J. Parallel Distrib. Comput. 73 (2013) 866–875

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

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

Efficient asynchronous executions of AMR computations and visualization on a GPU system

Hari K. Raghavan *, Sathish S. Vadhiyar

Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore-560012, India

HIGHLIGHTS

- Asynchronous execution between CPU and GPU is improved.
- Redundant computations found in the default scheme are eliminated.
- The order of computations is changed to get the high priority result first.
- Visualization of the high priority data is overlapped with the computation of the remaining portions.

ARTICLE INFO

Article history:
Received 19 July 2012
Received in revised form
25 January 2013
Accepted 12 March 2013
Available online 19 March 2013

Keywords:
Adaptive Mesh Refinement
GPGPU
GAMER
CPU–GPU asynchronism

ABSTRACT

Adaptive Mesh Refinement is a method which dynamically varies the spatio-temporal resolution of localized mesh regions in numerical simulations, based on the strength of the solution features. In-situ visualization plays an important role for analyzing the time evolving characteristics of the domain structures. Continuous visualization of the output data for various timesteps results in a better study of the underlying domain and the model used for simulating the domain. In this paper, we develop strategies for continuous online visualization of time evolving data for AMR applications executed on GPUs. We reorder the meshes for computations on the GPU based on the users input related to the subdomain that he wants to visualize. This makes the data available for visualization at a faster rate. We then perform asynchronous executions of the visualization steps and fix-up operations on the CPUs while the GPU advances the solution. By performing experiments on Tesla S1070 and Fermi C2070 clusters, we found that our strategies result in 60% improvement in response time and 16% improvement in the rate of visualization of frames over the existing strategy of performing fix-ups and visualization at the end of the timesteps.

© 2013 Elsevier Inc. All rights reserved.

1. Introduction

The numerical solutions for many science and engineering applications are obtained by discretizing the partial differential equations used to model the problem. The computational domain is covered by a set of meshes (also referred to as grids or patches) over which numerical methods are applied. The accuracy of the numerical method depends on the granularity of the mesh. In many applications, features of interest are found in localized portions of the domain. Adaptive Mesh Refinement (AMR) [1] is a method which dynamically varies the spatio-temporal resolution of mesh regions based on the strength of the solution feature in the region.

There exist many variants of the AMR implementation based on factors such as mesh characteristics (structured [17,11] and unstructured [13,7]) and models of refinement (block based [17,11,12] and cell based [5]). Some block structured AMR libraries like PARAMESH [11] and FLASH [4] use uniform patches in which all patches have the same number of cells, while others including SAMRAI [17], Enzo [16], etc., cover the regions of interests with patches of non-uniform sizes, where the patch configuration can vary arbitrarily. Several works have dealt with parallelization of AMR for distributed multi-CPU and shared memory systems [3,2,17,7]. The basic parallelization strategy involves decomposing the hierarchy of meshes and distributing the work among the worker units (cores or nodes of the cluster). Each unit computes the solution for the subdomain allotted to it while periodically synchronizing with other nodes to obtain boundary data [8].

There are very few works dealing with AMR on general purpose graphics processing units (GPGPUs) [15,16,6]. GAMER (GPUaccelerated Adaptive MEsh Refinement code) [15] is one of the first implementations of AMR astrophysics codes for GPUs. GAMER follows a block-structured octree model of refinement in which every level is composed of a set of mesh units called patches. GAMER provides CUDA based kernel implementations

* Corresponding author.
E-mail addresses: hari.k.raghavan@gmail.com, hari@ssl.serc.iisc.in (H.K. Raghavan), vss@serc.iisc.ernet.in (S.S. Vadhiyar).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.03.002

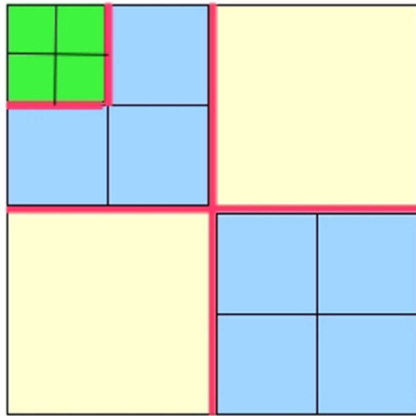


Fig. 1. AMR patch hierarchy with three levels of refinement. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

of 3D hydrodynamics and Poisson solvers. The GPU kernel advances the solutions while the CPU handles the AMR control functions and provides the input for the GPU solver. The GAMER execution scheme involves preparing patch data for the GPU solver, asynchronously solving the data on the GPU and storing the resulting data back in the CPU data structure. The stored solution is used to correct coarser patches by fix-up operations and to decide whether a level needs to be refined further. GAMER utilizes a hybrid OpenMP–CUDA parallelism strategy for efficient heterogeneous executions of AMR applications. It also uses CUDA streams to overlap memory transfers to the GPU with kernel executions.

In-situ visualization plays an important role for analyzing the time evolving characteristics of the domain structures. Continuous visualization of the output data for various timesteps results in a better study of the underlying domain and the model used for simulating the domain. While GAMER provides for asynchronous executions of phases of AMR on CPUs and GPUs, it performs fix-up operations after all the patches are solved by the GPUs. Consequently, it supports visualization only at the end of the timesteps. In this paper, we develop strategies for continuous visualization of time evolving data for AMR applications. We first reorder the computations for meshes on the GPU based on the user's input related to the subdomain that he wants to visualize thereby solving the subdomain required for visualization with high priority. We accommodate the fix-up operations and the visualization steps during the times CPU cores are idle rather than waiting for the timestep to complete. By performing experiments on Tesla S1070 and Fermi C2070 clusters, we found that our strategies result in up to 60% improvement in response time and 16% improvement in the rate of visualization of frames over the strategy of performing fix-ups and visualization at the end of the timesteps.

Section 2 describes the fundamental AMR and GAMER operations and defines the terminologies used. Section 3 presents our strategies to overcome the performance bottlenecks for fast online visualization. Section 4 describes the experimental setup and the various modes of operation and presents the results, along with salient observations. Section 8 presents conclusions and future work.

2. Background

The AMR approach considered in this work is the block-structured AMR developed by Berger and Oliger [1]. Fig. 1 shows a 2D AMR hierarchy with three levels of refinement. The red borders indicate the coarse–fine boundaries. A brief summary of the block-structured AMR algorithm is as follows [12].

The simulation proceeds in discrete units called *timesteps*. Every application of the numerical method on a mesh advances the simulation time by an amount. Starting from the coarsest level, the solution values are advanced in time using the numerical difference method. Once all the levels have been advanced, an operation referred to as *fix-up* is done. The fix-up operation consists of two sub-operations:

1. *Restrict*: The more accurate solution values from the finer mesh cells are injected into the underlying coarse mesh cells. The fix-up operation replaces the value of the coarse mesh cell by the average of the overlying fine mesh cells.
2. *Flux correction*: The finer level fluxes are used to adjust the solution values of coarse cells at the coarse–fine boundaries.

