Optimizations of Unstructured Aerodynamics Computations for Many-core Architectures

Mohammed A. Al Farhan and David E. Keyes

Abstract—We investigate several state-of-the-practice shared-memory optimization techniques applied to key routines of an unstructured computational aerodynamics application with irregular memory accesses. We illustrate for the Intel KNL processor, as a representative of the processors in contemporary leading supercomputers, identifying and addressing performance challenges without compromising the floating point numerics of the original code. We employ low and high-level architecture-specific code optimizations involving thread and data-level parallelism. Our approach is based upon a multi-level hierarchical distribution of work and data across both the threads and the SIMD units within every hardware core. On a 64-core KNL chip, we achieve nearly 2.9x speedup of the dominant routines relative to the baseline. These exhibit almost linear strong scalability up to 64 threads, and thereafter some improvement with hyperthreading. At substantially fewer Watts, we achieve up to 1.7x speedup relative to the performance of 72 threads of a 36-core Haswell CPU and roughly equivalent performance to 112 threads of a 56-core Skylake scalable processor. These optimizations are expected to be of value for many other unstructured mesh PDE-based scientific applications as multi and many-core architecture evolves.

Index Terms—Performance optimization, Thread-level parallelism, Data-level parallelism, AVX-512, Knights Landing, SIMD, Computational aerodynamics, Unstructured meshes, Intel Xeon Phi

1 Introduction

Simulations of flows around aerodynamically efficient bodies that employ unstructured meshes to represent the complex geometry comprise a scientifically and commercially important part of the workload for large-scale distributed-memory computers, whose nodes derive most of their floating point performance from multi or many-core processors. The distributed-memory weak scaling of such codes under the paradigm of domain decomposition and bulk synchronous processing using MPI has been routine for more than two decades. However, strong scaling within a node to hundreds of threads is now also essential to exploit contemporary supercomputers and extend engineering analysis of aerodynamics to large-eddy simulation (LES) or even Direct Navier-Stokes (DNS) at high Reynolds numbers – a holy grail of aerodynamic analysis and design. The irregularity of the computation and indirect addressing in inner loops due to the unstructured meshes make this a daunting challenge. Practitioners are compelled to master a hybrid distributed-shared programming model and invest in tuning the code to extract the maximum possible performance on a specific hardware.

As High Performance Computing (HPC) architectures have been pushed for reasons of efficiency to accommodate a large number of low frequency compute cores in a single processor, programmer productivity has been squeezed. Optimizing and tuning existing scientific codes, while aided by compiler features, has become a highly manual endeavor that requires integrating awareness of a complex memory subsystem, convoluted on-chip interconnects, a massive number of concurrent threads, and wide Single Instruction Multiple Data (SIMD) vector units within a single CPU [1]. The battle for performance on the most powerful supercomputers is now fought mostly by modeling and experimentation targeting thread strong scaling within a single shared-memory compute node [2].

We undertake this agenda herein with an unstructured tetrahedral mesh Euler and Navier-Stokes research code closely related to the export-controlled state-of-the-practice code FUN3D from NASA. For over two decades FUN3D has been under active development for modeling fluid flow, and design optimizations of airplanes, automobiles, and submarines with a number of vertices up to 10 billion [3], [4], [5], [6], [7], both extending physical modeling as mission requirements dictate and pursuing architectural adaptability to exploit hardware opportunities. PETSc-FUN3D is a research fork of the Euler subset of the original FUN3D code that uses the Portable, Extensible Toolkit for Scientific computation (PETSc) solver framework [8], [9], [10]. PETSc features distributed data structures of both Cartesian and irregular type and provides a rich polyalgorithmic set of iterative linear and nonlinear scalable parallel solvers for the nonlinear algebraic equations that result at every implicit time step of the aerodynamics simulation [11]. The performance of PETSc-FUN3D has been well characterized for distributed-memory Single Program Multiple Data (SPMD), beginning with a 1999 Gordon Bell computation undertaken jointly by the primary architect of FUN3D and members of the PETSc development team [12]. It was revisited to explore the hybrid programming paradigm of MPI+OpenMP on the IBM Blue Gene/P architecture [13], with its modest number of cores per node. Later, It was extended in collaboration with Intel researchers to shared-memory optimizations and tuning on the multi-core Ivy Bridge Xeon architecture [14]. Recently, the code was ported into the first generation of Intel Xeon Phi architecture (Knights Corner (KNC) [15]), specifically exploring the IMCI instruction set of KNC in both modes, native and offload, as well as varying the thread distribution mechanisms across the cores [16].

New studies of the strong scalability of PETSc-FUN3D

• *Mohammed A. Al Farhan and David E. Keyes are with Extreme Computing Research Center, King Abdullah University of Science and Technology, Thuwal 23955-6900, Saudi Arabia.*
E-mail: mohammed.farhan@kaust.edu.sa and david.keyes@kaust.edu.sa

at the node level are required with the emergence of the Knights Landing (KNL) and Skylake architectures, to evaluate the effectiveness of new tools the same challenges of extracting thread and data-level parallelism in the presence of indirect addressing. Using a complete reimplementation in C of the Gordon Bell fork of the FUN3D application, we employ several low and high-level algorithmic and architecture-specific code optimizations, getting as far as Amdahl's Law permits without yet going inside of the PETSc solver, itself.

Key contributions of this work are:

- We demonstrate various state-of-the-art shared-memory code optimizations applied to key unstructured mesh PDE kernels to adapt to many-core architectures.
- To utilize the thread-level parallelism, we implement a fine-grained workload distribution mechanism that performs data partitioning and load balancing across both the OpenMP threads and the SIMD vector lanes.
- We extract the data-level parallelism via hand-written AVX-512 intrinsics to utilize the SIMD units, and a reordering algorithm to perform a fine-grained data partitioning within every thread's data set that promotes conflict-free subsets within a SIMD lane.
- Overall, we achieve, relative to the baseline version, approximately 2.9x improvement in the flux kernel, which consumes about half of the original overall runtime.

The rest of the paper is organized as follows. In section 2, we analyze the major computational routines of PETSc-FUN3D, and how they are optimized for many-core hardware via applying several algorithmic/architecture-specific procedures. Section 3 illustrates the mechanisms by which we apply data-level parallelisms to improve the unstructured code on the AVX-512 instruction set hardware. In Section 4, we specify the hardware and software environment and the design of the experiments. Section 5 reports results in conjunction with performance models. Section 6 points out the related work and discusses our contributions to the state-of-the-art many-core optimizations. Finally, we conclude with a summary of contributions and ongoing work in Section 7.

2 PETSc-FUN3D Computational Routines

We provide a scientific software engineering view of the PETSc-FUN3D application code, and describe underlying implementation details. We relegate details of the modeling, discretization, numerical analysis, and solver to other publications [11], [14], [16], and focus herein upon the HPC and implementation aspects of the code.

We have completely recoded the original PETSc-FUN3D application routines while considering several algorithmic and architecture-specific optimizations, some of which are inherited from the original code, where they have previously been explored on different computing architectures, and specifically enhanced for KNL hardware.

The FUN3D application has two computational phases that consume the majority of the overall execution time. These phases are: 1) preprocessing and setup phase, and 2) the callbacks from PETSc's ψNKS solver. Following subsections describe these two phases in more detail.

2.1 Preprocessing and Setup Phase

In this phase, the code reads an input mesh data file consisting of domain-decomposed components. The mesh file is a "Big-Endian" binary that is physically stored in the hard disk drive. Its subdomain components are manipulated separately: ordered, partitioned, and stored in specific segments of the heap memory. In addition, this phase involves I/O, which is often a costly operation, and relatively more expensive on a many-core processor. In our implementation, the input mesh file stream is loaded initially from the hard disk drive to the main memory with a single I/O operation. Then, low-level parser routines manipulate every mesh component directly from the main memory, as follows.

Stream Buffer Parsing Phase: We develop a task-based parallel routine using OpenMP's task pragmas that takes a range of bytes, based upon the size and data type of the mesh components, and distributes chunks of equal size in bytes on the running OpenMP tasks. We use the divide-and-conquer mechanism to ensure equal workload distribution across the running tasks. In addition, every task walks through their assigned chunks of buffer, performs a word-by-word[1] swapping to convert the bytes sequence order from the "Big-Endian" source to "Little-Endian" internal format, and then places them in their allocated memory buffer. This phase ensures a fast and concurrent stream bytes parsing of the input mesh file.

Data Allocation and Reordering Phase: We then carry out a parallel data ordering to keep mesh components stored contiguously in the memory. This aims to preserve the spatial and temporal cache locality for the data items that are successively referenced in time; it is based upon the implementation of [11] with minor KNL-specific enhancements. Since this phase is primarily done through multiple *vertex-based* loops [13], [16], it is easy and straightforward to parallelize via OpenMP parallel for loop directives and static scheduling for the threads.

2.2 ψNKS Kernels Phase

PETSc implements the Pseudo-transient Newton Krylov Schwarz (ψNKS) parallel implicit algorithmic framework [17], [18], [19], [20], relying on its own linear algebraic kernels and on callbacks to the application. Callbacks from the PETSc solver to evaluate the residuals of the discrete PDE conservation laws and to perform other algebraic actions consume the majority of the PETSc-FUN3D runtime. They come at different levels in the nested loops of preconditioned Krylov iterations inside of Newton iterations inside of implicit timesteps.

2.2.1 Sparse Linear Algebraic Kernels

The PETSc library features Krylov subspace iterative methods (e.g., GMRES) with domain-decomposed (e.g., Schwarz) and subdomain (e.g., ILU) preconditioners. In the scope of the current work, PETSc is a "black-box". Profiling reveals the most important kernels to be the factorization that is done once per preconditioner evaluation and the sparse triangular solves that are performed in each application of the preconditioner. The concurrency available in these kernels is rather limited in terms of the shared-memory parallelizations [14]. They consume approximately 26% of the execution time prior to optimization of the FUN3D kernels. As an important open-source framework in hundreds

1. The word size is based on the data type: 8 bytes for double and 4 bytes for integer. For the integer data, we use `__bswap_32()` function available in byteswap.h of GNULib. However, for the double data we implement a local swapping function.

of appliations spanning dozens of areas in computational science, the hybrid implementation of PETSc's linear algebra kernels is beyond the scope of this paper, but, of course, their performance benefits without code modification from the orderings discussed here.

2.2.2 Edge-based Loop Kernels

In the overall application, edge-based loop kernels are the key routines, in which roughly 73% of the entire runtime is spent, prior to optimization. These kernels perform local stencil evaluations to calculate the residual vector, construct the preconditioner for the Jacobian, and form the Jacobian-vector product for the Newton-Krylov solver. (Note that the exact Jacobian is never formed, just its preconditioner, in which shortcuts are taken so that it occupies less memory and requires fewer flops to apply.) In the context of the software, edge-based loop kernels dominate in four different routines: 1) flux evaluation, 2) gradient evaluation, 3) preconditioner construction, and 4) pseudo-time step calculation for each mesh cell. The flux evaluation routine is a compute-intensive kernel that consumes in excess of 50% of the overall execution time. Therefore, the focus of our work is optimizing and scaling the flux routine.

