The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-05040-y

EPSILOD: efficient parallel skeleton for generic iterative stencil computations in distributed GPUs

Manuel de Castro[1] · Inmaculada Santamaria-Valenzuela[1] · Yuri Torres[1] · Arturo Gonzalez-Escribano[1] · Diego R. Llanos[1]

Accepted: 29 December 2022

Abstract

Iterative stencil computations are widely used in numerical simulations. They present a high degree of parallelism, high locality and mostly-coalesced memory access patterns. Therefore, GPUs are good candidates to speed up their computation. However, the development of stencil programs that can work with huge grids in distributed systems with multiple GPUs is not straightforward, since it requires solving problems related to the partition of the grid across nodes and devices, and the synchronization and data movement across remote GPUs. In this work, we present EPSILOD, a high-productivity parallel programming skeleton for iterative stencil computations on distributed multi-GPUs, of the same or different vendors that supports any type of n-dimensional geometric stencils of any order. It uses an abstract specification of the stencil pattern (neighbors and weights) to internally derive the data partition, synchronizations and communications. Computation is split to better overlap with communications. This paper describes the underlying architecture of EPSILOD, its main components, and presents an experimental evaluation to show the benefits of our approach, including a comparison with another state-of-the-art solution. The experimental results show that EPSILOD is faster and shows good strong and weak scalability for platforms with both homogeneous and heterogeneous types of GPU.

✉ Diego R. Llanos
diego@infor.uva.es

Manuel de Castro
manuel@infor.uva.es

Inmaculada Santamaria-Valenzuela
msantamaria@infor.uva.es

Yuri Torres
yuri.torres@infor.uva.es

Arturo Gonzalez-Escribano
arturo@infor.uva.es

[1] Departamento de Informática, Escuela de Ingeniería Informática, Universidad de Valladolid, Campus Miguel Delibes s/n, 47011 Valladolid, Spain

Published online: 14 January 2023

Keywords Distributed memory · GPU · Heterogeneous · Stencil · Parallel skeletons

1 Introduction

Iterative stencil computations, or iterative stencil loops (ISLs), are widely used in high-performance computing (HPC). They are used, for example, to compute the numerical solutions of partial differential equations in discretized spaces on a large-scale, in order to characterize and predict physical quantities such as heat, sound, velocity, pressure, density, elasticity, electromagnetism, and electrodynamics [1–4]. In stencil computations, grid cells from a multi-dimensional input are iteratively updated based on the previous values of neighbor cells. The neighborhood is determined using a specific pattern or stencil shape. The highest distance to a neighbor is called the *stencil radius*. The calculation involves accessing neighbor cells according to the stencil shape, multiplying each of them by a specific coefficient, adding all these results, and optionally dividing the result by a given factor. This computation is repeated independently for each cell.

Iterative stencil computations present a high degree of parallelism, high locality and mostly coalesced memory-access patterns. Therefore, many-core computational models and platforms, such as modern graphics processing units (GPUs), are good candidates to speed up this type of computations. GPUs are currently the most popular accelerator in supercomputers and are employed in most of the top 10 machines in the TOP500 list [5].

Programming stencil computations in GPUs for large grids is not straightforward. The big memory footprint required in many real case situations can easily become too large to be allocated on the memory space of a single GPU. In these cases, the use of multi-GPUs in one or more distributed nodes is a useful alternative. However, this requires mixing programming models for both GPUs and distributed memory across different nodes. For example, a programmer may combine CUDA [6] or OpenCL [7] to manage the devices, with a distributed memory tool, such as MPI [8], to manage inter-node communications. This implies a complex and error-prone programming, manually dealing with accelerator optimizations, data partitions, different memory hierarchies, and manually specifying sequences for data communication and synchronization, both across accelerators and host, and across distributed nodes. Nevertheless, the operations of a stencil computation across iterations are mainly repetitive, and it is possible to build higher-level abstractions to program them.

In this work, we present EPSILOD, a high-productivity parallel programming skeleton for iterative stencil computations on distributed multi-GPUs. The underlying idea was presented as a poster in SC'21 [9]. EPSILOD supports any type of n-dimensional geometric stencils (compact, non-compact, symmetric, or asymmetric) of any order. It is implemented as a C99 library function to maximize compatibility with any C/C++ environment.

It uses an abstract specification of the stencil pattern (neighbors and weights) to internally derive the data partition, synchronizations and communications, splitting the computation to better overlap with communications. To enable portability and fast

synchronization and device management, it is programmed with the Controllers model [10] and the Hitmap library [11]. These tools provide a heterogeneous portability layer that allows the transparent use of different types of devices.

The contributions of this work can be summarized as follows:

- We introduce EPSILOD's architecture and describe the programming tools used to implement it. We also show how to use EPSILOD to characterize stencils and automatically generate efficient kernels.
- We present a new skeleton version that supports clusters with devices of mixed types and vendors, such as NVIDIA and AMD GPUs. It includes the possibility of using load balancing techniques to transparently distribute the computation across devices with different computation power. The skeleton includes a generic GPU kernel that can be used for any stencil. EPSILOD also includes a tool to generate more optimized kernels for each stencil pattern, or even the possibility of providing a user-optimized kernel.
- We include extensive experimentation to show the benefits of our approach, comparing it with an state-of-the-art solution, and taking into account different architectures. This evaluation includes:
  - An experimental study in a BSC (Barcelona Supercomputing Center) cluster of up to 48 NVIDIA V100 GPUs, distributed among 12 nodes, which shows that our proposal can achieve both very good strong and weak scalability for several types of 2D stencils.
  - An experimental study in a cluster of up to 32 NVIDIA V100 GPUs, distributed among eight nodes, which shows that our proposal outperforms implementations of the same stencils using Celerity [12], a state-of-the-art programming tool for distributed GPUs built on top of MPI and SYCL.
  - An experimental study in a heterogeneous node comprising different kinds of GPUs (NVIDIA and AMD) that shows the impact of using the load-balancing technique integrated in the skeleton to achieve good strong and weak scalability in this kind of heterogeneous platforms.

The experimental results show that EPSILOD is faster than Celerity and shows good strong and weak scalability for both homogeneous and heterogeneous platforms. The EPSILOD source code and examples can be freely downloaded from https://gitlab.com/trasgo-group-valladolid/controllers/-/tree/epsilod_JoS22.

The rest of the work is structured as follows. Section 2 discusses some related work. In Sect. 3, we describe the main features of the Hitmap and Controllers libraries used to build our solution. Section 4 describes how EPSILOD is built on top of these libraries. Section 5 presents an extensive experimental evaluation of the capabilities of EPSILOD. Finally, in Sect. 6, we present our conclusions and discuss future work.

2 Related work

Exploiting GPUs for general-purpose computing has emerged as a high-performance computing technique to accelerate stencil-based applications in different domains. These works [13–17] are focused on specific physical problems, and they are manually developed using the native parallel programming model provided by the vendor, such as CUDA or OpenCL. When using these techniques in distributed systems, the programmer is responsible for manually developing the code for load partitioning, mapping, synchronizations and data communications.

In addition, there exist several CPU-based frameworks and libraries, such as [18–24], that are focused on a wide range of stencil computations for distributed systems, some of them even at an exascale level. The use of these frameworks is comparatively simple, due to the abstraction levels introduced. Nevertheless, their performance is significantly penalized in heterogeneous clusters, since they only use CPUs as computing units, without exploiting any kind of coprocessor devices, such as GPUs.

There exist some frameworks that aim to generate stencil code not only for CPUs, but also for hardware accelerators. DeVito [25] is a framework capable of generating highly optimized code given symbolic equations expressed in Python, specialized in, but not limited to, stencil codes. It internally uses OpenMP/OpenACC to target GPUs and MPI for distributed platforms. YASK [26] is another framework that allows the creation of high-performance stencil codes for distributed CPUs and GPUs, although it is focused on optimizations for Intel devices and it does not support other platforms. None of these proposal exploits several vendor-specific programming models and tools to target simultaneously different devices in heterogeneous environments.

There are works that propose an efficient GPU algorithm for computing stencils with arbitrary patterns, such as [26–32]. These works propose the use of stencil frameworks capable of automatically transforming stencil patterns that appear in a given C source code into optimized CUDA code. However, they do not support distributed environments. The advanced optimization techniques considered in some of these works include exploiting task granularity adjustment at runtime, depending on device performance [31], or the use of concurrent kernels, taking into account data dependencies in order to better exploit underlying GPU hardware resources [32].

There are other works oriented toward executing large stencil computations for distributed systems using GPU coprocessors, some of them based on parallel skeletons, such as [33–42]. These works achieve a better performance due to the use of this kind of hardware accelerators. In particular, Muesli [42] and SkelCL [39] are good examples of skeleton-based solutions that fall in this category. However, they do not implement a full overlapping of data communication and computation, combining the overlapping opportunities in transfers between host and devices with communication and synchronization across nodes. Thus, GPU coprocessors can sometimes be idle while waiting to be fed by data. Most of these works require the programmer to indicate operations in the correct order. Only [34] includes a technique to automatically deduce dependences and determine the

execution order of the different tasks, but it still needs the inclusion of a synchronization primitive to properly synchronize the operations related to the exchanges of *ghost-zones* (parts of the grid that are shared across devices).

Finally, we should mention other popular, general-purpose programming frameworks for heterogeneous systems that are not specific for stencils, but that can be used to develop a distributed application to compute a stencil on a parallel, heterogeneous system. They can also be used to build higher abstractions for programming stencil applications on this kind of platforms.

OpenMP [43] is a classical and extensively used programming model for multi-core shared-memory systems. Modern compilers have full support for the latest versions of OpenMP, which include features to offload the computation to devices such as GPUs. Although there are plenty of academic proposals to extend OpenMP to distributed memory systems, the model does not include a native communication system across nodes. Moreover, programming asynchronous operations to manage several devices and overlap computation for GPUs is not trivial [44]. Thus, the programmer should rely on an external communication model, such as MPI, and devise complex structures and synchronization mechanisms to move data across remote GPUs.

