"© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works." The published version can be found at: DOI: 10.1109/IPDPSW55747.2022.00075

A Novel Set of Directives for Multi-device Programming with OpenMP

Raul Torres, Roger Ferrer and Xavier Teruel
Computer Sciences Department
Barcelona Supercomputing Center
Barcelona, Spain
raul.torres1@bsc.es, roger.ferrer@bsc.es, xavier.teruel@bsc.es

***Abstract*—The latest versions of OpenMP have been offering support for offloading execution to the accelerator devices present in a variety of heterogeneous architectures via the `target` directives. However, these directives can only refer to one device at a time, which makes multi-device programming an explicit and tedious task. In this work, we present an extension of OpenMP in the form of a new set of directives (`target spread` directives) which offers direct support for multiple devices and allows the distribution of data and/or workload among them without explicit programming. This results in an additional level of parallelism between the host and the devices. The `target spread` directives were evaluated using the Somier micro-app in a PowerPC cluster node with up to four Nvidia Tesla V100 GPUs. The results showed a speedup of approximately 2X using four GPUs and the new directive set, in comparison with the baseline implementation which used one GPU and the existing `target` directive set.**

***Index Terms*—OpenMP, language extension, multi-device support, multi-GPU, heterogeneous architectures, offloading, LLVM, accelerators**

I. Introduction

High-Performance Computing (HPC) is entering the exascale era [1], where supercomputers are increasingly being built on top of heterogeneous architectures [2]. One commonly found component in such architectures are accelerators, which are hardware devices specially designed for performing intensive parallel computations and which have been progressively contributing on the overall increase of the performance of these machines [3]. Examples of accelerators are Graphic Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs).

The OpenMP community, aware of this trend, included direct support for accelerator offloading via the **`target`** directives since the specification 4.0 [4]. However, in the last years, the architectural landscape has drastically changed and it is more common to find clusters whose computing nodes have more than one accelerator. The CPU is progressively taking a coordination role while most of the intense computation is offloaded to the accelerators. Unfortunately, the current OpenMP specification does not offer a satisfactory answer for programming these multi-device systems in a straightforward way. In these systems, the current specification forces to program each device individually making the application code explicitly multi-device aware.

This work was supported 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.

Working explicitly with multiple devices poses two main challenges on the programmer side: workload- and data- distribution. At this level, the programmer should be responsible for manually distributing data and, once data is distributed, offloading the execution of code to the corresponding device. A better approach consists of letting the programmer to partition the loop iteration space using the traditional work-sharing constructs (e.g., **`parallel for`**).

Our proposal consists of avoiding explicit and ad-hoc implementations of multi-device programming in OpenMP, and favouring direct support for it in the compiler, in the form of an extended set of OpenMP directives specially designed for such purpose.

In this paper the terms *device* and *accelerator* can be used interchangeably. However we will favour the use of device for the sake of clarity and simplicity.

This paper is organized as follows: Section II describes concisely the current offloading model of OpenMP. Section III explains the details of our proposed model and directives, while Section IV makes a review of similar efforts. In Section V we show a use case for the evaluation of the proposed model while in Section VI we present the obtained results. Finally, Section VII discusses the learned lessons, Section VIII wraps up the conclusions and Section IX offers a glimpse of the future directions of this research.

II. Current device support for OpenMP

OpenMP supports offloading work from the host to a device via the **`target`** directive, which offloads work to a single device (see line 2 of Listing 1), and does not present any restriction about the type of the subsequent structured block of code annotated by the directive (line 5), as long as it is valid for the device.

Listing 1: OpenMP *target* directive example.

```
1  #pragma omp target \
2    device(0) \
3    map(to:  A[0:N]) \
4    map(from:B[1:N-2])
5  {    ...   }
```

When using a fully parallelized combined directive like **#pragma omp target teams distribute parallel for simd** (see Listing 2), the directly available parallelism levels are intra-device only [5]:

1) Multiple teams (**teams distribute**).
2) Multiple threads (**parallel for**).
3) Multiple vector lanes (**simd**).

Listing 2: OpenMP *target* combined directives example.

```
1  #pragma omp target teams distribute parallel for simd \
2    device(0) \
3    num_teams(2) \
4    map(to:  A[0:N]) \
5    map(from:B[1:N-2])
6  for(int i=1;i< N-1;i++){
7    B[i]=A[i-1]+A[i]+A[i+1];}
```

Unless the vendor provides a way to virtualize more than one accelerator under a single OpenMP device, distributing work over several devices has to be explicitly programmed.

III. Proposed multi-device support for OpenMP

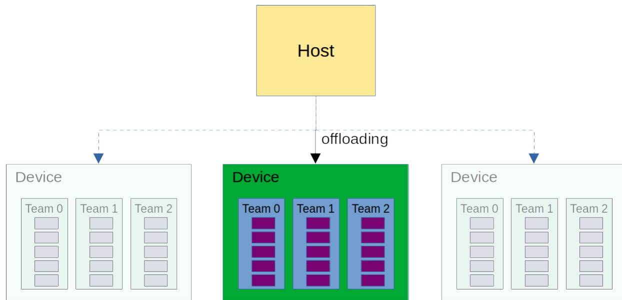
A. Proposed model

Our proposal consists on enabling direct multi-device support in OpenMP in the form of an additional level of parallelism, situated between the host and the target, and on top of the existing ones:

1) **Multiple devices (target spread).**
2) Multiple teams (**teams distribute**).
3) Multiple threads (**parallel for**).
4) Multiple vector lanes (**simd**).

In contrast to offloading the work to a single device, as in the current OpenMP model, the host first distributes that work among the available devices (see Figure 1).

Fig. 1: Extension of the OpenMP target offloading model.



