HASH-BASED ALGORITHMS FOR DISCRETIZED DATA

RACHEL N. ROBEY[†‡], DAVID NICHOLAEFF[§¶], AND ROBERT W. ROBEY[§]

Key words. Hash-based algorithms, Sorting, Neighbor calculation, Remap, GPGPUs, Mesh algorithms

Abstract. We explore the idea that all mesh operations in numerical methods can be implemented with efficient hash-based algorithms. The hash-based methods are presented with a view toward highly parallel implementations on both the CPU and GPU. A general set of applications, including sorting, neighbor calculation, remapping, and table look-up, demonstrate the practical value and several orders of magnitude speed-up of hash-based implementations.

1. Background. Historically, the development of sorting algorithms has largely emphasized comparison-based methods which even in a best case scenario have an ideal limit of $O(n \log n)$ complexity. One of the leading and most heavily used of these algorithms is the quicksort[9]. Also ranking as well-known and established implementations of comparison-based sorts are the heapsort[20] and the merge sort[19]. Previous work has explored the subject of non-comparison-based sorts, but their utilization was largely relegated to specialized and restricted situations. Although the performance of many of these non-comparison-based sorts appears to be $O(n)$ from their complexity, in practice, the constant multiplier often makes them slower in serial implementations. Representative methods include variations of the bucket sort including the bitonic sort[2], and radix sort[10].

The emergence of parallel computing later sparked a renewed interest in these non-comparison based sorts in order to take advantage of their often easier parallelization. The first GPU implementations to surface prior to the development of GPU computing languages used the bitonic sort, accommodating the limitations of the early OpenGL operations. With the advent of CUDA, and more recently OpenCL, there has been a shift to using radix sorts. Neither has parallelization been limited to just non-comparison-based methods. A successful implementation of the quicksort was recently demonstrated[18]. The latest class of GPU sorts were first introduced by Harris and Sengupta with their radix sort implementations[7, 18] based on the algorithm by Blelloch[3]. These algorithms minimized the work through two $\log n$ passes (work efficiency), first an upsweep reduction followed by a down-sweep pass in the prefix scan. It was soon realized that the number of steps (step efficiency) within a synchronized work group was more important, and faster implementations based on this concept were established from the basis of an algorithm by Hillis and Steele[8, 16]. Dotshenko demonstrated an approach that used a first step reduction to reduce the global memory needed on the GPU[4]. Further improvements have optimized the work-group sizes at each level to fully utilize the GPU computing resources[16, 11]. In his discussion of the radix sort, Merrill introduced ideas for future work tailoring the sort to exploit the underlying data by exposing more details of the implementation to the user, though he did not outline an exact mechanism to do so. A hash-based

*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.

[†]Corresponding Author: email rnrobey@gmail.com

[‡]University of New Mexico - Los Alamos, Los Alamos, NM

[§]XCP-2 Eulerian Codes, Los Alamos National Laboratory, Los Alamos, NM

[¶]Department of Physics & Astronomy, University of California at Los Angeles, Los Angeles, CA

sort was introduced by Gilreath[5], though without an important application, it has received little attention.

The sort is just one of the foundational computational functions that have been dominated by comparison-based algorithms; neighbor finding and remapping also often appear as fundamental parts of a numerical code. The cost and generality of neighbor calculations often limit the implementation of more innovative mesh representations. Current methods for neighbor calculations are generally tree-based, using either k-D trees or octrees. However, a recent work by Griebel and Zumbusch[6] presented a hash-based algorithm on the CPU with successful results, demonstrating that the idea of hashing and non-comparison-based methods apply not only to the sort, but to other expensive operations as well. The remapping operation is defined as the transferring of a conserved quantity from one mesh representation to another. Methods for remapping are predominantly tree-based, again falling back on the k-D tree[13]. These are $O(n \log n)$ in complexity, restricted by the same barrier found in comparison-based sorts. They are also difficult to parallelize, though there has been some success on the GPU[21].

In this paper, we seek to show that there is a wide-spread set of applications essential to numerical computing that can benefit from hash-based algorithms and that there is a way to exploit the underlying data structures in a systematic fashion for better performance.

2. Methodology. Over the course of the last decade, the lower limit of many common comparison-based algorithms has been increasingly challenged by methods such as hashing. By directly mapping data to buckets, hashing enables the bypass of comparisons to achieve an ideal limit proportional to $O(n)$, opening up significant speed-up possibilities compared to the $O(n \log n)$ algorithms that it is coming to replace. The hash-based methods presented in this paper are both a culmination and specialization of several accepted methods[15]. We develop a unique type of hash, the perfect spatial hash, that enables a straightforward parallel coding.

Making effective use of this hash-based method requires an understanding of the underlying properties of the data. As such, we define a general concept of discretized data in order to exploit spatial properties, such as the distribution and uniqueness of spatial coordinates.

Section 2.1 serves to develop the framework for a computational mesh in the context of differential discretized data. This involves the refinement and coarsening of cells to achieve the same order of error across the entire mesh. The idea here is that computational resources should not be wasted in regions which already have sufficient resolution, while regions with insufficient resolution demand more resources to improve accuracy. We go on in Section 2.2 to present an alternative mapping of the domain more suitable for spatial operations. With careful design, this representation may also be used as a hash table. The extension to the computational algorithm of the spatial hash is defined in Section 2.3 and Section 3 follows with examples of its application to four mesh operations demonstrating significant speed-ups.

Note that Sections 2.1. and 2.2 are very mathematical. An understanding of hash-based applications can be achieved independently of them. These two sections are provided to describe the extent of the theoretical underpinnings of hash-based algorithms.

With a clear description of the spatial structure of the data, spatial hashing flows naturally. Hence the goal of the following collection of definitions for discretized data is to provide a mathematical basis suggesting that an efficient hash-based algorithm

exists for all mesh operations in numerical methods.

2.1. Differential Discretized Data. The foundation of our hashing method is rooted in the exploitation of the relatively even distribution of discretized data.

By *discretized data* we mean an arbitrary collection* of information $\{s_l\}$, elements s_l similar in nature, selected by the algorithm designer, such that each element has unique characteristics and properties associated to it which allow a surjective (onto) mapping f of the collection of information to a metric space (A, d) with a minimal distance parameter:

$$f : s_l \mapsto A_i \subseteq A = \bigcup_{j}^{|\{A_j\}|} A_j, \tag{2.1}$$

where $\{A_j\}$ is a finite cover for A defined by f with $A_i \in \{A_j\}$ and the non-negative property of the metric, $d(x, y) \geq 0$ $(\forall x, y \in A)$, is augmented by

$$d(A_i, A_j) \geq \delta. \tag{2.2}$$

Here $\delta > 0$ is some finite real number — the minimal distance between the centers of subspaces (i.e. cells) A_i and A_j. For brevity, we define $s_j = \{s_l \mid f(s_l) = A_j\}$. It is important to note here that the data becomes discretized only once it has been given a representation, i.e. associated with A.

Furthermore, we say that the representation of the discretized data A is *connected* if

$$\forall i[\exists(j \neq i)] \ni A_i \cap A_j \neq \emptyset, \tag{2.3}$$

and that it is *tessellized* if

$$\forall i \forall (j \neq i) \, [A_i \cap A_j = \partial A_i \cap \partial A_j] \tag{2.4}$$

where ∂A_i represents the surface (i.e. boundary) of A_i. In the context of these characterizations, we say that A_i and A_j are *neighbors* if and only if $A_i \cap A_j = \partial A_i \cap \partial A_j \neq \emptyset$.

