OmpSs: Leveraging CUDA for productive programming in clusters of multi-GPU systems

BSC/UPC CUDA Center of Excellence
Barcelona Supercomputing Center (BSC-CNS) and Universitat Politècnica de Catalunya (UPC)

I. Living in the programming revolution

We are living the "real" parallel computing revolution. Something that was only a concern of a "few" forefront researchers and scientist has become mainstream and of concern to every single programmer. Multicore and accelerator based architectures now deliver dazzling performances at the expense of significant programming complexity. As with many revolutions, the fuzzy and shaky situation makes it difficult to predict what will be the stable (if any) structures that will remain after the transition. Many different architectures show up in the Top500 list and many more will probably be seen in the following years. When thinking of accelerator based systems, it is still unclear what will the prevailing system configurations be in the future, what mixes of devices and cores, memory hierarchies, and interconnection networks.

In this context we believe that it is time to decouple as neatly as possible the interface seen by the programmer from the detailed underlying hardware. The programming interface should provide an abstract model by which she/he can focus on the science or logic of interest. The interface may also provide ways to convey hints on potentials in terms of parallelism and/or locality exploitation, but not require strict statements of how resources have to be used. In the same way that we learnt to leave some responsibilities to the cache or virtual memory automatic policies, we need to learn to delegate resource management to intelligent runtime systems. Tasks are a key concept in this abstraction process, encapsulating self-contained computations of sufficient granularity. Handling dependences between tasks will become extremely important, especially at very large scale. This will enable the lookahead required to explore at runtime the future computation space of a program, avoiding stalls and enabling the detection of distant parallelism. Information on how data is used by the tasks is key to enable the system optimize the usage of memory.

The OmpSs programming model is an extension of the OpenMP directives in the direction explained in the previous vision statements. Data access directionality clauses (in, out, inout) for tasks provide the information required by the system to support dependencies and locality optimizations. OmpSs offers an elegant single source solution for programmers to achieve very good performance of a wide variety of system architectures. When targeting heterogeneous systems, several implementations can be provided by the programmer or used from libraries for low level tasks. The main logic of the program that orchestrates the chaining of those task is in itself a valid sequential program form which the system extracts the potential concurrency and data access or movements required. The system can dynamically choose the appropriate task implementations and schedules, adapting to the changes in application characteristics and resources availability and performance. The OmpSs model is being developed at BSC/UPC CCoE in Barcelona, with an implementation based on the Mercurium source–to–souce compiler and the NANOS++ runtime.

CUDA has proved an extremely successful approach. It is our believe that kernel based models offer very efficient implementation for particular types of tasks in extremely performant GPU devices. Allowing programmers to include CUDA code within OmpSs to define/reuse their very optimized kernels and their invocation, while leaving to OmpSs the responsibility of memory management, data movement, task scheduling, execution overlap with transfers and locality optimizations results in a very neat, productive and performing programming model. This was the target of the work here reported.

II. The visible part: leveraging CUDA

Our proposal is the integration of OmpSs and CUDA, leveraging CUDA with a minimalist set of directives that hide to the programmer the architectural details but provide precious information to the runtime system to do platform independent optimizations (parallelism discovery, locality-aware scheduling, ...) as well as architecture dependent optimizations (data movement between address spaces and overlapping, resource-aware scheduling, ...).

A. The task construct

The proposal is based on extending the **`task`** directive in two directions:

- Function tasks: OmpSs allows to annotate function declarations or definitions. 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.
- Argument directionality: OmpSs allows to annotate tasks with 3 clauses, **`input`**, **`output`** and **`inout`**, to express the directionality of task arguments: read, write and read/write, respectively. This information is used by the runtime to dynamically compute task dependencies.

B. The taskwait construct

The **`taskwait`** construct in OpenMP is extended as well in two directions:

- The **on** clause to allow the encountering task to block until the data specified in the clause is produced by previously generated child tasks.
- The **noflush** clause to avoid flushing all the data on remote devices.

