LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-664446

Parallel Block Structured Adaptive Mesh Refinement on Graphics Processing Units

D. A. Beckingsale, W. P. Gaudin, R. D. Hornung, B. T. Gunney, T. Gamblin, J. A. Herdman, S. A. Jarvis

November 18, 2014

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Parallel Block Structured Adaptive Mesh Refinement on GPUs

D. A. Beckingsale[a,*] W. P. Gaudin[a] R. D. Hornung[b]
B. T. Gunney[b] T. Gamblin[b] J. A. Herdman[a]
S. A. Jarvis[a]

[*]dab@dcs.warwick.ac.uk
[a]AWE, Aldermaston, UK
[b]Lawrence Livermore National Laboratory, Livermore, CA, USA

Abstract

Block-structured adaptive mesh refinement is a technique that can be used when solving partial differential equations to reduce the number of zones necessary to achieve the required accuracy in areas of interest. These areas (shock fronts, material interfaces, etc.) are recursively covered with finer mesh patches that are grouped into a hierarchy of refinement levels. Despite the potential for large savings in computational requirements and memory usage without a corresponding reduction in accuracy, AMR adds overhead in managing the mesh hierarchy, adding complex communication and data movement requirements to a simulation. In this paper, we describe the design and implementation of a *native* GPU-based AMR library, including: the classes used to manage data on a mesh patch, the routines used for transferring data between GPUs on different nodes, and the data-parallel operators developed to coarsen and refine mesh data. We validate the performance and accuracy of our implementation using three test problems and two architectures: an eight-node cluster, and over four thousand nodes of Oak Ridge National Laboratory's Titan supercomputer. Our GPU-based AMR hydrodynamics code performs up to 4.87× faster than the CPU-based implementation, and has been scaled to over four thousand GPUs using a combination of MPI and CUDA.

1 Introduction

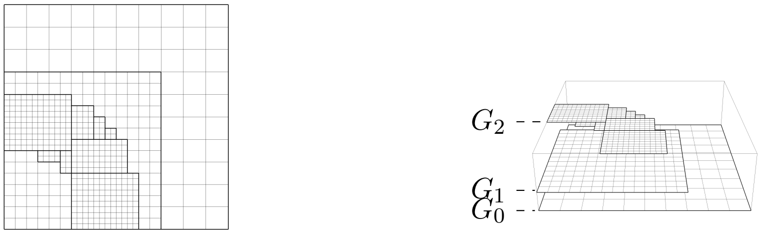
Block-structured adaptive mesh refinement (AMR) allows for fewer resources to be used to achieve the required accuracy in interesting areas of a problem [4, 5]. These areas of interest (shock fronts, material interfaces, etc.) are *refined*, and recursively covered with rectangular patches of computational mesh at a higher resolution. The patches are grouped into a hierarchy of levels of refinement that dynamically adapt throughout the computation as the areas of interest move. Despite the potential for large savings in resource usage with maintained accuracy, AMR requires dedicating a portion of application runtime to managing the mesh hierarchy; this requires complex data management and communication.

Most AMR applications run exclusively on the CPU, and those that do use GPUs often copy the necessary data between GPU and CPU memory at the beginning and end of every GPU-based routine [14, 16, 20]. We present a *native* implementation of AMR on GPUs. Building on the SAMRAI framework [2], we create classes that manage the life cycle of AMR patches where data is stored exclusively on the GPU. All routines that manage the patch hierarchy continue to be handled by SAMRAI on the CPU, but all AMR-specific routines that operate on patch data, such as the coarsening and refining of data between adjacent levels in the hierarchy, execute on the GPU.

Using the object-oriented interface of SAMRAI we develop a set of routines and data structures that allow patch-based data to reside on and be manipulated by the GPU. Using these extensions we write a GPU-based AMR hydrocode, CleverLeaf, that performs up to 4.87× faster than the CPU-based implementation on a single node, and has already been scaled to four thousand nodes using a combination of MPI and CUDA. We describe the design and implementation of our GPU-based AMR library extensions in this paper, including the classes used to manage patch data, the routines used for transferring data between GPUs on different nodes, and the data parallel operators developed to coarsen and refine mesh data. We also validate the accuracy of our implementation on three test problems, and present performance studies using up to 4096 NVIDIA K20x GPUs.

2 Background and Related Work

In this section we briefly describe the necessary background to understand the remainder of the paper, including a description of AMR and GPU computing; we also provide a thorough review of related work.



(a) Simple adaptive mesh, where each grid patch has a thick outline.

(b) Full hierarchy of three grid levels for the simple mesh.

Figure 1: Example adaptive mesh and corresponding grid hierarchy.