As a quick example, let's model a mass density field in 1-D over $I = [0, 10]$. Our collection of information is the set of relevant variables extending across the domain; $\{s_l\}$ would then become $\{(\rho(x),\ p(x))\}$, the elements being pairs of density ρ and pressure p at points x in the interval I. To move to a uniform regular grid, one choice for f is

$$f(\rho(x),\ p(x)) = \big[\lfloor x \rfloor, \lfloor x \rfloor + 1\big] \tag{2.5}$$

which has $\delta = 1$ and $A = \bigcup \{[0, 1], [1, 2], \ldots, [10, 11]\}$. Hence a computational mesh with state variables at the cell centers is an example of connected and tessellized discretized data; note that this is a result of the choice of f and the subsequent representation A.

We next formalize the sense of a relatively even distribution of discretized data by extending the notion to differential discretized data. By *differential discretized*

*Throughout this section, i, j, k serve to show that a set is indexed, typically via some spatial ordering; l also serves to show that a set is indexed, however it represents a more abstract ordering on the set.

data we mean connected and tessellized discretized data with two added properties: $m(A_i)/m(A_j) \approx 1$ if A_i and A_j are neighbors ($m(A_i)$ is the measure of A_i, i.e. its size; while we are not specifying a minimum size for the A_i, an appropriate selection of $\{s_l\}$ by the algorithm designer will accomplish this), and

$$d(A_i, A_j) = g_\epsilon(\nabla_{\overline{ij}}, \delta), \tag{2.6}$$

where g_ϵ, a function of the gradient over A and parameterized by δ, is constructed as follows. Choose $\epsilon > 0$. We define $\overline{s_i}$ as a representative value of s_i which exists over all of A_i and found as a function of s_i. For most applications, an average of s_i over A_i is an appropriate representative value, though the function or means of finding the average are constructed case by case. Here we assume s_i is a collection of continuous information (and integrable over A_i), as this is commonly encountered in numerical modeling, and find the average as

$$\overline{s_i} = \frac{1}{m(A_i)} \int_{A_i} dA_i \cdot s_i. \tag{2.7}$$

We then define $\epsilon_i = \epsilon_i(s_i, \overline{s_i})$ as some general error function. Again we proceed with a specific example, in consideration of our example definition of $\overline{s_i}$, where we set

$$\epsilon_i = \max(|s_i - \overline{s_i}|). \tag{2.8}$$

Next we define a "computable gradient" over A, with $s_{ij} = s_i \cup s_j$:

$$\nabla_{\overline{ij}} \equiv \frac{\overline{s_i} - \overline{s_j}}{d(A_i, A_j)} = \nabla s_{ij} + O(\epsilon_i) + O(\epsilon_j). \tag{2.9}$$

This is one definition available; the purpose here is that the algorithm designer constructs a discrete gradient to model a continuous gradient. Finally, we construct $g_\epsilon(s_i, s_j, \delta) = g_\epsilon(\nabla_{\overline{ij}}, \delta)$ by choosing the A_i and hence the $d(A_i, A_j)$ such that those choices minimize

$$\overline{\left|\epsilon - \left[\nabla_{\overline{ij}} - \nabla s_{ij}\right]\right|} \;\&\; \sigma\left(\left|\epsilon - \left[\nabla_{\overline{ij}} - \nabla s_{ij}\right]\right|\right) \tag{2.10}$$

while maintaining $d(A_i, A_j) \geq \delta$.

The goal with this definition of differential discretized data is that the representation of the discretized data is homogenized across the domain towards our free parameter ϵ with

- minimal variance σ,
- a lower bound on the metric via the parameter δ,
- and an upper bound on the difference between adjacent cell sizes.

As an example, in the context of a 2-D adaptive mesh, we are refining and coarsening cells ($A_i = A_{x,y}$) such that the error in the calculations is $O(\epsilon)$ while simultaneously enforcing a minimal cell size of $O(\delta^2)$. Additionally, through the neighbor condition, we enforce a maximum jump in adjacent cell size.

2.2. Hashing Discretized Data. The formalism above serves to define discretized data in a general, abstract sense. However, we develop the method with an emphasis on the examples of computational meshes and arbitrary mesh structures.

The definition encompasses adaptive meshes, unstructured meshes, and potentially many others. In this section, we perform a second mapping (hash) with the intention of constructing a regular, structured hash table as a reference on which to perform the spatial operations of our irregular discretized data. A natural hashing mechanism arises from the spatial properties assumed by discretized data when associated with a metric space.

We characterize *spatial hashing* as mapping a metric space (A, d_A) associated with differential discretized data to the powerset of a finite cover $\{B_k\}$ of a new metric space (B, d_B):

$$h : A \to \mathcal{P}\left(\{B_k\}\right) \quad (A_i \not\mapsto \emptyset \; \forall i)$$

The mapping is performed through the hash function h, where spatial locations in A have correspondening spatial locations in B. The hash table, represented by $\{B_k\}$, is a set of elements that cover a representation of the domain of the original data. By mapping to a power set of $\{B_k\}$ (i.e. the set of all subsets), we allow each element to map to either one or multiple elements in the hash table. A more visual idea of the process of spatial hashing is shown in Figure 2.1 in both 1-D and 2-D to supplement the following mathematical description.

This paper addresses *perfect spatial hashing*, a specific variant of spatial hashing taking the hash function h to be injective (one-to-one); every cell is mapped to one or several unique buckets without collisions. Since we are mapping to a power set, this condition is satisfied by the following: if $A_i \mapsto \left\{B_{i_1}, B_{i_2}, \ldots, B_{i_{n_i}}\right\}$, none of the elements in $\{B_{i_m}\}$ where $1 \leq m \leq n_i$ are contained in any of the subsets mapped to by the other elements $A_j (j \neq i)$.

To achieve a unique mapping for a hash table of uniform subspaces covering the domain, the hash table must have sufficient resolution to ensure spatial locations, usually cell centers or vertices, are not contained in the same subspace of B. The resolution is bounded by specifying $\Delta_{\min}$ for each spatial direction:

$$0 < \Delta_{\min_{x_i}} \leq \delta_{x_i} \tag{2.11}$$

with $\delta_{x_i} = \min(\{d(A_i, A_j)\}_{x_i})$, the minimal difference between cell centers. δ could also be based on alternative spatial locations such as vertices. The structure of some meshes such as the AMR mesh shown in Figure 2.1, the distance between cell centers may be conveniently equivalent to the minimal cell size. Thus the finite cover (hash table size) is equivalent to the domain divided by the determined size of each subspace in the hash table:

$$|\{B_k\}| = \frac{m(A)}{|\Delta_{\min}|} \tag{2.12}$$

The projection of the computational mesh to an abstract mesh allows quick setup and access to the cells. And though the structure of the data is essential to the selection of $\Delta_{\min}$, it also has a degree of freedom to be varied per the discretion of the algorithm designer. By choosing $\Delta_{\min}$ close to its upper limit, the number of buckets $|\{B_k\}|$ can be minimized. The formalism used to define differential discretized data allows the management of computational cells to be optimized through a careful consideration of the problem decomposition and appropriate choice of data representation and parameters ϵ, δ, and $\Delta_{\min}$.

