HETEROGENEOUS CONSTRUCTION

OF SPATIAL DATA STRUCTURES

by

ROBERT O BUTTS

B.S., Abilene Christian University, 2009

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Computer Science Program

2015

UMI Number: 1588178

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 1588178
Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Au
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower
Parkway
P.O. Box 1346

This thesis for the Master of Science degree by

Robert O Butts

has been approved for the

Computer Science Program

by

Gita Alaghband, Chair

Tom Altman

Ilkyeun Ra

April 24, 2015

Butts, Robert O (M.S., Computer Science)

Heterogeneous Construction of Spatial Data Structures

Thesis directed by Professor Gita Alaghband.

ABSTRACT

Linear spatial trees are typically constructed in two discrete, consecutive stages: calculating location codes, and sorting the spatial data according to the codes. Additionally, a GPU R-tree construction algorithm exists which likewise consists of sorting the spatial data and calculating nodes' bounding boxes. Current GPUs are approximately three orders of magnitude faster than CPUs for perfectly vectorizable problems. However, the best known GPU sorting algorithms only achieve 10–20 times speedup over sequential CPU algorithms. Both calculating location codes and bounding boxes are perfectly vectorizable problems. We thus investigate the construction of linear quadtrees, R-trees, and linear *k*-d trees using the GPU for location code and bounding box calculation, and parallel CPU algorithms for sorting. In this endeavor, we show how existing GPU linear quadtree and R-tree construction algorithms may be modified to be heterogeneous, and we develop a novel linear *k*-d tree construction algorithm which uses an existing parallel CPU quicksort partition algorithm. We implement these heterogeneous construction algorithms, and we show that heterogeneous construction of spatial data structures can approach the speeds of homogeneous GPU algorithms, while freeing the GPU to be used for better vectorizable problems.

The form and content of this abstract are approved. I recommend its publication.

Approved: Gita Alaghband

ACKNOWLEDGEMENTS

I would like to thank Dr. Alaghband for her support and encouragement with this thesis in particular and my graduate career and study of parallelism in general.

I would also like to thank Milsoft Utility Solutions, and especially Stan McHann and Adam Turner, for their support in furthering my education and improving my skills as a software engineer.

TABLE OF CONTENTS

CHAPTER

I. INTRODUCTION 1

II. BACKGROUND 4

Quadtree 4

R-tree 6

k-d tree 8

Linear Tree 10

III. ALGORITHMS 12

Linear Quadtree 13

Node Creation 13

Parallel Mergesort 17

R-tree 19

Linear *k*-d tree 22

Parallel Quicksort Partition 27

Vector Morton Code Calculation 32

Parallel Linear Fixed-Length Tree 34

Pipelining 37

IV. RESULTS 38

Methodology 38

Linear Quadtree 39

R-tree 41

Linear *k*-d tree 43

Pipelining 45

Summary 49

V. FUTURE WORK 51

GPU Optimizations 51

k-d tree Homogeneous Granularity 51

3-Dimensional Implementations 52

Investigation of *k*-d tree Splitting Algorithms 52

Investigation of the Effect of Memory Bandwidth on Parallel Sorting 53

VI. CONCLUSION 54

REFERENCES 56

CHAPTER I

INTRODUCTION

Traditionally, spatial tree construction algorithms have been homogeneous, being implemented entirely on the SIMD (single instruction, multiple data) GPU co-processor, or, less commonly, entirely on the MIMD (multiple instruction, multiple data) CPU.

For highly vectorizable problems, modern GPUs typically execute up to three orders of magnitude (1000×) faster than sequential execution on the CPU [Garland et al., 2008] [Ryoo et al., 2008], and well-vectorizable problems typically execute around two orders of magnitude (100×) faster [Owens et al., 2008]. However, poorly vectorizable problems may execute as slow as around one order of magnitude faster (10×) than sequential CPU execution [Lee et al., 2010]. Many algorithms which are poorly vectorizable still parallelize well for MIMD architectures. Hence, we posit that an algorithm which does not lend itself to vectorization but is well-parallelizable may execute on a manycore MIMD architecture comparably to a GPU architecture.

We therefore propose that existing homogeneous algorithms ought to be subdivided, poorly vectorizable functions identified, and the algorithm implemented heterogeneously, wherein well-vectorizable functions are executed on the SIMD GPU and poorly-vectorizable functions are executed on the MIMD CPU. We hypothesize that such a heterogeneous algorithm will execute approximately as fast or faster on a GPU-enabled manycore architecture, and with greater efficiency, than a traditional homogeneous algorithm.

To this end, we have chosen to analyze construction algorithms for spatial data structures. GPU algorithms for spatial trees are typically implemented in two phases. Tree nodes are constructed, and the spatial elements are sorted (not necessarily in that order). The node construction phase is well vectorizable, typically consisting entirely of arithmetic operations and each node being independent. Sorting is better suited for MIMD parallelization. GPU sort algorithms typically execute around 10–20× faster than serial algorithms, or around one order of magnitude [Satish et al., 2009a]. Contrariwise, sorting is a well-parallelizable problem, with MIMD algorithms theoretically seeing linear speedup with the number of cores available, primarily constrained by memory bandwidth [Cormen et al., 2009, p. 803] [McCool et al., 2012, p. 304] [Satish et al., 2010]. Hence, we hypothesize that a heterogeneous construction algorithm which uses the SIMD GPU to construct nodes, and the MIMD CPU to sort will perform comparably to a homogeneous GPU algorithm, with greater efficiency.

To test our hypothesis, we will develop heterogeneous algorithms for constructing R-trees, linear quadtrees, and linear *k*-d trees. We will adapt existing GPU algorithms for R-trees and linear quadtrees. We will also develop a novel heterogeneous linear *k*-d tree construction algorithm.

All the spatial trees we construct are over 2-dimensional space and contain points, for simplicity, without loss of generality. Our algorithms may be easily adapted for polygons in 3-dimensional space, and our general technique should be applicable to any existing GPU tree construction algorithm which involves discrete sorting or other poorly-vectorizable functions.

In Chapter II we review the spatial data structures which we investigate: quadtrees, R-trees, and *k*-d trees. We provide an overview of their design, describe how they each

subdivide space, and provide figures which illustrate the structures graphically.

In Chapter III we present our heterogeneous construction algorithms for quadtrees, R-trees, and *k*-d trees. We present the homogeneous GPU algorithms which we adapt, and we present how we have adapted them for heterogeneous construction. We also present our novel heterogeneous *k*-d tree construction algorithm, and the parallel quicksort around which our algorithm is based. We also present an overview of parallel pipelining, and how it may be used with heterogeneous algorithms when constructing multiple spatial trees.

In Chapter IV we present our results implementing and executing our heterogeneous algorithms.

In Chapter V we suggest future work which might be done to extend our research.

In Chapter VI we conclude by summarizing our algorithms and their results regarding the heterogeneous construction of spatial data structures.

CHAPTER II

BACKGROUND

There exist numerous spatial tree structures, with various advantages and disadvantages. We have selected three of the most widely used for our heterogeneous research: quadtrees, R-trees, and *k*-d trees.

Our motivations for this study were to ascertain whether heterogeneous physical architectures might be more efficiently used to construct spatial data structures, specifically spatial trees, with less time, cost, or power.

Here we provide an overview of quadtrees, R-trees, and *k*-d trees. We also overview how spatial trees may be constructed as linear trees, with various advantages, of which we are primiarly concerned with improved vectorizability. We briefly discuss their structure, how each might be constructed, and provide figures visualizing their structures graphically. We also briefly discuss applications of each tree.

Quadtree

Quadtrees [Samet, 1984] [Finkel and Bentley, 1974] are arguably the simplest spatial data structure. Quadtrees recursively subdivide space into quadrants. Each node in the tree represents a quadrant of its parent, and contains four child nodes, each representing a quadrant of its own space. Each node has a fixed capacity of spatial elements. Typically, but not necessarily, all nodes in the tree share the same constant capacity, to save space. When a node reaches its capacity, it splits into four children into which further insertions

will be recursively placed. Depending on the tree, upon splitting, existing elements may either be retained or inserted into the newly created children. If elements are inserted into children upon splitting, only leaf nodes will contain elements.

As with most tree structures, the quadtree can be searched in logarithmic time. To search, the quadtree is traversed, and for each node, it is determined which children are contained within the search space, and those children are in turn recursively searched. Thereby only nodes of the tree which contain the search space are inspected.

