International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 8, Number 2, pages 301–327, July 2018

# Performance Optimization Strategies for WRF Physics Schemes Used in Weather Modeling

T.A.J. Ouermi
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, Utah, USA

Robert M. Kirby
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, Utah, USA

and

Martin Berzins
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, Utah, USA

Received: February 1, 2018
Revised: April 19, 2018
Accepted: June 4, 2018
Communicated by Susumu Matsumae

**Abstract**

Performance optimization in the petascale era and beyond in the exascale era has and will require modifications of legacy codes to take advantage of new architectures with large core counts and SIMD units. The Numerical Weather Prediction (NWP) physics codes considered here are optimized using thread-local structures of arrays (SOA). High-level and low-level optimization strategies are applied to the WRF Single-Moment 6-Class Microphysics Scheme (WSM6) and Global Forecast System (GFS) physics codes used in the NEPTUNE forecast code. By building on previous work optimizing WSM6 on the Intel Knights Landing (KNL), it is shown how to further optimize WMS6 and GFS physics, and GFS radiation on Intel KNL, Haswell, and potentially on future micro-architectures with many cores and SIMD vector units. The optimization techniques used herein employ thread-local structures of arrays (SOA), an OpenMP directive, OMP SIMD, and minor code transformations to enable better utilization of SIMD units, increase parallelism, improve locality, and reduce memory traffic. The optimized versions of WSM6, GFS physics, GFS radiation run 70, 27, and 23 faster (respectively) on KNL and 26, 18 and 30 faster (respectively) on Haswell than their respective original serial versions. Although this work targets WRF physics schemes, the findings are transferable to other performance optimization contexts and provide insight into the optimization of codes with complex physical models for present and near-future architectures with many core and vector units.

*Keywords:* Parallel, Optimization, Physics Schemes, Numerical Weather Prediction, Structure of Arrays

## 1 Introduction

The Weather Research and Forecasting (WRF) [3] model is a widely adopted numerical weather prediction (NWP) software suite used by atmospheric researchers and weather forecasters at operational centers worldwide. WRF was developed to help scientists study and better understand weather phenomena. Optimizing the performance of NWP codes, such as WRF, is important for improving the accuracy and the time requirements for forecasts. In order to reach desirable resolution and accuracy new mathematical and computational approaches must be developed [1] Therefore there has been significant significant time and efforts invested to modernize NWP codes for current and future computer architectures.

In the last decade, various computational architectures have increased the core counts per node, decreased the clock frequency and adopted wide SIMD vector units. For instance, the Intel Xeon Phi Knights Landing (KNL) [13] has dual 8-lane double precision (DP) floating point units on each of its 64 cores with a clock frequency of 1.3 Ghz. This growing complexity of computing architectures makes it difficult to develop and maintain performance-portable codes. For this reason, codes such as WRF must be restructured to leverage thread and SIMD parallelism on modern architectures while maintaining data and temporal locality.

One example of a modern code written with future architectures in mind is the Navy Environmental Prediction sysTem Utilizing the Nonhydrostatic Unified Model of the Atmosphere (NUMA) corE [7] (NEPTUNE). The NEPTUNE code couples the scalable dynamical core NUMA, of Giraldo et al. [9], with physics schemes such as the WSM6 and GFS schemes considered here. These physics schemes represent physical parametrization of processes that are unresolved at the grid resolution. For instance, WSM6 uses a physical parameterization that simulates processes in the atmosphere that cause precipitation in the form of rain, snow, graupel, water vapor, cloud water and cloud ice. While the dynamics part of NEPTUNE is both fast and scalable [22], a remaining challenge is the performance of the physics routines. For this reason, this work focuses on optimizing GFS and WSM6 using approaches that have application to other numerical methods. The new optimization efforts described here target the Intel KNL [13] and potential future computer architectures that may employ similar multicore architectures that achieve performance through vector units. This work employs OpenMP 4 as a vehicle for portability across various platforms, as OpenMP 4 is a well-established and widely adopted interface for shared memory parallelism.

This paper introduces high-level and low-level approaches for shared memory parallelism using thread-local structures of arrays (SOA). The high-level approach employed here consists of parallelizing large blocks of code at the parent level in the call stack, whereas the low-level approach targets individual instructions. Thread-local SOA and OMP SIMD are employed to accelerate computation in GFS and WSM6 modules by improving data locality and taking advantage of thread and vector parallelism. In addition, a static memory allocation process is used instead of dynamic memory allocation process to help improve memory performance of the GFS code. As result of these optimizations there has been a significant speed-up over serial versions of the code and a previously optimized version [24]. For instance, the use of SOA coupled with OMP SIMD for vectorization led to significant speed-up improvements. The optimized versions of WSM6, GFS physics, and GFS radiation run 70, 27, 23 and 26, 18, 30 faster on KNL and Haswell respectively. In addition, these optimizations enabled a speed-up of 23.3 over a prior optimized version of WSM6 [24].

This paper is an extended version of the conference paper [25]. Sections 2 to 5.2 are revised versions of material in [25] as are the results in Tables 1–6 and Figures 1–5 with minor additions. Section 2 describes related work, and Section 3 gives an overview of the NEPTUNE code. The experimental methodology used is described in Section 4. In particular, the new approaches applied to the GFS code are described in Section 4.1. Section 5 describes the experiments that were conducted to help improve the performance of the WSM6 and GFS physics routines. Section 5.3 and 5.4 show how ideas described earlier in the paper are applied and extended to the GFS physics and GFS radiation routines. Sections 6 and 7 provide a discussion of the results, draw conclusions and consider the future work that is needed.

## 2 Related Work

There has been significant activity recently on porting and optimizing NWP codes on various new computer architectures. For example, Mielikaimen et al. [19] optimized the Goddard microphysics scheme on an Intel Xeon phi 7120P Knights Corner (KNC) [4] by removing temporary variables to reduce the code memory footprint and by refactoring loops for vectorization, leading to a 4.7 speed-up. Furthermore Mielikainen et al. [20], also optimized the Perdu-Lin microphysics scheme by using the same approaches. Again this resulted in a 4.7 speed-up by using vector alignment and SIMD directives. Similarly, Ouermi et al. [24] used a low-level optimization approach based upon OpenMP 4 [5] directives to improve the performance of WSM6 on the KNL. When combined with minor code restructuring to enable and improve locality and vectorization, this resulted in a speed-up of three on the whole of WSM6. This speed-up include unoptimized (serial bottleneck) code sections.

In optimizing the Weather Model Radiative Transfer Physics on Intel KNL, Michalakes et al. [18] focused on increasing concurrency, vectorization and locality. Improving concurrency involved increasing the number of subdomains to be computed by threads. Vectorization and locality were improved by restructuring the loops to compute over smaller tiles and exposing vectorizable loops. This effort led to a 3 speed-up over the original 1.3 speed-up over Xeon Sandybridge.

Data layout plays a key role in performance optimization. The optimal data layout minimizes the memory footprint, reduces cache misses and allows better usage of vector units. This study uses thread-local structures of arrays (SOA) data layout to improve memory access [12, 28] . The SOA approach and similar approaches have been used to accelerate many scientific applications on various architectures. Henretty et al. [10] used data layout transformation to improve the performance of stencil computations. These optimizations removed alignment conflicts, reduced cache misses and improved vectorization. Woodward et al. [15, 27] used briquette data structures to accelerate a Piecewise Parabolic Method (PPM) code by reducing memory traffic. A briquette is a small sub-block of a uniform grid. The size of the briquette is chosen in relation to the cache size and vector unit. These data transformations enabled high performance because they reduce the memory footprint and traffic. In addition, such transformations improve vectorization.

The work presented in this paper relies on the OpenMP runtime system for task scheduling and OpenMP "pragma" directives for parallelization. Other approaches could be employed. Mencagli et al. [16] used a runtime support to reduce the effective latency of inter-thread cooperation. This latency reduction is done with a "home-fowarding" mechanism that uses a cache-coherent protocol to reduce cache-to-cache interaction. Buono et al. [2] proposed a light-weight runtime system as an approach to optimize linear algebra routines on Intel KNC [4]. This run time system focuses on efficient scheduling of tasks from a directed acyclic graph (DAG) that is generated on the fly during execution. Danelutto et al. [6] suggested a pattern-based framework for parallelization. This parallelization approach targets known patterns that can be represented with well-known operations such as map, reduce, scan, etc.

Although this work focuses on the Intel KNL and Haswell architectures, it is important to point out that efforts have been made to port and optimize WRF physics schemes for GPUs [21, 17, 26, 8]. GPU-based optimizations show better performance than Intel KNC and KNL-based optimizations. For instance, Mielikainen et al. [21], using CUDA [23], were able to achieve a speed-up of two orders of magnitude. However, porting to GPUs often requires significant code rewrites. The present work is part of larger effort to develop and optimize a potential US Navy next generation weather code, NEPTUNE. The optimization strategies introduced in this paper target Intel KNL and Haswell because the operational version of NEPTUNE is intended to run on Intel micro-architecture instead of GPUs.

## 3 Overview of NEPTUNE, WSM6 and GFS

The code optimization work described here is related to an activity to improve the performance of the Navy Environmental Prediction sysTem Utilizing the Nonhydrostatic Unified Model of the Atmosphere (NUMA) corE [7] (NEPTUNE). The NEPTUNE code couples the dynamical core NUMA, of Giraldo et al. [9], with physics schemes such as the WSM6 and GFS, schemes considered here. NUMA is novel in that it makes use of a three-dimensional hexahedral spectral element technique with a sphere-centered Cartesian coordinate system. The NUMA spectral element method is potentially a good choice for modern computer architectures as it has a relatively large floating point operations count for a relatively small communication footprint, which helps large scalability. However to make use of this potential for good performance, it is important to ensure that the appropriate physics schemes, such as WSM6 and GFS perform well. WSM6 is a physical parameterization that simulates processes in the atmosphere that cause precipitation in the form of rain, snow, graupel, water vapor, cloud water and cloud ice. WSM6 improves on WSM5 by introducing graupel particles and other variables to better model the precipitation of hydrometeors. The computation in the scheme is organized along both the horizontal and vertical directions. There is no interaction among the horizontal grid points, which allows straightforward parallelism cases.

GFS is a weather forecast model developed by the National Center for Environmental Prediction (NCEP). GFS is a coupled model composed of an atmosphere model, an ocean model, land/soil model, and sea-ice model. The optimization efforts targets GFS physics and GFS radiation the two most expensive calls within the module driver. Similarly to WSM6, GFS has no dependencies along the horizontal direction. Thus, making amendable to performance improvement without the concern of communication.