2.1 Adaptive Mesh Refinement

Adaptive mesh refinement uses a hierarchy of nested, logically rectangular grids over which the partial differential equation being solved is discretised. As in Berger et al. [4, 5], we provide a formal notation for our hierarchy in terms of these grids. The coarsest grid is the base grid, specified at the start of the computation and denoted G_0. It may be composed of several possibly overlapping patches. This base grid remains fixed throughout the simulation. Each component patch is denoted $G_{0,j}$, and thus G_0 is the union of its components $G_{0,j}$:

$$G_0 = \cup_j G_{0,j} \tag{1}$$

During the simulation, refined sub-grids of patches will be created in response to features in the solution. Sub-grids are not placed *in* the coarse grid, but on top of it. Each sub-grid is defined independently and has its own solution vector, and can be advanced almost independently of all other grids. These independent grids provide a natural method of domain decomposition allowing for easy parallelisation of the algorithm.

Fine sub-grids can contain finer sub-grids within their boundaries. Sub-grids are recursively generated to provide the necessary level of refinement, creating a hierarchy of grid levels. The coarse grid G_0 is at level 0 in the hierarchy. Sub-grids of G_0 are part of G_1 and are described as level 1 refinements. Refined grids within G_1 are at level 2. A nested sequence of sub-grids may be created to cover a portion of the domain. Figure 1 shows an example hierarchy containing three grid levels.

The mesh spacing, or resolution, h_l for each grid level l is normally specified in advance, where each h_l is an integer multiple of h_{l-1}. The relationship between the mesh spacing at each level is typically specified as the refinement ratio:

$$r_l = \frac{h_{l-1}}{h_l} \tag{2}$$

Grids at different levels of the hierarchy must be properly nested. A fine grid must start and end at the corner of a cell in the next coarser grid, and there must be at least one level $l-1$ cell separating a grid cell at level l from a cell at level $l-2$ in any direction unless the cell is at the physical boundary of the domain. The proper nesting requirement does not require a fine grid to be contained in only one coarse grid, so one fine grid may be nested in two or more coarser grids.

The AMR algorithm has three main components: (i) advancing the simulation using some finite difference scheme, (ii) error estimation and hierarchy generation, and (iii) inter-level operations such as solution projection and the filling of patch boundaries. These procedures are interleaved to correctly and conservatively advance the simulation on the adaptive hierarchy. When the simulation is initialised, the error estimation and hierarchy generation procedure must be used to generate the hierarchy, since only the coarsest level is specified by the user. Once the hierarchy is created, the main loop of the simulation proceeds as follows: first, the boundary conditions of each patch are filled; second, the simulation is advanced in time using the integration algorithm; third, the error estimation and hierarchy generation procedure is used to update the simulation grid.

Each patch will require some data to be placed in additional cells around the patch edge to provide boundary conditions for the system of partial differential equations. Boundary data for each patch can be filled in one of three ways: (i) with the physical boundary conditions, (ii) with the data from a neighbouring patch on the same level, or (iii) with the data from a neighbouring patch on the next coarsest level. When data is transferred between levels it must be interpolated to correctly fill the increased number of smaller cells on the finer level.

Since each patch is defined as an independent computational entity with its own solution storage, each patch can be integrated in time independently once its boundary values are supplied. This independence means that, using the patch as a basic unit of work in the simulation, work can be easily shared between multiple processes. The solution on a patch is modified in the case when a cell is covered by a fine grid, and the coarse cell value is replaced by a conservative average of the fine cell values that cover the coarse cell.

At the beginning of the simulation, and with a given frequency, an error estimation procedure is invoked to determine the structure of the patch hierarchy. When more than one level of patches exists, the procedure is applied recursively from the second finest to the coarsest level of the hierarchy. This regridding procedure has three steps: flagging, where a heuristic is applied to determine which level l cells ought to be covered by the level $l+1$ patches; clustering, where the new set of level l patches is created from a set of flagged cells on level $l-1$; and solution transfer, where data is copied from the old to the new hierarchy. Once the regridding procedure is completed, the next time step starts and the main algorithmic steps (boundary value determination, integration, and regridding) are repeated until the end of the simulation.

2.2 Programming Models for Graphics Processing Units

Programming for GPUs requires the use of a programming model such as CUDA, OpenCL, or OpenACC. For this work, we use NVIDIA's CUDA programming model, as it is the most mature and feature-rich model for programming NVIDIA hardware. GPU functions are written as *kernels* which are executed simultaneously in a single-instruction-multiple-data (SIMD) fashion on the device. A CUDA-capable GPU is a collection of stream multiprocessors (SMs), consisting of a number of stream processors (SPs) that share an instruction cache.

The CUDA programming model revolves around the concept of threads, blocks, and grids that execute on these hardware units. A thread executes on a single SP, and blocks are groups of threads that are mapped to SMs and will execute concurrently. A grid is a collection of thread blocks, typically dependent on the size of the data being maninpulated. The grid can be either one- or two-dimensional, and defines the total index space for the threads. These grids are used to map threads onto portions of the application domain. When a device kernel is launched, each thread runs one instance of the kernel. The co-ordinates of a thread can be accessed inside the kernel, allowing each thread to determine which elements of global data to process.

