Self-Adaptive OmpSs Tasks in Heterogeneous Environments

Judit Planas
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
judit.planas@bsc.es

Rosa M. Badia
Barcelona Supercomputing Center
Artificial Intelligence Research Institute (IIIA)
Spanish National Research Council (CSIC)
rosa.m.badia@bsc.es

Eduard Ayguadé, Jesús Labarta
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
eduard.ayguade@bsc.es
jesus.labarta@bsc.es

***Abstract*—As new heterogeneous systems and hardware accelerators appear, high performance computers can reach a higher level of computational power. Nevertheless, this does not come for free: the more heterogeneity the system presents, the more complex becomes the programming task in terms of resource management.**

OmpSs is a task-based programming model and framework focused on the runtime exploitation of parallelism from annotated sequential applications. This paper presents a set of extensions to this framework: we show how the application programmer can expose different specialized versions of tasks (i.e. pieces of specific code targeted and optimized for a particular architecture) and how the system can choose between these versions at runtime to obtain the best performance achievable for the given application. From the results obtained in a multi-GPU system, we prove that our proposal gives flexibility to application's source code and can potentially increase application's performance.

***Keywords*-multi-gpu management; heterogeneous architectures; parallel programming models; scheduling techniques**

I. Introduction

Since the emergence of the current multi-core processors that encompass a few processors and with the promise of the future many-core processors with hundreds to thousands of cores, both academia and industry have reacted with proposals for extending current programming methodologies or with new programming models to support the concurrency of these devices. Between them, we can find extensions to already existing languages or new programming models, like CUDA [1], OpenCL [2] or OpenACC [3].

Heterogeneity and hierarchy of the many-core processors are basic aspects that have to be considered in these proposals. The forecast speaks about scalable multi-core architectures composed of streamlined processor cores and accelerators, with on-chip network and advanced power management technologies.

We can then expect that future many-core chips will have both heterogeneity (different type of processors, including accelerators) and hierarchy (for example, organized in clusters of cores, possibly with associated local memory). Programming models should be able to support this heterogeneity and hierarchy in such a way that the application is unaware of the underlying hardware and that can dynamically adapt to it.

The OmpSs programming model combines ideas from OpenMP [4] and StarSs [5]: it enhances OpenMP with support for irregular and asynchronous parallelism and heterogeneous architectures. It incorporates data-flow concepts that allow the compiler/runtime to automatically move data as necessary and perform different kinds of optimizations.

OmpSs is able to run applications on systems that combine symmetric multiprocessors (SMPs) and GPUs. Moreover, we can even use clusters of SMPs and/or GPUs transparently from the application point of view with OmpSs [6].

The contribution of this paper is the addition of a new scheduler to the OmpSs runtime. Our motivations are code performance at low-cost maintenance: as new architectures appear, new programming paradigms do too and applications may become obsolete in a relatively short amount of time. With this new feature of the OmpSs runtime, we offer the ability to join separate pieces of code (i.e. new task implementations) to the original application without having to modify it. We can then rewrite certain parts of the old application to improve performance, fit new architectures or even adjust to user needs and join these rewritten parts to the original code.

This new scheduler is able to choose the most appropriate task implementation at runtime each time a task must be run. As tasks are executed, the scheduler learns and keeps track of their behavior so that it can make accurate decisions in the immediate future of the application execution. The main targets of the scheduler are to make source code maintenance easier, give more flexibility towards portability in heterogeneous platforms and improve application's performance by means of resource (and thread) cooperation.

The paper is organized as follows: Section II explains the problem we want to address. Sections III and IV give an overview of OmpSs and explain the new presented scheduler respectively. An evaluation of how the runtime mechanisms are able to schedule task versions can be found in Section V. The related work with regard this paper proposal is presented in Section VI. Finally, Section VII concludes the paper and explains the planned future work.

II. Writing Efficient Code for Multiple Platforms

Generally, there is not a single piece of code that fits all the existing hardware architectures, and even if we find that code, it will not be the best (in terms of performance, energy consumption, ...) for all of them. Thus, it is not unusual to find different ways of implementing the same algorithm.

As an example, there are uncountable versions of the matrix multiplication algorithm. Figure 1 shows a simple implementation, with no optimizations that could run in several different architectures. However, this is not the optimal version for any of them. A few years ago, the IBM Cell/BE [7] architecture appeared. The user could run the code in Figure 1 in the SPE accelerators, but he would not get any benefit. Thus, by that time appeared several implementations of the matrix multiplication algorithm specifically targeted to the SPE architecture. The result was that the user had to rewrite an important portion of his code in order to take advantage of that powerful brand new accelerator.

As history repeats, we are currently experiencing the same effects with the emergence of general purpose GPUs (GPGPUs): a small amount of GPGPUs can give the same peak performance as a supercomputer, so it seems worth using them to make computational applications much faster. Nevertheless, in this case the problem is even worse than what happened with Cell/BE: GPUs cannot run normal CPU code. The only way is to write a specific version for the GPU architecture, something not trivial nowadays.

In short, with the emergence of new architectures that provide more performance to applications, code improvement and maintenance is getting more complicated and more expensive. We present in this paper a solution to alleviate this problem and we explain the details by giving several examples. The reader can find in Section III the description of the specific code related to OmpSs.

A. Motivating Example

In order to illustrate our proposal, we will show an example of a tiled matrix multiplication algorithm. The basic implementation of the computation with the appropriate OmpSs task directives is shown in Figure 1. The matrices are stored in tiles of $BS \times BS$ elements. The **input** and **inout** clauses express data dependencies between tasks and ensure that the computation over the different blocks is done in the correct order.

But, as this implementation is very simple and non-optimal we might want to try calling a GPU kernel instead. Figure 2 shows the additional code that we would have to add to the original matrix multiplication code in order to use GPUs. The **device** clause specifies that the task should be run in a GPU architecture. The **copy_deps** clause indicates that the data specified by the dependence clauses should be readily available in the GPU memory before the task starts its execution. In this case, we directly call the CUBLAS library (instead of calling a GPU kernel) for simplicity.

```
#pragma omp target device (smp) copy_deps
#pragma omp task inout([BS*BS]C) input([BS*BS]A, [BS*BS]B)
void matmul_tile (float *A, float *B, float *C , int BS)
{
  int i, j, k;
  for (i = 0; i < BS; i++)
    for (j = 0; j < BS; j++)
      for (k = 0; k < BS; k++) {
        C[i*BS+j] += A[i*BS+k] * B[k*BS+j];
      }
}

void matmul (int m, int l, int n, float **A, float **B,
    float **C, int BS)
{
  int i, j, k;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < l; k++) {
        matmul_tile(A[i*l+k], B[k*n+j], C[i*n+j], BS);
      }
}
```

