CAPS

Heterogeneous Architecture Programming

June 2011, F. Bodin, CTO

Introduction

- Today's heterogeneous architectures are GPU based and can be efficient in many fields
  - 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
  - Necessary for future manycores anyway
- A good methodology is paramount to reduce porting cost
  - HMPP provides an efficient solution

Overview of the Presentation

- GPU Parallelism Overview
- Programming GPUs
  - Cuda, OpenCL, HMPP Overview
- Code Migration Methodology
  - Handling Legacy Codes
- An Economical Analysis of GPU Computing
  - A CapEx – OpEx Study

A Hybrid/Heterogeneous Compute Node

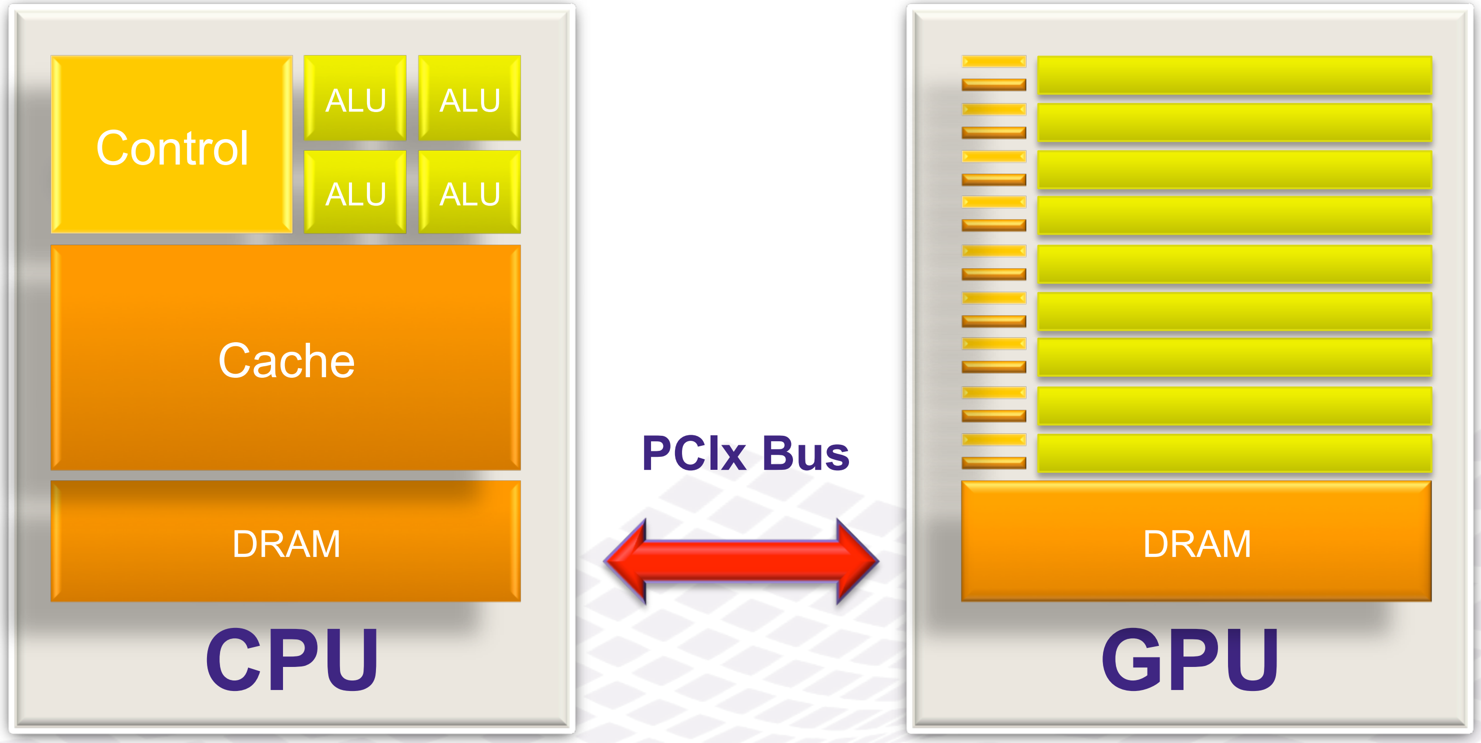
- General purpose cores
  - Share a main memory
  - Core ISA provides fast SIMD instructions
  - Large cache memories

•Streaming engines (e.g. GPU)

Application specific architectures (“*narrow band*”)

Vector/SIMD

Can be extremely fast

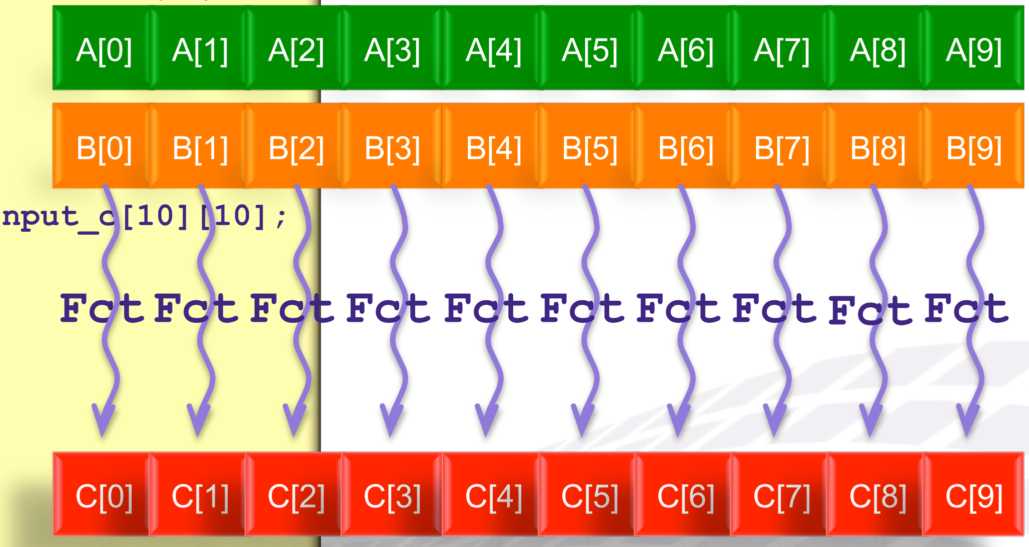


What is Stream Computing?

- A similar computation is performed on a collection of data (*stream*)
  - There is no data dependence between the computation on different stream elements
- Stream programming is well suited to GPU

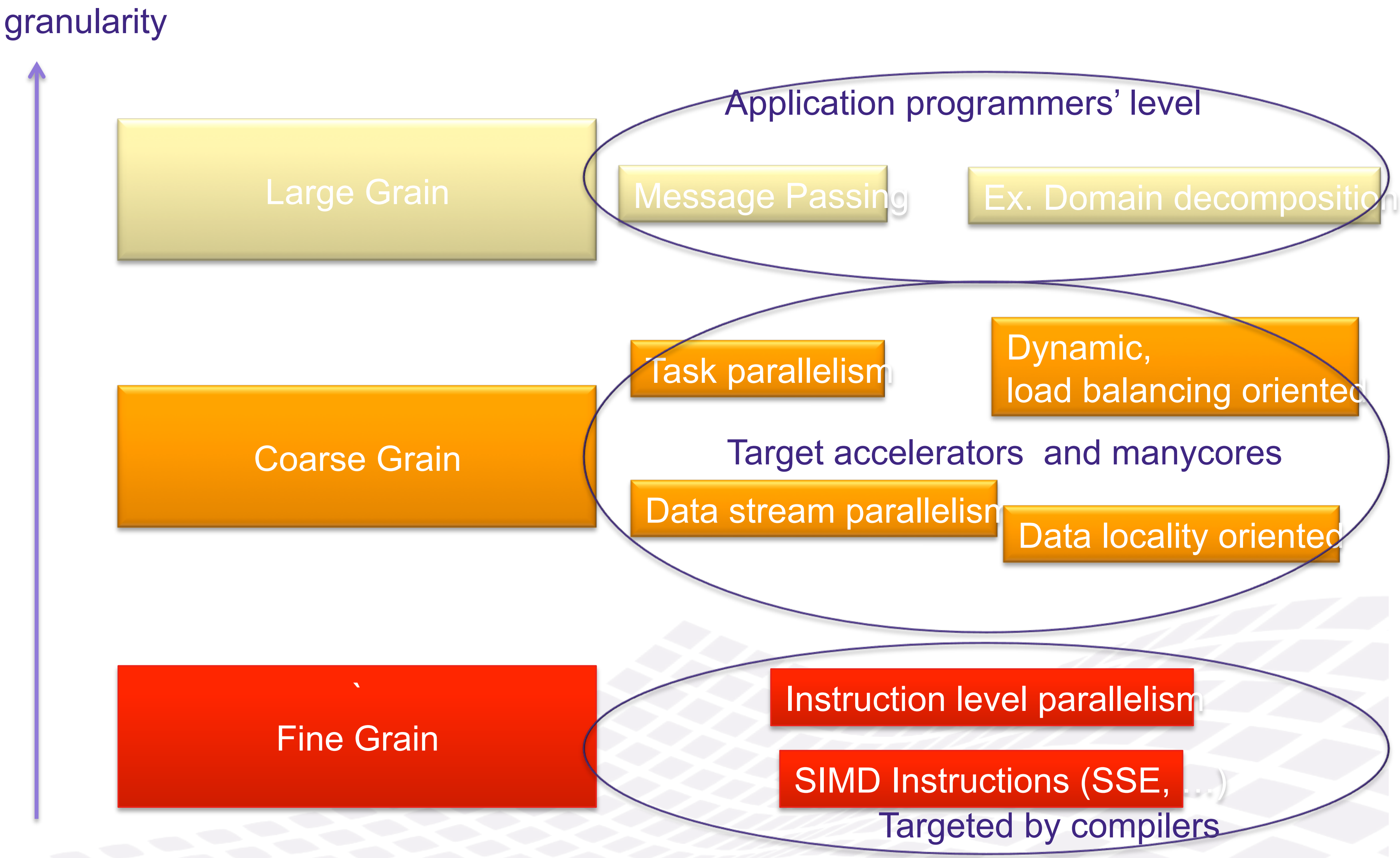
```
kernel void Fct(float a<>, float b<>, out float c<>) {
   c = a + b;
 }
int main(int argc, char** argv) {
   int i, j;
   float a<10, 10>, b<10, 10>, c<10, 10>;
   float input_a[10][10],input_b[10][10], input_c[10][10];
  for(i=0; i<10; i++) {
     for(j=0; j<10; j++) {
      input_a[i][j] = (float) i;
      input_b[i][j] = (float) j;
    }
  }
 streamRead(a, input_a);
 streamRead(b, input_b);
 Fct(a, b, c);
 streamWrite(c, input_c);
 ...
}
```

Brook+ example



A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]
B[0] B[1] B[2] B[3] B[4] B[5] B[6] B[7] B[8] B[9]
Fct Fct Fct Fct Fct Fct Fct Fct Fct Fct
C[0] C[1] C[2] C[3] C[4] C[5] C[6] C[7] C[8] C[9]

Heterogeneous hardware, multiple parallelism forms



granularity

Large Grain

Application programmers' level

Message Passing

Ex. Domain decomposition

Coarse Grain

Task parallelism

Dynamic,
load balancing oriented

Target accelerators and manycores

Data stream parallelism

Data locality oriented

`
Fine Grain

Instruction level parallelism

SIMD Instructions (SSE, …)

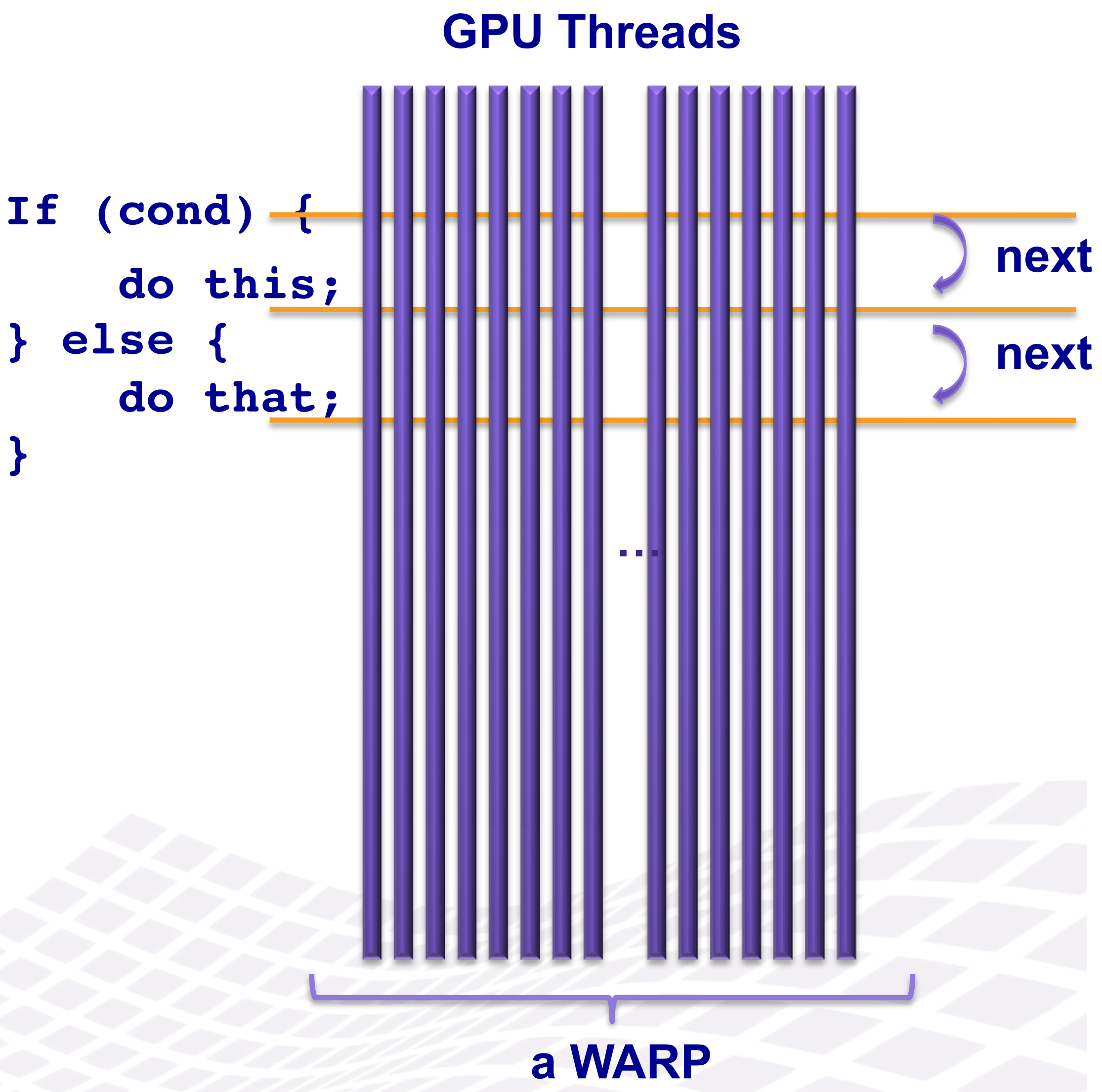
Targeted by compilers

Hybrid Programming for Future Manycores

- Agnostic programming is paramount
  - Highlight parallelism not its implementation
- Use the right parallelism level for each part
  - Software engineering is important
  - Separate application issues from performance issues
    - Specialized components, libraries, …
- Do no expect a common programming API for all levels
  - API always makes some underlying architecture assumptions
    - Fixing API makes hypothesis on the future of architectures
  - No low level programming API common to all devices
  - An API addresses a specific hardware component as a consequence we need many
- Plan for debugging and tuning
  - Parallel bugs are nasty
  - Tuning is target specific

What is a GPU Thread?

- A GPU thread is characterized by
  - A set of statements
  - A unique identifier
- All threads have the same code (SPMD)
- Two GPU threads belonging to a WARP (i.e. a set of threads) are executed in a lock step manner

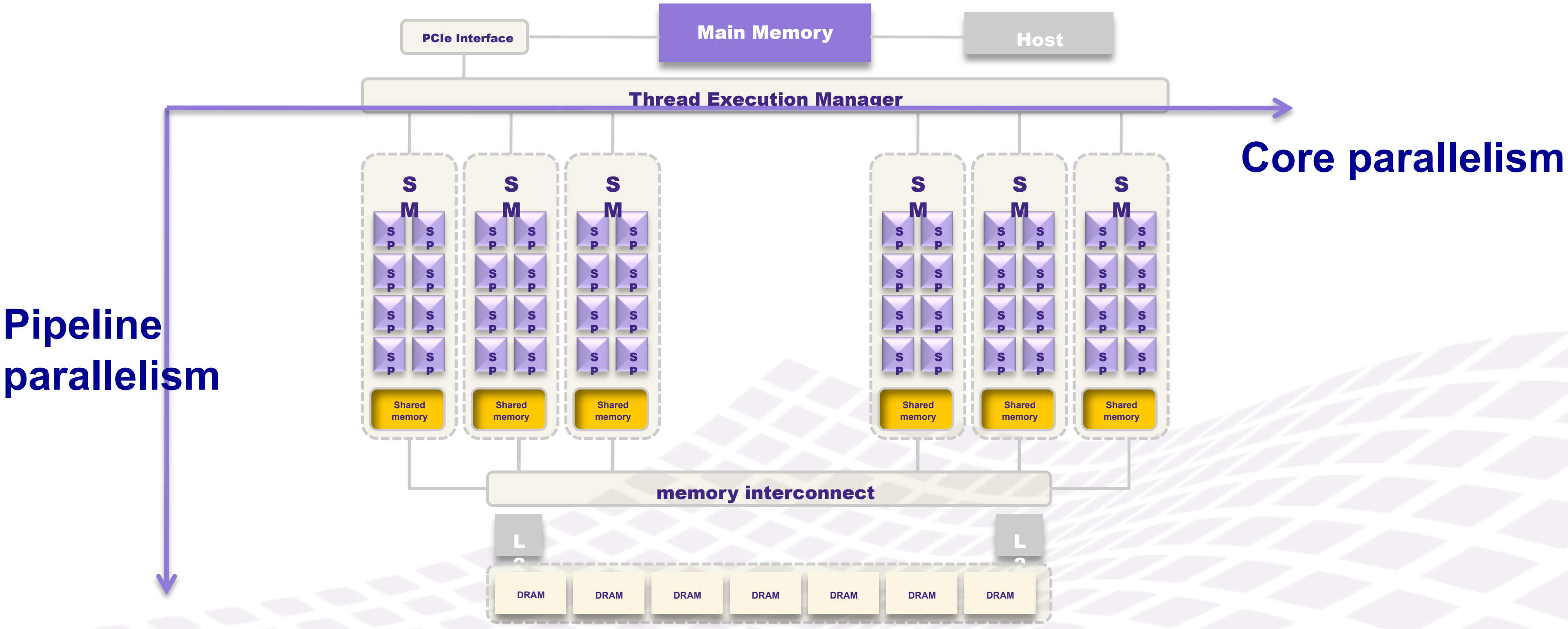


GPU Threads

```
If (cond) {
    do this;
} else {
    do that;
}
```

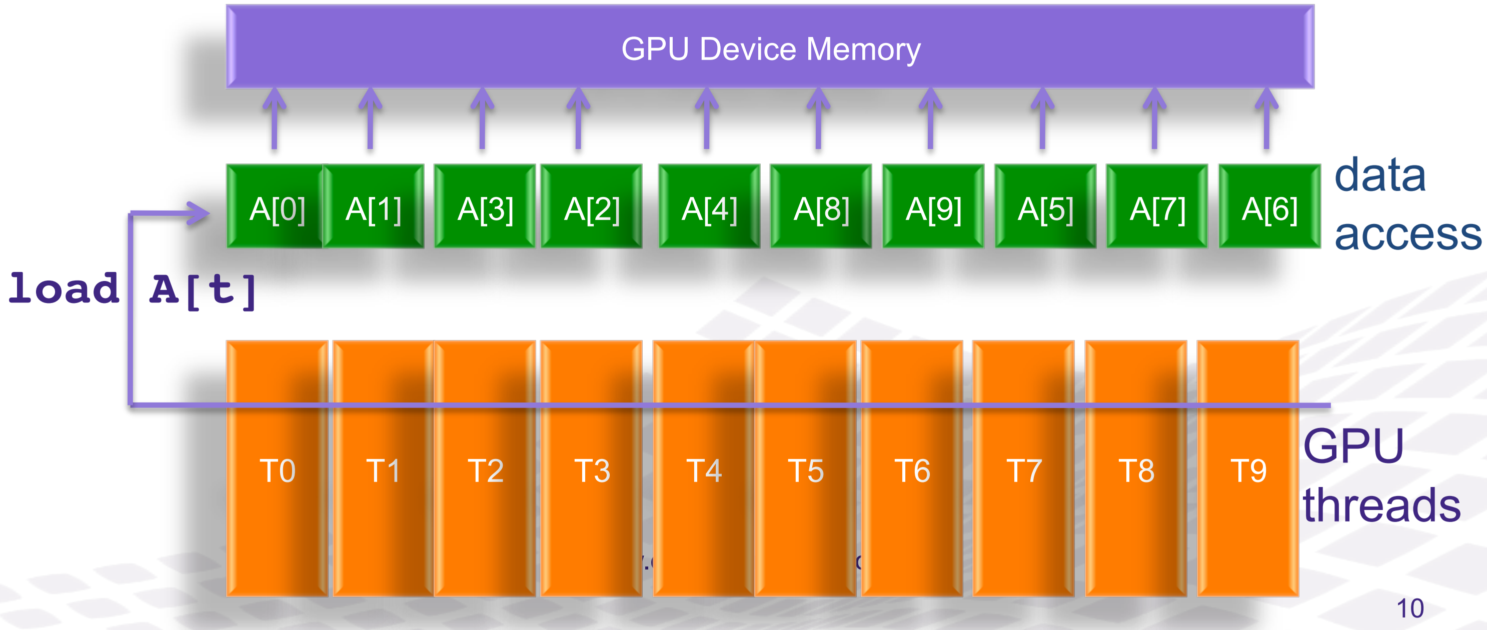
How Does a GPU Work?

- GPUs achieve high performance by exploiting massive thread parallelism
  - Threads are distributed over the numerous cores
  - Thread execution is pipeline on a core to avoid waiting for memory accesses
  - Memory accesses of multiple threads are grouped (coalesced) into faster accesses by exploiting spatial locality



Why is Memory Coalescing Important (NVidia)?

- Memory coalescing aggregates memory accesses to contiguous data in device memory into wider and more efficient ones
  - Depends on spatial locality between threads
  - When fails, memory accesses are serialized
  - Most algorithm are memory bound and so very sensitive to performance of the memory accesses



Toward Hardware Convergence

- Convergence of a mix of fast cores with many others in the same address space
  - Intel
    - Sandybridge, MIC
    - http://www.intel.com/technology/architecture-silicon/2ndgen/index.htm
  - AMD
    - Fusion (Accelerated Processing Unit)
    - http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx
  - Nvidia
    - Denvers
    - http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/
- But homogeneous not dead yet
  - IBM
  - Fujitsu
  - ARM (?)
- How to deal with succeeding generations of manycore hardware?

NVIDIA® CUDA™

- "Compute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

Extended C

CAPS

- Declspecs
  - global, device, shared, local, constant

```
__device__ float filter[N];
__global__ void convolve (float *image)  {
  __shared__ float region[M];
  ...
```

- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads

```
  region[threadIdx] = image[i];
  __syncthreads()
  ...
  image[j] = result;
}
```

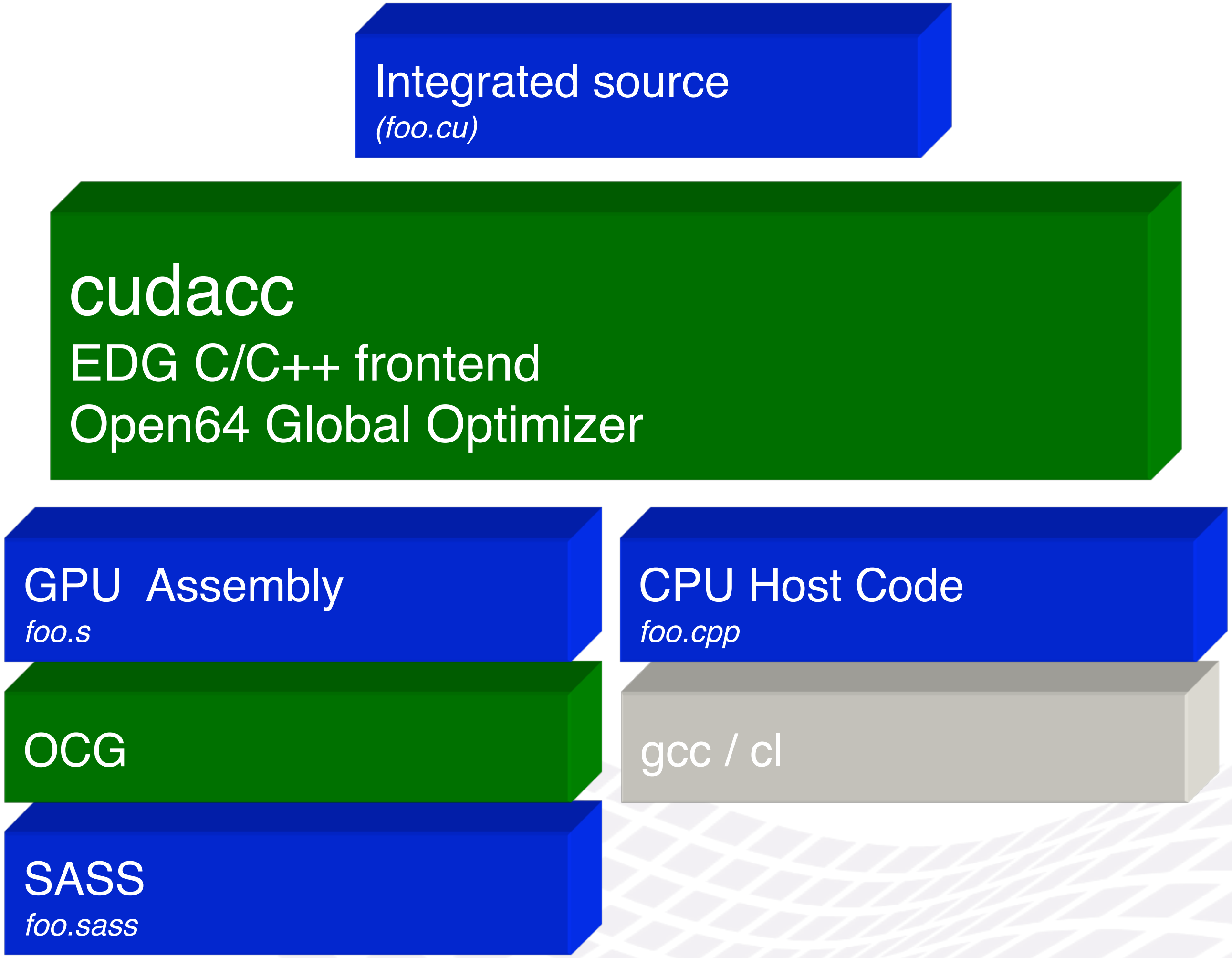
- Runtime API
  - Memory, symbol, execution management

```
// Allocate GPU memory
void *myimage = cudaMalloc(bytes)
```

