Hernández, M.; Cebrián, JM.; Cecilia-Canales, JM.; García, JM. (2020). Offloading strategies for Stencil kernels on the KNC Xeon Phi architecture: Accuracy versus performance. International Journal of High Performance Computing Applications. 34(2):199-297. https://doi.org/10.1177/1094342017738352

Journal Title
XX(X):1–13

DOI: 10.1177/ToBeAssigned
www.sagepub.com/

# Offloading Strategies for Stencil Kernels on the KNC Xeon Phi Architecture: Accuracy vs Performance

**Mario Hernández[1], Juan M. Cebrián[2], José M. Cecilia[3], José M. García[4]**

**Abstract**

The ever-increasing computational requirements of HPC and service-provider applications are becoming a great challenge for hardware and software designers. These requirements are reaching levels where the isolated development on either computational field is not enough to deal with such challenge. A holistic view of the computational thinking is therefore the only way to success in real scenarios. However, this is not a trivial task as it requires, among others, of hardware-software co-design. In the hardware side, most high-throughput computers are designed aiming for heterogeneity, where accelerators (e.g. GPUs, FPGAs, etc) are connected through high bandwidth bus, such as PCI-Express, to the host CPUs. Applications, either via programmers, compilers or runtime, should orchestrate data movement, synchronization, etc among devices with different compute and memory capabilities. This increases the programming complexity and it may reduce the overall application performance. This paper evaluates different offloading strategies to leverage heterogeneous systems, based on several cards with the first generation Xeon Phi co-processors, (Knights Corner - KNC). We use a 11-point 3-D *Stencil* kernel that models heat dissipation as a case-study. Our results reveal substantial performance improvements when using several accelerator cards. Additionally, we show that computing of an approximate result by reducing the communication overhead can yield 23% performance gains for double-precision datasets.

**Keywords**

Offloading Computation, *Stencil* Codes, Approximate Computing, Heterogeneous Computing.

[1] Facultad de Ingeniería, Universidad Autónoma de Guerrero, México.
[2] Centro de Supercomputación de Barcelona, España.
[3] Departamento de Ingeniería Informática, Universidad Católica San Antonio de Murcia, España.
[4] Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, España.

**Corresponding author:**
Mario Hernández, Facultad de Ingeniería, Universidad Autónoma de Guerrero, México.
Email: mhernandezh@uagro.mx

*Prepared using* sagej.cls *[Version: 2015/06/09 v1.01]*

## Introduction

We are witnessing the steady transition from multicore or manycore processors to heterogeneous processors mainly motivated by the physical limitations of silicon-based architectures (Esmaeilzadeh et al. 2012). These architectures use several cores with different functionality, performance, and energy efficiency, including latency-oriented cores for control-dominated tasks and throughput-oriented cores for data-driven tasks (Wen-mei 2015). The Intel Xeon Phi family of accelerators is an example of these evolving heterogeneous architectures designed for throughput-oriented applications based on data and thread-level parallelism (Chrysos 2014).

The most versatile and immediate way to use the Intel Xeon Phi co-processor is the *native* mode of execution (Jeffers and Reinders 2013b). This mode performs a direct execution of applications in the co-processor and it proposes a competitive difference with discrete NVIDIA GPUs; market-leading in accelerator development for general purpose computations (Ujaldón 2016). However, some applications require the use of the Intel Xeon Phi as an accelerator to be able to have several cards running in parallel to improve performance or increase the total amount of memory available for the simulation. The success of this strategy will depend on the amount of communication required to exchange data between the different co-processors that collaborate in solving the problem. These communications go through the PCI-Express bus, so a high volume of data transfers between them can involve low (or even zero) *speedup*. Intel Xeon Phi, particularly Knights Corner Architecture (KNC) provides a different programming model (*offload*) to handle the co-processor as an accelerator. The application starts the execution in the host processor (CPU) and it is responsible for *offload* those parts of code that are tagged by the programmer to Intel Xeon Phi co-processor, ideally, computing intensive parts of the code with high amounts of data and thread-level parallelism.

This paper evaluates different strategies on how to manage the first generation Intel Xeon Phi architecture (KNC) co-processor in *offload* mode to improve the performance of *Stencil* codes that do not fit in the memory of a single co-processor card. The analyzed Intel Xeon Phi co-processors are based on the *Knights Corner* architecture. Our case study performs several tests on a 3-D heat diffusion *Stencil* with 11 points. Specifically, major contributions of this article include:

- It describes the *offload* operation mode, providing guidelines to apply it to *Stencil* codes. The Xeon Phi management in *offload* mode is addressed using Intel's LEO (*Language Extension Offload*).
- It evaluates the Intel Xeon Phi as a *co-processor*. We focus on evaluating the performance of Xeon Phi when working in *offload* mode when multiple cards are used (two cards in our case).
  - It studies various forms of management accelerator cards by the host processor, evaluating the advantages and disadvantages of each one.
  - It considers how to split the data across multiple cards. This task is not trivial because there are data dependencies that make it difficult to divide the input between the different memory spaces, forcing high data communication between cards.
  - It assesses the impact on performance due to communications overhead between cards, and studies how to improve it by reducing the accuracy of the solution.
- Results show substantial performance improvements when using several accelerator cards, but far from linear scaling. By reducing the communication overhead, performance improves by an additional 20 to 23% (but computations produce an approximate result).