## 4 Experimental Setup and Methodology

### 4.1 Strategies for OpenMP Parallelism

#### 4.1.1 Motivation

Ouermi et al.[25, 24] used OpenMP directives to optimize individual loops in WSM6 at a low-level. This was appropriate given the small (3K lines) size of the code and the numerous serial sections obstructing parallelism. While this approach was successful, it is too time-consuming to apply to the large number of loops in the GFS physics module `http://www.dtcenter.org/GMTB/gfs_phys_doc/`. Moreover, GFS is not only a much larger code, but it has fewer serial bottlenecks and is thus potentially a candidate for high-level parallelism as will be discussed below. For this reason, this work investigates both high-level and low-level optimization approaches. While minor code transformations are used to improve performance of GFS and WSM6 physics schemes, there is an emphasis on high-level parallelism. The following two sections provide a brief overview of these concepts.

#### 4.1.2 Task Granularity (High-Level Versus Low-Level OpenMP)

High-level parallelism refers to parallelizing large blocks of code at the parent level in the function call stack, whereas low-level refers to parallelizing smalls blocks of codes at the instruction level (i.e., loops and arithmetic operations). The high-level approach has the advantage of using few individual parallel section, and few modifications within these sections. However, the high-level approach also requires the code blocks to be thread safe and free of serial bottlenecks.

In contrast, the low-level approach has the advantage of permitting parallelism in selectively parallelizable code punctuated by serial sections. If these serial bottlenecks are not easily removed, or if their relative cost is low, this may be a valid approach. Low-level approaches may also be appropriate for codes that require multiple different parallelization approaches (i.e., static versus dynamic scheduling, tasking, etc.) within different logical blocks or subroutines. Whether high-level or low-level parallelism is best depends on the individual code in question. High-level OpenMP is typically more elegant, but requires code that is sufficiently independent to be parallelizable at the parent level in the call stack. A low-level approach requires adding more parallel directives, but allows the original code structure to be used more or less as is. High-level and low-level approaches relate to task granularity, i.e., at which level logic is parallelized within a call stack. The length and the complexity of the logic within each task may have an impact on scheduling and load balancing, as well as on inter-task dependency.

### 4.1.3 Data Granularity, Chunks and SOA

In the physics schemes within NEPTUNE, data granularity refers to the size of arrays or sub-arrays that are processed by each thread. Coarse-grain data parallelism corresponds to dividing up the input data into the number of worker threads and fine-grain data parallelism corresponds to further subdividing input data into smaller chunks. The chunk size determine the size of the subdivided data as shown in as in Figure 1. For instance, an 2D input array ($im \times jm$) is dived up into multiple 2D sub-arrays of sizes $chunksize \times jm$ Typically input and output data are organized in arrays of structures (AOS) and regular arrays. WSM6, GFS physics, and GFS radiation uses large regular arrays and SOA. These input and output data are transformed into thread-local SOA. A thread-local SOA is a SOA that is private to a particular thread. The beneficial chunk size of the thread-local SOA is determined by the SIMD unit length (8 or 16 in the case of KNL and Haswell), or by the number of cores per block (SM) in a GPU. A more in depth study of SOA and other data structures can be found in [12, 28, 11]

In choosing the appropriate chunk size for an optimum data granularity, the goal is to keep the data as local as possible to each thread. Ideally, within the L1, and L2 caches, it is advantageous to use thread-local data structures and copy to and from global shared-memory arrays as necessary. The thread-local data are most effective when aligned to SIMD/chunk size and organized in SOA fashion. This data transformation allows the data for each thread to be packed closely in memory, thus reducing cache misses, and requests from L3/MCDRAM (on KNL), and or main memory.

The input and output data structures in WSM6 and GFS codes are not suitable for performance optimization because both SOA and regular arrays are large and do not fit into cache. In addition, the SOA are dynamically allocated which requires expensive memory operations. Using thread-local SOA instead of large regular arrays and statically instead of dynamically allocated arrays enable better memory usage and vectorization. Figure 1 shows examples of the data transformation. Arrays A and B represent original input and output data. The top half of A and B are copied into a thread-local SOA that is private to the thread to which it will be assigned. The same transformation is done for the bottom half of A and B. In the cases where the original input is a large SOA composed of A and B, the transformation would similar, from large shared memory SOA to thread-local SOA. This thread-local SOA ensures that data required for a calculation is close by other in memory hence fit into cache together. Overall, this modification enable memory locality.

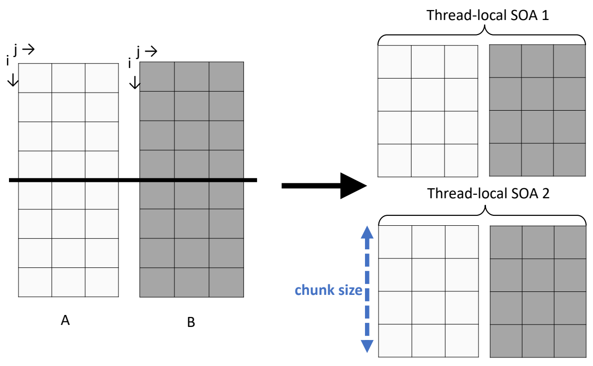


**Figure 1:** Transformation from AOS to SOA. The 2D arrays A and B are transformed into two thread-local SOA. The top and bottom parts are put next to each other as shown on the right. The chunk size shown in blue determined how to split and A and B. If the chunk size was chosen to be two, A and B would have been split into four parts which would give four thread-local SOA.

### 4.1.4 GFS physics Code Modifications

Though GFS physics and GFS radiation have similarities with WSM6, additional code transformations were applied to GSF code to achieve reasonable speed-ups. Thread-local SOA, transformation from dynamic to static allocation, and low-level transformation/vectorization, described below, are the key changes implemented in GFS physics and GFS radiation to enable better performance and are now described in turn.

- Data reorganization with thread-local SOA: A thread-local SOA transformation is applied to the input and output arrays as described in Figure 1. This transformation makes it possible to construct a thread-local SOA that is local to the thread to which it is assigned and small enough to fit in cache. This transformation requires copying original input and output data into the new thread-local SOA before passing it to the GFS driver function calls. In the work by Ouermi et al. [25], the data reorganization transformed regular arrays to thread-local SOA. Here, the data converted from large SOA, and regular arrays to thread-local SOA.
- Dynamic to static allocation: With static allocation the arrays sizes are known at compile time whereas in the dynamic case the arrays sizes are not known a priori. The original GFS code employs dynamic allocation for the input and output arrays in the GFS driver. This does not guarantee contiguous data. However, each array in the SOA will be contiguously allocated but the different allocations may be far apart in memory. Thus, accessing dynamically allocated arrays is often more expensive than accessing statically allocated arrays. Instead of using the original data or SOA that are dynamically allocated, the original inputs and outputs are copied to statically allocated thread-local SOA and then passed to the function calls in the GFS driver.
- Vectorization and low-level code transformations: The GFS physics and GFS radiation do not have many serial bottlenecks that requires major code transformation at a low-level as in WSM6 with niflv_rain_plm6 and niflv_rain_plm. Auto vectorization often fails to vectorize large body loops, or relatively complex code. Given that there are not many serial bottlenecks, the OpenMP directive OMP SIMD is used at the lower level in the physics parameterization codes to improve vectorization. This directive is applied to the most inner loop, the i-loop, which has no-dependencies. In the case of WM6, as shown by Ouermi et al. [25] major code transformation was required in some cases to enable better vectorization.

## 4.2 Experimental Setup

### 4.2.1 Methodology

The methodology used here follows Ouermi et al. [24, 25] to investigate various optimization strategies. This methodology consist of constructing standalone experiments to study the different approaches for parallelism in a more flexible and controlled environment. The findings from the standalone experiments inform the optimization decisions in the modules of interest, such as WSM6, GFS physics, and GFS radiation.

### 4.2.2 Architectures

The Intel Knights Landing (KNL) [13] architecture consists of 36 tiles interconnected with a 2D mesh, MCDRAM of 16GB high bandwidth memory on one socket. The KNL architecture has a clock frequency of 1.3 GHz, which is lower than the 2.5 GHz of Haswell. The Knights Landing tile is the basic unit that is replicated across the entire chip. This tile consists of two cores, each connected to two vector processing units (VPUs). Both cores share a 1 MB L2 cache. Two AVX-512 vector units process eight double-precision lanes each; a single core can execute two 512-bit vector multiply-add instructions per clock cycle. The Intel Xeon CPU E-7-8890 (Haswell) is composed of four sockets and four Non Uniform Memory Access (NUMA) nodes. Each node is made of 18 cores with 2 threads per core and clock frequency of 2.5 Ghz frequency.

## 5 Results

### 5.1 Standalone Experiments

#### 5.1.1 Synthetic Codes

These experiments analyze the thread-local SOA performance with different array sizes and dimensions in order to find a suitable structure for the physics schemes. The thread-local SOA in Code 1 use 1D arrays whereas those in Code 2 use 2D arrays. In Code 1 the k-loop is vectorized whereas in Code 2 the vectorization is along the i-loop. The access pattern is more involved in Code 1 compared to Code 2 because of the 1D versus 2D data layout. The performance results from the data transpose approach, as shown in Figure 2, and the GFS and WSM6 codes take 2D ($im \times jm$) and 3D ($im \times jm \times km$) arrays where $im > 800$ and $jm < 40$. For a long long rectangular data matrix ($im \times km$) as shown in Figure 2, thread parallelism across the k loop is limited by the number of iterations, i.e. $km$. In this case $km < 40$ which corresponds to less than 40 threads out of the 256 threads on KNL. Transposing the data matrix from $im \times km$ to $km \times im$ allows for better thread parallelism while maintaining a good memory access pattern as illustrated in Figure 2. This transformation does not have an impact on computation correctness because both the standalone experiment codes, and target physics codes have no dependencies along the horizontal direction (i-loop).

```
CODE 1

!$OMP PARALLEL DEFAULT(shared)
!$OMP PRIVATE(its,ite,ice,
                tsoa,thread_id,c)
!$OMP DO
do c=1,ite
  do j=1,je
    tsoa%a(j) = a(c,j)
    tsoa%b(j) = b(c,j)
    tsoa%d(j) = d(c,j)
    tsoa%e(j) = e(c,j)
  enddo
  call work(tsoa%a,tsoa%b,
            tsoa%d,tsoa%e,1,ice)
  do j=1,je
    a(c,j) = tsoa%a(j)
    b(c,j) = tsoa%b(j)
    d(c,j) = tsoa%d(j)
    e(c,j) = tsoa%e(j)
  enddo
enddo
!$OMP END DO
!$OMP END PARALLEL

subroutine work(a, b, c, d)
imlicit none
real,intent(inout):: a(:),b(:)
real,intent(inout):: c(:),d(:)
integer:: j
!$OMP SIMD
do j=2,je-1
  a(j) = 0.1+c(j)/d(j)
  b(j) = (0.2+c(j-1)-c(j))
       /(c(j)-c(j-1)+0.5)
enddo
end subroutine work
```

```
CODE 2

!$OMP PARALLEL DEFAULT(shared)
!$OMP PRIVATE(its,ite,ice,
              tsoa,thread_id,c)
!$OMP DO
do c=1,ite
  its = 1+ (c-1)*CHUNK
  ite = min(its+CHUNK-1, ie)
  ice = ite-its+1
  do j=1,je
    tsoa%a(1:ice,j) = a(its:ite,j)
    tsoa%b(1:ice,j) = b(its:ite,j)
    tsoa%d(1:ice,j) = d(its:tte,j)
    tsoa%e(1:ice,j) = e(its:ite,j)
  enddo
  call work(tsoa%a,tsoa%b,
            tsoa%d,tsoa%e,1,ice)
  do j=1,je
    a(its:ite,j) = tsoa%a(1:ice,j)
    b(its:ite,j) = tsoa%b(1:ice,j)
    d(its:ite,j) = tsoa%d(1:ice,j)
    e(its:ite,j) = tsoa%e(1:ice,j)
  enddo
enddo
!$OMP END DO
!$OMP END PARALLEL

subroutine work(a, b, c, d)
imlicit none
real, intent(inout):: a(:,:),b(:,:)
real, intent(inout):: c(:,:),d(:,:)
integer, intent(in):: is,ie
integer:: i,j
do j=2,je-1
  !$OMP SIMD
  do i=is,ie
    a(i,j) = 0.1+c(i,j)/d(i,j)
    b(i,j) = (0.2+c(i,j-1)-c(i,j))
             /(c(i,j)-c(i,j-1)+0.5)
  enddo
enddo
end subroutine work
```

Figure 3 shows a code example of the transposition. Following the column major ordering in Fortran, the i-loop becomes the outer loop with $im = 10586$ after transformation. Furthermore, there are no dependencies along the i index, which allows parallelism in index $i$ to be exploited.

