Introduction to HMPP
Hybrid Manycore Parallel Programming

Московский государственный университет, Лоран Морен, 30 августа 2011
The application should stay hardware resilient
- New architectures / languages to master
- Hybrid solutions evolve: we don’t want to redo the work each time

Kernel optimizations are the key to get performance
- An hybrid application must get the best of its hardware resources
Methodology to Port Applications
Methodology to Port Applications

- Define your parallel project
- Port your application on GPU
- Optimize your GPGPU application

Phase 1
- Hotspots
  - Performance goal
  - Validation process
  - Hotspots selection
  - Continuous integration
  - Hours to Days

Phase 2
- Parallelization
  - Optimize CPU code
  - Exhibit Parallelism
  - Move kernels to GPU
  - Days to Weeks

Tuning
- Exploit CPU and GPU
- Optimize kernel execution
- Reduce CPU-GPU data transfers
- Weeks to Months

GPGPU operational application with known potential
Take a decision

• **Go**
  - Dense hotspot
  - Fast GPU kernels
  - Low CPU-GPU data transfers
  - Midterm perspective:
    Prepare to manycore parallelism

• **No Go**
  - Flat profile
  - GPU results not accurate enough
    - cannot validate the computation
  - Slow GPU kernels
    - i.e. no speedup to be expected
  - Complex data structures
  - Big memory space requirement
Tools in the Methodology

- **Parallel Profiling Tools**
  - Paraver, TAU, Vampir
- **Profiling tools**
  - Gprof, Kache grind, Vampir, Acumem, ThreadSpotter
- **Debugging tools**
  - gdb, alinea DDT, TotalView

**Define your parallel project**
- Hours to Days

**Port your application on GPU**
- Days to Weeks

- **Parallel Profiling Tools**
  - Paraver, TAU, Vampir
- **HMPP Performance Analyzer**
- **Target Specific tools**
  - Nvidia Insight

**Optimize your GPGPU application**
- Weeks to Months

- **HMPP Workbench**
- **HMPP Wizard**

**Phase 1**
GPGPU operational application with known potential

**Phase 2**
- Hotspots
- Parallelization
- Tuning
Move Computation to GPU:
HMPP Workbench
HMPP Workbench

- HMPP: glue between applications and hardware accelerators
  - Abstract the architecture
  - Manage data transfers
  - Ensure application interoperability & resource management
What is HMPP? (Hybrid Manycore Parallel Programming)

- A directive based programming model
  - Provide an incremental tool to exploit GPU in legacy applications
  - Complement existing parallel APIs (OpenMP or MPI)
  - Avoid exit cost, future-proof solution
  - Based on an OpenStandard

- An integrated compilation chain
  - Code generators from C and Fortran to GPU (CUDA or OpenCL)
  - Source to source compiler: uses hardware vendor SDK
  - One single compiler driver interfacing GPU compilers
  - Complements existing parallel APIs (OpenMP or MPI)

- A runtime support of heterogeneous architectures
  - Manages hardware resources and its availability
  - Interface transfers and synchronizations of the hardware
  - Fast and easier application deployment on new hardware
  - Free of use
HMPP usage in three steps

1. A set of directives to program hardware accelerators
   • Drive your HWAs, manage transfers
   • Optimize kernels at source level

2. A complete tool-chain to build manycore applications
   • Build your hybrid application

3. A runtime to adapt to platform configuration
1 – HMPP Directives: drive Hybrid Applications

```c
int main(int argc, char **argv) {
    // ...
    #pragma hmpp compute advancedload, args[M;N]
    for (j = 0; j < 2; j++) {
        #pragma hmpp compute callsite
        compute(size, size, size, alpha, vin1, vin2, beta, vout);
    }
    // ...
}
```

```c
#pragma hmpp compute codelet, target=CUDA, args[a].io=inout
void compute ( int M, int N, float alpha, float a[n][n], float b[n][n]) {
    #pragma hmppcg hmppcg grid blocksize 16 X 16
    #pragma hmppcg unroll 2, jam
    for (i = 0; i < M; i += 1) {
        #pragma hmpp unroll 4, guarded
        for (j = 0; j < N; j += 1) {
            a[i][j] = cos(a[i][j]) + cos(b[i][j]) - alpha;
        }
    }
}
```
2 – HMPP Compilation Flow

- HMPP drives all compilation passes
  - Host application compilation
    - HMPP Runtime is linked to the host part of the application
  - Codelet production
    - Target code is produced
    - A dynamic library is built
3 – HMPP Runtime

• Ensure allocation of GPUs
  o Allocate GPU on a first-come-first-serve basis
  o Forbid sharing of GPUs to avoid performance degradation

• Load the dynamic libraries containing the GPU codes

• Insulate host code from lack of GPU
  o HMPP applications still run when the GPU driver or libraries are not available on the host machine

• Provide some information on the execution of the application
The functions are **insulated** and **offloaded**:

- Launch the execution of a single Remote Procedure Call
  - With a single code and its data to compute
- Or parallelize tasks on multiple HWAs
  - All tasks completely independents
- Or thread a set of functions on a single HWA
  - Executed in sequence and sharing data on the HWA
One directive to setup a GPU codelet...

```c
#pragma hmpp sgemm 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 sgemm callsite
        sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    }
    // ...
}
```

... and another inserted in the application
Starting with HMPP
Basic Concepts

• Pure function calls are offloaded on the accelerator
  o These functions must fit some constraints
    • No I/O on data
    • No access to global/volatile variables
    • Fixed number of arguments

• Controlled by directives
  o Pragmas in C
  o Special comments in Fortran

• A unique label associates a set of directives
General Syntax of the Directives (C)

- A single line HMPP directive is:

  \[\texttt{#pragma hmpp myLabel command [, attribute]}\]

- Long directives can be continued on multiple lines

  \[
  \texttt{#pragma hmpp myLabel command ... \&}
  
  \texttt{#pragma hmpp \& ... \&}
  
  \texttt{#pragma hmpp \& ...}
  
  ...\]
General Syntax of the Directives (Fortran)

- Any form of comment (F77, F90,...) starting with $hmpp

  ```fortran
  !$hmpp myLabel command [, attribute]
  ```

- Long directives can be continued on multiple lines

  ```fortran
  !$hmpp myLabel command ... &
  !$hmpp & ... &
  !$hmpp & ... &
  ...
  ```
Programming Hardware Accelerators with HMPP
Programming my Hardware Accelerator

• I want to launch the execution of a remote function
How can I do that?

