applied sciences

MDPI

Article

Acceleration of a Production-Level Unstructured Grid Finite Volume CFD Code on GPU

Jian Zhang [1,2], Zhe Dai [2], Ruitian Li [2], Liang Deng [2,*], Jie Liu [1] and Naichun Zhou [2]

[1] Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha 410073, China; zhangjian@cardc.cn (J.Z.); liujie@nudt.edu.cn (J.L.)
[2] Computational Aerodynamic Institute, China Aerodynamic Research & Development Center, Mianyang 621000, China; daizhe_cardc@163.com (Z.D.); to_ruitian@126.com (R.L.); zhounc@cardc.cn (N.Z.)
* Correspondence: dengliang11@nudt.edu.cn

Abstract: Due to the complex topological relationship, poor data locality, and data racing problems in unstructured CFD computing, how to parallelize the finite volume method algorithms in shared memory to efficiently explore the hardware capabilities of many-core GPUs has become a significant challenge. Based on a production-level unstructured CFD software, three shared memory parallel programming strategies, atomic operation, colouring, and reduction were designed and implemented by deeply analysing its computing behaviour and memory access mode. Several data locality optimization methods—grid reordering, loop fusion, and multi-level memory access—were proposed. Aimed at the sequential attribute of LU-SGS solution, two methods based on cell colouring and hyperplane were implemented. All the parallel methods and optimization techniques implemented were comprehensively analysed and evaluated by the three-dimensional grid of the M6 wing and CHN-T1 aeroplane. The results show that using the Cuthill–McKee grid renumbering and loop fusion optimization techniques can improve memory access performance by 10%. The proposed reduction strategy, combined with multi-level memory access optimization, has a significant acceleration effect, speeding up the hot spot subroutine with data races three times. Compared with the serial CPU version, the overall speed-up of the GPU codes can reach 127. Compared with the parallel CPU version, the overall speed-up of the GPU codes can achieve more than thirty times the result in the same Message Passing Interface (MPI) ranks.

Keywords: unstructured-grid; CFD; shared memory parallelization; GPU; data racing

Citation: Zhang, J.; Dai, Z.; Li, R.; Deng, L.; Liu, J.; Zhou, N. Acceleration of a Production-Level Unstructured Grid Finite Volume CFD Code on GPU. *Appl. Sci.* **2023**, *13*, 6193. https://doi.org/10.3390/app13106193

Academic Editors: Pavel Lyakhov and Maxim Deryabin

Received: 4 May 2023
Revised: 12 May 2023
Accepted: 12 May 2023
Published: 18 May 2023

1. Introduction

Computational fluid dynamics (CFD) play an important role in modern industry and science, which can be used to simulate and predict the physical and chemical properties of fluid motion, helping to improve the design and manufacturing of products such as aircraft [1] and buildings [2]. The development of CFD software has always been driven by high-performance computing (HPC) technology, and the improvement of HPC computing capabilities will bring revolutionary breakthroughs in CFD applications. The computational demands of CFD are increasing at an ever-faster pace as engineers attempt to model increasingly complex flow phenomena like chemically reactive flows using more high resolution method such as Large Eddy Simulation (LES) or Direct Numerical Simulation (DNS). The LES method requires a grid size of $Re^{1.8}$, and DNS even reaches $Re^{9/4}$. For general aircraft, the Reynolds number (Re) is usually at the million level, which requires a huge amount of grid. This places higher demands on high-performance computers. In recent years, the computing power of HPC has been advancing towards the exascale level [3]. However, manufacturing constraints and power requirements have forced computer vendors to seek continued improvements through vastly higher levels of parallelism and developing more complicated memory hierarchies [4]. In particular, with the rapid development of the General Purpose Graphics Processing Unit (GPGPU), GPUs with formidable computing

power and low power consumption have played an increasingly important role in scientific computing. Kim et al. [5] proposed a method to solve the one-dimensional magnetohydrodynamics (MHD) problem by building a CFD simulator within the Hadoop Ecosystem in heterogeneous environments, mainly in GPU, and achieved sufficient performance. Nowadays, many of the fastest supercomputers in the world's Top 500 are designed based on GPU architecture. ORNL's Summit supercomputer used a CPU/GPU heterogeneous architecture with 27,648 NVIDIA Volta V100 GPUs, and its next-generation system, Frontier, was based on AMD CDNA GPUs [6]. The Leonardo supercomputer of EuroHPC/CINECA uses 14,000 Nvidia GPUs [7]. Achieving the performance of exascale computing is necessitating a paradigm shift from traditional computing advances.

There are generally two types of grids used in CFD simulations: structured grids and unstructured grids. The structured grid was constructed by elements within the domain that have the same adjacent elements (hexahedron). Unstructured grids are composed of elements of different shapes (tetrahedral, hexahedral, polyhedral, etc.). Currently, the mainstream industrial CFD software uses unstructured grids, mainly because unstructured grids are more suitable for handling complex geometric shapes, such as impeller blade [8], automobile [9], stirred tank reactor [10], urban buildings [11], and so on, than structured grids, and can adaptively adjust the size and shape of the grids to better adapt to the characteristics of fluid flow. In addition, unstructured grids can better handle complex problems such as moving bodies and multiphase fluids and have better scalability and flexibility [12]. However, unstructured grid CFD solvers encounter significant challenges when porting to GPU. The traditional parallelization strategy uses the idea of domain decomposition to divide the computational grid and uses Message Passing Interface (MPI) for communication. However, as the number of partitions increases, the overhead of copying data into (and from) communication buffers will cause parallel inefficiency. Algorithm inefficiency will also incur when the subdomains are small [13]. Under the new architecture, implementation of heterogeneous parallelism is required between CPU and GPU, i.e., utilizing multiple nodes to simultaneously process a grid system with communication by using MPI, and efficient and scalable shared memory parallel algorithms are required within GPU. However, the irregularity of the computation and indirect addressing in inner loops due to the unstructured meshes makes this a daunting challenge [14]. Current methods for handling such data races lead to reduced parallelism and suboptimal performance. Particularly for GPUs that have increasing core/thread counts, reducing data movement and exploiting memory locality is vital for gaining good performance [15]. Moreover, the inherent sequential properties of most implicit schemes make it more difficult for the algorithm to achieve parallel acceleration. For example, the lower-upper symmetric Gauss-Seidel (LU-SGS) algorithm is the most popular implicit method for solving large sparse linear equations in CFD. However, the strong data dependency during the forward and backward sweep makes it a tough challenge for shared-memory parallelization [16].

In-depth shared memory parallelization studies for production-level CFD applications are still relatively limited and are a focus of active research. Jespersen accelerated NASA's overset grid CFD solver OVERFLOW by moving a portion of the code to the GPU. However, due to the dependencies of the iteration, the Jacobian algorithm used to solve the sparse linear system was ported to the GPU instead of the original SSOR algorithm and demonstrated a 25% improvement in performance over the CPU alone [17]. Williams et al. improved the performance of the famous open-source CFD code OpenFOAM by GPU. However, they only ported the linear solver part of the code, achieving a 2.5 times improvement. Utilizing NVIDIA's Tesla K20 GPU compared to a parallel CPU implementation on ten cores of an Intel Xeon E5-2670 v2 [18], Nastac et al. presented a CUDA C++ implementation of FUN3D's thermochemical nonequilibrium flow simulation capabilities targeting GPUs. They mainly used atomic operations to address the race condition issue [19]. In summary, when targeting GPU architecture and complex production-level unstructured CFD software, comprehensive research and analysis are needed in terms of parallelization methods and memory access optimization techniques.

