Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2015, Vol. 29(2) 135–153

DOI: 10.1177/1094342014545546
hpc.sagepub.com
SAGE

Adaptive executions of hyperbolic block-structured AMR applications on GPU systems

Hari K Raghavan and Sathish S Vadhiyar

Abstract

A block-structured adaptive mesh refinement (AMR) technique has been used to obtain numerical solutions for many scientific applications. Some block-structured AMR approaches have focused on forming patches of non-uniform sizes where the size of a patch can be tuned to the geometry of a region of interest. In this paper, we develop strategies for adaptive execution of block-structured AMR applications on GPUs, for hyperbolic directionally split solvers. While effective hybrid execution strategies exist for applications with uniform patches, our work considers efficient execution of non-uniform patches with different workloads. Our techniques include bin-packing work units to load balance GPU computations, adaptive asynchronism between CPU and GPU executions using a knapsack formulation, and scheduling communications for multi-GPU executions. Our experiments with synthetic and real data, for single-GPU and multi-GPU executions, on Tesla S1070 and Fermi C2070 clusters, show that our strategies result in up to a 3.23 speedup in performance over existing strategies.

Keywords

Adaptive mesh refinement, GPU executions, dynamic load balancing, asynchronous executions of CPUs and GPUs, coalesced access

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 patches) over which finite difference or other discretization schemes can be applied. The adaptive mesh refinement (AMR) (Berger and Oliger, 1984) technique dynamically varies the spatio-temporal resolution of mesh regions based on the local errors in the regions. Some AMR approaches use patches of uniform sizes over regions of interest (Fryxell et al., 2000; MacNeice et al., 2000; Ziegler, 2008) while other efforts including SAMRAI (Wissink et al., 2001), Enzo (Wang et al., 2010), and Uintah (Humphrey et al., 2012) cover the regions of interest with patches of non-uniform sizes. The uniform and non-uniform patches are illustrated in Figure 1.

In this work, we have developed strategies for adaptive execution of AMR applications with non-uniform patches on GPU systems. We primarily deal with directionally split hyperbolic solvers for hydrodynamics applications in which explicit timestepping is used for following solution features. The AMR approach we consider is the block-structured AMR developed by Berger and Oliger (1984). Our work considers uniform timestepping in which all the patches of the AMR hierarchy are advanced using the same timestep size. In these applications, the computational domain is approximated into a discrete set of *cells*. The cells are organized into units called meshes (or *patches*). The sets of cells along a given dimension that are solved are referred to as *columns*. The concept of patches, cells and columns is illustrated in Figure 2. The domain is initially covered by a set of low-resolution patches which form the base level of the hierarchy. The simulation proceeds in discrete units called *timesteps*. Every application of the numerical method to a patch advances the

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

Corresponding author:
Sathish S Vadhiyar, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore-560012, India.
Email: vss@serc.iisc.in

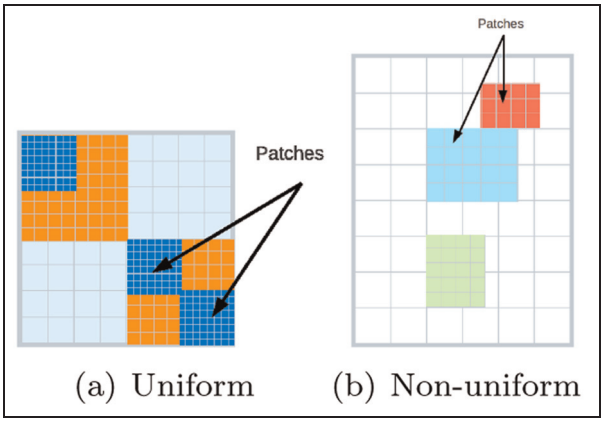


Figure 1. Uniform and non-uniform patches.

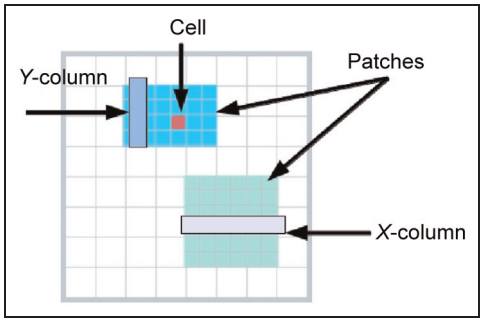


Figure 2. Illustration of patches, cells and columns.

simulation time by an amount. In a timestep, the solution values are advanced for all the levels in the hierarchy starting from the coarsest level. A fix-up operation, in which the solutions of the finer mesh cells are used to adjust the values of the coarser mesh cells, is performed after advancing all the levels. As the computations proceed, regions of interest with large local errors are marked for refinement. The refinement process involves overlaying these regions with a finer mesh having a higher resolution, resulting in the formation of patches. The pseudo code for the hyperbolic block-structured AMR algorithm can be represented as follows:

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

General purpose graphics processing units (GPGPUs) (NVIDIA, 2010), with their support for fine-grained parallelism, offer an attractive option for obtaining high performance for AMR applications. The most computationally intensive component of AMR is the numerical integration step which invokes a stencil-based solver on all the patches. Stencil-based computations involve large amounts of fine-grained and data parallelism, and hence can be parallelized on GPUs for high efficiency. The computations are mapped to the GPU using a CUDA kernel involving a number of thread blocks and threads, with each thread block solving a patch where the constituent threads cooperate and handle the sub-regions in a patch. While the GPU is in charge of the computational part, the host CPU handles the AMR control functions including making refinement decisions and managing the patch data structure, and providing inputs for the GPU. Hence, some recent efforts have focused on parallelization of AMR applications on GPUs, including GAMER (Schive et al., 2010), Enzo (Wang et al., 2010), Carpet (Blazewicz et al., 2012), and Uintah (Humphrey et al., 2012). However, these efforts do not adequately deal with non-uniform patches. For example, GAMER (Schive et al., 2010) provides solutions for efficient heterogeneous execution of AMR applications with only uniform patches on GPUs. Enzo (Wang et al., 2010) considers non-uniform patches, but does not exploit the opportunity for asynchronous executions between the CPU and GPU.

Efficient execution of AMR applications with non-uniform patches involves certain unique challenges. Due to the variation in the sizes of the patches, the computational loads in processing the patches using the GPU cores vary. The load imbalance among the GPU cores can in turn lead to high GPU execution times. Another challenge is related to efficient asynchronous executions between the CPU and GPU to maximize performance. This typically involves breaking down the execution into multiple batches so that while one batch is executed on the GPU, the next batch is processed or prepared and kept ready by the CPU. For AMR applications with non-uniform patches, the GPU execution time of a batch varies across different batches. Hence the CPU must dynamically select the next subset or batch with appropriate computational load for preparation based on the time required for execution of the previous batch by the GPU. Finally, executions on multi-GPU systems involve communicating boundary data between processes at the end of each timestep. This is an overhead that must be reduced to achieve scalability. Thus the executions of AMR applications with non-uniform patches involves challenges across multiple levels: within the GPU, between the GPU and the host CPU, and across multiple CPU–GPU systems.

Our work has developed strategies including dynamic scheduling of non-uniform patches for asynchronous execution on CPUs and GPUs, and load-balancing computations on GPU cores. We formulate the problem of assigning patches to thread blocks of GPUs for load-balanced executions by the GPU cores as a 3D bin-packing problem, and use an efficient

heuristic to solve the problem. We have also built a performance model that estimates the time for execution of a batch of patches by the GPU, and for preparation by the GPU. Using these estimates, we dynamically select a subset of patches for preparation by the CPU while the GPU executes the previous subset. We formulate this problem of selection of patches that can be prepared within a certain amount of time as a knapsack problem, and use a heuristic algorithm for the solution. Finally, we propose strategies for reordering computations and overlapping computations with inter-CPU communications for high performance and scalability on multi-GPU clusters. Our other optimizations include determining the geometry of patches based on GPU memory configuration, improved coalesced access of data by the GPU threads, and avoiding synchronization between the threads.

We conducted experiments with a hydrodynamics application with sets of synthetic inputs as well as inputs from real applications. Our experiments with single-GPU and multi-GPU executions, on Tesla S1070 and Fermi C2070 clusters, show that our strategies result in up to a 3.23 speedup in performance over existing strategies. Our bin-packing-based load-balancing gives performance gains of up to 39%, kernel optimizations give an improvement of up to 20%, and our strategies for adaptive asynchronism between CPU–GPU executions give performance improvements of up to 17% over default static asynchronous executions. In the future, we would like to explore making the bin sizes adaptive and introduce more dynamic schemes for assigning thread blocks to workloads.

Section 2 provides background on AMR application executions on GPUs, and the steps involved in the hydrodynamics application. Section 3 describes our strategies on load-balancing computational units among GPU cores, adaptive asynchronism for simultaneous use of CPU and GPU cores, multi-GPU executions, and other optimizations. In Section 4, we describe our experiments and results on single- and multi-GPU systems with synthetic and real applications on two systems. Section 5 gives related work. Finally, Section 6 summarizes the work and gives scope for future work.

2 Background

2.1 AMR and GPU executions

The set of patches advanced simultaneously per GPU kernel launch is termed a *workgroup*. Advancing a set of patches on a GPU system involves three steps:

1. Preparation: the array containing input data for the GPU solver is filled from the patch data structure. The boundary data of the patches is supplied by copying data from neighboring patches or by interpolating from their parent from the coarser level. The prepare step is performed on the CPU.
2. Solve: the input array containing the prepared workgroup is solved asynchronously on the GPU by invoking the kernel for the numerical method. 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.

2.2 Asynchronous executions

The solve step on the GPU can be asynchronously executed with the preparation and the closing steps on the CPU. GAMER (Schive et al., 2010) adopts this asynchronous execution model. Specifically, the CPU sends the i th workgroup to the GPU for solving, and while the kernel executes, it closes the $(i-1)$th workgroup (if there was a solve phase previously) and prepares the $(i+1)$th workgroup (if any workgroups are left in the level) asynchronously. For fixed-size patches, the workgroup size can be fixed. In our scheme with non-uniform patches, each patch has a different preparation and closing cost on the CPU and solve cost on the GPU. In order to overlap prepares and solves, we need to select a suitable subset of work items that can be prepared while the GPU solves its current workload.

2.3 Application

In this work, we consider a hydrodynamics application which is modeled using Euler equations. We explore two second-order directionally split hyperbolic schemes (LeVeque, 2002) to solve the equation, namely, the relaxing total variation diminishing (RTVD) scheme (Trac and Pen, 2003) modified from the GAMER implementation (Schive et al., 2010), and the total variation diminishing (TVD) scheme implemented in Enzo (Wang et al., 2010). Both these schemes reduce solving the 3D equations to a 1D problem by performing three sweeps, for each dimension.

