Parallelizing a high-order WENO scheme for complicated flow structures on GPU and MIC

Liang DENG[1,a,*], Fang WANG[1,b] , Han-Li BAI[1,c] and Qing-Xin XU[1,d]

[1] Computational Aerodynamics Institute, China Aerodynamics Research Development Center

No. 278, west section of Jianmen Road, Fucheng District, Mianyang City, Sichuan Province, China

[a]denglnudt@126.com, [b]wwangfang@sina.com, [c]baihanli@163.com, [d]xuqingxin@cardc.cn

Keywords: High-order, WENO, Fermi GPU, Kepler GPU, Intel MIC coprocessor, Optimization techniques.

Abstract. As a conservative, high-order accurate, shock-capturing method, weighted essentially non-oscillatory (WENO) scheme have been widely used to effectively resolve complicated flow structures in computational fluid dynamics (CFD) simulations. However, using a high-order WENO scheme can be highly time-consuming, which greatly limits the CFD application's performance efficiency. In this paper, we present various parallel strategies base on the latest many-core platform such as NVIDIA Fermi GPU, NVIDIA Kepler GPU and Intel MIC coprocessor to accelerate a high-order WENO scheme. Comparison analysis of the two generations GPUs between Fermi and Kepler, and cross-platform performance analysis (focusing on Kepler GPU and MIC) are also detailed discussed. The experiments show that the Kepler GPU offers a clear advantage in contrast to the previous Fermi GPU maintaining exactly the same source code. Furthermore, while Kepler GPU can be several times faster than MIC without utilizing the increasingly available SIMD computing power on Vector Processing Unit (VPU), MIC can provide the computing capability equivalent to Kepler GPU when VPU is utilized. Our implementations and optimization techniques can serve as case studies for paralleling high-order schemes on many-core architectures.

Introduction

High-order schemes have received considerable attention in computational fluid dynamics (CFD) simulations as they can resolve complicated flow structures and provide much more accurate flow details while utilizing less grid points [1]. As a kind of high-order schemes, the WENO schemes is widely used in large eddy simulations (LES) [2] and direct numerical simulations (DNS) [3]. However, the high requirement of computing power imposed by WENO schemes greatly limit the CFD application's performance and power efficiency. As a result, parallelizing the WENO schemes on high performance computes is necessary.

In recent years by the heterogeneous systems can achieve higher floating-point performance and lower power consumption than traditional CPU-only systems, there opens new horizons to utilize the accelerators, such as the Graphics Processing Units (GPUs) and the Many Integrated Core (MIC) coprocessors, for building high performance computes. In the latest TOP500 [4] list, there are 50 high-end supercomputer systems used NVIDIA GPUs, and 25 systems with Intel MIC coprocessors. Among them, the No. 1 supercomputer TianHe-2 designed by China's National University of Defense Technology (NUDT), which consists of 48,000 Intel Xeon Phi 31SP MIC coprocessors (57 cores per coprocessor), and the No. 2 Titan developed by Oak Ridge National Laboratory contains 18,688 NVIDIA K20x GPUs (14 Streaming Multiprocessors per GPU).

Since GPUs with the Compute Unified Device Architecture (CUDA) programming model support were first launched by NVIDIA Corporation in 2006, several groups have applied them to reduce the computational time of the WENO schemes for CFD simulations. Athanasios S. Antoniou et al. [5] presented a highly accelerated implementation of the finite-difference WENO scheme for large-scale simulations and reported a speedup of 53 on the average for the several mesh sizes compared to sequential execution. Michael Griebel et al. [6] implemented the 5th order WENO scheme for a

two-phase solver on CUDA-enabled GPUs and observed an impressive speedup of 69.6 on eight GPUs/CPUs in contrast to a single CPU. Diego Rossinelli et al. [7] presented an adaptive finite volume solver for multiphase compressible flows based on the 5th-order WENO scheme on a heterogeneous platform using OpenCL, and observed an overall speedup of 34 by employing six GPUs compared to the single-core execution. Chen Meng et al. [8] developed a three-dimensional 5th-order WENO scheme GPU code for large-scale cosmological hydrodynamic flow simulations, and achieved a 12-19 times speedup in contrast to a single CPU. Vahid Esfahanian et al. [9] focused on the use of GPUs for solving some hyperbolic equations using the 5th-order WENO scheme and showed that a simple GPU can reach to several hundreds times speedup over a single-core CPU. Lin Fu et al. [10] ported a multi-block viscous flow solver for steady and unsteady turbulent flows onto GPUs by WENO 5th-order reconstruction scheme and obtained a speedup of 32.6× over a single-core CPU. Konstantinos I. Karantasis et al. [11] evaluated the performance of a high-order WENO scheme for the simulation of compressible flows on a GPU cluster, and got about a speedup of 90 on eight GPUs compared to the sequential execution. Hossein Mahmoodi Darian et al. [12] detailed studied on the GPU implementation of up to 9th-order WENO schemes for the multi-dimensional Euler equations and reported that the speedups roughly reach 65 and 105 for the GTX 550 and GTX 480 GPUs, respectively.