Spatial and Temporal Locality of Reference: To alleviate the poor cache locality of reference of the indirect addressing, we systematically arrange the endpoints of the edges in a sequence ordered ascendingly. This enhances the access patterns of the auxiliary (velocity components in the three Cartesian directions (u, v, w) and the pressure (p)) and geometry data structures, which leverages the deep memory hierarchies. The ordering is carried out via an OpenMP task-based parallel merge-sort algorithm that performs recursive pointer swapping to build an index array. The index array is then utilized via an OpenMP parallel *vertex-based* loop to sort the left endpoints of the edges in increasing order. Then, the right endpoints and the edges' normals are accordingly sorted, so that the edges index table, which contains the edges' components data (i.e., endpoints and normals), is later traversed with one iterator pointer that is sequentially incremented. Hence, a kernel loop that iterates over the stencil data items is essentially transformed from a loop over the edges into a loop over the vertices [11], in which the iterative traversing is linearly based upon the left endpoints. For instance, in Figure 1, the leaf pointer node ($n0$) contains the sorted left endpoints of the edges (e.g., $0, 0, 0, \ldots, 1, 1, 1 \ldots, \ldots$). According to the sequence order of $n0$, we sort the right endpoint of every edge ($n1$) as well as the edge's normals (x, y, z, ln) of both endpoints. As such, at every cache line fetch/prefetch, the successive data items in the vector registers are reused multiple times before the cache line is disposed. Thus, the spatial and temporal locality of reference of the data items brought into the different cache levels are maximized. We increase the computations to the peak by performing significant time step updates to a single vertex before evicting the data to main memory. Consequently, once the data are evicted, they are less likely to be used again, which efficiently exploits the recently visited vertices without overruling the naturally enforced data dependencies. For example, a typical cache line of a 512-bit SIMD unit contains 16 edge indices. For simplicity, assume a cache line holds 16 elements of index 0's. Another cache line would have the 16 elements of the right endpoint indices associated with the index 0. the third and fourth cache lines will have 8 elements of the left and right endpoints normals, respectively[2]. Since index 0 is already present in the cache with all of its associated items, we process the vertex and its neighbors first before it gets discarded. Although we can only process 8 elements at a time, the other 8 elements are kept closely spaced in the L1 and L2 caches (using SIMD prefetching instructions) for faster read. However, a downside of our approach is that the data items of $n1$ and its normals will be refetched several times as needed based upon the cache replacement policy. Nevertheless, this can be almost negligible, since the kernels parse the stencil data items based on the left endpoints and their neighboring vertices. Likewise, the Degrees-of-Freedom (DoF) of every vertex are gathered contiguously in the memory, so that parsing the stencil working set based on the neighboring vertices is maintainable within the cache line boundaries. Since they are placed contiguously in the memory, we can exploit the strided gather/scatter instructions, on which we essentially lean to support vectorizing the kernels.

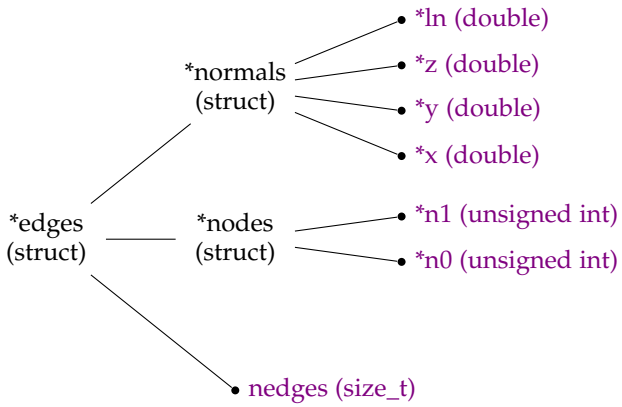


Fig. 1. Edges data structure tree

Geometric Data Layout: To mitigate the cost of the indirect addressing from unstructured meshing, geometric information is stored in multiple *Struct-of-Arrays* (SoA) data structures. The multiple nested C structs layout forms a tree structure with leaf nodes of multiple C pointers that point to different contiguous chucks of the heap memory (see Figure 1). Hence, geometric information is maintained in multiple cache lines, each of which is aligned to a 64-byte boundary so that at every cache line load, the required data being processed are close to the CPU. This layout that mimics the tree structure is vital for code vectorization, especially when we manually *hardcode* the vector instructions via handwritten intrinsics. Nevertheless, this technique introduces *pointer-chasing*, in the presence of referencing/dereferencing a memory location, which can be an optimization bottleneck. To overcome this issue, we pass only the pointers at the leaf nodes of the tree to the computational kernels, so that within the loop iterations, we directly reference the address of memory locations without manipulating pointer arithmetic.

Residual and Gradient Data Layout: We implement a variant of the hybrid/tiled *Array-of-Structs* (*Array-of-Structs-of-Arrays* (AoSoA)) [21], [22], [23] data structure layout to

2. The normals data are stored using double primitive data type, and all the kernels computations are carried out in double precision.

store the residual and gradient vectors. For simplicity, we call it *Array-of-Structs-of-Strided-Arrays* memory layout, in which the struct's arrays are tiled (see Figure 2). The tile size is equal to the number of the state variables (i.e., 4 DoF per mesh vertex). The gradient vector has 2 sets per cache line for every coordinate (i.e., for x, y, and z coordinates, so we have in total of 3 cache lines, 6 sets, and 24 elements). This layout is important for *loop unrolling*, by which the instructions that control the loops are minimized [24]. It also reduces the amount of pointer arithmetic performed per loop iteration to jump from one consecutive memory chunk to another [25]. Furthermore, at every loop iteration, a contiguous set of data within a coordinate are fetched into multiple SIMD lanes. The *Array-of-Structs-of-Strided-Arrays* layout can be vectorized straightforwardly with hand-written intrinsics via issuing multiple strided gather/scatter vector instructions. Furthermore, PETSc routines well exploit our data structures tiling, padding, and striding that are implemented herein to construct the right-hand side preconditioner sparse matrix (i.e., Jacobian matrix). We thereby use `MatCreateBAIJ` routine of PETSc to create a "structural blocking" Compressed Sparse Row (CSR) matrix with the block size equal to the tile size [11], through which the linear algebra operations of PETSc are highly optimized.

(A)

struct {

X:	u	v	w	p	.....	u	v	w	p
Y:	u	v	w	p	.....	u	v	w	p
Z:	u	v	w	p	.....	u	v	w	p

} Gradient;

(B)

struct {

Q:	u	v	w	p	.....	u	v	w	p

} Residual;

Fig. 2. Gradient and residual vectors data structure

Edge Workload Distribution and Load Balancing: To partition the edges across the running threads, we use the METIS graph partitioner [26], [27]. METIS searches for an adequate load balance across the OpenMP threads, by which it aims to minimize the aggregate cross edges' weight (i.e., reduce the edge-cut). To apply METIS, we build a routine that extracts a sparse graph connectivity representation of the mesh. We then reorder the vertices using a variant of the breadth-first traversal algorithm (i.e., Reverse Cuthill-McKee [28]) that is commonly used to reduce the fill-in of the sparsity pattern in matrix factorization. The constructed sparse graph representation is used by the multilevel k-way partitioning scheme of METIS to divide the workload uniformly across the threads. In particular, METIS generates almost equal partitioning for the nodes between the OpenMP threads. Afterwards, we use the generated partitioning to equidistribute the edges and their associated workload across the threads. In addition, this distribution conserves thread safety without the need of atomic operations by using work replication. For instance, if two threads share an edge, the work per edge is replicated for the both threads. This results in redundant computations performed by the two threads. Nevertheless, only the thread that owns the endpoint is allowed to write-back to the shared buffer [29]; in the context of our code development, we call that thread the "master" thread. The METIS strategy of redundancy in computations purges the overheads of using global synchronizing barrier between the working threads, since every thread works with its private buffer. It maximizes the throughput by launching and exploiting as many threads as the hardware permit. Thus, each thread works asynchronously with only a local synchronization barrier imposed by a conditional statement to orchestrate the SIMD computations and the shared write-back operations. In the vectorized code, the conditional statement is replaced with bitwise masking operations to eliminate branches. To mitigate the overhead of mesh partitioning and load balancing phases, which include invoking METIS partitioner, we leverage thread-level parallelism whenever it is possible relying on both OpenMP tasks as well as OpenMP parallel for loop. Similarly, we exploit the OpenMP version of METIS (i.e., *MT-METIS*). Thereby, our largest mesh experimental results demonstrate that the contribution of these steps to the bulk of the execution time becomes negligible, as manifested in the Performance Results and Analysis section (Section 5). With the thread partitioning using METIS, a skeleton of an edge-based loop code is shown in Listing 1.

```
1 #pragma omp parallel
2 {
3   const uint32_t t = omp_get_thread_num();
4   for(uint32_t i = ie[t]; i < ie[t+1]; i++){
5     /*Load and compute*/
6     if(parts[n0[i]] == t)
7       {/*Write-back into v[n0[i]]*/}
8     if(parts[n1[i]] == t)
9       {/*Write-back into v[n1[i]]*/}
10  }
11 }
```

Listing 1. A code skeleton of an edge-based loop

Explicit Memory Management: To utilize the MCDRAM of the KNL, we develop a low-level heap allocator routine on top of the Intel *memkind* heap manager library [30] and *jemalloc* library [31] to allocate the address space. In particular, we rely on `hbw_posix_memalign_psize()` function of the hbwmalloc API with 64-byte memory alignment (size of a KNL cache line), and a page size of 4 KB (`HBW_PAGESIZE_4KB`). We keep a variable for the alignment parameter, so that we can alternately vary the memory alignment to a specific boundary of one (64-byte), two (128-byte), and four (256-byte) cache lines, depending upon the requirements of the data structures. We further focus on improving the structure padding to eliminate *false sharing* across threads operating on a cache line [32].

Simplifications of the Gradient Kernel: We further investigate the gradient kernel for plausible performance advancements. We find that part of the kernel's arithmetic computations can be precomputed once before the kernel execution, and stored the results in the memory. Hence, we reformulate some of the kernel's instructions so that we compute once, and rely on fetching them from the memory. We minimize the indirect addressing of the kernel by replicating the precomputed arithmetic in the memory for every edge. Of course, this results in extra storage requirements (i.e., almost 1 GB extra). However, we in turn reduce the number of the gather instructions that are performed by the kennel from 70 to 50 instructions per loop iteration, and we replace them with an extra 6 load instructions.

Thread Partitioning of the Boundary Solid Faces: Computations enforcing boundary conditions require iterating over the mesh boundaries. Typically these computations

are not very expensive compared to the loops over interior edges, which are part of complex conservation laws. Nevertheless, to improve the overall timing, we use a variant of the edge coloring algorithm to distribute the triangular faces across the threads. Therefore, every thread can safely compute on their workload within a color, without explicit need for locking mechanisms. The edge coloring algorithm implemented is a modification of Algorithm 1, explained and described in Section 3, which is used for data-level parallelism. The key difference between the two implementations is that we color three vertices that form a triangular face, instead of two. Furthermore, we do a parallel prefix sum after the coloring to assign the start/end working set indices for every color a thread can operate on. With the thread partitioning using edge (face) coloring, a skeleton of the code that loops over the boundaries is shown in Listing 2.

```
1 for(uint32_t i = 0; i < ncolors; i++) {
2   const uint32_t t = omp_get_thread_num();
3 #pragma omp parallel for
4   for(uint32_t j = ifc[i]; j < ifc[i+1]; j++){
5     /*Load and compute*/
6     /*Write-back into:*/
7     /*v[fn0[j]]; v[fn1[j]]; v[fn2[j]]*/
8   }
9 }
```

Listing 2. A code skeleton of a loop over the boundaries

3 Data-level Parallelism

Mainstream x86 CPUs feature instruction set for SIMD operations on multiple vectorized data sets. In order to exploit these operations, the code has to be vectorization friendly, which may require data layout modifications. Even though relying on the compiler's auto-vectorization to maintain code portability can save coding efforts, the compiler could fail to extract the vector code for some applications. The primary cause for this failure is the ambiguity of "assumed" data dependencies and conflicts within a prefetched SIMD lane. Some codes may require extreme efforts of assembly-like, low-level programming beyond data structure transformations or data access reformulations.

