SIAM J. SCI. COMPUT.
Vol. 33, No. 2, pp. 512–540
© 2011 Society for Industrial and Applied Mathematics

MULTICORE/MULTI-GPU ACCELERATED SIMULATIONS OF MULTIPHASE COMPRESSIBLE FLOWS USING WAVELET ADAPTED GRIDS*

DIEGO ROSSINELLI†, BABAK HEJAZIALHOSSEINI†, DANIELE G. SPAMPINATO†, AND PETROS KOUMOUTSAKOS†

Abstract. We present a computational method of coupling average interpolating wavelets with high-order finite volume schemes and its implementation on heterogeneous computer architectures for the simulation of multiphase compressible flows. The method is implemented to take advantage of the parallel computing capabilities of emerging heterogeneous multicore/multi-GPU architectures. A highly efficient parallel implementation is achieved by introducing the concept of wavelet blocks, exploiting the task-based parallelism for CPU cores, and by managing asynchronously an array of GPUs by means of OpenCL. We investigate the comparative accuracy of the GPU and CPU based simulations and analyze their discrepancy for two-dimensional simulations of shock-bubble interaction and Richtmeyer–Meshkov instability. The results indicate that the accuracy of the GPU/CPU heterogeneous solver is competitive with the one that uses exclusively the CPU cores. We report the performance improvements by employing up to 12 cores and 6 GPUs compared to the single-core execution. For the simulation of the shock-bubble interaction at Mach 3 with two million grid points, we observe a 100-fold speedup for the heterogeneous part and an overall speedup of 34.

Key words. GPU, compressible flow, wavelets, multiresolution, adaptive grid, multiphase, multicore architectures

AMS subject classifications. 76T10, 76M12, 65T60, 68W10, 65M50

DOI. 10.1137/100795930

1. Introduction.

Heterogeneous, CPU/GPU based computer architectures provide us today with unprecedented computational power for the simulation of complex physical systems. We need, however, to develop suitable methods and algorithms to harness these capabilities. We can only stress the importance of an integrative development of computational methods and software that takes into consideration the hardware architecture. In turn, synchronizing the developments of methods, software and hardware holds the promise for even larger advances in all components involved.

A traditional testbed for the development of computational methods is fluid mechanics. The equations of fluid mechanics involve several classes of PDEs, and over the years a number of benchmark problems have been established for the validation of novel numerical methods. In fluid mechanics, exploiting the capabilities of computer architectures with thousands of cores has enabled simulations involving billions [27] of variables, resolving scales at Re numbers that have not been possible before. In recent years the advent of graphics processing units (GPUs) has provided another option for accelerated massively parallel flow simulations; albeit sometimes the best performance is obtained at the price of a reduced accuracy. A number of different CFD methods have been successfully implemented on GPUs such as marker and cell [46], finite difference/volume schemes [17, 29], Lattice–Boltzmann [6], and vortex methods [42]. These simulations rely on translating computational methods into intensive and localized computations that take advantage of the single-instruction-multiple-data (SIMD)

*Submitted to the journal's Software and High-Performance Computing section May 20, 2010; accepted for publication (in revised form) November 22, 2010; published electronically March 1, 2011.
http://www.siam.org/journals/sisc/33-2/79593.html

†Department of Computational Science, ETHZ, CH-8092, Zürich, Switzerland (diegor@inf.ethz.ch, babak.hejazialhosseini@inf.ethz.ch, daniele.spampinato@inf.ethz.ch, petros@ethz.ch).

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

capabilities of the GPUs, resulting in some cases in up to two orders of magnitude speedups over the corresponding CPU-only implementations [42].

The sheer number of processors and the corresponding number of computational elements often mask a key issue regarding the economy of flow simulations using uniform grids. Flow simulations that exhibit high parallel efficiency benefit from the regularity of structured grids that can readily translate into effective computer implementations by means of data parallelism. This efficiency, however, does not translate into short time to solution as the grid regularity is not commensurate with the multitude of scales often associated with fluid mechanics problems. Hence, seemingly paradoxically, high efficiency parallel flow simulations, using uniform grids, will not be sufficient to address in the foreseeable future problems of engineering interest such as the flow around an aircraft. We believe that this situation can be remedied by developing computational methods that employ temporal and spatial adaptivity while at the same time allow for implementations that can take advantage of modern computer architectures.

A number of computational methods have been developed in the last twenty years that dynamically adjust the resolution of computational elements to match the physical scales. Methods such as adaptive mesh refinement (AMR) [5, 35] or wavelet based multiresolution techniques [22, 30, 39, 47] have been developed to this end. AMR techniques support unstructured grids and can accommodate different grid orientations, while wavelet methods provide higher compression ratios and allow for efficient computations by the use of fast wavelet transforms. Wavelets are becoming a tool of choice for CFD (see [47] and references therein), and they have been extended to the multiresolution representation of geometries using level sets [4]. Wavelets, while efficient in providing spatial adaptivity, present a number of challenges for their efficient parallelization, making them subject to several efforts in the last ten years that in turn have resulted in hardware-accelerated wavelet transforms. The first GPU based two-dimensional fast wavelet transform (FWT) was introduced in 2000 [25]. Since then, many different parallel implementations have been proposed both on multicore architectures such as the Cell BE [2] and on GPUs [48]. These efforts, however, have been restricted to the *full* FWT. The full FWTs are used mainly in signal processing to find *all* the detail coefficients of a data set, and the subsequent processing is performed in the wavelet space, directly modifying the detail coefficients. However, wavelet-based adaptive PDE solvers differ in their computational implementation with respect to wavelet-based signal processing techniques. First, when wavelets are used for solving an initial value problem, one does not know in advance how the solution will evolve; therefore, the full time-space FWTs cannot be employed. Second, operations involving nonlinear terms (e.g., the convection term in the Navier–Stokes equations) need to be efficiently performed. This is not always possible when working in the wavelet space, and it is meaningful to keep the representation of the solution in the physical space and use the detail coefficients only to readapt the grid. In this approach not every detail coefficient is needed but only the ones directly related to points of the adapted grid (i.e., the finest scaling coefficients of the adapted grid). When the solution is represented in the physical space, a third difference between signal processing and PDE solvers becomes evident. In the latter, the choice of the wavelets is often limited to the biorthogonal ones associated with symmetric and smooth scaling functions. Conventional choices are average interpolating wavelets, first or second generation interpolating wavelets, or B-spline wavelets. Furthermore, the grid has to undergo refinement and compression very frequently as extra care is needed to capture

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

all the emerging scales.

The parallel implementation of wavelet-based adaptive PDE solvers is hindered by their, inherently sequential, nested structure. This difficulty limits their effective implementation on multicore architectures, and the limitation is even more severe for heterogeneous multicore/multi-GPU architectures. In the context of plain multi-core architectures, it has been recently demonstrated that this issue can be relaxed by introducing the concept of wavelet blocks and by employing *task-based parallelism* [43]. To the present knowledge of the authors, there are no works so far that address the issues of solving PDEs on a wavelet-based adapted grid with heterogeneous multi-GPU/multicore machines.

In this work we introduce such a solver for the simulation of multiphase compressible flows and report accuracy issues and performance gains for benchmark problems of shock-bubble interaction and Richtmeyer–Meshkov instability. We remark that the locality of flow structures, inherent to compressible flow simulations, is ideally suited to the locality of data structures that can be exploited by GPUs, as has been shown in the work of Elsen, LeGresley, and Darve [17], Brandvik and Pullan [8], and Hagen, Lie, and Natvig [21]. These works mainly focused on single phase flows on uniform resolution or unstructured grids, while the reduced accuracy of the GPUs has not been extensively addressed. The present work addresses the simulations of compressible, multiphase flows on multicore and multi-GPU architectures using grids adapted via wavelet analysis of the flow structures.

The paper is structured as follows: in section 2 we present the governing equations of multiphase compressible flows and their finite volume discretization. We briefly introduce wavelet-based multiresolution analysis to construct an adaptive grid and solve the flow equations in section 3. We then introduce a data structure referred to as a *wavelet block* to reduce the granularity of the method and make it more suitable for parallel execution. In section 4 we propose a way to couple CPU cores with an array of GPUs to obtain a heterogeneous solver. We report the strong scaling and a detailed study on the accuracy/discrepancy for the Sod shock-tube benchmark and shock-bubble interaction at different Mach numbers as well as the Richtmeyer–Meshkov instability in section 5.

2. Governing equations and uniform resolution discretization.

We model an inviscid compressible flow as described by the Euler equations using the one-fluid formulation [38]:

$$\begin{cases} \dfrac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \\ \dfrac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u} + p\mathbb{I}) = \mathbf{0}, \\ \dfrac{\partial (\rho E)}{\partial t} + \nabla \cdot ((\rho E + p)\mathbf{u}) = 0, \end{cases} \tag{2.1}$$

with ρ being the density, $\mathbf{u}$ the velocity vector, p the pressure, and E the total energy of the fluid per unit mass. Closure of this system of equations is enforced by ensuring that the two phases, gas and liquid fluid, follow the ideal gas equation of state,

$$p = (\gamma - 1)\rho \left(E - \frac{1}{2}|\mathbf{u}|^2 \right), \tag{2.2}$$

with γ being the ratio of specific heats of each phase.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Here, the interface is represented by a color function ϕ, where $\phi < 0$ represents phase 2 and $\phi \geq 0$ represents phase 1.

The evolution of the interface is governed by a linear advection equation of the form

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = 0. \tag{2.3}$$

The specific heat ratio γ couples the evolution of the interface to the system of equations (2.1)–(2.2), and it is based on ϕ:

$$\gamma(\phi) = \gamma_{\text{phase1}} H_\epsilon(\phi) + \gamma_{\text{phase2}}(1 - H_\epsilon(\phi)), \tag{2.4}$$

$$H_\epsilon(\phi) = \begin{cases} 0, & \phi < -\epsilon, \\ \frac{1}{2} + \frac{\phi}{2\epsilon} + \frac{1}{2\pi}\sin(\frac{\pi\phi}{\epsilon}), & |\phi| \leq \epsilon, \\ 1, & \phi > \epsilon, \end{cases} \tag{2.5}$$

where the mollification length ϵ is constant. The governing equations (2.1) can be cast in the form

$$\mathbf{q}_t + \nabla \cdot \mathbf{f}(\mathbf{q}) = \mathbf{0}, \tag{2.6}$$

where the subscript t denotes the time derivative. This system of equations requires an initial condition of the form $\mathbf{q}(\mathbf{x}, 0) = \mathbf{q}_0(\mathbf{x})$ and appropriate boundary conditions. The evolution of the fluid under this system of equations often leads to the development of shocks and contact discontinuities in the solution vector $\mathbf{q}$, and therefore the integral form of (2.6), i.e.,

$$\oint (\mathbf{q} d\mathbf{x} - \mathbf{f}(\mathbf{q}) dt) = \mathbf{0}, \tag{2.7}$$

is better suited for numerical simulations. If the computational domain is uniformly discretized by finite volumes, then cell averages $\{\mathbf{q}_i^n\}$ at time $t = t_n$ and the flux

$$\mathbf{F}_{i\pm1/2} = \frac{1}{\Delta t} \int_{t^n}^{t^{n+1}} \mathbf{f}(\mathbf{q}_{i\pm1/2}^n(t)) dt \tag{2.8}$$

determine the new solution $\mathbf{q}^{n+1}$ at time $t_{n+1} = t_n + \Delta t$. In order to avoid the expensive Riemann solver [49], $\mathbf{F}_{i\pm1/2}$ is approximated by a numerical flux $\hat{\mathbf{F}}_{i\pm1/2}$. The new cell averages $\{\mathbf{q}_i^{n+1}\}$ are found after one simulation step by evaluating the numerical fluxes and performing a time integration for all the averages:

$$\mathbf{q}_i^{n+1} = \mathbf{q}_i^n - \frac{\Delta t}{\Delta x}(\hat{\mathbf{F}}_{i+1/2} - \hat{\mathbf{F}}_{i-1/2}). \tag{2.9}$$

Since $\hat{\mathbf{F}}_{i+1/2}$ and $\hat{\mathbf{F}}_{i-1/2}$ depend on the local cell neighbors of $\mathbf{q}_i^n$, the simulation step formulated in (2.9) can be seen as a nonlinear uniform filtering at the location of $\mathbf{q}_i^n$. In this work we use the HLLE [16] numerical flux that has been shown to be well capable of capturing isolated shocks and rarefaction waves and is not as computationally expensive as the Riemann solver. We employ a fifth order WENO scheme [33, 28] to reconstruct the primitive quantities and a time-stepper based on the second order TVD Runge–Kutta scheme [51]. The interface evolution equation (2.3) is made suitable for our generic conservative flux-based solver using the method introduced in [45].