However, there are three major problems have not been fully addressed thus far. Firstly, while the performance gains of GPUs are impressive in many of the pioneering studies, a fair performance comparison has rarely been reported where the multi-core CPUs is also parallelized to make full use of the available hardware resource. Secondly, the new generation Kepler GPUs is increasing their number of cores and cache sizes, but it is unclear that the different generations of GPUs affect end-to-end performance for the same source code. Thirdly, the Intel MIC coprocessor based on x86 core architecture as a new type of hardware accelerator is a strong competitor of the NVIDIA GPU, but its application in CFD simulations used WENO schemes is still absent in the current literature.

To address these problems, we have been implementing efficient parallel a high-order WENO code for complicated flow structures on four platforms (Ivy Bridge CPUs, Fermi GPUs, Kepler GPUs and the Intel MIC). As to the GPU platforms, we carefully consider memory use, occupancy and concurrency to explore the performance optimization opportunities for our WENO code. For the Ivy Bridge CPUs and MIC, we employ a series of optimization techniques to maximize the parallelism and data locality and achieve the best performance. We also detailed comparison analysis the two generations GPUs between Fermi and Kepler, and conduct cross-platform performance analysis (focusing on Kepler GPU and MIC) to provide a few suggestions to help developers who need to choose the best accelerators for their specific applications.

The remainder of this paper is arranged as the following. Section 2 briefly introduces the Euler equations and the formulation of the WENO schemes. Section 3 details the GPU parallelization and optimization techniques, and MIC are demonstrated in Section 4. Experimental results are presented and analyzed in Section 5. Finally, we give the conclusion in Section 6.

Euler Equations and WENO Schemes

Euler Equations.

The two dimensional Euler equations of compressible gas dynamics in strong conservation form are:

$$
\begin{aligned}
&u_t + f(u)_x + \mathrm{g}(u)_y = 0,\\
&u = (\rho, \rho u_1, \rho u_2, E),\\
&f(u) = \left(\rho u_1, \rho u_1^2 + p, \rho u_1 u_2, u_1 (E + p)\right),\\
&g(u) = \left(\rho u_2, \rho u_1 u_2, \rho u_2^2 + p, u_2 (E + p)\right),\\
&E = \frac{p}{\gamma - 1} + \frac{1}{2}\rho\left(u_1^2 + u_2^2\right).
\end{aligned} \tag{1}
$$

Here ρ is the density, (u_1, u_2) is the velocity, E is the total energy, p is the pressure, γ is the ratio of specific heats. For the nonlinear of Euler equations, its corresponding Jacobian matrices and their distinct eigenvalues are defined as follows:

$$
\begin{aligned}
&A = \frac{\partial f}{\partial u}: \quad \lambda_1^x = u_1 - c, \quad \lambda_2^x = \lambda_3^x = u_1, \quad \lambda_4^x = u_1 + c\ ;\\
&B = \frac{\partial g}{\partial u}: \quad \lambda_1^y = u_2 - c, \quad \lambda_2^y = \lambda_3^y = u_2, \quad \lambda_4^y = u_2 + c.
\end{aligned} \tag{2}
$$

The distinct eigenvalues of the Jacobian are all real, and represent the wave speeds of each direction, where $c = \sqrt{\gamma p/\rho}$ is the sound speed. To handle both the negative and positive wave speeds, we choose the characteristic-wise Lax-Friedrichs flux splitting [13], which is defined as:

$$
\begin{aligned}
&f(u) = f^+(u) + f^-(u) \qquad f^\pm(u) = \frac{1}{2}\left(f(u) \pm \lambda_{\max}^x u\right);\\
&g(u) = g^+(u) + g^-(u) \qquad g^\pm(u) = \frac{1}{2}\left(g(u) \pm \lambda_{\max}^y u\right)\ .
\end{aligned} \tag{3}
$$

where $\lambda_{\max}^x = \max\left(|u_1| + c\right)$ and $\lambda_{\max}^y = \max\left(|u_2| + c\right)$ denote the maximum of the eigenvalues in x and y directions of the whole solution domain, respectively.

WENO Schemes.

The WENO schemes for the semi-discrete conservative form of the Euler equations are written as follows:

$$
\begin{aligned}
&\frac{du_{ij}(t)}{dt} = -\frac{1}{\Delta x}\left(\hat{f}_{i+1/2,j} - \hat{f}_{i-1/2,j}\right) - \frac{1}{\Delta y}\left(\hat{g}_{i,j+1/2} - \hat{g}_{i,j-1/2}\right),\\
&f_{i+1/2,j} = f^-{}_{i+1/2,j} + f^+{}_{i+1/2,j}, \qquad g_{i,j+1/2} = g^-{}_{i,j+1/2} + g^+{}_{i,j+1/2}\ .
\end{aligned} \tag{4}
$$

where the numerical flux $\hat{f}_{i+1/2,j} = \hat{f}\left(u_{i-p,j}, \cdots, u_{i+q,j}\right)$ is consistent with the physical flux $\hat{f}(u, \cdots, u) = f(u)$. Simply, $f(u)_x$ at (x_i, y_j) is approximated along the line $y = y_j$ by a conservative flux difference.

$$
f(u)_x\big|_{x=x_i} \approx \frac{1}{\Delta x}\left(\hat{f}_{i+1/2} - \hat{f}_{i-1/2}\right). \tag{5}
$$

The WENO scheme we use in this paper are the 9th-order finite difference version developed by Balsara and Shu in [14]. Therefore, the numerical flux $\hat{f}_{i+1/2}$ depends on 9 point values $f(u_{kj})$,

$k = i-4,...,i+5$. Fig. 1 shows the grid points involved in computing the value of the WENO9 flux $\hat{f}_{i+1/2}$.