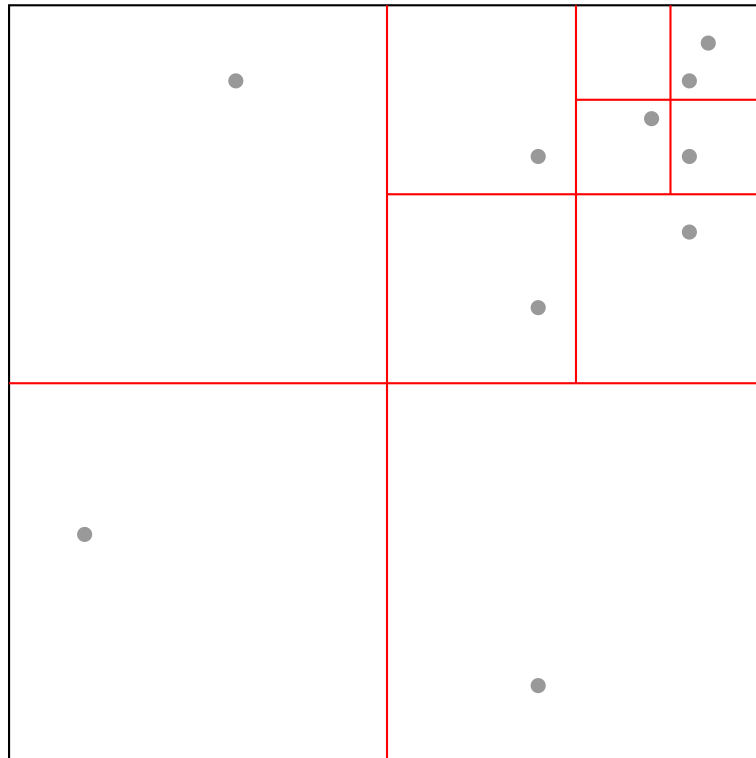

Figure 1: quadtree visualization

Several quadtree variations exist for storing points, lines, and polygons. Additionally, quadtrees may also be used to index images, whereby each pixel is a leaf node of the tree. Image quadtrees need not store the actual pixel information, but only the spatial location of the pixel. The quadtree paradigm also applies to higher dimensions. Quadtrees in

3-dimensional space are called octrees.

Figure 1 presents a visualization of a quadtree which splits when any node contains more than three spatial elements.

Quadtrees and octrees are often used in applications where a space must be searched, and where the simplicity of implementation outweighs the speed and other advantages of more complex trees.

For our research, we will be using 2-dimensional (linear) quadtrees which contain points, without loss of generality.

R-tree

R-Trees [Guttman, 1984] recursively subdivide an n-dimensional space into multiple n-dimensional rectangles. R-trees are typically constructed by enclosing elements in their minimum bounding rectangle, with each parent node representing the bounding rectangle of its children. The spatial tree may then be searched in logarithmic time like any other tree.

The R-tree structure does not require a specific rectangles selection algorithm. Various algorithms for selecting rectangles exist [Guttman, 1984] [Greene, 1989]. Spatial elements are typically enclosed in a minimum bounding rectangle, and it is generally desirable to enclose nearest-neighbors in their minimum bounding rectangles, in order to minimize the search space.

The bounding rectangles of an R-tree are usually axis-aligned. Axis aligned bounding boxes result in an increased search space over bounding boxes aligned to their contents, at the benefit of much less expensive searches. Searching must compare a set of points,

rather than checking for line intersection.

It may also be noted that bounding rectangles can and typically do overlap. It is a goal of R-tree insertion and construction algorithms to minimize overlap, in order to minimize the search space.

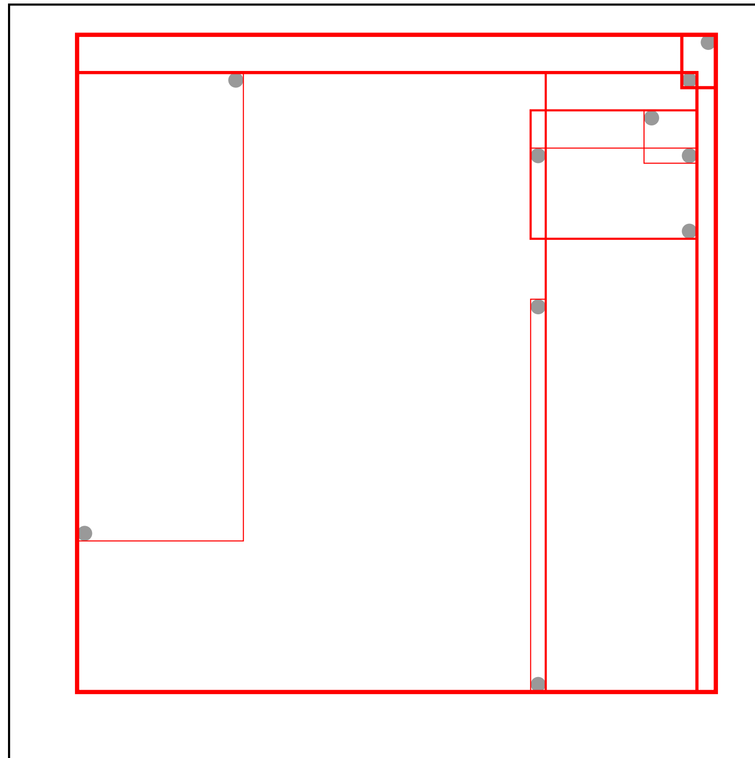

Figure 2: R-tree visualization

As with quadtrees, the nodes of an R-tree also have a fixed capacity. Each node of the R-tree has a fixed number of children, and each node stores the minimum bounding box of all its children. Hence, to search an R-tree, nodes are recursively traversed. For each node, if that node's bounding box contains the search space, the node's children are searched. Since rectangles may overlap, it is possible that multiple children will require searching.

Figure 2 presents a visualization of an R-tree with an arbitrary node selection algo-

rithm, in which the bounding boxes of each child node are drawn progressively thinner than their parents.

R-trees are often used for disk storage by spatial databases. Since each tree node may contain a set number of spatial elements, and a minimum fill may be guaranteed for nodes, the structure is suitable for paging data, similar to a B-tree [Kriegel et al., 2008].

For our research, we will be constructing an R-tree using axis-aligned bounding boxes. Due to the construction algorithm we use, [Luo et al., 2012] our bounding rectangles will be automatically constructed per a sort, and hence the rectangle selection does not apply. We will be constructing a 2-dimensional R-tree which contains points, without loss of generality.

***k*-d tree**

The *k*-d tree [Bentley, 1975] structure recursively subdivides space along a splitting hyperplane (for 2D space, a line; for 3D space, a plane). The “k” stands for “k-dimensional,” as the tree may be constructed for any number of dimensions.

The data structure does not specify how the dimension of the splitting hyperplane is to be chosen. It is common to alternate dimensions. That is, for a 3-dimensional tree, splitting along the x-axis, then the y-axis, then the z-axis. However, it is conceivable that alternate dimension splitting may be optimal for certain data sets or search patterns. For example, if it is known that most searches are over thin vertical subspaces, it may be optimal to always split along the y-axis.

The data structure also does not require a specific algorithm for selecting the splitting location, and thus, many splitting algorithms exist. The middle of the splitting node

can be trivially chosen, in which case the tree will be the binary equivalent of a quadtree. Alternatively, if the median element value is chosen, the tree will be balanced. A balanced *k*-d tree offers the same benefits of other balanced trees, such as guaranteed logarithmic search time. Since the dimensions of the splitting hyperplane are not necessarily the same, algorithms for balancing *k*-d trees are more complex, since the tree cannot simply be rotated due to its multi-dimensional sorted nature.

For 3-dimensional *k*-d trees containing polygons, it is often desirable to minimize or avoid splitting across polygons, which requires the split polygon either be divided or included in both child nodes of the tree. More complex splitting algorithms exist, such as the surface-area heuristic [MacDonald and Booth, 1990] [Feng, 2013] [Havran and Bittner, 2002], which are outside the scope of this paper.

