©Indian Academy of Sciences

A Fast MHD Code for Gravitationally Stratified Media using Graphical Processing Units: SMAUG

M. K. Griffiths[1,3,*], V. Fedun[2,3] & R. Erdélyi[3]

[1]*Corporate Information and Computing Services, The University of Sheffield, 10–12 Brunswick Street, Sheffield S10 2FN, UK.*
[2]*Department of Automatic Control and Systems Engineering, The University of Sheffield, Mappin Street, Sheffield S1 3JD, UK.*
[3]*Solar Physics and Space Plasma Research Centre (SP^{2}RC), School of Mathematics and Statistics, University of Sheffield, Hicks Building, Hounsfield Road, S7 3RH, UK.*
[*]*e-mail: m.griffiths@sheffield.ac.uk*

Abstract. Parallelization techniques have been exploited most successfully by the gaming/graphics industry with the adoption of graphical processing units (GPUs), possessing hundreds of processor cores. The opportunity has been recognized by the computational sciences and engineering communities, who have recently harnessed successfully the numerical performance of GPUs. For example, parallel magnetohydrodynamic (MHD) algorithms are important for numerical modelling of highly inhomogeneous solar, astrophysical and geophysical plasmas. Here, we describe the implementation of SMAUG, the Sheffield Magnetohydrodynamics Algorithm Using GPUs. SMAUG is a 1–3D MHD code capable of modelling magnetized and gravitationally stratified plasma.

The objective of this paper is to present the numerical methods and techniques used for porting the code to this novel and highly parallel compute architecture. The methods employed are justified by the performance benchmarks and validation results demonstrating that the code successfully simulates the physics for a range of test scenarios including a full 3D realistic model of wave propagation in the solar atmosphere.

Key words. Numerical simulations—magnetohydrodynamics—computer unified device architecture—graphical processing units—NVIDIA—Sheffield advanced code—the Sheffield magnetohydrodynamics algorithm using GPUs—versatile advection code.

1. Introduction

The Sheffield Advanced Code (SAC) described by Shelyag *et al.* (2008) is based on the Versatile Advection Code (VAC) (see Toth *et al.* 1998), a fully non-linear magnetohydrodynamic (MHD) code. With the increasing need to model complicated astrophysical processes with necessary resolution and higher magnetic Reynolds numbers, there is an urgent requirement for considerably larger amounts of storage

and CPU cycles. SAC was designed for simulations of linear and non-linear wave propagation in gravitationally strong stratified magnetized plasmas (Fedun *et al.* 2011a; Fedun *et al.* 2011b; Fedun *et al.* 2011c; Vigeesh *et al.* 2011; Scullion *et al.* 2011). Numerical modelling of waves in the complex and stratified solar atmosphere is undertaken by many researchers studying 2D and 3D MHD problems (see for example, Hasan *et al.* 2005; McLaughlin *et al.* 2009; Chmielewski *et al.* 2013; Chmielewski *et al.* 2014; Selwa *et al.* 2007; Wang *et al.* 2013).

For many years, such levels of compute cycles would have been restricted to dedicated supercomputing facilities. The advent of cluster computing led to the increased availability of high levels of compute cycles, enabling researchers to investigate more complex and challenging problems. The evolution of CPU design and systems architectures continue unimpeded. Currently, there are at least two key related developments, computer processors with multiple cores and general purpose Graphical Processing Units (GPUs).

For multimedia and gaming systems, it was found that a significant workload was used in the rendering of imagery (Lindholm *et al.* 2008). Given this scenario, developers recognized the need for a massively parallel architecture, capable of computing the pixel characteristics for high resolution display systems (Kirk & Hwu 2010). This resulted in the emergence of graphics cards with powerful graphical processing units for dedicated image rendering. These devices exploited multiple cores and massive parallelism to rapidly render complex scenes. It was not long before computational scientists exploited such architectures for their particular numerical problems. This advancement resulted in the Compute Unified Device Architecture (CUDA) (Cook 2012; Kirk 2010; Farber 2011). CUDA enables direct exploitation of GPUs without the need to use the high-level programming language required for image rendering and shader development. CUDA provides a means of developing applications using the C or Fortran programming languages and enables the realization of massively data parallel routines able to exploit the GPUs. Each GPU has its own memory and processing elements separate from the host computer.

A number of authors have reported success with the implementation of MHD algorithms on GPUs (Schive *et al.* 2011; Kestener & Fromang 2012; Pang *et al.* 2010; Lin *et al.* 2011; Wong *et al.* 2011). Many of these approaches use structured meshes, finite volume methods and TVD numerical schemes for solving the MHD equations. Also, these approaches employ different techniques for computing fluxes at cell boundaries, such methods are necessary for numerically stable shock capturing solvers. These solvers are applied to different problem domains, two examples are, studies of the solar wind or using general relativistic magnetohydrodynamics for investigations of ultra compact systems such as neutron stars and accretion disks around black holes (see Zink 2011; Wong *et al.* 2014a,b) reported speedups for double precison computations ranging from 50 to 5, when comparing performance against a single CPU core. Some of these algorithms have recently been extended by allowing problems to be solved using multi-GPU systems to overcome the limited memory of single GPU systems. These techniques have employed upto 216 GPUs to compute problems with upto 1200^3 grid points, additionally, these papers report results demonstrating the benefit of using GPU direct to provide more efficient inter-GPU communication.

The work reported here has enabled the Sheffield advanced code to run on massively parallel compute architectures such as general purpose graphical processing

units and multi-core processing systems. SMAUG (Sheffield Magnetohydrodynamics Algorithm Using GPUs) is a derivative of the code which was developed to run on CUDA-enabled GPUs. After a brief description of CUDA, we will describe the basic architecture of the SMAUG code and illustrate the performance benefits of running SAC on multi-core systems. Specifically, we show results for the NVIDIA C1060 (Tesla), M2070 (Fermi) and K20 (Kepler) GPU. We demonstrate that SMAUG is sufficiently verified and that it successfully validates the physics for a range of 1–3D test scenarios, including a full 3D model of wave propagation in the magnetized and stratified solar atmosphere. The novelty of SMAUG compared to many GPU enabled MHD codes is its implementation of the hyperdiffusion technique for controlling numerical diffusion arising in highly shocked systems.

2. GPU programming using CUDA

Recognizing the take up of graphical processing units for numerical computation, NVIDIA introduced the Tesla GPU architecture and the compute unified device architecture for general purpose parallel computing. CUDA-enabled graphics processors operate as co-processors within the host computer. In the CUDA-based programming model an application consists of sections of code known as kernels. These kernels may be executed in parallel as a large number of threads on a GPU. To achieve optimal levels of performance, it is understood that GPU application developers must make use of platform specific languages, for example, CUDA. There have now arisen further options for developing applications which are able to exploit GPUs. These include the Fortran implementation of CUDA developed by the Portland Group, directive or accelerator-based programming or Fortran to CUDA code generators. Examples of directive or accelerator-based programming include OpenMP-style pragma-based programming, e.g., developed by PGI, HMPP, hiCUDA or the OpenAcc standard (for details of these methods see, Wolfe 2008; Han & Abdelrahman 2011; Wienke *et al.* 2012).

An additional development was OpenCL, a framework enabling developers to write applications which run across heterogeneous platforms consisting of CPUs, GPUs and other processors. One possibility is the use of the application known as SWAN (Harvey & De Fabritiis 2011), which may convert a CUDA code to OpenCL. It is envisaged that applications developed using OpenCL and CUDA will be able to efficiently exploit the hybrid environment comprising multiple GPUs and multi-core CPUs.

Options such as the PGI, HMPP and OpenAcc accelerator libraries are welcome because of the ease of developing the code. Wienke *et al.* (2012), provide an informative account of programming using the OpenAcc standard. When development of the SMAUG commenced, tests indicated that significant effort was required to implement and test the accelerator-based code. It was found that the CUDA code generator (Govett 2009) was not suitable for our purposes. Although it is highly automated, this approach would still require a large amount of hand-coding, due to the number of parallelisable sections of code. Based on this experience, it was decided that the option chosen for developing the GPU implementation of SAC, known as SMAUG, was to use the CUDA toolkit. The compute unified device architecture introduced by NVIDIA is not just a hardware platform but also includes software to support the

development of parallel applications running on NVIDIA GPUs. The software tools include compilers, profilers and numerical libraries. Our main motivation for adopting CUDA is due to the open source approach and the availability of support for CUDA programming. Although we are using a high level language, it is still possible to achieve maximum performance. There is also greater scope for code optimization.

The new code described in this paper was tested on available NVIDIA C1060 (Tesla), M2070 (Fermi) and K20 (Kepler) architectures. Characteristics for these GPUs are summarized in Table 1. Unlike CPUs, GPUs have a parallel throughput architecture enabling the execution of many concurrent threads, rather than executing a single thread very quickly. These threads execute on many processing cores which comprise the GPU. The CUDA programming model facilitates multi-threaded programming by introducing the concept of a kernel which runs as an individual thread. The kernels are sections of programming instructions which will be executed by every thread. The developer of a CUDA-based GPU application must, therefore, correctly identify those parts of an application which are potentially multi-threaded. Additionally, the developer must also ensure the application correctly allocates GPU memory and transfers data between the host and GPU memory. For the SMAUG application, we attempted to minimize the number of transfers between host and GPU memory by performing the entire computation within the GPU. Achieving a high level of double precision performance is important for many numerical applications, computational magnetohydrodynamics is certainly no exception. NVIDIA cards with at least compute capability version 1.3 are able to support double precision. To use double precision performance in computations, the NVIDIA compiler must be called with the flag '-arch=sm_13' (see Whitehead & Fit-florea 2011).

A graphics card supporting CUDA consists of an array of streaming multiprocessors, each of these supports a group of streaming processors. Threads are grouped into batches of 32, which are called warps. The threads execute independently. GPUs feature four types of memory: global memory, register memory, shared memory and texture memory. To perform useful computations, data must be transferred between the memory space of the host computer and CUDA device(s). For this reason, when assessing the performance of a GPU code, it is important to account for the time taken for the transfer of data between host memory and GPU memory. Data from the host CPU is transferred using a PCI-e bus with access to the GPU global memory. This transfer rate is at around 8 GB/s, this is significantly lower than the bandwidth for data transfer between the GPU memory and its streaming processors. The GPU global memory can be accessed by all of the streaming processors. The shared memory is faster, but is limited to 16 kB on each block. Shared memory is accessible by the streaming processors in a multi-processor. The register memory consists of 16384 32-bit registers on each block. The differences between the

Table 1. Characteristics for NVIDIA GPU's.

