

A Unified Memory approach to GPU acceleration on task based programming models

Aimar Rodriguez*, Vicenç Beltran*,

*Barcelona Supercomputing Center, Barcelona, Spain

E-mail: {aimar.rodriguez, vbeltran}@bsc.es

Keywords—*High-performance computing, Programming Models, CUDA, GPGPU.*

I. EXTENDED ABSTRACT

Heterogeneous computing has become prevalent as part of High Performance Computing in the last decade, with asynchronous devices such as Graphics Processing Units rapidly advancing. As HPC becomes more specialised and heterogeneous devices improve and develop new features programming models and tools need to adapt in order to keep a competitive performance. In this context, a new version of the OmpSs task based programming model is being developed, which provides an opportunity to introduce the nuances of modern accelerators into the model. In this project, we introduce the implementation of the CUDA GPU programming framework into the OmpSs programming model [1]. The model makes use of the updated *Unified Memory* mechanisms on modern Nvidia GPUs in order to minimise runtime overhead and memory transfer times. This project is developed as part of the Nanos6 runtime project, used for the a new version of the OmpSs programming model.

A. OmpSs-2@CUDA

Graphics Processing Units (GPUs) are used in High Performance Computing (HPC) environments to accelerate the execution of highly parallel workloads. For this, specialised programs called *kernels* are developed, which are then launched using the CPU. A series of characteristics of this form of computation are relevant for the development of a heterogeneous runtime:

- The bulk of the execution is done by the accelerator, while the host CPU is usually only required for setup and launching of the accelerator tasks, leaving the CPU idle most of the time.
- Operations on the GPU are asynchronous, thus, there is no need for the host to be blocked while the accelerator code is running.

Considering this, it is possible to execute accelerator tasks in conjunction with SMP tasks using task based programming models. In order to do this, the runtime can perform a fast operation to launch the asynchronous tasks and use the time in which the GPU is busy to run synchronous CPU tasks.

An implementation of this functionality already exists for the first version of OmpSs [2], however, it present performance issues due to its core design and does not utilise the capabilities

introduced in more modern versions of existing GPU programming languages. To solve this, the CUDA support for the new OmpSs runtime, Nanos6 [3], has been redesigned from the ground up, based on the *CUDA Unified Memory* mechanism present on the latest Nvidia GPU architectures.

Launching a CUDA task requires a number of steps:

- Allocating the CUDA memory
- Transferring the data from the host to the CUDA device
- Launching the kernel
- Waiting for the kernel completion
- Copying the data back to the GPU

1) *CUDA Streams*: Most of these operation are performed on a *CUDA stream*, which is a queue mechanism used to synchronise multiple operations on the CPU. Functions sent to the same stream will be run in the order in which they are invoked, however, there is no guarantee for operations in different streams. Due to this, a usual programming scheme is to launch the copy operations needed for the execution of a kernel before the corresponding kernel in a single stream; this way, it is guaranteed that the kernel will only be run once the transfer operations are completed. This allows to run multiple transfers and kernels in batches without the need for the CPU to intervene beyond an initial launch. This mechanism is used in Nanos6, to launch multiple ready tasks in a batches to different streams.

2) *Unified Memory*: Three of the five steps involved in launching a CUDA tasks are related to memory management; memory allocation, input transfers and output transfers. In order to leverage this functionality from the runtime and reduce the overhead required to track memory, the *Unified Memory* functionality provided since the CUDA version 6 is used. The UM is a mechanism that provides automatic memory transferences between the host and the GPUs. In the latest Nvidia device architectures this is done with a page fault mechanism, which moves data between CPU and GPU when a page fault is triggered. With this, there is no need to implement memory management in the Nanos6 runtime, and thus the overhead it carries to the execution is reduced. The cost for this is that the memory transferences will be slower due to the page faulting mechanism and that the system will only provide CUDA support for Pascal or more modern architectures.

3) *Task synchronisation*: In order to detect the finalisation of asynchronous tasks an event polling method is used. To

check for task completions, a CUDA event is recorded in the stream used for the execution of the task immediately after the kernel. This way, it is possible to know that the kernel has finished execution once the status of the event changes. Similar to tasks, events are polled by the worker threads before running their CPU tasks, since it requires a small amount of time and, thus, will not add much overhead to the execution of SMP tasks.

While there are other possible synchronisation methods, polling is chosen since it allows to keep the asynchronous nature of accelerator task execution without adding much overhead. Waiting mechanisms are discarded since they require blocking a CPU to wait for the CUDA tasks, and *CUDA callbacks* are not used due to delays from kernel completion to callback execution.

