Programming Heterogeneous Many-cores Using Directives

HMPP - OpenAcc

F. Bodin, CAPS CTO
Introduction

• Programming many-core systems faces the following dilemma
  o Achieve "portable" performance
    • 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
    • Massive parallelism
      – Tens of thousands of threads needed
  o The constraint of keeping a unique version of codes, preferably mono-language
    • Reduces maintenance cost
    • Preserves code assets
    • Less sensitive to fast moving hardware targets
    • Codes last several generations of hardware architecture
• For legacy codes, directive-based approach may be an alternative
  o And may benefit from auto-tuning techniques
Profile of a Legacy Application

- Written in C/C++/Fortran
- Mix of user code and library calls
- Hotspots may or may not be parallel
- Lifetime in 10s of years
- Cannot be fully re-written
- Migration can be risky and mandatory

```c
while(many){
    ...
    mylib1(A,B);
    ...
    myuserfunc1(B,A);
    ...
    mylib2(A,B);
    ...
    myuserfunc2(B,A);
    ...
}
```
Overview of the Presentation

• Many-core architectures
  o Definition and forecast
  o Why usual parallel programming techniques won't work per se

• Directive-based programming
  o OpenACC sets of directives
  o HMPP directives
  o Library integration issue

• Toward a portable infrastructure for auto-tuning
  o Current auto-tuning directives in HMPP 3.0
  o CodeletFinder for offline auto-tuning
  o Toward a standard auto-tuning interface
Many-Core Architectures
Heterogeneous Many-Cores

• Many general purposes cores coupled with a massively parallel accelerator (HWA)

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

CPU and HWA linked with a PCIx bus
Where Are We Going?

Evolution of Processing Units in Future Processors

- **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**
- **CPU**
- **GPU**

- * Frequency based on Intel Processor (max.)
- ** Number of Processing Units (cores x threads)
- Intel CPU, NVIDIA GPU

**Forecast**
Heterogeneous Architecture Space

• Achieving "portable" performance

• Heterogeneity
  • Different parallel models
  • Different ISAs
  • Different compilers
  • Different memory systems
  • Different libraries

• A code must be written for a set of hardware configurations
  • 6 CPU cores + MIC
  • 24 CPU cores + GPU
  • 12 cores + 2 GPUs
  • ...

code need to move in this space and new HWs to come

Fat cores - OO

X86 multi-cores

Intel MIC

Light cores

NVIDIA/AMD GPUs

SIMT cores
Exploiting heterogeneous many-core with MPI parallel processes
  - Extra latency compared to shared memory use
    - MPI implies some copying required by its semantics (even if efficient MPI implementations tend to reduce them)
    - Cache trashing between MPI processes
  - Excessive memory utilization
    - Partitioning for separate address spaces requires replication of parts of the data
    - When using domain decomposition, the sub-grid size may be so small that most points are replicated (i.e. ghost cells)
    - Memory replication implies more stress on the memory bandwidth which finally prevent scaling

Exploiting heterogeneous many-core with thread based APIs
  - Data locality and affinity management non trivial
  - Reaching a tradeoff between vector parallelism (e.g. using the AVX instruction set), thread parallelism and MPI parallelism
  - Threads granularity has to be tuned depending on the core characteristics (e.g. SMT, heterogeneity)
  - Most APIs are shared memory oriented
Domain Decomposition Parallelism

32x32x32 cell domain
ghost cells 2 →
ghost cells / domain cells = 0.42

1 process → 8 processes

16x16x16 cell domain
ghost cells 2 ←
ghost cells / domain cells = 0.95
Flexible Code Generation Required

- The parallel programming API must not assume too much about the HW targets.

Cluster Level APIs
- MPI, PGAS, ...

Threads APIs
- OpenMP
- Intel TBB, Cilk, ...