	Tesla C1060	Fermi M2070	Kepler K20
Number of cores	240	448	2688
Single precision peak performance (GFlops)	933	1030	4710
Double precision peak performance (GFlops)	78	515	1218
GPU memory (GB)	4	6	5
GPU bandwidth (GB/s)	102	144	208

C1060 (Tesla) and M2070 (Fermi) GPUs are quite striking (see Table 1), there is an increased number of cores and improved memory bandwidth within the GPU. One of the most interesting changes relates to the performance. There is a small increase in the single precision performance and an order of magnitude improvement in the double precision performance. This can be attributed to the memory bandwidth for data transfer between the global GPU memory and block memory. It should be understood that, rather than use the single instruction multiple data (SIMD) approach, CUDA employs a single instruction multiple thread approach (SIMT). This implies that the number of threads for computation is not fixed, and can influence the number of registers available per thread.

One of the main differences between the Kepler and Fermi architecture is that the Kepler streaming multiprocessor features 4 warp schedulers which can support upto 2048 threads. Kepler was designed to deliver more computational performance per unit of power, this has been achieved by reducing the clock frequency. The challenge for the programmer is to implement a code which can provide more parallelism with a sufficiently large number of instructions. A number of improvements enhance the ability of Kepler to schedule threads. Using the technology known as Hyper-Q, multiple CPU cores can utilize a Kepler GPU, the new grid management unit enables better prioritization of grid execution. Dynamic parallelism on the Kepler GPUs enables kernels to execute child kernels; this is in contrast to the Fermi architecture for which only the CPU can execute a kernel. The updated inter-GPU communication using GPUDirect has an RDMA feature which allows devices such as network interface cards to directly access GPU memory. This significantly reduces the latency of MPI messaging to and from GPU memory.

Before considering the implementation of the SMAUG application, we now recall the numerical solution method employed for the SAC code.

3. Equations and numerical methods

The Sheffield advanced code (Shelyag *et al.* 2008) is a numerical MHD solver that allows the simulation of the various physical processes in magnetized plasmas. The general system of MHD equations are

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\mathbf{v}\rho) = 0, \tag{1}$$

$$\frac{\partial (\rho \mathbf{v})}{\partial t} + \nabla \cdot (\mathbf{v}\rho\mathbf{v} - \mathbf{B}\,\mathbf{B}) + \nabla p_{\mathrm{t}} = \rho\mathbf{g}, \tag{2}$$

$$\frac{\partial e}{\partial t} + \nabla. \, (\mathbf{v}e - \mathbf{B}\,\mathbf{B}\cdot\mathbf{v} + \mathbf{v}p_{\mathrm{t}}) + \nabla p_{\mathrm{t}} = \rho\mathbf{g}\cdot\mathbf{v}, \tag{3}$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{v}\mathbf{B} - \mathbf{B}\mathbf{v}) = 0. \tag{4}$$

In these equations, ρ is the mass density, $\mathbf{v}$ is the velocity, $\mathbf{B}$ is the magnetic field, e is the energy density, p_{t} is the total pressure, and $\mathbf{g}$ is the gravitational acceleration vector.

The total pressure p_{t} is written as

$$p_{\mathrm{t}} = p_{\mathrm{k}} + \frac{\mathbf{B}^2}{2}, \tag{5}$$

where p_k is the kinetic pressure given by

$$p_k = (\gamma - 1)\left(e - \frac{\rho \mathbf{v}^2}{2} - \frac{\mathbf{B}^2}{2}\right). \tag{6}$$

In the SAC, the variables ρ, e and $\mathbf{B}$ are expressed in terms of perturbed and background quantities as

$$\rho = \tilde{\rho} + \rho_b, \tag{7}$$

$$e = \tilde{e} + e_b, \tag{8}$$

$$\mathbf{B} = \tilde{\mathbf{B}} + \mathbf{B}_b, \tag{9}$$

where $\tilde{\rho}$ is the perturbed density, $\tilde{e}$ is the perturbed energy, and $\tilde{\mathbf{B}}$ is the perturbed magnetic field. The background quantities with a subscript 'b' do not change in time, as we assume a magneto-hydrostatic equilibrium of the background plasma, which may have a gravitational field present, denoted by $\mathbf{g}$. Hyperdiffusion and hyperresistivity are used to achieve numerical stability of the computed solution of the MHD equations (for implementation details, see Caunt & Korpi 2001; Stein & Nordlund 1998). The full set of MHD equations, including the hyper-diffusion source terms are shown in Appendix A.

4. Implementation of Sheffield advanced code on GPUs

Earlier, it was noted that the decision to use the CUDA programming language for the development of SMAUG was made in order to enhance portability. In this section, we focus on the implementation of the SMAUG. Our approach to designing SMAUG has been to use the original SAC code as the basis. Such an approach has aided the process of code testing and verification. This is because we were able to carry out close comparisons between the codes at each stage of development.

SMAUG has been written entirely using the C programming language and CUDA. The first task was to identify the structures within SAC, which would benefit from a GPU implementation and should be written as a kernel running on the GPU. An important realization was that the hyper-diffusion algorithm requires the storage of a number of intermediate datasets and the required kernels should be designed in such a way that we avoid the need to move large quantities of data between the GPU and CPU. It was, therefore, decided to store the entire problem and the temporary data on the GPU. The single GPU tests presented here are applied to numerical domains of sizes up to $128 \times 128 \times 128$ grid points.

Investigating SMAUG using the CUDA profiler indicated that one of the most computationally intensive parts of the calculation are the hyper-diffusion source terms (A9)–(A16) in Appendix A. One of the reasons for this is the necessity to compute global maxima for the hyper-viscosity coefficients (A19) and wave speeds involved. When it is necessary to save a particular configuration, data is copied to and from the GPU host memory. When writing the SMAUG code any kernels which become large were split into smaller units. Kernels using branching logic and with loops resulting in a large number of threads were avoided. We also ensured that any conditional jumps branched equally for all threads, this was a concern for kernels used for the computation of boundary conditions.

Although SAC and SMAUG are interoperable, they are both independent applications, the source code in SAC has not been re-used in SMAUG. Implementation of the CUDA kernels started with kernels used for initialization, and then proceeded on to the development of kernels for the computation of the flux terms which are used to update the perturbed fields. Before implementing the hyper-diffusion terms, the code was tested with background fields set to zero and without the hyper-diffusion terms. SMAUG comprises a collection of kernel routines which may be divided into the following groups:

- initialization;
- computation of flux terms;
- computation of derived quantities (pressures, wave speeds, etc.);
- computation of the hyper-viscosity coefficients;
- computation of the hyper-diffusion source terms;
- boundary update kernels;
- field update routines (used to combine contributions for Runge–Kutta steps);
- finalisation routines (used to free memory on the GPU).

The flow diagram for SMAUG, illustrating the main system elements, is shown in Figure 1. Figure 2 shows the routines computing the flux updates and hyperviscosity. Kernels running on the GPU are shown in red, routines running on the CPU are shown in green. The blue blocks show units with code which run on both the CPU and GPU.

Since the SMAUG frequently refers to a large number of multi-dimensional arrays in one, two or three dimensions, all data is kept in a single 1-dimensional array. We employed an encode routine which converts a 3-dimensional index to a 1-dimensional index. The encode formula is,

$$i = i_1 + i_2 n_1 + i_3 n_1 n_2 + f n_1 n_2 n_3, \tag{10}$$

where n_1, n_2 and n_3 denote the size of the computation mesh in the x, y and z directions, respectively. The indices of the mesh point in the x, y and z directions are i_1, i_2 and i_3, respectively. The index for a particular field component is denoted by f. Thus, the formula (10) returns the one-dimensional index for a field point. A reason for using this data representation is to reduce the number of variables and to reduce the coding required to allow SMAUG to switch between 1D, 2D and 3D geometries. With data correctly organised, all of the threads within a warp are able to access data within aligned blocks of memory. This behaviour is called memory coalescing (Farber 2011). Memory alignment issues have a significant impact on the utilisation of memory bandwidth and ultimately on performance. For example, the optimal global memory bandwidth is achieved when the threads in a half warp (16 threads) load data from the same memory segment in a single transaction, otherwise, performance is degraded as a result of serial accesses to memory. One method of coalescing memory is to allow consecutive threads to access consecutive memory locations. The approach would stagger the updates for each of the field points. It should be noted that, although the M2070 (Fermi) GPU hardware is significantly improved, the issue of coalescing is still a concern. One guideline for ensuring good peformance is to use models whose size scales as 2^n.

