
92
93
94 double t_exec_pr2c1 = ctkRealTimer();
95 fftw_execute(pr2c);
96 double t_exec_pr2c2 = ctkRealTimer();
97
98
99 double t_filter1 = ctkRealTimer();
100 filter(n, (double_Complex *) odata_intermediate, cf);
101 double t_filter2 = ctkRealTimer();
102
103
104 double t_exec_pc2r1 = ctkRealTimer();
```

Detected potential issue

HMPP-ALT-FFT/VERSION1
/exec_D1Z_Z2D.c @line 95 - Advice54: A call to the standard FFTW function "fftw_execute" has been detected inside a function.

Advice

Consider using an optimized library for you application with the HMPP ALT proxy.

```
11 for (i = 1; i < M-1; ++i) // 2
12 {
13     for (j = 1; j < N -1; ++j) // 1
14     {
15         int a = rename(A, A);
16         A[i-1][j-1] = a ;
17     }
18 }
19 }
20
21 #pragma hmpp initLoop codelet, target=CUDA
22 void initLoop(int M, int N, real A[N][M])
23 {
24     int i, j;
25     for (i = 1; i < M-1; ++i) // 2
26     {
27         for (j = 1; j < N -1; ++j) // 1
28         {
29             A[i-1][j-1] = 3.14 ;
30         }
31     }
32 }
33
34 #pragma hmpp loopUnrolled codelet, target=CUDA
```

Low computation density

Access to MyDevDeck web online resources

Detected potential issue

sample/data/src/mycode.c @line 25 - Advice2: The computation density is low.

Loop Statistics

- Number of array access: 1
- Number of operations: 2 including 0 flops
- Number of intrinsic operations: 0 including 0 flops

For more details, click here

Advice

- The computation may fetch few

Performance Analyzer view

- GPU execution Feedback
 - Based on CUDA profile
 - Several executions may be necessary to collect all the information
- Synthesize metrics based on GPU execution profile

Welcome to HMPP Wizard » Performance Analyzer

Filters Results #40:sgemm

Close this tab

```
9
10 typedef float (*array2D)[1];
11 typedef const float (*c_array2D)[1];
12
13 #define NB_FLOP(n) (2*n*n*n)
14 #define DATA_TYPE float
15
16 #pragma hmpp sgemmIII codelet, target=CUDA, args[vout].io=InOut
