Simulation Modelling Practice and Theory 87 (2018) 48–72

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Assessment of offload-based programming environments for hybrid CPU–MIC platforms in numerical modeling of solidification

Kamil Halbiniak, Roman Wyrzykowski, Lukasz Szustak*, Tomasz Olas

Czestochowa University of Technology Dabrowskiego 69, Czestochowa 42-201, Poland

ARTICLE INFO

Keywords:
Intel Xeon Phi
KNC coprocessor
KNL processor
Hetero Streams
Intel Offload
OpenMP
Numerical modeling of solidification

ABSTRACT

Heterogeneous (or hybrid) computing platforms with Intel Xeon Phi accelerators offer potential advantages of energy efficient, massively parallel computing, while supporting parallel programming models familiar for users of multicore CPUs. However, realizing this potential for real-world applications still remains a challenging issue. The main goal of this paper is the suitability assessment of offload-based programming environments for porting a real-life scientific application to hybrid platforms with Intel KNC and KNL accelerators, assuming no significant modifications of the application code. The main criterion of this assessment is the application performance. The evaluated environments include: 1) Intel Offload coupled with OpenMP, 2) OpenMP 4.0 and 3) OpenMP 4.5 Accelerator Models, and 4) hStreams Library with OpenMP. A real-life application dedicated to the numerical modeling of alloy solidification is used as a testbed in the assessment. An experimental evaluation of the four versions of the application code for a platform with KNC coprocessors shows that excluding OpenMP 4.0, the rest of them are able to adapt to expansion of available resources, however, with different efficiency. While the shortest execution time is achieved for Intel Offload, the high-level abstractions of hStreams contribute considerably to making porting and tuning the application easier, with low performance overheads in comparison to the low-level Intel Offload extension. Benchmarking the application performance and scalability on a platform with multiple KNL processors, using the Offload over Fabric technology with Intel Offload and OpenMP 4.5, concludes the assessment.

1. Introduction

Heterogeneous (or hybrid) computing platforms have become increasingly attractive in many application domains [34,41,52]. The combination of a general-purpose host CPU coupled with specialized computing devices (e.g., GPU, Intel Xeon Phi or FPGA) in many cases allow users to accelerate their applications significantly [24,40], as well as to increase the *performance per Watt ratio*. However, realizing these potentials in practice still remains a challenging issue.

Typically, the host of a hybrid platform controls the code execution, while the time consuming part of code is offloaded to the coprocessor (accelerators). Offloading needs to transfer data between the host and device before and after executing computing kernels. The performance overhead generated by these transfers determine where offloading is worth to utilize. To amortize these overheads, the execution of kernel is overlapped with data movements [33]. Another way of improving the overall performance of hybrid platforms is using the CPU host not only for the management of code execution, but also to perform computing-intensive pieces of code or/and other operations like storage accesses [44].

* Corresponding author.
E-mail address: lszustak@icis.pcz.pl (L. Szustak).

https://doi.org/10.1016/j.simpat.2018.06.002
Received 3 January 2018; Received in revised form 7 June 2018; Accepted 8 June 2018
Available online 15 June 2018
1569-190X/

The Intel Xeon Phi accelerators are based on the Intel Many Integrated Core (MIC) architecture [37,47]. It is designed for massively parallel applications, and includes a large number of cores with wide vector processing units. The important advantage of MIC architecture is support of parallel programming models familiar to users of conventional multicore CPUs. The first generation of Intel Xeon Phi devices known as Knight Corner (KNC) [18,37] is available as coprocessors connected to CPU through the PCIe bus. The second generation known as Knight Landing (KNL) [20] is delivered as a standalone, self-boot processor.

Effective parallelism is difficult to achieve in heterogeneous architectures, especially when both CPU and MIC devices are used to perform computing-intensive parts of code. Besides achieving high performance, the cost of software development becomes another bottleneck preventing the wide adoption of hybrid platforms. So there is a considerable interest in investigating the influence of programming environments on the efficiency of porting real-world applications to CPU–MIC platforms. In this work, we study the practical usage of the heterogeneous programming model â- a mixture of offload programming and shared-memory approaches.

The main goal of this paper is the suitability assessment of various programming environments implementing this model when porting a real-life scientific application to hybrid platforms with Intel Xeon Phi accelerators, assuming no significant modifications of the application code. The first environment is the Intel Compiler Assisted Offload [6] coupled with OpenMP [36], the most popular standard of programming shared-memory platforms. The next two ones are different versions of the OpenMP Accelerator Model [26,27,36], an extension of OpenMP. The last one is a mixture of OpenMP with the Hetero Streams Library [13,34], a new heterogeneous streaming framework which offers a higher level of abstraction to provide effective concurrency among tasks.

Following our works [10,44–46], a real-life scientific application is used as a testbed in the assessment. This application is dedicated to the numerical modeling of alloy solidification. The numerical scheme is based on the phase-field approach [48] with the generalized finite difference method (FDM) and explicit scheme of calculations. The basic assumption of our research is to accelerate the application by using efficiently the computing resources of hybrid platforms with Intel CPUs and MIC accelerators, without significant modifications of the code.

The contributions of this paper to areas of parallel computing and high performance computing simulation are as follows:

1. The sequence of steps required for porting and optimizing real-life scientific applications on hybrid CPU–MIC platforms, without significant modification of application codes, is thoroughly reformulated in comparison with [44]. The proposed approach allows us to minimize the amount of data transfers, as well as to overlap the following operations: 1) computation, 2) inter-device communication, and 3) writing outcomes to the file. Also, it becomes possible to achieve the flexible load balancing of workloads between devices, and to optimize the utilization of resources of devices.
2. The suitability assessment of four programming environments implementing the offload-based heterogeneous programming model is provided based on the way how these environments support the implementation of the proposed approach for the solidification modeling application. The main criterion of this assessment is the application performance. Among the considered aspects which differentiate these environments are:
   - management of accelerators;
   - way of separation of computations and writing results to the file;
   - mechanisms used for the inter-device and intra-CPU synchronization;
   - way of parallelizing the workload execution within devices.
3. An experimental evaluation of four versions of the solidification application for a platform with KNC coprocessors shows that three of them (using Intel Offload, OpenMP 4.5 Acceleator Model, and hStreams Library) are able to adapt to expansion of available resources, however, with not the same efficiency. The shortest execution time is achieved for Intel Offload. At the same time, the high-level abstractions of hStreams contribute considerably to making porting and tuning the application easier, with low performance overheads in comparison with the low-level Intel Offload extension.
4. Benchmarking the application performance and scalability on a platform with multiple KNL processors, using the Offload over Fabric (OoF) technology with Intel Offload and OpenMP 4.5, concludes the assessment. The results of benchmarking allow us to reveal capabilities and limitations of the offload programming model based on OoF. In particular, the usage of 4 KNLs permits the speedup of about 6.7x and 3.45x against the optimized OpenMP code on 2 CPUs, for respectively the static and dynamic intensity of computations.

The material of the paper is organized as follows. Section 2 contains an introduction to the heterogeneous programming model based on offloading, including the overview of hybrid platforms with KNC coprocessors. Section 3 provides a concise overview of programming environments considered in this paper, and the next section outlines the solidification application, which is used to assess these environments. An approach to adapting the application to hybrid CPU–MIC platforms is proposed in Section 5, while Section 6 describes how this approach is further adjusted to the considered environments. Section 7 presents performance results with discussion. While the focus of Sections 2–7 is on KNC coprocessors, Section 8 extends our research to include platforms with KNL processors, where CPU and MIC devices are connected by HPC fabric. Related works are discussed in Section 9, while Section 10 concludes the paper and addresses future works.

2. Offload-based heterogeneous programming model

2.1. Overview of hybrid CPU–MIC platforms with KNC coprocessors

The hybrid CPU–MIC architecture combines in various proportion two types of components: 1) general-purpose Intel Xeon

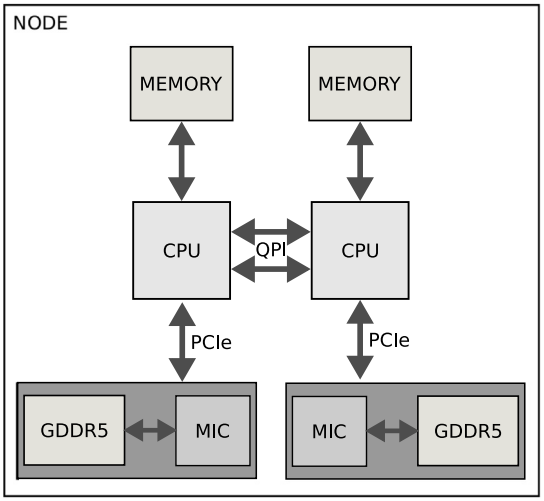

Fig. 1. Hybrid architecture combining CPUs and KNC coprocessors [44].

processors, and 2) Intel Xeon Phi devices designed for massively parallel computing. While HPC servers with KNC coprocessors come in a variety of configurations [29,32,37], the most common platforms usually contain two CPUs, and one or two coprocessors.

The KNC coprocessor contains more than 50 cores, caches, and necessary control subsystems [18]. All these components are connected together by the bidirectional ring interconnect. Cores are clocked at about 1 GHz, and allow running up to 4 hardware threads per core. An integral part of every core is the vector processing unit, that supports a 512-bit SIMD instruction set. Each core has 128 vector registers 512-bit wide, and is equipped with private L1 and L2 caches that are kept fully coherent by the ring interconnect. The coprocessor has over 6 GB of own on-board GDDR5 memory (maximum 16 GB). Fig. 1 presents an example of the hybrid CPU–MIC platform with two CPUs and two coprocessors. Following the concept of ccNUMA architectures, CPU processors are connected via the QPI bus. KNC coprocessors are delivered as PCIe devices, so exchanging data between the main memory and coprocessors is possible only via the PCIe bus.

The KNC coprocessors (as well as KNL processors) provide a general-purpose programming environment similar to that available for Intel CPUs [37]. It supports the source-code portability between CPU and MIC devices, that gives possibility to run the same code using different devices. Programmers can write their codes using the most popular programming languages like C/C++ and Fortran. This architecture supports also traditional parallel programming standards [9] such as OpenMP, Intel TBB, C++11 threads and MPI.

The coprocessor allows three execution modes that can be used to design and execute applications: 1) offload mode, 2) native mode, and 3) symmetric one. In this research, we use the offload mode to force Intel Xeon Phi to work together with CPU. In this mode, the programmer determines a section of the source code to run on coprocessors. When the Intel compiler encounters the region to be offloaded, it generates the binary for Xeon Phi. The resulting code starts on the host side, while the selected regions are transferred to the coprocessor using runtime mechanisms provided by the Intel MPSS software [17].

2.2. Basics of heterogeneous programming model

For hybrid platforms, one of ways to achieve the high performance of computations is to take advantage of both CPUs and coprocessors using the heterogeneous programming model based on a mixture of shared-memory and offload programming models. While the first one is used to harness the resources of cores/threads within both CPUs and MIC devices, the second one is utilized for offloading data and computations to coprocessors. The typical choices for shared-memory programming are: OpenMP, C++ threads, Intel TBB, Intel Cilk Plus and POSIX threads. The popular solutions for offloading codes to Intel Xeon Phi coprocessors include Intel Compiler Assisted Offload, and OpenMP Accelerator Model.

Among the main issues of achieving effective parallelism in hybrid CPU–MIC platforms are:

1. flexible management of computing resources;
2. minimization of overheads for management of memories, both in coprocessors and CPUs;
3. low-cost intra- and inter-device synchronizations of CPU/MIC devices;
4. minimization of overheads for communication between CPUs and MICs, including overlapping communication and computation.

The typical solution for management of computing resources is the host-centric model, when the CPU host transfers (or offloads) data and binaries to coprocessors. The offloaded code is then executed using resources of coprocessors in a parallel way. An important requirement is providing a suitable affinity of threads and cores, which is indispensable to minimize performance overheads. A rather common issue is also assuring a required partition of cores, for both CPU and coprocessors, in order to execute various parts of the application code.

Multiple allocations and deallocations of memories in MIC devices could considerably deteriorate the overall performance. An efficient way to alleviate this bottleneck is providing the flexible support for the data persistence between multiple calls of offloaded regions of code. At the same time, it is responsibility of programmers to use such optimization techniques as double buffering on the CPU side. This allows performing computations by CPUs and offloading data and computations to coprocessors concurrently.

Minimizing synchronization overheads is vital to optimize the overall performance of parallel computing. In the case of hybrid platforms, besides usual synchronization issues inside CPU and MIC devices, there are overheads of synchronizing activities performed by devices of different kinds (CPUs and MICs). As especially expensive, these inter-device synchronizations should be avoided as much as possible. Thus, the asynchronous approach has to be used whenever possible. This is mainly the responsibility of programmers, but a programming environment has to provide an efficient and convenient support for the asynchronous interaction.

The significance of such a support is increased drastically taking into account huge possibilities for minimizing overheads of inter-device communication by overlapping communication and computation. First, it is important to assure transferring data between CPUs and MICs during computations performed by coprocessors (asynchronous transfer from the coprocessor point of view). Secondly, enabling asynchronous computations on the host and inter-device data transfers requires the use of a non-blocking mechanism for offloading the code by the host.

3. Programming environments for CPU–MIC platforms

3.1. Intel Compiler Assisted Offload

The Intel Compiler Assisted Offload extension (Intel Offload shortly) [6,37] is a method to offload computations to Intel Xeon Phi that uses the Intel compiler and its offload pragma support to manage the computations and offloaded data. It is obligation of programmers to select a section of the source code to run on coprocessors. This extension is also known as Intel Language Extension for Offload (LEO) [33].

The execution model of Intel Offload is based on the host-centric view, where the host transfers data and computations to coprocessors, using **offload target (mic)** construct. The code region which is transferred to the MIC device is executed by one thread, that can spawn multiple threads using an appropriate parallel construct. On platforms with multiple coprocessors, Intel Offload allows determining explicitly which coprocessor is selected for the execution.

