Received: 28 March 2020 | Revised: 3 June 2020 | Accepted: 13 August 2020

DOI: 10.1002/cpe.6011

WILEY

SPECIAL ISSUE PAPER

Exploration of OpenCL Heterogeneous Programming for Porting Solidification Modeling to CPU-GPU Platforms

Kamil Halbiniak[1] | Lukasz Szustak[1] | Tomasz Olas[1] | Roman Wyrzykowski[1] | Pawel Gepner[2]

[1]Faculty of Mechanical Engineering and Computer Science, Czestochowa University of Technology, Czestochowa, Poland

[2]Faculty of Production Engineering, Warsaw University of Technology, Warsaw, Poland

Correspondence
Kamil Halbiniak, Czestochowa University of Technology, Dabrowskiego 69, Czestochowa 42-201, Poland.
Email: khalbiniak@icis.pcz.pl

Funding information
MICLAB, Grant/Award Number: POIG.02.03.00.24-093/13; Polish National Science Centre, Grant/Award Number: UMO-2017/26/D/ST6/00687; The Program of Polish Ministry of Science and Higher Education "Regional Initiative of Excellence", Grant/Award Number: 020/RID/2018/19

Summary

This article provides a comprehensive study of OpenCL heterogeneous programming for porting applications to CPU–GPU computing platforms, with a real-life application for the solidification modeling. The aim is to achieve a flexible workload distribution between available CPU–GPU resources and optimize application performance. Considering the solidification application as a use case, we explore the necessary steps required for (i) adaptation of an application to CPU–GPU platforms, and (ii) mapping the application workload onto the OpenCL programming model. The adaptation is based on a reformulation of steps developed previously for CPU–MIC architectures. The mapping process allows us to utilize OpenCL for harnessing CPU and GPU cores using data parallelism, as well as for the management of available compute devices with task parallelism. The resulting OpenCL code's performance and energy efficiency is experimentally studied for two platforms with powerful GPUs of various generations (with Kepler and Volta architectures). The experiments confirm the performance advantage of using computing resources of both GPUs and CPUs. The achieved benefit depends on the relationship between the computing power of CPUs and GPUs. Moreover, this gain entails the growth of the average power that increases the energy consumed during the application execution.

KEYWORDS

heterogeneous computing, portability, CPU–GPU platforms, OpenCL, numerical modeling of solidification, performance-energy trade-off

1 | INTRODUCTION

Heterogeneous computing architectures have become more and more popular in many application areas.[1-4] Today, heterogeneous platforms are deployed in many setups, ranging from low-power mobile systems to high-performance computing systems.[5] The composition of a general-purpose host CPU combined with specialized computing devices (e.g., GPU, FPGA, or AI accelerator) allows many users to speed up their codes significantly,[3,6] and improve the *performance per Watt ratio*. However, reaching these goals in practice is still a challenge.[7]

Typically, the host of a heterogeneous system controls the code execution, while the time-consuming pieces of the application are offloaded to accelerators. Offloading needs to transfer data between the host and device before and after executing computing kernels. The performance overhead generated by these transfers determines where offloading is worth to be used. For amortizing these overheads, the execution of kernels

is overlapped with data movements.[8] Another way of increasing the overall performance is using the CPU host not only for the control of code execution but also to implement computing-intensive parts of the application or/and other operations such storage accesses.[9]

However, the diversity of accelerators makes crossplatform programming a big challenge,[5] thus forcing programmers to write and maintain multiple source code for an application on different platforms, for example, CUDA[10] for NVIDIA GPUs and OpenMP for CPUs.[11] OpenCL[12] addresses this cross-platform programming challenge by providing a unified programming interface for various heterogeneous systems. A wide range of devices supports OpenCL, including multicore CPUs,[13] manycore MIC architectures,[14] GPU cards (both AMD and NVIDIA), FPGA accelerators,[15] and others.

In our previous works,[9,16-19] we dealt with porting and optimization of the application for modeling the alloy solidification on heterogeneous computing platforms with Intel MIC (also known as Xeon Phi) accelerators,[8] without significant modifications of the code. The last two articles present an approach that takes advantage of using both CPUs and MICs for the parallel execution of the application workload when all available cores of both devices are utilized coherently to solve the modeling problem. The proposed method allows not only overlapping computations with data movements but also aims at optimizing the utilization of computing devices. In consequence, using resources of two Intel Xeon E5-2699 v3 CPUs (18 cores each) and two Intel KNC coprocessors (each with 61 cores), the developed approach permits accelerating the application about 2.2 times against two CPUs.

None of the programming environments used in our works (e.g., OpenMP and Intel Offload) provides the opportunity to apply them for porting the solidification application to GPUs. In this article, we focus on adapting the approach developed for CPU–MIC systems onto computing platforms with GPUs. To reach this aim, we explore the OpenCL programming model, which provides a unique capacity to write codes that can be executed across various heterogeneous platforms equipped with CPUs and accelerators. The performance of the optimized application code is investigated on two platforms. The first one consists of two 18-core Intel Xeon CPUs (with Haswell architecture) and two NVIDIA K80 GPUs, while the other platform contains two Intel Xeon CPUs with Cascade Lake architecture (18 cores each) and a single NVIDIA V100 GPU.

The contributions of this work to areas of parallel computing are as follows:

1. For a real-world scientific application for the numerical modeling of alloy solidification, we provide a comprehensive study of porting applications to heterogeneous computing platforms with GPU accelerators, aiming at achieving a flexible workload distribution between available CPU–GPU resources and optimizing the application performance.
2. Considering the solidification application as a use case, we explore the basic steps required for (i) adaptation of an application to heterogeneous CPU–GPU platforms, based on a reformulation of steps developed previously[19] for CPU–MIC architectures, and then (ii) mapping the application workload onto the OpenCL programming model. As a result, the mapping process allows us to utilize OpenCL for harnessing CPU and GPU cores using data parallelism, as well as for the management of available compute devices with task parallelism.
3. Experimental evaluation of the performance of the resulting OpenCL code on two platforms with powerful GPUs of various generations (with Kepler and Volta architectures) confirms the performance advantage of using computing resources of both GPUs and CPUs. For example, the configuration with a single NVIDIA K80 card and two 18-core Intel Xeon E5-2699 v3 CPUs (Haswell architecture) permits accelerating the application 1.41 times against using only a single K80. At the same time, the achieved benefit depends strongly on the relationship between the computing power of CPU and GPU devices.
4. By investigating the correlation between the performance and energy/power consumed by the whole platform during the application execution, we discover that performance gain achieved by utilizing both GPU and CPU cores entails the growth of the average power that increases the energy consumption. For example, for the configuration with two Haswell-based CPUs and a single K80 card, the speedup of 1.41 times is achieved at the cost of 1.67 times higher average power, which increases the energy consumption 1.19 times against a single K80.

This article is organized as follows. Related works are discussed in Section 2, while Section 3 introduces the numerical model of alloy solidification, as well as describes the basic version of the solidification application. Section 4 provides a concise overview of heterogeneous computing systems with GPU accelerators, including platforms used in our experiments. The OpenCL programming model is also outlined in this section. Section 5 presents an approach to adapting the solidification application to heterogeneous platforms with GPU accelerators, while Section 6 describes how this approach is mapped onto the OpenCL programming model. Section 7 presents the performance results achieved for the OpenCL implementation of the studied application on two computing platforms with CPUs and GPUs of different generations, while Section 8 discusses the correlation between performance and energy/power consumption. Section 9 concludes the article and addresses future work.

2 | RELATED WORKS

The capability to utilize efficiently modern heterogeneous HPC systems becomes indispensable for improving the overall performance.[3,20,21] An example of research in this area is the approach proposed in our articles[2,22] for a stencil-based computational fluid dynamics application. The proposed methodology permits us to utilize the available resources effectively by dispersing computations across the entire CPU–GPU platform. A high level of heterogeneous parallel execution of a Monte Carlo photon transport simulation was achieved in Reference 23. This application was reinforced to execute in parallel on any combination of CPUs, GPUs, and MICs.

Apart from obtaining high performance, the complexity of software development is another bottleneck hampering the wide acceptance of heterogeneous systems. For this reason, there is sufficient interest in studying the impact of programming environments on the efficiency of porting real-life applications to heterogeneous architectures. For systems with MIC accelerators, we used[19] various offload-based programming environments, namely, (1) Intel Offload[24] coupled with OpenMP, (2) OpenMP Accelerator Model,[11] and (3) hStreams framework[1] with OpenMP. But none of them can be applied for GPUs.

The OpenCL standard permits the same code to be executed across various devices, including GPU accelerators, which are increasingly used for scientific computing.[25,26] This standard allows[27] the user to continue to use the same programming language, with no modifications, while benefiting automatically from the heterogeneous performance. While OpenCL provides functional portability, it does not necessarily offer performance portability. Programs have to be tuned and even rewritten to deliver performance when targeting new processors and platforms.[13,15,28] In this context, there is a need for a more formal analysis of the expected performances, that can help to understand the choices, identify bottlenecks, and drive directions of optimizations. Examples of relevant references concerning the performance analysis and prediction to support the design process can be found in articles.[29-32]

In particular, the evaluation of the applicability of OpenCL for multicore CPUs showed[13] that OpenCL yields competitive performance when compared with the two CPU specific framework—OpenMP and Intel TBB. Another important conclusion was that enabling compiler optimizations and explicit vectorization in OpenCL kernels is essential for tuning performance. The authors reported that OpenCL codes written for CPUs could successfully run on NVIDIA Tesla GPUs. However, the performance portability was still by far suboptimal. So there are multiple attempts to improve crossplatform performance portability for OpenCL platforms. One of them is a source-to-source OpenCL compiler, PPOpenCL.[5] It was implemented in Clang and is based on fusing the host and kernel thread codes of an OpenCl program. Developing a single data-parallel kernel in OpenCl is also implemented in the SKMD framework.[33] Its primary functionality is orchestrating a collaborative execution of this kernel across CPUs and GPUs transparently. However, all these experimental developments did not deliver a practicable environment to be used by application programmers.