Accelerator Directives
- HMPP, OpenACC

Accelerator Languages
- CUDA, OpenCL

X86 multi-core
- Intel MIC
- NVIDIA/AMD GPU
Auto-Tuning is Required to Achieve Some Performance Portability

- The more optimized a code is, the less portable it is
  - Optimized code tends to saturate some hardware resources
  - Parallelism ROI varies a lot
    - i.e. # threads and workload need to be tuned
  - Many HW resources not virtualized on HWA (e.g. registers, #threads)

Example of an optimized versus a non optimized stencil code

CC 2012

www.caps-entreprise.com
Directive-based Programming
Directives-based Approaches

- Supplement an existing serial language with directives to express parallelism and data management
  - Preserves code basis (e.g. C, Fortran) and serial semantic
  - Competitive with code written in the device dialect (e.g. CUDA)
  - Incremental approach to many-core programming
  - Mainly targets legacy codes

- Many variants
  - HMPP
  - PGI Accelerator
  - OpenACC
  - OpenMP Accelerator extension
  - ...

- OpenACC is a new initiative by CAPS, CRAY, PGI and NVidia
  - A first common subset
OpenACC Initiative

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

• Main OpenACC constructs
  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 April 2012 (beta available)

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

- Mirroring duplicates a CPU memory block into the HWA memory
  - Mirror identifier is a 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 describes iteration space partitioning to execute the loop; declares loop-private variables and arrays, and reduction operations

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

```c
#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)
  - Each loop nest can have different values for gang and worker numbers

- Clauses
  - if(condition)
  - async[(scalar-integer-expression)]
  - 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 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 without barrier
- Clauses
  - if(condition)
  - async[(scalar-integer-expression)]
  - num_gangs(scalar-integer-expression)
  - num_workers(scalar-integer-expression)
  - vector_length(scalar-integer-expression)
  - reduction(operator:list)
  - 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)
  - private(list)
  - 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
  o Explicit management of data transfers using clauses or directives

• Many clauses
  o if(condition)
  o copy(list)
  o copyin(list)
  o copyout(list)
  o 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)

```cpp
#pragma acc data copyin(A[1:N-2]),
    copyout(B[N])
{
    #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] = ...;
    }
}
```
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
DNA Distance Application with OpenACC

• Biomedical application part of Phylip package,
  o Main computation kernel takes as input a list of DNA sequences for each species
    • Code is based on an approximation using Newton-Raphson method (SP)
    • Produces a 2-dimension matrix of distances
  o Experiments performed in the context of the HMPP APAC CoC*

• Performance
  o OpenMP version, 4 & 8 threads, Intel(R) i7 CPU 920 @ 2.67GHz
  o 1 GPU Tesla C2070

*http://competencecenter.hmpp.org/category/hmpp-coc-asia/
• Codelet and region based directives for many-cores
  ○ CUDA, OpenCL code generation, soon Intel MIC, x86

```c
main()
{
  ...
  #pragma hmpp f1 callsite
  myfunc(V1[k], V2[k]);
  ...
}
```

```c
#pragma hmpp f1 codelet
myfunc(...){
  ...;
  for();
  for();
  ...
  ...;
}
```
What is in HMPP and not in OpenACC

• Multiple devices management
  o Data collection / map operation

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

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

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

• And many more features
  o Native functions, buffer mode, UVA support, codelets, …
Library Integration
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, ArrayFire)
  o No access to all application codes (i.e. avoid side effects)
  o **Want a unique source code**

• **Deal with multiple address spaces / multi-HWA**
  o Data location may not be unique (copies, mirrors)
  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.

• **Mostly an engineering issue**
Library Mapping Example

```c
fftw_plan  fftwf_plan_dft_r2c_3d(
    sz, sy, sx,
    work1, work2,
    FFTW_ESTIMATE);

fftwf_execute(p);

fftwf_destroy_plan(p);
```

```c

NVIDIA cuFFT

cufftHandle  plan;
cufftPlan3d(&plan,sz,sy,sx,CUFFT_R2C);

cufftExecR2C(plan,(cufftReal*) work1,
          (cufftComplex *) work2);

cufftDestroy(plan);
```
Proxy Directives "hmppalt" in HMPP3.0

- A proxy indicated by a directive is in charge of calling the accelerated library
- Proxies get the execution context from the HMPP runtime
- Proxies are used only to selected calls to the library

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

!$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)
```

