SIAM J. SCI. COMPUT.
Vol. 37, No. 1, pp. C31–C53

PARALLEL COMPACT HASH ALGORITHMS FOR COMPUTATIONAL MESHES*

REBECKA TUMBLIN†, WILLOW AHRENS‡, SARA HARTSE§, AND ROBERT W. ROBEY¶

Abstract. We employ compact hashing and the discrete properties of computational meshes to optimize spatial operations in scientific computing applications. Our target is to develop highly parallel compact hashing methods suitable for the fine-grained parallelism of GPU and MIC architectures that will scale to the next generation of computing systems. As a model, we apply spatial hashing methods to the problem of determining neighbor elements in adaptive mesh refinement (AMR) schemes. By applying memory savings techniques, we extend the perfect spatial hash algorithm to a compact hash by compressing the resulting sparse data structures. Using compact hashing and specific memory optimizations, we increase the range of problems that can benefit from our ideal $O(n)$ algorithms. The spatial hash methods are tested and compared across a variety of architectures on both a randomly generated sample mesh and an existing cell-based AMR shallow-water hydrodynamics scheme. We demonstrate consistent speed-up and increased performance across every device tested and explore the ubiquitous application of spatial hashing in scientific computing.

Key words. hashing, compact hash, parallel computing, AMR, GPU, cell-based adoptive mesh refinement

AMS subject classifications. 68W10, 68Q85, 68Q25, 65K05

DOI. 10.1137/13093371X

1. Introduction.

The compact spatial hash algorithms we have developed extend the domain of hashing to include operations on finely resolved computational mesh structures. Robey, Nicholaeff, and Robey [23] demonstrated that mesh operations can exploit the properties of discretized data, allowing for a hash-based approach to be implemented for spatial operations such as neighbor finding, sorting, remapping, and table look-ups. Using their perfect spatial hash, they were able to achieve the ideal limit of $O(n)$ time. Computational meshes represent a subset of discretized data that we explore in this current work. With exascale computing on the horizon, memory operations and their consequent power usage are quickly becoming the dominant consideration for production codes in terms of speed-up and cost efficiency. We introduce memory efficiencies in the perfect spatial hash which allow hashing functions to be applied to a variety of physical applications by increasing the breadth of their

*Submitted to the journal's Software and High-Performance Computing section August 20, 2013; accepted for publication (in revised form) October 20, 2014; published electronically January 20, 2015. This work was partially supported by the Los Alamos National Laboratory Director's Office. Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396.

http://www.siam.org/journals/sisc/37-1/93371.html

†Corresponding author. XCP-2 Eulerian Applications Group, Los Alamos National Laboratory, Los Alamos, NM 87545, and Institute for Theoretical Science, University of Oregon, Eugene, OR 97403 (rtumblin@uoregon.edu).

‡XCP-2 Eulerian Applications Group, Los Alamos National Laboratory, Los Alamos, NM 87545, and Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94709 (willow@csail.mit.edu).

§XCP-2 Eulerian Applications Group, Los Alamos National Laboratory, Los Alamos, NM 87575, and Department of Computer Science, Brown University, Providence, RI 02912 (sara.hartse@gmail.com).

¶XCP-2 Eulerian Applications Group, Los Alamos National Laboratory, Los Alamos, NM 87545 (brobey@lanl.gov).

performance enhancements to include a wide range of spatial structures.

Advances in high-powered computing have led to the emergence of computational science as a new paradigm in science while acting as a crutch for both the theoretical and experimental fields. Most advanced production codes can take years to develop, and scientists often lag behind the most recent advances in hardware and algorithms due to the difficulty and time involved in integrating new techniques into pre-existing codes. As scientific computing becomes more central to the scientific community, portability is quickly becoming an important consideration. We are able to show that compact hashes are robust, efficient, and easy to implement in highly resolved, complex physical applications while at the same time showing significant speed-up and increased performance on a variety of architectures.

As industry continues to proliferate parallel processors, adapting algorithms for fine-grained parallelism becomes necessary to fully utilize the capability of these machines. Hashing is an intrinsically parallel, noncomparison-based search algorithm which requires no interthread communication. This allows performance portability across CPU and GPU architectures. Spatial hashing functions which take advantage of discretized data properties provide a new and innovative approach for adapting algorithms to more highly parallel architectures with broad application to computational science.

This paper is organized as follows. In section 2 (background), we provide a synopsis of hashing and the advances that led to our research. We also briefly explore the foundations of cell-based adaptive mesh refinement (AMR) and current practices. In section 3 (methodology), we describe the memory optimizations that enable the spatial compact hash. In section 4 (performance results), we explore performance on a randomly generated sample mesh and develop a cost model. In section 5 (application to scientific computing), we demonstrate the application of compact spatial hashing to a cell-based AMR shallow-water hydrodynamics scheme, and we explore the performance enhancements achieved. In section 6 (conclusions), we review the impacts these methods have in the broader scope of computation and where there is potential for further development.

2. Background. A hash table consists of an array of memory spaces, called buckets, where data may be stored. This data takes the form of values which are accessed by their associated keys. These keys are used to map their values to locations within a hash table. Every key maps to a unique integer, called a hash code, by means of a hash function. This unique integer is then converted to an index corresponding to a bucket in the table. We call this index a hash location. An entry (key-value pair or just the value for a perfect hash) is inserted into a hash table, and then the table is later queried for a key to get its associated value if it exists. More formally, a perfect (or 1-probe) hash function, h, is an injection (one-to-one) mapping such that

$$h : U \to \{B_k\}\,. \tag{2.1}$$

The set of keys $\{S\}$ are members of the universe of possible values, $S_i \in U$, and the buckets $\{B_k\}$ comprise the hash table with a size $|\{B_k\}|$. In addition, for $x, y \in \{S\}$ such that $x \neq y$, it must be true that $h(x) \neq h(y)$. This last expression guarantees that no collisions may occur.

A perfect hash table is designed for data whose range and pattern of hash codes are known, so that creating a hash table large enough to accommodate all the possible keys is feasible. A compact hash uses a compression function to reduce the range of hash locations. If it cannot be guaranteed that the hash codes will be unique, we

must make plans to overcome the situation in which two different keys have made their way to the same bucket. This situation is referred to as a collision, and compact hashes are designed to handle collisions.

One of the earliest collision handling methods, chaining, developed by Williams [24], allows multiple keys to exist in one bucket through the use of singly linked lists (chains of pointers, leading to this method's name). Another early approach, called double hashing, uses a separate hash table to store colliding entries. An interesting approach to using the existing memory for the hash table was developed by Peterson [21] and is called open addressing. Peterson proposed, in the event of a colliding entry, to search to the right of its original location in a predetermined, repeatable sequence for an empty bucket in which to insert the entry. Regardless of how collisions are resolved, they are always more expensive to handle than simply inserting an entry into an empty bucket. The minimization of collisions is a key aspect of compression function design. Thus, the compression function in a compact hash table is designed to randomize the hash locations to reduce the number of collisions that occur due to patterns in the data. The table size then has no relation to the range of possible keys, but rather to the number of entries to be inserted. The number of entries divided by the number of buckets in a hash table is referred to as the load factor.

Advances have been made to both perfect hashing and compact hashing since their creation. Czech, Havas, and Majewski [7] present a thorough treatment of enhancements to perfect hashing in their monograph. Most of the monograph deals with static datasets where the implementation of a perfect hash function requires a lot of computing time. However, the last section of the monograph discusses dynamic datasets and also has a very short section on parallelization efforts for these methods. For the compact hashing domain, Munro and Celis [19] present some advanced reordering techniques such as Brent's method. Brent [5] proposed the smart placement of keys with the plan to reduce the number of probes needed for queries. These reordering techniques are based on the idea that when there is a collision, there is a choice of which key-value pair to move down the probe sequence.

The recent interest in parallel programming has led to the realization that hash tables are easy to parallelize. The nature of the perfect hash function requires virtually no interthread communication, allowing for fine-grained parallelism to be exploited. Each insertion or query can function alongside one another since each bucket holds its own unique key. This is unlike a k-D tree method, where searching operations are dependent on the previous iteration. A k-D tree bisects the data and searches to the left and right and then continues until the queried data is found. The running time of this algorithm is in $O(n \log n)$ time for all n elements of the mesh. The time for a single query is on average $O(\log n)$. In hash-based approaches, we directly query the memory location of the desired value. Hash-based approaches execute in expected $O(n)$ time for all n elements of the mesh. A single query will on average complete in $O(1)$ time.