The main goal of this article is to explore the capabilities of the OpenCL unified programming model as a tool for porting real-world scientific applications to heterogeneous CPU–GPU platforms, with numerical modeling of alloy solidification selected as a use case. At present, the fast expansion of computing capabilities enables modeling of complex solidification processes. Examples of this trend are peta-scale phase-field simulations of dendritic solidification performed on the TSUBAME2.0 supercomputer,[25] as well as large-scale phase-field simulations of ternary eutectic solidification.[34,35] However, to exploit parallel computing systems with GPU accelerators, these simulations utilize not a unified programming model but the mixture of two programming standards—OpenMP and CUDA. The significant highlight of our work is also the usage of the generalized finite difference method,[36] which allows us to model processes where the distribution of nodes in grids is diversified—concentrated in border domains of the interphase, and sparse in domains with a low diffusivity or already solidified. In this study, besides the performance, we also explore the energy efficiency of computations. Given the rapidly climbing power bills, and the negative impact of energy production technologies on the environment, achieving the energy efficiency of parallel systems and applications has become one of the most challenging issues.[37,38]

3 | APPLICATION: NUMERICAL MODELING OF SOLIDIFICATION

3.1 | Overview of numerical model

In this work, a binary alloy of Ni-Cu is modeled as a system of the ideal metal mixture in liquid and solid phases. The numerical model is based on the field-phase method proposed by Warren and Boettinger[39] and describes the dendritic solidification process[40,41] in the isothermal conditions with constant diffusivity coefficients for both phases. In this model, the growth of microstructure is calculated by solving a system of two partial differential equations, where the first one determines the phase content ϕ:

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

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:

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

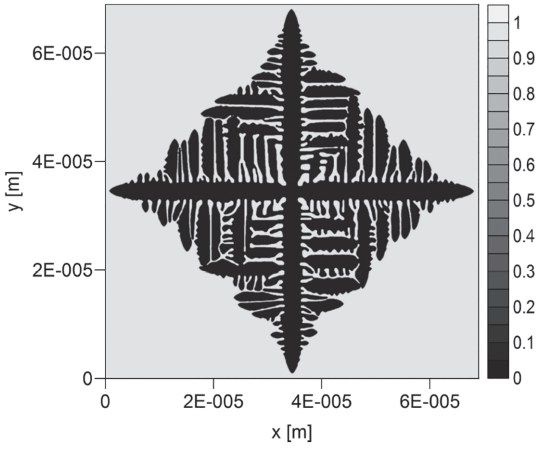


FIGURE 1 Phase content for the simulated time $t_S = 2.75 \times 10^{-3}$s

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

$$\frac{\partial c}{\partial t} = \nabla \cdot D_c \left[\nabla c + \frac{V_m}{R} c(1-c)(H_B(\phi, T) - H_A(\phi, T)) \nabla \phi \right], \tag{3}$$

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

In the model, the generalized finite difference method[42,43] is used to calculate partial derivatives in Equations (1) and (2). Parallelizing computations with the desired accuracy is based on the explicit scheme with a small time step $\Delta t = 1e - 7$s.

The resulting computations[9] correspond to a forward-in-time, iterative algorithm since all calculations performed in the current time step k depend on results determined in the previous step $k - 1$. 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 (Figure 1). For this aim, the values of derivatives in all nodes have to be calculated at every time step.

In our previous works,[9,19,44] two different cases were introduced—with the static and dynamic intensity of computations. In the first case, the workload of computing resources is constant during 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, which is changing during the simulation following the growth of microstructure. The use of a suitable selection criterion allows reducing significantly the amount of computations. However, this results in a significant workload imbalance, since the selection criterion is calculated after the static partitioning of the grid nodes across computing resources. In this work, we focus on the solidification application with static computational intensity.

3.2 | Basic version of the solidification application

Listing 1 illustrates the computational core of the solidification application with the static computational intensity for a single time step. All computations are organized as a single nested loop, which iterates over the entire grid and consists of four kernels. The kernels K_1 and K_3 are responsible for calculations performed in the boundary nodes, while kernels K_2 and K_4 execute computations within the internal part of the grid. Each of these kernels contains an inner loop that iterates over neighbors of a given node. This loop refers to stencil computations[22] responsible for calculating partial derivatives. The selection of the boundary and internal nodes of the grid is implemented using a single *if* statement. The structure of the application permits its parallelization with **omp parallel for** directive of OpenMP for the outer loop. The application code studied in this article (see Listing 1) is a result of performance optimizations proposed in our previous work.[44] Thus, it is out of the scope of this article how the code has been obtained.

```
#pragma omp parallel for
for(int i=0; i<grid_size; ++i) {
  // Computations performed within a grid node
  if(node_isBoundary[i]) {
   // Execution of the kernels K1 and K3
```

```
   // corresponding to the boundary nodes
   Kernel1;
   Kernel3;
  }
  else {
   // Execution of the kernels K2 and K4
   // corresponding to the internal nodes
   Kernel2;
   Kernel4;
  }
}
// Completing computations within a single time step
// with swapping arrays using pointers
swap(node_conc, node_conc0);
swap(node_Fi, node_Fi0);
swap(node_Dc, node_Dc0);
```

Listing 1: General scheme of basic version of solidification application

Listings 2 and 3 present the code snippets corresponding to the kernels K_1 and K_2. These kernels are responsible for determining the dopant concentration for the boundary and internal nodes of the grid, using a single stencil for kernel K_1 and 15 stencils for kernel K_2. The structure of kernels K_3 and K_4 is analogous; they perform computations using, respectively, a single and five stencils. For a particular grid node *i*, the indices `node_e[offset+j]` of its neighbors are kept in the configuration file that describes the entire grid. Consequently, the patterns of all 22 stencils are determined during the application execution.

The codes studied in this article uses the structure of arrays layout of memory. All computations are executed using a set of one-dimensional (1D) arrays that contain values of variables in grid nodes. For example, `node_Fi0[i]` and `node_Dc[i]` correspond, respectively, to the values of phase content and diffusion coefficient for the *i*th node.

```
const int offset = i*max_neighbors;
double d0(0.0), d2(0.0);
double z[max_neighbors];
/.../
for(int j=0; j<neighbors_count[i]; ++j) {
 // Stencil computations used for determination
 // of partial derivatives
  z[j] = 1.0/pow(node_h[offset+j],2*m)*
     ((node_g2[offset+0]*node_cosAlf[i]+
     node_g2[offset+2]*node_cosBet[i])*
     node_hx[offset+j]+
     (node_g2[offset+1]*node_cosAlf[i]+
     node_g2[offset+3]*node_cosBet[i])*
     node_hy[offset+j]);
  const int idx = node_e[offset+j];
  d2 += node_conc0[idx]*z[j];
  /.../
}
// Computations executed within nodes
/.../
```

Listing 2: The structure of kernel K_1

```
const int offset = i*max_neighbors;
const int gOffset = i*25;
double d1xCj(0.0), d1xDcj(0.0), d1xFj(0.0);
double zx[max_neighbors];
/.../
for(int j=0; j<neighbors_count[i]; ++j) {
 // Stencil computations used for determination
 // of partial derivatives
 // 3 of all 15 stencils used in kernel K2
 zx[j] = 1.0/pow(node_h[offset+j],2*m)*
     (node_g[gOffset+0]*node_hx[offset+j]+
     node_g[gOffset+1]*node_hy[offset+j]+
     0.5*node_g[gOffset+2]*
     node_hx[offset+j]*node_hx[offset+j]+
     0.5*node_g[gOffset+3]*
     node_hy[offset+j]*node_hy[offset+j]+
     node_g[gOffset+4]*
     node_hx[offset+j]*node_hy[offset+j]);
 const int idx = node_e[offset+j];
 d1xCj += zx[j]*node_conc0[idx];
 d1xDcj += zx[j]*node_Dc[idx];
 d1xFj += zx[j]*node_Fi0[idx];
 /.../
}
// Computations performed within nodes
/.../
```

Listing 3: The structure of kernel K_2

4 | HETEROGENEOUS COMPUTING SYSTEMS AND PROGRAMMING MODELS

4.1 | Architecture of CPU–GPU platforms

Heterogeneous computing platforms combining traditional CPU processors and GPU accelerators have become very popular in HPC solutions. The CPU–GPU servers are delivered in a variety of configurations with devices from different vendors (Intel, IBM, NVIDIA, AMD, and so forth). However, the most common platforms are equipped with Intel CPUs and NVIDIA GPUs (https://www.top500.org/lists/2019/06/).

The GPU accelerators are increasingly used for scientific computing. They allow achieving high performance for some of the workloads by providing thousands of relatively simple in-order execution cores, without branch prediction, and the caches much smaller than the CPU ones.[33] Unlike CPU cores, the best performance is usually obtained by running numerous threads on a single GPU core to overlap computation and memory delays. On the contrary, CPUs offer reasonable performance for a broad class of workloads, using multiple cores clocked at a higher frequency, SIMD (single instruction, multiple data) processing units, an efficient cache hierarchy, and low-level latency memory access.[33] Figure 2 shows an example of a heterogeneous CPU–GPU platform with two Intel Xeon CPUs and two NVIDIA Tesla K80 GPU accelerators. According to the concept of the ccNuma architecture, the CPU processors are connected to each other using the Intel QPI bus. The GPU accelerators are delivered in the form of external PCI devices, so transfers of data between the main memory of the platform and accelerator memory are performed via the PCIe bus.

The GPU accelerators do not support general-purpose programming environments available for traditional CPUs. Thus, GPUs' efficient exploitation requires using dedicated programming languages and tools (compilers, and so forth). Examples of such tools are CUDA and OpenCL. While the first one is dedicated only for NVIDIA GPUs, the second supports GPUs from different vendors and also allows utilization of heterogeneous resources of platforms equipped with both CPUs and GPUs. Besides the low-level OpenCL and CUDA programming environments, there are also high-level directive-based approaches for GPU programming like OpenACC.[45,46]

Running computations on GPU accelerators is commonly associated with the usage of the offload model. In this model, the programmer chooses sections of source code (usually functions) to be moved from CPU to accelerators for processing. For generating binaries for GPUs, the selected

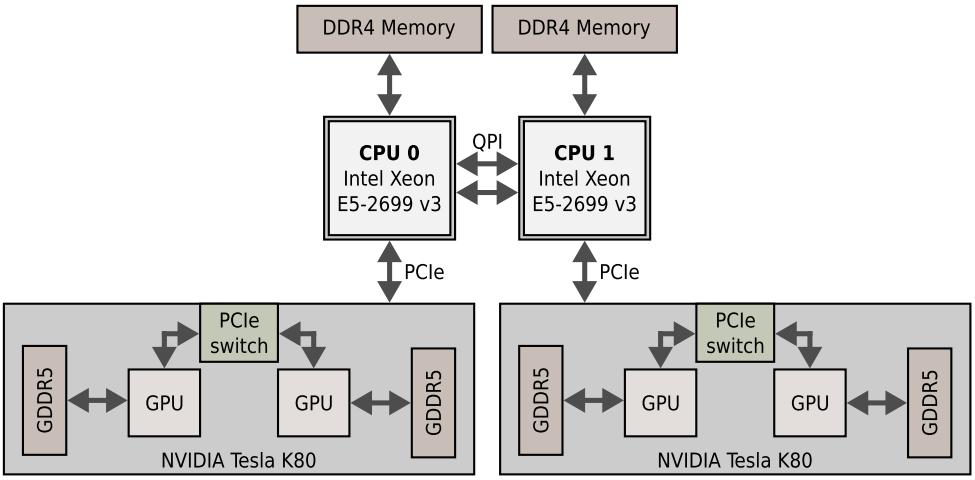


FIGURE 2 Heterogeneous architecture combining Intel Xeon CPUs and NVIDIA Tesla K80 accelerators

segments of the application code are compiled using a dedicated compiler, while the rest of the code is compiled with the host compiler. Executing the resulting code starts on the host, while the selected sections of the code are transferred to GPUs at runtime.