## Background

### *Stencil Patterns*

Partial differential equations are the base of many applications in a wide variety of fields of science and engineering. These equations can be solved by the finite difference method, which calculates an approximate solution to the problem in an iterative way. Frequently, these operations involve access to large volumes of data. These patterns are known as *Stencil* kernels. *Stencils* are computed by updating each of the input elements with correctly weighted values of neighboring elements. The coefficients used to weight the values of the different neighboring elements vary depending on the problem. Computations on the data matrix are applied for a finite number of times (or time units), either meeting a convergence criteria, or with the intention to analyze the state of the array after a certain number of steps.

In many cases, the performance of *Stencil* kernels is limited by the memory bandwidth of the evaluation platform. Such impact will depend on the number of neighbors considered in each step as well as by the dimensionality of the problem (Reinders and Jeffers 2014). 3-D *Stencil* algorithms are implemented as a triple nested loop that runs along the entire data structure, while each point of the *grid* is updated. The calculation of each output element usually requires: a) the weighted contribution of some close neighbors in each direction, defined by the nature of the problem, b) the value of the current element in previous time instants (*t-n*), and c) a unique corresponding point on different input matrices. Depending on the time order of the code, the algorithm can use two or three copies of the data (in time steps $t$ and $t + 1$, and sometimes *t-1*), exchanging their roles as source and destination in alternate time steps.

There is an additional outermost loop that simulates the execution time steps. After each iteration, the convergence condition for the values of the solution matrix is checked, and when this condition is satisfied the execution finishes. Checking the convergence criteria on each iteration is costly when working with a parallel version of this algorithm, since it becomes a serialization point. A common optimization is to estimate, *a priori*, the number of time steps required to converge to the desired solution. Subsequently, each new execution will always run with a fixed number of time iterations equal to that obtained *off-line*. This is the procedure we followed when evaluating the algorithms presented in this paper.

### *Intel Xeon Phi Architecture*

The Intel Xeon Phi KNC* co-processor is the first commercial product from the Intel MIC family. The design is purely throughput-oriented, offering a large number of simple cores (60+) with support for 512-bit wide vector processing units (VPU). The VPU can be used to process 16 single-precision or 8 double-precision elements per instruction. To keep power dissipation per unit area under control, these cores execute the instructions in-order, and run at a low frequency (less than 1.2 GHz). The architecture is backed by large caches and high memory bandwidth.

The KNC ISA is based on x86 and allows a certain degree of compatibility with x86 tools and compilers, but it is not binary compatible. This issue has been solved in the second generation of MIC processors, Knights Landing (KNL). The architecture is designed to run up to four threads per in-order core, where each thread can fetch up to two instructions per cycle. Unlike latency-oriented architectures, the MIC architecture assumes that applications running on the system will be highly parallel and scalable.

*Knights Corner.

In order to hide the cache/memory latency caused by the in-order nature of the cores, the scheduling policy swaps threads in each cycle. When an application runs with a single threaded per core the controller switches to a special null thread before returning to the application thread, reducing the system capacity by half. Intel recommends at least two threads per core, although the optimum can vary from 2 to 4.

### *The Offload Programming Model*

The *offload* programming model runs a program on the *host* processor, which can optionally launch or "*offload*" portions of code to one or more Intel Xeon Phi co-processors. Developers identify lines or sections of code that considered suitable for massively parallel processing and insert "pragma" directives similar to those used in OPENMP. The *host* and the Xeon Phi do not have a unified memory, so it is necessary to move data back and forth between them. The Intel Xeon Phi co-processor can be programmed in C, C++ or Fortran, using OPENMP 4.0. Intel has also developed the Intel Language Extension for *Offload* (LEO) programming language that supports both KNC and KNL. LEO allows the developers to establish a shared memory programming model (called Intel MYO) between the Xeon processor and Intel Xeon Phi co-processors; this is actually the one selected for this paper.

