Received: 2 May 2023 | Revised: 20 June 2023 | Accepted: 24 July 2023

DOI: 10.1002/cpe.7885

RESEARCH ARTICLE

WILEY

Compute units in OpenMP: Extensions for heterogeneous parallel programming

Marc Gonzàlez-Tallada | Enric Morancho

Computer Architecture Department, Universitat Politècnica de Catalunya–BarcelonaTech, Barcelona, Spain

Correspondence
Marc Gonzàlez-Tallada, Computer Architecture Department, Universitat Politècnica de Catalunya–BarcelonaTech, Campus Nord, Despatx C6-E207, Jordi Girona 1-3, 08034 Barcelona, Spain.
Email: marc@ac.upc.edu

Funding information
Ministerio de Ciencia e Innovación, Grant/Award Number: PID2019-107255GB

Summary

This article evaluates the current support for heterogeneous OpenMP 5.2 applications regarding the simultaneous activation of host and device computing units (e.g., CPUs, GPUs, or FPGAs). The article identifies limitations in the current OpenMP specification and describes the design and implementation of novel OpenMP extensions and runtime support for heterogeneous parallel programming. The *Compute Unit* (CUs) abstraction is introduced in the OpenMP programming model. The *Compute Unit* abstraction is defined in terms of an aggregation of computing elements (e.g., CPUs, GPUs, FPGAs). On top of CUs, the article describes dynamic work sharing constructs and schedulers that address the inherent differences in compute power of host and device CUs. New constructs and the corresponding runtime support are described for the new abstractions. The article evaluates the case of a hybrid multilevel parallelization of the NPB-MZ benchmark suite. The implementation exploits both coarse-grain and fine-grain parallelism, mapped to CUs of different nature (GPUs and CPUs). All CUs are activated using the new extensions and runtime support. We compare hybrid and nonhybrid executions under two state-of-the-art work-distribution schemes (Static and Dynamic Task schedulers). On a computing node composed of one AMD EPYC 7742 @ 2.250GHz (64 cores and 2 threads/core, totalling 128 threads per node) and 2× GPU AMD Radeon Instinct MI50 with 32GB, hybrid executions present speedups from 1.08× up to 3.18× with respect to a nonhybrid GPU implementation, depending on the number of activated CUs.

KEYWORDS
GPUs, heterogeneous computing, OpenMP, work distribution

1 | INTRODUCTION

OpenMP standard has evolved toward heterogeneous parallel computing architectures by including new functionalities and language constructs that mainly address the specification of device computation and offloading (e.g., *target* directive) and device data management (e.g., *data* and *map* clauses). For the current OpenMP specification (OpenMP 5.2), the existing support mainly targets the activation of the devices from a pure OpenMP program specification. That is, OpenMP has tried to incorporate all the necessary language features so that device computations can be specified using just OpenMP directives, minimizing the usage of manual code transformations. Before that, programmers ported their applications to hybrid architectures combining device-oriented programming models like OpenCL or OpenACC, and also using external frameworks in the form of runtime systems like CUDA or HIP. But no matter if the strategy corresponds to a pure OpenMP specification or to a port mixing several programming paradigms, there is still one open aspect not yet covered by the OpenMP specification. OpenMP has been proven effective for activating the

host computing units (e.g., CPUs) and the device computing units (e.g., GPUs, FPGAs) but always in a separate manner. In OpenMP multi-GPU applications, programmers manually code a hybrid strategy where threads are dedicated to control the devices. The host CPUs are just used as mere execution flows to offload computations and orchestrate data transfers between devices. If host CPUs have to participate in the execution, then OpenMP has not yet incorporated the necessary support to address the following aspects.

First, work-sharing constructs for loop execution do not support work distribution schemes that simultaneously activate all available device and host compute resources (e.g., GPUs and CPUs). OpenMP has constructs to specify parallelism at thread level and it also includes constructs to specify and offload device computations. But there is no support to merge both using the work distribution directives. OpenMP misses the *Computing Unit* abstraction that unifies both host and device computing elements, accompanied with appropriate work distribution schemes.

Second, devices and host expose two physically separated address spaces, unified under a single view from the application. This requires different mechanisms to synchronize and maintain memory consistency, resulting in a distributed shared address space. In general, and from past experience, distributed shared address spaces require data placement guidance from the programmer to achieve acceptable performance levels (e.g., Unified Parallel C,[1,2] HPF[3]). OpenMP currently does support the specification of such guidance. The existing support is focused on the explicit mirroring of data structures between host and devices, and memory allocation and data transfers. Programmers have to manually deploy a data placement strategy prior the computations are distributed among the computing resources (e.g., CPUs, GPUs, FPGAs). And given the inherent entanglement between data placement and work distribution, this results in significant limitations for work distribution. This is an essential limitation as the differences in compute power of devices and host computing units require dynamic and adaptive work schedulers to avoid work unbalance.

The main contribution of this article is the introduction of the *Compute Unit* in the OpenMP programming model. *Compute Units* (CUs) are defined as an aggregation of physical computing resources such as CPUs and GPUs or any other type. We introduce dynamic work distribution mechanisms not limited by the data placement and that are able to solve the potential work unbalance of the CUs. We have evaluated a hybrid implementation of the NPB-MZ benchmark suite where the new language constructs and runtime support have been proven effective both at the programming and at the performance levels. The proposed new constructs allow a hybrid multilevel paralellization of the suite, exposing both coarse and fine grain parallelisms that are simultaneously exploited by both host CUs (CPUs) and device CUs (GPUs). In a node composed of one AMD EPYC 7742 @ 2.250GHz (64 cores and 2 threads/core, totalling 128 threads per node) and 2× GPU AMD Radeon Instinct MI50 with 32GB, we observe speedups from 1.08× up to 3.18× for hybrid executions present with respect to a nonhybrid GPU implementation, depending on the number of activated CUs.

This article is organized as follows: Section 2 describes the NPB-MZ benchmark suite and shows its parallelism sources. Section 3 details the design and implementation of the extensions and their runtime support for OpenMP hybrid applications. Section 4 shows how a use case of the new extensions to achieve a hybrid implementation of the NPB-MZ benchmark suite. Section 5 evaluates the performance of our hybrid implementation. Section 6 discusses related works. Section 7 concludes the article.

2 | BENCHMARK CHARACTERIZATION

2.1 | NAS parallel benchmarks

The NAS parallel benchmarks (NPB)[4] suite provides the implementation of benchmarks for several programming languages (Fortran, C, …) and parallel paradigms (OpenMP, MPI, …). The suite defines several input classes (B, C, …) varying input size. The original NPB benchmark suite consisted of eight benchmarks: five kernels and three pseudo-applications. We focus on the three pseudo applications: BT, LU, and SP (Block Tri-diagonal, Lower-Upper Gauss-Seidel, and Scalar Penta-diagonal solvers). BT, LU and SP traverse a 3D volume (Figure 1A) to solve the Navier-Stokes equations. Figure 2A shows the flow graph of NPB benchmarks. The main loop, the time-step loop, applies a solver every iteration.

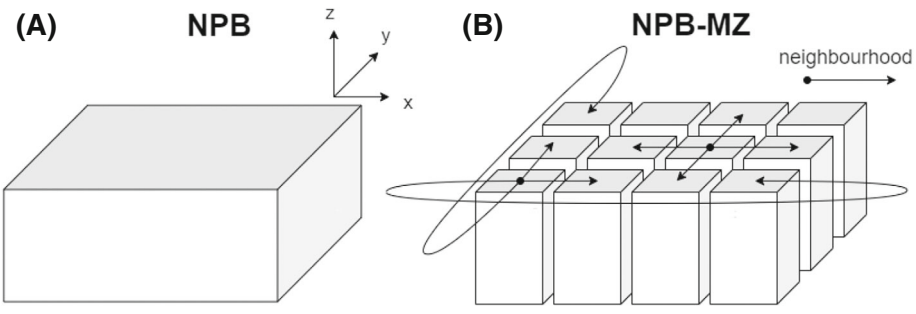

FIGURE 1 3D Volume: (A) NPB, (B) NPB-MZ, example 4 × 3 tiling into 12 zones and neighborhood relation between zones.

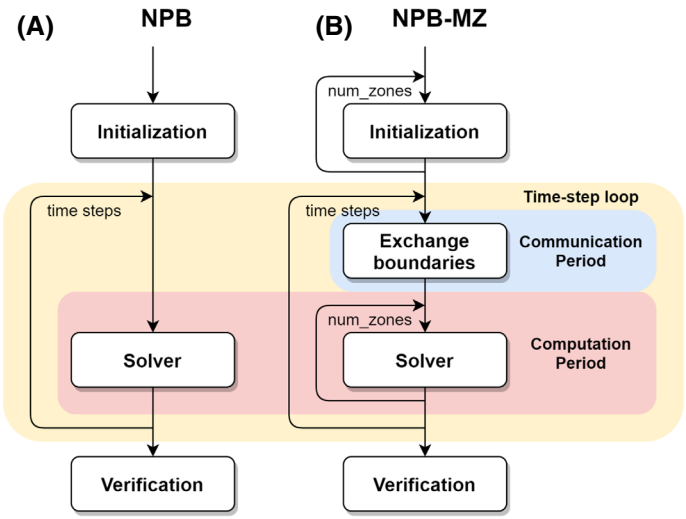

FIGURE 2 Flow graphs: (A) NPB and (B) NPB-MZ.

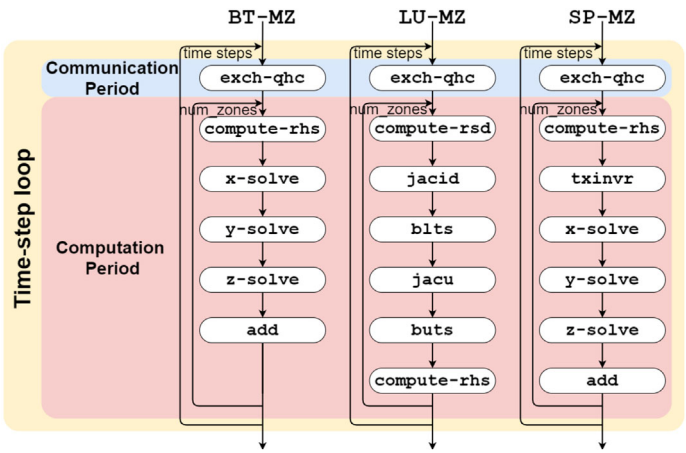

FIGURE 3 Time-step loop of NPB-MZ benchmarks. Sequence of procedures called at each iteration of the time-step loop.