In this study, we use two heterogeneous CPU–GPU platforms. The first one consists of two Intel Xeon e5-2699 v3 CPUs (Haswell-architecture) with 256 GB of DDR4-2133 main memory, and two NVIDIA Tesla K80 GPU accelerators (Kepler) architecture). Each CPU contains 18 cores, clocked by the base frequency of 2.3 GHz, while for the workloads using the AVX2 vector extension, the CPU reduces the clock frequency to 1.9 GHz. A single NVIDIA Tesla K80 accelerator consists of two GPU processors connected to each other via the PCIe switch. Each GPU processor contains 2496 CUDA cores, clocked by the base frequency of 562 MHz, while the usage of boost mode increases the clock frequency to 1456 MHz. The NVIDIA K80 accelerator provides 24 GB of on-board GDDR5 memory (12 GB per GPU processor). For double precision floating-point operations, the theoretical peak performance of the first platform is 6918 Gflop/s ($2 \times 547 + 4 \times 1456$). This value of peak performance assumes the usage of fused multiply-add instructions for both CPUs and GPUs. Table 1 presents the specification of Intel Xeon CPUs and NVIDIA Tesla K80 accelerators used in this work.

TABLE 1 Specification of the first platform

CPUs: 2 × Intel Xeon E5-2699 v3	
Number of cores (threads)	2 × 18 (2 × 36)
Base frequency (GHz)	2.3
(AVX2 frequency)	(1.9)
AVX2 peak for DP (Gflop/s)	2 × 748.5
(AVX2 Turbo frequency)	(2.6 GHz)
L3 cache (MB)	2 × 45
Memory size	2 × 128 DDR4-2133
Memory bandwidth (GB/s)	2 × 68.3
TDP (W)	2 × 145
GPU accelerators: 2 × NVIDIA Tesla K80	
Architecture of Tesla K80	Dual GPU
Number of cores per SM	192
Number of SMs	4 × 13
Numer of CUDA cores	4 × 2496
Boost frequency (GHz)	0.875
Peak for DP (Gflop/s)	2 × 2912 (Boost)
Shared memory per SM (KB)	112
Global memory size	4 × 12 GB GDDR5
Memory bandwidth (GB/s)	4 × 240
TDP (W)	2 × 300

TABLE 2 Specification of the second platform

CPUs: 2 × Intel Xeon Gold 6240	
Number of cores (threads)	2 × 18 (2 × 36)
Base frequency (GHz)	2.6
(AVX512 frequency)	(1.6)
AVX512 peak for DP (Gflop/s)	2 × 1440
(AVX512 Turbo frequency)	(2.5 GHz)
L3 cache (MB)	2 × 24.75
Memory size	2 × 96 DDR4-2993
Memory bandwidth (GB/s)	2 × 131
TDP (W)	2 × 150
GPU accelerator: NVIDIA Volta V100	
Number of cores per SM	32
Number of SMs	80
Numer of CUDA cores	2560
Boost frequency (GHz)	1.53
Peak for DP (Tflop/s)	7.8 (Boost)
Shared memory per SM (KB)	96
Global memory size	32 GB HBM2
Memory bandwidth (GB/s)	900
TDP (W)	300

The specification of the second platform is shown in Table 2. It consists of two Intel Xeon Gold 6240 CPUs (Cascade Lake architecture) and a single NVIDIA Tesla V100 GPU card (Volta architecture). This GPU accelerator contains 2560 CUDA cores and is equipped with 32 GB of HBM2 memory that provides the bandwidth of 900 GB/s.

4.2 | Portable heterogeneous programming with OpenCL

The OpenCL (Open Common Language) is a parallel programming standard for modern heterogeneous computing systems.[12] The OpenCL framework supports the program execution on CPUs, GPUs, FPGAs, and many other devices. It is an open standard maintained by a nonprofit technology consortium (named Khronos Group), with implementations provided by Intel, AMD, NVIDIA, Xilinx, ARM, and others. Although the framework supports code portability, performance portability is hard to achieve since optimizations of OpenCL implementations for various platforms are different.[13]

The main ideas of OpenCL are expressed based on the following hierarchy of models: (i) platform model, (ii) execution model, (iii) memory, and (iv) programming models. In particular, OpenCL provides a platform-independent abstract platform model that allows arranging computations and data access.[5] This model is based on the host-centric view, where the platform consists of a host connected to one or more OpenCL compute devices (e.g., GPUs). Each compute device is divided into one or more compute units, which are further divided into several processing elements. The platform device model closely maps to the architecture of GPUs. For example, for NVIDA GPUs, the processing element refers to a single GPU (or CUDA) core, while the compute unit corresponds to a streaming multiprocessor (SM). The concept of the OpenCL platform model is shown in Figure 3.

The OpenCL execution model consists of two components: (i) kernels and (ii) the host program. Kernels are basic units of code executed on compute devices. When the kernel is submitted for execution on a compute device, N-dimensional range of indexes (or NDRange) is created, where $N \in \{1, 2, 3\}$. Elements of this index space are called work-items and execute the same kernel on different data. The work-items are grouped into a local work-groups and can be synchronized within the work-groups during execution. OpenCL does not provide mechanisms for synchronization between work-groups—they are executed independently.[22] The work-items within a work-group are mapped onto processing elements within compute units. The host program is a sequential part of the application running on the CPU host processor. It is responsible for setting up and managing the execution of kernels on OpenCL devices using command queues. Commands that can be inserted into the queues are grouped into three categories: (i) memory commands, (ii) kernel execution commands, and (iii) synchronization commands. The OpenCL command queues support two modes of execution: in-order and out-of-order.

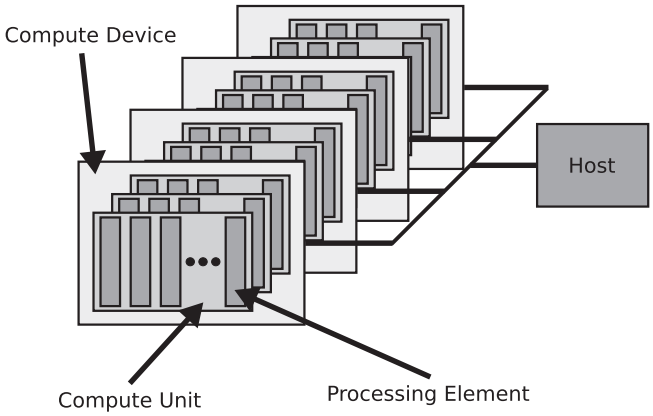


FIGURE 3 OpenCL platform model with one host and one or more OpenCL compute devices

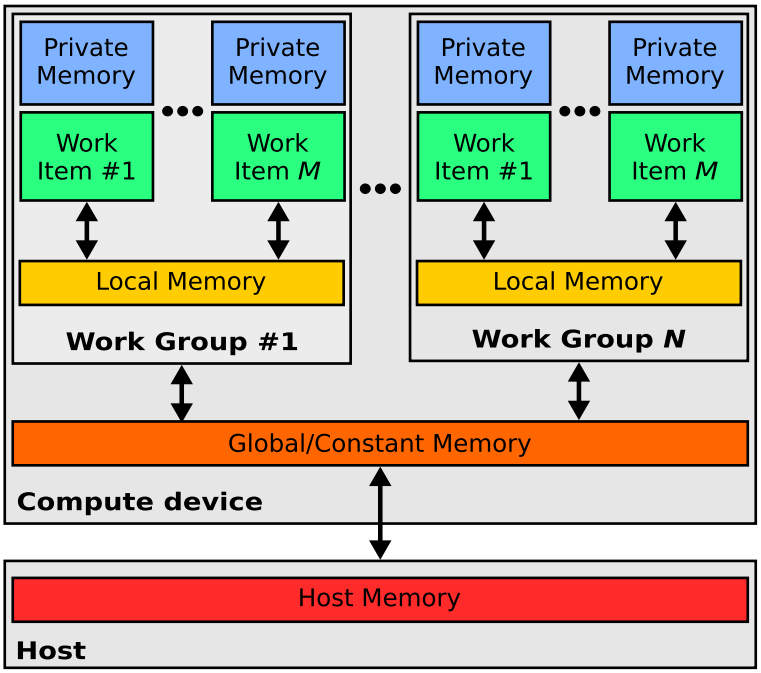


FIGURE 4 The structure of the OpenCL memory model

The OpenCL memory model assumes the host side memory and four types of memories on a compute device side: global, constant, local, and private[12,47] (Figure 4). The global memory is accessible for all work-items within all work-groups. It is also used to exchange data between the host and the compute device. The local memory is used to share data within a single work-group. All work-items within the same work-group can read and write data to this memory. The last memory type—private memory—is unique to individual work-items. The data stored in the private memory of a given work-item are not visible to other work-items. By default, all local variables and nonpointer arguments of kernels are private.

The memory model closely corresponds to the memory hierarchy of modern GPUs.[22] For NVIDIA GPUs, the private memory is implemented as a register file, the local memory refers to shared memories of SMs, while the global memory is the main memory of the accelerator. In the case of CPUs, the memory model is mapped onto the physical memory hierarchy differently. Now, the private memory refers to registers and L1 cache, the local memory is implemented as L2/L3 cache, and the global memory corresponds to the main memory.

The OpenCL standard supports two parallel programming models: (i) data-parallel and (ii) task-parallel. In the first one, the same kernel is submitted to be executed simultaneously by compute units or processing elements.[12,22] In the second model, different kernels are passed to be executed concurrently on compute units (or processing elements) within a single OpenCL compute device or multiple devices.[47]

5 | IDEA OF ADAPTING SOLIDIFICATION APPLICATION TO HETEROGENEOUS PLATFORMS WITH ACCELERATORS

In our previous works,[9,19] we developed an approach for porting and optimizing real-life scientific applications such as the solidification modeling to heterogeneous platforms with Intel MIC accelerators. This approach was used to adopt the solidification application to heterogeneous

