Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

CPC 50th anniversary article

Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures

Ioan Hadade [a,b,*], Feng Wang [b], Mauro Carnevale [b], Luca di Mare [b]

[a] *Rolls-Royce Vibration UTC, Imperial College London, London, SW7 2AZ, United Kingdom*
[b] *Oxford Thermofluids Institute, University of Oxford, Oxford, OX2 0ES, United Kingdom*

A R T I C L E I N F O

Article history:
Received 14 February 2018
Received in revised form 19 June 2018
Accepted 5 July 2018
Available online xxxx

Keywords:
Unstructured grids
Computational fluid dynamics
Code optimisation
High performance computing
Parallel programming

A B S T R A C T

This paper presents a number of optimisations for improving the performance of unstructured computational fluid dynamics codes on multicore and manycore architectures such as the Intel Sandy Bridge, Broadwell and Skylake CPUs and the Intel Xeon Phi Knights Corner and Knights Landing manycore processors. We discuss and demonstrate their implementation in two distinct classes of computational kernels: face-based loops represented by the computation of fluxes and cell-based loops representing updates to state vectors. We present the importance of making efficient use of the underlying vector units in both classes of computational kernels with special emphasis on the changes required for vectorising face-based loops and their intrinsic indirect and irregular access patterns. We demonstrate the advantage of different data layouts for cell-centred as well as face data structures and architectural specific optimisations for improving the performance of gather and scatter operations which are prevalent in unstructured mesh applications. The implementation of a software prefetching strategy based on auto-tuning is also shown along with an empirical evaluation on the importance of multithreading for in-order architectures such as Knights Corner. We explore the various memory modes available on the Intel Xeon Phi Knights Landing architecture and present an approach whereby both traditional DRAM as well as MCDRAM interfaces are exploited for maximum performance. We obtain significant full application speed-ups between 2.8 and 3X across the multicore CPUs in two-socket node configurations, 8.6X on the Intel Xeon Phi Knights Corner coprocessor and 5.6X on the Intel Xeon Phi Knights Landing processor in an unstructured finite volume CFD code representative in size and complexity to an industrial application.

Program summary
Program Title: some_opt_for_unstructured_cfd
Program Files doi: http://dx.doi.org/10.17632/zyh2zkf3jw.1
Licensing provisions: GNU General Public License 3 (GPL)
Programming language: C/C++
Nature of problem: The solution of fluid flow problems in the vicinity of complex geometries mandates the utilisation of unstructured grids. However, this flexibility of unstructured mesh methods in dealing with complicated geometries comes at a cost of increased difficulty in extracting high performance out of modern processors. We provide implementations for a number of optimisations useful for improving the performance of unstructured CFD codes on modern multicore and manycore architectures.
Solution method: grid renumbering via Reverse Cuthill–Mckee, code transformations necessary for enabling vectorisation, face colouring/reordering for removing dependencies at the face end-points when accumulating residuals, data layout transformations for reducing cache misses, hand-tuned gather and scatter primitives for in-register transpositions, software prefetching via auto-tuning and multithreading for exploiting SMT features of modern processors.

1. Introduction

The flexibility of unstructured mesh methods in dealing with complicated geometries comes at a cost of increased difficulty in achieving high performance on modern computer architectures. This is due in great part to the data structures required for

* Corresponding author at: Rolls-Royce Vibration UTC, Imperial College London, London, SW7 2AZ, United Kingdom.

E-mail address: i.hadade@imperial.ac.uk (I. Hadade).

https://doi.org/10.1016/j.cpc.2018.07.001
0010-4655/© 2018 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

expressing grid connectivity and the resulting indirect and irregular memory access patterns. In finite volume discretisations, such data structures and access patterns appear when iterating over the faces or edges of the computational domain for the purpose of evaluating fluxes, gradients and limiters. These kernels are usually structured as a sequence of gather, compute and scatter operations where variables are gathered from pairs of cells or vertices sharing a face or edge followed by the calculation and scatter of results to the respective face or edge end-points.

An example of a typical face-based loop can be seen in Listing 1 where unknowns are gathered from neighbouring cells and used together with mesh geometrical attributes to compute flux residuals that are subsequently accumulated and scattered back to the face-end points. For edge-based solvers, one can simply replace faces with edges and cells with vertices or nodes.

Listing 1: Example of a simplified face-based loop on unstructured grids

```
for( ic=ics;ic<ice;ic++ )
{
     i1= ifq[0][ic];          // index of cell 1
     i2= ifq[1][ic];          // index of cell 2
     u1= q[i1];               // gather from cell 1
     u2= q[i2];               // gather from cell 2
     f= geo[ic]*(u2-u1);      // compute residual
     rhs[i1]-= f;             // scatter to cell 1
     rhs[i2]+= f;             // scatter to cell 2
}
```

The gather and scatter operations in Listing 1 can operate across large and random strides in memory depending on the mesh topology and algorithms used at the grid generation stage [1]. As a result, they are inherently inefficient on modern processor architectures for a number of reasons. First of all, accessing memory at irregular strides does not exploit the cache hierarchy as data cannot be reused from the faster higher levels due to lack of spatial and temporal locality. Secondly, the compiler would be unable to vectorise the loop construct in Listing 1 due to the use of indirect addressing and potential data dependencies between the face end-points when accumulating the residuals. This also prevents the exploitation of thread-level parallelism due to the risk of race conditions. Finally, accessing memory indirectly can also prohibit the performance of hardware prefetchers and therefore severely reduce any opportunity for memory parallelism.

A typical unstructured finite volume code will spend more than two thirds of its execution time evaluating face-based loops. Consequently, addressing the limitations that prevent them from making optimal use of the available vector units, threads and memory hierarchy is imperative for achieving high performance on modern processors. However, some optimisations that are beneficial to face-based loops might be detrimental to the performance of cell-based loops where the remaining execution time is spent. This requires that both face-based and cell-based kernels are optimised and that the impact of each optimisation is assessed and evaluated across the whole application.

In this paper, we present a number of such optimisations for improving the performance of unstructured finite volume CFD applications on modern multicore and manycore processors. We demonstrate their implementation in a kernel computing inviscid second order fluxes as an example of face-based loops and in a kernel computing state vector updates as an example of cell-based loops. We discuss the trade off between programming effort and performance improvements for each optimisation and demonstrate how a single code base can be used to target a wide range of architectures via approaches such as auto-tuning and simple code abstractions. Finally, we report full application speed-ups in a code of complexity and size similar to that of an industrial application ranging between 2.8 and 3X across two-socket Intel Xeon CPU nodes and a speed-up of 8.6X on the Intel Xeon Phi Knights Corner and 5.6X on Intel Xeon Phi Knights Landing processor relative to their respective baseline.

The rest of the paper is organised as follows. Section 2 presents related work and a description of the code used as a test vehicle in this study along with details regarding its numerical implementation, selected test case and computational kernels. In-depth description of the evaluated hardware architectures and software environment is given in Section 3 whilst a description of the applied optimisations is presented in Section 4. Section 5 presents results and associated discussions whilst concluding remarks are given in Section 6.

2. Background

2.1. Related work

Anderson et al. [2] presented the optimisation of FUN3D [3], a tetrahedral vertex-centred unstructured mesh code developed at the NASA Langley Research Center for the solution of the compressible and incompressible Euler and Navier–Stokes equations and for which they received the 1999 Gordon Bell Prize [4]. Their optimisations were based on the concept of memory centric computations whereby the aim was to minimise the number of memory references as much as possible in the recognition that flops are cheap relative to memory load and store operations. The authors achieved this by increasing spatial locality with the help of interlacing in which data items that are required in close succession such as unknowns are stored contiguously in memory based on data layouts such as Array of Structures (AoS). They also reduced the impact of the underlying gather and scatter operations by renumbering the mesh vertices using the Cuthill–Mckee [5] sparse matrix bandwidth minimiser. Their work was subsequently extended in the context of the FUN3D code by a number of studies such as Gropp et al. [6] which introduced performance models in order to guide the optimisation process by classifying the operational characteristics of the computational kernels and their interaction with the underlying hardware, Mudigere et al. [7] who demonstrated shared memory optimisations on modern parallel architectures including vectorisation and threading through a hybrid MPI/OpenMP implementation, Al Farhan et al. [8] who presented optimisations specific to the Intel Xeon Phi Knights Corner processor as well as Duffy et al. [9] who ported FUN3D for execution on graphical processing units obtaining a factor of two speed-up as a result.

More recently, Economon et al. [10] presented the performance optimisation of the open-source SU2 [11] unstructured CFD code on Intel architectures. Their work demonstrated the impact of a number of optimisations such as vectorisation, edge reordering, data layout transformations for improving single core performance on modern multicore architectures as well as optimisations of the linear solver in order to remove the impact of performing collective operations at large core counts. As a result, they obtained speed-ups of more than a factor of two at single node and multi node granularities.

A different approach for optimising unstructured grid applications is by implementing such optimisations at a higher level of abstraction through a Domain Specific Language (DSL). Examples of such initiatives with respect to unstructured CFD solvers can be found by examining the work revolving the OP2 framework [12–15] and Liszt [16].

The work in this paper complements these endeavours in that it evaluates new architectures such as the Intel Xeon Skylake and Intel Xeon Phi Knights Landing processors and presents similar as well as novel optimisations. We study their impact across one of the broadest selection of multicore and manycore processors in literature and discuss commonalities and differences in their implementation depending on the underlying platform. Furthermore, the optimisations presented herein are useful for both approaches in code optimisations, be it at traditional levels of abstraction or at the level of a DSL.

2.2. Application

The test vehicle for this study is the in-house CFD solver AU3X [17,18]. AU3X uses a cell-centred finite volume approach to solve the unsteady Favre-averaged Navier–Stokes equations on unstructured meshes. Steady solutions are obtained by pseudo time marching and time accurate solutions by dual time stepping [19]. The governing equations, spatial discretisation and time integration schemes are briefly described in the following sections in order to complement the implementation details and code optimisation study.

2.3. Governing equations

The Favre-averaged Navier–Stokes equations for compressible flows in the differential form read:

$$\begin{aligned}
&\frac{\partial\overline{\rho}}{\partial t}+\frac{\partial(\overline{\rho}\tilde{v}_i)}{\partial x_j}=0\\
&\frac{\partial(\overline{\rho}\tilde{v}_i)}{\partial t}+\frac{\partial(\overline{\rho}\tilde{v}_i\tilde{v}_j)}{\partial x_j}=-\frac{\partial\overline{p}}{\partial x_i}+\frac{\partial}{\partial x_j}(\tilde{\tau}_{ij}+\tau^t_{ij})\\
&\frac{\partial(\overline{\rho}\tilde{E})}{\partial t}+\frac{\partial(\overline{\rho}\tilde{v}_j\tilde{H})}{\partial x_j}=-\frac{\partial}{\partial x_j}(\kappa\frac{\partial\tilde{T}}{\partial x_j}+\tilde{v}_i(\tilde{\tau}_{ij}+\tau^t_{ij}))
\end{aligned} \tag{1}$$

where,

$$\begin{aligned}
&\tilde{\tau}_{ij}=2\mu(\tilde{S}_{ij}-\frac{1}{3}\frac{\partial\tilde{v}_k}{\partial x_k}\delta_{ij}),\quad \overline{p}=(\gamma-1)\overline{\rho}(\tilde{E}-\frac{1}{2}\tilde{u}_j\tilde{u}_j),\\
&\kappa\frac{\partial T}{\partial x_j}=\frac{\gamma}{\gamma-1}\frac{\mu}{P_r}\frac{\partial}{\partial x_j}(\frac{\overline{p}}{\overline{\rho}}),
\end{aligned} \tag{2}$$

The tilde "∼" and overbar "−" represent Favre averaging and Reynolds averaging respectively. The working fluid is air and it is treated as calorically perfect gas while γ and the Prandtl number P_r are held constant at 1.4 and 0.72 respectively. μ is evaluated by Sutherland's law and is based on a reference viscosity of $1.7894\times10^{-5}\,\frac{\mathrm{kg}}{\mathrm{ms}}$ together with a reference temperature of 288.15 K and Sutherland's constant at 110 K. If the Boussinesq assumption holds, the Reynolds stress τ^t_{ij} can be written as a linear function of the mean flow gradient:

$$\tau^t_{ij}=2\mu_t(\tilde{S}_{ij}-\frac{1}{3}\frac{\partial\tilde{v}_k}{\partial x_k}\delta_{ij}) \tag{3}$$

Turbulent viscosity μ_t is computed by turbulence models. In this paper, the Wilcox $k-\omega$ turbulence model is used and additional equations are required for k and ω. Readers can refer to Wilcox [20] for more details.

2.4. Spatial discretisation

The flow variables are stored at the cell centres and the boundary conditions are applied at the ghost cells, the positions of which are generated by mirroring the positions of the cells immediately adjacent to the boundary. The inviscid and viscous fluxes are evaluated at the cell-to-cell and boundary-to-cell interfaces. The schematic of the finite volume scheme is shown in Fig. 1. Flow gradient is computed at the cell centre using the weighted least square procedure [21]. The matrix of the weighted least square gradient is evaluated once at pre-processing for static grids and at every nonlinear iteration for moving meshes.