In the offload mode, the utilization of coprocessors requires as a rule exchanging data between the host and coprocessors. Transferring data to coprocessors is possible using the following clauses: **in, out**, and **inout**. These clauses enable the implicit allocation of memory and determining the direction of data transfer. At the same time, Intel Offload permits omitting the exchange of data between the host and MIC devices by utilizing **nocopy** clause.

Using **offload target (mic)** directive to offload data and computation by default results in the allocation and deallocation of the coprocessor memory. When considering an application with multiple calls of offloaded regions, the memory allocations usually generate considerable performance overheads. To resolve this problem, Intel Offload gives the opportunity to control allocations and deallocations of memory using **alloc_if** and **free_if** modifiers, respectively. These modifiers work with **in, out, inout** and **nocopy** clauses, and specify how data are allocated and deallocated before and after the execution of offloaded regions. In consequence, this solution ensures the data persistence between multiple calls of these regions.

To offload solely data to MIC devices, the programming interface provides also **offload_transfer target (mic)** construction. This construction allows transferring data during computations executed by coprocessors (asynchronous transfer from the point of view of coprocessors). Specifying direction of data transfers is possible using **in** and **out** clauses. This mechanism enables also controlling the data persistence using the aforementioned **alloc_if** and **free_if** qualifiers.

The Intel offload model permits the asynchronous execution of **offload target** and **offload_transfer** directives. By default, a thread calling any of these pragmas is blocked until their completion. Enabling asynchronous computations and data transfers (from the host point of view) requires use of **signal** clause. In consequence, the host can immediately continue the execution of program. The synchronization of asynchronous activities is possible through **wait** clause of the dedicated **offload_wait target (mic)** directive. As a result, a thread that encounters this directive waits for the completion of asynchronous execution.

An important directive of Intel Offload is **offload_target**. It is used to specify the region of source code (functions and variables) available for coprocessors. It allows the compiler to generate the binaries which can be directly called from the offloaded region. A snippet of the source code written using Intel Offload is shown in Listing 1. It corresponds to offloading both the data arrays A, B, C, and computations performed on these arrays by the coprocessor, for the given number of time steps. While **in** clause of **offload_transfer** directive (Line 1) is used to transfer the input arrays B and C to the coprocessor, as well as to initalize the output array A, **out** clause of **offload target** directive (Line 7) is responsible for returning computed values of the array A to the host. The presented code utilizes **signal** clause (Line 9) for the asynchronous execution of the offloaded region. As a result, the host starts its activity (Line 17) immediately after initializing computations on the coprocessor. Then the activities of the host and coprocessor are synchronized using **offload_wait** directive (Line 18).

3.2. OpenMP Accelerator Model: OpenMP 4.0

OpenMP is the directive-based, parallel programming standard that supports multi-platform shared-memory parallel programming in C/C++ and Fortran [36]. Starting with version 4.0, the execution model of OpenMP is extended by the mechanism called the OpenMP Accelerator Model (OMPAM in short) [26]. It is design to simplify programming heterogeneous platforms with many-

```
1  #pragma offload_transfer target(mic) \
2     in(A : length(n) alloc_if(1) free_if(0)) \
3     in(B : length(n) alloc_if(1) free_if(0)) \
4     in(C : length(n) alloc_if(1) free_if(0))
5
6  for(int t=0; t<num_steps; ++t) {
7     #pragma offload target(mic) nocopy(B) nocopy(C) \
8        out(A : length(n) alloc_if(0) free_if(0)) \
9        signal(&s0)
10    {
11       //MIC computations
12       #pragma omp parallel for
13       for (int i=1; i<n-1; ++i ) {
14          A[i] = C[i] * (B[i-1] + B[i] + B[i+1]);
15       }
16    }
17    host_activity();
18    #pragma offload_wait target(mic) wait(&s0)
19 }
```

Listing 1. Offloading computations and data with Intel Offload.

core accelerators such as Intel MICs or GPUs. This extension assumes that a computing platform consists of multiple target devices connected to the host.

Similar to Intel offload, the execution model of OMPAM is based on the host-centric view, where **target** construct is used to copy (offload) data and computations to coprocessors. By default, the code region offloaded to an coprocessor is executed using only one thread, until it encounters an adequate parallel construction. Using **device** clause, OMPAM enables specifying a device where the target region has to be executed.

The OpenMP Accelerator Model–similar to Intel Offload -allows the management of memory allocation for target devices. It provides **target data** construct which creates the device data environment based on host buffers mapped to the accelerator. This construct specifies a single block of code where the accelerator memory is allocated on entry to the block, and deallocated on exit. This solution enables reusing the same buffers among multiple target regions.

Defining the data movements between the host and target devices is implemented using **map** clause with four data-mapping attributes that specify how buffers are allocated, initialized and copied to devices. The first three attributes: **to, from**, and **tofrom**, allow the implicit allocation of device memory and determination of data copying direction. The last attribute, **alloc** is used when the explicit allocation of device memory is required.

An important construction of OMPAM is **target update**. It allows the synchronization of host and device memory buffers, and can be used only within the device data environment. The direction of update is specified using two clauses: **to** and **from**, that provide the list of synchronized buffers corresponding to variables in the device data region. Another directive of OMPAM, **declare target** is used to determine regions of the source code mapped to the target device.

The important advantage of OMPAM is the provision of support for multiple hybrid platforms, through a growing number of compilers. This allows achieving the code portability, and makes this choice especially interesting for developers of parallel codes for platforms with various accelerators.

The code presented in Listing 1 is rewritten in Listing 2, using OpenMP 4.0. Here **map** clause of **target data** construct (Line 1) provides transferring the input arrays B and C to the coprocessor, as well as initializing the output array A, whilst the computed values of the array A are returned to the host using **from** clause of **target update** construct (Line 13). Since OpenMP 4.0 does not provide the asynchronous execution of **target** directive, the simultaneous execution of host and coprocessor activities is ensured through the OpenMP task parallelism. In the presented code, three threads are created. While the master thread of the parallel region is responsible only for spawning tasks, the other two threads provides the execution of tasks corresponding to the host and coprocessor activities, that are synchronized using **taskwait** construct (Line 18).

3.3. OpenMP Accelerator Model: OpenMP 4.5

Starting with OpenMP 4.5, the OpenMP accelerator model allows the asynchronous execution of **target** directive using **nowait** clause [36]. By default, a thread that encounters **target** construct is blocked until the completion of execution. At the same time, using **target nowait** clause, the programmer can guarantee that the target region is launched in the background, and a thread can immediately continue execution of the program serially or simultaneously with other threads in a parallel region. This clause works also with **target update** directive. In this case, the synchronization of the host and device buffers is performed in the background. The completion of an asynchronous activity is controlled using **taskwait** construct.

OpenMP 4.5 introduces two additional clauses, **enter data** and **exit data** [36], for the management of device memory. They give programmers more flexibility in creating a device data environment which is not associated with a single block of code. As a result, the device memory may be deallocated before the completion of application execution.

Again Listing 3 presents the code of Listing 1 rewritten with OpenMP 4.5. Now **map** clause of **target enter data** construction

```
1  #pragma omp target data map(to: B[0:n], C[0:n]) \
2     map(alloc: A[0:n])
3  {
4     #pragma omp parallel num_threads(3) {
5        for(int t=0; t<num_steps; ++t) {
6           #pragma omp master {
7              #pragma omp task {
8                 #pragma omp target map(to: A[0:n]) \
9                    map(to: B[0:n], C[0:n]) {
10                   mic_computations(A, B, C, n);
11                }
12                #pragma omp target update from(A[0:n])
13             }
14             #pragma omp task {
15                host_activity();
16             }
17             #pragma omp taskwait
18          }
19       }
20    }
21 }
```

Listing 2. Offloading computations and data in OpenMP 4.0.

```
1  #pragma omp target enter data map(to: B[0:n], C[0:n]) \
2     map(alloc: A[0:n])
3
4  for(int t=0; t<num_steps; ++t) {
5     #pragma omp target map(to: A[0:n], B[0:n]) \
6        map(to: C[0:n]) nowait depend(out: x)
7     {
8        mic_computations(A, B, C, n);
9     }
10    #pragma omp target update from(A[0:n]) nowait \
11       depend(inout: x)
12    host_activity();
13    #pragma omp taskwait
14 }
```

Listing 3. Offloading computations and data in OpenMP 4.5.

(Line 1) is responsible for initializing the output array A, as well as transferring the input arrays B and C to the accelerator. Similar to Listing 2, **from** clause of **target update** construction (Line 10) is used to transfer the output data to the host. The code utilizes **nowait** clause (Lines 6 and 10) for ensuring the asynchronous execution of the offloaded region and data transfer, while **depend** clause of **target** and **target update** constructions (Lines 5 and 11) is required to define the order of their execution in the asynchronous mode. After initializing both data transfers and computations in the accelerator, the host starts executing its activity (Line 11). The activities of CPUs and MICs are synchronized using **taskwait** (Line 12). The presented code is quite similar to that from Listing 1, where Intel Offload is used. In both cases, the way of achieving the asynchronous management of devices is based on additional clauses of offloading constructions. Comparing both versions of the OpenMP Accelerator Model, one can see that the asynchronous execution provided by OpenMP 4.5 simplifies considerably the application codes for hybrid platforms.

3.4. Hetero Streams Library

The Hetero Streams Library (hStreams in short) [13] is a library-based extension for stream programming in heterogeneous environments. This extension implements [34] the heterogeneous asynchronous multitasking model, assuming the existence of one or more FIFOs abstractions, where jobs (tasks) are submitted for the execution on computing devices [46]. The hStreams framework focuses on ensuring portability in heterogeneous environments. It is dedicated to platforms consisting of MIC coprocessors and Intel Xeon CPUs. This library provides an abstraction of physical resources by applying the notions of domains, streams and memory buffers.

Before proceeding further, it is necessary to introduce two key terms used in the hStreams programming model: **source** and **sink** [19,46]. The first one refers to the place where a job is enqueued to be executed, while the second one corresponds to the place where a job is actually executed. This approach ensures a separation of concerns between management of tasks and control how they are mapped and executed by computing resources. In typical scenarios, the source is a process running on the host CPU, while the sink can reside on the same CPU, as well as on coprocessors. The source and sink can share resources of the same CPU or be placed on separate devices. An example of placement of source and sink is shown in Fig. 2.

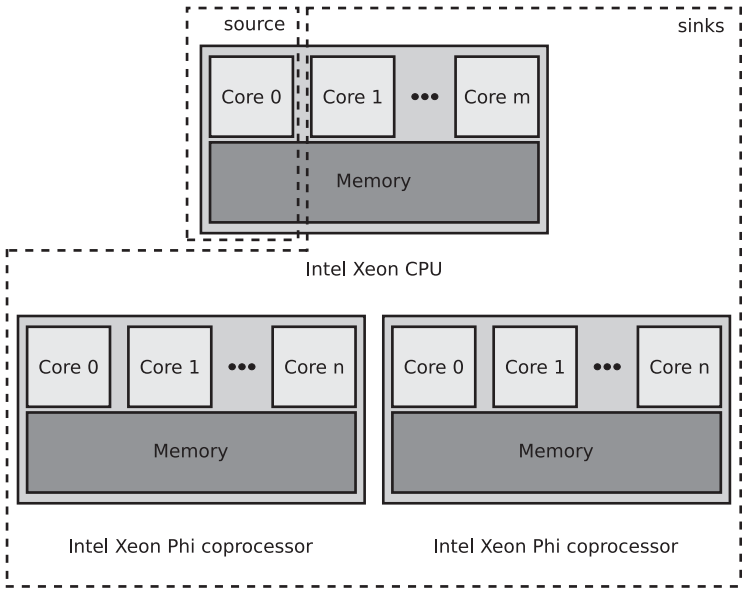

Fig. 2. Source and sink for platform with host CPU and two MICs.

The hStreams library distinguishes two types of domains: physical and logical ones. A **physical domain** represents computing resources which share one coherent memory domain. An example of these domains is a host processor, MIC coprocessor, or single node of cluster. The hStreams framework treats all the components of a platform uniformly. Physical domains are detectable and enumerable by the interface of hStreams. Each domain has a private set of properties, such as: number of threads, core frequency, ISA type, and size of each supported memory type. A **logical domain** is an abstraction of a physical domain [20], and one or more logical domains can be mapped to a certain physical domain (see Fig. 3). A given logical domain is specified by a mask which defines a subset of computing resources of a physical domain.

A **stream** is a primary building block of the hStreams execution model. It is a FIFO queue placed into a given logical domain. Streams have two endpoints: 1) source where the actions are inserted into the stream, and 2) sink where the actions are executed. All actions that may be placed into streams are grouped into three categories: **compute task, data transfers**, and **synchronization**. Compute tasks can naturally employ all the available threads of a stream by taking advantage of parallel programming standards like OpenMP. The hStreams library permits creating multiple streams within the same logical domain. An example of configuration of domains and streams is presented in Fig. 3.

The memory resources shared between source and sinks are called **logical buffers**. The logical buffers are registered by the application on the source. Once the hStreams run-time is aware of a buffer, a pointer to a memory location anywhere inside that buffer is recognized as a handle, and can be used for performing data transfers or compute actions involving that buffer. A logical buffer created by the user may have instantiations in many logical domains beside the source. These instantiations of the logical buffer are called **physical buffers**.

Internally, the hStreams framework has the implicit dependency management. By default, a task enqueued in a stream depends on the previous task in this stream and on all the buffers used by this task, but does not depend on memory transfers of buffers not related to the previous task. The dependencies can also be controlled explicitly by the application. For example, the application can wait for completion of one or more previously defined events. Such a dependency management allows programmers to hide communication behind computation.