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

3. Wavelet-based adaptive grids. In this work, the creation of adapted grids is based on average interpolating wavelets. These wavelets belong to the family of biorthogonal wavelets and are used to construct a multiresolution analysis (MRA) of the quantities of interest. We can use the scale information of the MRA to obtain a compressed representation by performing a thresholding that retains only the scaling coefficients which carry significant information.

In order to solve the flow equations on adapted grids, we apply standard finite volume (alternatively finite difference) schemes on the active coefficients. One way to simplify these operations is to first create a local, uniform resolution neighborhood around a grid point and then apply the corresponding finite volume scheme to it. This process in turn requires the introduction of auxiliary grid points, the so-called *ghosts*. For accuracy and efficiency reasons, the grid has to be readapted as the flow field evolves. This is done by coarsening some regions and refining some others after constructing a new MRA of the flow.

We refer the reader to the appendix for a detailed discussion on how average interpolating wavelets are used to solve the flow equations on adaptive computational grids.

3.1. Wavelet blocks. The wavelet-adapted grids for solving flow equations are often implemented with quad-tree or oct-tree structures whose leaves are single scaling coefficients (i.e., cell averages). The main advantage of such fine-grained trees is the very high compression rate which can be achieved when thresholding individual detail coefficients. The drawback of this approach is the large amount of sequential operations it involves and the number of memory indirections (or read instructions) necessary to access a group of elements. Even if we keep the grid structure unchanged while computing, these grids already perform a great number of neighborhood look-ups. In addition, operations like refining or coarsening single grid points are relatively complex and strictly sequential.

In this work, in order to expose more parallelism and to decrease the amount of sequential operations per grid point, the key idea is to simplify the data structure. We address this issue by coarsening the *granularity* of the solver, i.e., the atomic element of the grid, at the expense of a reduction in the compression rate. We introduce the concept of *block* of grid points, whose size is one or two orders of magnitude larger in every direction with respect to a scaling coefficient; i.e., in three dimensions, the granularity of the block is coarser by 3 to 5 orders of magnitude with respect to a single scaling coefficient. All the scaling coefficients contained in one block have the same level, and every block contains the same number of scaling coefficients. The grid is then represented with a tree which contains blocks, instead of single grid points, as leaves (Figure 3.1). The blocks are nested so that every block can be split and doubled in each direction and blocks can be collapsed into one. Blocks are interconnected through the ghosts and, in the physical space, the blocks have varying sizes and therefore different resolutions. The block structure of the grid provides a series of benefits. First, tree operations are now accelerated as they can be performed in $\log_2(N^{1/D}/s_{block})$ operations instead of $\log_2(N^{1/D})$, where N is the total number of active coefficients, D is the considered dimensionality, and s_{block} is the block size. The second benefit is that the random access at elements inside a block can be efficient because the block represents the atomic element. Another advantage is the reduction of the sequential operations involved in processing a local group of scaling coefficients. The cost c (in terms of memory accesses) of filtering a grid point with a filter of size $w_{stencil}D$ in a uniform resolution grid is proportional to $w_{stencil}D$.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

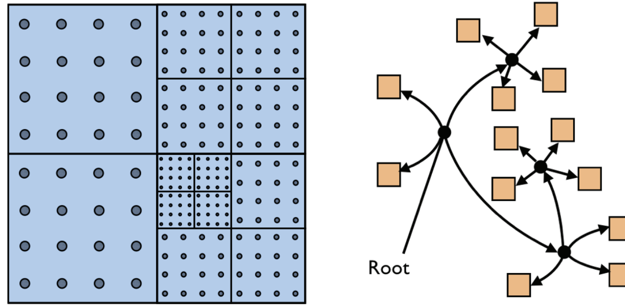

Fig. 3.1. *The adapted cell-centered grid is represented with* 13 *blocks at* 3 *different resolutions. Every block contains the same number of scaling coefficients (left). The blocks are the leaves of the grid tree (right).*

For a grid represented by a fine-grained tree, the number of accesses is proportional to $c = w_{stencil} D \log_2(N^{1/D})$, due to the sequential access to the tree nodes. Using the wavelet blocks approach and assuming that s_{block} is roughly one order of magnitude larger than $w_{stencil}$, the ratio of ghosts needed per grid point in order to perform the filtering for a block is

$$r = \frac{(s_{block} + w_{stencil})^D - s_{block}^D}{s_{block}^D} \approx D \frac{w_{stencil}}{s_{block}}. \tag{3.1}$$

Therefore, the number of accesses for filtering one grid point is

$$\begin{aligned} c &= (1 - r) w_{stencil} D + r w_{stencil} D \big(\log_2(N^{1/D}/s_{block}) \big) \\ &= w_{stencil} D + w_{stencil} D r \big(\log_2(N^{1/D}/s_{block}) - 1 \big). \end{aligned} \tag{3.2}$$

As an example we consider a two-dimensional worst case scenario: an adapted grid of four million points that is fully refined everywhere. For $N = 2^{22}, D = 2$ we want to perform a fifth-order WENO reconstruction; therefore, $w_{stencil} = 6$. For a uniform resolution grid representation we have $c = 12$, and for a fine-grained adapted tree we have $c = 132$. For a blocked adaptive grid with $s_{block} = 32$ we have $c \approx 36$. This means that in terms of memory accesses, by using blocks we are 3 times slower than a uniform grid and 3.7 times faster than a fine-grained adapted grid. On average, in places where we have an "actual" adaptation of the grid, the speed difference between the block-based and the fine-grained grids becomes substantially larger because of the number of ghosts that they need to reconstruct.

We note that none of the blocks overlap with any other block in the physical space as the blocks are the leaves of the tree. In order to improve the efficiency of finding the neighbors of a block, we define as neighbors all the blocks adjacent to the one considered. Because of this constraint, the highest jump in resolution allowed (between two adjacent blocks), L_j, is bounded by $\log_2(s_{block}/w_{stencil})$.

3.2. Local time stepping (LTS) for wavelet adapted grids. Additional speedup is obtained by combining spatially adapted grids with local time-stepping (LTS) schemes, which advance in time the properties of the computational elements by taking into consideration their corresponding different time scales necessary for a stable discretization. LTS schemes have been shown to speed up the computation by one order of magnitude or more depending on the number of blocks at each level of resolution [13, 14, 1, 44].

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Recently, we proposed a new perspective on LTS schemes [23] that allows us to reformulate existing LTS algorithms in a simpler way. In this view each computational element of the grid is considered as an independent entity. We associate a time-reconstruction function to every computational element (i.e., cell averages). The computation of the right-hand side is not performed by considering directly the computational elements, but by considering their time-reconstructed values. Each block in the grid is also associated with a state diagram, which is repeatedly traversed when the grid elements undergo a time integration. The operations involved in the state transitions of a grid point are deduced by imposing some interpolating postconditions on the reconstruction functions. In the new formulation we relax, or almost eliminate, the coupling between the evaluation of right-hand sides, the update of the blocks, and the recursive iteration through the blocks. This enables us to identify the compute-intensive parts of the LTS schemes and therefore obtain, in combination with the block-based representation of the grid, an effective implementation for multicore/multi-GPU machines.

Even though LTS schemes are capable of dramatically decreasing computing time, they present a major shortcoming: the grid blocks have to be grouped by their time scales, and therefore multiple groups cannot be processed in parallel. This means that there are additional synchronization points in the algorithm, and the degree of parallelism for processing a group of blocks is limited by the group size.

4. Implementation on multi-GPU/multicore architectures. In this section we discuss the retainment of scales on the grid, introduce OpenCL for GPUs, and explain the parts of the algorithm that have been redesigned for multi-GPU execution. Finally, we present the OpenCL-based GPU implementation of the solver.

4.1. One simulation step. In order to keep the grid adapted to the evolving solution, a simulation step modifies the grid in three different stages [32]—the refinement stage, the computing stage, and the compression stage—as illustrated in Figure 4.1 (left).

The refinement stage refines the grid in those regions where small scales are expected to emerge. In practice, we perform a one-level FWT to compute the finest detail coefficients of the grid. If the maximum detail over all detail coefficients within a block is bigger than ϵ_{refine}, we trigger the split of the block, and we fill the new blocks by performing a one-level inverse wavelet transform of the scaling coefficients of the old block.

In the computing stage we perform the time integration of the solution by combining finite volume schemes and the LTS scheme. Note that in this stage the grid structure remains unchanged. It is desired (and empirically observed) that most of the execution time is spent here. This stage generally takes between 80% and 98% of the execution time, depending on the problem, the chosen wavelet type, and other computational parameters. This means that the refinement/compression stage introduces a time overhead between 1–10%, which is the cost for maintaining a dynamically adapted grid.

Once the numerical time integration is performed, the grid is further processed in the compression stage. In this stage we perform another one-level FWT to retrieve the new detail coefficients; based on those coefficients we decide where to coarsen the grid by collapsing blocks. We traverse the grid considering, at once, all the blocks sharing the same parent. If the maximum detail over all the children blocks that share the same parent is less than $\epsilon_{\text{compress}}$ and if the local resolution jump is less than the

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

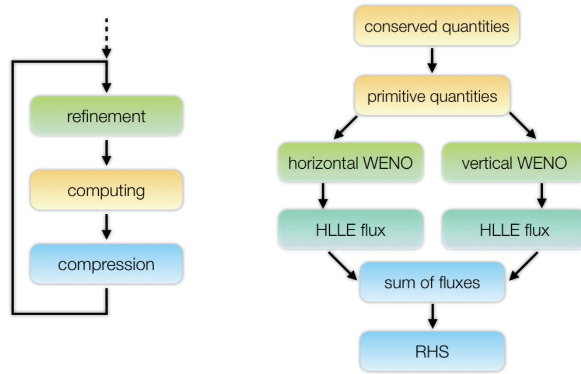

FIG. 4.1. *Stages to perform one simulation step (left) and data flow diagram for evaluating the right-hand side in the computing stage (right).*

maximum allowed, then we collapse the children. The data of the new leaf is created by performing a one-level FWT of the scaling coefficients within the blocks.

4.2. The computing stage. In order to integrate the solution in time, the computing stage performs one global step of the LTS scheme. We can identify three different operations during the grid traversing of the LTS: time-reconstruction of the quantities, evaluation of the right-hand side, and update of the quantities by time integration. Time-reconstruction and integration are not computationally very expensive as they involve a small number of multiplications and additions per grid point. Because of their poor/medium computing intensity, it is likely that the performance of these two operations are bounded by the speed of accessing the grid points in memory. The evaluation of the right-hand side is performed after the time-reconstruction and is computationally expensive. This consists in reconstructing the ghosts, converting the conserved quantities into primitive quantities, WENO reconstructing five quantities (ρ, u, v, E, ϕ) in two directions, evaluating HLLE fluxes for all the quantities, and summing them, as illustrated in Figure 4.1 (right).

WENO reconstructions involve a considerable amount of computation. A WENO reconstruction of order $2r-1$ can be written as [28]

$$\hat{f}^{+}_{j+1/2} = \sum_{k=0}^{r-1} \omega_k q_k^r (f_{j+k-r+1}, \dots, f_{j+k}), \tag{4.1}$$

where $\{f_j\}$ are primitive quantities and

$$q_k^r(g_0, g_1, \dots, g_{r-1}) = \sum_{l=0}^{r-1} a_{k,l}^r g_l, \tag{4.2}$$

$$\omega_k = \frac{\alpha_k}{\sum_{i=0}^{r-1} \alpha_i}, \tag{4.3}$$

$$\alpha_k = \frac{C_k^r}{(\epsilon + IS_k)^p}, \quad k = 0, 1, \dots, r-1, \tag{4.4}$$

with $\epsilon = 10^{-6}$ and $p = 2$ in our computations. For a fifth order WENO reconstruction, $r = 3$ and $a_{k,l}^r$, C_k^r are obtained from Table 4.1, and the indicators of smoothness IS_k

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

TABLE 4.1
Coefficients of the 5th order WENO reconstruction.

