Support in OpenMP for Multi-GPU Parallelism

Raul Torres [1], Vivek Kale [2], Abid Malik[2], Tom Scogland[4], Roger Ferrer[1], Barbara M. Chapman [3]

[1] Barcelona Supercomputing Center, Barcelona, Spain
[2] Brookhaven National Laboratory, Upton, NY, USA
[3] Stony Brook University, Stony Brook, NY, USA
[4] Lawrence Livermore National Laboratory, Livermore, CA, USA

1 INTRODUCTION

Nodes of emerging supercomputers have multiple GPUs, i.e., a multi-GPU, on them. An application is often parallelized across GPUs of a multi-GPU [3] using MPI, but a more performant as well as a portable solution for parallelizing the application on a multi-GPU is needed. OpenMP [1] is used to parallelize an application on a multi-core or on a GPU, but OpenMP could be employed to achieve parallelism on a multi-GPU. Doing this is beneficial because (1) OpenMP has a lower memory footprint than MPI [3], (2) OpenMP has load balancing features through OpenMP's `schedule` clause and tasking capabilities [7; 10], (3) OpenMP provides for implementing parallelism incrementally while MPI does not [8], and (4) using one programming model within the entire node can reduce programming complexity and potential inefficiencies of interoperability of MPI and OpenMP [4].

OpenMP has proven to be useful to assign work to multiple GPUs on a node by, for example, OpenMP threads collectively offloading tasks containing OpenMP target regions to the GPUs of a multi-GPU [6]. However, direct support is missing in the form of specialized directives in OpenMP and optimizations for parallelization of an application's work across the GPUs of a multi-GPU. An approach of using hybrid OpenMP+OpenACC for programming multi-GPUs has been suggested previously by Xu et al [11], but this work has been done on top of the OpenUH compiler rather than the widely-used LLVM, and it doesn't provide direct language or compiler support for multi-GPU programming in OpenMP.

In this work, we present and experiment with an approach implemented in the widely-used open-source clang/LLVM OpenMP implementation [2] to support OpenMP parallelization of an application's work across GPUs of a multi-GPU. Our contributions of this work are (1) language extensions to OpenMP for partitioning and scheduling work across GPUs and (2) compiler support in the clang/LLVM OpenMP implementation for programming multi-GPUs.

2 SUPPORT IN OPENMP FOR MULTI-GPUS

We propose enabling direct multi-GPU programming with OpenMP through language support and associated compiler optimizations through the widely-used and vendor clang/LLVM OpenMP implementation [2].

The language support in the proposal consists of a new directive called `target spread`, which aims to distribute the offload of a `for` loop onto multiple devices by dividing its execution into chunks. It introduces the `devices` clause, which specifies (1) the devices that will share the workload, and (2) the `spread_schedule` clause, which determines the distribution strategy and the size of the chunks in which the workload will be split. In order to also distribute the arrays according to the specified schedule, two new delimiters have been introduced: `omp_spread_start` and `omp_spread_size`, which will respectively translate into the start and the size of each chunk at execution time.

Example 1: Stencil Operations. In the following example, the `for` loop encloses a stencil sum and has its range adjusted in order to avoid accessing `A` out of bounds. Nevertheless, this could still happen at the chunk level inside the devices' memory. To overcome this issue, it is possible to perform operations involving the delimiters, in order to specify a convenient halo for `A`:

```
#pragma omp target spread \
  devices(2,0,1) \
  spread_schedule(static, 4) \
  map(to:  A[omp_spread_start-1:omp_spread_size+2]) \
  map(from:B[omp_spread_start  :omp_spread_size  ])
for(int i=1;i< N-1;i++){
  B[i]=A[i-1]+A[i]+A[i+1];}
```