The hStreams library provides two levels of API – the higher level **app API** and the lower level **core API**. The first one is dedicated

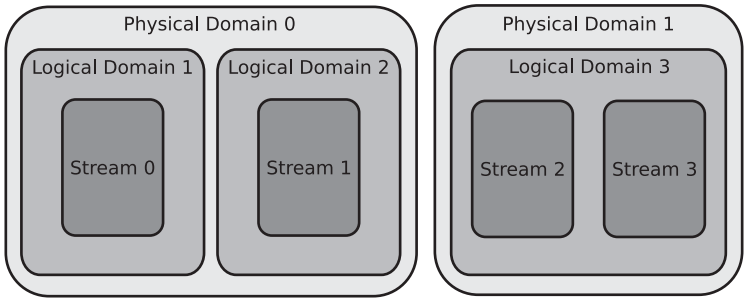

Fig. 3. An example of configuration of domains and streams.

to novice users, and offers only a part of hStreams functionality. Its productivity is boosted by helper functions and common building blocks. The core API is addressed for more advanced programmers, and offers the full functionality of hStreams. Currently, the library provides support for Intel Xeon CPUs, and the first generation of Intel Xeon Phi accelerators.

4. Numerical modeling of solidification

4.1. Numerical model

In the numerical modeling problem studied in the paper, a binary alloy of Ni-Cu is considered as a system of the ideal metal mixture in liquid and solid phases. The numerical model [50] refers to the dendritic solidification process in the isothermal conditions with constant diffusivity coefficients for both phases. It allows us to use the field-phase method defined by Warren and Boettinger [50]. In the model, the growth of microstructure during the solidification is determined by solving a system of two PDEs. The first equation defines the phase content ϕ:

$$\frac{1}{M_\phi}\frac{\partial \phi}{\partial t} = \varepsilon^2\left[\nabla\cdot(\eta^2\nabla\phi) + \eta\eta'\left(sin(2\theta)\left(\frac{\partial^2\phi}{\partial y^2} - \frac{\partial^2\phi}{\partial x^2}\right) + 2cos(2\theta)\frac{\partial^2\phi}{\partial x\partial y}\right)\right] - \frac{1}{2}\left(\eta'^2 + \eta\eta''\right)\left(-cos(2\theta)\left(\frac{\partial^2\phi}{\partial y^2} - \frac{\partial^2\phi}{\partial x^2}\right) + 2sin(2\theta)\frac{\partial^2\phi}{\partial x\partial y} - \frac{\partial^2\phi}{\partial x^2} - \frac{\partial^2\phi}{\partial y^2}\right) - cH_B - (1-c)H_A - cor\,, \tag{1}$$

where: M_ϕ is defined as the solid/liquid interface mobility, ε is a parameter related to the interface width, η is the anisotropy factor, H_A and H_B denotes the free energy of both components, *cor* is the stochastic factor which models thermodynamic fluctuations near the dendrite tip. The coefficient θ is calculated as follows:

$$\theta = \frac{\partial\phi}{\partial y}/\frac{\partial\phi}{\partial x}. \tag{2}$$

The second equation defines the concentration c of the alloy dopant, which is one of the components of the alloy:

$$\frac{\partial c}{\partial t} = \nabla\cdot D_c\left[\nabla c + \frac{V_m}{R}c(1-c)(H_B(\phi, T) - H_A(\phi, T))\nabla\phi\right],$$

where: D_c is the diffusion coefficient, V_m is the specific volume, R is the gas constant.

In this model, the generalized finite difference method [2,22] is used to obtain the values of partial derivatives in Eqs. (1) and (2). In order to parallelize computations with a desired accuracy, the explicit scheme is applied with a small value of the time step $\Delta t = 1e-7$ s. The resulting computations [44] belong to the group of forward-in-time, iterative algorithms since all the calculations performed in the current time step $t+1$ depend on results determined in the previous step t. The application code consists of two main blocks of computations, which are responsible for determining either the phase content ϕ or the dopant concentration c. In the model, the values of ϕ and c are determined for nodes distributed across a considered domain (Fig. 4). For this aim, the values of derivatives in all the nodes have to be calculated at every time step. In our previous work [44], two different cases were introduced – with either the static or dynamic intensity of computations. In the first case, the workload of CPUs and coprocessors is constant during

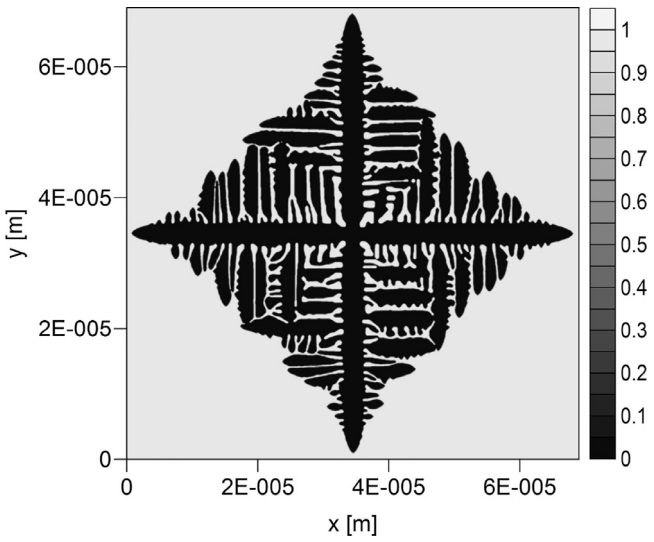

Fig. 4. Phase content for the simulated time $t = 2.75 \times 10^{-3}$ s (original code).

```
#pragma omp parallel for
for(int i=0; i<Grid->size; i++)
{
  double d1xFj = 0.0;
  double zx[Grid->node[i].neighbor_count];
  /.../
  for(int j=0; j<Grid->node[i].neighbor_count; j++)
  {
    // Stencil computations used for determination of partial derivatives
    zx[j] = 1/pow(Grid->node[i].h[j],2*m)*
            (Grid->node[i].g[0][0]*Grid->node[i].hx[j]+
            Grid->node[i].g[0][1]*Grid->node[i].hy[j]+
            0.5*Grid->node[i].g[0][2]*Grid->node[i].hx[j]*Grid->node[i].hx[j]+
            0.5*Grid->node[i].g[0][3]*Grid->node[i].hy[j]*Grid->node[i].hy[j]+
            Grid->node[i].g[0][4]*Grid->node[i].hx[j]*Grid->node[i].hy[j]);
    d1xFj+=zx[j]*Grid->node[Grid->node[i].e[j]].Fi0;
    /.../
  }
  // Computations within nodes of the grid
}
```

Fig. 5. Snippet of the original OpenMP code corresponding to one of 20 stencils.

the application execution, since a constant number of equations is solved. This assumption corresponds to modeling problems in which the variability of solidification phenomena in the whole domain has to be considered. In the second case, the model is able to solve differential equations only in part of nodes. The use of a suitable selection criterion allows reducing significantly the amount of computations. The consequence is a significant workload imbalance between CPU and MIC devices, since the selection criterion is calculated after the partitioning of nodes between devices. In our future work, we plan to develop an efficient algorithm for the dynamic load balancing in this case.

4.2. Original code of the solidification application

Fig. 5 illustrates the computational core of the original CPU code implementing the numerical model of solidification, for a single time step. The original code corresponds to the generalized FDM, and allows the partial differential equations to be solved not only for regular, but also irregular grids. All the computations are organized based on one-dimensional array containing elements of the grid. Data related to a certain element are encapsulated in the object `Grid->node[i]`. This organization of data enforces the memory management according to the array of structures (AoS) layout.

The computational core of the application represents two loops, where the outer and inner loops iterate over nodes of the grid and neighbors of each node, respectively. The inner loop relates to stencil computations used for determination of partial derivatives. For a given node `Grid->node[i]`, the indices `Grid->node[i].e[j]` of its neighbours are stored in a configuration file describing the whole grid. As a result, the patterns of all 20 stencils are determined at runtime. The structure of the application core allows its parallelization using **omp parallel for** directive of OpenMP for the outer loop.

For the studied version of the application, a 2D regular grid is used with nodes distributed uniformly across a square domain. To provide a required accuracy, 2000 nodes along each dimension are chosen as perfectly sufficient. In practice, computations are interleaved with writing partial results to the file. In the original version of the code, parallel computations are performed on CPUs for a sequence of time steps, when partial results are written to the file after the first, and then after every package of a selected number R of time steps ($R = 2000$ was specified in the studied case). This scheme (Fig. 6) permits us to observe and evaluate the grain growth during the simulation.

5. Approach to adapting the solidification application to hybrid CPU–MIC platforms

This section outlines an approach to adapting the solidification application to hybrid CPU–MIC platforms that can consist of more

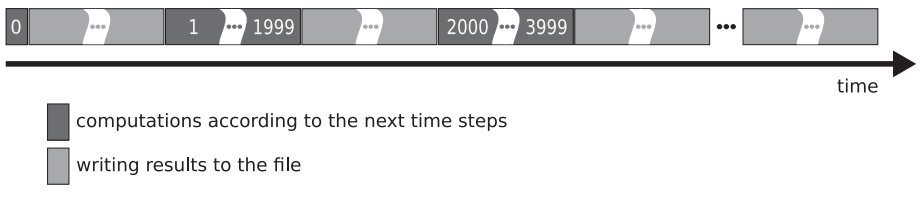

Fig. 6. Original version of the solidification application [44].

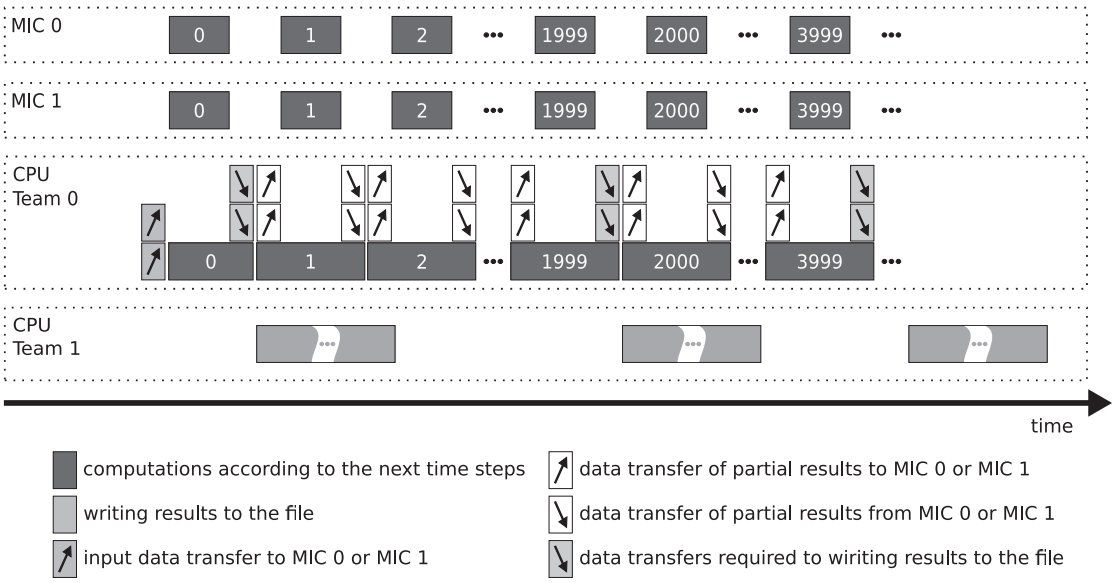

Fig. 7. Idea of adaptation of the solidification application to hybrid CPU–MIC platforms [44].

than one coprocessor. The goal is a significant reduction of the excessive execution time achieved with the original code (about 10 or 5 h for respectively static and dynamic intensity, as shown in Section 7). We propose to use both MIC and CPU devices for executing parallel workloads of the computational core of the application, while the rest of application is executed using CPUs only, including writing outcomes to the file. Such a solution allows the efficient utilization of resources, but requires flexible methods for workload distribution between CPUs and coprocessors. Another important assumption is avoiding significant modifications of the code. In particular, since OpenMP is used in the original code for CPUs, the same environment is applied in our development to parallelize computations on CPUs and coprocessors. Also, we do not provide any improvements for writing data to the file.

The idea [44] of adapting the solidification application to a CPU–MIC platform with two coprocessors is presented in Fig. 7. While both coprocessors and CPUs are responsible for executing the computational core of the application, writing data to the file is responsibility of CPU only. A critical performance challenge is to overlap all the computations with writing results to the file. Since a CPU performs both computations and writing data to the file, CPU threads are partitioned into two work teams. More precisely, the second team, containing a single thread, is responsible for writing results to the file, while the first team is utilized for the execution of calculations on CPUs, and management of computations for coprocessors. This solution allows us to simultaneously perform parallel computations using all the available devices, and writing data to the file.

In consequence, at the beginning of computations, the input data are transferred to coprocessors, which afterwards start computations for the first time step, together with the first CPU work team. After finishing the workloads assigned to both coprocessors, the results are returned to the main memory. During these transfers, the first CPU work team finishes computations for the first time step. Then the first CPU work team and coprocessors start computations for the next time step. Simultaneously, the second CPU work team starts writing results from the main memory to the file. In general, writing results obtained after finishing package of R time steps is overlapped with computations performed for the next package.

The next subsections present a sequence of steps required for adaptation of the studied application to CPU–MIC hybrid platforms, namely:

1. transformation of data layout;
2. partitioning of computations and optimizations of data movements;
3. load balancing of workloads between devices;
4. parallelization of computations across threads;
5. vectorization.

This sequence allows us to minimize the amount of data transfers, as well as to overlap the following operations: 1) computation, 2) inter-device communication, and 3) writing outcomes to the file. Also, it becomes possible to achieve the flexible load balancing of workloads between devices, and to optimize the utilization of resources of devices.

An issue of primary importance is also providing the numerical accuracy of computations, taking into account differences in architectures of Intel coprocessors and Intel Xeon CPUs. Following our previous paper [44], the accuracy of simulation results is evaluated experimentally for all the versions of code, considering both the phase content ϕ and concentration c. As in the case of work [44], this evaluation shows negligible differences of results between the original CPU code and each of the developed hybrid codes. What is important these differences are not cumulative, and they do not grow during the simulation.

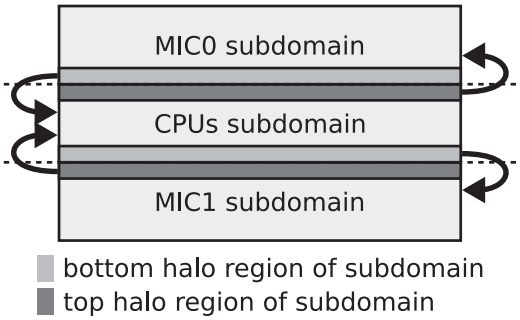

Fig. 8. Scheme of communication between CPUs and two KNC coprocessors.