	$a_{k,l}^{r=3}$			$C_k^{r=3}$
k	$l=0$	$l=1$	$l=2$	-
0	1/3	-7/6	11/6	1/10
1	-1/6	5/6	1/3	6/10
2	1/3	5/6	-1/6	3/10

for $r=3$ and $k=0,1,2$ are computed as

$$IS_0 = \frac{13}{12}(f_{j-2} - 2f_{j-1} + f_j)^2 + \frac{1}{4}(f_{j-2} - 4f_{j-1} + 3f_j)^2, \tag{4.5}$$

$$IS_1 = \frac{13}{12}(f_{j-1} - 2f_j + f_{j+1})^2 + \frac{1}{4}(f_{j-1} - f_{j+1})^2, \tag{4.6}$$

$$IS_2 = \frac{13}{12}(f_j - 2f_{j+1} + f_{j+2})^2 + \frac{1}{4}(f_j - 4f_{j+1} + 3f_{j+2})^2. \tag{4.7}$$

Because the reconstructed values are shared by adjacent cells in a block, we have to perform 10 WENO reconstructions per cell per block (the "south" and "west" cell faces for five scalar quantities).

The evaluation of the numerical flux HLLE is compute-intensive. It reads

$$\hat{F}_{j+1/2} = \begin{cases} \hat{f}^-_{j+1/2}, & a^-_{j+1/2} > 0, \\ \hat{f}^+_{j+1/2}, & a^+_{j+1/2} < 0, \\ \frac{a^+_{j+1/2}\hat{f}^-_{j+1/2} - a^-_{j+1/2}\hat{f}^+_{j+1/2} + a^+_{j+1/2}a^-_{j+1/2}(u^+_{j+1/2} - u^-_{j+1/2})}{a^+_{j+1/2} - a^-_{j+1/2}} & \text{otherwise,} \end{cases} \tag{4.8}$$

where a^- and a^+ are the left-running and right-running characteristic velocities, respectively.

4.3. OpenCL. OpenCL stands for "open computing language" and is a specification for programming on heterogeneous (parallel) computing devices. An extensive introduction to the OpenCL specification can be found in [36]. Here we restrict ourselves to discussing OpenCL for GPU computing. In this case OpenCL is similar to the CUDA framework [37], and, moreover, it provides portability between different GPU hardware. The OpenCL specification consists mainly in the description of three models: the execution model, the memory model, and the programming model. The execution model is split into two parts: *kernels* that execute on one or more OpenCL devices (in our case the GPUs) and a *host program* that executes on the host (CPU). The host program defines the context for the kernels and manages their execution. When a kernel is submitted for execution by the host, the OpenCL devices are capable of running multiple instances of the same kernel in parallel. One kernel instance is called a *work item*. Work items are executed in parallel and can process different data. One can submit kernels for execution, memory commands, and synchronization commands to an OpenCL device with a *command queue*. When the host places commands into the command queue, the queue schedules them onto the device. A very appealing feature of command queues is that they can support out-of-order execution: commands that are issued in order do not wait to complete before the next commands execute. Submissions of commands to a queue return event handles, the OpenCL *events*. They are used to describe the dependency between commands, to monitor the execution, and to coordinate execution between the host and the devices.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

4.4. OpenCL for GPUs. Optimal GPU performance dictates that the parallelism must be expressed with fine-grained homogeneous work items. These devices are *many-core* architectures, and each core processes work items in a single-instruction multiple-data (SIMD) fashion. As the number of resources available per core is very limited, the work items have to be fine-grained, with many threads performing small processing on a small amount of data. Because of their fine-grained properties, the lifetime of a work item is expected to be in the order of microseconds or less. At the same time, the number of work items is expected to be large—at least one order of magnitude larger than the amount of cores. Considering this and the fact that currently GPUs are *single-instruction multiple-threads* (SIMT) machines, it is clear that optimal performance is achieved when work items are *homogeneous*, i.e., when the small tasks perform the same operations. The performance gain achieved by GPUs is very fragile: severe performance degradation can be observed when work items require too many resources (i.e., they are not fine-grained) or contain too many condition dependent instructions (inhomogeneous work items) or do not access memory with the right pattern (memory-indirections, traversing sequential data structures, etc.).

4.5. Heterogeneous computing with multicore and multi-GPU. The above-mentioned performance fragility of the GPUs prompted us not to aim for a complete GPU implementation but instead to accelerate the computing stage, which is the most expensive part of the algorithm, using multiple GPUs. In order to achieve this, a number of modifications were necessary in our original algorithm. The main reason is that one single block does not expose enough fine-grained work items to keep all the GPU cores busy, as the block size is in the range of 8×8 and 32×32 grid points, while current generation of GPUs have hundreds of cores (on the computing hardware considered in this work we have 240 cores per GPU). Moreover, work items within a block are not homogeneous: grid points close to the block faces need to reconstruct ghosts in order to evaluate the right-hand side. Because of the ghosts' reconstruction, these work items demand a substantial number of extra registers to carry out the summation of a dynamic-sized array. This request of extra resources makes these work items not fine-grained anymore, which leads to a reduced number of active (concurrent) work items.

In order to achieve better performance, we pack blocks into *input tokens*, and we reconstruct ghosts on the CPU. We then process the tokens on the GPUs. The advantage of working with packs of blocks (ranging from 4×4 to 20×20 blocks) is that we expose enough work items to keep all the GPU cores busy. The CPU preparation of the tokens is multithreaded and is expressed with task-based parallelism. As illustrated in Figure 4.2, input tokens include not only grid points of blocks but also ghosts. The calculation of ghosts is nested in parallel tasks spawned inside the creation of input tokens, which are also parallel tasks. The ghosts' reconstruction tasks are not load balanced, as the time complexity for reconstructing ghosts depends on the local structure of the grid. However, dynamic load balance is achieved by the Work Stealing algorithm [7], which schedules these tasks in a near-optimal way. As soon as one input token is ready to be processed, it is asynchronously scheduled by OpenCL for an out-of-order GPU execution. In Figure 4.2 it is shown that the GPU processing fills *output tokens*, which contain the right-hand side of the flow quantities. The GPU-CPU transfer of the output tokens is asynchronously scheduled by OpenCL. When the transfer of one output token is complete, the token is unpacked in parallel and the grid blocks associated with that token are time-integrated using the new right-hand side values. As the memory resources of GPUs are limited, sometimes

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

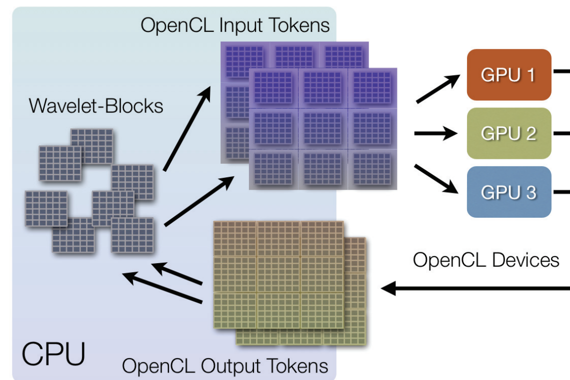

FIG. 4.2. *Wavelet blocks are prepared and sent to the GPUs for the evaluation of the right-hand side. The GPU output is then postprocessed to update the solution. The preparation of the input tokens is fully multithreaded and is responsible for reconstructing the ghosts, packing the blocks into tokens and sending them to the GPUs. Once the GPUs have processed some tokens, the postprocessing stage unpacks the output token in a multithreaded fashion.*

it is not possible to create and submit all the necessary input tokens at once. In that case the program waits for a GPU to be done with some of the tokens and reuses the allocated GPU memory to make new input tokens. As tokens are stored on both CPU and GPUs, their memory layout is divided into a CPU part and a GPU part. As we desire asynchronous CPU-GPU transfer of the tokens, the CPU part of each token is allocated through OpenCL requests for page-locked memory regions. The CPU part of a token consists of several plain two-dimensional arrays, where each entry has four components ("float4"). During the packing process, these arrays are filled by CPU threads with the necessary content from the wavelet blocks. This data is then sent to the GPUs and converted into image objects, which represent the GPU part of the token. The packing of the tokens consists in the extraction of the relevant quantities from the grid points, and it is performed by iterating over a group of wavelet blocks. Similarly, to unpack a token we iterate over the associated wavelet blocks and read the corresponding entries from their arrays.

4.6. Reimplementing the right-hand side evaluation for GPUs. The most straightforward way of reimplementing the right-hand side evaluation would be to directly map grid-points of one input token to work items and perform all the computation in one kernel. Since optimal performance on GPUs imposes a high number of work items that are fine-grained and homogeneous, we cannot afford a one-to-one mapping. Such a mapping would require many registers, and due to limited resources this would result in a reduced number of concurrently active work items. Instead, we split the processing into different kernels, as depicted in Figure 4.3: four kernels for the WENO reconstruction, two kernels for evaluating the HLLE fluxes, and one to sum up the fluxes. This splitting of the GPU processing into multiple kernels introduces an additional overhead associated to their invocation. The large number of work items involved in these kernels makes this overhead negligible as it takes less than 1% of the total execution time (from empirical observations). The GPU computation starts after the GPU uploads an input token. The four WENO kernels start to process the token in parallel. The group of four WENO kernels can be split into pairs: one pair performs the reconstruction between the grid points in the

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

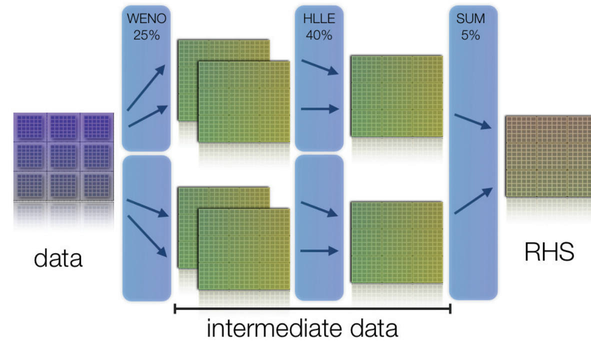

FIG. 4.3. *Data flow for the GPU computation. The input data (left) contains the current solution. Intermediate data (green blocks) are produced by executing the WENO and HLLE kernels (blue). The final data (RHS) is the result of the SUM kernel.*

x-direction, whereas the second pair performs the reconstruction in the y-direction. The first kernel in this pair carries out the "$-$" side reconstruction whereas the second one takes care of the "+" side. As soon as the WENO reconstruction in the x-(y-)direction is done, the execution of the kernels computing the HLLE x-(y-)fluxes can take place. Once both x- and y-fluxes are computed, the SUM kernel sums the fluxes in both directions and fills the output token with the right-hand side. At this point the output token is ready to be downloaded to the CPU. Except for the last one, every kernel fills some intermediate data that is subsequently read as input in the next kernel.

In order to potentially increase data-level parallelism and reduce the number of instructions, the computation inside the kernels is expressed with SIMD instructions with a vector width of four ("float4"). Inside the WENO and HLLE kernels, the work items are mapped to the work of a cell face. We observed that the WENO kernels take 48–51 registers per work item, which translates to an occupancy of 0.312 on the considered GPU hardware. The HLLE kernel is more expensive in terms of register usage, as it takes 107 registers per work item, leading to an occupancy of 0.125. The work items of the SUM kernels are mapped to the work of 2×2 cells to maximize the reuse of the fluxes. We observed an occupancy of 0.375 using 42 registers per work item. Both input and intermediate data are represented with image objects, as the WENO and SUM kernels show nontrivial data access patterns.

Control flow instructions could result in different branches among the work items (divergent branches). These branches are absent in the WENO and SUM kernels. Instead, the HLLE kernels show a rate of 0.05 divergent branches per work item. We explain this by the presence of three conditional assignments (per kernel) that are translated into if-statements.

As illustrated in Figure 4.3 most kernels (even of different types) can be executed in parallel. Our implementation submits these kernels for execution so as to take advantage of the newest hardware capability to support parallel execution of different types of kernels [18]. However, it should be noticed that the results reported in this work are obtained with GPUs without this capability.

