Information Sciences 586 (2022) 326–343

Contents lists available at ScienceDirect

Information Sciences

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

An efficient GPU implementation and scaling for higher-order 3D stencils

Omer Anjum [a,*,1], Mohammad Almasri [a,1], Simon Garcia de Gonzalo [b], Wen-mei Hwu [a,*]

[a] Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1308 West Main Street, Urbana, IL 61801, USA
[b] Department of Computer Science, University of Illinois at Urbana-Champaign, 1308 West Main Street, Urbana, IL 61801, USA

ARTICLE INFO

Article history:
Received 27 November 2019
Received in revised form 20 September 2021
Accepted 14 November 2021
Available online 2 December 2021

Keywords:
Finite difference solver
Fluid dynamics
GPU
High-order 3D stencil
MHD
Register blocking
Scaling
Scatter

ABSTRACT

Stencil computation patterns are the backbone of many scientific and engineering simulations. The stencil computation is known to be constrained by its high demand of memory bandwidth, which limits performance on accelerators such as GPUs. Prior GPU-based approaches concentrated on stencils with only axis-aligned grid points with 2D caching schemes. However, for stencils with non-axis grid points, prior approaches use 3D caching or multi-pass 2D caching schemes. These methods suffer from either large number of global memory accesses or required large size of the shared memory. In this work, we present an efficient GPU implementation scheme "Scatter Without Write Conflict" (SWiC) for large advanced 3D stencil patterns involving non-axis-aligned grid points. Unlike other 3D caching schemes, SWiC only needs 2D caching and a single pass over the stencil per iteration. SWiC achieves, significant reductions in the global memory accesses, without increasing the size of shared memory. SWiC can also be applied to simple axis-aligned stencils without any performance loss. Moreover, we propose a scalable implementation for the halo region exchange of 3D stencils on multi-GPU nodes. For evaluation, we test SWiC on three Nvidia GPU generations and show that our approach significantly outperforms existing state-of-the-art GPU implementations with a speedup ranging from 1.6× to 5.75×. We also provide a detailed scaling analysis in multi-node and multi-GPU environments.

1. Introduction

High-performance computing (HPC) applications extensively employ "stencils" to solve partial differential equations to characterize and predict physical entities such as heat, sound, velocity, pressure, density, elasticity, magnetohydrodynamics, electromagnetism, and electrodynamics. Some of the applications include earth weather predictions, space weather predictions, acoustics, star simulations, and geomagnetic field simulations.

In order to update a grid point, a "stencil" determines the neighboring points in a two-dimensional (2D) or a three-dimensional (3D) grid space to be used for the update. Different applications use stencils of different orders and numbers of dimensions based on their requirements. The order of a stencil specifies the number of neighboring grid points required along each dimension in the grid space. Lower-order stencils (usually requiring one or two axis-aligned neighbors), as shown in Fig. 1(b), might suffice for some applications. However, to accurately simulate more challenging systems such as the

* Corresponding author.
E-mail addresses: oanjum@illinois.edu (O. Anjum), w.hwu@illinois.edu (W.-m. Hwu).
[1] Omer Anjum and Mohammad Almasri have equal contribution.

https://doi.org/10.1016/j.ins.2021.11.042
0020-0255/

dynamics of turbulent fluids, higher-order stencils with both axis-aligned and non-axis-aligned grid points, as shown in Fig. 1(a), are often required. The red points in Fig. 1(a) are the non-axis grid points along diagonals in the x, y, and z planes. The Himeno Benchmark [1] also uses this stencil.

In this paper, we will focus on GPU-based algorithms for stencil computations. GPU-based stencil kernel optimizations use a wide range of tiling techniques. These techniques rely on automatic code generation, compiler optimizations, and manual tuning in the spatial domain [2–6,1,7–9] and time domain [10–15]. The goal of tiling is to increase data reuse and reduce memory bandwidth consumption. For spatial tiling, 3D stencils are particularly challenging due to their large tile size that requires a large amount of on-chip shared memory to be effective. To overcome this challenge, prior work resorts to 2D caching of grid points along the x-axis and the y-axis while storing grid points along the z-axis in thread private registers [2–6,1,7–9]. However, for 3D stencils, such as that shown in Fig. 2(a), the off-axis grid points (colored in blue) are not visible to the updating thread. These off-axis grid points are stored in the private registers of some other threads. In order to make those visible, prior approaches are needed to fall back to 3D caching in the shared memory. Unfortunately, 3D caching increases shared memory usage and results in poor GPU occupancy, especially for high-order stencils [3,6,7,5]. A multi-pass approach with 2D caching has been proposed [5] to improve the GPU occupancy. The main drawback of multi-pass caching is the high number of global memory transactions. In this paper, we take a different approach from the previous work. For example, to update grid point at the center of cube shown in Fig. 2(a) prior approaches require either multiple passes or 3D caching. We improve the data reuse in the spatial domain without requiring multiple passes per stencil iteration or 3D caching in shared memory. Our proposed approach is called as SWiC. Instead of using thread private registers for the input [2,5,3,6,1,7–9], we use thread private registers to accumulate partial output updates. A 2D plane is read into the shared memory. The contributions of the input grid points in the shared memory are then scattered to the accumulators corresponding to grid points in the past and future planes. Starting from the first plane of the 3D compute domain, as shown in Fig. 2(b), a partial update is calculated using each grid point in that plane. The colored square and circle indicate the axis and non-axis-aligned grid points of the same color, respectively, in the plane that is contributing to the partial update for all the points in any plane. As the thread progresses through the planes, the update reaches its final value when the thread has accu-

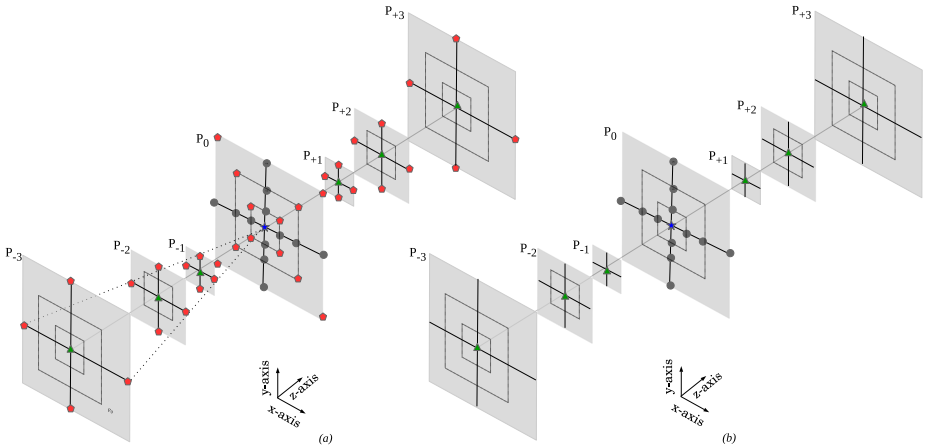

Fig. 1. A 3D illustration of stencils. (a) 55-point stencil to calculate momentum with cross derivative term. (b) 19-point stencil to calculate momentum without cross derivative term.

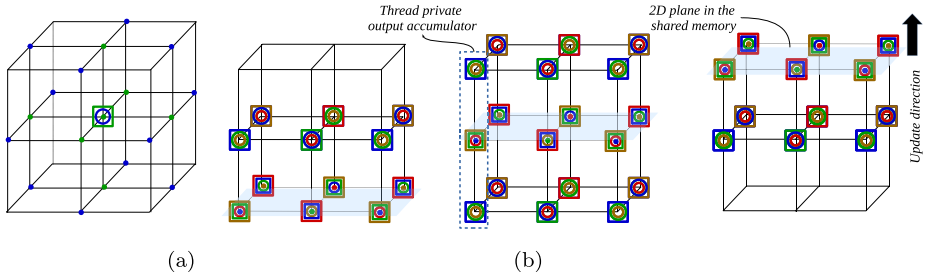

Fig. 2. An overview of the approach with an order-1 stencil. In the actual implementation, the stencil has mirror image around the front side coming out of the page. (a) With input register queue 3D caching is required to update the point at the center since the blue diagonal points other than those in the center plane are not visible to thread updating point at the center. (b) Square and circle indicate an update from axis-aligned or diagonal grid point, respectively. The color of the square and circle corresponds to the color of the grid point. With output register queue acting as accumulator, 2D caching is sufficient. As thread progresses through different planes, partial sums are added to the accumulator contents until thread has accessed all the required grid points.

mulated all partial results of all the required grid points. When the result is ready in an accumulator, it is sent to the global memory, releasing this accumulator for a new grid point as the thread moves to the next plane. Implementing accumulators as register queue helps to mitigate the register pressure where the size of the queue is bounded to $(2 \times \texttt{stencil} - \texttt{order} + 1)$. Since the accumulators are stored in the private registers of each thread, SWiC completely avoids update conflicts of the general scatter approach [8,9]. None of the prior work considered scatter method and accumulator register queue for the stencil computation.

Our proposed approach is motivated by stencils with non-axis-aligned grid points. Moreover, it can also be applied to 2D or 3D stencil patterns with or without non-axis-aligned grid points without any performance loss relative to prior approaches. Optimizations that apply tiling in the time domain [10–15] are orthogonal to our proposed approach and outside the scope of our discussion.