Figure 1. A simple C implementation of Matrix Multiplication

```
#pragma omp target device(cuda) implements(matmul_tile) \
    copy_deps
#pragma omp task inout([BS*BS]C) input([BS*BS]A, [BS*BS]B)
void matmul_tile_cublas(float *A,float *B,float *C,int BS)
{
  cublasSgemm ('T','T',BS,BS,BS,1.0,A,BS,B,BS,1.0,C,BS);
}
```

Figure 2. OmpSs Matrix Multiply task calling CUBLAS

So, by adding just the piece of code in Figure 2, OmpSs with our proposed extension will evaluate the performance of the two implementations of *matmul_tile* task and choose the most suitable version for each task invocation at runtime. There is no hard limit on the number of task versions, so we could add as many as we want (i.e. CBLAS library on CPU, specific implementation for SPE, ...). Figure 3 shows another example of a CUDA implementation of *matmul_tile* task. This implementation configures and calls a hand-coded CUDA kernel called *gemm_kernel*.

III. The OmpSs programming model

The OmpSs programming model [8] covers different homogeneous and heterogeneous architectures used nowadays

```
#pragma omp target device(cuda) implements(matmul_tile) \
    copy_deps
#pragma omp task inout([BS*BS]C) input([BS*BS]A, [BS*BS]B)
void matmul_tile_cuda(float *A,float *B,float *C,int BS)
{
  dim3 block(16, 16);
  dim3 grid(NB/block.x, NB/block.y);

  gemm_kernel<<<grid, block>>> (A, B, C);
}
```

Figure 3. OmpSs Matrix Multiply task calling a hand-coded CUDA kernel

and is open to cover future systems designed with new raising architectures.

OmpSs was initially based on the OpenMP programming model with some modifications to its execution model: it uses a thread-pool execution model instead of the traditional OpenMP fork-join model. There is a master thread that starts the execution and several other threads that cooperate executing the work it creates from work sharing or task constructs. Therefore, there is no need for a **`parallel`** region. Nesting of constructs allows other threads to generate work as well.

OmpSs also differs from OpenMP for its memory model: it assumes that multiple physical address spaces may exist. So, shared data may reside in memory locations that are not directly accessible from all processing elements in the system. Therefore, all parallel code can only safely access private data and shared data that have been marked explicitly with OmpSs extended syntax. The runtime takes care of where data resides and manages data transfers as tasks consume or produce them. Data can be replicated on different memory spaces and coherency is transparently managed by the runtime.

In addition, OmpSs allows annotating function declarations or definitions with a task directive. In this case, any call to the function creates a new task that will execute the function body. The data environment of the task is captured from the function arguments. It integrates the StarSs dependence support [9] and allows annotating tasks with three clauses: **`input`**, **`output`**, **`inout`**. They allow expressing, respectively, that a given task depends on some data produced before, that will produce some data, or both. The syntax in the clause allows specifying scalars, arrays, pointers and pointed data. Data addresses and sizes do not need to be constant at compile time since they are computed at runtime. Also, the **`taskwait`** construct (used as a barrier after parallel code) is extended as well with the **`on`** clause, which allows the encountering task to block until some data is produced.

The **`target`** construct [10] supports heterogeneity and data motion and can be applied to either tasks or functions. It accepts the **`device`** clause to express heterogeneity. It is used to specify which devices should run the associated code (e.g., cell, gpu, smp, ...). It also accepts the **`copy_in`**, **`copy_out`** and **`copy_inout`** clauses that are used to specify the memory spaces where data must be available and where they are produced. Another accepted clause is the **`copy_deps`** clause which is used to specify that if the attached construct has any dependence clauses, then they will also have copy semantics (i.e., **`input`** will also be considered **`copy_in`**, **`output`** **`copy_out`** and **`inout`** **`copy_inout`**). The different copy clauses do not necessarily imply a copy before and after the execution of each task. This allows the runtime to take advantage of devices with access to the shared memory or implement different data caching and prefetching techniques without the user needing to modify his code. To make sure that data that have moved to a device is valid again in the host, SMP tasks must also use the copy clauses or appear after an OpenMP flush (either explicit or implicit). The **`taskwait`** construct has also been extended with the **`noflush`** clause which allows synchronizing tasks without flushing all the data on remote devices. Finally, the **`target`** construct also accepts the **`implements`** clause which is used to specify that the annotated task is an implementation of another task[1]. The information kept in this last clause will be used at execution time by our proposed scheduler to group different versions of the same task together and decide which version is run each time.

Thanks to the flexible design and implementation of OmpSs runtime, it is very easy to extend any of its features, like adding a new scheduler or even the support for a new architecture. We can add a new feature as a new plug-in and later on, when we run an application, we can decide which plug-ins should be enabled through configuration arguments or environment variables. Thus, it is very easy to run several times the same application using, for example, different schedulers since there is no need to recompile neither the OmpSs runtime nor the application; we just have to set the appropriate environment variables or configuration arguments just before each execution.

IV. Versioning Scheduler

This section is focused on the implementation details of our proposal. It is divided into two different parts: the first part talks about the syntax and compiler support needed and the second one explains the runtime implementation.

A. Syntax and Compiler Support

As described in Section II, tasks must be annotated in a certain way to make the runtime aware of all task implementations.

Between all the existing task implementations, only one will be the main implementation and the other implementations will be versions of it. This distinction is only a compiler issue and will not affect the runtime execution; from the runtime point of view, all task versions are treated equally.

Figure 4 shows an example of a source code with three task versions implementing the same task. The main version is called *main_impl* and the other two versions are *another_impl* and *yet_another_one*. Note that all of them receive exactly the same parameters and have the same inputs and outputs. The **`implements`** clause is only valid for functions annotated as tasks (it cannot be used in inlined tasks) and its argument always references the main implementation. It is not possible to create an implementation of

[1]Although OmpSs syntax accepts the **`implements`** clause, none of the previous OmpSs runtime schedulers take it into account: only the main implementation of each task will be run and the rest will be ignored.

```
#pragma omp target device (cuda) copy_deps
#pragma omp task input ([N]A) output ([N]B) inout ([N]C)
void main_impl (int N, float *A, float *B, float *C)
{
  // Task code
}
#pragma omp target device (cuda) implements (main_impl) \
    copy_deps
#pragma omp task input ([N]A) output ([N]B) inout ([N]C)
void another_impl (int N, float *A, float *B, float *C)
{
  // Task code
}
#pragma omp target device (cuda) implements (main_impl) \
    copy_deps
#pragma omp task input ([N]A) output ([N]B) inout ([N]C)
void yet_another_one (int N, float *A, float *B, float *C)
{
  // Task code
}
```

Figure 4. Example of different task versions

another implementation if the second one is not the main version (i.e. *yet_another_one* task cannot be an implementation of *another_impl* because it is not the main version). There is no **implements** clause in the annotation for the main version. There is no restriction about task's **device**: the main implementation does not need to be SMP-targeted, the programmer can give more than one implementation for each device or even the same implementation can be targeted to more than one device (provided that all devices specified in the **device** clause are able to run the code). For each set of task implementations, the compiler creates a structure with the necessary information for the runtime to identify the different task versions as a set of implementations for the same task. Basically, this structure contains a list of devices where the task can be executed and a pointer to the corresponding task function for each device.

