Computer Physics Communications 216 (2017) 53–62

Contents lists available at ScienceDirect

Computer Physics Communications

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

Hybrid multicore/vectorisation technique applied to the elastic wave equation on a staggered grid

Sofya Titarenko *, Mark Hildyard

School of Earth and Environment, University of Leeds, Leeds, LS2 9JT, UK

ARTICLE INFO

Article history:
Received 15 June 2016
Received in revised form 6 February 2017
Accepted 22 February 2017
Available online 1 March 2017

Keywords:
OpenMP
Vectorisation
Multicore
Elastic waves
Staggered grid

ABSTRACT

In modern physics it has become common to find the solution of a problem by solving numerically a set of PDEs. Whether solving them on a finite difference grid or by a finite element approach, the main calculations are often applied to a stencil structure. In the last decade it has become usual to work with so called big data problems where calculations are very heavy and accelerators and modern architectures are widely used. Although CPU and GPU clusters are often used to solve such problems, parallelisation of any calculation ideally starts from a single processor optimisation. Unfortunately, it is impossible to vectorise a stencil structured loop with high level instructions. In this paper we suggest a new approach to rearranging the data structure which makes it possible to apply high level vectorisation instructions to a stencil loop and which results in significant acceleration. The suggested method allows further acceleration if shared memory APIs are used. We show the effectiveness of the method by applying it to an elastic wave propagation problem on a finite difference grid. We have chosen Intel architecture for the test problem and OpenMP (Open Multi-Processing) since they are extensively used in many applications.

1. Introduction

Numerical modelling remains one of the most powerful tools in the research areas of physics and engineering. It is often impossible to find an analytical solution describing a physical process and a numerical approach is the only choice. In past years a large number of numerical methods have been developed and successfully applied, which can be roughly divided into five groups: finite difference, boundary element, finite element, finite volume, spectral and mesh free methods [1–6].

All problems involving partial differential equations (PDEs) can be computationally heavy. It is also often important to perform the calculation with a high degree of accuracy which increases the time required even more. That is why many research groups nowadays use accelerators to speed up calculations. The use of CPU (Computer Processing Unit) and GPU (Graphics Processing Unit) clusters is normal. However, we strongly believe that optimisation needs to be started from a lower level. Once the code is fully optimised for a single CPU node, one can move on to many CPUs, CPU/GPU cluster optimisation.

Most modern processors are multicore systems and support shared memory APIs (Application Programming Interface) and SIMD (Single Instruction Multiple Data) instructions. SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) extensions designed by Intel, AltiVec instruction set designed by Apple and IBM and NEON designed by ARM are the ones which support SIMD instructions. OpenMP and POSIX Threads are the most extensively used APIs nowadays.

The main load of numerical calculations of a PDE system is often placed on stencil type loops. Examples of GPU parallelisation of stencil computations on nVidia cards can be found in [7] and on modern Xeon and Xeon Phi systems in [8]. In [9] can be found an example of CUDA and OpenMP types of parallelisation. However, applying *vectorisation* methods to a stencil structure appears to be particularly difficult due to problems with applying the usual high level SIMD principles to stencil loops. In this paper we propose a new method of memory rearrangement which allows these difficulties to be overcome. The method does not require any knowledge of low level programming since high level instructions are used. We have chosen to demonstrate the method on an Intel Core i7 multicore system using OpenMP directives since this is one of the most popular choices in numerical modelling. However, our method can easily be applied using the other architectures and APIs mentioned above.

As a test problem we have chosen a wave propagating through an elastic medium. The wave equation is widely used in such physics areas as acoustics, electromagnetics, fluid dynamics and seismology. In all those areas it has become usual to work with so called big data problems. Whereas in fluid dynamics finite

* Corresponding author.
E-mail address: S.Titarenko@leeds.ac.uk (S. Titarenko).

http://dx.doi.org/10.1016/j.cpc.2017.02.022

element and spectral element methods are dominant, in acoustics and seismology the finite difference method is extensively used. The finite difference method was first applied to solve elastic wave propagation by [10] and used to generate synthetic seismograms by [11]. Generating synthetic data is also used in acoustics for sound field visualisation, see for example [12,13].

Reducing the second order wave equation to a system of first order differential equations is one of the most popular finite difference approach. One goes from an ordinary grid to a *staggered grid*. The method was first proposed in [14] to solve wave propagation problems and is proved to have better stability for 4th order accuracy schemes [15]. It has been developed further by [16], for anisotropic media [17] and in 3D space [18]. Examples of it's application can be found in full wave inversion problems [19], forward wave propagation modelling and synthetic seismic data modelling in geophysics [20–23] and modelling of acoustic wave propagation [24].

GPU parallelisation has been successfully applied to elastic wave propagation on a staggered grid, see [25–27]. Further examples of MPI (Message Passing Interface) parallelisation can be found in [28], hybrid MPI-OpenMP methods in [29] and parallelisation of acoustic wave propagation through OpenMP in [30].

2. Elastic wave equation on a staggered grid

The wave propagation equation for an elastic medium in 2D space can be written as

$$\rho\ddot{u} = (\lambda + 2\mu)\nabla\nabla u - \mu\nabla \times \nabla \times u, \tag{1}$$

where $u(\mathbf{x}, t)$ is a displacement function and ö means its second order time derivative, ρ represents density and λ and μ are Láme parameters.

The equation above can be reduced to the following system of first order differential equations

$$\begin{cases} \rho\dfrac{\partial \dot{u}_1}{\partial t} = \dfrac{\partial \sigma_{11}}{\partial x_1} + \dfrac{\partial \sigma_{12}}{\partial x_2}, \\ \rho\dfrac{\partial \dot{u}_2}{\partial t} = \dfrac{\partial \sigma_{22}}{\partial x_2} + \dfrac{\partial \sigma_{12}}{\partial x_1}, \\ \dfrac{\partial \sigma_{11}}{\partial t} = \left(\lambda + 2\mu\right)\dfrac{\partial \dot{u}_1}{\partial x_1} + \lambda\dfrac{\partial \dot{u}_2}{\partial x_2}, \\ \dfrac{\partial \sigma_{12}}{\partial t} = \mu\dfrac{\partial \dot{u}_1}{\partial x_2} + \mu\dfrac{\partial \dot{u}_2}{\partial x_1}, \\ \dfrac{\partial \sigma_{22}}{\partial t} = \lambda\dfrac{\partial \dot{u}_1}{\partial x_1} + \left(\lambda + 2\mu\right)\dfrac{\partial \dot{u}_2}{\partial x_2}, \end{cases} \tag{2}$$

where $\dot{u}_i$ is the velocity vector, σ_{ij} are stress tensor components and λ and μ are Láme parameters.

Eqs. (2) are solved on a finite difference staggered grid. Fig. 1 shows the difference between a nonstaggered and staggered grid. It is clear to see on the staggered grid σ_{ii}, σ_{ij} and $\dot{u}_i$ are calculated at separate grid points. Stresses and velocities are also calculated at different times according to the "leap frog" technique: (1) for each $(t_n - 1/2)$ time step we find velocities at the points shown in green on Fig. 1; (2) we find stresses for each t_n time step using the velocities that have been calculated previously. The points for stress evaluation are coloured red on Fig. 1.

