Guidelines to Enhance 3-D Stencil Codes on the Intel Xeon Phi Coprocessor

Mario HERNÁNDEZ [a,c,1], Juan M. CEBRIÁN [a] José M. CECILIA [b] and José M. GARCÍA [a]
[a] *Dept. of Computer Engineering, University of Murcia, 30100, Murcia, Spain*
[b] *Computer Science Department, Universidad Católica San Antonio de Murcia, Spain*
[c] *Academic Unit of Engineering, Autonomous University of Guerrero, Chilpancingo, Gro., México*

Abstract. Accelerators like the Intel Xeon Phi aim to fulfill the computational requirements of modern applications, including stencil computations. Stencils are finite-difference algorithms used in many scientific and engineering applications for solving large-scale and high-dimension partial differential equations. However, programmability on such massively parallel architectures is still a challenge for inexperienced developers.

This paper provides firm foundations to guide developers in maximizing the benefits of hardware-software co-design for computing 3-D stencil codes running on the Intel Xeon Phi (Knights Corner) architecture. We propose a set of guidelines to optimize stencil codes based on a C/C++ OpenMP implementation. The guidelines are evaluated using three kernels that are widely applied to simulate heat, acoustic diffusion as well as isotropic seismic wave equations. Our experimental results yield performance gains over 25x when compared to high-level sequential implementations (e.g., Matlab).

Keywords. 3-D stencil codes, Xeon Phi, performance optimizations, 3-D finite difference

Introduction

In the last decade, there has been a technological shift for both hardware and software towards massively parallel architectures (accelerators). Intel Many Integrated Core (MIC) [1] [2] and Graphics Processing Units (GPUs) [3] clearly show the potential of these architectures, especially in terms of performance and energy efficiency. The most powerful supercomputers in the world are currently based on accelerators [4]. Concurrently, there has been a quick evolution on programming models for co-processors and GPUs. However, porting applications to these systems is still not a straightforward task. In order to maximize performance and energy efficiency of their systems, software developers need to use the latest breakthroughs in both high performance computing and the specific field of interest (e.g., image processing, modeling of acoustic or heat diffusion, etc). This vertical approach enables remarkable advances in computer-driven scientific simulations (the so-called hardware-software co-design).

As stated in [5], many applications are developed using an algorithmic method that captures a pattern of computation and communication (so-called *Dwarf*). These patterns are repeated in different applications and thus, hardware-software solutions can be extrapolated to many scientific areas. This is actually the case of stencils. Stencil codes comprise a family of iterative kernels that operate

[1] Corresponding Author: Department of Computer Engineering, University of Murcia, 30100, Murcia, Spain; E-mail: mario.hernandez4@um.es

over an N-dimensional data structure that changes over time, given a fixed computational pattern (stencil). Given the high level of abstraction and the wide variety of stakeholders that can benefit from stencil computations, the implementation is usually developed using high level programming languages such as Matlab [6]. Matlab provides an accessible entry-point for inexperienced programmers (e.g., engineers or physicists) to create a computational approximation model of their problem of interest. This abstraction from low level programming is achieved through a large number of complex mathematical libraries. However, this ease of programming usually compromises performance. In addition, those programming languages are inherently sequential, leading to a limited exploitation of the available resources in modern processors and accelerators. Moreover, even a low-level straightforward implementation of a stencil code may suffer from low performance [7].

In this paper, we compile and validate a set of optimization guidelines using three 3-D stencil kernels from different fields of research: 1) 3-D heat diffusion stencil (11-point), 2) 3-D acoustic diffusion stencil (7-point) and 3) 3-D isotropic seismic wave stencil (25-point). We start from a simple sequential implementation of the code in Matlab. The first step is to port the code to C/C++ using OpenMP. This version will be the starting point of our guidelines evaluation. Our results show substantial performance gains (over 25x) compared to a sequential high-level scientific programming language implementation (e.g., Matlab), while at the same time we experience a reduction in the energy consumption at a board level. The aim of this paper is to guide inexperienced software developers to optimize stencil computations for the Intel Xeon Phi architecture, although some of our conclusions are also applicable to other massively parallel architectures (e.g., GPUs).

