

A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures

Eduard Ayguade^{1,2}, Rosa M. Badia², Daniel Cabrera¹, Alejandro Duran^{1,2},
Marc Gonzalez^{1,2}, Francisco Igual³, Daniel Jimenez¹, Jesus Labarta^{1,2},
Xavier Martorell^{1,2}, Rafael Mayo³, Josep M. Perez²,
and Enrique S. Quintana-Orti³

¹ Universitat Politècnica de Catalunya (UPC)

² Barcelona Supercomputing Center (BSC-CNS)

³ Universidad Jaume I, Castellon

Abstract. OpenMP has evolved recently towards expressing unstructured parallelism, targeting the parallelization of a broader range of applications in the current multicore era. Homogeneous multicore architectures from major vendors have become mainstream, but with clear indications that a better performance/power ratio can be achieved using more specialized hardware (accelerators), such as SSE-based units or GPUs, clearly deviating from the easy-to-understand shared-memory homogeneous architectures. This paper investigates if OpenMP could still survive in this new scenario and proposes a possible way to extend the current specification to reasonably integrate heterogeneity while preserving simplicity and portability. The paper leverages on a previous proposal that extended tasking with dependencies. The runtime is in charge of data movement, tasks scheduling based on these data dependencies and the appropriate selection of the target accelerator depending on system configuration and resource availability.

1 Introduction and Motivation

Computer architecture is under a revolution. The gigahertz race has stopped due to power dissipation problems. Fortunately, extra transistors continue being available in new chip generations, thanks to the technological reduction in transistor area. So new solutions are needed to use the extra transistors, and get lower power consumption at the same time. As a nice alternative to the gigahertz race, and instead of going for more complex pipelined and superscalar processors, the new transistors are used to incorporate functionalities that were external to the processor, into a single chip. For instance, GPUs, which are currently being used for general purpose computing, are becoming part of the processing chip [1,2]. Other approaches, like the Cell/B.E. processor [3] are targeting physics, encryption, and encoding. The purpose is to accelerate specific algorithms, so that applications can take advantage of the extra performance.

Future supercomputers will be equipped with heterogeneous hardware, including Cell processors, GPUs and even FPGAs. This hardware can provide dramatic

performance advantages for high-performance computing applications, specially for applications featuring data-parallelism. In this scenario, programmers will have to deal with architectures composed by a mix of regular multicore CPUs and accelerators, possibly several types of accelerators, each one with its own programming environment and libraries, and possibly its own memory address space.

In order to overcome the programming challenges introduced by accelerator-based architectures, programming models need to evolve including features that allow to migrate applications to heterogenous architectures in a simple and portable way. If current application developers are still having a hard time trying to extract reasonable performance from homogeneous multicore architectures, the situation is about to get even worse with the emergence of heterogeneous multicore architectures.

The majority of proposals today assume a host-directed programming and execution model with attached accelerator devices. The bulk of a user application executes on the host while user-specified code regions are offloaded to the accelerator. In general, the specifics of the different accelerator architectures makes programming extremely difficult if one plans to use the vendor-provided SDKs (libspe for Cell, CUDA for Nvidia GPUs [4], ...). It would be desirable to retain most of the advantages of using these SDKs but in a much more accessible way, avoiding the mix of hardware specific code (for task offloading, data movement, ...) with application code.

In order to motivate the paper and show the complexities in using vendor-provided SDKs, we consider a simple matrix multiplication example in Figure 1. In this code, the programmer defines that each element of matrices A, B and C is a pointer to a block of $BS \times BS$ elements, which are allocated from inside the main function.

This simple example would require very different accelerator-dependent code in order to offload the execution, transfer data and synchronize host and accelerator(s). For example, to program an Nvidia GPU using CUDA, the programmer has to write the host and device codes. Figure 2 shows the code executed on the host, in which the programmer first needs to allocate memory on the GPU for the blocks of A, B and C (since the GPU executes from its own separate memory). Then the host copies matrices from host memory to GPU memory. The arguments give the destination pointer, the source pointer, the two dimension sizes, and copy direction. Then the host specifies the execution parameters and instantiates the kernel itself. Finally the host waits for the kernel to finish, moves results back from the GPU to the host and deallocates memory on the GPU.