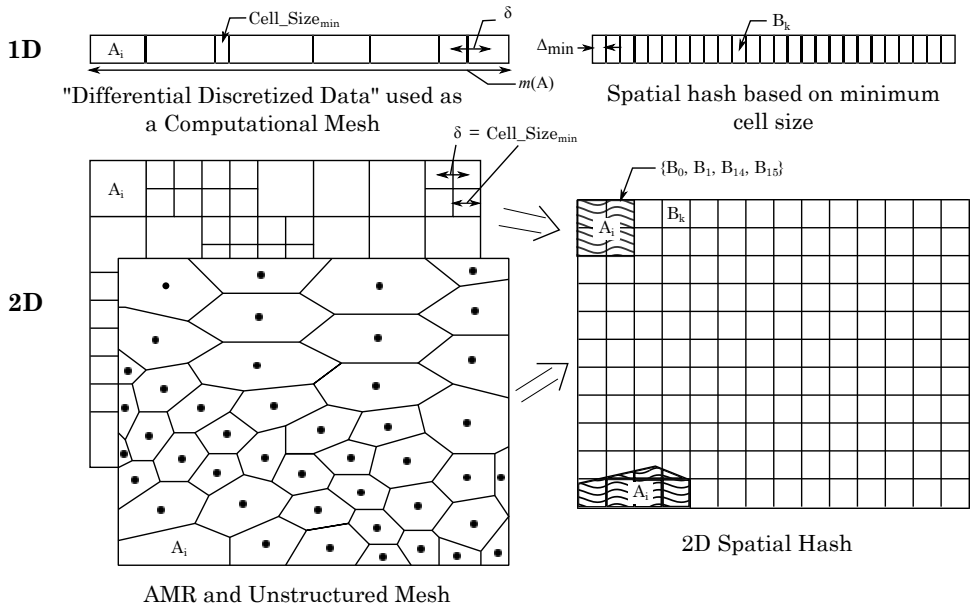

FIGURE 2.1. *The hash mesh is a structured mesh designed to facilitate mesh operations and linked to the computational mesh by* $\Delta_{\min} \leq \delta$ *thereby ensuring a perfect hash.*

2.3. Hashing Methodology. The heart of the hash-based approach in all of the following algorithms is the mapping of an original dataset of keys to a temporary, well-ordered hash table which can be more easily manipulated for spatial operations. The hash table divides the domain into a set of buckets spanning a range equal to a determined minimum $\Delta_{\min}$. This minimum exists as a consequence of the structure of discretized data and motivates the form of the hash table and corresponding hash function used in mapping. From a 1-D perspective, each key can then be placed in its respective bucket through a simple ratio:

$$b_k = \left\lfloor \frac{X_i - \min(X_i)}{\Delta_{\min}} \right\rfloor \tag{2.13}$$

with b_k referring to the index of the bucket (i.e. hash table element) to which a key should be mapped and X_i to the spatial coordinate of the subspace/key. The floor of the ratio is taken to get an integer for the bucket index. The significance of $\Delta_{\min}$ as a delimiter is that it ensures, using the properties of the data, that no collisions will occur; that is, every key returns a unique index and will not be mapped to the same bucket as another key. We create this perfect hash analytically, without relying on the odds of encountering many collisions in a large hash table as has been previously done.

A similar approach applies to 2-D problems and beyond. The multi-dimensional domain can also be partitioned by a generalized minimum size, using minimal lengths for each spatial axis. There is some complication in the ordering that must be accounted for in the mapping, but the use of $\Delta_{\min}$ continues to be a key part of designing the hash function as a perfect hash. A more complete discussion accompanies the hash-based algorithms descriptions in Section 3.

The major trade-off of a perfect spatial hash is the memory use. The size of a hash table is determined from the range and a minimum difference of a dataset whether or not such a fine layer of buckets is needed for the entirety of the hash table. Accordingly, more uniformly-distributed data is better suited to our method, approaching ideal performance and most efficient use of memory as the average difference between adjacent data approaches the minimum (δ, upon which our choice of $\Delta_{\min}$ hinges). This is not to say that a respectable divergence from a uniform distribution is prohibitive or lacks performance gains. Additional dimensionality has a consequence of a more rapidly growing hash table. In the case of recursive bisection along each dimension it is $O(2^{d \cdot levmx} N)$ where d is dimensionality and $levmx$ the numer of levels of refinement with respect to the coarse grid. However, this is checked by the reduction of $levmx$ and N on multi-dimensional grids to fit into memory. Thus far, our algorithmic studies in Section 3 have not been hindered by hash table size growth. By temporarily using additional memory, hash-based algorithms reap the benefits of performance gains.

The advantage of the hash-based approach is a result of more than just the algorithm; the inherent parallel characteristics are well-suited to the GPU architecture. We have described hashing as a non-comparison method with independent consideration of keys. Each is mapped based exclusively by their own value and a handful of global variables. This not only allows us to reduce the complexity, but also facilitates the parallelization of hash-based methods and requires little communication as the threads execute independently.

We present hashing as a viable, competitive option that could be extended to a wide range of algorithms to approach $O(n)$ performance by capitalizing on the distributed structure of discretized data and trading memory space for reductions in execution time.

3. Hash-Based Algorithms. Meshes are one of the most common types of discretized data found in numerical coding and often require the use of several fundamental functions which can take advantage of hash-based approaches to achieve better performance. Here we explore these hash algorithms in 1-D array base cases to convey the crux of the hash based methodology as well as how it extends to more complex 2-D problems. Implementations and performance are discussed for both the CPU and GPU. Though there are undoubtedly many operations that could benefit from a similar approach, we address some of the most common: sorting, neighbor finding, remapping, and table look-up.

The numerical studies recounted in the next sections are run on a heterogenous architecture test platform composed of 48 AMD Opteron 6168 cores on each node, with some of the nodes having 2 NVIDIA Tesla C2050 GPUs and others with 2 ATI FirePro 7800 GPUs.

3.1. Sorts. The application of the hash-based methodology to the sort is perhaps the simplest and most direct. It proceeds by mapping each key to a bucket in the hash table and performing a reduction to remove empty buckets, yielding a condensed, sorted array. In a 1-D sort problem, the minimum difference δ emerges as a natural choice for $\Delta_{\min}$ in constructing the hash table and while finding this difference and the range of a set of discretized data may cost extra computational time, these characteristics may be known by design or remain fairly constant between multiple invocations. The values are then ordered through mapping to the hash table, a process summarized in Figure 3.1. In practice, we diverge from our illustrated example in writing not the value, but rather the key's index in the original unsorted array to

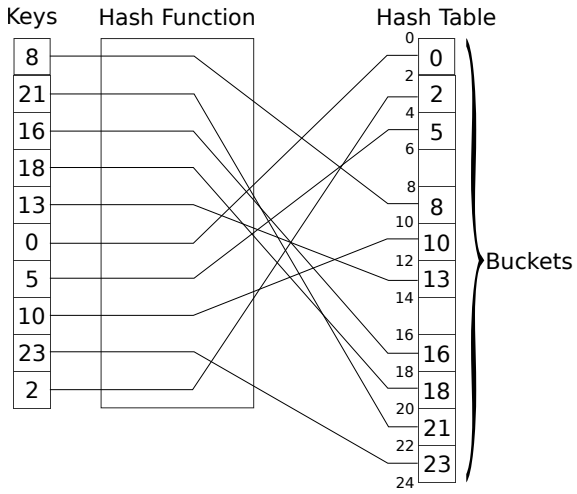

FIGURE 3.1. *In our hash-based sort, a set of ten values (keys) is mapped to a hash table through a hash function. The hash table is constructed using* $\Delta_{\min}$ *equal to 2, a minimum value of 0 and range of 23. The values fall into buckets as discussed in Section 2.3 and unassigned, empty buckets are removed in an additional pass. We show an example with integers and mapping values rather than indices for the sake of simplicity.*

the hash table. The hash table can then be stored as an array of integers rather than storing potentially larger data-types and the index provides an easy reference back to the sorted values or objects. In some applications, the hashed indices may even apply to multiple arrays and be used in ordering any number of them.

The principles used in this perfect hash sort are also found in pre-existing algorithms, especially the bucket and radix sorts. Bucket sorts operate on a similar idea of mapping values directly to buckets in a range, but have been largely limited to use with integers and database applications. The hash-based sort broadens this concept from its traditional roots to the domain of discretized data and in so doing finds a wide base of applications in the numerical field. At the same time, the hash-based sort is a specialization of the general purpose radix sort. Rather than requiring multiple passes to consider and map one or a few digits at a time as in the radix sort, the unique properties of discretized data allow the entire number to be mapped and ordered in a single pass.

