2012 IEEE 26th International Parallel and Distributed Processing Symposium

Productive Programming of GPU Clusters with OmpSs

Javier Bueno, Judit Planas, Alejandro Duran
Barcelona Supercomputing Center
{*javier.bueno,judit.planas,alex.duran*}*@bsc.es*

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

Xavier Martorell, Eduard Ayguadé, Jesús Labarta
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
{*xavier.martorell,eduard.ayguade,jesus.labarta*}*@bsc.es*

***Abstract*—Clusters of GPUs are emerging as a new computational scenario. Programming them requires the use of hybrid models that increase the complexity of the applications, reducing the productivity of programmers.**

We present the implementation of OmpSs for clusters of GPUs, which supports asynchrony and heterogeneity for task parallelism. It is based on annotating a serial application with directives that are translated by the compiler. With it, the same program that runs sequentially in a node with a single GPU can run in parallel in multiple GPUs either local (single node) or remote (cluster of GPUs). Besides performing a task-based parallelization, the runtime system moves the data as needed between the different nodes and GPUs minimizing the impact of communication by using affinity scheduling, caching, and by overlapping communication with the computational task.

We show several applicactions programmed with OmpSs and their performance with multiple GPUs in a local node and in remote nodes. The results show good tradeoff between performance and effort from the programmer.

***Keywords*-Cluster programming; GPGPU computing; accelerators; OpenMP**

I. Introduction

In recent years the rise of GPUs for general purpose computing has been steady. The combination of large amounts of computing power and low energy consumption footprint have appealed many scientists to use them to build a new generation of supercomputers. Unfortunately, these advances have come at the expense of increasing the complexity of application development and maintenance.

Now clusters of GPUs are emerging as a new computational scenario. Programming applications for clusters has always been a tricky affair. Applications for clusters have traditionally been programmed with MPI. But, while MPI allows achieving very good performance it comes at the cost of programming at a very low-level which is error-prone and difficult to debug. Even more, as clusters become even larger obtaining high-performance requires using asynchronous communication and overlapping of computation which exacerbates the previous problems. The use of GPUs in a cluster introduces an additional layer of complexity to develop cluster applications.

Lately, there has been a significant effort to improve programming models to yield more productive models which still are able to provide good performance. But most distributed models, particularly for asynchronous task parallelism, do not integrate GPUs (or accelerators in general) well yet, and programmers need to resort to hybrid models (e.g., MPI+CUDA) managing manually all the details of distributing and moving the data between the GPUs of different nodes, execution of kernels and synchronization between the host and GPUs.

We designed OmpSs, which combines ideas from OpenMP[1] and StarSs[2], to try to tackle these problems. It enhances OpenMP with support for irregular and asynchronous parallelism and heterogeneous architectures. It incorporates data-flow concepts that allows the compiler/runtime to automatically move data as necessary and perform different kinds of optimizations. Our previous work has shown successful implementations of these ideas for multicore[2], the Cell B.E.[3], GPUs[4] and clusters of multicores[5].

The contributions of this work are the following:

- A hierarchical implementation of the OmpSs model that works for clusters of GPUs and multi-GPU environment. In this implementation, the runtime supports heterogeneity in the application tasks' (CPU and GPU tasks) and takes care of moving the data between the different levels of computation (i.e., from GPU to host, to the remote host, to the target GPU) as needed.
- A performance evaluation of OmpSs both for clusters of GPUs and multi-GPUs comparing OmpSs with MPI and CUDA. This evaluation includes also how different runtime decisions (e.g., scheduling, caching policy, etc) affect the overall performance in both environments.
- A simple productivity evaluation of OmpSs applications comparing them with MPI and CUDA applications.

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.58

II. OmpSs: multicores, GPUs, clusters

A. Overview

Our proposal is to have a single programming model, OmpSs, covering the different homogeneous and heterogeneous architectures in use today and open to future ones. OmpSs is based on the OpenMP programming model with modifications to its execution and memory model. It also provides some extensions for synchronization, data motion and heterogeneity support.

1) Execution model: OmpSs uses a thread-pool execution model instead of the traditional OpenMP fork-join model. The master thread starts the execution and all other threads cooperate executing the work it creates (whether it is from worksharing or task constructs). Therefore, there is no need for a **`parallel`** region. Nesting of constructs allows other threads to generate work as well.

2) Memory model: OmpSs assumes that multiple address spaces may exist. As such shared data may reside in memory locations that are not directly accessible from some of the computational resources. Therefore, all parallel code can only safely access private data and shared data which has been marked explicitly with our extended syntax. This assumption is true even for SMP machines as the implementation may reallocate shared data to improve memory accesses (e.g., NUMA).

3) Extensions:

Function tasks: OmpSs allows to annotate function declarations or definitions, *a la Cilk*[6], 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.

Dependency synchronization: OmpSs integrates the StarSs dependence support[7]. It allows annotating tasks[1] with three clauses: **`input`**, **`output`**, **`inout`**. They allow expressing, respectively, that a given task depends on some data produced before, which will produce some data, or both. The syntax in the clause allows specifying scalars, arrays, pointers and pointed data. The StarSs implementation from Pï¿½rez et al.[8] allows the data specified in the clause to partially overlap between them. We currently do not support this in our implementation.

Also, the **`taskwait`** construct is extended as well with the **`on`** clause, which allows the encountering task to block until some data is produced.