B. Runtime implementation

We have implemented a new scheduling policy for OmpSs runtime: the versioning scheduler. It is based on the already existing dependency-aware scheduler that tries to follow task dependency chains in order to promote data locality and minimize data transfers in a fast and simple way.

The versioning scheduler keeps and updates several data structures during the whole application execution that collect information related to each set of task implementations. The information is divided into *TaskVersionSet*'s, as shown in Table I. A *TaskVersionSet* represents a set of task versions. Since data set size needed by each task at run time is taken into account, each set is divided into different groups, according to the amount of data needed (*DataSetSize*) by each task instance[2]. For each group of data set size, the information is kept per task implementation (*VersionId*).

[2]Calling a task may imply some data movements, so the scheduler takes into account data set sizes of tasks. Each task's parameter size is counted just once, even if it is an input/output parameter.

TaskVersionSet	DataSetSize	<VersionId, ExecTime, #Exec>
task1	2 MB	<task1-v1, 30ms, 200> <task1-v2, 18ms, 350> <task1-v3, 25ms, 230>
	3 MB	<task1-v1, 45ms, 80> <task1-v2, 25ms, 300> <task1-v3, 40ms, 120>
task2	5 MB	<task2-v1, 15ms, 40> <task2-v2, 20ms, 3>

Table I
TaskVersionSet DATA STRUCTURE

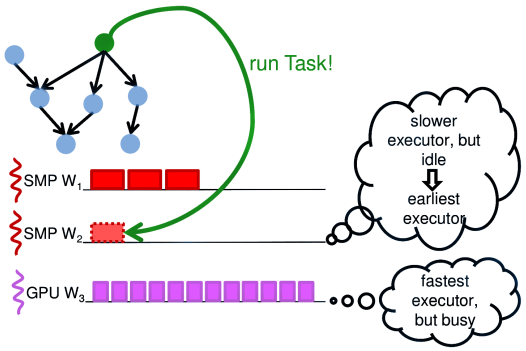

Figure 5. Scheduling decisions

The information associated to a task implementation is the number of executions *#Exec* and their mean execution time *ExecTime*. In Table I, we can see that there are two different task version sets, *task1* (with three different implementations) and *task2* (with two different implementations). In the case of *task1*, there are two different groups of data set sizes, because this type of task has been called with two different data set sizes. For each group, we can see that all the implementations have been run several times and their mean execution time has been recorded. Similarly, information for all the tasks implementations of *task2* has also been recorded, but in this case, there is only one group of data set sizes, because this type of task has always been called with the same data set size.

Each OmpSs worker thread is currently devoted to only one device (SMP, GPU, ...) and there can be as many workers as machine resources (cores, GPUs, ...). With the versioning scheduler, each worker has its own task queue. Each element of the queue is a pointer to a task that will be run by the corresponding thread. So, it will be used at runtime to assign tasks to threads and keep track of the amount of work each thread has in order to balance the task execution among all the threads.

Before explaining the scheduling policy, we need to give several definitions of scheduler-related concepts:

- *Mean execution time of a task version:* Each time a task is run, its execution time is recorded and its mean

execution time is updated as the arithmetic mean[3] of all the task executions. This value is used by the scheduler as the estimated execution time of that task version for future executions.

- *Fastest executor of a task:* For each group of data set sizes, this is the fastest task version. We will use this concept to refer to the OmpSs workers that can run such task versions.
- *Earliest executor of a task:* The OmpSs worker that can finish the execution of a task version at the earliest time. It will be usually the fastest executor, but in some cases, when the fastest executor is busy running other tasks, the earliest executor may be another OmpSs worker.
- *OmpSs worker estimated busy time:* Time estimation for an OmpSs worker to finish the execution of all the assigned task versions. It is computed as the addition of the estimated execution time for each task version in its queue.

We can divide application's execution into two different phases from the scheduler's point of view: the initial learning phase and the reliable information phase respectively. We consider that the initial learning phase finishes when the scheduler has enough reliable information about the execution of task version sets.

The initial learning phase consists of picking task versions from ready tasks in a Round-Robin fashion and distributing them among OmpSs workers. For each task version run, its execution time is recorded and thus data structures of Table I are filled with this information. We force the scheduler to run each task version at least λ times during the initial learning phase[4]. Once all tasks versions belonging to the same group of data set sizes have been run at least λ times, we consider that the scheduler has enough reliable information and it switches to the reliable information phase[5] for the given group of data set size. This means that the scheduler can have different criteria for the ready tasks that picks from the task graph, depending whether their corresponding group of data set size has enough reliable information or not.

During the reliable information phase, the scheduler tries to assign each task version to its earliest executor (taking into account task's data set size). To make this decision, it takes into account who is the fastest executor of the corresponding group of data set size, but also how busy is each worker (by checking each worker's task list). We represent in Figure 5 an example of this situation: for simplicity, we assume that all tasks in the task graph belong to the same task version set. Each rectangle represents a task assigned to a worker and its width represents the mean execution time of the task version. The scheduler picks the ready task colored as green and decides which worker should run this task. As we can see, the GPU worker 3 is the fastest executor of that task (GPU version runs faster than SMP version), but it is busy because it has many tasks in its task list, so the current task would have to wait until all the previous tasks are finished to be run[6]. Although the SMP version is slower, we can see that SMP worker thread 2 is idle and, in fact, can finish the execution of the current task before the GPU worker thread 3 has run all the tasks in its queue. The scheduler will then assign the current task to SMP worker 2 because it is the earliest executor.

In this phase, execution information is also recorded exactly in the same way as the previous phase: for each task version, its execution time is computed and its corresponding mean execution time is updated accordingly, so, somehow, the scheduler is always learning and recording execution information. This makes the scheduler more flexible and easily adapts to application's behavior, even if it changes over the whole execution.