The pseudo code for the algorithm can be represented as follows:

```
Advance (level lv)
do
 Numerical_Integration (lv )
 Advance (lv+1 )
 fix-up (lv <- lv+1 )
 Refine (lv )
done
```

GAMER follows the octree model of refinement and restricts every patch to have the same number of cells (set at 8^3). This restriction avoids synchronization overheads since all patches have the same workload. Since patches are always formed in groups of 8, every patch belongs to a *patch group* which consists of itself and its 7 siblings. The advance step for a level in GAMER consists of 3 phases:

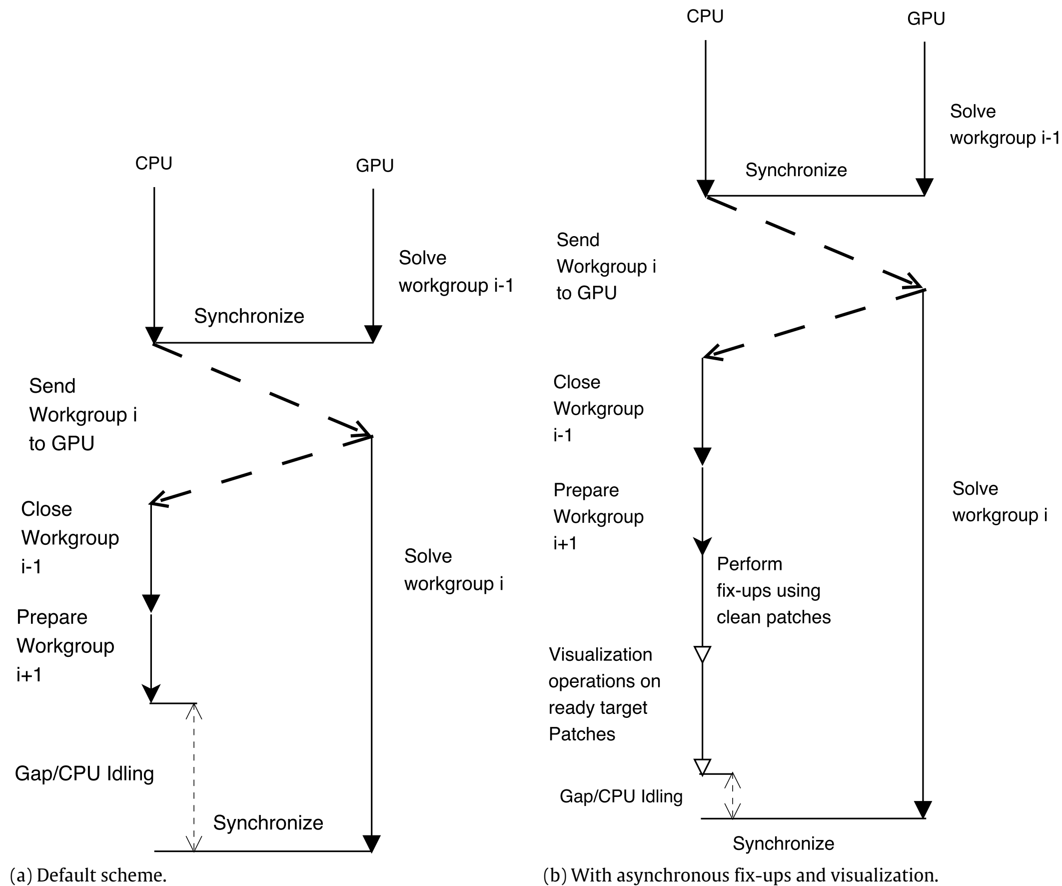
1. *Preparation*: the array containing input data for the GPU solver is filled from the patch data structure. The number of patch groups advanced simultaneously is termed as the *work group*. The prepare step is performed on the CPU.
2. *Solve*: the input array containing the prepared work group is solved asynchronously on the GPU and the solution is stored in the output array and copied back to the CPU.
3. *Closing*: the solved values from the output array are stored in the corresponding patch data structure. This step also stores the computed flux values along the coarse–fine boundaries. This step is performed on the CPU.

GAMER takes advantage of the asynchronous nature of the GPU kernel to overlap the solve step of a particular work group on the GPU along with the preparation of the next work group and the closing of the previous work group from the output array. This is illustrated in Fig. 2(a). GAMER provides scripts for enabling the visualization of the solution output using standard tools like Gnuplot and VisIt [10], a popular and efficient software used for scientific visualization. We assume that a set of subdomains are specified a priori for visualization (*target regions*). The user can select various 2D slices from the target region for visualization. The data from the leaf patches overlaying the target regions are output at the end of every timestep and are sent to a visualization server for processing.

3. Methodology

To our knowledge, GAMER is an established software for AMR execution on GPUs involving asynchronous operations on CPU and GPU cores. However, the execution flow in GAMER results in performance bottlenecks for fast online visualization.

1. Application execution with GAMER on a GPU system results in large idling times on CPUs. The time taken for the solve step on



(a) Default scheme.

(b) With asynchronous fix-ups and visualization.

Fig. 2. Workflow between the CPU and GPU.

the GPU is up to 1.7 times the combined total of preparation and closing step. This leads to the CPU being idle while the GPU is executing. We refer to this CPU idle time as *gap*. This is a potential source of asynchronism.
2. The fix-up operation is done when all levels have been advanced in a timestep. This results in the serialization between the level advances and fix-ups. We explore to see if it is possible to overlap some fix-up operations along with the computations on the GPU.
3. The restrict operation completely replaces the contents of a patch using the data from the child patch group. We observe that the patch needs to be computed only if it is adjacent to a coarser patch, in order to correct the flux values of the neighboring coarse patch.
4. In the default scheme, the visualization is done after the entire hierarchy is advanced for the timestep, even if the targeted regions for visualization correspond to only a subset of all the patches. Visualization can be performed at a faster rate if the fix-up and visualization steps are performed in the gaps.

We propose the following optimizations:

1. Asynchronous fix-up operations are carried out in the CPU–GPU gap. This is done by identifying patches that can be used to fix their parent patches once they are solved and closed.
2. Patch groups whose values will be completely overwritten during the fix-up phase are identified and are skipped from being solved. These patch groups are directly updated during their fix-ups.
3. The computations of the patches are reordered so that the patches corresponding to the region of the problem domain targeted for visualization (as specified in the user input) are computed at the earliest so that visualization of these patches can commence as early as possible.
4. Visualization of the patches are also accommodated in the CPU–GPU gaps, instead of at the end of the timesteps, to increase the throughput for visualization.

The default and the optimized schemes are illustrated in Fig. 2(a) and (b) respectively.

3.1. Asynchronous fix-ups

The fix-up operation proceeds from the finest level to the coarsest to ensure that the most accurate value is used to correct the coarser patch. We observe that unrefined patches which are not part of coarse–fine boundaries are ready to fix-up their parent patch for the timestep immediately after they are solved and closed. By definition, these patches do not have children and do not have neighbor patches of finer levels. Hence, these patches do not have to be fixed by the restrict operation or the coarse–fine flux correction operation. We term these as clean patches. These patches represent a subset of the leaves of the AMR octree, i.e., leaf patches that do not border finer patches. We note that the entire set of patches in the finest level are eligible for asynchronous fix-up. Once a patch group is closed, the constituting patches which are clean can fix their parent patch. If the entire patch group is clean (as in the case of the finest level), this leads to a complete fix-up. Otherwise, it would result in a partial fix-up with the rest of the fix-up of the parent patch hierarchy at the end of the timestep or when the remaining children of the patch become clean.