The target construct: To support heterogeneity and data motion we introduced a new construct: the **`target`** construct[9]. The **`target`** construct can be applied to either **`task`** or functions. Its syntax is the following:

```
#pragma omp target [clauses]
 task construct | function definition | function header
```

[1]We expect to also allow the annotation of OpenMP worksharings in the future, but they are not implemented yet.

```
void matmul( int m, int l, int n, int mDIM, int lDIM,
             int nDIM, float **A, float **B,
             float **C, int BS )
{
  int i, j, k;
  for ( i = 0;i < mDIM; i++)
     for (j = 0; j < nDIM; j++)
        for (k = 0; k < lDIM; k++)
           #pragma omp target device(cuda) copy_deps
           #pragma omp task inout([BS][BS] C[i*nDIM+j]) \
                 input([BS][BS] A[i*lDIM+j]) \
                 input([BS][BS] B[k*nDIM+j])
             cublasSgemm('T', 'T',BS, BS, BS, 1.0,
                         A[i*lDIM+k], BS,
                         B[k*nDIM+j], BS,
                         1.0, C[i*nDIM+j], BS);
}
```

Figure 1: OmpSs Matrix Multiply calling CUBLAS

Where the possible clauses are:

device	It specifies on which devices should run the associated code (e.g., cell, gpu, smp, ...). If no device is specified, then, it is assumed to be SMP.
copy_in	It specifies that some data must be accessible to the task when running. This may imply that it might need to be transferred between devices.
copy_out	It specifies that some data that was accessible to the task when running will be the only valid version when the task finishes its execution.
copy_inout	This clause is a combination of **`copy_in`** and **`copy_out`**.
copy_deps	It specifies 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 implementation to take advantage of devices with access to the shared memory or implement different caching and prefetch techniques without the user needing to modify his code. To make sure that data that has 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.

B. Examples

1) Matmul: The code in Figure1 implements a Matrix Multiplication using CUBLAS calls where the matrices are stored in tiles of $BSxBS$ elements. The **`input`** and **`inout`** clauses ensure that the computation over in the

```
#pragma omp target device(cuda) copy_deps
#pragma omp task input([N] a) output([N] c)
void copy(double *a, double *c, int N);

#pragma omp target device (cuda) copy_deps
#pragma omp task input([N] c) output([N] b)
void scale(double *b, double *c, double scalar, int N);

#pragma omp target device (cuda) copy_deps
#pragma omp task input([N] a, [N] b) output([N] c)
void add(double *a, double *b, double *c, int N)

#pragma omp target device(cuda) copy_deps
#pragma omp task input([N] b, [N] c) output([N] a)
void triad(double *a, double *b, double *c,
                      double scalar, int N);

void stream (int N, double a[N], double b[N], int BSIZE)
{
  scalar = 3.0;
  for (k=0; k<NTIMES; k++)
  {
     int j;
     for (j=0; j<N; j+= BSIZE)
        copy (&a[j], &c[j], BSIZE);
     for (j=0; j<N; j+=BSIZE)
        scale (&b[j], &c[j], scalar, BSIZE);
     for (j=0; j<N; j+=BSIZE)
        add (&a[j], &b[j], &c[j], BSIZE);
     for (j=0; j<N; j+=BSIZE)
        triad (&a[j], &b[j], &c[j], scalar, BSIZE);
  }
}
```

Figure 2: STREAM benchmark with OmpSs

```
void copy(double *a, double *c, int size);
{
  const int threadsPerBlock = 128;
  dim3 dimBlock;
  dimBlock.x = threadsPerBlock;
  dimBlock.y = dimBlock.z = 1;
  dim3 dimGrid;
  dimGrid.x = size/threadsPerBlock+1;

  copy_kernel<<<dimGrid,dimBlock>>>(size, 1, a, c);
}
```

Figure 3: Example of one of the CUDA tasks in STREAM

different blocks is done in the correct order. The **device** clause specifies that the task should be run in a thread attached to a GPU. 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.

2) STREAM: The code in Figure2 shows a possible implementation of the STREAM benchmark[10] with OmpSs. In this case, the *copy*, *scale*, *add* and *triad* function declarations are annotated as tasks. Each invocation to them will spawn a new task. The loops have been blocked so each task will execute $BSIZE$ iterations of the original loop.

Figure3 shows the definition of the *copy* function where a CUDA kernel is invoked. The generation of the kernels themselves is outside the scope of our research and must be provided by the user. Automatic generation of CUDA (or OpenCL) kernels is an active research topic and if successful it could be easily integrated into our model.

Note, that while the user is responsible to either write his own kernel code or use a library as in the Matrix Multiply example, OmpSs takes care of all data movement and kernel synchronization. Moreover, because these operations are not explicit the same application can be run in multiple GPUs (which can be either local or remote GPUs) and the runtime system can perform different kind of optimizations without these being hard-coded in the program.

III. The OmpSs implementation

Our implementation of OmpSs is built on two components: the Mercurium source-to-source compiler and the Nanos++ runtime library.

A. The Mercurium compiler

The compiler plays a relatively minor role on the implementation of the OmpSs model. On one side, the compiler recognizes the constructs and transforms them into calls to the Nanos++ runtime library. The data-flow clauses are transformed into a set of expressions. The evaluation of these expressions at execution time will generate addresses of memory that will be passed to the runtime library to build the task dependency graph.

On the other side, the compiler manages code restructuring for different target devices. When the compiler is about to generate the code for a **task** construct it looks if there is a matching **target** directive. If so, then the appropriate internal representation for the task is passed onto a device-specific ”handler” for each non-SMP device.

These ”handlers” generate the device-dependent data to be associated with the task. They also, if necessary, generate a specialized outline for the device which may need to be generated in a separate file. This additional file is reintroduced in the compiler pipeline usually following a different compilation profile that will invoke different backend tools (e.g., the Nvidia *nvcc* compiler for *cuda* devices).

The binary output for these different files are merged together into a single object file that contains additional information about the different subobjects. This allows the compiler to maintain the traditional behavior of generating one object file per source file to enable compatibility with other tools (i.e., makefiles). The information is recovered at the linkage step to generate the final binary with all the objects.

The code generated by the compiler is independent of whether it will run in a local or a cluster environment as this is managed transparently by the Nanos++ runtime library.

B. Nanos++ overview

Nanos++ is an extensible runtime library that supports OmpSs among other programming models. Its responsibility is to schedule and execute *parallel tasks* as specified by

the compiler based on the constraints specified by the user (order, coherence, ...).

Most of the runtime is independent from the actual target architectures supported (and more than one of these architectures can be active at the same time). Nanos++ currently supports the following "conceptual" architectures: *smp*, *smp-numa*, *gpu*[4], *tasksim* (a simulated architecture)[11] and cluster[5].