The same fundamental principle of mapping based on the underlying structure of discretized data carries over to a 2-D problem. The easiest approach would be to treat each value as a point and sort first on the y-value and then a secondary sort on the x-values within the same y-values. This point-based approach works equally well on AMR and unstructured meshes. Since this is such an obvious extension of the technique used in the 1-D sort, we will examine another approach in the 2-D sort.

Various orderings may be used in the 2-D sort. The most obvious is switching between a y-dominant ordering and an x-dominant order. Other orderings may be based on the i, j indexing with variants using different priorities for coarse cells and fine cells. And there may also be some spatial ordering to support partitioning across processors for distributed, MPI calculations. Indeed, the most likely use of a sort in mesh calculations is to switch between different orderings for different parts of the computation or output for graphics. In the 2-D implementation we used for this demonstration, we use standard lexicographic ordering which has a hash key of

$h = i + j * w$ where (i, j) is the cell index computed at the finest level of the mesh, and w is the width by the number of cells at the finest level of the mesh. Note that this views the world as cells and cell extents rather than points representing cell centers and distance between cell centers. This shift in world view can be accommodated by using a different concept for $\Delta_{\min}$ based on the minimum cell extents in x and y instead of the distance between cell centers. Though this ordering primarily addresses an AMR grid, similar issues arise with unstructured grids

The CPU version of the 1-D hash-based sort algorithm is shown in Listing 1.

Listing 1
Sort Algorithm

```
/* construct hash table and add a bucket for truncation error */
hash_table_size = (int)((max_val - min_val)/mindx + 1);

/* set all elements to -1 to tag empty buckets */
memset(hash_table, -1, hash_table_size*sizeof(int));

/* place index of each element in its respective bucket */
for(i = 0; i < unsorted_array_length; i++) {
    hash_table[(int)((arr[i]-min_val)/mindx)] = i;
}

/* discard empty buckets to create condensed, sorted array */
int count = 0;
for(i = 0; i < hash_table_size; i++) {
    if(hash_table[i] >= 0) {
        sorted[count] = arr[hash_table[i]];
        count++;
    }
}
```

These sorts were tested on generated datasets. In 1-D, a set of values is designed to meet conditions of size, a minimum starting value, and a minimum and maximum difference between adjacent data. Beginning with the provided starting value, a number between the given minimum and maximum difference is successively added to the previous element and the final product is randomized.

For 2-D, a random adaptive mesh of square cells is constructed by starting with a coarse mesh, randomly setting a level of refinement, and then smoothing across the mesh to ensure there is never greater than one level of refinement difference between neighboring cells. The mesh order is then randomized by shuffling the cells.

Parallelized on GPU. Mapping to the hash table is an inherently parallel task requiring no communication between work-items/threads which further capitalizes on the GPU's ability to write directly to global memory from each of these work-items. In fact, the most computationally expensive part of our parallelized hash-based sort is the reduction step to condense the sorted array from the hash table. This requires a prefix scan to determine where in the final array each work-item should write. Compared to the radix sort, we have the advantage of requiring only one reduction instead of reducing after each pass.

The prefix scan used for our test code was extracted from the cell-based adaptive mesh refinement (CLAMR) code[12] that originally stimulated this research. This prefix scan consists of three kernel calls as shown in Figure 3.2. As originally written, the hash sort algorithm using this scan was a factor of two slower than the CUDPP radix sort. The difference in performance was largely due to the prefix scan performance and a highly optimized scan in OpenCL is not yet available in libraries. So some effort has been made to speed up the OpenCL scan being used in the hash sort. Ideas were taken from Sengupta's work on optimizing scans[17] and from the implementation of

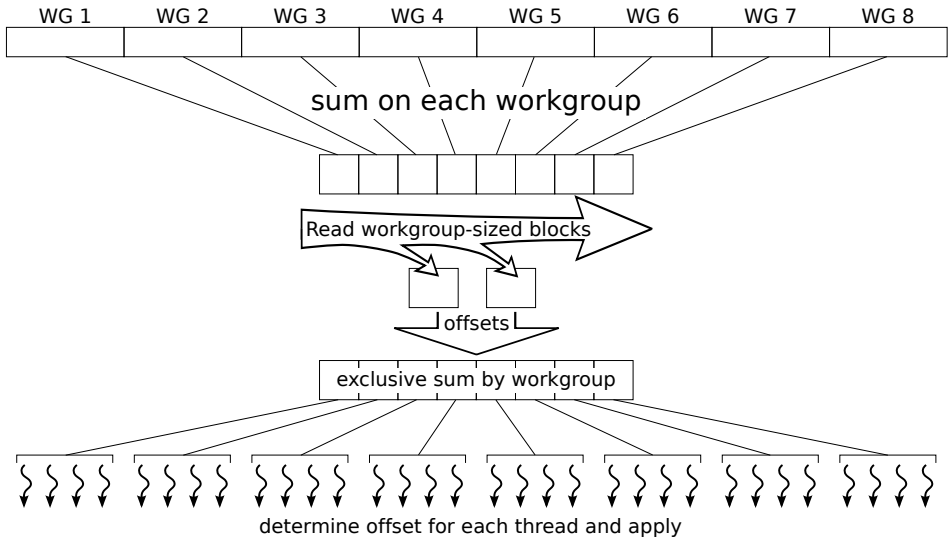

FIGURE 3.2. *Prefix scan synchronized through multiple kernel calls. (1) Performs a global sum on each workgroup, reducing the array by a factor equal to the workgroup size. (2) Performs a scan of workgroup-sized chunk of data, looping until all the data is processed and saving the calculated offsets for each workgroup. (3) Does a workgroup scan to determine additional offset for each thread and applies result in a write.*

some of these ideas into the CLPP scan in OpenCL [1]. These optimizations have all been added to the second kernel resulting in a speed-up of about a factor of three. During the literature search for speeding up the scans, it was found that the reduction sum in the first kernel is very similar to the method in the Dotsenko scan[4]. So the resulting scan is a hybrid of the ideas of these two highly optimized scan routines. With all of this work, the hash sort now is twice as fast as the radix sort and with more optimization of the prefix scan, it could further outperform the radix sort.

Performance. Each of the sorts was run on the same unsorted array. The preset conditions create a data set with fairly evenly spaced values which is preferable, though not ideal, for the hash-based sort. The minimum value was 0, the minimum difference 2.0, and the maximum difference 4.0. The actual differences between values is still a random number between the maximum and minimum differences, but the average difference increases along with the maximum difference.

On the CPU, our hash-based sort performed consistently better than the highly optimized quicksort from the C library. The speed-up is consistently about a factor of four faster as shown in Figure 3.3. Also shown, the GPU hash sort out-performs the CUDPP radix sort [17] by a factor of two to three. The CUDPP radix sort is currently the fastest GPU sort and was run on this same hardware to provide a basis of comparison. It should be remembered that the GPU hash sort is a specialized sort that exploits the properties of the differential discretized data while the radix sort is a general purpose sort. The performance scaling for the NVIDIA and ATI GPUs are a bit erratic and likely could still be improved from the performance here.

