Optimization Strategies for WRF Single-Moment 6-Class Microphysics Scheme (WSM6) on Intel Microarchitectures

T.A.J. Ouermi
University of Utah
SCI Institute
Email: touermi@sci.utah.edu

Aaron Knoll
University of Utah
SCI Institute
Email: knolla@sci.utah.edu

Robert M. Kirby
University of Utah
SCI Institute
Email: kirby@sci.utah.edu

Martin Berzins
University of Utah
SCI Institute
Email: mb@sci.utah.edu

***Abstract*—Optimizations in the petascale era require modifications of existing codes to take advantage of new architectures with large core counts and SIMD vector units. This paper examines high-level and low-level optimization strategies for numerical weather prediction (NWP) codes. These strategies employ thread-local structures of arrays (SOA) and an OpenMP directive such as OMP SIMD. These optimization approaches are applied to the Weather Research Forecasting single-moment 6-class microphysics schemes (WSM6) in the US Navy NEPTUNE system. The results of this study indicate that the high-level approach with SOA and low-level OMP SIMD improves thread and vector parallelism by increasing data and temporal locality. The modified version of WSM6 runs 70x faster than the original serial code. This improvement is about 23.3x faster than the performance achieved by Ouermi et al. [1], and 14.9x faster than the performance achieved by Michalakes et al. [2]**

***Index Terms*—Structure of arrays; OpenMP; Knights; Landing; Numerical Weather Prediction; Thread parallelism; Vector parallelism**

I. Introduction

The Weather Research and Forecasting (WRF) [23] model is a numerical weather prediction (NWP) software used by atmospheric researchers and weather forecasters at operational centers throughout the world. WRF is used in over 150 countries, thus making it one of the most widely used numerical weather prediction models. The model was developed to help scientists study and better understand weather phenomena. Optimizing the performance of NWP codes is important to improve the accuracy and the time requirements for forecasts. Thus the scientific community and governments have invested significant time and efforts to modernize NWP codes for current and future 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 [11] 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. Legacy codes such as the NWP codes must be re-architected to leverage thread and SIMD parallelism while maintaining data and temporal locality.

This work focuses on optimizing the WRF Single-Moment 6-Class Microphysics Scheme (WSM6) [3]. 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. The optimization efforts target the Intel KNL [11] and potential future computer architectures. This work employs OpenMP4 as a vehicle for portability across various platforms as it is a well-established and widely adopted interface for shared memory parallelism.

This paper presents an evaluation of high-level and low-level approaches for shared parallelism using thread-local structures of arrays (SOA). The high-level approach consists of parallelizing large blocks of code at the parent level in the call stack whereas the low-level approach targets individual instructions. SOA are employed to accelerate computation in WSM6 by improving data locality and taking advantage of thread and vector parallelism. To our knowledge, this is the first attempt to apply SOA to NWP codes. The various optimizations on WSM6 have resulted in a significant increase in speed-ups. For instance, the use of SOA coupled with OMP SIMD for vectorization led to a speed-up of 70x.

II. Related Work

A significant effort has been invested to port and optimize NWP codes on various architectures. Mielikaimen et al. [4] optimized the Goddard microphysics scheme for the Intel Xeon phi 7120P. This work on the Goddard microphysics scheme focused on removing temporary variables to reduce the memory footprint and restructuring loops to enable vectorization. This optimization effort led to a 4.7x speed-up from the original code on Xeon phi 7120P.

Mielikainen et al. [5] improved the Perdu-Lin microphysics performance by reducing the memory footprint and increasing vectorization. The memory footprint was reduced by fusing and collapsing loops. Vector alignment and SIMD directives were employed to improve vectorization. These various transformations resulted in a speed-up of 4.7x on Intel Xeon phi 7120P.

