

Modern multicore and manycore architectures: Modelling, optimisation and benchmarking a multiblock CFD code



Ioan Hadade*, Luca di Mare

Whole Engine Modelling Group, Rolls-Royce Vibration UTC, Mechanical Engineering Department, Imperial College London, South Kensington, SW7 2AZ, London, United Kingdom

ARTICLE INFO

Article history:

Received 3 October 2015

Received in revised form

4 April 2016

Accepted 13 April 2016

Available online 22 April 2016

Keywords:

Computational fluid dynamics

Code optimisation

SIMD

SandyBridge

Haswell

Xeon Phi

Parallel performance

ABSTRACT

Modern multicore and manycore processors exhibit multiple levels of parallelism through a wide range of architectural features such as SIMD for data parallel execution or threads for core parallelism. The exploitation of multi-level parallelism is therefore crucial for achieving superior performance on current and future processors. This paper presents the performance tuning of a multiblock CFD solver on Intel SandyBridge and Haswell multicore CPUs and the Intel Xeon Phi Knights Corner coprocessor. Code optimisations have been applied on two computational kernels exhibiting different computational patterns: the update of flow variables and the evaluation of the Roe numerical fluxes. We discuss at great length the code transformations required for achieving efficient SIMD computations for both kernels across the selected devices including SIMD shuffles and transpositions for flux stencil computations and global memory transformations. Core parallelism is expressed through threading based on a number of domain decomposition techniques together with optimisations pertaining to alleviating NUMA effects found in multi-socket compute nodes. Results are correlated with the Roofline performance model in order to assert their efficiency for each distinct architecture. We report significant speedups for single thread execution across both kernels: 2–5X on the multicore CPUs and 14–23X on the Xeon Phi coprocessor. Computations at full node and chip concurrency deliver a factor of three speedup on the multicore processors and up to 24X on the Xeon Phi manycore coprocessor.

© 2016 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/>).

1. Introduction

Modern research and engineering rely heavily on numerical simulations. In research, improvements in the speed and accuracy of scientific computations can lead to new discoveries or facilitate the exploitation of recent breakthroughs [1]. In engineering, improvements in solution time can lead to better designs and to a reduction of the development phase – and costs – of new products. There is a constant need, therefore, for sustained improvements in the speed of execution of scientific and engineering codes.

Historically, performance gains in scientific and engineering applications have been obtained through advances in hardware engineering which required little or no change to the programming paradigms. Examples of such innovations were out-of-order execution, branch prediction, instruction pipelining, deeper memory hierarchies and, of course, the increase in clock frequency [2] all of

which guaranteed improvements in serial performance with every new CPU generation and limited code intervention. Those days are now gone partly due to the recognition that clock frequency cannot be scaled indefinitely because of power consumption, and partly because circuitry density on the chip is approaching the limit of existing technologies which is problematic as innovations in sequential execution require a high fraction of die real estate [3].

The current trend in CPU design is parallelism [4] and has led to the rise of multicore and manycore processors whilst effectively ending the so called “free lunch” era [5,6] in performance scaling. Modern multicore and manycore processors now consist of double digit core numbers integrated on the same die, vector units with associated instruction set extensions, multiple backend execution ports and deeper and more complex memory hierarchies with features such Uniform-Memory-Access (UMA) and Non-Uniform-Memory-Access (NUMA), to name a few. Consequently, achieving any reasonable performance on these architectures mandates the exploitation of all architectural features and their intrinsic parallelism across all granularities (core, data and instruction) [6].

Core and data parallelism are exposed in modern applications through two principal mechanisms: threads and Single Instruction

* Corresponding author.

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

Multiple Data (SIMD) constructs. In order for core parallelism to be exploitable, the computations need to be arranged in such a way as not to contain dependencies among data being processed concurrently. Additionally, good data locality and temporality is needed to achieve efficient execution on a thread's local working set. The same considerations apply for efficient data parallelism where SIMD computations require data to be packed in sets of size matching the width of the underlying vector registers whilst being aligned at correct memory address boundaries.

Optimising existing applications to make full use of all available levels of parallelism can result in considerable speedups – up to one order of magnitude – in many fields of scientific computing [7,8]. However, extracting performance from SIMD and thread-level parallelism is not trivial and a careless implementation can easily obliterate the advantages of modern processors [9,10,6].

Firstly, the compiler may detect data dependencies, even when none really exist, unless the source code is made un-ambiguous through directives. In some cases, the data layout of the original application is not suitable for efficient execution of vector load and store operations. This happens if the data items are not aligned within the boundaries of vector registers. An example of this situation occurs in stencil computations on discretised PDEs: values at several neighbouring points on a lattice are used simultaneously, but values that would naturally be loaded in the same vector register appear in different operands. Algorithms which access simultaneously data stored at addresses far from each other, or access the same data repeatedly with separate load or store operations also do not perform well on modern multicore and many-core architectures. This is due to memory bandwidth limitations and Transfer-Lookaside-Buffer (TLB) misses. Furthermore, on the majority of these platforms, not all addresses in memory can be accessed by all cores with the same latency. This effect, known as Non-Uniform Multiple Access (NUMA) can limit scalability on multicore platforms. Finally, scalability deteriorates as single-core execution is optimised, because the latency of synchronisation and communication operations becomes more and more difficult to hide and because resources such as memory bandwidth become increasingly saturated.

A number of techniques have been used in literature to alleviate these issues. These techniques aim at hiding the latency of memory transfers, or at arranging the data in a way that is better suited for vector load/store operations. Henretty [11] applied data transposition for SIMD optimisations of stencil operations in PDE-based applications. Rostrup [12] applied similar techniques with the addition of SIMD shuffles for hyperbolic PDE solutions on Cell processors and NVIDIA GPUs. Rosales et al. [13] studied the effect of data transformations for SIMD execution and thread-scalability in a Lattice Boltzmann code for the manycore Intel Xeon Phi (Knights Corner) processor. Datta [14], Jaeger [15] and Schafer [16] studied the problem of determining and applying automatically the necessary architectural optimisations for stencil computations on modern multicore and manycore processors.

Incorporating these techniques in existing codes is no small task: accessing some of the features of the hardware may require very specialised, non portable code. On occasions, sections of code may need to invoke platform-specific intrinsic functions or to be written in assembler. For certain application fields, active research is under way towards automatic tuning [17].

For the general practitioner, however, the issue remains of finding a trade-off between programming effort and gains in performance, both in the short and in the long term [1]. For this purpose it is essential to expose domain scientists to the whole range of techniques available, and to their impact on realistic applications and up-to-date hardware.

This paper addresses a number of optimisation techniques and compares their effect on three different leading platforms.

The practicalities of SIMD/threading optimisation in a solver of complexity and structure representative of industrial CFD applications are discussed in detail. A fair and meaningful assessment of the gains afforded by each technique and on each platform is ensured by the use of a performance model.

2. Numerical algorithm

The test vehicle for this study is a fast Euler solver designed for quick calculations of transonic turbo-machinery flows. The solver computes inviscid flow solutions in the $m' - \theta$ coordinate system [18].¹

A typical computational domain is shown in Fig. 1 and consists of a number of blocks. The blocks are connected by stitch lines.

The Euler equations are solved in semi-discrete form

$$\frac{d}{dt} W_{i,j} \mathbf{U}_{i,j} = \mathbf{F}_{i-1/2,j} - \mathbf{F}_{i+1/2,j} + \mathbf{G}_{i,j-1/2} - \mathbf{G}_{i,j+1/2} + \mathbf{S}_{i,j} = \mathbf{RHS}_{i,j}. \quad (1)$$

In Eq. (1), $W_{i,j}$ is the volume of cell i, j , $\mathbf{U}_{i,j}$ is the vector of conserved variables and the vectors $\mathbf{F}_{i-1/2,j}$, $\mathbf{G}_{i,j-1/2}$ and $\mathbf{S}_{i,j}$ denote fluxes through i -faces, j -faces and source terms, respectively and are evaluated as follows:

$$\mathbf{F}_{i-1/2,j} = s_{i-1/2,j}^{\zeta} \begin{bmatrix} \rho w_{\zeta} \\ \rho w_{\zeta} u_m + p n_m^{\zeta} \\ \rho w_{\zeta} u_{\theta} + p n_{\theta}^{\zeta} \\ \rho w_{\zeta} h - p w_{\zeta} \end{bmatrix}_{i-1/2,j} \quad (2)$$

$$\mathbf{G}_{i,j-1/2} = s_{i,j-1/2}^{\eta} \begin{bmatrix} \rho w_{\eta} \\ \rho w_{\eta} u_m + p n_m^{\eta} \\ \rho w_{\eta} u_{\theta} + p n_{\theta}^{\eta} \\ \rho w_{\eta} h - p w_{\eta} \end{bmatrix}_{i,j-1/2} \quad (3)$$

$$\mathbf{S}_{i,j} = W_{i,j} \rho_{i,j} \begin{bmatrix} 0 \\ u_{\theta}^2 \sin \phi \\ u_m u_{\theta} \cos \phi \\ 0 \end{bmatrix}_{i,j}. \quad (4)$$

For the purpose of flux and source term evaluation, the contravariant velocities $w_{\zeta/\eta}$, the normals $n_{m,\theta}^{\zeta}$ and $n_{m,\theta}^{\eta}$ and the radial flow angle ϕ are also needed. Further details are reported in the Appendix.

Convergence to a steady state is achieved by a matrix free, implicit algorithm. At each iteration, a correction to the primitive variable vector \mathbf{V} is determined as solution to the linear problem [19]

$$\frac{\delta (W_{i,j} \mathbf{U}_{i,j})}{\delta t} = \mathbf{RHS}_{i,j} + \mathbf{J}_{i,j}^{h,k} \delta \mathbf{V}_{h,k} \quad (5)$$

or, equivalently

$$(W_{i,j} \mathbf{K}_{i,j} - \mathbf{J}_{i,j}^{h,k}) \delta \mathbf{V}_{i,j} = -(\delta W_{i,j}) \mathbf{U}_{i,j} + \mathbf{RHS}_{i,j} \quad (6)$$

where $\mathbf{V}_{i,j}$ is the vector of primitive variables at the cell i, j and $\mathbf{K}_{i,j}$ is the transformation Jacobian from primitive variables to conserved variables. The linear problem in Eq. (6) can be approximated by a diagonal problem if the assembled flux Jacobian $\mathbf{J}_{i,j}^{h,k}$ is replaced by

¹ θ is the angular position around the annulus. m is the arc length evaluated along a stream surface $dm = \sqrt{dx^2 + dr^2}$ and m' is a normalised (dimensionless) curvilinear coordinate defined from the differential relation $dm' = \frac{dm}{r}$. The $m' - \theta$ system is used in turbomachinery computations because it preserves aerofoil shapes and flow angles.

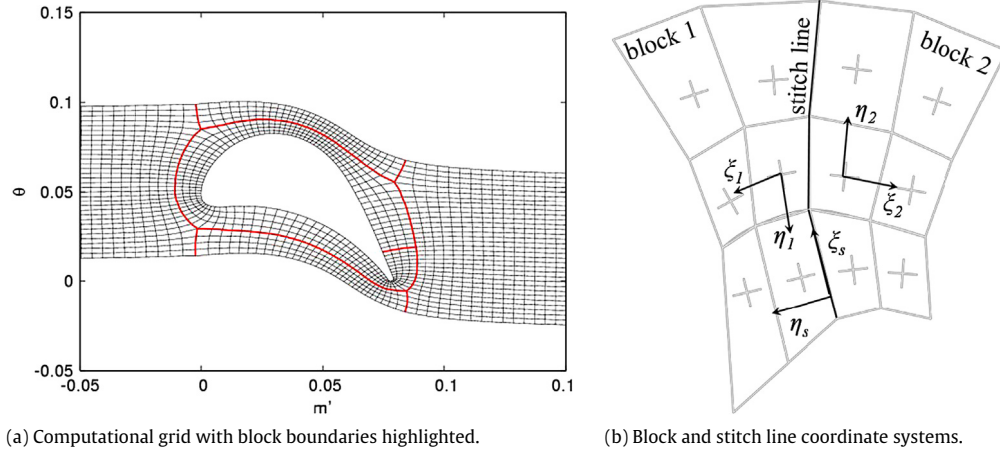


Fig. 1. Computational domain and topology.

a matrix bearing on the main diagonal the sum of the spectral radii $|\tilde{A}|$ of the flux Jacobian contributions for each cell

$$\mathbf{J}_{i,j}^{h,k} \approx -\text{diag} \left(\sum s |\tilde{A}|_{\max} \right)_{i,j}. \quad (7)$$

At a fixed Courant number $\sigma = \frac{\delta t_{i,j} \sum s |\tilde{A}|_{\max}}{W_{i,j}}$ this approximation yields the following update

$$\delta \mathbf{V}_{i,j} = \frac{1}{W_{i,j} (1 + \sigma)} \mathbf{K}_{i,j}^{-1} (-\mathbf{U}_{i,j} \delta W_{i,j} + \mathbf{RHS}_{i,j}). \quad (8)$$

The solver has been validated against MISES [20] for turbine test cases.

