Efficient and portable vectorised sweep kernels for the transport equation on 3D cartesian grids using the Kokkos framework

Gabriel Suau, Ansar Calloo, Remi Baron, Romain Le Tellier

M&C 2023 - The International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering · Niagara Falls, Ontario, Canada · August 13 – 17, 2023

Efficient and portable vectorised sweep kernels for the transport equation on 3D cartesian grids using the Kokkos framework

G. Suau[1], A. Calloo[1], R. Baron[1] and R. Le Tellier[2]

[1]Université Paris-Saclay, CEA,
Service d'Études des Réacteurs et de Mathématiques Appliquées,
F-91191, Gif-sur-Yvette, France

[2]CEA, DES, IRESNE, DTN, Cadarache
F-13108 Saint-Paul-Lez-Durance, France

gabriel.suau@cea.fr, ansar.calloo@cea.fr, remi.baron@cea.fr, romain.le-tellier@cea.fr

ABSTRACT

This paper describes the implementation of efficient and portable vectorised sweep kernels as part of the resolution of the neutron transport equation on 3D cartesian grids using the discrete ordinates ($\mathcal{S}_n$) method for the angular variable and the Diamond Differencing (DD) scheme for the spatial discretisation. Vectorisation is set up along the directions within the same octant, and is independent of the spatial discretisation order, therefore the extension of this technique to high-order DD or Discontinuous Galerkin (DG) schemes is immediate. Our implementation is written in C++17 and relies on the Kokkos performance portability framework. This library allows to express shared-memory parallelism (including vectorisation) in a machine-independent way, and supports many backends including CUDA and OpenMP. Our vectorisation procedure relies on the portable SIMD types provided by Kokkos. The method has been implemented for DD schemes up to order 2, and yields promising results on CPUs supporting standard vector instructions.

KEYWORDS: Discrete ordinates, Diamond Differencing schemes, Vectorisation, Performance portability, Kokkos

1. NUMERICAL BACKGROUND

In this work, we consider the solution to the within-group neutron transport equation on a 3D cartesian domain $\mathcal{D}$ using the discrete ordinates or $\mathcal{S}_n$ method [1]. The $\mathcal{S}_n$ problem can be written as

$$\begin{aligned} \vec{\Omega}_k \cdot \vec{\nabla}\psi_k(\vec{r}) + \Sigma(\vec{r})\psi_k(\vec{r}) &= Q(\vec{r}) \qquad \forall \vec{r} \in \mathcal{D}, \\ \phi(\vec{r}) &= \sum_{k=1}^{n_d} \omega_k \psi_k(\vec{r}), \end{aligned} \tag{1}$$

where $\{(\vec{\Omega}_k, \omega_k), k = 1, ..., n_d\}$ are the discrete angular directions and their weights defined by the angular quadrature formula, $\psi_k(\vec{r})$ is the angular flux in direction $\vec{\Omega}_k$, $\Sigma(\vec{r})$ is the macroscopic total cross-section, $Q(\vec{r})$ is the neutron source distribution, regrouping the external, fission and scattering sources, and $\phi(\vec{r})$ is the scalar flux. For each direction $\vec{\Omega}_k$, boundary conditions are given on the incoming border $\partial\mathcal{D}_- = \{\vec{r} \in \partial\mathcal{D},\ \vec{\Omega}_k \cdot \vec{n}(\vec{r}) < 0\}$.

The spatial domain is discretised with a cartesian grid and we use the High Order Diamond Differencing scheme (DD-M [*]) presented in [2,3]. With this method, a sweeping algorithm is employed to invert the transport operator using a source-flux iteration. Thus, this results in computing the angular flux in each cell as described by Algorithm 1.

Given a direction $\vec{\Omega}_k$, the computation in each space cell relies on the values of the angular flux moments in its immediate upstream neighbours. Therefore, the full domain resolution is done by sweeping through the cells following a propagation front perpendicular to the direction $\vec{\Omega}_k$ (see Figure 1). For each cell and direction, the $(M+1)^3$ volume moments of the angular flux are the solution of a linear system. The $(M+1)^2$ surface moments of the three outgoing angular fluxes are computed using closure relations, and are used as incoming surface fluxes for the downstream cells calculation, and so on until the full space-angle discrete domain has been swept [†].