Ouermi et al. [1] adopted a low-level optimization approach to improve the performance of WSM6 on the KNL. This work employed OpenMP 4 [6] directives. In addition, minor code restructuring is used to enable and improve locality and vectorization. This optimization approach on WSM6 yielded about 50x speed-up on the optimized part of WM6 and a 3x speed-up on the entire WSM6, including the unoptimized sections (serial bottleneck).

In optimizing the Weather Model Radiative Transfer Physics on Intel MIC, Michalakes et al. [2] 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 3x speed-up over the original 1.3x 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. The SOA approach and similar approaches have been used to accelerate many scientific applications on various architectures.

Henretty et al. [12] used data layout transformation to improve the performance of stencil computation. These optimizations remove alignment conflicts, reduce cache misses and improve vectorization.

Woodward et al. [13], [14] used briquette data structures to accelerate a Piecewise Parabolic Method (PPM) code by reducing memory traffic. A briquette is a small region of a uniform grid. The size of the briquette is chosen in relation to the cache size and vector unit. These data transformations enable 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. [21] 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. [20] prosed a light-weight runtime system as an approach to optimize linear algebra routines on MIC. This runtime system focuses on efficient scheduling of tasks from a directed acyclic graph (DAG) that is generated "on the fly" during execution. Danelutto et al. [19] 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 MIC, it is important to point out that efforts have been made to port and optimize WRF physics schemes for GPUs [7], [10], [8]. GPU-based optimizations show better performance than MIC-based optimizations. For instance, Mielikainen et al. [7], using CUDA [22], were able to acheive a speed-up of two orders of magnitude. However, porting to GPUs often requires significant code rewrites.

III. Overview of NEPTUNE and WSM6

The work presented in this paper is part of a larger effort to accelerate the Navy Environmental Prediction sysTem Utilizing the Nonhydrostatic Unified Model of the Atmosphere (NUMA) corE [17], [18] (NEPTUNE). NEPTUNE uses NUMA, introduced by Giraldo et al. [15], and various physics schemes. This paper presents optimization efforts focused on the WRF Single-Moment 6-class Microphysics (WSM6) scheme. NUMA, the dynamical core of NEPTUNE, uses a three-dimensional spectral element technique with a sphere-centered Cartesian coordinate sysem. A spectral element is the numerical method of choice because of the small communication footprint, which enables large scalability.

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.

IV. Experimental Setup and Methodology

A. Strategies for OpenMP Parallelism

1) Motivation: The work of Ouermi et al. [1] on WSM6 used low-level OpenMP optimization of individual loops, which was appropriate due to the relatively small size of the code (3K lines) and numerous serial sections obstructing high-level parallelism. This low-level approach is not suitable for many modules in NEPTUNE. For instance, the GFS physics module http://www.dtcenter.org/GMTB/gfs_phys_doc/ poses different challenges: principally, it is a far larger codebase, has fewer serial bottlenecks and is more amenable to high-level parallelism. This work presents studies on both high- and low-level approaches with synthetic examples, which are subsequently applied to WSM6. The following two sections provide a brief overview of these concepts.

2) Task Granularity (High-Level Versus Low-Level OpenMP): High-level parallelism consists of parallelizing large blocks of code at a parent level in the function call stack. This approach stands in contrast to low-level parallelization, which operates at the individual instruction level (i.e., loops, arithmetic, etc.). The high-level approach has the advantage of requiring few modifications within the parallelized code regions, assuming these code sections are thread-safe and do not contain serial bottlenecks. From a performance standpoint, the high-level approach entails relatively few individual parallel sections. 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 already intrinsically parallel. Low-level 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.

3) Data Granularity, Chunks and SOA: Orthogonally, data granularity considers how data are divided among threads. In the physics and microphysics systems within NEPTUNE, data granularity refers to the size of arrays (or subarrays) processed by each thread. Generally, coarse-grain data parallelism entails dividing up the available work by the number of workers (more amenable to static distribution with OpenMP). Fine-grain data parallelism involves further subdividing the workload into smaller chunks. The minimum granularity beneficial for modern architectures is determined by SIMD size (8 or 16 on Xeon Phi KNL), or the number of cores per compute block (SM) on a GPU. The granularity size is often referred to as the chunk size.

