Performance and Portability of a Linear Solver Across Emerging Architectures

Aaron C. Walden[1], Mohammad Zubair[2], and Eric J. Nielsen[1]

[1] NASA Langley Research Center, Hampton, Virginia, USA
[2] Old Dominion University, Norfolk, Virginia, USA

Abstract. A linear solver algorithm used by a large-scale unstructured-grid computational fluid dynamics application is examined for a broad range of familiar and emerging architectures. Efficient implementation of a linear solver is challenging on recent CPUs offering vector architectures. Vector loads and stores are essential to effectively utilize available memory bandwidth on CPUs, and maintaining performance across different CPUs can be difficult in the face of varying vector lengths offered by each. A similar challenge occurs on GPU architectures, where it is essential to have coalesced memory accesses to utilize memory bandwidth effectively. In this work, we demonstrate that restructuring a computation, and possibly data layout, with regard to architecture is essential to achieve optimal performance by establishing a performance benchmark for each target architecture in a low level language such as vector intrinsics or CUDA. In doing so, we demonstrate how a linear solver kernel can be mapped to Intel® Xeon™ and Xeon Phi™, Marvell® ThunderX2®, NEC® SX-Aurora™ TSUBASA Vector Engine, and NVIDIA® and AMD® GPUs. We further demonstrate that the required code restructuring can be achieved in higher level programming environments such as OpenACC, OCCA, and Intel® OneAPI™/SYCL, and that each generally results in optimal performance on the target architecture. Relative performance metrics for all implementations are shown, and subjective ratings for ease of implementation and optimization are suggested.

Keywords: Programming models · Performance portability · Emerging architecture · CFD · HPC · CUDA · OpenACC · OCCA · AVX-512 intrinsics · Neon intrinsics · arm · GPU · V100 · A100 · MI50 · Xeon Phi · SX-Aurora · ThunderX2.

1 Introduction

A diverse array of new hardware architectures continues to emerge across the High Performance Computing (HPC) landscape. The application developer is faced with the considerable challenge of providing near-optimal performance across these systems. This goal requires a detailed understanding of each target architecture and some means to accommodate specific data layouts and algorithm implementations that map appropriately. Ideally, this would be achieved in a unified code base that is easily maintained. In this vein, a number of general

portability approaches have recently been introduced that attempt to insulate the application developer from intricate details of the underlying hardware, yet still provide near-optimal performance on each. Unfortunately, some applications can require significant restructuring to achieve optimal performance on a particular system, which can be challenging to automate using general abstractions and run-time environments. In such cases, the developer may be required to address the needs of the underlying architecture at the application level.

The work reported here describes an ongoing effort to explore performance portability issues for the FUN3D computational fluid dynamics solver maintained at the NASA Langley Research Center [7]. FUN3D solves the Navier-Stokes (NS) equations, a system of highly nonlinear, tightly-coupled time-dependent partial differential equations. FUN3D is routinely used for a broad range of aerodynamics applications across the speed range, on both conventional x86-based systems [20], as well as GPU-based systems such as Summit at Oak Ridge National Laboratory (ORNL) [14]. FUN3D uses an implicit time-integration strategy with a node-based, finite-volume spatial discretization on general mixed-element unstructured grids. An approximate nearest-neighbor linearization of the discrete residual equations within each control volume gives rise to a large tightly-coupled system of block-sparse linear equations that must be solved at each time step. The block size is determined by the number of governing equations and may range from five to several dozen. To facilitate a practical investigation of the broad array of potential performance portability issues, the scope of the current effort is limited to optimization of the linear solver kernel used within FUN3D. The study is carried out across several familiar and emerging HPC architectures using a wide range of available programming models. While this study focuses on motifs related to linear algebra, parallel efforts aimed at unstructured-grid traversals with complex gather-scatter operations supporting flux and Jacobian construction are also ongoing but are beyond the current scope.

The block-sparse linear solver used here is memory-bound with a low arithmetic intensity. In such cases, it is critical to understand the increasingly complex memory hierarchies of today's advanced architectures and how memory bandwidth and potential reuse of computations can be effectively leveraged. For example, in the case of an NVIDIA® GPU, it is important to understand how to accommodate the application data layout and to restructure the solver algorithm to utilize the registers, shared memory, L1 and L2 caches, and DRAM effectively.

The dominant computation in the linear solver used here is a block-sparse matrix-vector product; for a broad range of applications encountered in practice, 5×5 blocks are common. The off-diagonal matrix coefficients are stored in a compressed sparse row (CSR) format [25], where two integer arrays capture the sparsity pattern of the nonzero blocks in the matrix. The nonzero blocks in a row are stored contiguously in memory, and the scalar entries within a block are stored in column-major order.

Efficient processing of such a matrix is challenging on recent CPUs offering vector architectures. Vector loads and stores are essential to effectively utilize available memory bandwidth on CPUs, and maintaining performance across different CPUs can be difficult in the face of varying vector lengths offered by each. For a sparse matrix with relatively large block sizes, it is reasonably straightforward to leverage vector loads and stores. For smaller block sizes, the computation calls for a restructuring based on the available vector length. For example, if the processor supports a vector length of 32 floating-point numbers, it is desirable to map a full dense block to a vector and organize the computation to work with this mapping. Alternatively, a CPU offering a vector length of 4 floating-point numbers may demand the mapping of a partial column of a dense block to a vector. For vector engines where the vector length may be large, say 256 elements, the data layout itself may require a substantial restructuring, leading to performance portability issues arising from different data layout requirements across architectures.

A similar challenge occurs on GPU architectures, where it is essential to have coalesced memory accesses to utilize memory bandwidth effectively. Modern GPUs support the Single Instruction Multiple Thread (SIMT) model, with a group of threads referred to as a warp (or wavefront). The dimension of this thread group can vary from one GPU to another, and the group must process consecutive memory locations to achieve coalesced memory accesses. This requires mapping the warp (or wavefront) to one or more blocks of a sparse matrix and restructuring the computation accordingly. In summary, restructuring the computation is essential, and in some cases, modifications to the underlying data layout may even be required.

The goal of this project is to assess the performance and portability of a wide variety of programming frameworks when applied to a production-scale CFD simulation code. The current work advances that goal in two ways. First, it attempts to establish, for both familiar and nascent HPC architectures, an optimal performance benchmark. In doing so, we demonstrate how a linear solver kernel can be mapped to Intel® Xeon™ and Xeon Phi™, Marvell® ThunderX2®, NEC® SX-Aurora™ TSUBASA Vector Engine, and NVIDIA® and AMD® GPUs. Second, this effort explores the ability of different programming frameworks to achieve the performance established by the benchmark for a subset of the target architectures.