17 void sgemm( int m, int n, int k2, float alpha, const float vin1[n][n], const float vin2[n][n], float beta, float vout[n][n] ) {
18     int j;
19
20     /* TO DO: Add unrolling 8 times and jam on 2 levels */
21     for( j = 0; j < n; j++ ) {
22         int i;
23         /* TO DO: Add unrolling 8 times, split mode */
24         for( i = 0; i < n; i++ ) {
25             int k;
26             float prod = 0.0f;
27             for( k = 0; k < n; k++ ) {
28                 prod += vin1[k][i] * vin2[j][k];
29             }
30             vout[j][i] = alpha * prod + beta * vout[j][i];
31         }
32     }
33 }
34
35 int * getSizes( int from, int to ) {
36     // Compute number of samples
37     #line 7899 "mc.c"
38     int i, i_prev, nb_samples;
```

Codelet "sgemm"

- Kernel #1
 - Grid: 2D Loop gridification
 - Kernel name: void hmpp_codelet__sgemm_loop0_<32u, 4u, (unsigned short)24, (unsigned short)32, (unsigned short)8>(float*, float*, float*)
 - Real kernel name: _Z26hmpp_codelet__sgemm_loop0_ILj32ELj4ELt24ELt32ELt8EEvPFS0_S0_

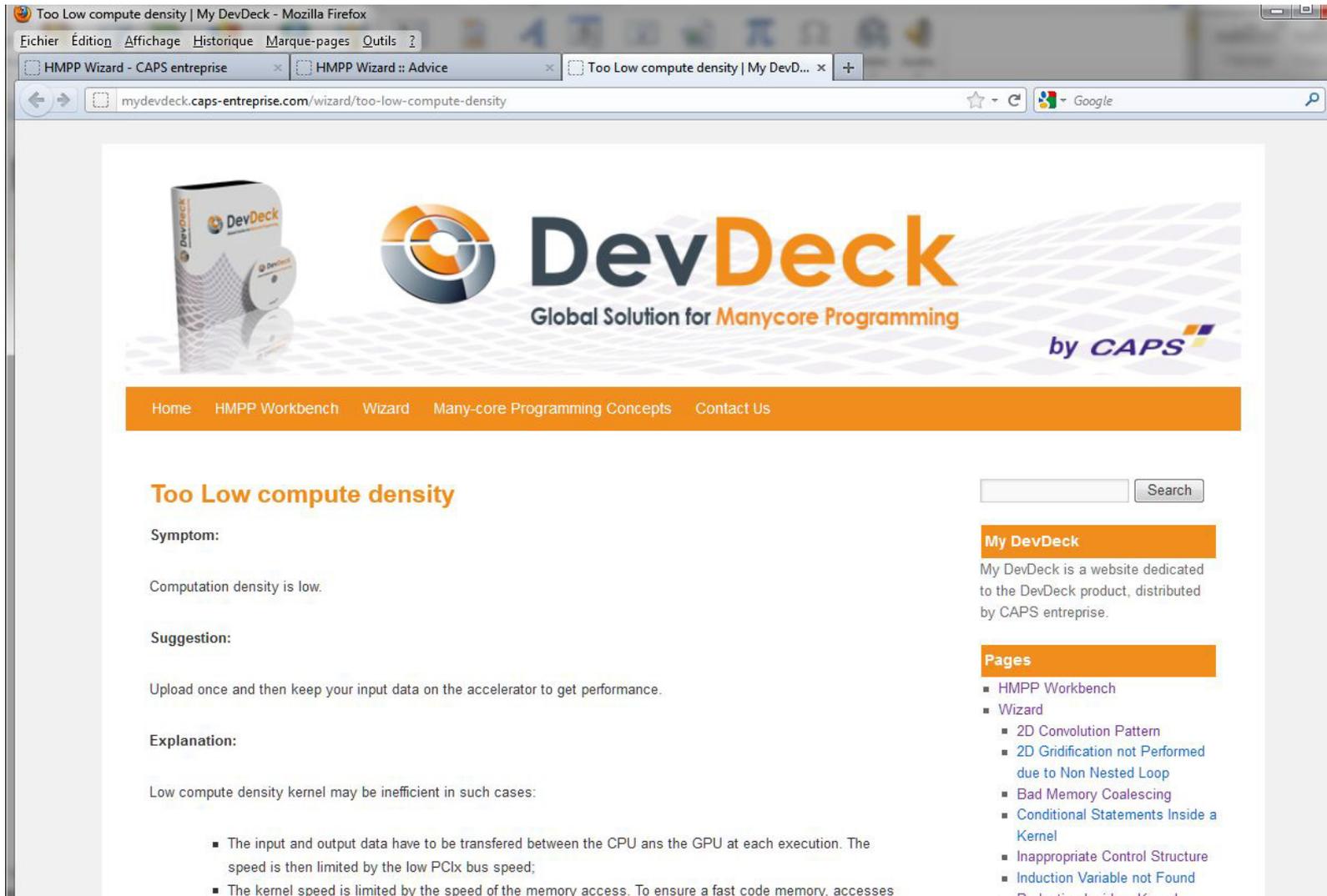
Performance Details

Kernel #1

Performance Metrics	value
▣ Synthetic metrics	
Grid description	grid 11x84=924 blocks, thread block size of 32x4x1, 118272 threads
Average gpu execution time	30073.3 us
Branch divergence / Branch Ratio	0.000987427
Computation density	23.9627 instructions/us (mul. by instr/cycle and div. by frequency to get the throughput)
▣ Detailed Metrics	
gputime	30073.3 us
cputime	20072.7 us
occupancy	1

**Tune your Codelet
for Your Hardware**

- Online Web resources completed by CAPS



Too Low compute density | My DevDeck - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

HMPP Wizard - CAPS entreprise x HMPP Wizard :: Advice x Too Low compute density | My DevD... x +

mydevdeck.caps-entreprise.com/wizard/too-low-compute-density

☆ Google



Home HMPP Workbench Wizard Many-core Programming Concepts Contact Us

Too Low compute density

Symptom:

Computation density is low.

