Optimizing Finite Volume Method Solvers on Nvidia GPUs

Jingheng Xu, Haohuan Fu, *Member, IEEE*, Wayne Luk, *Fellow, IEEE*, Lin Gan, *Member, IEEE*, Wen Shi, Wei Xue, *Member, IEEE*, Chao Yang, *Member, IEEE*, Yong Jiang, Conghui He, and Guangwen Yang, *Member, IEEE*

Abstract—As scientific applications are increasingly ported to GPUs to benefit from both the powerful computing capacity and high throughput, accelerating explicit solvers for GPU-based finite volume methods is gaining more and more attention. In this paper, based on the detailed analysis of the FVM algorithm, we present a set of novel optimization methods, including the explicit data cache mechanism, optimal global memory loading strategy, as well as the inner-thread rescheduling method, which derives a suitable mapping from the solver algorithm to the underlying GPU hardware architecture, so as to remarkably improve the solving performance of structured mesh based FVM. We demonstrate the impact of our tuning techniques on two widely-used atmospheric dynamic kernels (3-D Euler and 2-D SWE) on five kinds of mainstream GPU platforms, and make a detailed analysis of the different tuning methodologies so as to demonstrate how to select the proper tuning strategy to different applications on various GPU platforms. Specifically, 93.9x speedup is achieved for the 3D Euler solver on Nvidia V100 over one 12-core Intel E5-2697 (v2) CPU, which is a 77 percent improvement compared with the original speedup without adopting the tuning techniques presented in this work.

Index Terms—Finite volume method, GPU, performance optimization, scientific applications

1 Introduction

In the past few decades, constrained by the physical limits such as heat dissipation and power consumption, going purely for clock speed is no longer the best strategy in processor design. As a result, the increasing of processor frequency has come to a stop. To meet the growing demand of computing power, many-core and reconfigurable architectures, such as GPUs, MICs, and FPGAs, are developed so as to promote the performance-power ratio and to keep the continuous increase of the computing power. Among these new computing architectures, GPU is one of the most popular accelerators and is being widely used nowadays in both national laboratories and industry fields. Compared with general-purpose CPU platforms, GPU is equipped with a larger number of computing units with simplified control mechanisms inside each chip, making it more efficient when dealing with throughput-oriented algorithms.

- J. Xu, H. Fu, L. Gan, W. Shi, W. Xue, and G. Yang are with the Tsinghua University, Beijing 100084, China. E-mail: {18653236889, shiwensmile}@163.com, {haohuan, xuewei, ygw}@tsinghua.edu.cn, lin.gan27@gmail.com.
- W. Luk is with the Imperial College, London SW7 2AZ, United Kingdom. E-mail: wl@doc.ic.ac.uk.
- C. Yang is with the Peking University, Beijing 100080, China. E-mail: yangchao@iscas.ac.cn.
- Y. Jiang is with the Graduate School at Shenzhen, Tsinghua University, Shenzhen, Guangdong 518055, China. E-mail: jiangy@sz.tsinghua.edu.cn.
- C. He is with the Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: heconghui@gmail.com.

Manuscript received 16 Apr. 2018; revised 23 June 2019; accepted 25 June 2019. Date of publication 2 July 2019; date of current version 8 Nov. 2019.
(Corresponding author: Lin Gan.)
Recommended for acceptance by D. Arnold.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2019.2926084

The finite volume method (FVM) is a numerical method for solving partial differential equations that calculate the values of the conserved variables averaged across the volume [1]. As one of the most commonly used numerical method, FVM is widely used in the solving approach of many scientific applications, such as atmospheric modeling [2], room acoustics modeling [3] and hydraulic erosion simulation [4]. However, as one of the hot spots in these programs, the explicit FVM solver algorithm is suffering from the low flop-to-byte ratio and irregular memory access issues, thus to remarkably decrease the utilization rate of the high-density computing devices. At the same time, with the rapid increase of computing power demands, scientific applications (including the FVM-based scientific applications) are increasingly ported to GPUs to benefit from both the powerful computing capacity and the high throughput. Thus, to boost the performance of scientific applications on GPU, an accelerated FVM solver is in urgent demand.

Aiming at providing a GPU-based FVM solver that can fully take advantage of the hardware characteristics, first we should identify the features of the FVM-based solver. The explicit FVM solver mainly contains two time-consuming parts, the state reconstruction step and the Riemann solver part. The complexity of the solver algorithm can be summarized into three folds: 1) The huge amount of memory access caused by the stencil iteration leads to a low Flop-to-Byte ratio of the state reconstruction step. 2) Long latency operations within the Riemann solver part, such as $sqrt()$ and $pow()$, introduce further complexities for achieving high instruction throughput. 3) Instruction dependency and branches are unavoidable in FVM solvers, which would result in both irregular memory access and imbalanced workload distribution among different threads.

For the underlying GPU architecture, related to the above algorithmic issues, there are also challenges that programmers need to face: 1) the limited space of GPU's on-chip fast buffer when performing memory-oriented optimizations for FVM; 2) the warp model for scheduling and executing threads when dealing with the frequent branches; 3) the performance issue caused by long latency instructions, such as $sqrt()$ and $pow()$, combined with execution dependency problems.

To resolve the above algorithmic and architecture through a suitable mapping of FVM to GPU, in this paper, we propose a set of general optimization methods, and analyze how to identify its optimal implementation choice in different situations, and evaluate their effectiveness on two real-world applications by comparing the preliminary optimized version that adopts up to six existing tuning techniques and the fully-optimized version that adopts our proposed approaches. The experimental results indicate that by employing the FVM tuning methods presented in this work, remarkable performance speedup could be achieved on almost all kinds of modern GPU platforms.

The main contributions of this paper include:

- A high-level analysis of the finite volume method and its implementation on mainstream GPU platforms, identifying state reconstruction step as the hot-spot of the explicit FVM solver; employing up to 6 tuning techniques (including coalesced access, kernel splitting and so on) as the first-round optimization to provide an optimized GPU version for further comparison and effectiveness evaluation;
- Presenting a set of novel optimization methods that can fully combine the algorithm features with the hardware architecture, by performing algorithmic modifications of the original FVM solver, including an explicit cache mechanism and optimal global memory loading strategy to reduce redundant computations and global memory access, as well as an inner-thread rescheduling method to handle work balance versus synchronization tradeoffs;
- Employing these optimization methods to typical atmospheric dynamic solvers (2D SWE solver and 3D Euler solver) so as to exam the effectiveness of these tuning techniques on all kinds of mainstream GPU platforms nowadays (Fermi, Kepler, Pascal and Volta), and making a detailed analysis of the different tuning methodologies so as to demonstrate how to select the optimal tuning strategies to different FVM solvers on various GPU platforms.

2 Related Work

As FVM-based scientific applications are becoming increasingly complicated nowadays due to the consideration of accuracy and flexibility, to meet the rapid increasing performance demand, a huge amount of work has been done to accelerate the FVM solver on different kinds of high-performance processors, including but not limited to GPU ([5], [6]), Knight Landing ([7], [8]) and FPGA ([9], [10]). Specifically, in 2016, Yang et al. accelerated the FVM-based Shallow Water Equations on Sunway TaihuLight supercomputer [2]. In this work, the authors performed systematic optimizations on different hardware levels to achieve best utilization of the heterogeneous computing units and substantial reduction of data movement cost, and successfully scaled the solver to the entire system and achieved a 7.95 PFLOPS performance in double-precision. This work won the Gordon Bell Prize of that year [13].

With the fast development of GPU nowadays, to improve the overall performance of FVM-based applications on modern GPU platforms, a couple of optimization methods have been proposed to reduce the impact of the resource conflicts between the FVM algorithm and the underlying hardware architecture ([4], [11], [12]). For instance, in work [4], an efficient FVM-based physically-based hydraulic erosion algorithm is presented and implemented completely on a GeForce 9,400 GPU to simulate the dynamic erosion process. In work [12], a parallelization of a FVM-based shallow water numerical scheme suitable for GPU architectures (GTX 580 and Tesla M2070) is presented. However, through remarkable performance benefit is able to be achieved, these works are targeting at optimizing specific scientific applications but not the FVM algorithm in general. Thus, it is hard to scale these tuning methods on different applications.

Based on the experience of specific-application performance tuning on GPU, people manage to summarize the general GPU optimization techniques towards a set of commonly used problems ([14], [16], [17], [18], [46]). Specifically, in [14], the author presents a GPU parallelization technique of the 3D finite difference computation. The method could be widely used on almost all kinds of applications adopting finite difference method, and experimental results demonstrate the great effect of this technique. Inspired by these exciting achievements, we would like to explore the possibility of providing some generalized GPU optimization methods for FVM-based solver but not a specific application.

To design a generalized GPU-based FVM solver, Langguth et al. [19] accelerate the unstructured-mesh based FVM solver on Tesla K20 GPU platform and adopt a set of hardware tuning techniques to explore the upper bound of this kind of application. Within this work, mainstream GPU tuning techniques such as shared memory, coalesced access and read-only cache are involved, and result in a better performance compared with the 16-core CPU version. It is a good attempt to optimize FVM algorithms on the GPU platform, however, only one generation of GPU is involved in this work, and the performance benefit achieved in this work is mainly contributed by the ever-increasing computing capability of the hardware, but no algorithmic modification is included. Thus, the performance results (around 40 GFlops on Tesla K20c) are not as good as we expect, and the upper bound provided in the work is no longer effective once we change the solver algorithm according to the hardware. To fully unleash the performance potential of the FVM solvers, we expect to propose a set of tuning techniques that can fully combine its algorithmic features and hardware characteristics of all kinds of mainstream GPU platforms.

In this paper, targeting at providing generalized GPU optimizations to programs that employ structured-mesh based finite volume methods, we present a set of generalized

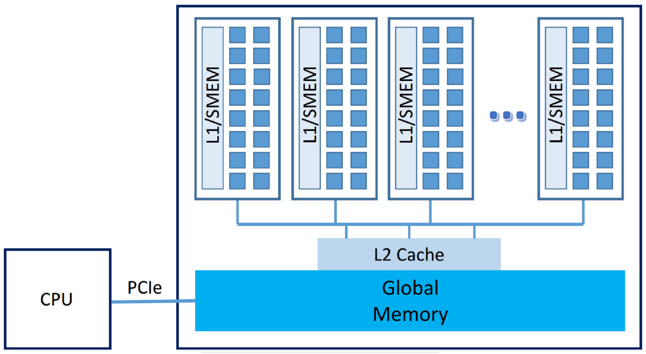


Fig. 1. A high-level hardware architecture of GPUs.

optimization methods that provide a suitable mapping between the algorithmic property and the hardware architecture, including an explicit cache mechanism and the recognition of optimal global memory loading strategy to reduce global memory access and redundant computations, as well as an inner-thread rescheduling method to handle boundary processing approaches so as to balance the computation time of each thread and the effective thread number. Compared with the conference paper [6], the modifications are mainly focused on three parts: 1) At the optimization method part, while the conference paper proposed two generalized tuning techniques to fully combine the algorithmic properties of the FVM solver and hardware architectures of Fermi and Kepler GPUs, in this journal work we employ some new tuning techniques (such as register shuffle and shared-memory partition techniques based on newer GPU platforms) and provide a set of tuning strategies, instead of one optimization choice, to fit into different FVM-based solvers on all kinds of mainstream GPU platforms (including Fermi, Kepler, Pascal and Volta) in scientific computing area; 2) At the testbed and application part, to provide a fair performance evaluation, before the adoption of tuning techniques presented in optimization method part, we first employ some hardware-based tuning methods to provide a baseline performance. To achieve this goal, while conference paper only focuses on the Fermi and Kepler oriented hardware-based tuning techniques, in this work we choose the most suitable hardware-based optimization methods for the four generations of GPU platforms. 3) At the content level, more comprehensive related work, more sufficient technique explanation and more detailed analysis are adopted in this paper, thus to expose the challenges and look into the nature of different tuning strategies. In the end, the performance comparison of the original GPU version, the basic optimized version and the fully optimized version prove the effectiveness of the tuning techniques presented in this paper, and the analysis of how to select optimal strategy would further benefit the performance tuning approach of similar applications. To the best of our knowledge, this is the first FVM tuning approach that fully combines the FVM algorithmic feature with the hardware character of GPU.

