



LAWRENCE  
LIVERMORE  
NATIONAL  
LABORATORY

# On the arithmetic intensity of high-order finite volume discretizations for hyperbolic systems of conservation laws

J. Loffeld, J. A. F. Hittinger

January 4, 2017

International Journal of High Performance Computing

## **Disclaimer**

---

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

**Abstract**

It has been conjectured that higher-order discretizations for partial differential equations will have advantages over lower-order counterparts commonly used today. The reasoning is that the increase in arithmetic operations will be more than offset by the reduction in data transfers and the increase in concurrent floating-point units. To evaluate this conjecture, the arithmetic intensity of a class of high-order finite volume discretizations for hyperbolic systems of conservation laws is theoretically analyzed for spatial discretizations from orders three through eight in arbitrary dimensions. Three cache models are considered: the limiting cases of no and infinite cache as well as a finite-sized cache model. Models are validated experimentally by measuring floating point operations and data transfers on an IBM BG/Q node. Theory and experiments demonstrate that high-order finite volume methods will be able to provide increases in arithmetic intensity that will be necessary to make better utilization of on-node floating-point capability.

**Keywords**

Arithmetic intensity, High-order finite volume methods, Hyperbolic systems of conservation laws, Processor-memory performance gap, Algorithmic balance

**1 Introduction**

From desktop workstations to supercomputers, the architectures of computers are changing. The end of Dennard scaling<sup>1</sup> has arrested increases in clock speed. Instead, gains in performance are coming from additional computational units. In addition, the ubiquity of hand-held devices is driving the consumer market towards low-power computing, and prohibitive operational costs are driving the high-performance computing market to low-power computing as well. The decreasing power cost of floating-point operations has exposed the power cost of data motion. At the same time the number computational units are increasing, the standard volatile memory per core is decreasing, because such memory is a major power consumer, and memory latency is effectively increasing due to contention. Thus, floating-point operations – if concurrency can be harnessed – are no longer the primary (on-node) cost factor for numerical simulation. A new trade-off must be made between data motion, memory usage, and operations<sup>2;3</sup>.

To this end, it has been proposed that high-order discretizations for partial differential equations will have advantages in this new computing environment over their low-order alternatives<sup>4</sup>. For a given level of required

accuracy, high-order methods need fewer total degrees of freedom (nodes, cells, or elements) to represent a fully-resolved solution<sup>5</sup>, although the stencil size or number of basis function per element increases. High-order methods make more use of the existing degrees of freedom, incurring higher operation counts than low-order methods. When flops are cheap and **on-node** data motion incurs the greater cost, this exchange of flops for bytes is desirable.

Here we seek to address systematically this trade-off for finite volume discretizations of hyperbolic systems of conservation laws. Such methods play an important role in simulations in many areas, such as gas dynamics, traffic flow, elastodynamics, and geophysics. Building on the formalism of Colella *et al.*<sup>6</sup>, we derive both central and upwind-biased flux discretizations of orders three through eight and determine bounds for each for both the *operation count* and the *arithmetic intensity*, that is, the work done

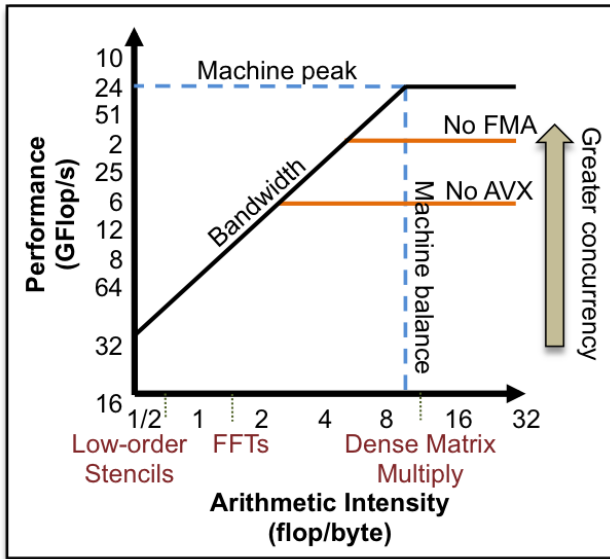
---

\*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory

**Corresponding author:**

J. Loffeld, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551-0808 USA

Email: loffeld1@llnl.gov



**Figure 1.** Example roofline model for a fictitious node architecture.

per unit byte transferred. These two quantities are useful in understanding on-node performance as characterized by the *roofline model*<sup>7</sup>, which has been used previously to study other algorithms. We consider the finite-volume framework for uniform, static grids, even though the formalism was originally proposed for mapped grids and subsequently extended to adaptive mesh refinement<sup>8</sup> and mapped, multiblock meshes<sup>9</sup>. These additional complications should favorably impact the arithmetic intensity, since they increase the number of floating point operations while potentially reducing the amount of data through mesh refinement and alignment with boundaries, flow directions, *etc.*

An example roofline model for a fictitious node architecture is shown in Figure 1. The vertical axis is the number of floating-point operations per second, which is ultimately bounded by the performance of the processor. The horizontal axis is the arithmetic intensity, the number of floating-point operations per byte transferred. The diagonal line is the upper bound on the bandwidth, the maximum rate at which data can be provided to the processor. The ideal place to compute on this plot is at the *balance point*, where the bandwidth limit and the processing limit intersect – corresponding to the point of perfect throughput. As demonstrated by several representative algorithms, the

arithmetic intensity of low-order methods is too low, such that the computation is bandwidth-limited and never exercises the full capability of the node. The conjecture is that high-order schemes will have higher intensities, and it is the goal of this work to develop a [theoretical model, validated by numerical experiments, to aid in systematic reasoning about higher-order methods](#). Note that moving vertically on the roofline model requires one to leverage all of the performance-enhancing features of a processor, which is mostly an implementation question that will not be addressed in this paper. There are numerous examples of such optimizations for a variety of stencil-based computations in the literature<sup>7:10–15:15–20</sup> and specifically for the finite-volume family of schemes considered here<sup>21</sup>.

We focus our analysis on on-node data motion and a conservative model for multi-threading because processing units per node are expected to increase much more than the number of nodes (the number of which may actually decrease) and because hyperbolic problems have finite domains of dependence. In Section 3.7, we will briefly discuss the impact of higher-order stencils on inter-node communication, but because decomposition across nodes introduces additional degrees of freedom (size of patches, number of patches per node, relationships between patches on-node and off-node, *etc.*), a complete treatment is beyond the scope of this paper.

This paper is organized as follows. In the next two sections, we review related prior work, introduce the high-order finite volume formalism, and derive both odd and even-order high-order formulas for the numerical flux function. In Section 4, we present the results of analyzing the schemes in the no-cache and infinite-cache limits, followed by similar analysis for a finite-sized cache [tiling](#) model. Details of these analyses are provided in the Appendix. We verify the analysis empirically, and these results are presented in Section 5. Finally, we draw conclusions and identify areas for future investigation.

## 2 Previous work

Historically, algorithmic performance was measured in terms of the number of arithmetic operations. With increases in processor floating-point performance outstripping increases in bandwidth, the bottleneck in many algorithms has for some time shifted to data motion. The move to distributed parallel computers further emphasized the cost of communications. Both measures are important metrics of algorithms (though not the only metrics). Callahan *et al.* introduced the *machine balance* metric, that is, the ratio of memory operations to arithmetic operations, in the context of pipelining of loops<sup>22;23</sup>. In some sense, the roofline model<sup>7</sup> is a generalization of this idea that is a simple visual performance model showing the trade-offs between computation and data transfer.

Since its introduction, improvements and alternatives to the roofline model have been proposed<sup>24-27</sup>, introducing additional details such as real cache effects and overlapping computation and communication. Of course, the original roofline model can accommodate such details; in the original paper<sup>7</sup>, the authors refer to the various “ceilings” under the roofline that limit actual performance below the maximum attainable performance at the roofline. The basic roofline model will suffice for the purpose of this work: to understand theoretically the trade-offs in designing higher-order finite volume discretizations. Specifically, we attempt to derive best and worst case estimates of the algorithmic intensity of higher-order finite volume methods to understand the effectiveness of higher-order methods as a strategy for moving beyond the bandwidth-limited region of the roofline model.

In contrast, other analytical work has focused on determining lower bounds for communication costs. The seminal work in this area is the work of Hong and Kung<sup>28</sup> on dense matrix-matrix multiply using a ‘red-blue pebble game’ analysis. Since then, there have been numerous lower-bounds derived (and algorithms achieving these lower bounds designed) across dense and sparse linear algebra for direct and iterative methods; Ballard *et al.*<sup>29</sup> is an excellent summary of these results. Minimizing communication increases arithmetic intensity by decreasing

the denominator, but we pursue a different question: what can be gained by increasing the numerator as well? Given a certain amount of data transferred, how many more useful operations (useful in the sense of increasing accuracy) are needed to reach machine balance?

The roofline model has been used to guide the performance optimization of a wide variety of algorithms. Williams *et al.*<sup>7</sup> considered four computational kernels: sparse matrix-vector multiply; a Lattice Boltzmann model for magneto-hydrodynamics (LBMHD); a sixth-order, 3D Laplacian stencil; and a 3D FFT. In Basu *et al.*<sup>10</sup>, compiler-directed reordering transformations that exploit data reuse and reduce floating-point operations were evaluated on discretizations of the Poisson equation up to tenth order; the resulting linear systems were solved using a geometric multigrid algorithm and the performance was interpreted in terms of the roofline model. The work on optimization of LBMHD using the roofline model was further detailed by Williams *et al.*<sup>11</sup> and later used as the basis for an auto-tuning strategy<sup>12</sup>. The performance of a molecular dynamics code, a quantum chromodynamics lattice code, and a pseudo-spectral turbulence code on three top HPC platforms was measured and compared in the context of the roofline model in Bhatele *et al.*<sup>30</sup>. Fast methods, such as treecode, fast multipole and hybrids, were identified in Yokota and Barba<sup>31</sup> as high operational intensity algorithms for  $N$ -body problems. In plasma physics, the roofline model has been used in optimization studies of a 5D gyrokinetic Eulerian core code<sup>13</sup> and a 1D, mixed-precision, nonlinearly implicit particle-in-cell algorithm<sup>32</sup>. The Lattice Boltzmann Method for fluid dynamics has been the subject of other optimization studies using the roofline model<sup>14;15</sup>, including the use of domain-specific languages to allow for automated code generation of optimized kernels (such as time tiling)<sup>15</sup>. The roofline model has also been used to interpret the performance of and guide the optimization of several modern, stencil-based compressible flow codes in applications such as multiphase flow<sup>16-18</sup> and direct numerical simulation of turbulence<sup>19</sup>. The multiphase studies consider high-order finite volume methods based on the fifth-order, adaptive Weighted Essentially Non-Oscillatory (WENO) method<sup>20</sup>,

so are quite relevant to the work here; these studies show that the key kernels of such algorithms are bandwidth-limited on current machines. In contrast to this previous work, however, we are using the roofline model to motivate algorithm design instead of using it as a tool to guide code optimization. In particular, we seek to develop theoretical guidance on the order necessary to obtain an operational intensity large enough to utilize all of the available FLOPs on a node.

### 3 High-order finite-volume discretizations

Finite-volume methods are useful for solving conservation laws of the form

$$\frac{\partial u}{\partial t} + \nabla \cdot \mathbf{F}(u) = 0, \quad (1)$$

where  $\mathbf{x} \in \mathbf{R}^D$  is the  $D$ -dimensional spatial coordinate,  $t \in \mathbf{R}^+$  is the time variable,  $u(\mathbf{x}, t) : \mathbf{R}^D \times \mathbf{R}^+ \rightarrow \mathbf{R}^m$  is the vector of  $m$  conserved variables, and  $\mathbf{F}(u) : \mathbf{R}^m \rightarrow \mathbf{R}^{m \times D}$  are the vector-valued flux functions.

Uniform Cartesian-grid methods begin by discretizing the spatial domain in  $\mathbf{R}^D$  into control volumes:

$$V_{\mathbf{i}} = \left[ x_{i_1} - \frac{h}{2}, x_{i_1} + \frac{h}{2} \right] \times \left[ x_{i_2} - \frac{h}{2}, x_{i_2} + \frac{h}{2} \right] \times \cdots \times \left[ x_{i_D} - \frac{h}{2}, x_{i_D} + \frac{h}{2} \right], \quad (2)$$

where  $\mathbf{i} = (i_1, i_2, \dots, i_D) \in \mathbb{Z}^D$  is a multi-index associated with the center point of the control volume,  $h$  is the grid spacing, and, for coordinates relative to  $\mathbf{x}_0$ ,

$$x_{i_d} = (\mathbf{x}_0)_{i_d} + (i_d + 1) \frac{h}{2} \quad (3)$$

The solution is spatially discretized in each cell  $\mathbf{i}$  by averaging the solution over the corresponding  $V_{\mathbf{i}}$ ,

$$\bar{u}_{\mathbf{i}}(t) = \frac{1}{h^D} \int_{V_{\mathbf{i}}} u(\mathbf{x}, t) d\mathbf{x}, \quad (4)$$

and the time evolution of the system can then be computed in a method-of-lines (MOL) approach by discretizing the

resulting system of ODEs,

$$\frac{d\bar{u}_{\mathbf{i}}}{dt} = -\frac{1}{h^D} \int_{V_{\mathbf{i}}} \nabla \cdot \mathbf{F} d\mathbf{x}. \quad (5)$$

Applying the divergence theorem, the average of the divergence of the fluxes can be computed as a sum of the face averages of the fluxes,

$$\frac{d\bar{u}_{\mathbf{i}}}{dt} = -\frac{1}{h} \sum_{d=1}^D \left[ \langle F^d \rangle_{\mathbf{i} + \frac{1}{2} \mathbf{e}^d} - \langle F^d \rangle_{\mathbf{i} - \frac{1}{2} \mathbf{e}^d} \right], \quad (6)$$

where the face averages are

$$\langle F^d \rangle_{\mathbf{i} \pm \frac{1}{2} \mathbf{e}^d} = \frac{1}{h^{D-1}} \int_{A_{\mathbf{i} \pm \frac{1}{2} \mathbf{e}^d}} F^d dA, \quad (7)$$

where  $A_{\mathbf{i} \pm \frac{1}{2} \mathbf{e}^d}$  denotes the faces of  $V_{\mathbf{i}}$  with normals pointing in the direction  $\mathbf{e}^d$  with face center points  $\mathbf{i} \pm \frac{1}{2} \mathbf{e}^d$ . The face integrals (7) are then approximated with a quadrature rule.

Following the approach taken by Colella *et al.*<sup>6:33</sup>, a quadrature is determined by expanding the flux function in Taylor expansion about the center of the face and integrating the result, i.e.,

$$\int_{A_d} F^d dA = \sum_{0 \leq |\mathbf{r}| < R} \frac{1}{r!} \nabla^{\mathbf{r}} F^d |_{\mathbf{x}=\mathbf{x}_{\mathbf{k}_d}} \int_{A_d} (\mathbf{x} - \mathbf{x}_{\mathbf{k}_d})^{\mathbf{r}} dA_x + O(h^{R+D-1}), \quad (8)$$

where  $\mathbf{r}$  is a  $D$ -dimensional multi-index,  $R$  is the ‘‘order’’ of the leading term neglected in the series expansion, and

for  $\mathbf{k}_d = \mathbf{i} + \frac{1}{2}\mathbf{e}^d$ . For example, when computed to eighth-order ( $R = 8$ ), the face average is

$$\begin{aligned}
\langle F^d \rangle_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d} &= F^d(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}) + \frac{h^2}{24} \sum_{d_i \neq d} \frac{\partial^2 F^d}{\partial x_{d_i}^2}(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}) \\
&+ \frac{h^4}{1920} \sum_{d_i \neq d} \frac{\partial^4 F^d}{\partial x_{d_i}^4}(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}) \\
&+ \frac{h^4}{576} \sum_{\substack{d_i, d_j \neq d \\ d_i \neq d_j}} \frac{\partial^4 F^d}{\partial x_{d_i}^2 \partial x_{d_j}^2}(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}) \\
&+ \frac{h^6}{322560} \sum_{d_i \neq d} \frac{\partial^6 F^d}{\partial x_{d_i}^6}(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}) \\
&+ \frac{h^6}{46080} \sum_{\substack{d_i, d_j \neq d \\ d_i \neq d_j}} \frac{\partial^6 F^d}{\partial x_{d_i}^4 \partial x_{d_j}^2}(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}) \\
&+ \frac{h^6}{13824} \sum_{\substack{d_i, d_j, d_k \neq d \\ d_i \neq d_j \neq d_k}} \frac{\partial^6 F^d}{\partial x_{d_i}^2 \partial x_{d_j}^2 \partial x_{d_k}^2}(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}) \\
&+ O(h^8).
\end{aligned} \tag{9}$$

The coefficients of odd-ordered derivative terms are always zero. When  $\mathbf{r}$  is odd, the terms  $(\mathbf{x} - \mathbf{x}_{\mathbf{k}_d})^{\mathbf{r}}$  necessarily have an odd exponent along at least one direction. Along such directions, the term is equally weighted but with opposite sign on each side of  $\mathbf{x}_{\mathbf{k}_d}$  to the edge of the face, and thus the integral over the face goes to zero. (This is similar to integrating an odd function over a region of equal length on each side of the origin.) The remaining even-ordered terms are computed using centered finite-difference formulas of appropriate order of accuracy.