For the purpose of implementation, the baseline solver stores the primitive variables $\mathbf{V}_{i,j}$ at each cell. Auxiliary quantities such as speed of sound and total enthalpy at each cell are also stored.

Inspection of Eqs. (1)–(8) reveals the existence of two types of operations. The first type evaluates cell attributes and is performed by looping over the cells. The evaluation of $\mathbf{S}_{i,j}$ in Eq. (1) or the block diagonal inversion in Eq. (8) falls within this category. The second type evaluates stencil operators and needs to be performed by looping over the neighbours of each cell or by looping over the cell boundaries. The evaluation of the numerical fluxes $\mathbf{F}_{i\pm 1/2,j}$ and $\mathbf{G}_{i,j\pm 1/2}$ in Eq. (1) is an example of such stencil kernels.

An example of the solver's solution output in terms of computed static pressure in first stage turbine blade configurations can be seen in Fig. 2. The application is implemented as a set of C++ classes. Eqs. (1)–(8) are grouped in the methods of a class embodying an abstraction of a gas model. The methods of this class are called at appropriate points in the execution of the program to compute fluxes, updates, etc. All the remaining elements of the discretisation, e.g. block geometry and layout, are handled by a different set of classes.

All floating point operations are carried out in double precision. The solver spends 75% of time in computing the Roe fluxes and approximately 15% on updating the primary flow variables after each main iteration. The aim of this work is to evaluate and compare the optimisation process of both the quasi-stencil-based Roe flux computations and the cell-wise primary flow variable update on the three different hardware platforms.

3. Hardware and performance model

The three architectures chosen for this study are representative of modern multicore and manycore processors in terms of SIMD features and core count, namely the Intel® Xeon® Sandy Bridge, Intel® Xeon® Haswell and Intel® Xeon Phi™ Knights Corner.

The following sections will delve deeper into the architectural characteristics of the selected processor candidates including details of the system configuration and applied performance model.

3.1. Intel® Xeon® Sandy bridge

The SandyBridge microarchitecture introduces 256 bit vector registers and an extension to the instruction set known as Advanced Vector eXtensions (AVX) [21]. The AVX extensions are backward compatible with SSE and, to this effect, the AVX registers on SandyBridge are implemented as two fused 128 bit SSE lanes. This complicates somewhat cross-lane elemental operations such as shuffle and permute. AVX supports three non-destructive operands for vector operations, e.g. $x = y + z$, as well as functions for merging and masking registers. On SandyBridge vector loads are performed using two ports simultaneously. This prevents the overlap of store and load operations when operating on whole 256 bit registers and has an impact on instruction level parallelism [22]. For arithmetic purposes, SandyBridge can execute one vector multiplication and one addition in the same clock cycle. The memory subsystem comes with an improved 32KB L1D cache compared to Nehalem which is required by AVX and can sustain two 128 bit loads and a 128 bit store every cycle. The 256KB L2 cache is similar to the previous generation implementation providing 8-way associativity and a 12 cycle load-to-use latency with a write-back design. The L3 cache is shared by all cores within a ring interconnect and is sliced and distributed across the ring allowing for better proximity to cores.

The SandyBridge-EP node evaluated in this paper holds a two-socket Xeon E5-2650 configuration with 16 physical cores and 32 GB of DDR3 main memory (ECC on). The CPUs are clocked at 2.0 GHz with HyperThreading and TurboBoost disabled. The node runs Scientific Linux 6.7 (Carbon) kernel version 2.6.32-504. Code compilation has been performed with Intel icpc 15.0 with the following flags: -O3 -xAVX -fargument-noalias -restrict -openmp.

3.2. Intel® Xeon® Haswell

The Haswell microarchitecture is based on a 22 nm design and is a successor to IvyBridge. Whereas IvyBridge did not contain any major architectural changes compared to its Sandy Bridge predecessor, Haswell provides a number of modifications through a new core design, improved execution unit based on the AVX2 extensions and a revamped memory subsystem [23]. The major improvements to the execution unit in the Haswell microarchitecture regard the addition of two ports, one for

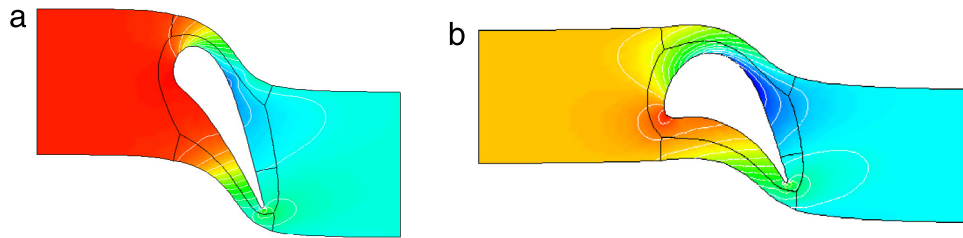


Fig. 2. Static pressure solutions for first stage stator (a) and first stage rotor (b).

memory and one for integer operations which aids instruction level parallelism as the remaining five can target floating point arithmetic. To that end, there are two execution ports for performing FMA operations as part of AVX2 with peak performance of 16 double precision FP operations per cycle, double that of SandyBridge and IvyBridge. Further improvements provided by AVX2 are gather operations which are particularly useful for packing non-contiguous elements within a vector register and whole register permutations and shuffles with the implementation of the `vpermq` instruction. From a memory standpoint, the Haswell fixes the backend port width issue by providing 64 byte load and 32 byte store per cycle functionality therefore alleviating the stalls in memory bound codes which occurred in SandyBridge and IvyBridge.

The Haswell-EP node consists of 2x 10-core Xeon E5-2650 CPUs (20 physical cores in total) clocked at 2.3 GHz and 64 GB of DDR4 main memory (ECC on). The node runs Oracle Linux Server 6.6 kernel 2.6.32-504. Code compilation has been done via the Intel `icpc` 15.0 compiler with the following flags: `-O3 -xCORE-AVX2 -fargument-noalias -restrict -openmp`.

3.3. Intel® Xeon® Xeon Phi™ knights corner

Knights Corner is the first commercial iteration of the Intel Xeon Phi manycore processor. The device can be described as an x86-based Shared-Memory-Multiprocessor-on-a-chip [24] with over 50 cores on the die and up to four hardware threads per core. The main computational device in each core is a Vector Processing Unit (VPU). The VPU operates on 32 512 bit wide registers and 8 mask registers. The VPU offers functionalities geared towards scientific computations such as FMA, gather and scatter, horizontal reductions and support for cross-lane permutations similar to AVX2 [25]. In theory, the VPU can execute one FMA (512 bit) operation every cycle, giving it a 16 DP FLOP per cycle ratio similar to the Haswell microarchitecture. However, due to the in-order execution nature of the core design, one must run on at least two hardware threads in order to hide memory latency and achieve peak FLOPS [24]. The cores are connected to a high-speed bidirectional ring interconnect providing full cache coherence. Communication with the host CPU is done via the PCI-Express bus in a similar fashion to Graphics Processing Units (GPUs). For this study the Xeon Phi card was used in native mode, i.e. the algorithm was executed by a process started by the card's own operating system. The card can be used alternatively in offload mode or in symmetric mode.

The Intel Xeon Phi Knights Corner coprocessor evaluated in this study is based on the 5110P product series and contains 60 physical cores (4 hyperthreads each) clocked at 1.053 GHz and 8 GB of GDDR5 memory. The coprocessor runs a bespoke version of Linux kernel 2.6.32-504.8.1 and Intel Manycore Platform Stack (MPSS) version 3.4.3 used for host and coprocessor communication. Compilation and runtime environments were performed via Intel's `icpc` 2015 compiler with the following flags: `-O3 -mmic -fargument-noalias -restrict -openmp`.

3.4. The roofline model

In order to determine the effect of interventions on the code rigorously, performance models are built and correlated with each platform. A good performance model can highlight how well the implementation of an algorithm uses the available hardware both in terms of in-core and intra-core performance. The Roofline [26,17,27] performance model is used in this work. The model is based on the assumption that the principal vectors of performance in numerical algorithms are computation, communication and locality [17]. The Roofline model defines computation as floating point operations, communication as unit of data from memory required by the computations and locality as the distance in memory from where the data is retrieved i.e. (cache, DRAM, etc.). However, due to the fact that modelling the performance of the entire cache system is a complex task for one architecture, let alone three, the locality component for this work will be set to DRAM main memory, similar to previous work performed in literature [17]. The correlation between computation and communication is defined by the model as the arithmetic intensity of a given kernel which can be expressed as the ratio of useful FLOPS to the corresponding number of bytes requested from the data source, in our case, main memory. As the units of measurement for both flops and byte transfers in current CPU architectures are given as GigaFLOPS/sec and GigaBYTES/sec, the maximum attainable performance of a computational kernel, measured in GigaFLOPS/sec, can be calculated as:

Max. GFlops/sec

$$= \min \left\{ \begin{array}{l} \text{Peak FP Performance} \\ \text{Max. Memory Bandwidth} \times \text{Kernel Flops/Bytes} \end{array} \right. \quad (9)$$

Peak floating point is obtained from the processor's documentation manual whilst maximum DRAM memory bandwidth from the STREAM [28,29] benchmark. The model visualises the metrics in a diagram [26] where the attainable performance of a kernel is plotted as a function of its arithmetic intensity. This acts as the main roofline of the model and exhibits a slope followed by a plateau when peak FLOPS is reached. The position at which the arithmetic intensity coupled with the available memory bandwidth equals peak FLOPS is called the "ridge" and signifies the transition from memory to compute bound. However, achieving either maximum memory bandwidth or peak FLOPS on modern multicore and manycore processors requires the exploitation of a plethora of architectural features even though the algorithm might exhibit a favourable arithmetic intensity. For example, achieving peak FLOPS requires a balance of additions and multiplications for fully populating all functional units, data level parallelism through SIMD and loop unrolling for instruction level parallelism. Reaching the memory roofline requires that memory accesses are performed at unit strides, some form of prefetching and that NUMA effects are catered for when running at full chip concurrency [17,30]. To this extent, subsequent rooflines can be added which need to be "pierced" through by specific optimisations in order to reach maximum attainable performance.

Roofline diagrams for all three architectures can be seen in Fig. 3. Memory bandwidth saturation effects [31] i.e. the inability

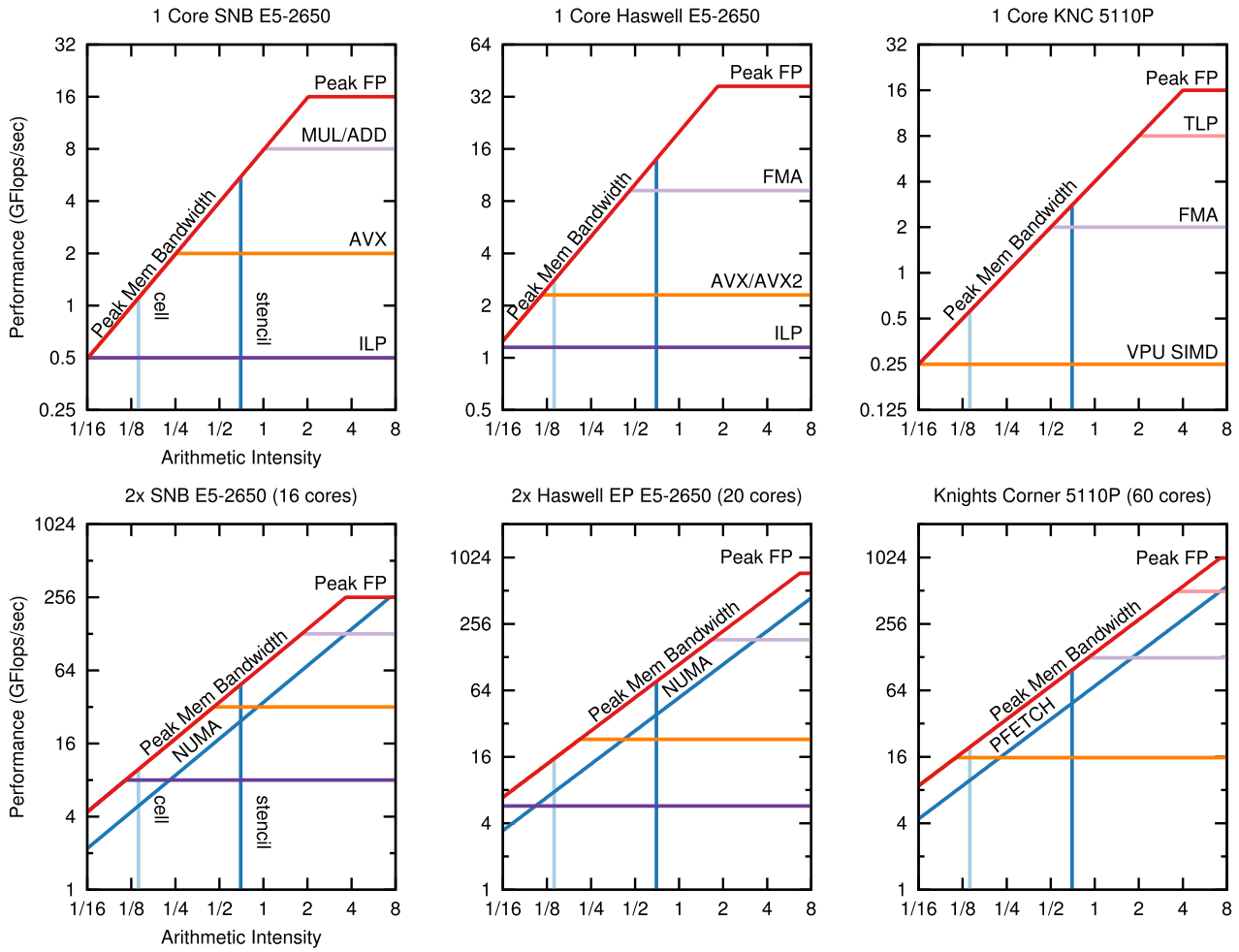


Fig. 3. Core (top) and Node (bottom) Roofline plots of all architectures.

of a single core to utilise all available controllers, mandates that separate diagrams are built for single-core and full node configurations. The primary flow variable update kernel (cell) and flux computations (stencil) are plotted at their respective arithmetic intensity in order to visualise their optimisation space and maximum attainable performance across all platforms. Rooflines for in-core performance optimisations are added with respect to the particularities of each processor for both single-core and full node configurations. Rooflines for memory optimisations horizontal to peak memory bandwidth are added only in the full node diagrams as most of them such as NUMA apply only at full-chip and full-node concurrency. The order of the rooflines is set in regard to how likely it is for the compiler to automatically apply such optimisations without external interventions [17].

