CAPS

Incremental Migration of C and Fortran Applications to GPGPU using HMPP

Peppher 2011

Introduction

- Many applications can benefit from GPU computing
  - Linear Algebra, signal processing
  - Bio informatics, molecular dynamics
  - Magnetic resonance imaging, tomography
  - Reverse time migration, electrostatic
  - …
- Porting legacy codes to GPU computing is a major challenge
  - Can be very expensive
  - Require to minimize porting risks
  - Should be based on future-proof approach
  - Implies application and performance programmers to cooperate
- A good methodology is paramount to reduce porting cost
  - HMPP provides an efficient solution

What is HMPP? (Hybrid Manycore Parallel Programming)

CAPS

- A directive based multi-language programming environment
  - Help keeping software independent from hardware targets
  - Provide an incremental tool to exploit GPU in legacy applications
  - Avoid exit cost, can be future-proof solution
- HMPP provides
  - Code generators from C and Fortran to GPU (CUDA or OpenCL)
  - A compiler driver that handles all low level details of GPU compilers
  - A runtime to allocate and manage GPU resources
- Source to source compiler
  - CPU code does not require compiler change
  - Complement existing parallel APIs (OpenMP or MPI)

HMPP Main Design Considerations

CAPS

- Focus on the main bottleneck
  - Communication between GPUs and CPUs
- **Allow incremental development**
  - Up to full access to the hardware features
- Work with other parallel APIs (e.g. OpenMP, MPI)
  - Orchestrate CPU and GPU computations
- Consider multiple languages
  - Avoid asking users to learn a new language
- Consider resource management
  - Generate robust software
- Exploit vendor tools/compilers
  - Do not replace, complement

How Does HMPP Differ from CUDA or OpenCL?

- HMPP 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 hmppcg parallel
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);
```

HMPP Codelets and Regions

- A codelet is a pure function that can be remotely executed on a GPU
- Regions are a short cut for writing codelets

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

```
#pragma hmpp myreg region, …
{
  for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```

Codelet Target Clause

- Target clause specifies what GPU code to generate
  - *GPU* can be CUDA or OpenCL
- Choice of the implementation at runtime can be different!
  - The runtime select among the available hardware and code

```
#pragma hmpp myLabel codelet, target=[GPU], args[C].io=out
void myFunc( int n, int A[n], int B[n], int C[n]){
   ...
}
```

```
#pragma hmpp myLabel codelet, target=CUDA
```

→ NVIDIA only GPU

```
#pragma hmpp myLabel codelet, target=OpenCL
```

→ NVIDIA & AMD GPU, AMD CPU

HMPP Codelets Arguments

- The arguments of codelet are also allocated in the GPU device memory
  - Must exist on both sides to allow backup execution
  - No hardware mechanism to ensure consistencies
  - Size must be known to perform the data transfers
- Are defined by the `io` clause (in Fortran use intent instead)
  - **`in`** (default) : read only in the codelet
  - **`out`**: completely defined, no read before a write
  - **`inout`**: read and written
- Using inappropriate **`inout`** generates extra PCI bus traffic

```
#pragma hmpp myLabel codelet, args[B].io=out, args[C].io=inout
void myFunc( int n, int A[n], int B[n], int C[n]){
    for( int i=0 ; i<n ; ++i){
        B[i] = A[i] * A[i];
        C[i] = C[i] * A[i];
    }
}
```

Running a Codelet or Section on a GPU - 1

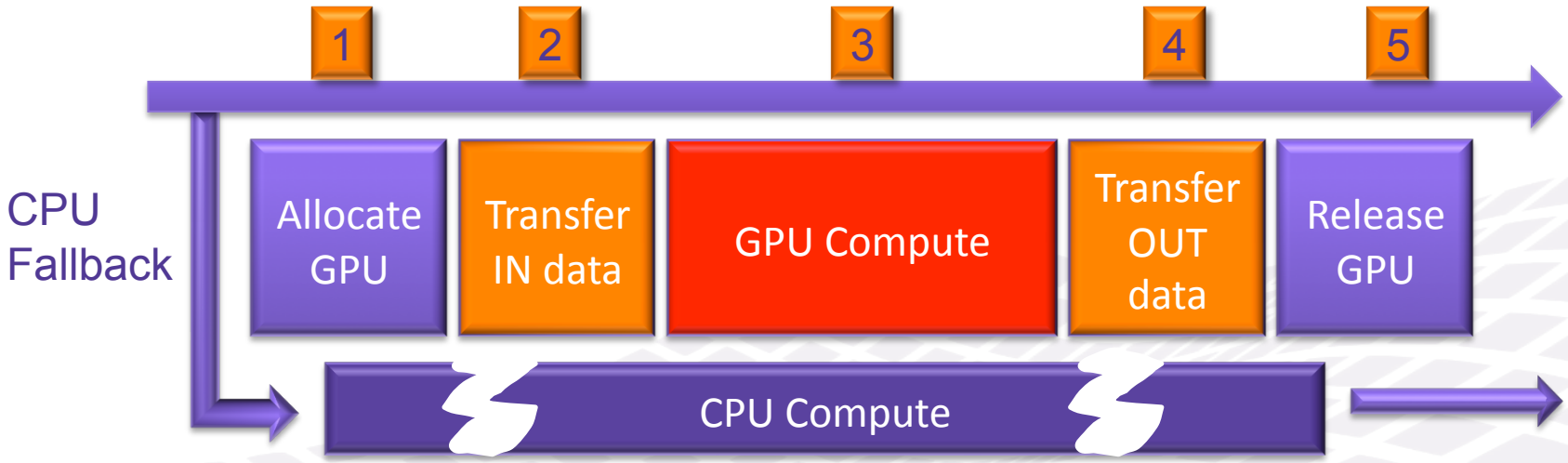
CAPS

- The callsite directive specifies the use of a codelet at a given point in your application.
- `callsite` directive performs a Remote Procedure Call onto the GPU

```
#pragma hmpp call1 codelet, target=CUDA
#pragma hmpp call2 codelet, target=OpenCL
void myFunc(int n, int A[n], int B[n]){
    int i;
    for (i=0 ; i<n ; ++i)
        B[i] = A[i] + 1;
}

void main(void)
{
    int X[10000], Y[10000], Z[10000];
    …
    #pragma hmpp call1 callsite,  …
    myFunc(10000, X, Y);
    ...
    #pragma hmpp call2 callsite,  …
    myFunc(1000, Y, Z);
    …
}
```

Running a Codelet or Section on a GPU - 2

- By default, a CALLSITE directive implements the whole Remote Procedure Call (RPC) sequence
- An RPC sequence consists in 5 steps:
  - (1) Allocate the GPU and the memory
  - (2) Transfer the input data: CPU => GPU
  - (3) Compute
  - (4) Transfer the output data: GPU=> CPU
  - (5) Release the GPU and the memory



Tuning Hybrid Codes

CAPS

- Tuning hybrid code consists in
  - Reducing penalty when allocating and releasing GPUs
  - Reducing data transfer time
  - Optimizing performance of the GPU kernels
  - Using CPU cores in parallel with the GPU
- HMPP provides a set of directives to address these optimizations
- The objective is to get efficient CPU **and** GPU computations

Reducing Data Transfers between CPUs and GPUs

- Hybrid code performance is very sensitive to the amount of CPU-GPU data transfers
  - PCIx bus is a serious bottleneck (< 10 GBs vs 150 GBs)

- Various techniques
  - Reduce data transfer occurrences
  - Share data on the GPU between codelets
  - Map codelet arguments to the same GPU space
  - Perform partial data transfers