Replaces the call to a proxy that handles GPUs and allows to mix user GPU code with library ones.
Library Interoperability in HMPP 3.0

```
... call libRoutine1(...) ...
... #pragma hmppalt call libRoutine2(...) ...
... call libRoutine3(...) ...
... call libRoutine2(...) ...
... proxy2(...) ...
... cpuLib1(...) ...
... cpuLib3(...) ...
... gpuLib(...) ...
```
Toward a Portable Auto-Tuning Infrastructure
Auto-Tuning

- Need to create an optimization space to explore
  - Auto-tuning capabilities intrinsically limited by coding APIs
  - Code generation must have a lot of freedom to deal with heterogeneous systems
  - Auto-tuning has to be integrated into parallel programming

- Need a way to explore optimization space
  - Not a compiler infrastructure issue

- Auto-tuning strategy
  - Online approach
    - JIT, Versioning
  - Offline approach
    - CodeletFinder
  - Mixed

- Separation of code generation/optimization infrastructure and exploration infrastructure is important
  - Many different ways to explore the optimization space (e.g. serial versus distributed)

not an issue considered here
many existing works
Auto-Tuning Approach for Heterogeneous HW

- Directive-based approach is pertinent
  - But directives need to be "high-level" but not too abstract

- Some issues are local
  - e.g. kernel optimizations

- Some issues are global
  - e.g. data movements, libraries

- Infrastructure needs to be compiler independent

- Exploration engine can exist in many configurations
  - Parallel exploration of the optimization space
  - Sequential exploration
  - Many strategies (e.g. random, ML)

Parallel HW independent code e.g. C, Fortran

Parallel dep. code e.g. CUDA, OpenCL

High level information cannot be reconstructed
code generation to get closer to HW code
Auto-Tuning in HMPP 3.0, a First Step

- Current approach based on code versioning
- Implementation can target multiple accelerator kinds

```
#pragma hmpp sgemm codelet,
   target=CUDA:OpenCL:MCPU,
   args[vout].io=inout
void func(int m,int n,int k,float alpha,
   const float vin1[n][n],
   const float vin2[n][n],...);
```
Global Auto-Tuning Infrastructure @ CAPS

- **CodeletFinder**
  - Off-line auto-tuning
- **HMPP Wizard**
  - Tuning advice
- **Tuning directives**
  - `hmppcg` set of directives
- **Exploration engine**
  - Runtime tool

![Diagram showing the flow of inputs and outputs from Application Code, through CodeletFinder, HMPP Wizard, Tuning Directives, Compiler, Executable program, and Exploration Engine.](image-url)
Code Tuning Directives

- Directive-based HWA kernel code transformations
- Directives preserve original 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);
      }
   }

1D gridification
Using 64 threads

Loop transformation
```
Auto-Tuning Example – 1*

- **HMPP-transformed PolyBench codes using CUDA and OpenCL:**
  - Given in terms of speedup over default (non-transformed) HMPP code
  - Compared with results of manually-written CUDA/OpenCL implementation
  - HMPP transformations gives speedup over default in 8 of the 14 transformed codes using CUDA and 6 of the 14 codes using OpenCL

- **CUDA Results:**

![Bar chart showing speedup comparison between best HMPP-transformed and manually-written PolyBench CUDA codes over default HMPP CUDA configuration](image)

*Figure 2: Speedup of best HMPP-transformed and manually-written PolyBench CUDA codes over default HMPP CUDA configuration*

*From "Autotuning a High-Level Language Targeted to GPU Kernels", S. Grauer-Gray, R. Searles, L. Xu, S. Ayalasomayajula, J. Cavazos Supercomputing 2011, University of Delaware*
Auto-Tuning Example – 2*

*From "Autotuning a High-Level Language Targeted to GPU Kernels", S. Grauer-Gray, R. Searles, L. Xu, S. Ayalasomayajula, J. Cavazos
Supercomputing 2011, University of Delaware
Simple Auto-tuning Directive in HMPP 3.0

- Provide an extension of the callsite directive to allow versioning
  - Declaration of multiple codelets
  - Declaration of the runtime selector expression
- Search engine is part of the application
  - Simple implementation, user function based