The following sections give an overview of the most important independent mechanisms of the library that serve as glue between the different architectures. Then we discuss the specific details of the *cluster* and *gpu* architectures.

C. Nanos++ Independent layer

The execution flow of a task inside the independent layer is the following: first, the task is added to the dependency task graph. When its dependencies are fulfilled it moves to the scheduler which will decide which resource it should run in. Before being executed, the coherence layer is invoked to ensure that all necessary data is available where the task is going to execute. Then, it is passed to the appropriate dependent layer that takes care of its execution. After, being executed the graph is notified to free new tasks based on its *output* dependencies.

1) Nanos++ Dependencies support: The runtime mantains a directed acyclic graph where tasks are connected following the dependencies specified by the user. Arcs are created for different kinds of dependencies: *read-after-write*, *write-after-read*, *write-after-write*.

The OmpSs model does not allow data dependencies outside the dynamic extend of a given task. This means that only sibling task will be connected together. This is particularly important as allows a hierarchical implementation of the graph for applications with multiple levels of task parallelism. This enables to easily distribute this information in a cluster environment.

2) Nanos++ task scheduler: Nanos++ allows changing the scheduler used for each execution. This allows us to experiment with different scheduling strategies. In this work we have used the following strategies:

breadth-first	Simple FIFO scheduling strategy.
dependencies	This is the same as the previous but before going to check in the queue it first tries to schedule a successor of the task that just finished. The idea behind this is that they will share data and it will end minimizing the number of data transfers.
locality-aware	This strategy is based in the work from Martinell et al.[12]. In this strategy, when a new task is submitted, the scheduler computes an affinity score for each location. This score is based on where each data specified by the task is located and also takes into account the size of that data (i.e., tries to prioritize big data). This score is used to place the task in the queue of the thread with the highest affinity. If there is no highest affinity, it is placed in a global queue. When threads request work they first look into their local queue, then into the global queue and last, they try to steal work from other threads to avoid load imbalance.

3) Nanos++ coherence support: Before a task is executed, the coherence support is invoked to ensure that an up-to-date copy of the data is available in the address space where the task is going to run.

A hierarchical directory keeps track of the physical location of data and of the most current version. In addition, a software *cache* exists for each device that has a separate address space. From this perspective a whole remote cluster node is a single device. But, GPUs inside that node will also have their own cache. So it works in a hierarchical fashion which provides better scaling. The *caches* keep track of which data is already in a different address space which allows to skip unnecessary data transfers. The *caches* can work in two different modes: write-through or write-back, being this last one the default policy.

The *cache* is prepared to support asynchronous operations, working as a *non-blocking cache*, if the underlying dependent layers support it. If supported, then the cache will not wait for data transfers to complete and will return control to the runtime so it can do more useful things (e.g., overlap user computation).

It is important to notice that the coherence mechanisms assume program correctness. Applications where the programmer provides tasks that write to the same data simultaneously without specifying proper synchronization (e.g., using the dependency clauses) result in undefined behavior.

D. Nanos++ dependent layer

1) The cluster architecture: The main difference of the cluster architecture with respect to other supported architectures is that, when running in a cluster, there will be more than one image of the runtime running at the same time (i.e., one on each cluster node). When the execution starts, the first image will become the *master* image and the remaining images will become the *slave* images.

All low level communications for control information and data transfers are implemented using *active messages*. We used GASNet [13] for this functionality since it offers a network-independent API with native support for various network technologies.

Initially there is only one task that is executed by the master. As this task starts creating new tasks, they will be scheduled to the local threads of the master node and send to a communication thread that represents the remote nodes. When a task is scheduled to the communication thread

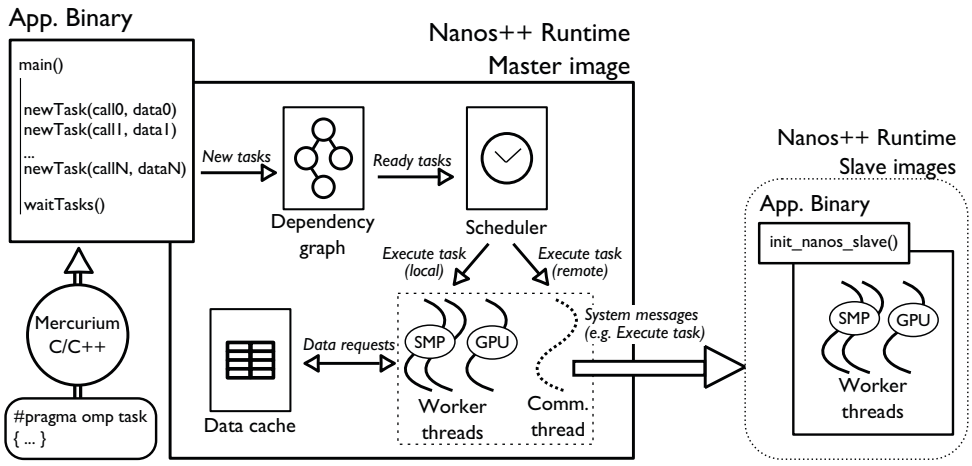

Figure 4: OmpSs overview for clusters of GPUs

it will be executed by a remote node. There is only one communication thread[2] that will be polling the task pool for each node of the cluster in a round-robin fashion.

The remote execution of tasks is a straightforward process. First, the general coherence mechanism of the runtime is invoked to ensure that all data that will be needed by a task is available in the remote node (and it is up-to-date). If not, data is gathered from its current location. If the data is available in the master node it is sent directly to the remote node. If it is only available in a remote node, then, a message is sent to that node so it sends the data to the new owner. This is done concurrently with the execution of other remote tasks to overlap communication and computation.

After this, the master sends a control message with the task information to start the execution of the remote task. The slave images are constantly waiting for upcoming requests. Once they receive a new request they will submit it to the local scheduler which will assign a local resource for its execution. When the task finishes another active message is sent back to the master to notify the completion of the task.

Tasks executed in a remote node can create new tasks that use the data transferred or created by their parent task. This allows scalable data decomposition to be coded in the application. These local tasks will be executed by any thread that becomes available in the node (and before going to fetch more work from the master node). Currently we do not implement stealing between the local queues of slave nodes.