The motivation of this paper is to accelerate a production-level unstructured CFD software on GPU. NNW-FlowStar (FlowStar for short) is an industrial CFD software developed by China Aerodynamics Research and Development Center (CARDC) [20]. Its solver is developed based on the unstructured finite volume method and MPI parallel technology. To adapt to the new trend of HPC development, it is necessary to further develop MPI+X hierarchical parallel technology, the CUDA programming model of GPU is a kind of X. The remainder of this paper is organized as follows. Section 2 details the governing equations for the finite volume method for unstructured grids and the data racing problem due to irregular memory access patterns. Section 3 introduces the GPU parallelization implementation for face-loop kernels with data conflicts problem and the implicit method of LU-SGS. Section 4 describes some optimization methods to improve memory access efficiency. Section 5 presents the performance and efficiency results analysis for practical engineering problems. Section 6 provides the conclusion of this work and a plan for future work.

2. Unstructured Grid Finite Volume Method (FVM) CFD Solver

2.1. Cell-Centered FVM Scheme

The governing equation solved by FlowStar is the Reynolds Averaged Navier Stokes (RANS) equation, discretized by a cell-centered finite volume method for unstructured grids,

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{W}d\Omega+\oint_{\partial\Omega}(\vec{F}_c-\vec{F}_v)dS=\vec{Q} \tag{1}$$

where Ω defines the control volume, bounded by the closed surface $\partial\Omega$, $\vec{W}$ is the vector of conservative variables, $\vec{F}_c$ is the vector of convective fluxes, and $\vec{F}_v$ is the vector of viscous fluxes. $\vec{Q}$ is the source term. For cell-centred schemes, the control volumes are identical with the grid cells and the flow variables are associated with the centres of the grid cells. If considering a particular volume Ω_I, the following expression will be obtained after space discretization:

$$\frac{d\vec{W}_I}{\partial t}=-\frac{1}{\Omega_I}\left[\sum_{m=1}^{N_F}\left(\vec{F}_c-\vec{F}_v\right)_m\Delta S_m-\left(\vec{Q}\Omega\right)_I\right] \tag{2}$$

where I is the index of the control volume, N_F denotes the number of the faces of the control volume Ω_I, and ΔS_m is the area of m^{th} face.

After solving the right-hand side of Equation (2), the final residual for every cell will be obtained. Then, the solving was advanced through a temporal discretization method, approximating the final solution:

$$A\Delta\vec{W}^n=-\vec{R}_I^n \tag{3}$$

where A is the system matrix of left-hand side, n denotes to the number of iteration, and $\Delta\vec{W}^n$ is the variable's update for all cells.

2.2. Data Structure for Unstructured Grid

For structured grids, the computational space can be represented by the index i, j, k, which directly corresponds to the storage way of flow field variables in the computer. This feature can easily obtain adjacency relationships between grid entities (nodes, faces, and cells). However, unstructured grids do not have this property, and grid entities typically do not have a regular arrangement order, requiring additional data structures to store grid topology information. Unstructured grid data structures can be abstractly viewed as sets of grid entities. Physical quantities are defined corresponding to these collections, e.g., coordinates, velocities, and fluxes, and explicit connectivity data is needed to represent the adjacent info between entities. For cell-centered schemes, the numerical operation is mainly on the faces of the control volume, so it is natural to adopt a face-based data structure [21]. For example, when solving the decentralized control Equation (2), the flux at the face center

of the control volume must be provided. As shown in Figure 1, to compute the flux at f_0, pointers to the two cells (c_0 and c_1) sharing f_0 should be provided to access the flow variables associated with them.

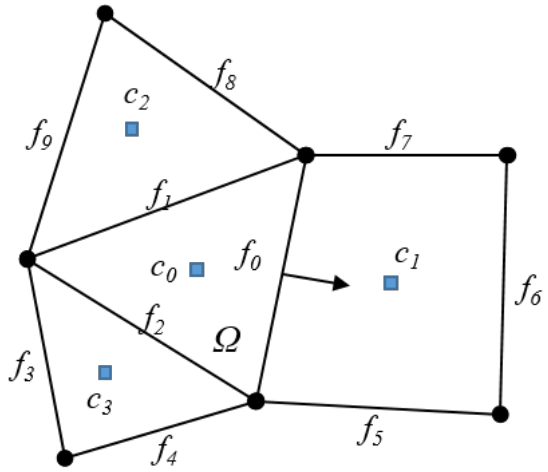


Figure 1. Data structure of cell-centred FVM scheme for unstructured grid.

2.3. Data Racing Due to Irregular Memory Access Pattern

The computations on unstructured grids are loops over a set of grid entities. If a loop over a set only writes the data defined on that set, then the loop can run in parallel. However, if there is indirect access to data, there may be situations where iterating from multiple entities simultaneously updates the same memory location value, leading to data race issues. This kind of loop with indirect memory access is common in CFD applications based on unstructured grids, such as the convective flux evaluation (Algorithm 1).

Algorithm 1 Calculation of convection flux based on face-based loop

Input: Pointers from face to cell $f2cl$, $f2cr$. Flow variables q.
Output: Right hand side residuals RHS

1: **for** $face = 0$ to $nFaces$ **do**
2: $c_0 = f2cl[face]$
3: $c_1 = f2cr[face]$
4: F_{ij} = InvicidFlux($q[c_0]$, $q[c_1]$, ...)
5: $RHS[c_0]$ -= F_{ij}
6: $RHS[c_1]$ += F_{ij}
7: **end for**

In Algorithm 1, the face-based loop follows a *gather-scatter* memory access pattern. Firstly, obtain the flow variables from a face's two adjacent cells (line 2 and line 3). Then calculate the fluxes by calling the inviscid flux-computing kernel (line 4). Finally, update RHS in cells using the computed fluxes (line 5 and line 6). Because one cell has multiple faces, simultaneous updates variables on it from different faces may lead tio data races. Take Figure 1 as an example. f_0, f_1, and f_2 belong to different threads, after completing flux calculation separately, simultaneously updating the value of c_0 will lead to data racing.

2.4. The Inherent Sequential Properties of Implicit Algorithms

The LU-SGS scheme is widely used in CFD because of its low numerical complexity and modest memory requirements. The LU-SGS scheme is based on the factorization of Equation (3) into the following form

$$(D + L)D^{-1}(D + U)\Delta\vec{W}^n = -\vec{R}_I^n \tag{4}$$

where L is the lower triangular matrix of system matrix, U is the upper triangular matrix, D is diagonal terms. Equation (4) is then solved in two steps, a forward and a backward sweep, that is,

$$\begin{aligned}(D+L)\Delta\vec{W}^{*} &= -\vec{R}_I^{n} \\ (D+U)\Delta\vec{W}^{n} &= D\Delta\vec{W}^{*}\end{aligned} \tag{5}$$

The process of LU-SGS solving is inherently serial. In solving a triangular system through a forward or backward sweep, the solution depends on pre-updated solutions whose elimination order is in the front. Take forward sweeping as an example, as shown in Algorithm 2. The outer loop corresponding to the variable i is sequential. Parallelizing the sparse dot product in the inner loop according to the variable j is also inefficient because the length of the vector involved in the dot product is typically short [22].

Algorithm 2 Forward sweeping of LU-SGS

1: **for** i = 1 to $nCells$ **do**
2: $\quad \Delta W_i = -R_i$
3: $\quad$ **for** all j in L(i) **do** ▷ L(i) : the lower neighbour cells of cell i.
4: $\quad\quad \Delta W_i = \Delta W_i - L_{i,j}\Delta W_j$
5: $\quad$ **end for**
6: $\quad \Delta W_i = \Delta W_i / D_i$
7: **end for**

3. Parallelization on GPUs

This work uses CUDA (compute unified device architecture) to port the code from CPU to CPU. CUDA is a parallel computing platform and programming model for the GPU to reduce the complexity of programming. As described in Section 2, the data racing in face-based loops and data dependency in the implicit algorithm are the two main obstacles to realizing the parallelization on GPU. This section will provide a detailed description of the solution.