- Warning: dealing with two address spaces may introduce inconsistencies

Tuning GPU Kernels

CAPS

- GPU kernel tuning set-up parallel loop suitable for GPU architectures

- Multiple issues to address
  - Memory accesses
  - Thread grid tuning
  - Register usage tuning
  - Shared memory usage
  - Removing control flow divergence

- In many cases, CPU code structure conflicts with GPU efficient code structure

Methodology to Port Applications

CAPS

- Prerequisite
  - Understand your performance goal
    - Memory bandwidth needs are a good potential performance indicator
  - Know your hotspots
    - Beware of Amdahl's law
  - Ensure you know how to validate the output of your application
    - Rounding may differs on GPUs
  - Determine if you goal can be achieved
    - How many CPUs and GPUs are necessary?
    - Is there similar existing codes for GPUs (in CUDA, OpenCL or HMPP)?
- Define an incremental approach
  - Ensure to check the results at each step
- Two phase approach
  - Phase 1: Application programmers validate the computed results
  - Phase 2: Performance programmers focus on GPU code tuning and data transfer reduction

Methodology to Port Applications

CAPS

Hotspots

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

Hours to Days

Parallelization

- **Optimize** CPU code
- **Exhibit** application SIMT parallelism
- Push application hotspot **on GPU**
- **Validate CPU-GPU execution**

Days to Weeks

Define your parallel project

Port your application on GPU

GPGPU operational application with known potential

Phase 1

Phase 2

Tuning

- **Exploit** CPU and GPU
- Reduce CPU-GPU **data transfers**
- **Optimize** GPU kernel execution
- Provide **feedback to application programmers** for improving algorithm data structures/…
- Consider **multiple GPUs**

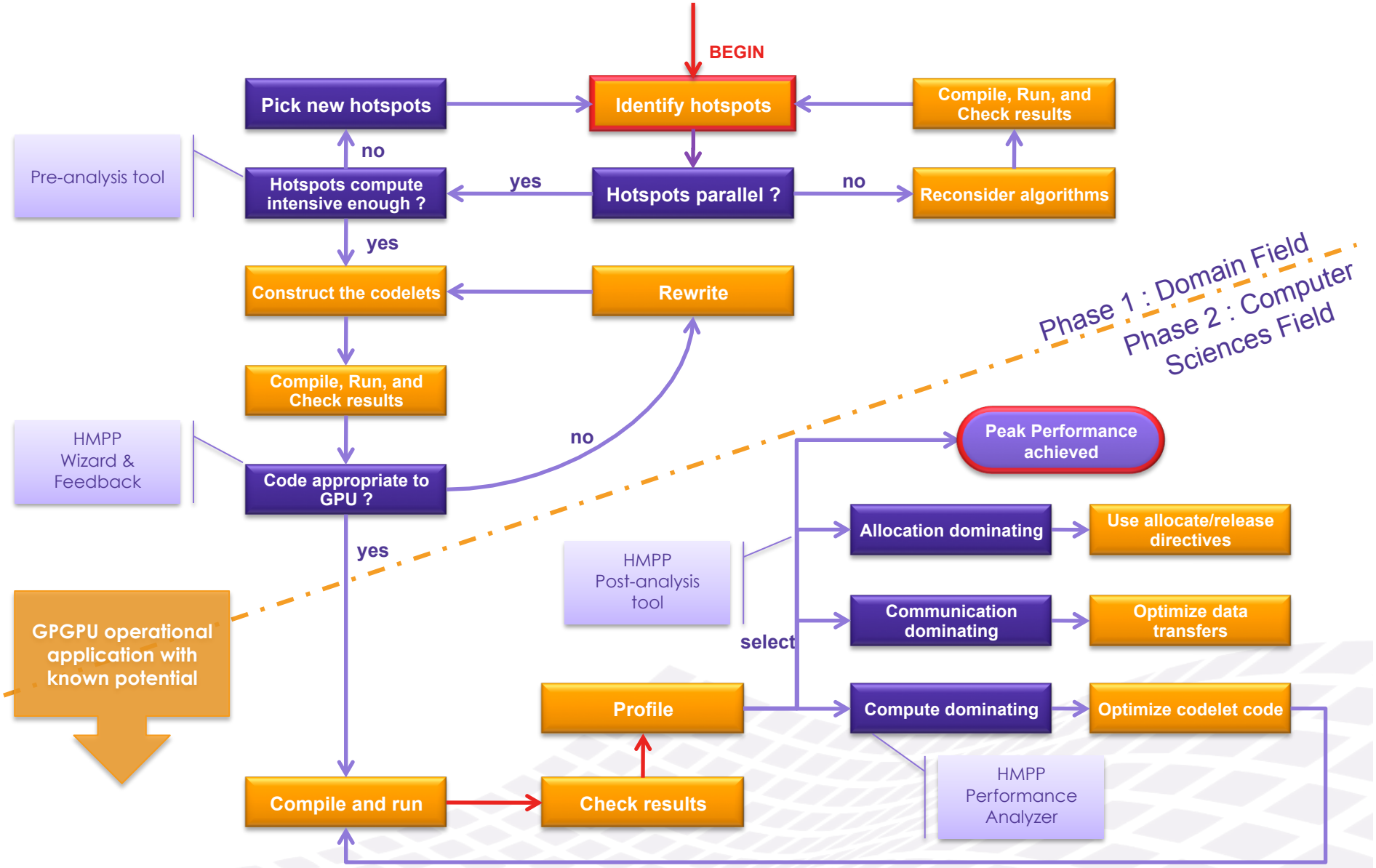
Weeks to Months

Optimize your GPGPU application

A corporate project

- Purchasing Department
- Scientists
- IT Department

Methodology Overview

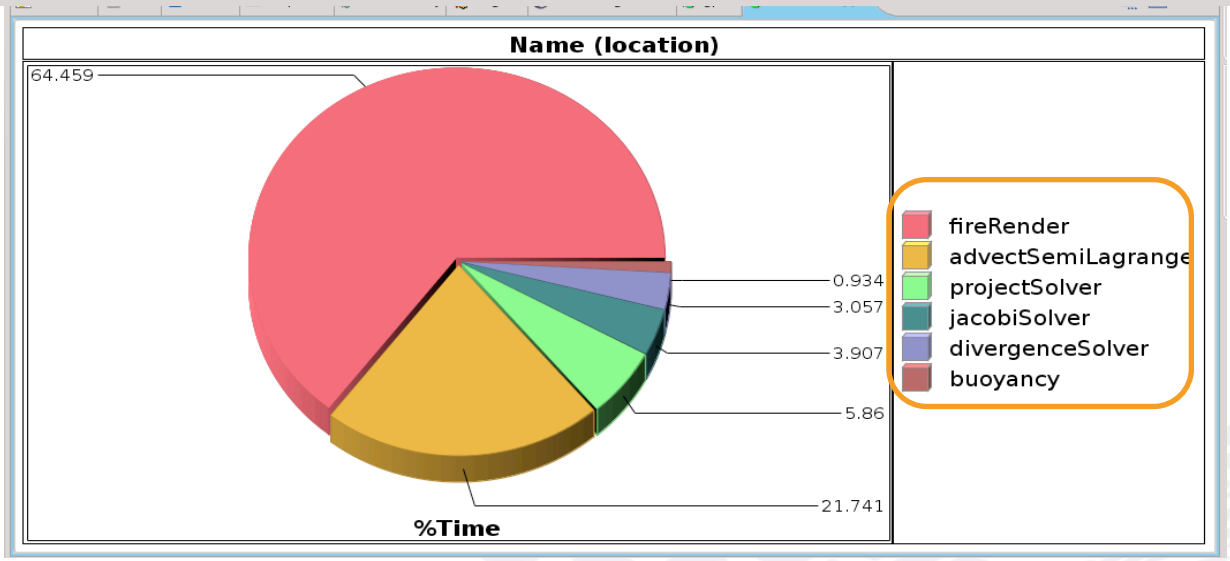


Focus on Hotspots

Problems | Tasks | Console | Properties | Call Hierarchy | Valgrind | HMPP Pragma View | gprof

gmon file: /home/demo/DemoWizard/EclipseWP-v1.5/DynFluid/lastRun/gmon.out
program file: /home/demo/DemoWizard/EclipseWP-v1.5/DynFluid/lastRun/DynFluid
4 bytes per bucket, each sample counts as 10.0ms

