Journal Title
XX(X):1–18

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Heterogeneous Programming Using OpenMP and CUDA/HIP for Hybrid CPU-GPU Scientific Applications

Marc González and Enric Morancho

Abstract

Hybrid computer systems combine compute units (CUs) of different nature like CPUs, GPUs and FPGAs. Simultaneously exploiting the computing power of these CUs requires a careful decomposition of the applications into balanced parallel tasks according to both the performance of each CU type and the communication costs among them. This paper describes the design and implementation of runtime support for OpenMP hybrid GPU-CPU applications, when mixed with GPU-oriented programming models (e.g.: CUDA/HIP). The paper describes the case for a hybrid multi-level parallelization of the NPB-MZ benchmark suite. The implementation exploits both coarse-grain and fine-grain parallelism, mapped to compute units of different nature (GPUs and CPUs). The paper describes the implementation of runtime support to bridge OpenMP and HIP, introducing the abstractions of *Computing Unit* and *Data Placement*. We compare hybrid and non-hybrid executions under state-of-the-art schedulers for OpenMP: static and dynamic task schedulings. Then, we improve the set of schedulers with two additional variants: a memorizing-dynamic task scheduling and a profile-based static task scheduling. 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 x GPU AMD Radeon Instinct MI50 with 32GB, hybrid executions present speedups from 1.10x up to 3.5x with respect to a non-hybrid GPU implementation, depending on the number of activated CUs.

Keywords

Heterogeneous Programming, Hybrid CPU-GPU, OpenMP, CUDA, HIP

Introduction

Hybrid computing systems have become the widest spread solution within the High Performance Computing (HPC) domain. Hybrid systems combine computing units (CUs) of different nature and computational power, for instance CPUs, GPUs and FPGAs. Given the impressive computing power delivered by these architectures, significant efforts have been devoted to port applications from many computing domains to this type of HPC systems. Machine Learning, BioInformatics, Scientific Computing and many more have clear examples of representative applications like TensorFlow (Abadi et al. 2015), Caffe (NVIDIA 2020), Smith-Waterman (Manavski and Valle 2008), Alya (Giuntoli et al. 2019) that recently developed the support for GPU-based systems.

Hybrid systems are programmed with language extensions to general-purpose programming languages such as C/C++ and Fortran. Nvidia's CUDA, AMD's HIP, OpenCL, OpenACC and OpenMP are reference programming models for heterogeneous computing that follow this approach. In general, all these programming frameworks focus on the essential actions for porting an application to a hybrid architecture: memory allocation, data transfers between CUs that now operate within a distributed memory address space, and the specification of those computations to be offloaded to GPU-based CUs and those to be executed by CPU-based CUs. OpenMP includes many language constructs to guide all these aspects.

Hybrid applications simultaneously exploit fine and coarse levels of parallelism. This requires the definition of memory-allocation strategies per each device, work-distribution schemes aware of the different nature of the CUs to balance the work execution, and communication phases whenever is necessary to exchange values between the CUs that reside in different memory spaces. In particular, OpenMP has evolved with new features so that all of these aspects are reasonably supported through specific constructs and practices. Nevertheless, most of this support focuses on the sole utilization of the OpenMP programming model, not addressing the fact that many applications now have been ported to GPU-based systems using other programming models such as CUDA or HIP. OpenMP has to be able to reasonably support too the interoperability with these defacto heterogeneous programming standards. Both CUDA/HIP runtime systems define very similar interfaces to operate with GPU devices. The main functionalities that need to be addressed are how computations are offloaded to the devices, how memory is allocated into devices and mirrored with host memory, and how data is transferred from/to devices and host. All of these aspects are supported by a set of runtime primitives included in CUDA/HIP that require

Computer Architecture Department, Universitat Politècnica de Catalunya-BarcelonaTECH, Barcelona, Spain

Corresponding author:
Marc González, marc@ac.upc.edu

the specification of how to interoperate with the OpenMP execution model.

Mixing CUDA/HIP and OpenMP is a common practice (Yang et al. 2011, 2021; Corni et al. 2016; Jacobsen and Senocak 2013; Guan et al. 2013). In this context, the threading level is utilized to activate several devices, having one or more threads responsible for: device memory allocation, data transfers and kernel offloading. But if this approach is extended to simultaneously activate both the CPU and GPU cores, then programmers have to face significant programming limitations. Essentially, programmers have to manually introduce code responsible for an appropriate work distribution that adapts to the inherent differences of CPU/GPU compute power, and to distribute data among the host and device memory spaces according to the work distribution. All of this has been observed in previous works where specific work-distribution mechanisms are described for hybrid architectures (Mittal and Vetter 2015; Belviranli et al. 2013; Choi et al. 2013; Zhong et al. 2012; Zhang et al. 2021; Gowanlock 2021).

In this context, the challenge of simultaneously activating both CPUs and GPUs is essentially limited by the interoperability of OpenMP with CUDA/HIP. The main contribution of this paper is the design and implementation of a runtime system support to improve the interoperabilty of OpenMP and CUDA/HIP. We introduce two main abstractions build on top of the OpenMP programming model: *Computing Unit* and *Data Placement*. Using the proposed runtime system, we describe how to effectively mix OpenMP and AMD's HIP to achieve a multi-level parallelization of the NPB-MZ benchmark suite where both the CPU an GPU cores execute simultaneously. We evaluate its performance in a computing node composed of one AMD EPYC 7742 @ 2.250GHz (64 cores and 2 threads/core, totalling 128 threads per node) and 2 x GPU AMD Radeon Instinct MI50 with 32GB. Hybrid configurations present speedups from 1.08x up to 3.18x with respect to a non-hybrid GPU configuration, depending on the number of activated CUs.

This paper is organized as follows: Section describes the NPB-MZ benchmark suite and points out its sources of parallelism. Section details the design and implementation of the runtime support for OpenMP hybrid applications. Section depicts how the runtime support is used to achieve a hybrid implementation of the NPB-MZ benchmark suite. Section evaluates the performance of our hybrid implementation. Section discusses related works and, finally, Section concludes the paper.

Benchmark characterization

NAS Parallel Benchmarks

The NAS Parallel Benchmark (NPB) (Bailey et al. 1991) suite provides the implementation of different benchmarks for several programming languages (Fortran, C, ...) and parallel paradigms (OpenMP, MPI, ...). The suite defines several input classes for each benchmark; classes S, W, A, B,... determine a sequence of increasing input sizes. Finally, to verify the result of the benchmarks, NPB gives the expected result for each class. The original NPB benchmark suite consisted of eight benchmarks: five kernels and three pseudo-applications. In this work we focus just on the three pseudo applications: BT (Block Tri-diagonal solver), LU (Lower-Upper Gauss-Seidel solver) and SP (Scalar Penta-diagonal solver).

BT, LU and SP traverse a 3D volume (Figure 1-a) to compute discrete solutions of the Navier-Stokes equations. Figure 2-a shows the flow graph of NPB benchmarks. The main loop, the time-step loop, applies a benchmark-specific solver on every iteration.

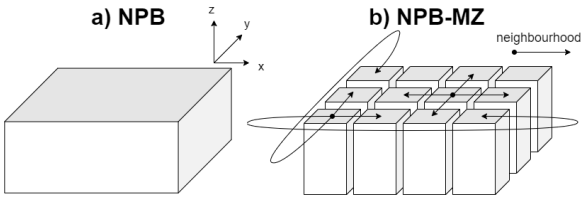

Figure 1. 3D Volume: a) NPB, b) NPB-MZ, example 4 × 3 tiling into 12 zones and neighbourhood relation between zones.

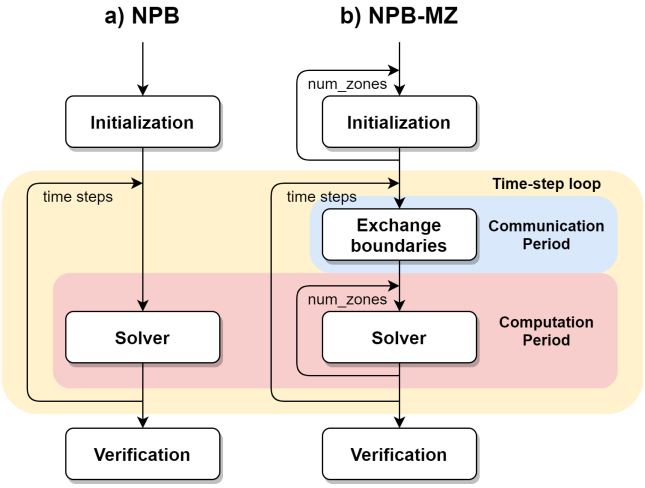

Figure 2. Flow graphs: a) NPB and b) NPB-MZ.

The Multi-Zone NPB (NPB-MZ) (der Wijngaart and Jin 2003) re-implement NPB to expose a coarse level of parallelism. To that end, the 3D input volume is tiled through both x and y dimensions producing a grid of prisms (known as *zones*, Figure 1-b) 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 neighbour zones. So, between two iterations of the time-step loop, an *exchange-boundaries* procedure must update the border faces of all zones.

Figure 2-b shows the flow graph of NPB-MZ benchmarks. Each time-step iteration must call 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 depicts the characterization of NPB-MZ benchmarks. For each input class, the table details the overall size, the number of zones, the zone size and the number of time steps of each benchmark. Notice some differences

Input Class	3D volume x × y × z (points)	Memory (GB)	Num. zones (x × y) LU	Num. zones (x × y) SP & BT	Zone size (points per zone) LU	Zone size (points per zone) SP	Zone size (points per zone) BT	Time steps LU	Time steps SP	Time steps BT
B	304 × 208 × 17	≈ 0.2	4 × 4	8 × 8	67 184	16 786	from 2 992 to 59 976	250	400	200
C	480 × 320 × 28	≈ 0.8	4 × 4	16 × 16	268 800	16 800	from 2 912 to 60 648	250	400	200
D	1 632 × 1 216 × 34	≈ 13	4 × 4	32 × 32	4 217 088	65 892	from 11 968 to 243 236	300	500	250
E	4 224 × 3 456 × 92	≈ 250	4 × 4	64 × 64	83 939 328	327 888	from 59 248 to 1 203 452	300	500	250

Table 1. Characterization of NPB-MZ benchmarks: overall size (number of points and memory requirements) and, for each benchmark, number of zones, zone size and number of time steps.

among the benchmarks that will be relevant in order to understand the performance of the hybrid implementations of the benchmarks:

- In LU, the number of zones is always 16, independently on the input class. For each input class, zone sizes are uniform.
- In SP, the larger the input class, the larger the number of zones. Like in LU, zone sizes are uniform at each input class; however, SP zones are smaller than LU zones.
- In BT, like in SP, 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×.