• Think codelet
  o The candidate function has to be **insulated** and **offloaded** on the remote hardware accelerator → this is the **codelet**
  o The HMPP directives provide the user with codelet control

• There are two essential HMPP clauses to know:
  o **CODELET**
    • Identify the native function to offload on a specific target
    • Order HMPP Codelet Generators to produce target code
  o **CALLSITE**
    • Explicit a call to this specialized function in the application
CODELET/CALLSITE

• The CODELET clause is placed just before the function

```c
#pragma hmpp myLabel codelet ...
void myFunc(...){
    ...
}
```

• The CALLSITE clause is placed just before the call statement

```c
...
#pragma hmpp myLabel callsite
myFunc(...);
...
```

» Each directive exists only once for a pair
CODELET/CALLSITE

• Simple C example:

```c
#pragma hmpp call1 codelet, target=CUDA
#pragma hmpp call2 codelet, target=CAL
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);
}
```
Simple Fortran example:

```fortran
SUBROUTINE myFunc(n,A,B)
    INTEGER, INTENT(IN) :: n, A(n)
    INTEGER, INTENT(OUT) :: B(n)
    INTEGER :: i
    DO i=1,n
        B(i) = A(i) + 1
    ENDDO
END SUBROUTINE

PROGRAM test
    INTEGER :: X(10000), Y(10000), Z(10000)
    ...!
    CALL myFunc(10000,X,Y)
    ...
    CALL myFunc(10000,Y,Z)
    ...
END PROGRAM
```
CALLSITE: Full Remote Procedure Call sequence

- By default, a CALLSITE directive implements the whole RPC sequence
  - RPC = Remote Procedure Call

- An RPC sequence consists in 5 steps:
  - (1) Allocate the HWA and the memory
  - (2) Transfer the input data: CPU => HWA
  - (3) Compute
  - (4) Transfer the output data: HWA => CPU
  - (5) Release the HWA and the memory
Full RPC sequence

- At this time, RPC steps are considered as synchronous
  - Each step is blocking
  - Application wait for the step completion
Ground Rules

• An HMPP program contains at least a pair of CODELET/CALLSITE directives
  ○ A CODELET is a specialization of a subroutine
  ○ A CALLSITE is the specialization of a call statement

• To each CODELET corresponds a single CALLSITE and vice versa

• A pair CODELET/CALLSITE is identified by a unique label
How do I choose the Accelerator?

- HMPP can address several HWAs:
  - C
  - CUDA
  - OpenCL

- But how can I specialize my codelet?
  - Use the TARGET attribute
Codelet Directive: the TARGET Attribute

- The target attribute tells HMPP for which HWA the specialized version must be generated:

```
#pragma hmpp myLabel codelet, target=CUDA
```
CODELET Directive: multiple TARGETS

• Specify for which HWA a specialized version must be generated
  o No target mean no specialized code to generate

  #pragma hmpp myLabel codelet, target=CUDA

• Multiple targets are possible (selected in order)
  o The first allocated without any errors is selected

  #pragma hmpp myLabel codelet, target=CUDA:CAL
HMPP Fallback

• **What HMPP does if a codelet file is not present at run time?**
  - If the hardware accelerator is unavailable?
  - If data allocation fails?
  - If the data transfer fails?
  - If the computation raises an error?

• **The native CPU version is used as a fallback**
  - This behavior can be inhibited with HMPP_NO_FALLBACK

• **Fallback is only possible until step 3 (computation) in synchronous mode**
  - In asynchronous mode, a computation failure leads to the end of the application
The IO attribute allows to specify if an argument is
- **IN** for an input (data needed on the HWA)
- **OUT** for an output (result to be sent back from the HWA)
- **INOUT** for both

In the RPC sequence
- **IN** and **INOUT** arguments are transferred in step (2)
- **OUT** and **INOUT** arguments are transferred in step (4)

Attached to the CODELET directive and used by CALLSITE to realize the right steps in the RPC sequence.
CODELET Directive: the IO Attribute (2)

• In C:
  o The default IO of all arguments is 'in'
  o The IO status of all outputs must be explicitly specified

```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];
    }
}
```

• In Fortran, the default IO status
  o matches the Fortran INTENT when is it is specified
  o otherwise set to IN exactly like in C

• Beware of the default behavior of Fortran which is INTENT(INOUT) and not INTENT(IN)

• The IO attribute has little use in Fortran
  o It is recommended to use INTENT as much as possible
HMPP Toolchain
HMPP Compilation Passes

- How do I split the building process of my application?
  - Use HMPP toolchain to generate separately
    - Host application
    - Codelets

- The compilation process can be split into:
  - Host application compilation
  - HMPP Codelet compilation (generate + compile)
  - HMPP Codelet compilation (generate only)
  - HMPP Codelet compilation (compile only)
  - HMPP Codelet template generation
HMPP Complete Application Compilation