We also study scaling of large 3D stencils with non-axis grid points on multi-GPU nodes. For large-scale stencil applications, the domain can grow larger than a single GPU's memory [16–18]. Modern HPC systems increasingly offer nodes with multiple CPU sockets and multiple GPUs. Thus, in large-scale stencil applications, it is common to partition the compute domain into smaller subdomains such that each GPU processes a separate subdomain. With this configuration, distributed stencil algorithms require halo exchange between neighbor GPUs, to update grid points at subdomain boundaries. Neighbor GPUs that are involved in the halo exchange can reside on the same node or across nodes. The Compute Unified Device Architecture (CUDA) and Message Passing Interface (MPI) libraries are low-level libraries that feature fine-grain control for intra-node (among GPUs) and inter-node data communication. In this paper, we introduce a lightweight library that efficiently exploits CUDA and MPI to automatically exchange halos of 3D stencils between neighboring GPUs on the same node and across nodes. This library partitions a stencil evenly across nodes along a single axis (e.g. the z-axis, user-defined). Each node's subdomain is then evenly partitioned across GPUs along either the same axis or the other two axes (e.g. the x and y axes). After the partition, the library uses aggregated buffers and MPI for inter-node communications. For intra-GPU communication, we use CUDA asynchronous data exchange to enable GPUs to send and receive multiple independent halo data simultaneously. If Nvidia peer access is enabled, our library exploits the NVlink to increase the communication bandwidth. We will show, in Section 5, a detailed scaling analysis of our library using 19-point and 55-point stencils.

This paper makes the following contributions:

- We propose an efficient GPU algorithm for computing stencils with arbitrary patterns involving axis- or non-axis grid points without the need for 3D caching or multiple passes per stencil iteration.
- The proposed approach for the kernel increases the data reuse and decreases the memory bandwidth consumption, resulting in significant kernel speedup.
- As the problem size increases, the runtime scales more efficiently than the previous kernel implementations.
- We show that our proposed approach is effective on the latest generations of GPU architectures.
- Our proposed approach requires fewer quantities to communicate during halo exchange when used in a multi-node execution environment, compared to the multi-pass approach with 2D caching.
- We also present a library to partition 3D stencils and efficiently communicate halo data between GPUs within and across nodes. We provide a detailed scaling profile of the application in a multi-GPU multi-node setup.

The rest of the paper is organized as follows: Section 2 briefly explains the constraints in GPU architectures. Section 3 focuses on the related work. Section 4 explains the implementation details of our solution. Section 5 presents the results and discussion. Finally, Section 6 presents our future work and conclusions.

2. GPU implementation constraints

2.1. Memory bandwidth

Stencil applications are known to be bandwidth bound, whereas memory bandwidth has not been increasing at the rate of the FLOPS increase from one GPU hardware generation to the next [19,20]. This presents a major challenge for those who seek an efficient higher-order stencil implementation, as highlighted by several prior works in Section 3. In general, any approach to reducing memory bandwidth limitation must exploit data locality/reuse. In GPUs, data reuse is mainly achieved through registers and shared memory usage. Unfortunately, the increased amount of shared memory required by each thread will significantly reduce the occupancy (i.e. the number of threads scheduled for simultaneous execution), and potentially degrade sustained performance.

2.2. Arithmetic intensity

Arithmetic intensity for an application is measured as the number of arithmetic operations performed for each operand it loads from memory. As an example, let us consider a basic Navier–Stokes equation [21], which is widely used in the fluid dynamics, with four physical entities - a 3D velocity vector and a density scalar - at each grid point. To update a single grid point in a 55-point non-axis-aligned stencil, as shown in Fig. 1(a), one needs to fetch (number of grid points in the stencil)×

(number of physical quantities) = 220 operands in order to perform a total of 291 operations, leading to an arithmetic intensity of $291/220 = 1.32$. The estimated number of operations is collected by the Nvidia Visual Profiler [22]. The maximal reuse factor for the 55-point stencil is 55, i.e., the number of grid points in the stencil. This is because each grid point is updated using 55 neighboring grid points including itself, as shown in Fig. 1(a). In the ideal case, if we are able to achieve this maximal reuse factor, it would result in an arithmetic intensity of $1.32 \times 55 = 72.75$. However, as one partitions the grid points for processing by CUDA thread blocks, neighboring thread blocks share some amount of grid point data with each other as "halos," as shown in Fig. 3. Such shared grid points are requested independently by the adjacent thread blocks leading to reduced data reuse and more global memory references. The achieved arithmetic intensity is thus expected to be less than 72.75.

For the Maxwell GPU generation, the peak memory bandwidth is 224 GB/s and the peak compute rate is 4,612 GFLOP/s, leading to a required arithmetic intensity of roughly 82.4 to sustain its peak compute rate. Similarly, for Pascal P100 and Volta V100, the required arithmetic intensity to sustain their peak compute rates are (10 TFLOP/s/ 720 GB/s) 55.56 and (15 TFLOP/s/ 900 GB/s) 66.64, respectively. We observe a reduction in the width of the memory wall [19,20] from Maxwell to Pascal. The reason is a different memory technology, GDDR5 in Maxwell vs HBM2 (on-chip stacked memory) in Pascal. In transition from Maxwell to Pascal, the memory bandwidth is increased by a factor of three whereas the peak compute rate is increased by a factor of two.

However, there is no other new memory technology in sight for another jump in the memory bandwidth. The memory technology remains largely the same from Pascal to Volta. As a result, we observe that the memory wall resumed its widening trend, requiring more reuse per operand for full GPU compute rate utilization. In the ideal case, where the reuse factor equals the number of grid points in the stencil, the proposed approach provides enough reuse to be compute the bound in both Pascal and Volta generations. However, the reuse is affected by the halo-zones and various limiting factors in both hardware and software. As a result, the sustained compute throughput for stencil applications tends to be significantly lower than the possible peak for these GPU generations.

2.3. Limited storage

Another challenge arises from the large size of the stencils. Let us consider a small CUDA thread block having only a single warp with (32,1,1) threads in (x,y,z) dimensions. A warp is the smallest unit of threads for lock-step execution. Assume that each thread updates one grid point. For an order-3 stencil, a thread block needs to access three halo grid points on each side. For a thread block with only one warp, the number of grid points needed is $38 \times 7 \times 7 = 1862$. For a hydrodynamic solver updating only velocity vector and density scalar in single precision, by storing all the input grid points in the shared memory, the number of bytes needed would be $1862 \times 4(\texttt{physicalquantities}) \times 4(\texttt{bytes/float}) = 29,792$ bytes. Keeping in mind that the size of shared memory per streaming multiprocessor (SM) is 96 KB for the latest GPU generation, only $\lfloor 96000\texttt{bytes}/29,792\texttt{bytes} \rfloor = 3$ thread blocks would be able to get scheduled to each SM at a time. If the GPU can support 64 active warps per SM, three thread blocks with a total of three warps would lead to $3/64 = 4.6\%$ occupancy. With such low occupancy, there are not enough active warps to hide memory latency and fully utilize the computing resources. Furthermore, the small number of grid points updated by each thread block severely limits the achievable arithmetic intensity ratio and limits the sustainable compute rate to an extremely small fraction of the peak compute rate. This is because the number of halo grid points far exceeds the number of grid points being updated for each thread block. Increasing the number of warps per thread block and thus the number of grid points updated by each thread block would increase thread-level parallelism as well as the data reuse. This can indeed be beneficial for memory bandwidth. However, the number of bytes required by each thread block also increases, which can unfortunately further reduce the occupancy per SM.

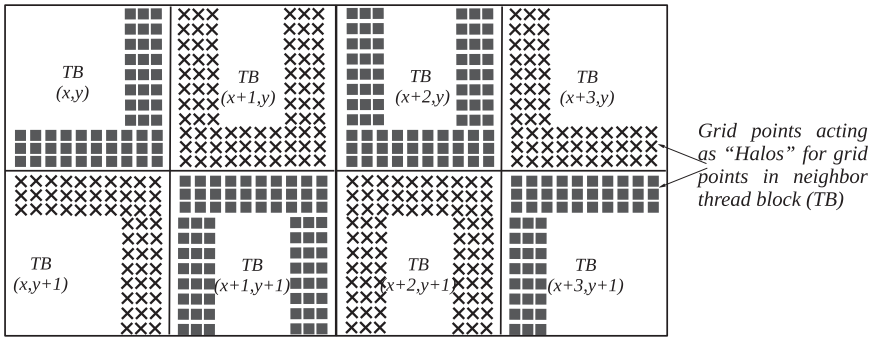

Fig. 3. "Square" and "cross" in a thread-block (TB) represents halo grid points. These halo grid points are exchanged between the adjacent TBs to update their grid points.

3. Related work

Magnetohydrodynamic (MHD) simulations [23–26], which require complex 3D stencils and up to 10^{10} grid points, employ a multiple of 10^4 CPU cores and up to 20 million CPU hours. "*Pencil Code*" [27] is a state-of-the-art CPU production code, used for MHD simulations. Due to the limited throughput of CPU-based clusters, the "*Pencil Code*" captures only the global phenomena and not the local phenomena such as "*sun spot*" formation. Since GPUs can provide much higher throughput for applications with regular data access pattern than CPUs, GPUs have been widely explored for stencil computations.

Earlier GPU stencil implementations [2–4] mainly considered only axis-aligned grid points and are inefficient when solving for non-axis-aligned grid points. For example, time-tiling [10–13] combines calculations from multiple time steps where shared memory is used to store intermediate results, reducing the number of thread blocks that can be simultaneously scheduled per SM. Applying time-tiling for 3D stencils requires thread block size to be reduced due to limited amount of shared memory available in the GPU. Furthermore, time-tiling also suffers from computation overhead due to redundant calculations at halo regions. To overcome the large shared memory requirement, sliding-window time-tiling [14] was proposed. However, sliding-window time-tiling still requires multiple shared memory buffers to cache the planes required to update a single plane. For Nvidia Kepler and Maxwell GPUs, time-tiling is restricted to only two time steps for an order-1 stencil in order to update only one physical quantity. When the stencil size and the number of quantities to update are increased, this technique faces significant reduction in GPU occupancy. Time-tiling is orthogonal to our proposed approach; however. the interaction between SWiC and time-tiling is outside the scope of this work.