B. Directives for multi-device programming

In order to employ this model, we have implemented a new set of directives whose syntax is very similar to the one of the existing **target** directives. By simply adding the keyword **spread** after the **target** keyword, we are enabling the distribution of data and/or workload among devices.

In a previous publication [6] we have proposed the standalone **target spread** directive as an extension of the **target** directive. In this paper we extend the set of directives that complement **target spread**, by adding support to intra-device parallelism (**target spread teams distribute parallel for**) and data management (**target data spread**, **target enter/exit data spread**, **target update spread**).

1) target spread: The aim of **target spread** is to offload the workload over multiple devices by dividing the execution into chunks (see Listing 3). This introduces one restriction: the subsequent structured block of code must be a loop (line 6).

Listing 3: Proposed *target spread* directive.

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

We have replaced the **device** clause of **target** with a new **devices** clause (line 2), to allow the specification of multiple accelerators, which are supposed to share the workload. The order of distribution is determined by the position in this list, not by the device identifier. As we offload work over several devices, the **spread_schedule** clause determines the distribution strategy and the size of the chunks in which the loop will be split (line 3). At the moment, the directive supports only **static** schedule, which performs distribution of chunks among the devices in a round-robin fashion, e.g, for Listing 3, if N is 14, the loop iterations would be distributed as follows:

- Iterations 1, 2, 3 and 4 go to device 2.
- Iterations 5, 6, 7 and 8 go to device 0.
- Iterations 9, 10, 11 and 12 go to device 1.

The distribution changes if the chunk size changes, e.g. if chunk size is now 2 (via **spread_schedule(static, 2)**):

- Iterations 1 and 2 go to device 2.
- Iterations 3 and 4 go to device 0.
- Iterations 5 and 6 go to device 1.
- Iterations 7 and 8 go to device 2.
- Iterations 9 and 10 go to device 0.
- Iterations 11 and 12 go to device 1.

To map arrays according to the specified schedule, two special variable identifiers have been introduced (line 4): **omp_spread_start** makes reference to the start of each chunk at execution time, while **omp_spread_size** does the same with its size. Mapping halos (i.e. borders) can be easily expressed by performing simple arithmetic with these delimiters.

Like **target**, **target spread** supports asynchronous execution via the **nowait** clause.

The OpenMP standard already includes the do across dependences [7], [8] based on the keywords **source** and **sink** to establish intra-loop dependencies (i.e., loop-carried

dependences) for the loop-worksharing construct. This mechanism is based on the iteration identifier. Existing proposals [9] extend dependencies to allow intra- and inter- loop dependences based on the data used rather than on a given iteration space. Data dependencies also allow us to describe the application parallelism following a dataflow schema, which may increase the level of parallelism and data locality. Our proposed **depend** clause follows this latter approach.

2) target spread teams distribute parallel for: The new directive **target spread** can be used as a combined directive as well (see Listing 4):

1) The **target spread** directive distributes the work among devices.
2) The **teams distribute** directive creates the teams and distributes the workload among them.
3) The **parallel for** activates the parallel threads of each team.

Listing 4: Proposed *target spread* combined directives.

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

The directive information related to **teams distribute** is applied in a per-device basis, e.g. each device will have as many teams as specified by the **num_teams** clause (line 4).

3) target data spread: Recall that the existing **target data** directive is applied over a subsequent structured block of code (see line 5 of Listing 5). The same applies to the new **target data spread** directive (line 15). However, in order to distribute data mappings, the new directive uses more than one device (line 10), and requires additional information, like the range (line 11) and the size of the chunks in which the range will be split (line 12). Because the proposed directive always assumes a static round-robin distribution of data, we did not include a **spread_schedule** clause. As its predecessor, the new directive does not support asynchronous transfers (there is no **nowait** clause) nor dependencies.

Listing 5: *target data* and *target data spread* compared.

```
1  #pragma omp target data \
2    device(0) \
3    map(to:A[0:N], \
4           B[1:N-2])
5  {
6    ...
7  }
8
9  #pragma omp target data spread \
10   devices(2,0,1) \
11   range(1:N-2) \
12   chunk_size(4) \
13   map(tofrom:A[omp_spread_start-1:omp_spread_size+2], \
14              B[omp_spread_start  :omp_spread_size  ])
15 {
16   ...
17 }
```

4) target enter/exit data spread: OpenMP provides the standalone data directives (i.e. not associated to user code) **target enter/exit data** (see Listing 6), which follow a similar syntax like the previous one. Our proposal includes **spread** equivalents for those directives as well. The **nowait** clause allows data transfers to happen asynchronously (lines 3, 9, 17 and 25). However, the **depend** clause is not supported yet.

Listing 6: *target enter data* and *target enter data spread* compared.

```
1  #pragma omp target enter data \
2    device(0) \
3    nowait \
4    map(to:A[0:N], \
5           B[1:N-2])
6
7  #pragma omp target exit data \
8    device(0) \
9    nowait \
10   map(from:A[0:N], \
11            B[1:N-2])
12
13 #pragma omp target enter data spread \
14   devices(2,0,1) \
15   range(1:N-2) \
16   chunk_size(4) \
17   nowait \
18   map(to:A[omp_spread_start-1:omp_spread_size+2], \
19          B[omp_spread_start  :omp_spread_size  ])
20
21 #pragma omp target exit data spread \
22   devices(2,0,1) \
23   range(1:N-2) \
24   chunk_size(4) \
25   nowait \
26   map(from:A[omp_spread_start:omp_spread_size], \
27            B[omp_spread_start:omp_spread_size])
```

5) target update spread: Similar to the original counterpart, the new directive performs updates of previously mapped data in the host and/or in the devices, but in a distributed manner (see Listing 7). Asynchronous updates are supported via the **nowait** clause (line 11). The **depend** clause is not supported yet.

