2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems

An Efficient GPU Implementation Technique for Higher-Order 3D Stencils

Omer Anjum, Garcia de Gonzalo Simon, Mert Hidayetoglu, Wen-Mei Hwu
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, USA
Email: oanjum@illinois.edu, grcdgnz2@illinois.edu, hidayet2@illinois.edu, w-hwu@illinois.edu

***Abstract*—Stencils are a family of widely used computational patterns that play a critical role in various scientific and engineering applications. Stencil computations are known to be memory-bandwidth bound, thus a number of different techniques and algorithms that optimizes memory bandwidth usage have been proposed. However, existing techniques fall short in addressing the needs of large stencils, particularly more advanced stencil patterns involving non-axis aligned grid points. To handle non-axis aligned grid points, existing methods either use 3D caching or 2D caching schemes with more than one pass over the stencil per iteration, which suffers from the high intensity of memory accesses. The large number of memory accesses in these methods hinder the available performance. In this work, we present a new GPU-based implementation technique called "SWiC" that focuses on using 2D caching to efficiently implement advanced 3D stencil patterns, involving non-axis aligned grid points, and reducing global memory transactions by increased data reuse while only requiring a single pass per iteration. In contrast to the current approaches that maintain input register queues, the proposed approach maintains and updates the output register queue instead. The analysis shows that SWiC achieves a significant reduction in memory transactions which translates to a significant application speedup, 1.6x to 5.76x, when compared to the current state-of-the-art GPU stencil implementation. "SWiC" was evaluated across the latest three Nvidia GPU architectures as of the writing of this paper, as well as various stencil patterns and sizes. We also show that "SWiC" does not suffer from performance penalties when applied to simpler 3D stencils without non-axis aligned grid points, covering a wide application range. When running on a multi-node setting, we study the scaling efficiency of SWiC and show that it is able to achieve a weak scaling efficiency of about 96%.**

I. INTRODUCTION

Stencil computation is widely used in high performance computing simulations to solve partial differential equations at large-scale that characterize and predict physical quantities such as heat, sound, velocity, pressure, density, elasticity, magnetohydrodynamics, electromagnetism, and electrodynamics. Applications that perform such characterization and prediction include earth weather prediction, space weather prediction, acoustics, star (e.g. the Sun) simulations, geomagnetic field simulations, etc.

In order to update a point in a grid space, "stencil" determines neighboring grid points in 2D or 3D in the same grid space to be used for the update. As an example consider Fig, 2(a), where the point at the center of the cube uses all the first-order neighbouring points in the cube to update itself. The same stencil sweeps over the whole grid to make updates at every point in the grid. Depending on the application

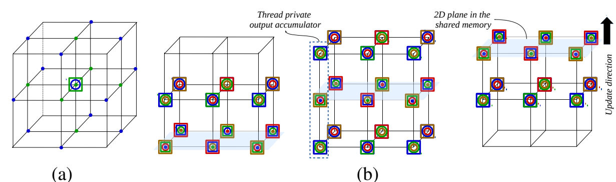

Fig. 1. An overview of the approach with an order-1 stencil. In actual implementation stencil has mirror image around the front side coming out of the page (a) With input register queue 3D caching is required to update the point at the center since the blue diagonal points other than those in the center 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.

requirement, stencils of different order in two dimensions (2D) and three dimensions (3D) are used. The order of a stencil specifies the number of neighboring grid points required for updating each grid point. For some applications, a lower-order finite-difference method (FDM) [1] with small stencil size (requiring one or two axis aligned neighbors) as shown in Fig. 2(b) might suffice. However, in order to accurately simulate more challenging systems such as the dynamics of turbulent fluids, higher-order stencils with a larger number of both axis and non-axis aligned grid points, i.e., points along diagonals in each plane, are often required, as shown in Fig. 2(a). The red points are the non-axis grid points along diagonals. Such a stencil has also been used to develop the Himeno Benchmark [22]. Stencil kernel optimizations for GPU has been widely studied. A range of different tiling techniques through automatic code generation, compiler optimizations, and manual-tuning, in spatial domain [18], [19], [20], [6], [21], [22], [23], [7], [8] and time domain [24], [25], [26], [27], [28], [29] have been proposed, to increase data reuse and decrease memory bandwidth consumption. For spatial tiling, 3D stencils are particularly challenging due to their very large tile sizes that require a large amount of on-chip shared memory to be effective. To overcome this difficulty, prior work resort to 2D caching while streaming or stepping through the third dimension[18]. However, as in shown in Fig. 1(a) for 3D stencils with off-axis grid points in blue color, not in the same plane as the grid point to be updated, are not visible to a thread. In order to make those visible prior approaches need to fallback to 3D caching in shared memory. Unfortunately, 3D caching increases shared memory usage and results in poor GPU occupancy, especially for high-order stencils [19], [21], [23], [6]. To increase GPU occupancy, a multi-pass approach

978-1-7281-2058-4/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2019.00086

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.

with 2D caching has been proposed [6], which suffers with higher number of global memory transactions. In this paper we take a different approach from previous work by improving data reuse further in the spatial domain without requiring multiple passes per stencil iteration or 3D caching in shared memory. Instead of using input register queues, we maintain output register queues for partial updates. A 2D plane is read in to the shared memory and updates are made for the past and the future planes. Colored square and circle in Fig. 1(b) indicates contribution from axis and non-axis aligned grid points of the same color, respectively. Partial sum approaches to maturity as thread progresses to other planes until it has accessed all the required points. Our proposed approach is motivated by non-axis aligned stencil computations. However, it can be applied to 2D or 3D stencil patterns with or without non-axis aligned grid points. Optimizations that apply tiling in time domain [24], [25], [26], [27], [28], [29] are orthogonal to our proposed approach, and outside the scope of our discussion.