OpenCL uses a similar programming model to CUDA, with GPU functions being written as kernels that will be executed in parallel on a given device. The OpenACC model is different, having more in common with OpenMP. It relies on source code annotation using directives to mark regions of code for execution on the GPU. The use of CUDA in our work is an implementation detail, and the techniques we apply would map equally well to OpenCL and OpenACC.

2.3 Adaptive Mesh Refinement with Graphics Processing Units

Berger's adaptive mesh refinement algorithm was presented in 1984, and many computational physics codes have been ported to GPUs since the release of CUDA in 2007. However, there is little work where AMR codes have been ported to GPUs. We suppose that this is due to the large amount of data management required when updating the adaptive hierarchy, and the fact that naïve method for porting codes to GPUs revolves around repeatedly copying simulation data to and from the GPU across the slow PCI bus.

An early paper by Wang et al. describes an implementation of a compressible flow solver with AMR on GPUs [20]. At the beginning and end of the Runge-

Kutta kernel used to advance the solution, the required data must be copied from the CPU to the GPU. This basic implementation achieves a 10x speedup over a single CPU core, although with today's supercomputer nodes typically having at least 16 processor cores, this number is not high enough to make this method useful.

In [6] the authors briefly describe a forest-of-octrees based AMR algorithm for seismic wave propogation on GPUs. The implementation doesn't appear to be native, as although the text lacks sufficient details about the GPU-based implementation, the results presented include timings for transferring the mesh and initial data to the GPU from the CPU memory. Nevertheless, the parallel performance of the code is scalable on up to 256 GPUs.

Schive et al. introduce GAMER, an astrophysical simulation code with both AMR and GPU support [17]. Both the Eulerian hydrodynamics and self-gravity phases of the application are solved on the GPU, but the necessary data is stored in the CPU memory, and must be transferred to the GPU memory before the computational kernel is launched. The data transfer is performed concurrently with other computation, so the imapact is minimised, and the authors note that data transfer time typcially only takes 30% of the application runtime. The Uintah framework from the University of Utah is an AMR framework that supports GPUs [11, 14]. The focus in Uintah is on heterogeneous platforms, and as with GAMER, solution data must be copied between the CPU and GPU memory as required by the numerical kernels. These data transfers are overlapped with other work, but nevertheless, this is not a fully native framework.

Shamrock is an Eulerian hydrodynamics code with AMR, similar to CleverLeaf, that supports exectution on GPUs via OpenCL [9]. Only a small fraction of the necessary methods are ported to the GPU, and data is again copied between the CPU and GPU memory at the beginning and end of the four routines that are ported. As is to be expected, the performance of the GPU-based routines was better than the CPU-based equivalents when data transfer time was excluded. Using a simple performance model and the application of Amdahl's law, the authors predict an order of magnitude speed up if 95% of the application is ported to the GPU and data transfer only occurs at the start and end of the simulation.

The CLAMR application developed at Los Alamos National Laboratory is a cell-based AMR code that solves the shallow-water equations [15]. Implemented in OpenCL, the code *is* native; initial conditions are set on the CPU and then copied to the GPU memory at that start of the simulation, but data is not copied back to the CPU during the simulation timestep. The cell-based scheme is different to the block-structured approach described by Berger and used in our work.

The most promising application is presented in an unpublished work [16] which describes a native implementation of patch-based AMR application for solving the

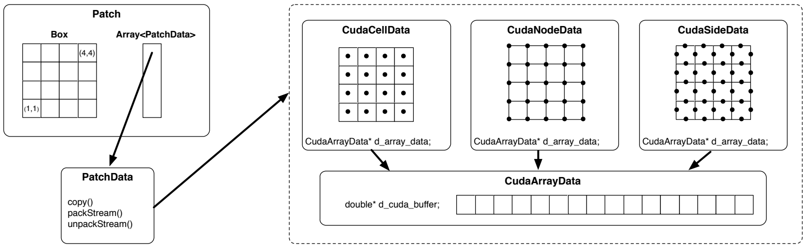


Figure 2: Class relationships for the CudaArrayData-based datatypes.

shallow-water equations. The authors take a similar approach to our library and ensure all computationally expensive parts of the AMR library are handled on the GPU, and they demonstrate performance improvements of up to $3.4\times$ compared to a uniform GPU-based implementation of the same algorithm. Despite the similarities to our work, the domain (shallow-water equations) is different, and this work remains unpublished.

To the best of our knowledge we have developed the only native GPU-based shock hydrodynamics code with AMR. Furthermore, by developing the necessary code as part of the SAMRAI library, we provide a collection of components that can be re-used in other block-structured AMR applications.

