Code Migration Methodology for Heterogeneous Systems

Directives based approach using HMPP - OpenAcc

F. Bodin, CAPS CTO
Introduction

• Many-core computing power comes from parallelism
  o Multiple forms of parallelism cohabiting
    • Multiple devices (e.g. GPUs) with their own address space
    • Multiple threads inside a device
    • Vector/SIMD parallelism inside a thread
  o Massive parallelism
    • Tens of thousands of threads needed

• Keeping a unique version of the codes, preferably mono-language, is a necessity
  o Reduce maintenance cost
  o Preserve code assets
  o Fast moving hardware target context
  o Write codes that will last many architecture generations

• Multiple directive based approaches in a converging process
  o OpenAcc a first step
Overview of the Talk

1. A few words about many-cores

2. Methodology for legacy codes migration

3. Directive based approach for many-core programming
   1. OpenACC
   2. HMPP
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
Where Are We Going?

![Evolution of Processing Units in Future Processors](image)

- **CUDA/NVIDIA Tesla release**
- **Specialized Manycores (GPGPU)**
- **Frequency based Performance Improvement Era**
- **Manycores processors reaching the general purpose market**
  - Non migrated applications do not scale up

* Clock Frequency
  - GPU
  - CPU

** Number of Processing Units
  - "cores x threads"
  - Intel CPU, NVIDIA GPU

* Frequency based on Intel Processor (max.)

---

Heterogeneous HPC

www.caps-entreprise.com
Dealing with Legacy Codes

THE CRITICAL STEP

Define your Parallel Project

Port your Application on Many-core

Optimize Your Many-core Application

Hotspots

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

Parallelization

• Optimize CPU code
• Exhibit application SIMT parallelism
• Parallelize for Many-core
• Validate execution

Tuning

• Reduce data transfers
• Optimize kernel execution
• Provide feedback to application programmers for improving algorithm data structures/...
• Consider multiple devices

Hours to Days

Days to Weeks

Phase 1

Phase 2

A corporate project

• Purchasing Department
• Scientists
• IT Department

www.caps-entreprise.com
Go / No Go for GPU Target

Go
• Dense execution hotspots
• Fast kernels
• Low CPU-GPU data transfers
• Prepare to many-core parallelism

No Go
• Flat profile
• Slow GPU kernels (i.e. no speedup to be expected)
• Binary exact CPU-GPU results (cannot validate execution)
• Memory space needed
Phase 1 (details)
Phase 2 (details)
An Example

Monte Carlo simulation for thermal radiation

- Resource spent
  - 1.5 man-month
- Size
  - ~1kLoC of C d (DP)
- GPU C2050 improvement
  - x6 over 8 Nehalem cores
- Full hybrid version
  - x23 with 8 nodes compare to the 8 core machine
- Main porting operation
  - adding a few HMPP directives
Directives Based Approaches

• Supplement an existing serial languages with directives to express parallelism and data managements
  o Preserves code basis (e.g. C, Fortran) and serial semantic
  o Competitive with code hand-written in device dialect (e.g. CUDA)
  o Incremental approach to many-core programming

• Many variants
  o HMPP
  o PGI Accelerator
  o OpenAcc
  o OpenMP Accelerator extension
  o ...

• OpenACC is a new initiative by CAPS, CRAY, PGI and Nvidia
  o A first common subset
OpenACC Initiative

• Express data and computation to be executed on an accelerator
  o Using marked code regions

• Main OpenAcc contracts
  o Parallel and kernel regions
  o Parallel loops
  o Data regions
  o Runtime API

• Subset of HMPP supported features
  o OpenAcc constructs interoperable with other HMPP directives
  o OpenAcc support to be released in HMPP in March 2012

• Visit http://www.openacc-standard.com for more information
OpenAcc Data Management in HMPP

- Mirroring duplicate a CPU memory block into GPU memory
  - Mirror identifier is CPU memory block address
  - Only one mirror per CPU block
  - Users ensure consistency of copies via directives
OpenAcc Execution Model

- Host-controlled execution
- Based on three parallelism levels
  - Gangs – coarse grain
  - Workers – fine grain
  - Vectors – finest grain
Parallel Loops

• The loop directive can describe what type of parallelism to use to execute the loop and declare loop-private variables and arrays and reduction operations