3 Background

In this section, we present the necessary background of this paper, including a brief introduction of GPU and CUDA, as well as an overview of the finite volume method and its implementation based on CUDA.

TABLE 1
Main Hardware Parameters of GPU and Contemporaneous CPU

	CPU (Intel E5-2697 v2)	GPU (Nvidia Tesla K40)
Peak Performance (Double Precision)	0.52TFlops	1.43TFlops
Clock Rate (Base Mode)	2.7GHZ	745MHZ
Physical Cores and Maximum Threads	12-cores / chip Supporting 24 threads by employing hyper-threading	15 SMXs / Chip 192 cores / SMX 2880 cores / Chip
How to Deal with Execution Dependency	Hyper-threading and out-of-order technique	Warp scheduling by context switching
On-chip Memory	Register: 1KB * 12 L1-d_cache: 32KB * 12 L2-cache: 256KB * 12	Register: 256KB * 15 L1-cache & SMEM: 64KB * 15
Off-chip Memory Size and Access Speed	Main Mem: Flexible, 768GB in maximum, 59.7 GB/s off-chip-cache(L3): 30MB	Global Mem: 12GB, 288GB/s off-chip-cache(L2): 1.5MB Read-only Memory: 64KB Constant Memory: 64KB

3.1 GPU and CUDA Programming Framework

Aiming at dealing with throughput-oriented tasks, the current generation of GPUs have thousands of processing cores that can be used for parallel computing, as well as a set of memory modules with limited size which provide the space for data storage. The basic architecture of GPU is demonstrated in Fig. 1.

Take the typical Tesla Kepler GPU as an example [21], inside each computational chip there are multiple (usually 12 to 15) Streaming Multiprocessors (SMXs), each of which is equipped with hundreds of (192 or 256) Stream Processors (SPs). Within each SMX, there are several kinds of fast-memory space including registers (256 KB or 512 KB) to store local variables of threads, constant caches for broadcasting of reads from a read-only memory, and on-chip memory (64 KB or 128 KB) which could be accessed both explicitly as shared memory or implicitly as L1 cache. In addition to the on-chip memory within each SMX, each GPU chip also introduces a L2 cache (around 1.5 MB) as well as a read-only data cache (around 64 KB).

On the hardware side, the schedulable execution unit on GPU is named as *warp* which is combined with 32 continuous threads. A warp is considered to be ready for execution only if all of its operands in each thread are ready for execution. Even there is only one operand is not ready (mainly caused by execution dependency), a process called *context switching* takes place which transfers control to another warp. This scheduling mechanism ensures the parallelism to the most extent.

Table 1 demonstrates the main hardware parameters of Nvidia Tesla K40 GPU and the contemporaneous Intel E5-2697 v2 CPU. There are two remarkable differences in perspective of the hardware architecture. First of all, though equipped with smaller on-chip memory spaces and lower clock rate, each GPU chip applies thousands of processor cores. This difference indicates that while CPUs are good at dealing with sequential codes where latency matters, GPUs

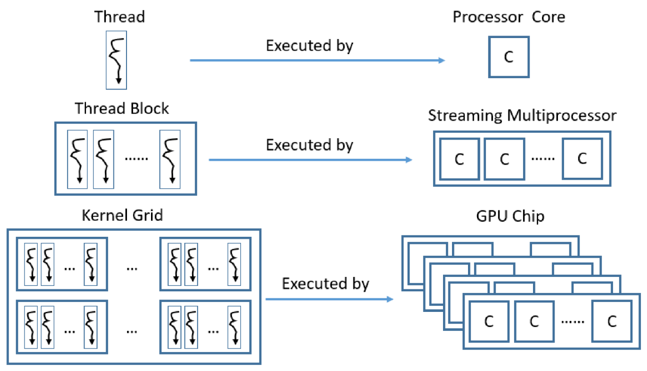


Fig. 2. Correlation between software concepts and hardware units.

could achieve high performance for parallel codes on condition that throughput meets demands. Second, as CPUs are using a few register files to decrease operation latency, GPUs use large amount of register files which have a capacity even higher than L1 and L2 caches. This feature ensures the low overhead of context switching between different threads and guaranteed the high-efficiency warp scheduling mechanism of GPU platform.

Programming on GPU was not easy until Nvidia released Compute Unified Device Architecture (CUDA), one of the most efficient and commonly used GPU programming frameworks nowadays. By using CUDA, programmers can arrange threads into thread-block which is essentially a group of threads that can coordinate among each other by synchronizing their execution streams via employing barrier instructions, and do not need to care about the execution in perspective of hardware [22]. With such features, programmers can schedule hardware resources easily by employing CUDA languages and calling CUDA libraries, rather than use the complex graphical APIs as before. Fig. 2 indicates the correlation between software concepts (e.g., CUDA thread, thread block, etc.) and hardware units of GPUs.

3.2 Finite Volume Method and Solving Approach

The finite volume method (FVM) is a common approach used in computational fluid dynamics simulations ([11], [23], [24]). Compared with other numerical algorithms such as finite difference method and finite element method, FVM has advantages in both memory usage and computation performance, especially for large-scale problems such as atmospheric modeling [25]. The core concept of FVM could be summarized as follows: first, by employing the divergence theorem, the FVM algorithm converts the volume integrals in a partial differential equation that contains a divergence portion into surface integrals; second, evaluate the portions as fluxes at the surface of each finite volume. From the description above we could demonstrate that the finite volume inside FVM refers to a small volume surrounding each node point on a mesh.

Take the general conservation law problem as an example. This problem could be represented by the following partial differential equation:

$$\frac{\partial u}{\partial t} + \nabla \cdot f(u)dv = 0, \tag{1}$$

where u represents a vector of states and f represents the corresponding flux tensor. By adopting the finite volume method, we can further divide the spatial domain into finite volumes or cells. For a certain cell i, we take the volume integral v_i over the total volume of the cell, which yields

$$\int_{v_i} \frac{\partial u}{\partial t} dv + \int_{v_i} \nabla \cdot f(u)dv = 0. \tag{2}$$

Integrating the first part over v_i and applying the Gauss divergence theorem to the second term into Equation (2), we achieve the following:

$$v_i \frac{\partial \bar{u}_i}{\partial t} + \oint_{S_i} f(u) \cdot ndS = 0, \tag{3}$$

where S_i represents the total surface area of the cell and n is a unit vector normal to the surface and pointing outward. Dividing v_i in both two sides we can derive the following form that is equivalent to Equation (3):

$$\frac{\partial \bar{u}_i}{\partial t} + \frac{1}{v_i} \oint_{S_i} f(u) \cdot ndS = 0. \tag{4}$$

To generate FVM solvers based on modern microprocessors, the Riemann solver is commonly used in estimating numerical fluxes of FVM [26]. To get the reconstruction values of the Riemann solver, we need to reconstruct the boundary values of the computational mesh based on the values of central points, so as to calculate the numerical flux. This computation step is called state reconstruction, which generally uses a piecewise linear reconstruction method as follows (Q is an intermediate variable detailed in [26])

$$\begin{aligned}\hat{Q^-}(\hat{x_{ijk}}, t) &= \frac{2-\kappa}{2} Q_{ij}(t) - \frac{1-\kappa}{4} Q_{i-1,j}(t) + \frac{1+\kappa}{4} Q_{i+1,j}(t) \\ \hat{Q^+}(\hat{x_{ijk}}, t) &= \frac{2-\kappa}{2} Q_{i+1,j}(t) + \frac{1+\kappa}{4} Q_{ij}(t) + \frac{1-\kappa}{4} Q_{i+2,j}(t),\end{aligned} \tag{5}$$

where $\kappa \in [0, 1)$. This method is widely employed in real-world applications such as Euler equations [27] and shallow water equations (SWEs) [25]. In particular, $\kappa = 0, 1/2$ and $1/3$ lead to the Fromm scheme [28], the QUICK scheme [29] and the QUICKEST scheme [29] respectively.

In consideration of the performance of FVM algorithm on modern GPUs, applying the state reconstruction step properly is of vital importance. While other parts of the FVM solver mainly consist of computational tasks, the state reconstruction step is combined with huge amount of global memory access which would cause long-latency load problems. To make things worse, the solving approach of this part takes huge amount of on-chip memory spaces on GPU and drives the useful data out of register files and caches, which remarkably slows down the FVM solver. Besides the serious cache pollution, the state reconstruction step also takes 46.5, 59.3 and 54.2 percent of the total time in 2-D SWE, 3-D regional Euler and 3-D global Euler respectively according to the experimental results based on Tesla P100, which indicates the state reconstruction step is no doubt the hot-spot of FVM-based solvers on GPU.

Algorithms 1 and 2 demonstrate the simplified implementation of 3-D and part of 2-D state reconstruction step.

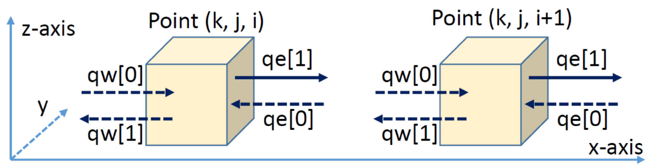


Fig. 3. Demonstration of local variables inside each mesh point. Physically, we could prove the value of $qw[0]$ of Point (k,j,i+1) equals to the value of $qe[1]$ of Point (k,j,i).

From these two algorithm we could figure out that within the state reconstruction step, each mesh element keeps 4 local variables to store fluxes in each axis, no matter in 2-D or 3-D version. Take the Kernel 1 of the 3-D version as an example, the [0] in qw[0] (or or [1] in $qw[1]$) refers to the flow that comes into (or out of) the point, while w indicates the direction to the west, as shown in Fig. 3.

Since finite volume schemes are conservative as cell averages change through the edge fluxes, we could prove that the flow that comes out of point (k,j,i) in the right direction is exactly the same as the flow comes into point (k,j,i+1) on the left side. For instance, the $qw[0]$ of point (k,j,i+1) and $qe[1]$ of point (k,j,i) shown in Fig. 3 are identical. This consideration indicates that the calculation of $qw[0]$ and $qe[1]$ is employing an identical rule on different elements from input array x, while the calculation of $qw[0]$ and $qe[0]$ are applying different rules on identical elements. These features offer us great potential for optimizing the FVMs.

3.3 Assumptions and Restrictions

The techniques discussed in this paper are based on the following assumptions about the targeted applications and platforms. First of all, since the optimization of FVM based on unstructured meshes has been presented in 2014 [20], in this paper only structure-mesh-based FVM are discussed. Second, we assume a second-order piecewise linear reconstruction is employed in the solving approach of the explicit FVM solver, since a certain reconstruction method would result in a relatively fixed algorithm format, which is beneficial for us to express our optimization methods. Third, each thread of GPU is processing one element per time step, which is a common choice to initialize huge number of threads inside GPU platforms. Finally, as the hardware architecture of AMD and Nvidia GPUs is similar but not totally the same, in this work we take Nvidia GPUs as examples to demonstrate the optimization methods.