3.1. Face-Loop Parallelization

3.1.1. Atomic Operations

The most direct solution is to use atomic operations, which can protect access to potentially conflicting memory locations. The advantage of atomic operation is that changes to the code are minimal to avoid data racing while maintaining acceptable parallelism. For example, implementing aggregate accumulation operation to RHS in Algorithm 1 by calling *atomicAdd* of CUDA kernel. However, when conflicts occur frequently, the scalability of the solution is limited. This is because when multiple atomic updates are applied simultaneously to the same memory location, each individual update is applied serially and in an arbitrary order, which reduces the throughput of atomic updates [23].

3.1.2. Face Colouring

Another approach to avoid a race condition is colouring the faces of the mesh. The faces of a mesh can be coloured into separate groups so that no cell possesses faces that belong to the same colour [24]. An example of a two-dimensional mesh and a face colouring strategy is shown in Figure 2.

Algorithm 3 gives the pseudocode of flux calculation based on face-colouring technology. The loops within each colour can be executed in parallel on GPU.

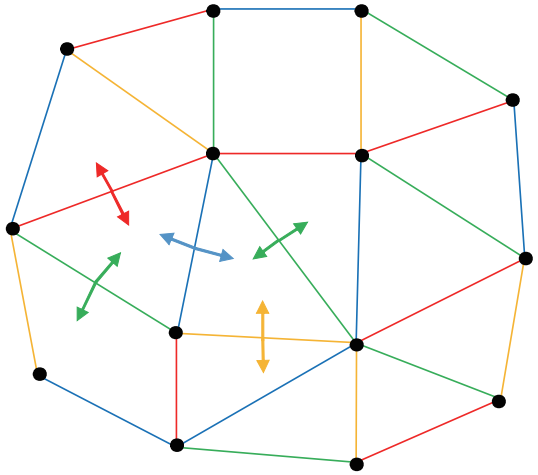


Figure 2. A two-dimensional example of face colouring. Edges represent the faces in three-dimensional case. Faces of the same colour can be executed in parallel.

Algorithm 3 Face-colouring approach for the flux computation.

Input: Pointers from face to cell $f2cl$, $f2cr$. Flow variables q. Start cell index of a colour ic.
Output: Right hand side residuals RHS

for $color$ = 0 to $nColors$ **do** ▷ Traverse all the colors.
  for $face$ = ic[$color$] to ic[$color$+1] **do** ▷ This loop can be parallelized.
    c_0 = $f2cl$[$face$]
    c_1 = $f2cr$[$face$]
    F_{ij} = InvicidFlux(q[c_0], q[c_1], ...)
    RHS[c_0] −= F_{ij}
    RHS[c_1] += F_{ij}
  end for
end for

3.1.3. Reduction

The disadvantage of face colouring is that it loops over discontinuous surfaces, resulting in poor data locality. Therefore, another parallel strategy based on reduction thought was proposed. The principle of the reduction strategy is separating the gather and scatter parts of face-based loops [13]. The former does not have data conflicts so it can be parallelized. Face quantities were stored (without scattering) during the loop over faces and were then reduced for each cell independently in a second loop over cells. As illustrated in Algorithm 4, the reduction approach makes the residual update loop of Algorithm 1 able to run in parallel.

Algorithm 4 Separating the gather and scatter parts of convective flux calculating loop.

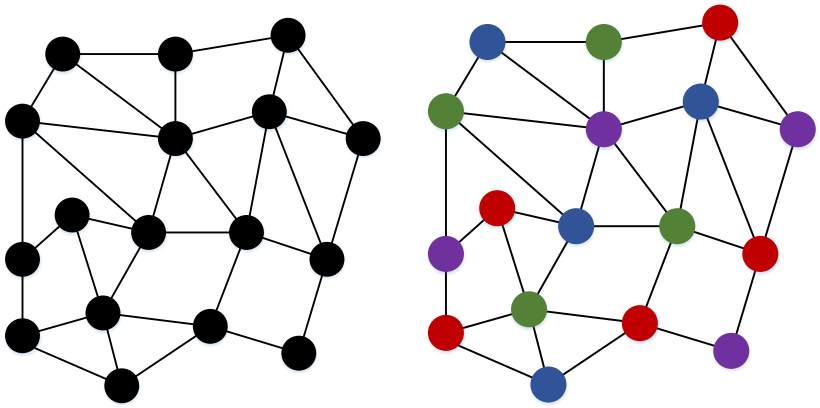
Input: Pointers from face to cell $f2cl$, $f2cr$. Flow variables q.
Output: Right hand side residuals RHS

for $face$ = 0 to $nFaces$ **do** ▷ This loop can be parallelized.
  c_0 = $f2cl$[$face$]
  c_1 = $f2cr$[$face$]
  flux[$face$] = InvicidFlux(q[c_0], q[c_1], ...) ▷ Store face quantities.
end for
for $cell$ = 0 to $nCells$ **do** ▷ This loop can be parallelized.
  for each $face$ on $cell$ **do** ▷ Reduction quantities on a cell.
    RHS[$cell$] += (face's normal point to cell) ? flux[$face$] : -flux[$face$]
  end for
end for

3.2. LU-SGS Algorithm Parallelization

3.2.1. Cell Colouring

As mentioned in Section 2.4, the updated solution appears on both the left-hand and right-hand sides when solving the linear systems. If executing line 4 of Algorithm 2 in parallel, the ordering of computation is not one way (the data race occurs for ΔW of the right), and the solution is unreproducible. Referring to the face colouring technique used in parallel flux calculation, this paper introduced the cell colouring technique [25] to solve this problem. As shown in Figure 3, transform the unstructured grid topology into an undirected graph. The vertices in the graph represent each grid cell, and the edges represent the adjacency relationship between two cells sharing the same face. All the computational cells are painted with multiple colours so the colours of neighbour cells are different from one another. Then traverse all the colours forward and backward respectively, each time performing the elimination in parallel within the same colour.



(**a**) (**b**)

Figure 3. Cell coloring for unstructured grid. (**a**) The original mesh topology. (**b**) Painted with multiple colors. Elements of the same colour can be solved in parallel.

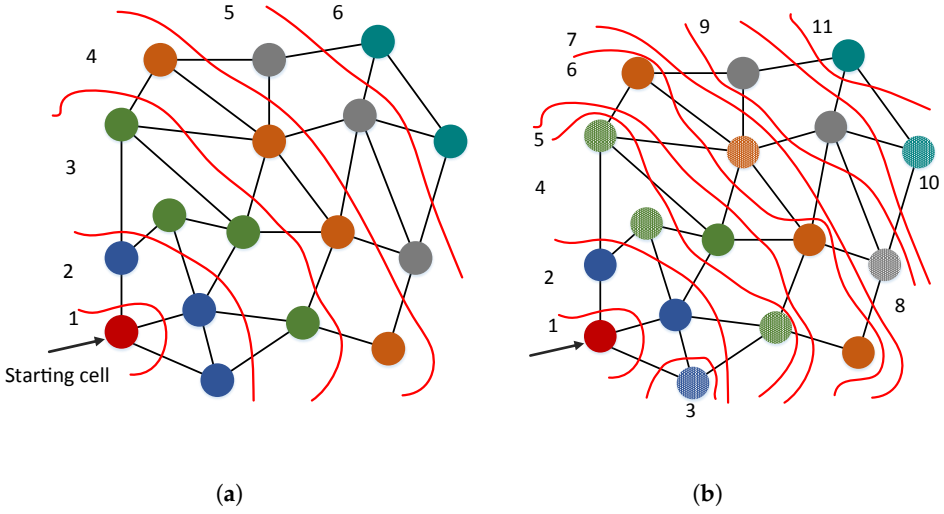
3.2.2. Hyperplane Partition

Cell colouring could solve the data dependency problem, but it degrades the implicit property of the LU-SGS algorithm. Because different colours are executed in sequential order, the value ΔW_j appearing on the right-hand side may not be the latest update. Therefore, this paper proposes a reordering method based on the hyperplane. This method was originally applied to the vectorization of the LU-SGS algorithm on unstructured grids [26].

