View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Warwick Research Archives Portal Repository

Original citation:
Beckingsale, David A., Gaudin, W. P., Herdman, J. A. and Jarvis, Stephen A. (2015) Resident block-structured adaptive mesh refinement on thousands of graphics processing units. In: 44th International Conference on Parallel Processing, Beijing, China, 1-4 Sep 2015 pp. 61-70.

Permanent WRAP URL:
http://wrap.warwick.ac.uk/90868

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Publisher's statement:
© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting /republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

A note on versions:
The version presented here may differ from the published version or, version of record, if you wish to cite this item you are advised to consult the publisher's version. Please see the 'permanent WRAP url' above for details on accessing the published version and note that access may require a subscription.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

warwick.ac.uk/lib-publications

Resident Block-Structured Adaptive Mesh Refinement on Thousands of Graphics Processing Units

David Beckingsale*, Wayne Gaudin†, Andy Herdman†, Stephen Jarvis*
*Department of Computer Science, University of Warwick, Coventry, UK
{dab,saj}@dcs.warwick.ac.uk
†High Performance Computing, AWE plc, Aldermaston, UK
{Wayne.Gaudin,Andy.Herdman}@awe.co.uk

***Abstract*—Block-structured adaptive mesh refinement (AMR) is a technique that can be used when solving partial differential equations to reduce the number of cells 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 resident 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 8 node cluster, and 4,196 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 is scalable on 4,196 K20x GPUs using a combination of MPI and CUDA.**

***Keywords*-adaptive mesh refinement; hydrodynamics; CUDA; mini-applications;**

I. Introduction

Block-structured adaptive mesh refinement (AMR) allows fewer resources to be used to achieve the required accuracy in interesting areas of a problem [1, 2]. 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 adapt throughout the computation as the areas of interest move. Despite the potential for large savings in resource usage without loss of accuracy, AMR requires dedicating a portion of application runtime to managing the mesh hierarchy; this requires complex data management and communication.

Massively parallel accelerator architectures like graphics processing unit (GPUs) can provide order of magnitude improvements in application performance [3–5]. With tremendous memory bandwidth and the ability to operate on hundreds of data items in parallel, these architectures provide the perfect platform for many high-performance computing applications providing they are ported well. These many-core architectures are the natural extension of the architectural trends introduced by multi-core processors, and consist of processors with even more cores, running at even lower frequencies. At the node level, it is now common to see an accelerator attached to at typical multi-core processor.

Most AMR applications run exclusively on the central processing unit (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 [4, 6, 7]. In this paper, we present the first *resident* implementation of block-structured AMR on GPUs, where all data is stored exclusively on the GPU. The SAMRAI library is a collection of software components for writing AMR codes [8], and has been to develop scalable CPU-based applications [9]. Building on SAMRAI, we create classes that manage the life cycle of AMR patches. 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. We use this library to write a GPU-based version of CleverLeaf, a hydrodynamics mini-application with AMR. Mini-applications are small, self-contained programs that embody the key performance characteristics of some key application [10], and provide the perfect vehicle for investigating new programming models, algorithms and architectures. The GPU-based version of CleverLeaf performs up to 4.87× faster than the CPU-based implementation on a single node, and has been scaled to over four thousand nodes using a combination of MPI and CUDA. In this paper, we make the following specific contributions:

- We describe the design and implementation of our GPU-based extensions to the SAMRAI library, 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.
- To the best of our knowledge we present the develop-

ment of the only resident GPU-based shock hydrodynamics application with AMR.
- The application, CleverLeaf, is designed as a proxy for shock hydrodynamics applications with AMR, and thus can be used to investigate the performance of GPU-based architectures for other large hydrodynamics codes.
- Furthermore, by developing the code as part of the SAMRAI library, we provide a collection of components that can be re-used in other block-structured AMR applications.
- Through performance analysis, we show that our GPU-based application performs up to $4.87\times$ faster than the CPU-based implementation, and present scalability results using up to 4,096 NVIDIA K20x GPUs.

