Daino: A High-level Framework for Parallel and Efficient AMR on GPUs

Mohamed Wahib, Naoya Maruyama
RIKEN Advanced Institute for Computational Science
JST, CREST
Kobe, Japan 650–0047
Email: {mohamed.attia,nmaruyama}@riken.jp

Takayuki Aoki
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku
Tokyo, Japan 650–0047
Email: taoki@gsic.titech.ac.jp

is a method of adapting the accuracy of a solution within certain sensitive or turbulent regions of simulation, dynamically and during the time the solution is being calculated.

***Abstract*—Adaptive Mesh Refinement methods reduce computational requirements of problems by increasing resolution for only areas of interest. However, in practice, efficient AMR implementations are difficult considering that the mesh hierarchy management must be optimized for the underlying hardware. Architecture complexity of GPUs can render efficient AMR to be particularity challenging in GPU-accelerated supercomputers. This paper presents a compiler-based high-level framework that can automatically transform serial uniform mesh code annotated by the user into parallel adaptive mesh code optimized for GPU-accelerated supercomputers. We also present a method for empirical analysis of a uniform mesh to project an upper-bound on achievable speedup of a GPU-optimized AMR code. We show experimental results on three production applications. The speedups of code generated by our framework are comparable to hand-written AMR code while achieving good and weak scaling up to 1000 GPUs.**

***Keywords*—Accelerator processing, Adaptive mesh refinement, Parallel programming, Performance analysis.**

存在

解决

I. Introduction

In many scientific and engineering simulations, Partial Differential Equations (PDEs) are solved in a uniform mesh arrangement by using finite difference schemes, referred to as iterative stencils. Typically, the resolution of the mesh is uniformly set to the highest resolution to provide accurate solutions. For meshes that require only high resolution for some portions of the mesh, an alternative method, known as Adaptive Mesh Refinement (AMR), can be used instead of the uniform mesh. The AMR method solves the problem on a relatively coarse grid, and dynamically refines it in regions requiring higher resolution. However, AMR codes tend to be far more complicated than their uniform mesh counterparts due to the software infrastructure necessary to dynamically manage the hierarchical mesh framework. Despite this complexity, it is generally believed that future applications will increasingly rely on adaptive methods to study problems at unprecedented scale and resolution. In fact, the US DoE report on exascale challenges named AMR as an important yet challenging strategy for multi-scale problems [1].

Discrete GPU accelerators can be used to significantly speed up a wide range of applications in a multitude of different scientific areas. However, achieving high-performance for applications ported to GPU is a notoriously difficult task. One reason for the difficulty is the lack of high-level unified programming models, which in practice forces the programmer to use a distributed memory technique, e.g., message passing, in addition to a low-level GPU-specific abstraction, e.g., CUDA. Another source of difficulty is the architecture-specific considerations that have significant impact on the performance such as the data movement along the memory hierarchy. Therefore, despite the potential performance gain made possible by GPU-accelerated systems, the programming complexity can be discouraging to non-experts.

Implementing efficient adaptive meshes in GPU-accelerated systems is significantly hard in comparison to traditional CPU systems. More specifically, GPUs add complexity overhead for managing the mesh hierarchy and optimization of data movement. This is made evident by the relatively wide use of AMR in CPU in comparison to GPU-based systems. For example, a mature AMR framework supporting CPU, namely FLASH [2] is reported to be in use by dozens of production applications [3]. On the contrary, a few number of individual applications adapted AMR solvers for GPU, with varying levels of optimization and scaling [4], [5], [6], [7]. Among the mature AMR frameworks that support CPU [8], *Enzo*, *Cactus*, and *SAMRAI* have extended their support to a few select of their solvers to GPU, mainly for experimentation ([9], [10], [11], respectively).

To the authors knowledge, there are two frameworks that provide support for GPU in AMR applications. The first is *GAMER* [12], a framework originally supporting CPU and later extended to GPU. GAMER provides an interface that allows for programmer-defined finite difference solvers. However, it is the programmer's duty to provide CUDA execution parameters and an optimized GPU kernel to calculate a single mesh patch. In addition, the patch-based rectangular domain decomposition can lead to a serious load-balance issue in AMR simulations, especially when the distribution of refined grids is highly inhomogeneous. The second framework, *Uintah* [13], supports GPU for AMR applications in the context of providing a runtime system for execution of tasks in heterogeneous environments. Uintah is most notable for optimizing heterogeneous parallelism and dynamic load balancing in Uintah runtime system. Similar to GAMER, Uintah requires the programmer to write his own target-optimized version of the solver. In addition, sharing data structures for task mapping between CPU-GPU can lead to a bottleneck due to the complex and frequent communication between CPU and GPU [14]. Finally, the complexity in managing the tasks by the runtime system,

不理解

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

i.e., the multi-stage queuing system, adds a significant overhead that would be detrimental to many scientific applications that are not necessarily comprised of workflow-like tasks.

To summarize the problems, only a few frameworks enable automated AMR transformations for GPU, and their programming models require the programmer to write his own versions of the target-optimized solvers. Moreover, there can be scalability limitations caused by the overhead of the CPU-GPU communication schemes in those frameworks. Finally, there is a lack of performance analysis/models in black box AMR frameworks. We argue that performance models are necessary for helping the programmer in quantifying the efficiency of the generated AMR solutions in terms of the measured speedup versus the estimated achievable speedup.

Solving the above problems is the target and contribution of this paper. To that end, we present a high-level framework called *Daino*. Daino specializes in enabling efficient and scalable structured AMR solutions to scientific applications running on GPU-accelerated systems. Daino uses a high-level programming model that provides an architecture-neutral programming interface and adopts an AMR strategy that would eliminate any CPU-GPU communication schemes that can limit scalability.

Our target scientific applications are those that employ structured finite difference methods in solving PDEs. The main concerns when delivering efficient AMR in GPUs are overheads of managing the mesh hierarchy and the frequent transfer of mesh application data between CPU-GPU. To address the first problem, i.e., management of the mesh, we use fully distributed algorithms for managing the mesh. That is, the data structures representing the mesh hierarchy use distributed encoding and storage, and all the operations required to manipulate and advance the mesh through the simulation are distributed as well. To address the second problem, i.e., CPU-GPU data transfer, we use a data-centric approach at which the CPU is specialized in managing the data structures representing the mesh hierarchy, while AMR-specific routines that operate on mesh application data are executed on the GPU. Hence keeping the mesh application data arrays on the GPU memory for the entirety of the simulation.

The programming model of Daino is designed to be exposed in an architecture-neutral manner; the programmer has no knowledge of the underlying architecture. We provide the programmer with a set of C language directives to identify the stencil functions and data arrays in a logically fixed and uniform mesh implementation of solver(s). The programming interface enables the underlying compiler-based framework to statically analyze the solvers, construct the adaptive mesh hierarchy, automatically parallelize the mesh partition over distributed memory, and apply optimizations required for keeping the mesh application data, i.e., stencil arrays, in GPU memory throughout the simulation. We also present a performance analysis that we use to inform the programmer of how much speedup is achieved by Daino in comparison to the practical limit possible for the application in hand.

Daino uses a compiler-based approach to generate AMR code optimized for GPU-accelerated systems. The generated code uses CUDA for the GPU and MPI for the message passing. The framework leverages the capabilities of LLVM infrastructure [15] by using the front end to analyze the annotated programmer code and then apply various transformations and optimizations as compiler passes on LLVM IR (Intermediate Representation). Finally, we use the back end code generator for the target architecture to produce object files that are later linked to be executable code. The framework can be extended in the future to support other targets. For instance, we show how x86 CPU could be targeted by applying target-specific passes to invoke the OpenMP runtime library, then use the back end code generator for the target architecture.

