An Efficient Compressible Multicomponent Flow Solver for Heterogeneous CPU/GPU Architectures

Fabian Wermelinger
Computational Science and Engineering Laboratory
ETH Zürich
Clausiusstrasse 33
CH-8092, Switzerland

Babak Hejazialhosseini
Cascade Technologies, Inc.
Palo Alto
CA 94303, USA

Panagiotis Hadjidoukas
Computational Science and Engineering Laboratory
ETH Zürich
Clausiusstrasse 33
CH-8092, Switzerland

Diego Rossinelli
Computational Science and Engineering Laboratory
ETH Zürich
Clausiusstrasse 33
CH-8092, Switzerland

Petros Koumoutsakos[*]
Computational Science and Engineering Laboratory
ETH Zürich
Clausiusstrasse 33
CH-8092, Switzerland

ABSTRACT

We present a solver for three-dimensional compressible multicomponent flow based on the compressible Euler equations. The solver is based on a finite volume scheme for structured grids and advances the solution using an explicit Runge-Kutta time stepper. The numerical scheme requires the computation of the flux divergence based on an approximate Riemann problem. The computation of the divergence quantity is the most expensive task in the algorithm. Our implementation takes advantage of the compute capabilities of heterogeneous CPU/GPU architectures. The computational problem is organized in subdomains small enough to be placed into the GPU memory. The compute intensive stencil scheme is offloaded to the GPU accelerator while advancing the solution in time on the CPU. Our method to implement the stencil scheme on the GPU is not limited to applications in fluid dynamics. The performance of our solver was assessed on Piz Daint, a XC30 supercomputer at CSCS. The GPU code is memory-bound and achieves a per-node performance of 462 Gflop/s, outperforming by 3.2× the multicore-based Gordon Bell winning CUBISM-MPCF solver [16] for the offloaded computation on the same platform. The focus of this work is on the per-node performance of the heterogeneous solver. In addition, we examine the performance of the solver across 4096 compute nodes. We present simulations for the shock-induced collapse of an aligned row of air bubbles submerged in water using 4 billion cells. Results show a final pressure amplification that is 100× stronger than the strength of the initial shock.

[*]Corresponding Author: petros@ethz.ch

PASC '16, June 08–10, 2016, Lausanne, Switzerland
© 2016 ACM. ISBN 978-1-4503-4126-4/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2929908.2929914

CCS Concepts

•**Computing methodologies → Massively parallel and high-performance simulations;** *Parallel computing methodologies;* Simulation types and techniques;

Keywords

Heterogeneous Architectures, GPGPU, Compute Accelerators, Texture Memory, Compressible Multicomponent Flow, Cavitation, Shock-Induced Bubble Collapse

1. INTRODUCTION

Cavitation collapse of air bubbles submerged in water induce pressure amplifications two orders of magnitude larger than the initial pressure configuration. The impact of the emitted shock waves on nearby surfaces causes material erosion that is detrimental to the life time of ship propellers or injection engines [17]. Medical applications such as shock wave lithotripsy harness the destructive power of cavitation to remove kidney stones [10].

We present a heterogeneous compressible multicomponent flow solver to study the shock-induced collapse of air bubbles submerged in liquid water. The compressible Euler equations are discretized on a structured grid using a finite volume scheme. The solution is advanced in time with an explicit low-storage Runge-Kutta method. Our solver exploits the compute power of GPUs for problems that are too large to fit into the GPU memory. We decompose the per-node workload into smaller work units, thus offloading the compute intensive part to the GPU by cycling through the data stored on the host processor. Textures are used to further exploit the spatial and temporal locality of the considered numerical schemes, alongside minimizing the L1D cache pollution by explicit use of two-dimensional thread blocks. The

Fluid	ρ [kg/m^3]	c [m/s]	γ	p_c [Pa]
Air	1.204	343	1.40	0
Water	1000	1450	6.12	3.43×10^8

Table 1: Density ρ, speed of sound c, specific heat γ and correction pressure p_c of air, helium and water at normal temperature and pressure [3].

resulting performance of the solver shows a 3.2× improvement over the multicore-based CUBISM-MPCF solver [16] for the offloaded computation on Piz Daint.

Previous work in the field of stencil computations on GPUs can be found in [12]. Recent work in the context of compressible single-phase flow is presented in [4]. The implementations in these works focus on GPU shared memory. Reference [9] and [11] present an approach which considers texture memory within the same context. A recent GPU accelerated solver for the two-phase incompressible Navier-Stokes equations is found in [21]. To the best of our knowledge, recent work in the context of GPU accelerated solvers for compressible multicomponent flow is not available.

The paper is structured as follows: The model equations and their discretization are introduced in Section 2. Section 3 describes our heterogeneous solver implementation using the CUDA/C++ languages. In Section 4 we present a performance analysis of our heterogeneous solver on Piz Daint. Simulation results for a shock-induced collapse of an array of 10 bubbles are discussed in Section 5. Finally, we summarize our findings in Section 6 and provide a perspective on our work for the next generation of GPUs.

2. EQUATIONS AND DISCRETIZATION

The equations for the flow model are given by the three-dimensional (3D) compressible Euler equations coupled with two transport equations to accommodate a multicomponent composition. The evolution of density, momenta and total energy of the fluid mixture is described by the conservation laws

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0, \tag{1a}$$

$$\frac{\partial (\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u}^{\intercal} + p\boldsymbol{I}) = \boldsymbol{0}, \tag{1b}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot ([E+p]\,\boldsymbol{u}) = 0. \tag{1c}$$

The closure of the system is provided by the stiffened equation of state

$$\Gamma p + \Pi = E - \frac{1}{2}\rho \boldsymbol{u}\cdot\boldsymbol{u}, \tag{2}$$

where $\Gamma = 1/(\gamma - 1)$ and $\Pi = \gamma p_c/(\gamma-1)$. The ratio of specific heats γ and the correction pressures p_c are empirical parameter used to describe each fluid phase. Values used for this work are specified in Table 1. The evolution of the two additional parameter in (2) is coupled to the governing system (1) by the transport equation

$$\frac{\partial \phi}{\partial t} + \boldsymbol{u}\cdot\nabla\phi = 0, \tag{3}$$

where $\phi = \{\Gamma, \Pi\}$. The multicomponent system is described by the variables $\boldsymbol{U} = (\rho, \rho\boldsymbol{u}, E, \Gamma, \Pi)$ according to equations (1), (2) and (3). The governing system (1) and (3) is written in quasi-conservative form. We rewrite it in flux-divergence form as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}}{\partial x} + \frac{\partial \boldsymbol{G}}{\partial y} + \frac{\partial \boldsymbol{H}}{\partial z} = \boldsymbol{R}, \tag{4}$$

