CAPS

hmpp

Introduction au calcul scientifique sur GPU

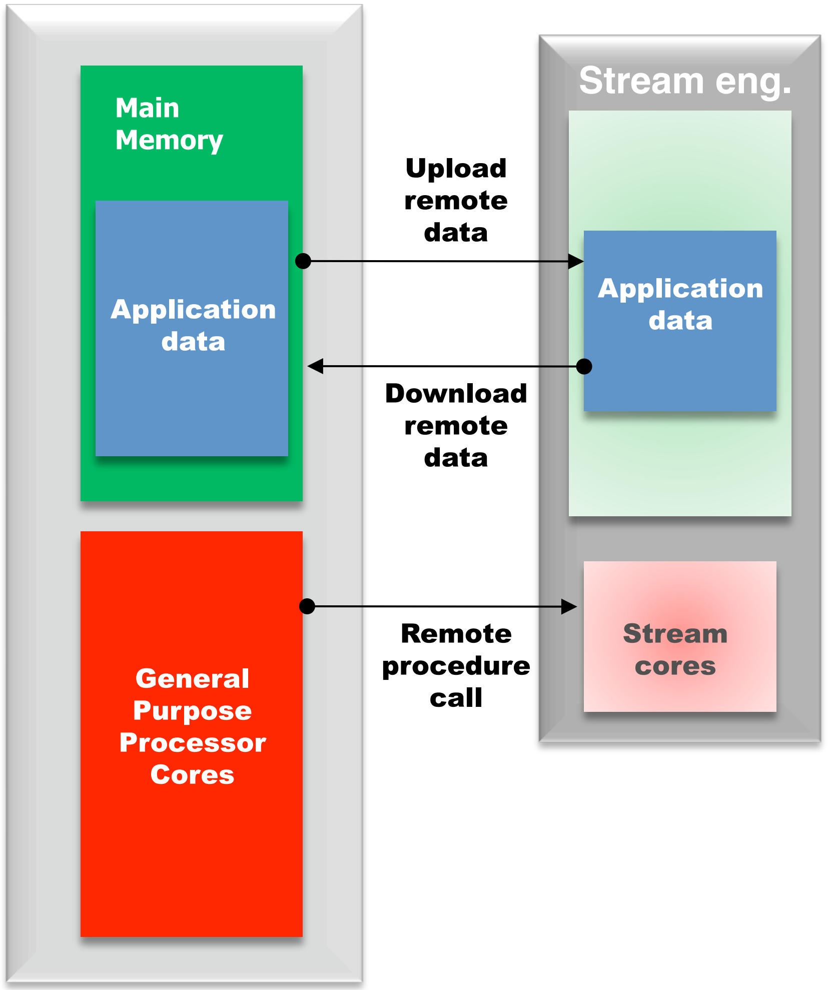
F. Bodin, francois.bodin@caps-entreprise.com

Introduction

- Main stream applications will rely on new multicore / manycore architectures
    - It is about performance not parallelism
- Numerous legacy applications can benefit from GPU computing
- Updating applications to benefit from parallel architectures may require extensive rewriting of the codes

Manycore Architectures

- General purpose cores
  - Share a main memory
  - Core ISA provides fast SIMD instructions
- Streaming engines (e.g. GPU)
  - Application specific architectures ("*narrow band*")
  - Vector/SIMD
  - Can be extremely fast
- Hundreds of GigaOps
  - But not easy to take advantage of
  - One platform type cannot satisfy everyone
- Operation/Watt is the efficiency scale



Outline

- History of GPUs as compute devices
- An overview of GPU architectures
- Stream computing
- HMPP
- High level code generation and tuning
- An example of seismic application (RTM)

The History of GPGPU: The Dark Ages

- The 80s / early 90s: the advent of discrete 3D pipeline
  - Sun GT “graphics tower”: external box hooked to the computer via an interface card
- The 90s / early 21st century: the advent of consumer-grade 3D systems
  - From the 3Dfx Voodoo to the NVIDIA GeForce FX and ATI Radeon R200
- Finally, DirectX 9.0c and OpenGL 2.0 brings 32 bits floating point (FP) to GPUs: the GeForce 6 and Radeon R500
- General-Purpose Computation on GPU (GPGPU)
  - http://www.gpgpu.org/

The History of GPGPU: The Dawn

- Early 2003, GPU overtakes CPU for single precision FP, with expectations for more (Khailany 2003)
- Programmable shaders appear at the same time
- First GPGPU applications (for instance, Bolz 2003)

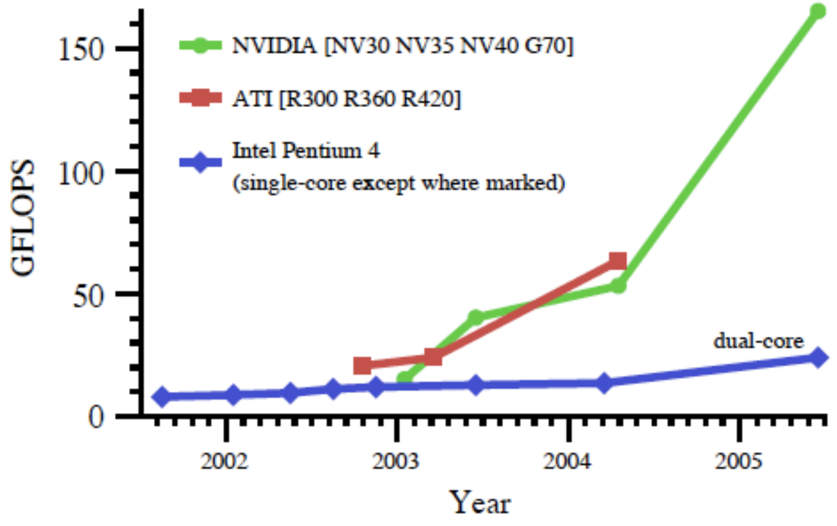
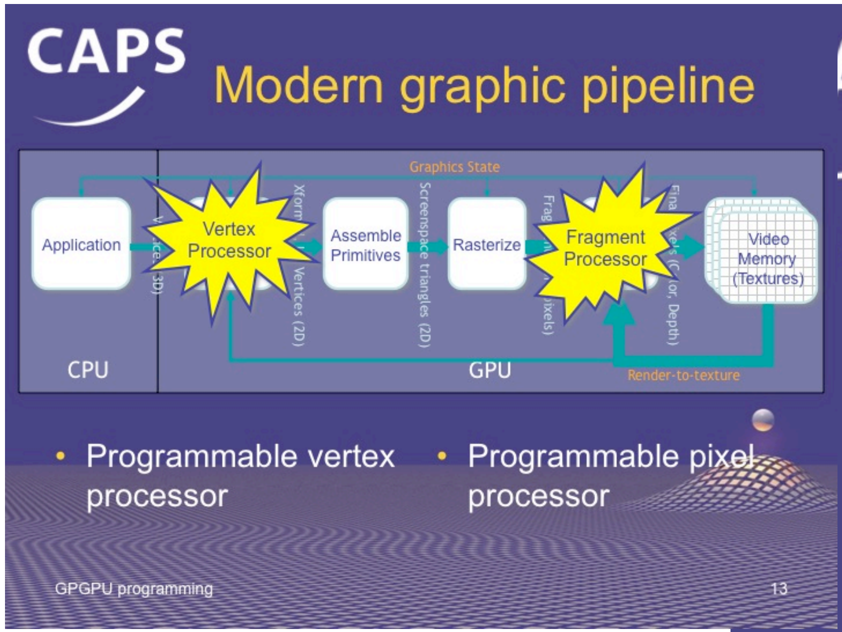


Figure 1: *The programmable floating-point performance of GPUs (measured on the multiply-add instruction as 2 floating-point operations per MAD) has increased dramatically over the last four years when compared to CPUs. Figure courtesy Ian Buck, Stanford University.*

The History of GPGPU: the OpenGL Era

- From 2003 to late 2006/ early 2007
  - GPU Programming using OpenGL 3D library
  - Needs understanding graphic pipeline and work around limitations