For the Cell/B.E. the programmer would have to write two codes: one for the PPE (Figure 3) and one for the SPE (Figure 4). The PPE code handles thread allocation and resource management among the SPEs. In the example, the PPE code is creating a thread context, loading program, creates the thread and gets thread control for each of the SPEs. Then PPE code leaves the SPEs to perform the computation and afterwards waits for the SPE threads to finish. The SPE code iterates while there are matrix blocks to calculate. The function *next_block*

```

1 void matmul (float *A, float *B, float *C ) {
2   for (int i=0; i < BS; i++)
3     for (int j=0; j < BS; j++)
4       for (int k=0; k < BS; k++)
5         C[i*BS+j] += A[i*BS+k] * B[k*BS+j];
6 }
7
8 float *A[NB][NB], *B[NB][NB], *C[NB][NB];
9
10 int main( void ){
11   int i, j, k;
12
13   for(i = 0; i < NB; i++)
14     for(j = 0; j < NB; j++) {
15       A[i][j] = ( float* ) malloc(BS*BS*sizeof(float));
16       B[i][j] = ( float* ) malloc(BS*BS*sizeof(float));
17       C[i][j] = ( float* ) malloc(BS*BS*sizeof(float));
18     }
19
20   for (i = 0; i < NB; i++)
21     for (j = 0; j < NB; j++)
22       for (k = 0; k < NB; k++)
23         matmul ( A[i][k], B[k][j], C[i][j] );
24 }

```

Fig. 1. Matrix multiplication example to motivate the extension of OpenMP for heterogeneous architectures

```

1 __global__ void matmul_kernel( float *A, float *B, float *C );
2
3 #define THREADS_PER_BLOCK 16
4
5 void matmul (float *A, float *B, float *C ) {
6   ...
7   // allocate device memory
8   float *d_A, *d_B, *d_C;
9   cudaMalloc((void**) &d_A, BS*BS*sizeof(float));
10  cudaMalloc((void**) &d_B, BS*BS*sizeof(float));
11  cudaMalloc((void**) &d_C, BS*BS*sizeof(float));
12
13  // copy host memory to device
14  cudaMemcpy(d_A, A, BS*BS*sizeof(float), cudaMemcpyHostToDevice);
15  cudaMemcpy(d_B, B, BS*BS*sizeof(float), cudaMemcpyHostToDevice);
16
17  // setup execution parameters
18  dim3 threads(THREADS_PER_BLOCK, THREADS_PER_BLOCK);
19  dim3 grid(BS/threads.x, BS/threads.y);
20
21  // execute the kernel
22  matmul_kernel<<< grid, threads >>>(d_A, d_B, d_C);
23
24  // copy result from device to host
25  cudaMemcpy(C, d_C, BS*BS*sizeof(float), cudaMemcpyDeviceToHost);
26
27  // clean up memory
28  cudaFree(d_A);
29  cudaFree(d_B);
30  cudaFree(d_C);
31 }

```

Fig. 2. Matrix multiplication example (non optimized) targeting CUDA

```

1 void matmul_spe ( float *A, float *B, float *C );
2
3 void matmul ( float *A, float *B, float *C ) {
4 for (i=0; i<num spus; i++) {
5 // Initialize the thread structure and its parameters
6 ...
7 // Create context
8 threads[i].id = spe_context_create (SPE_MAP_PS, NULL);
9 // Load program
10 rc = spe_program_load (threads[i].id, &matmul_spe) != 0;
11 // Create thread
12 rc = pthread_create (&threads[i].pthread, NULL,
13 &ppu_pthread_function, &threads[i].id);
14 // Get thread control
15 threads[i].ctl_area = (spe_spu_control_area_t *)
16 spe_ps_area_get(threads[i].id, SPE_CONTROL_AREA);
17 }
18 // Start SPUs
19 for (i=0; i<spus; i++) send_mail(i, 1);
20 // Wait for the SPUs to complete
21 for (i=0; i<spus; i++)
22 rc = pthread_join (threads[i].pthread, NULL);
23 }