3 Design and Implementation

The SAMRAI library uses object-oriented design patterns to allow for easy interaction with user-supplied code [10]. Each of the basic structural units of the AMR hierarchy: patches, patch levels, and the patch hierarchy itself; are provided as fundamental software constructs by SAMRAI. The `Patch` class is a container for all the data living in a particular mesh region, and provides a way to access this data. All the data on a patch are handled using `PatchData` objects, each of which represents some simulation quantity on the mesh. The `PatchData` interface uses the Strategy design pattern [19], and defines a set of operations that an object must provide in order to be interoperable with SAMRAI's data management and communication routines.

Our GPU-based implementation is based upon a collection of `PatchData` classes for data defined on the zones, nodes, and edges of a patch. All these datatypes have something in common: an array stored in GPU memory defined over some arbitrary box in the problem space. We group all common functionality into a `CudaArrayData` class that manages this data, and specialise it for each data centering (see Figure 2). Each `PatchData` object will be defined to store data over some region of the simulation domain, as defined by the Patch that owns the object. Upon construction, the upper and lower indices of this region will be passed to the class, so that an appropriate amount of data storage can be allocated. In the case of the CudaCellData class, this is one array element per element. However, in the case of the CudaNodeData and CudaSideData classes we must allocate additional space to store all the node- and side-centred elements. Each class provides an accesor method for the pointer allocated in GPU memory. This pointer can be passed to any CUDA kernel in order to advance the simulation. The use of our classes in a real application is described in detail in a later section.

During an AMR simulation, boundary conditions can be filled in one of three ways: (i) using the physical boundary conditions; (ii) with data from a neighbouring patch on the same level; or (iii) with data from a patch on the next coarser level. Filling the boundary cells with the physical boundary conditions is handled by the application, and requires no additional features to be added to our library. When data must be transferred between patches, the copy routine is used. For two patches on the same level the copy routine is passed the two Patch objects and a Box that defines the area that needs to be filled. Since the data for both patches resides in GPU memory, we use a CUDA kernel that copies the data in the overlapping region from one patch to another. The kernel is launched using one thread per cell in the overlap region, so all data is copied in parallel.

If the two patches involved in the copy operation are located on different nodes the required data must be transferred using MPI. Supporting MPI is essential for any modern scientific code, and by including the necesary routines in our library we can run on multiple GPUs. So far, these routines have been shown to scale to up to 4096 GPUs. The PatchData interface described previously defines routines (packStream and unpackStream) that are used by SAMRAI to transfer both halo data and coarse or fine data between patches residing on different nodes. We provide CUDA kernels to pack data from the required region into a contiguous buffer in GPU memory. This buffer is then copied to the host memory and passed to SAMRAI, which handles the MPI communications. To unpack received data, the buffer is copied into the GPU memory and then unpacked in parallel using another CUDA kernel. Once the data has been transferred, a new PatchData object is created locally and the copy operators described previously can be used to fill the boundary cells on the receiving processor. We launch one CUDA thread per element to be packed into the buffer, ensuring the maximum amount of parellism is exposed. As an example, Figure 3 shows how the overlapping region is copied into the contiguous buffer in parallel.

A key part of AMR is transferring data between levels. Fine patches that border coarser patches require interpolated ghost data, and new fine patches will be filled

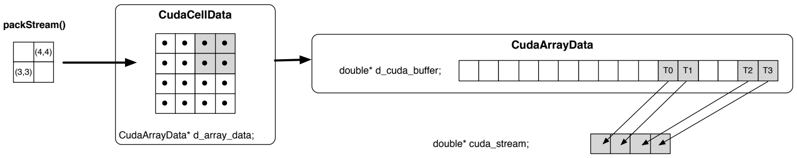


Figure 3: Parallel buffer packing for MPI operations.

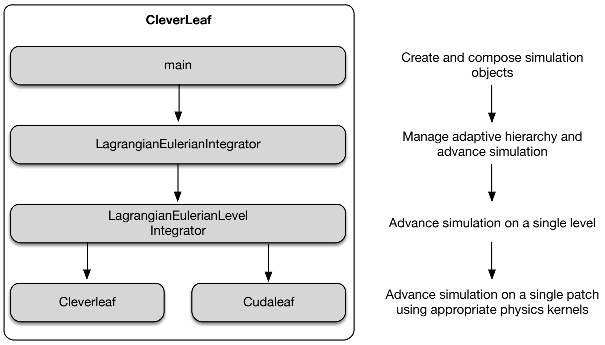

CleverLeaf
main
LagrangianEulerianIntegrator
LagrangianEulerianLevel Integrator
Cleverleaf
Cudaleaf
Create and compose simulation objects
Manage adaptive hierarchy and advance simulation
Advance simulation on a single level
Advance simulation on a single patch using appropriate physics kernels

Figure 4: Flexible CPU and GPU implementation of CleverLeaf.

with an interpolated version of the coarse solution. At the end of every timestep, fine solution data is coarsened onto the coarser levels of the hierarchy. We have developed refine operators that linearly interpolate zone-, node- and edge-centred data, and a coarsen operator that "inject" zone-centred data. All these operators run in parallel on a GPU. Operators that conservatively coarsen zone-centred data using volume-weighting and mass-weighting have also been developed as part of the CleverLeaf mini-application, and are described later in the paper.