Algorithm 1: Sweep algorithm on a 3D cartesian grid with the DD-M scheme.

forall $o \in \{1, ..., 8\}$ **do**
  forall $\vec{\Omega} \in \{\vec{\Omega}_d^o,\ d = \{1, ..., n_d^o\}\}$ **do**
    $\vec{\Omega} \equiv \{(\mu, \eta, \xi), \omega\}$;
    forall $c \in$ `Cells` **do**
      $\Delta \equiv (\Delta x, \Delta y, \Delta z)$;
      $\mathbb{S} = \mathbb{S}(\Delta, \Sigma_t, \vec{\Omega})$; `/* Assemble local matrix */`
      $b = b(\psi_x, \psi_y, \psi_z, \Delta, Q, \vec{\Omega})$; `/* Assemble local rhs */`
      $\psi = \mathbb{S}^{-1} b$; `/* Solve local linear system */`
      $\psi_x = f_x(\psi, \psi_x)$; `/* Apply closure relations */`
      $\psi_y = f_y(\psi, \psi_y)$;
      $\psi_z = f_z(\psi, \psi_z)$;
      $\phi_c = \phi_c + \psi \cdot \omega$; `/* Add contribution to scalar flux */`
    end
  end
end

For Cartesian grids, directions are generally divided in eight octants, where all directions in an octant share the same incoming border and the same sweeping pattern. In a classical $\mathcal{S}_n$ solver, the sweep algorithm is the most computationally intensive kernel and is in general responsible for more than 95% of the total execution time. Hence, an efficient optimisation and parallelisation strategy of this kernel is critical to obtain good performance.

2. PARALLEL SWEEP

2.1. Brief Overview of Massively Parallel Architectures

The architecture of massively parallel clusters exhibits several levels of hardware parallelism, each associated with one or more programming models at the software level. In addition to shared-memory (OpenMP, Intel TBB) and distributed-memory (commonly based on the MPI standard)

[*] M denotes the order of the scheme.

[†] for details on the linear system and on the closure relations, the reader can refer to [2,3]

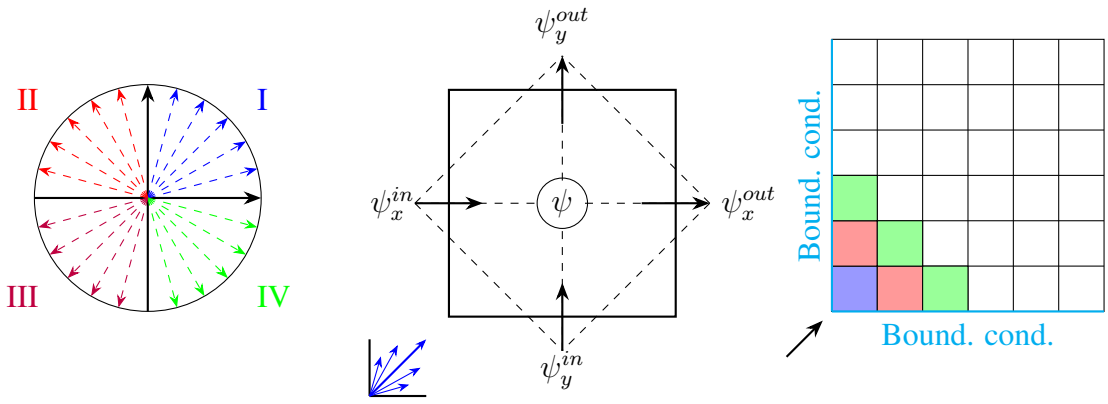

Figure 1: Illustration of a space-angle sweep on a 2D cartesian grid

architectures which exhibit intra-node and inter-node parallelism, modern CPU cores implement hardware vector instruction sets that operate on vector registers of fixed size. Those architectures are named *Single Instruction Multiple Data* (SIMD) in Flynn's taxonomy [4]. The most common vector instruction sets on modern CPUs and their associated register size are listed in Table 1. For computationally intensive algorithms, the use of vector instructions is critical to fully benefit from the available hardware parallelism.

Table 1: Vector instruction sets for x86 and associated register size

Instruction set	Register size (bits)	# of registers	Elements per register	
			simple precision	double precision
SSE	128	16	4	2
AVX	256	16	8	4
AVX512	512	32	16	8

2.2. Available Parallelism in the Sweep Algorithm

The sweep algorithm on cartesian 3D domains exhibits several levels of parallelism : all octants can be treated in parallel (provided void boundary conditions), all directions in an octant can be treated in parallel with a reduction at the end to compute the scalar flux from the angular fluxes and the quadrature formula, and, for a given direction, all cells that have resolved their upwind dependencies (*i.e.* all cells located on the same propagation front) can also be treated in parallel.