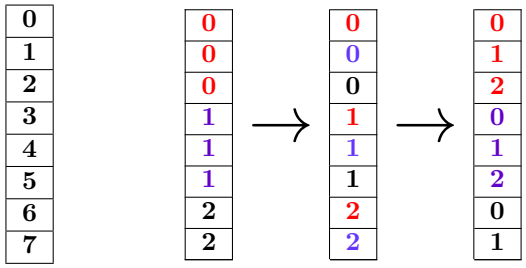
In the context of PETSc-FUN3D, the Intel compiler successfully auto-vectorizes most of the kernels (via `#pragma simd`) as a consequence of our low-level code optimizations and tuning. However, it fails to vectorize the most compute intensive kernels: the flux and gradient kernels. Due to irregular data access, which causes indirect memory addressing to reference the address space, the compiler struggles to generate conflict-free vector instructions. In addition, these kernels, especially the flux kernel, perform a large number of floating point operations within a single loop iterations. Since the flux kernel consumes more than 50% of the total execution time, leaving it unvectorized is unacceptable. Hence, our major efforts are devoted first to restructuring the kernels, and then developing hand-written vector codes via utilizing the AVX-512 intrinsics of KNL and Skylake Instruction Set Architecture (ISA) [33].

3.1 Vectorizing Edge-based Loop Kernels

Consecutive edge components are fetched via vector load instructions, whereas the indirectly addressed data are loaded via gather instructions (i.e., `VGATHERDPD`). The write operations, on the other hand, are all performed via utilizing the scatter instructions (i.e., `VSCATTERDPD`). Nearly all of the arithmetic operations are clubbed in such a way that every add/subtract and multiply pair is fused together in a single instruction. Hence, some of the kernel's mathematical calculations are restructured and reformulated so that they become Fused Multiply-Add (FMA)-friendly to utilize `VFMADD`, `VFMSUB`, `VFNMSUB`, and `VFNMADD` AVX-512 instructions. Furthermore, the control-flow (comparison/branching) operations are transformed into data-flow through exploiting the masking instructions to use the available vector mask registers for conditional evaluations. To utilize the transcendental hardware support of KNL, all of the *square root* and *division* operations are converted into reciprocal operations (i.e., `VRSQRTXXPD`, `VRCPXXPD`), through which we mitigate the cost of performing these operations. For example, to avoid an explicit square root instruction, we convert $\sqrt{x}$ to: `_mm512_mul_pd(_mm512_rsqrtXX_pd(x),x)`. Also, to avoid an explicit division instruction, we convert $\frac{c}{x}$ to: `_mm512_mul_pd(_mm512_rcpXX_pd(x),c)`.

Our data reordering for cache efficiency assists in the utilization of the AVX-512 prefetching instructions, since the next referenced data are successively stored in memory. At the very beginning of every loop iteration, we perform a number of software gather prefetching instructions (i.e., `VGATHERPF1DPD`) to fetch the next SIMD lane, before its data are needed, which helps to populate the L2/L1 caches. The prefetching gather instructions of KNL minimize the effects of indirect addressing bottlenecks. For example, the next working set of the indices are prefetched alongside the current working set, since at every iteration we populate 16 unsigned int vector lanes per Vector Processing Unit (VPU). However, we proceed with only 8 lanes per VPU, since the arithmetic computations are done in double precision, which is at most 8 double vector lanes per VPU.

The update (write-back) operations, on the other hand, are implemented via Conflict Detection (CD) instructions of AVX-512. Figure 3 (A) shows an index array that is free of conflicts, where an update operation can be safely performed within a SIMD lane. However, Figure 3 (B) presents an array with three groups of dependent indices. Hence, the indirect addressing updates that rely upon such an index array would overwrite the results of the three indices. This is a form of data hazard that effectively results in race conditions within a SIMD lane.



(A) Conflict-free Array (B) Array with Conflicts

Fig. 3. An illustration of conflict detection

To resolve the data hazards conflicts via the AVX-512 intrinsics, the update operations are issued several times on the distinct indices only. We, in particular, rely upon

multiple swizzle instructions to carry out data replication through broadcast operation. In other words, the swizzle instructions permute the data elements of the source operands, before the arithmetic calculations. They therefore clone the input data elements to create "multiplexed data patterns" that are used as sources for the arithmetic operations. In addition, to avoid modifying the original source operands, the "multiplexed data patterns" are generated via temporary values that are discarded when the operations are completed [34]. Furthermore, we construct two different write masks, one is based on the thread ID and the endpoint owned by the thread, and the other one is based on the conflicted indices. Thus, initially the write-back loop uses the first mask to perform the conflict detection operations, after which the second mask is generated. Then, the first mask gets overwritten by the second mask, while keeping a copy of the first one to be used in the next inner loop iteration. For instance, in Figure 3 (B), there are three write-back operations that occur on three subsets (red, purple, and black). Thus, instead of performing 8 concurrent updates, we perform three distinct concurrent updates operations. The worst case scenario is to perform 8 distinct updates, when the SIMD lane data are completely dependent. However, due to our reordering phase, the occurrence of the worst case scenario is very rare, and very often the write-back follows the average case, where it performs almost 2 to 3 distinct updates. The following describes the steps that we follow herein to perform the write-back operations with the AVX-512 CD instructions. Further, with code vectorization and conflict detection, an edge-based loop kernel with thread partitioning skeleton is presented in Listing 3, which extends the Listing 1 code.

1) Generate an initial mask based on the thread ID and node index
2) Scan the SIMD lane to identify the conflicted data
3) Generate a mask that separates a conflict-free data subset
4) Perform a safe gather mask operation (load)
5) Update the operands with multiple FMA instructions
6) Perform a safe scatter mask operation (write-back)
7) Perform SIMD boolean operation to remove the already written indices from the mask
8) Repeat the aforementioned steps again on all of the remaining subsets within the SIMD lane

```
#pragma omp parallel
{
  const uint32_t t = omp_get_thread_num();
  const uint32_t l = ie[t+1]-((ie[t+1]-ie[t])%8);
  for(uint32_t i = ie[t]; i < l; i += 8){
    /*Load and compute on the SIMD lane elements*/

    /*The Write-back inner loop*/
    _mm512_castsi256_si512(/*...*/);
    _mm512_cmpeq_epi32_mask(/*...*/);
    do {
      _mm512_mask_conflict_epi32(/*...*/);
      _mm512_broadcastmw_epi32(/*...*/);
      _mm512_mask_testn_epi32_mask(/*...*/);
      /*Gather, compute, and scatter*/
      _mm512_kxor(/*...*/);
    } while(/*...*/);
  }
  /*Peel and remainder loop*/
  for(uint32_t i = l; i < ie[t+1]; i++){
    /*load and compute*/
    if(parts[n0[i]] == t)
      {/*Write-back into v[n0[i]]*/}
    if(parts[n1[i]] == t)
      {/*Write-back into v[n1[i]]*/}
  }
}
```

Listing 3. A skeleton of a vectorized edge-based loop

3.2 Fine-grained Data Partitioning

As mentioned before, in the preprocessing step the edge data are sorted in increasing order to improve the cache locality of reference. In addition, a thread-level partitioning is performed to break down the edges workload across the running threads. This is a form of fine-grained parallelism within the shared-memory context that facilitates the load balancing. However, for the purpose of clarification, we refer to this partitioning as a coarse-grained level of parallelism, standard in MPI-based distributed-memory parallelism. We refer herein to the data-level partitioning within the SIMD unit as fine-grained parallelism, which is restricted to the size of the SIMD unit of the hardware.

To improve the vectorization efficiency and increase the size of the independent data subset within a SIMD lane, we further reorder the edge data. To perform the reordering, we develop a bucket sorting routine based on a variant of the edge coloring algorithm that extracts independent subsets of the edges. The bucket sort method employs a partial coloring [35], [36], in which it orders the edges heuristically so that the indirect increments of the endpoints indices form subsets of maximum conflict-free edges. Further, the CD instructions of KNL handle the possible occasional conflicts, which should be very minimal. In addition, the reordering aims to procure large subsets of independent data to prune the overhead of the innermost CD loop. Every thread resorts its workload by assigning every 8 distinct edges to a single color. The resorting could potentially disrupt the initial ordering for cache spatial and temporal locality of reference efficiency. However, we purposely restrict the sorting to extract only a bucket of at most 8 independent edges within a color. This resorting is different than the initial sorting in that it assigns the two endpoints of every edge to the same color. Hence, we assure that the write-back operations can safely be performed within a color with minimal conflicts. Of course, this causes an issue of finding large subsets with maximum size of independent data. However, as presented later in the results, this resorting is technically a finer-grained partitioning within the thread-level partitioning, so it is expected to become more valuable as we increase the problem size. Algorithm 1 shows a pseudo code for our implementation of the edge coloring sorting scheme. Every thread loops over its assigned edges and colors at most 8 edges (16 endpoints) with the same color. Within every color, we adjust the edges data next to each other in the data structure to ensure consecutive sequential access, and to preserve the memory references closely spaced.

Since every thread maintains its own private working buffer, the memory can be overfilled; especially when we launch an immense number of working threads (e.g., 288 threads in a 72-core KNL). Thus, to keep track of the color assignments of every endpoint, we use a bitmap data structure that performs bitwise operations for every color assignment. Using a bitmap has a great advantage in terms of the memory usage footprint. In addition, performing bitwise operations are significantly faster in comparison to other operations. Every bit of the bitmap represents a

Algorithm 1 Bucket sort based on edge coloring algorithm

1: Create an `nedges_per_color` array to track the colors
2: Create a bitmap to track the endpoints coloring scheme
3: **for** $c \leftarrow 0, k \leftarrow 0, i \leftarrow 0$ to $nndges$ **do**
4: Clear all of the `bitmap` bits
5: **for** $j \leftarrow i$ to $nndges$ **do**
6: Read $n0[j], n1[j]$ bits from the `bitmap` $\rightarrow b0, b1$
7: **if** $b0 \vee b1$ is TRUE **then**
8: CONTINUE
9: **end if**
10: **if** `nedges_per_color[c]` = 8 **then**
11: BREAK
12: **end if**
13: Set $n0[j], n1[j]$ bits in bitmap
14: Swap edge data from index j to k and vice versa
15: Increment k and `nedges_per_color[c]` by 1
16: **end for**
17: Increment c by 1
18: $i \leftarrow k$
19: **end for**

mesh vertex index to trace the coloring scheme of the edges endpoints (see Figure 4). Further, every thread constructs, allocates, and manipulates its own bitmap data structure. For example, in Figure 4, we have two SIMD registers that track 8 edges (i.e., in total 16 endpoints). At every loop iteration, a thread would construct a bitmap bucket that points to the distinct edges indices within the SIMD registers (i.e., line 16 of Algorithm 1). To construct the bitmap, we set the bits value (from 0 to 1) based on every vertex index. For instance, bitmap bucket 1 of Figure 4 contains the edges with the green color (i.e., vertices: 0, 1, 2, 3, 4, 5), which means these edges are completely independent and conflict-free within the two SIMD lanes. As such, they can be safety processed in parallel. Next, the already colored indices (e.g., green color) will be masked out in the next iterations, and the algorithm will color the next working set (i.e., bucket 2 (red color), bucket 3 (blue color), and finally, bucket 4 (black color)). The bitwise operations herein are low-level implementations of the bitwise set, clear, and logical OR (for comparison), which endorse fast and accurate execution of the algorithm.

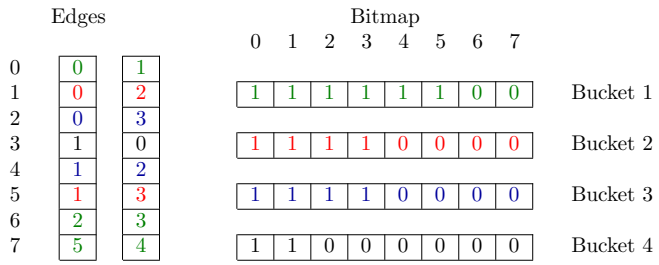


Fig. 4. Illustration of the bucket sort based on edge coloring

The kernels, thereafter, fetch every bucket individually within every loop iteration via load and gather AVX-512 instructions. Furthermore, during the write-back operations, the kernels can safely assume the absence of race condition within the SIMD lane. Nonetheless, in case of fetching a bucket where the extracted elements has subsets with less than 8 independent edges, the AVX-512 CD instructions is utilized to deal with that behavior similarly as it is expressed earlier.