The representant thread can be configured to presend tasks before the remote nodes are idle. With this technique the computation performed in remote nodes can be overlapped with the communication of the data that will be needed by tasks scheduled to be run on that node later on.

[2]Our design allows to have more than one if necessary.

2) The GPU architecture: The Nanos++ GPU layer works on top of the CUDA library from NVIDIA[14]. On startup it creates a GPU manager thread for each GPU in the system. These threads are in charge of transferring data from and to the GPUs, executing GPU tasks (that will launch GPU kernels) and synchronizing their execution.

GPU Memory Management: Both GPU memory and host pinned memory are allocated at startup, and then managed internally by the runtime. This avoids unnecessary calls to the CUDA runtime and also is needed in order to implement techniques to overlap data transfers and computation on the GPU.

Overlap of Data Transfers and GPU Computation: The runtime can transparently take advantage of the CUDA streams that allow overlapping data transfers with kernel computations.

In order to be able to overlap data transfers with computation, CUDA imposes several restrictions. The most relevant one is the fact that the host memory region that will be copied must reside in a page-locked memory. As we are not able to guarantee this requirement for user memory in general, we need to perform an internal copy to an intermediate buffer that is page-locked. We can allocate this intermediate buffer through the *cudaMallocHost()* function. We allocate a different intermediate buffer for each piece of data, and we deallocate it when it is not needed any more.

Data overlapping is disabled by default but can be requested via a config option when starting the execution. The overlapping mechanism requires extra memory operations which sometimes may not be worth enabling.

Data Prefetch: Once a GPU kernel is launched, the GPU threads request a new task to the scheduler. Then, we initiate the data transfer of any data that might be needed by the prefetched task. In this way, by the time this task can be executed the data will already be available. The prefetch is more effective when combined when the overlapping of

data transfers and computation as otherwise CUDA tends to serialize them after the kernel execution.

IV. Evaluation

This section covers the evaluation performed to measure the performance of Nanos++ when running OmpSs applications with GPU specific kernels.

A. Methodology

To evaluate our environment we selected a set of OmpSs applications and measured their scalability with our runtime environment. We have compared our approach with the same applications coded using CUDA and MPI+CUDA.

1) Environment: We have used two evaluation environments: a single node with multiple GPUs and a cluster where each node has one GPU.

The multi-GPU system was running CentOS 5.3 and it had two Intel Xeon E5440 with 4 cores each and 4 Tesla S2050 GPUs each with 2.62 GB of memory. The total amount of memory of the system was 15.66 GB of memory and a peak memory bandwidth of 148 GB/s.

In the GPU cluster environment, each node had two Intel Xeon E5620 processors with four cores each and one GTX 480 GPU with 1.5 GB of memory and peak performance in single precision 1.35 TFLOPS and a peak memory bandwidth of 177.4 GB/s. The total system memory of each node was 25 GB. The nodes were interconnected with a QDR Infiniband network with a bandwidth peak of 8 Gbits/s. OmpSs was compiled to use the native Infiniband conduit for GASNet.

All the codes where compiled with our Mercurium compiler with optimization $-O3$ level. GCC version 4.3.4 and CUDA version 3.2 were used as backend for the SMP and GPU parts respectively.

2) Experiments: We run the selected applications with different configurations of numbers of nodes and GPU devices to obtain the performance of each application. We also experimented with different configuration parameters in order of evaluating the impact of the different techniques implemented in the OmpSs runtime.

The OmpSs code run on the multi-GPU system was exactly the same as the one run on the GPU cluster, no modifications were made to handle the different architectures.

The applications selected were Matrix Multiplication, the STREAM benchmark, a Perlin Noise generator and a N-Body simulator.

The Matrix Multiplication used a matrix size of 12288x12288 single precision floats, the computation is performed using the CUBLAS sgemm call and it is divided in blocks of 1024x1024 floats each. For the MPI+CUDA version, we implemented a Summa algorithm as shown in [15] and it also used the CUBLAS kernel. While the MPI+CUDA implementation did not implement any techniques that could improve its performance, we believe it is representative to compare it in order to illustrate the difficulty of exploiting the whole potential of CUDA.

The STREAM benchmark implementations come from the original source code, the OmpSs version adapted to match the new pragmas and the MPI+CUDA based on the original MPI with added handmade kernels. The application allocated 768MB per GPU in each version.

Perlin noise is an image filter that generates noise to provide improved realism in computer generated images. We have used an image of 1024 x 1024 pixels. Two versions of this benchmark were developed, the first performs the computation and sends the data to the host memory after each computation step (we named this the *Flush* version), the second keeps the data on the GPU memory (named *NoFlush*). We believe these two approaches are interesting since sometimes perlin noise is applied as a sequence of image filters, thus not requiring this data movement back to main memory.

The N-Body simulation computes the gravitational interaction of a system of different bodies. The CUDA kernel comes from a set of NVIDIA examples. After each iteration of the system the data from the previous round must be distributed to all GPUs. We have simulated 10 iterations of a system with 20000 bodies.

B. Results

1) Multi-GPU environment: The multi-GPU experiments tested several configuration parameters of the runtime. The three cache policies available were tested: No-cache, which emulates the behavior of moving data in and out always, Write Through (shown as *wt* on the charts) which propagates writes to GPU memory to main memory, and Write Back (shown as *wb* on the charts) which delays the writing to main memory until de last moment. Also three scheduling policies were evaluated: breadth-first (*bf* in the charts), dependencies (*default* in the charts, as is the default scheduling policy of the runtime) and locality-aware (*affinity* in the charts).

In Figure 5 we can see the evaluation of the matrix multiplication. Reduction of data transfers impacts directly on the application's performance: when we avoid the use of any cache, we get the lowest performance, as data is moved back and forth each time a kernel needs it. Using a write-through policy improves the performance thanks to the data reuse, but writes still create a significant number of transfers that limit application's performance. Using the write-back policy helps to reduce this and obtains the best performance of all three policies. Nevertheless, the scheduling policy has also a big impact on performance as we increase the number of GPUs that we use: choosing a dependency-aware or locality-aware schedulers gives high performance benefits compared to the simple breadth-first scheduler, up to the point of almost doubling the performance in the case of

using 4 GPUs with a write-back-policy cache, where data locality is more critical.