The paper is structured as follows. The next section gives some fundamentals and related work about stencil computations and the MIC architecture. Guidelines for improving the parallel performance for 3-D stencil codes are introduced in Section 2. Section 3 shows our evaluation results. Finally, we summarize our conclusions in Section 4.

1. Background and Related Work

1.1. 3-D Stencil Computations

Stencil codes [7,8,9] are a type of iterative kernels which update data elements according to some fixed predetermined set or pattern. The stencil can be used to compute the value of several elements in an array at a given time-step based on its neighbour values. This may include values computed in previous time-steps (including the element itself). Stencils are the base of finite-difference algorithms used for solving large-scale and high-dimension partial differential equations (PDEs). PDEs provide numerical approximations to computational expensive problems, being widely used in many scientific and engineering fields. This allows scientists to accurately model phenomena such as scalar-wave propagation, heat conduction, acoustic diffusion, etc.

Algorithm 1 shows the pseudo-code of a generic three-dimension (3-D) stencil solver kernel. It is implemented as a triple nested loop traversing the complete data structure while updating each grid point. The computation of every output element usually requires: a) the weighted contribution of some near neighbors in each direction defined by the physics of the problem, b) the previous value of that element in a time *t-1* (for second order in-time stencils) and, c) a single corresponding point from other input arrays. The code normally uses two copies of the spatial grid swapping their roles as source and destination on alternate time steps as shown in the Algorithm 1.

An important feature of these algorithms is that 3-D stencil kernels usually suffer from a high cache miss rate and poor data locality. The reason is that, for input sizes that exceed the cache capacity, by the time we reuse an entry from the dataset it has already been replaced from the cache. Moreover, the non-linear memory access pattern of 3D based implementations creates additional memory stalls. As a result, standard implementations of the 3D stencil solvers typically reach a small fraction of the hardware's peak performance [10].

Algorithm 1 The 3-D stencil solver kernel. *width, height, depth* are the dimensions of the data set including border (*halo*) points.

```
for time = 0; time < TimeMax; time++ do
    for z = 1; z < depth − BorderSize; z++ do
        for y = 1; y < height − BorderSize; y++ do
            for x = 1; x < width − BorderSize; x++ do
                stencil_solver_kernel();
            end for
        end for
    end for
    tmp = Input_Grid; Input_Grid = Output_Grid; Output_Grid = tmp;
end for
```

1.2. Intel Xeon Phi Architecture

The Intel Xeon Phi (Knights Corner) coprocessor [1,2] is the first commercial product of the Intel MIC family. The design is purely throughput oriented, featuring a high 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 instructions in-order and run at a low frequency (<1.2Ghz). The architecture is backed by large caches and high memory bandwidth. Xeon Phi is based on the x86 ISA, allowing a certain degree of compatibility with conventional x86 processors (but not binary).

The architecture is tailored to run four independent threads per core, where each in-order core can execute 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. In order to hide the cache/memory latency caused by the in-order nature of the cores, the scheduling policy swaps threads on each cycle. When an application runs a single thread per core, the scheduler switches to a special *null thread* before going back to the application thread. Suffice it to say, Intel recommends at least two threads per core, although the optimal may range from 2 to 4. Running a single thread per core will reduce the peak capacity of the system by half.

1.3. Related Work

Multi-core architectures provide good opportunities for parallelizing stencil applications. Authors in [11] present a thorough methodology to evaluate and predict stencil code performance on complex HPC architectures. The authors in [12] introduce a methodology that directs programmer efforts toward the regions of code most likely to benefit from porting to the Xeon Phi as well as providing speedup estimates. Other researchers [13,14] investigate the porting and optimization of the test problem basic N-body simulation for the Intel Xeon Phi coprocessor, which is too the foundation of a number of applications in computational astrophysics and biophysics.