- HMPP drives all compilation passes
  - Host application compilation
    - HMPP Runtime is linked to the host part of the application
  - Codelet production
    - Target code is produced
    - A dynamic library is built

$ hmpp gcc myProgram.c
Programming Hardware Accelerators with HMPP: Managing Data
HMPP Main Directives

- **CODELET**: Specialize a subroutine
- **CALLSITE**: Specialize a call statement

Reminder

RPC SEQUENCE

1. Allocate GPU
2. Transfer IN data
3. GPU Compute
4. Transfer OUT data
5. Release GPU
Splitting the RPC sequence

- To optimize HWA usage according to your application algorithm, this RPC sequence can be split into several parts.
HMPP Directives Overview

- CODELET: Specialize a subroutine
- CALLSITE: Specialize a call statement
- SYNCHRONIZE: Wait for completion of the callsite
- ALLOCATE: Reserve HWA and allocate memory
- RELEASE: Release HWA and its memory
- ADVANCEDLOAD: Explicit data transfer CPU -> HWA
- DELEGATEDSTORE: Explicit data transfer HWA -> CPU
- GROUP: Define a group of codelets
- RESIDENT: Declare a resident (global) variable
- MAP: Map arguments together

» Directives in green are declarative
» Directives in blue are operational
ALLOCATE directive

- Step #1 in RPC sequence
- Lock the HWA and allocate memory for all codelet’s arguments
- Syntax:

```
#pragma hmpp myLabel allocate [ , ... ]
!$hmpp myLabel allocate [ , ... ]
```

- If not present, executed implicitly before
  - The CALLSITE directive
  - All ADVANCEDLOAD directives
ALLOCATE directive: the SIZE computation

• Allocation of array arguments requires to evaluate their dimensions
• Their value is obtained by combining the declaration in the CODELET with the argument values in the CALLSITE
• Example:

```c
#pragma hmpp myFunc1 codelet, ...
void myFunc( int n, int A[n*2]) {   // n*2
    ...
}

#pragma hmpp myFunc1 allocate         // A[(sz+1)*2]]
...
#pragma hmpp myFunc1 callsite
myFunc(sz+1,X) ;                     // n==sz+1
```

» The expression (sz+1)*2 is evaluated by the ALLOCATE
ALLOCATE directive: the SIZE attribute (1)

- The size expression may not be valid/available in the allocate context
  - Use an attribute size to specify an alternative expression

- Syntax:
  ```
  #pragma hmpp Label allocate, args[...].size={ expr , ... }
  !$hmpp Label allocate, args[...].size={ expr , ... }
  ```

- Specify one expression per dimension
  - The value evaluated by the SIZE attribute is supposed to be equal to the dimension at the callsite
ALLOCATION directive: the SIZE attribute (2)

• A full example:

```c
#pragma hmpp myFunc1 codelet, ...
void myFunc( int n, int A[n*2]) {
    ...
}
...
sz = 0 ;
#pragma hmpp myFunc1 allocate, args[A].size={2*(NB+5)}
...
sz = NB+4 ;
...
#pragma hmpp myFunc1 callsite
myFunc(sz+1,X) ;
```

• Warning: the size expression is inserted in the instrumented code “as is”
  o Expression is not preprocessed (no macros)
RELEASE directive (1)

- Release the HWA and free the used memory
- Syntax:

```
#pragma hmpp myLabel release

!$hmpp myLabel release
```

- Should be the last directive executed for a given label

  » Is required in case of explicit DELEGATEDSTORE
HMPP RPC iteration

- Repeat a callsite multiple times

```c
for (i=0; i<10; i++) {
    #pragma hmpp myFunc callsite, ...
    myFunc(n, X) ;
}
```

```c
#pragma hmpp myFunc allocate
for (i=0; i<10; i++) {
    #pragma hmpp myFunc callsite, ...
    myFunc(n, X) ;
}
#pragma hmpp myFunc release
```

The inefficient way:
Implicit ALLOCATE and RELEASE in each iteration

The efficient way:
Single ALLOCATE and RELEASE

» Repeating CALLSITES between ALLOCATE and RELEASE is called an RPC iteration
ADVANCEDLOAD directive (1)

• Transfer arguments from CPU to HWA

• Syntax:

```c
#pragma hmpp myLabel advancedload, args[...]
!$hmpp myLabel advancedload, args[...]
```

• The standalone attribute ARGS specifies the arguments to be transferred

• It can be performed safely several times: only the first one is effective

• Checked and done if necessary by the callsite
ADVANCEDLOAD directive (2)

- Example:

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

#pragma hmpp myFunc advancedload, args[n;A;B]
...
#pragma hmpp myFunc callsite
myFunc(100,X,Y,Z) ;
...
```
ADVANCEDLOAD: the ASYNCHRONOUS attribute (1)

• By default, the ADVANCEDLOAD directive is blocking until all specified arguments are transferred

• The ASYNCHRONOUS attribute makes the transfer non-blocking
  o The transferred variable should remain alive (and unchanged) until the CALLSITE

• Some targets may not implement asynchronous transfers
  o The transfer could be implemented in a blocking way
ADVANCEDLOAD: the ASYNCHRONOUS attribute (2)

Example:

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

#pragma hmpp myFunc advancedload, args[n;A;B], asynchronous
...
// The transfer of n, A and B is currently being performed
...
#pragma hmpp myFunc callsite
myFunc(100,X,Y,Z) ;
...```
The CONST argument attribute (1)

• Prevent reloading the same data (for the next RPC it.)
  o Can be specified on CALLSITE, ADVANCEDLOAD and RESIDENT

• Specifying this attribute for an argument indicates that its value remains constant between the ALLOCATE and the RELEASE

• CONST is a global setting
  o There is no difference between putting it in GROUP, ADVANCEDLOAD or CALLSITE directives
  o For readability reasons, it is recommended to write the CONST property in the GROUP directive

• In practice, the transfer is only performed once by the first ADVANCEDLOAD or CALLSITE directive
  o Do not forget the RELEASE
The CONST argument attribute (2)

- Example:

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

#pragma hmpp allocate
for (iter=1; iter<cpt; iter++) {
    #pragma hmpp myFunc callsite, args[A].const=true
    myFunc(100,X,Y,Z);
    ...
}
#pragma hmpp release
```