where $\boldsymbol{R} = (0,0,0,0,0,\phi\nabla\cdot\boldsymbol{u})$ is a source term and $\boldsymbol{F}, \boldsymbol{G}, \boldsymbol{H}$ are flux functions

$$\boldsymbol{F} = \begin{pmatrix} \rho u_x \\ \rho u_x^2 + p \\ \rho u_y u_x \\ \rho u_z u_x \\ (E+p)u_x \\ \phi u_x \end{pmatrix}, \boldsymbol{G} = \begin{pmatrix} \rho u_y \\ \rho u_x u_y \\ \rho u_y^2 + p \\ \rho u_z u_y \\ (E+p)u_y \\ \phi u_y \end{pmatrix}, \boldsymbol{H} = \begin{pmatrix} \rho u_z \\ \rho u_x u_z \\ \rho u_y u_z \\ \rho u_z^2 + p \\ (E+p)u_z \\ \phi u_z \end{pmatrix}.$$

We are interested in weak solutions of system (4). To maintain the correct physical behavior in discontinuous flow regions, a stable approximation is obtained by using a shock-capturing scheme. The discretization is based on a semi-discrete approach for structured 3D grids. For the present work, we use a uniform grid with a total of N cells. The finite-volume method is used for discretizing the governing system (4). Approximation of the spatial terms reduces the system to a time continuous system of ordinary differential equations

$$\frac{d\boldsymbol{V}(t)}{dt} = \mathcal{L}(\boldsymbol{V}(t)), \tag{5}$$

where the vector $\boldsymbol{V}(t) \in \mathbb{R}^N$ is a representation of the cell averaged variables $\boldsymbol{U}$ for the complete computational domain. Finally, we integrate (5) using an explicit third-order low-storage Runge-Kutta scheme [20].

The numerical operator $\mathcal{L}(\cdot)\colon \mathbb{R}^N \to \mathbb{R}^N$ returns the right-hand side (RHS) of Equation (5). It provides an approximation to the convective terms in (4) and is computationally the most intensive part when integrating (5). Furthermore, three evaluations are required for every timestep due to the three stages of the numerical integrator. Given the uniform mesh, we apply a dimensional splitting and write

$$\mathcal{L} = \mathcal{L}^x + \mathcal{L}^y + \mathcal{L}^z \tag{6}$$

for the individual contributions in x, y and z direction. The contribution of $\mathcal{L}^x$ for cell i,j,k reads

$$\mathcal{L}^x_{i,j,k} = -\frac{1}{h}\left(\mathcal{F}^n_{i+1/2,j,k} - \mathcal{F}^n_{i-1/2,j,k}\right), \tag{7}$$

where h is the uniform grid spacing and $\mathcal{F}^n_{i+1/2}$ is an approximation to the intercell flux $\boldsymbol{F}$ at cell face location $i+1/2$ for the time interval $[t_n, t_{n+1})$. The discrete time is given by $t_n = \sum_n \Delta t_n$ with variable timestep Δt_n. Subscripts j,k are omitted for simplicity. The remaining contributions of the dimensional splitting are defined analogously to (7). Note that the flux approximations depend on the cell averages $\boldsymbol{V}(t_n)$.

The numerical flux is determined by solving an approximate Riemann problem at each of the cell interfaces. For the present work, we use the Harten, Lax, van Leer contact scheme (HLLC) introduced by Toro [18] due to its superior resolution of interfaces, which is essential in multicomponent flows. Hence, the numerical flux at cell face $i+1/2$ is

Feature	Nominal Value
Chip	GK110
Primary Clock	732 MHz
Number of SP Cores	2688
Number of DP Cores	896
Peak SP Performance	3.94 Tflop/s
Peak DP Performance	1.31 Tflop/s
Memory Clock	2.6 GHz
GDDR5 Memory	6 GB
Memory Bandwidth	250 GB/s
PCIe Bus	Gen2.0 ×16

Table 2: Hardware specification for the Nvidia K20X GPU.

computed by

$$\begin{aligned}\mathcal{F}^n_{i+1/2} = &\frac{1+\mathrm{sgn}(s^*)}{2}\big(\boldsymbol{F}(\boldsymbol{U}_L) + s^-(\boldsymbol{U}^*_L - \boldsymbol{U}_L)\big)\\ &+\frac{1-\mathrm{sgn}(s^*)}{2}\big(\boldsymbol{F}(\boldsymbol{U}_R) + s^+(\boldsymbol{U}^*_R - \boldsymbol{U}_R)\big),\end{aligned} \tag{8}$$

where

$$s^- = \min(s_L, 0), s^+ = \max(0, s_R), \mathrm{sgn}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0. \end{cases}$$

Wavespeed estimates s_L and s_R, for the slowest and fastest moving waves, respectively, are obtained from Einfeldt [5]. The wavespeed estimate for the contact wave is computed by following Batten [1],

$$s^* = \frac{\rho_R u_R(s_R - u_R) - \rho_L u_L(s_L - u_L) + p_L - p_R}{\rho_R(s_R - u_R) - \rho_L(s_L - u_L)}. \tag{9}$$

The initial states for the Riemann problem $\boldsymbol{U}_k$ for $k = \{L, R\}$ are obtained by a fifth-order weighted essentially non-oscillatory (WENO) reconstruction [7]. Approximation of the intermediate states $\boldsymbol{U}^*_k$ for $k = \{L, R\}$ are computed according to Toro [18]. Finally, $\phi\nabla\cdot\boldsymbol{u}$ in the vector $\boldsymbol{R}$ of Equation (4) is computed such that the velocity $\boldsymbol{u}$ is consistent with s^* at interfaces with strong pressure variations [8]. To prevent pressure oscillations at material interfaces, the WENO reconstruction is performed using primitive variables $\boldsymbol{W} = (\rho, \boldsymbol{u}, p, \Gamma, \Pi)$ [8].

3. SOFTWARE DESIGN

In this section we describe the design and implementation of our solver on a heterogeneous CPU/GPU architecture. We begin with an overview of the steps required to advance the solution in time and continue with a detailed description of the implemented GPU kernels. Our target GPU is the Nvidia K20X, available on the Piz Daint XC30 supercomputer. Table 2 shows the main hardware features for this GPU. Our code is written using the C++ and CUDA C/C++ languages.