For the SandyBridge Xeon E5-2650 core, a balance between additions and multiplications is required due to the design of the execution ports where one multiplication and one addition can be performed every clock cycle. An imbalanced code will fully utilise the addition or multiplication execution unit whilst the other sits idle therefore reducing throughput by a factor of two. Furthermore, lack of SIMD computations via AVX (256 bit) further decreases performance by a factor of four from the previous roofline. Poor use of instruction level parallelism brings forth another factor of four in performance due to the reduced number of in-flight instructions from four to one in the worst case scenario.

On the Haswell Xeon E5-2650 core, lack of FMA operations produces a factor of four performance drop. This is the case for kernels that contain a large number of additions and a minimal

number of multiplications such as PDE-based stencil operations and as such, similar to the application presented in this work. The reason for such a drastic performance drop is due to the fact that the two FMA units sit idle whilst the separate single ADD unit gets overloaded. A possible solution to the above would be the manual insertion of a 1.0 multiplier, however, such optimisation would suffer from portability issues on other architectures that do not implement FMA i.e. SandyBridge. Furthermore, absence of AVX/AVX2 execution on Haswell leads to another factor of four decrease in performance, similar to SandyBridge.

On the Knights Corner 5110P core, the out of order execution mandates the use of multiple threads to hide memory latency and fully occupy the VPU unit. In theory, the VPU unit can be filled by running between two or three threads on the core. Running with only one thread in-flight reduces the attainable peak performance by a factor of two as the VPU would be capable of performing an FMA operation every other cycle. Furthermore, similarly to Haswell, an imbalanced kernel that cannot fully exploit FMA operations on Knights Corner will suffer a factor of four drop in performance when also running in single thread mode. Lack of SIMD brings forth an eight fold performance decrease due to the wide SIMD nature of the VPU (512 bit).

The node Roofline diagrams extend on their core counterparts by presenting memory optimisations such as NUMA on SandyBridge and Haswell and pre-fetching for Knights Corner. The optimisations applied to both the flow variable update and Roe numerical fluxes kernels and presented in the following section aim at piercing through the in-core and intra-core rooflines and

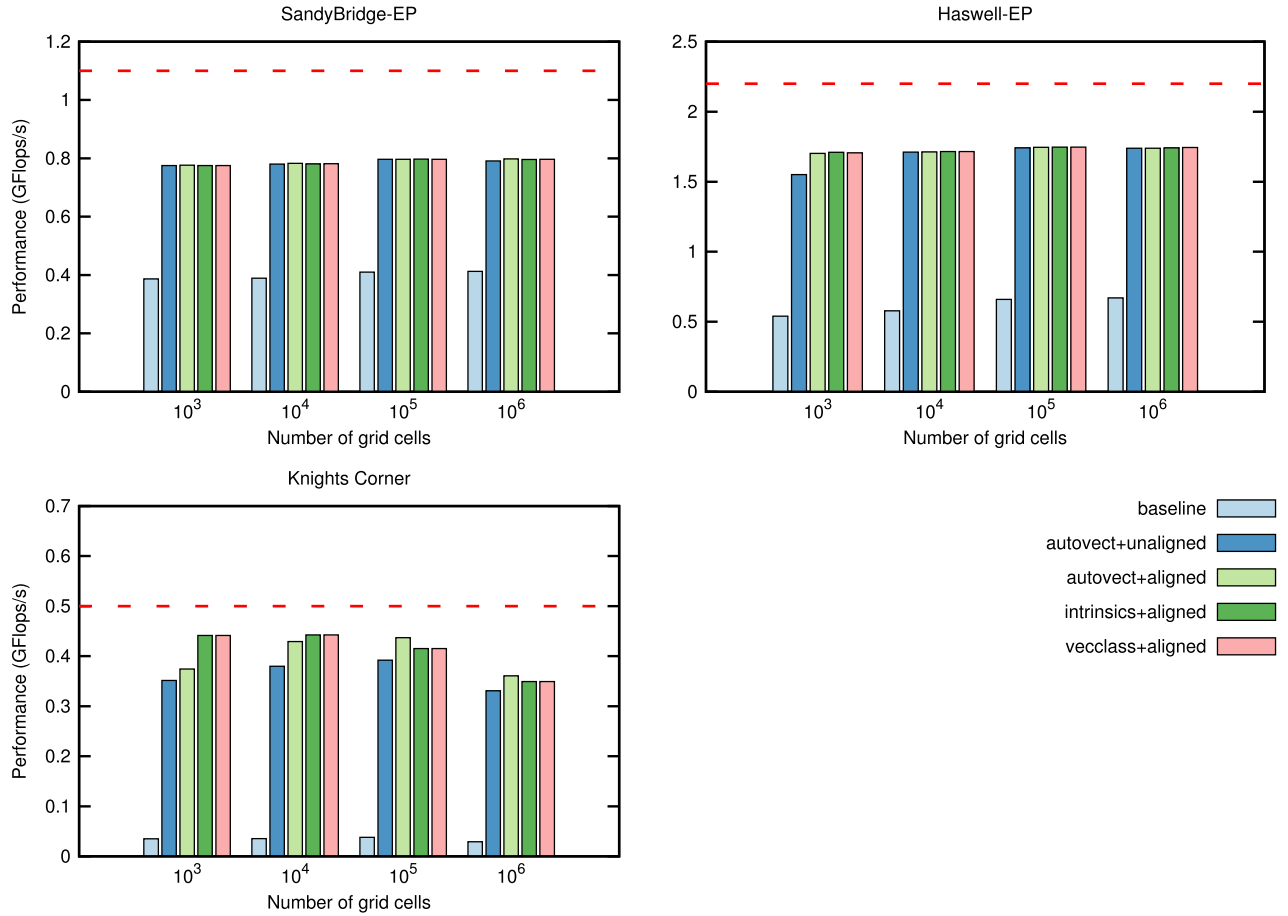


Fig. 4. Primary variable update SIMD optimisations with rooflines.

achieve a high percentage of the predicted maximum attainable performance for serial and full concurrency execution. The aim is also to highlight the fact that certain optimisations, such as efficient SIMD execution, have a high impact on performance across all architectures for kernels with moderate to high arithmetic intensity, as is the case for the stencil-based flux computations. For kernels which exhibit a low arithmetic intensity such as the cell-wise flow variable update, the model predicts that SIMD would only affect runs on the Knights Corner coprocessor. However, the nature of the optimisations required for vector computations usually leads to a higher degree of instruction parallelism due to loop unrolling and instruction dispatch reduction. Consequently, these might also provide speedups on SandyBridge and Haswell.

4. Results and discussions

This section presents in detail a number of optimisation techniques that have been applied to the selected computational kernels introduced in Section 2. Results are reported as GFLOPS/sec so as to validate them against the performance model and have been obtained as average wall clock times spent in the kernels over 10^2 main iterations, in order to mimic the natural utilisation of the application.

4.1. Flow variable update

The cell-wise kernel computes the primitive variables updates, based on the residuals of the discretised Euler equations, as defined in Eq. (8). The kernel accesses the arrays storing the flow variables, the residuals, an array storing auxiliary variables and an array

storing the spectral radii of the flux Jacobians. The arrays are passed to the function through their base pointers. The flow and auxiliary variables are used to compute the entries of the transformation Jacobian K_{ij}^{-1} between conserved variables and primitive variables. The kernel performs 40 FLOPS per cell and accesses 35 double precision values giving it a 0.14 flops/byte ratio. Fig. 4 presents the results derived from applying SIMD optimisations. Fig. 6 provides the results from applying SIMD and memory optimisations whilst results for full node concurrency (thread-level parallelism) performance and speedup can be examined in Figs. 7 and 8 respectively.

4.1.1. Data-level parallelism

Autovectorisation (**autovect + unaligned** – Fig. 4) has been achieved through the use of `#pragma omp simd` directive (OpenMP 4.0). The use of OpenMP 4.0 directives is found preferable to compiler-specific directives because it is more portable. In order to achieve efficient vectorisation, the qualifier `restrict` needs to be added to the function arguments. This guarantees to the compiler that arrays represented by the arguments do not overlap. In absence of further provisions, the compiler generates code for unaligned loads and stores. This is a safety precaution, as aligned SIMD load/store instructions on unaligned addresses lead to bus errors on the Xeon Phi Coprocessor. SandyBridge-EP and Haswell-EP processors can deal with aligned access instructions on unaligned addresses, albeit with some performance penalty due to inter-register data movements that are required. On SandyBridge-EP and Haswell-EP, even the auto-vectorised kernel performs twice and three times faster than the baseline kernel. On Knights Corner, the improvement is almost one order of magnitude as the VPU and therefore wide SIMD lanes were not previously exploited.

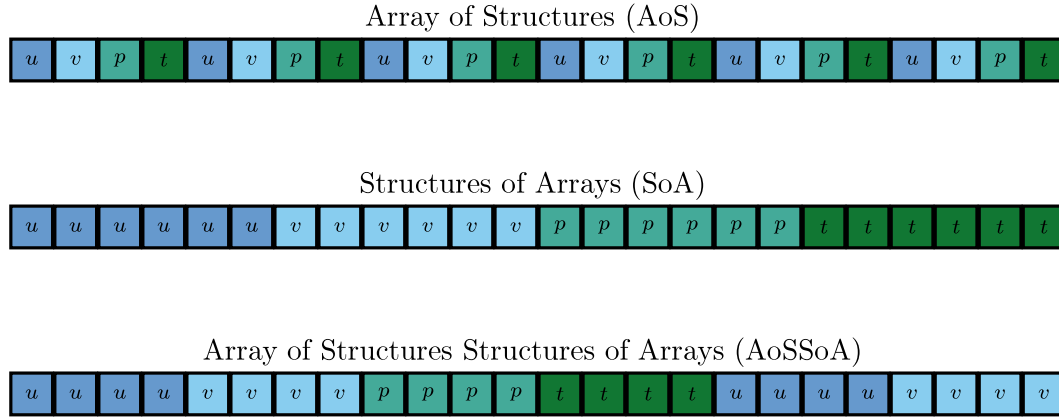


Fig. 5. Array of Structures, Structures of Arrays and hybrid Array of Structures Structures of Arrays format.

Aligned vector accesses (**autovect** + **aligned** – Fig. 4) can be achieved by issuing the **aligned** qualifier to the original **simd** directive and by allocating all the relevant arrays using the **_mm_malloc** function. **_mm_malloc** takes an extra argument representing the required alignment (32 bytes for AVX/AVX2 and 64 bytes for KNC VPU). A number of additional directives may be needed to persuade the compiler that aligned loads can be issued safely, as shown in the snippet below for a pointer storing linearly four variables in Structures of Arrays (SoA) arrangement, at offsets **id0**, **id1**:

```

1  __assume_aligned(rhs, 64);
2  __assume(id0%8==0);
3  __assume(id1%8==0);
4  __assume((id0*sizeof(rhs[0]))%64==0);
5  __assume((id1*sizeof(rhs[0]))%64==0);

```

Listing 1: Example of extra compiler hints needed for generating SIMD aligned/load stores on KNC.

The **assume_aligned** construct indicates that the base pointer is aligned with the SIMD register boundary. The following **__assume** statements indicate that subscripts and indices accessing the four sections of the array are also aligned on the SIMD register boundary. Stack-resident variables have to be aligned on the vector register boundary through the **aligned** attribute:

```

1  double du,dv,dp,dt; __attribute__((aligned(64)))
    );

```

Listing 2: Declaring stack variables on aligned boundaries on KNC

Aligned vs. unaligned accesses see no benefit on SandyBridge-EP and for smaller problem sizes a marginal improvement on Haswell-EP, due to better L1 cache utilisation. The aligned access version outperforms its unaligned counterpart on Knights Corner as a 512 bit vector register maps across an entire cache line therefore allowing for efficient load and store operations to/from the L1 cache.

