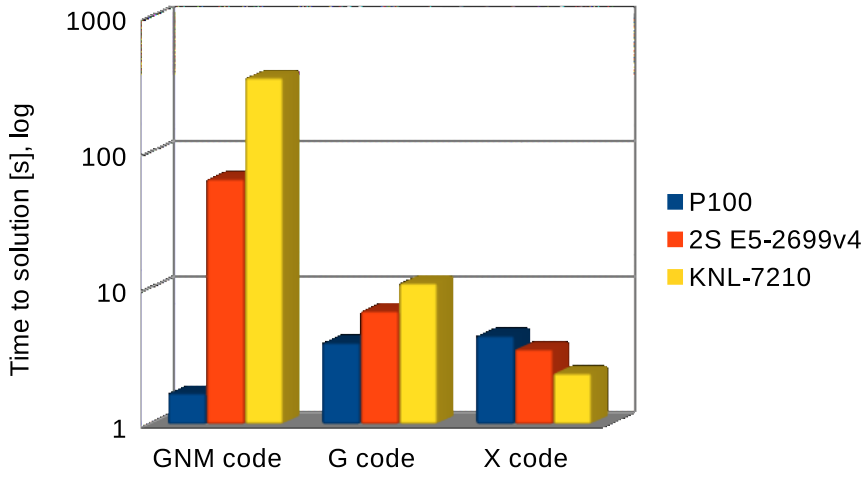
Danish Meteorological Institute
Ministry of Energy, Utilities and Climate

DMI Report 17-22

Tuning the implementation of the radiation scheme ACRANEB2

Jacob Weismann Poulsen and Per Berg



Copenhagen 2017

http://www.dmi.dk/fileadmin/user_upload/Rapporter/TR2017/SR17-22.pdf

Colophone

Serial title:
DMI Report 17-22

Title:
Tuning the implementation of the radiation scheme ACRANEB2

Subtitle:

Authors:
Jacob Weismann Poulsen and Per Berg

Other Contributers:

Responsible Institution:
Danish Meteorological Institute

Language:
English

Keywords:
performance, SIMD, OpenMP, OpenACC, GPU, HPC, exascale, NWP, ESCAPE, Xeon Phi, KNL, NVIDIA P100, roofline, energy efficient computing.

Url:
http://www.dmi.dk/fileadmin/user_upload/Rapporter/TR2017/SR17-22.pdf

ISSN:

ISBN:
978-87-7478-674-0

Version:
1.0

Website:
www.dmi.dk

Tuning the implementation of the radiation scheme ACRANEB2

Jacob W. Poulsen and Per Berg
IT department, DMI
Copenhagen, Denmark
Email: jwp@dmi.dk, per@dmi.dk

***Abstract*—It is not trivial to write code that leads to efficient performance on modern hardware and it becomes even more involved if the performance has to be *portable* and *competitive* across different architectures. This paper describes the work that was done to improve the performance of the radiation dwarf pertaining to the ESCAPE[1] project embracing the well-known IFS and ALADIN-HIRLAM numerical weather prediction models. The overall idea is to demonstrate that the *implementation* of the radiation scheme known as ACRANEB2 can indeed be refactored so that it runs with competitive performance on modern throughput architectures such as the 2nd generation Intel® Xeon Phi™ processors (codenamed Knights Landing™) and accelerator architectures from NVIDIA. We show that the refactored codes run significantly faster on KNL and on NVIDIA P100 than they run on the strongest dual-socket Intel® Xeon system[2] available on the market today. In addition, the refactored code also runs significantly faster than the original code on all the dual-socket Intel Xeon systems used during this study. The parallelism itself is expressed using directive based approaches, OpenMP and OpenACC, respectively. We show that competitive performance is obtained by completely different code bases and hence that performance on a given target architecture comes from the source code within the scope of the directives rather than from the directives themselves. The performance results are presented as *time-to-solution* and *energy-to-solution* and to be fair focus is on comparing performance across hardware released in 2016. The results of the refactored implementations are also related to Moores law and cross-compared with the evolution of the de-facto standard processor benchmarks HPL and Stream. Finally, we show how one have to use phenomenological modeling in order to apply the roofline model in cases like this where transcendental functions are heavily used.**

***Keywords*-Performance, SIMD, OpenMP, OpenACC, GPU, HPC, Exascale, NWP, ESCAPE, Xeon Phi, KNL, NVIDIA P100, roofline, Energy efficient computing.**

I. Introduction

Radiation physics is one of the most time-consuming physics components in NWP today, cf. figure 1. It is an interesting component of the physics due to its intensity in floating point operations which is unusual in NWP models which tend to be bound primarily by memory bandwidth. There is a strong desire from a physics perspective to run radiation physics at every timestep of the model instead of only intermittently[3] as dictated by the computational demands of the current operational production, and this desire increases with increasing resolution but the cost of doing so is simply too high today. This is the overall motivation for choosing the radiation as the physics component in the ESCAPE project. There are several schemes available for radiation today and the scheme chosen for this study is currently used in production setups in the ALADIN[4] community and one that is planned for near-future setups locally where HARMONIE-AROME is used, cf. [1]. The baseline version of this dwarf consists of the upstream ACRANEB2 code extracted from the full IFS code base as a stand-alone application but with loop and index ordering interchanged compared to the upstream implementation. The `SLOC` of the baseline dwarf is around 6.000[5]. The original ACRANEB2 scheme is described in [9] and [6]. Moreover, the upstream data structures and loop nesting is described in the IFS documentation, cf. [5].

The baseline implementation covers multiple radiation options of different complexity, and the original ACRANEB2 algorithm has support for selective intermittency too. This means, that the upstream code includes segments that are more or less frequently visited during a forecast simulation and describes more or less advanced physics, cf. the fourth bar in figure 1. For the present study, however, we have chosen to dive into the implementation of the most computationally expensive part, i.e. we choose the most challenging path through the call tree as seen both from a radiation science and a computer science perspective. This corresponds to the third bar in figure 1.

Throughout this paper we use a 400x400x80 test case, i.e. with 400 points in both horizontal directions and 80 layers in the vertical. This corresponds to the largest test case we could run with the baseline code on a single 64 GB node. The target problem size for real operational, regional models now and in near future has up to about 3 times as many points in each horizontal direction and 65-80 layers; for example, the largest operational setup locally at our institute is 1200x1080x65 at present. The refactored code can easily

[1] http://www.hpc-escape.eu
[2] Intel Xeon E5-2699v4.
[3] In current HARMONIE-AROME configuration the radiation physics is updated only every 12^{th} timestep corresponding to 15 minutes intervals in the model.
[4] http://www.umr-cnrm.fr/aladin/spip.php?article304
[5] as generated using David A. Wheeler's 'SLOCCount'

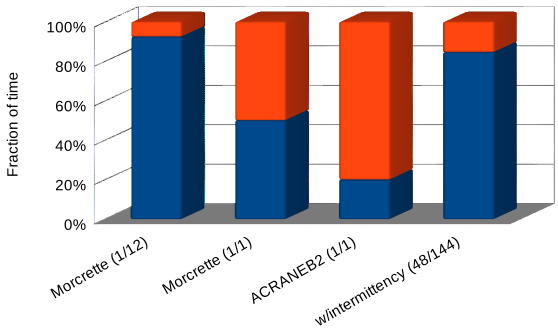


Figure 1. Fractional split of compute time spend in radiation (red) and in all remaining parts (blue) at a node count corresponding to a real production run and with settings corresponding to a real production run. The red area will increase as the number of nodes decreases and decrease as the number of nodes increases. The first bar represents the current state where the Morcrette radiation scheme (see [4]) is only called every 12^{th} timestep due to the computation resources required for each call. The second bar represents the split if the Morcrette radiation scheme was called at every timestep and clearly shows why this is not feasible. The third bar represents the split when the full ACRANEB2 scheme is run at every timestep and finally, the fourth bar represents the split in the ACRANEB2 scheme using the newly developed algorithm with intermittency that allows for a few expensive timesteps and several less expensive timesteps. The total time spend running the new ACRANEB2 with intermittency is more attractive than running the current operational algorithm at every timestep but still more expensive than can be afforded in production runs.

run such cases (not shown in this paper) on a regular 64 GB node. Moreover, the problem as layed out in our approach is embarrassingly parallel hence scaling to more nodes is trivial and will not be considered in this paper.

The paper is organized as follows: First, we define our perception of performance and explain the performance improvement process in general terms in section II. In section III we describe our initial refactoring and with a detailed presentation of the performance model used during the study placed in appendix A. In section IV we present the basic data structures and the parallelization of the code. Section V reveals the performance results obtained on the different target architectures by codes specifically crafted towards performance on each target, and we also describe some further refactorization steps needed to bring the GPU on a similar performance level as the Xeon Phi. Finally, based on our work, we draw some conclusions in section VI and suggest direction for future model development in section VII. Build and run specifications used throughout is placed in appendix B together with some system reference numbers.

II. Performance

It seems reasonable to define what we mean by *performance* and to specify how we can measure it. In this context, *performance* is *time-to-solution* $T2S$, i.e. the seconds it takes to complete a given task. That is, `INPUT` is fixed and `OUTPUT` is fixed by the algorithm itself and the freedom comes solely with the implementation of the algorithm in the source languages as well as in the target ISA and its runtime environment. Thus, we can not allow that results describing the physics are changed as a consequence of our changes in the implementation. Moreover, we will require that all results, both with respect to the physics and more technical measures like time-to-solution and energy-to-solution, are reproducible. Needless to say, we actually rate *correctness* and *reproducibility* higher than performance gain, and we strive towards securing these properties at all times. Of course, results might change numerically due to e.g. choices of different math libraries, use of SIMD reduction instead of scalar reduction, etc., but we always verify that we obtain identical results from one code release to the next, also across platforms, by performing ”safe math” experiments. We also verify measurements like timings by repeating the experiment many times.

Thus, *performance tuning* is a process where we tweak the implementation and its build and run environment in ways that allows us to benefit most from the silicon provided by a given architecture vendor, keeping the results fixed. We illustrate this in figure 2 where we seek an implementation I within one of the two circles in the subset of the left hand side that with a given set of build (b) and run-time (r) environment will attain $\inf_{I,b,r}\{T2S(r(b(I)))\}$ for target 1 and target 2, respectively. The figure also hints that the idea of *portable performance is a contradiction in terms* and we will elaborate further on this in section V. The real challenge, however, is that the infimum is not known beforehand and the tuning process will consequently attempt to take steps that will make $T2S$ decrease until one runs out of ideas or there is no more time to improve it further. Performance modelling is very useful in setting reasonable expectations and guiding this process, cf. appendix A.