4 Optimizations

In this section, we present a set of novel tuning techniques to improve the performance of the FVM solvers on mainstream GPU platforms. As stated earlier, the tuning techniques presented in this part would mainly focus on identifying a suitable mapping between algorithmic features and hardware characteristics, rather than directly employing hardware-based tuning techniques (such as coalesced access). A full-scale optimization approach, which adopts

Algorithm 1. Demonstration of 3D State Reconstruction Step

```
Kernel 1: $X$-axis original code:
1: $qw[0] = (c_0*x[k,j,i] + c_1*x[k,j,i-1] + c_2*x[k,j,i+1] + c_3*(x[k,j+1,i] + x[k,j-1,i] + x[k+1,j,i] + x[k-1,j,i])) / d_0$;
2: $qe[0] = (c_0*x[k,j,i] + c_1*x[k,j,i+1] + c_2*x[k,j,i-1] + c_3*(x[k,j+1,i] + x[k,j-1,i] + x[k+1,j,i] + x[k-1,j,i])) / d_0$;
3: $qe[1] = (c_0*x[k,j,i+1] + c_1*x[k,j,i] + c_2*x[k,j,i+2] + c_3*(x[k,j+1,i+1] + x[k,j-1,i+1] + x[k+1,j,i+1] + x[k-1,j,i+1])) / d_0$;
4: $qw[1] = (c_0*x[k,j,i-1] + c_1*x[k,j,i] + c_2*x[k,j,i-2] + c_3*(x[k,j+1,i-1] + x[k,j-1,i-1] + x[k+1,j,i-1] + x[k-1,j,i-1])) / d_0$;
Kernel 2: $Y$-axis original code:
1: $qs[0] = (c_0*x[k,j,i] + c_1*x[k,j-1,i] + c_2*x[k,j+1,i] + c_3*(x[k,j,i+1] + x[k,j,i-1] + x[k+1,j,i] + x[k-1,j,i])) / d_0$;
2: $qn[0] = (c_0*x[k,j,i] + c_1*x[k,j+1,i] + c_2*x[k,j-1,i] + c_3*(x[k,j,i+1] + x[k,j,i-1] + x[k+1,j,i] + x[k-1,j,i])) / d_0$;
3: $qn[1] = (c_0*x[k,j+1,i] + c_1*x[k,j,i] + c_2*x[k,j+2,i] + c_3*(x[k,j+1,i+1] + x[k,j+1,i-1] + x[k+1,j+1,i] + x[k-1,j+1,i])) / d_0$;
4: $qs[1] = (c_0*x[k,j-1,i] + c_1*x[k,j,i] + c_2*x[k,j-2,i] + c_3*(x[k,j-1,i+1] + x[k,j-1,i-1] + x[k+1,j-1,i] + x[k-1,j-1,i])) / d_0$;
Kernel 3: $Z$-axis original code:
1: $qb[0] = (c_0*x[k,j,i] + c_1*x[k-1,j,i] + c_2*x[k+1,j,i] + c_3*(x[k,j+1,i] + x[k,j-1,i] + x[k,j,i+1] + x[k,j,i-1])) / d_0$;
2: $qt[0] = (c_0*x[k,j,i] + c_1*x[k+1,j,i] + c_2*x[k-1,j,i] + c_3*(x[k,j+1,i] + x[k,j-1,i] + x[k,j,i+1] + x[k,j,i-1])) / d_0$;
3: $qt[1] = (c_0*x[k+1,j,i] + c_1*x[k,j,i] + c_2*x[k+2,j,i] + c_3*(x[k+1,j+1,i] + x[k+1,j-1,i] + x[k+1,j,i+1] + x[k+1,j,i-1])) / d_0$;
4: $qb[1] = (c_0*x[k-1,j,i] + c_1*x[k,j,i] + c_2*x[k-2,j,i] + c_3*(x[k-1,j+1,i] + x[k-1,j-1,i] + x[k-1,j,i+1] + x[k-1,j,i-1])) / d_0$;
Note: There must be $\frac{c_0+c_1+c_2+4*c_3}{d_0} = 1$, namely $c_0 + c_1 + c_2 + 4*c_3 = d_0$ in all of the equations above.
```

Algorithm 2. Code Segment of 2-D State Reconstruction

```
$X$-axis original code in 2-D state reconstruction step
1: $qL[0] = (c_0*x[j,i] + c_1*x[j,i-1] + c_2*x[j,i+1] + c_3*(x[j+1,i] + x[j-1,i])) / d_0$;
2: $qR[0] = (c_0*x[j,i] + c_1*x[j,i+1] + c_2*x[j,i-1] + c_3*(x[j+1,i] + x[j-1,i])) / d_0$;
3: $qR[1] = (c_0*x[j,i+1] + c_1*x[j,i] + c_2*x[j,i+2] + c_3*(x[j+1,i+1] + x[j-1,i+1])) / d_0$;
4: $qL[1] = (c_0*x[j,i-1] + c_1*x[j,i] + c_2*x[j,i-2] + c_3*(x[j+1,i-1] + x[j-1,i-1])) / d_0$;
Note: There must be $\frac{c_0+c_1+c_2+4*c_3}{d_0} = 1$, namely $c_0 + c_1 + c_2 + 4*c_3 = d_0$ in all of the equations above.
```

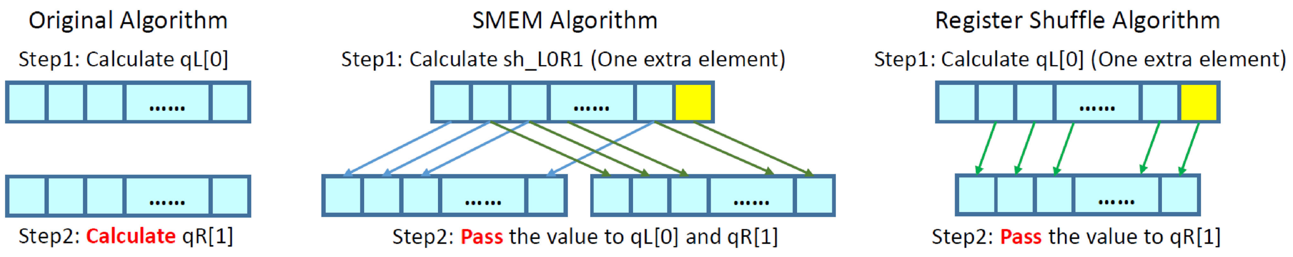

Original Algorithm
Step1: Calculate qL[0]
Step2: Calculate qR[1]
SMEM Algorithm
Step1: Calculate sh_L0R1 (One extra element)
Step2: Pass the value to qL[0] and qR[1]
Register Shuffle Algorithm
Step1: Calculate qL[0] (One extra element)
Step2: Pass the value to qR[1]

Fig. 4. An explicit cache mechanism. By employing shared memory or registers as data depot, both the global memory access and computations are saved. However, since there is a one-element offset between qL[0] and qR[1], one extra element in boundary should be calculated.

the hardware-based tuning techniques as the first-round optimization and then employs tuning methods indicated in this part as comparison, would be demonstrated in Sections 5.3 and 6 so as to indicate the effectiveness of tuning techniques presented in this section. In addition, due to the similarity of code segments of 2-D and 3-D module, here we only take Algorithm 2 as an example, and the same approach could be directly deployed in similar kernels of both the two algorithms.

This section includes the basic optimizing idea and tradeoffs of on-chip resources as well as the different organization methods of GPU threads. The contents of this section could be summarized as follows: 1) Identifying the optimizing potential of FVM algorithmic features on modern GPU platforms, proposing basic ideas of algorithmic modification. 2) Presenting a customizable data caching mechanism to find out the best data depot for intermediate data, so as to cut both the calculation and the memory access by half (or even 75 percent after step 3). 3) Recognizing the optimal global memory loading strategy to further eliminate unnecessary global memory access and increase its accessing speed. 4) Adopting inner-thread rescheduling to balance the computation time of each thread and the effective thread number. According to the experimental results of the five kinds of mainstream GPU platforms, compared with the performance of GPU base version, while adopting the hardware-oriented optimizations has already achieved 3-6 times performance speedup on different platforms for both Euler and SWE solver, an additional 62 to 102 percent (or 9 to 36 percent) performance benefit is able to be achieved for the Euler solver (or SWE solver) by properly use the hardware-software co-design methods presented in this section.

4.1 Observation of Algorithmic Feature

Global memory access in GPU is an expensive operation which could stall the execution pipeline for a long time. Programmers have every reason to minimize global read and write by adopting on-chip memory access on GPU, since access latency of on-chip memory is usually two orders of magnitude lower than that of global memory [30].

By analyzing the physical features of FVM as well as the code segment shown in Algorithm 2, elements used in the calculation of qR[1] are only one step forward of qL[0]. In other words, the value of qR[1] of mesh point (j,i) is absolutely the same with the value of qL[0] of point (j,i+1), as indicated in Fig. 3. However, this feature is not being used on CPU-based FVM solvers since CPUs are equipped with huge amount of cache spaces and the related data would be stored in cache automatically thus to minimize global memory access. In this case, each mesh element (such as point (j,i) and point (j,i+1)) need to calculate all the four values independently, and do not interconnect with other elements. However, since the on-chip memory space is very limited on GPU devices, data used in the calculation can not be fully loaded into fast memory spaces. As a result, the long latency global memory access will occur which would remarkably hit the performance of the FVM solver.

To fully take advantage of the algorithmic feature indicated above, if there is a proper data depot which could be used to store the computational results of qL[0] for each mesh element, we could load qR[1] for these mesh elements from the depot when we need their values, rather than recalculate them as before. In this case, we could minimize the global memory access of the state reconstruction step and reduce the computational overhead at the same time, which is exactly what we need on GPU devices.

4.2 Customizable Data Caching Depot

To find a suitable data depot on GPU devices, first we should identify the requirements. First of all, since each thread needs to load the value calculated by its neighbors (as shown in Figs. 4 and 7), the depot should be shared by a group of threads, and the more threads could access the buffer, the fewer depots are needed, resulting in less boundary processing overhead. In addition, the access speed of the data depot should be much faster than that of the global memory, thus memory-latency would not become the stall reason for the whole program.

Among the memory hierarchy of GPU, shared memory (SMEM) meets all these features well. All threads inside one thread-block could access the SMEM with a very low memory latency, making SMEM a perfect choice of being the data depot. By employing shared memory, all threads inside one thread-block write their own qL[0] into shared memory at time t and load the value from shared memory into qR[1] at time step $t+1$ with a one-element offset, as shown in the middle of Fig. 4.

According to the conservation law, there are four groups of components could adopt the offset feature in 2-D FVM solver (and five groups for 3-D cases). Thus, there are two considerable choices about how to adopt shared memory as data depot. Take the 2-D case as an example, we could either adopt one group of shared memory as a reusable data depot, and move the computational results into registers after each round of computation (choice 1 of Fig. 5), or employ 4 groups of shared memory to store data arrays of each component (choice 2 of Fig. 5).

Compared with the original program without adopting the data cache mechanism, by employing 4 groups of SMEM, around 50 percent (75 percent after adopting method

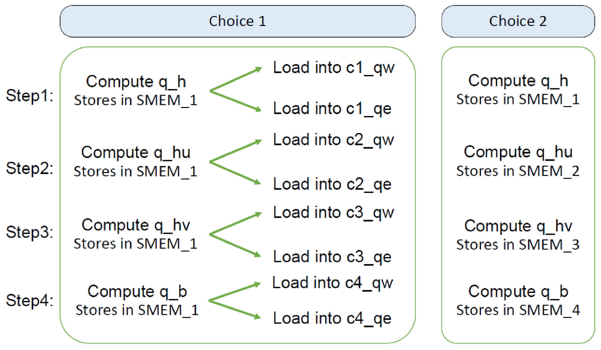


Fig. 5. Two choices of taking advantage of shared memory inside each streaming multiprocessor.

presented in Section 4.3) global reads of the input array could be saved in each round of state reconstruction step, at the cost of using 17KB SMEM for each 256-thread block and adding one more $\_\_synchthreads()$ after the whole step. On the other hand, if we adopt SMEM as a reusable data depot as shown in choice 2, to reduce 50 percent (75 percent after adopting method presented in Section 4.3) global reads of array x, 5 KB SMEM space, 32 more 64-bit register files, 4 $\_\_synchthreads()$ and more loading steps are needed inside each thread.