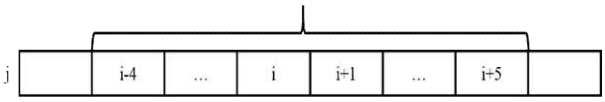

Fig.1 The points involved in computing $\hat{f}_{i+1/2}$ for the 9th-order WENO scheme

The time discretization is via the third-order TVD Runge-Kutta method developed in [15]

$$\begin{aligned} u^{(0)} &= u^n, \\ u^{(1)} &= u^{(0)} + \triangle tL\left(u^{(0)}\right), \\ u^{(2)} &= \frac{3}{4}u^{(0)} + \frac{1}{4}u^{(1)} + \frac{1}{4}\triangle tL\left(u^{(1)}\right), \\ u^{(3)} &= \frac{1}{3}u^{(0)} + \frac{2}{3}u^{(2)} + \frac{2}{3}\triangle tL\left(u^{(2)}\right), \\ u^{(3)} &= u^{n+1}. \end{aligned} \tag{6}$$

where the operator L is defined by

$$L(u) = -\frac{1}{\Delta x}\left(\hat{f}_{i+1/2,j} - \hat{f}_{i-1/2,j}\right) - \frac{1}{\Delta y}\left(\hat{g}_{i,j+1/2} - \hat{g}_{i,j-1/2}\right). \tag{7}$$

GPU Parallelization and Optimization

Traditionally, the WENO schemes parallel computing on CPUs is based on domain decomposition using MPI and OpenMP programming, in which each CPU core processes one or more portion. However, there are architectural disparities between CPUs and GPUs. For example, GPUs can execute a large number of threads concurrently and hide memory latency by efficient thread switching. Therefore we need to make full use of millions of lightweight threads on GPUs when designing GPU programs. Considering CUDA C programming features, we use one-dimensional arrays to represent multi-dimensional indexing of the variables for both CPU and GPU codes. Take the pressure p as example, its indexing is as follows:

$$\begin{aligned} &2D\ computation\ domain\ \left(N=\left(IIX+2\times MT\text{-}1\right)\times\left(IIY+2\times MT\text{-}1\right)\right): \\ &\qquad p_{i,j} = p[l], \qquad l = \left(IIY+2\times MT\text{-}1\right)\times i + j. \end{aligned} \tag{8}$$

where i and j are the computation domain point indexes in x and y directions, respectively, IIX and IIY is the number of cell in the x and y directions, respectively, MT denotes the number of node stencil of WENO scheme, and l is the equivalent index used in the codes.

For the GPU fine-grained parallelism, there are two kinds of CUDA threadblock configurations (2D and 1D) according to data dependency among cells in computation domain. If there is no data dependency in the solution process, we use a 2D configuration (each CUDA thread calculates a cell independently). Fig. 2 illustrates 2D CUDA threadblock configuration. The 2D thread block is configured as (nx, ny) and the size of GPU gird is $\left(\left[(IIX+2*MT-1)/nx\right],\left[(IIY+2*MT-1)/ny\right]\right)$, where $[x]$ is the minimum integer that is larger than x, and the corresponding relation the CUDA thread and cell is as follows:

$$\begin{aligned} i &= threadIdx.x + blockDim.x * blockIdx.x, \\ j &= threadIdx.y + blockDim.y * blockIdx.y. \end{aligned} \tag{9}$$

Since the CUDA model is lack of efficient global synchronization mechanism, if there is data dependency in the x direction, we use a 1D configuration (each CUDA thread computes all the cells in the x direction). The 1D thread block is configured as $(1, ny)$ and the size of GPU gird is $\left(1, \left\lceil (IIY + 2*MT - 1)/ny \right\rceil\right)$.

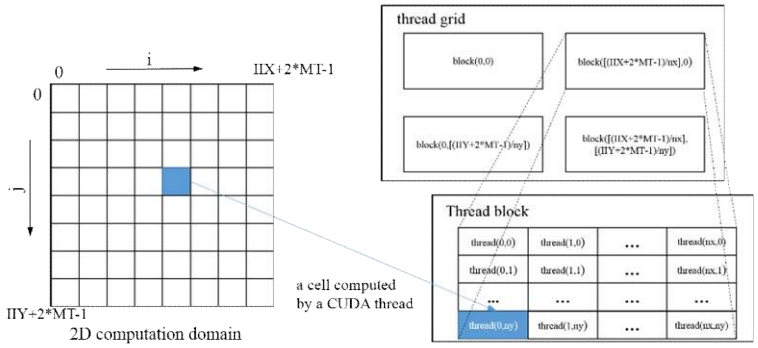

Fig. 2. 2D CUDA thread blocks configuration.