The inviscid fluxes are computed by the upwind scheme using the approximated Riemann solver of Roe [22]. Second order spatial discretisation is obtained by extrapolating the values from the cell centre to the interface via the MUSCL [23] with the van Albada limiter [24]. The viscous fluxes at the interface are computed by using the inverse of the distance weighting from the ones evaluated at the cell centres on both sides of the interface while source terms are evaluated at the cell centres and are assumed to be piecewise constant in the cell.

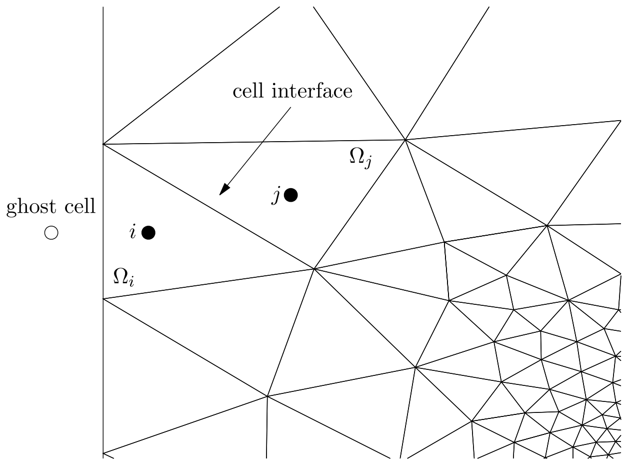


Fig. 1. Schematic of the cell-centred finite volume scheme on unstructured grids.

2.5. Time integration

After inviscid, viscous fluxes and source terms are computed for each cell, the coupled system in Eq. (1) can be described as the following:

$$\Omega_i\frac{dU_i}{dt}=-\sum_{j=1}^{N}R_i(U_j) \tag{4}$$

where U_i are the conservative variables of cell i, namely $(\overline{\rho},\overline{\rho}\tilde{v}_i,\overline{\rho}\tilde{E})^T$, Ω_i the cell volume, U_j the conservative variables of the neighbouring cells of U_i, N the number of neighbouring cells and R_i the right hand side of cell i, which are the fluxes evaluated at each cell. Here we assume no mesh motion and Ω_i remains a constant for each cell in the computation.

The system in Eq. (1) is solved implicitly by first applying the backward Euler scheme:

$$\Omega_i\frac{\Delta U_i}{\Delta t}=-\sum_{j=1}^{N}R_i(U_j^{n+1}) \tag{5}$$

where n is the solution at the current level, $n+1$ is the solution to be solved in the next level and $\Delta U_i=U_i^{n+1}-U_i^n$

Expanding $R_i(U_j^{n+1})$ in Taylor series, Eq. (5) becomes:

$$\Omega_i\frac{\Delta U_i}{\Delta t}=-\sum_{j=1}^{N}R_i(U_j^n)-\sum_{j=1}^{N}\frac{\partial(R_i(U_j^n))}{\partial U_j}\Delta U_j^n \tag{6}$$

where $\frac{\partial(R_i(U_j^n))}{\partial U_j}$ is the flux Jacobian. Eq. (6) can be re-arranged and the flux Jacobian is approximated by its spectral radius. The resulting linear system reads:

$$[J_i^n(\frac{\Omega_i}{J_i^n\Delta t}+1)]\Delta U_i^n=-\sum_{j=1}^{N}R_i(U_j^n) \tag{7}$$

Eq. (7) is the resulting linear system to march the solution from time level n to $n+1$, and it is solved by the Newton–Jacobi method where J_i^n is the spectral radius of the flux Jacobian matrix which is accumulated across the cell interfaces. Linearised fluxes $\frac{\partial(R_i(U_j^n))}{\partial U_j}\Delta U_j^n$ are required to update the solutions at each Newton–Jacobi iteration and they are evaluated exactly for the inviscid and viscous fluxes. For the Wilcox $k-\omega$ turbulence models, an approximation is used for linearising the governing equations for k and ω. The Newton–Jacobi is executed for user-specified iterations

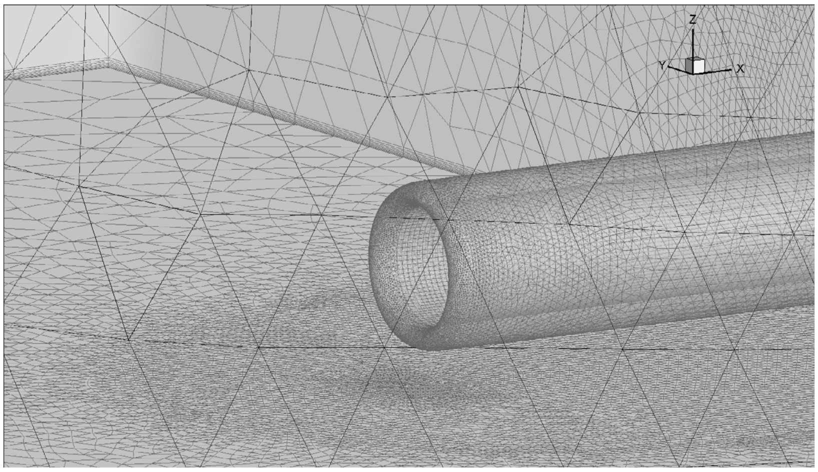


Fig. 2. Computational domain.

to march the solution from n to $n + 1$, the right and left hand sides are then updated, and the Newton–Jacobi is invoked again. This process proceeds until a user-specified convergence criterion is met.

2.5.1. Test case

The numerical test case used for this optimisation study represents an aero-engine intake operating near ground. Validation and numerical investigation using the AU3X code have been previously presented by Carnevale et al. [25,26] and utilised experimental data provided by Murphy et al. [27]. The computational domain based on an unstructured mesh can be seen in Fig. 2. Near wall regions have been discretised with hexahedral elements for boundary layer prediction whilst prismatic elements have been used in the free stream domain. Furthermore, two mesh sizes have been utilised throughout our work, namely `mesh 1` which contains 3.3×10^6 elements and `mesh 2` with 6.3×10^6 elements respectively in order to discount any effects of the problem size in the results.

2.5.2. Computational kernels

Face-based loops are characterised by a relatively high arithmetic intensity as they involve a large number of floating point operations per pair of adjacent cells. Furthermore, they contain a mixture of irregular and regular access patterns for gathering and scattering cell-centred variables and for evaluating face properties such as area and normals. For example, the face-based kernel used as an example in this study computes second order inviscid fluxes and exhibits a 1.3 arithmetic intensity (flops per byte ratio). In contrast, cell-based loops tend to perform a modest number of calculations per memory load operation. This is evidenced by the cell-based kernel used as an example in this study which computes updates to state vectors and exhibits an arithmetic intensity of 0.18. Thus, we expect face-based loops to benefit the most from optimisations that improve the throughput of floating point computations such as vectorisation and to scale with the available number of computational cores. For cell-based loops, we expect them to scale with the available memory bandwidth although it will be of interest to assess the impact that optimisations such as vectorisation or data layout transformations have on their performance and whether optimisations useful for face-based kernels have any negative impact on the performance of cell-based loops or vice-versa.

The optimisations presented in this paper have been implemented across all other face-based and cell-based kernels in the application. This is reflected in the results presenting full application performance as time per solution update (Newton–Jacobi iteration).

3. Experimental setup

We continue by describing the architectural features of the processors used in this paper and give special consideration to details that are relevant to the optimisation of unstructured mesh computations.

3.1. Intel Xeon Sandy Bridge

The Sandy Bridge (SNB) microarchitecture was the first to introduce 256-bit vector registers and an extension to the SIMD instruction set via Advanced Vector eXtensions (AVX) [28]. The AVX instruction set maintains compatibility with its predecessors and is implemented on top of Streaming SIMD Extensions (SSE) by fusing two 128-bit SSE registers. The latter design consideration has an impact on certain operations that target data elements across 128-bit lanes such as shuffles and permutations. In theory, the load ports in Sandy Bridge can perform 256-bit loads via AVX, however, achieving this bandwidth requires the simultaneous usage of the two available load ports. This limitation leads to situations where vectorisation might not deliver the expected performance boost due to load port pressure [29]. In terms of the cache hierarchy, the L1 is 32 KB and 8-way associative, L2 is 256 KB and L3 is 20 MB and shared across the cores on the die. Integration of all physical cores on the chip is done via a ring-based interconnect [29].

3.2. Intel Xeon Broadwell

The Broadwell (BDW) microarchitecture is the successor to Haswell to which it brings a number of enhancements such as latency improvements for floating-point multiply operations and throughput of gather instructions [29]. Compared to Sandy Bridge, the largest differences were introduced in the Haswell microarchitecture which came with a completely new core design, improved execution unit and a revamped memory subsystem [29]. The execution unit was extended with the addition of two ports, one for memory and one for integer operations which increases the scope of instruction-level parallelism and alleviates port pressure. To that end, Haswell, and Broadwell by extension, dedicate two execution ports for performing Fused-Multiply-Add (FMA) SIMD operations with a theoretical peak performance of sixteen double precision operations per cycle. Furthermore, they also support operations such as gather for loading non-contiguous elements into SIMD registers. The latter is particularly relevant to our work since the gather primitives in unstructured grid computations naturally map to such instruction extensions. These advancements in instruction set capabilities are delivered in Haswell and Broadwell through the AVX2 extensions. Furthermore, as opposed to Sandy Bridge, both Haswell and Broadwell deliver true 256-bit load capability due to improvements to the L1 cache which can service each of the two load ports with a combined 64 bytes (2 × 256-bit) of data and 32 bytes (256-bit) bandwidth for the store port per cycle. For this reason, the Broadwell microarchitecture is more likely to highlight the benefits of vectorisation and other memory related optimisations since the pressure on the load ports will be minimised.

3.3. Intel Xeon Skylake Server

The Skylake Server (SKX) microarchitecture was released in July 2017 and is the successor to Broadwell [30]. At core level, Skylake Server increases the vector register size to 512-bits and adds support for AVX-512 instructions. Although the execution unit has the same number of ports as Broadwell, port zero and one can either perform AVX/AVX2 vector computations on 256-bit lanes or fused 512-bit AVX-512 computations. Port five is exclusive to AVX-512 execution. Therefore, to take full advantage of this architecture, vector computations should target the wider vector lanes via AVX-512 as it utilises the highest available throughput. Skylake Server can perform in theory thirty two double precision computations per cycle when utilising both AVX-512 FMA units which is twice more than Broadwell. In terms of the cache subsystem, the L1 cache on Skylake offers similar latency and size to Broadwell, 32 KB at 4–6 cycles [29]. The major difference however is based on the increase in bandwidth to 128 bytes (2 × 512-bits) for loads and 64 bytes (512-bits) for stores which are required by AVX-512 operations. In essence, the L1 cache can service up to two entire cache lines (64 bytes each) to the load ports if AVX-512 is used and the underlying data is aligned to 64 byte boundary which make these considerations crucial for achieving best performance. Radical changes are also present in the L2 cache which sees a factor of four increase in size compared to Broadwell (1 MB versus 256 KB) [29]. The L3 cache is marginally smaller in size than on Broadwell and is configured as a victim cache (non-inclusive) to the higher levels. The effect of these changes means that applications making use of communication avoiding algorithms such as loop tiling or cache blocking should target the L2 cache rather than L3 with the same considerations applying for prefetching. Access to main memory can be serviced by up to six memory channels on Skylake Server versus four on Broadwell which should provide a 50% increase in available bandwidth as evidenced by our STREAM [31] results. However, this also means that for memory bound applications, the difference in performance between Broadwell and Skylake should be approximately 50% and 2X if the application is compute bound. Further changes are also present for the on-chip interconnect topology where the previous ring implementation is replaced with a 2D mesh interconnect which was initially implemented on the Intel Xeon Phi Knights Landing [29]. This enables the Skylake Server microarchitecture to scale to as many as 28 cores per die.

3.4. Intel Xeon Phi Knights Corner

The Intel Xeon Phi Knights Corner (KNC) coprocessor can be classed as an x86-based Shared-Memory-Multiprocessor-on-a-chip [32] with more than sixty physical cores on the die each supporting four hardware threads. A KNC core contains one Vector Processing Unit (VPU) that can operate on 32 512-bit wide registers and 8 mask registers based on the Integrated Many Core Instruction Set (IMCI) extensions. The IMCI extensions however are not compatible with other SSE/AVX/AVX2/AVX-512 implementations. The functionality offered by the VPU is heavily geared towards floating point computations with support for FMA, gather and scatter operations useful for unstructured mesh computations. Theoretically, the VPU can execute one FMA operation per cycle (sixteen DP FLOPs) however, due to the in-order execution nature of the core, either prefetching or more than one in-flight thread is required to keep the VPU busy. This is due to the fact that every missed load from the L1 cache leads to a pipeline stall unless context can be switched to another in-flight thread. The cores on the die are connected via a bi-directional ring interconnect that offers full cache coherence. Communication with the host CPU is performed via the PCI-Express bus in a similar fashion to GPGPUs. Although KNC can execute in a number of modes such as offload, symmetric or native, throughout our study we have chosen the latter as it removes any unnecessary synchronisation constructs and truly assess the computational characteristics and performance of the platform.