A number of other strategies for stencils with non-axis-aligned grid points have been proposed [6,1,7]. However, these techniques are limited to comparatively smaller size stencils. Furthermore, in order to handle off-axis grid points, current approaches require all the relevant input planes to be cached in shared memory (3D caching) in order to update a plane at the cost of reduced GPU occupancy. Domain specific language (DSL) for stencil computations [28] has been proposed with the ability to reorder instructions to reduce register pressure. However, it currently does not have the ability to automatically apply the optimizations proposed in our work. Large shared memory footprint is major concern for all the methods discussed above.

In order to mitigate the large memory footprint needed for large stencils with non-axis-aligned grid points, another approach, called 19P is proposed in [5] which decomposes computations into two passes. This method simplifies stencil computation, such that instead of using a 55-point stencil, Fig. 1(a), it uses a 19-point stencil, Fig. 1(b), in each pass. The drawbacks of this technique are: 1) Data reuse factor is small because of fewer grid points per stencil in each pass compared to the single pass approach; 2) partial results need to be stored in global memory after the first pass which increases the global memory traffic; 3) redundant calculations are needed on the outer halos, 4) Number of applications can be limited; and 5) communication overhead per time step is high. [3] In Table 1,2 we formulate the number of grid points that are communicated for the 19P approach, where N is the number of quantities and $N_{x,y,z}$ are grid dimensions. The term $(N+1)$ for 19P approach comes from the fact that, in addition to storing the final result, it also needs to communicate the intermediate results after the first pass. For SWiC, the additional term $\texttt{FDM}-\texttt{Order}\times\left(N_x+N_y+N_z\right)$ represents the dimension of an edge shown in Fig. 4, which is significantly smaller than the rest of the halo regions. For domain size of $N_{x,y,z}=128$ and stencil-order of three, the numbers of halo points in a hydrodynamic simulation for 19P and SWiC are 294,912×5 and 308,736×4, respectively. SWiC has a 16% lower communication overhead than the 19P approach. SWiC is also applicable to any other stencil irrespective of the size and shape, without performance degradation. SWiC does not require 3D caching in the shared memory, and instead "scatters" the input grid-points to output register queues. To our knowledge, the work proposed in this paper has not been considered in any prior GPU related work, whether it is a hand-tuned code or a DSL automatic code generation.

Moreover, we also present a lightweight library for halo exchange for large-scale stencil applications. In [29], 1D domain partition is suggested (e.g., partition along z-axis). In 1D partition, increasing domain size along x-axis and/or y-axis would require reducing the domain size along z-axis such that the total domain size still fits inside the GPU memory. This would increase the communication-to-compute ratio and limit the scaling along other directions. Moreover, stencils with only axis-aligned grid points are considered in [29]. Overlapped communication and execution is emphasized in [30] as a key to achieve high performance. However, it does not present further details on how to achieve the implementation. In [31,32] a node aware communication topology in HPC clusters equipped with GPUs is presented. These communication strategies apply in general to stencil applications as well. However, 3D stencils with non-axis-aligned grid points are more challenging due to the irregular memory accesses caused by exchanging non-contiguous halo grid points, as explained in Section 4.2. This challenge is not addressed in these prior works. We present an effective communication library for halo exchange. Our library supports stencils with and without non-axis-aligned grid points.

[3] No 3D caching is needed in the shared memory, similarly to SWiC, but has more communication overhead.

Table 1
Number of grid points in Halo exchange.

Method	Number of Halo points per quantity
19P*	$(N+1) \times (N_xN_y + N_yN_z + N_zN_x)(\texttt{FDM} - \texttt{Order}))$
SWiC	$N \times \texttt{FDM} - \texttt{Order} \times ((N_xN_y + N_yN_z + N_zN_x) +$ $\texttt{FDM} - \texttt{Order} \times (N_x + N_y + N_z))$

Table 2
Measurements of runtime spent in boundary (Halo) exchange.

		ρ and u Time (ms)	Grad (Div) Time (ms)	% of 19P runtime	% of SWiC runtime
Volta	256^3	0.3	0.062	8.6	13.4
	512^3	1.1	0.26	3.6	6.2
Pascal	256^3	0.35	0.086	7.1	9.7
	512^3	1.47	0.35	3.4	4.9

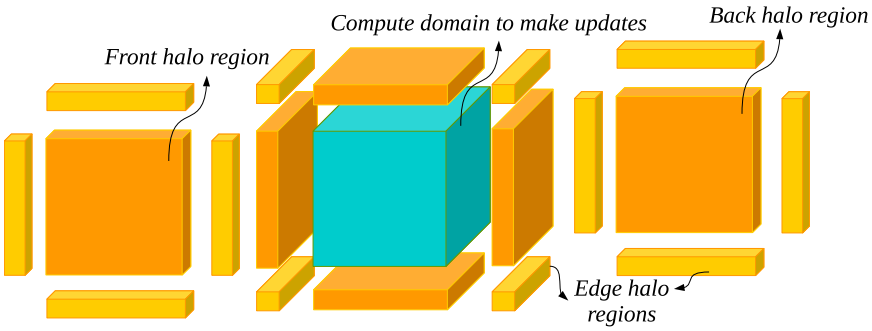

Fig. 4. An illustration of 3D grid with the compute domain at the center, surrounded by halos and edges.

4. The design and implementation of SWiC

4.1. Single-GPU implementation of SWiC

An overview of a 3D grid is shown in Fig. 4. The stencil sweeps through the compute domain shown as the big cube in the center (green) surrounded by the halo regions (orange and yellow). A high-level view of different steps in SWiC is shown in Fig. 5. Compared to the prior work where an input register queue per thread is used to store the input operands, we manage an output register queue per thread of the same length, acting as accumulators for the partial outputs. Regardless of the stencil pattern, axis or non-axis aligned, we read a 2D tile/plane from the global memory into shared memory, as shown in step A in Fig. 5, where the white small squares in the shared memory indicate the pattern of the data points needed by a single thread, at the center of a thread block, in order to update its private output register queue. This pattern is the same for each thread in a thread block. Each point in the pattern contributes to the partial sum of one or more of the private accumulators via "Scatter," (Step B). Note that the z-axis is rotated for readability. Once the computation is complete for the current z-plane, the thread moves to the next plane and adds another partial sum to its private accumulators. When the sum in the oldest accumulator is ready, it is sent to the global memory, (Step C). This accumulator is then reused for the grid point at the z-plane P_{c+1+hd}, where c and hd are the index of the current z-plane and halo depth, respectively. At step D pointers for input and output grid points are swapped. Scatter is applied in the z-direction in which a thread is streaming.

To understand the algorithm in more detail, let us now consider different states (steps) of SWiC in Fig. 6, which presents the updates made by a thread with index (i,j) to its private accumulators $A_{i,j}(-hd : +hd)$ as it traverses through the z-planes. Here, the notation $A_{i,j}(-hd : +hd)$ means an accumulator holding the partial sums for grid points on a single lane along the z-plane located at row i and column j on the xy plane. Here $(-hd : +hd)$ is the window size of accumulator which spans over the past, current, and future hd z-planes. The length of the accumulator window is $2 \times \texttt{stencil} - \texttt{order} + 1$. The stencil we use in Fig. 6 is of the order of two and also has off-axis grid points as in Fig. 1(a). The color at each state of the accumulator represents the points used to update its contents, as detailed in the bottom right corner of Fig. 6. The status and activities of the accumulators are shown at the bottom left corner and the accumulator's color at any state indicates the color of the points it uses in that state. The accumulator is circled in the bottom part of the Fig. 6 when it has the contributions from

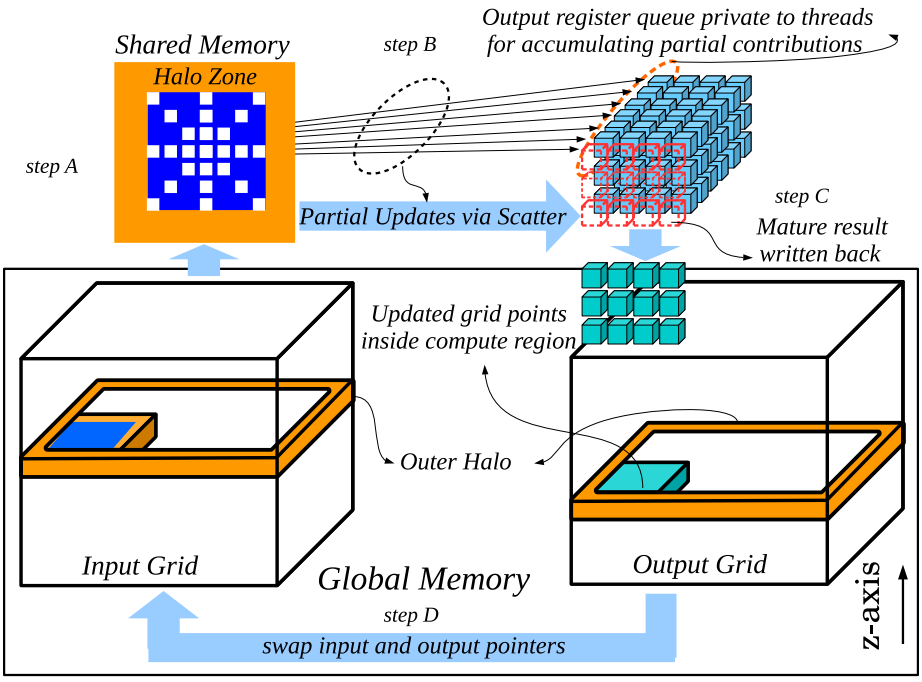

Fig. 5. An overview of application flow: Threads enter the integration loop and copy the current z-plane value to the shared memory, starting from halo region. Threads update their private accumulators with partial sums consuming the values from the shared memory. Once accumulator with the final result is stored back to the global memory it is reused for the upcoming grid point at z-plane P_{c+1+hd}, where c and hd are the index of current z-plane and halo depth, respectively. Once the grid is updated, the pointers for input and output grids are swapped, making the output from integration time step i the input for integration step $i + 1$.