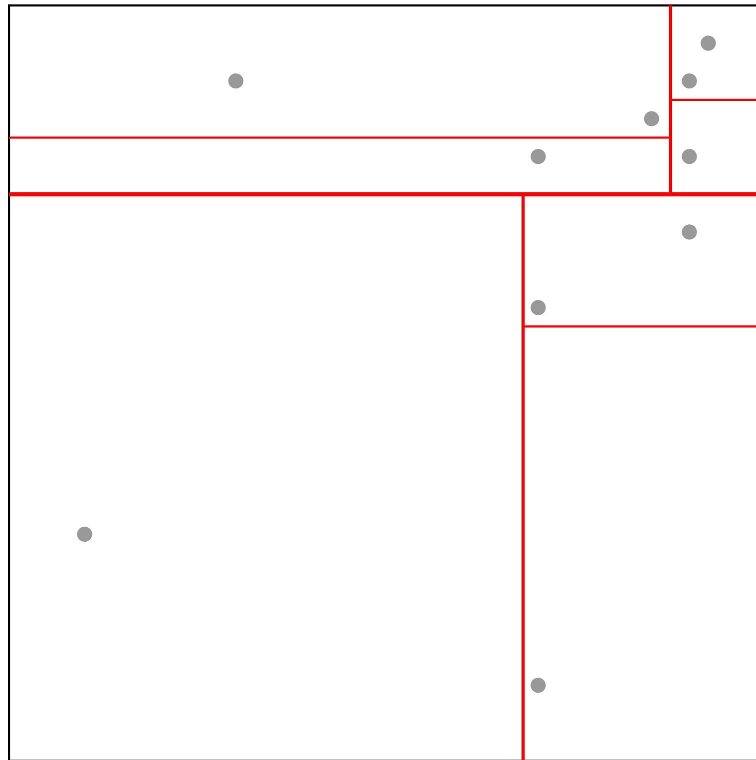

Figure 3: *k*-d tree visualization

Figure 3 presents a visualization of a *k*-d tree with a splitting algorithm which is approximately the average of the spatial elements and which alternates dimensions, in which the splitting hyperplanes (lines) of each child node are drawn progressively thinner than their parents.

The *k*-d tree structure is often used for computer graphics applications such as ray tracing and nearest-neighbor, due to its versatility. A *k*-d tree may be constructed very quickly with a trivial splitting algorithm, or with a variety of benefits and costs via more complex splitting algorithms [Feng, 2013] [Otair, 2013].

For our construction algorithm, we will be constructing a (linear) *k*-d tree which alternates dimensions and which uses a simple heuristic median splitting function. While the splitting function is not part of our research, simply selecting the middle of each subspace obviates many of the advantages of a *k*-d tree over other spatial data structures. Hence, we elect to use a simple function which results in an approximately balanced tree.

Linear Tree

Spatial tree GPU algorithms often use linear trees, which are better vectorizable than traditional trees. Linear trees [Gargantini, 1982] [Kim and Park, 1989] involve constructing the tree nodes from the given elements, wherein each node is constructed with a code representing its location in the tree – a Morton code – and storing the nodes in a linear array [Morton, 1966].

For example, for a quadtree, the position in a given node may be represented with 2 bits: 00 is top left, 01 is top right, 10 is bottom left, 11 is bottom right. Then, for a tree with a fixed height of 4, every element's position may be represented with 8 bits. The Morton

code 00110110 then means that the element is in the top-left quadrant of the space, within that, bottom-right, top-right, and bottom left, respectively.

The nodes are then sorted according to their Morton codes, and the result is a linear array representing a tree. This linear tree may be searched via a binary search of the sorted Morton codes.

00	01
10	11

Figure 4: linear tree Morton code visualization

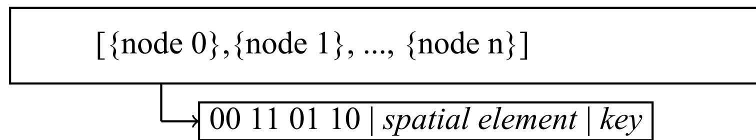

Figure 5: linear tree node visualization

Figure 4 presents a visualization of Morton codes indexing a quadtree level. Figure 5 presents a visualization of a linear tree array, and an expanded view of a single node. As we see, the tree is stored as a linear array, wherein each element of the array contains the Morton code used to search the tree, the spatial element of that node, and a key indexing external data. A key is not strictly required for a spatial element; however, a key will almost always be required for applications, and therefore we assume its existence.

This sorted linear tree may be considered analogous to the linear array typically used to store a binary heap for a heapsort. Once sorted by Morton code, the linear tree may be searched in logarithmic time to find an element in a given position, or the elements in a given range.

CHAPTER III

ALGORITHMS

Here we present our algorithms for heterogeneous construction of linear quadtrees, R-trees, and linear k-d trees. As aforementioned, we modify GPU algorithms for quadtree construction [Lauterbach et al., 2009] and R-tree construction [Luo et al., 2012, p. 356], and we present a novel heterogeneous k-d tree construction algorithm which is built around a parallel quicksort algorithm [Tsigas and Zhang, 2003]. We overview the existing homogeneous GPU tree construction algorithms which we modify, detail how we have modified them, detail our novel k-d tree construction algorithm, and overview a parallel quicksort algorithm which it requires.

An SIMD processor is incapable of executing jump instructions on multiple data, but must execute each jump serially. This is because a jump involves changing the instruction pointer, while each vector processing unit of the SIMD processor may only change data. Modern GPUs are not true SIMD, but rather have a few MIMD processors – perhaps one for every ten or hundred vector processing units [Lindholm et al., 2008]. The Nvidia corporation calls this architecture "single instruction, multiple threads" or SIMT [Nvidia, 2009]. Even so, jumps, which primarily involve if statements in structured programming languages, must be minimized to fully use the SIMD/SIMT GPU architecture. Part of our algorithm design involves this minimization of conditionals and jumps.

For our pseudocode, we use the following conventions to denote parallelism. Re-

gions which are executed in parallel on the MIMD CPU are denoted by $parallel for$, which indicates that the block is to execute for each index of the loop in parallel, or $parallel$, which indicates that all statements within the block are to execute in parallel. Regions which are executed in parallel on the SIMD GPU are denoted by $vector for$, which indicates that the block is to execute for each index of the loop in parallel on a vector processing element. Note that all data referenced within a $vector for$ block must be allocated and copied to the GPU co-processor, for a GPU implementation of the heterogeneous algorithm. The overhead of transferring the GPU instructions to the co-processor is also incurred.

Linear Quadtree

The basic serial algorithm for creating a linear quadtree is presented in Algorithm 1. We have adapted the GPU algorithm for linear bounded volume hierarchy construction presented by Lauterbach et al. [2009].

As we see in Algorithm 1, the tree is constructed in two steps. First, we create the nodes of the tree. Second, we sort the nodes based on their Morton code locations. The tree may then be searched in logarithmic time via a binary search using the Morton codes as keys.

Node Creation

The $CreateNodes$ function in Algorithm 1 iterates over each point. For each point, it creates that point's Morton code. For the 2-dimensional tree, the code for each depth of the tree consists of 2 bits: x and y. We calculate these bits, shift the existing location (of previous depths) left by two bits, and logical or the code with this current depth's bits. We then recalculate the boundary for the next quadrant depth, and loop, for each depth.

Algorithm 1 serial algorithm for linear quadtree construction in pseudocode

1: **procedure** CreateNode($point, node, boundary, depth$)
2: $node.point \leftarrow point$
3: **for** $i \leftarrow 0, depth$ **do**
4: $bit1 \leftarrow 0$
5: $bit2 \leftarrow 0$
6: **if** $point.y > (boundary.topleft.y + (boundary.bottomright.y - boundary.topleft.y)/2)$ **then**
7: $bit1 \leftarrow 1$
8: **end if**
9: **if** $point.x > (boundary.topleft.x + (boundary.bottomright.x - boundary.topleft.x)/2)$ **then**
10: $bit2 \leftarrow 1$
11: **end if**
12: $bits \leftarrow (bit1 << 1)|bit2$
13: $node.location \leftarrow node.location << 2)|bits$
14: $newwidth \leftarrow (boundary.bottomright.x - boundary.topleft.x)/2$
15: $newheight \leftarrow (boundary.bottomright.y - boundary.topleft.y)/2$
16: **if** $point.x - boundary.topleft.x > newwidth$ **then**
17: $boundary.topleft.x \leftarrow boundary.topleft.x + newwidth$
18: **end if**
19: **if** $point.y - boundary.topleft.y > newheight$ **then**
20: $boundary.topleft.y \leftarrow boundary.topleft.y + newheight$
21: **end if**
22: $boundary.bottomright.x \leftarrow boundary.topleft.x + newwidth$
23: $boundary.bottomright.y \leftarrow boundary.topleft.y + newheight$
24: **end for**
25: **return** $node$
26: **end procedure**
27: **procedure** CreateNodes($points, boundary, depth$)
28: $tree \leftarrow newLinearQuadtree$
29: **for** $i \leftarrow 0, points.length$ **do**
30: $tree.nodes[i] \leftarrow$ CreateNode($points[i], tree.nodes[i], boundary, depth$)
31: **end for**
32: **return** $tree$
33: **end procedure**
34: **function** Comparator($nodea, nodeb$)
35: **return** $nodea.location < nodeb.location$
36: **end function**
37: **procedure** CreateLinearQuadtree($points, boundary, depth$)
38: $tree \leftarrow$ CreateNodes($points, boundary, depth$)
39: $tree.nodes \leftarrow$ Sort($tree.nodes, comparator$)
40: **return** $tree$
41: **end procedure**

Algorithm 2 arithmetic equivalent of node boundary calculations in pseudocode

1: $boundary.topleft.x \leftarrow floor((point.x - boundary.topleft.x)/newwidth) * newwidth + boundary.topleft.x$
2: $boundary.topleft.y \leftarrow floor((point.y - boundary.topleft.y)/newheight) * newheight + boundary.topleft.y$

In order to efficiently execute this algorithm on a vector processor, we must eliminate conditionals as much as possible, such that our algorithm is primarily or entirely arithmetic. The node creation function as presented in Algorithm 1 has two pairs of identical conditionals – setting the new boundary, and calculating the Morton codes. For vector execution, the conditionals for setting the new boundary may be easily transformed into arithmetic as in Algorithm 2. An optimizing compiler may automatically perform this optimization. However, it may not, so it is better that our implementation explicitly remove the conditionals for vector execution.