### 3.1 Reconstruction

The face averages computed in (9) require point-valued fluxes  $F^d(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}) = F^d(u(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}))$  as inputs, so a method is needed for computing face-centered point values  $u(\mathbf{x}_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d})$  from the cell-averaged values  $\bar{u}_i$ . We use the *primitive function* approach of reconstructing the point values introduced by Colella and Woodward<sup>34</sup>. Limiting ourselves at first to a one-dimensional domain,  $x \in \mathbb{R}$ , the

primitive function for  $u(x)$  is defined as

$$w(x) = \int_{-\infty}^x u(\xi) d\xi. \tag{10}$$

The property that will be exploited is that  $u(x) = w'(x)$ . The lower limit of integration in the integral is not important and can be replaced by any fixed value. Since

$$w(x_{i+\frac{1}{2}}) = \int_{-\infty}^{x_{i+\frac{1}{2}}} u(\xi) d\xi, \tag{11}$$

$w(x)$  can be written in terms of the cell averages of  $u(x)$  by

$$w(x_{i+\frac{1}{2}}) = h \sum_{-\infty}^i \bar{u}_{i+\frac{1}{2}}, \tag{12}$$

and therefore we have exact values for  $w(x)$  at the cell boundaries. Assuming  $w(x)$  is sufficiently smooth, it can be interpolated through  $s$  cell boundary points to give a polynomial representation,

$$p(x) = w(x) + O(h^{s+1}). \tag{13}$$

Since  $u(x) = w'(x)$ , by differentiating  $p(x)$  (losing one order of accuracy in the process), we obtain a representation of  $u(x)$  as

$$p'(x) = u(x) + O(h^s). \tag{14}$$

Therefore,  $p'(x)$  gives the approximation we need to reconstruct point values for  $u(x)$  to a specified accuracy. Furthermore, it has the desirable property that its cell average equals that of  $u(x)$  itself:

$$\frac{1}{h} \int_{x_{i-1/2}}^{x_{i+1/2}} p'(\xi) d\xi = \frac{1}{h} (p(x_{i+1/2}) - p(x_{i-1/2})), \tag{15}$$

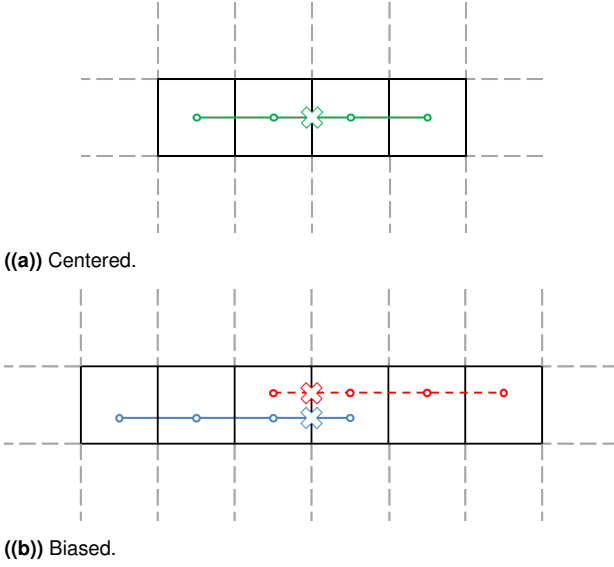
$$= \frac{1}{h} (w(x_{i+1/2}) - w(x_{i-1/2})), \tag{16}$$

$$= \frac{1}{h} \left( \int_{-\infty}^{x_{i+1/2}} u(\xi) d\xi - \int_{-\infty}^{x_{i-1/2}} u(\xi) d\xi \right), \tag{17}$$

$$= \frac{1}{h} \int_{x_{i-1/2}}^{x_{i+1/2}} u(\xi) d\xi, \tag{18}$$

which is the desired property.

Evaluating  $p'(x)$  at a particular point results in a stencil expression for the point value. For example, the stencil



**Figure 2.** Stencils for the centered and biased reconstruction of the face-averaged values from cell-averaged values using the primitive function approach. In the centered case, a single stencil computes the face-averaged value  $\langle u \rangle_{i+\frac{1}{2}\mathbf{e}^d}$ , denoted by the green cross. In the biased approach, a left-biased stencil computes  $\langle u \rangle_{i+\frac{1}{2}\mathbf{e}^d}^L$  (blue cross) and a right-biased stencil computes  $\langle u \rangle_{i+\frac{1}{2}\mathbf{e}^d}^R$  (red cross). The two values are passed to a Riemann solver to compute the final  $\langle u \rangle_{i+\frac{1}{2}\mathbf{e}^d}$ .

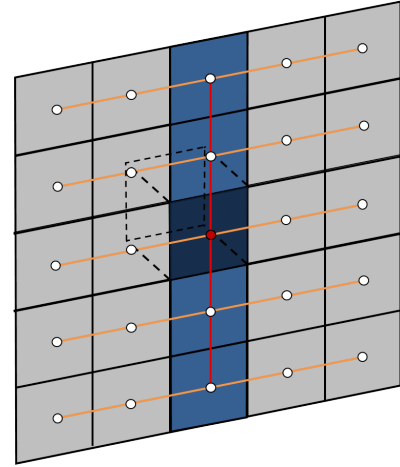
expression for computing the solution at the cell edge to fourth order is

$$u(x_{i+1/2}) \approx p'(x_{i+1/2}), \quad (19)$$

$$= -\frac{1}{12}\bar{u}_{i-1} + \frac{7}{12}\bar{u}_i + \frac{7}{12}\bar{u}_{i+1} - \frac{1}{12}\bar{u}_{i+2}. \quad (20)$$

The reconstruction is thus easily implemented in code.

For multi-dimensional problems, the reconstruction of the solution at the face center-point from cell averages can be accomplished by using the primitive function approach one dimension at a time. First,  $p'(x)$  is evaluated at the cell boundaries  $x_{i+\frac{1}{2}\mathbf{e}^d}$ , giving the average for  $u(x)$  over the face, resulting in the data being averaged over one less dimension than the cell average. The stencil for this process is shown in Figure 2(a). The same process is repeated recursively for each dimension within the plane of the face, except now to the center of the face instead of the edge. Each iteration “de-averages” the solution along that



**Figure 3.** The sequence of stencils for computing the face-centered point values  $u_{i+\frac{1}{2}\mathbf{e}^d}$  from face-averaged values  $\langle u \rangle_{i+\frac{1}{2}\mathbf{e}^d}$ . The de-averaging stencil, shown as orange lines, is applied along a direction within the plane to the face-averaged values, which are denoted by gray squares. The resulting values, denoted in the lighter shade of blue, are point values along the stencil direction but are still averaged in the remaining directions. A new direction is picked, and the de-averaging stencil is applied to the blue values, producing further de-averaged values, denoted in a darker shade of blue. This process repeats recursively along all remaining directions within the plane until the final point value is produced.

direction until, in the final step, the point value is produced. This process is depicted in Figure 3. The reconstruction process is thus accomplished with two stencil formulas. The first stencil is applied to the cell average to produce the face average, and the second stencil is then applied successively along each face dimension to produce the point value at the center of the face.

The original fourth-order methods by Colella *et al.*<sup>6:33</sup> used a streamlined reconstruction approach that exploited the fact that face-averaged values and face-centered point values agree to second-order accuracy, as can be seen in formula (9). Unfortunately, that technique can only be applied to methods of fourth-order or less. The approach described here is applicable to methods of arbitrary order.

### 3.2 Upwinded discretization

When of odd order of accuracy, the stencil for reconstructing the face averaged values utilizes an odd number of points, and therefore an unequal number of points on

each side of the face. Using an upwind-biased stencil has the advantage of introducing numerical diffusion along the direction normal to the face, which helps to improve stability. For a system, the upwind direction is determined by computing a value  $\langle u \rangle_{i+\frac{1}{2}e^d}^L$  using the stencil biased by a point to the left, computing the value  $\langle u \rangle_{i+\frac{1}{2}e^d}^R$  via the stencil biased by a point to the right, and using an exact or approximate Riemann solver  $H(\langle u \rangle_{i+\frac{1}{2}e^d}^L, \langle u \rangle_{i+\frac{1}{2}e^d}^R)$  to determine the constant solution along  $\mathbf{x}_{i+\frac{1}{2}e^d}/t = 0$ . In the case when the system (1) is multi-component, doing so effectively chooses the upwind direction on a per characteristic variable basis. An example of the two stencils is shown in Figure 2(b).

For simplicity, the remaining stencils of the step are computed using even-ordered centered stencils of the next highest order of accuracy. Specifically, the reconstruction of face-centered point values from the face averages and the computation of the face averages of the flux using the Taylor expansion are computed using centered formulas. These stencils are computed within the plane of the face, perpendicular to the normal direction of the face.

### 3.3 Streamlined formulations for scalar advection problems

Following the approach of Colella *et al.*<sup>6;33</sup>, methods up to ninth-order were derived for scalar advection equations in Chaplin and Colella<sup>35</sup>. The particular nature of the equations allows the methods to avoid the full reconstruction of the point-values on the faces, and instead compute the flux-averages directly from face-averaged values. Such an optimization lowers both the cost and arithmetic intensity of the methods, but is specific to that particular class of equations. In this paper, we focus our attention on methods general to fully nonlinear problems.

### 3.4 Temporal Discretization

To this point, we have adopted an MOL perspective and considered only the spatial discretization. Any high-order explicit ODE method will suffice for the temporal discretization provided that the eigenvalues of the spatial

operator are contained within the stability region of the chosen ODE method. For the even-order spatial discretizations presented here, for a linear system the eigenvalues will be pure imaginary, and linear stability requires that the time integration method contain some non-trivial segment of the imaginary axis about the origin. This is insufficient for variable-coefficient and nonlinear problems, which will require the addition of dissipation (such as artificial hyperviscosity) to the even-order flux discretizations.<sup>6</sup> We consider primarily explicit time integrators because hyperbolic systems naturally describe the propagation of signals, and so time accuracy is typically required. A simple choice is to use the standard explicit, four-stage, fourth-order Runge-Kutta method,<sup>6;8</sup> although an entire class of time integrators have been developed with a stronger notion of stability for nonlinear hyperbolic systems in MOL form.<sup>36</sup>

From the perspective of arithmetic intensity for these problems, it is most informative to consider the evaluation of the flux divergence operator (effectively, the right-hand side (RHS) of the ODE (6)). The majority of floating point operations occur in the evaluation of this RHS operator. Furthermore, the inputs of the RHS, primarily the solution state, are the same as for the calling time integration method. For multi-stage methods like Runge-Kutta, the flux divergence will be computed at least once for each stage, and the predicted states resulting from the stage evaluations are weighted and summed. Given that the RHS evaluation is dominant, we will not consider the details of the time stepping and will limit our attention to the approximation of the RHS of equation (6).

### 3.5 Limiting

Another issue for the practical application of the discretizations considered here is high-order nonlinear limiting, either of the reconstruction, as in the Piecewise Parabolic Method<sup>34</sup> or the Weighted Essentially Non-Oscillatory (WENO) methods,<sup>37-39</sup> or the flux, as in flux-corrected transport (FCT) methods.<sup>40;41</sup> Nonlinear limiting alters the stencil and/or reduces the order of the method in order to capture discontinuities in the solution without

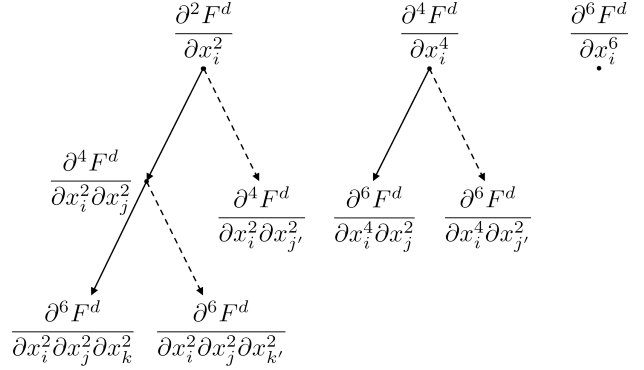
introducing numerical oscillations. Limiters can be viewed as an additional set of computations done for each flux evaluation that require no additional loads and stores, so on net they only increase the arithmetic intensity. The stencils we consider here are building blocks for each of these types of limiter, so we choose to consider the unlimited case for clarity and leave further specialization to a variety of limited schemes for future work.

### 3.6 The steps of the method

We now give a step-by-step summary of the method for evaluation of the flux divergence, which will allow us to refer to specific steps in the analysis section. We will use  $s$  to denote the order of the method. We limit our attention to the cases where  $3 \leq s \leq 8$ . Since odd order methods still use even-ordered centered stencils for the terms that lie within the plane of the faces,  $\hat{s}$  will denote the order of the method rounded up to the next highest even number.

#### 1. Reconstruct the state point values at the face centers by using the cell averages.

- (a) For every face  $\mathbf{i} + \frac{1}{2}\mathbf{e}^d$ , interpolate from cell-averaged  $\bar{u}_i$  to face-averaged  $\langle u \rangle_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}^d$  using the primitive function approach. In this step, even-ordered and odd-ordered methods are handled differently. For even-ordered methods, a single stencil of length  $s$  is computed. For odd-ordered methods, two stencils each of length  $s$  are computed, requiring a total of  $s + 1$  points. The two stencil results are passed to a Riemann solver, and the result is used as the face average value.
- (b) For every face  $\mathbf{i} + \frac{1}{2}\mathbf{e}^d$ , compute the face-centered  $u_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}^d$  by iteratively performing a one-dimensional reconstruction along each dimension  $d' \neq d$ . For one-dimensional problems, the average and point value are the same, so this operation is absent. In all other cases, the stencil is  $\hat{s} - 1$  in length and is centered and symmetric about the face center point.



**Figure 4.** Higher-order mixed-partial derivatives are computed using previously computed lower-order derivatives. The flow of dependency is from top to bottom. For example, the  $\partial^4 F^d / \partial x_i^2 \partial x_j^2$  terms are computed from the  $\partial^2 F^d / \partial x_i^2$  terms.

2. **For every face  $\mathbf{i} + \frac{1}{2}\mathbf{e}^d$ , use the face-centered state  $u_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}^d$  to compute the flux terms**

$$F^d(u_{\mathbf{i} + \frac{1}{2}\mathbf{e}^d}^d) \quad (21)$$

3. **For every face  $\mathbf{i} + \frac{1}{2}\mathbf{e}^d$ , compute the face-averaged fluxes using the Taylor expansion of the face-average integral (9).**

For one-dimensional problems, the average and point value are the same so this step does not exist. In all other cases, the average is computed using derivative terms up to an order of derivative two less than the order of accuracy. For example, sixth-order accurate methods must compute the second and fourth-order derivatives. The accuracy required per term is equal to the order of the method minus the order of the derivative term. For example, for a sixth-ordered method, the second-order derivative term must be computed to fourth-order accuracy and the fourth-order derivative terms (both single and mixed partial derivatives) must be computed to second-order accuracy. All derivative terms are computed using centered finite-difference stencils of appropriate width that are symmetric about the face center point. Higher-order mixed-partial derivatives are computed using lower-order derivatives. The dependency relationships up to eighth order are shown in Figure 4.