Name (location)	Samples	Calls	Time/Call	%Time
Summary	2355			100.0%
fireRender	1518	251	60.478ms	64.46%
advectSemiLagrange	512	251	20.398ms	21.74%
projectSolver	138	251	5.498ms	5.86%
jacobiSolver	92	251	3.665ms	3.91%
divergenceSolver	72	251	2.868ms	3.06%
buoyancy	22	251	876.494us	0.93%
init	1	1	10.0ms	0.04%
DrawGLScene	0	252	0ns	0.0%



Profile your CPU application

Build a coherent kernel set

Build Your GPU Computation with HMPP Directives (1)

Construct your GPU group of codelet

*fireRender.h | fireRender.c | filter.h | filter.c | solvers.h | solvers.c | fluidMotion.c

```
/*****************************************************************************
 *       solvers.c                                                           *
 *****************************************************************************/

#ifndef FIRERENDER_H
#define FIRERENDER_H

#include "solvers.h"


#pragma hmpp <smallDensRad> fireRender codelet, args[res,velx,vely,velz,density,screenRes,posCam,distCamScreen,noise].io=i
void fireRender(struct _coord res, struct _screenRes screenRes,
                float velx[res.m_z][res.m_y][res.m_x], float vely[res.m_z][res.m_y][res.m_x], float velz[res.m_z][res.m_y]
                float density[res.m_z][res.m_y][res.m_x],
                int distCamScreen, int posCam[3],
                float result[screenRes.m_y][screenRes.m_x][4], float noise[screenRes.m_y][screenRes.m_x]);

/* FIRERENDER_H */
#endif
```

Build Your GPU Computation with HMPP Directives (2)

CAPS

*fireRender.h | fireRender.c | filter.h | filter.c | solvers.h | solvers.c | fluidMotion.c

```
24 {
25   int N = SZ(res);
26
27   tmpFields     = (float*)malloc(N * 4 * sizeof(float));
28   tmpPressure   = (float*)malloc(N * sizeof(float));
29   tmpDivergence = (float*)malloc(N * sizeof(float));
30
31 #pragma hmpp <smallDensRad> allocate
32 }
33
34 void solvers(float dto, float dx, struct _coord res, float* velx, float* vely, float* velz, float * pressure, float * dens
35 {
36
37   buoyancy(res,velx,vely,velz,density, gravity);                    //<====   ||   ====>
38
39   divergenceSolver(dx,res,velx,vely,velz,tmpDivergence,obstacles);  //<====   ||   ====>
40
41   jacobiSolver(res,tmpDivergence,pressure,obstacles, tmpPressure);  //<====   ||   ====>
42
43   projectSolver(dx,res,velx,vely,velz,pressure,obstacles);          //<====   ||   ====>
44
45 #pragma hmpp <smallDensRad> advectSemiLagrange callsite
46   advectSemiLagrange(dto, res, density, velx, vely, velz, tmpFields);   //<====
47
48 }
49
50 void fireRenderTo2D(struct _coord res, float* velx, float* vely, float* velz, float* density, int* posCam, int distCamScre
51 {
52 #pragma hmpp <smallDensRad> fireRender callsite
53   fireRender(res, screenRes, velx, vely, velz, density, distCamScreen, posCam, result, noise);
54 }
55
56 void release(float* velx, float* vely, float* velz,float * pressure,float * density,float* obstacles) {
57 #pragma hmpp <smallDensRad> release
58   free(tmpDivergence);
59   free(tmpPressure);
60   free(tmpFields);
61 }
62
```

… and use Codelets in the application

Tune Your Kernels for GPUs with CAPS HMPP Wizard (1/2)

HMPP Wizard - [/home/demo/DemoWizard/EclipseWP-v1.5/DynFluid/src/solvers/filter.c]

File Directive Help

jacobiSolver

Codelet

```
#pragma hmpp <densRad> jacobiSolver codelet
void jacobiSolver(struct _coord res, float divergence[res.m_z][res.m_
{
  int  x, y, z;
  int  xMax = res.m_x, yMax = res.m_y, zMax = res.m_z;
  for (z = 1 ; z < zMax - 1 ; z++)
    {
      for (y = 1 ; y < yMax - 1 ; y++)
        {
          for (x = 1 ; x < xMax - 1 ; x++)
            {
              {
                float  div = divergence[z][y][x];
                float  dampingPres = 0.9f * pressure[z][y][x];
                float  obstL = obstacles[z][y][x - 1];
                float  obstR = obstacles[z][y][x + 1];
                float  pL = (1.f - obstL) * dampingPres + obstL * pre
                float  pR = (1.f - obstR) * dampingPres + obstR * pre
                float  obstB = obstacles[z][y - 1][x];
                float  obstF = obstacles[z][y + 1][x];
                float  pB = (1.f - obstB) * dampingPres + obstB * pre
                float  pF = (1.f - obstF) * dampingPres + obstF * pre
```

Advices

```
**** HMPP Wizard ****
  ----------
  /home/demo/DemoWizard/EclipseWP-v1.5/DynFluid/src/solver..
    Diagnosis
    Advice                                          The computation using y is globally not well coalesced
                                                    Insert a loop interchange pragma to the loop nest
    Loop Statistics
      Number of subscript access over X             0
      Number of subscript access over Y             0
      Number of subscript access over Z             16
      Number of subscript access constant over the gridification 0
                                                    #pragma hmppcg permute(z,x,y)
                                                    for (z = 1 ; z < zMax - 1 ; z++)
                                                     {
                                                      for (y = 1 ; y < yMax - 1 ; y++)
                                                       {
                                                        for (x = 1 ; x < xMax - 1 ; x++)
                                                         {
                                                          {
                                                           float  div = divergence[z][y][x];
                                                           float  dampingPres = 0.9f * pressure[z][y][x];
                                                           float  obstL = obstacles[z][y][x - 1];
                                                           float  obstR = obstacles[z][y][x + 1];
                                                           float  pL = (1.f - obstL) * dampingPres + obstL * pressure[z][y][x - 1];
    Result                                              float  pR = (1.f - obstR) * dampingPres + obstR * pressure[z][y][x + 1];
```

Analysis Log

```
**** Analysis Logs ****
  Codelet Validation                                Success
  Codelet's kernel Analysis                         Success
    Analysis                                        Loop analysis details
      Number of Kernels                             2
        Kernel number                               1
          3D Loop gridification
      Kernel number                                 2
        3D Loop gridification
```