- Function launch

```
// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

Extended C


Integrated source

(foo.cu)

cudacc

EDG C/C++ frontend

Open64 Global Optimizer

GPU Assembly

foo.s

CPU Host Code

foo.cpp

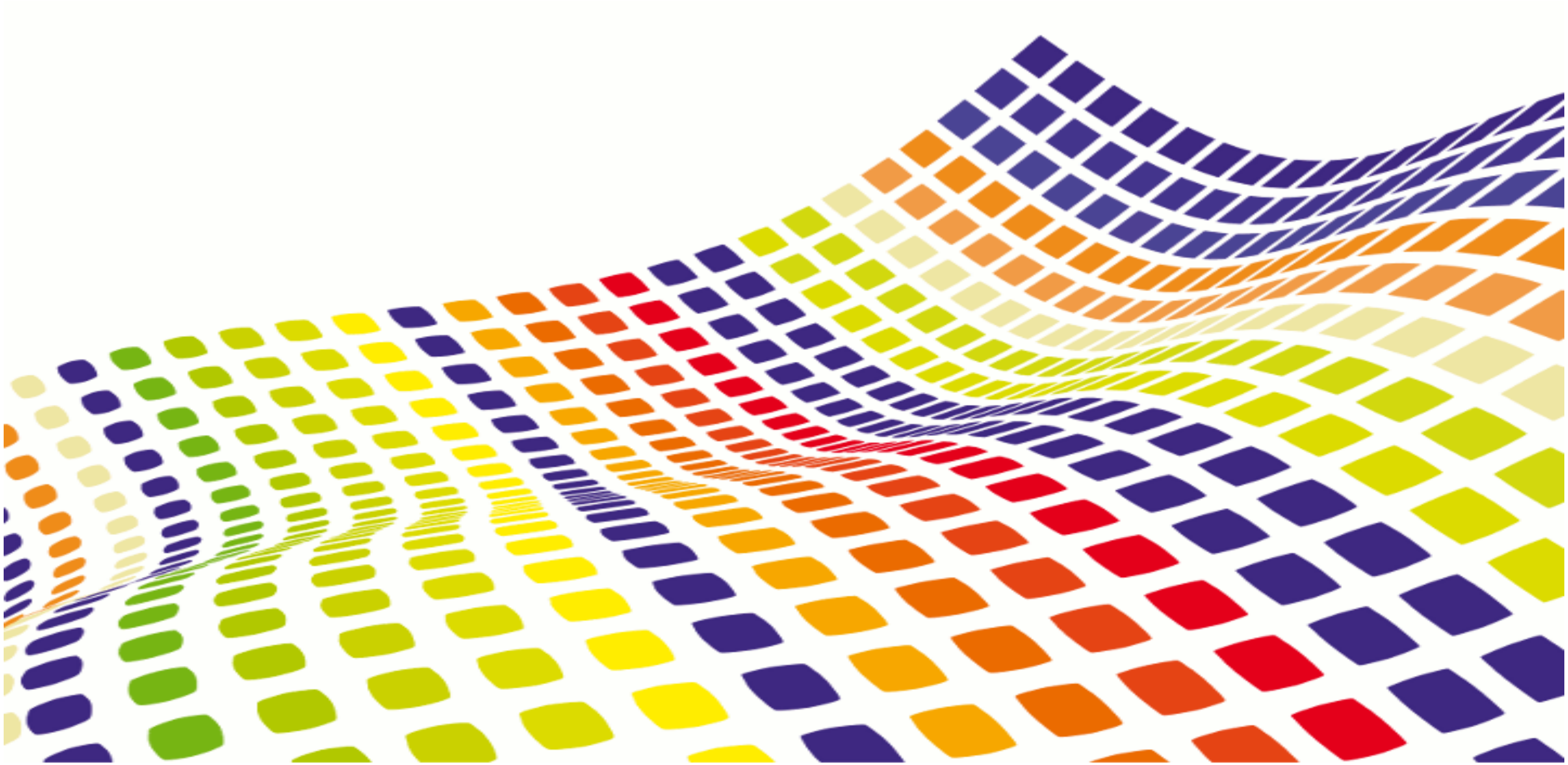
OCG

gcc / cl

SASS

foo.sass

CUDA Programming Model



A Highly Multithreaded Coprocessor

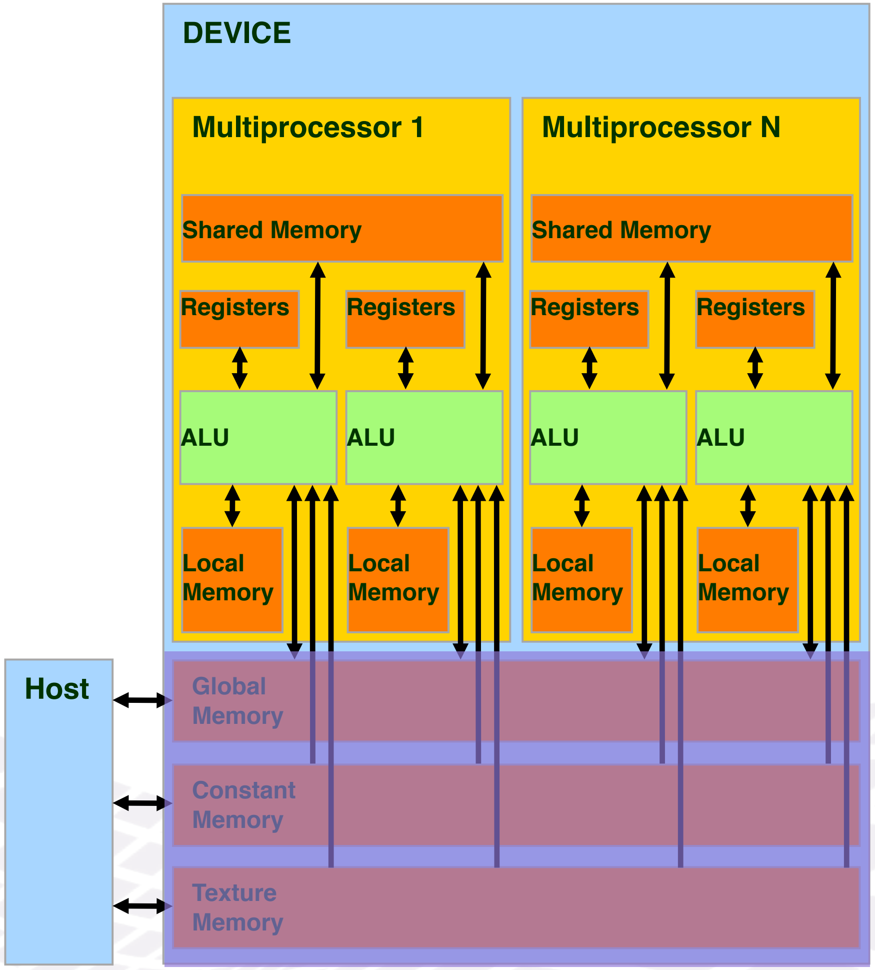
- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel

- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

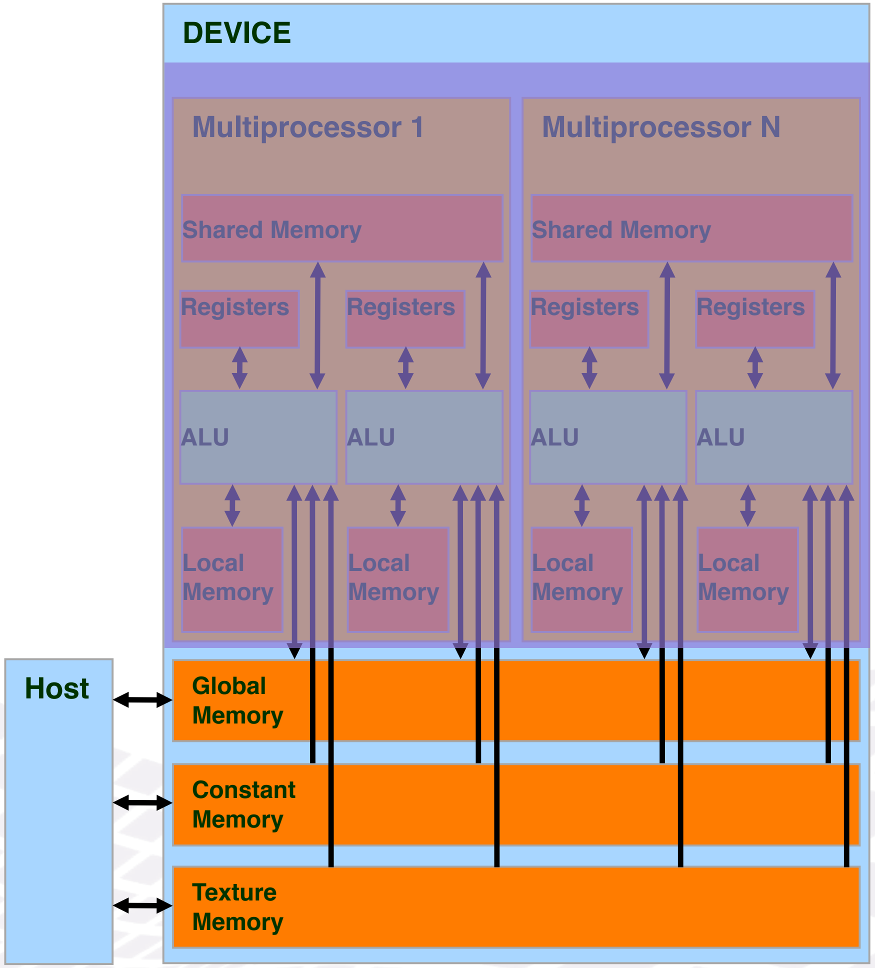
Hardware Overview

- Device contains
  - Multiprocessors
  - Host access interface
  - Memory
  - 4 generations :
    - 1.0 (8800GTX),
    - 1.1 (9800GTX), 1.3 (GTX280)
    - and 2.0 (Fermi)
- Multiprocessors contains
  - ALUs
  - Registers
  - Shared Memory
  - Access to Local Memory
  - Access to Global Memory



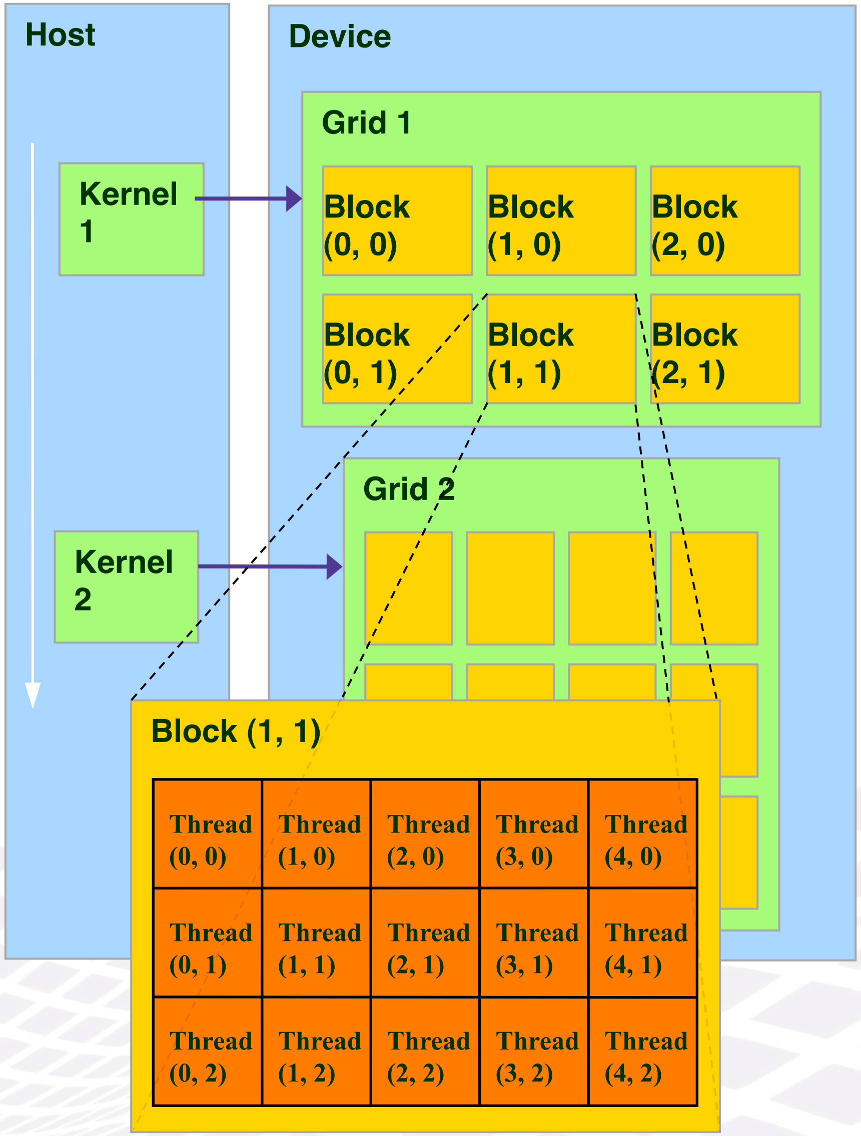
Global Memory Overview

- Global memory
  - Main means of communicating R/W data between host and device
  - Contents visible to all threads
  - No cache (appears with Fermi)
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads
  - Cache available



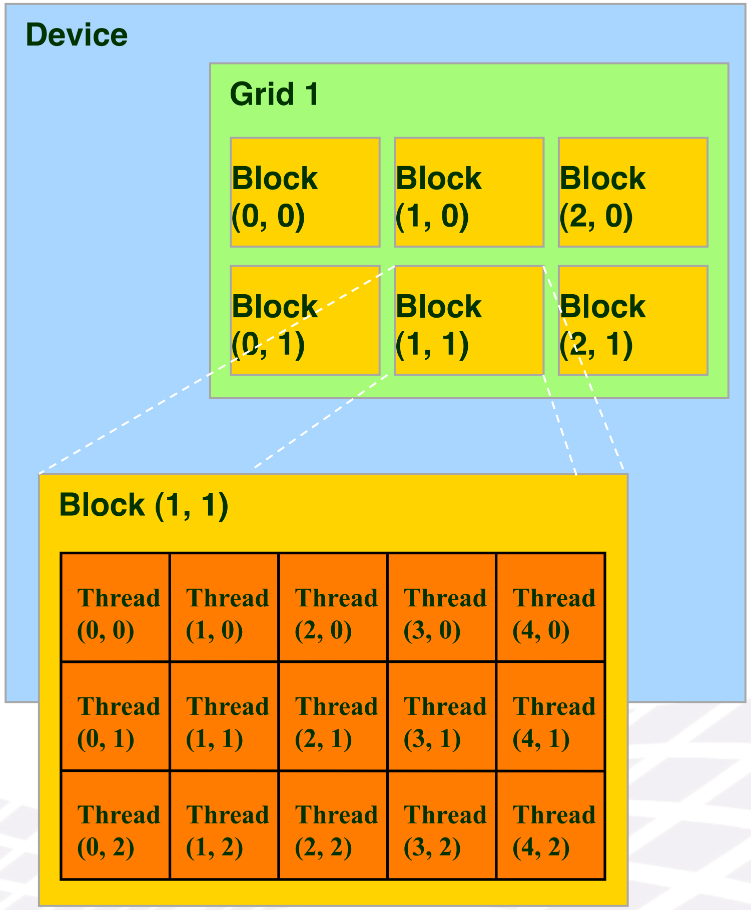
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
  - Atomic operations added in HW 1.1



Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Device

Grid 1

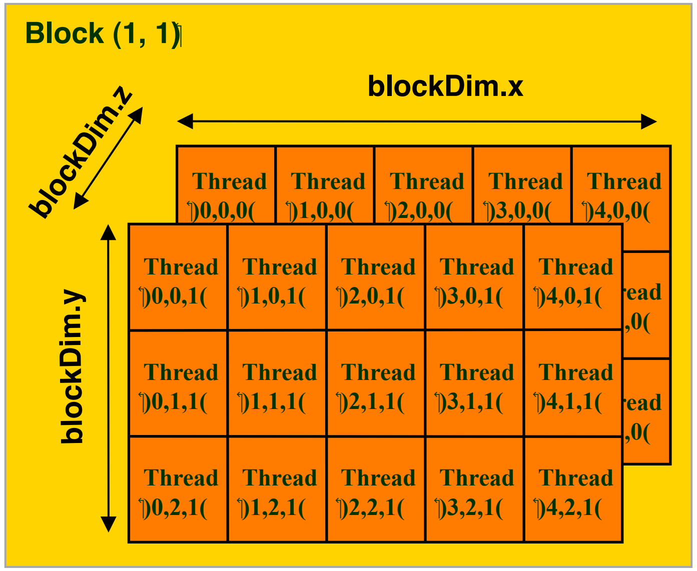
Block (0, 0)	Block (1, 0)	Block (2, 0)
Block (0, 1)	Block (1, 1)	Block (2, 1)

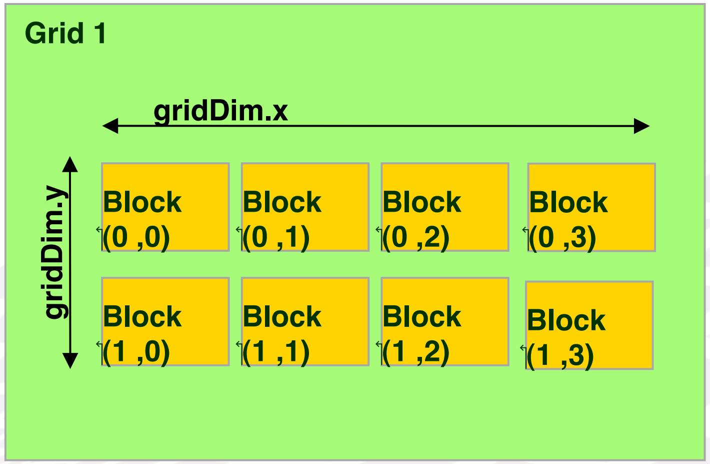
Block (1, 1)

Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)

Block and Thread keywords

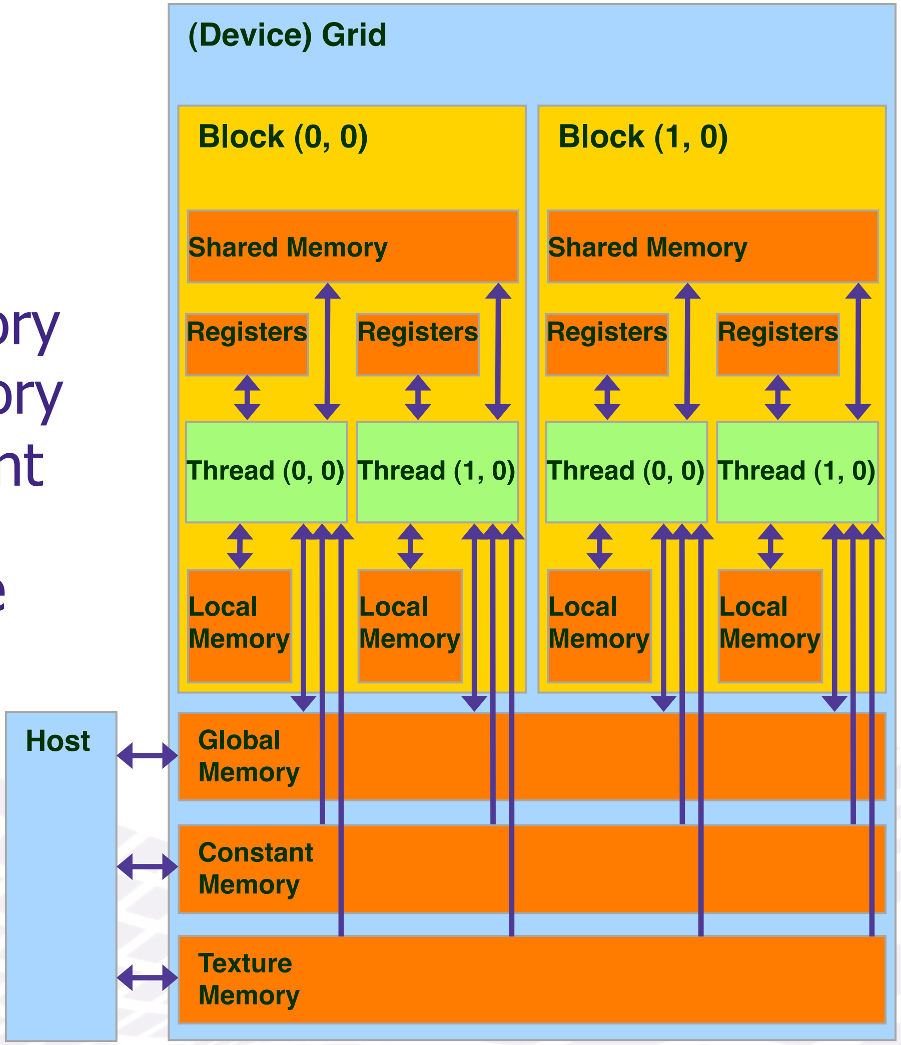
- Block keywords
  - threadIdx.{x,y,z} defines the thread index inside the block
  - blockDim.{x,y,z} defines the block dimensions
- Grid keywords
  - blockIdx.{x,y} defines the block index inside the grid
  - gridDim.{x,y} defines the grid dimension



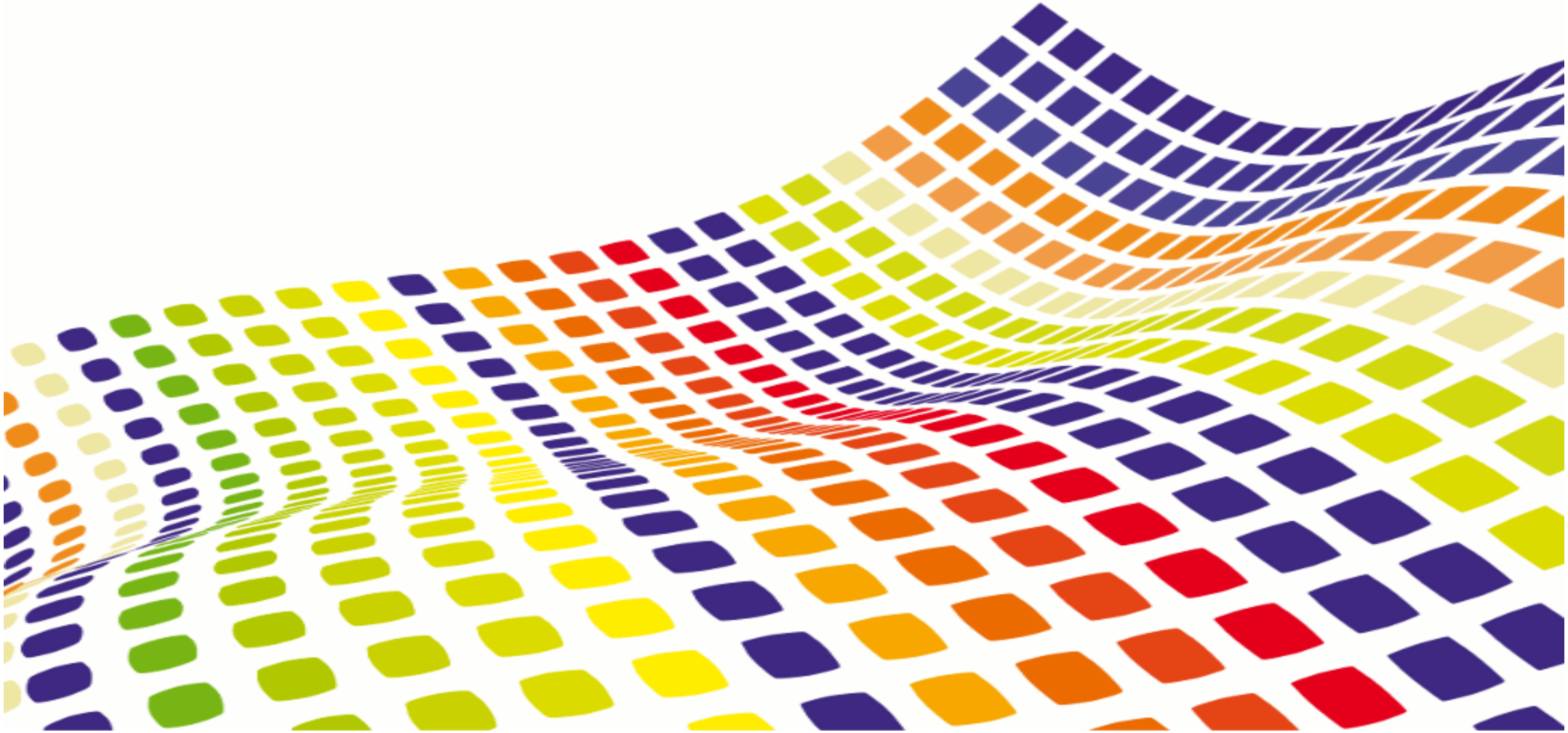


Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread shared memory
  - R/W per-block local memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can:
  - R/W global memory
  - R/W constant memory
  - R/W texture memory



CUDA API



CUDA Highlights : Easy and Lightweight

- The API is an extension to the ANSI C programming language

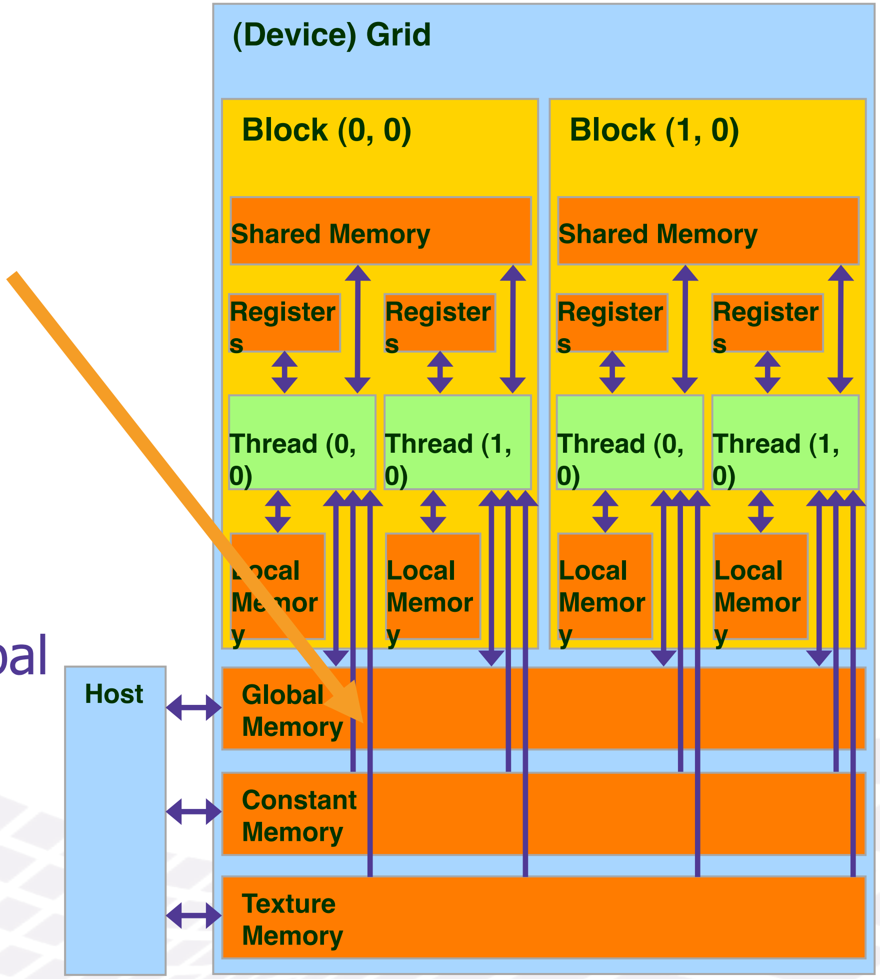
→ Low learning curve

- The hardware is designed to enable lightweight runtime and driver

→ High performance

Memory Allocation

- cudaMalloc()
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - Address of a pointer to the allocated object
    - Size of of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



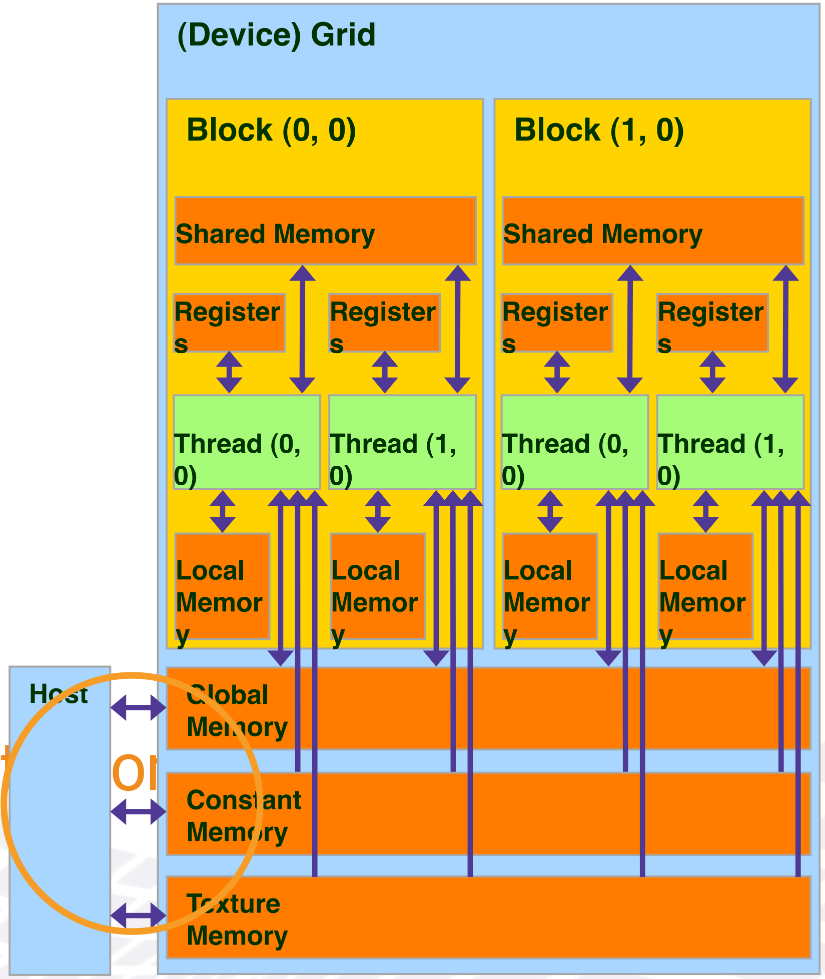
Memory Allocation Examples

- Code example:
  - Allocate a 1024 * 1024 single precision float matrix