3.1 Algorithm

The vector $\boldsymbol{V}$ is stored in a Structure of Array (SoA) format in host memory and is organized into K partitions that fit into the GPU memory. The data partition is virtual and does not modify the location of data in memory. Figure 1 illustrates the partition on $\boldsymbol{V} \in \mathbb{R}^N$, where $N = N_x \times N_y \times N_z$

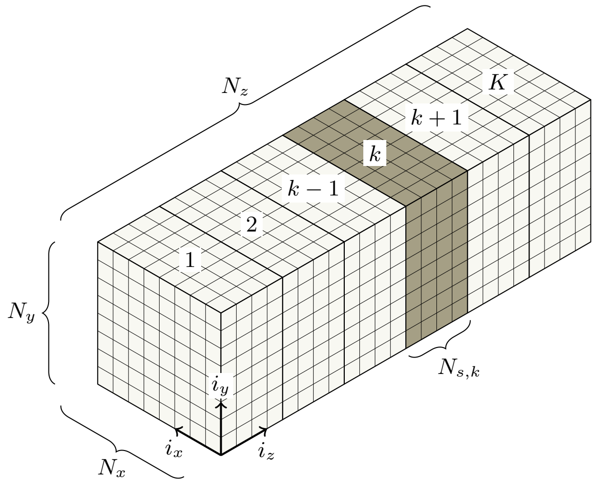


Figure 1: Partition of the computational domain into small enough segments for the GPU memory.

is the total number of cells. The number of cells for segment k is $N_k = N_x \times N_y \times N_{s,k}$ with $N_z = \sum_k N_{s,k}$. We perform the partition along i_z which is the slowest moving index. Algorithm 1 illustrates the main steps required to process one stage of the explicit Runge-Kutta time stepping. The notation $\boldsymbol{V}^k$ is used to denote the data in $\boldsymbol{V}$ that corresponds to segment k. For each of the flow variables, communication with the GPU is organized into a pair of pinned memory buffers, each of size N_k. The first buffer is used for GPU input, whereas the second buffer is used to receive GPU output. The left part of Figure 2 illustrates two buffer pairs that are utilized on the CPU. Similarly, a pair of global memory buffers is employed on the GPU. The first buffer is used for input and output (I/O) with the CPU. The second buffer is a CUDA array used for read-only kernel input by binding to texture references. We use texture memory to take full advantage of spatial and temporal locality. The right part of Figure 2 illustrates two buffer pairs utilized on the GPU, where we denote the I/O buffer by A and the CUDA array by B.

Algorithm 1 Processing of a Runge-Kutta Stage

```
 1: $k \leftarrow 1$
 2: Copy $V^k$ into input buffer of pair ($k$ mod 2)
 3: Enqueue GPU tasks on CUDA stream ($k$ mod 2)
 4: while $k < K$ do
 5:     if $k > 1$ then
 6:         Synchronize with CUDA stream ($k-1$ mod 2)
 7:         Update $V^{k-1}$
 8:     end if
 9:     $k \leftarrow k + 1$
10:     Copy $V^k$ into input buffer of pair ($k$ mod 2)
11:     Enqueue GPU tasks on CUDA stream ($k$ mod 2)
12: end while
13: Synchronize with CUDA stream ($k$ mod 2)
14: Update $V^k$
```

The processing of the segments in Algorithm 1 is implemented by cycling through the two buffer pairs to allow for simultaneous processing of two segments. The arrows indicated in Figure 2 illustrate the cycle. The algorithm starts

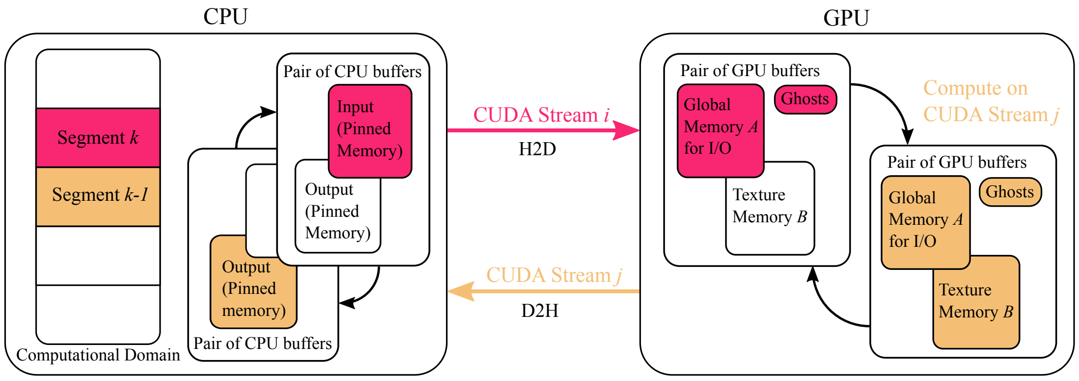


Figure 2: Illustration of the set of buffer pairs used on the CPU and GPU.

Kernel	Registers per thread	Occupancy
Convert	24	100 %
WENO5 (i_x, i_y)	29	100 %
WENO5 (i_z)	32	100 %
HLLC (i_x)	32	100 %
HLLC (i_y, i_z)	56	56 %
Transpose (i_x)	64	50 %

Table 3: GPU register usage per thread and occupancy for thread blocks with 4 warps.

by copying the data for the first segment into the pinned input of a buffer pair identified by a modulo operation. The tasks to be executed on this segment are enqueued on a CUDA stream in a next step. The following tasks are enqueued in order:

1. Asynchronous host to device copy (H2D)
2. Launch compute kernels to evaluate the right-hand side $\mathcal{L}$ on segment k
3. Asynchronous device to host copy (D2H)

Successive segments are processed in a loop similar to the first segment. The loop body performs the Runge-Kutta update on the previous segment and initializes the next segment by increasing the counter k. The data $\boldsymbol{V}^k$ for the next segment is then copied into the pinned input of the buffer pair identified by $(k \bmod 2)$. To ensure completion of the asynchronous device to host (D2H) copy operation, the CPU must synchronize with the CUDA stream identified by $(k-1 \bmod 2)$, before performing the Runge-Kutta update on data $\boldsymbol{V}^{k-1}$. Finally, the update for the last segment K must be performed after termination of the loop.

The host to device (H2D) operation asynchronously copies the data from the pinned input buffer on the CPU to the global I/O buffer A on the GPU, see Figure 2. Results computed on the GPU are written into the global I/O buffer A. After completion of the compute kernels, the data is asynchronously copied back (D2H) into the pinned output buffer on the CPU. The two copy engines on the K20X GPU allow for further overlap of the H2D and D2H operations.