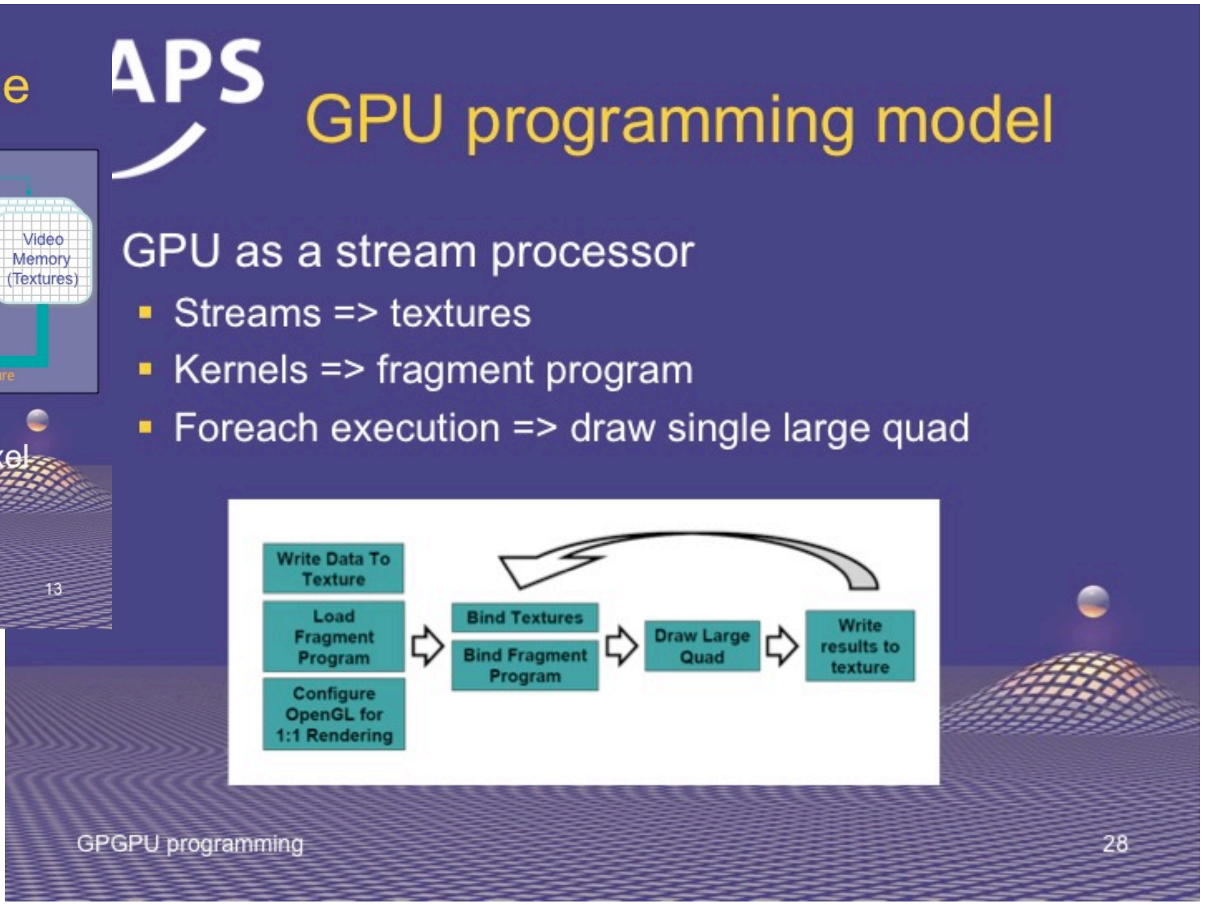
CAPS

Modern graphic pipeline

Graphics State
Application
Vertex Processor
Xform
Vertices (3D)
Assemble Primitives
Vertices (2D)
Screenspace triangles (2D)
Rasterize
Frag
Pixels
Fragment Processor
Final Pixels (Color, Depth)
Video Memory (Textures)
CPU
GPU
Render-to-texture

- Programmable vertex processor
- Programmable pixel processor

GPGPU programming 13



APS

GPU programming model

GPU as a stream processor

- Streams => textures
- Kernels => fragment program
- Foreach execution => draw single large quad

Write Data To Texture
Load Fragment Program
Configure OpenGL for 1:1 Rendering
Bind Textures
Bind Fragment Program
Draw Large Quad
Write results to texture

GPGPU programming 28

CAPS

The History of GPGPU: the Mainstream Era

- Brook first beta was in late 2004 but was OpenGL-based
- Specialized vendors such as PeakStream released libraries and tools, again using OpenGL at first
- CAPS to demo a C-to-OpenGL generator at SC'06

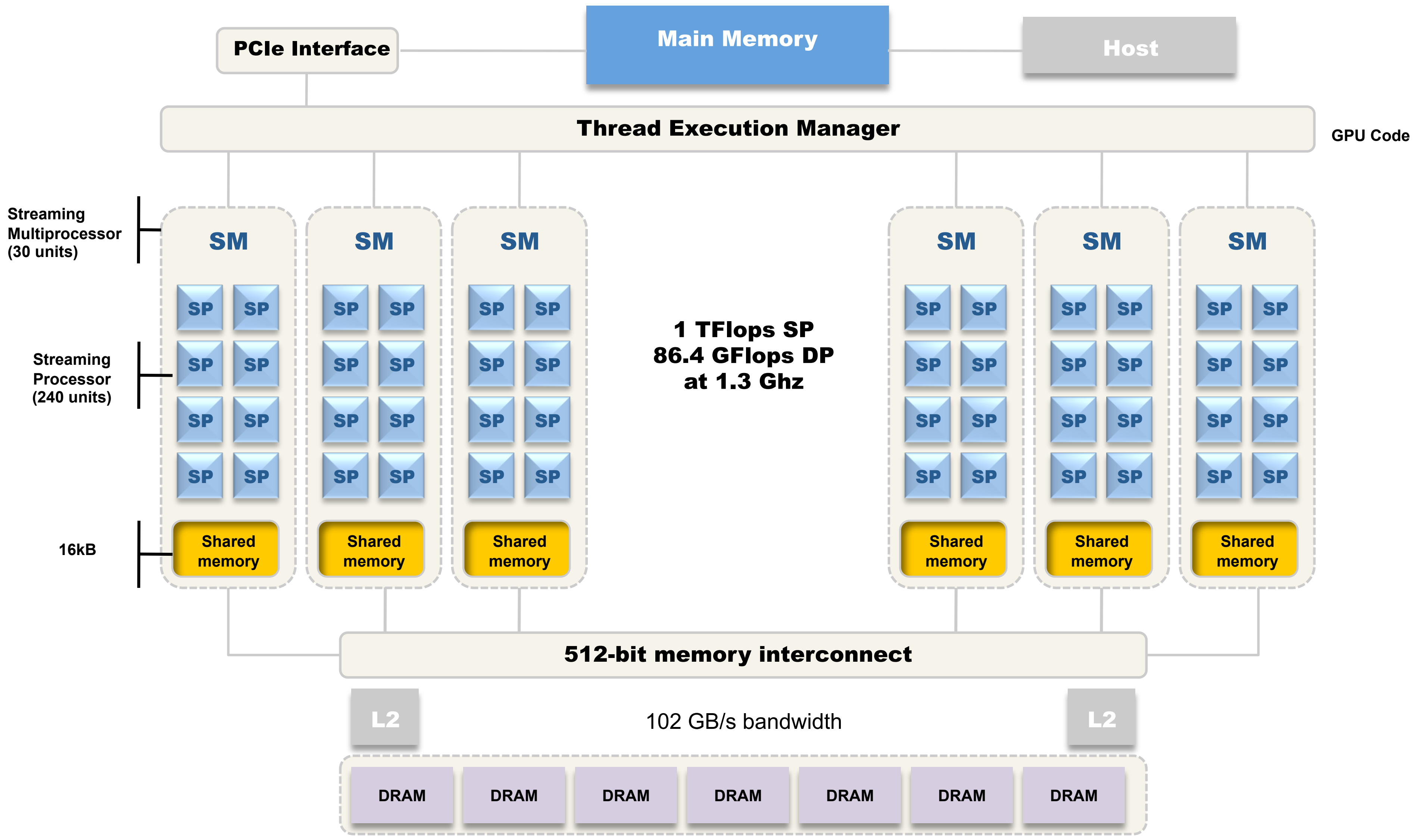
- Feb. 2007: the first release of NVIDIA CUDA
- Dec. 2007: the first release of the ATI Stream SDK, replacing CTM (Close To Metal)

- OpenGL is no longer needed, GPGPU becomes exploitable for more than research
- Late 2008, Double Precision FP is available !
- OpenCL as a new standard?

GPU Architecture Overview

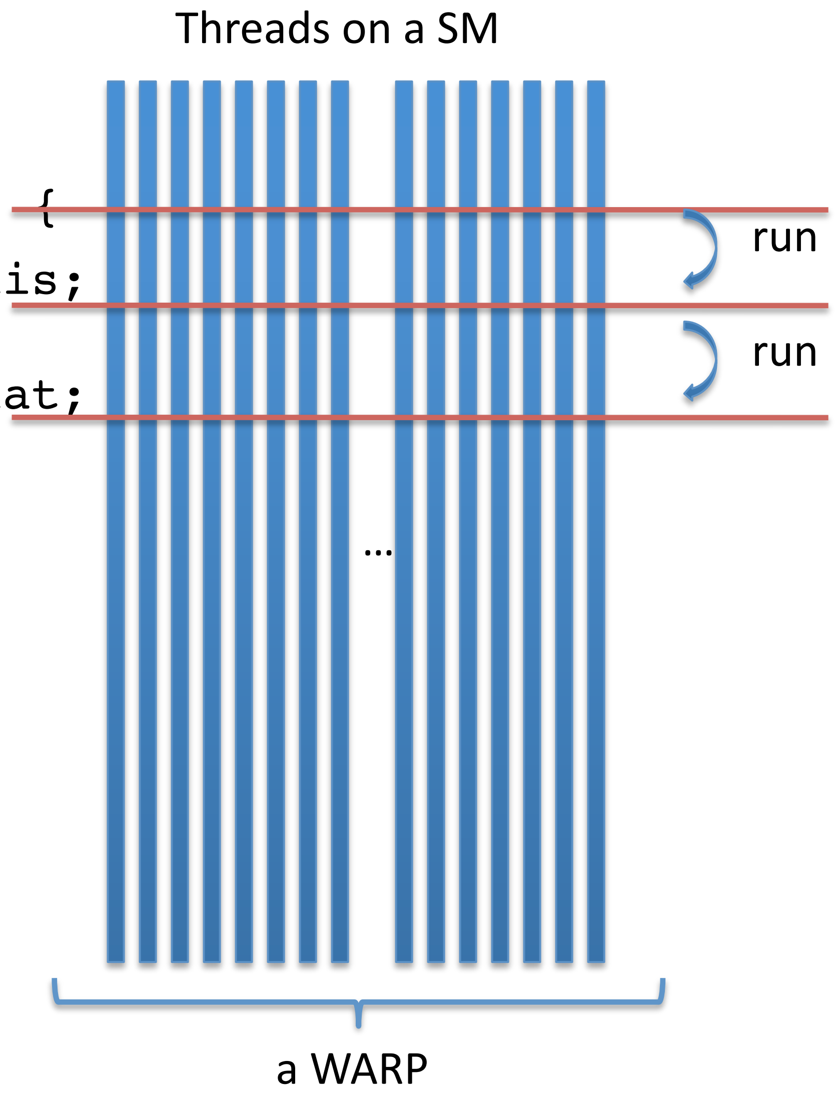
- Fine grain massive data parallelism
  - Many streaming processors
- Exposed memory hierarchy
  - Large device memory
  - Small partially *shared* memory, …
- Heavily pipelined memory
  - Small data caches
- Fixed amount of hardware resources
  - No virtual memory
  - Maximum number of threads
  - Maximum number of registers usage per group of threads
  - …