The target architectures that we aim at in this study have many similarities from an abstract point of view (see e.g. [8]) and this allows for a portable strategy towards optimization of the implementation. However, they are *not* identical and the devil is in the details. Eventually, one path will improve performance further on one architecture but will impair the performance on another. This is an important fact that requires special attention when one tries to compare performance across different platforms. The fact that the strategy towards optimization has many similarities makes it very tempting to approach the refactorization task using a classical computer science approach with abstraction layers etc. We tend to believe that this is a wrong approach for legacy codes like the one considered here since the restructuring required is simply too involved and there are no easy routes but analyzing the entire implementation line by line if the goal is to seriously improve the performance. We tend to think of the continued improvement process as depicted in figure 3. Imagine that you first shrink the algorithm representation to a minimal amount of memory transfers. Then the refactorization will attempt to organize

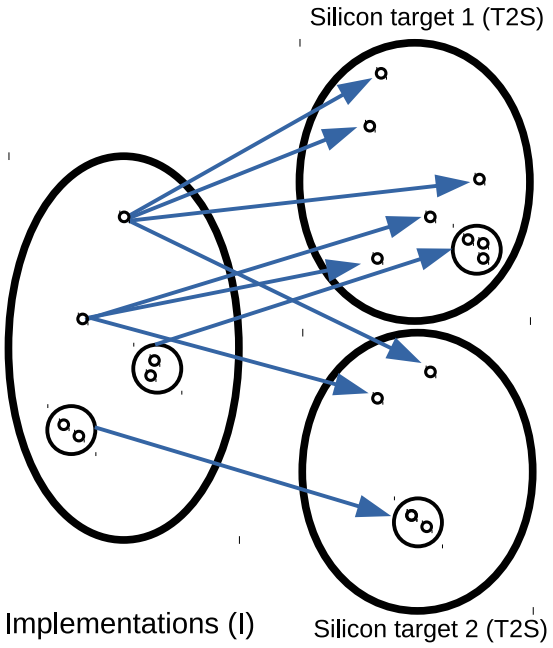


Figure 2. Implementation choices. The left hand side illustrates the set of all possible implementations of a given algorithm with total freedom in the choice of programming languages, parallelization models, etc. The two subsets on the right hand side illustrate the generated codes for two specific targets as a result of the implementation itself (I), the build and link instructions (b) and the run-time environment (r). The aim is to reach infimum; circles show that this is not attained by a unique combination.

all the loops such that parallel exposure is maximized while keeping the temporary memory overhead in storage and in transfers due to the implementation as low as possible. Eventually, trading additional memory transfers required for further splitting of the loops will not out-weight the benefits of added parallelism and the process will stop. This turning point differs from one target architecture to the next. Hence, the tuning process is much like the famous banana problem.

Worldwide, NWP codes are being refactored towards performance on the modern throughput architectures, cf. [7]. Since different versions of the source codes optimized for different target architectures are needed and when even different generations of hardware are considered, a fair comparison of performance results is a challenge and can often be misleading. We will keep this issue in mind when presenting performance results in section V.

III. Refactorization Steps

The initial optimization work aimed at ensuring a proper threading of the code. We used our usual SPMD approach to complete this, cf. [10]. This required a transition to Fortran-

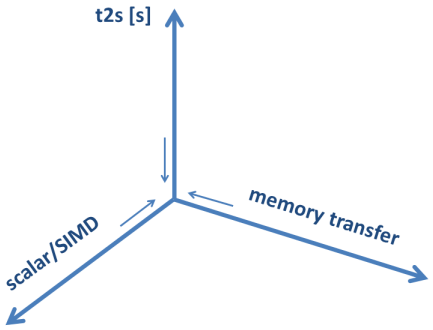


Figure 3. The tuning process. The aim is reducing $T2S$ as much as possible. Extra memory transfers need be traded for more SIMD vectorization. Splitting into more sub-loops implies increased temporary storage to provide an interface between these which again implies extra memory transfers that could have been handled in registers or at least in short latency cache parts.

90 assumed-shape interfaces and that the stack memory usage was trimmed considerably. The primary elements to the refactorization process were ensuring contiguous data; reduce overall stack pressure by turning local temporary 2D/3D variables into 1D/2D variables and even in a few cases into scalars by aligning computations properly such that temporary storage could be reduced or even omitted completely; the largest stack arrays was moved to the heap; proper NUMA-initialization of the heap arrays; collapsing loops over the outermost horizontal index; assuring no side effects in local functions (`pure` in Fortran); constant variables declared as constants (`parameter` in Fortran).

Further refactorization consisted of reducing the memory overhead and of pushing all branching out of the loops such that choices between different physical conditions are made from a top level of the dwarf. From the emerging more bare implementation we began to shuffle computations around to maximize the parallel exposure within each column, guided by recognition of commonly occurring computational patterns, cf. also appendix A. That is, we organized all the vertical loops into sub-loops that had no dependencies and those that did. Sub-loops with dependencies are those that do conventional operations such as prefix-sums and reductions. All sub-loops without dependencies was SIMD vectorized[6] and SIMD tuned, and the ones with dependencies were vectorized if it seemed beneficial, e.g. using OpenMP SIMD reductions. It should be mentioned that prefix-sum operations can indeed be parallelized but the parallel algorithms for prefix-sums do not work well for the trip-counts of relevance to our applications and they were

[6]For complicated loops, this does not happen automatically and one needs to tweak the code to allow the compiler to translate it into efficient SIMD instructions.

left as minimized non-SIMD parallelized vertical loops. Moreover, despite the fact that parallel prefix-sum operations are easily expressed using threads there are currently no directives in the OpenMP specification that allow for such expressions so one would have to express them explicitly. The result of these efforts are summarized in figure 4 and figure 5.

Figure 4 shows the classification of sub-components that resulted from our refactorization efforts. The time spend in the full ACRANEB2 code (dark blue bar) is divided between the thermal radiation scheme, called `transt` (yellow), and all the rest of the radiation physics (red). The `transt` is clearly the most time-consuming part of the full ACRANEB2 code. Diving into the `transt` component in the third bar, we separate that into three parts; a descending part (1), an ascending part (2), and a triangular part (3). The triangular part, which we shall denote by `transt3` in the following, is clearly the most expensive component, corresponding to ∼80% of the total ACRANEB2 compute time on a dual-socket Xeon E5-26xxv4. The fourth bar shows our final classification of the triangular part into a tiny preparation loop, a relatively expensive prefix-sum loop, a huge loop with high arithmetic intensity and referred to as the fat loop from now on, and a collection of smaller SIMD loops and non-SIMD loops.

Inside the fat loop, a large number of simple mathematical operations and transcendental functions as shown in table I are executed and 33 memory transfers of double precision data are performed. Since it is a triangular double nested loop the total trip count is $((81 * 80)/2 = 3240$ in our 80 layers test case for each horizontal point so the total FLOP-count becomes very large for this part of the code.

The cost of splitting the `transt` component into these sub-components is that the preparation loop must be repeated in each of the three parts (1), (2) and (3). However, this preparation loop was straightforward to SIMD vectorize and as a result time spend here was brought down to an insignificant contribution in the full context as indicated by the very thin slice which can hardly be seen on top of the fourth bar in figure 4. This is indeed well spent since it serves for preparing coefficient arrays for the more involved loops that follow, which can then concentrate on doing the work they are meant to do.

Figure 5 shows that our general refactorization efforts already look very promising on KNL, i.e. it was sufficient to SPMD thread parallelize the computations over the columns and SIMD vectorize over the vertical layers within each column in order to obtain competitive performance when running the complete dwarf on KNL. The actual time portions of the individual sub-components are slightly different on Xeon Phi than on Xeon: As expected, the non-SIMD vectorizable parts become relatively more expensive on Xeon Phi supporting AVX-512 than on Xeon supporting AVX2, and this appears as the blue `rest` part that has not been included in our major refactoring at all and in the green `transt3_rest` part that has been refactored but still contains prefix-sum and reduction patterns.

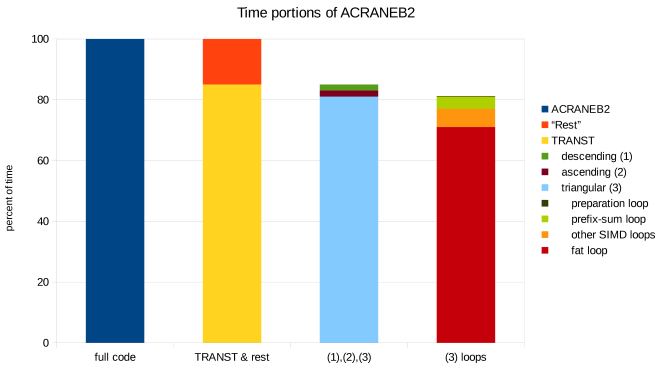


Figure 4. Classification of the main loops resulting from our initial tuning analysis. The bars indicates the portion of time that is spend in the respective code fragments on Xeon. See text for explanation.

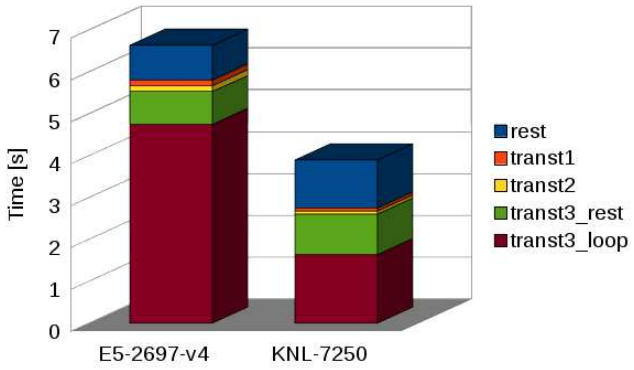


Figure 5. Time-to-solution when running the complete ACRANEB2 dwarf on a dual-socket Xeon (72 threads; left) versus single-socket Xeon Phi (272 threads; right). There is almost ∼2x difference in the overall performance but there are individual performance differences seen in individual sub-components.

The initial refactorization efforts allowed us to simplify the dwarf and confine our focus to the `transt` kernel and the `transt3` kernel with a `SLOC` around 1600 and 700, respectively. These kernels have been ported and tuned to the target throughput architectures (Intel Xeon Phi and NVIDIA

Table I
COUNT OF OPERATIONS/FUNCTIONS INSIDE THE FAT LOOP.

operation/ function	count
max	24
add	454
mul	308
div	48
sqrt	18
exp	14
log	8
pow	22

GPUs) of the ESCAPE project and have been evaluated against various SKUs from the Intel Xeon E5-26xxv4 series of CPUs released in 2016.

One of the most important steps during the continued refactorization process was to reorganize the loops such that the fat loop got a constant trip count. That is, in our 80 layers test case, for the triangular double nested loop with 80 iterations of varying trip count from 1 to 80, we paired short and long loop lengths and thereby obtained a constant trip count of 81 in the inner most loop but only half as many iterations.