3.5. Intel Xeon Phi Knights Landing

The Knights Landing (KNL) architecture is the successor of KNC and second iteration of the Xeon Phi family series. The KNL is the first self-boot manycore processor able to run a standard operating system [33] and therefore differentiates itself to all other coprocessor and accelerator platforms. The KNL architecture provides binary compatibility with the Xeon multicore CPUs and an integrated high performance memory-on-package (MCDRAM) [34]. The basic building block of the KNL architecture is the tile which is replicated across the entire chip on a 2D lattice interconnect. A tile is further composed of two cores sharing a 1MB L2 cache and associated memory control agents. The KNL core is based on the Intel Atom (Silvermont) architecture and includes additional features targeting floating point workloads and supports up to four in-flight threads [33]. For the purpose of compute, the core integrates two Vector Processing Units (VPUs) each supporting AVX-512 execution. Implementation of SSE/AVX/AVX2 instructions [33] is present on one of the two vector units and not on both. As such, similar to SKX, full throughput can only be achieved by AVX-512 calculations as SSE/AVX/AVX2 can only utilise half of the available vector units. Due to the out of order nature, one thread per KNL core can saturate all available core resources. The L1 cache can perform two 64 byte (2 × 512-bits) loads and one 64 byte (1 × 512-bits) stores per cycle and supports unaligned and split loads accesses across multiple cache lines which was not previously possible in KNC. Furthermore, the L1 cache also implements special logic for gather and scatter operations whereby a single instruction can access multiple memory locations at once without the need of a blocking loop implementation as found in KNC [35]. The L2 cache is shared by the two cores in a tile via the Bus Interface Unit (BIU) and has a bandwidth of 64 bytes (1 × 512-bits) for reads and 32 bytes (1 × 256-bits) for writes per cycle. This can therefore become a bottleneck for memory bound workloads when both cores issue AVX-512 vector loads and store operations. KNL can support various clustering and memory mode configurations which were traditionally hard-wired during the chip manufacturing process but are now exposed as bootable options [33]. Clustering modes target ways in which data is routed on the 2D mesh in the event of a cache miss in L2 [34]. An L2 miss in KNL involves the interaction of three actors: the tile from where the miss originated, the Cache Homing Agent (CHA) that tracks the location and state of that address and the actual memory location [33]. In the All-to-All configuration, all memory addresses are distributed uniformly across all CHA's without any locality considerations. This mode provides the lowest performance out of all available configurations and is usually used as fall-back in the event of memory imbalance or errors. In the Quadrant configuration, the entire die is divided into four distinct regions and memory addresses are mapped to the caching agents which reside in the same quadrant. This creates

Table 1
Hardware and software configuration of compute nodes used in this paper. The SIMD ISA represents the latest vector instruction set architecture supported by the platform.

	SNB	BDW	SKX	KNC	KNL
Version	E5-2650	E5-2680	Gold 6140	7120P	7210
Sockets	2	2	2	1	1
Cores	8	14	18	61	64
Threads	2	2	2	4	4
Clock (GHz)	2.0	2.4	2.3	1.2	1.3
SIMD ISA	AVX	AVX2	AVX-512	IMCI	AVX-512
SIMD width	256-bit	256-bit	512-bit	512-bit	512-bit
L1 Cache (KB)	32	32	32	32	32
L2 Cache (KB)	256	256	1024	512	1024
L3 Cache (MB)	20	35	25	–	–
DRAM (GB)	32	128	196	16	96/16
DRAM type	DDR3	DDR4	DDR4	GDDR5	DDR4/MCDRAM
Stream (GB/s)	71	118	186	181	82/452
Compiler			icpc 17.0		
MPI Library			Intel MPI 2017		

affinity between the memory location and the CHA which reduces latency and provides an increase in bandwidth. Finally, the Sub-Numa-Clustering (SNC) mode can be configured as either a four way or two way division of the chip analogous to a four or two socket Xeon node where affinity exists between all agents: tile, CHA and memory. This configuration introduces Non-Uniform-Memory-Access (NUMA) considerations that have to be exploited on the application-side as accessing data owned by a tile from a different region will lead to increased latency due to the longer path traversed on the 2D mesh. However, if implemented correctly, this mode of operation can provide the best performance as all communications are localised. In regard to memory modes, the KNL can support three distinct memory configurations. The first option is Cache mode in which the 16 GB MCDRAM acts as a transparent direct mapped memory side cache. In flat mode, the MCDRAM is exposed as an explicit memory buffer that must be managed by the application and which has a different address space than main DDR. Hybrid mode combines both of the previous options where MCDRAM can be used as cache and explicit high bandwidth memory based on pre-defined ratios. For our study, experimental results for the KNL processor were obtained using the ARCHER-KNL [36] evaluation platform. All runs have been performed in quadrant cluster mode as this was the global configuration across all nodes. In regard to memory modes, separate queues allowed for the evaluation of both cache and flat modes.

3.6. *Hardware configuration and software environment*

Table 1 presents the configuration of the computational nodes used in this study with respect to processor version, node architecture and software environment.

4. Optimisations

This section gives an in-depth account of the various optimisations applied to face-based and cell-based kernels together with a discussion on the particularities of their implementation depending on the underlying hardware platform.

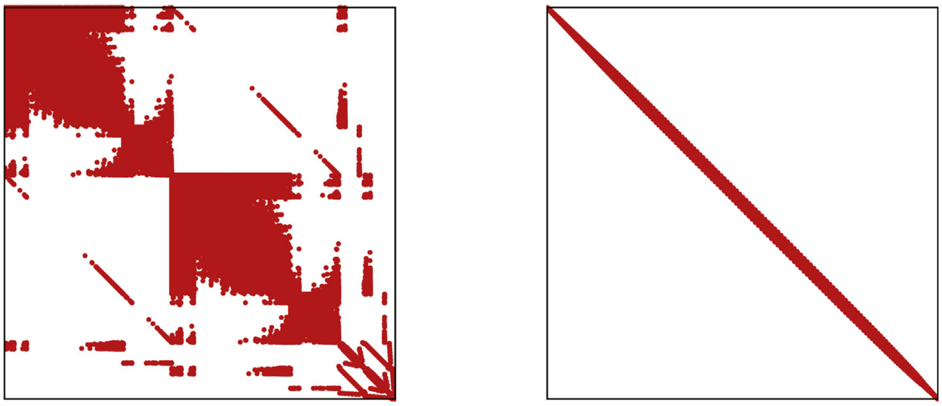
4.1. *Grid renumbering*

The first optimisation aims to improve the access pattern in face-based loops by minimising the distance between memory references when gathering and scattering data from and to pairs of cells that share a face. In our work, we achieved this using the Reverse Cuthill Mckee (RCMK) [5] sparse matrix bandwidth minimiser. The RCMK algorithm reorders the non-zero elements of the adjacency matrix derived from the underlying mesh topology so as to cluster them as close as possible to the main diagonal [37]. An example of the resulting bandwidth reduction when applying RCMK on the smallest mesh in our test case can be seen in Fig. 3. Following the renumbering, we sort the faces with respect to the first index which results in a list of cell numbers that increase monotonically. A subsequent sort is performed on the second index so that in groups of ordered faces where the first index is constant, the second index reference will be traversed in ascending order. The main benefits of these optimisations relate to improving both spatial and temporal locality as the cells referenced by the first index will be quasi contiguous in memory and therefore exploit the available memory bandwidth. Reducing the distance between the face end-points via RCMK leads to a better exploitation of the caches through reuse and also minimises Translation Look-aside Buffer (TLB) misses which are particularly expensive on many-core architectures due to their more simple core implementation which cannot handle a page walk efficiently. The final sort on the second index further improves memory performance since hardware prefetchers operate best on streams with ordered accesses whether in a forward or backward direction.

The grid renumbering is performed immediately after solver initialisation and across all MPI ranks. Each rank is in charge of reordering its local cells after which it traverses its local list of halos and relabels them with their new values. Consequently, this has a negligible effect on the overall application execution since the mesh is reordered only once at start-up and scales linearly with the available number of processors.

4.2. *Vectorisation*

The steady increase in vector register sizes and associated SIMD instruction set capabilities as evidenced in Section 3 have made vectorisation indispensable for extracting performance on modern hardware architectures. Although compilers have evolved over the years and are generally more capable in automatically vectorising computations, they can only do so when both safety and benefit are guaranteed. Complex constructs such as face-based loops inhibit vectorisation opportunities due to the existence of indirect accesses when scattering results back to the face end-points. Furthermore, even for constructs with linear and contiguous access patterns such as cell-based loops, compiler auto-vectorisation is not guaranteed due to various reasons such as pointer aliasing, inner function calls or conditional branching. In places where auto-vectorisation fails, the programmer can vectorise the kernel either via the utilisation of compiler directives, lower level intrinsics or in extreme cases, inline assembly. In this work, we have predominantly used the compiler directive approach available in OpenMP 4.0 [38]. The use of directives has been preferred over



(a) $\beta = 2808603$

(b) $\beta = 52319$

Fig. 3. Sparse matrix bandwidth (β) reduction via Reverse Cuthill Mckee on mesh 1 (3×10^6 elements) with figure (a) highlighting initial bandwidth and (b) the result of applying the renumbering.

other alternatives since they offer far greater flexibility and compatibility across SIMD architectures and compilers at a small cost in performance compared to lower level implementations. For cell-based loops, vectorisation was possible by the addition of omp simd directives either at loop level or at function declaration. This not only forces vectorisation but can also increase its efficiency by informing the compiler of auxiliary data attributes such as array alignment, variable scoping and vector lane length.

For face-based loops however, a rewrite is necessary in order to switch to a vector programming paradigm. Listing 2 presents the original implementation of the kernel computing the second order convective fluxes while Listing 3 shows the improved vector-friendly layout. In the vector-friendly implementation, the main loop over the faces is divided into three distinct stages which naturally map to the underlying gather–compute–scatter pattern. In essence, each main loop iteration will process a number of consecutive faces in parallel by exploiting the available vector lanes as defined by the VECLEN macro. The first nested loop gathers the cell data into local short vector arrays. Depending on the underlying architecture, the compiler will either generate SIMD gather instructions or serial load sequences if the architecture does not support such operations. Once all data is loaded into short vectors allocated on the stack, the second nested loop performs the computation. Similarly to the previous stage, computations are carried out in parallel on the available vector lanes. In AVX/AVX2, four faces would be processed at the same time or eight for IMCI and AVX-512 in double precision. Since all intermediate vector data is allocated on the stack as static arrays of sizes known at compile time as defined by the preprocessor macros, the compiler can easily generate efficient vector code as all dependencies have been eliminated. This includes loading contiguous data such as face geometrical properties at aligned addresses as expressed through the aligned clause in the directive. In finite volume applications, the computation stage involves a large amount of operations per pair of neighbouring cells making vectorisation mandatory for good performance. The last stage involves scattering back the residuals to the cells. This is done serially since generating vector code for this section would lead to incorrect results due to data dependencies as we process multiple successive faces in parallel. It is important to mention that the extra nested loops within the new implementation are unrolled automatically at compilation since they decay into single or multiple SIMD instructions therefore removing potential overheads as long as the iteration space (VECLEN) is equal or a relatively small multiple of the underlying vector lane width. The latter has been tested with VECLEN equal to twice or

Listing 2: Original implementation of face-based kernel computing second order inviscid fluxes

```
double ql[MAXNPDE], qr[MAXNPDE];
double f[MAXNPDE], fl[MAXNPDE];
double fr[MAXNPDE], fa[MAXNPDE];

 for( ic=ics;ic<ice;ic++ )
{
    for( ipde=0;ipde<npde;ipde++ )
   {
       ql[ipde]= q[ipde][ifq[0][ic+iv]];
       qr[ipde]= q[ipde][ifq[1][ic+iv]];
   }
  // Euler fluxes from the left  -- truncated
   fl[0]= ll1*rhol;
   fl[1]= fl[0]*ql[0]+wc[0][ic]*ql[4];
   fl[2]= fl[0]*ql[1]+wc[1][ic]*ql[4];
   // Euler fluxes from the right -- truncated
   fr[0]= lr1 *rhor;
   fr[1]= fr[0]*qr[0]+wc[0][ic]*qr[4];
   fr[2]= fr[0]*qr[1]+wc[1][ic]*qr[4];
   // Roe fluxes -- truncated
   fa[0]= dw1+dw3+dw4;
   fa[1]= dw1*qa[0]+dw2[0]+dw3*(qa[0]+ana[0])+dw4*(qa[0]-ana[0]);
   fa[2]= dw1*qa[1]+dw2[1]+dw3*(qa[1]+ana[1])+dw4*(qa[1]-ana[1]);
  // assemble fluxes
   for( ipde=0;ipde<npde;ipde++ )
  {
        f[ipde]= 0.5*(fr[ipde]+fl[ipde]-fa[ipde])*wc[3][ic];
  }
  // accumulate
   for( ipde=0;ipde<npde;ipde++ )
  {
        rhs[ipde][ifq[0][ic+iv]]-= f[ipde];
        rhs[ipde][ifq[1][ic+iv]]+= f[ipde];
  }
}
```

three times the architecture vector length, an optimisation called double or triple pumping. Promotion from scalar to short vectors allocated on the stack is done only for variables that appear in at least one of the three distinct stages (i.e. **f** representing the flux residual, computed in stage two and written back in stage three) or for variables that are used in conditionals in order to help the compiler to generate correct masking operations. In addition to rewriting all face-based kernels in the solver, we also aligned face and cell data structures to 32 or 64 byte boundary depending on SIMD architecture and padded the list of faces through the addition of redundant entries until we reached a size that is multiple of VECLEN. This removed the need for peel or remainder loops. Finally, we also performed various arithmetic optimisations whereby we replaced divisions with reciprocal multiplications where the divisors were geometrical variables since SIMD division operations are non-pipelined across the majority of architectures and suffer from very high latencies.