Example: NVIDIA T10



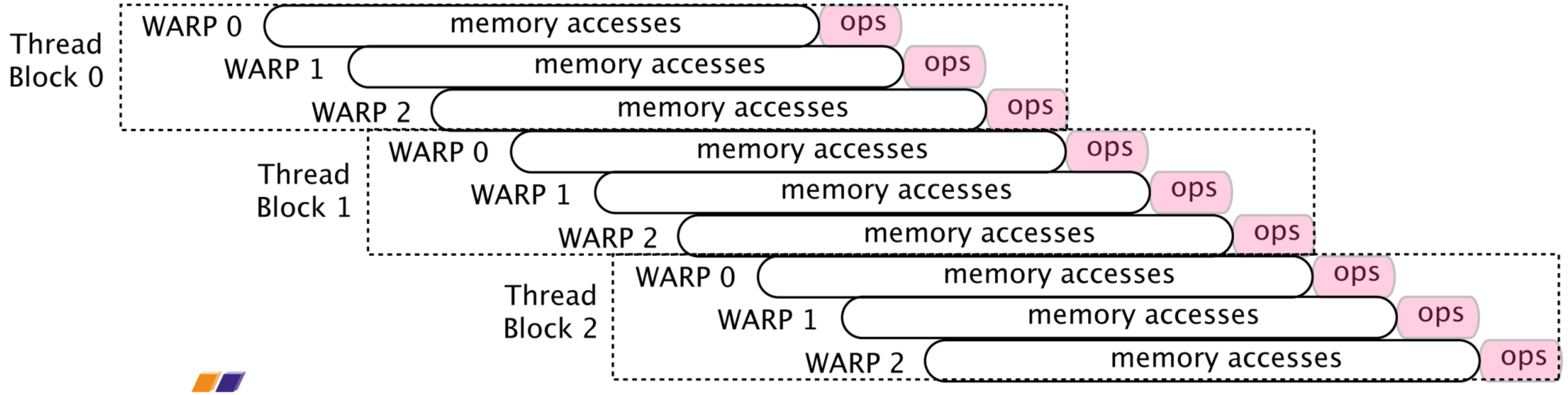
Many Threads Running in Lock Step

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



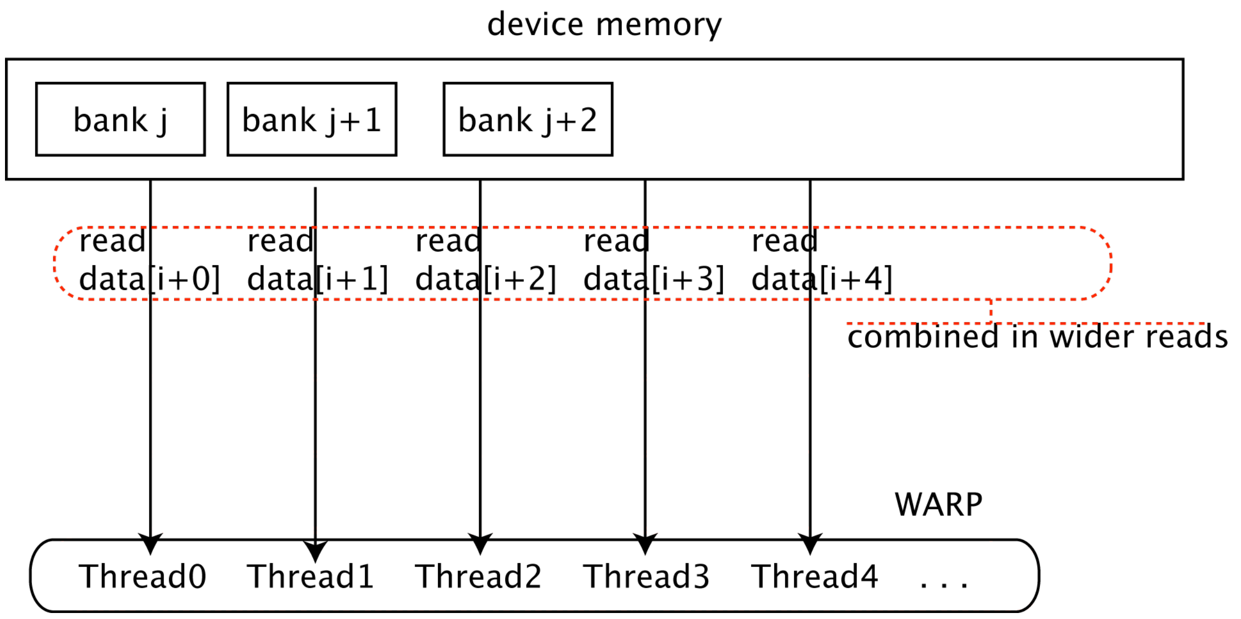
Thread Execution

- Streaming multiprocessors execute threads per groups called a WARP
  - 1 WARP = 32 threads
  - Threads are executed in a SIMD ways, they all execute the same statement at the same time but on different data (lock step).
  - WARPs from a thread block or from multiple blocks, depending on hardware resources use, are pipelined



Coalesced Memory Accesses

- Hardware coalescing combines memory accesses, at near addresses, by threads in a (half)-warp into a set of wider memory transactions (32B, 64B, or 128B)
  - Aligning the warp memory accesses to exact multiples of 32, 64 or 128 bytes is not strictly required but helps reducing the overall number of memory reads



Consequences of GPU Architectures

- Host-Device model
  - High overhead when moving data between CPU and GPU
  - Not always good to move a computation to GPU
- Large number of registers
  - Loop unrolling is an important transformation
- Small streaming multiprocessor shared memory
  - Help reducing pressure on device memory
- Penalty for over-spending a type of resource is huge
  - e.g. too many threads by block
  - e.g. too many registers per thread, spilling is very expensive
- Memory bound device
  - High arithmetic computing density needed
- Spatial locality between threads is crucial
  - Exploit memory access coalescing capabilities
- Expensive global synchronization
  - No global hardware thread communication
- Performance is very problem size dependent
  - The number of threads and complexity of threads depend on the number of computations to perform

Evolution (1)

- GPUs are still multiplying compute resources (“cores” is an ambiguous term in GPUs, and not directly comparable across vendors)
  - NVIDIA
    - 8800GTX: 128 “cores”
    - GTX280: 240 “cores”
    - Upcoming “Fermi”: up to 512 “cores”
  - ATI
    - Radeon HD3870: 320 “cores”
    - Radeon HD4870: 800 “cores”
    - Radeon HD5870: 1600 “cores”
- CPUs are doing the same, but at a slower pace
  - Xeon 55xx is still only 4 cores, like the Xeon 54xx
  - Opteron has moved to 6 cores, as did the Xeon 74xx

Evolution (2)

- Memory bandwidth is still a bottleneck
  - GPUs can offer 100+ GB/s of bandwidth, but with strict requirements on access patterns
  - Xeon 55xx have more than thrice the bandwidth of the Xeon 54xx. And bandwidth is doubled by going from one to two sockets, as is the case of the Opteron.
- GPU FP accuracy is now mostly in line with CPU, offering near-complete IEEE754 compliance
- GPGPU is still limited by PCIe connectivity
  - No direct access from the CPU to the GPU memory

Evolution (3)