We estimate the maximum number of fix-ups that can be accommodated in a gap by measuring the time taken for the fix-up operations and determining the approximate gap available. While the GPU solves a work group, an appropriate number of clean patches are selected based on the determined estimates and these

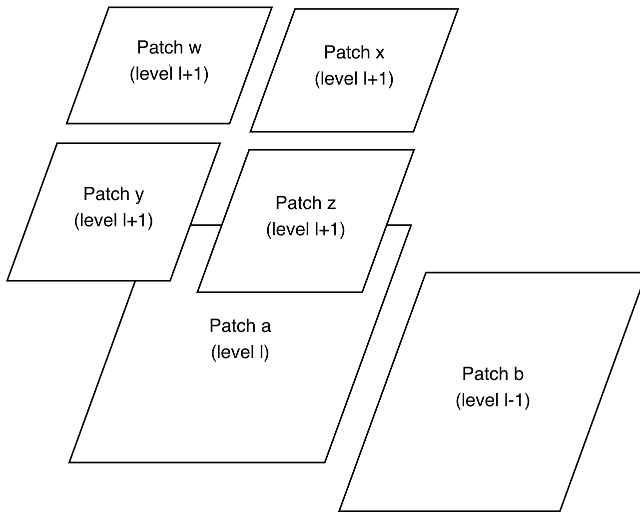


Fig. 3. Dependencies for completely refined patch groups.

patches are fixed. We found using experiments that the total gap in a timestep is sufficient to accommodate the fix-ups for all the clean patches. The concurrent fix-up operation also leads to previously unclean patches to become clean after their fix-up. Such patches are also considered to fix-up their parent patches during the gaps.

3.2. Pruning patch groups for reducing computations

Since the basic unit of execution on the GPU is a patch group, we identify patch groups which can be pruned from the computation list. A patch group which is completely refined (i.e., all 8 patches of the patch group are refined) will have its values completely replaced by the average of child patch group values during the restrict operation. The computation of such a patch group becomes necessary only when it is part of a coarse–fine boundary, in which case the flux values from its solution are needed to correct the adjoining coarse patch. Otherwise, such patch groups need not be computed and can have their values updated straightaway during their fix-ups. We term these as *avoidable* patch groups.

The aforementioned dependency issue is illustrated in Fig. 3. Patch b is an unrefined patch belonging to level $l-1$. Patch a is a neighboring patch of refinement level l. Thus, patch b is a part of a coarse–fine boundary. Patch a is completely refined and its children are patches w, x, y, z of refinement level $l+1$. Patch b must be corrected by the flux values computed by patch a through the coarse–fine boundary fix-up operation. Due to this, patch a must be computed even though its values will be overwritten by w, x, y, z by the restrict operation. For AMR implementations where the refinement procedure ensures that neighboring patches do not differ by more than one level, this dependency does not arise. GAMER imposes this constraint and thus all the completely refined patch groups are avoidable.

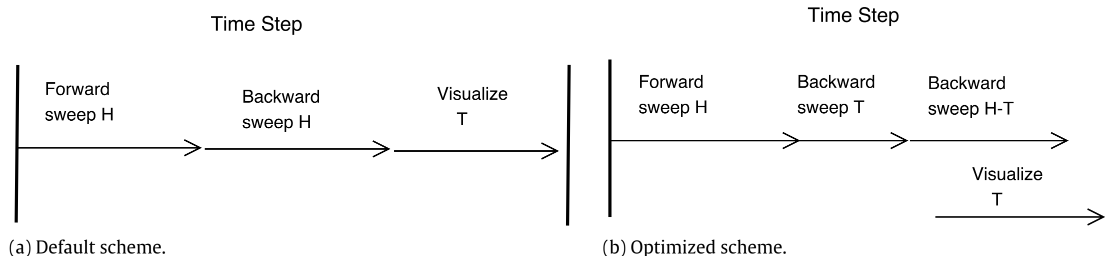
For a given patch hierarchy, we build an index list of the patch groups that need to be solved at every refinement level. When a level is advanced, the patch groups from the list are solved, omitting the avoidable patch groups. The fix-up operations are done for all the patch groups, by which the values of the avoidable patch groups are updated. In the hydrodynamics application, the amount of avoidable patch groups is significant. The pruning of such patch groups results in at least 10% improvement in the overall execution time.

3.3. Reordered execution for visualization

Based on the a priori specification of the region targeted for visualization, the sub-hierarchy of patches encompassing the target regions is identified. Every patch stores the information about its coordinates in the computational domain. The patches of all levels whose ranges overlap with the targeted regions are identified and added to a computation list. Thus, the patch hierarchy **H** is decomposed into 2 segments: **T** corresponding to the target regions and **H**–**T** which corresponds to the rest of the patch hierarchy. The partitioning procedure needs to be done every time refinements occur and the hierarchy changes.

In our application, we consider the second order accurate total variation diminishing (RTVD) hydrodynamics solver of GAMER [14]. The governing 3D equations are solved by first applying a forward sweep in the order *xyz*, and subsequently a backward sweep in the order *zyx*. The solutions from the forward sweep are required for the backward sweep. In the default scheme, the hierarchy **H** is solved once by forward sweep and then again by backward sweep. In the proposed method, **H** is solved in forward sweep like in the normal scheme. For the backward sweep, sub-hierarchy **T** is solved first, followed by **H**–**T**. Once **T** has been solved and the data from the GPU has been transferred to CPU, the visualization process can commence since the required data has been made available. The target region data is output to disk and is formatted for visualization concurrently while **H**–**T** is being advanced. The gaps during the execution of **H**–**T** are utilized for visualization along with its fix-ups operations. This is illustrated in Fig. 4.

One issue that arises from the partitioning approach is the problem of fragmentation in GPU execution. The GPU processes patch groups in batches and when the number of patch groups being solved is less than one work group, it leads to the GPU threads being idle. This ends up slowing the overall execution. For example, let the optimal GPU workload be W and suppose the target hierarchy has n levels, each with workloads $w_0, w_1, \ldots, w_n$, each of which are lesser than W. Executing the hierarchy purely on a level by level approach would not be optimal since the GPU is not utilized appropriately at any level. We note that in the shared timestep mode, the patches of various levels can be advanced in any order. This is because the hierarchy stays constant throughout the timestep. Hence, the problem of fragmentation is handled by pooling all the patch groups in the increasing order of their levels and selecting the ideal number of patch groups to be solved (240



(a) Default scheme. (b) Optimized scheme.

Fig. 4. Schemes for continuous visualization in GAMER.

Table 1
Split up of the execution time (seconds) for target volume of 288 patches.