Suggestion:

Upload once and then keep your input data on the accelerator to get performance.

Explanation:

Low compute density kernel may be inefficient in such cases:

- The input and output data have to be transferred between the CPU and the GPU at each execution. The speed is then limited by the low PCIx bus speed;
- The kernel speed is limited by the speed of the memory access. To ensure a fast code memory access

Search

My DevDeck

My DevDeck is a website dedicated to the DevDeck product, distributed by CAPS entreprise.

Pages

- HMPP Workbench
- Wizard
 - 2D Convolution Pattern
 - 2D Gridification not Performed due to Non Nested Loop
 - Bad Memory Coalescing
 - Conditional Statements Inside a Kernel
 - Inappropriate Control Structure
 - Induction Variable not Found
 - Reduction Inside a Kernel

- **Abstract the programming of manycore architectures**
 - A rich set of programming and tuning directives
 - Distribute computations to exploit CPU and GPU cores in a node
 - Mix CPU and GPU libraries in same binary
 - Incrementally develop and port applications

- **An open source-to-source compiler**
 - Work with standard compilers and hardware vendor tools
 - Ease maintenance by avoiding different languages
 - Preserve legacy code

Accelerator Programming Model

Parallelization



Directive-based programming

GPGPU **Manycore programming**

Hybrid Manycore Programming

HPC community

Petaflops

Parallel computing

HPC open standard

Multicore programming

Exaflops

NVIDIA CUDA

Code speedup

Hardware accelerators programming

High Performance Computing

Parallel programming interface

Massively parallel

OpenCL

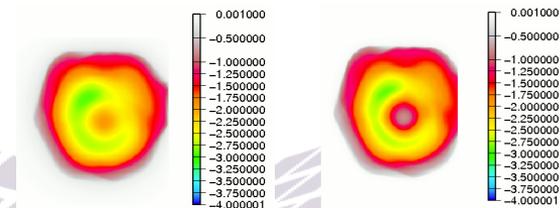
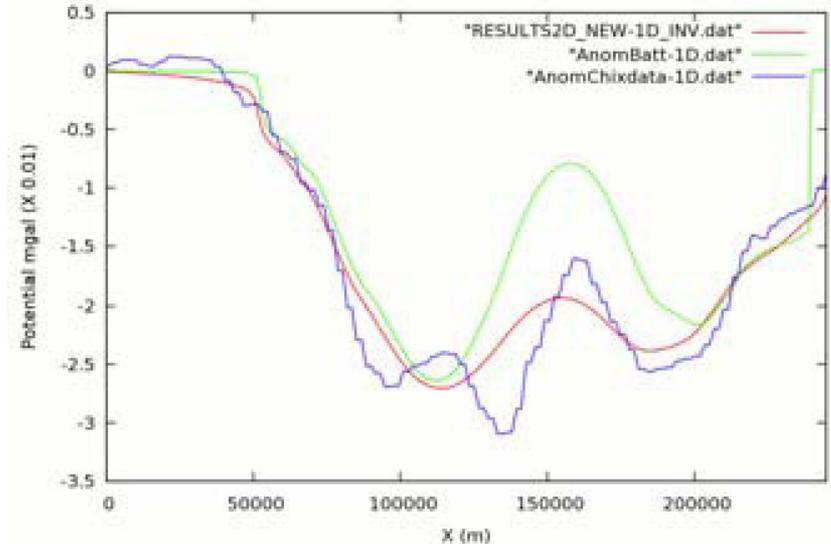


<http://www.caps-entreprise.com>
<http://twitter.com/CAPSentreprise>

jvasnier@caps-entreprise.com
jcv@ornl.gov

3D Poisson equation conjugate gradient

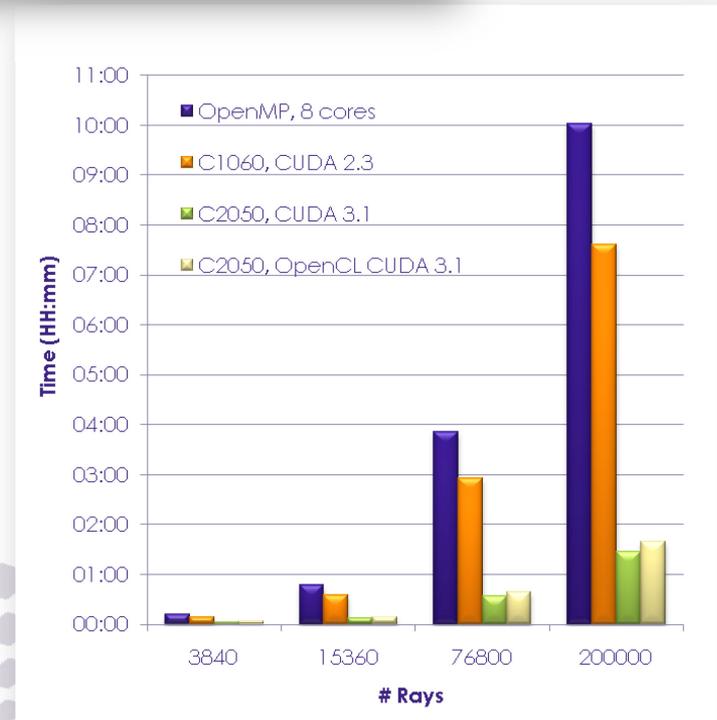
- Resource spent
 - 2 man-month
- Size
 - 2kLoC of F90 (DP)
- CPU improvement
 - **X1,73**
- GPU C1060 improvement
 - x 5,15 over serial code on Nehalem
- Main porting operation
 - highly optimizing kernels



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

Monte Carlo simulation for thermal radiation

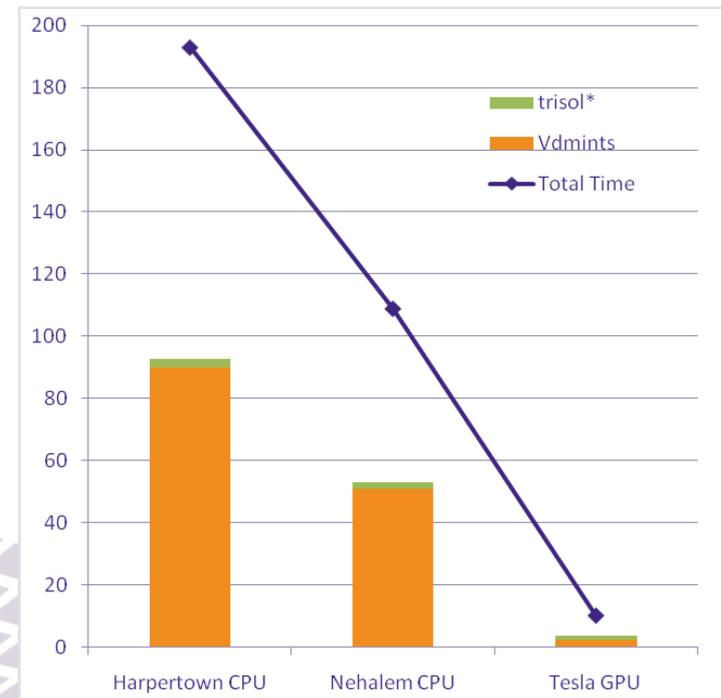
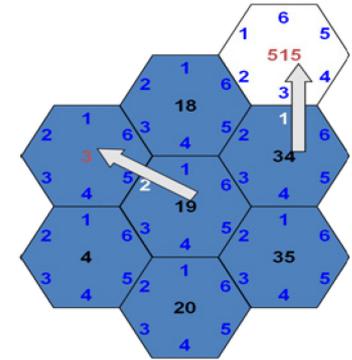
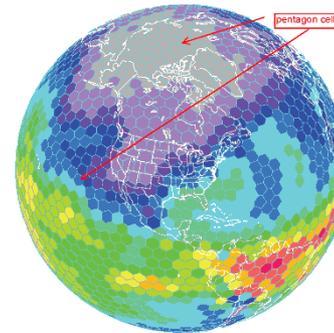
- Resource spent
 - 1,5 man-month
- Size
 - ~1kLoC of C d (DP)
- GPU C2050 improvement