We use a fourth order approximation for velocity and stress derivatives and first order for the time derivative. For example, the stress value σ_{11} can be found through

$$\begin{aligned} \sigma_{11}^t = \sigma_{11}^{t-1} &+ \frac{(\lambda + 2\mu)\Delta t}{24\Delta x_1}\left(\dot{u}_1^{i-2} - 27\dot{u}_1^{i-1} + 27\dot{u}_1^{i} - \dot{u}_1^{i+1}\right) \\ &+ \frac{\mu\Delta t}{24\Delta x_2}\left(\dot{u}_2^{j-2} - 27\,\dot{u}_2^{j-1} + 27\dot{u}_2^{j} - \dot{u}_2^{j+1}\right). \end{aligned} \tag{3}$$

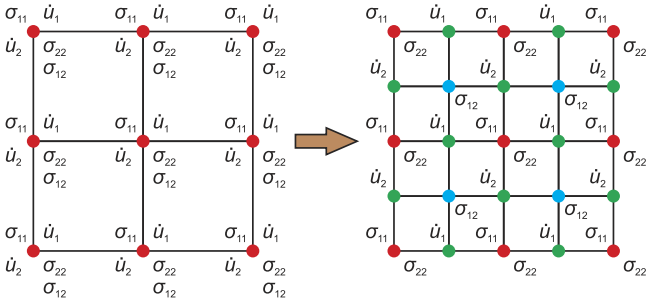

Fig. 1. Moving from nonstaggered grid to staggered grid, 2D case.

In the same way we can write down σ_{22} and σ_{12}:

$$\begin{aligned} \sigma_{22}^t = \sigma_{22}^{t-1} &+ \frac{\mu\Delta t}{24\Delta x_1}\left(\dot{u}_1^{i-2} - 27\dot{u}_1^{i-1} + 27\dot{u}_1^{i} - \dot{u}_1^{i+1}\right) \\ &+ \frac{(\lambda + 2\mu)\Delta t}{24\Delta x_2}\left(\dot{u}_2^{j-2} - 27\dot{u}_2^{j-1} + 27\dot{u}_2^{j} - \dot{u}_2^{j+1}\right), \end{aligned} \tag{4}$$

$$\begin{aligned} \sigma_{12}^t = \sigma_{12}^{t-1} &+ \frac{\Delta t\mu}{24\Delta x_2}\left(\dot{u}_1^{j-1} - 27\dot{u}_1^{j} + 27\dot{u}_1^{j+1} - \dot{u}_1^{j+2}\right) \\ &+ \frac{\Delta t\mu}{24\Delta x_1}\left(\dot{u}_2^{i-1} - 27\dot{u}_2^{i} + 27\dot{u}_2^{i+1} - \dot{u}_2^{i+2}\right). \end{aligned} \tag{5}$$

The equations for velocities are

$$\begin{aligned} \dot{u}_1^{t+1/2} = \dot{u}_1^{t-1/2} &+ \frac{\Delta t}{24\rho\Delta x_1}\left(\sigma_{11}^{i-1} - 27\sigma_{11}^{i} + 27\sigma_{11}^{i+1} - \sigma_{11}^{i+2}\right) \\ &+ \frac{\Delta t}{24\rho\Delta x_2}\left(\sigma_{12}^{j-2} - 27\sigma_{12}^{j-1} + 27\sigma_{12}^{j} - \sigma_{12}^{j+1}\right), \end{aligned} \tag{6}$$

$$\begin{aligned} \dot{u}_2^{t+1/2} = \dot{u}_2^{t-1/2} &\frac{\Delta t}{24\rho\Delta x_1}\left(\sigma_{12}^{i-2} - 27\sigma_{12}^{i-1} + 27\sigma_{12}^{i} - \sigma_{12}^{i+1}\right) \\ &+ \frac{\Delta t}{24\rho\Delta x_2}\left(\sigma_{22}^{j-1} - 27\sigma_{22}^{j} + 27\sigma_{22}^{j+1} - \sigma_{22}^{j+2}\right). \end{aligned} \tag{7}$$

We model the wave source as excitation of stresses σ_{11} and σ_{22} at the initial time. We apply Higdon absorbing boundary conditions (see for example [31]) of the first order

$$\frac{\partial\varphi}{\partial t} + C_p\frac{\partial\varphi}{\partial n} + \varepsilon \tag{8}$$

with $C_p = \sqrt{(\lambda + 2\mu)/\rho}$ as a P-wave velocity, φ unknown variables and ε a parameter chosen to increase the stability and absorption for the boundary.

3. Vectorisation

Automatic vectorisation has become a powerful method to enhance code performance on modern architectures. The most common modern architectures which provide support for SIMD instructions have already been mentioned. In this section we focus on Intel SIMD extensions as the architecture chosen for a test problem.

Without vectorisation switched on, a compiler uploads a *single* value in a SIMD register. SIMD extensions allow a software developer to apply a single arithmetic instruction to a vector rather than to a single value. A processor may contain several registers which can be used to perform an instruction on a small vector. If a processor has a SSE set of instruction, then there are 8 or 16 registers (128-bit) known as XMM0, …, XMM7 or XMM15. Thus four 32-bit single-precision floating point numbers can be stored and processed with the use of XMM registers. If AVX instructions are available the SIMD register file size is increased from 128 bits

to 256 bits and the registers are renamed as YMM0, …, YMM7 (or YMM15). In a similar way AVX-512 allows us to have 32 registers ZMM0,. . . , ZMM31 (512-bit).

To enable a general optimisation option one needs to compile a code with the `/O2` (or higher) flag (these flags are for Windows). However, higher levels of optimisation should be used with care as for a complicated code, options `/O3` and `/Ox` may not increase performance or may sometimes slow down calculations [32]. For processor-specific options (like SSE4.1, AVX, AVX-512) the corresponding flags (`/QxSSE4.1`, `/QxAVX`, `/QxMIC-AVX512`) should be used.

Intel has released a number of different microarchitectures, so it is possible now to take advantage of different memory structures [33]. For example the *Nehalem* structure is the basis for the first Core i5 and i7 processors. The instruction type is XMM, which means 128 bit SIMD registers and allows four 32-bit floats to be uploaded. Modern Core i7 processors often use *Haswell* or *Skylake* architecture, which include the AVX extension for 256 bit and 512 bit operations. To take advantage of the microarchitecture one needs to compile with `/QxAVX` switched on (different microarchitectures use different flags, which need to be verified on the Intel Website). If applying Assembly Code all the registers available could be used in the most efficient way. However this paper is focused on implementing high level instructions. It makes the method universally applicable to any type of a modern architecture (which supports SIMD operations) without going into low level code.

It must be kept in mind that vectorisation is not applicable to all loops. There is a set of rules and recommendations one should follow (see Intel documentation and vectorisation guide [34]). We list here the requirements for our specific application of wave propagation.

One of the main requirements for vectorisation is that access to memory should be contiguous. For a 128-bit register an Intel compiler may load four 32-bit float numbers if they are located in adjacent memory. Assuming `N` to be a multiple of 4 and vectors `a`, `b` and `c` are aligned, then a loop of the type

```
// loop with a unit stride
for(int i = 0; i < N; i++){
  a[i] = b[i] + c[i];
}
```

is automatically vectorisable after applying the directives mentioned above and with flag `/O2` (or higher) switched on.