Many proposals have been focused on improving cache reuse. Tiling is a program transformation that can be applied to capture this data reuse when data does not fit in cache. In [15,16] authors focus on exploiting data locality applying tiling techniques. On the other hand, several works like [17,18] consider locality and parallelism issues. Kamil *et al.* [17] examine several optimizations targeted to improve cache reuse across stencil sweeps. Their work includes both an implicit cache oblivious approach and a cache-aware algorithm blocked to match the cache structure. This enables multiple iterations of the stencil to be performed on each cache-resident portion of the grid. Authors in [18] develop an approach for automatic parallelization of stencil codes that explicitly addresses the issue of load-balanced execution of tiles.

2. Guidelines to Optimize Stencil Codes

This section focuses on key design decisions that should be considered to exploit massively parallel architectures at maximum. The main tradeoffs between the different optimization strategies are discussed to help designers in making an informed decision. Performance evaluation and energy results are shown in the Section 3.

2.1. Parallelization Strategies

The parallelization process consists of dividing an application in different ”threads” or ”tasks” that run in parallel on the target architecture. Libraries like *Boost* or *OpenMP* allow software developers to easily write parallel applications and orchestrate a parallel run. OpenMP development is based on *#pragma* statements that are captured by the compiler, validated and translated to the appropriate function calls to the OpenMP library and runtime system. For instance, the directive *#pragma omp parallel for private(i)* before a for loop instructs the compiler to parallelize the *for* loop using all available cores and that each core holds a private instance of variable *i*. Additional requirements for using OpenMP include the declaration of the OMP header file in the source code, and the compiler flag *-openmp* to link with the OpenMP libraries. The *collapse* clause is useful in stencils to merge loop iterations, increasing the total work units that will be partitioned across the available threads.

In this paper, we have parallelized the outer two loops using the *OMP parallel for* pragma with the *collapse* construct, as shown in Figure 2. The inner loop is left unchanged to be vectorized (described in Section 2.3). We have set the KMP_AFFINITY to *scatter* to distribute the threads across the Xeon Phi cores, maximizing the usage of the cache storage space.

Algorithm 2 The 3-D stencil solver kernel parallelized with OpenMP.

```
1:  for time = 0; time < TimeMax; time++ do
2:      #pragma omp for collapse (2);
3:      for z = 1; z < depth − BorderSize; z++ do
4:          for y = 1; y < height − BorderSize; y++ do
5:              for x = 1; x < width − BorderSize; x++ do
6:                  stencil_solver_kernel();
7:              end for
8:          end for
9:      end for
10:     tmp = Input_Grid; Input_Grid = Output_Grid; Output_Grid = tmp;
11: end for
```

2.2. Memory Guidelines

3-D stencils operate over an input data represented as a three-dimensional array of elements (single/double floating point precision). Next we summarize the best practices we have followed to improve the performance related to memory allocation and usage.

Data Allocation. Our first recommendation is related to the way data is allocated in memory. We advise to allocate all rows of the 3-D arrays consecutively in memory (i.e., row major order). This way of allocating memory follows the convention for n-dimensional arrays in C/C++, meaning that the right-most index of the array has an access pattern of stride one. Therefore, mapping the unit stride dimension to the inner loop in nested loop iterations produces a better use of cache lines. The dataset is thus accessed in order of planes (layers), columns, and finally rows from outer to inner level.

Data Alignment. Another key factor that can limit performance in the Xeon Phi architecture is the use of unaligned loads and stores. To perform data alignment correctly, the traditional memory

allocation function calls in C/C++ (i.e. *malloc()* and *free()*) are replaced by an alternative implementations that support data alignment (i.e. *_mm_malloc()* and *_mm_free()*).

For the Xeon Phi we select an alignment factor of 64 bytes, which is passed as a parameter to the *_mm_malloc()* routine. For example, to allocate the *Input_Grid* float data structure aligned to 64-bytes we can use `float Input_Grid = _mm_malloc(Input_Grid_Size, 64)`. Additionally, we have used the clause *__assume_aligned (Input_Grid, 64)* to provide the compiler with additional information regarding the alignment of the *Input_Grid* in vectorized loops. Without this information, the compiler may not be able to correctly identify the alignment used by the data structure.

