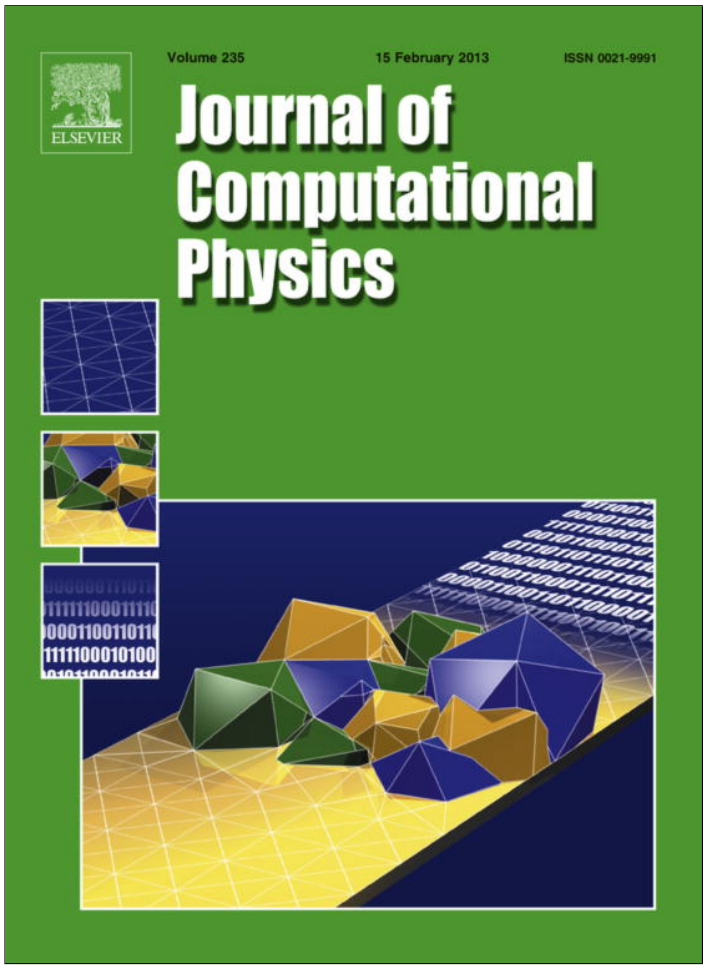
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

Journal of Computational Physics 235 (2013) 241–257

Contents lists available at SciVerse ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Direct numerical simulation of turbulence using GPU accelerated supercomputers

Ali Khajeh-Saeed, J. Blair Perot *

Theoretical and Computational Fluid Dynamics Laboratory, University of Massachusetts, Amherst, MA 01003, USA

ARTICLE INFO

Article history:
Received 25 November 2011
Received in revised form 28 September 2012
Accepted 26 October 2012
Available online 22 November 2012

Keywords:
Direct numerical simulation
Multi-GPU
Turbulence

ABSTRACT

Direct numerical simulations of turbulence are optimized for up to 192 graphics processors. The results from two large GPU clusters are compared to the performance of corresponding CPU clusters. A number of important algorithm changes are necessary to access the full computational power of graphics processors and these adaptations are discussed. It is shown that the handling of subdomain communication becomes even more critical when using GPU based supercomputers. The potential for overlap of MPI communication with GPU computation is analyzed and then optimized. Detailed timings reveal that the internal calculations are now so efficient that the operations related to MPI communication are the primary scaling bottleneck at all but the very largest problem sizes that can fit on the hardware. This work gives a glimpse of the CFD performance issues will dominate many hardware platform in the near future.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

The direct numerical simulation (DNS) of turbulence is a computationally intensive scientific problem that can benefit significantly from improvements in computational hardware performance. Graphics processors (GPUs) are special purpose hardware that are designed for interactive gaming, not for large scientific computations. Nevertheless, GPUs are reasonably well suited for many tasks that appear in scientific computing, and it is of some interest to examine their potential impact on computationally intensive algorithms such as the simulation of turbulent flow.

The fundamental advantage of GPUs over classic CPUs comes from the entirely different design of the memory subsystem. GPUs are designed to display millions of pixels at a rate of at least 60 times per second. The hardware is therefore directly designed to stream large sets of data in, do a few calculations on the data, and then stream the data out (often to the screen). To do this, GPUs use fast memory (DDR5) whereas most CPUs still use DDR3. They also use many duplicate channels to memory (16 at a time) rather than the typical two channels for a quad core CPU. And finally these processors do not use cache approaches to try and hide the memory latency as CPUs do. Caches are designed to speed up email, operating system tasks, and word processing. But caches are not typically useful for enhancing game performance or scientific computations. The long memory streams encountered in graphics applications (or PDE solvers) defeat caches.

The memory streams in a direct numerical solution of turbulence, such as a pressure or velocity field, are on the order of 1 billion bytes each. We would like to efficiently stream this data in, do a few computations and return the same field but at the next time level. Because processor speeds have been increasing over the last decade but memory speeds have not, all CFD simulations (and in fact almost all PDE solution techniques) are memory bound. This means, that on modern computers the computations are not important to the performance of the algorithm. The critical factor is the ability to read data in, and

* Corresponding author.
E-mail address: perot@ecs.umass.edu (J. Blair Perot).

0021-9991/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jcp.2012.10.050

Author's personal copy

write results out. GPU memory subsystems do this an order of magnitude more quickly than CPU memory subsystems based on caches. Perhaps not surprisingly, the GPU memory subsystem functions (and can be optimized by the user) remarkably similarly to the memory subsystem of the original Cray supercomputer vector processors.

Early examples of CFD calculations on GPUs are discussed in [1–3]. However, the GPU architectures have been evolving rapidly since that time in many different areas [4–7]. More recent CFD implementations that involve the CUDA programming paradigm are discussed in Elsen et al. [8] and Corrigan et al. [9]. Micikevicius [10] applied 3D finite difference computation using CUDA on 4 GPUs and achieved linear speedup for up to four GPUs. Rossinelli et al. [11] describe a 2D simulation using a vortex particle method on the GPU that achieves a speedup of 25. Jacobsen et al. [12] discretized the Navier–Stokes equations on a uniform Cartesian staggered grid with a second-order central difference scheme and achieved a 11× speedup with one GPU compared with two quad-core CPUs on the same node. They also achieved a 130× speedup for 128 GPUs compared to two quad-core CPUs (8 cores). In this work, a speedup of 20× is found for 192 GPUs vs. 192 CPU cores. The large reduction in the calculation time when using GPUs makes the MPI communication time, and methods for hiding this communication time, paramount in the following discussion.

2. Implementation

2.1. Hardware

Results for two different NSF GPU supercomputers are discussed in this work. Forge has replaced Lincoln at NCSA. It contains 288 Tesla M2070 NVIDIA Fermi GPUs that each have 6 GB DDR5 memory. Forge's 36 servers each hold two AMD Opteron Magny-Cours 6136 with 2.4 GHz eight-core CPUs (16 CPU cores per node) with 3 GB of RAM per core. Each server is also connected to 8 Tesla processors via PCI-e Gen2 X16 slots. The Forge results were compiled using Red Hat Enterprise Linux 6 (Linux 2.6.32) and the GNU compiler [13]. Another GPU cluster, Keeneland Initial Delivery (KID) resides at the National Institute for Computational Science (NICS) and will become an NSF XSEDE resource in late 2012. KID has 360 Tesla M2070 NVIDIA Fermi GPUs that each have 6 GB DDR5 memory. KID's 120 servers each hold two 6-core Intel Xeon (Westmere-EP) 2.93 GHz (11.72 GFlops) and 2 GB of DDR3 RAM per CPU core. Each server is connected to 3 Tesla processors via PCI-e Gen2 X16 slots. Keeneland nodes are connected by an x8 InfiniBand QDR (single rail) network [14]. In both Forge and Keeneland, error-correcting code (ECC) is on and the memory bandwidth of the GPU's is therefore reduced by roughly 12%.

The GPU performance of these machines will be compared with calculations using the CPUs on those machines. In addition, we will also perform tests on Orion, which is an in-house GPU machine containing 4 NVIDIA 295 cards (8 GPUs). On Orion, each GPU has 0.9 GB of memory and a theoretical bandwidth of 112 GBs. The connection between the GPUs on Orion uses an 8× PCI-e Gen2 connection (4 GB/s) and for simplicity, the communication still uses the MPI protocol even though this is a shared CPU memory machine. Also we replaced the first and second GPUs with GTX 480 and Tesla C2070 cards in order to run some cases with these newer GPUs.

The low-level CFD algorithm structure is dictated by two key features of the GPU hardware. First, the GPUs read/write memory is an order of magnitude faster when the memory is read linearly with stride 1. Random reads/writes or long strides are comparatively slow on a GPU. In addition, each multi-processor on the GPU has some very fast on-chip memory (shared memory) which serves essentially as an addressable program-supervised cache. CFD, like most three-dimensional PDE solution applications, requires considerable random memory accesses (even when using structured meshes) for sparse matrix–vector multiplies. Roughly 90% of these slow random memory accesses can be eliminated by: (1) linearly reading large chunks of data into the shared-memory space, which is fast for all accesses, (2) operating on the data in the shared-memory, and then (3) writing the processed data back to the main GPU memory (global memory) linearly. This optimization is the key to obtaining the roughly 20x speedup of the GPU over a CPU.

2.2. Software

The solution method uses a three-step, low-storage Runge–Kutta scheme [15] for time advancement that is second-order accurate in time. This scheme is stable for eigenvalues on the imaginary axis less than 2, which implies $CFL < 2$ for advective stability. The simulations always use a maximum $CFL < 1$. The diffusive terms are advanced with the trapezoidal method for each Runge–Kutta substep, and the pressure is solved using a classical fully-discrete fractional step method [16], although an exact fractional step method [17] is also possible. The solution of the elliptic pressure Poisson equation dominates the solution time (90%), so further details of this portion of the code are presented in Section 3, and the timing results in Section 4 also focus on this part of the algorithm. The Poisson solution in this work uses a preconditioned conjugate gradient method. This Poisson solver allows arbitrary boundary conditions (and physically relevant flow initialization) to be computed. FFT methods for the solution of the pressure, which are common in turbulence simulations, are restricted to fully periodic domains or extremely simple wall geometries.

For the spatial discretization, a second order Cartesian staggered-mesh scheme is used. This not only conserves mass and momentum to machine precision, but because it is a type of discrete calculus method [18] it also conserves vorticity (or circulation) and kinetic energy in the absence of viscosity. As a result, there is no artificial viscosity/diffusion in this method except that induced by the time-stepping scheme [19]. In addition, the staggered mesh discretization is free from pressure

Author's personal copy

modes and the need for pressure stabilization terms. This discretization method also treats the wall boundary condition well because the wall normal velocity unknown lies exactly on the wall, so no interpolation is required to enforce the kinematic no penetration condition. Higher order versions of this method exist but are more complicated to parallelize [20].

Many of the simulations presented below were performed on 512^3 meshes with fully periodic boundary conditions on the exterior of the computational domain, and wall boundary conditions on interior embedded objects. These internal "mixing cubes" (see Fig. 1) are the reason an FFT based Poisson solver is not possible. They are used to generate the turbulence physically (by driving the flow past them) rather than imposing some false structure on the turbulence via an *ad hoc* initial condition or forcing scheme.