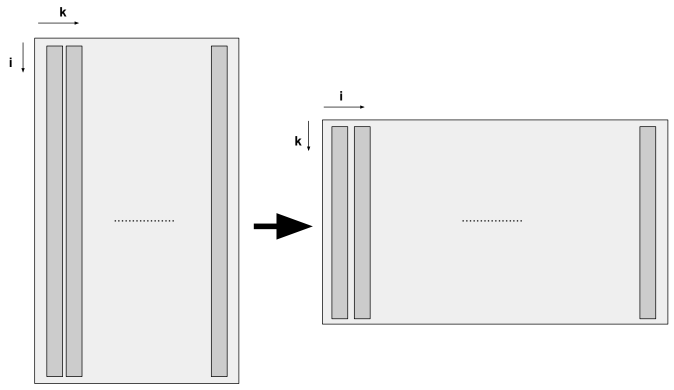
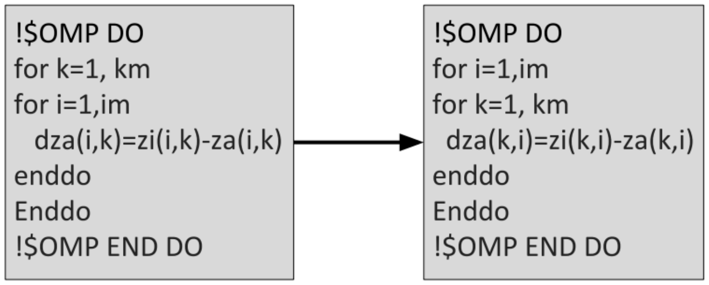


**Figure 2:** Transpose representation. This shows transposition of a 2D ($im \times jm$) array, where $im > 800$ and $jm < 40$. Given that Fortran is column major $k$ is the outer loop before the transpose shown on the left. The outer loop becomes $i$ after the transposition as shown on the right. This transformation increases thread parallelism on the outer loop.



```
!$OMP DO
for k=1, km
for i=1,im
  dza(i,k)=zi(i,k)-za(i,k)
enddo
Enddo
!$OMP END DO
```

```
!$OMP DO
for i=1,im
for k=1, km
  dza(k,i)=zi(k,i)-za(k,i)
enddo
Enddo
!$OMP END DO
```

**Figure 3:** Code transformation with transpose. This shows how the transposition is implemented with a simple code. The loops and the indices are swapped.

Table 1 shows performance results from using SOA with 1D arrays, transposed data matrices and unmodified original data. The SOA approach yields significant speed-ups with a maximum speed-up of about 34. The data transpose approach performs the best in this particular experiment, with a maximum speed-up of about 41. The length of the arrays in the SOA is 48. This small array length translates to small amount of work for the innermost loop in Code 1. In this experiment the peak performance is observed at 128 threads with two threads per cores. In hyper-threading, each core resources are shared between the hyper-threads. The instructions from hyper-threads flow through the same pipeline. This can help improve core utilization as observed in table 1. However, sharing resource between hyper-threads may lead to performance decrease as observed with 256 threads. In addition after 128 threads the work is not enough to enable further speed-up improvement.

Table 2 shows performance results similar to those in Table 1 with an increased problem size given by ke = 768. The arrays in the SOA are 16 times larger that those used in previous experiments. In both cases, these results indicate that the transpose approach for data organization yields better results. After 64 threads each core uses hyper-threading, with two to four threads per core. For a given core this divides up the resources between the hyper-threads causing cause the performance to decrease.

| Threads | Time (ms) | | | Speed-up | | |
|---|---|---|---|---|---|---|
| | Orig. | Transp. | SOA | Orig. | Transp. | SOA |
| 1 | 2.06 | 3.3 6 | 3.33 | 1 | 0.61 | 0.62 |
| 2 | 1.59 | 1.97 | 1.74 | 1.30 | 1.05 | 1.18 |
| 4 | 0.91 | 1.44 | 0.84 | 2.26 | 1.43 | 2.45 |
| 8 | 0.67 | 0.5 | 0.41 | 3.07 | 4.12 | 5.02 |
| 16 | 0.55 | 0.26 | 0.18 | 3.75 | 7.92 | 11.44 |
| 32 | 0.54 | 0.17 | 0.15 | 3.81 | 12.12 | 13.73 |
| 64 | 0.72 | 0.05 | 0.11 | 2.86 | 41.20 | 18.73 |
| 128 | 0.87 | 0.05 | 0.06 | 2.37 | 41.20 | 34.33 |
| 256 | 1.35 | 0.1 | 0.49 | 1.53 | 20.60 | 4.20 |

**Table 1:** Results from CODE 1 compared to transpose approach and original code. The maximum speed-ups for transpose and thread-local SOA are at 128 threads. At 128 threads each core uses two hyper-threads per core. The hyper-threads share the same instruction pipeline which helps improve utilization of cores. Sharing resource can contribute to reducing core utilization as observed at 256 threads. Hyper-threading performance is dependent on how the shared resources are manage between hyper-threads.

| Threads | Time (ms) | | | Speed-up | | |
|---|---|---|---|---|---|---|
| | Orig. | Transp. | SOA | Orig. | Transp. | SOA |
| 1 | 33.82 | 29.53 | 75.45 | 1.00 | 1.15 | 0.45 |
| 2 | 26.98 | 19.44 | 45.7 | 1.25 | 1.74 | 0.74 |
| 4 | 15.54 | 13.47 | 23.37 | 2.18 | 2.51 | 1.45 |
| 8 | 10.9 | 5.09 | 7.44 | 3.10 | 6.64 | 4.55 |
| 16 | 8.86 | 2.98 | 5.96 | 3.82 | 11.35 | 5.67 |
| 32 | 8.93 | 2.61 | 1.72 | 3.79 | 12.96 | 19.66 |
| 64 | 10.97 | 0.95 | 1.39 | 3.08 | 35.60 | 24.33 |
| 128 | 16.14 | 1.17 | 5.93 | 2.10 | 28.91 | 5.70 |
| 256 | 22.27 | 2.17 | 9.57 | 1.52 | 15.59 | 3.53 |

**Table 2:** Results from CODE 1 compared to transpose approach and original code with large array sizes. In this experiment the maximum performance occurs at 64 threads. After 64 threads hyper-threading is used and the resource per core is divide up between the hyper-threads. This causes the performance to slow down.

Table 3 shows performance results from using thread-local SOA with 2D arrays, transposed data matrices and unmodified original data. In this experiment, the OpenMP chunk size is set to 8. In contrast to the previous experiments, these results show that the thread-local SOA approach yields higher speed-ups than the other methods for data organization. The maximum speed-up observed is 103 at 32 cores. After 32 threads there is not enough work per thread to enable performance scalability.

| Threads | Time (ms) | | | Speed-up | | |
|---|---|---|---|---|---|---|
| | Orig. | Transp. | SOA | Orig. | Transp. | SOA |
| 1 | 2.06 | 3.36 | 1.99 | 1.00 | 0.61 | 1.04 |
| 2 | 1.59 | 1.97 | 1.07 | 1.30 | 1.05 | 1.93 |
| 4 | 0.91 | 1.44 | 0.53 | 2.26 | 1.43 | 3.89 |
| 8 | 0.67 | 0.5 | 0.14 | 3.07 | 4.12 | 14.71 |
| 16 | 0.55 | 0.26 | 0.07 | 3.75 | 7.92 | 29.43 |
| 32 | 0.54 | 0.17 | 0.02 | 3.81 | 12.12 | 103.00 |
| 64 | 0.72 | 0.05 | 0.06 | 2.86 | 41.20 | 34.33 |
| 128 | 0.87 | 0.05 | 0.27 | 2.37 | 41.20 | 7.63 |
| 256 | 1.35 | 0.1 | 0.04 | 1.53 | 20.60 | 51.50 |

**Table 3:** Results from CODE 2 compared to transpose approach and original code. The best performance is observed at 64 threads for the thread-local SOA. At 128 and 256 threads each core uses about two and four threads per core. The core resources, such as L1 cache, are shared between the hyper-threads. This causes the performance to slow down for large core counts.

The results from Table 1 – 3 indicate that the size and the structure of the arrays in the thread-local SOA play an important role in the performance. Vectorizing along the k-loop, in the 1D case, has a more involved access pattern than vectorizing along the i-loop, in the 2D case. In addition, there are no dependencies along the i-loop, which allows for trivial vectorization. Furthermore, the L2 cache is about 16 times the size of the input data in each SOA. Thus the thread-local SOA fit in the L2 cache, which allows for fast memory access. When the thread-local SOA does not fit in the L2 cache, as shown in Table 4, the speed-ups are significantly lower than the ones observed in Table 3. In Table 4 the peak performance for thread-local SOA is observed at about 16 threads. This is lower than the previous cases because of the high rate of cache misses. This occurs because the thread-local SOA does not fit in cache

| Threads | Time (ms) | | | Speed-up | | |
|---|---|---|---|---|---|---|
| | Orig. | Transp. | SOA | Orig. | Transp. | SOA |
| 1 | 264.71 | 194.94 | 159.98 | 1.00 | 1.36 | 1.65 |
| 2 | 119.93 | 120.69 | 113.15 | 2.21 | 2.19 | 2.34 |
| 4 | 98.89 | 61.57 | 57.08 | 2.68 | 4.30 | 4.64 |
| 8 | 54.17 | 25.57 | 34.25 | 4.89 | 10.35 | 7.73 |
| 16 | 30.11 | 16.3 | 22.83 | 8.79 | 16.24 | 11.59 |
| 32 | 16.87 | 13.51 | 34.23 | 15.69 | 19.59 | 7.73 |
| 64 | 13.81 | 13.15 | 29.72 | 19.17 | 20.13 | 8.91 |
| 128 | 15.74 | 6.56 | 38.25 | 16.82 | 40.35 | 6.92 |
| 256 | 23.33 | 13.24 | 45.51 | 11.35 | 19.99 | 5.82 |

**Table 4:** Results from CODE 2 compared to transpose approach and original code with large arrays. The peak performance is observed at 16 threads. In this case the array thread-local SOA do not fit in L2 cache. This lead to lower performance than the cases where the thread-local SOA fit in cache.