2 Algorithm

For a spatial mesh containing n grid vertices, the implicit approach used within FUN3D requires frequent solutions of a large $n \times n$ linear system of equations of the form $\mathbf{A}\boldsymbol{\Delta}\mathbf{Q} = \mathbf{R}$, where $\mathbf{R}$ represents the vector of discrete residual equations, $\mathbf{A}$ is an $n \times n$ block-sparse matrix composed of dense $n_b \times n_b$ blocks, and $\boldsymbol{\Delta}\mathbf{Q}$ is the vector of unknowns required to advance the nonlinear solution $\mathbf{Q}^k$ at time-level k to $k+1$. The coefficient matrix $\mathbf{A}$ is based on a strictly nearest-

Algorithm 1 Multicolor Linear Solver

1: $\Delta\mathbf{Q} = 0$
2: **for** $i \leftarrow 1$ **to** n_{iter} **do**
3: **for** $c \leftarrow 1$ **to** n_c **do**
4: $\Delta\mathbf{r} \leftarrow \mathbf{R_c} - \mathbf{O_c}\Delta\mathbf{Q}$
5: $\Delta\mathbf{Q_c} \leftarrow \mathbf{D_c^{-1}}\Delta r$
6: **end for**
7: **end for**

neighbor stencil. To provide flexibility in the implementation, $\mathbf{A}$ is segregated into diagonal and off-diagonal components stored separately, namely

$$\mathbf{A} \equiv \mathbf{D} + \mathbf{O} \tag{1}$$

where $\mathbf{D}$ and $\mathbf{O}$ represent the diagonal and off-diagonal blocks of $\mathbf{A}$, respectively. The implementation in FUN3D uses 32-bit precision for $\mathbf{O}$ and $\mathbf{\Delta Q}$, while 64-bit precision is used for $\mathbf{D}$ and $\mathbf{R}$.

The block-sparse $n \times n$ matrix $\mathbf{O}$ contains nnz nonzero $n_b \times n_b$ blocks that are stored using a compressed sparse row (CSR) format [25]. Each of the n rows and columns containing $n_b \times n_b$ blocks are referred to as a *brow* and a *bcol*, respectively. Two integer arrays ia and ja are used to efficiently capture the sparsity pattern of the matrix. The array ia is a rank-1 array of size $n+1$ whose i-th entry indicates the leading nonzero block index in the i-th *brow* of $\mathbf{O}$. The array includes a fictitious $n+1$ entry to facilitate traversal of the elements through the n-th *brow*. The ja array is a rank-1 array of size nnz that provides the *bcol* index for each nonzero block. A third array is used to store the block entries proceeding from $ia(1)$ to $ia(n+1)-1$, where the scalar entries within each $n_b \times n_b$ block are stored in column-major order.

Several linear-solver options are provided within FUN3D; the scheme most commonly used in practice is the multicolor point-implicit relaxation shown in Algorithm 1 [27,28]. In this approach, the grid vertices are grouped into n_c color groups, such that no two adjacent vertices are assigned the same color. Typical values of n_c for meshes encountered in practice are 10-15. Since the matrix $\mathbf{A}$ involves only a nearest-neighbor stencil, unknowns within a color may be updated in parallel in a Jacobi-like fashion. Color groups are processed sequentially, where solution updates within each color depend solely on the latest values of $\mathbf{\Delta Q}$ in neighboring color groups. The overall process may be repeated using n_{iter} sweeps over the entire system; a value of 15 is often observed to result in suitable convergence of the nonlinear solution.

To improve cache performance, the system of equations is renumbered such that unknowns within a color appear in consecutive order. In Algorithm 1, $\mathbf{O_c}$ and $\mathbf{D_c}$ represent submatrices of $\mathbf{O}$ and $\mathbf{D}$, respectively, for the unknowns contained in color c. $\mathbf{R}_c$ represents the nonlinear residual subvector defined by unknowns belonging to color c. Line 4 of Algorithm 1 represents a standard block-sparse matrix-vector product. Line 5 requires an inversion of each $n_b \times n_b$ block of

the matrix $\mathbf{D_c}$. Here, a lower-upper (LU) decomposition of these blocks is computed beforehand and stored in place. The solution for the current block row is then obtained through a forward-backward substitution procedure. Throughout this work, the terms *block row* and *row* are used interchangeably, both referring to a matrix row of 5×5 dense blocks.

In addition to the shared-memory programming models to be presented here, the solver also accommodates an MPI message-passing approach using a standard domain-decomposition strategy for architectures with multiple sockets and/or multiple NUMA domains, as well as general multi-node, distributed-memory environments necessary for large-scale simulations. To recover the serial algorithm when using this approach, halo exchanges of partition boundary data are required at the completion of each color group before processing of the next color may proceed. To hide communication latencies associated with these halo exchanges each color group is further subdivided into values along partition boundaries and those remaining values lying entirely interior to the partition. When processing the unknowns within a color group, values along partition boundaries are determined first, then nonblocking MPI calls are used to initiate halo exchanges with neighboring partitions. Values interior to the partition are then evaluated while halo values are in flight. At the completion of the current color, each process waits for communication to complete prior to initiating the next color.

3 Architectures

Table 1 summarizes the relevant characteristics of the target architectures detailed in this section. Only characteristics relevant to the current study, which focuses on memory performance, are shown.

Table 1: Relevant characteristics of target architectures. *NUMA Domains Used* is the number of domains used (if configurable) to obtain optimal performance in this study. *Cores* refers to physical CPU cores, streaming multiprocessors, or compute units. *SP* refers to the single-precision (32-bit) vector length. *Peak Bandwidth* refers to the theoretical, as opposed to measured, peak.

	SKL	KNL	TX2	VE	V100	A100	MI50
NUMA Domains Used	2	1	2	2	1	1	1
Cores	40	64	56	8	80	128	60
Vector / Warp Length, SP	16	16	4	512	32	32	64
Peak Bandwidth, GB/s	256	485	318	1220	900	1600	1024

SKL Intel® Xeon™ Gold 6148 (SKL) is a dual-socket CPU with 20 physical cores per socket and 2 threads per core. Its theoretical peak aggregate memory

bandwidth is 256 GB/s. It has two vector units per core with 512-bit SIMD registers that support most AVX-512 instructions.

KNL Intel® Xeon Phi™ Knights Landing (KNL) is a family of manycore x86 processors equipped with up to 72 low-frequency cores each with four hardware threads, two 512-bit vector units per core, and up to 16 GB of configurable high-bandwidth (at least 485 GB/s) 3D-stacked MCDRAM. The KNL 7230 used in this study has 64 cores. All results are run in *flat* mode, where MCDRAM is exposed as a NUMA domain, as our test case requires less than 16 GB of memory.