For each sweep, GAMER's RTVD scheme processes all the data columns along the sweeping dimensions individually. For example, for a patch having dimensions $W \times H \times D$ (including the ghost cells), a sweep along the Y-dimension involves sweeping the $W \times D$ Y-columns, each having H cells. Each sweep consists of a number of substeps including calculating inter-cell flux values and updating half step solutions. The RTVD stencil uses a seven-point constant coefficient stencil with one cell per dimension for calculating midpoint values. It requires two more cells per dimension to evaluate the full step values. Thus, the scheme

requires a total of three ghost cells on each side. On the GPU, the column-sweeps along the X-dimension take less time than those for the Y- and Z-dimensions, since the memory accesses are coalesced.

Enzo implements a second-order TVD scheme which regards the patch as one large 1D column (read along the direction of the sweep). This 1D column is tiled using multiple thread blocks and solved. The advantage of this approach is that it functions for arbitrary dimensions and achieves good load balance and minimal idling. In contrast to the RTVD scheme, the primary input is not updated until all the three sweeps are completed. This implementation also requires that each sweep is invoked as a separate kernel to ensure that all thread blocks have computed the data necessary for the next sweep.

3 Methodology

In GAMER, a set or a batch of patches is solved on a GPU using a single kernel invocation. A thread block acts on one or more patches. For each patch, the thread block reads a data column into shared memory, finishes the sweep for the column including all the substeps, and writes the final results back into global memory. The scheme is illustrated in Figure 3 in which all the three dimensions of the patch are equal.

For non-uniform patches with unequal dimensions, the grid and the thread block configuration of a CUDA kernel will have to be carefully chosen. Specifically, the number of thread blocks, and organization of a thread block in terms of the number of threads in each dimension, are important parameters that determine performance. We bear in mind that CUDA allocates threads only in fixed granularities of 32 threads (referred to as *warps*). Requesting for 100 threads would be inefficient since 128 threads are allotted anyway and the last 28 are kept idle.

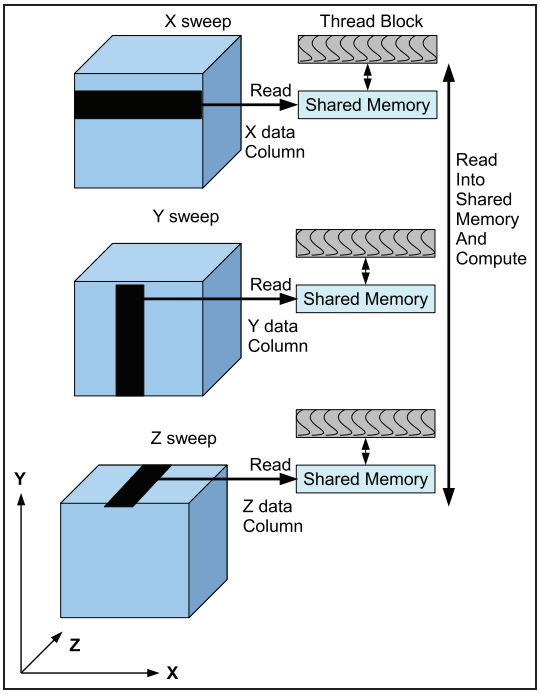


Figure 3. Sweeping the columns of a patch by a thread block.

For our application, a naive option is to launch thread blocks with the maximum number of active threads possible and fix the organization of the thread blocks statically based on the maximum patch dimension across all the patches in the current workload. A given thread block will use the same configuration in terms of the number of registers and shared memory usage for all the three sweeps, and the same structure is used in all the thread blocks. The maximum number of active threads, defined as the number of resident threads currently utilizing the GPU's resources, is determined by hardware constraints as well as by limits on the amount of resources requested by a kernel. As an example, if the maximum number of active threads possible is 128 and the maximum dimension observed in the current workload is 42, this approach tries to launch all thread blocks organized as 42×3 units, that is, 42 rows and three columns of threads. Thus, the 126 threads in a thread block load and sweep three columns at a time, before loading the next set of three columns. While this model is an effective mechanism for uniform patches with equal dimensions, this leads to severe performance degradations for non-uniform patches.

First, within a thread block, thread divergence and idling will occur due to variation in dimensions of a patch. Additional conditional statements will have to be added in the kernel to disable threads with thread ids that are beyond the data column width. This has a heavy performance penalty in the CUDA model depending on the number of idle threads, since clock cycles are wasted with no useful work being done. This is illustrated with a simple hypothetical example for a 2D problem considering a patch of dimensions 22×16. A thread block of organization 42×3 threads acting on this patch will cause idling of 60 threads during the X-sweeps and 78 threads during the Y-sweeps as illustrated in Figure 4(a).

Another scheme is a dynamic scheme in which each thread block tiles as many columns completely per iteration. In this scheme, each thread block is logically organized individually depending on the patches it acts upon. While this seemingly is a better strategy than the static scheme, it is still incapable of avoiding thread idling in a satisfactory manner, and can also result in reduced occupancy due to the number of active threads less than the maximum number of 128 active threads. For the 2D problem with a 22×16 patch, the thread block can organize itself at most as a 22×5 tile since the RTVD kernel requires each column to be

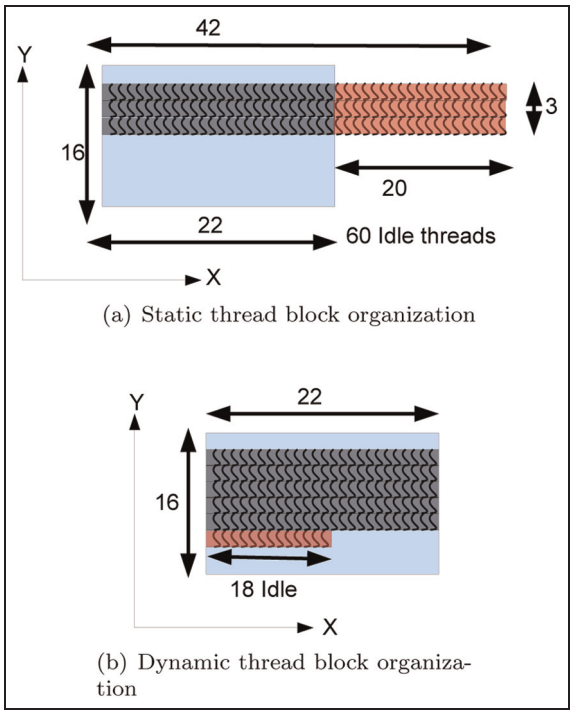


Figure 4. Thread idling in non-uniform patches.

completely processed. This leads to 18 $(128 - 110)$ threads being inactive during each sweep. This is illustrated in Figure 4(b).

Second, across thread blocks, load imbalances can occur since the patches handled by different thread blocks can be different in size and each thread block can possess different workloads. This results in load imbalances between the Streaming Multiprocessors (SMs) of the GPU which would lead to longer a execution time per kernel launch. Thus, it is necessary to implement a strategy to allocate the patches into equal chunks of work and a scheme to distribute the thread blocks efficiently among these chunks. It is also important to asynchronously utilize the CPU for useful computations while the GPU solves a batch of patches. We also need to optimize for latency-hiding by ensuring adequate occupancy, and minimize communication overheads for multi-GPU executions. We describe our optimizations related to intra-GPU load-balancing, asynchronous executions, and other optimizations in the following subsections.

3.1 Intra-GPU load-balancing

The basic idea in our load-balancing scheme is to combine or fuse multiple non-uniform patches into fixed uniformly sized patch groups, and make a thread block act on multiple patches of a group. We illustrate our load-balancing scheme using the same 2D example involving a patch of dimensions 22×16 and employing thread blocks of 128 threads. This patch can be joined with a 10×16 patch if one is present. Processing the patches individually by the thread block will result in 18 $(128 - 110)$ and 8 $(128 - 120)$ threads being inactive, respectively, during their X-sweeps. When combined to form a patch group of dimensions 32×16, they can then be optimally tiled by 32×4 threads in a thread block with no idling. This is illustrated in Figure 5. A thread block thus acts on parts of both the patches, and all the threads of the two thread blocks are kept busy during both the X- and Y-sweeps. Thus, by forming uniformly sized patch groups or bins from non-uniform patches, and having a fixed number of thread blocks acting on a patch group, we ensure both minimal thread idling within a thread block and maximum load-balancing across thread blocks.

We extend this idea for realistic 3D problems, where the goal would be to pack together multiple patches within 3D *bins* with minimal wastage of space. All the bins are cubic (the three dimensions of the bins are equal) and are of equal sizes, thus ensuring that all thread blocks are approximately work-balanced during all 3D sweeps. This also implicitly reduces thread divergence, resulting in an increased amount of productive work performed per clock cycle.

The RTVD kernel is an arithmetically intense scheme with heavy register (approximately 50) and shared memory (80 bytes per thread) requirements. On

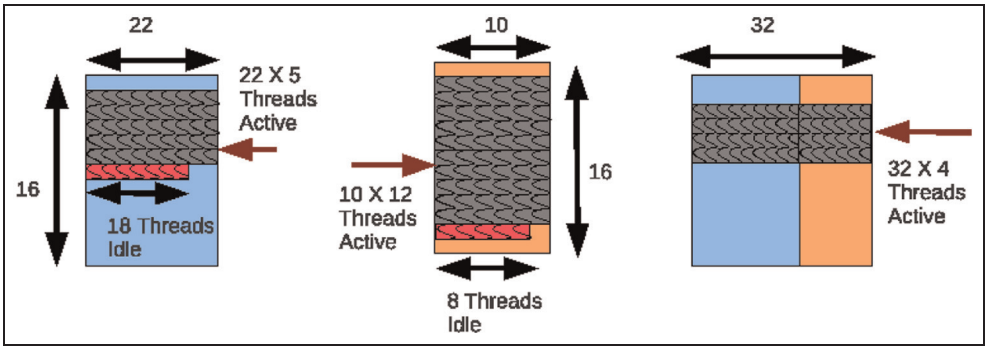


Figure 5. Illustration of patch groups.

devices with compute capability 1.3 (Tesla S1070, etc.), this achieves at most 192 active threads per SM. In our scheme, we organize these 192 threads as three thread blocks of 64 threads each, thus employing three active thread blocks per SM. This ensures that we have sufficient thread blocks to cover synchronization latencies. The total number of thread blocks launched per kernel is thus three times the number of SMs. The number of bins processed by a GPU kernel in our scheme varies for different batches, and is typically greater than the number of thread blocks. A bin is processed by all the thread blocks before the next bin is processed. The advantages of this model are twofold. This implicitly ensures that all thread blocks have equal loads. Secondly, the model where each bin is exclusively processed by a thread block becomes inefficient when the number of bins in the batch is not a multiple of the number of active thread blocks. Thus, we employ the strategy where all the active threads of all active thread blocks co-operate to solve a bin.