Padding. Padding is an interesting technique that rearranges data in cache memory by allocating extra unused "dummy" information in data structures. In our case, padding has been a very convenient technique to be applied in stencil codes for two main reasons:

1. To avoid the misalignment among rows. As the dataset structure is allocated in memory as a whole, i.e. by using a single *_mm_malloc()* instruction, this may lead to a misalignment of the data between rows depending on the width of the *x* dimension. Thus, we use a new *width* adding some elements (if needed) to ensure that the first element of each row is on the desired address boundary (64 bytes in Xeon Phi). The new *width* with padding is calculated as $width\_PADD = ((((width * sizeof(REAL)) + 63)/64) * (64/sizeof(REAL)))$. REAL stands for the data type used (float or double) in the kernel.
2. Avoiding pathological conflict misses. Conflict misses[2] may appear under certain combinations of blocking size, input size, etc. In some unlikely scenarios, several cache lines mapping to the same cache set may cause a low cache hit rate. For the Xeon Phi this happens when a kernel accesses data with a *4KB* stride (*L1*) or a *64KB* stride (*L2*). In our kernels, we have experienced this problem in the seismic kernel when accessing data in the Z dimension in some specific input grid sizes. Our recommendation is to use *padding* in the problematic dimension to change the access stride.

Blocking. Stencil codes with an input size that does not fit on the higher cache levels of the processor will experience a significant performance degradation due to cache capacity misses. Code transformations that improve data locality can be useful to hide the complexities of the memory hierarchy, improving overall performance of 3-D stencil codes. Basic transformations include loop transformations[3] and data transformations[4] (e.g., blocking).

Blocking is a transformation which groups loop iterations into subsets (or tiles) of size N. The size of the tiles needs to be adjusted to fit in the cache in order to obtain maximum performance gains by exploiting data locality. In this way, cache misses can be minimized by bringing a data block into cache once for all necessary accesses.

In 3-D stencil codes our goal is to exploit data locality, focusing on increasing the reuse of the elements of the plane (*X-Y*) for some layers. The first step is to create tiles of size *bz*, *by* and *bx*. Next, three additional loops are created over the three existing loops to traverse the dataset in tiles of the selected sizes. A blocking version of a generic 3-D stencil is shown in algorithm 3. After an analysis of different block sizes we empirically found that *width_TBlock=width*, *height_TBlock=4* and *depth_TBlock=4* offer good results for all the evaluated kernels, that is, blocking four rows over four Z planes while keeping all columns.

2.3. Vectorization

One of the key design features of the Xeon Phi architecture is the use of wide SIMD registers and vector functional units to improve performance. The MIC Knights Corner (KNC) architecture

[2]Misses caused by cache lines that map to the same cache set.
[3]Loop rearrange.
[4]Changing the layout of data.

Algorithm 3 Blocking technique applied to the 3-D stencil solver.

```
1:  for bz = 1; bz < depth − BorderSize; bz+ = depth_Tblock do
2:      for by = 1; by < height − BorderSize; by+ = height_Tblock do
3:          for bx = 1; bx < width − BorderSize; bx+ = width_Tblock do
4:              for z = bz; z < MIN(bz + depth_Tblock, depth − BorderSize); z++ do
5:                  for y = by; y < MIN(by + height_Tblock, height − BorderSize); y++ do
6:                      for x = 1; x < MIN(width_Tblock, width − BorderSize − bx); x++ do
7:                          stencil_solver_kernel();
8:                      end for
9:                  end for
10:             end for
11:         end for
12:     end for
13: end for
```

implements a subset of the AVX512 instruction set that operates over 512-bit wide registers. It is crucial to make use of SIMD features in order to get the best performance out of this architecture.

There are several issues that need to be addressed to achieve the automatic vectorization of the code. The first one is data alignment. Unaligned loads are not available on the KNC architecture, but will be in the upcoming Knights Landing (KNL). Therefore, data accesses must start with an address aligned to 64 bytes (as discussed previously). The second issue is remainders. Vectorizing a loop requires to handle the remainder data items when the number of iterations is not a multiple of the vector length (peeling the loop). Nevertheless, this is handled automatically by the compiler in our evaluation.