Algorithm 3 arithmetic equivalent of Morton code calculations in pseudocode

1: $bit1 \leftarrow point.y > (boundary.topleft.y + (boundary.bottomright.y - boundary.topleft.y)/2)$
2: $bit2 \leftarrow point.x > (boundary.topleft.x + (boundary.bottomright.x - boundary.topleft.x)/2)$

We would also like to transform the Morton code calculations into logical operations. The assignment of logical operations to integral values is language-dependent. The C and C++ languages guarantee that logical operations assigned to an integral type will have a value of 0 when false, and a value of 1 when true [ISO, 1999, p. 86] [C++, 2003, p. 62]. Standard Fortran does not permit the assignment of logical values to integral types [J3, 1991, p. 72], but GNU Fortran Extensions (and some other compilers' extensions) permit said assignment with the same guarantees [FSF, 2007, p. 28]. Furthermore, in assembly languages, logical instructions operate on numeric values. When using a language which

permits the assignment of logical operations to integral values, and guarantees 0 or 1 values, the Morton code calculations from Algorithm 1 may be transformed as in Algorithm 3.

We note that Algorithm 3 could be easily adapted for languages with different guarantees. For example, for a language which guarantees true values are greater than zero, and false values are less than zero; or that false values are zero, and true values are nonzero. We also note that, as in Algorithm 2, the compiler may or may not perform this optimization automatically.

Our node creation function is now perfectly vectorizable at point-granularity. Typically, tree depth will range from 16–64, in order to fit the Morton code into an integer approximately the size of the architecture, for performance. Since sorting and searching the tree must compare each Morton code, codes of significantly greater size than the architecture will impact construction and search performance. Then, since depth is small, it is sufficiently granular for each vector processing element to compute a whole point.

Our GPU algorithm then simply executes each iteration of the for loop in the $CreateNodes$ function on a GPU vector core, replaces the boundary calculations per Algorithm 2, and replace the $Sort$ call in $CreateLinearQuadtree$ with a GPU sort. For our results, we will be using a preexisting GPU radix sort library.

For our heterogeneous algorithm, we observe that the Morton code calculation functions ($CreateNodes$, $CreateNode$) are perfectly vectorizable. There are no conditionals; it is entirely arithmetical. Therefore, the only poorly vectorizable part of this algorithm is the sort. As stated previously, sort is poorly vectorizable, and typically only results in a single order of magnitude speedup over serial execution. Contrariwise, parallel MIMD algorithms typically see linear speedup with the number of cores, minus overhead and memory band-

width. So, for our heterogeneous algorithm, we will be implementing a parallel mergesort which executes on the multicore CPU.

Parallel Mergesort

Our heterogeneous linear quadtree construction algorithm requires an MIMD parallel sort. We present an example parallel mergesort, as presented in McCool et al. [2012]. For a basic introduction to Mergesort, see Cormen et al. [2009, p. 34].

Parallel mergesort has two parallel components: subdivision and merge. Parallelizing the subdivision is trivial. The data of each subdivision is independent, and hence we simply execute each partition of the subdivision in parallel. By virtue of the parallelized subdivisions, merging the sorted sub-arrays will also be parallelized. However, because the subdivisions create a tree structure, for an array of size n we will only see a parallel speedup of $\log n$. In order to achieve a linear speedup with the number of MIMD cores, we must parallelize the merge step. This is less trivial, and we present the basic algorithm here as part of our overall construction algorithm.

The basic algorithm for parallel merging of two sorted subsequences A and B is as follows [McCool et al., 2012, p. 300].

1. Let A_0, A_1 be two contiguous subsequences of A of approximately equal length.
2. Let a_0 be the first element in A_1.
3. Use a binary search ($O(\log n)$) on B to find the element b_q such that $b_q < a_0 < b_{q+1}$.
4. Let B_0, B_1 be two contiguous subsequences of B such that $b_q \in B_0, b_q \notin B_1$.
5. Let C_0 be the recursive, parallel merge of A_0 and B_0.

6. Let C_1 be the recursive, parallel merge of A_1 and B_1.

7. Let C be the joining of sequences C_1 and C_2 such that $C = \{c_{0i}|i = 0..q, c_{1i}|i = q..\}$.

Algorithm 4 parallel merge in pseudocode

```
 1: procedure Merge(A, B, C)
 2:     mergecutoff ← 2000                ▷ may be varied to optimize for architecture
 3:     if A.size + B.size < mergecutoff then
 4:         SerialMerge(A, B, C)
 5:         return
 6:     end if
 7:     amiddle ← A.first
 8:     bmiddle ← B.first
 9:     if A.size < B.size then
10:         bmiddle ← B.first + B.size/2
11:         amiddle ← upperbound(A.first, A.last, bmiddle)
12:     else
13:         amiddle ← A.first + A.size/2
14:         bmiddle ← lowerbound(B.first, B.last, amiddle)
15:     end if
16:     cmiddle ← C.first + (amiddle − A.first) + (bmiddle − B.first)
17:     parallel
18:         Merge(A.first, amiddle, B.first, bmiddle, C.first)
19:         Merge(amiddle, A.last, bmiddle, B.last, cmiddle)
20:     end parallel
21: end procedure
```

The parallel merge algorithm [McCool et al., 2012, p. 301] is demonstrated in Algorithm 4.

Let $T_\infty(n)$ be the time to execute a parallel algorithm on data of size n with ∞ processors. The parallel mergesort has asymptotic runtime of $T_\infty(n) = \Theta(\lg_3 n)$, and an asymptotic speedup over serial mergesort of $\Theta(n/\lg_2 n)$ [McCool et al., 2012, p. 304]. This suggests that thousands of processors might be used for sequences with millions of elements. Unfortunately, as we will see, architectural constraints such as memory bandwidth become a practical issue constraining parallel speedup.

We note that parallel mergesort is a stable sort [McCool et al., 2012, p. 300].

R-tree

For our heterogeneous R-tree construction, we modify the GPU algorithm presented in Luo et al. [2012, p. 356]. The GPU algorithm is presented in Algorithm 5.

Algorithm 5 GPU R-tree construction in pseudocode

```
 1: function Point.Compare(a, b)
 2:     return a.x < b.x                              ▷ sorting creates an "xpack"
 3: end function
 4: function Rectangle.Compare(a, b)
 5:     return a.BottomLeft < b.BottomLeft            ▷ sorting creates a "lowxpack"
 6: end function
 7: ELEMENTS-PER-NODE ← 10 ▷ may be changed for architecture optimization
 8: procedure CreateRtree(Data)
 9:     Data ← Sort(Data, Data.Type.Compare)
10:     Nodes ← newList
11:     level ← 0                                     ▷ second level
12:     Nodes[level] ← newList
13:     for i ← 0, Data.length/ELEMENTS-PER-NODE do
14:         for j ← 0, ELEMENTS-PER-NODE do
15:             nodes[level].Insert(data[i + j])
16:         end for
17:     end for
18:     level ← level + 1
19:     while Nodes[level].length > 1 do              ▷ until the root node
20:         len ← Nodes[level − 1].Length/ELEMENTS-PER-NODE
21:         vector for i ← 0, len
22:             node ← newNode
23:             for j ← 0, ELEMENTS-PER-NODE do
24:                 node.Insert(Nodes[level - 1][i + j])
25:             end for
26:             nodes[level] ← node
27:         end vector for
28:     end while
29: end procedure
```

We first sort the spatial data, and then create leaf nodes for each spatial element. Then, for each level of the tree, we assign the sorted nodes, in order. Since the nodes are