When the data set size of a ready task differs from what the scheduler has registered in its previous executions, it is considered like a new group of data set sizes and it switches back to the initial learning phase behavior until it has again enough reliable information to move forward to the next phase (the reliable information phase).

The learning costs of our proposed scheduler are very application-dependent. Although the scheduler never stops learning (because it is always updating *TaskVersionSet* structures), we could say that there is an initial learning phase while one or more of task versions have not run several times. The cost of this initial learning phase is still application-dependent. For example, if one of the task versions is much slower than the others, the impact of the learning cost will be higher. In the same way, a short run with a few task instances -less than 10- will be highly affected by the learning costs (applications with 50-100 or more task instances have low learning costs).

V. Evaluation

The following sections cover the evaluation work that we have done in order to measure the performance of the presented OmpSs scheduler while running OmpSs applications using SMP and GPU specialized kernels.

A. Methodology

In order to evaluate our environment we selected a set of OmpSs applications and measured their scalability and performance with our scheduler, comparing it to the other schedulers of the runtime environment.

[3]Optionally, we could try computing a weighted mean to give more weight to recent execution information and less weight to past information, but we have not tried this option yet.

[4]This threshold can be configured by the user.

[5]Changing from one phase to the other just means that the scheduler changes the criteria to assign task versions to workers.

[6]This information is just an estimation based on the past history of each task version, but we can assume that a future execution of a task versions will behave similarly to a past execution of the same task version.

Additionally, and with the purpose of having a better understanding of the results, we also measured the amount of data transferred between host and device memory over the whole execution of the application. It may happen that the amount of data transferred is much bigger than the total size of application's data, because we may need to transfer data back and forth several times and all these memory transfers are taken into account. The amount of data is classified between three categories:

- Device Tx: when using two GPU devices, the amount of data transferred between these devices.
- Output Tx: the total amount of data transferred from all GPU memory spaces to the host memory space (main memory).
- Input Tx: the total amount of data transferred from the host memory space (main memory) to any of the GPU devices. If a piece of data is transferred to two different devices, both transfers are taken into account.

Finally, we also counted for the versioning scheduler the number of times each implementation was run. This gives us an idea of how SMP and GPU devices cooperate together with application's execution.

1) Environment: We used a node from MinoTauro cluster at the Barcelona Supercomputing Center. The system runs Linux and each node has two Intel Xeon E5649 6-Core at 2.53 GHz and two NVIDIA GPUs M2090 with 512 CUDA cores. The total amount of main memory for a node is 24 GB. Each GPU has 6 GB of memory.

All the codes were compiled with OmpSs compiler version 1.3.5.8 with optimization $-O3$ level. GCC version 4.4.4 and CUDA version 4.0 were used as back-end compilers for SMP and GPU codes respectively. OmpSs version is 0.7a.

2) Experiments: We run the selected applications with different configurations of numbers of cores and GPU devices to obtain the performance or execution time of each application.

For each application, we show the results of running the regular application (where each annotated task of the source code is targeting only one device) with the hybrid version of the application (where annotated tasks can have one or more implementations for different devices).

For the regular version of the application, we chose the fastest combinations of task implementations, and this is used to evaluate the quality of our presented scheduler.

For the hybrid version, we gave several SMP and GPU implementations of tasks and let the versioning scheduler choose at run time.

We used three different OmpSs schedulers to compare the results:

- Dependenciy-aware scheduler: In order to reduce the number of transfers between devices, this is a simple implementation of a scheduler that tries to find chains of dependencies and schedule consecutive tasks of the same chain to the same device. Its decisions are fast, but in some cases cannot fully exploit data locality.
- Affinity scheduler: This scheduler is a smarter implementation that tries to minimize the amount of transfers between devices. For each task, it evaluates the amount of data that should be transferred to a certain device in order to execute the task. The scheduler chooses the device where the minimum amount of data must be transferred. We can exploit data locality this way, and reduce significantly the time spent in memory transfers.
- Versioning scheduler: The scheduler we present in this paper, described in Section IV.

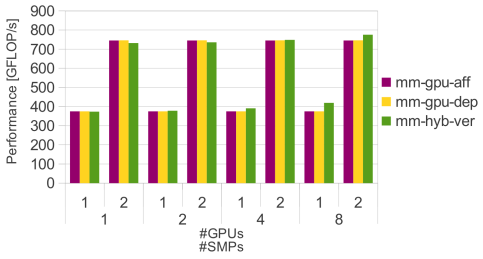

Figure 6. Matrix multiplication performance results

Since the dependency-aware and affinity schedulers do not support having more than one implementation for a task, we can only run hybrid applications with several task implementations using the versioning scheduler.

Even though OmpSs tries to minimize the amount of data transfers, they still represent a significant amount of execution time, so we configured OmpSs to overlap data transfers with task execution. We also combined this feature with prefetching task data to achieve higher performance. These features do not depend on the scheduling policy we are using, so we could enable them for all the three different scheduling policies.

B. Results

In this section we are going to present the evaluation of the versioning scheduler presented in this paper. We are going to compare the results of three applications (matrix multiplication, Cholesky factorization and PBPI) run with the three different scheduling policies mentioned before. We will also evaluate how the number of resources impacts application's performance.

1) Matrix Multiplication: The application performs a dense matrix multiplication of two square matrices. Each matrix is divided in tiles; each created task performs a matrix multiplication operation on a given block of the destination matrix: each product of a tile of the two input matrices is a task. We used three different kernels to do this computation: the CUBLAS kernel and a hand-coded CUDA implementation (both for a GPU architecture) and an SMP-targeted kernel calling the CBLAS library. The matrix size

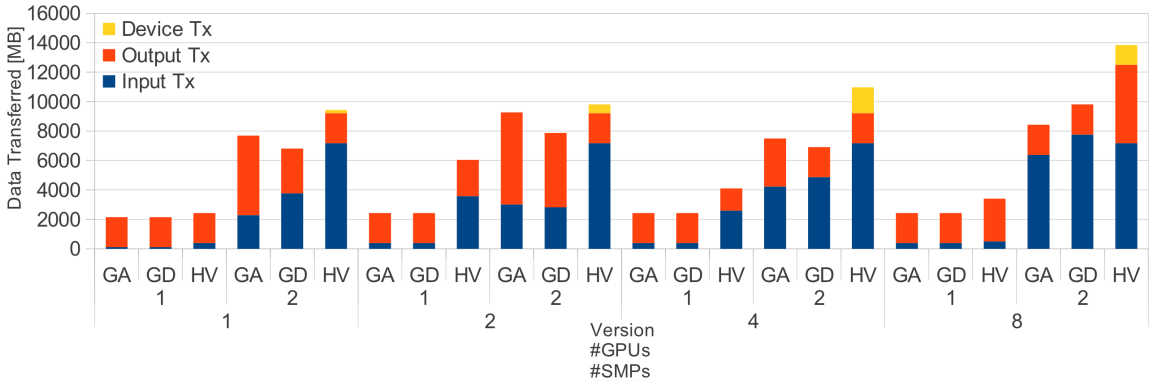

Figure 7. Data transferred for matrix multiplication

is 16384×16384 double-precision floating point elements (2 GB) and the tile size is 1024×1024 elements (8 MB).