Figure 4 shows the performance results from choosing different lengths for the index i. All the chunk sizes considered yield higher speed-ups than using transpose approach. The best performance is observed when using a chunk size of 32. The chunk size of 32 provides enough work to make better use of the SIMD units. The choice of the chunk size is application dependent.

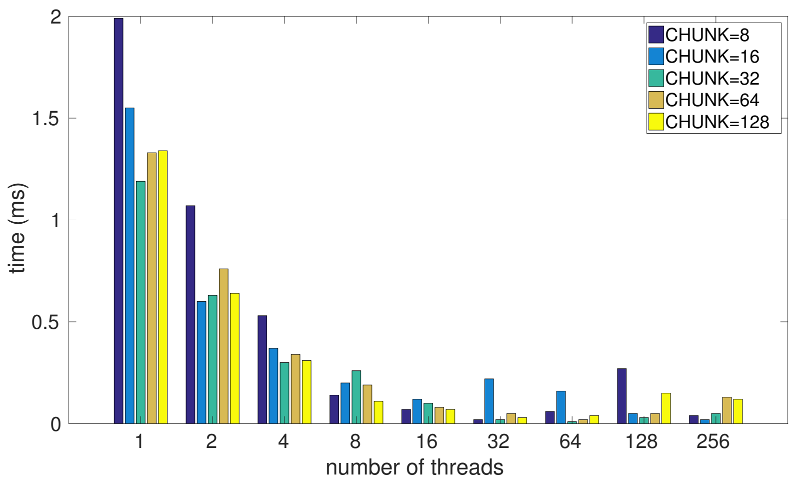


**Figure 4:** Plots of thread-local SOA performance wieh different chunk sizes. The bars indicate the run time of the optimized standalone Code 2 with different size thread-local SOA which determined by the choice of chunk. The lowest run time occurs at 64 with $chunk = 32$. This indicates that $chunk = 32$ provide enough work per thread and one thread per core enables a better usage of core resources compare to two and four threads per core

### 5.1.2 Rain Routines

The WSM6 module contains semi-Lagrangian routines [14], nisflv_rain_plm6, and nisflv_rain_plm6 for simulating falling hydrometeors. These semi-Lagrangian routines, an alternative to a traditional eulerian scheme, use forward advection to calculate the path of the falling hydrometeors. Initially, nisflv_rain_plm6, and nisflv_rain_plm6 used Fortran keywords cycle, goto, and exit. With these keywords, the termination criteria is not known a priori, which prevents parallelism. This limitation was resolved by substituting the keywords with carefully engineered logic that performs the same computation. The exits were replaced by masking, the gotos by loops couple with conditionals and cycle by conditionals. After removing these serial bottlenecks, the thread-local SOA and transpose approaches from CODE 2 are applied to the rain routines. As in CODE 2, the rain routines have no dependencies along the i-loop and the a thread-local SOA with a chunk size of 32 now fits into the L2 cache.

The results in Figure 5 and Table 5 for the optimized rain routine with $chunk = 32$ demonstrate that using thread-local SOA produces larger speed-ups than transposing the input data. In this case, the thread-local SOA are chosen to fit in cache and designed for contiguous memory access to improve performance. In contrast, transposing the input data increase parallelism at the thread level but does not improve memory performance. The optimized, thread-local SOA, version of the rain routine runs 50 faster than the original serial version, and 2 faster than the transpose version. Sections 5.2–5.3 present results of applying thread-local SOA to WSM6, GFS physics, and GFS radiation.

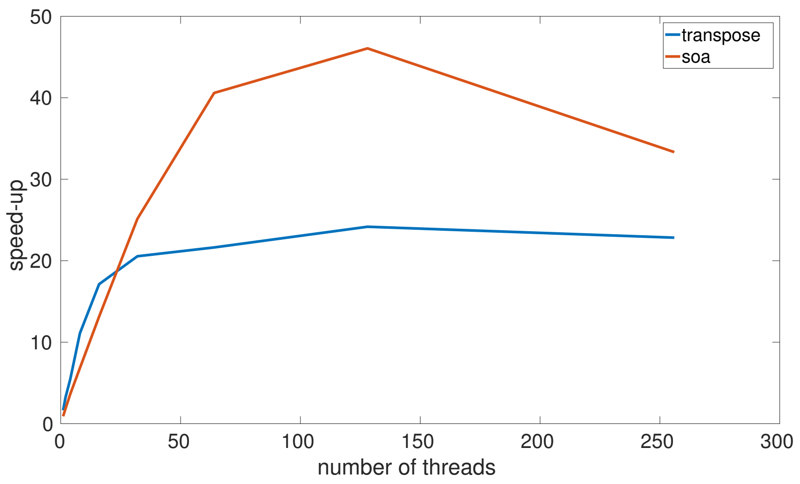


**Figure 5:** Transpose vs SOA speed-ups on nisflv_rain_plm6. This thread scalability plot reaffirms that using thread-local SOA scales better than transposing the input data.

| Threads | Transpose (ms) | SOA (ms) |
|---|---|---|
| 1 | 250 | 450 |
| 2 | 127 | 220 |
| 4 | 74 | 112 |
| 8 | 37 | 60 |
| 16 | 24 | 31.2 |
| 32 | 20 | 16.3 |
| 64 | 19 | 10.1 |
| 128 | 17 | 8.9 |
| 256 | 18 | 12.3 |

**Table 5:** Thread-local SOA and Transpose approach applied to nisfl_rain_plm6. This show run times of transpose and thread-local SOA on a subroutine in WSM6.

## 5.2 WSM6

In addition to the transformations in nisflv_rain_plm6 and nisflv_rain_plm6 routines, the OMP SIMD directive is applied at the lower level to the innermost loops instead of relying onto the Intel compiler auto vectorization. Thread parallelism is implemented at the parent level in the WSM6 module. Figures 6 -10 show the results of these optimization efforts. The bar plots in Figure 6 and 7 does not show significant differences in run time for various thread-local SOA sizes on KNL and Haswell. These figures indicate that the different chunk sizes used in this experiment achieve about the same performance improvement with a best speed-up of about 26 when using static scheduling.

Figures 8, and 9 compare static versus dynamic scheduling performance on KNL and Haswell respectively. In both systems, dynamic scheduling performs better than static scheduling. The dynamic scheduler helps load balance the work between the threads. Because of the conditionals and complexity within physics routines, the work distributed between the threads may be unbalanced, causing some threads to run longer than necessary. With dynamic scheduling an internal work queue is used to give block of iterations to each thread. When a thread finishes its current task it retrieves the next ready block for the top of the queue. This help reduce the wait time observed in the static scheduling case. In the case of KNL the performance can be further improved by enabling the flat configuration. In the flat configuration, the high band width memory (HBM) MCDRAM is used as a physical address instead of cache. This flat configuration in WSM6 make better usage of the

HBM compare to cache configuration. Figure 10 compares flat versus cache performance on KNL. The flat KNL configuration provides better performance than the cache configuration by a factor of 1.6. Overall the optimized version of WSM6 runs 70 faster, and 26 faster on KNL and Haswell respectively. Haswell performs better than KNL by a factor of by a factor of 1.3. In addition, the optimized version of WSM6 on KNL runs 23.3 faster than the optimized version presented by Ouermi et al. [24]

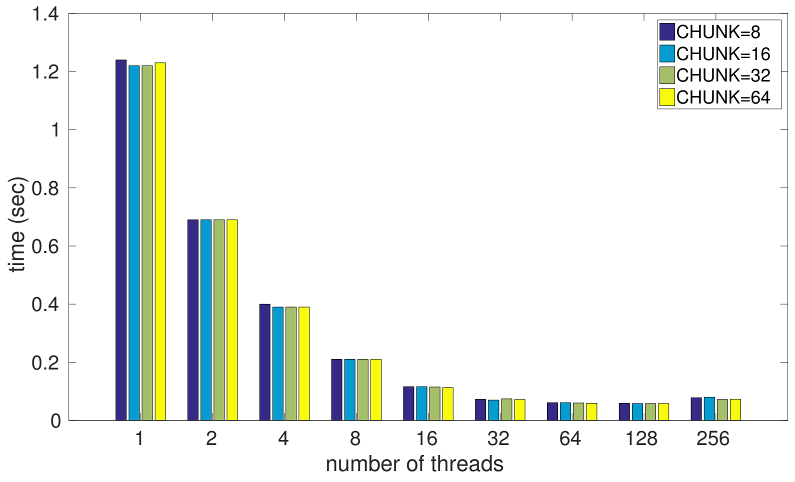


**Figure 6:** WSM6 run time with various thread-local SOA sizes and static scheduling on KNL. The bars shows that the run times decreases exponentially as the number of threads increase regardless of the chunk sizes. Each chunk provide enough work for thread and vector parallelism. The lowest run time occurs at 64 threads and plateaus after that.