RAJA [45] is a portability layer that enables C++ applications to leverage various programming models, and thus architectures, with a single-source codebase. RAJA targets loop kernel parallelism for C++ applications by relying on standard C++11 language features for its external interface, and common programming model extensions such as OpenMP and CUDA for its implementation. RAJA is designed to integrate with legacy codes simply and to provide a model for the development of new codes that are portable from inception. However, Raja does not include mechanisms to do inter-node communication. Thus, as in OpenMP, applications targeting distributed memory systems usually rely on message-passing models, such as MPI, manually dealing with the complexities of the synchronization of the devices and the data movements (see e.g., [46]).

The Kokkos C++ library [47] enables applications and domain libraries to achieve performance portability on diverse manycore architectures by unifying abstractions for both fine-grain data parallelism and memory access patterns. Its most recent extension [48] includes support for hierarchical parallelism, containers, task graphs and arbitrary-sized atomic operations. Kokkos Remote Spaces [49] adds distributed shared memory (DSM) support to Kokkos. It enables a global view of data for a convenient multi-GPU, multi-node and multi-device programming. It is built on top of libraries implementing a PGAS model or even MPI one-sided communications. However, it provides a fine grain communication model that does not scale well even with several GPUs in the same node, as reported by the authors [49]. More complex data descriptions could be used to manually express aggregate communications, but dealing with the same complexity as expressing communications in a message-passing style. Finally, the Kokkos mechanism to perform efficient data transfers across different memory spaces (deep-copy) implies synchronization fences, preventing the programmer from exploiting the full potential of overlapping device computation with communication.

SYCL [50] is a standard model for cross-platform programming with several implementations and compilers, such as DPC++ in Intel OneAPI (DPC++) or hipSYCL. Kernels and host tasks are organized by a task graph that is implicitly constructed by the SYCL runtime. The control flow and the data transfers needed are derived from programmer declarations of the input/output role of the buffers used by computations. Both computations and device-host communications can be implicit and asynchronous. Celerity [12] extends the ease of use of the SYCL domain-specific embedded language to distributed clusters. While the execution of parallel kernels is still handled by the SYCL runtime on each individual worker node, the Celerity runtime acts as a wrapper around each compute process, handling the data partition and the inter-node communication and scheduling. The programmer declares a *range mapper* that indicates to the runtime system the data elements to be accessed by each range of elements. This allows the inter-process communications for the data partition to be derived. Indeed, Celerity includes a *neighborhood* range mapper that is specifically indicated to express the access pattern of symmetric stencil computations. The communications derived for asymmetric stencils are non-exact, including non-needed communications or elements, but they can still be programmed. Celerity is a good candidate to implement stencil computations for distributed multi-GPUs.

In order to build a higher abstraction for stencil computations on distributed multi-GPU environments, we decided to rely on two parallel programming tools developed by our research group. Controllers [10] is a heterogeneous parallel programming model implemented as a C library. It transparently manages the coordination, communication and kernel launching on different heterogeneous computing devices. It exploits native or vendor-specific programming models and compilers, such as OpenMP, CUDA, or OpenCL, enabling the potential performance obtained by using them. It supports asynchronous operations using both data buffers and selected subparts of them. Hitmap [11] is a distributed programming library, built on top of MPI, that simplifies the partition and communication of data structures in distributed memory environments. It allows the construction of customized and reusable communication patterns in terms of data-access patterns and partition policies chosen at runtime. They can express the exact communications needed for both symmetric and asymmetric stencil computations of any order. Moreover, a precompiled program or library using Hitmap calculates, at runtime, the communication pattern for the given data-access pattern, adapting itself to the chosen stencil. Controllers also uses Hitmap to abstract the management of the data structures across accelerators and host. Hitmap is included in the Controller release. Thus, they are perfectly coupled and seem appropriate to build higher-level abstractions for stencil computations on distributed GPUs. The following section provides additional information on these tools.

To facilitate comparison, Table 1 summarizes the features of some of the stencil-computations frameworks discussed in this section. As can be seen, among the main frameworks analyzed, only Celerity presents part of the heterogeneous portability features of our proposal.

Table 1 Main features of the frameworks described in the related work

Framework	Multi-GPU	Multinode	Heterogeneous portability	Mixed GPUs
SkelCL ∗	X			
Muesli ∗	X	X		
DeVito ‡	X	X		
YASK ‡	X	X		
Celerity	X	X	X	
EPSILOD	X	X	X	X

Multi-GPU Support for several GPU devices in the same node. *Multinode*: Support for distributed computing across GPU devices in several distributed nodes. *Heterogeneous portability*: Built on top of a heterogeneous programming layer that targets different programming models for supporting GPUs of different type/vendor on each compilation or run. *Mixed GPUs*: Support for mixing GPU devices of different types/vendors on the same execution with load balancing

∗ Examples of parallel solutions used to program stencils on multi-GPU systems

‡ Examples of domain-specific frameworks used for stencil applications in multi-GPU systems

3 EPSILOD building blocks: Hitmap and Controllers

Gorlatch and Cole defines a parallel skeleton as "a programming construct (or a function in a library), which abstracts a pattern of parallel computation and interaction. To use a skeleton, the programmer must provide the code and type definitions for various application-specific operations, usually expressed sequentially. The skeleton implementation takes responsibility for composing these operations with control and interaction code in order to effect the specified computation, in parallel, as efficiently as possible" [51]. Our proposal, EPSILOD, is an efficient framework to allow the execution of large stencil computations on distributed, heterogeneous systems, thanks to the use of a skeleton specially designed to carry out stencil computations. EPSILOD relies on an internal mechanism of the Controllers model and the Hitmap library to start asynchronous communications when the runtime detects that a part of a data structure has been fully updated. Afterward, it automatically launches the next tasks and synchronizes execution and communication. Thus, the opportunities for overlapping computation and data movement are maximized.

This section presents an overview of the programming tools used to implement our stencil skeleton. We discuss the core concepts of the Hitmap library and the Controllers programming model, focusing on those exploited by our proposal. They are both implemented as libraries of functions written in C99. They are built with an object-oriented paradigm, comprising multiple structures, each one with its corresponding API of C functions, that work like classes. They are compatible with any C99/C++ compiler, and they are easily interoperable with other libraries and parallel programming models.

The source code of both modules is freely available on the repository of the Trasgo group.[1]

3.1 Hitmap

The Hitmap library [11] provides a portable interface to manage distributed arrays and other data structures. It is built on top of MPI, providing higher-level abstractions. It implements functions to efficiently create, handle, map and communicate hierarchical tile arrays. The library supports three main sets of functionalities:

- *Tiling functions* Hitmap provides the *HitTile* structure, a kind of *fat-pointer*. It stores, alongside the pointer to the memory space, metadata related to the size, dimensions and partitions of an array. The index domains of the tiles are declared and queried using the interface associated to the *HitShape* structure. Accessing these abstract arrays is performed using a generic function *hit*, with a variable number of index parameters.
- *Mapping functions* The *HitTopology* and *HitLayout* structures provide abstractions for modular functionalities for the partition and mapping of data structures across virtual processes. The library includes from common n-dimensional partition policies to heterogeneous partitions based on different weights for each process that can be used for load balancing in heterogeneous environments.
- *Communication functions* They allow the creation of reusable communication patterns, *HitPattern*, based on a given *HitLayout* distribution. They provide an abstraction of a message passing programming model to communicate tiles between multiple virtual processes. Each new pattern is built in terms of the runtime information found in a layout object. Thus, they transparently adapt the communication structure to any change in the distribution policy or number of processes.

More architecture details and use cases of the Hitmap library can be obtained in [11].

3.2 Controllers programming model

Controllers [10] is a heterogeneous programming model implemented as a library that enables performance portability across CPU-cores (using OpenMP), GPUs (using CUDA or OpenCL), Intel FPGAs (using Intel FPGA SDK for OpenCL), or Xilinx FPGAs (using Xilinx Vitis). The Controllers model integrates the use of different vendor specific technologies through different backends coordinated by the same runtime layer, effectively handling the architectural differences between different computing devices [52]. The programmer can write a single generic portable kernel code that can target different kinds of devices. The programmer can also provide different versions of the same kernel, optimized for different devices, including

[1] https://gitlab.com/trasgo-group-valladolid/controllers.

code and features of the vendor or low-level programming model if needed. These kernels are then compiled with the desired vendor or native compiler. All object files generated by these compilers are then linked together. In this way, programs developed using the Controllers model include executable codes for all the backends and their runtime support, allowing the program to select at runtime the most efficient kernel for the type of device chosen, either from different vendors or from different families of the same vendor. Controllers also allows the declaration of host-tasks, a special type of kernel that are executed within a host-space backend. The host-tasks allow generic host operations, such as calls to I/O operations.

For programming the host coordination code, the Controllers framework proposes an abstract class, called Controller. A Controller object is associated with a particular instance of a device during its construction. Each object transparently manages the coordination and communication of the host code with that device. For the transparent and portable management of data structures, Controllers uses the *HitTile* abstraction of the Hitmap library (described in Sect. 3.1) to encapsulate variables and data structures, such as multi-dimensional arrays. The HitTiles can be allocated only in the host, only in the device (internal variables), or in both host and device sides (tied variables). The *hit* function is used in host and kernels as an abstraction to transparently access data with both functional and performance portability. It provides a portable view, with consistent row-major order, across any device and the host.

When programming a kernel or host-task, the programmer should also provide information that includes the role of each parameter (input/output). By analyzing the dependencies of launched kernels and host-tasks, the Controllers model automatically handles the execution order, asynchronous data transfers between host and devices, and ensures that computations are performed on up-to-date data, regardless of what entity executes them (host or devices). Controllers extends the *HitTile* structure with more meta-data related to the location and state of the data in the device, that is internally used to automatically keep the memory consistency across the different memory hierarchies.

4 Parallel stencil skeleton

This section discusses the proposed parallel stencil skeleton from the programmer's point of view, through the description of the programming interface. It presents with examples: (a) How to provide a stencil description; (b) a description of a generic kernel provided with the skeleton that can use any stencil description; (c) a kernel generation tool to build more efficient kernels from stencil descriptions; and (d) how to call the parallel stencil skeleton with both the generic kernel and an optimized kernel.