Implementations of compact hashing with its collisions also introduce some dependencies on adjacent threads, though to a lesser degree than a tree-based method. In highly parallel environments, the collision handling can generate performance and correctness issues. These can be resolved with a locking mechanism, but more recent work by Gao, Groote, and Hesselink [10] developed a lock-free and nearly wait-free implementation of open-address hashing. An interesting aspect of their work is the lengthy proof of correctness they felt necessary to undertake for their hash implementation.

Alcantara (see [2, 1]) saw that advancing technology allowed for efficient imple-

mentation of hashing methods on GPUs. He developed a set of parallel implementations for open addressing, chaining, and cuckoo hashing on GPUs. He then examined their performance based on memory usage, table construction speed, and retrieval efficiency. Alcantara showed that tables with millions of elements can be effectively implemented by using the new atomic operations on the GPUs. He also noted that each application requires different considerations for the inclusion of specialized features when developing the hashing method. Ultimately, a balance between memory usage, table construction speed, and retrieval efficiency must be tailored to each specific application.

2.1. Background on cell-based adaptive mesh refinement (AMR). Many physical applications in scientific computing rely on acquiring numerical solutions to partial differential equations (PDEs) on a discrete grid structure that approximates a continuous space. The size of each grid element is directly proportional to the numerical error associated with algebraically approximating solutions to PDEs. Often, physical applications estimate solutions across steep shocks or gradients where the physics changes rapidly in time. Resolving discontinuities on a discrete mesh requires finely spaced grid elements, and, usually, the discontinuities that arise cover only a sparse region of the grid. It is therefore inefficient to finely resolve the entire mesh. AMR allows finer resolution over regions of interest while keeping the computational costs down over regions where low resolution is suitable.

The most common type of AMR in published literature is block- or patch-based methods which use small refined patches that have a regular grid structure. These methods are based on the work of Berger and Oliger [4]. In our current work, we use an approach called cell-based AMR. This approach, originally specified in Young et al. [26], uses a quad-tree in two dimensions and an oct-tree structure in three dimensions with each cell having a pointer to its parent and four or eight pointers to children, respectively. Even in this first implementation, coding tricks at the parent oct-tree level were used to shave the memory requirements down to as little as two additional words of memory per cell. The advantage of this approach is that the programmer has much finer control over the cells that are refined, which results in fewer cells being needed to model many problems. The disadvantage is that the determination of and access to neighboring cells is more difficult, and the memory overhead of the additional pointers is substantial. In addition, the parent tree structure is normally included in the cells, adding a memory overhead of approximately 14% in three dimensions, 33% in two dimensions, and 100% in one dimension.

Another implementation shortly thereafter by De Zeeuw and Powell [9] explored the trade-off between memory usage and computing time. The grid level of each cell and the cell neighbors can be calculated by tree-traversals when they are needed to avoid storage costs. Storing neighbors requires eight integers for a two-dimensional mesh (two possible neighbors for each face), while recalculating neighbors requires about 25% of the run-time. However, storing the mesh level for each cell requires only one integer and reduces the run-time by 15%. Based on these trade-offs for the measured performance and memory size of their computer, they chose to store the mesh level and recalculate the cell neighbors.

Several years later, the fully threaded tree (FTT) algorithm by Khokhlov [12] favored keeping all the neighbor pointers in cell arrays. A straightforward implementation would have required 17 additional words per cell, but this was mitigated by keeping the connectivity data at the parent of the oct-tree only, thereby reducing the additional memory requirements to 2 words per cell. The FTT method also developed better modifications of the tree structure that could be done in parallel.

More recent work by Ji, Lien, and Yee [11] proposes the use of a hash table at the parent of an oct-tree rather than a tree-traversal for performance and easier parallelization. Their work was targeted toward traditional CPUs and used the hash routines from the open-source GLib library. Our work has explored eliminating the additional storage of the parent tree structure for an on-the-fly generation of the connectivity using a k-D tree. The only data storage requirements would be the grid level of each cell and the Cartesian index location on that grid level. This results in 3 short words of storage per cell. A more recent concept replaces the k-D tree with a hashing approach that is much faster, easier to parallelize, and, most importantly, easier to implement on fine-grained computing engines such as the GPU and MIC.

3. Methodology. We now apply hashing techniques to operations involving data associated with computational meshes and the spatial layout of a grid. The hash-based spatial method treats a mesh's cells as discretized data and uses a hash function to map this data to a hash table. For our definition of differential discretized data, we mean a collection of information that is mapped onto a metric space $(A,\, d_A)$ with the properties defined by Robey, Nicholaeff, and Robey [23]. These properties are a connected, tessellated, discretized dataset that is bounded by a minimum cell size. For our current work, we will require two additional properties. The first is that cell sizes are of the same relative order of magnitude; e.g., for our AMR mesh refinement, if A_i and A_j are neighbors, then they cannot differ in size by more than one symmetric bisection. The second is that cell sizes are determined to minimize the discrete gradient representation, or computable gradient, relative to the contiguous gradient.

The connected property assures that each cell intersects at least one other cell:

$$\forall i[\exists(j \neq i)] \ni A_i \cap A_j \neq \emptyset. \tag{3.1}$$

The tessellated property ensures that there is no overlap of cells or that the intersection of a cell with another cell is only at the boundary:

$$\forall i, \forall (j \neq i)\, [A_i \cap A_j = \partial A_i \cap \partial A_j]\,. \tag{3.2}$$

These are recognized as the typical properties of an adaptive computational mesh in space or time.

Spatial hashing is characterized as mapping a metric space $(A,\, d_A)$ containing differential discretized data to the powerset of a finite cover $\{B_k\}$ of a new metric space $(B,\, d_B)$ whose elements form a hash table. Note that the use of a metric space endows a distance attribute to the set, distinguishing this concept as spatial hashing, in contrast with the general hashing concepts introduced earlier. We define this mapping as a hash function, h_s:

$$h_s : A \to \mathcal{P}\left(\{B_k\}\right) \quad (A_i \not\to \emptyset\ \forall i)\,, \tag{3.3}$$

where $\{B_k\}$ represents a bucket in the hash table. The hash table size, or finite cover, can then be defined as

$$|\{B_k\}| = \frac{m(A)}{|\Delta_{\min}|}, \tag{3.4}$$

where $|\{B_k\}|$ is the set of buckets, $m(A)$ is the range of the original data being hashed, and $|\Delta_{\min}|$ is some minimal integer size of the dataset. Mapping to a powerset allows the possibility of mapping elements of $\{A\}$ to multiple locations in the hash table. If we know the range our data encompasses, i.e., the maximum and minimum values

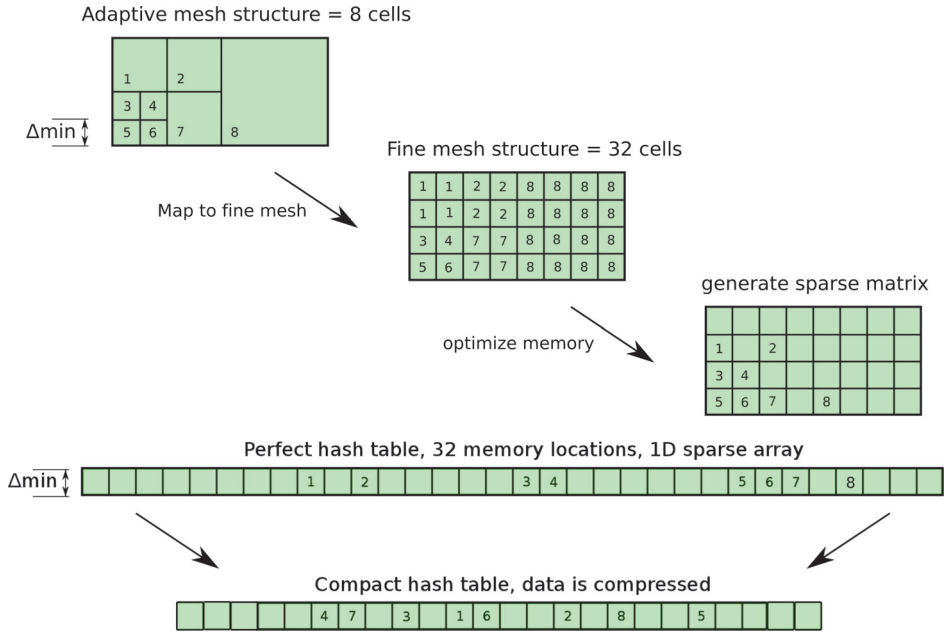


FIG. 1. *This concept diagram depicts the steps of hashing spatial data in AMR.*

it can take, $m(A)$, and we know the minimum difference between values, $\triangle_{min}$, this allows us to generate a perfect hash table containing the minimum number of buckets for an injective mapping. By choosing $\triangle_{min}$ to be the minimal distance between cell centers on a computational mesh, we can guarantee a bucket for the data contained in the computational mesh. Extending this formalism to compact hashing involves compressing the range of the new metric space $(B,\, d_B)$ so that there is no longer a one-to-one correspondence between the original elements of the metric space and the locations in the hash table. We explore specific methods for achieving this while at the same time preserving the relevant data in the original set in section 3.2.