The computation of the hyper-viscosity coefficients (A19) requires the calculation of the maximum wave speed in the computational domain. An interleaved addressing

M. K. Griffiths et al.

Figure 1. Flow diagram for SMAUG, illustrating the main system elements. The kernels running on the GPU are shown in red, and routines running on the CPU are shown in green. The blue blocks show units with code which run on both the CPU and GPU.
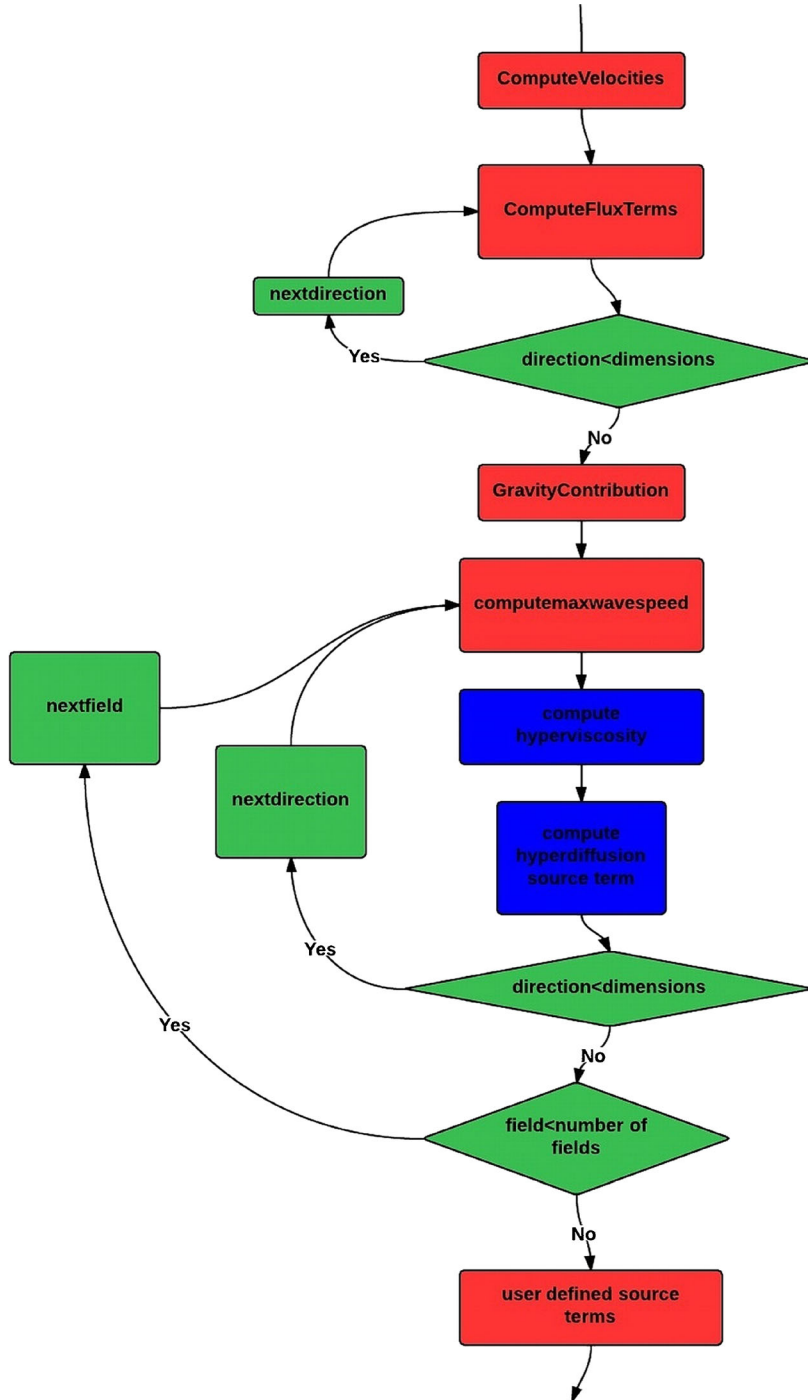
Figure 2. Flow diagram for SMAUG, illustrating the main system elements. The kernels running on the GPU are shown in red, and routines running on the CPU are shown in green. The blue blocks show units with code which run on both the CPU and GPU.

method was used for the computation of the maximum value of a vector of double precision values of the maximum wave speed. Figure 3 illustrates the interleaved addressing method for parallel reduction. The example illustrated here is a global summation. The sum of data items accessed by adjacent threads is evaluated. At the first step, every other thread is used to compute part of the reduction, as further steps are taken, the stride is doubled. This algorithm results in a significant enhancement in the performance. As discussed earlier, shared memory is much faster than global memory, so the parallel reduction algorithm benefits greatly.

Figures 4 and 5 show code segments from the reduction algorithm used to compute the wave speed maximum. Figure 4 illustrates how the interleave method is applied to each of the thread blocks. It is necessary to call the available number of threads per block to determine the size of the GPU shared memory required to store and compare the wave speeds for this computation. By accessing shared memory, we take advantage of the greater GPU data bandwidths. After the execution of all of the thread blocks, a single loop is used to obtain the required maximum value. Figure 3 shows the kernel for the reduction routine; the first part allocates the shared memory and transfers data from the GPU global memory. The interleave method is implemented by the section of enclosed comments, indicating the start and end of the repeating code block. The final line of the segment writes the partial value for the first thread, and stores the maximum values for each of the thread blocks.

For the computation of the fluxes and hyper-diffusion terms, we expect the performance to scale as N^2, this is due to the fact that we can map each thread to an element of the computational mesh. In the case of the reduction operation, the performance for techniques such as the interleave method scales as $N \ln N$. This provides a limit for the speed-up which may be achieved by SMAUG.

1	thread	0	1	2	3	4	5	6	7
Stride 1	values	11	2	3	5	-2	6	-9	5
2	thread	0		2		4		6	
Stride 2	values	13	2	8	5	4	6	-4	5
3	thread	0				4			
Stride 4	values	21	2	13	5	0	6	1	5
4	thread	0							
	values	21	2	18	5	-4	6	6	5

Figure 3. Illustration of the interleaved addressing method used for parallel reduction.

```
//Number threads per block
int NTPB=tnumThreadsPerBlock;
//Num blocks is determined by size of zeropadded 2^n size array
int numBlocks = (ndimp+NTPB-1) / NTPB;

//Shared memory
int smemSize = NTPB * sizeof(double);

//Array to store maximum values for reduction in host memory
double *h_cmax = (double*)malloc(numBlocks*sizeof(double));

cudaMalloc((void**)&d_cmax, numBlocks*sizeof(double));
//Array to store maximum values for reduction in GPU global memory
cudaMalloc((void**)&d_bmax, numBlocks*sizeof(double));

//set maximum value to zero and update values in GPU memory
(*p)->cmax=0.0;
cudaMemcpy(*d_p, *p, sizeof(struct params), cudaMemcpyHostToDevice);
cudaMemcpy(*wd, *d_wd, NDERV*dimp*sizeof(real), cudaMemcpyDeviceToHost);
cudaMemcpy(*d_wtemp, ((*wd)+(cfast*dimp)), dimp*sizeof(real),cudaMemcpyHostToDevice);
int i=0;

//find the maximum in each block
for(i=0;i<numBlocks;i++)
            h_cmax[i]=0;
cudaMemcpy(d_bmax, h_cmax, numBlocks*sizeof(double), cudaMemcpyHostToDevice);
reductionmax_parallel<<<numBlocks,NTPB,smemSize>>>(d_bmax,*d_wtemp,ndimp);
cudaThreadSynchronize();
cudaMemcpy(h_cmax, d_bmax, numBlocks*sizeof(double), cudaMemcpyDeviceToHost);

//compare the maxima for all of the blocks and determine maximum value
for( i=0;i<numBlocks;i++)
              if(h_cmax[i]>((*p)->cmax)) ((*p)->cmax)=h_cmax[i];

//determine maximum value of sound speed
cudaMemcpy(*d_wtemp, ((*wd)+(soundspeed*dimp)), dimp*sizeof(real), cudaMemcpyHostToDevice);
```

Figure 4. Host code used to call reduction routine CUDA kernel. The start of the section shows the initialization of the shared memory size and the number of threads per block. The middle section shows the allocation of CUDA device memory used to store the maximum values for each thread block. The final section calls the CUDA kernel, determines the maximum value for the thread blocks, and writes the global maxima (for all thread blocks); this resulting value is finally copied to the device memory.

SMAUG consists of a collection of 3 source files for code that runs on the host system and 23 kernel source files for code that executes on the GPU. Using conditional compilation through the compiler preprocessor, the code has a degree of flexibility and provides a number of defined preprocessor variables which may be altered within an input file used by the *make* routine. The optional preprocessor variables enable switching between 1D, 2D or 3D geometries, running across multiple GPUs, user-provided source terms or switching between double and single precision.

SMAUG was developed using a Waterfall software engineering methodology (Pressman 1997). SMAUG is licensed under the GNU General Public License (GPL v3.0). The decision to make SMAUG freely available should ensure that the software will always be available and should encourage the user community to extend and to accelerate new scientific and algorithmic developments. SMAUG is available at https://code.google.com/p/smaug/.

An important consideration for SMAUG is its inter-operability with the existing SAC code. For this reason, SMAUG has been deliberately designed to read and write configuration files generated by the SAC initialisation routines. The distribution routines are used to scatter and gather configuration files for use in MPI enabled

```
extern __shared__ double partialResult[];

int i;
partialResult[tid]=0.0;
if(iindex<ndimp)
            partialResult[tid]=temp[iindex];
__syncthreads();

/*Start of reduction loop*/
for(unsigned int s=1; s < blockDim.x; s *= 2) {
      if ((tid % (2*s)) == 0) {
          if(partialResult[tid+s]>partialResult[tid])
              partialResult[tid]=partialResult[tid + s];
      }
      __syncthreads();
}
/*End of reduction loop*/


__syncthreads();
if(tid==0)
{
    cmax[blockIdx.x]=partialResult[0];
    temp[blockIdx.x]=partialResult[0];
}
    __syncthreads();
```

Figure 5. CUDA kernel implementation of the interleave method for the reduction algorithm. The first part of the kernel initializes the shared memory for each thread block. The array of shared memory values is set with the wave speed values stored in the GPU memory. The interleave operation can only perform after synchronization of the threads for each block. The repeating segment of code which follows is the interleave operation; each even numbered thread compares a pair of values in the shared memory. The reduction occurs as the repetition strides over each set of neighbouring values. Before each stride can commence, it is necessary for all the threads in the block to be synchronized. The nature of this reduction operation is such that the required value is stored in the first memory location in the array, this result in shared memory is transferred to the array of results for each of the thread blocks.

problems. IDL and Matlab visualization routines already used with SAC and VAC may be re-used. The source distribution provides a number of sample problems, which are configured using the *make* routine. Verification tests have been performed to verify the results generated by SMAUG and to assess the performance. Results are discussed in the following two sections.

5. Verification tests

A challenge for solving hyperbolic differential equations such as the MHD equations (A1)–(A8) arises when solutions are sought for systems exhibiting discontinuities in

their solution. A number of tests have been performed in 1D, 2D and 3D geometries. Such tests are crucial for ensuring the robustness of the solver. We verified that SMAUG reproduces the same results as SAC for the Brio-Wu, Orszag-Tang, a 3D blast-wave problem, and a 3D model of wave propagation in a gravitationally stratified solar atmosphere. Details of the first three standard tests are given in Appendix B. One of the objectives of the tests is to ensure the correct implementation of the hyper-diffusion algorithm for stabilizing the solutions from numerical instabilities.