Besides adopting SMEM as data depot, aiming at efficiently executing some commonly-used computational patterns such as reduction, a new instruction set called register shuffle was introduced with the advent of Kepler GPUs. This technique enables a thread to directly read a register from another thread in the same warp without going through shared (or global) memory, and this function exactly meet our demand of data depot. Thus, we could also implement the algorithm adjustment indicated above by using register shuffle, as demonstrated in Part 3 of Algorithm 3.

Compared with the shared-memory based implementation, register shuffle has lower latency than shared memory access and does not consume shared memory space for data exchange, so this can present an attractive way for applications to rapidly interchange data among threads [31]. However, more register files are needed when we employ register shuffle, which may decrease the GPU kernel occupancy and hit the performance when huge number of register files have been already adopted in the program.

Specifically, in the case of the FVM solver, compared with the first choice of the SMEM, we could figure out that while same percentage of memory access and computation are saved, the register shuffle method employs the same number of registers but no shared memory is involved. Thus, we could affirm that register shuffle must be better than the choice 1 of shared memory. However, when compared with the SMEM choice 2, different kinds of resources are adopted. Thus, though both ways could take advantage of the algorithmic features and result in a better performance compared with the original algorithm, a set of experiments should be taken to identify the best strategy for specific applications on certain platforms.

4.3 Optimal Global Memory Loading Strategy

Through the optimization steps mentioned above, we have reduced the amount of calculation by half and alleviated

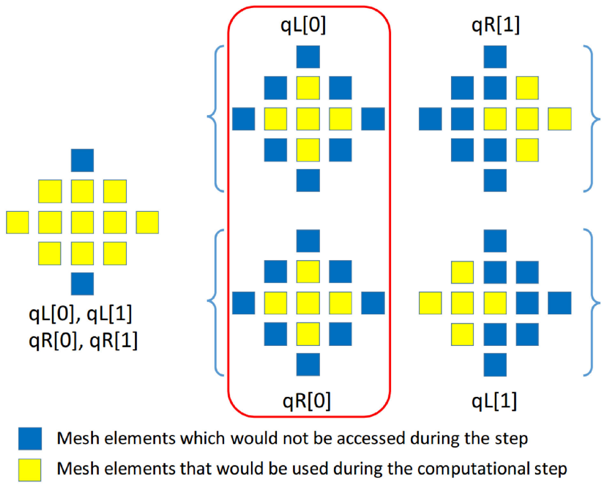

qL[0]
qR[1]
qL[0], qL[1]
qR[0], qR[1]
qR[0]
qL[1]
Mesh elements which would not be accessed during the step
Mesh elements that would be used during the computational step

Fig. 6. Yellow points are mesh elements needed to be accessed for getting corresponding state variables. To minimize the total number of points that need to be accessed, qL[0] and qR[0] are chosen among the four choices, since in this case only five elements are needed.

the pressure of global-memory access to some extent. We could further reduce global-memory access by choosing the optimal element pattern to compute.

In order to provide a clear description, here we introduce the term ‘element pair’ to describe relationships between qL [0] and qR[1]. Originally, there are two ‘element pair’ need to be calculated for each thread, i.e., qL[0], qR[1] and qL[1], qR [0]. By adopting the suitable data depot (SMEM or Register), inside each element pair, only one element needs to be calculated for each thread, as shown in Fig. 6. Among the four choices, we should choose qL[0] and qR[0] as computational patterns that need to be solved, since in this case only five mesh elements are needed, rather than access 11 mesh elements originally.

In addition, since mesh elements employed in the calculation of qL[0] are the same of qR[0], we employ static variables to avoid redundant memory access so as to take benefits from their locality, as shown in Part 1 of Algorithm 3. Besides, since division is an expensive operation for GPU, and all of the coefficients (c_0, c_1, c_2, c_3 and d_0) we need in the whole step are fixed, the division operation could be done outside of the FVM solver. In the following steps, we could directly employ the calculation results of the reconstruction step, as r_1 to r_4 shown in Part 2 and 3 of Algorithm 3.

Moreover, to achieve the best performance in the y-axis (and z-axis in the 3D case), once the thread-block size is settled, the following two principles should be adopted: (1) ensuring the number of threads on the x-axis is a multiple of 16, in consideration of ‘warp execution’ of the GPU processor. (2) making the number of threads along the ‘acting dimension’ as large as possible. For instance, in a 3-D FVM solver whose thread-block size is 256, if the kernel is acting on the y-axis, a (16,16,1) thread-block would be chosen so that we could achieve the best performance.

4.4 Inner Thread Rescheduling Tradeoff

Though a significant performance boost could be achieved up to now, we should notice that the real case is not as simple as described above. Since there is an one-element offset between qL[0] and qR[1], we need to load the halo part

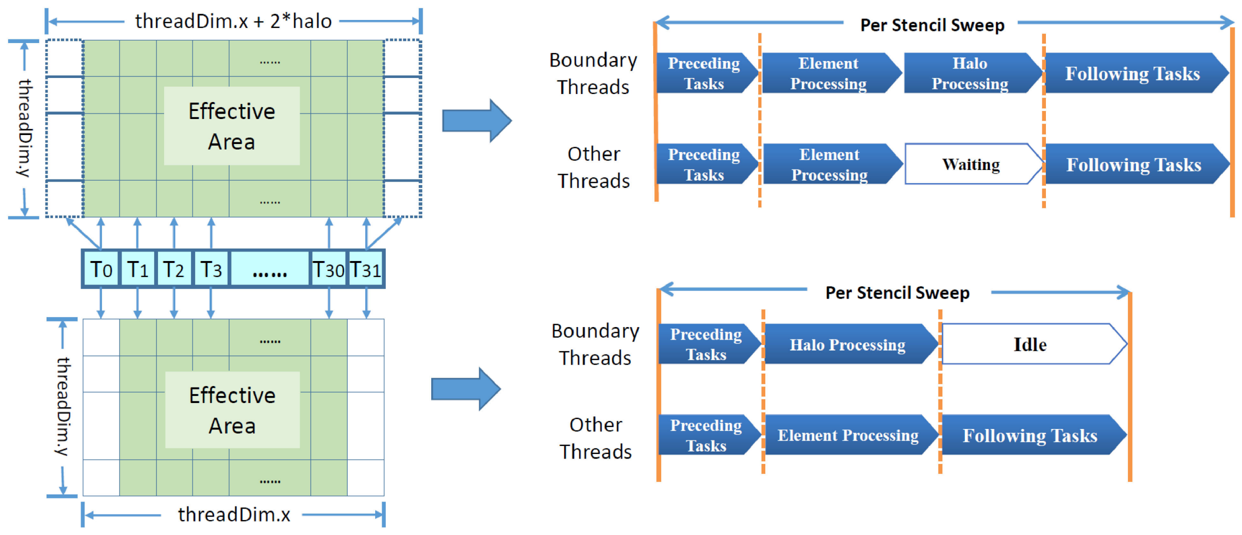


Fig. 7. By adopting halo threads, all threads are liberated from waiting at the cost of wasting the computational ability of halo threads in the following steps (Riemann solver step in our case) This method could be extremely useful for stencil computation with small halos.

(yellow area of Fig. 4) into shared memory (or register files). However, since the thread number is the same as the array size, some threads have to focus on both their own computations and the halo processing (as shown in the top of Fig. 7). This approach will lead to the workload imbalance between different threads within each warp.

Algorithm 3. Optimized State Reconstruction Step

```
Part 1: Avoid Redundant Memory Access
  1: $t\_curr$ = x[j,i];
  2: $t\_left$ = x[j,i-1];
  3: $t\_rigt$ = x[j,i+1];
  4: $t\_rest$ = x[j+1,i] + x[j-1,i];
Part 2: Shared-Memory Implementation
  1: sh_L0R1[j,i] = $r_1$*t_curr + $r_2$*t_left + $r_3$*t_rigt + $r_4$*t_rest;
  2: sh_R0L1[j,i] = $r_1$*t_curr + $r_2$*t_rigt + $r_3$*t_left + $r_4$*t_rest;
  3: __syncthreads();
  4: $qL$[0] = sh_L0R1[j,i];
  5: $qR$[1] = sh_L0R1[j,i+1];
  6: $qR$[0] = sh_R0L1[j,i];
  7: $qL$[1] = sh_R0L1[j,i-1];
  8: __syncthreads();
Part 3: Register Shuffle Implementation
  1: $qL$[0] = $r_1$*t_curr + $r_2$*t_left + $r_3$*t_rigt + $r_4$*t_rest;
  2: $qR$[0] = $r_1$*t_curr + $r_2$*t_rigt + $r_3$*t_left + $r_4$*t_rest;
  3: __syncthreads();
  4: $qL$[1] = __shfl_up(qR[0], 1, 32);
  5: $qR$[1] = __shfl_down(qL[0], 1, 32);
  6: __syncthreads();
```

For instance, suppose two dimensions of our thread-block are 8, 32 respectively on the y-axis and x-axis, an array of 8*(32+1) needs to be created in order to handle the one-element offset of each element pair, which would result in branching statements inside or outside the warp. As a result, while some threads focus on boundary processing, the others have to wait due to the synchronization. According to the experimental results on five GPU platforms, such wait time could take 21-29 percent of the overall time (namely nearly half of the time consumption of the whole state reconstruction step), which is very expensive. The whole process could be demonstrated as Algorithm 4.

Algorithm 4. Code Segment before Thread Rescheduling

```
Inside each GPU Kernel
  1: int j = blockDim.y * blockIdx.y + threadIdx.y + WIDTH;
  2: int i = blockDim.x * blockIdx.x + threadIdx.x + WIDTH;
  3: Preceding tasks
  4: ======== State Reconstruction Begin ========
  5: Shared-memory computation (or register shuffle);
  6: if $threadId < halo\_width$
  7:     left-halo element computation;
  8: else if $threadId > (blockDim.x - halo\_width)$
  9:     right-halo element computation;
 10: end if
 11: $\_\_synchronous()$
 12: ========= State Reconstruction End ========
 13: Following tasks
```

To avoid divergence in Algorithm 4, one choice for us is to set some threads as 'halo threads' that specifically take charge of boundary processing, as shown in Fig. 7. These halo threads do exactly the same work as other threads do before the element processing step and keep idle afterwards. By adopting these halo threads, we could eliminate the branch statements (line 6 to line 10) in Algorithm 4.

The new algorithm and its workflow are summarized as Algorithm 5 and Fig. 7 respectively. Compared with the original algorithm, we suppose each thread need to spend t_1 and t_2 to deal with the two times of $StateReconstructionStep$ and the one step of $Followingtasks$ shown in Algorithm 4 respectively. To finish the computation of one time step, without applying this inner-thread rescheduling method, each warp (32-thread) need to spend $T_0 = t_1 + t_2$ to finish the whole task. After adopting the new technique, $T_1 = \frac{32}{32-2*halo} * (\frac{t_1}{2} + t_2)$ are needed. After simplifying the equation, we could find out T_0 - $T_1 = \frac{1}{16-halo}[(8 - halo) * t_1 - halo * t_2]$. If this number is positive, it means the original algorithm takes more time than the new program does, which indicates that the new algorithm (adopting inner-thread rescheduling method) is better.