In the scaling studies above, we used a maximum difference to minimum difference ratio of two. As the ratio grows, the size of the hash table grows larger also. And the reality is that the size of the hash table has an influence on performance. In theory,

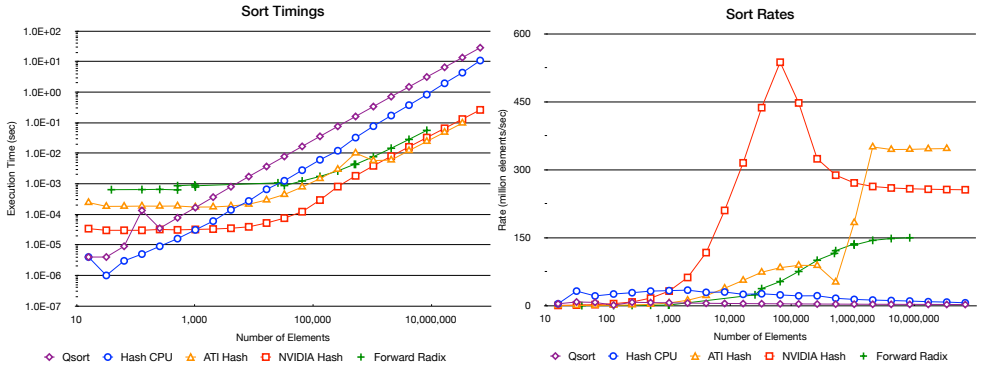

FIGURE 3.3. *Performance graphs of the hash method on the CPU and GPU as compared with quicksort and radix sort. The properties of the test data are favorable to this perfect hash method, with a minimum difference of 2. and a maximum difference of 4.*

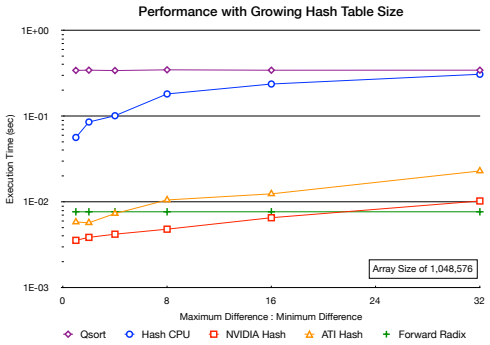

FIGURE 3.4. *The performance of our hash sort is dependent to some degree on the size of the hash table. As the ratio of the maximum difference between data and the minimum difference increases, the size of the hash table grows. Performance will be slower even on the same sized array, but even with a fairly large hash table, up to 32 times as large as the original unsorted array, the hash sort continues to be competitive with other methods.*

the other sorts (quicksort and radix) should not be largely affected by a change in the distribution, staying constant with its variation while execution time for the hash-based method steadily increases. To determine at what point the size of a hash table hinders the performance enough to make hashing comparable to other methods, we conducted a series of runs with increasing maximum differences. The decreasing performance can be seen in Figure 3.4 but it takes a ratio of about 16 before the NVIDIA performance becomes slower than the radix sort. This corresponds to an adaptive mesh refinement level of 16; a high number of refinement levels to use in a calculation. Similarly, the CPU hash at about 32 levels of refinement slows down close to the level of the quicksort.

For the 2-D sort, the quicksort is slower due to the more complex comparison operator and as a result the CPU hash is about 16 times faster throughout the range as is shown in Figure 3.5. The performance of the 2-D GPU hash sorts is still faster than the 1-D CUDPP radix sort though not as fast as the 1-D GPU hash versions.

3.2. Determining Neighbors. When dealing with a computational mesh, it is often necessary for a cell to gather information from neighboring cells, and determin-

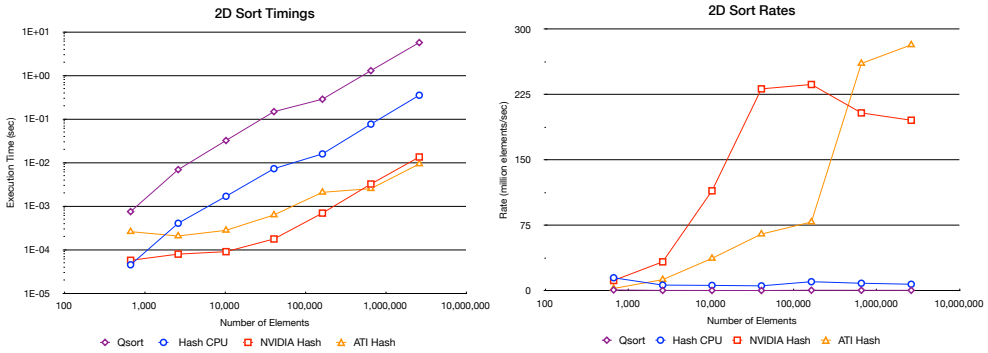

FIGURE 3.5. *Performance graphs of the hash method on the CPU and GPU as compared with quicksort. The maximum level of refinement in the problems is 1.*

ing these neighbors in adaptive and unstructured meshes is nontrivial. The hashing method proceeds in a similar fashion as in the sort, but with some modification. These variants lead us to introduce the ideas of "single write, multiple reads" and "multiple writes, single read" to spatial hashing.

In the case of a 1-D mesh, finding neighbors is a matter of ordering the cells and looking left and right. The hash used is equivalent to that in the sort and only a single bucket is used for each cell (single write). However, we also refrain from performing the scan reduction, working directly from the hash table. Finding the two neighbors becomes a multiple read process to search left and right until a non-empty element is reached.

The 2-D mesh case (and higher dimensions) is more complex and has the added challenge of dealing with the discrepancy in the sizes of some neighboring cells. Hashing again provides a means for quick spatial queries. The concept behind the method essentially comes down to projecting a maximally refined mesh on top of the actual mesh. Instead of writing to just one bucket based on a cell's center coordinates, the 2-D method uses the range of a cell, writing to every refined cell that it contains in the hash table projection. In this sense the 2-D code uses multiple writes, however then a cell no longer needs to make multiple reads as it can simply look immediately at its boundaries in the refined mesh. A cell can directly look up neighbors without concern of disparate refinements of cells. We present the 2-D CPU code in Listing 2.

LISTING 2
2-D CPU Hash Neighbor Code

```
1  /* construct table of powers of two (1,2,4,...) and set the number of rows
       and columns at the finest level */
2  int* levtable = (int*)malloc(levmax+1);
3  for(lev = 0; lev < levmax + 1; lev++) levtable[lev] = pow(2,lev);
4  int jmaxsize = meshsize*levtable[levmax];
5  int imaxsize = meshsize*levtable[levmax];
6
7  /* initialize 2-D hash table of ints to -1 */
8  int** hash_table = (int**)genmatrix(jmaxsize, imaxsize, sizeof(int));
9  memset(hash_table, -1, jmaxsize*imaxsize*sizeof(int));
10
11 /* map cells to hash table; references 1-D arrays level with level of
       refinement, i and j which are the row and column of the cell in the mesh
       at that cell's refinement level */
12 for(ic = 0; ic < ncells; ic++) {
13     int lev = level[ic]; //refinement level of current cell
14     int levmult = levtable[levmax-lev]; //converts to index on finest level
15     /* maximally refined cells write to single bucket */
```