The section then provides a description of the skeleton implementation, including how it distributes the load, coordinates data transfers and inter-node communications and achieves an efficient overlap of computation and communication using device computation partition and asynchronous techniques.

4.1 Interface and usage

The proposed stencil skeleton is presented as a reentrant, higher-order function written in C99, compatible with any modern C/C++ compiler. The function solves a fixed number of iterations of a geometric stencil computation, applying the stencil operator to each element of an n-dimensional array. It is implemented using the Hitmap library and the Controllers programming model. As we will see, from a programmer's perspective, the interface is simple to understand and its use is straightforward.

Stencil Description.

The stencil operator is expressed by a *Stencil Description* that comprises the following data: (a) A *Shape*, expressed as a *HitShape* object, indicating the distance to the farthest neighbor (or radius) on each dimension and direction; (b) a *Pattern*, an array of weights/coefficients for each element in the shape, with zeros in the elements that should be ignored, and positive/negative weights for the actual neighbors; and (c) a *Division Factor* for the weighted sum. Listing 1 shows an example of a stencil description for a classic 2D 4-point star stencil, to solve Poisson's equations of heat transfer using a Jacobi method.

The skeleton provides a generic kernel that works for any Stencil Description. The user can also provide a more optimized kernel code. Moreover, a tool to generate specific kernels from Stencil Descriptions is also provided.

Listing 1 Description of a 2D 4-points star stencil to compute Poisson's equation for heat transfer with a Jacobi method.

```
HitShape shp_jacobi2d4 = hitShape((-1, 1), (-1, 1));

float patt_jacobi2d4[] = {
  0, 1, 0,
  1, 0, 1,
  0, 1, 0};

float factor_jacobi2d4 = 4;
```

The Generic Kernel.

The generic kernel included in the skeleton works for any kind of stencil, once provided its Description (shape, pattern, and division factor). It works by iterating over the stencil pattern elements. For each element, the corresponding neighbor is accessed, multiplied by the weight, and summed into the result. The internal implementation of the generic kernel for 2D stencils is shown in Listing 2. The arguments provided to the kernel, related to the pattern size (*begin_x*, *begin_y*, *end_x*, and *end_y*), are automatically provided by the skeleton from the Stencil Description's shape.

Listing 2 Implementation of the generic kernel for any 2D stencil.

```
CTRL_KERNEL(updateCell_generic_2D, GENERIC, DEFAULT,
    KHitTile_float matrix, KHitTile_float matrixCopy,
    KHitTile_float weight, int begin_x, int begin_y,
    int end_x, int end_y, float factor, {
  int x = thread_id_x;
  int y = thread_id_y;

  float sum = 0;

  int i;
  int j;
  for (i = begin_x; i <= end_x; i++)
    for (j = begin_y; j <= end_y; j++)
      sum += hit(matrixCopy, x + i, y + j)
        * hit(weight, i - begin_x, j - begin_y);

  hit(matrix, x, y) = sum / factor;
});
```

Kernel Generation Tool (KGT).

The use of a single generic kernel for any type of stencil computation is inefficient in terms of performance. Using nested loops and accessing all the elements of the stencil pattern, regardless of their weights, derives in a performance penalty for the generic kernel. This performance penalty is especially noticeable for stencils with patterns where most elements have a weight of zero.

We provide a tool, called Kernel Generation Tool (KGT), that is able to generate a tailored, more efficient kernel from the very same Stencil Description the generic kernel uses. The generated kernel avoids accesses to elements that do not participate in the computation, as well as the usage of loops. We refer to this kind of kernels as *specific* kernels. The KGT is currently implemented as a Bash+Awk script. It receives a C code file as the argument with the Stencil Descriptions in the same format discussed previously (see Listing 1). The script traverses the stencil pattern and generates a code without loops that directly accesses neighbors with positive/negative weight values, avoiding elements with zero weight. Thus, these kernels require less memory bandwidth and achieve a higher throughput than the generic kernel. The kernels are generated with the necessary Controllers wrapper syntax. Listing 3 shows the stencil kernel generated by the tool, from the stencil description provided in Listing 1. The user can try to optimize the generated kernels further to achieve even higher throughput.

In Sect. 5.1, we provide a performance comparison between the generic kernel and a tailored kernel generated by the KGT tool for different test scenarios.

Listing 3 Implementation generated by KGT for the 2D 4-points star stencil for Poisson's equation.

```
CTRL_KERNEL(jacobi2d4, GENERIC, DEFAULT,
    KHitTile_float matrix, KHitTile_float matrixCopy, {
  int x = thread_id_x;
  int y = thread_id_y;

  hit(matrix, x, y) = (hit(matrixCopy, x - 1, y) +
             hit(matrixCopy, x, y - 1) +
             hit(matrixCopy, x, y + 1) +
             hit(matrixCopy, x - 1, y)) / 4;
});
```

Skeleton function prototype.

Listing 4 presents an excerpt of the header file of the skeleton function, showing the function prototype and the type definitions for the initialization and results output functions provided by the user. The skeleton function has the following parameters:

1. The sizes of the space array on each dimension, provided as an array of integers.
2. The Stencil Description:
   (a) A *HitShape* object describing the stencil shape.
   (b) The stencil pattern, an array of weights for each stencil element in the shape.
   (c) The division factor for the weighted sum. This argument is used only with the generic kernel, and it is ignored when using specific or custom kernels.
3. The number of iterations to apply the operator.
4. The name of the stencil kernel to use, or a *NULL* pointer to use the generic kernel.
5. A pointer to a function to initialize the values of the space array. This function will receive a *HitTile* with the part of the array assigned to the local process, the number of dimensions of the array, and information about the boundary conditions. Line 2 of the header file excerpt shows the type definition for this function.
6. A pointer to a function to output the results of the computation. This function has the same parameters as an initialization function. Line 3 of the header file excerpt shows the type definition for this function.

Listing 5 shows an example program calling the stencil skeleton function using the generic kernel provided with the skeleton. The example uses the 2D 4-point star stencil for Poisson's equation described in Listing 1.

Listing 4 Excerpt of the header file of the skeleton function.

```
1  /* Type definitions included in header file */
2  typedef void (*initDataFunction)(HitTile_float, int, int[], int[]);
3  typedef void (*outputDataFunction)(HitTile_float, int, int[], int[]);
4
5  /* Parallel skeleton prototype included in header file */
6  void stencilComputation( int sizes[], HitShape stencilShape,
7      float stencilPattern[], float factor, int numIterations,
8      stencilFunction f_updateCell,
9      initDataFunction f_init, outputDataFunction f_output );
```

Registering new kernels.

Listing 6 shows an example of using the skeleton function with a kernel written manually by the user or generated with the provided tool (KGT). These kernels must be *registered* to be used in the skeleton function. This is done with the macro-function *REGISTER_STENCIL*. This macro-function receives the name of the stencil kernel to be registered as the argument. The macro expands into the special kernel prototype required by the Controllers model, and into a host wrapper function that launches the kernel. The name of the kernel used, passed to the stencil skeleton function as the fourth argument, is transparently translated to a pointer to the wrapper function, which launches the device kernel with the Controllers framework. All the low-level details are hidden from the programmer.

Listing 5 Example program calling the proposed skeleton's function with the generic kernel.

```
1  #include "Ctrl.h"
2
3  /* Main program: stencil execution */
4  int main( int argc, char *argv[] ) {
5    ...
6    HitShape shpStencil = hitShape( (-1, 1), (-1, 1) );
7    float stencilPattern[] = {
8      0, 1, 0,
9      1, 0, 1,
10     0, 1, 0
11   };
12
13   stencilComputation( sizes, shpStencil, stencilPattern, 4,
14     numIter, NULL, initData, outputData );
15   ...
16 }
```

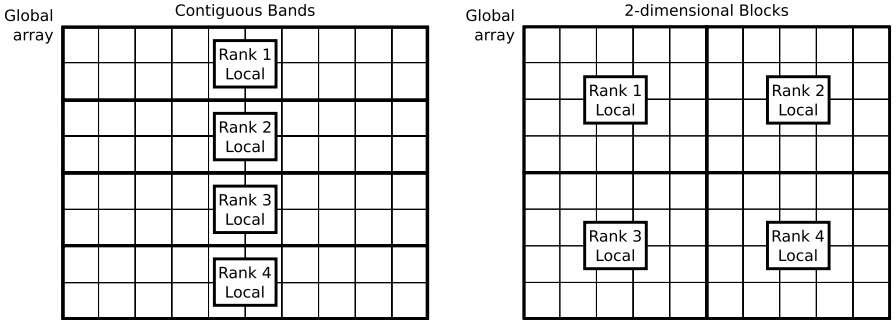

Fig. 1 Partition of the global array to different processes by contiguous bands (left) or two-dimensional blocks (right)

Listing 6 Example program calling the proposed skeleton's function with a pre-generated kernel.

```
#include "Ctrl.h"

/* Declare/Register kernel before using it */
REGISTER_STENCIL( jacobi2d4 );

/* Main program: stencil execution */
int main( int argc, char *argv[] ) {
  ...
  stencilComputation( sizes, shpStencil, stencilPattern, ignored_int,
    numIter, jacobi2d4, initData, outputData );
  ...
}
```

4.2 Implementation details

This section describes the main ideas to implement the skeleton structure with efficient overlapping of computation, data transfers, and inter-node communications.

Distribution of the global array.

The skeleton program is launched as an MPI application. Internally, the ranks of the MPI processes in the same node determine the associations between processes and each of the available devices in the node. A Controller object is automatically created on each process to manage the associated device.