```
#define MATRIX_SIZE 1024*1024
float* MyMatrixOnDevice;
int size = MATRIX_SIZE * sizeof(float);

cudaMalloc((void**)&MyMatrixOnDevice, size);
cudaFree(MyMatrixOnDevice);
```

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires 4 parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous variant support or 1.1+HW



CUDA Host-Device Data Transfer Examples

- Code example:
  - Transfer a 1024 * 1024 single precision float array
  - MyMatrixOnHost is in host memory and MyMatrixOnDevice is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(MyMatrixOnDevice, MyMatrixOnHost, size,
                  cudaMemcpyHostToDevice);

cudaMemcpy(MyMatrixOnHost, MyMatrixOnDevice, size,
                      cudaMemcpyDeviceToHost);
```

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`

CUDA Functions Declaration

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Calling a Kernel Function Thread Creation

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);     // 5000 thread blocks
dim3   DimBlock(8, 8, 4);    // 256 threads per block

KernelFunc<<< DimGrid, DimBlock >>>(...);
```

- Any call to a kernel function is asynchronous, explicit synchronization needed for blocking

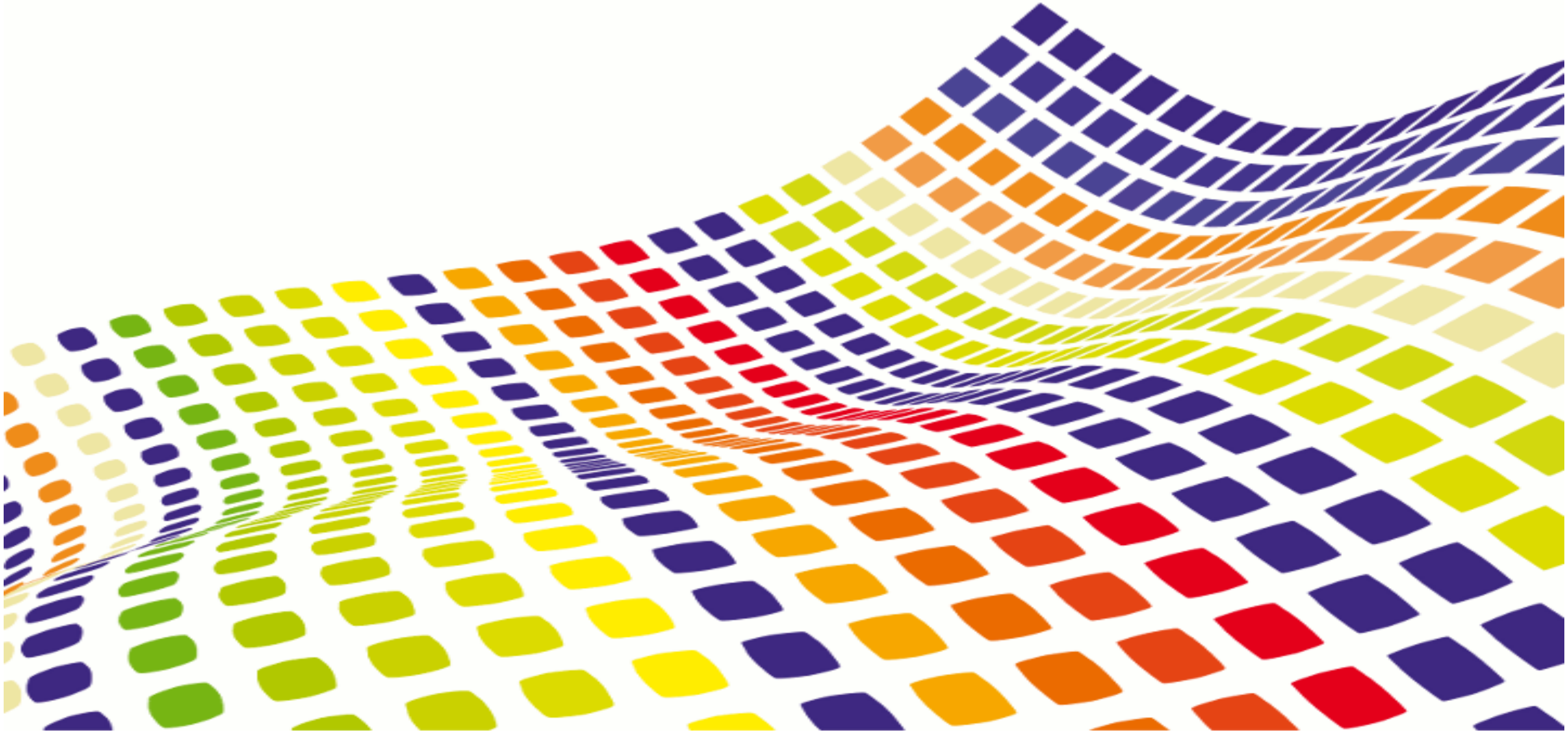
CUDA Example (1)

```
#include <stdio.h>
#include <cutil.h>
__global__
void simplefunc(float *v1, float *v2, float *v3) {
   int i = blockIdx.x * 100 + threadIdx.x;
   v1[i] = v2[i] * v3[i];
}

int main(int argc, char **argv) {
  unsigned int n = 400;
  float *t1 = NULL;float *t2 = NULL; float *t3 = NULL;
  unsigned int i, j, k, seed = 2, iter = 3;
  /* create the CUDA grid 4x1 */
  dim3 grid(4,1);
  /* create 100x1 threads per grid element */
  dim3 thread(100,1);

  t1 = (float *) calloc(n*iter, sizeof(float));
  t2 = (float *) calloc(n*iter, sizeof(float));
  t3 = (float *) calloc(n*iter, sizeof(float));

  printf("parameters: seed=%d, iter=%d, n=%d\n", seed, iter, n);
```

CUDA Example (2)

```
  /* initialize CUDA device */
  CUT_DEVICE_INIT()
  …
  /* allocate arrays on device */
  float *gpu_t1 = NULL;
  float *gpu_t2 = NULL;
  float *gpu_t3 = NULL;
  cudaMalloc((void**) &gpu_t1, n*sizeof(float));
  cudaMalloc((void**) &gpu_t2, n*sizeof(float));
  cudaMalloc((void**) &gpu_t3, n*sizeof(float));
 for (k = 0 ; k < iter ; k++) {
    /* copy data on gpu */
    cudaMemcpy(gpu_t2,&(t2[k*n]), n*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_t3,&(t3[k*n]), n*sizeof(float), cudaMemcpyHostToDevice);
    simplefunc<<<grid,thread>>>(gpu_t1,gpu_t2,gpu_t3);
    /* get back data from gpu */
    cudaMemcpy(&(t1[k*n]),gpu_t1, n*sizeof(float), cudaMemcpyDeviceToHost);
  }

  …
  return 0;
}
```

OpenCL



What is Hybrid Computing with OpenCL?

- OpenCL is
  - Open, royalty-free, standard
  - For cross-platform, parallel programming of modern processors
  - An Apple initiative
  - Approved by Intel, Nvidia, AMD, etc.
  - Specified by the Khronos group (same as OpenGL)

- It intends to unify the access to heterogeneous hardware accelerators
  - CPUs (Intel i7, AMD, …)
  - GPUs (Nvidia, AMD/ATI, …)
  - Embedded systems

- Portable syntax, not portable performance!

OpenCL vs CUDA

- OpenCL and CUDA share the same parallel programming model
- Runtime API are different
  - OpenCL is lower level than CUDA
- OpenCL and CUDA may use different implementations that could lead to different execution times for a similar kernel on the same hardware

OPENCL	CUDA
kernel	kernel
host pgm	host pgm
NDrange	grid
work item	thread
work group	block
Global mem	global mem
cst mem	cst mem
local mem	shared mem
private mem	local mem

OCL Kernel

```
__kernel void DotProduct ( __global const float16* a,
__global const float16* b, __global float4* c,
__local float16 f16InA[LOCAL_WORK_SIZE],__local float16 f16InB
[LOCAL_WORK_SIZE],__local float4 f4Out[LOCAL_WORK_SIZE]){
    // find position in global oct-float array
    int iGID = get_global_id(0);
    int iLID = get_local_id(0);
    // read 16 floats into LMEM from GMEM for each input array
    f16InA[iLID] = a[iGID];
    f16InB[iLID] = b[iGID];
    // process 4 pixels into output LMEM
    f4Out[iLID].x = f16InA[iLID].s0 * f16InB[iLID].s0
                  + f16InA[iLID].s1 * f16InB[iLID].s1
                  + f16InA[iLID].s2 * f16InB[iLID].s2
                  + f16InA[iLID].s3 * f16InB[iLID].s3;
             . . .

    f4Out[iLID].w = f16InA[iLID].sc * f16InB[iLID].sc
                  + f16InA[iLID].sd * f16InB[iLID].sd
                  + f16InA[iLID].se * f16InB[iLID].se
                  + f16InA[iLID].sf * f16InB[iLID].sf;
    // write out 4 floats to GMEM
    c[iGID] = f4Out[iLID];
}
```

An Example-1

```
int main(int /*argc*/, char **argv)
{
    cl_context cxMainContext;       // OpenCL context
    cl_command_queue cqCommandQue;  // OpenCL command que
    cl_device_id* cdDevices;        // OpenCL device list
    cl_program cpProgram;           // OpenCL program
    cl_kernel ckKernel;             // OpenCL kernel
    cl_mem cmMemObjs[6];            // OpenCL memory buffer objects host & device
    size_t szGlobalWorkSize[1];     // Total # of work items
    size_t szLocalWorkSize[1];      // # of work items in the work group
    size_t szParmDataBytes;         // Byte size of context inf.
    size_t szKernelLength;          // Byte size of kernel code
    cl_int ciErr1, ciErr2;          // Error code var
    int iTestN = 350000 * 16;       // Size of Vectors to process
    // set Global and Local work size dimensions
#define LOCAL_WORK_SIZE 32
    szGlobalWorkSize[0] = iTestN >> 2;  // compute 4 at a time
    szLocalWorkSize[0]= LOCAL_WORK_SIZE;
 // start log and timer 0 and 1
    ocutSetLogFileName ("OpenclSdkDotProductTest.txt");
    ocutWriteLog(LOGBOTH, 0.0, "oclDotProduct.exe Starting, . . .  »);
    ocutDeltaT(0);
    ocutDeltaT(1);
```

An Example-2

```
// Allocate and initialize host arrays for golden computations
srcA = (void *)malloc(sizeof(cl_float4) * iTestN);
srcB = (void *)malloc(sizeof(cl_float4) * iTestN);
dst = (void *)malloc(sizeof(cl_float) * iTestN);
Golden = (void *)malloc(sizeof(cl_float) * iTestN);
ocutFillArray((float*)srcA, 4 * iTestN);
ocutFillArray((float*)srcB, 4 * iTestN);
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1),  "Allocate . . .  \n");
// Create the OpenCL context on a GPU device
cxMainContext = clCreateContextFromType (0, CL_DEVICE_TYPE_GPU, . . .);
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "clCreateContextFromType\n");

// Get the list of GPU devices associated with context
ciErr1 |= clGetContextInfo(cxMainContext, CL_CONTEXT_DEVICES, 0, . . .);
cdDevices = (cl_device_id*)malloc(szParmDataBytes);
ciErr1 |= clGetContextInfo(cxMainContext, CL_CONTEXT_DEVICES, . . .);
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "clGetContextInfo\n");
ocutPrintDeviceInfo(cdDevices[0]);
// Create a command-queue
cqCommandQue = clCreateCommandQueue (cxMainContext,cdDevices[0],0,&ciErr2);
ciErr1 |= ciErr2;
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1),"clCreateCommandQueue\n");
```

An Example-3

```
// Allocate and initialize OpenCL source and result buffer Pinned
   memory objects on the host
    cmMemObjs[0] = clCreateBuffer (cxMainContext,...);
    ciErr1 |= ciErr2;
    cmMemObjs[1] = clCreateBuffer(cxMainContext, ...);
    ciErr1 |= ciErr2;
    cmMemObjs[2] = clCreateBuffer(cxMainContext, ...);
    ciErr1 |= ciErr2;
    ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "clCreateBuffer pinned\n");

// Allocate the OpenCL source and result buffer memory objects on the device GMEM
    cmMemObjs[3] = clCreateBuffer (cxMainContext, ...);
    ciErr1 |= ciErr2;
    cmMemObjs[4] = clCreateBuffer(cxMainContext, ...);
    ciErr1 |= ciErr2;
    cmMemObjs[5] = clCreateBuffer(cxMainContext, CL_MEM_WRITE_ONLY, ...);
    ciErr1 |= ciErr2;
 if (ciErr1 != CL_SUCCESS) exit (...);
    ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "clCreateBuffer GMEM\n");
// Read the kernel in from file
    const char* cPathAndName = ocutFindFilePath(clSourcefile, argv[0]);
    char* cDotProduct = ocutLoadProgramSource (cPathAndName,
      "// My comment\n", &szKernelLength);
    ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "ocutLoadProgramSource\n");
```

An Example-4

```
// Create the program
cpProgram = clCreateProgramWithSource (cxMainContext, 1,
            (const char **)&cDotProduct, &szKernelLength, &ciErr1);
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "clCreateProgramWithSource\n");
// Build the program
ciErr1 |= clBuildProgram (cpProgram, 0, NULL, NULL, NULL, NULL);
if (ciErr1 != CL_SUCCESS) {
    // write out standard error
    ocutWriteLog(LOGBOTH | ERRORMSG, (double)ciErr1, STDERROR);
    // write out the build log
    char cBuildLog[10240];
    clGetProgramBuildInfo (cpProgram, ocutGetFirstDevice(cxMainContext),
         CL_PROGRAM_BUILD_LOG, sizeof(cBuildLog), cBuildLog, NULL);
    ocutWriteLog(LOGBOTH, 0.0, "\n\nLog:\n%s\n\n\n", cBuildLog);
    // write out the ptx and then exit
    char* cPtx;
    size_t szPtxLength;
    ocutGetProgramBinary (cpProgram, ocutGetFirstDevice(cxMainContext),
      &cPtx, &szPtxLength);
    ocutWriteLog(LOGBOTH| CLOSELOG, 0.0, "\n\nPtx:\n%s\n\n\n", cPtx);
    exit (-1);
}
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "clBuildProgram\n");
```

An Example-5

```
// Create the kernel
ckKernel = clCreateKernel(cpProgram, "DotProduct", &ciErr1);
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "clCreateKernel\n");
// Set the Argument values
ciErr1 = clSetKernelArg (ckKernel, 0, sizeof(cl_mem), (void*)&cmMemObjs[3]);
ciErr1 |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&cmMemObjs[4]);
ciErr1 |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&cmMemObjs[5]);
ciErr1 |= clSetKernelArg(ckKernel, 3, (LOCAL_WORK_SIZE*sizeof(cl_float16)),NULL);
ciErr1 |= clSetKernelArg(ckKernel, 4, (LOCAL_WORK_SIZE*sizeof(cl_float16)),NULL);
ciErr1 |= clSetKernelArg(ckKernel, 5, (LOCAL_WORK_SIZE*sizeof(cl_float4)), NULL);
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "clSetKernelArg\n");
// Warmup GPU driver
ciErr1 |= clEnqueueNDRangeKernel(cqCommandQue, ckKernel, 1, NULL, szGlobalWorkSize,
          szLocalWorkSize, 0, NULL, NULL);
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "Warmup GPU Driver\n");
// Execute kernel iNumIterations times
for (int i = 0; i < iNumIterations; i++){
    ciErr1 |= clEnqueueNDRangeKernel(cqCommandQue, ckKernel, 1, NULL,
                            szGlobalWorkSize, szLocalWorkSize, 0, NULL, NULL);
}
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1)/iNumIterations,
            "clEnqueueNDRangeKernel (compute)\n");
```

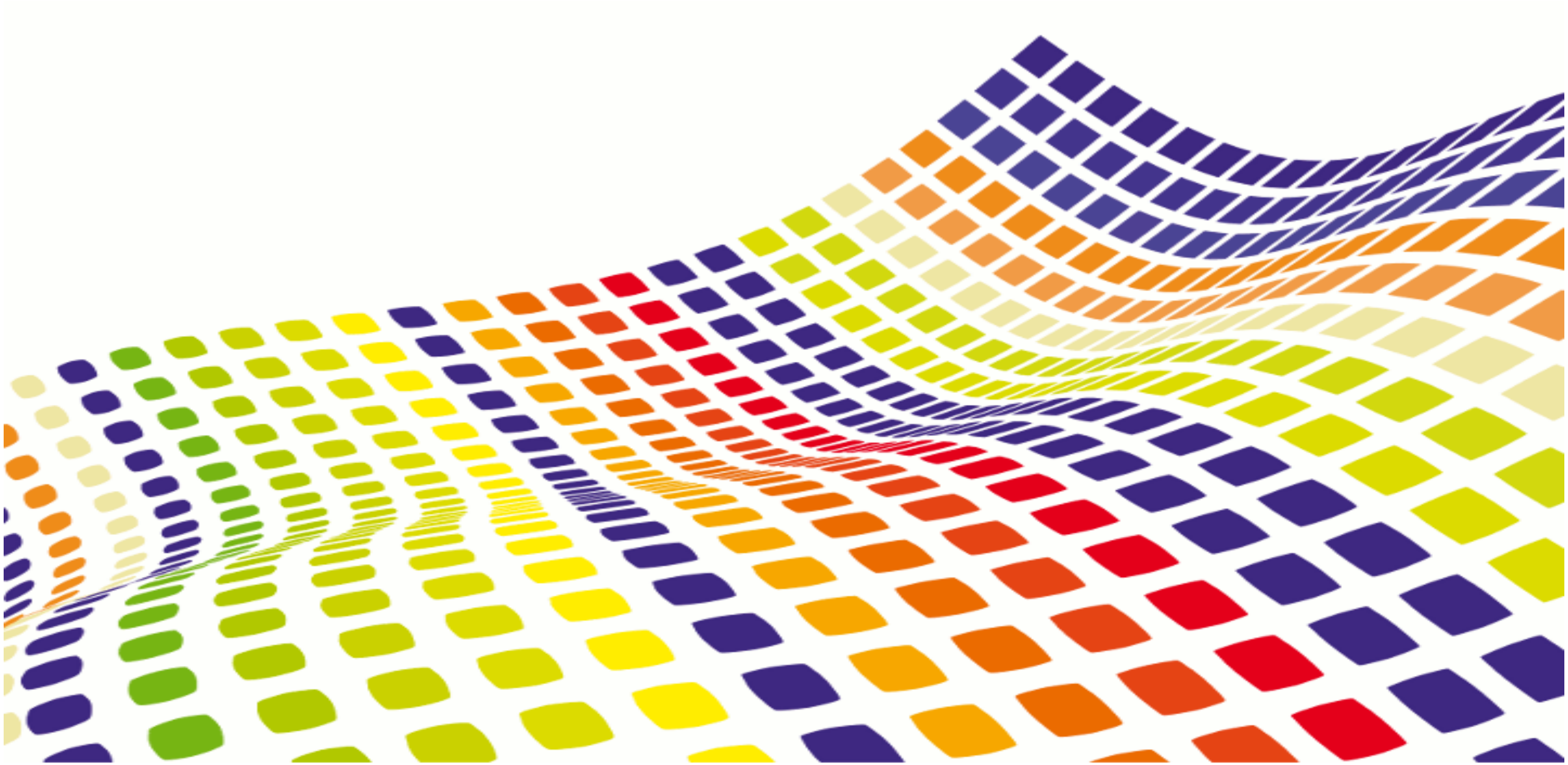
An Example-6

```
// Read output
ciErr1 |= clEnqueueReadBuffer (cqCommandQue, cmMemObjs[5], CL_TRUE, 0,
          sizeof(cl_float4) * szGlobalWorkSize[0], dst, 0, NULL, NULL);
if (ciErr1 != CL_SUCCESS) exit (ocutWriteLog(LOGBOTH | ERRORMSG | CLOSELOG,
                                (double)ciErr1, STDERROR));
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "clEnqueueReadBuffer\n");
// Release kernel, program, and memory objects
free(cdDevices);
free(cDotProduct);
clReleaseKernel (ckKernel);
clReleaseProgram (cpProgram);
clReleaseCommandQueue (cqCommandQue);
clReleaseContext (cxMainContext);
ocutDeleteMemObjs(cmMemObjs, 6);
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1), "Release OpenCL objects\n");
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(0), "Total Program Time\n\n");

// Compute results for golden-host (execute iNumIterations times)
 for (int i = 0; i < iNumIterations; i++){
    DotProductHost ((const float*)srcA,(const float*)srcB,(float*)Golden,iTestN);
}
ocutWriteLog(LOGBOTH | DELTAT, ocutDeltaT(1)/(float)iNumIterations,
        "Host Processing\n");
// Compare results (golden-host vs. device) and report errors and pass/fail
ocutDiffArray((const float*)dst, (const float*)Golden, iTestN);
```

An Example-7