- (a) For methods of order three and above, each second-order derivative term

$$h^2 \frac{\partial^2 F^d}{\partial x_{d_i}^2} (u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d}^d) \approx \sum_{j=0}^{\hat{s}-2} c_j u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d+(j-\frac{s-2}{2})\mathbf{e}^{d_i}}^d \quad (22)$$

is computed with a stencil  $\hat{s} - 1$  in length. For problems of dimension three or greater, the terms are also stored for later use in computing higher-order mixed partial derivative terms.

- (b) For methods of order five and above, each fourth-order derivative term

$$h^4 \frac{\partial^4 F^d}{\partial x_{d_i}^4} (u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d}^d) \approx \sum_{j=0}^{\hat{s}-2} c_j u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d+(j-\frac{s-2}{2})\mathbf{e}^{d_i}}^d \quad (23)$$

is computed with a stencil  $\hat{s} - 1$  in length. For problems of dimension three or greater, the terms are stored for later use in computing higher-order mixed partial derivative terms.

- (c) For methods of order five and above on problems of dimension three or greater, each fourth-order mixed partial derivative

$$h^4 \frac{\partial^4 F^d}{\partial x_{d_i}^2 \partial x_{d_j}^2} (u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d}^d) \quad (24)$$

$$\approx \sum_{j=0}^{\hat{s}-4} c_j u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d+(j-\frac{s-4}{2})\mathbf{e}^{d_i}}^d \quad (25)$$

is computed by calculating the second-order derivative along direction  $d_j$  of the second-order derivative along  $d_i$ , using the values stored previously. The stencil is  $\hat{s} - 3$  in length. For problems of dimension four or greater, the terms are stored for later use in computing higher-order mixed partial derivative terms.

- (d) For methods of order seven and above, each sixth-order derivative

$$h^6 \frac{\partial^6 F^d}{\partial x_{d_i}^6} (u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d}^d) \approx \sum_{j=0}^{\hat{s}-2} c_j u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d+(j-\frac{s-2}{2})\mathbf{e}^{d_i}}^d \quad (26)$$

is computed to second-order accuracy with a stencil  $\hat{s} - 1$  in length.

- (e) For methods of order seven and above, for problems of dimensions three or greater, each sixth-order mixed partial derivative in two variables

$$h^6 \frac{\partial^6 F^d}{\partial x_{d_i}^4 \partial x_{d_j}^2} (u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d}^d) \quad (27)$$

$$\approx \sum_{j=0}^{\hat{s}-6} c_j u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d+(j-\frac{s-6}{2})\mathbf{e}^{d_i}}^d \quad (28)$$

is computed to second-order accuracy using the stored values for the fourth-order accurate, fourth-order derivatives. The stencil is  $\hat{s} - 5$  in length.

- (f) For methods of order seven and above, for problems of dimension four or greater, each sixth-order mixed partial derivative in three variables

$$h^6 \frac{\partial^6 F^d}{\partial x_{d_i}^2 \partial x_{d_j}^2 \partial x_{d_k}^2} (u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d}^d) \quad (29)$$

$$\approx \sum_{j=0}^{\hat{s}-6} c_j u_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d+(j-\frac{s-6}{2})\mathbf{e}^{d_i}}^d \quad (30)$$

is computed to second-order accuracy using the stored values for the second-order accurate fourth-order mixed partial derivatives in two variables. The stencil is  $\hat{s} - 5$  in length.

#### 4. For every cell $\mathbf{i}$ , compute the cell-averaged divergence using the face-averaged fluxes through

$$-\frac{1}{h^D} \int_{V_{\mathbf{i}}} \nabla \cdot \vec{F} dx \approx -\frac{1}{h} \sum_{d=1}^D \langle F^d \rangle_{\mathbf{i}+\frac{1}{2}\mathbf{e}^d} - \langle F^d \rangle_{\mathbf{i}-\frac{1}{2}\mathbf{e}^d} \quad (31)$$

Rather than compute the method in a step-by-step manner as described, it is tempting to consider implementing the method by combining the stencils into a single fused stencil and calculating the divergence for each cell at once. The stencil for each cell would be a block stencil with the same number of dimensions as the problem. The arithmetic intensity of such an approach

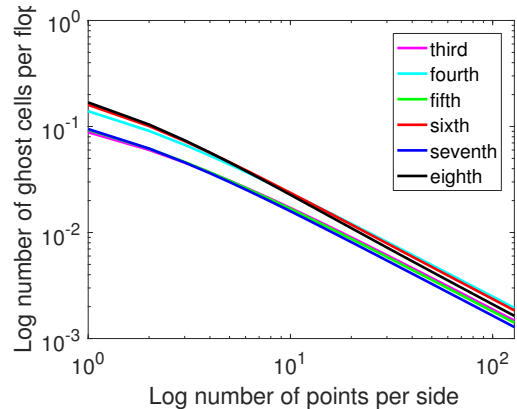
is far higher than the step-by-step approach, exceeding a hundred on three-dimensional problems using boxes of  $32^3$  for example. This is because there is no sharing of intermediate calculations with neighbors, so intermediate results are effectively being re-computed, driving up the number of flops. Such an approach is infeasible on current architectures, and is difficult to understand and code without automatic code generation, so we will not consider it further.

### 3.7 Implementation on box-structured grid frameworks and ghost widths

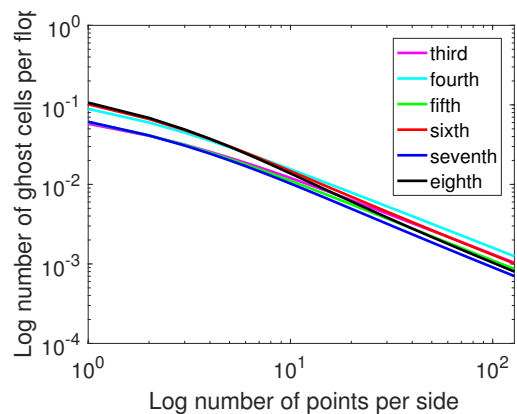
For large-scale problems implemented using structured grids, it is common to subdivide the domain into smaller disjoint rectangular boxes that can be computed in parallel. At each step, boxes are surrounded by a layer of ghost cells containing the neighboring data upon which the box is dependent, allowing computation on boxes to be done independently. At the edge of the problem domain, boundary data is used to fill the ghost cells. Boxes are distributed over processors, and each box is assigned to one processor. A processor may possess multiple boxes, and these boxes can be computed in parallel on a multi-core processor, either by assigning a thread to each box and/or by assigning multiple threads within a box. At the beginning of each step (and as necessary during a step), processors communicate to exchange ghost data. Examples of box-structured frameworks that use this procedure in some form include Chombo<sup>42</sup>, BoxLib<sup>43</sup>, SAMRAI<sup>44</sup>, Overture<sup>45</sup>, Clawpack<sup>46</sup>, and AMROC<sup>47</sup>.

(This paragraph was originally at the end of the section.)

A complication of implementing the current class of methods on such frameworks is that the width of the ghost region increases with order. For each direction  $d$ , the derivative stencils in Step 3 have dependencies that extend in each direction orthogonal to  $d$  (in the plane of the face) up to  $\frac{\hat{s}}{2} - 1$  faces per side (i.e., in both the “left” and “right” directions of each direction). However, each of the faces involved in a derivative stencil must first be reconstructed from face-averaged solution values using a stencil that itself extends the dependencies in each direction by  $\frac{\hat{s}}{2} - 1$  faces.



(a) 2D



(b) 3D

**Figure 5.** Ratio of the number of ghost cells to the amount of work in the interior of a  $D$ -dimensional box with sides of length  $N$  for schemes of several orders. Note that, for a given bias (all even or all odd schemes), the higher-order schemes have smaller ratios for a given  $N$ .

This results in an overall extension of  $\hat{s} - 2$  faces in each direction orthogonal to  $d$ . Finally, the face-averaged values must be interpolated from cell-averaged values by stencils that have dependencies extending by  $\frac{\hat{s}}{2}$  in each direction along  $d$ . Over all  $d$ , the final ghost width of the box then becomes  $w = \hat{s} - 2$  cells in each direction. For example, the eighth-order method has a ghost width extending six cells in each direction, i.e., six in the “left” and six in the “right” directions, for a total of twelve cells per dimension.

It is straightforward to see that the total number of ghost cells surrounding a box of length  $N$  cells per side in  $D$

dimensions is

$$(N + 2w)^D - N^D = 2wDN^{D-1} + \sum_{d=2}^D \frac{D!}{(D-d)!d!} N^{D-d}(2w)^d. \quad (32)$$

We can make a worst-case relative estimate of how the amount of data transferred by internode-communication compares to the work on the box as a function of order and box size. If we assume one box per node, then all ghost-filling operations are off-node; in practice, there are typically multiple boxes per node, so not all communications between boxes are off-node. The number and size of inter-node messages is very dependent on the algorithm used to distribute boxes across the nodes. Nevertheless, this assumption should show the relative behavior as order increases. Using the estimate for the total flops for the infinite-cache case (Section 4, Table 4), which asymptotically is  $O(N^D)$ , we plot the ratio of ghost cells to flops in Figure 5. One sees that the expected asymptotic behavior  $1/N$  occurs by roughly  $N = 100$ , that the lower-order schemes of the same bias (even/central or odd/upwind) are all ordered such that the ratio is always smaller for higher-order methods, and that the relative cost of communication can be reduced by increasing the patch size. Thus, we expect a worse performance penalty for off-node ghost cell transfers for lower-order schemes.

Olschanowsky et al. studied approaches to optimizing the performance of high-order finite-volume methods (up to fourth-order in the study) that are implemented using box-structured frameworks<sup>21</sup>. To combat the overhead of wider ghost regions, they improved the ratio of ghost cells to interior cells by increasing the box sizes on 3D problems to  $128^3$ , compared to a more typical  $16^3$  or  $32^3$ . A consequence of this approach is that boxes no longer fit within cache, and the on-node performance and thread scalability of a non-tiled implementation suffered greatly as a result. To recover good on-node performance, they compared a variety of loop schedules for cache tiling and thread parallelism. Simple cubical tiling, with threads applied either one per cache tile or one per box, were the best performing schedules, and they recovered the

performance and ideal thread scalability of small boxes. This approach thus allowed a good ghost-cell-to-interior-cell ratio that comes from using large boxes without a penalty to on-node performance.

Under box-structured frameworks, the on-node cost of computing a step of the method can be understood simply in terms of the computation over a single box. We assume the use of this approach and therefore focus on just the per box memory and flops costs, taking into consideration the size of the box and the width of the ghost layer. We do not explicitly consider the cost of interprocessor communication used to populate the ghost layer between steps, but we do include ghost cell counts in load estimates. However, as in Olschanowsky et al.<sup>21</sup>, in order to maintain acceptable internode performance, we do assume high-order methods will need larger boxes than the cache-friendly sizes traditionally used. As such, an implementation will require a cache tiling strategy to attain high AI, which is a matter we discuss at length. As in Olschanowsky et al.<sup>21</sup>, we assume threads are assigned either one per cache tile or one per box, with enough tiles or boxes available per multicore processor to keep all cores busy.

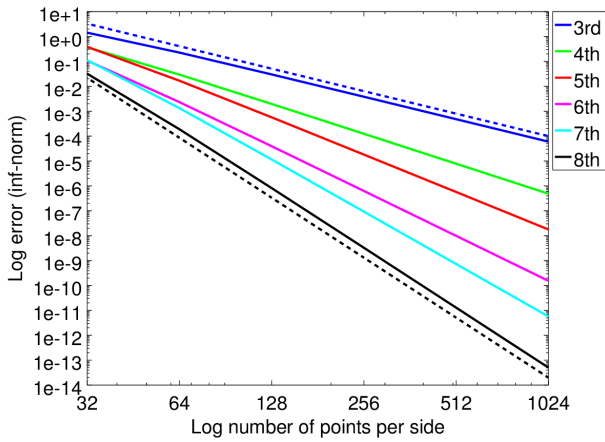
### 3.8 Numerical verification of the orders of accuracy

To verify that the derived methods achieve the expected orders of accuracy, the methods were tested on the two-dimensional and three-dimensional scalar advection problem

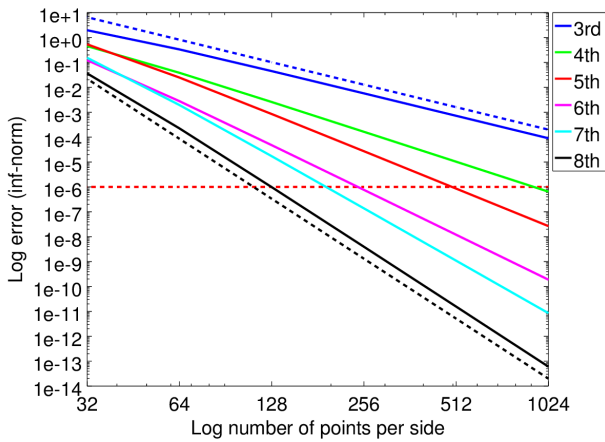
$$\frac{\partial u}{\partial t} + \nabla \cdot u\mathbf{a} = 0, \quad (33)$$

with periodic boundary conditions and with  $x_i \in [0, 1]$ . For the two-dimensional case, the propagation velocity,  $\mathbf{a}$ , was chosen as  $\mathbf{a} = (1, 1)^T$ , and in the three-dimensional case as  $\mathbf{a} = (1, 1, 1)^T$ . The initial conditions were given by sinusoidal data

$$u_0(\mathbf{x}) = \prod_{d=1}^D \sin(2\pi x_d). \quad (34)$$



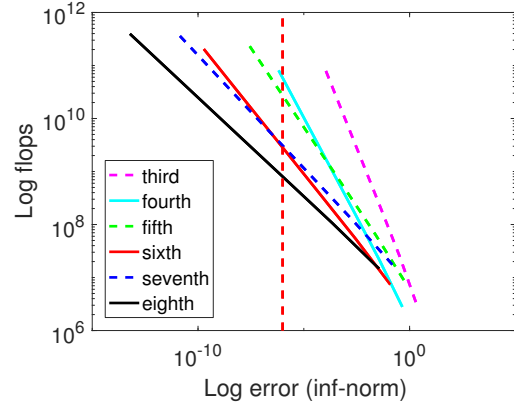
(a) Order of accuracy on a 2D scalar advection problem.



(b) Order of accuracy on a 3D scalar advection problem.

**Figure 6.** For both the 2D and 3D problems, the methods achieve the predicted orders of accuracy when the grid density is sufficiently fine. As a visual aid, the dashed blue line in each figure is a straight line with a slope of  $2^3$ , and the dashed black line has a slope of  $2^8$ . The horizontal dashed red line in plot for the 3D problem corresponds to an error or  $10^{-6}$ . Higher-order methods achieve that level of accuracy with a significantly coarser mesh than lower-order methods, making them **computationally** more memory efficient.

The flux divergences of the initial conditions were computed with the methods of orders three through eight, and the results were compared against the analytically computed solutions. The max-norm measure of the errors are plotted as a function of the grid density in Figure 6. The same data is also given numerically in Tables 13 through 24 in the Appendix. Dashed lines are shown for



**Figure 7.** The number of flops computed to achieve a particular accuracy on the 3D test problem. The accuracy was controlled by varying the grid density. Despite needing more flops than lower-order methods for the same grid size, higher-order methods can achieve the same accuracy using a smaller number of points, resulting in an overall lower cost to compute an equivalently accurate solution. The dashed red line corresponds to the same tolerance of  $10^{-6}$  as shown in Figure 6(b). Along the line, higher-order methods are lower than lower-order ones.

the theoretically expected slopes for the third- and eighth-order methods. We see that the two methods exhibit less than the expected order on coarse grids, but both methods achieve the expected order of accuracy as the grid density increases. For clarity, dashed lines were not added for the intermediate orders, but, when the grid is sufficiently dense, the expected accuracy is achieved in all other cases as well.