Parallelisation strategies of the sweep algorithm are widely studied, but most strategies in literature focus on inter- and intra-node parallelism, and generally do not focus on on-core performance. The explicit use of the vector registers and instructions sets of CPU cores for the sweep algorithm is seldom discussed and most publications offer no comparison between the performance results and the actual peak performance of the hardware. In this paper, we restrict ourselves to the optimisation of the single-core performance of the sweep kernel using vectorisation.

3. VECTORISATION OF THE SWEEP ALGORITHM

In [5], an efficient multicore parallel DD-0 sweep algorithm on Cartesian grids using SIMD parallelism to maximise single-core performance is presented. This implementation achieves good performance *via* the use of Eigen [‡] arrays that internally use vector instructions. In this work, we propose a new and portable implementation of the method described in [5], for DD schemes up to order two, using the Kokkos performance portability framework [7].

3.1. Description of the Vectorisation Method

The method relies on the vectorisation of the intra-octant loop on directions (see Algorithm 1). Indeed, on cartesian grids, the mesh cells can be swept in the exact same order for all directions in the same octant.

Most modern compilers have the ability to automatically vectorise blocks of independent instructions or loops that exhibit independent iterations. However, this approach has three main limitations:

- compilers often fail to vectorise loops with complex or long bodies because they lack knowledge of the actual data dependencies;
- the quality of the generated vector code can vary a lot from one compiler to another;
- vectorisation always happens in the innermost loop (if it happens at all).

In this work, we chose a different approach that relies on the portable SIMD types provided by Kokkos [7]. Those SIMD types are small packs of scalars that support classical arithmetic operators that map directly to hardware-specific vector instructions. The pack size and the set of machine instructions are chosen at compile-time depending on the scalar type and the target architecture (see Table 1). Instead of treating single directions one after the other, the algorithm now operates on packs of directions belonging to the same octant. For the algorithm to work with angular quadratures of any order, a padding strategy is used: fictive directions are added to the quadrature formula with null weights until the total number of directions per octant is a multiple of the pack size. An example using SSE and an $\mathcal{S}_8$ angular quadrature is shown in Figure 2.

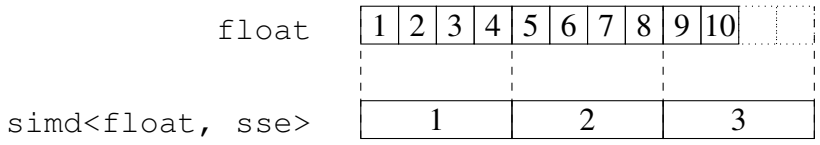

Figure 2: Illustration of the padding strategy with SSE and an $\mathcal{S}_8$ quadrature.

Vectorisation along directions requires that the same instructions must be called for all directions in the same pack. Therefore, the method to solve the local linear system must not introduce divergence between different directions, and we use a Gaussian Elimination without pivoting. In short, the vectorised algorithm is the same as Algorithm 1, with two minor changes:

[‡]Eigen [6] is a C++ template library for linear algebra that provides data structures for matrices and vectors and numerical solvers.

- angle-dependent quantities are not scalars anymore, but small packs of scalars,
- n_d^o is replaced by $n_d^{o,v} = (n_d^o + n_{pad}^o)/V$, where n_{pad}^o is the number of fictive directions per octant and V is the SIMD pack size. $n_d^{o,v}$ is the number of direction packs per octant.

Table 2 summarises the number of directions per octant without and with padding for different orders of Level-Symmetric angular quadrature for each vector instruction set, and the associated maximum theoretical speedups that can be achieved. The last detail to handle is that the direction loop iterations are in fact not totally independent because of the reduction at the end to compute the scalar flux. This problem is adressed in the next section.

Table 2: Padding and maximum theoretical speedup of vectorisation in simple precision for different orders of Level-Symmetric angular quadrature.

	n_d	Directions per octant n_d^o ($n_d^{o,v}$)				Maximum speedup		
		w/o padding	w/ padding					
			SSE	AVX	AVX512	SSE	AVX	AVX512
$\mathcal{S}_2$	8	1	4 (1)	8 (1)	16 (1)	1	1	1
$\mathcal{S}_4$	24	3	4 (1)	8 (1)	16 (1)	3	3	3
$\mathcal{S}_8$	80	10	12 (3)	16 (2)	16 (1)	3.3	5	10
$\mathcal{S}_{12}$	168	21	24 (6)	24 (3)	32 (2)	3.5	7	10.5
$\mathcal{S}_{16}$	288	36	36 (9)	40 (5)	48 (3)	4	7.2	12

The approach based on SIMD types has at least two advantages compared to the one relying on compiler auto-vectorisation. Firstly, it enforces the use of vector instructions and it does not depend on the compiler's ability to analyse data dependencies or to generate efficient vectorised code by itself. We should therefore expect similar running times from one compiler to another. Secondly, it is more flexible:

- it allows to achieve outer-loop vectorisation;
- if multiple instruction sets are available, we can chose the one that is best fitted to our needs (AVX512 may not always be the best choice, as we will point out in Section 4);
- further parallelisation on directions (using threads for example) is straightforward because SIMD packs behave exactly like scalars.

