Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Stellar Atmosphere Simulation code Bifrost on Intel Xeon Phi Knights Landing

Mikołaj Szydlarski[aa]* and Vegard Eide[b]

[a] *Institute of Theoretical Astrophysics, UiO*
[b] *IT Department, NTNU*

Abstract

In this white paper we report our experiences in porting the stellar atmosphere simulation code Bifrost to Intel Xeon Phi - Knights Landing Architecture (KNL). Bifrost is a parallel, Fortran 95/MPI code for solving the 3D radiation magnetohydrodynamic (RMHD) equations on a staggered grid using a high order compact finite difference scheme. The focus is on finding the most performant runtime setup and estimate possible performance gain when compared with Intel Haswell based nodes.

1. Description of Bifrost

Bifrost is a general, flexible and massively parallel, Fortran 95 code for solving the 3D radiation magnetohydrodynamic (RMHD) equations, described in detail in Gudiksen et al. (2011) [1]. In short, Bifrost solves explicitly the MHD partial differential equations (PDEs) on a staggered grid using a high order compact finite difference scheme. Bifrost is a very general modeling code with a variety of modules available for boundary conditions and the equation of state.

Bifrost employs the *Message Passing Interface* (MPI) for parallelization. The computational grid is split into subdomains and distribute one subdomain to each node. Computing the derivatives and interpolating the variables in Bifrost use a stencil of six grid points. The two or three grid points nearest the edge of a subdomain then depend on data that is outside the subdomain belonging to the local node. To provide those missing data for subdomains efficiently, Bifrost includes ghost cells around each subdomain that are copies of cells belonging to neighboring nodes. This makes it possible to do less communication as the ghost cells are filled with the correct values from the neighboring nodes less often. The typical size of five ghost cells makes it possible to do both derivative and interpolation along the same direction without having to communicate with neighboring nodes. Setting the size of subdomains sufficiently big (typically 64^3 grid points per MPI rank) would typically bound communication cost making Bifrost run computationally dominant.

[a]* Corresponding author. *E-mail address*: mikolaj.szydlarski@astro.uio.no

1.1. Test cases and experiment setup

We have been provided with two test cases with the same initial setup but different sizes, namely 192x192x192 (small case) and 384x384x384 (medium case) grid points. Domain sizes have been selected small enough to fit into memory of one computational node but at the same time are large enough to enable necessary physics typically set in the production runs.

1.2. Runtime characteristics

The Bifrost code is capable of finding automatically an optimal domain decomposition for a given number of MPI processes. This means test cases can be mapped to any chosen number of MPI processes. In Figure 1, we show the average time cost of one iteration as a function of number of MPI processes. For both test cases, the local minimum is found to be 64 which is the number of physical cores on the KNL-ninja test node (see description of the hardware in next section). We have also plotted the measured average memory usage per-core (dash-dot line) along with the theoretical scaling curve (dot line) which however does not take into account ghost cells. Those two curves deviate significantly for runs with a high number of MPI ranks which visualizes the storage penalty due to the use of ghost cells.

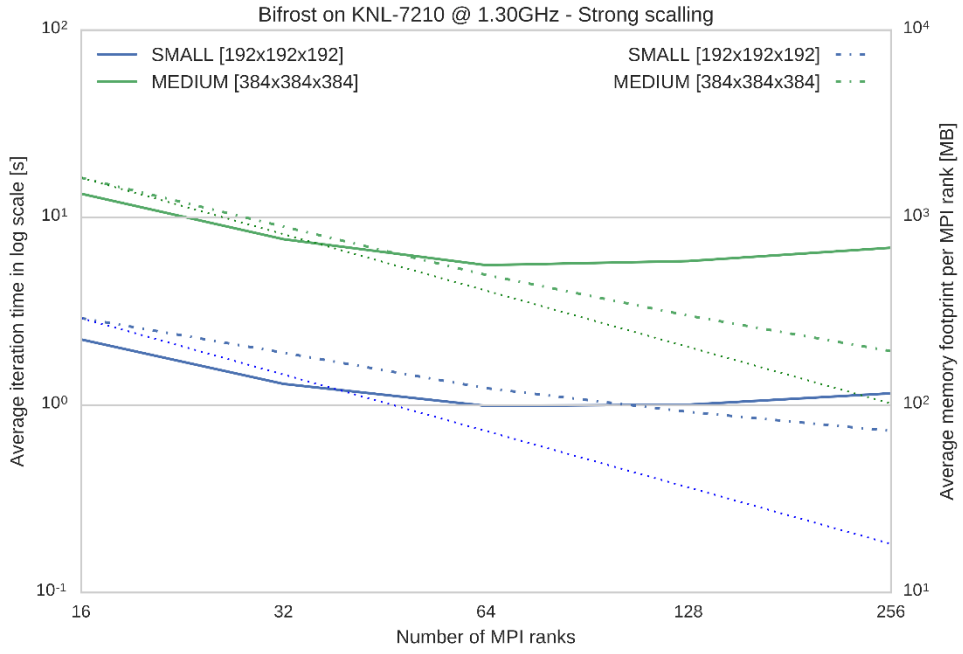

Figure 1. Strong scaling on the KNL node for an average iteration time cost (solid lines) as a function of MPI processes. The dash-dot lines shows the average memory usage per core. The dotted line is the theoretical memory scaling curve without taking into account ghost cells.