The problem of assigning 3D patches into fixed-size cubic bins to achieve maximum utilization of the bin capacity, and thereby achieve load-balancing, reduces to a 3D geometric bin-packing problem. The maximum utilization objective is to pack as many available patches as possible and reduce the number of kernel invocations. Besides, the objective also minimizes thread idling and divergence as mentioned earlier. When using the single-timestepping mode, the patches across different levels can be packed together into bins since all levels are advanced for the same number of times per timestep. For multi-timestepping, the patches have to be processed level by level.

3D bin-packing is an NP-hard problem (Garey and Johnson, 1979) for which we use an approximation algorithm designed by Crainic et al. (2008). This uses the best-fit heuristic using the concept of extreme points. The algorithm first sorts the set of patches according to their volume with ties broken by the height. Given a set of objects already packed in a bin, the algorithm heuristically places the next object at a location in the bin which is designed to minimize wastage of space. When an object of dimensions $w \times h \times d$ is placed in a bin at the co-ordinates x, y, z, its co-ordinates $(x + w, y, z)$, $(x, y + h, z)$, and $(x, y, z + d)$ are projected on other items of the bin and the bin walls. The intersection of these projections and the surfaces of the patches defines the *extreme points*. When an object is considered for placing in a bin, only the set of extreme points is considered for placement. It is observed that an object can generate at most six extreme points and hence the search space is limited to $6 \times n$ points where n is the number of items packed. Also, extreme points keep the fragmentation to a minimum and thus generate the best packing. The best-fit strategy is a heuristic which looks at all the extreme points in every open bin and accommodates the object at the optimal extreme point. The best extreme point to fit a bin is one such that the residual volume around the point is reduced to the maximum by accommodating the object. If no such location is found, a new bin is opened for the object in question. The bins are then processed by the thread blocks. We implemented the algorithm and integrated it into our codebase.

In our application, the bin width impacts the shared memory usage since it is the minimum unit that is read and processed by the threads. Since the amount of shared memory usage by a thread block determines the number of active thread blocks, the bin width affects the GPU occupancy. The efficiency of the bin-packing algorithm is also dependent on the bin width as well as on the dimensions of the patches. Thus, the width of the bin has to be chosen for optimal GPU occupancy and high efficiency of the bin-packing algorithm. We determined experimentally that the best dimensions for the bins for high performance is $64 \times 64 \times 64$. This gave the best occupancy as well as the best packing ratio. The bin width being a multiple of the warp size (32) also ensures that there are no underpopulated warps.

3.2 Adaptive asynchronism

The CPU invokes a kernel on the GPU with a set of bins containing the patches. We refer to this set as a *batch*. The kernel executes the numerical integration step involving stencil-based computations on the patches. All the bins and patches are executed on the GPU by multiple kernel invocations by the CPU with multiple batches. While a batch is executed on the GPU, the CPU can asynchronously prepare the next batch corresponding to another set of bins. The preparation involves filling an input array for the GPU solver with data corresponding to the patches in the bins, copying boundary or ghost zone data for the patches from their neighbors, and interpolating from the coarse-level patches.

The next batch for preparation by the CPU will have to be carefully chosen for efficient asynchronism of CPU and GPU execution. A simple method of implementing asynchronism would be to choose some k fixed number of subsequent bins as the next batch for preparation. However, this strategy is inefficient for non-uniform patches, since the GPU execution and CPU preparation times can vary for different patches and different bins. Since bin-packing does not result in perfect or complete packing of bins due to different patch dimensions, the workloads of the different bins can vary, resulting in different GPU execution and CPU preparation times for different bins. Moreover, the dimensions of a patch determine the time taken to interpolate or fetch its ghost (boundary) cells during

Table 1. Estimation of solve times.

Patch dimension	Actual time (ms)	Predicted time (ms)	Relative difference $\left(\frac{predicted-actual}{actual}\right)$(%)
24,20,20	1.011	1.036	2.47
44,36,16	2.517	2.717	7.94
40,32,36	5.880	5.504	−6.40
16,44,20	1.486	1.574	5.92
24,16,36	1.367	1.440	5.34
20,44,28	2.739	2.832	3.39

preparation by the CPU. The total cost to prepare a bin is the net sum of computing the ghost data and copying the data for all the constituent patches. Thus, choosing a fixed number of subsequent bins can result in a large difference between the GPU execution and CPU preparation times, leading to idling of either the CPU (if GPU execution is greater) or the GPU (if CPU preparation is greater), depending on the workload of the bins.

Hence, the CPU must dynamically select the next batch for preparation based on the time required for its preparation and the time needed for execution of the previous batch by the GPU. We implement a strategy which estimates the time taken by the GPU to complete its current workload. Using this cost plus a tolerance limit as the maximum time available, we pick a subset which can be prepared without exceeding this cost. We formulate this problem of selection of patches that can be prepared within a certain amount of time as a knapsack problem (Martello and Toth, 1990). We also consider the problem of efficiently distributing the work among the CPU cores employing OpenMP parallelism. We formulate the knapsack problem with P queues, where P is the number of CPU threads. The capacity of each queue is the estimated time available for the GPU to complete its current workload. Thus, we aim to maximize the number of bins we can prepare while ensuring that the CPU threads are equally balanced. We solve the knapsack problem heuristically by first sorting the available bins in non-increasing order of volume to prepare cost ratios of the bins. The bins are then considered in this order for selection. This heuristic thus gives priority to bins of large volumes and lower prepare costs, thus maximizing the number of bins prepared at a time. The constituent patches of a selected bin are then distributed to the queues such that the patch under consideration is assigned to the least loaded queue. Finally, each OpenMP thread prepares the patches present in its queue.

Our asynchronous execution strategy needs estimates of CPU preparation and GPU execution of bins. The execution time of a bin depends on the total work done in all three sweeps and is represented as follows:

$$t^{bin}_{exec} = C_1 * N_x + C_2 * N_y + C_3 * N_z \qquad (1)$$

Here N_x, N_y, and N_z are the total number of columns in the X-, Y-, and Z-sweeps respectively. C_1, C_2, and C_3 are constant costs associated with each sweep direction. We estimate these by performing each sweep individually with one thread block. We then normalize the total cost, assuming that all the active thread blocks solve the bin. The efficiency of the estimator is illustrated in Table 1 for a set of patches. The actual times shown in the table are the average values obtained with 10 different runs. It can be seen that the estimated times are within 8% of the actual values.

The net solve time is then determined by the sum of the execution times of the bins plus the effective time to transfer the bins to the GPU. By using CUDA streams, the net transfer time can be limited to the time taken to send the first bin downstream and transfer back the last bin upstream. Hence, the time taken by the GPU for the ith batch is represented as

$$t^{i}_{solve} = t_{trans} * Size_0 + \sum_{j=1}^{n} t^{j}_{exec} + t_{trans} * Size_n \qquad (2)$$

where t^{j}_{exec} represents the solve time of bin j of the batch and t_{trans} is the time taken to transfer a byte of data. This is set based on the available bandwidth of the CPU–GPU interconnection. $Size_0$ and $Size_n$ are the sizes in bytes of the first and last batches. We add these costs since they do not have any asynchronous activity to hide them.

For estimating the CPU preparation time of a bin, we need to estimate the time required to acquire the ghost zones and copy the cells into the input array for the GPU. The RTVD kernel requires three ghost zones which are either copied from neighbors or interpolated from the coarser patches. In the worst case, all 26 surfaces of the patches will have to be interpolated. The prepare cost of a patch can then be represented as follows:

$$t^{bin}_{prep} = \sum_{i=1}^{n} K_1 * S_i + \sum_{j=1}^{m} K_2 * S_j + K_2 * V \qquad (3)$$

In this expression, K_1 and K_2 represent the cost of interpolation and copying, respectively, and S_i is the total number of cells interpolated for the ith surface.

Table 2. Estimation of prepare times.

Patch dimension	Predicted copy time (ms)	Actual copy time (ms)	Predicted interpolation time (ms)	Actual interpolation time (ms)	Overall relative difference $(\frac{predicted-actual}{actual})$ (%)
50,62,50	1.719	1.591	2.094	2.027	5.39
51,55,51	1.586	1.449	1.969	1.860	7.43
35,58,44	0.990	0.865	1.429	1.375	7.98
54,27,43	0.695	0.617	1.127	1.050	9.30
48,49,49	1.278	1.115	1.684	1.597	9.20
31,35,33	0.397	0.345	0.713	0.697	6.54
31,43,29	0.429	0.368	0.765	0.747	7.02

The first term sums up the cost to interpolate the n surfaces which do not have neighbors. The second term represents the cost of copying the data into the m surfaces from sibling patches. The third term is the net cost of copying V interior cells into the array. The costs, K_1 and K_2, were obtained using profiling runs. The actual and predicted prepare times for a set of patches of different sizes are tabulated in Table 2. The actual times were obtained by performing 10 different runs and calculating the averages of the times. It can be seen that the estimation function produces predictions that are within 10% of the actual values.

To compute C_1, C_2, and C_3 the patches were solved one at a time in the GPU and along one direction at a time. For example, to compute C_1, each patch was swept only along the X-direction and this execution time was divided by the number of columns of the X-sweep to obtain an estimate of C_1. The C_1-value used in the estimator function is the average value obtained over all patches. A similar process was done for C_2 and C_3. The overall solve time would be the sum of the X-, Y-, and Z-sweep times. Similarly, for K_1 and K_2, the copy and interpolation times were timed separately and were averaged. The patches were prepared one at a time without any OpenMP acceleration. The time taken to interpolate a surface is divided by the number of cells to obtain K_1. The time taken to copy the patch cells to the GPU array is timed and this is divided by the number of cells to obtain K_2. We performed around 10 to 15 runs, and each constant was obtained by averaging over hundreds of patches. We also performed multiple runs over the same set of patches, and found very little or no variation in the constant values obtained across runs.