To isolate algorithmic differences, we ran both codes with the hyper-diffusion switched off. As the agreement between SAC and SMAUG was progressively improved, each of the hyper-diffusion contributions was tested in turn. For example, testing would be undertaken with just the hyper-diffusion contribution for the density enabled. Finally, all of the hyper-diffusion source terms were included. The percentage difference between SAC and SMAUG for the total density and total energy for the Orszag-Tang test is shown in Figure 6. To understand the differences observed in Figure 6, we performed a number of experiments using the Orszag-Tang test. As well as comparing results generated using SAC and SMAUG we compared the difference between results for SAC and SMAUG, when a small error in the density was introduced at the centre of the numerical grid. The magnitude of the introduced difference was at the level of the round-off error. These experiments resulted in deviations in the total energy and momentum, which were of the same order of magnitude as those presented in Figure 6. It is well understood that NVIDIA GPUs are fully compliant
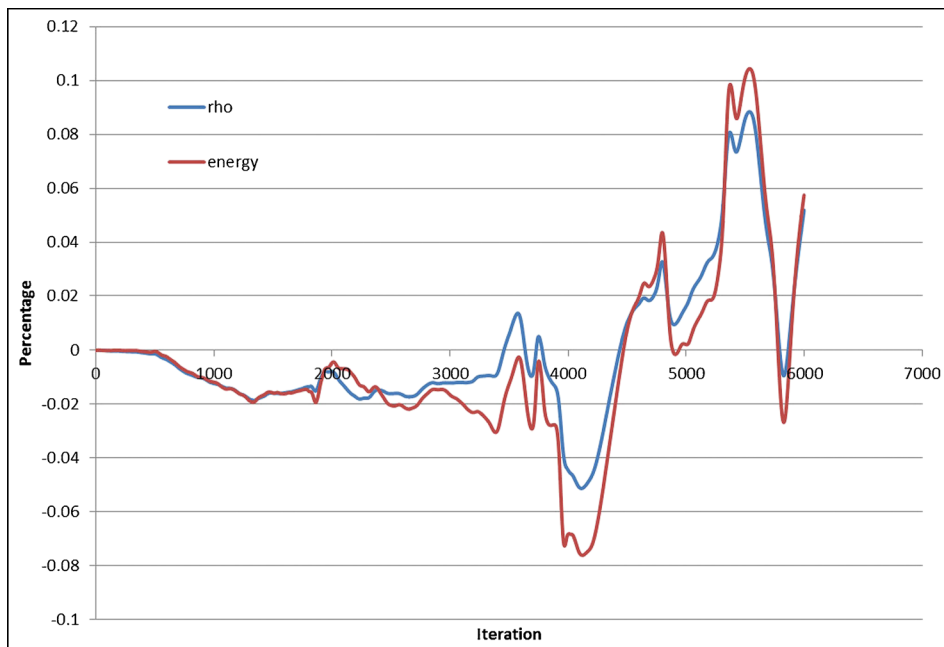

Figure 6. Percentage difference between SAC and SMAUG calculation of total energy and total density at each iteration for the Orszag-Tang vortex problem computed on a 256×256 grid.

with the IEEE-754 standard for floating point operations (Whitehead & Fit-florea 2011). Differing computer processor architectures can lead to numerical differences when performing a computation with a large number of operations, which may have large differences in magnitude.

5.1 *MHD wave propagation in magnetized and gravitationally stratified 3D solar atmosphere*

In this section, we provide the numerical results of modelling of MHD wave propagation in an open magnetic flux tube embedded in the solar atmosphere. The background model and initial setup of the problem are similar to that used previously (Fedun 2009; Fedun *et al.* 2011a, b, c). This simulation examines SMAUG's capability for modelling the physical processes in the magnetized and stratified media, expanding from the photosphere through chromosphere, transition region and into the solar corona.

It is planned to use SMAUG for modelling the physical processes of the magnetically coupled solar interior and atmosphere. Work is also in progress to study the propagation of energy into the solar atmosphere by using a vibrating membrane to drive the lower boundary of the simulation box. Such studies provide insight into the coupling of the solar global eigenmodes, with motions in the solar atmosphere.

Here, we present initial results for modelling of MHD waves in a chromospheric magnetic flux tube. The problem tests SMAUG's capability for modelling a gravitationally strongly stratified 3D solar atmosphere. Vortex motion at the foot point of the magnetic flux tube is excited by the a time-dependent torsional driver (Fedun *et al.* 2011a, b, c), we used a velocity amplitude of 20 m/s with no time-dependence for the driver. An additional model was repeated with the velocity amplitude set to 200 m/s, and the period of the driver set to 120 s. In Figure 7 we have shown a 2D vertical cross-cut through the 3D numerical domain for the V_y velocity component. Mumford *et al.* (2014) are using SMAUG in a study of the effect of photospheric motions on the generation of MHD waves in low solar atmospheric flux tubes.

6. Performance measurements

The performance of SAC and SMAUG was assessed by running the Orszag-Tang problem for varying problem sizes on Intel Xeon (X5650 CPU (6 cores, 2.67 GHz clock frequency), E5 2670 CPU (8 cores, 2.6 GHz)) and GPU NVIDIA (C1060 (Tesla), M2070 (Fermi), K20 (Kepler)) computer architectures.

The NVIDIA GPU device architecture and programming methodology was described in section 2. For all of the CPU tests, SAC was compiled using the Portland Fortran90 compiler version 10.2 with full optimization. For the CPU runs, the problems were initially run using a single processor core. Figures 8 and 9 show comparisons of the computational time taken to perform 100 iterations with a varying grid size for each of the processors. Figures 8 and 9 compare the timings for the Intel X5650 and Intel E5-2670 CPU, respectively. The result clearly indicates that as the grid size increases, both the C1060 and M2070 GPUs offer significant improvement
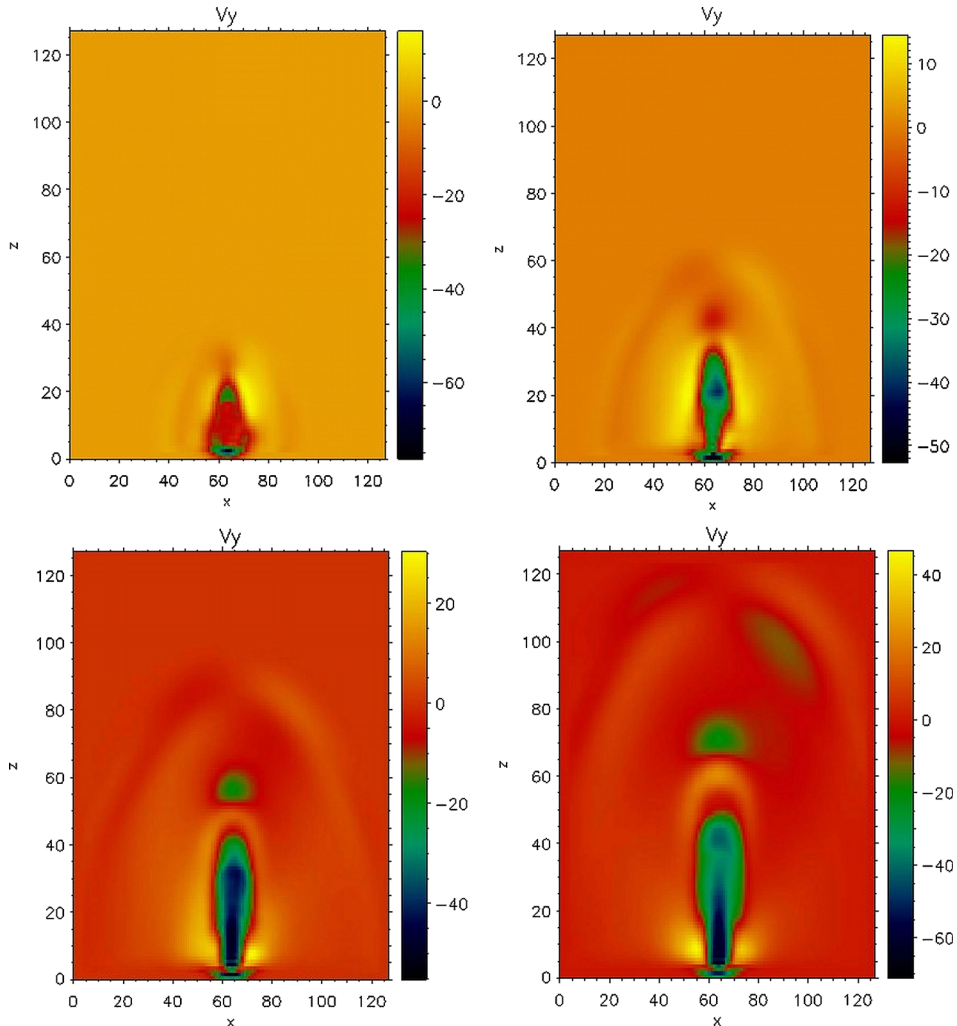
Figure 7. The vertical cross-cut through the 3D numerical domain. Snapshots of the V_y component of the velocity response showing the temporal evolution of the initial perturbation generated by 120 s period torsional driver at $t = 32$, 64, 96 and 128 s in the open magnetic flux tube, respectively. The color scale shows the V_y perturbations in km/s. For more details of mathematical set-up of this physical problem, see Fedun *et al.* (2011a, b, c).

in the time taken to complete an iteration. This is the initial evidence that the GPU performance scales well with grid size.

