Many-Core Programming with HMPP 3.0

Write once, deploy many(-core)

Jean-Charles VASNIER
Introduction

• **Computing power comes from parallelism**
  o Hardware (frequency increase) to software (parallel codes) shift
  o Driven by energy consumption: heterogeneity is source of efficiency

• **Context of fast moving hardware targets**
  o e.g. fast GPU improvements (RT and HW), new massively parallel CPU
  o Write codes that will last many architecture generations

• **Keeping a unique version of the code, preferably mono-language, is a necessity**
  o Reduce maintenance cost
  o Directive-based approaches suitable
  o Preserve code assets

• **Addressing many-core programming challenge implies**
  o Massively parallel algorithms
  o New development methodologies / code architectures
  o New programming tools
Agenda

1. About many-cores
2. Some principles for many-core programming
3. Methodology to migrate legacy codes
4. Many-core programming with HMPP 3.0
5. HMPP Wizard
Many-Cores

- Massively parallel devices
  - Thousands of threads needed

Data/stream/vector parallelism to be exploited by GPUs e.g. CUDA / OpenCL

CPU and GPUs linked with a PCIx bus

Message passing

Multi-threading
Software Main Driving Forces

• **ALF - Amdahl’s Law is Forever**
  - A high percentage of the execution time has to be parallel
  - Many algorithms/methods/techniques will have to be reviewed to scale

![Diagram of serial and parallel execution]

• **Data locality is expected to be the main issue**
  - Limit on the speed of light
  - Moving data will always suffer latency
Addressing many-cores
• A directive-based approach for many-core
  o CUDA, OpenCL and soon Intel MIC, ...

```c
#pragma hmpp f1 codelet
myfunc(...){
  ...
  for()
  for()
  for()
  ...
  ...
}

main(){
  ...
  #pragma hmpp f1 callsite
  myfunc(V1[k],V2[k]);
  ...
}
```
Some Principles for Many-Core Programming
Express Parallelism, not Implementation

- Rely on code generation for implementation details
  - Usually not easy to go from a low level API to another low level one
  - Tuning has to be possible from the high level
  - But avoid relying on compiler advanced techniques for parallelism discovery, ...
  - You may have to change the algorithm!

- An example with HMPP

```c
#pragma hmppcg gridify(j,i)
#pragma hmppcg unroll(4), jam(2)
for( j = 0 ; j < p ; j++ ) {
  for( i = 0 ; i < m ; i++ ) {
    for (k = ...) { ...}
    vout[j][i] = alpha * ...;
  }
}
```
Exposing Massive Parallelism

• Do not hide parallelism with complex coding structure
  o Data structure aliasing, …
  o Deep routine calling sequences
  o Separate concerns (functionality coding versus performance coding)

• Data parallelism when possible
  o Simple form of parallelism, easy to manage
  o Favor data locality
  o But sometimes too static

• Kernels level
  o Expose massive parallelism
  o Ensure that data affinity can be controlled
  o Make sure it is easy to tune the ratio vector / thread parallelism
Data Structure Management

• Data locality
  o Makes easy to move from one address space to another one
  o Makes easy to keep data coherency
• Do not waste memory
  o Memory per core ratio is not improving
• Choose simple data structures
  o Enable vector/SIMD computing
  o Use library friendly data structures
  o May come in multiple forms, e.g. sparse matrix representation
• For instance consider “data collections” to deal with multiple address spaces or multiple devices or parts of a device
  o Gives a level of adaptation for dealing with heterogeneity
  o Load distribution over the different devices is simple to express
Debugging Issues

• Keep code debug-able.

• Keep serial semantic
  o For instance, implies keeping serial libraries in the application code
  o Directives-based programming makes this easy

• Ensure validation is possible even with rounding errors
  o Reductions, …
  o Aggressive compiler optimizations

• Use defensive coding practices
  o Events logging, parameterize parallelism, add synchronization points, …
  o Use debuggers (e.g. Allinea DDT)
Dealing with Libraries

• Library calls can usually only be partially replaced
  o No one to one mapping between libraries (e.g. BLAS, FFTW, CuFFT, CULA, LibJacket)
  o No access to all code (i.e. avoid side effects)
  o Don’t create dependencies on a specific target library as much as possible
  o Still want a unique source code

• Deal with multiple address spaces / multi-GPU
  o Data location may not be unique (copies)
  o Usual library calls assume shared memory
  o Library efficiency depends on updated data location (long term effect)

• Libraries can be written in many different languages
  o CUDA, OpenCL, HMPP, etc.