In determining appropriate data granularity, the goal is to keep the data as local as possible to each thread, i.e., within the L1 and L2 caches. It is often advantageous to use thread-local data structures and copy to and from global (shared-memory) arrays as necessary. Thread-local subarrays are most effective when aligned to SIMD/chunk-size boundaries in memory, and organized in SOA fashion. Thus, for each thread, the local array data can be packed together closely in memory, requiring fewer cache misses and requests from L3, MCDRAM (on the KNL) or main memory.

Intuitively, data are often organized as an array of structures (AOS). However, such an approach for data organization is not suitable for vectorization and memory locality. Using SOA instead of AOS is a common technique used to address this limitation. Figure 1 shows an example of transformation from AOS to SOA. SOA improve memory locality and allow for more contiguous memory accesses.

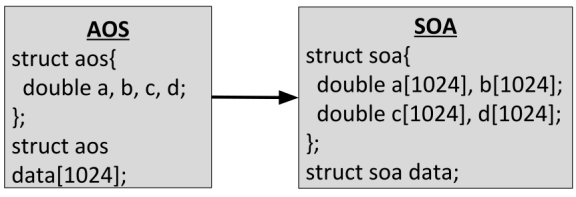

Fig. 1. Transformation from AOS to SOA

B. Experimental Setup

1) Methodology: This paper adopts a methodology similar to that employed by Ouermi et al. [1] to explore different parallelism strategies. This methodology consists of designing standalone experiments to study the behavior of the various approaches for parallelism. The findings from the standalone experiments inform the optimization decision in the module of interest, in this case WSM6.

2) KNL Architecture: The Intel Knights Landing (KNL) [11] 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.

V. Results

A. Standalone Experiments

These experiments analyze SOA with different array sizes and dimensions in order to find a suitable structure for WSM6. The 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 original code are compared against results from the SOA approach. The original WSM6 code takes 2D and 3D arrays. For a long "skinny" data matrix as shown in Figure 2, thread parallelism across the k loop is limited to 39 of the 256 threads on the KNL. With this approach, the computer resources are underutilized. Transposing the data matrix from $im \times km$ to $km \times im$ allows for better thread parallelism and maintains a good memory access pattern as shown in Figure 2. This transformation does not impact computation results because the standalone experiments and WSM6 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(:),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(:,:),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. Following the column major in Fortran, the i loop becomes the outer loop with im = 10586. Furthermore, there are no dependencies along the i indices, which allows parallelism in i to be exploited.

Table I 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 34x. The data transpose performs the best in this particular experiment, with a maximum speed-up of about 41x. 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.

Table II shows performance results similar to those in Table I 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.

Table III shows performance results from using SOA with 2D arrays, transposed data matrices and unmodified original data. In this experiment, the OpenMP chunk size is set to 8.

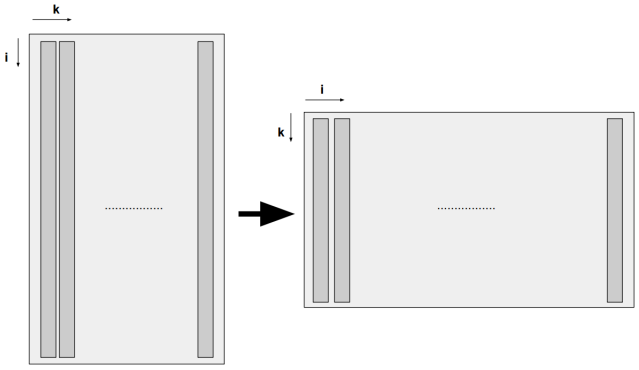

Fig. 2. Transpose representation.