To obtain a better understanding of the performance benefits of running the problem on a GPU, we compare the speed-up factor for the C1060, M2070 and K20 GPUs for each of the tested CPUs. Defining t_c as the time to complete 100 iterations on the CPU and t_g as the time to complete 100 iterations on the GPU, the speed-up factor is defined as t_c/t_g. Figures 10 and 11 show the speed-up factor versus the model numerical size, comparing C1060, M2070 and K20 against the Intel Xeon
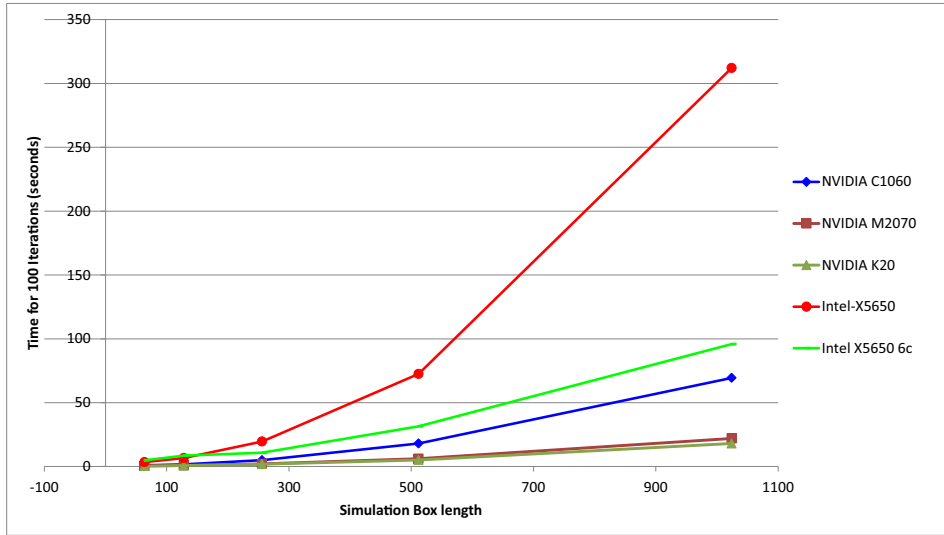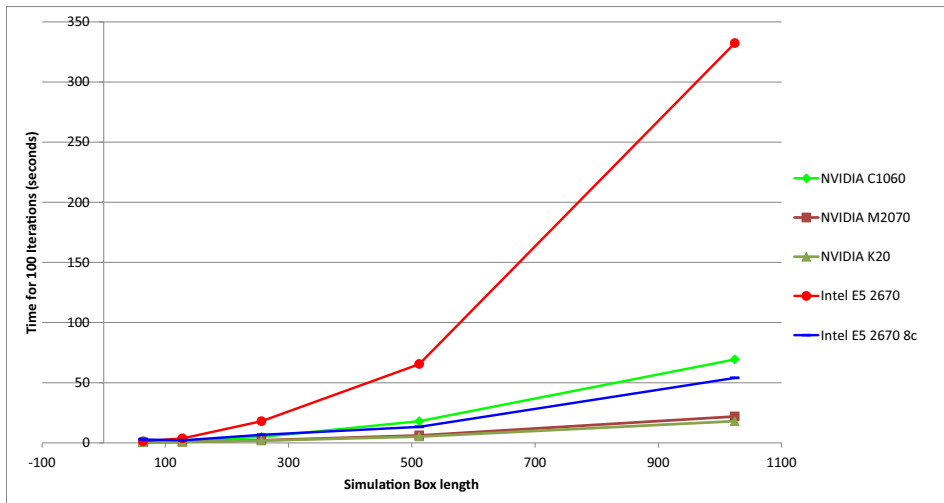
Figure 8. Comparison of the run times for 100 iterations of the Orszag-Tang problem. The run times obtained with the Intel X5650 processor and different NVIDIA GPUs and with varying problem sizes are shown.

X5650 CPU and Intel Xeon E5-2670 CPU, respectively. The C1060 offers a speed-up of 4 over the Intel Xeon X5650 CPU, in contrast to the M2070 GPU, we observe that this benefit does not improve with increasing the problem size. This may be understood in terms of the enhanced double precision performance for the M2070

Figure 9. Comparison of the run times for 100 iterations of the Orszag-Tang problem. The run times obtained with the Intel E5-2670 processor and different NVIDIA GPUs and with varying problem sizes are shown.
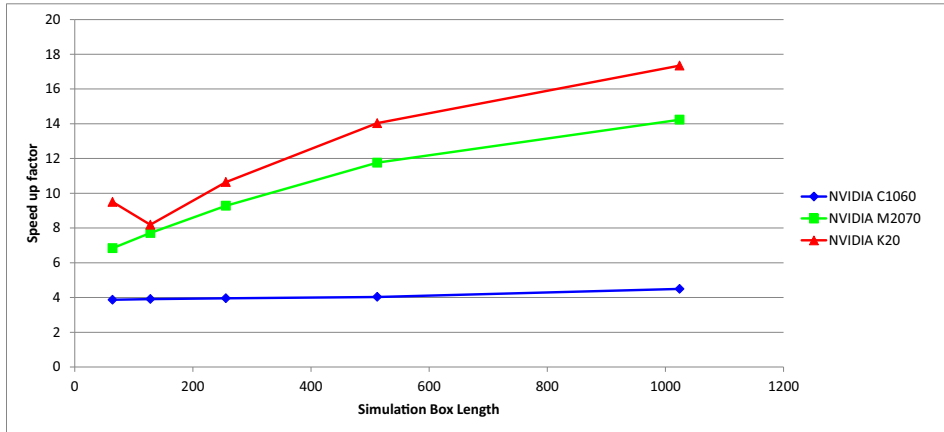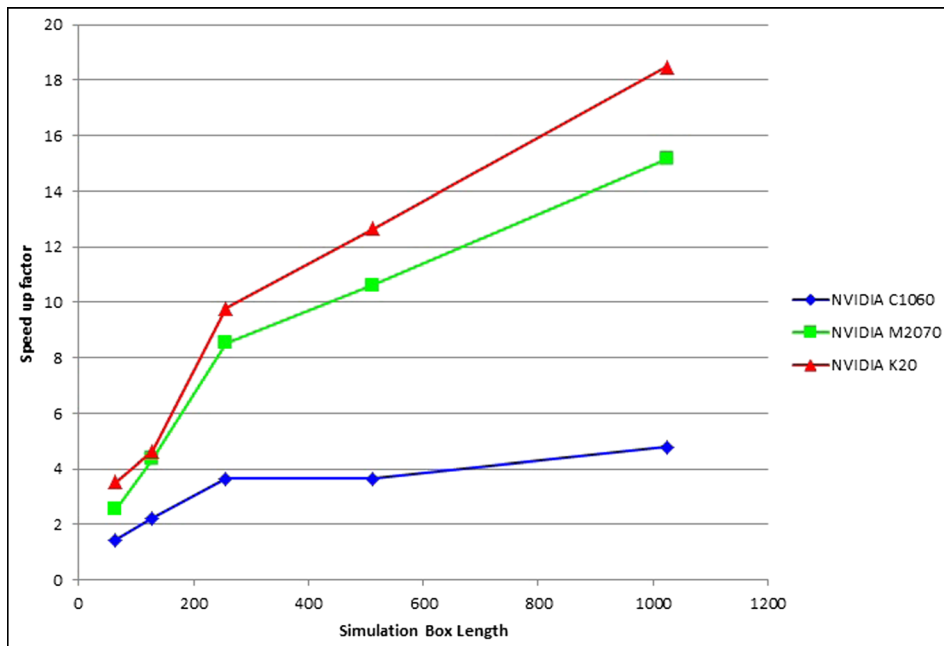
Figure 10. Measured speed-up factors for the Orszag-Tang test problem performed on different numerical grids, comparing NVIDIA GPUs against the Intel X5650 CPU.

GPU. This is a result of the higher memory bandwidth achieved for double precision values. One of the reasons for the enhanced performance with increased problem size is due to the number of threads assigned to the cores. Increasing the occupancy in this way hides some of the effects of memory latency in transferring data from the GPU global memory to the block memory. The results demonstrate that for the

Figure 11. Measured speed-up factors for the Orszag-Tang test problem performed on different numerical grids, comparing NVIDIA GPUs against the Intel E5-2670 CPU.
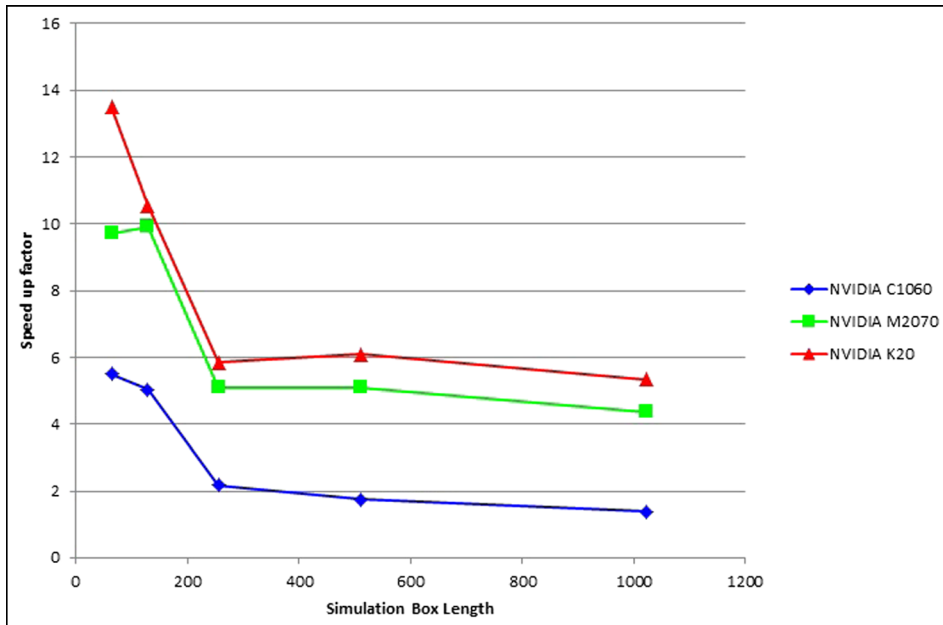
Figure 12. Measured speed-up factors for the Orszag-Tang test problems of different numerical grid size and comparing NVIDIA GPUs against the fully occupied Intel X5650 CPU.

NVIDIA K20, the SMAUG achieves a speed-up of up to 18 times when compared to a single Intel Xeon X5650 processor core, such a speed-up is achieved for problem sizes which fully occupy the GPU.