• There is not one binding choice depending on applications/users
  o Binding needs to adapt to uses depending on how it interacts with the remainder of the code
  o Choices depend on development methodology
A Methodology for Legacy Code Migration
Legacy Codes Migration Challenges

• Mastering migration cost
  o Ensuring an adequate return on investment
  o Minimizing risk as well as manpower

• Producing code that will last many architecture generations
  o It is safe to assume that the node architecture may change with the renewal of the computer

• Writing developer friendly code
  o Application developers may not be multicore / accelerator / parallelism savvy
  o Once ported, the application still needs to evolve

• Keeping a unique version, preferably mono-language, of the codes
  o Reduce maintenance cost

• Able to use libraries
  o No one-to-one replacement (e.g. FFT libraries)
  o Must interact with non library accelerated kernels
Dealing with Legacy Codes

THE CRITICAL STEP

Define your Parallel Project
- 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

Port your Application on Many-core
- Optimize CPU code
  - Optimize application SIMT parallelism
  - Parallelize for Many-core
  - Validate execution

Optimize Your Many-core Application
- Reduce data transfers
- Optimize kernel execution
- Provide feedback to application programmers for improving algorithm data structures/
- Consider multiple devices

Hotspots
- Hours to Days

Parallelization
- Days to Weeks

Tuning
- Weeks to Months

Many-core operational application with known potential

Phase 1
A corporate project
- Purchasing Department
- Scientists
- IT Department

Phase 2
Go / No Go for GPU Target

Go

• Dense hotspots
• 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
Many-Core Programming with HMPP 3.0
Scope of HMPP 3.0 Programming

- Remote procedure calls (RPCs) on accelerator devices
  - Parallel loop nests to exploit multiple compute units

```c
main(){
    ...
    #pragma hmpp f1 callsite
    myfunc(...){
        ...
        for()
        for()
        for()
        ...
        ...
    }
}
```

```c
#pragma hmpp f1 codelet
myfunc(...){
    ...
    for()
    for()
    for()
    ...
    ...
}
```

January 2012
HMPP Comes in 3 Parts

- A set of directives to program hardware accelerators
  - Drive your HWAs, manage transfers

- A complete toolchain to build manycore applications
  - Build your hybrid application

- A runtime to adapt to platform configuration
  - With its API
HMPP Overview

• C and Fortran GPU programming directives
  o Define and execute GPU-accelerated versions of code
  o Optimize CPU-GPU data movement
  o Complementary to OpenMP and MPI

• A source-to-source hybrid compiler
  o Generates CUDA and OpenCL kernels
  o Works with standard compilers and target tools
  o Tuning directives to optimize GPU kernels

• A runtime library
  o Allocates and manages computing resources
  o Dispatches computations on CPU and GPU cores
  o Scales to multi-GPUs systems
HMPP Compilation Paths

• HMPP drives the whole compilation
  
  o Host application compilation
    • HMPP runtime is linked to the host part of the application
  
  o Codelet production
    • Target code is produced
    • A dynamic library is built

$ hmpp gcc myProgram.c
HMPP Directives Drive Hybrid Applications

- Directives
- HMPP application
- HMPP Runtime
  - Codelet
  - HWA
  - Data

HMPP-specific code generation
What’s New in HMPP 3.0?

- **Dynamic data management mechanism**
  - Mirrors identified by their host address
  - Simplifies management of data with less directives

- **Multi-device programming**
  - Exploit multiple devices in one compute node
  - Distribute collections of data over multiple devices

- **New run-time API**
  - Three bindings for C, C++ and Fortran 90-2003
  - Low level OpenCL style programming with OpenCL/CUDA kernel generation

- **Open library integration system**
  - CPU and GPU libraries coexist in same binary (proxy mechanism)
  - Data sharing between HMPP user codelets and libraries
  - User can write their own HMPP proxies
  - Proxies provided for cuBLAS, CULA, cuFFT, keeping CPU API.
Step One: Find Hot Spots

void derive(int nx, double _Complex ...) {
    int i;
    for (i=1; i<nx/2; ++i) {
        wrkq[i] = (0+I-1) * wrkq[i] * cf;
    }
    wrkq[0] = 0.0+I*0;
    wrkq[nx/2] = 0.0+I*0;
}

...  
pr2c = fftw_plan_dft_r2c_1d(n, idata_real, ... 
pc2r = fftw_plan_dft_c2r_1d(n, odata_intermediate, ... 
fftw_execute(pr2c);
derive(n, odata_intermediate, cf);
fftw_execute(pc2r);
fftw_destroy_plan(pr2c);
fftw_destroy_plan(pc2r);
...