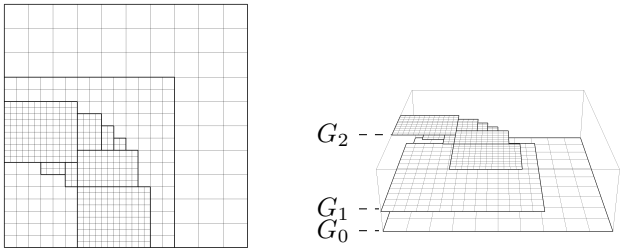
The remainder of this paper is structured as follows: Section II contains a description of the AMR technique and algorithm; Section III discusses related work; Section IV details the design and development of our GPU-based library and corresponding hydrodynamics code; Section V presents a performance analysis library and hydrodynamics code; and finally, Section VI concludes the paper and offers suggestions for future work.

II. Adaptive Mesh Refinement

Solving equations at higher resolution is more expensive both in terms of computational time and memory used. AMR is a computational technique where the resolution of the simulation is only increased in areas where it is necessary. For example, when simulating a tsunami travelling across the ocean the location of the wave is the most important feature in the solution. The rest of the ocean is much less interesting, and the impact of the wave is either negligible or easily approximated. An adaptive simulation would only simulate the area containing the wave at a high-resolution, saving both time and memory.

Developed by Berger et al. [1, 2], block-structured AMR has been successfully applied to many domains that display disparate physical scales, including cosmology, astrophysics, and shock hydrodynamics [11–13]. Other domains in which these disparate scales are observed include various military applications (small projectiles impacting much larger structures) and laser fusion experiments. Improving the performance of AMR will allow more of these important problems to be solved without increasing resource usage.

Block-structured adaptive mesh refinement uses a hierarchy of nested, logically rectangular grids over which the partial differential equation being solved is discretised. We briefly present a formal notation for the 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 patches, which remain fixed throughout the simulation. Each component patch is denoted $G_{0,j}$, and



(a) Simple adaptive mesh. Each grid patch has a thick outline.

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

Figure 1: Example adaptive mesh and the corresponding grid hierarchy.

thus G_0 is the union of its components $G_{0,j}$:

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

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, 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:

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

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 AMR algorithm we use 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.

III. Related Work

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 [1, 3, 14–17]. 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 the nave 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 [4]. 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 [18] the authors briefly describe a forest-of-octrees based AMR algorithm for seismic wave propagation on GPUs. The implementation doesn't appear to be resident, 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 [19]. 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 a computational kernel is launched. The data transfer is performed concurrently with other computation, so its impact is minimised, and the authors note that data transfer time typically only takes 30% of the application runtime.

The Uintah framework from the University of Utah is an AMR framework that supports GPUs [7, 20]. 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 resident framework.

The CLAMR application developed at Los Alamos National Laboratory is a cell-based AMR code that solves the shallow-water equations [21]. Implemented in OpenCL, the code is resident; 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 [6], which describes a resident implementation of patch-based AMR application for solving the 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 there is not a focus on large-scale parallel performance analysis.

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

IV. Design and Development

In this section we describe our GPU-based extensions to the SAMRAI library that allow AMR simulations to execute on accelerator-based node architectures. To test the library in the context of a real application, we extend the CleverLeaf mini-application using our newly developed library, and perform a performance analysis on over 4,000 GPUs. The development of library and the extensions to CleverLeaf are made easier by the adherence to the design patterns present in SAMRAI. We highlight the essential object-oriented abstractions that allow our GPU-based library to be fully compatible with existing SAMRAI code.

A. Programming Models

The design of the GPU-based extensions to SAMRAI are constrained by the design of accelerator-based nodes. Accelerators such as GPUs are devices specialised for fast floating point performance, that are attached to a CPU, but have their own memory space. Currently, the CPU and GPU communicate by transferring data across the PCI bus. This link between the two memory spaces is much slower than access to main memory, so a key design point is avoiding unnecessary transfer of data over this interface. Typically, the network interface will also be connected to the CPU, and when data must be transferred from the GPU across the network, it must first be copied to the CPU memory.