4 Experimental Setup

In the original PETSc-FUN3D code, the ψNKS phase is executed twice [11], [12], [13], [14], [16], [37] . The first one is a brief execution that performs a single pseudo time step with one nonlinear iteration to page in the data and warm up the memory hierarchy. The aim of the authors was to concentrate on the best possible performance of a typical timestep. However, in our implementation of the code, we purge the brief execution step, and invoke the kernels directly to report the most representative performance measurements irrespective of the hardware states and conditions. Furthermore, we use several state-of-the-practice scientific performance engineering methodologies to overcome any possible performance variations caused by the hardware [38].

We use the compiler-specific implementation of the *RDSTC* time stamp counter instruction (`__rdtsc()`) to measure the execution time. *RDSTC* returns the number of the clock cycles spent by the CPU. To convert the cycles to seconds, we divide that number by the CPU clock frequency. For precise CPU frequency measurement, we use both the MKL-specific implementation: (`mkl_get_clocks_frequency()`), and we implement our own function that estimates the frequency of the CPU. In addition, to verify our timing calculations, we use the wall clock time function of both POSIX API implementation and of the OpenMP implementation. To read the hardware performance counters for the memory bandwidth utilization, we develop a low-level interface customized to KNL, which exploits the POSIX performance monitoring API [39]. We, in particular, report the MC (DDR)/EDC (MCDRAM) RPQ and WPD inserts from the Performance Monitoring Unit (PMU) registers [40].

The reported runtime results are summarized using the arithmetic mean across multiple independent runs that form the sample space. However, reported memory bandwidth and floating point rates are summarized using the harmonic mean of the arithmetic mean of the absolute counts across multiple independent runs [38]. Unless otherwise reported, we average more than 20 runs for every experiment, and before every run, the source code is recompiled, and the memory and cache are completely flushed. An error bar is drawn to show the +/- standard deviation of the mean for each experimental sample. The core performance experimentations are carried out on a mesh of size $2,761,774$ vertices ($11,047,096$ DoF) and $18,945,809$ edges. In 1999 [12] this was considered to be a large mesh but it is hosted conveniently today within a single node.

Software and Hardware Specifications: The source code is written purely in C. We use PETSc release version 3.8 built on top of Intel Parallel Studio version 2018 Update 1, which includes Intel C/C++ compiler, OpenMP, MPI, and Math Kernel Library (MKL). For graph partitioning, we use METIS version 5.1 and ParMETIS version 4.0. All of the experiments are performed with the `-O3` compiler optimization flag, and we set the OpenMP affinity to scatter: `KMP_AFFINITY=scatter`. In addition, to further ensure the threads pinning and binding, we use the Linux system call for CPU affinity mask available in `sched.h`:

`sched_setaffinity()`. So, we *hard-code* the pinning and binding of the thread contexts to target a specific quadrant/tile/core on KNL, and a specific socket/core on CPU. Further, we use `numactl` tool of Linux to ensure the memory channels binding and interleaving. We use 6 different servers, each of which is powered with a different Intel processor architecture. Table 1 summarizes the specifications of each chip within every server. In particular, we use three different KNL chips, each of which has a different configuration. Utilizing different KNL dies allows to experiment over a wide range of different configurations, including the power scaling, frequency driver, number of cores, and Turbo boost. However, some of the KNL results are reported for a specific die to save space whenever a particular configuration does not undergo a significant performance impact, and we focus upon exhibiting more of breadth test cases. In addition, in [41], [42], [43], [44], [45], details of the key aspects of the primary architecture considered herein (i.e., KNL) are illustrated.

TABLE 1
Hardware specifications

	A	B	C	HSX	BDX	SKX
Family	x200	x200	x200	E5V3	E5V4	Scalable
Model	7210	7210	7290	2699	2680	8176
Socket(s)	1	1	1	2	2	2
Cores	64	64	72	36	28	56
GHz	1.30	1.30	1.50	2.30	2.40	2.10
Watts/socket	215	215	245	145	120	165
DDR4 (GB)	96	96	192	264	132	264
Freq. Driver*	I	A	A	A	I	A
Max GHz	1.50	1.30	1.50	2.30	3.30	2.10
Governor**	P	C	C	O	P	O
Turbo Boost	×	✓	✓	✓	✓	✓

Freq. Driver*: I: Intel pstate; A: Acpi-cpufreq
Governor**: P: Powersave; C: Conservative; O: Ondemand

5 Performance Results and Analysis

Before performing optimizations on PETSc-FUN3D, we evaluate the potential of certain selections of the KNL clustering/memory modes. Figure 5 shows our optimized flux routine performance with respect to the choice of the KNL configuration modes. Our results align with the findings of [46], in which a KNL performance model is developed to micro-benchmark the implications of varying the clustering/memory modes. They show nearly negligible influence on performance of cluster and memory mode combinations, except for SNC-4, where there is a small perceptible degradation in the runtime. Thus far, SNC modes are experimental modes [41]. Therefore, application code performance on SNC modes is highly unpredictable and, thereby, very hard to justify it [46]. Nonetheless, to further ensure that SNC mode specifically does not have an influence on the performance of our application, we enable SNC on Skylake [47], and we experiment the code there. The performance of our code is almost similar in both cases on Skylake (i.e., SNC enabled/disabled), which essentially means that SNC mode has not effect on the code performance likewise other modes. Accordingly, our future experiments are conducted with the default KNL modes (i.e., Quadrant/Flat).

In most of our performance experiment figures, we present three different thread combinations, as follows: 1 thread/core, 2 threads/core, and 4 threads/core (KNL); 1 thread/core and 1 socket, 1 thread/core and 2 sockets, and 2 threads/core and 2 socket (multi-core x86 architectures). In the case of having more than one thread per core, we aim to compare the performance impact of hyperthreading on a specific optimization[3]. However, the likelihood of having a significant performance boost out of hyperthreading is minimal, since the threads would compete to gain the shared core's resources, as manifested in our upcoming findings. On the other hand, experimenting with the effect of having the workload reside on a single socket as opposed to dual-socket leads to very insightful findings that express the behaviors of NUMA on the performance, and whether our optimizations are NUMA-aware or not.

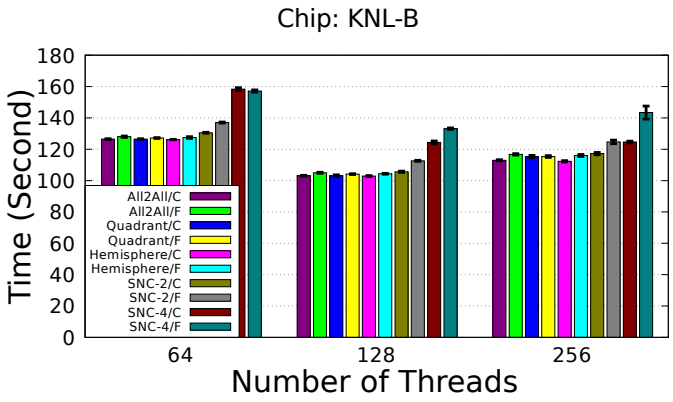


Fig. 5. Flux performance with different KNL clustering/memory modes [SNC: Sub-NUMA Clustering mode; F: Flat mode; C: Cache mode]

5.1 Performance of the Flux Routine

Figure 6 presents the incremental progression of the flux kernel performance from the baseline to the most optimized code. The baseline code is the FORTRAN 77 code of [16] running out-of-the-box on KNL. The optimized code, on the other hand, is our reimplementation of PETSc-FUN3D that is written in C, and enhanced with several KNL-specific optimizations. "HBW" means that the code explicitly uses our high-bandwidth heap allocator routine to allocate the MCDRAM.

The initial HBW transformation provides only a small improvement compare to the baseline, whereas using the `-xMIC-AVX512` compiler switch[4] boosts the performance by a speedup of almost 1.6x. In terms of the scalability, the three code versions (baseline, optimized, and `-xMIC-AVX512`) strong thread scale monotonically with hyperthreading.

In contrast to both the baseline as well as the optimized code, using the MCDRAM and data-level parallelism optimization via vectorization with the handwritten intrinsics advances the performance to almost 2.9x speedup. Further, without the MCDRAM, the performance speedup drops to 2.5x compare to the baseline. The gain from the handwritten vectorization, however, is diluted down to almost 1.8x

3. The choice of 2 threads/core on KNL is to launch one thread for each VPU; and 4 threads/core is to launch one thread for each VPU and interleave another extra thread to maximize the throughput.

4. `-xMIC-AVX512` compiler optimization switch generates Intel AVX-512 instructions for KNL.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2826533, IEEE Transactions on Parallel and Distributed Systems

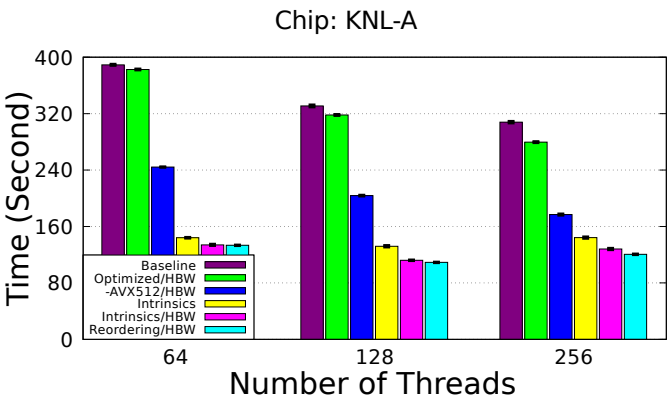


Fig. 6. Flux performance optimizations

speedup in comparison to the optimized code compiled with the `-xMIC-AVX512` switch.

The speedup from vectorization with edges reordered to minimize the data dependencies and conflicts within the SIMD lane has marginal value compared to the vectorization without reordering. Indeed, it does not significantly improve the performance of the vectorization (i.e., at best from ∼112 to ∼109 seconds on KNL-A). Nevertheless, it is expected to further elevate the performance when we employ larger problem sizes, through which we can extract buckets of more independent/conflict-free subsets of data.

Simultaneous MultiThreading (SMT): As mentioned above, the hyperthreading impact on the performance is obvious in the cases of baseline, optimized, and `-xMIC-AVX512`. In the three cases, the code exhibits similar speedup behaviors, namely about 1.2x and 1.4x speedup, if we scale from 64 (1 thread/core) to 128 (2 threads/core), and then to 256 (4 threads/core). This shows that our code optimizations and tuning purposely preserve the monotonic scaling aspects of the original code. Nevertheless, when we optimize for data-level parallelism, the overall performance is improved but not the SMT scalability. In other words, the performance of the data-level parallelism is improved when we execute two threads per core (middle set of bars), and drops with four threads per core (right set). The improvement with the two threads is barely 1.1x speedup. Indeed, four threads/core achieve 1.1x speedup compare to 1 thread/core, but that achievement fades, and in some cases, it gets even worst in comparison to 2 threads/core. Even though the SMT performance outcomes are unpredictable, typical latency-bound kernels would perform very well with SMT. Bandwidth-bound codes, on the other hand, are faster with a single thread per core. In conclusion, there is a significant trade-off between: 1) maximizing a single thread performance, when exclusively reserving the core's resources for one thread per core, and 2) maximizing a single core throughput, when multiple thread contexts are executed per core. In our case with data-level parallelism, the vector code explicitly exploits the two VPUs of KNL, when executing one thread context per VPU (two threads per core). It somehow attempts to maximize both a single thread performance with one thread per VPU, as well as a single core throughput with two threads per core. On the other hand, with 4 threads per core, the thread scheduler interleaves between every 2 threads for each VPU [41], and thereby at least 2 threads will be executed and 2 are going to be queued in the thread pool waiting for VPUs to be allocated for concurrent execution. Therefore, launching 4 threads per core is better than a single thread per core, which maximizes the work concurrency.