Forward sweep (t_f)	Backward sweep (t_b)	Fixup (t_{fix})	Output (t_{out})	Format (t_{format})	Visualization (t_{vis})	Communication (t_{comm})
4.180	4.85	0.819	0.31	0.12	0.62	0.234

for Tesla and 112 for Fermi). The ideal numbers are determined such that all the GPU thread blocks are utilized in a work-efficient manner. At every phase, one work group is selected from the pool (possibly consisting of patch groups from multiple levels) and the constituent patch groups are prepared accordingly based on their level. The selected work group is solved and the patch groups are closed appropriately.

3.4. Visualization steps

As mentioned above, visualization for the target region is performed asynchronously on the CPU while the other regions are advanced on the GPU cores. The primary steps related to visualization are as follows:

1. Data output: the data from the leaf patches corresponding to the target region is output in a binary file format. The time taken for the output is denoted as t_{out}.
2. Formatting: the binary file data is converted into a standard data format known as SILO [9], which is used to store 2D and 3D mesh data. The time corresponding to this step is denoted as t_{format}.
3. Communication: if the visualization server is remotely located, the formatted data must be transferred over a network. The corresponding time is denoted as t_{comm}.
4. Visualization: the formatted data is visualized using VisIt [10]. The visualization time is denoted as t_{vis}.

Thus, the total time for steps of the visualization is the summation, $t_{out} + t_{comm} + t_{format} + t_{vis}$.

4. Experiments and results

4.1. Experiment setup

We ran the GAMER hydrodynamics code on a Tesla cluster, containing one Tesla S1070 GPU and 8 AMD Opteron 8378 cores with 8 OpenMP threads, and a Fermi cluster, containing one Tesla C2070 GPU and 4 Intel Xeon W3550 cores with 4 OpenMP threads, in the shared timestep mode. In this mode, every level is advanced by the same timestep value. We first show the results on the Tesla cluster. Let t_0^f, t_0^b represent the default forward and backward sweep times and t_1^f, t_1^b represent the forward and backward sweep times after the proposed optimizations are applied. t_v represents the visualization time. The following metrics are adopted to study the effectiveness of the proposed scheme:

1. Improvement in response time: response time is defined as the time elapsed since the start of the timestep till the point when the data is visualized. Original response time $= t_0^f(\mathbf{H}) + t_0^b(\mathbf{H}) + t_v$.
   Optimized response time $= t_1^f(\mathbf{H}) + t_1^b(\mathbf{T}) + t_v$.
2. Utilization of the gaps: this denotes the percentage utilization of the gaps in the CPU and hence gives a measure of amount of asynchronous computations on the CPU and GPU. We consider only operations that utilize the CPU cores and hence exclude network communication time even though it proceeds in parallel. We use asynchronous MPI calls, and hence the communication proceeds independent of the CPU activity.
3. Frequency of visualization update: this measures the time between two successive visualizations corresponding to two consecutive timesteps.

We apply the above metrics to two visualization modes: visualizing the entire hierarchy corresponding to the regions targeted and visualizing only the base level data of the targeted regions.

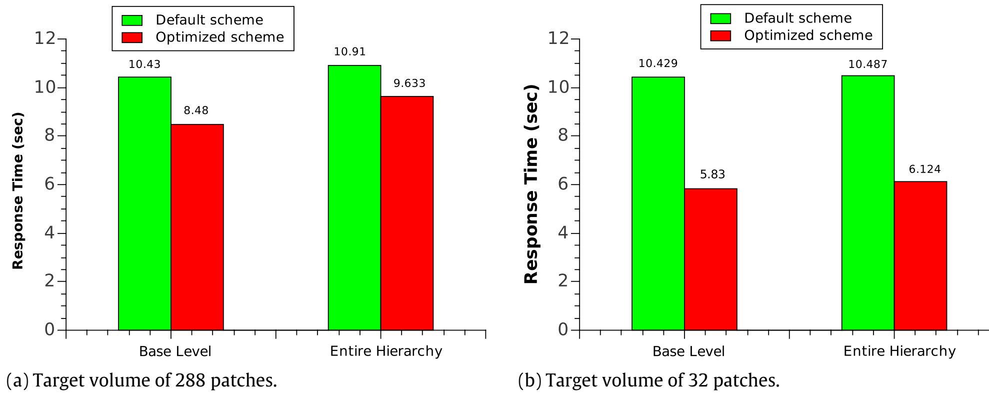
We explored the visualization process in different settings: *on-site* and *remote-site* visualization. For remote-site visualization, data has to be transferred over a network to the remote visualization server. We simulated different network configurations with bandwidths of 1 Gbps, 80 Mbps, and 700 kbps. We refer to these as *intra-department*, *intra-country* and *cross-continent* configurations, respectively. The distributed architecture of VisIt provides an efficient model for remote visualization where the data from the local site need not be completely transferred to the visualization server. In our application, we enable the visualization of various 2D slices from the targeted region, where only the data required by the query is transferred to the remote server. In our experiments, we deal with transactions up to a maximum of 30 MB per query.

The experiments compare the performance of the optimized version with that of the default scheme for the same patch hierarchy. The results are averaged over four timesteps during which the hierarchy remains the same. The input for visualization involves specifying subdomains of varying volumes. The experiments were conducted for the Kelvin–Helmholtz Instability test suite. For this application, we have chosen two domain sizes in which the base level consists of 512 patches (arranged as a $16 \times 16 \times 2$ grid) and 1024 patches (arranged as a $16 \times 16 \times 4$ grid). These correspond to patch hierarchy datasets of approximately 190 MB and 400 MB respectively. For our experiments, we consider subdomains of 5 different sizes: $2 \times 2 \times 2$, $4 \times 4 \times 2$, $8 \times 8 \times 2$, $12 \times 12 \times 2$, $16 \times 16 \times 2$ corresponding to volumes of 8, 32, 128, 288 and 512 patches respectively. For a given size, 20 different random regions spanning the entire domain were selected for visualization.

The effectiveness of the proposed optimization depends on the fact that the computation time dominates the overall execution time so that the visualization cost can be hidden. This is indeed the case when visualization is done onsite or when the visualization server is accessible by a high bandwidth network. The times for different sub components are shown in Table 1. As shown in the table, the total cost of intra-department remote visualization is a fraction of the computation time.