```

Fig. 3. Matrix multiplication example (non optimized and omitting conditional statements to control error codes) for Cell/B.E. using IBM's SDK: PPE side

```

1 void matmul_spe ( float *A, float *B, float *C )
2 {
3 ...
4 while (blocks_to_process()){
5     next_block(i, j, k);
6     calculate_address (baseA, A, i, k);
7     calculate_address (baseB, B, k, j);
8     calculate_address (baseC, C, i, j);
9     mfc_get(localA, baseA, sizeof(float)*BS*BS, in_tags, 0, 0);
10    mfc_get(localB, baseB, sizeof(float)*BS*BS, in_tags, 0, 0);
11    mfc_get(localC, baseC, sizeof(float)*BS*BS, in_tags, 0, 0);
12    mfc_write_tag_mask((1<<(in_tags)));
13    mfc_read_tag_status_all(); /* Wait for input data
14    for (ii=0; ii < BS; ii++)
15        for (jj=0; jj < BS; jj++)
16            for (kk=0; kk < BS; kk++)
17                localC[i][j] += localA[i][k] * localB[k][j];
18    mfc_put(localC, baseC, sizeof(float)*BS*BS, out_tags, 0, 0);
19    mfc_write_tag_mask((1<<(out_tags)));
20    mfc_read_tag_status_all(); /* Wait for output data
21    }
22 ...
23 }

```

Fig. 4. Matrix multiplication example for Cell/B.E. (non optimized) using IBM's SDK: SPE side

performs atomic counter increments between all SPEs. The SPE code performs three DMA read operations (for blocks from matrices A, B and C) and blocks until the transfers have finished. The block matrix operation is then performed locally and finally a DMA write operation is performed for the block from matrix C. This code is quite naive, since for example no double buffering nor SIMDization

```

1 void matmul (float *A, float *B, float *C ) {
2 // configure device
3 int al_desc = rasclib_algorithm_open("matrixmult");
4
5 // queues up command to send inputs
6 res = rasclib_algorithm_send(al_desc, "a", A, BS*BS*sizeof(float));
7 res = rasclib_algorithm_send(al_desc, "b", B, BS*BS*sizeof(float));
8 res = rasclib_algorithm_send(al_desc, "c", C, BS*BS*sizeof(float));
9
10 // queues up command to execute bitstream
11 rasclib_algorithm_go(al_desc);
12
13 // queues up command to receive results
14 res = rasclib_algorithm_receive(al_desc, "c", C, BS*BS*sizeof(float));
15
16 // wait for termination
17 rasclib_algorithm_committ(al_desc, NULL);
18 rasclib_algorithm_wait(al_desc);
19 }

```

Fig. 5. Matrix multiplication example (non optimized and omitting conditional statements to control error codes) targeting a RASC Altix blade

are used to optimize the code. The sample matrix multiply code from the IBM SDK has more than 600 lines for the PPE code and more than 1300 lines for the SPE code.

For a system like the SGI Altix with a Reconfigurable Application Specific Computing (RASC) FPGA blade, the code in Figure 5 using calls to the RASCLib library would be used (in addition to generate the bitstream `matrixmult` with the FPGA compiler). The code assumes that the application has already reserved and configured the FPGA device. The `rasclib_algorithm_open` allocates all necessary internal data structures for a logical algorithm. The three `rasclib_algorithm_send` calls pull data down to the input data areas on the device. The `rasclib_algorithm_go` starts execution. The `rasclib_algorithm_receive` call pushes the result back out to host memory. The `rasclib_algorithm_committ` causes all of the commands that have been queued up by the previous calls to be sent to the device. The `rasclib_algorithm_wait` blocks until all the command that were sent to all of the devices are complete, then returns.