IV. Data structures and parallelization

Figure 6 shows the data-structure layout from the upstream code as documented in the IFS documentation, cf. [5], whereas figure 7 shows the new data-structure layout that we have mainly focused on in this paper. The upstream IFS thread parallelization is as shown in figure 8 done over the horizontal with a block granularity of tunable size `nproma`. Our new thread parallelization, shown in figure 9, is done over the horizontal too but with the fine granularity of a single horizontal point. It is important to stress that `nproma`=1 is *not* the same as a granularity of a single horizontal point. Both thread parallelization approaches are done using outlined constructs in order to minimize synchronizations costs and thus allowing the threads maximum freedom for parallel work. The refactorization can be summarized as

- a significant reduction in the thread-local stack pressure
- a more fine grained thread parallel decomposition unit
- full exposure of yet another dimension of parallelism in the algorithm itself

The first item allows us to run far more threads simultaneously without hitting stack limits; this is a *necessary* condition that must be met if one wishes to scale the runs to many threads. The second item allows a better load balancing between the threads and the importance of this again increases at scale. The third item which carefully exposes the vertical parts that have no dependencies and hence can run in parallel from those that have dependencies is another *necessary* condition for running on highly parallel architectures, i.e. all parallelism inherited in the scheme must be explicitly exposed to the compiler. Finally, the size of the sub-chunks with dependencies have been minimized to allow for as much parallelism as possible.

At this point it seems reasonable to consider if we could benefit from reintroducing the blocked `jlon`-approach allowing the non-SIMD innermost sub-loops to SIMD vectorize by re-interchanging the loops. Figure 10 is an attempt to integrate our improvements with the upstream data structures assuming that the complete interchange of array indices is too time-consuming to do for the whole physics code base at once and that one therefore in practise must do the refactoring component by component. The cost of this integrated approach compared to our proposal in figure 7 is a more bulky thread-local stack frame. Needless to mention this transition will come at a cost of higher thread stack pressure so it would only work well up to a certain size. With fewer threads this may not be an issue but as the number of threads increases so does issues related to this overhead, making it a competitive candidate on multi-core architectures with relatively few threads per node but less attractive on modern many-thread architectures. Thus, will we benefit from trading the overhead introduced with the added stack pressure with that of faster computations in the small loops that cannot be SIMD vectorized with the new data layout? This is an open question that we will address in section V.

```
subroutine foo_orig(...,jup,jlow,klon,klev,...)
  ! arguments a*
  real(kind=jprb), intent(in)    :: a1(klon)        ! size klon
  real(kind=jprb), intent(in)    :: a2(klon,klev)   ! size klon*klev
  real(kind=jprb), intent(inout) :: a3(klon,0:klev) ! size klon*(klev+1)
  ...
  ! local variables l*
  real(kind=jprb)                :: l1(klon)        ! size klon
  real(kind=jprb)                :: l2(klon,klev)   ! size klon*klev
  ...
  ! typical loop nest
  do jlev=0,klev     ! vertical loop with loop-carried dependencies
    do jlon=jlow,jup ! no loop-carried dependencies
      ...
      a3(jlon,jlev)= ...
      ...
    enddo
  enddo
  ...
end subroutine foo_orig
```

Figure 6. Fragment of the original ACRANEB2 code using Fortran-77 fixed-size dummy argument declarations implying that the actual arguments must be contiguous in memory. If the actual argument is not or might not be contiguous, the semantics of the language will force the compiler to copy the actual argument array to a contiguous temporary array and back upon return. The innermost `jlon`-loop will be SIMD vectorizable by definition and this holds for all physics subroutines whereas the outermost `jlev`-loop often will suffer from loop carried dependencies. Note the artificial memory overhead for all stack variables `l1,l2,...` at this point in the call-tree and beyond imposed by this way of implementing the loop nests within the physics.

V. Performance results

We confine ourselves to present the *performance* attained on the reduced `transt3` kernel. We have verified (not shown here) that the results and the timings for the `transt3` component are the same if we perform measurements on this reduced `transt3` kernel or on the more involved `transt` kernel or on the full `acraneb2` dwarf, so that there is no need to complicate things more than necessary. Table II lists the architectures and SKUs used in this study, and throughout this paper we shall use the abbreviations shown in the first row of the table.

Table II
List of architectures

	SNB	**BDW**	**KNL**	**P100**
μ-arch	SandyBridge	Broadwell	KnightsLanding	Pascal
Released	2012	2016	2016	2016
SKUs	E5-2680v1	E5-2697v4 E5-2699v4	7210 7250	P100

```
subroutine foo_new(...,jup,jlow,klon,klev,...)
  ! arguments a*
  real(kind=jprb), intent(in)    :: a1(:)    ! size klon
  real(kind=jprb), intent(in)    :: a2(:,:)  ! size klev*klon
  real(kind=jprb), intent(inout) :: a3(0:,:) ! size (klev+1)*klon
  ...
  contiguous :: a1,a2,a3
  ...
  ! local variables l*
  real(kind=jprb)                :: l1       ! size 1
  real(kind=jprb)                :: l2(klev) ! size klev
  ...
  ! typical loop nest
!$acc parallel                                                    &
!$acc present(...)
!$acc loop gang private(...)
  do jlon=jlow,jup  ! no loop carried dependencies
!$acc loop vector
    do jlev=0,klev  ! vertical sub-loop with no loop carried dependencies
      ...
      a3(jlev,jlon)= ...
      ...
    enddo
!$acc loop seq
    do jlev=0,klev  ! vertical sub-loop with loop carried dependencies
      ...
      a3(jlev,jlon)= ...
      ...
    enddo
  enddo
  ...
end subroutine foo_new
```

Figure 7. Fragment of our new ACRANEB2 code using Fortran-90 assumed-shape dummy argument declarations and with interchanged loop ordering. The bulk of the computations in the innermost `jlev`-loop are SIMD vectorizable but some are not as shown here. There are no column dependencies so the outermost `jlon`-loop is thread parallelizable by definition and each thread will handle its own contiguous chunk of the global loop over all columns. Note that the artificial stack overhead caused by the original loop nest ordering is completely gone now.

```
program bar_old
  ...
!$omp parallel do schedule(dynamic,1) private (jkglo,ibl)
  do jkglo=1,kgpcomp,nproma
    ibl=(jkglo-1)/nproma+1
    call foo_old(a1(1,1,ibl), ....)  ! F77-style
  enddo
  ...
end program

subroutine foo_old(a1,jlow,jup,klev,nproma...)
  real(kind=jprb),intent(inout)  :: a1(nproma,0:klev)
  ...
  real(kind=jprb)                :: l1(nproma)        ! size nproma
  real(kind=jprb)                :: l2(nproma,klev)   ! size nproma*klev
  ...
  do jlev=0,klev     ! vertical loop with loop carried dependencies
    do jlon=jlow,jup ! innermost loop without loop carried dependencies
      ...
    enddo
  enddo
  ...
```

Figure 8. Fragment of how threading is implemented in upstream IFS and HIRLAM-ALADIN codes. Unfortunately, the dwarf that we received for this study did not have surrounding OpenMP loop and comparisons with the original code beyond one core should therefore be treated with care.

A. Time-to-solution results

Figure 11 summarizes the best single node, core and thread performance we attained on different Xeon and Xeon Phi systems. It is seen that the **weakest** KNL significantly outperforms the **strongest** dual-socket BDW at the node level with our refactored code. The fact that KNL at all comes close to BDW even at the core level is due to the strong SIMD parallelism that has been achieved as part of the refactorization of the implementation. The boxes in the same figure show the result of 4 years of Moores law by cross-comparing a dual-socket SNB in the high end of the SKUs released in 2012 with that of a dual-socket BDW and single socket KNL which both emerged in 2016. Note that there is a remarkable improvement as a result of Moores law even at the core and thread level for our refactored code.

Figure 13 shows time-to-solution for two different refactored codes on three platforms, BDW, KNL and P100. The code refactored for the GPU target is not really suited for the Xeon/Xeon Phi target, a property we shall come back to

```
program bar_new
  ...
!$OMP PARALLEL DEFAULT(shared)
  call acraneb2_numainit(...)
!$OMP END PARALLEL
  ...
!$OMP PARALLEL DEFAULT(shared)
  call acraneb2( ....)
!$OMP END PARALLEL
  ...
end program

subroutine acraneb2(...)
  ...
  call domp_get_domain(...,jlow,jup) ! get thread bounds
  ...
  do jlon=jlow,jup    ! chunk of horizontal loop handled by this thread
    do jlev=0,klev    ! innermost vertical loop
      ...
    enddo
  enddo
  ...
  call acraneb_subr(jlow,jup,...)  ! thread local calls to subroutines
  ...
end subroutine acraneb2
```

Figure 9. Fragment of how threading is implemented in our new code. Note that it is designed such that the load can be balanced among the threads based on the local properties of ACRANEB2 (or other properties that one might wish to expose to the implementation) through the `domp_get_domain` call. A straight-forward balancing would simply distribute the `jlon`-iterations evenly among the threads but for some physics component this could give rise to ill-balanced load. Thus, the design allows for flexible hooks to balance the load.

```
subroutine foo_nproma(...,jup,jlow,klon,klev,...)
  ! arguments a*
  real(kind=jprb), intent(in)    :: a1(:)           ! size klon
  real(kind=jprb), intent(in)    :: a2(:,:)         ! size klon*klev
  real(kind=jprb), intent(inout) :: a3(:,0:)        ! size klon*(klev+1)
  ...
  contiguous :: a1,a2,a3
  ...
  ! local variables l*
  real(kind=jprb)                :: l1(nproma)      ! size nproma
  real(kind=jprb)                :: l2(nproma,klev) ! size nproma*klev
  ...
  do j=jlow,jup,nproma  ! no loop carried dependencies
    iup = min(nproma,jup+1-j)
    do jlev=0,klev      ! vertical sub-loop with no loop carried dependencies
      do i=1,iup        ! no loop carried dependencies
        jlon = j + i - 1
        l2(i,jlev)= ...
        ...
      enddo
    enddo
    do jlev=0,klev      ! vertical sub-loop loop carried dependencies
      do i=1,iup        ! no loop carried dependencies
        jlon = j + i - 1
        a3(jlon,jlev)= ... l2(i,jlev)
        ...
      enddo
    enddo
  enddo
  ...
end subroutine foo_nproma
```

Figure 10. Fragment of the new ACRANEB2 code using Fortran-90 assumed-shape dummy argument declarations and with all the sub-loop rewrites from `foo_new()` but with the original `nproma`-blocked loop nest ordering inlined into the subroutine itself. All innermost loops are now SIMD vectorizable and the bulk of the outermost loops are SIMD vectorizable too. The block size `nproma` is a tunable parameter that one can used to tune the size of the individual stack-frames for performance.