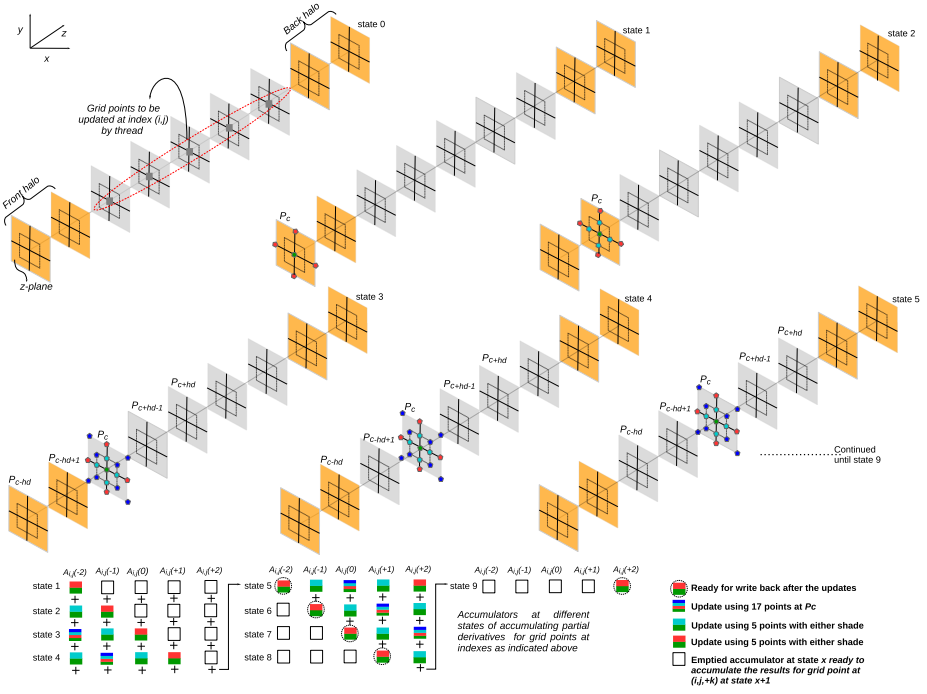

Fig. 6. A 3D illustration of a thread activity at index (i,j) as it traverses through the z planes. How a thread updates it accumulators $A_{i,j}(-k:+k)$ is also shown at each state.

all the partial sums and the final result is ready for the write back to the global memory. The first column at the bottom of Fig. 6 indicates that the final content of $A_{ij}(-2)$ is the sum of the contributions made to the accumulator at different states. At state 1, it uses the red and green grid points, at state 2 the light blue and green grid points, at state 3 the grid points of all the colors in plane P_c, at state 4 the light blue and green grid points, and finally at state 5 the red and green grid points. At the end of state 5, the content of accumulator $A_{ij}(-2)$ is complete and ready to be written back to the global memory.

Algorithm 1: SWiC

Require: Integer i belongs to the set of indices of full domain D including boundaries.
Require: Accumulator A of size $(1 + 2 \times hd)$ where hd is the halo depth
Require: B is boundary region in D
Require: Subscript s indicates axis-aligned grid points within stencil
Require: Subscript p indicates stencil grid points at current plane along axis x,y and diagonal xy
Require: A_i is the accumulator corresponding to input at index i
  for integration step s = 1 **to** 3 **do**
    for all $i \in D$ & $i \notin B$ **in parallel do**
      ρ[i] = Compute_rho(ρ_s, ...)
      A[-hd:hd] = Compute_partial_gradient_of_divergence_of_velocity($\mathbf{u}_p$)
      if A_i is ready **then**
        $\mathbf{u}$[i] = $\widetilde{\mathbf{u}}$[i] + A_i
      end if
      Store update to the global memory for $\mathbf{u}$[i] and ρ[i]
    end for
  Exchange periodic boundary conditions for ρ and $\mathbf{u}$
end for

Algorithm 2: 19P

Require: Integer i belongs to the set of indices of the full domain D including boundaries.
Require: density(ρ) and velocity($\mathbf{u}$) as input.
Require: B is boundary region in D
Require: Subscript s indicates axis-aligned grid points within stencil
  for integration step s = 1 **to** 3 **do**
    First pass kernel:
    for all $i \in D$ **in parallel do**
      if $i \notin B$ **then**
        ρ[i] = Compute_rho(ρ_s, ...)
        $\widetilde{\mathbf{u}}$[i] = Compute_partial_velocity($\mathbf{u}_s$, ...)
      end if
      if $i \in B$ **then**
        $\mathbf{u}_{\text{div}}$[i] = Compute_divergence($\mathbf{u}_s$, ...)
      end if
      Store to the global memory $\widetilde{\mathbf{u}}$[i], $\mathbf{u}_{\text{div}}$[i] and ρ[i]
    end for
    Periodic_boundary_exchange($\mathbf{u}_{\text{div}}$)
CUDA implicit synchronize
    Second pass kernel:
    for all $i \in D$ & $i \notin B$ **in parallel do**
      Read from global memory $\widetilde{\mathbf{u}}$[i], $\mathbf{u}_{\text{div}}$[i]
      $\mathbf{u}$[i]=Gradient_of_divergence($\mathbf{u}_{div_s}$, ...)+$\widetilde{\mathbf{u}}$[i]
    end for
    Store to the global memory $\mathbf{u}$[i]
    Periodic_boundary_exchange(ρ, $\mathbf{u}$)
end for

The following are the per thread steps in SWiC, as shown in Algorithm 1:

1. Set the index of current z-plane P_c pointing to the front halo as shown in state 1.
2. Read the 2D tile of input grid points from the global memory at the plane (i,j,P_c) to the shared memory as illustrated in Fig. 5 step A.
3. Read from the shared memory all the grid points required to update $A_{i,j}(-2:+2)$. The green grid point at the center of each plane is also the value which needs to be updated. Thus, in addition to the partial contributions a thread makes, the value to be updated is also added to the accumulator.
4. The result in the oldest accumulator is written back to the global memory, as shown in step C of Fig. 5. In Fig. 6 the first result is ready at state 5 stored in $A_{i,j}(-2)$.
5. Shift the contents of output accumulator register queue by 1. The accumulator at index $A_{i,j}(hd+1)$ is now free to be used for updating the input grid point at P_{c+hd+1}.
6. Set P_c to the next z-plane and repeat steps (2) through (5) until P_c is set to the last slice in the back halo region.
7. Once the integration step over whole grid is complete, the pointers for the input and the output grids are swapped for the next integration step as in Fig. 5 step D. By swapping the pointers, the output of the current integration step will become the input for next integration step.

4.2. Multi-GPU multi-node Halo exchange

For large-scale stencils, full domain size can be too large for the GPU memory. Prior work such as [16–18] considered stencils with up to 10^{10} grid points, a range of 1–8 quantities, and subdomains per GPU of 512^3. On multi-GPU nodes, large domains can be partitioned such that each subdomain fits inside a single GPU memory. However, partitioning the compute domain incurs inter-GPU and/or inter-node communication due to halo exchange.

Halo exchange is required to update the grid points at the boundaries of each subdomain. The number of neighbors involved in the halo exchange depends on the shape (e.g., with or without non-axis-aligned grid points) and the type (2D/3D) of the stencil, and the boundary conditions used in the application. In this work, we consider 3D stencil with non-axis-aligned grid points, shown in Fig. 1(a), with periodic boundary conditions. This stencil requires the six face halo regions of the subdomain to be exchanged with the neighbors. Moreover, because of the non-axis-aligned grid points in the stencil along the xy, yz, and zx planes, four edges of the subdomain along each of these planes should be exchanged with the neighbors. In Fig. 7, we show an example where the subdomain at the center receives halos from the neighboring subdomains in the xy-plane.

Due to the cumbersome nature of the halo exchange process, especially along the diagonals in the 3D domain, we introduce a lightweight library that automatically partitions a 3D domain across nodes and GPUs and effectively communicates halos with neighbors. The library starts with dividing 3D domain evenly across nodes of a cluster along one of the three axes (e.g., the z-axis, a user-defined parameter). The assumption here is that the nodes are symmetric processing units that also have identical numbers and generation of GPUs. The nodess' subdomain is further divided evenly along the remaining two axes and each subdivision is assigned to a single GPU (say x and/or y axes, user-defined parameters). An overview of domain partition in our multi-GPU multi-node setup is shown in Fig. 8. The halo exchange pattern resulting from our partitioning strategy of the compute domain is illustrated in Fig. 9. On the same node (Fig. 9 (a)), GPU0, for example, exchanges the face halo along the x-axis with GPU1, the face halo along the y-axis with GPU2, and the non-axis xy edge with GPU3. Across the nodes, GPU0 of node0 exchanges the face halo along the z-axis with GPU0 of node1 (Fig. 9 (b)). Moreover, GPU0 of node0 exchanges the edge halos along the yz-plane with GPU1 of node1 and the edge halo along the xz-plane with GPU2 of node1((Fig. 9 (c)). The library uses MPI and CUDA APIs to perform the inter- and intra-node communications, respectively.

4.2.1. Inter-node communication

Our library uses MPI API to send and receive halo regions between nodes in the z-direction (this is the default axis; however, any axis can be used). Fig. 10 describes the inter-node communication steps in our library. To exchange halo regions between two neighbor nodes, the library asynchronously writes all GPU face halos in the z-direction (stored sequentially in the GPU memory) to a single CPU memory-pinned buffer (the sending MPI buffer). To overlap multiple CPU-GPU communications and attain the highest possible bandwidth between the GPU and the CPU, we use *cudaMemcpyAsync* with pinned memory on the host CPU. Notice that the halo edges are parts of some halo faces; therefore, they are not transferred redundantly. Moreover, each node reserves another CPU pinned-memory buffer (the receiving MPI buffer) to receive the face halos along the z-direction from other nodes. Then, the library performs a single *MPI_Sendrecv* to collectively exchange the faces and edges in the z-direction between the two neighbor nodes. The same procedure is followed to exchange the faces and edges in the other direction. Once the MPI message is received on a node, the library asynchronously writes the face and edge halos to their designated GPU buffers on the target GPUs using *cudaMemcpyAsync*.