The principle of this method is to partition and reorder all grid cells into different groups (hyperplanes), achieving the decoupling of intra-group dependencies. The majority of cells from a group have connections to cells from groups with lesser numbers as well as to cells from groups with greater numbers. Every cell from one group has no connections with other cells of the same group. As shown in Figure 4, first, partition the grid cells into preliminary hyperplanes. Select the initial grid cell as the first plane, and according to the grid topology, spread from the initial cell to the whole grid from near to far. The external grid cells with the same topological distance are divided into the same plane, that is, the distance between the grid cells in the same plane and the initial grid cell is the same. Following that, further group the cells in each hyperplane. For each hyperplane, no cells of the same group are connected.

Thus, the lower matrix in Equation (5) was computed by surrounding cells $j(i)$ whose group indexes are less than a group of cell i, $j \in L(i) : (group(j) < group(i))$, while the upper matrix is computed by surrounding cells $j(i)$ whose group index is greater than the current group of i, $j \in U(i) : (group(j) > group(i))$. The forward sweep is performed by the group numbers from 1 to the maximum and vice versa for the backward sweep. For all

cells of the same group, the computation can be done concurrently, and thus the LU-SGS algorithm can be parallelized on GPU.



(**a**) (**b**)

Figure 4. Hyperplane partition for unstructured grid. (**a**) Preliminary partition into hyperplanes. (**b**) Final groups. The numbers in the figure represent the number of the group. Elements belong to the same group are marked with same colour.

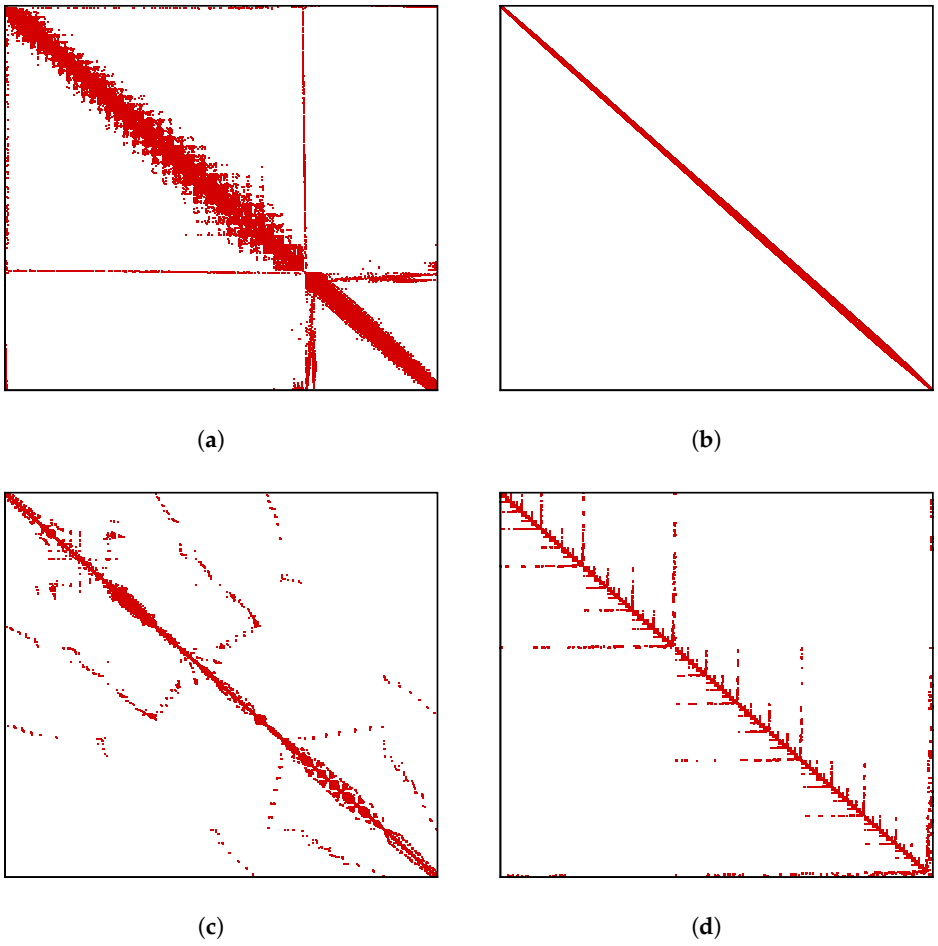
4. Data Locality Optimization

Exploiting memory locality is vital for unstructured mesh algorithms to gain good performance on GPUs. This section provides several data locality optimization strategies.

4.1. Grid Reordering

Due to the irregular shape of the unstructured grid, the numbering index span between adjacent grid elements is large. Therefore, when accessing data, the caching system may not be able to effectively cache the required data, resulting in low memory access efficiency. Reordering the grid elements is a way to improve the access pattern in face-based loops by minimising the distance between memory references when gathering and scattering data from and to pairs of cells that share a face [27]. Moreover, it does not alter the shape and distribution of the grid, so it will not affect the reliability of CFD calculations. In this work, three reordering methods, including Cuthill–McKee [28], space-filling curve [29], and graph partitioning [30], were implemented. Figure 5 shows the system matrix form obtained by reordering a grid using different algorithms, where the non-zero block of the matrix is represented by a filled rectangle. It can be seen that after reordering, the index between adjacent grid cells is closer, and the bandwidth of the matrix is significantly reduced. Among the three methods, the Cuthill–McKee method had the best result. Detailed comparisons of the three reordering methods will be provided later.

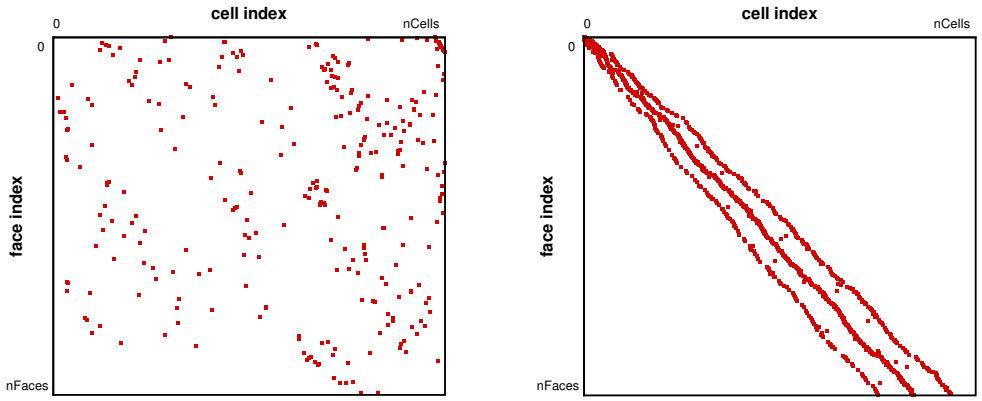
We renumbered the cells using the above bandwidth-minimization techniques then subsequently renumbered the faces according to the minimum cell number on each face, as shown in Figure 6. All of these renumbering algorithms are of complexity O(N), or at most O(Nlog(N)), and are well worth the effort [31].



(**a**) (**b**)

(**c**) (**d**)

Figure 5. The system matrix after grid reordering using different algorithms: (**a**) The original not reordered. (**b**) Cuthill–McKee. (**c**) Space-filling curve. (**d**) Graph partitioning.