We note that the dependency graph of all the operations involved in processing one token is not trivial (some nodes in the graph have a degree that is higher than two). An efficient implementation that follows this dependency graph with the explicit use of synchronization points is challenging, especially if we take into account that

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

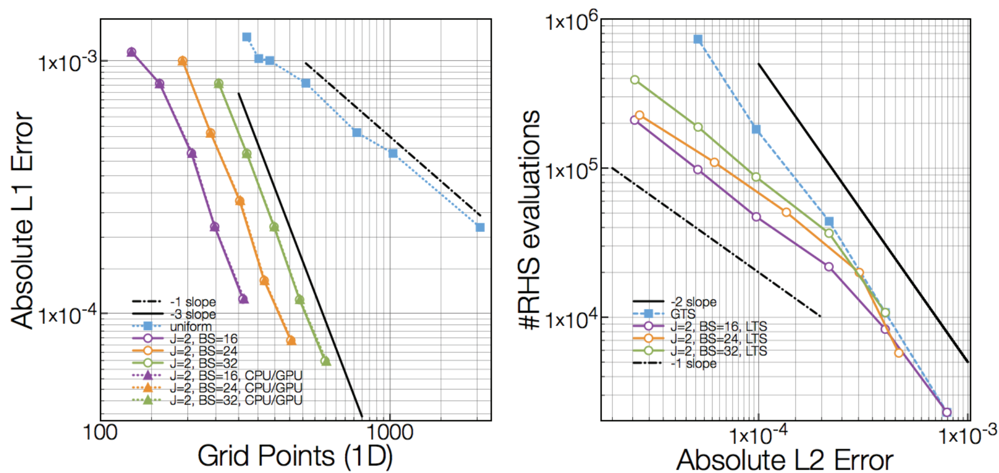

Fig. 5.1. *L_1 (left) space convergence for uniform and adaptive simulation on multicore and multi-GPU using different block sizes and a jump in resolution of 2. Number of right-hand side evaluations versus the absolute L_2 norm error using global time stepping (GTS) and local time stepping (LTS) versions of RK2 time stepper (right).*

different tokens can be processed at the same time. With OpenCL, however, we do not have to handle these synchronization points explicitly, as we can describe the complete sequence of actions by means of the OpenCL events. The submission of the commands to transfer a token, process it on the GPU, and transfer it back is done in one shot. The events are used by OpenCL to optimally schedule these tasks in order to achieve a near-optimal execution.

5. Results. In this section we report the accuracy of the heterogeneous solver for the simulation of the Sod shock-tube benchmark [31] and simulations of shock-bubble interaction and Richtmeyer–Meshkov instability. We highlight the discrepancy in the flow quantities computed on the CPU and the GPU and present the achieved performance gain with the heterogeneous solver. All the results have been obtained with the hardware listed in section 5.4.1 and by using fifth-order average interpolating wavelets and a maximum resolution jump of two. We have used block sizes of 16×16, 24×24, and 32×32 grid points for the shock-tube problem and a block size of 32×32 in our two-dimensional simulations.

5.1. One-dimensional benchmark: Sod shock-tube. Figure 5.1 shows the convergence plots for the Sod shock-tube problem simulated on both multicore CPU and multicore/multi-GPU. We observe that there is no significant difference between the two plots. This test case is a standard benchmark for the accuracy of numerical methods in solving compressible flows and has an analytical solution. We notice the expected first-order convergence in space for the L_1 (left picture) norm errors using uniform resolution grids. The discontinuities present in the exact solution of this test case prohibit convergence orders of more than one in finite volume based methods. This slow convergence can be overcome by using fewer grid points when possible, therefore increasing the convergence rate to 3 in both cases. We note that to achieve the same accuracy (in terms of L_1 norm errors), fewer grid points are necessary when using smaller block sizes with a jump in resolution of 2. It is clear from Figure 5.1 (left picture) that in almost all the cases, the CPU+GPU execution produces the same

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

accuracy in space as the CPU-only execution. This test case is a standard benchmark for the accuracy of numerical methods in solving compressible flows and has an exact solution [31].

To show the performance improvement obtained with LTS, we present the number of right-hand side evaluations needed when using global time-stepping (GTS) and LTS versions of RK2 time-stepper versus the L_2 norm error in Figure 5.1 (right) in the simulation of the Sod shock-tube problem up to a given time. It is clear that to achieve the same accuracy (here in the L_2 norm), we need fewer right-hand side evaluations with LTS. In fact, LTS asymptotically improves the complexity of the solver from $O(1/e^2)$ to $O(1/e)$, where e is the upper bound for the L_2 error at the final time. Smaller L_2 norm errors translate to having finer resolutions (and more adaptivity in space), and, therefore, the performance improvement obtained by LTS becomes more significant at smaller errors.

5.2. Two-dimensional benchmark: Shock-bubble interaction. We perform simulations of the shock-bubble interaction problem in order to study the discrepancy between the solutions of CPU+GPUs and CPU-only executions. Furthermore, we report on the achieved speedup using different computer architectures for the simulations at different Mach numbers.

In the shock-bubble interaction problem a supersonic shock wave in one medium (air) collides with a cylindrical inhomogeneity (helium) with a nonunit ratio of specific heat constants. The generated baroclinic vorticity modifies the interface between the two phases and produces a complex structure of waves in the vicinity of the inhomogeneity.

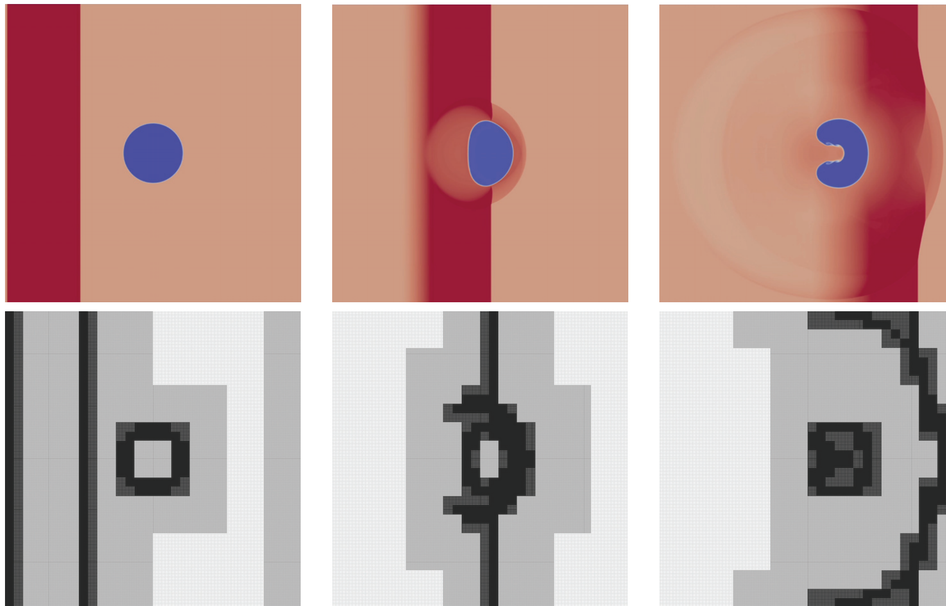

FIG. 5.2. *Density field (top) and the adaptive grid (bottom) in the simulation of the shock-bubble interaction at $M = 1.2$. From left to right: the initial condition, density at the nondimensional time $T = 1$, and density at $T = 5$.*

We present the density field as well as the adaptive grid for two simulations at $M = 1.2$ and $M = 3$ in Figures 5.2 and 5.3 at three different times. In these

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

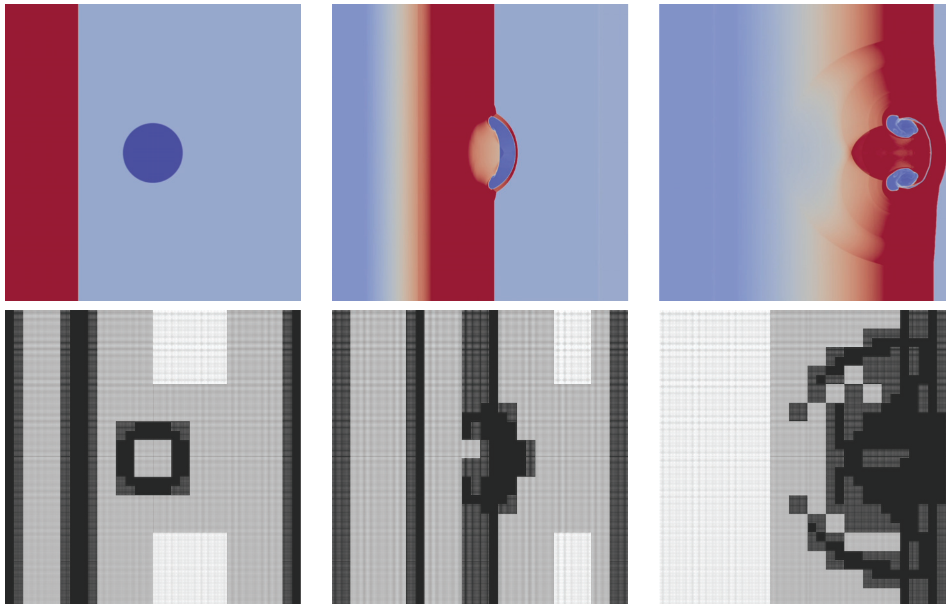

FIG. 5.3. *Density field (top) and the adaptive grid (bottom) in the simulation of shock-bubble interaction at $M = 3$: the initial condition (left), at nondimensional time $T = 1.5$ (center), and at $T = 4$ (right).*

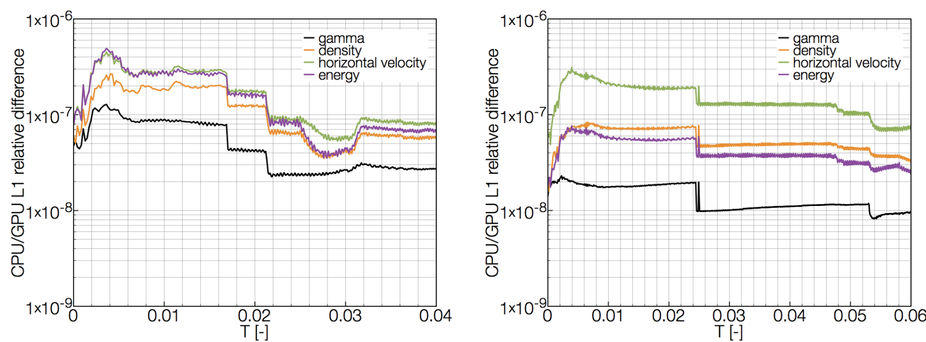

FIG. 5.4. *L_1 discrepancy between the solution computed with CPU+GPUs and the CPU one versus time. Left: $M = 1.2$. Right: $M = 3$.*

simulations, grid adaptation was driven by the detail coefficients of the density field as well as the location of the interface. The results of $M = 1.2$ and $M = 3$ simulations are in very good agreement with the experimental and numerical results of [20, 40, 3, 26].

In Figure 5.4 we present the discrepancy in the solution of integration steps using the LTS version of the RK2 time-stepper for $M = 1.2$ and $M = 3$ between CPU+GPUs and CPU-only executions. We notice that, although the differences in both cases become smaller in time, those in the $M = 1.2$ case are higher than those in the $M = 3$ case. Since the solution is updated by multiplying the right-hand side by a small time step, which is on the order of 10^{-3} nondimensional time units in our two-dimensional simulations, it is more appropriate to consider the relative and absolute discrepancies of the right-hand side computation stage.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

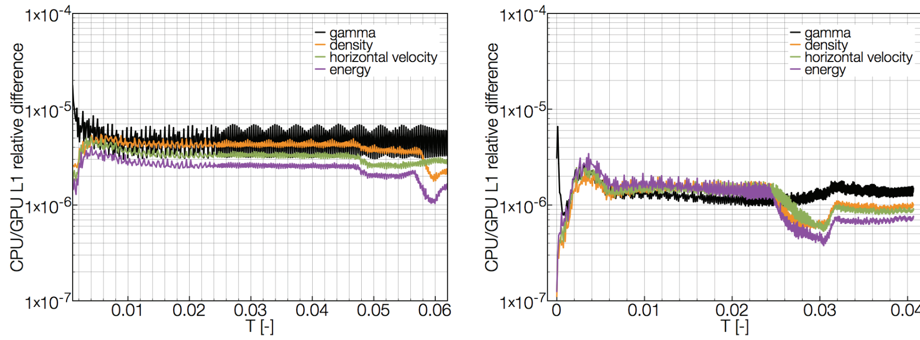

Fig. 5.5. *Time evolution of the L_1 discrepancy of the right-hand side between CPU+GPUs and CPU executions. Left: $M = 1.2$. Right: $M = 3$.*