4.2.2. Intra-node communication

CUDA API is used to exchange halo regions between neighbor GPUs on the same node along the x and/or y axes. Fig. 10 describes the intra-node communication steps in our library. Since the face and edge halos in the x and y directions are not

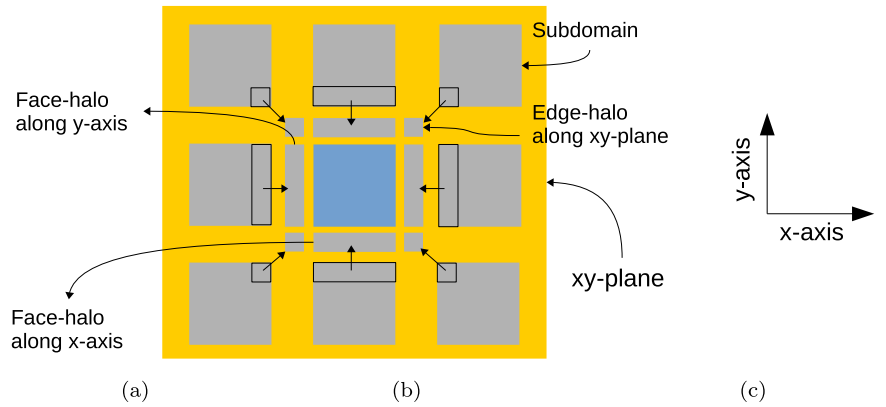

Fig. 7. An illustration of halo exchange in the xy-plane. The subdomain at the center receives two face-halos along each of the x and y axes. Due to the non-axis-aligned grid points in the stencil along xy-plane, four edges are copied from neighboring subdomains in the xy-plane. Not shown in this figure, two face-halos are also exchanged along the z-axis and four edge halos each in the yz-plane and xz-plane.

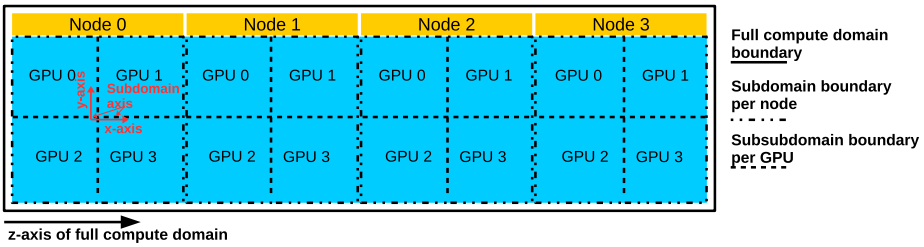

Fig. 8. An overview of full compute domain partition across nodes and GPUs.

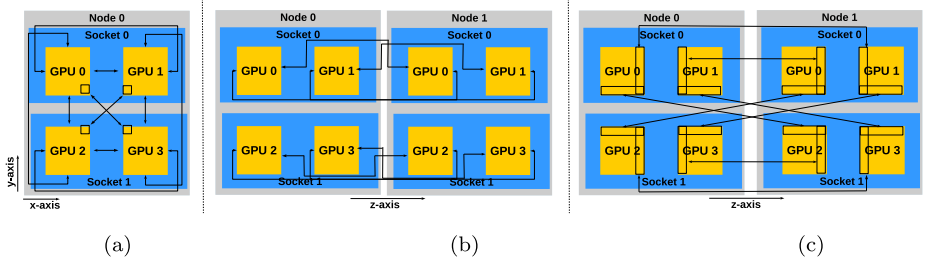

Fig. 9. An illustration of halo exchange in multi-node environment where different nodes are on different z-planes: (a) Intra-node communication along x and y axes. (b) Inter-node communication to exchange edges along z-axis. (c) Inter-node communication to exchange faces along z-axis.

stored sequentially in memory, the library uses a GPU kernel to pack the scattered face and edge halo grid points into a GPU buffer. Then, it uses *cudaMemcpyAsync* with *cudaMemcpyDeviceToDevice* option to perform an asynchronous data copy from one GPU to another. If CUDA peer access is supported and enabled (the library enables peer access if it is supported in hardware), GPUs exchange data between each other directly though the Nvlink without involving the CPU. Otherwise, the CPU mediates the data transfer between GPUs. Lastly, a GPU kernel is used to unpack the GPU buffer to the actual halo region. This packing and unpacking technique achieves the best performance compared to either using direct GPU kernels or CUDA unified memory. A kernel on a GPU can directly access the halo regions of other GPUs when the peer access between this GPU and the other GPUs is enabled. However, the halo exchange time can be quite high due to irregular memory accesses. On the other hand, the CUDA unified memory approach is easy to implement since each GPU can directly access other GPU compute domains and halo regions from the CUDA kernel. However, sharing unified memory pages between GPUs causes many page faults due to page migration among GPUs and CPU [33]. CUDA unified memory runtime *cudaMemAdviseSetReadMostly* routine can be used to create read-only copies at the destination GPUs to avoid page faults. However, during the halo exchange, data is frequently read and written in the unified memory space. Thus, this runtime support routine is not useful for halo exchange in stencil applications. For self-halo exchange where the axis is not partitioned, we use a GPU kernel to exchange halo regions on the same GPU.

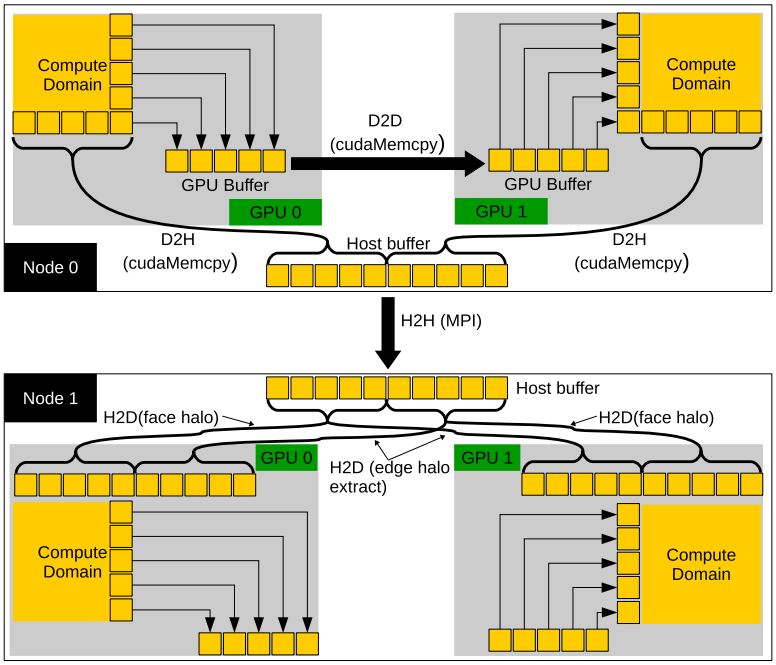

Fig. 10. An illustration of our approach to exchange halo regions on single-node and multi-node setups. For single-node, GPU0 gathers non-contiguous data from its compute domain to a buffer, uses *cudaMemcpy* to transfer data to GPU1 (D2D), and GPU1 scatters the received GPU buffer to the halo region of GPU1. For multi-node, each GPU on node0 transfers its data to a shared buffer (D2H) and the shared buffer is transferred using MPI to node1 shared buffer (H2H). Then, node1 transfers data from the shared buffer to each GPU halo region (H2D). Note that we did not need extra buffers for D2H and H2D steps because the halo region of each GPU is stored sequentially in the GPU memory.

5. Results and discussion

In our experiments, we use a high-order 3D stencil with both axis and non-axis grid points, as shown in Fig. 1(a). We use a periodic boundary condition after each integration step along all the faces of the compute domain. Based on the existing work [5,6,1,7,10–14], 3D caching in the shared memory increases the shared memory footprint which reduces the number of thread blocks that can be simultaneously scheduled on GPU SMs. As discussed earlier in Section 2, only one thread block can be scheduled on an SM if the stencil as in Fig. 1(a) is cached in 3D. The only recent work which does not require 3D caching to solve the stencil in Fig. 1(a) is 19P method [5]. It is closest to our approach, open source, and part of the *"Pencil Code"*, which is a production grade code widely used by the astrophysics community. In order to make direct comparison with 19P method we have implemented stencil computation for Navier–Stokes equation also implemented by the 19P method in [5], to solve the 55-point stencil, depicted in Fig. 1(a). Rather than using a 55-point stencil, the 19P approach uses two phases using 19-point stencil, as in Fig. 1(b), to solve the 55-point stencil. A challenge with the Navier–Stokes equation is the stress tensor term which couples the density and velocity. This coupling requires us to maintain the input register queue of the same size as the output register queue. It increases the register pressure for SWiC and serves as a rigorous test case for our proposed approach. A high-level pseudocode is also presented for both SWiC and 19P in Alg. 1 and Alg. 2, respectively, in order to clarify the algorithmic differences when implementing the Navier–Stokes equation. We will also show the results without the stress tensor term which does not require SWiC to maintain the input register queue.