We present the results of two different application versions we tested. We omitted an SMP-only version of matrix multiplication algorithm because its performance was too low to be comparable to the tested versions:

- mm-gpu: only the GPU version of a matrix multiplication task is given. The task calls the *cublasDgemm* function from CUBLAS library.
- mm-hyb: hybrid application with three different implementations for the matrix multiplication task are given: the main implementation is the same as the one given in the *mm-gpu*. One of the other two implementations runs on the GPU, too, and calls a hand-coded CUDA kernel that performs the multiplication. The last implementation is an SMP-targeted task that calls the CBLAS library to compute the result.

Figure 6 shows the performance results of the two tested versions of matrix multiplication enabling the different schedulers. We can see that for the *mm-gpu* version there is no difference between using the affinity scheduler (*mm-gpu-aff*) or the dependency-aware scheduler (*mm-gpu-dep*). In addition, the application shows the lineal scalability when using one or two GPUs. There is no difference between using one, two, four or eight SMP threads because there is no parallelism to exploit for the SMP threads.

The only OmpSs scheduler that exploits the hybrid version of the application is the versioning scheduler, so we can only present the results of running the matrix multiplication with this scheduler (*mm-hyb-ver*). For a small number of threads, we can see that the overall performance is slightly lower due to several reasons: firstly, the overheads of sharing data between different memory spaces and secondly, because the execution time of the SMP version of matrix multiplication tasks is much higher than the execution time of the GPU version (SMP task duration is about 60 times the GPU task duration). Nevertheless, the more SMP worker threads collaborate in the application execution, the more benefit versioning scheduler takes despite more data is transferred.

The performance benefit may look very small in this case, but we cannot expect a huge speed-up because the peak performance of eight SMP cores is still quite far from the performance of a single GPU: one SMP core represents less than 1% of the machine's peak performance and one GPU represents around 45% of the peak.

Figure 7 shows the amount of data transferred for each execution. *GA* represents the *mm-gpu* version run with the affinity scheduler, *GD* represents the same version run with the dependency-aware scheduler and *HV* represents the *mm-hyb* version run with the versioning scheduler. Because part of the computation is done on SMP devices and partial results are shared between CPU and GPU memory spaces, the amount of data transfers for the *mm-hyb-ver* increases. As we increase the number of SMP workers, we can see that memory transfers increase too, because more work is done by SMP workers and, thus, they need to share more data between SMP and GPU memory spaces. The versioning scheduler is also transferring data between GPU devices due to a lack of data locality.

Finally, we show in Figure 8 the number of times each version is run for the *mm-hyb* version. As we mentioned before, the application provides three different task versions: SMP version (that calls CBLAS library), CUDA version (that calls a hand-coded CUDA kernel) and CUBLAS version (that calls CUBLAS library). The fastest implementation (the CUBLAS version) is picked most of the times while the CUDA version is called only a few times at the beginning of the execution, because there is a faster implementation for the same device (its corresponding portion in the chart of Figure 8 is in the middle of each bar, but it is almost invisible). The SMP worker threads keep picking the SMP version while the GPUs are busy (except for the final part of the computation, where only the GPUs run the fastest implementation to avoid losing performance), and still take about 10% of the work on average that helps improving application's performance. We can notice that as we add

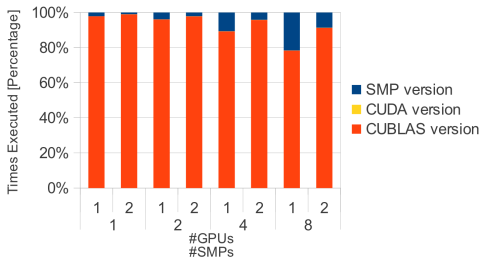

Figure 8. Matrix multiplication task statistics for versioning scheduler

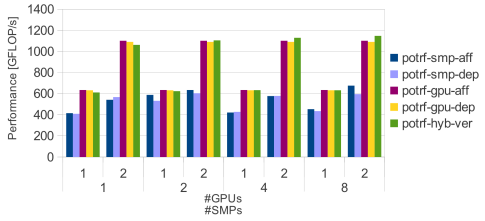

Figure 9. Cholesky factorization performance results

more SMP workers, more work is done by them. And for the same number of SMP worker threads, we can see that they do more work when there is only one GPU, because the GPU computation is slower and they have more time (and more chances) to pick tasks.

2) Cholesky Factorization: The Cholesky factorization is a matrix operation commonly used to solve normal equations in linear least square problems. It mainly calculates a triangular matrix (L) from a symmetric and positive definite matrix (A). The product of this triangular matrix L and its transposed copy is A: $Cholesky(A) = L$, where $L \cdot L^t = A$.

The source code is the main algorithm of a tiled Cholesky factorization. The matrix A is organized in blocks of 2048×2048 single-precision floating point elements (4 GB), with a total of 32768×32768 elements (16 MB). The annotated application primitives (tasks) operate on these blocks. There are four annotated tasks: *potrf*, *syrk*, *gemm* and *trsm*. For the last three tasks we give a single GPU-targeted implementation that calls MAGMA [11] or CUBLAS libraries. For the *potrf*, we give two different implementations: one calls CBLAS and runs on the CPU and the other one calls MAGMA and runs on the GPU.

In order to get good performance in this application, it is important to schedule carefully the execution of *potrf*, because in Cholesky's task graph, there are some points where all the following tasks depend on the *potrf* task. So, it acts like a bottleneck and if it is not run as soon as its data dependencies are satisfied, there is less parallelism to exploit and, thus, we observe a slowdown in application's performance.

The different application versions we used are described below:

- potrf-smp: only SMP-targeted implementation of *potrf* task is given. Although we add the "smp" suffix, the other three tasks are always run on the GPU, because running them on the CPU would take too much time for the amount of data they are computing.
- potrf-gpu: a single GPU-targeted implementation is given for each task.
- potrf-hyb: two different implementations (SMP and GPU versions) are given for the *potrf* task. The other three tasks are always run on the GPU.