The highest Reynolds numbers simulated in this work are comparable to the Reynolds numbers found in laboratory wind tunnel turbulence experiments (such as Comte-Bellot and Corrsin [21]). In addition, the highest Reynolds numbers tested in this work are sufficient to show decay rates that are very consistent with high Reynolds number decay theories [22]. The simulations of turbulence are initialized by driving fluid past the stationary "mixing boxes" with a mean pressure gradient that varies slowly in time. This initialization procedure has the advantage of allowing the initial turbulence spectrum to develop naturally rather than being imposed as an initial condition. Further details of the simulation initial conditions can be found in Ref. [22].

Physical units can sometimes be helpful to put the simulations in perspective. If the simulated fluid is water at standard temperature and pressure (with $\nu = 10^{-6}$ m/s^2) then the total domain size is a cube that is 48 cm on a side. The small cubes that initialize the turbulence are 1.4 cm on a side. In the 512^3 simulations there are 768 initialization cubes randomly placed in the domain. The total volume of all the stirring elements is therefore 1.92% of the total simulation volume. The mesh size itself is 0.9375 mm (which is 1/15th of the stirring cube size). At early times in the simulation, the timestep can be as small as 1/1000th of a second. In all the simulations, the timestep is never larger than a 1/10th of a second.

2.3. Partitioning

The large cuboid computational domain is divided into smaller cuboid subdomains (Fig. 2(a)). Each subdomain is allocated to one GPU, and each GPU communicates via MPI with its six local neighbors (Fig. 2(b)). A few calculations, such as a sum( ), require very small amounts of data to be globally communicated between all the GPUs. For example, for a 512^3 simulation running on 64 GPUs, each GPU solves a 128^3 subdomain problem (2 million mesh points), but it communicates only the data at the surface of that subdomain with its six neighbors. So $6 \times 128^2 = 0.1$ million data items (or about 5% of the data) is communicated from/to each GPU. In addition, most of the communication instructions (MPI) can be overlapped with regular computations, so communication is 'hidden' and does not take any extra time to perform. This is an important optimization because even the fastest MPI interconnect systems (such as the Infiniband used on these supercomputers) is very slow compared to the GPU's memory access rates.

Although the boundary data is typically small (<10%) compared to the internal data on the partitions, the MPI communication operations are also typically very slow (>10× slower) than the internal operations. It is therefore quite possible for the boundary operations, and not the large number of bulk internal operations, to dominate the solution time and dictate the scaling behavior of the code. When the boundary operations dominate, the code speeds up by roughly a factor of

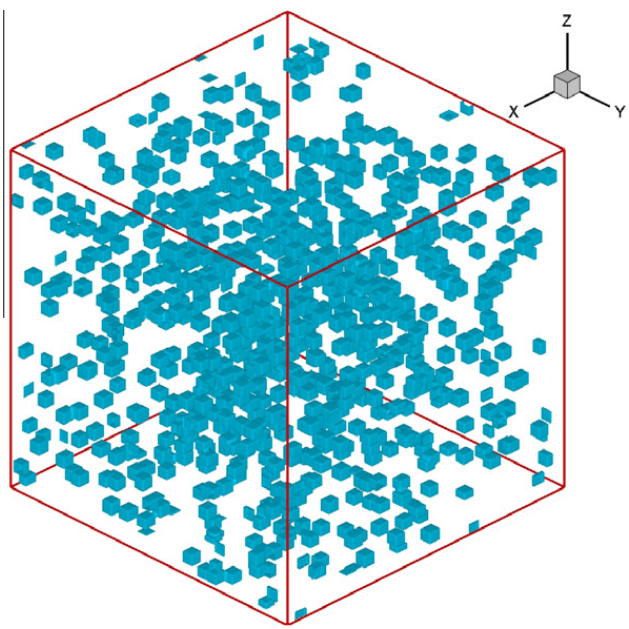


Fig. 1. Simulation domain with 768 randomly distributed cubes.

Author's personal copy

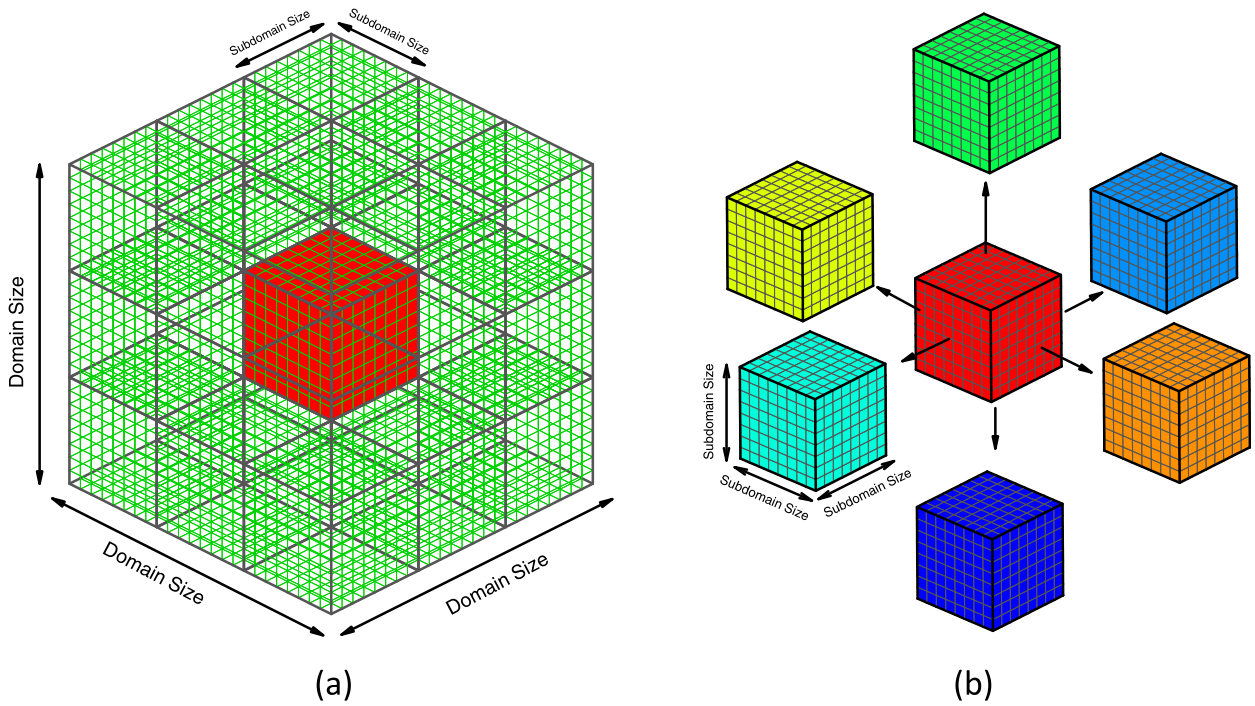


Fig. 2. (a) Domain and (b) subdomains with boundary planes for MPI communication.

$2^{2/3} = 1.59$ when the number of processors is doubled (rather than producing the expected doubling of the speed). This issue is highlighten by the GPU architecture, but will become a factor in the near future even for classical CPU-based supercomputers.

For efficient GPU solution, the subdomains (one subdomain per GPU) must be further partitioned into smaller 'chunks' (what NVIDIA calls blocks). On the GPU, each multi-processor will handle one or more chunks of data. In the current implementation each chunk is 16 mesh points in x (or a multiple of 16), any dimension in the y-direction (though typically we chose a multiple of 16 in this work), and the full subdomain extent in the z-direction. For efficient processing it is best to have at least two chunks per multiprocessor. The GPU multiprocessors can hide memory latencies by having two chunks active at the same time. The trade-off of having two active chunks per multiprocessor is that each chunk can only use half as much shared memory, registers, etc. The size of 16 in the x-direction is dictated by the fact that the GPU multiprocessors each have 32 SIMD cores that read data much more quickly ('coalesced memory access' in NVIDIA terminology) if they can perform the read in two sets of 8, linearly, and with unit stride. If the mesh size is not originally a multiple of 16 in the x-direction, the code pads the data with dummy cells so that it is a multiple of 16. Because the chunks in our implementation tend to be longer in the z-direction than in the other two directions, the chunks in this implementation are similar to fat 'data pencils'.

Almost all CFD algorithms, and most PDE solvers, are strongly memory bound. That means the code performance scales with the number of memory reads and writes, not with the number of mathematical operations. Math operations are now an order of magnitude faster than memory operations and therefore are close to instantaneous (compared to memory ops). The memory speeds are therefore critical to the efficiency and scaling of the code. The key issue with multi-GPU computing is that there are a minimum of three widely differing memory speeds. On a GPU, a chunk of data typically is read into and operated on using very fast shared-memory (which is effectively a cache). The amount of shared-memory is fixed by the hardware and is 8 KB on the Tesla S1070 or 24 KB per chunk when running 2 chunks per multiprocessor on the Fermi GPUs. Because it is not stride one, a chunk's boundary data must be accessed via an order of magnitude slower memory reads to the GPU's main memory. In addition, the subdomain boundary data must be transferred and accessed via MPI calls that are yet an order of magnitude even slower yet. Although our CFD implementation aggressively minimizes the number of these slower memory accesses, these two separate boundary communications still impact the code performance on everything but the very largest problem sizes.

2.4. Optimizations

Memory allocation and deallocation is expensive on the GPU (and the CPU) because they require calls to the operating system. To avoid this, the code sets up its own temporary arrays at start up. These can be held and dropped very quickly by any subroutine. They are only deallocated when the code completes.

Author's personal copy

Grid information such as Δx, Δy, Δz are stored in a special GPU memory space called constant memory. Constant memory is cached and therefore fast for repeatedly called items, such as the geometry parameters. Constant memory is a hold-over from older GPU designs, that is difficult to use, and has fairly limited utility. We also looked at using texture memory (a memory type designed for real graphics applications) and could find no circumstance where texture memory was beneficial.

Data transfers between the CPU and the GPU are relatively slow and expensive (5–10 GB/s vs. 150 GB/s on the GPU itself). Almost all data for the CFD calculation therefore resides on the GPU and stays on the GPU. Only when data is needed (for MPI, or to write data to files) is data copied to the CPU.

2.5. Isotropic, homogeneous decay

Turbulence is generated in these simulations rather than set from an arbitrary initial condition. The generation takes place by 'shaking' the simulation domain (simulating it in a non-inertial reference frame). This shaking drives fluid past small cubes that are randomly placed in the domain. The direction of this acceleration is random, but its magnitude is fixed. In these simulations the direction of the acceleration is changed to a new random direction (with the same magnitude) every 0.3 s [22] and 17 accelerations are performed in total (5 s). The primary acceleration is then turned off from 5 s to 7 s and during this time the domain is returned to its start position. After this 2 s the restoring acceleration causes the mean flow to be extremely close to zero. Also during this 2 s time period the turbulence changes from being accelerated to being in isotropic decay. At the end of this period (when the mean flow is zero), the boxes instantaneously turn into (zero velocity) fluid [22]. From 7 s to 12 s the flow adjusts to become pure isotropic decay. After 12 s the turbulent flow is in isotropic decay [22]. Fig. 3(a) shows the turbulent kinetic energy (TKE) compared with data from an FFT-based simulation by de Bruyn Kops et al. [23]. Fig. 3(b) shows the decay exponent and large-eddy length scale and Re number (divided by 100) as a function of time. This figure shows very good agreement with theoretical predictions (of $n = 1.2$ at high Reynolds numbers) up until a time of 30 s (or roughly six large eddy turnover times), at which time the length scale becomes constrained by the simulation domain size. Constrained decay (with a constant large eddy length scale) has a theoretical decay rate of $n = 2$ which the simulation approaches at very long times. Contour plots of the x component of the velocity at six different times are shown in Fig. 4. These plots show both the decay of the energy and the growth of the turbulent length scales. Most wind tunnel experiments have a test section with effectively 1–2 s of resolution, so these computations cover a comparatively long time frame. In the last two pictures the large eddies are large enough to become domain constrained.