5.2 Performance of the Gradient Routine

Our baseline is the work of [16], which is mainly focused upon optimizing the flux kernel. The gradient kernel is a different matter, as Figure 7 shows. Our optimized code achieves nearly 1.8x speedup compared to the baseline. However, an unexpected outcome is that the handwritten vectorization of the gradient kernel shows a significant drop in the performance (i.e., in excess of 50%) compared to the scalar code. To understand this phenomena, a vectorization analysis of the flux and gradient vector codes is carried out via the Vectorization Advisor tool of Intel [48]. The next subsection illustrates the vectorization efficiency analysis of the two kernels. On the other side, our further optimized code via kernel simplifications achieves nearly 2x additional speedup compare to the baseline, and it results in almost 1.2x speedup compared to the initial optimized version.

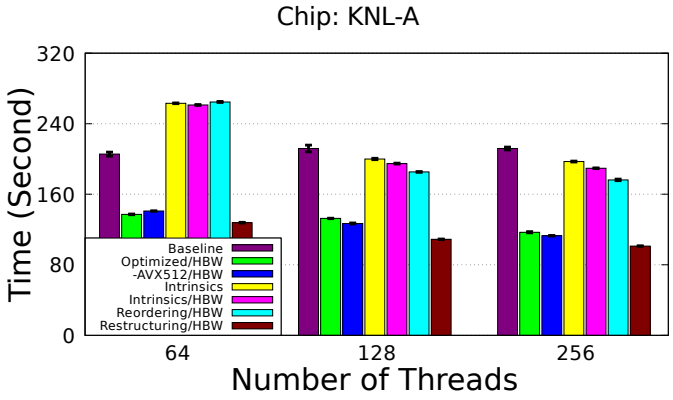


Fig. 7. Gradient performance optimizations

The performance behaviors of SMT on the gradient kernel are fluctuating. Essentially, SMT does not exhibit considerable performance boosts or scalability aspects in most of the cases. In other words, the original baseline code runtime drops with more threads/core. However, our initially optimized version gains roughly 1.2x speedup with more threads/core compare to 1 thread/core, and similar speedup is noticed with `-xMIC-AVX512` as well as our further optimized version. Nonetheless, the data-level parallelism, in particular, strong thread scales monotonically with hyperthreading (on average 1.5x speedup with more than one threads/core); even though it does not boost the runtime performance compare to the non-vectorized version, but rather results in performance degradation. In addition, similar justifications and analysis of the flux kernel can be applied here, except that the gradient kernel is more of bandwidth-bound than latency-bound. It, thereby, performs memory references more than the arithmetic, and hence, single thread/core would perform better.

5.3 Vectorization Efficiency of the Edge-based Loop

We consider the best case scenario, where, at every loop iteration, the loaded vector data is completely independent

and conflict-free. We use Intel vectorization analysis (i.e., Intel Advisor [48]) metric to thoroughly inspect the vectorization performance of both kernels: flux and gradient. The metric is based upon counting the number of arithmetic and memory reference instructions, and thereby calculating the latency and bandwidth costs corresponding to every instruction. However, for the purpose of having vigorous analysis, we count the read and write instructions manually, and we rely on the Advisor to compute the latency and bandwidth of every instruction. We thereafter confirm our cost calculations with the final numbers reported by the Advisor for correctness. Therefore, accumulating the cost of scalar/vector instructions builds a representative analytical performance metric for the vectorization efficiency. In particular, these costs are relative to the instructions latency and bandwidth associated with specific operations, and the total number of iterations performed by the code. On the other hand, the reported speedup is relative to the cost of the scalar code as a baseline.

The flux routine features on average 6 load, 68 gather, and 8 scatter instructions, in addition to a copious amount of arithmetic instructions. On the other hand, the gradient routine performs nearly 2 loads, 70 gathers, and 24 scatter instructions, in addition to a limited number of the arithmetic instructions. The vector code cost of the flux kernel is then roughly 414, whereas the scalar code cost is about 1426. Further, the gradient kernel vector code cost is about 324, whereas the scalar code cost is roughly 346. Thus, the anticipated potential speedup out of explicitly vectorizing the flux routine is nearly 3.21 and about 0.99 for the gradient routine, required by the latency/bandwidth of the vectorized code as opposed to the scalar code. Illustratively, the 0.99 speedup means rather a performance degradation, which what the aforementioned experiments and results of the gradient kernel are confirming. However, our code performs conflict detection instructions at every loop iteration, by which extra scatter/gather/load instructions are additionally executed to resolve the conflicts. For instance, in the worst case scenario the gradient code performs an extra 8 scatter and 8 gather instructions for each edge's endpoint (32 in total)), whereas the flux code executes additional 8 scatter/gather instructions (in total of 16 instructions for the two endpoints). Therefore, at most, the flux kernel gains nearly 60% of the expected potential speedup (1.8x) with vectorization and edge reordering. This requires that we replace the division/square root operations with reciprocal instructions. We expect that the overall speedup gain would grow as we increase the problem size, by which we can extract larger subsets of independent SIMD lanes. Nevertheless, the gradient kernel gains no speedup out of the explicit data-level parallelism, and rather results in performance degradation. Furthermore, the kernel features minimal arithmetic calculations compare to the memory reference instructions. Also, as a consequence of the indirect addressing overhead, the gradient kernel uses a significant number of registers, which causes *register spilling* whenever the register allocator exhausts the available registers, and therefore must swap their values with the memory [49], [50].

5.4 Performance of Explicit Vectorization

Figure 8 shows the vectorization performance of the pseudo timestep kernels. We exhibit two different approaches to vectorize the pseudo timestep kernel: 1) using the compiler auto-vectorization via `#pragma simd`, and 2) writing explicit AVX-512 handwritten intrinsics, similar to the flux kernel. As manifested from the figure, the performance of the compiler's vector code is nearly the same as employing handwritten intrinsics. This effectively means that the Intel compiler rather generates optimal vector code for the kernel. Hence, writing an intrinsics code for a routine, where the compiler successfully manages to vectorize it, is very often unnecessary [51].

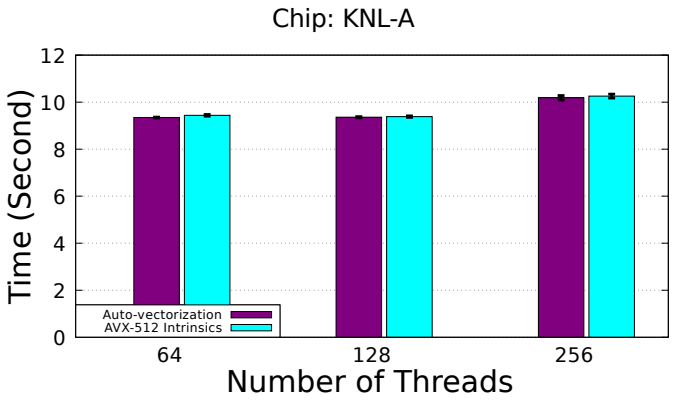


Fig. 8. Vectorization of the pseudo timestep kernel

5.5 Strong Thread Scalability Study

The culmination of our work is the significant reduction on the percentage of the execution time contributions of the edge-based loop kernels, from almost 74% down to 5%. The PETSc routines become the predominant contributors by nearly 94% (from 26%). In terms of kernel-wise scalability, Figure 9 shows strong thread scaling of the entire PETSc-FUN3D from a single thread to the full number of the available KNL threads. As the thread count increases, nearly all of the computational kernels execution times drop significantly, except for the PETSc routines, which are running sequentially. Figure 10 presents the speedup gains of the optimized kernels relative to their single-thread executions.

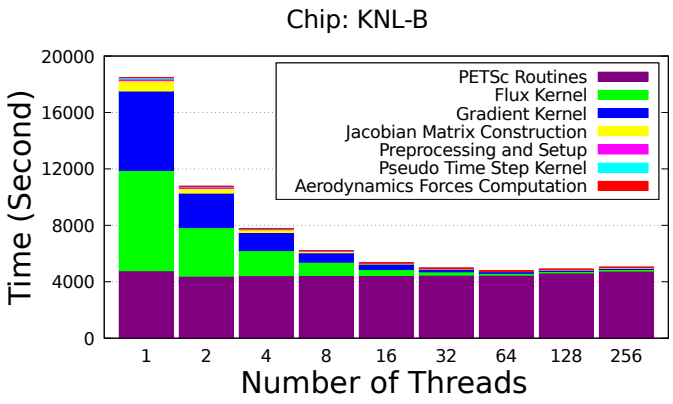


Fig. 9. Strong thread scaling of the entire code

Figure 11 shows the measurements of our speedup gain from concurrency compared with the available speedup using Amdahl's law [52]. If f is the fraction of the code that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2826533, IEEE Transactions on Parallel and Distributed Systems

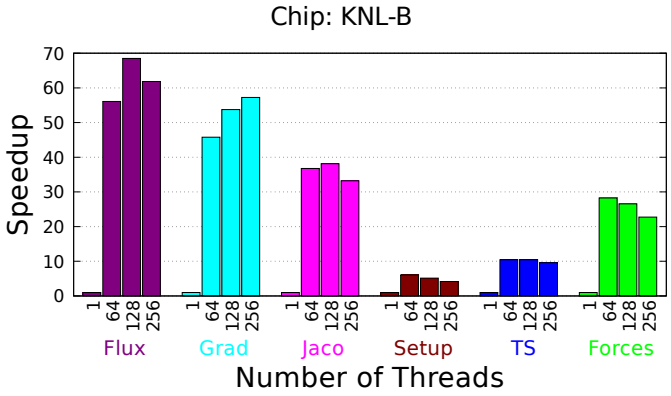


Fig. 10. Optimized routines performance speedup

is parallelizable, here 74%, and c is the number of threads, then the potential speedup can be calculated by Equation 1.

$$Speedup = \frac{1}{(1-f) + \frac{f}{c}} \quad (1)$$

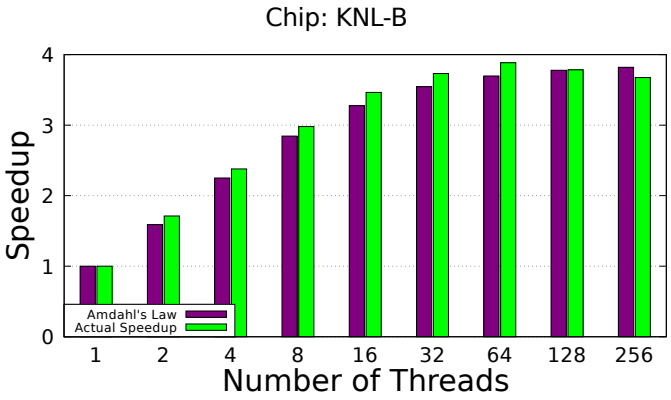


Fig. 11. Speedup available/achieved from concurrency

The flux routine is the main kernel that benefits from our algorithmic/architecture-specific optimizations and tuning. To further explore this, Figure 12 shows the near linear strong thread scaling of the flux kernel as we increase the number of KNL thread contexts. However, when more than one thread per core is executed, the performance declines. Figure 13 presents the gradient kernel strong thread scalability, which exhibits nearly the same scalability behaviors as the flux routine. The blue numbers on top of the data points are the speedup relative to one thread, whereas the black numbers underneath the data points are the parallel efficiency.