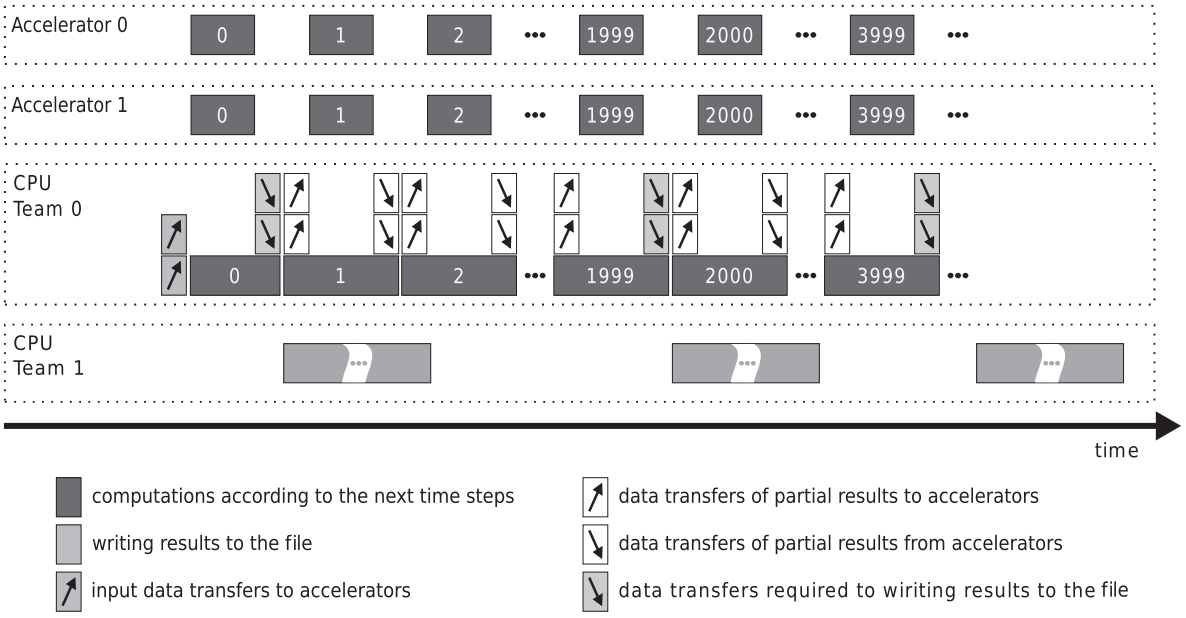


FIGURE 5 Idea of adapting the solidification application to heterogeneous platforms with accelerators

platforms equipped with Intel KNC/KNL accelerators. The main goal was to accelerate the simulation using all available computing resources of target platforms. An important assumption was avoiding significant modifications to the application code. In the proposed approach, both CPU and accelerators are used for executing major parallel workloads, while the rest of the application is performed using CPUs only, as it does not require massively parallel resources. Such a solution allows the efficient utilization of computing resources but requires flexible methods for workload distribution between CPUs and accelerators.

The idea of adapting the solidification application to the heterogeneous platform with two accelerators (MICs or GPUs) is shown in Figure 5. The presented scheme could be easily extended to more accelerators. While both accelerators and CPUs are used for executing the computational core of the studied application, writing data to the file is a task of CPUs only. A crucial performance issue was to overlap computations with writing partial results to the file. Since CPUs are utilized for both calculations and data writing, CPU threads are partitioned into two work teams. More precisely, writing data to the file is assigned to the second team consisting of a single thread, while the first team is used for the execution of calculations on CPUs, and control of accelerators. Such a solution allows us to simultaneously perform parallel computations with all devices and to write data to the file.

At the beginning of computations, all input data are transferred simultaneously to accelerators, which then start calculations for the first time step, together with the first CPU work team. After finishing the computations by accelerators, the results are returned to the main memory. During these transfers, the first CPU work team finishes calculations for the first time step. The second CPU work team start writing results to the file immediately after obtaining outcomes from the accelerators, as well as from the first CPU work team. At the same time, this work team and accelerators start computations for the next time step. Generally, writing results obtained after finishing a package of R steps is overlapped with computations implemented for the next package.

In this work, we focus on mapping the proposed approach onto heterogeneous platforms with GPU accelerators. The next subsections present a sequence of steps that are necessary for the adaptation of the studied application to CPU–GPU heterogeneous platforms, namely:

1. Partitioning of computations and optimizations of data movements;
2. Load balancing of workloads between CPUs and GPUs;
3. Parallelization of computations across cores;
4. Vectorization.

5.1 | Partitioning of computations and optimizations of data movements

The first CPU–GPU platform (see Table 1) used in this work consists of three components: CPUs with ccNUMA architecture, and two GPU accelerators. Since each accelerator contains two GPU processors, five devices are finally used in the computations. To split computations across the

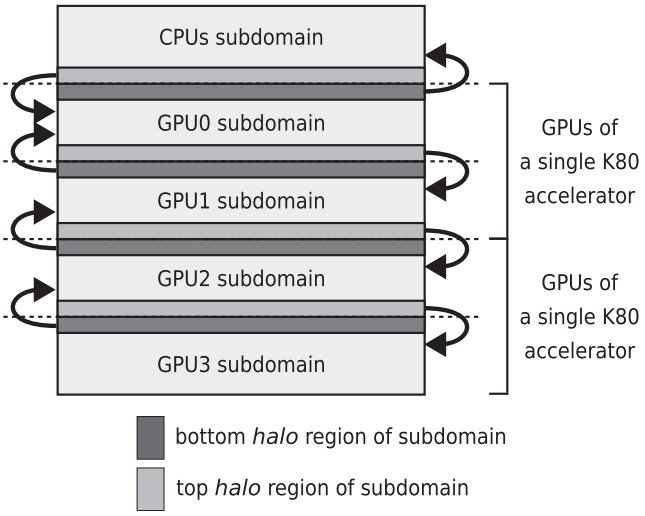


FIGURE 6 Scheme of communication between CPUs and K80 accelerators

available devices, we apply a 1D data partitioning scheme. In consequence, all computations are partitioned along the index *i* of the outer loop (see Listing 1) into five parts corresponding to grid subdomains, with the first one assigned to CPUs, while the next subdomains are allocated to the subsequent four GPU processors of two K80 accelerators (Figure 6).

Due to data dependencies between grid nodes, some data have to be transferred between devices in successive time steps. Since GPUs are used in the offload mode, all CPU–GPU data transfers are performed through the PCIe bus and host memory. OpenCL provides two scenarios for the data exchange: (i) implicit and (ii) explicit. In this work, we use the second scenario, which assumes the utilization of buffers and memory commands of OpenCL runtime to implement the transfer of data. To reduce communication overheads, we exchange only data corresponding to halo regions of the subdomains (see Figure 6). Moreover, nonblocking memory commands are used to provide overlapping data transfers between the host and all compute devices.

5.2 | Load balancing of workloads

To determine the optimal workload distribution between the five subdomains determined in the previous subsection, we develop the algorithm for dynamic load balancing. This algorithm assumes that all accelerators in the platform are homogeneous. Initially, all computations are partitioned uniformly along rows of the grid. The calibration of load balancing is performed during the simulation's initial steps, based on measurements of execution times for subdomains. This process relies on comparing the time T_1 measured for the CPUs subdomain, and the time T_2 corresponding to the shortest execution time among subdomains assigned to GPU devices. Depending on the comparison result, the size of the CPUs subdomain is increased or decreased by an amount of rows equal to the number of GPU processors. As a result, a single row is, respectively, removed or add to each GPU subdomain.

To ensure the correctness of the simulation, such a redistribution of computations leads to additional data transfers between devices. This calibration is finished when the execution times T_1 and T_2 become equal with a given accuracy Δ determined in the following way:

$$\Delta = \frac{|T_1 - T_2|}{\min(T_1, T_2)} * 100\%. \tag{4}$$

Assuming $\Delta = 3\%$, the proposed algorithm requires at most 100 time steps to determine the optimal workload distribution between CPU and GPU devices.

5.3 | Parallelization of computations across cores of CPUs and GPUs

To distribute computations across the available cores of both CPUs and GPUs, we use the primary work-sharing mechanism of OpenCL, which is based on the creation of the *N*-dimensional space of work-items. Due to the grid representation as a 1D array (see Listing 1), a 1D space of work-items is created to parallelize computations. An OpenCL kernel executed by each work-item in this space is responsible for calculating new values in a single node of the grid. Consequently, the number of threads during the application execution corresponds to the number of grid nodes. For ensuring the best overall performance of computations, different sizes of work-groups for CPUs and GPUs are evaluated.

5.4 | Vectorization

The usage of SIMD processing capabilities is vital for performance optimization. The implementation of vector processing in OpenCL is possible using vector data types that represent SIMD registers. The OpenCL framework provides arithmetic and logic operations for these vector types. From the programmer point of view, vector data types can be used in the same way as scalar ones. The usage of vector types makes it easy to write a portable code since the OpenCL compiler is responsible for mapping the OpenCL vector operations to appropriate vector instructions of a device.[48]

At the same time, modern GPUs implement SIMD processing implicitly. It is achieved using the single instruction, multiple threads (SIMT) execution model, where threads simultaneously execute the same scalar instruction on different data elements. The block of threads which execute the same instruction on a multiprocessor in the SIMT fashion is called the warp. From the programmer point of view, there is no necessity to implement any special SIMD instructions (e.g., intrinsics) in the code executed by the GPUs. In consequence, we decide to apply the OpenCL vector data types only for the workloads executed by the CPUs. In particular, these vector types are used to vectorize the innermost loops (see Listings 2 and 3), which refer to stencil computations.

6 | MAPPING APPLICATION WORKLOAD ONTO OPENCL PROGRAMMING MODEL

The OpenCL framework supports heterogeneous data and task parallelism by inserting commands into command queues resided on devices of a platform. This advantage can be successfully used for adapting the solidification application to computing platforms with GPUs. In practice, it is natural to use OpenCL for harnessing cores of compute devices using the data-parallel programming model, as well as for the management of available devices based on task parallelism.

The next subsections outline steps that are required to map the application workload onto OpenCL, namely:

1. Mapping command queues onto devices;
2. Implementing the kernel executed on CPUs and GPUs;
3. Management of memory resources;
4. Synchronization of queues.

6.1 | Mapping command queues onto devices

Figure 7 presents the idea of mapping the solidification modeling application onto the OpenCL heterogeneous environment corresponding to the platform specified in Table 1 that consists of five compute devices. Consequently, the parallelization of the application with OpenCL

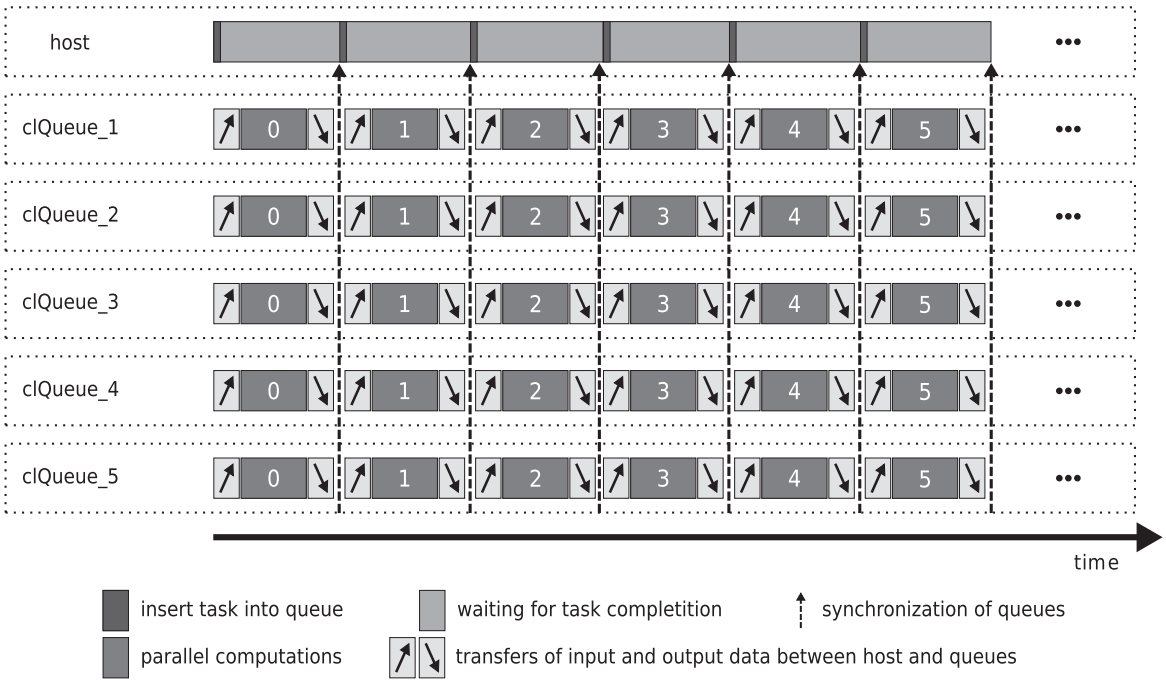


FIGURE 7 Mapping the application workload onto OpenCL heterogeneous programming environment

requires creating five command queues mapped onto these devices. The queues are responsible for submitting commands which provide execution of parallel computations within devices and data transfers between the host and devices. A single queue is assigned to both CPUs, while the rest of the queues reside on four GPU processors of two K80 accelerators. The queues execute commands following the in-order fashion.

