CAPS

Innovative tools for a new paradigm

Parallel Hybrid Computing

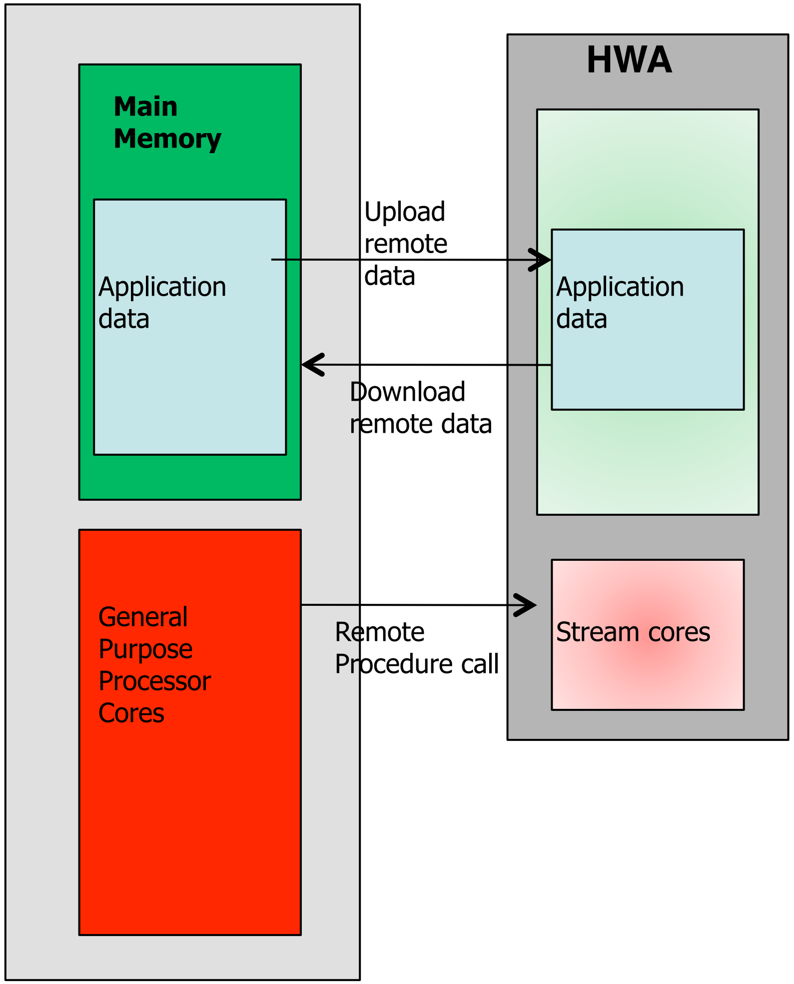
F. Bodin, CAPS Entreprise

Introduction

- Main stream applications will rely on new multicore / manycore architectures
  - It is about performance not parallelism
- Various heterogeneous hardware
  - General purpose cores
  - Application specific cores – GPU (HWA)
- HPC and embedded applications are increasingly sharing characteristics

Manycore Architectures

- General purpose cores
  - Share a main memory
  - Core ISA provides fast SIMD instructions
- Streaming engines / DSP / FPGA
  - 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
  - e.g. one rack SGI ICE Harpertown : 64 CPU nodes (**22** kW) vs one rack SGI ICE GPGPU : 8 Tesla S1070 (32 GPUs) + 16 CPU nodes (**12** kW)