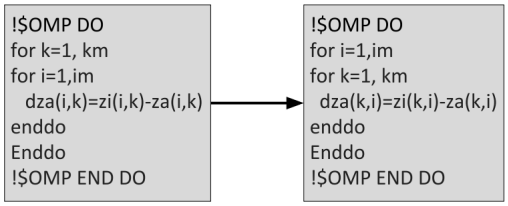

Fig. 3. Code transformation with transpose.

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 I
RESULTS FROM CODE 1 COMPARED TO TRANSPOSE APPROACH AND ORIGINAL CODE.

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 II
RESULTS FROM CODE 1 COMPARED TO TRANSPOSE APPROACH AND ORIGINAL CODE WITH LARGE ARRAY SIZES.

In contrast to the previous experiments, these results show that the SOA approach yields higher speed-ups than the other methods for data organization. The maximum speed-up observed is 103x.

The results from Table I, Table II and Table III indicate

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 III
RESULTS FROM CODE 2 COMPARED TO TRANSPOSE APPROACH ANI ORIGINAL CODE.

that the size and the structure of the arrays in the 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 do not fit in the L2 cache, as shown in Table IV, the speed-ups are significantly lower than the ones observed in Table III.

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 IV
RESULTS FROM CODE 2 COMPARED TO TRANSPOSE APPROACH AND ORIGINAL CODE WITH LARGE ARRAYS.

Figure 4 shows the performance results from choosing different lengths for i. All the chunk sizes considered yield higher speed-ups than transpose. The best performance is observed when using a chunk size of 32.

B. Rain Routines and WSM6

nisflfv_rain_plm6 and nisflv_rain_plm are semi-Lagrangian routines [16] for falling hydrometeors. These semi-Lagrangian routines employ forward advection to determine the advection path, and they are designed to replaced the traditional Eulerian scheme.

The original nislfv_rain_plm6 routine makes use of Fortran keywords exit, cycle and goto. These keywords prevent parallelism because the termination criteria of a given loop are not known a priori. These key words were replace by carefully designed logics that performed the same task. The keyword exit was replaced by masking, goto by a loop coupled with a conditional and cycle by a conditional.

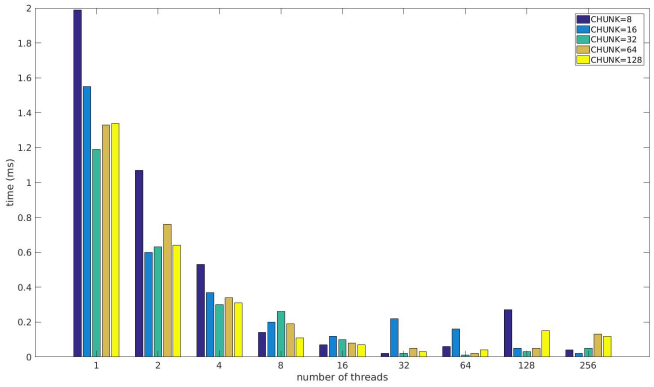

Fig. 4. Plots of SOA performance with different chunk sizes

The findings in previous sections are applied to the nisfl_rain_plm6 routine in WSM6. Table V and Figure 5 show performance results from applying the SOA approach to the nisfl_rain_plm6 routine with chunk=32. As shown in Figure 5, the SOA technique yields higher speed-ups than the transpose technique. These results indicate significant speed-ups with a maximum speed-up of 50x. In the same way as Code 2 above, the nisfl_rain_plm6 routine does not have dependencies along the i loop, and the thread-local SOA version of it fits in the L2 cache.