```
        // Free host memory, close log and return success
        free(srcA);
        free(srcB);
        free (dst);
        free(Golden);
        ocutWriteLog(LOGBOTH | CLOSELOG, 0.0,
               "\nOpenCL DotProduct Demo End...\nPress <Enter> to Quit...\n");
        getchar();
        exit (0);
    }
```

HMPP



What is HMPP? (Hybrid Manycore Parallel Programming)

- 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

- 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

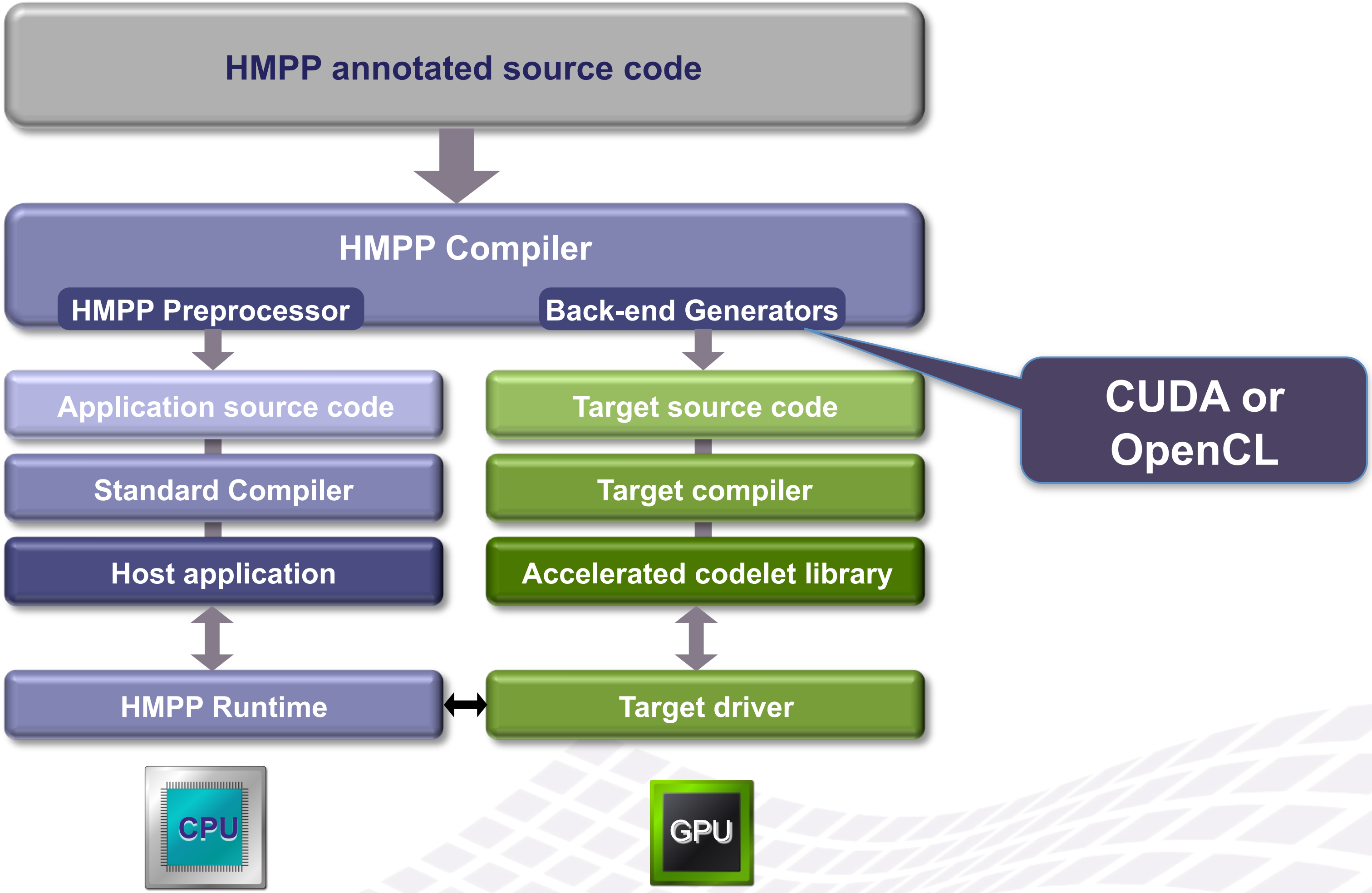
How Does HMPP Differ from CUDA or OpenCL?

- 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 Compilation Flow



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

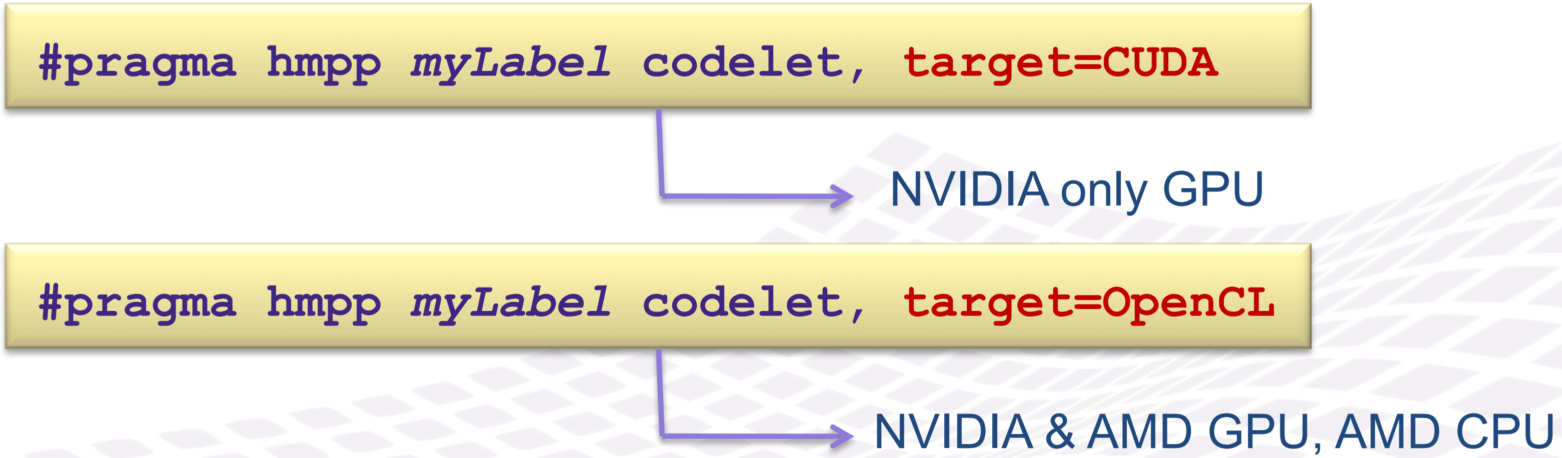
```
#pragma hmpp myfunc codelet, …
void saxpy(int n, float alpha, float x[n], float y[n])
{
#pragma hmppcg parallel
  for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```

```
#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

Parallel Loops in Codelets and Regions

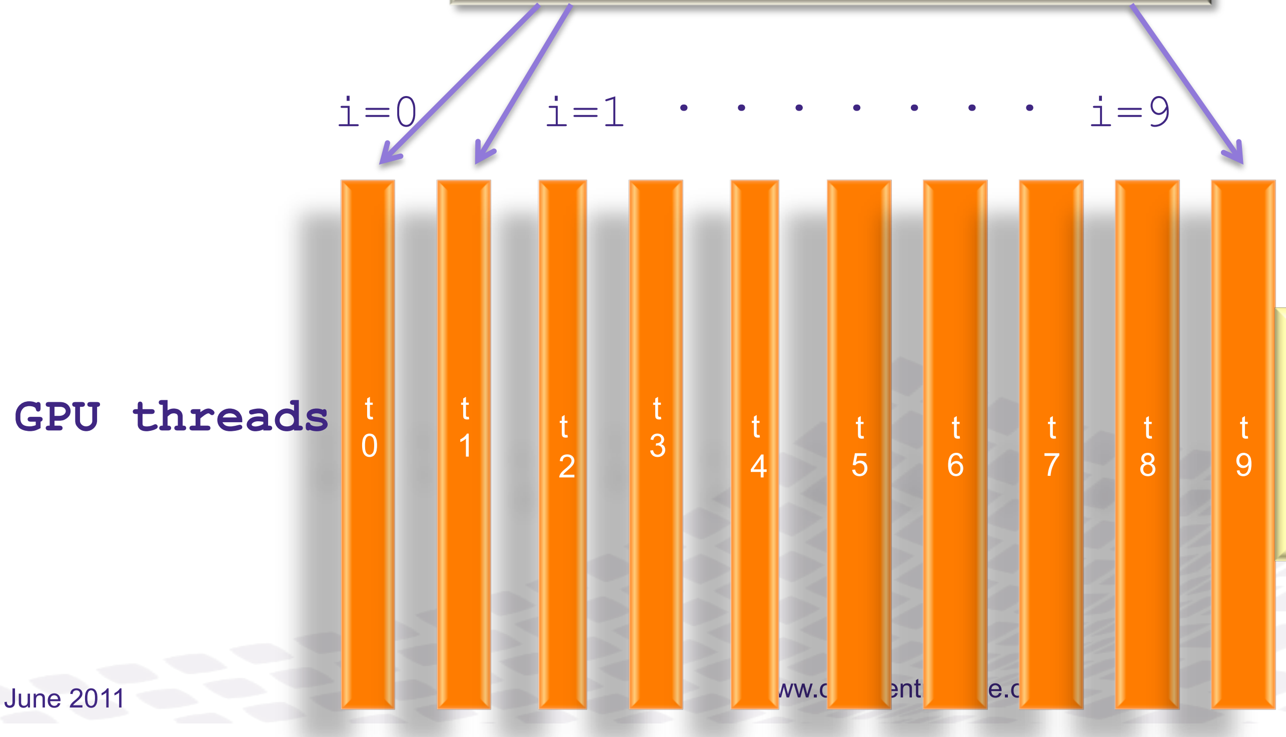
- Parallel loops are the code constructs converted in GPU threads
  - Using a process call *loop nest gridification*
  - Directive hmppcg parallel forces parallelisation
- Two levels of parallelism can be used to generate the threads

```
#pragma hmppcg parallel
for (i = si; i < ei; ++i){
#pragma hmppcg parallel
 for (j = sj; j < ej; ++j) {
   A[i][j] =  c11*A[i-di][j-dj]+c12*A[i][j-dj];
 }
}
```

Loop Nest *Gridification*

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

```
#pragma hmppcg parallel
for(int i = 0; i<10; ++i){
    y[i] = alpha*x[i] + y[i];
}
```



```
{
 int i = blockIdx.x*
 blockDim.x + threadIdx.x;
 if( i<10 )
   y[i]=alpha*x[i]+y[i];
}
```

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];
    }
}
```

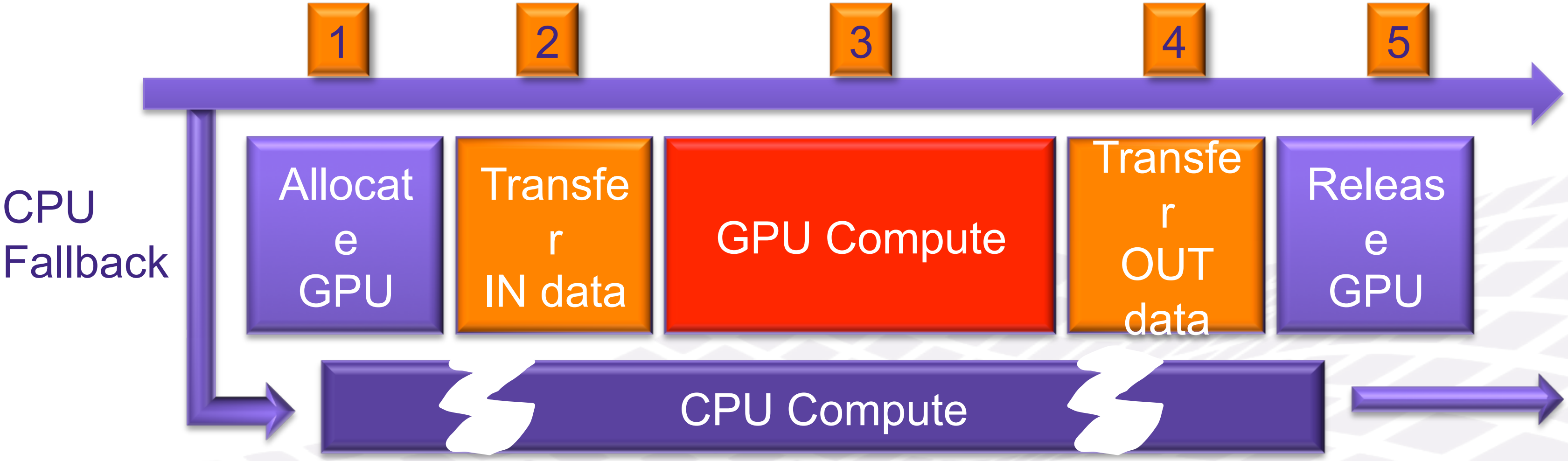
Running a Codelet or Section on a GPU - 1

- 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);
    …
}
```

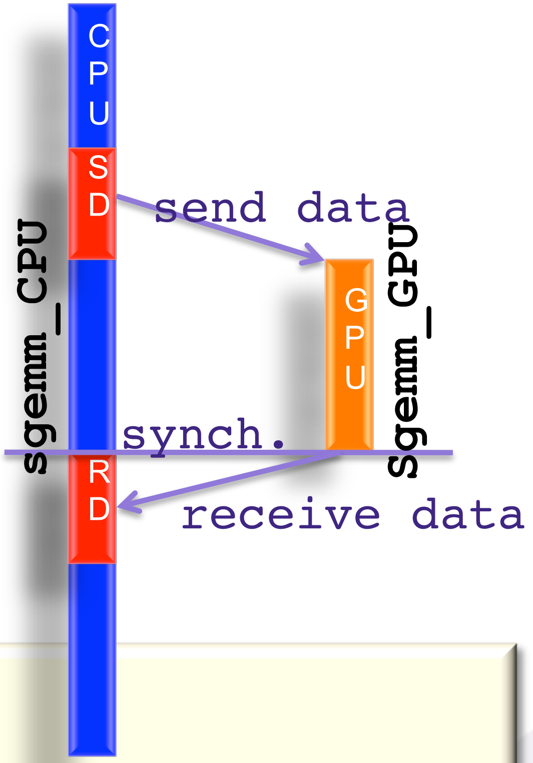
Running a Codelet or Section on a GPU - 2

- 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



Asynchronous Execution of Codelets

- Keep the CPU busy while the GPU is computing
- There should be no data dependence between the CPU and the GPU code executed after the *callsite* and the *synchronize* directive



```
int main(int argc, char **argv) {
    . . .
#pragma hmpp sgemm callsite, asynchronous
    sgemm_GPU( size, size, size, alpha, vin1, vin2g, beta, voutg );
    sgemm_CPU( size, size, size, alpha, vin1, vin2c, beta, voutc );
#pragma hmpp sgemm synchronize
#pragma hmpp sgemm delegatedstore, args[voutg]
```

Tuning Hybrid Codes

- 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

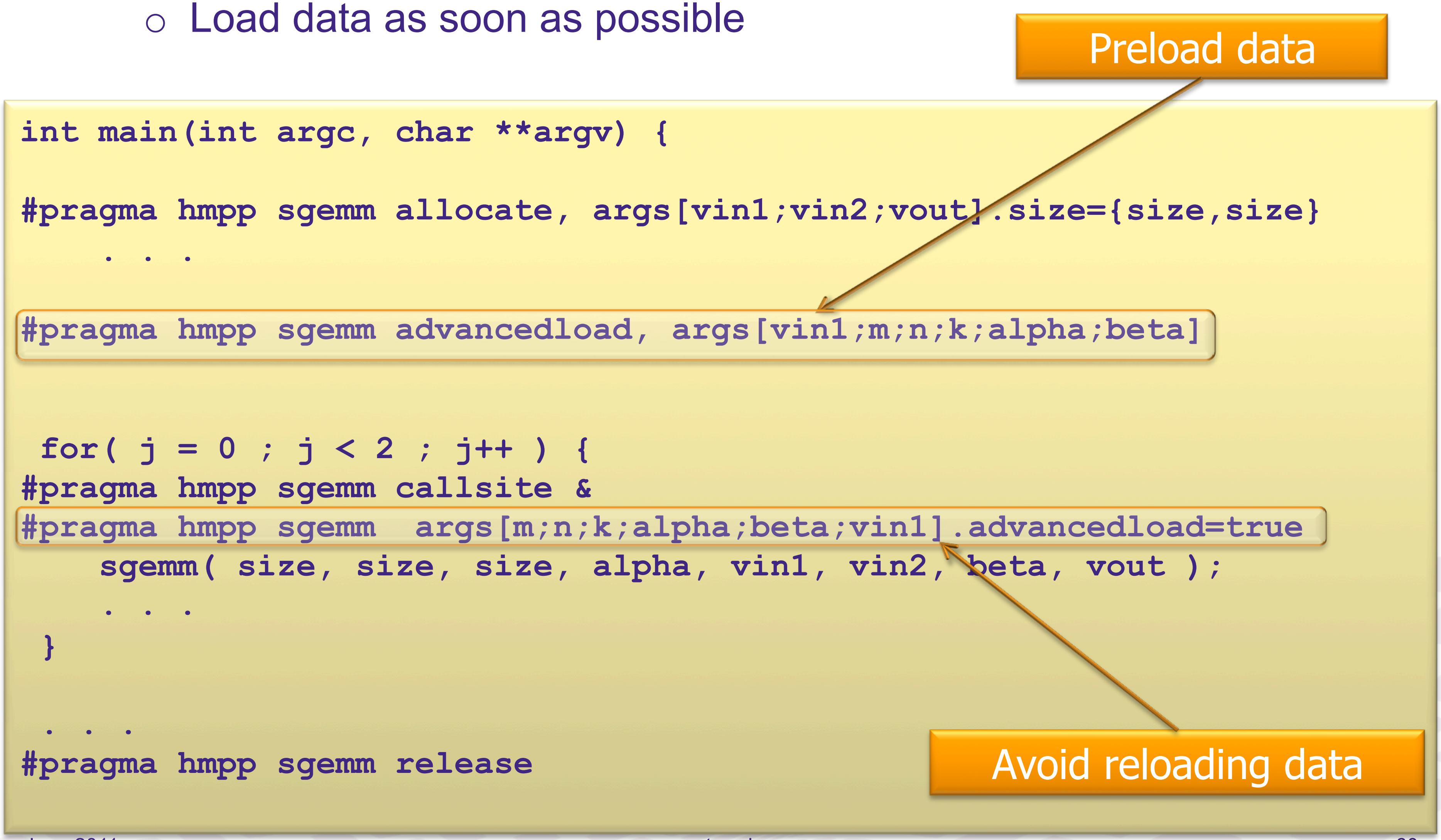
- Hybrid code performance is very sensitive to the amount of CPU-GPU data transfers
  - PCIx bus is a serious bottleneck (< 10 GBs vs 150 GBs)

- 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

Reducing Data Transfers Occurrences

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



```
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
```

Preload data

Avoid reloading data

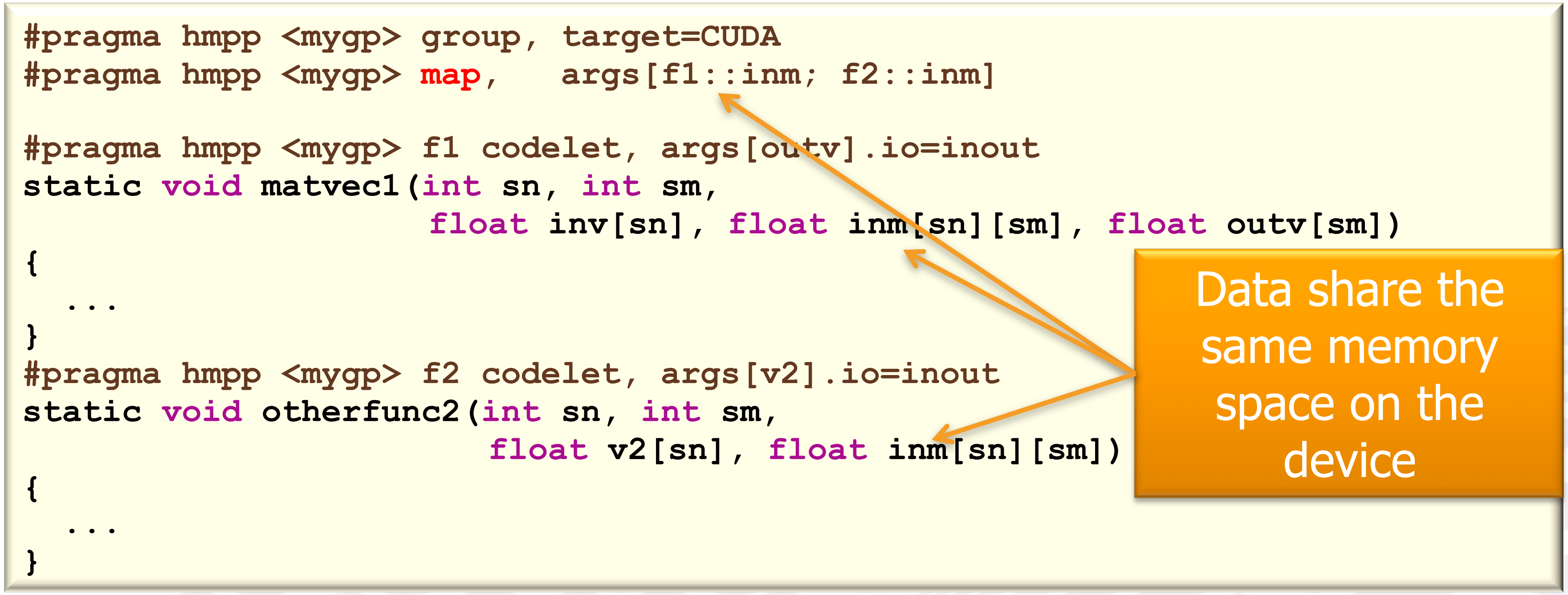
Sharing Data Between Codelets with Resident Data

- 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;
    }
  }
}
```

Sharing Data Between Codelets using Argument Mapping

- Map arguments of different functions in same GPU memory location
  - Save data transfers
  - Save GPU memory space
  - Cannot be used if the two arguments must have simultaneous live data



```
#pragma hmpp <mygp> group, target=CUDA
#pragma hmpp <mygp> map,   args[f1::inm; f2::inm]

#pragma hmpp <mygp> f1 codelet, args[outv].io=inout
static void matvec1(int sn, int sm,
                    float inv[sn], float inm[sn][sm], float outv[sm])
{
  ...
}
#pragma hmpp <mygp> f2 codelet, args[v2].io=inout
static void otherfunc2(int sn, int sm,
                       float v2[sn], float inm[sn][sm])
{
  ...
}
```

Data share the same memory space on the device

Some Other Techniques to Look At

- Partial data transfers
- Moving CPU serial code to GPU
- …

Tuning GPU Kernels

- 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

Improving Memory Access Coalescing and Grid Block Size Tuning (NVIDIA Tesla 1060)

```
# pragma hmppcg grid blocksize 16 X 16
# pragma hmppcg parallel
   for (i = 1; i < m-1; ++i){
      # pragma hmppcg parallel
       for (j = 1; j < n -1; ++j) {
          # pragma hmppcg parallel
           for (k = 0; k < p; ++k){
               B[i][j][k] = c11 * A[i - 1][j - 1][k]  +  c12 * A[i + 0][j -1][k]  +] . . . ;
           }
       }
    }
```

- Questions to answer
  - What is the best IJK loop order (6 possibilities)?
  - What is the best thread block configuration (24 tested here, 8x1 to 512x1)?

Example
of performance
for one data size

	JIK	JKI	IJK	IKJ	KJI	KIJ
Min Perf	0,8	5	0,1	10	0,1	0,8
Max Perf	1	14,4	0,2	15,9	0,5	3

Example (DP) of Impact of the Various Tuning Steps

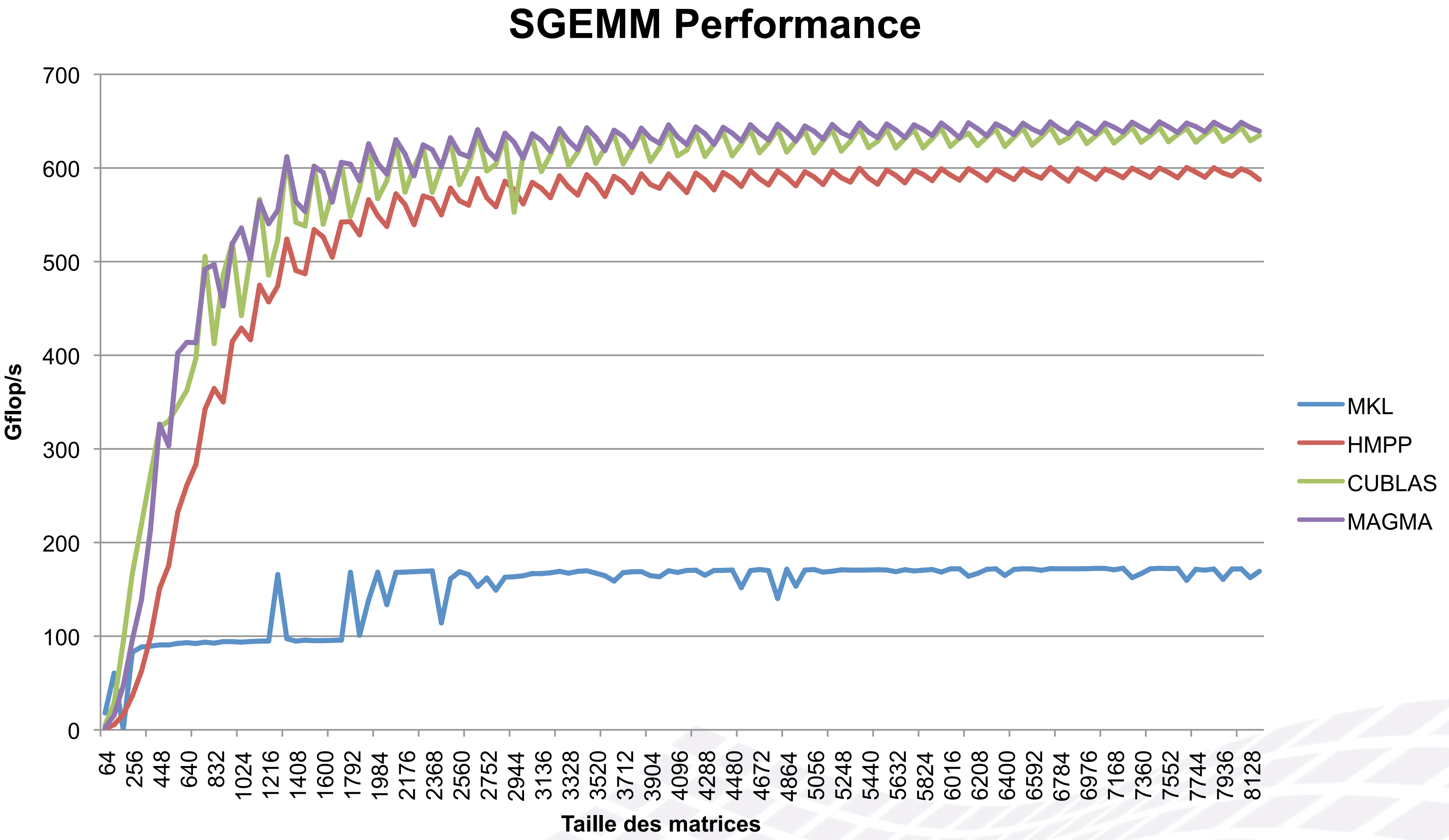
- Original loop nest = 1.0 Gflops
- Improved coalescing (change loop order) = 15.5 Gflops
- Exploit SM shared memories = 39 Gflops
- Better register usage (unroll & jam) = 45.6 Gflops

```
DO j=1+2,n-2
  DO i=1+2,n-2
     DO k=1,10
      B(i,j,k) = &
          & c11*A(i-2,j-2,k) + c21*A(i-1,j-2,k) + c31*A(i+0,j-2,k) + c41*A(i+1,j-2,k) +  c51*A(i+2,j-2,k) + &
          & c12*A(i-2,j-1,k) + c22*A(i-1,j-1,k) + c32*A(i+0,j-1,k) + c42*A(i+1,j-1,k) +  c52*A(i+2,j-1,k) + &
          & c13*A(i-2,j+0,k) + c23*A(i-1,j+0,k) + c33*A(i+0,j+0,k) + c43*A(i+1,j+0,k) +  c53*A(i+2,j+0,k) + &
          & c14*A(i-2,j+1,k) + c24*A(i-1,j+1,k) + c34*A(i+0,j+1,k) + c44*A(i+1,j+1,k) +  c54*A(i+2,j+1,k) + &
          & c15*A(i-2,j+2,k) + c25*A(i-1,j+2,k) + c35*A(i+0,j+2,k) + c45*A(i+1,j+2,k) +  c55*A(i+2,j+2,k)
     ENDDO
  END DO
END DO
```