Additional memory is required for the communication of ghost cells due to the stencil of the WENO reconstruction. Ghost cells are obtained on the CPU by either evaluating a boundary condition or communicating with neighboring nodes through the message passing interface library (MPI). The H2D operation includes transfer of ghost cells, which are copied into global memory.

Algorithm 1 is executed three times to perform one timestep. The timestep size Δt_n is determined based on the CFL condition and data $\boldsymbol{V}$ prior to the execution of Algorithm 1. We compute the timestep size on the CPU to avoid additional pressure on the PCIe bus.

3.2 GPU Kernels

This section describes the implementation of the GPU kernels required to evaluate the right-hand side operator $\mathcal{L}$ on a segment k. The approximate Riemann problem discussed in Section 2 is solved by the steps shown in Algorithm 2. The first step is a conversion of the flow quantities from

Algorithm 2 Evaluation of $\mathcal{L}$

```
1: Convert data in $A$ to primitive variables $\boldsymbol{W}$ (in-place)
2: Copy data in $A$ into CUDA 3D array $B$
3: Bind $B$ to global texture reference
4: for all $j \in \{x, y, z\}$ do
5:     WENO reconstruction from data $B$ in direction $j$
6:     HLLC flux divergence in direction $j$
7: end for
```

variables $\boldsymbol{U}$ to primitive variables $\boldsymbol{W}$. This step is required as discussed in Section 2. The conversion kernel operates on each data point independently and writes the converted value back to the same memory location. In the second step, the CUDA API reorders the data based on a space filling curve (SFC) and stores the result in a 3D CUDA array. The remaining steps are the approximation of the Riemann problem on cell interfaces and the computation of the flux divergence in all spatial directions.

A single kernel implementation of Algorithm 2 would lead to an excessive register pressure, in turn resulting in poor occupancy levels and a frequent register spilling. Based on a roofline analysis, a $2\times$ improvement could be estimated for a series of lightweight kernel invocations instead of one single kernel invocation.[1] Our implementation achieves a $1.7\times$ improvement compared to a single kernel. For these reasons, we organize Algorithm 2 into a set of lightweight kernels that are executed in sequence. The first kernel per-

[1]Assuming compulsory cache misses only.

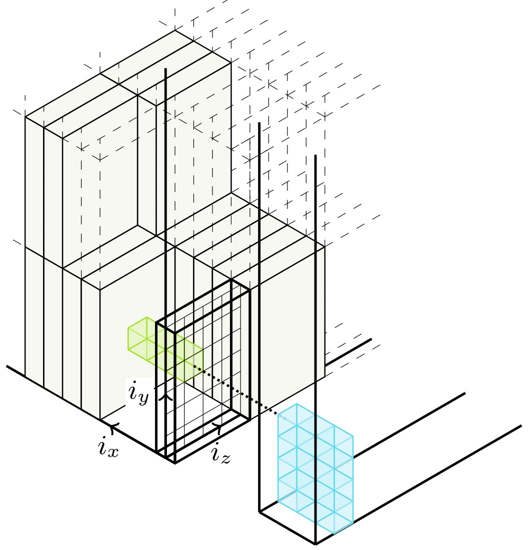


Figure 3: Kernel grid and thread block layout for the reconstruction and HLLC flux computation in x direction. Stencil data for boundary blocks is obtained from global memory (right half of stencil in the negative i_x space, blue cells) as well as texture memory (left half of the stencil in the positive i_x space, green cells).

forms the conversion to primitive variables. A second templated kernel is used for the reconstruction in each of the three spatial directions. A fused kernel is used for both the computation of the HLLC flux and the divergence in y and z directions, respectively. Combining the two steps avoids additional memory accesses but is more expensive in terms of register usage compared to the WENO reconstruction. Table 3 shows the number of registers per thread for each of the lightweight kernels. The computation of the flux divergence in x direction must be split in two kernel calls, where the second kernel involves a data transposition to ensure coalesced memory accesses as discussed below.

WENO Reconstruction

The fifth-order WENO reconstruction is based on a 6-point stencil, which requires ghost cells at subdomain boundaries. We employ two-dimensional (2D) CUDA thread blocks of 4 warps that map onto cell faces normal to the direction of reconstruction. We implement the 3D stencil scheme by using a three-pass approach, where the 2D thread blocks are aligned in normal direction for each pass. Figure 3 shows the thread block alignment for the x direction. The stencil input data is obtained from texture memory. The choice of 2D thread blocks explicitly enforces temporal locality which allows to harness the data cache efficiently. In contrast, the work of Micikevicius [12] employs a single-pass strategy, where shared memory is used to minimize read redundancy and 2D thread blocks are aligned static along the z direction.

Thread blocks at subdomain boundaries require data from two different memory locations. The left part of the stencil fetches data from the textures. The data for the right part of the stencil is obtained from a global memory buffer. Figure 3 illustrates this special case. The choice of 2D thread blocks additionally avoids warp divergence due to irregularities at domain boundaries. Thread blocks entirely inside the domain get all of the data from texture memory, which is the case for the majority of blocks. Reconstruction in y and z direction is similar to the x direction. Ghost cells for reconstruction in z direction are directly embedded into the textures.

HLLC Flux Divergence

The last step in the loop of Algorithm 2 computes the flux divergence of Equation (7) based on the HLLC flux shown in Equation (8). The flux divergence for individual directions is added up according to the dimensional splitting in Equation (6) and written into the global memory buffer A, see Figure 2. To guarantee coalesced writes into buffer A, the computation of the flux divergence in the x direction must be split into two parts. The first part computes the HLLC flux on all cell faces normal to x. Because the 2D thread blocks map to cell faces, it is not possible for a warp to span along the i_x index, see Figure 3. Therefore, writes to the buffer A are non-coalesced in that particular case. To avoid this problem, a second kernel is launched that transposes the data using shared memory. The flux divergence is then computed from the transposed data and written back into buffer A.

3.3 MPI

We rely on the message passing interface (MPI) for running simulations on distributed memory architectures. Processes are organized on a cartesian topology for point-to-point communication of ghost cells, required at each Runge-Kutta stage. The current implementation exchanges ghost cells for all K segments before a Runge-Kutta stage is executed. Ghost cells are then extracted from the MPI messages during the memory copies in Algorithm 1. This implementation does reduce the granularity of message exchange but does not allow for compute/transfer overlap. Similar techniques for compute/transfer overlap already used in [16] can be applied in this context. Additionally, at the start of every timestep n, a reduction is required for the computation of the timestep size Δt_n.