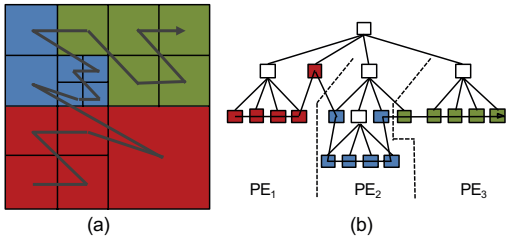

Fig. 1: Octree-based meshes. The blocks are equally partitioned into three PEs using a space filling curve. (a) Adaptive mesh (b) Tree representation (Note: 2D quadtree is used for illustration)

We evaluate our framework by transforming three production applications and showing the achieved speedups using TSUBAME2.5 supercomputer at Tokyo Tech. We present results of speedup using up to 1000 Nvidia Kepler K20x GPUs, and demonstrate that our framework can achieve speedup comparable to hand-written AMR versions (in both strong and weak scaling cases). We also demonstrate the efficiency of Daino automated transformations by comparing the empirical speedups to the maximum achievable speedup we identify by our performance analysis. Finally, we demonstrate the productivity by highlighting the lines of codes of hand-written AMR implementations in comparison to the negligible lines of code required for auto-generating AMR code by our framework.

II. Background

Structured AMR methods are rooted in the work of Berger et al. [16], where logically-rectangular meshes are used in the implementation of the adaptive mesh. Structured AMR utilizes a hierarchy of levels of spatial, and often temporal, mesh spacing with each level being composed of a union of logically rectangular mesh regions. There are two common implementation strategies that differ with respect to management of the mesh hierarchy: *patch-based* and *tree-based*. Patch-based AMR operates on the granularity of the smallest mesh unit (known as *cell*), i.e., each cell is assessed for refinement individually. Rectangular patches are then constructed to contain all cells at the same level. On the other hand, tree-based AMR divides the domain into fixed *blocks*[1]. If any cell within a block requires refinement, the whole block is refined. It is well acknowledged that octree-based AMR requires less programming effort and is computationally more efficient than patch-based AMR, with respect to management overhead and

[1]A *block* in this paper is a leaf octant in the octree. Not to be confused with patch-based AMR that often refer to patches as blocks.

memory requirements. In this paper we focus on the tree-based AMR. Nonetheless, the techniques described in the paper can be adapted to fixed size patch-based AMR with relative ease.

In the tree-based scheme, e.g., [17], the mesh is organized into a hierarchy of refinement levels. The mesh is usually decomposed into relatively small fixed-sized octants of mesh cells. Each octant can be recursively refined into a set of octants of fine cells. The mesh configuration is managed using a tree-based data structure that maintains explicit child-parent relationships between coarse and fine octants. Size relations between neighboring octants are typically enforced in structured AMR, which means neighboring octants can have at most one level of refinement difference (referred to as *2:1 balance*). An important feature of octrees is that the traversal of an octree across its leaves corresponds to a Morton z-shaped Space Filling Curve (SFC) in the geometric domain [18]. Accordingly, sorting the blocks by their Morton ID and equally partitioning them leads to a uniform distribution of the blocks of a mesh among different Processing Elements (PEs), while benefiting from the locality provided by SFC affinity. Figure 1 illustrates how the domain and tree are represented in AMR, and the use of SFC to divide the blocks among three PEs.

III. Adaptive Mesh Refinement in the Framework

This section introduces the tree implementation used in Daino, discusses the AMR components in Daino, and overviews the algorithms that are used. Modern octree-based AMR methods use a distributed tree [19] and adapt the mesh in a parallel fashion [6] with minimum inter-node communication. In the distributed node model of the tree, that we also use in Daino, each node has a local view of the octree that includes the octants residing on the node, while octants residing on other nodes are pruned in the local octree.

Our AMR method builds on a foundation of previous work on parallel octrees [17]. The AMR method used in Daino has three main components. First, advancing the simulation using some finite difference scheme. Second, error estimation and consequent mesh adaptation. Third, load balancing by uniformly redistributing the blocks over the PEs. These components are interleaved to correctly and conservatively advance the simulation on the adaptive hierarchy. To initialize AMR, the error estimation drives the hierarchy generation procedure starting from the uniform mesh specified by the user. Once the hierarchy is created, the main loop of the simulation advances the time loop using the time integration algorithm. As the simulation proceeds, and at a given frequency, the error estimation procedure is invoked and the mesh is accordingly adapted. The load balancing component is typically invoked at much less frequency.

In this paragraph we overview the algorithms used to manage the adaptive mesh and load balance. The `Initialize` algorithm creates a uniformly refined mesh. In the case of Daino, the programmer provides an initialized uniform mesh. We treat the uniform mesh as an adaptive mesh at which all octants are set to the maximum refinement level. Note that the initialization algorithm can easily be extended to use pre-refined meshes as an input. The `Refine` algorithm replaces an octant with eight child octants based on the error estimation, once or recursively. The `Coarsen` algorithm replaces eight sibling octants with a common parent octant, once or recursively. The `2:1 Balance` ensures the 2:1 size relation between neighbors is respected. `Ghost` pack/unpack and exchange the ghost layers for the blocks having neighbors residing on a different GPU. `Load balance` redistributes the blocks evenly on the nodes. The `Initialize`, `Refine`, and `Coarsen` are performed on local data and require no communication for neither inter-node nor intra-node with multi-GPUs. The communication for `Ghost` and `2:1 Balance` algorithms scales with the number of blocks who have neighbors on other nodes. Finally, `Load balance` algorithm requires an *Allgather* operation for a small number of bytes for querying the loads. In addition, *P2P* communication is required to migrate blocks from the over-loaded nodes at a volume proportional to the total number of blocks.

A. Data-centric AMR in GPU-based Systems

Typical GPU AMR implementations only offload the stencil operations to the GPU, while the mesh adaptation is done on the CPU [4], [5], [6]. This requires moving the stencil arrays between the CPU and GPU every time the mesh is adapted, which can become a bottleneck. A communication and computation overlap scheme, proposed by GAMER [5] can resolve the problem, albeit imperfect effectiveness of the overlap approach [20]. One of the main features in Daino is as follows: we base our GPU implementation on a data-centric approach that avoids data movement by moving all the operations touching stencil arrays to be natively executed on the GPU. In addition to the stencil kernels, the following operations are also executed on the GPU: error estimation, refinement, coarsening, and optional post-correction operations. On the other hand, the CPU specializes on: a) managing the octree data structures, b) applying inter-node load balancing heuristics, and c) orchestrating the simulation on the intra-node level by managing the execution plan in a multi-GPU environment.

It is important to note that the GPU is oblivious of the octree data structure residing on the CPU; the GPU has no knowledge of the location and relations of the octants in the mesh. We argue that this specialization is conceptually aligned with the hierarchical nature of modern HPC systems. Especially when considering the trend of increasing the number of accelerators per nodes in accelerator-based systems; CPUs are expected to have a bigger role in orchestrating and less in actual computation (e.g., DoE SUMMIT supercomputer is expected to include 8 to 16 GPUs per node). Finally, there are recent attempts to introduce the data-centric approach in GPU-based AMR [7], [11], [20]. However, the cited attempts were tested on specific individual applications and limited to single node execution ([7], [20]). Daino is a generic framework that builds on the data-centric approach to produce scalable AMR codes. Specifically, compared to a our earlier work [20], our framework enables large-scale runs by supporting: a) multi-node, b) multi-GPU, c) load balancing, and d) automation.