Unroll & Jam Example

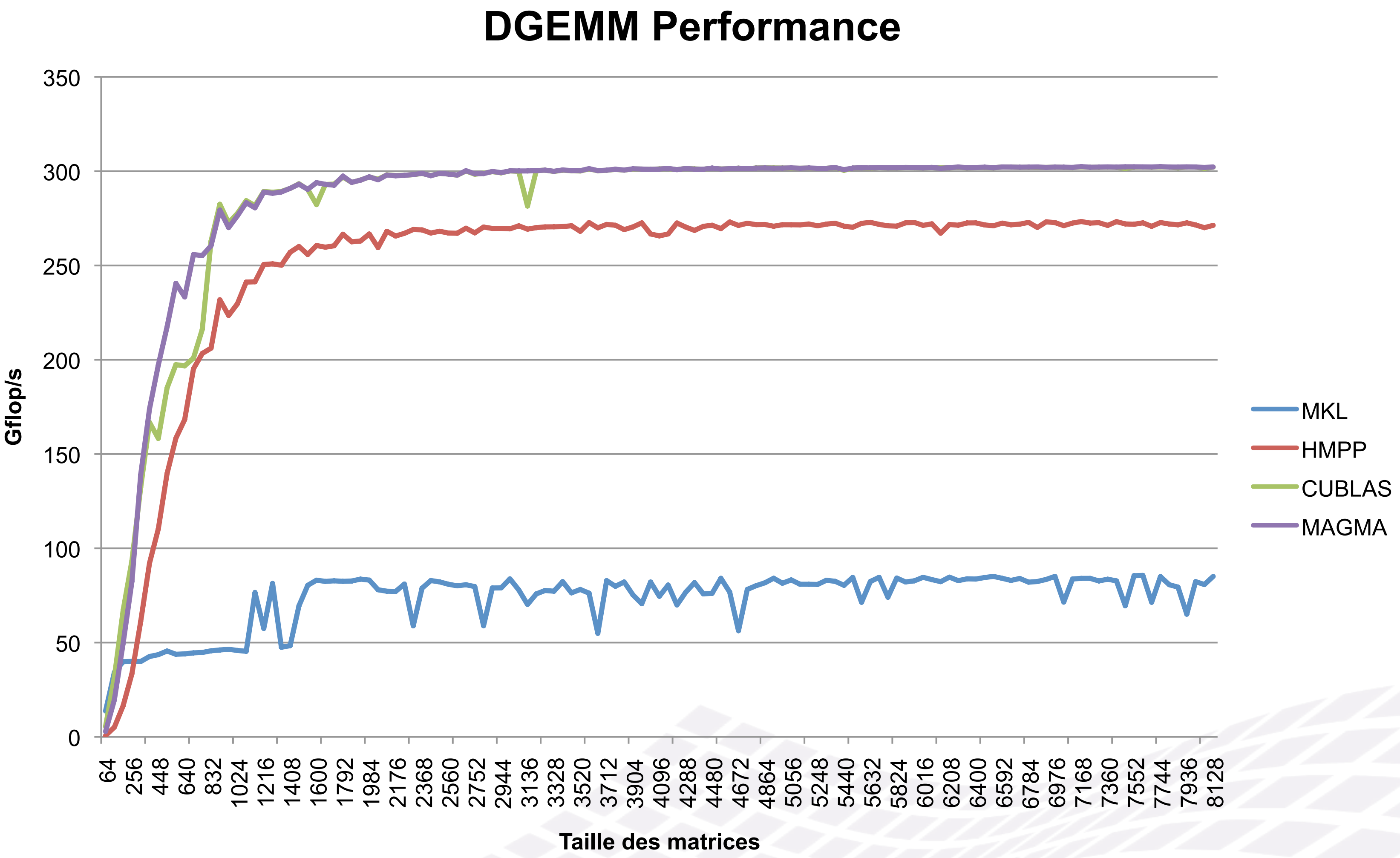
```
#pragma hmppcg unroll(4), jam(2), noremainder
 for( j = 0 ; j < p ; j++ ) {
  #pragma hmppcg unroll(4), split, noremainder
  for( i = 0 ; i < m ; i++ ) {
   double prod = 0.0;
   double v1a,v2a ;
   k=0 ;
   v1a = vin1[k][i] ;
   v2a = vin2[j][k] ;
   for( k = 1 ; k < n ; k++ ) {
    prod += v1a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

HMPP BLAS Performance on NVIDIA Fermi



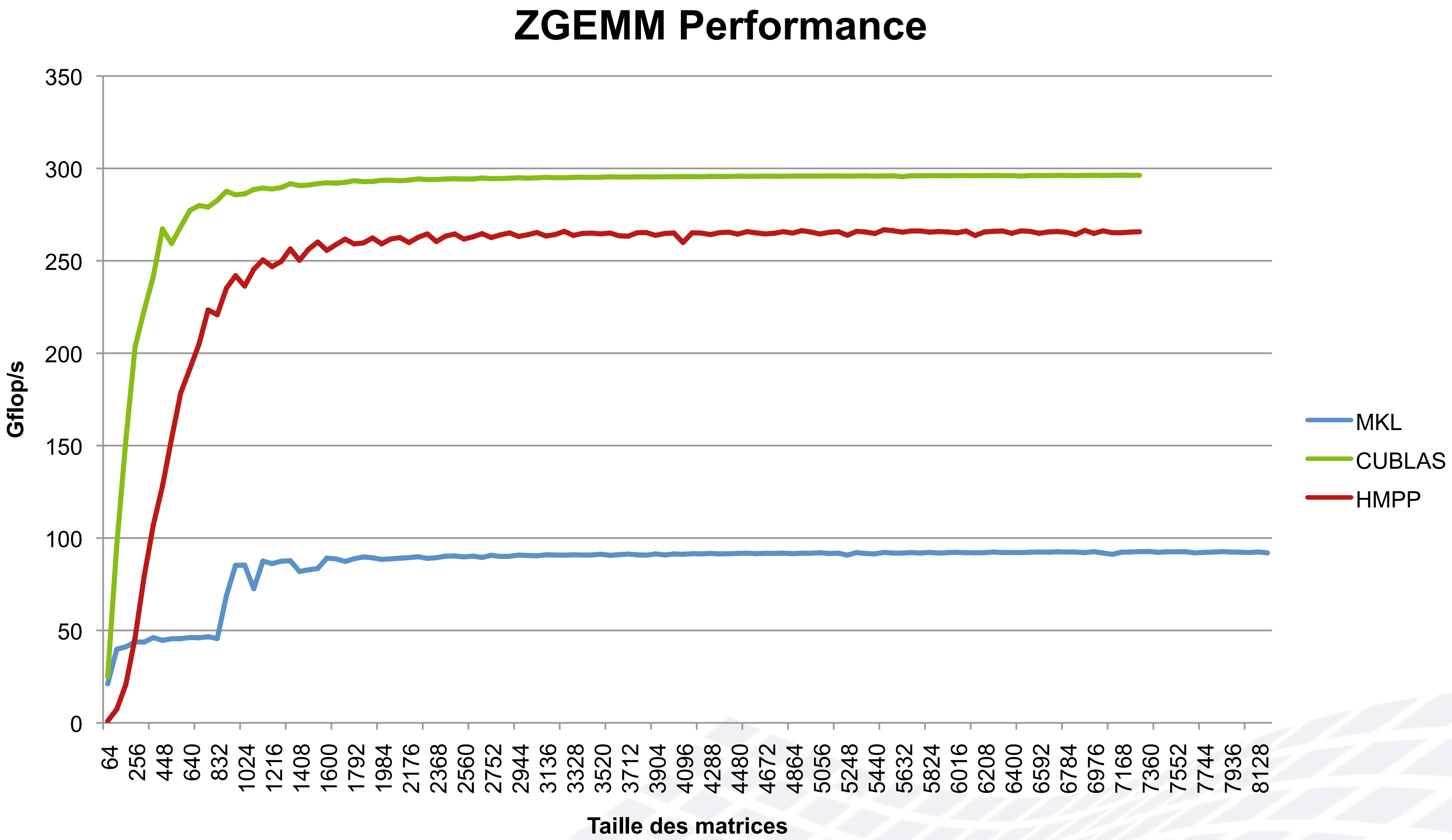
2 x Intel(R) Xeon(R) X5560 @ 2.80GHz (8 cores) - MKL
NVidia Tesla C2050, ECC activated – HMPP, CUBLAS, MAGMA

HMPP BLAS Performance on NVIDIA Fermi



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



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

Other Available Features in HMPP

- Directives for loop tiling, …
- Directives to handle reductions
- Directives to use shared/local memory (for fast communication within thread blocks)
- Seamless use of available libraries (CuBlas, ACML, …) or hand-written optimized kernels

- See HMPP cookbook for more details

Hand-written/Library Codelets in HMPP

- Take advantage of existing libraries available for a device

- Take advantage of proprietary codes directly developed in some target language

- Thanks to the codelet HMPP approach, details of implementations are hidden
  - At execution time, calls to the external library will be directly performed by the runtime based on the execution context

Reduction in HMPP

- Reduction operations are expressed under the form of clause of the parallel hmppcg directive

```
#pragma hmppcg parallel, reduce(max:tmp)
  for (i=1;i  <= n;i++) {
#pragma hmppcg parallel
#pragma hmppcg unroll(16),guarded, split
    for (j=1; j <= m;j++) {
      tmp= fmax(tmp,newa[i][j]);
    }
  }
```

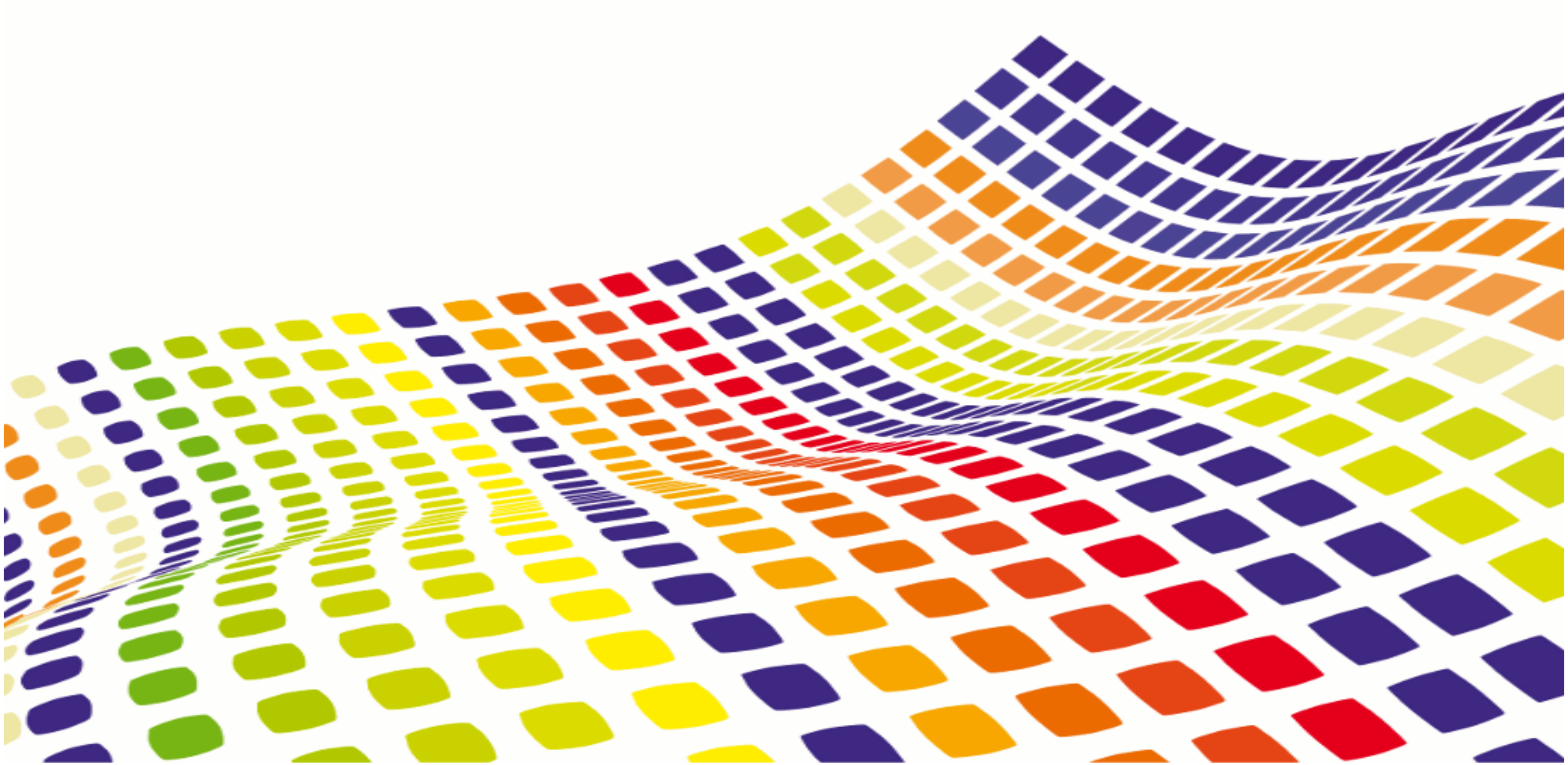
- Without this clause, the parallel execution of a loop with such an operation could lead to a wrong result.

HMPP for Future Manycores

- Current HMPP
  - Agnostic directive based style
  - Target stream parallelism on accelerators
  - High level expression of stream oriented parallelism
  - Mostly deals with one GPU per threads, no GPU sharing
  - Oriented toward performance and device memory saving

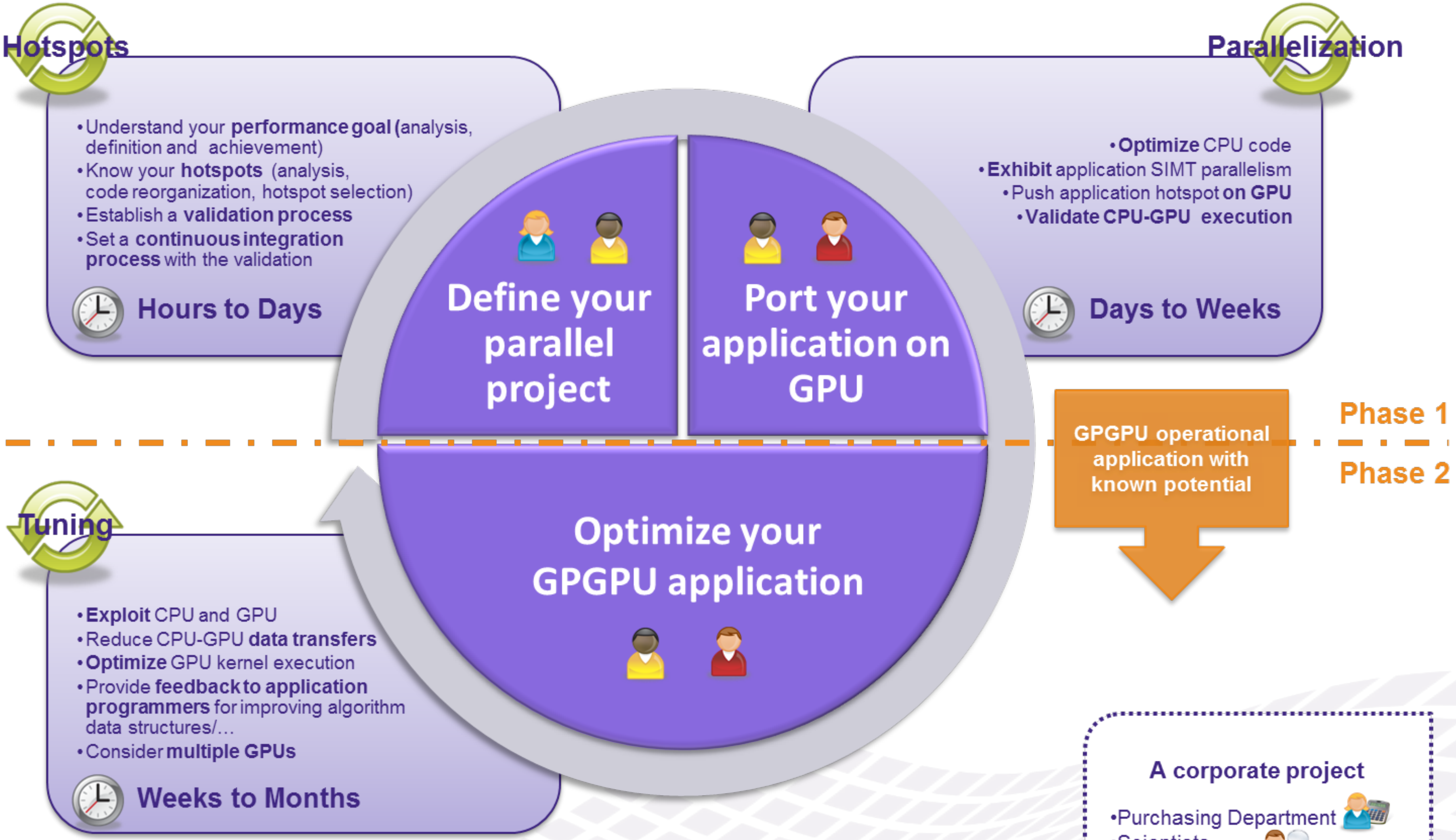
- Next HMPP
  - Agnostic directive based with more API for expert programmers
    - Adaptive programming
  - High level expression of stream oriented parallelism
  - Target accelerators and CPU cores
  - Handles multiple GPUs, data distribution
  - Easier handling of data management between CPU and GPU

A Methodology for Code Migration



Methodology to port applications

CAPS



Hotspots

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

Hours to Days

Parallelization

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

Days to Weeks

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**

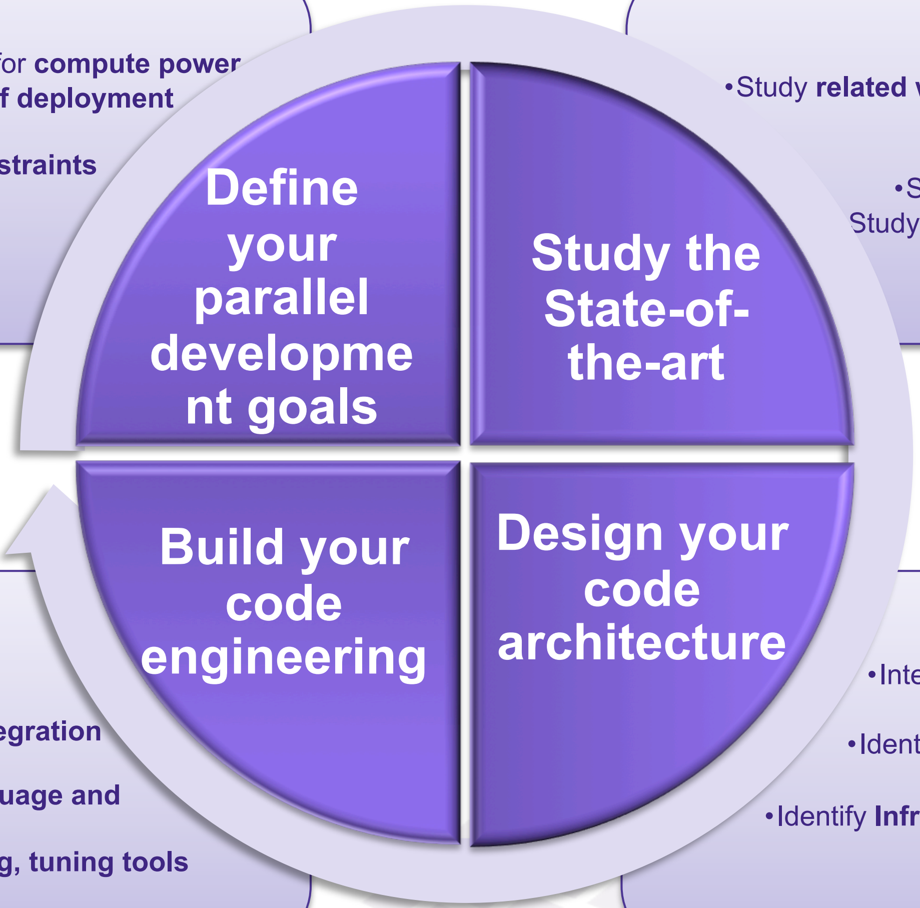
Weeks to Months

Define your parallel project
Port your application on GPU
Optimize your GPGPU application
Phase 1
Phase 2
GPGPU operational application with known potential

A corporate project

- Purchasing Department
- Scientists
- IT Department

Methodology for New Applications (Draft)



Define your parallel development goals

- Determine your needs for **compute power**
- Determine the **scope of deployment** (Linux, Windows, …)
- Fix your **licensing constraints**

Study the State-of-the-art

- Study **related work** on parallel algorithm addressing the issues
- Study existing **Libraries**
- Study **open source codes**

Study existing **data structures**

Design your code architecture

- Integrate **debugging issues**
- Define **coding rules**
- Identify **domain Components**
- Specify **interfaces**
- Identify **Infrastructure Components (building blocks)**

Build your code engineering

- Set up **continuous integration systems**
- Decide for a **base language and parallel coding API**
- **Set tracing, debugging, tuning tools**
- **Code the application**

Go / No Go

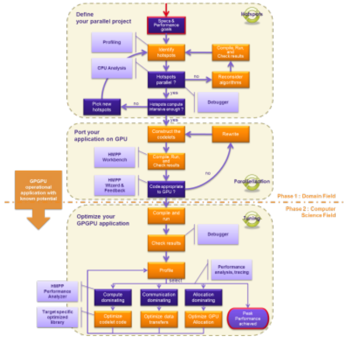
Go

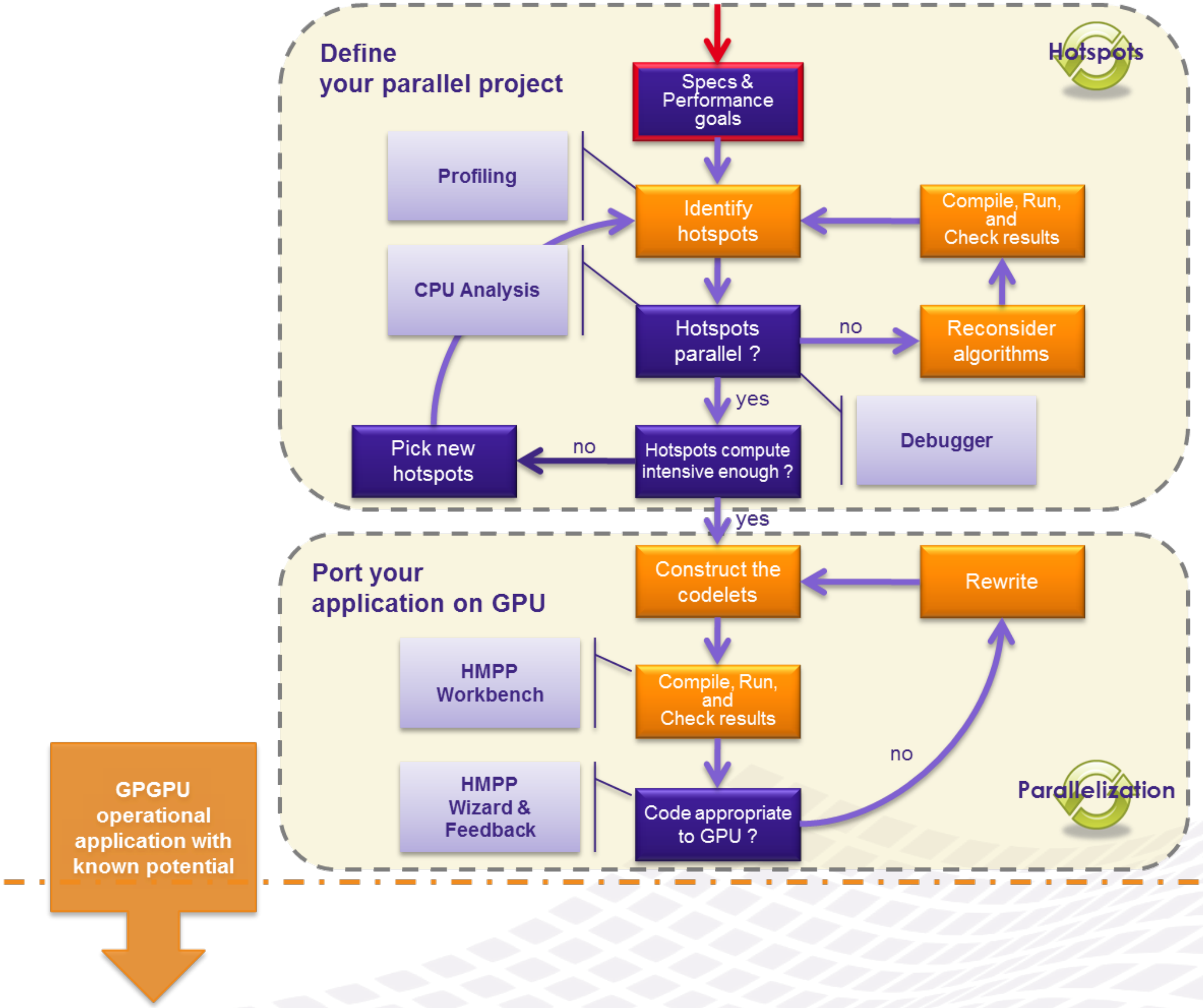
- Dense hotspot
- Fast kernels
- Low CPU-GPU data transfers
- Prepare to manycore parallelism

No Go

- Flat profile
- Slow GPU kernels (i.e. no speedup to be expected)
- Binary exact CPU-GPU results (cannot validate execution)
- Memory space needed

Phase 1 (details)





Define your parallel project

Hotspots

Specs & Performance goals

Profiling

Identify hotspots

Compile, Run, and Check results

CPU Analysis

Hotspots parallel ?

no

Reconsider algorithms

yes

Debugger

Pick new hotspots

no

Hotspots compute intensive enough ?

yes

Port your application on GPU

Construct the codelets

Rewrite

HMPP Workbench

Compile, Run, and Check results

no

HMPP Wizard & Feedback

Code appropriate to GPU ?

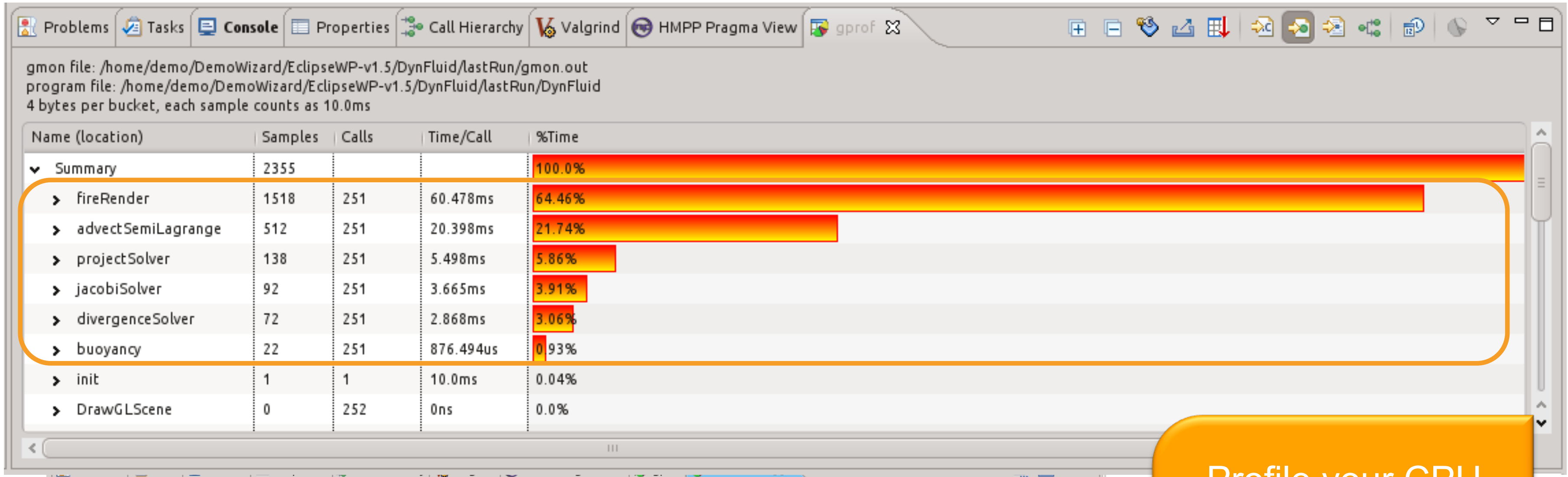
Parallelization

GPGPU operational application with known potential

Phase 1 : Domain Field

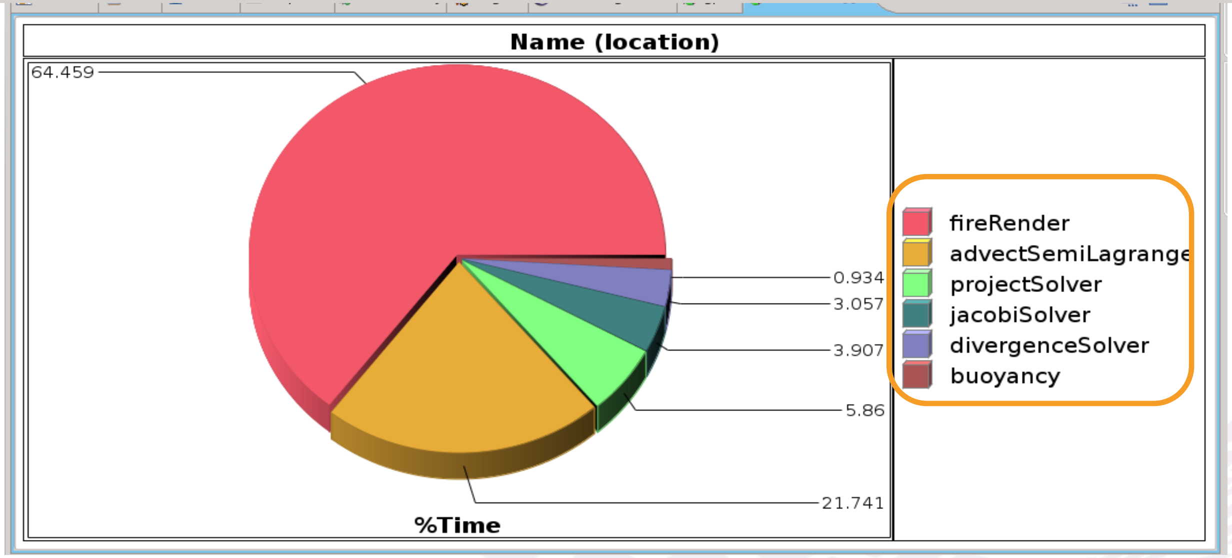
Phase 2 : Computer Science Field

Focus on Hotspots