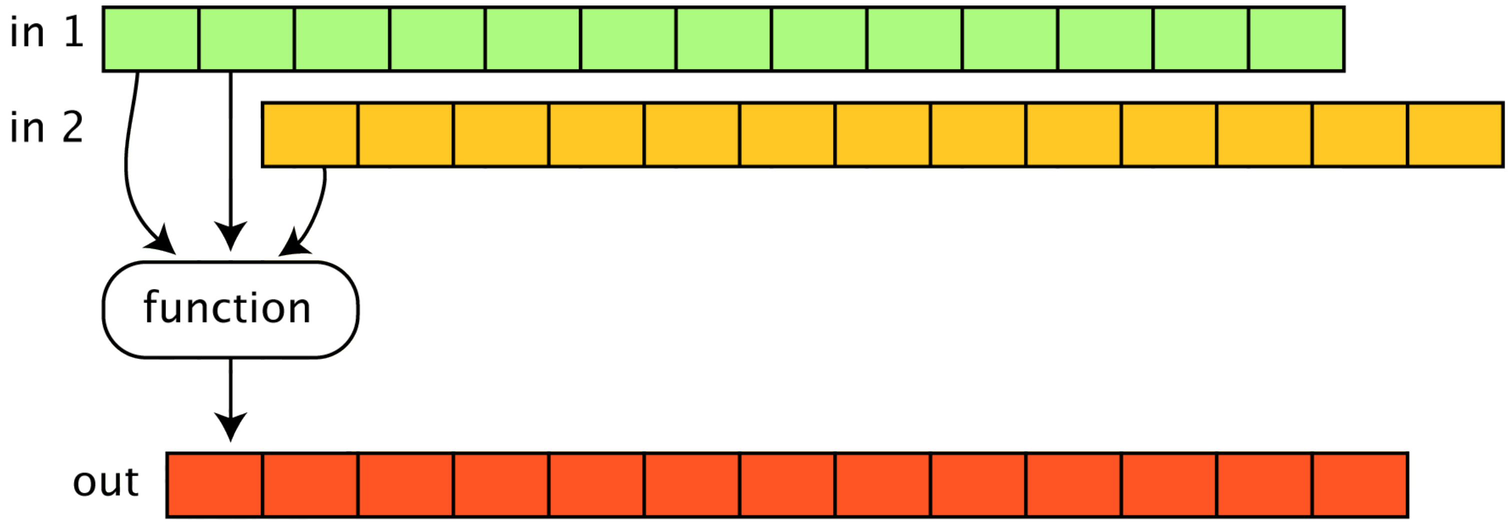
- CPU vendors try to close the gap
  - On-chip GPU
  - AMD Fusion (Llano), Intel Clarkdale / Arrandale, …
  - Mostly embedded/desktop for now, high-perf to come?
- Is GPU tending to become General Purpose Unit?
  - Upcoming NVIDIA Fermi is more flexible than the current GT200 architecture
  - Intel Larrabee (?) presentations insist on the versatility of this upcoming x86-based "GPU": CPU-GPU convergence
  - GPGPU users push towards an easier-to-program architecture
  - … but do graphics users agree?
  - Strength of GPU was specialization

GPGPU Today

- Supported on consumer and professional hardwares
  - Tesla S1070: external box hooked to the computer via an interface card
- Programmed with dedicated tools
  - NVIDIA CUDA, ATI Stream SDK
  - OpenCL
  - CAPS HMPP compiler, PGI compiler
- Not as “easy” as traditional homogeneous computing, but getting better
- Only 3 years after wide availability, already some success stories and major deployment
- OpenCL from Khronos consortium
  - C based, low level, great for code generation tool
- Upcoming NVIDIA Fermi architecture
  - Enhanced DP architecture
  - Unified memory space, ECC memory, concurrent kernels, …

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



Brook+

```
kernel voidsum(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);
 sum(a, b, c);
 streamWrite(c, input_c);
 ...
}
```

RapidMind

- Based on C++
  - Runtime + JIT
  - Internal data parallel language

From RapidMind

```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++)
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

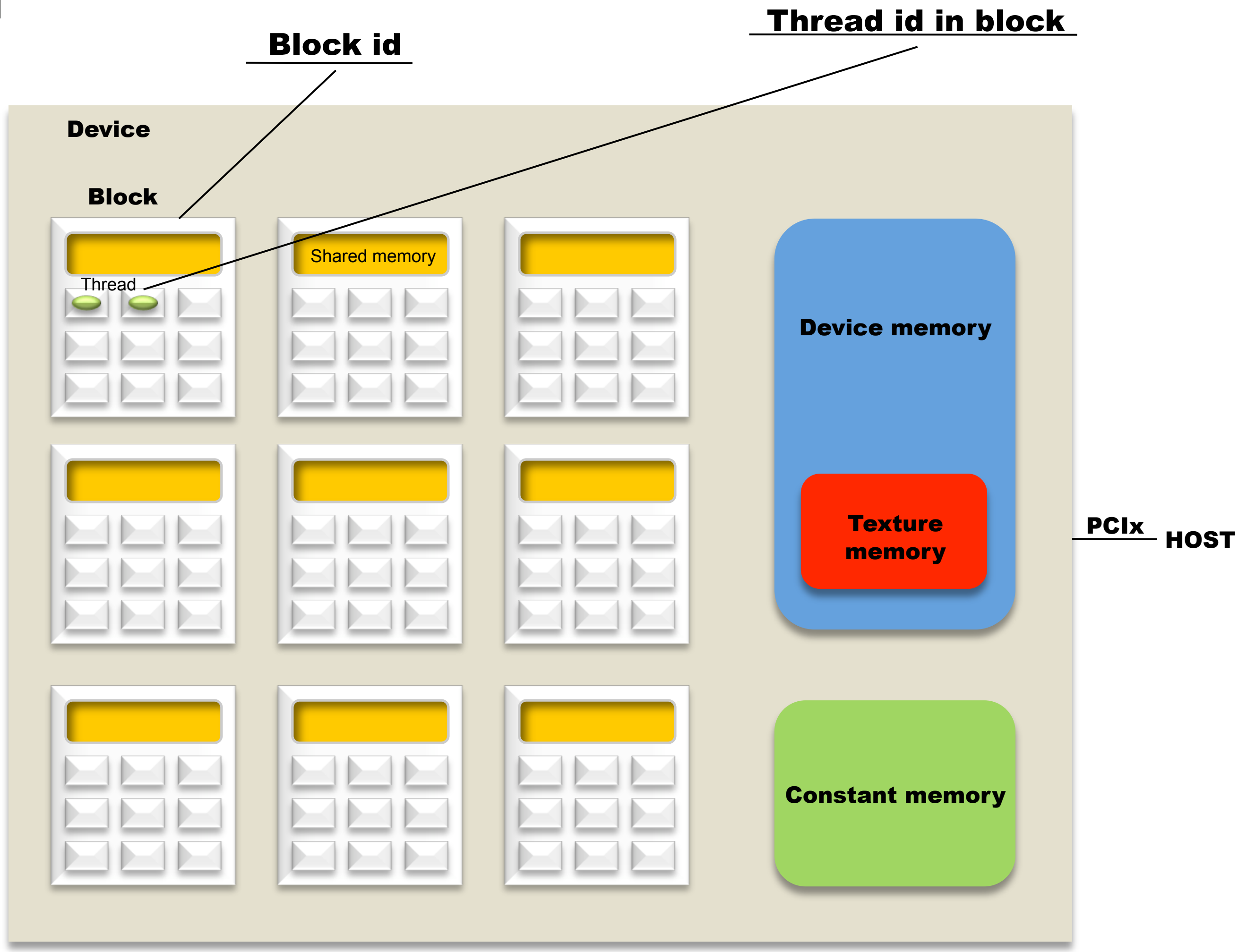
```
#include <rapidmind/platform.hpp>
using namespace rapidmind;

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}

void func_arrays() {
 Program func_prog = BEGIN {
  In<Value3f> r, s;
  Out<Value3f> q;
  q = func(r,s);
 } END;
 a = func_prog(a,b);
}
```

CUDA Overview

- “Compute Unified Device Architecture”
- C base language but with syntax and semantic extensions
- GPU is a coprocessor to a host (CPU)
- Make use of data parallelism thanks to the massively parallel GPU architecture

CUDA (OpenCL) Grids and Blocks

- GPUs need 1000s of threads to be efficient
  - Highly pipelined
  - Highly parallel



- Blocks of thread are executed on the streaming multiprocessors

CUDA (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 (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;
}
```