After preliminary parallel GPU implementation of our WENO code, we focus on three issues: memory usage, occupancy and concurrency to improve the performance.

Memory Use.

To improve the performance, it is very important to make full use of the GPU's memory hierarchy. Therefore, based on the characteristics of data structures in our code, flow parameter and standard value such as IIX, IIY and MT, are stored in constant memory to reduce the data transfers between CPU and GPU, and flow variables such as the density ρ, the velocity (u_1, u_2), the total energy E, the pressure p, are stored in global memory to benefit the data exchanges between CPU and GPU. The shared memory is only used to buffer the temporary data coming from the global memory at the computation of largest eigenvalues $\lambda_{\max}$ in each direction where the largest eigenvalue coming from each thread are collected and summed up together using parallel reduction algorithm [16]. All data structures are stored contiguously for one dimension which is good for coalescing global memory accesses.

Occupancy.

Occupancy is the ratio of the number of active warps per multiprocessor to maximum number of possible active warps [17]. The maximum number of possible active warps in each multiprocessor is constant, which is only decided by the computing capacity of GPU. Through the NVIDIA occupancy spreadsheet [18], we determine the proper number of threads per block and the number of blocks per grid to maximize occupancy. In our code, when the number of threads per block is 32, x direction is 8 and y direction is 4, the optimal occupancy is 0.333 and the performance is best.

Concurrency.

Concurrency is generally the ability of a system to perform multiple operations simultaneously [19]. The greater concurrency of application, the better performance to obtain on GPU. Due to the solution process of WENO schemes, we firstly place computation of largest eigenvalues in each direction, local computation of matrices of left and right eigenvectors, flux splitting and calculation of right-side hand (RHS) values in a single 1D kernel. The concurrency is very small and do not make full use of millions of lightweight threads on GPUs, the performance is poor. Therefore, we decompose a large 1D kernel to several small kernels and use a 2D configuration to maximize the concurrency of our code. By carefully investigating the data dependency, 1D kernel fx for the inviscid flux of x direction is decomposed into six 2D kernels as Fig. 3 shows: fx_1 is used to compute largest eigenvalues in each direction am; fx_2 is used to compute matrices of left and right eigenvectors evl and evr; fx_3 is used to compute the negative and positive of flux terms $gg1$ and $gg2$; fx_4 is used

to compute the negative of half-node flux terms ff ; fx_5 is used to compute the positive of half-node flux terms fh ; fx_6 is used to compute the right side hand rhs .

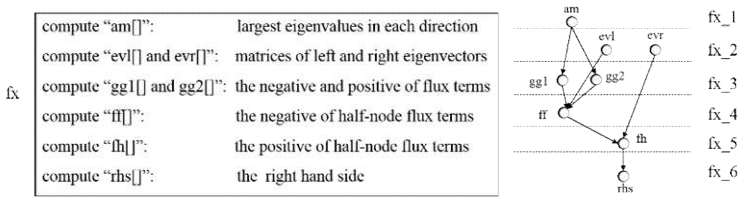

Fig.3. Data dependency in x -direction inviscid flux term computation and kernel decomposition.

After the aforementioned optimization techniques are implemented, the procedure of a full GPU version of our WENO code is illustrated as Fig. 4. To remove redundant data transfers between CPU and GPU, there are only two CPU/GPU data transfers, one is from CPU to GPU, and the other is from GPU to CPU. Computation of inviscid flux terms is split into two kernels, which are fx for calculating the finite difference in x direction used WENO scheme, gy for computing the finite difference in y direction used WENO scheme. After getting the RHS, calculating the new values on computation domain with Runge-Kutta method. All of the solution process of Euler equations is calculated on the GPU, the CPU only executes termination condition judgment.

Finally, we visualize flow and analyze the output results during the post-processing step.

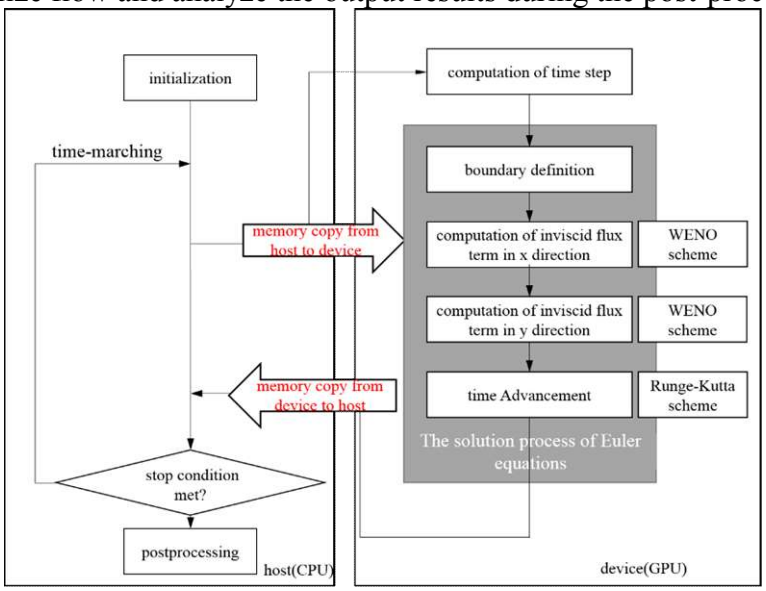

Fig. 4. An illustration of the complete flow chart of our WENO code.

