Innovative software for manycore paradigms

Incremental Migration of C and Fortran Applications to GPGPU using HMPP

HPC Advisory Council China Conference 2010
Introduction

• Many applications can benefit from GPU computing
  o Linear Algebra, signal processing
  o Bio informatics, molecular dynamics
  o Magnetic resonance imaging, tomography
  o Reverse time migration, electrostatic
  o …

• Porting legacy codes to GPU computing is a major challenge
  o Can be very expensive
  o Require to minimize porting risks
  o Should be based on future-proof approach
  o Implies application and performance programmers to cooperate

• A good methodology is paramount to reduce porting cost
  o HMPP provides an efficient solution
What is HMPP? (Hybrid Manycore Parallel Programming)

- A directive based multi-language programming environment
  - Help keeping software independent from hardware targets
  - Provide an incremental tool to exploit GPU in legacy applications
  - Avoid exit cost, can be future-proof solution

- HMPP provides
  - Code generators from C and Fortran to GPU (CUDA or OpenCL)
  - A compiler driver that handles all low level details of GPU compilers
  - A runtime to allocate and manage GPU resources

- Source to source compiler
  - CPU code does not require compiler change
  - Complement existing parallel APIs (OpenMP or MPI)
HMPP Main Design Considerations

• Focus on the main bottleneck
  o Communication between GPUs and CPUs

• **Allow incremental development**
  o Up to full access to the hardware features

• Work with other parallel APIs (e.g. OpenMP, MPI)
  o Orchestrate CPU and GPU computations

• Consider multiple languages
  o Avoid asking users to learn a new language

• Consider resource management
  o Generate robust software

• Exploit vendor tools/compilers
  o Do not replace, complement
How Does HMPP Differ from CUDA or OpenCL?

• HMPP parallel programming model is parallel loop centric
• CUDA and OpenCL parallel programming models are thread centric

```c
void saxpy(int n, float alpha,
           float *x, float *y) {
    #pragma hmppcg parallel
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

__global__
void saxpy_cuda(int n, float
                alpha,
                float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i<n) y[i] = alpha*x[i]+y[i];
}

int nblocks = (n + 255) / 256;
saxpy_cuda<<<nblocks, 256>>>(n, 2.0, x, y);
```
HMPP Codelets and Regions

- A codelet is a pure function that can be remotely executed on a GPU
- Regions are a short cut for writing codelets

```c
#pragma hmpp myfunc codelet, ...
void saxpy(int n, float alpha, float x[n], float y[n])
{
#pragma hmppcg parallel
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
```

```c
#pragma hmpp myreg region, ...
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
```
HMPP Codelets Arguments

• The arguments of codelet are also allocated in the GPU device memory
  o Must exist on both sides to allow backup execution
  o No hardware mechanism to ensure consistencies
  o Size must be known to perform the data transfers
• Are defined by the io clause (in Fortran use intent instead)
  o in (default) : read only in the codelet
  o out: completely defined, no read before a write
  o inout: read and written
• Using inappropriate inout generates extra PCI bus traffic

```c
#pragma hmpp myLabel codelet, args[B].io=out, args[C].io=inout
void myFunc( int n, int A[n], int B[n], int C[n]){
    for( int i=0 ; i<n ; ++i){
        C[i] = C[i] * A[i];
    }
}
```
Codelet Target Clause

- Target clause specifies what GPU code to generate
  - *GPU* can be CUDA or OpenCL
- Choice of the implementation at runtime can be different!
  - The runtime select among the available hardware and code

```c
#pragma hmpp myLabel codelet, target=[GPU], args[C].io=out
void myFunc( int n, int A[n], int B[n], int C[n]){
    ...
}
```

- `#pragma hmpp myLabel codelet, target=CUDA`
  - NVIDIA only GPU
- `#pragma hmpp myLabel codelet, target=OpenCL`
  - NVIDIA & AMD GPU, AMD CPU
Running a Codelet or Section on a GPU - 1

• The callsite directive specifies the use of a codelet at a given point in your application.