```
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%

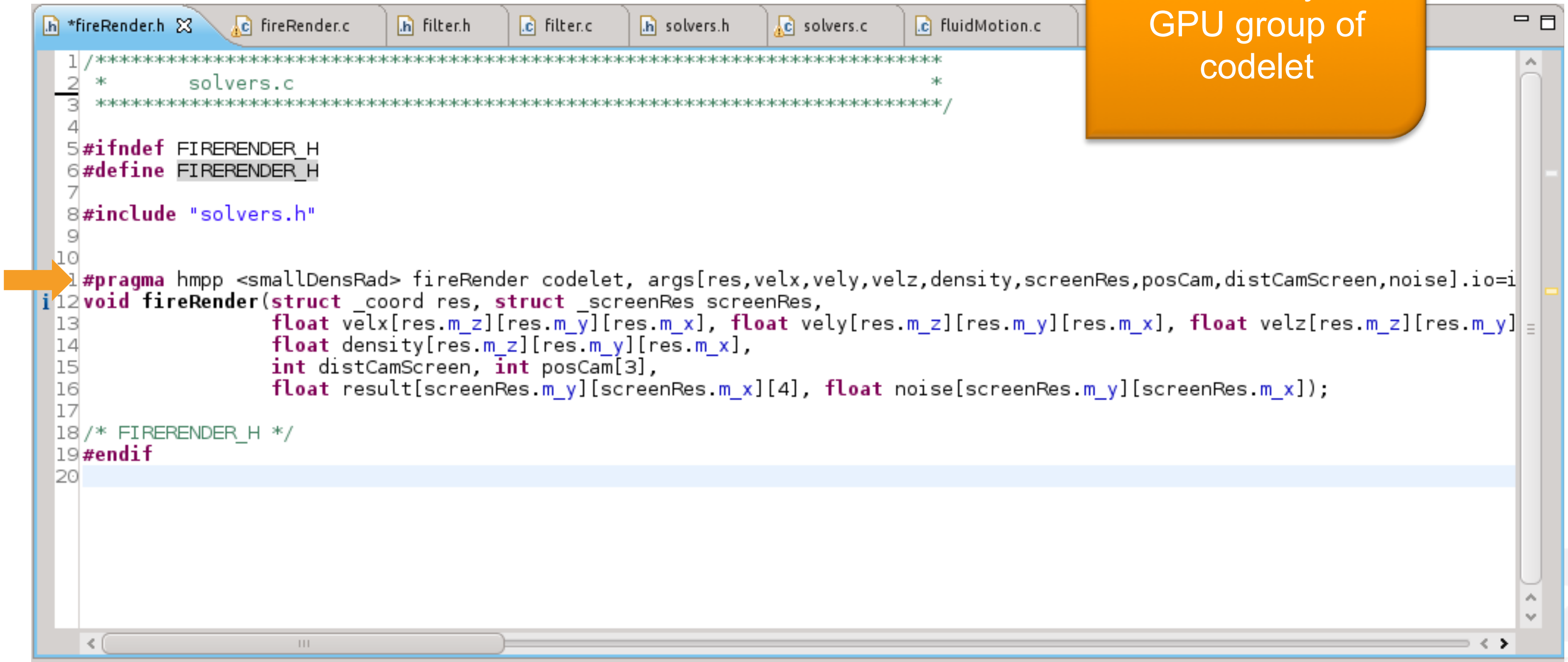


Profile your CPU application

Build a coherent kernel set

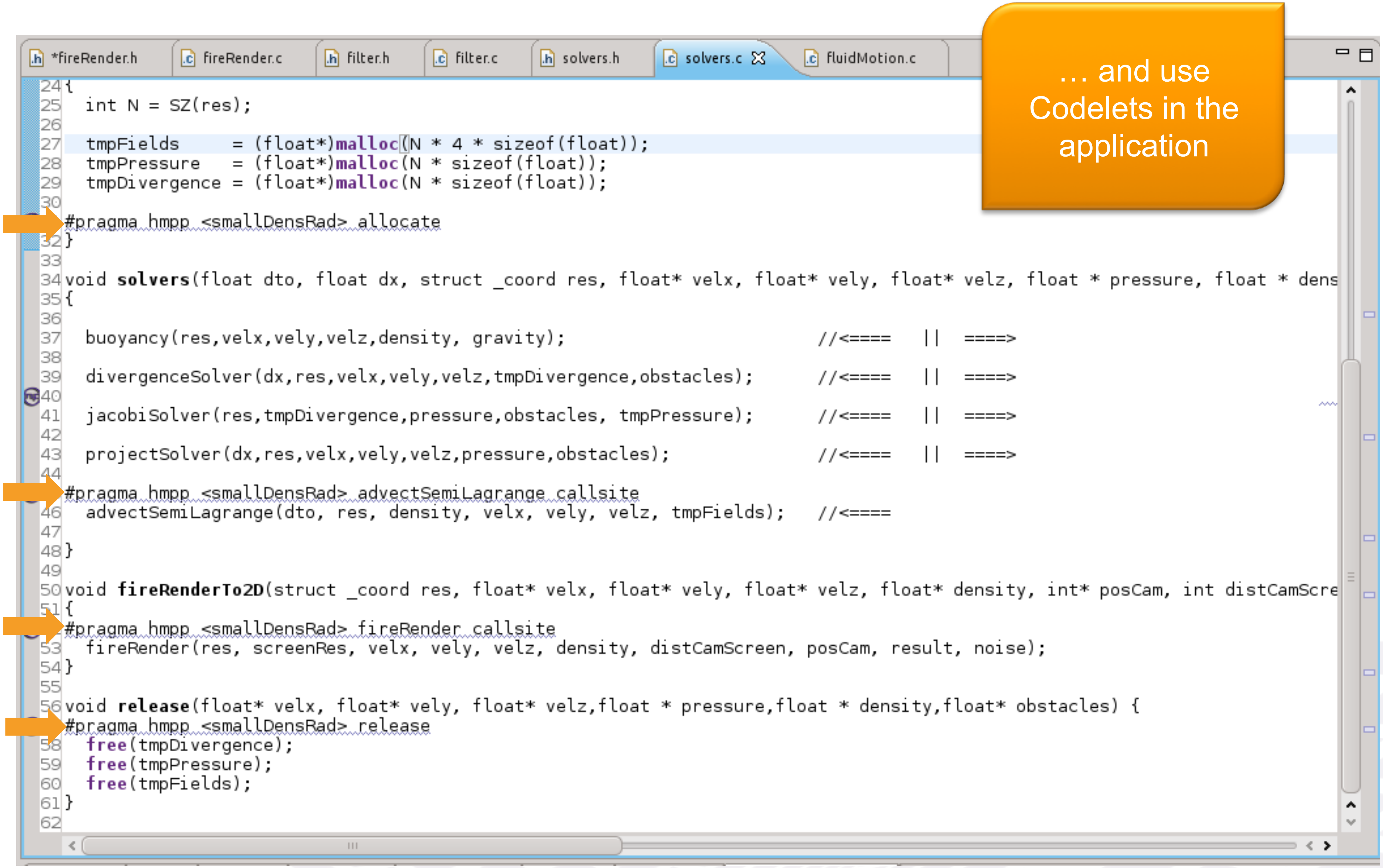
Build the GPU Computation with HMPP Directives (1)

Construct your GPU group of codelet



```
/************************************************************************
 *       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

```

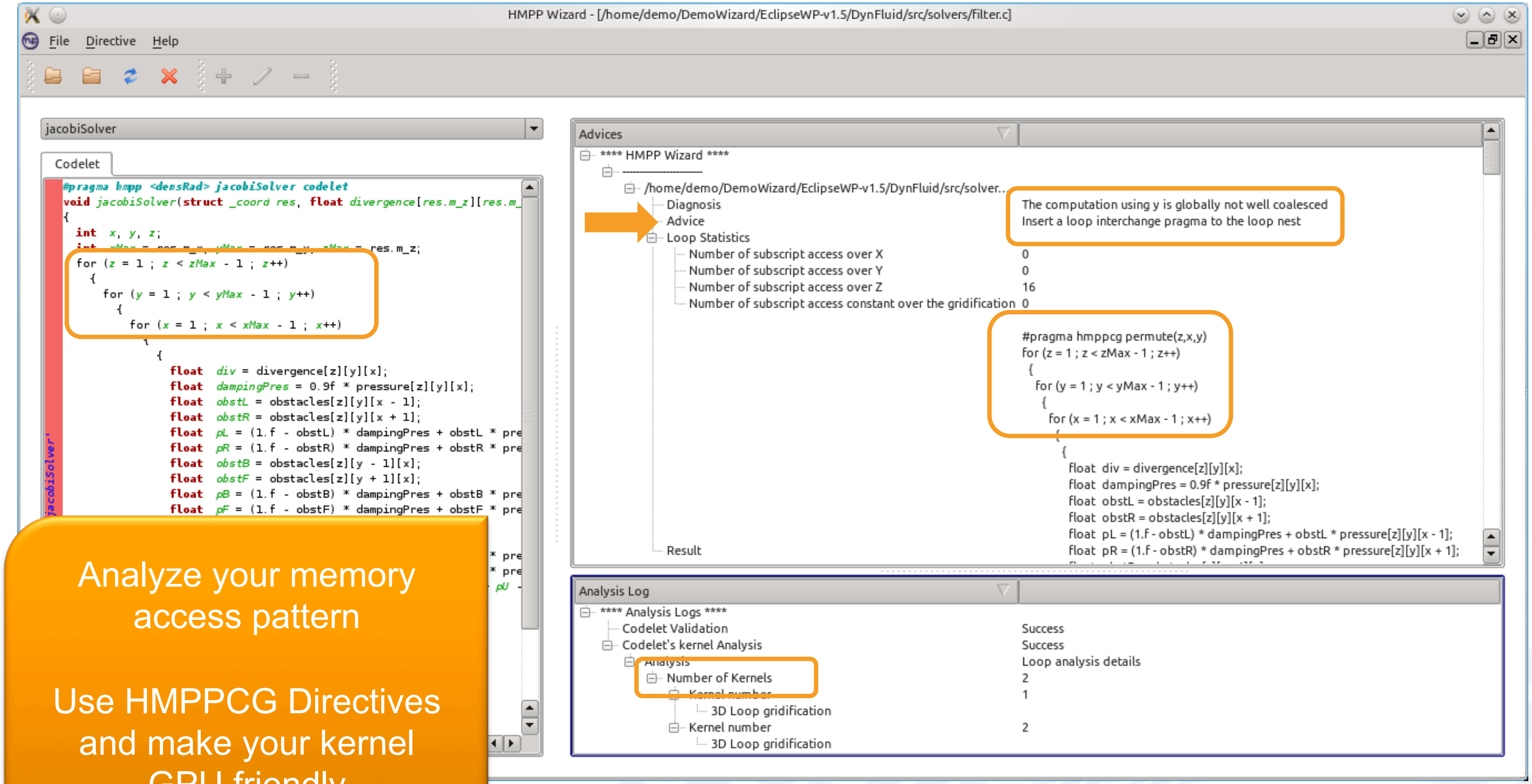
Build the GPU Computation with HMPP Directives (2)



```
{
  int N = SZ(res);

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

#pragma hmpp <smallDensRad> allocate
}

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

  buoyancy(res,velx,vely,velz,density, gravity);                        //<====   ||  ====>

  divergenceSolver(dx,res,velx,vely,velz,tmpDivergence,obstacles);      //<====   ||  ====>

  jacobiSolver(res,tmpDivergence,pressure,obstacles, tmpPressure);      //<====   ||  ====>

  projectSolver(dx,res,velx,vely,velz,pressure,obstacles);              //<====   ||  ====>

#pragma hmpp <smallDensRad> advectSemiLagrange callsite
  advectSemiLagrange(dto, res, density, velx, vely, velz, tmpFields);   //<====

}

void fireRenderTo2D(struct _coord res, float* velx, float* vely, float* velz, float* density, int* posCam, int distCamScre
{
#pragma hmpp <smallDensRad> fireRender callsite
  fireRender(res, screenRes, velx, vely, velz, density, distCamScreen, posCam, result, noise);
}

void release(float* velx, float* vely, float* velz,float * pressure,float * density,float* obstacles) {
#pragma hmpp <smallDensRad> release
  free(tmpDivergence);
  free(tmpPressure);
  free(tmpFields);
}
```

… and use Codelets in the application

Tune the 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
jacobiSolver'

```
#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
```

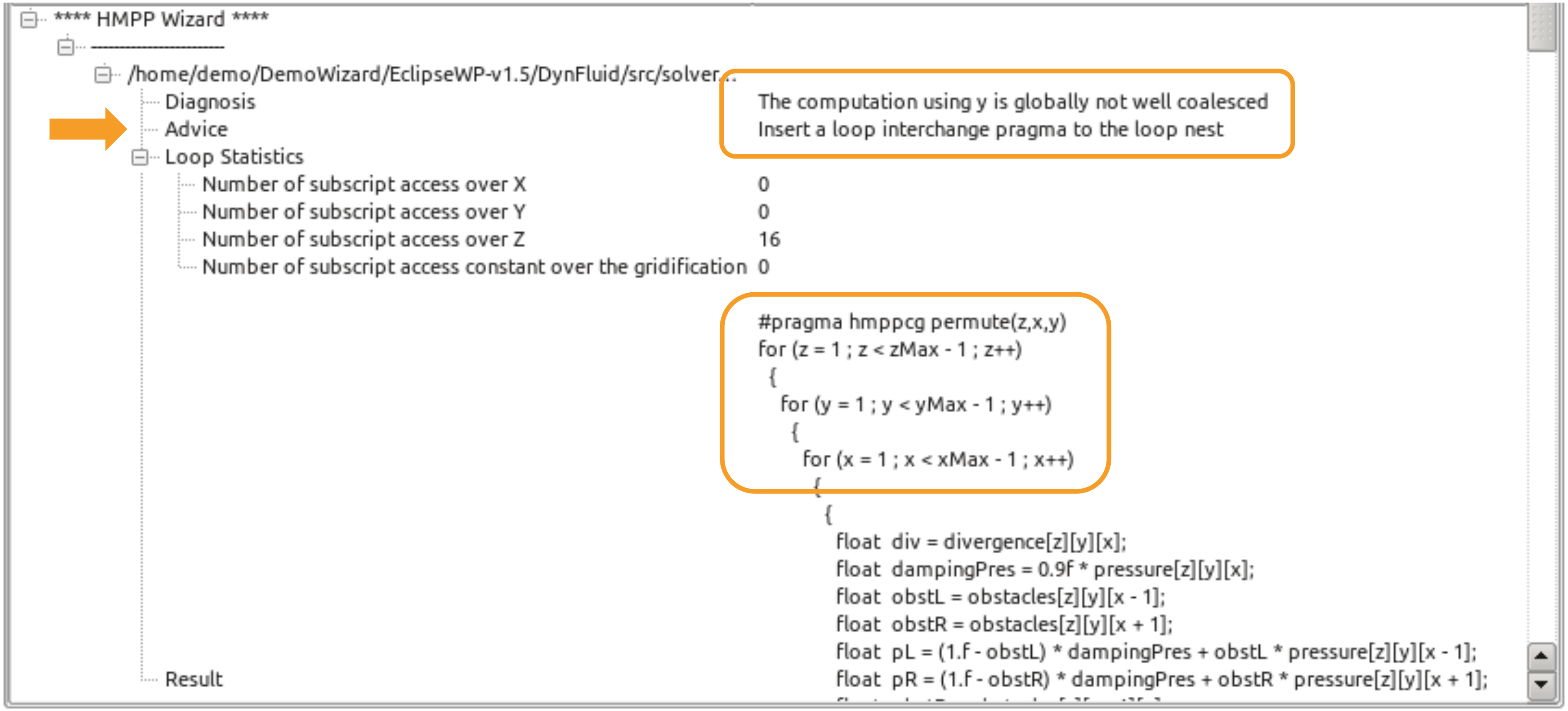
```
Advices
**** HMPP Wizard ****
  ----------------------
    /home/demo/DemoWizard/EclipseWP-v1.5/DynFluid/src/solver...
      Diagnosis                                              The computation using y is globally not well coalesced
      Advice                                                 Insert 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];
```

```
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

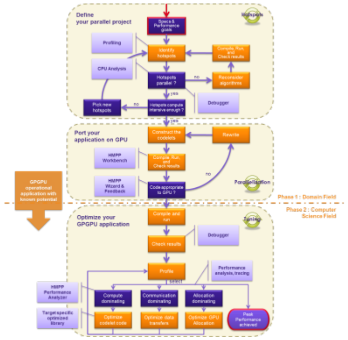
Use HMPPCG Directives and make your kernel GPU friendly

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



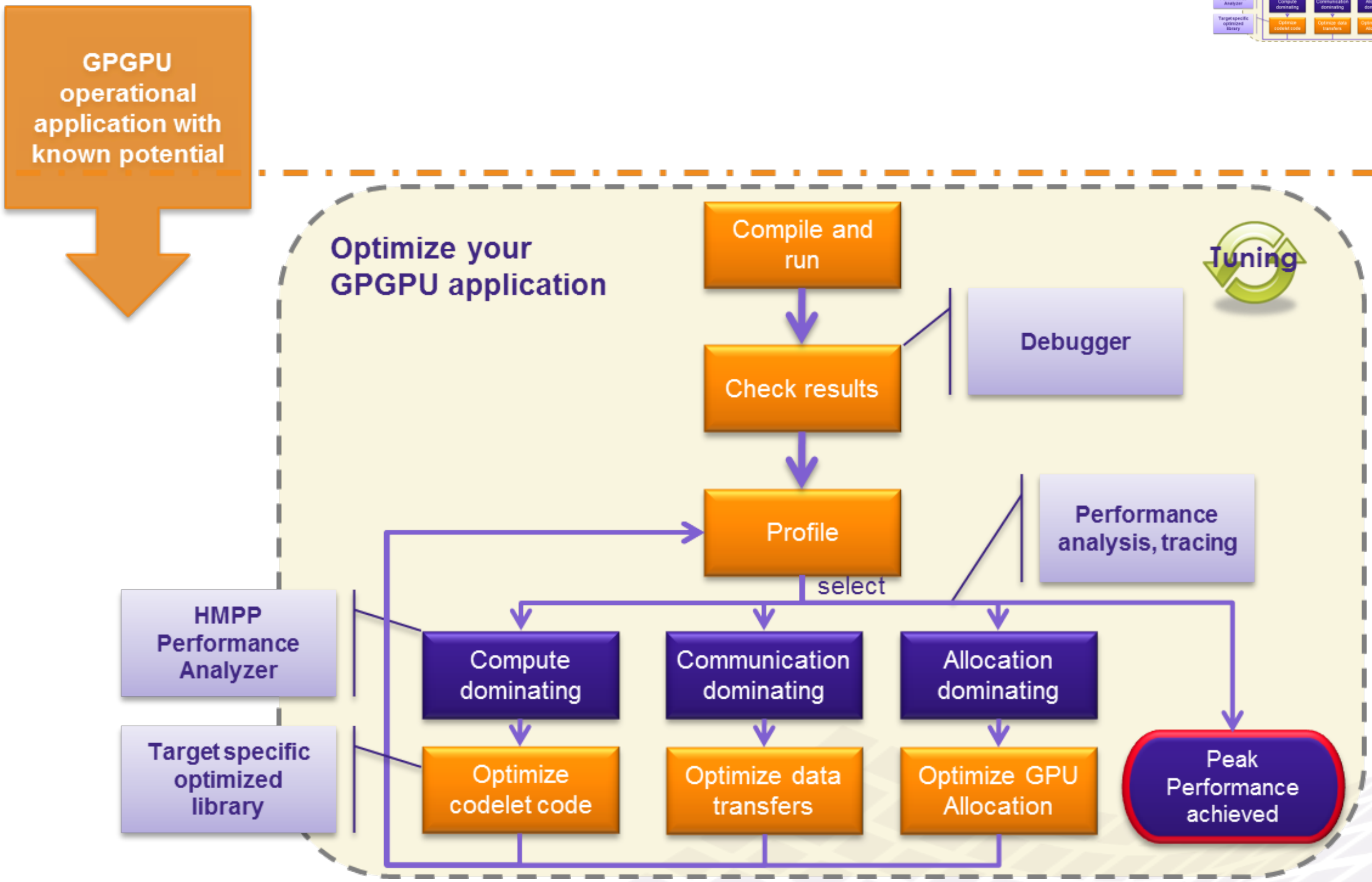
```
**** HMPP Wizard ****
  --------------------
    /home/demo/DemoWizard/EclipseWP-v1.5/DynFluid/src/solver...
      Diagnosis                                                    The computation using y is globally not well coalesced
      Advice                                                       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];
```

Phase 2 (details)



Phase 1 : Domain Field

Phase 2 : Computer Science Field



GPGPU operational application with known potential

Optimize your GPGPU application

Tuning

Compile and run

Check results

Debugger

Profile

Performance analysis, tracing

select

HMPP Performance Analyzer

Target specific optimized library

Compute dominating

Communication dominating

Allocation dominating

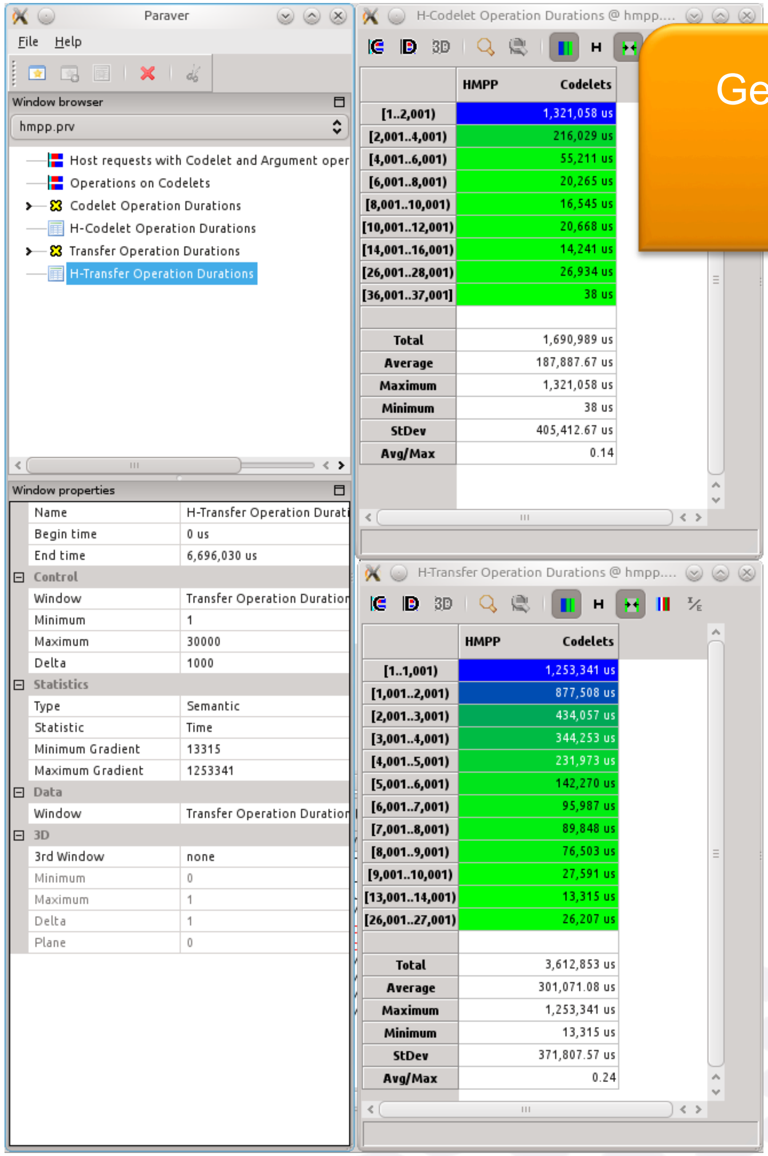
Optimize codelet code

Optimize data transfers

Optimize GPU Allocation

Peak Performance achieved

Analyze the GPU Code Efficiency



hmpp.prv

- Host requests with Codelet and Argument oper
- Operations on Codelets
- Codelet Operation Durations
- H-Codelet Operation Durations
- Transfer Operation Durations
- H-Transfer Operation Durations

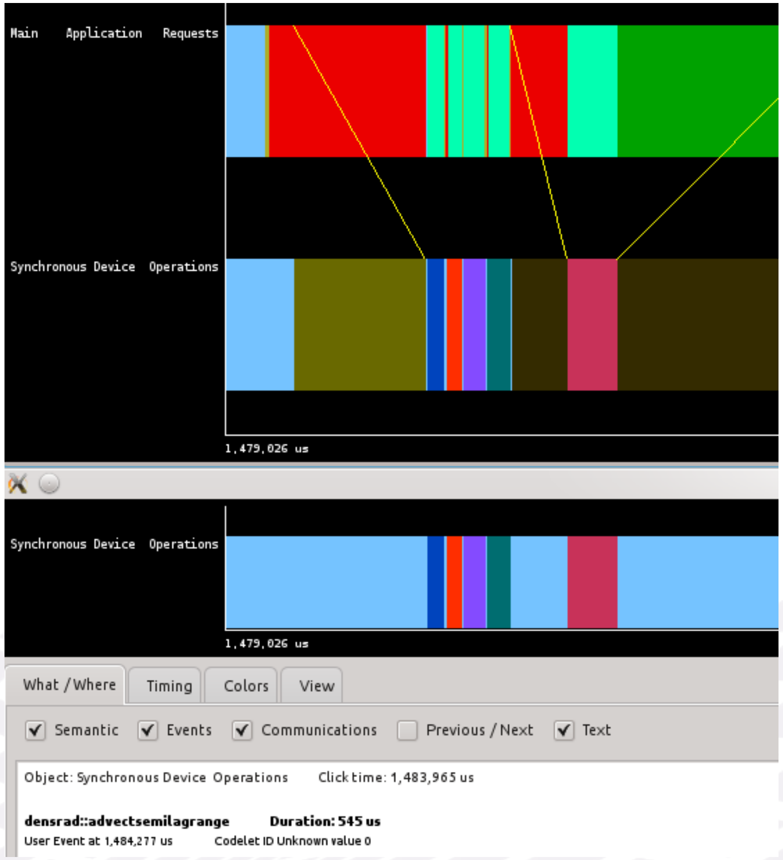
Name	H-Transfer Operation Durati
Begin time	0 us
End time	6,696,030 us
⊟ Control	
Window	Transfer Operation Duration
Minimum	1
Maximum	30000
Delta	1000
⊟ Statistics	
Type	Semantic
Statistic	Time
Minimum Gradient	13315
Maximum Gradient	1253341
⊟ Data	
Window	Transfer Operation Duration
⊟ 3D	
3rd Window	none
Minimum	0
Maximum	1
Delta	1
Plane	0

HMPP	Codelets
[1..2,001)	1,321,058 us
[2,001..4,001)	216,029 us
[4,001..6,001)	55,211 us
[6,001..8,001)	20,265 us
[8,001..10,001)	16,545 us
[10,001..12,001)	20,668 us
[14,001..16,001)	14,241 us
[26,001..28,001)	26,934 us
[36,001..37,001]	38 us
Total	1,690,989 us
Average	187,887.67 us
Maximum	1,321,058 us
Minimum	38 us
StDev	405,412.67 us
Avg/Max	0.14

HMPP	Codelets
[1..1,001)	1,253,341 us
[1,001..2,001)	877,508 us
[2,001..3,001)	434,057 us
[3,001..4,001)	344,253 us
[4,001..5,001)	231,973 us
[5,001..6,001)	142,270 us
[6,001..7,001)	95,987 us
[7,001..8,001)	89,848 us
[8,001..9,001)	76,503 us
[9,001..10,001)	27,591 us
[13,001..14,001)	13,315 us
[26,001..27,001)	26,207 us
Total	3,612,853 us
Average	301,071.08 us
Maximum	1,253,341 us
Minimum	13,315 us
StDev	371,807.57 us
Avg/Max	0.24

Get a precise view of HMPP element behavior

Get statistics on GPU operations



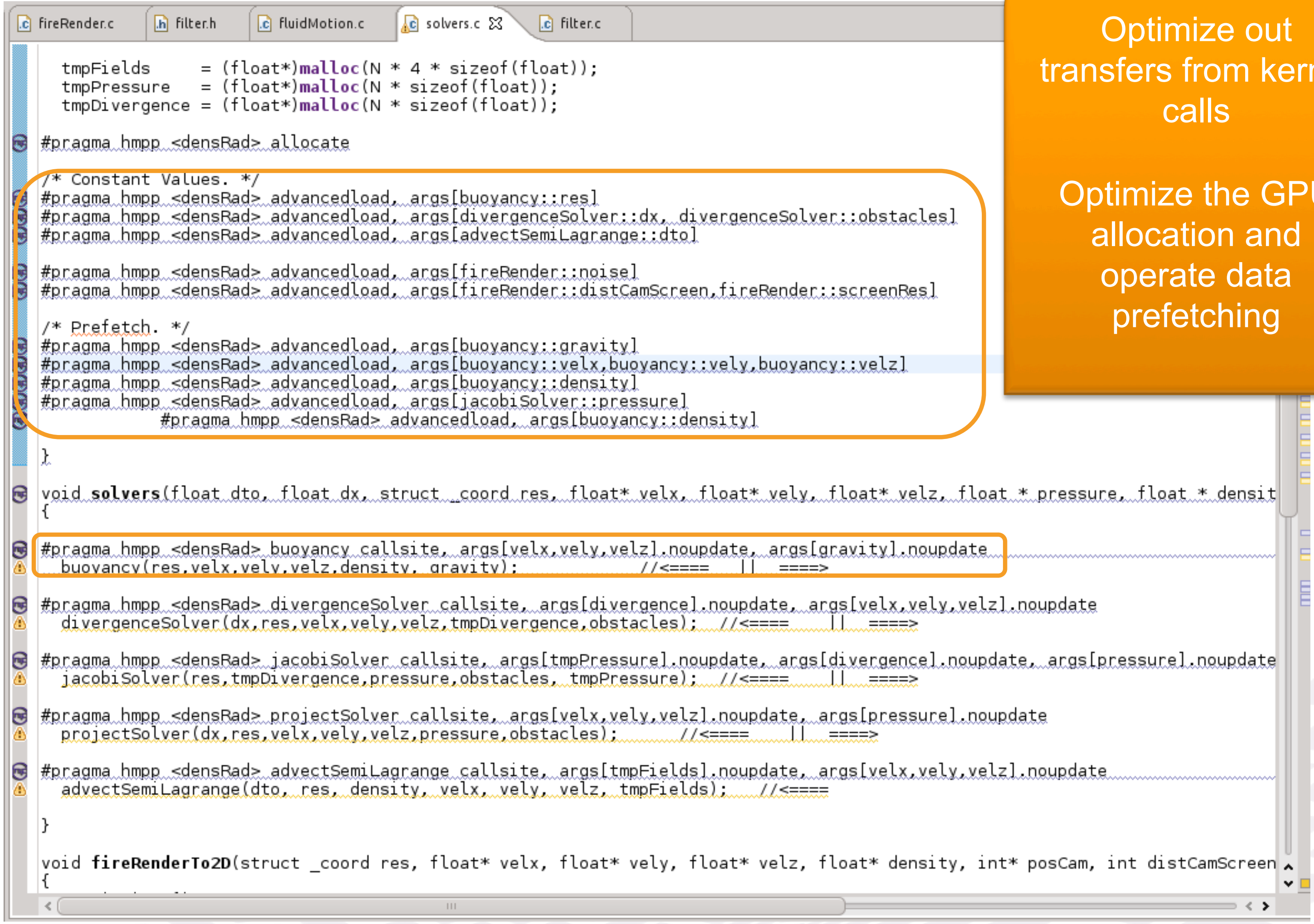
Main Application Requests
Synchronous Device Operations
1,479,026 us
Synchronous Device Operations
1,479,026 us
What / Where
Timing
Colors
View

☑ Semantic ☑ Events ☑ Communications □ Previous / Next ☑ Text

Object: Synchronous Device Operations Click time: 1,483,965 us

densrad::advectsemilagrange **Duration: 545 us**
User Event at 1,484,277 us Codelet ID Unknown value 0

Tune the GPU Execution Integration in the Application with HMPP Directives

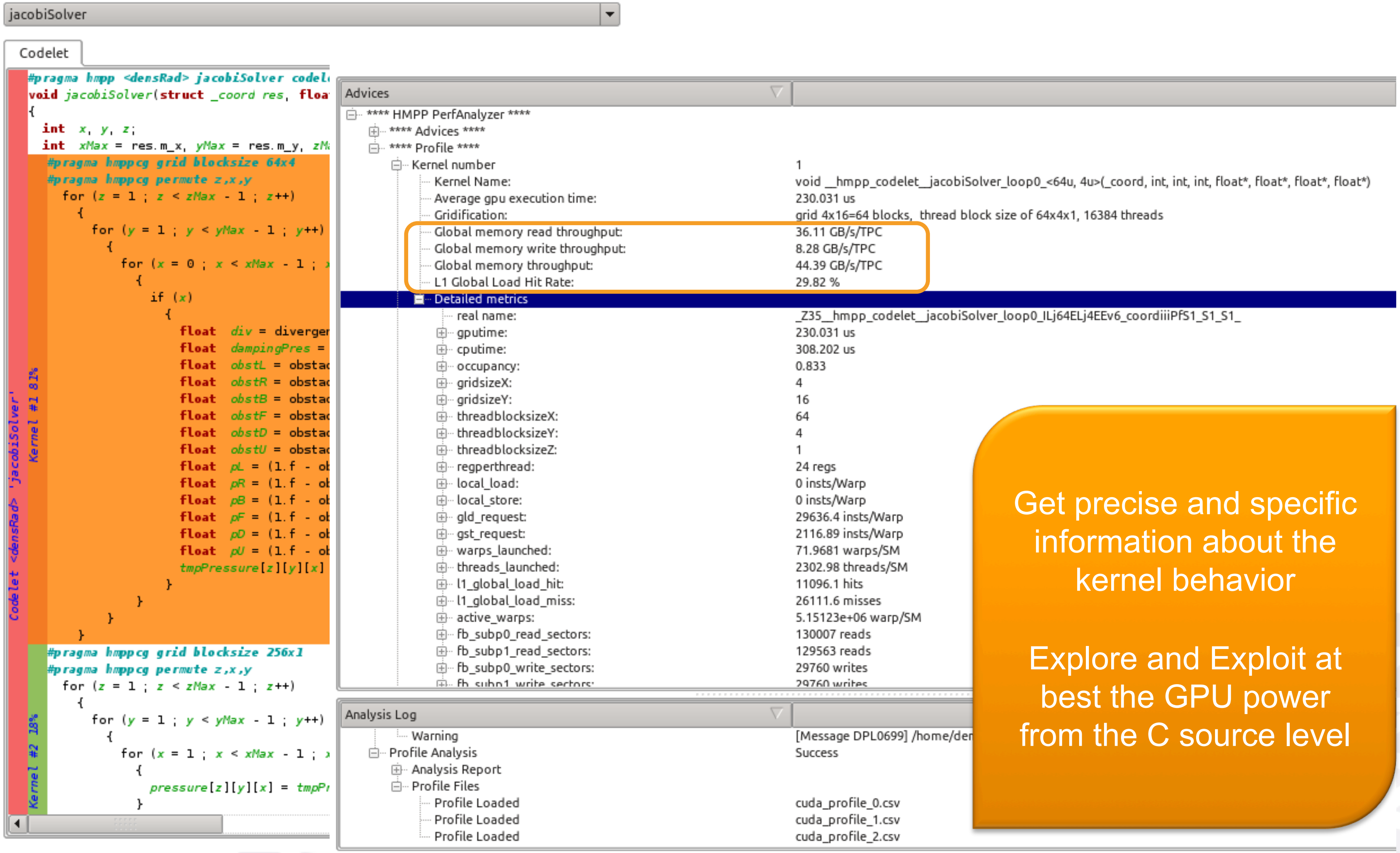