5.6 Memory Bandwidth and FLOPS Performance

Table 2 presents the floating point performance (GFlop/s) of the flux and gradient kernels on KNL, whose peak flop/s rating is roughly 3 TFlop/s. At best, this unstructured code achieves on average 5% of the peak for the flux routine [37]. Table 3 shows the memory bandwidth performance (GB/s) of the flux and gradient kernels on KNL. The performance of the STREAM read benchmark [53], [54] on KNL is roughly

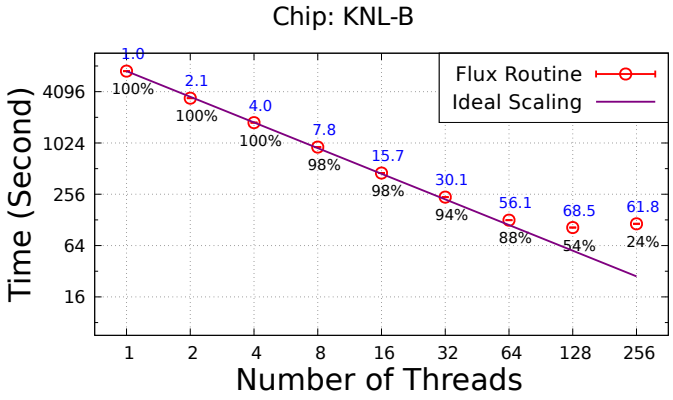


Fig. 12. Strong thread scaling of the flux kernel

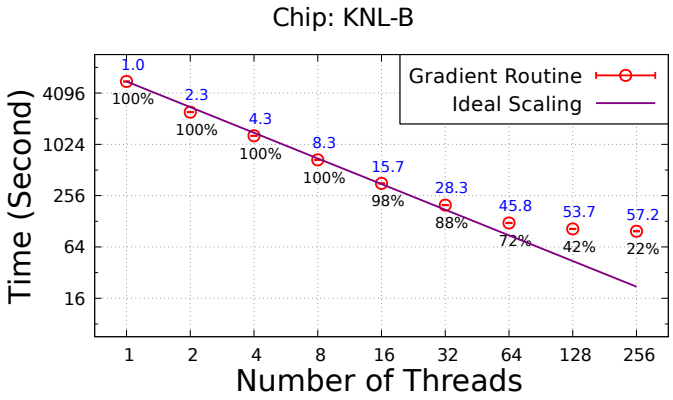


Fig. 13. Strong thread scaling of the gradient kernel

77 GB/s for the DRAM and 314 GB/s for the MCDRAM, whereas the STREAM write benchmark is about 36 GB/s for the DRAM and 171 GB/s for the MCDRAM [46]. At best, we achieve roughly 25% out of the STREAM read/write performance for the DRAM, and about 17% for the MCDRAM. The MCDRAM write bandwidth for the flux and the DRAM write bandwidth for the gradient is almost negligible. This is because the residual vector to which the flux kernel writes is stored in the DRAM[5], and the gradient vector to which the gradient kernel writes is stored in the MCDRAM.

TABLE 2
Floating Pointing Performance on KNL-C

	Flux Kernel			Gradient Kernel		
Threads	72	144	288	72	144	288
GFlop/s	117	144	123	21	24	25

5.7 Performance on Multi/Many-core Hardware

We compare the performance of PETSc-FUN3D across a variety of x86 architectures. Skylake is the only multi-core architecture that supports a subset of AVX-512 instruction set of KNL to some other AVX-512 extensions that are only featured in Skylake [55], [56], [57]. The subset of AVX-512

5. The residual vector is not stored in MCDRAM because it is allocated via PETSc routine and we do not control the PETSc allocation mechanisms.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2018.2826533, IEEE Transactions on Parallel and Distributed Systems

TABLE 3
Memory Bandwidth Performance on KNL-C

		Flux Kernel			Gradient Kernel		
Threads		72	144	288	72	144	288
DDR	Read: GB/s	14	17	17	8	10	12
	Write: GB/s	7	9	9	0.2	0.4	1
HBW	Read: GB/s	40	51	45	43	53	54
	Write: GB/s	0.1	0.1	0.1	18	22	25

instructions that are supported by KNL but not Skylake includes: 1) AVX-512 exponential and reciprocal, and 2) AVX-512 prefetching [55], [56], [58]. Hence, on Skylake, we run PETSc-FUN3D code with AVX-512 intrinsics after replacing any instructions that are related to these unsupported ones. However, on Haswell and Broadwell, we run PETSc-FUN3D without intrinsics, since the AVX2 does not provide conflict detection instructions, which, as explained earlier, are vital to vectorization. To emphasize this, the experiments on Haswell and Broadwell use our optimized and highly performant code without the data-level parallelism (i.e., the scalar version of the flux and other CD-dependent kernels), by which we build insightful comparisons across different architectures. In addition, some parts of the code are slightly modified to better-perform on a specific architecture, including Haswell and Broadwell. However, the modifications are not significant, and they are mostly related to the data structures formulations based upon the memory subsystem of a specific architecture.

As outlined earlier, on multi-core x86 hardware, our thread number selection is based upon: 1) one socket, 2) dual-socket, and 3) dual-socket with hyperthreading (2 threads per core). In other words, we first fill a socket, then fill a node, and finally exploit the complete number of the available hardware threads.

Figure 14 presents the flux kernel performance across different x86 architectures, whereas Figure 15 shows the gradient kernel performance. The enhanced performance of KNL outperforms Haswell and Broadwell architectures.

The best KNL performance of both kernels (i.e., 144 threads) accomplishes almost 1.7x speedup compare to the best multi-core performance (i.e., 72 threads of Haswell). Also, since the flux kernel, specifically, is a very compute-intensive kernel, our thread-level optimizations exploit as many thread contexts as the architectures can provide.

The best KNL performance (i.e., 144 threads) outperforms the best Skylake performance (i.e., 112 threads) with nearly 1.1x speedup. The gradient kernel on KNL, on the other hand, maintains its speedup compare to Skylake (about 1.4x). In addition, the KNL socket is rated at 215 Watts, whereas the comparable Skylake configuration of two sockets is rated at 330 Watts.

Figure 16 shows the entire execution time of PETSc-FUN3D across different architectures. The well-known downside of KNL is that the single thread performance is poor in contrast to a Xeon CPU. It functions at design for parallel compute-intensive kernels. Due to the sequential portion of PETSc-FUN3D, the overall performance falls behind other x86 architectures. In particular, comparing the flux performance, for instance, in Figure 14 to the entire PETSc-FUN3D performance in Figure 16 on the same hardware, shows the performance pitfalls of a single thread on KNL. Running only the flux kernel on KNL, which is well

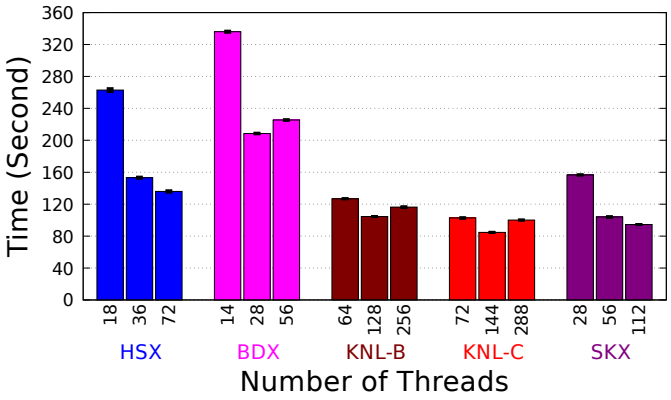


Fig. 14. Flux performance on different hardware

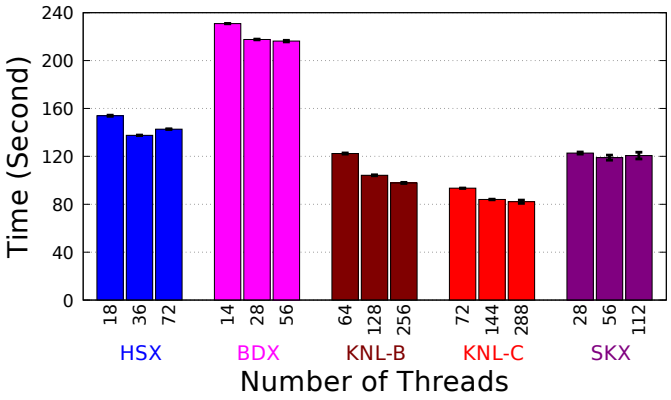


Fig. 15. Gradient performance on different hardware

parallelized and optimized to specifically favor the architecture, significantly outperforms other hardware, even top-of-the-line x86 architecture, Skylake. However, involving single threaded, less architecture-specific optimized kernels (i.e., PETSc routines) spikes the overall execution time on an architecture that does not provide good single thread performance, like KNL.

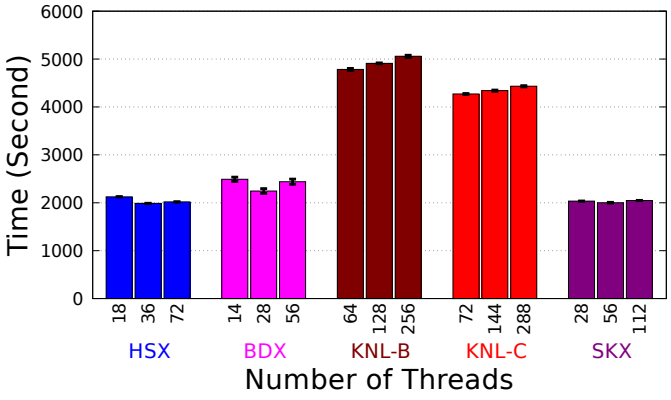


Fig. 16. Entire code performance on different hardware

Figures 14, 15, and 16 show that our shared-memory optimizations and tuning share value across the span of multi and many-core architectures. On Skylake, which is the

cutting-edge Intel architecture that embraces many closely related features of KNL, including, but not limited to, the support of AVX-512 ISA, we show that we maintain roughly similar speedup with less power consumption. In particular, all of our optimizations payoff on Skylake and well exploit the hardware. The performance is close to KNL and would evidently even outperform KNL with more added compute cores.

6 Related Work

The field of performance engineering and optimizations has natural follow-ons in a number of different directions; spanning wide-range of scientific applications. Each application addresses algorithmic-specific challenges related to the performance merits and detriments of a given hardware, which essentially has developed a thorough body of knowledge relevant to the architecture. Therefore, different optimization goals and constrains are undertaken based upon certain objectives, by which porting scientific kernel codes into emerging computing ecosystems is plausible.

6.1 Unstructured Aerodynamics Computations

The PETSc-FUN3D application code has undergone several cycles of development. It has been a representative testbed of unstructured computations stressing different HPC hardware generations. The shared-memory optimization efforts in [11], [12], [37] are arguably very conservative, focused upon transforming the original vector processor optimized code into cache-based architectures. The key optimization techniques, which have been evolving over years as the architecture landscape keeps changing, include field interlacing, structural blocking, and reordering. Later in [13], [14], [16], crucial efforts of evaluating the thread-level performance potentials of PETSc-FUN3D on wide spectrum of architectures are presented. On the other hand, SU2 code of Stanford [59] and OP2 code of Oxford [60] are considered to be the state-of-the-practice unstructured CFD research codes, which both have recently been ported into many emerging HPC architectures [61], [62].

To the best of our knowledge, none of the above mentioned works had dedicated ports to the KNL architecture in particular. Indeed, since KNL is an x86 architecture, the code that works on an x86 CPU will work on KNL. To this end, this paper initially considers evaluating all of the previously implemented and explored node-level optimizations, which are applicable to PETSc-FUN3D (e.g., using METIS partitioner, and data layout and edges workload reordering), on KNL architecture. We consider these optimizations as our baseline code. Then, we fine-tune them for KNL architecture, such that we squeeze the best possible performance on the architecture, before developing our particular algorithmic and architecture-specific shared-memory optimizations. Our implementation is based on a hierarchical, multi-level workload distribution and balancing that takes into account the underlying hardware, and equidistributes and orders the edges workload accordingly. Hence, our work maps the workload to first the vector registers, then caches (L1 and L2), and finally MCDRAM/DRAM. In addition, on Skylake, we follow nearly the same strategies of ordering and distribution, however, the mapping considers L3 cache as an exclusive Last Level Cache (LLC). Our work mainly targets optimizing the workload for throughput and latency to favor KNL architecture, which distinguishes this work from other CFD research, thus far. METIS is tuned based on the underlying KNL architecture, taking into consideration minimizing the edge replication (i.e., edge-cut) overhead that is naturally occurred when we have fine-grained partitions, all of which share the same hardware resources. Similarly, our code utilizes the AVX-512 capabilities by leveraging the data-level parallelism in the context of indirect addressing through exploiting the conflict detection instructions, and a novel data dependency conflicts resolution approach based on partial edge coloring and swizzling.