C. The target construct

The **target** construct is added to support heterogeneity and data movement between address spaces. It can be applied to both **task** and functions:

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

Where the possible clauses are:

device — It specifies on which devices should run the associated code (e.g., **gpu**, **smp**, ...). if not specified the default is **smp**. When set to **gpu**, the programmer can include CUDA code, which is directly forwarded to the **nvcc** back-end compiler.

copy_in — It specifies that some data must be accessible to the task before running.

copy_out — It specifies that for some data the task will generate a new valid version.

copy_inout — This clause is a combination of **copy_in** and **copy_out**.

copy_deps — It specifies that if the associated **task** construct has any dependence clauses then they will also have copy semantics (i.e., **input** will also be considered **copy_in**, and so on).

implements — Specifies that a function implements a task for a specific **device**. This allows the possibility of specifying multiple implementations for the same task tailored to different devices; the runtime will decide among them based on availability of resources, location of data needed, ...

The different **copy** clauses are used by the runtime to decide on the best strategy to minimize the impact of data movement, so they do not necessarily imply a copy before and after the execution of each task. On the contrary, the runtime will take advantage of devices with access to the shared memory or implement different caching and prefetch techniques without changes in the source code.

D. Concurrency control

Two additional clauses for the **task** construct are provided. The **concurrent** clause specifies that dependences among tasks with this clause should not be considered. As it relaxes the synchronization between tasks the programmer must ensure that either the task has no dependences or that they are satisfied with a different synchronization mechanism. This is useful for example to handle reduction operations in tasks.

The **commutative** clause specifies that only one task in the group of commutative tasks can be run at a time; however, commutative tasks can be executed in any order.

III. The hidden intelligence: runtime system

Our proposal delegates to the runtime system the responsibility of schedule and execute the *tasks* in a dataflow way according to the directionality (and concurrency control) clauses specified by the programmer and the associated data management. 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).

The execution flow of a task inside the independent layer is the following: first, when instantiated the task is added to the dependency task graph. When its dependencies are fulfilled the task is ready and scheduled to be executed on the most appropriate computational resource. Before being executed, the coherence layer is invoked to ensure that all necessary data is available in that computational resource. Then, it is passed to the appropriate architecture dependent layer that takes care of its execution. When the task finishes its execution, the graph is notified to potentially free all those tasks that consume the results produced by the task.

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.

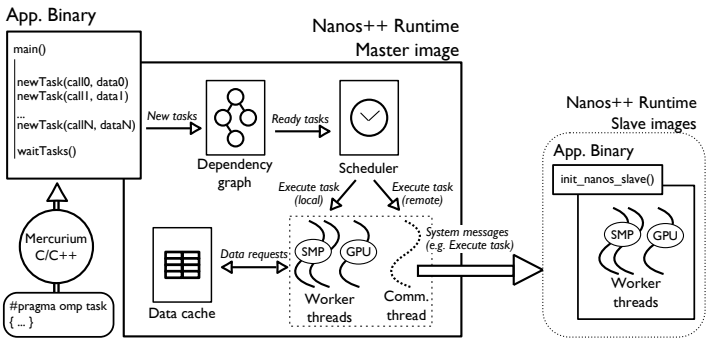

Fig. 1: OmpSs overview for clusters of GPUs

A. Independent layer

The most important independent mechanisms of the library that serve as glue between the different architectures are the following:

- Task dependencies support. The runtime maintains a directed acyclic graph where tasks are connected following the dependencies specified by the user. The OmpSs model does not allow data dependencies outside the dynamic extend of a given task. This means that only sibling tasks will be connected together. This is particularly important as allows a hierarchical implementation of the graph for applications with multiple levels of task parallelism.
- Task scheduler. Different scheduling strategies are available, from simpler ones that schedule the tasks in FIFO order, to more elaborated ones that take into account the data locality and its size when scheduling a task to a given thread.