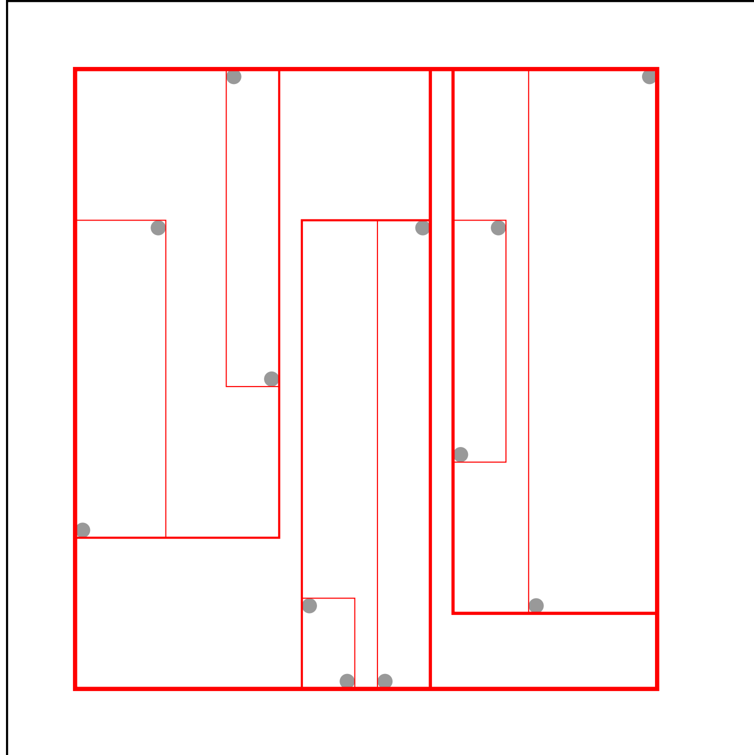

Figure 6: r-tree xpack visualization

sorted according to their bottom-left x coordinate, the structure of the tree is known as a lowxpack [Luo et al., 2012, p. 356].

Figure 6 presents a visualization of an R-tree wherin nodes are selected by their x-coordinate proximity, that is, a "xpack," as performed by the GPU algorithm presented by Luo et al. [2012, p. 356] and hence also our modified heterogeneous version of their algorithm. Contrast this tree with the arbitrary selection algorithm presented in Figure 2. Observe how nodes which are nearby on both axes may not be selected in favor of nodes which are closer on the x-axis but distant on the y-axis.

Note that this is not a linear tree. While the construction algorithm is similar to the previous linear quadtree, the constructed R-tree is an ordinary tree wherein each node contains pointers to its children. This has the advantage of having logarithmic insert time. This

has the disadvantage that for infinite processors, the parallelization speedup is $O(\log n)$. The nature of the tree notwithstanding, if we inspect the construction algorithm, we notice that we know the size of the tree and where each node will be placed beforehand. Therefore, the tree's memory may be allocated beforehand in a single allocation. This allows for both faster allocation, as well as spatial locality (which may improve the performance of hardware caching). We note that if memory is pre-allocated and the tree is mutable, additional logic may be required to free memory of inserted nodes versus original nodes.

We observe that this algorithm is similar in structure to the linear quadtree construction algorithm, despite not constructing a linear tree. Both algorithms involve a discrete sort, preceded or followed by node construction.

As with the linear quadtree, the nodes for this R-tree may be constructed entirely arithmetically. Node construction for branches involves calculating the pointers to each child node, and then calculating the bounding box given the boundaries of the child nodes. Node construction for leaf nodes involves storing the spatial data. All of these operations may be done entirely with arithmetical and memory operations, without conditionals or jumps, and hence on a vector processing element.

Therefore, for our heterogeneous construction algorithm, we will be first sorting the spatial data on the MIMD CPU, and then creating the nodes on the SIMD GPU.

As with the linear quadtree, the parallel mergesort presented in McCool et al. [2012, p. 300] may be used, or any other fast parallel MIMD sorting algorithm.

Linear *k*-d tree

For our linear *k*-d tree, we have designed a novel construction algorithm based around quicksort partitioning. Linear *k*-d tree construction is similar to linear quadtree construction. For the tree's data layout, we will construct a linear array, wherein each element is a spatial data with a corresponding location code. The location code value for each level of the tree is a simple boolean: left or right of the splitting hyperplane.

Algorithm 6 heterogeneous *k*-d tree construction pseudocode

```
 1: function CreateKdtreeIteration(Data, Splits, splitvalue)
 2:     buffer ← Copy(Data)
 3:     parallel
 4:         nextsplitright ← FindNextSplitRight(buffer, splitvalue)
 5:         nextsplitleft ← FindNextSplitLeft(buffer, splitvalue)
 6:         splitposition ← ParallelQuicksortPartition(Data, splitvalue)
 7:     end parallel
 8:     parallel
 9:         CreateKdtreeIteration(Data[:splitposition], Splits, nextsplitleft)
10:         CreateKdtreeIteration(Data[splitposition:], Splits, nextsplitright)
11:         Splits.Insert((splitposition, splitvalue))
12:     end parallel
13: end function
14: function CreateKdtree(Data)
15:     Splits ← newList
16:     Splits[0] ← FindSplit(Data)
17:     CreateKdtreeIteration(Data, Splits, Splits[0])
18:     MortonCodes ← CreateMortonCodes(Data, Splits)
19:     return (Data, Splits, MortonCodes)
20: end function
```

Unlike the linear quadtree, however, the *k*-d tree must also store the location of each partition. We must know at what point the split takes place. We could avoid the necessity of storing partition locations by always splitting along the middle of the parent's bounding box. However, this eliminates the *k*-d tree's advantages from being able to select the split location, and essentially reduces the *k*-d tree to a binary version of a quadtree.

Algorithm 7 logarithmic parallel splitting hyperplane calculation pseudocode

```
1:  function FindSplits(Data, Splits, splitlocation)
2:      Splits[splitlocation] ← FindSplit(Data)
3:      splitleftindex ← splitlocation * 2 + 1
4:      splitrightindex ← splitlocation * 2 + 2
5:      if splitleftindex > Data.length or splitrightindex > Data.length then
6:          return
7:      end if
8:      parallel
9:          FindSplits(Data, Splits[splitleft], splitleft)
10:         FindSplits(Data, Splits[splitright], splitright)
11:     end parallel
12:     return Splits
13: end function
```

Therefore, we desire to create a heterogeneous construction algorithm which permits the selection of the splitting hyperplane.

Our algorithm is presented in basic pseudocode in Algorithm 6. The primary difficulty, and the reason we cannot simply apply an existing sorting algorithm as-is, is that we cannot efficiently calculate the splitting hyperplane for each node of the tree beforehand.

We could calculate each split in parallel before sorting by calculating the first splitting hyperplane, then calculating the splitting hyperplanes of its children, and so on, as presented in Algorithm 7. This would allow us to subsequently perform a standard sort. However, we observe that this method only experiences a parallel speedup of $O(\log n)$, which is not scalable for manycore architectures.

Alternatively, for each leaf node in the tree, we could calculate all the preceding splitting hyperplanes on the same MIMD core, in order to achieve maximum parallelization, as presented in Algorithm 8. While parallelizing linearly, it is obvious this solution will be no faster than the preceding option.

Because each splitting hyperplane is dependent on the previous splitting hyperplane,

Algorithm 8 linear inefficient parallel splitting hyperplane calculation pseudocode

```
1: function FindSplits(Data, TreeNodes, Splits)
2:     parallel for each node in TreeNodes
3:         nextsplit ← FindSplit(Data)
4:         for split ← 0, node.depth do
5:             nextsplit ← FindSplit(Data, nextsplit)
6:         end for
7:         Splits[node] ← nextsplit
8:     end parallel for
9:     return Splits
10: end function
```

it is not possible to independently parallelize the splitting hyperplane calculations with a speedup greater than $\log n$ for n spatial elements.

Our solution for a scalable splitting hyperplane calculation is to perform the calculation in parallel with another calculation. Both the sort and the location code calculations require the splitting hyperplanes for the parent of each node being calculated, and furthermore the sort must precede the location code calculation.

Therefore, our construction algorithm will calculate each preceding splitting hyperplane in parallel with each stage of the sort. That is, the splitting hyperplanes cannot be independently calculated, but rather, at each stage of the sort, the preceding splitting hyperplane must be calculated in order to be used by the sort.

We observe that quicksort partitioning is a natural candidate for our construction algorithm. At each stage of the sort, we wish to calculate the splitting hyperplane, and then sort the spatial data such that all elements less than the hyperplane precede all elements which are greater than the hyperplane. This is precisely how each stage of quicksort operates.

Our basic heterogeneous construction algorithm is presented in Algorithm 6. We

begin by calculating the first split. Then, given the split point for the first node, in parallel, we:

1. calculate the splitting hyperplane for the left child node, given the current splitting hyperplane
2. calculate the splitting hyperplane for the right child node, given the current splitting hyperplane
3. partition the spatial data such that all nodes less than the splitting hyperplane precede all nodes greater than the splitting hyperplane.