- Do not forget the RELEASE
  - Otherwise the variable ‘A’ remains on the HWA only until the end of each CALLSITE (because of implicit RELEASE)
The DELEGATEDSTORE directive

- Transfer arguments from HWA to HOST
  - RPC sequence step #4
- Syntax:
  
  ```
  #pragma hmpp label delegatedstore, args[...]
  ```

- The standalone attribute “args[...]” specifies the arguments to be transferred
- Arguments with a DELEGATEDSTORE are not implicitly transferred in the CALLSITE
Asynchronous CALLSITEs

- The ASYNCHRONOUS attribute makes the CALLSITE non-blocking.
- A SYNCHRONIZE directive is mandatory to wait for the CALLSITE completion.
- All DELEGATEDSTORE on all outputs must be explicit and executed after the SYNCHRONIZE directive.
- A RELEASE directive is mandatory.
- Syntax:
  ```
  #pragma hmpp label callsite, asynchronous, ...
  ...
  #pragma hmpp label synchronize
  ...
  #pragma hmpp label delegatedstore, ...
  ...
  #pragma hmpp label release
  ```
Quick Summary so far

- For a single CODELET/CALLSITE pair the RPC sequence is composed of:
  - (a) an optional ALLOCATE directive
  - (b) some optional ADVANCEDLOAD directives
  - (c) a mandatory CALLSITE directive
    + a mandatory SYNCHRONIZE directive if ASYNCHRONOUS
  - (d) some optional DELEGATEDSTORE directives
    (mandatory if ASYNCHRONOUS)
  - (e) an optional RELEASE directive
    (mandatory if ASYNCHRONOUS)

  » Step (b), (c) and (d) can be repeated and mixed (with great care)
Codelet Code Generation in HMPP
A Hybrid/Heterogeneous Compute Node

- **General purpose cores**
  - Share a main memory
  - Core ISA provides fast SIMD instructions
  - Large cache memories

- **Streaming engines (e.g. GPU)**
  - Application specific architectures ("narrow band")
  - Vector/SIMD
  - Can be extremely fast
Stream computing

- Stream programming is well suited to GPU
  - But memory hierarchy is exposed
- A similar computation is performed on a collection of data (stream)
  - There is no data dependence between the computation on different stream elements
Parallel Loops in Codelets and Regions

- Parallel loops are the code constructs converted in GPU threads
  - Using a process call loop nest gridification
  - Directive `hmppcg parallel` forces parallelisation
- Two levels of parallelism can be used to generate the threads

```c
#pragma hmppcg parallel
for (i = si; i < ei; ++i){
    #pragma hmppcg parallel
    for (j = sj; j < ej; ++j) {
    }
}
```
Loop Nest Gridification

- The loop nest gridification process converts parallel loop nests in a grid of GPU threads
  - Use the parallel loop nest iteration space to produce the threads

```cpp
#pragma hmppcg parallel
for(int i = 0; i<10; ++i){
  y[i] = alpha*x[i] + y[i];
}
```

```
{ int i = blockIdx.x* blockDim.x + threadIdx.x;
  if( i<10 )
    y[i]=alpha*x[i]+y[i];
}
```
Memory Coalescing? Accesces in the grid

- $A(i,k)$ is well coalesced ($i$ on the 1\textsuperscript{st} array dimension)
- $C(j,i)$ is not well coalesced ($i$ on the 2\textsuperscript{nd} array dimension)
  - But does not really matter here (not deep inside the loop nest)
- And for $B(k,j)$?

```fortran
DO j=1,m    ! 2\textsuperscript{nd} grid dimension
  DO i=1,n  ! 1\textsuperscript{st} grid dimension (the warps)
    tmp = 0
    DO k=1,n
      tmp += A(i,k)*B(k,j)
    ENDDO
    C(j,i) = tmp
  ENDDO
ENDDO
```
Memory Accesses and Warps

• $B(k,j)$ is well coalesced
  o The array indices do not depend on $i$ so all threads in the warp are accessing the same element
  o A broadcast (since CUDA 1.2) of a single scalar to all members of the warp
    • That’s good!
How Does HMPP Differ from CUDA or OpenCL?

- HMPP parallel programming model is parallel loop centric

```cpp
#pragma hmpp saxpy codelet, target=CUDA, args[y].io=inout
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];
}
```

- CUDA and OpenCL parallel programming models are thread centric

```cpp
__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];
}
```

```cpp
int nblocks = (n + 255) / 256;
saxpy_cuda<<<nblocks,256>>>(n, 2.0, x, y);
```
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
The HMPP Code Generator proposes a dedicated performance directive set.