Listing 7: *target update* and *target update spread* compared.

```
1  #pragma omp target update \
2    device(0) \
3    nowait \
4    to(  A[0:N]) \
5    from(B[1:N-2])
6
7  #pragma omp target update spread \
8    devices(2,0,1) \
9    range(1:N-2) \
10   chunk_size(4) \
11   nowait \
12   to(  A[omp_spread_start-1:omp_spread_size+2]) \
13   from(B[omp_spread_start  :omp_spread_size  ])
```

C. Implementation details

The proposed model and related directives where implemented on top of the LLVM compiler [10][1]. Most of the changes were done in the OpenMP side of the Clang front-end, and involved the following parts of the infrastructure:

- Lexical module.

[1]Commit 84adaabf3e04d1938a137b1299a677d2fa489383

- Parser.
- AST builder.
- Semantics module.
- Code generator.

Minor changes in the OpenMP runtime were necessary for supporting the static scheduling for the **`target spread`** directive.

IV. Related work

Work and data partition among different processing elements is a common problem that must be addressed in any HPC programming model. Distributed environments could solve data partition using explicit or implicit communication (both implemented in MPI [11] through two-sided and one-sided communication services, respectively). Distributed systems may also implement *Partitioned Global Address Spaces* (PGAS), where the implicit communication still happens at runtime level but they offer the perception of a single memory address space. Such programming models usually first distribute the data and then execute the associated work where the data is placed.

X10 [12] provides the concepts of *places*, *objects*, and *activities*. Objects and activities remain tied to the associated place while places can migrate across different physical locations. Chapel [13] implements this distribution employing *domains*, and there were also proposes to extend this functionality allowing users to express more complex policies (e.g., Dynamically Load-Balanced Domain Maps [14]). The Unified Parallel C (UPC) [15] provides a work-sharing construct with the *affinity* annotation which specifies the association among work and data.

Concerning OpenMP, several proposals have been done to address the usage of multi-devices. One of them shows how OpenMP can be useful to assign work to multiple GPUs on a node by collectively offloading tasks containing OpenMP target regions to the GPUs of a multi-GPU environment [16]. However, their implementation is explicitly performed using the current language features, and not directly implemented into the compiler infrastructure. Other authors have proposed a source-to-source translator capable to parse C code annotated with OpenACC directives and generate a multi-GPU version in the same language but annotated with OpenMP and OpenACC directives [17]. The resulting code still has to be compiled afterwards. Yet another effort using a hybrid OpenMP+OpenACC approach for programming multi-GPUs has been proposed in [18], which was done on top the OpenUH compiler.

The following proposals are closer to the ideas we are developing in this work.

A. target device(any) and if_device(...)

Similar to our approach, previous work has been done in order to extend the **`target`** directive in order to enhance OpenMP to support execution in multiple devices [19]. It was implemented on top of the MACC compiler and the OmpSs runtime [20]. Their approach consisted on letting the **`device`** clause accept the keyword `any` as parameter, which would force the generation of code for all the different types of devices available in the system. The clause **`if_device`** permits the specification of tailored code for an specific type of device. However, the main goal is enabling the use of devices of different types; there is no notion of work distribution among the devices, which has to be programmed by hand, i.e. by embedding the target region inside a for loop that decomposes the problem into chunks beforehand.

B. parallel target and parallel for distribute

Another previously proposed language extension was implemented on top of the HOMP compiler [21]. In this work, a) they extended the **`target`** directive by adding the **`parallel`** keyword before, which creates a context for multi-device execution; b) they added the **`distribute`** keyword after a **`for`** loop to indicate that the iterations must be distributed among the devices of the parallel target region; c) they added the **`partition`** modifier to the **`map`** clause, in order to specify how the data should be distributed; d) the **`device`** clause was modified to accept more than one device.

In contrast to our approach, their parallel target region is not restricted to be applied to a loop only; instead it creates something similar to a classical OpenMP parallel region but instead of having threads, it has devices. If the goal is to distribute a loop, the **`parallel for distribute`** construct has to be used inside that parallel region. In our proposal the particular interest was the design of a model that eases the distribution of the workload among the present devices, so we preferred the invention of a new directive that could be applied directly to **`for`** loops, hence skipping the case of a parallel region of devices.

Another important difference is that in their approach, only **`parallel target`** or **`parallel target data`** directives perform data mapping and distribution, which forces execution directives, like **`parallel for distribute`** or **`parallel for target`**, to align to them or viceversa via the **`align`** modifier. Our approach preserves the semantics of the current OpenMP specification, and makes the **`target spread`** data directives independent from the **`target spread`** executable directive, meaning they are both allowed to map and distribute data. This imposes on the programmer the duty of ensuring that both distributions match, but maintains the flexibility that the current standard offers.

We also noticed that in their model, distribution and alignment are performed in the scope of the **`map`** clause, which allows different variables to be mapped and distributed in different ways, as well as aligned to different loops inside the parallel target region. However, the list of targeted devices is rather applied at the directive level. This means the programmer is given the flexibility to shape the distribution at will but has to work always with the same device list. Moreover, dependencies via a **`depend`** clause, which is not addressed in their paper, if implemented, would have

to implant similar modifiers inside the clause scope, which introduces more complexity to an already complex OpenMP specification.