The ICC compiler checks for vectorization opportunities whenever the code is compiled using *-O2* or higher. Developers can help the compiler to face loop dependencies by providing additional information to guide the vectorization process. By using the `#pragma simd` sentence before the inner most loop the programmer instructs the compiler to vectorize without performing any dependency, aliasing or performance analysis. This *pragma* is designed to minimize the amount of source code changes required to vectorize the code. In addition, the *restrict* keyword before a pointer variable informs ICC that the memory referenced by that pointer is not accessed in any other way (avoiding pointer aliasing), and may be necessary in the vectorization process (requires the `-restrict` compiler flag). Alternatively, the `-fargumentnoalias` compiler flag would instruct the compiler that function arguments cannot alias each other along the whole program. End users can get a detailed report of the vectorization process using the *-vec-report3* flag. This can give useful insights about obstacles found in the vectorization process, such as non-contiguous memory accesses and loop data dependencies.

2.4. Other Easy-to-Use Low-Level Optimizations

We have restricted ourselves to easy-to-use low level hardware optimizations that can be used by non-experienced programmers. Other optimization techniques, such as the use of intrinsics, FMA instructions, or prefetching are therefore out of scope of this paper.

Streaming stores. When an application writes its output in a memory location, the destination data block is loaded from memory and moved along the memory hierarchy until it reaches the L1 and thus it can be written. However, if the data block is not required for any computation other than storing the output, this operation can seriously pollute the memory hierarchy. Streaming stores address this problem by storing a continuous stream of output data without gaps between data items directly into memory, skipping the intermediate levels of the memory hierarchy. This method stores data using a non temporal buffer, improving memory bandwidth utilization and potentially performance. We have used non-temporal stores in the 3-D stencils by putting a *#pragma vector*

non temporal directive before the inner-most loop. Alternatively, the developer can use the compiler flag *-opt-streaming-stores* to control the generation of streaming stores.

Huge page size. The Xeon Phi architecture can be configured to run using either 4KB or 2MB page sizes (*aka* huge pages). This configuration allows the Xeon Phi architecture TLB to map 128 MB of memory, as compared to the 256KB mapped by default. This might reduce page faults significantly (up to 15%) in certain applications. The allocation of huge pages is done by replacing the *_mm_malloc()* calls with the *mmap()* function.

ECC Memory and Turbo. Error-Correcting Code (ECC) is used to provide error detection and correction in case of hardware memory errors. On Xeon Phi, it is possible to enable/disable ECC, slightly increasing performance. Turbo mode is another feature of the Xeon Phi KNC to allow overclocking based on the current power usage and temperature. ECC can be enabled/disabled using the command *micsmc –ecc disable/enable mic0*, whereas *micsmc –turbo enable/disable mic0* can be used to enable/disable the Turbo mode.

3. Evaluation

3.1. Target Platforms

Our evaluation uses the Intel's `ICC` compiler (version 14.0.2), running on CentOS 6.5 with kernel 2.6.32 and Intel's MPSS 3.4.3. The target system contains two Ivy Bridge-EP Intel Xeon E5-2650v2 CPUs (2x8 cores in total) running at 2.6 GHz and 32GB of DDR3-1600 main memory. The evaluated Intel Xeon Phi coprocessor is the 7120P model. The 7120P has 61 cores working at 1.238 GHz, 32KB of the L1 data and instruction caches and 512 KB of L2 cache per core. The architecture provides a theoretical peak computation of 2420 gigaflop per second (GFlop/s) for single precision variables (32 bits). In addition, another important feature of Intel Xeon Phi coprocessors is the high memory bandwidth. The 7120P has 16 memory channels, each 32-bits wide, adding up to a theoretical bandwidth of 352 GB/s (transfer speeds of 5.5 GT/s).