4. PERFORMANCE

The performance of our solver was evaluated on Piz Daint. A compute node on Piz Daint consists of an 8-core Intel Xeon E5-2670 CPU and a Nvidia Tesla K20X GPU. We rely on GCC 4.7.2 and the CUDA 6.5 toolkit. The Nvidia nvprof profiler from the CUDA toolkit was used for measuring GPU performance figures. All measurements were performed on simulations running in single precision.

4.1 GPU

In this section, we show the performance of the implemented GPU kernels by means of the roofline model [19]. We utilize the scalable heterogeneous computing benchmark suite[2] (SHOC) to build the roofline ceiling of the memory-bound region. A small compute kernel has been written to build the ceiling for peak performance. We measure a memory bandwidth of 176.3 GB/s (70.5 % of nominal peak) and a single precision performance of 3.51 Tflop/s (89.2 % of nominal peak). ECC was enabled for the memory bandwidth

[2]http://keeneland.gatech.edu/software/keeneland/shoc

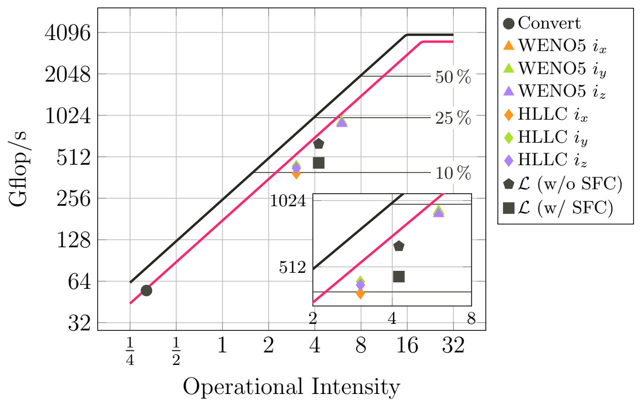


Figure 4: Roofline for the Nvidia K20X GPU with measured kernel performances.

measurements. Figure 4 shows the roofline for the K20X GPU on a single compute node. All of our compute kernels exhibit excellent performance on the K20X. References for 10 %, 25 % and 50 % of nominal peak performance are indicated by the black roofline. The red ceilings correspond to our measurements based on the micro benchmarks mentioned above. The operational intensity of the GPU kernels is calculated under the assumption of compulsory memory reads only. The conversion kernel has an operational intensity of 0.3 Flop/Byte. We use the Nvidia profiler to count the number of floating point instructions and measure the runtime of the kernel. From these numbers, we determine a performance of 54.8 Gflop/s. This corresponds to 97.1 % of the performance predicted by the roofline ceiling and is indicated by the black circle in Figure 4. The template for the reconstruction kernel has an operational intensity of 6 Flop/Byte. Because of the high floating point instruction density, the WENO reconstruction kernel is expected to show the highest performance among the other kernels. Profiling for this kernel yields a performance of 907.6 Gflop/s, which corresponds to 85.8 % of the roofline prediction and 23 % of nominal peak performance. The WENO reconstruction is indicated by the triangles in Figure 4. The operational intensity for the fused HLLC kernel is 3 Flop/Byte. Note that we assume the same operational intensity for the split HLLC kernel in the x direction. The measured performance for this kernel is 430.9 Gflop/s for the y and z directions and 388.4 Gflop/s for the x direction. These numbers are indicated by the diamonds in Figure 4 and correspond to 81.5 % and 73.4 % of the roofline prediction, respectively. The performance of the HLLC computation in x direction is lower due to the data transposition required to ensure coalesced memory access. The measured throughput of the transposition kernel is 136.3 GB/s, corresponding to 77.3 % of the bandwidth measured by the SHOC benchmark. We limit the register usage of the HLLC kernel in x direction to 32 registers per thread by declaring individual launch bound specifiers [14]. This allows for running more thread blocks simultaneously at a small number of register spills. The transposition kernel is not forced by launch bounds and requires 64 registers per thread, see Table 3. The measurement for HLLC i_x in Figure 4 corresponds to this configuration.

Finally, if all of the kernel calls are combined, an overall operational intensity of 4.2 Flop/Byte is computed for the

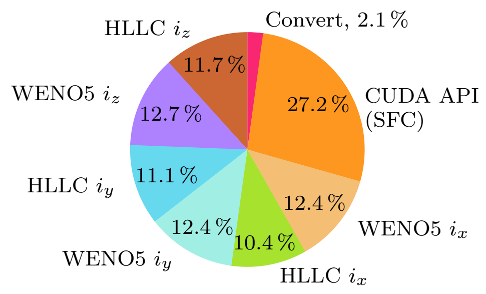


Figure 5: Execution time distribution for the kernels on the K20X.

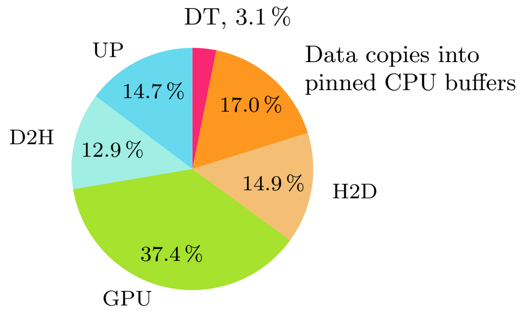
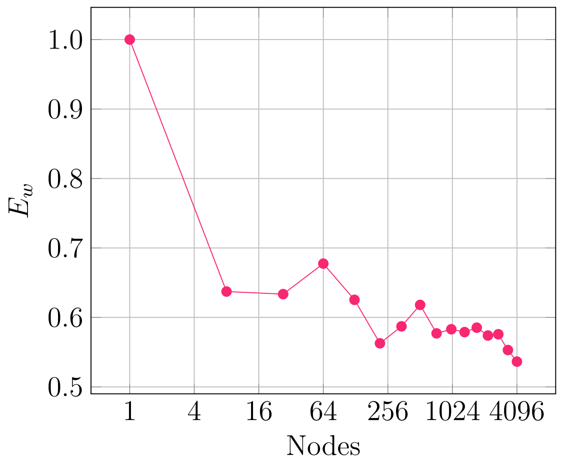


Figure 6: Execution time distribution for the completion of timestep n. The abbreviation DT stands for the computation of timestep Δt_n, H2D/D2H for host to device and device to host copies, respectively, and UP for the Runge-Kutta update.