Overview of the Presentation

1. GPUs Programming
2. CUDA
3. OpenCL
4. Miscellaneous Environments
5. HMPP Overview
6. High Level GPU Code Generation

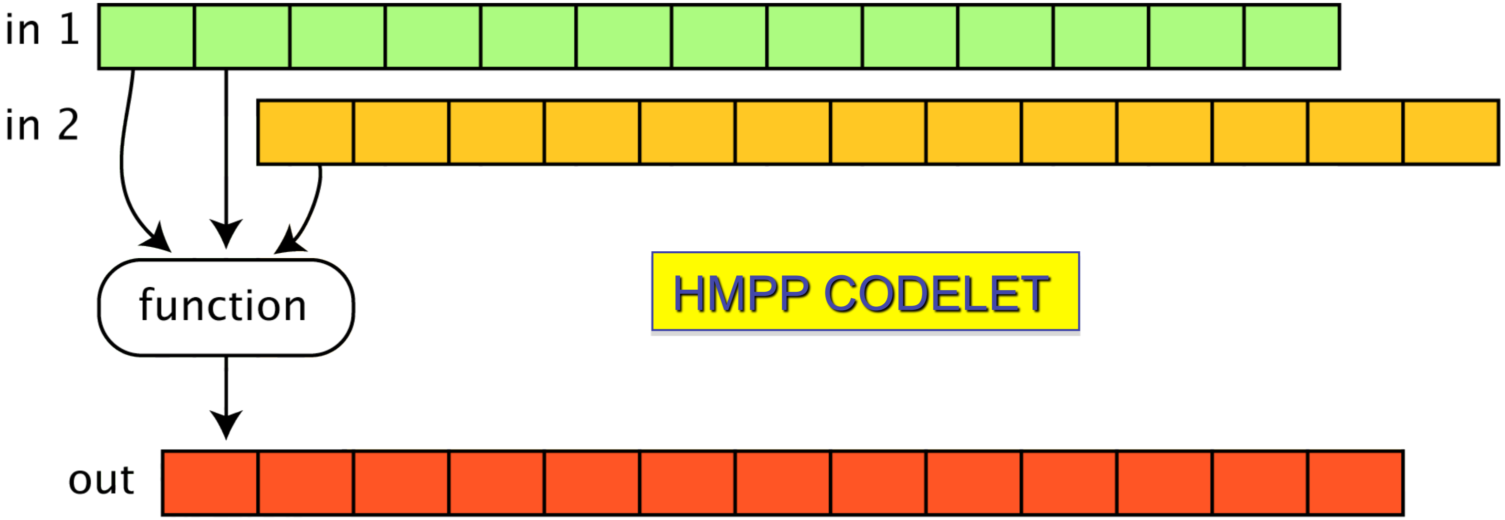
GPUs Programming

Introduction

- GPUs are heavily pipelined and parallel
  - Share many characteristics with vector machines
- Stream programming is well suited
  - But memory hierarchy is exposed
- Require to rethink the computation organization/ algorithm
- See GPGPU (http://gpgpu.org)

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



HMPP CODELET

A Few Stream Languages

- Brook+
  - Mostly AMD
- CUDA Nvidia
  - NVIDIA Only
- RapidMind
  - Cell, AMD, …
- OpenCL