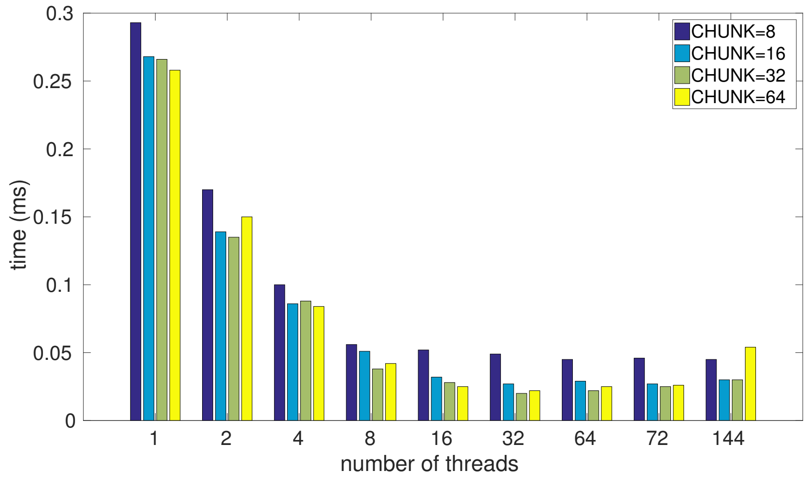


**Figure 7:** WSM6 run time with various SOA sizes and static scheduling on Haswell. The bars show that the run times decrease exponentially as the number of thread increase up to about 32 threads. The best run time is observed at 64 threads with *chunk* = 32. The performance plateaus after the 64 threads. This indicate that hyper-threading doesn't improve performance in WSM6. In addition after 64 threads the amount of work per thread is not large enough to enable scalability.

| Threads | cache (ms) | flat (ms) |
|---|---|---|
| 1 | 1079.3 | 1084.32 |
| 2 | 570.51 | 574.92 |
| 4 | 325.86 | 324.91 |
| 8 | 171.67 | 167.61 |
| 16 | 93.3 | 90.32 |
| 32 | 53.66 | 50.21 |
| 64 | 35.4 | 31.66 |
| 128 | 45.39 | 23.45 |
| 256 | 65.59 | 24.2 |

**Table 6:** SOA approach applied to WSM6 with flat and cache modes.

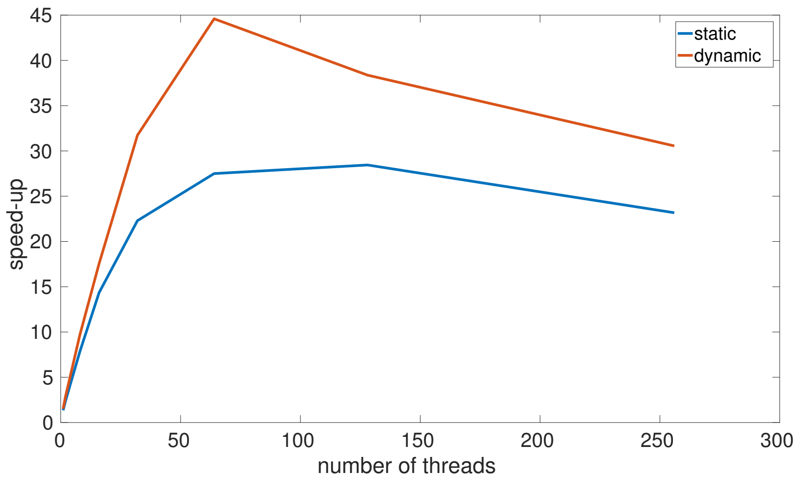


**Figure 8:** WSM6 speed-ups on KNL. This shows scalability plots of WSM6 with static and dynamic scheduling. The chunk size is chosen to be 32 in this case. The performance scales up to 64 threads and then decreases. Hyper-threading is used at 128 and 256 threads. In hyper-threading, the core resources are divided up between hyper-thread and this may limit the performance as seen in this case.

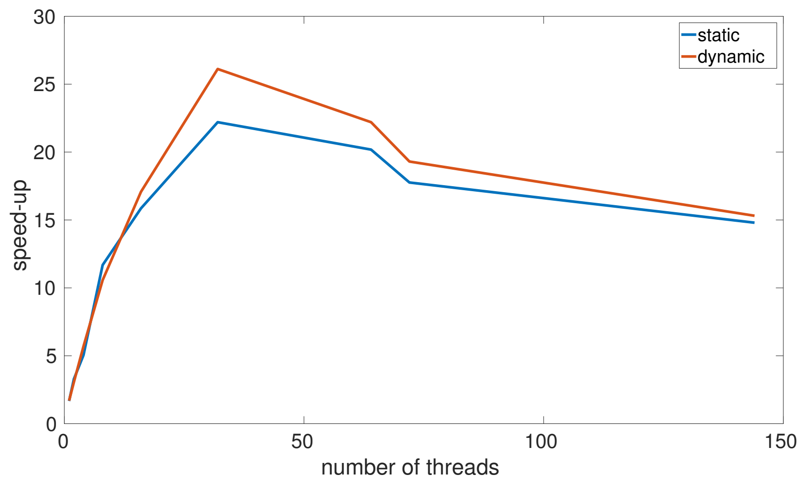


**Figure 9:** WSM6 speed-ups with static and dynamics scheduling on Haswell. The best performance occurs at 32 threads. After 16 threads performance is limited by NUMA affect because OpenMP is not suitable for parallelism across NUMA nodes. The performance could be improve by using MPI.

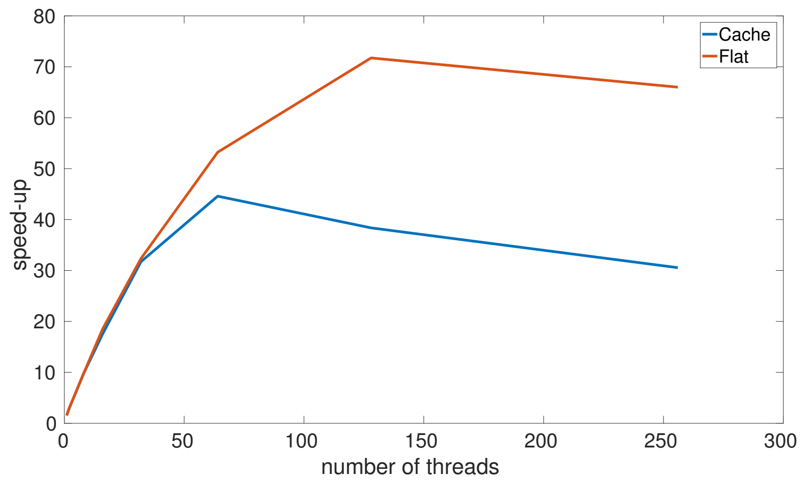


**Figure 10:** WSM6 speed-ups on KNL with flat configuration. In this scalability plot of WSM6 with cache and flat configuration, dynamic scheduling is used for both and the chunk size is set to 32. The maximum speed-up is observed at 64 threads in the case of cache configuration and 128 threads in the case flat. In the case of the flat mode, hyper-threading with two hyper-threads per core improved performance. Though the resources per core are share between hyper-threads, in this case the set instructions in the shared instruction pipeline enable a better utilization of core resources.

## 5.3 GFS physics Results

GFS physics does not have many serial bottlenecks that requires major code transformations as in the case of WSM6 with niflv_rain_plm6 and niflv_rain_plm. Thread parallelism is applied at a high level in the GFS driver using thread-local SOA. OMP SIMD directives are instrumented at lower level, innermost loops, to enable better vectorization. In addition, static allocation is used for the thread-local SOA instead of dynamic allocation as in the original input data.

Figures 11–17 summarize GFS physics performance results. Figures 11 and 12 show run time performance on KNL and Haswell respectively with different chunk sizes. In the case of KNL the run time decreases exponentially, indicating good scalability. As shown in Figure 12 the run time decreases up to about 16 threads. The first 16 threads are running in one NUMA node. After 16 threads more NUMA nodes are used. Shared memory parallelism is not suitable for parallelism across NUMA nodes. This limitation is addressed by using four MPI ranks, one for each node. Figure 13 indicates that coupling the four MPI ranks with shared memory parallelism led to significant improvement on run time past the 16 threads.

Figures 14–16 compare static and dynamic scheduling scalability. In Figure 15 the speed-up increases up to 16 threads and decrease rapidly after the 16 threads because of difficulties shared memory parallelism across the NUMA nodes. In both Figures 14 and 16 static scheduling performs better than dynamic scheduling. The work load between threads is sufficiently balanced that using a dynamic scheduler does not yield any improvement. In the case of KNL, the flat configuration improves the speed-up by a factor of 1.04 compared to the cache configuration. The optimized version of GFS physics runs about 2.4 faster on Haswell compared to KNL. This corresponds to speed-ups of 27 and 18 on KNL and Haswell respectively over the original serial versions.

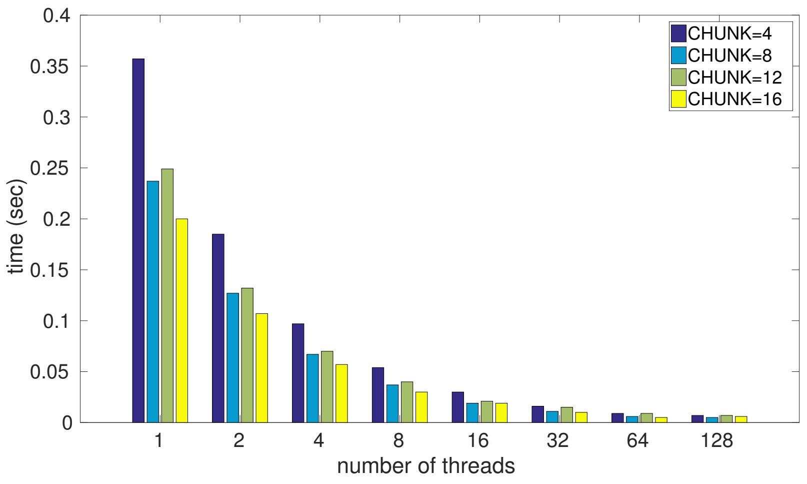


**Figure 11:** GFS physics run time with various thread-local SOA sizes on KNL. This plot shows the run times of different thread-local SOA to help guide the choice of chunk size. Static scheduling is used in this experiment. The maximum speed-up occurs at 128 threads with $chunk = 8$. The maximum number of loop iterations is 108. Thus using 256 threads is largely more than necessary given there is only 108 loop iterations.

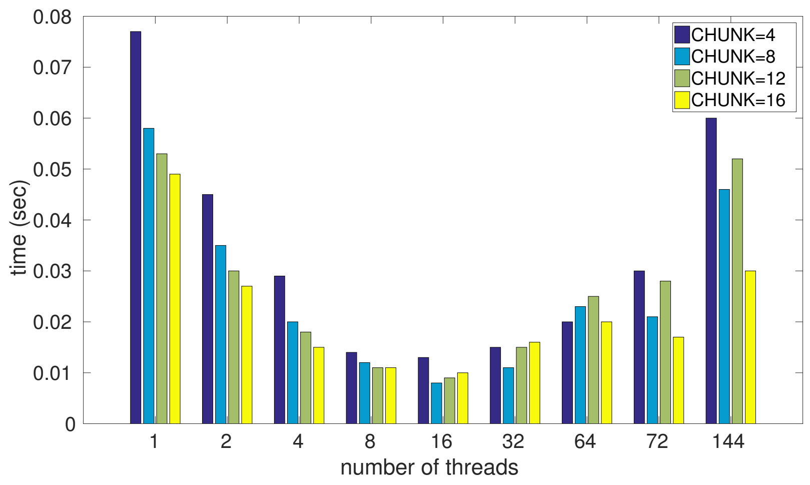


**Figure 12:** GFS physics run time with various SOA sizes on Haswell. This plots shows run times of different thread-local SOA to help inform on the appropriate chose for the chunk size. MPI was not used. OpenMP is used for shared parallelism across NUMA nodes. the default static scheduler is used in this experiment. The lowest run time occurs a 16 threads. After 16 threads NUMA effect start limiting performance. This can be addressed by using MPI for parallelism across NUMA nodes.