The essential requisite of each queue is the OpenCL event profiling mechanism. In this work, we used this mechanism to measure the execution time T_s for grid subdomains while processed on different devices. For a given subdomain, T_s is determined as the elapsed time between two events:

- `CL_PROFILING_COMMAND_START`,
- `CL_PROFILING_COMMAND_END`.

6.2 | Implementation of kernel on CPUs and GPUs

Listing 4 shows the structure of the OpenCL kernel used to parallelize the application on CPU–GPU platforms. As OpenCL supports code portability across various computing architectures, we utilize the same kernel for submitting workload to CPUs and GPUs. Therefore, the same parallelization scheme is applied to both devices.

```
void solidificationKernel(__global const int *node_isBoundary,
           /* rest of arrays used in computatuons */,
           const int grid_size)
{
  const int i = get_global_id(0);
  if(i < grid_size) {
   // Computations performed within the i-th node of the grid
   if(node_isBoundary[i]) {
     // If the i-th node is a boundary one, then execute kernels K1 and K3
     // corresponding to the boundary condition
     Kernel1;
     Kernel3;
   }
   else {
     // For internal nodes execute kernels K2 and K4
     // corresponding to computations within the internal part of the grid
     Kernel2;
     Kernel4;
   }
  }
}
```

Listing 4: The structure of OpenCL kernel used to parallelize the solidification application

While CPUs and GPUs use the same OpenCL kernel to run the parallel workload, there are some differences in the implementation of the application kernels that perform computations within grid nodes. These differences correspond to various ways of utilizing SIMD processing in the CPUs and GPUs (see Section 5.4). In the case of GPUs which implement vectorization implicitly, it is enough to use the scalar version of all application kernels (see Listings 2 and 3). Simultaneously, the efficient use of CPU SIMD capabilities requires the explicit vectorization of the application kernels. Listing 5 illustrates such a vectorization for the kernel K_2. Here, the `double4` type corresponds to a 4-component vector of `doubles`. To vectorize stencil computations performed within the innermost loops, we introduce temporary vectors (e.g., `conc0_temp[]`) responsible for loading the necessary data from irregular memory regions. For a given node, these temporary vectors are used to load the values of its neighbors into the OpenCL vector types. For the other application kernels, the vectorization is implemented similarly. The choice between scalar and vectorized versions of the kernels is performed at runtime, during the compilation of the OpenCL kernel for CPU and GPU architectures.

```
const int offset = i*max_neighbors;
const int gOffset = i*25;
double d1xCj(0.0), d1xDcj(0.0), d1xFj(0.0);
double conc0_temp[max_neighbors];
double Dc_temp[max_neighbors];
double Fi0_temp[max_neighbors];
double4 zx;
double4 ones = (double4) (1.0);
double4 half = (double4) (0.5);
int4 m2 = int4 (2*m)
int vecCounter = 0;
/.../
for(int j=0; j<max_neighbors; ++i) {
  const int idx = node_e[offset+j];
  conc0_temp[j] = node_conc0[idx];
  Dc_temp[j] = node_Dc[idx];
  Fi0_temp[j] = node_Fi0[idx];
}
for(int j=0; j<neighbors_count[i]; i+=4) {
  double4 g0 = (double4) node_g[gOffset+0];
  double4 g1 = (double4) node_g[gOffset+1];
  double4 g2 = (double4) node_g[gOffset+2];
  double4 g3 = (double4) node_g[gOffset+3];
  double4 g4 = (double4) node_g[gOffset+4];
  double4 h = vload4(vecCounter, &node_h[offset]);
  double4 hx = vload4(vecCounter, &node_hx[offset]);
  double4 hy = vload4(vecCounter, &node_hy[offset]);
  zx = ones/pown(h,m2)*
    (g0 * hx + g1 * hy +
    half * g2 * hx * hx +
    half * g3 * hy * hy +
    g4 * hx * hy);
 double4 conc0_i = vload4(vecCounter, conc0_temp);
 double4 Dc_i = vload4(vecCounter, Dc_temp);
 double4 Fi0_i = vload4(vecCounter, Fi0_temp);
 d1xCj += dot(zx*conc0_i, ones);
 d1xDcj += dot(zx*Dc_i, ones);
 d1xFj += dot(zx*Fi0_i, ones);
 /.../
 ++vecCounter;
}
/.../
```

Listing 5: SIMD processing within kernel K_2 using OpenCL vector data types

The parallelization of computations within both CPUs and GPUs is based on the creation of a 1D space of work-items. Each work-item is responsible for calculating new values of variables in a grid node. The index of the grid node corresponding to a given work-item is determined using `get_global_id()` routine, which returns the unique global ID of work-item in the space. Depending on the index of a grid node, different application kernels are executed. While the kernels K_1 and K_3 are executed for the boundary nodes of the grid, kernels K_2 and K_4 are performed for the internal nodes.

When considering the distribution of computations across the available cores of CPU and GPU devices, the important difference between them is the way of selecting the optimal size of work-groups. For GPUs, the size of work-groups is determined as multiple of the warp size (32 for GPUs of K80 accelerators). At the same time, the size of work-groups for CPUs is chosen experimentally.

In the basic version of the solidification application (see Listing 1), each time step of the simulation is completed by swapping arrays using pointers. In the OpenCL code, swapping operation is performed outside the kernel on the host side. In practice, swapping of arrays is performed after each time step by changing arguments passed to the OpenCL kernel. An example of swapping arrays using the arguments of the OpenCL kernel is presented in Listing 6. In this example, the kernel perform computations using buffers `bufferA` and `bufferB` which are swapped after each time step with `setArg()` routine. The code from Listing 6 is an example of using the OpenCL C++ wrapper[49] to simplify the process of developing OpenCL applications. While built on top of the OpenCL C API, it corresponds closely to the underlying API and introduces no additional execution overhead. This wrapper makes the application code more readable and easier to understand, as well as more resistant to programmers faults such as memory leaks.

```
cl::Buffer bufferA = cl::Buffer(...);
cl::Buffer bufferB = cl::Buffer(...);
cl::Kernel kernel(...);
// The first argument refers to input buffer
kernel.setArg(0, bufferA);
// The second one corresponds to output buffer
kernel.setArg(1, bufferB);
cl::CommandQueue queue(...);
/.../
for(int ts=0; ts<timeSteps; ++ts) {
  queue.enqueueNDRangeKernel(kernel, ...);
  queue.finish();
  if(ts+1
    kernel.setArg(0, bufferB);
    kernel.setArg(1, bufferA);
  }
  else {
    kernel.setArg(0, bufferA);
    kernel.setArg(1, bufferB);
  }
}
```

Listing 6: Swapping arrays using kernel arguments

6.3 | Management of memory resources

The appropriate management of memory resources is vital for the efficient adaptation of the studied application to a heterogeneous platform with GPUs. In fact, an adequate initialization of the OpenCL memory buffers allows improving the performance of data transfers. OpenCL provides several flags that permit setting properties for buffer allocation and usage. In our code, OpenCL buffers are created with `CL_MEM_USE_HOST_PTR` flag, which indicates that the OpenCL runtime must not allocate memory for a buffer and instead uses the provided host pointer directly. Such a solution allows avoiding implicit copying of data between the host memory and OpenCL memory buffer. Our benchmarks show that this approach permits achieving over 11 GB/s of data transfers between CPUs and GPUs via the PCI Express 3.0 bus. Listing 7 presents an example of the buffer created with `CL_MEM_USE_HOST_PTR` flag. In this example, the memory region for one thousand integer values is allocated in the heap area. Then, an OpenCL buffer is created using `CL_MEM_READ_WRITE` and `CL_MEM_USE_HOST_PTR` flags, where `ptr` points to the address of the heap. As a result, the buffer `buffer` is created in the read-write access mode, with the underlying memory at address `ptr`.

```
int size = 1000*sizeof(int);
int *ptr = (int*) malloc(size);
cl::Buffer buffer(ctx,
        CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
        size, ptr);
```

Listing 7: Example of initialization of OpenCL buffer

Another vital issue is adopting a suitable scheme of data distribution across compute devices, beginning with load balancing. Usually, the redistribution of computations between CPUs and GPUs requires reallocation of the device memory since the size of subdomains assigned to devices is changing. In our approach, each compute device contains a separate copy of all input data. As a result, the reallocation of memory buffers is not required during the redistribution of computations. It is enough to update the values of variables pointing to the beginning and end of grid subdomains, and exchange only small amounts of data corresponding to borders of subdomains. This solution allows reducing the performance overheads generated by the device memory reallocation at the cost of extra memory space of devices.

6.4 | Synchronization of queues

The studied application enforces some dependencies between command queues. These dependencies are a consequence of data transfers of *halo* regions between compute devices in successive time steps of the simulation. Providing an efficient synchronization of queues is essential for the overall performance and correctness of the simulation. The OpenCL framework offers two scenarios for achieving this goal. The first one is based on the OpenCL completion events, when the synchronization is defined using *event wait*list during inserting commands into queues. This list contains events that must signal the completion status before the start of command execution. The second scenario (called the active synchronization) engages the host in queue synchronization. Here, the host inserts commands into queues, then waits for their completion before submitting the next commands. In our case, the second scenario is selected as more reliable to ensure the synchronization of devices (see Figure 7). In practice, all compute devices are synchronized with the host after each time step of the simulation.