MIC Parallelization and Optimization

Due to the emphasis on common programming languages, models, tools across the Ivy Bridge CPU and the Intel MIC coprocessor, it is easy to port our WENO code from the traditional CPU to the newly available Intel MIC, and the similar optimization techniques can be employed for both to enable the best performance.

The solution process of Euler equations can take 98% of the total CPU computation time, thus we choose the "native" mode of MIC where the WENO code runs natively on MIC. The key of performance optimization on MIC is maximizing parallel computations and minimizing data movement. These optimization techniques generally fall into three categories: intra-node parallelism for scaling, intra-core parallelism to exploit SIMD (single-instruction multiple-data) mechanism and improve data locality in the on-chip cache.

Intra-node Parallelism.

In this paper, we choose OpenMP as the programming model for intra-node parallelism. For OpenMP, the optimization techniques are the proper thread number configuration and the suitable thread affinity strategy.

Making full use of the available hardware resource, it is usually a practical way to set the number of threads corresponding to the number of physical cores in each devices. For MIC, there are 57 physical cores placed in a ring interconnect. Additionally, four hardware threads on each MIC core resulting in 228 hardware threads available on a single MIC coprocessor. The hardware thread on MIC is designed for in-order execution that typically requires 2-4 threads per core for optimal performance [20]. Therefore, we set the number of threads as 24 and 171 for the Ivy Bridge CPU and MIC, respectively.

The suitable thread affinity can minimize the transitions from one core to another, improve cache efficiency and make each core load balancing. Fig. 5 shows the three thread affinity strategies.The first strategy, called compact, binds threads to the next free thread context. That is, all four contexts in a physical MIC core are used before threads are placed in the contexts of another core. The second strategy, called scatter, the new thread will firstly be allocated to the core that has the lightest load. The last strategy, balanced, is a comprehensive one. Threads are balanced among cores and subsequent threadIDs are assigned to neighbor contexts [21]. The most effective thread affinity strategy is compact for Ivy Bridge CPU and balanced for MIC in our WENO code

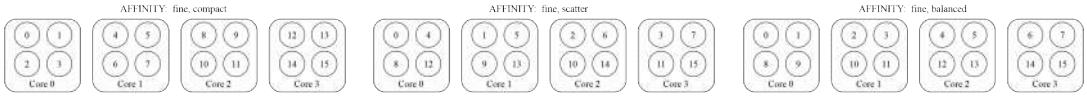

Fig. 5. The three thread affinity strategies. Example mappings of 16 threads to physical cores on a 4-way multithreading device.

Intra-core Parallelism.

Intra-core parallelism to efficiently exploit SIMD mechanism is one of the most important aspects in achieving high performance of the application code running on Intel MIC coprocessors [22]. MIC uses a 512-bit vector unit, which means 16 single-precision or 8 double-precision floating point operations can be executed at one time. We use serval SIMD vectorization techniques such as auto vectorization counting on the compiler, pragmas to assist vectorization(#pragma vector always can be used to instruct the compiler to always vectorise outer loops and #pragma vector nontemporal can use the streaming store capability to achieve higher effective memory bandwidth) and memory alignment optimization by using "_mm_malloc()" to ensure the base address of the array is aligned and using padding to ensure the address of the first element on each row is also aligned for achieving a more efficient intra-core parallelism.

Data Locality.

Normally, the two tiers parallelism can offer a significant performance improvement. However, the higher parallelism, the larger amount of data is required during a short period of time, which may exceed the peak memory bandwidth. Since the existence of cache whose access is significantly shorter than memory access, it is important to find techniques that can help reduce the memory access pressure by reusing cached data. As a kind of well-known optimization technique, cache blocking can help avoid memory bandwidth bottlenecks to exploit data that remains in the cache from recent, previous iterations. Fig. 6(a) shows an example of an iteration row by row is updated. For cache blocking the 2D computation domain is partitioned into block in x direction as shown in Fig. 6(b). The best configuration of block size is able to theoretically computed, based on the data size accessed with each iteration and cache details. Here we make an automated search over all the possible size in one and two dimensions to experimentally verify, respectively.

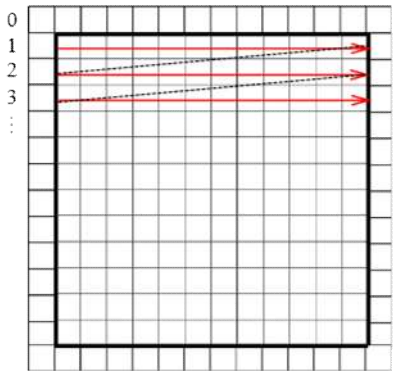

Fig. 6(a). Row by row computation domain traversal.