There have been substantial efforts to propose and develop programming models for hybrid architectures that abstract the target architecture. These differ in the objects they manipulate, in general arrays or matrices. For example both RapidMind [5] and PeakStream are stream languages that operate on streams (vectors of arbitrary length) of data embedded in C or C++ and they are oriented to GPGPUs. For the ClearSpeed floating-point accelerator, the C^n programming language adds new datatypes (mono and poly) to indicate if there is only one instance of the data or all functional units have a portion of the data. Others (e.g. PGI) propose directives to delineate accelerator regions and extract parallelism in loops or follow a task parallelism model, and offer architecture independent abstractions for offloadable functions (e.g. Sequoia [6], Merge [7], CellSs [8], HMPP [9]). A growing number of OpenMP compiler frameworks are also intended to offer support for heterogeneous architectures (e.g. Octopiler [10] for Cell or PGI [11] and [12] for CUDA).

Most of the proposals and environments available target a single type of accelerator at a time. In order to have the same code prepared to compile for several target accelerators, the programmer needs to use conditional compilation to isolate declaration of variables and calls to different APIs for the different target devices.

2 Proposed OpenMP Extensions

In this section we propose a set of extensions to OpenMP 3.0 to express the execution of tasks on a hardware accelerator. This extension leverages on a previous proposal to allow the specification of dependencies between tasks [13], although it can be considered totally independent.

Tasks are the most important new feature of OpenMP 3.0. A programmer can define deferrable units of work, called tasks, and later ensure that all the tasks defined up to some point have finished.

```
#pragma omp task [clause-list]
    structured-block
```

Valid clauses are **shared**, **private**, **firstprivate** and **untied**. The first three are used for setting data sharing attributes of variables in the task body; the last one specifies that the task can be resumed by a different thread after a possible *task switching point*. The proposal in [13] extended the **task** construct with some additional clauses that are used to derive dependencies among tasks at runtime: **input**, **output** and **inout**. Although in some cases the compiler can analyze the code and determine the input and output data sets, we provided these additional clauses to modify or augment the compiler analysis.

OpenMP allows the specification of any structured block inside the **task** construct. This motivates the presentation of our proposal in two parts. The first part of our proposal just allows the specification of target devices for the execution of a task. In the second part, we consider a subset of the possible tasks than can be expressed in OpenMP: tasks composed of a function call. In this case, the programmer will be able to specify alternative implementations for different target devices. For the general case we have not found a portable way to specify alternative implementations (each one targeting an accelerator device) for structured blocks of code.

2.1 Specifying Target Devices

Our proposal consists of a new pragma that may precede an existing pragma **task**:

```
#pragma omp target device(device-name-list) [clause-list]
```

The **target** construct specifies that the execution of the task could be offloaded on any of the devices specified in **device-name-list** (and as such its code must

be handled by the proper compiler backends). If the task is not preceded by a **target** directive, then the default **device-name**, which is **smp** and corresponds to a homogeneous shared-memory multicore architecture, will be used. Other **device-names** are vendor specific (we will use along this paper three possible examples: **cell**, **cuda** and **fpga**). When a task is ready for execution (i.e. it has no dependencies with other previously generated tasks) the runtime can choose among the different available targets to decide in which device to execute the task. This decision is implementation-dependent but it will ideally be tailored to resource availability. If no resource is available, the runtime will stall that task until one becomes available.

Some restrictions may apply to tasks that target a specific device (for example, they may not contain any other OpenMP directives, do any input/output, ...). In addition, tasks offloaded in some specific devices should be tied or they should execute in the same type of device if thread switching is allowed.

Some additional clauses can be used with this pragma **device**:

- **copy_in**(data-reference-list)
- **copy_out**(data-reference-list)

These two clauses, which are ignored for the **smp** device, specify data movement for **shared** variables used inside the task. **Copy_in** will move variables in **data-reference-list** from host to device memory. **Copy_out** will move variable in **data-reference-list** back from device to host memory. Once the task is ready for execution, the runtime system will move variables in the **copy_in** list. Once the task finishes execution, the runtime will move variables in the **copy_out** list, if necessary.