To help elucidate the multiple layers of abstraction, we wish to provide a short conceptual run-through of spatial hashing in the context of AMR datasets. However, we do not want to limit the scope of spatial hashing to this one specific application. We will be using cell locations in our concept diagram, but physical data such as density fields could be used instead. Our approach involves mapping the AMR cell locations to the fine mesh structure as shown in Figure 1. By exploiting the mesh's inherent properties, we can reduce the number of writes which uniquely determine cell locations; these methods will be explored in section 3.2. Reducing the writes generates a sparse matrix which contains the spatial layout of the computational mesh.

Once we have reduced the number of writes, we map the mesh structure to a sparse, one-dimensional perfect hash table. The sparsity of the mesh makes it a viable candidate for compression. We can compress the data into a compact hash table using a compression function which randomizes the data to reduce the number of collisions. Later in the process, we will have to deal with the collisions. The terminology for perfect and compact spatial hashing is shown in Figure 2.

We rely on the spatial layout of the computational mesh to generate a unique key associated with each value. The idea of a unique key is explored in depth in an earlier paper by Robey, Nicholaeff, and Robey [23]. This key for a cell-based AMR grid is the

Terminology:

	compressibility	sparsity	load factor
perfect hash	32/8 = 4	8/32 = .25	8/32 = .25
compact hash	24/8 = 3	8/24 = .33	24/32 = .75

FIG. 2. *This table defines compressibility, sparsity, and load factor based on the examples in Figure* 1.

spatial location of the cell at the finest level of the mesh. This directly corresponds to the hash bucket index for the perfect hash. The bucket index is simply the unique key generated from the $(i, j, level)$ location and becomes just a simple array assignment or retrieval. In our current extension to compact hashes, we continue to exploit the unique key. However, rather than marking all of the cells at the finest level that are covered by the current cell as done previously, we will only write to the cell located at the lower-left corner to enable the hash compression. This is done through memory write optimizations that will be discussed in section 3.2.

3.1. The neighbor finding problem. In this paper, we only address the neighbor finding operation for a cell-based AMR application. This spatial operation is important to the performance and scalability of AMR codes. Optimizing each spatial mesh operation can only be done by exploiting the details of the application. There are two unique steps to this optimization. First, we reduce the number of writes to the minimum needed to support the spatial queries (see section 3.2). Second, we compress the data in the fine, regular grid structure of the mesh into a smaller compact hash (see section 3.3). We will exploit some of the application rules in this process. The most important rule is that refinement jumps can only be a factor of two at each interface. This means that each adjacent cell can be a factor of two finer, the same size, or a factor of two larger. The second rule for this use case requires that the hash table be temporary; it is constructed and then used only once per iteration. If it is fast enough to be used in this scenario, then memory can be freed for other needs in the computation. The programmer still has the flexibility to keep the hash table and make modifications to the table every iteration, saving much of the table initialization costs and set-up time.

Other operations such as remapping and table look-up can also be optimized in a similar manner, but the details of the spatial operation and the rules of the application must be considered in the process. This same methodology can be applied to other applications such as cell-based AMR with different refinement rules, patch-based AMR, and unstructured general polyhedral grids.

3.2. Memory write optimizations. The nature of the neighbor finding problem and the structure of the mesh determine whether the insertions and queries performed in the perfect spatial hash are actually necessary. We can exploit aspects of the neighbor finding problem to reduce the number of insertions necessary in the perfect hash table, as illustrated in Figure 3. Optimization 1 writes only to the outermost cells because the interior cells are never accessed and, thus, are irrelevant for determining a neighbor cell. Interior elements are given a null value, represented by a sentinel value. Optimization 2 takes advantage of the fact that the level of refinement

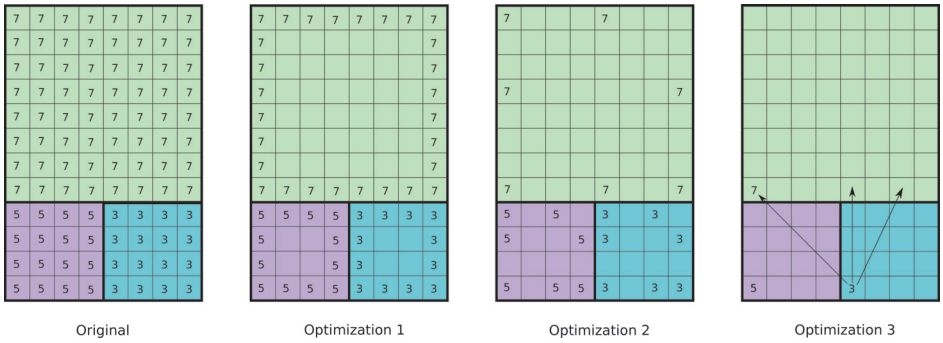


FIG. 3. *Memory write optimizations introduce sparsity in the hash table.*

LISTING 1
Write optimization 2 (seven writes and one read).

```
// Write (Insert) cells
for(int icell=0; icell<ncells; icell++){
    int levmult = (int)pow(2,levmx-level[icell]);
    int ii = i[icell]*levmult; int jj = j[icell]*levmult;

    hash[jj][ii] = icell; // lower left corner
    ii += levmult/2;
    hash[jj][ii] = icell; // lower boundary mid-point
    if(levmult > 2) {
        ii += levmult/2 - 1;
        hash[jj][ii] = icell; // lower right corner
        ii -= levmult/2 - 1;
    }
    ii -= levmult/2; jj += levmult/2;
    hash[jj][ii] = icell; // left boundary mid-point
    ii += levmult - 1;
    hash[jj][ii] = icell; // right boundary mid-point
    if(levmult > 2) {
        ii -= levmult - 1; jj += levmult/2 - 1;
        hash[jj][ii] = icell; // upper left boundary
        ii += levmult/2;
        hash[jj][ii] = icell; // upper boundary mid-point
    }
}
```

between cells can only differ by one symmetric bisection and reduces the writes to only seven outer cells. The code for this optimization is shown in Listing 1. Optimization 3 extends this idea, reducing the process to only one write with three reads, each checking for a finer, same size, or coarser cell, as shown in Listing 2. All of our optimizations involve only mapping some elements of the original dataset, (A, d_A), to the hash table, (B, d_B), while preserving the information needed to determine neighbor cells. This allows us to compress the number of buckets in the hash table without affecting the relevant data containing the cell locations on the adapted mesh.

Reducing the insertions and queries speeds up the execution time of the perfect hash and, as it reduces the number of necessary buckets, introduces sparsity into the problem. Sparsity is related to the memory compressibility of the mesh, which we define as the ratio of the maximum number of cells on the finest level of the mesh

LISTING 2
Write optimization 3 (one write and three reads).

```
//Read (Query) Neighbors
for (int ic=0; ic<ncells; ic++){
    int ii = i[ic]; int jj = j[ic]; int lev = level[ic];
    int levmult = (int)pow(2,(levmx-lev);
    int iicur = ii*levmult;
    int iilft = MAX( (ii-1)*levmult, 0          );
    int iirht = MIN( (ii+1)*levmult, imaxsize-1);
    int jjcur = jj*levmult;
    int jjbot = MAX( (jj-1)*levmult, 0          );
    int jjtop = MIN( (jj+1)*levmult, jmaxsize-1);

    int nlftval = -1, nrhtval = -1, nbotval = -1, ntopval = -1;

    // Need to check for finer neighbor first
    // Right and top neighbor don't change for finer, so drop through
    // Left and bottom need to be half of same size index for finer test
    if (lev != levmx) { // Can it be finer?
        nlftval = hash[jjcur][iicur-(iicur-iilft)/2];
        nbotval = hash[jjcur-(jjcur-jjbot)/2][iicur];
    }

    // same size neighbor
    nrhtval = hash[jjcur][iirht];
    ntopval = hash[jjtop][iicur];
    if (nlftval < 0) nlftval = hash[jjcur][iilft];
    if (nbotval < 0) nbotval = hash[jjbot][iicur];

    // coarser neighbor
    if (lev == 0) continue; // Can't be coarser
    if (nlftval < 0) nlftval = hash[(jj/2)*2*levmult][iilft-(iicur-iilft)];
    if (nrhtval < 0) nrhtval = hash[(jj/2)*2*levmult][iirht];
    if (nbotval < 0) nbotval = hash[jjbot-(jjcur-jjbot)][(ii/2)*2*levmult];
    if (ntopval < 0) ntopval = hash[jjtop][(ii/2)*2*levmult];
}
```

to the number of cells in the current timestep of the simulation (see Figure 2). Also, reducing the number of insertions helps with thread divergence issues on the GPU by decreasing the difference between the number of insertions and queries between adjacent threads. Thread divergence occurs when threads in a branching operation wait for the slowest thread to finish its calculation before advancing to the next instruction.