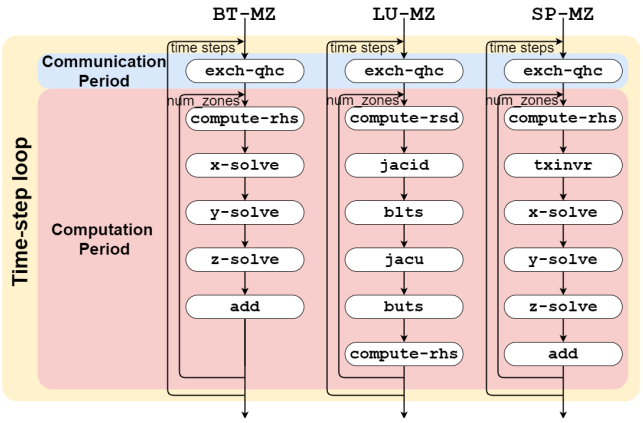

Figure 3. Time-step loop of NPB-MZ benchmarks. Sequence of procedures called at each iteration of the time-step loop.

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 Computing Units (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., multi-threading, vectorization and porting to GPU). The following subsections describe these parallelism levels.

Computation period: Inter-zone Parallelism

Figure 4 shows the skeleton of the time-step loop of the 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, the exploitation of *inter-zone* parallelism implies the parallelization of this loop.

```
for (step=0; step < num_steps; step++) {
  /* BORDER EXCHANGE */
  exch_qbc(...);

  /* ZONE PROCESSING */
  for (zone=0 ; zone < num_zones; zone++) {
    comp_phase_1 (...);
    comp_phase_2 (...);
    ...
    comp_phase_N (...);
  } /* zone loop */
} /* Time step loop */

void comp_phase_N(...) {
  /* Loop NEST 1 */
  for (k=0; k < zdim; k++)
    for (j=0; j < ydim; j++)
      for (i=0; i < xdim; i++) {
        /* MATRIX BASED COMPUTATION */
        ...
      }
  /* Loop NEST 2 */
  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 benchmark suite. Border elements of zones are computed in function *exch-qbc*. Zones are processed sequentially by means of a series of computational phases where each one contains several loop nests that perform matrix-based computations.

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 computational phase. The *intra-zone* parallelism corresponds to the exploitation of the parallelism exposed by these loop nests. These loops can be annotated with OpenMP directives or transformed to device kernels.

Communication period: Intra-zone Parallelism

Figure 5 shows the code skeleton for the exchange of boundary values. In this period, zones are processed sequentially, so no inter-zone parallelism exists. However, processing each zone exposes some *intra-zone* parallelism. The computations are organized in the form of *memcpy* operations that pack/unpack border elements into temporary buffers (copy_face function) and then these are exchanged

```
/* EXCH_QBC */
for (zone=0; zone < num_zones; zone++) {
   east = adjacency_east[zone];
   north = adjacency_north[zone];
   copy_face(tmpEast, mesh[east], "IN",...);
   copy_face(tmpNorth, mesh[north], "IN",...);

   compute_border(mesh[zone], tmpEast, tmpNorth,...);

   copy_face(tmpEast, mesh[east], "OUT" ...);
   copy_face(tmpNorth, mesh[north],"OUT",...);
} /* zone loop */

void copy_face(...) {
  ...
  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 by means of a sequence of memcpy operations (copy_face function) that pack/unpack border elements to temporary buffers. The buffers are exchanged between adjacent zones (compute_border function).

between adjacent zones (compute_border function). These computations are implemented as parallelizable loop nests that can be annotated with OpenMP directives or transformed into device kernels.

Hybrid parallelization

Figures 6 and 7 depicts the parallel host and device code schemes for the NPB-MZ benchmark suite. The inter-zone parallelism is easily exploited by annotating the loop that traverses zones with an OpenMP *parallel for* directive. Exploiting intra-zone parallelism depends on the nature of the CUs: for CPU-based CUs, by means of additional OpenMP directives that parallelize loop nests within the different phases of zone processing; for GPU-based CUs, these loop nests must be transformed into kernel functions.

To simultaneously activate both devices and host, the codes in Figures 6 and 7 need additional support to identify whether an OpenMP thread corresponds to a CPU-based or to a GPU-based CU. In the context of hybrid executions, nested parallelism requires associating threads in an outer level of parallelism to a subset of the computational resources (i.e: a compute unit). Therefore, when these threads generate the inner level of parallelism, a decision has to be taken regarding whether this parallelism has to be mapped to device or host cores. We define that a thread diverges to a host compute unit if it deploys the inner level of parallelism to a host compute unit. Similarly, a thread diverges to a device compute unit if it offloads the inner level of parallelism to a device. This support is not available in the current specification of OpenMP 5.2 and neither exists in Nvidia's CUDA nor AMD's HIP runtime specifications. One main contribution of this paper is the design and implementation of the missing runtime support to effectively merge OpenMP with CUDA or HIP. Notice that new features in OpenMP like *contexts, traits* and *meta-directives* could be used to parallelize the loop nests using solely OpenMP directives. But these do not solve the problem of determining if a thread in the outermost level of parallelism (e.g.: inter-zone parallelism) has to be diverged to a GPU or CPU version for the innermost parallelism (e.g.:intra-zone parallelism). Even with meta-directives, OpenMP does not define what should be the association between threads and devices. Only a manual solution performed by the programmer can solve this limitation. Notice too that this association is essential for the work distribution to be applied between threads executing the outermost parallelism. In addition, if the application already is coded using CUDA or HIP primitives (e.g.: memory allocation, data transfers and kernel offloading), the programmer might want to utilize this version of the code but combined with OpenMP.

```
for (step=0; step < num_steps; step++) {
  /* BORDER EXCHANGE */
  exch_qbc(...);

  /* ZONE PROCESSING */
#pragma omp parallel for /* Inter-zone parallelism */
  for (zone=0; zone < num_zones; zone++) {
    comp_phase_1 (...);
    comp_phase_2 (...);
    ...
    comp_phase_N (...);

  } /* zone loop */
} /* Time step loop */

void comp_phase_N(...) {
#pragma omp parallel for /* Intra-zone parallelism */
  for (k=0; k < zdim; k++)
    for (j=0; j < ydim; j++)
      for (i=0; i < xdim; i++) {
        /* MATRIX BASED COMPUTATION */
        ...
      }
#pragma omp parallel for /* Intra-zone parallelism */
  for (k=0; k < zdim; k++)
    for (j=0; j < ydim; j++)
      for (i=0; i < xdim; i++) {
        /* MATRIX BASED COMPUTATION */
        ...
      }
  ...
}
```

Figure 6. Multi-level OpenMP parallel code. Outermost level of parallelism (inter-zone) is exploited by means of an OpenMP directive that parallelizes the loop that traverses the zones. Innermost level of parallelism (intra-zones) is exploited by means of OpenMP annotations for the loop nests that implement the compute phases of zone processing.

Hybrid execution requires to manage different memory spaces, so memory allocation and placement requires some guidance from the programmer. Consequently, at the programming model level, the programmer needs some explicit support to indicate when to allocate memory, where to allocate and, if necessary, to change the placement of that

```
__global__ kernel_1_comp_phase_N(...) {
  k = f_k(block grid);
  j = f_j(block grid);
  i = f_i(block grid);

  /* MATRIX BASED COMPUTATION */
  ...
}

__global__ kernel_2_comp_phase_N(...) {
  k = f_k(block grid);
  j = f_j(block grid);
  i = f_i(block grid);

  /* MATRIX BASED COMPUTATION */
  ...
}

void comp_phase_N(...) {
 kernel_1_comp_phase_N<<<grid, block, ... >>> (...);
 kernel_2_comp_phase_N<<<grid, block, ... >>> (...);

  ...
}
```

Figure 7. Device code for exploiting innermost parallelism (intra-zone). Loop nests corresponding to compute phases of zone processing have been transformed into device kernels.

memory. In the context of Figures 6 and 7, the programmer must ensure that memory allocation and placement has happened prior any work is assigned to the OpenMP threads that will be diverged to CPU or GPU execution. But this is highly entangled with the work distribution applied among the hybrid execution flows. Similarly as with the computing unit abstraction, we need additional runtime support to make possible the appropriate policies for memory placement according to the work-distribution schemes.

Given the inherent difference in compute power between host and devices, new scheduling schemes are necessary. Current specification of OpenMP supports static and dynamic schedulers. In this paper we implement variants of those to address the eventual unbalance caused by the different compute power of CPU and GPU cores.

Runtime support for OpenMP hybrid applications

This section describes the design of a runtime support that allows the programmer to develop hybrid OpenMP applications with CUDA/HIP. The design is divided into three runtime subsystems: a) *libCU-rtl* implements the necessary support to introduce the computing unit abstraction, b) *libPLACEMENT-rtl* implements the necessary support to manage with different address spaces, c) *libSCHEDULING-rtl* includes the implementation of additional scheduling schemes for hybrid applications.

The content of this section contributes to identify runtime primitives that can improve the overall programmability of hybrid OpenMP applications. It should be understood as a case of study for improvements in the OpenMP runtime support regarding the interoperability with GPU-oriented programming frameworks such as CUDA or HIP.

libCU-rtl

Computing units can be either CPU-based or GPU-based. For CPU-based CUs, they correspond to an aggregation of one or more CPUs. For instance, a CU can be formed with a pair of CPUs giving an organization that we identify as 1x2: 1 stands for one CU, the 2 identifies the number of CPUs in the CU. Similarly, CUs can be defined in the form of 1x4, 1x8 or whatever the CPU aggregation.

GPU-based CUs are defined only in the form of K+1 configuration where K CPUs are bounded to one GPU in order to orchestrate all actions over the GPU: memory allocation, data transfers and computation offloading. For this work, the most common usage cases have been covered with hybrid configurations in the following form: (N x M + G) CPUs where N stands for the number of CPU-based CUs executing with M CPUs, and G stands for the number of GPU-based CUs. All CPU-based CUs are composed of the same number of CPUs and GPU-based CUs are managed by solely one thread per GPU (e.g.: K=1). For these cases, a total number of NxM + G threads are necessary for execution.

Threads are created using the OpenMP programming directives. For thread-CU association, we rely on the thread affinity support already existing in OpenMP. The environment variables *OMP_PROC_BIND* and *OMP_PLACES* control the thread affinity to the actual CPUs. Our CU runtime support relies on correctly setting these variables to ensure the particular thread-CU association that will allow the simultaneous activation of CPUs and GPUs.

```
OMP_PLACES="{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11},
            {12, 13, 14, 15}, {16}, {17}, {18}, {19}"
OMP_PROC_BIND="close, master"
```

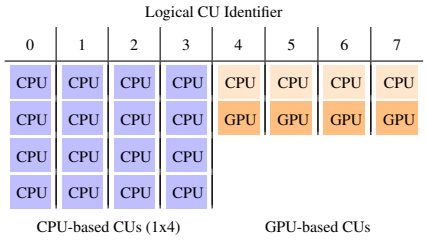

Figure 8. Example of an 8 CUs organization: 4 CPU-based CUs composed each one of 4 CPUs, 4 GPU-based CUs composed of 1 CPU and 1 GPU. This configuration needs 20 physical CPUs. OpenMP thread binding is used to configure the thread-CU association. OpenMP variables *OMP_PLACES* and *OMP_PROC_BIND* are set to achieve the desired configuration.