A *data-reference* in a *data-reference-list* can contain a variable identifier or a reference to subobjects. References to subobjects include array element references (like **a[4]**), array sections (like **a[3:6]**), field references (like **a.b**) and shaping expressions (like **[10] [20] p**). Since C does not have any way to express ranges of an array, we have borrowed the *array-section* syntax from Fortran 90. These array sections, with syntax **a[e1:e2]**, designate all elements from **a[e1]** to **a[e2]** (both ends are included and **e1** shall yield a lower or equal value than **e2**). Multidimensional arrays are eligible for multidimensional array sections (like **a[1:2] [3:4]**). While not technically naming a subobject, non-multidimensional array section syntax can also be applied to pointers (i.e.: **pA[1:2]** is valid for **int *pA**, but note that **pB[1:2] [3:4]** is invalid for **int **pB**, also note that **pC[1:2] [3:4]** is valid for **int *a[N]** and so it is **pD[1:2] [3:4] [5:6]** for **int *a[N] [M]**). For syntactic economy **a[:x]** is the same as **a[0:x]** and, only for arrays where the upper bound is known, **a[x:]** and **a[:]** mean respectively **a[x:N]** and **a[0:N]**. Designating an array (i.e.: **a**) in a data reference list, with no array section nor array subscript, is equivalent to the whole array-section (i.e.: **a[:]**). Shaping expressions are a sequence of dimensions, enclosed in square brackets, and a data reference, that should refer to a pointer type (like **[10] [20] p**). These shaping expressions are aimed at those scenarios where an array-like structure has been allocated but only a pointer to its initial element

is available. The goal of shaping expressions is to provide to the compiler such unavailable structural information.

Other vendor-specific clauses in the `target` construct for each particular `device-name` are possible.

2.2 *Taskifying* Functions

Our second proposal applies to those tasks that are just composed of a function call

```
#pragma omp task [clause-list]
    function-call
```

In this paper we also consider another way to specify tasks in OpenMP, which we have found very convenient to *taskify* functions that are always executed as tasks:

```
#pragma omp task [clause-list]
    {function-header|function-definition}
```

Whenever the program calls a function annotated in this way, the runtime will create an explicit task.

In this case, the pragma proposed in the previous subsection applies to a function header or definition:

```
#pragma omp target device(device-name-list) [clause-list]
    {function-header|function-definition}
```

The `target` construct specifies that the function contains code prepared for its execution on all devices specified in `device-name-list`. If a function is not preceded by a `target` directive, then the default `omp` device is used. In addition to the possible clauses specified in the previous section, we allow in this case the following one:

- `implements(function-name)`

This clause `implements` is used to specify an alternative implementation for a function. For example:

```
#pragma omp task
void matmul( float *A, float *B, float *C );
...
#pragma omp target device(cell) implements(matmul)
void matmul_cell( float *A, float *B, float *C ) {
... // optimized version for target device
}
```

or directly in the header of a routine in an optimized library:


```
#pragma omp task
void matmul( float *A, float *B, float *C );
...
#pragma omp target device(cell) implements(matmul)
void matmul_spe( float *A, float *B, float *C );
```

The programmer is specifying that the alternative function (`matmul_cell` or `matmul_spe`) should be used instead of function `matmul` when offloading the task to one of the Cell SPE units. If the device `cell` is not available, then the runtime will launch the execution of the original `matmul` function on the default `smp` device. Different names are used for the different implementations in order to avoid duplicated symbols.

If the original implementation is appropriate for one of the accelerator types, then the programmer should precede the definition of the task with the specification of the target device

```
#pragma omp target device(smp,cell)
#pragma omp task
void matmul ( float *A, float *B, float *C ) {
... // original sequential code
}
```

In this case, the compiler would generate two versions for the same function, one going through the native optimizer for the default device and another going through the accelerator-specific compiler.