3.2. Implementation Details

The solver has been implemented from scratch in a C++17 mockup and consists of approximately 10k lines of code. Our solver relies entirely on Kokkos Views to store multidimensional arrays, such as scalar flux, sources, surface angular fluxes and quadrature directions and weights.

Kokkos is an open-source framework dedicated to parallel programming and performance portability. It provides data structures and powerful abstractions to express shared-memory parallelism in an architecture-agnostic way. Views are the Kokkos abstraction for multidimensional arrays. They are light objects that hold a pointer to a memory chunk along with light metadata describing

how data is organized (dimensions, type of the elements, memory layout, location etc.). Views behave like shared pointers and can be aliased, allowing two Views to point to the same data with different metadata. Finally, Views can allocate and manage their own data, and Views allocations are always aligned with the size of their elements.

The scalar flux, the source and the quadrature formula exist independently of the sweep kernel. However, for a full space-angle sweep, we need to allocate three arrays to store the surface angular fluxes, each of which must be able to store a full mesh plane. For instance, the ψ_x array is a Kokkos View of rank 4 with dimensions $(n_y, n_z, (M+1)^2, n_d^o)$.

The transition from scalar types to SIMD types is straightforward and requires very little work, as only the angular quadrature storage, the surface fluxes storage and the reduction in the sweep kernel are modified. Thanks to Kokkos Views, the storage transition is immediate: we only have to change the types of the elements from `float` to `simd<float>`, and to replace n_d^o with $n_d^{o,v}$. Data alignment is automatically ensured by Views, which enables fast vector load operations.

As for the reduction handling, we implemented three different strategies (hereafter named A, B and C), based on a two step method : a first partial reduction inside the direction loop and an additional final reduction outside the loop. SIMD-A computes the SIMD contributions and store them all, and performs a fully scalar reduction at the end using an alias scalar view. SIMD-B accumulates the SIMD contributions inside the loop in a single pack, and then performs a scalar reduction over this pack by using an alias scalar view. SIMD-C is similar to SIMD-B, except that the final reduction is itself vectorised. Figure 3 illustrates how each implementation works.

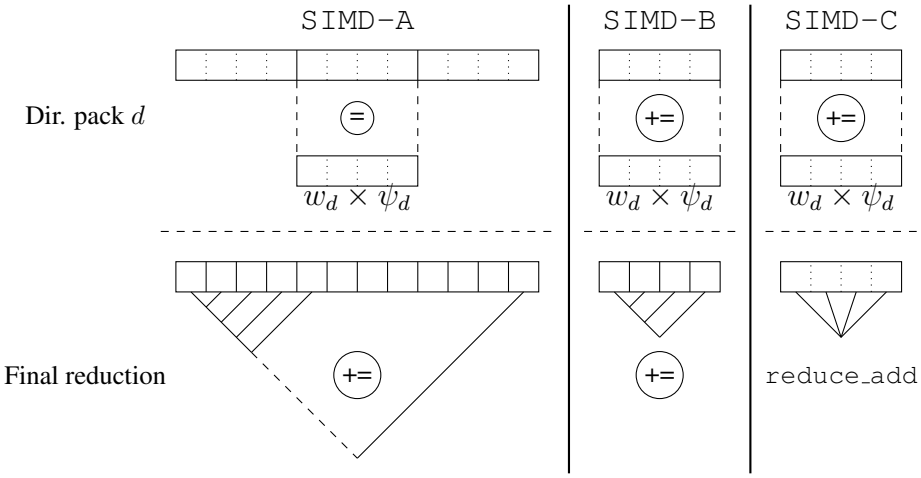

Figure 3: Illustration of the three SIMD implementations with SSE and an $\mathcal{S}_8$ quadrature.

Note that SIMD reduction operations are not implemented natively in Kokkos' SIMD types. The `reduce_add` function was implemented by hand using intrinsics for SSE, AVX and AVX512.

4. TESTS AND RESULTS

The implementation has been validated on the Takeda Model 1 benchmark [8], and our results show a good agreement with the Monte Carlo reference, both in term of k_{eff} and region-averaged fluxes. Performance tests have also been carried out to measure the speedup brought by this vectorisation strategy.