- Find hotspots, estimate potential (e.g. Amdahls’ Law)
- Check CPU performance, optimize CPU execution
- Setup a validation process
- Estimate parallelism, complexity, ...
Analysis of the CPU Code

• Find hotspots, estimate potential (e.g. Amdahls’ Law)
  o Using profiling tools
  o gprof, oprofile, …
  o Code instrumentation (gettimeofday(), … )
  o …

• Check CPU performance
  o Is the machine enough loaded?
  o Optimize CPU execution, CPU code

• Setup a validation process
  o To validate that after each porting steps results are correct

• Estimate hot spots parallelism, complexity, …
What is a GPU-friendly Profile

• Spikes, bumpy profile
  o Few sections of code to focus on for a good speedup factor
  o The less functions to port, the less cost it involves

• Anyway, a GPU-friendly profile is
  o A profile for which the sections of code to focus on are data-parallel

• Don’t forget the Gustafson’s law
  o You may discover computational intensive kernels just by varying the amount of their input data
  o Sometimes the parallelism is placed at compute node level, with independent data distributed over the nodes
    • Then gather groups of data onto a same node and parallelize at hardware level
Initial Porting, Highlighting Parallelism

- Exhibit parallelism
- Push the code onto the GPU
- Validate execution

```c
#pragma hmpp <g> group, target=CUDA[/OpenCL]
#pragma hmpp <g> derive codelet, args[*] transfer=atcall

void derive(int nx, double _Complex ...) {
    int i;
    for (i=1; i<nx/2; ++i) {
        wrkq[i] = (0+I-1) * wrkq[i] * cf;
    }
    wrkq[0] = 0.0+I*0;
    wrkq[nx/2] = 0.0+I*0;
}
```

Build a GPU version of the function
Accelerate Codelet Function

- Declare and call a GPU-accelerated version of a function

```c
#pragma hmpp sgemm codelet, target=CUDA:OPENCL, args[*].transfer=atcall
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 < 1000 ; j++ ) {
        #pragma hmpp sgemm callsite
            sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    } /* . . . */
}
```

Declare CUDA and OPENCL codelets

Synchronous codelet call
Initial Porting, Highlighting Parallelism

- Select implementation for library calls and hotspots
- Insert calls to execute on GPU

Call GPU version of derive

```c
#pragma hmpp <g> derive callsite
derive(n, odata_intermediate, cf);
```

Call GPU version of library call

```c
#pragma hmppalt cufft call, name="fftw_plan_dft_r2c_1d"
pr2c = fftw_plan_dft_r2c_1d(n, idata_real, ...);
#pragma hmppalt cufft call, name="fftw_plan_dft_c2r_1d"
pc2r = fftw_plan_dft_c2r_1d(n, odata_intermediate, ...);
#pragma hmppalt cufft call, name="fftw_execute"
fftw_execute(pr2c);
```

Call GPU version of library call
First Porting Steps using HMPP 3.0

• First thing you want is to validate GPU results
  o If your algorithm produces wrong results
    • Maybe you have a numerical stability problem
    • Or your algorithm is not enough parallel
    • ...

• Insert the codelet directive before the definition of the function to offload
  o Use the ATCALL transfer policy
  o HMPP will automatically transfer
    • Scalars as INPUT
    • Arrays, pointers, … as INPUT and OUTPUT

• Insert the HMPPALT directive before calls to library functions

• Validate the result
  o To check that the GPU is a valid target for application
  o It may take time to execute the application
    • Due to all data transfers
    • And not optimized kernels
Transfer Optimizations

- Reduce CPU-GPU communication overhead
- Exploit reuse of data on the GPU

```c
int main(int argc, char **argv) {
    #pragma hmp sgemm acquire
    #pragma hmp sgemm allocate, data[vin1;vin2;vout], size={size,size}
    ...
    #pragma hmp sgemm advancedload, data[vin1;vin2;vout]

    for( j = 0 ; j < 1000 ; j++ ) {
        #pragma hmp sgemm
        sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    }
    ...
    #pragma hmp sgemm delegatedstore, data[vout]
    #pragma hmp sgemm free
    #pragma hmp sgemm release
}
```
Storage Policy