We show the performance results for the three Cholesky versions in Figure 9. Running the *potrf* task in the SMP implies several data transfers from and to the GPU memory, plus the SMP version is slower than the GPU version. Thus, the *potrf-smp* is the version that gets less performance in all cases.

Although we can see the same performance trend for the *potrf-hyb-ver* version as for the matrix multiplication test (as we increase the number of SMP workers, the performance of the versioning scheduler gets better than the other tested versions and schedulers), the situation is different here. In this case, there is a small number of task instances, so the initial learning phase of the versioning scheduler impacts on application's performance. However, as we can see in Figure 10, having more SMP worker threads benefits performance for two reasons: the initial learning phase takes less time and a smaller amount of data is transferred compared to the other schedulers. In this case, the affinity scheduler cannot exploit data locality because there is load unbalance, so there is one GPU that steals tasks from the other one and this increases the number of memory transfers. However, it still gets good performance because the data transfers can be overlapped with computation.

Figure 11 shows the percentage of times each task version has been run for the *potrf-hyb-ver* version. In contrast with the matrix multiplication case and due to Cholesky's data dependency graph complexity, there is not enough ahead scheduling to assign a slow SMP task version to an SMP worker thread. Then, the scheduler decides to assign all the work to the GPUs because they become the earliest executors.

3) PBPI: PBPI is a parallel implementation of a Bayesian phylogenetic inference method for DNA sequence data. This solution is based on the construction of phylogenetic trees from DNA or AA sequences using a Markov chain Monte Carlo (MCMC) sampling method. There are two factors that determine the computation time: the length of the Markov chains used later to approximate the probability of the phylogenetic trees and the time actually needed to evaluate the likelihood values at each generation. The data set size used for this application is 50000 elements (500 MB).

In the implementation used in this paper, three different tasks are defined for each of the three computational loops that account for the majority of the execution time of the program.

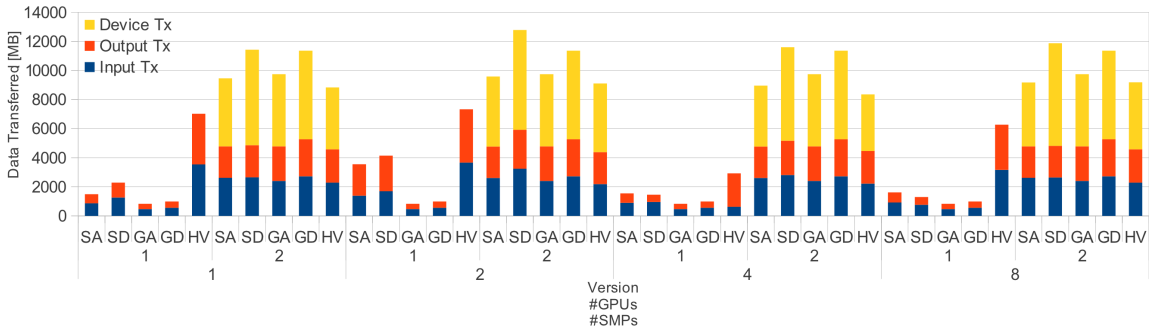

Figure 10. Data transferred for Cholesky

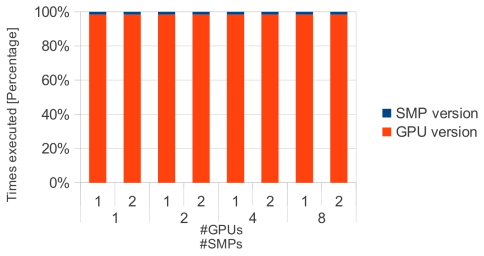

Figure 11. Cholesky task statistics for versioning scheduler

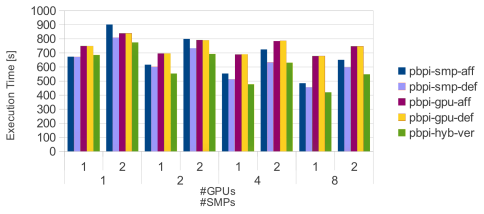

Figure 12. PBPI execution time results

In order to simplify the presentation of the results, we have provided up to two implementations for each of the first two computational loops. The third computational loop, although it is still a task, has a single SMP-targeted version. We present the results of three different application versions:

- pbpi-smp: only the SMP versions of each of the three tasks are given. In this case, data always stay in the host memory and no data transfers will be needed.
- pbpi-gpu: for the first two computational loops (that have been taskified), we give only the GPU version. The third computational loop is always run on SMP architecture.
- pbpi-hyb: we give two implementations for the first two computational loops: the first ones run on the GPU and the other ones run on an SMP device. The third computational loop is always run on SMP architecture.

Unfortunately, this application has no floating point operations, so we cannot give its results in GFLOP/s like we did with the other two applications presented in this paper; we have to report its total execution time instead[7].

Figure 12 shows the execution time of each version (remember that lower is better in this chart). We can see that *pbpi-smp* versions run faster than the *pbpi-gpu* versions. This is due to the fact that sending all the computational work of first and second loops to the GPU is not worth, since all the data will have to be transferred back and forth to run the third loop on the SMP workers and memory transfers cannot be overlapped properly due to data dependences.

However, the versioning scheduler is able to find the appropriate balance between SMP and GPU execution to take advantage and decrease the execution time. Although the amount of data transfers is higher, as shown in Figure 13, thanks to the look-ahead scheduling, it is able to overlap more data transfers with computation, so that we can see some benefit.

Figures 14 and 15 show the percentage of times each task version has been run for first and second loop respectively. For the first loop, the versioning scheduler decides to send it most of the times to the GPU, but the execution of tasks of the second loop is shared between GPU and SMP. Although it may not be clear in the chart (because hundreds of thousands of tasks are run for the second loop), the SMP version is run many times (thousands of times) and this helps balancing the trade-off between sending data back and forth and running the tasks on SMP workers (the task itself is between three and four times slower for the SMP versions, but the time of transferring all the data to the GPU is high enough to consider sending all the work to the SMP workers).

VI. Related Work

Heterogeneous architectures that combine different types of computational units (i.e., GPUs and traditional processors) are every time more common. However, the programmability of these nodes is an issue since the program-

[7]Although we could also report matrix multiplication and Cholesky results as their execution time to make all the charts homogeneous, we think that giving their results in GFLOP/s is better for the reader understanding, since this measure for such applications is widely used among the Computer Scientific community.