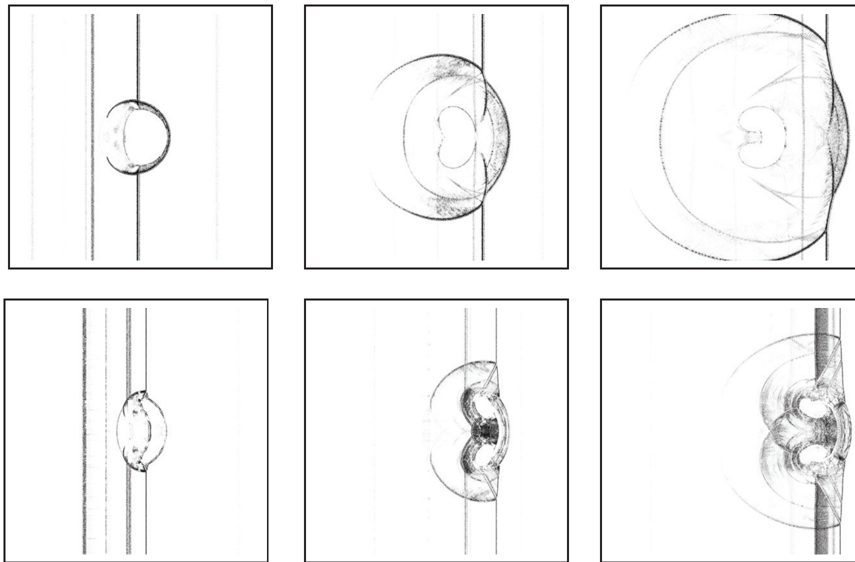

Fig. 5.6. *Localization of the GPU-CPU discrepancy in space. Top: $M = 1.2$. Bottom: $M = 3$. White/black: low/high L_1 relative difference, black $= 10^{-6}$, chronologically from left to right.*

Figure 5.5 shows the L_1 relative difference in computing the right-hand side for γ, ρ, ρu, and ρE components of the solution between CPU+GPUs and CPU-only executions versus time (T) in the simulation of the shock-bubble interaction at $M = 1.2$ (left) and $M = 3$ (right). The discrepancy of the right-hand side for the vertical velocity component v is not shown here, as during the early times of the simulation its value was zero everywhere in the field. We observe that the L_1 relative difference in computing the right-hand side remains smaller than 10^{-5} for both Mach numbers.

We also present the distribution of the relative difference of the flow quantities in space at three different times in Figure 5.6. We observe that the highest discrepancies are located in the proximity of the shocks.

We further investigate the discrepancies introduced in the substeps involved in computing the right-hand side, i.e., the WENO reconstruction, HLLE flux computation, and the summation of fluxes. The space and time averaged relative and absolute differences are shown in Figure 5.7 (left), where it is clear that the significant discrep-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

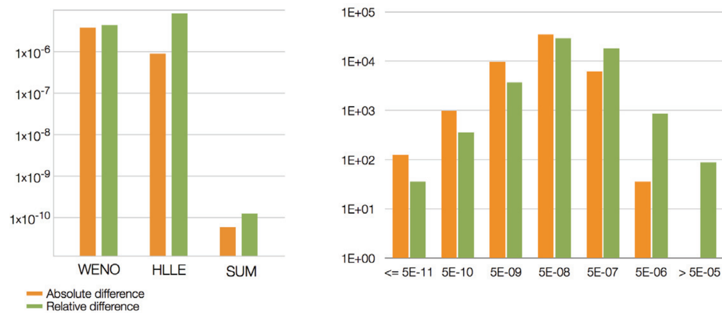

FIG. 5.7. *Averaged relative and absolute discrepancies in the WENO reconstruction, HLLE flux computation and the summation of fluxes for the simulation of the shock-bubble interaction (left). The histogram of relative and absolute discrepancies produced in WENO reconstruction stage on random initial data contains some large discrepancies that are in the order of* 10^{-4} *(right).*

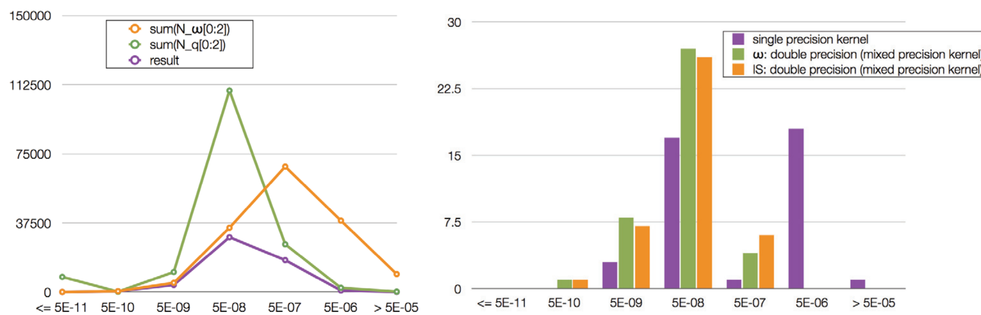

FIG. 5.8. *Histogram of the absolute differences (left) in computing the internal elements of the WENO reconstruction:* $\sum_{i=0}^{2} N_{\omega_i}$ *(orange),* $\sum_{i=0}^{2} N_{q_i}$ *(green), result (violet). Histogram of the absolute differences between single/mixed precision and the reference computations of the internal components in the WENO reconstruction stage (right). The significant errors are eliminated by using double precision in computing* IS_i*. Little improvement is observed by using double precision in all the stages of computing* ω_i*.*

ancies are generated in the WENO and the HLLE stages. Although the average WENO discrepancy is similar to that of HLLE, we observe that some of the WENO discrepancies can assume values of two orders of magnitude higher than those of HLLE. Our goal is therefore to analyze the internal elements of the WENO reconstruction stage in detail and to reveal where relatively high discrepancies lie within this stage so as to eliminate the significant errors.

We perform the WENO reconstruction with random initial data on the GPU using OpenCL using single precision as well as on the CPU using double precision as the reference. The histogram of relative and absolute differences is plotted in Figure 5.7 (right). We have also performed the same computation on the CPU using OpenCL in single precision and compared it to the reference, and we noticed no significant difference between the results obtained from the GPU (OpenCL, single precision) and those from the CPU (OpenCL, single precision). Therefore, in our further analysis of the discrepancy we compare our computations on the CPU using OpenCL in single precision with those on the CPU using double precision.

In the next step, we investigate the discrepancies introduced by the internal computations involved in the WENO reconstruction stage. Figure 5.8 (left) shows the

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

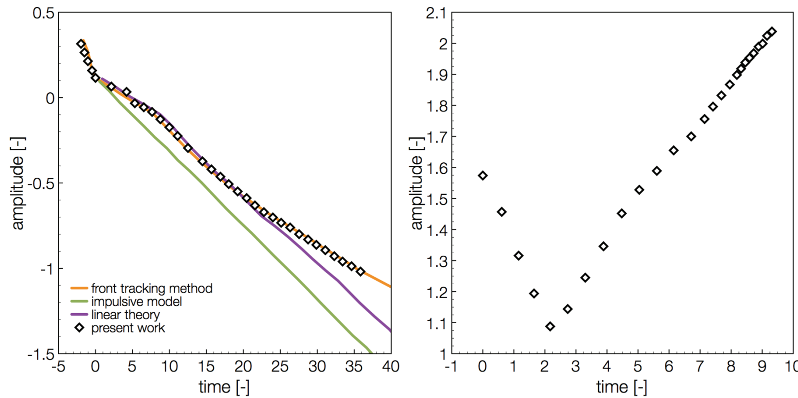

FIG. 5.9. *Evolution of the interfacial perturbation amplitude in time: air/helium at $M = 1.52$ (Meshkov experiment) (left) and helium/air at $M = 3$ (right).*

population distributions of the discrepancies in ω_i and q_i $(i = 0, 1, 2)$ as well as the histogram of the discrepancies in the final result of the WENO reconstruction stage, i.e., $\hat{f}_{j+1/2}$. It is clear that the distribution of differences of ω_i has a larger mean than that of q_i. By using mixed precision we try to improve some of the stages used to compute (4.2)–(4.7). We focus only on the critical cases that generate relatively higher discrepancies in plots of Figure 5.8 and use double precision first in all the stages for computing ω_i, i.e., in (4.2)–(4.4), and then only in the computation of smoothness indicators, i.e., (4.5)–(4.7). The improvements are presented in the histograms of the Figure 5.8 (right) versus the original single precision computations. It can be observed that the significant errors on the rightmost part of this plot are suppressed by using double precision in computing ω_i. We also notice that if we use double precision only for the computation of IS_i, we maintain almost the same accuracy as we would when using the double precision for the whole computation of ω_i together with saving processing time.

5.3. Simulations of the Richtmeyer–Meshkov instability. The Richtmeyer–Meshkov instability (RMI) [41, 9] is another benchmark problem in gas dynamics where baroclinic vorticity is generated on the perturbed interfaces by shock-induced pressure gradients. We validate our solver against the Meshkov experiment [34], which consists of a $M = 1.52$ normal shock wave in air with an Atwood number $A = 0.76$ for the air/helium interface. The initial amplitude of the sinusoidal perturbation is $\eta_0 = 0.2\,cm$ with a wavelength $\lambda = 4\,cm$ and a nondimensional amplitude $(\tilde{\eta} = 2\pi\eta/\lambda)$ of 0.314. Time is nondimensionalized using $T = Ma_\infty t/\eta_0$, where a_∞ is the speed of sound inside the preshock air. The quantity measured in the literature and in our simulations is the amplitude η of the interfacial perturbations (defined as half the interpenetration length) [24]. In Figure 5.9 (left), we present the evolution of η in time using our method and compare with the results found in [24]. The interface undergoes the linear stage of RMI with a phase change at later times $(T \approx 18)$ and deviates from the linear theory predictions. The results are in good agreement with those achieved by the front tracking method [19, 11].

In Figure 5.9 (right), we present the evolution of the interfacial amplitude for a $M = 3$ shock wave in helium interacting with an $A = 0.76$ interface between helium and air. We set $\eta_0 = 0.125$ and $\lambda = 0.5$, which gives a nondimensionalized initial amplitude of 1.57 for the interfacial perturbation. The evolution of the interface is

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

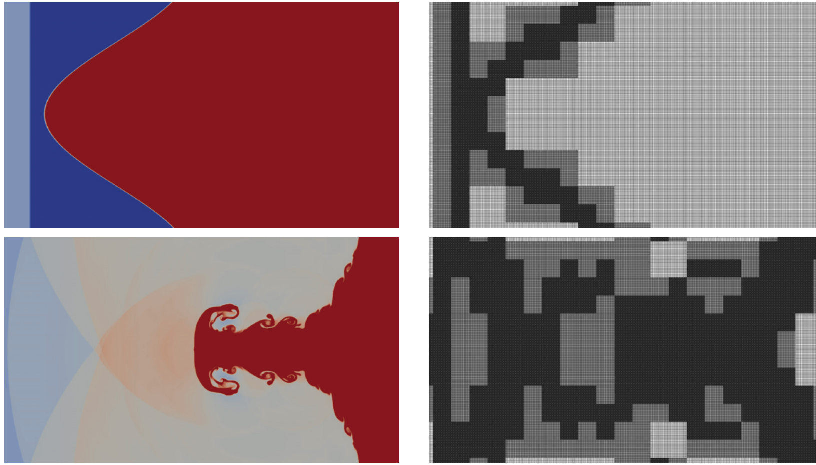

FIG. 5.10. *Density field (left) and the adaptive grid (right) of the simulation of Richtmeyer–Meshkov instability at $M = 3$. Red/blue denote high/low density. Initial condition (top), solution at $T = 7.68$ (bottom).*

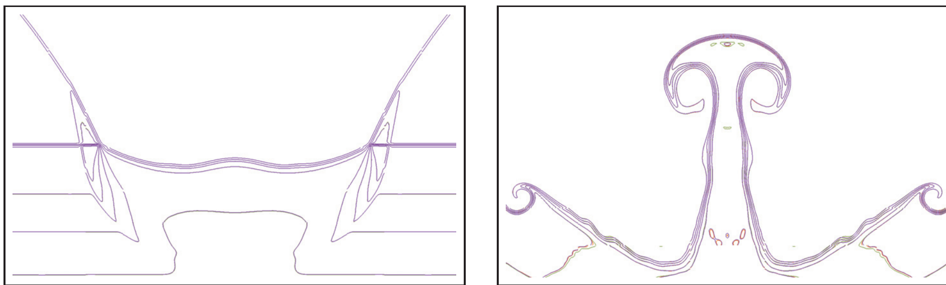

FIG. 5.11. *Closeup on the density contours of the Richtmeyer–Meshkov instability simulated using CPU with double precision (green) and single precision (orange) as well as CPU+GPUs in single precision (violet), from left to right. The contours coincide except at later times of the simulation.*

no longer predictable by linear theory and, as the phases are swapped, the amplitude of the interfacial perturbation does not undergo a phase change as in the air/helium case. Until time $T \approx 2$ (in our simulation), i.e., when the shock has passed over the whole interface, the interface is compressed to $\tilde{\eta} \approx 1.1$.