```
#pragma hmpp <group> clabel callsite
variants(variantLabel1,variantLabel2, ...)
selector(variantSelector)
functioncall(......)
```

Integer expression to select variant at runtime
void filterStencil5x5_T2050(const uint32 p_heigh[1],
                          const uint32 p_width[1], const RasterType filter[5][5],
                          const RasterType *p_inRaster, RasterType *p_outRaster)
{
    ...  
#pragma hmppcg grid blocksize "64x4"
#pragma hmppcg unroll 4, jam
    for (i = stencil; i < heigh - stencil; i++) {
        for (j = stencil; j < width - stencil; j++) {
            RasterType v;
            v = filter[0][0] * inRaster[i-2][j-2] + filter[0][1] *
                outRaster[i][j] = v;
        }
    }
}
Tuning Stencil Example - 2

- Tesla C1060 optimized version

```c
void filterStencil5x5_C1060(const uint32 p_heigh[1],
    const uint32 p_width[1], const RasterType filter[5][5],
    const RasterType *p_inRaster, RasterType *p_outRaster)
{
    ...
    #pragma hmppcg grid blocksize "32x4"
    #pragma hmppcg unroll 6, jam
    for (i = stencil; i < heigh - stencil; i++) {
        for (j = stencil; j < width - stencil; j++) {
            RasterType v;
            v = filter[0][0] * inRaster[i-2][j-2] + filter[0][1] â€¦
            outRaster[i][j] = v;
        }
    }
}
```
• Declare the variants at the callsite

```c
int filterVariantSelector = variantSelectorState(  
    "main-autotune.c@filterStencil5x5", 3);
...
kernelStart = wallclock();

#pragma hmpp <convolution> filter5x5 callsite variants( &  
#pragma hmpp & filterStencil5x5@<convolution>[C], &  
#pragma hmpp & filterStencil5x5_C1060@<convolution>[CUDA], &  
#pragma hmpp & filterStencil5x5_T2050@<convolution>[CUDA]) &  
#pragma hmpp & selector(filterVariantSelector)  
    filterStencil5x5(&fullHeigh, &width, stencil1, raster1, raster2);