However, if the stride in the loop is not unit

```
// loop with  stride = 2
for(int i = 0; i < N; i += 2){
  a[i] = b[i] + c[i];
}
```

the vectorisation may become less efficient. The vectorisation fails if we have a loop of the type

```
for(int i = 0; i < N; i++){
  a[i] = b[i] * b[i + 3];
}
```

Let us look at this loop in more detail. If we unroll the loop we get

```
for(int i = 0; i < N - 4; i += 4){
  a[i]   = b[i]   * b[i+3];
  a[i+1] = b[i+1] * b[i+4];
  a[i+2] = b[i+2] * b[i+5];
  a[i+3] = b[i+3] * b[i+6];
}
```

A SIMD register cannot upload all four values of vector `b` and apply a SSE or AVX instruction to them because the machine needs value `b[i+5]` to calculate value `a[i+2]` and `b[i+6]` to calculate `a[i+3]`. In the loop applied for a stencil structure we meet exactly the same problem which prevents successful vectorisation (see Section 4).

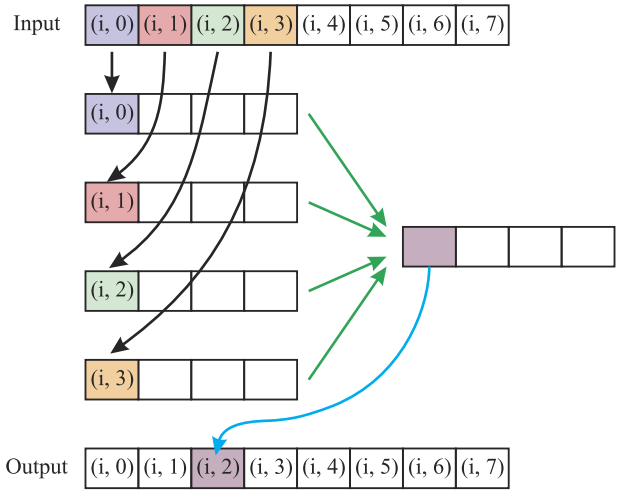

Fig. 2. Performing calculations with single values. Memory is unaligned. 4 unaligned loads.

4. Memory organisation

For the 2D wave propagation problem we need to store all variables as matrices. As computer memory can be represented as a 1D array of elements, the standard approach is to stitch rows of these matrices (as it is done for C/C++ code). From formulas (3)–(7) it follows that to find a derivative along the x coordinate we need to use four neighbouring values in memory. Suppose, we want to calculate σ_{11} values. According to formula (3) we need to find velocity derivatives $\frac{\partial \dot{u}_1}{\partial x}$ and $\frac{\partial \dot{u}_2}{\partial y}$. Let us calculate $\frac{\partial \dot{u}_1}{\partial x}$ derivative at a point i. According to (3) we have to use values of $\dot{u}_1$ velocity at points $\{i-2, i-1, i, i+1\}$.

Now we want to utilise SIMD 128-bit instructions (XMM registers). The simplest approach shown in Fig. 2 is to load each of these four data elements, in our example $\dot{u}_1$ values at $\{i-2, i-1, i, i+1\}$, to four XMM registers using function `_mm_load_ss`. In this case one element of the input vector is loaded to the lowest element of a register while the other three elements of the register are set to zero. The advantage of this function is that the memory address of the input element should not be aligned on any particular boundary. The disadvantage is that we need to perform four load operations and use only one 32-bit element of a 128-bit register.

Ideally we want to load four elements in a register using one instruction. So to find four derivatives at a time we can try to load four 128-bit vectors as it is shown in Fig. 3. In our example, to find a derivative $\frac{\partial \dot{u}_1}{\partial x}|_{x=i}$ we load all four values of $\dot{u}_1$ at the points $\{i-2, i-1, i, i+1\}$ in a SIMD register. SSE allows us to use function `_mm_load_ps` for this purpose. However there is a requirement that the address of the first element of the input vector should be aligned on 128-bit boundary. Therefore loading of the first vector to the first register can be implemented if the address is aligned. However the other three load operations cannot be implemented with the `_mm_load_ps` instruction as the corresponding addresses of the first elements of these vectors are not within 128-bit boundary. In other words, a compiler cannot apply `_mm_load_ps` to the next vector $\{i-1, i, i+1, i+2\}$ and for every velocity value at a point $\in \{i-1, i, i+1, i+2\}$ vector instruction `_mm_load_ss` will be used. As a result we will have one aligned load instruction and 12 unaligned instructions (versus 16 unaligned instructions if we use the previous method shown in Fig. 2).

Fortunately we can overcome this bottleneck by storing elements from different areas of a matrix in adjacent memory (the idea has been first suggested in [35]). In Fig. 4 we use data from four

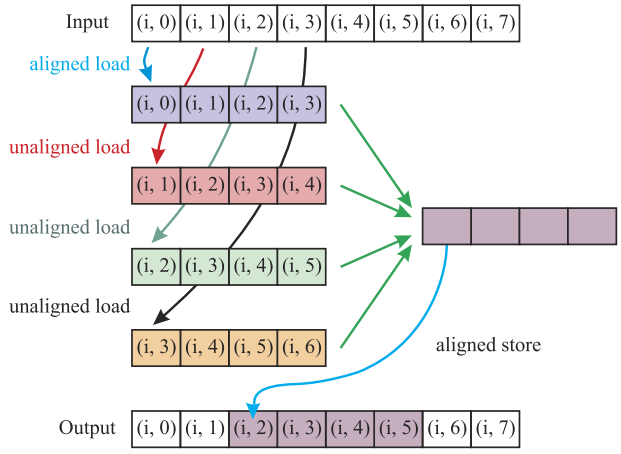

Fig. 3. Performing SIMD instructions. One aligned and 3 unaligned vectors. 1 aligned and 12 unaligned loads.

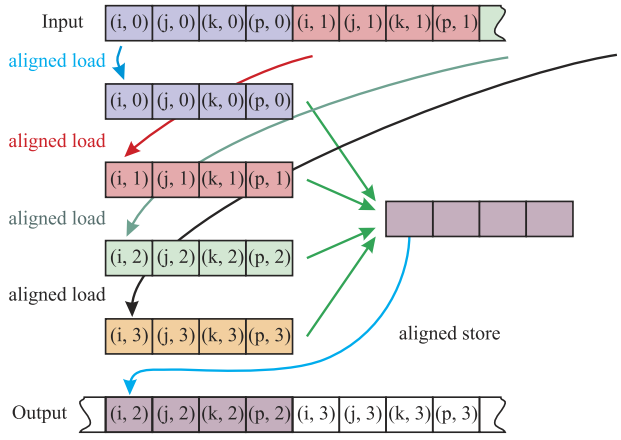

Fig. 4. Performing SIMD instructions. Memory is aligned. 4 aligned loads.

different rows of a matrix. As a result we have 4 load instructions `_mm_load_ps` and all elements of registers are utilised. In other words, instead of keeping *single* values $\{\dot{u}_1^{i-2}, \dot{u}_1^{i-1}, \dot{u}_1^{i}, \dot{u}_1^{i+1}\}$ used for calculating a *single* derivative $\frac{\partial \dot{u}_1}{\partial x}|_{x=i}$ next to each other we store *four* values of $\dot{u}_1^{i-2}$, *four* values of $\dot{u}_1^{i-1}$, *four* values of $\dot{u}_1^{i}$ and *four* values of $\dot{u}_1^{i+1}$ next to each other and therefore can calculate *four* derivatives $\frac{\partial \dot{u}_1}{\partial x}$ at the points $\{i-1, i, i+1, i+2\}$ at a time (see Fig. 4).