Test case	Memory usage per core (peak/average) [MB]	
	`mpirun -n 64`	`mpirun -np 20`
Small (192x192x192)	390.23 / 311.28	921.35 / 738.21
Medium (384x384x384)	524.86 / 489.77	1474.09 / 1352.87

Table 1. Memory usage per core for the runtime setup used in experiments for the most performant runs.

1.3. Performance analysis

A performance analysis with Intel VTune revealed that the application is already well optimized and consist of hotspots that are somewhat evenly distributed between bandwidth- and compute-bound workload. Most of the CPU time is spent on memory operations (`memcpy` and `memset`) and float-point intense interpolation on stagger grid (`z[dn,up]_set`, `y[dn,up]_set` and `x[dn,up]_set`).

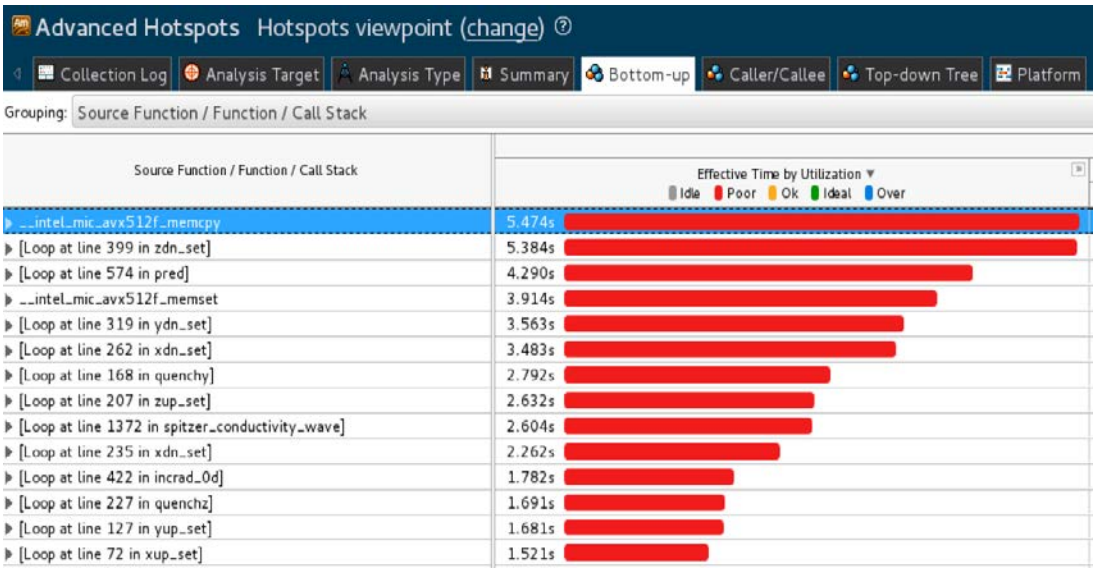

Figure 2. Intel VTune screenshot displaying information for the medium size run on the KNL with the optimized code. The first column shows the function name, the second column the CPU time in seconds spent in the function. Hot spots analysis reveals a relatively flat profile where the substantial amount of time is spent in memory operations (`memcpy` and `memset`) and floating-point intense interpolation on the staggered grid (`z[dn,up]_set, y[dn,up]_set` and x[dn,up]_set). The `pred` function is a predictor function, from third-order Hyman predictor-corrector timestamping scheme. It is a very simple subroutine consisting of four array multiplications, but also two calls to `memcpy`, hence its high position in the list.

2. Hardware description and software environment

Knights Landing (KNL) is the codename for the second generation Many-Integrated Core (MIC) product line from Intel. KNL comes with up to 72 cores based on the «Silvermont» Atom core, each with support for 4 hardware threads, and with two 512-bit vector processing units. Like the previous MIC generation, Knights Corner, it can be delivered as a coprocessor, but in addition KNL can work as a standalone processor running its own host OS. The KNL is available with up to 384 GB of DDR4 memory and in addition have on-package high-bandwidth memory, up to 16 GB, that can operate in different modes, as cache and/or addressable memory.

In our test runs we used two different nodes, one single socket node equipped with KNL and one dual socket system with Haswell E5-2660v3. The detailed characteristics are listed in Table 2.

Test system	KNL-ninja	Haswell E5-2660v3
Processors per node	64	20
Multithreading, threads per core	4	2
Processor speed	1.30 GHz	2. 60 GHz
Memory (DDR)	192 GB	64 GB
High Bandwidth Memory (MCDRAM)	16 GB	-
Intel compiler version	2017	2017
MPI	Intel MPI 2017	Intel MPI 2017
Instruction set compiler switch	`-xMIC-AVX512`	`-xCORE-AVX2`

Table 2. Basic parameters describing the hardware and software environment for the tests.

2.1. Compilation and code preparation

We used the Intel Fortran Compiler Version 17.0.0 with Intel MPI Library 2017 for compilation. For runs with time measurements we set following compilation flags:

```
-O3 -xHost
```

Where the `-xHost` flag tells the compiler to generate an executable for the highest instruction set available on the compilation host processor. For that reason, we recompiled the code on each available architecture. The instruction set switches specific to each test node are listed in Table 2.