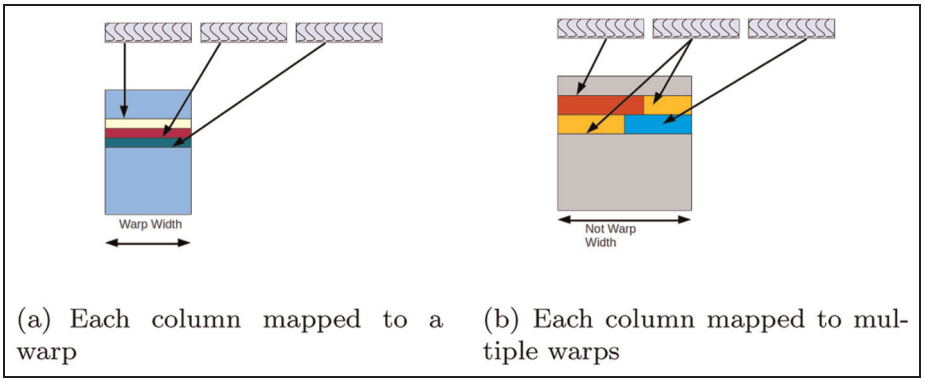
3.3 Multi-GPU optimization

The application is executed on multiple GPU systems by decomposing the domain into different subdomains and assigning a subdomain for an MPI process on a CPU–GPU system. One MPI process runs per node (CPU–GPU system) and integrates the patches in its subdomain by performing GPU kernel invocations. OpenMP threads perform the prepare and close operations of the patches in the subdomain on the CPU cores of a node, while MPI is used for inter-node communications. The patches of the boundary regions of the subdomains are needed by the neighboring MPI processes, and hence the processes communicate the boundary patches at the end of every timestep. In order to hide the overhead due to communication costs between different CPU–GPU systems, we re-order the sequence of computations such that the results of the boundary patches are obtained with high priority. The communication of the boundary data is then overlapped with the computation of bins with interior patches. Thus the MPI processes incur smaller idling times waiting for the boundary patches. The costs due to communication grow with the size of the problem and the scale of the cluster. This optimization ensures that computation costs always dominate the overall execution time and high scalability is obtained.

3.4 Other optimizations

We identified and addressed other potential bottlenecks in the kernel implementation. We optimize to improve the number of coalesced accesses. We implement a method which uses the final output array as a buffer for re-ordered writes after each sweep. Sweeping in the X-direction involves reading and writing in a coalesced manner since the data is in row-major order. The data from the sweep is written into the output array. During the Y-sweep, the data is read from the output array and the updated data is written in a transposed manner into the array for coalesced reading during the Z-sweep. After the Z-sweep, the data is written back to the output array in a coalesced manner. If the final output is in a transposed format, the CPU performs the task of writing the data back in the appropriate manner into the patch structure during the closing phase. To minimize the traffic to global memory as much as possible, we use the fast, read-only constant memory to supply the indices and dimensions of the bins.

When the width of the patch swept is 32, we dynamically organize the thread block as multiple units of 32 threads. The advantage of this design is that each column is acted exclusively by one warp. Thus, there is no



(a) Each column mapped to a warp

(b) Each column mapped to multiple warps

Figure 6. Mapping columns to warps.

need to call synchthreads after reading from or writing to the shared memory, since the warps execute in lock step. If, on the contrary, each warp is made to execute only parts of the columns, then the threads have to periodically synchronize in order to ensure that their necessary neighboring data in the columns are loaded into shared memory by the other threads acting on the same columns. This would involve threads waiting for other threads in the block doing unrelated operations. Our optimization ensures that the number of active warps is high throughout the computation. This is illustrated in Figure 6(a) and (b).

The prepare cost of a patch is an overhead that must be minimized. We fix the minimum dimensions of a patch to ensure that the prepare time of the patch is less than its solve time. We fix the maximum dimensional size of the patch such that it fits in shared memory, while allowing for a good load balance. The range of patch sizes also affects the bin-packing efficiency. We experimentally set the minimum and maximum sizes of a patch (including the ghost cells) along any dimension as 16 cells and 64 cells respectively. Table 3 illustrates the efficiency of bin-packing for a set of patches, and the times taken for bin-packing. The efficiency of packing is the ratio of the total volume used for packing and the total volume of the bins used. The times shown in the last column of the table were obtained by performing five runs and obtaining averages across the runs. For the chosen dimensions, we find that the packing efficiency is at least 84%. We note that this procedure needs to be invoked only when the refinement occurs and the hierarchy changes. Thus, the cost of bin-packing is amortized over a number of timesteps.

The inputs for our benchmarking experiments, for example, to estimate the prepare and solve times, and to judge the efficiency of the bin-packing procedure, consist of sets of randomly generated patches. The input sets are of two classes: stand-alone patches having dimensions chosen from uniform distribution, and patches from randomly generated complete AMR hierarchies. The experiments show similar results for both classes of inputs. To generate the random AMR hierarchy, the procedure starts with a base grid consisting of uniform patches (level 0). The patches from higher levels are formed by choosing an existing patch at random and then 'refining' it by adding a patch, which fits within the selected patch, as its child. This procedure is also used to generate the synthetic inputs for our runtime experiments and is explained in detail in Section 4.1.

Figure 7 shows the distributions of the patch sizes along the X-dimensions for all the levels of the AMR hierarchy with a total of 600 patches across all levels, having patches of size 32^3 in the base level. This corresponds to the last row of Table 3. The sizes along the Y- and Z-dimensions, and for the other number of patches, show similar distributions.

3.5 Putting it all together

The overall flow of the application execution on CPUs and GPUs with the specified optimizations are shown in Figure 8.

4 Experiments and results

4.1 Experimental setup

The tests were run on a Tesla cluster with each node of the cluster containing one Tesla S1070 GPU and 16 AMD Opteron 8378 cores, and a Fermi cluster with each node containing one Tesla C2070 GPU and four Intel Xeon W3550 cores. The prepares and closing steps on CPUs were parallelized using OpenMP. Experiments on both platforms were conducted with four CPU cores and four OpenMP threads per node. We performed both single-GPU and multi-GPU runs. Multi-GPU runs on Tesla were performed by using a single node with four GPUs. Multi-GPU nodes on Fermi were performed by using four nodes with a single GPU per node. We implemented our optimizations related to the GPU kernel, namely, load-balancing with

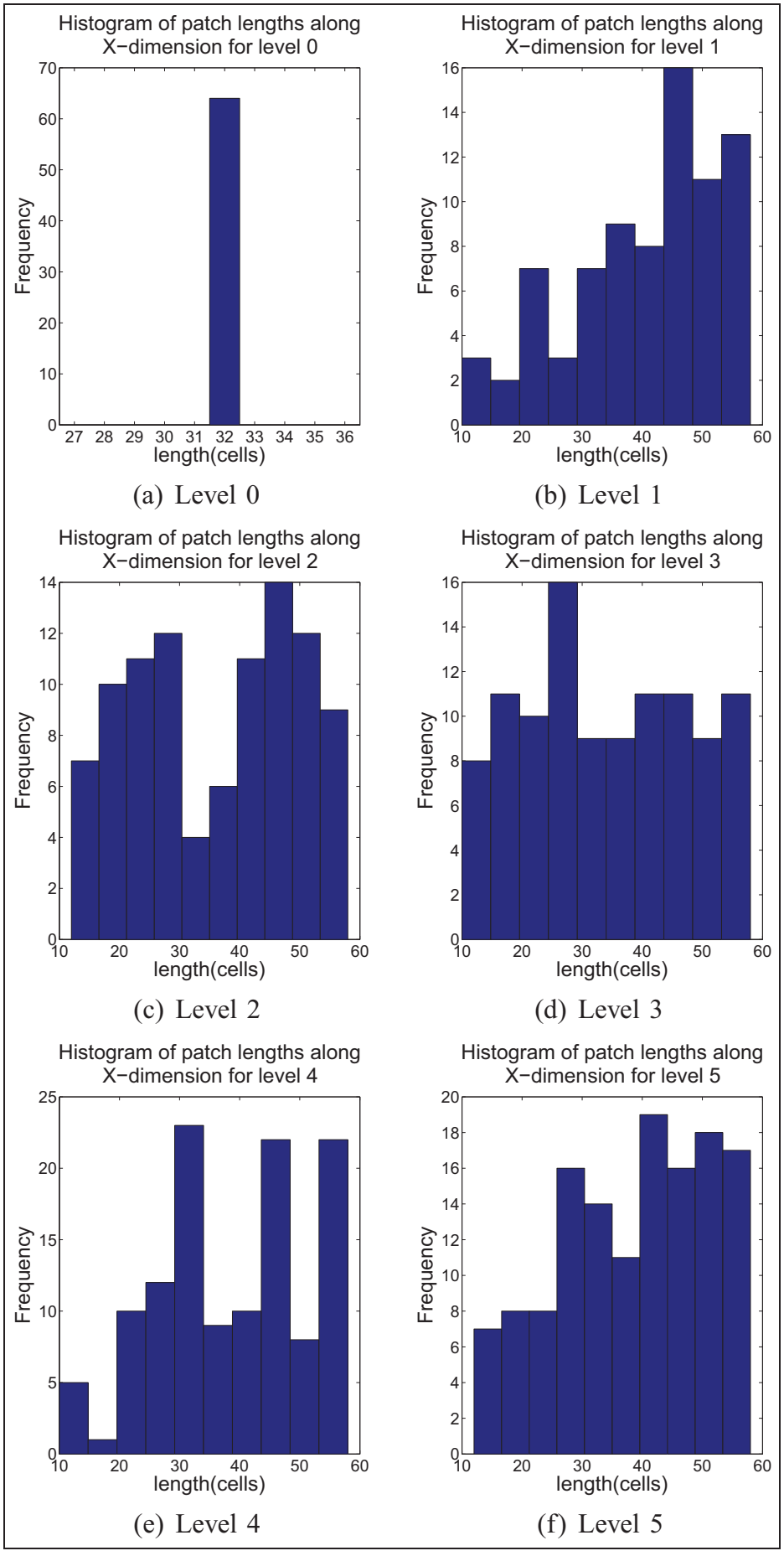


Figure 7. Distributions of patch sizes along the X-dimensions for all levels of AMR hierarchy for 600 patches.

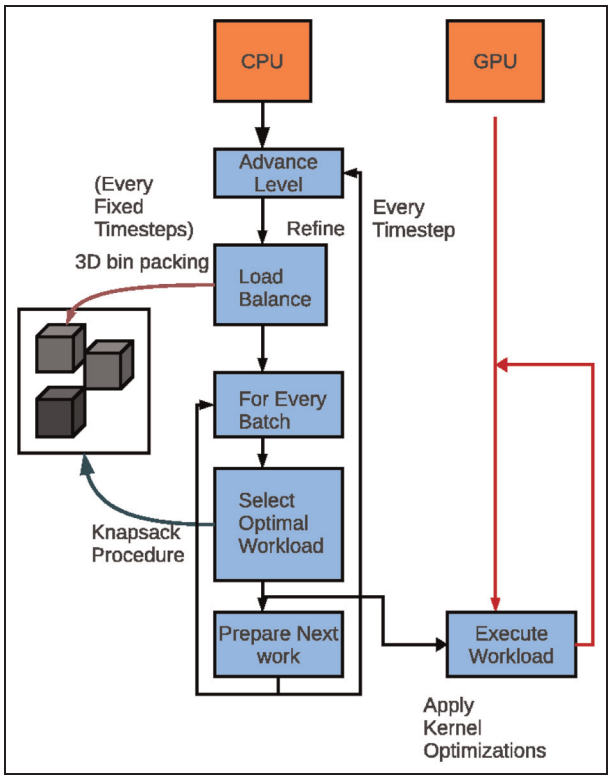


Figure 8. Execution flow.