2.6. Isotropic, homogeneous decay with plane strain

In this simulation, the same the procedure is used to generate the turbulence up to 12 s. However, the domain is a rectangular cube with dimensions of $29 \times 79 \times 48$ cm. After 12 s, uniform plane strain is imposed on the turbulence (in X and Y directions) up to 20 s. In this simulation the domain deforms with the mean flow (allowing the simulation to continue to use periodic boundary conditions). So by 20 s the domain has become a perfect cube. From 20 s up to 100 s the turbulence is allowed to decay with no imposed strain. Fig. 5 shows the TKE, ε, decay exponent, large-eddy length-scale and Re number for this simulation. After 60 s, when the large-eddy length exceeds roughly one-quarter of the domain size, the turbulence becomes domain constrained and statistically very sensitive to the exact configuration of the largest eddies. The value of $n = 1.5$ is the low Re theoretical value. Strain appears to trigger this low Re decay mode. Contour plots of the x component

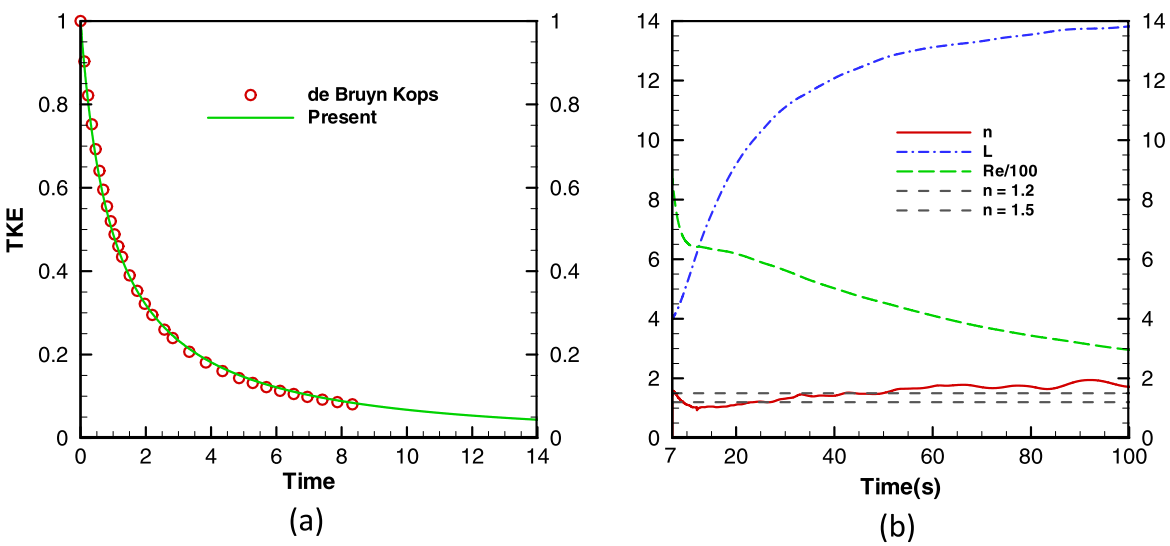


Fig. 3. (a) Turbulent kinetic energy and (b) decay exponent, large-eddy length scale, and Re number (divided by 100) for isotropic homogenous decay run on the Keeneland supercomputer using 64 GPUs.

Author's personal copy

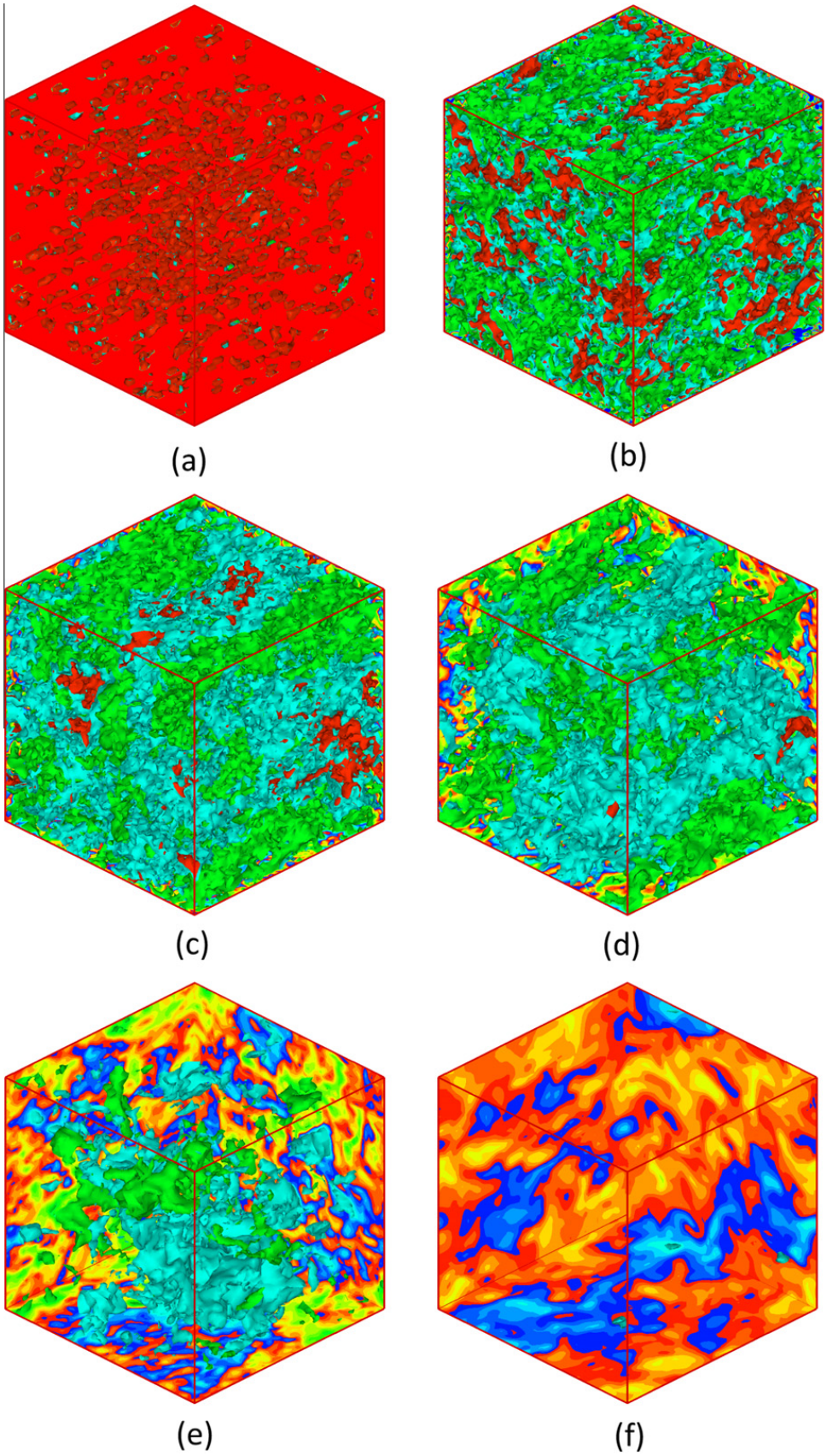


Fig. 4. Snapshots of 512^3 turbulence field showing u iso-surfaces in isotropic homogenous decay at: (a) 5 s (b) 7 s (c) 12 s (d) 20 s (e) 40 s and (f) 110 s running on the 64 GPUs on Keeneland.

Author's personal copy

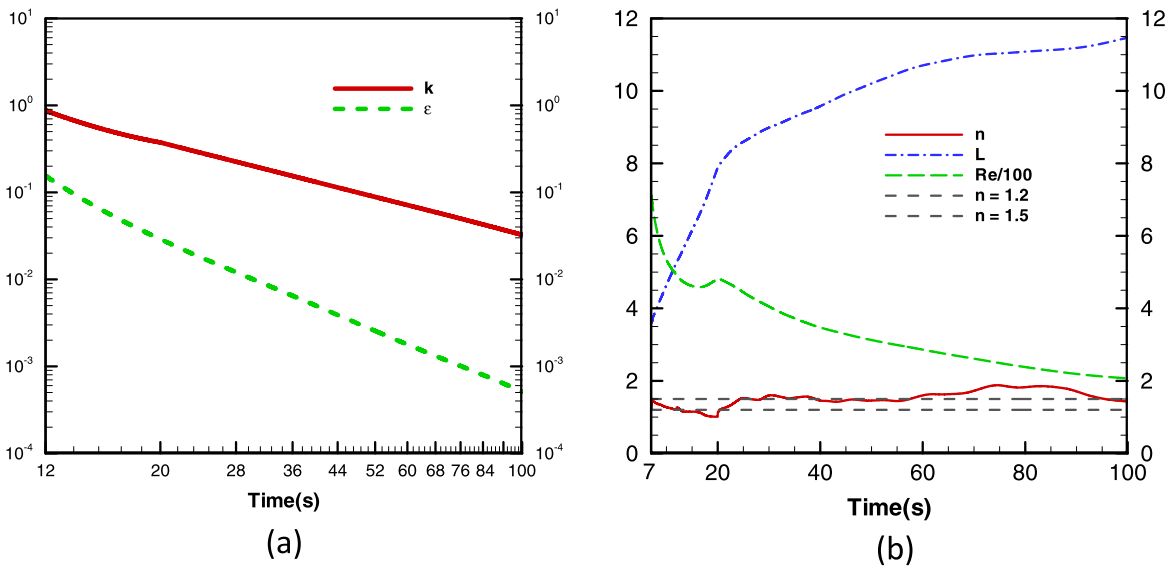


Fig. 5. (a) TKE and ε and (b) decay exponent and large-eddy length scale and Re number (divided by 100) for isotropic homogenous decay with plane strain 1 case running on Keeneland supercomputer using 64 GPUs.

of the velocity at six different times are shown in Fig. 6. These plots show both the decay of the energy, the growth of the turbulent length scales, and the change in the domain shape due to the plane strain.

The code is written so that the low level operations come in either a CPU or GPU version. The choice is made at compile time, and the code is then optimized by the compiler for that given hardware. In theory, the code could operate on virtually any hardware platform by rewriting only these core lowest-level routines to accommodate the new hardware.

3. Algorithm details

3.1. GPU implementation

The code solves the pressure Poisson equation using a polynomial preconditioned conjugate gradient (CG) iterative method. The conjugate gradient method is an efficient iterative method and is guaranteed to converge for a symmetric positive-definite matrix. The CG method is composed of one matrix multiply, one preconditioner matrix multiply, 2 scalar (dot) products, and 3 AXPY (Alpha *X* Plus *Y*) operations. The three AXPY parts are easily mapped to the GPU architecture. However, the most computationally intensive part of the solution procedure is the matrix multiply which computes the Cartesian-mesh discrete Laplacian. In the current implementation the preconditioner has the same sparsity pattern as the Laplacian matrix and is therefore implemented in exactly the same way as the Laplacian. To compute the Laplacian matrix for a particular cell, all neighboring cells and the central cell are needed (7 cells in 3D). This is more difficult than the simple AXPY to map to the GPU.