Analyze your memory access pattern

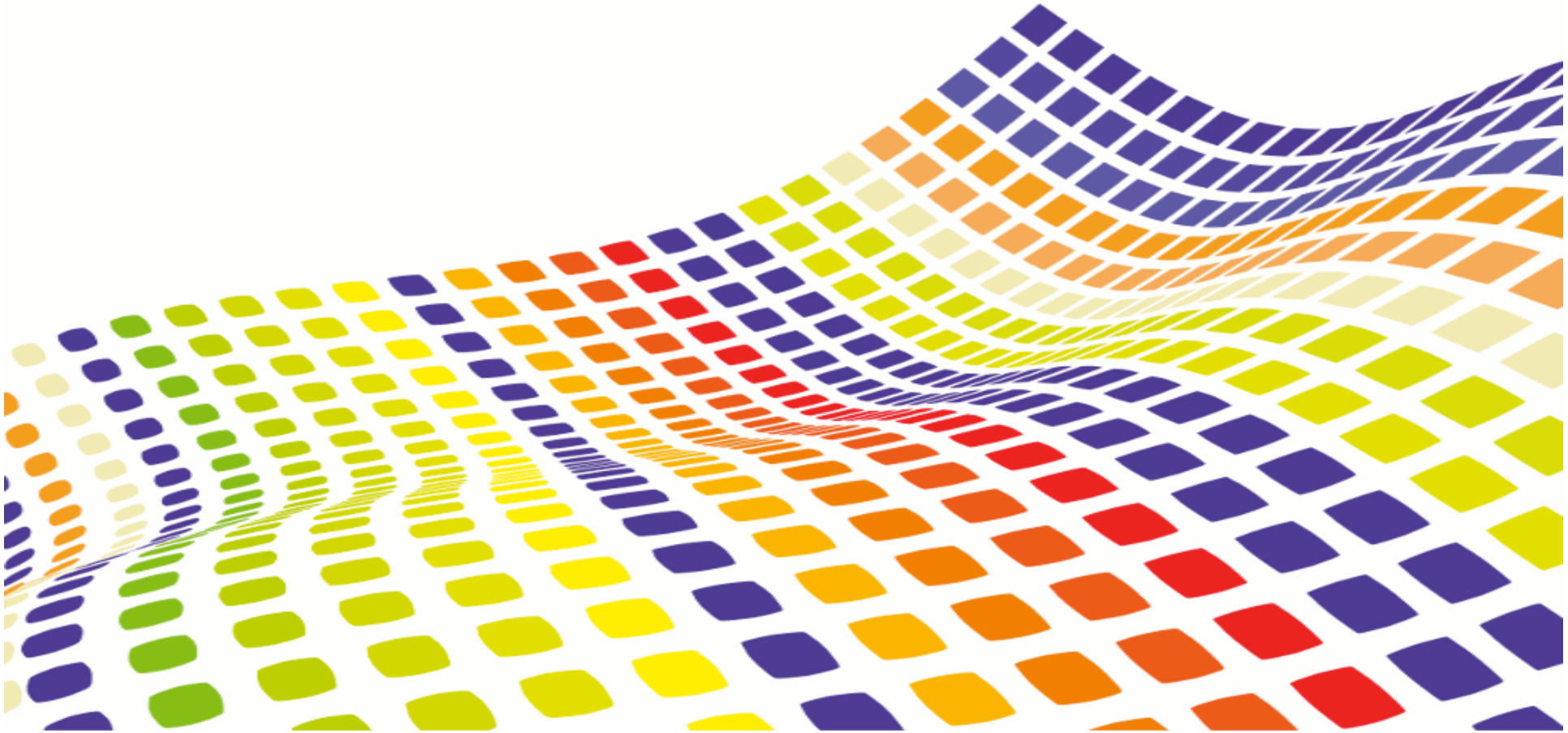
Use HMPPCG Directives and make your kernel GPU friendly

Tune Your Kernels for GPUs with CAPS HMPP Wizard (2/2)

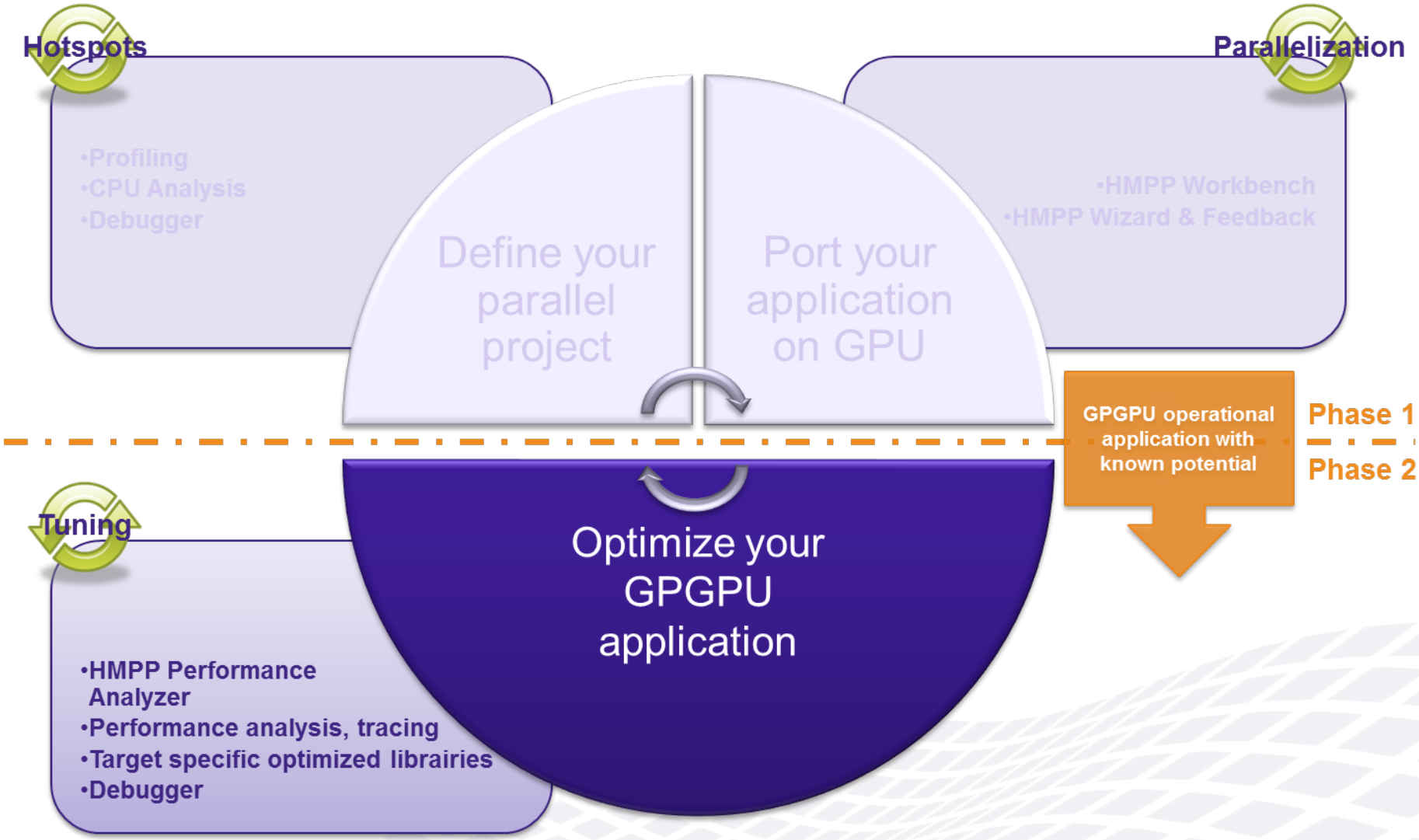
CAPS

```
**** HMPP Wizard ****
  ----------------------
    /home/demo/DemoWizard/EclipseWP-v1.5/DynFluid/src/solver...
      Diagnosis                                          The computation using y is globally not well coalesced
      Advice                                             Insert a loop interchange pragma to the loop nest
      Loop Statistics
        Number of subscript access over X                0
        Number of subscript access over Y                0
        Number of subscript access over Z                16
        Number of subscript access constant over the gridification 0

                                                         #pragma hmppcg permute(z,x,y)
                                                         for (z = 1 ; z < zMax - 1 ; z++)
                                                          {
                                                           for (y = 1 ; y < yMax - 1 ; y++)
                                                            {
                                                             for (x = 1 ; x < xMax - 1 ; x++)
                                                              {
                                                               {
                                                                float  div = divergence[z][y][x];
                                                                float  dampingPres = 0.9f * pressure[z][y][x];
                                                                float  obstL = obstacles[z][y][x - 1];
                                                                float  obstR = obstacles[z][y][x + 1];
                                                                float  pL = (1.f - obstL) * dampingPres + obstL * pressure[z][y][x - 1];
      Result                                            float  pR = (1.f - obstR) * dampingPres + obstR * pressure[z][y][x + 1];
```