7 | BENCHMARKING OPENCL HETEROGENEOUS PROGRAMMING FOR NUMERICAL MODELING OF SOLIDIFICATION

7.1 | Performance results for the first platform

In this subsection, we present the performance results for the OpenCL code of the solidification modeling application, developed, using the approach described in the two previous sections. All experiments are performed on the CPU–GPU platform presented in Table 1. The developed code uses the double precision floating-point format and is compiled using the GNU C++ compiler (ver. 8.3.0) with Ubuntu 19.04. For the CPU side, the compiler is empowered by Intel SDK for OpenCL Application 2019,[50] which implements the OpenCL 2.1 standard. In addition, CUDA 10.1 Toolkit[51] provides the implementation of OpenCL in version 1.2 for NVIDIA GPUs. The benchmarks are executed for the two-dimensional grid consisting of 4,000,000 nodes (2000 nodes along each dimension) and 50,000 time steps.

The proposed concept of workload distribution (see Figure 7) permits the flexible utilization of different configurations of computing resources to execute the simulation. In the tests, the following configurations are investigated:

- configuration **A** with two Intel Xeon CPUs (18 cores per CPU);
- four configurations **B**, **C**, **D**, and **E** that use different number p of GPU processors, where $p \in \{1, 2, 3, 4\}$. This version also employs a single CPU core to run the OpenCL host process,
- heterogeneous configuration **F** with two Intel Xeon CPUs and a single NVIDIA Tesla K80 accelerator equipped with two GPU processors, where 69.7% of the grid is uniformly assigned to GPU processors, while the rest of the grid is allocated to CPUs;
- heterogeneous configuration **G**, which uses all available resources of the platform, including two CPUs and two K80 cards (four GPU processors). In this case, 82.2% of the grid is uniformly allocated to both accelerators.

For optimizing the performance of computations, the different sizes of work-groups for CPUs and GPUs are evaluated. While in the case of GPUs the best performance corresponds to the work-group size equal to 32 (warp size), for the CPUs the best results are achieved for work-groups of size 64 each.

Table 3 shows the total execution time obtained for all configurations of computing resources. For ensuring the reliability of experimental results, all measurements are repeated $r = 10$ times, and the median values are finally calculated. All tests have been repeated after some time to verify the reproducibility of results. What is important, the test platforms are located in the professional data center with stable environmental conditions.[52] Besides the execution time, the table also presents the following speedups:

- speedup S_1 obtained against the configuration **A** with two CPUs;
- speedup S_2 achieved against the configuration **B**, which uses a single GPU processor. This speedup is determined only for the configurations **C**,**D**, and **E**, to characterize the scalability of expanding the number of GPU processors.

TABLE 3 First platform: Performance results achieved for different configurations of computing resources

Configuration ID	Configuration of computing resources	Execution time (s)	Speedup S_1	Speedup S_2
A	2 × CPU	2440	–	–
B	1 × GPUP (+ 1 × CPU core)	2117	1.15	–
C	2 × GPUP (+ 1 × CPU core)	1101	2.22	1.93
D	3 × GPUP (+ 1 × CPU core)	765	3.19	2.76
E	4 × GPUP (+ 1 × CPU core)	571	4.27	3.71
F	2 × CPU + 1 × K80	779	3.13	–
G	2 × CPU + 2 × K80	491	4.97	–

Note: GPUP refers to one GPU processor of K80 accelerator.

The analysis of Table 3 permits us to conclude that the usage of merely a single GPU processor allows accelerating computations 1.15 times against the configuration with two CPUs. By incorporating both GPU processors of the K80 accelerator, the simulation is performed 2.22 times faster. Finally, employing all resources of two GPU accelerators (four GPU processors) permit us to speed up computations 4.27 times, so the simulation takes now 571 s, while for the two CPUs, it takes 2440 s. Regarding the scalability of using GPU resources (see the last column of Table 3), it can be concluded that four GPU processors perform computations 3.71 and 1.93 times faster than a single GPU processor and two GPUs processors, respectively. These figures confirm the satisfactory scalability of using GPU resources by the OpenCL code.

The primary benefit of OpenCL is the possibility to use the unified programming model for CPU and GPU devices. The last two rows of Table 3 show the performance of the OpenCL heterogeneous application for the solidification modeling. As we can see, the heterogeneous configuration with two CPUs and a single K80 card allows us to perform the simulation 3.13 times faster than two CPUs. But more importantly, the performance gain achieved in this case against a single K80 card is equal to 1.41 times. These figures support the idea of using the heterogeneous programming approach for the studied application. At the same time, the efficiency of this approach for the most powerful heterogeneous configuration with two CPUs and two K80 cards is considerably lower. In this case, the performance gain obtained against two K80 accelerators is equal only to 1.15 times.

7.2 | Performance comparison of OpenCL versus OpenMP

It is of considerable significance to compare the performance of the developed OpenCL code against the OpenMP application already implemented by us for CPUs.[19,44] For this purpose, the OpenMP code corresponding to Listing 1 is compiled using the Intel `icpc` compiler (ver. 19.0.5) with the optimization flag `-O3`. As a result, the usage of OpenMP for the configuration with two Haswell-based CPUs allows executing the simulation 1.28 times faster as compared with OpenCL.

One of the reasons for this advantage of OpenMP is undoubtedly the usage of the Intel compiler, while the GNU C++ compiler is applied for the OpenCL code. More objective results could be obtained using the same GNU C++ compiler for both codes. However, the GNU compiler gives disastrous performance results for the OpenMP code (more than five times worse than the Intel compiler).

7.3 | Benchmarking the second platform

Table 4 presents the performance results achieved for various configurations of computing resources of the second platform which contains the powerful NVIDIA Tesla V100 (see Table 2). The OpenCL codes are compiled using the GNU C++ compiler (version 9.2.0) with CentOS 7. For CPUs, the compiler is supplemented by Intel SDK for OpenCL Application 2019, implementing OpenCL in version 2.1, while CUDA 10.2 Toolkit provides the implementation of OpenCL 1.2 for GPU.

The first and second rows of Table 4 show the execution times achieved on two CPUs for, respectively, the OpenMP version (with `icpc` compiler ver. 19.0.5) and the OpenCL code denoted as **A1**. The conclusion is that the OpenMP version is 1.33 times faster than the OpenCL code **A1**, which is developed using the SIMDization approach described in Section 5.2 (see listing 5. Fortunately, we discover that in the case of Cascade Lake CPUs, it is enough to switch off this SIMDization procedure and use the automatic vectorization provided by the compiler. The resulting OpenCL code denoted as **A2** in Table 4 allows speeding up the execution 1.19 times compared with the version **A1**. In consequence, the OpenMP code is only 1.11 times faster than the OpenCL version **A2**.

The V100 GPU is so powerful that it permits accelerating the computation about 3.5 times against K80. Thus, although the performance of two Intel Xeon Gold 6240 is comparable to the performance of K80, the heterogeneous configuration C with two CPUs and a single V100 GPU

TABLE 4 Second platform: Performance results achieved for different configurations of computing resources

Configuration ID	Configuration of computing resources	Execution time (s)	Speedup S
OpenMP	2 × CPU	1080	–
A1	2 × CPU	1432	–
A2	2 × CPU	1201	–
B	1 × V100 (+ 1 × CPU core)	316	3.80
C	2 × CPU + 1 × V100	280	4.29

Note: Besides the executions time, the speedup *S* achieved against the configuration **A2** is shown.

allows us to execute the simulation only 1.13 times faster than a single V100. This performance benefit is significantly smaller than in the case of the first platform. At the same time, one should take into account that Intel Xeon Gold 6240 is clearly not the top-of-the-line CPU, which is Intel Xeon E5-2699 v3 in the first platform.

In fact, for the first platform (Table 1), the ratio *R* of theoretical peak performances between a single GPU card and two CPUs can be expressed as $R = 2.912/1.497 = 1.95$. For the second server (Table 2), the ratio *R* is much high since now $R = 7.8/2.88 = 2.7$. However, if we replace Gold processors with Intel Xeon Platinum 8280 CPUs[53] (each with 28 cores) as the top-of-the-line Intel Xeon processors with the Cascade Lake architecture, then the ratio $R = 7.8/4.3 = 1.81$ becomes even lower than for the first platform. Despite these estimations' rough character, we can expect a quite reasonable performance benefit for the heterogeneous configuration with a single V100 and two Intel Xeon Platinum 8280 CPUs. At the same time, the expected profit could be reduced to some extent by increasing the contribution of communication overheads to the total execution time. For the heterogeneous configuration C of the second server, this contribution rises to about 9%, while for the counterpart F from Table 3, the contribution does not exceed 5%.

8 | CORRELATION OF PERFORMANCE AND ENERGY CONSUMPTION

Besides the performance, this work explores the energy efficiency of computations as well. To measure energy consumption and power, we use the hardware-based technique[8,54] that assumes the utilization of an external AC power meter to monitor the energy consumption and power for the whole computing system. The AC power meter is a device that passes power to the server under the load and measures the power and energy consumption in real-time. While this technique does not provide the possibility to analyze energy and power consumption for independent components of computing platforms, its main advantage is high accuracy.[54]

Figure 8 presents the scheme used in this work for measuring energy and power consumption. For this purpose, the Yokogawa WT310 power meter[55] is installed between the wall AC outlets and the input power sockets of the server. The power meter is equipped with the USB interface that allows us to access energy and power consumption data, and provides management of the measurement process. The YokoTool command-line tool is used to get these data.[56] This tool is written in Python and operates via the serial USB interface. The tool is nonintrusive and does not have any noticeable influence on measurements.

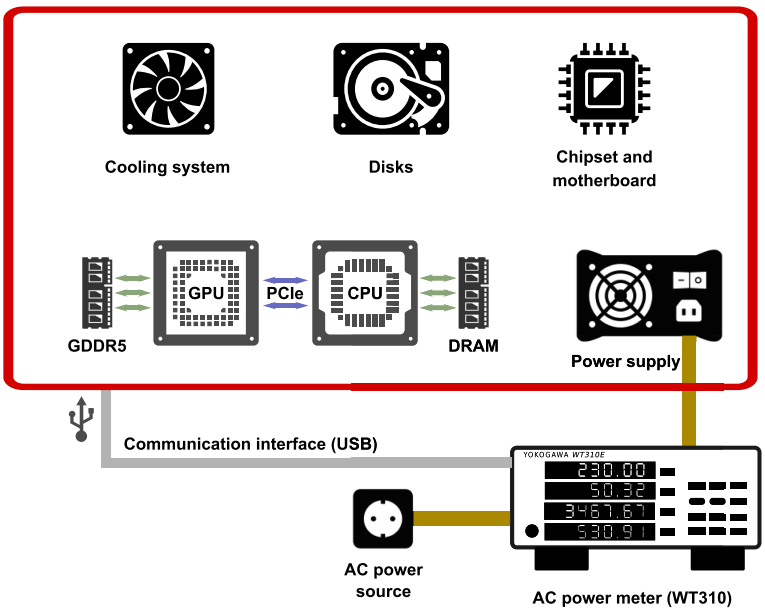
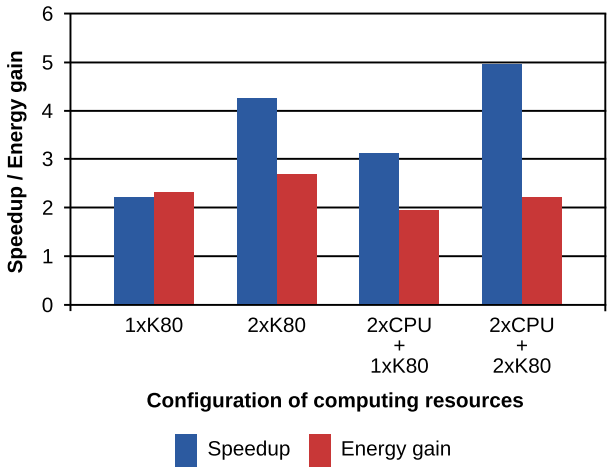