TABLE 2
Key Architectural Parameters of the Evaluated Platforms

Description	Fermi C2070	Tesla K40	Tesla K80[1]	Tesla P100	Tesla V100[2]
Chip	GF100	GK110	2*GK210	GP100	GV100
TPCs	16	15	2*15	28	40
SMs	16	15	2*15	56	80
FP64/SM	16	64	2*64	32	32
FP64 / GPU	256	960	2*960	1792	2560
Clock Rate (base/boost)	1150 MHz	745 MHz/875 MHz	562 MHz/875 MHz	1328 MHz/1480 MHz	Unknown/1370 MHz
Peak FP64 Perf (TFlops)	0.51	1.68	2.91	5.3	7.0 (V100 for PCIe)
Memory Interface	DDR5	DDR5	DDR5	HBM2	HBM2
GMEM Size	6 GB	12 GB	2*12 GB	16 GB	16 GB
Bandwidth	144 GB/s	288 GB/s	2*240 GB/s	732 GB/s	900 GB/s
L2 Cache	768 KB	1536 KB	2*1536 KB	4096 KB	6144 KB
SMEMSize/ SM (KB)	16/32/48	16/32/48	80/96/112	64	Up to 96
Register Size /SM (KB)	128	256	512	256	256
Register Size /GPU (KB)	1920	3840	2*7680	14336	20480
Inter connect	PCIe 2.0	PCIe 3.0	PCIe 3.0	PCIe 3.0 + NVLink	PCIe 3.0 + NVLink
ARCH-FBR[3]	3.58	5.83	6.06	7.24	7.96

[1] *Though each GK210 contains 15 SMs, due to the power limitation, Tesla K80 could only start 2*13 SMs at the same time.*
[2] *The Tesla V100 GPU adopt here is Tesla VI00 for PCIe.*
[3] *Architectural Flop to Byte Ratio equals to Peak Performance / Bandwidth.*

This inner-thread rescheduling method could be widely used in GPU solvers aiming at optimizing stencil computations employed register shuffle or shared memory, and according to the conclusion indicated above, this technique would be extremely effectual when the halo area is small. Still take Algorithm 2 as an example, in this case the halo equals one and the value of t_1 is around 2 times of that of t_2. Thus, T_0 - $T_1 = \frac{1}{15} * (7 * t_1 - t_2) = \frac{13}{15} * t_2$, it is definitely a positive number. Thus, we could declare that the new algorithm which adopts inner-thread rescheduling method is better then the original one, and experimental results shown in Section 6 proves our judgement.

Algorithm 5. Code Segment after Thread Rescheduling

```
Inside each GPU Kernel
 1: int j = blockDim.y * blockIdx.y + threadIdx.y + WIDTH;
 2: int i = (blockDim.x-2) * blockIdx.x + threadIdx.x + WIDTH;
 3: Preceding tasks
 4: ======== State Reconstruction Begin ========
 5: Shared-memory computation (or register shuffle);
 6: $\_\_synchronous()$
 7: ======== State Reconstruction End ========
 8: if $((threadIdx.x >= halo\_width)$ && $(threadIdx.x < blockDim.x - halo\_width))$
 9:     Following tasks
10: end if
```

5 Testbed and Applications

In order to make the evaluation of customized FVM tuning techniques presented in this paper (Section 4) more fair and convincing, the main way to highlight the effectiveness of the tuning methods is to compare the performance of the same generation GPU before and after using the optimization methods, and the CPU performance is mainly adopted as a standard unit to indicate the performance relationship between different GPUs. Therefore, in this section we first provide a brief introduction of the four generations (Fermi, Kepler, Pascal and Volta) of GPU platforms as well as the target atmospheric simulation programs. Then, a set of GPU-based general-used hardware optimization methods are adopted to the two test applications so as to provide a hardware-optimized performance as the baseline in future comparison (while only the general CPU optimization methods such as MPI, OpenMP, compiler-based optimization and SIMD are used as it is not the key point of this work). Finally, summarization and performance metrics are provided to prepare for the performance evaluation and analysis shown in the next section.

5.1 Evaluated Platforms

To provide a full scale analysis of the optimization methods, in this work we measure the performance of the optimized code on 5 kinds of mainstream GPU devices which are widely used to accelerate scientific applications nowadays, including Fermi C2070, Tesla K40, Tesla K80, Tesla P100 and Tesla V100. Table 2 demonstrates the key architectural parameters of these platforms, in which only double-precision related features are listed.

Though the Fermi GPU is released in 2009 and quite out-of-date, it is one of the most successful template of modern GPUs and is still equipped in many Top500 supercomputers such as Tianhe-1A [32] and Nebulae [33]. As a great innovation compared with previous GPU platforms, many efficient designs are first adopted in Fermi architecture, resulting in a remarkable performance boost in some cases compared to general-purpose CPUs. When it comes to Kepler GPUs, to achieve higher performance, additional execution and memory resources (more CUDA cores, registers and caches) are employed. Such feature significantly increases the performance of Kepler GPUs when compared to Fermi. In Tesla K80, Nvidia further doubled the on-chip memory spaces of each computational chip. That is of vital importance for latency-bound programs such as FVM-based solvers. Therefore, among four kinds of Kepler GPUs we select K40 and K80 as test platforms in this work.

While Maxwell GPUs are not good at dealing with double precision scientific applications, Tesla Pascal and Volta architectures enable the extreme performance for both

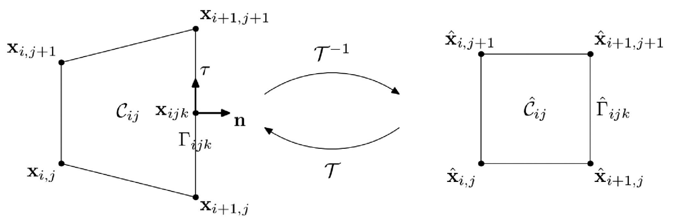


Fig. 8. A coordinate transform between the physical mesh and the computational mesh.

scientific programs and deep learning applications. The key innovations of Tesla P100 and V100 includes: 1) Extreme performance of its computational chip (GP100 and GV100) 2) Fast global memory interface (HBM2) 3) High speed interconnect (NVLink) 4) Deep learning oriented features and architectures (such as tensor cores inside GV100). Due to the extreme performance potential and balanced ability for both scientific applications and deep learning tasks, Tesla P100 and V100 are chosen as the accelerator processor for huge number of top supercomputers of Green500 [37] and even upcoming fastest supercomputers (such as Summit [35] and Sierra [36]). Since this work mainly focuses on the algorithmic optimization within computational chips of GPU, we will only pay attention to the scientific-application-relative innovations within GP100 and GV100. Such differences are summarized as follows:

First of all, compared with computational chips of previous GPU architectures, GP100 and GV100 adopt new high-performance manufacturing processes (16 and 12 nm FinFET, compared with 28 nm for Kepler and Maxwell) thus to provide better power efficiency. As a result, with similar power consumption, each GP100 (or GV100) is equipped with more computational units (namely FP64/FP32 cores) and has higher clock rate when comparing with previous generation GPU chips such as GM200 and GK110. Therefore, when we move our program from previous GPU platforms onto Tesla P100 or V100, in most cases remarkable performance benefit can be achieved even without the modification of the code.

Second, the SM architecture within P100 and V100 are redesigned to provide extreme performance for both AI and scientific applications. 1) In the view of on-chip memory (L1 cache, SMEM and register files), the ratio of its capacity to computational unit (FP64 cores) keeps increasing from version to version, such feature matches the increasing fast memory demand for modern complex applications. 2) Though the global memory access speed and L2 cache size are remarkably higher than previous GPUs, the growth rate of such features is lower than that of computational units. Combining these two features, sophisticated manual designs are required so as to fully take advantage of the fast memory spaces on P100 and V100. 3) Some new attempts are made to explore the optimal design of modern GPUs, such as the combination mode between L1 cache and shared memory, the execution strategy of INT32 and FP64, and the new thread scheduling mechanism. Under the circumstances, specialized tuning is required thus to fully unleash the performance potential of new GPU architectures.

5.2 Atmospheric Simulation Applications

3D Euler equations and 2D shallow water equations (SWE) are two most essential dynamic components for non-hydrostatic atmospheric modeling. Compared with physical schemes such as WSM5 in WRF [38] and the shortwave radiation parameterization in CAM [39], dynamic core is inherently more difficult to achieve performance benefits from heterogeneous platforms. Investigating the performance potential of these two atmospheric equation solvers is of vital importance in terms of improving the overall performance of atmospheric model based on heterogeneous platforms.

In this part we take the 3-D compressible Euler equation as an example ([11], [25], [27]), the equation could be written as

$$\frac{\partial Q}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial z} + S = 0. \tag{6}$$

We define a nonsingular mapping M as $\hat{x_{ij}} \rightarrow x_{ij}$ for *i* = 0, $1 \dots n_1$ and *j* = 0, 1, $\dots n_2$, denote C_{ij} as a mesh cell formed by mesh points $x_{ij}, x_{i+1,j}, x_{i+1,j+1}, x_{i,j+1}$ and $\hat{C_{ij}}$ as a mesh cell formed by mesh points $\hat{x_{ij}}, x_{i\hat{+1},j}, x_{i+\hat{1},j+1}, x_{i,\hat{j}+1}$ as shown in Fig. 8.

By adopting cell-centered finite volume scheme, we could define the approximate solution at time *t* as

$$Q_{ij}(t) = \frac{1}{|C_{ij}|}\int_{C_{ij}} Q(x,t)dx, \tag{7}$$

where $i = 0, 1, \dots, n_1 - 1, j = 0, 1, \dots, n_2 - 1$. Then we have

$$\frac{\partial Q_{ij}(t)}{\partial t} + \frac{1}{|C_{ij}|}\int_{\partial C_{ij}} (F(Q(x,t))n_x + G(Q(x,t))n_z)ds + S_{ij}(t) = 0, \tag{8}$$

where $(n_x, n_z)^T$ is the unit outward normal of ∂C_{ij}. The boundary of C_{ij} should be further decomposed into four segments, i.e., $\partial C_{ij} = \cup_{k=1}^4 \Gamma_{ijk}$, in order to evaluate the numerical fluxes of F and G. On Γ_{ijk}, we denote the unit outward normal vector as $n = (n_x, n_z)^T$ and correspondingly the unit tangent vector as $\tau = (-n_z, n_x)^T$. As a result, a new Cartesian coordinate could be formed by vector n and τ. Based on the new Cartesian coordinates, given a state variable Q, we could express it as $q = T_{ijk}Q$ where $T_{ijk} = diag\{1, L_{ijk}, 1\}$ and $L_{ijk}^T = (n, \tau)$.

Based on the new Cartesian coordinates, the second term in Equation (8) becomes

$$\begin{aligned}\int_{\Gamma_{ijk}} &(F(Q(x,t))n_x + G(Q(x,t))n_z)ds \\ &= T_{ijk}^{-1}\int_{\Gamma_{ijk}} F(T_{ijk}Q(x,t))ds \\ &\approx T_{ijk}^{-1}|\Gamma_{ijk}|F(q(x_{ijk},t)).\end{aligned} \tag{9}$$

The numerical flux $F(q(x_{ijk},t))$ in Equation (9) is then estimated by employing a Riemann solver together with state reconstruction step. On heterogeneous platforms, in order to achieve a balanced task division between host and device, there are several intra-node partition methods available for us, such as process-level partition ([40], [41]), function-partition ([42], [43]) and so on. Among these choices, we choice the adjustable inner-outer partition method as indicated in [11]. The hybrid domain decomposition algorithm and the computational structures of these two stencils are shown below (Algorithm 6 and Fig. 11).