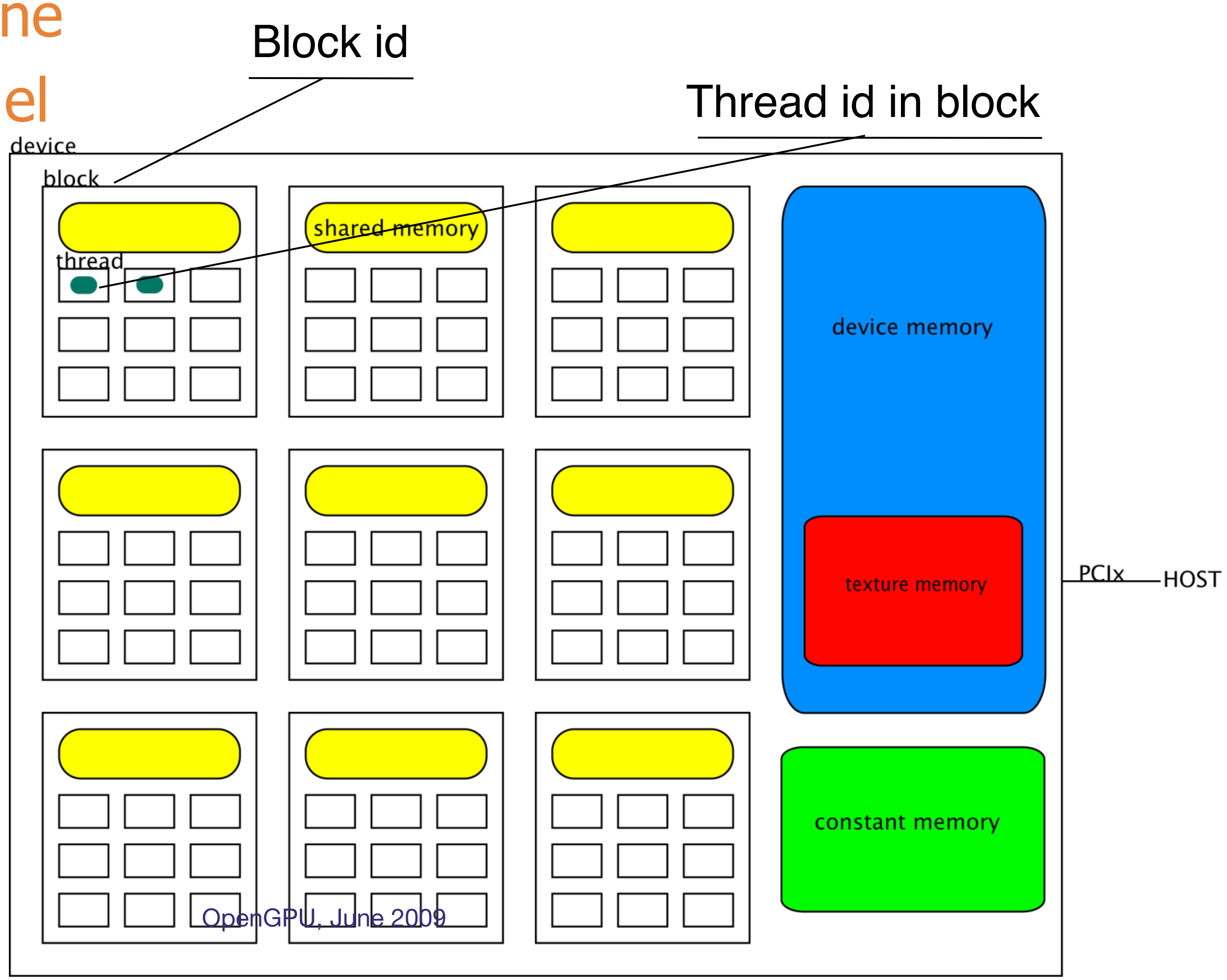
CUDA

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 Grid and Blocks

- GPUs need 1000s of threads to be efficient
  - Highly pipeline
  - Highly parallel
- ~SIMD
- Many memories



Block id

Thread id in block

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)