FIGURE 8 Hardware-based technique used for energy and power measurements

TABLE 5 First platform: Performance and energy/power comparison for different configuration of computing resources

Configuration ID	Configuration of computing resources	Execution time (s)	Speedup	Energy consumption (kJ)	Energy gain	Average power (W)
A	2 × CPU	2440	–	1208	–	495
C	1 × K80 (+ 1 × CPU core)	1101	2.22	520	2.32	472
E	2 × K80 (+ 1 × CPU core)	571	4.27	447	2.70	784
F	2 × CPU + 1 × K80	779	3.13	618	1.96	792
G	2 × CPU + 2 × K80	491	4.97	542	2.23	1095

FIGURE 9 First platform: comparison of performance and energy gains achieved against configuration with two Intel Xeon CPUs



8.1 | Experiments with the first platform

For the first server, we investigate the correlation of performance and energy/power efficiency for five out of seven configurations of computing resources, namely, **A**, **C**, **E**, **F**, and **G** (Table 5). For ensuring the correctness of energy and power measurements, the computing devices that are unnecessary for a given configuration are removed from the platform physically. For instance, in the case of the configuration **A**that uses only CPUs, all K80 accelerators are unplugged. Analogously, for the heterogeneous configuration **F** with a single accelerator, only one K80 card is connected to the platform during the measurement.

The comparison of performance and energy/power consumption is presented in Table 5, for different configurations of computing resources. Besides the execution time, the table shows the total energy and average power consumed by all configurations during the application execution. This table also reports the speedup and energy gain achieved against the configuration **A** with two CPUs. The speedup and energy gain are also presented graphically in Figure 9.

The analysis of Table 5 and Figure 9 shows that for the configuration **C** with a single K80 card the values of the speedup and energy gain are similar. In fact, the usage of a single GPU accelerator allows reducing the execution time and energy consumption 2.22 and 2.32 times against two CPUs, respectively, with practically the same average power. Employing two K80 devices (configuration **E**) permits improving the performance of simulation 4.27 times against two CPUs. At the same time, the total energy consumption is reduced only 2.70 times. The above discrepancy is the result of increasing the average power 1.58 times. These results mean that using two K80 devices (configuration **E**) allows increasing the performance significantly against the configuration **C** with a single accelerator (1.92 times), while the energy gain achieved in this case is rather low (only 1.16 times).

As was mentioned in the previous section, the heterogeneous configurations **F** with two CPUs and a single K80 device allows increasing the performance 1.41 times against using a single K80. However, it is accompanied by 1.67 times higher average power that results in increasing the energy consumption 1.19 times. The most powerful heterogeneous configuration **G** allows further decreasing the execution time (1.58 times against the configuration **F**) and reducing the energy consumption 1.14 times. However, this advantage is burdened with increasing the average power 1.38 times.

8.2 | Benchmarking the second platform

The results achieved for the second server (Table 6) confirms the conclusions about the higher energy efficiency of using V100 GPU to execute the simulation compared with the implementation on CPUs (version **A2** of OpenCL code) as well as on the heterogeneous CPU–GPU configuration **C**.

Configuration ID	Configuration of computing resources	Execution time (s)	Speedup	Energy consumption (kJ)	Energy gain	Average power (W)
A2	2 × CPU	1201	–	564	–	470
B	1 × V100 (+ 1 × CPU core)	316	3.80	137	4.12	433
C	2 × CPU + 1 × V100	280	4.29	185	3.05	662

TABLE 6 Second platform: Performance and energy/power comparison for different configuration of computing resources

In particular, the usage of a single V100 allows reducing the energy consumption more than four times. Furthermore, the average power required by the heterogeneous configuration **C** is about 1.5 times higher than for a single V100 card that increases the energy consumption by the factor of 1.35.

9 | CONCLUSIONS AND FUTURE WORKS

This article explores the OpenCL heterogeneous programming model for porting applications to CPU–GPU platforms, with a real-life application for the phase-field modeling of alloy solidification. The aim is to achieve a flexible workload distribution between available CPU–GPU resources and optimize application performance. Considering the solidification modeling application as a use case, we explore the basic steps required for (i) adaptation of an application to CPU–GPU platforms, and (ii) mapping the application workload onto the OpenCL model. The adaptation is based on a reformulation of steps developed previously for CPU–MIC architectures. They can be successfully adjusted to heterogeneous platforms with GPU devices in a relatively straightforward way. The mapping process allows us to utilize the capabilities of OpenCL heterogeneous programming for harnessing CPU and GPU cores using data parallelism, as well as for the management of available compute devices with task parallelism. Among important issues that are solved at this stage are: (i) mapping command queues onto devices, (ii) implementing the kernel on CPUs and GPUs addressing possible differences in vectorization, (iii) management of memory resources, and (iv) synchronization of queues.

The resulting code's performance and energy efficiency are experimentally studied for various configurations of two computing platforms with powerful GPUs of various generations. The first one is equipped with two Haswell-based Intel Xeon E5-2699 v3 CPUs (18 cores each) and a single NVIDIA K80 card with the Kepler architecture. The experimental evaluation confirms the performance advantage of using computing resources of both GPUs and CPUs. For example, the configuration with two CPUs and a single K80 card permits accelerating the application 1.41 times against a single K80.

At the same time, the achieved benefit depends strongly on the relationship between the computing power of CPU and GPU devices. The V100 GPU used in the second platform is so powerful that it allows accelerating the computation about 3.5 times against the K80 card. As a result, although the peak performance of two Intel Xeon Gold 6240 CPUs (with Cascade Lake architecture) is comparable to the performance of K80, the heterogeneous configuration with two CPUs and a single GPU allows us to execute the simulation only 1.13 times faster than a single GPU. This benefit is significantly smaller than in the case of the first platform. At the same time, one should take into account that Xeon Gold 6240 is not the top-of-the-line CPU, as is the Intel processor in the first platform. If we replace Gold processors with Intel Xeon Platinum 8280 CPUs (each with 28 cores) as the top processors with the Cascade Lake architecture, we can again expect a quite reasonable performance benefit for the heterogeneous configuration.

Besides the performance of the application, we investigate its energy efficiency as well and use the hardware measurement technique. The usage of the Yokogawa WT310 digital power meter allows us to perform reliable measurements of energy and power. The experiments performed for both platforms permit us to discover that performance gain achieved by utilizing both GPU and CPU cores entails the growth of the average power that increases energy consumption. For example, for the configuration with two Haswell-based CPUs and a single K80 card, the speedup of 1.41 times is achieved at the cost of 1.67 times higher average power that increases the energy consumption 1.19 times against a single K80.

This article also presents the performance comparison of OpenCL and OpenMP versions of the application implemented on CPUs only. For the first platform with Haswell-based processors, the usage of OpenMP allows executing the simulation at a rate of 1.28 times faster than the OpenCL code that implements the explicit SIMDization. We discover that using the automatic vectorization instead of the explicit SIMDization allows us to obtain the OpenCL code that is only 1.11 times slower than the OpenMP version on the other platform based on Cascade Lake CPUs. Among possible reasons for this advantage of OpenMP is the usage of the Intel compiler, while the GNU C++ compiler is applied for OpenCL.

The performance and energy efficiency results achieved in this work provide the basis for further development and optimization of the OpenCL code of the solidification modeling application. The primary direction of our future work is to better utilize the memory hierarchy of GPUs and CPUs. We also plan to study the possibility of accelerating the application by exploiting the newest AMD architectures, including EPYC Rome CPUs[57] and Radeon Instinct GPUs.[58] Another topic of future works is exploring CPU[59] and GPU[60] frequency scaling as a tool to optimize the energy efficiency of our application. The method of frequency scaling is known as an efficient technique[61] for energy savings in memory-bound applications when processor cycles are being wasted as they are stalled on memory.

ACKNOWLEDGMENTS

This research was supported by the National Science Centre (Poland) under grant no. UMO-2017/26/D/ST6/00687 and by the project no. 020/RID/2018/19 financed within the program of the Polish Ministry of Science and Higher Education "Regional Initiative of Excellence" (years 2019–2022, the amount of financing 12,000,000 PLN). The authors are grateful to Czestochowa University of Technology for granting access to HPC platforms provided by the MICLAB project no. POIG.02.03.00.24-093/13.

ORCID

Kamil Halbiniak https://orcid.org/0000-0001-9116-8981
Lukasz Szustak https://orcid.org/0000-0001-7429-6981