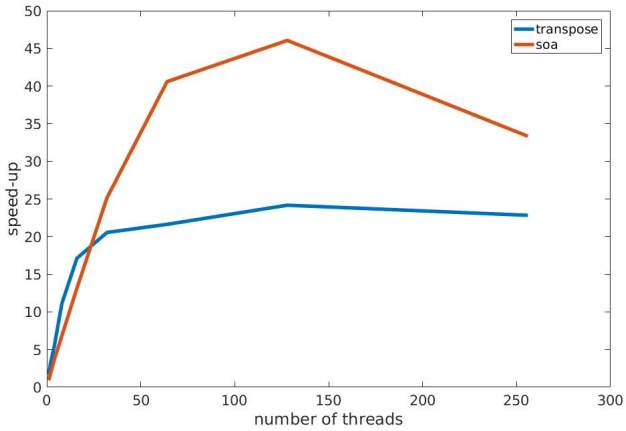

Fig. 5. Transpose vs SOA speed-ups on nisflv_rain_plm6

The high-level SOA coupled with the low-level SIMD approach was applied to the WSM6 module. In a similar way to the optimization in the nivfl_rain_plm6 routine, all the key-words preventing parallelism were removed. In addition, the nivfl_rain_plm6 and nivfl_rain_plm routines were restructured to allow thread and vector parallelism across the i loop.

Table VI and Figure 6 summarize performance results from applying the SOA approach to WSM6. Figure 6 indicates that setting the KNL to the flat mode yields better results than the cache mode. The flat mode maximum performance is about 1.5x the cache mode maximum performance. Overall,

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 V
SOA AND TRANSPOSE APPROACH APPLIED TO NISFL_RAIN_PLM6

SOA coupled with SIMD led to a speed-up of 70x. This performance result is about 23.3x faster than the results presented by Ouermi et al. [1].

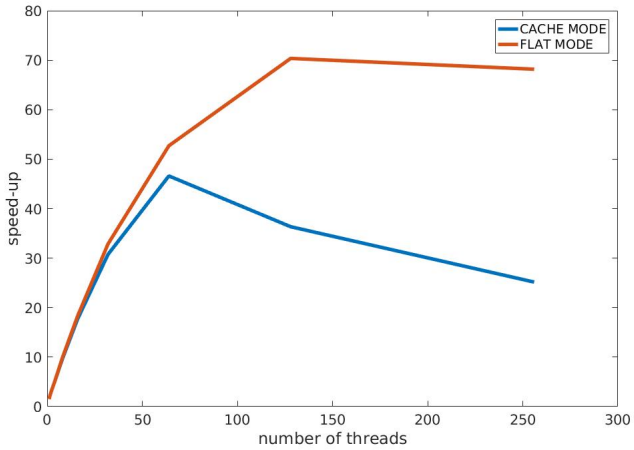

Fig. 6. SOA speed-ups on WSM6

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 VI
SOA APPROACH APPLIED TO WSM6 WITH FLAT AND CACHE MODES

VI. DISCUSSION

The results presented in previous sections indicate that the use of thread-local SOA is a suitable approach for optimizing WSM6 and other physics schemes in NEPTUNE. The standalone experiments were instrumental in identifying the appropriate techniques to optimize WSM6 and nisfl_rain_plm6. These experiments enable the study of different high-level and low-level optimization strategies that are not easily or trivially implementable in WSM6.

The size of the SOA is chosen to fit in the L2 cache. The data in the L2 cache are comprised of the inputs necessary to compute the physics in one or multiple columns. Since there are no dependencies between the columns, the computation is self-contained. This approach to data granularity reduces memory traffic and increases locality. The transpose approach to two-dimesnional loops requires code transformations that introduce temporary arrays. The temporary arrays often cause cache spills, thus increasing cache misses, which limit performance. In addition, the transpose approach often requires a significant code restructuring compared to the SOA approach.

The use of OpenMP directive OMP SIMD at the low level improves vectorization. Furthermore, it enables vectorization in the cases where the loop body has conditionals present. The dependencies along the k loop limit vectorization. This limitation is addressed by vectorizing along the i loop, which has no dependencies. Each SOA i loop is chosen to be a multiple of the vector unit length by setting the chunk size to 8, 16, 32, 64, 128, which improves data alignment.