evaluation of $\mathcal{L}$ (per segment). Measuring the number of floating point instructions as well as the execution time of a single evaluation of $\mathcal{L}$ yields a performance of 462.4 Gflop/s. This corresponds to 62% of the roofline prediction and 11.7% of nominal peak performance. The measurement is indicated by the black square in Figure 4. The performance of the CUBISM-MPCF flow solver is 143.8 Gflop/s on one node for the same problem [16]. Compared to this measurement, the present GPU implementation offers a 3.2× speedup in terms of the right-hand side computation in Equation (5). Furthermore, the CUBISM-MPCF performance for a BlueGene/Q node reported in [6] is 149.1 Gflop/s, corresponding to a speedup of 3.1× for the same problem. A maximum speedup of 1.4× for techniques involving textures is reported in Table 1 of reference [13]. Moreover, the GPU-CPU improvement is consistent with the maximum speedup imposed by the faster GDDR5 memory on the GPU with respect to the CPU.

The execution time distribution of the GPU kernels is shown by the pie-chart in Figure 5. The GPU kernels indicated in that figure correspond to the steps executed in Algorithm 2. The CUDA API slice (SFC) corresponds to the CUDA 3D array copy of step two of Algorithm 2. Each of the three WENO slices involves seven kernel invocations, one for each primitive variable in $\boldsymbol{W}$. The HLLC slices involve two kernel invocations for the i_x direction and one fused kernel invocation for the i_y and i_z directions. A large amount of time is spent in the CUDA 3D array copy function, which is needed for texture memory. Copying the data



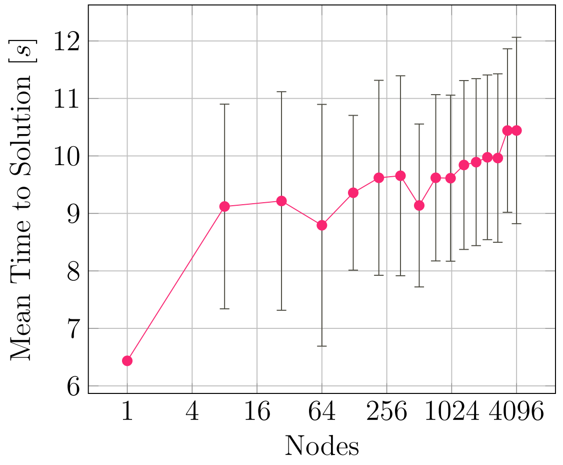


Figure 7: Weak efficiency for up to 4096 nodes on Piz Daint (left) and time to solution (right) averaged over nodes.

into the CUDA 3D array on the GPU is still faster than converting to primitive variables on the CPU and performing the CUDA array copy while sending the data to the GPU. The reordering of the 3D data by the CUDA 3D array copy operation ensures spatial locality and therefore an efficient use of the texture cache. The gap between the evaluation of the right-hand side $\mathcal{L}$ (black square) and the roofline ceiling in Figure 4 is due to the expensive data reordering. For example, if we neglect the CUDA 3D copy operation, the roofline predicts an overall performance of 635.4 Gflop/s corresponding to 86 % of the predicted performance and 16 % of nominal peak performance. The black pentagon in the roofline of Figure 4 illustrates this case.

Figure 6 shows the execution time distribution for the completion of timestep n. The pie-chart in Figure 6 corresponds to a single evaluation of Δt_n followed by the execution of Algorithm 1 (three times). The copy operations into pinned memory buffers correspond to line 2 and 10 in Algorithm 1. The GPU slice corresponds to the time spent in Algorithm 2. A detailed pie-chart for this slice is shown in Figure 5. The H2D transfers are slightly more expensive compared to the D2H transfers due to the additional ghost cells that must be sent to the GPU.

4.2 Weak Scaling

We have conducted a weak scaling study on 4096 nodes of the Piz Daint cluster using a domain with 110 Million cells on each node. The domain is decomposed into four segments, where each segment contains roughly 28 Million cells. The decomposition occupies 96 % of the K20X memory, accounting for two sets of buffers on the GPU as well as for ECC. Domain boundaries are periodic. Figure 7 shows the weak efficiency (left) and time to solution (right) averaged over nodes. The current implementation exchanges ghost cells for all of the segments on a node. Because the processing of individual segments depends on the completion of inter-node communication, the loss in efficiency is justified by the lack of compute/transfer overlap in our present solver. Furthermore, we observe a large deviation in the mean time to solution for more than one node. One reason for this observation is network contention due to other running jobs on the cluster. The data shown in Figure 7 accounts for 20 measurements. A highly tuned MPI implementation was beyond the goals of this work.

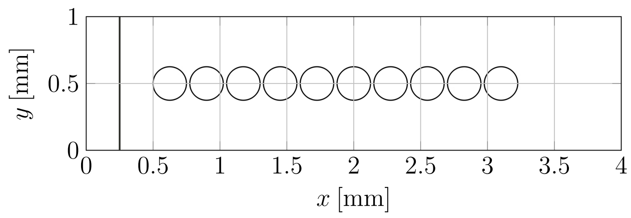


Figure 8: Initial condition for 10 bubbles aligned on a straight line. The position of the initial 40 MPa planar shock is at $x_s = 0.25$ mm.

5. SIMULATION

We simulate the shock-induced collapse (SIC) of 10 equally-spaced and identical bubbles with radii $r_b = 125\ \mu$m. Bubbles are aligned on a straight line through the center of the domain with a gap of $r_b/5$ between neighboring bubble shells. The domain extent is $4 \times 1 \times 1$ mm in the x, y and z direction, respectively. The gas inside the bubbles is non-condensable air, initially in equilibrium at 1bar with the surrounding water. The specific heats and correction pressures for the two materials are given in Table 1. The collapse of the bubbles is initiated by a planar shock wave located at $x_s = 0.25$ mm. The pressure jump across the shock wave is $\Delta p_0 = 400$ bar. The speed of the wave is roughly equal to the speed of sound in water at the initial conditions. Figure 8 illustrates a numerical schlieren [15] visualization for the initial condition in the xy-plane at $z = 0.5$ mm. The air/water interfaces are illustrated by the circles and the initially planar shock is visible by the vertical line.

The domain is discretized with 4096 cells in the x direction and 1024 cells in the y and z direction, respectively. The total number of cells used for this simulation is 4.29 Billion. Such discretization leads to a resolution of 128 p.p.r.[3]for r_b and a gap resolution of 25 cells. The simulation was run

[3]Points per radius, where a point is equivalent to a cell.