4 CleverLeaf

The GPU-based SAMRAI extensions described so far have been used to port the CleverLeaf mini-app to GPUs. The original version of CleverLeaf is a CPU-based code, which extends the CloverLeaf mini-app by adding AMR [3]. CloverLeaf is a 2D explicit hydrodynamics mini-app that solves Euler's equations on a structured grid [8,12,13]. Both CloverLeaf and CleverLeaf are available for download as part of the Mantevo suite [1].

CleverLeaf uses a single class to control the integration of the numerical solution over patches. This class functions as a black box, and the remaining routines written to advance the simulation on the mesh hierarchy can remain un-

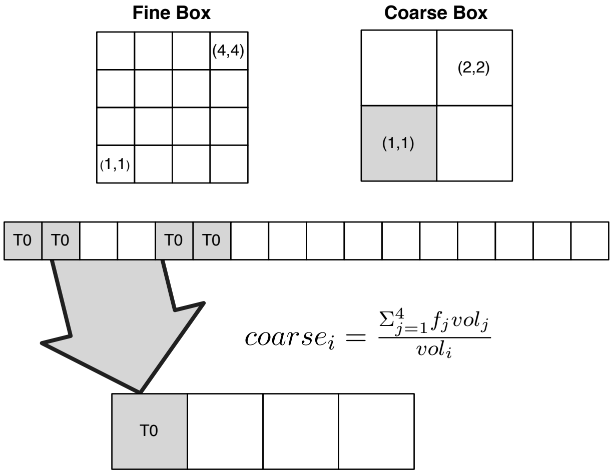


Figure 5: Data-parallel coarsen operators implemented in CUDA.

changed even when a different patch integrator is used. The similarity in the interfaces between the CPU-based `PatchData` classes and their GPU-based counterparts meant that we were able to easily modify the existing code. To port CleverLeaf to GPUs, we created a new patch integrator, and all references to CPU-based PatchData objects were replaced with GPU-based objects, and the data from these objects was passed to CUDA kernels rather than the existing numerical methods written in Fortran. Figure 4 shows how the two patch integrator classes are driven by the top level algorithm. The communications are handled by the `LagrangianEulerianLevelIntegrator` class and SAMRAI. The `PatchData` interface ensures that no additional changes are needed when using data allocated on the GPU.

Additionally, we have written mass-weighted and volume-weighted coarsen operators that run on the GPU; these are are required to conserve mass in the simulation. Each coarsen operator follows the same general pattern, with one CUDA thread being launched for every coarse value that needs to be filled. This thread then reads the relevant fine values and performs the necessary mathematical operations to calculate the coarse value. Figure 5 shows this operation for the volume-weighted coarsen.

The only other routine that requires data to be copied from GPU to host memory is the regridding phase, where the patch hierarchy is recreated to track moving features in the solution. Cells are flagged for refinement on the GPU using a data-parallel tagging kernel that evaluates the tagging heuristic for each cell on a patch

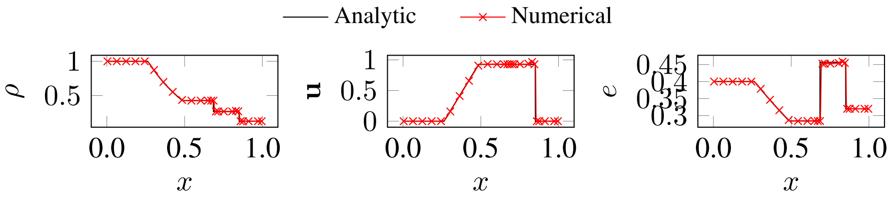


Figure 6: Plot of density, velocity, and energy at $t = 0.2$ for Sod's shock tube problem.

in parallel. This set of flagged cells is then copied to host memory and passed to SAMRAI to generate the new patch hierarchy. For improved performance, we compress the set of tags on the device from integers to bits, this reduces the amount of data that must be transferred by $\frac{1}{32}$. We also only copy back the array of tags for patches where some tag is set, avoiding redundant copies of arrays that will not affect the structure of the hierarchy.

5 Accuracy and Performance

In this section we present a detailed study of the accuracy and performance of CleverLeaf. Despite the main role of mini-apps being to investigate issues surrounding application performance, we present an accuracy study to reassure the reader that our GPU-based library functions correctly and is ready to be used in a production application.

5.1 Accuracy

To verify the accuracy of the results produced by CleverLeaf, we have used three test problems, comparing numerical results to exact, or converged, solutions. Whilst these test problems are somewhat contrived, the goal of CleverLeaf is not to simulate complex hydrodynamic systems, and instead to provide an accurate indicator into the performance of AMR and hydrodynamics algorithms of interest. Validating the accuracy of CleverLeaf allows us to be confident that the AMR and hydrodynamics routines are correctly implemented, and that they will represent more complex codes using similar algorithms.