To get a clearer picture of the suggested memory rearrangement it is convenient to "colour" elements according to their location in a SIMD register. As started above, we work with 32-bit floats and 128 bit registers. This means we can fit in four 32-bit floats and therefore we can "colour" all the data we have in four different colours (see Fig. 5). As shown on the same figure, all adjacent elements in memory would be "coloured" in one of 4 colours (since each of them would be potentially uploaded in one of 4 possible locations of a SIMD register). Such memory organisation should provide us with a desirable improvement in performance.

However, when working with real problems we often have to apply boundary conditions. As we want to use only vector operations it is easier to create some virtual elements for each of these four "coloured" matrices (see Fig. 5, and Fig. 11). At every time step "real" data needs to be copied to the corresponding "virtual" data. If a problem is blocked (corresponding examples are mentioned below) it is more convenient to work with "virtual" blocks rather than with "virtual" points.

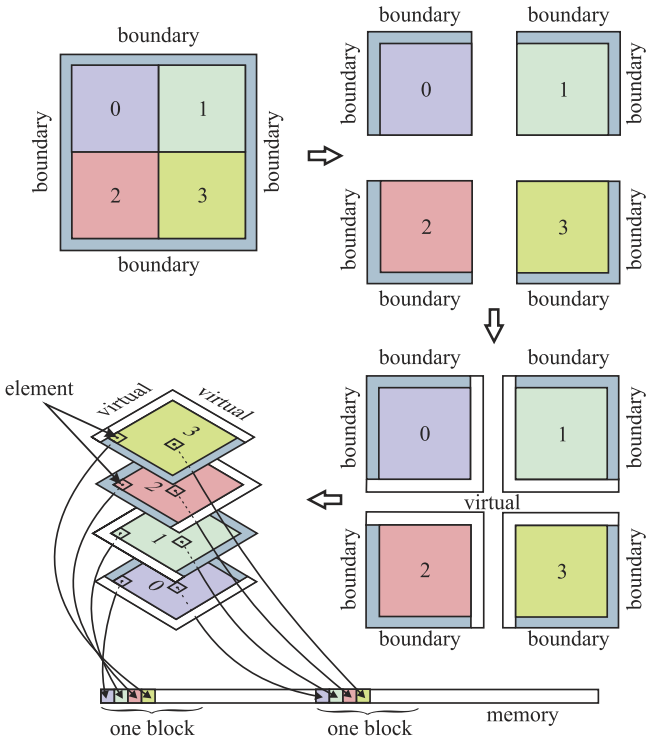

Fig. 5. Rearrangement of memory. Each matrix of variables is split onto four matrices and extra virtual elements are added. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

All modern architectures have a complicated memory structure. Together with DRAM it includes several levels of fast memory (caches). Caches may or may not be shared between the cores. For example the Intel Core i7-8930K has 12 MB of L3 cache shared between all the cores, 6 × 256 KB of L2 cache and 6 × 32 KB of L1 cache. L3 cache is the slowest, but the largest, L1 cache is the smallest, but with the highest bandwidth. Well optimised programs use fast memory as much as they can.

The classical way to create a cache aware algorithm is to block data into chunks which can fit into fast memory. It has to be mentioned that calculating derivatives on a stencil structure requires the introduction of "halo" points (see Fig. 6). Adding extra rows/columns of elements from each side of a block allows us to process all data for each block independently from others. Of course, before each finite difference operation can be performed, the values for these extra elements should be found by copying from neighbouring blocks (see Fig. 7). We have experimented with different sizes of block and also tested our approach on simple benchmark problems.

As a test problem we chose calculation of the 10th order derivative along the x direction. For the 10th order finite difference approximation we use values at 10 grid points. We run (1) a plain version, (2) a blocked version without the suggested memory rearrangement, and (3) a blocked version with the suggested memory rearrangement. We define an algorithm as *plain* if it uses a classical dynamic memory allocation. We define an algorithm as *loop-blocking* if it allocates memory as an array of dynamically allocated arrays (blocks). We define an algorithm as *plain +stride* if it is not blocked, but has rearranged memory. We call the algorithm *loop-blocking + block-strided* if it is blocked and has rearranged memory (for the code details please see Supplementary Material in Appendix A)

Fig. 8 shows a cache aware roofline model (examples of roofline models can be found in [36,37]) built for all these four cases on

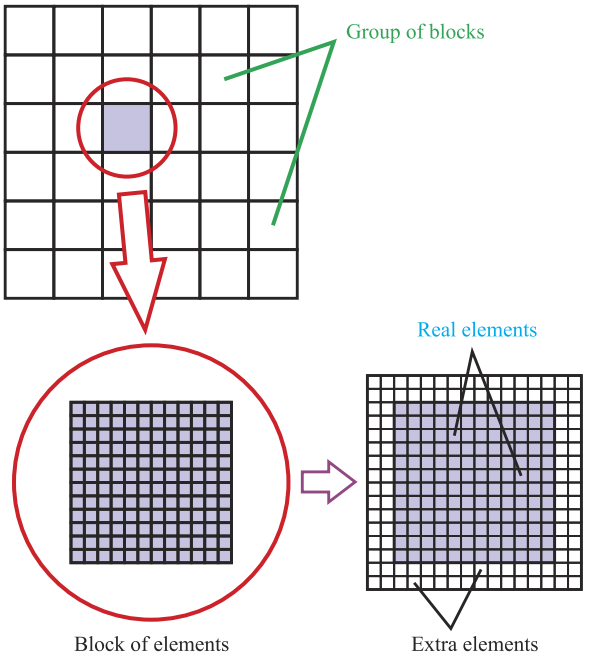

Fig. 6. Grid of blocks, each block contains extra elements which are copied from neighbouring blocks. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

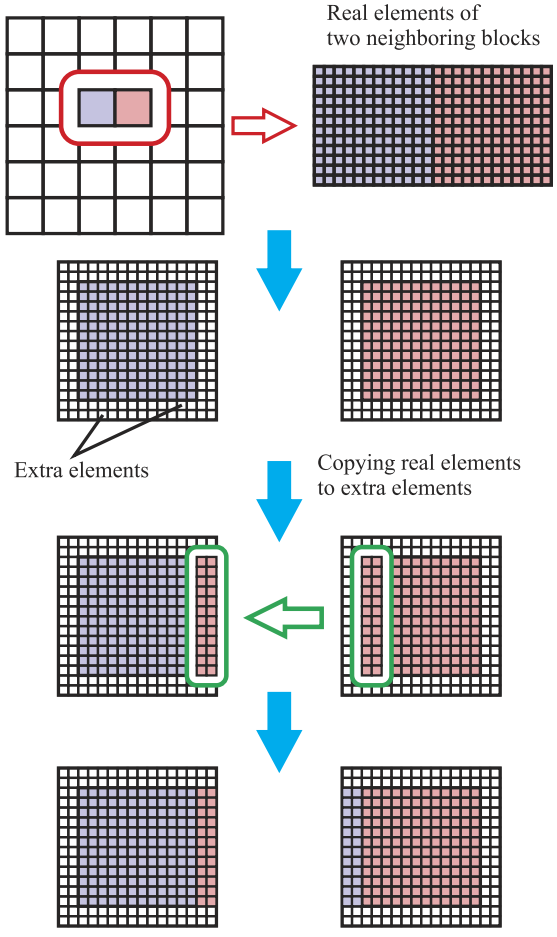

Fig. 7. Data copying between neighbouring horizontal blocks.