With these features, the theoretical arithmetic intensity[5] (AI) to exploit the full performance of the Xeon Phi 7120P is around 10.9 Flop/Byte. Considering the true achievable maximum memory bandwidth is limited to 50-60% of the peak memory bandwidth, the feasible AI is around 6.8 Flop/Byte. This means that, for this architecture, we can characterize a given compute kernel as compute bounded if its AI is greater that 6.8 Flop/Byte, or memory bounded in the opposite case.

3.2. Target Codes

We have evaluated three different stencil solvers to test our approach. These solves cover a wide research area and have distinct computational features. The most common stencil code is represented by the 3-D acoustic diffusion stencil, which uses a stencil of 7-point spatial neighbors and second order in time. It uses three different matrices of the same size for the kernel calculation and has a low AI (slightly more than 1). Our second kernel is the 3-D isotropic seismic wave stencil of 25-point spatial neighbors and also second order in time. This solver has greater AI than the previous one (slightly greater than 2), although it uses an additional matrix for storing physical characteristics (four matrices in total). Finally, we have evaluated our simplest solver, the 3-D heat diffusion stencil of 11-point spatial neighbors and first order in time, which only uses two matrices for the stencil calculation and has an AI close to 3. As we can observe, all codes are memory-bounded and thus, they could only achieve a small fraction of the peak performance of the Xeon Phi.

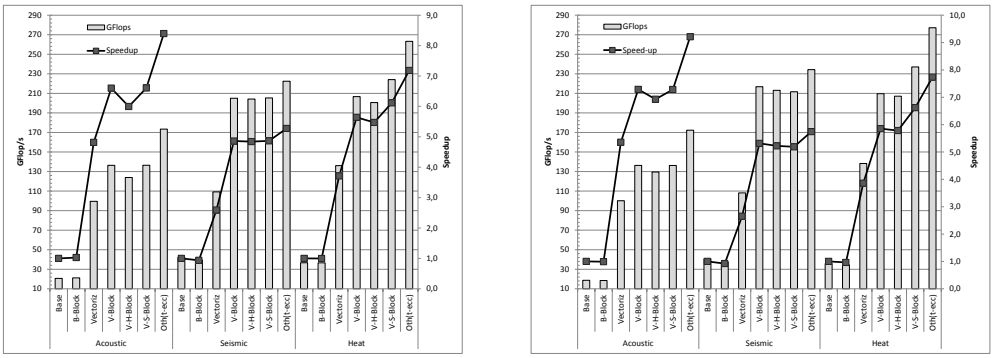
[5]Number of floating point operations per byte of data [10].

Related the shape of the grid, as it dictates the memory access patterns, rectangular cuboid shapes of *width x height x depth* have been chosen for the experiments. Finally, two sizes of the input grid of the kernels have been considered: small one (grid of 400x300x200 and 92 MB of size) and large one (400x600x1000 grid, 916 MB size). Note that these sizes are referred as to the size of only one of the input matrices of the kernels.

3.3. Experimental Performance Evaluation

The base implementations of our kernels are developed using C and OpenMP. The C versions outperform the equivalent Matlab codes by a factor of $\sim 5\times$ when running on a single thread. Additionally, we validate our results by performing a performance profile using both Intel VTune Amplifier XE 2015 (for detailed code analysis) and PAPI (for power measurements and L1 cache analysis).

Figure 1 shows the performance and speedup (secondary Y axis) for the two analyzed matrix sizes and for the different optimizations we have presented along the paper. We have set four threads per core and the KMP_AFFINITY to *scatter*, and the block size used in the graphs are *width_TBlock=width*, *height_TBlock=4* and *depth_TBlock=4*. The labels of the plots mean the following: *base*) stands for unvectorized code, *B-Block*) for unvectorized code with blocking, *Vectoriz*) for vectorized code, *V-Block*) for vectorized code with blocking, *V-H-Block*) for vectorized code with blocking and huge pages, *V-S-Block*) for vectorized code with blocking and sstores and, finally, *oth(t-ECC)*) for vectorized code with blocking, sstores, turbo and ECC disabled.