Listing 3: Vector friendly implementation of face-based kernel computing second order inviscid fluxes.

```
double ql[MAXNPDE][VECLEN], qr[MAXNPDE][VECLEN];
double f[MAXNPDE][VECLEN], wn[4][VECLEN];
double fl[MAXNPDE], fr[MAXNPDE], fa[MAXNPDE];

 for( ic=ics;ic<ice;ic+=VECLEN )
{
  // 1. gather unknowns from cells
   #pragma omp simd simdlen(VECLEN) safelen(VECLEN)
   for( iv=0;iv<VECLEN;iv++ )
  {
      for( ipde=0;ipde<npde;ipde++ )
     {
         ql[ipde][iv]= q[ipde][ifq[0][ic+iv]];
         qr[ipde][iv]= q[ipde][ifq[1][ic+iv]];
     }
  }
   // 2. compute flux
  #pragma omp simd simdlen(VECLEN) safelen(VECLEN)
  for(iv=0;iv<VECLEN;iv++ )
 {
     // fluxes from the left  -- truncated
     fl[0]= ll1*rhol;
     fl[1]= fl[0]*ql[0][iv]+wn[0][iv]*ql[4][iv];
     fl[2]= fl[0]*ql[1][iv]+wn[1][iv]*ql[4][iv];
     // fluxes from the right -- truncated
     fr[0]= lr1 *rhor;
     fr[1]= fr[0]*qr[0][iv]+wn[0][iv]*qr[4][iv];
     fr[2]= fr[0]*qr[1][iv]+wn[1][iv]*qr[4][iv];
     // Roe fluxes -- truncated
     fa[0]= dw1+dw3+dw4;
     fa[1]= dw1*qa[0]+dw2[0]+dw3*(qa[0]+ana[0])+dw4*(qa[0]-ana[0]);
     fa[2]= dw1*qa[1]+dw2[1]+dw3*(qa[1]+ana[1])+dw4*(qa[1]-ana[1]);
     // assemble fluxes -- truncated
     f[0][iv]= 0.5*(fr[0]+fl[0]-fa[0])*wn[3][iv];
     f[1][iv]= 0.5*(fr[1]+fl[1]-fa[1])*wn[3][iv];
     f[2][iv]= 0.5*(fr[2]+fl[2]-fa[2])*wn[3][iv];
 }
  // 3. accumulate and scatter to cells, serially
  for( iv=0;iv<VECLEN;iv++ )
 {
     for( ipde=0;ipde<npde;ipde++ )
    {
        rhs[ipde][ifq[0][ic+iv]]-= f[ipde][iv];
        rhs[ipde][ifq[1][ic+iv]]+= f[ipde][iv];
    }
 }
}
```

Listing 4: Preprocessing macros.

```
# define MAXNPDE 8

# if defined __MIC__
    # define VECLEN 8
    # define ALIGN  64
# elif defined __AVX512F__
    # define VECLEN 8
    # define ALIGN  64
# elif defined __AVX__
    # define VECLEN 4
    # define ALIGN 32
# elif defined __SSE3__
    # define VECLEN 2
    # define ALIGN 16
# endif
```

4.3. Face colouring

The data dependencies present when scattering back to the face end-points and which prevent vectorisation of scatters can be removed by further colouring and reordering the list of faces. Löhner et al. [39] presented two such algorithms which were originally developed for the purpose of avoiding memory contention but which can be extended to serve an additional purpose, that of allowing for the vectorisation of scatter primitives in unstructured field solvers. The first algorithm applies a simple colouring approach whereby it renumbers the list of faces such that groups of successive faces of size equal to the requested vector size i.e. VECLEN, have no dependencies at both cell end-points. The second algorithm builds upon the first with the addition of performing the renumbering while also attempting to reduce the distance in the second index. In our experience, both algorithms exhibited approximately the same performance characteristics after vectorisation even though the second algorithm did reduce the spread in distances between the second indices. It is important to note however that further colouring and reordering leads to a small increase in distances among first indices. Furthermore, these algorithms are unable to guarantee a complete dependency free order therefore extra care has to be taken for the last remaining set of faces which have to be further coloured as convergence is not guaranteed. Nevertheless, by applying the above algorithms in conjunction with the use of an additional OpenMP 4.0 simd directive for the remaining scatter loop, we were able to fully vectorise the entire gather, compute and scatter work flow across all face-based loops and therefore the entire application.

In terms of the computational cost of colouring and reordering the faces, this is performed only once, at solver initialisation and in a similar fashion to and after the grid renumbering. Consequently, this has a minuscule impact on application performance.

4.4. Array of structures

Our original implementation used Structures of Arrays (SoA) to store both face and cell data. For example, primitive variables (unknowns) were represented by a single structure holding pointers to individual long vectors as seen in Fig. 4.

As we process through faces in consecutive order within face-based loops, accessing face data such as normals, coordinates and frame speed is done via contiguous vector load operations which map effectively to the CPU vector registers. However, the accesses for cell data are irregular since cells are traversed non-consecutively. As a result, within each cell, variables have to be gathered from their respective position in the corresponding vector and into the vector registers. For SIMD architectures that do not support vector gather operations, the compiler will generate a large number of sequential loads in order to fill in the available SIMD registers. Similarly, scattering values such as residuals back to the corresponding cells translates into a very large number of sequential stores if scatter vector instructions are not available. On architectures with available gather and/or scatter support, the number of issued instructions is reduced by a factor equal to the underlying number of vector lanes. When compared to regular SIMD load and store operations, gather and scatter primitives are known to suffer from significantly higher latencies [29]. As a result, we have modified the data structures storing cell data to an Array of Structures (AoS) implementation whilst maintaining face data as SoA. In the AoS layout, cell variables such as the unknowns are grouped together in short arrays within a cell as seen in Fig. 5 rather than being stored in separate vectors. We further pad each short array to the underlying cache line size or the nearest multiple of the underlying SIMD register. Although AoS does not fully remove all gather and scatter operations, irregular load and stores are only executed once for each cell vector as subsequent successive elements are loaded automatically at cache line granularity. This improves locality and minimises cache misses although quantities still need to be transposed into their respective vector lane positions.

The transition from SoA to AoS was by no means a trivial endeavour as it required significant changes to the message passing interface, array access semantics and switching the order of all nested loop constructs that manipulated cell centred data. We therefore recommend that abstraction is implemented with regard to the layout in memory of such data structures so that a switch between different implementations can be performed at compile time which would be a useful design trait. This is important when considering that for structured codes, SoA or a hybrid of, is the best implementation [40] since cells are traversed consecutively following an i,j,k indexing system which therefore allows for vector load and store operations. Finally, in regard to unstructured grid applications, modifying cell data to the AoS layout has an impact

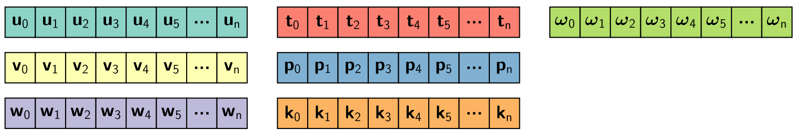

u_0 u_1 u_2 u_3 u_4 u_5 ⋯ u_n
v_0 v_1 v_2 v_3 v_4 v_5 ⋯ v_n
w_0 w_1 w_2 w_3 w_4 w_5 ⋯ w_n
t_0 t_1 t_2 t_3 t_4 t_5 ⋯ t_n
p_0 p_1 p_2 p_3 p_4 p_5 ⋯ p_n
k_0 k_1 k_2 k_3 k_4 k_5 ⋯ k_n
ω_0 ω_1 ω_2 ω_3 ω_4 ω_5 ⋯ ω_n

Fig. 4. Structures of Array (SoA) layout in memory of primitive variables (unknowns).

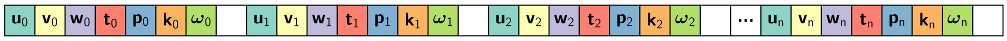

u_0 v_0 w_0 t_0 p_0 k_0 ω_0 u_1 v_1 w_1 t_1 p_1 k_1 ω_1 u_2 v_2 w_2 t_2 p_2 k_2 ω_2 ⋯ u_n v_n w_n t_n p_n k_n ω_n

Fig. 5. Array of Structures (AoS) layout in memory of primitive variables (unknowns) with extra padding represented by blank cell.

on cell-based kernels. Whereas previously with SoA, these kernels could exploit SIMD contiguous load store operations since cells were traversed in successive order, switching to AoS means that variables within a cell have to be loaded in vector registers and subsequently transposed into the SoA format which adds extra latency and decreases performance. It is therefore important to empirically evaluate the impact of this optimisation as performance will be lost in the cell-based loops although improvements are expected for the face-based constructs which are the main bottleneck in the application.

4.5. Gather and scatter optimisations

The AoS memory layout for cell data requires that gathers, arising from indirect addressing, are only executed once per structure and not for every element as it is the case with SoA. However, successive elements in AoS although contiguous, need to be transposed into the correct vector register and lane positions. Some compilers will recognise the access pattern and aim to generate sequences of transpose operations although the programmer's domain knowledge can improve on this significantly. Let us consider the primitive variables in AoS layout with padding in the last position at a given cell index *i*. Depending on the underlying SIMD architecture (i.e. 256-bit or 512-bit wide register), we can load all eight double precision values of the structure with either one or two aligned vector loads for each cell end-point. As vectorisation is applied across consecutive faces, the primitive variables of all gathered pairs of cells need to be re-arranged on the fly and packed into a SoA format. This process is highlighted more clearly in Fig. 6. In our work, we implement these transpose primitives using compiler intrinsics and target each individual SIMD architecture by using instructions that exhibit the lowest latency and highest degree of instruction parallelism. This is relevant since the SIMD implementations across the studied hardware platforms differ significantly and a generic solution would leave performance on the table. For example, the Knights Corner architecture provides swizzle operations which can perform on the fly data multiplexing prior to execution from the register. Consequently, some permutations and shuffles can be done with "zero" penalty whereas on all other architectures, these will be forwarded to an execution port (port 5 on multicores) which can create a bottleneck. Similarly, scatters for writing back residuals at each pair of cells are implemented by transposing back from SoA to AoS and utilising aligned vector stores which is possible since faces have been coloured. Similar work to the above has been presented by Pennycook et al. [35] for optimising the gather/scatter patterns in molecular dynamics applications although our solution extends this to cover additional AVX-512 SIMD architectures and applies them to an unstructured computational fluid dynamics application.

4.6. Array of structures structure of arrays

While the SoA format is ideal for mapping face data to SIMD registers, it requires that multiple memory streams are serviced in parallel for each vector of variables. Since prefetchers can only operate on a limited number of streams, maintaining a large number of them in flight wastes memory bandwidth and other valuable resources. Consequently, we replace the SoA layout with the hybrid Array of Structures Structure of Arrays (AoSSoA) format. Essentially, face attributes such as normals and coordinates are clubbed together in short vectors equal to the underlying vector register size or a multiple of it. As an example, let us consider the normals to the face in three dimensions: x, y, z on AVX/AVX2 with 256-bit wide registers. The hybrid AoSSoA implementation would map these to the layout see in Fig. 7. This increases locality since variables are stored at a close stride in memory and whereas before we required three independent streams to load normals in each dimension, the new layout merges this into a single stream whilst still allowing for aligned vector load and store operations.