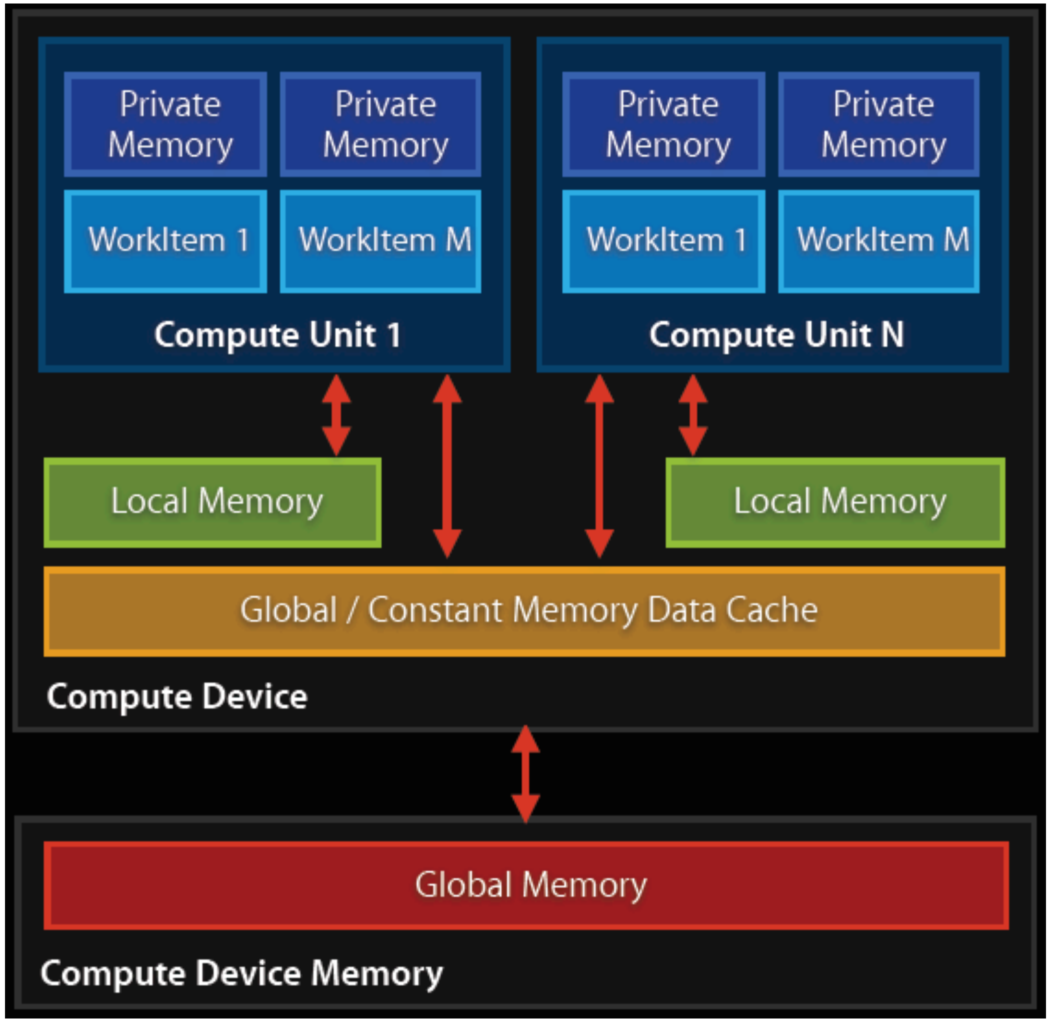
OpenCL Overview

- Open Computing Language
  - C-based cross-platform programming interface
  - Subset of ISO C99 with language extensions
  - Data- and task- parallel compute model

- Host-Compute Devices (GPUs) model

- Platform layer API and runtime API
  - Hardware abstraction layer, …
  - Manage resources

OpenCL Memory Hierarchy



From Aaftab Munshi's talk at Siggraph2008

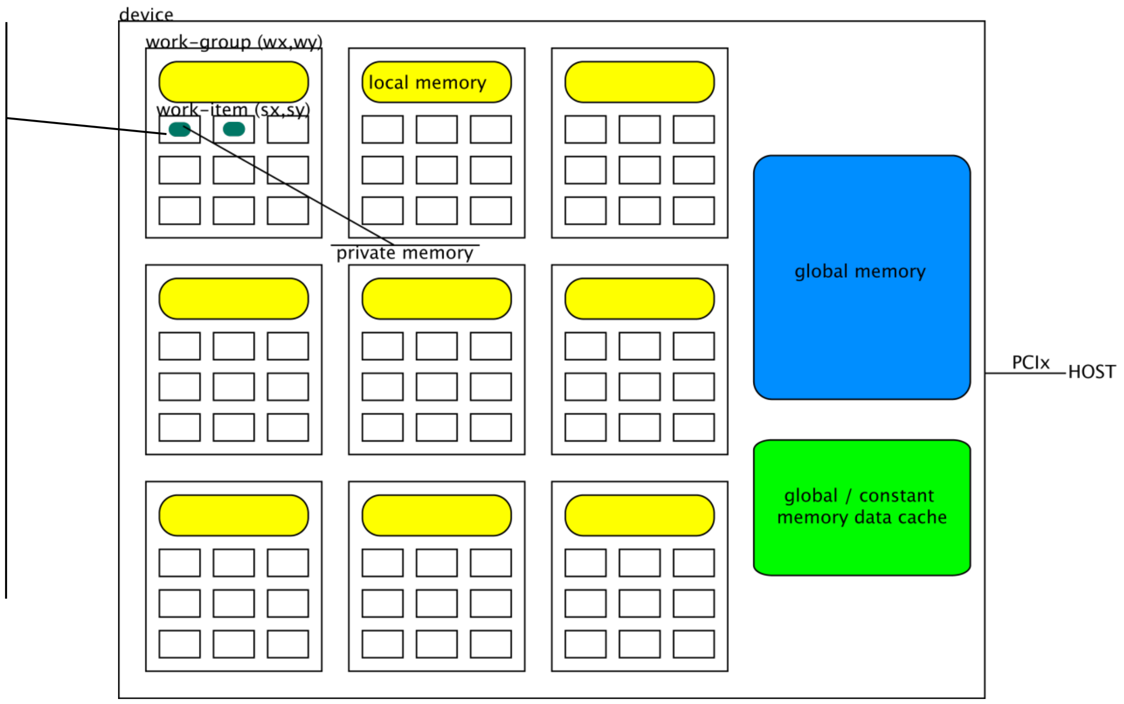
Platform Layer API& Runtime API

- Command queues
  - Kernel execution commands
  - Memory commands (transfer or mapping)
  - Synchronization
- Context
  - Manages the states
- Platform Layer
  - Querying devices
  - Creating contexts

Data-Parallelism in OpenCL

- A kernel is executed by the work-items

```
// OpenCL Kernel Function for element by element vector addition
__kernel void VectorAdd(__global const float8* a, __global const float8* b, __global float8* c)
{
    // get oct-float index into global data array
    int iGID = get_global_id(0);

    // read inputs into registers
    float8 f8InA = a[iGID];
    float8 f8InB = b[iGID];
    float8 f8Out = (float8)0.0f;

    // add the vector elements
    f8Out.s0 = f8InA.s0 + f8InB.s0;
    f8Out.s1 = f8InA.s1 + f8InB.s1;
    f8Out.s2 = f8InA.s2 + f8InB.s2;
    f8Out.s3 = f8InA.s3 + f8InB.s3;
    f8Out.s4 = f8InA.s4 + f8InB.s4;
    f8Out.s5 = f8InA.s5 + f8InB.s5;
    f8Out.s6 = f8InA.s6 + f8InB.s6;
    f8Out.s7 = f8InA.s7 + f8InB.s7;

    // write back out to GMEM
    c[get_global_id(0)] = f8Out;
}
```



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);
}
```

Why HMPP?

- GPU computing shown to be a great opportunity for many applications
  - Performance is critical in many applications fields
- Software needed for heterogeneous targets
  - Can we make it easy to use?
  - Can we address a large fraction of programmers?
- Recognize a technology transition period before standards settle
  - Parallel computing still evolving fast
- Supplement existing parallel APIs

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 API (OpenMP, MPI)
  - Do not oppose GPU to CPU,
- Consider multiple languages
  - Avoid asking users to learn a new language
- Consider resource management
  - Generate robust software
- Exploit HWA constructors programming tools
  - Do not replace, complement

HMPP Basis

- Directive based approach
  - Do not require a new programming language
    - And can be applied to many based languages
  - Already state of the art approach (e.g. OpenMP)
  - Keep incremental development possible

- Portability
  - To various hardware accelerators (HWAs) and host code is independent of the HWA

- Avoid exit cost

HMPP

- Brings remote procedure call on GPU
    - Host & compute device model
    - Synchronous and asynchronous

- Allows to optimize communication between CPU and GPU

- Provides high level GPU code generation from sequential code

- Provides device resource management

HMPP Example

```
#pragma hmpp sgemmlabel codelet, target=CUDA, args[vout].io=inout
extern void sgemm( int m, int n, int k, float alpha,
                   const float vin1[n][n], const float vin2[n][n],
                   float beta, float vout[n][n] );


int main(int argc, char **argv) {
…
 for( j = 0 ; j < 2 ; j++ ) {
#pragma hmpp sgemmlabel callsite
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
 }
```

HMPP Example

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

#pragma hmpp sgemm callsite, asynchronous
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    do something
    . . .
#pragma hmpp sgemm synchronize
#pragma hmpp sgemm delegatedstore, args[vout]

     do something else
#pragma hmpp sgemm release
```

Allocate and initialize device early

Execute asynchronously

Download result when needed

Release HWA

Optimizing Communications

- Allow to exploit
  - Communication / computation overlap
  - Temporal locality of RPC parameters

- Various techniques
  - Advancedload and delegatedstore
  - Constant parameter
  - Resident data
  - Actual argument mapping
  - Partial data transfers