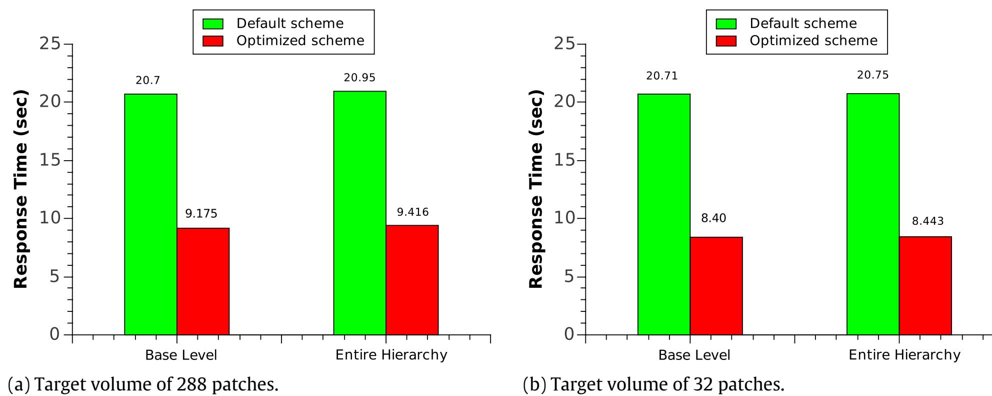
4.2. Response times

Fig. 5(a) compares the response times of the optimized scheme and the default scheme for visualizing the base level and the entire hierarchy when the target volume is 288 patches and the domain size is 512 patches. The improvement in response time is 18.7% in the case of visualizing the base level. In the case when the entire target hierarchy is visualized, the improvement in response time is 11.7%. The improvement is relatively lesser when compared to the base level visualization, due to the increase in the amount of data that is output and processed. AMR applications typically have small regions of interests that scientists visualize. The performance gain obtained from the optimizations are much more significant for smaller target volumes. Fig. 5(b) shows the gain in response times obtained for visualizing the base level and the entire hierarchy when the target volume is 32 patches and domain size is 512 patches. The gain obtained is 44.1% for base level visualization and 41.6% for visualizing the entire hierarchy. For smaller volumes the data from the target region is available quickly which provides an adequate amount of gaps for asynchronous visualization. Also, the time required to output, format and visualize the data for small



(a) Target volume of 288 patches. (b) Target volume of 32 patches.

Fig. 5. Response times for the domain size of 512 patches.



(a) Target volume of 288 patches. (b) Target volume of 32 patches.

Fig. 6. Response times for the domain size of 1024 patches.

volumes is less. For the target volume of 32 patches, the entire visualization is completed asynchronously.

Fig. 6(a) and (b) illustrate the improvement in response times for the test volumes of 288 and 32 patches, when the domain size is 1024 patches. For the test volume of 288 patches, the observed improvements for base level and entire target visualization are 57% and 55% respectively. For the test volume of 32 patches, the corresponding improvements are 60% and 59% respectively. In both cases, the observed improvements are higher when compared to the domain size of 512 patches. This is along expected lines, since we have doubled the domain size without varying the two test volumes which leads to a higher fraction of the hierarchy **H**–**T** available for asynchronism.

4.3. CPU gap utilization

Fig. 7 shows the CPU gap utilization for target volumes of 32 and 288, for the domain size of 512 patches. For each test volume, the figure shows a stacked bar graph for the gap usage for visualizing the base level and the entire target hierarchy. For the two volumes, the percentage of CPU idle time or gap utilized for fix-up and visualization varies from 38.5% to 39.3% for the base level and from 50.7% to 64.3% for the entire target hierarchy. As the size of target hierarchy **T** increases, the size of **H**–**T** decreases consequently leading to a decrease in the amount of gap available for visualization. Reduction in the gaps available results in a decrease in the amount of visualization operations that can be overlapped with computations. The amount of gap used for fix-up operations remains almost constant for all cases. This is because the number of the clean patches asynchronously fixed is constant for all the test cases.

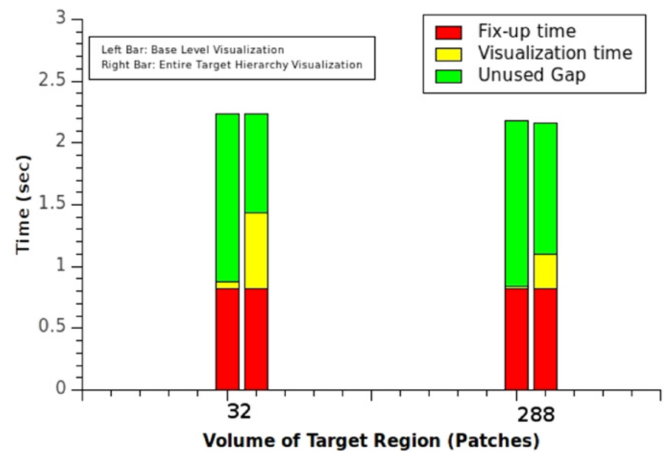


Fig. 7. CPU gap usage for domain size of 512 patches.

Fig. 8 shows a similar gap utilization graph for the domain size of 1024 patches. In this case, the percentage of gaps utilized ranges from 37.4% to 38.3% for base level visualization and from 39.3% to 42.8% for entire target visualization. Similar to the previous case, we observe a decrease in utilization between the test volume sizes of 32 and 288 patches. We also notice that the utilization obtained for this case are lesser when compared to the previous one. This is due to the fact that we are doing the same amount of work for visualization, while the amount of gap available has increased proportionately corresponding to the increase in domain size.

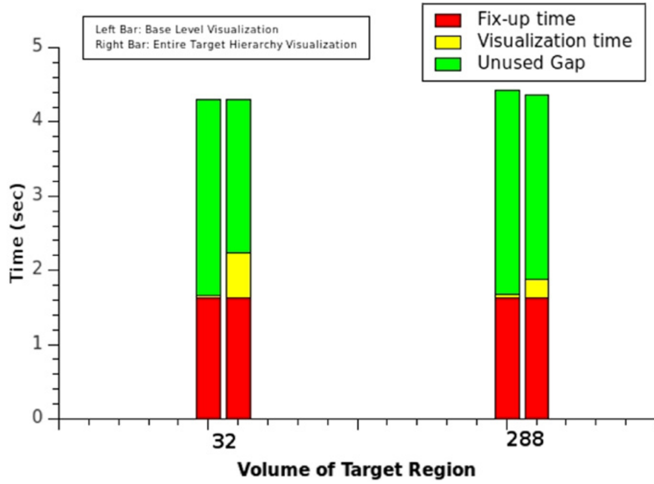


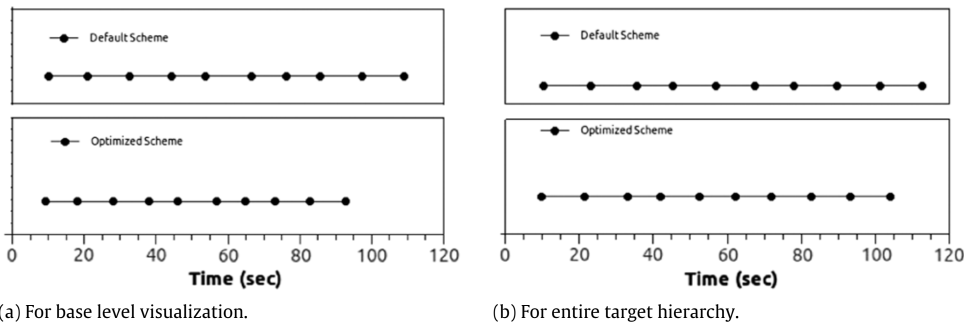
Fig. 8. CPU gap usage for domain size of 1024 patches.