**3.8.1 Order of accuracy and computational cost.** The thesis of this work is that higher-order methods can significantly improve machine utilization over low-order ones. Higher-order methods make more *efficient* use of each byte moved by allowing the processor to minimize the amount of time it remains idle while data is transferred. Of course, the more usual reason to use a high-order method is to be able to produce a solution of sufficient accuracy using a smaller grid size. In other words, higher-order methods also *minimize* data movement. Before returning to the first issue in Section 4, we briefly demonstrate the reduction in time-to-solution higher-order methods achieve by virtue of higher accuracy alone.

From the horizontal red dashed line in Figure 6(b), we see that the eighth-order method requires a grid size of

about  $128^3$  to achieve an error below  $10^{-6}$ , the sixth-order method requires a grid size of about  $256^3$ , and the fourth-order method needs a grid of size  $1024^3$ . Using a smaller grid not only saves memory, but also reduces the amount of computational work required to achieve the error tolerance. Figure 7 plots the flops required to achieve a given error tolerance on the 3D test problem. The flops were computed using the formulas derived in Section 4 that relate the number of flops to grid size. The vertical red dashed line corresponds to the error tolerance of  $10^{-6}$  shown in Figure 6(b). We see that higher-order methods are predicted to require an order of magnitude fewer flops than the next lowest-order method of the same type (centered or biased) to achieve the same accuracy.

Figure 7 is based on a predicted flops cost, but we can see the same effect by measuring the time to solution for the method when computing the test problem. Table 1 lists the time in seconds required by our implementation of each method to compute the flux divergence for the problem over grids ranging in size from  $128^3$  to  $1024^3$ . The implementation is written in C++ and compiled with version 4.9.3 of the GNU compiler, is single-threaded, and was run on an Intel Sandy Bridge E5-2670 machine. The reported times are the average over ten trials for each case. The numbers in red are the times for each order of the method when computing the smallest grid size that achieves a solution with infinity-norm error lower than  $10^{-6}$ . The eighth-order method requires less than one-fifth the time as the sixth-order method, which in turn needs about one-fiftieth the time as the fourth-order method. However, for the same size of grid, the sixth-order method requires up to 45% more time than the fourth-order method, and the eighth-order method requires up to 50% more time than the sixth-order one. The larger overhead of the higher-order methods is due to the high number of integer operations incurred by our implementation to compute array offsets. Since the purpose of the implementation was to measure flops cost and data movement, and since local array-offset integer calculations have no effect on the flops count or the number of bytes moved, we chose convenience of implementation over integer operation efficiency. In particular, the approach taken to accommodate problems

with an arbitrary number of spatial dimensions results in an integer operation cost much higher than would be found in a production implementation. The time cost for the methods therefore becomes proportional to the number of stencils that must be computed, which is larger in higher-order methods as they must compute more derivative terms.

Note that even with an integer-bound implementation, for a particular size of grid, the overhead of using a higher-order method is a constant factor inflation over a lower-order method. However, the grid size needed by the eighth-order method for a given tolerance scales by  $O(h^2)$  relative to the sixth-order method and by  $O(h^4)$  relative to the fourth-order method. The mathematical accuracy of the higher-order methods therefore scales more quickly than the overhead due to implementation. We see this effect in the timings for the test problem, which demonstrates that using high-order methods can have a profound effect on the time to solution for a given accuracy requirement.

## 4 Arithmetic intensity analysis

In this section, we derive formulas for the flops and data transfer costs of the methods as functions of the box size, the number of dimensions of the problem, and the order of accuracy. We assume the boxes are cubical in shape and of length  $N$  on a side when not including ghost cells. The number of dimensions in the problem is  $D$ . Except for the cost of the Riemann solver and the cost of the flux function, both the flops and data transfer costs increase in proportion with the number of system components, i.e., a system with  $k$  components has  $k$  times as many floating point and memory operations as a system with a single component. Therefore, for brevity, the formulas are given for the single component case. The costs of the Riemann solver and flux function are problem specific, so we leave their costs as parameters in the formulas, denoting their flops costs per face as  $f_R$  and  $f_F$  respectively.

Our overall strategy is the following. As a starting point, we first count operations in the degenerate case where no cache is used. This provides an upper bound on the number of memory operations that must be performed and a lower bound on arithmetic intensity. This case is

Points/side	Order				
	4th	6th / 4th	6th	8th / 6th	8th
N=128	0.34	138%	0.47	148%	0.70
N=256	2.63	137%	3.59	148%	5.29
N=512	20.6	137%	28.2	148%	41.7
N=1024	172	142%	244	152%	371

**Table 1.** Time cost in seconds for the fourth-, sixth-, and eighth-order methods when computing the flux divergence for the 3D scalar advection problem. The red highlighted numbers indicate the time required to achieve an accuracy of  $10^{-6}$  for that particular order of the method. The ability to use a coarser mesh to achieve equivalent accuracy allows the higher-order methods to reach the error tolerance with greatly less compute time than lower-order methods.

relevant to implementations without a tiling strategy where the cache utilization is typically poor. We then count the number of operations in the case of a fictional infinite size cache. This gives a lower bound on the number of memory operations and gives the limit to performance in the ideal. We then calculate the number of operations for a finite size cache when using a simple tiling strategy. Our goal is not to determine the optimal arithmetic intensity under the best tiling strategy, but rather demonstrate that higher-order methods show a worthwhile degree of improvement to intensity even with a basic tiling strategy. A more sophisticated implementation approach would do even better. Finally, we compute the arithmetic intensity for a caching strategy with lower cache space requirements in the case of 3D problems.

#### 4.1 Machine model and operation costs

For the purposes of calculating arithmetic intensity, we assume the following abstract machine model. The machine is composed of a processor and two levels of memory: an unlimited amount of slow memory and a limited amount of fast memory. Data can be transferred in both directions between slow and fast memory. Data transferred to a location in either memory overwrites the previous value held there. At the start of processing, all data is stored in the slow memory and the fast memory is empty. Computation is only performed by the processor on data residing in the fast memory, so any data to be operated on must first be transferred from slow memory. Computation comes in the form of arithmetic operations that load one or more operands from fast memory and return the result of the

computation back to fast memory, never to slow memory. The problem is not considered complete until the final result resides in slow memory.

The two costs in this model are data transfers between slow and fast memory and the cost of the arithmetic operations. Data transfers are considered to have the same cost whether to or from fast memory. All arithmetic operations are considered to have the same cost. In particular, the method requires only addition and multiplication operations. Fractional terms, such as the stencil coefficients, can be assumed to be pre-computed by the compiler, so we consider the method to be free of division operations.

It is common on current processors, both CPUs and accelerators, for floating point operations to be computed in a vectorized manner. For example, rather than computing  $c = a + b$ , where  $a$ ,  $b$ ,  $c$  are all single floating point numbers, four such operations might be grouped together and executed in one instruction as the vector operation  $\vec{c} = \vec{a} + \vec{b}$ , where  $\vec{a}$ ,  $\vec{b}$ , and  $\vec{c}$  all have length four. For the purposes of counting operations, we adopt the Golub and Van Loan definition of flops<sup>48</sup>. In particular, we will consider the latter case to be four floating point operations. Similarly, most current architectures supply fused-multiply-add (FMA) operations of either the form  $a = a + b * c$  or  $d = a + b * c$ . We will consider scalar FMA to account for two floating point operations, and vectorized FMA to compute a number of operations two times the length of the vector.

The above model can be mapped onto current CPU architectures where DRAM is considered the slow memory

and the last level of cache is considered the fast memory. We ignore the benefits of multi-level caches and assume only a single level. The arithmetic intensity analysis of the methods is not affected by whether data transfers to and from fast memory can explicitly schedule the locations to which data is stored and written or whether the fast memory is treated like a cache with an automatic replacement policy. For concreteness, we will assume the fast memory is a fully associative cache with a strict least-recently-used replacement policy.

Multi-core CPUs generally share the last level of cache across cores. To that extent, the model accommodates multi-threading in the limited sense that there is no question of data being in a level of cache visible to some threads but not to others. There is still the issue of how threads operating in parallel affect the scheduling of what data is resident in the cache at what time. Because we are assigning a single thread to a box or cache tile, threads can only affect each other via data common between them due to overlap of a stencil applied at the periphery of a box or tile. Specifically, a thread might benefit from having data along the boundary of its box or tile be pre-loaded into cache by one of its neighbors. In this study, we conservatively ignore such a benefit and assume each thread must load all data it needs for a block or tile by itself.

Clearly the computation and data transfer costs for the algorithm are both minimized if the size of fast memory is large enough to hold all intermediate results, i.e., if the only data transfers that occur are at the beginning of the problem, when loading the entire problem data into fast memory, and at the end, when transferring the final result to slow memory. In general, the size of fast memory is too limited to allow this, so additional memory transfers of intermediate results are necessary. Note that the case of unlimited fast memory is not necessarily an upper bound on arithmetic intensity. To minimize the amount of data that must be kept in fast memory, an implementation might redundantly compute some intermediate results, such as halo data, and this can actually increase the arithmetic intensity over the optimal case. However, the number of operations would be higher as well, possibly to a degree that results in a net loss in performance. Therefore, a tiling

strategy with an AI that is higher than the infinite-cache case would not necessarily be a higher performing strategy. The infinite-cache case is simply the case that minimizes the total number of operations of the algorithm.

## 4.2 Subexpression elimination

The number of flops incurred when computing the stencils depends on how terms are grouped. Combining terms to eliminate redundant calculation of common subexpressions minimizes the number of flops. The most basic example of this is first adding terms in a stencil that share the same coefficient, e.g. computing  $au_{i-1} + bu_i + au_{i+1}$  as  $a(u_{i-1} + u_{i+1}) + bu_i$ , which eliminates one multiplication by  $a$ . When counting flops in the subsequent sections, we have grouped terms in this manner to minimize the number of multiplications.

Beyond that, the primary means of reusing prior computations is using lower-order derivative terms to compute higher-order ones, as shown in Figure 4. Computing the final flux average of equation (9) in this manner is a highly effective way of minimizing the number of flops. If the stencils of equation (9) were combined into a single multi-dimensional block stencil, there would be no reuse of prior computations at all, and the AI for 3D problems would be in the hundreds.

## 4.3 The no-cache case

As a starting point, we begin by counting operations under the assumption that there is no cache reuse of data between cells or faces. The no-cache case provides a lower bound on the arithmetic intensity in the case where the cache is poorly utilized. We assume there is just enough local storage (“registers”) to hold all data needed to compute the stencil at each cell or face, but when moving to a new cell or face, all involved data must be loaded from slow memory anew. We also assume there is no re-use of data between steps. Note that, because there is no use of cache and because each stencil must load its operands anew, multi-threading does not affect the total number of data transfers or flop computations, i.e., there is no data reuse.

When computing stencils over a large domain of data, the amount of data being iterated over may greatly exceed the size of cache. The computation can degenerate into a “streaming” regime where the data transferred to compute a cell may need to be re-transferred when later computing a neighbor due to the large data volume. This case acts as a worst-case bound of the performance for such a situation. As a lower bound, it is more pessimistic than the expected performance in practice for larger-than-cache situations. See Olschanowsky et al.<sup>21</sup> for measurements of the performance degradation in such a case of high-order finite-volume type stencil calculations.

For illustration of computing the formulas, we break-down the costs for the first step of the eighth-order algorithm and then simply list the formulas for the remaining steps in Table 2 for the flops and Table 3 for the memory operations. Since the formulas for the seventh-order method differ only in step 1a, we list the formulas for that case with the eighth-order formulas. The formulas for the lower-order methods are given in the Appendix. Note that when using double-precision floating-point values, each transfer is eight bytes.

#### 4.3.1 Floating point operations

Step 1: Reconstruct the point values at the face centers.

- (a) For each face, interpolate from cell-averaged  $u$  to face-averaged  $u$ .
  - (i) Sum the pairs of terms in each parenthesized group. 4 *additions*.
  - (ii) Multiply each summed pair by a coefficient. 4 *multiplications*.
  - (iii) Sum the resulting four terms. 3 *additions*.

To determine the numbers of faces, we first note that, along any direction  $D$ , there are  $(N + 1)$  planes of faces that touch the interior  $N^D$  cells of a box. There are  $(N + 12)$  cells in each of the  $(D - 1)$  directions transverse to each plane; recall that the additional 12 cells per direction, 6 at each of the low and high boundaries (at eighth order), are the halo cells needed for subsequent stencils. Thus, the number of faces in

any plane is  $(N + 12)^{D-1}$ , there are  $(N + 1)$  planes per direction, and there are  $D$  directions for a total of  $D(N + 1)(N + 12)^{D-1}$  faces that must be computed in this step.

Total flops: 11 flops per face for a total of approximately  $11D(N + 1)(N + 12)^{D-1}$  flops. Here and in subsequent estimates, the count is approximate because we include reconstructions at *all* faces in the ghost regions, even if they are not used in the final update. This is a small number of additional faces (relative to the used faces) because they occur near the outer edges of the  $(N + 12)^D$  box. These edges are of codimension two or greater. Furthermore, a realistic implementation would likely include these calculations to avoid the complicated logic of determining only the faces that contribute.

- (b) For each face, iteratively compute an approximation to face-centered  $u$ . For each of  $(D - 1)$  iterations, one iteration per direction lying in the plane of the face:
  - (i) Sum the pairs of terms in each parenthesized group. 3 *additions*.
  - (ii) Multiply the four coefficients. 4 *multiplications*.
  - (iii) Sum the four resulting terms. 3 *additions*.

At the start of the sub-step, each plane of faces is  $(N + 12)$  faces wide in each of the  $(D - 1)$  directions by the same argument as the previous sub-step. However, at each iteration over transverse directions, the width of the face is reduced by six faces along the direction computed by that iteration. The six faces were the halo faces needed by the reconstruction stencil, and they are not needed along that direction in subsequent iterations. An illustration of this effect is shown in Figure 8. The total number of faces computed in this step is therefore  $D(N + 1) \sum_{i=1}^{D-1} (N + 6)^i (N + 12)^{D-1-i}$ .

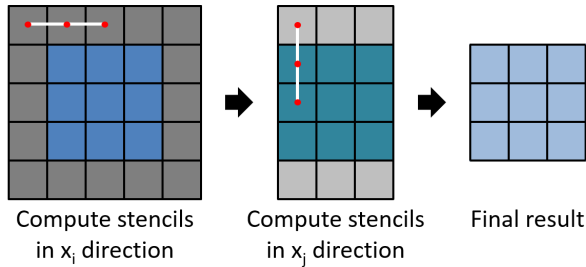
Total flops: 10 flops per face for a total of approximately  $10D(N + 1) \sum_{i=1}^{D-1} (N + 6)^i (N + 12)^{D-1-i}$  flops.

#### 4.3.2 Data transfer operations

Step 1: Reconstruct the point values at the face centers.

Step	Flops
1a-eighth	$11D(N+1)(N+12)^{D-1}$
1a-seventh	$(26 + f_R)D(N+1)(N+12)^{D-1}$
1b	$10D(N+1) \sum_{i=1}^{D-1} (N+6)^i (N+12)^{D-1-i}$
2	$f_F \times D(N+1)(N+6)^{D-1}$
3a	$D(N+1) \left[ 10 \sum_{i=1}^{D-1} N^i (N+4)^{D-1-i} + 2(D-1)N^{D-1} \right]$
3b	$10D(D-1)N(N+1)(N+2)^{D-2} + 2D(D-1)(N+1)N^{D-1}$
3c	$7D(N+1) \sum_{i=1}^{D-1} \sum_{j=i+1}^{D-1} N^j (N+2)^{D-1-j} + 2D \binom{D-1}{2} (N+1)N^{D-1}$
3d	$11D(D-1)(N+1)N^{D-1}$
3e	$5D(D-1)(D-2)(N+1)N^{D-1}$
3f	$5D \binom{D-1}{3} (N+1)N^{D-1}$
4	$2DN^D$

**Table 2.** Flops per step for seventh- and eighth-order methods. Valid for the no-cache, infinite-cache and cubically tiled case.