Other executables have also been generated, one with vectorization optimizations switch off:

```
-O3 -no-vec -no-simd
```

and one with debug symbols to facilitate analysis with Intel VTune,

```
-O3 -xHost -g -no-ipo -debug inline-debug-info
```

In order to prepare the code to run efficiently on KNL we first identified with the Intel VTune profiler Bifrost hotspots, i.e. code regions in the application that consume a lot of CPU time. We focused mainly on improving auto vectorization in routines from the `mhd_mod` and `stagger_mod` modules in the code (`zdn_set`, `ydn_set` and `xdn_set` subroutines) following guidance from Intel materials (see e.g. [2] and [3]). Note that such optimization improves not only performance on KNL but also execution time on any modern Intel processor.

3. Vectorization

Knights Landing comes with the support for 512-bit vector instructions (Intel AVX-512). Vectorization can improve performance by executing a single instruction on multiple data (SIMD), e.g. 16 single precision floating point numbers. Taking advantage of vector instructions can be done enabling automatic vectorization by the compiler. The AVX-512 instruction set includes gather/scatter support for data in non-contiguous memory locations, and new prefetch instructions for prefetching data into level 1 and level 2 cache. Adding hints through directives may assist the compiler in generating vectorized code. Directives added in the Bifrost code are:

```
!DIR$ ATTRIBUTES ALIGN : 64
```

Align data on 64-byte boundaries. Data alignment is a method to force the compiler to create data objects in memory on specific boundaries. This is done to increase efficiency of data loads and stores to and from the processor. For KNL, memory movement is optimal when the data starting address lies on 64 byte boundaries. In addition to creating the data on aligned boundaries, the compiler is able to make optimizations when the data is known to be aligned by 64 bytes. This is normally done in Fortran by adding the `!DIR$ ASSUME_ALIGNED` directive so that the compiler can generate optimal code, but Fortran module data also receives alignment information at `USE` sites. In our Bifrost modification we relied on this exception, thus only adding `ALIGN` attributes to major arrays in modules leaving further optimization to the compiler.

```
!DIR$ IVDEP
```

Help the compiler to vectorize code that the compiler will not auto-vectorize because of assumed vector dependencies

```
!DIR$ SIMD
```

Enforces vectorization of loops

The code has been analyzed for vectorization using compiler options listed in Table 3.

Compiler option	Description
`-vec`	Enables vectorization, enabled with –O2 or higher optimization level
`-qopt-report`	Generate optimization report
`-diag-enable=vec`	Enable diagnostic messages issued by the vecorizer
`-guide-vec`	Generate messages to improve optimizations for vectorization

Table 3. Intel compiler vectorization options.

4. On-pacakage high bandwidth memory

The Knights Landing processor comes with an on-package high bandwidth memory (HBM) called Multi-Channel DRAM (MCDRAM) that can deliver up to ~5x the performance of the DDR4 memory. The MCDRAM can be used in three memory modes, selected at boot time. These are *cache mode*, i.e. as cache for the off-package DDR4 memory, or *flat mode* where the MCDRAM is used as physical address space, i.e. as extra, fast memory in addition to DDR4, or in *hybrid mode* as a mixture of cache and flat memory. When used in cache mode no software source modifications are required. The MCDRAM works as an extra cache between the L2 cache and the DDR4 memory. In flat memory mode the MCDRAM is visible to the OS as a separate NUMA node that can be controlled by the `numactl` tool command. This gives the programmer more detailed control on how to use the MCDRAM. If the total memory footprint of the code is smaller than the size of the MCDRAM all memory can be allocated on the MCDRAM with the command:

```
numactl -membind=<mcdram_id> ./program
```

Use the command `numactl -hardware` to find the *id* of the MCDRAM node. If the memory need exceeds the size of the MCDRAM, `numactl` can still be used to allocate parts of the code onto the MCDRAM using the command:

```
numactl -preferred=<mcdram_id> ./program
```

Data that do not fit into MCDRAM will then be allocated in DDR4 memory. Detailed control over what data to allocate on the MCDRAM can be obtained using the *Memkind* library which is built on top of jemalloc, a C library of memory allocation functions. In Fortran this allows allocatable arrays to be placed in MCDRAM using the `ATTRIBUTES` directive option `FASTMEM`:

```
REAL, ALLOCATABLE :: data_a(:,:), data_b(:,:),…
!DIR$ ATTRIBUTES FASTMEM :: data_a, data_b,…
```

When building the model add `-lmemkind` at the link stage. This enables allocation of the data on the MCDRAM at runtime. Tests have been done running Bifrost in flat memory and hybrid mode in addition to cache mode. For test cases using the `FASTMEM` directive the most heavily used data sets in the code was identified.

5. Hardware threading

The Knights Landing processor supports up to four hardware threads per core. This means the possibility to run up to 256 threads on the KNL-ninja test node. To take advantage of this, applications should try adding an extra layer of parallelism on top of the baseline MPI parallelization. For codes that scale well with respect to memory, i.e. the memory footprint per process decreases when the number of MPI processes increases, this extra layer can be increasing the number of MPI ranks per physical core. Other applications that may not fit into the available node memory when the number of MPI processes increases may benefit from adding a threading layer, e.g. OpenMP, on top of MPI.