We call our approach as "Scatter Without Write Conflict" (SWiC), where a thread scatters the effect of an input grid point read from 2D tile in the shared memory to all the accumulators in its private output register queue. When a result in the output register queue is ready, it is sent to the global memory, releasing an accumulator for a new grid point as the thread moves to the next plane. The reuse of accumulators helps to mitigate the register pressure by bounding the size of the output register queue to $(2 \times \text{stencil-order} + 1)$. Since the accumulators are stored in the private registers of each thread, SWiC completely avoids the update conflicts of general scatter approach [7], [8]. None of the prior work considers scatter and output register queue for stencil computation. 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 increases the data reuse and decreases the memory bandwidth consumption, resulting in significant application speedup.
- The run time scales more efficiently than previous approaches as the problem size increases
- 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 multiple-node execution environment, compared to the multi-pass approach with 2D caching.

The rest of the paper is organized as follows: Section II briefly explains the constraints in GPU architectures. Section III focuses on the related work. Section IV explains the implementation details of our solution. Section V presents the results and discussion. Finally, Section VI presents our future work and conclusions.

II. GPU Implementation Constraints

A. Memory Bandwidth

Stencil applications are generally known to be bandwidth bound whereas memory bandwidth has not been increasing at the rate of increase in FLOPS from one hardware generation to the next [14], [15]. This presents a major challenge for those who seek an efficient implementation of higher-order stencils as highlighted in several other prior works in Section III. In general, any approach to reducing the effect of memory bandwidth limitation must exploit data locality/reuse. In GPUs, data reuse is mainly achieved through registers and shared memory usage. Unfortunately, an increased amount of shared memory used by each thread can significantly reduce the occupancy, or the number of threads that can be scheduled for simultaneous execution, potentially lowering sustained performance.

B. Arithmetic Intensity

Arithmetic intensity for an application is measured as the number of arithmetic operations performed for each operand it fetches. As an example, let us consider a basic Navier-Stokes equation [5], which is widely used in the fluid dynamics, with four physical quantities, 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 in Fig. 2(a), one needs to fetch (number of grid points in the stencil)$\times$(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 confirmed from actual profile data collected by the Nvidia Visual Profiler [34]. The maximal reuse factor for a 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 neighbouring grid points including itself, as shown in Fig. 2(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 partition the grid points for processing by CUDA thread blocks, neighbor thread blocks share some amount of grid point data with each other as "halos/ghost zones", 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 generation GPUs, the peak memory bandwidth is 224 GB/sec and the peak compute rate is 4,612 GFLOP/sec, leading to a required arithmetic intensity of roughly 82.4. Similarly, for Pascal P100 and Volta V100, the required arithmetic intensity to sustain their peak compute rates are (10 TFLOP/sec / 720 GB/sec) 55.56 and (15 TFLOP/sec / 900 GB/sec) 66.64, respectively. We observe a reduction in the width of the memory wall [14], [15] 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. For example, the memory technology remains largely the same from Pascal to Volta. As a result, we observe that the memory wall resumed

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.

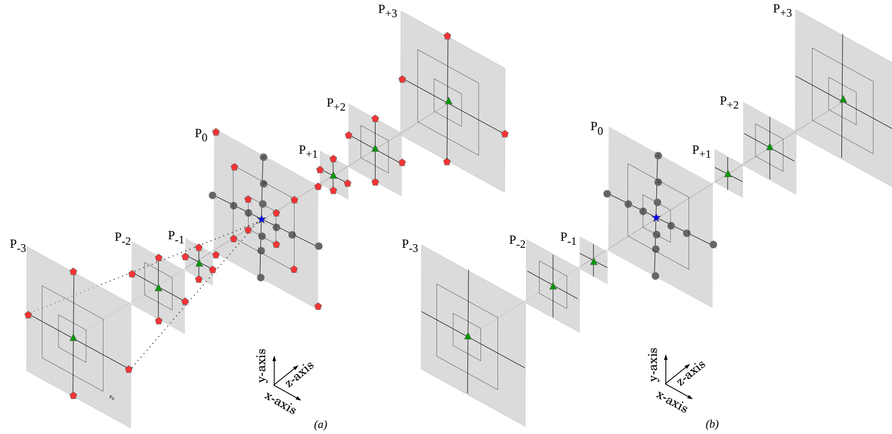

Fig. 2. A 3-D illustration of stencils (a)55-point stencil to calculate momentum with cross derivative term (b)19-point stencil to calculate momentum without cross derivative term

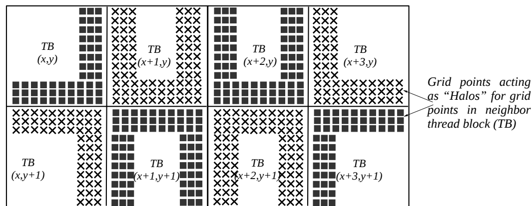

Fig. 3. Neighbour thread blocks in an xy-plane where grid points represented as small "squares" use grid points in neighbor thread-block (TB) represented as "cross" as halos and vice versa

its trend to widen, requiring more reuse per operand for full GPU compute rate utilization. In the ideal case, where the reuse factor equals to the number of grid points in the stencil, the proposed approach provides enough reuse to be compute 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.