The performance of STREAM benchmark can be found in Figure 6. Since the structure of the STREAM is pretty simple, every scheduler performs well enough. The key point of the STREAM is the memory management; no-cache and write-through moves data to main memory every time a task writes to the GPU memory, which overloads the runtime with useless memory transfers. Write-back handles better the situation and obtains a good performance.

The chart in Figure 7 represents the number of Mpixels/sec processed by the Perlin Noise algorithm. The application also stresses memory usage, similar to the STREAM benchmark, when we minimize the memory transfers we achieve a good performance. For the *Flush* version, the data movement is always done, thus we can not achieve as good performance as the *NoFlush* version.

The results shown in Figure 8 are singular and different from what we have observed in the other evaluated applications. The N-Body uses a lot of GPU memory which is also transfered between all the devices. This causes that the nocache policy outperforms the rest of policies, which fill the GPU memory and trigger the replacement mechanism and delay the writing to main memory. The no-cache policy avoids these situations which are more costly than just keep sending data back and keeping the GPU memory free. With this we still achieve a good scalability with 2 and 4 GPUs.

2) GPU cluster environment: For the GPU cluster evaluation, we have used the best parameters for the cache and GPUs, while we have studied other parameters that are available while running cluster applications.

Figure 9 shows the performance obtained with the matmul application with different configuration parameters. The X axis contain the different number of nodes used by each execution, for each number of nodes, there may be two groups depending on if the transfers between slave nodes was enabled or not (*MtoS* means that no Slave-to-Slave transfers were done, *StoS* otherwise). Also the last grouping describes the initialization type that was done before computing the matmul, *seq* means that all the data initialization was done sequentially on the master node, *smp* means that the initialization was parallelized using OmpSs tasks that were executed on the CPUs of the cluster, and *gpu* means that the initialization parallelized with OmpSs tasks and done by the GPUs of the system. Also, for each setup we present the performance achieved by using three different values of the task presend mechanism. The results show that Slave-to-Slave transfers are a must to achieve a proper scalability since they greatly reduce the data that must be moved around the network. Initializing the data in parallel also turns out to be a critical factor, since also reduces the amount of data transfers during the execution. SMP initialization provides in general better results than GPU initialization since moving data to remote nodes is more expensive when the data is available only on the GPUs. Presend also helps to improve scalability, specially when incrementing the number of nodes. Presend must be used along with Slave-to-Slave transfers in order to not overload the master with network operations that disrupt the execution of presends. In Figure 10 the performance of the best setup of OmpSs it is compared against the Matrix Multiplication implemented using MPI+CUDA. While the MPI obtains better performance with 1 and 2 nodes, the techniques implemented by our runtime outperform the MPI+CUDA version.

The STREAM benchmark results are shown in Figure 11. The application scales perfectly since there are no data transfers among the nodes of the cluster, thus it achieves a good performance using MPI+CUDA and OmpSs. No different options are shown, since when evaluated they did not present a significant impact due to the low network usage.

The Perlin Noise application shows the same issues as the ones presented on the previous subsection, we can see the results in Figure 12. Also, due to the nature of the communications -in the *Flush* version-, it cannot be overlapped easily with computation. Therefore, the presend and Slave-to-Slave transfers are not useful for this benchmark. The MPI+CUDA version also faces these issues and achieves the same performance as the OmpSs version.

For the Nbody application, Figure 13 shows the performance achieved on the cluster. Again we present the results for our best setup, however we did not observe significant differences by using the other cluster specific options. We believe that this is because the Nbody presents an all-to-all communication pattern that leaves almost no space to overlap communication and computation with the available options. However the scalability obtained by the OmpSs version is better than the one obtained by the MPI+CUDA, even though the OmpSs performs worse with 1 and 2 nodes.

V. Productivity Evaluation

Productivity is key for programmers to effectively exploit the resources available in current systems. Although evaluating productivity of a programming model is very complex, involving development times and effort spend in the work, in this section we provide a basic evaluation based on the number of lines that implement each version of the benchmarks.

After developing the benchmarks in CUDA, MPI+CUDA and OmpSs, we have counted the number of useful lines of code that result in each version, and computed the percentage of variation between them. Table I shows these results. For each benchmark and version, the number of lines is shown, and the percentage increase with respect the serial version is indicated in parenthesis.

As is can be observed, the common trend is that the CUDA version adds some lines of code, and the