**Figure 8.** Each iteration of the reconstruction of face-valued  $u$ , the halo values for that direction are “consumed”. This results in the width of the plane of faces being reduced in each iteration. In the figure, the grey colored squares represent halo faces, and the colored squares are the non-halo faces. A change in color indicates the face was updated by the stencil calculation. The direction of the stencil is indicated by the white line. In particular, note that the halo data for subsequent steps also needs to be reconstructed, not just the non-halo data.

- (a) For each direction, for each face on a box extending six halo cells in each direction transverse to the normal of the faces, interpolate from cell-averaged  $u$  to face-averaged  $u$ .
- Load each of the eight cell averages. *8 loads*.
  - Store the face average. *1 store*.

The total number of faces computed is  $D(N+1)(N+12)^{D-1}$ , as explained in the flops case.

Total transfers: 9 transfers per face for a total of approximately  $D(N+1)(N+12)^{D-1}$  transfers.

- (b) For each direction, for each face on a box extending six halo cells in each direction transverse to the normal of the faces, iteratively compute an approximation to face-centered  $u$ . For each of  $(D-1)$  iterations:

- Load seven face values. *7 loads*.
- Store the iterations result. *1 store*.

As explained in the flops case,  $D(N+1) \sum_{i=1}^{D-1} (N+6)^i (N+12)^{D-1-i}$  total faces are computed in this sub-step.

Total transfers:  $8(D-1)$  transfers per face for a total of approximately  $8(D-1)D(N+1) \sum_{i=1}^{D-1} (N+6)^i (N+12)^{D-1-i}$  transfers.

**4.3.3 Arithmetic intensity** The per-step arithmetic intensity for the no-cache case is plotted in Figure 9. When the length of the box is 128 or below, the arithmetic intensity is below 0.2 for all steps. In the asymptotic limit, the maximum arithmetic intensity is achieved by Step3a, with an intensity of 1.5 for a two-dimensional problem and of 1.33 for a three-dimensional problem. **Most current**

Step	Transfers
1a-even	$9D(N+1)(N+12)^{D-1}$
1a-odd	$9D(N+1)(N+12)^{D-1}$
1b	$8D(N+1) \sum_{i=1}^{D-1} (N+6)^i (N+12)^{D-1-i}$
2	$2D(N+1)(N+6)^{D-1}$
3a	$D(N+1) \left[ 7 \sum_{i=1}^{D-1} N^i (N+4)^{D-1-i} + \sum_{i=1}^{D-2} N^i (N+4)^{D-1-i} + (2D-3)N^{D-1} \right]$
3b	$8D(D-1)(N+1)N(N+2)^{D-2} + 2D(D-1)(N+1)N^{D-1}$
3c	$D(N+1) \left[ 5 \sum_{i=1}^{D-1} \sum_{j=i+1}^{D-1} N^j (N+2)^{D-1-j} + \sum_{i=1}^{D-2} \sum_{j=i+1}^{D-2} N^j (N+2)^{D-1-j} + 2 \binom{D-1}{2} N^{D-1} \right]$
3d	$9D(D-1)(N+1)N^{D-1}$
3e	$5D(D-1)(D-2)(N+1)N^{D-1}$
3f	$5D \binom{D-1}{3} (N+1)N^{D-1}$
4	$(2D+1)N^D$

**Table 3.** Data transfers per step in the no-cache case.

machines have a flops to byte ratio of 5 or greater. As expected, without a cache, a significant arithmetic intensity is not achievable.

#### 4.4 The infinite-cache case

If we assume an idealized machine with an unlimited size of cache, when cell or face data are first touched when computing a step, they would remain in cache for the rest of the step. Subsequent access of the data for computing neighboring stencils would not require another transfer of the data. At the end of the step, data that needs to be output would still be transferred out of the cache to slow memory. Of course, in an actual implementation, data would ideally remain in cache between steps. We will address that situation in the next subsection.

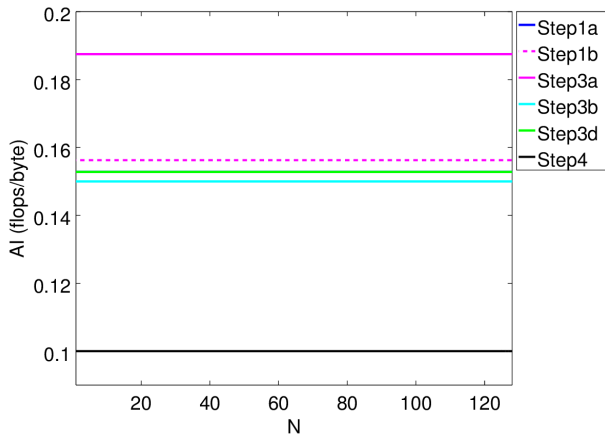
With an infinite-cache, the number of data values that must be loaded each step is the entire box worth of cells or faces that must be used to compute each step. Each value must be transferred exactly once. Likewise, the entire box worth of results must be written at the end of the step. Multi-threading may allow the values to be loaded or stored more quickly, but does not change the total number of values that

must be transferred, and therefore the formulas are agnostic to multi-threading in this case.

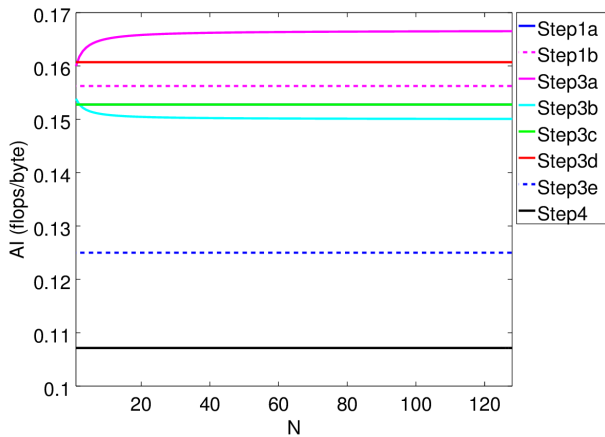
The number of floating point operations computed in this case is identical to the no-cache case, so the formulas are those in Table 2. The number of memory transfers for the infinite-cache case is listed in Table 4. Since the cache is infinite in size, the number of loads in each step is simply the number of cells or faces that must be touched for input, and the number of stores is the number of cells or faces that are written to for output. The breakdown for the first step is described as an example.

Step 1: Reconstruct the point values at the face centers.

- (a) Interpolate from cell-averaged  $u$  to face-averaged  $u$ .
  - (i) Including a ghost-width of six per side, there are  $(N+12)^D$  cells in a cubical box of length  $N$  per side. The corner cells of the ghost layer are not needed, but for simplicity we pessimistically assume they must be loaded. For approximately  $(N+12)^D$  cells, load  $\bar{u}_i$ .  $(N+12)^D$  loads.
  - (ii) As noted in the no-cache case, there are  $D(N+1)(N+12)^{D-1}$  faces in a cubical box of length  $N$  per side. Here too, the corner faces of each



(a) 2D problem. The lines for Step1a and Step3d overlap.



(b) 3D problem.

**Figure 9.** Arithmetic intensity in the no-cache case for each step of the eighth-order method as a function of box size. Without the use of cache, the arithmetic intensity is extremely low. The machine balance on current machines is between 5 and 10 and rising over time. Note that, particularly in 2D, the AI is independent of  $N$  in many cases because the form of the dependency on  $N$  for the number of flops and the number of bytes transferred is identical; with no cache, the amount of data transferred is often proportional to the number of operations. In higher dimensions, additional terms in both the numerator and denominator prohibit this cancellation (See Tables 2 and 3). As a reminder, Steps 3c and 3e only involve mixed partial derivative approximations that contribute in dimensions over 2D.

plane are not needed, but for approximately each of those  $D(N+1)(N+12)^{D-1}$  faces, store  $\langle u^d \rangle_{i+\frac{1}{2}\mathbf{e}^d}$ .  $D(N+1)(N+12)^{D-1}$  stores.

Total:  $(N+12)^D + D(N+1)(N+12)^{D-1}$  transfers.

(b) Iteratively compute an approximation to face-centered  $u$ .

(i) For approximately  $D(N+1)(N+12)^{D-1}$  faces, load  $\langle u^d \rangle_{i+\frac{1}{2}\mathbf{e}^d}$ .  $D(N+1)(N+12)^{D-1}$  loads.

(ii) For approximately  $D(N+1)(N+12)^{D-1}$  faces, store  $u^d_{i+\frac{1}{2}\mathbf{e}^d}$ .  $D(N+1)(N+12)^{D-1}$  stores.

Total:  $2D(N+1)(N+12)^{D-1}$  transfers.

**4.4.1 Arithmetic intensity** The per-step arithmetic intensity for the eighth-order method, the method with the highest arithmetic intensity, is plotted in Figure 10 as a function of box size for 2D and 3D problems. Step 2 is problem dependent so is not included. Even with the highest-order method, we see that when the length of the box is 128 or less, the arithmetic intensity remains below 3, even for a 3D problem. In the asymptotic limit, the arithmetic intensity is unbounded by problem dimension. For two- and three-dimensional problems, the intensity is bounded by 10, which is achieved by Step1b on three-dimensional problems. Even in the limit, most steps have an intensity of less than 5. **Most current machines have a flops to byte ratio of 5 or greater. Furthermore,** box sizes much larger than the main memory of current machines are needed to achieve such intensities on a per-step basis. In practice, a high arithmetic intensity is not achievable without keeping data in cache between steps.

In the idealized case of an infinite size cache, all data between steps are kept in the cache and only the input data for the first step and final result at the last step would be transferred. The total number of flops would still be the sum over all steps. Plots for the arithmetic intensity in such a case are shown in Figure 11 for the case of 2D and 3D problems. As can be seen, the aggregate arithmetic intensity is much higher than the per-step arithmetic intensity. For 2D problems, when the box size has length 128, the arithmetic intensity of the eighth order method is slightly

Step	Transfers
1a-eighth	$(N + 12)^D + D(N + 1)(N + 12)^{D-1}$
1a-seventh	$(N + 12)^D + D(N + 1)(N + 12)^{D-1}$
1b	$D(N + 1)(N + 12)^{D-1} + D(N + 1)(N + 6)^{D-1}$
2	$2D(N + 1)(N + 6)^{D-1}$
3a	$D(N + 1)(N + 6)^{D-1} + D(N + 1) \sum_{i=1}^{D-2} N^i (N + 4)^{D-1-i} + D(N + 1)N^{D-1}$
3b	$D(N + 1)(N + 6)^{D-1} + D(D - 1)N(N + 1)(N + 2)^{D-2} + 2D(N + 1)N^{D-1}$
3c	$D(N + 1) \sum_{i=1}^{D-2} N^i (N + 4)^{D-1-i} + D(N + 1) \sum_{i=1}^{D-2} \sum_{j=i+1}^{D-2} N^j (N + 2)^{D-1-j} + 2D(N + 1)N^{D-1}$
3d	$D(N + 1)(N + 6)^{D-1} + 2D(N + 1)N^{D-1}$
3e	$D(D - 1)N(N + 1)(N + 2)^{D-2} + 2D(N + 1)N^{D-1}$
3f	$D(N + 1) \sum_{i=1}^{D-2} \sum_{j=i+1}^{D-2} N^j (N + 2)^{D-1-j} + 2D(N + 1)N^{D-1}$
4	$D(N + 1)N^{D-1} + N^D$

**Table 4.** Data transfers per step for the infinite-cache and cubically tiled case.

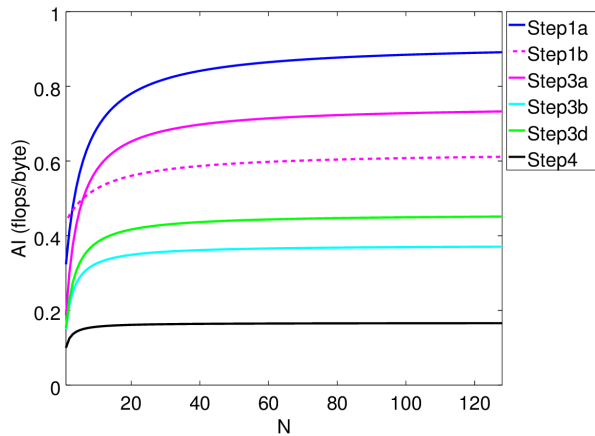
over 6. This is about the machine balance for most current mid-life production HPC machines such as Intel Sandy Bridge based machines or IBM Blue Gene/Q machines. The corresponding arithmetic intensity for the sixth-order method is between 3.5 and 4. While not high enough to reach the machine balance of most machines, it is still a large improvement over low-order methods. The fourth order method remains at an arithmetic intensity of only about 2. For 3D problems, when the box size is of length 128, the sixth-order method achieves an arithmetic intensity of about 10, which is on par with state-of-the-art machines at the time of publication. For the same box size, the eighth-order method has an arithmetic intensity over 20. The arithmetic intensity of the fourth order method reaches about 5. As in the per-step case, the arithmetic intensity can be considerably higher in the asymptotic limit of an infinitely large box. For example, the eighth-order method reaches 38.25 in the limit on three-dimensional problems. However, box sizes need to be much larger than machine memories in order to approach such numbers.

The arithmetic intensities for the idealized case suggest that the methods have potential for improving machine utilization, provided the cache can retain most intermediate data between steps. Of course, the caches on physical

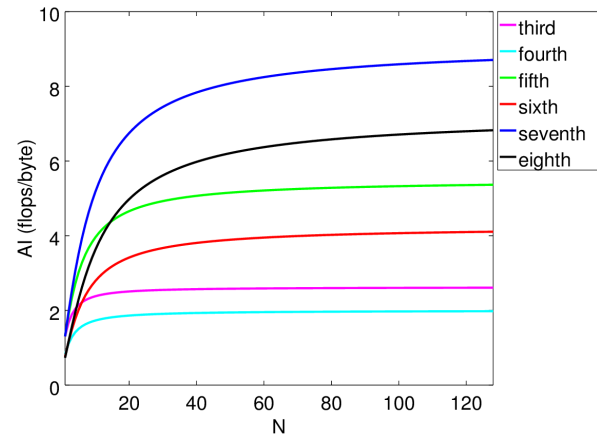
machines do not have infinite size, so a tiling strategy is required to make good use of the cache. We consider the arithmetic intensities under a simple tiling strategy in the next subsection.

#### 4.5 The finite-size cache case with simple tiling

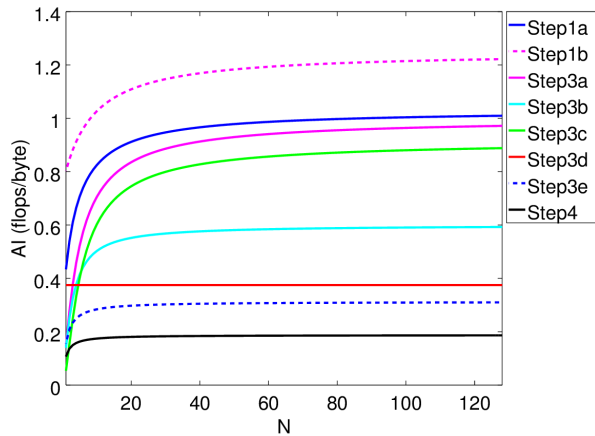
The simplest approach to using a finite size cache is to divide the **box** into cubical tiles. Figure 12 illustrates how a **box** is sub-divided and the data that are kept in the cache in this case. The starting **box** is depicted as the outer box with dashed lines. The **box** is partitioned further into cubical tiles of length  $T$ . Three types of data are used when computing the divergences over a tile. There is a  $T^D$  cube of cells that hold the accumulating divergences, depicted in the figure as the inner cube of the tile. The divergence data must be re-accessed each change of direction  $d$  in order to accumulate the contribution to the divergence from that direction. As such, it is ideally kept in cache to avoid an additional transfer from slow memory each change in direction. An extended cube of  $T + 2w$  in length per side, shown as the outer surrounding cube of the tile, holds the starting cell-average data, including the halo of dependencies of width



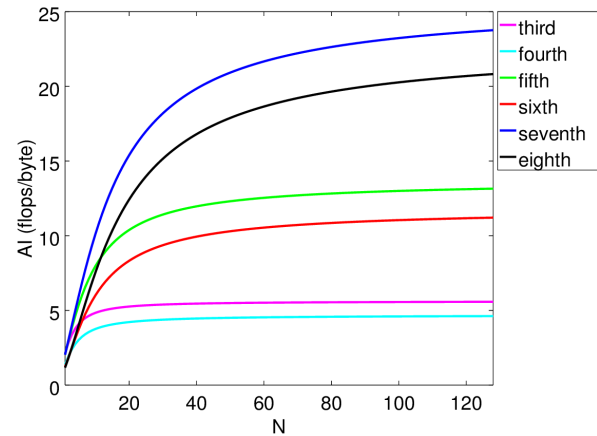
(a) 2D problem



(a) 2D.