kernelEnd = wallclock();
...
double kernelTime = kernelEnd - kernelStart;
variantSelectorUpdate(heigh, width, "main-autotune.c@filterStencil5x5",  
    filterVariantSelector, kernelTime);
```
CodeletFinder Paradigms

• Programs as a whole are fairly opaque and difficult to handle
  o Decomposing applications in hotspots
  o Each hotspot can be efficiently analyzed separately

• Performance-wise a code is a set of hotspots interacting together
  o Data flow make the link between the hotspots

• Extract codelets / regions and data sets to run them “in vitro”:
  o Don't have to run the whole application to tune/analyze a kernel
  o Can use "destructive test" to check the impact of some instructions
    • e.g. DECAN ("Decremental Performance Analysis Tool", Souad Koliai, UVSQ)
  o Help building reference kernels repository
  o Help checking performance on new hardware
  o Automation is key here
CodeletFinder Overview

Program

Hotspot 1

Hotspot 2

Can be compiled and executed in a standalone manner

Extracted memory data

Codelet wrapper

Extracted hotspots

Codelet 1

Codelet 2
CodeletFinder Process Overview

- For C and Fortran codes

**Project Capture**
- Captures build process
- Capture execution parameters
- Replays the build on demand

**Hotspot Finder**
- Finds hotspots in the application using execution profiles
- Statically extracts potential hotspots

**Codelet Builder**
- Builds the codelets based on identified hotspots
- Creates standalone micro-benchs
- Patterns are given to build the codelets

**Micro Bencher**
- Captures data for the micro-benchs
- Runs the micro-benchs

Performance, tuning and analysis tools plugged here
SUBROUTINE codelet_l6lyb3v7(nx, ny, nz, i, j, k, x, y, twiddle)
    IMPLICIT NONE
    INTEGER :: nx
    INTEGER :: ny
    INTEGER :: nz
    INTEGER :: i
    INTEGER :: j
    INTEGER :: k
    DOUBLE COMPLEX :: x(nx + 1, ny, nz)
    DOUBLE COMPLEX :: y(nx + 1, ny, nz)
    REAL*8 :: twiddle(nx + 1, ny, nz)
    CALL hmppcf_prologue()
    DO i=1, nz
        DO k=1, ny
            DO j=1, nx
                y(j, k, i) = y(j, k, i) * twiddle(j, k, i)
                x(j, k, i) = y(j, k, i)
            END DO
        END DO
    END DO
    CALL hmppcf_epilogue()
END SUBROUTINE codelet_l6lyb3v7
SUBROUTINE codelet_nj312bpm(n, m, ku, i, j, ln, t, ti, pi, exponent)
    IMPLICIT NONE
    INTEGER :: n
    INTEGER :: m
    INTEGER :: ku
    INTEGER :: i
    INTEGER :: j
    INTEGER :: ln
    DOUBLE PRECISION :: t
    DOUBLE PRECISION :: ti
    DOUBLE PRECISION :: pi
    DOUBLE COMPLEX :: exponent(n)
    CALL hmppcf_prologue()
    DO j=1, m
        t = pi / ln
        DO i=0, ln - 1
            ti = i * t
            exponent(i + ku) = dcmplx(cos(ti), sin(ti))
        END DO
        ku = ku + ln
        ln = 2 * ln
    END DO
    CALL hmppcf_epilogue()
END SUBROUTINE codelet_nj312bpm
CodeletFinder Status

- Successful experimented on various C and Fortran codes
  - Numerical recipes, NAS, SPECFEM3D, Reverse Time Migration, …
  - Can be used with MPI codes running in parallel

- Not yet a product
  - Full technology ready Q2 2012
  - Product to be released Q4 2012

- More experimentations needed
  - Work with ExaScale Computing Research (CEA, GENCI, Intel, UVSQ join entity)*

Toward a Standard Auto-Tuning Interface

- Should be compiler independent as much as possible
  - Multiple, target specific exploration engines need to be used

- What would provide a standard interface?
  - Decision point description
    - e.g. callsite
  - Variants description
    - Abstract syntax trees
    - Execution constraints (e.g. specialized codelets)
  - Execution context
    - Parameter values
    - Hardware target description and allocation
  - Runtime control to select variants or drive runtime code generation

- Hope to setup this effort in OpenHMPP consortium and the Autotune project (http://www.autotune-project.eu/)
Conclusion

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

• Auto-tuning is key to efficient portability
  o But a "standard" interface is required for the long term
  o Auto-tuning must be part of the many-core programming

• Need to integrate libraries and user codes
  o Requires a common backbone for user and library data, e.g. StarPU* or at least interoperability

*http://runtime.bordeaux.inria.fr/StarPU/
Many-core programming Parallelization GPGPU NVIDIA CUDA OpenHMPP
Directive-based programming Code Porting Methodology
OpenACC Hybrid Many-core Programming HPC community Petaflops
Parallel computing HPC open standard Exaflops OpenCL
High Performance Computing Code speedup Multi-core programming
Massively parallel Hardware accelerators programming GPGPU
HMPP Competence Center Parallel programming interface DevDeck

Global Solutions for Many-Core Programming

http://www.caps-entreprise.com