The Multi-Zone NPB (NPB-MZ)[5] re-implement NPB to expose a coarse level of parallelism. The 3D input volume is tiled through both x and y dimensions producing a grid of prisms (*zones*, Figure 1B) that can be processed in parallel.

However, the correct values of the border faces of each zone also depend on the values of the border faces of its neighbor zones. So, between two iterations of the time-step loop, a *exchange-boundaries* procedure must update the border faces of all zones.

Figure 2B shows the flow graph of NPB-MZ benchmarks. Each time-step iteration calls the exchange-boundary procedure before applying the solver to all zones (potentially in parallel). Consequently, each time-step iteration can be divided into two periods: the *Communication Period*, devoted to transfer zone faces, and the *Computation Period*, devoted to update zones. Figure 3 details the sequence of high-level procedures called at each iteration.

Table 1 characterizes NPB-MZ benchmarks. For each input class, the overall size, the number of zones, the zone size and the number of time steps are detailed. Notice some differences that will be relevant to understand the performance of our strategy in every benchmark:

- In LU-MZ, the number of zones is always 16, independently of the input class. For each input class, zone sizes are uniform.
- In SP-MZ, the larger the input class, the larger the number of zones. Like in LU, zone sizes are uniform at each input class; however, SP-MZ zones are smaller than LU-MZ zones.
- In BT-MZ, like in SP-MZ, the larger the input class, the larger the number of zones. However, zone sizes are not uniform; for each input class, the ratio between the size of the biggest zone and the size of the smallest zone is about 20×.

TABLE 1 Characterization of NPB-MZ benchmarks: overall size and, for each benchmark, number of zones, zone size and number of time steps.

Input	3D volume	Memory	Num. zones (x × y)		Zone size (points per zone)			Time steps		
Class	x × y × z (points)	(GB)	LU	SP & BT	LU	SP	BT	LU	SP	BT
B	304 × 208 × 17	≈ 0.2	4 × 4	8 × 8	67,184	16,786	From 2992 to 59,976	250	400	200
C	480 × 320 × 28	≈ 0.8	4 × 4	16 × 16	268,800	16,800	From 2912 to 60,648	250	400	200
D	1632 × 1216 × 34	≈ 13.0	4 × 4	32 × 32	4,217,088	65,892	From 11,968 to 243,236	300	500	250
E	4224 × 3456 × 92	≈ 250.0	4 × 4	64 × 64	83,939,328	327,888	From 59,248 to 1,203,452	300	500	250

```
for (step = 0; step < num_steps-1; step++) {
  // Border exchange
  exch_qbc(...);
  // Zone processing (sequentially)
  for (zone = 0; zone < num_zones; zone++) {
    comp_stage_1 (...);
    comp_stage_2 (...);
    ...
    comp_stage_N (...);
  } // Zone loop
} // Time step loop

void comp_stage_N(...) {
  // Loop nest
  for (k = 0; k < zdim; k++)
    for (j = 0; j < ydim; j++)
      for (i = 0; i < xdim; i++) {
        // Matrix-based computation
      }
  // Loop nest
  for (k = 0; k < zdim; k++)
    for (j = 0; j < ydim; j++)
      for (i = 0; i < xdim; i++) {
        // Matrix-based computation
      }
  ...
}
```

FIGURE 4 Time-step loop for the NPB-MZ suite. Border elements of zones are computed in function *exch-qbc*. Zones are processed sequentially. Each zone processing is organized in a sequence of phases that contain several loop nests that implement matrix-based computations.

2.2 | Sources of parallelism

The NPB-MZ suite exposes several levels of parallelism. Its main difference with respect to NPB is the exposure of a new parallelism level, the *inter-zone* parallelism. It can be exploited in the *Computation Period* by the parallel processing of zones through several CU's (i.e., CPU's and GPU's). Moreover, NPB-MZ also exposes an *intra-zone* parallelism in both periods while processing each individual zone. It can be exploited by several parallelization techniques (i.e., multithreading, vectorization, and porting to GPU).

2.2.1 | Computation period: Inter-zone parallelism

Figure 4 shows the skeleton of the time-step loop of NPB-MZ applications. The *Computation Period* is implemented by a loop that traverses the zones and processes them. As processing a zone is independent of processing the other zones, exploiting the *inter-zone* parallelism implies the parallelization of this loop.

```
for (zone = 0; zone < num_zones; zone++) {
   east = adjacency_east[zone];
   north = adjacency_north[zone];

   // Pack border elements into temporary buffers
   copy_face(tmpEast,  mesh[east],  "IN", ...);
   copy_face(tmpNorth, mesh[north], "IN", ...);

   compute_border(mesh[zone], tmpEast, tmpNorth, ...);
   // Unpack temporary buffers
   copy_face(tmpEast,  mesh[east],  "OUT", ...);
   copy_face(tmpNorth, mesh[north], "OUT", ...);
} // Zone loop (sequential)

void copy_face(zoneid_t id, data_t *zone,
               string Direction, ...) {
  ...
  for (k = 0; k < dim; k++)
    for (j = 0; j < dim; j++) {
        // Matrix-based computation
      }
  ...
}

void compute_border(...) {
  for (k = 0; k < dim; k++)
    for (j = 0; j < dim; j++) {
        // Matrix-based computation
      }
}
```

FIGURE 5 Computation of border elements for each zone. Zones are processed sequentially and each zone processing is organized in a sequence of memcpy operations (copy_face calls) that pack/unpack border elements to temporary buffers. Then these are exchanged between adjacent zones (compute_border calls).

2.2.2 | Computation period: Intra-zone parallelism

Figure 4 depicts that each computational phase calls a subroutine with several loop nests that implement the computation for one zone and a phase. The *intra-zone* parallelism corresponds to the exploitation of the parallelism exposed by these loop nests. These can be annotated with OpenMP directives or coded as device kernels.

2.2.3 | Communication period: Intra-zone Parallelism

Figure 5 shows the code skeleton for exchanging boundary values. In this period, zones are processed sequentially, so no inter-zone parallelism exists. However, each zone can be processed in parallel by exploiting its *intra-zone* parallelism. The computations are organized in the form of *memcpy* operations that pack/unpack border elements into temporary buffers (copy_face calls in Figure 5) and then these are exchanged between adjacent zones (compute_border function calls). All of these computations are organized in the form of parallel loop nests that can be annotated with OpenMP directives or transformed to device kernels.

2.3 | Hybrid parallelization

The challenges to achieve a hybrid parallel implementation of an OpenMP application correspond to the introduction of novel support for abstractions like the CU and new work distribution schemes to address the inherent difference in computational power between CUs.

To understand these challenges, Figures 6 and 7 show the parallel host and device code schemes for the NPB-MZ suite. The inter-zone parallelism is easily activated through the annotation with a *parallel for* directive of the loop that implements the zone processing. Intra-zone parallelism

```
for (step = 0; step < num_steps-1; step++) {
  // Border exchange
  exch_qbc(...);

  // Zone processing
#pragma omp parallel for
  for (zone=0; zone<num_zones; zone++) {
    comp_stage_1 (...);
    comp_stage_2 (...);
    ...
    comp_stage_N (...);
  } // Zone loop
} // Time step loop

void comp_stage_N(...) {
  // OpenMP CPU code annotation
#pragma omp parallel for
  for (k = 0; k < zdim; k++)
    for (j = 0; j < ydim; j++)
      for (i = 0; i < xdim; i++) {
        // Matrix-based computation
      }
  ...
}
```

FIGURE 6 Multilevel OpenMP parallel code. Outermost level (e.g., inter-zone parallelism) is activated through an OpenMP directive that parallelizes the loop that processes all zones. Innermost level of parallelism is activated through OpenMP annotations for the loop nests that implement the compute stages of one zone processing.

can be activated for host execution with additional OpenMP directives that parallelize loop nests within the different stages of one zone processing. For device execution, one option is to use the the OpenMP *target* and *teams* constructs to annotate these loop nests. Another option is manually transform the code porting it to CUDA or ROC frameworks (e.g., define computational kernels and introduce library calls to offload the computation and perform the necessary data transfers between host and devices).

But if we want to activate simultaneously both device and host CUs, the codes in Figures 6 and 7 need additional support not available in OpenMP. Notice that we do not refer to the OpenMP support to indicate the definition of different versions of the code. We refer to the fact that once parallelism is being executed, OpenMP lacks support to define the thread behavior when nested parallelism arises. The *declare variant*

```
__global__ kernel_1_comp_stage_N(...) {
  k = f_k(block grid);
  j = f_j(block grid);
  i = f_i(block grid);
  // Matrix-based computation
  ...
}
__global__ kernel_2_comp_stage_N(...) {
  k = f_k(block grid);
  j = f_j(block grid);
  i = f_i(block grid);
  // Matrix-based computation
  ...
}
void comp_stage_N(...) {
  kernel_1_comp_stage_N<<<grid, block, ... >>>(...);
  kernel_2_comp_stage_N<<<grid, block, ... >>>(...);
  ...
}
```

FIGURE 7 Device parallel code for innermost parallelism execution. Loop nests corresponding to compute stages of one zone processing have been transformed into device kernels.

```
void comp_stage_N(...) {
  // OpenMP GPU code annotation for kernel 1
#pragma omp target map (from: ... , to:...) teams
#pragma distribute
  for (k = 0; k < zdim; k++)
#pragma omp parallel for
    for (j = 0; j < ydim; j++)
      for (i = 0; i < xdim; i++) {
        // Matrix-based computation
      }

  // OpenMP GPU code annotation for kernel 2
#pragma omp target map (from: ... , to:...) teams
#pragma distribute
  for (k = 0; k < zdim; k++)
#pragma omp parallel for
    for (j = 0; j < ydim; j++)
      for (i = 0; i < xdim; i++) {
        // Matrix-based compuation
      }
   ...
}
```

FIGURE 8 OpenMP device parallel code for innermost parallelism execution. Loop nests corresponding to compute stages of one zone processing have been annotated with *target* and *teams* directives.

or *meta-directives* constructs help to generate code versions, but not the definition of the conditions to make one thread to take a particular path within the execution of nested parallelism. Previous work has addressed the same limitation,[6] but the proposal in this article neatly introduces the definition of heterogeneous parallel regions with the necessary runtime support to control their execution from the programmer perspective. Even more, OpenMP also has to consider the support to applications where the device kernels have been already provided using other programming models like CUDA or HIP. In this regard, *declare variant* or *meta-directives* cannot be used. Therefore, OpenMP misses the support to combine the two programming models to guide the OpenMP threads toward the appropriate code version. In the MZ-NPB case, threads executing the inter-zone parallelism have to be associated to host or device CUs so that their execution can be diverged to the corresponding versions of the code (e.g., for GPU-based CUs, this corresponds to the codes in Figures 7 and 8 with *target* versions of the loop nests in Figure 6; for CPU-based CUs this corresponds to the code in Figure 6). This support is not available in the current specification of OpenMP 5.2 and neither exists in the CUDA or HIP runtime specifications. Even if this is manually coded, there is no support either at the programming language or at the runtime level to diverge the thread execution toward host or device code. For intra-zone parallelism and host CUs, there are no mechanisms to set how many threads and where they should run (e.g., onto which cores) when executing the innermost parallelism in loops in Figure 6. For distributing work, there are not alternative schedulings that are able to adapt to the inherent differences in compute power of host and device computing units.