4.4. Frequency of visualization

The improvement in the frequency of visualization can be represented by means of a *heart-beat* graph which linearly plots the points in time when visualization updates occur. The distance between two successive points represents the time elapsed between two visualizations. Thus, the graph enables one to judge the frequency of visualization based on the density of points on the line. Fig. 9(a) compares the heartbeat graphs for the default scheme and the optimized scheme when the base level is visualized on-site for the target volume of 128 patches. Fig. 9(b) compares the heartbeat graphs for the default scheme and the optimized scheme when the entire hierarchy is visualized on-site for the same target volume and domain size. The graphs show that the frequency of visualization in the optimized scheme is higher than that in the default scheme. The average interval of visualization in the optimized scheme is greater for visualizing the entire target hierarchy than for the base level. We observe that points on the heart-beat graph for the optimized scheme in 9(a) are more clustered than the points on the heart-beat graph for the optimized scheme in 9(b). This indicates that the gain obtained is more for the former case when compared to the latter. Fig. 10(a) and (b) show the corresponding heartbeat graphs for the domain size of 1024 patches. We observe similar improvements in refreshment frequency for both domain sizes. The observed gain ranges from 12% to 15.4% for base level visualization and from 7% to 15.9% for entire target hierarchy.

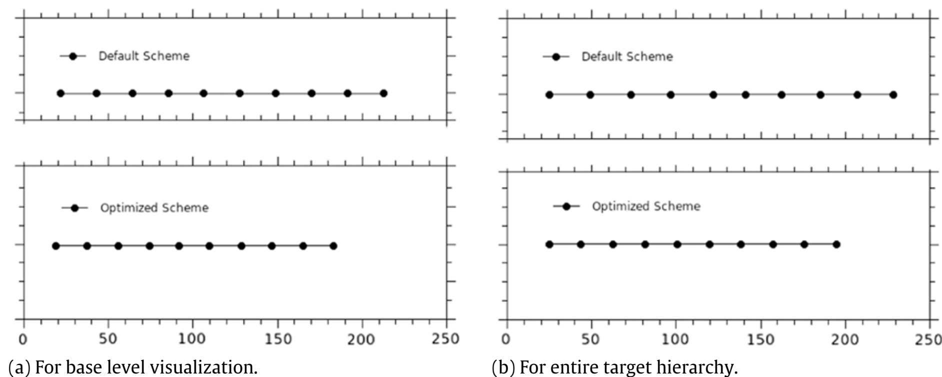
4.5. Remote visualization

Fig. 11(a)–(c) show the results for response times, average visualization interval and the gap utilization for the remote visualization of the entire target for the domain size of 512 patches. The size of the region visualized corresponds to the target volume of 128 patches. For the entire target hierarchy visualizations, the network speed plays an important role in response times. As shown in Fig. 11(a), the gain in response time is maximum for the *intra-department* setting at 47% and minimum for the *cross-continent* setting at 3.1%. This is because the data transfer time is much higher than the execution time for the *cross-continent* setting, which results in negligible benefits from the execution optimizations. The *intra-department* setting has the least data transfer time and hence obtains the best performance. A similar trend in performance is observed for the average visualization interval metric as observed in Fig. 11(b). The gain in frequency of visualization ranges from 1% to 13%. We observe in Fig. 11(c) that the amount of gap usage is almost constant for all three remote visualization settings. The three cases differ only in the network transfer time which does not play a part in gap utilization, since the network transfer for remote visualization does not utilize the CPU during the gap time. Fig. 12(a)–(c) show the results for response



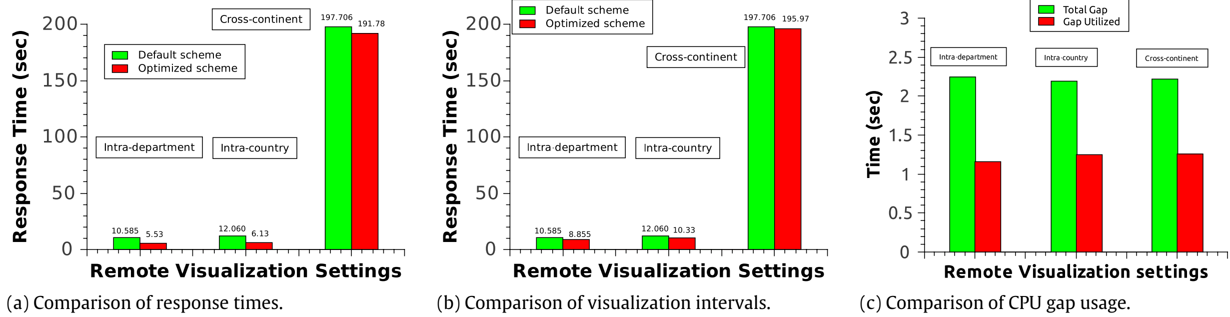
(a) For base level visualization. (b) For entire target hierarchy.

Fig. 9. Comparison of heart-beat graphs for continuous visualization for the domain size of 512 patches.



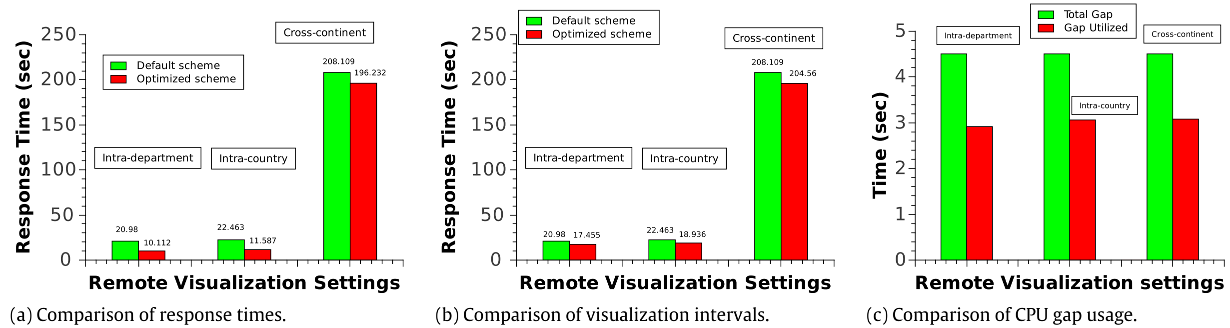
(a) For base level visualization. (b) For entire target hierarchy.

Fig. 10. Comparison of heart-beat graphs for continuous visualization for the domain size of 1024 patches.



(a) Comparison of response times. (b) Comparison of visualization intervals. (c) Comparison of CPU gap usage.

Fig. 11. Performance results for remote visualization of entire target hierarchy for the domain size of 512 patches.



(a) Comparison of response times. (b) Comparison of visualization intervals. (c) Comparison of CPU gap usage.

Fig. 12. Performance results for remote visualization of entire target hierarchy for the domain size of 1024 patches.

Table 2
Improvement in performance for visualizing the entire hierarchy on Fermi for the domain size of 512 patches.