```
  tmpFields     = (float*)malloc(N * 4 * sizeof(float));
  tmpPressure   = (float*)malloc(N * sizeof(float));
  tmpDivergence = (float*)malloc(N * sizeof(float));

#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



```
#pragma hmpp <densRad> jacobiSolver codel
void jacobiSolver(struct _coord res, floa
{
  int  x, y, z;
  int  xMax = res.m_x, yMax = res.m_y, zMa
   #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
              }
```

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
```

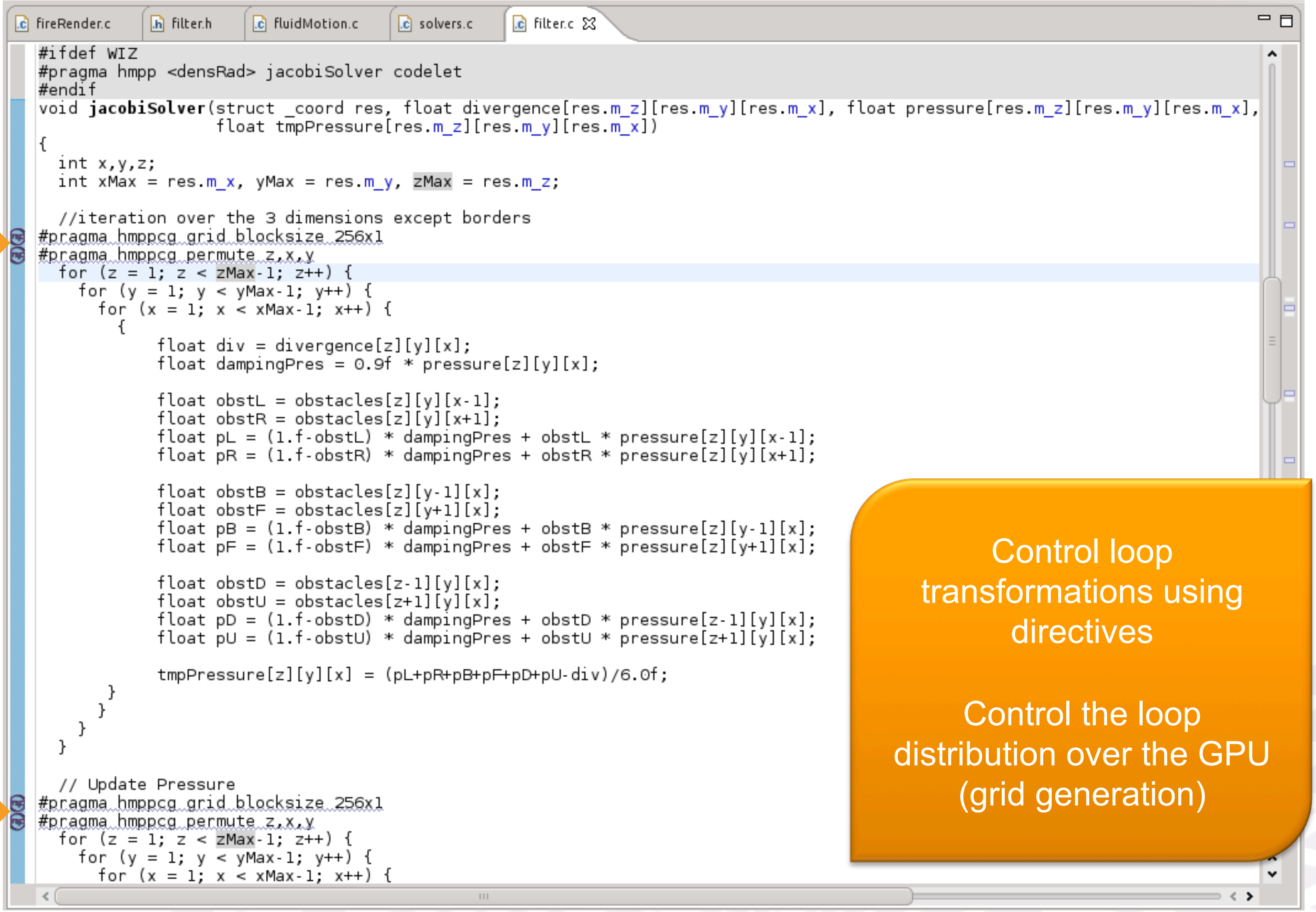
Analysis Log

```
    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



```
#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++) {
```

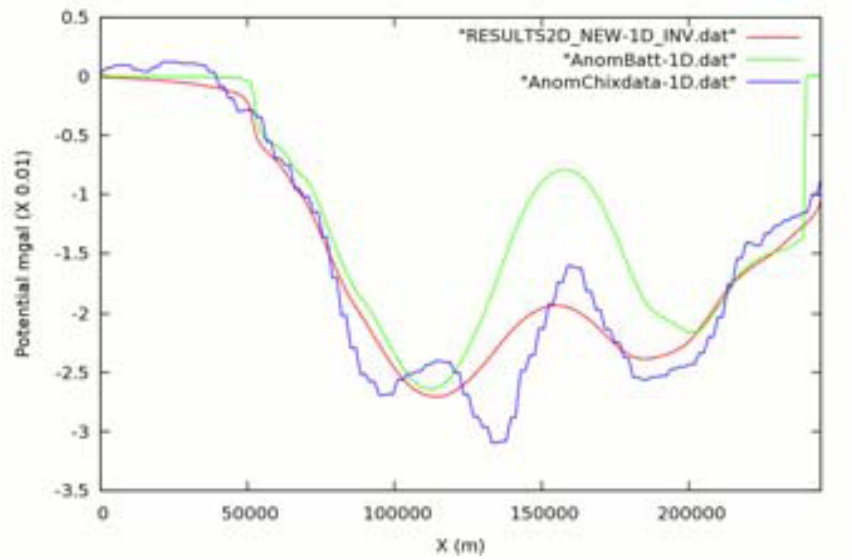
Control loop transformations using directives

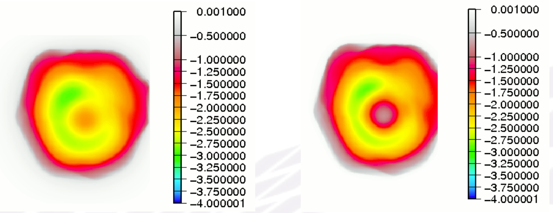
Control the loop distribution over the GPU (grid generation)

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



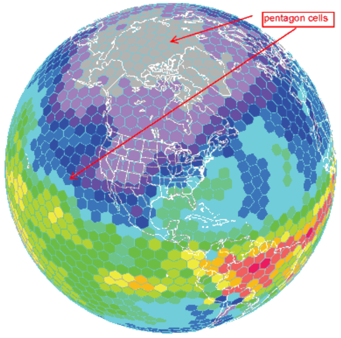


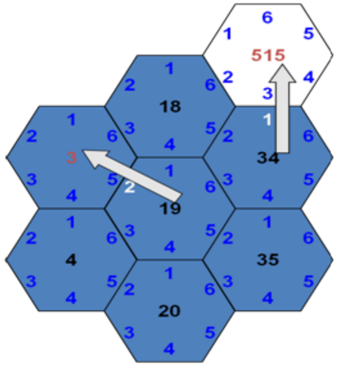
The finite volumes method (left) is more accurate than the analytic solution (right) which over estimates the central peak

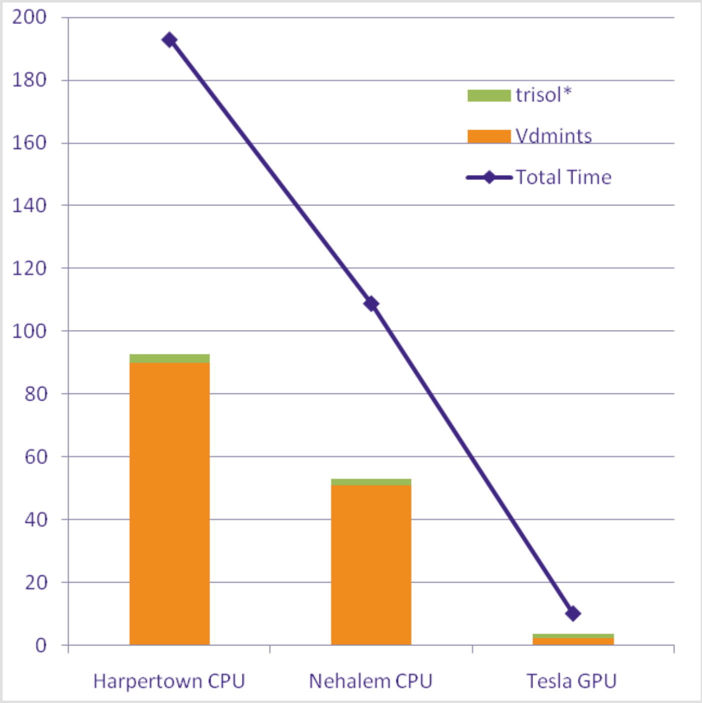
Weather Forecasting

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

- Resource spent
  - 1 man-month
- Size
  - ~1kLoC of C99 (DP)
- CPU Improvement
  - x 4,86
- GPU C2050 improvement
  - x 120 over serial code on Nehalem
- Main porting operation
  - Rethinking algorithm

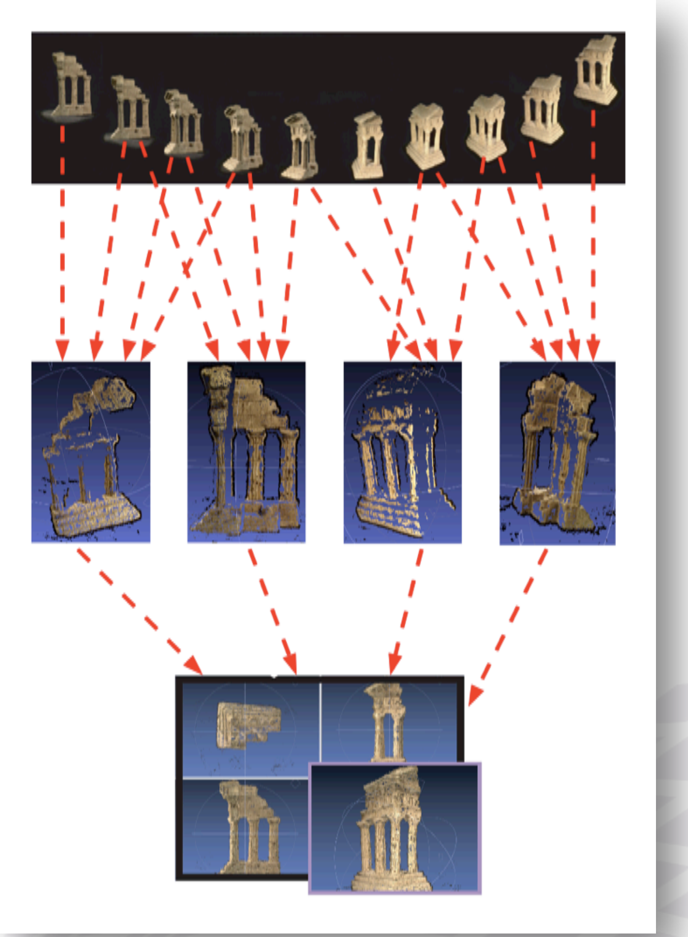
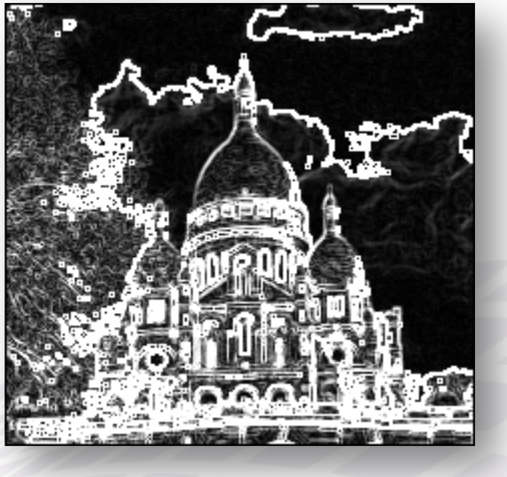


Image processing

Edge detection algorithm

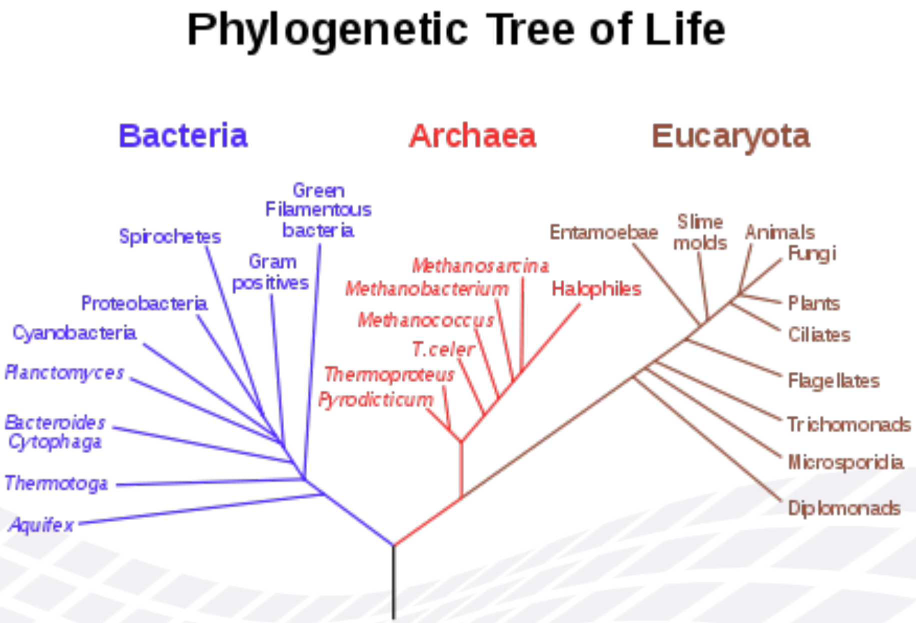
- Sobel Filter benchmark
- Size
  - ~ 200 lines of C code
- GPU C2070 improvement
  - x 25,8 over serial code on Nehalem
- Main porting operation
  - Use of basic HMPP directives



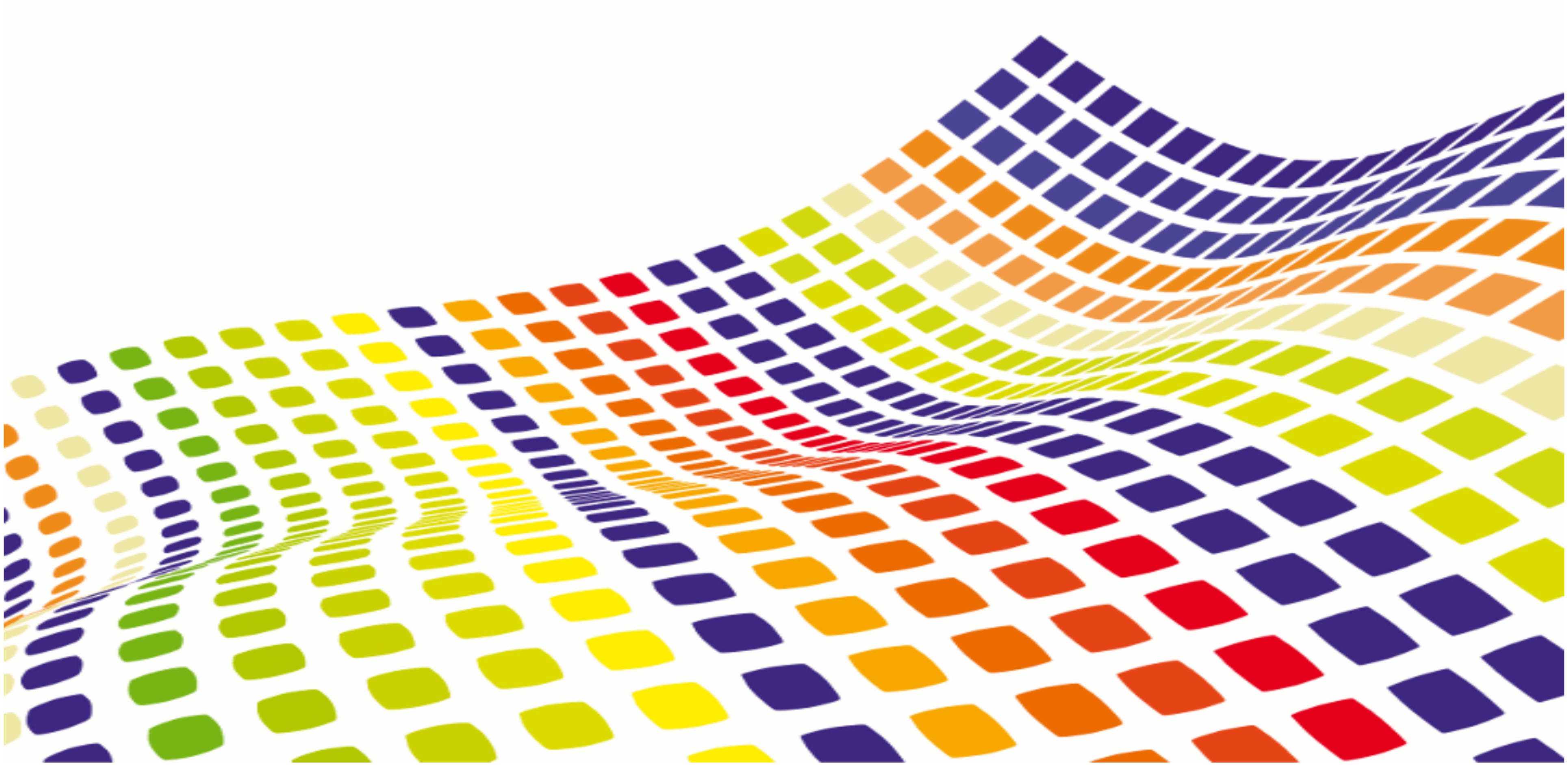
Biosciences, phylogenetics

Phylip, DNA distance

- In association with the HMPP Center Of Excellence for APAC
- Computes a matric of distances between DNA distances
- Resource spent
  - A first CUDA version developed by Shanghai Jiao Tong University, HPC Lab
  - 1 man-month
- Size
  - 8700 lines of C code, one main kernel (99% of the execution time)
- GPU C2070 improvement
  - x 44 over serial code on Nehalem
- Main porting operation
  - Kernel parallelism & data transfer coalescing leverage
  - Conversion from double precision to simple precision computation



An Economical View of GPU Computing



CAPEX-OPEX Analysis for a Heterogeneous System

- Capital Expenses (CapEx)
  - System acquisition cost
  - Software migration cost
  - Software acquisition cost
  - Teaching cost
  - Real estate cost

- Operational Expenses (OpEx)
  - Energy cost (system + cooling)
  - Maintenance cost

- For a given amount of compute work, the CapEx-Opex analysis indicates the “real” value of a given system
  - For instance, if I add GPU do I save money? And how many should I add?
  - Then should I use slower CPU?