In conclusion, two main novel contributions become essential. First, allowing the programmer to specify a parallel region where CUs of different type will participate, including the specification of how many of each type. This implies that threads need to be associated to computing units according to new rules. Second, given the differences in compute power between host and devices, new scheduling schemes are necessary to achieve a balanced execution.

Programmers of heterogeneous architectures manage different address spaces, so memory allocation and placement requires some guidance from the programmer. Then, at the programming model level, we need some explicit support to indicate when to allocate memory, where to allocate and, if necessary, to change the placement of that memory. In the context of Figures 6–8, we need to ensure that memory allocation and placement has happened prior any work is assigned to the OpenMP threads that will be mapped to host or device execution. But this is highly entangled with the work distribution applied among the hybrid execution flows. As with the CU and work distribution abstractions, we need additional runtime support to make possible the appropriate policies for memory placement according to the work distribution schemes.

OpenMP defines a set of new directives to allow the programmer to offload a computation to a device. For OpenMP there are two problems to be solved. The first one corresponds to the actual translation of the computation described in the program to the binary format of the target device. The directives described in this section do not consider this problem. For OpenMP 5.2, this is well solved with the directives and clauses associated to the *target*, *teams* constructs. The second one refers to the data sharing between devices and host address spaces. Optional clauses

to these constructs make possible to specify what are the mirroring data structures in the devices that map on to host data structures. Also, it is possible to indicate if data transfers are needed between the host and the device for specific data structures.

OpenMP is a host-centric programming model. It assumes that, in the absence of programmer indications, the host address space always has to be updated to mirror the device-memory state. That is, the host space contains the latest and valid version of any mirrored data structure. Although that, OpenMP supports *map* clauses to change this behavior. We propose two new abstractions. These correspond to new OpenMP extensions that allow the programmer to associate threads to CUs, either corresponding to devices and host, and new constructs to allow the programmer to guide the data distribution and placement within a hybrid architecture composed of different address spaces. Finally, we describe a methodology to use the new constructs to implement hybrid parallelizations, along with the necessary runtime and compiler support to implement it. Next section describes the new OpenMP constructs, their runtime support and examples applied to the NPB-MZ suite.

3 | OPENMP HETEROGENEITY EXTENSIONS

Defining new OpenMP extensions for heterogeneity implies introducing new language constructs and modifying its execution model and its corresponding runtime support. The following subsections describe the modifications and extensions proposed to each of these components of the OpenMP programming model.

3.1 | Execution model

3.1.1 | Computing units (CUs)

Computing units can be either host-based or device-based. For host-based CUs, they correspond to an aggregation of one or more CPUs under a particular organization. For instance, a CU can be formed with a pair of CPUs giving an organization that we identify as 1×2: 1 stands for one CU, the 2 identifies the number of CPUs in the CU. Similarly, CUs can be defined in the form of 1×4, 1×8 or whatever the CPU aggregation.

Device-based CUs are defined as an aggregation of one CPU and one or more devices. Devices can be GPUs or FPGAs, for ease of description, we assume the case for GPUs: 1 + G configuration, where G stands for the number of GPUs in the CU. One CPU is bounded to the GPUs in order to orchestrate all actions over them: memory allocation, data transfers and computation offloading. The usual configurations for hybrid executions are defined in the following form: $(N \times M + D \times G)$ where N stands for the number of CPU-based CUs executing with M CPUs, and D stands for the number of GPU-based CUs executing with G GPUs. Notice that a total number of $N \times M + D$ threads are necessary to execute in a hybrid fashion.

CUs are identified with integer numbers that range from 0 up to NCUS-1, where the NCUS correspond to the number of executing CUs. This is totally in accordance with how OpenMP identifies the threads that execute within the same parallel region. The OpenMP thread name space matches exactly the CU name space. Within the extended execution model, CPU-based CUs are identified from 0 up to NCPUS-1 where NCPUs is the number of CPU-based CUs. GPU-based CUs are identified by numbers in the range [NCPUS … NCUS-1]. The CU name space is used to implement work distribution schemes like the OpenMP programming model does.

Figure 9 shows an 8-CU hybrid system, with 4 CPU-based CUs composed each one of 4 CPUs and 4 GPU-based CUs composed of 1 CPU and 1 GPU. This configuration needs a total number of 20 physical CPUs. One possible way to achieve the CPU aggregation is setting the two OpenMP environment variables that control affinity and thread binding to CPUs. `OMP_PLACES` lists aggregation of physical CPU identifiers where threads are bounded to. These aggregations are named *places*. `OMP_PROC_BIND` describes how to interpret the content of `OMP_PLACES` across nested levels of parallelism. The proposed execution model for CU definition is in total accordance with the current OpenMP specification regarding the setting of thread affinity.

3.2 | Language extensions

3.2.1 | Computing units

OpenMP extensions for heterogeneity require new constructs to specify the quantity and nature of CUs. The main changes affect the original *teams* construct and its combined forms. The clause *num_teams* is extended so that now its main argument is a list of computing unit descriptors (CUD). Each CUD specifies the number and type of the CUs. A CUD has the syntax of *int-expr*::*CU-type*::*int-expr* where *CU-type* is one of *CPU*, *GPU* or *FPGA*,

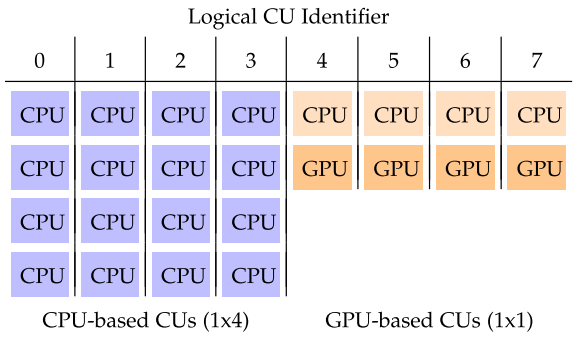

FIGURE 9 Example of a 8 CU organization: 4 CPU-based CUs composed each one of 4 CPUs, and 4 GPU-based CUs composed of 1 CPU and 1 GPU. This configuration needs 20 physical CPUs.

```
#pragma omp teams [ clause_list ]
      structured-block

#pragma omp teams loop [ clause_list ]
      loop-nest

#pragma omp teams distribute [ clause_list ]
      loop-nest
```

where `clause_list` is a list of one or more `clause`

where `clause` is one of the following

```
num_teams( [lower_bound:] upper_bound )
num_teams( CU_descriptor [, CU_descriptor] ... )
```

where `CU_descriptor` is

```
integer-expression : CU_family : integer-expression
```

and `CU_family` is one of `CPU`, `GPU` or `FPGA`

Other original clauses:

```
thread_limit( integer_expression )
default( data_sharing:attribute )
private( list )
firstprivate( list )
shared( list )
reduction( [default ,] reduction_identifier : list )
allocate( [allocator :] list )
```

FIGURE 10 C/C++ syntax for the *teams* new construct.

and *int-expr* is an integer expression. The keywords identifying the possible computing units can be easily extended if new type of computing units appear. Figure 10 shows the syntax for the *teams* construct in all its combined forms.

The semantic associated to:

```
#pragma omp teams num_teams( 2::CPU::4, 2::GPU::1 )
```

is the definition of a parallel region executed by 4 CU's. Four threads, one per CU, orchestrate the execution. Two of the threads correspond to two CUs composed of 4 CPUs each; any work aroused along their execution it will be assigned to any or all of the 4 CPUs assigned to the CUs.

TABLE 2 CU runtime primitives.

Signature		Description
`unsigned int`	`omp_get_num_cus()`	Get the current number of active CUs
`unsigned int`	`omp_get_num_gpus()`	Get the number of GPU-based CUs
`unsigned int`	`omp_get_num_cpus()`	Get the number of CPU-based CUs
`unsigned int`	`omp_get_num_threads()`	Get the number of CPUs contained in the CPU-based CU associated to the invoking thread
`bool`	`omp_is_gpu()`	Check if invoking thread is associated to a GPU-based CU
`bool`	`omp_is_cpu()`	Check if invoking thread is associated to a CPU-based CU
`unsigned int`	`omp_get_gpu()`	Get the device ID for the CU associated to the invoking thread. Returns limits::max if the invoking thread is not associated to a GPU-based CU
`unsigned int`	`omp_get_cpu()`	Get the OpenMP thread ID for the CU associated to the invoking thread. Returns limits::max if the invoking thread is not associated to a CPU-based CU
`unsigned int`	`omp_get_cu_num()`	Get the CU ID, from 0 up to the number of CUs minus one
`void`	`omp_synchronize()`	Synchronize with the device corresponding to the CU associated to the invoking thread

The other two threads are associated to CUs composed of one GPU. Therefore, all the work they are assigned with will be offloaded to their corresponding GPU.

Only one clause of the *teams* construct generates a conflict with the *teams* extensions. Bottom part of Figure 10 lists those clauses, where data scoping clauses (e.g., *default*, *shared*, *private* and *firstprivate*) are not affected by the new extensions. Neither the *allocate* and *reduction* clauses interfere with the CU definition. The *thread_limit* imposes limits on the number of threads to be used in inner levels of parallelism of the *teams* construct. Clearly, this generates some conflict with the new extensions. To solve any possible conflict, if any CU is defined with more CPUs than the current limit for this clause, then the CU definition takes the limit as the number of CPUs contained in the CU.