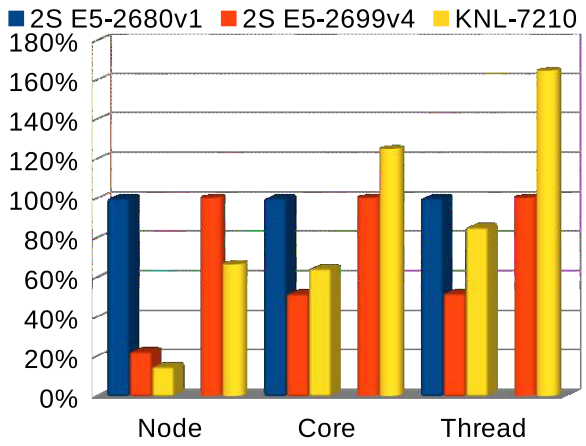


Figure 11. Node, core and thread performance for `transt3` on different Xeon and Xeon Phis for the refactored code. The cylinders cross-compare the performance of the **strongest** dual-socket BDW SKU aka E5-2699v4 (red) with that of the **weakest** KNL SKU aka KNL-7210 (yellow). The boxes cross-compare a dual-socket SNB (blue) with that of BDW (red) and KNL (yellow).

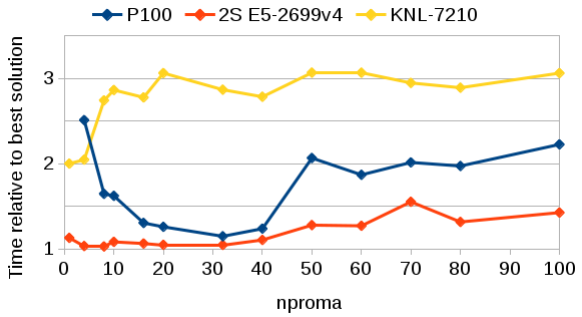


Figure 12. The `nproma` experiment with *time-to-solution* relative to the *time-to-solution* obtained for each of the three platforms with our refactored data organization for varying values of `nproma`. Again, timings are for `transt3`. The GPU timings do *not* include PCI communication.

when discussing portable performance in section V-B.

To answer the question posed at the end of section IV we summarize in figure 12 the result from using the refactored code but retaining the original data-structures and original loop structures and the corresponding tuneable parameter `nproma`, cf. figure 10. All timings from this blended approach are consistently higher than the timings we can attain with our new codes, i.e. those shown in figure 13. The performance loss that comes from the original data organization is significant on KNL. The performance loss is consistent but less significant on the more traditional BDW for all values of `nproma`[7]. This experiment suggests that the traditional data structures and corresponding loop structures in atmospheric models is up for a reconsideration when one targets KNL and even BDW to a lesser extent, though. It is interesting to note that while the conclusion is clear for KNL, the conclusion for the P100 is less clear. Figure 12 reveals a sweet spot for `nproma=32` on P100. It is still 15% slower than the version with the new data-structures but the fact that none of the GPU alternatives so far have shown competitive absolute performance makes the `nproma`-version of the code another good candidate for tuning for the P100. Actually, when we got stuck in attempting to improve the performance on P100 further, we turned our attention to this `nproma`-candidate again and the best GPU result shown in figure 15 in section V-D stems from further GPU tuning of this implementation.

[7]Note that we had to increase `OMP_STACKSIZE` in order to run with the larger `nproma` values

B. *Portable performance*

The source code used for all the targets is Fortran. The baseline code was written in Fortran and the authors have no reason to believe that code generation could be improved by switching entirely to or by combining it with source code written in another programming language. The parallelization, on the other hand, is expressed using the OpenMP programming model when targeting Xeon and Xeon Phi and using the OpenACC programming model when targeting NVIDIA GPUs. According to our experience the GPU does not like to treat larger chunks at the same time since this will lead to data spill, i.e. data that cannot reside in registers will get evicted to global memory and if the corresponding latencies can not be hidden the processor will simply idle. So, for performance on the GPU we need to confine the loops to treat smaller fractions one by one. Moreover, if shared memory is used too then this will also limit the number of thread-blocks that can run concurrently on the device and again be a performance obstacle. All in all this leads to a poor utilization of the available bandwidth, and the GPU will be mostly waiting for data and overall performance will suffer. Thus, the GPU tends to prefer more loop splitting (assuming that the latencies from the additional memory transfers resulting from this can be hidden behind real work) whereas with KNL one would stop the splitting once all SIMD potential is exposed and the caching system is well utilized. Therefore, *competitive performance* can not be portable across very different architectures such as the GPU and the Xeon Phi. The traditional Xeon line, on the other hand, seems to be less sensitive to the number of loop splits compared to Xeon Phi.

In order to treat all targets equal we have decided not to focus on the code resulting from refactoring for the GPU nor for the Xeon Phi solely since as revealed in figure 13 this could have led to too simple conclusions, especially in the case where the refactoring was done for the GPU. The figure also demonstrates that what one could refer to

as *portable performance* can be quite far from *competitive performance* so in weighting the importance of portability versus performance one may sometimes have to choose between a portable layout of the loops resulting in *portable performance* and a less performance portable layout of the loops resulting in *competitive performance* on the primary target platform. However, on the GPU, with a gap of ∼12% the performance of the Xeon targeted code is not too far from that of the GPU targeted code in our case which could guide the choice if one had to stick to a single code version due to e.g. maintenance costs. This would imply that the GPU target become less interesting since the GPU performance is by no means competitive with this source code. In this context it should be stressed that both code versions could be improved further for their respective targets, thus certainly enlarging the gap; this is shown later in section V-D for the GPU target.

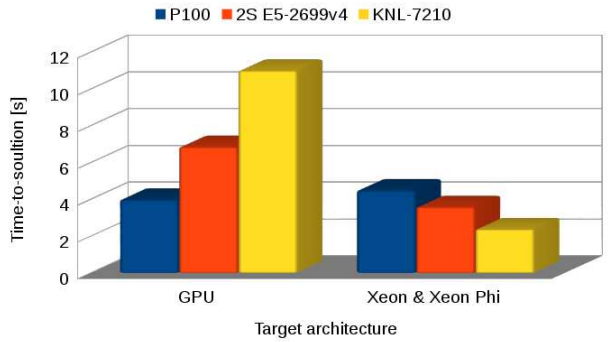


Figure 13. Time-to-solution for two different refactored codes that both retain the same interface on three platforms. The left-hand side shows the performance attained on the three platforms when the code was refactored for the NVIDIA GPU target whereas the right-hand side shows the performance attained on the three platforms when the code was refactored for the Xeon Phi target. Timings are again for `transt3`. Note that section V-D investigates a faster version on the GPU where we allowed the interface to change too. The GPU timings do *not* include PCI communication.

C. Absolute performance

If we only present *time-to-solution* in a relative context as we did in the previous sections then we may cheat ourselves by a poor baseline for performance. Thus, we now turn our attention to absolute performance measures to put the results into a proper context. We used the Intel SDE tool[8] and instrumented the code with an SDE portion surrounding the fat loop within the `transt3` kernel.

Figure 14 shows absolute performance on KNL-7210 for the fat loop. The fat loop sustains approximately 800 GFLOP/s DP and 900 GFLOP/s DP on KNL-7210 and KNL-7250, respectively. Being an absolute measure, we can cross-compare it with other published numbers. For instance, [11] shows that the fastest kernel out of 8 kernels in the NERSC/Trinity benchmark sustains 506 GFLOP/s. In appendix A we will treat the question if sustaining 41%-46% of achievable peak (HPL performance) constitutes a roof or if there is opportunities for improvements. The good absolute performance on KNL translates to BDW too, not directly one-to-one but in the sense that improvements from refactoring for KNL also yields improved absolute performance on BDW. This is the case for NVIDIA P100 too, i.e. efforts on improving for KNL also improved the performance on P100 but as shown in figure 13 this did not lead to competitive performance on P100 nor did the further tunings efforts on this version of the code. A profile on P100 confirmed (not shown here) that the GPU utilization is limited by register usage and each SM is limited to execute only 4 blocks simultaneously. Thus, in theory there is indeed room for improvements on P100 if we can manage to split the computations further and at the same time be able to hide the memory latencies resulting from extra memory transfers required to bind the smaller chunks together. For this code, however, we were not able to improve it in practice despite the theoretical potential. The totally different `nproma`-candidate was much easier to improve for the GPU target as revealed in section V-D.

The algorithm used in this chunk is *compute minimal* in the sense that all computations are necessary and sufficient for defining the output. The algorithm delivers results in two output arrays, O_1 and O_2, and is consequently not considered to be *memory output minimal*. On Xeon and Xeon Phi there is sufficient cache memory available to benefit from computing O_1 and O_2 in one go. On the GPU, on the other hand, the fastest version shown in figure 15 consists of two independent *memory output minimal* chunks, one computing O_1 and another computing O_2. As revealed above this split is not sufficient so further splitting is needed and this will - by definition - introduce additional overhead that has to be compensated for, either by completely hiding this overhead or by exceeding the sustained KNL performance in order to become competitive with the KNL performance.

D. Best performance

As hinted in previous subsections we needed to tune the GPU code variant further to utilize the GPU potential better and achieve competitive performance. Thus, we departed from the `nproma`-candidate and introduced more loop splitting to overcome the obstacles revealed above to create our best performing code for the GPU target. We gained a further ∼1.9 times speedup such that instead of the 4.0 s for the GPU to the left in figure 13 we achieved 2.1 s which is faster than our best timing on the smallest KNL to the right in figure 13. It should be stressed, that when running with this more dedicated GPU code version on Xeon and Xeon

[8]https://software.intel.com/en-us/articles/calculating-flop-using-intel-software-development-emulator-intel-sde. It is our experience that this tool is the most reliable tool to measure the FLOP-counts.

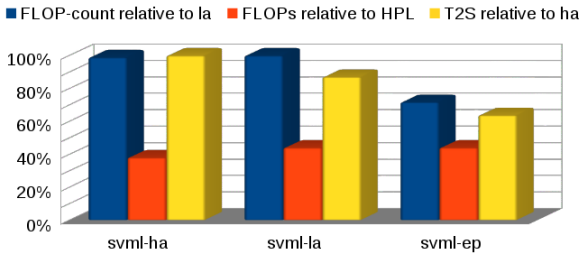


Figure 14. Absolute performance on KNL 7210 using different modes for the MKL vector math library revealing that GFLOP/s is a poor performance measure of performance for the fat loop in this particular kernel. Moreover, we sustain 41%-46% of HPL performance for two of the 3 modes of the MKL vector math library.