The skeleton receives the sizes of the global array as input. It uses Hitmap's topology and layout functions, to organize the available MPI processes in a grid and divide the index space evenly among them. Partition can be done in contiguous bands by the first dimension of the array or in two or three dimensional blocks. See Fig. 1. The partition method is chosen by configuring the skeleton function. For heterogeneous clusters, we also introduce a second skeleton

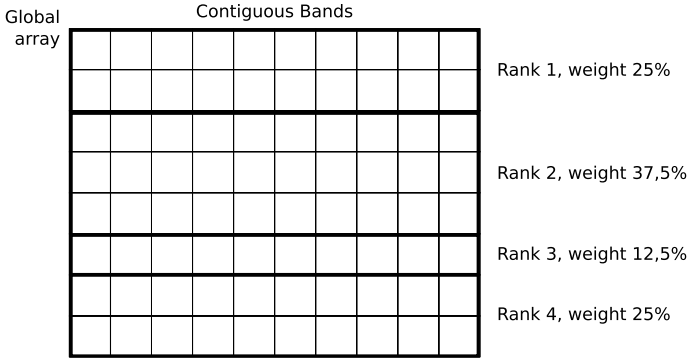

Fig. 2 Partition of the global array to different processes by contiguous bands using a previously determined weighted distribution

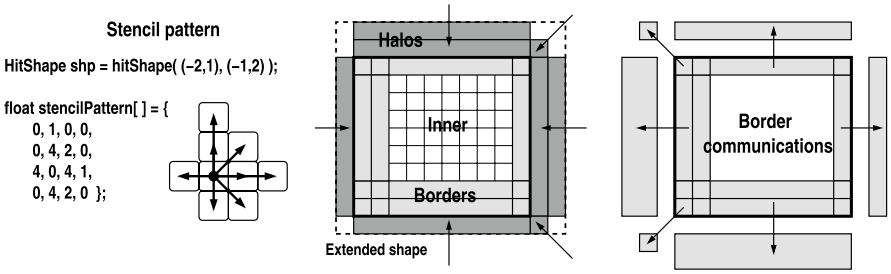

Fig. 3 Left: Asymmetric irregular 2D stencil pattern definition and its graphical representation. Middle: The different parts and extended shape calculated for the local part of an array. Right: Different instances of border sub-parts to be communicated to neighbor processes

version using a specific layout function that makes a weighted division of the index domain in horizontal bands. The amount of rows mapped to each process is proportional to the relative weight of that process. Weights are provided by the programmer, according to the estimated or previously measured computational power of the different devices used by the MPI processes. Thus, the computation load can be balanced across different types of devices. See Fig. 2. The skeleton creates a *HitShape* object to represent the part of the global index domain mapped to the local process.

Subparts of the local arrays.

The stencil shape received as parameter contains the radiuses of the stencil pattern on each dimension and direction. They are used to create space for *halos*, memory zones to have copies of remote data not owned by the process, but needed to compute the local cells.

We distinguish three different non-overlapping parts in each process' local array (see Fig. 3):

- Halos: Elements of the local arrays not owned by the process, but needed to compute the local part. These elements are not assigned to the process by the partition function. They are included in the local part during the expansion of the shape determined by the stencil radiuses and should be received on every iteration from the neighbor processes. They are the outermost elements of the local arrays, and they are not updated on the local process.
- Borders: These are the elements owned by the local process that are duplicated as halos in other processes. They are updated locally and they should be sent to their corresponding neighbors on every iteration.
- Inner: The innermost elements of the local arrays, skipping halos and borders. These elements are updated on every iteration, and they are not communicated to any neighbor.

Figure 3 shows a highly asymmetric stencil pattern (on the left), and the corresponding halos and extended shape generated for a local array (in the middle). Hitmap provides functionalities to extend the shape of the local part of the array according to the information concerning radiuses. The extension automatically takes into account that it should not be expanded beyond boundaries in the first or last processes of each dimension.

The skeleton proceeds to allocate memory for the expanded local part of the array including the halos. Each process allocates memory for two local arrays: One to store the data computed in the previous iteration, and another to store the results of the current iteration. The skeleton uses one HitTile object as a handler to manage each array. A functionality of the Controller object is used to allocate each HitTile. This functionality creates two related memory images for a HitTile, one in the host and one in the device.

Before the first iteration, the user's data-initialization function is called to introduce the initial values in the array. The data are copied into the device memory image. After computing each iteration, the handlers (*HitTiles*) of the two arrays are swapped to exchange their internal meta-data, including the data pointers in host and device. This relocates the new computed data in the array of previous results, avoiding array data copies. After all the iterations have been executed, the results are copied from the device to the host, and the user's output function is called on each process. The user's function can use the functionalities provided by Hitmap to easily write a distributed array in the file system.

Generation of the communication pattern.

At the same time that the halos and extended shape of the local array are determined, the same analysis of the stencil pattern is used to automatically calculate the inter-node communications needed across iterations. The elements with nonzero weights in the pattern indicate the borders which should be effectively sent to neighbor processes and received in the halos. For each dimension of the processes grid, it is checked whether communication is needed from/to the previous and next processes, also computing the width of the border or halo to communicate from the shape of the stencil. For asymmetric stencils, the widths could be different for each direction, and communications in just one direction may be needed. In partitions of

two or three dimensions, if the pattern has nonzero weights in diagonal positions with respect to the cell to compute, communications are also needed for the corners of the borders or halos, with processes which are also in diagonal positions in the processes grid. In these cases, some data in the corners of the borders should be communicated to more than one process. See an example in the leftmost corner of Fig. 3 (right).

Let d be the number of dimensions of the partition, or processes grid. The maximum number of active border send operations, or active halo receive operations, is $3^d - 1$. The processes at the limits of the grid deactivate the send/receive operations in directions where there is no neighbor process.

The communication pattern to be repeated on each iteration for the local process is defined by the active borders/halos and the shapes of the corresponding pieces. This pattern is used in two ways. First, to build an inter-process communication pattern. Hitmap provides a functionality to build a *HitPattern* object in terms of borders and halo shapes. The shapes' intersections lead to exact communications. No data is communicated if it is not needed, and no data is communicated more times, or to more processes, than needed. The *HitPattern* contains the information to perform the MPI communication operations for all the data movements indicated during its construction. Hitmap provides functionalities to perform the communications of a *HitPattern* at any moment.

Second, the pattern is used to coordinate data transfers from and to the device. Hitmap allows the declaration of hierarchical tile structures: A *HitTile* can be a sub-selection of a given part of another *HitTile*. Thus, it is possible to declare HitTiles for each part of a border, for a halo, or for the inner part of the local array. Their internal data pointers indicate the first element of the sub-selection, and the metadata indicates the sizes and strides needed to access the data. Controllers provides functionalities to move data from/to the device for a *HitTile*. The Controllers model abstracts the underlying hardware details of the different devices and the low-level communication mechanisms from the skeleton. Each Controller object used in the program transparently leverages the native or vendor-specific programming model for the chosen device, such as CUDA or OpenCL, to perform these data transfers. This allows the skeleton to achieve data-transfer throughputs similar to those of the native or vendor-specific models [52]. If the tile to move is a sub-selection, appropriate data movements for only that specific part of the array are executed. The Controllers backends transparently use the proper CUDA or OpenCL functions to efficiently transfer sub-selections of an array, which can include non-contiguous memory pieces. These data transfers are done asynchronously by the Controllers runtime.

Algorithm 1 Operations performed on each iteration to maximize the overlapping of computation and communication.

```
1: for i ← 0 to numBorders − 1 do                    ▷ Launch split computations for borders
2:     launch kernel(newBorderTiles[i], oldTile)
3: end for
4: launch kernel(newInnerTile, oldTile)              ▷ Launch inner-part computation
5: for i ← 0 to numBorders − 1 do                    ▷ Start data transfers for borders
6:     if activeBorders[i] then
7:         copy device-to-host newBorderTiles[i]
8:     end if
9: end for
10: for i ← 0 to numBorders − 1 do                   ▷ Wait end of data transfers for borders
11:     if activeBorders[i] then
12:         wait newBorderTiles[i]
13:     end if
14: end for
15: doCommunication( hit-pattern )                   ▷ Do inter-process communication
16: for i ← 0 to numBorders − 1 do                   ▷ Start data transfers for received halos
17:     if activeHalos[i] then
18:         copy host-to-device newHaloTiles[i]
19:     end if
20: end for
21: swap( newTile, oldTile )                         ▷ Swap HitTiles handlers for new/old local arrays
22: swap( newBorderTiles, oldBorderTiles )
23: swap( newHaloTiles, oldHaloTiles )
24: swap( newInnerTile, oldInnerTile )
```

Overlapping of computation and communication.

In order to obtain the maximum possible overlapping of computation, data transfers and inter-process communications, we should consider separately the dependencies between the halos, borders and inner part of the local and remote arrays.

Algorithm 1 shows the order of operations performed by the skeleton on each iteration. It uses arrays of HitTiles representing borders and halos, and two extra HitTiles for inner parts. Those that start with the prefix *new* are sub-selections of the local array for computing new values on this iteration (*newTile*). Those that start with the prefix *old* are sub-selections of the local array that holds the data of the previous iteration (*oldTile*). Recall that the number of borders is $3^d - 1$, d being the number of dimensions of the partition, or processes grid. There are two Boolean arrays, *activeBorders* and *activeHalos*. Their elements are set on positions where there is a communication to send out that border, or to receive in that halo, according to the stencil pattern and the position of the local process in the processes grid.

The computation in the local array can start as soon as the halos computed remotely in the previous iteration have reached the device. The computation for the borders and the inner part can be launched as separate computations. The skeleton splits the computation and works in two stages (see Fig. 4). As soon as the computation of the local borders has finished, the data transfer and inter-process communication of the new border values can start. These data transfers and communications can be done asynchronously, overlapping with the computation in the inner part of the local array. For data sizes that really need parallel distributed computing, the computation load in the inner part of the local array is

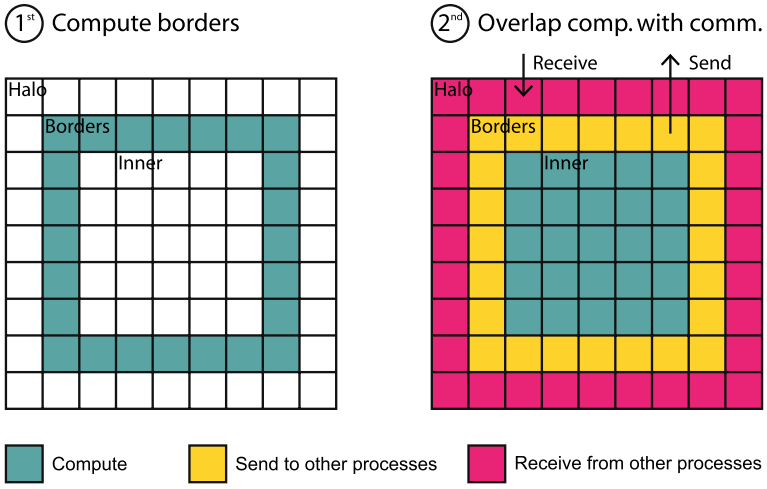

Fig. 4 Computations and data movements performed on each iteration of the proposed skeleton

far higher than in the borders, maximizing the opportunities to hide the latencies of data transfers and inter-process communications.