When performed naively, the 7 point matrix stencil reads each data item 7 times from the main GPU memory. Only 3 of the points are linear, stride one, and therefore fast, the others are large stride memory accesses and in terms of speed are essentially random memory operations. The code is made more efficient by using a modified version of Micikevicius [10] implementation. This involves reading the data once into the shared memory on each GPU multiprocessor, and then accessing it from the fast memory location seven times. To do this each multiprocessor keeps three *XY* planes of data (from the tall-in-*Z* data chunk) in its memory (Fig. 7). The middle *XY* plane contains five of the stencil points (in the *X* and *Y* directions) saved in shared memory, and the upper and lower *XY* planes contain the 6th and 7th stencil values (just above and below the middle *XY* plane) saved in a register. After the discrete Laplacian is computed for the middle plane, the middle (shared memory) and upper (register memory) planes are copied to the lower (register memory) and middle (shared memory) planes respectively. The top plane reads in the new data from the main (global) GPU memory to the register memory.

Note, however, that in order to compute a 16×16 Laplacian stencil result, a 18×18 data input is actually required (minus the four corners). This is read in as a 16×18 block (with stride 1 fast access), and two 1×16 strips for the two sides. These last two strips have a stride equal to the subdomains size in the *x*-direction, and therefore are much slower to read. It therefore takes roughly the same amount of time to read the two 1×16 strips as it does the rest of the data (18×16). This is the first example of where the internal data is processed so efficiently that the unusual operations (two boundary strips in this case) take just as much time as the far more numerous (but much more efficient) common operations.

The other option would be to read 16×16 blocks efficiently (no side strips) but only compute a 14×14 region of the stencil. Because the eight SIMD cores on the GPU multiprocessor compute 16 items at a time, this means that the code would have an instruction divergence. This is where some cores do an operation, and some others do something else. On these SIMD

Author's personal copy

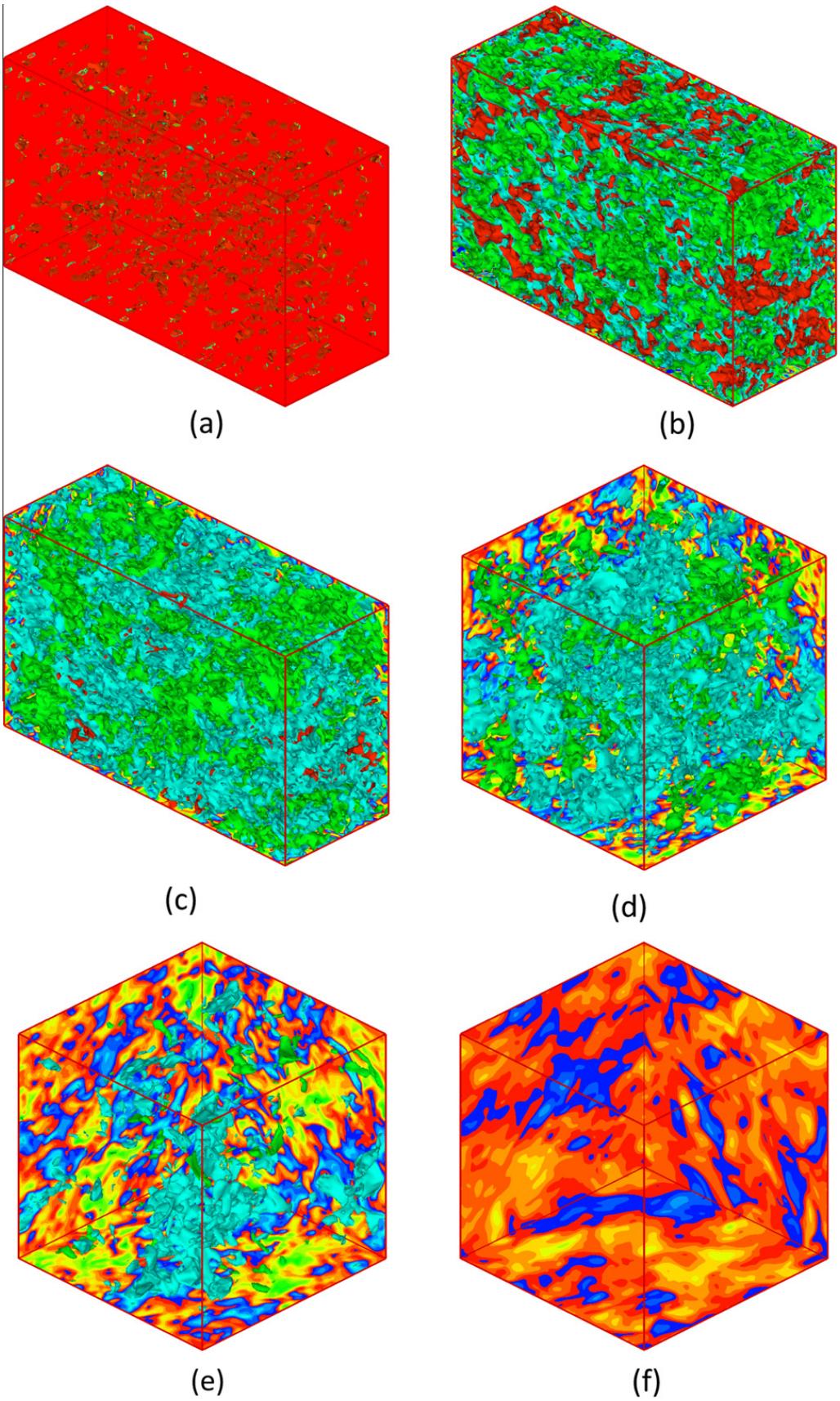


Fig. 6. Snapshots of 512^3 turbulence field showing u iso-surfaces for the plane strain case at: (a) 5 s (b) 7 s (c) 12 s (d) 20 s (e) 40 s and (f) 110 s running on the 64 GPUs on Keeneland.

Author's personal copy

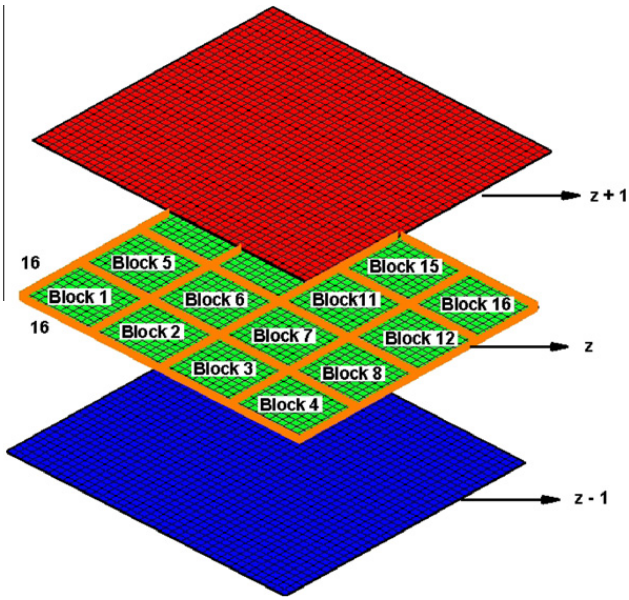


Fig. 7. Chunk distribution with in a GPU subdomain.

cores this results in slower execution (by about a factor of 2). In addition, this approach means the multiprocessors are reading blocks of 16 that overlap at the edges. This makes the 16×16 read slower. Therefore, the advantage of reading no boundary strips is actually lost. One way or the other, there is a price to be paid for the fact that the chunks of data being computed on each multiprocessor must use data from a different chunk. We have structured the algorithm so that the data transfer between chunks is an absolute minimum, (it is about 1/8th of the internal data on each chunk). Nevertheless, the slower times for boundary data between chunks means that this smaller amount of data still has a significant impact on the total solution time.

The second major optimization in the CG algorithm is to perform the two dot products at the same time as the matrix multiply and the preconditioner matrix multiply (one dot-product for each). Both arrays for the dot-product are already in fast shared memory when performing the matrix multiply, so this saves reading the two arrays for each dot-product (four array reads in total). The dot-products are therefore essentially free of any time impact on the code, except that their final result must be summed among all the GPUs. This requires an MPI all-to-all communication that cannot be hidden by any useful computations (but the amount of data communicated is very small, one word per GPU). We recognize that restructuring of the CG algorithm can be performed in order to overlap dot-product summations with computation, however this also leads to a CG algorithm with more storage and more memory read/writes. So the speed improvement of a modified CG algorithm is not expected to be significant.

With 512^3 meshes, naive summation (for turbulence averages) can lead to round-off errors that are on the order of 10^9 times the machine precision (for single precision this would mean an order 1 error). Even though we use double precision in all the computations, summation is still performed in stages to reduce the round-off error. The 3D array is first collapsed into a 2D array using the GPU, by summing along the *Z*-direction. Further reduction is then performed by reducing in the *Y*-direction, and then the *X* direction on the CPU, and then by summing the results from all the GPUs using MPI all-to-all communication (four stages in total). This procedure only looses roughly two decimal places of accuracy during the summation, and allows the expensive portion of the computation (the first reduction to *XY* planes) to be performed on the fast GPUs.

3.2. Multi-GPU implementation

The approach to parallelism when using many GPUs together is quite different from the type of parallelism used within each GPU. The key aspect of the inter-GPU algorithm is the relatively long communication times (using MPI) between GPU subdomains. These long times are due to GPU-to-CPU copy times and CPU-CPU MPI communication times. For a transfer, all data must be copied from the GPU to the CPU over the PCI express bus. Only then can the CPU core use MPI (or CPU-threads) to communicate the data. The MPI (or thread) communication then occurs at CPU memory speeds (which is slower than GPU main memory speeds). Orion (our in house machine), uses MPI on shared memory which is as fast as MPI can theoretically function (and is about 8% slower than using threads directly). Still, MPI copy times are significant on that machine. After the MPI calls, data must be copied back to the GPU.