Now, with the reduction in insertions, the compact hash is viable because it is able to take advantage of this memory compression and make smaller hash tables, offsetting the much more costly insertions and allowing for the possibility of much larger problems.

The specifications of the grid we are examining, as well as the requirements of the neighbor finding problem, allow for these particular memory optimizations and the introduction of sparsity. In order to implement the compact hash for other spatial operations, it will be necessary to find comparable optimizations.

3.3. The compact hash. When making design decisions for the compact hash on the CPU and GPU, priority was given toward optimizing the GPU implementation, while making the two implementations as similar as possible to avoid code repetition. One example would be our choice of open addressing. Open addressing works incredibly well on the GPUs, but for the CPU, we could have used chaining in place of open addressing. Thus, some decisions made when optimizing the GPU governed the CPU algorithms.

In a perfect hash table, we don't have to handle collisions because each key's hash code is its hash location. When implementing a compact hash table, not only must we handle collisions, we must reduce their frequency of occurrence. By randomizing the hash codes before reducing their range to fit in the hash table, patterns in the input data that would normally lead to collisions can be avoided. Mitzenmacher and Vadhan [17] showed that a 2-universal compression function is sufficient to be competitive with more complex methods even though it is simple to implement and cost-effective to execute. The definition of what it means to be 2-universal is given by Carter and Wegman [6], and we advise the reader to refer to this work for a more complete understanding of this family of functions. Our compression function is taken from the first example of such a function given by Carter and Wegman:

$$hashLocation = ((a * hashCode + c)\%m)\%tableSize, \tag{3.5}$$

where a, c, and m are constants and the modulo (%) operator is used to generate randomness and keep the result in the range of the table. The constants a and c must satisfy certain constraints to achieve good performance. These are the same constraints necessary to guarantee a full period of the similar-looking linear congruential pseudorandom number generator [13, 18]. The factors c and m should be relatively prime, $a - 1$ should be divisible by all prime factors of m, and $a - 1$ should be a multiple of 4 if m is a multiple of 4. Our initial implementation uses simple random numbers, but for the library under development, we used accepted constants for linear congruential generators ($a = 2147483629, c = 2147483587$, m is the largest 32-bit prime). We plan on randomly generating numbers that satisfy the above constraints.

Although they can be reduced, collisions cannot be avoided entirely. To handle collisions, a quadratic probing open addressing hash table with the linear congruential compression function was chosen due to the superior construction and retrieval speeds for large data sizes on the GPU as shown by Alcantara [1]. When inserting or querying within a hash table with open addressing, the goal is the same: to find a bucket containing the given key or to find an empty bucket where the key can be placed. If we are inserting into the table, finding an identical key allows us to overwrite the entry there. Finding an empty location would allow us to simply insert the entry there. When querying the table for a key, finding the key means that we can return its associated value. Finding an empty bucket means that the key does not exist. The search for such a bucket is the same whether we are inserting into or querying the table. We obtain a hash location from the compression function and hash code (we get the hash code from the hash function and key). We first check the bucket at this location to see if it is null or contains a matching key. If not, we check buckets at locations in the sequence according to

$$hashLocation_i = (hashLocation_0 + P(i)), \tag{3.6}$$

where P is some predetermined probe sequence.

In a linear probing open addressing hash table, the probe sequence is

$$P_{linear}(i) = B * i. \tag{3.7}$$

It has been shown that 1 is a good value for B [13]. Linear probing does have some issues, however, as long clusters of keys are created that must be traversed iteratively to find empty locations or keys. This effect is called primary clustering and can be

LISTING 3
Pseudocode for compact hash using quadratic probing.

```
//* insert operation
set hashLocation to ((inputKey * a + c) % m) % tableSize
do{
    set insertLocation to next location in quadratic probe sequence
}while insertLocation isn't a valid spot for insertion and we haven't tried
     too many times
if insertLocation is a valid spot for insertion{
    set the key at insertLocation to inputKey
    set the value at insertLocation to inputValue
}

//* query operation
set hashLocation to ((inputKey * a + c) % m) % tableSize
do{
    set queryLocation to next location in quadratic probe sequence
}while the bucket at queryLocation isn't empty and doesn't match our
     inputKey and we haven't tried too many times
if the bucket at queryLocation is empty{
    let user know query is unsuccessful
}else if the keys match{
    return the value at queryLocation
}
```

remedied through the use of a quadratic probe sequence as developed by Maurer [16]. A quadratic probe sequence is

$$P_{quadratic}(i) = A * i^2 + B * i. \tag{3.8}$$

It has been shown that good values for A and B are 1 and 0 [13]. Quadratic probing solves the problem of primary clustering but suffers from another effect, secondary clustering. This is the issue of chains forming when multiple keys are mapped to the same initial spot and must probe the same sequence to find an empty spot to insert. It is known that the first $tableSize/2$ locations probed in a quadratic probe sequence will be unique if the table size is prime [3]. We need a prime number, and since there are simple tests for the primality of Proth numbers, we generate a Proth prime close to the desired table size. Proth primes are numbers of the form $N = k \cdot 2^n + 1$ that satisfy $a^{(N-1)/2} \equiv -1 (\mathrm{mod}\, N)$ [22]. The Proth primes occur with sufficient frequency for our fast computation needs. For a compact hash using open addressing with a quadratic jump, the operations can be expressed in a single yet complex loop as shown in the pseudocode in Listing 3 and C code for the CPU in Listing 4.

Hashing in parallel is very similar to hashing serially. Every thread is assigned an entry to insert, or a key to query, and all threads probe the same hash table in global memory in parallel for the appropriate locations. This model is particularly suited to GPU computing with its high degree of concurrency. The limiting factor for GPUs is that memory access time to global memory is 400–600ns versus 20–100ns for a CPU. However, GPUs are effective at hiding memory latency through context switching between workgroups, which the GPU does almost instantaneously if there is another resident workgroup. The parallel insertion algorithm differs from the serial one in one important way. When checking to see if a bucket is empty and, if it is, inserting a key in it, we must impose mutual exclusion on the bucket's key field and its empty or full field. This means that for the duration of the above set of operations, only one thread may access these fields at a time. If a sentinel key value is chosen to mean that a bucket is empty, this mutual exclusion can be performed with an atomic check and exchange operation. For an operation or set of operations to be considered

LISTING 4
Compact hash coding for the CPU using quadratic probing.

```
//* insert operation
iteration = 0;
hashLocation = ((inputKey * a + c) % m) % tableSize;
for (insertLocation = hashLocation;
    hash[2*insertLocation] != -1 && hash[2*insertLocation] != inputKey &&
        iteration < maxTries;
    insertLocation = (hashLocation + iteration * iteration) % tableSize)
        { iteration++;}
if(iteration < maxTries){
    hash[2*insertLocation] = inputKey;
    hash[2*insertLocation+1] = inputValue;
}

//* query operation
iteration = 0;
hashLocation = ((inputKey * a + c) % m) % tableSize;
for (queryLocation = hashLocation;
    hash[2*queryLocation] != -1 && hash[2*queryLocation] != inputKey &&
        iteration < maxTries;
    queryLocation = (queryLocation + iteration * iteration) % tableSize)
        { iteration++;}
if (hash[2*queryLocation] == inputKey){
    return &hash[2*queryLocation+1];
}else{
    return NULL;
}
```

LISTING 5
Psuedocode for the parallel insert operation.

```
set hashLocation to ((inputKey * a + c) % m) % tableSize
while we haven't tried too many times{
    set insertLocation to next location in quadratic probe sequence
    acquire lock on the key memory at insertLocation
    if insertLocation is a valid spot for insertion{
        set the key at insertLocation to inputKey
        unlock the key memory at insertLocation
        set the value at insertLocation to inputValue
        break
    }
    unlock the key memory at insertLocation
}
```

atomic, it must be performed without interruption. While not all atomic operations are implemented using a form of mutual exclusion, they must guarantee the behavior. If the insertion thread did not use one of these techniques during its execution, it would be possible for two threads to check whether a bucket was empty, and both write their (potentially different) keys into the same memory address. This creates a race condition, where the key that gets written depends on which thread reaches the memory address first. Obviously, this must be avoided because the other thread will, in this situation, fail to write its entry into the table at all. The pseudocode for a parallel insert is shown in Listing 5. In OpenCL, there are atomic operations defined for 32-bit and 64-bit integers, and if a sentinel value is used to mean that buckets are empty, then mutual exclusion doesn't need to be imposed by more explicit means such as a second flag array. An example of OpenCL code with an atomic operation is shown in Listing 6.