5.1. Transformation of data layout

The memory organization is very important for optimization of the overall performance. The basic solution is to ensure the linear access to data. In practice, there are two major choices for the memory arrangement: Array of Structures (AoS) or Structures of Arrays (SoA). The original version of the application uses the AoS option. To provide a contiguous access to memory, data structures are transformed to the SoA layout, at the initialization phase. This conversion also permits us to avoid copying some unnecessary data.

5.2. Partitioning of computations and optimizations of data movements

A computing platform (see Fig. 1) is viewed as consisting of three components: 1) CPUs with ccNUMA architecture, 2) the first coprocessor MIC0, and 3) the second coprocessor MIC1. In consequence, all the computations in the original code will be partitioned along the index i of the outer loop (see Fig. 5) into three parts that correspond to grid subdomains, in a proportion that takes into account performances of components (see the next subsection). Due to data dependencies between grid nodes, some data have to be exchanged between platform components after every time step. Since coprocessors are utilized in the offload mode, all the exchanges are carried out through the PCIe bus and host memory.

To optimize the performance of data exchanges, we transfer (Fig. 8) only data that correspond to halo regions of grid subdomains [53], with asynchronous data transfers used to overlap data movements with computations. Moreover, to reduce overheads of data transfers between components, it was proposed [44] to assign the first subdomain to MIC0, the second subdomain – to CPUs, and the last one – to MIC1. This assignment reduces communication costs in comparison with the assignment of the adjacent subdomains to two MICs. Finally, to decrease significantly the performance overheads due to memory allocations, data within the coprocessor memory are allocated only once at the beginning of computations, and then reused many times.

5.3. Load balancing of workloads for CPU and MIC devices

In the case of the static computation intensity, the optimal load balancing between the three subdomains defined in the previous subsection can be determined in an empirical way, for a fixed problem size. Initially, computations are partitioned uniformly along rows of the grid. Based on measurements of the execution time for each part, a redistribution of computations is then performed. The redistribution is finished when the execution times for all the three subdomains (MIC0, CPU, MIC1) are the same, with a given accuracy. In consequence, the partitioning determined in this way provides the required load balancing for the successive execution of the application.

5.4. Parallelization of computations across threads

Like the original version, the basic work-sharing directive **omp parallel for** is used in a new code to assign computations to available threads, for both CPUs and coprocessors. To ensure the best overall performance, different setups for the scheduling clause of this directive have to be evaluated for different devices, including **static, dynamic**, and **guided**, with different sizes of chunks. Regarding CPUs and MICs, the important difference between them is that while all cores/threads of MICs are involved in computations only, at least one CPU core is loaded by the management of coprocessors as well.

5.5. Vectorization

An important step for performance optimization is utilization of capabilities of vector processing units available in CPUs and coprocessors. The most convenient way to reach this goal is the compiler-based automatic vectorization. The automatic vectorization is provided by the Intel compiler that automatically uses SIMD instructions available in the Intel Streaming SIMD Extensions such as AVX [37,49]. Usually the Intel compiler generates packed SIMD instructions through unrolling the innermost loop.

In the studied case, however, the compiler cannot vectorize the innermost loops safely because of features of calculations and data dependencies. In fact, the computations performed by the innermost loops require input data with dynamically determined indices. In

Table 1
Comparison of different versions of the application code.

Version of code	Management of coprocessors	Separation of writing results and computations	Synchronization		Parallel computations	
			Devices	CPU teams	CPU	MIC
A	Intel Offload	OpenMP nested parallelism with teams	Offload mechanisms	Custom	OpenMP parallel for $(n-1)$ cores	OpenMP parallel for
B	OpenMP 4.0 AM	OpenMP task parallelism	OpenMP taskgroup	OpenMP taskgroup	OpenMP task parallelism $(n-3)$ cores	OpenMP parallel for
C	OpenMP 4.0 AM	OpenMP nested parallelism with teams	OpenMP mechanisms	Custom	OpenMP parallel for $(n-1)$ cores	OpenMP parallel for
D	hStreams	Streams of hStreams	hStreams active synchronization	hStreams active synchronization	OpenMP parallel for $(n-2)$ cores	OpenMP parallel for

this case, the automatic vectorization fails due to the irregular data access, unpredictable during the compilation. To resolve this problem, it was proposed [44,45] to change the source code slightly by adding temporary buffers responsible for loading the required data from the irregular memory regions. Also, appropriate keywords and directives should be included as compiler hints, in order to increase the auto-vectorization efficiency. The auto-vectorization is also assisted with providing appropriate data alignments for the vectorized data. This forces the compiler to create data structures in memory aligned to specific byte boundaries.

Remark 1. Two of these steps, namely, the transformation of data layout and vectorization can be successfully used to optimize the performance of the original OpenMP code executed on CPUs only. In order to overlap computations with writing data to the file, CPU threads have to be partitioned into two work teams as well.

6. Adaptation of the proposed approach to different heterogeneous programing environments

This section outlines evaluation of four programming environments used to implement the approach proposed in the previous section. The evaluated environments include: Intel Offload coupled with OpenMP, OpenMP 4.0 and OpenMP 4.5 Accelerator Models, and hStreams with OpenMP. They correspond to four different versions of code in Table 1. Among the considered aspects which differentiate these environments are:

1. Management of coprocessors;
2. Way of separation of computations and writing results to the file;
3. Mechanisms used for synchronization between: (a) CPU devices (CPU and MICs), (b) CPU work teams responsible for computations and writing results to the file;
4. Way of parallelizing the workloads executed within CPUs and coprocessors, including the number of CPU cores available for parallel computations.

While the first aspect corresponds to mechanisms which enable offloading data and computations to coprocessors, the next one refers to methods used for partitioning of CPU threads into work teams. The third aspect addresses mechanisms utilized for both the inter- and intra-device synchronization, while the last aspect corresponds to ways of using available cores/threads to perform computations in parallel.

6.1. Intel Offload with OpenMP

Mapping the application workloads shown in Fig. 7 onto the programming environment which combines Intel Offload and OpenMP is rather straightforward. This version takes advantages of the asynchronous execution model of Intel Offload to ensure overlapping parallel workloads executed by CPUs with data transfers and computations performed by coprocessors. In practice, **offload target** directive with **signal** clause is used to initialize the asynchronous execution of these activities. Moreover, the management of coprocessors is assigned to the master thread of that CPU work team which is responsible for parallel computations. Since one CPU core is engaged in writing outcomes to the file, $(n-1)$ CPU cores are finally available for computations, where n is the total number of CPU cores.

To separate parallel computations and writing results to the file, all CPU threads/cores are partitioned into two work teams. For this aim, a custom solution is implemented based on the nested parallelism provided by OpenMP (Listing 4). This allows us to create two threads initially (line 5), and then the first of them spawns threads assigned to parallel computations (line 9). In order to avoid assignment of two or more threads to the same core, the required thread affinity is adjusted manually (lines 11–16), using **cpu_set_t** data structure to define the bit mask of CPU cores.

The nested parallelism allows increasing the overall performance by hiding writing outcomes to the file behind parallel computations. In addition, an appropriate synchronization of work teams is required to ensure the correctness of computations. Since

```
1  int teamsNO = 2;
2  int teams[teamsNO] = {totalThreads-1, 1};
3  int thread_offset = {0, totalThreads-1}
4  #pragma omp parallel num_threads(teamsNO) {
5    int idT = omp_get_thread_num();
6    omp_set_nested(1);
7    #pragma omp parallel num_threads(teams[idT]) {
8       const int ompID = omp_get_thread_num();
9       const int threadID = ompID + thread_offset[teamID];
10      cpu_set_t mask;
11      CPU_ZERO(&mask);
12      CPU_SET(threadID, &mask);
13      sched_setaffinity(0, sizeof(mask), &mask);
14   }
15 }
```

Listing 4. Partitioning CPUs threads into two work teams.

OpenMP does not provide the synchronization of nested regions, a custom synchronization mechanism based on **critical** directive of OpenMP is used to assure the team synchronization, while the synchronization of devices is implemented through the mechanism offered by Intel Offload.

To parallelize computations performed by CPUs and coprocessors, the work-sharing construction **omp parallel for** is used. Since the master thread of the first CPU team is utilized for both computations and management of coprocessors, an appropriate scheduling of loop iterations is vital to optimize the overall performance. The static scheduling generates performance overheads since the rest of threads in this team have to wait for the master. To resolve this problem, we use the dynamic scheduling of loop iterations. As a result, after offloading data and computations to coprocessors, the master thread joins the rest of threads performing computations. This solution enables hiding the management of coprocessors behind parallel computations performed by CPUs. At the same time, it was evaluated experimentally that the static scheduling is the most efficient way for splitting loop iterations among threads in coprocessors. Such a mixture of static and dynamic scheduling is also chosen for other programming models evaluated in this paper.

6.2. OpenMP 4.0 Accelerator Model

In this case, we assume that only mechanisms offered by the OpenMP Accelerator Model are used to implement the application workloads shown in Fig. 7. As compared to the previous solutions, this version differs in all the aspects, except for the way of parallelizing computations in coprocessors. The idea of mapping the application workloads onto the OpenMP 4.0 Accelerator Model is illustrated in Fig. 9.

To offload data and computations to coprocessors, this version uses a mixture of three basic constructions: **target, target update**, and **target data**. While the first two ones are responsible for oflloading computations and data to coprocessors, the last one is used to keep data in the coprocessor memory between executing the offloaded regions. Since version 4.0 of OpenMP does not provide mechanisms for the asynchronous execution of **target** and **target update** directives, we propose to use the OpenMP task parallelism for the asynchronous management of coprocessors. The task parallelism is also utilized to perform CPU computations and write results to the file simultaneously.

Following the scheme shown in Fig. 9, there are n tasks spawned on different CPU cores. One of these tasks is engaged in writing results to the file, while the next two tasks are responsible for offloading computations and data to coprocessors. The rest of tasks correspond to computations performed by CPUs.

Version 4.0 of OpenMP does not provide a solution for the automatic partitioning of loop iterations among tasks. Thus, in the case of CPUs, we have to split loops manually into equal blocks of the size determined by the total amount of iterations divided by the number of tasks engaged in computations. At the same time, utilizing **omp parallel for** directive with the static scheduling is enough for parallelizing computations in coprocessors, where is no need to apply the task parallelism.

The usage of task parallelism to implement the OpenMP 4.0 Accelerator Model is coupled with providing an adequate task synchronization. For this aim, **omp taskgroup** construct is applied, in order to provide two scenarios of synchronization. The first scenario is used to synchronize CPUs and coprocessors after every time step. The second one is required after every package of R time steps, and is responsible for synchronizing computations and writing results to the file. The idea of task synchronization is illustrated in Listing 5.

6.3. OpenMP 4.5 Accelerator Model

The asynchronous execution is the key feature of the OpenMP 4.5 Accelerator Model as compared with OpenMP 4.0. This feature allows us to implement the solidification application quite similar to the version based on Intel Offload. In practice, **target** and **target update** directives with **nowait** clause are used to ensuring the asynchronous execution of computations and data exchanges. The difference against Intel Offload is the scenario of management of coprocessors. Instead of initializing the asynchronous execution of an offloaded region during CPU computations, now the asynchronous activity is initialized first, and only then CPU threads start the workload execution. Some performance overheads are disadvantage of such a scenario.

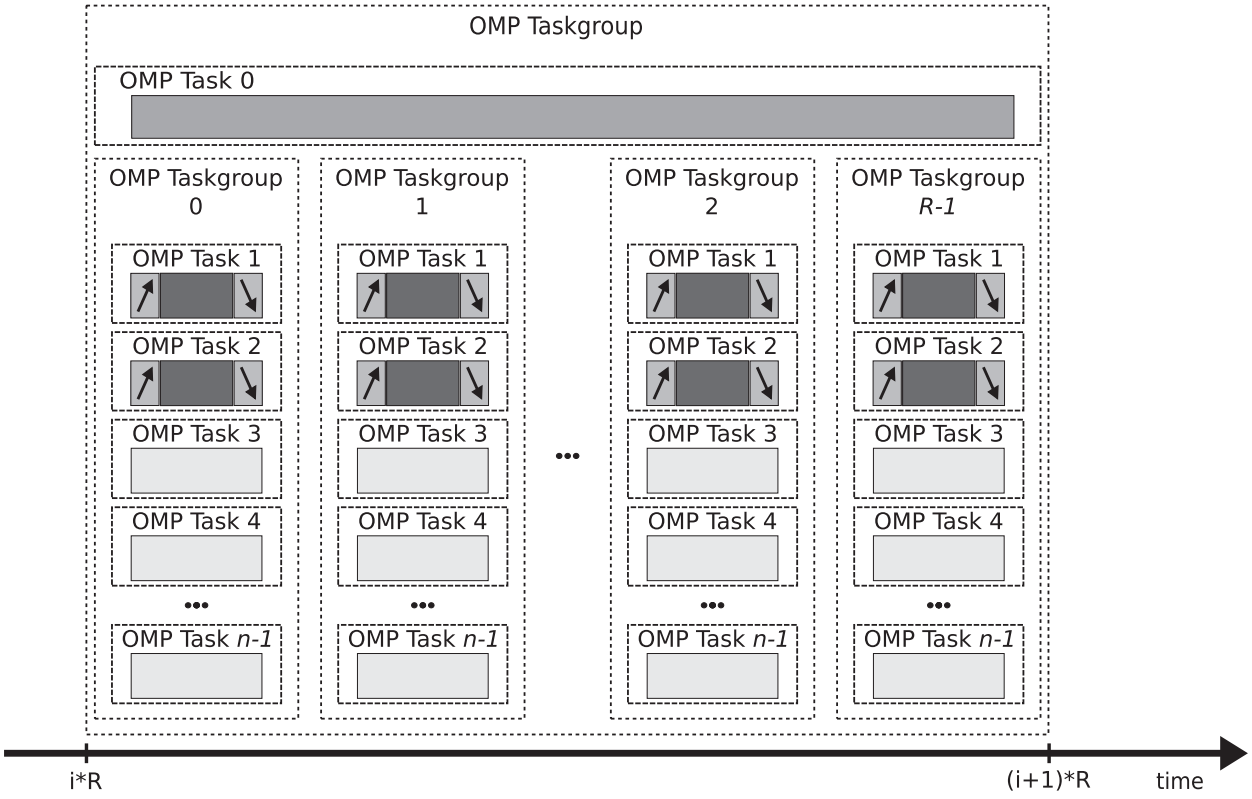

Fig. 9. Mapping the application workloads onto OpenMP 4.0 Accelerator Model.

```
1  #pragma omp taskgroup {
2     #pragma omp task {
3        // Task of writing results
4     }
5     for(int ts=0; ts<R; ++ts) {
6        #pragma omp taskgroup {
7           // Computational tasks
8        }
9     }
10 }
```

Listing 5. Scheme of task synchronization.