4.7. Software prefetching

In structured codes with contiguous and regular access patterns, memory parallelism is exploited by the hardware transparently via the available hardware prefetchers across different cache levels. In unstructured codes, the irregular and indirect access patterns make prefetching more difficult to accomplish. In most cases, hardware prefetchers are unable to anticipate which data is needed in the upcoming iterations for face/edge-based loops due to the non-consecutive traversal of the cell/node end-points. This limitation can be overcome using the programmer's domain knowledge since the order in which cells are traversed is known at run-time and can be deduced from the associated connectivity arrays. In this study, we have inserted prefetching hints via compiler intrinsics for prefetching data either in L1, L2 or both cache levels depending on architecture and at an auto-tuned distance. The compiler hints were placed in the same kernels as the optimised intrinsics gather functions so that before we gather and transpose data for the cells of successive faces, we already issue prefetches for the next set of cell data a few iterations ahead. In this way, all platform specific routines can be bundled together in specific header files and abstracted away from the main application code. The prefetch distance parameter is important for obtaining any palpable performance improvement. If it is too small, the data will be brought into the higher level caches after it is required leading to unnecessary memory traffic. On the other hand, if this is too large, it will dislodge useful data therefore leading to cache evictions and subsequent misses. There have been a large amount of studies pertaining to the selection of an optimal prefetch distance such as the one conducted by Mowry et al. [41] or more recently, by Lee et al. [42]. In this work however, we have based our prefetching strategy on the work of Ainsworth et al. [43]. Their work demonstrated that when inserting software prefetches for indirect memory addressing, one has to not only prefetch the data of interest but also the index used to reference it. Furthermore, they have demonstrated that the distance of prefetches for the index values should be twice that of the referenced data since cache

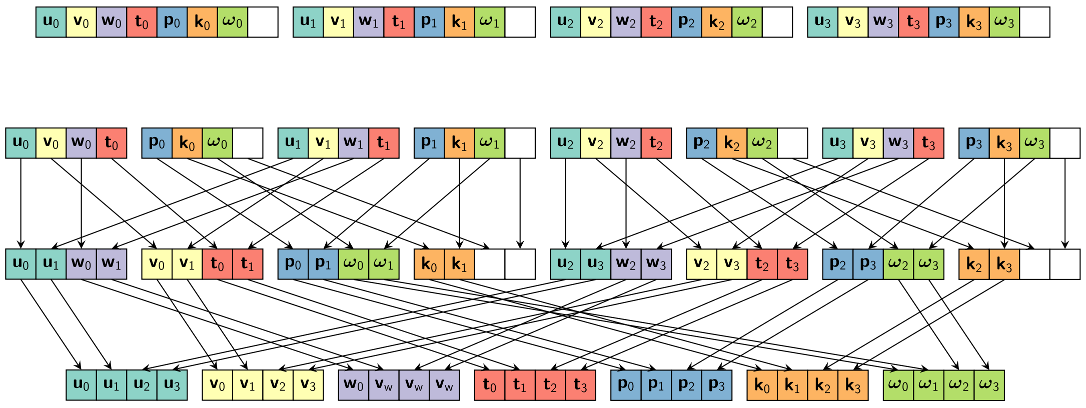

u_0 v_0 w_0 t_0 p_0 k_0 ω_0
u_1 v_1 w_1 t_1 p_1 k_1 ω_1
u_2 v_2 w_2 t_2 p_2 k_2 ω_2
u_3 v_3 w_3 t_3 p_3 k_3 ω_3

Fig. 6. On the fly in-register conversion from AoS to SoA of unknowns at a given cell and including padding on AVX/AVX2 256-bit SIMD registers. Variables in each short array are loaded using aligned vector load/store operations in registers and then transposed via shuffles and cross-lane permutations.

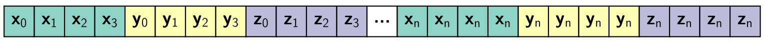

x_0 x_1 x_2 x_3 y_0 y_1 y_2 y_3 z_0 z_1 z_2 z_3 ⋯ x_n x_n x_n x_n y_n y_n y_n y_n z_n z_n z_n z_n

Fig. 7. Array of Structures Structure of Arrays (AoSSoA) layout in memory of three dimensional coordinates of normals to the interfaces.

misses in loading the values for the reference array can offset any potential improvements from prefetching the actual data. Building upon these observations, we have created an auto-tuning script with the help of one of the authors in [43] whereby we telescoped through various ranges of both index prefetch and data prefetch distances whilst maintaining this ratio. The above is mainly applicable to software prefetching for exploiting the indirect access patterns in face-wise kernels. In regard to cell/vertex-based loops, the hardware prefetchers of all but the KNC architecture performed well and software prefetching saw little benefit. Finally, if software prefetching is implemented explicitly as described above, compiler prefetching should be switched off in order to avoid for competing prefetch instructions to be issued which can degrade performance.

4.8. Multi-threading

Thread-level parallelism has been exploited in our application via the utilisation of OpenMP directives and specifically targets the Xeon Phi manycore architectures. For the multicore CPUs, multi-threading in conjunction with MPI did not bring forth any performance improvement and was therefore abandoned. However, on the Intel Xeon Phi architecture, running more than one thread context per physical core is highly encouraged especially for KNC where it can hide memory latencies due to the in-order core execution engine. Although exposing parallelism in loops over the degrees of freedom is trivial, exposing thread parallelism in face loops is more challenging and requires colour concurrency. There are a number of approaches which can be utilised with respect to the latter [44]. In this work, we used the available colouring algorithms to reorder the list of faces such that the number of dependency free faces is not only a multiple of the underlying vector register size but can also be divided so that each active thread per core can process an equal amount of vector iterations. We then schedule the iteration space as static at a chunk equal to one vector iteration whereby each thread executes vector iterations in a round robin fashion. The main rationale behind this approach is that it guarantees correctness and integrates well with MPI whereby each rank is pinned to a physical core with subsequent threads spawned within the same core domain. This maintains data locality and affinity to the underlying cache hierarchy, reduces traffic caused from the protocols in charge of cache coherence and mitigates against the risk of false sharing. Most importantly, this approach hides latency best especially for KNC since it guarantees that each thread will access different cell data due to the colouring and every miss and stall in L1 can be circumvented by switching between active threads that have data available.

5. Results and discussions

5.1. Effects of optimisations

Figs. 8–10 present the impact that each optimisation has on the performance of both classes of computational kernels and overall solver across every processor architecture. The results were obtained by executing the application on one MPI rank in order to discount any effects of the message passing implementation. The KNL results presented in this section were obtained with the MCDRAM configured as cache. We discuss these in more detail in the following sections.

5.1.1. Grid renumbering

Renumbering the grid via the Reverse Cuthill Mckee algorithm led to minor improvements in performance for the face-based kernel (flux computations) when compared directly to the baseline implementation. However, when we account for all other optimisations, the difference in performance between face-based kernels with and without grid renumbering grows to as much as 40%. This is due to the fact that once we efficiently vectorise these types of kernels, their performance is bound by the memory bandwidth. As such, performing grid renumbering in unstructured grid applications is mandatory for improving the performance of face-based or edge-based kernels especially when they are already tuned for exploiting the arithmetic units.

5.1.2. Vectorisation

Vectorisation sees the largest increases in performance across all architectures and computational patterns. For the face-based kernel, we obtain improvements between 2-5X on the multicore CPUs and 2-3X on the KNC and KNL processors. For cell-based kernels, vectorisation also provided the highest performance impact compared to all other optimisations. Furthermore, all of these translate to an overall application speed-up between 2-3X relative to the original baseline.

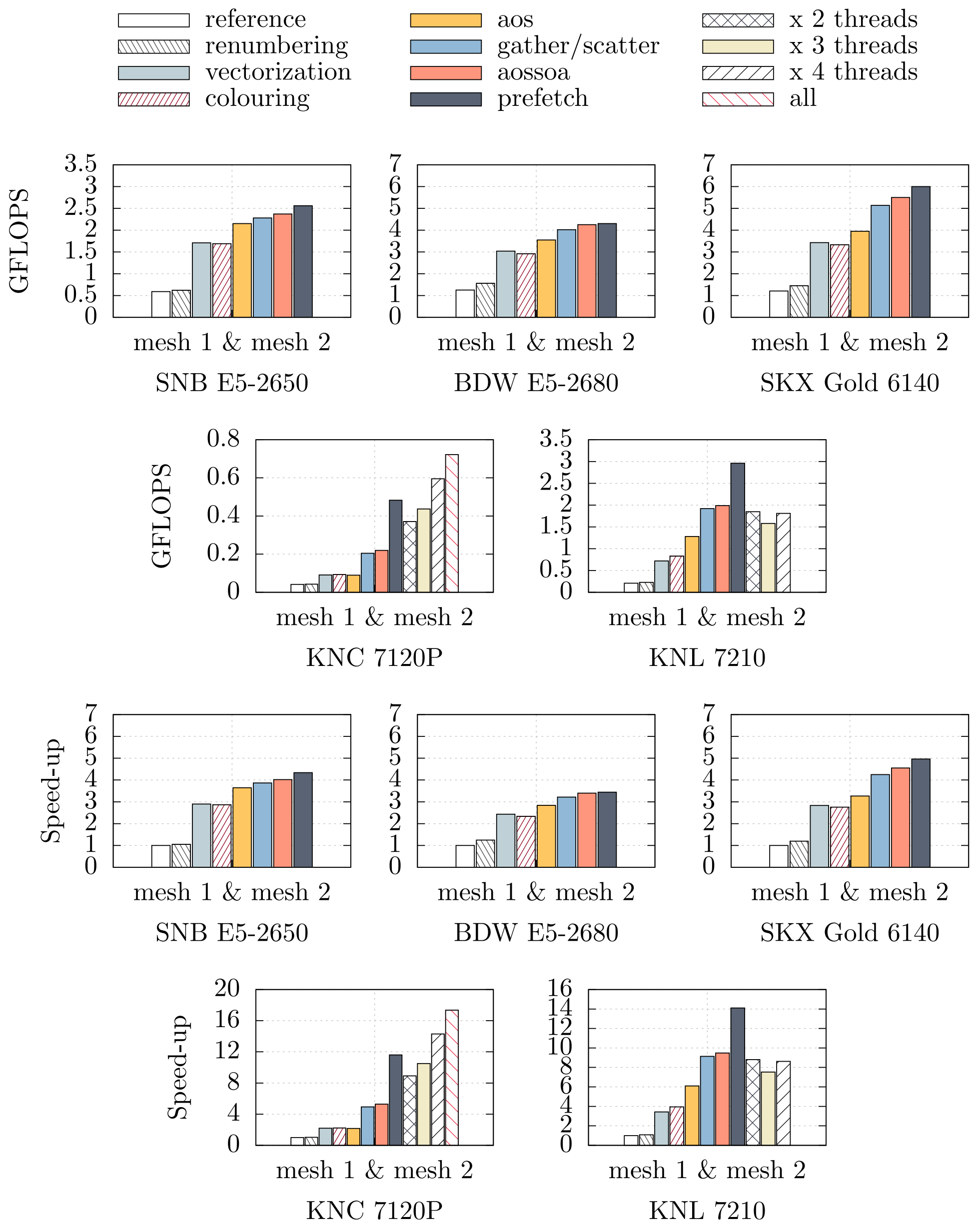


Fig. 8. Effects of optimisations on the performance of face-based kernels (second order inviscid flux computations) measured as GFLOPS and Speed-up relative to the reference version. Results are averaged across `mesh 1` and `mesh 2` and obtained by running the application on 1 MPI rank (serially). Results for `all` are based on running the application with both software prefetching enabled and utilising all of the 4 available hyperthreads in the KNC 7120P core.

5.1.3. Face colouring

The colouring of faces for enabling vectorised scatters to the face end-points does not yield any benefits on the multicore CPUs where it is in fact detrimental to performance. In contrast, this is beneficial on the manycore processors and leads to marginal speed-ups although not higher than 10%. This difference might stem from the fact that non-vectorised serial stores are significantly slower on the manycore processors than the SIMD scatter instruction. On the multicore CPUs, the Sandy Bridge and Broadwell architectures implement AVX/AVX2 which do not provide vector scatter functionality. Scatters are performed as a series of vector inserts and stores that have a higher cumulative latency than their serial store counterpart. On Skylake, the AVX-512F implementation does provide support for vector scatters but we attribute the small decrease in performance to the increase in the distance across the first and second index due to reordering which leads to an increase in cache misses since the distance among each cell index has to be higher than eight which is also equal to the number of variables that can be stored in a 64-byte cache line. Nevertheless, colouring is important further down the line once we implement additional optimisations since it exposes parallelism in writing back values across all face-based kernels and allows for the introduction of multi-threading.

5.1.4. Array of structures

The conversion of cell centred data structures from SoA to AoS has a positive impact on the performance of face-based loops on the multicore CPUs where we obtain as much as 25% speed-up on top of previous optimisations. On KNL, the AoS layout sees significant speed-ups of as much as 50% for the face-based kernel. This is opposite to KNC where we see no improvement. Furthermore, modifying the layout in memory of cell-centred data structures has a negative effect in cell-based loops. This is important when bearing in mind that the overall application performance decreases after we convert cell-centred data structures from SoA to AoS. However, as with colouring, this optimisation is required for