3.3 | Runtime support

This subsection describes the necessary runtime support for the new extensions. The runtime primitives are organized according to their functionality for CU management.

3.3.1 | Computing units

Table 2 lists the new OpenMP runtime primitives proposed to introduce the CU abstraction within the OpenMP programming model. The primitives give support to the programmer to query the runtime system about the parameters of the hybrid execution: how many CUs are executing, how many of them correspond to CPU-based CUs or to GPU-based CUs. Also, there are primitives to get and set the CU identifier, and to translate from the CU identifier to the device identifier the thread is assigned to, and from the CU identifier to the OpenMP thread identifier in the case of CPU-based CUs. For GPU-based CUs there is a specific primitive to synchronize with the associated device. Notice that this support could be easily extended for any other type of existing device (e.g., FPGAs).

4 | PROGRAMMING ASSESSMENT

This section evaluates the expressiveness and the programming support given by our proposal. We deploy a hybrid design and implementation of NPB-MZ benchmarks using our OpenMP extensions and runtime support. We attempt to evaluate the extensions in terms of ease of programming and the runtime support. We will show how the extensions solve the limitations described in Section 3.

4.1 | NPB-MZ multilevel hybrid parallelization

The multilevel parallelization is described in terms of *inter* and *intra* zone parallelism exposed in both the *Computation Period* and the *Communication Period* of each benchmark.

```
data_t *mesh[NUM_ZONES];

// Zone initialization
for (zone = 0; zone < num_zones; zone++) {
  register_zone ( mesh[zone], ... );
  ...
}

for (step = 0; step < num_steps-1; step++) {
  // Border exchange
  exch_qbc(...);
  // Zone processing
#pragma omp teams num_teams(4:CPU:4, 2:GPU:1)
#pragma omp distribute dist_schedule(...)
  for (zone = 0; zone < num_zones; zone++) {
    migrate_zone( mesh[zone] );
    comp_stage_1 (...);
    comp_stage_2 (...);
    ...
    comp_stage_N (...);
  } // Zone loop
} // Time-step loop
```

FIGURE 11 Parallel code for *Computation Period*. Definition of CUs using OpenMP extensions for thread teams and their correspondence to CUs.

4.1.1 | Computation period

Figure 11 shows how to use the proposed extensions in the *Computation Period* of NPB-MZ applications. The loop responsible for the zone processing has been annotated with the *teams* and *num_teams* constructs. We define a hybrid execution composed of 4 CPU-based CUs, each one with 4 CPUs. Additionally, 2 GPU-based CUs each one composed of one GPU. Work distribution is specified with the *distribute* and *dist_schedule* constructs. Regarding to the data placement used in each task, notice the introduction of the *register_zone* and *migrate_zone* calls. The first one informs the application runtime system to track the placement of the memory regions associated to each zone. In our implementation, this happens during the initialization phase, where zones are allocated. In Figure 11 only one *register* action is indicated, but each zone within the mesh is represented by several data structures that require the zone registering. Similarly, the introduction of the *migrate* action ensures that the zone data structures are allocated and migrated to the address space of the CU that computes the zone.

Figure 12 depicts the code for one stage. Notice the usage of the corresponding primitives to check whether the executing thread is associated to a host or to a device CU. For host CUs, the execution is diverged to loop nests annotated with loop level parallelism directives. These inner parallelism is exploited by as many threads as the result of the primitive *omp_get_num_cpus*. Thus, this innermost parallelism is deployed over the CPUs associated to the host CU. Notice the usage of the *num_threads* clause to specify the number of threads that execute the loop nest: the number of CPU associated to the CPU-based CU.

In case the thread corresponds to a device CU, then the execution is diverged to the invocation of the kernels that correspond to device versions of the original loop nests. Notice the translation from the device CU identifier to the actual device identifier (e.g., call to the *omp_get_gpu* primitive). Notice the usage of the *map* clause to generate the mirroring of data structures between device and host.

Figure 13 shows a possible implementation of the *teams* construct (Figure 11). The loop that traverses the zones have been substituted by a parallel region that is executed by as many threads as CUs take part in the hybrid execution. These threads execute a *while* loop with primitives for acquiring, executing and committing tasks. Notice that the *migrate* primitive ensures that the memory allocation is performed before processing a zone (e.g., task). In this case, the task identifier coincides with the zone index processed at each iteration of the *while* loop. For threads associated to a GPU-based CU, after the zone is processed, a synchronization primitive is invoked. The computation stages are executed by a host CU or a device CU according to the nature of the CU the thread is associated to. The scheduling object SCHED points to the desired scheduling for the hybrid execution of the *Computation Period*. In our implementation we support a STATIC scheduling, corresponding to the default and only supported scheduling in the current OpenMP specification. Given the inherent unbalance that this scheduling generates among the different types of the CUs, we have implemented variants of the STATIC and DYNAMIC schedulings; they are made available through the SCHED object that incorporates the methods *omp_get_task*, *omp_execute_task*, and *omp_commit_task*.

```
void comp_stage_N(...) {
  if (omp_is_cpu()) {  // CPU version
  // OpenMP CPU code annotation
#pragma omp parallel num_threads(omp_get_num_threads()) for
    for (k = 0; k < zdim; k++)
      for (j = 0; j < ydim; j++)
        for (i = 0; i < xdim; i++) {
          // Matrix-based computations
        }
     ...
  }
  else if (omp_is_gpu()) { // GPU version
    int Device = omp_get_gpu();
    omp_set_default_device(Device);
    // OpenMP GPU code annotation
#pragma omp target map (from: ... , to:...) teams
#pragma distribute
    for (k = 0; k < zdim; k++)
#pragma omp parallel for
      for (j = 0; j < ydim; j++)
        for (i = 0; i < xdim; i++) {
          // Matrix-based computation
        }
    ....
  }
}
```

FIGURE 12 Parallel code for stage processing, using OpenMP annotations. Runtime diverges the threads toward host or device execution.

```
#pragma omp parallel num_threads("number-of-teams")
{
  unsigned int task = SCHED->omp_get_task();
  while (task != NO_TASK) {
    migrate_zone(mesh[zone]);
    SCHED->omp_execute_task(task);
    // Zone processing
    zone = task;
    comp_stage_1 (...);
    comp_stage_2 (...);
    ...
    comp_stage_N (...);
    if (omp_is_gpu) omp_synchronize();
    SCHED->omp_commit_task(task);
    task = SCHED->omp_get_task();
  }// while
} // parallel
```

FIGURE 13 Code transformation for inter-zone parallelism. Original parallel loop is transformed into a parallel region containing a *while* loop. CU threads iterate and invoke the runtime primitives to acquire tasks and execute them. The scheduling is determined by the SCHED object.

4.1.2 | Communication period

Within the *Communication Period* there is no *inter-zone* parallelism. All zones are processed one after the other, exchanging and updating border values. The processing of a zone exposes some amount of parallelism exploited through fine-grain parallelism. For the *Communication Period* we have manually implemented, using the ROC framework, all kernel computations and data transfers. The main purpose is to show how the new runtime support in OpenMP can be mixed with existing device libraries such as Hip or Cuda. Figure 14 shows the code scheme for the *Communication Period* using the novel runtime extensions. Zone faces containing border elements are copied to temporary buffers (calls to *copy-face* procedure). Then, zone

placement is checked to perform the border exchange computation on a device or host CU accordingly. Notice the introduction of OpenMP runtime calls to check the zone placement. If a zone resides on a device address space, the border computation will be offloaded to the corresponding device. In this case, Figure 14 assumes the ROC framework, using a Hip call to set the device where to offload. The usage of data placement directives during the *Computation Period* allows the OpenMP runtime support to track the placement of zones. This simplifies the programming for the *Communication Period*.

The implementation of the *copy-face* procedure is more complex because buffering of border elements is conditioned by the fact that buffers and adjacent zones can reside in different address spaces. Figure 15 shows the code skeleton for this procedure. Notice the structure of *if* statements that covers the 4 possibilities: buffer resides on host/device, zone resides on host/device. Notice that for data transfer, Hip primitives appear to move

```
for (zone = 0; zone < NUM_ZONES; zone++) {
  east_zone = adjacency_east[zone]
  north_zone = adjacency_north[zone];
  zonePlct = omp_get_data_placement(mesh[zone]);
  copy_face(tmpEast[zonePlct], mesh[east_zone], "IN", ...);
  copy_face(tmpNorth[zonePlct], mesh[north_zone], "IN", ...);

  if (omp_is_device(zonePlct)) { // Device version
    HipSetDevice(zonePlct));
    gpu_compute_border(mesh[zone],
           tmpEast[zonePlct], tmpNorth[zonePlct]);
  }
  else if (omp_is_host(zonePlct)) { // Host version
    cpu_compute_border(mesh[zone],
           tmpEast[zonePlct], tmpNorth[zonePlct]);
  }
  copy_face(tmpEast[zonePlct], mesh[east_zone], "OUT", ...);
  copy_face(tmpNorth[zonePlct], mesh[north_zone], "OUT", ...);
}
```

FIGURE 14 Hybrid implementation of the exchange boundary computation. Host and device versions of the compute border computation are introduced. Border computation happens where the zone resides.

```
void copy_face(void* Buffer, void* Zone, string Dir, ...) {
  zonePlct = omp_get_data_placement(mesh[Zone]);
  bufferPlct = omp_get_data_placement(Buffer);
  if (Dir == "IN") {
    if (omp_is_device(zonePlct)) { // From device ...
      HipSetDevice(zonePlct);
      gpu_copy_face<<<grid, block, ...>>>(...);
      if (omp_is_device(bufferPlct)) { // ... to device
        HipMemCpyPeer(Buffer, ...,);
      }
      else if (omp_is_host(bufferPlct)) { // ... to host
        HipMemCpy(Buffer, ...,HipDeviceToHost);
      }
    }
    if (omp_is_host(zonePlct)) { // From host ...
      cpu_copy_face(...);
      if (omp_is_device(bufferPlct)) { // ... to device
        HipSetDevice(bufferPlct);
        HipMemCpy(Buffer, ...,HipHostToDevice);
      }
      else if (omp_is_host(bufferPlct)) { // ... to host
        // Nothing
      }
    }
  } // if Dir == "IN"
  else if (Dir == "OUT") {
     ...
  } //if Dir == "OUT"
}
```

FIGURE 15 Hybrid implementation of the copy face computation. Placement checks are introduced for buffers and zones. Host and device versions of the copy face procedure are needed to cover all 4 possibilities: buffer resides on host/device, zone resides on host/device.