(a) Small size (92 MB)

(b) Large size (916 MB)

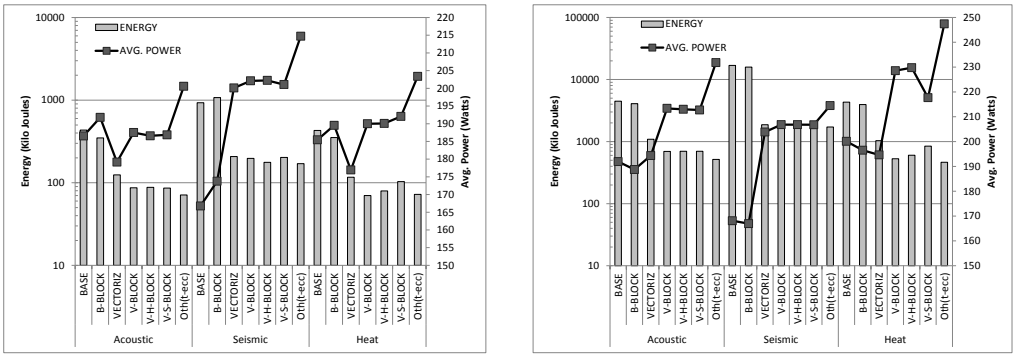
Figure 1. Performance (Gflops) and speedup (secondary Y) of different optimization strategies for the 7120P.

The use of blocking on the parallel base code offers little to none performance improvement in both sizes and for all kernels. This suggests that the Xeon Phi is able to significantly hide the memory latency by switching threads during cache misses. On the other hand, when vectorization is enabled, data elements from matrices are consumed much faster by the VPU, and blocking mechanisms start to pay off. Vectorization shows a substantial performance improvement for all kernels and sizes, but is far from the ideal $16\times$ speedup expected from the 512-bit registers. Checking at the VTune profile we notice that the memory-boundedness nature of the kernels is severely limiting the performance of the vector units. As a consequence, blocking obtains a clear improvement when applied to the vectorized version.

We have also tested our kernels using *huge pages* and *sstores* but found no performance improvement but for heat stencil kernel, specially for big matrix sizes. Again, we attribute this behavior to the low AI of the kernels and the way that the Xeon Phi hides latency by constantly context switching between execution threads. Finally, disabling ECC and overclocking the board yields

substantial performance improvements for all kernels (although most of the performance benefit comes from disabling ECC).

Figure 2 shows the energy profile and the average power (secondary Y axis) for the two analyzed matrix sizes. Energy measurements show a similar trend to that of performance. The greatest energy saving comes from vectorization, showing a slight increase in the average power consumed by the board for big matrix sizes. This gives an idea on how beneficial is this technique to improve energy efficiency. Moreover, for small matrix sizes, both acoustic and heat kernels show a reduction on the average power dissipated by the board. This can only mean that the cores spend more time idle waiting for data, and power saving mechanisms remain active for longer periods of time.



(a) Small size (92 MB)

(b) Large size (916 MB)

Figure 2. Energy and average power (secondary Y) of different optimization strategies for the 7120P.

The average power remains barely unchanged for both acoustic and seismic kernels when using other optimizations rather than vectorization. This suggests that the core activity remains relatively unchanged, something that does not happen with the heat kernel. This could be attributed to its higher AI, meaning that code changes have a greater impact on the computations performed by this kernel, even if they do not translate into a noticeable speedup. Finally, when ECC is disabled and the cores are overclocked, we see a substantial increment on the average power dissipated by the board. There is a direct relationship between power and frequency, when overclocking the cores the power dissipated by the board increases accordingly.

4. Conclusions and Future Work

This paper presents a set of guidelines to help developers in maximizing the benefits of hardware-software co-design for computing 3-D stencil codes running on the Intel Xeon Phi (Knights Corner) architecture. Programmability on massively parallel architectures can be a challenge for inexperienced developers, but the proposed guidelines will ease the porting of any stencil-based application to these architectures. Real world applications based on Stencil computations can take great advantage of the proposed optimizations, not only providing faster results, but also improving accuracy by allowing more detailed simulations of different phenomena. This has a real impact on the society enabling scientists to overcome emerging challenges.