We ran both the SWiC and 19P approach on the latest three Nvidia GPU architectures: Maxwell, Pascal and Volta. We used CUDA version 9.1 with `-O3` as the optimization level. For the initialization the whole input grid is initialized with a constant 'ρ' and sinusoidal '**u**' along one of the axis in single-precision. Single-precision simulations are valid for smaller runtimes and also for a class of scientific applications such as MHD where 55-point stencil as in Fig. 1(a) is used [34] to capture slowly varying global phenomena. However, if some interesting local phenomenon happens where high resolution is required, such as sun spot formation, the simulation for that local region requires double-precision. All metrics were measured using Nvidia Visual Profiler [22]. For Pascal and Volta, we also did an experiment with a larger grid size of 512^3, taking advantage of the larger memory capacity available in those architectures. For Maxwell we used 256^3 grid points as this is the largest cubic grid size that fits on the Maxwell global memory. The runtimes in Table 3 and Table 4 also include the time required for the boundary exchange time as given in Table 2. Those measurements are roughly the same for 19P and SWiC and significantly smaller than the integration kernel. The time for exchanging gradient of divergence is not relevant for SWiC since it only takes one pass per stencil iteration.

Table 3
Comparison of execution runtime on Maxwell, Pascal P100, and Volta V100 for grid size 256^3.xmllabelt0015

	Maxwell			Pascal			Volta		
	Method		Gain	Method		Gain	Method		Gain
	19P	SWiC		19P	SWiC		19P	SWiC	
Runtime (ms)	20.1	12.7	1.6	6.13	4.47	1.37	4.19	2.7	1.55
SWiC Runtime	–	–	–	12.7	4.47	2.84	4.47	2.7	1.65
Global Loads in Gbytes	4.01	2.29	1.75	4.01	2.29	1.75	2.26	1.53	1.47
Global Stores in Gbytes	1.0	0.54	1.85	1.0	0.54	1.85	1.0	0.54	1.85
Ops/Operand without Reuse	2.41	1.32	0.54	2.41	1.32	0.54	2.41	1.32	0.54
Ops/Operand with Ideal Reuse	45.9	72.76	1.6	45.9	72.75	1.6	45.9	72.75	1.6
Ops/Operand Achieved	3.87	8.5	2.2	3.87	8.5	2.2	6.88	12.74	1.85
Peak Performance Achieved (%)	4.20	8.7	2.07	5.99	10.31	2.02	6.19	12.06	1.94
Data Request Stall (%)	73.8	54.3	1.35	65.65	52.97	1.23	87.25	65.16	1.29

Table 4
Comparison of execution runtime on Pascal P100 and Volta V100 for the Grid Size 512^3.

	Pascal			Volta		
	Method		Gain	Method		Gain
	19P	SWiC		19P	SWiC	
Runtime	50.9 ms	36.8 ms	1.38	35.1 ms	21.8 ms	1.6
Runtime Gain over Pascal	–	–	–	1.45	1.68	1.16
Global Loads in Gbytes	32.14	18.34	1.7	16.74	10.57	1.5
Global Stores in Gbytes	8.0	4.3	1.86	8.0	4.3	1.86
Ops/Operand without Reuse	2.41	1.32	0.54	2.41	1.32	0.54
Ops/Operand with Ideal Reuse	45.9	72.75	1.6	45.9	72.75	1.6
Ops/Operand Achieved	3.87	8.51	2.2	7.43	14.77	1.98
Peak Performance Achieved(%)	5.77	10.02	1.73	5.91	11.95	2.02
Data Request Stall (%)	61.2	44.3	1.38	82.03	66.24	1.23

5.1. Thread block tuning

We ran a CUDA thread block size and dimension tuning experiment on Maxwell, Pascal, and Volta. The optimal configuration, in terms of both runtime and global memory transactions, was 32 threads (in the x dimension) by 4 threads (in the y dimension), as shown in Fig. 11. If we increase the thread block size to 32 by 8, occupancy is increased as there is a bigger pool of warps available for latency tolerance. However, with more threads per block, the number of registers per thread is decreased, which in turn causes register spilling. We can observe the impact of register spilling in Fig. 11 as a sudden rise in the local memory traffic. When we drop the thread block size back to 32 by 2, the halos to compute grid point overhead increase, as shown in Fig. 12, and the total number of grid points being used as halos also increases. These factors contribute to an increase in global memory traffic as shown in 11. We also observe a very slight degradation in runtime. We found these trends to be consistent over different GPU architectures.

5.2. Maxwell, Pascal, and Volta runtimes with varying domain sizes

Table 3 shows a comparison between SWiC and 19P across three different GPU architectures for grid size of 256^3. With increased reuse, increased workload per thread and reduced global memory traffic, SWiC shows a speedup of 1.6×, 1.37×, and 1.55× over 19P for Maxwell, Pascal, and Volta, respectively. As presented in [5] 19P is faster than 55P by 3.6×, implying that SWiC is faster than 55P by 5.76×. The ideal arithmetic intensity for SWiC is 72.75, which is higher than the ideal arithmetic intensity of 19P, which is 45.9. In contrast to 19P, SWiC does all the calculations in one pass to achieve higher reuse factor. Increasing the reuse factor reduces global memory requests and also the required load/store resources and data stalls. Moreover, we also observed that with the increased reuse, the execution speed achieved in terms of FLOPS by the SWiC, is on average twice that of the 19P approach, for all the GPU architectures.

On Nvidia Pascal P100 and Volta V100 we used grid size of 512^3 which is the maximum number of grid points we were able to fit in the global memory. Note that this grid size does not fit into the Maxwell memory. Results are shown in Table 4. We observe that SWiC has, similarly to Maxwell, reduced global memory transactions and data stalls while improving reuse and peak performance compared to 19P. We also observed that as we move from Pascal to Volta we get better performance for both approaches. The primary reason is a significant reduction in Volta global memory transactions (a 73% improvement in global memory efficiency for SWiC) compared to Pascal. As per Nvidia specifications for Volta [35], up to 95% increase in global memory efficiency compared to earlier architectures is expected. Our roofline analysis as shown in Fig. 13 indicates that our application is memory bound and SWiC outperforms 19P in terms of achieved performance and data reuse for any

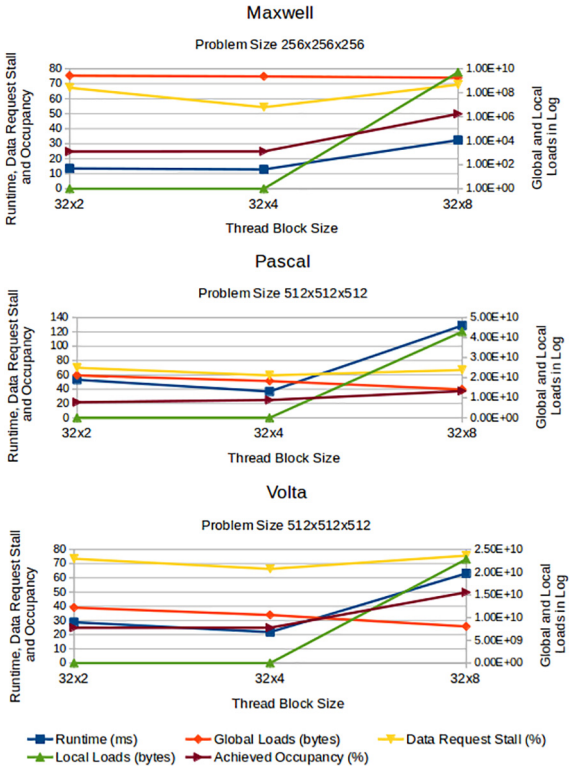

Fig. 11. Trends across different thread block sizes for SWiC on Maxwell and Volta for grid size of 128^3 and 512^3, respectively.

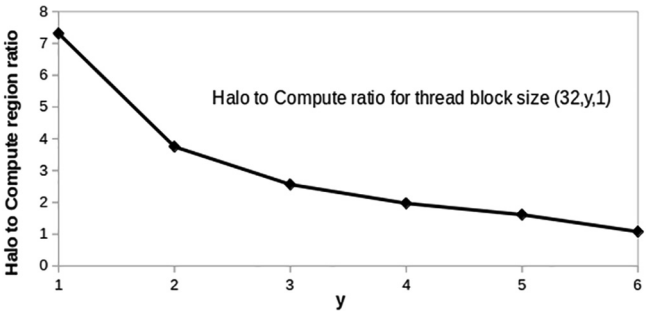

Fig. 12. Number of halos to compute grid point ratio for a thread block of size (32,y,1).

given problem size and architecture. We also measured single GPU scaling behavior for both SWiC and 19P on Pascal and Volta as shown in Fig. 14 (top). We found that the proposed approach is not only faster but also scales better as the problem size increases. Furthermore, we also observed that the speedup advantage of SWiC over 19P increases with the problem size, as shown in Fig. 14 (bottom).

5.3. SWiC applied to small and simple stencils

In some scientific applications, a smaller stencil may suffice. We investigated the application performance with smaller stencil sizes (19-point and 37-point), requiring one and two axis-aligned and non-axis-aligned neighbor grid points, respectively. The results of the experiment are shown in Table 5. We performed measurements on Maxwell, Pascal and Volta. We observed that, even for smaller stencil sizes, SWiC achieves a speedup of 1.51–1.7× to that of 19P. In order to compare with the earlier approaches using 2D caching with simple 19P stencil, we further tested SWiC without the input register queue by ignoring the stress tensor term in the Navier–Stokes equation. In this case, the stencil is simplified to a 19-point stencil as in 1(b) for direct comparison with multiple earlier works [2–5]. The results are shown in Table 6. The SWiC register usage is at the same level as the prior approach with 2D caching without any loss in performance.