Figure 8 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. Besides, the CPU aggregation is achieved by 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. Table 3 shows the appropriate values for these variables in a OpenMP compact form for some particular CU configurations.

Signature	Description
`unsigned int nCUS()`	Get the current number of active CUs.
`unsigned int nGPUS()`	Get the number of GPU-based CUs.
`unsigned int nCPUS()`	Get the number of CPU-based CUs.
`unsigned int nInnerCPUs()`	Get the number of CPUs contained in the CPU-based CU associated to the invoking thread. Returns 0 if thread corresponds to a GPU-based CU.
`bool isGPU(()`	Check if invoking thread is associated to a GPU-based CU.
`bool isCPU()`	Check if invoking thread is associated to a CPU-based CU.
`unsigned int getGPUid()`	Get the device ID for the CU associated to the invoking thread. Returns error (-1) if the invoking thread is not associated to a GPU-based CU.
`unsigned int getCPUid()`	Get the OpenMP thread ID for the CU associated to the invoking thread. Returns (-1 limits::max) if the invoking thread is not associated to a CPU-based CU.
`unsigned int getCUid()`	Get the CU ID, from 0 up to the number of CUs minus one.
`void synchronize()`	Synchronize with the device corresponding to the CU associated to the invoking thread. In case the CU corresponds to a CPU, no synchronization happens.

Table 2. Runtime primitives of libCU-rtl

	Configuration	OMP_PLACES
OpenMP Thread Binding	1x2 + 4	”{0:2}:1:1, {2}:4:1”
	1x4 + 1	”{0:3}:1:1, {4}:1:1”
	4x2 + 1	”{0:2}:4:2, {8}:1:1”
	7x2 + 2	”{0:2}:7:2, {14}:2:1”
	3x4 + 4	”{0:4}:3:4, {12}:4:1”

Table 3. Examples for OMP_PLACES variable and thread binding control in hybrid execution using the OpenMP programming model.

Within the libCU-rtl, 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 libCU-rtl, CPU-based CUs are identified from 0 up to NCPUS-1 where NCPUs is the number of 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 similarly to the OpenMP programming model.

Table 2 lists the runtime primitives implemented to operate with 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, and how many of them correspond 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. Section describes in detail how all these primitives have been used to code a hybrid version of the NPB-MZ benchmark suite.

libPLACEMENT-rtl

OpenMP hybrid applications require additional runtime support to control the data sharing between the host and device address spaces e.g.: data sharing happens at system memory page level). Memory allocation and placement, and eventual data transfers have to be guided by the programmer in order to achieve satisfactory levels of performance. For instance, CUDA runtime primitives to manage unified virtual memory allow programmers to provide hints for data placement and usage in computational kernels (González and Morancho 2021; NVIDIA 2023). For hybrid OpenMP applications, similar support is needed and should be available in accordance to the OpenMP conventions for parallelism exploitation. The role of the *libPLACEMENT-rtl* is to bridge the OpenMP programming model with the existing support to control the memory allocation, placement and if necessary, migration.

Guidance from the programmer is needed so that when a computation is assigned to a CU, the data associated to the computation is placed on the corresponding CU memory space. This requires memory allocation and, potentially, data migration prior the actual computation is executed. Usually, for hybrid applications, programmers introduce a pre-computed work distribution and memory allocation before any computation takes place. This happens in the initialization phases of the application. In contrast, the libPLACEMENT-rtl includes the necessary runtime support so that memory allocation, placement and migration can happen at any time of the execution.

Within the libPLACEMENT-rtl, memory spaces are identified with an integer value that ranges from 0 up to NAS-1, where NAS stands for the total number of memory spaces in use. Host address space is identified with value 0, and the devices' memory spaces are numbered from 1 up to the number of GPU-based CUs.

The main data structure for the libPLACEMENT-rtl is a map between key values and actual memory addresses belonging to any of the different address spaces. A *key* corresponds to a host memory address that identifies uniquely a memory region. Programmers can register memory regions using the primitive *registerMem*. Along the application execution, programmers can allocate, deallocate, copy and migrate memory regions using key values and CU identifiers that identify memory regions in a particular CU. The libPLACEMENT-rtl strictly gives support for this functionalities and no memory consistency mechanisms are supported. The replicas and actual state of data is totally under the programmer responsibility. The runtime distinguishes between *memory placement* and *memory allocation*. Memory regions only have one active placement, although they might have been allocated in several memory spaces. Only migration actions change the placement, while allocation or copy do not. Besides,

Primitive	Description
`unsigned int registerMem(void* key, unsigned int size)`	Register within the runtime a memory region identified by *key* of *size* bytes.
`unsigned int getPlacement(void* key)`	Get the memory space ID for the input key, from 0 up to the number of active memory spaces for the application.
`void* alloc(void* key, unsigned int CU)`	Perform memory allocation on the given CU memory space for a memory chunk. Size is determined by the previous registration of the *key* value associated to the memory region. Return value corresponds to the pointer to memory region.
`void* dealloc(void* key, unsigned int CU)`	Perform memory deallocation on the given CU memory space for a memory chunk.
`void* mem_addr(void* key, unsigned int CU)`	Get base address of the registered memory region associated to *key* within the *CU* memory space.
`unsigned int migrate(void* key, unsigned int to)`	Perform memory migration from the current placement of data (*key*) to the destination memory space.
`unsigned int copy(void* key, unsigned int from, unsigned int to)`	Perform a memory copy from the origin memory space to the destination memory space.
`bool isGPU(unsigned int AS)`	Check if the memory space AS corresponds to a device memory space.
`bool isCPU(unsigned int AS)`	Check if the memory space AS corresponds to host memory space.
`unsigned int getGPUid(unsigned iny AS)`	Get the device ID for the AS. Returns error (-1) if the AS is not associated to a GPU-based CU.

Table 4. Runtime primitives of libPLACEMENT-rtl

primitives that involve data transfers between CUs are not optimized regarding the overlapping of communication and computation. In general, the primitives in libPLACEMENT-rtl expect that once memory regions have been allocated, then the application rarely will require memory migration due to changes in the work distribution among the CUs. But if this were the case, the runtime supports this feature. Section describes the hybrid implementation of the NPB-MZ benchmark suite and shows that this has not represented a limitation for the hybrid parallelization and execution.

Table 4 describes the runtime primitives that introduce the placement 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 address spaces are in use and check if an address space identifier corresponds to a host or device address space. For managing the memory allocation, the runtime support includes primitives to *alloc* and *copy*. In this regard, the alloc and de-alloc operations are mapped on the already existing primitives in the native OpenMP runtime support, but execute additional control code to monitor the allocated memory regions. For placement management, the runtime supports one main primitive to *migrate* memory regions. Section describes in detail how all these primitives have been used to code a hybrid version of the NPB-MZ benchmark suite.

libSCHEDULING-rtl

Hybrid applications require work distribution schemes that adapt to the different computing power of the CUs. We have implemented a runtime library with generic support to implement different schedulers. The implementation follows an object-oriented paradigm, with virtualization of methods that need to be redefined by the programmer to implement new schedulings.

Within the libSCHEDULING-rtl, the input parameters for the scheduler are the number of tasks, the number of CPU-based CUs and the number of GPU-based CUs. All schedulers assume that the tasks are identified by an integer value ranging from 0 up to NTASKS-1 , where NTASKS stands for the total number of tasks to be executed.

Table 5 describes the runtime primitives that allow the programmer to define and operate with the schedulers. The primitives also give support to the programmer to monitor execution times for tasks and also for CUs. Section describes in detail how these primitives have been used to code a hybrid version of the NPB-MZ benchmark suite.

Table 6 describes the main data structures that have been used to implement the scheduling support. The runtime system records how many tasks have to be scheduled, how many CUs are available, how many of them correspond to CPU-based CUs and how many correspond to GPU-based CUs. The runtime has the ability to memorize the task assignment between the CUs, so that if a computation is repeated several times, the scheduling is only applied once and later reused across the computation instances. For this purpose, the runtime uses several vectors to retain the the task-CU assignment (e.g.: vectors *mTaskCU, mTaskHowmany*). Along the execution of the application, CUs require tasks to be executed (e.g.: *getTask*). This corresponds to the main virtual method. A programmer can redefine this method to implement dynamic work-distributions schemes. CUs notify that a task starts execution (*executeTask*) and communicate to the runtime system that a task has been completed (*commitTask)*. These primitives are used to monitor task execution time and total CU execution time (vectors *mTimersPerCU* and *mTimersPerTask*).

Within libSCHEDULING-rtl we have implemented variants of two state-of-the-art schedulers: Static and Dynamic Task scheduling. The following subsections describe each scheduling in detail. In the context of this work, these schedulers represent a case of study to evaluate the support that the *libSCHEDULING-rtl* offers to enable user-defined schedulers. Hence, this section should not be understood as comprehensive study on work scheduling for heterogeneous systems.

SCHEDULING::Static

This scheduling assigns the set of tasks evenly among the CUs: the total number of tasks is divided by the number of CUs. The same number of consecutive tasks (according to the tasks numbering) is assigned to each CU. If the number of CUs does not evenly divide the number of tasks, then

Signature	Description
`void commitTask(unsigned int task)`	Commit the task executed by the CU associated to the invoking thread.
`unsigned int getTask()`	Get a task assigned to the CU associated to the invoking thread. Returns NOTASK when runs out of pending tasks.
`void executeTask(unsigned int task)`	Start execution of the task assigned to the CU associated to the invoking thread.
`unsigned int getCUid(int Task)`	Get the CU identifier that executes the task.
`void setTaskTime(unsigned int task, float Time)`	Set the time sample for a task.
`void getTaskTime(unsigned int task)`	Get the time sample for a task.
`void setCUTime(int CU, float Time)`	Set the time sample for a CU.
`void getCUTime(int CU, float Time)`	Get the time sample for a CU.
`bool isDynamic()`	Check if scheduling is a variant of a dynamic scheduling.
`bool isStatic()`	Check if scheduling is a variant of a static scheduling.
`void printScheduling()`	Output the assignment between CUs and tasks.
`void printTaskTimers()`	Output the time samples for tasks.
`void printCUTimers()`	Output the time samples for CUs.