6.2 State-of-the-art Shared-memory Optimizations

Node-level optimizations have been a subject-of-interest in many diverse applications that target wide-range of HPC architectures. For example, seismic [63], stencil [64], [65], electromagnetic [66], molecular dynamics [67], Fast Multipole Methods [68], tensors [39], deep learning [69], [70], databases [49], [71], [72], big data [73], systems and graph engines [74], and many more.

Systems and Graph Engines: Maass *et al.* [74] processes a trillion edges on a compute node of multiple KNCs. Techniques like tiling/blocking, out-of-core graph processing, and asynchronous CPU+KNC hybrid execution model are employed to optimized the performance of the graph operations on a single compute node. Similarly, MapReduce framework is optimized for KNC architecture in [73], in which the thread-level parallelism is leveraged. Vectorization through explicit SIMDization is explored in many database primitive operations including hashing [71], [72], as well as sequential scans, aggregation, index operations, and joins [49].

Performance Analytical Models: The work of [1], [75] optimizes for cache line awareness, where an analytical performance model is built to tune the cache line transfers of different architectures, including KNC and Sandy Bridge. Their model is recently extended to explore KNL [46], which includes constructing several performance models for certain combinations of KNL clustering and memory modes. Furthermore, the work of [76] performs several experimentations on KNL with different applications, through which Roofline performance models are drawn for different configurations of KNL. The performance of the hybrid memory system of KNL is investigated in [77], which provides an analytic model for performance tuning. A Roofline model specifically for benchmarking the performance of a well-optimized OpenMP implementation of the tall-skinny matrix multiplication kernel for a molecular dynamics application code is proposed in [67], which essentially leverages the thread-level parallelism on KNL.

Thread and Memory Performance Optimizations: Thread-level workload balancing with minimal synchronization overheads is proposed in [39], while providing a sophisticated implementation of a memory-aware allocation strategy to maintain the working sets on both MCDRAM and DRAM based on the frequency of the memory accesses. In addition, KNL-specific optimizations and tuning are proposed in [63] for seismic computations, with detailed comparisons between KNC and Haswell. In [78], the performance of the hybrid MPI+OpenMP programming paradigm is provided on Theta supercomputer based on KNL compute nodes. The many-core scalability of the edge-based graph coloring algorithm performance is provided

in [79], which targets both GPU as well as KNC. Data layout transformation and memory access pattern are key optimization mechanisms for the single node performance in many research areas, including compiler-specific auto-optimization (e.g., ISPC [22]), C-like extensions to assist vectorization [80], a runtime scheduling framework for data distribution and load balancing between the cores and the SIMD units [81], as well as a just-in-time compilation framework to build an architecture-specific, SIMDized code for small matrix multiplications, and low-level data structure transformations [68].

Most of the aforementioned optimizations target structure/regular memory access patterns, in which thread-level and memory optimizations favor KNL-type architectures. In addition, data-level parallelism in their context is a straightforward approach through either compiler's auto-vectorization or handwritten intrinsics, once the data dependency conflicts within a SIMD lane is adjusted. Some of this work is inherited and customized to our application code. For instance, SoA of [68], AoSoA of [22], low-level, MCDRAM-aware allocator of [39], data dependency conflicts migration of [71], Hilbert-based recursive tiling/blocking of [74], cache line aware optimization of [1], [46], [75], and partial coloring of [79]. In our work, we deal with irregular memory access patterns through optimizing for the cache line awareness based upon minimizing memory reference arithmetic and pointer chasing, as well as localizing a large bulk of computations inside a compute core. We thereby confine the workload within a core to SMT threads sharing the same resources, and further restrict the locality to the compute tiles of KNL to better-utilize the L2 cache of the tiles.

7 Conclusion and Ongoing Work

Scalability of scientific workloads on next-generation computing systems is critically dependent upon the performance of a shared-memory multi and many-core compute node [82], to which this paper adds a chapter for the important problem class of unstructured meshes.

In this work, motivated by KNL architecture, we demonstrate several shared-memory optimizations, including threading, vectorization, data structure transformation, memory management, and arithmetic instruction reformulations without compromising the original numerics. We extract both 1) the thread-level parallelism through careful workload distributions and load balancing, and 2) the data-level parallelism via utilizing the capabilities of AVX-512 instruction set, as well as employing further fine-grained data partitioning that assists constructing conflict-free subsets of data. Our implemented workload distributions takes into consideration the mapping across the hardware memory hierarchy, from the DRAM and MCDRAM, to the L2 and L1 caches, and finally down to the floating point registers. Ultimately, we achieve roughly 2.9x speedup in the performance of the flux kernel relative to the baseline code on KNL, and 1.4x speedup relative to the performance of a Haswell CPU. We also exhibit almost linear scalability up to the full core count of KNL (64 cores), and continued scalability with SMT. Compared with 112 threads of Skylake, we exhibit nearly comparable performance. Though illustrated on an incompressible Euler test case, the performance enhancements should extend to richer models with compressibility and viscosity, and more advanced workloads like adjoint-based optimization. Richer physical models increase the flop intensity and the block size of the unrolled loops, for the same size working sets. Post-forward analyses such as engineering optimization, place our forward solves inside of another loop.

In the future, in conjunction with the author we plan to improve and optimize essential library routines in the PETSc toolkit. Although, these routines are limited in terms of thread-level parallelism, their numerical algorithms can be reformulated and restructured to favor certain aspects of many-core hardware, see, e.g., [83] for more details.

Acknowledgments

Support in the form of computing resources was provided by KAUST Extreme Computing Research Center, KAUST Supercomputing Laboratory, KAUST Information Technology Research Division, and Intel Parallel Computing Centers.

References

[1] S. Ramos and T. Hoefler, "Cache line aware algorithm design for cache-coherent architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2824–2837, 2016.

[2] M. Zandifar, M. AbdulJabbar, A. Majidi, D. Keyes, N. M. Amato, and L. Rauchwerger, "Composing algorithmic skeletons to express high-performance scientific applications," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. ACM, 2015, pp. 415–424.

[3] W. K. Anderson and D. L. Bonhaus, "An implicit upwind algorithm for computing turbulent flows on unstructured grids," *Computers and Fluids*, vol. 23, no. 1, pp. 1–21, 1994.

[4] W. K. Anderson, R. D. Rausch, and D. L. Bonhaus, "Implicit/-Multigrid algorithms for incompressible turbulent flows on unstructured grids," *Journal of Computational Physics*, vol. 128, no. 2, pp. 391–408, 1996.

[5] A. C. Duffy, D. P. Hammond, and E. J. Nielsen, *Production Level CFD Code Acceleration for Hybrid Many-Core Architectures*, NASA/tm-2012-217770 ed., NASA Langley Research Center, October 2012, https://ntrs.nasa.gov/search.jsp?R=20120014581 [Accessed: March 20, 2018].

[6] M. Zubair, E. Nielsen, J. Luitjens, and D. Hammond, "An optimized multicolor point-implicit solver for unstructured grid applications on graphics processing units," in *Proceedings of the 6th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA^3 '16. IEEE, 2016, pp. 18–25.

[7] W. K. Anderson, P. Balakumar, K. Bibb, B. Biedron, J. Carlson, M. Carpenter, J. Derlaga, P. Gnoffo, D. Hammond, W. Jones, B. Kleb, B. R. Lee, E. Nielsen, H. Nashikawa, M. Park, C. Rumsey, J. Thomas, V. Vatsa, and J. White, "FUN3D Web page," 2017, [Accessed: March 20, 2018]. [Online]. Available: http://fun3d.larc.nasa.gov/

[8] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[9] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang, *PETSc Users Manual*, Argonne National Laboratory, April 2016, http://www.mcs.anl.gov/petsc/documentation/index.html [Accessed: March 20, 2018].

[10] ——, "PETSc Web page," 2016, [Accessed: March 20, 2018]. [Online]. Available: http://www.mcs.anl.gov/petsc

[11] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "High-performance parallel implicit CFD," *Parallel Computing*, vol. 27, no. 4, pp. 337–362, 2001, Parallel Computing in Aerospace.

[12] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Achieving high sustained performance in an unstructured mesh CFD application," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, ser. SC '99. ACM, 1999.

[13] D. Kaushik, D. Keyes, S. Balay, and B. Smith, "Hybrid programming model for implicit PDE simulations on multicore architectures," in *OpenMP in the Petascale Era: 7th International Workshop on OpenMP (IWOMP 2011)*. Springer, June 2011, pp. 12–21.

[14] D. Mudigere, S. Sridharan, A. Deshpande, J. Park, A. Heinecke, M. Smelyanskiy, B. Kaul, P. Dubey, D. Kaushik, and D. Keyes, "Exploring shared-memory optimizations for an unstructured mesh CFD application on modern parallel systems," in *29th International Parallel and Distributed Processing Symposium (IPDPS 2015)*. IEEE, May 2015, pp. 723–732.

[15] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, 1st ed. San Francisco, California: Morgan Kaufmann, 2013.

[16] M. A. Al Farhan, D. K. Kaushik, and D. E. Keyes, "Unstructured computational aerodynamics on many integrated core architecture," *Parallel Computing*, vol. 59, pp. 97–118, 2016, Theory and Practice of Irregular Applications.

[17] D. A. Knoll and D. E. Keyes, "Jacobian-free Newton-Krylov methods: A survey of approaches and applications," *Journal of Computational Physics*, vol. 193, no. 2, pp. 357–397, 2004.

[18] W. Gropp, D. Keyes, L. C. Mcinnes, and M. D. Tidriri, "Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD," *International Journal of High Performance Computing Applications*, vol. 14, no. 2, pp. 102–136, 2000.

[19] T. S. Coffey, C. T. Kelley, and D. E. Keyes, "Pseudotransient continuation and differential-algebraic equations," *SIAM Journal on Scientific Computing*, vol. 25, no. 2, pp. 553–569, 2003.

[20] C. T. Kelley and D. E. Keyes, "Convergence analysis of pseudotransient continuation," *SIAM Journal on Numerical Analysis*, vol. 35, no. 2, pp. 508–523, 1998.

[21] A. Mathuriya, Y. Luo, A. Benali, L. Shulenburger, and J. Kim, "Optimization and parallelization of B-Spline based orbital evaluations in QMC on multi/many-core shared memory processors," in *31st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017)*, May 2017, pp. 213–223.

[22] M. Pharr and W. R. Mark, "ISPC: A SPMD compiler for high-performance CPU programming," in *Innovative Parallel Computing (InPar)*. IEEE, May 2012, pp. 1–13.

[23] R. Yokota and M. AbdulJabbar, "N-Body methods," in *High Performance Parallelism Pearls*, 1st ed., J. Reinders and J. Jeffers, Eds. Morgan Kaufmann, 2015, pp. 175–183.

[24] A. Valdimirov, *Optimization Techniques for the Intel MIC Architecture. Part 2 of 3: Strip-Mining for Vectorization*, Colfax International, June 2015, https://colfaxresearch.com/category/papers/case-studies/ [Accessed: March 20, 2018].

[25] R. Asai, *Optimization of Hamerly's K-Means Clustering Algorithm: CFXKMeans Library*, Colfax International, July 2017, https://colfaxresearch.com/cfxkmeans/ [Accessed: March 20, 2018].

[26] G. Karypis, "METIS Web page," 2013, [Accessed: March 20, 2018]. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

[27] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[28] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *24th ACM National Conference (ACM 1969)*. ACM, 1969, pp. 157–172.

[29] R. Asai and A. Valdimirov, *Optimization Techniques for the Intel MIC Architecture. Part 1 of 3: Multi-Threading and Parallel Reduction*, Colfax International, May 2015, https://colfaxresearch.com/category/papers/case-studies/ [Accessed: March 20, 2018].