5.1.1 Sod's Problem

The shock tube problem described by Sod [18] provides a good test of a code's ability to capture contact discontinuities, shocks, and the rarefaction profile. Consisting of two regions of fluid of different intial densities and pressures, the fluids are initially at rest, with the initial conditions being specified as follows:

$$\rho_l = 1 \qquad \rho_r = 0.125 \tag{3}$$

$$p_l = 1 \qquad p_r = 0.1 \tag{4}$$

The interface is at the point $x = 0.5$. At time $t > 0$ the two regions begin to interact, with a shock wave forming and travelling towards the right-hand boundary of the domain.

Figure 6 shows the density, velocity and energy profiles of the domain at time $t = 0.2$. CleverLeaf was run with 3 levels of AMR, a refinement ratio of 2, and a resolution of $\Delta x = 0.001$ at the finest level. The solution contains a small error at the contact discontinuity and in the rarefaction, however across the rest of the domain the solution is almost exact, and no oscillations are present. These small errors at the discontinuity are expected due to the numerical noise associated with modelling a discontinuity using a second-order method.

5.1.2 Interacting Blastwaves

Despite not having an analytic solution, Woodward and Collela's interacting blastwave problem [21] can be solved to mesh convergence. We can then compare the error at lower mesh resolutions to understand the behaviour of CleverLeaf as the number of refinement levels is increased.

The interacting blast wave problem consists of a two reflecting walls seperated by distance unity. The density $\rho = 1.0$ throughout the problem, and three regions of ideal gas with different initial pressures are used to create the strong shocks. The left region makes up the leftmost tenth of the volume; the right region, the rightmost tenth. The initial pressures in the left, middle and right regions of the domain are:

$$p_l = 1000 \qquad p_m = 0.001 \qquad p_r = 100 \tag{5}$$

We ran the Woodward-Colella problem with between one and six maximum levels of refinement, giving effective resolutions from 100 to 3,200 cells. We also ran the problem with a resolution of 50,000 cells in order to obtain a highly accurate solution with which to compare each of the resolutions. Figure 7 contains plots of the density, energy, and velocity of the system at time $t = 0.038$ for both the converged, and the one- and two- and four-level solutions. As the maximum number of levels is increased, the AMR solution converges towards the reference values for all three field quantities.

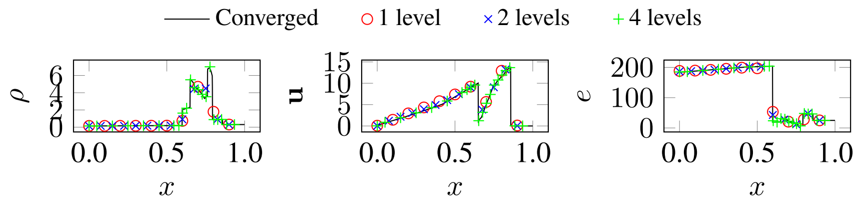


Figure 7: Plot of density, velocity, and energy at $t = 0.038$ for the Woodward-Colella interacting blastwaves problem.

	IPA	Titan
Processor	Intel Xeon E5-2670	AMD Opteron 6274
Clock	2.6 GHz	2.2 GHz
Accelerator	NVIDIA Tesla K20x	NVIDIA Tesla K20x
PCI gen		
Nodes	8	18,688
CPUs/node	2 × 8 cores	1 × 16 cores
GPUs/node	2	1
CPU RAM/node	128 Gb	32 Gb
GPU RAM/node	6 Gb	6 Gb
Interconnect	Mellanox FDR Infiniband	Cray Gemini
Compiler	Intel 13.1.163	Intel 13.1.3.192
MPI	MVAPICH 1.9	Cray MPT
CUDA Version	5.5	5.5

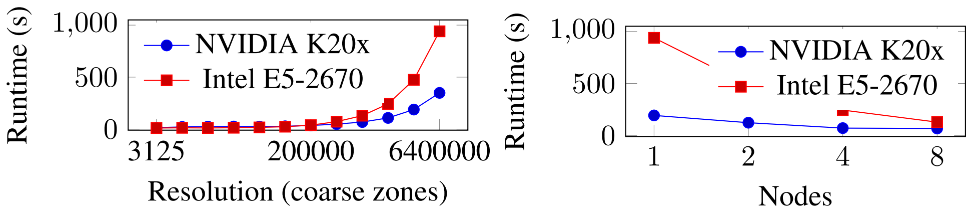
Table 1: IPA and Titan: hardware and software configurations.

5.2 Performance

To asses the performance and scalability of our implementation we performed a series of experiments using two different architectures: the IPA testbed machine at Lawrence Livermore National Laboratory and the Titan supercomputer at Oak Ridge National Laboratory. The hardware and software configuration of each platforms is detailed in Table 1. The experiments use a range of problem sizes and node counts, and are designed to test both serial performance and parallel scalability.