Table 5. Runtime primitives of libSCHEDULING-rtl

Type Name	Description
`unsigned int mNtasks`	Number of tasks to be scheduled.
`unsigned int mNcpus`	Number of CPU-based CUs.
`unsigned int mNgpus`	Number of GPU-based CUs.
`std::vector<int> mTaskCU`	Vector of size *mTasks* containing the CU identifier that executes each task.
`std::vector<int> mTaskHowmany`	Vector of size mNcpus+mNgpus containing the number of tasks assigned to each CU.
`std::vector< std::vector<int> > mCUTasks`	Vector of size *mNcpus+mNgpus*. Each element is a vector containing the sequence of tasks assigned to a CU.
`std::vector<bool> mCommitTasks`	Vector of size *mNtasks* that indicates if a task has been completed.
`std::vector<unsigned int> mIndexTask`	Vector of size mNcpus+mNgpus containing an index value for the task being executed by a CU. This value is used to access the corresponding vector within the *mCUTasks* vector.
`omp_lock_t mLock`	Mutex used for mutual exclusion to support dynamic schedulers.
`unsigned int mChunk`	Chunk parameter applied to the dynamic scheduler.
`std::vector<float> mTimersPerCU`	Vector of size mNcpus+mNgpus to store the execution time for a CU.
`std::vector<float> mTimersPerTask`	Vector of size *mTasks* to store the execution time for a task.
`std::vector<unsigned int> mIndexFirstTask`	Vector of size mNcpus+mNgpus to store first task assigned to a CU. The scheduler assumes a sequence of consecutive tasks being assigned to each CU.
`std::vector<unsigned int> mIndexLastTask`	Vector of size mNcpus+mNgpus to store last task assigned to a CU. The scheduler assumes a sequence of consecutive tasks being assigned to each CU.

Table 6. Data structures of libSCHEDULING-rtl

an additional task is assigned to CUs with identifiers in the range [0, NREMAIN-1] where NREMAIN is the number of remaining tasks. This scheduler behaves exactly as the STATIC scheduling included in the OpenMP specification for loop level parallelism.

SCHEDULING::STATIC-hybrid

This scheduler is a variant of the Static scheduler. According to a pre-computed parameter, the Performance Conversion Factor (PCF), the scheduler divides the set of the tasks into two subsets: one for the CPU-based CUs, the other for the GPU-based CUs. Then the scheduler applies a Static scheduling per each subset, separately assigning tasks to CPU-based CUs and GPU-based CUs. The PCF is the ratio between the execution time needed to process a task by a CPU-based CU and by a GPU-based CU according to the hybrid configuration. For instance, a PCF_{1x2} equal to 2.5 indicates that the execution time needed by a CU composed by 2 CPUs to process a task is equal to 2.5 multiply by the time needed by one GPU-based CU (e.g.: one GPU). Similarly, PCF_{1x4} is the ratio between the execution times of a CU composed by 4 CPUs and a GPU-based CU. In general, this scheduler needs the PCF values to be pre-computed; in our implementation, through profiling techniques.

Given T tasks, this scheduler assigns to the set of GPU-based CUs the number of tasks: T_{gpu} = T / (G x PCF + C), where C corresponds to the number of CPU-based CUs, G corresponds to the number of GPUs. If the division generates a remaining of tasks, these are grouped in GxPCF groups and assigned to the GPUs. The number of tasks assigned to the CPU-based CUs is T_{cpu} = T - T_{gpu}. Once the T_{gpu} and T_{cpu} values are computed, then the scheduler applies a Static scheduling to each set according to the number of CPU-based and GPU-based CUs. The reasoning behind this arithmetic is the following: dividing the number of tasks by the value (G x PCF + C), defines the number of groups that can be formed with sufficient work to distribute among CPU-based and GPU-based CUs in a uniform, balanced way. The balancing occurs because of the PCF parameter in the expression. For instance, if T=10, G=1, C=1 and PCF=3, then we want to form groups of 4 tasks (4=GxPCF+C), because 3 of them will be mapped to a GPU, one will be mapped to a CPU-based CU. This results in balanced distribution, according to the PCF value. We count how many of these groups can be formed by computing T / (G x PCF + C).

This STATIC-hybrid scheduling has several similarities with previous heterogeneous work-schedulings in Scogland et al. (2012, 2014). In these works, performance models are introduced to estimate the execution time of a task on the available computing resources (e.g.: CPUs/GPUs). The estimates are generated dynamically, through regression techniques. The STATIC-hybrid is very similar to that of the heterogeneous work-scheduling in Scogland et al. (2012,

2014). Given the estimates, the tasks are assigned to the compute units in a similar fashion. The essential difference is that the STATIC-hybrid scheduler needs of an off-line profiling step to generate samples of execution times for tasks and compute units. The performance comparison factor is derived from these samples, which is the key element in the STATIC-hybrid scheduling to decide where to place a particular task. In general, this approach is less general that those based on dynamic techniques, but in the context of scientific applications it is yet a feasible and effective technique. The impact of the profiling step is evaluated in Section where we describe how much time is needed and what are the programming efforts that it requires.

SCHEDULING::Dynamic

This scheduler assigns tasks to CUs dynamically. It divides the set of tasks into chunks of *chunk* consecutive tasks according to the task numbering; where *chunk* is a parameter of the scheduler. A chunk is assigned to each CU and as soon as a CU finishes processing its chunk, the scheduler assigns a new chunk to the CU. This scheduling behaves as a task scheduler where the chunk parameter determines the task granularity.

The *chunk* parameter can smooth the effect of CU synchronizations to acquire chunks of work. Besides, this scheduler is able to memorize the task-CU assignment. The runtime system keeps track of which tasks executed each CU. From one execution of the scheduled computation, to the next instance of the same computation, the scheduler ensures that the same task scheduling is applied. Thus, no data migration occurs when the parallelism execution is repeated. Only the first instance of the work execution suffers from overheads related to memory registering, allocation and/or data transfers associated to the data communication between the CUs. This data locality preservation is only effective if no other computation under a different scheduling moves data between host and devices. This scheduler behaves exactly as the DYNAMIC scheduling included in the OpenMP specification for loop level parallelism, but with the additional property of memorizing the task assignment from one execution to the next one.

Design and Implementation of Hybrid NPB-MZ

This section describes the hybrid design and implementation of the NPB-MZ benchmark suite using the runtime support described in the previous section.

Multi-Level Hybrid Parallelization

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

Computation Period

Figure 9 shows the OpenMP implementation for a hybrid parallel strategy for the *inter-zone* parallelism. The original loop responsible for traversing the zones (Figure 4) has been substituted by a parallel region. The parallel region is executed with as many threads as CUs are used for the hybrid execution. The code executed by all threads corresponds to a *while* loop where the primitives for acquiring, executing and committing tasks are inserted. Notice the usage of the *migrate* primitive to ensure that the memory allocation is performed prior any attempt is done to process a zone (e.g.: task). In this case, the task identifier coincides with the zone index to be 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 different computation phases are executed by a host CU or device CU according to the nature of the thread. Notice that this code structure is independent from the applied scheduling. In Figure 9 a static scheduling is used, but it could be substituted by a dynamic or hybrid scheduling.

Figure 10 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 OpenMP directives. These inner parallelism is exploited by as many threads as the output of the primitive *nInnerCPUs*. Thus, this innermost parallelism is deployed over the CPUs associated to the host CU. This is ensured because OpenMP thread affinity is set so that appropriate *places* are described through `OMP_PLACES` and `OMP_PROC_BIND` environment variables. If the thread corresponds to a device CU, then the execution is diverged to the invocation of the kernels that codify the 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 *getGPUid* primitive).

```
Static(num_zones, CU::nGPUS(), CU:nCPUS());

#pragma omp parallel num_threads(CU::GPUS()+CU:CPUS())
{
  task = Static::getTask();
  while (task != NO_TASK) {

    PLACEMENT::migrate(mesh[zone],...);

    Static::executeTask(task);

    /* ZONE PROCESSING */
    zone = task;
    comp_phase_1 (...);
    comp_phase_2 (...);
    ...
    comp_phase_N (...);

    if (CU::isGPU()) CU::synchronize();

    Static::commitTask(task);
    task = Static::getTask();
  }// while
} // parallel
```

Figure 9. Code transformation for exploiting inter-zone parallelism. Original parallel loop is transformed into a parallel region containing a *while* loop. Threads iterate and invoke the runtime primitives to acquire tasks and execute them. The scheduling to be applied is determined by the call to the primitive *Static*.

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. Process a zone

```
void comp_phase_N(...)
  if (CU::isCPU()) {
    NT = CU::nInnerCPUs();

#pragma parallel for num_threads(NT)
    for (k=0; k < zdim; k++)
      for (j=0; j < ydim; j++)
        for (i=0; i < xdim; i++) {
          /* MATRIX BASED COMPUTATION */
          ...
        }
  } // if
  else if (CU::isGPU()) {
    HipSetDevice(CU::getGPUid());
    kernel_1_comp_stage_N<<<grid, block, ... >>> (...);
  } // else
   ...
}
```

Figure 10. Code transformation for exploting intra-zone paralellism. Host CUs are diverged to loop nests annotated with loop level parallelism directives. This innermost level of parallelism is executed by the CPUs associated to the host CU; the number of threads is determined by the call to *nInnerCPUs*. Device CUs are diverged to kernel invocations.

```
for (zone=0; zone<NUM_ZONES; zone++) {
  east_zone = adjacency_east[zone]
  north_zone = adjacency_north[zone];

  zonePlacement = PLACEMENT::getPlacement(mesh[zone]);

  copy_face(tmpEast[zonePlacement], mesh[east_zone],
                                    "IN", ...);
  copy_face(tmpNorth[zonePlacement], mesh[north_zone],
                                    "IN", ...);

  if (PLACEMENT::isGPU(zonePlacement)) {
    HipSetDevice(PLACEMENT::getGPUid(zonePLacement));
    gpu_compute-border(mesh[zone],
                       tmpEast[zonePlacement],
                       tmpNorth[zonePlacement]);
  }
  else if (PLACEMENT::isCPU(zonePLacement)) {
    cpu_compute_border(mesh[zone], tmpEast[zonePlacement],
                               tmpNorth[zonePlacement]);
  }

  copy_face(tmpEast[zonePlacement], mesh[east_zone],
                                    "OUT", ...);
  copy_face(tmpNorth[zonePlacement], mesh[north_zone],
                                    "OUT", ...);
}
```

Figure 11. Hybrid implementation of the exchange boundary computation. Host and device versions of the compute border computation are introduced. Zone placement is checked and border computation happens where the zone resides.

exposes some amount of parallelism that can be exploited through fine-grain parallelism.

Figure 11 depicts the code scheme for the *Communication Period*. Zone faces containing border elements are copied to temporary buffers (e.g.: calls to *copy-face procedure*). Then, zone placement is checked to perform the border exchange computation on a device or host accordingly.