• Mirrored data or simply mirror
  o An area of memory on the host is mirrored on the accelerator
  o The HMPP runtime dynamically makes the link between the host address and the device address

• Simple data management
  o Few directives to manage mirrored data

• Easy to dynamically allocate and free a mirror
  o Use the ALLOCATE and FREE directives
Compute Asynchronously

• Perform CPU/GPU computations asynchronously

```c
int main(int argc, char **argv) {
    /* . . . */
    #pragma hmpp sgemm allocate, data[vin1;vin2;vout], size={size,size}
    /* . . . */
    for (j = 0; j < 1000; j++) {
        #pragma hmpp sgemm callsite, asynchronous
        sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
        /* . . . */
    }
    /* . . . */
    #pragma hmpp sgemm synchronize
    #pragma hmpp sgemm delegatedstore, data[vout]
    #pragma hmpp sgemm release
}
```
HMPP Directives Overview

• CODELET : Specialize a subroutine
• CALLSITE : Specialize a call statement
• SYNCHRONIZE : Wait for completion of the callsite
• ACQUIRE : Set a device for the execution
• ALLOCATE : Allocate memory
• FREE : Free allocated memory
• RELEASE : Release HWA
• ADVANCEDLOAD : Explicit data transfer CPU -> HWA
• DELEGATEDSTORE : Explicit data transfer HWA -> CPU
• GROUP : Groups codelets

» Directives in green are declarative
» Directives in Red are operational
What About Directives for Code Generation?

Directives

- HMPP
- RunEme
- HWA
- DirecEves
- Codelet
- HW-specific code generation
- HMPP application

HMPP Runtime

- Codelet
- HWA
- Data

January 2012

www.caps-entreprise.com
Improving Code Generation

- Directive-based GPU kernel code transformations

```c
#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] ;
            v1a = vin1[k][i] ;
            v2a = vin2[j][k] ;
        }
        prod += v1a * v2a;
        vout[j][i] = alpha * prod + beta * vout[j][i];
    }
}
```

Use pragma to preserve CPU code
Codelet Tuning Directives for High Level Optimization

- By adding properties
  - 1D or 2D gridification
- Applying code transformations
  - Loop tiling, unroll, jam, permute, fuse, …
- Using target specific directives
  - Micro architecture management (warp size…)
  - Memory management (CUDA shared memory, constant…)

High level application tweaking

January 2012
Scaling to Many-many cores

- Spread computations on available devices
- Manage data over several memory spaces

```c
float data[n][x][y];
#pragma hmp parallel, device="k%3"
for(k=0;k<n;k++) {
    #pragma hmp <MyGroup> f1 callsite
    myparallelfunc(&data[k],n);
}
```
#pragma hmpp <mygroup> group, target=CUDA

#pragma hmpp <mygroup> doit codelet, args[*].mirror, &
#pragma hmpp & args[*].transfer=manual
void doit(float A[1234]) {
  ...
}

float X[100][1234] ; // I have 100 arrays
#pragma hmpp <mygroup> acquire, device=0
#pragma hmpp <mygroup> acquire, device=1
...
for (k=0;k<100;k++) {
  float *ptr = X[i] ;
  #pragma hmpp <mygroup> allocate, data[ptr], size={1234}, &
  #pragma hmpp & device=“k%2"
}
#pragma hmpp parallel
for (k=0;k<100;k++) {
  #pragma hmpp <mygroup> advancedload, data[“X[k]”]
  #pragma hmpp <mygroup> doit callsite
  doit(X[k]) ;
  #pragma hmpp <mygroup> delegatedstore, data[“X[k]”]
}

Acquire two devices
Allocate data on a device then the other
Execute the codelets on the device that owns each mirror
Extern Functions

- Support for function calls inside codelets or regions
  - Functions called in codelets can be defined in other files
  - Avoid code duplication

```c
#include "sum.h"

int main(int argc, char **argv) {
  int i, N = 64;
  float A[N], B[N];
  ...
  #pragma hmpp cdlt region, args[B].io=inout, target=CUDA
  {
    #pragma hmppcg extern, sum
    for( int i = 0 ; i < N ; i++ )
      B[i] = sum( A[i], B[i] );
  }
  ...
}

#include "sum.h"

float sum( float x, float y )
{
  return x+y;
}
```

```c
#ifndef SUM_H
#define SUM_H

float sum( float x, float y );
#endif /* SUM_H */

#include "sum.h"

#include "sum.h"

#pragma hmpp function,target=CUDA
float sum( float x, float y )
{
  return x+y;
}
```
HMPP Runtime API