The code is compiled for the *host* by the Intel compiler compatible with the Intel Xeon Phi architecture. When it finds *offload* directives it inserts the code necessary to transparently transfer both code and data between the *host* and the Xeon Phi, executes the code and retrieves the output data. If the co-processor is not available, the line or block of code is executed on the *host*. The keywords used to mark a code section as suitable for *offloading* are `#pragma offload target (MIC)`. The *offload* mode is a good programming approach as long as a series of conditions are met: a) the code spends most of its time doing computations without input/output, b) both the computationally intensive code sections and the required data for the computations are relatively easy to identify and encapsulate, c) the computation time is substantially greater than the data transfer time (at least $N^2$ computations for $N$ data), and d) data fits in the co-processor memory.

To *offload* code to the Intel Xeon Phi co-processor we have identified six major activities:

1. Initialization. To initialize the MIC devices in the system, we enable the environment variable OFFLOAD_init by running the "*export OFFLOAD_init = on_start*" command. In this first initialization activity, the system evaluates the conditions for offloading.
2. Memory allocation in the co-processor. Our 3-D STENCIL code uses two three-dimensional arrays, a main matrix (for reading) and a secondary matrix (for writing).
3. Data Transfer. When the application is *offloaded* to the co-processor, the data transfers between the processor and co-processor take place, at least at the beginning and at the end of the computation, but can also happen when running the code.
4. Run the *offloaded* code in the co-processor. The core loops of the 11-point 3-D *Stencil* kernel.
5. Copy the results from the co-processor to the *host* memory. Once the computations are done for all time-steps, results of the output matrix are copied from the co-processor to the host.
6. Free the data allocation in the co-processor.

## Offloading strategies for *stencil* codes on multiple device systems

This section shows how to improve the performance of the application by splitting the execution of the *Stencil* code among different devices (two cards in our case). Two main characteristics need to be

considered for the parallelization: 1) how to assign the workload to each of the execution threads and 2) the affinity of the various logical threads to the physical cores of the architecture. Due to the large number of threads and cores featuring in the Intel Xeon Phi, it is recommended to evaluate both features for improved performance in the execution of any parallel code.

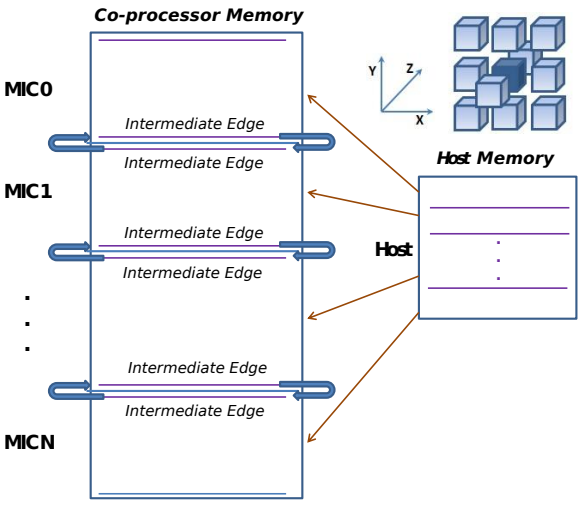


**Figure 1.** Parallelization of the 3-D Stencil on *N* co-processors

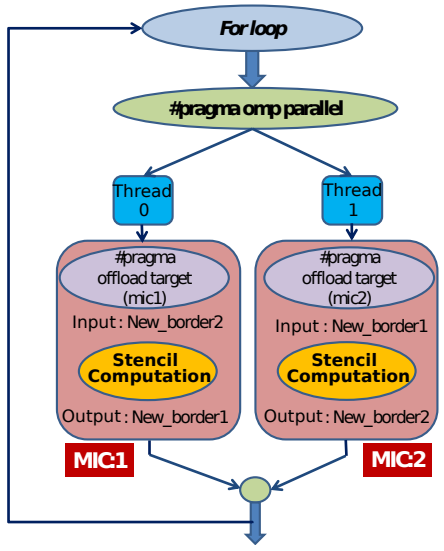


**Figure 2.** Parallelization using two threads inside the main global loop.

To *offload* the evaluated kernel to *N* co-processors, the input array *M* has to be divided into in *N* submatrices of size *M/N* (see Figure 1). In our case, the code is *offloaded* to two Xeon Phi cards; namely, MIC1 and MIC2. First, the input data need to be distributed among the two cards. We consider dividing the input data into two different submatrices by the Z axis, and transferring each subarray to a different card. However, the algorithm uses the neighboring elements in the three dimensions (11 points in total) for calculating the elements of an iteration (iteration i). This creates the need to establish a bi-direction communication between the cards to exchange data from the division borders at the end of each iteration, which could potentially limit the scalability of the application.

Then, different strategies to *offload* work to the co-processor are carried out in this paper. Please note that LEO pragmas allow the programmer to overlap computation with communication as well as make data persistent in the co-processor, so that data between iterations is not copied back and forth. Figure 2 shows the first *strategy (A)*. It uses two threads inside the main global loop (created with *#pragma omp parallel*). This loop corresponds to the iterations to achieve the desired convergence criteria. Each thread manages the computations on their assigned co-processor, including: a) transfer half of the data to the

corresponding MIC[†], b) update the edge values from the other co-processor, c) compute the kernel and d) copy the edges back to the host for the next iteration.