The implementation of the *copy-face* procedure now is more complex because buffering of border elements is conditioned by the fact that buffers and adjacent zones can

```
void copy_face(void* Buffer, void* Zone, string Dir, ...) {
  zonePlacement = PLACEMENT::getPlacement(Zone);
  bufferPlacement = PLACEMENT::getPlacement(Buffer);
  if (Dir == "IN") {
    if (PLACEMENT::isGPU(zonePlacement) &&
        PLACEMENT::isGPU(bufferPlacement)) {
      HipSetDevice(PLACEMENT::GPU(zonePlacement));
      gpu_copy_face<<<grid, block, ...>>>(...);
      HipMemCpyPeer(Buffer, ...,);
    }
    if (PLACEMENT::isGPU(zonePlacement) &&
        PLACEMENT::isCPU(bufferPlacement)) {
      HipSetDevice(PLACEMENT::GPU(zonePlacement));
      gpu_copy_face<<<grid, block, ...>>>(...);
      HipMemCpy(Buffer, ...,HipDeviceToHost);
    }
    if (PLACEMENT::isCPU(zonePlacement) &&
        PLACEMENT::isGPU(bufferPlacement)) {
      cpu_copy_face(...);
      HipSetDevice(PLACEMENT::GPU(bufferPlacement));
      HipMemCpy(Buffer, ...,HipHostToDevice);
    }
    if (PLACEMENT::isCPU(zonePlacement) &&
        PLACEMENT::isCPU(bufferPlacement)) {
      cpu_copy_face(...);
    }
  } // if Dir == "IN"
  else if (Dir == "OUT") {
     ...
  } //if Dir == "OUT"
}
```

Figure 12. Hybrid implementation of the copy_face computation. Placement checks are introduced for buffers and 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.

reside on different address spaces. Figure 12 shows the code skeleton for this procedure. Notice the structure of *if* statements that cover the 4 possibilities: buffer resides on host/device, zone resides on host/device. Notice that, for data transfer, AMD's HIP primitives appear to move the data acrros the host and devices. This communication arises according to the memory footprint determined by the applications, which in turn is totally conditioned by the scheduling applied during the *Computation Period*. The effect of this communication is evaluated in Section .

Programmability Assessment

The runtime support described in this section has made possible the deployment of a methodology to enable hybrid executions mixing OpenMP and CUDA/HIP. In terms of programmability, we have introduced the abstractions of the *Computing Unit, Data Placement and Scheduling* 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.

The following subsections compare our proposal versus existing solutions that achieve similar levels of programmability mixing different programming models with CUDA/HIP.

Comparing with MPI + CUDA/HIP

As different devices in a compute node might or might not share their memory spaces, several works have tried to achieve a hybrid parallelization mixing distributed memory paradigms such as MPI and CUDA/HIP (Karunadasa and Ranasinghe 2009; Peña et al. 2020; Kraus 2013; Awan et al. 2019; Jacobsen et al.). These solutions require similar runtime primitives as those explored in this work. In MPI (Message Passing Interface Forum 1994), 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).

Comparing with OpenMP/OmpSS

OmpSS (Elangovan et al. 2013; Duran et al. 2011; Bueno et al. 2012) is a task-based programming model that can effectively support the hybrid parallel strategy described for the NPB-MZ benchmark suite. The inter-zone parallelism can be supported through task definition directives, and data associated to tasks can be specified through the usage of *in, out* clauses to describe producer consumer relations. The entanglement between the task scheduling and the data placement is solved using this information. Compared to our proposal, both require the programmer to identify which data is associated to each task, however the programming efforts are simpler in OmpSS than in our proposal.

For OpenMP, the similarities and limitations are the same. OpenMP supports task-level parallelism. The common usage of a *parallel* directive combined with a *single* directive for task definition can be used for the inter-zone parallelization. But OpenMP does not allow the programmer to control how tasks have to be mapped to compute units. This combination can not be accompanied with the specification of a particular scheduling. Therefore, similar runtime functionalities as those proposed in this paper would be necessary to deploy a reasonable interoperability between OpenMP and CUDA/HIP. Specially for data placement and work scheduling. Notice that using *target* and meta-directives to offload computations to devices does not solve the problem of whether at the inter-zone level of parallelism, decide where to execute a task. The *target* support just solves the device code generation, not the scheduling and data placement problems.

The CoreTSAR proposal (Scogland et al. 2014) addresses the limitations of these programming models. In particular, it identifies the missing runtime functionalities to simultaneously deploy task-based parallelizations with appropriate task scheduling, including a reasonable programming level regarding the data placement guidance. Both CoreTSAR and the runtime described in this paper share some functionalities, being the main difference the fact that CoreTSAR does not target the interoperability for OpenMP and CUDA/HIP. CoreTSAR complements the OpenMP standard to enable OpenMP strategies for heterogeneous programming. Besides, CoreTSAR scheduling is based on dynamic techniques, in contrast we explore offline profiling techniques to define an appropriate task scheduling.

Evaluation

This section evaluates the overall performance of our hybrid implementation of the NPB-MZ benchmark suite. Applications have been coded and compiled within the ROCm-3.5.0 framework and *llvm 12* compiler suite. The CPU code has been compiled combining a C++ NPB-MZ implementation (Dümmler and Rünger 2013) and the original NPB-MZ Fortran implementation (der Wijngaart and Jin 2003) to generate a version compatible with the ROCm implementation of the applications. All experiments have been performed in a system composed of AMD EPYC 7742 @ 2.250GHz (64 cores and 2 threads/core, totalling 128 threads per node) and 2 x GPU AMD Radeon Instinct MI50 with 32GB. We run class D NPB-MZ benchmarks (Table 1), so the input mesh is composed by 16 zones (LU-MZ) or 1024 zones (BT-MZ and SP-MZ); memory usage is 13 GB.

Overall Performance

The basis for the performance evaluation is the comparison between four versions of the benchmarks in the suite: GPU-based and CPU-based non-hybrid versions already studied in previous studies for the NPB-MZ benchmark suite (Duran et al. 2005; Gonzalez and Morancho 2020; González and Morancho 2021) and two hybrid executions under two work-distribution schemes. On one hand, a static scheduling that corresponds to the default and only supported scheduling for hybrid executions in the latest OpenMP specification (OpenMP 5.2). On the other hand, we include a dynamic task scheduling with the ability of memorizing the task-CU assignment across different instances of the scheduled computation. The comparison between the static and dynamic hybrid versions exposes the current performance limitations in OpenMP for hybrid executions.

SP-MZ benchmark:

Figure 13 shows the speedup achieved by several parallel configurations with respect to the single CPU configuration in benchmark SP-MZ. Leftmost bars correspond to executions under a STATIC scheduling. We take the performance of this scheduler as a performance reference. From left to right, first CPU-based configurations range from 16 to 64 threads exploiting the *inter-zone* parallelism (e.g.: for 64x1 configuration, each thread processes 16 zones). Maximum speedup is observed with 32 threads: 19.71x. Then, with 48 and 64 threads, the performance drops significantly (up to 18.23x and 13.26x respectively). This trend will also appear in the hybrid configurations. Table 7 shows average execution time per zone under different configurations. For SP-MZ, the first thing to notice is that for 1-GPU, processing one task (e.g: zone) takes 0.83ms on average. In contrast, with 1-CPU, it takes 8.70ms. When

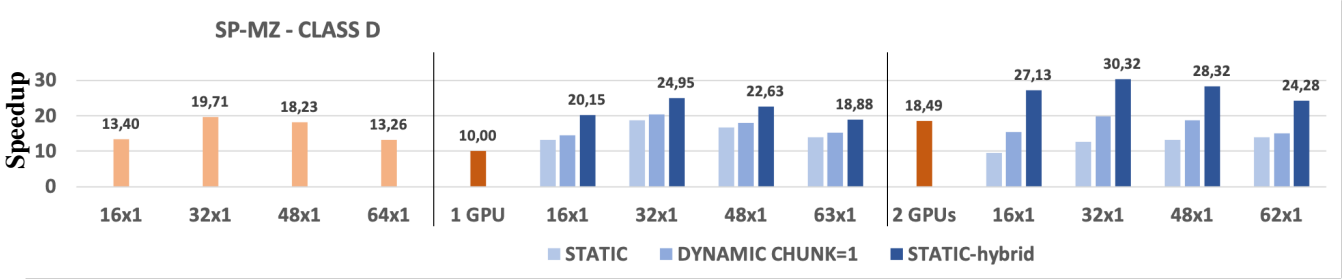

Figure 13. Benchmark SP-MZ with input CLASS D. Overall speedup for different configurations: from left to right, CPU only configurations, 1 GPU configurations and 2 GPU configurations. CPU-based CUs are defined as BxT, where B corresponds to CPU-based CUs executing with T CPUs each. Speedup is measured with respect to single CPU configuration.

more CPUs are added, the time for processing one tasks increases. Notice how for a non hybrid 32x1 configuration the average task processing time is 11.25ms, but for 64x1 is 36.38ms: more than 3x of slowdown. Consequently, CPU-based versions 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 main reason for this is that last level cache memory has a small capacity compared to the input data size: 256 MB in contrast to 13GB (see Table 1). 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. This aspect is essential to explain the performance of hybrid configurations.

Conf.	SP-MZ	BT-MZ	Conf.	LU-MZ
1-GPU	0.83	2.96	1-GPU	38.44
1x1	8.70	31.11	1x1	1653.38
16x1	9.00	64.75	1x16	258.94
32x1	11.25	69.19	1x32	98.13
48x1	18.61	87.66	1x48	88.44
64x1	36.38	102.50	1x64	38.44

Table 7. Average task time (ms) for all NPB-MZ applications with different CU configurations : BxT where B stands for the number of CUs and T stands for the number of threads in each CU, non hybrid 1-GPU and 2-GPU (B=0).

Conf.	SP-MZ	BT-MZ	Conf.	LU-MZ
1-GPU	1	1	1-GPU	1
16x1	10.89	21.91	1x16	6.74
32x1	13.62	23.41	1x32	2.55
48x1	22.52	29.66	1x48	2.30
64x1	44.03	34.69	1x64	2.16

Table 8. PFC values used per each application and configuration. Values are obtained from the relation between task execution times in 1 GPU and B x T CPUs configurations (BxT where B stands for the number of CUs and T stands for the number of threads in each CU).

Hybrid configurations with 1 GPU show a similar trend. 1-GPU configuration speeds-up by 10x. Hybrid configurations under the STATIC scheduling do not outperform CPU-based configurations. For instance, 32x1+1 configuration speeds-up by 18.72x, below the performance of the 32x1 configuration (19.71x). This is due to the way the 1024 zones are distributed among CU's. For 32x1 configuration, there are 32 CU's and the scheduler assigns 32 zones to each one. For the 32x1+1 configuration, there are 33 CUs, so the largest piece of work still is a 32-zone work unit. For hybrid configurations with higher count of CPU-based CUs, adding one or two GPUs does not significantly reduce the work assignment per each CU. The STATIC scheduler does not take into account the differences between the CUs respect their compute power.