With regard to peak performance, some of the challenges faced by the WSM6 code are illustrated by CODE 2 in Section V. 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 show a performance decrease from 128 to 256 threads with two to four threads per core. In the KNL, 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 between 128 threads and 256 threads.

VII. CONCLUSION AND FUTURE WORK

In conclusion, this study demonstrates the efficiency of a high-level method using thread-local SOA, coupled with low-level SIMD using OMP SIMD. As shown in the various experiments, this optimization approach enables a better utilization of the KNL resources by improving locality and vectorization. The use of thread-local SOA increases locality and decreases cache misses. The use of OMP SIMD along the i loop, coupled with chunk sizes that are multiples of SIMD length, improves vectorization. The high-level and low-level optimization techniques increase WSM6 performance by 70x speed-up compared to the original code and 23.3x speed-up compared to the results of Ouermi et al. [1]. As shown in the discussion, it is still a challenge to achieve high percentages of peak performance for the relatively simple and short loops of WSM6, and this is our continuing focus. As this work continues, we plan to investigate other optimization strategies with various data layouts, including a block-base data layout. In addition, applying the findings of this study to the remaining

physics modules in NEPTUNE may result in performance gains. Understanding how to better use hyper-threading may further improve performance on KNL. Studying performance of larger test cases with MPI and OpenMP level parallelism will help us understand how to better parallelize NEPTUNE on supercomputers.

ACKNOWLEDGMENT

This work is supported by DOD PETTT contract PP-CWO-KY07-001-P3, and by the Intel Parallel Computing Centers program. The authors would like to thank Alex Reinecke and Kevin Viner at Navy Research Laboratory, John Michalakes at UCAR, and Rajiv Bendale at Engility Corporation for their help and insights.

REFERENCES

[1] T. A.J. Ouermi, Aaron Knoll, Robert M. Kirby, and Martin Berzins. 2017. 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). ACM, New York, NY, USA, Article 12, 8 pages. DOI: https://doi.org/10.1145/3093338.3093387

[2] John Michalakes, Michael J. Iacono, and Elizabeth R. Jessup. 2016. Optimizing Weather Model Radiative Transfer Physics for Intelfis Many Integrated Core (MIC) Architecture. Parallel Processing Letters 27, 04 (2016), 1650019. DOI: http://dx.doi.org/10.1142/S0129626416500195

[3] Song-You Hong and Jeong-Ock jade Lim. 2006. The WRF Single-Moment 6-Class Microphysics Scheme (WSM6). Journal of the Korean Meteorological Society 42, 2 (April 2006), 129151.

[4] Jarno Mielikainen, Bromin Huang, and Hung-Lung Allen Huang. 2014. Intel Xeon Phi accelerated Weather Research and Forecasting (WRF) Goddard microphysics scheme. Geoscientific Model Development Discussions 7, 6 (December 2014), 89418973. DOI:http://dx.doi.org/10.5194/gmdd-7-8941-2014

[5] J. Mielikainen, B. Huang, and H. L. A. Huang. 2016. Optimizing Purdue-Lin Microphysics Scheme for Intel Xeon Phi Coprocessor. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 9, 1 (Jan 2016), 425438. DOI:http://dx.doi.org/10.1109/JSTARS.2015.2496583

[6] OpenMP Architecture Review Board. 2013. OpenMP Application Program Inter- face Version 4.0. (2013). http://www.openmp.org/wp-content/uploads/OpenMP4. 0.0.pdf

[7] J. Mielikainen, B. Huang, H. L. A. Huang, and M. D. Goldberg. 2012. 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 (Aug 2012), 12561265. DOI:http:

[8] Erik Price, Jarno Mielikainen, Bormin Huang, HungLung A. Huang, and Tsengdar Lee. 2013. GPU acceleration experience with RRTMG long wave radiation model. (2013). DOI:http://dx.doi.org/10.1117/12.2031450 //dx.doi.org/10.1109/JSTARS.2012.2188780

[9] Greg Ruetsch, Everett Phillips, and Massimiliano Fatica. 2010. GPU Acceleration Of The Long-wave Rapid Radiative Transfer Model in WRF Using CUDA Fortran.