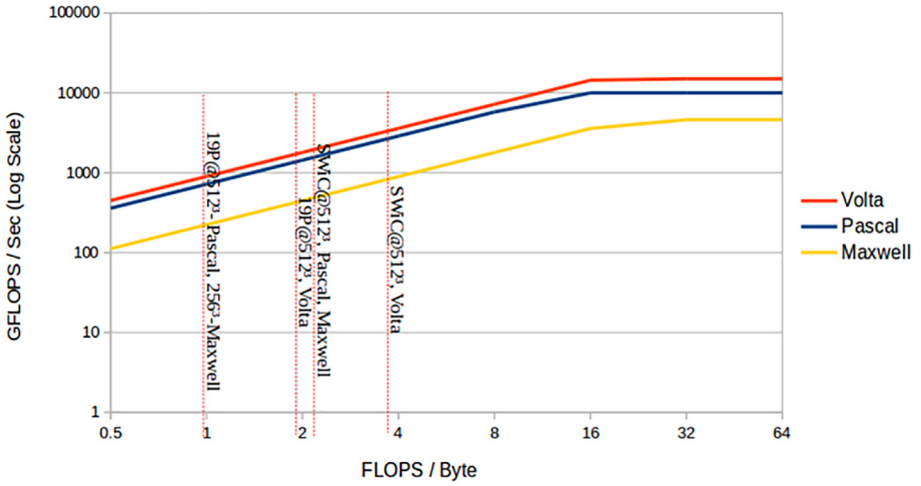

Fig. 13. Roofline analysis indicates that all the variants are memory bound, whereas SWiC outperforms 19P in terms of achieved performance and data reuse for any given problem size and architecture.

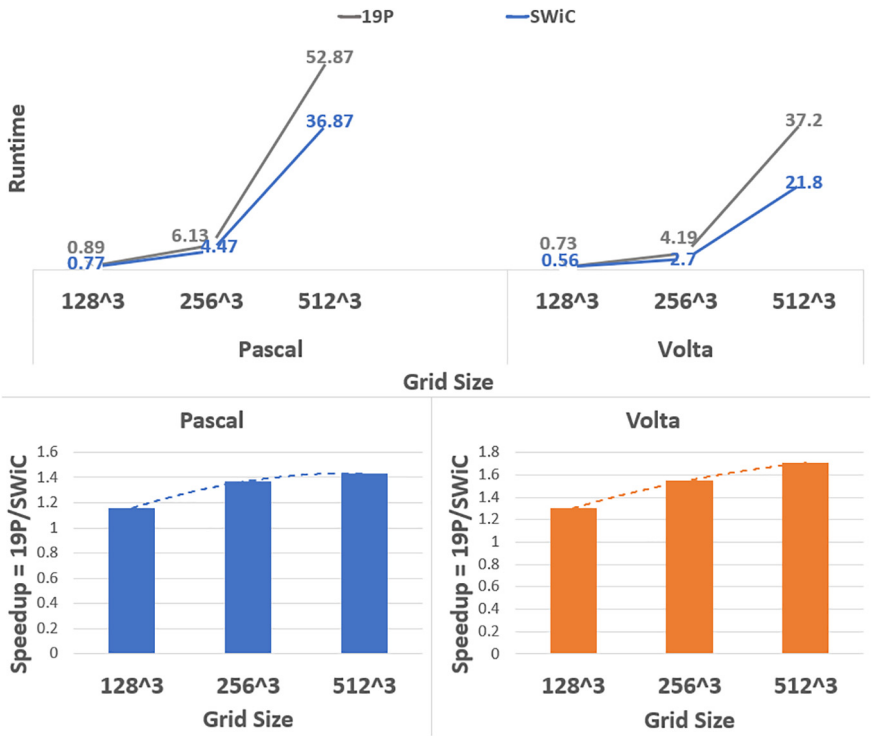

Fig. 14. SWiC vs 19P comparison on Pascal and Volta for different grid sizes.

Table 5
Runtime per Integration Step in ms on Maxwell with Grid Size 256^3 and Pascal and Volta with Grid Size 512^3 for Smaller Stencils Containing Both Axis- and Non-axis-Aligned Grid Points

	37-point stencil			19-point stencil		
	Method		Gain	Method		Gain
	19P	SWiC		19P	SWiC	
Maxwell	18.87	12.53	1.51	15.34	9.67	1.6
Pascal	43.2	34.1	1.26	67.9	47.3	1.43
Volta	30	20.1	1.49	25.1	14.7	1.7

Table 6
Comparison on Volta with Grid Size 512^3.

	Prior Approach with 2D Caching	SWiC with Less Register Usage
Runtime ms	22.8	21.3
Occupancy(%)	49.7	49.8
Registers(%)	64	64

5.4. Scaling

For the scaling experiment, we use four nodes from the NCSA HAL cluster [36]. Each node has two sockets, each socket with a 20-core IBM POWER9 CPU @ 2.4 GHz and 256 GB DDR4. In each socket, there are two NVIDIA V100 GPUs. GPUs on the same socket communicate over NVLINK 2.0. For GPU-GPU communication across sockets and nodes, our cluster uses 2-Port EDR 100 Gb/s IB ConnectX-5 Adapter. Node communication is performed using OpenMPI 4.0. On the HAL cluster, SWiC and 19P take 26.5 ms and 42.9 ms on V100, respectively. We use the single-GPU single-node running time (including the kernel and halo exchange) as our baseline for the scaling experiments. Table 7 shows a detailed breakdown of our weak scaling experiments for SWiC. Fig. 15 illustrates the weak scaling efficiency when using a single GPU and multiple GPUs per node. We make the following observations:

1. D2H, H2H, and H2D communication overheads remain unchanged as the number of nodes increases, and the number of GPUs per node is fixed.
2. D2D and H2H communication overhead increases almost linearly as the number of GPUs per node is increased. This is because the number of face halo regions linearly increases with the number of GPUs.
3. H2D communication overhead increases linearly because the host transfers the relevant face and other faces that the GPU needs to extract its edges from, which increases the amount of data transfer. For three and four GPUs per node, H2D communication overhead is superlinear because some GPUs reside on a different socket, which adds inter-socket communication overhead.
4. D2H communication overhead almost does not change as the number of GPUs per node is increased.
5. For three and four GPUs per node, the primary reason for the increased D2D communication overhead is the inter-socket communication.
6. When the compute domain is divided along both the x-axis and y-axis, D2D communication increases as each GPU is now exchanging two faces along each axis in addition to edge halos.
7. The minimum scaling efficiencies achieved by using 1, 2, 3, 4, 2x2 GPUs per node are 91%, 78%, 71%, 60%, and 54%, respectively.

We also observed that scaling behaviour does not change beyond two nodes as shown in Fig. 15. It is because the communication overhead per node is fixed given the domain size and does not depend on the number of nodes.

Table 7
Weak scaling on multi-GPU and multi-Node topologies. Kernel time with no halo exchange is 26.5 ms on HAL V100. Baseline is single-node single-GPU time (kernel + halo exchange). The table breaks down the communication overhead to: device to device (D2D), host to device (H2D), device to host (D2H), and host to host (H2H). Nodes communicate using MPI.

# of GPUs	# of Nodes	D2D	H2D	D2H	H2H	Total Time (ms)	Efficiency
1 GPU	1	0.0	0.0	0.0	0.0	27.8	100%
	2	0.0	0.4	0.7	2.2	30.3	92%
	3	0.0	0.4	0.7	2.2	30.5	91%
	4	0.0	0.4	0.7	2.2	30.6	91%
2 GPUs	1	1.1	0.0	0.0	0.0	29.3	95%
	2	1.1	0.8	1.0	4.3	35.5	78%
	3	1.1	0.8	1.0	4.3	35.3	79%
	4	1.1	0.8	1.0	4.3	35.4	79%
3 GPUs	1	2.7	0.0	0.0	0.0	31.6	88%
	2	2.3	2.0	1.1	6.4	40.3	69%
	3	2.3	2.0	1.1	6.4	39.2	71%
	4	2.3	2.0	1.1	6.4	39.3	71%
4 GPUs	1	3.2	0.0	0.0	0.0	32.3	86%
	2	3.3	5.5	1.1	8.7	46.2	60%
	3	3.3	5.5	1.1	8.7	46.3	60%
	4	3.3	5.5	1.1	8.7	46.5	60%
2×2 GPUs	1	7.1	0.0	0.0	0.0	36.2	77%
	2	7.1	5.5	1.3	8.7	51.1	54%
	3	7.1	5.5	1.3	8.7	51.2	54%
	4	7.1	5.5	1.3	8.7	51.1	54%

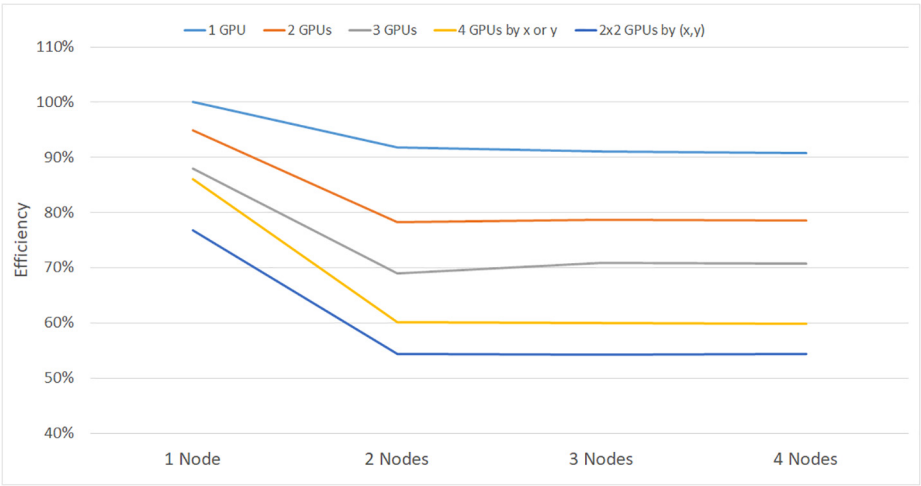

Fig. 15. Weak scaling on multi-GPU and multi-Node topologies on HAL cluster.

Table 8
Communication overhead of our single-node and multi-node implementations when unified memory is used. The kernel execution time is 26.5 ms.