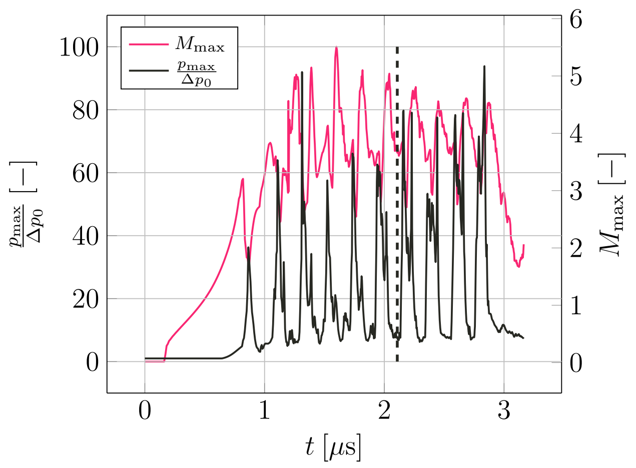


Figure 9: Temporal evolution of the pressure amplification factor and maximum Mach number.

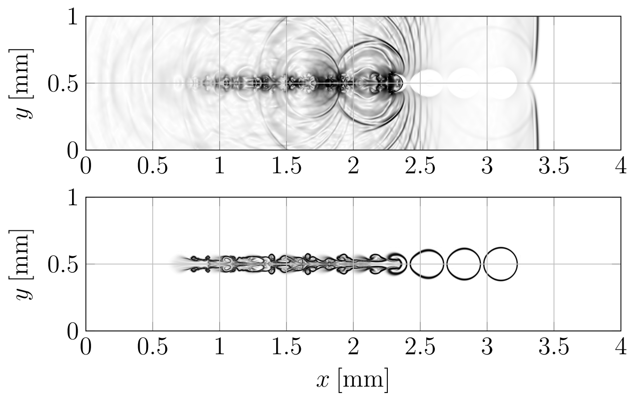


Figure 10: Schlieren images of the pressure wave structure (top) and air/liquid interfaces (bottom) at time $t = 2.11$ μs.

in single precision on 64 nodes of Piz Daint for 32800 steps until $t = 3.16$ μs. The CFL number is 0.32, resulting in an average timestep of 100 ps. We specify a Dirichlet inflow boundary and absorbing at the remaining boundaries.

The planar shock impacts the proximal surface of the first bubble after 0.16 μs. This generates a supersonic compression wave inside the bubble and causes the local maximum Mach number to rise. The sudden pressure jump at the proximal side and the shielding of the downstream bubble force a non-spherical collapse of the bubble. At the final stage of the collapse, the bubble emits a strong pressure shock that extenuates rapidly as it propagates through the liquid. Just before the emission, a high kinetic energy jet develops on the proximal bubble side that pierces through the bubble center and eventually transforms into potential energy. Figure 9 illustrates the temporal evolution of the pressure amplification $p_{\max}/\Delta p_0$ and the maximum Mach number. Termination of a bubble collapse is identified by a strong peak in the pressure amplification. Each peak is preceded by an immediate decrease of the Mach number due to the kinetic/potential energy transformation. This process continues until the collapse of the final bubble with a pressure amplification of two orders of magnitude relative to the initial shock strength.

Collapse Type	Collapse Time
Single Bubble SIC	654 ns
Rayleigh-Plesset	573 ns

Table 4: Collapse time for a single bubble SIC case and an inertia driven Rayleigh-Plesset collapse [2].

Figure 10 illustrates schlieren images for the xy-plane at $z = 0.5$ mm at time $t = 2.11$ μs. The top image shows the pressure wave structure, where dark regions are compression waves and continuous gray regions expansion waves. The bottom image shows the air/liquid interfaces. The dashed line in Figure 9 corresponds to the time instant of images shown in Figure 10. Figure 11 shows a rendering of the pressure field and iso-surfaces for the air/liquid interface. The iso-value for the shown image is set to $\Gamma = 2.4$. The dense orange fields correspond to pressure around 2500 bar while thin green fields corresponds to pressure around 800 bar.

The shock-induced collapse time for a single bubble and for an inertia driven Rayleigh-Plesset collapse [2] are given in Table 4. The single bubble SIC case is computed with the present solver at similar resolution and precision. The collapse time is measured from the shock impact on the proximal side of the bubble until the emission of the strong shock wave that terminates the collapse. The collapse time of the single SIC bubble is expected to be longer compared to the Rayleigh-Plesset collapse due to the time required for the shock to traverse the bubble as well as the non-spherical collapse. The measured collapse time for the first bubble in the array is 688 ns, which is longer than the single SIC bubble due to the shielding effect of the downstream bubble. As the initial planar shock is ruptured by the presence of the cavities, it is difficult to define a shock impact on proximal bubble sites for successive collapses. The Mach number in Figure 9 increases immediately after a pressure peak is observed, indicating a fast formation of the next piercing jet at the downstream bubble. This event appears almost immediately after a pressure peak is observed. Therefore, we identify the collapse time for successive bubbles by the time interval between two peaks in pressure. We measure a collapse time of 212 ± 15 ns for the successive collapses, three times faster than the collapse of the leading bubble.

6. CONCLUSION

We have presented a technique for implementing stencil computations on the GPU using textures. Our explicit choice of aligned 2D thread blocks enforces temporal locality and allows for optimal utilization of the data cache for each pass. The approach is general and can be straightforwardly applied to other stencil problems on structured grids. We have demonstrated a GPU based implementation of a compressible multicomponent flow solver that is 3.2× faster compared to the Gordon-Bell winning CUBISM-MPCF [16] multicore-based flow solver for the offloaded right-hand-side computation. The overall floating point performance of the right-hand side computation is 462.4 Gflop/s for the Nvidia K20X GPU on one node of the Piz Daint XC30 supercomputer. The observed performance corresponds to 11.7 % of the nominal single precision peak performance for the K20X.