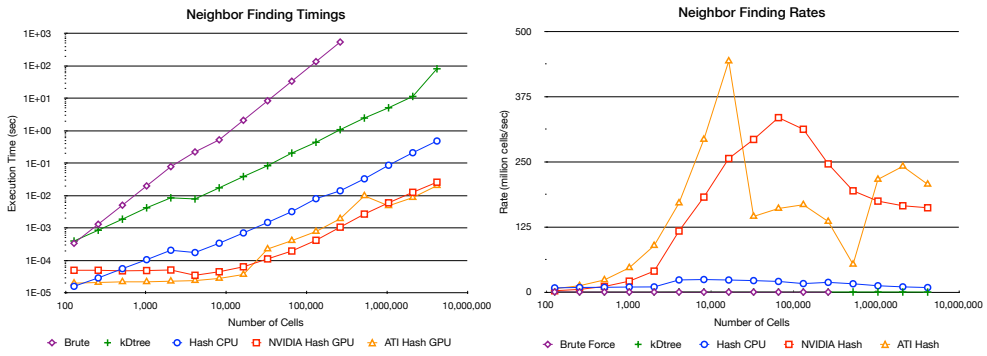

FIGURE 3.6. *Performance graphs of the hash methods on the CPU and GPU for neighbor finding in a 1-dimensional mesh and a maximum difference two times larger than the minimum (one level of refinement).*

```
16      if(lev == levmax) hash_table[j[ic]][i[ic]] = ic;
17      /* for coarser cells, write to multiple buckets */
18      else {
19          for(jj = j[ic]*levmult; jj < (j[ic]+1)*levmult; j++) {
20              for(ii = i[ic]*levmult; ii < (i[ic]+1)*levmult; i++)
21                  hash_table[jj][ii];
22          }
23      }
24  }
25
26  /* retrieve neighbors by reading on finest level of mesh */
27  for(ic = 0; ic < ncells; ic++) {
28      int levmult = levtable[levmax-level[ic]]; //converts to finest level
29      int ii = i[ic] * levmult; //indices of lower left bucket of cell in hash
30      int jj = j[ic] * levmult;
31      neigh2d[ic].left   = hash_table[jj][MAX(ii - 1, 0)];
32      neigh2d[ic].right  = hash_table[jj][MIN(ii+levmult, imax-1)];
33      neigh2d[ic].bottom = hash_table[MAX(jj - 1, 0)][ii];
34      neigh2d[ic].top    = hash_table[MIN(jj+levmult, jmax-1)][ii];
35  }
```

Parallelized on GPU. Just as in the sort, the similar structure of the hash-based neighbor algorithm lends itself to easy parallelization. The first kernel call constructs and fills the hash table; each cell independently writing to the one or several buckets in 1-D and 2-D respectively. The second kernel reads from the hash table as required. Working directly from a hash table, there is no need for a scan reduction call and consequently, the neighbor hash algorithm is extremely parallel.

Performance. There's an interesting metric to consider with the hashing in terms of speed-up. First we can consider how the algorithm itself speeds up the application, which is easily seen by the CPU speed-up. Second, given the inherently data-independent parallel nature of the hashing algorithm, there is the speed-up achieved due to parallelism. In the hash sort we saw a CPU speed-up factor over the quicksort of about 4x whereas the GPU speed-up over quicksort was a factor closer to 100x, showing that the primary boost came from the easy parallelization.

In the case of the neighbor lookup, as shown in Figure 3.6 in both our test code and a large scale test, we see huge speed-ups even in the CPU code, showing that the fundamental architecture of the algorithm is a vast improvement over the standard k-D tree. Also shown is the brute force, an $O(n^2)$ algorithm where all other cells are searched to find the neighbor for each cell. The 2-D performance as shown in

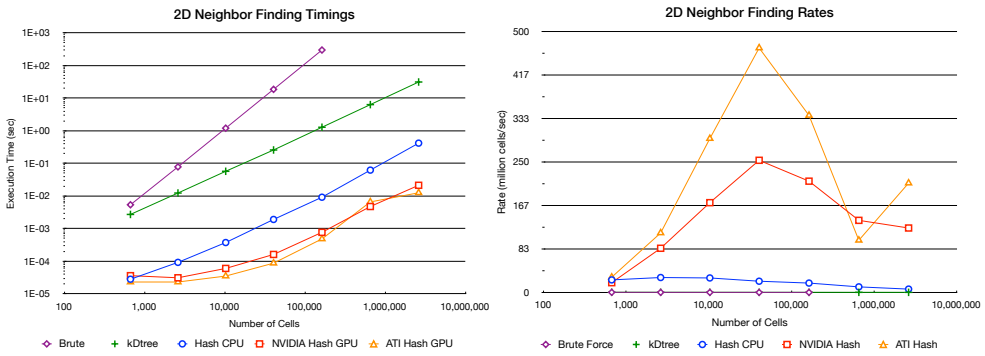

FIGURE 3.7. *Performance of hash-based neighbor finding in a 2-D mesh on the CPU and GPU. There is a maximum of one level of refinement.*

Figure 3.7 has similar speed-ups, but the details of the performance scaling with array size have some interesting and different inflections for both the NVIDIA and ATI GPUs.

Increasing the levels of refinement slows down the neighbor hash implementations, but the CPU hash is still far faster than the k-D tree even at 32 levels of refinement.

This algorithm has been implemented in CLAMR [12], a large scale adaptive mesh refinement code. In CLAMR there was a 350× speed-up in the CPU code, which when combined with the GPU's parallel boost gives a speed-up of over $20,000\times$. This is much higher than seen here, but some additional lines of setup code could easily increase the speed-up. The CLAMR code also has an MPI/OpenCL version of the neighbor algorithm that sets up the MPI communication pattern and the ghost cells. Studies of the performance have just begun, but one of the most notable aspects of this hybrid parallelization model is that the simple neighbor hash look-up code went from 50 lines to 800 lines. This is due to having to pull the data off of the GPU and setup the communication pattern and then push the ghost data down onto the GPU mesh. This will be the most complicated part of the hybrid programming, so it may well be reasonable complexity of code if it is isolated from the main physics algorithm.

3.3. Remaps. It is often desirable to change mesh representations, requiring a remap of the state variables or values stored in the cells of the original mesh into the cells of the new mesh. This is generally achieved by finding the cells from the original mesh that are contained in each of the new cells and calculating the fraction of those cells that are actually included. In most methods it is the first step that dominates performance. Here we explore two possible methods stemming from hashing.

The first hash algorithm brings yet another slight variation into play: a "multiple write, multiple read". Just as in the 2-D neighbor hash, each cell writes to all the buckets contained in its range (multiple writes). Here we further restrict the structure of the meshes to assume that the cell boundaries in both the original mesh coincide with the edges of the buckets in the hash table. In other words, all cells must be an integer multiple of Δ_{min} as any bucket associated with a cell is assumed to be wholly contained by that cell. Once constructed, the hash table acts as an easy reference that the cells from the new mesh, in no particular order, can query to read all the buckets in their range (multiple reads). The retrieval is illustrated in Figure 3.8. Again, using indices in the hash table provides easy access to the cell and relevant information of value and size. The remap is performed on two arrays of cells with the left and right

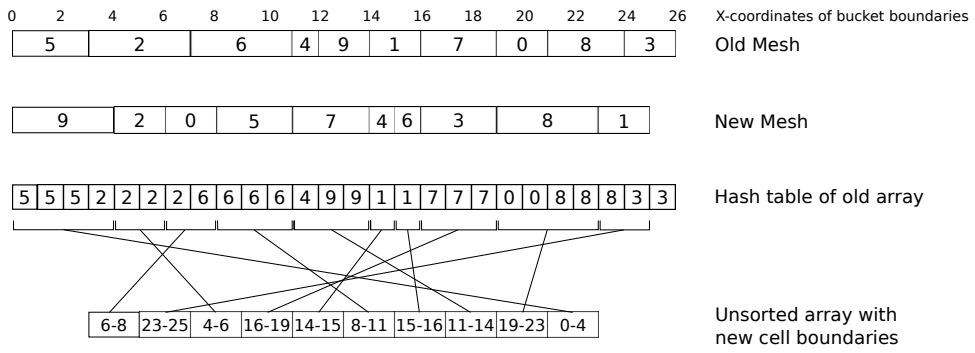

FIGURE 3.8. *A remap transfers state variables from one mesh representation to another. In order to do this, the fractions of the old cells contained in each of the new cells must be determined. Here we show a hash-based approach in which each cell from the old mesh writes its index into each of the refined cells it contains in a hash table. Each of the new cells can directly look up which cells it contains by referencing that hash table.*

boundary information. The 1-D remap implementation is shown in Listing 3.