SIMD execution with aligned load and stores can also be achieved invoking compiler intrinsics (**intrinsics aligned** – Fig. 4) or by using a vector class (**vector-class aligned** – Fig. 4). For this work, we have used Agner Fog’s Vector Class Library (VCL) [32]. Its main advantage is that ugly compiler intrinsics are encapsulated away allowing for a more readable code. On Knights Corner, an extension of the VCL library was used which was developed by Przemysław Karpiński at CERN [33]. The utilisation of intrinsics and a vector class did not bring forth any speedups to the directive based vectorisation for this particular kernel. This would indicate the fact that the compiler was able to vectorise the code efficiently.

4.1.2. Memory optimisations

A simple way to improve the performance of a memory bound kernel is to replace stored values with re-computed ones. This technique is applied to the data set **memory-reduction** in Fig. 6. The technique is beneficial on Haswell-EP, but holds no palpable improvements on SandyBridge-EP or Knights Corner. A reason for this being that re-computing the conservative variables requires a number of divisions and square root operations with a large latency penalty. Although the overall flop/byte ratio is improved, there is no palpable performance gain.

4.1.3. Software prefetching

Software prefetching (**intrinsics-aligned-prefetch** – Fig. 6) can also be used to improve the performance of memory-bound codes. A prefetching kernel was built upon the previous **intrinsics-aligned** implementation, by using the compiler directive **#pragma prefetch** on SandyBridge-EP and Haswell-EP. On the Knights Corner coprocessor, the available **_mm_prefetch** intrinsics was evaluated. This instruction can take as arguments the level of cache the prefetch is to appear in. When using **_mm_prefetch** it is advisable to disable the compiler prefetcher using the **-no-opt-prefetch** compilation flag (Intel compilers). Prefetching is marginally beneficial on SandyBridge-EP and positively beneficial, for smaller problem sizes, on Knights Corner. This is due to the fact that the compiler might not be able to estimate the overall loop count i.e. input based on geometry file which leads to a less aggressive prefetching regime compared to manual intervention.

4.1.4. Data layout transformations

A version using the Array of Structures Structures of Arrays (AoSSoA) data layout (**intrinsics-aligned-aossoa** – Fig. 6) has also been studied. Four sub vector lengths were tested: 4, 8, 16, 32 double precision values on the multicore processors and 8, 16, 32, 64 on the coprocessor. The best performing sub vector length of AoSSoA proved to be the one equal to the SIMD vector width of the processor (i.e. 4 and 8 respectively). The difference between the AoS, SoA and hybrid AoSSoA format can be seen in Fig. 5. Although AoS exhibits very good intra and inter structure locality by grouping all elements together, it is inefficient from a SIMD perspective as same type operands would have to be manually packed within a single vector register. The generic data layout advisable for SIMD is SoA where large contiguous arrays are fused together allowing for contiguous SIMD load and stores. This was also the initial format of our test vehicle application. However, the latter can lead to performance penalties such as TLB misses for very large arrays. A hybrid approach such as the AoSSoA can alleviate them by offering the recommended SIMD grouping in sub vectors coupled with improved intra and inter structure locality, similar to the AoS layout.

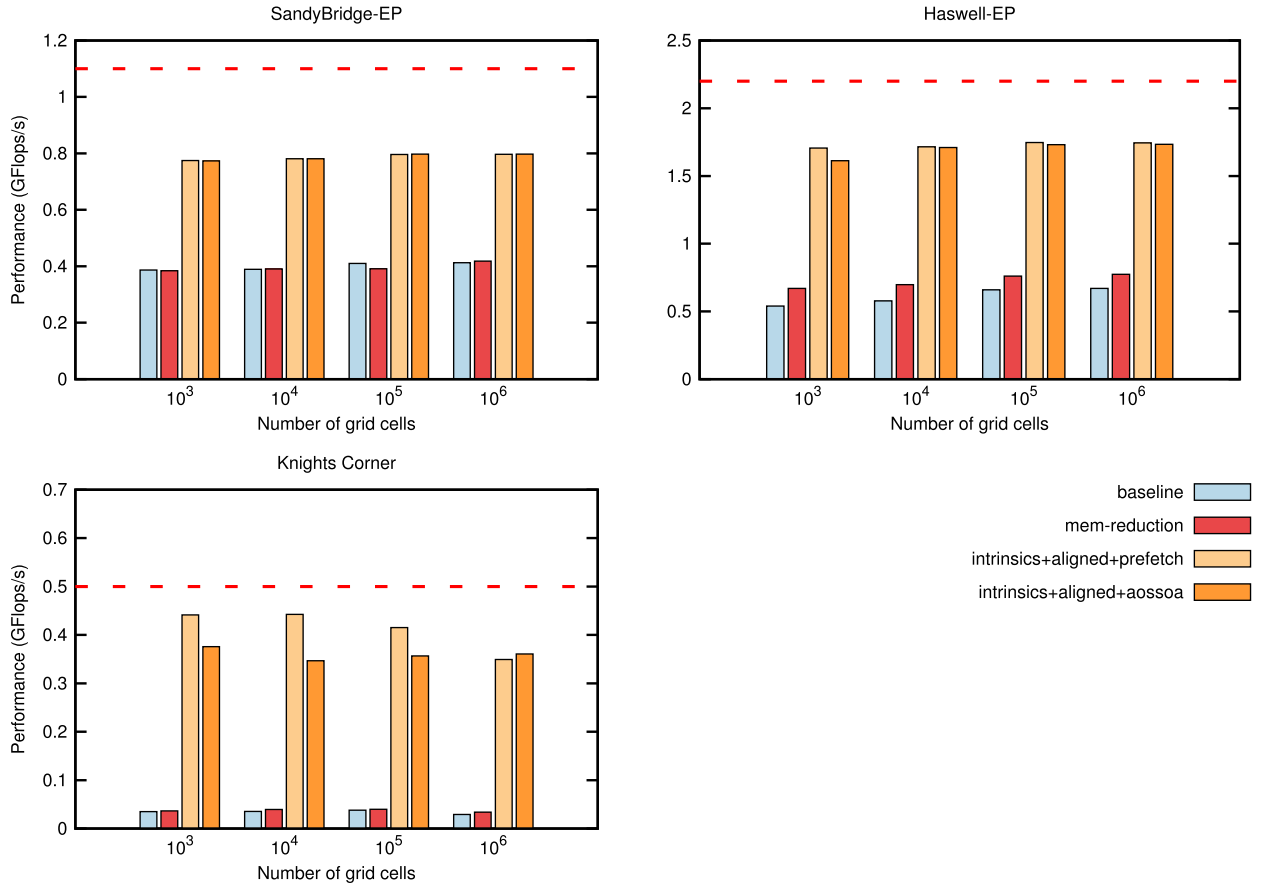


Fig. 6. Primary variable update SIMD + Memory optimisations with rooflines.

4.1.5. Thread parallelism

Thread parallelism of the primary variable update kernel was performed using the `#pragma omp parallel for simd` OpenMP 4.0 construct for the auto vectorised version which complements the initial `#pragma omp simd` statement and further decomposes the loop iterations across all of the available threads in the environment. The intrinsics-based versions used the generic `pragma omp parallel for` construct. The work decomposition has been carried out via a static scheduling clause, whereby each thread receives a fixed size chunk of the entire domain. Additionally, the first touch policy has been applied for all data structures utilised in this method in order to alleviate NUMA effects when crossing over socket boundaries. The first touch is a basic technique which consists in performing trivial operations – e.g. initialisation to zero – on data involved in parallel loops before the actual computations. This forces the relevant sections of the arrays to be loaded on the virtual pages of the core and socket where the thread resides. The first touch needs to be performed with same scheduling and thread pinning as the subsequent computations. In terms of thread pinning, the compact process affinity was used for SandyBridge-EP and Haswell-EP running only with one thread per physical core. On Knights Corner, the scatter affinity brought forth better results compared to compact. The kernel under consideration is memory-bound on all three architectures and is therefore very sensitive to NUMA effects. For Haswell-EP, not using the first touch technique can lead to a large performance degradation as soon as the active threads spill outside a single socket. This can be seen by the behaviour of the data sets **autovect** and **autovect + numa** in Fig. 7.

On the SandyBridge-EP node, all single core optimisations coupled with first touch (Fig. 7) managed to pierce through the predicted memory roofline. The same applies to Haswell-EP runs

where all versions utilising NUMA optimisations have exceeded the model's performance prediction achieving good scaling. A reason as to why discrepancies in NUMA and non-NUMA runs are larger on Haswell-EP when compared to SandyBridge-EP is due to the larger core count on the former which means that the QPI interconnect linking the two sockets gets saturated more quickly due to the increase in cross-socket transfer requests. On Knights Corner there are no NUMA effects due to the ring-based interconnect and the interleaved memory controller placement. However, aligned loads play an important role on this architecture: the SIMD length is 64 bytes (8 doubles) which maps to the size of an L1 cache line. If unaligned loads are issued, the thread is required to go across cache boundaries, loading two or more cache lines and performing inter register movements for packing up the necessary data. This wastes memory bandwidth and forms a very damaging bottleneck. This can be seen from the drop in performance above 60 cores in the **intrinsic + unaligned** and **autovect** data set in Fig. 7. The best performing version was based on the hybrid AoSSoA format and obtained the predicted Roofline performance when run on 96, 128 and 240 threads. Aligned intrinsics combined with manual prefetching performed second best followed by aligned intrinsics with compiler issued prefetches.

Finally, Fig. 8 presents scalability results plotted as parallel speedup (performance on p processors divided by its corresponding $p = 1$ run).

4.2. Roe numerical fluxes

The flux computations kernel loops over a set of cell interfaces and computes numerical fluxes using Roe's approximate Riemann solver. The kernel accepts pointers to the left and right states, to

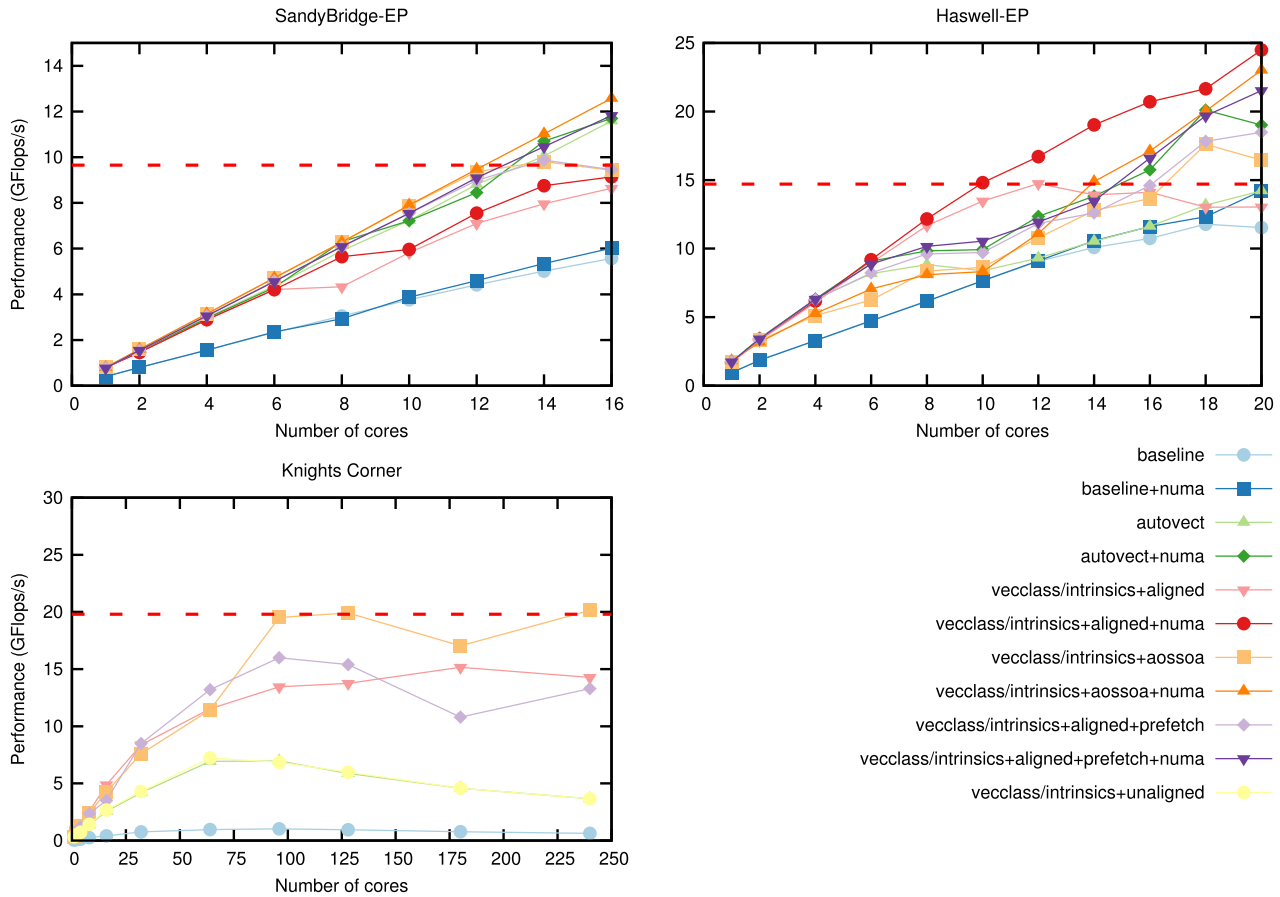


Fig. 7. Primary variable update kernel scalability performance on 10^6 grid cells.

the corresponding residuals, and to the normals of the interfaces. The computations performed at each interface are: the evaluation of Euler fluxes for the left and right states – Eqs. (2) and (3), evaluation of Roe averages and corresponding eigen-vectors and eigen-values – Eqs. (A.2)–(A.10), assembly of the numerical fluxes, and accumulation of residuals. The kernel computes two external products (left and right Euler fluxes), and one 4×4 GAXPY. A judicious implementation requires 191 FLOPS and accesses 38 double precision values per direction sweep. The pointers to the left and right states, and corresponding residuals are accessed through indirect references in the baseline implementation. This makes the kernel directly reusable in an unstructured solver, but at the same time complicates the generation of auto-vectorised code. In fact, it is not possible to guarantee, at compile time, the absence of race conditions unless special arrangements are made with regard to the order by which the faces are visited. Furthermore, the arguments passed as left and right states in most situations are alias for the same pointer. This is a source of ambiguity and it too prevents auto-vectorisation.