C. Limited Storage

Another challenge arises from the large size of the stencils. Let us consider a small CUDA thread block having only one 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 3 halo grid points on each side. For a thread block with only one warp the number of grid points needed are $38 \times 7 \times 7 = 1862$. 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 $1862 \times 4(\text{physical quantities}) \times 4(\text{bytes/float}) = 29,792$ bytes. Keeping in view that the size of shared memory per streaming multiprocessor (SM) is 96 KB for the latest GPU generation, only $\lfloor 96000 \text{ bytes}/29,792 \text{ bytes} \rfloor = 3$ thread blocks would be able to get scheduled to each SM at a time. If the GPU is capable to support 64 active warps per SM, 3 thread blocks with a total of 3 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 utilizing the compute resources. Furthermore, the small number of grid points updated by each thread block severely limits the achievable arithmetic intensity ratio and thus 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 in memory bandwidth. However, the number of bytes required by each thread block also increases, which can unfortunately further reduce the occupancy per SM.

III. Related Work

Magnetohydrodynamic (MHD) simulations [11], [12], [9], [10], 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" [13] is a state-of-the-art CPU production codes for high-order finite difference codes, used for MHD simulations. Due to the limited throughput of CPU based clusters "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 [18], [19], [20] mainly considered only axis aligned grid points and are inefficient when solving for non-axis aligned grid points. Time-tiling [24], [25], [26], [27] 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 the thread block size to be reduced due to the limited amount of shared memory available in the GPU. Furthermore, time-tiling also suffers from computation overhead due to redundant calculations at halo regions. Sliding-window time-tiling [28] overcomes large shared memory requirement

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.

with 3D stencils. However, the technique 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 strategies for non-axis aligned stencils have been proposed [21], [22], [23], 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 [30], 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. For all of the above methods the major concern in achieving high GPU performance has been the large shared memory foot-print.

In order to mitigate the large memory foot-print needed for large non-axis aligned stencils, another approach decomposes the stencil into two passes and the method is indicated as 19P [6]. This method simplifies the stencil, such that instead of using a 55-point stencil, Fig. 2(a), it uses a 19-point stencil, Fig. 2(b) in each pass. However, the drawbacks of this technique are: 1) Smaller data reuse factor because of less number of points per stencil in each pass compared to a single pass approach; 2) Write back of partial results to the global memory after the first pass which increases the global memory traffic; 3) Redundant calculations are needed on the outer halos; 4) The simplification of the stencil may not be possible in every application; 5) The communication overhead per time step is high. In Table I we formulate the number of grid points that are needed to be communicated for the 19P approach, where N is the number of quantities and $N_{x,y,z}$ are grid dimensions. The term $(N+1)$ in 19P comes from the fact that in addition to store the final result it also needs to communicate the intermediate results after the first pass. For SWiC the additional term FDM-Order $\times$ $(N_x + N_y + N_z)$ 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 3 the number of halo points in a hydrodynamic simulation for 19P and SWiC are 294912×5 and 308736×4, respectively. SWiC has a 16% lower communication overhead than the 19P approach.

TABLE I. NUMBER OF GRID POINTS IN HALO EXCHANGE

Method	Number of Halo Points per Quantity
19P[*]	$(N+1) \times (N_xN_y + N_yN_z + N_zN_x)$(FDM-Order)
SWiC	$N \times$ FDM-Order $\times$ $((N_xN_y + N_yN_z + N_zN_x)+$ FDM-Order $\times$ $(N_x + N_y + N_z))$

In this work we proposed a new strategy for computing non-axis aligned stencil computation, which is also applicable to other more general stencils without performance degradation. Our approach does not requiring 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 work whether it is hand-tuned code or DSL automatic code generation.

[*] No 3D caching needed in shared memory as similar to SWiC but has more communication overhead

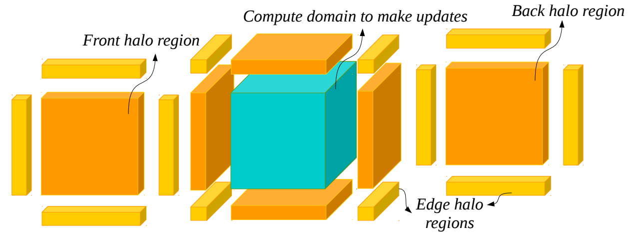

Fig. 4. An illustration of 3D grid with the compute domain at the center, surrounded by halos and edges.