In Figures 12 and 13 we present the performance for the GPUs against fully occupied Intel CPUs. Figure 12 compares the speed-up achieved using a fully occupied 6 core Intel Xeon X5650 CPU and Figure 13 compares the speed-up achieved compared to the 8 core Intel Xeon E5-2670. The results shown in Figures 12 and 13 display a characteristic dip for moderate levels of the problem size. This dip results from the performance for the CPU, at this domain size, the multicore CPU benefits from optimal utilization of CPU cache memory. The performance comparison with multi-core CPUs is dependent on the decomposition of the problem across the multicore processor; we will return to this issue when SMAUG is distributed across multiple GPUs. We note, however, that the C1060 GPU is still able to offer significant performance enhancement compared to the Intel Xeon X5650 CPU. Using the Kepler K20, we have acheived a speed-up factor of 3, compared to a fully occupied Intel Xeon E5-2670 CPU.

It was pointed out that the performance of the algorithm scaled as $N \ln (N)$ arises from the reduction operation used by the computation of the maximum wave speed. Application profiling indicates that the computation of the boundary conditions does not provide a significant contribution to the acceleration of the algorithm. This is due to the limited quantity of data used in the update process. As indicated in the results, it was observed that the GPU performance improved as the problem size was increased. A further issue with the computation of the boundary terms is due to the non-coalescent memory access.
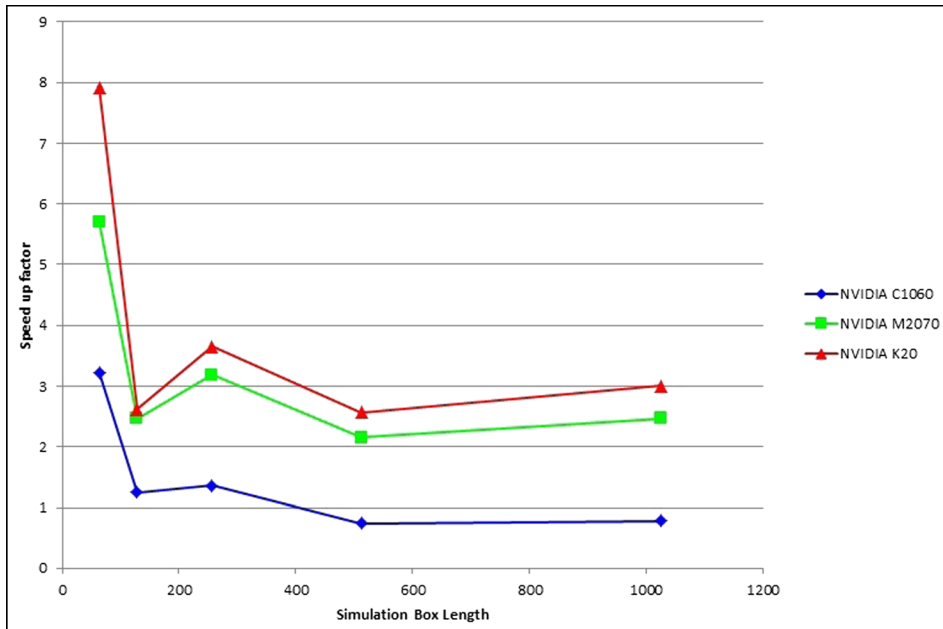
Figure 13. Measured speed-up factors for the Orszag-Tang test problems of different sizes of the numerical grid comparing with NVIDIA GPUs against the fully occupied Intel Xeon E5-2670 CPU.

7. Conclusions

We have presented a description of a GPU implementation of a non-linear MHD solver employing artificial numerical diffusion to control numerical instabilities. The code is developed on the basis of the well-known VAC (Toth *et al.* 1998) and SAC (Shelyag *et al.* 2008) platforms. Both of these applications have been tested extensively and provide good agreement with the standard 1–3D HD and MHD tests. The design of the SMAUG implementation has followed that of VAC and SAC very closely.

The main achievement of SMAUG is the performance benefit which we have demonstrated. This has resulted in a significant cost saving for hardware. To summarise, running the 1024×1024 grid Orszag-Tang problem on a single M2070 GPU is 18 times faster than the same problem running on a single core of an Intel Xeon X5650 CPU. Running the same problem over all 6 cores of the Intel Xeon X5650 reduced the speed-up achieved by the M2070 to a factor of 4. Our results demonstrate that the performance benefits of using a GPU improved as the problem size was increased.

One of the limitations of the current code is related to the problem size. With the 6 GB of GPU RAM, the problem size is limited to a model of $128 \times 128 \times 128$ grid points. This will be improved in two ways. Firstly, by making use of MPI to distribute the numerical grid over many GPUs; and, secondly, optimizing the usage of temporary data storage employed by SMAUG. Although we have already successfully scaled the solution to many GPUs, one of the main improvements will be to employ

the so-called GPU direct software to enable direct peer-to-peer communications between GPU host memory.

Further motivations for using GPUs are due to good characteristics of compute performance per unit of power and compute performance per unit of costs. The development roadmap presented by NVIDIA indicates that the Kepler GPU card will aim for 5 double precision gigaflops per watt and the Maxwell GPU will achieve around 15 gigaflops per watt. This represents a 3-fold and 10-fold improvement when compared to the current Fermi GPUs. Future work will consider the scaling of the performance, as SMAUG is used to distribute a problem across multiple GPUs and multiple CPU cores. For this performance assessment, it is necessary to evaluate the dependency of the scaling on the different ways of distributing a domain across the processing elements (the GPUs or CPU cores). It is also important to understand how MHD algorithms may best exploit future multi-core processor architectures such as the Intel many integrated core (MIC) architecture or the NVIDIA Maxwell GPU, expected in 2015.

The GPU-based algorithm described here, will be used to provide a platform for the simulation of wave propagation in the strongly stratified and magnetised solar interior and atmosphere. Initial testing has demonstrated that the code is more than able to handle such problems with an excellent agreement compared with the original SAC code of Shelyag *et al.* (2008). For the various complex test cases which have been considered, the results show that the SMAUG successfully reproduces physical processes. Consequently, we are led to the conclusion that the code can be used as a reliable tool for modelling the physics and, perhaps, even more importantly, predicting new features of astrophysical plasmas.

Acknowledgements

Appendices

Appendix A. The full MHD equations

The full MHD equations, including the hyper-diffusion source terms, read as

$$\frac{\partial \tilde{\rho}}{\partial t} + \nabla \cdot (\mathbf{v}(\rho_{\rm b} + \tilde{\rho})) = 0 + D_\rho(\tilde{\rho}), \qquad (A1)$$

$$\frac{\partial [(\tilde{\rho} + \rho_{\rm b})\mathbf{v}]}{\partial t} + \nabla \cdot (\mathbf{v}(\tilde{\rho} + \rho_{\rm b})\mathbf{v} - \tilde{\mathbf{B}}\tilde{\mathbf{B}})$$

$$-\nabla[\tilde{\mathbf{B}}\mathbf{B_b} + \mathbf{B_b}\tilde{\mathbf{B}}] + \nabla \tilde{p}_{\rm t} = \tilde{\rho}\mathbf{g} + \mathbf{D}_{\rho\rm v}[(\tilde{\rho} + \rho_{\rm b})\mathbf{v}], \qquad (A2)$$

$$\frac{\partial \tilde{e}}{\partial t} + \nabla \cdot (\mathbf{v}(\tilde{e} + e_{\mathrm{b}}) - \tilde{\mathbf{B}}\tilde{\mathbf{B}} \cdot \mathbf{v} + \mathbf{v}\tilde{p}_{\mathrm{t}}) - \nabla[(\tilde{\mathbf{B}}\mathbf{B}_{\mathrm{b}} + \mathbf{B}_{\mathrm{b}}\tilde{\mathbf{B}}) \cdot \mathbf{v}]$$

$$+ p_{\mathrm{tb}}\nabla \mathbf{v} - \mathbf{B}_{\mathrm{b}}\mathbf{B}_{\mathrm{b}}\nabla \mathbf{v} = \tilde{\rho}\mathbf{g} \cdot \mathbf{v} + D_{\mathrm{e}}(\tilde{e}), \quad \text{(A3)}$$

$$\frac{\partial \tilde{\mathbf{B}}}{\partial t} + \nabla \cdot (\mathbf{v}(\tilde{\mathbf{B}} + \mathbf{B}_{\mathrm{b}}) - (\tilde{\mathbf{B}} + \mathbf{B}_{\mathrm{b}})\mathbf{v}) = 0 + \mathbf{D}_{\mathrm{B}}(\tilde{\mathbf{B}}), \quad \text{(A4)}$$

where

$$\tilde{p}_{\mathrm{t}} = \tilde{p}_{\mathrm{k}} + \frac{\mathbf{B}^2}{2} + \mathbf{B}_{\mathrm{b}}\tilde{\mathbf{B}}, \quad \text{(A5)}$$

$$\tilde{p}_{\mathrm{k}} = (\gamma - 1)\left(\tilde{e} - \frac{(\rho_{\mathrm{b}} + \tilde{\rho})\mathbf{v}^2}{2} - \mathbf{B}_{\mathrm{b}}\tilde{\mathbf{B}} - \frac{\tilde{\mathbf{B}}^2}{2}\right), \quad \text{(A6)}$$

$$p_{\mathrm{tb}} = p_{\mathrm{kb}} + \frac{\mathbf{B}_{\mathrm{b}}^2}{2}, \quad \text{(A7)}$$

$$p_{\mathrm{kb}} = (\gamma - 1)\left(e_{\mathrm{b}} - \frac{\mathbf{B}_{\mathrm{b}}^2}{2}\right). \quad \text{(A8)}$$