For Bifrost an attempt was made to add fine grained thread parallelism into the code using the Intel compiler's ability for auto parallelization, and by adding OpenMP directives. With the automatic parallelization, using the `-parallel` compiler option, the compiler generates multithreaded code for loops in the code that can safely be executed in parallel. For OpenMP, loops and whole array operations were parallelized using the `PARALLEL` construct, e.g. parallelizing outermost loop of nested loops:

```
!$OMP PARALLEL DO
do k = 1, nk
  do j = 1, nj
     do i = 1, ni
       …do work…
     end do
  end do
end do
!$OMP END PARALLEL DO
```

and e.g. parallelizing array assignments using the `WORKSHARE` clause:

```
REAL, DIMENSION(ni, nj, nk) :: A, B, C
REAL :: alpha
...
!$ OMP PARALLEL WORKSHARE
A = B + alpha * C
!$ OMP END PARALLEL WORKSHARE
```

For perfectly nested loops, i.e. loops where all content are in the innermost loop, we have also tested the `COLLAPSE` construct:

```
!$ OMP PARALLEL DO COLLAPSE(3)
do k = 1, nk
  do j = 1, nj
     do i = 1, ni
       ...do work...
```

This means that iterations of two or more loops of a nested loop section are collapsed into one larger iteration loop and thus increasing the total number of iterations that are partitioned between the available threads. Experiments were performed running on *physical* and *logical* cores on KNL using the `KMP_AFFINITY` environment variable:

`export KMP_AFFINITY=compact,1` : assign threads to consecutive physical cores
`export KMP_AFFINITY=compact,0` : assign threads to consecutive logical cores

6. Results

We present most of our results in a form of figures depicting average iteration wall clock time measured in seconds. This allows better comparison between the different runs for the same data set. We divided our comments into subsections to relate our findings with the optimization techniques described in the previous three sections.

6.1. KNL specific optimizations

The Bifrost code data organization which base on structure-of-arrays generally favour vector instruction sets on modern processors today. A performance analysis with Intel Advisor (see Figure 3) revealed that ~60% of the total CPU time is spent in vectorized loops. On the other hand, the remaining 40% CPU time is spent in scalar code which on KNL must be executed on relatively slow cores (1.30 GHz in our test system).

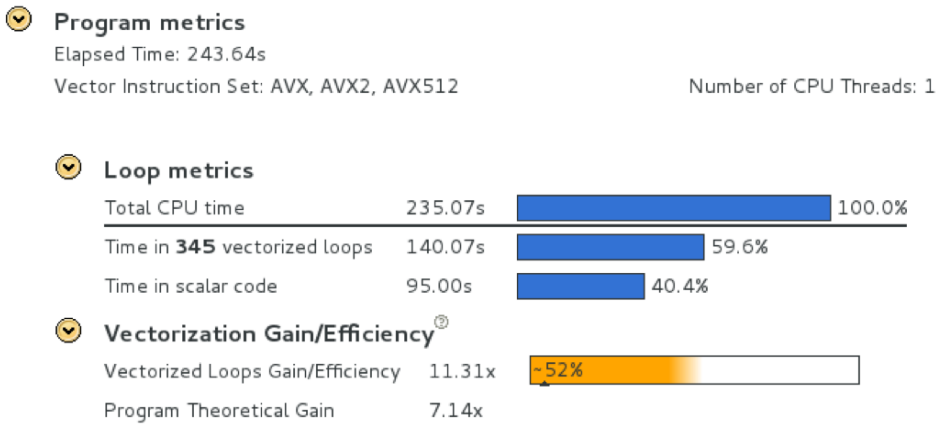

Figure 3. Report from Intel Advisor for the medium size test case run on KNL.

Figures 4 and 5 show average iteration time comparisons for runs with the optimized code, with and without vectorization. On both tested architectures, runs without vectorization are significantly slower when compared to fully optimized versions. The performance loss seems to be size independent, the version without SIMD instructions is ~1.2x slower on Haswell and ~3x on KNL.

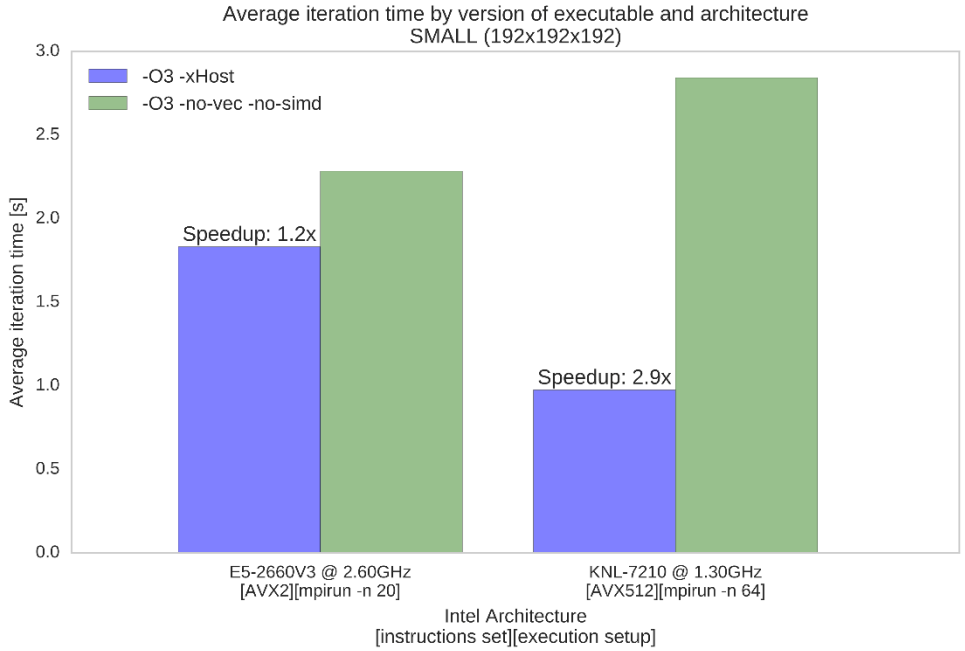

Figure 4. Average wall clock time per iteration for the small test case. Comparison between fully optimized version and one with vectorization switched off by compiling with the '`-no-vec -no-simd`' flags. Speedups indicate performance improvements from the vectorization on a given architecture.