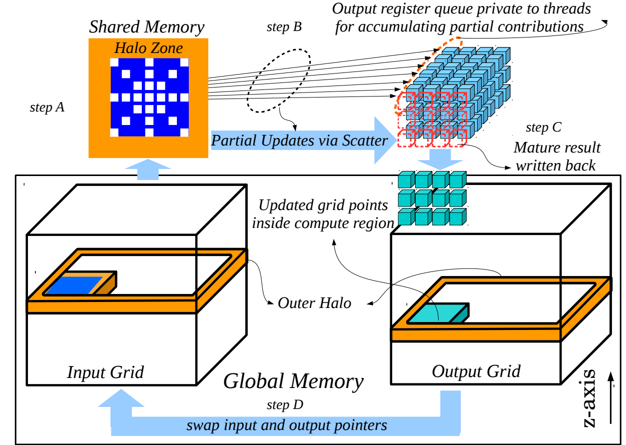

Fig. 5. An overview of application flow: Threads enter the integration loop and copy the current z-plane value to the shared memory, starting from halo region. Threads update their private accumulators with partial sums consuming the values from the shared memory. One accumulator with the final result is stored back to the global memory and is reused for the upcoming grid point at z-plane P_{c+1+hd}, where *'c'* and *'hd'* is 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* as the input for integration step *i+1*.

IV. The Design and Implementation of SWiC

An overview of the 3D grid used is shown in Fig. 4. The stencil sweeps through the compute domain shown as big cube in the center (green) surrounded by the halo regions (orange and yellow). A high level view of different steps in "SWiC" are 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, step A. White small squares in the shared memory in Fig. 5 indicates the pattern of the data points needed by a single thread, at the center of the thread block, in order to update it's private output register queue. This pattern is the same for each thread in the thread block with it's respective offset. Each of the points in the pattern contributes to the partial sum of one or more

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.