The similarities between these two versions concern also the way how to separate parallel computations and writing results to the file. As before, CPU threads are partitioned into two work teams using the OpenMP nested parallelism. Also, the synchronization of work teams is performed in the same way, based on **critical** directive.

6.4. Hetero Streams Library with OpenMP

The hStreams library supports the asynchronous task parallelism on hybrid platforms by providing multiple streams that are mapped onto different computing domains (devices). This advantage is successfully used for adapting the solidification application to hybrid platform with more than one KNC coprocessor. Thus, it is natural to use hStreams for the flexible management of computing devices, while the usage of OpenMP is limited only to harnessing the resources of cores within devices.

The idea of mapping the application onto heterogeneous streams is shown in Fig. 10. Four logical streams are created within three logical domains. Here Stream_4 is used for writing results to the file, and Stream_3 is utlized for parallel computations on CPUs. The remaining streams are responsible for parallel computations and data exchanges performed by coprocessors. The streams are assigned

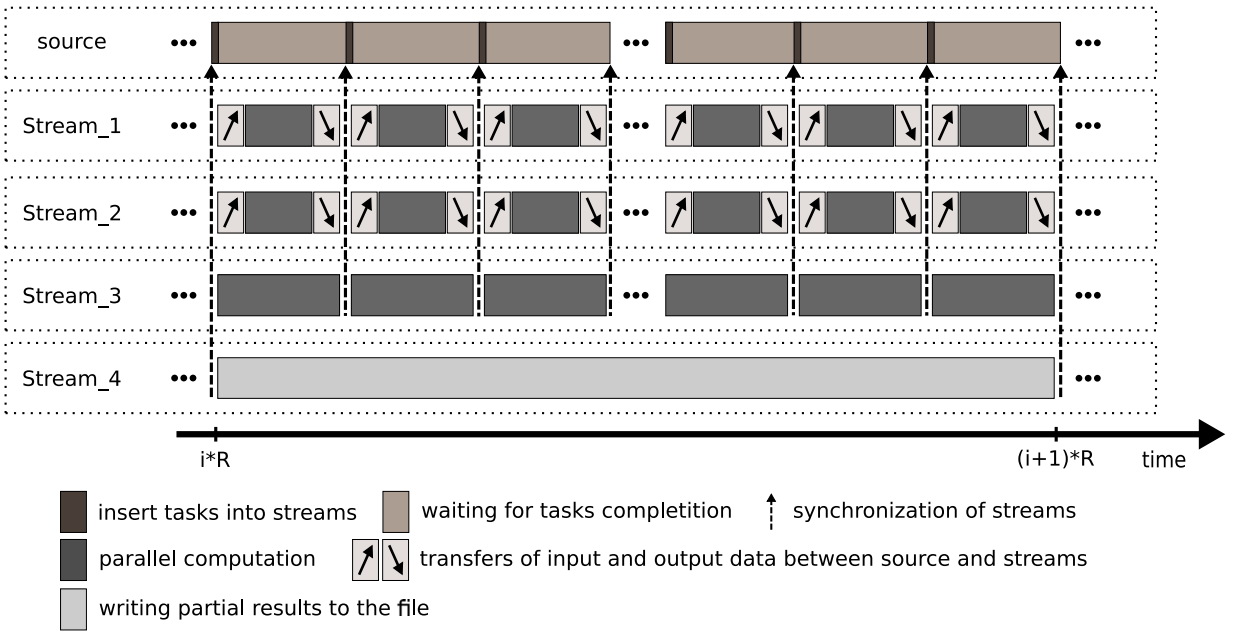

Fig. 10. Mapping the application workload onto heterogeneous streams.

to the devices as follows: Stream_1 and Stream_2 are executed by coprocessors, while Stream_3 and Stream_4 are assigned to CPUs. The creation of two streams on the CPU side allows us to overlap computations with writing results to the file. These two streams are created using Core API.

To parallelize computations within the streams, we use **omp parallel for** directive. As two CPU cores are dedicated to running the source process of hStreams and writing data to the file, $(n-2)$ CPU cores are used for computations directly.

The studied application enforces some inter-stream dependencies. Therefore, providing the required synchronization of streams in an efficient way is vital for the correctness and performance of the application. The hStreams library supplies two ways to achieve this goal. They correspond to various methods of filling the queues. The first way assumes that the FIFO queue is filled before the execution of a stream, while in the second case the FIFO queue is filled during the stream execution. For both ways, the source is responsible for managing the streams. Specifically, in the first case, the synchronization of streams is based on completion the events that are placed into streams before the task execution. For the second way, called the active synchronization, the source puts the tasks into streams, then waits for their completion. In our case, the second way is chosen as more efficient to provide the synchronization of computations after every time step. Additionally, all the streams have to be synchronized after completing a package of R time steps, in order to overlap computations with writing outcomes to the file correctly.

The efficient management of memory buffers is another important issue when mapping the application onto heterogeneous streams. By default, the memory is allocated within all the existing logical domains. Although in many cases logical domains are mapped to the same physical domain, each of them is equipped with own set of logical buffers. The undesirable consequence are explicit data transfers between them. To resolve this issue, we use the mechanism known as aliased buffers, provided by Core API. It guarantees that logical domains which belong to one physical domain share memory resources. As a result, all these domains will use the same logical buffers. This allows optimizing the data exchange, especially for applications which includes more logical domains. An example of using this technique is shown in Listing 6. Here `HSTR_BUFFER_PROPS` is a data structure that allows defining the properties of buffers, while `hStreams_Alloc1DEx` function is responsible for allocating memory. In consequence, the buffers A and B are shared among all the logical domains within the same physical domain.

We propose to use aliased buffers on the CPU side. By default, during computations for a package of R time steps it is necessary to exchange data between the source and the streams used for parallel computations. In every time step, data are transferred from the source to streams at the beginning of computations, and back to the source after their completion. Then after finishing the execution

```
1  HSTR_BUFFER_PROPS props;
2  props.flags = HSTR_BUF_PROP_ALIASED;
3  props.mem_type = HSTR_MEM_TYPE_NORMAL;
4  props.mem_alloc_policy = HSTR_MEM_ALLOC_PREFERRED;
5  hStreams_Alloc1DEx((void*) A, n*sizeof(float),
6     &buff_props, -1, NULL));
7  hStreams_Alloc1DEx((void*) B, n*sizeof(float),
8     &buff_props, -1, NULL));
```

Listing 6. Initialization of aliased buffers.

Table 2
Specification of the tested platform with KNC coprocessors [29].

Device type	CPUs	KNCs
Number of devices	2	2
Name of each device	Intel Xeon E5-2699 v3	Intel Xeon Phi 7120P
Number of cores	18	61
Number of threads	36	244
SIMD width [bits]	256	512
Base freq. [GHz]	2.3	1.238
AVX freq. [GHz]	1.9	1.238
AVX peak [Gflop/s]	547.2	1208.3
Scalar peak [Gflop/s]	331.2	151.0
LLC* size [MB]	45	30.5
Memory size	256GB DDR4	16GB GDDR5
Memory bandwidth [GB/s]		

* LLC (Last Level Cache) refers to aggregated L2 caches for Intel Xeon Phi and L3 cache for CPU.

of every package of R steps, data are transferred from the source to Stream_4. Although Stream_3, Stream_4 and the source are placed in the same physical domain, data movements generate significant performance overheads. Utilizing aliased buffers guarantees that all the streams of CPUs and the source share the same memory regions. As a result, the total amount of data transfers is significantly reduced, since now data are transferred only between the source and the streams mapped to coprocessors.

7. Benchmarking heterogeneous programming environments on hybrid CPU–KNC platform

7.1. Performance results

In this subsection, we present the performance results obtained for the double precision floating-point format. The specifications of Intel Xeon CPUs (with Haswell microarchitecture) and KNC coprocessors used in our experiments are presented in Table 2. All the tests are compiled using the Intel icpc compiler (ver. 17.0.0) with the optimization flag -O3, and executed on Red Hat 3.8, and Intel MPSS 3.7.2. The benchmarks are performed for the 2D grid containing 4,000,000 nodes (2000 nodes along each dimension x and y) and 110,000 time steps. Table 3 presents values of the total execution time achieved for the four versions of code, assuming both the static and dynamic intensity of computation. To ensure the reliability of performance results, the measurements of the execution time are repeated $r = 10$ times, and the median values are used finally. The statistical properties of measurements are controlled by calculating the relative standard deviation (RSD) for each set of r measurements. Also, all the experiments have been repeated after some time to check the reproducibility of results. It is worth noting that the tested platform is located in the professional data center with stable environmental conditions [29].

The proposed general scheme of workload distribution (Fig. 7) enables the flexible usage of various configurations of devices to perform computations. In the tests, the following configurations of computing resources are studied:

1. single KNC coprocessor,
2. two KNC coprocessors,
3. two Intel Xeon CPUs and single coprocessor,
4. two Intel Xeon CPUs and two coprocessors.

In addition, a pure OpenMP code is executed on the homogeneous configuration with two CPUs, providing the reference performance. The hints described in Remark 1 from Section 5 have been used to optimize this code.

Table 3
Total execution time (in seconds) measured for different versions of the application, and various platform configurations, both for the static (S) and dynamic (D) intensity of computations. The statistical properties of measurements are given by RSD less than 2.4%, and the average value of RSD for various hardware configurations with KNC equal to 1.06%, 0.80%, 1.08%, and 1.40% (static intensity), as well as RSD less than 2.9%, and the average RSD for various hardware configuration with KNC equal to 2.10%, 1.04%, 0.87%, and 1.63% (dynamic intensity).

Code version	1 × KNC		2 × KNC		2 × CPU + 1 × KNC		2 × CPU + 2 × KNC		2 × CPU	
	S	D	S	D	S	D	S	D	S	D
A	11,131	5528	5609	2808	4718	2114	3409	1650	7764(RSD = 0.85%)	2922 (RSD = 1.35%)
B	11,439	5801	10,878	6429	12,258	6100	10,643	4908		
C	11,395	5798	5697	2904	4982	2325	3992	1744		
D	10,911	5201	5730	2810	4729	2203	3585	1791		

Remark: For the original code, the execution time on 2 × CPUs was 38,492 and 16,072 s for the static and dynamic intensity, respectively.

Table 4
Load balancing setups (in %) for different CPU–MIC configurations.

Code version	2 × CPU + 1 × KNC		2 × CPU + 2 × KNC	
	S	D	S	D
A	61.5 : 38.5	60 : 40	43.8 : 28.02 : 28.18	38 : 31 : 31
B	60 : 40	58 : 42	44.5 : 27.6 : 27.9	34 : 33 : 33
C	60 : 40	58 : 42	44.5 : 27.6 : 27.9	34 : 33 : 33
D	61.2 38.8	60 : 40	44.9 : 27.5 : 27.6	34 : 33 : 33

For hybrid CPU–KNC configurations, the presented values of the execution time are achieved for load balancing setups shown in Table 4. For example, the setup of 43.8 : 28.02 : 28.18 for the configuration with two CPUs and two MICs means that 43.8% of grid nodes are processed by CPUs, while respectively 28.02% and 28.18% of nodes are assigned to MIC0 and MIC1.

To optimize the performance of computations, we evaluate experimentally various setups for the scheduling clause of **omp parallel for** directive. As already mentioned, the best performance is achieved for the static scheduling on coprocessors (4 threads per core), and dynamic one on CPUs. In both cases, the optimal size of chunks is selected as 128.

For the static computation intensity, the analysis of Table 3 allows us to conclude that all the versions give quite similar performance results only for a single KNC. Already for two KNCs, the OpenMP 4.0 version yields a radically worse performance then others, providing only a small improvement in comparison with a single MIC. As resources expand and a true hybrid CPU–KNC platform is used, the total execution time of this version is even increased with each additional coprocessor. At the same time, the rest of versions are able to adapt to expansion of available resources, however, not with the same efficiency (see Table 5). In particular, for the version A, two KNCs allows us to accelerate the applications 1.39 times against two CPUs. Furthermore, the hybrid platform with 2 CPUs and a single KNC gives the speedup of 1.63x, while the most advanced configuration with two CPUs and two KNCs yields the speedup of 2.28x. For the versions C and D, the achieved speedups are respectively: 1.36x, 1.56x and 1.94x for the version C, and 1.35x, 1.64x and 2.16x for the version D. These values of the speedup S_{CPU} are calculated against the execution time obtained on two CPUs. At the same time, the last two column of Table 5 includes values of the speedup S_{KNC} achieved against the configuration with two coprocessors, for a given version of code. Thus for the versions A, C, and D of the application with the static computation intensity, the utilization of two CPUs together with two KNCs allows accelerating the application 1.65, 1.43, and 1.60 times, respectively, in comparison with only two KNCs.