Figure 13 shows the performance of hybrid configurations under a DYNAMIC scheduler with CHUNK=1. Notice that now configurations 32x1+1 and 32x1+2 speedup by 20.38x and 19.88x respectively. Both outperform the maximum performance observed for CPU-based configurations. But we have observed that the speedup for hybrid configurations under the DYNAMIC scheduler is limited by the *Communication Period*. For the SP-MZ, the input working set is composed of *many* and *very small* zones where the ratio between the number of boundary elements and the total number of zone elements makes that a very high count of elements are processed within the border computation. This becomes even worse when we combine the memory spaces for host and devices, slowing down its execution time. Zones are dynamically distributed among the CUs. This makes that adjacent zones now are placed in different memory spaces. Therefore, data transfers are necessary to compute the exchange of boundary values. In particular we have observed execution times for the *Communication Period* that change from 97ms for CPU-based configurations to 140ms for hybrid configurations with 1 GPU, and 200ms with 2 GPUs. The *Computation Period* ranges from 576ms to 280ms depending on the number of active CUs in the configuration. Therefore, speedups for the *Computation Period* are considerably higher than those observed for the overall performance. For instance, for hybrid configurations with 2 GPUs, we have observed speedups that range between 21x and 31x. But the increments in the execution times for the *Communication Period* limit the overall performance of the application. In this regard, one particular aspect of the DYNAMIC scheduling is that of memorizing the task-CU assignment from one execution instance to the next one. In the context of hybrid architectures with different physical memory spaces, this aspect becomes essential to ensure that in phases where data is interchanged between tasks, the communication overheads do not become a performance bottleneck. We have checked that if the same scheduling is applied but removing the memorizing ability (e.g., each instance of the *Computation Period* remaps the task-CU

assignment), then both the *Computation Period* and the *Communication Period* suffers from slowdown factors between 10x-15x.

The STATIC-hybrid scheduler is based on a performance conversion factor (PCF) between the GPUs and the CPUs. In this case, the PCF value for each configuration has been obtained from the data in Table 7. We divide the average task execution under a CPU and non hybrid configuration by the average task execution time when executed in one GPU. This should describe how faster a GPU is computing one task compared to CPU-based execution with 16, 32, 48 and 64 CPUs. Table 8 exposes the PCFs values for each configuration, directly computed from Table 7. In general, collecting the data in Tables 7 and 8 requires the execution of several iterations of the benchmarks. In our case, 5 iterations were sufficient to observe steady execution times per task on both the GPU case and the CPU case. Overall this corresponded to experiments that needed less than 10 minutes for all 3 benchmarks in the NPB-MZ suite.

For the STATIC-hybrid scheduling, we observe that the work imbalance caused by the different nature of the CUs is solved. Even more, the scheduler tends to assign adjacent zones to the same CU. Therefore, this scheduling balances the execution without incurring in communication overheads related to the boundary values exchange. This explains the observed performance levels: speedup values range between 20x and 30x, outperforming the single GPU configurations by factors ranging from 1.75x and 2.5x. Although these improvements, both the 1-GPU and 2-GPU hybrid configurations expose a performance degradation with more than 32 CPUs. This fact is explained by the data in Table 7. As it was described previously, the zone processing time is not constant, with increments of more that 3x of slowdown when doubling the number of available CPU-based CUs. This explains that for hybrid configurations with more than 32 CPUs the STATIC-hybrid scheduling is not able to perform a zone-CU assignment that keeps improving the performance as the total count of CUs is increased. In the context of heterogeneous computing, the STATIC-hybrid corresponds to a a state-of-the-art scheduling aligned to most recent advances in heterogeneous scheduling. The reference works (Scogland et al. 2012, 2014; Elangovan et al. 2013; Duran et al. 2011) explore similar approaches based on estimates for the execution time in different types of compute units. In particular, the proposal Scogland et al. (2014) is based on dynamic techniques that implement a regression based on architectural parameters. In contrast, the STATIC-hybrid is based on offline data obtained through profiling techniques. In this regard, we consider a the STATIC-hybrid scheduling as a variant for a reference scheduling in heterogeneous computing.

LU-MZ benchmark: Figure 14 shows the speedup achieved by several parallel configurations with respect to the single CPU configuration in benchmark LU-MZ. From left to right, the performance of CPU-based configurations are shown using from 1 to 64 threads exploiting the *intra-zone* parallelism (e.g.:for a 1x32 configuration, each zone is processed by 32 threads, one zone after the other). Maximum speedup is observed with 64 threads: 19.78x. In general, from 16 CPU to 64 CPU configurations, there is poor scalability, although speedup increases with the increment of CPUs. This trend will also appear in the hybrid configurations. As it has been observed in the previous subsection, the main reason for this is that last level cache memory has a small capacity compared to the input data size: 256 MB in contrast to 13GB (see Table 1). This makes that the CPU-based versions 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 (see Table 7). The pressure over the memory subsystem is different in each configuration, diminishing the effect of the utilization of high counts of CUs.

Hybrid configurations show a very different range of speedups. Single GPU configurations delivers an impressive speedup of 42.91x. Configurations under the STATIC scheduling improve the speedup as the number of CPU-based CUS increases, reaching a maximum speedup of 48.80 for the 1x48+1 configuration. The STATIC distribution does not take into account the differences in compute power of the CUs. The LU-MZ operates over a mesh of 16 zones. In all 1-GPU hybrid configurations the number of CUs are 2. This means that 8 zones are assigned to 1 GPU, 8 zones are assigned to 1 CPU-based CU composed of 16, 32, 48 or 64 CPUs depending on the configuration. For 2 GPUs, the initial speedup is 83.79x. But the STATIC scheduling is not able to adapt the work distribution now for 3 CUs. Dividing the 16 zones among 3 CUs defines the largest piece of work to be composed of 6 zones that are assigned to the CPU-based CU.

The DYNAMIC scheduling improves the performance significantly. For this scheduling the maximum speedup corresponds to 55.89x for 1-GPU and 90.47x for 2-GPU configurations. In this case, faster CUs (the GPUs) take more work to execute. 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. As a result, we do not observe a significant increment in the *Communication Period*, as happened with the SP-MZ benchmark. Despite that, we have observed that the memorizing ability of the DYNAMIC scheduling still is essential in this case as we have checked that if the same scheduling is applied but removing the memorizing ability (e.g.: each instance of the *Computation Period* remaps the task-CU assignment), then both the *Computation Period* and the *Communication Period* suffers from slowdown factors between 15x-20x.

The STATIC-hybrid achieves the maximum performance in all hybrid configurations. For 1-GPU, speedups range from 48x to 59x, with an increment of performance as the number of CUs is increased. For 2-GPU observed speedups range from 82x to 93x. This scheduling succeeds on identifying the differences in compute power among the available CUs, and distribute tasks (e.g.: zones) accordingly to balance the work distribution. As has been described previously for the SP-MZ application, for the LU-MZ application Tables 7 and 8 expose the PCF values used for the STATIC-hybrid scheduling.

In conclusion, 1-GPU hybrid configurations speed up from 1.13x to 1.30x with respect to the non-hybrid 1-GPU configuration. For 2-GPU hybrid configurations speedups are more modest (up to 1.10x). For LU-MZ, the reduced number of zones (only 16) and their huge size limit the chances

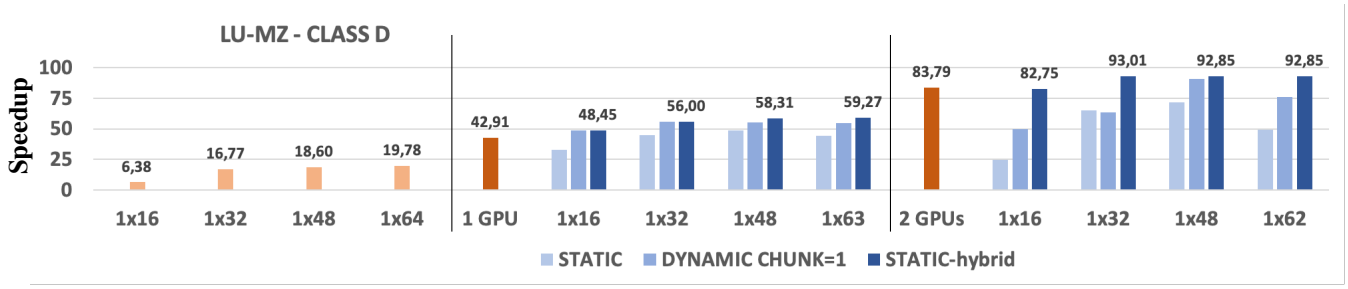

Figure 14. Benchmark LU-MZ with input CLASS D. Overall speedup for different configurations: from left to right, CPU only configurations, 1 GPU configurations and 2 GPU configurations. CPU-based CUs are defined as BxT, where B corresponds to CPU-based CUs executing with T CPUs each. Speedup is measured with respect to single CPU configuration.

for the DYNAMIC scheduler to succeed in producing an appropriate work distribution. The configurations that present a slowdown suffers from a work imbalance, biased to the slowest computing units, the CPUs. Only the STATIC-hybrid scheduling succeeds in maintaining increments of performance as the number of CUs is increased.

BT-MZ benchmark:

Figure 15 shows the speedup achieved by several parallel configurations with respect to the single CPU configuration in benchmark BT-MZ. From left to right, first CPU-based configurations use from 16 to 64 threads exploiting the *inter-zone* parallelism (e.g.: for 64x1 configuration, each thread processes 16 zones). Maximum speedup is observed with 64 threads: 18.48x. The poor scalability is related to two main aspects. On the one hand, 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 (seet Table 7). The pressure over the memory subsystem is different in each configuration, making configurations with a higher count of CUs process one zone with higher execution times. On the other hand, 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, defining a considerable amount of work imbalance. This corresponds to a very different situation compared to the scenarios for the LU-MZ and SP-MZ cases.

For STATIC scheduling, both 1-GPU and 2-GPU cases of hybrid configurations do not present either a good response to the increment of the CU count. In contrast, the DYNAMIC scheduling delivers significant speedups. For 1 GPU, maximum performance is obtained with the 63x1+1 configuration and a 33x speedup factor. For 2 GPUs, the maximum performance is obtained with 62x1+2 configuration with a speedup factor of 36.97x. In both cases, all hybrid configurations outperform their corresponding non-hybrid configurations by factors the a range from 1.5x an 2x. The effect of the DYNAMIC scheduling in the *Communication Period* is similar to that observed in the SP-MZ benchmark. In particular, we have observed that for CPU-only configurations the *Communication Period* takes 88ms, while for hybrid configurations it executes with 205ms. But the work balance benefits of the DYNAMIC scheduling hide the increment of the execution time for the *Communication Period*. As for SP-MZ and LU-MZ, the effectiveness of the Dynamic scheduling is totally conditioned by its memorizing ability for the task-CU mapping. Removing this ability makes the scheduling totally ineffective.