4.2.1. Data-level parallelism

Code vectorisation has been achieved by evaluating a number of avenues such as autovectorisation via compiler directives, intrinsics and the VCL vector class, similar to the cell-wise flow variable update kernel.

In order for the code to be vectorised by the compiler, a number of modifications had to be performed. First of all, the indirect references for obtaining the neighbours indices were removed and their position explicitly computed for each iteration. As these increase in a linear fashion by a stride of 1 once their offset is computed, the compiler was then able to replace gather and scatter instructions

with unaligned vector loads and stores, albeit with the help of the OpenMP4.0 linear clause. This had a particularly large effect on Knights Corner where performance of unaligned load and stores was a factor of two higher compared to gather and scatter operations. Furthermore, the `restrict` qualifier was also necessary for any degree of vectorisation to be attempted by the compiler. The autovectorised version (**autovect** – Fig. 9) brought forth a 2.5X improvement on SandyBridge-EP, 2.2X on Haswell-EP and 13X on Knights Corner when compared to their respective baseline.

In multiblock codes, it is natural to group the faces according to the transformed coordinate that stays constant on their surface, e.g. *i*-faces and *j*-faces. Thus fluxes can be evaluated in two separate sweeps visiting *i*-faces or the *j*-faces, respectively. Each operation consists of nested *i*- and *j*-loops. When visiting *i*-faces, if the left and right states are stored with unit stride, they cannot be simultaneously aligned on vector register boundary and prevent efficient SIMD execution. When visiting *j*-faces this problem does not appear, as padding is sufficient to guarantee that left and right states of all *j*-faces can be simultaneously aligned on vector register boundary. The **intrinsics-unaligned** and **vector-unaligned** versions use unaligned loads and stores when visiting *i*-faces, but aligned load/store instructions when visiting *j*-faces, for the reasons discussed above. The penalty for unaligned access on the *i*-faces is better tolerated on SandyBridge-EP and Haswell-EP, where inter-register movements can be performed with small latency, but is very damaging on Knights Corner, as already seen. Furthermore, the VPU ISA does not contain native instructions for unaligned vector loads and stores that are found in AVX/AVX2. To this extent, the programmer has to rely upon the `unpacklo`, `unpackhi`, `packstorelo` and `packstorehi` instructions and their corresponding intrinsics. Examples of intrinsic unaligned load and store functions used in this work for Knights Corner

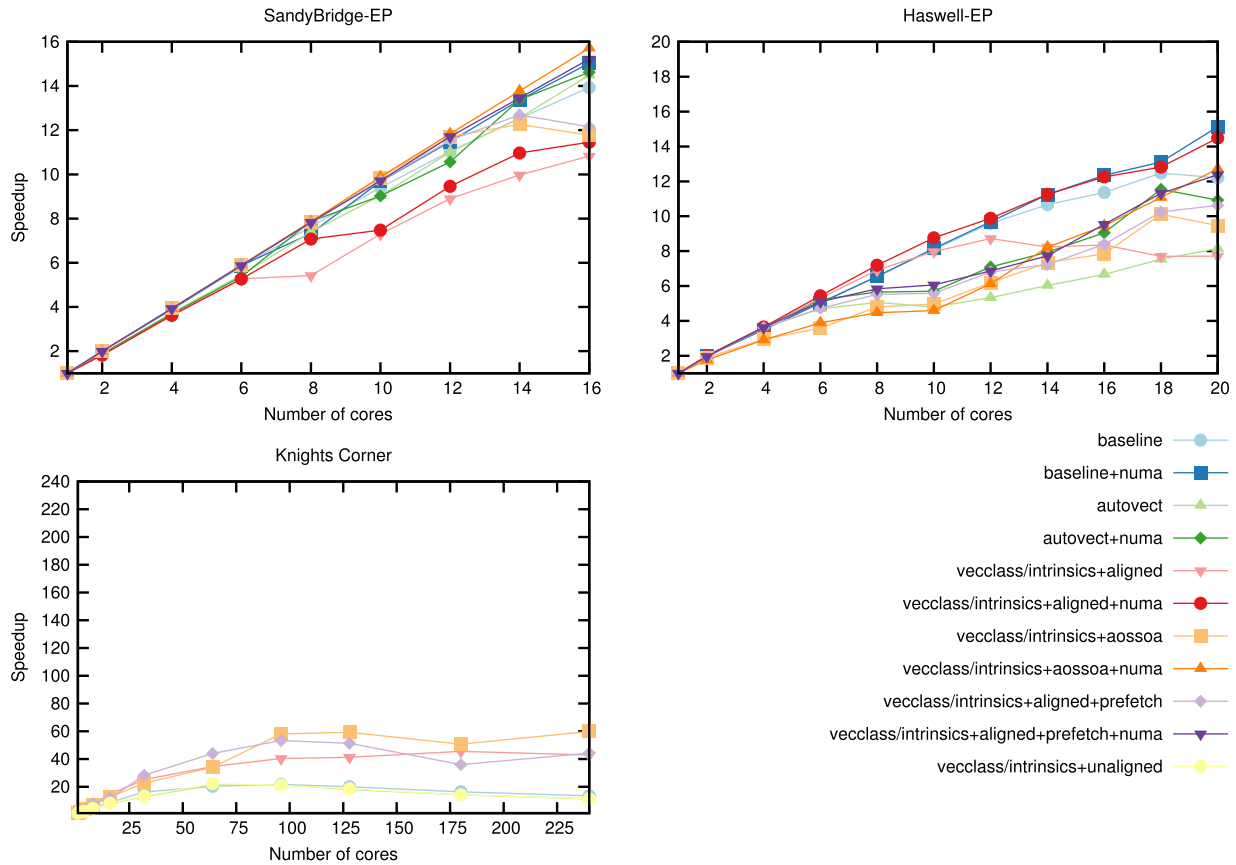


Fig. 8. Primary variable update kernel scalability speedup on 10^6 grid cells.

can be seen in code snippet 4.2.1 and 4.2.1 which requires two aligned load/store and register manipulations hence the penalty in performance compared to their aligned counterparts.

```

1 inline __m512d loadu(double *src)
2 {
3     __m512d ret;
4     ret = _mm512_loadunpacklo_pd(ret, src);
5     ret = _mm512_loadunpackhi_pd(ret, src+8);
6     return ret;
7 }

```

Listing 3: Unaligned SIMD loads on KNC

```

1 inline void storeu(double *dest, __m512d val)
2 {
3     _mm512_packstorelo_pd(dest, val);
4     _mm512_packstorehi_pd(dest+8, val);
5 }

```

Listing 4: Unaligned SIMD stores on KNC

The lack of alignment for the arguments of the i -faces loop can be resolved by two techniques. Shuffling or dimensionally lifted transposition. Shuffling consists in performing the loads using register-aligned addresses, and then modifying the vector registers to position correctly all the operands. On AVX, shuffling requires three instructions as register manipulations can only be performed on 128 bit lanes. The kernel using intrinsics and register shuffle (**intrinsics + shuffle** – Fig. 9) issues aligned load and store operations combined with register vector shuffling operations when handling i -faces. The use of register shuffling also saves a load at each iteration, as the current right state can be re-used as left state in the next iteration therefore allowing for register blocking.

The optimisation of the vector register rotations and shifts, however, is critical. On AVX, register rotation was achieved by

using `permute2f128` followed by a shuffle. AVX2 supports the `vpermd` instruction which can perform cross-lane permutations, therefore a more efficient approach can be used. The Intel Xeon Phi also supports `vpermd` although the values have to be cast from `epi32` (integers) to the required format (double in the present case). The choice of the intrinsics for each operation is based on its latency and throughput: functions with throughput smaller than one can be performed by more than one execution port, if these are available, leading to improved instruction level parallelism. As an example, the AVX/AVX2 `blend` intrinsic has a latency of one cycle and a throughput of 0.3 on Haswell-EP, and one cycle latency and throughput 0.5 on SandyBridge-EP, therefore it is preferable to the AVX `shuffle` instruction, which has one cycle latency, but unit throughput on both chips. The code snippets below show how shuffling can be achieved on the three architectures.

```

1 // v1 = 3 2 1 0
2 // vr = 7 6 5 4
3 __m256d t1=_mm256_permute2f128_pd(v1, vr, 33);
4 // t1 = 5 4 3 2
5 __m256d t2=_mm256_shuffle_pd(v1, t1, 5);
6 // t2 = 4 3 2 1

```

Listing 5: Register shuffle for aligned accesses on the i -face sweep with AVX.

```

1 // v1 = 4,3,2,1
2 // vr = 8,7,6,5
3 // blend = 4,3,2,5
4 __m256d blend = _mm256_blend_pd(v1, vr, 0x1);
5 // res = 5,4,3,2
6 __m256d res = _mm256_permute4x64_pd(blend,
7     _MM_SHUFFLE(0,3,2,1));

```

Listing 6: Register shuffle for aligned accesses on the i -face sweep with AVX2.

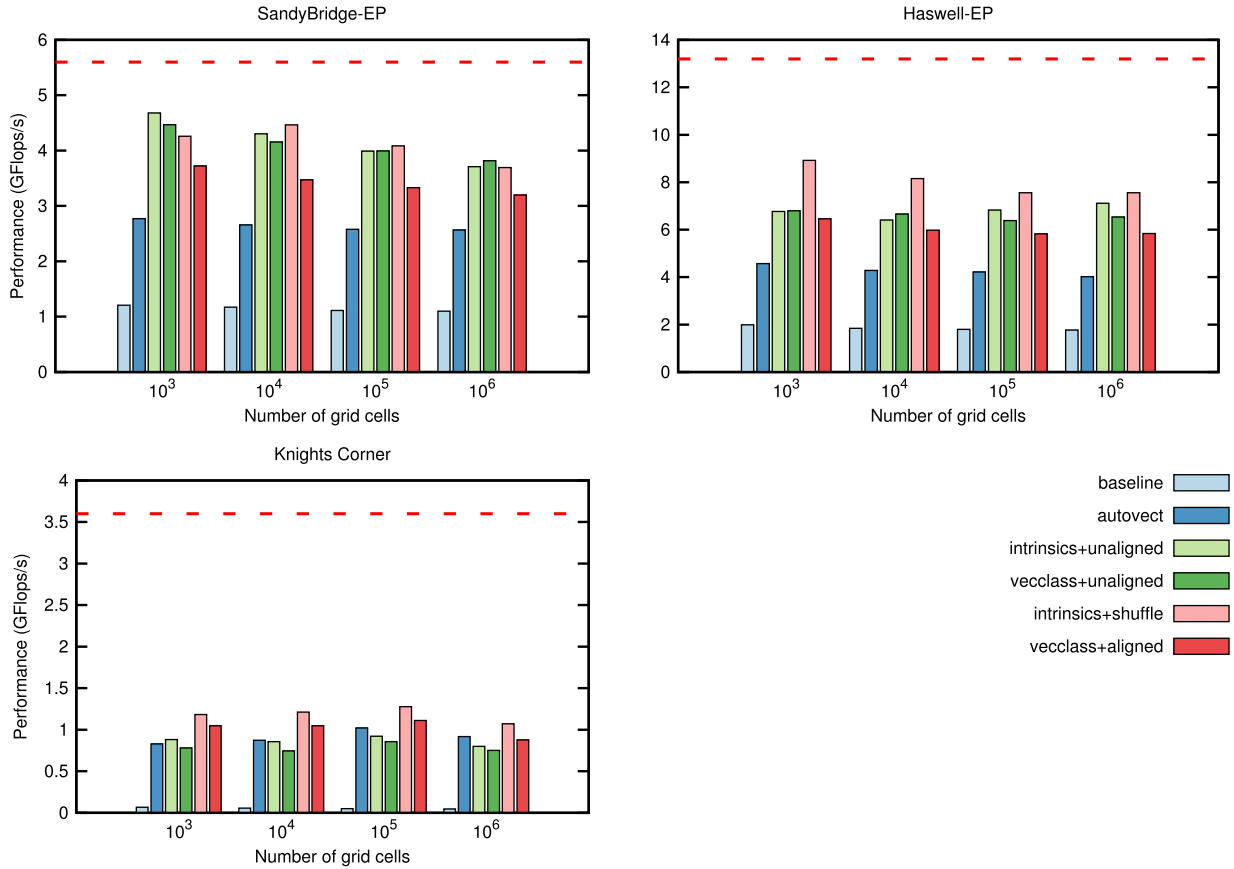


Fig. 9. Flux computations SIMD optimisations with rooflines.

```

1 // v1 = 8,7,6,5,4,3,2,1
2 // vr = 16,15,14,13,12,11,10,9
3 __m512i idx =
4   {2,3,4,5,6,7,8,9,10,11,12,13,14,15,0,1};
5 // blend = 8,7,6,5,4,3,2,9
6 __m512d blend = _mm512_mask_blend_pd(0x1, v1, vr
7   );
8 // res = 9,8,7,6,5,4,3,2
9 __m512d res = _mm512_castsi512_pd(
10   _mm512_permutevar_epi32(idx,
11   _mm512_castpd_si512(blend)));

```

Listing 7: Register shuffle for aligned accesses on the *i*-face sweep with MIC VPU.