TABLE 3 Average task time (ms) in several CU configurations.

Conf.	SP-MZ	BT-MZ	Conf.	LU-MZ
1-GPU	0, 83	2, 96	1-GPU	38, 44
1 × 1	8, 70	31, 11	1 × 1	1653, 38
16 × 1	9, 00	64, 75	1 × 16	258, 94
32 × 1	11, 25	69, 19	1 × 32	98, 13
48 × 1	18, 61	87, 66	1 × 48	88, 44
64 × 1	36, 38	102, 50	1 × 64	38, 44

the data across the host and devices. This communication arises according to the memory footprint the application determines, which is totally conditioned by the scheduling applied during the *Computation Period*. Similarly as in the case of the border computation, the usage of data placement directives during the *Computation Period* allows the OpenMP runtime support to track the placement of zones. This simplifies the programming for the *Communication Period*. The effect of this communication is evaluated in Section 5.

5 | EVALUATION

This Section describes the overall performance for hybrid executions of NPB-MZ benchmarks. We have manually coded the benchmarks using OpenMP 5.2 and ROCM-3.5.0 and applied the transformation described in Section 3 to activate the CPUs and GPUs simultaneously. We have implemented the necessary runtime support in the form of 2 libraries: *libCU-rtl* and *libPLACEMENT-rtl*. These components were used next to the OpenMP 5.2 and ROCM-3.5.0 native libraries to support the features described in Section 3. Besides, we have implemented a dynamic scheduling with memorization[7] and a profile-based scheduling, used to compare against the current default and unique available scheduling for the *teams* construct in OpenMP 5.2.

All experiments have been developed in a system composed of AMD EPYC 7742 @ 2.250GHz (64 cores and 2 threads/core, totalling 128 threads per node) and 2× GPU AMD Radeon Instinct MI50 with 32GB. All applications have been coded combining OpenMP 5.2 and the ROCM-3.5.0 framework and compiled with *llvm 12*. The CPU code has been implemented combining a C++ NPB-MZ implementation[8] and the original NPB-MZ Fortran implementation[5] to generate a version compatible with the ROCM implementation of the applications. The input mesh sizes correspond to class D using a total 13GB of memory and 1024 zones for SP-MZ and BT-MZ, and 16 zones for LU-MZ. As described in Section 2, NPB-MZ applications operate over a mesh composed of several *zones*. Along this section we use the term *task* to identify the computations performed over a *zone*. We have evaluated two OpenMP schedulers: STATIC and DYNAMIC with chunk = 1, plus an optimized profile-based variant of the static scheduler. For the latter, a pre-computed performance conversion factor (PCF) between GPUs and CPUs is derived through profiling techniques and then used to balance the work distribution between GPU-based CUs and CPU-based CUs. Table 3 shows the profiling data in terms the average task processing time for one task in a GPU compared to different CPU configurations. For instance, one zone (e.g., task) is processed by 1 GPU in 0.83 ms for the SP-MZ benchmark. While 16 CPUs process one zone in 9.00 ms giving a performance factor conversion between the two configurations of approximately 1. This relation is used to adapt the STATIC work distribution in accordance to this factor so that 1 GPU and a group of 16 CPUs are given the same or similar number of tasks.

5.1 | Performance analysis

Figures 16–18 show the performance of the class-D SP-MZ, LU-MZ, and BT-MZ respectively. All figures include execution times and speedups for several configurations and schedulings (STATIC, DYNAMIC and the profile-based optimized STATIC). The charts expose the performance of the *Computation Period* and the *Communication Period* (execution time in ms). Rightmost axis of the chart shows the speedup respect the nonhybrid 1 CPU-based execution.

5.1.1 | SP-MZ benchmark

For only-CPU configurations the *Computation Period* speedups range from 15× and 24×, exposing a clear lack of scalability. The main reason for this is that last level cache has a small capacity compared to the input data size: 256 MB in contrast to 13 GB (Table 1). Consequently, CPU-based versions

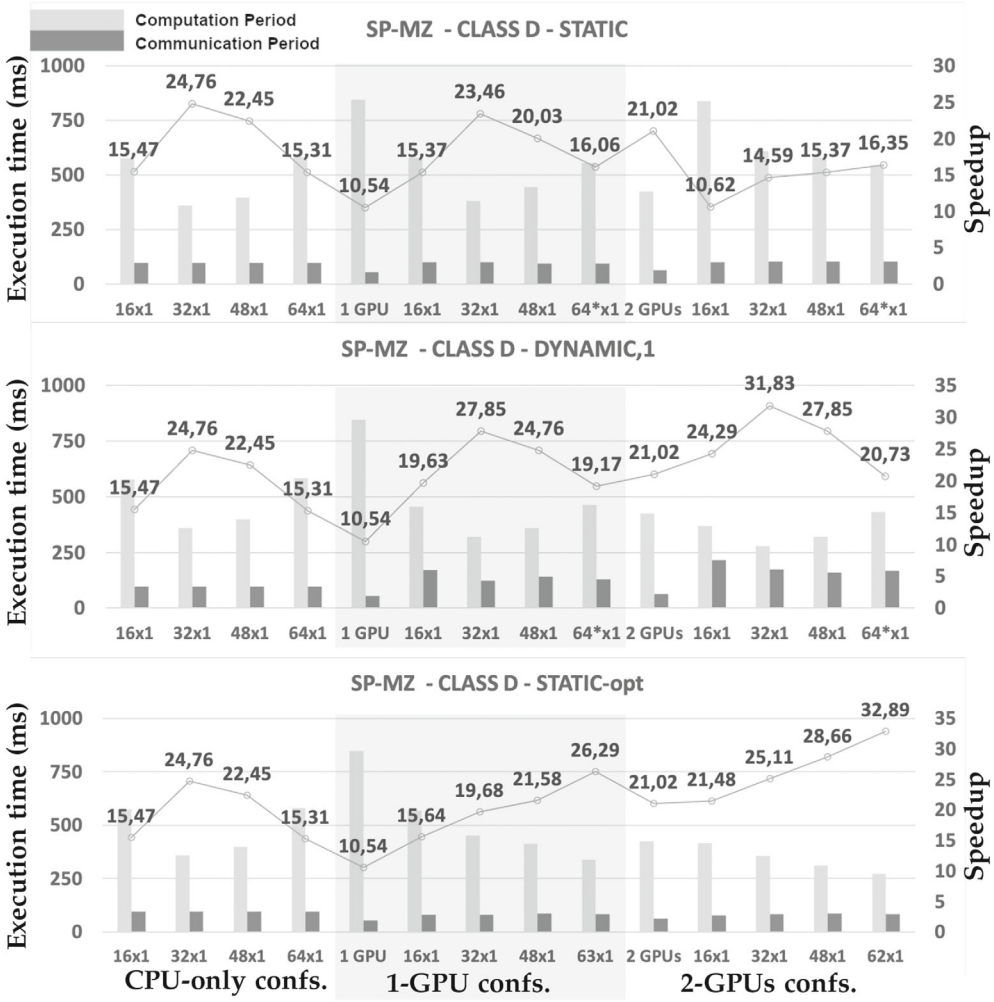

FIGURE 16 Benchmark SP-MZ. *Computation Period* (light grey) and *Communication Period* (dark grey) execution times (left axis) and speedup (right axis) for *Computation Period* with respect to 1-CPU configuration. From left to right, first group of bars depicts CPU-only configurations, second group (shaded area) depicts 1-GPU configurations and third group depicts 2-GPU configurations.

do not scale well with the increment of CUs. The time it takes to compute one zone when executing with 16, 32, 48, and 64 CUs is not constant. The pressure over the memory subsystem is different in each configuration, making executions with a higher count of CUs process one zone with higher execution times. Table 3 exposes the average execution time per task under different configurations. Executions from 16×1 up to 64×1 expose that task time almost increases by a factor of 4.5×. This affects the hybrid configurations similarly. Notice that, from data in Table 3, it is clear the difference of the one task processing time between GPUs and CPUs.

- *STATIC*: Hybrid configurations with 1 and 2 GPUs expose a similar trend than the nonhybrid CPU-only. Here, the scheduler is not able to match the difference in compute power of the CPUs and GPUs. Thus, the hybrid executions of the *Computation Period* suffer from a high degree of work unbalance. The observed speedups range between 10× and 23× for the STATIC scheduling. The lack of scalability for the zones processed in CPU-based CUs is observed when executing with more than 32 threads.
- *DYNAMIC*: For hybrid configurations, the observed speedups range from 19× to 31×. The scheduler succeeds in capturing the different compute power of CUs by assigning more tasks to faster CUs (the GPUs). With more than 32 CUs there is a performance degradation again due to the increment of the task processing time (Table 3). Although the improvements in the *Computation Period* compared to the STATIC scheduler, there is a significant increment in the execution time for the *Communication Period*: from ∼ 97 to ∼ 200 ms depending on the number of active CUs. This scheduler tends to distribute the zones so that adjacent zones no longer are assigned to the same CU. Thus, requiring many more data transfers between the CUs when computing border values. This is essential to understand the overall performance later described in Section 5.2.
- *STATIC-opt*: this scheduler applies a STATIC scheduling but conditioned by a PCF value that rearranges the task-CU assignment so that faster CUs receive more tasks. In addition, the scheduler tends to assign adjacent zones to the same CU in contrast to what the DYNAMIC scheduling does.