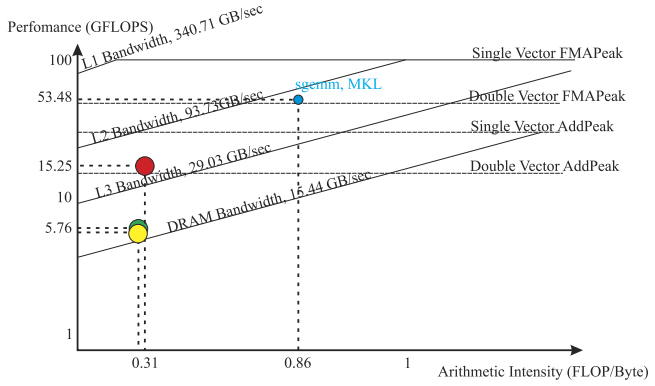

Fig. 8. Roofline model for plain (yellow dot), loop-blocking (green dot) and *loop-blocking* + *block-strided* (red dot) algorithms. Blue dot represents MKL function sgemm which calculates matrix C as $C = \alpha A \cdot B + \beta C$.

an Intel Core i7-8930K machine running on a single thread. On the same graph we present the result from profiling MKL function `sgemm`. We decided to use this function as an additional performance benchmark. All MKL functions are well optimised and of course well vectorised and give good understanding the potential capabilities of a processor.

It is clear to see that blocking alone does not give a considerable advantage over non blocking algorithms. However memory rearrangement gives us considerable improvement in the performance. Table 1 shows the results of a VTune Amplifier profiler for two different architectures. It can be seen that memory rearrangement makes the problem memory bound while algorithms without memory rearrangement are core bound. Overall the suggested memory rearrangement accelerates the benchmark model by about 5 times on both architectures. It is possible the performance of either algorithm could be improved through further processor-specific tuning. However, the aim of this paper is propose a *simple* solution which sufficiently accelerates the code within high level instructions and using only a compiler options switches and is therefore universal for any modern processor.

Let us have a closer look at the level of vectorisation for *loop-blocking* and *loop-blocking* + *block-strided* algorithms. The core function in both of them is the one calculating the 10th order derivative. Profiling the code with a VTune Amplifier provides us with an estimate of CPU time for all the functions. Table 2 shows that the 10th order derivative calculation takes about 90% of the CPU time. It shows, for a *loop-blocking* algorithm, changing switch from `/O1` to `/O3` does not give any improvement. This proves that the loop performing the heaviest calculations remains poorly optimised/vectorised. However, for the case of *loop-blocking* + *block-strided* algorithm going from `/O1` to `/O3` gives ≈2.85 times acceleration. Since according to Intel `/O2` and higher switches enable automatic vectorisation we may conclude this performance improvement occurs because of a higher level of vectorisation. Indeed, after activating automatic vectorisation the internal loop (see Fig. 9) will become completely unrolled.

Supplementary Material in Appendix A provides us with Assembly code for 10th order derivative calculation loop for all four examples listed in Table 2. It is interesting to see that our recommended *loop-blocking* + *block-strides* algorithm provides us with better results even for `/O1` optimisation flag (see Table 2).

5. Multithreading with OpenMp

Modern processors have multiple cores and can run several threads at a time. To achieve better performance each thread should access its own area of memory in order to avoid possible

Table 1
VTune Amplifier Microarchitecture analysis for two different architectures. Red dots represent *loop-blocking + block-strided* algorithms. Blue dots represent *loop-blocking* algorithms. Pink highlights indicate bottlenecks identified by VTune Amplifier. (For interpretation of the references to colour in this table, the reader is referred to the web version of this article.)

Architecture	Memory bound, %	Core bound, %	Retiring,%	Elapsed time, s
Skylake ●	24.9	0.012	55.3	7.868
Skylake ●	0.9	12.1	86.2	33.181
Haswell ●	10.2	21.8	63.9	8.515
Haswell ●	2.0	24.0	73.3	40.556

Table 2
VTune Amplifier analysis for *strided* (red dot) and *plain* (blue dot) algorithms. The problems run on Intel Core i5-6400 (Skylake architecture) with switches /O3 and /O1. Columns represent CPU time spent on the function performing the heaviest calculations. (For interpretation of the references to colour in this table, the reader is referred to the web version of this article.)

Algorithm	Switch	CPU time, %	CPU time, s
●	/O3	90.7	6.5
●	/O3	88.6	26.0
●	/O1	89.0	18.6
●	/O1	89.1	26.2

```
#pragma simd
for (int k = 0; k < 4; k++){
    v1[k] += C1*(v2[k]      - v2[k-4])    +
             C2*(v2[k+4]    - v2[k-2*4]) +
             C3*(v2[k+2*4]  - v2[k-3*4]) +
             C4*(v2[k+3*4]  - v2[k-4*4]) +
             C5*(v2[k+4*4]  - v2[k-5*4]);
}
```

Fig. 9. Internal loop in a 10th order derivative calculation completely unrolls with automatic vectorisation switched on.

conflicts with other threads. Here we come again to the classical idea of data blocking. This time, however, the aim is to parallelise the code on a multicore system.

The OpenMP application programming interface (API) is one of the most popular implementations of multithreading used to parallelise codes in HPC (High Performance Computing). It is based on spreading the work amongst the available threads. OpenMP is a free package, including directives and libraries, developed by OpenMP ARB (Architecture Review Board) with contributions from software developers such as IBM, Intel and others. Their web-site (http://openmp.org/wp/) provides a list of rules and examples to help to program, optimise and run an application successfully on different operating systems.

We split memory on a grid of blocks, see Fig. 6. Each block contains a matrix of elements and each element corresponds to four "colours" from original matrix. When running on a multicore processor we are going to treat every block independently from the others so we have to add to every block "halo" points from the left, right, top and bottom. The number of points depends on the order of derivative we have to calculate.

This approach is called geometrical decomposition and is a classical way of parallelising within MPI on a cluster (see for example [23,28]). It can also be used on a multicore system within OpenMP (examples can be found in [9,38,39]).

Depending on topology there could be different ways of exchanging data between blocks. For example, for a Cartesian 2D grid the approach pictured on Fig. 10 could be used. In this section we discuss how to combine multithreading with suggested memory rearrangement and achieve even better performance.

When applying a classical OpenMP approach to rearranged memory structure the method need some modification. As mentioned in the previous section, for real life problems we often need to apply boundary conditions. The rearranged memory structure would apply boundary conditions to internal blocks. This can be avoided by introducing "virtual" blocks (see Fig. 11). The idea comes from the classical data exchange between blocks widely used in OpenMP and MPI. However, it should be remembered that elements belonging to "virtual" blocks are no longer located next to each other like they are in a classical algorithm. "Virtual" elements are "spread over" all the data set as is shown in Fig. 5.

We have tested the approach of combining memory rearrangement with OpenMP multithreading on calculating a 10th order benchmark problem. We compared the results for the same problem but with an ordinary memory structure. From Fig. 12 it is clear that the suggested memory rearrangement gives the best advantage over classical memory structure when running on one thread. However, even when running on the maximum number of cores (the problem has been tested on an Intel Core i7-8930K processor) the suggested memory reorganisation still provides us with $\approx$3.66 times faster results then a classical approach.

Fig. 13 shows roofline model for 10th order problem running on maximum number of threads (12 threads) available on Intel Core i7-8930K processor.