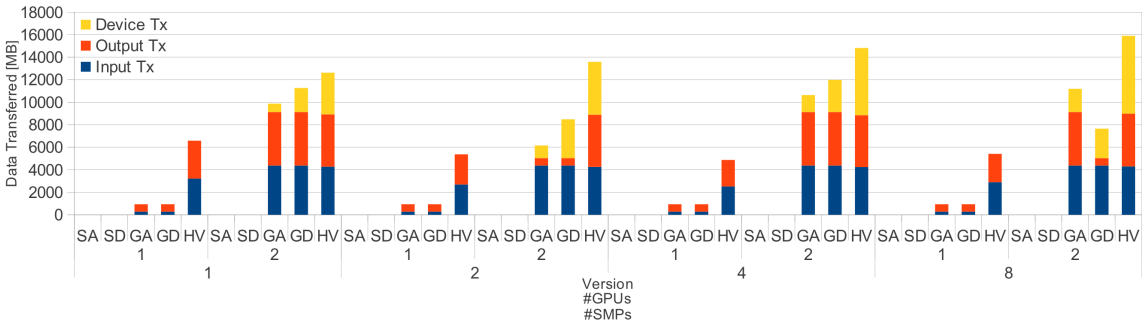

Figure 13. Data transferred for PBPI

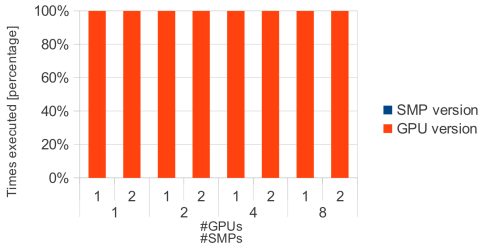

Figure 14. PBPI task statistics for versioning scheduler (first loop)

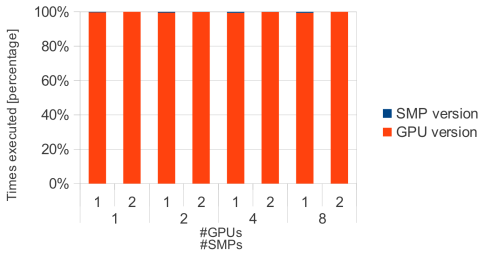

Figure 15. PBPI task statistics for versioning scheduler (second loop)

mer needs to take into account several aspects at a time: parallelism, different programming styles (i.e., traditional programming languages like C/C++ for the CPUs and CUDA for the GPUs), and the existence of different memory spaces and therefore the management of the data transfers between them.

With regard traditional node level parallel programming models, OpenMP is a widely adopted approach that focuses on shared memory systems. Designed for productivity, initially focused on loop parallelism. Recently has been extended with task based parallelism in its version 3.0 [4].

Cilk [12] is a task-based programming model based on the identification of tasks with the *spawn* keyword, and the *sync* statement is used to wait for spawned tasks. Both OpenMP and Cilk consider nested tasks (tasks that generate new tasks), but data dependency detection is not supported and additional synchronization points are required. Although Cilk only supported parallel tasks, Cilk++ also supports parallel loops.

NVIDIA GPUs had become very popular in the recent years and are extensively integrated in the HPC clusters. CUDA is almost the de-facto standard for programming GPUs. CUDA [1] is an extension to C++ and it is based on *kernels* that are executed concurrently by several threads. With CUDA, the programmer is not only responsible of writing the application code and computational kernels, but also of performing memory allocation and managing the data transfers from host memory to device memory. In [13] tools to better map the algorithms to the memory hierarchy have been proposed. In this approach, programmers provide straight-forward implementations of the application kernels using only global memory and the CUDA-lite tools do the transformations automatically to exploit local memories.

OpenCL has been proposed as an alternative to program accelerators that can also be used to program general purpose multi-cores [2]. Although the portability is a strong aspect of OpenCL, it offers a too low-level API to the programmer, exposing her to explicitly manage the data and threads.

In Lee et al. [14] propose OpenMPC, which is an OpenMP-to-CUDA translation system, which performs a source-to-source conversion of a standard OpenMP program to a CUDA program and applies various optimizations to achieve high performance. The compiler interprets OpenMP semantics under the CUDA programming model, identifies kernel regions, transforms them into CUDA kernel functions and inserts necessary memory transfer code to move data between CPU and GPU. Compared to OmpSs, OpenMPC focuses in loop parallelization, while OmpSs focuses on the exploitation of task-based concurrency.

HiCUDA [15] proposes the use of a set of directives that give hints to the compiler about regions of code that can be exploited in the GPUs, and data directionality. Mint [16] implements a translator that transforms stencil computations expressed in C, into CUDA code. Right now Mint does not support multi-GPU environments.

The CAPS HMPP [17] toolkit is a set of compiler directives, tools and software runtime that supports parallel programming in C and Fortran. HMPP works based on

codelets that define functions that will be run in a hardware accelerator. These codelets can either be hand–written for a specific architecture or be generated by some code generator. Offload [18] is a programming model for offloading portions of C++ applications to run on accelerators. Code to be offloaded is wrapped in an *offload* block, indicating that the code should be compiled for an accelerator, and executed asynchronously as a separate thread. Call graphs rooted at an offload block are automatically identified and compiled for the accelerator. Data movement between host and accelerator memories is also handled automatically. The Sequoia [19] alternative focuses on the mapping of the application kernels onto the appropriate engines to exploit the memory hierarchy. The PGI Accelerator Compilers [20] and the Cray OpenMP Accelerator compilers [21] provide support for NVIDIA GPUs. Both compiler systems recognize regions of code annotated with a special pragma, and they outline the code to be run on GPUs. Data directionality clauses are also incorporated in both approaches.

StarPU [22] is based on a tasking API and also on the integration of a data-management facility with a task execution engine. With regard to data management, StarPU proposes a high level library that automates data transfers throughout heterogeneous machines [23]. In StarPU, codelets are defined as an abstraction of a task that can be executed on a core or offloaded onto an accelerator. StarPU offers low level scheduling mechanisms so that scheduler programmers can use them in a high level fashion, regardless of the underlying (possibly heterogeneous) target architecture. OmpSs and StarPU present several similarities with regard the execution model, but StarPU is not proposing a programming model and therefore the programmer is exposed with lower APIs and execution details that are hidden in the OmpSs case.

In [24] the authors describe a runtime framework to schedule Direct Acyclic Graphs (DAGs) in heterogeneous parallel platforms. The proposal is based on four important criteria: Suitability, Locality, Availability and Criticality (SLAC) and show that all these criteria must be considered by a heterogeneous runtime framework in order to achieve good performance under varying application and platform characteristics. Again as StarPU, the authors propose a runtime scheduler while OmpSs is proposing a whole programming and execution model.