Optimizing Communications Example

Preload data

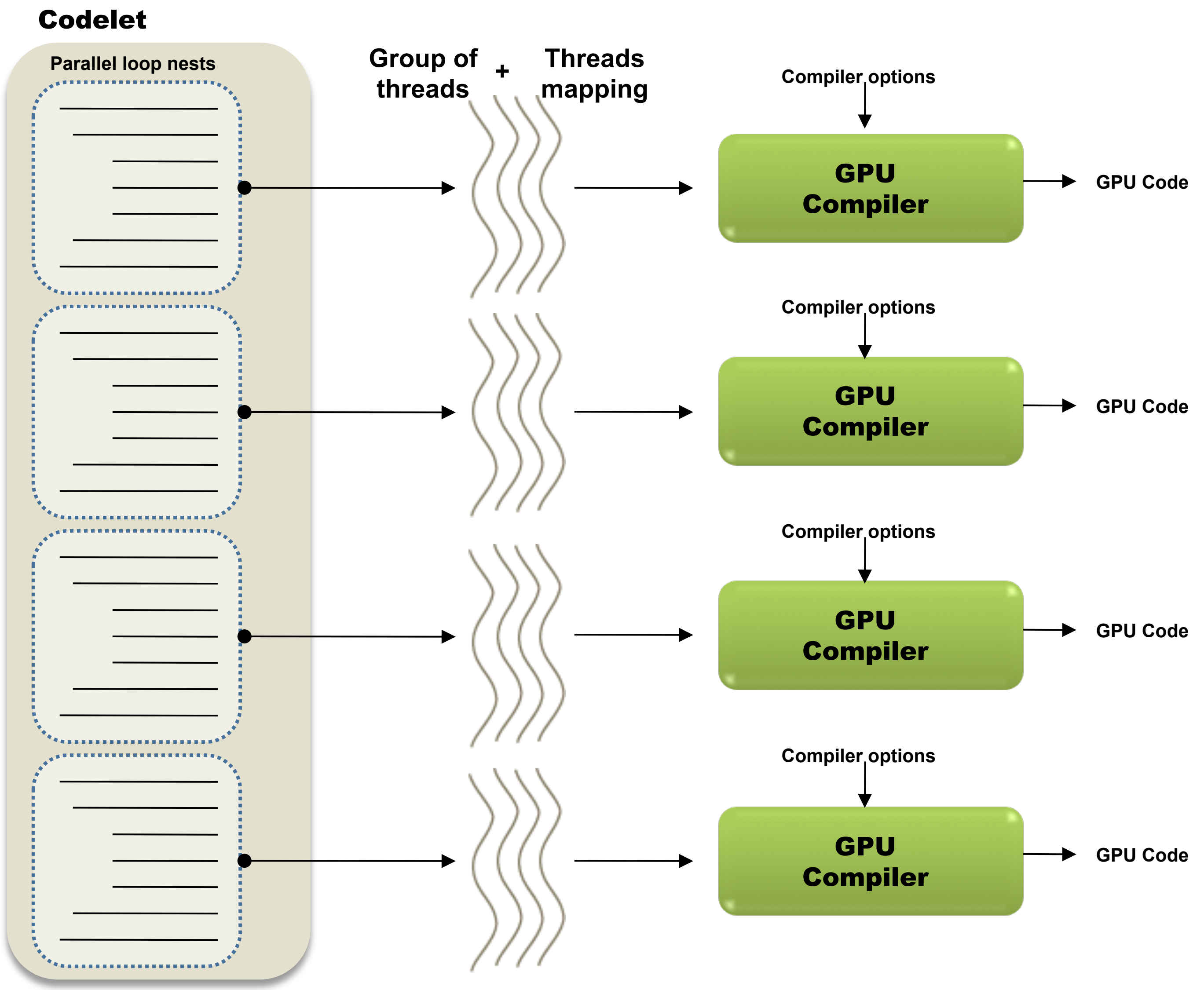
```
int main(int argc, char **argv) {

#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

High Level GPU Code Tuning

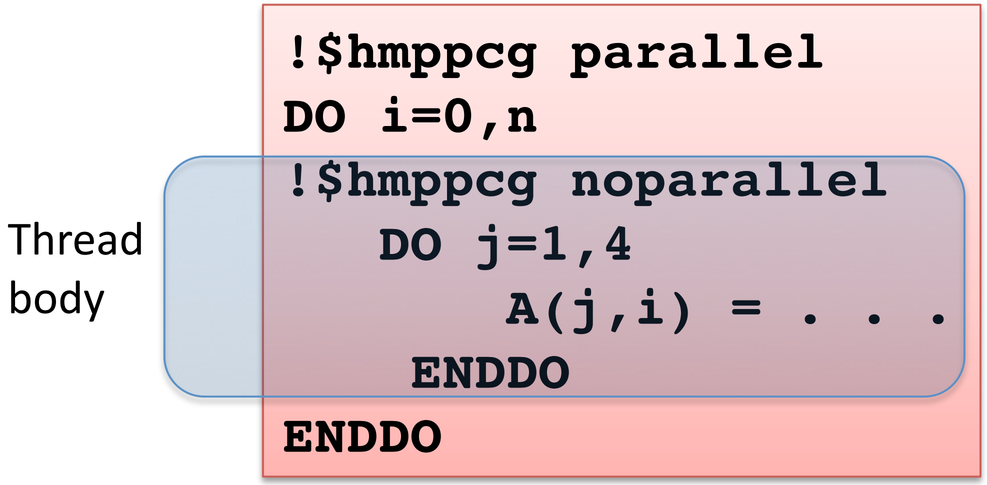
- High level GPU code generation aims at removing low-level implementation details
  - But users still need to understand the underlying process
- Two different sets of tuning issues
  - Global to application: CPU-GPU data transfers
  - Local to threads: GPU kernels
- Many parameters to deal with when exploring the thread optimization space
  - Easier to explore at higher code level
  - Many useful loop transformations
  - Interaction with thread layout on the streaming multiprocessors

High Level Code Generation



Iteration Space to Thread Mapping

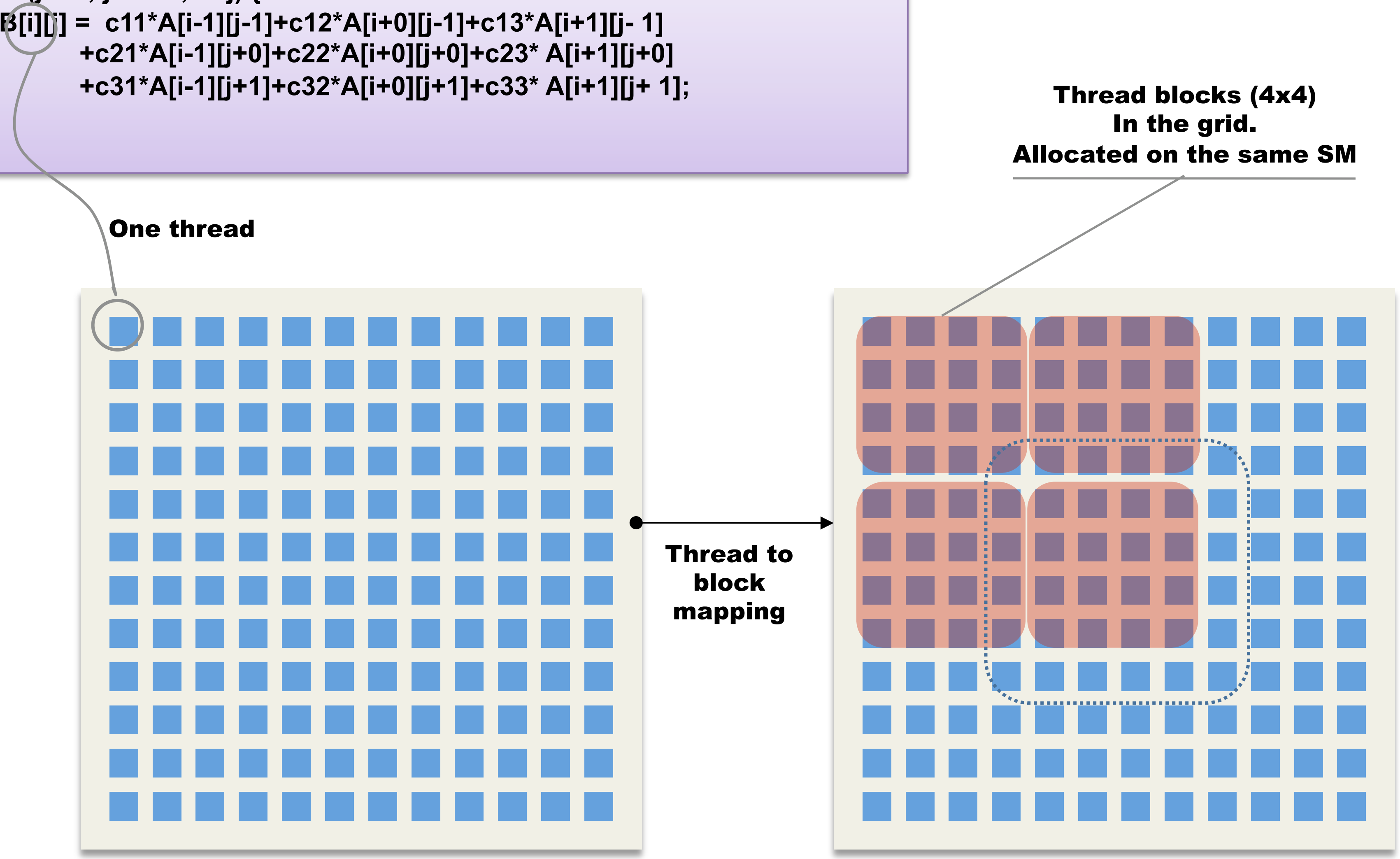
- Parallel iterations spaces are transformed into set of blocks of threads
- Selecting the iteration space fixes the number of threads and their mapping
- Changing to the iteration spaces impact on the threads processing
- Changes to the loop bodies modify the threads computations