The comparison of two graphs, Figs. 8 and 13, once more shows the best improvement in performance the algorithm gives when running on a single thread. However, it still demonstrates $\approx$1.42 times higher performance. We believe it could be improved farther with microarchitecture tuning and low level instructions.

We have profiled both benchmarks with a VTune Intel Amplifier 2017. The results of profiling shows again that the problem with a classical memory structure is core bound while the suggested approach gives high memory bound values. It may be concluded that the better performance for the *loop-blocking + block-strided* algorithm has been achieved through a much higher level of vectorisation. It is possible that both benchmarks can be further tuned for a specific processor with low level instructions. However, this is not the main focus of this research.

As previously mentioned, most modern processors have a complicated memory structure which together with DRAM includes several levels of caches (fast memory). The latest generations of Intel processors share L3 cache between the cores. However, the faster L2 and L1 caches are separated and belong to a single core. This could result in slowdown in performance because of threads migrating from one core to another and attempts to access data from a distant core local memory (NUMA issues). One of

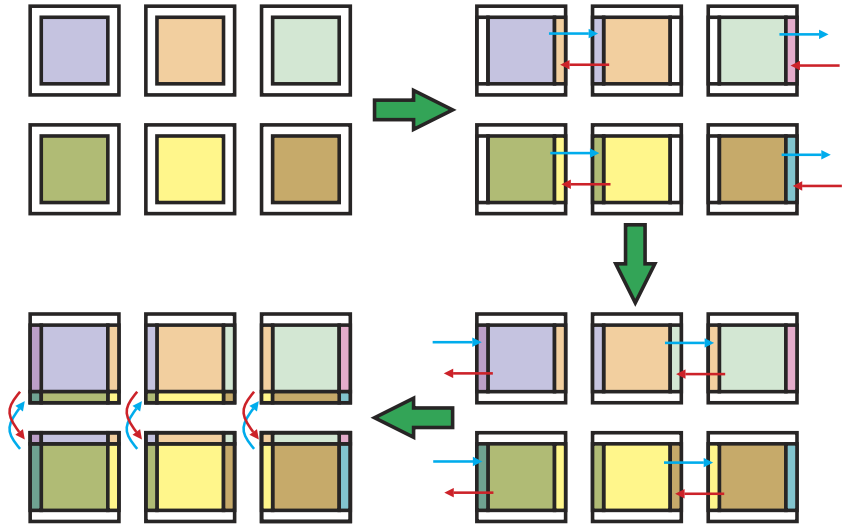

Fig. 10. Copying data between blocks in a grid. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

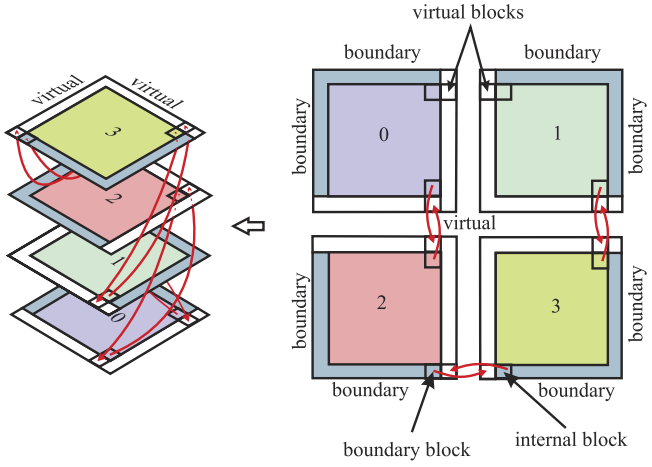

Fig. 11. Exchange between virtual blocks and real blocks. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

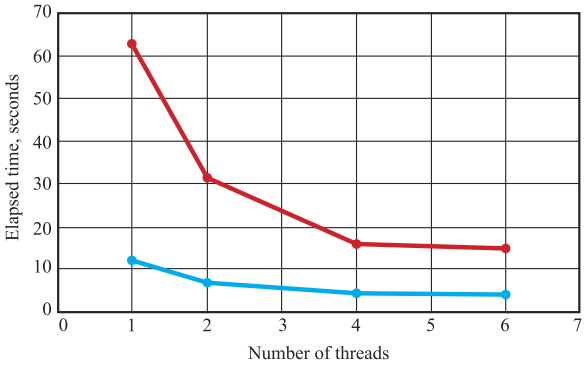

Fig. 12. Acceleration of the calculation of a 10th order problem using multithreading: red line is for classical memory structure; blue line is for rearranged memory.

the ways to resolve this situation is to "pin" a thread to a specific core. We have found that for our problem setting variable `KMP_AFFINITY=scatter,verbose` gives a slight improvement over default settings.

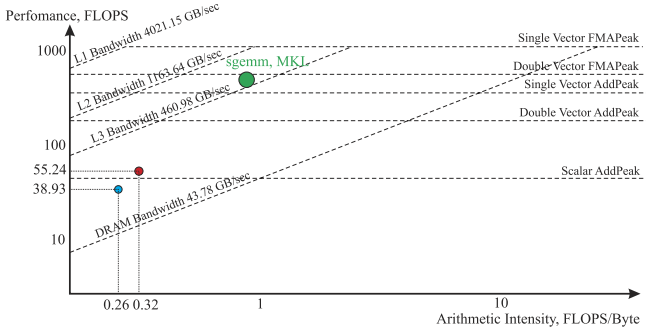

Fig. 13. Roofline for calculation of a 10th order problem using multithreading: blue dot is for classical memory structure; red dot line is for rearranged memory; green dot is MKL results for sgemm function.

6. Application to an elastic wave propagation problem

6.1. Running on a single thread

From Eqs. (3)–(7) it is easy to see that the heaviest part of the calculation for elastic wave propagation consists of calculating derivatives. We have experimented with 4th order derivatives. Applying similar logic for the 4th order case we can write the loop calculating a $\frac{\partial \hat{u}_1}{\partial x}$ derivative as shown on Fig. 14.

The loop for a corresponding derivative in the y direction can be found in Supplementary Material (see Appendix A).

It should be mentioned it is always possible to choose `Nx` divisible by 4. Then the derivative along the y coordinate loop could always be vectorised with high level instructions. However, this is not the case for x coordinate derivative and the loop in Fig. 14 can be vectorised only with our suggested memory rearrangement.

We have tested the elastic wave propagation problem on Intel Core i7-8930K (Haswell architecture) and Intel Core i5-6400 (Skylake architecture) machines. The results for a single thread are presented in Table 3.

Table 3 shows the suggested memory rearrangement gives considerable acceleration. It also highlights the problematic areas for both algorithms (coloured in pink). Note that a classical memory structure Intel Core i7-8930K (Haswell architecture) gives a better balance than Intel Core i5-6400 (Skylake architecture).

Table 3
VTune Amplifier microarchitecture analysis for two different architectures for the wave propagation problem. Red dots represent algorithms with our suggested memory arrangement, blue dots represent algorithms with standard memory arrangement. Pink highlights show bottlenecks detected by the analysis. (For interpretation of the references to colour in this table, the reader is referred to the web version of this article.)