TX2 The Marvell® ThunderX2® (TX2) used in this study is a dual-socket processor with 28 cores per socket. The theoretical peak memory bandwidth for a dual-socket system is 318 GB/s. STREAM Triad results [26] suggest that the maximum bandwidth achievable on the system is approximately 240 GB/s, or roughly 120 GB/s for each NUMA node. The TX2 in the current study was of unknown SKU, and STREAM Triad results were at best 201 GB/s. The system can be configured to use up to four-way SMT; however, the system was configured for two-way SMT for the testing considered here.

VE The NEC® SX-Aurora™ TSUBASA Vector Engine (VE) is a floating point coprocessor that interfaces with an x86 host through PCIe. Legacy CPU code can be compiled by the NEC® compiler and run through a seamless offloading process that does not require explicit data transfer between the host and coprocessor. Thus, legacy applications are ported and run with minimal effort. The VE is a long vector architecture with a 256 × 8-byte vector length, an order of magnitude beyond even the most recent AVX-512-equipped CPUs. Each VE has eight out-of-order 1.6 GHz cores and up to 48 GB of second generation High Bandwidth Memory (HBM2) with a theoretical peak aggregate memory bandwidth of 1.22 TB/s. The VE has a NUMA mode [19] that partitions its cores into two sets of four which share equal amounts of the last level cache and memory, decreasing cache conflicts. All results in the current study use this mode, which improves performance by a small amount ($\sim 1\%$).

V100 and A100 NVIDIA® Tesla™ V100 and A100 are the previous and current (as of this writing) generation of NVIDIA® Tesla™ GPUs. They are equipped with 16-32 and 40 GB of HBM2 memory with approximately 900 and 1600 GB/s of theoretical peak memory bandwidth, respectively. NVIDIA® GPU hardware leverages a SIMT approach distributed across a number of streaming multiprocessors (SMs), which in turn consist of multiple cores. Threads are organized in blocks, or cooperative thread arrays, where one or more blocks run on an SM. The threads in a block are further partitioned into subgroups of 32 threads known as warps. A warp runs on eight or sixteen cores of an SM in multiple clock cycles. The NVIDIA® GPUs used in the current study are of the SXM2 variant.

MI50 AMD® Radeon™ Instinct™ GPUs, which will comprise the ORNL exascale Frontier system [24], are based on the Vega architecture and there are several models currently available, including the MI50 used in this study, MI25, and MI60. The MI50 has 60 compute units with 64 stream processors per compute unit for a total of 3,840 stream processors [4]. It has 16 GB of HBM2 memory with a theoretical peak memory bandwidth of 1,024 GB/s. From the application developer's perspective, major differences between the NVIDIA® Tesla™ V100 and the AMD® MI50 include (a) memory bandwidth (900 GB/s and 1 TB/s, respectively); (b) the warp size of 32 threads on V100 and wavefront size of 64 threads on MI50; and (c) the lack of hardware support for floating-point atomic operations on MI50.

4 Test Case

The test case used here is based on transonic turbulent flow over the semispan wing-body configuration described in Ref. [16]. The freestream Mach number is 0.85, the angle of attack is zero degrees, and the Reynolds number based on the mean aerodynamic chord is 5 million. The computational mesh consists of 1,123,718 grid vertices, 1,172,171 prisms, 3,039,656 tetrahedra, and 7,337 pyramids. This problem size is representative of the workload that would typically be placed on a single compute node in practice. For the purposes of the current study, a single linear system is extracted from an arbitrary time step during the nonlinear convergence. The linear system contains a total of 18,998,518 nonzero off-diagonal blocks, or an average of approximately 17 off-diagonal blocks per mesh vertex. Timings reported below are for 15 sweeps over the entire system.

5 Fortran Implementation

The legacy FUN3D solver implementation is written in Fortran 90 and supports both MPI [3] and MPI+OpenMP [2] programming models. In the latter case, a separate MPI rank is typically placed on each NUMA domain. The memory layout is the CSR layout described in Section 1 and this implementation is referred to as "Fortran (CSR)" throughout, where it is used as a performance baseline (if applicable). Figure 1 shows the loop executed for each color. The outer loop is over matrix block rows in the color. The inner loop is over blocks in a matrix row. The matrix-vector product is manually unrolled over the inner $n_b \times n_b$ dimensions and computed using scalar variables. Forward-backward substitution is also manually unrolled. This structure has been determined to perform best on common CPUs such as Intel® Xeon™ processors. When using OpenMP, parallelization occurs over block rows of the matrix. Unless stated otherwise, benchmark results use the MPI+OpenMP model with one rank per NUMA domain and one thread per hardware thread.

```
!$omp parallel
!$omp do private(f1,f2,f3,f4,f5,n,j,icol,istart,iend)
  rhs_solve : do n = start, end

     f1 = -res(1,n)
     f2 = -res(2,n)
     f3 = -res(3,n)
     f4 = -res(4,n)
     f5 = -res(5,n)

     istart = iam(n)
     iend   = iam(n+1)-1

     do j = istart,iend
       icol = jam(j)

       f1 = f1   - a_off(1,1,j)*dq(1,icol)
       f2 = f2   - a_off(2,1,j)*dq(1,icol)
       f3 = f3   - a_off(3,1,j)*dq(1,icol)
       f4 = f4   - a_off(4,1,j)*dq(1,icol)
       f5 = f5   - a_off(5,1,j)*dq(1,icol)
       f1 = f1   - a_off(1,2,j)*dq(2,icol)
       f2 = f2   - a_off(2,2,j)*dq(2,icol)
       f3 = f3   - a_off(3,2,j)*dq(2,icol)
       f4 = f4   - a_off(4,2,j)*dq(2,icol)
       f5 = f5   - a_off(5,2,j)*dq(2,icol)
       f1 = f1   - a_off(1,3,j)*dq(3,icol)
       f2 = f2   - a_off(2,3,j)*dq(3,icol)
       f3 = f3   - a_off(3,3,j)*dq(3,icol)
       f4 = f4   - a_off(4,3,j)*dq(3,icol)
       f5 = f5   - a_off(5,3,j)*dq(3,icol)
       f1 = f1   - a_off(1,4,j)*dq(4,icol)
       f2 = f2   - a_off(2,4,j)*dq(4,icol)
       f3 = f3   - a_off(3,4,j)*dq(4,icol)
       f4 = f4   - a_off(4,4,j)*dq(4,icol)
       f5 = f5   - a_off(5,4,j)*dq(4,icol)
       f1 = f1   - a_off(1,5,j)*dq(5,icol)
       f2 = f2   - a_off(2,5,j)*dq(5,icol)
       f3 = f3   - a_off(3,5,j)*dq(5,icol)
       f4 = f4   - a_off(4,5,j)*dq(5,icol)
       f5 = f5   - a_off(5,5,j)*dq(5,icol)

     end do