CAPS Tools to Port Your Applications – Phase 2



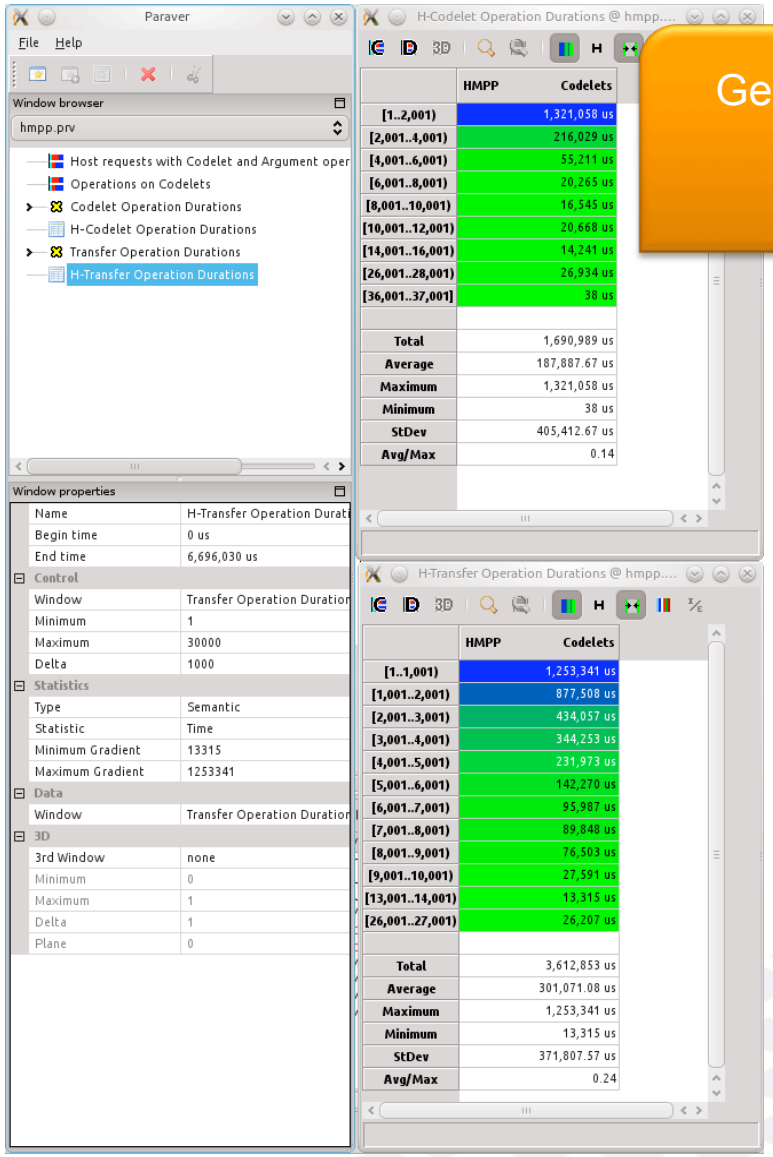
Optimizing Tools

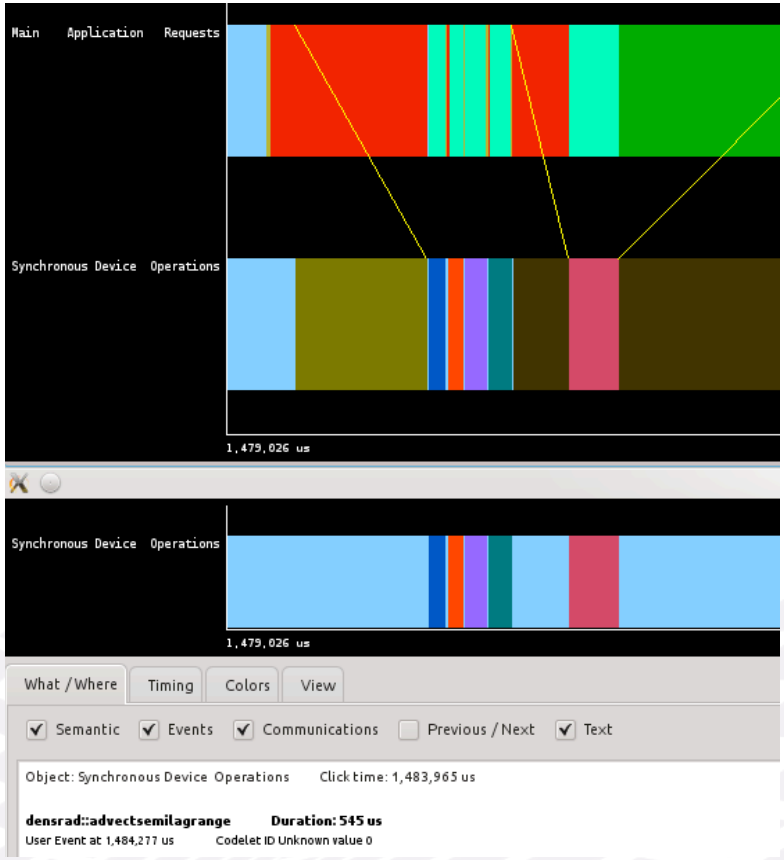


Analyze the GPU Code Porting Efficiency

Get a precise view of HMPP element behavior

Get statistics on GPU operations





Tune the GPU Execution Integration in Your Application with HMPP Directives

fireRender.c | filter.h | fluidMotion.c | solvers.c | filter.c

```
  tmpFields     = (float*)malloc(N * 4 * sizeof(float));
  tmpPressure   = (float*)malloc(N * sizeof(float));
  tmpDivergence = (float*)malloc(N * sizeof(float));

#pragma hmpp <densRad> allocate

/* Constant Values. */
#pragma hmpp <densRad> advancedload, args[buoyancy::res]
#pragma hmpp <densRad> advancedload, args[divergenceSolver::dx, divergenceSolver::obstacles]
#pragma hmpp <densRad> advancedload, args[advectSemiLagrange::dto]

#pragma hmpp <densRad> advancedload, args[fireRender::noise]
#pragma hmpp <densRad> advancedload, args[fireRender::distCamScreen,fireRender::screenRes]

/* Prefetch. */
#pragma hmpp <densRad> advancedload, args[buoyancy::gravity]
#pragma hmpp <densRad> advancedload, args[buoyancy::velx,buoyancy::vely,buoyancy::velz]
#pragma hmpp <densRad> advancedload, args[buoyancy::density]
#pragma hmpp <densRad> advancedload, args[jacobiSolver::pressure]
            #pragma hmpp <densRad> advancedload, args[buoyancy::density]

}

void solvers(float dto, float dx, struct _coord res, float* velx, float* vely, float* velz, float * pressure, float * densit
{

#pragma hmpp <densRad> buoyancy callsite, args[velx,vely,velz].noupdate, args[gravity].noupdate
  buoyancy(res,velx,vely,velz,density, gravity);            //<====    ||   ====>

#pragma hmpp <densRad> divergenceSolver callsite, args[divergence].noupdate, args[velx,vely,velz].noupdate
  divergenceSolver(dx,res,velx,vely,velz,tmpDivergence,obstacles);  //<====    ||   ====>

#pragma hmpp <densRad> jacobiSolver callsite, args[tmpPressure].noupdate, args[divergence].noupdate, args[pressure].noupdate
  jacobiSolver(res,tmpDivergence,pressure,obstacles, tmpPressure);  //<====    ||   ====>

#pragma hmpp <densRad> projectSolver callsite, args[velx,vely,velz].noupdate, args[pressure].noupdate
  projectSolver(dx,res,velx,vely,velz,pressure,obstacles);       //<====    ||   ====>

#pragma hmpp <densRad> advectSemiLagrange callsite, args[tmpFields].noupdate, args[velx,vely,velz].noupdate
  advectSemiLagrange(dto, res, density, velx, vely, velz, tmpFields);   //<====

}

void fireRenderTo2D(struct _coord res, float* velx, float* vely, float* velz, float* density, int* posCam, int distCamScreen
{
```

Optimize out transfers from kernel calls

Optimize the GPU allocation and operate data prefetching

Analyze and profile kernel execution on the GPU with HMPP Performance Analyzer

jacobiSolver

Codelet

```