(b) 3D problem



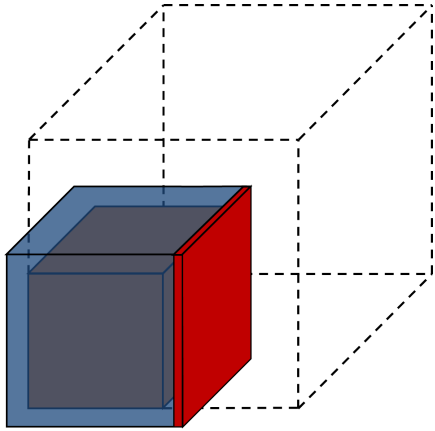
(b) 3D.

**Figure 10.** Arithmetic intensity in the infinite cache case for each step of the eighth-order method as a function of box size. Even for the highest-order method, the per-step arithmetic intensity is low. To fully utilize most current machines, the AI should be at least 5. As a reminder, Steps 3c and 3e only involve mixed partial derivative approximations that contribute in dimensions over 2D.

**Figure 11.** Predicted arithmetic intensities for 2D and 3D problems when using cubical tiles; here  $N$  represents the length per side of the tile. Higher-order methods have significantly increased arithmetic intensity on par with machine balances of current machines (generally between 5 and 10) on 2D problems and expected machine balances of future machines on 3D problems.

$w$ . This data also must be re-accessed for each change of direction, so ideally is kept in cache to avoid multiple transfers from slow memory. Finally, depending on the order of the method and the dimension of the problem, several planes of faces of temporary scratch buffers of size  $(T + 2w)^{D-1}$  must be kept in cache in order to compute the derivative terms for the face-average formula in Step 3. The planes of data are illustrated as the panel on the side of the tile. As the faces are iterated over and as the direction is changed, the same data buffers are moved and re-used.

The number of planes of scratch data that must be kept is determined by the number of levels in Figure 4. One plane of scratch data is needed for each level in a tree, as well as two additional planes to hold the input flux values. (The output flux averages are accumulated directly into the divergence buffer.) For the eighth-order method applied to a 4D problem or larger, a total number of four planes of scratch data are needed. On a 3D problem, only three planes are needed, as the triple mixed-partial term does not exist. The sixth-order method requires three planes on 3D



**Figure 12.** The pattern of data that must be stored in cache to compute the divergences over a cubical tile. The **box** (dashed lines) is further sub-divided into cubical tiles. Within the tile, one cubical region of cell averages with halo data, one cubical region of divergences without halo data, and several **planes** of faces with halo data for intermediate values must be stored at a time in the cache.

problems or higher. The total amount of data that must be kept in cache per tile is then

$$T^D + (T + 2w)^D + k(T + 2w)^{D-1}, \quad (35)$$

where  $k$  is the number of **planes** of scratch data needed.

Subdividing the **box** into tiles in this manner is basically equivalent to subdividing the problem domain into **boxes**, except that data for the halo of dependencies around the **box** are not duplicated through explicit ghost cell exchange, but rather are still taken from the shared array of input values. For now, we will assume that none of the halo data that overlap with neighboring tiles carry over within the cache when moving between tiles. This means the formulas for the flops in Table 2 and data transfers in Table 4 of the infinite-cache case, as well as the arithmetic intensity plots from Figures 10 and 11, carry directly over to this case. The size  $N$  in the formulas is now simply re-interpreted as the length of the tile,  $T$ , where the size of  $T$  is understood to be restricted to allow the tile to fit within cache.

We observed previously that multi-threading would not affect the total number of operations within a box, and that holds for the same reasons within a tile. If tiles are computed concurrently, however, the ordering of when tiles are computed could be affected, and this would change

whether a tile could expect halo data shared with a neighbor to be already resident in the cache or not. Since we are conservatively assuming halo data is never already in the cache, multi-threading over tiles does not change the formulas for the number of operations that must be performed. Multi-threading can change the *rate* (bandwidth and flop rate) at which the operations are performed, but does not affect the *balance* of operations (AI).

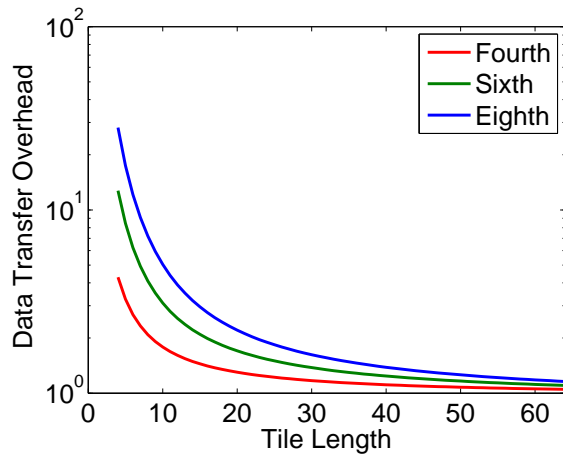
There are two costs to using finite-sized tiles. The first is that the arithmetic intensity is smaller when partitioning into smaller regions than large ones, as seen in Figure 11, so constraints on the size of the tile limit the maximum arithmetic intensity. The other is that the halo data that overlaps with a neighboring tile need to be reloaded when moving to that tile, which increases the total communication volume due to some values being reloaded multiple times.

We define the data transfer overhead of tiling as the increase in data transfers that must be enacted from computing the divergence for a **box** through tiling versus the data transfers for computing without tiling using an infinite size cache. Assuming that the length of each tile divides the **box** length evenly, i.e.,  $N = m \times T$ , the overhead of tiling is

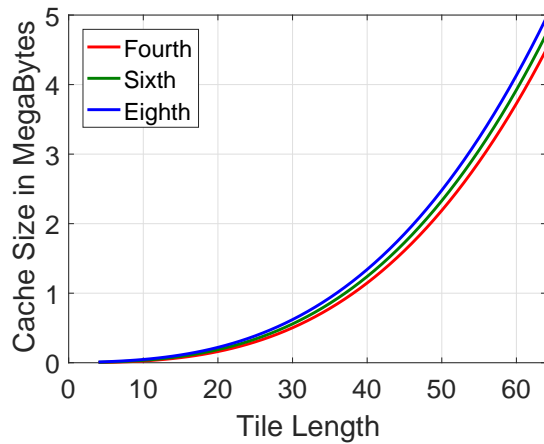
$$\frac{m^D \times (\text{cost per tile})}{(\text{cost per box})}, \quad (36)$$

i.e., the ratio of data transfers needed to compute a **box** worth of tiles versus that for computing the **box** without tiling. The overhead on 3D problems for the even-ordered methods when subdividing a box of size  $N = 128^3$  into sub-tiles is shown in Figure 13(a). The overhead for the odd-ordered methods is similar. We see that the overhead becomes exorbitant if the tile length becomes much smaller than  $T = 32$  in length. In particular, the overhead when  $T = 32$  for the eighth-order method is 1.56 times and that of the sixth-order method is 1.34 times.

For 2D and 3D problems, the cache space requirement per component for cubical tiles of size  $T = 32$  are shown in Table 5. We see that the eighth order methods take up about a megabyte of data for a  $T = 32$  tile. Last-level caches on current high-performance computing grade machines are sized to about 2 to 2.5 MB per core. If we are computing



(a) Data transfer overhead of tiling on a  $128^3$  problem as a function of tile size in a box of size  $128^3$ .



(b) Cache sizes in megabytes required for 3D tiles of length up to 64 on a side.

tiles in a multi-threaded manner, and if we assume one thread per tile per core, the space requirements for a single-component 3D problem fit well into the last level of cache sizes of current machines. However, most equations

Order	Space (MB)	
	2D	3D
Sixth	0.022	0.81
Eighth	0.025	1.01

**Table 5.** Cache space requirements per component for cubical tiling for 2D and 3D problems when  $T = 32$ .

in application problems are multi-component. The Euler equations, for example, have four components in 2D and five components in 3D problems. For 2D problems, the cache requirements are far less than the cache sizes, even

for a four component problem. However, on a 3D problem, the eighth-order method would require 5 MB of cache for cubical tiles of length 32, which is far in excess of current cache sizes [under multi-threading](#). At least in the case of using one thread per core, a more economical caching strategy would be needed to realize the full arithmetic intensity on 3D problems.

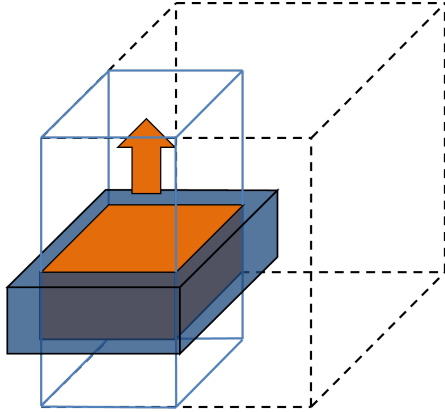
#### 4.6 Lowering cache space requirement on 3D problems

A simple modification to cubical tiling that would conserve cache space would be to use flattened rectangular tiles, for example of size  $32 \times 32 \times 8$ . The reduced volume would take up less space in cache. The trade-off would be that the surface-to-volume ratio of the tile would be increased, resulting in higher overhead for re-loading of the halo data. This problem can be mitigated by moving from tile to tile in a vertically iterated manner. Figure 14 illustrates this idea. For every tile-wide column in the box, the first tile is computed at the bottom of the column. When a tile in the column is computed, the next tile is always the one immediately above it. This repeats until the column is completely computed and then a new column is chosen. [In a multi-threading situation, threads would be assigned to columns. We conservatively assume neighboring columns do not share halo data through the cache.](#) By iterating vertically in this manner, the halo data at the bottom of a tile are still in cache from the previous tile. Part of the non-halo data will be as well, but for simplicity we will assume that data is re-fetched.

For a 3D problem with tile sizes of  $T \times T \times H$ , where  $H$  is the height of the tile, the amount of cache space taken up becomes

$$(H + 2w)(T + 2w)^2 + HT^2 + 4(T + 2w)^2. \quad (37)$$

For tiles of size  $32 \times 32 \times 8$ , the space requirement for the eighth-order method moves from 5 MB to 2.1 MB. The space requirement for the sixth-order method falls to 1.8 MB. Since the halo does not need to be reloaded in the vertical direction, the overhead is equivalent to that of loading an entire column at once, which for tiles  $32$  by  $32$



**Figure 14.** Vertical flattened tile iteration

wide is 1.21 times for the sixth-order method and 1.33 times for the eighth-order method.

## 5 Experimental validation

To verify the arithmetic intensity formulas, the methods from orders three through eight were implemented in C++, and hardware performance counters on a node of an IBM Blue Gene/Q (BG/Q) system were used to measure the flops and data transfers. The counters were read through the BG/Q hardware performance monitoring API (BGPM)<sup>49</sup>, which provides access to the counters from user-level processes.

Only the steps of the method as described in Section 3.6 were tested. Each measurement was for a single evaluation of the right hand side of formula (6), and no time integration was performed.

### 5.1 Relevant hardware features

A node on a BG/Q system has sixteen 64-bit cores. The last level of cache is 32 MB of L2 cache shared amongst all 16 cores. The L2 is 16-way set-associative, write-back, and has a least-recently-used replacement policy. The cache lines are 128 bytes long, so all data transfers to and from DRAM are counted in 128 byte chunks.

The BG/Q flops counters allow fine-grained filtering of the different types of floating point instructions. For example, floating point register to floating point register

moves can be excluded while floating point additions can be included in the measurements. Vectorized instructions can be counted as single instructions or weighted by the vector length. We configured the API to count floating point operations in accordance with the machine model from Section 4.1. We included all arithmetic instructions and excluded all non-arithmetic instructions such as floating-point register moves. FMA scalar instructions were counted as two floating point operations. Vector instructions on the BG/Q system are four values in length, so non-FMA instructions were counted as four operations and vector FMA instructions as eight operations. The API incurs no flops costs of its own, so there is no API overhead in using BGPM to measure the flops costs of an application. As such, the flops measurements are completely without measurement error.

It is common on write-back caches for a write operation to a cache line not already present in the cache to first induce a load of the line from DRAM. ~~The BG/Q L2 cache also follows this approach, except under particular conditions.~~ In personal communication, Dr. M. Ohmacht from IBM explained that the BG/Q L2 cache avoids loading a line on a write except in particular non-aligned cases. Specifically, the cache does not use a single dirty bit for an entire line, but rather has a dirty bit per eight bytes of the line. When storing eight byte values (such as double-precision floating-point numbers) in an eight-byte aligned manner, the cache does not require a load of the line, although the line is allocated into the cache. Further testing shows that if the entire line is fully dirty before a load, i.e., if the line has been fully filled by writes, subsequent loads from the line do not invoke a load from DRAM. ~~This allows the additional loads to be optimized away under streaming write conditions.~~ By being careful to take advantage of this feature in our tests, we can use the counters to effectively differentiate between the load and store operations of the algorithm. Under these conditions, store operations do not increment the load counter and the counter value, therefore, only measures the load operations induced by the algorithm itself.

## 5.2 Measurement methodology

The counters measure the loads and stores from DRAM independently. All transfers are done in units of 128-byte cache lines. While the counters measure DRAM transfer requests directly at the memory, correlating counter events to load and store requests in the application itself is not entirely straightforward. Load events are the more simple of the two cases. When an application’s instruction stream executes a load instruction for data not found within cache, a request for the cache line containing that data is made to DRAM, and the counter is incremented. Therefore, the counter increments in direct response to a load request from the CPU. However, when a store instruction is executed by the CPU, the data are sent to the cache but not directly to DRAM, as the cache is write-back. Store requests are made to DRAM only when a previously written (“dirty”) data value is evicted from the cache and sent back to main memory to make room for the newly stored data. Furthermore, loads of data not within cache will also evict a line from the cache and induce a DRAM store if the line is dirty.

To deal with this, before running any experiments, a long array of at least the length of the cache is written, so that every line in the cache is marked dirty. Afterwards, the experiment is begun. Since the replacement policy is least-recently-used, any subsequent load or store instructions for data not in cache will cause older dirty data to be flushed back to memory, invoking an increment of the store counter. The total number of loads to non-cached data by the application is then equal to that given by the load counter, whereas the number of non-cached stores directly caused by the application equals the number of stores minus the number of loads. In an actual application, over time the number of actual DRAM loads and stores should reach a “steady state” equal to the non-cached loads and stores induced by the processor. Naturally this approach only works if the capacity of the cache is not exceeded. The L2 cache is 32 MB in length, which is ordinarily meant to be shared among many threads, but we only use a single thread during our experiments. The total amount of memory used is a little over 5 MB in the case of cubical [tiling](#) in the three-dimensional case, so consuming the pool of dirty data was

not an issue. Furthermore, the working set size was small enough that the set-associativity of the cache also did not result in a significant extra number of stores.

Recall from Section 4.5 that there are three types of data buffers used to perform the step: the input buffer of cell averages with a halo, the output buffer of divergences without a halo, and the slices of face data used for intermediate calculations. At the start of the step, the data in the input and output buffers are assumed to reside outside the cache. However, the scratch data buffers are assumed to be re-used from [tile to tile](#), and therefore the data remains within the cache over the run except for the first touch. To simulate this, at the beginning of the test, the cache is flushed with dirty data, and the scratch buffers are then touched to place them in the cache.