```

(a) Setup and matrix-vector product.

```
     f2 = f2 - a_diag_lu(2,1,n)*f1
     f3 = f3 - a_diag_lu(3,1,n)*f1
     f4 = f4 - a_diag_lu(4,1,n)*f1
     f5 = f5 - a_diag_lu(5,1,n)*f1

     f3 = f3 - a_diag_lu(3,2,n)*f2
     f4 = f4 - a_diag_lu(4,2,n)*f2
     f5 = f5 - a_diag_lu(5,2,n)*f2

     f4 = f4 - a_diag_lu(4,3,n)*f3
     f5 = f5 - a_diag_lu(5,3,n)*f3

     f5 = f5 - a_diag_lu(5,4,n)*f4

     dq(5,n) = f5 * a_diag_lu(5,5,n)
     f1 = f1 - a_diag_lu(1,5,n)*dq(5,n)
     f2 = f2 - a_diag_lu(2,5,n)*dq(5,n)
     f3 = f3 - a_diag_lu(3,5,n)*dq(5,n)
     f4 = f4 - a_diag_lu(4,5,n)*dq(5,n)

     dq(4,n) = f4 * a_diag_lu(4,4,n)
     f1 = f1 - a_diag_lu(1,4,n)*dq(4,n)
     f2 = f2 - a_diag_lu(2,4,n)*dq(4,n)
     f3 = f3 - a_diag_lu(3,4,n)*dq(4,n)

     dq(3,n) = f3 * a_diag_lu(3,3,n)
     f1 = f1 - a_diag_lu(1,3,n)*dq(3,n)
     f2 = f2 - a_diag_lu(2,3,n)*dq(3,n)

     dq(2,n) = f2 * a_diag_lu(2,2,n)
     f1 = f1 - a_diag_lu(1,2,n)*dq(2,n)

     dq(1,n) = f1 * a_diag_lu(1,1,n)

  end do rhs_solve
!$omp end do
!$omp end parallel
```

(b) Forward-backward substitution.

Fig. 1: Baseline FUN3D Fortran point-implicit multicolor solver.

6 Optimized Performance Benchmarks

Each section herein describes the optimization of the solver for the section's respective architecture. The resulting optimized performance is shown in Table 2.

Table 2: Optimized solver results. The time given is for 15 sweeps through the linear system in milliseconds. *% Peak Bandwidth* is the application requested bandwidth divided by the theoretical peak bandwidth for the architecture (see Table 1). Application requested bandwidth is computed by dividing the amount of bytes that must pass at least once through main memory (DRAM/MCDRAM/HBM2) by the execution time. It does not consider cache effects.

	SKL	KNL	TX2	VE	V100	A100	MI50
Optimized Time, ms	166.0	140.0	167.0	102.0	48.8	30.9	64.9
% Peak Bandwidth	78.3	52.8	62.7	26.7	75.8	67.3	51.3

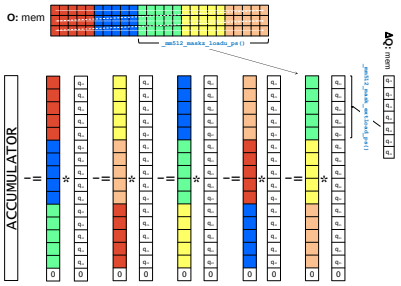
6.1 Intel® Xeon™ and Xeon Phi™ Knights Landing

The Fortran solver implementation (see Section 5) did not perform as expected for a bandwidth-bound code given KNL's main memory bandwidth of approx-

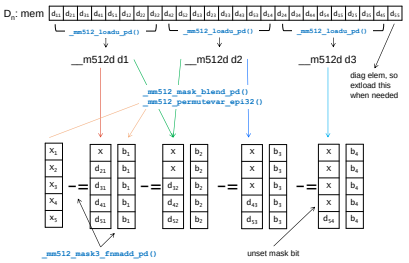
imately 485 GB/s. For this reason, an AVX-512 vector intrinsic [10] solver was developed. AVX-512 vector intrinsics are an abstraction just above the assembly level that can be used in a higher level language such as C++ and give the programmer fine-grained control over a thread's vector registers. There are also intrinsic instructions for memory prefetching, which is of interest in part due to the high latency of MCDRAM.

AVX-512 Intrinsic Solver The AVX-512 intrinsic solver processes a single matrix block row, computing the matrix-vector product of each 5×5 block and the vector $\mathbf{\Delta Q}$, performing forward-backward substitution using the resultant vector, and storing the updated $\mathbf{\Delta Q}$.

The matrix-vector product is performed on chunks of three 5×5 blocks. The vector length of 512 bits holds up to 16 32-bit values. Three columns of $\mathbf{O}$ are loaded into a vector register with the final lane being zero. Avoiding splitting the columns across registers minimizes code complexity and load instructions while retaining over 90% vector efficiency. Corresponding values of the vector $\mathbf{\Delta Q}$ are broadcast to 5 vector lanes in groups of three to fill a vector register using the `_mm512_mask_extload_ps` intrinsic. These two registers are multiplied and subtracted from an accumulator register using the `_mm512_fnmadd_ps` intrinsic. This process is repeated over the entire row. This produces 15 partial sums in the accumulator register. This register is permuted and summed to produce $\mathbf{b}$ in the first 5 lanes of the accumulator register. See Figure 2a for an illustration of the matrix-vector product on a chunk of three 5×5 blocks. A remainder loop handles rows with lengths not divisible by three.



(a) Mapping from memory to vector registers and partial summation.



(b) Forward-backward substitution.

Fig. 2: AVX-512 solver.

Forward-backward substitution cannot achieve efficient vectorization without processing multiple matrix rows. The implementation instead attempts to minimize register usage and maximize vectorization through register permutation intrinsics. $\mathbf{D}$ is loaded once into three vector registers and permuted into

operand registers as needed. Appropriate values of **b** are broadcast into multiple lanes using register permutations and summed using `_mm512_mask3_fnmadd_pd` with an appropriate mask. The resulting $\mathbf{\Delta Q}$ is stored to main memory. Streaming stores are not used as $\mathbf{\Delta Q}$ may reside in cache. See Figure 2b for an illustration of AVX-512 forward-backward substitution.