• Available bindings in C/C++ and Fortran
  o Low level OpenCL style programming with OpenCL/CUDA kernel generation
  o C++ API throws exceptions

• API call allows you to
  o Acquire a device
  o Allocate data
  o Transfer data
  o Launch codelets
  o Free data
  o Asynchronous operations
  o …

• Really useful for C++ programmers
Abstract the programming of manycore architectures
  o A rich set of programming and tuning directives
  o Distribute computations to exploit CPU and GPU cores in a node
  o Mix CPU and GPU libraries in same binary
  o Incrementally develop and port applications

An open source-to-source compiler
  o Work with standard compilers and hardware vendor tools
  o Ease maintenance by avoiding different languages
  o Preserve legacy code
HMPP Wizard
GPU library usage detection

Detected potential issue

HMPP-ALT-FFT/VERSION1
/exec/DZ2_Z2D.c @line 95 - Advice54: A call to the standard FFTW function "fftw_execute" has been detected inside a function.

Advice

Consider using an optimized library for your application with the HMPP ALT proxy.

Detected potential issue

sample/data/src/mycode.c @line 25 - Advice2: The computation density is low.

Loop Statistics

- Number of array access: 1
- Number of operations: 2 including 0 fops
- Number of intrinsic operations: 0 including 0 fops

Advice

- The computation may fetch few
HMPP Wizard

• HMPP wizard synthetizes metrics based on static and dynamic information
  o The result is shown as a HTML page

• Getting dynamic information from profilers
  o Gprof
  o Oprofile

• Getting static information from code analysis
  o Library calls
  o Code transformation inside codelets
Performance Measurements

• Parallelism is about performance!

• Track Amdahl’s Law Issues
  o Serial execution is a killer
  o Check scalability
  o Use performance tools

• Add performance measurement in the code
  o Detect bottleneck asap
  o Make it part of the validation process
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

January 2012
**Phylip, DNA distance**

- In association with the HMPP Center Of Excellence for APAC
- Computes a matrix 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

---

**Phylogenetic Tree of Life**

- **Bacteria**
  - Cyanobacteria
  - Proteobacteria
  - Proteobacteria
  - Aquilae
  - Spirochetae
- **Archaea**
  - Methanococci
  - Methanomicrobia
  - Thermoprotei
  - Thermopylids
  - Aquilae
- **Eucaryota**
  - Entamoebae
  - Flagellae
  - Amoebae
  - Oligochaetes
  - Ciliates
  - Planes
  - Plants
  - Fungi
  - Animals
  - Proteobacteria
  - Cyanoacteria
  - Archaea
  - Bacteria

---

January 2012  www.caps-entreprise.com
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

January 2012
### Performances (max)

**NO TRANSFERTS**

<table>
<thead>
<tr>
<th></th>
<th>T in s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalar</strong></td>
<td>18457.52</td>
</tr>
<tr>
<td>OMP=8</td>
<td>5040.1</td>
</tr>
<tr>
<td>HMPP</td>
<td>820.34</td>
</tr>
<tr>
<td>CUDA</td>
<td>1267.66</td>
</tr>
<tr>
<td><strong>ALL</strong></td>
<td></td>
</tr>
<tr>
<td><strong>NOISE</strong></td>
<td>3030.9</td>
</tr>
<tr>
<td><strong>DIFFUS</strong></td>
<td>4668.67</td>
</tr>
<tr>
<td><strong>KERSBS</strong></td>
<td>7531.1</td>
</tr>
<tr>
<td><strong>SHIFT</strong></td>
<td>1054.4</td>
</tr>
<tr>
<td><strong>BOUND</strong></td>
<td>15.68</td>
</tr>
<tr>
<td><strong>FLUX</strong></td>
<td>1933.82</td>
</tr>
<tr>
<td><strong>OMP / SEQ</strong></td>
<td>3.66</td>
</tr>
<tr>
<td>HMPP / SEQ</td>
<td>22.50</td>
</tr>
<tr>
<td>CUDA / SEQ</td>
<td>14.56</td>
</tr>
</tbody>
</table>

**Speedup**

\[
\text{Diffus} = \text{FFT FW + dfrac + FFTBW} \\
\text{KERSBS} = \text{KER + SBS}
\]

**Geom:** 128 x 128 x 256  
No I/O  
1 MPI