Architecture	Memory bound, %	Core bound, %	Retiring, %	Elapsed time, s
Skylake ●	46.3	6.9	46.2	87.518
Skylake ●	5.4	11.5	78.0	310.0
Haswell ●	26.2	23.2	36.7	87.539
Haswell ●	13.8	14.9	53.2	230.25

Table 4
VTune Amplifier profiling on multiple threads. Wave propagation problem. Time spent copying data in between blocks. Blue dot represents OpenMP implementation with *loop-blocking* structure, red dot represents OpenMP algorithm for *loop-blocking + block-strided* structure. *Bottom* label represents times spent on copping bottom ↔ top "halo" points; *Left* represents times spent on copping left ↔ right "halo" points; *virtual* label shows times spent on copying real → virtual points. (For interpretation of the references to colour in this table, the reader is referred to the web version of this article.)

Number of threads		Elapsed time, sec	CPI	Bottom, sec	Left, sec	Virt, sec	CPU time, sec
●	1	68.68	0.67	4.30	4.12	2.73	108.48
	2	36.33	0.68	4.83	4.37	3.00	92.67
	4	21.82	1.05	7.32	5.11	3.83	104.88
	6	18.83	1.07	11.0	6.44	5.05	131.44
●	1	246.82	0.50	1.60	8.77		543.41
	2	124.18	0.44	1.77	10.19		363.18
	4	64.30	0.40	2.54	16.33		313.42
	6	47.08	0.42	3.61	23.67		327.75

```
void mbDiff_x(float *v1, float *v2){
int ind;
for (int i = 0; i < Nblock_y; i++){
  for (int j = 0; j < Nblock_x; j++){
  float *vec1, *vec2;
  ind = 4*((i + pad_y_start)*Nx
          + pad_x_start + j);
  vec1 = v1+ind;
  vec2 = v2+ind;

  __assumed_aligned(vec1,128);
  __assumed_aligned(vec2,128);
#pragma simd
    for (int k = 0; k < 4; k++){
      vec1[k] += vec2[k-2*4] - vec1[k + 4]
              + 27.0f * (vec2[k] - vec2[k-4]);
    }
  }
}
```

Fig. 14. Loop calculating 4th order derivative along x direction.

6.2. Running on a multicore system

We have run the elastic wave propagation problem on an Intel Core i7-8930K (Haswell architecture) and present the results in Table 4. The Table shows CPU times spent in copying in between the neighbouring blocks. It can be seen that the process of copying takes a small proportion of the whole CPU time (≈17.48% in total for the *loop blocking + block-stride* algorithm and ≈3.68% in total for the *loop-blocking* algorithm when running on 6 threads). It can be seen for our suggested *loop blocking + block-stride* algorithm CPI rate increases with the number of threads. Nevertheless, the suggested algorithm still gives good acceleration (≈2.5 times for the case of running on 6 threads) over a standard blocking algorithm. We believe even higher acceleration is achievable through further processor-specific tuning.

Fig. 15 and Table 5 show the values of performance for main calculation functions for the wave propagation problem with and without memory rearrangement (and their position on a roofline model graph). It is interesting to see, that performance of calculating a derivative differs for directions x and y. This can be explained by the fact that the values we use to calculate the y derivative are located further from each other in memory than ones used to calculate the derivatives along the x direction.

In the wave propagation problem we have implemented an algorithm where a derivative is calculated in the following way:

- for every four points a finite difference sum is found using `diff_x`, `diff_x_plus`, `diff_y` or `diff_y_plus`;
- for every four points a sum of type $C \cdot \sum_{i=0,3}$ `vec1[i]` is calculated and added to the previously stored values (using function `addC`).

It can also be seen that the residual procedure gained less from memory rearrangement algorithm.

7. Conclusions

In this paper we have presented a new approach for parallelisation of a finite difference code on a single processor. We

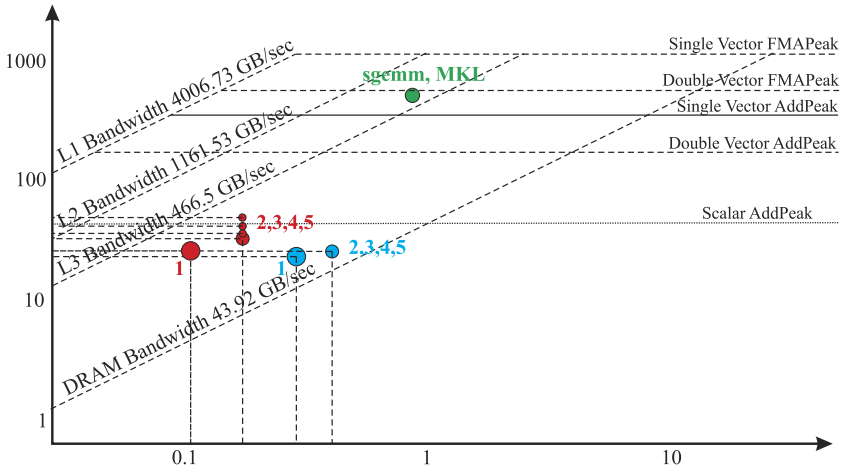

Fig. 15. Roofline for calculation elastic wave propagation problem using multithreading: blue dots are for classical memory structure (main functions); red dots are for rearranged memory (main functions); green dot is MKL results for `sgemm` function.

Table 5
Values of performance and CPU times for the most heavy functions of wave propagation problem running on multithreads. Performance is measured with Intel Advisor 2017, CPU time is estimated by VTune Amplifier. Blue dot represents OpenMP implementation with *loop-blocking* structure, red dot represents OpenMP algorithm for *loop-blocking* + *block-strided* structure. (For interpretation of the references to colour in this table, the reader is referred to the web version of this article.)

Functions		Performance, GFLOPS	AI GByte/sec	CPU time, sec
●	`diff_x`	51.96	0.200	8.910
	`diff_x_plus`	35.53	0.200	13.051
	`diff_y`	44.63	0.200	10.050
	`diff_y_plus`	39.22	10.324	11.000
	`addC`	28.41	0.120	151.563
●	`diff_x`	26.04	0.40	126.311
	`diff_x_plus`	25.53	0.40	126.955
	`diff_y`	26.32	0.40	123.534
	`diff_y_plus`	25.80	0.40	123.003
	`addC`	23.51	0.29	363.830

have discussed the difficulties of vectorisation when applied to a stencil structure and its hybrid application to shared memory APIs. While we have demonstrated the efficiency of the code in application to an elastic wave propagation problem we would like to highlight that the approach can be effectively applied to *any* general calculation on a stencil structure (or to a loop with non unit stride). Moreover, with the tendency towards more complicated hardware and increased capacity of registers one can expect this rearrangement of the data structure to become even more important in utilising of the increased advantage of vectorisation.

Acknowledgements

The authors would like to acknowledge support from RCUK under grant NE/L000423/1. The work is co-funded by the Natural Environment Research Council, the Environment Agency and Radioactive Waste Management Ltd. (RWM), a wholly-owned subsidiary of the Nuclear Decommissioning Authority. ST would like to thank Igor Kulikov from Novosibirsk State University and Andy Nowacki from University of Leeds for fruitful discussion.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.cpc.2017.02.022.