Qilin [25] aims at distributing kernel computations between CPUs and GPUs. The runtime component creates a directed acyclic dependency graph of kernels as the is being run. The runtime determines which kernels can be run in parallel and maps them dinamically to available processing units (either CPU or GPU). Qilin uses an analytical performance model to determine the execution time of individual kernels on specific accelerators, but while it can only exploit parallelism within a single basic block, OmpSs exploits parallelism between tasks (that can be build by several basic blocks).

VII. Conclusions and Future Work

In this work we have presented the implementation of a new feature for the OmpSs programming model. OmpSs runtime has been extended with the ability to run different implementations of a task, collect information about these task executions, learn how they perform and, finally, decide by itself which is the most appropriate implementation to run each time a task is called. This new feature has been presented as a new scheduling policy for the OmpSs framework: the versioning scheduler.

With this new scheduler, the programmer can write hybrid applications where more than one task implementation for one or more devices (SMP, GPU, ...) is given. This feature enhances the programmability of applications and makes its maintenance easier, because the programmer, at any time, can develop a new implementation for an already existent task in his code that targets the same or a different device and that can potentially improve application's performance.

From our results, we observe that, in most of the cases, the versioning scheduler outperforms the other existent schedulers for the OmpSs runtime and at the same time, gives more flexibility to the programmer. Only in a few cases the versioning scheduler slightly slows down the application compared to the other OmpSs schedulers.

Nevertheless, we have detected some weaknesses and limitations in our scheduler when using it in some specific applications. Firstly, the amount of data transfers is not optimal because data locality is not taken into account. We are going to provide the versioning scheduler with data locality information in order to further improve the performance of applications. Secondly, each record of a task version has to exactly match the data set size of its group. It is true that the execution time of a task can potentially depend on the size of the data that it computes or processes, but this implementation decision means that if the data needed by two calls to the same task varies from only 1 byte, the scheduler will consider that these calls belong to different groups of data set sizes and will not reuse the data collected at the first call when the task is called for the second time. In this case, it would be better to define the data sizes of each group in a reasonable range so that different calls to a task that process similar amounts of data would be joined together. With this optimization, the initial learning phase would take less time in some cases, so better decisions would be taken earlier. Finally, as a new feature, the scheduler should also offer the possibility to receive external hints for tasks versions: for example, read an XML file with additional information about tasks versions. This file can be written by the user, but it could also be written by OmpSs runtime from a previous application's execution.

Acknowledgment

This work has been supported by the European Commission through the ENCORE project (FP7-248647), the

TERAFLUX project (FP7-249013), the TEXT project (FP7-261580), the HiPEAC-3 Network of Excellence (FP7-ICT 287759), the Intel-BSC Exascale Lab collaboration project, the support of the Spanish Ministry of Education (CSD2007-00050 and FPU program), the projects of Computación de Altas Prestaciones V and VI (TIN2007-60625, TIN2012-34557) and the Generalitat de Catalunya (2009-SGR-980).

REFERENCES

[1] *NVIDIA CUDA Compute Unified Device Architecture Version 2.0*, NVIDIA Corporation, 2008.

[2] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[3] *The OpenACC standard*, http://www.openacc-standard.org.

[4] OpenMP ARB, "OpenMP Application Program Interface, v. 3.0," May 2008.

[5] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," *IEEE Int. Conference on Cluster Computing*, pp. 142–151, September 2008.

[6] J. Bueno-Hedo, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive programming of gpu clusters with ompss," in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2012, May 2012.

[7] C. Johns and D. Brokenshire, "Introduction to the Cell Broadband Engine Architecture," *IBM Journal of Research and Development*, vol. 51, pp. 503–519, September 2007.

[8] E. Ayguade, R. Badia, P. Bellens, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez-Gonzalez, J. Labarta, L. Martinell, X. Martorell, R. Mayo, J. Perez, J. Planas, and E. Quintana-Ortí, "Extending OpenMP to Survive the Heterogeneous Multi-core Era," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 440–459, June 2010.

[9] A. Duran, J. M. Pérez, E. Eduard Ayguadé, R. M. Badia, and J. Labarta, "Extending the OpenMP Tasking Model to Allow Dependent Tasks," in *OpenMP in a New Era of Parallelism*. Springer Berlin / Heidelberg, 2008, pp. 111–122.

[10] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Perez, and E. S. Quintana-Orti, "A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures," in *IWOMP: Evolving OpenMP in an Age of Extreme Parallelism*, vol. 5568. Dresden, Germany: Springer, June 2009, pp. 154–167.

[11] R. Nath, S. Tomov, and J. Dongarra, "An Improved MAGMA GEMM for Fermi GPUs," University of Tennessee Computer Science (also LAPACK Working Note 227), Tech. Rep. UT-CS-10-655, July 2010.

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.

[13] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu, "CUDA-lite: Reducing GPU Programming Complexity," in *In Languages and Compilers for Parallel Computing (LCPC) 21st Annual Workshop*, August 2008.

[14] S. Lee and R. Eigenmann, "Openmpc: Extended openmp programming and tuning for gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, 2010, pp. 1–11.

[15] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 78–90, 2011.

[16] D. Unat, X. Cai, and S. B. Baden, "Mint: Realizing cuda performance in 3d stencil methods with annotated c," in *Proceedings of the 25th ACM International Conference on Supercomputing*, ser. ICS '11, 2011, pp. 214–224.

[17] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-core Parallel Programming Environment," in *Workshop on General Processing Using GPUs*, 2006.

[18] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell, "Offload – automating code migration to heterogeneous multicore systems," in *Lecture Notes in Computer Science, HiPEAC Conference 2010*, 2010, pp. 307–321.

[19] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan, "Compilation for explicitly managed memory hierarchies," in *Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[20] Portland Group Inc., "PGI Accelerator Compilers," Sep 2011.

[21] Alistair Hart and Harvey Richardson and Alan Gray and Karthee Sivalingham, "Directive-based programming for GPUs, accelerators and HPC," December 2010. [Online]. Available: www.many-core.group.cam.ac.uk/ukgpucc2/talks/Hart.pdf

[22] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue*, 2010, accepted for publication, to appear.

[23] C. Augonnet and R. Namyst, "A unified runtime system for heterogeneous multicore architectures," in *Proceedings of the International Euro-Par Workshops 2008, HPPC'08*, ser. Lecture Notes in Computer Science, vol. 5415. Las Palmas de Gran Canaria, Spain: Springer, Aug. 2008, pp. 174–183.

[24] J. A. Pienaar, A. Raghunathan, and S. Chakradhar, "Mdr: performance model driven runtime for heterogeneous parallel platforms," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11, 2011, pp. 225–234.

[25] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 45–55.