The general syntax of a code generation directive is:

```
#pragma hmppcg [(target)]? directive_type [clause] [, clause]*
```

Example of available features:

- Specification of loop properties
  - Parallel/no parallel, reductions
- Loop transformations
  - Permute, distribute, fuse, unroll, …
- Specification of grid properties
  - Grid size, global/private variables, …
- Grid block ‘s shared-memory programming
GPU Kernels Tuning Strategies

• Improve memory coalescing
  o Choose loop order (permutation directive)

• Grid size tuning
  o Choose a grid block size (grid blocksize directive)

• Exploit GPU memory hierarchy
  o Exploit data reuse (temporal data locality) in shared/local memory
  o Exploit read only data with the constant memory

• Reuse data in registers
  o Use unroll directives with various options (jam, guarded, split, …)

• ... example:

```c
#pragma hmppcg grid blocksize 16 X 16
#pragma hmppcg unroll 2, jam
for (i = 0; i < M; i += 1) {
  #pragma hmpp unroll 4, guarded
  for (j = 0; j < N; j += 1) {
    a[i][j] = cos(a[i][j]) + cos(b[i][j]) - initValue;
  }
}
```
Loop Properties
PARALLEL & NOPARALLEL Directives (1)

- Force HMPP to consider a loop as being intrinsically parallel or nonparallel

- Use PARALLEL to force the gridification of a loop (or two) when HMPP does not automatically detect parallelism (ideally never)
  - GRIDIFY directive is most of the time more powerful, especially in loop nests

- Use NOPARALLEL to prevent the gridification
  - For the same reason, prefer GRIDIFY and use NOPARALLEL only for single loops
GRID Directive: Blocksize Attribute

- Allow the user to control the number of threads per block for the next loop nest
- Use the directive before a loop nest

```cpp
#pragma hmppcg grid blocksize "nxm"
...
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
  {
    ...
  }
```

- Where `n` is the first dimension of the block
- Where `m` is the second dimension of the block
- The default block size is 32x4 in 2D and 64x1 in 1D
CONSTANT Memory

- Allows user to address specific constant memory for CUDA and OpenCL target
- Used to put data in read-only memory
  - Filter coefficient
  - Constant value
- Place the directive inside the codelet definition

```c
#pragma hmpp mycodelet codelet, ...
Void myFunc (int var[n], ...) {
  #pragma hmppcg constantmemory var, n*sizeof(int)
  ...
}
```

- **Var**: variable to put in constant memory, parameter of the codelet
- **Size**: size of the data in bytes (optional)
Loop Transformation: Focus on Unroll
UNROLL Directive

- A directive placed just before the loop

```
#pragma hmppcg unroll var:factor, policy, noremainder
for(i=0;i<n;i++)
{

...
}
```

- Multiply the 'step' of the loop by the given constant factor
  - But each iteration executes *factor* bodies
- *var* identify the induction name of the loop to unroll (here: i)
- *policy* attribute: *contiguous* (default), *split*, *changestep*
  - Only on well-formed *for* or *DO* loops
  - Remark: an unroll of 1 does nothing!
UNROLL Directive : Contiguous Attribute

- Default unrolling case (or contiguous attribute)

```c
#pragma hmppcg unroll i:3
for (i=0;i<90;i++)
{
    ... calc(i)
}
```

```c
(!$hmppcg unroll i:3
DO i=1,90
    ... calc(i)
ENDDO
```

```c
for(i=0;i<90/3;i+=1)
{
    ... calc(3*i)
    ... calc(3*i+1)
    ... calc(3*i+2)
}
```

```c
DO i=1,90/3,1
    ... calc(3*i)
    ... calc(3*i+1)
    ... calc(3*i+2)
ENDDO
```
UNROLL Directive : Reuse Expression

- To reuse identical computations

```c
!$hmppcg unroll i:3
DO i=1,90
A(i)=cos((i-1)*x)-cos(i*x)+cos((i+1)*x)
ENDDO

DO i=1,30,1
A(i+0)=cos((3*i-1)*x)-cos((3*i+0)*x)+cos((3*i+1)*x)
A(i+1)=cos((3*i+0)*x)-cos((3*i+1)*x)+cos((3*i+2)*x)
A(i+2)=cos((3*i+1)*x)-cos((3*i+2)*x)+cos((3*i+3)*x)
ENDDO
```
UNROLL Directive: Reuse Memory Access

- To reuse identical memory reads (e.g. convolution, ...)

```c
!$hmppcg unroll i:3
DO i=1,90
    A(i) = 3*B(i-1) - 4*B(i) + 3*B(i+1)
ENDDO
```

- Beware with GPU coalescing ...

```c
DO i=1,30,1
    A(i+0) = 3*B(3*i-1) - 4*B(3*i+0) + 3*B(3*i+1)
    A(i+1) = 3*B(3*i+0) - 4*B(3*i+1) + 3*B(3*i+2)
    A(i+2) = 3*B(3*i+1) - 4*B(3*i+2) + 3*B(3*i+3)
ENDDO
```
UNROLL Directive: the remainder loop

- In previous slides, the number of iterations is a multiple of the unrolling factor
- Otherwise, remove the NOREMAINDER attribute
  - to add an additional loop to finish the work

- In case of doubt
  - Don't use NOREMAINDER
  - Else the program may be incorrect
UNROLL Directive: the SPLIT attribute (1)

- Split the iteration space before unrolling

```
!$hmppcg unroll i:5, split
DO i=1,100
    ... calc(i)
ENDDO
```

```
DO k=1,20
    ... calc( 0+k)
    ... calc(20+k)
    ... calc(40+k)
    ... calc(60+k)
    ... calc(80+k)
ENDDO
```
UNROLL Directive: the SPLIT attribute (2)

• For GPUs, SPLIT is usually recommended to unroll the loop on which the Warps are mapped
  o To keep a good coalescing