(**a**) (**b**)

Figure 6. Pointer from face to cell: (**a**) The original not reordered. (**b**) After reordering the face.

4.2. Loop Fusion

In order to improve the modularity of the code, FlowStar encapsulates some subroutines that contain a large number of loops, such as loading variables, reconstruction, flux calculation, and updating RHS in the module of convective flux calculation. This will decrease the data locality. It is common in CFD code that an array is written sequentially

in a previous loop, and is likely to be read again in subsequent loops. When the length of the array is large, the data that need to be read are no longer in the cache and must be reloaded from memory. It can optimize data locality through loop fusion operations. In this way, the written values can be read in a timely manner, greatly reducing the probability of cache misses.

4.3. Multi-Level Memory Access by Shared Memory

GPU's shared memory is a type of memory that can be accessed within a block, with storage hardware located on the chip, and has fast access speed. The shared memory life-cycle management are unlike L1 caches—its usage is controlled by the user, while L1 is controlled by the system. Therefore, shared memory can be used as an explicitly managed cache to store the data that need to be read and written frequently to improve memory access efficiency. Papers [15,32] attempted to adopt this caching mechanism by loading indirectly accessed elements into GPU's shared memory based on colouring algorithms, but the performance improvement was limited, and colouring also brought further memory discontinuity. In this work, a new shared memory optimization method based on the reduction strategy (Algorithm 4) was proposed. The GPU kernel first loads the data from global memory into shared memory units. After completing the face-to-cell reduction operation, write the updated data from shared memory back to the global memory. An example is given in Figure 7, which shows the data flow of updating the RHS of Algorithm 4. For three-dimensional unstructured grids, each cell has at least four or more faces. Multiple faces-to-cell data writes on shared memory can increase the data reuse rate. After counterbalancing the overhead of loading from and write-back to global memory, this will bring additional performance improvement. Compared to the colouring scheme, this method has minimal changes to the original code and is relatively simple to implement.

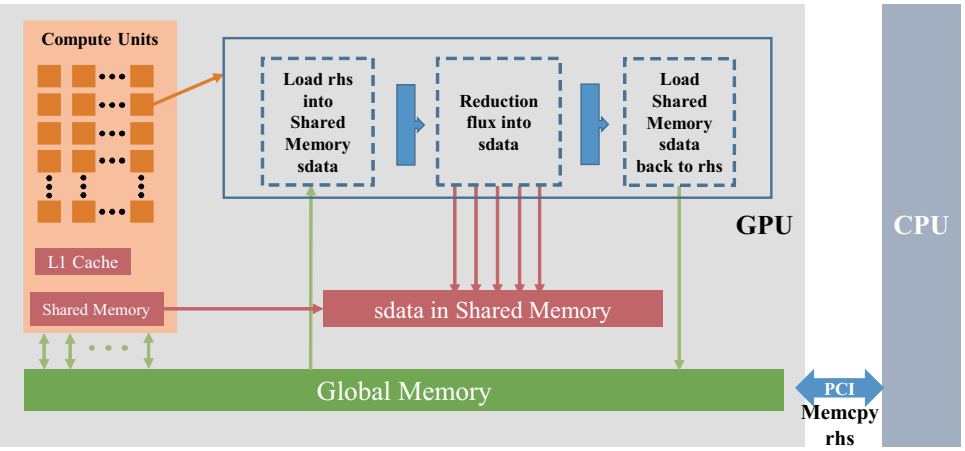
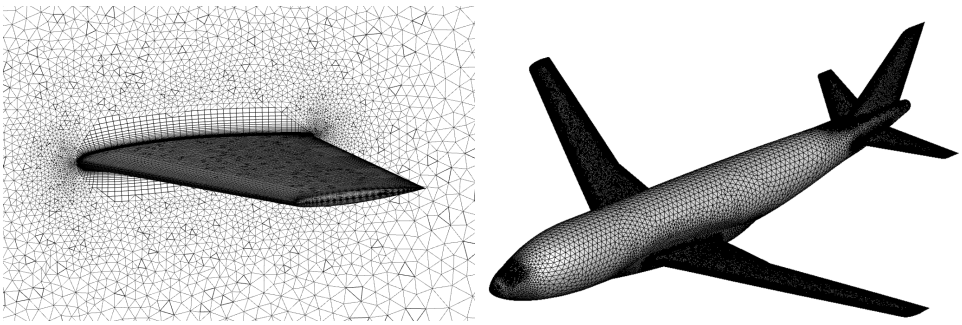


Figure 7. Data flow of using GPU shared memory as an explicitly-managed cache.

5. Performance and Discussion

5.1. Test Cases and Environment

To analyse the performance of different parallel algorithms and the effectiveness of related optimization techniques, performance tests were conducted by three-dimensional unstructured grids of the ONERA M6 wing and the CHN-T1 aircraft standard model [33], respectively. The computational mesh are shown in Figure 8. To make the image more clear, only a very coarse level grid is illustrated here. In actual calculations, the growth rate of the height of the grid in the boundary layer is approximately 1.2.



(**a**) (**b**)

Figure 8. The computational grid used for performance test. (**a**) The ONERA M6 wing. (**b**) The CHN-T1 airplane.

Different size levels of grids were supplied to validate how the performance varies with the size of the problem, as shown in Tables 1 and 2. The testing was conducted on a GPU workstation, configured with Intel(R) Xeon(R) Platinum 8268 @ 2.90 GHz and NVIDIA RTX™ A6000 48 G GPU.

Table 1. Grid information of M6 wing.

Size Level	Total Cells	Total Faces
Grid 1	123,158	55,109
Grid 2	804,367	352,639
Grid 3	3,718,474	1,321,496
Grid 4	10,834,928	4,519,009

Table 2. Grid information of CHN-T1.

Size Level	Total Cells	Total Faces
Grid 1	6,518,485	2,318,948
Grid 2	17,376,357	5,952,147

5.2. Performance of Loop Parallelization

The effectiveness of different reordering methods was compared to determine the best grid index manner. By simulating the M6 wing with the same calculation condition, the residual convergence history of different reordering methods was obtained. Figure 9 illustrates the results of Grid 1. The results show that the Cuthill–McKee method has the best performance. Compared to the original, it takes about 10% less time to reduce the residual by four orders of magnitude. The result is in line with expectations. According to Figure 5, the Cuthill–McKee method has the best data locality and the minimum matrix bandwidth after reordering. In the subsequent testing, all implementations are based on this method.

The biggest performance obstacle of parallelization on GPU is the subroutines with write data conflicts. The performance of these subroutines can best reflect the effect of different parallelization methods. Figure 10 shows the acceleration achieved by the *UpdateRHS* function in FlowStar after GPU parallelization compared to CPU serial computation. It can see that the acceleration effect of the reduction and face colouring strategy is almost the same, and both are better than atomic operations. It can also find from the results that the larger the grid size, the more significant the acceleration effect. It is because as the problem size increases, the proportion of data movement overhead between CPU and GPU decreases, and the benefits of utilizing GPU multi-core parallel acceleration will be more prominent.

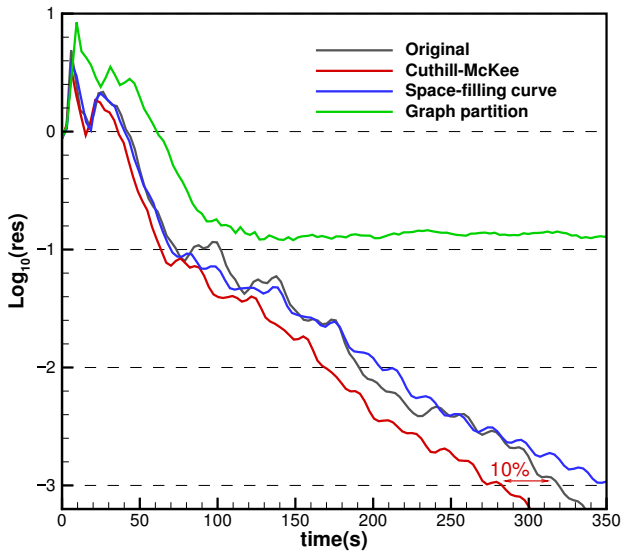


Figure 9. Comparison of convergence history using different reordering methods.