IV. High-level Framework for Efficient AMR

We design a high-level programming framework that provides a highly productive programming environment for AMR. The framework is transparent and requires minimal involvement from the programmer, while generating efficient and

Listing 1: Minimal Example of Using Directives in Daino

```
1 #pragma dno kernel
2 void func(float ***a, float ***b, ..) {
3 #pragma dno data domName(i, j, k)
4 a, b;
5 #pragma dno timeloop
6 for(int t; t< TIME_MAX;t++) {
7 for(int i; i<NX; i++)
8  for(int j; i<NY; j++) {
9     ... // comput. not related to a and b
10    for(int k; k<NZ; k++) {
11     a[i][j][k] = c*(b[i-1][j][k] + \
12      b[i+1][j][k] + b[i][j][k])+ \
13      b[i][j+1][k] + b[i][j-1][k]);
14 } } } }
```

scalable AMR code. The framework consists of a compiler and runtime components. A set of directives allows the programmer to identify stencils of a uniform mesh in an architecture-neutral way. The uniform mesh code is then translated to GPU-optimized parallel AMR code, which is then compiled to an executable. The runtime component encapsulates the AMR hierarchy and provides an interface for the mesh management operations.

A. Design Goals

Our design goals for providing programming support for Daino are as follows:

Minimal programming interface. The interface should be intuitively familiar to programmers and should not change existing language primitives. The interface must balance this simplicity against the requirements for expressiveness.

Efficiency. Code automatically inserted by the compiler to manage the adaptive mesh hierarchy, orchestrate node-level execution environment, and inter-node load balancing can entail a performance penalty that exceeds the benefits of an adaptive mesh. Target-specific optimizations should be implemented to reduce the expected overheads.

Modular Implementation. Future support of other target architectures is a key point in the design of Daino. To maximize the flexibility, the compiler and runtime should be designed in a modular fashion.

B. Programming Model

The Daino framework provides directives to be used with standard C (details on directives in next section). The programmer is required to add the directives to a serial uniform mesh code in order to identify the operations and stencil data arrays that are the target for transformation. Note that the directives are not changing the uniform mesh implementation; the programmer can still use the uniform mesh implementation if the directives are ignored by compiler. From the programmer's perspective, using Daino for transforming an application can be described in the following steps.

Step 1: Provide a serial implementation for a stencil program that uses a uniform mesh.

Step 2: Use directives to specify the functions that include stencil operations applied on the mesh. Those functions would later be transformed to CUDA kernels executed on the GPU. In addition, the programmer specifies the data arrays of the stencils and their iterators.

Step 3: Invoke a shell script that manages end-to-end code compilation to produce executables for the programmer.

Step 4: Daino includes a list of refinement interpolation schemes to choose from. Similarly, the programmer can choose from different coarsening and error estimation functions. However, there can be cases, specially for the error estimation, when the programmer would like to use his own function. In such case, the function should be provided by the programmer. We are currently extending the framework programming interface to support the transformation of user-defined functions to hook them with the AMR code.

Step 5: Provide a parameter text file that includes the choice of the refinement functions mentioned in the previous step and values for the AMR runtime parameters such as frequency of meshing, maximum and minimum refinement levels, ..., etc.

Step 6: *Optional.* Run efficiency experiments by setting an environment variable to run the performance analysis with the code (details in Section VI).

1) Using Directives in Daino: Daino transforms stencil code from uniform to adaptive mesh. However, it is most likely that production applications would include non-stencil code in the program. Our goal is minimum involvement from the programmer, and hence we provide him with an interface to identify the stencil code without changing the stencil code itself or being exposed to the target architecture. To enable Daino's translator to identify a stencil operation, we require the programmer to annotate the following: the C function that includes the stencil operations, the data arrays and iterators of the stencil operations, and the time integration loop. The directives in Daino are as follows:

```
#pragma dno kernel
// declaration of a target function
#pragma dno data clause_list(attributes)
// symbols that share the attributes
// defined here, separated by comma
#pragma dno timeloop
// the time loop
```

We elaborate on using the directives and why each directive is required by using a simple example (shown in Listing 1). We require the programmer to enclose code that applies a stencil operation(s) in a C function. A function annotated by using the *kernel* clause would typically be translated into a device kernel (as shown in lines 1 and 2 in the code listing). The programmer is also required to specify the names of the data arrays that are target as well as the iterators used with those data arrays. As shown in lines 3 and 4, arrays a and b are target arrays accessed using the i, j, and k iterators (identified using the *domName* clause). The rationale for requiring the identification of the data arrays arrays and the iterator is to be enable better flexibility; the programmer can include stencil-independent and/or iterator-independent code in the function. For instance, in the code example, line 9 includes stencil-independent code that uses the iterators used with the arrays a and b, which means the framework should account for the change in iterators in that code but not for the mesh management operations. In one of the applications that we discuss in the results section, a phase-field simulation, this was particularly useful considering that the phase-field function

consists of a relatively large number of complex nonlinear terms that are not all stencil related. Finally, the programmer is required to annotate the time loop by using the *timeloop* clause (as shown in lines 5 and 6).

The programming model of Daino bares differences to the programming models in other mature AMR frameworks. The most notable difference, touching on both the productivity and portability, is that in Daino we provide a single interface that is architecture-neutral. That is in comparison to committing the programmer to provide a target-optimized C function, or subroutine in Fortran, that computes the stencil in a single patch of a patch-based AMR (as in FLASH, Uintah, and GAMER). The later approach exposes the programmer to architecture and also requires different versions of the code to be submitted to use different targets.

It is important to note that for some applications, cells at the physical boundary of the domain are filled by the framework based on the boundary conditions. We currently support standard boundary conditions that the user can choose from, such as periodic/cyclic, wall, and Neumann. A basic set of flux correction algorithms is similarly supported. Similar to our intention for supporting user-defined error estimation functions, we intend on supporting user-specified boundary conditions and flux corrections by enabling precompiled user-defined functions to be hooked to the AMR runtime

Limitations and restrictions that the programmer must be aware of are summarized in the following points:

Supported stencils: Our framework specifically targets stencils for dense multidimensional Cartesian grids. For example, stencils for unstructured grids are not possible.

Data access: The framework supports limited data access patterns in the stencil. For example, stencil geometry and directions must be statically determined. This restriction implies that the framework cannot efficiently transform applications that use irregular data access patterns at the block level.

Pointer aliasing: Pointer aliasing is extremely complicated in static analysis, so we limit the programmer to restrict the pointers for data arrays that are marked by the programmer using the *data* clause directive.

Solution convergence: It is also important to note that Daino does not assure the solution convergence of the AMR code; it is the responsibility of the user to verify the accuracy of different refinement levels in the generated AMR code. As future work, we are considering to extend Daino to generate an ensemble of program instances having different settings to help the user in studying the convergence behavior.

C. Optimizations in Daino

When an AMR code generated by Daino is executed on a GPU-accelerated cluster, the stencil and mesh adaptation kernels run on the GPU, while managing the octree data structures and load balancing is done on the CPU side. Since we pursue efficiency and scalability, code on both the CPU and GPU should be optimized. In this section we briefly discuss the optimizations we apply in Daino.

1) Stencil Kernels: The stencil data arrays reside on the GPU main memory. Each block is calculated independently by a single CUDA thread block. Ghost layers for the blocks having neighbors on a different GPU, *exterior blocks*, are also kept on the GPU main memory. For a 3D mesh, we use 2D thread blocks at which each thread calculates the stencil for a vertical column. The values for each horizontal plane are loaded to the shared memory at each iteration to reduce the main memory traffic.