Programming for GPUs typically requires the use of a programming model such as CUDA or OpenCL [22, 23]. More recent developments in directive-based approaches like OpenACC and OpenMP 4 provide another way to execute code on an attached accelerator [24, 25]. For this work, we use NVIDIA's CUDA programming model. 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 manipulated. 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 use of CUDA in our work is an implementation detail, and the techniques we apply would map equally well to OpenCL. The OpenACC and OpenMP programming models rely on source code annotation to mark regions of code for execution on the GPU. These annotations are flexible and portable between different architectures, and hence tend to discourage explicit control of important parameters such as the number of threads launched. They also hide the low level control required to explicitly manage memory, a feature of CUDA (and OpenCL) that is essential in our library.

B. CudaPatchData

The SAMRAI library uses object-oriented design patterns to allow for easy interaction with user-supplied code [26]. 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 [27], and defines a set of operations that an object must provide in order to be interoperable with SAMRAI's data management and communication routines. We use this interface to develop a library capable of storing patch-based data in GPU memory whilst still using SAMRAI for mesh management, communication, and visualisation.

1) PatchData Interface: The `PatchData` interface defines the operations a class must provide to allow SAMRAI to gather data from the patch in order to transfer it to other patches in the hierarchy. The functions that the `PatchData` routines must perform include copying data from one `PatchData` object to another, packing the data corresponding to a given region of the patch into a buffer, and unpacking data from a buffer into a given patch region. These methods are the key points of the `PatchData` interface that we implement. By allowing an application to fully control data management, SAMRAI is easy to use in an existing application. In the case of our GPU-based extensions, the abstraction provided by the `PatchData` interface is at the perfect level to let us store simulation data in the GPU memory at all times and only copy data across the PCI bus when necessary. Figure 2 documents the full `PatchData` interface.

2) CudaPatchData Libray: The CudaPatchData library we have developed contains two packages: `pdat`, which contains three different `PatchData` implementations for

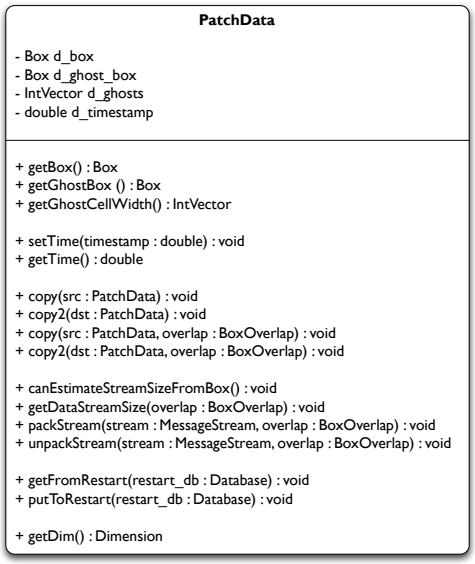


Figure 2: The SAMRAI PatchData inteface.

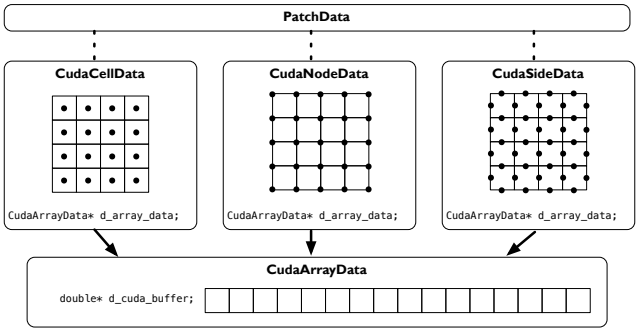


Figure 3: The SAMRAI CudaPatchData data-types.

managing data in GPU memory; and `geom`, which provides a collection of coarsen and refine routines that are essential when copying data between patches at different levels of the hierarchy.