The STATIC-hybrid scheduling balances the work distribution but not up to the level the DYNAMIC scheduling does. Even the improvements in the *Communication Period* related to zone adjacency and communication overheads for the boundary values exchange, these are not sufficient to reach the performance of the DYNAMIC scheduling: speedups range between 19x and 36x for hybrid configurations. The main reason for that is that the STATIC-hybrid divides the set of zones assuming all of them are equally sized. This is not the case of the BT-MZ application. Therefore, the DYNAMIC scheduling deploys a much better work distribution, mainly lead by the different speed of processing that each CU delivers. As in the previous cases, all PCFs values have been obtained from Table 7 and are exposed in Table 8.

Related works

NPB-MZ studies: Dümmler and Rünger (2013) evaluated NPB-MZ benchmarks on hybrid CPU+GPU architectures. They decompose the workloads and, using a static scheduling, distribute them among the CPU's or the GPU. Their evaluations show a significant performance improvement with respect to both pure GPU and pure CPU implementations. Pennycook et al. (2011) detail their implementation of the LU-NPB application on CUDA. Moreover, they developed an analytical model to estimate the execution time of the benchmark on a range of architectures. They validate the model using evaluation environments that range from a single GPU to a cluster of GPU's. Xu et al. (2014) focused on directive-based parallelization of NPB benchmarks. After analyzing and profiling the OpenMP version of NPB, they annotate the source code with OpenACC directives to automatically generate GPU versions of the benchmarks.

Loop and Task Scheduling: In Bull (1998); Polychronopoulos and Kuck (1987); Yong Yan et al. (1997) loop schedulers combine information gathering (e.g.: runtime execution times or actual sizes of data structures) with the ability of memorizing the work assignment produced by the scheduler itself. Specifically for NPB-MZ benchmarks, Duran et al. (2005) describe a feedback scheduler based on execution times to determine thread distribution and work assignment for NUMA shared memory architectures.

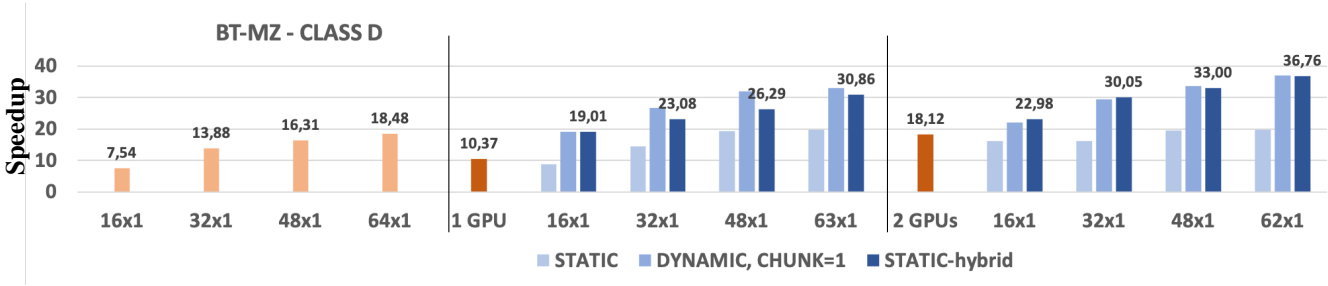

Figure 15. Benchmark BT-MZ with input CLASS D. Overall speedup for different configurations: from left to right, CPU only configurations, 1 GPU configurations and 2 GPU configurations. CPU-based CUs are defined as BxT, where B corresponds to CPU-based CUs executing with T CPUs each. Speedup is measured with respect to single CPU configuration.

In Lucco (1992); Tzen and Ni (1993) loop schedulers address the work imbalance in irregular applications based on runtime information used to deploy dynamic schedulers. Similarly, but targeting thread-data affinity, loop schedulers have been proposed for the preservation of data locality but minimizing the rations of work imbalance (Markatos and LeBlanc 1994; Subramaniam and Eager 1994; Markatos and LeBlanc 1991). In Hamidzadeh and Lilja (1994), an on-line mechanism is presented that dedicates a single processor to execute a branch-and-bound algorithm to search for partial schedules concurrent with the execution of tasks previously assigned to the remaining processors. Task dynamic schedulers have been also studied, specially in the context of OpenMP programming model (Olivier et al. 2012; LaGrone et al. 2011). They focus on the following issues: work balance, locality and cut-off mechanisms for divide-and-conquer algorithms which usually present implementations based on recursive iterative structures.

Heterogeneous Work Scheduling: Heterogeneous computing have generated contributions related to the porting of 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 take the role of orchestrate the parallelism execution, and GPUs act as accelerators (Hermann et al. 2010; Nere et al. 2013; Toharia et al. 2012; Chen et al. 2012; Yang et al. 2013). Other works describe a cooperative heterogeneous computing frameworks which enables the efficient utilization of available computing resources of host CPU cores for CUDA kernels. In Yang et al. the Linpack benchmark is deployed on hybrid petascale architecture introducing adaptive optimizations to balance the work distribution between CPU/GPU cores and with much emphasis in software pipeling techniques to hide communications between the architecture nodes.

The *CoreTSAR* framework (Scogland et al. 2012, 2014) addresses the same challenges as the runtime system described in this paper. On one side, these works take advantage of two existing programming models OpenMP and OpenACC to mix them and activate simultaneously compute units of different nature. If CUDA or HIP were to be also included, the system described in Scogland et al. (2014) should include runtime primitives similar to those described in this paper, specially for solving the entanglement between the data placement and the task scheduling. In terms of the proposed schedulers, the CoreTSAR framework includes dynamic support to solve the mapping between compute units and tasks at runtime, not needing offline profiler-based data samples to perform the actual scheduling. Nevertheless, the proposed static scheduler, makes the work distribution in a similar fashion as CoreTSAR, given that both schedule the tasks according to estimates for the task execution time. The differences arise in how the estimates are generated. For CoreTSAR, regression techniques are used. For our proposal, direct measurements are made prior the actual execution. Therefore, CoreTSAR is more versatile and can cover a broader spectrum of applications, while the proposal in this paper is limited to applications that expose sources of parallelism that are executed many times and always deploying the same structure. But for scientific computing, it is common that applications fall in to this category.

OmpSS Elangovan et al. (2013); Duran et al. (2011); Bueno et al. (2012) is a task-based programming model targeting heterogeneous architectures. It includes a full-stack software support including a directive-based language and compiler and run-time system support for its own parallel task execution model. In general, the parallel strategy described in this paper is supported in OmpSS. For instance, the inter-zone loops in the NPB-MZ benchmarks can be mapped to task-definition directives in OmpSS. Communications for the border computations can be mapped to *in,out* constructs to specify producer-consumer relations.

StarPU Augonnet et al. (2011, 2010) is a framework based on a runtime library that enable the execution of linear algebra kernels on heterogeneous architectures. StarPU is based on *codelets* to specify the computations. StarPU then implements codelets drivers using vendor libraries such as CUDA to automate the execution of StarPU kernels on hybrid platforms. StarPU includes several scheduling strategies to offload the computation on CPUs or GPUs, or co-processors like the IBM Cell BE (Pham et al. 2005). One main difference with StarPU is the utilization of OpenMP as main programming framework to deploy a hybrid parallelization. StarPU does not consider OpenMP nor its inter-operability with CUDA/HIP frameworks. Besides, the concept of compute unit including the aggregation of CPU cores is not considered. Another significant difference corresponds to the architectures StarPU targeted and those explored in this paper. For StarPU, the core count is very small compared to current heterogeneous architectures with more than 64 CPU cores and 2-4 GPU devices.

Performance models have been proposed to implement work distribution schemes (Choi et al. 2013; Zhong et al. 2012). In Ogata et al. (2008), the authors present a library

for 2D Fast Fourier Transform (FFT) that automatically uses both CPUs and GPUs to achieve optimal performance. Using a performance model, it evaluates the respective contributions of each computing unit and then makes an estimation of total execution times. Other recent works introduce application specific work distributions between CPUs and GPUs. In Zhang et al. (2021), a cooperative framework to access a data base is described showing the benefits of the hybrid execution in terms of total throughput. Similarly, in Gowanlock (2021) a hybrid knn-joins algorithm is implemented with an CPU/GPU approach for low-dimensional KNN-joins, where the GPU is not yielding substantial performance gains over parallel CPU algorithms. The paper introduces priority work queues that enable the computation over data points in high density regions on the GPU, and low density regions on the CPU.

Conclusions

In this paper we have described and evaluated the design and implementation of the runtime support for hybrid applications that mix both OpenMP and CUDA/HIP programming models. The runtime bridges both programming frameworks and allows the simultaneous activation of both host and device computing units. For that purpose, we have introduced the abstractions of *computing unit* and *data placement* in order to bridge the two programming models. These abstractions are exposed to the programmer in the form of additional runtime primitives build on top of the native OpenMP runtime system. Besides, we have introduced variants of the static and dynamic task-based schedulings to cope with the different computing power of the CPU and GPU compute units.

The methodology and runtime support have been evaluated within the NPB-MZ benchmark suite. We have shown how the suite has been effectively ported to a hybrid system, being able to simultaneously offload computation over both host and device computing units. On a computing node composed of one AMD EPYC 7742 @ 2.250GHz (64 cores and 2 threads/core, total 128 threads per node) and 2 x GPU AMD Radeon Instinct MI50 with 32GB each we have observed speedup factors that range from 1.08x up to 3.18x over a non-hybrid implementation, depending on the number of activated computing units.

Acknowledgments

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

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado G, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y and Zheng X (2015) TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. URL `http://download.tensorflow.org/paper/whitepaper2015.pdf`.

Augonnet C, Thibault S and Namyst R (2010) StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Research Report RR-7240, INRIA. URL `https://hal.inria.fr/inria-00467677`.

Augonnet C, Thibault S, Namyst R and Wacrenier PA (2011) Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.* 23(2): 187–198. DOI:10.1002/cpe.1631. URL `https://doi.org/10.1002/cpe.1631`.

Awan AA, Manian KV, Chu CH, Subramoni H and Panda DK (2019) Optimized large-message broadcast for deep learning workloads: MPI, MPI+NCCL, or NCCL2? *Parallel Computing* 85: 141–152.

Bailey D, Barszcz E, Barton J, Browning D, Carter R, Dagum L, Fatoohi R, Frederickson P, Lasinski T, Schreiber R, Simon H, Venkatakrishnan V and Weeratunga S (1991) The NAS Parallel Benchmarks. *Int. J. High Perform. Comput. Appl., 5(3)* : 63–73DOI:10.1177/109434209100500306. URL `http://dx.doi.org/10.1177/109434209100500306`.

Belviranli ME, Bhuyan LN and Gupta R (2013) A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.* 9(4). DOI:10.1145/2400682.2400716. URL `https://doi.org/10.1145/2400682.2400716`.