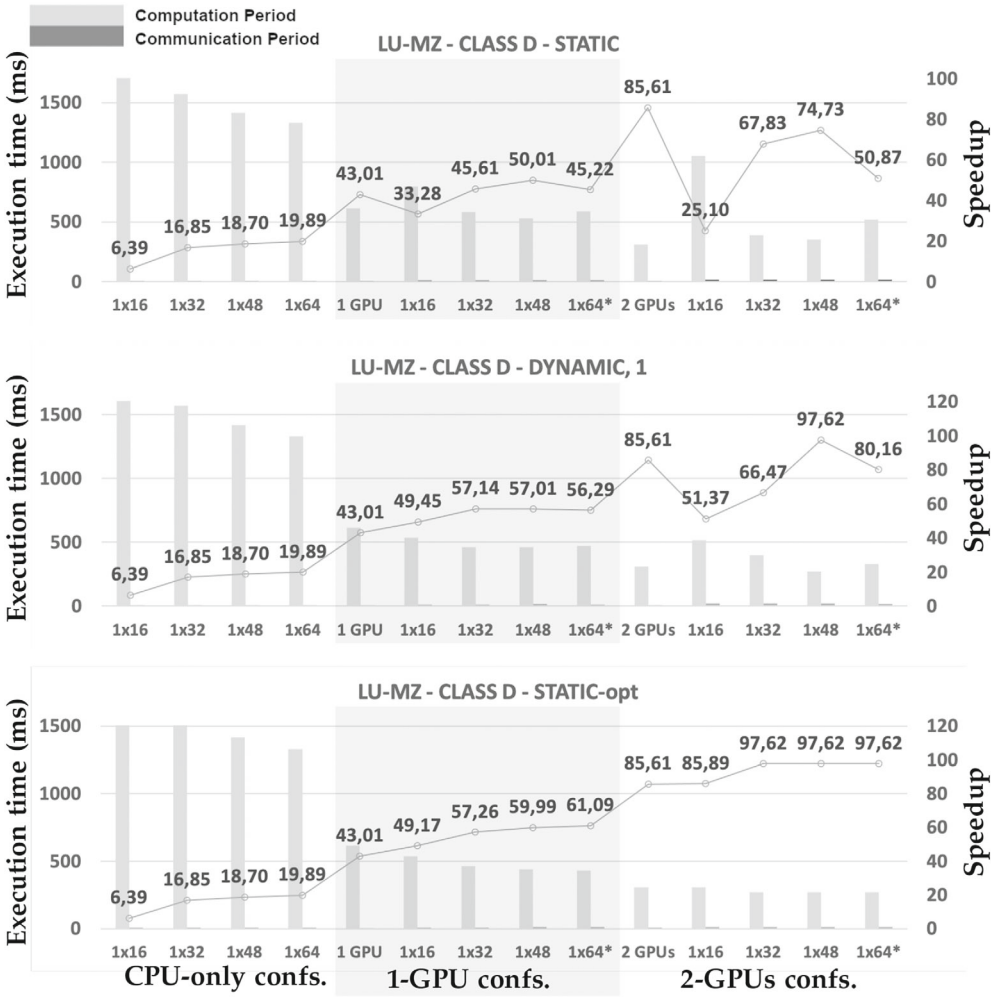

FIGURE 17 Benchmark LU-MZ. *Computation Period* (light grey) and *Communication Period* (dark grey) execution times (left axis) and speedup (right axis) for *Computation Period* with respect to 1-CPU configuration. From left to right, first group of bars depicts CPU-only configurations, second group (shaded area) depicts 1-GPU configurations and third group depicts 2-GPU configurations.

In this regard, this scheduler tries to solve the limitations for the DYNAMIC and original STATIC schedulers looking for a trade-off between the impact on the *Computation Period* and *Communication Period*. The speedups from the *Computation Period* range between 15× and 32×. Because of the PCF being build upon the task execution time, this scheduler improves scalability. The *Computation Period* experiments increments of performance as long as the number of active CUs is also increased. The *Communication Period* exposes a constant time no matter the number of CUs, as tasks are assigned so that adjacent zones are processed by the same CU. This is essential for later understand why this scheduler exposes higher overall performance in Section 5.2.

5.1.2 | LU-MZ benchmark

Figure 17 shows the performance of the class-D LU-MZ application. For the *Computation Period*, speedups range from 6× and 19×. Although these are not very significant increments, notice that for these configurations there is just one single CPU-based CU, composed of 16, 32, 48, and 64 CPUs. The performance increments are showing that innermost loops in the computational stages respond moderately well to the increment of threads.

- *STATIC*: the resulting work distribution fails to balance the assignment between CPU-based and GPU-based CUs. Thus, we observe a very poor response to the increment on the total count of CUs: speedups for 1-GPU hybrid configurations range between 30× and 50×. For 2-GPU hybrid configurations, speedups range 25× and 74×. In both cases, the results are far below from the speedup value observed for nonhybrid 2-GPU and 1-GPU configurations: speedups of 43× and 85× with 1 and 2 GPUs respectively.

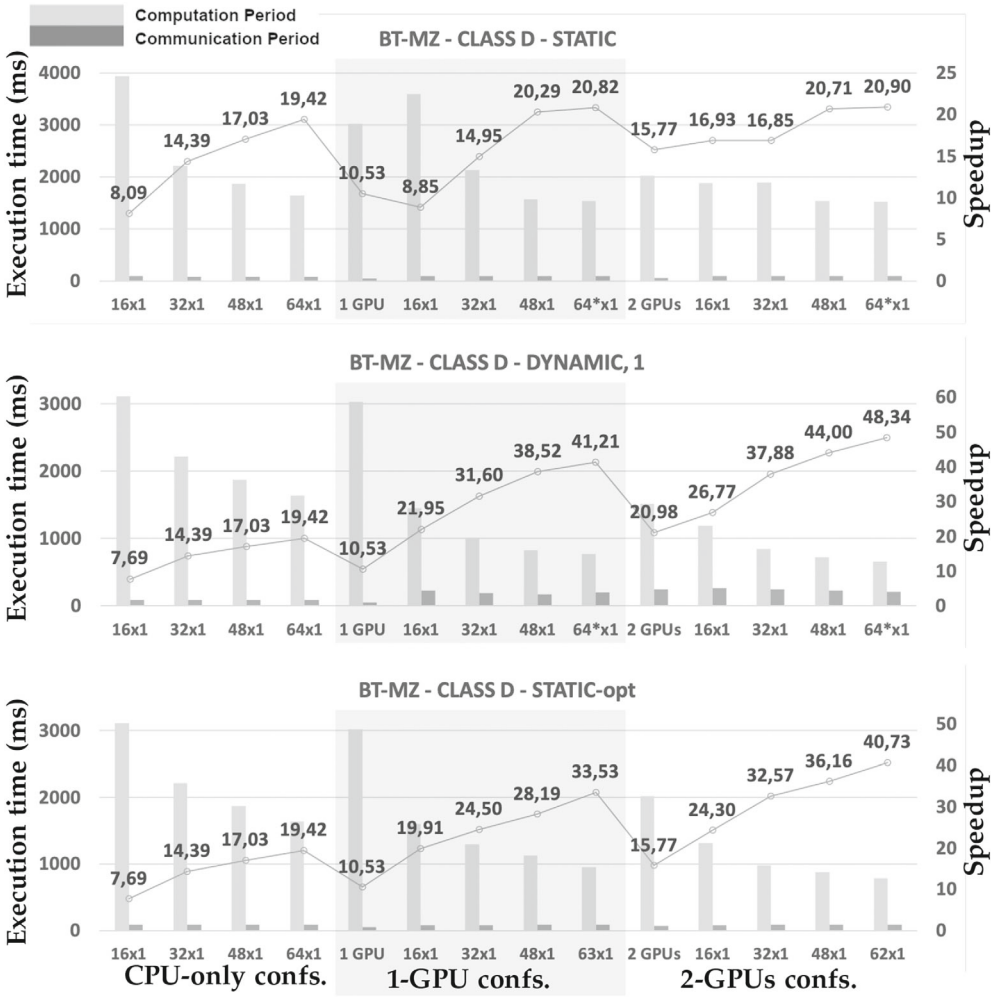

FIGURE 18 Benchmark BT-MZ. *Computation Period* (light grey) and *Communication Period* (dark grey) execution times (left axis) and speedup (right axis) for *Computation Period* with respect to 1-CPU configuration. From left to right, first group of bars depicts CPU-only configurations, second group (shaded area) depicts 1-GPU configurations and third group depicts 2-GPU configurations.

- *DYNAMIC*: the work balance problem is largely improved for 1-GPU configurations. We observe a maximum speedup of 57× with 1-GPU and 48 CPU threads. But for 2-GPU configurations we have observe that the optimal work distribution corresponds to just assign one task to the CPU-based CU. This is not achieved in the 1×16, 1×32, and 1×64 configurations, where 2 or more tasks are assigned to the CPU-based CU. This explains the poor performance. In all configurations, the *Computation Period* dominates the total execution time over the *Communication Period* execution time. Even for the DYNAMIC scheduler which distributes adjacent zones to different CUs, it does not present an increment of the *Communication Period* execution time. In contrast to the SP-MZ benchmark, the LU-MZ input mesh is organized in *very few* and *very large* zones. This implies that the ratio between border elements and zone elements is very small. This causes that we do not observe a significant increment in the execution time of the *Communication Period*.
- *STATIC-opt*: with this scheduler, the performance of the hybrid configurations responds well to the increment of active CUs. We observe speedups that range from 57× and 61× for 1-GPU configurations, and between 85× and 97× fro 2-GPU configurations. In this case, the problem observed in the DYNAMIC scheduling is solved, as for this scheduler just one task is assigned to the CPU-based CU.

5.1.3 | BT-MZ benchmark

Figure 18 shows the performance of the class-D BT-MZ application. For the *Computation Period*, speedups range from 8× and 19×. The modest scalability is related to two main aspects. Once again, the last level cache memory has a small capacity compared to the input data size: 256 MB in contrast to 13GB (Table 1). The time it takes to compute one zone when executing with 16, 32, 48, and 64 CUs is not constant (see Table 3). Moreover,

BT-MZ computes over a mesh composed of 1024 not equally sized zones. The input set defines an scenario of *many* tasks combining *small*, *medium* and *large* tasks compared to the scenarios for the LU-MZ and SP-MZ cases.

- *STATIC*: hybrid configurations show speedup factors from 8× to 20×. This corresponds to poor performance as the scheduler fails to distribute the tasks in a balanced manner. Compared to the cases of SP-MZ and LU-MZ, there is an additional factor that complicates the definition of a well-balanced work distribution: the zones are not equally sized. In none of the hybrid configurations the scheduler exposes more than a speedup 20× factor compared to nonhybrid configurations with 1-GPU (10×) and 2-GPU (15×) (notice how this corresponds to poor scaling, essentially caused by the degree of work unbalance in the input mesh).
- *DYNAMIC*: speedups for the *Computation Period* range from 21× to 48×, much better than using the STATIC scheduling. This scheduler adapts to both the difference in compute power of the CUs and the differences in the zone sizes. The *Communication Period* is affected by an increment in its execution time mainly due to the fact that adjacent zones are assigned to different CUs, thus requiring additional data transfers to compute border values. We have observed that for CPU-only configurations the *Communication Period* takes ∼ 80 ms, while for hybrid configurations it takes ∼ 200 ms.
- *STATIC-opt*: the performance of this scheduler is lower than the DYNAMIC scheduler. The *Computation Period* speedup ranges between 20× and 40× depending on the number of CUs. The reason for that is that the STATIC-opt only adapts to the difference in compute power of the CUs, but not to the work unbalance generated by the differences in the zone sizes. The *Computation Period* dominates the total execution time over the *Communication Period* which in contrast to its counterpart under the DYNAMIC scheduling, does not increase its execution time.