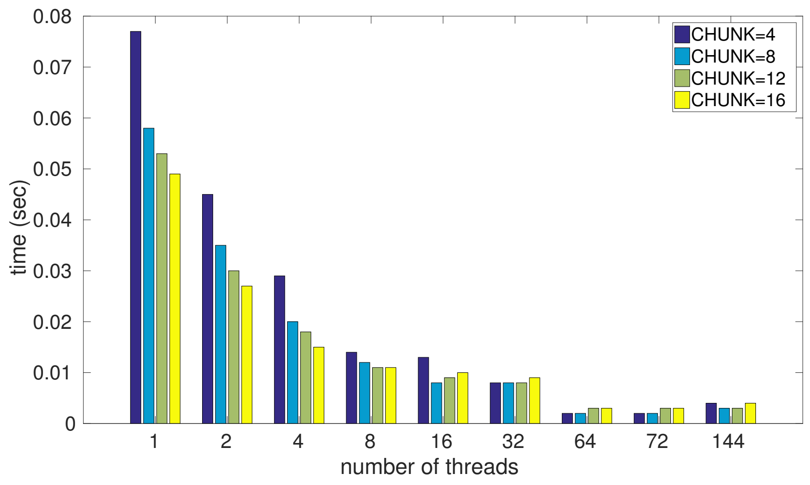


**Figure 13:** GFS physics run time with various SOA sizes on Haswell with MPI. This plots shows run times of different thread-local SOA to help inform on the appropriate chose for the chunk size. MPI was used for parallelism across NUMA nodes and OpenMP for shared parallelism within NUMA nodes. The default static scheduler is used in this experiment. The performance scales up to 72 threads. Hyper-threading does not help improve speed-ups.

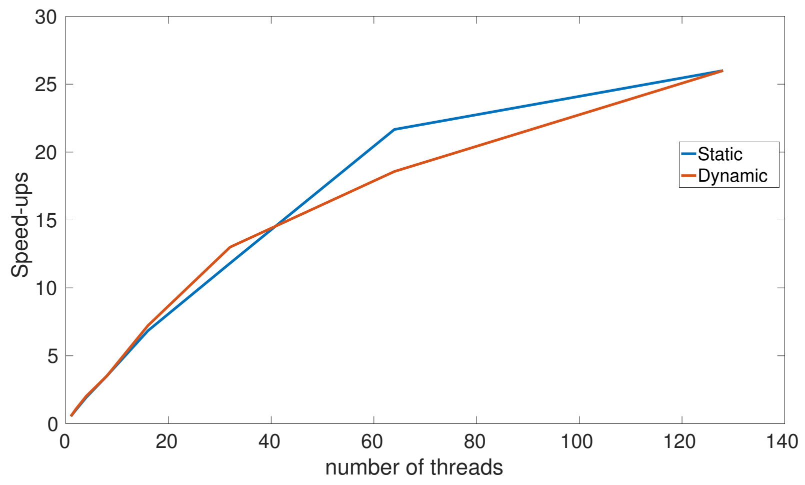


**Figure 14:** GFS physics speed-ups on KNL. These plots show thread scalability performance with static and dynamic scheduling on KNL. The performance scales up to 128 threads. The uses a maximum of 128 threads because there is 108 iteration. Using 256 would oversubscribing. In this case hyper-threading enable better performance.

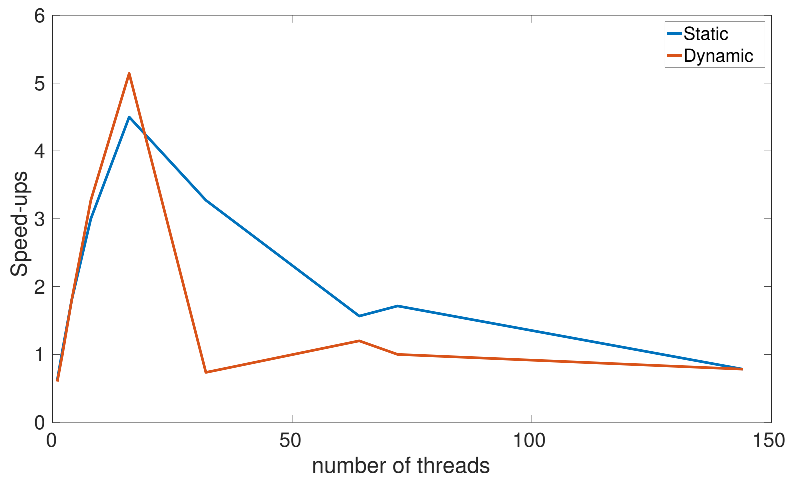


**Figure 15:** GFS physics speed-ups on Haswell. These plots show thread scalability performance with static and dynamic scheduling on Haswell. OpenMP is used for parallelism within and across NUMA nodes. The perfomance decrease after 16 threads because of NUMA effects. Using OpenMP fr parallelism across NUMA nodes does not improve speed-ups.

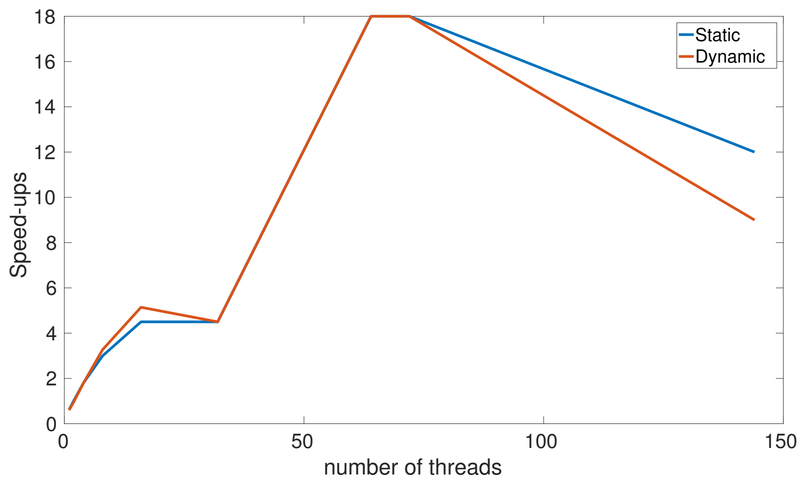


**Figure 16:** GFS physics speed-ups on Haswell with MPI across nodes. These plots show thread scalability performance with static and dynamic scheduling on Haswell. MPI and OpenMP are used for parallelism across and within NUMA nodes respectively. This optimization scales up to 72 cores. Hyper-threading does not improve the utilization of core resources.

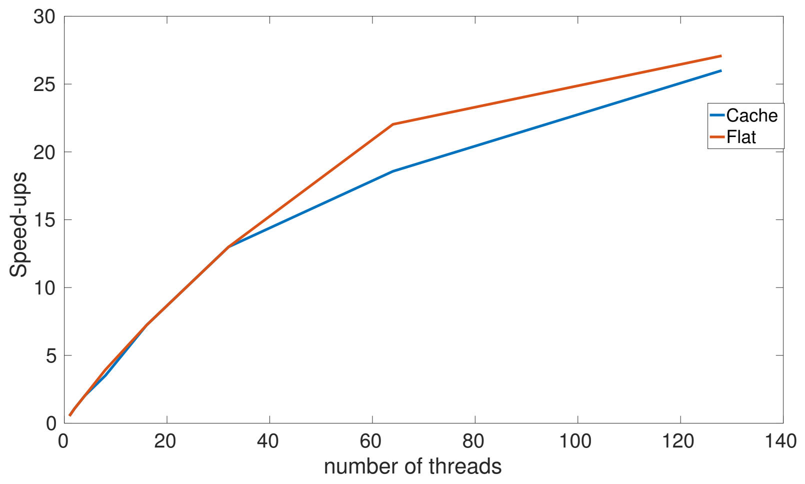


**Figure 17:** GFS physics speed-ups on KNL. These plots show thread scalability performance with flat and cache configuration on KNL. Dynamic scheduling is used. Both cache and falt configuration scale up to 128 threads.

## 5.4 GFS radiation

As in GFS physics, GFS radiation is optimized at the high-level with thread-local SOA to improve thread parallelism and at the low-level with OMP SIMD to improve utilization of SIMD units. Static instead of dynamic allocation is used as well to improve memory accesses. Figures 18 – 24 show GFS radiation performance results. Similarly to WSM6 and GFS physics, the bar plots in Figures 18 - 20 inform on the appropriate thread-local SOA size to choose for optimization. Figure 19 indicates limitation of using OpenMP for parallelism across NUMA nodes. Figures 18 and 20 show that the chunk sizes of 8 and 16 yield lowest run time on both KNL and Haswell.

Figures 21 – 23 compare static and dynamic performance on KNL and Haswell. Similar to previous case with GFS codes no using MPI for parallelism across NUMA nodes does not scale as shown in Figure 22. On both KNL and Haswell dynamic scheduling performs better than static scheduling. The dynamically assigned work loads to threads reduces threads wait time compared to statically distributing work between the threads. Further performance improvement is observed when using flat configuration in the case of KNL by a factor of 1.05 as shown in Figure 24.

The optimized version of GFS physics run 23 and 30 faster on KNL and Haswell respectively over the serial times. The run time on Haswell is 6.5 faster than the run time on KNL.

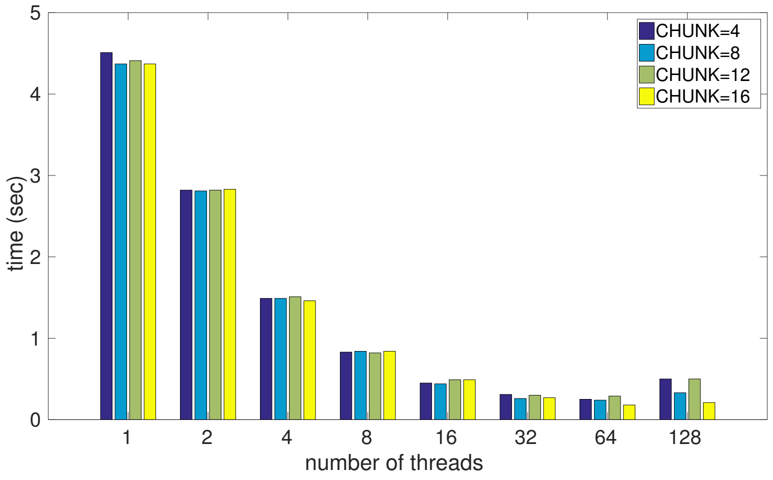


**Figure 18:** GFS radiation run time with various SOA sizes on KNL. This plots shows run times of different thread-local SOA to help inform on the appropriate chose for the chunk size. Static scheduling is used in this experiment. The best speed-up is observed at 64 threads. This indicates that one thread per core allows for better utilization of core resources compare to two threads per core.