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 × 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 ∈ D & i ∌ B in parallel do
      ρ[i] = Compute_rho(ρ_s, ...)
      A[-hd:hd] = Compute_partial_gradient_of_divergence_of_velocity(u_p)
      if A_i is ready then
          u[i] = ũ[i] + A_i
      end if
      Store update to the global memory for u[i] and ρ[i]
    end for
    Exchange periodic boundary conditions for ρ and 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(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 ∈ D in parallel do
        if i ∌ B then
            ρ̃[i] = Compute_rho(ρ_s, ...)
            ũ[i] = Compute_partial_velocity(u_s, ...)
        end if
        if i ∈ B then
            u_div[i] = Compute_divergence(u_s, ...)
        end if
        Store to the global memory ũ[i], u_div[i] and ρ[i]
      end for
      Periodic_boundary_exchange(u_div)
    CUDA implicit synchronize
    Second pass kernel:
      for all i ∈ D&i ∌ B in parallel do
        Read from global memory ũ[i], u_div[i]
        u[i]=Gradient_of_divergence(u_divs, ...)+ũ[i]
      end for
    Store to the global memory u[i]
    Periodic_boundary_exchange(ρ, u)
  end for
```

of the private accumulators via "Scatter", step B. Note that the direction of the Z has been rotated so it is perpendicular to the page to make the drawing more readable. Once the computation is complete for the current z-plane, thread moves to the next plane, and adds another partial sum to it's 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 z-plane P_{c+1+hd}, where *'c'* and *'hd'* is the index of the current z-plane and halo depth, respectively. At the step D pointers for input and output grid points are swapped. Scatter is applied in the Z direction in which a thread is streaming through.

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 z-planes. Here 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 xy plane. Here $(-hd : +hd)$ is the window size of accumulator which spans over past 'hd'', current and future 'hd'' z-planes. The length of accumulator window is $2\times$stencil-order$+1$. The stencil we use in Fig. 6 is of order 2 and also has off-axis grid points as in Fig. 2(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 the Fig. 6. The status and activities of the accumulators are shown at the bottom left corner and the an 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 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 $A_{i,j}(-2)$ is the sum of the contributions made to the accumulator at different states. At state 1 it uses red and green grid points, at state 2 light blue and green grid points, at state 3 all the color grid points at plane P_c, at state 4 light blue and green grid points, and finally at state 5 red and green grid points. At the end of state 5 the content of accumulator $A_{i,j}(-2)$ is complete and ready to be written back to the global memory.

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 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.

V. 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. 2(a). We use periodic boundary exchanges after each integration step along all of the faces of the compute domain, as shown in Fig. 7. Based on earlier work [6], [21], [22], [23], [24], [25], [26], [27], [28], it is an already established fact that 3D caching in shared memory increases the shared memory foot print which reduces the number of thread blocks that can be simultaneously scheduled for an SM. As discussed earlier in Section II, only one thread block can be scheduled for an SM if the stencil as in Fig. 2(a) is cached in 3D. The only recent work which

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.

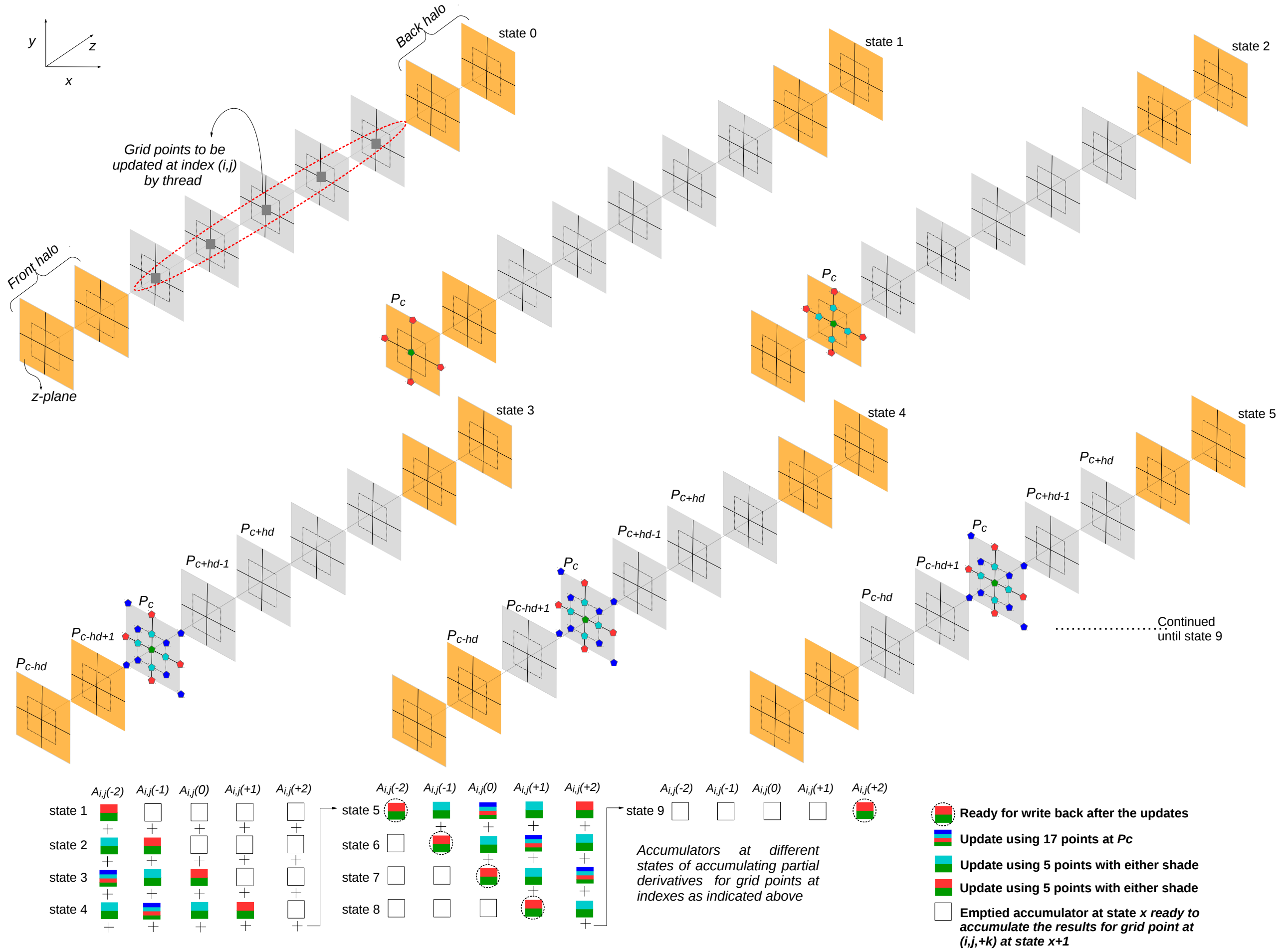

Fig. 6. A 3-D illustration of a thread activity at index (i,j) as it traverses through the z planes. How a thread updates it accumulators $A_{i,j}(-k:+k)$ is also shown at each state

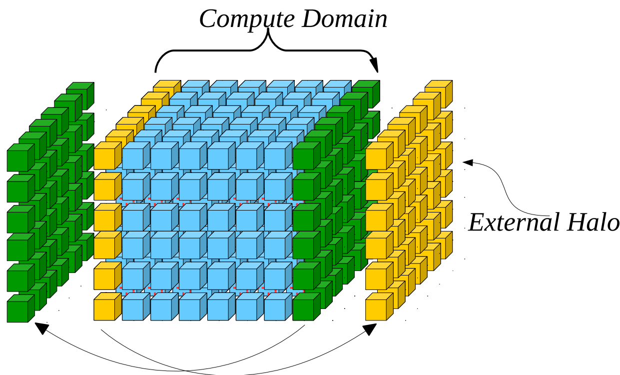

Fig. 7. An illustration of periodic boundary exchange along any face of the compute domain. The width of halo plane is one grid point.

does not require 3D caching to solve the stencil in Fig. 2(a) is 19P method [6]. It is closest to our approach, open source and part of the "Pencil Code" [13]. "Pencil Code" 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. The 19P method implements the same Navier-Stokes equation [6], to solve the 55 point stencil,depicted in Fig. 2(a). Rather than using a 55 pint stencil, the 19P approach uses two phases that only require a 19 point stencil, as in Fig. 2(b). A challenge with 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. CUDA version 9.1 with default optimization level was set to `-O3`. The input grid is initialized with constant 'ρ' and sinusoidal '$\mathbf{u}$' 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. 2(a) is used [35]. Single precision might suffice