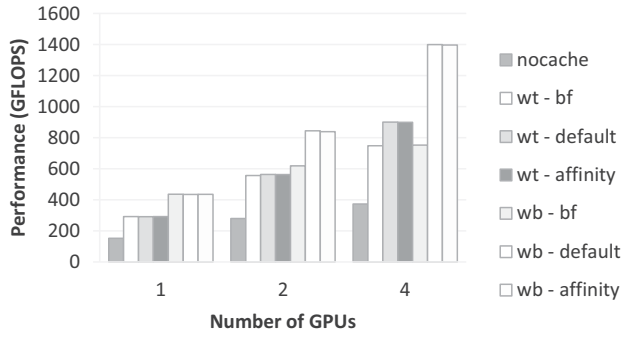

Figure 5: Matrix multiply performance results on the multi-GPU environment

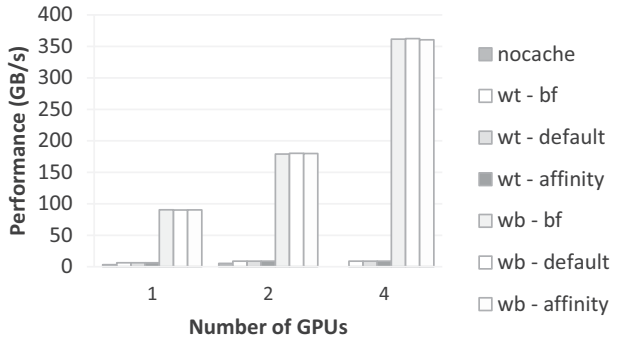

Figure 6: STREAM performance results on the multi-GPU environment

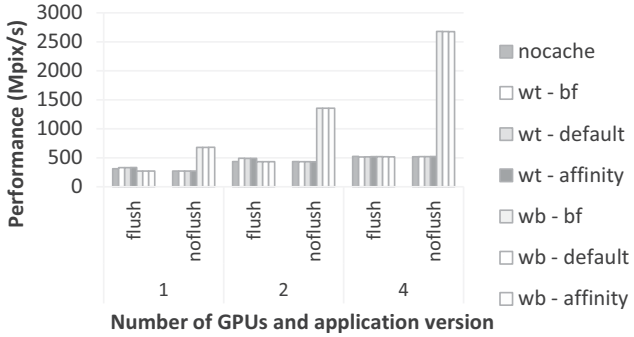

Figure 7: Perlin noise performance results on the multi-GPU environment

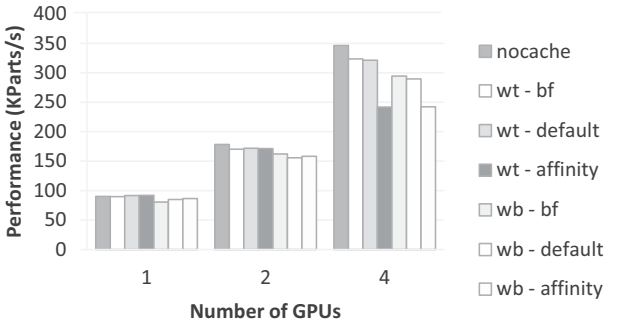

Figure 8: Nbody performance results on the multi-GPU environment

MPI+CUDA version even more. Instead, the increase in the number of lines is lower when writing the OmpSs+CUDA version. CUDA versions add lines of code to initialize the GPU device, allocate memory on it, copy the data between the host memory and the GPU, prepare the kernel parameters, and invoke the kernel. On top of this scheme, the MPI versions always add new lines to communicate messages between the nodes, and barriers when synchronization points are needed.

Instead, the increase when using OmpSs is due to the use of the compiler directives, and the fact that in our approach we still need to write the CUDA kernel invocation with the CUDA special syntax. The number of directives is usually limited to two per parallel region, to indicate to the compiler that, first, there is a task to be executed, and, second, the target architecture where it has to run. The argument calculation and kernel invocation are always around one third of the number of lines added in the CUDA version. In the future, we can further reduce the need of CUDA lines in our environment, using one of the proposals to generate the CUDA code automatically.

Benchmark	Serial	CUDA	MPI+CUDA	OmpSs+CUDA
Matmul	643	683(+6.2%)	696(+9.2%)	677(+5.2%)
STREAM	378	485(+28%)	496(+31%)	420(+11%)
Perlin	562	761(+35%)	788(+40%)	632(+12%)
Nbody	888	908(2.2%)	1049(+18%)	908(2.2%)

Table I: Comparison of total number of lines in Serial, CUDA, MPI+CUDA and OmpSs+CUDA versions of the benchmarks (in parenthesis, the percentage of increment, with respect to the Serial version)

VI. Related Work

Parallel programming models have been dominated by MPI [16], especially when programming distributed memory systems. It is based on the use of a library that implements explicit communication calls to transfer data between the set of processes running in the nodes of a cluster. Although MPI is not especially friendly to the programmer and requires most of times an overall re-writing of the application and an increase in the size of the code, it has been widely adopted by the scientific community with a very large number of applications currently written using this library. However, global synchronous communications are not affordable with current systems that have each time a larger and larger