We note that in order to calculate the splitting hyperplanes in parallel with sorting, we are required to iterate over the pre-sorted data. Thus, our split functions must traverse approximately twice the spatial elements (assuming a median split) of the child node for which they are calculating. Therefore the sequential execution time for our split calculations then becomes $2 \log n$. While this constant does not increase the asymptotic time, it does double the work which must be done, and decrease the efficiency. Unfortunately, without this inefficiency we cannot calculate the splitting hyperplanes in parallel with the sort, since each sort requires the splitting hyperplane for that node of the tree.

We also note that in order to calculate the splitting hyperplanes of the children in parallel with the sort, we cannot operate on the same data which is being concurrently sorted. Therefore, we must create a buffer into which the data is copied, such that the split and sort calculations do not operate on the same data. We also note that our partition function requires a slight modification from the standard quicksort partition algorithm. In the standard quicksort algorithm, one typically selects a pivot element, partitions the array,

and then recursively partitions each side. With our construction algorithm, we do not have an element, but rather a coordinate value for the splitting hyperplane. Thus, we partition on this value rather than an element of the data. Once complete, the partition function returns the index for which all elements preceding are less than the given hyperplane and all elements following are greater. We then store both the index and the value of the splitting hyperplane in the array of splits (which is required for searching the tree), and recursively sort.

We observe that our algorithm is not dependent on any particular splitting algorithm. Many splitting algorithms and heuristics exist, with various benefits and performance. It is beyond the scope of our research to investigate or assert an ideal splitting algorithm. Rather, our linear k-d tree construction algorithm is designed to work with any given splitting algorithm. Ideally, the splitting algorithm itself should be parallelized in order to provide further over-decomposition.

Algorithm 9 serial split calculation pseudocode

```
 1: procedure CreateKdtreeIteration(Data, Splits, splitvalue)
 2:     nextsplitright ← FindNextSplitRight(Data, splitvalue)
 3:     nextsplitleft ← FindNextSplitLeft(Data, splitvalue)
 4:     splitposition ← ParallelQuicksortPartition(Data, splitvalue)
 5:     parallel
 6:         CreateKdtreeIteration(Data[:splitposition], Splits, nextsplitleft)
 7:         CreateKdtreeIteration(Data[splitposition:], Splits, nextsplitright)
 8:         Splits.Insert(splitposition, splitvalue)
 9:     end parallel
10: end procedure
```

We note an alternative to the algorithm presented in Algorithm 6. Rather than using a buffer and computing the split points in parallel with the quicksort partition, we may compute them in serial without a buffer, as in Algorithm 9. This saves the cost of copying

memory for each partition of the data. Since there are $\log n$ partitions, and we must copy the entire data being partitioned, using a buffer to calculate split points in parallel with the partitioning requires $2 \cdot n$ memory copies. Whether the cost of these memory copies is greater than the cost of calculation splitting hyperplanes in serial will depend on the cost of the split calculation function.

Our simple median heuristic function is less expensive to serially compute than the buffer cost. Therefore, for our results, we calculate splitting hyperplanes in serial, as in Algorithm 9.

Parallel Quicksort Partition

Our heterogeneous linear *k*-d tree construction requires a parallel quicksort partition function. Because children are recursively sorted in parallel, the recursion itself is only parallelized to degree $\log n$ for infinite processing elements. Hence, our partitioning function must be parallelized to achieve scalability on manycore architectures.

For our parallel quicksort partition, we use the algorithm presented by Tsigas and Zhang [2003]. We briefly describe their algorithm as we have implemented it here. Note that we only use their partition function, not a unified quicksort. However, the parallel partition is the greater part of parallelizing the quicksort algorithm.

Their partition algorithm has two basic phases. First, the array to be partitioned is divided into blocks, which are independently partitioned in parallel with work-acquiring parallel processes. Once the first stage is complete, the array is partitioned such that some blocks are fully partitioned and some are not, and there exists a position in the array such that every element of all fully partitioned blocks preceding the given position is less than

the pivot, and every element of all fully partitioned blocks after the given position is greater than the pivot. That is, the array is partitioned at a given point, with the exception of some incomplete blocks. The number of incomplete blocks after the parallel partition stage will not exceed the number of parallel processes.

The second stage involves serially sorting and moving the incompletely partitioned blocks, such that the entire array is partitioned. The algorithm therefore scales well with the size of the data and the number of parallel processes, as long as the ratio of data to processes is sufficiently large.

Pseudocode for the parallel stage is presented in Algorithm 10. The parallel stage consists of executing a partitioner task with any number of parallel processes. The partitioner function takes the data and a block size, as well as iterators to the first and last block. Each execution of the partitioner function increments the first iterator with an atomic fetch-and-add function, and decrements the last iterator with an atomic fetch-and-subtract. Thus, that parallel task acquires the fetched first and last iterators as block indexes. The task then moves elements in the first block which are greater than the pivot to the second block, and vica versa, until either the first block consists entirely of elements less than the pivot, or the second block consists entirely of elements greater than the pivot (or both). The task then replaces the completed block, and atomically fetches another block.

All tasks continue partitioning blocks in parallel, until there are no more blocks. At this point, each task has at most one unfinished block. The indices of these unfinished blocks are placed in a shared array via atomic fetch-and-add operations to acquire an index into the unfinished array.

We note that because all operations to acquire data indices involve only atomic

Algorithm 10 parallel quicksort partitioner pseudocode

```
 1: procedure Partitioner(Data, pivot, nextleft, nextright, RemainingIndices, nextremaining)
 2:     leftindex ← AtomicFetchAdd(nextleft)
 3:     rightindex ← AtomicFetchSubtract(nextright)
 4:     while leftindex < rightindex do
 5:         neutralized ← Neutralize(Data[leftindex], Data[rightindex], pivot)
 6:         if neutralized = LEFT or neutralized = BOTH then
 7:             leftindex ← AtomicFetchAdd(nextleft)
 8:         end if
 9:         if neutralized = RIGHT or neutralized = BOTH then
10:             rightindex ← AtomicFetchSubtract(nextright)
11:         end if
12:     end while
13:     if leftindex is remaining then
14:         RemainingIndices[AtomicFetchAdd((nextremaining)] ← leftindex
15:     end if
16:     if rightindex is remaining then
17:         RemainingIndices[AtomicFetchAdd((nextremaining)] ← rightindex
18:     end if
19: end procedure
20: function Neutralize(Left, Right, pivot)
21:     leftindex ← 0
22:     rightindex ← 0
23:     while leftindex < Left.size and rightindex < Right.size do
24:         while Left[leftindex] < pivot do
25:             leftindex ← leftindex + 1
26:         end while
27:         while Right[rightindex] > pivot do
28:             rightindex ← rightindex + 1
29:         end whileSwap(Left[leftindex], Right[rightindex]))
30:     end while
31:     if leftindex = Left.size and rightindex = Right.size then
32:         return BOTH
33:     end if
34:     if leftindex = Left.size then
35:         return LEFT
36:     end if
37:     return RIGHT
38: end function
```

increment or decrement functions which cannot fail, this work-acquiring algorithm is *wait-free* and no less efficient than a work-stealing algorithm.

We also note that the number of identical partitioner tasks which may be executed in parallel is only constrained by the overhead of parallel task scheduling and memory bandwidth.

Algorithm 11 serial quicksort partitioning pseudocode

```
 1: function PartitionRemaining(Data, pivot, RemainingIndices, middle)
 2:     Sort(RemainingIndices)
 3:     remainingleft ← RemainingIndices[0]
 4:     remainingright ← RemainingIndices[RemainingIndices.size]
 5:     while remainingleft < remainingright do
 6:         neutralized ← Neutralize(Data[remainingleft], Data[remainingright])
 7:         if neutralized = LEFT or neutralized = BOTH then
 8:             if remainingleft > middle then
 9:                 SwapBlocks(Data[remainingleft], middle)
10:                 middle ← middle + BLOCKSIZE
11:             end if
12:             remainingleft ← remainingleft + 1
13:         end if
14:         if neutralized = RIGHT or neutralized = BOTH then
15:             if remainingright < middle then
16:                 SwapBlocks(Data[remainingright], middle)
17:                 middle ← middle − BLOCKSIZE
18:             end if
19:             remainingright ← remainingright − 1
20:         end if
21:     end while
22:     remainingindex ← 0
23:     if neutralized = LEFT then
24:         remainingindex ← remainingright
25:     else
26:         remainingindex ← remainingleft
27:     end if
28:     SwapBlocks(Data[remainingindex], Data[middle])
29:     Sort(Data[remainingindex])
30:     pivotposition ← remainingindex
31:     while Data[pivotposition] < pivot do
32:         pivotposition ← pivotposition + 1
33:     end while
34:     return pivotposition
35: end function
```

Once the parallel partition stage is complete, the array will be partitioned such that

arbitrary blocks are incomplete. The serial phase will sort these blocks and place them such that the final array is partitioned. Pseudocode for the serial phase is presented in Algorithm 11.