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.

TABLE II. MEASUREMENTS OF RUN TIME SPENT IN BOUNDARY (HALO) EXCHANGE

		ρ and $\mathbf{u}$ 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

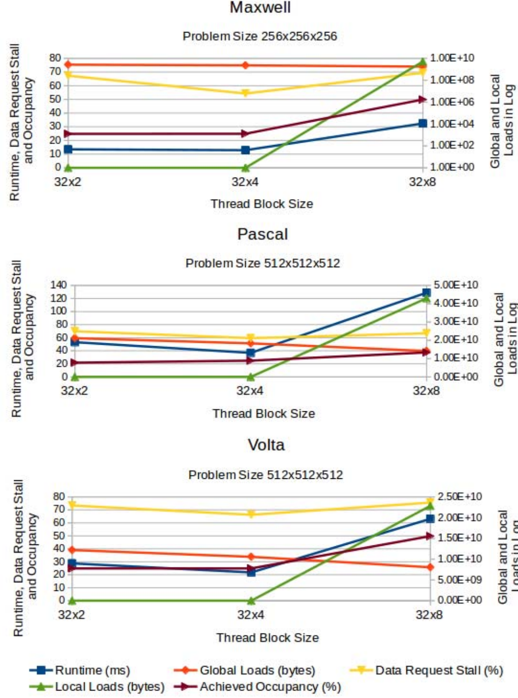

Fig. 8. Trends across different thread block sizes for SWiC on Maxwell and Volta for grid size of 128^3 and 512^3, respectively

for slow varying global phenomena in such MHD simulations. However, if some interesting local phenomena happens where high resolution is required, such as sun spot formation, the simulation for that local region might be switched to double-precision. All metrics were measured using Nvidia Visual Profiler [34]. For Pascal and Volta, we also did experiment with a larger grid size of 512^3 , taking advantage of the 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 runtime in Tables III and IV also includes the time required for the boundary exchange time as given in Table II. Those measurements are roughly the same for 19P and SWiC and also 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.

A. Thread Block Tuning

We ran CUDA thread block size and dimension tuning experiment on Maxwell, Pascal and Volta. The most optimal configuration, in terms of both run time and global memory transactions, was 32 threads (in the x dimension) by 4 threads (in the y dimension), as shown in Fig. 8. If we increase thread

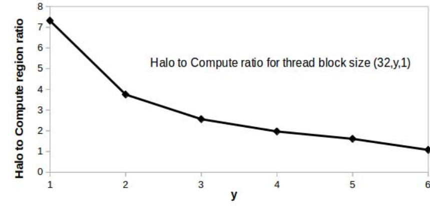

Fig. 9. Number of halo to compute grid point ratio for a thread block of size (32,y,1)

block size to 32 by 8, occupancy is increased as there is a bigger pool of warps available for latency tolerance. However, with increased number of threads per block, the compiler is given fewer registers per thread, which in turn causes register spilling. We can observe the impact of register spilling in Fig. 8 as a sudden rise in the local memory traffic. When we drop the thread block size back to 32 by 2, we increase the halo to compute grid point overhead, as shown in Fig. 9 and also the total number of grid points being used as halos. These factors contribute to increase in global memory traffic as shown in 8. We also observe a very slight degradation in run time. We found these trends to be consistent over different GPU architectures.

B. Maxwell, Pascal and Volta Runtimes With Varying Domain Sizes

Table III shows a comparison between the proposed approach and the earlier GPU implementation 19P across three different GPU architectures for grid size of 256^3. Because SWiC is designed to increase the achievable reuse, increase workload on a thread and reduce the global memory traffic, it shows a speedup of 1.6x, 1.37x and 1.55x over 19P for Maxwell, Pascal and Volta respectively. Since 19P is faster than 55P by 3.6x [6] which means that, indirectly, SWiC is faster than 55P by 5.76x. For SWiC the ideal arithmetic intensity, 72.75, is already higher than the 19P, 45.9, due to the large stencil size we use as in Table III. By doing all the calculation in one pass, avoiding extra memory references that 19P makes during the two passes, SWiC achieves higher reuse factor for any of the GPU architectures we used. Reducing global memory requests also reduces the demand for the required load/store resources which help us reducing the stalls due to data requests in SWiC compared to 19P. Further, we also observed that with the increased reuse the execution speed achieved in terms of FLOPS by the SWiC, is higher on average by a factor of 2 than the 19P approach, for all the GPU architectures.

On Nvidia Pascal P100 and Volta V100 we found SWiC to be faster than 19P for grid size 512^3 as shown in Table IV. Note that this grid size does not fit into the Maxwell memory. Similar to Maxwell we observe that SWiC has reduced global memory transactions, better reuse, better peak performance and reduced data stalls when 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 [33], up to 95% increase in global memory efficiency compared to earlier architectures is

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.

TABLE III. COMPARISON OF EXECUTION RUNTIME ON MAXWELL, PASCAL P100 AND VOLTA V100 FOR THE GRID SIZE 256^3

	Maxwell			Pascal			Volta		
	19P	**SWiC**	gain	**19P**	**SWiC**	gain	**19P**	**SWiC**	gain
Runtime(ms)	20.1	12.7	1.6	6.13	4.47	1.37	4.19	2.7	1.55