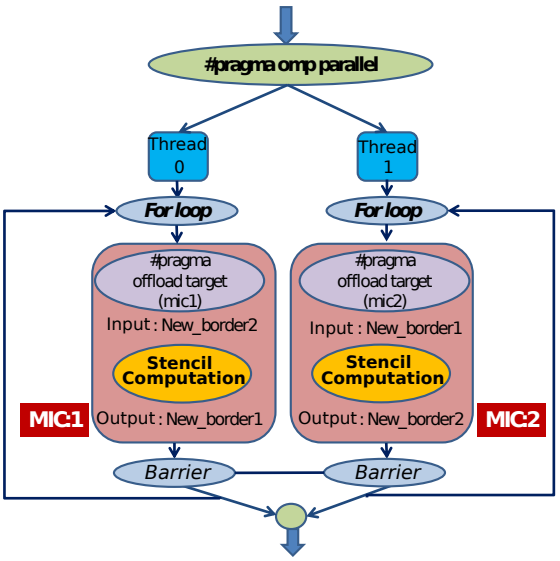


**Figure 3.** Parallelization using two parallel threads and an explicit barrier.

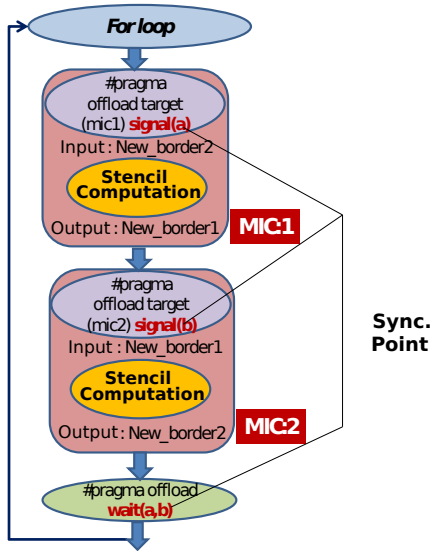


**Figure 4.** Parallelization using a single thread for asynchronous data transfers.

Figure 3 shows the details of our second *strategy (B)*. It defines a parallel region with two threads (*#pragma omp parallel*) and an explicit barrier, so that each thread performs all the steps of the algorithm independently on a different co-processor. The parallel steps include: a) copy the corresponding data (note that intermediate data persists in the co-processor), b) update the edges with the values obtained by the other co-processor, c) compute the current step of the kernel, d) copy the new edge to the host and e) wait for the rest of co-processors on an explicit barrier.

The third strategy to manage the *offloading* (*strategy (C)*) consists of a single thread loop that manages both co-processors asynchronously. This can be achieved by means of two clauses, *signal* and *wait*. When the *signal* clause is used, the *offload* process is performed in a non-blocking way (asynchronously), allowing the concurrent use of both processor and co-processor. The *wait* clause forces the host to wait for the completion of the computations in both co-processors. Internal computation steps are the same as the strategy (A). The whole process is summarized in Figure 4. Let us remind the reader that intermediate data persists in the co-processor. This last strategy is interesting as it enables the host CPU to also perform computations instead of only orchestrate the execution in each co-processor.

[†] Intermediate data persists in the co-processor

## Evaluation Environment

### *Hardware*

The evaluation platform is equipped with two Intel Ivy Bridge-EP Xeon E5-2650 CPUs (2x8 cores in total) and two Intel Xeon Phi 7120P co-processors (61 cores in each co-processor). Each Xeon E5-2650v2 CPU runs at a clock frequency of 2.6 GHz, has 8 physical cores and 16 execution threads, with a private L1 cache for instructions and one for data of 32KB each. It also has a 20 MB shared L3 cache and supports up to 32 GB of main memory DDR3-1600. This machine is used as our host.

The evaluated 7120P Xeon Phi co-processor has 61 cores operating at 1.238 GHz, and each core has a 512-bit vector unit. One key feature of the Xeon Phi co-processors is the high memory bandwidth. The evaluated Xeon Phi has 16 memory channels (32 bits wide). With up to 6GT/s (giga-transfers) transfer rate, the 7120P offers a theoretical bandwidth of 384GB/s. The system runs CentOS 6.5 Linux operating system with kernel 2.6.32, codes are build and linked using the Intel `icc` compiler (version 14.0.2), and the co-processor runs version 3.4.3 of Intel MPSS (Manycore Platform Software Stack).