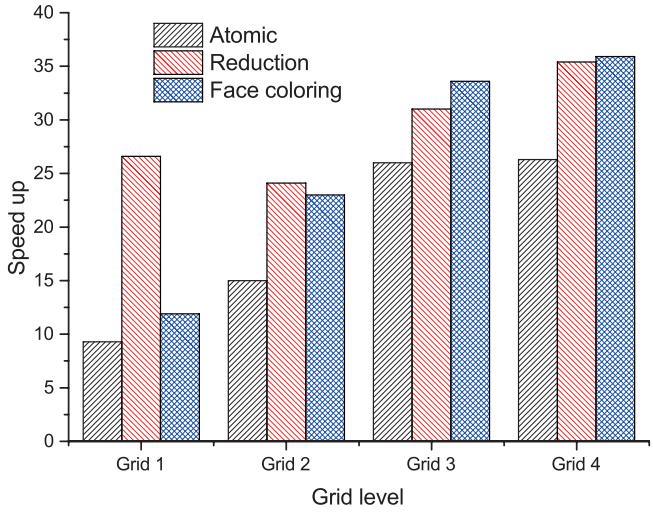


Figure 10. Speed up of UpdateRHS subroutine for different grid level of M6 wing.

Section 4.3 discusses that explicit management of shared memory can further improve memory access efficiency by caching data that need to be read and written repeatedly. Figure 11 gives the results of the speed-up ratio of the *UpdateRHS* function before and after shared memory optimization. The effect of shared memory optimization based on the face colouring method is limited (red curve in the figure). This conclusion is consistent with the Reference [32]. The group colouring method (black curve in the figure) refers to the two-layered colouring or hierarchical colouring method of Reference [15]. The performance is still poor. The reason for achieving a good acceleration ratio in the original literature is that its results considered changing variable storage from the Array of Structure (AoS) to the Structure of Array (SoA). However, the data structure of FlowStar is already SoA, so the effect is not significant. In contrast, the optimization of shared memory proposed by this work based on the reduction strategy (blue curve) is impressive. After optimization, the performance can be further improved by three times, verifying the effectiveness of the method proposed in this paper.

Furthermore, based on the above reduction strategy and shared memory optimization, this work investigated the acceleration effect of loop fusion described in Section 4.2. Table 3 gives the time consumption (s) before and after loop fusion for convective flux and

viscous flux. It is obvious that the time consumed after fusion is less, especially for the calculation of convective flux, which can be reduced by nearly half.

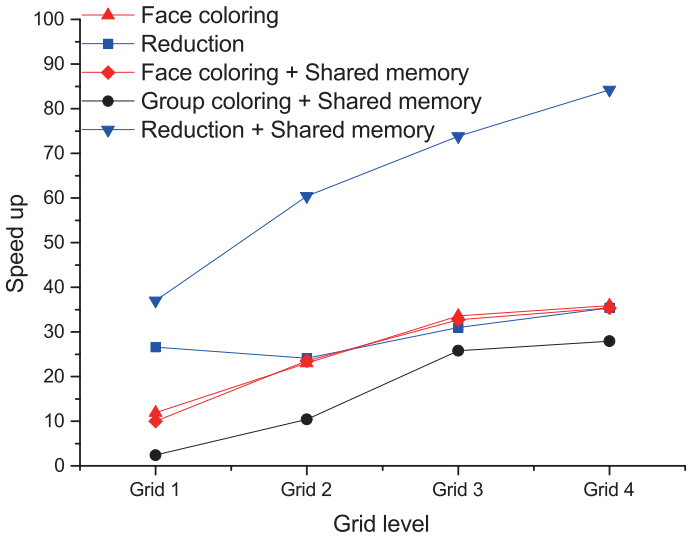


Figure 11. Comparison of speed up of UpdateRHS of simulating M6 wing before and after shared memory optimization.

Table 3. Comparison of time consumption (s) before and after loop fusion for convective flux and viscous flux.

Grid Level	Convective Flux		Viscous Flux	
	Original	Loop Fusion	Original	Loop Fusion
Grid 1	18.7	9.6	12.0	10.3
Grid 2	102.1	48.0	69.4	50.1
Grid 3	565.5	317.6	372.2	274.9
Grid 4	1622.8	861.5	1077.2	801.3

5.3. Performance of LU-SGS Parallelization

The performance of two LU-SGS parallelization methods was investigated using the M6 wing grid at various levels. Figure 12a shows the speed-up ratio of the single-card GPU calculation of colouring and hyperplane algorithms compared with the serial version on the CPU. These data reflect the concurrency level of the algorithm. Considering that the colouring or hyperplane method will bring additional overhead, it is necessary to compare it with the original LU-SGS algorithm to evaluate the acceleration benefits of the parallel algorithm. The result is shown in Figure 12b. Results show that the concurrency of the face colouring method is significantly higher than that of the hyperplane. Both algorithms require dividing the grid cells into different colours of groups, with each group executing in parallel. Thread initialization and synchronization are needed before and after parallel execution. Therefore, the more groups divided, the lower the degree of concurrency, and the higher the overall parallel overhead. The number of divided hyperplanes is large, and the number of grid cells between the planes is uneven. In extreme situations, one hyperplane has only one grid cell, resulting in high parallel synchronization overhead and low concurrency. As the scale of the grid increases, the problem of low concurrency will be alleviated, so the speed up gradually increases.

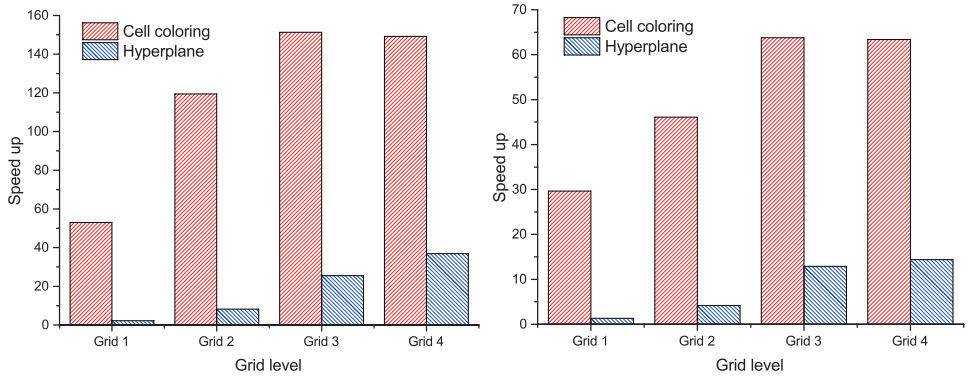


Figure 12. Speed up of LU-SGS subroutine for different grid level of M6 wing: (**a**) Compared with the same method running on CPU. (**b**) Compared with the original LU-SGS running on CPU.

5.4. Overall Performance

Finally, we tested the parallelized GPU code by coarse and fine grids of CHN-T1 and counted the performance of the acceleration ratio of each sub-module compared to the serial CPU version, as shown in Figure 13. It can see that the acceleration effect is very obvious, and the overall speed-up ratio of the program can reach 127.7. In the flux calculation part, the acceleration ratio of the subprogram can reach more than 400, demonstrating the effectiveness of the proposed parallelization method and the memory access optimization technique.