The MHD equations for momentum (A2) and energy (A3) are re-ordered by taking into account the magnetohydrostatic equilibrium condition. It was found advantageous to use this, apparently more complex form, of the MHD equations for numerical simulations of processes in a gravitationally stratified plasma. The 0+ term in equations (A1) and (A4) indicates that the right hand side of these expressions are zero without the inclusion of the numerical diffusion terms. It is important to note that the fully perturbed expressions include diffusive source terms on the right-hand-side. These terms are employed to stabilise the computational solutions using proven hyperdiffusion method (Caunt & Korpi 2001; Stein & Nordlund 1998). A helpful discussion of the hyper-diffusion method and its implementation is provided by Sødergaard and Hansen (1999). The numerical algorithm implemented to solve equations (A1)–(A8) is based on a fourth order central differencing method for spatial derivatives and a second or fourth order Runge-Kutta solver for the time derivatives. It is important to note that, since we use central differencing, the quantity $\nabla \cdot \mathbf{B}$ is conserved and may be constrained to zero. Schemes preserving the constraint $\nabla \cdot \mathbf{B} = 0$ have been reviewed by Tóth (2000). The mechanism of hyperdiffusion is used to control the stability of the solutions at discontinuities and shock fronts in equations (A1)–(A4). The additional hyper-diffusive terms for the perturbed density and energy are written as

$$D_{\rho} = \sum_i \frac{\partial}{\partial x_i} \nu_i(\rho) \frac{\partial}{\partial x_i}\tilde{\rho}. \quad \text{(A9)}$$

The term D_{e} in the energy equation (A3) consists of three parts:

$$D_{\mathrm{e}} = D_{\mathrm{e}}^{\mathrm{diffusive}} + D_{\mathrm{e}}^{\mathrm{viscous}} + D_{\mathrm{e}}^{\mathrm{ohmic}}, \quad \text{(A10)}$$

which describe thermal diffusion, viscous and ohmic heating of the plasma, respectively,

$$D_{\mathrm{e}}^{\mathrm{diffusive}} = \sum_i \frac{\partial}{\partial x_i} v_i(e) \frac{\partial}{\partial x_i} \tilde{\epsilon}, \tag{A11}$$

where $\tilde{\epsilon}$ is the thermal energy perturbation,

$$\tilde{\epsilon} = \tilde{e} - (\rho_{\mathrm{b}} + \tilde{\rho}) \frac{\mathbf{v}^2}{2} - \frac{\tilde{\mathbf{B}}^2}{2}. \tag{A12}$$

The hyperviscous and ohmic heating terms in equation (A10) are set as follows:

$$\mathbf{D}_{\mathrm{e}}^{\mathrm{visc}} = \nabla \cdot (\mathbf{v} \cdot \tau), \tag{A13}$$

and

$$\mathbf{D}_{\mathrm{e}}^{\mathrm{ohmic}} = \nabla \cdot (\mathbf{B} \times \varepsilon). \tag{A14}$$

For the vector variables the hyper-diffusive terms are

$$\mathbf{D}_{\rho v} = \nabla \cdot \tau, \tag{A15}$$

and

$$\mathbf{D}_{\mathbf{B}} = -\nabla \times \varepsilon. \tag{A16}$$

Here, τ is the viscous stress tensor

$$\tau_{kl} = \frac{1}{2}(\tilde{\rho} + \rho_b) \left[v_k(v_l) \frac{\partial v_l}{\partial x_k} + v_l(v_k) \frac{\partial v_k}{\partial x_l} \right], \tag{A17}$$

and ε is defined by

$$\varepsilon_k = \epsilon_{klm} \left[v_l(\tilde{B}_m) \frac{\partial \tilde{B}_m}{\partial x_l} \right], \tag{A18}$$

where ϵ_{klm} is the Levi-Civita symbol and a summation of the indices l and m is taken. The hyper-viscosity coefficient for the variable u in the i-th direction is

$$v_i(u) = c_2^u \Delta x_i v_t \frac{\max \left| \Delta_i^3 u \right|}{\max \left| \Delta_i^1 u \right|}, \tag{A19}$$

where $v_t = v_a + v_s$ is the sum of the maximum Alfvén and sound speeds in the domain, Δ_i^3 and Δ_i^1 are forward difference operators to third and first order, respectively, in the i-th direction. The spatial resolutions are given by Δx_i. The coefficient c_2^u is set such that it ensures numerical stability of the solution. The SAC code uses the MPI VAC software as the basis for its implementation (Tóth 1996) that includes the MPI domain decomposition. However, this MPI implementation had to be generalized to include the exchange of the hyper-diffusive contributions for the neighbouring domains.

Appendix B. Verification tests

B1. Brio and Wu shock tube

The Brio and Wu shock tube is a 1D ideal MHD test problem in which the initial conditions of the model feature a discontinuity in the centre of the configuration, i.e., the left and right states are initialized to different values at $x = 0.5$ (Brio & Wu 1988). On either side of the discontinuity the initial parameters are: $p_l = 1$, $p_r = 0.1$, $\rho_l = 1$, $\rho_r = 0.125$, $B_{yl} = 1$, $B_{yr} = -1$, $B_x = 0.75$. Brio and Wu used an upwind differencing scheme and a linearization method called the Roe procedure to solve a Riemann-type problem (Roe 1981). The Riemann problem is the solution of a conserved system which has states separated by a discontinuity. In the method employed by Brio and Wu, the exact solution is approximated by a linearised version, which is averaged on either side of the discontinuity. Running the problem on a numerical domain with 800 grid points, gives an excellent agreement with the original SAC code results (Shelyag *et al.* 2008). Figure B1 exhibits features such as the slow compound wave at $x = 0.475$, the contact discontinuity at $x = 0.55$, the slow shock at $x = 0.65$; and the fast rarefaction at $x = 0.85$ are observed. This test ensures the validity of the code by running the numerical simulation along the x, y and z axis in turn. There was found to be no dependency on the orientation of the Brio–Wu and shock tube problem.

B2. Orszag–Tang vortex

The complexities in the Orszag–Tang vortex present a challenging problem with which the code may be further validated in two dimensional geometry (Orszag & Tang 1979). The code is required to be sufficiently robust so that it can handle
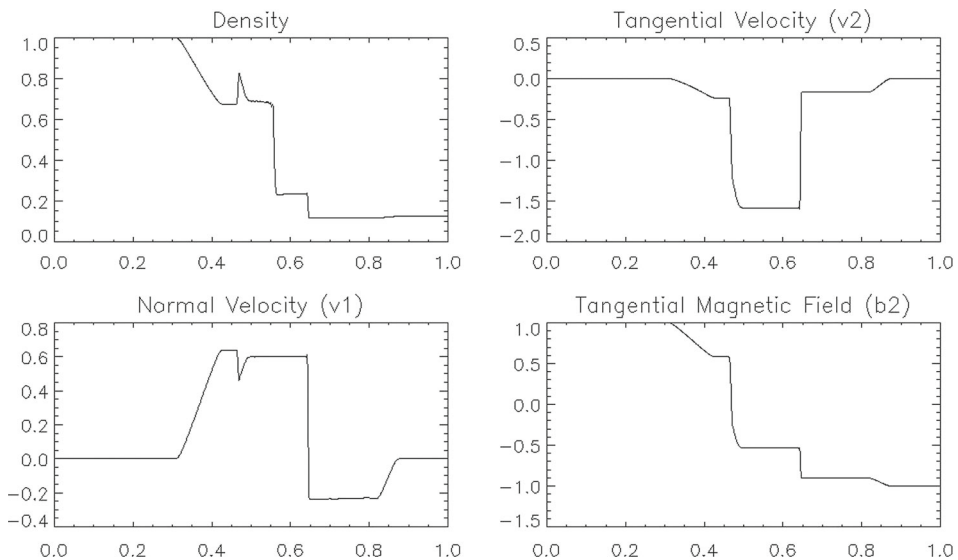

Figure B1. A snapshot of the Brio and Wu problem numerical solution taken at time $t = 0.1$. The density, tangential velocity, normal velocity and tangential magnetic field are shown.

supersonic MHD turbulence. For this test, a 512×512 simulation box was used. The boundaries of the computational domain were set as periodic. This problem provides an excellent opportunity to test the background terms by assigning the initial density, initial magnetic field and internal energy to the respective background fields.

The density distributions at $t = 0.1, 0.26, 0.42$ and 0.58 s generated by SMAUG are shown in Figure B2 and demonstrate a very good and convincing agreement with their counterpart output of SAC (Shelyag *et al.* 2008). The results demonstrate the robustness of the code since it is now able to provide well-defined shock fronts.

B3. Blast waves in 2D and 3D

This section extends the testing of SMAUG from two-dimensions to a three-dimensional grid with an emphasis on well-characterised blast problems. A blast wave in hydrodynamics is the pressure and flow which results after the release of a highly localized amount of energy. A good astrophysical example is the explosion of a supernova. The comparison between SAC and SMAUG using blast wave problems was particularly useful for checking the extension of the application from 2D to 3D. Also, by employing the Sedov shock wave, we were able to concentrate the
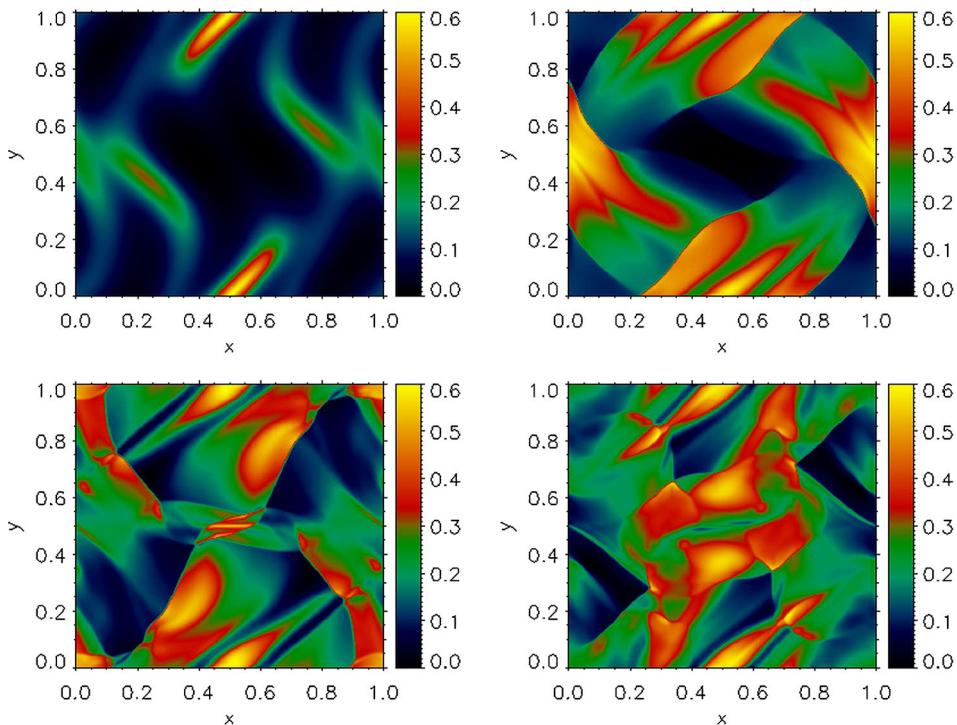

