Memory-Coherence between host and devices in a runtime

Ruben Cano, Daniel Jiménez, Vicenç Veltran
*Barcelona Supercomputing Center, Barcelona, Spain
†Universitat Politècnica de Catalunya, Barcelona, Spain
E-mail: [ruben.canodiaz, daniel.jimenez, vicenc.beltran]@bsc.es
Special thanks to Carlos Álvarez.

Keywords—Runtime, Nanos6, Main memory, Coherence, High-performance computing.

I. Extended Abstract

As the end of the Moore's law approaches, more specific devices such as GPUs, FPGAs or AI accelerators tend to steal the workload that was traditionally run on the CPU, allowing with this offload more specific solutions that improve the execution time of specific applications. One of the main problems that arise with this approach, is that now, the data is not centralized in one main memory, but distributed among the different accelerators which need a correct and coherent data to perform its operations. This can potentially limit the performance an accelerator can achieve, as well as delegates the programmer the task of enforcing the coherence between memories.

To relieve this model, in which the programmer has to take into account the memory of devices, models like NVIDIA Unified Memory[1] manage the hard work of maintaining the memory-coherence, potentially hurting performance but making the per-device memory management much easier.

In this work, the main objective is to develop an extension for a task-based runtime, which maintains the coherence between SMP and multiple devices in the system, using the dependency information of a task, acting as a Translation-Allocation Layer between the multiple memory spaces defined by the accelerators.

For our development, we use Nanos6 Runtime[2] as the target of our implementation. Nanos6 is a Runtime that implements the OmpSs-2 programming model, it is an SMP task-based runtime, with cluster support and is able to run CUDA tasks using unified memory. With this implementation, our objective is to extend Nanos6 capabilities to allow running CUDA with distributed memory, as well as other kind of accelerators such as FPGAs.

A. Memory abstraction for devices

To manage the different memories, each memory should provide a interface that allows to: copy from and into it, allocate memory and free memory, in our system, we will manage the memories using this abstraction, which will allow to map any kind of esoteric memory into the same system that will be shared by all the devices.

B. Task dependencies and symbols

A task dependency from a memory perspective, is a region of host memory that a task needs to coherently access, to write or read, in order to be executed, however, a symbol can be a superset of various dependency regions, and has the particularity that each symbol needs to be continuous in memory, this is due to the different already-compiled algorithms take into account the different offsets while accessing the data so we don't only has to have the data valid on the device, but has to be in the way the already-existing software expects them to be. This means that the dependency can be multiple parts of a symbol, but the whole symbol has to be in memory.

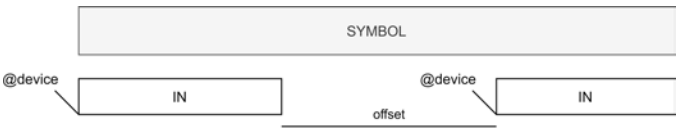

Fig. 1. In this figure we can see how, for a Symbol, we have only two dependencies that are not contiguous, however, we have to maintain the offset address value between the two dependencies because the algorithm could expect the data to be in a certain offset.

C. Non-blocking Software Memory Management Directory

The Directory is a range-based multipurpose software system that given multiples address spaces, manages the allocation, copies and translation from host memory to any other address space. So, for the runtime, it's only a translation layer where you give an address and a range, and it returns a list of copy operations that are needed to satisfy the access, and an address to the data in the given address space. When a task finishes using that region, that information is passed to the directory, and the access is released. However, due to the possibility of an already-existing region on an address-space being part of a symbol that needs more contiguous allocation, we can ensure that when the directory returns an address, that address will remain valid and accessible until the task finishes using it, but consequent tasks that may access that region, may have different addresses. So, the coherence of a returned address is not enforced, and a returned address shouldn't be used outside its task scope, and all accesses have to be checked by the Directory.

For explaining the functionality without going into too much detail, we will divide the Directory duties in three main steps:

1) Allocation Step: The directory checks if for a symbol region, there is an already-valid allocation on the address-space. If there is no allocations, it will try to allocate a paged-size chunk in the address-space. If there is a partial allocation, it will allocate a new paged-size sufficient to allocate all the symbol again, and generate the copies of the data to the new address, and, lastly, if there is no space for a new allocation, will return an error and the runtime will need to try again later when some data has been released.