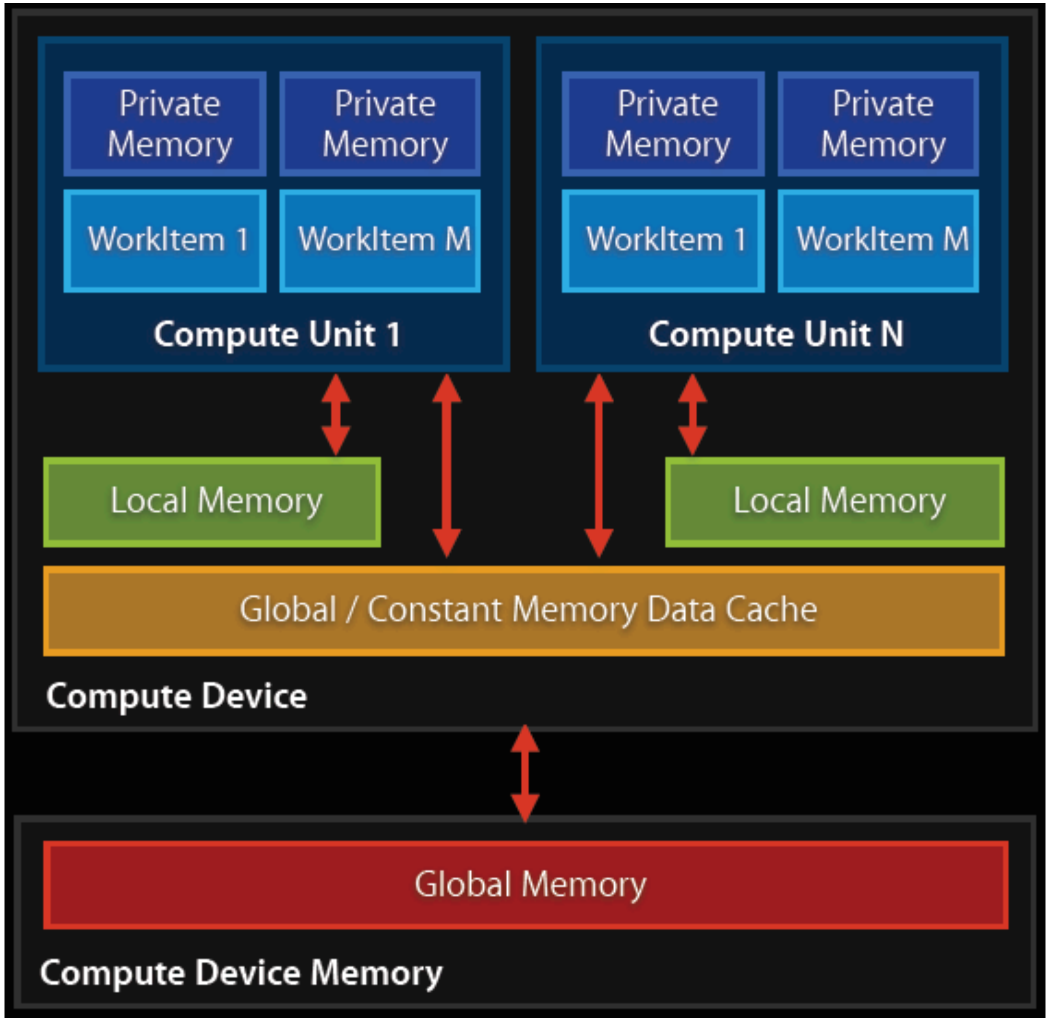
```
  /* initialize CUDA device */
  CUT_DEVICE_INIT()
  …
  /* allocate arrays on device */
  float *gpu_t1 = NULL;
  float *gpu_t2 = NULL;
  float *gpu_t3 = NULL;
  cudaMalloc((void**) &gpu_t1, n*sizeof(float));
  cudaMalloc((void**) &gpu_t2, n*sizeof(float));
  cudaMalloc((void**) &gpu_t3, n*sizeof(float));
 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

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

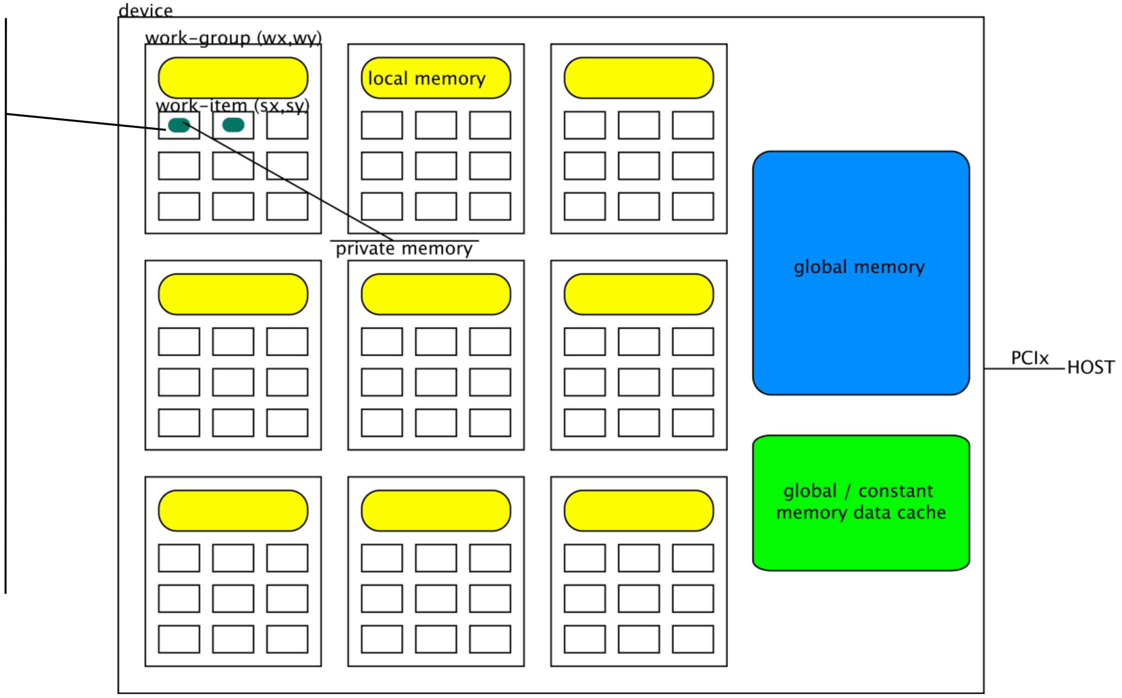
Platform Layer API& Runtime API

- 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;
}
```