Phi the performance suffered seriously on the Xeons[9] to a degree much worse than apparent from left part of figure 13 due to severe cache pollution.

The best node performance that we attained on different architectures released in 2016 is summarized in figure 15. This is a direct head-to-head comparison of our implementations on architectures that one could purchase at the same time. Note that in order to obtain a performance on the GPU that is competitive with the performance on Xeon and Xeon Phi (and vice versa) we need to handle different refactored code versions, but when doing so, performance become almost identical on the largest KNL and on the largest GPU that were available for purchase at the same time.

If one further as an experiment relaxes a little bit on the restriction not to modify the mathematical functions that come with the algorithm developed by renowned radiation physicists, one can replace the power function `x**y` with the mathematically equivalent but numerically different expression `exp(y*log(x))` (in Fortran, that is). The replacement forces a more straightforward implementation which avoids too high local memory usage by the compiler. The result is a further $\sim$1.25 times speedup on P100 which we show as alternative (b) in figure 15. A similar gain can not be achieved on Xeon or Xeon Phi where the compiler and the performance math library (svml-ep) already are doing a similar job. It should be mentioned that in the testcase used here we obtained the same results with the two formulations on P100, but this will generally, of course, not be the case, maybe not even for the range of values that occur in radiation physics, so one should be careful not to draw conclusions too soon; it is out of the scope of the present paper to question the mathematical formulas used in the radiation code.

Note, there is a $\sim$3x between the fastest and the slowest timings on figure 15 if one allows for both transposed data structures and thereby changed interface as well as changed mathematical formulation. It should, however, be stressed that the performance attained is also a function of the algorithm at hand and not just a function of the hardware capabilities. A given algorithm may map better to some architectures than to others and this does not imply that some architectures are better than others. Thus, this figure does *not* imply that best possible performance of *any* algorithm is always almost the same on KNL and GPU. It only shows the status of our work on the various refactorizations of the implementation of the ACRANEB2 algorithm.

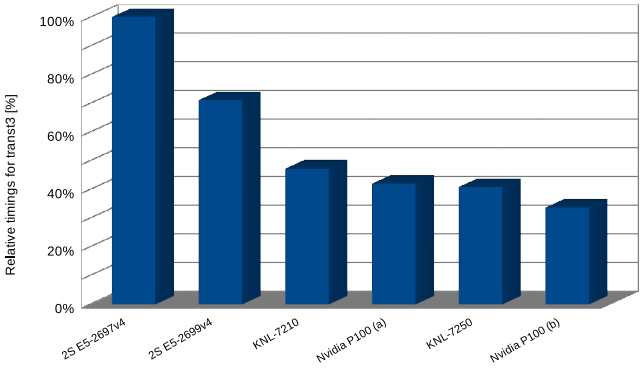


Figure 15. Relative time-to-solution for `transt3` from the best performing code versions on the respective architectures, i.e. this is not portable performance but a result of cross-comparing different versions of the source code, each one explicitly crafted to target an individual architecture. For P100, (a) and (b) are without and with algebraic rewrite of the power function, respectively. The GPU timings do *not* include PCI communication.

E. Energy results

We will now treat performance using the measure *energy-to-solution*. We ran this test on E5-2697v4 and KNL-7250 without turbo mode using 72 and 272 threads, respectively and we ran 500 iterations of the fat loop in order to get sufficient samples for the power measurements[10]. The power was measured using the method described in [3] using the `ISCoL` tool. Table III presents the entire system characteristics, including measured time and power consumption for the fat loop.

The normalized node performance relative to the dual-socket E5-2697v4 is summarized in figure 16 for our refactored code and shows that *time-to-solution* is improved by 2.4x by choosing the KNL over a dual-socket BDW but *energy-to-solution* is improved even more by 2.9x so KNL is indeed delivering more performance per Watt. Thus, our refactored code is more efficient on KNL compared to on BDW than what would be suggested from the HPL performance in table III (HPL ratio 1.57x and EER 2.32x, respectively) both with respect to time and energy.

[9]Timings increased to more than 1 minute on the BDW and 5 minutes on the KNL, cf. left part of figure 18.

[10]This is system power measurements for the entire node, i.e. including both CPU and memory system. Energy is power times time.

Table III
COMPARISON OF TRANST3 PERFORMANCE TO SYSTEM CHARACTERISTICS. PERCENTAGES ARE RELATIVE TO BDW. HPL EER IS THE HPL ENERGY EFFICIENCY RATIO, I.E. THE HPL PERFORMANCE PER WATT, RELATIVE TO BDW. THE TWO LAST ROWS ARE MEASUREMENTS ON THE FAT LOOP.

	BDW	KNL
SKU	E5-2697v4	7250
HPL [GFLOP/s]	1236	1939
HPL [GFLOP/s/W]	2.26	5.24
HPL ratio [%]	100	157
HPL time [%]	100	64
HPL EER [%]	100	232
loop power [W], 500 iterations	4.59	3.71
loop time [s], 1 iteration	3.377	1.428

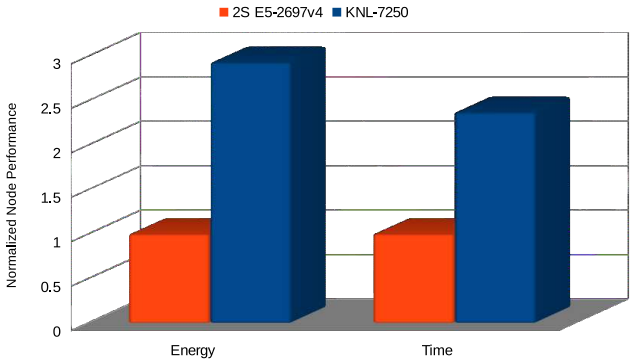


Figure 16. Node performance improvement normalized to BDW. Note that the ratio exceeds the ratio obtained by HPL both in time and energy.

F. Scaling

The 400x400 setup exceeds Amdahl-99.95% strong scaling and it also weak scales perfectly from 400x400 to 1500x1500 on KNL-7210 (not shown here). Thus, up-scaling the timing so that we account for 100% and not just the 80% accounted for by `transt3`, we reach a first crude estimate of the number of nodes needed for running a setup of size 1200x1080x80 which in the horizontal corresponds to the largest setup that we run in production today and in the vertical exceeds the largest setup by 15 layers. This means that 5 to 10 KNL-7210 nodes would be sufficient to run the full ACRANEB2 on this large setup in 0.5 to 1 seconds.

VI. CONCLUSION

Our results suggest that investments in software development and performance maintenance[11] certainly pays off and refactoring of legacy code may have a significant impact on performance on modern hardware. There are multiple arguments as summarized in the following.

[11]We consider a continued effort in refactoring the code to adjust to trends in hardware evolution as a MUST for the daily maintenance of the code. The surroundings are moving, new conditions are being prescribed and one will have to follow in order not to contribute to the technical debt of the project.

First, we may draw the attention to the challenge we started off with, namely that the radiation scheme is a bottleneck in today's operational NWP production, cf. figure 1. There is a vast potential for improving the current implementation as revealed in this paper. Our completely refactored implementation of the most expensive algorithm outperforms the effects of improving it at the algorithmic level, i.e. by adding support for intermittency. This software re-factoring immediately pays off since it allows for doing much more physics under the fixed constrains on time-to-solution and on hardware investment as well as on the energy budget.

Secondly, the importance of our software refactoring becomes even more important on the newer architectures as shown in figure 17. The baseline code was clearly not suited for the modern throughput architectures. To be able to run the baseline code at all on a NVIDIA GPU, we had to do a significant amount of non-trivial code preparation just to ensure the semantics ended up being correctly understood by the compiler. The correctness of this work was verified with the Cray compiler on an older NVIDIA K20x. Further, with this modified baseline code we had to use smaller testcases on the GPUs due to lag of sufficient memory space and up-scale the timings to the 400x400x80 reference. The performance of this GPU-ported baseline code is better when instead the PGI compiler is used with similar performance on K20x (not shown) as on P100, but unfortunately the initial results were also slightly off so the initial preparation steps were apparently not sufficient to obtain portable OpenACC behaviour. On Intel Xeon and Xeon Phi the baseline code ran correctly out of the box. The completely refactored codes gave correct results on all the tested hardware and with all compilers tested across all incarnations (testcase size, thread count, etc) and this includes the OpenACC ports to the GPUs too.

Figure 17 shows that the two 2016 technologies Intel KNL and NVIDIA P100 perform much worse than the 2012 technology (SNB) when we run the baseline code, and the gap is significant with KNL-7210 being $\sim$2 times slower and P100 being $\sim$4500 times slower than a dual-socket SNB from 2012. However, running the refactored code on all platforms reveals a very different picture. Now P100 and KNL-7210 beat SNB by more than a factor of 6. For single core, the baseline code on the 2016 Xeon technology (BDW) beats the 2012 Xeon technology (SNB) by a factor of 1.7, but with our refactored version the factor is more than doubled to 3.8.

Figure 18 is showcasing the difference between portable performance and competitive performance. In this figure code bases X and G (which are also shown earlier in figure 13) are pretty much the same code except for the splitting in G, while code base GNM is essentially a complete re-write with modified data-structures, interface, loop order, reformulation of power function, and on top of that a more

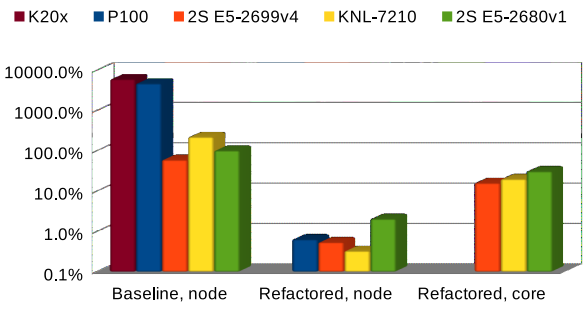


Figure 17. Time-to-solution relative to the baseline implementation on a SNB node for `transt` in the full `acraneb2` dwarf. Note, the vertical axis is logarithmic in order to embrace the range of performance results. The baseline code performance on single nodes of different architectures is shown to the left. Bars in the middle show the single node performance of the refactored codes, and the right bars show the single core performance of the refactored codes. Using the Cray compiler on NVIDIA GPU K20x (brown) and the PGI compiler on P100 (blue), and the Intel compiler on Intel BDW (red), KNL (yellow) and SNB (green). Single-core performance is not sensible for the GPU.