As a rule of thumb, the achieved values of S_{CPU} and S_{KNC} can be explained based on the peak performance P of the considered configurations. For example, the speedup S_{CPU} of the most advanced hybrid configuration against two CPUs is less than 2.28, while the ratio $R_1 = P_{2\times KNC+2\times CPU}/P_{2\times CPU}$=(2*547.2 + 2 * 1208.3)/(2 * 547.2) is about 3.2. The difference is the result of such factors as low performance of KNC for non-vectorized computations, higer overheads for communication between coprocessors than between CPUs, etc. These factors are also the reason that the speedup $S_{KNC} = 1.65$ is higher in this case than the ratio $R_2 = P_{2\times KNC+2\times CPU}/P_{2\times KNC} = 1.45$.

Remark 2. The peak performance of Haswell CPUs given in Table 2 corresponds to the base frequency of AVX vector instructions. The tested platform utilize the Intel Turbo Boost Technology 2.0 [38], which automatically allows CPU cores to run faster than the base frequencies if they are operating below power, current and temperature limits. In fact, the AVX turbo frequency f^*_{AVX} of the Intel Xeon E5-2699v3 CPU can change in the range between 1.9GHz and 2.6GHz. As a result, the estimations of ratios R_1 and R_2 will also change. For example, assuming $f^*_{AVX} = 2.3$GHz, we obtain $R_1 = 2.82$ and $R_2 = 1.54$.

Going to the case of the dynamic computation intensity, it can be concluded that except for OpenMP 4.0 the rest of the studied versions are also able to adapt to expansion of resources. At the same time, by comparing values of the speedup S_{CPU} for the static and dynamic intensity, we conclude that the efficiency of using coprocessors in hybrid configurations is considerably lower in the second case. For example, when utilizing the configuration with two KNCs and two CPUs to execute the Intel Offload version for the dynamic intensity of computations, we obtain the acceleration of $S_{CPU} = 1.77x$, instead of $S_{CPU} = 2.28x$ in the case of the static intensity.

Remark 3. Below Table 3, we present the total execution times achieved by the original, non-optimized OpenMP code on two CPUs,

Table 5
Speedups achieved for different versions of the applications, where S_{CPU} and S_{KNC} refer to speedups achieved against configurations with two CPUS and two KNC coprocessors, respectively.

Version of code	2 × KNC		2 × CPU + 1 × KNC		2 × CPU + 2 × KNC			
	S_{CPU}		S_{CPU}		S_{CPU}		S_{KNC}	
	S	D	S	D	S	D	S	D
A	1.39	1.04	1.64	1.38	2.28	1.77	1.65	1.70
B	0.71	0.45	0.63	0.48	0.73	0.59	1.02	1.30
C	1.36	1.01	1.56	1.26	1.94	1.67	1.43	1.67
D	1.35	1.04	1.64	1.33	2.16	1.63	1.60	1.57

both for the static and dynamic intensity of computations. These values of about 10.5 and 4.5 h were the starting point of our development, that was originally aimed at accelerating the application by using KNC coprocessors [45]. The hidden bonus of this development was the possibility of using some of the proposed optimizations to speed up the original code when executed on CPUs only (see Remark 1). The result was decreasing the execution time on two CPUs to less than 2 h 10 min for the statics intensity, and less than 50 min for the dynamic intensity. Finally, using the most advanced hybrid configuration we are able to execute the application in less than respectively 57 and 28 min, that corresponds to the overall acceleration of about 11.3x and 9.7x, respectively.

7.2. Discussion: performance optimization issues

Affinization of cores/threads

To identify the reasons for the poor performance of the OpenMP 4.0 version, its execution has been analyzed in details. The conclusion is that the main reason are issues related to the affinization of threads on the CPU side.

In fact, the OpenMP standard does not provide a mechanism which enables us to define the affinity for tasks. As a result, OpenMP tasks are executed by different threads in each time step. In particular, this relates to tasks responsible for offloading computations and data to coprocessors (see Fig. 9); these tasks migrate between CPU threads in successive time steps. As a result, coprocessor threads are created in each time step, that generates prohibitive performance overheads. As creating a KNC thread takes a rather long time, the execution of a single time step by KNC takes up to 10 times longer than in the other three versions. For these versions, the affinization of CPU threads (see Listing 4) allows us to avoid the migration of threads responsible for offloading computations and data to coprocessors. In consequence, the performance overheads due to the creation of these threads are observed only in the first time step, and once created they are reused in the subsequent steps.

Only when parallel computations are performed by a single coprocessor, this performance bottleneck of OpenMP 4.0 is not observed. In this case, only two tasks are created on the CPU side: 1) offloading computations to the accelerator, for a package of R time steps, and 2) writing results to the file. These tasks are activated inside the parallel region of OpenMP by the master thread. As a result, each of the tasks is always executed by the same thread, for a package of R time steps.

Double buffering for CPUs

Another factor influencing the overall performance is organization of memory buffers for CPUs. The evaluated codes use two sets of data buffers on the CPU side. In this double buffering solution, first set is utilized for parallel computations performed by CPUs, while the second one is responsible for exchanging data with accelerators. At the beginning and end of each time step, data corresponding to halo regions of grid subdomains have to be copied between these sets. Such a solution enables us to fully overlap CPU computations with transferring data to/from coprocessors, as shown in Fig. 7. Using a single set of buffers enforces performing CPU computations and data exchanges sequentially. Due to this overhead, the performance of CPU computations is about 20% lower in this case.

7.3. Observations on productivity of software development

Scientists are finding it increasingly costly and time consuming to write, port, or rewrite their software to take advantage of new architectures [8]. While the machine performance remains the main driver for HPC applications, the cost (or productivity) of software development becomes a critical bottleneck preventing a wider adoption of the HPC technology. Effective parallelism is difficult to achieve in heterogeneous platforms. If these efforts were more reasonable, more people would tune their codes to achieve efficient performance [34].

One key to improve the productivity of programming in HPC environments is increasing the level of abstraction for the code development. This direction is especially significant for hybrid platforms because of their highly heterogeneous and complex nature. A streaming abstraction is one of several compelling choices for mapping task parallelism to heterogeneous platforms [34]. The hStreams library studied in this paper is based on this high-level abstraction. This framework is aimed at making it easier to port and tune task-parallel codes.

The experience gained in this study highlights separation of concerns as the main feature of hStreams which contributes to making porting and tuning a real-life application simpler. This feature allows a separation of concerns between 1) the expression of functional semantics and exposure of task parallelism, and 2) the hardware-aware performance tuning and control over mapping workloads onto a platform [34]. As a result, while creators of scientific algorithms receive an intuitive tool, code tuners may work long after them, having the freedom to control over the code details without the need for application domain expertise. For the studied application, such a separation is distinctly demonstrated in Fig. 10, which shows that the usage of the stream abstraction provides the high-level exposure of parallelism, where each of streams encapsulates a clear functional semantics. Furthermore, Table 1 shows another advantage of hStreams. Opposite to other versions, the hStreams framework delivers ready-to-use constructions for all the aspects of control over mapping the application workloads to CPU–KNC platforms: 1) management of coprocessors, 2) separation of computations and writing results to the file, as well as 3) inter- and 4) intra-device synchronizations.

From the practical point of view, the important merit of hStreams is the independence of the compiler version, since hStreams is a library-based API extension, while Intel Offload and OpenMP are compiler-supported language extensions. In the later case, the actual behavior of the application code during its compilation and execution may change considerably along with changing the compiler version. For this study, we observed such problems with Intel Offload when we had to move from version 15.0.2 of the Intel icpc compiler to version 17.0.0, in order to provide support for OpenMP 4.5.

The described advantages of the high-level hStreams framework in easier code development are achieved at low performance overheads in comparison with the low-level Intel Offload solution, which gives the shortest execution time for hybrid CPU–KNC configurations. Following Table 3, these overheads are less than 5% and 9% for the static and dynamic intensity of computations, respectively.

8. Benchmarking the solidification application on platform with KNL processors

8.1. Adapting the application to CPU–KNL platforms

The second generation of Intel Xeon Phi introduces many improvements over the first one [20,25]. The KNL processor is composed of up to 72 cores based on the Silvermont (Atom) microarchitecture. They are organized into tiles connected by the 2D mesh topology with improved on-package latency. Each tile consists of two cores (two vector processing units per core), and 1MB L2 cache shared between two cores in tile. The caches are connected to each other with a mesh, and are kept coherent with the MESIF protocol. KNL is equipped with the new vector instruction set – AVX-512. Each vector processing unit (VPU) operates independently on 512-bit vector registers, which support up to eight double-precision FMA operations. KNL cores utilize multithreading to allow running four threads per core, and are binary compatible with prior Intel CPUs.

KNL has two types of memory: on-package high-bandwidth memory (HBM) based on MCDRAM technology (16 GB), and large capacity DDR4 (up to 384GB). The HBM memory can be configure in one of three modes: cache (MCDRAM works as a cache for DDR4), flat (HBM is addressable memory in the same address space as DDR4), and hybrid [25].

The KNL architecture has a theoretical peak performance of about 3 TFLOP/s in double precision, which is about three times higher than what KNC provides. This performance gain is partly due to the presence of two VPUs per core, doubled compared to the previous generation. KNL devices are delivered as standalone processors designed for massively parallel workloads. Fig. 11 illustrates a hybrid CPU–KNL platform, where KNL accelerators are connected to CPU via the HPC fabric using either the proprietary Intel Omni-Path Architecture (OPA) interconnect [3] or Infiniband. Table 6 presents specifications of Intel Xeon CPUs (with Broadwell microarchitecture) and KNL processors used in our research. The suffix F in the Intel Xeon Phi model number (7250F) denotes the OPA fabric integrated with KNL devices.

The solidification modeling application can be adapted to platforms with KNL processors by using the Offload over Fabric (OoF) technology [16,30]. Among the programming environments studied in this paper, only Intel Offload and OpenMP Accelerator Model supports this technology at the moment. To our best knowledge, there are no implementation of hStreams for the OoF technology. For OpenMP 4.5 and Intel Offload, the adaptation of codes developed for the first generation of the MIC architecture to the second generation is almost straightforward by recompiling the source codes with the flags `-O3` and `-qoffload-arch=mic-avx512`. Furthermore, KNL accelerators are used in the quadrant clustering mode [20], with the MCDRAM memory configured in the flat mode.

However, after discovering high performance overheads for the scenario when CPUs offload computations to KNL processors and concurrently execute parallel workloads, we have decided to release CPUs from executing the computational core of the application,

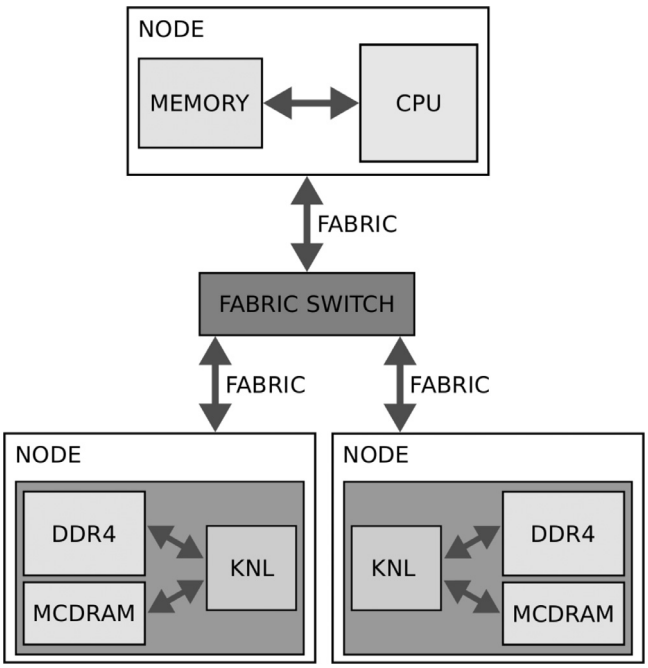

Fig. 11. Hybrid CPU–KNL platform.

Table 6
Specification of the tested platform with KNL processors.

Device type	CPUs	KNL accelerators
Number of devices	2	8
Name of each device	Intel Xeon E5-2697 v4	Intel Xeon Phi 7250F
Number of cores	18	68
Number of threads	36	272
SIMD width [bits]	256	512
Base freq. [GHz]	2.3	1.4
AVX freq. [GHz]	2.0	1.2
AVX peak Gflop/s	576	2611.2
Scalar peak Gflop/s	331.2	380.8
LLC* size MB	45	34
Memory size	128GB DDR4	16GB MCDRAM 96 GB DDR4
Memory bandwidth [GB/s]	76.8	MCDRAM: 400+ DDR4: 115.2

* LLC (Last Level Cache) refers to aggregated L2 caches for Intel Xeon Phi, and L3 cache for CPU.

and study the possibilities of accelerating the application by expanding the number of KNLs as much as possible. Due to properties of our testbed, CPU is responsible as before for writing partial results to the file.

The proposed general scheme of workload distribution (Fig. 6) allows flexible expanding the number of MIC devices. In the case of the first generation of the MIC architecture, the tested platform permits us to use no more then two KNCs. Now it is possible to utilize up to 8 KNL processors. At the same time, using more than two KNLs requires the exchange of halo regions corresponding to both borders of grid subdomains assigned to MIC devices (Fig. 12), while for the CPU–KNC platform it was enough to exchange halo regions for a single border. As shown in Fig. 12, KNL1 has to receive data corresponding to the bottom border of KNL0, and the top border of KNL2, etc. As before, these data are exchanged through the CPU memory. The communication is implemented using the asynchronous mechanisms of data transfers available in Intel Offload and OpenMP 4.5 Accelerator Model.

When adapting the application to CPU–KNL platforms, we manage MIC devices in the same way as in the case of KNC coprocessors. As before, a single CPU thread is responsible for transferring data and computations between CPU and MIC devices. This interaction is implemented using asynchronous mechanisms supported by both environments. A noticeable difference concerns the way of synchronizing the KNL processors during computations in successive time steps. For Intel Offload, each KNL has to be synchronized separately using **offload_wait** construction, while in the case of the OpenMP Accelerator Model it is enough to use a single **taskwait** construction to synchronize all the KNL devices.