• 1st (half)-warp from $k=1$ to $k=16$
• That warp is accessing the marked statement as
  » $A(501), A(502), A(503), \ldots, A(516)$
• This is contiguous so well coalesced
• For a regular unrolling that would be
  » $A(3), A(7), A(11), \ldots, A(59), A(63)$

```c
!$hmppcg unroll i:4, split
DO i=1,1000
  A(i) = 1
ENDDO
```
```
DO k=1,250
  A( 0+k) = 1
  A(250+k) = 1
  A(500+k) = 1
  A(750+k) = 1
ENDDO
```
UNROLL Directive: the JAM attribute (1)

- Unrolling can be applied to multiple loops
- But the result may be ... Unexpected

```cpp
%!hmppcg unroll i:2, j:2, noremainder
DO i=1,n
  DO j=1,m
    ... calc(i,j)
  ENDDO
ENDDO

DO i=1,n,2
  DO j=1,m,2
    ... calc(i+0,j+0)
    ... calc(i+0,j+1)
  ENDDO
DO j=1,m,2
  ... calc(i+1,j+0)
  ... calc(i+1,j+1)
ENDDO
ENDDO
```
UNROLL Directive: the JAM attribute (2)

- Use the JAM attribute to jam the unrolled bodies
- Eventually use a Variable to control which level of loops must be merged

```c
!$hmppcg unroll i:2, j:2, jam, noremainder
DO  i=1,n
   DO  j=1,m
      ...  calc(i,j)
   ENDDO
ENDDO
```

```c
DO  i=1,n,2
   DO  j=1,m,2
      ...  calc(i+0,j+0)
      ...  calc(i+0,j+1)
      ...  calc(i+1,j+0)
      ...  calc(i+1,j+1)
   ENDDO
ENDDO
```
Programming Hardware Accelerators with HMPP: Grouping Codelets
What can I do with my Hardware Accelerator?

- I can thread functions on a HWA:
  - To share data and save memory or transfers
Standalone Codelets

- A standalone codelet monopolizes its HWA
Grouped Codelets

- Multiple pairs CODELET/CALLSITE can be grouped on a same HWA to share resources
The GROUP directive

• Create a group of codelets
  o Since then, all the codelets belonging to the group will share the same HWA

• Syntax:

```
#pragma hmpp <groupLabel> group, ...
```
Directive syntax within groups

• Two new syntaxes
  o The first one applies to the whole group:

  ```c
  #pragma hmpp <groupLabel> 'command' , ...
  !$hmpp <groupLabel> 'command' , ...
  ```

  o The second one applies to a specific codelet:

  ```c
  #pragma hmpp <groupLabel> codeletLabel 'command' , ...
  !$hmpp <groupLabel> codeletLabel 'command' , ...
  ```

• Within a group, the codelet label is:
  o Mandatory for CALLSITE, CODELET, SYNCHRONIZE
  o Optional for ADVANCEDLOAD and DELEGATEDSTORE
  o Illegal for all other directives
Referencing arguments in a group

- When a codelet label is specified, reference argument as usual:

\[
\texttt{#pragma hmpp <myGroup> cdl1t1 advancedload, args}\left[A;B\right]
\]

- Otherwise prefix all arguments by the “codeletLabel::”

\[
\texttt{#pragma hmpp <myGroup> advancedload, args[cdlt1::A; cdlt2::B]}
\]

- Use ‘*’ to refer to all codelets:

\[
\texttt{#pragma hmpp <myGroup> advancedload, args[*:A; *:B]}
\]
Sharing Data between Codelets
Argument Mapping
#pragma hmpp <myGroup> group

#define pragma hmpp <myGroup> f1 codelet, args[B].io=out
void f1( int n, float A[n], float B[n] )
{
    int i;
    for (i=0 ; i<n ; ++i)
        A[i] = B[i] + 1 ;
}

#define pragma hmpp <myGroup> f2 codelet, args[C].io=out
void f2( int n, float S[n], float C[n] )
{
    int i;
    for (i=0 ; i<n ; ++i)
        C[i] = S[i] * 2 ;
}

float XA[10], XB[10], XC[10] ;
#pragma hmpp <myGroup> f1 callsite
f1( 10, XA, XB ) ;
#pragma hmpp <myGroup> f2 callsite
f2( 10, XB, XC ) ;

f1 produce XB and f2 consume XB
Problem: duplicated on GPU
f1::B and f2::S
are the same object XB on the CPU
f1::B  <=>  f2::S
How to have a single object on the HWA?
Solution:
By using MAP, you can associate arguments with different names.

On the GPU
B & S are now the same object and have the same value for both f1 and f2.
The MAP directive

- MAP arguments in the group
  - The group considers these data equivalents
    - Same type, same size, same value at any time
  - Only one allocation on the device
  - Enables transfer optimizations

- Syntax:
  ```
  #pragma hmpp <myGroup> map, args[cdlt1::A; cdlt2::B]
  ```

- The types and dimensions of all mapped arguments must be identical
The MAPBYNAME directive

- Map together all the arguments sharing the given name in the all codelets of the group
  - A single object on the HWA will be created for all codelets

- Syntax example:
  
  ```cpp
  #pragma hmpp <myGroup> mapbyname, A
  ```

- The types and dimensions of all mapped arguments must be identical
CALLSITE: the NOUPDATE attribute

- Prevent all the implicit loads and stores on some arguments
- Syntax:
  
  ```
  #pragma hmpp <myGroup> myLabel callsite, args[A;B].noupdate=true
  ```

- Explicit ADVANCEDLOAD and DELEGATEDSTORE are still performed
- No synchronization on asynchronous transfers
- Avoid transferring intermediate results to and from the CPU
Keep data on HWA using MAP and NOUPDATE