### *Software*

Our experimental evaluation is based on an optimized 3-D heat diffusion *Stencil* (11 points) (similar results were obtained for other kernels but not shown due to space limitations). This kernel uses two matrices for the *Stencil* calculations. It is important to mention that Intel recommends to save one of the cores for communication when using the *offload* mode. Therefore, our *offload* version runs on 60 cores (240 threads), while the native code uses all 61 cores. We performed experiments with both single-precision and double-precision elements using an array of size 800x400x600, establishing four threads per core and affinity *balanced*. The performance results are practically independent of the size and shape of the matrices, as the memory bandwidth is saturated for all these matrix sizes regardless their shapes. Our codes use blocking to improve cache locality. After some off-line evaluations we selected and fixed a block size on the axis *X* 400 (width _TBlock = 400), 4 (height _TBlock = 4) in the axis *Y* and 4 (length _Tblock = 4) in the axis *Z*. For all the results shown in the next Section our input matrix is initialized to the following values (in degrees Celsius): center at 10, left edge 150, right edge 150, top edge 70, bottom edge 70, leading edge 70 and trailing edge 70.

## Evaluation

Our evaluation is divided into the main processes of the offloading technique to understand the impact of our technique in each of them. First, the runtime initialization is analyzed as it is a common task for all strategies. In this first step, the system evaluates the *offload* conditions and starts up the co-processors. To minimize the overhead associated to this activity, the environment variable `OFFLOAD_init` is modified so that right after the application starts the execution in the host, the co-processor cards are initialized and the activation time is reduced (command `export OFFLOAD_init=on_start` run in the host). Moreover, there are two other common aspects to all implementations that we consider here: the time spent in transferring the input data from the host to the `MIC1` and `MIC2` co-processors and the time to copy the output data back to the host. Initialization takes between 5 and 9 seconds, while data transfer takes around 0.2 seconds. These times are obtained by running an *offload* diagnostic report ("export OFFLOAD_report=2") for the evaluated 3-D *Stencil* code and corresponds to the sum of the

**Table 1.** Performance (GFLOPS) evaluation of different offloading strategies on two Intel Xeon Phi.

| Solution Activities | Single-Precision Performance of *Stencil* computations (GFLOPs) | | |
|---|---|---|---|
| | 4 | 4,3,2 | 4,3,2,5 |
| Native | 226.81 | - | - |
| Single_Co-processor | 208.92 | 168.14 | 168.51 |
| Strategy(A) | 317.59 | 310.14 | 307.59 |
| Strategy(B) | 320.67 | 307.65 | 309.98 |
| Strategy(C) | 354.65 | 342.55 | 340.58 |
| **Solution Activities** | **Double-Precision Performance of *Stencil* computations (GFLOPs)** | | |
| | 4 | 4,3,2 | 4,3,2,5 |
| Native | 112.4 | - | - |
| Single_Co-processor | 107.1 | 107.2 | 105.5 |
| Strategy(A) | 190.93 | 184.22 | 180.44 |
| Strategy(B) | 189.69 | 182.61 | 182.16 |
| Strategy(C) | 199.53 | 192.50 | 191.02 |

time consumed by the two cards (`MIC1` and `MIC2`). The transferred data corresponds to two sub-arrays of size 800x400x300, which represent 96,000,000 elements each given an amount of 384 MB (Single-Precision) or 768 MB (Double-Precision) of transferred data.

Next, we focus on the evaluation of the different offloading strategies previously described (see Table 1). Strategy (A) uses two threads inside the main global loop, strategy (B) uses two parallel threads and an explicit barrier, and finally, strategy (C) uses a single thread for asynchronous data transfers.

For strategy A on two cards, we achieve a peak performance in the range of 307.59 to 317.59 GFLOPs for single precision and 180.44 to 190.93 GFLOPs for double precision, that is 60 to 70% better than native performance on a single card. For strategy B, results are very similar to the one that uses two threads due to the explicit synchronization. It behaves slightly better for single precision and worse for double precision (1-2%). Finally, strategy (C) achieves the highest performance, improving by 6-8% over A and B. The advantage of the C strategy comes from two points. On the one hand, this strategy avoids the overhead of explicitly creating two threads (using the pragma omp parallel) to handle the data transfer. On the other hand, the strategy benefits from performing the transference of data in a non-blocking way (asynchronously), exploiting the efficient implementation of the signal/wait clauses that the architecture provides. Based on these results, we believe that the strategy (C) is the most appropriate way to perform computational work concurrently on several Intel Xeon Phi, achieving 1.7x speedup with respect to the native mode on a single card for both double and single precision. Moreover, we could increase the computational horsepower by adding the CPU as a processor to perform computation using this strategy.