```
!$hmppcg parallel
DO i=0,n
!$hmppcg noparallel
   DO j=1,4
       A(j,i) = . . .
    ENDDO
ENDDO
```

Iteration Space Mapping

```
for (i = 1; i < m-1; ++i){
  for (j = 1; j < n-1; ++j) {
    B[i][j] =  c11*A[i-1][j-1]+c12*A[i+0][j-1]+c13*A[i+1][j- 1]
              +c21*A[i-1][j+0]+c22*A[i+0][j+0]+c23* A[i+1][j+0]
              +c31*A[i-1][j+1]+c32*A[i+0][j+1]+c33* A[i+1][j+ 1];
  }
}
```



Iteration Space Mapping and Coalesced Memory Accesses

- `A(i,k)` is well coalesced (`i` on 1st array dimension)
- `C(j,i)` is badly coalesced (`i` on 2nd array dimension)
  - But that does not really matter here
- `B(k,j)` is also well coalesced

```
DO j=1,m    ! 2nd grid dimension
 DO i=1,n   ! 1st grid dimension (the warps)
  tmp = 0
  DO k=1,n
    tmp = A(i,k)*B(k,j)
  ENDDO
  C(j,i) = tmp
 ENDDO
ENDDO
```

A Small Experiment with Loop Permutations

```
# 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 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**

A Simple Tuning Strategy

1. Improve memory coalescing
   - Choose loop order
2. Grid size tuning
   - Choose a grid block size
3. Shared memory (omitted in this presentation)
   - Exploit data reuse (temporal data locality)
4. Register exploitation
   - Unroll (and Jam)

Unroll & Jam

- Unroll & Jam combines blocking and unrolling
  - Can reduce the number of memory accesses
  - Can increase in-register locality
  - But more registers implies less threads/blocks in parallel

```
   $!hmppcg unroll(2), jam(1), noremainder
   DO i=1,n
      $!hmppcg unroll(2), noremainder
      DO j=1,n
           ... calc(i,j)
      ENDDO
   ENDDO
```

```
   DO i=1,n,2
      DO j=1,m,2
        ... calc(i+0,j+0)
        ... calc(i+0,j+1)
        ... calc(i+1,j+0)
        ... calc(i+1,j+1)
      ENDDO
   ENDDO
```

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];
   }
 }
```

S/DGEMM Performance Example (GTX 275)

HMPP SGEMM [1024 x 1024]	397.02 GFLOPS
cuBLAS SGEMM [1024 x 1024]	405.03 GFLOPS
HMPP SGEMM [2048 x 2048]	409.28 GFLOPS
cuBLAS SGEMM [2048 x 2048]	418.81 GFLOPS
HMPP SGEMM [4096 x 4096]	413.64 GFLOPS
cuBLAS SGEMM [4096 x 4096]	424.47 GFLOPS
HMPP DGEMM [1024 x 1024]	82.33 GFLOPS
cuBLAS DGEMM [1024 x 1024]	83.16 GFLOPS
HMPP DGEMM [2048 x 2048]	84.13 GFLOPS
cuBLAS DGEMM [2048 x 2048]	84.94 GFLOPS
HMPP DGEMM [4096 x 4096]	85.08 GFLOPS
cuBLAS DGEMM [4096 x 4096]	85.77 GFLOPS

CGEMM Performance Example (GTX 275)

HMPP	CGEMM [512 x 512]	363.12 GFLOPS
cuBLAS	CGEMM [512 x 512]	324.49 GFLOPS
HMPP	CGEMM [1024 x 1024]	397.35 GFLOPS
cuBLAS	CGEMM [1024 x 1024]	336.85 GFLOPS
HMPP	CGEMM [2048 x 2048]	404.31 GFLOPS
cuBLAS	CGEMM [2048 x 2048]	339.21 GFLOPS
HMPP	CGEMM [4096 x 4096]	409.80 GFLOPS
cuBLAS	CGEMM [4096 x 4096]	340.51 GFLOPS

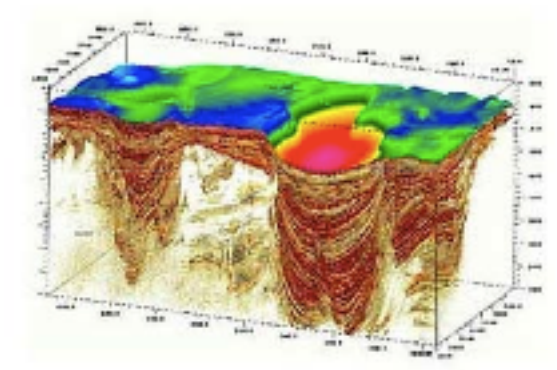
Example (DP) of Impact of the Various Tuning Steps

- Original code = 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
```

Seismic Modeling Application

- Reverse Time Migration modeling at Total
  - Acceleration of critical functions
  - Use HMPP with CUDA
- Data domain decomposition
  - Large data processing
  - One sub-domain running on a node with two GPU cards
- Main issue
  - Optimization of communications between CPUs and GPUs
- Results (Q2/09)
  - 1 GPU-accelerated machine is equivalent to 4.4 CPU machines
  - GPU: 16 dual socket quadcore Hapertown nodes connected to 32 GPUs
  - CPU: 64 dual socket quadcore Hapertown nodes



Conclusion

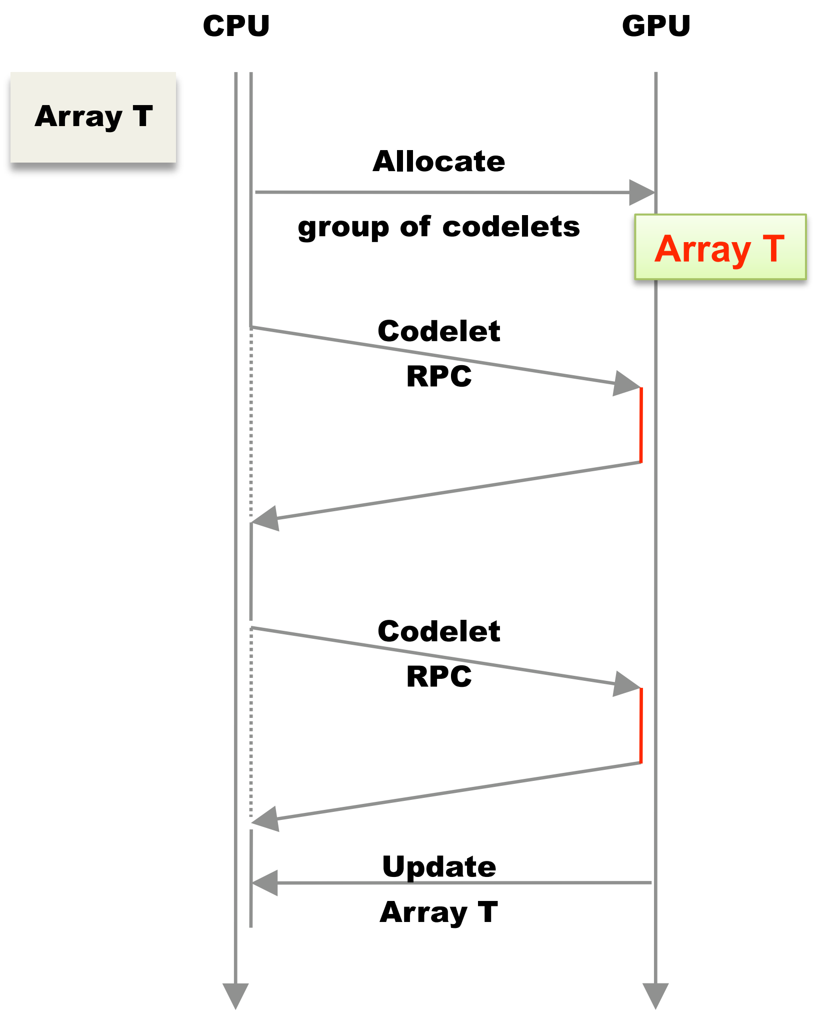
- High level GPU code generation allows many GPU code optimization opportunities
  - Easier to tune programs at high level
- Hardware cannot be totally hidden to programmers
  - e.g. exposed memory hierarchy
  - Efficient programming rules must be clearly stated
- Quantitative decisions as important as parallel programming
  - Performance is about quantity
  - Tuning is specific to a GPU configuration
  - Runtime adaptation is a key feature
    - Algorithm, implementation choice
    - Programming/computing decision

Some Links

- http://www.gpgpu.org/
- http://www.nvidia.com/object/cuda_home.html
- http://www.amd.com/stream/
- http://www.intel.com/technology/visual/microarch.htm
- http://www.khronos.org/

Group of Codelets (V2.0)

- Several callsites grouped in a sequence corresponding to a given device
- Memory allocated for all arguments of all codelets
- Allow for resident data without consistency management



Actual Argument Mapping

- Allocate arguments of various codelets to the same memory area
  - Allow to exploit reuses of argument to reduce communications
  - Close to equivalence in Fortran

```
#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])
{
  ...
}
```