The SSE and KNCI intrinsic sets contain a memory prefetch intrinsic, `_mm_prefetch`, with a hint argument that specifies L1, L2, and nontemporal prefetches with additional exclusivity options (for memory to be modified). The AVX-512 intrinsic solver uses this intrinsic to prefetch data for the current matrix row into L1 followed by prefetching of the next row's data into L2.

Processing three matrix rows simultaneously seems a natural extension of this algorithm that would triple vectorization efficiency of the forward-backward substitution and $\mathbf{\Delta Q}$ writes, but improved performance has not been observed for this variant.

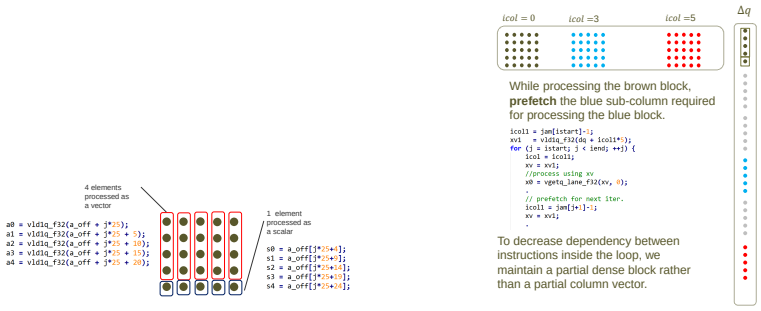
Though originally developed for KNL, the AVX-512 intrinsic solver is also used on Intel® Xeon™ processors that support common AVX-512 instructions.

6.2 Marvell® ThunderX2®

The ThunderX2® architecture offers Neon vector units capable of supporting 128-bit vector lengths. Effective use of these vector units is challenging for block-sparse matrix-vector operations when the block size is not a multiple of the vector length. This becomes particularly difficult for a Fortran or C compiler to address in an automated fashion, and experiments confirmed that compiler-generated code yields suboptimal performance on the ThunderX2®. For this reason, an implementation based on Neon intrinsics is described here.

The ThunderX2® can be configured to use up to four-way SMT; however, the system was configured for two-way SMT for the testing considered here. Optimal performance was observed while executing a single thread per core, where the thread has access to nearly all of the resources on the core. To address NUMA issues, a hybrid approach based on the use of MPI and OpenMP is used, with one MPI rank assigned to each of the two NUMA domains.

Vectorization Using Neon Intrinsics Processing a row of blocks for a sparse matrix-vector product involves multiplying each dense 5×5 block with a dense vector of size 5 corresponding to the column index of the block. This operation is repeated across the row, with results accumulated into a resultant vector of size 5. Since the vector length available on ThunderX2® is 128 bits, four simultaneous single-precision multiplies are possible. For $n_b = 4$, vectorization is straightforward. However, for the value of $n_b = 5$ used in the current study, each column of the 5×5 block is partitioned into two segments. The first segment consists of four elements that can be processed as a vector, while the remaining element is processed as a scalar. Figure 3a shows this partitioning and the Neon intrinsics instructions necessary to load the first four elements of each column as a vector and the remaining element as a scalar. Prefetching as shown in Figure 3b is used to further improve performance.



(a) Partitioning of a 5×5 block to enable vectorization.

(b) Prefetching $\mathbf{\Delta Q}$.

Fig. 3: ThunderX2® optimization strategies.

6.3 NEC® SX-Aurora™ TSUBASA Vector Engine

The primary challenge in achieving performance on the SX-Aurora™ is effective utilization of the long vector. The Fortran solver implementation (see Section 5) initially performed an order of magnitude slower on SX-Aurora™ than a conventional CPU (Intel® Xeon™ Gold 6148). To allow the NEC® Fortran compiler to vectorize over matrix block rows, the loops over rows and blocks were interchanged. Because each row may have a different number of blocks, a maximum number of blocks is computed for each color and used as the block loop range. Rows with fewer blocks than the maximum are conditionally computed and it is assumed the compiler will efficiently mask these operations when vectorizing. These changes increased the baseline performance by approximately 4.5×, but no further attempts at optimization using the original matrix memory layout were successful. In principle, one could extend the AVX-512 implementation described in Section 6.1 to a longer vector by vectorizing over the matrix rows. However, the AVX-512 implementation relies heavily on arbitrary register lane permutations, which are not easily done with the SX-Aurora™ instruction set.

SX-Aurora™ Optimizations Using Modified ELLPACK Memory Layout The ELLPACK memory layout [13] regularizes a sparse matrix by treating each matrix row as having the same length, padding with zero values to extend short rows up to the maximum row length. We modified this format and applied it to the matrix $\mathbf{O}$ as follows. The dimensions of the matrix (Fortran order) become $neq \times n_b \times n_b \times l_m$ where neq is the number of matrix rows, n_b is 5 in this case, and l_m is the maximum matrix row length. For the case described in Section 4, l_m is 29 and the average number of rows is approximately 17, thus significant padding is introduced.

This implementation uses the interchanged loop described in the previous section. It also makes use of the NEC® Fortran compiler's `vreg` directives [18],

which direct the compiler to treat local arrays as vector registers. The documentation states that packed registers (`pvreg`) of 512 floats are supported, but the `pvreg` directive did not produce working code in these experiments. An unroll directive was added to the outermost loop. The modified ELLPACK, loops, and directives improve performance by approximately another 3×, surpassing the performance of Intel® Xeon™ Gold 6148 for this kernel.

SX-Aurora™ Optimizations Using Modified SELL-C-σ Memory Layout The SELL-C-σ memory layout [15] improves upon ELLPACK at the cost of additional complexity. Matrix rows are sorted in groups of σ and zero-padded to the maximum row length in chunks of C rows. For the case described in 4, the parameters $C = 256$ and $\sigma = n_c$ were used, where n_c is the number rows in each color group. This results in less than 2% padding. The SELL-256-n_c layout improves performance by 1.25× over the modified ELLPACK layout.

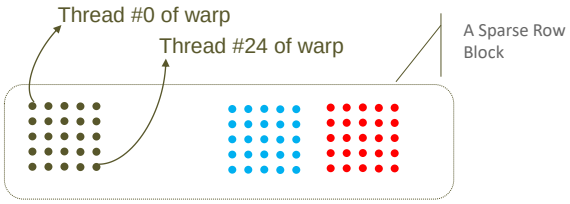
6.4 NVIDIA® Tesla™ V100 and A100 GPUs

CUDA [22] is a nonportable C++ language extension offering low-level control of NVIDIA® GPU hardware. To develop an efficient GPU implementation of the multicolor point-implicit solver, functions provided by the *cuSPARSE* [23] and *cuBLAS* [21] libraries were initially considered. The function *cusparseSbsrmv* multiplies a block-sparse matrix with a vector, and the function *cublasStrsmBatched* solves block systems of equations by performing forward and backward substitutions using an LU-decomposition of the diagonal block. Experiments showed that this approach yields suboptimal performance for linear systems representative of those encountered in typical FUN3D simulations.