Figure B2. Orszag–Tang vortex problem computed with SAMUG on a 512×512 grid. The density distribution at $t = 0.1$ s (*top left*), 0.26 s (*top right*), 0.42 s (*bottom left*), 0.58 s (*bottom right*) is shown.
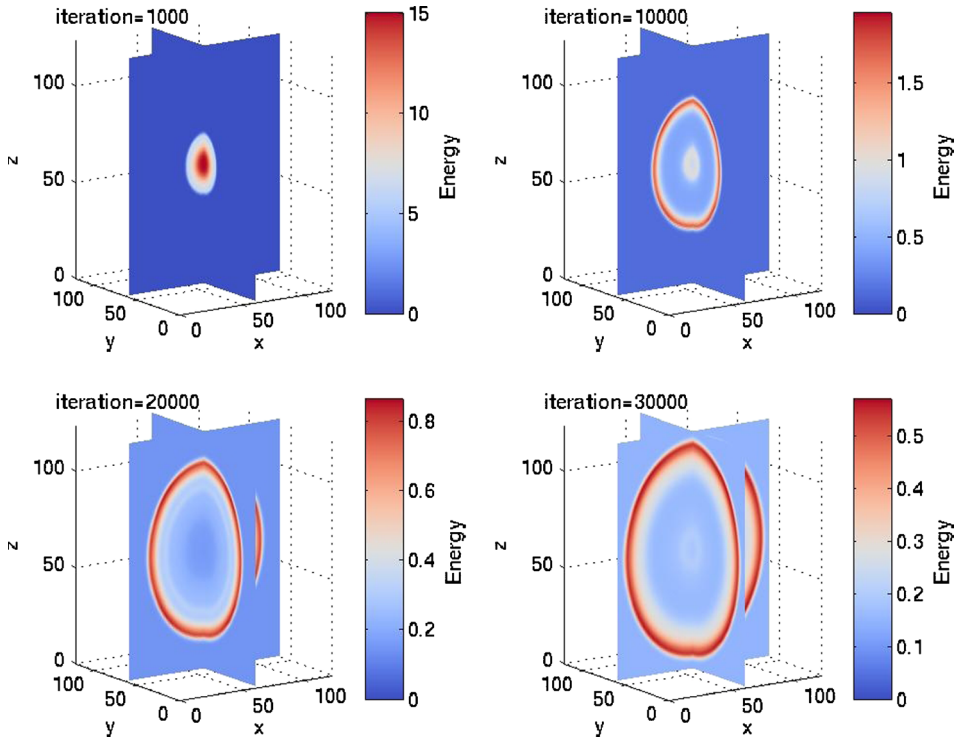
Figure B3. The result of simulation of Sedov blast test problem on $128 \times 128 \times 128$ numerical grid with SMAUG. The energy distributions as a function of time are shown at time 1000 (*top left*), 10000 (*top right*), 20000 (*bottom left*) and 30000 (*bottom right*) iterations, respectively.

hyperdiffusion tests on the momentum terms. Additionally, the Sedov shock wave problem enables a test of the code's capability to model spherically symmetric systems using Cartesian geometry. A solution of the blast wave problem, known as the similarity solution, was developed independently by Taylor (1950) and Sedov (1946). The similarity analysis provides a prediction of the radius of the shock front as a function of time, i.e.,

$$R(t) = \left(\frac{E}{\rho_0} \right)^{\frac{1}{5}} t^{\frac{2}{5}}, \tag{B1}$$

where E is the energy released by the explosion, ρ_0 is the density of the surrounding medium. The similarity solution can be derived by starting from the expression for the radius of the shock front as

$$R(t) = E^{\zeta} \rho_0^{\eta} t^{\xi}. \tag{B2}$$

By ensuring that both sides of equation (B2) have the same dimensions, the indices ζ, η and ξ can be determined using the similarity solution given by equation (B1). The simulations were performed using a $128 \times 128 \times 128$ numerical domain with a side of length 1.0 and a surrounding space of uniform density of 1.0 (Figure B3). The initial

pressure was set to $P = 0.1$, however, for a shell of radius, $r < 0.1$, the pressure was set to 10.0. The initial velocity was zero and the adiabatic parameter was set to 5/3.

References

Brio, M., Wu, C. C. 1988, *Journal of Computational Physics*, **75**, 400.
Caunt, S. E., Korpi, M. J. 2001, *AA*, **369**, 706.
Chmielewski, P., Srivastava, A. K., Murawski, K., Musielak, Z. E. 2013, *MNRAS*, **428**, 40–49.
Chmielewski, P., Murawski, K., Musielak, Z. E., Srivastava, A. K. 2014, *AJ*, **793(1)**, 43.
Cook, S. 2012, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs* (Applications of GPU Computing Series), Morgan Kaufmann.
Farber, R. 2011, CUDA Application Design and Development, Morgan Kaufmann.
Fedun, V., Erdélyi, R., Shelyag, S. 2009, *SP*, **258**, 219.
Fedun, V., Verth, G., Jess, D. B., Erdélyi, R. 2011c, *AJL*, **740**, L46.
Fedun, V., Shelyag, S., Verth, G., Mathioudakis, M., Erdélyi, R. 2011b, *Annales Geophysicae*, **29**, 1029.
Fedun, V., Shelyag, S., Erdélyi, R. 2011a, *AJ*, **727**, 17.
Govett, M. 2009, The Fortran-to-CUDA compiler.
Han, T. D., Abdelrahman, T. S. 2011, *IEEE Transactions*, **22**, 78.
Harvey, M. J., De Fabritiis, G. 2011, *Computer Physics Communications*, **182**, 1093.
Hasan, S. S., van Ballegooijen, A. A., Kalkofen, W., Steiner, O. 2005, *AJ*, **631**, 1270.
Kestener, P., Fromang, S. 2012, *Astronomical Society of the Pacific Conference Series*, **459**, 222.
Kirk, D. 2010, *Programming Massively Parallel Processors: A Hands-on Approach* (Applications of GPU Computing Series), Morgan Kaufmann.
Kirk, D., Hwu, W. 2010, *Programming Massively Parallel Processors: A Hands-on Approach*, Elsevier Direct.
Lin, L., Ng, C.-S., Bhattacharjee, A. 2011, GPU Accelerated Reduced MHD Simulations of Coronal Loops, *Twenty Second International Conference on Numerical Simulations of Plasmas*.
Lindholm, E., Nickolls, J., Oberman, S., Montrym, J. 2008, *IEEE*, **28**, 39.
McLaughlin, J. A., De Moortel, I., Hood, A. W., Brady, C. S. 2009, *AA*, **493**, 227.
Mumford, S., Fedun, V., Erdelyi, R. 2014, *Generation of Magnetohydrodynamic Waves in Low Solar Atmospheric Flux Tubes by Photospheric Motions*, ArXiv e-prints.
Orszag, S. A., Tang, C. M. 1979, *Journal of Fluid Mechanics*, **90**, 129.
Pang, Bijia, li Pen, Ue, Perrone, Michael 2010, *Magnetohydrodynamics on Heterogeneous Architectures*: *A Performance Comparison*.
Pressman, R. S. 1997, *Software Engineering* (*A Practitioners Approach*), McGraw Hill.
Roe, P. L. 1981, *Journal of Computational Physics*, **43**, 357.
Schive, H.-Y., Zhang, U.-H., Chiueh, T. 2011, *Directionally Unsplit Hydrodynamic Schemes with Hybrid MPI/OpenMP/GPU Parallelization in AMR*, ArXiv e-prints.
Scullion, E., Erdélyi, R., Fedun, V., Doyle, J. G. 2011, *AJ*, **743**, 14.
Sedov, Leonid I. 1946, *Journal of Applied Mathematics and Mechanics*, **10**, 241.
Selwa, M., Ofman, L., Murawski, K. 2007, *AJL*, **668**, L83.
Shelyag, S., Fedun, V., Erdélyi, R. 2008, *AA*, **486**, 655.
Sødergaard, Peter, Hansen, Jan Vittrup 1999, *Numerical Computation with Shocks*.
Stein, R. F., Nordlund, A. 1998, *AJ*, **499**, 914.
Taylor, G. 1950, *Proceedings of the Royal Society: A Mathematical Physical and Engineering Sciences*, **201**, 159.
Tóth, G. 1996, *AJL*, **34**, 245.
Tóth, G. 2000, *Journal of Computational Physics*, **161**, 605.

Toth, G., Keppens, R., Botchev, M. A. 1998, *AA*, **332**, 1159.

Vigeesh, G., Fedun, V., Hasan, S. S., Erdélyi, R. 2011, 3*D Simulations of Magnetohydrodynamic Waves in the Magnetized Solar Atmosphere*, ArXiv e-prints.

Wang, T., Ofman, L., Davila, J. M. 2013, *AJL*, **775**, L23.

Whitehead, N., Fit-florea, A. 2011, *Precision and Performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. NVIDIA white paper*, **21(10)**, 767.

Wienke, S., Springer, P., Terboven, C., an Mey, D. 2012, *Lecture Notes in Computer Science*, **7484**, 859.

Wolfe, M. 2008, *Compilers and More: A GPU and Accelerator Programming Model*.

Wong, H. C., Wong, U. H., Feng, X, Tang, Z. 2011, *Computer Physics Communications*, **182**, 2132.

Wong, U. H., Aoki, T., Wong, H. C. 2014a, *Computer Physics Communications*, **185**, 1901.

Wong, U. H., Wong, H. C., Ma, Y. 2014b, *Computer Physics Communications*, **185**, 144.

Zink, B. 2011, *HORIZON: Accelerated general relativistic magnetohydrodynamics*, ArXiv e-prints.