#pragma hmpp <densRad> jacobiSolver codel
void jacobiSolver(struct _coord res, floa
{
  int  x, y, z;
  int  xMax = res.m_x, yMax = res.m_y, zM
  #pragma hmppcg grid blocksize 64x4
  #pragma hmppcg permute z,x,y
    for (z = 1 ; z < zMax - 1 ; z++)
      {
        for (y = 1 ; y < yMax - 1 ; y++)
          {
            for (x = 0 ; x < xMax - 1 ; x
              {
                if (x)
                  {
                    float  div = diverger
                    float  dampingPres =
                    float  obstL = obstac
                    float  obstR = obstac
                    float  obstB = obstac
                    float  obstF = obstac
                    float  obstD = obstac
                    float  obstU = obstac
                    float  pL = (1.f - ob
                    float  pR = (1.f - ob
                    float  pB = (1.f - ob
                    float  pF = (1.f - ob
                    float  pD = (1.f - ob
                    float  pU = (1.f - ob
                    tmpPressure[z][y][x]
                  }
              }
          }
      }
  #pragma hmppcg grid blocksize 256x1
  #pragma hmppcg permute z,x,y
    for (z = 1 ; z < zMax - 1 ; z++)
      {
        for (y = 1 ; y < yMax - 1 ; y++)
          {
            for (x = 1 ; x < xMax - 1 ; x
              {
                pressure[z][y][x] = tmpPr
              }
```

Codelet <densRad> 'jacobiSolver'

Kernel #1 81%

Kernel #2 18%

Advices

**** HMPP PerfAnalyzer ****	
**** Advices ****	
**** Profile ****	
Kernel number	1
Kernel Name:	void __hmpp_codelet__jacobiSolver_loop0_<64u, 4u>(_coord, int, int, int, float*, float*, float*, float*)
Average gpu execution time:	230.031 us
Gridification:	grid 4x16=64 blocks, thread block size of 64x4x1, 16384 threads
Global memory read throughput:	36.11 GB/s/TPC
Global memory write throughput:	8.28 GB/s/TPC
Global memory throughput:	44.39 GB/s/TPC
L1 Global Load Hit Rate:	29.82 %
Detailed metrics	
real name:	_Z35__hmpp_codelet__jacobiSolver_loop0_ILj64ELj4EEv6_coordiiiPfS1_S1_S1_
gputime:	230.031 us
cputime:	308.202 us
occupancy:	0.833
gridsizeX:	4
gridsizeY:	16
threadblocksizeX:	64
threadblocksizeY:	4
threadblocksizeZ:	1
regperthread:	24 regs
local_load:	0 insts/Warp
local_store:	0 insts/Warp
gld_request:	29636.4 insts/Warp
gst_request:	2116.89 insts/Warp
warps_launched:	71.9681 warps/SM
threads_launched:	2302.98 threads/SM
l1_global_load_hit:	11096.1 hits
l1_global_load_miss:	26111.6 misses
active_warps:	5.15123e+06 warp/SM
fb_subp0_read_sectors:	130007 reads
fb_subp1_read_sectors:	129563 reads
fb_subp0_write_sectors:	29760 writes
fb_subp1_write_sectors:	29760 writes

Analysis Log

Warning	[Message DPL0699] /home/der
Profile Analysis	Success
Analysis Report	
Profile Files	
Profile Loaded	cuda_profile_0.csv
Profile Loaded	cuda_profile_1.csv
Profile Loaded	cuda_profile_2.csv