To hide the copy time and the slow MPI communication times, data is prefetched and overlapped with GPU computations as much as possible. There are two possible GPU memory types that can be used when hiding the communication. The first possibility is regular pinned (or page-locked) memory and the second possibility is using mapped or write-combined

Author's personal copy

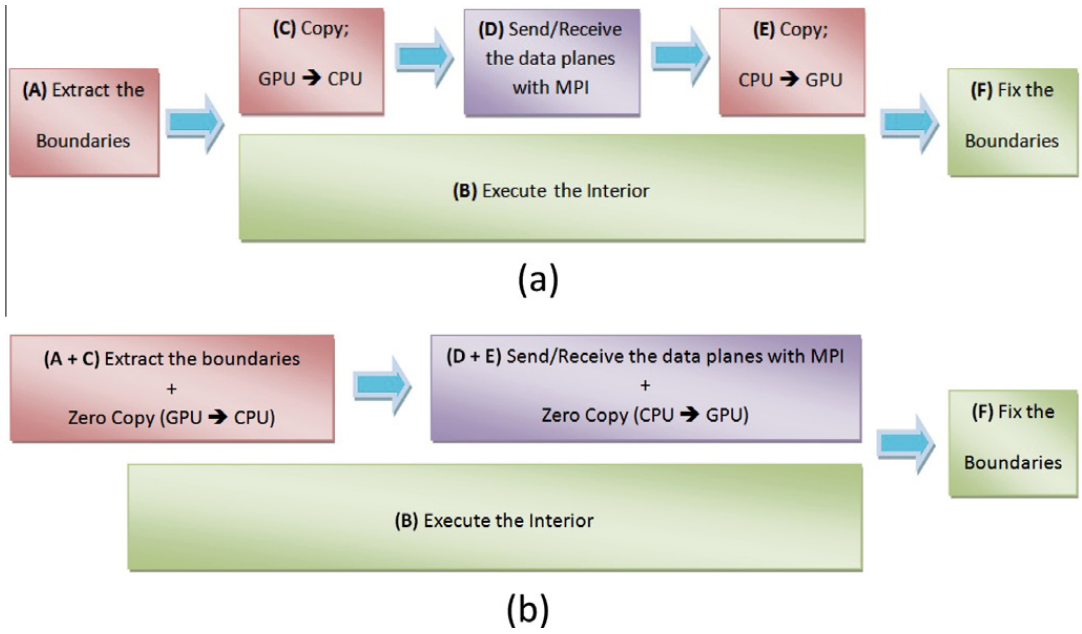


Fig. 8. Typical flow chart for subdomain processing with (a) regular pinned memory and (b) mapped and write-combined memories for MPI buffers. Red indicates boundary operations (running on stream 2); green indicates the primary bulk operation (running on stream 1) and purple (running on the CPU) indicates MPI send/receive instructions.

memory (zero copy memory) for the MPI buffers. The basic structure of a subroutine with regular pinned memory is shown in Fig. 8(a).

(A) On the GPU, load the six boundary planes of the subdomain data (which resides in the GPU main memory) into six smaller (and stride 1) arrays. This requires the GPU and cannot be well overlapped with GPU computations.
(B) On the GPU, start the internal calculation. This step is the primary action of the subroutine.
(C) Copy the small boundary arrays from step (A) to the CPU. This can overlap with part (B).
(D) When part (C) is finished Send/Receive the data planes using MPI. The CPU handles all MPI operations and is otherwise idle, so this can also overlap with part (B) which executes on the GPU.
(E) Copy the received data from the CPU back to the GPU. Again, this still can overlap with part (B).
(F) When both part (B) and part (E) are finished, apply the boundary data to the calculation.

In the second approach, because mapped memory is used for MPI buffers there is no need to explicitly copy data from the GPU to CPU or vice versa (though this still happens implicitly). In this approach, when part (A) is done, MPI can start to send data. In addition, after MPI receives a data message that data can be immediately read by the GPU without explicitly copying that data from CPU the GPU. The write-combined memory is most efficient when the CPU writes to that memory and GPU just reads from that memory. So we just used write-combined memory for receiving buffers. Fig. 8(b) shows the basic structure of a subroutine mapped and write-combined memory is used.

In the latest GPU architecture (Fermi), it is possible to run up to 16 kernels at the same time. So in theory part A and B can be executed at the same time if there are available resources in the GPU. In practice, for CFD calculations the code rarely goes faster when doing this. If the internal calculation (part B) takes long enough it can hide the communication occurring in parts C, D and E. Part F is the portion of the subdomain boundary calculation that cannot be hidden. Typically the final boundary operation (Part F) involves large stride memory writes and it can therefore never be optimized as well as the internal bulk calculations (part B). For smaller subdomain sizes (32^3 per subdomain and less) parts (A) and (F) can take much longer than part B (the actual bulk calculation).

4. Results

4.1. Strong scaling

Figs. 9 and 10 show the speedup (vs. the same number of CPU cores) and millions of cell updates per second per GPU or CPU core (MCUPS/Processor) for strong scaling results on Forge and Keeneland supercomputers. MCUPS represents how many millions of finite-volume cells can be updated (one CG iteration) during a second of wall-clock time. In the strong scaling situation, the problem size is constant, and as the number of processors is increased, the problem size per processor gets smaller and smaller.

Author's personal copy

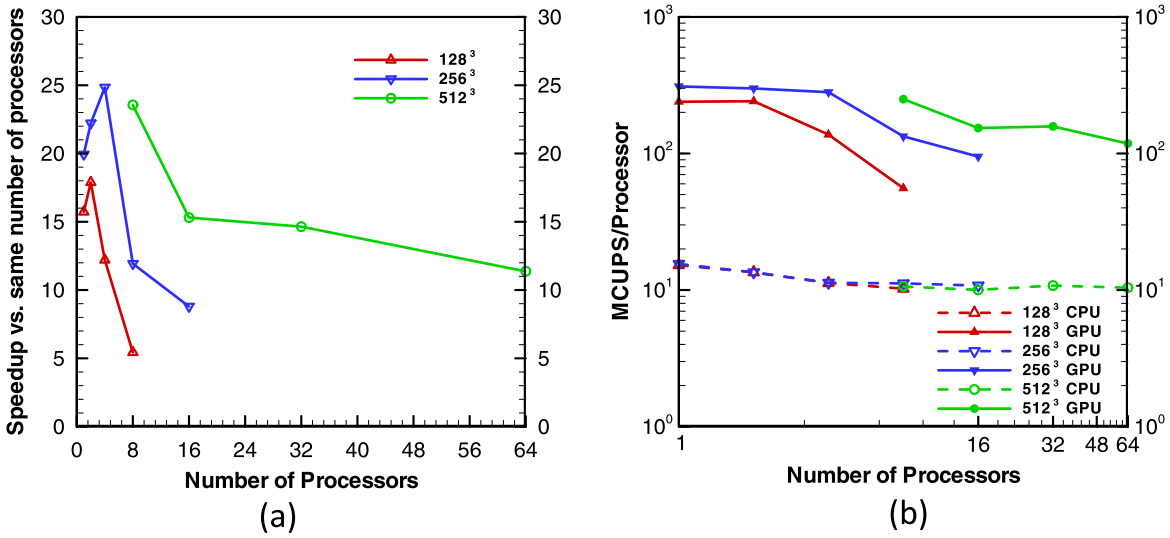


Fig. 9. (a) Speedup (CPU time/GPU time) and (b) performance per processor for strong scaling of the 128^3, 256^3 and 512^3 CFD problem on Forge supercomputer using GPUs and CPUs.

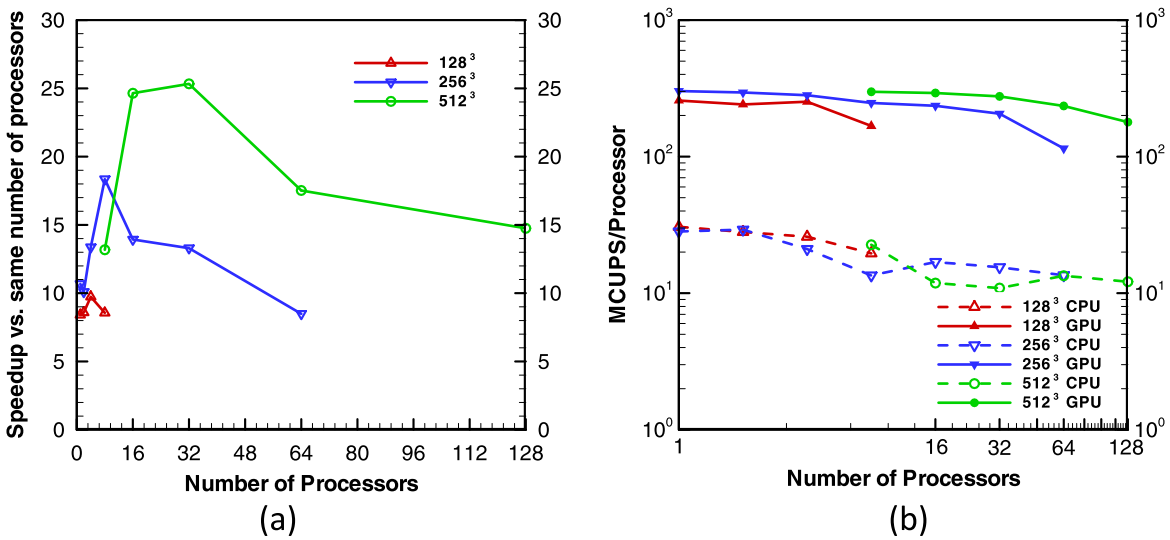


Fig. 10. (a) Speedup and (b) performance per processor for strong scaling of the 128^3, 256^3 and 512^3 CFD problem Keeneland using GPUs and CPUs.

For strong scaling, the highest speedup (about 25×) occurred for 4–8 GPUs compared to 4–8 CPU cores (Forge's CPU) for the Forge supercomputer and 32 GPUs compared to 32 cores (Keeneland's CPU). In theory, the MCUPS/Processor is directly related to the hardware efficiency and should be a constant horizontal line. The CPU efficiency decreases from 1 core to 2 and 4 as the cores contend for the limited available memory bandwidth on a single node. The GPU efficiency decreases for large numbers of processors when the amount of work per GPU drops and overlapping the computation with MPI communication becomes difficult. Note that this difficulty hiding the MPI communication does not occur on the CPU cores (yet) because for these problem sizes the CPU computations take much longer to complete (rough 15× longer) and can hide the equally slow MPI communication.

The performance of the CPU is related to the number of memory pipelines. Unfortunately, in most CPUs, the number of memory pipelines is not equal to the number of cores. Therefore, in memory bound problems, the CPU shows good performance only up to the number of memory pipelines. After that there is not much performance benefit for adding more cores (only cache benefit). When running on the relatively slow CPU all MPI communications can be hidden by useful computations, so increasing the number of nodes leads to better performance and an almost constant (horizontal line) for the CPUs. But for the GPUs, the PCI-e speed (between the CPU and GPU) plays an important role. On Forge, as the number of GPUs increases from 4 to 8 GPUs the PCI-e speed is reduced to x8 from x16. The PCI-e speed for Keeneland is always ×16 so with an increasing number of processors the only reason for losing performance is that the problem size per GPU is getting smaller eventually making it difficult to hide the communication time of the boundary data.

Author's personal copy

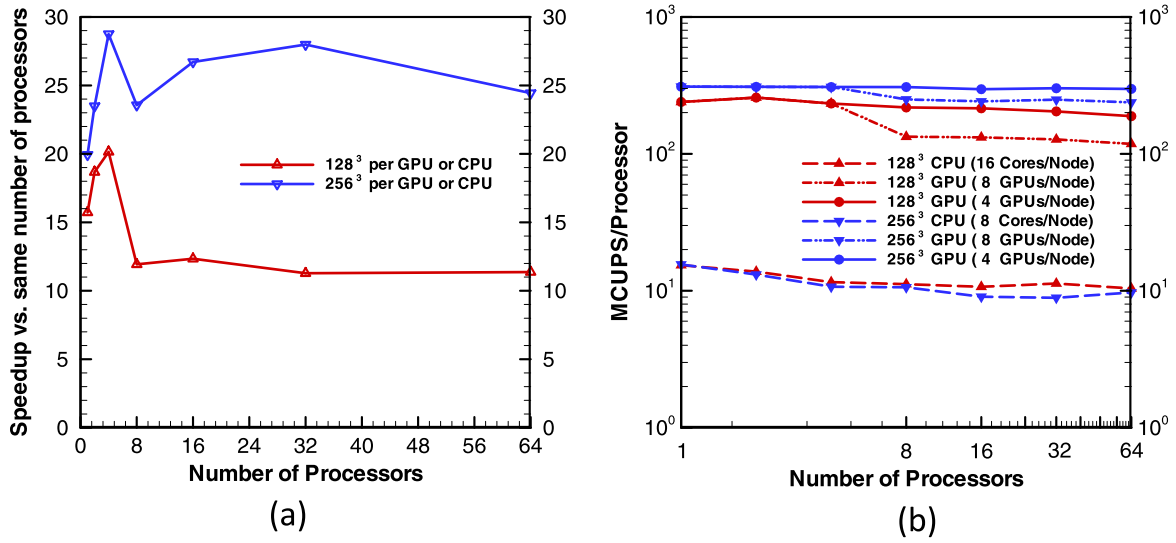


Fig. 11. (a) Speedup and (b) Performance per processor for weak scaling of the 128^3 and 256^3 subdomain sizes for the CFD problem on Forge using GPUs and CPUs.