LISTING 6
Compact hash coding for parallel insert operation.

```
iteration = 0;
hashLocation = ((inputKey * a + c) % m) % tableSize;
oldKey = atomic_cmpxchg(&hash[2*insertLocation], -1, inputKey);
for (insertLocation = hashLocation;
     oldKey != -1 && oldKey != inputKey && iteration < maxTries;
     insertLocation = (hashLocation + iteration * iteration) % tableSize) {
     iteration++;
     oldKey = atomic_cmpxchg(&hash[2*insertLocation], -1, inputKey);
}
if(iteration < maxTries){
     hash[2*insertLocation] = inputKey;
     hash[2*insertLocation+1] = inputValue;
}
```

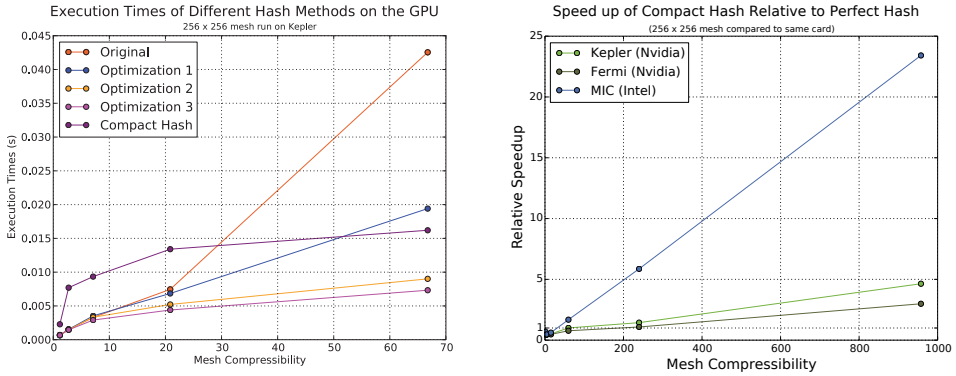
3.4. Delivering a GPU library. A library for hashing on both the CPU and parallel devices was developed to better create an abstraction barrier between neighbor calculation and hashing, to improve code modularity and reuse in our organization, and to encourage others to adopt and explore hashing techniques. The library was created in C for use by Fortran, C, and C++. OpenCL was chosen as a parallel computing language for its portability across different GPU devices.

The software coding techniques for delivering an OpenCL library are very new. OpenCL compiles at run-time on the currently selected hardware, yielding strong portability. A string containing the code is compiled for that hardware during execution. For portability, there are two situations that must be handled. First, the kernel source code should be bundled with the library so that the library and source code cannot get separated. Conventional techniques involve storing the CL code in a file and reading that file when the string is needed. We developed a string encapsulation for the source code into the hash library. The run-time compile of the OpenCL code now uses a string embedded into the library instead of a separate file. Thus, users no longer need to specify the path to the OpenCL file. Some of the functions in the library are subroutines rather than kernels. These must be present in the *application's* OpenCL source at run-time. To handle this case, the source code is embedded in a string, and the application retrieves it with a get source command, prepends it to the application OpenCL code, and then compiles it.

In OpenCL, function pointers and arrays of memory pointers are not allowed, rendering many object-oriented design concepts obsolete. To work around this, an object-based approach using macros was taken for the design of the hash library. Macros allowed us to hash values of a generic type by imitating polymorphism and inheritance. The heavy use of macros also allowed us to code key sections for both OpenCL and C, as the respective syntax does not vary too much. To allow us better error detection, the macros were expanded into separate files so that the compiler could use the correct line numbers. Another advantage of using macros in OpenCL is that code passed to the hardware is already in string format and doesn't need a separate file. This in effect embeds the OpenCL source code and, in the process, solves the problems discussed above.

We considered many use cases in developing a versatile OpenCL library. The data to be operated on may be on the host, the device, or some combination of the two. The library must be capable of handling all of these cases. In order to do so, we automatically transfer data, if necessary, to support each situation.

We are including the version of the hash library used in this paper as supplemental



(a) (b)

FIG. 4. (a) *Memory optimizations improve performance.* (b) *The compact hash method shows speed-ups on different devices.*

material. This will improve the reproducibility of this research. It is being released with the Apache license, which allows free use but requires attribution so that the impact of this work may be ascertained. The actively developed version of the library is available at http://www.github.com/losalamos/CompactHash.

4. Performance results. After developing the compact hash algorithm utilizing the rules of our cell-based AMR method to reduce memory usage and generate sparse data structures, we implement our methods on a randomly generated sample mesh to test their performance capabilities. We test the different memory optimizations we developed and explore performance variation which occurs due to spatial locality within the mesh structure. Our methods were developed to be portable across a variety of architectures, making it necessary to consider the impact hardware will have on performance and portability. After we obtain performance results, we develop an analytic model and compare to our numerical results.

4.1. Stand-alone neighbor testbed. Here we investigate the performance of our spatial compact hash to determine neighbor elements on a 256×256 coarse grid refined up to five levels. The results are shown relative to the mesh compressibility of the given problem. We define the mesh compressibility as $N_{finecells}/N_{cells}$. For this problem, the refinement levels are randomly set for each cell, resulting in 1,005,373 cells at five levels of refinement. Here we have let refinement occur at random locations in the mesh; in addition, the data is scattered randomly so that there is no spatial order to the AMR cells. Refinement is still subject to the constraint that only one symmetric bisection is allowed between neighboring cells. Randomizing refinement patterns changes the compressibility of the mesh, and randomly scattering the data changes the spatial locality of the data to be hashed. The compressibility is then $8192^2/1{,}005{,}373 = 66.75$. The compact hash only starts saving memory at a mesh compressibility of six because we need to store twice as much data per entry (key and value versus value). Also, to reduce collisions, we need to use a table with three times the number of cells, which yields a hash table load factor of 0.33.

Figure 4(a) shows the timings of each successive optimization as applied to the perfect hash, as well as the timing of the compact hash using optimization 3. Each of the optimizations adds improvement from the original perfect hash, with an espe-

cially significant difference between the original method and optimization 1. Initially, the compact hash is slower than any of the perfect hash methods, indicating that for smaller scale problems with fewer refined cells, the perfect hash is preferable. With increasing compressibility, the performance of the perfect hash using the original method and optimization 1 begins to decline, and the compact hash, despite its initial performance penalty, is faster. For the problem sets tested, the last two perfect hash memory optimizations remain faster, but the trend of scalability with mesh compressibility is a good sign for the utility of the compact hashing for large scale, compressible problems.

Figure 4(b) shows the compact hash tested across a variety of GPU architectures. These include Nvidia's M2090 (Fermi) and K20Xm (Kepler) cards (released in 2011 and 2012, respectively, and both members of Nvidia's General Purpose GPU Tesla family) and Intel's Many Integrated Core (MIC). The timings are presented as a speed-up relative to the perfect hash on each card for optimization level 3 and through a much larger compressibility range. The first thing we notice is that at a larger compressibility, the compact hash overtakes the performance of the perfect hash. This compressibility threshold varies between devices. The crossover where the compact hash becomes faster for the GPU is a compressibility of about 200. At a compressibility of 1000, the Nvidia card is three times faster with the compact hash. This is a bonus since the main goal of the compact hash is to save memory. Despite being a CPU device, the MIC shows that the compact hash is dramatically faster than the perfect hash, but with similar trends. The crossover for the MIC is a compressibility of about 20. What is not shown in the graph is that the time for the compact hash on the MIC is an order of magnitude slower than for the GPU, and the perfect hash is far slower than that. These results for the MIC were with 180 threads for the kernels and 240 for the table initialization, which is plenty of concurrency. The reason for the slow perfect hash on the MIC is that the memory access times are much slower due to cache misses or flushes and translation lookaside buffer misses. This causes the initialization of the perfect hash table to be very slow. Running with or without randomization gives nearly identical results. Strangely enough, using the compact hash on the MIC has the surprising effect of compensating for the weak memory performance and makes it possible to mitigate this performance issue. Apart from this, these results are a strong indication of performance portability, which, when combined with the code portability offered by OpenCL, demonstrates that the speed-ups from the compact hash are relatively easy to attain across architectures despite the newness of the atomic operations. These consistent performance enhancements are most likely a function of improvements at the algorithmic level which expose and utilize parallelism inherent to the problem, as opposed to custom programming for the individual devices. OpenCL is designed to be highly portable, allowing us to write routines once that can then be compiled and run on any OpenCL-compliant hardware, including multiple types of devices simultaneously.