Instead, optimized CUDA implementations of these functions were developed in Ref. [28]. To perform a block sparse matrix-vector product, the proposed algorithm allocates a number of warps to process a subset of the blocks in a single row of the sparse matrix. The mapping of a warp to process a block of a sparse matrix with $n_b = 5$ is illustrated in Figure 4. To perform forward and backward substitutions, a second kernel is invoked that assigns a single warp to process one diagonal block. Several challenges were encountered, including a variable extent of available parallelism, indirect memory addressing, low arithmetic intensity, and the need to accommodate different block sizes. To address these challenges, particular emphasis was placed on coalesced memory loads, the use of shared memory and prefetching, minimal thread divergence within warps, and strategic use of shuffle instructions available on recent hardware. Depending on the value of n_b, the new implementations realized performance gains of up to 7× over existing *cuSPARSE* and *cuBLAS* library functions [28].

6.5 AMD® Radeon™ MI50 GPU

The restructuring of the computation required for AMD and NVIDIA GPUs (see Section 6.4) is very similar. Since the AMD hardware calls for 64 threads




```
k = threadIdx.x % 5;
l = threadIdx.x / 5;

for (j=istart-1; j < iend; j++) {
    colid = jam[j];
    fk += A_OFF(k,l,j)*DQ(l,colid-1);
}
//save partial aggregation in shared memory
sm_f[k][l][threadIdx.y] = fk;
```

Fig. 4: Assignment of a warp to process a complete 5×5 block to ensure that consecutive threads of the warp load and process data from consecutive locations of device memory. The warp processes a complete row one block at a time, and aggregates partial results into a 5×5 block. The columns of the final aggregated block are reduced using shuffle instructions or shared memory (not shown here).

per wavefront, two versions of the algorithm have been implemented: (a) one block-row per wavefront with two nonzero blocks mapped to a wavefront, and (b) two block-rows per wavefront with half of a wavefront mapped to a nonzero block of a row. We used HIP to develop an optimized implementation on AMD GPU. HIP, or Heterogeneous-Computing Interface for Portability [6], is a C++ API similar to CUDA that has been developed by AMD.

One block-row per wavefront. In this algorithm, a wavefront processes two consecutive nonzero blocks of a row concurrently. Since a wavefront on the AMD GPU consists of 64 threads, 14 threads remain idle. The wavefront processes a row of the block-sparse matrix in a loop, where 2 consecutive nonzero blocks are processed by the wavefront at each iteration. The wavefront handles 50 ($2 \times (5 \times 5)$) matrix entries during each iteration. The appropriate elements of $\mathbf{\Delta Q}$ are also loaded from the read-only data cache, multiplied by the corresponding elements of the matrix, and then results are accumulated. After completion of the loop, the 50 partial results are aggregated into an output of 5 elements. The code segment to illustrate this computation is shown in Figure 5.

Two block-rows per wavefront. In this algorithm, a wavefront is assigned to process two consecutive block-rows with the first set of 32 threads (half-wavefront) processing the first block-row and the second set of 32 threads processing the second block-row. A half-wavefront processes one nonzero block of a row concurrently. Note that in this algorithm, it is not necessary that the two consecutive block-rows have an identical number of nonzero blocks. Consequently, not all of the 50 threads of a wavefront will always be active. The implementation of this algorithm is similar to the NVIDIA GPU version discussed in Section 6.4.

```
// nbk = 2
for (j = istart; j < iend - nbk + 1; j += nbk) {
    fk += A_OFF(k, l, j + nbki) * DQ(l, jam[j + nbki] - 1);
}
// process left over blocks
nbl = (iend - istart) % nbk;
if (tidx < nbl * nb2) {
    fk += A_OFF(k, l, j + nbki) * DQ(l, jam[j + nbki] - 1);
}
sm_f[k][l + nbki * 5][threadIdx.y] = fk;
```

Fig. 5: Code for one block-row per wavefront on AMD GPU.

7 Optimization of Programming Frameworks

This section attempts to address the question of whether a given programming framework allows the programmer to map a computation efficiently onto an architecture and recover the performance of an optimized implementation written in a sufficiently low level language (see Section 6).

7.1 OpenACC

The OpenACC programming model [1] is based on the use of compiler directives and offers the potential for portable implementations across multiple GPU architectures.

NVIDIA® Tesla™ V100 and A100 GPUs Prior development of an optimal CUDA implementation provided valuable insight in achieving a straightforward OpenACC implementation. Here, the launch parameters for each CUDA kernel were replaced with for-loops over thread blocks and the threads within each block. The sequential code annotated with OpenACC directives is shown in Figure 6; note the similarities with the CUDA implementation shown in Figure 4.