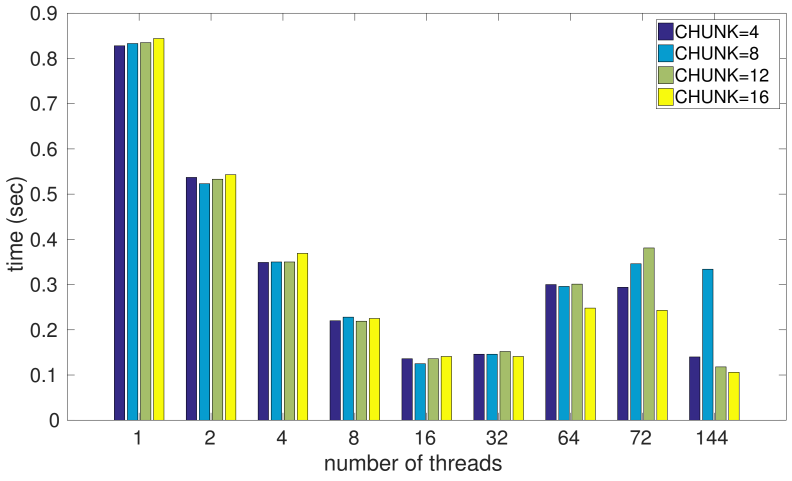


**Figure 19:** GFS radiation run time with various SOA sizes on Haswell This plots shows run times of different thread-local SOA to help inform on the appropriate chose for the chunk size. MPI was not used. OpenMP is used for shared parallelism across NUMA nodes. the default static scheduler is used in this experiment. The best performance is observed at 16 threads. After 16 threads using OpenMP for parallelism across NUMA nodes limits performance. OpenMP is designed for shared memory parallelism.

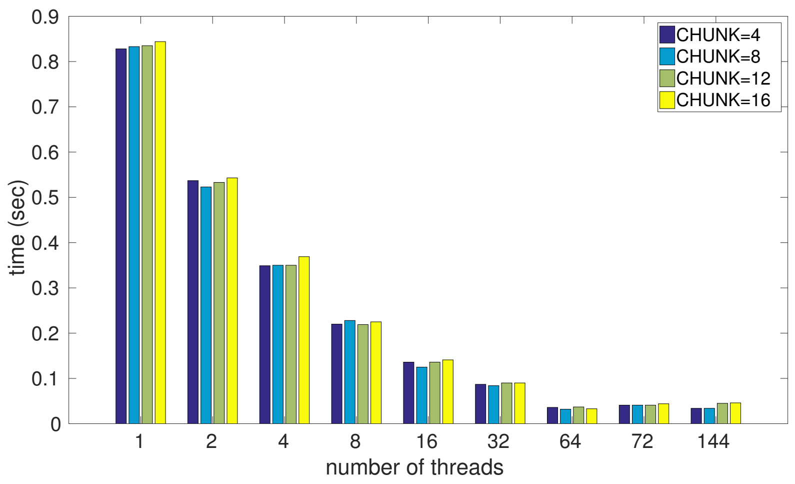


**Figure 20:** GFS radiation run time with various SOA sizes on Haswell. This plots shows run times of different thread-local SOA to help inform on the appropriate chose for the chunk size. MPI was used for parallelism across NUMA nodes and OpenMP for shared parallelism within NUMA nodes. The default static scheduler is used in this experiment. This experiment scales up to 64 threads. Using hyper-threads does not help improve performance.

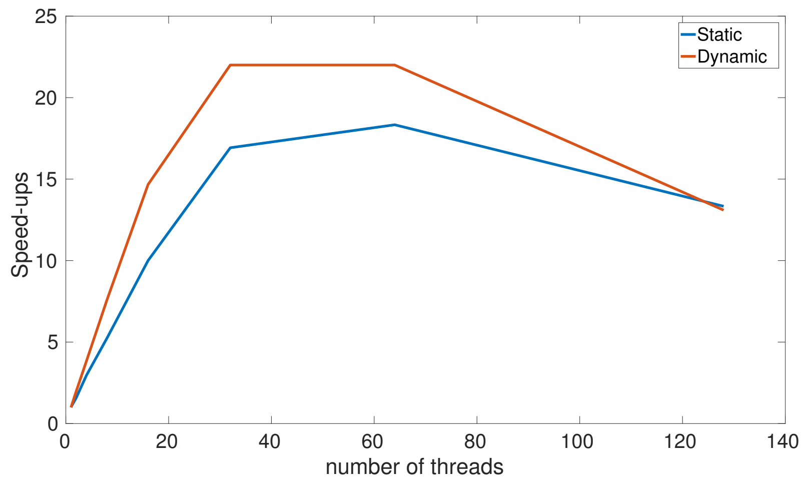


**Figure 21:** GFS radiation speep-ups on KNL These plots show thread scalability performance with static and dynamic scheduling on KNL. The maximum performance is observed at about 64 threads. The performance does not change much between 32 and 64 threads because there is not enough work per thread to improve scalability After 64 threads the performance decreases because the hyper-threading does not enable better utilization of core's resources.

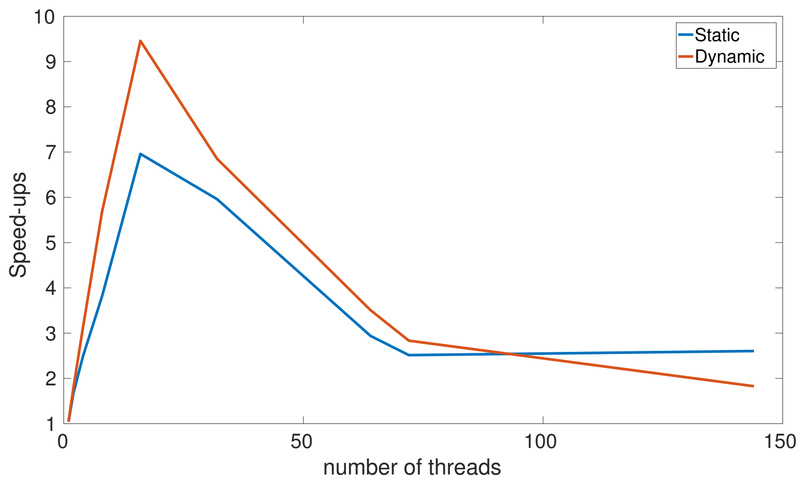


**Figure 22:** GFS radiation speed-ups on Haswell. These plots show thread scalability performance with static and dynamic scheduling on Haswell. OpenMP is used for parallelism within and across NUMA nodes. The best performance is observed at 16 threads. After 16 threads, the performance decreases because OpenMP is not suitable for parallelism across NUMA nodes.

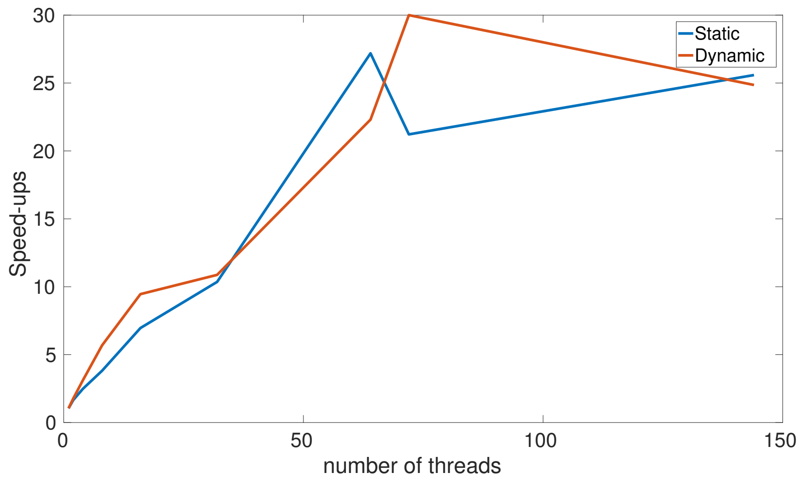


**Figure 23:** GFS radiation speed-ups on Haswell with MPI across nodes. These plots show thread scalability performance with static and dynamic scheduling on Haswell. MPI and OpenMP are used for parallelism across and within NUMA nodes respectively. This optimized code scales up to 72 threads. After the 72 threads the performance decreases. This indicates that one thread per core enable a better utilization of core's resources than two threads per core.

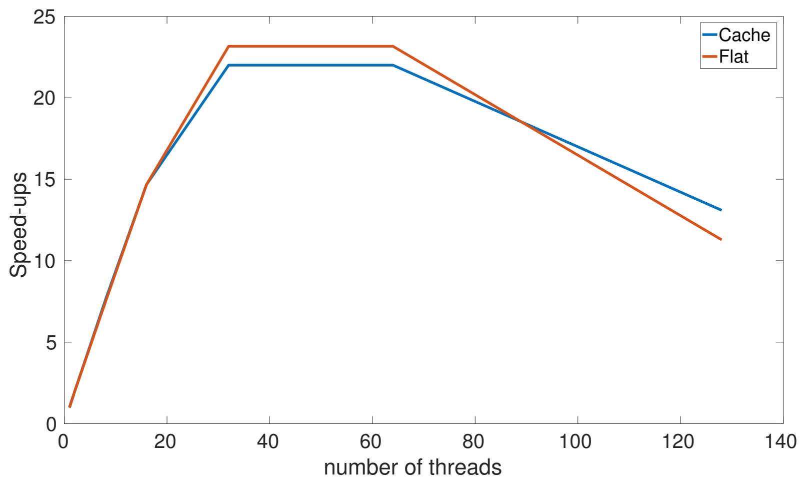


**Figure 24:** GFS radiation speed-ups on KNL These plots show thread scalability performance with flat and cache configuration on KNL. Both flat and cache scale up to 64 threads. The performance s decreases after the 64 threads because hyper-threads do not improve utilization of core's resources. Because the test case fit in MCDRAM, the flat configuration enable a slightly better memory usage which is translated into better performance.

## 6 Discussion

The results from the standalone experiments in Section 5.1 demonstrate that the thread-local SOA approach is suitable for optimizing the physics schemes with in NEPTUNE. These standalone experiments are instrumental in identifying the modifications necessary to optimize WSM6, GFS physics and GFS radiation on the KNL and Haswell. This study exploits the flexibility and simplicity of the standalone experiments to prototype and test the different optimization strategies which are not easily and trivially testable in NEPTUNE.

The transformation of the input and output data into thread-local SOA is the main approach used in optimizing the WSM6 and GFS codes. The size of the thread-local SOA is chosen to fit in the L2 cache. Each thread-local SOA is composed of the inputs and outputs required to calculate the physics for few columns. This data transformation reduces memory traffic and increase data locality. In the transpose approach, the data might be far apart in memory and to large to fit in the L2 cache. This causes cache misses which limits performance. In addition, applying the transpose to the entire physics routines requires significant code modification compared to the thread-local SOA approach.