In addition to devices, other variables to consider are cache storage and data locality. A test case for a random problem involves generating the sample mesh's refinement randomly, as well as randomizing keys in the compact hash table during the compression stage. This means that the program does not take advantage of any spatial locality. In a physical application the mesh will not be random, there will be patterns inherent in the mesh structure, and there may be some benefit to taking advantage of the cache.

We experimented with turning off randomization in generation of the hash table, which adds correlation between the spatial data and the hash table. As seen in

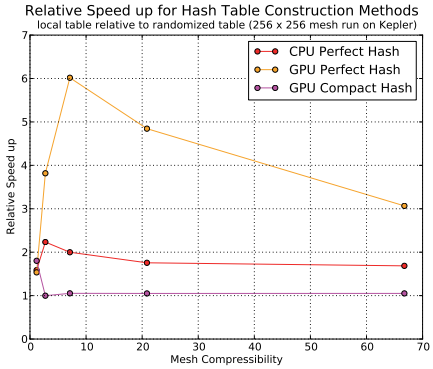


FIG. 5. *Randomizing the table construction shows the effect of cache on algorithm performance.*

Figure 5, a spatially organized hash table offers speed-up in the perfect hash methods, especially on the GPU, but does not make a significant difference for the compact hash. This is understandable since the compression scheme results in scattering. The question of utilizing cache efficiency on the GPU has to be countered against the necessity for sparsity. Taking advantage of locality with the perfect hash offers speed-up on the GPU, but introducing compressibility in the compact hash allows it to scale to larger problems.

4.2. Cost model. The cost model for the neighbor finding calculation is composed of an initialization of the hash table to the sentinel value, the insertion of all of the cell data, and the query for four neighbors for each cell. The cost model is complicated by the probabilities of collisions and the number of queries needed to find a fine neighbor, same size neighbor, or coarser neighbor.

Perfect Hash Cost (C_{PH}):

$$C_{PH} = HashTableSize * C_{Init} + N_{Cells} * C_{Write} + 4 * NQueries_{ave} * N_{Cells} * C_{Read}, \tag{4.1}$$

where $1 < NQueries_{ave} < 3$ is the average number of queries to find the neighboring cell with tries at the finer level, the same level, and, finally, a coarser level. $NQueries_{ave}$ should be close to 2 for a typical AMR mesh.

Compact Hash Cost with generic probing (C_{CH}):

$$\begin{aligned} C_{CH} = {} & \frac{N_{Cells}}{LF} * C_{Init} \\ & + N_{Cells} * ((C_{Probe} + C_{Atomic}) * SuccProbeSeq_{ave} + C_{Write2Words}) \\ & + N_{Cells} * (C_{Probe} * (SuccProbeSeq_{ave} \\ & \quad + (NQueries_{ave} - 1) * UnsuccProbeSeq_{ave}) + C_{Read2Words}). \end{aligned} \tag{4.2}$$

The load factor is the fraction of the hash table that is filled:

$$LoadFactor(LF) = N_{Cells}/HashTableSize_{Compact}. \tag{4.3}$$

By rearranging,

$$HashTableSize_{Compact} = N_{Cells}/LF, \tag{4.4}$$

and that is what shows up in the initialization cost term in (4.2). Note that the compact hash has to write both the key and the value to aid in the collision handling, whereas the perfect hash has to write only the value. These are distinguished by the $C_{Write2Words}$ and $C_{Read2Words}$ in the compact hash and the C_{Write} and C_{Read} in the perfect hash. Though twice as much information is being written or read, cache effects make the cost of the additional reads and writes less than a factor of two.

The compact hash cost with linear probing (C_{CHL}) and compact hash cost with quadratic probing (C_{CHQ}) are obtained by substituting $LinProbeSeq_{ave}$ or $QuadProbeSeq_{ave}$ for $ProbeSeq_{ave}$ in the above equation. Note also that the initial insertion will never have a collision, but by the final insertion, the collision probability is the load factor of the table. Thus the collision cost must be integrated over the insertions starting at 0 and ending at the final load factor. There is a closed form solution for the linear probes (from Knuth [13]), but the quadratic probe sequence contains an integral in the equation (from Bell [3]).

The average successful linear probe sequence, $SuccLinProbeSeq_{ave}$, is

$$\frac{1}{2}\left(1+\frac{1}{1-LF}\right). \tag{4.5}$$

The average unsuccessful linear probe sequence, $UnsuccLinProbeSeq_{ave}$, is

$$\frac{1}{2}\left(1+\frac{1}{(1-LF)^2}\right). \tag{4.6}$$

The average successful quadratic probe sequence, $SuccQuadProbeSeq_{ave}$, is

$$\frac{1}{LF} * \int_{x=0}^{LF} \left\{ -\left(\frac{1}{x}\right) ln(1-x)(x-(1-e^{(-x)}))dx \right\}. \tag{4.7}$$

The average unsuccessful quadratic probe sequence, $UnsuccQuadProbeSeq_{ave}$, is

$$\frac{1}{LF} * \int_{x=0}^{LF} \left\{ -\left(\frac{1}{x}\right) ln(1-x)(x-(1-e^{(-x)}))dx \right\} - \frac{ln(1-LF)}{LF}. \tag{4.8}$$

To get a sense of the magnitude of the collision frequency and cost, some examples are helpful. In a hash table with chaining, a load factor of 0.5 will result in an average of 1.25 buckets that are checked per search. In a hash table using open addressing with linear probing for collision resolution, a 0.3 load factor will traverse on average 1.21 buckets per search from Tables 3 and 4 of section 6.4 of [13]. The lower load factor for the open addressing is due to collisions being resolved within the table memory space rather than in an external linked list. This requires a larger table size for the same collision probability, but is much easier to implement on a GPU. Comparing different open addressing methods at a load factor of 0.5, there will be on average 1.50 buckets per search for the linear probing [13] and 1.44 buckets for the quadratic probing [3]. Double hashing gets even lower average buckets per search, but does not benefit as much from cache performance.

The additional cost for the collisions is not simply the average buckets touched, which for a load factor of 0.3 would be only 20–25% more. The key and value must be stored so that the owner of the bucket can be determined, whereas without the possibility of collision, only the value must be stored. Offsetting this is the reduced cost of the initialization of the smaller hash table.

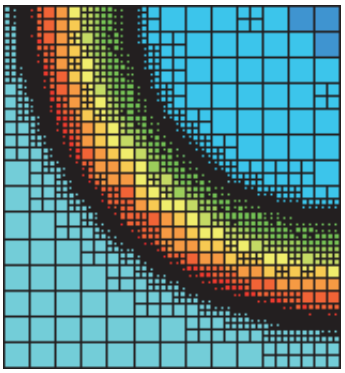


FIG. 6. *This AMR grid, taken from a CLAMR simulation, shows the mesh refinement at the boundary of a water wave.*

5. Application to scientific computing. We have shown that the compact hash offers a robust, efficient, and easy-to-implement method for neighbor determination on a randomly generated sample mesh. We now wish to demonstrate that our methods are viable to implement in a working scientific application where complications such as patterns in the spatial data arise. Our other goal in this section is to show that the compact hash scales to large, finely resolved meshes.

For scientific computing applications, using a hash table simplifies access to the cell information and also simplifies the performance of various spatial operations. Robey, Nicholaeff, and Robey [23] showed that, for every spatial mesh operation, there is an efficient $O(n)$ hash-based algorithm. The simplest method, the perfect spatial hash, avoids the possibility of collisions by directly mapping to a hash table with buckets corresponding to the spatial layout of the grid at the finest level of refinement. At the finest level, each cell is of a size $\triangle_{min}$. For scientific applications on AMR grids with many levels of refinement, it is computationally costly to create and destroy large arrays every timestep. By exploiting the rules of our AMR implementation, we reduce the number of buckets which contain information about the spatial layout of the grid. This reduction yields a sparse, one-dimensional, perfect hash table. This sparse array may be compressed to a compact hash table, but with the cost of dealing with data collisions. This work extends spatial hashing techniques to compact hashing, allowing these methods to be viable for large problems on finely resolved meshes. This extension has broad implications in the field of computational science.