During the design process of our proposal we also experimented with a similar setup, even letting the programmer determine the device list per **map** clause. Nevertheless we realized this level of customization would impose serious restrictions for further extension of the model e.g. arising ambiguities for the dependencies support, which are important to reduce unnecessary global barriers. For that reason, we discarded such model and moved all the distribution customization from the **map** clause to the directive level, which then applies to all the mappings associated to it. In case the programmer needs different mappings, they can be fully expressed using unstructured data constructs like **target enter/exit data spread**, where the list of devices for each construct can be different too (see lines 2 and 9 of Listing 8):

Listing 8: Mapping variables in different ways with multiple *target data spread* directives.

```
1  #pragma omp target enter data spread \
2    devices(2,0) \
3    range(1:N-2) \
4    chunk_size(4) \
5    nowait \
6    map(to:A[omp_spread_start-1:omp_spread_size+2])
7
8  #pragma omp target enter data spread \
9    devices(1,3) \
10   range(100:M) \
11   chunk_size(10) \
12   nowait \
13   map(to:B[omp_spread_start:omp_spread_size])
```

Finally, some of the syntax proposed by the referenced authors does not clearly integrate with the current OpenMP standard, e.g. how the **parallel for distribute** would compose with the existing **teams distribute parallel for**, or how would **target enter data**, **target exit data** or **target update** could be employed together with **parallel target**, just to mention a few.

In summary, in our research we have tended towards preserving the current semantics of OpenMP by adding new directives designed to look as natural extensions of the existing ones, which make them easier to learn, and with the ulterior goal of making the implementation of multi-GPU code a less tedious and error-prone task.

V. Use case: Somier mini-app

We tested the performance and usefulness of the new **target spread** directive set against the existing **target** directive set in the Somier mini-app, for which we implemented three different versions that will be detailed later in this section[2]. The Somier mini-app simulates a grid of springs. To do so, for each time step it performs the computation of four variables stored in the cells of a three-dimensional grid:

- Forces.
- Acceleration.
- Velocities.
- Positions.

While most of the kernels only need the local cell information to perform operations, the forces kernel is a stencil computation that uses the neighbour cells. This requires the use of halos between buffers and between chunks in the outermost dimension.

An additional kernel computes centers using the positions in a reduction operation. We currently do not support a **reduction** clause yet, so we implemented a manual reduction for this kernel.

For our experiment, we used GPUs as the accelerator device, due to their widespread use and availability in the HPC centers. The three implementations were compiled using our modified version of the Clang/LLVM compiler and were run on the Barcelona Supercomputing Center's CTE-POWER cluster, using a node with up to four NVIDIA V100 GPUs with CUDA 10.1. Each accelerator has 16 gigabytes (GBs) of global memory. The underlying operating system was Red Hat Enterprise Linux Server version 7.5.

There has been a variety of studies on the problem of work distribution among multiple accelerators with OpenMP, e.g. [22], [23]. There, for example, the authors tried to provide a handful of distribution strategies that could eventually fit to real-world programs where the distribution of data is more complex than simply splitting the data among the available devices. But sometimes, certain problems require migrating data between accelerators, which could cause that, in the long run, the gained parallelism does not compensate the generated overhead. In other situations, the intensiveness of computation might differ from accelerator to accelerator, leading to imbalance when a static scheduling strategy is used. Mitigating such effects would require implementing a dynamic scheduling strategy.

It is not in the scope of this paper the study of such complex techniques. That is why we chose a scenario in which the problem was too big to fit at once in the accelerators' memory but could be split in a relatively easy manner into smaller parts that fit into them, and which can be distributed in a straightforward way using the round-robin strategy of the static scheduling of **target spread**.

In order to reflect these conditions, we set the problem size to approximately ten times the memory capacity of each GPU (8 bytes $\times 1200^3 \times 3 \times 4 = 154.5$ GB):

- Each cell of the grid stored a double precision floating point value of 8 bytes.
- Each dimension of the 3D grid had a size of 1200 cells.
- Each of the 4 variables of the problem required 3 3D-grids.

The number of time steps was set to 31.

A. Implementation 1: One buffer at a time

This corresponds to straightforward implementation that aims to distribute a buffer among the available GPUs using

[2] https://repo.hca.bsc.es/gitlab/rtorres/target-spread-benchmarks/-/tree/master/kernels/somier

their full memory capacity.

1) Baseline implementation with target directives: Listing 9 depicts a simplified view of the process using the existing **target** based directives. The problem is split into buffers that fully occupy the device memory (line 2). Due to the fact that buffers are processed in a sequential way, there is no risk of overlapping halo memory, so one GPU can be used safely. The data are mapped to the device (line 5), making them available beforehand for the computation, where we deploy the intra-device parallelism using combined directives. As mentioned above, the computation consists of more than one kernel: forces, accelerations, velocities, positions and centers (lines 10, 15 and 20). Finally, we update host data with the results of the computation (line 23).

Listing 9: Baseline implementation of *One Buffer* strategy with *target* directives.

```
1  /* process 1 buffer at a time */
2  for(int buffer_start=0; i<N; i+=buffer_size)
3  {
4    /* map data from host to devices */
5    #pragma omp target enter data \
6      device(0) map(...)
7
8    /* perform kernel computations in device */
9    // forces kernel
10   #pragma omp target teams distribute parallel for \
11     device(0) map(...)
12   for(int i=buffer_start;i<buffer_start+buffer_size;i++)
13     ...
14   // accelerations kernel
15   #pragma omp target teams distribute parallel for \
16     device(0) map(...)
17   for(int i=buffer_start;i<buffer_start+buffer_size;i++)
18     ...
19   // other kernels
20   ...
21
22   /* map data from host to devices */
23   #pragma omp target exit data \
24     device(0) map(...)
25 }
```

2) Implementation with target spread directives: The **target spread** version (see Listing 10) introduces some minor changes to the baseline version. First, the problem is split into buffers that sum up for the total amount of memory of the devices. Secondly, the chunk size for each GPU is simply calculated as the size of the current buffer divided by the number of devices (line 5); this parameter is needed by the **chunk_size** (line 11 and 37) and **spread_schedule** (line 18 and 25) clauses of the **target spread** directive set in order to guide the static distribution among the GPUs, which is then performed automatically. Finally, data are mapped now to more than one device (line 12, 19, 26 and 38).