2) Mesh Adaptation: The mesh adaptation operations are natively executed on the GPU to avoid the CPU-GPU data transfer every time the mesh is adapted. Hence, the mesh adaptation operations in Daino are highly optimized GPU kernels. The *error estimation kernel* checks all the cells in each block against a given refinement criteria. Different schemes for error calculation exist. However, they are mostly data parallel memory-bound operations. Similar to the stencil kernel(s), each block is assigned to a thread block. The number of cells fulfilling the refinement criteria in each vertical column is stored in the shared memory. Next, a shared memory reduction operation is applied such that the number of cells fulfilling the refinement criteria per each block is stored in the GPU main memory. The CPU then copies back the result to further decide how to handle each block. Since only a single value per block needs to be copied back to the CPU, the overhead for copying the entire error array is considered negligible. After receiving the error array, the CPU divides the blocks into three sets: to be refined, to be coarsened, and those to be leaved intact. The CPU will then invoke the proper kernel for each set of blocks requiring similar action.

The *refinement kernel* applies an interpolation function on input blocks to produce child blocks. The input arrays are loaded to the shared memory in a manner similar to the stencil kernel, while the output arrays, i.e., child blocks, are written to the GPU main memory. For applications that allow recursive refinement, we experimented with *dynamic parallelism*: recursive CUDA kernel invocation. Yet achieving noticeable performance improvement appeared to be a difficult task in practice. The *coarsening kernel* is the opposite of the refinement kernel: a single block replaces eight sibling blocks. The coarsening kernel is a memory-bound kernel, and since each cell is used only once, all values are loaded and stored directly to the GPU main memory.

3) Load Balancing:

Intra-node: Load balancing is the main overhead in AMR. Therefore it is applied at much less frequency in comparison to remeshing. In systems with multi-GPU per node, we introduce a novel *intra-node* load balancing scheme to be applied at every remeshing step with zero overhead. Since the CPU is aware of the number of blocks per GPU, the CPU can balance the load among GPUs by moving blocks between GPUs after remeshing. The blocks are asynchronously moved between GPUs to avoid adding any overheads. We also use, *GPUDirect RDMA*, a technology introduced in Nvidia Kepler class GPUs to enable direct data exchange between GPUs, bypassing the CPU. No significant speedup gain was achieved in our experimentation when using intra-node load balancing, most likely due to the small number of GPUs per node in our test platform. However, with the trend of increased GPU count per node, we expect intra-node load balancing to be effective, specially that it comes at no overhead.

Inter-node: The SFC is first constructed then partitioned to decide on how to redistribute the blocks. Next, the nodes start P2P communication to migrate the excess blocks to their new

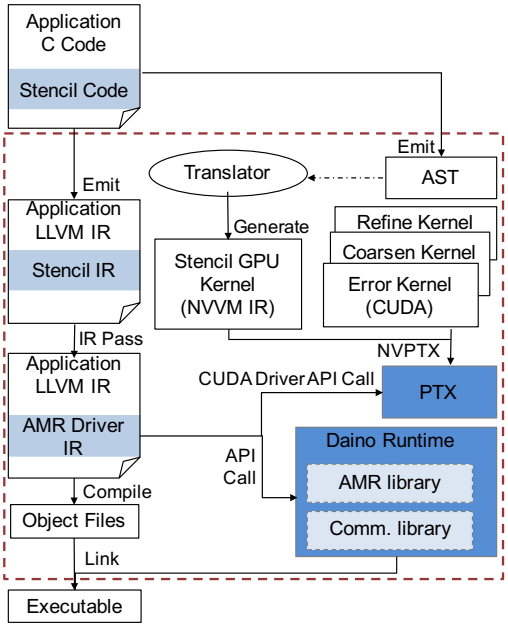

Fig. 2: The Daino framework overview. Application C code is transformed to an optimized executable. Daino components enclosed in red dotted line.

node(s). In Daino, excess blocks on the over-loaded nodes are asynchronously migrated while the CPU starts invoking the stencil kernels for blocks that are to remain on their respective GPUs. Accordingly, for the under-loaded nodes, the CPU manages multiple stencil kernel invocations to compute blocks that arrive from remote nodes.

4) Memory Management and CPU Orchestration: Frequent refinement can cause a significant overhead if memory would be allocated and deallocated on the GPU main memory. We implemented a memory manager that pre-allocates the entire memory space of the GPU main memory at the granularity of block size. The CPU tracks the use of the allocated memory and keeps pointer maps of the block arrays. The CPU passes the necessary memory pointers to each GPU kernel invocation depending on the needs of the kernel, e.g., memory space for newly created blocks.

V. Implementation

Our framework consists of a compiler and runtime components. We generate executables optimized for GPU execution by leveraging the LLVM compiler infrastructure. This section presents the implementation of the compiler and runtime components of our framework.

A. Compilation

Our compiler builds on the LLVM compiler infrastructure [15]. First, we use the front end to analyze and translate the stencil source code into GPU-optimized code in the form of LLVM Intermediate Representation (IR). Next, compiler passes are applied on the IR to add the AMR management code, which in turn make API calls to: a) the runtime API, and b) the GPU-optimized code generated by Nvidia back end code generator. Finally, LLVM IR is compiled and linked with the runtime libraries to generate an executable. The stages of compilation in Daino are depicted in Figure 2. The different stages of the compilation is managed by a shell script that the programmer invokes. We elaborate on the compilation stages in the following sections.

1) Front end: First, we use Clang front end to emit the Abstract Syntax Tree (AST) of the application C code. The AST is parsed by Daino's translator to identify the directives and analyze the stencil operations. It is important to note that parsing customized pragmas in AST is not a straightforward task. Direct interfacing with the *lexer* and *preprocessor* components of the front end is required, and core data structures have to be modified. In addition to the static analysis, the front end lowers the application C code to LLVM IR.

2) Passes: We apply a pass on the IR of the uniform mesh code to replace the stencil function with *AMR driver* IR, which is the code that advances the simulation and runs the different parts of the AMR management software. We have designed the AMR driver to be as minimal as possible, leaving the bulk of the work in AMR management to be done by an AMR library. Supporting other target architectures would require changing runtime components and using the appropriate back end. However, the AMR driver code at the IR level would remain unchanged.

In a separate step, a translator component of Daino analyzes the AST and creates an optimized GPU stencil kernel that would replace the original stencil kernel (the translator is enclosed in an oval shape in Figure 2). We create the GPU stencil kernel using NVVM IR, which is LLVM-compliant IR presented by Nvidia to extend the standard LLVM IR for target GPU. Similarly, NVVM IR translations would be done for the refine, coarsen, and error estimation functions if they are to be defined by the programmer. Otherwise, Daino uses pre-written CUDA device kernels for the three functions selected by the user.

3) Back end: LLVM back end performs standard optimizations not related to Daino and compiles the LLVM IR to target-specific object files. First, the application IR is lowered down to CPU object files. Next, we use Nvidia NVPTX CodeGen (code generator) to compile the GPU kernels of the stencil and AMR operators from NVVM IR to PTX byte code.

4) Linking: Daino compilation does not require special linker support beyond linker support required by C. However, object files must be linked with the Daino's runtime. Currently, the runtime includes a library for managing AMR data structures and operations, and another communication library that wraps the MPI runtime. Linking to the CUDA driver is also required since we use the CUDA driver API to load and JIT compile PTX to a native GPU device. Alternatively, it is possible to reduce startup time by precompiling the PTX kernels; the *ptxas* tool provided by the CUDA Toolkit can be used in offline compilation of PTX to machine code (SASS).

B. Runtime

We have built two libraries into the Daino runtime. First, the *AMR library* that encapsulates the AMR hierarchy management software. Second, the *communication library* that

wraps the MPI runtime library to simplify data movement operations for the AMR driver. We have designed the runtime in this manner so that AMR library can be replaced with other versions of the runtime that are not necessarily limited to octree-based AMR, facilitating experimentation with other AMR mesh representations in the future.