The three different `PatchData` implementations are specialised for the three data-centrings required for the hydrodynamics scheme implemented in CleverLeaf. The common data store for each class is the `CudaArrayData` object. This class is responsible for allocating a contiguous array of data in GPU memory, corresponding to a given box size. This class also contains data-parallel routines to copy data, pack a region of the array into a buffer, and unpack a buffer into a region of the array. Each data-centring passes a slightly different `Box` to the `CudaArrayData` object it owns, ensuring the necessary data is stored. The three centrings required for CleverLeaf are: cell-centred, node-centred, and side-centred. Figure 3 shows the design of each class, as well as the data that each stores.

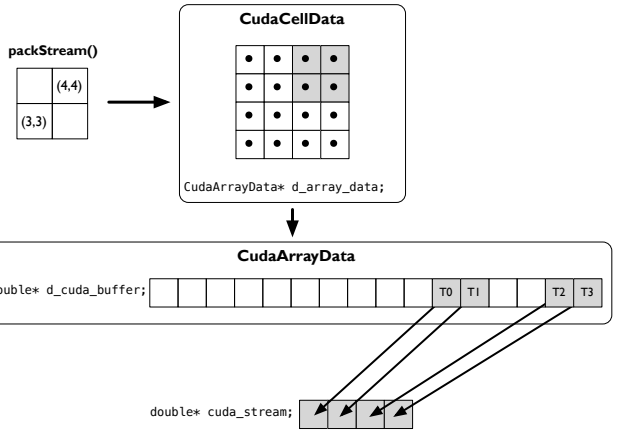


Figure 4: Data-parallel buffer packing for MPI operations.

During an AMR simulation, boundary conditions can be filled using the physical boundary conditions, with data from a neighbouring patch on the same level, or 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 two patches at the same level of refinement, a copy routine is used. If the two patches involved in the copy operation are located on different nodes the required data must be packed and transferred using MPI. Supporting large parallel architectures is essential for any modern scientific code, and by including the necessary routines in our library we can use MPI to run on multiple GPUs.

The data-parallel copy and packing operators use the same general design. Each operation will receive a `Box` as one of its parameters. This box describes the region of the patch that needs to be operated on. In all cases, the size of this box controls the number of CUDA threads that will be launched. Each thread will then be responsible for copying, packing, or unpacking one array element.

In the case of the pack and unpack methods, 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 parallelism is exposed. As an example, Figure 4 shows how the overlapping region is copied into the contiguous buffer in parallel.

The routines described by the `PatchData` interface are sufficient for transferring data between objects at the same

level of refinement. However, to transfer data between objects at different refinement levels, we must use a refinement operator or a coarsen operator. These operators interpolate the data to fill the differing number of cells on the receiving level. In SAMRAI, these operations are handled by two interfaces: `CoarsenOperator` and `RefineOperator`; that provide the necessary methods for coarsening or refining data. To allow the `CudaPatchData` classes to be used in an AMR simulation, we must provide operators to coarsen and refine data resident in GPU memory between different levels of the hierarchy.

The four operators we provide for coarsening and refining are fully data-parallel. As with the copy, pack, and unpack routines, each method executes using multiple CUDA threads. These are, to the best of our knowledge, the first data-parallel implementations for each of these operators. As an illustrative example, we consider linear interpolation for node-centred data. The code listing for our data-parallel algorithm is shown in Figure 5. In a typical implementation, data dependencies exist between temporary variables in different loop iterations and the algorithm is not immediately amenable to the data-parallel programming model of a GPU. Through substitutions and some operation re-ordering, we remove these dependencies and develop an algorithm that is fully data-parallel. When this kernel is launched to refine some region of data, one CUDA thread can be used per fine node, offering massive parallelism. We also provide conservative linear refine operators for the cell- and side-centred data, as well as a node-centred injection operator.

Together, the `pdat` and `geom` packages provide all the necessary components for a block-structured AMR simulation to be solved on a GPU, all that the user code must provide is a black-box integrator that can advance the simulation on a single patch.

C. Adding GPU Support to CleverLeaf

We have used the GPU-based SAMRAI extensions described so far 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. CloverLeaf is a 2D explicit hydrodynamics mini-app that solves Euler's equations on a structured grid [28–30]. Both CloverLeaf and CleverLeaf are available for download as part of the award-winning Mantevo suite [31].