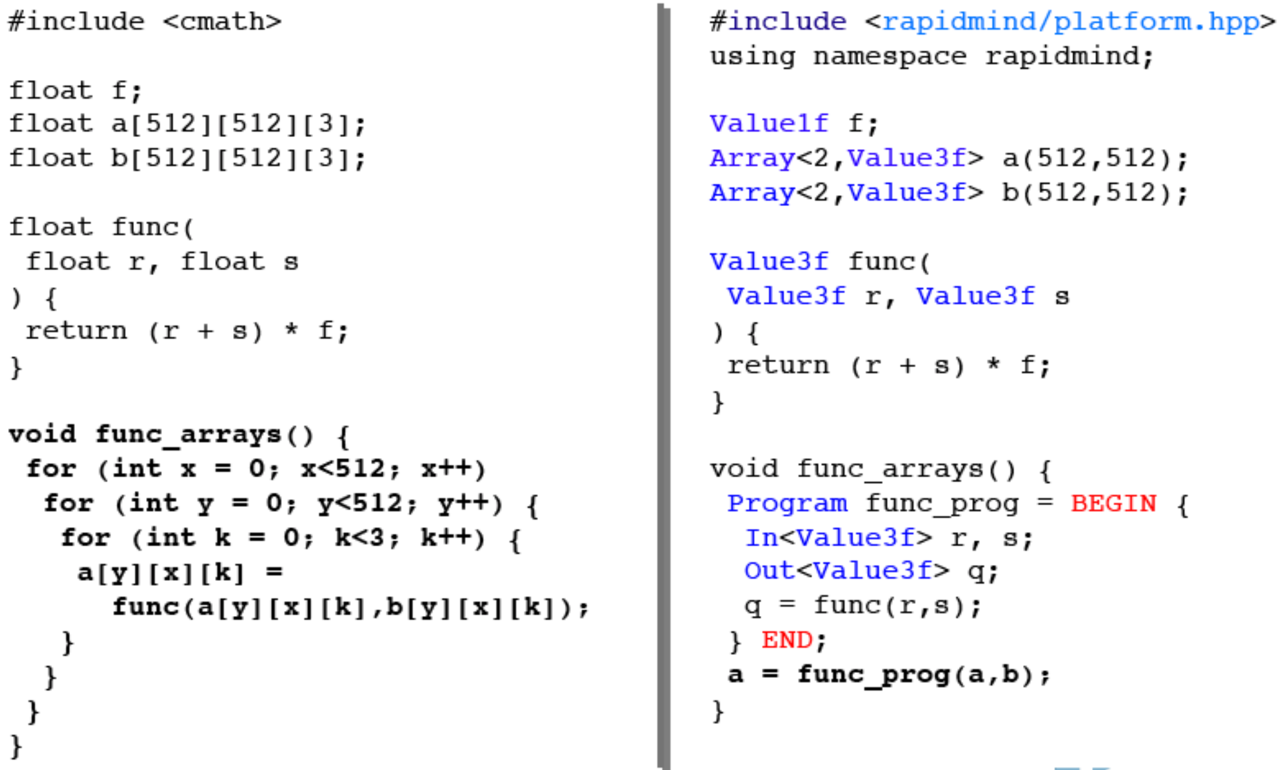
Miscellaneous Environments

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



```
#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);
}
```