The serial phase iterates over two blocks at a time and partitions them, much like the parallel partitioner function. However, a block that is partitioned on one side will not necessarily be on the correct side in the array. When this occurs, the serial phase must swap that block with an unfinished block on the other side. If no unfinished blocks exist on the other side, the block immediately beyond the partition will be swapped and the partition location moved accordingly. Finally, when a single unfinished block remains, that block will be sorted and placed in the middle, and the precise partition location calculated within the final block.

Algorithm 12 parallel quicksort partitioning pseudocode

1: **function** ParallelQuicksortPartition(Data, pivot, threadsnumber)
2: $nextleft \leftarrow 0$
3: $nextright \leftarrow Data.size$
4: $nextremaining \leftarrow 0$
5: $RemainingIndices \leftarrow newArray[threadsnumber]$
6: **parallel for** 0 to $threadsnumber$
7: Partitioner(Data, pivot, nextleft, nextright, RemainingIndices, nextremaining)
8: **end parallel for**
9: $pivotposition \leftarrow$ PartitionRemaining$(Data, pivot, RemainingIndices, nextleft)$
10: **return** pivotposition
11: **end function**

Then, the final parallel partition algorithm involves executing these two steps: the parallel block partition phase, and the serial partitioning of remaining blocks, as presented in Algorithm 12.

The parallel quicksort algorithm presented by Tsigas et al is significantly more complex than the brief descriptions and pseudocode we present here for comprehension. For

full details of the parallel quicksort algorithm, implementation, analysis, and performance, see Tsigas and Zhang [2003].

We also note that our tree construction is not specific to this quicksort algorithm. Any partitioning algorithm which may be given a value to partition on (rather than an index) and returns the index is suitable. For example, Altman and Igarashi [1989] and Altman and Chlebus [1990].

Vector Morton Code Calculation

Once the linear k-d tree is sorted and the splitting hyperplanes calculated, the Morton codes for each contained element must be calculated, to be used by search and other traversal operations. The pseudocode for the GPU Morton code calculation function is presented in Algorithm 13.

Recall that $Splits$ is a tree stored in a linear array. Calculating the Morton code for each element is trivial, and entirely arithmetical. For each element, we traverse the $Splits$ linear tree. If the element is less than the current split, we insert an 0 in the Morton code; else, a 1. We then traverse to the left or right child of the $Splits$ tree, accordingly, for the height of the tree.

We have presented human-readable pseudocode for Morton code calculation in Algorithm 13. As with the linear quadtree, the conditionals may be transformed into arithmetic for the vector processing element, either manually or by the compiler. We present the pseudocode for this in Algorithm 14. Note that, as with the linear quadtree Morton code calculations, this pseudocode assumes boolean logic may be manipulated as arithmetical values, and that the language guarantees that a boolean value of false evaluates to 0 and

Algorithm 13 vector Morton code calculation pseudocode

```
1: procedure CreateMortonCodes(Data, Splits)
2:     MortonCodes ← newArray[Data.size]
3:     vector for each element in Data
4:         MortonCodes[element] ← CreateMortonCode(element, Splits)
5:     end vector for
6: end procedure
7: function CreateMortonCode(element, Splits)
8:     code ← 0
9:     currentsplit ← 0
10:    for 0 do to Splits.depth
11:        code ← code << 1
12:        if element < Splits[currentsplit] then
13:            code ← code|0        ▷ no effect, but shown for comprehension
14:            currentsplit ← GetLeftChild(currentsplit)
15:        else
16:            code ← code|1
17:            currentsplit ← GetRightChild(currentsplit)
18:        end if
19:    end for
20:    return code
21: end function
```

true evaluates to 1. We also note that the split depth is a constant value, and thus the loop in Algorithm 13 and Algorithm 14 may be unrolled by the compiler, in order to eliminate the jump instruction required by the loop (which cannot be executed by a vector processing element).

Algorithm 14 arithmetic morton code calculation pseudocode

```
1: function CreateMortonCode(element, Splits)
2:     code ← 0
3:     currentsplit ← 0
4:     for 0 do to Splits.depth
5:         code ← code << 1
6:         code ← code|(element > Splits[currentsplit])
7:         currentsplit ← (GetLeftChild(currentsplit) ∗ (element <
       Splits[currentsplit])) + (GetRightChild(currentsplit) ∗ (element >=
       Splits[currentsplit]))
8:         return code
9:     end for
10: end function
```

Parallel Linear Fixed-Length Tree

Finally, the implementation of our linear *k*-d tree construction algorithm requires a unique data structure for storing the calculated splitting hyperplanes: a parallel, linear, fixed-length tree.

Our structure is presented in Algorithm 15. We require adding nodes to the tree to be safe for parallel access, since many threads of execution will add to the tree in parallel during construction. This would not be an issue for a traditional pointer tree [Cormen et al., 2009, p. 246], as in Algorithm 16, wherein each child is represented with a different variable, as long as memory allocation for each node were safe for parallel access, since parallel assignment to different variables is safe. However, we also require a linear tree, which may be easily passed to the vector co-processor (the GPU), and which may be traversed arithmetically. It is not strictly necessary that the linear structure be of fixed length. However, since we know the maximum length required – the length of the spatial input data – creating a linear structure which does not require reallocations is simpler and faster.

Note that we do not require that, nor is the structure presented in Algorithm 15, safe for parallel construction of the same tree node. In our linear *k*-d tree construction algorithm (presented in Algorithm 6), only one thread of execution will ever attempt to create any given child node of the splitting hyperplanes tree.

Also note that the static indexing traditionally used by heapsort is not sufficient, because our *k*-d tree may not be perfectly balanced. When the tree is not perfectly balanced, statically indexing children (e.g. $2n + 1$ and $2n + 2$) will produce large empty spaces in the linear tree. Then, either a very large array will be required to store the Morton codes,

Algorithm 15 parallel linear fixed-length tree pseudocode

```
1: structure FixLenTree
2:     treeend ← ∞
3:     structure Node
4:         Spatial Value ▷ this is the hyperplane, of one dimension smaller than the input space
5:         Integer Right
6:         Integer Left
7:     end structure
8:     procedure FixLenTree(length)
9:         Length ← length
10:        Nodes ← newArray
11:        NodeAcquirer ← 0
12:    end procedure
13:    function InsertRoot(value)
14:        position ← AtomicFetchAdd(NodeAcquirer)
15:        node ← Nodes[position]
16:        node.Value ← value
17:        node.Left ← treeend
18:        node.Right ← treeend
19:        return position
20:    end function
21:    function Insert(parent, left, value)
22:        position ← AtomicFetchAdd(NodeAcquirer)
23:        if position≥Length then    ▷ this safeguard will never occur in the k-d tree construction algorithm
24:            return treeend
25:        end if
26:        node ← Nodes[position]
27:        node.Value ← value
28:        node.Left ← treeend
29:        node.Right ← treeend
30:        if left then
31:            Nodes[parent].Left ← position
32:        else
33:            Nodes[parent].Right ← position
34:        end if
35:        return position
36:    end function
37:    Integer Length
38:    Node Array Nodes
39:    Integer NodeAcquirer
40: end structure
```

Algorithm 16 traditional pointer tree pseudocode

1: **structure** TreeNode
2: Data key
3: TreeNode $left\text{-}child$
4: TreeNode $right\text{-}child$
5: **end structure**

or the tree depth (as specified by the Morton codes) will be very small. By using dynamic indexing (which must be safe for parallel access) we only require an array as large as we need, with no empty locations.

Pipelining

In the event that multiple trees must be constructed, heterogeneous algorithms may be pipelined. Homogeneous algorithms cannot be pipelined, since the tree construction is fully utilizing the CPU or GPU for each tree which is constructed. However, a heterogeneous algorithm can be pipelined such that for two sequential procedures which must execute on the CPU and GPU, the nth tree may execute the first procedure on one processor architecture while the $(n+1)^{th}$ tree is executing the second procedure on the other architecture.

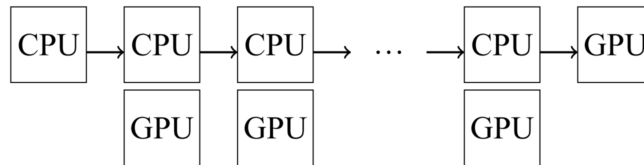

Figure 7: pipeline visualization

Figure 7 presents a visualization of pipelining. As we can see, the time to construct each of n trees, the construction of each of which consists of two sequential stages on different types of processors, where p is the time to execute on the first processor and q is the time to execute on the second processor, will be $p+n \cdot max(p,q)+q$, or $O(n \cdot max(p,q))$ for each tree. That is, the amortized time to construct n trees is n times the slowest execution time of the two architectures. In contrast, the time to construct n trees with a homogeneous algorithm which may not be pipelined is $n \cdot k$, where k is the execution time of the algorithm for a single tree.