8.2. Performance results

Table 7 presents the performance results achieved for two programming environments on the platform specified in Table 6, with KNL processors connected via the OPA fabric. We assume the same configurations of the application parameters as in Section 7.1. The versions with the static and dynamic intensity of computations were compiled using Intel icpc compiler (ver. 17.0.2). Besides the total execution time T_p obtained for a different number $p = 1, 2, ...,8$ of KNL devices, this table shows the speedup S_p and efficiency E_p defined as $S_p = T_1/T_p$ and $E_p = T_1/S_p$*100%, as well the speedup S_{CPU} calculated against the execution time T_S or T_D achieved on two CPUs for respectively the static and dynamic intensity. These values of speedups and efficiency are specified only for Intel Offload. Based on small (practically negligible) differences in performance between Intel Offload and OpenMP 4.5 (see last column of Table 7), the speedups and efficiency are shown only for the Intel Offload version, which is also the only version used to study the performance in the case of dynamic intensity. Fig. 13 complements Table 7 by giving a comparison of execution times achieved for the platforms with the first (green colour) and second (red colour) generations of MIC.

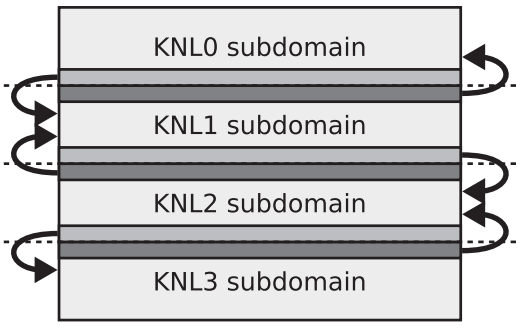

Fig. 12. Scheme of communication between KNL processors for configuration with four KNLs.

Table 7
Performance results for the platform with KNL processors achieved for Intel Offload and OpenMP 4.5 versions of the application. The statistical properties of measurements are given by RSD less than 1.56% (static intensity), and RSD less than 1.81% (dynamic intensity).

	Intel Offload with OpenMP								OMPAM 4.5	
	S				D				S	
	T_p [s]	S_{CPU}	S_p	E_p [%]	T_p [s]	S_{CPU}	S_p	E_p [%]	T_p [s]	ΔT [%]
1 × KNL	3960	2.16	—	—	1641	1.82	—	—	4070	2.77
2 × KNL	2200	3.88	1.80	90	1085	2.76	1.51	76	2266	3.00
3 × KNL	1614	5.29	2.45	82	1045	2.86	1.57	52	1628	0.87
4 × KNL	1271	6.72	3.12	78	866	3.45	1.89	47	1278	0.55
5 × KNL	1109	7.70	3.57	71	844	3.54	1.94	39	1100	-0.81
6 × KNL	1031	8.28	3.84	64	740	4.04	2.21	37	1042	1.06
7 × KNL	953	8.96	4.16	59	729	4.10	2.25	32	979	2.72
8 × KNL	930	9.18	4.25	53	710	4.21	2.31	29	957	2.90

Remark: The execution time on 2 CPUs is $T_S = 8539$ s (RSD = 1.14%) and $T_D = 2991$ s (RSD = 2.28%) for the static and dynamic intensity, respectively.

In the benchmarks, the best performance is achieved with 4 threads per each core of KNL. In addition, for the static computation intensity, we select the static scheduling option in **omp parallel for** directive, without specifying the size of chunks, while the dynamic scheduling option is used for the dynamic intensity, with the size of chunks selected as 2000, the size of rows of the grid.

For the static intensity of computations, a general conclusion from the analysis of Table 7 and Fig. 13 is a good performance of KNL processors in comparison with CPUs and KNC coprocessors. In fact, already a single KNL allows accelerating the application more than 2 times in comparison with 2 Broadwell CPUs, and about 2.8 times against a single KNC. The configuration with 2 KNL processors yields the speedup of about 1.55x against the most advanced hybrid platform with 2 CPUs and 2 KNCs. Increasing the number p of KNL devices allows further decreasing the execution time, which for $p = 6$ is reduced about 8x against 2 CPUs. While this configuration, which gives $S_p = 3.84x$, can be difficult to accept from the point of view of efficiency, the usage of 4 KNL processors seems to be fully reasonable as now $S_p = 3.12$ and $E_p = 78\%$. In this case, it becomes possible to accelerate the application about 6.7x and 3.7x against respectively 2 Broadwell CPUs and the most advanced hybrid platform.

At the same time, the performance results achieved in the case of the dynamic intensity of computations indicate pretty large

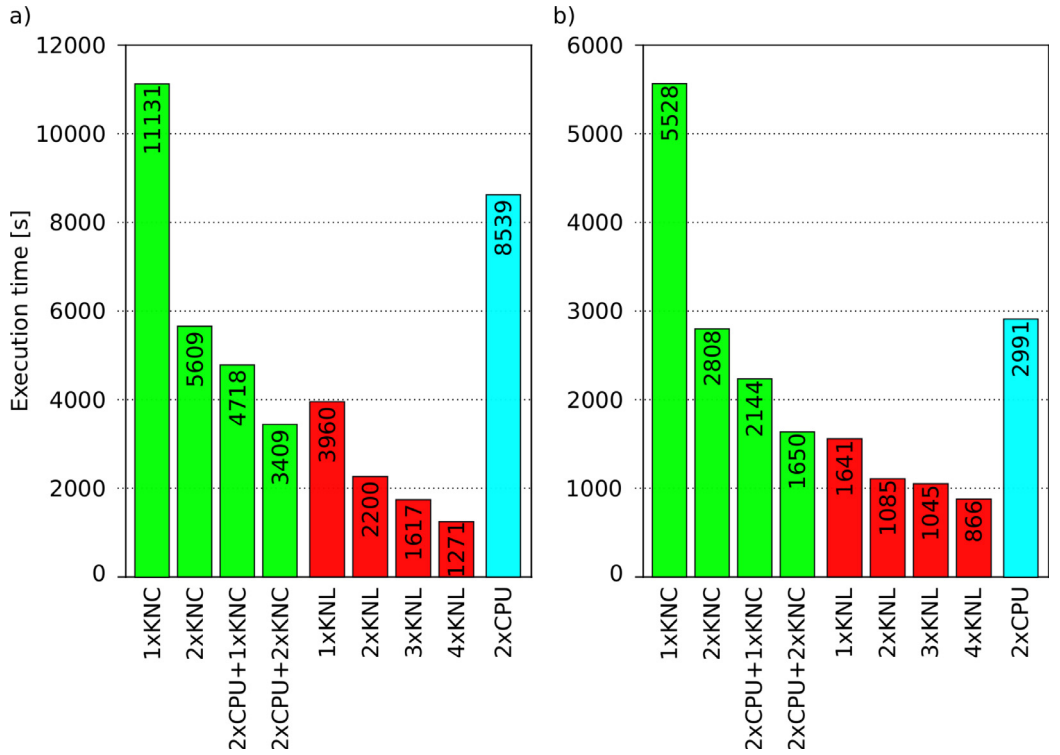

Fig. 13. Comparison of execution times achieved for platforms with KNC and KNL devices, for the static (a) and dynamic (b) intensity of computations.

room for further optimizations, beginning from a noticeably lower speedup $S_{CPU} = 1.82x$ obtained for a single KNL against 2 Broadwell CPUs, as compared to $S_{CPU} = 2.18x$ for the static intensity. It is not a surprise then that the scalability drops significantly in the whole range of tests for the dynamic intensity. For example, the usage of 4 KNL devices gives only $S_p = 1.89x$, $E_p = 47\%$, that permits accelerating the application about 3.45x and 1.9x against 2 Broadwell CPUs and the most advanced hybrid platform, respectively.

Remark 4. The platform specified in Table 6 does not utilize the Intel turbo boost technology. It is the reason that the execution times for Broadwell CPUs are somewhat higher than those for Haswell CPUs (see Table 3) despite of the slightly lower base AVX frequency of Haswell CPUs.

9. Related works

The Intel MIC architecture has been investigated by many researchers in order to speedup their applications. An exhaustive collection of such investigations for the first generation of this architecture is included in the Intel Xeon Phi coprocesor application catalog [15]. Examples of research on accelerating applications with the second generation of Intel Xeon are presented, e.q., in papers [7,14,31] and books [20,25].

The ability to fully exploit modern heterogeneous HPC systems becomes vital for optimizing the overall performance [7,14,28,40]. An example of research in this direction is the methodology proposed in our previous works [41,53] for a stencil-based application. This approach enabled us to utilize completely the available resources by distributing computations across the entire CPU–GPU hybrid platform. A new level of heterogeneous concurrent execution of a Monte Carlo photon transport simulation was presented in [51]. This simulation was extended to execute on any combination of CPUs, GPUs, and MICs concurrently. The proposed approach allows each device to repeatedly grab portions of the domain, and compute concurrently until the entire domain has been simulated. Mapping parallel graph processing on a node with KNC coprocessors is a challenge considered in [4]. The resulting hybrid CPU–MIC execution achieved the speedup of up to 1.41x over the better of CPU-only and MIC-only executions. However, this solution required a deep interference in the basic code, while in this work we assume no significant modifications of the code. The modernization of an application in order to achieve high performance on hybrid CPU–KNL platforms is described in paper [7]. In this work, MPI processes on the CPU host are responsible for offloading computations to KNL devices using the OoF technology. The shortcoming of this research is constraining the number of KNLs in the platform by only two devices.

The phase-field method is a powerful tool for solving interfacial problems in materials science [43]. It has mainly been applied to solidification dynamics [39], but it has also been used for other phenomena such as fracture dynamics [21], and vesicle dynamics [43]. The number of scientific papers related to the phase-field method exponentially increases from 1990 after Kobayashis successful dendrite growth phase-field simulation, reaching about 400 positions in 2012 (according to the SCOPUS database) [48].

A quick growth of computing power allows modeling complex solidification processes with many grains, also for 3D space. An example of this trend is the peta-scale phase-field simulation of dendritic solidification performed on the TSUBAME2.0 supercomputer powered by GPUs [42]. Therefore, the presented research is a part of global tendency to use modern computing platforms for modeling the phase-field phenomena. There are many papers devoted to modeling dendritic solidification phenomena that use such approaches as cellular automata, finite element and finite difference methods [1,5,54]. The important highlight of this study is the utilization of the generalized FDM, which permits us to model phenomena where the distribution of nodes in grids is diversified–concentrated in border areas of the inter-phase, and sparse in areas with a low diffusivity or already solidified.

In our previous works [10,44–46], we dealt with porting and optimization of the phase-field simulation of alloy solidification on CPU–KNC hybrid platforms, without signification modifications of the original application code. The second of these papers took advantages of using both CPUs and KNCs for the parallel execution of computational workloads, while in the rest of our papers parallel workloads were assigned to KNCs or CPUs only. What is also important, only a single variant of programming environments was studied in each of these works: Intel Offload in papers [45] and [44], OpenMP 4.0 Accelerator Model in [10], and hStreams in [46].

Thus, there is considerable interest in a comparative study of utilizing different programming environments to exploit resources of CPU–MIC architectures, with a real-life application as a testbed. To the best of our knowledge, the existing literature do not provide such a study. It is a common practice [28] to investigate only a single programming environment when porting an application to hybrid platforms with MICs. At the same time, there are papers presenting such comparisons for hybrid systems with GPU accelerators. For example, works [11,12] provide a comparison of performance results achieved for porting a CFD mini-application using the OpenMP 4.0/4.5 Accelerator Models and OpenACC programming standard.

There is always an open question which environments should be selected for such a study. We settled this question in favor of a widely supported standard (OpenMP) and de facto standard (Intel Offload), on the one hand, and an innovative framework offering high-level programming abstractions (hStreams), on the other hand. That is why we do not consider, for example, the HAM-Offload framework [35] that provides the means for intra- and inter-node offloading on clusters with KNC coprocessors. This framework, which was developed by Zuse Institute Berlin, allows overcoming the single-node limitation while keeping the convenient offload programming model. However, the evident disadvantage of this solution is its limited support. It is available only for the first generation of MIC architecture, while Intel Offload and OpenMP 4.5 support offloading computations to KNL processors over the HPC fabric as well.

Our study of heterogeneous programming environments for hybrid CPU–MIC platforms is restricted to the offload model. As the original code used OpenMP for parallelizing the solidification application, this choice allows us to avoid significant modification of

the initial code, while remaining within the shared-memory parallelism. Thus, we exclude the symmetric-mode approaches for KNC coprocessors that permit parallelizing applications using MPI supplemented with an appropriate threading model [7,18]. At the same time, benchmarking our application on the platform with multiple KNL processors has shown a clear need to use MPI [20,25] in this case, as a remedy against scalability limitations of the offload model and the way to take full advantage of HPC fabric.

10. Conclusions and future works

The general-purpose programming environments provided by the Intel MIC architecture allows straightforward porting of an application code to hybrid CPU–MIC platforms, assuming no significant modifications of the code. At the same time, effective parallelism is difficult to achieve in heterogeneous architectures, especially when both CPUs and accelerators are used to perform computing-intensive pieces of code. Besides achieving efficient performance, the cost of software development becomes another bottleneck preventing the wide adoption of HPC applications for hybrid platforms. Thus there is a considerably interest in studying the influence of programming environments on the efficiency of porting real-world applications to CPU–MIC platforms.

Among the main issues of achieving effective parallelism in CPU–MIC platforms are: 1) flexible management of computing resources; 2) minimization of overheads for management of device memories; 3) low-cost intra- and inter-device synchronization; 3) minimization of cost of communication between CPUs and MICs, including overlapping communication and computation. In our work, resolving these issues is based on using the heterogeneous programming model–a mixture of shared-memory and offload programming.

The main goal of this paper is the suitability assessment of various programming environments implementing this model when porting the solidification modeling application, a real-life scientific code used as a testbed. Concerning the question which environments should be selected for such a study, we settled this question in favor of a widely supported standard (OpenMP) and de facto standard (Intel Offload), on the one hand, and an innovative framework offering high-level programming abstractions (hStreams), on the other hand.