SWiC runtime gain over previous	-	-	-	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

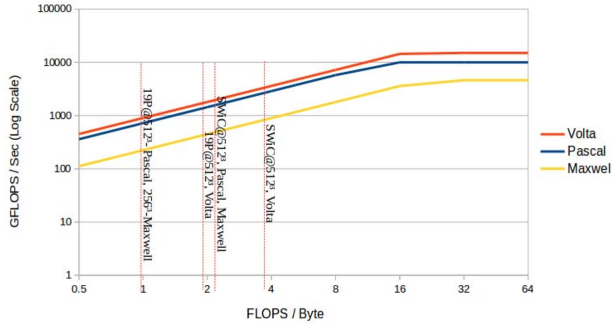

Fig. 10. Roofline analysis indicates that all the variants are memory bound whereas SWiC outperforms 19P in terms for achieved performance and data reuse for a given problem size and architecture

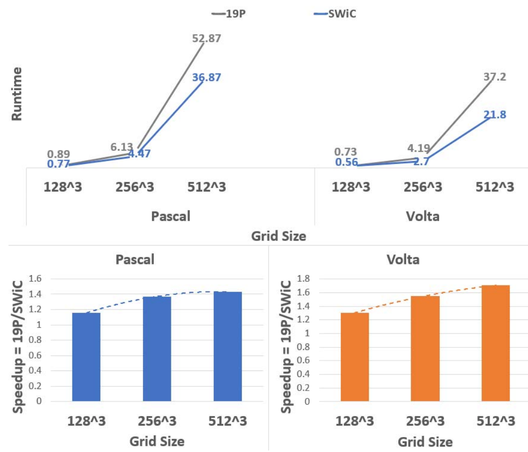

Fig. 11. SWiC vs 19P comparison on Pascal and Volta for different grid sizes

expected. For roofline analysis see Fig. 10. We also measured the single GPU scaling behaviour of both SWiC and 19P on Pascal and Volta as shown in Fig. 11(top) and found that the proposed approach is not only faster but also scales better as the problem size is increased. Furthermore we also observe that the speedup advantage of SWiC over 19P increases with the problem size, as shown in Fig. 11(bottom).

C. SWiC applied to small and simple stencils

In some scientific applications smaller stencil may suffice the requirement. We investigated 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 V. We performed measurements on Maxwell, Pascal and Volta. We observed that even for smaller stencil size SWiC gains a speed up over 19P ranging from 1.51 to 1.7 times. In order to compare with the earlier approaches using 2D caching with simple 19P stencil we further run the experiment without the input register queue by ignoring the stress tensor term in Navier-Stokes equation. In this case the input register queue is not required and the stencil is simplified to a 19 point stencil as in 2(b). We also ran the implementation as used in multiples of earlier work [18], [19], [20], [6] and compared it with the SWiC applied to 19 point stencil. The results are shown in Table VI. The SWiC register usage is at the same level as the prior approach with 2D caching without any loss in performance.

D. Weak scaling

This section addresses our preliminary work on scaling which we would like to extend as mentioned in our future work. We performed weak scaling experiment on BlueWaters [31]. The software environment is a 64-bit linux OS. We use GNU gcc 4.9.3 and Nvidia nvcc 7.5.17 compilers. Each GPU node in BW is equipped with Kepler GPU. This is our preliminary experiment on scaling as we currently do not overlap communication with the execution. The timings were recorded around the main kernel execution and MPI communication happening between the nodes. The problem size is changed by changing the number of z-planes. The maximum number of GPUs used is 64. In this weak scaling experiment, the per-GPU problem size is fixed at 256^3 and we are able to achieve around 96% efficiency for 64 GPUs in preserving the baseline run time, as shown in Fig. 12.

VI. FUTURE WORK AND CONCLUSIONS

Our next few steps include 1) Executing scaling with overlapped communication and computation as explained above, 2) Scaling experiments on more recent generations of GPUs in multi-node setup, 3) Comparison to the scaling behaviour with the prior methods should also be done.

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.

TABLE IV. COMPARISON OF EXECUTION RUNTIME ON PASCAL P100 AND VOLTA V100 FOR THE GRID SIZE 512^3

	Pascal			Volta		
	19P	**SWiC**	gain	**19P**	**SWiC**	gain
Runtime	50.9ms	36.8ms	1.38	35.1ms	21.8ms	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

TABLE V. 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		
	19P	**SWiC**	gain	**19P**	**SWiC**	gain
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 VI. 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

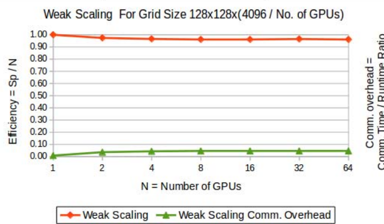

Fig. 12. Scaling for SWiC on 64 GPUs on BW[31]: For strong scaling grid size per GPU is $128 \times 128 \times (4096/\text{No. of GPUs})$ and for weak scaling domain Size per GPU remains at 256^3

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 state-of-the-art for several stencil sizes and better scaling for larger grid sizes per GPU. Results further compares performance of SWiC with 19P approach which also does not require 3D caching, on the latest three generations of GPU architectures. The approach also depicts good weak scaling performance.

REFERENCES

[1] Polyanin, A. D. (2002), Handbook of Linear Partial Differential Equations for Engineers and Scientists, Boca Raton: Chapman & Hall/CRC Press, ISBN 1-58488-299-9.

[2] H. Dursun, K. ichi Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, A multilevel parallelization framework for high-order stencil computations, in Euro-Par, 2009, pp. 642653