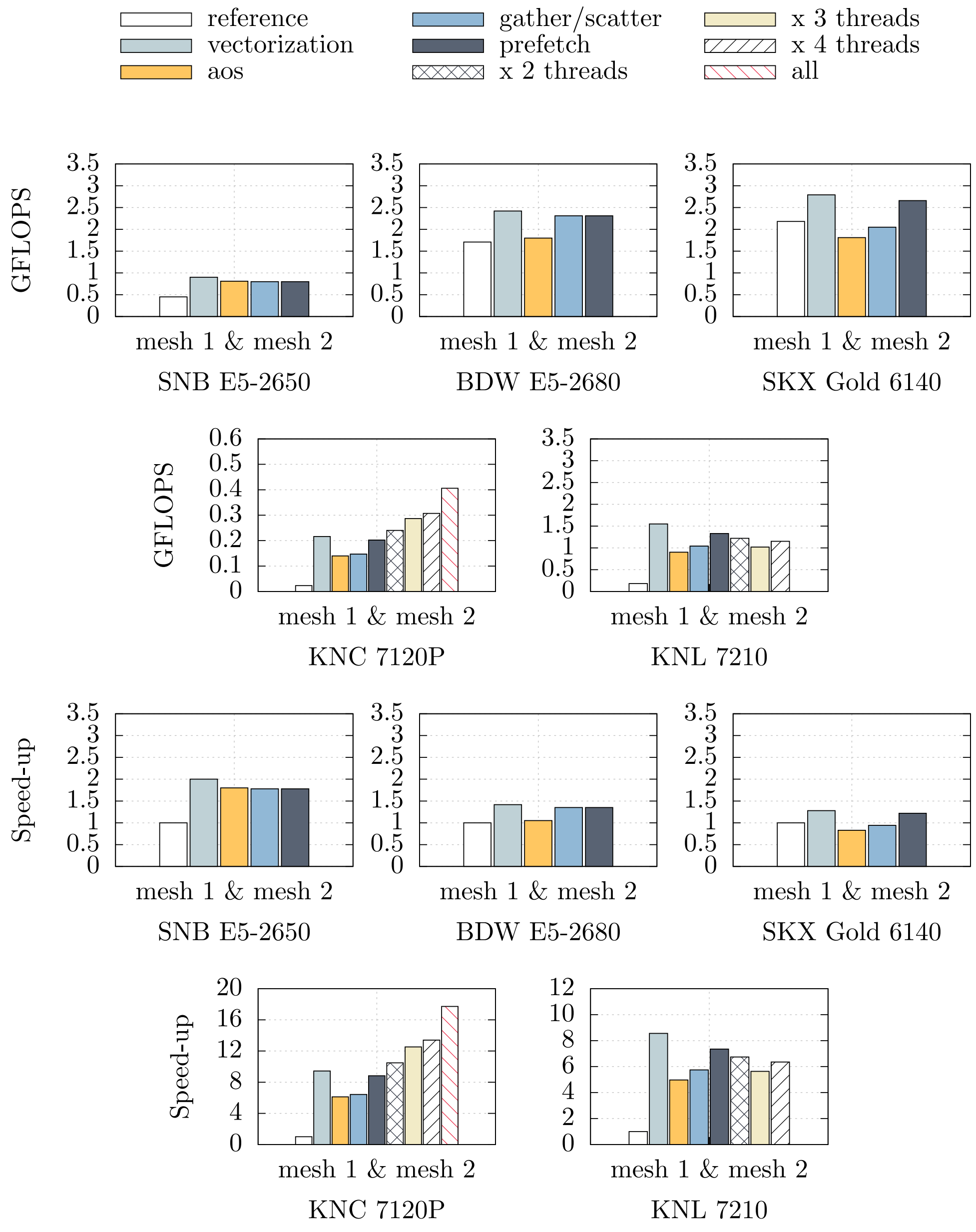


Fig. 9. Effects of optimisations on the performance of cell-based kernels (linearised update to primitive variables) measured as GFLOPS and Speed-up relative to the reference version. Results are averaged across `mesh 1` and `mesh 2` and obtained by running the application on 1 MPI rank (serially). Results for `all` are based on running the application with both software prefetching enabled and utilising all of the 4 available hyperthreads in the KNC 7120P core.

further optimisations which will exploit the additional data locality available within each vector of cell-centred variables.

5.1.5. *Gather and scatter*

The on the fly transposition from AoS to SoA using specialised intrinsics-based functions improves performance across all architectures. The impact is significantly higher on processors with wider vector registers such as SKX, KNC and KNL. This is due to the fact that consecutive variables are gathered via a single aligned vector load and subsequently transposed and arranged into the correct lane via architectural specific permute instructions and the fact that the number of SIMD instructions required for the transposition scales logarithmically compared to the serial implementation. It is worth mentioning that on KNL and Skylake with AVX-512F, our tests have shown that permute instructions using a single source operand (`vpermpd`) perform better compared to the newly available two source operand instructions (`vpermt2pd`) since they exhibit superior throughput. On SNB and BDW with AVX/AVX2, the best version is the one based on a combination of interleave operations (`vunpckhpd`/`vunpacklpd`) and 128-bit wide permutations (`vperm2f128`) even when we compare it with other implementations that performed the first step of the shuffle on the load ports with the utilisation of the `vinsertf` instruction on a memory operand. This would indicate that the bottleneck on these architectures is not port 5 pressure where all of the interleave and shuffle operations are executed. In cell-wise loops, performing the conversion from AoS to SoA and back on the fly via compiler intrinsics sees a moderate improvement in performance for our candidate kernel and in some cases, it offsets the performance dropped from switching to AoS from SoA in the first place. More importantly, the implementation of these highly tuned primitives leads to whole application speed-up on all processors between a few percentages on the multicore CPUs where it amends for the loss in performance due to the switch to AoS in cell-based loops and to as much as 50% on the Xeon Phi processors on the previous optimisations.

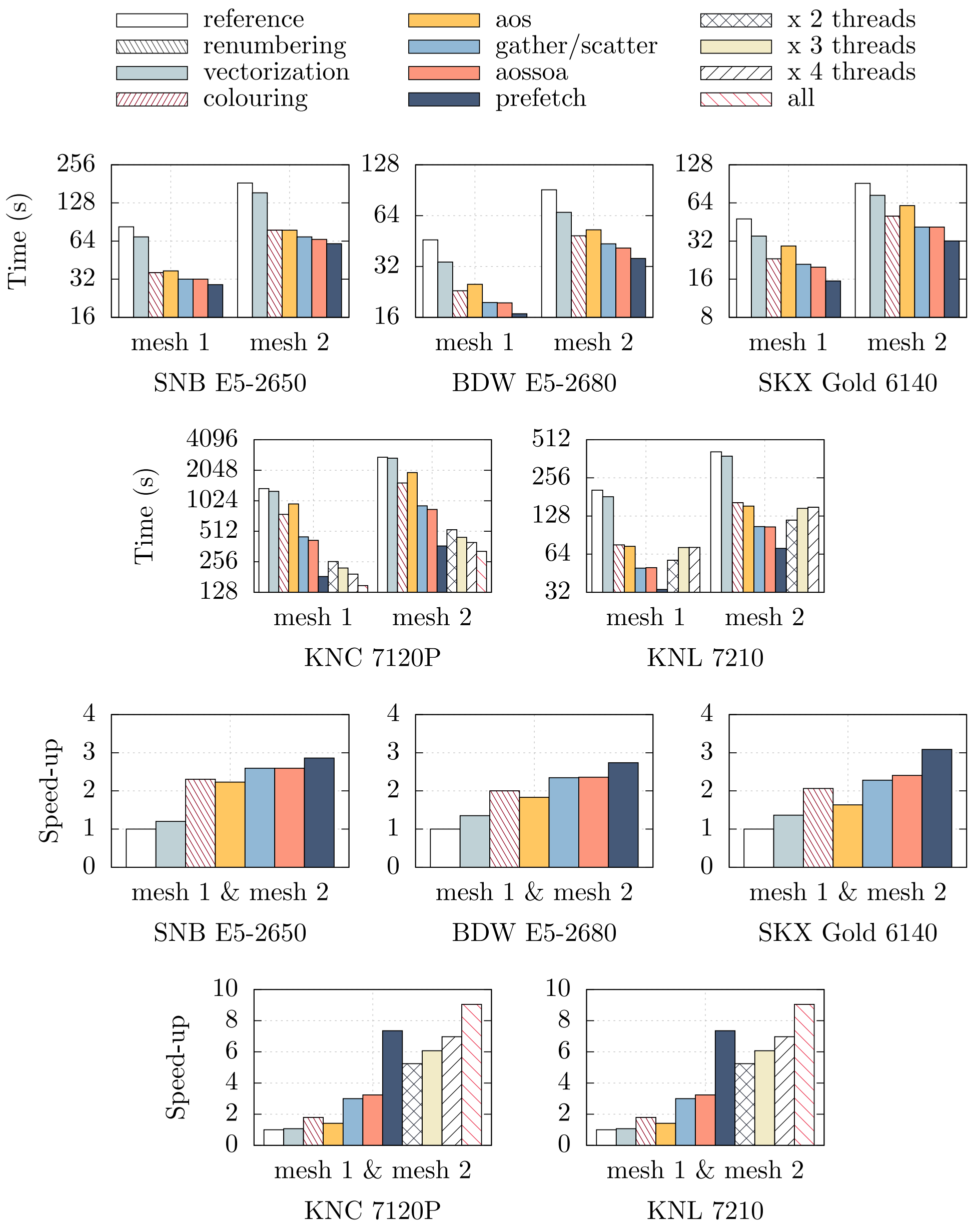


Fig. 10. Effects of optimisations on the whole solver measured as average time per Newton–Jacobi iteration and Speed-up relative to the reference version and averaged across `mesh 1` and `mesh 2`. Results for `all` are based on running the application with both software prefetching enabled and utilising all of the 4 available hyperthreads in the KNC 7120P core.

5.1.6. Array of structures structure of arrays

The conversion of face data structures to the hybrid AoSSoA layout concerns only face-based loops and yields minor improvements. However, these are not significant enough to warrant its usefulness as it did not generate any substantial speed-ups on any architectures and as such, the effort of implementing such hybrid memory layout within a code of representative size might not be warranted.

5.1.7. Software prefetching

Software prefetching exhibits substantial speed-ups on the KNC due to the out of order core design and on the SKX and KNL architecture due to the larger L2 cache. On KNC, the best performing strategy was to issue prefetch instructions at a distance of 32 and 64 for the indices in L1 and L2 respectively and half of that for the actual data as aforementioned in the methodology. For KNL and SKX, the best strategy was to only issue prefetch instructions for the L2 cache at a distance of 32 for the index and therefore 16 for the data as any prefetch instructions for L1 were in fact detrimental to performance. We suspect this is due to the small size of the L1 cache (32 KB) which remained unchanged from SNB and BDW while the L2 was increased by a factor of four. Furthermore, in the case of KNL, this approach bears fruit since the L1 prefetcher can operate on irregular streams however the L2 is not hence why issuing prefetch instructions for L2 in software is advisable. On SNB and BDW, the advantage of software prefetching is minimal and virtually non existent. We believe this is due to the smaller L2 caches on these architectures as face-based loops tend to exhibit a large amount of work per iteration and access a large number of data structures from memory. As such, we are probably running out of space in the caches and the prefetch instructions actually replace useful cache lines. In our cell-based loop example, software

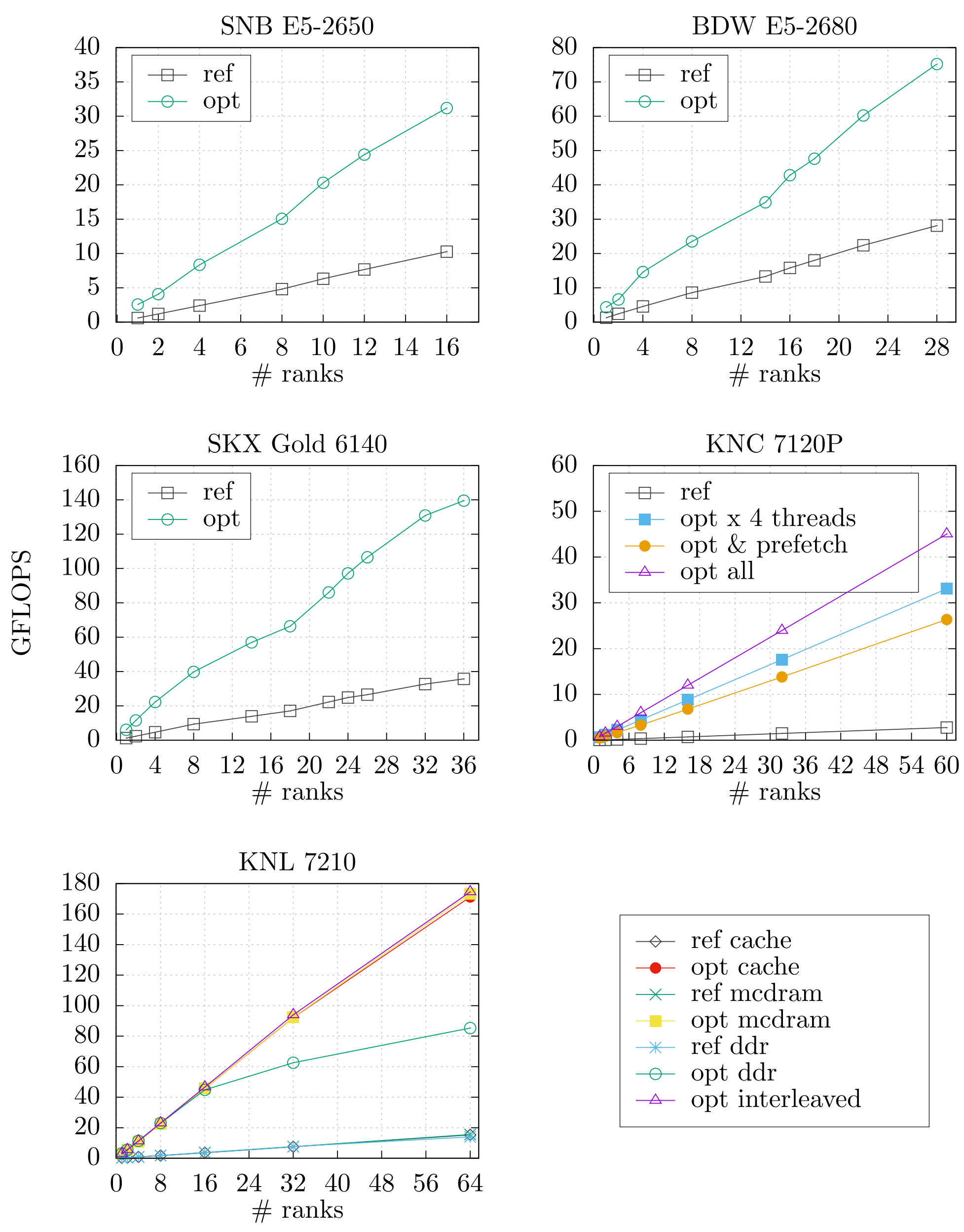


Fig. 11. Performance strong scaling of the face-based kernel (flux computations) averaged across `mesh 1` and `mesh 2` except for the KNC 7120P where results are only presented for `mesh 1`.

prefetches have no effect on SNB and BDW but provide benefits on SKX, KNC and KNL, similar to face-based loops. This is not surprising for KNC where the lack of an L1 hardware prefetcher mandates the utilisation of prefetch instructions either in software or through the compiler even for regular unit stride access patterns however, it was unexpected for the other two architectures. The prefetch distances for cell-based kernels with linear unit stride loads have been selected via auto-tuning as well. For L1, the best linear prefetch distance was deemed to be 32 and 64 for the L2 cache respectively across KNC, KNL and SKX.