5.2 | Overall performance

Figure 19 exposes overall performance in terms of speedup with respect the serial (e.g., 1 CPU) version of each benchmark.

5.2.1 | *SP-MZ* benchmark

For 1-GPU executions, speedup is 10×. As more CUs are added, the response is different in each scheduler. STATIC fails to adapt the work distribution according to the different type of CUs. Maximum speedup is 18× with 32 additional CUs, but dropping with 48 and 63 CUs. DYNAMIC performs better for 32 or less additional CUs with speedups of 14× and 20× for 16 CUs and 32 CUs configurations. But, as we have already seen, this scheduler tends to map adjacent zones to different CUs, thus causing additional overheads in the *Communication Period*. Only the STATIC-opt scheduler keeps increments in overall speedup as long as the CU count is incremented. Maximum speedup for 1-GPU hybrid configurations is 21× and 63 additional CUs. For 2-GPUs we have observed a 25× maximum factor of speedup. This scheduler adapts well the work distribution between the CUs and minimizes the data transfers within the *Communication Period* in contrast to the DYNAMIC scheduler. In general, the speedups with respect the GPU-only executions range from 1.4× to 2.1×, in the case of the STATIC-opt scheduler. The STATIC and DYNAMIC are in ranges from 1.3× to 2×.

5.2.2 | *LU-MZ* benchmark

For 1-GPU nonhybrid executions, speedup is 42×. With additional CUs, speedups range between 44× and 48× showing a drop for the 1-GPU and 63 CPUs. Both DYNAMIC and STATIC-opt schedulers outperform from 55× to 59×, having a maximum value for the STATIC-opt scheduler with 1-GPU

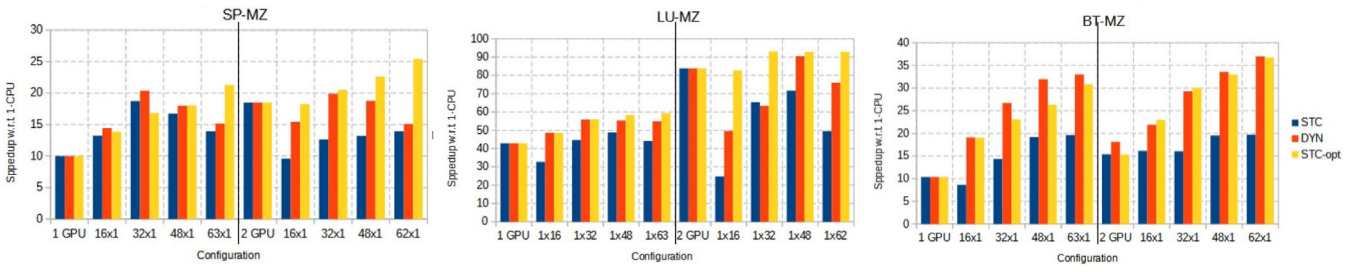

FIGURE 19 Overall performance for SP-MZ, LU-MZ and BT-MZ. Input class D. Static (STC), Dynamic with chunk=1 (DYN) and Static optimized with performance conversion factor (STC-opt) schedulers. Speedup factors compared to sequential execution with 1 × 1 configuration (1 CPU).

and 63 CPUs. For 2-GPU executions, the base speedup is 83× and trends are similar to those observed previously. In this case, maximum speedup is 92× and corresponds to STATIC-opt with 62 additional CUs. With respect the GPU-only executions, hybrid configurations expose speedups in the range of 1.2× to 2.1×, in the case of the STATIC-opt scheduler. The STATIC scheduler presents significant slowdown, from 0.5× to 1.08. The DYNAMIC performance is in the range of 0.6× to 1.20×.

5.2.3 | *BT-MZ* benchmark

For 1-GPU, initial speedup is 10×. As the CU count is increased, the response is very poor: speedups range between 8× and 19× for the STATIC scheduler. The reason for this trend is related to 2 factors: the difference between CUs and the fact that the zones are not equally sized. DYNAMIC and STATIC-opt schedulers solve both issues outperforming from 19× to 33×, having the maximum value for the DYNAMIC scheduler and 63 additional CUs. In this case, the dynamic assignment of zones to CUs, balances the load across the CUs in a much more effective manner than a profile-based mapping. STATIC-opt is guided by average task time (Table 3), while DYNAMIC is guided by actual execution time of tasks. The work unbalance caused by the differences in the zone sizes has more impact that the different compute power of GPUs and CPUs. For 2 GPUs, the trends are similar with speedups from 18× to 36×. The BT-MZ application exposes significant speedups respect the GPU-only executions for all the schedulers, unless for the STATIC. Speedups range from 1.08× to 3×, being the DYNAMIC scheduler the one that gives the best performance levels. In this case, the fact that the input set is organized in zones of very different size makes the DYNAMIC scheduler more suitable to achieve a balanced work distribution.

5.3 | Programmability assessment

The OpenMP extensions described in this article have made possible the deployment of a methodology to enable hybrid executions mixing OpenMP and CUDA/HIP. In terms of programmability, we have introduced the abstraction of the *Computing Unit* within the OpenMP programming model in a transparent fashion and in accordance to the current specification of the OpenMP programming paradigm. In addition, we have provided with a code transformation to enable task parallelism accompanied with a set of placement and scheduling primitives that make possible the simultaneous activation of host and device computing units.

In case of mixing MPI+OpenMP,[9-13] these solutions require similar runtime primitives as those explored in this work. In MPI,[14] work distribution would happen manually across the different MPI ranks. And within each of the ranks, the simultaneous activation of the host cores and the device cores would also happen through manual re-coding of the application. One immediate drawback of this approach corresponds to the mapping of host cores to MPI ranks for locality purposes. OpenMP includes support to guide the thread affinity to hardware resources. Though for MPI it is also possible to use tools like *numactl* or *cpusets*, this complicates the task of the programmer, compared to the current OpenMP solution, fully integrated in the specification with the definition of *places* and thread binding (i.e., usage of environment variables OMP_PLACES and OMP_PROC_BIND).

6 | RELATED WORKS

NPB-MZ studies: Dümmler and Rünger[8] evaluated NPB-MZ benchmarks on hybrid CPU+GPU architectures. Workloads are decomposed and, using a static scheduling, distributed among CPUs and GPUs. The evaluations show a significant performance improvement with respect to both pure GPU and pure CPU implementations. Pennycook and Hammond[15] implemented the LU application on CUDA and developed an analytical model to estimate the execution time of the benchmark on a range of architectures. They validated the model using evaluation environments that range from a single GPU to a cluster of GPU's. Porting scientific codes to heterogeneous architectures has been a general trend within the HPC community. For instance, Lattice-Boltzmann (LBM)[16,17] and Computational Fluid Dynamic (CFD)[18] codes have been ported to heterogeneous architectures mixing more than one programming model: MPI+CUDA, OpenMP+CUDA, or MPI+OpenMP+CUDA. The proposals in these papers point out the necessity of extending OpenMP to support the simultaneous activation of compute units of different nature. Our proposal solves this necessity with the introduction of the Compute Unit abstraction within the OpenMP programming model.

Extensions for Heterogeneity: For OpenMP and heterogeneous architectures, one main research line corresponds to the introduction of meta-directives to guide the compiler for code versioning and run-time code selection.[6,19,20] These works solve some of the current OpenMP limitations addressed in this article. For instance, the usage of meta-directives can direct an OpenMP thread toward the offloading of computations to devices. But none of these works addresses the current inability of distributing work from work-sharing constructs among threads that have to be directed to different types of computing resources (e.g., CPUs or GPUs). In this regard, new work-distribution schemes, specially for task-level parallelism[21-24] have been proposed. These works include proposals to facilitate the specification of whether a computation should be

offloaded to a device or to the host. The main strategy is combining OpenMP and OpenACC, and manually code control-flow statements that direct the thread execution toward either a device or the host. Compared to our proposal, the introduction of this kind of thread control is similar, but these previous works do not address the configuration for OpenMP threads and devices. Neither they include the dynamic adaptation for the work-distribution schemes to consider the eventual work unbalance between the host and the devices. In a similar fashion, other works also combine OpenMP with device-oriented paradigms to strictly activate the devices generating a multi-GPU execution.[25] In this case, the proposal does not consider the simultaneous activation of both the devices and the CUs in the host. White[26] proposes a clustering directive for computational resources (e.g., cores) which goes in the same direction as the proposal described in this article. But the main differences arise from the introduction of an abstraction such as *Compute Unit* and also the lack of support for a distributed shared memory approach. More recent works have evaluated the OpenMP 5.2 specification,[27,28] and have indicated the lack for appropriate work-distribution schemes for hybrid executions, as well as the nonexistence of support to solve the entanglement between the work distribution and the data placement in a distributed shared-memory architecture.

Heterogeneous work scheduling: Research on heterogeneous computing has focused on porting applications to this type of architectures. Many works from different domains describe the adaptation of specific frameworks to execute on multi-GPU systems, where CPUs orchestrate the parallel execution, and GPUs act as accelerators.[29-33] Other works describe cooperative heterogeneous computing frameworks which introduce adaptive and dynamic work distributions that enable efficient utilization of both CPUs and GPUs for CUDA kernels.[19,20,34] Performance models have been proposed to implement work-distribution schemes.[35,36] Ogata et al.[37] present a library for 2D Fast Fourier Transform (FFT) that automatically uses both CPUs and GPUs to boost performance. Using a performance model, they evaluate the contributions of each computing unit and then make an estimation of total execution time. Gao et al.[38] present a framework where compute units are used in the context of a graph representation of the computation and available parallelism, based on *codelets*. Similar work is developed by Lauderdale et al.[39] In both cases, the problems addressed are the same as those OpenMP has to face, regarding heterogeneous work distribution and compute unit abstraction. But the execution framework in these solutions is largely different as compared to the actual OpenMP execution model. Therefore it becomes very difficult to import the solutions in these works to the OpenMP programming model.