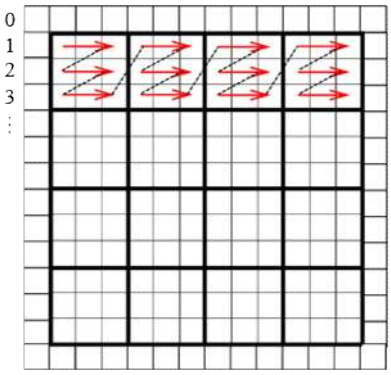

Fig. 6(b). Cache blocking is applied, one block after another is updated.

Results

Computational Examples.

The double Mach reflection problem was used to evaluate the performance of GPU and MIC implementation of the high-order WENO scheme. The double Mach reflection problem was initially proposed and studied in detail by Woodward and Colella [23]. The computation domain is chosen to be $[0,4]\times[0,1]$. There is an initial right-moving Mach 10 shock at $x=1/6, y=0$, which makes a 60° angle with the x-axis. The reflecting wall lies at the bottom of the computation domain starting from $x=1/6$ to $x=4$. For the bottom boundary, the exact post-shock condition is imposed for the region $0\le x\le 1/6$, and a solid wall boundary condition is used for the rest. At the top boundary, the solution is set to describe the exact motion of the Mach 10 shock. The left boundary is set as the inflow boundary condition, while the right boundary is set as outflow boundary condition. The computed density contours on different grid sizes are displayed in Fig. 7. In CPU, GPU and MIC computations, the same results are obtained.

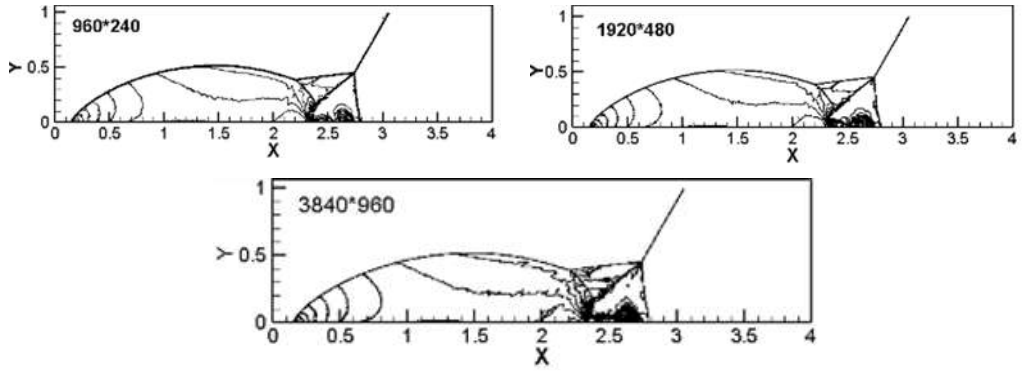

Fig. 7. Computed density contours using a high-order WENO scheme for different size of double Mach reflection problem.

Performance Evaluation on GPU and MIC.

To evaluate the performance of our implementation, we have conducted experiments using diferent grid sizes on the lastest multi-core and many-core architectures including Intel Ivy Bridge CPU, the NVIDIA Fermi M2050 GPU, the NVIDIA Kepler K20c GPU and the Intel MIC coprocessor. The important hardware features of these architectures are listed in Table 1. The GPU code is implemented using CUDA C and the CUDA vesion is 5.0. All the code is compied with –O2 option. Double precison is used in all computations.

Table 1. List of related parameters of architecture for experiments.

Architecture	Ivy Bridge	Fermi	Kepler	MIC
Frequency (GHz)	2.20	1.15	0.73	1.01
#of cores	2*12	14*32	14*64	57
SIMD width single	8	32/warp	32/warp	16
L1 Cache (kB)	32/core	64/SM	64/SM	32/core
L2 Cache (kB)	256/socket	768/card	1536/card	512/core
L3 Cache (MB)	24/socket	0	0	0
Peak performance double (Gflops)	2*105.6	394.83	1320	921.12
Theoretical memory bandwidth (GB/s)	2*59.7	144	250	240

In Fig.3, we present the results obtained with Intel MIC in three different thread affinity strategies. We find that the balanced specification works better with fewer threads, and the best thread number configuration is 171 when three threads per core on MIC. Fig. 4 shows the results for kernel decomposition optimization to improve the concurrency of our code. We achieve an 8.8 times speedup for the x direction, but for the y direction, the speedup is only 4.0. This is because the decomposition in the x direction is good for coalescing global memory accesses.

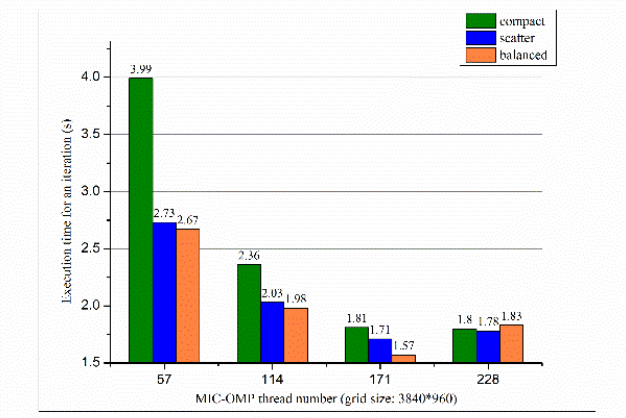

Fig. 3. Execution time(s) for an iteration for fixed grid size using a MIC in native mode: scaling with respect to the number of OpenMP threads using different affinity specifications.