2.3 A Couple of Examples

Figure 6 shows the same matrix multiplication example used in section 1. The programmer specifies in the code example that the task could be offloaded into one of the Cell SPEs. The code to be offloaded should be generated by the native compiler for Cell using the function definition in `matmul`. Note that the `inout` clause [13] is used in the definition of the task to express the data dependence that exists among tasks computing the same block of `C`.

Several target accelerator devices can be specified in the application. For example in Figure 7 the programmer is specifying three possible options to execute function `matmul`. The first one is using the original definition of function `matmul` for the default target architecture. Two alternatives specified by the user are the implementation specified in `matmul_cuda` for an Nvidia GPU or the library implementation named `matmul_spe` for the IBM Cell. For all the devices, the runtime is in charge of moving data before and after the execution of the task.

2.4 Changing Memory Association for Data

In some kind of accelerators, as for instance in the Cell SPUs or in GPUs, it may be necessary to change the memory association for the data in memory prior to the execution on the device. Our proposal based on alternative implementations

```

1 #pragma omp target device(smp,cell) copy_in(A[BS][BS], B[BS][BS], C[BS][BS])
2                               copy_out(C[BS][BS])
3 #pragma omp task inout(C[BS][BS])
4 void matmul( float *A, float *B, float *C ) {
5     // original sequential code in Figure 1
6 }
7
8 float *A[NB][NB], *B[NB][NB], *C[NB][NB];
9
10 int main( void ){
11     for (int i = 0; i < NB; i++)
12         for (int j = 0; j < NB; j++)
13             for (int k = 0; k < NB; k++)
14                 matmul ( A[i][k], B[k][j], C[i][j] );
15 }

```

Fig. 6. Example specifying the execution on a device

```

1 #pragma omp task inout(C[BS][BS])
2 void matmul( float *A, float *B, float *C) {
3     // original sequential code in Figure 1
4 }
5
6 #pragma omp target device(cuda) implements(matmul)
7     copy_in(A[BS][BS], B[BS][BS], C[BS][BS]) copy_out(C[BS][BS])
8 void matmul_cuda ( float *A, float *B, float *C) {
9     // optimized kernel for cuda
10 }
11
12 #pragma omp target device(cell) implements(matmul)
13     copy_in(A[BS][BS], B[BS][BS], C[BS][BS]) copy_out(C[BS][BS])
14 void matmul_spe ( float *A, float *B, float *C);
15
16 float *A[NB][NB], *B[NB][NB], *C[NB][NB];
17
18 int main( void ){
19     for (int i = 0; i < NB; i++)
20         for (int j = 0; j < NB; j++)
21             for (int k = 0; k < NB; k++)
22                 matmul (A[i][k], B[k][j], C[i][j]);
23 }

```

Fig. 7. Example with the specification of alternative implementations for several target devices

could allow a runtime to change of memory association by specifying different headers for the implemented functions.

For example consider the code in Figure 8 that uses contiguous storage for matrices A, B and C. Notice that `matmul` defines the arguments as `[N][N]` and only accesses blocks of size `[BS][BS]` while `matmul_block` defines the arguments as `[BS][BS]`. The compiler can recognize this and instruct the runtime system to do the data movement to the local memory of the Cell SPE in a blocked way.

The extensions proposed in this paper are orthogonal to other possible extensions to generate efficient code by a compiler (e.g., vectorization width, number of threads running on accelerators, code transformations, ...) could be necessary for the compiler to generate good code for the target device. Proposals commented in the next section address some of these issues.

```

1 #pragma omp task inout(C[BS][BS])
2 void matmul (float A[N][N], float B[N][N], float C[N][N] ) {
3   // original sequential code in Figure 1
4 }
5
6 #pragma omp target device(cell) implements(matmul) copy_in(A, B, C) copy_out(C)
7 void matmul_block( float A[BS][BS], float B[BS][BS], float C[BS][BS] ) {
8   // original sequential code in Figure 1
9 }
10
11 float A[N][N], B[N][N], C[N][N];
12
13 int main( void ){
14   for (int i = 0; i < N; i=i+BS)
15     for (int j = 0; j < N; j=j+BS)
16       for (int k = 0; k < N; k=k+BS)
17         matmul ( &A[i][k], &B[k][j], &C[i][j] );
18 }

```

Fig. 8. Example with the specification of an implementations with change of memory association

3 Related Work

In this section we focuss on two approaches more closely related to our proposal (HMPP [9] and PGI [11]). We summarize both proposals in terms of specification of code regions to be executed on accelerator devices and how/when they decide to offload the execution.