## *Approximate computing: Improving the Offloading process*

Edge-value data exchange between cards causes an important time penalty. On every iteration, the cards exchange two edges of 800x400 elements, which represent around 2 MB in SP (4 MB in DP). This communication needs to go through the host since it is not allowed for the cards to exchange data directly. If we ignore this communication we will produce a result that is inconsistent with the result when running

**Table 2.** Inconsistency margins (% error) for different values of the contour.

| Contour(*) | error % |
|---|---|
| C=50, L=90, R=90, T=90, B=90, P=90, N=90 | 44.41 |
| C=50, L=140, R=140, T=140, B=140, P=140, N=140 | 64.25 |
| C=40, L=90, R=90, T=120, B=120, P=60, N=60 | 65.94 |
| C=20, L=30, R=70, T=50, B=120, P=60, N=20 | 87.07 |
| C=100, L=10, R=20, T=46, B=300, P=10, N=90 | 82.84 |
| C=10, L=150, R=150, T=70, B=70, P=70, N=70 | 93.10 |
| C=5, L=150, R=150, T=150, B=150, P=150, N=150 | 100.0 |

(*) C=Center, L=Left, R=Right, T=Top, B=Bottom, P=Previous, N=Next

on a single card. However, since Stencil algorithms have a convergence criteria based on the number of steps that we run one could wonder, could this inconsistency be admissible?

We define the error for a given point of the grid as: $point\_error = abs(real\_value - approx\_value)/real\_value$ (in %). The total error obtained for the algorithm (i.e., the error showed in the paper) is the maximum value of the point_errors measured for every point of the grid. By definition, the error obtained for exchanging information every 1 iteration is 0%. Table 2 shows the differences in the results calculated by single Intel Xeon Phi and those obtained by *offloading* to two cards without any communication. The initial contour values are modified to evaluate the impact of the input data. The error ranges from 44.41% up to 100% depending on the starting values of the contour. It should be noted that when the value of the center (eg C = 10) is closer to the values of the edges, the margin of error is lower (in this case 44.41%) as the center value (eg C = 10) is further from the values of the edges, the error margin is greater (in this case 93.10%). Even in the case of center value of 5 (C = 5) and all edges with a value of 150, we found an error of 100.0%. Based on these results, we realized that the complete elimination of the edge exchange is quite aggressive and not admissible, due to the high error obtained.

Next, we studied the behavior of the algorithm when reducing the communications to intervals of N iterations, instead of sending the contour on each iteration; i.e. having a communication delay. This is actually a trade-off between accuracy and performance that allowed us to find an acceptable margin of error. We test different values for the communication delay, i.e. to send information each 1, 2, 5 or 10 iterations. Table 3 shows the margin of error that different edge values may have when computing the 3-D *Stencil offloaded* into two MICs with delayed communications. Using this approach, we are reducing the total amount of exchanged data for a factor of 2, 5 or 10. With 1000 iterations, this amount goes from 2.5 GB down to 1.2, 0.5 or 0.25 GB (in SP) depending of the actual communication delay.

We note that when the center value (e.g., C = 50) is close to the values of the edges, the margin of error is low (in this case 0, 0.15, 0.60 and 1.43, respectively). However, when the center value (e.g., C = 10) is further from the values of the edges, the margin of error increases (0, 0.71, 2.91 and 6.73, respectively). Therefore, we will limit our evaluation to these values: center 10, left 150, right 150, top 70, bottom 70, previous 70 and next 70. If instead of fixing our number of iterations to 1000 we increase that value to 2000, 4000 or 6000 iterations, this maximum error of 6.73 is reduced to 4.67, 3.25 and 2.64 respectively.

Table 4 shows the performance evaluation of the three studied strategies for various communication scenarios of the edges (results are given in GFLOPs). We have compared the three strategies accounting all the activities performed during the computation of the 3-D *Stencil* (4,3,2 and 5 of our classification). GFLOPs for the isolated computation activity (4) are slightly higher (1-3%).

**Table 3.** Inconsistency margins (% error) for different values of the contour with delayed updates.

| Contour(*) | Iterations without communication | | | |
|---|---|---|---|---|
| | 1 | 2 | 5 | 10 |
| C=50, L=90, R=90, T=90, B=90, P=90, N=90 | 0 | 0.15 | 0.60 | 1.43 |
| C=50, L=140, R=140, T=140, B=140, P=140, N=140 | 0 | 0.27 | 1.11 | 2.16 |
| C=40, L=90, R=90, T=120, B=120, P=60, N=60 | 0 | 0.25 | 1.03 | 2.44 |
| C=20, L=30, R=70, T=50, B=120, P=60, N=20 | 0 | 0.27 | 1.15 | 2.78 |
| C=100, L=10, R=20, T=46, B=300, P=10, N=90 | 0 | 0.43 | 1.77 | 4.14 |
| C=10, L=150, R=150, T=70, B=70, P=70, N=70 | 0 | 0.71 | 2.91 | 6.73 |