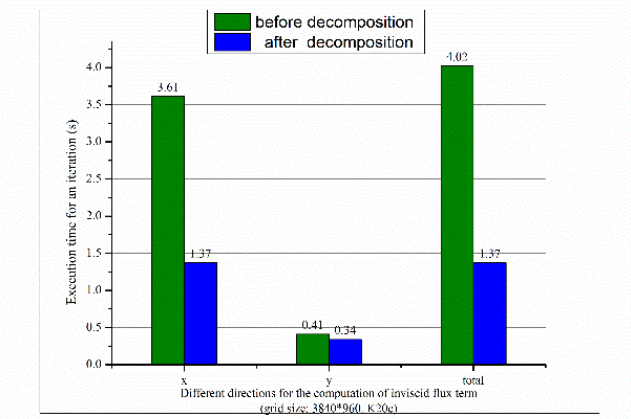

Fig. 4. Execution time(s) for an iteration for fixed grid size using a K20c GPU: performance comparison for before and after decomposition

From Table 2, we see that signficant improvement in performance over the serial code on CPU are archieve for CPU, Fermi M2050 GPU, Kepler K20c GPU and MIC after a series of optimizations. The SC of MIC is 15.66 times lower than Ivy Bridge CPU. The Kepler GPU offers 1.26 times speedup in contrast to the previous Fermi GPU maintaining exactly the same source code. Furthermore, while Kepler GPU can be 3.12 times faster than MIC without utilizing the increasingly available SIMD computing power on Vector Processing Unit (VPU), MIC can provide the computing capability equivalent to Kepler GPU when VPU is utilized. However, when GPU and MIC compare with Ivy Bridge CPU parallelized to make full use of the available hardware resource, we are surprised to find that the speedup is low. This is mainly because poor vectorization. Though vectorization also affects performance on CPU as well, we find the SIMD speedup is only 2.86 that is far from tapping the capaility of 512-bit wide vector on MIC from Table 3. Large L3 cache might be another reason which can hold more data on cache.

Table 2. Execution time(s) for an iteration for fixed grid size (3840*960) using different processors. "SC" means the performance of the serial code. "MC" means the performance of intra-node parallelism using OpenMP. "SIMD" means using vectorization for intra-core parallelism. "BLC" means using cache blocking to improve data locality in the on-chip cache.

Configuration	Ivy Bridge CPU	Fermi M2050	Kepler K20c	MIC
SC	17.79			278.67
SC+SIMD	11.41			118.80
MC	1.49			4.49
MC+SIMD	1.20			1.57
MC+SIMD+BLC (overall)	1.12	1.82	1.44	1.45

The speedups for scaling, SIMD and data locality are shown in Table 3. We can observe that the speedups on MIC for SC over MC and SC+SIMD over MC+SIMD using 171 threads are 62.06× and 75.67×, respectively. The speedups on CPU for SC over MC and SC+SIMD over MC+SIMD using 171 threads are 11.94× and 9.51×, respectively. It is disappointed to see that the speedups for SIMD is very low, thus tapping the available SIMD computing power on VPU using SIMD intrinsic programming is future work. The overall speedup of CPU and MIC is 15.88× and 192.19×, respectively.

Table 3. List of speedup for fixed grid size (3840*960) using CPU and MIC.

	Configuration	Ivy Bridge CPU	MIC
Scaling	SC/MC	11.94	62.06
Speedup	(SC+SIMD)/ MC+SIMD	9.51	75.67
SIMD Speedup	SC/(SC+SIMD)	1.56	2.35
	MC/(MC+SIMD)	1.24	2.86
Data locality speedup	MC+SIMD/(MC+SIMD +BLC)	1.07	1.08
Overall Speedup	SC/(MC+SIMD+BLC)	15.88	192.19

In Table 4, we present the speedups of an Ivy Bridge CPU using 24 threads over a single-core CPU, a Fermi M2050 over a single-core CPU, a Kepler K20c over a single-core CPU and a MIC using 171 threads over a single-core CPU for different grid sizes. We see that Kepler GPU and MIC can achieve a speedup of more than 10 over a single-core CPU. We observe performance degradation for Ivy Bridge CPU when the gird size is increased. This can be explained by the fact that the performance improvement of L3 cache is flooded by the grid size increases. We also find slight performance improvement for both GPU and MIC. This is because higher workloads can better overlap the computation and memory accesses for GPU and MIC. If the memory of GPU and MIC is large enough, their performance may be better than the Ivy Bridge CPU.

Table 4. List of speedup over a single-core CPU for different grid sizes using different processors.