3.1 CAPS Hybrid Multi-core Parallel Programming (HMPP)

HMPP [9] is designed to simplify the use of accelerators while keeping the application code portable (a sequential binary version can be built using a traditional compiler). The HMPP approach is to declare, by means of directives, functions (named codelets) suitable for hardware acceleration and callsites to them. Codelets are pure functions (i.e. functions that always evaluate the same result value given the same argument value(s), and have no side effects and no I/O). The accelerator functions are written in the own accelerator language in a specific file while keeping the original computation in the main source; then the developer uses the accelerator specific provided tools (compiler, library, ...) to generate the function binary.

The general syntax of the HMPP pragmas

```
#pragma hmpp <label> <directive-type> [, <directive-parameter>]*
```

The main **directive-types** are **codelet** (allows the declaration of a function as a codelet) and **callsite** (allows the call of a codelet). **label** is a unique identifier for a couple (codelet, callsite). The **directive-parameters** also specify the accelerator target (e.g. **target=cuda:sse**), conditional execution of the codelets (e.g. **cond=expr("n==1024")**), their desired synchronous or asynchronous properties and the data transfers (**input**, **output** and **inout** followed by the name of a function argument). In order to support these data transfers, there are constraints on how the codelet arguments are specified (the argument coding rules

have to permit to compute the amount of data to transfer runtime). For example, for n-dimensional data structures, the argument is followed by a one-dimensional array argument whose n elements give the size of each dimension.

The **synchronize** directive types allow to wait for the termination of a codelet. Other directive types are for decoupling the data transfers from the computations. By preloading (**advanceload**) data and downloading the results (**delegatedstore**) whenever they are required in the main application, the programmer can optimize the use of the memory bandwidth. Programmer can use the **asynchronous** directive parameter to interlace data transfers and codelet execution or the **const** directive parameter to preload data only once.

At execution, the HMPP runtime takes care of discovering the attached accelerators and their availability. When a codelet is indicated to be run on an accelerator, if the device is available and if the shared library corresponding codelet is present, HMPP loads it just as a software plug-in. Otherwise the native version is run on the host CPU or in a worker thread. The use of dynamic linking in HMPP allows to add improved codelet versions or add codelets for new hardware accelerators, without recompiling the overall application source.

The HMPP approach is quite similar to the proposal in this paper, since it also annotates functions to be offloaded in the accelerators that are specified in the **target** clause. We think that our approach has a better potential to express multiple implementations of functions, it is better integrated in the OpenMP specification and makes programming easier by delegating intelligence to the runtime system.

3.2 PGI Directives and Intrinsic Functions

The directives and programming model defined by PGI [11] allow programmers to specify the regions of a host program to be targeted for offloading to an accelerator device (mainly GPUs), without the need to explicitly initialize the accelerator and manage data or program transfers between the host and accelerator. Rather, all of these details are implicit in the programming model and are managed by their accelerator compilers. The bulk of a user's program are executed on the host. Their current version does not support multiple accelerators of the same type or different types.