7 | CONCLUSIONS

In this article, we have introduced the concepts of *Compute Unit* (CU) and *Data Placement* to the OpenMP programming model. CUs are defined as an aggregation of CPUs or GPUs. OpenMP threads are associated to these CUs. New OpenMP constructs have been described to guide the definition of the CUs. Data placement is supported with new constructs that guide both the compiler and the runtime so that application data structures are placed and replicated as necessary according to the work distribution among the CUs. This has been shown to be an essential aspect of an heterogeneous execution, where the compute power of CUs is not homogeneous. Thus, it is necessary to avoid work unbalance and to adapt the work distribution to this aspect.

In conclusion, the new constructs have been proven to be effective at the programming level, enriching the expressiveness of the OpenMP programming model, covering the necessary support for specifying heterogeneous parallelizations in OpenMP. The article shows how to use the new constructs, making possible a hybrid multilevel parallelization of the NPB-MZ benchmarks. The implementation exploits both coarse-grain and fine-grain parallelism, mapped to CUs of different nature (either GPU-based or CPU-based). We compare hybrid and nonhybrid executions under two state-of-the-art work-distribution schemes (Static and Dynamic). On a computing node composed of one AMD EPYC 7742 @ 2.250GHz (64 cores and 2 threads/core, totalling 128 threads per node) and 2× GPU AMD Radeon Instinct MI50 with 32GB, hybrid executions speedup from 1.08× up to 3.18× with respect to a nonhybrid GPU implementation, depending on the number of activated CUs.

ACKNOWLEDGMENT

This work was supported by the Spanish Ministry of Science and Technology (PID2019-107255GB).

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Marc Gonzàlez-Tallada https://orcid.org/0000-0002-3780-1106
Enric Morancho https://orcid.org/0000-0003-2403-8145

REFERENCES

1. Zheng Y, Kamil A, Driscoll MB, Shan H, Yelick K. UPC++: A PGAS extension for C++. *International Parallel and Distributed Processing Symposium*. IEEE; 2014:1105-1114.

2. Shan H, Blagojević F, Min S-J, et al. A programming model performance study using the NAS parallel benchmarks. *Sci Program*. 2010;18(3–4):153-167. 10.1155/2010/715637
3. Frumkin M, Jin H, Yan J. Implementation of NAS parallel benchmarks in high performance Fortran.
4. Bailey D, Barszcz E, Barton J, et al. The NAS Parallel Benchmarks. *Int J High Perform Comput Appl*. 1991;5(3):63-73. 10.1177/109434209100500306
5. der Wijngaart RFV, Jin H. *NAS Parallel Benchmarks, Multi-Zone Versions*. NASA Ames Research Center; Tech. Rep. NAS-03-010; 2003.
6. Yan Y, Wang A, Liao C, Scogland TRW, de Supinski BR. Extending openmp metadirective semantics for runtime adaptation. In: Fan X, de Supinski BR, Sinnen O, Giacaman N, eds. *OpenMP: Conquering the Full Hardware Spectrum*. Springer International Publishing; 2019:201-214.
7. Bull JM. Feedback guided dynamic loop scheduling: Algorithms and experiments. In: Pritchard D, Reeve J, eds. *Euro-Par'98 Parallel Processing*. Springer Berlin Heidelberg; 1998:377-382.
8. Dümmler J, Rünger G. Execution schemes for the NPB-MZ benchmarks on hybrid architectures: a comparative study. *Procs. of the Intl. Conf. on Parallel Computing, ParCo 2013, ser. Advances in Parallel Computing*. Vol 25. IOS Press; 2013:733-742 10.3233/978-1-61499-381-0-733
9. Karunadasa NP, Ranasinghe DN. Accelerating high performance applications with cuda and MPI. *2009 International Conference on Industrial and Information Systems (ICIIS)*. IEEE ; 2009:331-336.
10. Peña AJ, Lai J, Yu H, Tian Z, Li H. Hybrid MPI and CUDA parallelization for CFD applications on multi-GPU HPC clusters. *Sci Program*. 2020;2020:8862123.
11. Kraus J. An introduction to cuda-aware MPI. 2013 https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/
12. Awan AA, Manian KV, Chu C-H, Subramoni H, Panda DK. Optimized large-message broadcast for deep learning workloads: MPI, MPI+NCCL, or NCCL2? *Parallel Comput*. 2019;85:141-152.
13. Jacobsen D, Thibault J, Senocak I. An MPI-cuda implementation for massively parallel incompressible flow computations on multi-GPU clusters. *Inanc Senocak*. 2010;16:1.
14. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee; 1994.
15. Pennycook SJ, Hammond SD, Jarvis SA, Mudalige GR. Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark. *SIGMETRICS Performance Evaluat Rev*. 2011;38(4):23-29.
16. Riesinger C, Bakhtiari A, Schreiber M, Neumann P, Bungartz H-J. A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters. *Computation*. 2017;5(4):26. https://www.mdpi.com/2079-3197/5/4/48
17. Lee S, Gounley J, Randles A, Vetter JS. Performance portability study for massively parallel computational fluid dynamics application on scalable heterogeneous architectures. *J Parallel Distrib Comput*. 2019;129(C):1-13. 10.1016/j.jpdc.2019.02.005
18. Borrell R, Dosimont D, Garcia-Gasulla M, et al. Heterogeneous CPU/GPU co-execution of CFD simulations on the power9 architecture: Application to airplane aerodynamics. *Futur Gener Comput Syst*. 2020;107:31-48.
19. Scogland TR, Rountree B, Feng W-c, de Supinski BR. Heterogeneous Task Scheduling for Accelerated OpenMP. *International Parallel and Distributed Processing Symposium*. IEEE; 2012:144-155.
20. Scogland TRW, Feng W-c, Rountree B, de Supinski BR. CoreTSAR: Adaptive worksharing for heterogeneous systems. *Supercomputing*. Springer International Publishing; 2014:172-186.
21. Ferrer R, Planas J, Bellens P, et al. Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL. *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg; 2011:215-229.
22. Ayguade E, Badia RM, Cabrera D, et al. A proposal to extend the OpenMP tasking model for heterogeneous architectures. *Evolving OpenMP in an Age of Extreme Parallelism*. Springer Berlin Heidelberg; 2009:154-167.
23. Scogland TRW, Rountree B, Feng W c, de Supinski BR. Heterogeneous task scheduling for accelerated *OpenMP*. *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. Vol 2012. IEEE; 2012:144-155.
24. Neth B, Scogland T, Duran A, de Supinski B. Beyond explicit transfers: shared and managed memory in OpenMP. *LNCS, OpenMP: Enabling Massive Node-Level Parallelism. IWOMP*. Vol 12870; Springer International Publishing; 2021:2021.
25. Xu R, Chandrasekaran S, Chapman B. Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model. *2013 IEEE International Symposium on Parallel Distributed Processing*. IEEE; 2013:1169-1176.
26. White L. OpenMP extensions for heterogeneous architectures. *OpenMP in the Petascale Era*. Springer Berlin Heidelberg; 2011:94-107.
27. Tian S, Chesterfield J, Doerfert J, Chapman B. Experience report: Writing a portable GPU runtime with OpenMP 5.1. In: McIntosh-Smith S, de Supinski BR, Klinkenberg J, eds. *OpenMP: Enabling Massive Node-Level Parallelism*. Springer International Publishing; 2021:159-169.
28. Kale V, Lu W, Curtis A, Malik AM, Chapman B, Hernandez O. Toward supporting multi-GPU targets via taskloop and user-defined schedules. *OpenMP: Portable Multi-Level Parallelism on Modern Systems*. Lecture Notes in Computer Science. Springer International Publishing; 2020:295-309.
29. Hermann E, Raffin B, Faure F, Gautier T, Allard J. Multi-GPU and multi-CPU parallelization for interactive physics simulations. *Euro-Par 2010-Parallel Processing*. Springer Berlin Heidelberg; 2010:235-246.
30. Nere A, Franey S, Hashmi A, Lipasti M. Simulating cortical networks on heterogeneous multi-GPU systems. *J Parallel Distribut Comput*. 2013;73(7):953-971. https://www.sciencedirect.com/science/article/pii/S0743731512000408
31. Toharia P, Robles OD, Suárez R, Bosque JL, Pastor L. Shot boundary detection using Zernike moments in multi-GPU multi-CPU architectures. *J Parallel Distrib Comput*. 2012;72(9):1127-1133. doi:10.1016/j.jpdc.2011.10.011
32. Chen L, Huo X, Agrawal G. Accelerating MapReduce on a coupled CPU-GPU architecture. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis ser. SC'12*. IEEE Computer Society Press; 2012.
33. Yang C, Xue W, Fu H, et al. A peta-scalable CPU-GPU algorithm for global atmospheric simulations. *SIGPLAN Not*. 2013;48(8):1-12. doi:10.1145/2517327.2442518
34. Yang C, Wang F, Du Y, et al. Adaptive optimization for petascale heterogeneous CPU/GPU computing. *in 2010 IEEE International Conference on Cluster Computing*. IEEE Computer Society; 2010:19-28. doi:10.1109/CLUSTER.2010.12
35. Choi HJ, Son DO, Kang SG, Kim JM, Lee H-H, Kim CH. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *J Supercomput*. 2013;65(2):886-902. doi:10.1007/s11227-013-0870-6
36. Zhong Z, Rychkov V, Lastovetsky A. Data partitioning on heterogeneous multicore and Multi-GPU systems using functional performance models of data-parallel applications. *IEEE International Conference on Cluster Computing*. IEEE; 2012:191-199.
37. Ogata Y, Endo T, Maruyama N, Matsuoka S. An efficient, model-based CPU-GPU heterogeneous FFT library. *2008 International Symposium on Parallel and Distributed Processing*. IEEE; 2008:1-10. doi:10.1109/IPDPS.2008.4536163

38. Suettlerlein J, Zuckerman S, Gao GR. An implementation of the codelet model. In: Wolf F, Mohr B, Mey D a, eds. *Euro-Par 2013 Parallel Processing*. Springer Berlin Heidelberg; 2013:633-644.
39. Lauderdale C, Khan R. Towards a codelet-based runtime for exascale computing: Position paper. *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, ser. EXADAPT '12*. Association for Computing Machinery; 2012:21-26. doi:10.1145/2185475.2185478

How to cite this article: Gonzàlez-Tallada M, Morancho E. Compute units in OpenMP: Extensions for heterogeneous parallel programming. *Concurrency Computat Pract Exper*. 2023;e7885. doi: 10.1002/cpe.7885