(*) C=Center, L=Left, R=Right, T=Top, B=Bottom, P=Previous, N=Next

**Table 4.** Performance (GFLOPS) evaluation of different offloading strategies on two MIC cards.

| | Single-Precision | | | |
|---|---|---|---|---|
| Solution | Performance of *Stencil* computations (GFLOPs) | | | |
| Edge Communication Interval | 1 | 2 | 5 | 10 |
| Strategy(A) | 307.59 | 337.82 | 353.23 | 365.65 |
| Strategy(B) | 309.98 | 336.23 | 353.14 | 371.05 |
| Strategy(C) | 340.58 | 378.48 | 405.90 | 418.38 |
| % of Deviation | 0.00%(*) | 0.71%(*) | 2.91%(*) | 6.73%(*) |
| | Double-Precision | | | |
| Solution | Performance of *Stencil* computations (GFLOPs) | | | |
| Edge Communication Interval | 1 | 2 | 5 | 10 |
| Strategy(A) | 180.44 | 197.81 | 211.58 | 216.90 |
| Strategy(B) | 182.16 | 201.27 | 214.00 | 218.87 |
| Strategy(C) | 191.02 | 213.70 | 228.13 | 235.06 |
| % of Deviation | 0.00%(*) | 0.71%(*) | 2.91%(*) | 6.73%(*) |

**Table 5.** Time (Seconds) evaluation of different offloading strategies on two MIC cards.

| Solution | Time to compute the kernel (Seg.) | | | |
|---|---|---|---|---|
| Edge Communication Interval | 1 | 2 | 5 | 10 |
| Strategy(A) | 25.901 | 23.679 | 22.593 | 21.752 |
| Strategy(B) | 25.887 | 23.575 | 22.521 | 21.940 |
| Strategy(C) | 25.132 | 23.009 | 21.687 | 21.284 |
| % of Deviation | 0.00%(*) | 0.71%(*) | 2.91%(*) | 6.73%(*) |

For all the strategies, the edge communication frequency is reduced to once every ten cycles, improving the performance of the kernel. If we can allow a deviation of 6.73%, then the system obtains 216.90 (Strategy A), 218.87 (Strategy B), and 235.06 (Strategy C) GFLOPs for double precision, and 365.65, 371.05 and 418.38 GFLOPs for strategies A, B and C respectively for single precision. To sum up, for 6.73% of deviation the performance improves around 20% for strategies A and B, and around 23% for strategy C, giving these results almost twice as fast as the obtained by native execution. The performance in two Xeon Phi cards moved from 1.7x when using exact values to 1.95x when allowing approximate values. That means we can achieve linear speedup with an increased number of cards if we can tolerate an approximate result. This deviation will depend on the number of iterations we perform of the algorithm, 6.73% for 1000 iterations to 2.74% for 6000.

## Related Work

The *offload* programming mode for the Intel Xeon Phi architecture has not been deeply studied by the scientific community. Most references related to the usage of this programming mode point to the programming pearls books (Jeffers and Reinders 2013a; Wende et al. 2015; Feng 2015). In the complete books, the authors show many examples of applications using the Intel Xeon Phi, and some of them running in *offload* mode (Reinders and Jeffers 2015a,b). As examples, it shows how to run simulations using NWChem for quantum chemistry or the Black-Scholes algorithm for financial in *offload* mode.

Rahman (2013) presents in generic terms some *offload pragmas* that are included in OpenMP 4.0. It uses small C++/ Fortran applications to show how to *offload* work to the co-processor. Wang et al. (2014) presents a collection of very simple applications using the *offload* mode, with emphasis on data transfer to the Intel Xeon Phi. It also presents a summary of the grammar for *offloading*. In addition, Shareef et al. (2015) present the implementation of Monte Carlo methods for Intel Xeon Phi co-processor to calculate Feynman integrals in high energy physics. This paper presents applications running on both native and *offload* as in native mode. Finally, Brown et al. (2015) describe modifications in the LAMMPS molecular dynamics code to allow simultaneous execution in the CPU and the co-processor through *offload* directives, obtaining an acceleration of up to 4.7X.

## Conclusions

Both research centers and service providers have increased their computation requirements significantly over the last decade. The Xeon Phi accelerators are an example of the evolving heterogeneous architectures that emerges as a possible solution to modern application requirements. This trend is being consolidated in upcoming hardware architectures. We expect the study introduced in this paper will serve as a stepping stone for moving to the new product: Knights Landing. Many of the techniques described in the paper will be equally applicable to the new hardware given a similar design and similar core count.