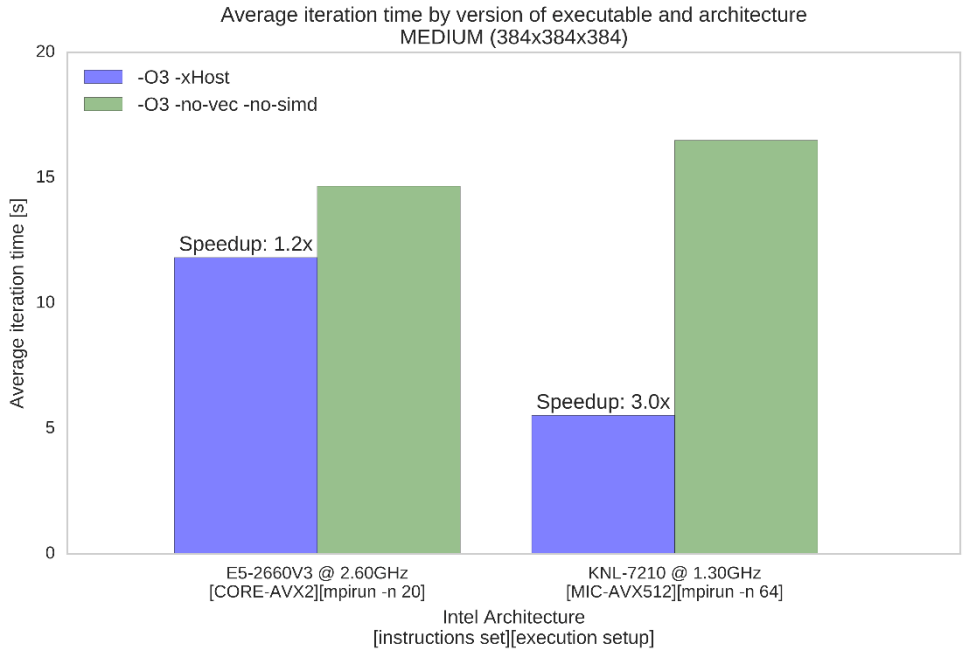

Figure 5. Average wall clock time per iteration for the medium test case. Comparison between fully optimized version and one with vectorization switched off by compiling with the '`no-vec -no-simd`' flags. Speedups indicate performance gain from vectorization on a given architecture.

Additionally, to measure the impact of the AVX-512 instruction set we compiled Bifrost with AVX2 (256-bit instructions) and ran on KNL-ninja with the same runtime configuration. We measured that enabling AVX-512 instructions gives a ~1.3x performance improvement over the 256-bit version, which is significant from a performance perspective.

To optimize memory allocations it may be beneficial to replace standard functions for memory allocation, e.g. `malloc`, with corresponding Intel Threading Building Blocks (Intel TBB) function calls. This can be done by preloading the optimized TBB libraries at run-time using the `LD_PRELOAD` environment variable:

```
export LD_PRELOAD=$TBBROOT/lib/intel64/gcc<version>/libtbbmalloc_proxy.so.2: \
                  $TBBROOT/lib/intel64/gcc<version>/libtbbmalloc.so.2
```

This does not require any source code change or relinking the application. In addition, performance may improve from enabling huge memory pages to reduce TLB misses and page faults. This can be done setting the environment variable:

```
export TBB_MALLOC_USE_HUGE_PAGES=1
```

Figure 6 shows collected timings for different versions of the executable. The results exhibit the importance of the vectorisation on KNL, where CPU frequency is low but SIMD registers are very wide.