 - x6 over 8 Nehalem cores
- Full hybrid version
 - **x23 with 8 nodes compare to the 8 core machine**
- Main porting operation
 - adding a few HMPP directives
- « *The use of HMPP™ and the expertise of CAPS allowed us to cut by four the time needed to migrate our existing code to CUDA and OpenCL* »
 - *Marc DAUMAS and Patrick VILAMAJO From PROMES Laboratory*



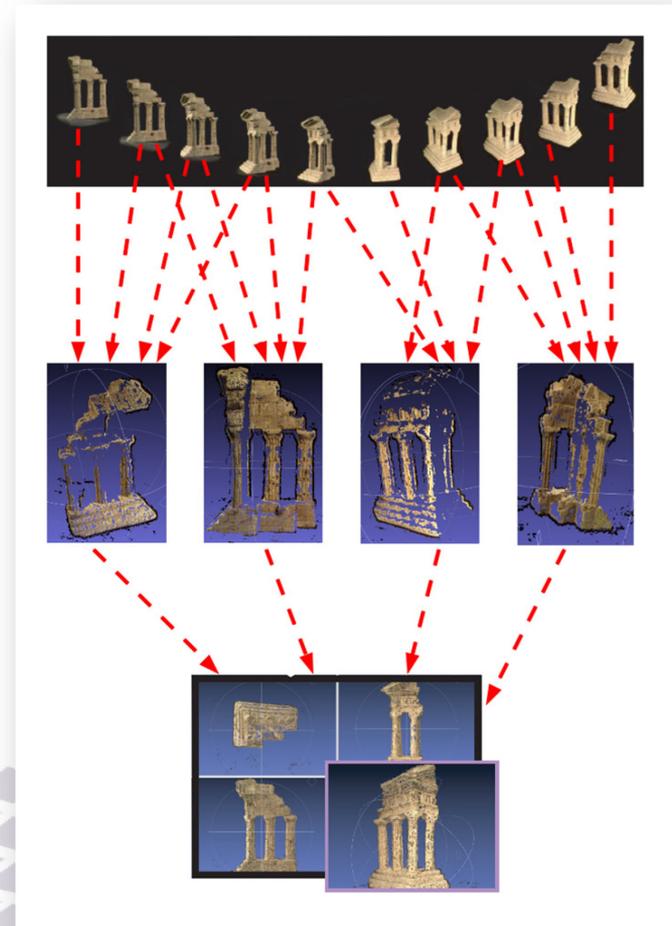
Weather Forecasting

A global cloud resolving model

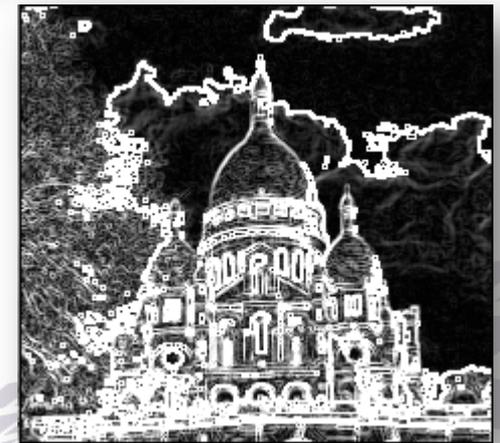
- **Resource spent**
 - 1 man-month (part of the code already ported)
- **GPU C1060 improvement**
 - 11x over serial code on Nehalem
- **Main porting operation**
 - reduction of CPU-GPU transfers
- **Main difficulty**
 - GPU memory size is the limiting factor
- *“We regard this initial HMPP result very favorably because we have not yet finished optimizing with HMPP directives and the effort required has been small compared to our hand-tuned CUDA development.”*
 - Thomas B. Henderson, Mark Govett, Jacques Middlecoff - NOAA



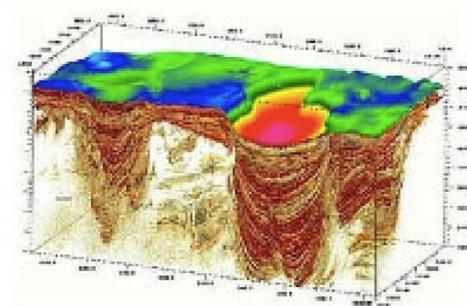
MultiView Stereo



Edge detection algorithm



GPU-accelerated seismic depth imaging



GPU accelerated Rack

4.4 CPU Racks



performance