Get precise and specific information about the kernel behavior

Explore and Exploit at best the GPU power from the C source level

Optimize the GPU Kernel Code Generation with HMPPCG Directives

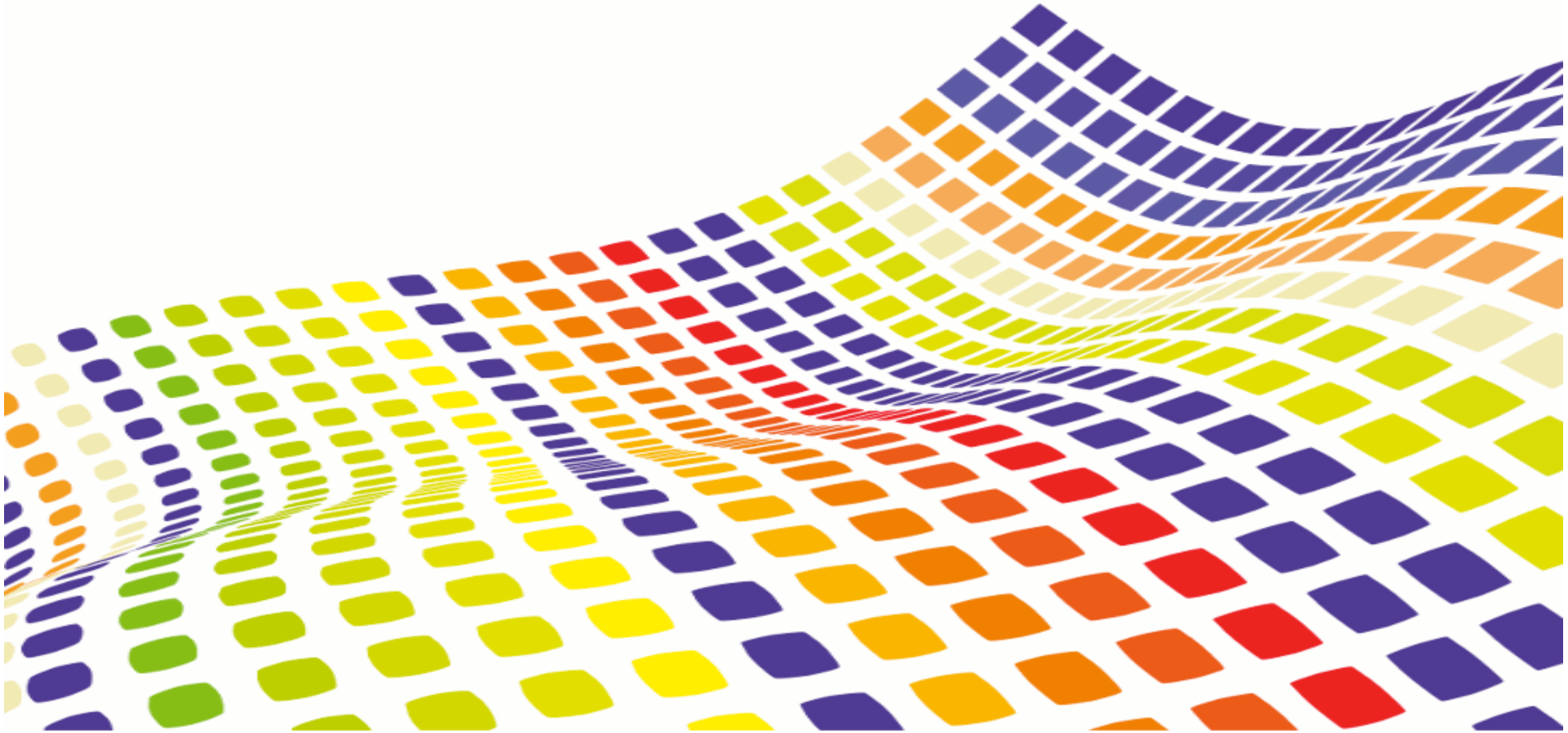
fireRender.c | filter.h | fluidMotion.c | solvers.c | filter.c

```
#ifdef WIZ
#pragma hmpp <densRad> jacobiSolver codelet
#endif
void jacobiSolver(struct _coord res, float divergence[res.m_z][res.m_y][res.m_x], float pressure[res.m_z][res.m_y][res.m_x],
                  float tmpPressure[res.m_z][res.m_y][res.m_x])
{
  int x,y,z;
  int xMax = res.m_x, yMax = res.m_y, zMax = res.m_z;

  //iteration over the 3 dimensions except borders
#pragma hmppcg grid blocksize 256x1
#pragma hmppcg permute z,x,y
  for (z = 1; z < zMax-1; z++) {
    for (y = 1; y < yMax-1; y++) {
      for (x = 1; x < xMax-1; x++) {
        {
            float div = divergence[z][y][x];
            float dampingPres = 0.9f * pressure[z][y][x];

            float obstL = obstacles[z][y][x-1];
            float obstR = obstacles[z][y][x+1];
            float pL = (1.f-obstL) * dampingPres + obstL * pressure[z][y][x-1];
            float pR = (1.f-obstR) * dampingPres + obstR * pressure[z][y][x+1];

            float obstB = obstacles[z][y-1][x];
            float obstF = obstacles[z][y+1][x];
            float pB = (1.f-obstB) * dampingPres + obstB * pressure[z][y-1][x];
            float pF = (1.f-obstF) * dampingPres + obstF * pressure[z][y+1][x];

            float obstD = obstacles[z-1][y][x];
            float obstU = obstacles[z+1][y][x];
            float pD = (1.f-obstD) * dampingPres + obstD * pressure[z-1][y][x];
            float pU = (1.f-obstU) * dampingPres + obstU * pressure[z+1][y][x];

            tmpPressure[z][y][x] = (pL+pR+pB+pF+pD+pU-div)/6.0f;
        }
      }
    }
  }

  // Update Pressure
#pragma hmppcg grid blocksize 256x1
#pragma hmppcg permute z,x,y
  for (z = 1; z < zMax-1; z++) {
    for (y = 1; y < yMax-1; y++) {
      for (x = 1; x < xMax-1; x++) {
```

Control loop transformations using directives

Control the loop distribution over the GPU (grid generation)

Examples of Ported Applications



Examples of Ported Applications – 1

CAPS

- Smoothed particles hydrodynamics
  - Effort: 2 man-month
  - Size: 22kLoC of F90 (SP or DP, MPI)
  - GPU C1060 improvement: x 2 over serial code on Nehalem (x1.1 DP)
  - Main difficulty: kernels limited to 70% of the execution time

The ratio performance over resource is the important information here.

- 3D Poisson equation, conjugate gradient
  - Effort: 2 man-month
  - Size: 2kLoC of F90 (DP)
  - CPU improvement: x 2
  - GPU C1060 improvement: x 5 over serial code on Nehalem
  - Main porting operation: highly optimizing kernels
  - Main difficulty: none

Examples of Ported Applications - 2

CAPS

- Electron propagation - solver
  - Effort: 2 man-month
  - Size: 10 kLoC of F95 (DP, MPI)
  - CPU improvement: x 1.3
  - GPU C1060 improvement: x 1.15 over 4 thread code on Nehalem
  - Main porting operation: solver algorithm modifications
  - Main difficulty: small matrices, many data transfers
- 3D combustion code
  - Effort: 2 man-month
  - Size: x100 kLoC of F90 (DP)
  - GPU C1060 improvement: ~x1 (data transfer limited) over serial code on Nehalem; C2050 x1.3
  - Main difficulty: execution profile shows few hot-spots (70%)
  - Next: code/algo. is being reviewed according to current results

Examples of Ported Applications - 3

CAPS

- Euler equations
  - Effort: <1 man-month
  - Size: ~40kLoC of F90 (DP)
  - CPU improvement: x 3 over the original code
  - GPU C1060 improvement: x 3 over serial code on Nehalem
  - Main porting operation: specializing the code for the main execution configuration
  - Main difficulty: reorganizing computational kernels (CPU dev. legacy)
- Tsunami/flood simulation
  - Effort: 0.5 man-month
  - Size: ~4kLoC (DP, MPI)
  - GPU C1060 improvement: x 1.28 over serial code on Nehalem (kernels speedup x30 and x18)
  - Next: highlight more parallelism, reducing data transfers (high performance potential)