Example 2: Dependencies among `target spread` *Regions.* Here, we have two `target spread` regions running asynchronously. By using the new delimiters in the depend clause, it is possible to synchronize them at the chunk level:

```
#pragma omp parallel
#pragma omp single
{#pragma omp target spread \
   nowait \
   devices(2,0,1) \
   spread_schedule(static, 4) \
   map(to:    A[omp_spread_start:omp_spread_size]) \
   map(from:  B[omp_spread_start:omp_spread_size]) \
   depend(out: B[omp_spread_start:omp_spread_size])
 for(int i=0;i<N;i++){
   B[i]=A[i]+10;}
 #pragma omp target spread \
   nowait \
   devices(3,5,4) \
   spread_schedule(static, 4) \
   map(to:   B[omp_spread_start:omp_spread_size]) \
   map(from: A[omp_spread_start:omp_spread_size]) \
   depend(in: B[omp_spread_start:omp_spread_size])
 for(int i=0;i<N;i++){
   A[i]=B[i]+20;}}
```

In our current implementation, the `spread_schedule` clause supports only a static schedule, which consists of orderly assigning the generated chunks, dictated by the chunk size, to the devices of the `devices` clause in a round-robin fashion. If the chunk size is not specified, our scheduler will try to provide each device a single chunk of similar size (`loop_size/num_devices`). OpenMP does not allow explicitly mapping the same array to the same device when the new section overlaps and extends any previously mapped array sections. Here the round-robin strategy automatically complies with the restriction without performing any checks because it provides

each device a set of non-overlapping chunks. A dynamic schedule is more complex because it would have to add a verification step before mapping arrays to avoid violating the mentioned restriction.

3 EVALUATION

We tested the performance of the new `target spread` directive against the existing `target` directive in a micro-kernel called Stream (triad operation)[1]. We compiled both versions of the kernel using our modified clang/LLVM OpenMP implementation (version 12) supporting the new directive. We ran the experiments on the Barcelona Supercomputing Center's CTE-POWER cluster, using a node with 4 NVIDIA V100 GPUs with CUDA 10.1, each GPU capable of running up to 40 streams (streams are command queues which can overlap their execution according to the available resources). The `target` version uses 1 GPU. We have two `target spread` versions, one using 1 GPU, and another using 4 GPUs. The problem size was set to 10 million elements.

Directive	Time (s)	Bandwidth (MB/s)
`target` (1 GPU)	0.880737	273.1
`target spread` (1 GPU)	0.925076	271.7
`target spread` (4 GPUs)	**0.145715**	**1977.5**
target spread (1) vs target	0.95x	1.01x
target spread (4) vs target	6.04x	7.24x
target spread (4) vs target spread (1)	6.38x	7.29x

Table 1: `target` vs. `target spread`.

Chunk size	25000	**62500**	250000	2500000
Time (s)	0.246214	**0.145715**	0.187575	0.324199
Kernels per GPU	100	**40**	10	1
Streams per GPU	**40**	**40**	10	1
Occupancy (streams/max_streams)	**1**	**1**	0.25	0.025
Load Balance (kernels/max_streams)	2.5	**1**	0.25	0.025

Table 2: Different chunk sizes in `target spread` (4 GPUs `max_streams=40`).

In Table 1, the `target spread` directive using 1 GPU showed a small slowdown over the existing `target` directive and very similar bandwidth. Using *target spread* with 4 GPUs achieved 6.38x speedup and 7.29x increase in the bandwidth usage (processed MB/s) over using *target spread* with 1 GPU. We expect the superlinear speedup to be due to the higher bandwidth, and thus lower CPU-to-GPU data movement costs, when running on multiple GPUs with OpenMP. Such improvements suggest benefits of the usage of our language extension and its associated static schedule. Table 2 shows how the variation of the chunk size of the `target spread` impacts performance. We note that as we vary the chunk size, the number of kernels per GPU and the number of used streams per GPU changes. Although generating more than 40 kernels per GPU will ensure the usage of the maximum number of streams of each GPU (40), some of the streams will have a larger queue, resulting in load imbalance. A chunk size of 62,500 generates exactly 40 kernels per GPU, achieving maximum occupancy while also preserving load balance, and ultimately ensuring the best execution time.

[1] https://repo.hca.bsc.es/gitlab/rtorres/target-spread-benchmarks/-/tree/master/kernels/stream/src

4 REMARKS

The `target spread` directive presents two crucial limitations: (1) the distribution of data is tied to the loop distribution (an execution-centric approach), and (2) a `target spread` region must perform data movement between the devices at the time of entering and exiting the region, which may add up to a significant overhead. We note that the `target data spread` directive is now under development in order to decouple loop and data distribution. Its purpose is to create a data region for several devices at the same time where variables can be mapped individually with different strategies (a data-centric approach). This approach allows the presence of multiple enclosed `target spread` regions whose variables have been mapped beforehand, hence skipping the costly stages of data movement between devices and host.