# of GPUs	# of Nodes	Communication Overhead (ms)
2 GPUs	1	332
	2	356
3 GPUs	1	716
	2	743
4 GPUs	1	823
	2	823

For the unified memory model, the communication overhead is presented in Table 8. We observe that as we increase the number of GPUs per node, the communication overhead between GPUs increases prohibitively due to many expected page faults, as explained earlier.

It is also important to mention that we do not overlap communication and execution. When the compute node is divided only along the *x*-axis, the longest communication time is 46.5 ms, as shown in Table 7, which is 20 ms higher than the kernel running time. In case the compute domain of a node is divided along the *x*-axis and *y*-axis, the communication time is 51.1 ms, which is about twice the kernel running time. By overlapping communication with execution, about 50% of the communication time can be hidden to improve the scaling efficiency. Overlapping communication and execution is left to future work.

6. Conclusions

This paper presents SWiC to perform 3D stencil computation without the need for 3D caching, even for stencils with off-axis grid points. This approach reduces global memory access, data movement within a single GPU, and the number of halo exchanges. Results demonstrate faster execution than the state-of-the-art for several stencil sizes and better scaling for larger grid sizes per GPU. We also show the performance of SWiC on the latest three generations of GPU architectures. We also provide an efficient library to partition 3D stencils and automatically communicate halo regions along axes and diagonals. We performed weak scaling experiments with a recent GPU generation. We show that the proposed solution scales with the number of nodes. We also show that in most cases communication overhead is smaller than the kernel execution time and can be overlapped with the execution either completely or partially to improve scaling.

CRediT authorship contribution statement

Omer Anjum: Conceptualization, Methodology, Software, Writing-original-draft, Writing-review-editing. **Mohammad Almasri:** Software, Writing-original-draft, Writing-review-editing. **Simon Garcia de Gonzalo:** Writing-original-draft, Writing-review-editing. **Wen-mei Hwu:** Writing-original-draft, Writing-review-editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR)–a research collaboration as part of the IBM AI Horizons Network. This work utilizes resources supported by the National Science Foundation's Major Research Instrumentation program, grant #1725729, as well as the University of Illinois at Urbana-Champaign. This work is partly supported by the Center for Applications Driving Architectures (ADA) and Center for Research on Intelligent Storage and Processing-in-memory (CRISP), JUMP Centers with the prime award coming from SRC.

References

[1] E.H. Phillips, M. Fatica, Implementing the himeno benchmark with cuda on gpu clusters, IEEE International Symposium on Parallel Distributed Processing (IPDPS) 2010 (2010) 1–10.
[2] P. Micikevicius, 3d finite difference computation on gpus using cuda, in: Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, ACM, New York, NY, USA, 2009, pp. 79–84..
[3] A. Vizitiu, L. Itu, C. Niţă, C. Suciu, Optimized three-dimensional stencil computation on fermi and kepler gpus, IEEE High Performance Extreme Computing Conference (HPEC) 2014 (2014) 1–6.
[4] S. Cygert, D. Kikoła, J. Porter-Sobieraj, J. Sikorski, M. Słodkowski, Using gpus for parallel stencil computations in relativistic hydrodynamic simulation, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Waśniewski (Eds.), Parallel Processing and Applied Mathematics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 500–509.
[5] J. Pekkilä, M. Väisälä, M. Käpylä, P. Käpylä, O. Anjum, Methods for compressible fluid simulation on gpus using high-order finite differences, Comput. Phys. Commun. doi:10.1016/j.cpc.2017.03.011..
[6] Y. Zhang, F. Mueller, Auto-generation and auto-tuning of 3d stencil codes on gpu clusters, in: Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12, ACM, New York, NY, USA, 2012, pp. 155–164. doi:10.1145/2259016.2259037. doi:10.1145/2259016.2259037..
[7] A. Nguyen, N. Satish, J. Chhugani, C. Kim, P. Dubey, 3.5dd blocking optimization for stencil computations on modern cpus and gpus, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–13. doi:10.1109/SC.2010.2. url:https://doi.org/10.1109/SC.2010.2..
[8] J.A. Stratton, C. Rodrigues, I. Sung, L. Chang, N. Anssari, G. Liu, W.W. Hwu, N. Obeid, Algorithm and data optimization techniques for scaling to massively threaded systems, Computer 45 (8) (2012) 26–32, https://doi.org/10.1109/MC.2012.194.
[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008, pp. 1–12, https://doi.org/10.1109/SC.2008.5222004.
[10] M. Christen, O. Schenk, H. Burkhart, Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures, IEEE International Parallel Distributed Processing Symposium 2011 (2011) 676–687.
[11] J. Holewinski, L.-N. Pouchet, P. Sadayappan, High-performance code generation for stencil computations on gpu architectures, in: Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12, ACM, New York, NY, USA, 2012, pp. 311–320. doi:10.1145/2304576.2304619. doi:10.1145/2304576.2304619..
[12] T. Grosser, A. Cohen, P.H.J. Kelly, J. Ramanujam, P. Sadayappan, S. Verdoolaege, Split tiling for gpus: Automatic parallelization using trapezoidal tiles, in: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6, ACM, New York, NY, USA, 2013, pp. 24–31. doi:10.1145/2458523.2458526. doi:10.1145/2458523.2458526..
[13] Y. Tang, R.A. Chowdhury, B.C. Kuszmaul, C.-K. Luk, C.E. Leiserson, The pochoir stencil compiler, in: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '11, ACM, New York, NY, USA, 2011, pp. 117–128. doi:10.1145/1989493.1989508. doi:10.1145/1989493.1989508..
[14] P.S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, A. Rountev, P. Sadayappan, Resource conscious reuse-driven tiling for gpus, in: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT '16, ACM, New York, NY, USA, 2016, pp. 99–111..
[15] P.S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, P. Sadayappan, Effective resource management for enhancing performance of 2d and 3d stencils on gpus, in: Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, GPGPU '16, ACM, New York, NY, USA, 2016, pp. 92–102. doi:10.1145/2884045.2884047. doi:10.1145/2884045.2884047..
[16] J. Skála, F. Baruffa, J. Büchner, M. Rampp, The 3D MHD code GOEMHD3 for astrophysical plasmas with large Reynolds numbers-code description, verification, and computational performance, Astron. Astrophys. 580 (2015) A48.
[17] J. Pekkilä, M.S. Väisälä, M.J. Käpylä, P.J. Käpylä, O. Anjum, Methods for compressible fluid simulation on GPUs using high-order finite differences, Comput. Phys. Commun. 217 (2017) 11–22.
[18] P. Chen, M. Wahib, S. Takizawa, R. Takano, S. Matsuoka, A versatile software systolic execution model for GPU memory-bound kernels, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–81.
[19] W.A. Wulf, S.A. McKee, Hitting the memory wall: Implications of the obvious, SIGARCH Comput. Archit. News 23 (1) (1995) 20–24. doi:10.1145/216585.216588. doi:10.1145/216585.216588.
[20] [link]. url:http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf..
[21] R. Temam, J. Lions, G. Papanicolaou, R. Rockafellar, Navier-Stokes Equations: Theory and Numerical Analysis, Studies in mathematics and its applications, Elsevier Science, 2016.
[22] [link]. url:https://developer.nvidia.com/nvidia-visual-profiler..
[23] H. Hotta, M. Rempel, T. Yokoyama, Astrophys. J. 786 (1) (2014) 24, https://doi.org/10.1088/0004-637x/786/1/24. url:https://doi.org/10.1088/0004-637x/786/1/24.
[24] A. Beresnyak, Astrophys. J. 784 (2) (2014) L20, https://doi.org/10.1088/2041-8205/784/2/l20. url:https://doi.org/10.10882041-8205/784/2/l20.
[25] I. Kulikov, Astrophys. J. Suppl. Ser. 214 (1) (2014) 12, https://doi.org/10.1088/0067-0049/214/1/12. url:https://doi.org/10.1088/0067-0049/214/1/12.
[26] E.E. Schneider, B.E. Robertson, Astrophys. J. Suppl. Ser. 217 (2) (2015) 24, https://doi.org/10.1088/0067-0049/217/2/24. url:https://doi.org/10.1088/0067-0049/217/2/24.
[27] [link]. url:http://pencil-code.nordita.org/..
[28] P.S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, P. Sadayappan, Register optimizations for stencils on gpus, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '18, ACM, New York, NY, USA, 2018, pp. 168–182. doi:10.1145/3178487.3178500. doi:10.1145/3178487.3178500..

[29] D. Jacobsen, J. Thibault, I. Senocak, An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters, in: 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, 2012.
[30] M. Sourouri, J. Langguth, F. Spiga, S.B. Baden, X. Cai, Cpu+gpu programming of stencil computations for resource-efficient use of gpu clusters, in: 2015 IEEE 18th International Conference on Computational Science and Engineering, 2015, pp. 17–26.
[31] M. Seyed, Improving communication performance through topology and congestion awareness in hpc systems, Ph.D. dissertation, PhD thesis, Queen's University, Ontario, 2017..
[32] I. Faraji, S.H. Mirsadeghi, A. Afsahi, Exploiting heterogeneity of communication channels for efficient gpu selection on multi-gpu nodes, 2017..
[33] Mohammad, O. Almasri, C. Anjum, Z. Pearson, V. Qureshi, S., R. Mailthody, J. Nagi, W.-M. Xiong, Hwu, Update on k-truss decomposition on gpu, IEEE High Performance Extreme Computing Conference..
[34] K. Reuter, F. Jenko, C. Forest, R. Bayliss, A parallel implementation of an mhd code for the simulation of mechanically driven, turbulent dynamos in spherical geometry, Comput. Phys. Commun. 179 (2008) 245–249, https://doi.org/10.1016/j.cpc.2008.02.011.
[35] [link]. url:http://docs.nvidia.com/cuda/volta-tuning-guide/index.html..
[36] (Mar 2019). [link]. url:https://wiki.ncsa.illinois.edu/display/ISL20/HAL cluster..