From RapidMind

HMPP

Introduction

- Hybrid Multicore Parallel Programming (HMPP)
    - Focus on programming multicore nodes, not on dealing with large scale parallelism
- Directives based programming environment
- Centered on the codelet / pure function concept
- *Focus on CPU – GPU communications optimizations*
- Complementary to OpenMP and MPI

Directives Based Approach for Hardware Accelerators (HWA)

- 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
- Avoid exit cost

What is Missing in OpenMP for HWA

- Remote Procedure Call (RPC) on a HWA
  - Code generation for GPU, …
  - Hardware resource management
- Dealing with non shared address space
  - Explicit communications management to optimize the data transfers between main the CPU and the HWA

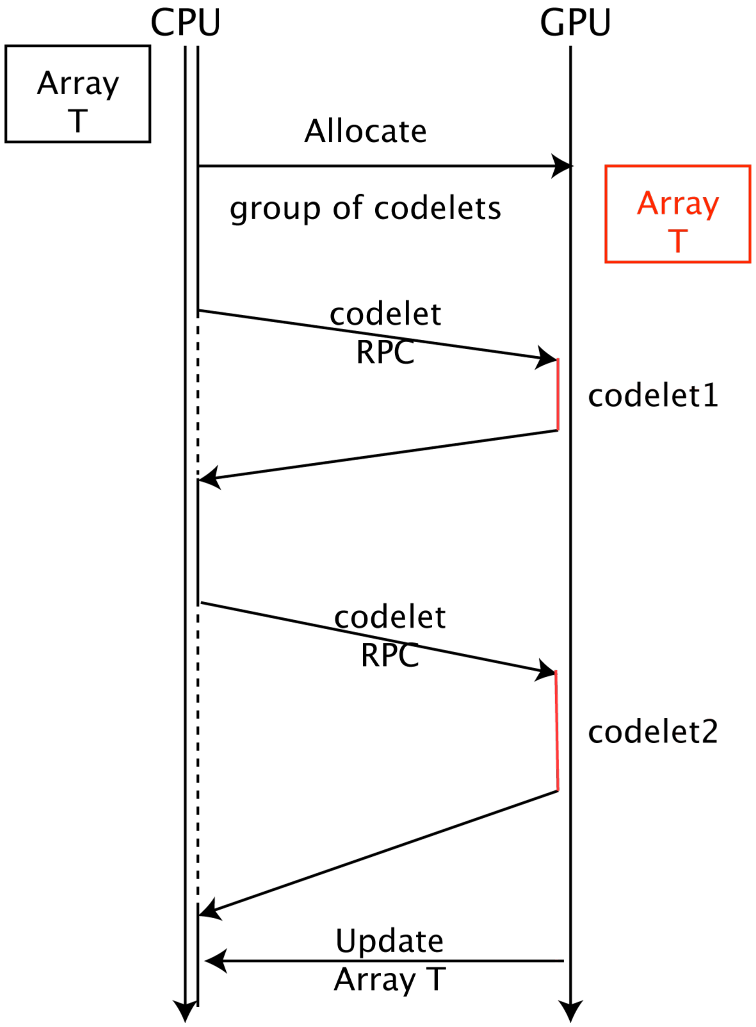
HMPP1.5 Simple Example

```
#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 );
 }
```

```
#pragma hmpp label codelet, target=CUDA:BROOK, args[v1].io=out
#pragma hmpp label2 codelet, target=SSE, args[v1].io=out, cond=“n<800“
void MyCodelet(int n, float v1[n],  float v2[n], float v3[n])
{ int i;
  for (i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i];
  }
}
```

Group of Codelets (HMPP 2.0)

- Several callsites grouped in a sequence corresponding to a given device
- Memory allocated for all arguments of all codelets
- Allow for resident data but no consistency management



Optimizing Communications

- Exploit two properties
  - Communication / computation overlap
  - Temporal locality of RPC parameters

- Various techniques
  - Advancedload and Delegatedstore
  - Constant parameter
  - Resident data
  - Actual argument mapping