The OpenMP directive OMP SIMD is used to improve vector parallelism at the low-level. In cases similar to the rain routines, significant low-level code modifications are required to enable vectorization. Given that there are dependencies along the vertical direction, the OMP SIMD directive is applied along the horizontal direction ($i$ loop). In the thread-local SOA, the i-loop corresponding to the chunk size is chosen to be a multiple of the SIMD unit length.

The original serial version of WSM6, GFS physics, and GFS physics ran for 1.65 sec, 0.130 sec, 4.40 sec on KNL and 0.444 sec, 0.036 sec, 0.870 sec on Haswell. This about 3.7, 3.6 and 5.05 faster on Haswell compared to KNL for serial codes. The original codes rely on auto vectorization which does not work well with complex and large body of code. Haswell has lower run times because it has a higher clock frequency and a turbo boost. Table 7 show a summary of performance improvement from original codes to optimized thread-local SOA on both KNL and Haswell.

The optimized version of WSM6 yields a speed-up of 70 and 26 on KNL and Haswell respectively over the serial times. This about 1.3 faster on Haswell compare to KNL. In the case of Haswell, the maximum performance is observed at about 32 cores compared to 64 cores on KNL. Haswell performs better than KNL because it has higher clock speed and transactional synchronization extensions (TSX-NI) technology to improve threading. The performance of WSM6 on Haswell could be improved by designing the code to run with 4 MPI ranks for parallelism across NUMA nodes. On KNL it could be further improved by better using the SIMD units.

The optimized version of GFS physics runs about 2.4 faster on Haswell compared to KNL. GFS physics scales up to the 72 cores Haswell and the 64 cores on KNL. The large SIMD units on KNL are not sufficient to outperform Haswell which has more cores and a higher clock frequency than KNL. After optimization, GFS physics runs 27, and 18 faster on KNL and Haswell respectively over the serial times.

The optimized GFS radiation runs 23 faster on KNL and 30 faster on Haswell with respect to their serial times. In this case Haswell performs about 6.5 better than KNL. As in the GFS physics optimization, the GFS radiation scales up to the 64 cores on KNL and the 72 cores on Haswell.

The test cases used in this study have about $10K$ iterations for WSM6 and 800 iterations for GFS codes. These test cases are not large enough to provide sufficient work to each threads and scale well to the 64 cores on KNL and 72 cores on Haswell. As this work continues large test cases will be studied.

With regard to peak performance, some of the challenges faced by physics codes are illustrated by CODE 2 in Section 5. In this case, there are only 9 flops in the inner loop. This is typical of some of the loops in WSM6. As a result, with array dimensions of 10592 and 39, there are only 3.7M flops. A loop time of 0.02ms gives a flop rate of 185 GFLOPs, which is about 6.6% of peak and is not unexpected for loops that have low flop counts.

All the tables and plots show a performance decrease after 128 threads for KNL and 72 threads for Haswell. This corresponds to two or four thread per core. In the KNL and Haswell, all active threads in a given core flow through the same pipeline, and thus they share resources such as instruction cache and instruction queue. The increase in the number of threads per cores leads to the division of the shared resources among threads, and to an increase in memory access conflicts. This competition for resources indicates why a performance decrease is observed after 128 threads and 72 threads on KNL and Haswell respectively.

| physics schemes | | WSM6 | GFS physics | GFS radiation |
|---|---|---|---|---|
| KNL | best time (ms) | 23.0 | 4.8 | 190.0 |
| | speed-up | 70 | 27 | 23 |
| | threads | 64 | 128 | 64 |
| | configuration | dynamic+flat | static+flat | dynamic+flat |
| Haswell | best time (ms) | 17.0 | 2.0 | 29.0 |
| | speed-up | 26 | 18 | 30 |
| | threads | 32 | 72 | 72 |
| | configuration | dynamic | static | dynamic |

**Table 7:** Performance summary. This shows best performance results for the different physics codes used on KNL and Haswell. On KNL the chunk size is set to 8 and on Haswell it is set 32.

## 7 Conclusion and Future Work

This work demonstrated the efficiency of high-level optimization approach using thread-local SOA paired with low-level optimization technique using OMP SIMD directive. As presented in the results section, these optimization approaches enables a better utilization of the KNL and Haswell resources by improving locality, memory allocation and vectorization. The use of thread-local SOA and static allocation enable better memory traffic by increasing locality and decreasing cache misses. The use OMP SIMD directives coupled with SOA chunk sizes, set to be multiples SIMD length, enable a better utilization of SIMD units in KNL and Haswell. Overall, the various optimizations achieved a speed-ups of 70, 27, 23 on KNL, and 26, 18, 30 on Haswell over the original serial version of WSM6, GFS physics, GFS radiation respectively. In addition, the results indicated that WSM6, GFS physics, GFS radiation run 1.3, 2.4 and 6.5 faster on on Haswell compared to KNL. This is because the Haswell system used here has more cores and a higher clock frequency than KNL. As mentioned in the discussion peak performance is still relatively challenging to achieve given the complexity of the physics schemes and we continue to investigate methods for improving the percentage of peak performance. In terms of future work, a better understanding of how to use hyper-threading, a study of MPI + OpenMP on large test cases will better inform on more effective optimization approaches in NEPTUNE on supercomputers.

## 8 Acknowledgements

The authors would like to thank the Intel Parallel Computing Center at the University of Utah for funding one of us (T.A.J.O.). Kevin Viner, Alex Reinecke of NRL and John Michalakes of UCAR and thanked for their constant help and support with the NEPTUNE code and with many aspects of code organization.

## References

[1] Peter Bauer, Alan Thorpe, and Gilbert Brunet. The quiet revolution of numerical weather prediction. *Nature*, 525(7567):47–55, September 2015.

[2] D. Buono, M. Danelutto, T. D. Matteis, G. Mencagli, and M. Torquati. A lightweight run-time support for fast dense linear algebra on multi-core. *Proc. of the 12th International Conference on Parallel and Distributed Computing and Networks (PDCN 2014)*, Feb 2014.

[3] Skamarock W. C., J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, M. G Duda, X.-Y. Huang, W. Wang, and J. G. Powers. 2008: A description of the advanced research wrf version 3. NCAR Tech. Note NCAR TN/475-STR 113 p.

[4] G. Chrysos. Intel xeon phi coprocessor (codename knights corner). In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–31, Aug 2012.

[5] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[6] Marco Danelutto, Tiziano De Matteis, Daniele De Sensi, Gabriele Mencagli, and Massimo Torquati. P3arsec: towards parallel patterns benchmarking. In *In Proceedings of the Symposium on Applied Computing (SAC '17)*, pages 1582–1589, New York, NY, USA, April 2017. ACM.

[7] James D. Doyle. A next generation atmospheric prediction system for the navy. Sept 2014. Award Number: N0001412WX20683, Tech. rep., DTIC Document, Naval Research Lab.

[8] Price E., Mielikainen J., Huang M., Huang B, Huang H. L. A., and Lee T. Gpu-accelerated longwave radiation scheme of the rapid radiative transfer model for general circulation models (rrtmg). *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 7(8):3660–3667, Aug 2014.

[9] F. X. Giraldo, J. F. Kelly, and E. M. Constantinescu. Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (numa). *SIAM Journal on Scientific Computing*, 35(5):B1162–B1194, 2013.

[10] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Martin Hirzel. Data layouts for object-oriented programs. *SIGMETRICS Perform. Eval. Rev.*, 35(1):265–276, June 2007.

[12] Holger Homann and Francois Laenen. Soax: A generic C++ structure of arrays for handling particles in HPC codes. *CoRR*, abs/1710.03462, 2017.

[13] Jim Jeffers, James Reinders, and Avinash Sodani. Chapter 4 - knights landing architecture. In Jeffers Jim, Reinders James, and Avinash Sodani, editors, *Intel Xeon Phi Processor High Performance Programming (Second Edition)*, chapter 4, pages 63–84. Morgan Kaufmann, Boston, second edition edition, 2016.

[14] Hann-Ming Henry Juang and Song-You Honmg. Forward semi-lagrangian advection with mass conservation and positive definiteness for falling hydrometeors. *Monthly Weather Review*, 138(04):1778–1791, 2010.

[15] P. Lin, P. Yew, P. R. Woodward, and J. Jayaraj. Moving scientific codes to multicore microprocessor cpus. *Computing in Science and Engineering*, 10(6):16–25, 2008.

[16] Gabriele Mencagli, Marco Vanneschi, and Silvia Lametti. The home-forwarding mechanism to reduce the cache coherence overhead in next-generation cmps. *Future Generation Computer Systems*, 2017.

[17] J. Michalakes and M. Vachharajani. Gpu acceleration of numerical weather prediction. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7, April 2008.

[18] John Michalakes, Michael J. Iacono, and Elizabeth R. Jessup. Optimizing weather model radiative transfer physics for intels many integrated core (mic) architecture. *Parallel Processing Letters*, 26(04):1650019, 2016.

[19] J. Mielikainen, B. Huang, and A. H.-L. Huang. Intel Xeon Phi accelerated Weather Research and Forecasting (WRF) Goddard microphysics scheme. *Geoscientific Model Development Discussions*, 7:8941–8973, December 2014.

[20] J. Mielikainen, B. Huang, and A. H.-L. Huang. Optimizing purdue-lin microphysics scheme for intel xeon phi coprocessor. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 9*, Jan 2016.

[21] J. Mielikainen, B Huang, H. L. A. Huang, and M. D. Goldberg. Improved gpu/cuda based parallel weather and research forecast (wrf) single moment 5-class (wsm5) cloud microphysics. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(4):1256–1265, Aug 2012.

[22] Andreas Müller, Michal A. Kopera, Simone Marras, Lucas C. Wilcox, Tobin Isaac, and Francis X. Giraldo. Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA. *CoRR*, abs/1511.01561, 2015.

[23] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[24] T. A.J. Ouermi, Aaron Knoll, Robert M. Kirby, and Martin Berzins. Openmp 4 fortran modernization of wsm6 for knl. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 12:1–12:8, New York, NY, USA, 2017. ACM.

[25] T.A.J. Ouermi, A. Knoll, R.M. Kirby, and M. Berzins. Optimization strategies for wrf single-moment 6-class microphysics scheme (wsm6) on intel microarchitectures. In *Proceedings of the fifth international symposium on computing and networking (CANDAR 17). Awarded Best Paper*. IEEE, 2017.

[26] Erik Price, Jarno Mielikainen, Bormin Huang, HungLung A. Huang, and Tsengdar Lee. Gpu acceleration experience with rrtmg long wave radiation model. In Bormin Huang, Antonio J. Plaza, and Zhensen Wu, editors, *Proceedings SPIE 8895 High-Performance Computing in Remote Sensing III,*, volume 8895, 2013.

[27] Paul R. Woodward, Jagan Jayaraj, and Richard Barrett. mppm, viewed as a co-design effort. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 33–40, Piscataway, NJ, USA, 2014. IEEE Press.

[28] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 93–104, New York, NY, USA, 2007. ACM.