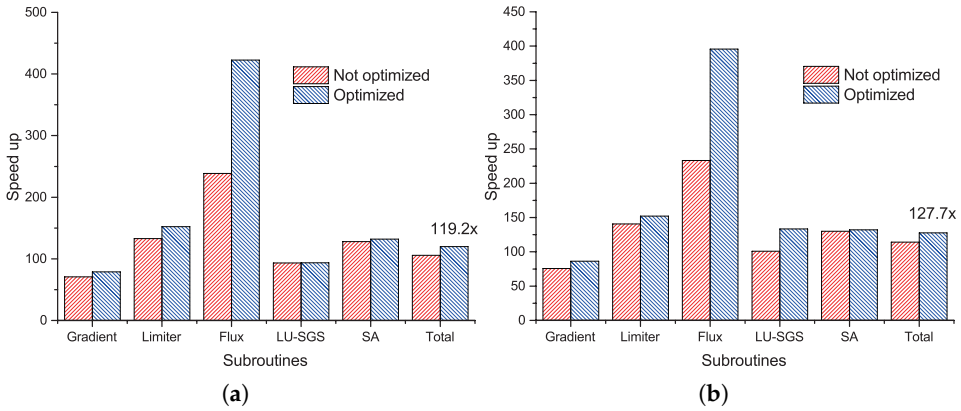
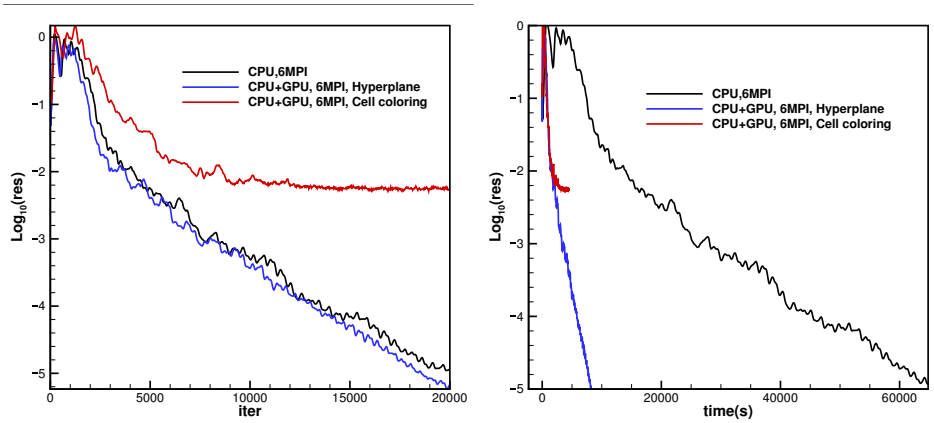


Figure 13. Overall speed-up of parallel computing the CHN-T1 on GPU: (**a**) Grid level 1. (**b**) Grid level 2.

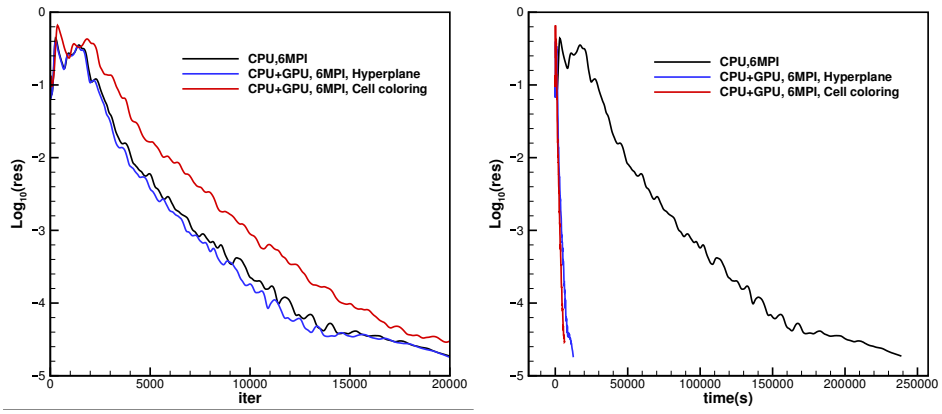
In practical CFD engineering calculations, it is usually necessary to compute until the residual converges to a specified level to obtain stable aerodynamic parameters. Next, this section evaluated the actual performance of the implemented parallel algorithm in terms of convergence speed indicators. Figures 14 and 15 show the residual convergence history of complete calculations for M6 and CHN-T1 with grid size orders of more than 10 million, respectively. This experiment uses CPU+GPU heterogeneous computing cluster with six computing nodes. It is obvious that the colouring algorithm has an impact on convergence efficiency due to the degradation of the implicit property of LU-SGS. It requires more iterative steps to reduce the residual to the same order and even does not converge for the M6. However, the concurrency of colouring algorithms is high, requiring less time for every single iteration, which can compensate for the loss of algorithm convergence efficiency. The concurrency of hyperplane algorithms is not as high as colouring, but it improves the convergence efficiency of LU-SGS because it reduces the bandwidth of the system matrix by cell reordering. By comprehensive comparison, the hyperplane method is more stable

and efficient. With the same number of processes, CPU+GPU heterogeneous computing can speed up more than 30 times compared with pure CPU computing.



(**a**) (**b**)

Figure 14. Convergence history of M6 wing. (**a**) Residual versus iterations. (**b**) Residual versus time.



(**a**) (**b**)

Figure 15. Convergence history of CHN-T1. (**a**) Residual versus iterations. (**b**) Residual versus time.

6. Conclusions

In this study, different shared memory parallel methods were implemented for unstructured CFD applications on the GPU platform. Performance was further improved through three memory access optimization techniques. Performance tests were conducted using two three-dimensional unstructured-grid cases of the M6 wing and the CHN-T1 airplane. The specific main conclusions are presented as follows:

1. Three parallel strategies based on loops have been implemented, among which the face colouring and reduction strategies are superior to atomic operations.
2. The reduction strategy combined with shared memory optimization has a significant acceleration effect. Compared to the serial CPU version, single GPU parallel computing can achieve an acceleration ratio of 127×.
3. Improving data locality has a significant effect on improving the computational performance of unstructured CFD. Using Cuthill–McKee grid renumbering and loop fusion techniques can improve the memory access performance by 10%.
4. The proposed multi-level memory-access optimization strategy can speed up the hot spot subroutine with data racing by three times.
5. The implicit algorithm part is the main obstacle to parallel scalability. Multiple colouring strategies have high concurrency, but the degrading of implicit property may affect convergence efficiency. The concurrency of hyperplane algorithms is not as high as

colouring, but it improves the convergence efficiency of LU-SGS because it reduces the bandwidth of the system matrix by cell reordering.

6. With the same number of MPI ranks, CPU+GPU heterogeneous computing can speed up more than 30 times compared with pure CPU computing to reduce the residuals to the same order of magnitude.

There is still a great deal of potential for improvement in the performance optimization of production-level CFD software GPU parallel computing, such as communication/computing overlap, mixed precision computing, etc. Further research will be conducted in future work.