Advancedload Directive

- Avoid reloading constant data

```
int main(int argc, char **argv) {
…
#pragma hmpp simple advancedload, args[v2], const
  for (j=0; j<n; j++){
#pragma hmpp simple callsite, args[v2].advancedload=true
    simplefunc1(n,t1[j], t2, t3[j],  alpha);
   }
#pragma hmpp label release
…
}
```

`t2` is not reloaded at each loop iteration

Actual Argument Mapping

- Allocate arguments of various codelets to the same memory space
  - 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

High Level GPU Code Generation

Introduction

- HMPP allows direct programming of GPU in C and Fortran
- GPU Fortran/C code tuning similar to CPU tuning code but strategy differs a lot
- Fortran/C coding easier and does not require to learn all the intricacies of GPUs specific languages
- How to deal with multiple code/binary versions
  - Rollback CPU codes must be optimized too

Tuning GPU Codes

- GPU micro-architectures impact heavily on tuning
- Performance difference between bad and right may be huge
- Not exactly the usual tricks
  - e.g. Thread conscious optimizations
  - e.g. Memory coalescing important

Heterogeneous Tuning Issue Example

```
#pragma hmpp astex_codelet__1 codelet &
#pragma hmpp astex_codelet__1 , args[c].io=in &
#pragma hmpp astex_codelet__1 , args[v].io=inout &
#pragma hmpp astex_codelet__1 , args[u].io=inout &
#pragma hmpp astex_codelet__1 , target=CUDA &
#pragma hmpp astex_codelet__1 , version=1.4.0
void astex_codelet__1(float u[256][256][256], float v[256][256][256], float c[256][256][256],
                      const int K, const float x2){
 astex_thread_begin:{
  for (int it = 0 ; it < K ; ++it){
    for (int i2 = 1 ; i2 < 256 - 1 ; ++i2){
      for (int i3 = 1 ; i3 < 256 - 1 ; ++i3){
        for (int i1 = 1 ; i1 < 256 - 1 ; ++i1){
          float  coeff = c[i3][i2][i1] * c[i3][i2][i1] * x2;
          float  sum = u[i3][i2][i1 + 1] + u[i3][i2][i1 - 1];
          sum += u[i3][i2 + 1][i1] + u[i3][i2 - 1][i1];
          sum += u[i3 + 1][i2][i1] + u[i3 - 1][i2][i1];
          v[i3][i2][i1] = (2. - 6. * coeff) * u[i3][i2][i1] + coeff * sum - v[i3][i2][i1];
        }
      }
    }
    for (int i2 = 1 ; i2 < 256 - 1 ; ++i2){
      for (int i3 = 1 ; i3 < 256 - 1 ; ++i3){
        for (int i1 = 1 ; i1 < 256 - 1 ; ++i1{

        . . . . .
 }astex_thread_end:;
}
```

Need interchange
If aims at NVIDIA GPU

Examples of Kernel Tuning Rules

- **Rule 1**: Create a sufficient amount of independent tasks (i.e. some 1D or 2D loop nests with hundreds or even thousands of independent iterations in each dimension).
- **Rule 2**: Maximize the coalescing of memory accesses (i.e. the threads in a given half-warp should have a good spatial locality).
- **Rule 3**: Reduce the number of accesses to the global memory.
- **Rule 4**: Use aligned coalescent memory accesses when possible.
- **Rule 5**: Limit the resources (registers, shared memory, ...) used by each thread to allow more warps to be executed in parallel on each multiprocessor.
- **Rule 6**: Increase the amount of concurrent memory accesses to maximize the use of the memory bus.
- **Rule 7**: Tune the *gridification* and the CUDA block size. This can affect in good or in bad any of the rules above.

Conclusion

- Multicore/Manycore ubiquity is going to have a large impact on software industry
  - New applications but many new issues
  - It is not GPU versus CPU but how to combine them efficiently
- Will one parallel model fit all?
  - Surely not but multi languages programming should be avoided
  - Directive based programming is a safe approach
  - Ideally OpenMP will be extended to HWA
- Toward Adaptative Parallel Programming
  - Compiler alone cannot solve it
  - Compiler must interact with the runtime environment
  - Programming must help expressing global strategies / patterns
  - Compiler as provider of basic implementations
  - Offline-Online compilation has to be revisited