Table 3. Bin-packing efficiency for the chosen bin width and patch sizes.

Number of patches	Number of bins	Efficiency of packing	Time taken (ms)
120	41	0.8441	2.459
240	67	0.8689	8.668
360	101	0.8574	11.01
480	142	0.8656	17.21
600	178	0.8516	29.61

bin-packing and kernel optimizations, with two versions of solvers, namely, RTVD and TVD solvers. 'Kernel optimizations' refers to the optimizations described in Section 3.4, namely, our methods for improving memory coalescing and avoiding warp synchronization. In our results, we denote the results obtained with load-balancing as *LB* and those obtained with the kernel optimizations as K-Opt. We compared our method that dynamically selects the number and set of patches in a batch for kernel execution with a default scheme in which the total number of patches advanced in GPU per batch (kernel execution), n_{fixed}, is equal to the order of the number of SMs (product of number of streams and number of SMs). The number, n_{fixed}, is 120 for Tesla and 56 for Fermi systems. The default scheme employs one thread block per patch, resulting in load imbalance among threads, while our scheme performs load-balancing using bin-packing of patches. The default scheme also employs four streams to overlap memory transfer with kernel execution. The CPU–GPU asynchronism is achieved by simultaneously preparing and solving consecutive sets of fixed n_{fixed} patches. We denote this asynchronism as a 'static asynchronism'. In our scheme, the asynchronism is dynamic and adaptive since the batch of patches chosen for preparation on the CPU depends on the estimated time of the previous batch asynchronously executed on the GPU.

For a given input set, we apply the default scheme followed by an incremental application of each optimization. In order to comprehensively test the efficiency of our methodology, we tested our scheme using both sets of randomly generated synthetic patches, and real application domains. For synthetic patches, we developed a program that randomly generated N ($N > 64$) patches. Our program generated an initial domain (level 0) of 64 patches of size $32 \times 32 \times 32$. We considered the domain size for the synthetic experiments as $128 \times 128 \times 128$ cells covered with 64 patches organized as a $4 \times 4 \times 4$ grid, with each patch consisting of $32 \times 32 \times 32$ cells. Thus the base region is regularly gridded and all the base patches are uniform. The program then performed $N - 64$ iterations, generating a random refined patch in each iteration. In each iteration, an existing patch is randomly chosen for refinement. The refinement process involves randomly generating a patch having dimensions that would fit within the selected patch. Simultaneously, it is also ensured that the patch dimensions lie within the minimum and maximum values of 16 and 64 respectively along any direction. The dimensions are generated with uniform distribution. The newly generated patch is also checked to ensure that it does not overlap with any other existing ones. The program finally outputs the hierarchy consisting of the patch dimensions, level details, and the parent–child relationships. This is read by the AMR code as input. In our experiments, we show results for six different numbers of patches, namely, 120, 180, 240, 300, 360, and 420. For each number of patches, we ran our above-mentioned program five times to generate five different datasets. We show the average values obtained across these five runs for each number of patches. We show results for different numbers of patches, since the volume of computations in our AMR applications increases with the increase in the number of patches.

For real application data, we considered the patches generated for the Euler application suite found in SAMRAI (Wissink et al., 2001) with the same domain size. For obtaining these traces from SAMRAI, we executed SAMRAI and printed the hierarchy information consisting of the generated patch dimensions to a file.

Table 4. Speedups over CPU-only version for Euler application on a single-GPU Tesla S1070.

Patches	CPU with OpenMP (s)	GPU optimized (s)	Speedup
120	2.6413	0.2958	8.93
180	3.7107	0.3865	9.60
240	6.3279	0.5490	11.52
300	8.2655	0.6787	12.18
360	9.7024	0.7936	12.22
420	11.6559	1.1006	10.59

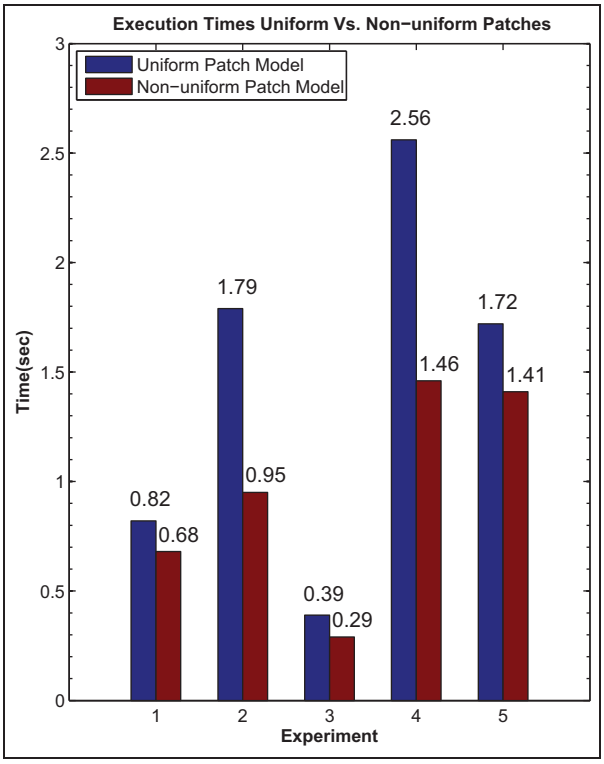


Figure 9. Execution times with uniform and non-uniform patches for the Euler application on a single-GPU Fermi C2070.

This was used in our experiments to generate a similar hierarchy initialized to random data. The purpose was to check our performance for realistic patch sizes and count distributions. We generated three traces by running a 3D sphere problem for 20 timesteps with a constant maximum of six refinement levels. We use pressure gradient and shock as refinement criteria and the Berger–Rigoutsos algorithm for clustering. The interpolation scheme used is linear and the mesh is cell-centered. The three real traces, RT I, RT II, and RT III, consist of 525, 383, and 635 patches respectively.

The following subsections show the results for various experiments with single- and multi-GPU systems using synthetic traces and real applications.

4.2 Speedup over CPU-only implementation

We first show the speedups obtained with our GPU implementation over a CPU-only version. For the latter, the patches are advanced on the CPU cores using OpenMP parallelization. Table 4 shows the times per timestep and the speedups obtained for the Euler application for varying input sizes on the Tesla cluster. The CPU-only version was executed on eight cores. We find that our optimized GPU implementation gives an average speedup of about $\times$ 10 over the CPU-only implementation, thus demonstrating the benefits of GPUs for these AMR applications.

4.3 Comparison with uniform patch models

Figure 9 shows the performance difference in AMR models using uniform and non-uniform patches. The results shown correspond to five execution runs of the Euler application for varying inputs and domain sizes on the Fermi system. For each experiment, the first bar illustrates the result obtained with a fixed patch size of 16^3, while the second bar shows the execution times for a scheme with varying patch sizes. We observe that using non-uniform patches results in 1.21 to 1.88 times less execution time than when using uniform patches. This can be attributed to the additional work due to over-refinement.

4.4 Single-GPU experiments for RTVD solver with synthetic traces

Figure 10 illustrates the average performance improvements observed in the RTVD solver (GPU kernel) timing on a single-GPU Tesla system with synthetic inputs for varying numbers of patches. Each result is an average for five random traces. In the figure, LB refers to the load-balancing optimization, and K-Opt, to kernel optimization. The results show that we obtain average speedup of about 1.47 due to our load-balancing strategy. This high improvement over the default scheme shows the efficiency of our load-balancing with bin-packing and the importance of load-balanced executions when considering AMR with non-uniform patches. The results also show an average overall speedup of about 1.85 for the GPU solver with both the load-balancing and kernel optimizations. In general, we obtain 1.27 to 1.64 speedup with load-balancing and 1.41 to 2.22 speedup considering both load-balancing and kernel optimizations. The average performance improvement shows little variation with increasing numbers of patches, thus demonstrating the scalability of our solver optimizations.

Figure 11 shows the improvement in the overall execution time for a timestep, due to the solver optimizations and with the adaptive asynchronism between CPU and GPU executions. The timestep involves the

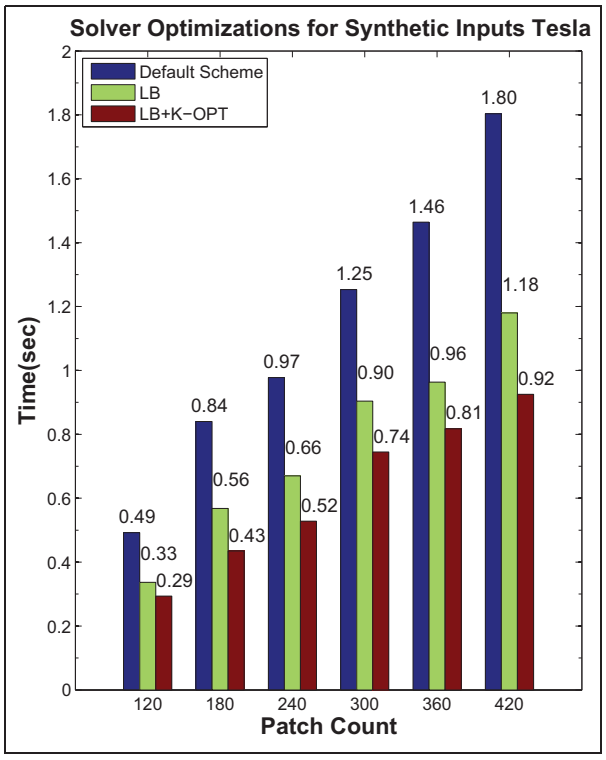


Figure 10. Performance gains in the RTVD solver for synthetic inputs on a single-GPU Tesla S1070.