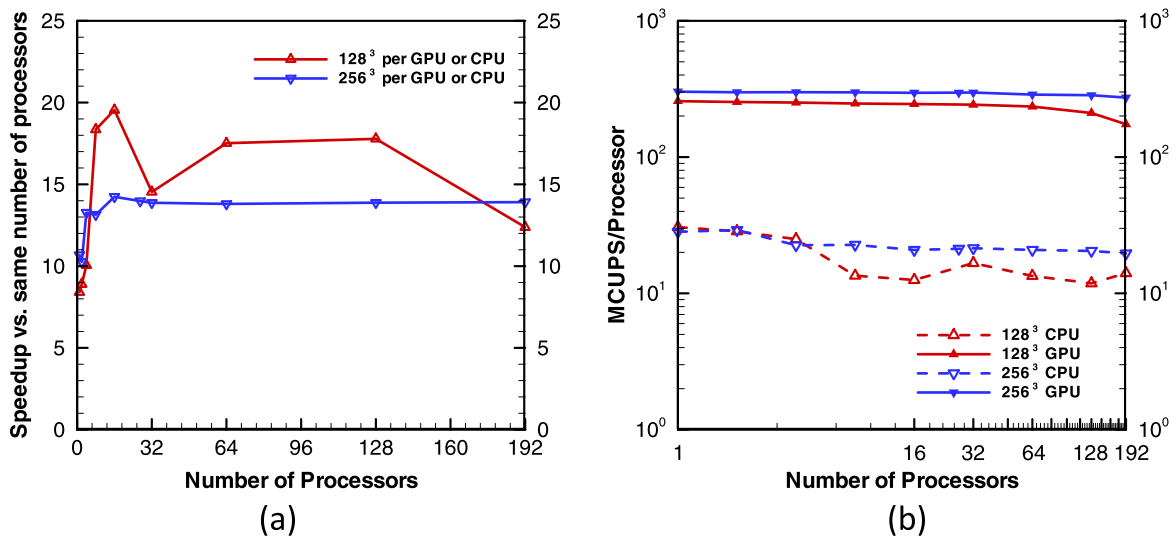


Fig. 12. (a) Speedup and (b) Performance per processor for weak scaling of the 128^3 and 256^3 subdomain sizes for the CFD problem on Keeneland using GPUs and CPUs.

4.2. Weak scaling

Figs. 11 and 12 show the speedup (vs. the same number of CPU cores) and millions of cell updates per second per GPU or CPU core (MCUPS/Processor) for weak scaling results on Forge and Keeneland supercomputers. In the weak scaling situation, the problem size per processor is constant. In general, weak scaling shows if communication times are affecting the solution time.

Fig. 11(b) shows the effect of PCI-e speed on the Forge supercomputer. And Fig. 12(b) shows that with a 256^3 subdomain size, the code performs well on Keeneland for up to 192 GPUs.

4.3. Efficiency of Forge and Keeneland supercomputers

Fig. 13 shows the performance of the code compared to a single processor (single GPU for the GPU cases and a single core for the CPU results). The efficiency drop for both Forge (4 GPUs/node) and Keeneland supercomputers is 4% for 64 GPUs. On Keeneland the efficiency drops by 10% when 192 GPUs are used for the 256^3 subdomain size. On the CPUs, efficiency is lost as soon as the number of cores exceeds the number of memory channels.

Author's personal copy

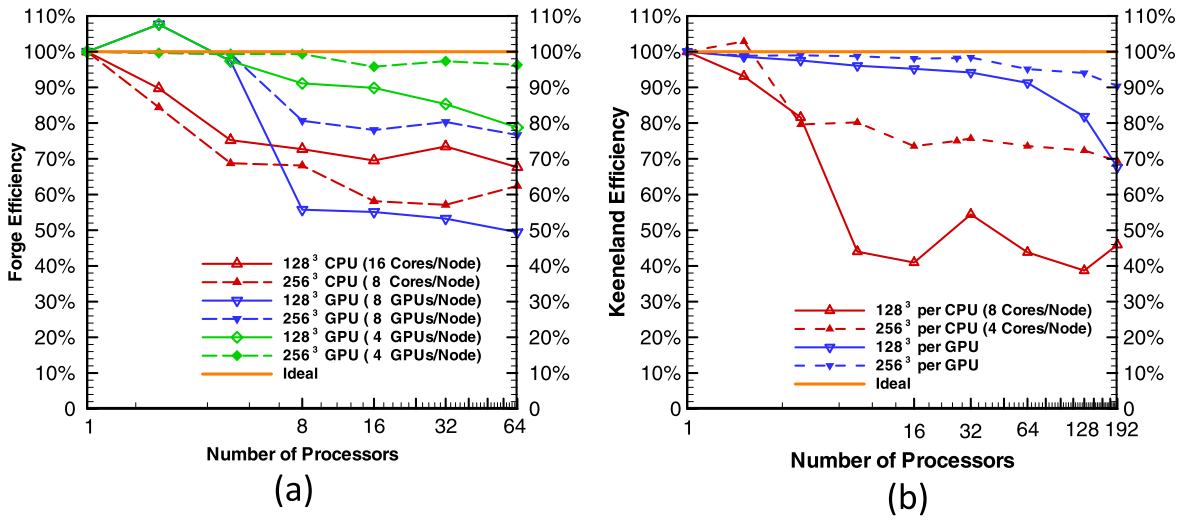


Fig. 13. (a) Forge and (b) Keeneland efficiency for the weak scaling of the 128^3 and 256^3 subdomain sizes for the CFD problem using GPUs and CPUs.

4.4. Poisson solution

In this section, the performance of the code on 1 GPU on Orion (our in-house GPU cluster) is more closely analyzed. Timings indicate that 87% of the code execution time is spent in the conjugate gradient (CG) solver (which solves for the pressure and implicit viscous diffusion terms) and fully 50% of the total time is spent in the sparse matrix multiply subroutines. A breakdown of the time spent in the CG and Laplace algorithms is shown in Fig. 14. Laplace_Inv is the CG preconditioner.

The summation item in Fig. 14(b) includes copying the dot product results from GPU to CPU plus the last steps of summation on the CPU. This figure shows that the interior calculation (Part B) is the most time consuming part in the Laplace solver for large problem sizes. The copying time plus MPI for the largest problem sizes is 4 times smaller than the interior time. So on Orion, subdomain problem sizes of 128^3 per GPU and larger are sufficient to hide the MPI and copying time. On Forge when 8 GPUs per node are used, this is not quite the same, because the copy time is $2\times$ slower (dashed black line). Also when a single node is used, MPI is not using the Infiniband card for communications. So when the number of GPUs exceeds the number of GPUs on a node, MPI times increase significantly. Therefore, in order to use many GPUs efficiently, the subdomain size per GPU should be at least 128^3. Current GPUs do not have enough memory to handle problem sizes greater than 288^3 per GPU. This means that the GPUs can operate efficiently only on the very largest problem sizes that the hardware can handle. For smaller problem sizes the communication time (via MPI) is larger than all the internal calculations.

Because every GPU subdomain has to send data on its boundaries to 6 other subdomains, it needs to copy six boundary surfaces to the CPU. So part (A), (C), (D) and (E) scale like N^2. But part (B), solving the interior points, grows like N^3. Fig. 14(b) shows the extrapolated time for copying between the CPU and GPU plus the MPI time on Orion with a single GPU and PCI-e

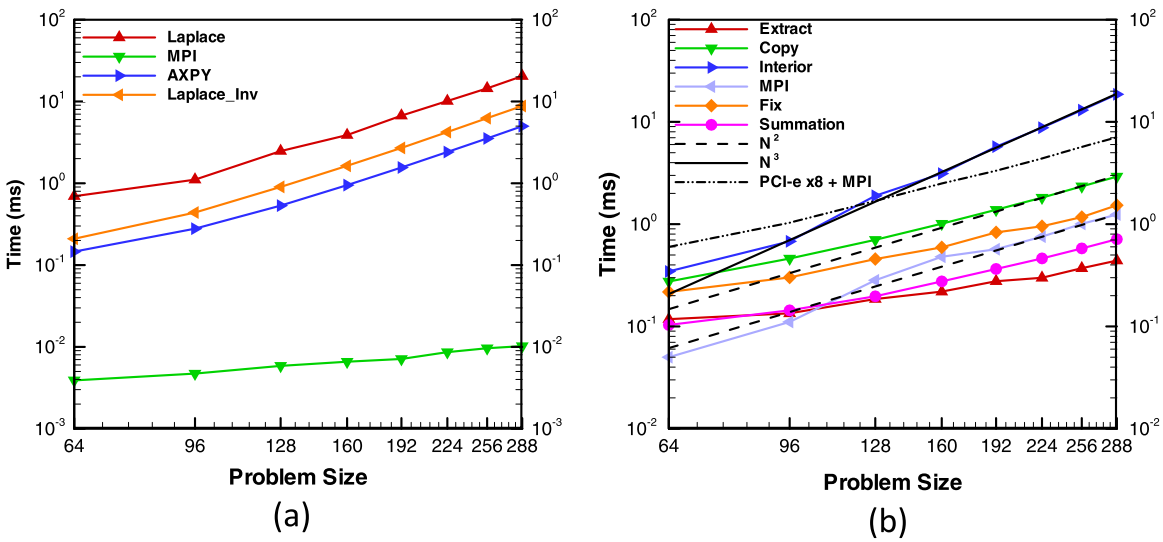


Fig. 14. Time for (a) CG and (b) Laplace subroutines for different problem sizes using the Tesla S1070 on Orion.

Author's personal copy

x8. As you can see even for 128^3 subdomain sizes the copying times are barely overlapped with the interior domain computation.

We should also note that in all Orion results, ECC (error correcting memory) was off, but on Forge and Keeneland ECC is on and therefore the GPU memory speeds are slower. When ECC is on, the Tesla C2070's raw bandwidth is reduced at least by 10%. Fig. 15 shows the speedup of single Tesla C2070 with ECC on and off, compared to the single core of an AMD CPU. Fig. 15(b) shows that when ECC is on, the performance of the code is reduced by 30% on average.

The NVIDIA Parallel Nsight tool [24] was used to analyze the Laplace matrix multiply routine in detail on a Tesla C2070 GPU installed on Orion with ECC memory error checking turned off. Fig. 16 shows the timeline for the Laplace matrix-multiply subroutine with two different communication hiding approaches on a problem size of 64^3.