The Controllers model allows the launch of a kernel using HitTiles which are sub-selections of another tile. Thus, launching the kernels to compute the borders and the inner part separately is straightforward. The kernels are enqueued and executed asynchronously in the device. The Controllers runtime ensures that: (1) Requests for data transfers are not started until the kernels using the data are not yet finished; and (2) kernels are not started if there are pending data transfers for their parameters. Controllers offers an explicit synchronization operation to wait in the host until all the pending operations for a given *HitTile* have finished (kernel executions and/or data transfers). We use this to synchronize the start of the inter-process communications with the end of the data transfers of the borders from the device. This explicit synchronization is needed because these operations work on different tiles (although they are sub-selections of the same tile). Finally, the iterations loop is unrolled once, to avoid the unneeded communication of halos at the end of the last iteration.

Figure 5 shows an example image obtained with the NVIDIA Visual Profiler during the execution of some iterations of the skeleton in two processes with two NVIDIA Titan Black GPUs in two different nodes, interconnected with 10Gb Ethernet. It shows the computation-communication overlap achieved. The small blue rectangles represent the computation of the borders. Brown rectangles represent both the data transfers from GPU to CPU of the borders, and the data transfers from CPU to GPU of the received halos. Green rectangles represent the time dedicated to inter-process data communication. Finally, the big blue rectangles represent the time dedicated to the computation of the inner part of the local arrays. As can be seen, the computation of the inner part of the local arrays completely overlaps the CPU-GPU transfers and inter-process data communications.

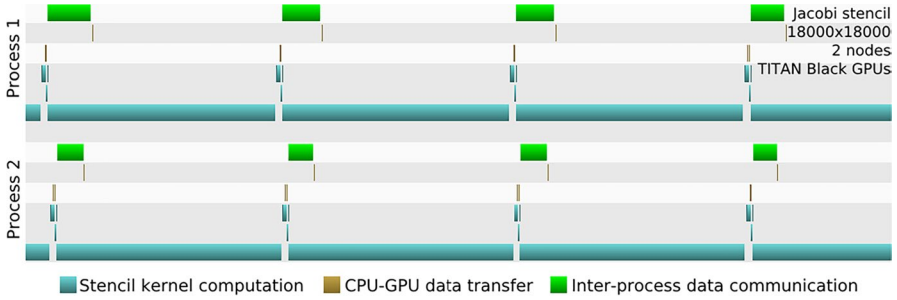

Fig. 5 Timeline from a profiling session of the execution of a Jacobi 2D stencil in 2 distributed NVIDIA Titan Black GPUs

5 Experimental studies

In this section, we discuss the results of three different experimental studies to test our proposal in different scenarios, with different types of stencils:

1. Strong and weak scalability of the skeleton in up to 48 NVIDIA GPUs in 12 distributed nodes.
2. Comparison with the strong and weak scalability obtained by a Celerity implementation of the same stencil computations, in up to 32 NVIDIA GPUs in eight distributed nodes.
3. Strong and weak scalability of the skeleton in a heterogeneous node comprising both NVIDIA and AMD GPUs, using load balancing techniques.

All tests were performed using single precision floating-point (FP32) calculations to leverage the highest amount of compute units in the GPUs. We have not considered half precision (FP16) representation, since it is not a native data type in certain GPUs, and our goal is to maximize compatibility across heterogeneous devices. Regarding double precision (FP64), its use is not optimal in terms of GPU computing resource utilization. Thus, it does not present the most demanding scenarios for synchronization and computation/communication overlapping.

5.1 Experimental study 1: strong and weak scaling in CTE-POWER

We have selected several stencil operators in 2D, including compact, non-compact, symmetric, and asymmetric stencils.

- 2D 4-point compact stencil (Jacobi).
- 2D 9-point compact stencil.
- 2D 9-point non-compact, second order stencil.
- 2D 5-point non-compact, asymmetric stencil.

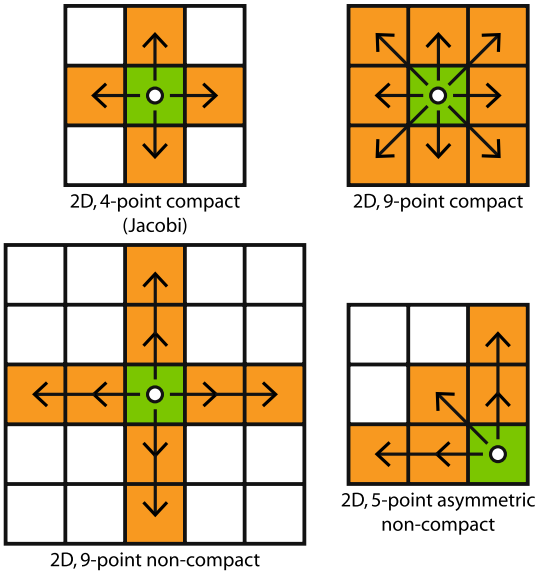

Fig. 6 Visual description of the 2D stencils tested

A visual representation of the 2D stencils described is shown in Fig. 6. For these stencils, we compare the performance of the EPSILOD generic kernel and the kernels automatically generated by the KGT tool (referred to as *specific kernels*).

The experimentation platform chosen is the CTE-POWER cluster of BSC (Barcelona Supercomputing Center). Each node has four NVIDIA 16GB Tesla V100 GPUs and two IBM Power9 8335-GTH @ 2.4GHz CPUs. They are interconnected with an Infiniband network. The stencils are executed in up to 48 GPUs in 12 distributed nodes. They are interconnected with an Infiniband network. The input array size for strong-scale experiments is $44\,000 \times 44\,000$, that is near the maximum size that fits in a 16GB Tesla V100 GPU. For weak-scaling analysis, the size of the array grows proportionally to the number of GPUs, increasing the size in one different dimension with each additional GPU. Each experiment executes 15 repetitions of 1000 iterations of each stencil, with enough load to obtain representative and stable results. Strong and weak scalability have been tested with the parallel skeleton proposed using both the generic and the automatically generated kernel implementations. Performance measures are the execution times of the program for the stencil computation part, excluding initialization and output of results.

The deviations of the measures are small in general. On specific experiments using 16 and 32 GPUs (four and eight nodes respectively), we obtained a high deviation with inconsistent mean values. The BSC's staff reported that some nodes for those configurations contained malfunctioning GPUs with lowered computational performance. The corresponding experiments were discarded and the malfunctioning GPUs were not used again. However, our quota did not allow us to re-run the experiments needed to obtain a meaningful average. Thus, we present results for the minimum execution time value observed for each experiment, which is consistent with the trend of the means for the normal performance scenarios.

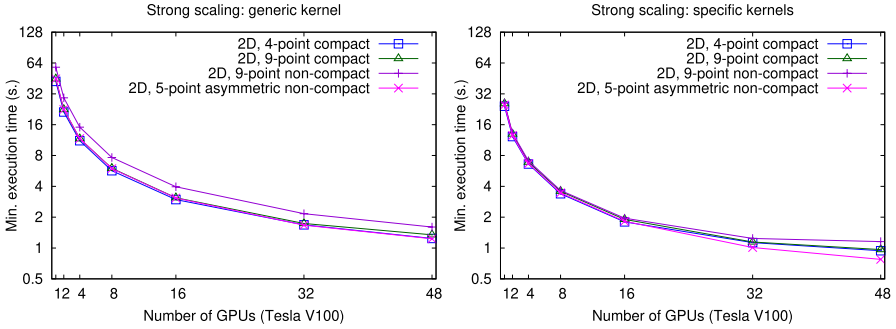

Fig. 7 Strong scalability in CTE-Power for 2D stencils. Left: Generic kernel. Right: Automatically generated kernels using the KGT tool described in Sect. 4.1

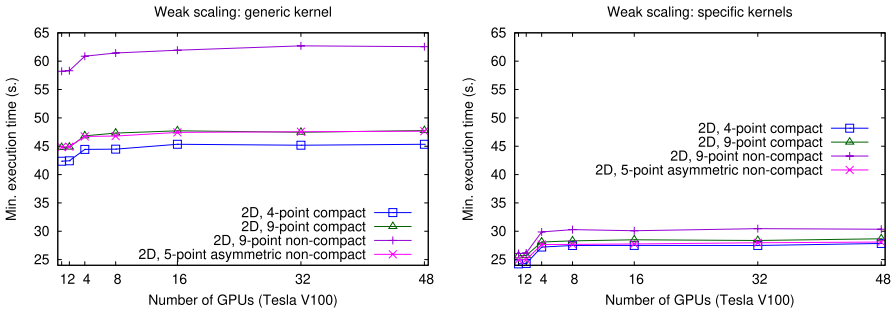

Fig. 8 Weak scalability in CTE-Power for 2D stencils. Left: Generic kernel. Right: Automatically generated kernels using the KGT tool described in Sect. 4.1

Figure 7 shows the execution times for the strong scalability tests with the 2D stencils, with logarithmic scale in the y-axis. Figure 8 shows the execution times for the weak scalability tests with the 2D stencils. In both cases, the plots on the left present the results with the generic kernel, and the plots on the right using the automatically generated kernel.

Table 2 shows the GFLOPS achieved by our proposal in the weak scalability tests for the 2D 9-point stencil. Only the GFLOPS for one of the tested stencils are provided for readability reasons. Nevertheless, all the other stencils behave similarly in terms of GFLOPS scalability. We choose to show the data for the 2D 9-point stencil, since it achieves the highest GFLOPS of all the tested stencils due to its high pattern density (i.e., high element utilization per memory line accessed).