• callsite directive performs a Remote Procedure Call onto the GPU

```c
#include <hmpp/indications.h>

#pragma hmpp call1 codelet, target=CUDA
#pragma hmpp call2 codelet, target=OpenCL

void myFunc(int n, int A[n], int B[n]){
    int i;
    for (i=0 ; i<n ; ++i)
        B[i] = A[i] + 1;
}

void main(void)
{
    int X[10000], Y[10000], Z[10000];
    ...
    #pragma hmpp call1 callsite, ...
    myFunc(10000, X, Y);
    ...
    #pragma hmpp call2 callsite, ...
    myFunc(1000, Y, Z);
    ...
}
```
• By default, a CALLSITE directive implements the whole Remote Procedure Call (RPC) sequence

• An RPC sequence consists in 5 steps:
  o (1) Allocate the GPU and the memory
  o (2) Transfer the input data: CPU => GPU
  o (3) Compute
  o (4) Transfer the output data: GPU => CPU
  o (5) Release the GPU and the memory
Tuning Hybrid Codes

• Tuning hybrid code consists in
  o Reducing penalty when allocating and releasing GPUs
  o Reducing data transfer time
  o Optimizing performance of the GPU kernels
  o Using CPU cores in parallel with the GPU

• HMPP provides a set of directives to address these optimizations

• The objective is to get efficient CPU and GPU computations
Reducing Data Transfers between CPUs and GPUs

• Hybrid code performance is very sensitive to the amount of CPU-GPU data transfers
  o PCIx bus is a serious bottleneck (< 10 GBs vs 150 GBs)

• Various techniques
  o Reduce data transfer occurrences
  o Share data on the GPU between codelets
  o Map codelet arguments to the same GPU space
  o Perform partial data transfers

• Warning: dealing with two address spaces may introduce inconsistencies
Reducing Data Transfers Occurrences

- Preload data before codelet call
  - Load data as soon as possible

```c
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
```
Sharing Data Between Codelets with Resident Data

- Share data between codelets of the same group
  - Keep data on the HWA between two codelet calls
  - Avoid useless data transfers

```c
#pragma hmpp <process> group, target=CUDA
#pragma hmpp <process> resident
float initValue = 1.5f, offset[9];
...
#pragma hmpp <process> reset1 codelet, args[t].io=out
void reset(float t[M][N]){
    int i,j;
    for (i = 0; i < M; i += 1) {
        for (j = 0; j < N; j += 1) {
            t[i][j] = initValue + offset[(i+j)%9];
        }
    }
}
#pragma hmpp <process> process codelet, args[a].io=inout
void process(real a[M][N], real b[M][N]){
    int i,j;
    for (i = 0; i < M; i += 1) {
        for (j = 0; j < N; j += 1) {
            a[i][j] = cos(a[i][j]) + cos(b[i][j]) - initValue;
        }
    }
}
```
Tuning GPU Kernels

- GPU kernel tuning set-up parallel loop suitable for GPU architectures

- Multiple issues to address
  - Memory accesses
  - Thread grid tuning
  - Register usage tuning
  - Shared memory usage
  - Removing control flow divergence

- In many cases, CPU code structure conflicts with GPU efficient code structure
Methodology to Port Applications

• Prerequisite
  o Understand your performance goal
    • Memory bandwidth needs are a good potential performance indicator
  o Know your hotspots
    • Beware of Amdahl’s law
  o Ensure you know how to validate the output of your application
    • Rounding may differ on GPUs
  o Determine if your goal can be achieved
    • How many CPUs and GPUs are necessary?
    • Is there similar existing codes for GPUs (in CUDA, OpenCL or HMPP)?

• Define an incremental approach
  o Ensure to check the results at each step

• Two phase approach
  o Phase 1: Application programmers validate the computed results
  o Phase 2: Performance programmers focus on GPU code tuning and data transfer reduction
Phase 1

• Phase 1 – Dealing with application issues
  o Exhibit application SIMT parallelism
  o Push application hotspot on GPU
  o Validate execution
  o Optimize CPU code
  o Runaway from flat profile

• Phase 1 - Tools
  o CPU code profiler
  o CPU code optimizer: Vtune, Acumen, …
  o HMPP, …
Phase 2

• Phase 2 – Performance
  o Optimize application GPU execution
  o Provide feedback to application programmers for improving application algorithm/data structures/…
  o Exploit CPU and GPU
  o Reduce CPU-GPU data transfers

• Phase 2 - Tools
  o GPU profiling: Vampir, Paraver, Nsight, TAU
  o Debugger: Allinea DDT, …
  o HMPP, …
Methodology Overview

BEGIN

Pick new hotspots

Hotspots compute intensive enough ?

yes -> Hotspots parallel ?

no -> Reconsider algorithms

yes -> Construct the codelets

no -> Compile, Run, and Check results

Hotspots parallel ?

yes -> Rewrite

no -> Compile, Run, and Check results

Code appropriate to GPU ?

yes -> Compile, Run, and Check results

no -> HMPP/ Vampir Analysis

Profile

Compile and run

Check results

Peak Performance achieved

Allocation dominating

Use allocate/ release directives

Communication dominating

Optimize data transfers

Compute dominating

Optimize codelet code

HMPP/ Vampir Analysis

HMPP Performance Analyzer

HMPP Wizard & Feedback

Code appropriate to GPU ?