Application Speedup and CapEx-OpEx

- Adding GPUs/accelerators to the system
  - Increases system cost
  - Increases base energy consumption (one GPU = x10 watt idle)

- Exploiting the GPUs/accelerators
  - Decreases execution time, so potentially the energy consumption for a given amount of work
  - Reduces the number of nodes of the architecture
    - Threshold effect on the number of routers etc.
  - Requires to migrate the code

- Multiple views of the value of application speedup
  - Shorten time-to-market
    - Threshold effect
  - More work performed during the lifetime of the system

CapEx Hardware Parameters

- Choice of the hardware configuration can be:
  - Fast CPU + Fast GPU (expensive node)
  - Slow CPU + Fast GPU
  - Fast CPU + Slow GPU
  - Slow CPU + Slow GPU
  - Fast CPU
  - Slow CPU

Small systems:
- a few nodes (1-8)
- cost x10k€

Large systems
- many nodes (x100)
- cost x1M€

- Nodes performance impact on the number of nodes
  - More nodes means more network with non negligible cost and energy consumption
  - Less nodes may limit scalability issues if any
- Application workload analysis is the only way to decide
  - Optimizing software can significantly increase performance and so reduce needed hardware
  - Code migration to GPU is on the critical path

CapEx: Code Migration Cost

Mastering the cost of migration has
a significant impact on the total cost
for small systems

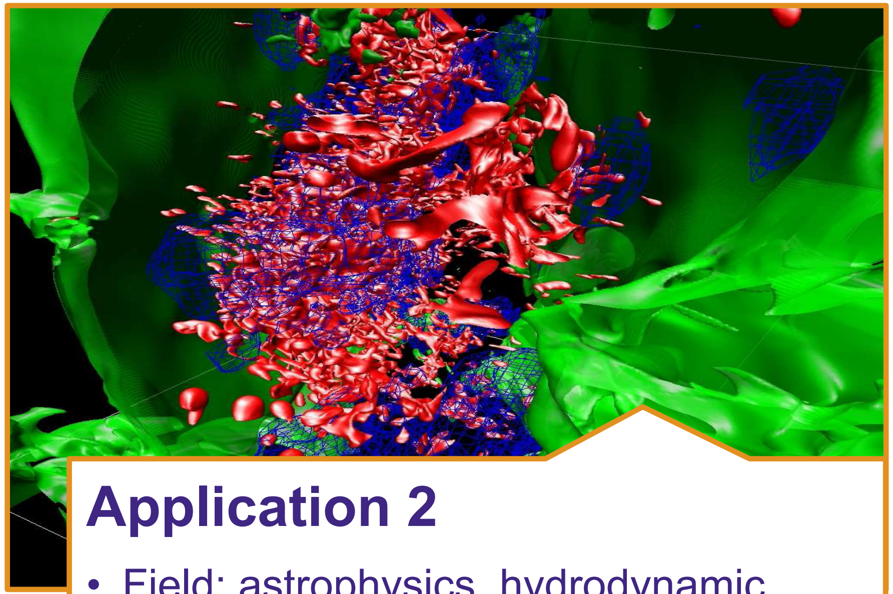
Typical effort:
- Manpower: a few Man-Months
- Cost: x 10k€

- Migration cost
  - Learning cost
  - Software environment cost
  - Porting cost
- Migration cost is mostly hardware size independent
  - Not an issue for dedicated large systems
  - Different if the machine aims at serving a large community
- Main migration benefit is to highlight manycore parallelism
  - Not specific to one kind of device
  - Implementation is specific
- Constructor specific implementation solution
  - Amortize period similar to the one of the hardware (3 years)
- Agnostic parallelism expression
  - Using portable solution for multiple hardware generations (amortized on 10 years)
  - Of course not that simple! Still requires some level of tuning
- May be very useful for non scalable message passing code

Two Applications Examples

Application 1

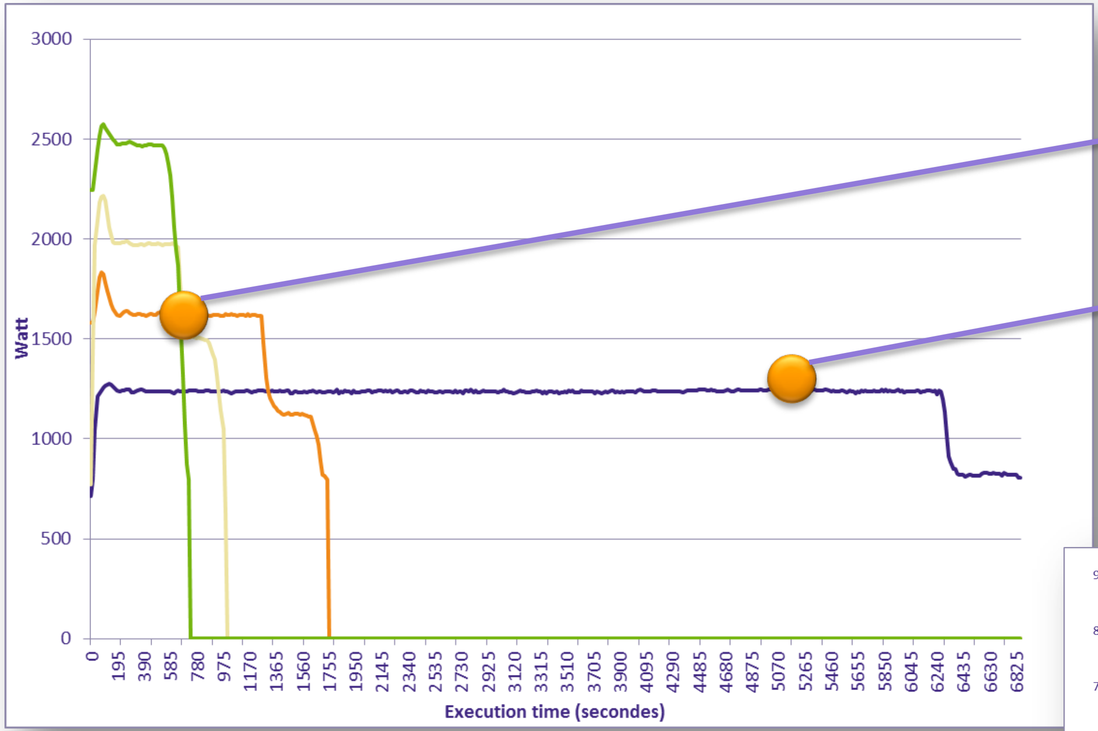
- Field: Monte Carlo simulation for thermal radiation
- MPI code
- Migration cost: 1 man month



Application 2

- Field: astrophysics, hydrodynamic
- MPI code
- Requires 3 GPUs per node for having enough memory space
- Migration cost: 2 man month

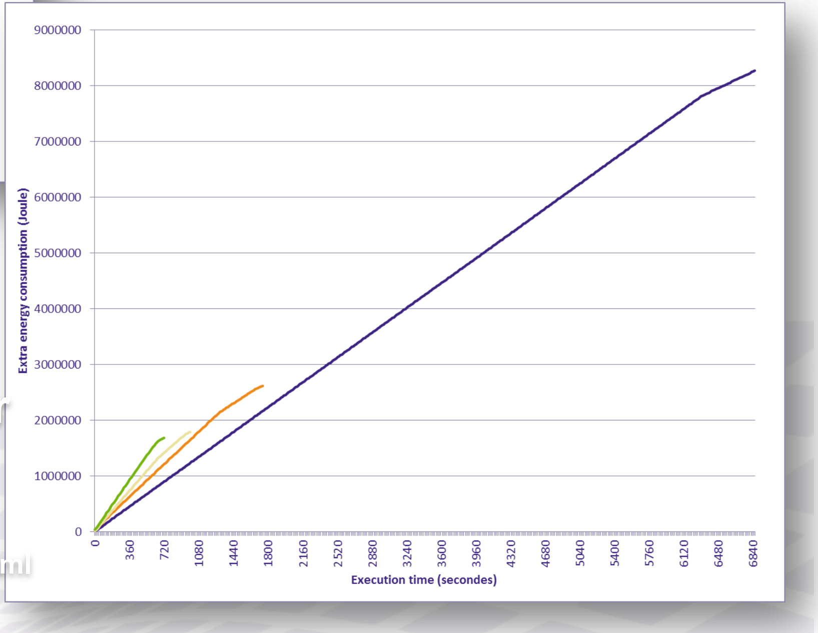
Power Consumption Application 1



GPU energy

CPU energy

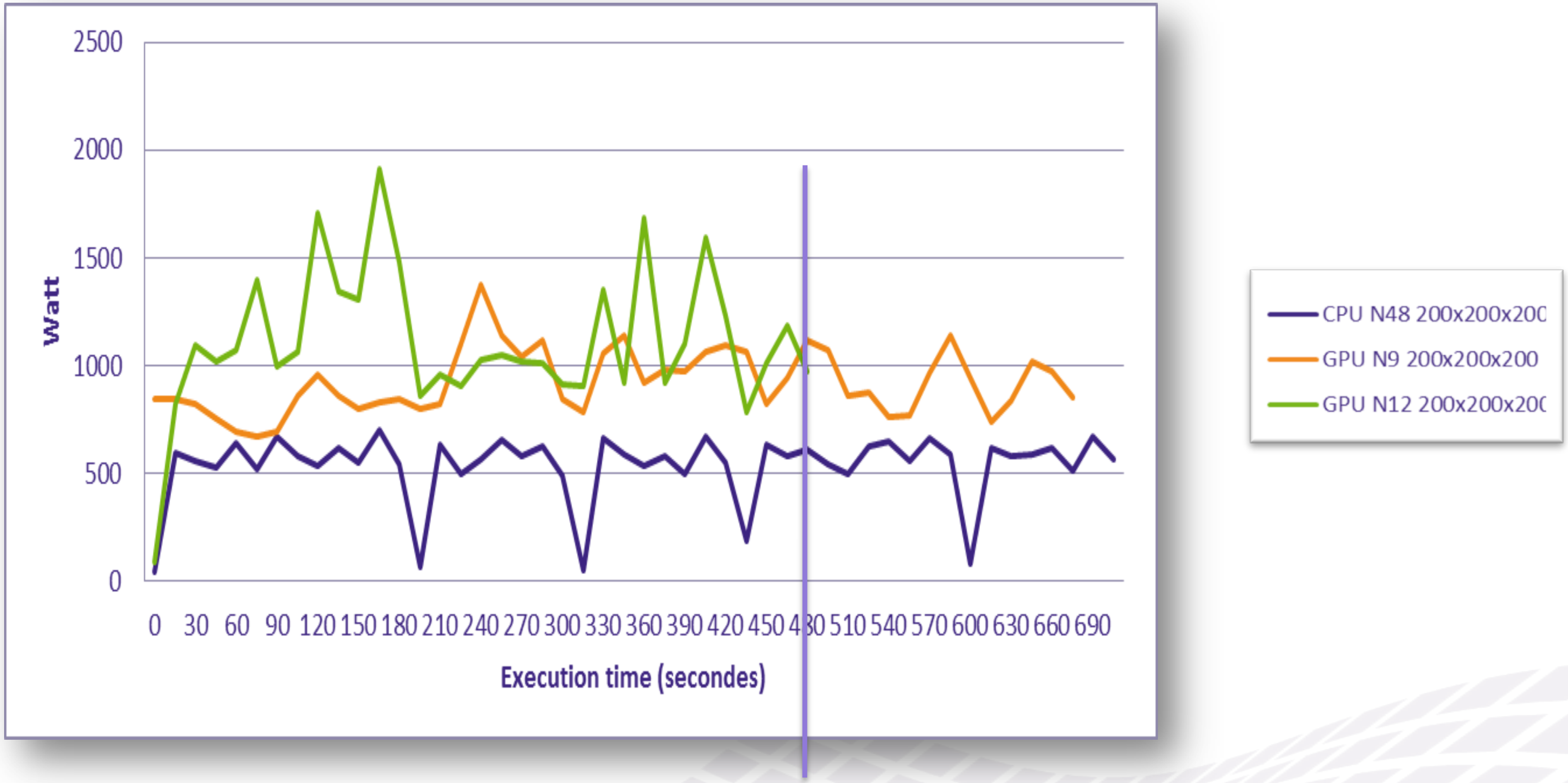
0 = Baseline Energy Consumption



Power usage effectiveness (PUE) =
Total facility power / IT equipment power

Current 1.9, best practice 1.3
Src: http://www.google.com/corporate/datacenter/efficiency-measurements.html

Power Consumption Application 2

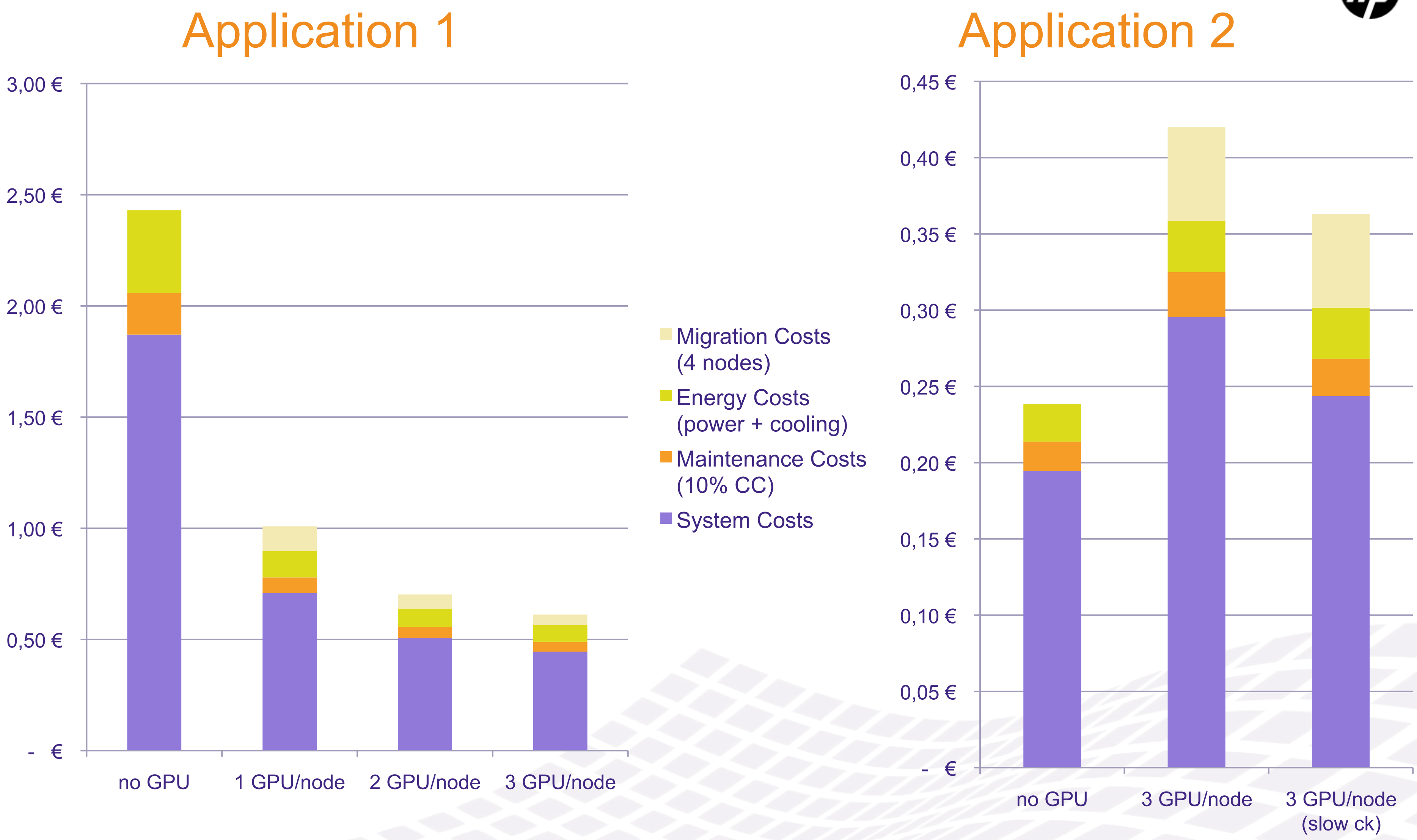


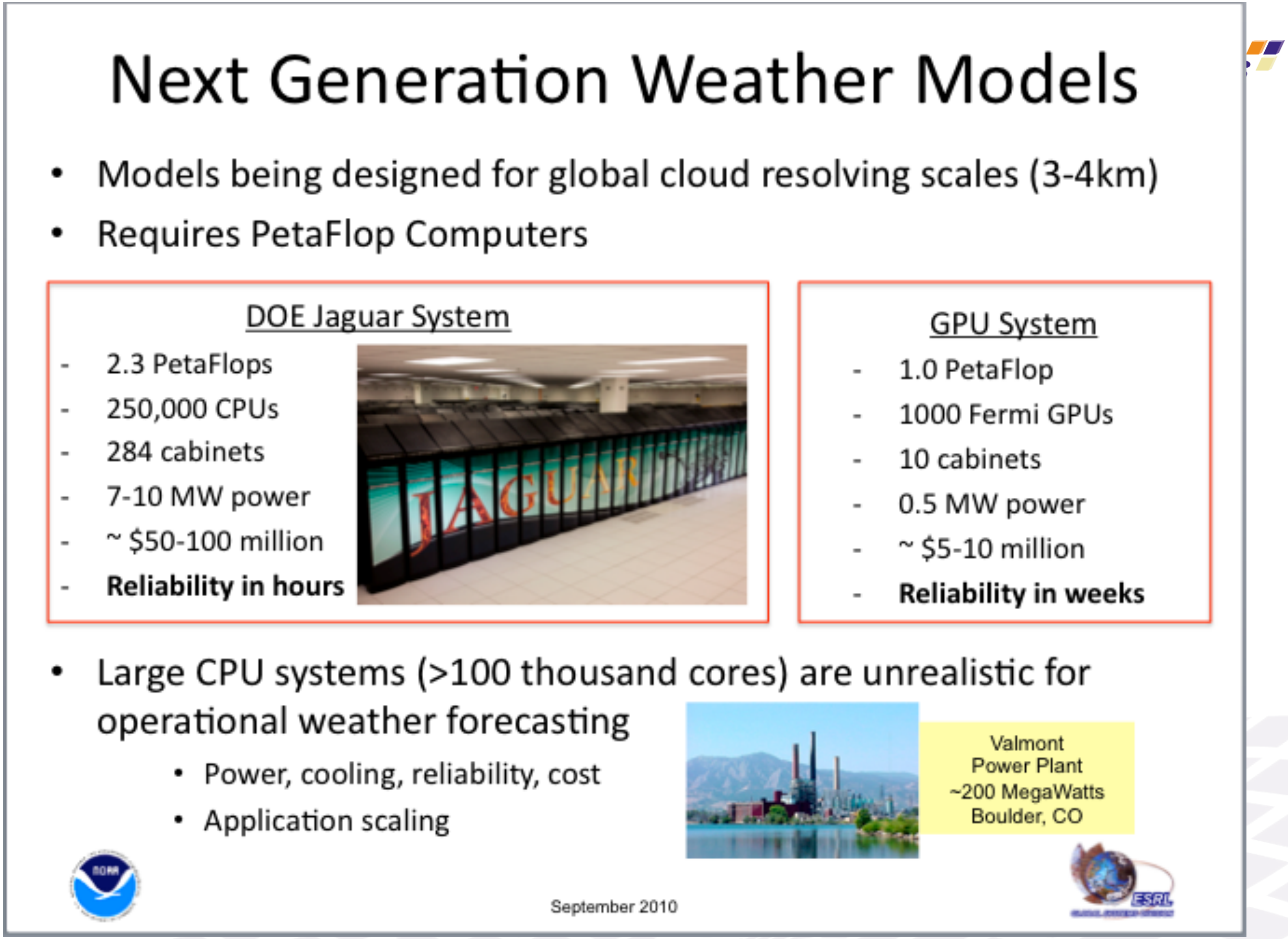
CAPEX-OPEX Overview

- Comparison on an equivalent workload
  - CAPEX = System costs + Migration costs
  - OPEX = Energy cost + Computer maintenance cost (10% Computer costs)

Configuration	Execution time (s)	System Costs	Maintenance Costs	Energy Costs	CAPEX +OPEX
Application 1		**Migration cost = 1 man.month**			
4 nodes	6862	1.87€	0.19€	0.37€	**2.43€**
4 nodes + 4 GPUs	1744	0.71€	0.07€	0,12€	**0.90€**
4 nodes + 8 GPUs	1000	0.51€	0.05€	0,08€	**0.64€**
4 nodes + 12 GPUs	731	0.45€	0.04€	0,08€	**0.57€**
Application 2		**Migration cost = 2 man.month**			
4 nodes	713	0.19€	0.02€	0.025€	**0.239€**
4 nodes + 12 GPUs	485	0.30€	0.03€	0.034€	**0.358€**
4 nodes (slow ck)+ 12 GPUs	500 (estim.)	0.24€	0.02€	0.034€	**0.302€**

Cost per Run


Application 1
3,00 €
2,50 €
2,00 €
1,50 €
1,00 €
0,50 €
- €
no GPU
1 GPU/node
2 GPU/node
3 GPU/node
Migration Costs (4 nodes)
Energy Costs (power + cooling)
Maintenance Costs (10% CC)
System Costs
Application 2
0,45 €
0,40 €
0,35 €
0,30 €
0,25 €
0,20 €
0,15 €
0,10 €
0,05 €
- €
no GPU
3 GPU/node
3 GPU/node (slow ck)



Next Generation Weather Models

- Models being designed for global cloud resolving scales (3-4km)
- Requires PetaFlop Computers

DOE Jaguar System

- 2.3 PetaFlops
- 250,000 CPUs
- 284 cabinets
- 7-10 MW power
- ~ $50-100 million
- **Reliability in hours**

GPU System

- 1.0 PetaFlop
- 1000 Fermi GPUs
- 10 cabinets
- 0.5 MW power
- ~ $5-10 million
- **Reliability in weeks**

- Large CPU systems (>100 thousand cores) are unrealistic for operational weather forecasting
  - Power, cooling, reliability, cost
  - Application scaling

Valmont
Power Plant
~200 MegaWatts
Boulder, CO

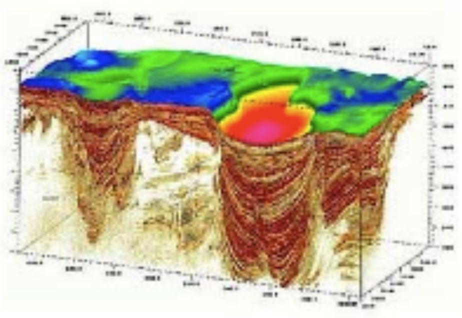
September 2010

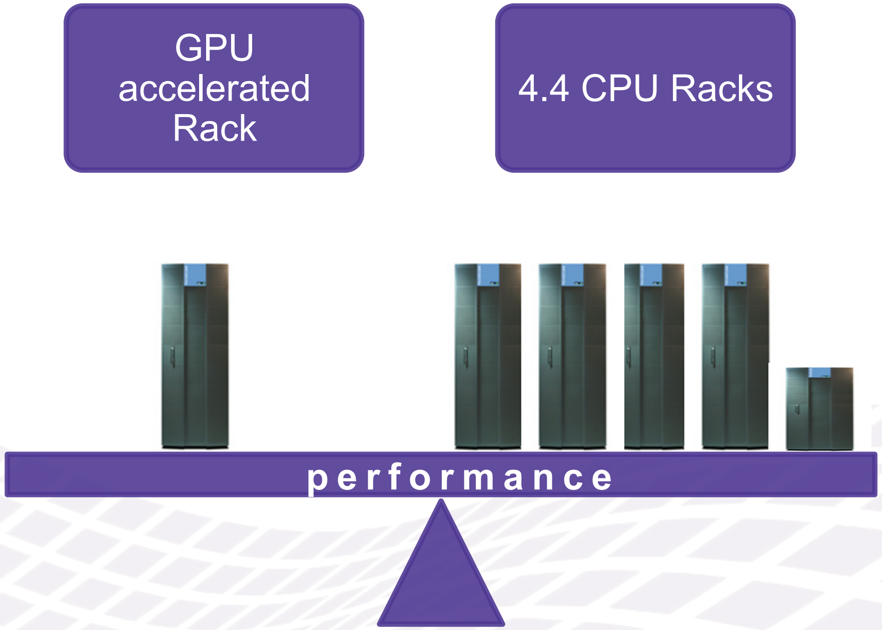
SRC: NOAA

Oil & Gas

GPU-accelerated seismic depth imaging

- 1 GPU accelerated machine = 4.4 CPU machines
  - GPU: 16 dual socket quadcore Intel Hapertown nodes connected to 32 GPUs
  - CPU: 64 dual socket quadcore Intel Hapertown nodes





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
  - Software migration required understanding options
    - Do not want to backtrack
  - 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

Perspectives

- Need for new standard programming
  - OpenHMPP initiative launch by CAPS
  - http://www.openhmpp.org/
- Energy consumption control at software level
  - Is energy saving cost worthwhile the software tuning cost?
- Cloud technology
  - All manycore issues and more …

http://www.caps-entreprise.com

http://twitter.com/CAPSentreprise