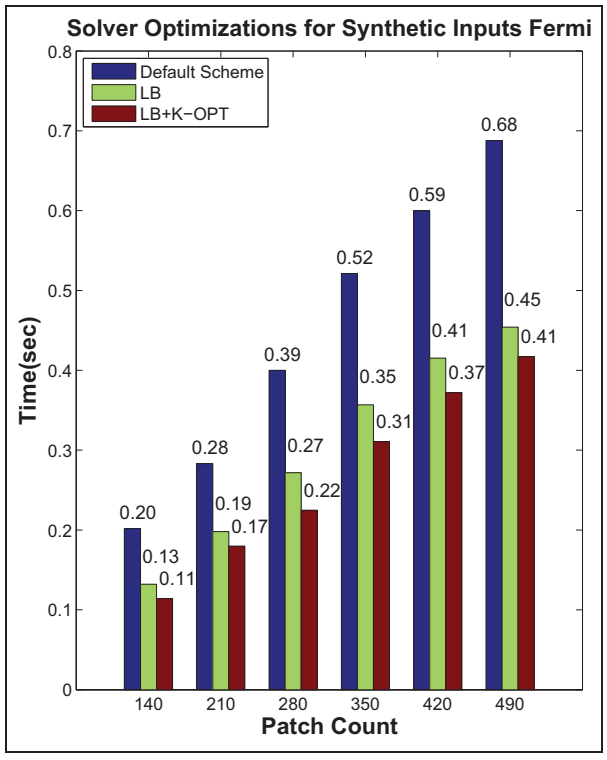


Figure 12. Performance gains in the solver for synthetic inputs on a single-GPU Fermi C2070.

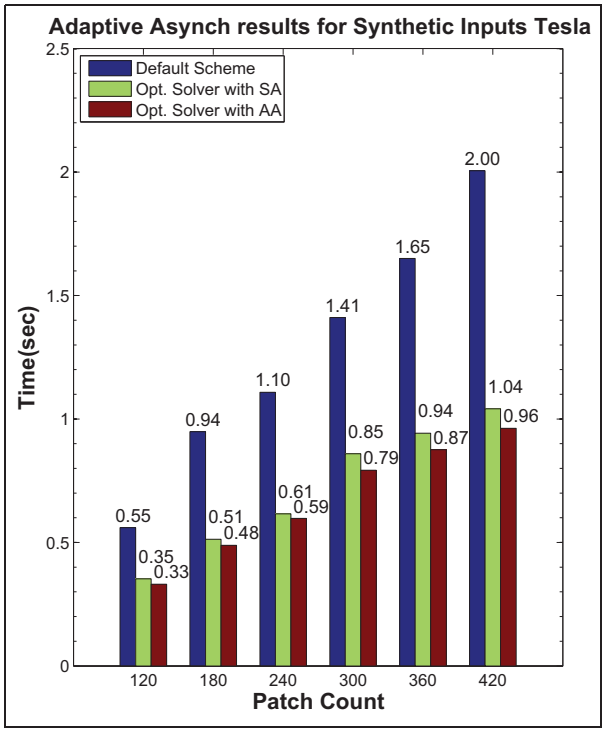


Figure 11. Overall performance gains for synthetic inputs on a single-GPU Tesla S1070.

numerical integrations, preparation, fix-ups, and closing steps. We do not consider mesh refinement since our work focuses on irregular workloads in the integration and preparation steps. In the figure, SA refers to the static default asynchronism and AA refers to adaptive asynchronism. With the static scheme of asynchronism, our method with the optimized solver shows an average speedup of only 1.79 over the default scheme, which is less than the performance improvement due to only the optimized solver shown earlier. This shows that not performing careful asynchronous executions can diminish the performance benefits due to GPU executions. The results also show an average speedup of about 1.85 in the overall execution time due to adaptive asynchronism. In general, we obtain 1.51 to 2.22 speedup with our overall method for the overall application. The gains from adaptive asynchronism are due to two reasons. Mainly, it ensures that the maximum amount of work on the CPU and GPU overlaps thereby hiding the cost of operation. Secondly, it provides as much work as possible to the GPU, reducing the number of kernel invocations and utilizing the memory bandwidth. The average performance improvement shows little variation with increasing numbers of patches, thus demonstrating the scalability of our overall method as well.

Figures 12 and 13 show the results with synthetic traces for the single-GPU Fermi system. The results show that we obtain average speedup of about 1.47 due to our load-balancing strategy and an average overall speedup of about 1.67 for the GPU solver with both the

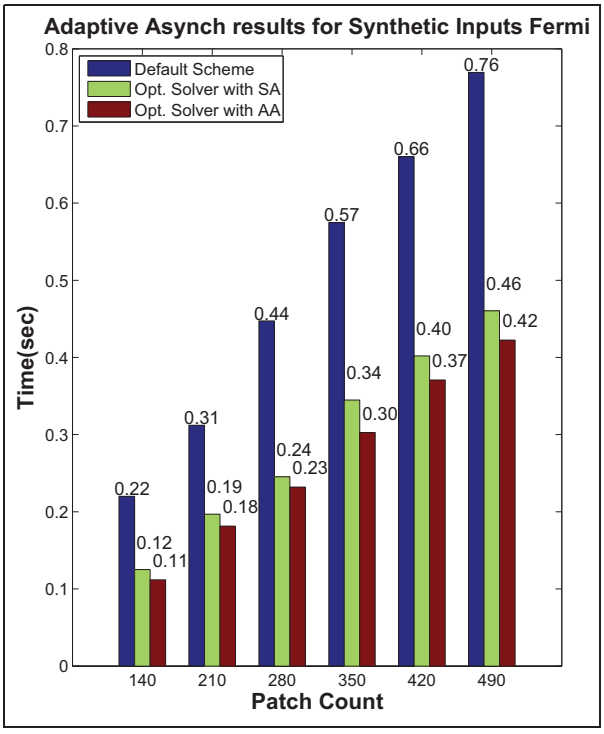


Figure 13. Overall performance gains for synthetic inputs on a single-GPU Fermi C2070.

load-balancing and kernel optimizations. In general, we obtain 1.37 to 1.59 speedup with load-balancing and 1.47 to 1.92 speedup considering both load-balancing and kernel optimizations. The overall improvement in solver timings is lower than what is observed for the Tesla system. The gains due to load-balancing are almost the same, while the net improvement with the kernel optimizations is lower. This could probably be attributed to the improved memory system of Fermi with its cache as well as the facility to execute concurrent kernels from different streams. We observe a similar overall speedup in the range of 2.22 compared to the gains obtained on the Tesla system.

4.5 Single-GPU experiments with real applications

Table 5 gives the performance results on Tesla when our methodology is applied to patches formed in the execution of the three real applications. The table shows both the solve times corresponding to GPU kernel executions, and the overall times for the entire application that include both the solve times on the GPU and the times for preparation and closing of patches on the CPU. We observe gains similar to the results with synthetic inputs. We obtain speedup of at least 1.46 in the performance of the GPU solver with our load-balancing and kernel optimizations, and speedup of at least 1.67 in the performance of the overall applications with our overall method.

Table 6 gives the performance results on Fermi for real traces. We obtain speedup of at least 1.41 in the performance of the GPU solver with our load-balancing and kernel optimizations, and speedup of at least 1.48 in the performance of the overall applications with our overall method. We observe that the overall execution times of both the default and the optimized schemes are lower when compared to the values observed for Tesla. We also observe a lower performance gain in the solver stage compared with that noted for synthetic inputs. These are due to the improved memory system of the Fermi system.

Thus our methods show high performance improvements over the default scheme with both synthetic and real traces on the single-GPU system of both Tesla and Fermi clusters.

4.6 Multi-GPU experiments

Figure 14 shows the execution time results for our multi-GPU experiments with the four-node, four-GPU (one GPU each) Fermi for the RTVD scheme on real traces, with and without our multi-GPU optimization of communication–computation overlap, described in Section 3.3, along with the execution times of the default scheme. The figure also shows the speedups obtained by multi-GPU executions with respect to the results for single-GPU executions shown in Table 6. We find that multi-GPU executions without communication–computation overlap show about a 1.47 speedup over the default scheme for four GPUs. Applying our optimization of communication–computation overlap results in up to 1.16 additional speedup. We also observe speedups of 3.4 without

Table 5. Improvements in RTVD solver time with load-balancing and kernel optimizations, and overall improvements on Tesla S1070 for real traces.

Trace	Default scheme (s)		Scheme with optimized solver (s)				Speedups	
	GPU solver time	Overall time	GPU solver time		Overall time			
(I)	(II)	(III)	with LB (IV)	with LB + K-Opt (V)	with SA (VI)	with AA (VII)	(VIII = (V/II))	(IX = VII/III)
RT I	3.5298	3.8517	2.6092	2.4164	2.6080	2.3023	1.46	1.67
RT II	1.0875	1.1647	0.8148	0.6212	0.6912	0.5798	1.75	2.00
RT III	3.3184	3.6824	2.5821	1.8230	1.9633	1.9446	1.82	1.89

Table 6. Improvements in RTVD solver time with load-balancing and kernel optimizations, and overall improvements on Fermi C2070 for real traces.

Trace	Default scheme (s)		Scheme with optimized solver (s)				Speedups	
	GPU solver time	Overall time	GPU solver time		Overall time			
(I)	(II)	(III)	with LB (IV)	with LB + K-Opt (V)	with SA (VI)	with AA (VII)	(VIII = (V/II))	(IX = VII/III)
RT I	1.2277	1.3250	0.9070	0.7900	0.8816	0.7486	1.55	1.77
RT II	0.2983	0.3283	0.2362	0.2106	0.2257	0.2223	1.42	1.48
RT III	1.0950	1.1757	0.7862	0.7397	0.8084	0.6941	1.48	1.69

communication–computation overlap, and 3.8 with our optimization of communication–computation overlap for four GPUs, over single-GPU executions. The trends in the figure also show that multi-GPU executions with our communication–computation overlap optimization show better scalability than the other two schemes. We observe that we get a speedup of 3.8 and 3.6 for the first and second traces respectively while we obtain only around 2.8 for the third trace. It is also seen for the third trace that there is not any significant increase in speedup while going from three to four GPUs. This is because currently we follow a simple domain decomposition technique to distribute patches, without any load-balancing strategy to balance the work among the MPI processes. However, we note that even in this case we obtain up to 1.56 speedup over the default scheme.

4.7 Comparison with weighted average flux scheme

The weighted average flux (WAF) (Toro, 1992) solver is another directionally split scheme with similar sweeping patterns and a column update model like the RTVD scheme. Thus, our proposed optimizations are also applicable for this scheme. Figure 15 illustrates the performance benefits obtained with the WAF scheme for non-uniform patches on the Tesla system. We observe speedups in the range 1.28–1.41 in the solver times, and up to 1.23 speedup due to load-balancing.

4.8 Comparison with Enzo

Enzo (Wang et al., 2010) is an existing method for AMR applications with non-uniform patches. It implements the TVD scheme for the GPU solver. For comparison with Enzo, we applied our optimizations of bin-packing-based load-balancing and kernel optimizations to the Enzo TVD solver, and used adaptive asynchronism of CPU–GPU executions. We show results for only the Fermi system since we obtained similar results on the Tesla system.