5.2.1 Serial Performance Analysis

Our first study compares a single NVIDIA Kepler K20x to one node (16 cores) of dual-socket Intel Xeon E5-2670 "Sandy Bridge" running at 2.6GHz. We use the Sod problem described previously and run 1000 timesteps at a range of coarse resolutions from 3 thousand to over 6 million zones, using 3 levels of refinement and a refinement ratio of 2. Figure 8(a) contains the results of this experiment. At



(a) Serial performance.

(b) Strong-scaling parallel performance.

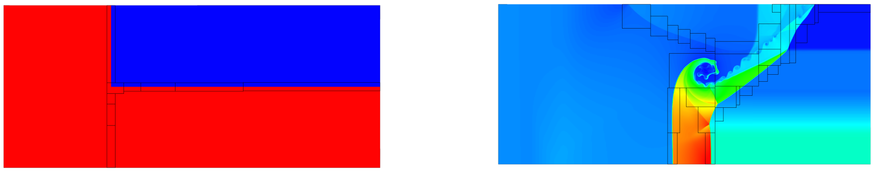
Figure 8: CPU vs. GPU performance comparison on up to 16 GPUS.

small problem sizes the GPU and CPU performance are similar, however, at large problem sizes, we see a performance improvement of over 2.6×. This performance improvement at larger problem sizes is typical of the throughput-oriented GPU architecture.

5.2.2 Parallel Performance Analysis

The second performance experiment investigates the scalability of our code as the number of GPUs is increased from 2 to 16 (1 to 8 nodes), we also include equivalent results for the CPU-based code. The experiment is a *strong-scaling* study, where the problem size remains constant as the number of GPUs (or nodes) is increased. We use the 6.4 million zone problem and run for 1000 timesteps. The results of this experiment are detailed in Figure 8(a), and for all node counts, the performance of the GPU-based code is better than the GPU-based code. For a single node, with two GPUs compared against two CPUs (16 cores), the GPUs are 4.87× faster. At eight nodes (16 GPUs vs. 128 cores) the GPU-based code is still 1.92× faster. We attribute this reduction in performance to the data transfer required during the boundary exchanges and the regridding phase beginning to dominate the simulation runtime; a consequence of running our experiment as a strong-scaling study and the effects of Amdahl's law. Since the parallel region of the code is so small, runtime is dominated by the serial fraction and as additional GPUs are added, the parallel region represents only a small portion of overall runtime compared to the serial regions of the code [**?**].

Our third experiment investigates the performance of our code at large scale, running on over four thousand GPUs on the Titan system at Oak Ridge National Laboratory. This experiment is a *weak-scaling* study, where the problem size is increased as the number of GPUs is increased. In theory, this means that each GPU will have a constant amount of work, and any costs associated with using an increasing number of nodes will be highlighted. We use a modified version of the triple point shock interaction probelm presented in [7]. A rectangular domain is



(a) ρ and finest level patches at $t = 0$.

(b) ρ and finest level patches at $t = 2.5$.

Figure 9: Triple point shock interaction: initial and final density and patch configuration.

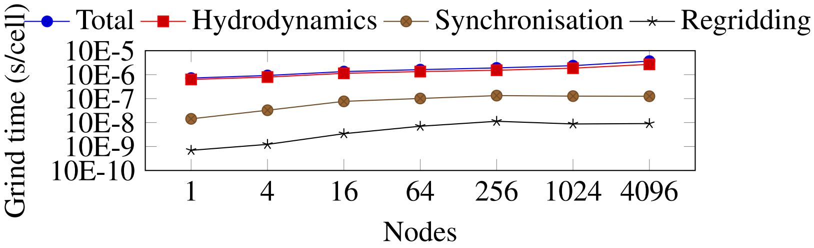


Figure 10: Weak-scaling performance analysis on Titan.

split into three regions, and as the simulation progresses from the initial state shown in Figure 9(a), a strong shock travels from left to right. This shock generates a large amount of vorticity and creates a complex area of interest, creating a large number of patches that are shown as black lines in Figure 9(b).

We run at seven different node counts, from 1 to 4,096; we use effective resolutions from 2 million to over 8 billion cells with 3 levels of refinement and a refinement ratio of 2. Weak scaling an AMR problem can be difficult since keeping the computational work per-GPU the same is difficult. In this experiment we increase only the coarse resolution and always run to the same physical end time regardless of the number of timesteps required. Figure 10 presents our results, normalised as average grind times per-cell for each node count. Each component of simulation runtime gradually increases as more nodes are added, however, we are able to run the problem on over four thousand nodes. It is also interesting to note that the majority of the simulation runtime is spent in the hydrodynamics of the application (including numerical kernels and halo exchanges). The AMR-specific runtime components, regridding and synchronisation, comprise only a fraction of the overall runtime.