The **depend** clause makes it possible to synchronize kernels at the chunk level. The **nowait** clause and the enclosing **taskgroup** areas make sure that mapping and/or computation occur asynchronously among devices, but forces synchronization at exit from the group.

It is important to mention that the coherence between the mappings of the different directives is the programmer's responsibility, e.g. the **range** and **chunk_size** configurations of the **target enter data spread** directive should respectively match the loop range and **spread_schedule** configurations of the **target spread** directive.

Listing 10: *One Buffer* implementation with *target spread* directives.

```
1  /* process 1 buffer at a time */
2  for(int buffer_start=0; i<N; i+=buffer_size)
3  {
4    /* each device gets a chunk from a buffer */
5    int chunk=buffer_size/num_devices;
6
7    /* map data from host to devices asynchronously */
8    #pragma omp taskgroup
9    {
10     #pragma omp target enter data spread \
11       range(buffer_start:buffer_size) chunk_size(chunk) \
12       devices(1,0,3,...) map(...) nowait
13   }
14
15   /* perform computation in devices asynchronously */
16   // forces kernel
17   #pragma omp target spread teams distribute parallel for\
18       spread_schedule(static, chunk) \
19       devices(1,0,3,...) map(...) nowait \
20       depend(out:...)
21     for(int i=buffer_start;i<buffer_start+buffer_size;i++)
22       ...
23   // accelerations kernel
24   #pragma omp target spread teams distribute parallel for\
25       spread_schedule(static, chunk) \
26       devices(1,0,3,...) map(...) nowait \
27       depend(in:...) depend(out:...)
28     for(int i=buffer_start;i<buffer_start+buffer_size;i++)
29       ...
30   // other kernels
31     ...
32
33   /* map data from host to devices asynchronously */
34   #pragma omp taskgroup
35   {
36     #pragma omp target exit data spread \
37       range(buffer_start:buffer_size) chunk_size(chunk) \
38       devices(1,0,3,...) map(...) nowait
39   }
40 }
```

B. Implementation 2: Two Buffers

This implementation assumes that by processing two half buffers at a time, we increase the probability of overlapping in time transfers and computations from different buffers, which in theory would reduce the execution time. However, we have no control on when the overlap could happen.

Listing 11 depicts a simplified view of the process. In this version, instead of using a full buffer size, we used a half of it (line 2). In that way, we can process two half buffers at the same time without running out of memory in the devices. The **taskloop** directive makes possible to process multiple half buffers at the same time, while its **num_tasks** clause limits the number of simultaneous half buffers to two (line 5). The rest works similar to the previous version.

At a certain point, a GPU could be receiving data from two consecutive buffers at the same time. If we had only one GPU, the halo memories might overlap in space and the runtime will detect it as an explicit extension of an array, which is forbidden in OpenMP. In order to avoid this situation, more than one GPU has to be used, this way the round-robin schedule makes sure there is always a gap between the array sections mapped to a particular device.

Listing 11: *Two Buffers* implementation with *target spread* directives.

```
/* split buffers into half buffers */
int half_size=buffer_size/2;

/* process 2 half buffers at a time */
#pragma omp taskloop num_tasks(2)
for(int half_start=0; i<N; i+=half_size)
{
  /* each device gets a chunk from a half buffer */
  int chunk=half_size/num_devices;

  /* map data from host to devices asynchronously */
  #pragma omp taskgroup
  {
    #pragma omp target enter data spread \
      range(half_start:half_size) chunk_size(chunk) \
      devices(1,0,3,...) map(...) nowait
  }

  /* perform computation in devices asynchronously */
  // forces kernel
  #pragma omp target spread teams distribute parallel for\
      spread_schedule(static, chunk) \
      devices(1,0,3,...) map(...) nowait \
      depend(out:...)
    for(int i=buffer_start;i<buffer_start+buffer_size;i++)
      ...
  // accelerations kernel
  #pragma omp target spread teams distribute parallel for\
      spread_schedule(static, chunk) \
      devices(1,0,3,...) map(...) nowait \
      depend(in:...) depend(out:...)
    for(int i=buffer_start;i<buffer_start+buffer_size;i++)
      ...
  // other kernels
    ...

  /* map data from host to devices asynchronously */
  #pragma omp taskgroup
  {
    #pragma omp target exit data spread \
      range(half_start:half_size) chunk_size(chunk) \
      devices(1,0,3,...) map(...) nowait
  }
}
```

C. Implementation 3: Double Buffering

The rationale behind this implementation is to gain more control over the overlap by dispatching memory transfers from host to device at the same time with kernel computations.

Similar to the previous version, it works with two half buffers to avoid running out of memory in the devices, but instead of using a **`taskloop`**, it employs a recursive routine. Listing 12 depicts a simplified view of the process. We start processing the first half buffer (line 5), which calls the recursive routine (line 9) performing the following operations:

1) Data transfer from host to device (line 17).
2) Recursive call with next buffer data (line 25) embedded in a task (line 24) for asynchronous execution.
3) Kernel computations (line 30 and on).
4) Data transfer from device to host (line 49)

Notice that, due to the **`task`** directive in line 24, the kernel computations in line 30 and on, do not wait for the called function in line 25 to finish; instead they start executing immediately. This has the intention to dispatch serialized transfers from host to device, while increasing the chances of overlapping to the kernel computations.

