Parallel Hybrid Computing
F. Bodin, CAPS Entreprise
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**
Multicore/Manycore Workload

- Multiple applications sharing the hardware
  - Multimedia, game, encryption, security, health, ...

- Unfriendly environment with many competitions
  - Global resource allocation, no warranty on availability
  - Must be taken into account when programming/compiling

- Applications cannot always be recompiled
  - Most applications are distributed as binaries

- A binary will have to run on many platforms
  - Forward scalability or “write once, run faster on new hardware”
  - Loosing performance is not an option
The Past of Parallel Computing, the Future of Manycores?

- **The Past**
  - Scientific computing focused
  - Microprocessor or vector based, homogeneous architectures
  - Trained programmers willing to pay effort for performance
  - Fixed execution environments

- **The Future**
  - New applications (multimedia, medical, ...)
  - Thousands of heterogeneous systems configurations
  - Unfriendly execution environments
Overview of the Presentation

1. GPUs Programming
2. CUDA
3. OpenCL
4. Miscellaneous Environments
5. An Overview of Parallel Heterogeneous Computing
6. HMPP Overview
7. High Level GPU Code Generation
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
A Few Stream Languages

- Brook+
  - Mostly AMD
- CUDA Nvidia
  - NVIDIA Only
- RapidMind
  - Cell, AMD, ...
- OpenCL
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
#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);
/* initialize CUDA device */
CUDA_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
Hybrid Parallel Computing, June 2009
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
A kernel is executed by the work-items

```c
// 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

```c
__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];
}
```
Miscellaneous Environments
kernel void sum(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

```cpp
#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++)
      a[y][x][k] =
        func(a[y][x][k], b[y][x][k]);
}
}
```