CleverLeaf uses a single class to control the integration of the numerical solution on a patch. This class functions as a black box, and the remaining routines written to advance the simulation on the mesh hierarchy can remain unchanged even when a new programming model is used.

To develop the GPU-based version of CleverLeaf, we created a new patch integrator class that contains the code specific to advancing the solution on a single patch on a GPU. All references to the CPU-based `PatchData` objects provided by SAMRAI were replaced with references to the GPU-based `CudaPatchData` objects we have developed. We advance the simulation by passing a pointer to the data from these objects to CUDA kernels functions. Figure Figure 6 shows how the two patch integrator classes are driven by the top level algorithm.

Control of data communication and mesh

```
const int nblocks = (fine_box_size + BLOCK_SIZE - 1)/BLOCK_SIZE;

const int2 ratio = make_int2(ratio_vector[0], ratio_vector[1]);

cudaEvent_t fine_kernel;
cudaStreamSynchronize(coarse_stream);

cartcudanodelinrefine2d_kernel
  <<<nblocks, BLOCK_SIZE, 0, fine_stream>>>(
    coarse_data,
    coarse_data_offset,
    coarse_data_width,
    fine_data,
    fine_data_offset,
    fine_data_width,
    fine_box_width,
    fine_box_height,
    ratio);

cudaEventCreate(&fine_kernel);
cudaEventRecord(fine_kernel, fine_stream);
cudaStreamWaitEvent(coarse_stream, fine_kernel, 0);
```

(a) Host C++ code for launching the data-parallel linear refine kernel.

```
if (column < fine_box_width && row < fine_box_height) {

  const double realrat0 = 1.0/static_cast<double>(ratio.x);
  const double realrat1 = 1.0/static_cast<double>(ratio.y);

  const int ic0 = floor(column/static_cast<double>(ratio.x));
  const int ic1 = floor(row/static_cast<double>(ratio.y));

  const int ir0 = column - (ic0*ratio.x);
  const int ir1 = row - (ic1*ratio.y);

  const double x = ir0*realrat0;
  const double y = ir1*realrat1;

  const int coarse_index = (coarse_data_offset+ic0)
    + (ic1*coarse_data_width);

  arrayf[fine_index] = (arrayc[coarse_index]*(1.0-x)
      + arrayc[coarse_index+1]*x)
    *(1.0-y) + (arrayc[coarse_index+coarse_data_width]*(1.0-x)
        + arrayc[coarse_index+1+coarse_data_width]*x)*y;
}
```

(b) Data-parallel CUDA linear refine kernel for node-centred data.

Figure 5: Host and device code for data-parallel node-centred linear refine.

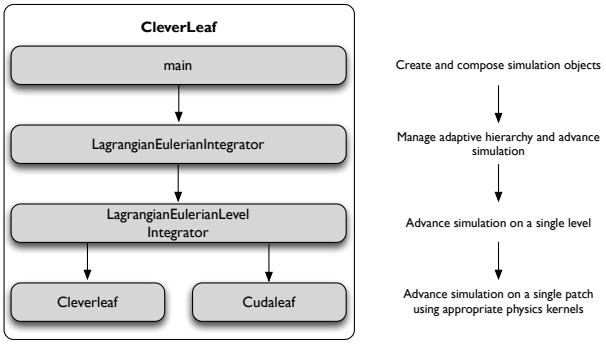


Figure 6: Flexible CPU and GPU implementation in CleverLeaf.

management continues to be provided by the `LagrangianEulerianIntegrator` and `LagrangianEulerianLevelIntegrator` classes using SAMRAI's various packages. This code is identical to the CPU-based version of the mini-app, and observing and implementing the `PatchData` interface meant that no additional changes were needed to allow CleverLeaf to run on GPUs.

To support adaptive simulation of Euler's equations on NVIDIA GPUs, we require three additional routines to allow data-parallel execution on GPU hardware. These routines are used to flag cells for refinement, and coarsen data between two levels in two specific ways: mass-weighted and volume-weighted.