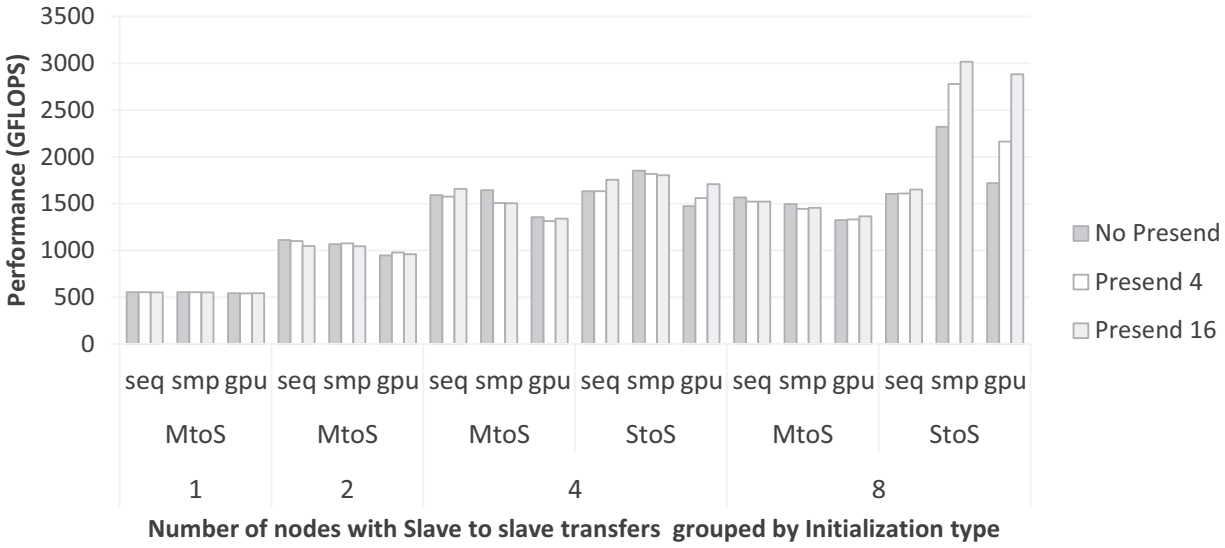

Figure 9: Matrix Multiply performance results on the GPU cluster environment

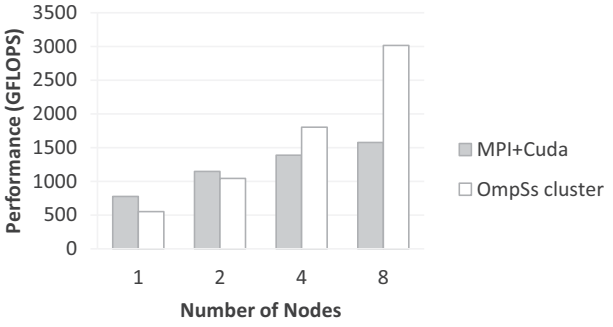

Figure 10: Matrix Multiplication: OmpSs vs MPI+CUDA performance

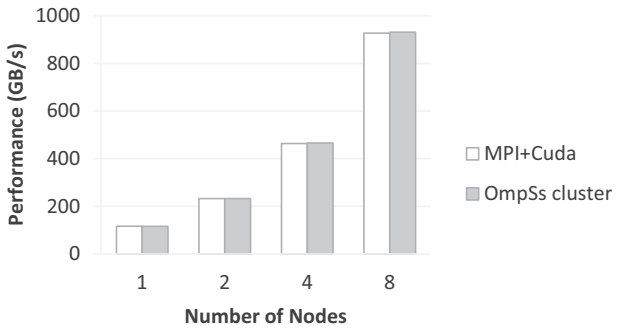

Figure 11: STREAM performance results on the GPU cluster environment

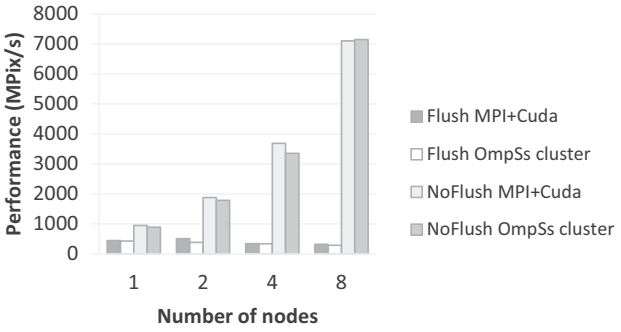

Figure 12: Perlin noise performance results on the GPU cluster environment

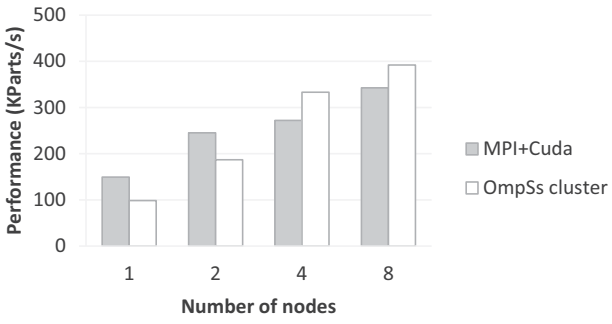

Figure 13: Nbody performance results on the GPU cluster environment

number of nodes since this become in most cases a source of imbalance.

OpenMP is another wide 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 [1].

Partitioned Global Address Space (PGAS) programming models expose an abstracted shared address space to the programmer simplifying its task, while data and thread locality awareness is kept to enhance performance. Representative PGAS languages are UPC [17], and X10 [18]; and Chapel [19], which implement Asynchronous PGAS model, offering asynchronous parallelism.

An alternative way to provide asynchronous parallelism on clusters is the one explored by Marjanovic et al.[20], a hybrid programming model that composes SMPSs, that inspired OmpSs, with MPI. The main idea is to encapsulate the communications in tasks so they are executed when the data is ready. This technique achieves an asynchronous dataflow execution of both communication and computation.

GPUs had irrupted in the recent years in the scene for HPC. Due to the popularity of NVIDIA GPUs, CUDA have almost become a de-facto standard for programming GPUs. CUDA [14] is an extension to C++ and it is based on *kernels* that are run n times in parallel by n threads. However, the programmer is not only responsible of writing the application code and computational kernels, but also of performing memory transfers from host memory to device memory. Tools to better map the algorithms to the memory hierarchy have been proposed [21]. They advocate that programmers should provide straight-forward implementations of the application kernels using only global memory and that tools like CUDA-lite will do the transformations automatically to exploit local memories.

An alternative to program accelerators that can also be used to program general purpose multicores is the new standard OpenCL [22]. Although the portability is a strong aspect of OpenCL, we find that OpenCL offers a too low-level API to the programmer, exposing her to explicitly manage the data and threads. For example, it is responsibility of the programmer to build the program executable or to move data between the cores and accelerators.

With regard the deployment of applications in clusters of GPUs, most of approaches follow a combination of MPI between nodes and CUDA inside the node [23], [24], [25]. There are very few approaches that consider alternative programming models for clusters of GPUs.

The cudaMPI and glMPI[26] provide a MPI-like message passing interface that enables to communicate data stored on the GPU cards of a cluster. While cudaMPI extends MPI to work with clusters of GPUs using CUDA, glMPI does the same for OpenGL. cudaMPI's programming model is CPU centric, in a way that CPU codes initiate each communication operation and the GPUs only provide the communicated data. This allows communication operations to be performed massively on large blocks of data.

An implementation of HPL benchmark for a heterogeneous cluster of CPUs and GPUs is presented by Fatica[27]. This implementation is based on a host library that intercepts the calls to DGEMM and DTRSM and is able to execute them simultaneously on both GPUs and CPU cores. The application parallelizes across the cluster nodes with MPI and inside the node, the matrix is divided in two: one part is processed by the CPU cores and the other by the GPU. To make the adequate split, they statically take into account the process time and the transfer time required to send the data to the GPU.

Lee et al. [28] propose OpenMPC. The proposal is based on 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. This system has been built on top of the Cetus compiler infrastructure. The compiler interprets OpenMP semantics under the CUDA programming model and identifies kernel regions (code sections to be executed on a GPU) and transforms eligible kernel regions into CUDA kernel functions and inserts necessary memory transfer code to move data between CPU and GPU. Compared with our approach, OpenMPC focuses in loop parallelization, while our approach exploits the concurrency according to the application tasks with data-dependencies. Another difference is that the approach does not consider clusters but a single node connected to a GPU.