Generating aligned load/stores in the VCL library (**vecclass-aligned** – Fig. 9) can be performed by using the `blend4d` method which requires as argument an index shuffle map. However, this was found to be less efficient than the authors' **intrinsics** functions since an optimised implementation targeting each architecture was not available.

The unaligned **intrinsics** version delivers a 4X speedup over the baseline implementation on SandyBridge-EP and Haswell-EP and 13.5X on Knights Corner. Comparison with the autovectorised version sees a 60% speedup on SandyBridge-EP and 50% on Haswell-EP although no noticeable improvements on the coprocessor. The increase in performance for the **intrinsics** version on the multicore processors is due to the manual inner loop unrolling when assembling fluxes which allows for more efficient instruction level parallelism. The **intrinsics** and **shuffle** kernel perform similarly on SandyBridge-EP compared to the unaligned version due to AVX cross-lane shuffle limitations whilst performing 30% and 10% faster on Haswell-EP and Knights Corner. As earlier mentioned, the vector class versions perform worse on both multicore CPUs due to the

inefficient implementation of shuffling in the `blend4d` routine however on Knights Corner, these were replaced with the **intrinsics**-based kernels listed in 4.2.1 as they were not implemented in the VCLKNC library. For this reason, the aligned vector class version on the manycore processor delivers very similar performance compared to the hand-tuned **intrinsics + shuffle**.

Another technique for addressing the alignment stream conflict issue is via Henretty's dimension-lifted transposition [11] (**intrinsics-transp** – Fig. 10). Transposition solves the alignment conflict for *i*-faces by changing the stride of the left and right states. There is however a penalty in that the data has to be transposed before the *i*-face loop and re-transposed prior to the *j*-face sweep. Furthermore, since the result of the transposition is an array with the number of columns equal to that of the SIMD length (i.e. rectangular matrix transposition), the transposition kernel can have a degrading effect on performance as the working set is increased and cannot be held in cache. This can be seen across all architectures, especially SandyBridge-EP where the drop in performance is significant. On problems of small size transposition is more efficient than shuffling (SandyBridge-EP), because of the high cost of shuffling. For larger problems, however, the shuffling version was preferable.

4.2.2. Cache blocking

In the kernels tested up to this point, the loops visiting *i*-faces and *j*-faces have been kept separate. Data locality and temporality can be improved by fusing (blocking) the evaluation of fluxes across the *j*-face and the *i*-face of each cell. This technique is known as cache blocking and the corresponding results are shown as the data set **intrinsics-shuffle-cacheblocked** in Fig. 10. The cache blocking kernel fuses both *i*- and *j*-passes and it is based on the shuffling kernels described above for allowing aligned loads and

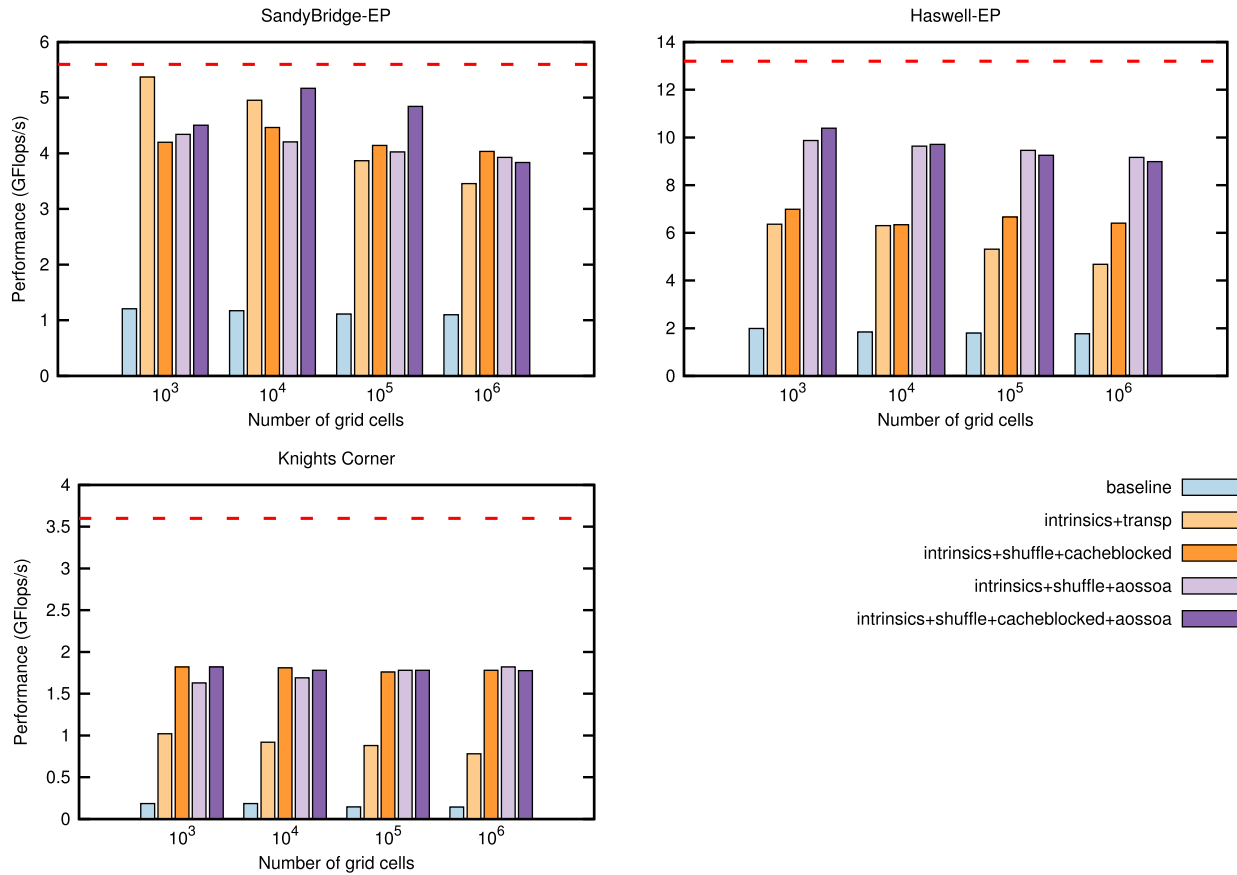


Fig. 10. Flux computations SIMD+Memory optimisations with rooflines.

stores on the i -faces. The cache blocking kernel further benefits from the fact that it can save an extra load/store operation when writing back results for each cell. It should be noted that in a multiblock code, a first level of cache blocking can be achieved by visiting the blocks in succession, rather than sweeping through the whole domain. The second level is obtained by nesting loops visiting i - and j -faces, but without re-arranging the data. The nesting of i - and j -face loops allows data to be reused before being evicted from the cache. The blocking factor should be a multiple of the SIMD vector length as to allow for efficient vectorisation. Across all three platforms, cache blocking did not bring forth any palpable benefits for single core runs. The reason for this is that although data cache reuse is increased, fusing both loops brings forth an increase in register pressure and degrades performance as the fused kernel performs 382 FLOPS for every iteration.

4.2.3. Data layout transformations

Discussion so far for the flux computations has assumed a SoA data layout format. A kernel using the AoSSoA data layout (**intrinsics-shuffle-aossoa** – Fig. 10) and based on the shuffling kernel was tested. The best performing sub vector size was the size of the SIMD register, just like for the primitive variables update kernel.

Similarly, the **intrinsics-shuffle-aossoa-cacheblocked** (Fig. 10) kernel uses the new hybrid data layout on top of the fused kernel.

4.2.4. Thread parallelism

For the purpose of studying performance at full chip and node concurrency, domain decomposition is performed within every block in order to avoid load-imbalance due to discrepancies in block size (see Fig. 11). Decomposition is performed in two ways.

One option is to split each block into slabs along the i - and j -planes depending on sweep, the number of slabs being equal to the number of available threads. The second option is to split each block into square tiles. This method increases locality and complements the cache-blocking processing of the flux reconstruction scheme discussed above. For the non-fused kernels where separate plane sweeps are performed, the first decomposition method was chosen due to the elimination of possible race conditions. For the fused kernels, the latter decomposition was required together with handling race conditions at the tile boundary.

Thread and process affinity has been applied similar to the flow variable update kernel i.e. `compact` for the multicore CPUs and `scatter` for the coprocessor. Task and chunk allocation have been performed manually as required by the custom domain decomposition within the OpenMP parallel region. Another reason for doing the above and not relying on the OpenMP runtime is due to the observed high overhead that this caused on the Knights Corner processor when running on more than 100 threads. Furthermore, all runs imply NUMA-aware placement via applying the first touch technique.

The performance on both SandyBridge-EP and Haswell-EP at full concurrency beats the roofline prediction attaining 53 GFLOPS and 101 GFLOPS respectively. On Knights Corner, the best performing version achieves 95% (94 GFLOPS) performance efficiency compared to its corresponding roofline when running on 180 threads. While on the multicore CPUs, the best performing kernels were based on the intrinsics-based cacheblocked and hybrid data layout optimisations, the scaling on the coprocessor of this particular kernel was second best. This can be attributed to register pressure as more threads get scheduled on the same core which compete for available resources. Our assumption can be validated by examining the scaling behaviour of the cacheblocked

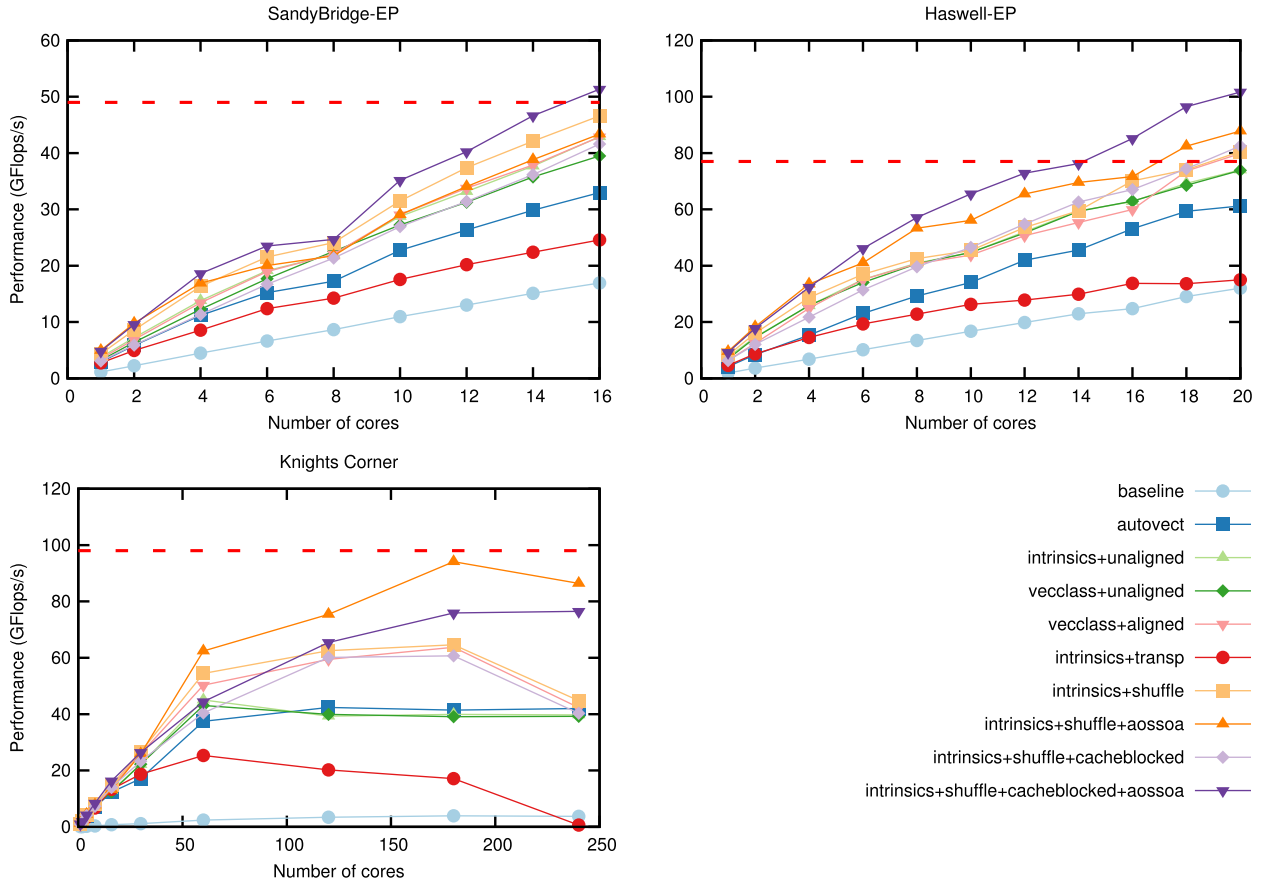


Fig. 11. Flux computations scalability performance on 10^6 grid cells.

kernel without the hybrid AoSSoA data layout which suffers from poor scalability as more than one thread gets allocated per core.