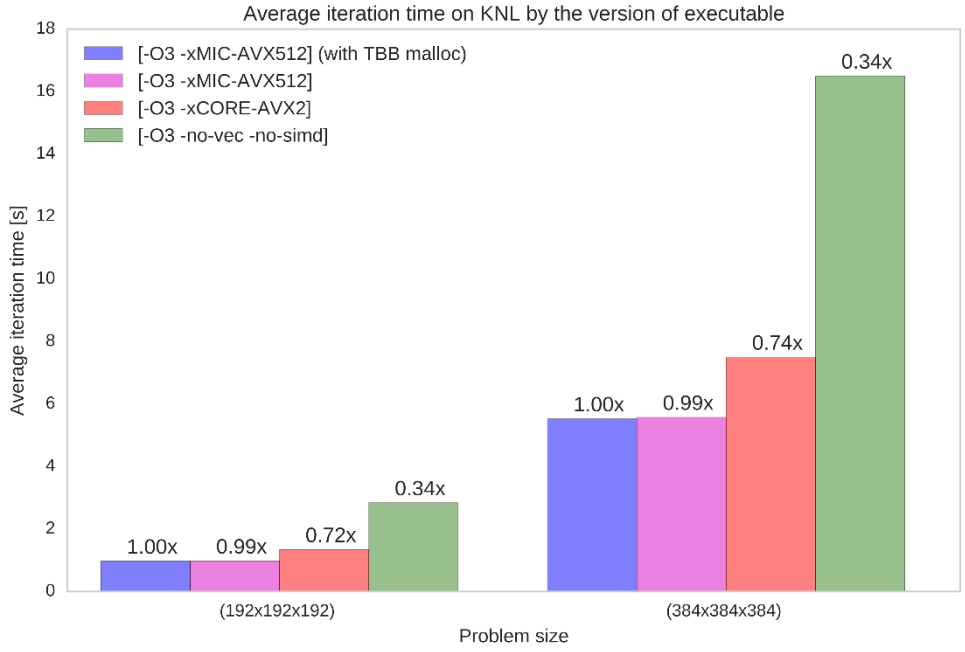

Figure 6. Average iteration time for the same run setup with different vesrions of the executable. Numbers on top of the bars indicates proportional performance loss when compared to the most optimal version.

6.2. *MCDRAM*

For the small test case when running in flat memory mode nearly all the data fit into the 16 GB MCDRAM and using the `numactl` tool gives the best result, see Figure 7. Cache mode also works well since nearly all the data fit into the MCDRAM cache and few MCDRAM cache misses occur. The total size of the selected data allocated on the MCDRAM is probably too small to effectively utilize the MCDRAM. For the medium test case running in cache mode shows better results than running in flat memory mode, see Figure 8. The reason for this is that the total data size of this test case is much larger than the size of the MCDRAM and there will be many cache misses. Even when selecting critical data to allocate on the MCDRAM this is likely too large to fit into the available 16 GB of MCDRAM memory.

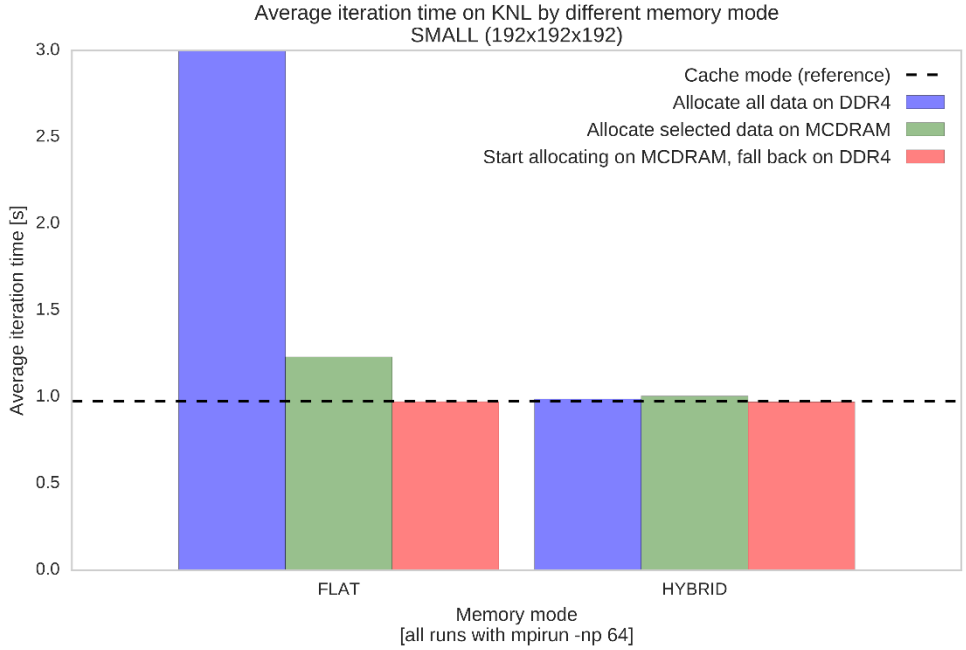

Figure 7. Small test case, 64 MPI processes, MCDRAM in flat and hybrid mode.

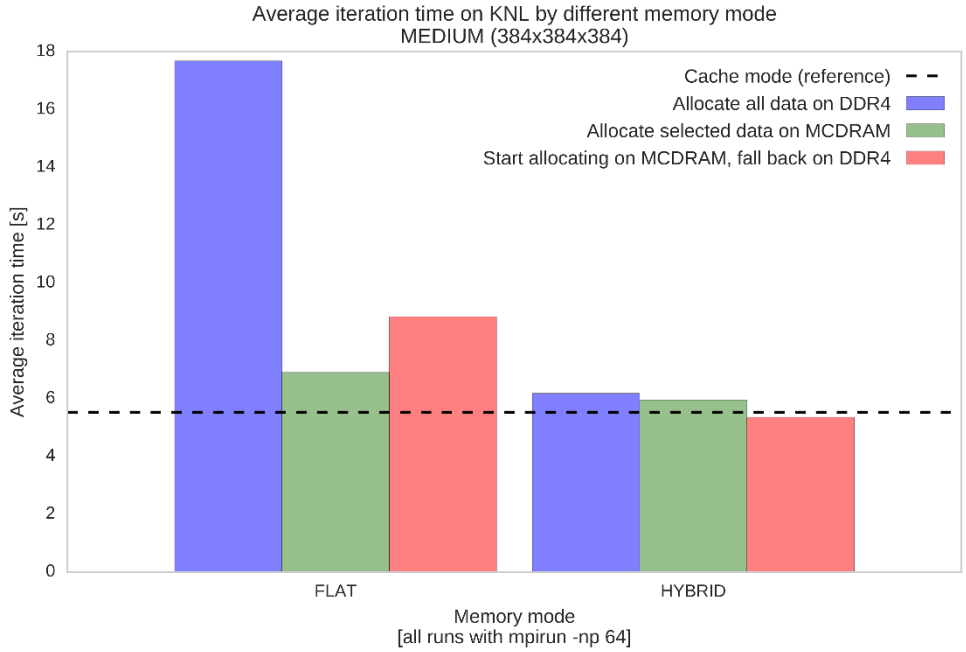

Figure 8. Medium test case, 64 MPI processes, MCDRAM in flat and hybrid mode.