Performance tests were performed in simple-precision for the scalar implementation and for the three SIMD implementations on an Intel Xeon Gold 6226, using SSE, AVX and AVX512 instruction sets on a single core, and for three different compilers: GCC 8.3.0, Clang 11.1.0 and Intel's ICPC 2019-3. All implementations were compiled using the same compiler flags: `-O3 -march=native -mtune=native -m{sse,avx,avx512f}`. All tests run 10 power iterations on the Takeda Model 1 benchmark, resulting in 20 full space-angle sweeps. For the sake of conciseness, and because Intel compiler's results are quite similar to Clang's results, we will only describe here the tests and results for GCC and Clang.

Results of the tests are shown in Figures 4, 5 and 6. The bars represent the *grind-time* for each case, *i.e.* the average time needed to compute one space-angle unknown (one space moment of the angular flux in one cell for one direction). We also show the ideal grind-time, *i.e.* the scalar implementation's grind-time divided by the maximum attainable speedup (see Table 2). The measured speedup is also plotted for convenience, along with the maximum attainable speedup.

The first observation that can be made is that, although implementation SIMD-C is the most efficient at low order, the performance gap between SIMD implementations tends to disappear as the spatial order grows. This is an expected result. Indeed, if we call $m_s = (M+1)^3$ the number of spatial moments, then the time spent in the resolution of the local linear system in each direction-cell grows as m_s^3, whereas the time spent in the reduction operation grows linearly with m_s. Therefore, the reduction implementation is less and less important as the order grows.

Secondly, if we look at the scalar implementation performances for the DD-2 scheme, GCC produces much faster code that Clang (2 to $2.5\times$ faster). A quick examination of the assembly code did not show any sign of auto-vectorisation by GCC. However, further tests showed that this implementation compiled with GCC and with the `-O2` flag has approximately the same results as with Clang and `-O3`, so it would seem GCC is able to perform very efficient optimisations in `-O3` mode that Clang cannot do. This result may also be caused by the fact that the spatial order is a compile-time value, so the compiler knows the number of spatial moments, and may be able to optimize the local linear system resolution. Note however that this discussion is speculative, and a further in-depth investigation would be needed to conclude.

As for the SIMD implementations, both compilers are equivalent for DD-0, but GCC slightly outperforms Clang for DD-1 and vice-versa for DD-2. Looking at Clang's results for DD-2, we achieve nearly ideal speedup for SSE and AVX instructions sets, but AVX512 speedups plateau at approximately 60% of the maximum. We see two possible explanations for this:

- with our padding strategy, the use of AVX512 registers incurs a non-negligible memory overhead, aswell as more memory traffic, which can induce less cache efficiency;
- the use of AVX512 floating point instructions can cause CPU downclocking. To effectively measure the downclocking effect, we would have to be able to monitor the CPU frequency throughout the computation for both scalar and SIMD implementation.

The second item is crucial for multithreaded applications, because the frequency drop grows with the number of cores in use. Moreover, AVX512 speedups are only slightly above AVX speedups, with twice as much directions processed in parallel, which means the efficiency of AVX512 vectorisation is about half the efficiency of AVX. Finally, AVX512 leaves less room for further angle parallelisation or pipelining (in the perspective of a distributed KBA-like sweep algorithm). Therefore, it may be advantageous to use only SSE or AVX instructions.

5. CONCLUSION AND FUTURE WORK

In this work, we implemented transport sweep kernels vectorised along directions within the same octants, using the SIMD types provided by the Kokkos library. Our performance tests demonstrate that nearly ideal speedups can be achieved for the DD-2 scheme on SSE or AVX-enabled cores. The use of AVX512 instructions may not be ideal when combined with other parallelisation strategies, especially if the number of directions per octant is low.

Our tests raised some questions we could not address in this paper. First, a further investigation is needed to understand why GCC vastly outperforms Clang (and Intel) for the scalar DD-2 implementation. Secondly, it would be interesting to carry out tests with more directions per octant in order to compare the behaviour of our strategy with the different vector instruction sets at high direction count. Finally, an implementation of a shared-memory OpenMP-SIMD sweep algorithm with parallelisation over the cells on the propagation front would allow to test our speculations on the CPU frequency drops incurred by the use of AVX and AVX512 instructions, and to compare the relative efficiencies of each instruction set in a multi-core configuration.

REFERENCES

[1] K. D. Lathrop and B. G. Carlson. "Discrete Ordinates Angular Quadrature of the Neutron Transport Equation." (1964). URL https://www.osti.gov/biblio/4666281.