A key consideration to take into account is the fact that piercing through or achieving near parity to the performance roof on all architectures required both the advanced SIMD optimisations for efficient load store operations and memory optimisations such as cacheblocking and hybrid data layout transformations. The only exception is on the Haswell-EP where kernels implementing aligned SIMD with the SoA arrangement attained a speedup similar to the prediction model although not piercing through it. This can be attributed to the new core design that was focused on high SIMD throughput via the increase in back-port bandwidth and number of functional units.

Comparing best performing results to their respective baselines reveals a 3.1X speedup on SandyBridge-EP and Haswell-EP and 24X on the Knights Corner coprocessor at full concurrency.

If we were to analyse speedup metrics (Fig. 12), an interesting observation could be made. Perhaps surprisingly, the baseline kernel scales best although it is performing significantly worse compared to all others on the merits of actual attained compute performance. This further highlights the importance of choosing the correct metrics to judge computational performance on modern architectures with the aid of performance models and not solely on the basis of scalability. The slowest kernels will undoubtedly achieve superior speedup if efficient domain decomposition and load balancing are performed, as it was in our case, since there is ample room for available resources such as memory bandwidth and FLOPS that can be utilised. In reality however, the baseline kernel utilises approximately 30%–35% of the attainable performance on the multicore CPUs and only 5% on the coprocessor while the best optimised versions match or exceed it on all architectures.

4.3. Roofline visualisation

A validation of the Roofline model based on the results of the applied optimisations can be seen in Fig. 13. For brevity, the main classes of optimisations were grouped as *baseline*, *autovect* for autovectorised versions, *intrinsic+aligned* assuming best performing intrinsics version on that particular kernel and platform (i.e. shuffle, transposition, etc.) and *intrinsic+aligned* that encapsulates best performing aligned intrinsics and specific memory optimisation (cacheblocked, AoSSoA, etc.).

Observations on the single core diagrams highlight interesting aspects on the two computational kernels and their evolution in the optimisation space. For the cell-wise flow variable update, applying autovectorisation managed to pierce through the ILP wall on the multicore CPUs and the SIMD roofline on the coprocessor. This is due to the fact that SIMD execution automatically brings forth improvements in instruction level parallelism on the SandyBridge and Haswell CPUs due to loop unrolling as it was earlier mentioned. On Knights Corner, autovectorisation allowed for instructions to be routed towards the more efficient VPU unit. The reason as to why autovect performed as well as the hand-tuned kernels on all three platforms can also be evidenced by the fact that there is little to no optimisation space remaining as the kernel obtains 80%–90% efficiency and is limited by the reduced available memory bandwidth since only one memory channel is utilised by the core. For flux computations, the baseline sits above the multicore CPUs due to the considerable higher FLOP count of the kernel and better arithmetic intensity. Autovectorisation pierces through their respective SIMD wall whilst hand tuned intrinsics and subsequent memory optimisations deliver close to the attainable performance

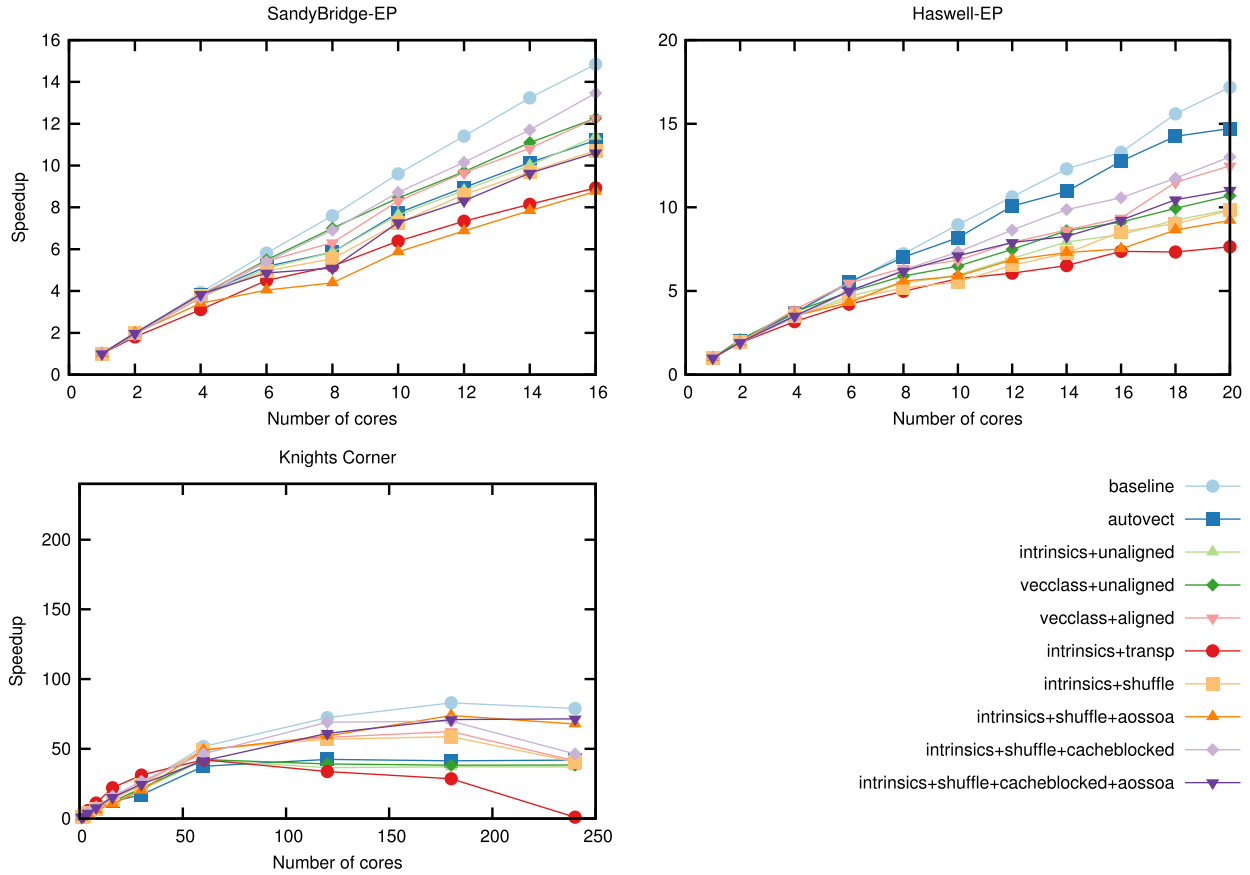


Fig. 12. Flux computations scalability speedup on 10^6 grid cells.

prediction. On Haswell-EP and Knights Corner however, speedup is limited by the fact that the FMA units are not fully utilised due to the algorithmic nature of PDE-based stencils and their inherent lack of fused mul/add operations. Even so, for single core runs, our best optimised version in the `intrin.+memopt` class achieves between 80%–90% of the available performance compared to the baseline which delivers 20% on the multicore CPUs and 1.67% on the coprocessor.

At full node and chip concurrency, the sizeable increase in memory bandwidth permits the baseline implementation to obtain a larger degree of speedup across all architectures. For the cell-wise flow variable update kernel, NUMA first touch optimisations applied to all versions allows them to bypass the NUMA optimisation wall. On the coprocessor, prefetching only works when coupled with SIMD due to the poor performance of the scalar processing unit compared to the VPU. As more memory bandwidth becomes available, subsequent optimisations such as the hybrid data layout transformation that prevents TLB misses and provides better data locality performs better than the autovectorised and SoA-based hand tuned intrinsics version. For flux computations, vectorisation through compiler directives and intrinsics delivers close to peak roof performance whilst subsequent memory optimisations bypass the model's prediction as earlier stated. On Haswell-EP, the baseline model is placed above the SIMD roof due to the significantly higher in-core bandwidth and the fact that the compiler is still able to vectorise some inner loops of the kernel i.e. flux assembly. The highest speedup can be seen on Knights Corner, bearing in mind the log scale nature of the axes where the provision of further memory optimisations such as data layout transformations to existing efficient SIMD kernels (i.e. via shuffling) delivers close to peak attainable performance.

5. Conclusions

We have applied a wide variety of optimisation techniques on two distinct computational kernels in a multiblock CFD code and evaluated their performance on three modern architectures. A detailed description was given on the exploitation of all levels of parallelism available in modern multicore and manycore processors through efficient code SIMDisation and thread parallelism. Memory optimisations described in this work included software prefetching, data layout transformations through hybrid data structures such as Array of Structures Structures of Arrays and multi-level cache blocking for the numerical fluxes.

The practicalities of enabling efficient vectorisation were discussed at great length. We have established that for relatively simple kernels such as the primary flow variable update, the compiler can generate efficient SIMD code with the aid of portable OpenMP 4.0 directives. This approach however does not fully extend to more complex kernels such as flux computations involving a stencil-based access pattern where best SIMD performance is mandated through the use of aligned load and store operations made possible via inter-register shuffles and permutations. Implementations of such operations were performed using compiler intrinsics and the VCL vector library and included bespoke optimisations for each architecture. Vectorised and non-vectorised computations exhibit a 2X performance gap for the flow variable kernel and up to 5X in the flux computations on the SandyBridge-EP and Haswell-EP multicore CPUs. The difference in performance is significantly higher on the manycore Knights Corner coprocessor where vectorised code outperforms the non-vectorised baseline by 13X in updating the flow variables and 23X for computing the numerical fluxes. These figures correlated with projections that future multicore and manycore architectures will bring forth further

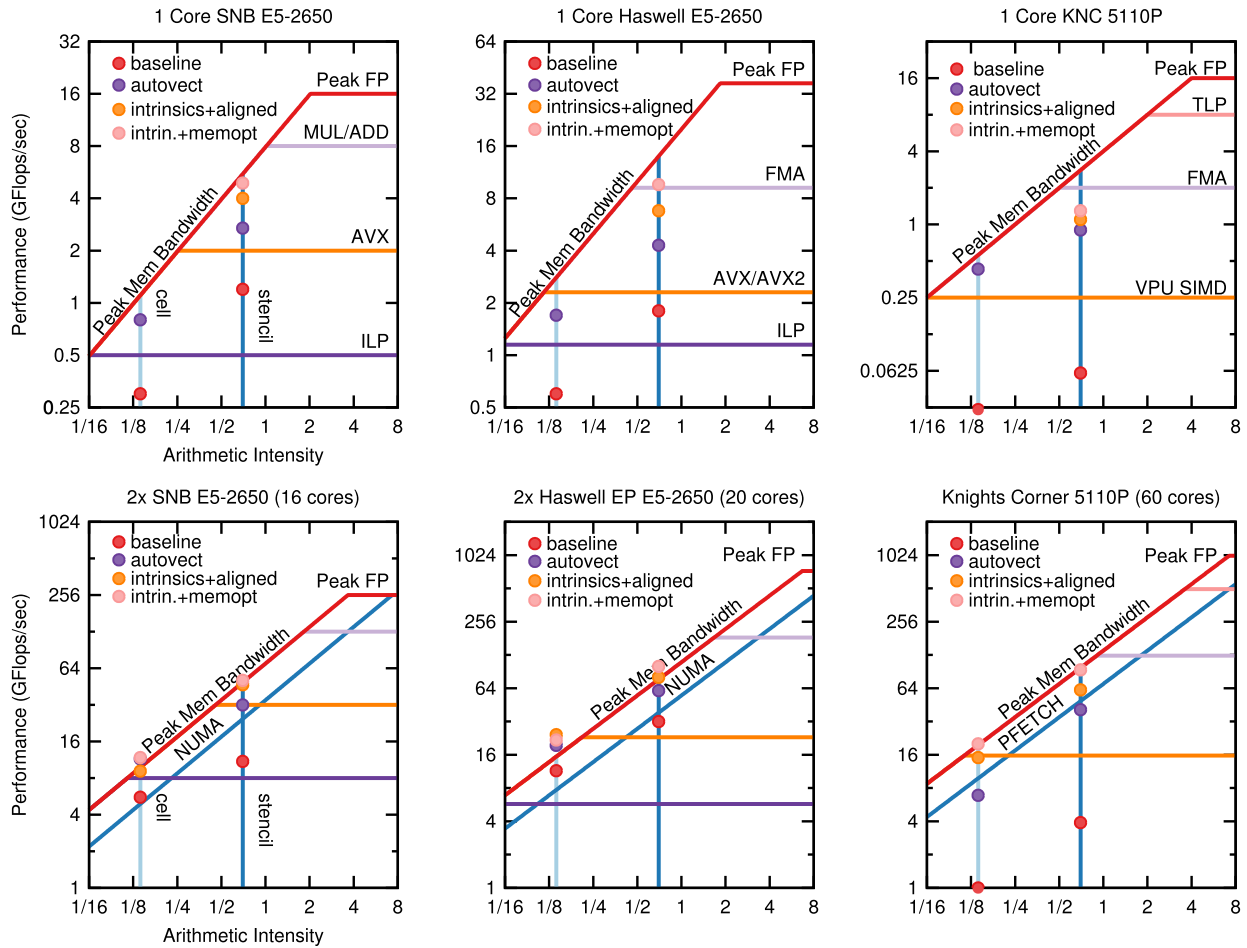


Fig. 13. Core (top) and node (bottom) roofline plots correlated with applied optimisations.

improvements to the width and latency of SIMD units mandates efficient code SIMDisation as a crucial avenue for attaining performance in numerical computations on current and future architectures.