For the hybrid mode test runs the MCDRAM was split in two with 8 GB in cache mode and 8 GB as addressable memory. The effect of the part of MCDRAM used as cache, even though divided in two, seem to be large enough to work effectively, and all results are close to running in full cache mode. For both test cases, the effect of the fast MCDRAM is clear when comparing allocating all data on DDR4 vs. using the MCDRAM as cache or fast addressable memory. Effectively utilizing the MCDRAM shows ~3.1x performance improvement for the small test case and ~3.2x improvement for the medium test case.

6.3. Hardware threading

For the threaded versions of the code, as can be seen from the results shown in Figure 9, there is a speedup when running threads on physical cores (`KMP_AFFINITY=compact,1`) but a performance *deterioration* when running on logical cores (`KMP_AFFINITY=compact,0`). The same behavior can be seen when running Bifrost in pure MPI mode and increasing the number of MPI ranks above the number of physical cores, see Figure 1, or when running in hybrid mode with MPI processes on physical cores and adding threads on logical cores, see Figure 10. The reason for this can be that Bifrost processes consume a lot of resources and the cpu spend most of the time moving data, and since physical cores share L2 cache with the VPU time is lost when two processes or threads are trying to access the same physical resource. Also, Bifrost algorithms have a block structure, i.e. there is a lot of computation followed by blocked communication to update ghost zones. Because Bifrost operate on regular sized grids (cubes, squares, etc.) it is likely that each cpu will get the same data to process. Thus, processes will work in sync and will try to communicate about the same time. As can be seen from Figure 9, running in MPI mode on physical cores performs better than running in hybrid MPI/thread mode.

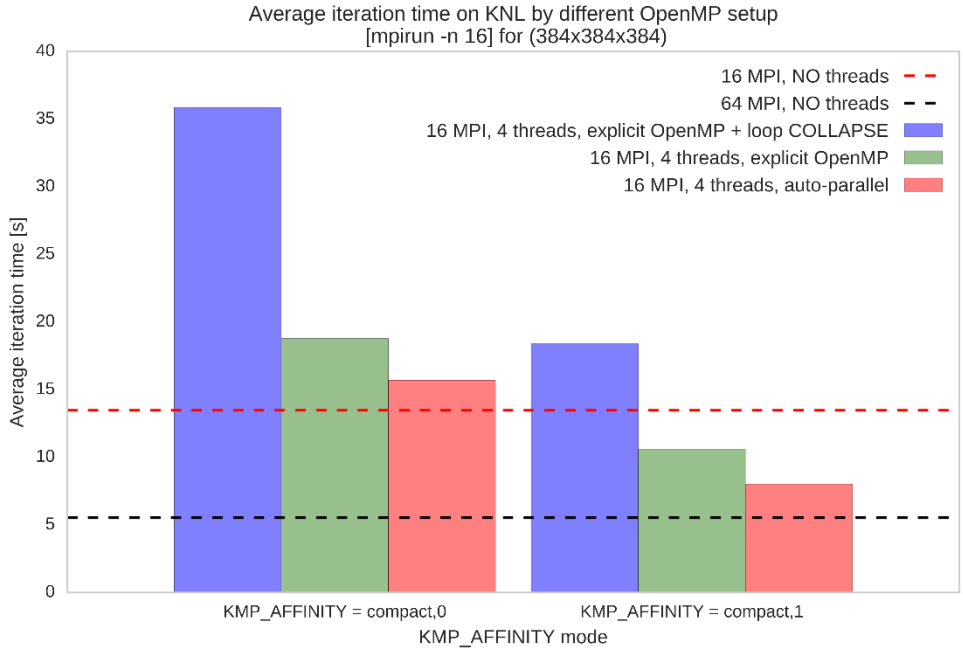

Figure 9. Medium test case, 16 MPI processes with different thread parallelization models.