Evaluating the tagging heuristic at each mesh cell is trivially parallel. Since the heuristic does not update any mesh data, and since each point can be calculated independent of any other, the routine can evaluate each point in the patch using a separate CUDA thread. However, once cells have been flagged for refinement, they must be transferred to the host memory to allow SAMRAI to construct the updated mesh hierarchy.

To transfer the data, we compress the array of tags (stored as `ints`) to an array of bits, where a 1 represents a flag, and 0 represents no flag. This compression minimises the amount of data that must be transferred, and is particularly important when a patch is large. Additionally, we store a `tagged` flag for each patch. If no cells in a patch are flagged for refinement then we don't copy data, since re-creating the appropriate data in the host memory is trivial.

Volume- and mass-weighted coarsen operations are essential in hydrodynamics simulations using AMR because they ensure that the quantities being simulated are conserved. To the best of our knowledge, we present the first implementation of these data-parallel operators. 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 7 shows this operation for the volume-weighted coarsen schematically, and the algorithm we use is presented in Figure 8.

Combining the `CudaPatchData` classes and the routines described in this section allows CleverLeaf to simulate Euler's equations natively in GPU memory. Simulation data is stored in global memory at all times, and the relevant regions of data are copied to the host memory in three situations: regridding, boundary updates, and synchronisation.

V. Performance Analysis

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

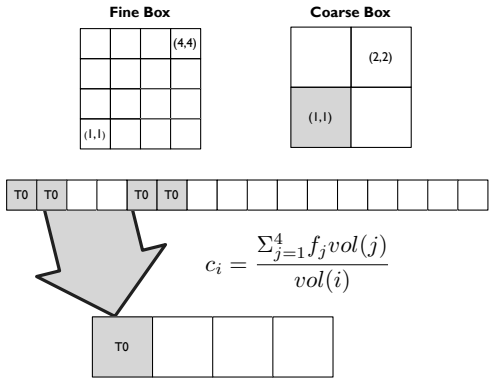


Figure 7: Data-parallel volume-weighted coarsen operator.

```
const int global_index = blockDim.x * blockIdx.x + threadIdx.x;

const int coarse_row = global_index / box_width;
const int coarse_column = global_index % box_width;

const int fine_row = coarse_row*row_ratio;
const int fine_column = coarse_column*column_ratio;

const int coarse_index = (coarse_data_offset + coarse_column)
  + (coarse_row * coarse_data_width);
const int fine_index = (fine_data_offset + fine_column)
  + (fine_row * fine_data_width);

if (coarse_row < box_height && coarse_column < box_width) {

  double spv = 0.0;

  for (int j = 0; j < row_ratio; j++) {
    for (int i = 0; i < column_ratio; i++) {
      const int current_fine_index = fine_index + i
        + (j * fine_data_width);
      spv += fine_data[current_fine_index]*Vf;
    }
  }

  coarse_data[coarse_index] = spv/Vc;
}
```

Figure 8: Code listing for the data-parallel volume-weighted coarsen kernel.

at Oak Ridge National Laboratory. The hardware and software configuration of each platform is detailed in Table I. The experiments use a range of problem sizes and node counts, and are designed to test both serial performance and parallel scalability.

	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 I: IPA and Titan: hardware and software configurations.

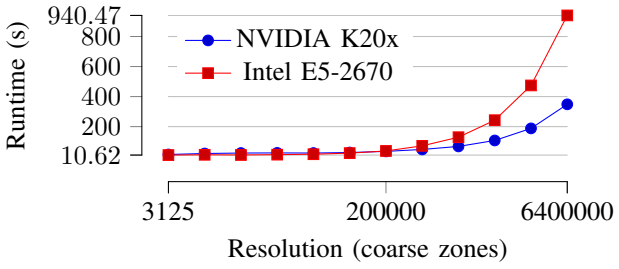


Figure 9: Serial performance.