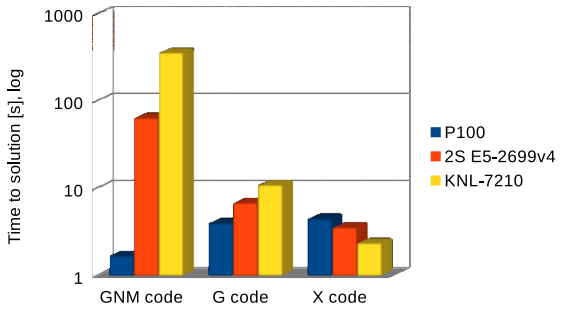


Figure 18. Time-to-solution for the three different code bases on three different architectures. X is the Xeon target code. G is the GPU target code using the data structures as X, but with split into seven chunks. GNM is the GPU target with transposed data structures as compared to X and G, reformulated power function and even more splits (into 12 chunks).

involved splitting. Note, we had to use a log-axis to cover the range of timings.

The obtained gains in performance should be seen in the perspective of how much one can expect from the hardware evolution, and to this end we compare node performance of the `transt3` kernel with HPL and STREAM TRIAD node performance relative to SNB in table IV. For BDW vs SNB the ratios are ∼4.2 and ∼1.6, respectively, and thus with a factor of ∼4.5 our refactored code performs slightly better than expected from the hardware evolution alone. For the smaller KNL-7210 the ratios are ∼5.6 and ∼5.6, and for the larger KNL-7250 the ratios are ∼5.7 and ∼6.2, and our refactored code with ∼6.7 and ∼7.8, respectively, performs significantly out of these ranges which we attribute to the fact that KNL has some of the transcendental functions implemented in hardware[12] and this part of the ISA is not exercised by HPL. Thus, good SIMD vectorization in the code therefore becomes even more important. For P100, the performance improvement is better for our refactored code than for STREAM TRIAD, and including the rewrite of the power function the improvement factor is getting quite close to the high HPL improvement factor, thus utilizing a major portion of the potential performance boost from the SNB to P100 evolution. Note, such improvements as demonstrated in table IV can not be obtained with the baseline code for any of the architechtures, only with the refactored code. Even on the single core the refactorization pays off compared to the baseline code on a full node; this holds for the older SNB hardware too but even more on the newer BDW and KNL. The improvements of the refactoring on newer hardware compared to the older SNB is more than accounted for by increased thread-count times clock-frequency, which demonstrates the importance of proper utilization of SIMD vectorization[13]. Thus, it is evident that our months on refactoring of the code has orders of magnitude higher impact on the performance for this code than 4 years of hardware evolution. It is important to stress that this does *not* prove lack of progress in evolution of hardware but rather it emphasizes the issue with legacy code.

Table IV
NODE PERFORMANCE IMPROVEMENT FACTORS RELATIVE TO SNB. FOR P100, (A) AND (B) ARE WITHOUT AND WITH ALGEBRAIC REWRITE OF THE POWER FUNCTION, RESPECTIVELY.

Architecture	HPL	Stream Triad	`transt3`
E5-2680v1	1.0	1.0	1.0
E5-2699v4	4.2	1.6	4.5
KNL-7210	5.6	5.6	6.7
KNL-7250	5.7	6.2	7.8
NVIDIA-P100 (a)	11.4	6.9	7.6
NVIDIA-P100 (b)	11.4	6.9	9.5

The improvement in time-to-solution due to our refactoring for the entire `transt` code and not only for `transt3` is summarized in table V and figure 17. It is interesting but not surprising to observe that the refactoring has a more significant impact on newer hardware than on older hardware. It is important to stress that the improvements at the node level are somewhat incomplete in the sense that the dwarf that we received was single threaded. It is also important to stress that we did not have time to merge the fastest implementation of `transt3` on P100[14] into the `transt` code; completing this step will bring the refactored node performance for the GPU to be fastest of all the architectures considered in this paper, and in the last row (italicized) in table V we have estimated the corresponding improvement factor for the GPU target.

[12]ISA improvements in SQRT, DIV and AVX-512ER

[13]SNB has AVX with 4 SIMD lanes, BDW has AVX2 with 4 SIMD lanes but also FMA, KNL has AVX-512 with 8 SIMD lanes and FMA.

[14]i.e. from the best performing code version shown in figures 15 and 18.

Table V
REFACTORIZATION IMPROVEMENT FACTOR ON A SINGLE NODE AND ON A SINGLE CORE FOR DIFFERENT ARCHITECTURES.

Architecture	Core	Node
E5-2680v1	3.3	50
E5-2699v4	3.7	110
KNL-7210	11.0	667
NVIDIA P100	N/A	7302
NVIDIA P100	N/A	*17000*

Based on our experience from working with operational met-ocean models, we believe that the radiation dwarf considered in the present paper serves as a typical example with respect to refactoring potential for NWP components, so for entire models we will expect that speed-up in orders of magnitudes can indeed be achieved on modern hardware by a deep refactoring of the entire code. It will, however, take a huge and continued effort to deal with the technical debts inherent in many of these models currently as well as to prevent it from growing further as the hardware trends evolve.

We have also shown that the process of tuning code for different architectures is the same but also that it will diverge eventually and one will end up with completely different code bases in the end. To quantify the differences in the two incomplete attempts of today (one for the GPU target and one for the Xeon target), the relative `SLOC` difference is 50% and the size of the `diff` between the two source files exceeds the size of each of the files. The local variables in the two implementations have different dimensions and the input/output used in one implementation are transposed in the other implementation so even the interfaces differ. In practice, one would consequently have to maintain two code bases despite the fact that we have confined ourselves to the use of directive based approaches. Moreover, we have seen that the latency tuned architectures are less sensitive to where we stop the splitting process and also less sensitive to the choice of loop nest ordering. The highly parallel throughput tuned architectures, on the other hand, are very sensitive to this. Thus, from this particular study we can conclude that *portable performance* is quite far from *competitive performance* and we need to be very cautious when cross-comparing performance obtained on KNL vs GPU. One could have chosen to stop refactoring at the simplest code X in figure 18, claiming that one code base is sufficient, sacrificing competive performance on the GPU for increased portability and maintenance costs. There is already some orders of magnitudes gain in performance on both KNL and P100 using the X target code compared to using the legacy code, cf. figure17, so it might be tempting to stop the refactoring process here. But if performance really matters we would have to discriminate the refactoring. We can certainly *not* expect that we can just decorate the very same code base both with OpenMP and OpenACC directives and then get code generated that will run efficiently on both targets. Numerous attempts on a pure OpenMP and OpenACC directive approach using the very same code base have been made by the present authors and their collaborators, and failed.

Finally, we have seen that refactoring of legacy code *is* indeed required for getting performance out of investment in newer hardware, and with refactored code we can improve *time-to-solution* by choosing one of the new highly-parallel and throughput tuned architectures but we have also seen that on top of this we gain even more performance improvement if the metric is energy. Thus, it seems obvious to us that we have to prepare our entire workload such that it will be able to embrace the future technologies. There is a vast potential in legacy codes that will be revealed when we start to invest in refactorization and we say *when* and not *if* because the latter - to the best of our knowledge - is not a sustainable option.

VII. FUTURE DIRECTIONS

Our refactorization plan follows a pattern that is directly applicable to all the other physics components in IFS and ALADIN-HIRLAM systems too, thus accounting for about half of the total runtime in today's operational NWP models. As we have shown earlier in a previous study, cf. [2], it is indeed possible to refactor the more involved dynamics too and thereby the entire model which would require a new in-depth analysis of the current implementation of dynamics.

It is an open question if it would pay off combining the improved algorithm using intermittency with our improved implementation of the expensive step and reap the harvest from both improvements simultaneously. This will for sure increase maintenance costs and there might not be gain in physics results so the gain in time-to-solution should be significant to justify such an approach.

Our present study also revealed a significant use of transcendental functions and we demonstrated that a straightforward reformulation of the power function could lead to a 20% gain. So, an obvious question would be if one could relax on the mathematical physics formulation by substituting the use of these functions with purpose build Padé polynomials instead to gain even more. Finally, as the resolution scale becomes even finer, it also seems relevant to investigate multi-grid strategies and our colleagues have actually pursued this idea further.

ACKNOWLEDGMENT

The authors would like to thank Ján Mašek, ONPP/CHMI, Kristian Pagh Nielsen and Bent Hansen Sass, DMI for not just providing us with the wrapped dwarf code, the corresponding test cases and input for figure 1 but also for countless discussions on radiation physics. Moreover, we wish to express our gratitude to Karthik Raman, Ruchira Sasanka and Michael Greenfield, Intel, Peter Messmer and

Stan Posey, NVIDIA and John Levesque, Cray for their great support of this study. Special thanks go to Alan Gray, NVIDIA for analysis of the register usage in the `transt3` kernel and for pointing out the advantage of reformulating the power function for the P100. Thanks to numerous people in the HPC and NWP communities for commenting constructively on various early drafts of the manuscript. Thanks to Cray for allowing us to use the Marketing Partner Network system `swan`, to Intel for allowing us to use the `Endeavour` cluster and to NVIDIA for allowing us to use their `PSG` cluster to complete this study. The ESCAPE project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 671627.

APPENDIX

A. Roofline analysis

Roofline analysis is centered around a definition of *operational intensity* and a sometimes naive throughput assumption and it often serves as a valuable tool in guiding code optimization work. For any given implementation I, one may calculate the operational intensity $J(I) = W(I)/Q(I)$ defined as the ratio between the work W to the memory traffic Q. A common metric for work is FLOP-count and a common metric for memory traffic is number of bytes being moved in which case intensity will be the arithmetic intensity denoted AI and measured in FLOP/byte. The naive roofline model uses achievable peak bandwidth B_{max} sustained by the stream triad benchmark and achievable peak performance P_{max} sustained by HPL to limit the performance of I by $P(I) = min(P_{max}, J(I) \times B_{max})$. Measuring sustained performance of $S(I)$ and cross-comparing this with the computed $P(I)$ may sometimes reveal room for improvements for the implementation I at the platform given implicitly by (B_{max}, P_{max}). In cases dealing with fat loops, the I1 instruction cache may be too small to hold the loop and if that happens, then P_{max} will be too optimistic. Moreover, successive iterations of the loop can only overlap by a small relative amount if I is fat and the throughput assumption will not hold true. Instead, the true in-core execution is dictated by the critical path execution time and hence P_{max} will again be too optimistic.

Roofline analysis is useful and reasonably accurate when the implementation mainly contains simple operations that translate directly to hardware instructions, e.g. `ADD` and `MUL`. It is is less suited for comparing different implementations of different algorithms nor is it simple to use in cases where the implementation contains many complicated operations.

For this particular kernel, we have already revealed that the main loop is rather packed with complicated transcendental functions, cf. table I. Nevertheless we will attempt to construct a performance model for the fat loop and use it in the context of roofline analysis.