Listing 12: *Double Buffering* implementation with *target spread* directives.

```
/* split buffers into half buffers */
int half_size=buffer_size/2;

/* call the routine for the first time */
foobar(half_start, half_size, ...);

...

void foobar(int half_start, int half_size, ...)
{
  /* each device gets a chunk from a half buffer */
  int chunk=half_size/num_devices;

  /* map data from host to devices asynchronously */
  #pragma omp taskgroup
  {
    #pragma omp target enter data spread \
      range(half_start:half_size) chunk_size(chunk) \
      devices(1,0,3,...) map(...) nowait
  }

  /* the routine calls itself inside an asynchronous task */
  if(half_start+half_size < total_size){
    #pragma omp task
    foobar(half_start+half_size, half_size, ...);
  }

  /* perform computation in devices asynchronously */
  // forces kernel
  #pragma omp target spread teams distribute parallel for\
      spread_schedule(static, chunk) \
      devices(1,0,3,...) map(...) nowait \
      depend(out:...)
    for(int i=buffer_start;i<buffer_start+buffer_size;i++)
      ...
  // accelerations kernel
  #pragma omp target spread teams distribute parallel for\
      spread_schedule(static, chunk) \
      devices(1,0,3,...) map(...) nowait \
      depend(in:...) depend(out:...)
    for(int i=buffer_start;i<buffer_start+buffer_size;i++)
      ...
   // other kernels
    ...

  /* map data from host to devices asynchronously */
  #pragma omp taskgroup
  {
    #pragma omp target exit data spread \
      range(half_start:half_size) chunk_size(chunk) \
      devices(1,0,3,...) map(...) nowait
  }
}
```

VI. Results

A. target vs target spread in One Buffer implementation

For this comparison, the **`target`** based implementation targeting one GPU was used as baseline. The other implementation used the **`target spread`** directives targeting one, two and four GPUs (Table I). The reported numbers correspond to the total execution time of the program, measured with the operating system tool `time`.

TABLE I: Execution times for the One Buffer implementation ((B) stands for baseline).

Directive	**target (B)**	**target spread**		
GPUs	**1**	**1**	**2**	**4**
Time	17m40.231s	17m38.932s	13m15.486s	8m22.019s

The results showed that using one GPU, the baseline implementation and the one based on the new directives

have similar execution times. This indicates that a negligible overhead is introduced by using these new directives.

Adding up a second GPU showed approximately 1.4X overall speedup, while the usage of four GPUs raised it to more than 2X. However, it is important to notice that, internally, the kernel computations had near to linear speedup when more GPUs were added to the configuration. This suggests the occurrence of a communication bottleneck introduced when transferring data to and from multiple GPUs, which downgrades the overall speedup of the program.

B. *One buffer vs Two buffers vs Double buffering*

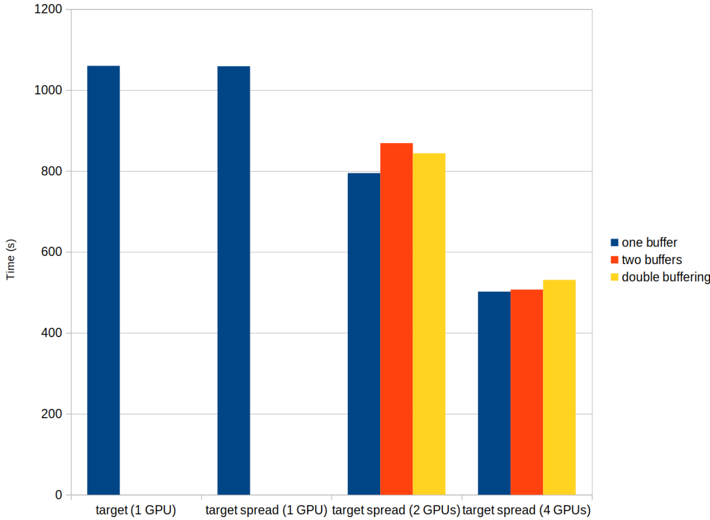
Due to the mentioned risk of overlapping halo memory, the *Two Buffers* and *Double Buffering* versions could not be tested with any of the directives using only one GPU, so a comparison with the **`target`** based directives was not feasible. For this reason, the baseline in this case is the *One Buffer* version based on **`target spread`** (see Table II).

With two GPUs, the *One Buffering* version seemed to be faster than the other two (Figure 2), while with four GPUs, the three versions showed more similar execution times.

TABLE II: Execution times for the different Somier implementations ((B) stands for baseline).

Directive	**target spread**	
GPUs	**2**	**4**
One Buffer (B)	13m15.486s	8m22.019s
Two Buffers	14m29.599s	8m26.674s
Double Buffering	14m4.230s	8m51.176s

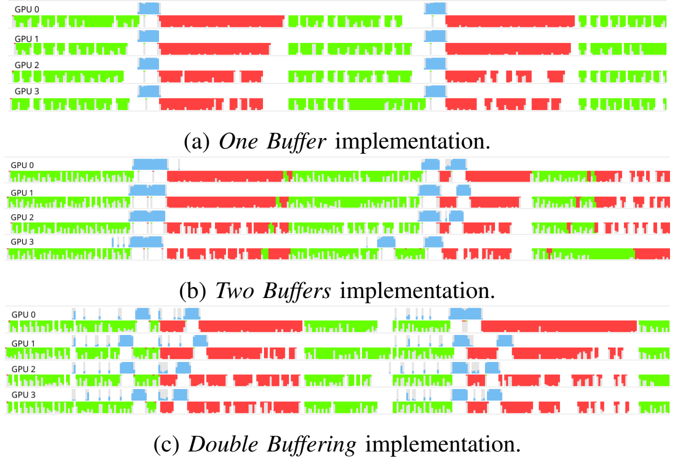
Fig. 2: Time comparison of the Somier implementations.