The runtime provides APIs to maintain an octree, including creating and traversing the tree, load balancing, and advancing the simulation. There are also APIs for creating and maintaining a memory manager to facilitate the orchestration of work at node level. We also allocate buffers for the boundary data of neighbor processes and manage the packing/unpacking for the data exchanged. We use *cudaMemcpy* to copy from GPU memory to host page-locked memory for unit stride access. For non-unit stride cases, we invoke a CUDA kernel that reads the non-unit stride and write the data to host memory.

Boundary data exchange between nodes is implemented using the runtime communication API. Each node is kept transparent about the nodes at which neighbor blocks reside. All nodes call the same function to request the exchange of neighbors. Collective data exchange during load balancing is also a transparent operation for the nodes. Note that the blocks reside on the memory of multi-GPU, and hence the CPU has to take that into consideration when packing blocks that reside on different GPUs but are to be sent to the same destination node. In the case that the blocks residing on GPU memory exceed the GPU memory capacity, we move the excess blocks to the CPU memory until memory slots are freed up on one of the GPUs due to coarsening, or when load balancing is due.

The user should provide the total number of MPI ranks when executing the application. The runtime automatically creates the same number of MPI processes as the number specified by the user. It is expected for the total number of MPI processes to be the number of nodes multiplied by the number of accelerators per node. One of the runtime MPI processes runs as the master process to maintain application semantics which can include non-stencil code executed sequentially by one process. The master process also performs computation like other nodes.

As described above, the runtime hides many of the details involved in the AMR and data movement. While our current implementation supports optimized GPU, we conducted early experimental work on supporting x86 targets. More specifically, in the case of OpenMP, the Clang front end lowers the pragma directives into necessary runtime calls and code alterations. We added a separate runtime library to Daino that wraps the LLVM OpenMP runtime library (*libiomp*), and adjusted the AMR driver IR calls to our runtime accordingly.

VI. Performance Analysis of Generated AMR

This section discuss a performance analysis that we use to evaluate the efficiency of the AMR codes generated by Daino. It is relatively easy to model the performance of the steady-state computation of uniform meshes. On the contrary, the speedup promised by AMR is subjected to the dynamic changes in the mesh structure in addition to the overhead of mesh management. The problem of performance analysis is further complicated with high-level frameworks like ours. Therefore, we develop an analytical and experimental performance model that allows the user of Daino to understand the efficiency of AMR, i.e., how close the achieved speedup is to the maximum achievable speedup for a specific application.

To that end, we introduce a performance analysis that projects the achievable AMR speedup of an executed applications. This speedup projection informs the programmer of the efficiency of the AMR code in analogy to how the Roofline model [21] informs the programmer of how a kernel performance fairs against the theoretical limits of a machine. We enable the automated performance analysis by instructing Daino to generate an executable instrumented for relevant performance and timing measurements.

It is important to note the following assumptions and limitations about the performance analysis. First, this analysis is a not designed to help the programmer get the highest speedup of an AMR implementation while maintaining accuracy. Second, the analysis in this section is dedicated for octree-based structured AMR and relies on empirical measurements. However, the analysis can be adapted to patch-based structured AMR models that use fixed size for patches. Third, both the measured and projected speedups use the uniform mesh as baseline for speedup. In this analysis, the uniform mesh is considered a worst case adaptive mesh at which all blocks are set to the maximum refinement level for the entirety of the simulation. Fourth, the projected speedups are valid on a single-run bases as different runs in adaptive meshes typically have different behaviors. Therefore, the programmer would need a well-sized test suite to draw conclusions about the efficiency of the transformation.

The theoretical AMR speedup of a given program can be calculated by simply dividing the total number of blocks in the adaptive mesh code, throughout the entire simulation, by the total number of blocks in the equivalent uniform mesh. The theoretical AMR speedup can be informative, yet is not ideal as no consideration is given to the overheads of managing an adaptive mesh. Hence we introduce a practical AMR speedup projection model that accounts for the overheads in octree AMR.

The following is a formal definition of the practical AMR speedup. For a number of time steps in a simulation D and frequency of remeshing F, there are S stages in the simulation where all iterations in a stage S_i, $i \in 1, D/F$, execute an unchanging mesh. Assuming N_i total blocks at each stage S_i, N_i is broken down to blocks at different refinement level, i.e, $N_i = \sum N_i^j$, where $j \in R$ and R is the set of refinement levels. The total runtime of the adaptive implementation to be as follows:

$$T_A = \sum_{k=1}^{D/F} \left(\sum_{j \in R} T_{Block} * N_i^j + T_{Ghost} * N_i^j * Aff + T_{Remesh} \right) + T_{LoadBalance} * Freq_{LoadBalance} \quad (1)$$

The following two terms in the equation above represent the two major overheads, and they are application-

specific: time to remesh T_{Remesh} and time for load balancing $T_{LoadBalance}$ (later shown in equations 2 and 3, respectively). $Freq_{LoadBalance}$ is the frequency of load balancing and T_{Ghost} is the time required to exchange the ghost layers with the neighbor blocks residing on a different memory. T_{Ghost} is in turn broken down to the following: a) time to the query the neighbors, b) time required to interpolate when the neighbors have different refinement levels (time is $O(N * \tau_i)$ where τ_i is the empirically measured time for a single block interpolation), and c) time spent in transferring the data of the ghost layers, and that would include the extra data exchange in the Z-direction (time is $O(N * \tau_g)$ where τ_g is the time for transfer the ghost layer for a single side of a 3D block). To account for the memory hierarchy in modern supercomputers, the affinity factor Aff is used to account for different throughputs that vary with different locations of the neighboring blocks, i.e., time for ghost exchange is variable according to where the neighbor blocks reside. For example, if the neighboring block resides on the same GPU memory, Aff is set to 1. Alternatively, if the neighboring block resides on the memory of another GPU on the same node, Aff is set to the value of 32, which is the measured ratio in memory throughput of local GPU versus remote GPU in TSUBAME2.5 (can vary depending on the system).

The time to remesh a block, T_{Remesh}, is further broken down to:

$$T_{Remesh} = T_{ErrFunc} + max(T_{Refine}, T_{Coarsen}) + T_{Transfer} \quad (2)$$

Where $T_{ErrFunc}$ is the time required for evaluating the error estimation function for a single block, T_{Refine} and $T_{Coarsen}$ are the times required for applying a refinement and coarsening on a single block, respectively. T_{Refine} is set to zero if the block is to coarsened, and vice-versa. Finally, $T_{Transfer}$ is the time required to transfer the block to the location where the error function is evaluated at, e.g., a block transferred from a GPU to the CPU for remeshing. Daino uses a data-centric approach to keep the blocks on the GPU, hence $T_{Transfer}$ would be equal to zero. It should be noted that both T_{Refine} and $T_{Coarsen}$ include the time to query and access the sibling blocks, which would differ depending on the distance to the sibling, e.g., siblings residing on the same global memory of a single GPU versus siblings residing on different GPUs. The three terms $T_{ErrFunc}$, T_{Refine}, and $T_{Coarsen}$ are proportional to the term T_{Block}, which is the measured time required to perform the actual computation (i.e., stencil operation) on a single block. This is based on empirical observations that the three GPU kernels *Error*, *Refine*, *Coarsen* are memory-bound data parallel operations baring high resemblance to the stencil kernel. Hence, we set $T_{ErrFunc}$, T_{Refine}, and $T_{Coarsen}$ be a factor of T_{Block} based on the FLOPS/Byte count of the three kernels to that of the stencil kernel.