[3] Dursun, H., Kunaseth, M., Nomura, K., Chame, J., Lucas, R.F., Chen, C., Hall, M., Kalia, R.K., Nakano, A., and Vashishta, P. Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters. The Journal of Supercomputing, 62, 2012, 946966

[4] T. Shimokawabe, T. Aoki, N. Onodera, "High-productivity framework on GPU-rich supercomputers for operational weather prediction code ASUCA", Proceedings of the 2014 ACM/IEEE International Conference for High Performance Computing Networking Storage and Analysis ser. SC'14, pp. 1-11, 2014.

[5] Temam, Roger (2001), NavierStokes Equations, Theory and Numerical Analysis, AMS Chelsea, pp. 107112

[6] Johannes Pekkilä, Miikka S. Väisälä, Maarit J. Käpylä, Petri J. Käpylä, Omer Anjum, Methods for compressible fluid simulation on GPUs using high-order finite differences, Computer Physics Communications, Volume 217, August 2017, Pages 11-22.

[7] J. A. Stratton et al., "Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems," in Computer, vol. 45, no. 8, pp. 26-32, August 2012

[8] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08). IEEE Press, Piscataway, NJ, USA,

[9] Kulikov, I.: GPUPEGAS: A New GPU accelerated Hydrodynamic Code for Numerical Simulations of Interacting Galaxies, Astrophys. J. Suppl., 214, 12, 2014

[10] Schneider, E. E.; Robertson, B. E., Cholla: A New Massively Parallel Hydrodynamics Code For Astrophysical Simulation, arXiv:1410.4194, 2014

[11] Hotta, H.; Rempel, M.; Yokoyama, T., High resolution Calculations of the Solar Global Convection with the Reduced Speed of Sound Technique. I. The Structure of the Convection and the Magnetic Field without the Rotation, Astrophys. J. (2014a)

[12] Beresnyak, A., Spectra of Strong Magnetohydrodynamic Turbulence from High resolution Simulations, Astrophys. J. Lett., 2014

[13] (Online May 20 2018) http://pencil-code.nordita.org/

[14] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News 23, 1 (March 1995), 20-24

[15] http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[16] (Online March 15, 2018) https://www.top500.org/

[17] Antonio Ferriz-Mas, Manuel Núñez (Eds.), Advances in Nonlinear Dynamics, CRC Press (2003), pp. 269-344

[18] Paulius Micikevicius. 2009. 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, 79-84

[19] A. Vizitiu, L. Itu, C. Ni and C. Suciu, "Optimized three-dimensional stencil computation on Fermi and Kepler GPUs," 2014 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, 2014, pp. 1-6.

[20] S. Cygert, D. Kikoa, J. Porter-Sobieraj, J. Sikorski,M. Sodkowski, Using GPUs for parallel stencil computations in relativist ic hydrodynamic

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.

simulation, in: Parallel Processing and Applied Mathematics, Vol. 8384 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 500509

[21] 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, ACM, New York, NY, USA, 2012, pp. 155164.

[22] E. H. Phillips and M. Fatica, “Implementing the Himeno benchmark with CUDA on GPU clusters”, 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS), Atlanta, GA, 2010, pp. 1-10.

[23] A. Nguyen, N. Satish, J. Chhugani, C. Kim, P. Dubey, 3.5D 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, IEEE Computer Society, Washington, DC, USA, 2010

[24] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures, IPDPS 2011.

[25] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. ICS 12, pages, ACM, 2012.

[26] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles, GPGPU-6. ACM, 2013.

[27] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler, SPAA 2011. ACM.

[28] P. S. Rawat et al., ”Resource conscious reuse-driven tiling for GPUs,” PACT, Haifa, 2016.

[29] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Nol Pouchet, and P. Sadayappan. 2016. Effective resource management for enhancing performance of 2D and 3D stencils on GPUs, (GPGPU 2016)

[30] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Nol Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register optimizations for stencils on GPUs. (PPoPP ’18).

[31] NCSA, Blue Waters user portal user guide, 2012.

[32] Temam, Roger (2001), NavierStokes Equations, Theory and Numerical Analysis, AMS Chelsea, pp. 107112

[33] http://docs.nvidia.com/cuda/volta-tuning-guide/index.html

[34] (Online March 20 2018) https://developer.nvidia.com/nvidia-visual-profiler

[35] K. Reuter; F. Jenko; C. B. Forest; R. A. Bayliss,“A parallel implementation of an MHD code for the simulation of mechanically driven, turbulent dynamos in spherical geometry”, Computer Physics Communications Volume 179, Issue 4, 2008.

Authorized licensed use limited to: Stanford University. Downloaded on August 28,2024 at 06:06:49 UTC from IEEE Xplore. Restrictions apply.