[2] A. Hébert. "High order diamond differencing schemes." *Annals of Nuclear Energy*, **volume 33**, pp. 1479–1488 (2006).

[3] A. Hébert. "A three-dimensional SN high-order diamond-differencing discretization with a consistent acceleration scheme." *Annals of Nuclear Energy*, **volume 37**, pp. 1787–1796 (2009).

[4] M. Flynn. "Very High Speed Computing Systems." *Proceedings of the IEEE*, **volume 54**, pp. 1901–1909 (1967).

[5] S. Moustafa, I. Dutka-Malen, L. Plagne, A. Ponçot, and P. Ramet. "Shared memory parallelism for 3D Cartesian discrete ordinates solver." *Annals of Nuclear Energy*, **volume 33**, pp. 179–187 (82).

[6] G. Guennebaud, B. Jacob, et al. "Eigen v3." http://eigen.tuxfamily.org (2010).

[7] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke. "Kokkos 3: Programming Model Extensions for the Exascale Era." *IEEE Transactions on Parallel and Distributed Systems*, **volume 33**, pp. 805–817 (2022).

[8] T. Takeda and H. Ikeda. "3-D Neutron Transport Benchmarks." *Journal of Nuclear Science and Technology*, **volume 28**, p. 656–669 (1991).

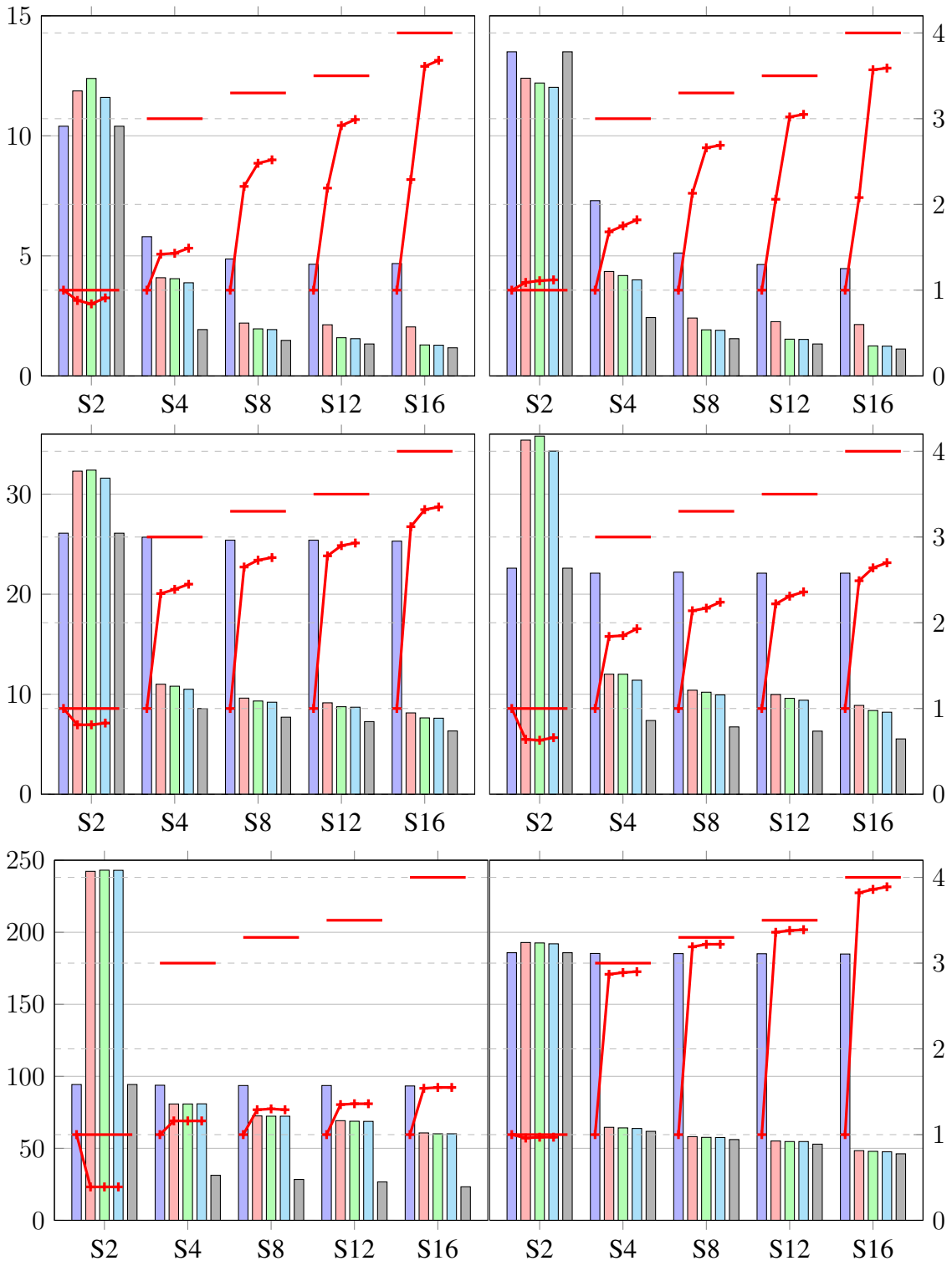

Figure 4: Performance results with GCC (left) and Clang (right) using SSE. Top to bottom: DD-0, DD-1, DD-2. Grind time (ns): left y-axis, bars (left to right : Scalar, A, B, C, Ideal). Speedup: right y-axis, lines.

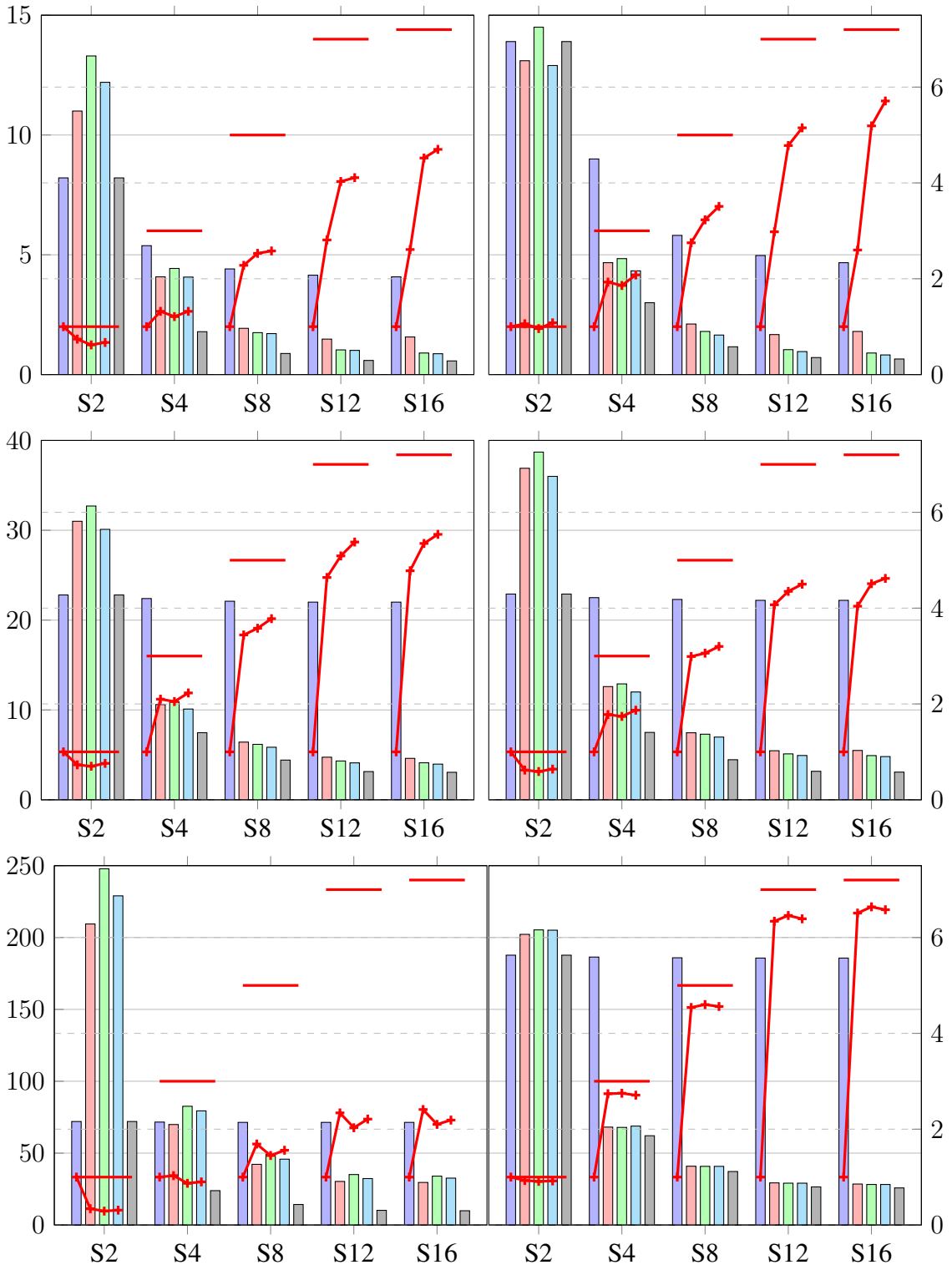

Figure 5: Performance results with GCC (left) and Clang (right) using AVX. Top to bottom: DD-0, DD-1, DD-2. Grind time (ns): left y-axis, bars (left to right : Scalar, A, B, C, Ideal). Speedup: right y-axis, lines.