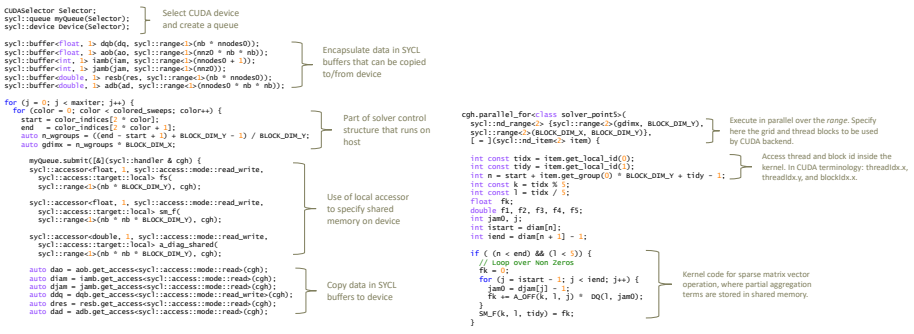
7.2 SYCL

SYCL is a cross-platform programming model based on C++ with support for different architectures [12]. SYCL implements a single-source, multiple compiler-passes model that allows the integration of source code for different architectures. The Intel® Data Parallel C++ (DPC++) compiler is based on SYCL with additional extensions, and provides support for a variety of OpenCL [11] devices including CPUs, FPGAs and GPUs [9]. Codeplay recently added experimental SYCL support for NVIDIA® GPUs, which avoids the use of OpenCL through use of the LLVM compiler [8]; OpenCL implementations for NVIDIA® GPUs are generally not effective due to limited NVIDIA support for OpenCL 1.2. Instead, this approach provides a plugin to DPC++ that enables compilation of SYCL code with direct CUDA support. This approach is used to evaluate SYCL performance for the NVIDIA® Tesla V100 GPU.

```
#pragma acc loop gang
for (blockID = 0; blockID < nBlocks; blockID++) {
#pragma acc loop worker
    for (tidy = 0; tidy < BLOCK_DIM_Y; tidy++) {
        n = start + blockID * BLOCK_DIM_Y + tidy;
        #pragma acc loop vector
        for (tidx = 0; tidx < 32; tidx++) {
            k = tidx % 5;
            l = tidx / 5;
            if ((n < end) && (l < 5)) {
                istart = iam[n] - 1;
                iend = iam[n + 1] - 1;
                fk = 0.0;
                #pragma acc loop seq
                for ( j = istart; j < iend; j++) {
                    icol = jam[j] - 1;
                    fk += A_OFF(k, l, j) *  DQ(l, icol);
                }
                // store partial terms in shared memory
                sm_f[k][l][tidy] = fk;
            }
        }
    }
.
.
}
```

Fig. 6: Listing of sequential code with OpenACC directives. Note the similarity of this code to the CUDA code shown in Figure 4, illustrating an identical restructuring of the computation.

NVIDIA® Tesla™ V100 GPU A SYCL implementation of the solver kernel has been developed and compiled with the Codeplay LLVM implementation. The SYCL code for the solver kernel is shown in Figure 7. Note the similarity of the SYCL implementation to the CUDA code in Figure 4, illustrating that SYCL exposes sufficient features to achieve a CUDA-like implementation. This flexibility is useful in expressing the restructured SYCL computation in a manner necessary to achieve good performance on NVIDIA GPUs.



(a) Data management. **(b)** Solver kernel.

Fig. 7: SYCL implementation.

7.3 HIP

The *HIPify* tool provided by AMD [5] has been used to convert the CUDA kernel implementation to HIP for execution on the NVIDIA® Tesla™ V100 GPUPUs. In this experiment, the *HIPify* tool did not alter any of the original CUDA kernel code.

7.4 OCCA

OCCA is an open source approach that enables development for a variety of devices including CPUs, GPUs, and FPGAs [17]. Back-end support is provided for targets such as CUDA, OpenMP, HIP, and OpenCL. The implementation is a simple extension to C and uses "attributes" to map code to a particular device. An implementation of the solver kernel using OCCA is shown in Figure 8. The *@outer* attribute in the outer for-loop indicates that the computation inside the loop can be parallelized, and this loop is mapped to thread blocks when using the CUDA back-end. The *@inner(0)* and *@inner(1)* loops map to the two dimensions of the thread block. The *@shared* attribute indicates the use of shared memory. Note that the code shown in Figure 8 is quite similar to the OpenACC and CUDA implementations.

```
for (int bx = 0; bx < Nblocks; ++bx; @outer(0)) {
    @shared float sm_f[nb][nb][NTY];;
    @shared double a_diag_lu_shared[nb][nb][NTY];
    @shared float fs[nb][NTY];
    for (int ty = 0; ty < NTY; ++ty; @inner(1)) {
        for (int tx = 0; tx < NTX; ++tx; @inner(0)) {
            int const k = tx % 5;
            int const l = tx / 5;
            int n = start + bx * NTY + ty -1 ;
            if ( (n < end) && (l < 5)) {
                int istart = iam[n]-1;
                int iend = iam[n + 1] - 1;
                double  fk = 0.0;
                for ( j=istart; j < iend; j++) {
                    jam0 = jam[j];
                    fk += A_OFF(k, l, j) *  DQ(l, jam0 - 1);
                }
                sm_f[k][l][ty] = fk;
            }
        }
    }
```

Fig. 8: OCCA implementation of the solver kernel.

8 Results

Table 3 summarizes all results. Although the vector intrinsic results are no more than 1.16× higher than Fortran (CSR) for SKL and TX2, this is due to their

limited memory bandwidth as the performance bottleneck. Run on a single core of SKL, the AVX-512 solver speedup over Fortran is greater than 1.5×. Moreover, the AVX-512 vector intrinsic solver on SKL achieves the highest percent of theoretical peak memory bandwidth among all codes in this study.

Table 3: Summary of results across two portability dimensions: architecture and programming model. Numeric values indicate performance relative to Fortran (CSR) on SKL (higher is better). Subjective ratings represent ease of implementation and optimization, respectively: E – easy, M – moderate, and H – hard. Percent values show the percent of theoretical peak bandwidth achieved. Red values indicate the optimized benchmark implementation for a given architecture. A "-" indicates an invalid or unimplemented combination.

	SKL	**KNL**	**TX2**	**VE**	**V100**	**A100**	**MI50**
Fortran (CSR)	1.0 E/M 69.1%	0.79 E/M 31.1%	0.97 E/M 53.9%	0.53 M/M 7.7%	-	-	-
Fortran (SELL-C-σ)	-	-	-	1.84 M/H 26.7%	-	-	-
OpenACC	-	-	-	-	3.77 E/H 74.1%	5.22 E/H 57.8%	-
CUDA	-	-	-	-	3.86 M/H 75.8%	6.08 M/H 67.3%	-
HIP	-	-	-	-	3.85 M/H 75.8%	-	2.90 M/H 51.3%
SYCL for CUDA	-	-	-	-	3.79 M/H 74.5%	-	-
Vector Intrinsics	1.13 H/H 78.3%	1.34 H/H 52.8%	1.13 H/H 62.6%	-	-	-	-
OCCA	-	-	-	-	3.76 M/H 74.0%	-	2.89 M/H 51.2%

TX2 performance should not be interpreted as representative of the architecture. The machine used in this study was an anomalous prototype with seemingly lower memory bandwidth than that reported by other TX2 users.

Optimizations for SX-Aurora^{TM} should not be considered complete. Though considerable speedup was achieved, a lower level approach such as intrinsics has yet to be implemented.

For the additional programming frameworks considered (OpenACC, HIP on V100, SYCL, and OCCA), optimized implementations were able to match (within $\sim 3\%$) the optimized benchmark for the architecture. In this work, each code is specific to a single architecture, so, for example, there are two HIP implementations, one for V100 and one for MI50. The exception to that is A100, where both the CUDA benchmark and the OpenACC version were developed and optimized for V100.