Visualization scheme	% improvement in response time	% of gaps utilized	% improvement in frequency of visualization
On-site	44.4	51.6	20.7
Intra-department	43.4	39.3	18.5
Intra-country	32.1	39.3	16.4
Cross-continent	2.3	39.2	1.5

times, average visualization interval and the gap utilization for the remote visualization of 128 patches for the domain size of 1024 patches. Similar to the previous experiments, we obtain better gains when compared to the domain size of 512 patches.

4.6. Fermi results

We also performed experiments on the Fermi C2070 cluster. The results for visualizing the entire target hierarchy for the volume set at 128 patches are shown in Table 2. When compared to the results obtained on the Tesla S1070 cluster, the results on the Fermi cluster show considerable improvement in the frequency of visualization updates. Though the gap time is lesser for the Fermi when compared to Tesla due to a faster GPU, this is compensated by the fast Xeon W3550 core CPU on the Fermi system. The response times on both systems show similar trends.

5. Our optimizations applied to advanced AMR models

5.1. Self gravity

Along with pure hydrodynamic solvers, we also investigate astrophysical simulations. In this mode, the gravitational potential of all patches are advanced by solving the Poisson equations. For the shared timestepping mode, the fluid variables are advanced for the forward sweep, then the gravitational potentials are updated. After the variables are updated by self gravity, the backsweep is performed. For this model, we study the applicability of our proposed optimizations.

The Poisson solver requires the updated solution data of the coarse patches to provide the boundary data for finer patches by interpolation. In the hydrodynamics scheme, the input for the finer level patches are the values obtained from the coarse level resulting from the previous advance. This implies that the coarse and fine patches can be advanced independently before synchronizing. In contrast, the input for the finer patches in the self gravity module depends on the updated values of the coarse patches, which leads to inter-level dependencies. All coarse patches need to be updated, and thus, we cannot prune any computations in the self gravity module. Also, the gaps arising in this mode cannot be used for fix-ups. This is because the self gravity module requires all patches to be fixed after the forward sweep before it can commence. However, our reordering and asynchronous visualization optimizations can still be applied. In order to accelerate obtaining the data from target regions, we consider the self gravity module as an extension of the forward sweep. Thus, we complete the forward sweep and gravity update for the complete region, **H**, and then perform backward sweep in two phases: for the target region, **T**, and for the other regions, **H**–**T**. We perform asynchronous visualization of **T** while the backward sweep is performed for **H**–**T**. The gravity module takes almost as much time as the hydrodynamic advance. For a target volume of 128 patches and domain size of 512, Fig. 13 illustrates the response times for the onsite and various remote network configurations. We see that the overall improvement in response time for visualizing the entire hierarchy is 24%. Thus applying only our reordering and asynchronous visualization optimizations can still provide significant benefits for the self gravity module.

5.2. Individual timestepping

In the individual timestep mode, the integration of a level advances its solution by half the timestep value of its previous level.

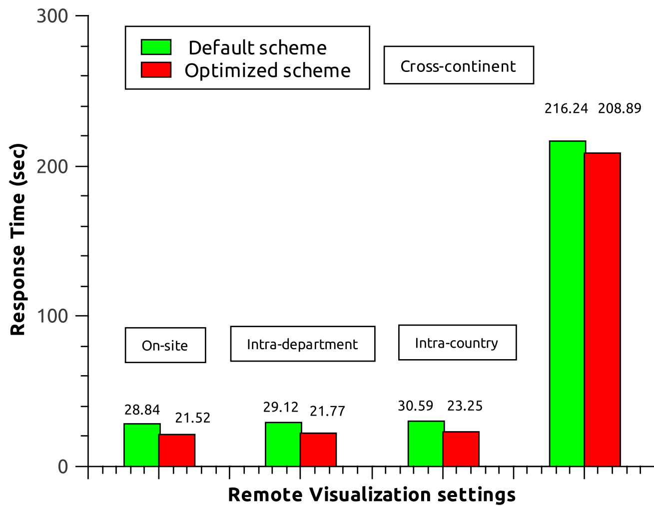


Fig. 13. Response times with self gravity enabled for a target size of 128 patches.

Thus, a level has to be advanced twice before it can synchronize with the previous level. Hence, level l must be advanced 2^l times to synchronize with level 0. For the pure hydrodynamics scheme, we see that the optimizations of asynchronous fixups and patch group pruning are directly applicable. For a domain size of 512 patches, we obtain around 11% reduction in execution time due to these optimizations. It is observed that the time taken to solve the finest level takes up to 85% of the total time. Since pruning of computations happen for levels below the finest and since fixups contribute to only about 5% of the total time, the gains obtained are limited to these values.

Our reordering scheme cannot be applied in our current implementation due to the complex order in which the levels and patches need to be advanced. In this scheme, the first advance does a forward sweep while the subsequent advance performs a backsweep. Unlike the shared timestep scheme, the levels have to be advanced in order and have to synchronize periodically. Also, refinement of levels happen within a timestep and hence the hierarchy is modified during the execution of the timestep. Thus, interleaved nature of the forward and backward schemes prevents us from applying our partitioning technique directly.

However, we see that even for this scheme, it is possible to obtain the data for visualization faster than the default scheme by adopting the following methodology: along with the target patch groups, we also identify patch groups which form the boundaries around the target patch groups. A patch group belongs to a *k-layer* boundary if it is separated by a distance of k from the target region. Our proposed solution involves first applying the advances on the target patches and specific layers of boundary patch groups for each level. We illustrate the technique for level one and then generalize it for all levels. Level 1 requires four applications of advances which consist of alternating forward and backward sweeps. For the first advance, we forward sweep the target patches of level 1 along with 4 layers of boundary patch groups. For the next advance, we backward sweep the target patch groups with 3 boundary layers. The fourth layer's solutions are used to update the layer 3 patch groups. For each substep, we skip a layer and for the final backsweep we solve only the target patch groups. This technique ensures that the target patch groups are solved using the accurate values.

This solution can be generalized for all levels. Specifically, level lv requires $2^{(lv+1)}$ boundary patch groups since it requires as many substeps. This results in choosing 2 layers of boundary patch groups on the coarsest level. These base level patch groups when refined would consist of the required layers of boundaries.

The implementation of this methodology requires a complex book keeping mechanism to track the state of the solutions for each patch. This is because this method leaves different patches belonging to the same level at different states. The implementation of this methodology and performance estimates and evaluations will be considered in the future.

6. Discussion

We see that when 4 or more high end CPU cores (like Xeon) and OpenMP threads are utilized, the idling always occurs on the CPU, and hence our optimizations can be applied to obtain performance benefits. Since our strategy is to overlap as much CPU operations as possible, we resort to using at least 8 cores so that we have the capability to perform visualizations as well. Thus our demonstration with 8 cores is close to the minimal number of cores required for obtaining performance benefits over the default scheme. In this setup, the performance ratio of the solve step on the GPU and the (prepare+close) steps on the CPU is 1.4 and is close to the threshold value for obtaining better performance. With more high-end CPUs, this ratio will increase, resulting in more idling times on CPU. This would improve the scope to perform more compute intensive visualization operations asynchronously.