In order to understand this behavior, we analyzed the profiling traces obtained with NVIDIA's tool **nsys**[3]. The analysis showed that the execution time was mainly dominated by memory transfers and not by kernel computations (see Figures 3a, 3b and 3c).

[3]https://docs.nvidia.com/nsight-systems/UserGuide/index.html

Fig. 3: 10 seconds of NVIDIA's nsys traces showing memory transfers from host to four GPUs and vice-versa (in green and red) and kernel computations (in blue).



(a) *One Buffer* implementation.

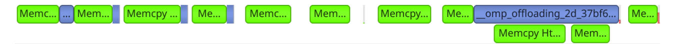
(b) *Two Buffers* implementation.

(c) *Double Buffering* implementation.

A deeper look into the single-GPU level of the traces of both *Two Buffers* and *Double Buffering* implementations (see Figure 4) showed the following:

- The five kernel computations were not executed subsequently, but interleaved with data transfers from a different buffer.
- Overlap of computation and transfers from different buffers happened in vary rare occasions.
- Transfers from different buffers did not overlap.

Fig. 4: Sample of a trace showing transfers and computations in a single GPU.



The interleaving of transfers and computations could be generated by an inadequate granularity of the memory transfers, i.e. having 4 variables represented by 3 separated grids implies making 12 sequential calls to the underlying CUDA memory copy API per mapped chunk. Added to this is the fact that the kernel computation is small compared to the transfers and it is compound by multiple kernels.

VII. Discussion

The usage of the **`target spread`** directives has allowed us to deploy a multi-device version of the Somier mini-app without having to explicitly write the code to coordinate the worksharing among the devices, and without adding up to a significant overhead. Instead, we only had to modify the baseline version written with the existing **`target`** directives, by using the new **`target spread`** directives, which share a similar syntax, and are supported by additional clauses that make it possible to specify the involved devices, the distribution strategy, the range and the chunk size.

A detailed look at the code will show that the kernel computation is compound by multiple kernels synchronized at the chunk level with the usage of dependencies among subsequent **target spread** directives. These synchronization is not yet possible with **target enter/exit data spread**, for the implementation of the **depend** clause on these directives is still a work in progress. Therefore, one must ensure the whole data are available before the computation, which is only possible by adding a barrier that synchronizes all devices. To do so, we opted for the usage of **taskgroup** over **taskwait**, because the former forces synchronization only among the tasks of the group, while the latter does it for all the tasks of the parent parallel region. However, such barrier is preventing that chunks already present in the device memory can be used immediately by the kernel computations: all chunks in all devices must have been copied in order to start calculations.

Such synchronization configuration lowered the chances of overlapping computation and transfers when two half buffers at a time were being processed. Moreover, data transfers dominated the time, while the kernel computations were really short, hence being interleaved with, rather than overlapped to the transfer operations.

At the end, dispatching the kernel computation together with the data transfers in the *Two Buffers* and *Double Buffering* versions decreased the overall performance of the program.

target enter/exit data spread create unstructured data transfers, meaning they can happen anywhere in the code, and the mappings are not directly related to each other. In contrast, structured data transfers like **target data spread** create a scope during which the mappings are valid. The choice of using unstructured instead structured data transfers was motivated by two factors:

- Unstructured data transfers can be done asynchronously.
- The mapping at enter and exit can be different, which is important when working with halos.

VIII. Conclusions

In this paper we have presented a new set of directives for multi-device targeting in OpenMP, implemented on top of the popular LLVM compiler. We have shown as well how a single-GPU implementation of the Somier micro-app can be converted to multi-GPU code by using this new set of directives and its corresponding support clauses, which spares the time required to explicitly implement a multi-GPU version by hand using the existing **target** directives. Experimentation showed that executing this new code in a cluster node with 4 GPUs was 2 times faster than doing the same with the baseline single-GPU code. This demonstrates the suitability of the proposed approach, whose model can serve as a guideline for future OpenMP specifications. However, currently the implementation does not support dependencies, which harms this particular approach requiring strong synchronization constructs (i.e., **taskgroup**).

IX. Future Work

We are aware that there are two fronts susceptible for improvement: implementation and experimentation.

The immediate step of our research efforts is the implementation of the support for dependencies in the **target enter/exit data spread** directives, which will be useful for synchronizing data transfers with kernel computations at the chunk level, making the enclosing **taskgroup** directive no longer necessary. This feature will effectively eliminate the gaps in time where some of the devices remain idle while waiting for the full transfer to finish. They will be easily expressed in a similar fashion to the mappings and with the same semantics for the array sections (see Listing 13).

Listing 13: Future support for dependencies in *target spread* data directives (Not implemented in red, already implemented in blue).

```
1  #pragma omp target enter data spread \
2    devices(1,3) \
3    range(100:M) \
4    chunk_size(10) \
5    nowait \
6    map(to:B[omp_spread_start:omp_spread_size]) \
7    depend(out:B[omp_spread_start:omp_spread_size])
8
9  #pragma omp target spread \
10   devices(1,3) \
11   spread_schedule(static, 10) \
12   nowait \
13   map(B[omp_spread_start:omp_spread_size]) \
14   depend(in:B[omp_spread_start:omp_spread_size])
15 for(...)
```

Recall that at the moment and by default, the usage of the **nowait** clause with this new set of directives makes the kernels/transfers run asynchronously until the next barrier, which is not implied in the mentioned **target spread** constructs. If synchronization is needed right after the construct, **taskwait** or **taskgroup** must be employed. A future design choice would be to create an implicit **taskgroup** for the **target spread** directives that forces synchronization at the end of the construct, but conveys also the implementation of the support for the **nogroup** clause, in case the programmer needs to cancel such barrier. However, having a **nowait nogroup** clause combination could be misleading.