In Figure 5.10 we present the density field and the adapted grid at two different times. It can be observed that the grid adapts to follow the helium/air interface and the discontinuities in density, i.e., transmitted and reflected shock waves. The initial grid illustrated in Figure 5.10 (top right) contains 70,000 grid points. After an execution time of 4 hours, the heterogeneous solver has a grid (bottom right) consisting of 2,000,000 grid points, with 5 levels of resolution.

Figure 5.11 shows a close-up of the contours of density for three different executions of an RMI simulation on the CPU using double and single precision as well as on the heterogeneous CPU+GPU architecture. We notice that the small discrepancies appear at very late times in the simulation, these differences mostly exist between the double and single precision executions, and they are not as significant between the single precision executions on CPU and on CPU+GPUs.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

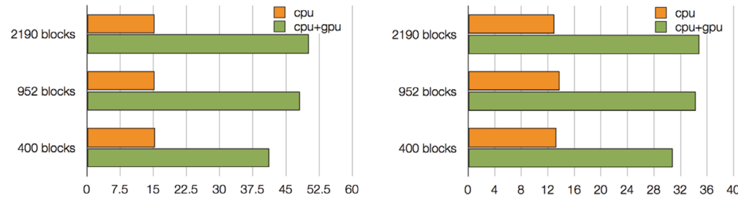

FIG. 5.12. *Speedup achieved in computing the right-hand side (left) and in the overall computing stage (right) by multicore/multi-GPU execution (green) and CPU-only execution (orange) on a compute node with* 16 *cores and* 2 *GPUs.*

5.4. Performance. We compare the performance of the multicore/multi-GPU solver for the adaptive simulation of a shock-bubble interaction. The results were obtained using a grid with 7 levels of resolution and about 2,200,000 grid points. We report the strong scaling with respect to multicore and single-core CPU-only execution, and we analyze how strong scaling varies by changing different parameters. Both single-core and multicore CPU-only solvers were profiled using VTune and were found to be reasonably efficient; they make extensive use of SSE 3 intrinsics to improve the computing performance as well as the bandwidth and were compiled using Intel C++ Compiler 11.1 with the flags "`-O3 -axTW -ip`." The multicore CPU-only solver exploits the task-based parallelism and processes the blocks as in [43]. In this study we consider two machines: a compute node with 16 cores and 2 GPUs, and a compute node with 12 cores and 6 GPUs.

5.4.1. Performance on 16 CPU cores and 2 GPUs. In Figure 5.12 we show the speedup for different numbers of wavelet-blocks (and therefore grid points) on a machine with 4 quad-core AMD Opteron 8380 processors ("Shanghai" cores, 2.5 GHz, 6 MB L3-cache) and 2 NVIDIA Tesla T10 S1070 (480 GPU cores in total). On the left we report the speedup for the evaluation of the right-hand side compared to a single-CPU core only. With 16 cores (orange) and no GPUs we observe a speedup of 15, meaning that the solver achieved a strong efficiency of roughly 94%. If we make use of two GPUs we get a considerable improvement in the speedup (green) that monotonically increases with the number of wavelet blocks. For a simulation that handles about 2000 blocks ($\sim 2,200,000$ grid points) the GPUs are able to improve the multicore execution by a factor of 3.3.

The right part of Figure 5.12 shows the overall speedups, i.e., the speedups including also the parts that were not executed on GPUs. Using only the CPU cores the overall speedup is almost 12, and by including both GPUs the solver shows a maximum speedup of roughly 35. This means that GPUs improve the multicore execution time by a factor of roughly 2.8.

5.5. Performance on 12 CPU cores and 6 GPUs. The following results are obtained with a machine with two six-core AMD Opteron 2435 processors ("Istanbul," 2.8 GHz, 6 MB L3-cache) and 32 GB of memory, connected to six GPUs (Tesla S1070). Figure 5.13 shows the strong scaling of the right-hand side evaluations against the number of cores and the number of GPUs. We observed a peak internal GPU bandwidth of 73 GB/s out of the theoretical 102 GB/s and a peak computing performance of 680 GFLOP/s out of the theoretical 690 GFLOP/s.

The first observation is that the highest performance gains are around 102 (versus a single core execution) and are obtained by using all the 12 cores with 5–6 GPUs. Over the multicore only execution, GPUs amplify the improvement factor by 9.2. We

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

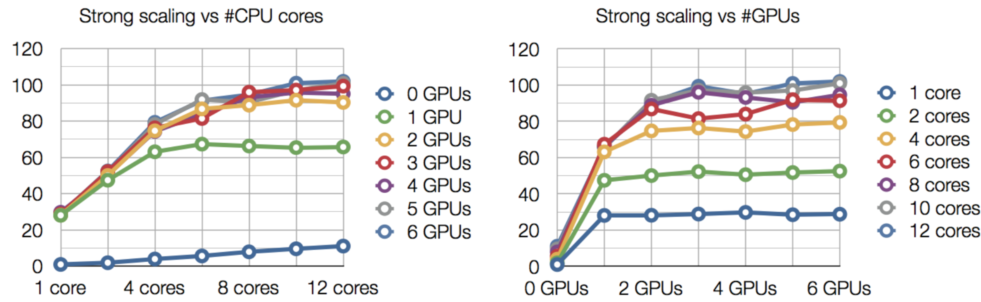

FIG. 5.13. *Strong scaling for the evaluation of the RHS against the number of CPU cores (left) and against the number of GPUs.*

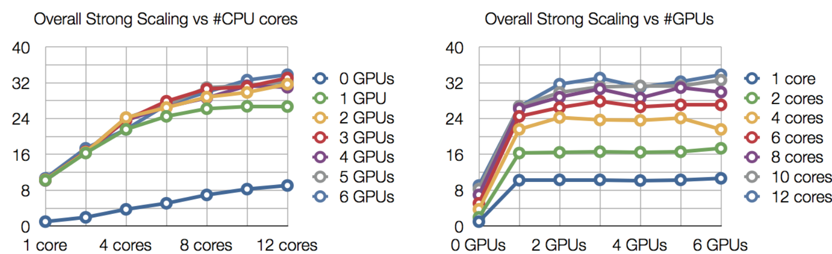

FIG. 5.14. *Overall strong scaling against the number of CPU cores (left) and GPUs (right).*

also note that increasing the number of cores to more than six and using 1 GPU does not give any improvements. Also, there is a substantial difference between the strong scaling obtained with one GPU compared to 2 GPUs. The difference between 2 GPUs and 3–6 GPUs is not substantial anymore and we obtain approximately the same scaling for more than two GPUs (left plot).

A general observation about the right plot of Figure 5.13 pertains to the poor scaling in the number of GPUs. It seems that this scaling always reaches a plateau regardless of the number of (CPU) cores. Despite this issue, we also note that increasing the number of cores matters: it shifts the plateau to a higher number of GPUs. This is clearly visible by comparing the single-core execution with the 12-core execution: while the first one reaches the plateau with one GPU, the second one reaches the plateau with 3 GPUs. The saturation indicates that the bottleneck is in the CPU. In terms of input tokens, the CPU cores do not provide a sufficient input rate to "feed" more GPUs.

Figure 5.14 shows the overall strong scaling that also includes the timings of the code that do not run on the GPU. The best overall scaling measured is 33.8, obtained with 6 GPUs and 12 cores. This value, however, does not substantially differ from those obtained by employing 4–5 GPUs with the same number of cores.

Figure 5.15 shows the total time spent on the GPUs and the strong scaling versus the granularity of the tokens, i.e., the number of packed blocks per token. The total GPU timings reported in the chart (left) were collected by using the OpenCL events. The main observation is that the time distribution does not change while increasing the number of GPUs. A second observation is that 70–75% of the total GPU time is spent in computing. This means that the evaluation of the right-hand side is computation-intensive also for the GPUs. The remaining 20–25% of the time is spent

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

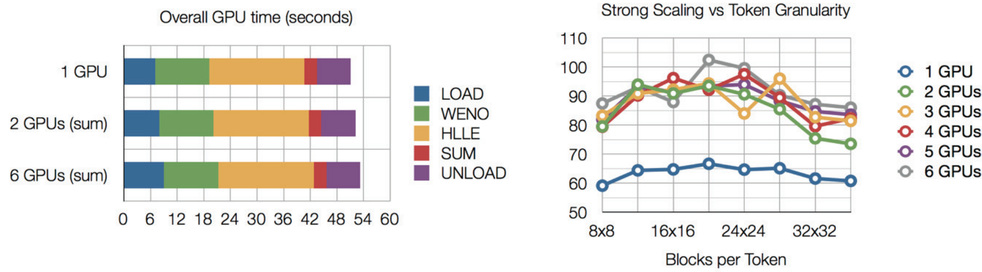

FIG. 5.15. *Distribution of the GPU-time for the different steps (left) and the strong scaling versus the granularity of the input tokens (in terms of blocks per token) (right).*

in transferring the data from and to the CPU in roughly equal parts. It is interesting to note that the total GPU time increases with the number of GPUs. With one GPU we obtain a total GPU time of 51.1 seconds, whereas with 6 GPUs the measured GPU time is 53.2 seconds. This means that we have an overhead cost of 4% for managing 6 GPUs. It is also interesting to note that the heterogeneous execution took a total time of 127 seconds with 1 GPU and 101 seconds with 6 GPUs. This implies that the overall GPU usage is 40% for one GPU and 8% for 6 GPUs.

By observing the plot of the strong scaling versus the granularity of the input tokens (right picture of Figure 5.15), we note that there is an optimal range of blocks per input token. This range seems to be between 16×16 and 28×28 blocks per token. The precise optimal number of blocks, however, seems to depend on the particular number of employed GPUs. We observe a "wiggly" behavior of the strong scaling with more than 2 GPUs for large token sizes. As the token size increases, the number of tokens per GPU decreases. With a reduced set of large tokens, it becomes crucial to have a number of tokens that is a multiple of the number of GPUs. If this is not the case, drops in the scaling curve are to be expected, as only a fraction of the GPUs is effectively processing tokens.

We used the PAPI library [10] to measure the FLOPS involved in the shock-bubble interaction benchmark. In total, it consists of $5700\cdot10^9$ floating point operations. Around $5600\cdot10^9$ of these operations were performed in the computing stage, meaning that the refinement and compression stages are not computation-intensive as they took only 1% of the floating point computation. The evaluation of the right-hand side takes around $5200\cdot10^9$ floating point operations, namely, the 92% of the floating point computation. With 3 GPUs (or more) and a token granularity of 16×16 blocks, the solver is able to evaluate the right-hand side computing 61.4 million grid points per second, leading to a maximum performance of 121 GFLOPS (using all the GPUs). In theory, taking into account that the CPU-GPU (and vice versa) bandwidth of the machine is around 4–5 GB/s, the solver should process up to 600 million grid points per second. This rate is reduced to 400 million grid points if one considers that only 70% of the GPU time is spent in computing. We believe that the CPU preparation of the input tokens, which processes 60 million grid points per second, is the major bottleneck of the current implementation and limits the output rates in the evaluation of the right-hand side.

We used the NVIDIA OpenCL profiler to estimate the performance of the WENO, HLLE, and SUM kernels. The instruction throughput of the WENO kernels is around 1.2 instructions per cycle, whereas the HLLE and SUM kernels show an instruction

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

throughput of 0.65 instructions per cycle. The WENO kernels show a performance of 130 GFLOPS per token and contain 8 floating point divisions per grid point component. As we estimate that these kernels have an operational intensity [50] of 3.1 FLOP/Bytes, the performance of these kernels reaches 55% of the maximum attainable performance (73 GB/s $\times$ 3.1 FLOP/Bytes $\approx$ 230 GFLOPS). The HLLE kernels show a performance of 24 GFLOPS per token and contain three divisions per component. For this kernel we estimate an operational intensity of 1.9 FLOP/Bytes. This means that the HLLE kernels reach 17% of the maximum attainable performance. This relatively poor performance is explained by the substantial number of divergent branches observed during the execution of these kernels. The SUM kernels show a performance of 20 GFLOPS per token and have an estimated operational intensity of 0.3 FLOP/Bytes. This means that the SUM kernels reach 90% of the maximum attainable computing performance, which is estimated to be 22 GFLOPS.