This problem size is not sufficient to completely hide the communication time. Fig. 16(a) shows that the actual MPI time overlaps with the computation, but the copy back to the GPU does not overlap well. With this first approach it takes a long time for the boundary fix (part F) code to load onto the GPU and start executing (the bright green is idle GPU time). This case is an example of how the GPU can be strongly affected by the long times associated with GPU synchronization. Synchronization takes on the order of 200 μs to execute, which is the same order of magnitude as the interior execution, which is the primary point of the subroutine.

With mapped and write-combined memories for the send and receive buffers, as shown in Fig. 16(b), the parts (A) and (C), and parts (D) and (E) are combined into one single transaction. When mapped and write-combined memories are used for send and receive buffers, the time for (A) plus (C) is smaller than when regular pinned memory is used for these parts. However, part (F) is faster in regular memory than when using write-combined memory. Reading from regular pinned memory is faster than write-combined memory. Also Fig. 16(a) shows that part (B) in regular memory is faster than the pinned memory case. The main reason for the deference in times is that the mapped memory parts (A), (C) can copy concurrently with the GPU as it executes computations (Part (B)). So some of the GPU resources are used to execute part (A). When write-combined memory is used for receive buffers there is no need for cuStreamSynchornize after the MPI calls. The removal of synchronization improves overall performance.

The total run time for 64^3 when pinned memory is used is 704 μs and the total execution time when using the mapped memory is 467 μs. For the relatively small problem (64^3 per GPU) mapped memory is almost 50% faster than pinned memory. The primary reason for this difference is removal of the second cuStreamSynchornize.

Fig. 17 shows the timeline for the Laplace matrix-multiply with for both approaches for a problem size of 128^3. In this case, MPI and copying times are four times longer and the interior calculation time is eight times longer than for the 64^3 case shown in Fig. 16. Although this result is just for a single GPU the code still uses MPI and message sending to implement the periodic boundary conditions. This MPI communication is fast (since it is on the same CPU), but sufficient overlap time exists in this calculation to tolerate longer MPI send times.

Fig. 17(b) shows that parts (A) and (C) are running fully concurrently with the actual calculation in part (B). This feature is available only for the Fermi architecture. However, part B now takes longer, so this overlap is largely superficial and does not lead to reduced run times. The total run time for the 128^3 case when pinned memory is used is 1990 μs and it is 2332 μs when mapped memory is used. This is the opposite of the 64^3 case. Now the pinned memory is almost 17% faster than mapped memory. The main reason for this difference is that writing and reading from mapped memory is more expensive than reading and writing from regular pinned memory. So parts (A + C) and (F) in mapped and write-combined memories

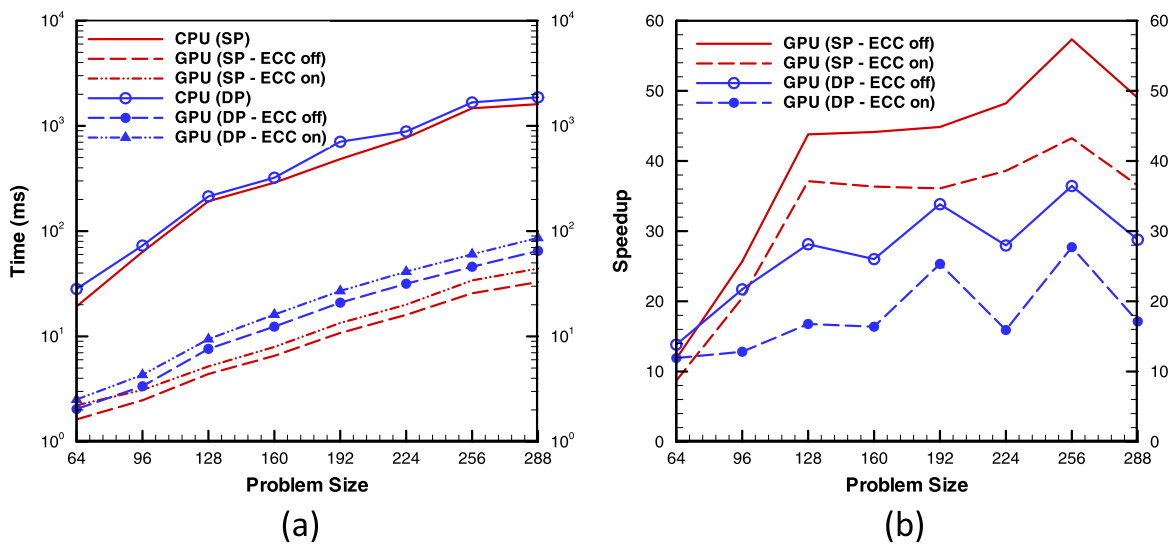


Fig. 15. (a) Performance and (b) speedup for different problem sizes using Tesla S1070 on Orion with ECC on/off options. Both single precision (SP) and double precision (DP) calculations are analyzed.

Author's personal copy

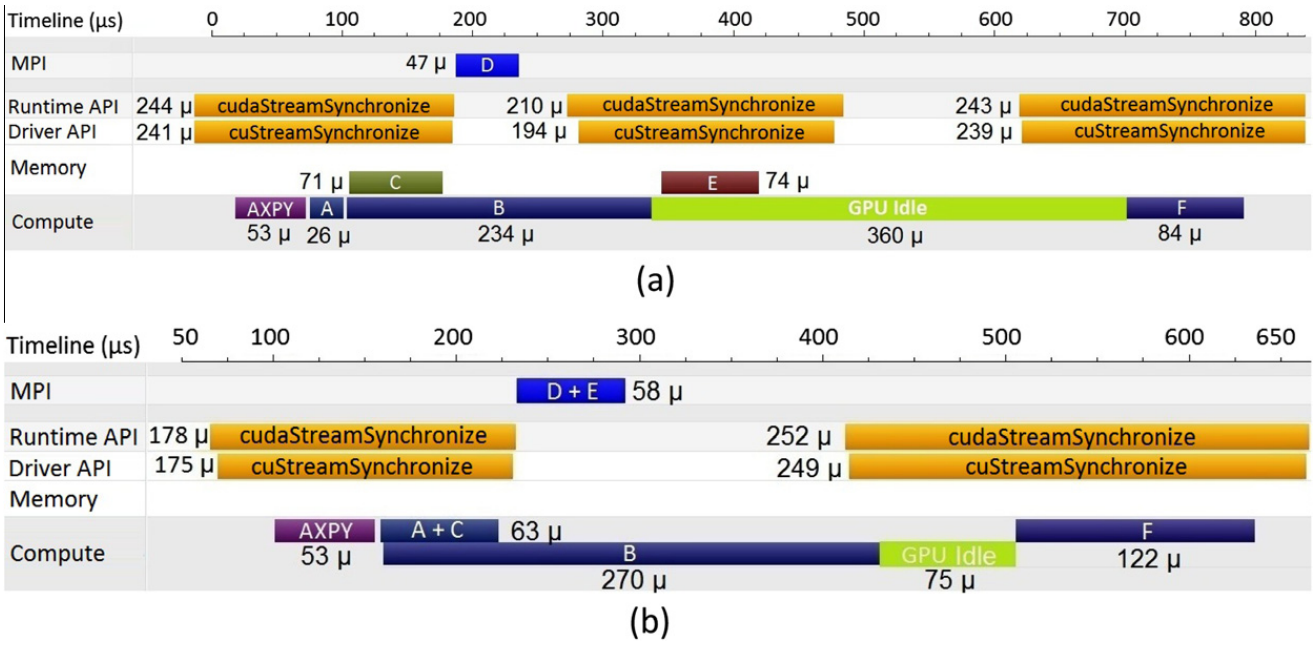


Fig. 16. Timeline for Laplace kernel for 64^3 (with total time) for (a) regular pinned memory (704 µs) and (b) mapped and write-combined memories (467 µs) using Tesla C2070 on Orion (ECC is off).

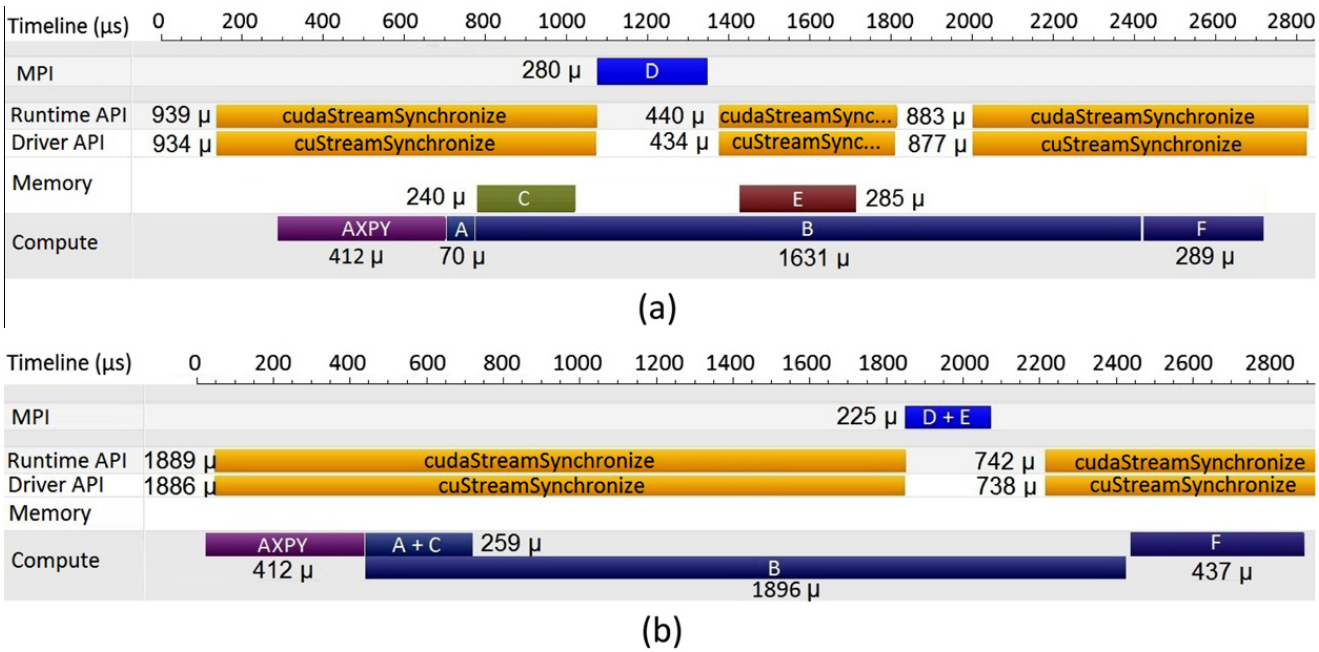


Fig. 17. Timeline for Laplace kernel for 128^3 (with total time) for (a) regular pinned memory (1990 µs) and (b) mapped and write-combined memories (2332 µs) using Tesla C2070 on Orion (ECC is off).

cases take much more time than part (A) and (F) in regular pinned memory (the part (C) in pinned memory is completely hidden with the part (B)).

Fig. 18 shows the timeline for the Laplace matrix-multiply with for both approaches for a problem size of 256^3. In this case, the interior calculation is now four times larger than MPI plus copying times so there is plenty of spare time for slower MPI communication times over a large network. There is now an almost 8500 µs safety margin (the gap between the last two *cuStreamSynchornize* in Fig. 18(a)) for MPI communications. Fig. 18(b) shows that parts (A) and (C) are running partly concurrently with part (B) for the 256^3 case. The reason for this is that parts (A) and (C) are requesting more resources than are now available in the GPU device. So only some section of parts (A) and (C) can run concurrently on the GPU. The total run time for the 256^3 case when pinned memory is used is 12,403 µs, and for the mapped and write-combined memory it is 14,043 µs. As with the 128^3 case, the pinned memory is almost 13% faster than mapped and write-combined memory.