Bueno J, Planas J, Duran A, Badia RM, Martorell X, Ayguadé E and Labarta J (2012) Productive programming of gpu clusters with ompss. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. pp. 557–568. DOI:10.1109/IPDPS.2012.58.

Bull JM (1998) Feedback guided dynamic loop scheduling: Algorithms and experiments. In: Pritchard D and Reeve J (eds.) *Euro-Par'98 Parallel Processing*.

Chen L, Huo X and Agrawal G (2012) Accelerating MapReduce on a Coupled CPU-GPU Architecture. ISBN 9781467308045.

Choi HJ, Son DO, Kang SG, Kim JM, Lee HH and Kim CH (2013) An efficient scheduling scheme using estimated execution time for heterogeneous computing systems 65(2).

Corni E, Morganti L, Morigi MP, Brancaccio R, Bettuzzi M, Levi G, Peccenini E, Cesini D and Ferraro A (2016) X-ray computed tomography applied to objects of cultural heritage: Porting and testing the filtered back-projection reconstruction algorithm on low power systems-on-chip. In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. pp. 369–372. DOI:10.1109/PDP.2016.60.

der Wijngaart RFV and Jin H (2003) NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Ames Research Center.

Dümmler J and Rünger G (2013) Execution Schemes for the NPB-MZ Benchmarks on Hybrid Architectures: A Comparative Study. In: *Procs. of the Intl. Conf. on Parallel Computing, ParCo 2013*, *Advances in Parallel Computing*, volume 25. pp. 733–742.

Duran A, Badia RM, Martinell L, Martorell X and Planas J (2011) Ompss: A proposal for programming heterogeneous multicore architectures. In: *Parallel Processing Letters*. ISBN https://doi.org/10.1142/S0129626411000151, pp. 173–193.

Duran A, Gonzàlez M and Corbalán J (2005) Automatic Thread Distribution for Nested Parallelism in OpenMP. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery. ISBN 1595931678, p. 121–130. DOI: 10.1145/1088149.1088166.

Elangovan VK, Badia RM and Parra EA (2013) Ompss-opencl programming model for heterogeneous systems. In: Kasahara H and Kimura K (eds.) *Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-37658-0, pp. 96–111.

Giuntoli G, Grasset J, Figueroa A, Moulinec C, Vázquez M, Houzeaux G, Longshaw S and Oller S (2019) Hybrid CPU/GPU FE2 multi-scale implementation coupling Alya and micropp.

Gonzalez M and Morancho E (2020) Multi-GPU Parallelization of the NAS Multi-Zone Parallel Benchmarks. In: *IEEE Transactions on Parallel and Distributed Systems*, volume 32. IEEE, pp. 229–241.

González M and Morancho E (2021) Multi-gpu systems and unified virtual memory for scientific applications: The case of the nas multi-zone parallel benchmarks. *Journal of Parallel and Distributed Computing* 158: 138–150. DOI:https://doi.org/10.1016/j.jpdc.2021.08.001. URL https://www.sciencedirect.com/science/article/pii/S0743731521001672.

Gowanlock M (2021) Hybrid knn-join: Parallel nearest neighbor searches exploiting cpu and gpu architectural features. *Journal of Parallel and Distributed Computing* 149: 119–137. DOI:https://doi.org/10.1016/j.jpdc.2020.11.004. URL https://www.sciencedirect.com/science/article/pii/S0743731520304056.

Guan J, Yan S and Jin JM (2013) An openmp-cuda implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-gpu computing systems. *IEEE Transactions on Antennas and Propagation* 61(7): 3607–3616. DOI:10.1109/TAP.2013.2258882.

Hamidzadeh B and Lilja DJ (1994) Self-adjusting scheduling: An on-line optimization technique for locality management and load balancing. In: *Intl. Conf. on Parallel Processing*. USA: IEEE Computer Society. ISBN 0849324939, p. 39–46. DOI: 10.1109/ICPP.1994.179.

Hermann E, Raffin B, Faure F, Gautier T and Allard J (2010) Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In: *Euro-Par 2010 - Parallel Processing*.

Jacobsen D, Thibault J and Senocak I (????) *An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters*.

Jacobsen DA and Senocak I (2013) Multi-level parallelism for incompressible flow computations on gpu clusters. *Parallel Computing* 39(1): 1–20. DOI:https://doi.org/10.1016/j.parco.2012.10.002. URL https://www.sciencedirect.com/science/article/pii/S0167819112000804.

Karunadasa NP and Ranasinghe DN (2009) Accelerating high performance applications with cuda and mpi. In: *2009 International Conference on Industrial and Information Systems (ICIIS)*. pp. 331–336. DOI:10.1109/ICIINFS.2009.5429842.

Kraus J (2013) "an introduction to cuda-aware mpi" URL https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.71.

LaGrone J, Aribuki A, Addison C and Chapman B (2011) A runtime implementation of openmp tasks. In: *OpenMP in the Petascale Era*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-21487-5, pp. 165–178.

Lucco S (1992) A dynamic scheduling method for irregular parallel programs. In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery. ISBN 0897914759, p. 200–211. DOI: 10.1145/143095.143134.

Manavski S and Valle G (2008) CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman string alignment. *BMC bioinformatics* 9 Suppl 2: S10. DOI:10.1186/1471-2105-9-S2-S10.

Markatos EP and LeBlanc TJ (1991) Load balancing vs. locality management in shared-memory multiprocessors. Technical report, University of Rochester, USA.

Markatos EP and LeBlanc TJ (1994) Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 5(4): 379–400.

Message Passing Interface Forum (1994) MPI: A Message-Passing Interface Standard. Standard, USA.

Mittal S and Vetter JS (2015) A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.* 47(4). DOI: 10.1145/2788396. URL https://doi.org/10.1145/2788396.

Nere A, Franey S, Hashmi A and Lipasti M (2013) Simulating cortical networks on heterogeneous multi-GPU systems. *Journal of Parallel and Distributed Computing* 73(7): 953–971.

NVIDIA (2020) GPU-accelerated Caffe. URL https://www.nvidia.com/en-gb/data-center/gpu-accelerated-applications/caffe/.

NVIDIA (2023) CUDA Toolkit Documentation 12.1. URL https://docs.nvidia.com/cuda/.

Ogata Y, Endo T, Maruyama N and Matsuoka S (2008) An efficient, model-based CPU-GPU heterogeneous FFT library. In: *2008 International Symposium on Parallel and Distributed Processing*. IEEE, pp. 1–10.

Olivier SL, Porterfield AK, Wheeler KB, Spiegel M and Prins JF (2012) Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl. 26(2)* 26(2): 110–124. DOI:10.1177/1094342011434065.

Peña AJ, Lai J, Yu H, Tian Z and Li H (2020) Hybrid mpi and cuda parallelization for cfd applications on multi-gpu hpc clusters. *Scientific Programming* 2020: 8862123.

Pennycook SJ, Hammond SD, Jarvis SA and Mudalige GR (2011) Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark. *SIGMETRICS Performance Evaluation Review* 38(4): 23–29. DOI:10.1145/1964218.1964223.

Pham D, Asano S, Bolliger M, Day M, Hofstee H, Johns C, Kahle J, Kameyama A, Keaty J, Masubuchi Y, Riley M, Shippy D, Stasiak D, Suzuoki M, Wang M, Warnock J, Weitzel S, Wendel D, Yamazaki T and Yazawa K (2005) The design and implementation of a first-generation cell processor. In: *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.* pp. 184–592 Vol. 1. DOI: 10.1109/ISSCC.2005.1493930.

Polychronopoulos CD and Kuck DJ (1987) Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers* C-36(12): 1425–1439.

Scogland TR, Rountree B, Feng Wc and de Supinski BR (2012) Heterogeneous Task Scheduling for Accelerated OpenMP. In: *International Parallel and Distributed Processing Symposium*. pp. 144–155. DOI:10.1109/IPDPS.2012.23.

Scogland TRW, Feng Wc, Rountree B and de Supinski BR (2014) CoreTSAR: Adaptive Worksharing for Heterogeneous Systems. In: *Supercomputing*.

Subramaniam S and Eager DL (1994) Affinity scheduling of unbalanced workloads. In: Johnson GM (ed.) *Proceedings Supercomputing '94, Washington, DC, USA, November 14-18, 1994*. IEEE Computer Society, pp. 214–226. DOI:10.1109/SUPERC.1994.344281. URL https://doi.org/10.1109/SUPERC.1994.344281.

Toharia P, Robles OD, Suárez R, Bosque JL and Pastor L (2012) Shot Boundary Detection Using Zernike Moments in Multi-GPU Multi-CPU Architectures 72(9).

Tzen TH and Ni LM (1993) Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems* 4(1): 87–98.

Xu R, Tian X, Chandrasekaran S, Yan Y and Chapman BM (2014) NAS parallel benchmarks for gpgpus using a directive-based programming model. In: Brodman JC and Tu P (eds.) *Languages and Compilers for Parallel Computing*. ISBN 978-3-319-17472-3, pp. 67–81. DOI:10.1007/978-3-319-3-0_5.

Yang C, Wang F, Du Y, Chen J, Liu J, Yi H and Lu K (????) Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. In: *2010 IEEE International Conference on Cluster Computing*.

Yang C, Xue W, Fu H, Gan L, Li L, Xu Y, Lu Y, Sun J, Yang G and Zheng W (2013) A Peta-Scalable CPU-GPU Algorithm for Global Atmospheric Simulations 48(8).

Yang C, Zhang YK, Liang X, Olschanowsky C, Yang X and Maxwell R (2021) Accelerating the lagrangian particle tracking of residence time distributions and source water mixing towards large scales. *Computers & Geosciences* 151: 104760. DOI:https://doi.org/10.1016/j.cageo.2021.104760. URL https://www.sciencedirect.com/science/article/pii/S0098300421000674.

Yang CT, Huang CL and Lin CF (2011) Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. *Computer Physics Communications* 182(1): 266–269. DOI:https://doi.org/10.1016/j.cpc.2010.06.035. URL https://www.sciencedirect.com/science/article/pii/S0010465510002262. Computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15-19, 2009.

Yong Yan, Canming Jin and Xiaodong Zhang (1997) Adaptively scheduling parallel loops in distributed shared-memory systems, 8 (1). *IEEE Trans. on Parallel and Distributed Systems* 8(1): 70–81.

Zhang F, Zhang C, Yang L, Zhang S, He B, Lu W and Du X (2021) Fine-grained multi-query stream processing on integrated architectures. *IEEE Transactions on Parallel and Distributed Systems* 32(9): 2303–2320. DOI:10.1109/TPDS.2021.3066407.

Zhong Z, Rychkov V and Lastovetsky A (2012) Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications. In: *IEEE International Conference on Cluster Computing*. pp. 191–199. DOI:10.1109/CLUSTER.2012.34.