Neighbor determination, a computationally expensive spatial operation in AMR computing methods, can be greatly enhanced using hashing techniques applied to discretized data. As an example, we implement both the compact and perfect spatial hash in a cell-based AMR shallow-water hydrodynamics code, CLAMR, as shown in Figure 6. Merging these two methods into a hybrid implementation allows the developer to tailor algorithm performance to the application characteristics and available computing resources. While neighbor determination was the only operation explored in this current work, the properties of computational meshes allow for hash-based approaches to be utilized in a variety of scientific applications such as connectivity remapping in Lagrangian hydrodynamics schemes, equation-of-state table look-ups, and ray-tracing in Monte Carlo–type schemes, just to name a few; the extensions of spatial hashing are ubiquitous in scientific computing. With the ease of parallelizing these methods for multicore processors and GPUs, spatial hashes provide the computational scientist with a powerful tool for tackling a variety of physical problems.

5.1. Implementation in hydrodynamics application. To test our algorithms, we have developed a simple shallow-water hydrodynamics scheme capable of running on GPUs and CPUs. The utilization of AMR with GPU portions of the code written in the OpenCL 1.1 standard prompted the name CLAMR to be adopted. CLAMR is a second-order accurate hydrodynamics scheme evolved on a cell-based Cartesian adaptive mesh. CLAMR is built on a heterogeneous platform utilizing GPU parallelization in tandem with CPU memory management to achieve significant speed-up and performance. The code was built to be highly versatile, allowing us to test and compare the performance of our developed algorithms across a variety of architectures [20].

We evolve the conservative shallow-water wave equations using a total variation diminishing finite difference scheme based on a two-step Lax–Wendroff method [15] in conjunction with a minmod symmetric flux limiter developed by Davis [8] and Yee [25] to provide an upwind weighted artificial viscosity term. The Lax–Wendroff method is second-order accurate in space and time, and hence, it is a suitable choice for smooth regions. The wave equations in conservative form are given by

$$\frac{\partial h}{\partial t} + \frac{\partial (hu)}{\partial x} + \frac{\partial (hv)}{\partial y} = 0, \tag{5.1}$$

$$\frac{\partial (hu)}{\partial t} + \frac{\partial}{\partial x}\left(hu^2 + \frac{1}{2}gh^2\right) + \frac{\partial}{\partial y}(huv) = 0, \tag{5.2}$$

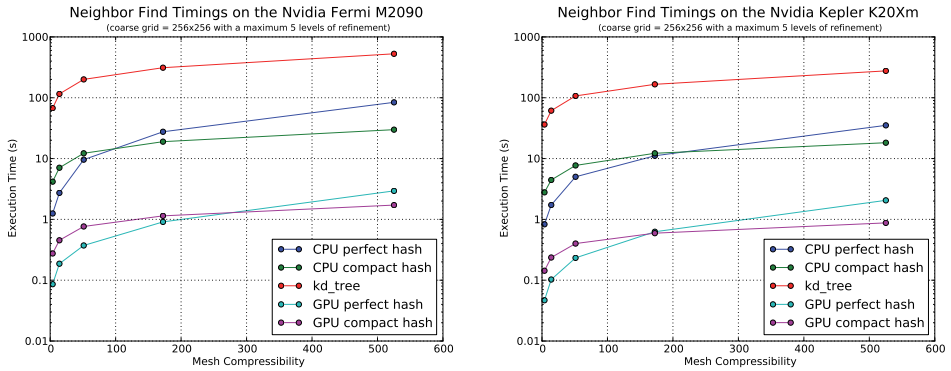
$$\frac{\partial (hv)}{\partial t} + \frac{\partial}{\partial x}(hvu) + \frac{\partial}{\partial y}\left(hv^2 + \frac{1}{2}gh^2\right) = 0, \tag{5.3}$$

where h is the height of a column of water, g is the acceleration due to gravity, and u and v are the wave velocities in the x and y directions, respectively. The water is approximated to be incompressible, allowing height to be substituted for mass while the pressure is approximated as $gh^2/2$. For a more rigorous presentation, see the sections in Landau and Lifshitz [14] on long gravity waves and shallow-water theory.

We impose several conditions when implementing our cell-based AMR scheme: (i) Two neighboring cells are allowed to differ by no more than one level of refinement so as to reduce errors generated by a reflection of information at cell boundaries during the refinement step. (ii) A cell is refined by symmetrically bisecting along all axes. (iii) Cells are refined only over regions of physical interest where high resolution is necessary to the calculations. (iv) Refinement precedes the arrival of steep gradients or wavefronts. (v) Grid elements are indexed in standard Cartesian coordinates.

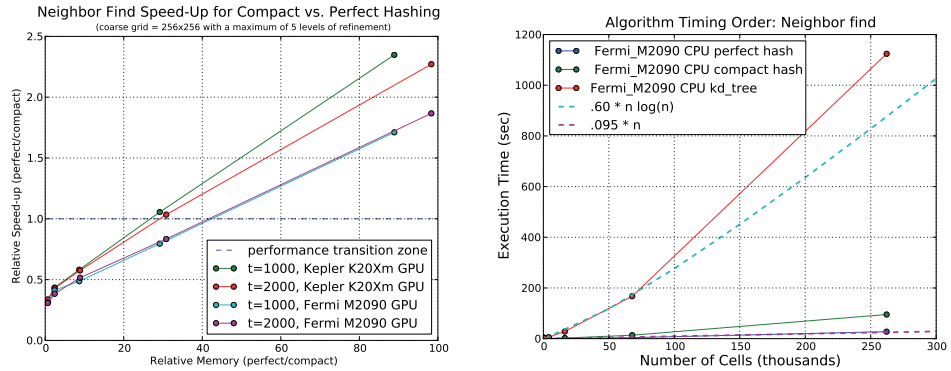
The nature of AMR grids will result in patterns of concentrated fine mesh around wavefronts and other areas with steep gradients. Also, refinements will come in blocks of four for two-dimensional grids. The randomization of the compact hash is designed to break up these patterns and reduce collisions in the hash table while saving considerable memory in comparison to the perfect hash.

5.2. Performance results using CLAMR. Here we explore the performance of perfect and compact spatial hashing across different devices within the context of a hydrodynamics application. We attained timings for determining neighbor cells as a function of the compressibility of the mesh using single node processing on Nvidia's Kepler K20Xm and Fermi M2090 graphics cards and on the 3.5GHz Intel i7 CPU, as shown in Figure 7. In these calculations, we began with a coarse grid of 256×256 elements and allowed up to five levels of refinement. At five levels of refinement, the perfect hash table can exceed 67 million elements. In the wave simulation only a small circular region around the wavefront is refined, and, therefore, the mesh is



(a) (b)

FIG. 7. *Hashing methods improve neighbor find timings in CLAMR on* (a) *Fermi GPU and* (b) *Kepler GPU.*



(a) (b)

FIG. 8. (a) *Relative memory usage scales linearly with relative performance in compact versus perfect hashing methods.* (b) *The scaling of the hashing method is an improvement upon the scaling of the k-D tree as the problem size increases.*

very compressible. Here we have mesh compressibility calculated as the ratio of the number of cells at the finest level of the mesh to the number of cells in the AMR mesh after $t = 1000$ iterations in the simulation.

The perfect hash achieves higher computational speed at low compressibility, but eventually the compact hash has increased speed and reduced memory usage (see Figures 7(a), 7(b), and 8(a)). This occurs because the compact hash requires extra operations for querying and accessing keys due to data collisions, while the perfect hash has higher memory cost for initializing sparse hash tables. As we increase beyond four levels of refinement, the array size of the perfect hash becomes exceedingly large, offsetting the costs of queries and data access in the compact hash. In the vicinity of this transition region, the code developer may tailor the method used to suit the specific needs of the problem or the available computational resources.

The performance of the compact hash is rather remarkable. The compact hash on the CPU is roughly 20× faster than the k-D tree method, and on the GPU it is roughly 200× faster. Another trend to note in Figures 7(a) and 7(b) is that the perfect

hash has a slope that approaches that of the k-D tree method. By extrapolating the slope, it becomes evident that the perfect hash will overtake the k-D tree method, becoming slower at some higher mesh compressibility. However, the compact hash has a slope which tracks that of the k-D tree method. This trend demonstrates the scalability of the compact hash with larger problem sets.