From the results obtained we can make the following observations:

- In the weak scaling experiments, we observe smaller execution times when using one or two GPUs than when using four GPUs or more. This is explained by the architecture of the cluster nodes. There are four GPUs on each node

Table 2 GFLOPS achieved in weak scalability by our proposal, when executing the 2D 9-point stencil in CTE-Power using the generic kernel and a specific kernel generated by the Kernel Generation Tool

2D 9-point stencil GFLOPS	1 GPU 1 node	2 GPUs 1 node	4 GPUs 1 node	8 GPUs 2 nodes	16 GPUs 4 nodes	32 GPUs 8 nodes	48 GPUs 12 nodes
EPSILOD weak scalability, generic kernel	821.59	1640.67	3142.45	6219.52	12333.38	24800.17	36968.81
EPSILOD weak scalability, KGT-generated kernel	1282.60	2557.95	4684.85	9302.92	18478.60	37114.85	55087.86

grouped in two different zones, with different CPUs (NUMA nodes), and different power and ventilation areas. With four GPUs executing in the node, there are two devices executing in the same zone, and both devices experiment a little performance degradation.

- We also observe a very good weak scalability, specially for the optimized kernel. With up to four GPUs, inter-process communication is fast, as it is done internally in the node. With eight or more, there are also inter-process communications using the network. We observe no relevant execution time increment when using more than four GPUs, indicating that all the data transfers and communications are fully overlapped with the computation.
- The strong scaling results also show how good the overlapping is, with a parallel efficiency of more than 90% with up to 48 GPUs.
- Regarding performance, the execution time is dominated by the computation of the inner part. The kernels of these stencil computations are memory-bound. The performance differences between different stencils are mainly originated by the lower or higher memory bandwidth needed to access the neighbor elements. In general, non-compact stencils perform worse than compact ones. For stencils accessing a similar amount of memory lines, slight performance differences can be appreciated depending on the number of arithmetic operations per cell. These differences are more noticeable when using the less optimized generic kernel. For example, the 2d9 non-compact stencil presents the higher execution times. It accesses more memory lines than the others due to the higher stencil radius ($r = 2$ on each of the four directions). The 2d4 stencil is faster because it has a lower number of floating point operations per cell. In the case of the asymmetric 2d5 stencil, the effects of the higher radius on two directions, zero-radius in another two, and the number of floating point operations are compensated, obtaining a similar performance to the 2d9 compact stencil. Nevertheless, the communication pattern of the asymmetric 2d5 stencil derives in a pipeline structure, simpler than the full neighbor synchronization structure of the other stencils. Thus, its performance suffers less from occasional small delays on remote synchronizations. This is noticeable in the strong scaling tests when using the automatically generated kernel for 32 or 48 GPUs. The load of the inner part becomes so small that the inter-process synchronization effects become relevant. Also, in the same situations, the higher volume of communication of the 2d9 non-compact stencil leads to a little lower performance for a high number of GPUs.

5.2 Experimental study 2: comparison with celerity implementations

We have conducted an experimental study to compare the performance achieved using our proposal with the one obtained using Celerity [12], a state-of-the-art tool for distributed heterogeneous computing. Celerity uses MPI for inter-node communications, and SYCL to manage the heterogeneous devices. Strong and weak scalability have been tested for optimized implementations of the following stencil operations using both our parallel skeleton and a Celerity-based application:

- 2D 4-point compact stencil (Jacobi).
- 2D 9-point compact stencil.
- 2D 9-point non-compact, second order stencil.
- 2D 5-point non-compact, asymmetric stencil.

The experimentation platform is again the CTE-POWER cluster of BSC, comprising nodes with four NVIDIA Tesla V100 GPUs. We use Celerity 0.2.1, built on top of hipSYCL v0.9.1. Each experiment is executed three times. The number of iterations is 1 000 for all experiments. The stencils are executed in up to 32 GPUs in eight distributed nodes. In this case, we do not present results for 48 GPUs. The reason is that, in our experimental setup, we found that Celerity takes so much time in this configuration that it would consume our entire quota resources. The Controllers model has a much lower memory footprint than Celerity for the distributed arrays on the devices. Thus, it is not possible to experiment with the sizes used in the previous experimental study. For a fair comparison, the array sizes in this study are smaller. The array size for strong-scaling experiments is 30 000×30 000, which is near the maximum size that fits in a 16GB Tesla V100 GPU using Celerity. For weak-scaling analysis, the size of the array grows in all dimensions proportionally to the number of GPUs.

Figure 9 shows the comparison between the Celerity implementation and our proposal in terms of strong scaling, and Fig. 10 in terms of weak scaling.

Table 3 shows the GFLOPS achieved by the Celerity implementation and our proposal in the weak scalability tests of this experimental study, for the 2D 9-point stencil. As it was the case in the previous experimental study, the 2D 9-point stencil achieves the highest GFLOPS of all the tested stencils, for both the Celerity implementation and our proposal. It also shows a scaling behavior that is similar for all of the tested stencils. However, we also consider of interest to show the GFLOPS achieved by both implementations in the case of the 2D 4-point stencil, since it is the only scenario in which the Celerity implementation achieves higher GFLOPS than our proposal for some number of GPUs. That data are shown in Table 4.

As the load is smaller than in the previous experimental study, the impact of the data transfers and communications is higher, affecting scalability in some situations. It can be noted that our proposal outperforms the Celerity implementations for every type of stencil tested, except in the simplest stencil (the 2D 4-point stencil) when using fewer than 32 GPUs, which show a similar behavior. Our proposal has a better management of communications, reducing their number for non-symmetric stencils, and improving the computation-communication overlapping on each iteration.

Regarding the corresponding memory footprints, Celerity presents a memory footprint around 8 GB, while EPSILOD presents a memory footprint of less than 1 GB. Based on these results, our guess is that Celerity may be introducing extra copies of some matrix structures that are not needed for this specific kind of application, although the study of the exact reasons are beyond the scope of this paper.

Table 3 GFLOPS comparison for the weak scalability of a Celerity implementation and our proposal when executing the 2D 9-point stencil

2D 9-point stencil GFLOPS	1 GPU 1 node	2 GPUs 1 node	4 GPUs 1 node	8 GPUs 2 nodes	16 GPUs 4 nodes	32 GPUs 8 nodes
Celerity implementation, weak scalability	1238.37	2341.60	4238.52	8417.58	15777.26	30940.34
EPSILOD, KGT-generated kernel, weak scalability	1426.74	2830.06	4953.20	9785.37	17965.59	35145.91

Table 4 GFLOPS comparison for the weak scalability of a Celerity implementation and our proposal when executing the 2D 4-point stencil

2D 4-point stencil GFLOPS	1 GPU 1 node	2 GPUs 1 node	4 GPUs 1 node	8 GPUs 2 nodes	16 GPUs 4 nodes	32 GPUs 8 nodes
Celerity implementation, weak scalability	817.36	1520.13	2712.66	5361.13	10020.10	18011.26
EPSILOD, KGT-generated kernel, weak scalability	774.52	1536.50	2681.57	5287.03	9773.10	19390.49

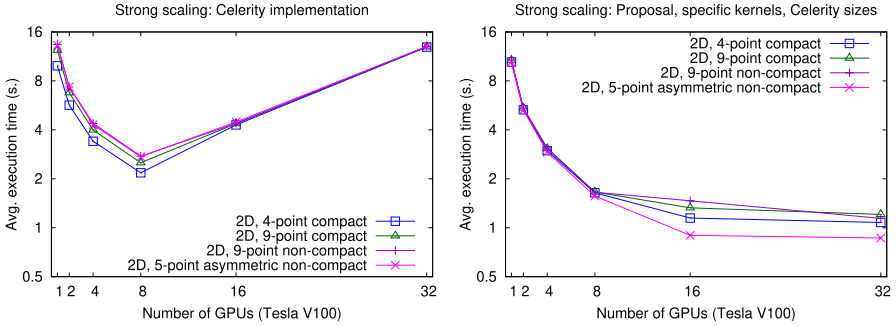

Fig. 9 Strong scaling comparison for a Celerity implementation and our proposal using different 2D stencils

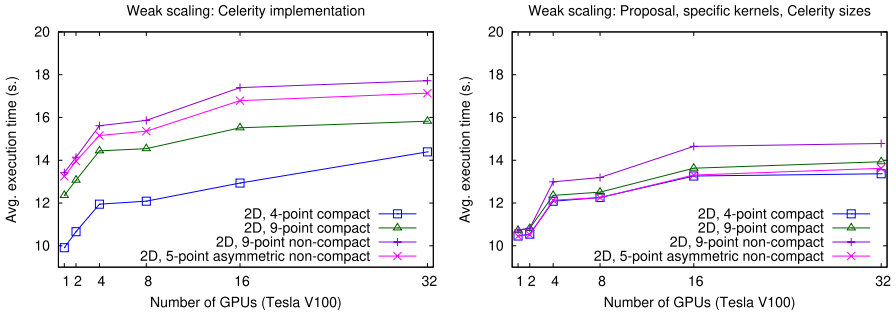

Fig. 10 Weak scaling comparison for a Celerity implementation and our proposal using different 2D stencils

5.3 Experimental study 3: using different GPU types

We conduct an experimental study to test the performance achieved with our proposal in a heterogeneous node which features different kinds of GPUs. We conduct the tests in the node Manticore, comprising 2 32GB NVIDIA Tesla V100 GPUs, and 2 AMD Radeon PRO WX 9100 GPUs. Weak scalability is tested for specific kernel implementations of the 2D stencils presented.