- Example: Y is never exchanged with the CPU

```c
#pragma hmpp <MyGroup> group
#pragma hmpp <MyGroup> map, args[Pass1::B;Pass2::A]

#pragma hmpp <MyGroup> Pass1 codelet, args[B].io=out
void pass1( int n, float A[n], float B[n]) {
...
}

#pragma hmpp <MyGroup> Pass2 codelet, args[B].io=out
void pass2( int n, float A[n], float B[n]) {
...
}

float X[100], Y[100], Z[100] ;
#pragma hmpp <MyGroup> allocate
#pragma hmpp <MyGroup> Pass1 callsite, args[2].noupdate=true
pass1(100,X,Y) ;
#pragma hmpp <MyGroup> Pass2 callsite, args[1].noupdate=true
pass2(100,Y,Z) ;
#pragma hmpp <MyGroup> release
```
HMPP Runtime Control
Codelet Directives: the COND attribute (1)

• Specify a condition for selecting codelets

```
#pragma hmpp myLabel codelet, cond="n>1000"
```

• The condition will be evaluated at run time
• Multiple times in all directive contexts
  o Once for ALLOCATE, once for CALLSITE, …
  o Each iteration…
• Except in the CODELET directive context
• Beware of side effects (n++, I/O, …)

```
#pragma hmpp myLabel codelet, cond="(++n)>1000"
```
Codelet Directives: the COND attribute (2)

• Simple example of 2 competing callsites:

```fortran
SUBROUTINE myFunc(n,A,B)
    INTEGER, INTENT(IN) :: n, A(n)
    INTEGER, INTENT(OUT) :: B(n)
    INTEGER :: i
    ...
END SUBROUTINE

PROGRAM test
    INTEGER :: sz
    ...
    !$hmp Foo1 decodelet, target=CUDA, cond="sz>1024"
    !$hmp Foo2 codelet, target=SSE, cond="sz<=1024"
    CALL myFunc(sz,X,Y)
    ...
END PROGRAM
```
HMPP Dynamic Mapping

- Allow to change target order for every codelets
  - Without recompiling the application

  
  $ export HMPP_DYNAMIC_MAPPING=myLabel_a:CUDA,SSE

- Dynamic Mapping file
  - A more readable implementation of the Dynamic Mapping when codelets become numerous

  
  $ export HMPP_DYNAMIC_MAPPING_FILE=/myDynamicMappingFile/dir
  $ cat > my_dynamic_mapping_file
  mylabel_a:CUDA
  mylabel_b:SSE
  ...
HMPP and other Parallel Programming Models
Mixing HMPP with MPI or OpenMP

- Each OpenMP thread or MPI process can use a GPU
  - HMPP runtime takes care of the allocation on a first-come-first-serve basis
  - Exploit multi-GPU nodes

- Caveat: should not have more OpenMP threads or MPI processes than GPU processors
  - Some threads/processes will use GPU and be fast
  - Some threads/processes will use CPU and be slower
Multiple Parallelism Levels

- Adding a new layer of specific hardware is adding a new workload to the developer.
- Programming various hardware components of a node cannot be done separately.
HMPP raw performance
HMPP BLAS Performance on NVIDIA Fermi

SGEMM Performance

2 x Intel(R) Xeon(R) X5560 @ 2.80GHz (8 cores) - MKL
NVidia Tesla C2050, ECC activated – HMPP, CUBLAS, MAGMA
HMPP BLAS Performance on NVIDIA Fermi

DGEMM Performance

2 x Intel(R) Xeon(R) X5560 @ 2.80GHz (8 cores) - MKL
NVidia Tesla C2050, ECC activated – HMPP, CUBLAS, MAGMA
HMPP BLAS Performance on NVIDIA Fermi

ZGEMM Performance

2 x Intel(R) Xeon(R) X5560 @ 2.80GHz (8 cores) - MKL

NVidia Tesla C2050, ECC activated – HMPP, CUBLAS
Back to Tools to improve the methodology productivity
Tools in the Methodology: Phase 1

Define your parallel project
- Profiling tools: Gprof, KacheGrind, Vampir, Acumem, ThreadSpotter
- Debugging tools: gdb, alinea DDT, TotalView

Port your application on GPU
- HMPP Workbench
  - HMPP Wizard

Optimize your GPGPU application
- Parallel Profiling Tools: Paraver, TAU, Vampir
- HMPP Performance Analyzer
- Target Specific tools: Nvidia Insight

Hotspots
- Hours to Days

Parallelization
- Days to Weeks

Tuning
- Weeks to Months

Phase 1
- GPGPU operational application with known potential

Phase 2
Build a Parallel Project: HMPP Wizard
Make your kernels more GPU friendly
HMPP Wizard

An interactive diagnosis tool to make the code “GPU Friendly”

• Features:
  o Generate GPU porting advices based on static analysis
    • Unsupported programming (pointers, goto-labels, …)
    • Loop structure and parallelism validation
    • Diagnosis the loop nest translation process into a GPU grid of threads
  o Estimate kernels behavior on a GPU and provide specific advices
    • Reduction inside a kernel;
    • Bad memory coalescing;
    • Inefficient memory coalescing;
    • Too low computation density;
    • Detection of well-known patterns such as convolutions
  o Propose basic GPU Tuning directives

• Core technology:
  o Analyze memory access patterns on the GPU
  o Takes into account the full HMPP code generation process
    • Evaluates HMPPCG optimization directives
Analyze your memory access pattern

Use HMPPCG Directives and make your kernel GPU friendly
The Wizard provides help for users not familiar to GPU programming

Static analysis at two Levels:
1. An HMPP Diagnosis
   - Kernel Validation
   - Transformation applied
   - Grid computed