Figure 16 illustrates the average performance improvements observed in the TVD solver timings on the Fermi system for synthetic inputs. The results show that we obtain average speedup of about 1.06 due to our load-balancing strategy and an average overall speedup of about 1.39 for the GPU solver with both the load-balancing and kernel optimizations. Unlike the RTVD case, the load-balancing scheme gives considerably lower performance benefits. We observe that the TVD scheme is by default well load-balanced with thread idling constrained only to the last block. The TVD kernel, however, solves only one patch at a time and thus potentially incurs overheads due to multiple kernel launches, poor utilization when the total number of cells in the patch is lower, and also suboptimal use of the CPU–GPU memory bandwidth. Our 3D bin-packing scheme packs together multiple patches and ensures that the size of the patch being solved substantially utilizes the GPU cores. In terms of kernel optimizations, we note that the default Enzo scheme does not utilize streams to overlap its memory transfers with kernel executions. We implement the same in our model and observe that the major bulk of the gains is due to this optimization.

The Enzo scheme does not overlap operations on the CPU like interpolating ghost zones with the GPU computations. Thus, the default scheme does not have any CPU–GPU asynchronism. Figure 17 illustrates the gains observed due to our adaptive asynchronism scheme on the Fermi system for the Enzo solver for the overall executions. We obtain up to 1.92 speedup for the advance phase.

For real traces, we observe speedup of at least 1.50 in solver times on Fermi for the TVD scheme, as shown in Table 7. In this case, the load-balancing scheme also does considerably better than what was observed for synthetic inputs. We see a speedup of about 1.14 for the second case due to the load-balancing alone. Table 8 shows the overall improvement in execution time when our adaptive asynchronism scheme is applied with the solver optimizations to the real traces on Fermi. We observe that our methods result in speedup of at least 1.98.

We also performed comparisons of multi-GPU executions with Enzo. Figure 18 shows the improvements in multi-GPU executions with four nodes for real

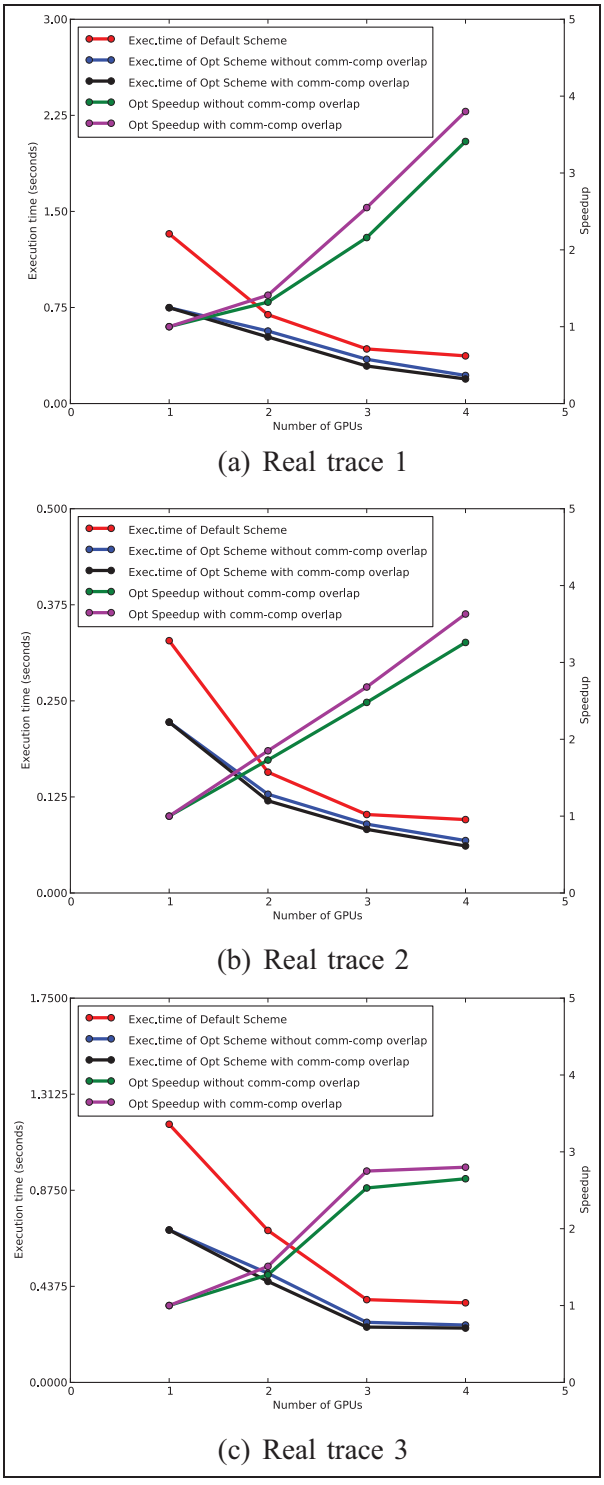


Figure 14. Multi-GPU performance improvement for the RTVD scheme on Fermi C2070 for real traces with four nodes.

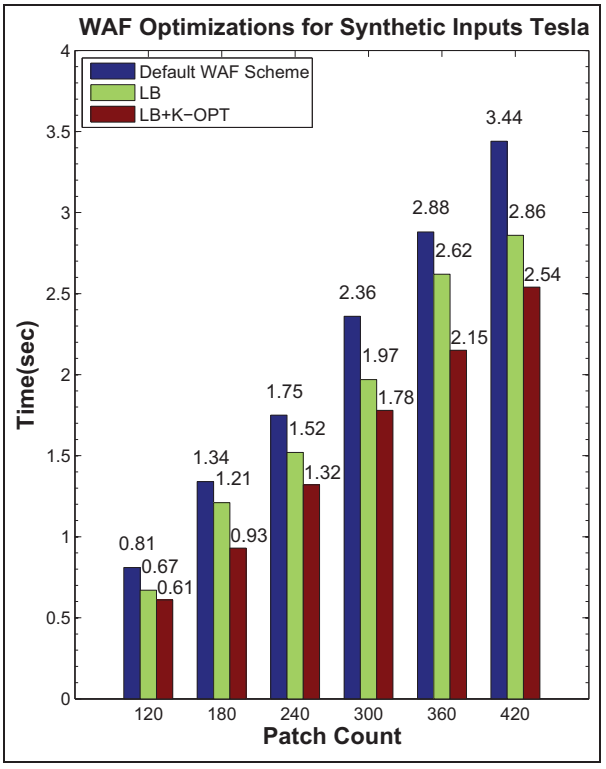


Figure 15. Performance gains for WAF solver on single-GPU Tesla S1070.

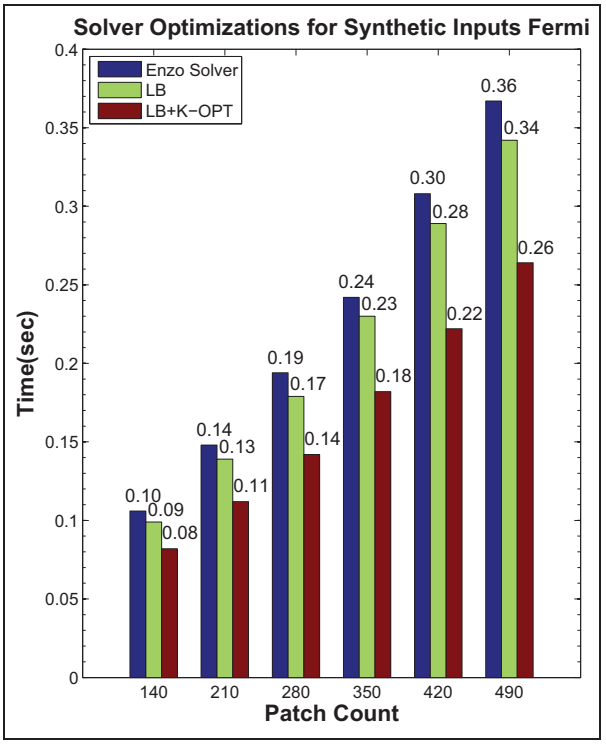


Figure 16. Performance gains in the TVD solver for synthetic inputs on single-GPU Fermi C2070.

traces on the Fermi system. We find that our multi-GPU optimizations obtain an overall speedup of at least 2.04 over Enzo multi-GPU executions. The optimized scheme with communication–computation overlap achieves up to a 3.34 speedup while the optimized

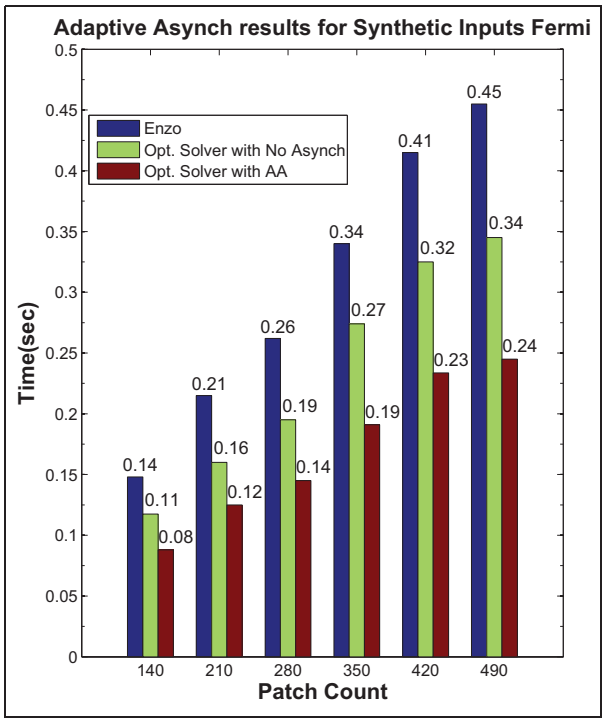


Figure 17. Overall performance gains for synthetic inputs on single-GPU Fermi C2070.

Table 7. Improvements in TVD solver time with load-balancing and kernel optimizations on Fermi C2050 for real traces.

Trace	Enzo solver (s)	LB (s)	LB + K-Opt (s)	Speedup
RT I	0.7063	0.6348	0.4716	1.50
RT II	0.3242	0.2828	0.1358	2.39
RT III	0.6460	0.5449	0.4159	1.55

Table 8. Overall improvements for TVD scheme on single-GPU Fermi C2050 for real traces.