Modifying the data layout from Structures of Arrays to a hybrid Array of Structures Structures of Arrays brought forth improvements for the vectorised kernels by minimising TLB misses when running on large grids and increasing locality. Performance gains were particularly noticeable when running at full concurrency and performed best on the Knights Corner coprocessor. Cache blocking on the flux computations coupled with the hybrid data layout delivered best results on the multicore CPUs when running across all available cores however performed second best on the coprocessor due to increased register pressure.

Core parallelism has been achieved through the use of OpenMP4.0 directives for the flow variable update kernel which offers mixed SIMD and thread granularity through common constructs. For the numerical fluxes, the domain was decomposed into slabs on the i and j faces for implementations that performed separate plane sweeps and into tiles for the fused sweep implementation with cache blocking. For the primary flow variable update we achieved a 2X speedup on SandyBridge-EP and Haswell-EP and 20X on the Knights Corner coprocessor when compared to their respective baselines. Computations of the numerical fluxes at full concurrency also obtained a 3X speedup on the multicore CPUs and 24X on the coprocessor over the baseline implementation.

The Roofline performance model has been used to appraise and guide the optimisation process with respect to the algorithmic nature of the two computational kernels and the three compute architectures. For single core execution, the optimised flow

variables update kernel achieved approximately 80% efficiency on the multicore CPUs and 90% on the coprocessor. Flux computations obtained 90% efficiency on SandyBridge-EP, 80% on Haswell-EP and approximately 60% on Knights Corner. The reason for the relatively low efficiency on the coprocessor is due to the out-of-order core design which requires at least 2 threads for fully populating the coprocessor VPU. The computational efficiency at full concurrency outperforms the model's predictions for both kernels across all three platforms the only exception being Knights Corner where flux computations deliver a maximum of 94% efficiency when running on 180 threads.

On a core-to-core comparison among the three processors, the Haswell-based Xeon E5-2650 core performs on average 2X and 4-5X faster compared to a single SandyBridge Xeon E5-2650 and Xeon Phi Knights Corner 5110P core across both kernels. However, at full node and chip concurrency, the two socket Haswell Xeon E5-2650 node is approximately on par with the Knights Corner coprocessor for flux computations and 25% faster for updating the primary flow variables. The Xeon Phi Knights Corner coprocessor and Haswell node outperform the two socket SandyBridge Xeon E5-2650 node by approximately a factor of two on flux computations and 50% to 2X on the flow variable update. However, on a flop per watt and flop per dollar metric, the Knights Corner 5110P coprocessor delivers superior performance compared to both multicore CPUs at the cost of higher development and optimisation time needed for exploiting its underlying features in numerical simulations. The increase in time spent on fine tuning the code on the coprocessor is attenuated by the fact that the majority of optimisations targeting fine and coarse grained levels of parallelism such as SIMD and threads are transferable when porting from multicore to manycore including GPGPUs.

Acknowledgements

This work has been supported by EPSRC and Rolls–Royce plc through the Industrial CASE Award 13220161. We are also very much indebted to Christopher Dahnken of Intel Corporation for his invaluable help and guidance on intrinsic optimisations and Agner Fog for the support provided in using the VCL library. Furthermore, we wish to thank Sophoclis Patsias and Paolo Adami of Rolls–Royce plc for approving publication clearance and Simon Burbidge of Imperial College London for providing access to Haswell-EP nodes.

Appendix. Further details

The physical fluxes are approximated with second order TVD–MUSCL [34–36] numerical fluxes

$$\mathbf{F}_{i-1/2,j}^* = \frac{1}{2} (\mathbf{F}_{i-1,j} + \mathbf{F}_{i,j}) - \frac{1}{2} \mathbf{R} |\mathbf{A}| \mathbf{L} (\mathbf{U}_{i,j} - \mathbf{U}_{i-1,j}) - \frac{1}{2} \mathbf{R} |\mathbf{A}| \Psi \mathbf{L} \Delta \mathbf{U}_{i-1/2,j}. \quad (\text{A.1})$$

The term $\mathbf{R} |\mathbf{A}| \Psi \mathbf{L} \Delta \mathbf{U}_{i-1/2,j}$ represents the second order contribution to the numerical fluxes. Ψ is the limiter and the flux eigenvectors and eigenvalues \mathbf{R} , \mathbf{A} , \mathbf{L} are evaluated at the Roe-average [36] state.

\mathbf{L} is the matrix of the left eigenvectors of the Roe-averaged flux Jacobian.

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & -\frac{1}{\tilde{a}^2} \\ 0 & t_m & t_\theta & 0 \\ 0 & n_m & n_\theta & -\frac{1}{\tilde{\rho}\tilde{a}} \\ 0 & n_m & n_\theta & -\frac{1}{\tilde{\rho}\tilde{a}} \end{bmatrix}. \quad (\text{A.2})$$

The evaluation of the numerical fluxes also requires the Roe-averaged eigen-values and right-eigenvectors of the flux Jacobian, which are evaluated as follows

$$\mathbf{A} = \text{diag} (\tilde{u}_n, \tilde{u}_n, \tilde{u}_n + \tilde{a}, \tilde{u} - \tilde{a}) + \varepsilon \quad (\text{A.3})$$

$$\mathbf{R} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & t_m & (\tilde{u}_n + \tilde{a})n_m & (\tilde{u}_n - \tilde{a})n_m \\ 0 & t_\theta & (\tilde{u}_n + \tilde{a})n_\theta & (\tilde{u}_n - \tilde{a})n_\theta \\ \tilde{k} & \tilde{u}_t & \tilde{h} + \tilde{u}_n\tilde{a} & \tilde{h} - \tilde{u}_n\tilde{a} \end{bmatrix} \quad (\text{A.4})$$

where ε is Harten's entropy correction [37]. The Roe-averaged state is defined by

$$\tilde{u}_m = \omega u_m^{i+1,j} + (1 - \omega) u_m^{i,j} \quad (\text{A.5})$$

$$\tilde{u}_\theta = \omega u_\theta^{i+1,j} + (1 - \omega) u_\theta^{i,j} \quad (\text{A.6})$$

$$\tilde{h} = \omega h^{i+1,j} + (1 - \omega) h^{i,j} \quad (\text{A.7})$$

$$\tilde{a}^2 = (\gamma - 1)(\tilde{h} - \tilde{k}) \quad (\text{A.8})$$

$$\tilde{\rho} = \sqrt{\rho^{i+1,j} \rho^{i,j}} \quad (\text{A.9})$$

$$\omega = \frac{\sqrt{\rho^{i+1,j}}}{\sqrt{\rho^{i+1,j}} + \sqrt{\rho^{i,j}}}. \quad (\text{A.10})$$

Similar definitions are applied for the $\mathbf{G}_{i,j-1}^*$ numerical flux vector.

References

- [1] P. Prabhu, T.B. Jablin, A. Raman, Y. Zhang, J. Huang, H. Kim, N. P. Johnson, F. Liu, S. Ghosh, S. Beard, T. Oh, M. Zoufaly, D. Walker, D.I. August, State of the Practice Reports, SC'11, ACM, New York, NY, USA, 2011, pp. 19:1–19:12.
- [2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiawicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzyniak, D. Wessel, K. Yelick, Commun. ACM 52 (2009) 56–67.
- [3] D.A. Patterson, J.L. Hennessy, Computer Organization and Design: The Hardware/software Interface, Newnes, 2013.
- [4] D.S. McFarlin, V. Arbatov, F. Franchetti, M. Püschel, Proceedings of the International Conference on Supercomputing, ICS'11, ACM, New York, NY, USA, 2011, pp. 265–274.
- [5] H. Sutter, J. Larus, Queue 3 (2005) 54–62.
- [6] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, P. Dubey, Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA'12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 440–451.
- [7] M. Bader, A. Breuer, W. Holz, S. Rettenberger, High Performance Computing Simulation (HPCS), 2014 International Conference on, pp. 193–201.
- [8] 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.
- [9] X. Tian, H. Saito, S.V. Preis, E.N. Garcia, S.S. Kozhukhov, M. Masten, A.G. Cherkasov, N. Panchenko, Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW'13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 1149–1158.
- [10] N.G. Dickson, K. Karimi, F. Hamze, J. Comput. Phys. 230 (2011) 5383–5398.
- [11] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, P. Sadayappan, Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software, CC'11/ETAPS'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 225–245.
- [12] S. Rostrup, H.D. Sterck, Comput. Phys. Comm. 181 (2010) 2164–2179.
- [13] C. Rosales, Extreme Scaling Workshop (XSW), 2013, pp. 1–7.
- [14] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC'08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 4:1–4:12.
- [15] J. Jaeger, D. Barthou, High Performance Computing (HiPC), 2012 19th International Conference on, pp. 1–10.
- [16] A. Schäfer, D. Fey, High Performance Computing for Computational Science – VECPAR 2012, in: Lecture Notes in Computer Science, vol. 7851, Springer, Berlin, Heidelberg, 2013, pp. 451–466.
- [17] S. Williams, L. Oliker, J. Carter, J. Shalf, Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11, ACM, New York, NY, USA, 2011, pp. 55:1–55:12.
- [18] M. Vavra, Aero-Thermodynamics and Flow in Turbomachines, John Wiley, Los Alamitos, CA, USA, 1960.
- [19] F. Grasso, C. Meola, Handbook of Computational Fluid Mechanics, Academic Press, London, 1996.
- [20] MIT, Mises, <http://web.mit.edu/drela/Public/web/mises/>, 2009, Accessed: 13-05-2015.
- [21] P. Gepner, V. Gamayunov, D.L. Fraser, Procedia Computer Science 4 (2011) 452–460. Proceedings of the International Conference on Computational Science, ICCS 2011.
- [22] W. Eckhardt, A. Heinecke, R. Bader, M. Brehm, N. Hammer, H. Huber, H.-G. Kleinhenz, J. Vrabec, H. Hasse, M. Horsch, M. Bernreuther, C. Glass, C. Niethammer, A. Bode, H.-J. Bungartz, in: J. Kunkel, T. Ludwig, H. Meuer (Eds.), Supercomputing, in: Lecture Notes in Computer Science, vol. 7905, Springer, Berlin, Heidelberg, 2013, pp. 1–12.
- [23] T. Piazza, H. Jiang, P. Hammarlund, R. Singhal, Technology Insight: Intel(R) Next Generation Microarchitecture Code Name Haswell, Technical Report, Intel Corporation, 2012.
- [24] J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Kaufmann, Boston, United States, 2013.
- [25] I.D. Zone, Avx-512 instructions, <http://software.intel.com/en-us/blogs/2013/avx-512-instructions>, 2013, Accessed: 04-03-2014.
- [26] S. Williams, A. Waterman, D. Patterson, Commun. ACM 52 (2009) 65–76.
- [27] G. Ofenbeck, R. Steinmann, V. Caparros, D. Spampinato, M. Puschel, Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on, pp. 76–85.
- [28] J.D. McCalpin, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter (1995) 19–25.
- [29] 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/>.
- [30] H.D. Bailey, F.R. Lucas, W.S. Williams, Performance Tuning of Scientific Applications, CRC Press, NY, United States, 2011.
- [31] J. Treibig, G. Hager, Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I, PPAM'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 615–624.
- [32] A. Fog, C++ vector library class, <http://www.agner.org/optimize/>, 2015, Accessed: 10-05-2015.
- [33] VCLKNC, <https://bitbucket.org/veclibknc/vclkn>, 2015, Accessed: 10-08-2015.
- [34] G. Albada, B. Leer, J. Roberts, W.W., in: M. Hussaini, B. Leer, J. Rosendale (Eds.), Upwind and High-Resolution Schemes, Springer, Berlin, Heidelberg, 1997, pp. 95–103.
- [35] C. Hirsch, Numerical Computation of Internal and External Flows, John Wiley and Sons, Chichester, West Sussex, UK, 1990.
- [36] P. Roe, J. Comput. Phys. 43 (1981) 357–372.
- [37] A. Harten, P.D. Lax, B. van Leer, SIAM Rev. 25 (1983) 35–61.