The time to load balance, $T_{LoadBalance}$, is further broken down to:

$$T_{LoadBalance} = T_{QueryLoads} + T_{SortBlocks} + T_{Migrate} \quad (3)$$

Where $T_{QueryLoads}$ is the time required to gather information about the loads, $T_{SortBlocks}$ is the time required to sort the blocks by their Morton ID to assure high affinity, and $T_{Migrate}$ is the total time to migrate all the blocks. $T_{QueryLoads}$ is an AllGather communication of a few bytes, hence its value is dictated mainly by MPI runtime and is not a variable overhead dependent on the AMR operations. Hence we use the empirically measured $T_{QueryLoads}$. For $T_{SortBlocks}$, we use parallel Bitonic sort like other octree AMR solvers [19]. The sort algorithm is not part of the Daino framework; a faster sorting algorithm would improve the end performance for any transformation framework, yet it does not imply that the transformation by Daino is more efficient than other frameworks. Therefore, we set $T_{SortBlocks}$ to the empirically measured sorting time. The main time consuming part in load balancing $T_{Migrate}$ is $O(P * ExtBlocks_k * \tau_m)$ where P is the number of nodes, $ExtBlocks_k$ is the number of exterior blocks on node P_k ($P_k \in P$), and τ_m is the empirically measured time for migrating a single block.

Finally, the projection of the practical AMR speedup is:

$$S = T_U / T_A \quad (4)$$

Where time for the adaptive version T_A is defined in equation 1 and time for the uniform version T_U is:

$$T_U = \sum_{k=1}^{D/F} \left(T_{Block} * N_i^{max} + T_{Ghost} \right) \quad (5)$$

Where the uniform mesh is assumed to a have all blocks set to the maximum refinement level N^{max}. Note that equation 5 excludes all overheads that would normally be encountered with the adaptive mesh version (as in equation 1).

The performance analysis above can be useful in informing the programmer about the efficiency of the transformations. In the remainder of this section, we discuss briefly other potential ways to benefit from the performance analysis. One way would be auto-tuning the compile time and runtime parameters to enable more efficient transformations. When the complexities of applications and systems increase, auto-tuning methods become a necessity. Nogina et. al reported auto-tuning to be very effective in some cases of AMR [22]. The performance analysis can be extended to be used in auto-tuning in Daino. One particular parameter that can highly influence the performance is the block size. Part of the future work would be auto-tuning the block size by taking into consideration the accuracy of resolution, size of the ghost layers, and the GPU device occupancy.

The performance analysis can also be used in studying the scalability limitations of AMR solvers. We do not explore scalability limitations in this paper due to space limits. However, as a practical example, we briefly highlight a scalability issue we observed while experimenting with the framework. When measuring the overheads in our transformations, the time spent in migrating blocks while load balancing showed high variance from values projected by the practical speedup model. Upon investigating, it became clear that for each node, there can be a highly varying number of neighbor nodes to which excess nodes are migrated. For instance, in one of the

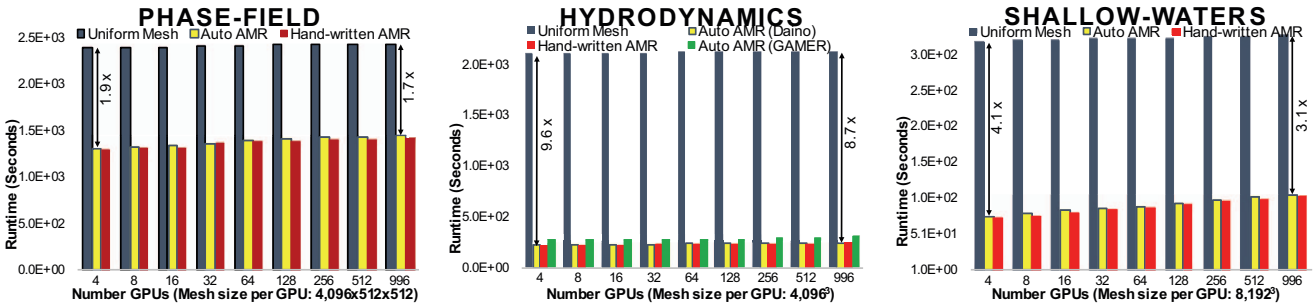

Fig. 3: Weak scaling of uniform mesh, hand-written and automated AMR (GAMER-generated AMR included in hydrodynamic)

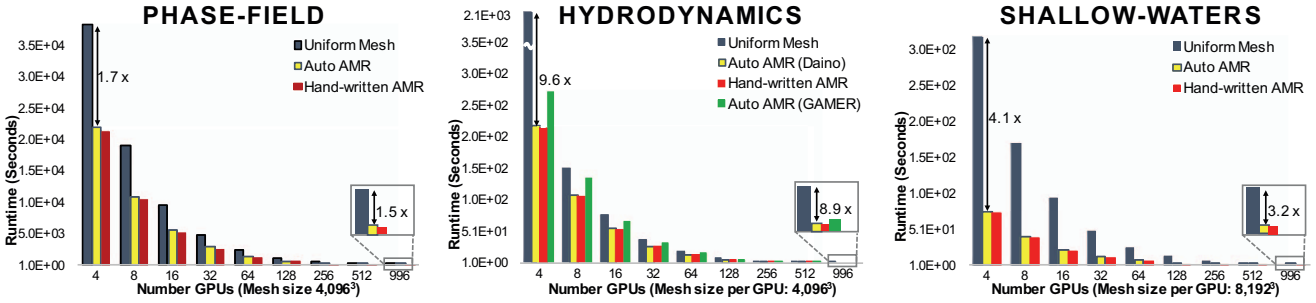

Fig. 4: Strong scaling of uniform mesh, hand-written and automated AMR (GAMER-generated AMR included in hydrodynamic)

applications, some nodes conducted P2P communication with up to 10 neighbor nodes. While SFC give good affinity, they do not optimize for reducing the number of neighbors at which neighbor blocks reside, which causes the mentioned effect. To address this issue, we are investigating a hybrid graph-based representation of the blocks along with the SFC.

VII. Evaluation

In this section, we first evaluate the speedup of three production applications and demonstrate the scalability of auto-generated AMR code. We compare the speedup and scalability with hand-written AMR of all three applications and also with an automated transformation realized by GAMER framework for one application. Next, we evaluate the efficiency of the transformations achieved via Daino by comparing against the AMR speedup projections. Finally, we highlight the effectiveness of our framework in improving programmer productivity by showing the lines of code of hand-written AMR for all three applications.

A. Applications

Phase-field simulation for dendritic growth: We simulate 3D dendritic growth during solidification in a binary alloy using a phase-field model. The computation requires a 2^{nd} order finite difference scheme for space with 1^{st} forward Euler-type finite difference method for time on a 3D uniform mesh. The phase-field model consists of a relatively large number of complex nonlinear terms compared to other stencil computations, such as advection and diffusion calculations. Hence, well-optimized phase-field implementations tend to be compute-bound. We use the code by Shimokawabe et. al as our reference implementation [23]. Note that this code uses a uniform mesh and is highly optimized (2011 Gordon Bell prize winner in the time-to-solution category).

Hydrodynamics Solver: We model a hydrodynamics application using 3D Euler equations. We explore a 2^{nd} order directionally split hyperbolic schemes to solve the equation. Namely, the Relaxing Total Variation Diminishing (RTVD) scheme [24] proposed by the GAMER implementation [25]. The RTVD scheme utilizes directional splitting to reduce solving 3D equations to a 1D problem by performing three sweeps, for each dimension. Sweeps are data-independent; each sweep calculates inter-cell flux values and updates half step solutions. We use the code by GAMER framework for the Hydrodynamics solver as our reference implementation [25].

Shallow-water Solver: We model shallow water simulations by depth-averaging the Navier–Stokes equations. We use a numerical method based on the semi-discrete, 2^{nd} order, central scheme by Kurganov and Petrova [26]. To evolve the solution in time, we use a 2^{nd} order total variation diminishing RungeKutta method. To advance the time step, we base our flux, max step, boundary condition, and time integration kernels on the model proposed by Saetra et. al [7].