REFERENCES

1. Newburn CJ. Heterogeneous streaming. Paper presented at: Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IEEE Computer Society, Chicago, IL; 2016.
2. Rojek K, Wyrzykowski R. Performance modeling of 3D MPDATA simulations on GPU cluster. *J Supercomput*. 2017;73:664-675.
3. Rico-Gallego J, Lastovetsky A, Diaz-Martin J. Model-based estimation of the communication cost of hybrid data-parallel applications on heterogeneous clusters. *IEEE Trans Parall Distrib Syst*. 2017;28(11):3215-3228.
4. Chen Y, Yang T, Emer J, Sze V. Eyeries v2: a flexible accelerator for emerging deep neural networks on mobile devices. *IEEE J Emerg Select Topocs Circ Syst*. 2019;9(2):292-308.
5. Liu Y, Huang L, Wu M, et al. PPOpenCL: a performance-portable opencl compiler with host and kernel thread code fusion. Paper presented at: Proceedings of the 28th International Conference on Compiler Construction – CC 2019, Washington, DC; 2019:2-16.
6. Lastovetsky A, Szustak L, Wyrzykowski R. Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing. *IEEE Trans Parall Distribut Syst*. 2017;28(3):787-797.
7. Laccetti G, Lapegna M, Mele V, Romano D. A study on adaptive algorithms for numerical quadrature on heterogeneous GPU and multicore based systems. Paper presented at: Proceedings of the 10th International Conference Parallel Processing and Applied Mathematics - PPAM 2013, Warsaw, Poland. 2014:8384;704-713.
8. Jeffers J, Reinders J, Sodani A. *Intel Xeon Phi Processor High Performance Programming*. San Francisco, CA: Knights Landing Edition, Morgan Kaufmann; 2016.
9. Szustak L, Halbiniak K, Kuczynski L, Wrobel J, Kulawik A. Porting and optimization of solidification application for CPU-MIC hybrid platforms. *Int J High Perform Comput Appl*. 2018;32(4):523-539.
10. CUDA C++ programming guide; 2019. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
11. OpenMP Application programming interface, version 5.0; 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. November.
12. Khronos Group OpenCL overview; 2020. https://www.khronos.org/opencl. Accessed January 2020.
13. Ali A, Dastgeer U, Kessler C. OpenCL for programming shared memory multicore CPUs. Paper presented at: Proceedings of the 5th Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2012) at HiPEAC-2012, Paris, France; 2012.
14. Kruzel F, Banas K. Vectorized OpenCL implementation of numerical integration for higher order finite elements. *Comput Math Appl*. 2013;66(10):2030-2044.
15. Minhas UI, Woods R, Karakonstantis G. Exploring functional acceleration of OpenCL on FPGAs and GPUs through platform-independent optimizations. In: Voros N, Huebner M, Keramidas G, Goehringer D, Antonopoulos C, Diniz PC, eds. *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Vol 10824. Berlin, Germany: Springer-Verlag; 2018:551-563.
16. Szustak L, Halbiniak K, Kulawik A, Wrobel J, Gepner P. Toward parallel modeling of solidification based on the generalized finite difference method using Intel Xeon Phi. Paper presented at: Proceedings of the 11th International Conference Parallel Processing and Applied Mathematics - PPAM 2015, Krakow, Poland; vol. 9573, 2016:411-412.
17. Halbiniak K, Szustak L, Lastovetsky A, Wyrzykowski R. Exploring OpenMP Accelerator Model in a real-life scientific application using hybrid CPU-MIC platforms. Paper presented at: Proceedings of the 3rd International Workshop on Sustainable Ultrascale Computing Systems – NESUS, Sofia, Bulgaria; 2016:11-14.
18. 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. Paper presented at: Proceedings of the International Conference Algorithms and Architectures for Parallel Processing - ICA3PP 2016, Lecture Notes in Computer Science, Granada, Spain; 2016;10049:373-382.
19. Halbiniak K, Wyrzykowski R, Szustak L, Olas T. Assessment of offload-based programming environments for hybrid CPU-MIC platforms in numerical modeling of solidification. *Simulat Modell Pract Theory*. 2018;87:48-72.
20. Liu Y, Yang C, Liu F, et al. 623 Tflop/s HPCG run on Tianhe-2: leveraging millions of hybrid cores. *Int J High Perform Comput Appl*. 2016;30(1):39-54.
21. Arcucci R, D'Amore L, Carracciuolo L, Scotti G, Laccetti G. A decomposition of the Tikhonov regularization functional oriented to exploit hybrid multilevel parallelism. *Int J Parall Program*. 2017;45(5):1214-1235.
22. Wyrzykowski R, Szustak L, Rojek K. Parallelization of 2D MPDATA EULAG algorithm on hybrid architectures with GPU accelerators. *Parallel Comput*. 2014;40(8):425-447.
23. Wolfe N, Liu T, Carothers C, Xu XG. Heterogeneous concurrent execution of Monte Carlo photon transport on CPU, GPU and MIC. Paper presented at: Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms. New Orleans, LA: IEEE Press; 2014:49-52.
24. 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. Paper presented at: Proceedings of the 28th International Supercomputing Conference, ISC 2013, Lecture Notes Computer Science, Leipzig, Germany. 2013;7509:239-254.

25. T. Shimokawabe, T. Aoki, T. Takaki, et al., Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. Paper presented at: Proceedings of the 2011 ACM/IEEE International Conference High Performance Computing, Networking, Storage and Analysis – SC'11, Washington, DC; 2011.
26. Wyrzykowski R, Szustak L, Rojek K, Tomas A. Towards efficient decomposition and parallelization of MPDATA on hybrid CPU-GPU cluster. Paper presented at: Proceedings of the International Conference on Large-Scale Scientific Computing – LSCC 2013, Lecture Notes in Computer Science, Sozopol, Bulgaria. 2014;8353:457-464.
27. K. Banas, F. Kruzel, OpenCL performance portability for Xeon phi coprocessor and NVIDIA GPUs: a case study of finite element numerical integration. Paper presented at: Proceedings of the Euro-Par 2014: Parallel Processing Workshops, Lecture Notes in Computer Science, Porto, Portugal. 2014;8806:158-169.
28. Chang L, Gómez-Luna J, El Hajj I, Huang S, Chen D, Hwu W, Collaborative computing for heterogeneous integrated systems. Paper presented at: Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering, L'Aquila, Italy; 2017:385-388.
29. D'Amore L, Mele V, Romano D, Laccetti G. Multilevel algebraic approach for performance analysis of parallel algorithm. *Comput Inform*. 2019;38(4):817-850.
30. Mele V, Romano D, Constantinescu EM, Carracciuolo L, D'Amore L. Performance evaluation for a PETSc parallel-in-time solver based on the MGRIT algorithm. Paper presented at: Proceedings of the Euro-Par 2018: Parallel Processing Workshops, Lecture Notes in Computer Science, Turin, Italy. 2019;11339:716-728.
31. Rojek K, Szustak L, Wyrzykowski R. Performance analysis for stencil-based 3D MPDATA algorithm on GPU architecture. Paper presented at: Proceedings of the 10th International Conference Parallel Processing and Applied Mathematics - PPAM 2015, Lecture Notes in Computer Science, Krakow, Poland. 2014;8384:145-154.
32. D'Amore L, Mele V, Laccetti G, Murli A. Mathematical approach to the performance evaluation of matrix multiply algorithm. Paper presented at: Proceedings of the 11th International Conference Parallel Processing and Applied Mathematics - PPAM 2015, Lecture Notes in Computer Science, Krakow, Poland. 2016;9574:25-34.
33. Lee J, Samadi M, Park Y, Mahlke S. SKMD. *ACM Transactions on Computer Systems*. 2015;33 (3):1–27. http://dx.doi.org/10.1145/2798725.
34. Hötzer J, Jainta M, Steinmetz P, et al. Large scale phase-field simulations of directional ternary eutectic solidification. *Acta Materialia*. 2015;93:194-204.
35. M. Bauer, J. Hötzer, D. Ernst, et al. Code generation for massively parallel phase-field simulations. Paper presented at: Proceedings of the 2019 International Conference High Performance Computing, Networking, Storage and Analysis – SC'19, Denver; 2019.
36. Trobec R, Korosec G. *Parallel Scientific Computing: Theory, Algorithms, and Applications of Mesh Based and Meshless Methods*. New York, NY: Springer; 2015.
37. Meneses E, Sarood O, Kalé L. Energy profile of rollback-recovery strategies in high performance computing. *Parallel Comput*. 2014;40(9):536-547.
38. Montella R, Kosta S, Oro D, et al. Accelerating Linux and Android applications on low-power devices through remote GPGPU offloading. *Concurr Comput*. 2017;29(24):e4286.
39. Warren J, Boettinger W. Prediction of dendritic growth and microsegregation patterns in a binary alloy using the phase-field method. *Acta Metallurgica et Materialia*. 1995;43(2):689-703.
40. Adrian H, Spiradek-Hahn K. The simulation of dendritic growth in Ni-Cu alloy using the phase field model. *Arch Mater Sci Eng*. 2009;40(2):89-93.
41. Takaki T. Phase-field modeling and simulations of dendrite growth. *ISIJ Int*. 2014;54(2):437-444.
42. Kulawik A. *The modeling of the phenomena of the heat treatment of the medium carbon steel*. Wydawnictwo Politechniki Czestochowskiej Monographs (in Polish); Czestochowa, Poland. 2013;281:158.
43. Benito J, Ureñ F, Gavete L. The generalized finite difference method. In: Àlvarez MP, ed. *Leading-Edge Applied Mathematical Modeling Research*. New York, NY: Nova Science Publishers; 2008:251-293.
44. Halbiniak K, Szustak L, Kulawik A, Gepner P. Performance optimizations for parallel modeling of solidification with dynamic intensity of computations. Paper presented at: Proceedings of the 13th International Conference Parallel Processing and Applied Mathematics - PPAM 2019, Lecture Notes in Computer Science, Bialystok, Poland. 2020;12043:370-381.
45. Harris M. Performance portability from GPUs to CPUs with OpenACC; 2015. https://devblogs.nvidia.com/performance-portability-gpus-cpus-openacc. Accessed September 2019.
46. OpenACC specification; 2019. https://www.openacc.org/specification. Accessed September 2019.
47. Munshi A, Gaster B, Mattson TG, Fung J, Ginsburg D. *OpenCL Programming Guide*. Boston, MA: Pearson Education; 2012.
48. Vectorization: SIMD processing within a work-group. https://software.intel.com/en-us/iocl -tec-opg-vectorization-simd-processing-within-a-work-group. October 30, 2018.
49. OpenCL *C++ Wrapper API, Version: 1.2.6*. Beaverton, OR: Khronos Group; 2012. https://www.khronos.org/files/OpenCLPP12-reference-card.pdf.
50. Intel SDK for OpenCL Applications. https://software.intel.com/en-us/opencl-sdk. accessed February 2020.
51. NVIDIA CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. accessed February 2020
52. MICLAB: pilot laboratory of manycore systems; 2015. http://miclab.pl. Accessed October 2019.
53. Xeon platinum 8280 – intel; 2019. https://en.wikichip.org/wiki/intel/xeon_platinum/8280. accessed April, 2019.
54. Fahad M, Shahid A, Manumachu RR, Lastovetsky A. A Comparative Study of Methods for Measurement of Energy of Computing. *Energies*. 2019;12(11):2204. http://dx.doi.org/10.3390/en12112204.
55. WT300 Series digital power meter analyzer; 2018. https://tmi.yokogawa.com. Accessed December 2019.
56. Yoko tool; 2020. https://01.org/yoko-tool. Accessed Febuary 2020.
57. AMD EPYC 7002 series processors; 2020. https://www.amd.com/en/processors/epyc-7002-series. Accessed March 2020.
58. AMD Radeon instinct accelerators; 2020. https://www.amd.com/en/graphics/servers-radeon-instinct-mi. Accessed March 2020.
59. Haj-Yahya J, Mendelson A, Ben Asher Y, Chattopadhyay A. *Energy Efficient High Performance Processors: Recent Approaches for Designing Green High Performance Computing*. New York, NY: Springer; 2018.

60. Tang Z, Wang Y, Wang Q, Chu X. The impact of GPU DVFS on the energy and performance of deep learning: an empirical study; 2019. arXiv:1905.11012, 12 pp.
61. Rojek K, Ilic A, Wyrzykowski R, Sousa L. Energy-aware mechanism for stencil-based mpdata algorithm with constraints. *Concurr Comput Pract Exp*. 2016;29(8):e4016.

How to cite this article: Halbiniak K, Szustak L, Olas T, Wyrzykowski R, Gepner P. Exploration of OpenCL Heterogeneous Programming for Porting Solidification Modeling to CPU-GPU Platforms. *Concurrency Computat Pract Exper*. 2021;33:e6011. https://doi.org/10.1002/cpe.6011