Author's personal copy

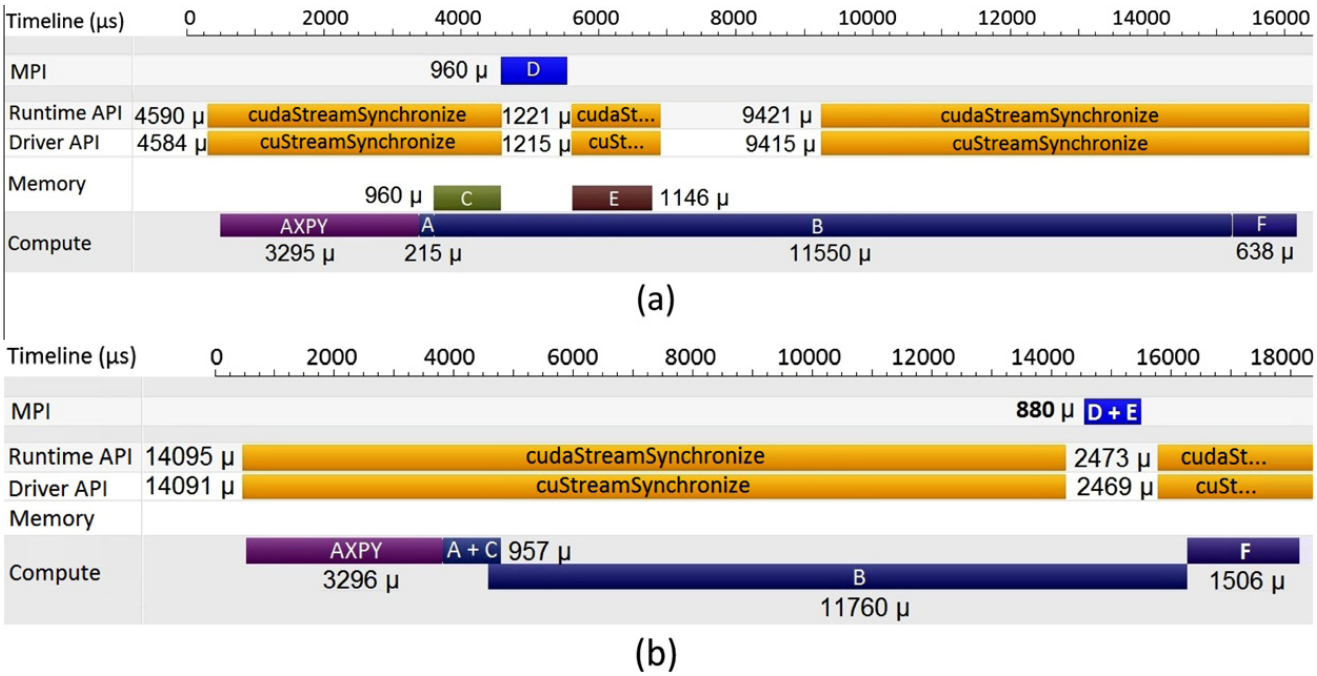


Fig. 18. Timeline for Laplace kernel for 256^3 (with total time) for (a) regular pinned memory (12,403 μs) and (b) mapped and write-combined memories (14,043 μs) using Tesla C2070 on Orion (ECC is off).

Due to the hardware setup on Forge (the GPUs share the PCI-e bus), it actually takes 2× longer to copy than on Orion. This means that on Forge the 128^3 simulation takes longer to copy boundary data off the GPU (part C and E) than it takes to compute (part B). The MPI and copy back barely overlap.

The larger simulations show good overlapping of the communication with computation. On Orion, the 128^3 case even has some leeway for expanded MPI communication times. Note that for the 128^3 case, the boundary computing time (part A and F) is 22% of the bulk time (reducing the effective performance of the code by roughly the same amount). In an ideal calculation, this boundary processing would be as fast as the bulk and therefore take only 5.4% of the bulk time.

The 256^3 case shows relatively small communication times compared to the computations. This large problem size involves 67 million unknowns and requires 3.5 GB of memory for double precision calculations. The Tesla M2070 has 6 GB of memory so this is more than half of the GPU memory. The largest problem size that can be computed is 288^3 per GPU (5 GB, the GPU reserves some memory for its actual job of providing video). The maximum memory available for less expensive GPUs is 3 GB.

5. Discussion

It was determined that GPUs can significantly enhance the speed of CFD calculations (by roughly a factor of 20×). However, the large speed increases in the primary part of the computation, now bring the lesser (and slower) parts of the calculation directly to the forefront. Hiding communication times and boundary data calculations become difficult for anything but the largest problem sizes that the hardware can support.

The mantra in computer engineering is to make the common case fast. GPUs follow this philosophy to great effect. However, at some point the common operations become so fast that the uncommon operations dictate the overall performance. In our CFD calculations on GPUs, as little as 5% of the data can control the code performance (on anything but the largest problem sizes). The GPU now performs common calculations so fast that other factors (large stride memory accesses and GPU–GPU communication) become the bottleneck even though those operations are relatively uncommon.

This work indicates that GPU supercomputers will enable larger CFD calculations to be performed, but they will not be as helpful at allowing existing CFD simulations to simply run faster. A 128^3 simulation can be run on either a single GPU or all 16 CPU cores of a shared memory machine in roughly the same amount of time. While adding 7 GPUs to this same motherboard is possible (see Orion) it will not allow this same problem to be computed 8 times faster. The additional 7 GPUs only allow a 256^3 simulation to be performed (in the same amount of time as before).

It was determined that GPU synchronization calls can have a profound effect on the GPU performance. This is particularly true for smaller subdomain sizes like 64^3 per GPU. With only $0.25 \cdot 10^6$ operations, the GPU executes code faster than it can synchronize. The command *cudaThreadSynchronize* sometimes takes 200 μs and can stop the CPU from loading the next executable kernel to the GPU. The command *cudaStreamSynchronize* is similar. It makes the CPU and GPU stall if the specified

Author's personal copy

stream is still running. However, if the stream has been already finished, *cudaStreamSynchronize* does not take much time. This code uses implicit synchronization (no explicit commands) as much as is possible.

The ECC memory has a significant effect on the performance of the Fermi GPUs. It slows them down by about 30% so that the Fermi GPUs work almost the same as the GT200 series (previous generation). Also the amount available memory is reduced by 12.5%.

In general, if there is a possibility of hiding the copying time with a kernel execution it is efficient to use regular pinned (page-locked) memory instead of mapped (or write-combined) memory. Mapped memory is useful on the GPU only if it is not possible to hide the copying time with kernel execution. This occurs when finding the summation or maximum value of a large field of variables.

The GPU has a fairly narrow operating range in terms of the number of unknowns per subdomain that the GPU can process. With less than 2 M mesh points CFD calculations do not have enough internal work to hide communication times. And with more than 14 M mesh points, standard GPUs run out of memory. The Tesla GPUs can go another $2\times$ larger (up to about 25 M mesh points).

Acknowledgements

This work used the NSF Teragrid/XSede supercomputers, Forge, located at NCSA, and Keeneland, located at NICS. The project was supported primarily by the Dept. of Defense via a subcontract from the Oak Ridge National Laboratory (Stephen Poole), and in part by the National Science Foundation.

References

[1] M.J. Harris, Fast fluid dynamics simulation on the GPU, in: GPU Gems, Addison Wesley, 2004, pp. 637–665 (Chapter 38).
[2] T.R. Hagen, K. Lie, J.R. Natvig, Solving the Euler equations on graphics processing units, in: International Conference on Computational Science, University of Reading, UK, 2006.
[3] S. Menon, J.B. Perot, Implementation of an efficient conjugate gradient algorithm for Poisson solutions on graphics processors, in: Proceedings of the 2007 Meeting of the Canadian CFD Society, Toronto Canada, June, 2007.
[4] A. Khajeh-Saeed, J. Blair Perot, in: Emerald (Ed.), GPU-Supercomputer Acceleration of Pattern Matching, Morgan Kaufman, 2011, pp. 185–198.
[5] A. Khajeh-Saeed, S. Poole, J.B. Perot, Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors, J. Comput. Phys. 229 (2010) 4247–4258.
[6] T. McGuiness, J.B. Perot, Parallel graph analysis and adaptive meshing using graphics processing units, in: 2010 Meeting of the Canadian CFD Society, London, Ontario, 2010.
[7] A. Khajeh-Saeed, J. Blair Perot, Computational fluid dynamics simulations using many graphics processors, Comput. Sci. Eng. 14 (3) (2012) 10–19.
[8] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, J. Comput. Phys. 227 (2008) 10148–10161.
[9] A. Corrigan, F. Camelli, R. Lohner, J. Wallin, Running unstructured grid based CFD solvers on modern graphics hardware, in: 19th AIAA Computational Fluid Dynamics, San Antonio, Texas, 2009.
[10] P. Micikevicius, 3D finite difference computation on GPUs using CUDA, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009.
[11] D. Rossinelli, M. Bergdorf, G. Cottet, P. Koumoutsakos, GPU accelerated simulations of bluff body flows using vortex particle methods, J. Comput. Phys. 229 (2010) 3316–3333.
[12] D.A. Jacobsen, J.C. Thibault, I. Senocak, An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters, in: 48th AIAA Aerospace Sciences, Orlando, Florida, January 2010.
[13] <http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/DellNVIDIACluster/TechSummary/index.html>.
[14] J.S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, S. Yalamanchili, Keeneland: bringing heterogeneous GPU computing to the computational science community, Comput. Sci. Eng. 13 (5), 90–95.
[15] J.B. Perot, J. Gadebusch, A stress transport equation model for simulating turbulence at any mesh resolution, Theor. Comput. Fluid Dyn. 23 (4) (2009) 271–286.
[16] J.B. Perot, An analysis of the fractional step method, J. Comput. Phys. 108 (1) (1993) 183–199.
[17] W. Chang, F. Giraldo, J.B. Perot, Analysis of an exact fractional step method, J. Comput. Phys. 179 (2002) 1–17.
[18] J.B. Perot, V. Subramanian, Discrete calculus methods for diffusion, J. Comput. Phys. 224 (1) (2007) 59–81.
[19] J.B. Perot, Conservation properties of unstructured staggered mesh schemes, J. Comput. Phys. 159 (1) (2000) 58–89.
[20] V. Subramanian, J.B. Perot, Higher-order mimetic methods for unstructured meshes, J. Comput. Phys. 219 (1) (2006) 68–85.
[21] G. Comte-Bellot, S. Corrsin, Simple Eulerian time correlation of full and narrow-band velocity signals in grid-generated isotropic turbulence, J. Fluid Mech. 48 (1971) 273–337.
[22] J.B. Perot, Determination of the decay exponent in mechanically stirred isotropic turbulence, AIP Adv. 1 (2011) 022104, http://dx.doi.org/10.1063/1.3582815.
[23] S.M. de Bruyn Kops, J.J. Riley, Direct numerical simulation of laboratory experiments in isotropic turbulence, Phys. Fluids 10 (9) (1998) 2125–2127.
[24] NVIDIA, Parallel Nsight 2.0 User Guide, 2011.