B. Experimental Setup

We use the TSUBAME2.5 supercomputer at Tokyo Tech. Each node has two socket Intel Xeon X5670 2.93GHz CPUs (12 cores), and three Nvidia Kepler K20x GPUs with total 54GB and 18GB of system and GPU memory. The compute nodes are interconnected by dual QDR Infiniband networks with a full bisection-bandwidth fat-tree topology network. We

TABLE I: Phase-field runtime breakdown (%) for Daino AMR

GPUs	Stencil (GPU)	AMR				Total
		Ld. Blc.	Remesh	Ghost	2:1 Blc.	
32	88.7%	8.8%	<1%	1.8%	<1%	100%
128	87.2%	9.7%	<1%	2.7%	<1%	100%
512	84.3%	11.1%	<1%	3.2	1.2	100%

TABLE II: Hydrodyn. runtime breakdown (%) for Daino AMR

GPUs	Stencil (GPU)	AMR				Total
		Ld. Blc.	Remesh	Ghost	2:1 Blc.	
32	94.1%	4.5%	<1%	1.2%	<1%	100%
128	91.8%	5.5%	<1%	1.9%	<1%	100%
512	88.0%	8.3%	<1%	2.4	1.1	100%

use CUDA v7.0 Toolkit for GPU code and LLVM compiler infrastructure v3.8 for the framework. Single precision variables are adopted in all experiments for all applications. The hand-written and automated AMR versions of all applications use the data-centric AMR approach mentioned in Section III-A. All experiments used 16^3 mesh block size and 2D CUDA thread blocks of size 16x16 threads. All test runs are collected for $100,000$ time steps with a constant maximum of six refinement levels. All applications use a cell-centered mesh and a linear interpolation scheme. Phase-field simulation uses a phase-field threshold as a refinement criterion while the hydrodynamics solver uses pressure gradient and shock as refinement criteria. We initialize the adaptive mesh in phase-field by remeshing the base mesh uniformly across the initial domain. For the other two applications, the AMR spatial resolution of the initial condition is solely provided by the original code.

The results for the AMR versions of all applications were verified as follows: a) phase-field: we compared the numerical results to converged solution of the phase-field, concentration of solution, and temperature tensors, b) hydrodynamics: we compared the numerical results of the velocity, pressure, and density tensors to exact analytical solutions derived by Bertschinger [27], [5], and c) shallow-waters: we used exact analytical solutions computed by the SWASHES library [28] to compute a steady-state reference solution for a transcritical flow with a shock over a bathymetry.

We do not discuss the experimental conditions for the applications due to space limit[2]. However, we like to note that we adhere to the experimental conditions in [23], [25], and [7] for the phase-field, hydrodynamics and shallow-waters simulations, respectively.

C. Speedup and Scaling

Adaptive meshes aim to speedup computation by reducing the computational requirements. Hence, it is meaningless to use time-normalized performance metrics, such as FLOPs, to compare adaptive meshes to uniform meshes. In this paper, since we use a single system, we base our comparison of AMR code implementations to uniform mesh based on speedup. It is worth mentioning that the uniform mesh implementations for all three applications are highly optimized. The performance of the uniform mesh implementations in experiments with 128 GPUs are as follows: 101.1 TFlops for the phase-field using a 4096^3 mesh (20% of peak), 131.4 TFlops for hydrodynamics using a 4096^3 mesh (26% of peak), and 60.6 TFlops for shallow-waters using a 8192^3 mesh (12% of peak).

[2]We elaborate on the experimental conditions in Daino's documentation. Daino is available online at http://github.com/wahibium/Daino

TABLE III: Shallow. runtime breakdown (%) for Daino AMR

GPUs	Stencil (GPU)	AMR				Total
		Ld. Blc.	Remesh	Ghost	2:1 Blc.	
32	93.7%	4.7%	<1%	1.2%	<1%	100%
128	92.0%	6.1%	<1%	1.6%	<1%	100%
512	90.5%	7.3%	<1%	1.9	<1%	100%

In a weak scaling experiment, shown in Figure 3, the runtime for uniform mesh, hand-written AMR, and auto-generated AMR are compared. The following points are important to note. First, more than 1.7x speedup is achieved using Daino with 1000 GPUs for the phase-field simulation. This is a considerable improvement considering that the uniform mesh implementation is a Gordon Bell prize winner for time-to-solution. Second, we included a comparison with the auto-generated AMR by GAMER framework for the hydrodynamics solver. The AMR code generated by Daino is faster than the code generated by GAMER, mainly because GAMER uses a pipeline to hide data movement latency while Daino uses a data-centric approach to avoid data movement altogether. Third, Daino achieves good scaling that is comparable to the scalability of the hand-written AMR code.

Figure 4 shows a strong scaling comparison for hand-written AMR and auto-generated AMR against uniform mesh implementation. The auto-generated AMR by Daino achieves runtime and scalability comparable to that of the uniform mesh implementation. However, when using more GPUs, reduction in speedup starts to occur as the management of AMR starts to dominate the simulation runtime.

Tables I, II, and III list the breakdown of runtimes achieved using Daino for Phase-field, hydrodynamics, and shallow-waters, respectively. We use the mesh sizes in Figure 4 to achieve the results in the three tables. The following points are important to note. First, time integration, i.e., GPU stencil kernel(s), is the major component of the runtime. Second, the refine, coarsen, and error estimation algorithms (4^{th} column in table) have a negligible percentage of runtime since GPU is ideal for executing those data parallel kernels. Third, the AMR overhead for the shallow-waters application is lower than the other two applications since the stencil kernels in shallow-waters are less computational intensive.

We conducted early experiments on some kernels in the shallow-waters application to generate AMR code for Intel x86 target using OpenMP. The resulting AMR code is far from being optimized for OpenMP. Nonetheless, we highlight those early results. In one experiment, we used 16 TSUBAME nodes with the number of OpenMP threads equal to 12. The MPI+OpenMP hand-written code was optimized and achieved 1.17 TFlops of performance with a speedup of 3.42x over the MPI+OpenMP uniform mesh code. The AMR code generated by Daino achieved 3.11x over the uniform mesh code.

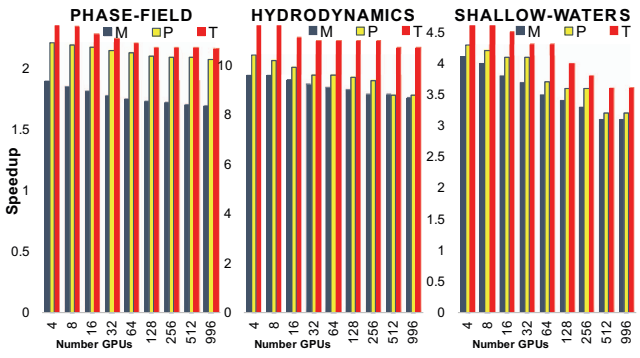

Fig. 5: Speedup: measured vs. projected. *M* is measured, *P* is the practical AMR speedup projection, and *T* is the theoretical AMR speedup projection. The mesh sizes for respective number of GPUs are defined in Figure 3.

D. Efficiency of Transformation in Daino

Figure 5 shows the AMR speedups projected by the performance analysis. The theoretical AMR speedup, bars with the letter *T*, is an upper-bound on the speedup, with no regard to AMR overheads. The measured speedup of AMR code auto-generated by Daino achieves an average of 78% of theoretical AMR speedup for phase-field (ranging between 78% and 80%), 83% for hydrodynamics (ranging between 80% and 84%), and 85% for shallow-waters (ranging between 81% and 89%). Comparing the measured speedup to the practical AMR speedup, bars with the letter *P*, further confirms the high efficiency of the AMR code generated by Daino. More importantly, the difference between the measured and projected speedups highlights the potential areas for improving the code transformation. For instance, the phase-field simulation achieves smaller percentage of the projected speedup, in comparison to the other applications. This is mainly due to observed inefficiency in block migration. Accordingly, the code transformation in this case should prioritize the optimization of block distribution over the nodes. In contrast, solvers having stencil kernels of very low computational intensity, such as shallow-waters, are mainly bounded by the memory bandwidth at the block level and less with the AMR management software. Hence, optimizing the transformation for such type of solvers should prioritize high level stencil optimizations, such as temporal blocking.