```cpp
#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);
}
```
An Overview of Heterogeneous Parallel Computing
Introduction

- Programming heterogeneous platforms implies to take into account all parallelism levels
  - And all micro-architectures characteristics

- Address spaces are not shared between GPU and CPU
  - Data distribution/replication is necessary

- GPUs are not time-shared devices
  - Resource allocation is an issue to consider
Multiple Parallelism Levels

- Amdahl’s law is forever, all levels of parallelism need to be exploited
  - Hybrid parallelism needed
- Programming various hardware components of a node cannot be done separately
Programming Multicores/Manycores

- Physical architecture oriented
  - Shared memory architectures
    - OpenMP, CILK, TBB, automatic parallelization, vectorization...
  - Distributed memory architectures
    - Message passing, PGAS (Partition Global Address Space), ...
  - Hardware accelerators, GPU
    - CUDA, OpenCL, Brook+, HMPP, ...

- Different styles
  - Libraries
    - MPI, pthread, TBB, SSE intrinsic functions, ...
  - Directives
    - OpenMP, HMPP, ...
  - Language constructs
    - UPC, Cilk, Co-array Fortran, UPC, Fortress, Titanium, ...
Multi (languages) programming

- Happens when programmers need to deal with multiple programming languages
  - E.g. Fortran and Cuda, Java and OpenCL, ...

- Multiprogramming impacts on
  - Programmer’s expertise
  - Program maintenance and correctness
  - Long-term technology availability

- Performance programming versus domain specific programming
  - Libraries, parallel components to be provided to divide the issues
Manycore = Numerous Configurations

- Heterogeneity brings a lot of configurations
  \[ \text{Proc.} \times \text{Nb Cores} \times \text{HWA} \times \text{Mem. Sys.} = 1000^s \text{ of configurations} \]
- Code optimization strategy may differ from one configuration to another

Is it possible to make a single (a few) binary that will run efficiently on a large set of configurations?
Asymmetric Behavior Issue

- Cannot assume that all cores with same ISA provide equal performance
  - Core frequency/voltage throttling can change computing speed of some cores
    - e.g. Nehalem “turbo mode”
  - Simple (in order) versus complex (out-of-order) cores
  - Data locality effects
  - ...

*How to deal with non homogeneous core behavior?*
Manycore = Multiple \( \mu \)-Architectures

- Each \( \mu \)-architecture requires different code generation/optimization strategies
  - Not one compiler in many cases
- High performance variance between implementations
  - ILP, GPCore/TLP, HWA
- Dramatic effect of tuning
  - Bad decisions have a strong effect on performance
  - Efficiency is very input parameter dependent
  - Data transfers for HWA add a lot of overheads

*How to organize the compilation flow?*
CAPS Compiler Flow for Heterogeneous Targets

- Dealing with various ISAs
- Not all code generation can be performed in the same framework
Can the Hardware be Hidden?

- Programming style is usually hardware independent but
  - Programmers need to take into account available hardware resources

- *Quantitative* decisions as important as parallel programming
  - Performance is about quantity
  - Tuning is specific to a configuration

- Runtime adaptation is a key feature
  - Algorithm, implementation choice
  - Programming/computing decision
Varying Available Resources

- Available hardware resources are changing over the execution time
  - Not all resources are time-shared, e.g. a HWA may not be available
  - Data affinity must be respected

_How to ensure that conflicts in resource usage will not lead to global performance degradation?_
Competition for Resources

AThread_0  AThread_1

C0  C1

L1 Cache  L1 Cache

L2 Cache

shared bus

Main Memory

BThread_0  AThread_2  BThread_1

C2  C3

L1 Cache  L1 Cache

L2 Cache

ARPC_0  ARPC_1

HWA (GPU, FPGA, ...)

Application A  Application B

Hybrid Parallel Computing, June 2009
OpenMP in Unfriendly Environment

- OpenMP programs performance is strongly degraded when sharing resources
  - Example with NAS parallel benchmark, 2 cores, one *rogue* application using one of the cores
  - Best loop scheduling strategy not identical on loaded or unloaded machine
Peak Performance is Not the Goal

- Maximizing the Return on Investment

Hybrid Parallel Computing, June 2009
Difficult Decisions Making with Alternative Codes (Multiversioning)

- Various implementations of routines are available or can be generated for a given target
  - CUBLAS, MKL, ATLAS, ...
  - SIMD instructions, GPcore, HWA, Hybrid

- No strict performance order
  - Each implementation has a different performance profile
  - Best choice depends on platform and runtime parameters

- Decision is a complex issue
  - How to produce the decision?
Illustrating Example:
Dealing with Multiple BLAS Implementations

- Runtime selection of DGEMM in High Performance Linpack
  - Intel(R) Xeon(R) E5420 @ 2.50GHz
  - CUBLAS - Tesla C1060, Intel MKL

- Three binaries of the application
  - Static linking with CUBLAS
  - Static linking with MKL
  - Library mix with selection of routine at runtime
    - Automatically generated using CAPS tooling

- Three hardware resource configurations
  - GPU + 1, 2, and 4 cores used for MKL
Performance Using One Core

- Performance in Gigaflops
- 4 problem sizes: 64, 500, 1200, 8000

Problem Size

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Cublas</th>
<th>MKL</th>
<th>Dyn. Sel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.07</td>
<td>1.3</td>
<td>1.4</td>
</tr>
<tr>
<td>500</td>
<td>1.2</td>
<td>7.3</td>
<td>6.5</td>
</tr>
<tr>
<td>1200</td>
<td>4.4</td>
<td>8.1</td>
<td>8</td>
</tr>
<tr>
<td>8000</td>
<td>23</td>
<td>9</td>
<td>23.3</td>
</tr>
</tbody>
</table>
Performance Using Two Cores

Performance (GFLOPS) vs Problem Size

- Cublas
- MKL
- Dyn. Sel.

Hybrid Parallel Computing, June 2009
Performance Using Four Cores

![Graph showing performance using four cores with problem sizes ranging from 64 to 8000. The graph compares performance across different libraries: Cublas, MKL, and Dynamic Selection. Performance metrics are given in GFLOPS.]
The Challenges

- Programming
  - Medium
- Resources management
  - Medium
- Application deployment
  - Hard
- Portable performance
  - Extremely hard
Research Directions

- **New Languages**
  - X10, Fortress, Chapel, PGAS languages, OpenCL, MS Axum, ...

- **Libraries**
  - Atlas, MKL, Global Array, Spiral, Telescoping languages, TBB, ...

- **Compilers**
  - Classical compiler flow needs to be revisited
  - Acknowledge lack of static performance model
  - Adaptative code generation

- **OS**
  - Virtualization/hypervisors

- **Architectures**
  - Integration on the chip of the accelerators
    - AMD Fusion, ...
  - Alleviate data transfers costs
    - PCI Gen 3x, ...

Key for the short/mid term
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
#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

```c
int main(int argc, char **argv) {
    ...
    #pragma hmpp simple advancedload, args[v2],
    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

```c
#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

```c
#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];
                    }
                }
            }
        }
    }

    astex_thread_end:;
}
```

Need interchange
If aims at NVIDIA GPU

Hybrid Parallel Computing, June 2009
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.
Application Example
Introduction

- Many real applications can achieve performance if
  - Computation kernels can be efficiently implemented
  - Communication cost is reduced to a minimum by exploiting data locality available in the applications
    - Temporal locality & partial data transfer
    - Overlapping communications and computations

- A Fortran example
  - Seismic application (RTM) at Total
  - Attend Rached Abdelkhalek, « Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster » at AASC’09 for more recent results.
Seismic Modeling Application

- **Reverse Time Migration modeling**
  - 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
  - Bottleneck is MPI communication

- **Latest results**
  - 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
Overlapping Kernel Execution with Data Transfers

- Use asynchronicity to hide data transfers between CPU and GPU
  - Divide sub-domain computations in streams
CPU Versus GPU
(Domain size varies)

Lower is better

1 GPU vs 8 cores
speedup 3.3

Hybrid Parallel Computing, June 2009
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

Hybrid Parallel Computing, June 2009