5.1.8. Multi-threading

Multithreading via OpenMP on Knights Corner yields significant speedups with performance increasing almost linearly with the number of threads. The reason for this lies again in the in-order nature of the KNC core where a missed load in the L1 cache leads to a complete pipeline stall. Having more than one thread in flight can help minimise memory latency as context can be switched to a thread for which data is available. Since both software prefetching and threading on KNC have the same purpose, in the KNC results, the runs with 2, 3, 4 threads per MPI rank and physical core have been done with software prefetching disabled while for the software prefetching $\times$ 4 threads analysis, both software prefetching and four active threads were utilised per MPI rank. The results indicate that best performance on KNC is obtained when both threading and prefetching are enabled however, if required to choose between the two, careful prefetching based on an auto-tuning approach such as ours yields better performance than just multithreading. This is an important aspect to take into account since the implementation of software prefetching is more trivial in nature and less error prone than exposing another level of parallelism in the application for multithreading within an MPI rank. Surprisingly, on KNL, multithreading was actually detrimental to performance. We believe this to be due to the fact that the core

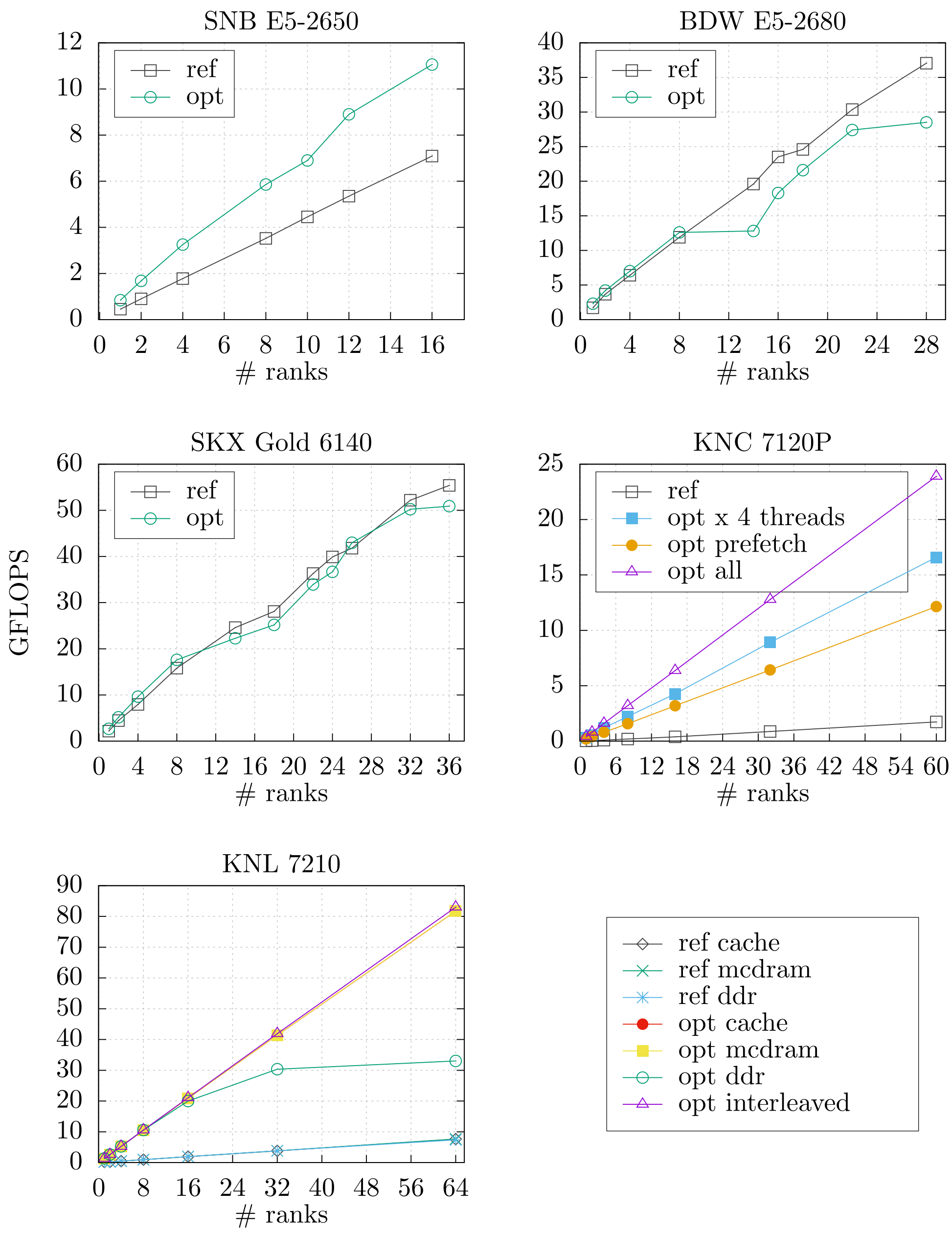


Fig. 12. Performance strong scaling of the cell-based kernel (variable update) averaged across `mesh 1` and `mesh 2` except for the KNC 7120P where results are only presented for `mesh 1`.

architecture of KNL is out of order and therefore capable of utilising all core resources with a single thread per core. Running more than one thread per core on KNL leads to available core resources being divided among the in-flight threads. We obtain similar results when running with 3 and 4 threads per core due to the fact that resources are divided in the same manner for both configurations.

For cell-based kernels, 2, 3 and 4 threads per core improves performance significantly on KNC although, similar to the face-based kernels, it degrades it on KNL. The above also holds true for the overall execution time of the solver.

5.2. Performance scaling across a node

We present performance scaling across the compute nodes for each computational kernel and the whole solver. For the multicore-based two-socket compute nodes, we compare baseline implementation across both mesh sizes with the best optimised version.

On the KNC co-processor, we perform strong scaling studies of three different versions besides the baseline in order to study how they scale across the entire device. These are the optimised version with software prefetching enabled and no multithreading, optimised version with four threads per MPI rank and no software prefetching and lastly, optimised version with both software prefetching and four threads per rank. On KNL, we study the difference in performance between baseline and optimised versions with the additional exploration of the memory configurations that are available on this architecture. In cache mode (`cache`), the MCDRAM is configured as a direct mapped cache and acts as a memory side buffer transparent to the user. In flat mode (`mcdram`), all memory allocations are performed explicitly in MCDRAM using the `numactl` utility. For the larger `mesh 2`, approximately 10% of the allocations were re-routed to DDR memory after utilising all of the available 16 GB in MCDRAM. In DDR mode (`ddr`), the allocation of memory has been done only in DDR memory and

Table 2
Comparison between time to solution update measured in seconds of baseline and best optimised implementation on multicore CPUs.

MPI ranks	mesh 1			mesh 2		
	reference	optimised	speed-up	reference	optimised	speed-up
SNB E5-2650						
1	83.0	29.0	2.8	184.0	61.0	3.0
2	42.0	15.3	2.7	86.8	29.7	2.9
4	23.0	8.9	2.5	44.7	15.1	2.9
8	13.6	5.8	2.3	24.7	9.0	2.7
10	10.7	4.5	2.3	19.5	7.5	2.6
12	9.4	3.7	2.5	18.3	7.2	2.6
16	7.4	2.8	2.6	14.4	5.5	2.6
BDW E5-2680						
1	46.5	16.8	2.7	91.0	35.7	2.5
2	24.9	9.1	2.7	47.4	20.2	2.3
4	14.6	5.5	2.6	26.5	11.2	2.3
8	8.4	3.8	2.2	16.1	7.4	2.1
14	6.4	3.0	2.1	10.4	5.9	1.7
16	5.8	2.5	2.3	9.1	5.2	1.7
18	5.3	2.4	2.2	8.8	4.6	1.9
22	4.6	1.9	2.4	7.9	3.8	2.0
28	3.8	1.4	2.7	7.6	2.8	2.7
SKX Gold 6140						
1	47.8	15.5	3.0	91.3	32.1	2.8
2	25.5	8.0	3.1	49.9	16.4	3.0
4	13.4	4.5	2.9	28.1	8.8	3.1
8	7.4	2.9	2.5	15.5	5.5	2.8
14	5.1	2.2	2.3	10.7	4.2	2.5
18	4.7	2.1	2.2	9.0	3.7	2.4
22	3.9	1.6	2.4	7.8	3.1	2.4
24	3.8	1.5	2.5	7.2	2.8	2.5
26	3.5	1.4	2.5	6.6	2.6	2.5
32	2.9	1.2	2.4	6.0	2.2	2.7
36	2.8	1.0	2.8	5.3	1.9	2.7

MCDRAM has not been utilised at all. The last option was a hybrid evaluation (`interleaved`) where we allocated the storage of cell-centred data structures in MCDRAM whilst face data was allocated in DDR together with MPI buffers in order to interleave access and therefore exploit the bandwidth of both memory systems. This has been implemented in the code using the libmemkind interface.

5.2.1. *Face-based loops*

Fig. 11 presents strong scaling results of the face-based kernel (flux computations) over the compute nodes.

On the multicore CPUs, the difference between baseline and optimised versions remains approximately unchanged from the single core runs as we scale across the entire node. More specifically, at full concurrency, the optimised face-based kernel is 3X faster on Sandy Bridge (16 cores), 2.7X on Broadwell (28 cores) and 3.9X on Skylake (36 cores).

We observe the same characteristics on the manycore processors where at full concurrency, the difference between baseline and best optimised version for the face-based kernel range is 16.3X on Knights Corner (60 cores × 4 threads) and 11.4X on Knights Landing (64 cores). Since face-based loops involve a relatively large number of floating point operations per pair of evaluated cells, these kernels tend to exhibit significant speed-ups due to our optimisations and scale almost linearly with the number of cores.

On KNC, for the flux computation kernel, the best version as we scale across all physical cores is the one with software prefetching and 4 threads per core. This is followed by the version with no software prefetching and threading while software prefetching only is last by a significant margin.

In regard to the various memory modes available on KNL, we can observe that if we do not exploit the MCDRAM either as a cache or explicitly as a memory side buffer, running out of only DDR is as much as 3X worse than the other versions as we scale past 16 physical cores. This is to be expected since the difference in bandwidth between MCDRAM and DDR is more than a factor of four as measured with STREAM. The best performing version is our interleaved implementation where we utilise MCDRAM and DDR together allocating specific data structures across both.

5.2.2. *Cell-based loops*

For our cell-based kernel (Fig. 12), we observe how the baseline implementation slightly outperforms our optimised version on SKX and BDW as we strong scale across the node. We attribute this to the extra latency incurred by the transposition from AoS to SoA. The optimised implementation outperforms the baseline on Sandy Bridge and the Xeon Phi processors since serial instructions are significantly slower on the manycore architectures than their vector counterparts and the Sandy Bridge architecture has fewer cores per node and can therefore perform more work per rank to hide the effects of the transpositions. As cell-based kernels are memory bound with low flop per byte ratios, they tend to scale with the available memory bandwidth and as such, optimisations focused on improving floating-point performance such as vectorisation fail to give any benefit as we increase the number of cores and saturate all memory channels. We can certainly observe the negative effect that switching from SoA to AoS had on these kernels due to the overhead transpositions which become a bottleneck on the newer multicores.

On KNC, as with the face-based kernels, the best version by a large margin was the one running with 4 threads per MPI rank and software prefetching. On KNL, we can see how DDR scales better in cell-based kernels compared to face-based loops due to the regular unit stride access pattern however, it is almost 3X worse than the other alternatives.

Table 3
Comparison between time to solution update measured in seconds of baseline and optimised versions with 4 threads and no software prefetching, software prefetching and no threading and both software prefetching and 4 threads on the Intel Xeon Phi Knights Corner coprocessor. The speed-up is calculated based on baseline results and the timings of the best optimisation (`optimised prefetch x 4 threads`). On this architecture, we could only perform experimental runs on `mesh 1` at full concurrency due to the 16 GB memory limit.

MPI ranks	mesh 1				
	reference	optimised × 4 threads	optimised prefetch	optimised prefetch × 4 threads	speed-up
KNC 7120P					
1	1349.0	193.6	183.4	149.1	9.0
2	703.0	111.9	89.7	78.9	8.9
4	385.8	56.7	48.2	40.8	9.4
8	192.9	34.4	28.5	25.1	7.6
16	110.1	17.5	14.3	13.6	8.0
32	62.5	10.0	8.4	7.9	7.9
60	43.8	6.0	5.0	5.0	8.7

Table 4
Comparison between time to solution update measured in seconds of baseline and best optimised implementation on the Intel Xeon Phi Knights Landing 7210 processor and across different memory modes.