Algorithm 6. CPU-GPU Hybrid Euler Algorithm

```
1:  CPU Begin
2:      Data Initialization for CPU(x,xs,xs1,xs2,xs3 and f)
3:      Data Initialization for GPU(inputx,inputxs,and so on)
4:      for $(k, j, i) \leftarrow (0, 0, 0)$ to $(nzl, nyl, nxl)$ do
5:          if$(k, j, i)$ $\epsilon$ Boundary then;
6:              Halo Updating and Boundary Process
7:          end if
8:          else
9:          ============GPU Begin============
10:             Calculate Coordinate
11:             Calculate Fluxes{
12:                 State Reconstruction
13:                 Riemann Solver}
14:             Compute Source Terms
15:         ============GPU End=============
16:         end else
17:     end for
18: CPU End
```

5.3 Hardware Based Optimizations

As indicated in Section 1, there are three main challenges we need to face when we optimize the two programs on heterogeneous CPU-GPU platforms, as most dynamic parts do in atmospheric models: 1) low data communication bandwidth between CPU and GPU; 2) low Flop-to-Byte ratio of the state reconstruction step; 3) low IPC (instruction per cycle) caused by the long-latency operations within the Riemann solver step. To make things worse, since there is execution dependency between the state reconstruction step and the Riemann solver step, we need to accelerate both parts so as to achieve extreme performance. In order to solve the above challenges, taking the 3D Euler solver as an example, the general used GPU-based hardware optimization methods could be summarized as follows:

5.3.1 Minimizing Access Latency of Global Memory

1. *L1/Shared Memory Configuration.* Based on the memory hierarchy of GPU platform, the first level cache, whose total space is fixed, can be configured as two types of cache-like memories: the L1 cache and shared memory (SMEM). For high dimensional complex stencil computations, general shared memory usage may lead to a performance drop [45]. This conclusion is approved by the experimental results when we applying the general used 3-D and 2.5-D [14] shared memory buffering method to our Euler solver. To take full use of on-chip memory spaces of GPU, we maximize the L1-cache space by employing 'preferL1' option in our Euler solver, at the cost of minimize the space of shared memory. At the same time, since L1 cache is no longer used for DRAM load caching by default since Kepler GPUs, to eliminate the disadvantages caused by this feature, on some GPU platforms (such as Tesla K40 or K80) we could employ '-Xptxas -dlcm=ca' flag to take the Fermi style caching of both global and local loads [44].

2. *Coalesced Access.* In all kinds of GPU platforms mentioned in this work, global memory access within one warp can be coalesced into one memory transaction, as long as the data is continuously accessed. In the original code of Euler solver, array of structure (AoS) is adopted for data storage in order to facilitate the code implementation. However, the access of this kind of data structure is not able to be coalesced, which would greatly increase the memory access latency on GPU platform.

To achieve coalesced access of global memory, we reformat the data structure on the GPU side from Array of Structure (AoS) to Structure of Array (SoA). According to the profiling result shown by nvvp (NVIDIA Visual Profiler), both global memory access speed and L1 global hit rate are increased by adopting this approach, resulting in more than 30 percent performance boost.

3. *Read-Only Cache.* Read-only data from global memory could be loaded by Read-only cache in a relatively high bandwidth compared with global memory accesses, cause it is a separate cache line with special memory pipe and relaxed memory coalescing rules. Read-only cache could be explicitly employed by annotating arrays with the __ldg intrinsic. In our Euler solver, we put the un-coalesced read-only variables into the read-only cache to improve the overall performance.

5.3.2 Tuning the Register Usage within Each Thread

Register usage of each thread is a key factor of the tradeoff between memory latency and thread-level parallelism. Since register is a limited resource that all threads residing on a multiprocessor must share, if one thread uses too many registers, the number of active warps (namely, thread groups) that can reside on a multiprocessor would be reduced, thus lowering the occupancy of the multiprocessor.

In our Euler solver, 256 registers (which is the maximum number in Tesla GPUs) are needed in each thread by default. Accordingly, the occupancy of our program is less than 25 percent, which significantly reduces the thread concurrency. To determine the optimum balance of register usage and occupancy, we write a script to find out the SM-level optimal register size within each thread respectively in different kinds of GPUs.

5.3.3 Kernel Splitting and Streaming Concurrency

Kernel splitting is another optimizing options for applications stalled by execution dependency, as it can decrease the usage of local memory and achieve higher occupancy and cache hit rate of the kernel. According to the algorithm of Euler solver, computations along three different dimensions are independent with each other. Thus, we split the kernel into 3 parts to further reduce register usage of each kernel, at the cost of subtle increase of memory transactions. According to the experimental results, a more than 15 percent performance benefits could be achieved via this approach.

Concurrent kernel execution could partially overlap the kernel execution with the data transfers. As the stencil computations along the three dimensions are independent, we employ different streams to deal with these three kernels to achieve a further speedup.

5.4 Summarize and Performance Metrics

To fully explore the performance of these programs on modern GPUs, the two dynamic cores are rigorously optimized

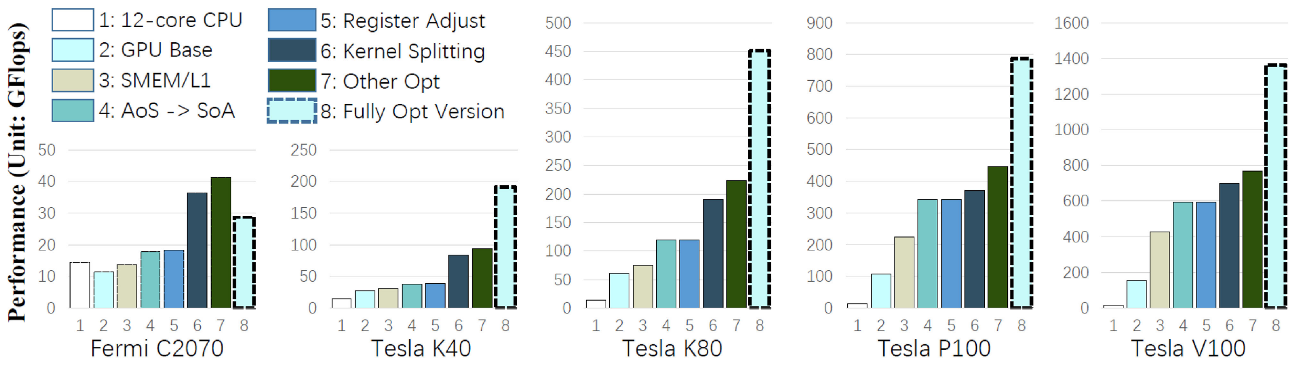


Fig. 9. Euler performance tuning by employing tuning techniques based on five kinds of GPU platforms. Mesh size: 484*232*116. Bar 7 indicates the best hardware optimized version, and Bar 8 shows the optimal performance result after adopting tuning techniques indicated in Section 4.

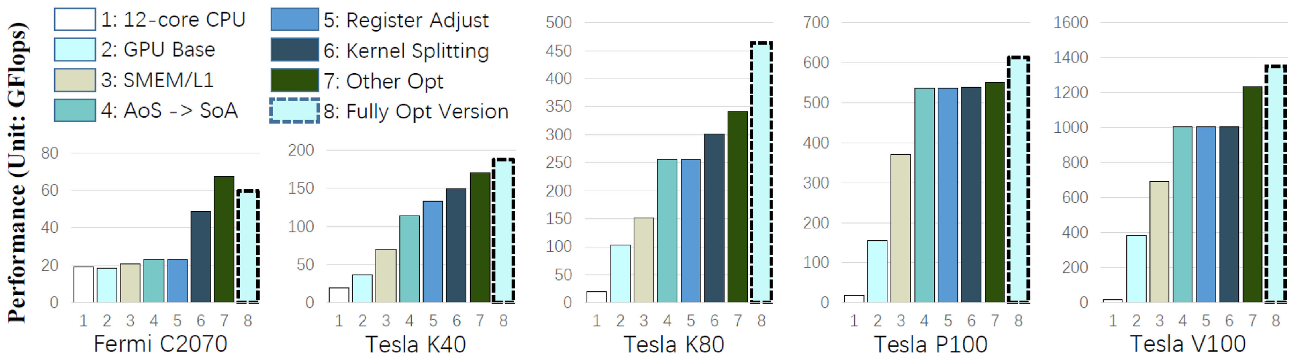


Fig. 10. SWE performance results by employing tuning techniques based on five kinds of GPU platforms. Mesh size: 1024*1024. Bar 7 indicates the best hardware optimized version, and Bar 8 shows the optimal performance result after adopting tuning techniques indicated in Section 4.

by determining the best choice of preferring SMEM or L1 for kernels that have data elements accessed by more than one thread within a block (Bar 3 of Figs. 9 and 10), ensuring all off-chip GMEM access are coalesced (Bar 4), maintaining optimal occupancy to ensure enough active warps in flight (Bar 5), employing read-only cache and kernel splitting to improve memory accessing speed (Bar 6), reducing redundant calculations by properly use of registers, and adopting automatic adjustment programs to identify the optimal parameters such as block size and thread number in each dimension (Bar 7). As demonstrated in Figs. 9 and 10, by adopting these hardware-based tuning methods, a 53.0 and 64.3 times speedup could be achieved on Tesla V100 over the original 12-core CPU results based on Intel E5-2697 (v2) for Euler and SWE respectively (Bar7 versus Bar1), and the performance of the optimized version is 3 to 6 times faster than the GPU base version on different GPU platforms for both Euler and SWE (Bar 7 versus Bar 2).

As for the evaluation of the performance, employing performance as the metric of evaluating kernel efficiency is the

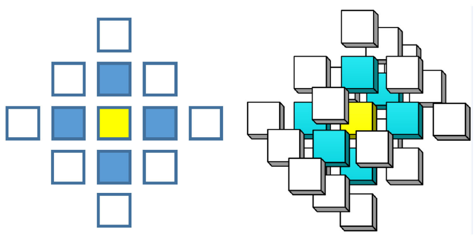


Fig. 11. Left: 2-D 13 points stencil in shallow water equations (SWE). Right: 3-D 25 points stencil in Euler equation.

simplest option. However, the framework presented in this paper is not designed to track the change of the computations. What we really care about is the number of mesh elements processed per second. A more effective method is to use a simple model based on empirical measurement of the number of points we could process (Points/s) and operation numbers of the original kernel. According to the testing results of the base version, the number of operations of Euler and SWE are 1,588 and 839 respectively.