In support of the scalability on the CPUs, we show the $O(n)$ behavior of the perfect hashing method versus the $O(n \log n)$ behavior of the k-D tree method in Figure 8(b). Here we calculated execution time as a function of the number of cells on the coarse grid. Although the compact hash does not exactly scale with the perfect hash, the figure indicates that its behavior is closer to $O(n)$ than to $O(n \log n)$. These calculations were performed only on the Fermi GPUs.

To further demonstrate the memory savings of the compact hashing method, we plot speed-up as a function of memory usage for the perfect hash normalized by the compact hash in Figure 8(a). Again, we tested on both the Kepler and Fermi GPUs. Due to the symmetry of the spherical wave, a longer iteration corresponds to a larger wave radius, and, therefore, the mesh will adapt a larger number of refined cells to capture the steep gradients around the wavefront. We wanted to see how the compact hash performed as the number of refined cells increased on a fixed allowable refinement, and so we show the results from simulations at two maximum time iterations, $t = 1000$ and $t = 2000$. We see that the performance improvements of the compact hash scales linearly with the memory usage improvements of the compact hash. For example, when the memory used by the perfect hash is 20–40× the memory needed for compact hashing, we are crossing the threshold where the compact hash becomes the faster method. Again, the programmer may tailor the method being used to suit his or her needs.

We have not tested the compact hashing technique directly against an oct-tree implementation. However, Ji, Lien, and Yee [11] showed that hashing was twice as fast as their oct-tree implementation on a CPU. However, there is the additional benefit for us that the hashing is easier to implement on the GPU. Our approach also does not need a lot of additional data for each cell or to carry the parent cells along, making the compact hash very memory efficient. The overhead of creating the neighbor pointers on the fly is about 6% on the CPU and 7.5% on the GPU, which is a much smaller fraction than the 9.6% reported by Ji, Lien, and Yee [11], the 17–20% by Khokhlov [12], and the 25% by De Zeeuw and Powell [9].

6. Conclusions. We extended the perfect hash techniques to a highly parallel compact spatial hashing method for determining neighbor cells on an AMR grid and showed that compact hashing offers speed-up and memory savings over the perfect hash. We demonstrated that compact hashing is a memory efficient method of exploiting $O(n)$ algorithms for computational mesh operations. Spatial hashes provide a full range of performance and memory options for the computational code developer, allowing scientific applications to exploit the performance enhancements of hashing while operating over diverse ranges of resolution in large problem sets.

We applied spatial hashing methods on AMR grids with 67 million elements at the finest level of refinement, and our results showed a 20× speed-up on the CPU compared to a k-D tree method, and a 200× speed-up over the CPU k-D tree on every GPU device tested. Beyond four levels of refinement, the compact hash is faster than the perfect hash and reduces the memory usage by a factor of approximately 30. Hybridization of the perfect and compact methods gives the developer the ability to tailor the method used to the constraints on time or available resources. The

thread-based parallelism of the compact hash method allows performance portability on CPU and GPU architectures. CLAMR demonstrates the use and performance of hashes for applications on future architectures. The applications of spatial hashing methods are ubiquitous in scientific computing.

Future avenues. Hash tables can be employed for spatial operations such as sorts, remaps, and table look-ups; neighbor determination is only one operation which benefits from the methods we have developed. In the future, data sparsity could be exploited in these operations to allow the use of a compact hash. These improved algorithms could provide a broader model for developers to follow when tailoring methods to suit their specific application. The scalability of hash algorithms in parallel computing with more distributed memory could be evaluated. Further work on the hash library could also be pursued. Although we implemented many standard hashing optimizations in the library, there are plenty of other optimizations that could be explored. The performance of these optimizations and their compatibility with the hash-based algorithms in the massively parallel environment could then be studied.

Acknowledgments. We would like to acknowledge our families, who have been continuous sources of encouragement. Special thanks go to to Scott Runnels for organizing the wonderful Computational Summer Physics Workshop, and to Los Alamos National Laboratory for hosting us. We pay homage to the technical wizards who protect and maintain Darwin, our favorite experimental architecture computer in CCS-7. We offer many thanks to Rachel Robey for offering her wisdom and experience in the matters of hashing. Also, we'd like to thank David Nicholaeff for his work on the memory write optimizations that laid the foundation for this work. Lastly, we thank the excellent reviewers for their thoughtful criticism that made this a better paper.

REFERENCES

[1] D. A. Alcantara, *Efficient Hash Tables on the GPU*, Ph.D. thesis, Department of Computer Science, University of California, Davis, Davis, CA, 2011.

[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, *Real-time parallel hashing on the GPU*, ACM Trans. Graph., 28 (2009), 154.

[3] J. R Bell, *The quadratic quotient method: A hash code eliminating secondary clustering*, Comm. ACM, 13 (1970), pp. 107–109.

[4] M. J. Berger and J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comput. Phys., 53 (1984), pp. 484–512.

[5] R. P. Brent, *Reducing the retrieval time of scatter storage techniques*, Comm. ACM, 16 (1973), pp. 105–109.

[6] J. L. Carter and M. N. Wegman, *Universal classes of hash functions*, J. Comput. System Sci., 18 (1979), pp. 143–154.

[7] Z. J. Czech, G. Havas, and B. S. Majewski, *Perfect hashing*, Theoret. Comput. Sci., 182 (1997), pp. 1–143.

[8] S. F. Davis, *A simplified TVD finite difference scheme via artificial viscosity*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 1–18.

[9] D. DeZeeuw and K. G. Powell, *An adaptively refined Cartesian mesh solver for the Euler equations*, J. Comput. Phys., 104 (1993), pp. 56–68.

[10] H. Gao, J. F. Groote, and W. H Hesselink, *Efficient Almost Wait-Free Parallel Accessible Dynamic Hashtables*, Technical report, CS-Report 03-03, Eindhoven University of Technology, Eindhoven, The Netherlands, 2003.

[11] H. Ji, F.-S. Lien, and E. Yee, *A new adaptive mesh refinement data structure with an application to detonation*, J. Comput. Phys., 229 (2010), pp. 8981–8993.

[12] A. M. Khokhlov, *Fully threaded tree algorithms for adaptive refinement fluid dynamics simulations*, J. Comput. Phys., 143 (1998), pp. 519–543.

[13] D. E. Knuth, *The Art of Computer Programming, Volume* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
[14] L. D. Landau and E. M. Lifshitz, *Fluid Mechanics*, Course of Theoretical Physics 6, Pergamon Press, Oxford, UK, 1987.
[15] P. Lax and B. Wendroff, *Systems of conservation laws*, Comm. Pure Appl. Math., 13 (1960), pp. 217–237.
[16] W. D. Maurer, *Programming technique: An improved hash code for scatter storage*, Comm. ACM, 11 (1968), pp. 35–38.
[17] M. Mitzenmacher and S. Vadhan, *Why simple hash functions work: Exploiting the entropy in a data stream*, in Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 2008, pp. 746–755.
[18] J. K. Mullin, *A caution on universal classes of hash functions*, Inform. Process. Lett., 37 (1991), pp. 247–256.
[19] J. I. Munro and P. Celis, *Techniques for collision resolution in hash tables with open addressing*, in Proceedings of the 1986 ACM Fall Joint Computer Conference, IEEE Computer Society Press, Los Alamitos, CA, 1986, pp. 601–610.
[20] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey, *Cell-Based Adaptive Mesh Refinement Implemented with General Purpose Graphics Processing Units*, Technical Report LA-UR-11-07127, Los Alamos National Laboratory, Los Alamos, NM, 2011.
[21] W. W. Peterson, *Addressing for random-access storage*, IBM J. Res. Dev., 1 (1957), pp. 130–146.
[22] F. Proth, *Théorèmes sur les nombres premiers*, C.R. Acad. Sci. Paris, 85 (1878), p. 926.
[23] R. N. Robey, D. Nicholaeff, and R. W. Robey, *Hash-based algorithms for discretized data*, SIAM J. Sci. Comput., 35 (2013), pp. C346–C368.
[24] F. A. Williams, *Handling identifiers as internal symbols in language processors*, Comm. ACM, 2 (1959), pp. 21–24.
[25] H. C. Yee, *Construction of explicit and implicit symmetric TVD schemes and their applications*, J. Comput. Phys., 68 (1987), pp. 151–179.
[26] D. P. Young, R. G. Melvin, M. B. Bieterman, F. T. Johnson, S. S. Samanth, and J. E. Bussoletti, *A locally refined finite rectangular grid finite element method: Application to computational fluid dynamics and computational physics*, J. Comput. Phys., 92 (1991), pp. 1–66.