MPI ranks	mesh 1			mesh 2		
	reference	optimised	speed-up	reference	optimised	speed-up
KNL 7210 Cache						
1	205.0	33.7	6.0	411.0	71.1	5.7
2	116.0	17.8	6.5	238.4	37.8	6.3
4	57.1	9.4	6.0	120.4	19.3	6.2
8	29.0	5.3	5.4	63.3	10.0	6.3
16	16.2	2.8	5.7	32.1	5.3	6.0
32	9.1	1.6	5.6	18.23	3.1	5.8
64	5.9	1.0	5.9	10.5	2.0	5.2
KNL 7210 MCDRAM						
1	204.5	33.2	6.1	405.1	69.1	5.8
2	115.0	17.8	6.4	230.9	37.4	6.1
4	57.3	9.4	6.0	119.4	18.9	6.3
8	28.9	5.3	5.4	61.6	9.8	6.2
16	16.0	2.8	5.7	31.2	5.3	5.8
32	8.9	1.6	5.5	17.9	3.0	5.9
64	5.7	1.0	5.7	10.3	1.9	5.4
KNL 7210 DDR						
1	190.7	31.9	5.9	377.1	66.2	5.6
2	106.8	17.1	6.2	213.6	35.9	5.9
4	53.1	9.1	5.8	111.2	18.3	6.0
8	27.1	5.2	5.2	57.3	9.7	5.9
16	15.4	3.0	5.1	29.6	5.7	5.1
32	9.2	2.6	3.5	18.1	4.9	3.6
64	7.6	2.6	2.9	13.6	4.9	2.7
KNL 7210 DDR+MCDRAM						
1	–	30.2	–	–	62.5	–
2	–	17.5	–	–	36.4	–
4	–	9.1	–	–	18.7	–
8	–	5.3	–	–	9.6	–
16	–	2.8	–	–	5.1	–
32	–	1.6	–	–	3.0	–
64	–	0.9	–	–	1.8	–

5.2.3. *Full application*

Results for whole application strong scaling can be seen in Tables 2–4. At full concurrency and on both mesh sizes, our best optimised version is between 2.8 and 3X faster on the multicore CPUs, 8.6X on KNC and 5.6X on the best performing configuration of KNL in quadrant and flat mode with MCDRAM and DDR accesses interleaved. As expected, running only with DDR is 2.5X slower than the other modes that involve the utilisation of MCDRAM. Our approach of utilising both memories explicitly is the fastest version although by a very small percentage. On KNC, in terms of time per solution update, running with 4 threads per MPI rank and software prefetching or only performing software prefetching exhibit the same performance as we scale across the whole coprocessor and saturate the available memory.

6. Conclusions

We have presented a number of optimisations useful for improving the performance of unstructured finite volume CFD codes on a range of multicore and manycore architectures that form the backbone of current and likely future HPC systems. We have discussed their implementation in two distinct classes of computational kernels which form the foundation of unstructured finite volume CFD codes: face-based and cell-based loops and measured their impact on the performance of the whole application across every architecture. The importance of making efficient use of the available vector units in both types of computational kernels has been presented and the optimisations required to vectorise face-based kernels despite their underlying indirect and irregular access patterns. We have demonstrated the advantage of the AoS memory layout for cell-centred variables in kernels with indirect and irregular access patterns due to their more efficient utilisation of the cache hierarchy and presented hand tuned architectural specific compiler intrinsics for performing in-register AoS to SoA transpositions in order to allow for increased performance of gather and scatter operations. We have discussed the utility of auto-tuning for finding parameters such as prefetch distances across all processors and the large impact these can have on performance. Moreover, we have demonstrated that software prefetching is highly recommended on architectures with large L2 caches and on the Knights Corner co-processor due to the in-order core design. We have also established and confirmed the previous work of Ainsworth et al. [43] that issuing prefetch instructions for the indices at twice the distance of the data while finding the former through an auto-tuner is a very good recipe for best performance across a wide range of processors.

With regard to the configurable memory modes available on the Intel Xeon Phi Knights Landing processor, we have studied their performance and have introduced a third option in which we utilise both MCDRAM and DDR4 interfaces simultaneously. The main take away point is that the exploitation of MCDRAM, whether transparently in cache mode or explicitly via libraries or utilities such as `numactl`, is imperative for best performance on architectures that integrate such high-bandwidth memory packages.

Although using multiple threads per MPI rank leads to very good performance on the Knights Corner architecture, the opposite is true on the Knights Landing processor due to the improvements to the core design and out of order execution capabilities.

Finally, our optimisations efforts led to full application speed-ups between 2.8 and 3X on the multicore CPUs and 5-8X on the manycore Xeon Phi processors at double precision and across two different mesh sizes.

Acknowledgements

This work has been supported by the Engineering and Physical Sciences Research Council (EPSRC) and Rolls-Royce plc through the Industrial CASE Award 13220161. The authors are particularly indebted to Timothy Jones at the University of Cambridge for discussions and help with software prefetching, David Power and Konstantinos Mouzakitis at Boston Limited for access to Broadwell and Knights Corner nodes, Dheevatsa Mudigere and Michael Klemm of Intel Corporation for discussions and advice on the Skylake and Knights Landing architectures and to Adrian Jackson at the University of Edinburgh and Harvey Richardson of Cray Inc for access and help with the ARCHER Knights Landing development system.

References

[1] M.B. Giles, I. Reguly, Philos. Trans. R. Soc. Lond. A Math. Phys. Eng. Sci. 372 (2022) (2014). http://dx.doi.org/10.1098/rsta.2013.0319.
[2] W.K. Anderson, W.D. Gropp, D.K. Kaushik, D.E. Keyes, B.F. Smith, Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, SC '99, ACM, New York, NY, USA, 1999. http://dx.doi.org/10.1145/331532.331600.
[3] FUN3D, 2017. https://fun3d.larc.nasa.gov/. (Accessed 31 August 2017).
[4] ACM Gordon Bell Prize, 2017. http://awards.acm.org/bell. (Accessed 31 August 2017).
[5] E. Cuthill, J. McKee, Proceedings of the 1969 24th National Conference, ACM '69, ACM, New York, NY, USA, 1969, pp. 157–172.
[6] W.D. Gropp, D.K. Kaushik, D.E. Keyes, B.F. Smith, Parallel Comput. 27 (4) (2001) 337–362. http://dx.doi.org/10.1016/S0167-8191(00)00075-2. Parallel computing in aerospace.
[7] D. Mudigere, S. Sridharan, A. Deshpande, J. Park, A. Heinecke, M. Smelyanskiy, B. Kaul, P. Dubey, D. Kaushik, D. Keyes, 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 723–732 http://dx.doi.org/10.1109/IPDPS.2015.114.
[8] M.A.A. Farhan, D.K. Kaushik, D.E. Keyes, Parallel Comput. 59 (2016) 97–118. http://dx.doi.org/10.1016/j.parco.2016.06.001. Theory and Practice of Irregular Applications.
[9] A.C. Duffy, D.P. Hammond, E.J. Nielsen, Production Level CFD Code Acceleration for Hybrid Many-Core Architectures, Technical Report NASA/TM-2012-217770, National Aeronautics and Space Administration, 2012.
[10] T.D. Economon, D. Mudigere, G. Bansal, A. Heinecke, F. Palacios, J. Park, M. Smelyanskiy, J.J. Alonso, P. Dubey, Comput. & Fluids 129 (2016) 146–158. http://dx.doi.org/10.1016/j.compfluid.2016.02.003.
[11] SU2, the Open-Source CFD Code, 2017. http://su2.stanford.edu. (Accessed 31 August 2017).
[12] M.B. Giles, G.R. Mudalige, Z. Sharif, G. Markall, P.H. Kelly, SIGMETRICS Perform. Eval. Rev. 38 (4) (2011) 9–15. http://dx.doi.org/10.1145/1964218.1964221.
[13] G.R. Mudalige, M.B. Giles, I. Reguly, C. Bertolli, P.H.J. Kelly, 2012 Innovative Parallel Computing, InPar, 2012, pp. 1–12 http://dx.doi.org/10.1109/InPar.2012.6339594.
[14] I.Z. Reguly, G.R. Mudalige, C. Bertolli, M.B. Giles, A. Betts, P.H.J. Kelly, D. Radford, IEEE Trans. Parallel Distrib. Syst. 27 (5) (2016) 1265–1278. http://dx.doi.org/10.1109/TPDS.2015.2453972.
[15] I.Z. Reguly, E. László, G.R. Mudalige, M.B. Giles, Concurr. Comput.: Pract. Exper. 28 (2) (2016) 557–577. http://dx.doi.org/10.1002/cpe.3621.
[16] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, P. Hanrahan, Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 9:1–9:12. http://dx.doi.org/10.1145/2063384.2063396.
[17] L. Di Mare, D.Y. Kulkarni, F. Wang, A. Romanov, P.R. Ramar, Z.I. Zachariadis, Proceedings of ASME TurboExpo, Vancouver, Canada, 2011.
[18] F. Wang, M. Carnevale, G. Lu, L. Di Mare, D. Kulkarni, ASME Turbo Expo 2016: Turbomachinery Technical Conference and Exposition, American Society of Mechanical Engineers, 2016, V001T01A009–V001T01A009.
[19] A. Jameson, AIAA Pap. 1596 (1991).
[20] D. Wilcox, AIAA J. 26 (1988) 1299–1310. http://dx.doi.org/10.2514/3.10041.
[21] D. Mavriplis, Revisiting the Least-squares Procedure for Gradient Reconstruction on Unstructured Meshes, Technical Report NASA/CR-2003-212683, National Aeronautics and Space Administration, 2003.
[22] P.L. Roe, J. Comput. Phys. 43 (2) (1981) 357–372.
[23] B. van Leer, J. Comput. Phys. 32 (1) (1979) 101–136. http://dx.doi.org/10.1016/0021-9991(79)90145-1.
[24] C. Hirsch, Numerical Computation of Internal and External Flows, John Wiley and Sons, Chichester, West Sussex, UK, 1990.
[25] M. Carnevale, J.S. Green, L. Di Mare, Proceedings of ASME Turbo Expo 2014: Turbine Technical Conference and Exposition, 2014, pp. 16–20.
[26] M. Carnevale, F. Wang, L. Di Mare, J. Eng. Gas Turbines Power 139 (4) (2017) 041203.
[27] J.P. Murphy, D.G. MacManus, Exp. Fluids 50 (1) (2011) 109–124. http://dx.doi.org/10.1007/s00348-010-0902-4.
[28] P. Gepner, V. Gamayunov, D.L. Fraser, Procedia Comput. Sci. 4 (2011) 452–460. Proceedings of the International Conference on Computational Science, ICCS 2011.
[29] Intel Corporation, Intel® 64 and IA-32 Architectures Optimization Reference Manual, 248966–037, 2017.
[30] Intel Xeon Scalable Processors, 2017. https://newsroom.intel.com/press-kits/next-generation-xeon-processor-family/. (Accessed 18 July 2017).
[31] J.D. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers, Technical Report, University of Virginia, Charlottesville, Virginia, 1991–2007, A continually updated technical report, http://www.cs.virginia.edu/stream/.
[32] J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Kaufmann, Boston, United States, 2013.
[33] J. Jeffers, J. Reinders, A. Sodani, Intel Xeon Phi Processor High Performance Programming, Morgan Kaufmann, 2016.
[34] A. Sodani, R. Gramunt, J. Corbal, H.S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, Y.C. Liu, IEEE Micro 36 (2) (2016) 34–46. http://dx.doi.org/10.1109/MM.2016.25.
[35] S.J. Pennycook, C.J. Hughes, M. Smelyanskiy, S.A. Jarvis, Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 1085–1097. http://dx.doi.org/10.1109/IPDPS.2013.44.
[36] ARCHER. Knights Landing Testing & Development Platform, 2017. http://www.archer.ac.uk/documentation/knl-guide/. (Accessed 1 August 2017).
[37] D.A. Burgess, M.B. Giles, Adv. Eng. Softw. 28 (3) (1997) 189–201. http://dx.doi.org/10.1016/S0965-9978(96)00039-7.
[38] OpenMP 4.0 Specifications, 2016. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf. (Accessed 15 May 2016).

[39] R. Löhner, Int. J. Numer. Methods Biomed. Eng. 26 (5) (2010) 628–636. http://dx.doi.org/10.1002/cnm.1160.

[40] I. Hadade, L. Di Mare, Comput. Phys. Comm. 205 (2016) 32–47. http://dx.doi.org/10.1016/j.cpc.2016.04.006.

[41] T.C. Mowry, M.S. Lam, A. Gupta, SIGPLAN Not. 27 (9) (1992) 62–73. http://dx.doi.org/10.1145/143371.143488. URL http://doi.acm.org/10.1145/143371.143488.

[42] J. Lee, H. Kim, R. Vuduc, ACM Trans. Archit. Code Optim. 9 (1) (2012) 2:1–2:29. http://dx.doi.org/10.1145/2133382.2133384.

[43] S. Ainsworth, T.M. Jones, Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 305–317. URL http://dl.acm.org/citation.cfm?id=3049832.3049865.

[44] R. Aubry, G. Houzeaux, M. Vzquez, J.M. Cela, Internat. J. Numer. Methods Engrg. 85 (5) (2011) 537–561. http://dx.doi.org/10.1002/nme.2973.