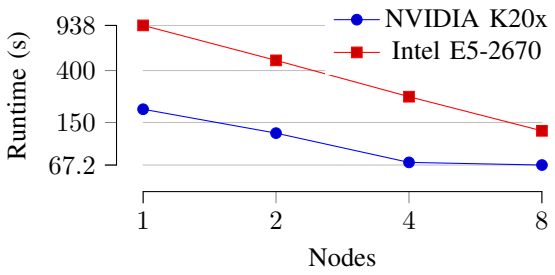


Figure 10: Strong-scaling parallel performance.

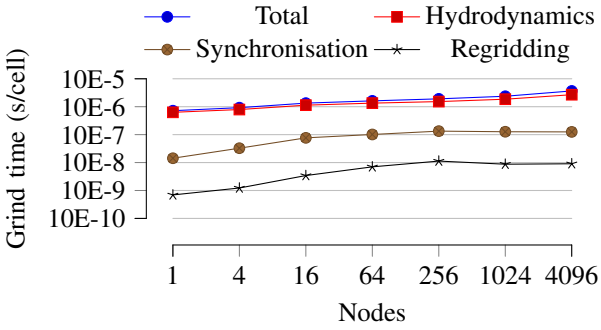


Figure 11: Weak-scaling performance analysis on Titan.

A. *Serial*

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 9 contains the results of this experiment. At small problem sizes the GPU and CPU performance are similar, and in all cases less than 200,000 cells the performance of the GPU is an average of $1.6\times$ slower than the CPU. However, at large problem sizes, we see a performance improvement of up to $2.67\times$. The average speedup of the GPU on problem sizes 200,000 cells and over is $1.99\times$ This performance improvement at larger problem sizes is typical of the throughput-oriented GPU architecture.

B. *Parallel*

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 10, 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 $4.87\times$ faster. At eight nodes (16 GPUs vs. 128 cores) the GPU-based code is still $1.92\times$ 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 [32].

Our third experiment investigates the performance of our code at large scale, running on over 4 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 problem presented in [33]. A rectangular domain is split into three regions, and as the simulation progresses from its initial state a strong shock travels from left to right. This shock generates a large amount of vorticity and creates a complex area of interest, with a large number of patches moving throughout the simulation domain.

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 11 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, com-

prise only a fraction of the overall runtime.

Specifically, at 4,096 nodes 44% of the runtime is spent advancing the simulation; this includes the hydrodynamics kernels and boundary exchanges. Calculating the timestep, which contains the only global reduction operation consumes 6% of the runtime. Synchronising fine data to the coarser levels takes an average of 3% of the runtime. In contrast, on a single node 59% of the runtime is spent advancing the simulation, with only 1% of time spent synchronising levels, and less than 1% calculating the global timestep. In both cases the time taken to fill boundaries remains roughly the same.

VI. Conclusions and Future Work

In this paper we have described our GPU-based AMR package, and shown how it can be used in a shock 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 *resident*, with data located 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 resident, and that we have developed the first fully data-parallel versions of a range of coarsen and refine operators. We have compared the performance and scalability of our GPU-based code to the existing CPU-based code. The GPU-based code is up to $4.87\times$ faster than the CPU-based code. Finally, 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 efficient use of the CPU and GPU memory simultaneously, such as allowing patches to be "spilled" into CPU memory and then be transferred back to the device when necessary. Using both CPU and GPU resources will allow larger problems to be solved and increase the relevance of our implementation to production codes.

Acknowledgment

This work is supported in part 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).

The research presented in this work has made use of the resources at Lawrence Livermore National Laboratory, which is supported by the U.S. Department of Energy under Contract No. DE-AC52-07NA27344, and the 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] M. J. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations," *Journal of Computational Physics*, vol. 53, no. 3, pp. 484–512, Mar. 1984.

[2] M. J. Berger and P. Colella, "Local Adaptive Mesh Refinement for Shock Hydrodynamics," *Journal of Computational Physics*, vol. 82, no. 1, pp. 64–84, May 1989.

[3] J. A. Anderson, E. Jankowski, T. L. Grubb, M. Engel, and S. C. Glotzer, "Journal of Computational Physics," *Journal of Computational Physics*, vol. 254, no. C, pp. 27–38, Dec. 2013.