The proposed directives are used to: 1) delineate accelerator regions and 2) augment information available to the compiler for scheduling of loops. **#pragma acc region** defines the region of the program in which loops will be compiled into accelerator kernels. It accepts clauses to specify data that needs to be copied from the host memory to the accelerator memory (**copyin**) and result data that needs to be copied back (**copyout**). The **local** clause is used to declare that the data needs to be allocated in the accelerator memory. The programmer can use the **if(condition)** clause to instruct the compiler to generate two copies of the region, one copy to execute on the accelerator and one copy to execute on the host, and decide which one to execute based on the evaluation of **condition**. The accelerator loop mapping directive **#pragma acc for** applies to loops. It can describe what type of parallelism to use to execute the loop (**host [(width)]**),

`parallel [(width)]`, `seq [(width)]` and `vector [(width)]`). If more than one scheduling clause appears on the loop directive, the compiler will strip-mine the loop to get at least that many nested loops, applying one loop scheduling clause to each level. The pragma also allows to declare loop `private` and `cache` variables, arrays and subarrays.

In summary, the main differences with our proposal is that PGI is based on compiler technology to optimize the offloading of loops in accelerators. Also the data movement between the memories is managed by the compiler, not by the runtime system.

4 Conclusions and Future Work

This paper proposes an extension to the OpenMP 3.0 tasking model to reasonably integrate heterogeneity while preserving simplicity and portability. Our proposal allows the programmer to easily specify the target devices for the execution of a task as well as, for a subset of the tasks that can be expressed in OpenMP, alternative implementations of the task for different target devices. We have shown how with this proposal the programmer could extend a simple matrix multiply to specify the execution in an heterogenous environment.

An implementation of this proposal is currently undergoing. We are currently trying accommodate in our extension other proposals targeting streaming architectures, such as the one proposed in [14], in which tasks become stream filters and `copy_in` and `copy_out` clauses are used to indicate input and output streams. Finally, we are also investigating new pragmas to direct program transformation (for instance to specify loop blocking) and their interaction with OpenMP constructs and our proposed extensions.

Acknowledgments

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contracts no. TIN2007-60625 and CSD2007-00050), the European Commission in the context of the SARC project (contract no. 27648) and the HiPEAC Network of Excellence (contract no. IST-004408), and the MareIncognito project under the BSC-IBM collaboration agreement. The researchers at the Universidad Jaime I were supported by the Spanish Ministry of Science and Innovation/FEDER (contracts no. TIN2005-09037-C02-02 and TIN2008-06570-C04-01) and by the *Fundación Caixa-Castellón/Bancaixa* (contracts no. P1B-2007-19 and P1B-2007-32).

References

1. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.* 27(3), 1–15 (2008)

2. AMD Corporation. AMD Fusion
3. Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., et al.: The Design and Implementation of a First-Generation Cell Processor. In: IEEE International Solid-State Circuits Conference, ISSCC 2005 (2005)
4. NVIDIA corporation. NVIDIA CUDA Compute Unified Device Architecture Version 2.0 (2008)
5. RapidMind. RapidMind Multi-core Development Platform, <http://www.rapidmind.com/pdfs/RapidmindDatasheet.pdf>
6. Knight, T.J., Park, J.Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: Compilation for explicitly managed memory hierarchies. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2007)
7. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: a programming model for heterogeneous multi-core systems. In: ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, pp. 287–296 (2008)
8. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing (2006)
9. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A hybrid multi-core parallel programming environment. In: First Workshop on General Purpose Processing on Graphics Processing Units (October 2007)
10. O'brien, K., O'brien, K., Sura, Z., Chen, T., Zhang, T.: Supporting OpenMP on Cell. International Workshop on OpenMP 2007, International Journal of Parallel Programming 36(3), 289–311 (2008)
11. The Portland Group. PGI Fortran and C Accelerator Compilers and Programming Model Technology Preview
12. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In: Symposium on Principles and Practice of Parallel Programming (PPoPP 2009) (February 2009)
13. Duran, A., Pérez, J.M., Ayguadé, E., Badia, R.M., Labarta, J.: Extending the OpenMP tasking model to allow dependent tasks. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 111–122. Springer, Heidelberg (2008)
14. Martorell, X., Ramirez, A., Carpenter, P., Rodenas, D., Ayguade, E.: Streaming machine description and programming model. In: Proceedings of the 7th International Symposium on Systems, Architectures, Modeling and Simulation (SAMOS), pp. 107–116 (2007)