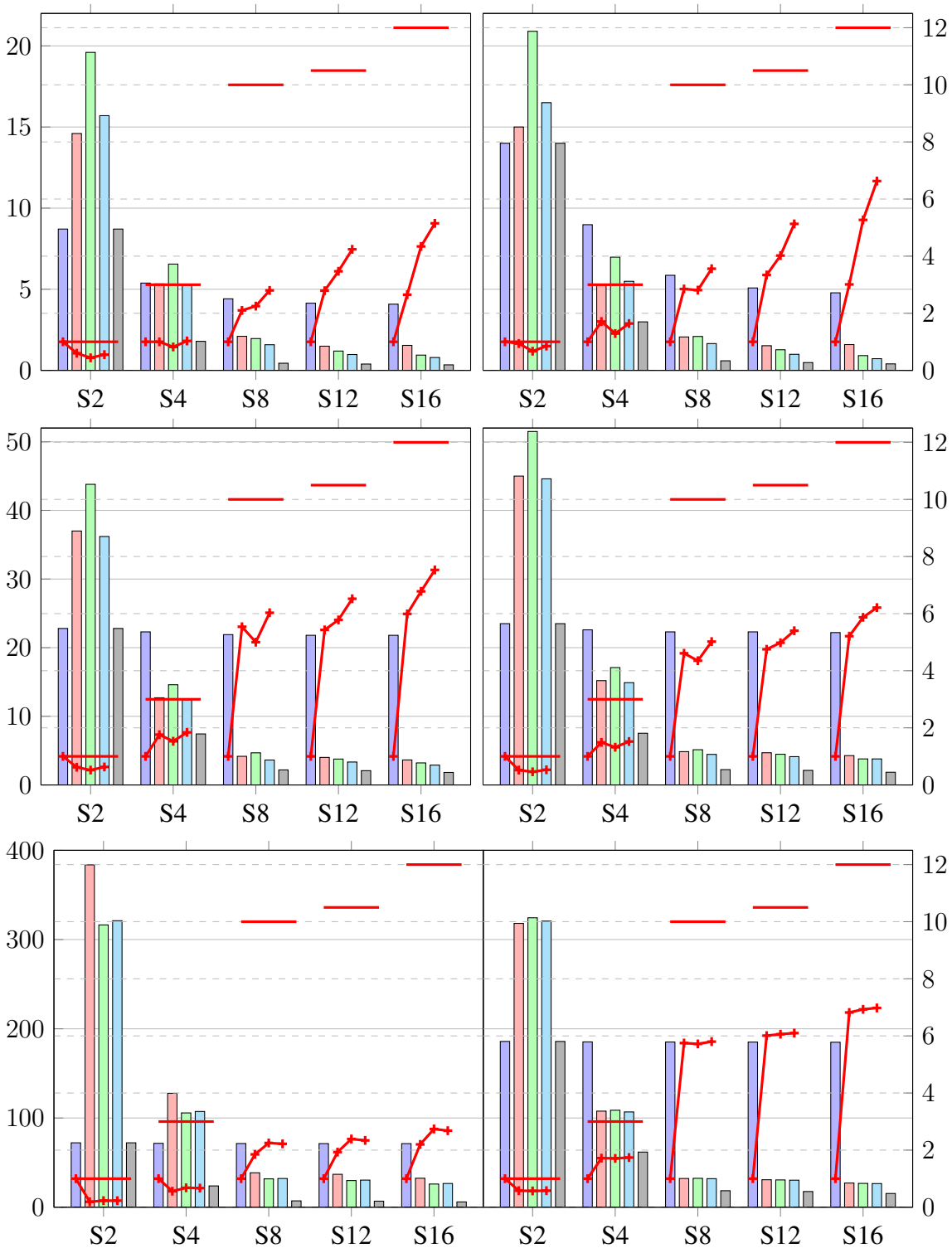

Figure 6: Performance results with GCC (left) and Clang (right) using AVX512. Top to bottom: DD-0, DD-1, DD-2. Grind time (ns): left y-axis, bars (left to right : Scalar, A, B, C, Ideal). Speedup: right y-axis, lines.