LISTING 3
1-D Remap Algorithm

```
1  memset(hash_table, -1, hash_table_size*sizeof(int));
2
3  /* Create a hash table for the first (old) array */
4  for(a = 0; a < asize; a++) {
5     start = (int)((arr_old[a].low+min_val)/mindx);
6     end = (int)((arr_old[a].high+min_val)/mindx);
7     while( start < end ) {
8        hash_table[start] = a;
9        start++;
10    }
11 }
12
13 /* Remap into new array */
14 for(b = 0; b < bsize; b++) {
15    remap[b] = 0;
16    if( (start = (arr_new[b].low - min_val)/mindx) < hash_table_size) {
17       end = MIN(hash_table_size, (arr_new[b].high - min_val)/mindx);
18       for(i = start; i < end; i++) {
19          if(hash_table[i] >= 0) {
20             remap[b] += original_values[hash_table[i]] * 1.0/(arr_old[
                   hash_table[i]].high-arr_old[hash_table[i]].low);
21          }
22       }
23    }
24 }
```

While the 2-D remap is effectively identical in structure to this first 1-D remap, the use of lexicographic ordering in 2-D equates to non-contiguous writes and reads in the global hash table. Other ordering methods could improve cache access/performance.

We present a second possible remap method that allows more flexibility in the structure of the 1-D mesh by beginning with a sort of both meshes. This approach, in a sense, overlays the cell boundaries of the two meshes, transferring each of the subsections delineated by the combined boundaries. The 1-D setting simplifies the problem considerably, and the algorithm in Listing 4 demonstrates the process of iterating through the dual meshes in the form of two arrays of cell boundaries.

LISTING 4
Alternate Remap Algorithm

```
for(a = 1; a < asize; a++) {
    while(arr_b[b] <= arr_a[a] && b < bsize) {
        range = arr_b[b] - MAX(arr_a[a-1], arr_b[b-1]);
        fraction = range/(arr_a[a] - arr_a[a-1]);
        remap[b-1] += fraction * 1;
        b++;
    }
    range = arr_a[a] - MAX(arr_a[a-1], arr_b[b-1]);
    fraction = range/(arr_a[a]-arr_a[a-1]);
    remap[b-1] += fraction * original_values[__];
}
```

Parallelized on GPU. Although the alternate method is much more serial in nature, the trend of intrinsic parallelism continues in the first of the remap methods. We can take full advantage of the parallelism of the hash table construction and work directly from the table to eliminate any need for a reduction. As compared with the other algorithms, we do encounter a problem with load balancing. In both the write and read, the iterations required are variable given a cell's level of refinement. However, the desired relative similarity of size dictated by the structure of discretized data checks this somewhat and we achieve a significant boost from parallelization even without optimizations to equalize work loads.

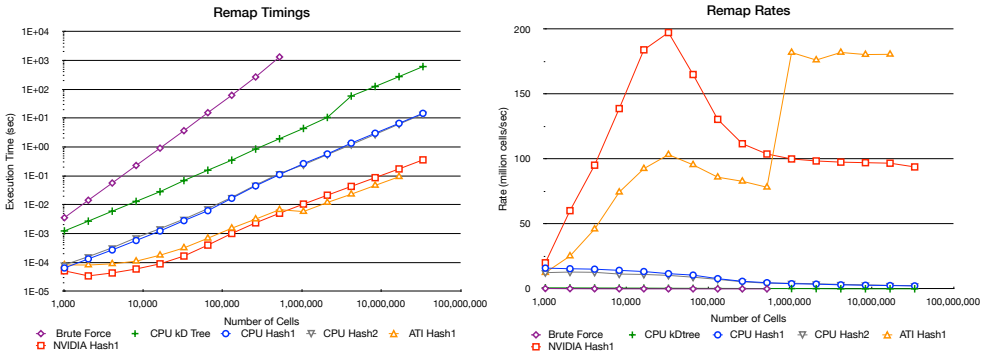

FIGURE 3.9. *Remap performance results on problems with one level of refinement. The number of cells applies to both the original and remap meshes, although the area they span may vary.*

Performance. The remap performance also shows a big jump in performance with the CPU hash version of almost 20x over the k-D tree. The multiple write, multiple read in the algorithm reduces the efficiency of the hash implementation from that in the neighbor version. The algorithm is very parallel and shows good speed-up when moved to the GPU for a total of 500–800x faster than the k-D tree algorithm on the CPU. The 2-D performance shown in Figure 3.10 has very similar performance showing that the implementations scale well to multi-dimensional calculations.

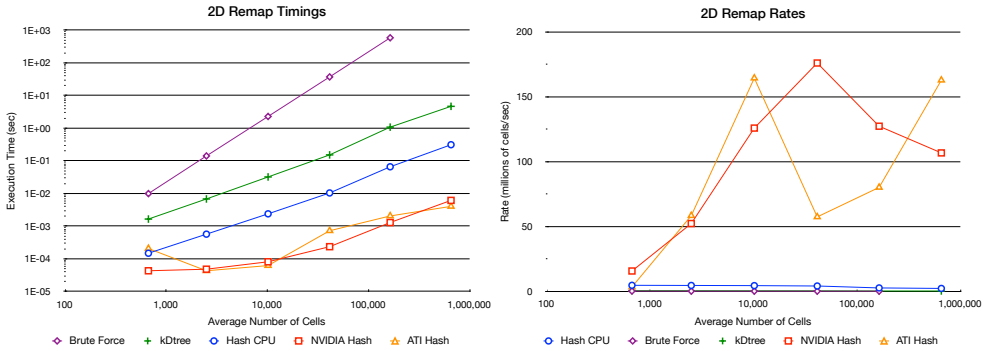

FIGURE 3.10. *2-D remap performance results with one level of refinement. The number of cells applies to both the original and remap meshes, although the area they span may vary.*

3.4. Table Look-up. Table look-ups are common operations in production-level numerical codes, particularly for Equation-of-State (EOS) data or other material properties. It is unique in the algorithms presented here in that it operates on physics data rather than on the mesh. Still, there are some similarities in that the material property look-up tables are discretized to regular-sized finite dimensions to keep the material property errors within reasonable bounds.

The data used for this study is the Oxygen data from the Sesame table, a LANL standard data table used in many physics calculations. The data table is composed of 23 values of density and 51 temperature values. Interpolating between these data values, the calculation returns the pressure for a given density and temperature. The axis values of density and temperature are on fairly uniform intervals, but not regular enough for easy use in this study. So the density and temperature intervals were replaced with regular intervals across the range of data. It should be noted at this point that some modification of the table data will in general be desirable to minimize the if() blocks for the GPU code. The suggestion would be to make the data intervals regular across the middle of the data range and much wider at the ends to avoid going off the end of the table. Some of this could be done automatically through resampling the data into a larger data set based on the minimum interval similarly to the methods used throughout this study.

The sampling of the data set is done through a random number biased to generate a normal distribution so that most of the interpolations are done in the center of the data table. This is to mimic the proper use of a data table where the data at the ends is for extremes that do not occur with great frequency in the calculation. The interpolation method used here is a bi-linear interpolation. This is one of the options also in the Sesame data look-ups, but the more commonly used method is a bi-rational interpolation using 12 data values. The bi-rational interpolation would have more floating-point operations and would potentially result in greater speed-ups, but it also requires more programming and special cases at the boundaries.

The methods used for the standard for comparison are a brute force where the axis data is searched linearly from the starting point until it finds the proper interval for the interpolation and a standard bi-section method where the mid-point of the data is tested and the search picks one side or the other to continue. This method is well described in the popular Numerical Recipes [14] as one of the best choices for ordered data table look-ups. The new methods demonstrated here are a hash-based look-up on the CPU and the GPU.