We focus on a static schedule for the `spread_schedule()` given our focus on the compile-time optimization, but using a variety of dynamic schedules [7; 9] is an important step especially for load imbalanced applications. To support dynamic schedules, we can use a scheduling strategy provided in a prototype runtime system where OpenMP threads (1) generate OpenMP tasks (each of which contain one or more kernels) on the CPU, keeping a shared state of GPUs of the multi–GPU, and then (2) work together to dynamically map the OpenMP tasks to GPUs [6]. We note that `spread_schedule` could use a user-defined schedule [5]: a scheduling strategy's name and dequeue function could be defined and used in the application by an application programmer, and the OpenMP runtime system would invoke the dequeue function during the application's execution.

5 CONCLUSION

We have shown in this work our design choices, implementation and early results of features for OpenMP to parallelize across GPUs of a multi-GPU. Further versions of the `target spread` directive might allow the specification of device classes, e.g., `devices(gpu)`, `devices(cpu)`, `devices(fpga)`, which are beneficial for future architectures with extremely heterogeneous processors. Also, we will compare our approach of the OpenMP paralleization across GPUs of a multi-GPU with a more conventional approach of MPI parallelization across the GPUs. Finally, we plan to add support of dynamic schedules in the `spread_schedule()` clause using task-to-GPU scheduling strategies from prior work [6] as well as a more sophisticated set of task-to-GPU scheduling strategies that are (1) adaptive based on historical information about the GPUs during application execution or (2) have variable-sized chunks, e.g., *guided* scheduling, or a custom chunk size distribution in the queue.

6 ACKNOWLEDGMENTS

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was supported in part by MEEP project, which has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 946002. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Croatia, Turkey. We thank Bronis de Supinski and Xavier Teruel for feedback and guidance on the directions of this work.

REFERENCES

[1] OpenMP 5.0 Reference Guide. https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-1119-01-TSK-web.pdf.

[2] The LLVM Compiler Infrastructure. http://llvm.org/.

[3] Leopold Grinberg, Carlo Bertolli, and Riyaz Haque. Hands on with OpenMP4.5 and Unified Memory: Developing Applications for IBM's Hybrid CPU + GPU Systems (Part II), 2017.

[4] Mike Heroux. Preparing for the Sustainable Delivery of the DOE Exascale Software Stack.

[5] Vivek Kale, Christian Iwainsky, Michael Klemm, Jonas H Müller Korndörfer, and Florina M Ciorba. Toward a Standard Interface for User-Defined Scheduling in OpenMP. In *International Workshop on OpenMP*, pages 186–200. Springer, 2019.

[6] Vivek Kale, Wenbin Lu, Anthony Curtis, Abid Muslim Malik, Barbara M. Chapman, and Oscar R. Hernandez. Toward supporting multi-gpu targets via taskloop and user-defined schedules. In Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg, editors, *OpenMP: Portable Multi-Level Parallelism on Modern Systems - 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22-24, 2020, Proceedings*, volume 12295 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 2020.

[7] Yiling Liu. OpenMP - Scheduling(static, dynamic, guided, runtime, auto). https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP/, July 2020.

[8] Ananya Muddukrishna, Peter A. Jonsson, Vladimir Vlassov, and Mats Brorsson. Locality-Aware Task Scheduling and Data Distribution on NUMA Systems. In Alistair P. Rendell, Barbara M. Chapman, and MatthiasS. Miller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 156–170. Springer Berlin Heidelberg, 2013.

[9] C. D. Polychronopoulos and D. J. Kuck. Guided Self-scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions of Computing*, 36:1425–1439, December 1987.

[10] Jaka Speh. OpenMP: For & Scheduling. http://jakascorner.com/blog/2016/06/omp-for-scheduling.html, June 2016.

[11] Rengan Xu, Sunita Chandrasekaran, and Barbara Chapman. Exploring programming multi-gpus using openmp and openacc-based hybrid model. 05 2013.