In this experiment, we first use the basic skeleton that maps the same amount of data to each process and device. Then, we use the skeleton version that uses static load-balancing techniques. This latter version uses a different layout function from the ones provided by the lower-level Hitmap library. It includes a parameter for the programmer to provide an array of weights that indicates the relative amount of data to supply to each specific GPU device. We have experimentally estimated the weights for each GPU in the node to redistribute the computing array in a balanced way across the processes. The weight for the V100 GPUs is normalized to 1 in all cases. The weights for the Radeon GPUs are 0.41 for the 2D 9-point non-compact, and 0.46 for the other stencils. For these weak-scaling experiments, we start with a size of $44\,000 \times 44\,000$ in a Tesla V100 GPU,

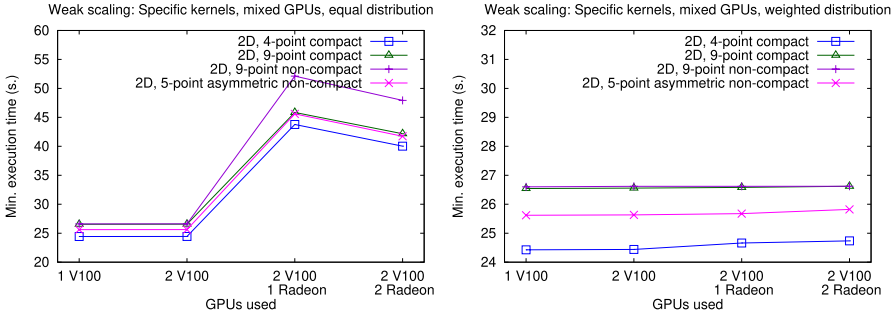

Fig. 11 Execution times for weak scaling tests using 2D stencils in a node with two different types of GPUs: NVIDIA V100, AMD Radeon PRO WX 9100. Left: Basic skeleton with the same amount of data per GPU. Right: Skeleton with irregular distribution to balance the workload

Table 5 GFLOPS comparison for the weak scalability of the basic skeleton and the skeleton with statically load-balanced array distribution, when executing the 2D 9-point stencil

2D 9-point stencil GFLOPS	1 Tesla V100	2 Tesla V100	2 Tesla V100, 1 Radeon PRO WX 9100	2 Tesla V100, 2 Radeon PRO WX 9100
EPSILOD, unbalanced distribution	1239.86	2478.34	1762.18	2216.63
EPSILOD, balanced distribution	1239.97	2478.48	3044.86	3657.75

and the array size increases in the first dimension proportionally to the accumulated weight of the GPUs used to execute the program. Figure 11 shows the execution times obtained. The results show that with minimal modifications, the proposed skeleton can use static load-balancing techniques and achieve good weak scalability for 2D stencils in heterogeneous platforms comprising different types of GPUs.

Table 5 shows the GFLOPS obtained in the case of the 2D 9-point stencil (the stencil that achieves the highest amount of GFLOPS). For readability reasons, only the GFLOPS for one of the tested stencils are provided. Nevertheless, the other stencils present a GFLOPS scaling behavior similar to the 2D 9-point stencil.

5.4 Summary of results

We can summarize our main findings as follows:

- Regarding strong and weak scaling in CTE-POWER, we observe a very good weak scalability, specially for the optimized kernel. We also observe that the execution time does not increase with more than four GPUs, indicating that all the data transfers and communications are fully overlapped with the computa-

tion. Strong scaling also demonstrate how good the overlapping is, with a parallel efficiency of more than 90% with up to 48 GPUs.
- Regarding the comparison with Celerity implementations, our proposal outperforms the Celerity implementations for every type of stencil tested, except in the simplest stencil when using fewer than 32 GPUs, where performance is similar. This is due to our better communications management.
- Regarding the use of different GPU types with different computation power simultaneously, the results show that simple, static load-balancing techniques allow to achieve good weak scalability with the proposed skeleton.

6 Conclusions

In this paper, we present EPSILOD, an improved parallel programming skeleton for iterative stencil computations on distributed multi-GPUs. It supports any type of n-dimensional geometric stencils (compact, non-compact, symmetric, or asymmetric) of any order. It is compatible with any C/C++ programming environment. The stencils are described with simple data structures that determine the neighbors and their weights. The data partition, synchronization and communications are automatically derived using that information. It splits the computation in the devices to maximize the overlapping of computation and communication. The skeleton provides a generic kernel, a tool to generate more specific and efficient kernels from the stencil description, and the possibility to use kernels manually written and optimized by the programmer. It exploits the Controllers model and the Hitmap library for an efficient management of the devices and MPI communications. We also provide an additional version supporting techniques to balance the load when using different types of GPUs in the same platform, from different vendors or with different computational power.

We present results for experimental studies testing the strong and weak scalability obtained for different stencils using a cluster with up to 12 nodes with 32 NVIDIA V100, or a heterogeneous node with 2 NVIDIA V100 GPUs and 2 AMD Radeon Pro WX 9100 GPUs. We also compare the scalability obtained with the implementation of the same stencils using a state-of-the-art programming framework for distributed heterogeneous devices: Celerity, built on top of MPI and SYCL. The results show that our proposal can obtain good strong and weak scalability for the 2D stencils tested, outperforming the Celerity implementations. They also show that a good scalability can be obtained using the integrated load-balancing techniques when exploiting nodes with different types of GPUs.

Future work includes improving the skeleton to support more heterogeneous environments with more types of devices, and including new optimization techniques. Regarding more complex iterative stencils with several computational stages, such as stencils that are second-order in time, staggered-grid stencils that read and write multiple grids over two stages, and algorithms with absorbing boundary conditions, among others, we plan to extend the functionalities of EPSILOD to support them. In any case, these improvements will extend the framework following the fundamentals of the solutions proposed in this work.

Acknowledgements This work has been funded by the Consejería de Educación of Junta de Castilla y León, Ministerio de Economía, Industria y Competitividad of Spain, European Regional Development Fund (ERDF) program: Project PCAS (TIN2017-88614-R) and Project PROPHET-2 (VA226P20). This work was supported in part by grant TED2021-130367B-I00 funded by MCIN/AEI/10.13039/501100011033 and by "European Union NextGenerationEU/PRTR". The authors thankfully acknowledges the computer resources at CTE-POWER and Minotauro and the technical support provided by Barcelona Supercomputing Center (RES-IM-2021-2-0005, RES-IM-2021-3-0024, RES-IM-2022-1-0014). The authors would also like to thank Carmelo de Castro for his help with the figures of this paper.

Author Contributions All authors wrote the main manuscript text. M.D.C., Y.T., and A.G.E. carry out the experimentation. M.D.C., Y.T., A.G.E., and D.R.L. generate the figures. All authors reviewed the manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. Consejería de Educación of Junta de Castilla y León, Spain: VA226P20. Ministerio de Economía, Industria y Competitividad of Spain, European Regional Development Fund (ERDF) program: TIN2017-88614-R. Red Española de Supercomputación (RES): RES-IM-2021-2-0005, RES-IM-2021-3-0024, RES-IM-2022-1-0014.

Declarations

Conflict of interest The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Ethical approval Not applicable.

Availability of data and materials The source code of both modules is freely available on the repository of the Trasgo group: https://gitlab.com/trasgo-group-valladolid/controllers.

References

1. Ao Y, Yang C, Wang X, Xue W, Fu H, Liu F, Gan L, Xu P, Ma W (2017) 26 pflops stencil computations for atmospheric modeling on sunway taihulight. In: 2017 IEEE International parallel and Distributed Processing symposium (IPDPS), pp 535–544. https://doi.org/10.1109/IPDPS.2017.9
2. Rossinelli D, Hejazialhosseini B, Hadjidoukas P, Bekas C, Curioni A, Bertsch A, Futral S, Schmidt SJ, Adams NA, Koumoutsakos P (2013) 11 pflop/s simulations of cloud cavitation collapse. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '13. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2503210.2504565
3. Shimokawabe T, Aoki T, Muroi C, Ishida J, Kawano K, Endo T, Nukada A, Maruyama N, Matsuoka S (2010) An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code'. In: SC '10: Proceedings of the 2010 ACM/IEEE International

Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–11. https://doi.org/10.1109/SC.2010.9
4. Shimokawabe T, Aoki T, Takaki T, Endo T, Yamanaka A, Maruyama N, Nukada A, Matsuoka S (2011) Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2063384.2063388
5. TOP500.org (2022) TOP 500 Main Page. https://www.top500.org/lists/top500/
6. NVIDIA (2022) CUDA Toolkit Documentation v11.7.0. http://docs.nvidia.com/cuda/, Last visit: May, 2022
7. Khronos (2022) Open Computing Language (OpenCL). http://www.khronos.org/opencl/, Last visit: May, 2022
8. Forum M (2022) Message Passing Interface (MPI). https://www.mpi-forum.org/, Last visit: May, 2022
9. de Castro M, Santamaria-Valenzuela I, Miguel-Lopez S, Torres Y, Gonzalez-Escribano A (2021) Towards an efficient parallel skeleton for generic iterative stencil computations in distributed gpus. In: SC21—ACM/IEEE Conference on High Performance Networking and Computing. https://sc21.supercomputing.org/proceedings/tech_poster/tech_poster_pages/rpost167.html
10. Moreton-Fernandez A, Ortega-Arranz H, Gonzalez-Escribano A (2018) Controllers: an abstraction to ease the use of hardware accelerators. Int J High Perform Comput Appl (IJHPCA) 32(6):838–853. https://doi.org/10.1177/1094342017702962
11. Gonzalez-Escribano A, Torres Y, Fresno J, Llanos DR (2014) An extensible system for multilevel automatic data partition and mapping. IEEE Trans Parallel Distrib Syst 25(5):1145–1154. https://doi.org/10.1109/TPDS.2013.83
12. Thoman P, Salzmann P, Cosenza B, Fahringer T (2019) Celerity: high-level C++ for accelerator clusters. In: Yahyapour R (ed) Euro-Par 2019: parallel processing. Springer, Cham, pp 291–303. https://doi.org/10.1007/978-3-030-29400-7_21
13. Sourouri M, Langguth J, Spiga F, Baden SB, Cai X (2015) CPU+ GPU programming of stencil computations for resource-efficient use of GPU clusters. In: 2015 IEEE 18th International Conference on Computational Science and Engineering, pp 17–26. https://doi.org/10.1109/CSE.2015.33
14. Feichtinger C, Habich J, KöStler H, Hager G, RüDe U, Wellein G (2011) A flexible patch-based lattice Boltzmann parallelization approach for heterogeneous GPU–CPU clusters. Parallel Comput 37(9):536–549. https://doi.org/10.1016/j.parco.2011.03.005
15. Shimokawabe T, Aoki T, Ishida J, Kawano K, Muroi C (2011) 145 TFlops performance on 3990 GPUs of TSUBAME 2.0 supercomputer for an operational weather prediction. In: Proceedings of the International Conference on Computational Science, ICCS 2011, Nanyang Technological University, Singapore, 1-3 June, 2011, pp 1535–1544. https://doi.org/10.1016/j.procs.2011.04.166
16. Shimokawabe T, Aoki T, Muroi C, Ishida J, Kawano K, Endo T, Nukada A, Maruyama N, Matsuoka S (2010) An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC '10, pp 1–11. IEEE Computer Society, Washington, DC, USA. https://doi.org/10.1109/SC.2010.9
17. Shimokawabe T, Aoki T, Takaki T, Endo T, Yamanaka A, Maruyama N, Nukada A, Matsuoka S (2011) Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11, pp 3–1311. ACM, New York, NY, USA. https://doi.org/10.1145/2063384.2063388
18. Schäfer A, Fey D (2008) libgeodecomp: a grid-enabled library for geometric decomposition codes. In: Lastovetsky A, Kechadi T, Dongarra J (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer, Berlin, pp 285–294. https://doi.org/10.1007/978-3-540-87475-1_39
19. Stark DT, Barrett RF, Grant RE, Olivier SL, Pedretti KT, Vaughan CT (2014) Early experiences co-scheduling work and communication tasks for hybrid MPI+ X applications. In: 2014 Workshop on Exascale MPI at Supercomputing Conference, pp 9–19. https://doi.org/10.1109/ExaMPI.2014.6
20. Chakroun I, Vander Aa T, De Fraine B, Haber T, Wuyts R, Demeuter W (2015) Exashark: A scalable hybrid array kit for exascale simulation. In: Proceedings of the Symposium on High Performance Computing. HPC '15, pp 41–48. Society for Computer Simulation International, San Diego, CA, USA. http://dl.acm.org/citation.cfm?id=2872599.2872605