To maximize measurement accuracy, all data was allocated on cache line boundaries, and the prefetcher was disabled. Nevertheless, the API itself incurs an overhead of about five to ten cache lines, even if the API is started and immediately stopped with no work in between. This overhead remained constant regardless of problem size and is a fixed overhead of the measurements.

[We tried measuring the operations when using OpenMP. The flops counts were unaffected, but invoking the OpenMP runtime caused a significantly sized constant size overhead in the number of data transfers. The size of the overhead remained invariant to the number of threads, but was large compared to the amount of data motion for small problems and had a non-trivial variance. For accuracy, we limited measurements to only a single thread running over a single box or tile. This is in line with our conservative assumption that neighboring tiles do not share halo data, even if computed concurrently or subsequently from one another.](#)

[For each combination of problem dimension and tile size, we measured the counters 100 times. There was exactly zero variation in reported flops between trials for the flops counters. There was some variation for the DRAM counters, with the following pattern. Most counts were very close to each other, but occasionally the counts would spike to far larger numbers. The increase was always at least several times larger than the typical value, and usually well](#)

over an order of magnitude larger. We interpret those cases as the process being context switched out for some other process that pollutes the cache. In our reporting, we have filtered such cases out. Of the remaining trials, the ratio of variance to mean was always very low. In the smallest case of a 2D problem with a tile length of 4, the ratio of variance to mean and the ratio of variance to minimum were both about 0.014. In that case, the systematic overhead of about five to ten cache lines drowned out the variance, so the discrepancy between the predicted bytes and measured bytes was only negligibly due to variation between trials. For 3D problem with larger tile sizes, the ratio of variance to mean fell below  $1e-4$ . Because we are using counters that tally system events in a single-threaded implementation, we consider the *minimum* count, not the mean, to be the correct value to report. As such, the values we report are the minimum over 100 trials after filtering away the exceptional cases. However, because the variation across the remaining trials is so low, the actual difference between the minimum and mean is negligible.

We have not found the measured number of flops or amount of transferred data to be to be chanced by either the choice of compiler or the optimization level of the compilation. We have tested with both version 4.4.7 of the GNU C++ compiler and version 12.1 of the IBM XL C++ compiler. Partially this is because we took care to group terms manually in the implementation to minimize the number of operations. When compiling with `-O3` versus `-O0` optimization flags, we did find significant restructuring of the loops and a high use of FMA operations when examining the executable using the *objdump* utility. However, since FMA and vector instructions are weighted to give the same number of operations as if they were done with non-FMA scalar instructions, the final flops count becomes the same in all cases.

### 5.3 Results

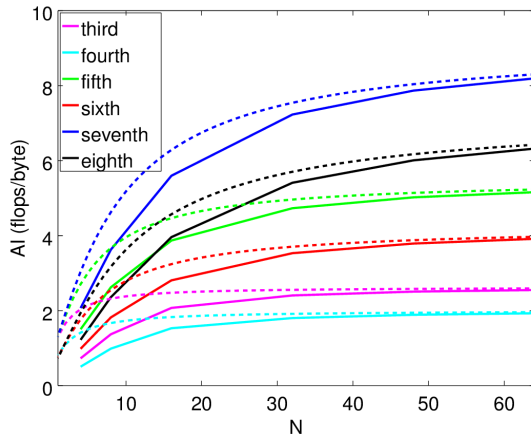
The predictions from Figure 11 were verified using a two-dimensional and three-dimensional single-component test problem. A single thread was used to measure the arithmetic intensities for a single *tile*. The sizes of the *tiles*

were 4, 16, 32, and 64 in length on a side. Since the cost of the flux function is ignored in the predictions, we used a flux function that made no changes to the inputs (a no-op function). Recall that the flux function is assumed to operate on the input data in-place, so there is no additional data transfer cost for the flux function if the data is read from the cache. For the odd-ordered methods, a no-op function was similarly used for the Riemann solver. The data was initialized with random numbers. Since there is no flux function or Riemann solver, the cost of the method is wholly independent of the content of the data. The results are shown in Figure 15.

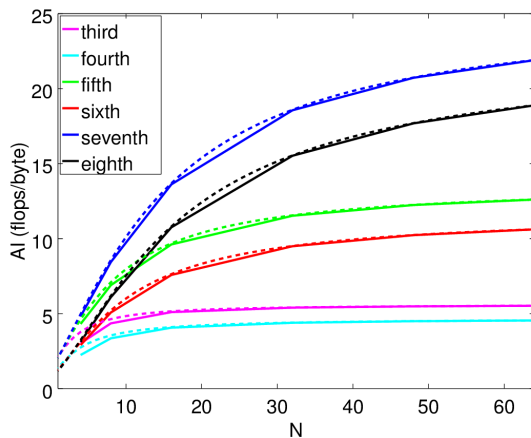
The measured flops costs agree with the model exactly. The measured number of bytes moved is always greater than predicted by a fixed amount of five to ten cache lines. As a result, the measured arithmetic intensity is always lower than the predicted arithmetic intensity, which is most evident in 2D at small *tile* sizes. For three-dimensional problems, even when the length per side is small, the predictions agree closely with the model. For two-dimensional problems, the agreement is close when the problem size is of length 32 or greater, but deviates significantly for smaller size problems. This is because the amount of data moved on small problems is modest enough that the five to ten cache line overhead becomes of size similar to the predicted total data movement. For three-dimensional problems, the data movement remains large compared to the overhead, even when the length per-side is small.

For the case of using flattened *tiles* iterated in the vertical direction (Section 4.5), the arithmetic intensity was measured for a single vertical column of height 128, for just the case of tiles with width of 32 and height of 8. This was the configuration deemed to give the best balance between cache storage cost versus the data transfer overhead of the halo. The measured results are shown in Table 6. For comparison, the arithmetic intensity of the cubical case with length 32 is also shown.

The arithmetic intensity of the vertically iterated case is higher than that of the cubical case. This is because intermediate halo flux averages need to be re-computed from *tile to tile*, increasing the overall work. The *number*



(a) 2D.



(b) 3D.

**Figure 15.** Measured versus predicted arithmetic intensity for 2D and 3D problems on cubical sub-tiles. The discrepancy between predicted and measured arithmetic intensity is accounted for by the API overhead of measuring the data transfers. The measurements were taken only at 4, 8, 16, 32, 48, and 64 size tiles, whereas there is a predicted intensity for all sizes between 4 and 64.

Order	Flops	Bytes	AI	Cache cost
6 (vertical)	3.5e7	2.8e6	12.6	1.8 MB
6 (cubical)	2.9e7	3.1e6	9.5	3.2 MB
8 (vertical)	7.2e6	3.3e6	22.4	2.1 MB
8 (cubical)	5.9e6	3.8e6	15.5	5.1 MB

**Table 6.** The difference in costs of the cubical versus vertically iterated caching strategy when computing a region 128 cells tall. 16 tiles of size  $32 \times 32 \times 8$  are computed successively in the z-direction. For comparison, the arithmetic intensity of the cubical strategy for a  $32 \times 32 \times 32$  tile is also listed. Note that using flattened tiles in tall columns increases the intensity relative to the cubical case.

of bytes transferred is also reduced in the vertically iterated case compared to the cubical case, due to the halo data along the z-direction not needing to be re-loaded from tile to tile.

Since we only tested the arithmetic intensity using a single thread for a single vertical column, we did not test whether the vertically iterated strategy preserved the predicted arithmetic intensity when the cache was shared amongst 16 threads leaving only 2 MB per thread. Of course, the working set size is known because the allocated size of the data buffers is known, so the reduction in required memory space is visible.

## 6 Conclusions and future work

Many algorithms currently used in high-performance scientific computing fail to reach high utilization of the full machine. While parallel scalability is typically the focus of algorithm design, attention should also be given to on-node performance. With data motion increasingly becoming the dominant cost, algorithms that fail to make efficient use of every byte transferred will likely be bandwidth limited. The algorithmic intensity of a method should therefore be an important design criteria.

In this paper, we have demonstrated that higher-order finite volume discretizations for hyperbolic systems of conservation laws will have advantages over the lower-order counterparts commonly used today. Through analysis and measurements on an IBM BG/Q node, we have shown that the increase in arithmetic operations will be more than offset by the reduction in data transfers and the recovery of wasted compute cycles. We considered spatial discretizations from orders three through eight in arbitrary dimensions, and we developed our estimates for three cache models: the limiting cases of no and infinite cache as well as a finite-sized cache model. Both theory and experiments demonstrated that high-order finite volume methods will be able to provide increases in arithmetic intensity that will be able to better utilize on-node floating-point capability.

In this work, we have not addressed nonlinear algorithms necessary for capturing discontinuous solutions, for

example, solution-limited reconstructions or flux limiting. There are several techniques that should be compatible with the discretizations we consider, but limiters are local and only increase the amount of operations on data already loaded into cache; thus our estimates are conservative, and nonlinear schemes would not require as high an order to reach machine balance. We leave detailed analysis in this case for future work.

Another challenge associated with high-order finite volume schemes (and finite difference schemes as well) is that these methods reconstruct information by using neighboring data. In contrast, finite element and spectral element schemes increase the number of degrees of freedom inside each element. With the increase in order, the size of the ghost cell region used with a finite volume/difference scheme therefore increases. In high dimensions, the number of ghost cells needed becomes a large percentage of the number of cells within a `box`, resulting in an increase in data transfers. While we have included such costs in our analysis, there may be additional advantage to finding more efficient means to transfer this ghost data, for example, by using representations that effectively or literally compress the ghost data exchanged.

Finally, there is a gap between that which is theoretically possible and that which is practical. In this work, we have demonstrated that high algorithmic intensity can be obtained, but this required the assumption of several cache models and hand-optimized code fused across functions that are typically separated. In addition, obtaining high algorithmic intensity is only part of the story; it is a necessary but insufficient condition to achieving maximal performance. In addition, one must make many machine-dependent optimizations that leverage all of the capabilities of a given architecture, such as fused multiple-add instructions and advanced vector extensions, to obtain a high number of floating-point operations. Often, code that is written to improve modularity and maintainability will not provide either high algorithmic intensity or floating-point performance. It remains an open question: how to balance the need for rational software development of maintainable software while providing implementations that can access the full capability of the hardware?

## Acknowledgements

The authors wish to acknowledge the many helpful discussions with P. Colella, B. van Straalen, A. Myers, and A. Dubey. The authors also wish to acknowledge M. Ohmacht, S. Parker, V. Morozov, and J. Hammond for their invaluable help in understanding the L2 cache on IBM BG/Q machines.

## Funding

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program. [LLNL-JRNL-681075](#).

## Declaration of conflicting interests

None declared.

## References

1. Bohr M. A 30 year retrospective on Dennard's MOSFET scaling paper. *Solid-State Circ Soc Newslet, IEEE* 2007; 12(1): 11–13.
2. The ASCAC Subcommittee on Exascale Computing. The opportunities and challenges of exascale computing. Technical report, U.S. Department of Energy Advanced Scientific Computing Advisory Committee, 2010.
3. ASCAC Subcommittee for the Top Ten Exascale Research Challenges. Top ten exascale research challenges. Technical report, U.S. Department of Energy Advanced Scientific Computing Advisory Committee, 2014.
4. Exascale Mathematics Working Group. Applied mathematics research for exascale computing. Technical report, U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, 2014.
5. Kreiss HO and Olinger J. Comparison of accurate methods for the integration of hyperbolic equations. *Tellus* 1972; 24(3): 199–215.
6. Colella P, Dorr MR, Hittinger JAF et al. High-order, finite-volume methods in mapped coordinates. *J Comput Phys* 2011; 230(8): 2952–2976.

7. Williams S, Waterman A and Patterson D. Roofline: An insightful visual performance model for multicore architectures. *Commun ACM* 2009; 52(4): 65–76.
8. McCorquodale P and Colella P. A high-order finite volume method for conservation laws on locally refined grids. *Comm App Math and Compu Sci* 2011; 6(1): 1–25.
9. McCorquodale P, Dorr MR, Hittinger JAF et al. High-order finite-volume methods for hyperbolic conservation laws on mapped multiblock grids. *J Comput Phys* 2015; 288: 181–195.
10. Basu P, Hall M, Williams S et al. Compiler-directed transformation for higher-order stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. pp. 313–323.
11. Williams S, Carter J, Oliker L et al. Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms. *J Parallel Distrib Comput* 2009; 69(9): 762–777.
12. Williams S, Oliker L, Carter J et al. Extracting ultra-scale lattice boltzmann performance via hierarchical and distributed auto-tuning. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11, pp. 55:1–55:12.
13. Idomura Y and Jolliet S. Performance evaluations of gyrokinetic eulerian code gt5d on massively parallel multicore platforms. In *State of the Practice Reports*. SC '11, pp. 4:1–4:9.
14. Godenschwager C, Schornbaum F, Bauer M et al. A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13, pp. 35:1–35:12.
15. Pananilath I, Acharya A, Vasista V et al. An optimizing code generator for a class of lattice-boltzmann computations. *ACM Trans Archit Code Optim* 2015; 12(2): 14:1–14:23.
16. Rossinelli D, Hejazialhosseini B, Spampinato DG et al. Multicore/multi-gpu accelerated simulations of multiphase compressible flows using wavelet adapted grids. *SIAM J Sci Comput* 2011; 33(2): 512–540.
17. Rossinelli D, Hejazialhosseini B, Hadjidoukas P et al. 11 pflop/s simulations of cloud cavitation collapse. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13, pp. 3:1–3:13.
18. Hejazialhosseini B, Rossinelli D, Conti C et al. High throughput software for direct numerical simulations of compressible two-phase flows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '12, pp. 1–12.
19. Bermejo-Moreno I, Bodart J, Larsson J et al. Solving the compressible navier-stokes equations on up to 1.97 million cores and 4.1 trillion grid points. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13, pp. 62:1–62:10.
20. Jiang GS and Shu CW. Efficient implementation of weighted eno schemes. *J Comput Phys* 1996; 126(1): 202–228.
21. Olschanowsky C, Strout MM, Guzik S et al. A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14, pp. 793–804.
22. Callahan D, Cocke J and Kennedy K. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing* 1988; 5(4): 334 – 358.
23. Carr S and Kennedy K. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans Program Lang Syst* 1994; 16(6): 1768–1810.
24. Sim J, Dasgupta A, Kim H et al. A performance analysis framework for identifying potential benefits in gpgpu applications. *SIGPLAN Not* 2012; 47(8): 11–22.
25. Langguth J, Wu N, Chai J et al. On the GPU performance of cell-centered finite volume method over unstructured tetrahedral meshes. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*. pp. 7:1–7:8.
26. Zhang W, Wei W and Cai X. Performance modeling of serial and parallel implementations of the fractional adams-bashforth-moulton method. *Frac Calc App Anal* 2014; 17(3): 617–637.
27. Stengel H, Treibig J, Hager G et al. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15, pp. 207–216.
28. Jia-Wei H and Kung HT. I/O complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM*

*Symposium on Theory of Computing*. STOC '81, pp. 326–333.

29. Ballard G, Carson E, Demmel J et al. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica* 2014; 23: 1–155.
30. Bhatel e A, Wesolowski L, Bohm E et al. Understanding application performance via micro-benchmarks on three large supercomputers: Intrepid, Ranger and Jaguar. *Int J High Perform Comput Appl* 2010; 24(4): 411–427.
31. Yokota R and Barba LA. Hierarchical n-body simulations with autotuning for heterogeneous systems. *Comput Sci Engin* 2012; 14(3): 30–39.
32. Chen G, Chac on L and Barnes DC. An efficient mixed-precision, hybrid CPU-GPU implementation of a nonlinearly implicit one-dimensional particle-in-cell algorithm. *J Comput Phys* 2012; 231(16): 5374–5388.
33. McCorquodale P and Colella P. A high-order finite-volume method for hyperbolic conservation laws on locally-refined grids. *Comm App Math Comput Sci* 2011; 6(1): 1–25.
34. Colella P and Woodward PR. The piecewise parabolic method (PPM) for gas-dynamical simulations. *J Comput Phys* 1984; 54(1): 174–201.
35. Chaplin C and Colella P. A single stage flux-corrected transport algorithm for high-order finite-volume methods. *pre-print* 2015; .
36. Gottlieb S, Shu CW and Tadmor E. Strong stability-preserving high-order time discretization methods. *SIAM Review* 2001; 43(1): 89–112.
37. Liu XD, Osher S and Chan T. Weighted essentially non-oscillatory schemes. *J Comput Phys* 1994; 115(1): 200–212.
38. Jiang GS and Shu CW. Efficient implementation of weighted ENO schemes. *J Comput Phys* 1996; 126(1): 202–228.
39. Shu CW. Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. Technical Report ICASE-97-65, NASA, 1997.
40. Boris JP and Book DL. Flux-Corrected Transport. I. SHASTA, a fluid transport algorithm that works. *J Comput Phys* 1973; 11: 38–69.
41. Zalesak ST. Fully multidimensional Flux-Corrected Transport algorithms for fluids. *J Comput Phys* 1979; 31(2): 335–362.
42. Adams M, Colella P, Graves D et al. Chombo software package for AMR applications - design document.