1) Roofline analysis to guide code refactoring: In section III we described the initial refactoring in kind of a hand-waving way, but one could also describe the process using a more formal roofline-based argumentation. For example, cf. upper part of figure 19, analysis of legacy codes will often reveal an inner loop with a contents which is recognized as a mixture of a non-SIMD patterns and some SIMD patterns. In this particular case, the prefix-sum will prevent SIMD vectorization thus spoiling performance of the entire loop. If n_f and n_b denotes the number of FLOP and BYTEs, respectively, referenced in the function `foo`, the arithmetic intensity of the mixed loop will be $(n_f + 1)/(n_b + 3 * 8)$, assuming 8 byte reals. In the refactored code, cf. lower part of figure 19, the loop has been split into an explicit prefix-sum loop and a SIMD vector loop for the remaining part. The prefix-sum loop has a very low AI of only $1/(3 * 8) \approx 0.04$ but will be able to run at full memory bandwidth, or possibly even directly out of the cache. The AI of the SIMD loop is $n_f/(n_b + 3 * 8)$ which is slightly lower than the AI of the mixed loop, but this is insignificant for performance when n_f is relatively high, i.e. especially when the SIMD loop is fat. What is important here is that this loop will now SIMD vectorize and we can sustain much better utilization of the hardware for the refactored code, even when some portions are inherently non-SIMD friendly.

```
!-  mixed-loop code with a hidden prefix-sum pattern -------
sum = 0.0_jprb
do i=1,n
  sum  = sum + z(i)
  a(i) = foo( sum, ... )
enddo


!-  refactored code w/loop split --------------------------
!
! explicit prefix-sum:
zsum(0) = 0.0_jprb
do i=1,n
  zsum(i) = zsum(i-1) + z(i)
enddo
!
! SIMD vector loop:
do i=1,n
  a(i) = foo( zsum(i), ... )
enddo
```

Figure 19. Sketch of pattern identification and the following loop splitting in a typical refactorization process. The loop-carried dependency in the first loop will prevent SIMD vectorization. It is assumed that the function `foo` has no dependencies or side-effects (`pure function` in Fortran) and that it can be inlined.

2) Establishing a performance model: First, we notice that the fat loop contains a vast number of long-latency operations (`DIV` and `SQRT`), and of transcendental functions (`POW`, `EXP`, `LOG`) with corresponding FLOP-counts being highly implementation-, context- and argument-dependent. So, even if we could translate each operation or function into an equivalent FLOP-count on a given platform we must be aware that the issues like pipelining of instructions, latency, dependencies and argument range may obscure the performance model, making it more crude and maybe even less useful in practice.

Then, we created series of small stand-alone kernels, one

for each operation that is considered. We used the Craypat tool with both the Intel compiler and with the Cray compiler on BDW and KNL to estimate the FLOP-count for each of these kernels on each platform and thereby we are able to translate the results into a representative FLOP-count for each operation. The consistency of this approach was then tested by repeating the experiment using the Intel SDE tool with the Intel compiler on BDW and KNL. The results of these experiments are shown in tables VI - VII. We show FLOP-counts for the simple operations and transcendental functions that appear in the fat loop. In table VI the results are from using the Intel compiler on BDW (upper part) and KNL (lower part) and both the Craypat tool (left) and the SDE tool (right). We have here considered the four MKL variants, i.e. the serial libm and the three vector modes svml-ha, svml-la and svml-ep[15]. In table VII the results are from using the Cray compiler on BDW (left part) and KNL (right part) and the Craypat tool. Also, different math translations are considered through different choices of compiler flag, `-O0` and `-O2`, respectively.

As expected the obtained FLOP-count for the transcendental functions varies with choice of math library. With the Intel compiler, the results obtained with Craypat are very consistent with the results obtained from SDE on BDW but not on KNL; also note here that except for the simplest operations the vector modes of MKL have relatively high FLOP-counts on KNL even for the fast low-accuracy (la) and extended-performance (ep) modes. Moreover, the results with the Cray compiler are consistent with the results with the Intel compiler on BDW (both using Craypat), but on KNL the results differ quite a lot. Finally, it must be stated (not shown) that the FLOP-count obtained in this way is of course heavily dependent on the actual values of the arguments to the functions, and we have here limited ourselves to show only results obtained with some ”representative” argument values.

Note that on KNL, the `DIV` operation is converted to ”`MUL` 1/x” with mode la and ep, and this explains the jump from 1 FLOP to 6 FLOP in the SDE Intel runs. It is expected that something similar but not quite the same happens with the Craypat tool using the Intel compiler (8 FLOP for mode la and ep).

This exercise so forth just demonstrates that it will be necessary to operate with a different set of FLOP-count numbers for functions from different math libraries and that care should be taken before relying too much on these FLOP-count numbers as a basis for code optimization like e.g. in roofline analysis.

Assuming that partial FLOP-count numbers have been collected for each considered operation and function, then it is simply a matter of using these to build a performance model for a loop: Add up the operations weighted by their respective occurrence count. Divide this total FLOP-count by the number memory transfers that you have in the loop to obtain the AI. In our case, as shown in tables VIII - IX, we obtain AI values from ∼6 with svml-ep from Intel MKL on BDW to a staggering ∼170 using low optimization with the Cray compiler on KNL. A typical application would use the more safe math for precision and accuracy studies during testing and development but jump to faster but lower precision (e.g. svml-la reached through the compiler flag `-fimf-precision=medium` for performance runs, and in these cases the AI from our performance model is ∼8-10 for the loop. Arithmetic intensities of this order is very high compared to what is usually seen for loops in NWP models, thus deserving its fat loop label.

3) Tool vs model: The applied tools, i.e. Craypat and SDE, can of course be used to directly measure the FLOP-count for the loop in question. One might even be so lucky that tools can be applied to profile performance of smaller fragments in a real context and obtain reliable results. But, honestly, it is our experience that this is not always the case and we also encourage to treat such measurements with great

[15]https://software.intel.com/sites/products/documentation/doclib/mkl/vm/vmdata.htm

Table VI
FLOP-COUNT EXPERIMENT USING THE INTEL COMPILER.

	BDW, Craypat tool				BDW, SDE tool			
	libm	**ha**	**la**	**ep**	**libm**	**ha**	**la**	**ep**
max	1	1	1	1	1	1	1	1
add	1	1	1	1	1	1	1	1
mul	1	1	1	1	1	1	1	1
div	1	1	1	1	1	1	1	1
sqrt	1	1	1	1	1	1	1	1
exp	19	20	14	10	19	21	14	10
log	25	21	16	12	24	23	19	15
pow	55	64	47	21	55	64	49	22

	KNL, Craypat tool				KNL, SDE tool			
	libm	**ha**	**la**	**ep**	**libm**	**ha**	**la**	**ep**
max	2	2	2	2	1	1	1	1
add	1	1	1	1	1	1	1	1
mul	1	1	1	1	1	1	1	1
div	2	16	8	8	1	1	6	6
sqrt	2	15	15	15	1	14	14	14
exp	88	17	16	11	19	23	22	13
log	66	29	22	21	28	34	28	19
pow	201	77	72	41	55	78	72	40

Table VII
FLOP-COUNT EXPERIMENT USING THE CRAY COMPILER.

	BDW, Craypat tool		KNL, Craypat tool	
	-O0	**-O2**	**-O0**	**-O2**
max	1	1	4	2
add	1	1	1	1
mul	1	1	1	1
div	1	1	2	16
sqrt	1	1	19	16
exp	18	18	130	16
log	19	19	241	29
pow	190	95	1769	195

Table VIII
ARITHMETIC INTENSITY (AI) AND FLOP-COUNT FOR THE FAT LOOP USING THE INTEL COMPILER. PM IS GFLOP/S IN THE LOOP FROM OUR PERFORMANCE MODEL, TM IS GFLOP/S MEASURED BY THE TOOL, DEV IS DEVIATION BETWEEN PM AND TM IN %.

	BDW, SDE tool				BDW, Craypat tool			
	libm	**ha**	**la**	**ep**	**libm**	**ha**	**la**	**ep**
AI	9.7	10.4	8.7	6.2	9.7	10.3	8.4	5.9
PM	41.3	44.6	37.4	26.3	41.5	44.0	36.0	25.3
TM	37.9	43.9	36.2	24.4	38.8	44.0	35.3	23.8
DEV	-8.1	-1.4	-3.1	-7.2	-6.6	0.0	-1.9	-5.9

	KNL, SDE tool				KNL, Craypat tool			
	libm	**ha**	**la**	**ep**	**libm**	**ha**	**la**	**ep**
AI	9.9	13.1	13.2	9.8	26.5	15.5	13.2	10.2
PM	42.2	56.1	56.3	41.7	113.3	66.2	56.4	43.6
TM	38.9	54.1	54.6	40.4	107.9	65.1	55.6	41.7
DEV	-8.0	-3.6	-3.0	-3.1	-4.7	-1.6	-1.5	-4.5

Table IX
ARITHMETIC INTENSITY AND FLOP-COUNT FOR THE FAT LOOP USING THE CRAY COMPILER. PM IS GFLOP/S IN THE LOOP FROM OUR PERFORMANCE MODEL, TM IS GFLOP/S MEASURED BY THE TOOL, DEV IS DEVIATION BETWEEN PM AND TM IN %.

	BDW, Craypat tool		KNL, Craypat tool	
	-O0	**-O2**	**-O0**	**-O2**
AI	20.6	12.7	170.5	25.3
PM	88.2	54.3	729.1	108.4
TM	89.8	51.6	729.1	121.2
DEV	1.9	-5.1	0.0	11.8

care.

The performance model described in the previous subsection is based on measuring the individual FLOP-count for each function, and this may come handy during development when one e.g. tries to implement a new code piece by piece or tries to optimize legacy code, while keeping focus on performance. We do, however, need to verify that this approach is good enough for the specific purpose at hand. We expect that our performance model is crude, but if we should be able to use it in a larger context, our model must still be sufficiently reliable under the conditions described (i.e. system by system, library by library, argument by argument, ...).

We have compared our performance model with results from the tools. Tables VIII - IX show the total FLOP-count in GFLOP/s for the loop in a semi-real context (the test case was tuned to 500 iterations of a 1x10x80 grid configuration in order to satisfy the tools). The overall picture is that our performance model in this case explains pretty much all the FLOP measured by the tools, and most of the runs using the more optimized libraries are within ∼5% (again disregarding the model result from using Cray compiler on KNL which is ∼12% off in the fast version).

4) Roofline analysis for the fat loop: In figure 20 we show some selected results from roofline analysis of the loop in full context using the performance model. The model grid size is 400x400x80 which we were not able to profile in a reliable way with the Craypat tool and therefore we had to stick to our performance model. Since the test case is supposed to mimic a realistic situation we have used the performance library MKL svml-la, but we have for comparison also showed one result using the serial MKL libm. Actually, this serial result is placed higher in the roofline diagram and therefore has a better FLOP/s performance than the results using the vectorized library on the same node and on a smaller BDW node, but does this then mean that the performance number that really matters, i.e. the time-to-solution, also is better?