Author Contributions: Conceptualization, J.Z. and J.L.; methodology, L.D.; software, R.L. and Z.D; validation, J.Z., L.D. and J.L.; formal analysis, J.Z.; investigation, L.D.; resources, Z.D.; data curation, R.L.; writing—original draft preparation, J.Z.; writing—review and editing, L.D.; visualization, Z.D. and R.L.; supervision, J.L. and N.Z.; project administration, N.Z.; funding acquisition, J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Numerical Wind Tunnel (NNW) Project of China, the Sichuan Science and Technology Program (2023YFG0152), the National Key Research and Development Program of China (2021YFB0300101).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data that support the findings of this study are available from the corresponding author upon reasonable request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Synylo, K.; Krupko, A.; Zaporozhets, O.; Makarenko, R. CFD simulation of exhaust gases jet from aircraft engine. *Energy* **2020**, *213*, 118610. [CrossRef]
2. Tan, H.; Wong, K.Y.; Othman, M.; Kek, H.Y.; Nyakuma, B.B.; Ho, W.S.; Hashim, H.; Wahab, R.A.; Sheng, D.D.C.V.; Wahab, N.H.A.; et al. Why do ventilation strategies matter in controlling infectious airborne particles? A comprehensive numerical analysis in isolation ward. *Build. Environ.* **2023**, *231*, 110048. [CrossRef]
3. Zhang, L.P.; Deng, X.G.; He, L; Li, M. He, X. The opportunity and grand challenges in computational fluid dynamics by exascale computing. *Acta Aerodyn. Sin.* **2016**, *34*, 13. [CrossRef]
4. Cary, A.; Chawner, J.; Duque, E.; Gropp, W.; Kleb, B.; Kolonay, R.; Nielsen, E.; Smith, B. Realizing the Vision of CFD in 2030. *Comput. Sci. Eng.* **2022**, *24*, 64–70.
5. Kim, M.; Lee, Y.; Park, H.; Hahn, S.; Lee, C. Computational fluid dynamics simulation based on Hadoop Ecosystem and heterogeneous computing. *Comput. Fluids* **2015**, *115*, 1–10.
6. ORNL. Available online: https://www.olcf.ornl.gov (accessed on 20 March 2023).
7. CINECA. Available online: http://www.cineca.it (accessed on 1 April 2023). [CrossRef]
8. Gomez-Flores, A.; Heyes, G.W.; Ilyas, S.; Kim, H. Effects of artificial impeller blade wear on bubble–particle interactions using CFD (k–ϵ and les), PIV, and 3D printing. *Miner. Eng.* **2022**, *186*, 107766. [CrossRef]
9. Jadhav, C.; Chorage, R. Modification in commercial bus model to overcome aerodynamic drag effect by using CFD analysis. *Results Eng.* **2020**, *6*, 100091.
10. Mittal, G.; Kikugawa, R. Computational fluid dynamics simulation of a stirred tank reactor. *Mater. Today Proc.* **2021**, *46*, 11015–11019.
11. Saddok H.; Rafik B.; Noureddine Z. A CFD Comsol model for simulating complex urban flow. *Energy Procedia* **2017**, *139*, 373–378. [CrossRef]
12. Wong, K.Y.; Tan, H.; Nyakuma, B.B.; Kamar, H.M.; Tey, W.Y.; Hashim, H.; Chiong, M.C.; Wong, S.L.; Wahab, R.A.; Mong, G.R.; et al. Effects of medical staff's turning movement on dispersion of airborne particles under large air supply diffuser during operative surgeries. *Environ. Sci. Pollut. Res.* **2020**, *29*, 82492–82511. [CrossRef]
13. Gomes, P.; Economon, T.D.; Palacios, R. Sustainable high-performance optimizations in su2. In Proceedings of the AIAA Scitech 2021 Forum, Online, 19–21 January 2021. [CrossRef]
14. Farhan, M.A.; Keyes, D.E. Optimizations of Unstructured Aerodynamics Computations for Many-core Architectures. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *29*, 2317–2332.
15. Sulyok, A.A.; Balogh, G.D.; Reguly, I.Z.; Mudalige, G.R. Locality optimized unstructured mesh algorithms on GPUs. *J. Parallel Distrib. Comput.* **2019**, *134*, 50–64. [CrossRef]

16. Li, D.; Xu, C.; Cheng, B.; Xiong, M.; Gao, X.; Deng, X. Performance modeling and optimization of parallel LU-SGS on many-core processors for 3D high-order CFD simulations. *J. Supercomput.* **2017**, *73*, 2506–2524.
17. Jespersen, D.C. Acceleration of a CFD code with a GPU. *Sci. Program.* **2010**, *18*, 193–201. [CrossRef]
18. Williams, J.; Sarofeen, C.; Shan, H.; Conley, M. An accelerated iterative linear solver with GPUs for CFD calculations of unstructured grids. *Procedia Comput. Sci.* **2016**, *80*, 1291–1300. [CrossRef]
19. Nastac, G.; Walden, A.; Nielsen, E.; Frendi, A. Implicit thermochemical nonequilibrium flow simulations on unstructured grids using gpus. In Proceedings of the AIAA Scitech 2021 Forum, Online, 19–21 January 2021. [CrossRef]
20. Chen, J.Q.; Wu, X.J.; Zhang, J.; Li, B.; Jia, H.Y.; Zhou, N.C. FlowStar: General unstructured-grid CFD software for National Numerical Windtunnel(NNW) Project. *Acta Aeronaut. Astronaut. Sin.* **2021**, *42*, 625739. [CrossRef]
21. Blazek, J. *Computational Fluid Dynamics: Principles and Applications: Third Edition*; Elsevier: Amsterdam, The Netherlands, 2015; pp. 139–142. [CrossRef]
22. Saad, Y. *Iterative Methods for Sparse Linear Systems*, 2nd ed.; Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003; pp. 369–392.
23. Stone, C.P.; Walden, A.; Zubair, M.; Nielsen, E.J. Accelerating unstructured-grid CFD algorithms on NVIDIA and AMD GPUs. In Proceedings of the IA3 2021: Workshop on Irregular Applications: Architectures and Algorithms, Held in Conjunction with SC 2021: The International Conference for High Performance Computing, Networking, Storage and Analysis, Saint Louis, MO, USA, 14–19 November 2021.
24. Giuliani, A.; Krivodonova, L. Face coloring in unstructured CFD codes. *Parallel Comput.* **2017**, *63*, 17–37.
25. Sato, Y.; Hino, T.; Ohashi, K. Parallelization of an unstructured Navier-Stokes solver using a multi-color ordering method for OpenMP. *Comput. Fluids* **2013**, *88*, 496–509.
26. Sharov, D.; Nakahashi, K. Reordering of 3-D hybrid unstructured grids for vectorized lu-sgs navier-stokes computations. In Proceedings of the 13th Computational Fluid Dynamics Conference, Snowmass Village, CO, USA, 29 June–2 July 1997; pp. 131–138.
27. Hadade, I.; Wang, F.; Carnevale, M.; di Mare, L. Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures. *Comput. Phys. Commun.* **2019**, *235*, 305–323. [CrossRef]
28. Cuthill, E.; McKee, J. Reducing the bandwidth of sparse symmetric matrices. In Proceedings of the ACM National Conference, New York, NY, USA, 26–28 August 1969; pp. 157–172. [CrossRef]
29. Fournier, Y.; Bonelle, J.; Moulinec, C.; Shang, Z.; Sunderland, A.G.; Uribe, J.C. Optimizing Code_Saturne computations on Petascale systems. *Comput. Fluids* **2011**, *45*, 103–108.
30. Oliker, L.; Heber, G.; Biswas, R. Parallel conjugate gradient: Effects of ordering strategies, programming paradigms, and architectural platforms. *Off. Sci. Tech. Inf. Tech. Rep.* **2000**. [CrossRef]
31. Rainald, L. Cache-efficient renumbering for vectorization. *Int. J. Numer. Methods Biomed. Eng.* **2010**, *26*, 628–636.
32. Zhang, X.; Sun, X.; Guo, X.H.; Du, Y.F.; Lu, Y.T.; Liu, Y. Optimizations of graph coloring method for unstructured finite volume computational fluid dynamics on GPU. *J. Natl. Univ. Def. Technol.* **2022**, *44*, 24–34. [CrossRef]
33. Yu, Y.G.; Zhou, Z.; Huang, J.T.; Mou, B.; Huang, Y.; Wang, Y.T. Aerodynamic design of a standard model CHN-T1 for single-aisle passenger aircraft. *Acta Aerodyn. Sin.* **2018**, *36*, 505–513.