4) *Execution Model*: The execution model of the CUDA support for Nanos6 uses all the available worker threads of the runtime to launch asynchronous tasks. Before running a CPU tasks, each worker thread will check if there are any available GPU tasks; if any are available, it will proceed to launch as many as possible in batch before continuing with the SMP task execution. In the same manner, before launching any CUDA task, it will check if any of the currently executing tasks have finished their execution, in order to mark their finalisation and release their dependencies on the runtime.

In addition an additional helper thread exists whose sole task is to launch and check for CUDA task finalisation. This thread runs a loop at a low frequency only performing asynchronous tasks in order to avoid using high CPU time. The reason for the existence of this thread is to avoid starvation of CUDA tasks when all the worker threads are busy executing long running SMP tasks, thus, a minimal service is provided for the GPU tasks as a fallback.

B. Experimentation

The performance of the runtime has been tested using various benchmarks on a single node of CTE-POWER cluster, whose characteristics are the following:

- 2x IBM PowerNV 8335-GTB @ 4.00GHz (10 cores and 8 threads/core, total 160 threads per node)
- 2x nVidia Pascal P100 GPU with 16GB of memory.

A series of benchmarks obtained from the BSC application repository and the Rodinia benchmark suite have been run using both OmpSs and OmpSs-2. A summary of the results can be seen on figure 1

The results of the execution show varying results on different application; both performance losses and improvements have been observed, as well as applications which are not largely affected by the runtime used. Further analysis has shown that the needs for computation or memory transfers on the application determine this.

Compute intensive applications which do not make high usage of memory see speedups when using the new CUDA support at the Nanos6 runtime. This is due to the reduced runtime overhead present on the runtime due to the choice of using Unified Memory and eliminating the need to manually

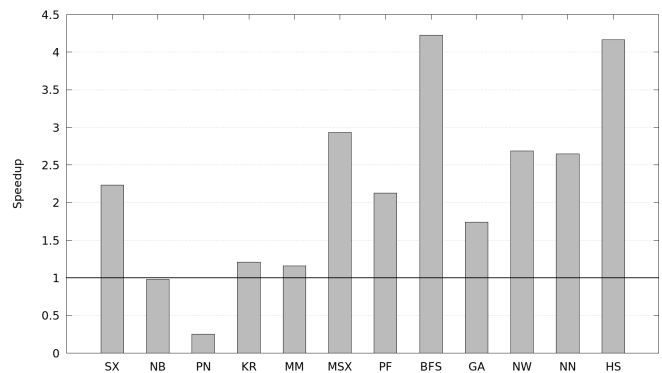


Fig. 1. Speedup evaluation of Nanos6 compared to the previous version

track memory. On the other hand memory intensive applications see their performance lowered, since the Unified Memory mechanism is slower than regular memory transferences when moving data between devices.

C. Conclusion

Finally, we conclude that the new CUDA support system offers opportunities for performance improvements over the old version. While the Unified Memory mechanism does have drawbacks regarding memory intensive applications and running on older systems, it shows speedups on certain application types. Additional optimizations can be applied to the system, such as improved scheduling and usage of CUDA hints to improve memory times, however, it is also possible to explore the performance of other design choices, such as the usage of streams in way that allows for batch asynchronous task execution.

Future work will be focused on implementing a system with manual memory management which includes the design choices on the existing system, as well as finding additional optimizations and exploring different application types and how their performance is affected by the choice of runtime.

REFERENCES

- [1] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. González, F. Igual, D. Jiménez-González, J. Labarta *et al.*, “Extending openmp to survive the heterogeneous multi-core era,” *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 440–459, 2010.
- [2] J. Planas Carbonell, “Programming models and scheduling techniques for heterogeneous architectures,” Ph.D. dissertation, Universitat Politcnica de Catalunya, 2015.
- [3] BSC, “Nanos6 runtime,” <https://github.com/bsc-pm/nanos6>, 2018.



Aimar Rodriguez is a PhD student in Programming Models for Heterogeneous Systems. Aimar studied computer science engineering at the University of Deusto in Bilbao, Spain, obtaining his degree on 2014. After this, he enrolled on the Master in Innovation and Research in Informatics (MIRI) on the *Universitat Politcnica de Catalunya* (UPC) at Barcelona. During his studies, he joined the *Barcelona Supercomputing Center*, where he developed his final master thesis and is currently working on a PhD on the development of Heterogeneous Architecture support for Programming Models.