- Coherence support. Before a task is executed, the coherence support ensures 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 locations of data objects and of the most current version. In the hierarchy, a node in a cluster is considered as a single device inside which multiple devices (e.g. GPUs) may exist. In addition, a software *cache* exists for each device that has a separate address space. The *caches* keep track of which data is already in a different address space which allows to skip unnecessary data transfers.

B. The GPU dependent layer

The Nanos++ GPU layer works on top of the CUDA library. 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 to overlap data transfers with kernel computations.
- Data Prefetch. Once a GPU kernel is launched, the GPU threads request the next task to the scheduler. Data transfer of any data that might be needed by the prefetched task is started at this point with the expectation that 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. An example: Nbody

The code in Fig.2 shows an excerpt of the source code for a N-body simulation. This code uses tiling to split the simulation in several tasks to maximize the potential parallelism when multple nodes with multiple GPUs are available. In this example the CUDA code is a custom kernel provided by the programmer, which appears in the code as a regular CUDA kernel call. The runtime handles data transfers between the CPU and the GPU inside each node and between nodes in the cluster, if available.

Figure 3 shows the performance achieved in two different systems: 1) a multi-GPU system with two quad-core Intel Xeon E5440 and 4 Tesla S2050 GPUs; and 2) a GPU cluster environment with 4 nodes, each with two quad-core Intel Xeon E5620 processors and one GTX 480 GPU, interconnected with a QDR Infiniband network. The same OmpSs code has been used in both architectures. More results are available in a paper to be published in IPDPS-2012 [1].

```
1  void calc_forces_cuda (
2       int n_particles,
3       Particle particle_array[n_particles],
4       Particle output_array[n_particles],
5       float time_interval) {
6
7    const int bs = number_of_particles / NBLOCKS;
8    size_t num_blocks;
9    int first, last, i;
10
11   for ( i = 0; i < n_particles; i += bs ) {
12     first = i;
13     last = i + bs - 1;
14     last = (last > n_particles) ? n_particles : last;
15     num_blocks = (last - first + THxBLOCK) / THxBLOCK;
16
17 #pragma omp target device(cuda) copy_deps
18 #pragma omp task output(output_array[first:last]) \
19                  input(particle_array[0:n_particles-1])
20     calc_forces_kernel <<< num_blocks, THxBLOCK >>>
21         (time_interval, particle_array, n_particles,
22          &output_array[first], first, last);
23     }
24   }
```

Fig. 2: N-Body simulation with OmpSs

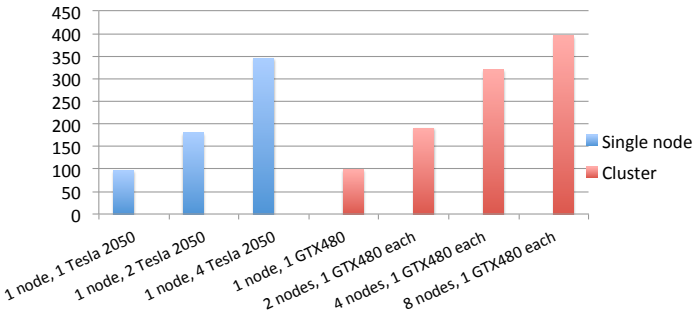

Fig. 3: Nbody performance in Kparticles/second

V. Conclusions

CUDA has proved an extremely successful approach. It is our believe that kernel based models offer very efficient implementation for particular types of tasks in extremely performant GPU devices. Allowing programmers to include CUDA code within OmpSs to define/reuse their very optimized kernels and their invocation, while leaving to OmpSs the responsibility of memory management, data movement, task scheduling, execution overlap with transfers and locality optimizations results in a very neat, productive and performing programming model.

OmpSs and related publications are available for download from the project website [2].

References

[1] Javier Bueno-Hedo, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguadé and Jesús Labarta. Productive Programming of GPU Clusters with OmpSs. In The 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2012), Shanghai (China), May 2012.

[2] Programming models group at BSC. http://pm.bsc.es.