[10] J. Michalakes and M. Vachharajani. 2008. GPU acceleration of numerical weather prediction. In 2008 IEEE International Syposium on Parallel and Distributed Processing. 1-7. DOI:http://dx.doi.org/10.1109/IPDPS.2008.4536351

[11] Jim Jeffers, James Reinders and Avinash Sodani, Chapter 4 - Knights Landing architecture, In Intel Xeon Phi Processor High Performance Programming (Second Edition), Morgan Kaufmann, Boston, 2016, Pages 63-84, ISBN 9780128091944, https://doi.org/10.1016/B978-0-12-809194-4.00004-1.

[12] Tom Henretty, Kevin Stock, Louis-Nol Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. 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), Jens Knoop (Ed.). Springer-Verlag, Berlin, Heidelberg, 225-245.

[13] P. R. Woodward, J. Jayaraj, P. H. Lin and P. C. Yew, "Moving Scientific Codes to Multicore Microprocessor CPUs," in Computing in Science & Engineering, vol. 10, no. 6, pp. 16-25, Nov.-Dec. 2008. doi: 10.1109/MCSE.2008.152

[14] P. R. Woodward, J. Jayaraj and R. Barrett, "mPPM, Viewed as a Co-Design Effort," 2014 Hardware-Software Co-Design for High Performance Computing, New Orleans, LA, 2014, pp. 33-40. doi: 10.1109/Co-HPC.2014.13

[15] F. X. Giraldo, J. F. Kelly, and E. M. Constantinescu. 2013. Implicit-Explicit Formulations of a ree-Dimensional Nonhydrostatic Unied Model of the Atmosphere (NUMA). SIAM Journal on Scientic Computing 35, 5 (2013), B1162B1194. DOI: hŁp://dx.doi.org/10.1137/120876034 arXiv:hŁps://doi.org/10.1137/120876034

[16] Hann-Ming Henry Juang and Song-You Hong. 2009. Forward Semi-Lagrangian Advection with Mass Conservation and Positive Definiteness for Falling Hy- drometeors. Monthly Weather Review 138, 5 (Septempber 2009), 17781791. DOI: http://dx.doi.org/10.1175/2009MWR3109.1

[17] James D. Doyle. 2017. A Next Generation Atmospheric Prediction System for the Navy. (January 2017). https://ams.confex.com/ams/97Annual/webprogram/ Paper304323.html.

[18] James D. Doyle, S. Gabersek M. Martini D. D. Flagg J. Michalakes D. R. Ryglicki P. A. Reinecke, K. C. Viner, and F. X. Giraldo. 2017. Next Generation NWP Using a Spectral Element Dynamical Core. (January 2017).

[19] Marco Danelutto, Tiziano De Matteis, Daniele De Sensi, Gabriele Mencagli, and Massimo Torquati. 2017. P3ARSEC: towards parallel patterns benchmarking. In Proceedings of the Symposium on Applied Computing (SAC '17). ACM, New York, NY, USA, 1582-1589. DOI: https://doi.org/10.1145/3019612.3019745

[20] 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. In Proc. of the 12th International Conference on Parallel and Distributed Computing and Networks (PDCN 2014). IASTED, ACTA press, Feb. 2014

[21] Gabriele Mencagli, Marco Vanneschi, Silvia Lametti, The home-forwarding mechanism to reduce the cache coherence overhead in next-generation CMPs, In Future Generation Computer Systems, 2017, , ISSN 0167-739X, https://doi.org/10.1016/j.future.2017.01.009. (http://www.sciencedirect.com/science/article/pii/S0167739X17300286)

[22] CUDA Fortran Programming Guide and Reference http://www.pgroup.com/doc/pgi17cudaforug.pdf

[23] Weather Research and Forcasting Model http://www.wrf-model.org/index.php