6. Discussion. We presented an adaptive finite volume solver for multiphase compressible flows based on fifth-order average interpolating wavelets. The solver is, to the best of our knowledge, the first of its kind in that it is capable of exploiting the computing power of modern heterogeneous multicore/multi-GPU machines for adapted grids. This is achieved by grouping grid points into blocks of fixed size at different resolutions and using an array of GPUs for the evaluation of the right-hand side. We optimize the GPU execution by packing blocks into bigger tokens that are preprocessed by the CPU cores. The preprocessing of the tokens also includes the time reconstruction of the grid values and the calculation of the ghosts, which is inherently load unbalanced. Load balance is, however, dynamically achieved by expressing the preprocessing with task-based parallelism. The preprocessed tokens are asynchronously enqueued for GPU execution by means of OpenCL events.

Events allow the CPU cores to prepare tokens without waiting for the GPU processing of previous tokens to be completed. When the preprocessing of all tokens is ended, the CPU cores extract the values of the output tokens and time-integrate the corresponding blocks.

The present heterogeneous solver was validated on the shock-tube problem, resulting in the same accuracy as the CPU-only solver. We analyzed the discrepancy between the heterogeneous and the CPU-only solvers for the simulation of the shock-bubble interaction, showing that the relative L_1 discrepancy of the two solutions is in the range of 10^{-7}–10^{-6}. As the discrepancy in the solution depends on the chosen time step, we also reported the discrepancy in the right-hand side. We noted that the discrepancy was in the range of 10^{-6}–10^{-5}. We also showed the discrepancy in space, and we noted that the highest values in the discrepancy are located in the proximity of the shocks. As WENO-based solvers are only first order accurate at the shock locations, the discrepancy of the heterogeneous solver does not dramatically degrade the accuracy with respect to the CPU-only solver.

We also analyzed the discrepancy of each individual step of the algorithm, identifying the WENO reconstruction as the most inaccurate one. We could improve the WENO operator by switching the computation of three internal variables of the WENO reconstruction from single to double precision.

We further validated the solver against the previous works on the Meshkov experiment for the simulation of RMI and also provided the same diagnostic for a higher Mach number test. We showed that the density contours of an RMI simulation performed on the CPU and on CPU+GPU match very well.

The heterogeneous solver results in an overall performance gain of 25X and 34X,

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

over the CPU-only single-core execution, for two different multicore/multi-GPU machines. The improvement factors for the part executed on the GPUs on the same machines are 50X and 102X.

We reported the strong scaling against the number of blocks, number of cores, number of GPUs, and number of blocks per token. The number of blocks plays a major role as it dictates the degree of parallelism achievable in the simulation. We noted that the number of cores is also a crucial parameter for scaling, although the effect they have on the performance becomes less substantial for more than 8 CPU cores. This decrease in parallel efficiency is attributed to the nonideal scalability of the preprocessing of the tokens which includes a considerable number of memory transfers. This issue limits the effective use of more than 3 GPUs, as the "input tokens per second" cannot be sufficiently increased to "feed" the array of GPUs. A way to further improve the performance is to increase the bandwidth of the memory transfers performed in the preprocessing of the tokens.

We observed that the optimal range of blocks per token is between 16×16 and 28×28 blocks. For more than 2 GPUs, the choice of suboptimal ratios of blocks per token can lead to a performance penalty of almost 30%. By profiling the GPU-computation we noticed that the computing time is around 75%, and the time spent in transferring the data is around 25%. This means that the evaluation of the right-hand side is computationally bounded even when executed on the GPUs.

7. Conclusion and future work. We have presented a state-of-the-art wavelet-adaptive finite volume solver for compressible flows suitable for execution on heterogeneous computing environments using OpenCL. We developed a methodology to overcome the implementation hurdles of wavelet-based adaptivity on multicore CPUs by means of wavelet blocks and task-based parallelism. We discussed the accuracy issues raised by using single precision computation on GPUs for the Sod shock-tube, the shock-bubble interaction, and the RMI and reported the performance improvement achieved by executing the computation-intensive part of the solver on the GPUs. These reports indicate that the results from GPU-assisted simulation are very competitive with those of multicore CPUs in terms of accuracy and that the computation-intensive part of the solver can be performed 50 to 100 times faster on the discussed compute nodes.

Future works include the extension of the present method to three dimensions and the simulation of bluff-body flows and cavitation-induced bubble collapse.

Appendix A. Biorthogonal wavelets are used to construct a multiresolution analysis (MRA) of the quantities of interest, and they are combined with finite difference/volume approximations to discretize the governing equations. Biorthogonal wavelets are a generalization of orthogonal wavelets, and they can have associated scaling functions that are symmetric and smooth [12]. Biorthogonal wavelets introduce two pairs of functions: ϕ, ψ for the synthesis and $\tilde{\phi}, \tilde{\psi}$ for analysis. Given a signal in the physical space, the functions $\tilde{\phi}, \tilde{\psi}$ are used in the *forward* wavelet transform to compute the wavelet coefficients. The functions ϕ, ψ are used in the *inverse* wavelet transform to reconstruct the signal in the physical space from the wavelet coefficients. The functions $\phi, \psi, \tilde{\phi}, \tilde{\psi}$ introduce four refinement equations:

$$\phi(x) = \sum_m h_m^S \phi(2x+m), \quad \psi(x) = \sum_m g_m^S \phi(2x+m), \tag{A.1}$$

$$\tilde{\phi}(x) = \sum_m h_m^A \tilde{\phi}(2x+m), \quad \tilde{\psi}(x) = \sum_m g_m^A \tilde{\phi}(2x+m). \tag{A.2}$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

For the case of average interpolating wavelets, $\tilde{\phi}(x) = T(x/2)/2$, $\psi(x) = -T(x) + T(x-1)$, where T is the "top-hat" function. Because of their average-interpolating properties, the functions $\tilde{\psi}$ and ϕ are not known explicitly in analytic form [15]. The analysis filters h^A_m, g^A_m are used in the fast wavelet transform (FWT), whereas the synthesis filters h^S_m, g^S_m are used in the inverse FWT.

The forward wavelet transform computes two types of coefficients: the scaling $\{c^l_k\}$ and detail coefficients $\{d^l_k\}$. From the scaling and detail coefficients of f obtained in the forward transform, we can reconstruct f as follows:

$$f = \sum_k c^0_k \phi^0_k + \sum_{l=0}^{L} \sum_k d^l_k \psi^l_k, \tag{A.3}$$

where $\phi^l_k = \phi(2^l x - k)$ and $\psi^l_k(x) = \psi(2^l x - k)$.

If f is uniformly discretized in space with cell averages $\{f_i\}$, we can find $\{c^0_k\}$ and $\{d^l_k\}$ by first considering the finest scaling coefficients to be $c^L_k = f_k$ and then perform the full FWT by repeating the step

$$c^l_k = \sum_m h^A_{2k-m} c^{l+1}_m, \quad d^l_k = \sum_m g^A_{2k-m} c^{l+1}_m \tag{A.4}$$

for l from $L-1$ to 0. To reconstruct $\{f_i\}$ we use the inverse FWT, which repeats the following step for l from 0 to $L-1$:

$$c^{l+1}_k = \sum_m h^S_{2m-k} c^l_m + \sum_m g^S_{2m-k} d^l_m. \tag{A.5}$$

A.1. Active scaling coefficients. Using the full FWT, we can decompose uniformly discretized functions into scaling and detail coefficients, resulting in an MRA of our data. We can now exploit the scale information of the MRA to obtain a compressed representation by keeping only the scaling coefficients that carry significant information. This is done by thresholding the detail coefficients:

$$f_{\geq\varepsilon} = \sum_k c^0_k \phi^0_k + \sum_l \sum_{k:|d_k|\geq\varepsilon} d^l_k \psi^l_k, \tag{A.6}$$

where ε is the compression threshold used to truncate relatively insignificant terms in the reconstruction. Active scaling coefficients therefore consist of the scaling coefficients c^l_k needed to compute d^l_k (such that $|d^l_k| \geq \varepsilon$) as well as the coefficients at coarser levels needed to reconstruct c^l_k. The pointwise error introduced by this thresholding is bounded by ε. Each scaling coefficient has a physical position; therefore, the compression results in an adapted grid $\mathcal{K}$, where each grid node is an active scaling coefficient.

A.2. Ghosts. We can solve the fluid flow equations on $\mathcal{K}$ by applying standard finite-volume (or finite-difference) schemes on the active coefficients. One way to simplify these operations is to first create a local, uniform resolution neighborhood around a grid point and then apply the corresponding finite-volume scheme on it. In order to do this, we need to temporarily introduce artificial auxiliary grid points, the so-called *ghosts*. In providing the ghosts required around a grid point, finite-volume schemes can be viewed as (nonlinear) filtering operations in a uniform resolution frame. Formally,

$$F(\{c^l_{k'}\}_{k'\in\mathbb{Z}})_k = \sum_{j=s_f}^{e_f-1} c^l_{k+j}\beta^l_j, \quad \beta^l_j \text{ function of } \{c^l_m\}, \tag{A.7}$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

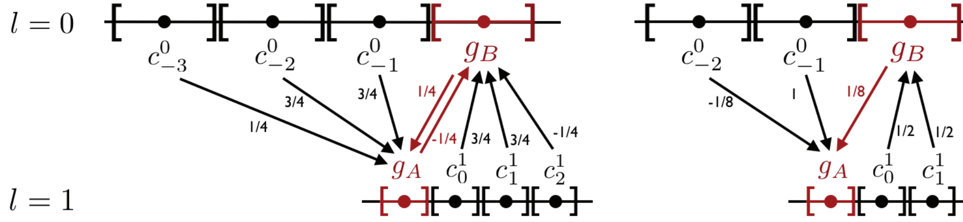

FIG. A.1. *Graph of the contributions for the reconstruction of the ghost g_A, with a resolution jump of* 1 *for the case of third-order B-spline wavelets (left) and third-order average interpolating wavelets (right). The arrows, and their associated weights, denote contributions from the grid points to the ghosts g_A and g_B. Both wavelets need the secondary ghost g_B to evaluate g_A, but average-interpolating wavelets are more efficient as the evaluation of g_B does not depend on g_A (i.e., there is no loop in the graph).*

where $\{s_f, e_f - 1\}$ is the support of the filter in the index space and $e_f - s_f$ is the number of nonzero filter coefficients $\{\beta_j\}$. We need to ascertain that for every grid point k in the adapted set $\mathcal{K}$, its neighborhood $[k - s_f, k + e_f - 1]$ is filled with either other points k' in $\mathcal{K}$ or ghosts g. By using ghosts we are now able to apply the filter F to all points k in $\mathcal{K}$. A ghost is constructed from the active scaling coefficients as a weighted average $g_i^l = \sum_l \sum_j w_{ijl} c_j^l$, where the weights w_{ijl} are provided by the refinement equations (A.4). It is convenient to represent the construction of a ghost as $g_i = \sum_j w_{ij} p_j$, where i is the identifier for the ghost and j represents the identifier for the source point p_j which is an active scaling coefficient in the grid. Calculation of the weights $\{w_{ij}\}$ is done by traversing a graph associated with the FWT and the inverse wavelet transform (Figure A.1).

This operation can be computationally expensive for several reasons: first, if the graph contains loops, we need to solve a linear system of equations to compute $\{w_{ij}\}$. Figure A.1 (left) shows this issue for a two-resolution grid associated with the third-order B-spline wavelets. According to the one-level inverse wavelet transform, the evaluation of the ghost g_A simply consists of a weighted average of the points $\{c^0_{-3}, c^0_{-2}, c^0_{-1}, g_B\}$, where g_B is a secondary ghost. The value of g_B is obtained from the points $\{c^1_0, c^1_1, c^1_2, g_A\}$, according to the one-level FWT. As g_A depends on g_B and vice-versa, one has to solve the following linear system to find g_A:

$$\begin{cases} g_A = \frac{1}{4}c^0_{-3} + \frac{3}{4}c^0_{-2} + \frac{3}{4}c^0_{-1} + \frac{1}{4}g_B, & \text{(A.8)} \\ g_B = \frac{3}{4}c^1_0 + \frac{3}{4}c^1_1 - \frac{1}{4}c^1_2 - \frac{1}{4}g_A. & \text{(A.9)} \end{cases}$$

For any order of B-spline wavelets, one can find g_A by introducing a vector of ghost values $\mathbf{g}$ and having g_A as the first component of $\mathbf{g}$, i.e., $g_A = \mathbf{e}_1^{\mathrm{T}} \cdot \mathbf{g}$. The remaining components are secondary ghosts involved in the calculation of g_A. It holds that

$$\mathbf{g} = \mathbf{W}_{ghosts} \cdot \mathbf{g} + \mathbf{W}_{points} \cdot \mathbf{k}, \tag{A.10}$$

where the matrices $\mathbf{W}_{ghosts}$ and $\mathbf{W}_{points}$ are specific to the wavelet type, and $\mathbf{k}$ is a vector containing all the grid point values involved in the reconstruction of g_A. The entry $(\mathbf{W}_{ghosts})_{ij}$ contains the weight that the ghost g_i receives from the ghost g_j, whereas the entry $(\mathbf{W}_{points})_{ij}$ contains the weight that g_i receives from the point p_j. The ghost g_A can be found with an expensive matrix inversion:

$$g_A = \mathbf{e}_1^{\mathrm{T}} \cdot (\mathbf{I} - \mathbf{W}_{ghosts})^{-1} \mathbf{W}_{points} \cdot \mathbf{k}. \tag{A.11}$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Another reason that makes the evaluation of the ghosts costly is related to the resolution jump, i.e., the difference in resolution across two adjacent wavelet blocks. The number of secondary ghosts involved in the reconstruction of g_A, and therefore the cost of inverting $(\mathbf{I} - \mathbf{W}_{ghosts})$, grows exponentially with the resolution jump.