This paper presents and evaluates different *offloading* strategies for the Intel Xeon Phi (KNC) co-processor. We use 3D *Stencil* codes in our evaluation, since they are essential for many scientific and engineering applications. The *offload* operation mode is specially useful to scale a problem to several co-processors for either performance or to tackle problems with memory requirements higher than what a single co-processor can provide. However, the associated data movement costs to/from the co-processor can be a limiting factor when running in this mode.

Using a single thread for asynchronous data transfers outperforms both using two threads inside the main global and having two threads with an explicit barrier by 6 and 8% respectively. However, all of the evaluated *offloading* strategies share a common performance limiting factor, the communication between the co-processors of the common edges on every iteration. Due to this fact, the speedup obtained going from one Phi card to two cards was 1,7x, instead of the 2x expected. Removing all communication proves to be inadmissible, since the margin of error is very high. However, decreasing edge communication frequency to once every 2, 5 or 10 iterations significantly improves the execution time of the kernel (by 20-23%) boosting the performance to 1,94x for the C strategy, with a deviation from the original result below 7% for 1000 iterations and 2.74% for 6000 iterations.

Given the iterative nature of the kernel, we have the opportunity to improve performance at the cost of losing precision in the obtained results. Moreover, NVDIDIA CUDA 7.5 adds new half datatypes to

support 16-bit floating point data storage and arithmetic. We definitely think, approximate computing, i.e computation which returns inaccurate results rather than a guaranteed result, has a place in the HPC.

### Acknowledgements

This work is jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grants 15290/PI/2010 and 18946/JLI/13, and by the Spanish MINECO, as well as European Commission FEDER funds, under grants TIN2015-66972-C5-3-R and TIN2016-78799-P (AEI/FEDER, UE). Mario Hernández was supported by a research grant from the PRODEP under the Professional Development Program for Teachers (UAGro-197) México.

## References

Brown WM, Carrillo JMY, Gavhane N, Thakkar FM and Plimpton SJ (2015) Optimizing Legacy Molecular Dynamics Software with Directive-based Offload. *Computer Physics Communications* 195: 95 – 101.

Chrysos G (2014) Intel® Xeon Phi™ Coprocessor Architecture. *Intel Whitepaper* 176.

Esmaeilzadeh H, Blem E, St Amant R, Sankaralingam K and Burger D (2012) Power Limitations and Dark Silicon Challenge, the Future of Multicore. *ACM Transactions on Computer Systems (TOCS)* 30(3): 11.

Feng L (2015) Chapter 20 - Data Transfer Using the Intel COI Library. In: Reinders J and Jeffers J (eds.) *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume 1. Boston, MA, USA: Morgan Kaufmann. ISBN 978-0-12-802118-7, pp. 341 – 348.

Jeffers J and Reinders J (2013a) Chapter 7 - Offload. In: Reinders J and Jeffers J (eds.) *Intel Xeon Phi Coprocessor High Performance Programming*. Boston, MA, USA: Morgan Kaufmann Publishers Inc. ISBN 9780124104945, pp. 189 – 241.

Jeffers J and Reinders J (2013b) *Intel Xeon Phi Coprocessor High-Performance Programming*. Newnes.

Rahman R (2013) *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. 1st edition. Berkely, CA, USA: Apress. ISBN 9781430259268.

Reinders J and Jeffers J (2014) *High Performance Parallelism Pearls, Multicore and Many-core Programming Approaches*, chapter Characterization and Auto-tuning of 3DFD. Morgan Kaufmann, pp. 377–396.

Reinders J and Jeffers J (eds.) (2015a) *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume 1. Boston, MA, USA: Morgan Kaufmann Publishers Inc. ISBN 9780128021187.

Reinders J and Jeffers J (eds.) (2015b) *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume 2. Boston, USA: Morgan Kaufmann Publishers Inc. ISBN 9780128038192.

Shareef B, de Doncker E and Kapenga J (2015) Monte Carlo Simulations on Intel Xeon Phi: Offload and Native Mode. In: *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. Waltham, MA, USA, pp. 1–6.

Ujaldón M (2016) CUDA Achievements and GPU Challenges Ahead. In: Francisco José Perales JK (ed.) *International Conference on Articulated Motion and Deformable Objects*. Palma, Mallorca, Spain: Springer, pp. 207–217.

Wang E, Zhang Q, Shen B, Zhang G, Lu X, Wu Q and Wang Y (2014) *High-Performance Computing on the Intel Xeon Phi*. Springer.

Wen-mei WH (2015) *Heterogeneous System Architecture: A New Compute Platform Infrastructure*. Waltham, MA, USA: Morgan Kaufmann.

Wende F, Klemm M, Steinke T and Reinefeld A (2015) Chapter 12 - Concurrent Kernel Offloading. In: Reinders J and Jeffers J (eds.) *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume 1. Boston, MA, USA: Morgan Kaufmann. ISBN 978-0-12-802118-7, pp. 201 – 223.