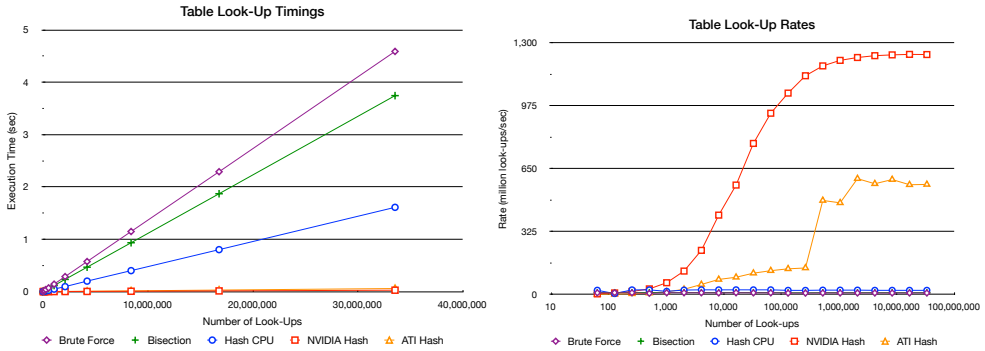

FIGURE 3.11. *Table look-up performance for a series of multiple look-ups.*

The method used on the GPU is for each work-group to make a copy of the data to the local memory space. Then the two input arrays are read from the GPU global memory in work-group sized chunks and the interpolation is done. At the end the the output array is written to GPU global memory. A more complex interpolation method would generate even more speed-up. If blocks for different intervals or end of table handling would reduce the speed-up. The table size could be larger than the 1173 values tested in this problem. The implementation is shown in Listing 5.

LISTING 5
Table Look-ups using a Hash-based Algorithm

```
/* computes a constant increment for each axis data look-up */
double density_increment = (density_axis[50]-density_axis[0])/50.0;
double temp_increment = (temp_axis[22]-temp_axis[0])/22.0;

for (int i = 0; i < isize; i++){

   /* determine the interval for interpolation and the fraction in the interval*/
   int temp_slot = (temp_array[i]-temp_axis[0])/temp_increment;
   int density_slot = (density_array[i]-density_axis[0])/density_increment;
   double xfrac = (density_array[i]-density_axis[density_slot])/
      (density_axis[density_slot+1]-density_axis[density_slot]);
   double yfrac = (temp_array[i]-temp_axis[temp_slot])/
      (temp_axis[temp_slot+1]-temp_axis[temp_slot]);

   /* bi-linear interpolation */
   value_array[gid] =       xfrac *      yfrac *dataval(islot+1 + (jslot+1) * xstride)
                    + (1.0-xfrac)*      yfrac *dataval(islot   + (jslot+1) * xstride)
                    +       xfrac *(1.0-yfrac)*dataval(islot+1 +  jslot    * xstride)
                    + (1.0-xfrac)*(1.0-yfrac)*dataval(islot    +  jslot    * xstride);
}
```

Performance. Among the standard techniques, the bi-section method did not show any appreciable speed-up over the brute force method because of the small axis data sizes and the boost provided by the cache for the linear search. The linear search should be $\frac{1}{2}n$ operations since the search should be half the data size on average and the bi-section should be $\log n$ in operation count. For the smaller axis the operation count is 13 versus 5 and the larger axis is 26 versus 6.

The hash operation should be constant order of one operation for regularly spaced data. On the CPU the performance of the hash-based look-up is about twice as fast. Given that the majority of the flops are in the bi-linear interpolation, the speed-up from the interval search is substantial. Moving the calculation to the GPU, the speed-up compared to the standard methods is over 100 times faster as is shown in Figure 3.11.

4. Conclusion. In this paper, we have presented a formal definition of differential discretized data, developed new, innovative algorithms for spatial mesh oper-

TABLE 4.1
Algorithm Speed-ups for Array sizes of about 2 million

	CPU Hash	NVIDIA	ATI	NVIDIA	ATI
Relative to	Reference CPU	CPU Hash		Reference CPU	
Sort	4.16	21.5	28.6	89.3	118.9
Sort2D	16.2	26.2	37.8	424.1	611.5
Neighbor	54.4	16.6	24.2	903.5	1316.0
Neighbor2D	75.5	19.1	19.1	1444.0	1445.3
Remap	18.4	26.9	48.1	495.2	885.8
Remap2D	13.6	42.2	61.6	574.0	837.8
Table	2.44	55.7	27.2	136.2	66.5

TABLE 4.2
Hash Sort versus fastest general purpose GPU sort

	NVIDIA Hash	ATI Hash
Relative to	CUDPP Radix	
Sort	2.68	3.57

ations, and ported the algorithms to the relatively new parallel environment of the GPGPU computing accelerator hardware. The formal mathematical definition has two purposes:

1. To demonstrate that for every spatial mesh operation there is an efficient hash-based algorithm. By efficient we mean the algorithm is $O(n)$ or better (but not necessarily the fastest).
2. Optimizing the mesh through the process formalized by differential discretized data also optimizes the hash operation. By optimization we mean the most benefit is gained given the work done.

Both of these results are significant. Not only do we have interesting algorithms that lend themselves to parallelization, but we also have a whole class of potential uses that occur with great frequency and even greater impact. We can use them to speed-up many of the calculations that compose the top workload of high performance computing.

Now, examining the speed-ups of these algorithms in Table 4.1, the potential impact becomes clearer. The speed-ups range anywhere from significant to outstanding. This is due to the multiplicative effect of

- replacing an $O(n \log n)$ algorithm with an $O(n)$ algorithm
- harnessing the massively parallel compute capability of the GPU.

We could also gain some speed-up through MPI parallelism, though we have not demonstrated that here.

It is important to note that these algorithms are specialized algorithms that exploit the characteristics of the underlying data. To get a better sense of this, we compare the hash sort performance to the best general purpose sort in Table 4.2. The hash sort is two to three times faster than the sort found in the CUDPP and Thrust CUDA libraries.

Another important aspect of these results addresses the performance portability of OpenCL. These algorithm kernels yield reasonably similar results without a lot of work optimizing for each architecture.

In conclusion, we consider this work a starting point for further studies. Extending the algorithms to unstructured mesh data is sorely needed. The code presented here has not yet been highly optimized and more performance can be gained from additional work. Also needed are extensions of these algorithms to a heterogenous MPI layer; some coding work has been done in that direction, but it is far from complete. Lastly are the application studies to demonstrate the impact of adopting these techniques in important scientific applications.

Acknowledgments. Thanks go for the use of the Darwin compute cluster in LANL CCS-7 and to Marcus Daniels for debugging beyond the call of caffeine and his declaration of truth that solves the most obscure of bugs.

REFERENCES

[1] CLPP – OpenCL Parallel Primitives Library. Software Library: http://code.google.com/p/clpp/.

[2] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[3] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[4] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 205–213, New York, NY, USA, 2008. ACM.

[5] William F. Gilreath. Hash sort: A linear time complexity multiple-dimensional sort algorithm. *CoRR*, cs.DS/0408040, 2004.

[6] Michael Griebel and Gerhard Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Computing*, 25(7):827 – 843, 1999.

[7] Mark Harris, Shubhabrata Sengupta, and John D. Owens. *Parallel Prefix Sum (Scan) with CUDA*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[8] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986.

[9] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[10] Hollerith. Radix sort. Attribution given in Wikipedia, 1887.

[11] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[12] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey. Cell-based adaptive mesh refinement implemented with general purpose graphics processing units. *SIAM Journal of Scientific Computing*, 2012 (in review).

[13] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879 –899, May 2008.

[14] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, New York, 3rd edition, 2007.

[15] R. Ramey. Postman's sort. *C Users Journal*, August 1992.

[16] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D. Owens. *Efficient Parallel Scan Algorithms for Many-core GPUs*, chapter 19, pages 413–442. Chapman & Hall/CRC Computational Science. Taylor & Francis, January 2011.

[18] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[19] J. von Neumann. Merge sort. Attribution due to Knuth, 1945.

[20] J.W.J. Williams. Algorithm 232 - Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[21] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):669 –681, May 2011.