Grid	a single-core CPU	Ivy Bridge CPU	Fermi M2050	Kepler K20c	MIC
960*240	1.00	26.00	8.81	10.83	10.40
1920*480	1.00	18.91	9.58	12.24	11.45
3840*960	1.00	15.88	9.77	12.35	12.27

Conclusion

In this paper, we proposed methodologies to implement efficient parallel a high-order CFD simulation using WENO scheme for complicated flow structures on Ivy Bridge CPUs, Fermi GPUs, Kepler GPUs and the Intel MIC. For the GPU platforms, we carefully consider memory use, occupancy and concurrency to explore the performance optimization opportunities for our WENO code. For the Ivy Bridge CPUs and MIC, we employ a series of optimization techniques such as auto

vectorization counting on the compiler, pragmas to assist vectorization, and cache blocking to achieve the best performance.

Besides, we have evaluated the performance of our parallel implementation of the WENO code. GPU and MIC can achieve a speedup of more than 10 over a single-core CPU. The Kepler GPU offers 1.26 times speedup in contrast to the previous Fermi GPU without special tuning. Furthermore, while Kepler GPU can be 3.12 times faster than MIC without utilizing the increasingly available SIMD computing power on VPU, MIC can provide the computing capability equivalent to Kepler GPU when VPU is utilized.

For future work, we plan to tap the capaility of VPU using SIMD intrinsic programming to improve the implementation on MIC.

References

[1] Wang Z J. High-order methods for the Euler and Navier-Stokes equations on unstructured grids[J]. Progress in Aerospace Sciences, 2007, 43(1-3):1–41.

[2] Grube N. Assessment of WENO Methods with Shock-Confining Filtering for LES of Compressible Turbulence [J]. Aiaa Journal, 2007.

[3] Martín M P, Taylor E M, Wu M, et al. A bandwidth-optimized WENO scheme for the effective direct numerical simulation of compressible turbulence [J]. Journal of Computational Physics, 2006, 220(1):270-289.

[4] Information on http://www.top500.org.

[5] Antoniou A S, Karantasis K I, Polychronopoulos E D, et al. Acceleration of a finite-difference WENO scheme for large-scale simulations on many-core architectures[C]//48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, Orlando, FL, Jan. 2010: 4-7.

[6] Griebel M, Zaspel P. A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations [J]. Computer Science - Research and Development, 2010, 25(1-2):65-73.

[7] Rossinelli D, Hejazialhosseini B, Spampinato D G, et al. Multicore/multi-gpu accelerated simulations of multiphase compressible flows using wavelet adapted grids [J]. SIAM Journal on Scientific Computing, 2011, 33(2): 512-540.

[8] Meng C, Wang L, Cao Z, et al. Acceleration of a High Order Finite-Difference WENO Scheme for Large-Scale Cosmological Simulations on GPU[C]//Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International. IEEE, 2013: 2071-2078.

[9] Esfahanian V, Darian H M, Gohari S M I. Assessment of WENO schemes for numerical simulation of some hyperbolic equations using GPU [J]. Computers & Fluids, 2013, 80: 260-268.

[10] Fu L, Gao Z, Xu K, et al. A multi-block viscous flow solver based on GPU parallel methodology [J]. Computers & Fluids, 2014, 95(21):19–39.

[11] Karantasis K I, Polychronopoulos E D, Ekaterinaris J A. High order accurate simulation of compressible flows on GPU clusters over Software Distributed Shared Memory [J]. Computers & Fluids, 2014, 93: 18-29.

[12] Darian H M, Esfahanian V. Assessment of WENO schemes for multi-dimensional Euler equations using GPU [J]. International Journal for Numerical Methods in Fluids, 2014, 76(12): 961-981.

[13] Jiang G S, Shu C W. Efficient implementation of weighted ENO schemes[R]. INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING HAMPTON VA, 1995.

[14] Balsara D S, Shu C W. Monotonicity preserving weighted essentially non-oscillatory schemes with increasingly high order of accuracy [J]. Journal of Computational Physics, 2000, 160(2): 405-452.

[15] Shu C W, Osher S. Efficient implementation of essentially non-oscillatory shock-capturing schemes[J]. Journal of Computational Physics, 1988, 77(2): 439-471.

[16] Harris M. Optimizing parallel reduction in CUDA[J]. NVIDIA Developer Technology, 2007, 2(4).

[17] NVIDIA CUDA. CUDA C best practices guide. NVIDIA; 2010.

[18] NVIDIA, 2013b. GPU occupancy calculator [EB/OL]. http://developer.download.nvidia.com/compute/cuda/.

[19]Rennich. S. 2011. CUDA C/C++ streams and concurrency [EB/OL]. http://on-demand.gputechconf.com/gtc-express/2011/presentations/

[20] Jeffers J, Reinders J. Intel Xeon Phi coprocessor high-performance programming[M]. Newnes, 2013.

[21] Teodoro G, Kurc T, Andrade G, et al. Performance Analysis and Efficient Execution on Systems with multi-core CPUs, GPUs and MICs[J]. arXiv preprint arXiv:1505.03819, 2015.

[22] Tian X, Saito H, Preis S V, et al. Effective SIMD Vectorization for Intel Xeon Phi Coprocessors[J]. Scientific Programming, 2015, 501: 269764.

[23] P. Woodward, P. Colella, The numerical simulation of two-dimensional fluid flow with strong shocks, Journal of Computational Physics 54 (1984) 115–173.