We believe there is room for developing more static scheduling strategies, for example, one that allows irregular chunk sizes. Dynamic scheduling is also an important issue that must be addressed in order to mitigate the slowdown cause by load imbalance. Once they are implemented, we will integrate them into the syntax of the **target spread** data transfer directives via the **spread_schedule** clause.

The support for reduction clauses among devices would facilitate even more the implementation of complex algorithms that perform this kind of operations.

From the side of the experimentation, research has to be done on problems where the computation dominates the execution time over the data transfers, in order to see if a double buffering implementation performs better. Another

relevant kind of problems are those where the load balancing is an issue, in order to evaluate how poorly the static round-robin schedule performs, and what are the characteristics that a dynamic schedule should have.

References

[1] T. Shimizu, "Supercomputer Fugaku: Co-designed with application developers/researchers," in *2020 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pp. 1–4, 2020.

[2] TOP500, "TOP500 Supercomputers 2021," 2021. https://www.top500.org/lists/top500/2021/11/ (visited 2022-01-21).

[3] F. Gagliardi, M. Moreto, M. Olivieri, and M. Valero, "The international race towards exascale in europe," *CCF Transactions on High Performance Computing*, vol. 1, 04 2019.

[4] OpenMP Architecture Review Board, "OpenMP Application Program Interface version 4.0," 2013. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf (visited 2022-01-20).

[5] C. Shen, X. Tian, D. Khaldi, and B. Chapman, "Assessing One-to-One Parallelism Levels Mapping for OpenMP Offloading to GPUs," in *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'17, (New York, NY, USA), p. 68–73, Association for Computing Machinery, 2017.

[6] R. Torres, V. Kale, A. Malik, T. Scogland, R. Ferrer, and B. Chapman, "Support in OpenMP for Multi-GPU Parallelism." SC21 Research Poster, Nov. 2021. sc21.supercomputing.org/proceedings/tech_poster/poster_files/rpost111s2-file3.pdf.

[7] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar, "A Practical Approach to DOACROSS Parallelization," in *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, (Berlin, Heidelberg), p. 219–231, Springer-Verlag, 2012.

[8] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar, "Expressing DOACROSS Loop Dependences in OpenMP," in *OpenMP in the Era of Low Power Devices and Accelerators - 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings* (A. P. Rendell, B. M. Chapman, and M. S. Müller, eds.), vol. 8122 of *Lecture Notes in Computer Science*, pp. 30–44, Springer, 2013.

[9] M. Maroñas, X. Teruel, and V. Beltran, "OpenMP Taskloop Dependences," in *OpenMP: Enabling Massive Node-Level Parallelism - 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14-16, 2021, Proceedings* (S. McIntosh-Smith, B. R. de Supinski, and J. Klinkenberg, eds.), vol. 12870 of *Lecture Notes in Computer Science*, pp. 50–64, Springer, 2021.

[10] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[11] Message Passing Interface Forum, "A message-passing interface standard version 3.0," 2012.

[12] 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*, OOPSLA '05, (New York, NY, USA), p. 519–538, Association for Computing Machinery, 2005.

[13] Cray Inc., "Chapel Language Specification. Version 0.984," 2017.

[14] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-Defined Distributions and Layouts in Chapel: Philosophy and Framework," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, (USA), p. 12, USENIX Association, 2010.

[15] T. A. El-Ghazawi, W. W. Carlson, T. L. Sterling, and K. A. Yelick, *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[16] V. Kale, W. Lu, A. Curtis, A. M. Malik, B. M. Chapman, and O. R. Hernandez, "Toward Supporting Multi-GPU Targets via Taskloop and User-Defined Schedules," in *OpenMP: Portable Multi-Level Parallelism on Modern Systems - 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22-24, 2020, Proceedings* (K. Milfeld, B. R. de Supinski, L. Koesterke, and J. Klinkenberg, eds.), vol. 12295 of *Lecture Notes in Computer Science*, pp. 295–309, Springer, 2020.

[17] K. Matsumura, M. Sato, T. Boku, A. Podobas, and S. Matsuoka, "MACC: An OpenACC Transpiler for Automatic Multi-GPU Use"," in *Supercomputing Frontiers* (R. Yokota and W. Wu, eds.), (Cham), pp. 109–127, Springer International Publishing, 2018.

[18] R. Xu, S. Chandrasekaran, and B. Chapman, "Exploring programming multi-GPUS using OpenMP and OpenACC-based hybrid model," *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*, 05 2013.

[19] G. Ozen, S. Mateo, E. Ayguadé, J. Labarta, and J. Beyer, "Multiple Target Task Sharing Support for the OpenMP Accelerator Model," in *OpenMP: Memory, Devices, and Tasks* (N. Maruyama, B. R. de Supinski, and M. Wahib, eds.), (Cham), pp. 268–280, Springer International Publishing, 2016.

[20] G. Ozen, E. Ayguadé, and J. Labarta, "On the Roles of the Programmer, the Compiler and the Runtime System When Programming Accelerators in OpenMP," in *IWOMP 2014. Lecture Notes in Computer Science*, vol. 8766, 09 2014.

[21] Y. Yan, J. Liu, K. W. Cameron, and M. Umar, "HOMP: Automated Distribution of Parallel Loops and Data in Highly Parallel Accelerator-Based Systems," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 788–798, 2017.

[22] Y. Yan, P.-H. Lin, C. Liao, B. R. de Supinski, and D. J. Quinlan, "Supporting multiple accelerators in high-level programming models," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, (New York, NY, USA), p. 170–180, Association for Computing Machinery, 2015.

[23] M. Nakao, H. Murai, and M. Sato, "Multi-accelerator extension in openmp based on pgas model," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2019, (New York, NY, USA), p. 18–25, Association for Computing Machinery, 2019.