For the lesser number of cores, the prepares and closing operations become bottlenecks and hence there are no CPU gaps available to exploit. We note that our optimizations about removing redundant computations will always be feasible. When the CPU is slower than the GPU, the net execution time is the sum of all the prepares and closes performed. The pruning optimization removes patch groups whose computations are redundant, from being considered and thus avoids some of the potentially expensive prepare operation.

7. Related work

The typical CPU based optimization techniques cannot be trivially extended for an efficient execution on the GPU. While dealing with GPU systems, the priority is to use the architectural aspects for efficient execution. Enzo [16] is an AMR codebase for large scale cosmological simulations which implements GPU solvers which can deal with arbitrary patch sizes. However, Enzo does not exploit concurrency between the CPU and GPU or between the various independent GPU operations. Uintah [6] is an object oriented framework which provides an environment for solving fluid–structure interaction problems with support for ray tracing radiation transport models, on structured adaptive grids. It uses a task-based approach with automated load balancing and its runtime system handles the details of asynchronous memory copies to and from the GPU. The work does not discuss optimizations to improve the rate of online visualization.

8. Conclusion and future work

This work presented optimization strategies for performing continuous visualization of a GPU based AMR hydrodynamics code. We reorder the computations to obtain the data for visualization at a faster rate. We accommodate the fix-ups and visualization steps in the CPU idling times and prune unnecessary computation of patches. We proposed response time, average visualization interval and CPU gap usage as metrics to evaluate the performances of the default and optimized scheme. We performed experiments on two datasets to show the scalability of the proposed solutions. We observe a 60% improvement in response time and 16% improvement in frequency of visualization on Tesla S1070. We observe up to a 21% improvement in frequency of visualization on the Fermi C2070 cluster.

In the future, we plan to extend the scheme for the individual timestepping mode. We also plan on building generic runtime frameworks for efficient executions of AMR applications on GPUs. Our runtime framework should be capable of estimating the average gap available and the relative costs of the various CPU and GPU operations for various hardware configurations and target volumes. Since GAMER uses fixed size patches, the costs of the various operations can be estimated a priori. Our runtime framework will perform quick profiling runs when the application begins execution to obtain the various estimates and then judge the applicability of asynchronous CPU execution. We also plan on implementing optimization strategies for auto-reorganization of data layout to improve the number of coalesced access, based on the memory access patterns in the solver kernel. Our optimization schemes have been integrated in the GAMER codebase. We plan to release our optimizations as a patch for GAMER users.

Acknowledgments

We would like to thank Hsi-Yu Schive of National Taiwan University, for providing us the GAMER source code and for his invaluable assistance which aided our understanding of the codebase. We would like to thank the anonymous reviewers for their valuable comments that helped significantly improve the paper.

References

[1] M.J. Berger, J. Oliger, Adaptive mesh refinement for hyperbolic partial differential equations, Journal of Computational Physics 53 (1984) 484–512.
[2] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, L. Wilcox, Extreme-scale AMR, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–12.
[3] J.E. Flaherty, R.M. Loy, M.S. Shephard, B.K. Szymanski, J.D. Teresco, L.H. Ziantz, Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws, Journal of Parallel and Distributed Computing 47 (1997) 139–152. Elsevier.
[4] B. Fryxell, K. Olson, P. Ricker, F.X. Timmes, M. Zingale, D.Q. Lamb, P. MacNeice, R. Rosner, J.W. Truran, H. Tufo, Flash: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes, The Astrophysical Journal Supplement Series 131 (2000) 273.
[5] M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Coker, E. Dendy, R. Hueckstaedt, K. New, W.R. Oakes, D. Ranta, R. Stefan, The RAGE radiation-hydrodynamic code, Computational Science & Discovery 1 (2008) 015005.
[6] A. Humphrey, Q. Meng, M. Berzins, 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: Bridging from the eXtreme to the campus and beyond, Chicago, Illinois, XSEDE '12, ACM, New York, NY, USA, 2012, pp. 4:1–4:8.
[7] B.S. Kirk, J.W. Peterson, R.H. Stogner, G.F. Carey, `libMesh`: a C++ library for parallel adaptive mesh refinement/coarsening simulations, Engineering with Computers 22 (2006) 237–254.
[8] Z. Lan, V. Taylor, G. Bryan, Dynamic load balancing for structured adaptive mesh refinement applications, in: International Conference on Parallel Processing, 2001, pp. 571–579.
[9] Lawrence Livermore National Laboratory, Silo user's guide 4.8, 2010. https://wci.llnl.gov/codes/visit/3rd_party/Silo.book.pdf.
[10] Lawrence Livermore National Laboratory, VisIt visualization tool, 2010. http://www.llnl.gov/visit.
[11] P. MacNeice, K.M. Olson, C. Mobarry, R. de Fainchtein, C. Packer, PARAMESH: a parallel adaptive mesh refinement community toolkit, Computer Physics Communications 126 (2000) 330–354.
[12] C.A. Rendleman, V.E. Beckner, M. Lijewski, W. Crutchfield, J.B. Bell, Parallelization of structured, hierarchical adaptive mesh refinement algorithms, Computing and Visualization in Science 3 (2000) 147–157.
[13] R. Sampath, S. Adavani, H. Sundar, I. Lashuk, G. Biros, Dendro: parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008, 2008, pp. 1–12.
[14] H. Schive, Y. Tsai, T. Chiueh, GAMER: a graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics, The Astrophysical Journal Supplement Series 186 (2010) 457.
[15] H.-Y. Schive, U.-H. Zhang, T. Chiueh, Directionally unsplit hydrodynamic schemes with hybrid MPI/OpenMP/GPU parallelization in AMR, International Journal of High Performance Computing Applications 26 (4) (2012) 367–377.
[16] P. Wang, T. Abel, R. Kaehler, Adaptive mesh fluid simulations on GPU, New Astronomy 15 (2010) 581–589.
[17] A.M. Wissink, R.D. Hornung, S.R. Kohn, S.S. Smith, N. Elliott, Large scale parallel structured AMR calculations using the SAMRAI framework, in: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM), Supercomputing'01, ACM, New York, NY, USA, 2001, p. 6.

Hari K. Raghavan is a masters student in Supercomputer Education and Research Centre, Indian Institute of Science and is a member of the Grid Applications Research Laboratory. He obtained his B.Tech. degree in Computer Science and Engineering at National Institute of Technology Tiruchirapalli.

Sathish Vadhiyar is an Assistant Professor in Supercomputer Education and Research Centre, Indian Institute of Science. He obtained his B.E. degree in the Department of Computer Science and Engineering at Thiagarajar College of Engineering, India in 1997 and received his Masters degree in Computer Science at Clemson University, USA, in 1999. He graduated with a Ph.D. in the Computer Science Department at University of Tennessee, USA, in 2003. His research areas are in parallel and grid computing with primary focus on grid applications, fault tolerance, scheduling and rescheduling methodologies for grid systems. Dr. Vadhiyar is a member of IEEE and has published papers in peer-reviewed journals and conferences. He was a tutorial chair and session chair of escience 2007 and served on the program committees of conferences related to parallel and grid computing including IPDPS, CCGrid, eScience and HiPC.