• Clauses
  o gang [(scalar-integer-expression)]
  o worker [(scalar-integer-expression)]
  o vector [(scalar-integer-expression)]
  o collapse(n)
  o seq
  o independent
  o private(list)
  o reduction(operator:list)

```
#pragma acc loop gang(NB)
for (int i = 0; i < n; ++i){
  #pragma acc loop worker(NT)
  for (int j = 0; j < m; ++j){
    B[i][j] = i * j * A[i][j];
  }
}
```
Kernel Regions

• Parallel loops inside a region are transformed into accelerator kernels (e.g. CUDA kernels)
  o Each loop nest can have different values for gang and worker numbers

• Clauses
  o if(condition)
  o async[(scalar-integer-expression)]
  o copy(list)copyin(list)
  o copyout(list)create(list)
  o present(list)
  o present_or_copy(list)
  o present_or_copyin(list)
  o present_or_copyout(list)
  o present_or_create(list)
  o deviceptr(list)

```c
#pragma acc kernels
{
    #pragma acc loop independent
    for (int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            for (int k = 0; k < n; ++k){
                B[i][j*k%n] = A[i][j*k%n];
            }
        }
    }
}

#pragma acc loop gang(NB)
for (int i = 0; i < n; ++i){
    #pragma acc loop worker(NT)
    for (int j = 0; j < m; ++j){
        B[i][j] = i * j * A[i][j];
    }
}
```
Parallel Regions

- Start parallel activity on the accelerator device
  - Gangs of workers are created to execute the accelerator parallel region
  - Exploit parallel loops
  - SPMD style code but no barrier

- Clauses
  - \texttt{if(condition)}
  - \texttt{async[(scalar-integer-expression)]}
  - \texttt{num_gangs(scalar-integer-expression)}
  - \texttt{num_workers(scalar-integer-expression)}
  - \texttt{vector_length(scalar-integer-expression)}
  - \texttt{reduction(operator:list)}
  - \texttt{copy(list)}
  - \texttt{copyin(list)}
  - \texttt{copyout(list)}
  - \texttt{create(list)}
  - \texttt{present(list)}
  - \texttt{present_or_copy(list)}
  - \texttt{present_or_copyin(list)}
  - \texttt{present_or_copyout(list)}
  - \texttt{present_or_create(list)}
  - \texttt{deviceptr(list)}
  - \texttt{private(list)}
  - \texttt{firstprivate(list)}

```c
#pragma acc parallel num_gangs(BG),
num_workers(BW)
{
#pragma acc loop gang
for (int i = 0; i < n; ++i){
  #pragma acc loop worker
  for (int j = 0; j < n; ++j){
    B[i][j] = A[i][j];
  }
}
for(int k=0; k < n; k++){
  #pragma acc loop gang
  for (int i = 0; i < n; ++i){
    #pragma acc loop worker
    for (int j = 0; j < n; ++j){
      C[k][i][j] = B[k-1][i+1][j] + ...;
    }
  }
}
```
Data Management Directives

- Data regions define scalars, arrays and sub-arrays to be allocated in the device memory for the duration of the region
  - Explicit management of data transfer using clauses or directives
- Many clauses
  - `if(condition)`
  - `copy(list)`
  - `copyin(list)`
  - `copyout(list)`
  - `create(list)`
  - `present(list)`
  - `present_or_copy(list)`
  - `present_or_copyin(list)`
  - `present_or_copyout(list)`
  - `present_or_create(list)`
  - `deviceptr(list)`

```c
#pragma acc data copyin(A[1:N-2][1:M-2]),
copyout(B[N][M])
{
    #pragma acc kernels
    {
        #pragma acc loop independant
        for (int i = 0; i < N; ++i){
            A[i][0] = ...;
            A[i][M - 1] = 0.0f;
        }
        ...
    }
    #pragma acc update host(A)
    ...
    #pragma acc kernels
    for (int i = 0; i < n; ++i){
        B[i] = ...;
    }
}
```
Runtime API

- Set of functions for managing device allocation (C version)

  - int acc_get_num_devices( acc_device_t )
  - void acc_set_device_type ( acc_device_t )
  - acc_device_t acc_get_device_type ( void )
  - void acc_set_device_num( int, acc_device_t )
  - int acc_get_device_num( acc_device_t )
  - int acc_async_test( int )
  - int acc_async_test_all( )
  - void acc_async_wait( int )
  - void acc_async_wait_all( )
  - void acc_init ( acc_device_t )
  - void acc_shutdown ( acc_device_t )
  - void* acc_malloc ( size_t )
  - void acc_free ( void* )
  - ...