Furthermore, as the FVM solver is combined with the memory-bound state reconstruction step and the computation-bound Riemann solver step with execution dependency between them (detailed in Section 3.2), to the best of our knowledge there is no existing performance model could provide the accurate performance evaluation for such case. For instance, the most commonly-used Roofline model [47] is a good way to estimate the performance of memory-bound or compute-bound programs (such as stencil, LBM, FFT, CNN). However, to the best of our knowledge it cannot handle the programs with both memory-bound and compute-bound properties and execution dependency between the two parts (such as WRF, CESM, COSMO, etc.), cause the upper bound provided by such model will never be able to even get close. Specifically, still take the Euler solver as an example. If we insist on using the Roofline Model to measure the performance of the program, for each grid element there are 1,588 operations within each time step, and to finish the computation of each time step, an averaged 40 input elements and 5 output elements are need to be accessed if fast-memory buffer (such as shared-memory, L1-cache and read-only cache) are properly used. As a result, the arithmetic

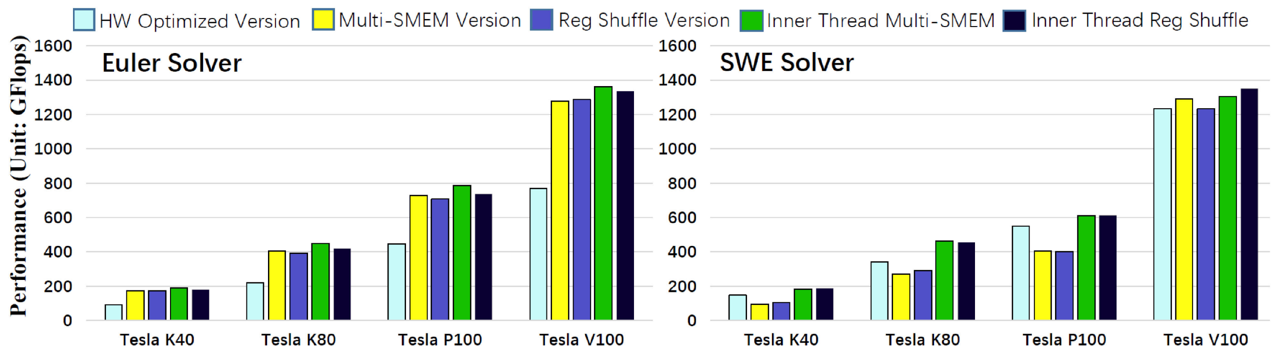


Fig. 12. Performance (GFlops) of the four kinds of optimized versions of Euler equations (Euler Solver) and shallow water equation (SWE Solver) on four mainstream GPU platforms. Mesh size of Euler and SWE are 484*232*116 and 1024*1024 respectively.

intensity of the complete Euler solver is 1588/(45*sizeof(double)) = 4.41. However, as most of the data loading is needed for the state reconstruction step, the arithmetic intensity of state reconstruction step is only 240/(40*sizeof(double)) = 0.75, and during such state reconstruction step no other computation operations could be executed. This will make our estimation results have a huge gap with the actual situation. Therefore, in this work we adopt the speedup of hardware themselves and the time-to-solution as metrics to measure the performance improvement, like most works do in the field [48], [49].

6 Performance Evaluation and Analysis

Fig. 12 demonstrates the performance of Euler and SWE after adopting different kinds of optimization strategies based on four mainstream GPU platforms, from Kepler to Volta. (On Fermi GPU, since register shuffle is not supported, we only demonstrate the results of inner-thread SMEM strategy, as shown in the Bar 8 of Figs. 9 and 10). In this section, we will analyze these experimental results in the following two folds: 1) discuss why the best strategy varies among the four strategies (select proper data depot and determine whether should we use inner-thread rescheduling method, as indicated in Section 4) with the program and platform changes, and how to choose the optimal strategy accordingly. 2) compare the optimal performance (after adopting the best choice of Section 4) with the performance of hardware optimized version, so as to provide a full-scale analysis for the optimization methods in both the application side and the platform side.

TABLE 3
SWE and Euler Speedups after Employing the Optimization Methods

	Program Version	12-core E5-2697 CPU	Fermi C2070	Tesla K40	Tesla K80	Tesla P100	Tesla V100
Euler	Hardware Opt (H0)	1x	2.91x	5.50x	15.68x	34.3x	53.0x
	Fully Opt (F0)		2.02x	10.13x	31.64x	55.4x	93.9x
	Speedup (S0 = F0/H0)	------	**0.69x**	**1.84x**	**2.02x**	**1.62x**	**1.77x**
SWE	Hardware Opt (H1)	1x	3.53x	9.11x	18x	28.9x	64.3x
	Fully Opt (F1)		3.10x	11.32x	24.42x	32.2x	70.4x
	Speedup (S1 = F1/H1)	------	**0.88x**	**1.24x**	**1.36x**	**1.11x**	**1.09x**

First we would like to discuss how to identify the optimal choice among the four versions of optimized program. From Fig. 12 we could figure out that inner thread rescheduling method could benefit both SWE and Euler on all Kepler, Pascal and Volta platforms, which indicates that for one-element halo FVM programs, no matter complex or simple it is, it is highly probable to get performance benefit by adopting the inner thread rescheduling method. This conclusion is in perfect accordance with the calculation presented in Section 4.4.

As for choosing the optimal data depot, for 3-D Euler solvers employing multiple groups of shared memory would come up with a better performance, while for the 2-D SWE programs, adopting register shuffle would be a better choice in most cases. Through further analysis we could find out that within the complex Euler solver, huge number of register files have already been used before we adopt register shuffle, thus the register file rather than shared memory becomes the determinant factor of the GPU kernel occupancy. While in the 2-D SWE solver, less than 50 register files are used before we adopt register shuffle, thus the adoption of register shuffle will not have an immense impact on the GPU kernel occupancy. These results indicate that a set of experiments should be conducted in order to find out which kind of memory space is the optimal data depot for specific application kernels on certain platforms, but generally the register shuffle method should be selected unless it would significantly decrease the *occupancy* of GPU computational kernels, as it happens in our complex 3-D Euler solver.

Following we will regard the four optimizing strategies presented in Section 4 as a union (namely regard the best result as the final-optimized performance, no matter which of the four strategies it is), and analyze its effectiveness in both program side and platform side. To make it more intuitive, Table 3 summarizes the performance comparison of the CPU version, GPU hardware optimized version and GPU fully optimized version (Bar 1, Bar 7 and Bar 8 of Figs. 9 and 10). In this table, while the *HardwareOpt* shown in Table 3 aims at solving the three challenges indicated in Section 5.3, the *Speedup* brought by the tuning techniques presented in Section 4 will only be related to memory access latency and complex computation operations.

In aspect of different computational programs, the more complex a program is (in other words, more mesh elements stored out of GPU on-chip memory), the more benefits we could get by using the optimization methods. In the Euler based dynamic core, a 3-D 25-points stencil is employed in each computational step following by the solving approach of a complex 3-D Riemann solver. In such case, some local variables are inevitable to be driven out to global memory originally, leading to a significant speedup after adopting the optimization methods. However, when it comes to the 2D shallow water equations, almost all mesh elements inside the 13-points stencil have already been stored in register files or L1 cache originally (and even half of the of-chip memory are idle after adopting kernel splitting), as a result, employing the optimizations could only bring us a limited performance enhancement. This feature also indicates that a larger space of on-chip memory is of vital importance for modern many-core accelerators.

In terms of various GPU platforms, our tuning techniques come out with a poor speedup ratio on Fermi GPUs (even less than 1) for both Euler and SWE. The reason is that on Fermi GPUs, there is no enough on-chip memory resources to perform as a data depot. Things become much better when it comes to K40 platform, since the streaming multiprocessors of GK110 both increased the on-chip memory size (within each SM) and the computation ability for complex operations. In this case, adopt our tuning techniques would have remarkable benefits for both memory access latency and complex computation operations. As for Tesla K80, the remarkable performance boost is mainly benefited from the higher capacity of both shared memory and registers inside each Texture/Processor Cluster (TPC). Such device adjustment improves the occupancy of application, and leads to a significant performance speedup. Similar results are achieved on Tesla P100 and V100, however, the performance benefit of the algorithm adjustment is not as good as before, cause the hardware-based version has already taken great benefits from the new architectural feature of GP100 and GV100. As a result, the computation benefit brings by our tuning methods would not be as significant as before. However, even in this case remarkable performance boost is still able to be achieved (mainly due to the memory access benefit bring by our tuning techniques), which fully proves the effectiveness of the tuning techniques presented in this work.

To summarize, based on the experimental results of the five kinds of mainstream GPU platforms, compared with the performance of GPU base version, while adopting the hardware-oriented optimization methods has already achieved 3-6 times performance speedup on different platforms for both Euler and SWE solver, an additional 62 to 102 percent performance benefit is able to be achieved for the Euler solver by properly use the hardware-software co-design methods presented in this work. Similar experimental results are obtained for the optimization of the simpler 2D shallow water equation solver, and 9 to 36 percent further performance speedup is able to be achieved compared with the hardware-optimized version.

7 Conclusion

Scientific HPC applications are increasingly ported to GPUs to benefit from both the high throughput and the powerful computing capacity. Many of these applications, such as atmospheric modeling, room acoustics modeling and hydraulic erosion simulation, are adopting the finite volume method as the solver algorithm. However, large amount of communications within these applications decrease the Flop-to-Byte ratio of these applications, resulting in an insufficient resource usage of GPU platforms.

In this paper, we formulate structured-mesh based FVM tuning on GPU platforms as an optimization task, and introduce a set of general optimization methods that provide a suitable mapping between the algorithmic property of FVMs and the GPU hardware architecture. While the explicit cache mechanism and the optimal global memory loading strategy could cut the global memory transactions by around 75 percent in the state reconstruction step, the inner-thread rescheduling method is able to eliminate unnecessary wait within execution warps. To provide best optimization for different applications and platforms, four kinds of tuning strategies are provided, and experimental results indicate that applying these tuning methods could avoid execution stall of the explicit FVM-based solver to the most extent.

To the end, we evaluate these optimization methods by using two dynamic kernels of real-world atmospheric models, namely 3-D Euler and 2-D SWEs. According to the experimental results on five kinds of mainstream GPU platforms, the fully optimized programs upgrade the hardware utilization and 93.9x speedup and 70.4x speedup is achieved for the 3-D Euler and 2-D SWE solver respectively on Nvidia V100 over one 12-core Intel E5-2697 (v2) CPU, which is a great promotion compared with the original speedup without adopting the tuning techniques presented in this work. By comparing the results of different strategies as well as the performance boost on both application and platform side, we could claim that our generalized optimization methods are able to achieve significant performance boost on all kinds of Kepler, Pascal and Volta GPU platforms.

Acknowledgments

This work was supported in part by the National Key Research & Development Plan of China (grant no. 2016YFA0602200, and 2017YFA0604500), by the National Natural Science Foundation of China (Grant No. 51761135015), by the Basic Research Discipline Project of Shenzhen (Grant No. JCYJ20180508152204044), by the Key Science & Technology Laboratory Project of Shenzhen (Grant No. ZDSYS201 40509172959989) and by Center for High Performance Computing and System Simulation, Pilot National Laboratory for Marine Science and Technology (Qingdao).

References

[1] E. W. Weisstein, "Finite volume method," From MathWorld–A Wolfram Web Resource. 2019. [Online]. Available: http://mathworld.wolfram.com/FiniteVolumeMethod.html
[2] C. Yang, W. Xue, H. Fu, et al., "10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics," in *Proc. Int. Conf. IEEE High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 57–68.
[3] B. Hamilton and C. J. Webb, "Room acoustics modelling using GPU-accelerated finite difference and finite volume methods on a face-centered cubic grid," in *Proc. Digit. Audio Effects Workshop*, 2013, pp. 336–343.
[4] M. Long and D. He, "Hydraulic erosion simulation using finite volume method on graphics processing unit," in *Proc. Int. Conf. Inf. Eng. Comput. Sci.*, 2009, pp. 1–4.

[5] M. Boubekeur, F. Benkhaldoun, and M. Seaid, "GPU accelerated finite volume methods for three-dimensional shallow water flows," in *Proc. Int. Conf. Finite Volumes Complex Appl.*, 2017, pp. 137–144.

[6] J. Xu, H. Fu, L. Gan, et al., "Generalized GPU acceleration for applications employing finite-volume methods," in *Proc. 16th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2016, pp. 126–135.

[7] A. Heinecke, A. Breuer, M. Bader, et al., "High order seismic simulations on the Intel Xeon Phi processor (knights landing)," in *Proc. Int. Conf. High Perform. Comput.*, 2016, pp. 343–362.