Step	Flops
1a-fourth	$5D(N+1)(N+4)^{D-1}$
1a-third	$(10 + f_R)D(N+1)(N+4)^{D-1}$
1b	$4D(N+1) \sum_{i=1}^{D-1} (N+2)^i (N+4)^{D-1-i}$
2	$f_F \times D(N+1)(N+2)^{D-1}$
3a	$5D(D-1)(N+1)N^{D-1}$
4	$2DN^D$

**Table 7.** Flops per step for third- and fourth-order methods. Valid for the no-cache, infinite-cache and cubically tiled case.

- Technical Report LBNL-6616E, Lawrence Berkeley National Laboratory, 2014.
43. Bell J, Almgren A, Beckner V et al. *BoxLib User's Guide*. Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory, 2013.
  44. Hornung RD and Kohn SR. Managing application complexity in the samrai object-oriented framework. *Concurr Comput* 2002; 14: 347–368.
  45. Henshaw W. Overture: An object-oriented toolkit for solving partial differential equations in complex geometry. webpage, 2015. URL [www.overtureframework.org](http://www.overtureframework.org).
  46. Clawpack Development Team. Clawpack software, 2014. URL [www.clawpack.org](http://www.clawpack.org).
  47. Deiterding R. AMROC: Adaptive Mesh Refinement in Object-oriented C++. webpage, 2016. URL [www.vtf.website/asc/wiki/bin/view/Amroc](http://www.vtf.website/asc/wiki/bin/view/Amroc).
  48. Golub G and Van Loan C. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, 2012.
  49. International Business Machines Corporation. *BGPM - BG/Q Hardware Perf Monitoring API*, 2012.

## Appendix

Step	Transfers
1a-fourth	$5D(N+1)(N+6)^{(D-1)}$
1a-third	$5D(N+1)(N+6)^{(D-1)}$
1b	$4D(N+1) \sum_{i=1}^{D-1} (N+2)^i (N+4)^{D-1-i}$
2	$2D(N+1)(N+2)^{D-1}$
3a	$D(6D-7)(N+1)N^{D-1}$
4	$(2D+1)N^D$

**Table 8.** Data transfers per step for third- and fourth-order methods in the no-cache case.

Step	Transfers
1a-fourth	$(N+4)^D + D(N+1)(N+4)^{D-1}$
1a-third	$(N+4)^D + D(N+1)(N+4)^{D-1}$
1b	$D(N+1)(N+4)^{D-1} + D(N+1)(N+2)^{D-1}$
2	$2D(N+1)(N+2)^{D-1}$
3a	$D(N+1)(N+2)^{D-1} + D(N+1)N^{D-1}$
4	$D(N+1)N^{D-1} + N^D$

**Table 9.** Data transfers per step for the third- and fourth-order methods for the infinite-cache and cubically tiled case.

Step	Flops
1a-sixth	$8D(N+1)(N+8)^{D-1}$
1a-fifth	$(18 + f_R)D(N+1)(N+8)^{D-1}$
1b	$7D(N+1) \sum_{i=1}^{D-1} (N+4)^i (N+8)^{D-1-i}$
2	$f_F \times D(N+1)(N+4)^{D-1}$
3a	$D(N+1) \left[ 7 \sum_{i=1}^{D-1} N^i (N+2)^{D-1-i} + 2(D-1)N^{D-1} \right]$
3b	$8D(D-1)(N+1)N^{D-1}$
3c	$5D \binom{D-1}{2} (N+1)N^{D-1}$
4	$2DN^D$

**Table 10.** Flops per step for fifth- and sixth-order methods. Valid for the no-cache, infinite-cache and cubically tiled case.

Step	Transfers
1a-sixth	$7D(N+1)(N+8)^{(D-1)}$
1a-fifth	$7D(N+1)(N+8)^{(D-1)}$
1b	$6D(N+1) \sum_{i=1}^{D-1} (N+4)^i (N+8)^{D-1-i}$
2	$2D(N+1)(N+4)^{D-1}$
3a	$D(N+1) \left[ 5 \sum_{i=1}^{D-1} N^i (N+1)^{D-1-i} + \sum_{i=1}^{D-2} N^i (N+1)^{D-1-i} + (2D-3)N^{D-1} \right]$
3b	$7D(D-1)(N+1)N^{D-1}$
3c	$5D \binom{D-1}{2} (N+1)N^{D-1}$
4	$(2D+1)N^D$

**Table 11.** Data transfers per step for fifth- and sixth-order methods in the no-cache case.

Step	Transfers
1a-sixth	$(N+8)^D + D(N+1)(N+8)^{D-1}$
1a-fifth	$(N+8)^D + D(N+1)(N+8)^{D-1}$
1b	$D(N+1)(N+8)^{D-1} + D(N+1)(N+4)^{D-1}$
2	$2D(N+1)(N+4)^{D-1}$
3a	$D(N+1)(N+4)^{D-1} + D(N+1) \sum_{i=1}^{D-2} N^i (N+2)^{D-1-i} + D(N+1)N^{D-1}$
3b	$D(N+1)(N+4)^{D-1} + 2D(N+1)N^{D-1}$
3c	$D(N+1) \sum_{i=1}^{D-2} N^i (N+2)^{D-1-i} + 2D(N+1)N^{D-1}$
4	$D(N+1)N^{D-1} + N^D$

**Table 12.** Data transfers per step for fifth- and sixth-order methods for the infinite-cache and cubically tiled case.

	1-norm	2-norm	inf-norm
N=32	5.63e-01	6.94e-01	1.43e+00
ratio	6.24	6.21	6.29
rate	2.64	2.63	2.65
N=64	9.02e-02	1.12e-01	2.27e-01
ratio	7.46	7.46	7.54
rate	2.90	2.90	2.91
N=128	1.21e-02	1.50e-02	3.01e-02
ratio	7.83	7.84	7.88
rate	2.97	2.97	2.98
N=256	1.55e-03	1.91e-03	3.82e-03
ratio	7.94	7.95	7.97
rate	2.99	2.99	2.99
N=512	1.95e-04	2.40e-04	4.79e-04
ratio	7.98	7.98	7.99
rate	3.00	3.00	3.00
N=1024	2.44e-05	3.01e-05	6.00e-05

**Table 13.** Convergence of the 3rd-order method when computing the flux divergence for one step of the 2D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	2.37e-01	2.63e-01	3.72e-01
ratio	12.6	12.6	12.6
rate	3.65	3.66	3.66
N=64	1.88e-02	2.09e-02	2.95e-02
ratio	15.1	15.1	15.1
rate	3.91	3.92	3.92
N=128	1.25e-03	1.38e-03	1.96e-03
ratio	15.8	15.8	15.8
rate	3.98	3.98	3.98
N=256	7.90e-05	8.77e-05	1.24e-04
ratio	15.9	15.9	15.9
rate	3.99	3.99	3.99
N=512	4.95e-06	5.50e-06	7.78e-06
ratio	16.0	16.0	16.0
rate	4.00	4.00	4.00
N=1024	3.10e-07	3.44e-07	4.87e-07

**Table 14.** Convergence of the 4th-order method when computing the flux divergence for one step of the 2D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	1.53e-01	1.88e-01	3.87e-01
ratio	22.9	22.8	23.1
rate	4.52	4.51	4.53
N=64	6.67e-03	8.26e-03	1.68e-02
ratio	29.2	29.2	29.5
rate	4.87	4.87	4.88
N=128	2.28e-04	2.83e-04	5.69e-04
ratio	31.1	31.2	31.4
rate	4.96	4.96	4.97
N=256	7.34e-06	9.07e-06	1.81e-05
ratio	31.7	31.8	31.8
rate	4.99	4.99	4.99
N=512	2.31e-07	2.86e-07	5.70e-07
ratio	31.9	31.9	32.0
rate	5.00	5.00	5.00
N=1024	7.25e-09	8.95e-09	1.78e-08

**Table 15.** Convergence of the 5th-order method when computing the flux divergence for one step of the 2D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	6.86e-02	7.62e-02	1.08e-01
ratio	46.4	46.4	46.4
rate	5.54	5.54	5.54
N=64	1.48e-03	1.64e-03	2.32e-03
ratio	59.1	59.1	59.1
rate	5.88	5.89	5.89
N=128	2.50e-05	2.78e-05	3.93e-05
ratio	62.7	62.7	62.7
rate	5.97	5.97	5.97
N=256	3.99e-07	4.43e-07	6.26e-07
ratio	63.7	63.7	63.7
rate	5.99	5.99	5.99
N=512	6.26e-09	6.96e-09	9.84e-09
ratio	63.9	63.9	63.9
rate	6.00	6.00	6.00
N=1024	9.80e-11	1.09e-10	1.54e-10

**Table 16.** Convergence of the 6th-order method when computing the flux divergence for one step of the 2D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	4.42e-02	5.44e-02	1.12e-01
ratio	83.9	83.3	84.2
rate	6.39	6.38	6.40
N=64	5.28e-04	6.53e-04	1.33e-03
ratio	114	114	115
rate	6.83	6.83	6.85
N=128	4.62e-06	5.72e-06	1.15e-05
ratio	124	124	125
rate	6.95	6.95	6.96
N=256	3.73e-08	4.61e-08	9.23e-08
ratio	127	127	127
rate	6.98	6.99	6.99
N=512	2.95e-10	3.64e-10	7.26e-10
ratio	128	128	128
rate	6.99	7.00	7.00
N=1024	2.31e-12	2.85e-12	5.68e-12

**Table 17.** Convergence of the 7th-order method when computing the flux divergence for one step of the 2D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	2.05e-02	2.27e-02	3.22e-02
ratio	170	170	170
rate	7.41	7.41	7.41
N=64	1.21e-04	1.34e-04	1.89e-04
ratio	231	231	231
rate	7.85	7.85	7.85
N=128	5.21e-07	5.79e-07	8.19e-07
ratio	250	250	250
rate	7.96	7.96	7.96
N=256	2.09e-09	2.32e-09	3.28e-09
ratio	254	254	254
rate	7.99	7.99	7.99
N=512	8.21e-12	9.12e-12	1.29e-11
ratio	255	255	255
rate	7.99	7.99	7.99
N=1024	3.22e-14	3.58e-14	5.06e-14

**Table 18.** Convergence of the 8th-order method when computing the flux divergence for one step of the 2D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	4.87e-01	6.67e-01	1.97e+00
ratio	5.82	5.77	5.90
rate	2.54	2.53	2.56
N=64	8.36e-02	1.16e-01	3.34e-01
ratio	7.30	7.32	7.43
rate	2.87	2.87	2.89
N=128	1.15e-02	1.58e-02	4.50e-02
ratio	7.78	7.80	7.86
rate	2.96	2.96	2.97
N=256	1.47e-03	2.02e-03	5.73e-03
ratio	7.92	7.93	7.96
rate	2.99	2.99	2.99
N=512	1.86e-04	2.55e-04	7.19e-04
ratio	7.97	7.98	7.99
rate	2.99	3.00	3.00
N=1024	2.34e-05	3.20e-05	9.00e-05

**Table 19.** Convergence of the 3rd-order method when computing the flux divergence for one step of the 3D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	2.03e-01	2.38e-01	4.54e-01
ratio	11.7	11.7	11.8
rate	3.55	3.55	3.56
N=64	1.73e-02	2.03e-02	3.84e-02
ratio	14.8	14.8	14.9
rate	3.89	3.89	3.89
N=128	1.17e-03	1.37e-03	2.58e-03
ratio	15.7	15.7	15.7
rate	3.97	3.97	3.97
N=256	7.47e-05	8.72e-05	1.64e-04
ratio	15.9	15.9	15.9
rate	3.99	3.99	3.99
N=512	4.70e-06	5.48e-06	1.03e-05
ratio	16.0	16.0	16.0
rate	4.00	4.00	4.00
N=1024	2.94e-07	3.43e-07	6.45e-07

**Table 20.** Convergence of the 4th-order method when computing the flux divergence for one step of the 3D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	1.32e-01	1.80e-01	5.34e-01
ratio	21.4	21.2	21.6
rate	4.42	4.40	4.44
N=64	6.17e-03	8.53e-03	2.47e-02
ratio	28.5	28.6	29.0
rate	4.83	4.84	4.86
N=128	2.16e-04	2.98e-04	8.49e-04
ratio	30.9	31.0	31.2
rate	4.95	4.95	4.97
N=256	7.00e-06	9.61e-06	2.72e-05
ratio	31.6	31.7	31.8
rate	4.98	4.99	4.99
N=512	2.21e-07	3.03e-07	8.55e-07
ratio	31.9	31.9	32.0
rate	4.99	5.00	5.00
N=1024	6.94e-09	9.51e-09	2.67e-08

**Table 21.** Convergence of the 5th-order method when computing the flux divergence for one step of the 3D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	5.49e-02	6.45e-02	1.23e-01
ratio	43.7	43.8	44.1
rate	5.45	5.45	5.46
N=64	1.26e-03	1.47e-03	2.79e-03
ratio	58.1	58.2	58.4
rate	5.86	5.86	5.87
N=128	2.17e-05	2.53e-05	4.77e-05
ratio	62.4	62.4	62.6
rate	5.96	5.96	5.97
N=256	3.47e-07	4.05e-07	7.63e-07
ratio	63.6	63.6	63.6
rate	5.99	5.99	5.99
N=512	5.46e-09	6.37e-09	1.20e-08
ratio	63.9	63.9	63.9
rate	6.00	6.00	6.00
N=1024	8.55e-11	9.98e-11	1.88e-10

**Table 22.** Convergence of the 6th-order method when computing the flux divergence for one step of the 3D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	3.80e-02	5.20e-02	1.54e-01
ratio	77.9	77.2	78.9
rate	6.28	6.27	6.30
N=64	4.88e-04	6.74e-04	1.95e-03
ratio	111	112	114
rate	6.80	6.81	6.83
N=128	4.38e-06	6.03e-06	1.72e-05
ratio	123	123	124
rate	6.94	6.95	6.96
N=256	3.56e-08	4.89e-08	1.38e-07
ratio	126	127	127
rate	6.98	6.98	6.99
N=512	2.82e-10	3.86e-10	1.09e-09
ratio	127	128	128
rate	6.99	6.99	7.00
N=1024	2.21e-12	3.03e-12	8.52e-12

**Table 23.** Convergence of the 7th-order method when computing the flux divergence for one step of the 3D linear advection problem.

	1-norm	2-norm	inf-norm
N=32	1.62e-02	1.91e-02	3.63e-02
ratio	160	160	162
rate	7.32	7.33	7.34
N=64	1.02e-04	1.19e-04	2.25e-04
ratio	227	228	229
rate	7.83	7.83	7.84
N=128	4.47e-07	5.22e-07	9.84e-07
ratio	248	248	249
rate	7.96	7.96	7.96
N=256	1.80e-09	2.10e-09	3.96e-09
ratio	254	254	254
rate	7.99	7.99	7.99
N=512	7.09e-12	8.27e-12	1.56e-11
ratio	257	257	257
rate	8.01	8.01	8.01
N=1024	2.76e-14	3.22e-14	6.05e-14

**Table 24.** Convergence of the 8th-order method when computing the flux divergence for one step of the 3D linear advection problem.