www.caps-entreprise.com
• Codelet and region based directives for many-cores
  o CUDA, OpenCL code generation, soon Intel MIC, x86
• HMPP support of OpenAcc syntax planned for March 2012
What is in HMPP that is not in OpenACC

- Multiple devices management
  - Data collection / map operation

- Library integration directives
  - Needed for a “single source many-core code” approach

- Loop transformations directives for kernel tuning
  - Tuning is very target machine dependent

- Open performance APIs
  - Tracing
  - Auto-tuning (H2 2012)

- And many more features
  - Native functions, buffer mode, codelets, …
Collections of Data in HMPP 3.0

```
struct {
    float * data;
    int size;
} Vec;
```
HMPP 3.0 Map Operation on Data Collections

```c
#pragma hmpp <mgrp> parallel
for(k=0;k<n;k++) {
    #pragma hmpp <mgrp> f1 callsite
    myparallelfunc(d[k],n);
}
```

- CPU 0
- CPU 1
- GPU 0
- GPU 1
- GPU 2

Main memory
- d0
- d1
- d2
- d3

Device memory
- d1
- d2
- d3
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-GPUs
  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
Example of Library Uses with HMPP 3.0

Replaces the call to a proxy that handles GPUs and allows to mix user GPU code with library ones

```c
CALL INIT(A,N)
CALL ZFFT1D(A,N,0,B)  ! This call is needed to initialize FFTE
CALL DUMP(A,N)

C TELL HMPP TO USE THE PROXY FUNCTION FROM THE FFTE PROXY GROUP
C CHECK FOR EXECUTION ERROR ON THE GPU

!$hmppalt ffte call , name="zfft1d", error="proxy_err"
CALL ZFFT1D(A,N,-1,B)
CALL DUMP(A,N)

C
C SAME HERE

!$hmppalt ffte call , name="zfft1d" , error="proxy_err"
CALL ZFFT1D(A,N,1,B)
CALL DUMP(A,N)
```
Library Interoperability in HMPP 3.0

... call libRoutine1(...) ...
... ...
#pragma hmppalt
call libRoutine2(…)
...
...
...
call libRoutine3(…)
...

HMPP Runtime API

... ...
...
...
...
...
...
...
...

Native GPU Runtime API

...
...
...
...
...
...
...
...
gpuLib(…)
...

GPU Lib

...
...
...
...
...
...
...
...
cpuLib1(…)
...
...
...
...
...
...
...
cpuLib3(…)
...

C/CUDA/…

...
...
...
...
...
...
...
...
cpuLib(…)
...

CPU Lib
Code Tuning Directive

- Directive-based GPU kernel code transformations
- Uses directives to preserve CPU code

```c
#pragma hmpp dgemm codelet, target=CUDA, args[C].io=inout

void dgemm( int n, double alpha, const double *A, const double *B,
            double beta, double *C ) {
    int i;

#pragma hmppcg(CUDA) grid blocksize "64x1 »
#pragma hmppcg(CUDA) permute j,i
#pragma hmppcg(CUDA) unroll(8), jam, split, noremainder
#pragma hmppcg parallel
    for( i = 0 ; i < n; i++ ) {
        int j;
#pragma hmppcg(CUDA) unroll(4), jam(i), noremainder
#pragma hmppcg parallel
            for( j = 0 ; j < n; j++ ) {
                int k; double prod = 0.0f;
                for( k = 0 ; k < n; k++ ) {
                    prod += VA(k,i) * VB(j,k);
                }
                VC(j,i) = alpha * prod + beta * VC(j,i);
            }
    }
}
```
Conclusion

• Software has to expose massive parallelism
  o The key to success is the algorithm!
  o The implementation has “just” to keep parallelism flexible and easy to exploit

• Directive-based approaches are currently one of the most promising track for many-cores
  o Preserve code assets
  o At node level help separating parallelism exposure from the implementation

• “Performance aware” directive programming most promising track for “portable performance”
  o Auto-tuning needs to be integrated into programming to define the search space
  o See the http://www.autotune-project.eu European project

• Mid-term convergence of directive dialects to be expected
  o OpenAcc is a first start
  o Convergence in OpenMP most likely