With the objective of tackling GPUs, an extension to UPC[29] with hierarchical data distribution is presented. The approach extends the semantics of upc_forall to support multi-level work distribution. This work also presents features based on compiler analysis such as affinity-aware loop tiling and the implementation in the runtime of a unified data management on each UPC thread to optimize data transfers between CPU and GPU.

New computer architecture designs based on heterogeneous multicores have raised the question about their programmability. HiCUDA [30] proposes a set of directives giving hints to the compiler about regions of code that can be exploited in the GPUs, and data directionality. Mint [31] implements a translator to transform stencil computations expressed in C, into CUDA code, to run on a GPU. According to the publication, Mint extensions would be needed to handle a multi-GPU environment. The CAPS HMPP [32] 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 [33] 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 [34] alternative focuses on the mapping of the application kernels onto the appropriate engines to exploit the memory hierarchy. The PGI Accelerator Compilers [35] and the Cray OpenMP Accelerator compilers [36] 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. The latter proposals, HiCUDA, CAPS HMPP, Offload, Sequoia, and the PGI compilers do not support clusters. Instead, they target SMP nodes.

VII. Conclusions and Future work

In this work we have presented the implementation of the OmpSs model for clusters of GPUs. The model allows programmers to focus on the computation itself without having to deal with how to distribute the work nor its data in environments with GPUs. Furthermore, the same code will work with one or multiple GPUs and, independently if the GPUs are in the same node or in remote ones. To our knowledge this is the first attempt to give a productive programming model based on asynchronous task parallelism for clusters of GPUs.

This approach allows the underlying implementation to take care of the details and perform different optimizations based on the actual environment where the application is run. Our current implementation, based on the Mercurium and Nanos++ runtime, already does prefetching and overlapping of computation and communication at the cluster and GPU levels. It also comes with a locality-aware scheduler that tries to minimize the movement of data by allocating computation where its data is located. The evaluation of a set of benchmarks with this implementation shows that our current results are encouraging as in several scale on a small cluster of GPUs and get a performance similar to a multi-GPU node. For clusters of GPUs, the performance reported by our experiments is on par with implementations of the same benchmarks using the standard MPI and CUDA.

Even so, this preliminary evaluation also shows that the runtime implementation can be improved considerably. Aspects like better data distribution, or improvements in the scheduling policy could improve our results. The good thing is that applications do not need to be changed to benefit from this improvements, but just recompiled.

Further improvements that we envision to the model are better support of reduction operations, supporting non-contiguous memory regions, partial overlapping of data dependencies and the application of the dependencies clauses and target construct to worksharing constructs in addition to tasking. All these features would considerably increase the programmability of OmpSs.

Acknowledgments

We would like to thank the VU University from Amsterdam for the access to their DAS-4 system and the Universitat Jaume I from Castello for the access to their GPU machines. Also We thankfully acknowledge the support of the European Commission through the ENCORE project (FP7-248647), the TERAFLUX project (FP7-249013), the TEXT project (FP7-261580), and the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the support of the Spanish Ministry of Education (TIN2007-60625, CSD2007-00050 and FPU program) and the Generalitat de Catalunya (2009-SGR-980).

References

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

[2] 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.

[3] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta, "CellSs: Making it easier to program the Cell Broadband Engine processor," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 593–604, September 2007.

[4] "Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL," in *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*, October 2010.

[5] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, " Productive Cluster Programming with OmpSs ," in *Europar'11 (to appear)*, 2011.

[6] 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.

[7] 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.

[8] J. M. Perez, R. M. Badia, and J. Labarta, "Handling task dependencies under strided and aliased references," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 263–274.

[9] 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.

[10] J. J. Dongarra, I. High, and P. C. Systems, "Overview of the hpc challenge benchmark suite."

[11] "Trace-driven Simulation of Multithreaded Applications," in *Proceedings of the 2011 ISPASS (to appear)*, 2011.

[12] L. Martinell, ""Memory usage improvements for the SMPSs runtime"," Master's thesis, Computer Architecture Department, Universitat Politècnica de Catalunya, 2010.

[13] D. Bonachea, "GASNet Specification, v1.8 ," http://gasnet.cs.berkeley.edu/, U.C. Berkeley, Tech. Rep., 2006.

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

[15] R. A. V. D. Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," Tech. Rep., 1997.

[16] MPI Forum, "MPI: A Message Passing Interface Standard," *Intl. Journal of Supercomputer Applications and High Performance Computing*, vol. 8, no. 3/4, pp. 159–416, 1994.

[17] U. Consortium, "UPC Language Specifications v1.2," May 2005.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.

[19] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007.

[20] E. A. V. Marjanovic, J. Labarta and M. Valero, "Effective communication and computation overlap with hybrid mpi/smpss," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 337–338.

[21] 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.

[22] 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

[23] T. Hamada and K. Nitadori, "190 tflops astrophysical n-body simulation on a cluster of gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–9.

[24] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige, "Performance analysis of a hybrid mpi/cuda implementation of the naslu benchmark," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 23–29, March 2011.

[25] N. Karunadasa and D. D. N. Ranasinghe, "Accelerating high performance applications with CUDA and MPI," in *Proceedings of the Fourth International Conference on Industrial and Information Systems (ICIIS 2009)*, Sri Lanka, 28-31 December 2009.

[26] O. S. Lawlor, "Message passing for gpgpu clusters: Cudampi," in *Proceedings of the IEEE International Conference on Cluster Computing*, 2009, pp. 1–8.

[27] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 46–51.

[28] 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. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[29] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou, "Unified parallel c for gpu clusters: Language extensions and compiler implementation," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, K. Cooper, J. Mellor-Crummey, and V. Sarkar, Eds. Springer Berlin / Heidelberg, 2011, vol. 6548, pp. 151–165.

[30] 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.

[31] 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. New York, NY, USA: ACM, 2011, pp. 214–224.

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

[33] 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.

[34] 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.

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

[36] 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