Trace	Enzo (s)	Opt. solver with no asynchronism (s)	Opt. solver with AA (s)	Speedup
RT I	0.9662	0.6913	0.4880	1.98
RT II	0.4455	0.1998	0.1392	3.2
RT III	0.8996	0.6176	0.4248	2.12

version without the overlap method obtains only 2.92. We see a similar degradation in performance for the third trace as observed for the RTVD scheme, since the same data distribution strategy is followed in both cases.

5 Related work

While there have been many works dealing with AMR for single- and multi-CPU environments (MacNeice et al., 2000; Wissink et al., 2001; Deiterding, 2005), the works dealing exclusively with block-structured AMR for GPGPUs are limited, to the best of our knowledge. The typical CPU-based optimization techniques cannot be trivially extended for efficient execution on GPUs. Most efforts on load-balancing for the CPU focus on

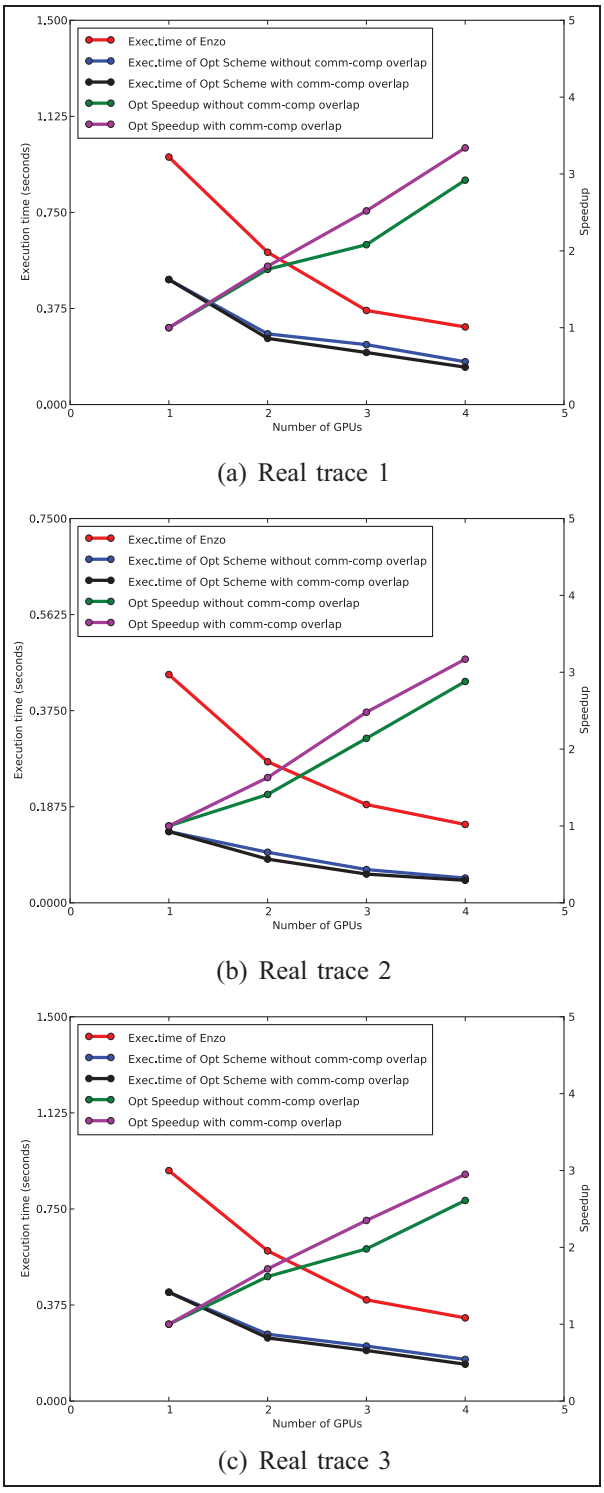


Figure 18. Multi-GPU performance improvement for the TVD scheme on Fermi C2050 for real traces with four nodes.

dividing the workload among various distributed processors while keeping the communication volume minimal (Aluru and Sevilgen, 1997).

While dealing with GPU systems, the priority is to use the architectural aspects for efficient execution. GAMER (Schive et al., 2010) is a GPU astrophysics codebase which implements a hydrodynamics-based AMR scheme. GAMER avoids the problem of load-balancing by fixing all patches to be of the same size. Uniform patches can lead to over-refinement of regions, and hence result in increased computational costs. Enzo (Wang et al., 2010) deals with a similar large-scale application which implements GPU solvers which can deal with arbitrary patch sizes. While Enzo deals with arbitrary patches, it fails to exploit concurrency between the CPU and GPU, and involves a large number of GPU kernel invocations. Our experiments have shown that our adaptive and asynchronous strategies show large improvements over the Enzo model.

Uintah (Humphrey et al., 2012) is an object-oriented framework for solving fluid–structure interaction problems which recently proposed a CPU–GPU version that uses a task-graph-based approach for scheduling computations. The runtime system handles the details of asynchronous memory copies to and from the GPU. Carpet (Blazewicz et al., 2012) is another such computational framework for massively data parallel codes based on the Cactus codebase. In our work, we consider directionally split hydrodynamics solvers to optimize, which require a different methodology from Uintah's and Carpet's applications. For these solvers, Uintah and Carpet do not address issues such as thread idling due to irregular workloads. When Uintah uses the tiled algorithm for refinement, the patches generated are regular which could potentially lead to over-refinement.

6 Conclusions and future work

In this work, we propose optimizations for the efficient execution of block-structured AMR applications with arbitrary patch sizes, on GPGPUs. We proposed a novel solution based on 3D bin-packing that serves as a load-balancing technique while simultaneously reducing thread idling. We designed a scheme for exploiting the asynchronism between the CPU and the GPU by adaptively selecting the workload at each phase. Our optimizations for multi-GPU platforms include reordering the computations to compute the boundary patches first for overlapping its communication with the computation of the interior patches. Our experiments with synthetic and real data, for single-GPU and multi-GPU executions, on Tesla S1070 and Fermi C2070 clusters, show that our strategies result in up to 3.23 speedup in performance over existing strategies. Our bin-packing-based load-balancing gives performance gains of up to 39%, kernel optimizations give an improvement of up to 20%, and our strategies for adaptive asynchronism between CPU–GPU executions give performance improvements of up to 17% over default static asynchronous executions. In the future, we would like to explore making the bin sizes adaptive and introduce more dynamic schemes for assigning thread blocks to workloads.

Acknowledgements

We would like to thank Dr. Bobby Philip of Oak Ridge National Laboratory for his significant contributions and very useful comments on improving the quality of the paper.

Funding

This research was partly supported by the NVIDIA CUDA Research Center (CRC) award and by the Centre for Development of Advanced Computing research project (Award number: CDAC/ESE/STV/0016).

References

Aluru S and Sevilgen F (1997) Parallel domain decomposition and load balancing using space-filling curves. In: *Proceedings of the fourth international conference on high-performance computing*, pp. 230–235.

Berger M and Oliger J (1984) Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics* 53(3): 484–512.

Blazewicz M, Brandt S, Diener P, et al. (2012) A massive data parallel computational framework for petascale/exascale hybrid computer systems. In: De Bosschere K, et al. (eds) *Applications, Tools and Techniques on the Road to Exascale Computing* (*Advances in Parallel Computing, vol. 22)*. Clifton, VA: IOS Press.

Crainic T, Perboli G and Tadei R (2008) Extreme point-based heuristics for three-dimensional bin packing. *INFORMS Journal on Computing* 20(3): 368–384.

Deiterding R (2005) Detonation structure simulation with AMROC. In: *Proceedings of high performance computing and communications international conference, HPCC* 2005, pp. 916–927.

Fryxell B, Olson K, Ricker P, et al. (2000) FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series* 131(1): 273.

Garey M and Johnson D (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, USA: W. H. Freeman & Co. ISBN: 0716710447.

Humphrey A, Meng Q, Berzins M, et al. (2012) Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system. In: *Proceedings of the first conference of the extreme science and engineering discovery environment (XSEDE'12)*.

LeVeque RJ (2002) *Finite Volume Methods for Hyperbolic Problems* (Cambridge Texts in Applied Mathematics). Cambridge: Cambridge University Press.

MacNeice P, Olson K, Mobarry C, et al. (2000) PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications* 126: 330–354.

Martello S and Toth P (1990) *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY: John Wiley & Sons.

NVIDIA (2014) C Programming Guide Version 6.0. Available at: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

Schive HY, Tsai YC and Chiueh T (2010) GAMER: A GPU-accelerated adaptive mesh refinement code for astrophysics. *Astrophysical Journal Supplement Series* 186: 457.

Toro E (1992) The weighted average flux method applied to the Euler equations. *Philosophical Transactions of the Royal Society of London Series A: Physical and Engineering Sciences* 341(1662): 499–530.

Trac H and Pen U (2003) A primer on Eulerian computational fluid dynamics for astrophysics. *Publications of the Astronomical Society of the Pacific* 115(805): 303–321.

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

Wissink A, Hornung R, Kohn S, et al. (2001) Large scale parallel structured AMR calculations using the SAMRAI framework. In: *Proceedings of the 2001 ACM/IEEE conference on supercomputing* (CDROM).

Ziegler U (2008) The NIRVANA code: Parallel computational MHD with adaptive mesh refinement. *Computer Physics Communications* 179(4): 227–244.

Author biographies

Hari K Raghavan obtained his MSc(Engg) degree from the Supercomputer Education and Research Centre, Indian Institute of Science, in 2013. He received his BTech degree in Computer Science and Engineering from the National Institute of Technology, Tiruchirapalli. He is currently employed at MathWorks India.

Sathish Vadhiyar is an Associate Professor in the 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 PhD from the Computer Science Department of the University of Tennessee, USA, in 2003. His research areas are in building application frameworks including runtime frameworks for irregular applications, hybrid execution strategies, and programming models for accelerator-based systems, processor allocation, mapping and remapping strategies for Torus networks for different application classes including irregular, multi-physics, climate and weather applications, middleware for production supercomputer systems, and fault-tolerance for large-scale systems. Vadhiyar is a member of the IEEE and has published papers in peer-reviewed journals including JPDC, CPE, and IJHPCA, and at conferences including SC, IPDPS, ICS, HPDC, ICPP, HiPC, and CCGrid. He is an Associate Editor of IEEE Transactions on Parallel and Distributed Systems (TPDS). He was a tutorial chair in eScience 2007, and session chair in eScience 2007 and ICS 2013, and has served on the program committees of conferences related to parallel and grid computing including IPDPS, ICPP, CCGrid, eScience, and HiPC. He has won awards including the NVIDIA Innovation Award in 2013, the Yahoo! Faculty Research Award in 2011, the Indian National Academy of Engineering (INAE) Young Engineer Award in 2009, and a University of Tennessee citation for Extraordinary Professional Promise in 2003.