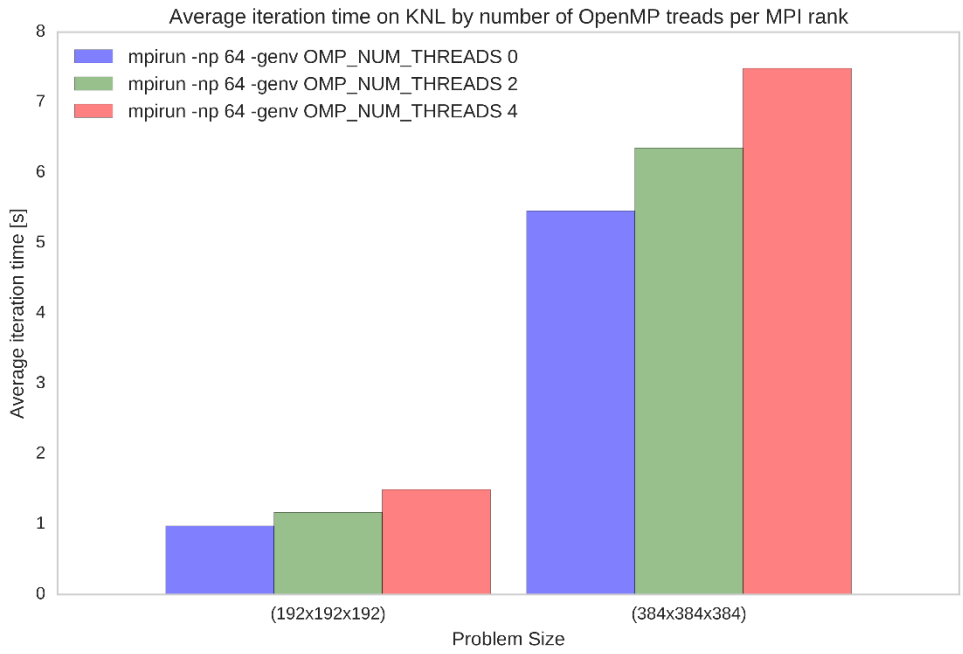

Figure 10. Medium test case, 64 MPI processes, over-subscription effect, auto parallelized version.

7. Conclusions

Undoubtedly the Bifrost code can benefit from the new Intel architecture. Knights Landing does better than Haswell in all our tests because of its higher memory bandwidth (MCDRAM) and superior vector capabilities (AVX-512). However, Bifrost do not seem to utilize the possibility to run many threads per physical core and MPI-only runs exhibits superior performance. Overall we found that running on KNL delivered a ~2x performance improvement over a Haswell node (see Figure 11 for the final comparison).

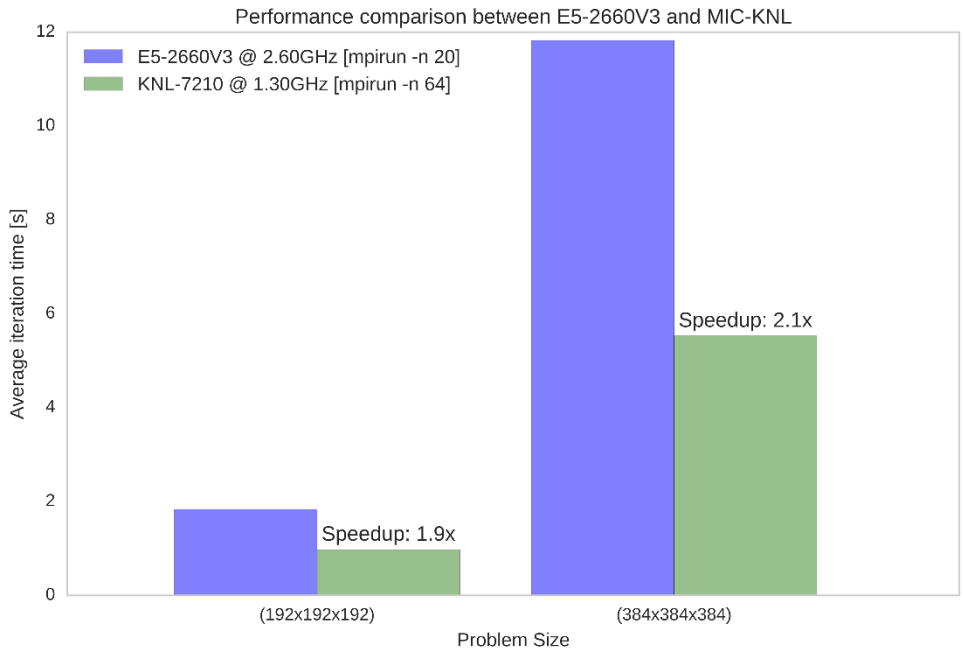

Figure 11. KNL vs. E5-2670v3 speedup

One should note that running the Bifrost code on KNL delivers superior «out-of-box» performance (no major source code changes) when compared to Haswell node. This is partially due to the fact that Bifrost has already been optimized for better in-core (SIMD) parallelism. Furthermore, we found that optimizing Bifrost for KNL also improves performance on other multicore architectures. However, running on KNL introduces a few new problems. Setting an optimal execution setup is not trivial. One has to choose between the different memory modes and look for an optimal domain decomposition.

For our tests we found that Quadrant-cache mode for memory give us the maximum performance with MPI-only run using 64 ranks with one hardware thread per core.

We would like to point out that for the purpose of this report we have tested Bifrost only in its most frequently used configuration where all physics is solved with explicit methods which rely on high-order interpolations on staggered grid. It is not clear however whether the same KNL setup will be optimal for Bifrost runs with additional modules where some selected physics is treated implicitly by the different algorithms and supplementary data structures.

References

[1] Gudiksen et al.,"The stellar atmosphere simulation code Bifrost Code description and validation", Astronomy & Astrophysics Vol.53, 2011
[2] Aart Bik, "Vectorization with the Intel Compilers", Intel Developer Zone, 2012
[3] J. Jeffers, J. Reiders and A. Sodani, "Intel Xeon Phi Coprocessor High Performance Programming (Knights Landings Edition)", Morgan Kaufmann Publ., 2016

Acknowledgements

The authors would like to thank Mikko Byckling and Ole Widar Saastad for their help and countless hints that ultimately led to the success of this project.

M. Szydlarski gratefully acknowledge support by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement 291058.

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838.