[4] P. Wang, T. Abel, and R. Kaehler, "Adaptive mesh fluid simulations on GPU," *New Astronomy*, vol. 15, no. 7, pp. 581–589, Oct. 2010.

[5] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[6] M. L. Sætra, A. R. Bordtkorb, and K.-A. Lie, "Efficient GPU-Implementation of Adaptive Mesh Refinement for the Shallow-Water Equations," *Journal of Scientific Computing*, Jul. 2014.

[7] 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 24th ACM/IEEE International Conference on Supercomputing*, Nov. 2013.

[8] Lawrence Livermore National Laboratory. (2014, May) SAMRAI Overview. [Online]. Available: http://computation.llnl.gov/casc/SAMRAI/

[9] R. W. Anderson, N. S. Elliott, and R. B. Pember, "An arbitrary Lagrangian–Eulerian method with adaptive mesh refinement for the solution of the Euler equations," *Journal of Computational Physics*, vol. 199, no. 2, pp. 598–617, Sep. 2004.

[10] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, Sep. 2009.

[11] B. Fryxell, K. Olson, P. Ricker, F. Timmes, M. Zingale, D. Lamb, P. MacNeice, R. Rosner, J. Truran, and H. Tufo, "FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes," *The Astrophysical Journal Supplement Series*, vol. 131, pp. 273–334, Nov. 2000.

[12] J. Quirk, "A parallel adaptive grid algorithm for computational shock hydrodynamics," *Applied Numerical Mathematics*, vol. 20, no. 4, pp. 427–453, Apr. 1996.

[13] G. Bryan, "Fluids in the Universe: Adaptive Mesh Refinement in Cosmology," *Computing in Science & Engineering*, vol. 1, no. 2, pp. 46–53, Apr. 1999.

[14] C. Gheller, P. Wang, F. Vazza, and R. Teyssier, "Numerical cosmology on the GPU with Enzo and Ramses," *arXiv.org*, Dec. 2014.

[15] J. K. Holmen and D. L. Foster, "Accelerating Single Iteration Performance of CUDA-Based 3D Reaction–Diffusion Simulations," *International Journal of Parallel Programming*, May 2013.

[16] Z. H. Ma, H. Wang, and S. H. Pu, "GPU computing of compressible flow problems by a meshless method with space-filling curves," *Journal of Computational Physics*, vol. 263, no. C, pp. 113–135, Apr. 2014.

[17] B. P. Rybakin, L. I. Stamov, and E. V. Egorova, "Accelerated Solution of Problems of Combustion Gas Dynamics on GPUs," *Computers & Fluids*, pp. 1–18, Nov. 2013.

[18] 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*, Nov. 2010.

[19] 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*, vol. 186, no. 2, pp. 457–484, Feb. 2010.

[20] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, "Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system," in *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment*, Jul. 2012.

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

[22] NVIDIA Corporation. Parallel Programming and Computing Platform — CUDA — NVIDIA. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[23] The Khronos Group Inc. OpenCL - The open standard for parallel programming of heterogeneous systems. [Online]. Available: http://www.khronos.org/opencl/

[24] OpenACC.org. OpenACC Home — openacc.org. [Online]. Available: http://www.openacc.org

[25] OpenMP Architecture Review Board, "OpenMP Application Program Interface," Tech. Rep., Jul. 2013.

[26] R. D. Hornung and S. R. Kohn, "Managing application complexity in the SAMRAI object-oriented framework," *Concurrency and Computation: Practice & Experience*, vol. 14, no. 5, pp. 347–368, Apr. 2002.

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

[28] 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*, Nov. 2012, pp. 465–471.

[29] 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*, May 2013.

[30] 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*, May 2013.

[31] Mantevo.org. Mantevo - Home. [Online]. Available: http://mantevo.org

[32] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the AFIPS Spring Joint Computer Conference*, Apr. 1967, pp. 483–485.

[33] 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*, vol. 229, no. 16, pp. 5755–5787, Aug. 2010.