[8] A. Breuer, A. Heinecke, and Y. Cui, "EDGE: Extreme scale fused seismic simulations with the discontinuous Galerkin method," in *Proc. Int. Supercomput. Conf.*, 2017, pp. 41–60.

[9] L. Gan, H. Fu, O. Mencer, et al., "Chapter four-Data flow computing in geoscience applications," *Advances Comput.*, vol. 104, pp. 125–158, 2017.

[10] Z. Nagy, C. Nemes, A. Hiba, et al., "Accelerating unstructured finite volume computations on fieldgate arrays," *Concurrency Computation: Practice Experience*, vol. 26, no. 3, pp. 615–643, 2014.

[11] C. Yang, W. Xue, H. Fu, et al., "A peta-scalable CPU-GPU algorithm for global atmospheric simulations," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 1–12, 2013.

[12] M. J. Castro, S. Ortega, M. De la Asuncion, et al., "GPU computing for shallow water flow simulation based on finite volume schemes," *Comptes Rendus Mcanique*, vol. 339, no. 2/3, pp. 165–184, 2011.

[13] ACM Gordon Bell Prize Organizing Committee Chinese Research Team that Employs High Performance Computing to Understand Weather Patterns Wins 2016 ACM Gordon Bell Prize. 2016. [Online]. Available: https://www.acm.org/media-center/2016/november/gordon-bell-prize-2016

[14] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proc. 2nd Workshop Gen. Purpose Process. Graph. Process. Units*, 2009, pp. 79–84.

[15] Y. Chen, X. Cui, and H. Mei, "Large-scale FFT on GPU clusters," in *Proc. 24th ACM Int. Conf. Supercomput.*, 2010, pp. 315–324.

[16] A. Magni, C. Dubach, and M. F. P. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *Proc. Int. Conf. IEEE High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 1–11.

[17] S. W. Skillman, M. S. Warren, M. J. Turk, et al., "Dark sky simulations: Early data release," arXiv:1407.2600, 2014. [Online]. Available: https://core.ac.uk/display/25041440

[18] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound GPU applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 191–202.

[19] J. Langguth, N. Wu, J. Chai, et al., "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes," *J. Parallel Distrib. Comput.*, vol. 76, pp. 120–131, 2015.

[20] J. Langguth and X. Cai, "Heterogeneous CPU-GPU computing for the finite volume method on 3D unstructured meshes," in *Proc. 20th IEEE Int. Conf. Parallel Distrib. Syst.*, 2014, pp. 191–199.

[21] Nvidia Tesla Kepler Tuning Guide. 2019. [Online]. Available: http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html

[22] E. Lindholm, J. Nickolls, S. Oberman, et al., "NVIDIA tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar./Apr. 2008.

[23] L. Gan, H. Fu, C. Yang, et al., "A highly-efficient and green data flow engine for solving euler atmospheric equations," in *Proc. 24th Int. Conf. Field Programmable Logic Appl.*, 2014, pp. 1–6.

[24] L. Gan, H. Fu, W. Luk, et al., "Solving the global atmospheric equations through heterogeneous reconfigurable platforms," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 2, 2015, Art. no. 11.

[25] C. Yang, J. Cao, and X. C. Cai, "A fully implicit domain decomposition algorithm for shallow water equations on the cubed-sphere," *SIAM J. Sci. Comput.*, vol. 32, no. 1, pp. 418–438, 2010.

[26] E. F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Berlin, Germany: Springer Verlag, 1999, isbn 3-540-65966-8.

[27] C. Yang and X. C. Cai, "A scalable fully implicit compressible Euler solver for mesoscale nonhydrostatic simulation of atmospheric flows," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. S23–S47, 2014.

[28] J. E. Fromm, "A method for reducing dispersion in convective difference schemes," *J. Comput. Physics*, vol. 3, no. 2, pp. 176–189, 1968.

[29] B. P. Leonard, "A stable and accurate convective modelling procedure based on quadratic upstream interpolation," *Comput. Methods Appl. Mech. Eng.*, vol. 19, no. 1, pp. 59–98, 1979.

[30] Nvidia CUDA C Programming Guide. 2019. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[31] Nvidia Kepler Tuning Guide. 2019. [Online]. Available: http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html

[32] X. J. Yang, X. K. Liao, K. Lu, et al., "The TianHe-1A supercomputer: Its hardware and software," *J. Comput. Sci. Technol.*, vol. 26, no. 3, pp. 344–351, 2011.

[33] N. H. Sun, J. Xing, Z. G. Huo, et al., "Dawning Nebulae: A Peta-FLOPS supercomputer with a heterogeneous structure," *J. Comput. Sci. Technol.*, vol. 26, no. 3, pp. 352–362, 2011.

[34] Nvidia Tesla Pascal Architecture Whitepaper. [Online]. Available: http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[35] Oak Ridge National Laboratory. 2019. [Online]. Available: https://www.olcf.ornl.gov/summit/

[36] Lawrence Livermore National Laboratory. 2016. [Online]. Available: https://computation.llnl.gov/computers/sierra

[37] Green 500 List of Nov. 2017. 2017. [Online]. Available: https://www.top500.org/green500/lists/2017/11/

[38] J. Mielikainen, B. Huang, H. L. A. Huang, et al., "GPU acceleration of the updated Goddard shortwave radiation scheme in the weather research and forecasting (WRF) model," *IEEE J. Select. Topics Appl. Earth Observations Remote Sens.*, vol. 5, no. 2, pp. 555–562, Apr. 2012.

[39] R. Kelly, "GPU computing for atmospheric modeling," *Comput. Sci. Eng.*, vol. 12, no. 4, pp. 26–33, 2010.

[40] I. Demeshko, N. Maruyama, H. Tomita, et al., "Multi-GPU implementation of the NICAM atmospheric model," in *Proc. Eur. Conf. Parallel Process.*, 2012, pp. 175–184.

[41] T. Shimokawabe, T. Takaki, T. Endo, et al., "Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–11.

[42] W. Xue, C. Yang, H. Fu, et al., "Enabling and scaling a global shallow-water atmospheric model on Tianhe-2," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 745–754.

[43] W. Xue, C. Yang, H. Fu, et al., "Ultra-scalable CPU-MIC acceleration of mesoscale atmospheric modeling on Tianhe-2," *IEEE Trans. Comput.*, vol. 64, no. 8, pp. 2382–2393, Aug. 2015.

[44] Nvidia CUDA C Best Practices Guide. 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

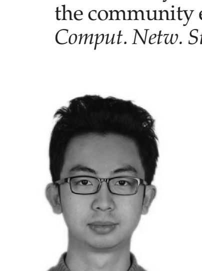
[45] A. Nguyen, N. Satish, J. Chhugani, et al., "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput.*, 2010, pp. 1–13.

[46] B. Hamilton, C. J. Webb, A. Gray, et al., "Large stencil operations for GPU-based 3-D acoustics simulations," in *Proc. Int. Conf. Digit. Audio Effects*, 2015, pp. 292–299.

[47] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, 2009.

[48] H. Fu, J. Liao, W. Xue, et al., "Refactoring and optimizing the community atmosphere model (CAM) on the sunway TaihuLight supercomputer," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, SC, 2016, pp. 969–980.

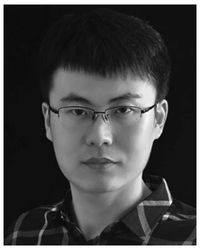
[49] P. H. Worley, A. P. Craig, and J. M. Dennis, et al., "Performance of the community earth system model," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–11.



Jingheng Xu received his PhD degree in the Department of Computer Science and Technology at Tsinghua University. His research interests include the state-of-art of computing architectures (such as Intel/IBM CPUs, Nvidia GPUs and Sunway processors) as well as high performance computing of scientific applications (such as atmospheric simulation and seismic modeling). His research mainly focuses on finding the best computing solutions based on the combination of architecture, algorithm and application. He has got the Chinese National Scholarship for Graduate Students.

Haohuan Fu received the PhD degree in computing from Imperial College London. He is a professor with the Ministry of Education Key Laboratory for Earth System Modeling, and the Department of Earth System Science, Tsinghua University, and the deputy director of the National Supercomputing Center in Wuxi. His research interests include high-performance computing in earth and environmental sciences, computer architectures, performance optimizations, and programming tools in parallel computing. He has been awarded the ACM Gordon Bell Prize (2016, 2017), Tsinghua-Inspur Computational Geosciences Youth Talent Award (2015), and the most significant paper award by FPL 2015. He is a member of the IEEE.

Wayne Luk received the doctorate degree in engineering and computing science from the University of Oxford. He is a professor of computer engineering with Imperial College London and the director of the EPSRC Centre for doctoral training in High Performance Embedded and Distributed Systems. His research focuses on theory and practice of customizing hardware and software for specific application domains, such as genomic data analysis, climate modeling, and computational finance. He is a fellow of the Royal Academy of Engineering, IEEE, and the British Computer Society.



Lin Gan is an assistant professor in the Department of Computer Science and Technology at Tsinghua University, and the assistant director of the National Supercomputing Center in Wuxi. His research interests include HPC solutions based on state-of-the-art platforms such as FPGAs, GPUs, and Sunway processors. Gan has a PhD in computer science from Tsinghua University. He is the recipient of the 2016 ACM Gordon Bell Prize, the 2017 ACM Gordon Bell Prize Finalist, the 2018 IEEE-CS TCHPC Early Career Researchers Award for Excellence in HPC, the 2015 IEEE FPL Most Significant Paper Award in 25 Years, and the 2017 Tsinghua-Inspur Computational Earth Science Young Researcher Award, etc. He is a member of IEEE.

Wen Shi is working toward the master's degree at Tsinghua University. She is familiar with parallel computer architecture and programming. Her research interests include accelerating scientific application on platforms such as POWER and GPU, focusing on improving the performance of kernel and the whole program.

Wei Xue received the PhD degree in electrical engineering from Tsinghua University. He is an associate professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include scientific computing and uncertainty quantification. He has received the ACM Gordon Bell Prize (2016, 2017) and the Tsinghua-Inspur Computational Geosciences Youth Talent Award. He is a member of the IEEE.

Chao Yang is a professor at Peking University and Peng Cheng Laboratory. He received his BS in mathematics from University of Science and Technology of China in 2002 and earned his PhD from Institute of Software, Chinese Academy Sciences in 2007. His research interests include numerical analysis and modeling, large-scale scientific computing, and parallel numerical software. He has received the 2016 ACM Gordon Bell Prize, the 2017 CAS Outstanding Science and Technology Achievement Prize, and the 2017 CCF-IEEE CS Young Computer Scientist Award. He is a member of IEEE, ACM and SIAM.

Yong Jiang received the BS and PhD degrees in computer science and technology from Tsinghua University, Beijing, P.R. China, in 1998 and 2002, respectively. In 2002, he joined the Graduate School at Shenzhen, Tsinghua University, P.R. China, and he is currently a professor with the Department of Computer Science and Technology, Tsinghua University, P.R. China. His research interests include computer architecture, network coding, and compressed sensing.

Conghui He received the BE degree in software engineering from Sun Yat-sen University. He is working toward the PhD degree in the Department of Computer Science and Technology, Tsinghua University. His research interests include GPU and FPGA-based solutions to exploration geophysics and financial applications, focusing on algorithmic development and performance optimizations.

Guangwen Yang received the PhD degree in computer science from Tsinghua University. He is a professor with the Department of Computer Science and Technology, Tsinghua University, and the director of the National Supercomputing Center in Wuxi. His research interests include parallel algorithms, cloud computing, and the earth system model. He has been awarded the ACM Gordon Bell Prize (2016, 2017). He is senior member of the CCF and a member of the IEEE.