2) Data Copies Generation Step: When an access has an already-valid allocation, we need to maintain the coherency between all the address-spaces. This means, for a range, we can have multiple entries on our directory, each one with it's own state. We use a MESI-Like protocol in each entry, which allows us to have read-valid data in more than one device, and invalidate if necessary the data in any address space.

This means that if the data is invalid in the given address-space, we will check where the last version of the data is, and copy from there to the address space, updating the state or invalidating if necessary. However, this is not that simple, since we are not synchronously getting the data, but generating the copy that will be performed asynchronously. For this, we create a "promise" of validity, put the entry status in a transitory state, and only changing the status to valid when the data has finished copying. This has implications on following accesses to this region, that will, instead of creating a new copy information, create a directory lookup, to check if the data is already copied or not.

3) Data free step: When a taskwait arrives, the directory must ensure that all the data the tasks have modified, it's valid again on the host memory, invalidating and freeing allocations on the devices memories.

II. CUDA Comparisons between Unified Memory - Prefetch - Directory

To compare the performance achieved, we compared the performance in GFLOPS using a Blocked Matmul kernel with three memory models, our Directory model implementation, the Prefetch method and Unified memory model with Nanos6 Runtime. The algorithm has already a Tiled matrix, and will call our CUDA MATMUL BLOCK function, which is a Nanos6 task that performs the block multiplication on GPU, having as an in dependence [BS*BS]tileA[i][k] and [BS*BS]tileB[k][j] and as an out dependence [BS*BS]tileC[i][j].

```
for (int i = 0; i < BS; i++)
 for (int j = 0; j < BS; j++)
   for (int k = 0; k < BS; k++)
     CU_MAT_BLOCK(tileC[i][j],
                  tileA[i][k],
                  tileB[k][j], BS);
}
```

Listing 1. Mamtul code which shows how the different tasks are created for performing a tiled-matmul.

The Directory model, is the one that is mentioned in this document, the Prefetch model is using unified memory, but telling the CUDA Driver which regions of memory we are going to use (Basically, prefetching the symbols using the CUDA API), and the Unified memory is calling CUDA without any modification nor notification, as is.

The experiment setup is using one node of Marenostrum 4 Power9 Cluster which contains 2 x IBM Power9 8335-GTH at max frecuency of 3.8GHz, 512GB RAM and we used one of the four NVIDIA V100 (Volta).

The Nanos6 and mercurium versions are the master branch from 8 april 2020, and an experimental Nanos6 branch with support for prefetching CUDA memory.

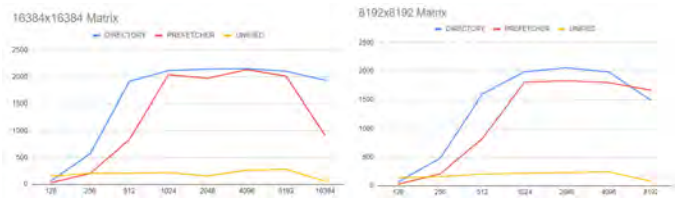

Fig. 2. Graphic representation of the performance results, in BLUE, Directory approach, in RED, prefetcher approach and in YELLOW, unified memory approach, as we can see, the horizontal axis contains the block size, and the vertical axis the performance in GFLOPS, higher is better, where we can see the benefits of our method.

As we can see on the figure 2, for almost all the cases, our directory outperforms the unified memory with, or without prefetcher, with the prefetcher, unified memory seems to be more close to our approach, and with unified memory, without any kind of prefetching, is the clear loser in terms of performance

A. Conclusion

There is a lot of work to be done in this subject, but seeing the preliminary performance results, we can see that it has a huge potential as a transparent way to the user to improve the performance of its applications and allowing multiple devices to communicate independently of it's nature, in fact, this software solution is meant to work in heterogeneous systems, with any kind of device that can compute data.

Implementing this inside Nanos6 will allow to improve performance for already-existing applications built for the OmpSs-2[3] Programming model.

References

[1] "NVIDIA training webpage." [Online]. Available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/

[2] "Nanos6 runtime github." [Online]. Available: https://github.com/bsc-pm/nanos6

[3] "OmpSs-2 webpage." [Online]. Available: https://pm.bsc.es/ompss-2

Ruben Cano Díaz received his BSc degree in Computer Engineering from Universitat Politècnica de Catalunya (UPC), Barcelona, Spain in 2018. He currently is coursing a Master in High Performance Computing at UPC, while working in programming models at the Barcelona Supercomputing Center, Spain.