E. Productivity

The hand-written AMR version of phase-field, hydrodynamics, and shallow-waters required the addition/changing of 0.5K (18%), 0.9K (12%), and 0.8K (13%) of the code lines in the baseline implementation, respectively. This is a significant effort required by the programmer in comparison to the automated AMR version, i.e., a uniform mesh annotated with Daino directives. Because Daino's programming model does not require the programmer to modify the original codes or write target-optimized stencils, the number of lines for adding the directives is negligible in comparison to the lines of code for the hand-written AMR. Finally, the transformation time required by Daino averages at less than 4 minutes in our experiments with the three applications.

VIII. Conclusion

This paper introduces a framework that supports high-level programming of structured AMR applications. Our programming model is based on a set of architecture-neutral directives. The framework transforms annotated serial C code of uniform meshes to parallel AMR code optimized for GPUs. Daino's compiler and runtime are together the driver for enabling efficient AMR optimized for GPUs. Most notably, the compiler and runtime enable a data-centric approach at which all mesh adaptation operations are executed on the GPU to reduce CPU-GPU data traffic. This paper presented our current framework implementation and evaluation of its productivity and efficacy. We demonstrated how our framework successfully generates scalable and efficient AMR code for up to 1000 GPUs, for different production applications.

The main directions for future work are as follows. First, we need to experiment with more applications to fully understand and evaluate the limitations of the programming model. Second, we plan on leveraging the mesh adaptation techniques in Daino and LLVM modularity to extend our framework to support Xeon Phi. Finally, expanding the performance analysis and using it for scalability studies and auto-tuning is a viable direction that we seek to pursue.

Acknowledgments

This project is partially supported by JST, CREST through its research program: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing.". This research is also partly supported by KAKENHI, Grant-in-Aid for Scientific Research (S) 26220002 from the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan and "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (JHPCN)" and "High Performance Computing Infrastructure (HPCI)" in Japan.

The authors thank the Global Scientific Information and Computing Center, Tokyo Institute of Technology for use of the resources of the TSUBAME2.5 supercomputer.

References

[1] R. Lucas, J. Ang, B. K., S. Borkar *et al.*, "Top Ten Exascale Research Challenges," Office of Science, U.S. Department of Energy, Washington, D.C., http://science.energy.gov/ /media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf, Tech. Rep. DOE ASCAC Subcommittee Report, Feb. 2014.

[2] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide, "Extensible Component-based Architecture for FLASH, a Massively Parallel, Multiphysics Simulation Code," *Parallel Comput.*, vol. 35, no. 10-11, pp. 512–522, Oct. 2009.

[3] http://flash.uchicago.edu.

[4] H. Ji, F.-S. Lien, and F. Zhang, "A GPU-accelerated Adaptive Mesh Refinement for Immersed Boundary Methods," *Computers and Fluids*, vol. 118, pp. 131 – 147, 2015.

[5] H.-Y. Schive, Y.-C. Tsai, and T. Chiueh, "GAMER: a GPU-Accelerated Adaptive Mesh Refinement Code for Astrophysics," *Astrophys. J. Suppl.*, vol. 186, pp. 457–484, 2010.

[6] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox, "Extreme-Scale AMR," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, 2010, pp. 1–12.

[7] M. L. Sætra, A. R. Brodtkorb, and K.-A. Lie, "Efficient GPU-Implementation of Adaptive Mesh Refinement for the Shallow-Water Equations," *J. Sci. Comput.*, vol. 63, no. 1, pp. 23–48, 4 2015.

[8] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Lffler, B. OShea, E. Schnetter, B. V. Straalen, and K. Weide, "A Survey of High Level Frameworks in Block-structured Adaptive Mesh Refinement Packages," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3217 – 3227, 2014.

[9] C. Gheller, P. Wang, F. Vazza, and R. Teyssier, "Numerical Cosmology on the GPU with Enzo and Ramses," *Journal of Physics: Conference Series*, vol. 640, no. 1, pp. 12–58, 2015.

[10] B. Zink, "A general relativistic evolution code on cuda architectures," in *The 9th LCI International Conference on High-Performance Clustered Com- puting at the National Center for Supercomputing Applications in Urbana, IL, USA*, 2008, pp. 1–23.

[11] D. Beckingsale, W. Gaudin, A. Herdman, and S. Jarvis, "Resident Block-Structured Adaptive Mesh Refinement on Thousands of Graphics Processing Units," in *Parallel Processing (ICPP), 2015 44th International Conference on*, 2015, pp. 61–70.

[12] H. Shukla, H.-Y. Schive, T.-P. Woo, and T. Chiueh, "Multi-science Applications With Single Codebase -gamer- for Massively Parallel Architectures," ser. SC '11, 2011, pp. 1–11.

[13] S. G. Parker, "A Component-based Architecture for Parallel Multi-physics PDE Simulation," *Future Gener. Comput. Syst.*, vol. 22, no. 1-2, pp. 204–216, Jan. 2006.

[14] Q. Meng, A. Humphrey, and M. Berzins, "The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 2441–2448.

[15] http://www.llvm.org.

[16] M. J. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations," *Journal of Computational Physics*, vol. 53, no. 3, pp. 484 – 512, 1984.

[17] T. TU, D. R. O'HALLARON, and O. GHATTAS, "Scalable Parallel Octree Meshing for Terascale Applications," ser. SC '05, 2005, pp. 4–18.

[18] H. Tropf and H. Herzog, "Multidimensional Range Search in Dynamically Balanced Trees," *Angewandte Informatik*, 1981.

[19] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[20] M. Wahib and N. Maruyama, "Data-centric GPU-based Adaptive Mesh Refinement," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '15, 2015, pp. 3:1–3:7.

[21] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, pp. 65–76, 2009.

[22] S. Nogina, K. Unterweger, and T. Weinzierl, "Autotuning of Adaptive Mesh Refinement PDE Solvers on Shared Memory Architectures," in *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part I*, ser. PPAM'11, 2012, pp. 671–680.

[23] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka, "Peta-scale Phase-field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer," ser. SC '11, 2011, pp. 3:1–3:11.

[24] H. Trac and U.-L. Pen, "A primer on eulerian computational fluid dynamics for astrophysics," *Publications of the Astronomical Society of the Pacific*, vol. 115, p. 303, 2003.

[25] H.-Y. Schive, U.-H. Zhang, and T. Chiueh, "Directionally Unsplit Hydrodynamic Schemes with Hybrid MPI/OpenMP/GPU Parallelization in AMR," *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 4, pp. 367–377, Nov. 2012.

[26] A. Kurganov and G. Petrova, "A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System," *Commun. Math. Sci.*, vol. 5, no. 1, pp. 133–160, 03 2007.

[27] E. Bertschinger, "Self-similar Secondary Infall and Accretion in an Einstein-de Sitter Universe," *Astrophys. J. Suppl.*, vol. 58, p. 39, 1985.

[28] O. Delestre, C. Lucas, P.-A. Ksinant, F. Darboux, C. Laguerre, T.-N.-T. Vo, F. James, and S. Cordier, "SWASHES: a Compilation of Shallow Water Analytic Solutions for Hydraulic and Environmental Studies," *International Journal for Numerical Methods in Fluids*, vol. 74, no. 3, pp. 229–230, 2014.