The criteria of our assessment is the overall performance of developed application codes. The software development implements the sequence of steps required for porting and optimizing the solidification application to CPU–MIC platforms, without significant modification of the original application code. This sequence includes: transformation of data layout, partitioning of computations and optimization of data movements, load balancing of workloads between devices, parallelization of computations across cores/threads, and vectorization. The way how the studied environments supports the implementation of this sequence is the basis of the assessment. Among the studied aspects which differentiated these environments are: 1) management of coprocessors; 2) way of separation of computations and writing results to the file; 3) mechanisms used for inter-device and intra-CPU synchronizations; 4) way of parallelizing the workload execution within devices.

The experimental evaluation of the four versions of the solidification application code for hybrid CPU–KNC platforms shows that excluding the OpenMP 4.0, the rest of them are able to adapt to expansion of available resources, but with different efficiency. The shortest execution time is achieved for Intel Offload. Also, the detailed analysis shows that the main reason for the poor performance of OpenMP 4.0 are issues related to the affinization of CPU threads.

At the same time, the experience gained in this study highlights separation of concerns as the main feature of hStreams which contributes to making porting and tuning a real-life code easier. For the studied application, the usage of the stream abstraction provides the high-level exposure of parallelism, where each stream encapsulates a clear functional semantics. Also, opposite to other versions, the hStreams library delivers ready-to-use constructions for all the aspects of control over mapping the application workloads to CPU–KNC platforms. What is important is that these advantages of the high-level hStreams framework are achieved at low performance overheads in comparison with the low-level Intel Offload solution.

The whole assessment is concluded with benchmarking the application performance on the platform with KNL processors. By using the Offload over Fabric technology, the adaptation of codes developed for the first generation of the MIC architecture to the second generation is almost straightforward. However, due to high performance overheads for the scenario when CPUs concurrently offload computations to KNL devices and execute parallel workloads, we release CPUs from executing the computational core of the application. Instead, we investigate the possibilities of accelerating the application by expanding the number of KNLs in the testbed up to 8 devices, using Intel Offload and OpenMP 4.5 Accelerator Model.

This work has shown the efficiency of the proposed methodology for solidification modeling problems with the static intensity of computation. However, in the case of the dynamic intensity, the scalability of utilizing computing devices drops significantly. Eliminating this disadvantage requires development of an efficient algorithm enabling run-time load balancing of workloads between devices. This is primary topic of our ongoing research. The topics of our future work include the usage of MPI to overcome scalability limitations of the offload approach on platforms with multiple KNL processors, as well as development of a performance model of the application and verifying the model based on experimental data. The functional performance model [23,24,40], which is expected to be used for this aim, requires carrying out an exhaustive set of tests for different sizes of the problem.

Acknowledgement

This work was supported by the National Science Centre (Poland) under grant no. UMO-2017/26/D/ST6/00687, and the Ministry of Science and High Education under grant no. BS/PB-1-112-3030/2018/P. The authors are grateful to: (i) Intel Technology Poland, and (ii) MICLAB project no. POIG.02.03.00.24-093/13 for granting access to HPC platforms.

References

[1] H. Adrian, K. Spiradek-Hahn, The simulation of dendritic growth in Ni-Cu alloy using the phase field model, Arch. Mater. Sci. Eng. 40 (2) (2009) 89–93.
[2] J. Benito, F. Ureñ, L. Gavete, The generalized finite difference method, in: M.P. Àlvarez (Ed.), Leading-Edge Applied Mathematical Modeling Research, Nova Science Publishers, 2008, pp. 251–293.
[3] M. Birrittella, et al., Enabling scalable high-Performance systems with the intel omni-Path architecture, IEEE Micro 36 (4) (2016) 38–47.
[4] L. Chen, X. Huo, B. Ren, S. Jain, G. Agrawal, Efficient and simplified parallel graph processing over CPU and MIC, Proc. 2015 IEEE Int. Parallel and Distributed Processing Symposium, IPDPS'15, IEEE Computer Soc., 2015, pp. 819–828.
[5] A. Choudhury, K. Reuther, E. Wesner, A. August, B. Nestler, M. Rettenmayr, Comparison of phase-field and cellular automaton models for dendritic solidification in alcu alloy, Comput. Mater. Sci 55 (2012) 263–268.
[6] Compiler assisted offload, 2017, Nov. 20, (https://software.intel.com/en-us/mkl-windows-developer-guide-compiler-assisted-offload).
[7] M. Cytowski, Z. Szymanska, P. Uminski, G. Andrejczuk, K. Raszkowski, Implementation of an agent-Based parallel tissue modelling framework for the intel MIC architecture, Sci. Program. (2017) 11pp. https://doi.org/10.1155/2017/8721612
[8] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, L. Votta, Measuring high performance computing productivity, Int. J. High Perform. Comput. Appl. 18 (4) (2004) 459–473.
[9] G. Hager, G. Wellein, Introduction to High Performance Computing for Science and Engineers, CRC Press, 2011.
[10] K. Halbiniak, L. Szustak, A. Lastovetsky, R. Wyrzykowski, Exploring OpenMP accelerator model in a real-life scientific application using hybrid CPU–MIC platforms, Third Int. Workshop on Sustainable Ultrascale Computing Systems, NESUS 2016, (2016), pp. 11–14.
[11] A. Hart, First experiences porting a parallel application to a hybrid supercomputer with OpenMP4.0 device constructs, 11th Int. Workshop on OpenMP, IWOPMP 2015, vol. 9342, Lect. Notes Comp. Sci., 2015, pp. 73–85.
[12] A. Hart, Porting the parallel Nek5000 application to GPU accelerators with OpenMP4.5, 7th UK Many-Core Developer Conference, (2016). http://conferences.inf.ed.ac.uk/UKMAC2016
[13] Hetero streams library, 2017, Nov. 20, (https://01.org/hetero-streams-library).
[14] Z. Hou, C. Zhong, C. Perez, Q. Zhang, Y. Wang, Experiences of performance optimization for large eddy simulation on intel MIC platforms, Chen G., Shen H., Chen M. (eds) Parallel Architecture, Algorithm and Programming - PAAP 2017, vol. 729, Communications in Computer and Information Science, 2017, pp. 610–625.
[15] Intel Xeon Phi coprocessor application catalog, 2014, Oct. 31, (https://software.intel.com/en-us/xeonphionlinecatalog).
[16] Intel Xeon Phi Processor x200 offload over fabric users guide, 2016, (http://registrationcenter-download.intel.com/akdlm/irc_nas/9325/oof_user_guide.pdf).
[17] Intel manycore platform software stack (Intel MPSS), 2017, October 2, (https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss).
[18] J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High-Performance Programming, Morgan Kaufmann, 2013.
[19] J. Jeffers, J. Reinders, Fast matrix computations on heterogeneous streams, in: J. Jeffers, J. Reinders (Eds.), High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches, vol. 2, Morgan Kaufmann, 2015, pp. 49–52.
[20] J. Jeffers, J. Reinders, A. Sodani, Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition, Elsevier, 2016.
[21] A. Karma, D. Kessler, H. Levine, Phase-Field model of mode III dynamic fracture, Phys. Rev. Lett. 87 (4) (2001) 4pp.
[22] A. Kulawik, The Modeling of the Phenomena of the Heat Treatment of the Medium Carbon Steel. Monographs, vol.281, Wydawnictwo Politechnki Czestochowskiej, 2013. (in Polish)
[23] A. Lastovetsky, R. Reddy, Data partitioning with a functional performance model of heterogeneous processors, Int. J. High Perform. Comput. Appl. 21 (1) (2007) 76–90.
[24] A. Lastovetsky, L. Szustak, R. Wyrzykowski, Model-based optimization of EULAG kernel on intel xeon phi through load imbalancing, IEEE Trans. Parallel Distrib. Syst. 28 (3) (2017) 787–797.
[25] J. Levesque, A. Vose, Programming for Hybrid Multi/Manycore MPP Systems, CRC Press, 2018.
[26] C. Liao, Y. Yan, B. de Supinski, D. Quinlan, B. Chapman, Early experiences with the OpenMP accelerator model, 9th Int. Workshop on OpenMP, IWOPMP 2013, vol. 8122, Lect. Notes Comp. Sci., 2013, pp. 84–98.
[27] P. Lin, C. Liao, D. Quinlan, S. Guzik, Experiences of using the OpenMP accelerator model to port DOE stencil applications, 11th Int. Workshop on OpenMP, IWOPMP 2015, vol. 9342, Lect. Notes Comp. Sci., 2015, pp. 45–59.
[28] Y. Liu, C. Yang, F. Liu, X. Zhang, Y. Lu, Y. Du, C. Yang, M. Xie, X. Liao, 623 Tflop/s HPCG run on tianhe-2: leveraging millions of hybrid cores, Int. J. High Perform. Comput. Appl. 30 (1) (2016) 39–54.
[29] MICLAB: pilot laboratory of massively parallel systems, 2015, (http://miclab.pl).
[30] Migrating offloading software to Intel Xeon Phi processor, 2018, February (https://www.intel.com/content/www/us/en/products/docs/processors/xeon-phi/migrating-offloading-software-paper.html).
[31] V. Mironov, et al., An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel Xeon Phi processor, SC'17 Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis, ACM, New York, NY, USA, 2017.
[32] National supercomputing center IT4Innovations, 2017, Nov. 16,(https://docs.it4i.cz/salomon/introduction/).
[33] C. Newburn, R. Deodhar, S. Dmitriev, R. Murty, R. Narayanaswamy, J. Wiegert, F. Chinchilla, R. McGuire, Offload compiler runtime for the Intel Xeon Phi coprocessor, 28th. Int. Supercomputing Conf., ISC 2013, vol. 7509, Lect. Notes Comp. Sci., 2013, pp. 239–254.
[34] C.J. Newburn, et al., Heterogeneous streaming, 2016 IEEE Int. Parallel and Distributed Processing Symposium Workshops, IEEE Computer Society, 2016.
[35] M. Noack, F. Wende, T. Steinke, F. Cordes, A unified programming model for intra- and inter-node offloading on Xeon Phi clusters, Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis, SC'14, IEEE Press, 2014, pp. 203–214.
[36] OpenMP Application Programming Interface, version 4.5, 2015, Nov.,(http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf).
[37] Parallel programming and optimization with Intel Xeon Phi coprocessors, 2013, (http://www.colfax-intl.com/nd/xeonphi/book.aspx).
[38] Power optimizations - Intel Xeon E5 version 3: up to 18 Haswell EP cores, 2014, Sept.,(https://www.anandtech.com/show/8423/intel-xeon-e5-version-3-up-to-18-haswell-ep-cores-/5).
[39] N. Provatas, K. Elder, Phase-Field Methods in Materials Science and Engineering, Wiley, 2010.
[40] J.-A. Rico-Gallego, A. Lastovetsky, J.-C. Diaz-Martin, Model-Based estimation of the communication cost of hybrid data-Parallel applications on heterogeneous clusters, IEEE Trans. Parallel Distrib. Syst. 28 (11) (2017) 3215–3228.
[41] K. Rojek, R. Wyrzykowski, Performance modeling of 3D MPDATA simulations on GPU cluster, J. Supercomput. 73 (2017) 664–675.
[42] T. Shimokawabe, T. Aoki, T. Takaki, Y. Yamanaka, A. Nukada, T. Endo, N. Maruyama, S. Matsuoka, Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer, Proc. 2011 ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis, SC'11, IEEE Computer Society, 2011.
[43] I. Steinbach, Phase-field models in materials science, Modell. Simul. Mater. Sci. Eng. 17 (7) (2009) 31pp.
[44] L. Szustak, K. Halbiniak, L. Kuczynski, J. Wrobel, A. Kulawik, Porting and optimization of solidification application for CPU–MIC hybrid platforms, Int. J. High Perform. Comput. Appl. (2016) 17, http://dx.doi.org/10.1177/1094342016677740.
[45] L. Szustak, K. Halbiniak, A. Kulawik, J. Wrobel, P. Gepner, Toward parallel modeling of solidification based on the generalized finite difference method using Intel Xeon Phi, Int. Conf. Parallel Processing and Applied Math., PPAM 2015, vol. 9573, Lect. Notes Comp. Sci., 2016, pp. 411–412.
[46] L. Szustak, K. Halbiniak, A. Kulawik, R. Wyrzykowski, P. Uminski, M. Sasinowski, Using hstreams programming library for accelerating a real-life application on Intel MIC, Int. Conf. Algorithms and Architectures for Parallel Processing - ICA3PP 2016, vol. 10049, Lect. Notes Comp. Sci., 2016, pp. 373–382.
[47] L. Szustak, K. Rojek, T. Olas, L. Kuczynski, K. Halbiniak, P. Gepner, Adaptation of MPDATA heterogeneous stencil computation to intel xeon phi coprocessor, Sci Program (2015) 14pp, http://dx.doi.org/10.1155/2015/642705.

[48] T. Takaki, Phase-field modeling and simulations of dendrite growth, ISIJ. Int. 54 (2) (2014) 437–444.
[49] Using automatic vectorization, 2017, Nov. 20, (https://software.intel.com/en-us/node/522572).
[50] J. Warren, W. Boettinger, Prediction of dendritic growth and microsegregation patterns in a binary alloy using the phase-field method, Acta Metall. Mater. 43 (2) (1995) 689–703.
[51] N. Wolfe, T. Liu, C. Carothers, X.G. Xu, Heterogeneous concurrent execution of monte carlo photon transport on CPU, GPU and MIC, Proc. Fourth Workshop on Irregular Applications: Architectures and Algorithms, IEEE Press, 2014, pp. 49–52.
[52] R. Wyrzykowski, K. Rojek, L. Szustak, Model-driven adaptation of double-Precision matrix multiplication to the cell processor architecture, Parallel Comput. 38 (2012) 260–276.
[53] R. Wyrzykowski, L. Szustak, K. Rojek, Parallelization of 2D MPDATA EULAG algorithm on hybrid architectures with GPU accelerators, Parallel Comput. 40 (8) (2014) 425–447.
[54] M. Zaeem, H. Yin, S. Felicelli, Modeling dendritic solidification of al3%cu using cellular automaton and phase-field methods, Appl. Math. Model. 37 (5) (2013) 3495–3503.