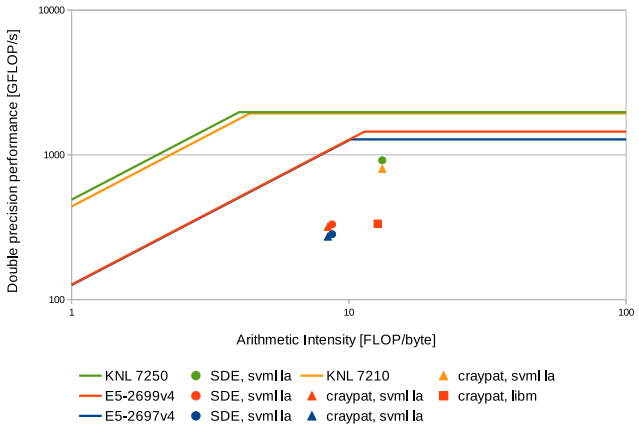


Figure 20. Roofline diagram showing some selected results. Green color is used for KNL-7250, orange is for KNL-7210, red is for E5-2699v4 and blue is for E5-2697v4. Horizontal lines are from the HPL benchmark while the sloping lines are from the STREAM TRIAD benchmark, cf. table X. Markers are results obtained from the performance model using the Intel compiler and the maximum number threads on each SKU, i.e. 272 on the KNL-7250, 256 on the KNL-7210, 88 on the larger BDW and 72 on the smaller BDW. Results from the performance library MKL svml-la are shown as circles using SDE and as triangles using Craypat. The square marker indicates the result from using Craypat and the default MKL libm library.

No, obviously not. In figure 21 we compare the time-to-solution from using the vector MKL svml-la on all four SKUs with that of using the serial MKL libm on the largest BDW. From this figure it is clear that MKL libm on the 88 thread BDW is slower than the rest, a conclusion that can not be drawn clearly from figure 20. This demonstrates that roofline on its own can be of limited use as a performance-measuring tool in these more involved contexts. However, the 272 threads KNL-7250 is best performing according to both roofline and time-to-solution. On the smaller KNL our implementation sustains ∼41% of the HPL performance both with svml-la as shown in figure 20 and with svml-ep as shown in figure 14. On the larger KNL it reaches ∼46%. The AI is 13.2 on the KNLs. On the two BDWs, however, AI is 8.7 which is on the left hand side of the knee of the roofline and thus STREAM TRIAD is the proper measure here; our implementation sustains ∼26% and ∼30% of the STREAM TRIAD performance on the smaller and larger

BDW, respectively.

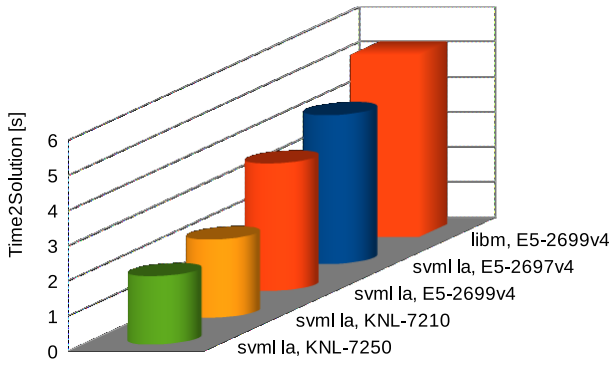


Figure 21. Time to solution for four selected cases using the Intel compiler: The circular bars are from using the performance library MKL svml-la on the four platforms, while the square bar is from using the default library MKL libm on the large BDW. Color of the bars correspond to the color of the markers in the roofline diagram in figure 20.

B. Build and Run specifications

Figure 22 summarizes the build instructions and figure 23 summarizes run instructions in `BASH`-syntax for Xeon/Xeon Phi and NVIDIA, respectively. The `ifort` compiler versions used was `17.0.1.132 Build 20161005` whereas the `pgi` compiler version used was `17.4-0` and the cce compiler was version `8.5.8`. We used `turbo` mode on all SKUs but E5-2697v4. For the KNL systems, this benchmark is so highly flop bound that it is neutral to whether it runs out of DDR or MCDRAM and also neutral to whether we run in flat or cache mode. As for KNL kernel configurations, we found that `CONFIG_HZ_250=y` gave slightly better timings than `CONFIG_HZ_1000=y`.

Finally, table X summarizes the HPL and Stream Triad numbers used for roofline analysis and for evaluations of the absolute performance sustained. The numbers for Intel hardware were received from private correspondence with Intel while the NVIDIA P100 numbers were obtained from a Dell published study[16] and from private correspondence with NVIDIA.

The authors are strong supporters of Nature's theme on transparent and reproducible science and code sharing[17] and welcome anyone to contact us if they are interested in the implementations mentioned in this paper.

[16]http://en.community.dell.com/techcenter/high-performance-computing/b/general_hpc/archive/2017/03/14/application-performance-on-p100-pcie-gpus

[17]https://www.nature.com/polopoly_fs/1.16232!/menu/main/topColumns/topLeftColumn/pdf/514536a.pdf

```
tar -zxvf dwarf-transt3_v<version>.tar.gz
cd dwarf-transt3_v<version>

# bdw/knl
BDW_TARGETF="-xCORE-AVX2"; KNL_TARGETF="-xMIC-AVX512"
TARGETF=<your_choice>
Fep="-O2 $TARGETF -ipo -fimf-precision=low -fp-model fast=2"
Fla="-O2 $TARGETF -ipo -fimf-precision=medium"
Fha="-O2 $TARGETF -ipo -fimf-precision=high"
FCFLAGS=$Fep FC=ifort ./configure --enable-openmp --host=x86_64-linux-gnu
make

# p100
TAF_DEFAULT="-ta=nvidia"; TAF_MAX80REGS="-ta=nvidia,maxregcount:80"
TAF=<your_choice>
F="-mp $TAF -acc -fast -Minline=levels:3 -Mcuda=cuda8.0 -Mcuda=fastmath"
FCFLAGS=$F FC=pgf90 ./configure --enable-openmp --enable-openacc && make
```

Figure 22. Build instructions for reproducing the builds used in this paper.

```
tar -zxvf dwarf-transt3_testcase.tar.gz
cd dwarf-transt3_testcase

#bdw/knl, cray system
export OMP_NUM_THREADS=<threads>; export KMP_AFFINITY="disabled,verbose"
aprun -n1 -N1 -d<threads> -j2 -cc depth dwarf # bdw
aprun -n1 -N1 -d<threads> -j4 -cc depth dwarf # knl

#bdw/knl, non-cray system
export OMP_NUM_THREADS=<threads>; export KMP_AFFINITY="compact,verbose"
dwarf

#p100
export OMP_NUM_THREADS=1
srun dwarf
```

Figure 23. Run instructions.

REFERENCES

[1] Lisa Bengtsson, Ulf Andrae, Trygve Aspelien, Yurii Batrak, Javier Calvo, Wim de Rooy, Emily Gleeson, Bent Hansen Sass, Mariken Homleid, Mariano Hortal, Karl-Ivar Ivarsson, Geert Lenderink, Sami Niemelä, Kristian Pagh Nielsen, Jeanette Onvlee, Laura Rontu, Patrick Samuelsson, Daniel Santos Muñoz, Alvaro Subias, Sander Tijm, Velle Toll, Xiaohua Yang, and Morten Ødegaard Køltzow. The HARMONIE–AROME Model Configuration in the ALADIN–HIRLAM NWP System. *Monthly Weather Review*, 145(5):1919–1935, 2017.

[2] Per Berg, Karthik Raman, and Jacob Weismann Poulsen. Complete HBM model runs on Intel Xeon processors and Intel Xeon Phi processors - part I. Technical report, DMI, Copenhagen, 2016.

[3] W. Michael Brown, Andrey Semin, Michael Hebenstreit, Sergey Khvostov, Karthik Raman, and Steven J. Plimpton. Increasing molecular dynamics simulation rates with an 8-fold increase in electrical power efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 8:1–8:14, Piscataway, NJ, USA, 2016. IEEE Press.

[4] ECMWF. *Part IV: Physical Processes*. IFS Documentation. ECMWF, 2013.

Table X
HPL AND STREAM TRIAD PERFORMANCE REFERENCE NUMBERS.

Architecture	**HPL** [GFLOP/s]	**Stream Triad** [Gbyte/s]	**TDP** (W)
E5-2680v1	343	79	260
E5-2697v4	1278	126	290
E5-2699v4	1446	127	290
KNL-7210	1933	440	215
KNL-7250	1971	490	215
NVIDIA-P100	3900	540	300

[5] ECMWF. *Part VI: Technical and Computational Procedures*. IFS Documentation. ECMWF, 2016.

[6] J.-F. Geleyn, J. Mašek, R. Brožková, P. Kuma, D. Degrauwe, G. Hello, and N. Pristov. Single interval longwave radiation scheme based on the net exchanged rate decomposition with bracketing. *Quarterly Journal of the Royal Meteorological Society*, 143(704):1313–1335, 2017.

[7] Mark Govett, Jim Rosinski, Jacques Middlecoff, Tom Henderson, Jin Lee, Alexander MacDonald, Ning Wang, Paul Madden, Julie Schramm, and Antonio Duarte. Parallelization and Performance of the NIM Weather Model on CPU, GPU and MIC Processors. *Accepted for Bulletin of the American Meteorological Society*, 2017.

[8] Brent Leback, Douglas Miles, and Michael Wolfe. Tesla vs. Xeon Phi vs. Radeon - A Compiler Writer's Perspective. In *CUG Conference Proceedings*, CUG 2013, Napa Valley, California, USA, 2013.

[9] J. Mašek, J.-F. Geleyn, R. Brožková, O. Giot, H.O. Achom, and P. Kuma. Single interval shortwave radiation scheme with parameterized optical saturation and spectral overlaps. *Quarterly Journal of the Royal Meteorological Society*, 142(694):304–326, 2016.

[10] Jacob Weismann Poulsen, Per Berg, and Karthik Raman. Chapter 3 - Better Concurrency and SIMD on HBM. In James Reinders and Jim Jeffers, editors, *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume 1, pages 43 – 67. Morgan Kaufmann, Boston, MA, USA, 2015.

[11] Antonio C. Valles, Chuck Yount, and Sundaram Chinthamani. Chapter 25 - Trinity Workloads. In James Reinders, Jim Jeffers, and Avinash Sodani, editors, *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition*, pages 549–579. Morgan Kaufmann, Boston, MA, USA, 2016.