Arguments share the same space on the HWA

HMPP Regions (V2.3)

- Reduce code restructuring when using HMPP
- The codelet is automatically built from the region code

```
#pragma hmpp region
{
  int i;
  for (i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i]*alpha;
  }
}
```

Double Buffering Example

```
…
 for (index=0; index < CHUNKS; index+=2)    {
  #pragma hmpp <simple> f1 callsite, ..., asynchronous
      matvec(N, chunksize, &t3[index*chunksize], t1, &t2[N*index*chunksize]);

   #pragma hmpp <simple> f2 advancedload, … ,asynchronous

   #pragma hmpp <simple> f1 synchronize
   #pragma hmpp <simple> f1 delegatedstore, args[outv]

   #pragma hmpp <simple> f2 callsite, …, asynchronous
     matvec(N, chunksize, &t3[(index+1)*chunksize], t1, &t2[N*(index+1)*chunksize]);

     if (index+2 < CHUNKS) {
        #pragma hmpp <simple> f1 advancedload, args[outv; inm], …,asynchronous
     }

   #pragma hmpp <simple> f2 synchronize
   #pragma hmpp <simple> f2 delegatedstore, args[outv]
}
…
```

chunksize needs tuning
to achieve perfect overlap

Other HMPP Features (V2.2)

- Fallback management when GPU is busy (or unavailable)
- Provide ways to deal with machine specific features
  - e.g. pin memory, …
- Windows and Linux versions
- Seamless integration of GPU libraries such as cuFFT an cuBLAS
- GPU code tuning directives

Exploiting Shared Memory (V2.3)

- Some data are temporarily mapped to the streaming multiprocessors shared memories
- Need to specify a mapping function from a large memory address space to a smaller one
- Strong interaction with unroll (& jam)

```
!$hmppcg block scratch S[BS(i)+2]
do i=2,n-1
!$hmppcg block mapping T1[read|write] S[$1%(BS(i)+2)]
!$hmppcg block update S[:] <- T1[i-1:i+1]
     T2(i) = T1(i) + T(i-1) + T(i+1)
!$hmppcg end block mapping T1
enddo
```

- HMPP also provides a more explicit shared memory management

Miscellaneous

- Other useful loop transformations
  - Loop coalescing
  - Blocking
  - Distribution and fusion
  - Software pipelining
  - …
- Global operations
  - Reductions and Barriers
- Avoiding control flow divergence
- Data access alignment
- …

Tuning Directives List

- Many directives to
  - Express parallelism and reductions
  - Deal with threads grid configuration
  - Deal with register usages
  - Deal with shared/local memory
- Most of loop transformations (permute, unroll, tiling, …)
  - Can be combined to reach a loop structure
    - Loop transformation interpreter for advanced usage

Before

```
!$hmppcg tile I:8,J:16, &
!$hmppcg addtoouter(i){grid blocksize 8x8}
DO I = 1, N
  DO J = 1, N
    C(I, J) = D(I, J)
  ENDDO
ENDDO
```

After

```
!$hmppcg grid blocksize 8x8
DO II = 0, N-1, 8
  DO JJ = 0, N-1, 16
    DO I = II+1, MIN(II+8, N)
      DO J = JJ+1, MIN(JJ+16, N)
        C(I, J) = D(I, J)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Adding Third Party Libraries in HMPP

- Allows to mix user code with library code efficiently on the GPU
  - Avoid CPU-GPU data transfers

- Allows to provide hand-optimized CUDA code

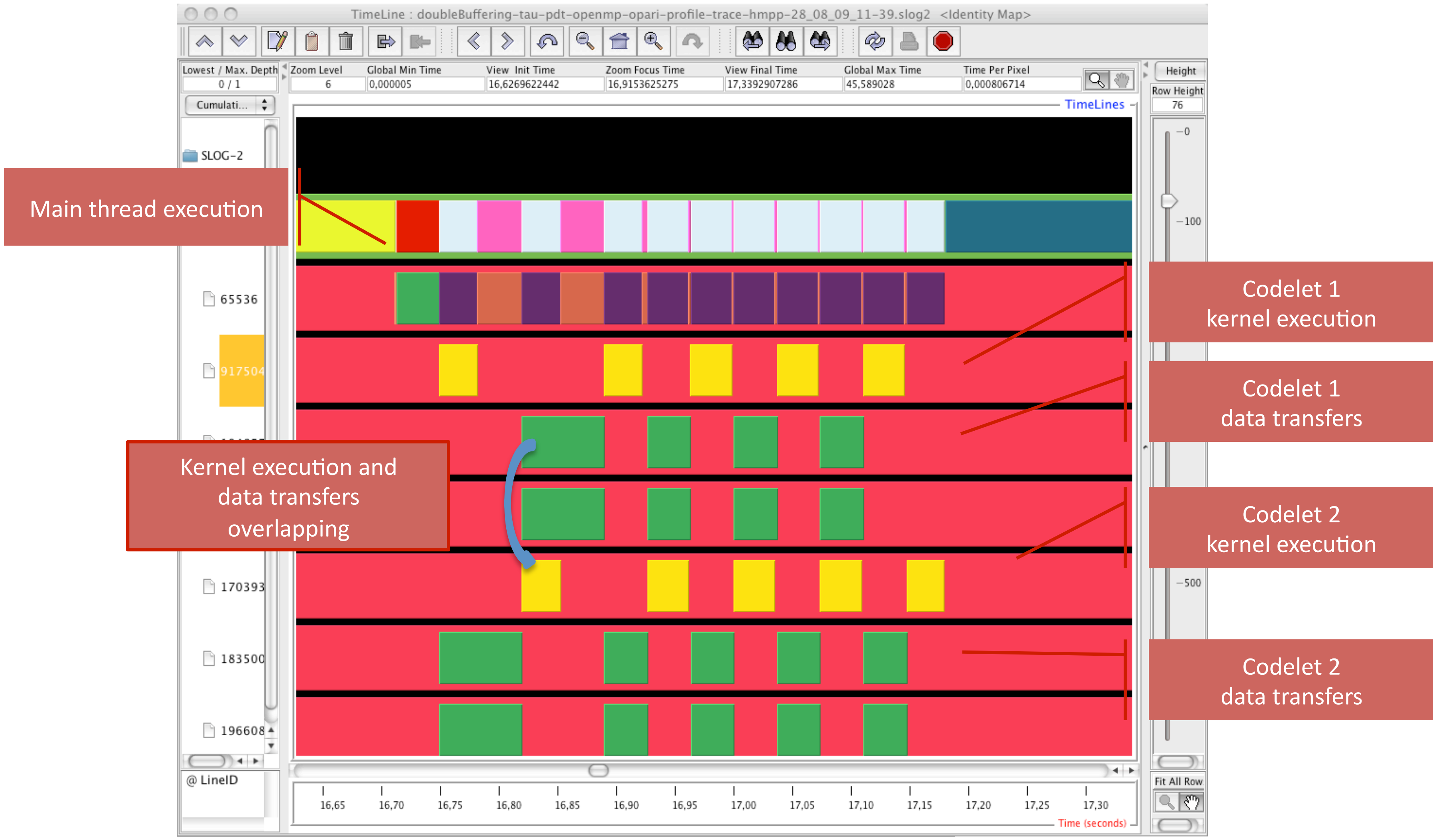
- Users gets a list of already integrated library routines

CULA Integration - Example

```
 /* --------- Execution on GPU with an HMPP codelet and an
             indirect call to the CULA library ------------- */
#pragma hmpp <culaSgeqrf> allocate, args[prepro::a;
        sgeqrf::a].size={M*N}, args[sgeqrf::tau].size={N}

  fprintf(stderr, "Launch HMPP/CULA Computation\n");
  start_measure(&hmppTime);

//USER CODELET
#pragma hmpp <culaSgeqrf> prepro callsite, args[2].noupdate=true
  preprocess(M,N,A,M);

//LIBRARY CODELET
#pragma hmpp <culaSgeqrf> sgeqrf callsite, args[2].noupdate=true
  hmpp_culaSgeqrf(M, N, A, M, TAU, status);

#pragma hmpp <culaSgeqrf> sgeqrf delegatedstore, args[2]

  stop_measure(&hmppTime);
  checkStatus(status);
  fprintf(stderr, "HMPP+CULA time: %gms\n", hmppTime.duration);
```

Dealing with CPU-GPU Communication Tuning: HMPP-TAU

Double buffering example


TimeLine : doubleBuffering-tau-pdt-openmp-opari-profile-trace-hmpp-28_08_09_11-39.slog2 <Identity Map>
Lowest / Max. Depth 0 / 1
Zoom Level 6
Global Min Time 0,000005
View Init Time 16,6269622442
Zoom Focus Time 16,9153625275
View Final Time 17,3392907286
Global Max Time 45,589028
Time Per Pixel 0,000806714
Height
Row Height 76
TimeLines
SLOG-2
65536
917504
170393
183500
196608
@ LineID
16,65
16,70
16,75
16,80
16,85
16,90
16,95
17,00
17,05
17,10
17,15
17,20
17,25
17,30
Time (seconds)
Fit All Row
Main thread execution
Kernel execution and data transfers overlapping
Codelet 1 kernel execution
Codelet 1 data transfers
Codelet 2 kernel execution
Codelet 2 data transfers