21. Baskaran M, Pradelle B, Meister B, Konstantinidis A, Lethin R (2016) Automatic code generation and data management for an asynchronous task-based runtime. In: 2016 5th Workshop on Extreme-Scale Programming Tools (ESPT), pp 34–41. https://doi.org/10.1109/ESPT.2016.009
22. Bachan J, Bonachea D, Hargrove PH, Hofmeyr S, Jacquelin M, Kamil A, van Straalen B, Baden SB (2017) The UPC++ PGAS library for exascale computing. In: Proceedings of the Second Annual PGAS Applications Workshop. PAW17, pp 7–174. ACM, New York, NY, USA. https://doi.org/10.1145/3144779.3169108
23. Tanaka H, Ishihara Y, Sakamoto R, Nakamura T, Kimura Y, Nitadori K, Tsubouchi M, Makino J (2018) Automatic generation of high-order finite-difference code with temporal blocking for extreme-scale many-core systems. In: 2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), pp. 29–36. https://doi.org/10.1109/ESPM2.2018.00008
24. Kronawitter S, Lengauer C (2018) Polyhedral search space exploration in the exastencils code generator. ACM Trans Archit Code Optim 15(4):40–14025. https://doi.org/10.1145/3274653
25. Luporini F, Louboutin M, Lange M, Kukreja N, Witte P, Hückelheim J, Yount C, Kelly PHJ, Herrmann FJ, Gorman GJ (2020) Architecture and performance of devito, a system for automated stencil computation. ACM Trans Math Softw. https://doi.org/10.1145/3374916
26. Hagedorn B, Stoltzfus L, Steuwer M, Gorlatch S, Dubach C (2018) High performance stencil code generation with lift. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization. CGO 2018, pp 100–112. ACM, New York, NY, USA. https://doi.org/10.1145/3168824
27. Pereira AD, Castro M, Dantas MAR, Rocha RCO, Góes LFW (2017) Extending OpenACC for efficient stencil code generation and execution by skeleton frameworks. In: 2017 International Conference on High Performance Computing Simulation (HPCS), pp 719–726. https://doi.org/10.1109/HPCS.2017.110
28. Schäfer A, Fey D (2011) High performance stencil code algorithms for GPGPUs. Procedia Comput Sci 4:2027–2036. https://doi.org/10.1016/j.procs.2011.04.221. (**Proceedings of the International Conference on Computational Science, ICCS 2011**)
29. Anjum O, Simon GdG, Hidayetoglu M, Hwu W-M (2019) An efficient GPU implementation technique for higher-order 3d stencils. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp 552–561. https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00086
30. Matsumura K, Zohouri HR, Wahib M, Endo T, Matsuoka S (2020) AN5D: automated stencil framework for high-degree temporal blocking on GPUs. In: Proceedings of the 18th International Symposium on Code Generation and Optimization, pp 199–211. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3368826.3377904
31. Rawat PS, Vaidya M, Sukumaran-Rajam A, Rountev A, Pouchet L-N, Sadayappan P (2019) On optimizing complex stencils on GPUs. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp 641–652. https://doi.org/10.1109/IPDPS.2019.00073
32. Oh C, Zheng Z, Shen X, Zhai J, Yi Y (2020) Gopipe: A granularity-oblivious programming framework for pipelined stencil executions on GPU. In: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques. PACT '20, pp 43–54. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3410463.3414656
33. Pereira AD, Ramos L, Góes LFW (2015) Pskel: a stencil programming framework for CPU–GPU systems. Concurr Comput Pract Exper 27(17):4938–4953. https://doi.org/10.1002/cpe.3479
34. Viñas M, Fraguela BB, Andrade D, Doallo R (2017) Facilitating the development of stencil applications using the heterogeneous programming library. Concurr Comput Pract Exp 29(12):4152. https://doi.org/10.1002/cpe.4152
35. Steuwer M, Haidl M, Breuer S, Gorlatch S (2014) High-level programming of stencil computations on multi-GPU systems using the SkelCL library. Parallel Process Lett 24(03):1441005. https://doi.org/10.1142/S0129626414410059
36. Maruyama N, Sato K, Nomura T, Matsuoka S (2011) Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In: SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–12. https://doi.org/10.1145/2063384.2063398

37. Lutz T, Fensch C, Cole M (2013) Partans: an autotuning framework for stencil computation on multi-GPU systems. ACM Trans Archit Code Optim 9(4):59–15924. https://doi.org/10.1145/2400682.2400718
38. Shimokawabe T, Aoki T, Onodera N (2014) A high-productivity framework for multi-gpu computation of mesh-based applications. In: Grösslinger A, Köstler H (eds), Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna, Austria, pp 23–30. https://hgpu.org/?p=11286
39. Breuer S, Steuwer M, Gorlatch S (2014) Extending the SkelCL skeleton library for stencil computations on multi-GPU systems. In: HiStencils 2014, First International Workshop on High-Performance Stencil Computations, pp 1–13. https://hgpu.org/?p=11368
40. Aldinucci M, Danelutto M, Drocco M, Kilpatrick P, Misale C, Peretti Pezzi G, Torquati M (2018) A parallel pattern for iterative stencil + reduce. J Supercomput 74(11):5690–5705. https://doi.org/10.1007/s11227-016-1871-z
41. Kim H, Hadidi R, Nai L, Kim H, Jayasena N, Eckert Y, Kayiran O, Loh G (2018) Coda: enabling co-location of computation and data for multiple GPU systems. ACM Trans Archit Code Optim 15(3):32–13223. https://doi.org/10.1145/3232521
42. Herrmann N, de Melo Menezes BA, Kuchen H (2022) Stencil calculations with algorithmic skeletons for heterogeneous computing environments. Int J Parallel Program 50(5):433–453. https://doi.org/10.1007/s10766-022-00735-4
43. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Parallel Programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco
44. Tian S, Doerfert J, Chapman B (2020) Extending the SkelCL skeleton library for stencil computations on multi-GPU systems. In: Fourth LLVM Performance Workshop at CGO. https://llvm.org/devmtg/2020-02-23/
45. Beckingsale DA, Burmark J, Hornung R, Jones H, Killian W, Kunen AJ, Pearce O, Robinson P, Ryujin BS, Scogland TR (2019) Raja: Portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012
46. Beckingsale DA, Burmark J, Hornung R, Jones H, Killian W, Kunen AJ, Pearce O, Robinson P, Ryujin BS, Scogland TR (2019) Raja: Portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). IEEE, New York, NY, USA. https://doi.org/10.1109/P3HPC49587.2019. IEEE/ACM
47. Edwards HC, Trott CR, Sunderland D (2014) Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J Parallel Distrib Comput 74(12):3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003. (**Domain-Specific Languages and High-Level Frameworks for High-Performance Computing**)
48. Trott CR, Lebrun-Grandié D, Arndt D, Ciesko J, Dang V, Ellingwood N, Gayatri R, Harvey E, Hollman DS, Ibanez D, Liber N, Madsen J, Miles J, Poliakoff D, Powell A, Rajamanickam S, Simberg M, Sunderland D, Turcksin B, Wilke J (2022) Kokkos 3: Programming model extensions for the exascale era. IEEE Trans Parallel Distrib Syst 33(4):805–817. https://doi.org/10.1109/TPDS.2021.3097283
49. Ciesko J (2020) Distributed memory programming and multi-GPU Support with KOKKOS. Presented at SC'20. https://doi.org/10.2172/1829951. https://www.osti.gov/biblio/1829951
50. Khronos OpenCL working group (2020) SYCL 1.2.1 specification standard. (accessed February 1, 2022). https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
51. Gorlatch S, Cole M (2011) In: Padua D (ed), Parallel Skeletons, pp 1417–1422. Springer, Boston. https://doi.org/10.1007/978-0-387-09766-4_24
52. Rodriguez-Canal G, Torres Y, Andujar FJ, Gonzalez-Escribano A (2021) Efficient heterogeneous programming with FPGAs using the Controller model. J Supercomput 77:1–16. https://doi.org/10.1007/s11227-021-03792-7