9 Conclusions and Future Work

Optimized implementations of the linear solver kernel have been established for the target architectures. For each additional programming framework considered, a solver has been implemented for a subset of the target architectures. Performance relative to the original Fortran (CSR) implementation on SKL has been reported, as well as the percent of theoretical peak bandwidth attained. Subjective ratings of implementation and optimization difficulty have been given for each combination. For this linear solver kernel, we conclude that, for the additional programming frameworks considered (OpenACC, HIP on V100, SYCL, and OCCA), it is possible to match the performance of a lower level implementation optimized specifically for the architecture. Performance of a single code across multiple architectures has not been considered and that is to be the subject of future work. A more optimized benchmark for SX-Aurora^{TM} will also be developed.

Acknowledgments The authors would like to express their appreciation to the following people for many helpful conversations pertaining to the current work: Justin Luitjens (NVIDIA Corporation), Erich Focht and Rudolf Fischer (NEC Corporation), John Linford (Arm Limited), Tim Warburton (Department of Mathematics, Virginia Tech); Noel Chalmers (AMD Incorporated), Sameer Shende (Department of Computer and Information Science, University of Oregon), and Jeff Hammond, Varsha Madananth, and Kevin O'Leary (Intel Corporation). The authors also wish to thank the High Performance Computing Incubator at the NASA Langley Research Center and the NASA Headquarters Office of Chief Engineer Research and Analysis program for providing support for this work. The support of Dr. Mujeeb Malik, Technical Lead for the Revolutionary Computational Aerosciences subproject within the NASA Aeronautics Research Mission Directorate Transformational Tools and Technologies Project, is also acknowledged.

References

1. OpenACC, https://www.openacc.org, last accessed 8/24/20
2. OpenMP, https://www.openmp.org, last accessed 8/24/20
3. The MPI Forum Website. http://www.mpi-forum.org, last accessed 8/24/20
4. AMD Incorporated: AMD Radeon Instinct MI50 Accelerator, https://www.amd.com/en/products/professional-graphics/instinct-mi50, last accessed 8/24/20
5. AMD Incorporated: HIP Porting Guide, https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-porting-guide.html, last accessed 8/24/20
6. AMD Incorporated: HIP Programming Guide, https://rocm-documentation.readthedocs.io/en/latest/Programming_Guides/HIP-GUIDE.html, last accessed 8/24/20
7. Biedron, R., Carlson, J., Derlaga, J., Gnoffo, P., Hammond, D., Jones, W., Kleb, W., Lee-Rausch, E., Nielsen, E., Park, M., Rumsey, C., Thomas, J., Thompson, K., Wood, W.: FUN3D Manual 13.6. NASA/TM-2019-220416 (2019)

8. Codeplay: Codeplay Contribution to DPC++ Brings SYCL Support for NVIDIA GPUs, https://www.codeplay.com/portal/news/2020/02/03/codeplay-contribution-to-dpcpp-brings-sycl-support-for-nvidia-gpus.html, last accessed 8/24/20
9. Intel Corporation: Intel oneAPI DPC++ Compiler (Beta), https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compiler.html, last accessed 8/24/20
10. Intel Corporation: Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide/, last accessed 8/24/20
11. Khronos Group: OpenCL, https://www.khronos.org/opencl/, last accessed 8/24/20
12. Khronos Group: SYCL, https://www.khronos.org/sycl/, last accessed 8/24/20
13. Kincaid, D.R., Oppe, T.C., Young, D.M.: ITPACKV 2D User's Guide (May 1989)
14. Korzun, A., Nielsen, E., Walden, A., Jones, W., Carlson, J.R., Moran, P., Henze, C., Sandstrom, T.: Effects of Spatial Resolution on Retropropulsion Aerodynamics in an Atmospheric Environment. AIAA SciTech Forum (2020)
15. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Bishop, A.R.: A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. SIAM Journal on Scientific Computing **36**(5), C401–C423 (Jan 2014). https://doi.org/10.1137/130930352
16. Laflin, K.R., Klausmeyer, S.M., Zickuhr, T., Vassberg, J.C., Wahls, R.A., Morrison, J.H., Brodersen, O.P., Rakowitz, M.E., Tinoco, E.N., Godard, J.L.: Data Summary from Second AIAA Computational Fluid Dynamics Drag Prediction Workshop. AIAA Journal of Aircraft **42**(5), 1165 – 1178 (2005)
17. Medina, D.S., St-Cyr, A., Warburton, T.: OCCA: A Unified Approach to Multi-Threading Languages. arXiv preprint arXiv:1403.0968 (2014)
18. NEC Corporation: SX-Aurora TSUBASA Fortran Compiler User's Guide, https://www.hpc.nec/documents/sdk/pdfs/g2af02e-FortranUsersGuide-018.pdf, last accessed 8/24/20
19. NEC Corporation: SX-Aurora TSUBASA VEOS NUMA Mode Guide for Partitioning Mode, https://www.hpc.nec/documents/guide/pdfs/VEOS_NUMA_Mode4PartitioningMode_E.pdf, last accessed 8/24/20
20. Nielsen, E.J., Diskin, B.: High-Performance Aerodynamic Computations for Aerospace Applications. Parallel Computing **64**, 20 – 32 (2017)
21. NVIDIA Corporation: cuBLAS, https://developer.nvidia.com/cublas, last accessed 8/24/20
22. NVIDIA Corporation: CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4Hicq83a9, last accessed 8/24/20
23. NVIDIA Corporation: cuSPARSE, https://developer.nvidia.com/cusparse, last accessed 8/24/20
24. Oak Ridge National Laboratory: Exascale System Expected to be World's Most Powerful Computer for Science and Innovation, https://www.olcf.ornl.gov/2019/05/07/no-scaling-back-doe-cray-amd-to-bring-exascale-to-ornl/, last accessed 8/24/20
25. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edn. (2003)
26. ANANDTECH: Assessing Cavium's ThunderX2: The Arm Server Dream Realized At Last. https://www.anandtech.com/show/12694/assessing-cavium-thunderx2-arm-server-reality (2018)

27. Walden, A., Nielsen, E., Diskin, B., Zubair, M.: A Mixed Precision Multicolor Point-Implicit Solver for Unstructured Grids on GPUs. In: Proceedings of the Ninth Workshop on Irregular Applications: Architectures and Algorithms. pp. 23–30. IA3 2019, IEEE Press, Los Alamitos, CA, USA (2019)
28. Zubair, M., Nielsen, E., Luitjens, J., Hammond, D.: An Optimized Multicolor Point-Implicit Solver for Unstructured Grid Applications on Graphics Processing Units. In: Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms. pp. 18–25. IA3 2016, IEEE Press, Piscataway, NJ, USA (2016)