References

[1] D. Braess, Finite Elements, third ed., Cambridge University Press, ISBN: 9780511618635, 2007. http://dx.doi.org/10.1017/CBO9780511618635. Cambridge Books Online.
[2] M. Griebel, M.A. Schweitzer (Eds.), Meshfree Methods for Partial Differential Equations, sixth ed., Springer, ISBN: 9783642329791, 2007. http://dx.doi.org/10.1007/978-3-642-32979-1.
[3] J.C. Strikwerda, Finite Difference Schemes and Partial Differential Equations, second ed., SIAM, ISBN: 978-0-898716-39-9, 2004.
[4] G. Liu, Y. Gu, An Introduction to Meshfree Methods and their Programming, Springer, ISBN: 978-1-4020-3228-8, 2005.
[5] S. Gopalakrishnan, A. Chakraborty, D. Roy Mahapatra, Spectral Finite Element Method, Springer, ISBN: 978-1-84628-356-7, 2008.
[6] G. Beer, I.M. Smith, C. Duenser, The Boundary Element Method with Programming, Springer, ISBN: 978-3-211-71576-5, 2008. http://dx.doi.org/10.1007/978-3-211-71576-5.
[7] M. Krotkiewski, M. Dabrowski, Parallel Comput. (ISSN: 0167-8191) 39 (10) (2013) 533–548. http://dx.doi.org/10.1016/j.parco.2013.08.002.
[8] C. Andreolli, P. Thierry, L. Borges, G. Skinner, C. Yount, in: J. Reinders, J. Jeffers (Eds.), High Performance Parallelism Pearls, Morgan Kaufmann, Boston, ISBN: 978-0-12-802118-7, 2015, pp. 377–396. http://dx.doi.org/10.1016/B978-0-12-802118-7.00023-6.
[9] A. Arteaga, D. Ruprecht, R. Krause, The Fourth European Seminar on Computing (ESCO 2014), Appl. Math. Comput. (ISSN: 0096-3003) 267 (2015) 727–741. http://dx.doi.org/10.1016/j.amc.2014.12.055.
[10] Z. Alterman, F.C. Karal, Bull. Seismol. Soc. Am. 58 (1968) 367–398.
[11] K.R. Kelly, R.W. Ward, S. Tritel, R.M. Alford, Geophysics 41 (1976) 2–27.
[12] S. Sakamoto, T. Seimiya, H. Tachibana, Acoust. Sci. Technol. 23 (1) (2002) 34–39. http://dx.doi.org/10.1250/ast.23.34.

[13] S. Siltanen, P.W. Robinson, J. Saarelma, J. Pätynen, S. Tervo, L. Savioja, T. Lokki, J. Acoust. Soc. Am. 135 (6) (2014) EL344–EL349. http://dx.doi.org/10.1121/1.4879670.
[14] J. Virieux, Geophysics 51 (4) (1986) 889–901. http://dx.doi.org/10.1190/1.1442147.
[15] P. Moczo, J. Kristek, E. Bystrický, Stud. Geophys. Geod. (ISSN: 1573-1626) 44 (3) (2000) 381–402. http://dx.doi.org/10.1023/A:1022112620994.
[16] A.R. Levander, Geophysics 53 (11) (1988) 1425–1436. http://dx.doi.org/10.1190/1.1442422.
[17] H. Igel, P. Mora, B. Riollet, Geophysics 60 (4) (1995) 1203–1216. http://dx.doi.org/10.1190/1.1443849.
[18] R.W. Graves, Bull. Seismol. Soc. Am. 86 (4) (1996) 1091–1106.
[19] E.B. Raknes, B. Arntsen, W. Weibull, Geophys. J. Int. 202 (3) (2015) 1877–1894. http://dx.doi.org/10.1093/gji/ggv258.
[20] N. Khokhlov, N. Yavich, M. Malovichko, I. Petrov, 4th International Young Scientist Conference on Computational Science, Procedia Comput. Sci. (ISSN: 1877-0509) 66 (2015) 191–199. http://dx.doi.org/10.1016/j.procs.2015.11.023.
[21] D. Michéa, D. Komatitsch, Geophys. J. Int. 182 (1) (2010) 389–402. http://dx.doi.org/10.1111/j.1365-246X.2010.04616.x.
[22] H. Klimach, P.S. Roller, J. Utzmann, C.-D. Munz, Parallel Computational Fluid Dynamics 2007: Implementations and Experiences on Large Scale and Grid Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-540-92744-0, 2009, pp. 339–345. http://dx.doi.org/10.1007/978-3-540-92744-0_42.
[23] Y. Zhang, J. Gao, J. Appl. Geophys. (ISSN: 0926-9851) 109 (2014) 281–291. http://dx.doi.org/10.1016/j.jappgeo.2014.08.007.
[24] M. Hernández, B. Imbernón, J.M. Navarro, J.M. García, J.M. Cebrián, J.M. Cecilia, Comput. Electr. Eng. (ISSN: 0045-7906) 46 (2015) 190–201. http://dx.doi.org/10.1016/j.compeleceng.2015.07.001.
[25] Z. Wang, S. Peng, T. Liu, J. Softw. 6 (8) (2011) 1554–1561.
[26] D. Michéa, D. Komatitsch, Geophys. J. Int. (ISSN: 1365-246X) 182 (1) (2010) 389–402. http://dx.doi.org/10.1111/j.1365-246X.2010.04616.x.
[27] F. Rubio, M. Hanzich, A. Farrés, J. de la Puente, J.M. Cela, Comput. Geosci. (ISSN: 0098-3004) 70 (2014) 181–189. http://dx.doi.org/10.1016/j.cageo.2014.06.003.
[28] D.-H. Sheen, Kagan-Tuncay, C.-E. Baag, P.J. Ortoleva, Comput. Geosci. 32 (2006) 1182–1191.
[29] H. Aochi, F. Dupros, Procedia Comput. Sci. (ISSN: 1877-0509) 4 (2011) 1496–1505. http://dx.doi.org/10.1016/j.procs.2011.04.162.
[30] A. Caserta, V. Ruggiero, P. Lanucara, Comput. Geosci. (ISSN: 0098-3004) 28 (9) (2002) 1069–1077. http://dx.doi.org/10.1016/S0098-3004(02)00024-9.
[31] R.L. Higdon, Math. Comp. 47 (176) (1986) 437–459. http://dx.doi.org/10.1090/S0025-5718-1986-0856696-4.
[32] Intel, Quick Reference Guide to Optimization with Intel C++ and Fortran Compilers v16, https://software.intel.com/sites/default/files/managed/12/f1/Quick-Reference-Card-Intel-Compilers-v16.pdf, 2015.
[33] Intel, Intel 64 and IA-32 Achitectures Optimization Reference Manual, http://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2016.
[34] Intel, A Guide to Auto-vectorization with Intel C++ Compilers, 2012, https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf.
[35] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, P. Sadayappan, Data layout transformation for stencil computations on short-vector simd architectures, in: International Conference on Compiler Construction, Springer, Berlin, 2011, pp. 225–245.
[36] A. Ilic, F. Pratas, L. Sousa, IEEE Comput. Architecture Lett. (ISSN: 1556-6056) 13 (1) (2014) 21–24. http://dx.doi.org/10.1109/L-CA.2013.6.
[37] S. Williams, A. Waterman, D. Patterson, Commun. ACM (ISSN: 0001-0782) 52 (4) (2009) 65–76. http://dx.doi.org/10.1145/1498765.1498785.
[38] A.V. Gorobets, F.X. Trias, A. Oliva, Comput. & Fluids (ISSN: 0045-7930) 88 (2013) 764–772. http://dx.doi.org/10.1016/j.compfluid.2013.05.021.
[39] I.M. Kulikov, I.G. Chernykh, A.V. Snytnikov, B.M. Glinskiy, A.V. Tutukov, Comput. Phys. Comm (ISSN: 0010-4655) 186 (2015) 71–80. http://dx.doi.org/10.1016/j.cpc.2014.09.004.