6 Conclusions and Future Work

In this paper we have described our native GPU-based AMR package, and shown how it can be used in a hydrodynamics mini-application. Using the object-oriented design of the SAMRAI library we developed a set of classes that allocate and manipulate patch-based data on the GPU. Our implementation is *native*, with data residing in GPU memory at all times, and we provide the routines necessary for transferring data between GPUs on different nodes, and coarsening and refining data in parallel on the GPU. The novelty of this work lies in the fact that our implementation is *native*, and that we have developed the first fully data-parallel versions of a range of coarsen and refine operators. We validated the accuracy of our implementation, and compared the performance and scalability of our GPU-based code to the exsting CPU-based code. The GPU-based code is up to 4.87× faster than the CPU-based code, and we have demonstrated scalability on up to 4096 GPUs on the Titan system at Oak Ridge National Laboratory. In future work we plan to investigate ways to mitigate the performance impact of copying data between the GPU and host memory by overlapping data transfer and computation. We also plan to investigate mechanisms to allow patches to be "spill over" into CPU memory and then be transferred back to the device when necessary, this will allow larger problems to be solved and increase the relevance of our implementation to production codes.

Acknowledgements

This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy (LLNL-XXXXXXXXXXX). This work is supported by The Royal Society through their Industry Fellowship Scheme (IF090020/AM), and by the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Programme). This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S.B Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Mantevo - Home. http://mantevo.org.

[2] SAMRAI Overview. `http://computation.llnl.gov/casc/SAMRAI/`, May 2014.

[3] D. A. Beckingsale, O. F. J. Perks, W. P. Gaudin, J. A. Herdman, and S. A. Jarvis. Using Mini-Applications to Improve AMR Performance on Contemporary Compute Platforms. 2014. Submitted to *Journal of Parallel and Distributed Computing*.

[4] M. J. Berger and P. Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, May 1989.

[5] M. J. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53(3):484–512, Mar. 1984.

[6] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. C. Wilcox. Extreme-Scale AMR. In *Proceedings of the 22nd IEEE/ACM International Conference on Supercomputing*, pages 1–12, 2010.

[7] S. Galera, P.-H. Maire, and J. Breil. A two-dimensional unstructured cell-centered multi-material ALE scheme using VOF interface reconstruction. *Journal of Computational Physics*, 229(16):5755–5787, Aug. 2010.

[8] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpeCL and CUDA. In *Proceedings of the 24th IEEE/ACM International Conference on Supercomputing*, pages 465–471, Nov. 2012.

[9] J. A. Herdman, W. P. Gaudin, D. Turland, and S. D. Hammond. Benchmarking and Modelling of POWER7, Westmere, BG/P, and GPUs: An Industry Case Study. *SIGMETRICS Performance Evaluation Review*, 38(4):16–22, Mar. 2011.

[10] R. D. Hornung and S. R. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency and Computation: Practice & Experience*, 14(5):347–368, 2002.

[11] A. Humphrey, Q. Meng, M. Berzins, and T. Harman. Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system. In *XSEDE '12: Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*. ACM Request Permissions, July 2012.

[12] A. C. Mallinson, D. A. Beckingsale, W. P. Gaudin, J. A. Herdman, and S. A. Jarvis. Towards Portable Performance for Explicit Hydrodynamics Codes. In *Proceedings of the 1st International Workshop on OpenCL*, pages 1–12, Atlanta, GA, May 2013.

[13] A. C. Mallinson, D. A. Beckingsale, W. P. Gaudin, J. A. Herdman, J. M. Levesque, and S. A. Jarvis. CloverLeaf: Preparing Hydrodynamics Codes for Exascale. In *Proceedings of the Cray User Group*, Napa, CA, May 2013.

[14] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers. In *Proceedings of the th ACMIEEE International Conference on Supercomputing*, Nov. 2013.

[15] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey. Cell-Based Adaptive Mesh Refinement Implemented with General Purpose Graphics Processing Units. Technical Report LA-UR-11-07127, Los Alamos National Laboratory, Mar. 2012.

[16] M. L. Sætra, A. R. Bordtkorb, and K.-A. Lie. Efficient GPU-Implementation of Adaptive Mesh Refinement for the Shallow-Water Equations. Oct. 2013.

[17] H.-Y. Schive, Y.-C. Tsai, and T. Chiueh. GAMER: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 186(2):457–484, 2010.

[18] G. A. Sod. A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws. *Journal of Computational Physics*, 27(1):1–31, Apr. 1978.

[19] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[20] P. Wang, T. Abel, and R. Kaehler. Adaptive mesh fluid simulations on GPU. *New Astronomy*, 15(7):581–589, Oct. 2010.

[21] P. Woodward and P. Colella. The Numerical Simulation of Two-Dimensional Fluid Flow with Strong Shocks. *Journal of Computational Physics*, 54(1):115–173, Apr. 1984.