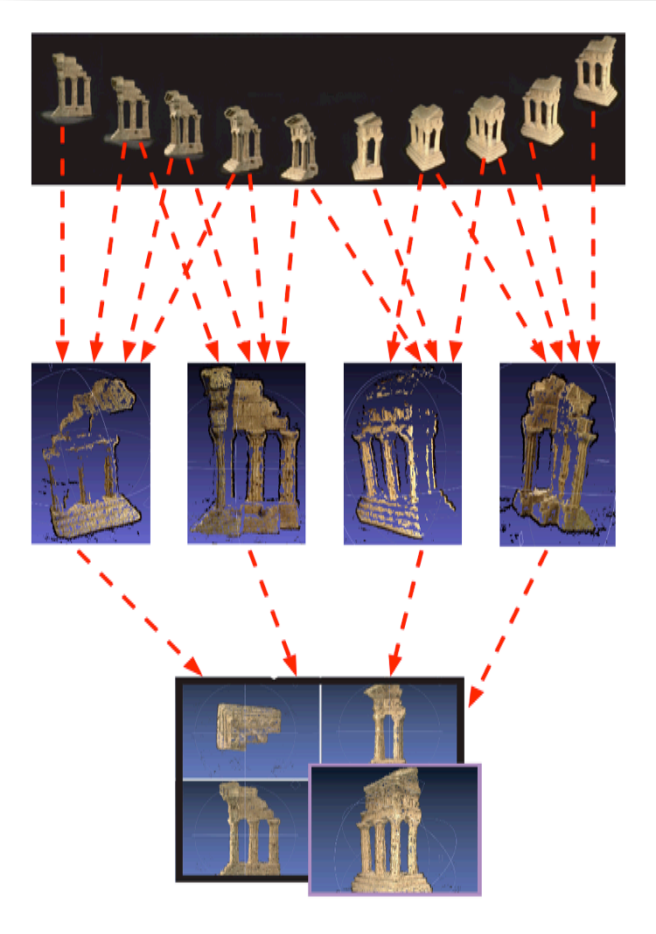
Examples of Ported Applications - 5

- Weather models (GTC 2010 talk, M. Govett, NOAA)
  - Effort: 1 man-month (part of the code already ported)
  - GPU C1060 improvement: 10x over the serial code on Nehalem
  - Main porting operation: reduction of CPU-GPU transfers
  - Main difficulty: GPU memory size is the limiting factor

Computer vision & Medical

MultiView Stereo

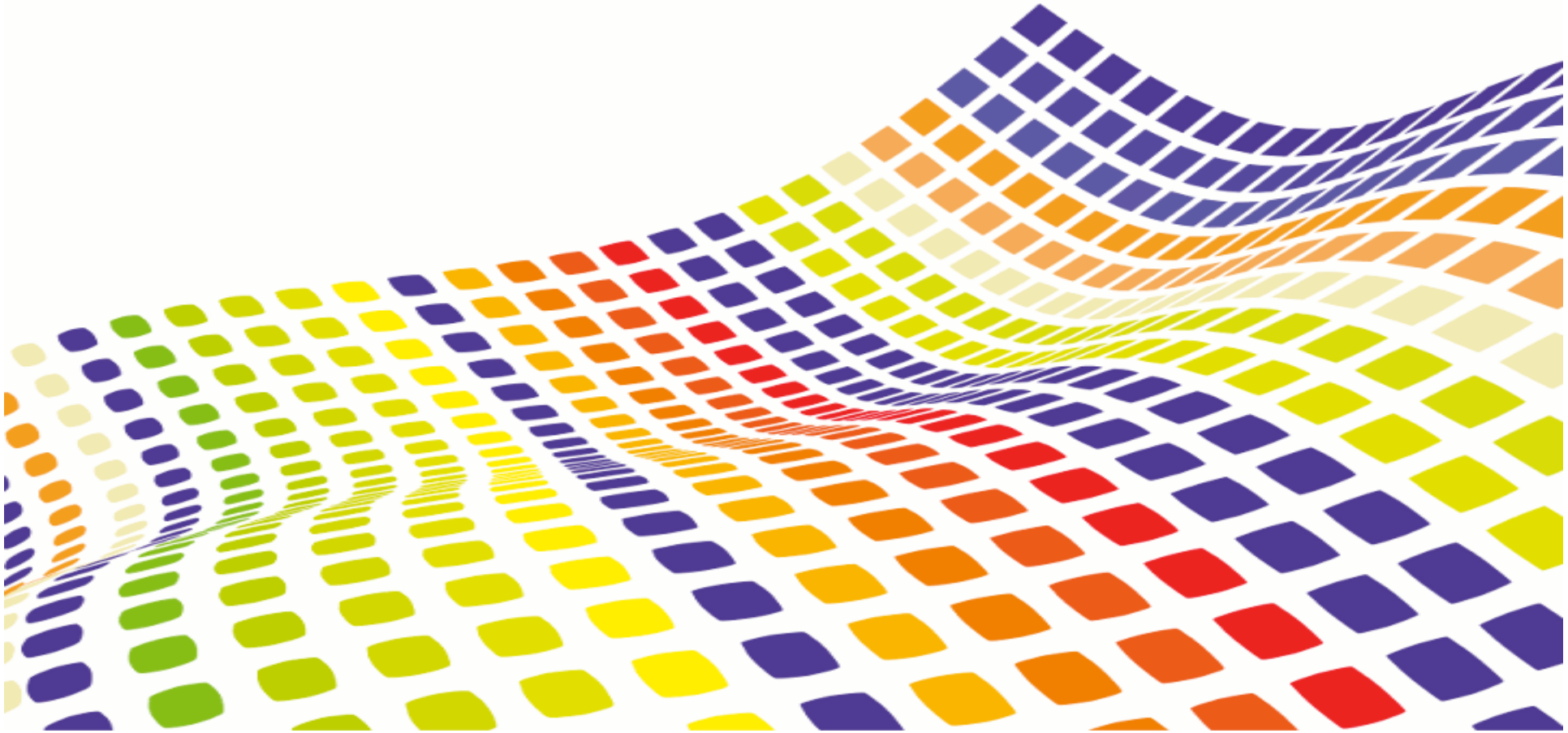
- Resource spent
  - 1 man-month
- Size
  - ~1kLoC of C99 (DP)
- HMPP Basic version (1hour)
  - GPU C2050 improvement
    - X 30
  - Main porting operation
    - Adding 4 directives
- HMPP fine tune version (2 weeks)
  - GPU C2050 improvement
    - X 500
  - Main porting operation
    - Rethinking algorithm



Conclusion

- Heterogeneous architectures are becoming ubiquitous
  - In HPC centers but not only
  - Tremendous opportunities but not always easy to seize
  - CPU and GPU have to be used simultaneously
- Legacy codes still need to be ported
  - An efficient methodology is required
  - A methodology supporting tools is needed and must provide a set of consistent views
  - The legacy style is not helping
  - Highlighted parallelism for GPU is useful for future manycores
- HMPP based programming
  - Helps implementing incremental strategies
  - Is being complemented by a set of tools
  - Engage in an Open Standard path with Pathscale

CAPS



Reducing Data Transfers Occurrences

- Preload data before codelet call
  - Load data as soon as possible

Preload data

```
int main(int argc, char **argv) {

#pragma hmpp sgemm allocate, args[vin1;vin2;vout].size={size,size}
    . . .

#pragma hmpp sgemm advancedload, args[vin1;m;n;k;alpha;beta]


 for( j = 0 ; j < 2 ; j++ ) {
#pragma hmpp sgemm callsite &
#pragma hmpp sgemm  args[m;n;k;alpha;beta;vin1].advancedload=true
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    . . .
 }


 . . .
#pragma hmpp sgemm release
```

Avoid reloading data

Sharing Data Between Codelets with Resident Data

- Share data between codelets of the same group
  - Keep data on the HWA between two codelet calls
  - Avoid useless data transfers

```
#pragma hmpp <process> group, target=CUDA
#pragma hmpp <process> resident
float initValue = 1.5f, offset[9];
…
#pragma hmpp <process> reset1 codelet, args[t].io=out
void reset(float t[M][N]){
  int i,j;
  for (i = 0; i < M; i += 1) {
    for (j = 0; j < N; j += 1) {
      t[i][j] = initValue + offset[(i+j)%9];
    }
  }
}
#pragma hmpp <process> process codelet, args[a].io=inout
void process(real a[M][N], real b[M][N]){
  int i,j;
  for (i = 0; i < M; i += 1) {
    for (j = 0; j < N; j += 1) {
      a[i][j] = cos(a[i][j]) + cos(b[i][j]) - initValue;
    }
  }
}
```