Tune your application with the HMPP Wizard

```c
#pragma hmp <densRad>
jacobiSolver codelet

void jacobiSolver(struct _coord res, float divergence[ res.m_z ][ res.m_y ][ res.m_x ], float pr)
{
    int i, j, k;
    int xMax = res.m_x, yMax = res.m_y, zMax = res.m_z;
    for (k = 1; k < zMax - 1; k++)
    {
        for (j = 1; j < yMax - 1; j++)
        {
            for (i = 1; i < xMax - 1; i++)
            {
                float div = divergence[k][j][i];
                float dampingPres = 0.9f * pressure[k][j][i];
                float obstL = obstacles[k][j][i - 1];
                float obstR = obstacles[k][j][i + 1];
                float pL = (1.f - obstL) * dampingPres + obstL * pressure[k][j][i - 1];
                float pR = (1.f - obstR) * dampingPres + obstR * pressure[k][j][i + 1];
                float obstB = obstacles[k][j - 1][i];
                float obstF = obstacles[k][j + 1][i];
                float pB = (1.f - obstB) * dampingPres + obstB * pressure[k][j - 1][i];
                float pF = (1.f - obstF) * dampingPres + obstF * pressure[k][j + 1][i];
                float obstD = obstacles[k - 1][j][i];
                float obstU = obstacles[k + 1][j][i];
                float pD = (1.f - obstD) * dampingPres + obstD * pressure[k - 1][j][i];
                float pU = (1.f - obstU) * dampingPres + obstU * pressure[k + 1][j][i];
                tmpPressure[k][j][i] = (pL + pR + pB + pF + pD + pU - div) / 6.0f;
            }
        }
    }
}
```
Tune your application with the HMPP Wizard

2. An advice: is the loop GPU friendly?
   - Example: memory access pattern over the Z dimension (3D)
     - The GPU grid is 2D → the coalescing must be on the X dimension
   - A pragma permute is proposed to fix the problem

```c
#pragma hmppcg permute(k,i,j)
for (k = 0 ; k < zMax ; k++)
  for (j = 0 ; j < yMax ; j++)
    for (i = 0 ; i < xMax ; i++)
      {
        float xTrace = i - DTO * velx[k][j][i];
        float yTrace = j - DTO * vely[k][j][i];
        float zTrace = k - DTO * velz[k][j][i];
        xTrace = fminf(fmaxf((xTrace), (1.5f)), (xMax - 2.5f));
        yTrace = fminf(fmaxf((yTrace), (1.5f)), (yMax - 2.5f));
        zTrace = fminf(fmaxf((zTrace), (1.5f)), (zMax - 2.5f));
        int x0 = (int) xTrace;
```
Methodology: Tuning Phase
Tools in the Methodology: Phase 2

- **Hotspots**
  - Profiling tools: Gprof, Kachegrind, Vampir, Acumem, ThreadSpotter
  - Debugging tools: gdb, alinea DDT, TotalView

- **Parallel Profiling Tools**
  - Paraver, TAU, Vampir

- **Parallelization**
  - HMPP Workbench
  - HMPP Wizard

- **Define your parallel project**
  - Hours to Days

- **Port your application on GPU**
  - Days to Weeks

- **Optimize your GPGPU application**
  - Weeks to Months

- **Phase 1**
  - GPGPU operational application with known potential

- **Phase 2**
  - Define your parallel project
  - Port your application on GPU
  - Optimize your GPGPU application
Optimize GPU Performance:
HMPP Performance Analyzer
Understand how well kernels are performing
HMPP Performance Analyzer

Provide information about the GPU behavior at source level for
HMPP applications

• Main features
  o Analyze execution profiles made on the GPU by target specific profilers
  o Compute synthetic metrics from raw profile data
    • Memory throughput, grid size, computation density…
  o Provide detailed statistics from raw profile data

• Benefits
  o Avoid using target specific profiling tools for HMPP applications
    • Nvidia insight profiles Cuda code !
  o Get precise and specific information about the loop nest behavior
  o Explore and exploit at best the GPU power from the source level
  o Fine tune kernels using HMPPCG directives
Get precise and specific information about the kernel behavior.

Explore and Exploit at best the GPU power from the C source level.
HMPP Performance Analyzer: example

Global view of the execution time distribution

Details statistics of raw profiling data

Synthetic kernel metrics
What’s else in HMPP?

• Sections (Partial transfers)
  o Split your transfers to fit with hardware
• Regions and resident variables (globals)
  o Create a Codelet without any function!
• Automatic Transfers
• Complex numbers
• Reductions
• Native functions
  o Hand-written CUDA or OpenCL kernels provided by the user
• External functions
  o External C of Fortran functions called from the codelet
• Alternative implementations
  o Third-party optimized library functions instead of HMPP’s generated code
• More native compilers
  o Icc, Ifort, Gcc, Gfortran…
What’s next in HMPP?

- Input code
  - C
  - Fortran 90
  - C++
  - Java

- Programming model
  - Open Runtime API
  - Argument resolution by address
  - Data Collection Distribution
  - Automatic Data Distribution

- Targeted accelerators
  - CUDA (Nvidia)
  - OpenCL
  - CAL/IL (AMD/ATI)

- Targeted Operating Systems
  - Linux
  - Windows

123
Conclusion
Conclusion

• Standardization effort is going on
  o OpenCL from Kronos
  o OpenMP accelerator subcommittee
  o HMPP as an Open Standard

• High level GPU code generation allows many GPU code optimization opportunities
  o Easier to tune applications at high level

• Hardware cannot be totally hidden to programmers
  o e.g. exposed memory hierarchy
  o Efficient programming rules must be clearly stated

• Quantitative decisions as important as parallel programming
  o Performance is about quantity
  o Fine tuning is (unfortunately) specific to a GPU configuration
OpenHMPP – http://www.openhmpp.org – CAPS