yes -> Compile, Run, and Check results

no -> Construct the codelets

yes -> Reward

no -> Reconsider algorithms

PHASE 1

PHASE 2
CAPS HMPP Wizard (Beta)

- Provide advice to programmers to help them to write GPU friendly code
HMPP Performance Analyzer (Beta)
Examples of Ported Applications – 1

• Smoothed particles hydrodynamics
  o Effort: 2 man-month
  o Size: 22kLoC of F90 (SP or DP)
  o GPU C1060 improvement: x 2 over 1 Nehalem core (x1.1 DP)
  o Main difficulty: kernels limited to 70% of the execution time

• 3D Poisson equation, conjugate gradient
  o Effort: 2 man-month
  o Size: 2kLoC of F90 (DP)
  o CPU improvement: x 2
  o GPU C1060 improvement: x 5 over 1 Nehalem core
  o Main porting operation: highly optimizing kernels
  o Main difficulty: none

The ratio performance over resource is the important information here.
Examples of Ported Applications - 2

- **Electron propagation - solver**
  - Effort: 2 man-month
  - Size: 10 kLoC of F95 (DP, MPI)
  - CPU improvement: x 1.3
  - GPU C1060 improvement: x 1.15 over 4 Nehalem cores
  - Main porting operation: solver algorithm modifications
  - Main difficulty: small matrices, many data transfers

- **3D combustion code**
  - Effort: 2 man-month
  - Size: ~1MLoC of F90 (DP)
  - GPU C1060 improvement: ~x1 (data transfer limited) over 1 Nehalem core; C2050 x1.3
  - Main difficulty: execution profile shows few hot-spots (70%)
  - Next: hybrid parallelization
Examples of Ported Applications - 3

• **Euler equations**
  - Effort: <1 man-month
  - Size: ~40kLoC of F90 (DP)
  - CPU improvement: x 3 over the original code
  - GPU C1060 improvement: x 3 over 1 Nehalem core
  - Main porting operation: specializing the code for the main execution configuration
  - Main difficulty: reorganizing computational kernels

• **Tsunami/flood simulation**
  - Effort: 0.5 man-month
  - Size: ~4kLoC (DP, MPI)
  - GPU C1060 improvement: x 1.28 over 1 Nehalem core
  - Next: highlight more parallelism, reducing data transfers (High performance potential)
Examples of Ported Applications - 4

- **Molecular fluid dynamics**
  - Effort: 1 man-month
  - Size: ~1kLoC of C99 (DP)
  - GPU C2050 improvement: ~x 2 over 4 Nehalem cores
  - Main porting operation: pointer based data structure rewriting
  - Main difficulty: Very efficient original OpenMP code

- **Monte Carlo simulation for thermal radiation**
  - Effort: 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 over 8 Nehalem cores
  - Main porting operation: adding a few HMPP directives
  - Main difficulty: none
Examples of Ported Applications - 5

• **Weather models (GTC 2010 talk, M. Govett, NOAA)**
  - Effort: 1 man-month (part of the code already ported)
  - GPU C1060 improvement: 10 over a Nehalem CPU
  - Main porting operation: reduction of CPU-GPU transfers
  - Main difficulty: GPU memory size is the limiting factor
Conclusion

• Heterogeneous architectures are becoming ubiquitous
  o In HPC centers but not only
  o Tremendous opportunities but not always easy to seize
  o CPU and GPU have to be used simultaneously

• Legacy codes still need to be ported
  o An efficient methodology is required
  o A methodology supporting tools is needed and must provide a set of consistent views
  o The legacy style is not helping
  o Highlighted parallelism for GPU is useful for future manycores

• HMPP based programming
  o Helps implementing incremental strategies
  o Is being complemented by a set of tools
  o Engage in an Open Standard path with Pathscale
Innovative Software for Manycore Paradigms
S/D/CGEMM Performance Example (NVIDIA GTX 275)

- **sgemm**
  - Bar charts showing gigaflops x matrix size for different matrix sizes (1024, 2048, 4096)
  - Comparison between HMPP and cuBLAS

- **dgemm**
  - Bar charts showing gigaflops x matrix size for different matrix sizes (512, 1024, 2048, 4096)
  - Comparison between HMPP and cuBLAS

- **cgemm**
  - Bar charts showing gigaflops x matrix size for different matrix sizes (512, 1024, 2048, 4096)
  - Comparison between HMPP and cuBLAS