[30] C. Cantalupo, V. Venkatesan, J. R. Hammond, K. Czurylo, and S. Hammond, *User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies*, Intel, March 2015, http://memkind.github.io/memkind/ [Accessed: March 20, 2018].

[31] J. Evans, *A Scalable Concurrent malloc(3) Implementation for FreeBSD*, FreeBSD, April 2006, http://jemalloc.net/ [Accessed: March 20, 2018].

[32] A. Valdimirov, *Optimization Techniques for the Intel MIC Architecture. Part 3 of 3: False Sharing and Padding*, Colfax International, August 2015, https://colfaxresearch.com/category/papers/case-studies/ [Accessed: March 20, 2018].

[33] B. Zhang, *Guide to Automatic Vectorization with Intel AVX-512 Instructions in Knights Landing Processors*, Colfax International, May 2016, https://colfaxresearch.com/knl-avx512/ [Accessed: March 20, 2018].

[34] R. Rahman, *Intel® Xeon® Phi™ Coprocessor Vector Microarchitecture*, Intel, May 2013, https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture [Accessed: March 20, 2018].

[35] N. Bansal and S. Garg, "Algorithmic discrepancy beyond partial coloring," in *49th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2017)*. ACM, 2017, pp. 914–926.

[36] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2017)*. ACM, 2017, pp. 235–248.

[37] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Towards realistic performance bounds for implicit CFD codes," in *11th International Parallel Computational Fluid Dynamics Conference (Parallel CFD 1999)*. Elsevier, pp. 233–240.

[38] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015)*. ACM, Nov 2015, pp. 73:1–73:12.

[39] S. Smith, J. Park, and G. Karypis, "Sparse tensor factorization on many-core processors with high-bandwidth memory," in *31st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017)*. IEEE, May 2017, pp. 1058–1067.

[40] Intel, *Intel® Xeon® Phi™ Processor Performance Monitoring Reference Manual*, Intel, March 2017, https://software.intel.com/en-us/articles/intel-xeon-phi-x200-family-processor-performance-monitoring-reference-manual [Accessed: March 20, 2018].

[41] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming (Knights Landing Edition)*, 2nd ed. Boston, Massachusetts: Morgan Kaufmann, 2016.

[42] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights Landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[43] R. Asai, *MCDRAM as High-Bandwidth Memory (HBM) in Knights Landing Processors: Developer's Guide*, Colfax International, May 2016, https://colfaxresearch.com/knl-mcdram/ [Accessed: March 20, 2018].

[44] R. Asai and A. Vladimirov, *Clustering Modes in Knights Landing Processors: Developer's Guide*, Colfax International, May 2016, https://colfaxresearch.com/knl-numa/ [Accessed: March 20, 2018].

[45] V. Codreanu, J. Rodriquez, and O. W. Saastad, *Best Practice Guide - Knights Landing*, Partnership for Advanced Computing in Europe (PRACE), January 2017, http://www.prace-ri.eu/best-practice-guide-knights-landing-january-2017/ [Accessed: March 20, 2018].

[46] S. Ramos and T. Hoefler, "Capability models for manycore memory systems: A case-study with Xeon Phi KNL," in *31st International Parallel and Distributed Processing Symposium (IPDPS 2017)*. IEEE, May 2017, pp. 297–306.

[47] D. Mulnix, "Intel® xeon® processor scalable family technical overview," 2017, [Accessed: March 20, 2018]. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview

[48] Intel, "Intel advisor 2017," 2017, [Accessed: March 20, 2018]. [Online]. Available: https://software.intel.com/en-us/intel-advisor-xe

[49] J. Zhou and K. A. Ross, "Implementing database operations using simd instructions," in *International Conference on Management of Data (SIGMOD 2002)*. ACM, June 2002, pp. 145–156.

[50] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Register optimizations for stencils on GPUs," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. ACM, 2018, pp. 168–182.

[51] M. Abduljabbar, M. Al Farhan, R. Yokota, and D. Keyes, "Performance evaluation of computation and communication kernels of the fast multipole method on intel manycore architecture," in *23rd International European Conference on Parallel and Distributed Computing (Euro-Par 2017), LNCS vol. 10417*. Springer, August 2017, pp. 553–564.

[52] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Spring Joint Computer Conference (AFIPS 1967 (Spring))*. ACM, April 1967, pp. 483–485.

[53] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, December 1995.

[54] ——, *STREAM: Sustainable Memory Bandwidth in High Performance Computers*, University of Virginia, December 1991, url = http://www.cs.virginia.edu/stream/ [Accessed: March 20, 2018].

[55] A. Eltablawy and A. Valdimirov, *Capabilities of Intel® AVX-512 in Intel® Xeon® Scalable Processors (Skylake)*, Colfax International, September 2017, https://colfaxresearch.com/skl-avx512/ [Accessed: March 20, 2018].

[56] S. Ragate and R. Asai, *Optimization of Real-Time Object Detection on Intel® Xeon® Scalable Processors*, Colfax International, November 2017, https://colfaxresearch.com/yolo-optimization/ [Accessed: March 20, 2018].

[57] V. Kasliwal and A. Vladimirov, *A Performance-Based Comparison of C/C++ Compilers*, Colfax International, November 2017, https://colfaxresearch.com/compiler-comparison/ [Accessed: March 20, 2018].

[58] A. Vladimirov, *A Survey and Benchmarks of Intel Xeon Gold and Platinum Processors*, Colfax International, November 2017, https://colfaxresearch.com/xeon-2017/ [Accessed: March 20, 2018].

[59] T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, and J. J. Alonso, "Su2: an open-source suite for multiphysics simulation and design," *AIAA Journal*, vol. 54, no. 3, pp. 828–846, 2016.

[60] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–12.

[61] T. D. Economon, F. Palacios, J. J. Alonso, G. Bansal, D. Mudigere, A. Deshpande, A. Heinecke, and M. Smelyanskiy, "Towards high-performance optimizations of the unstructured open-source SU2 suite," in *AIAA SciTech Forum (AIAA Infotech @ Aerospace)*. American Institute of Aeronautics and Astronautics, January 2015, pp. 1–30.

[62] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. J. Kelly, and D. Radford, "Acceleration of a full-scale industrial CFD application with OP2," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1265–1278, May 2016.

[63] A. Heinecke, A. Breuer, M. Bader, and P. Dubey, "High order seismic simulations on the Intel Xeon Phi processor (Knights Landing)," in *31st International Conference on High Performance Computing (ISC 2016)*, ser. Lecture Notes in Computer Science, vol. 9697. Springer, 2016, pp. 343–362.

[64] T. M. Malas, G. Hager, H. Ltaief, and D. E. Keyes, "Multidimensional intratile parallelization for memory-starved stencil computations," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 3, pp. 12:1–12:32, 2017.

[65] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and K. D., "Multicore-optimized wavefront diamond blocking for optimizing stencil updates," *SIAM Journal on Scientific Computing*, vol. 37, no. 4, pp. C439–C464, 2015.

[66] T. M. Malas, J. Hornich, G. Hager, H. Ltaief, C. Pflaum, and D. E. Keyes, "Optimization of an electromagnetics code with multicore wavefront diamond blocking and multi-dimensional intra-tile parallelization," in *International Parallel and Distributed Processing Symposium (IPDPS 2016)*. IEEE, 2016, pp. 142–151.

[67] M. Jacquelin, W. D. Jong, and E. Bylaska, "Towards highly scalable Ab initio molecular dynamics (AIMD) simulations on the Intel Knights Landing manycore processor," in *31st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017)*, May 2017, pp. 234–243.

[68] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc, "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures," in *24th Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010, pp. 1–12.

[69] A. Zlateski and H. S. Seung, "Compile-time optimized and statically scheduled N-D convnet primitives for multi-core and many-core (xeon phi) cpus," in *21st ACM International Conference on Supercomputing (ICS 2017)*. ACM, June 2017, pp. 8:1–8:10.

[70] Y. You, A. Buluç, and J. Demmel, "Scaling deep learning on GPU and Knights Landing clusters," in *30th International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2017)*. ACM, 2017, pp. 9:1–9:12.

[71] P. Jiang and G. Agrawal, "Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation," in *21st ACM International Conference on Supercomputing (ICS 2017)*, ser. ICS '17. ACM, June 2017, pp. 24:1–24:11.

[72] X. Cheng, B. He, X. Du, and C. T. Lau, "A study of main-memory hash joins on many-core processor: A case with Intel Knights Landing architecture," in *26th International Conference on Information and Knowledge Management (CIKM 2017)*. ACM, 2017, pp. 657–666.

[73] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. S. M. Goh, "MrPhi: An optimized mapreduce framework on Intel Xeon Phi coprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 11, pp. 3066–3078, Nov 2015.

[74] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *12th European Conference on Computer Systems (EuroSys 2017)*. ACM, 2017, pp. 527–543.

[75] S. Ramos and T. Hoefler, "Modeling communication in cache-coherent SMP systems - a case-study with Xeon Phi," in *22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2013)*. ACM, Jan 2013, pp. 97–108.

[76] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J. L. Vay, and H. Vincenti, "Applying the roofline performance model to the Intel Xeon Phi Knights Landing processor," in *31st International Conference for High Performance Computing (ISC 2017), LNCS vol. 9945*. Springer, 2016, pp. 339–353.

[77] A. Li, W. Liu, M. R. B. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song, "Exploring and analyzing the real impact of modern on-package memory on HPC scientific kernels," in *30th International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2017)*. ACM, 2017, pp. 26:1–26:14.

[78] V. Mironov, Y. Alexeev, K. Keipert, M. D'mello, A. Moskovsky, and M. S. Gordon, "An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of intel® xeon® phi™ processor," in *30th International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2017)*. ACM, 2017, pp. 39:1–39:12.

[79] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures," in *30th International Parallel and Distributed Processing Symposium (IPDPS 2016)*. IEEE, 2016, pp. 892–901.

[80] R. Leißa, S. Hack, and I. Wald, "Extending a C-like language for portable SIMD programming," in *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012)*. ACM, February 2012, pp. 65–74.

[81] B. Ren, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, "Exploiting vector and multicore parallelism for recursive, data- and task-parallel programs," in *22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2017)*. ACM, February 2017, pp. 117–130.

[82] M. Abduljabbar, G. S. Markomanolis, H. Ibeid, R. Yokota, and D. Keyes, "Communication reducing algorithms for distributed hierarchical N-Body problems with boundary distributions," in *32nd International Conference for High Performance Computing (ISC 2017), LNCS vol. 10266*. Springer, June 2017, pp. 79–96.

[83] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM Journal on Scientific Computing (SISC)*, vol. 37, no. 2, pp. C169–C193, 2015.

Mohammed A. Al Farhan is a computer science PhD candidate working in the KAUST Extreme Computing Research Center under the supervision of Professor Keyes. His research interests are mainly in HPC architectures and applications in computational science and engineering. Al Farhan earned his BS and MS in computer science at KFU and KAUST, respectively, and has previously worked at the Saudi Electricity Company and Saudi Aramco as a Software Engineer. He is a member of SIAM, ACM, and IEEE.

David E. Keyes David Keyes directs the Extreme Computing Research Center at KAUST. He earned a BSE in Aerospace and Mechanical Sciences from Princeton in 1978 and a PhD in Applied Mathematics from Harvard in 1984. Keyes works at the interface between parallel computing and the numerical analysis of PDEs, with a focus on scalable implicit solvers. Newton-Krylov-Schwarz (NKS), Additive Schwarz Preconditioned Inexact Newton (ASPIN), and Algebraic Fast Multipole (AFM) are methods he helped name and is helping to popularize. Before joining KAUST as a founding dean in 2009, he led multi-institutional scalable solver software projects in the SciDAC and ASCI programs of the US DOE, ran university collaboration programs at LLNL's ISCR and NASA's ICASE, and taught at Columbia, Old Dominion, and Yale Universities. He is a Fellow of SIAM and AMS, and has been awarded the ACM Gordon Bell Prize, the IEEE Sidney Fernbach Award, and the SIAM Prize for Distinguished Service to the Profession.