Figure A.1 (right) shows the evaluation of g_A with the use of third-order average interpolating wavelets. By virtue of their interpolating property, the ghost reconstruction is straightforward and leads to efficient reconstruction formulæ. Since the evaluation of g_B does not involve g_A, we have

$$g_A = -\frac{1}{8}c^0_{-2} + c^0_{-1} + \frac{1}{8}\left(\frac{1}{2}c^1_0 + \frac{1}{2}c^1_1\right). \tag{A.12}$$

The extension to two dimensions is straightforward due to the separability of the two-dimensional FWT and inverse FWT. In this work we used fifth-order average interpolating wavelets, which have similar efficient reconstruction formulæ but involve more points.

A.3. Dynamic grid adaptation. In order to accurately and efficiently solve the flow equations, the grid has to be readapted as the flow variables evolve. This is done by coarsening some regions and refining some others. To readapt the grid we need to threshold the finest detail coefficients of $\mathcal{K}$; therefore, we perform a one-level FWT at all the grid points. Based on the thresholds $\leq \epsilon_{\text{compress}}$ and $\geq \epsilon_{\text{refine}}$, we can then decide where to coarsen and where to refine the grid.

Acknowledgments. We would like to thank Hauke Kreft, from the IT Service Group in the Department of Computer Science of ETH Zurich, for the priceless help in installing OpenCL and various GPUs on different computers. We would like also to thank Dr. Oliver Byrde and Teodoro Brasacchio, from the Brutus cluster support team at ETH Zurich, for the invaluable help and support in using the heterogeneous compute nodes employed in this work. We would like also to thank Dr. Serban Georgescu (Fujitsu) and Dr. Michael Bergdorf (D. E. Shaw research) for their precious suggestions and discussion throughout the course of this work.

REFERENCES

[1] J. M. Alam, N. K.-R. Kevlahan, and O. V. Vasilyev, *Simultaneous space-time adaptive wavelet solution of nonlinear parabolic differential equations*, J. Computat. Phys., 214 (2006), pp. 829–857.

[2] D. A. Bader, V. Agarwal, and S. Kang, *Computing discrete transforms on the Cell Broadband Engine*, Parallel Computing, 35 (2009), pp. 119–137.

[3] A. Bagabir and D. Drikakis, *Mach number effects on shock-bubble interaction*, Shock Waves, 11 (2001), pp. 209–218.

[4] M. Bergdorf and P. Koumoutsakos, *A Lagrangian particle-wavelet method*, Multiscale Model. Simul., 5 (2006), pp. 980–995.

[5] M. J. Berger and J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comput. Phys., 53 (1984), pp. 484–512.

[6] M. Bernaschi, L. Rossi, R. Benzi, M. Sbragaglia, and S. Succi, *Graphics processing unit implementation of lattice Boltzmann models for flowing soft systems*, Phys. Rev. E, 80 (2009), 066707.

[7] R. D. Blumofe and C. E. Leiserson, *Scheduling multithreaded computations by work stealing*, J. ACM, 46 (1999), pp. 720–748.

[8] T. Brandvik and G. Pullan, *Acceleration of a 3D Euler solver using commodity graphics hardware*, in Proceedings of the 46th AIAA Aerospace Sciences Meeting, American Institute of Aeronautics and Astronautics, Reston, VA, 2008, AIAA-2008-607.

[9] M. Brouillette, *The Richtmyer-Meshkov instability*, Ann. Rev. Fluid Mech., 34 (2002), pp. 445–468.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, *A portable programming interface for performance evaluation on modern processors*, Int. J. High Perform. Comput. Appl., 14 (2000), pp. 189–204.

[11] I. L. Chern, J. Glimm, O. McBryan, B. Plohr, and S. Yaniv, *Front tracking for gas-dynamics*, J. Comput. Phys., 62 (1986), pp. 83–110.

[12] A. Cohen, I. Daubechies, and J. C. Feauveau, *Biorthogonal bases of compactly supported wavelets*, Comm. Pure Appl. Math., 45 (1992), pp. 485–560.

[13] M. O. Domingues, S. M. Gomes, O. Roussel, and K. Schneider, *An adaptive multiresolution scheme with local time stepping for evolutionary PDEs*, J. Comput. Phys., 227 (2008), pp. 3758–3780.

[14] M. O. Domingues, S. M. Gomes, O. Roussel, and K. Schneider, *Space-time adaptive multiresolution methods for hyperbolic conservation laws: Applications to compressible Euler equations*, Appl. Numer. Math., 59 (2009), pp. 2303–2321.

[15] D. L. Donoho, *Smooth wavelet decompositions with blocky coefficient kernels*, in Recent Advances in Wavelet Analysis, Academic Press, New York, 1993, pp. 259–308.

[16] B. Einfeldt, *On Godunov-type methods for gas dynamics*, SIAM J. Numer. Anal., 25 (1988), pp. 294–318.

[17] E. Elsen, P. LeGresley, and E. Darve, *Large calculation of the flow over a hypersonic vehicle using a GPU*, J. Comput. Phys., 227 (2008), pp. 10148–10161.

[18] P. N. Glaskowsky, *NVIDIA's Fermi: The First Complete GPU Computing Architecture*, Tech. report, NVIDIA, Santa Clara, CA, 2009.

[19] J. W. Grove, *Applications of front tracking to the simulation of shock refractions and unstable mixing*, Appl. Numer. Math., 14 (1994), pp. 213–237.

[20] J. F. Haas and B. Sturtevant, *Interaction of weak shock-waves with cylindrical and spherical gas inhomogeneities*, J. Fluid Mech., 181 (1987), pp. 41–76.

[21] T. R. Hagen, K. A. Lie, and J. R. Natvig, *Solving the Euler equations on graphics processing units*, Computational Science - ICCS 2006, 3994 (2006), pp. 220–227.

[22] A. Harten, *Adaptive multiresolution schemes for shock computations*, J. Comput. Phys., 115 (1994), pp. 319–338.

[23] B. Hejazialhosseini, D. Rossinelli, M. Bergdorf, and P. Koumoutsakos, *High order finite volume methods on wavelet-adapted grids with local time-stepping on multicore architectures for the simulation of shock-bubble interactions*, J. Comput. Phys., 229 (2010), pp. 8364–8383.

[24] R. L. Holmes, J. W. Grove, and D. H. Sharp, *Numerical investigation of Richtmyer-Meshkov instability using front tracking*, J. Fluid Mech., 301 (1995), pp. 51–64.

[25] M. Hopf and T. Ertl, *Hardware accelerated wavelet transformations*, in Proceedings of EG/IEEE TCVG Symposium on Visualization, IEEE, Washington, DC, 2000, pp. 93–103.

[26] X. Y. Hu, B. C. Khoo, N. A. Adams, and F. L. Huang, *A conservative interface method for compressible flows*, J. Comput. Phys., 219 (2006), pp. 553–578.

[27] T. Ishihara, T. Gotoh, and Y. Kaneda, *Study of high-Reynolds number isotropic turbulence by direct numerical simulation*, Ann. Rev. Fluid Mech., 41 (2009), pp. 165–180.

[28] G. S. Jiang and C. W. Shu, *Efficient implementation of weighted ENO schemes*, J. Comput. Phys., 126 (1996), pp. 202–228.

[29] I. C. Kampolis, X. S. Trompoukis, V. G. Asouti, and K. C. Giannakoglou, *CFD-based analysis and two-level aerodynamic optimization on graphics processing units*, Comput. Methods Appl. Mech. Engrg., 199 (2010), pp. 712–722.

[30] N. K.-R. Kevlahan and O. V. Vasilyev, *An adaptive wavelet collocation method for fluid-structure interaction at high Reynolds numbers*, SIAM J. Sci. Comput., 26 (2005), pp. 1894–1915.

[31] R. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, Cambridge University Press, Cambridge, UK, 2002.

[32] J. Liandrat and P. Tchamitchian, *Resolution of the* 1*D regularized Burgers equation using a spatial wavelet approximation*, Tech. report 90-83, 1CASE, NASA Contractor Report 18748880, 1990.

[33] X. D. Liu, S. Osher, and T. Chan, *Weighted essentially nonoscillatory schemes*, J. Comput. Phys., 115 (1994), pp. 200–212.

[34] E. E. Meshkov, *Instability of a shock wave accelerated interface between two gases*, NASA Tech. Trans., 1970.

[35] F. Miniati and P. Colella, *Block structured adaptive mesh and time refinement for hybrid, hyperbolic plus n-body systems*, J. Comput. Phys., 227 (2007), pp. 400–430.

[36] A. Munshi, *The OpenCL specification, version* 1.0, Khronos Group Std., Beaverton, OR, 2009.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

[37] Nvidia, *NVIDIA CUDA Compute Unified Device Architecture: Programming guide*, NVIDIA, Santa Clara, CA, 2007.

[38] A. Prosperetti and G. Tryggvason, eds., *Computational Methods for Multiphase Flow*, Cambridge University Press, Cambridge, UK, 2007, Ch. 3.

[39] L. Qianlong and O. V. Vasilyev, *A Brinkman penalization method for compressible flows in complex geometries*, J. Comput. Phys., 227 (2007), pp. 946–966.

[40] J. J. Quirk and S. Karni, *On the dynamics of a shock-bubble interaction*, J. Fluid Mech., 318 (1996), pp. 129–163.

[41] R. D. Richtmyer, *Taylor instability in shock acceleration of compressible fluids*, Commun. Pure Appl. Math., 13 (1960), pp. 297–319.

[42] D. Rossinelli, M. Bergdorf, G.-H. Cottet, and P. Koumoutsakos, *GPU accelerated simulations of bluff body flows using vortex particle methods*, J. Comput. Phys., 229 (2010), pp. 3316–3333.

[43] D. Rossinelli, M. Bergdorf, B. Hejazialhosseini, and P. Koumoutsakos, *Wavelet-based adaptive solvers on multi-core architectures for the simulation of complex systems*, in Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 721–734.

[44] O. Roussel, K. Schneider, A. Tsigulin, and H. Bockhorn, *A conservative fully adaptive multiresolution algorithm for parabolic PDEs*, J. Comput. Phys., 188 (2003), pp. 493–523.

[45] R. Saurel and R. Abgrall, *A simple method for compressible multifluid flows*, SIAM J. Sci. Comput., 21 (1999), pp. 1115–1145.

[46] C. E. Scheidegger, J. L. D. Comba, R. D. da Cunha, and N. Corporation, *Practical CFD simulations on programmable graphics hardware using SMAC*, Computer Graphics Forum, 24 (2005), pp. 715–728.

[47] K. Schneider and O. V. Vasilyev, *Wavelet methods in computational fluid dynamics*, Ann. Rev. Fluid Mech., 42 (2010), pp. 473–503.

[48] C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, and F. Tirado, *Parallel implementation of the 2D discrete wavelet transform on graphics processing units: Filter bank versus lifting*, IEEE Trans. Parallel Distributed Systems, 19 (2008), pp. 299–310.

[49] E. F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics*, Springer-Verlag, Berlin, 1999.

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

[51] J. H. Williamson, *Low-storage Runge-Kutta schemes*, J. Comput. Phys., 35 (1980), pp. 48–56.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.