We have used the C/C++ language with OpenMP extensions to code the three stencils kernels evaluated in this work. Experimental results show the performance evolution for different kernels through the optimization process. Vectorization is the key strategy from our results, from both performance and energy points of view. In addition, the application of blocking techniques improves

memory locality for these kernels, and thus performance and energy. Finally, the use of overclocking and non-ECC memory improves the performance but expending more energy at the board level.

As for future work, we are interesting to extend our evaluation to larger datasets using also double data types. To properly handle big sizes, we plan to split input data among different Xeon Phi cards, analyzing the communication effects on the performance.

Acknowledgements

This work is jointly supported by the Fundación Séneca under grant 15290/PI/2010, and the Spanish MINECO as well as EC FEDER funds under grant TIN2012-31345 and CAPAP-H5 NoE (TIN2014-53522-REDT). In addition, to Nils Coordinated Mobility under grant 012-ABEL-CM-2014A (partly financed by the ERDF). Mario Hernández was supported by the PROMEP under the Teacher Improvement Program (UAGro-197) - México.

References

[1] Jim Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Elsevier Waltham (Mass.), Amsterdam, Boston, 2013.

[2] Rezaur Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Berkely, CA, USA, 1st edition, 2013.

[3] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, and V. Volkov. Parallel Computing Experiences with CUDA. *IEEE, Micro*, 28(4):13–27, July 2008.

[4] Top 500 supercomputer site, [last access 15 June 2015]. `http://www.top500.org/`.

[5] Krste Asanovic and et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Berkeley, Dec 2006.

[6] Timmy Siauw and A. M. Bayen. *An introduction to MATLAB programming and numerical methods*. Elsevier, 2015.

[7] Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schuller. *Multigrid*. Academic press, 2000.

[8] Matteo Frigo and Volker Strumpen. Cache Oblivious Stencil Computations. In *Proc. of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 361–366, New York, USA, 2005. ACM.

[9] V.T. Zhukov, M.M. Krasnov, N.D. Novikova, and O.B. Feodoritova. Multigrid effectiveness on modern computing architectures. *Programming and Computer Software*, 41(1):14–22, 2015.

[10] Jim Reinders and James Jeffers. *High Performance Parallelism Pearls, Multicore and Many-core Programming Approaches*, chapter Characterization and Auto-tuning of 3DFD, pages 377–396. Morgan Kaufmann, 2014.

[11] Raúl de la Cruz and Mauricio Araya-Polo. Modeling stencil computations on modern HPC architectures. In *5th Int. Workshop (PMBS14) held as part of SC14*. Springer, 2014.

[12] J. Peraza, A. Tiwari, M. Laurenzano, L. Carrington, W.A. Ward, and R. Campbell. Understanding the performance of stencil computations on Intel's Xeon Phi. In *Int. Conf. on Cluster Computing (CLUSTER)*, pages 1–5, Sept 2013.

[13] Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. Test-driving Intel Xeon Phi. In *Proc. of the 5th ACM/SPEC Int. Conf. on Performance Engineering*, pages 137–148. ACM, 2014.

[14] Karpusenko Vadim Vladimirov Andrey. Test-driving Intel Xeon Phi coprocessors with a basic N-body simulation. *Coflax International*, 2013.

[15] G. Rivera and Chau-Wen Tseng. Tiling Optimizations for 3D Scientific Computations. In *Supercomputing, ACM/IEEE Conference*, pages 32–32, Nov 2000.

[16] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Data locality enhancement by memory reduction. In *Proc. of the 15th int. conf. on Supercomputing*, pages 50–64. ACM, 2001.

[17] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and Explicit Optimizations for Stencil Computations. In *Proc. of the Workshop on Memory System Performance and Correctness*, MSPC '06, pages 51–60, New York, USA, 2006. ACM.

[18] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '07, pages 235–244, New York, USA, 2007. ACM.