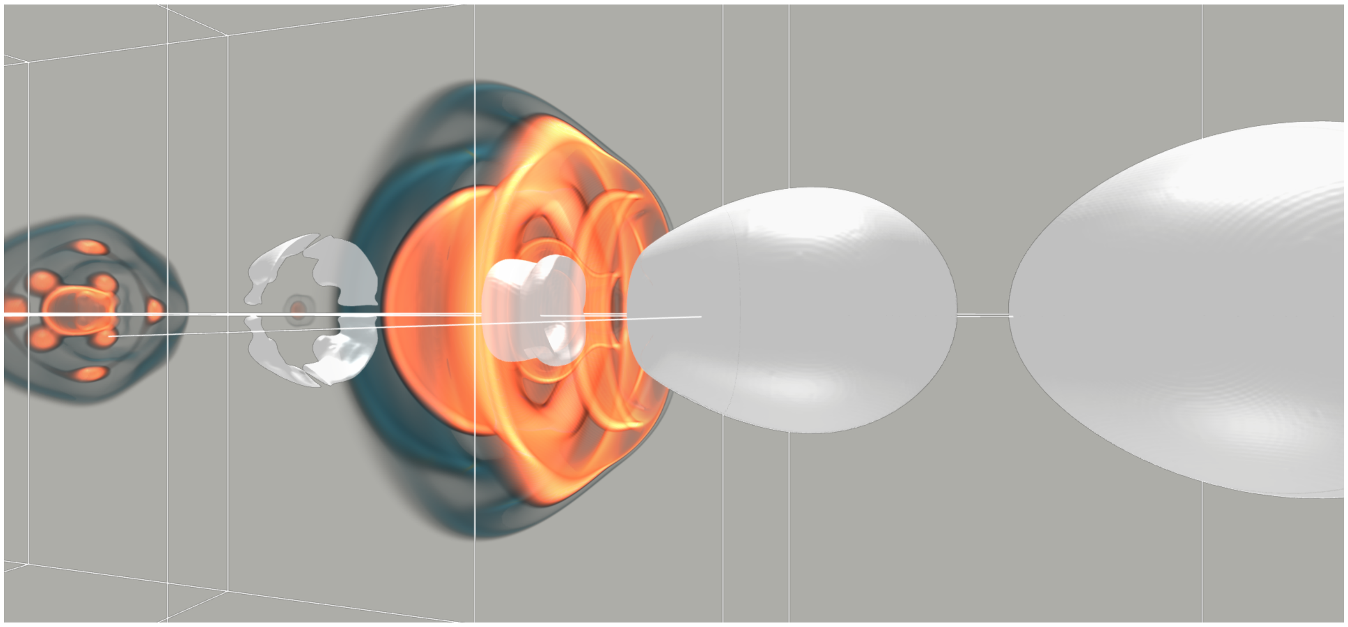


Figure 11: Volume rendering of pressure at time $t = 1.78\,\mu$s. Iso-surfaces represent air/liquid interfaces. Dense orange corresponds to pressure around $2500\,\mathrm{bar}$ while thin green corresponds to pressure around $800\,\mathrm{bar}$. Rebounds are observed upstream for previous bubble collapses.

Future work will focus on improved compute/transfer overlap. Furthermore, the overhead due to the SFC reordering will be addressed by using a memory layout that resembles a given static ordering for the data stored on the CPU.

7. ACKNOWLEDGMENTS

The authors would like to thank the support staff of the Swiss Supercomputing Center (CSCS) for granting access to Piz Daint through project s500.

8. REFERENCES

[1] P. Batten, N. Clarke, C. Lambert, and D. Causon. On the choice of wavespeeds for the HLLC Riemann solver. *SIAM Journal on Scientific Computing*, 18(6):1553–1570, 1997.

[2] C. E. Brennen. *Cavitation and bubble dynamics.* Cambridge University Press, 2013.

[3] V. Coralic and T. Colonius. Finite-Volume WENO scheme for viscous compressible multicomponent flows. *Journal of Computational Physics*, 274:95–121, 2014.

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

[5] B. Einfeldt. On Godunov-type methods for gas dynamics. *SIAM Journal on Numerical Analysis*, 25(2):294–318, 1988.

[6] P. Hadjidoukas, D. Rossinelli, F. Wermelinger, J. Sukys, U. Rasthofer, C. Conti, B. Hejazialhosseini, and P. Koumoutsakos. High throughput simulations of two-phase flows on BlueGene/Q. Advances in Parallel Computing, ParCo, 2015. Submitted.

[7] G.-S. Jiang and C.-W. Shu. Efficient implementation of weighted ENO schemes. *Journal of Computational Physics*, 126(1):202–228, 1996.

[8] E. Johnsen and T. Colonius. Implementation of WENO schemes in compressible multicomponent flow problems. *Journal of Computational Physics*, 219(2):715 – 732, 2006.

[9] K. I. Karantasis, E. D. Polychronopoulos, and J. A. Ekaterinaris. High order accurate simulation of compressible flows on GPU clusters over software distributed shared memory. *Computers & Fluids*, 93:18–29, apr 2014.

[10] T. Kodama and K. Takayama. Dynamic behavior of bubbles during extracorporeal shock-wave lithotripsy. *Ultrasound in Medicine & Biology*, 24(5):723–738, 1998.

[11] C. Meng, L. Wang, Z. Cao, L. long Feng, and W. Zhu. Large-scale parallelization based on CPU and GPU cluster for cosmological fluid simulations. *Computers & Fluids*, 110:152–158, mar 2014.

[12] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*. Association for Computing Machinery (ACM), 2009.

[13] K. E. Niemeyer and C.-J. Sung. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *J Supercomput*, 67(2):528–564, sep 2013.

[14] Nvidia. *CUDA C Programming Guide*, 6.5 edition, August 2014. Design Guide.

[15] J. J. Quirk and S. Karni. On the dynamics of a shock-bubble interaction. *Journal of Fluid Mechanics*, 318:129–163, jul 1996.

[16] D. Rossinelli, B. Hejazialhosseini, P. Hadjidoukas, C. Bekas, A. Curioni, A. Bertsch, S. Futral, S. J. Schmidt, N. A. Adams, and P. Koumoutsakos. 11

PFLOP/s simulations of cloud cavitation collapse. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 3:1–3:13, New York, NY, USA, 2013. ACM.

[17] D. P. Schmidt and M. Corradini. The internal flow of diesel fuel injector nozzles: a review. *International Journal of Engine Research*, 2(1):1–22, 2001.

[18] E. F. Toro, M. Spruce, and W. Speares. Restoration of the contact surface in the HLL-Riemann solver. *Shock Waves*, 4(1):25–34, 1994.

[19] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[20] J. Williamson. Low-storage Runge-Kutta schemes. *Journal of Computational Physics*, 35(1):48–56, 1980.

[21] P. Zaspel and M. Griebel. Solving incompressible two-phase flows on multi-GPU clusters. *Computers & Fluids*, 80:356–364, jul 2013.