The pipeline pattern for parallel processing is discussed in McCool et al. [2012, p. 99] and detailed in Mattson et al. [2010, p. 103].

CHAPTER IV

RESULTS

We present our results of executing implementations of our heterogeneous algorithms, contrasted with the running time of analogous homogeneous CPU and GPU algorithms. We present the running times of constructing an individual tree, for each of our linear quadtree, R-tree, and linear *k*-d tree construction algorithms. We also present the running times for pipelined construction of ten trees, for each respective algorithm.

Methodology

Our tests are performed on a machine with two 64-bit 2.2 Ghz 6-core AMD Opteron 2427 processors, each with 6x64KB L1 instruction cache, 6x64KB L1 data cache, 6x512KB L2 cache, and 6MB L3 cache. The GPU is part of an NVidia Tesla S2050 Computing System, with 448 CUDA cores at 1.15Ghz and 3GB GDDR5 memory. The operating system is CentOS 5.8 running Linux kernel 2.6.18 x86_64.

All tests are the average (mean) of three executions, rounded to the nearest millisecond. There were no outliers. All tree construction algorithms are controlled for both the number of MIMD threads and the number of points. Charts are plotted as line graphs with straight lines between measured values. For varying point tests, numbers of points are tested for each order of magnitude from 10 to 10,000,000 (10, 100, 1000, et cetera). For varying threads of execution, threads are tested for 1, 2, 4, 8, 12, 14, and 16 on the 12-core machine.

All our algorithms are implemented in the C++11 language, with CUDA bindings for vector GPU execution, and the Intel Threading Building Blocks library for parallel CPU execution. CUDA is a compute platform for general purpose computing on NVidia GPUs [Garland et al., 2008]. Intel Threading Building Blocks is a C++ library which allows for directly programming high-level parallel constructs such as $parallelfor$ [Kim and Voss, 2011].

All timing is gathered via the C++11 *std::chrono::high_resolution_clock*. Timing results are only for the construction of the trees, not for the entire program execution. Results include both the time to create CPU threads and initialize the threading library, and the time to initialize the GPU kernel and transfer data.

Code is compiled with G++ 4.8 and NVCC 5.0, using $-O3$ optimization. We note that the AMD Opteron 2427 has SSE vector extensions [AMD, 2005], and therefore compiled with optimizations [FSF, 2013], the CPU code itself is heterogeneous, in that it includes vector instructions which will be executed by the MIMD processor.

Linear Quadtree

Our linear quadtree for testing is implemented in C++11 and CUDA, as previously noted.

For our homogoeneous GPU comparison algorithm, we use the CUB library [Nvidia, 2014] which implements a GPU radix sort as presented by Satish et al. [2009b]. For our heterogeneous algorithm, we found that the Threading Building Blocks $parallel_sort$ was faster than our parallel mergesort implementation on our testing architecture. Therefore, our we use *tbb::parallel_sort* for our heterogeneous results [Kim and Voss, 2011].

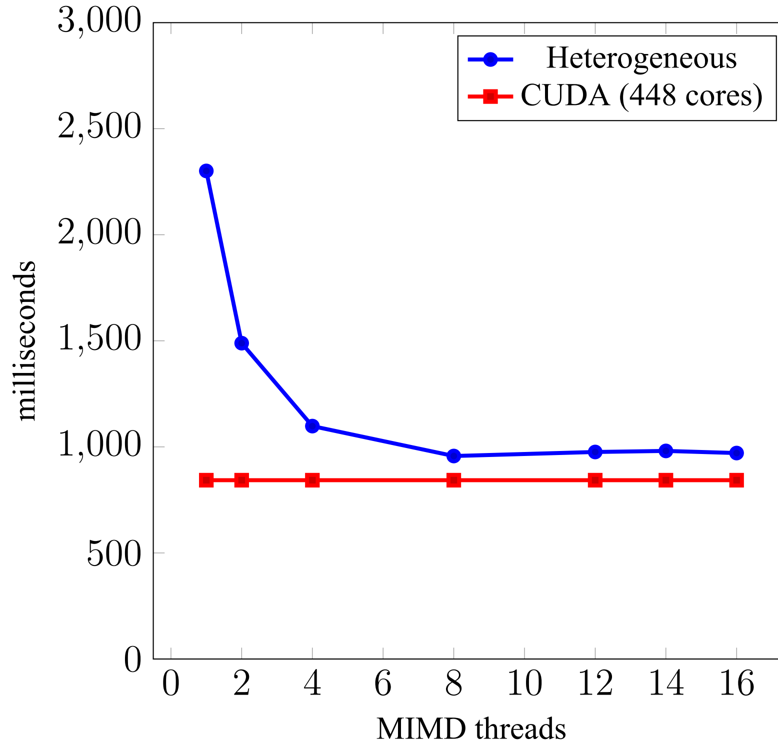

Figure 8: linear quadtree for 1–12 MIMD threads, with 10,000,000 points

Our testing results are plotted in Figure 8 and Figure 9. Note the x-axis of Figure 9 is logarithmic. We see that our heterogeneous algorithm is approximately 90% as fast as the homogeneous GPU algorithm (843 versus 957 milliseconds with 8 threads).

We note that the performance of the homogeneous CUDA algorithm in Figure 8 is not affected by the number of MIMD threads, because the algorithm is executed entirely on the GPU. For all homogeneous GPU and heterogeneous results, we fully utilize all vector processing elements (CUDA cores) on the GPU.

We expect this to be the slowest relative heterogeneous algorithm, since the quadtree is an extremely simple data structure, and only a single pass is required by the GPU for morton code calculation and sorting.

With only a 10% difference, we would expect that the results are dependent on

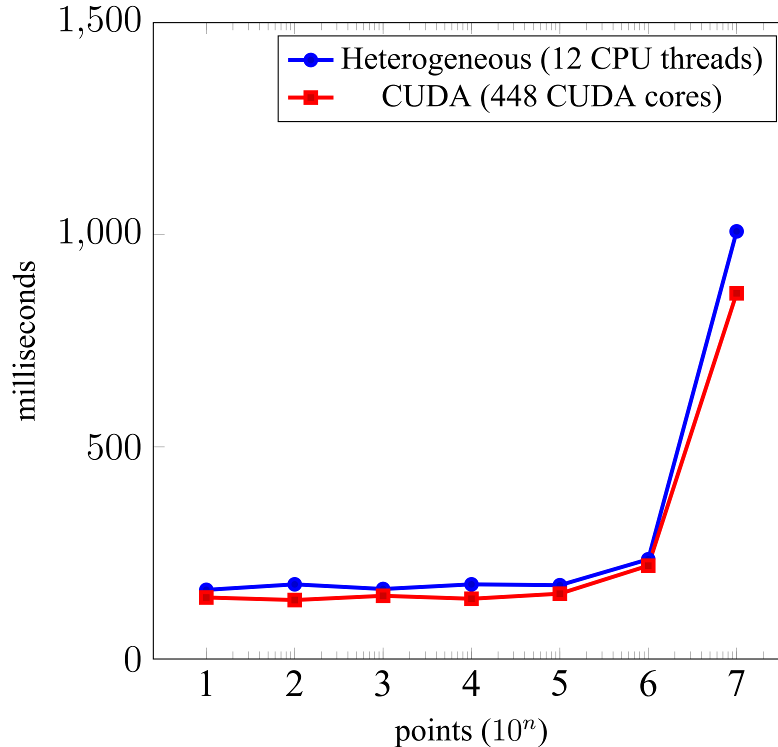

Figure 9: linear quadtree for 10–10,000,000 points, with 12 threads

the architecture. That is, with a GPU which is slower or has less vector processing elements, the heterogeneous algorithm might outperform the homogeneous GPU algorithm. Likewise, with a faster CPU, or with a greater number of cores, assuming the MIMD sort scales with the number of cores and is not limited by memory bandwidth or other architectural constraints, the heterogeneous algorithm may also perform better relative to the homogeneous algorithm. An improved parallel MIMD sorting algorithm would also affect the relative performance.

R-tree

Our R-tree for testing is implemented in C++11 and CUDA, as aforementioned. As with our linear quadtree analysis, for our R-tree homogeneous GPU comparison algorithm we use the CUB library [Nvidia, 2014] which implements a GPU radix sort as presented by

Satish et al. [2009b].

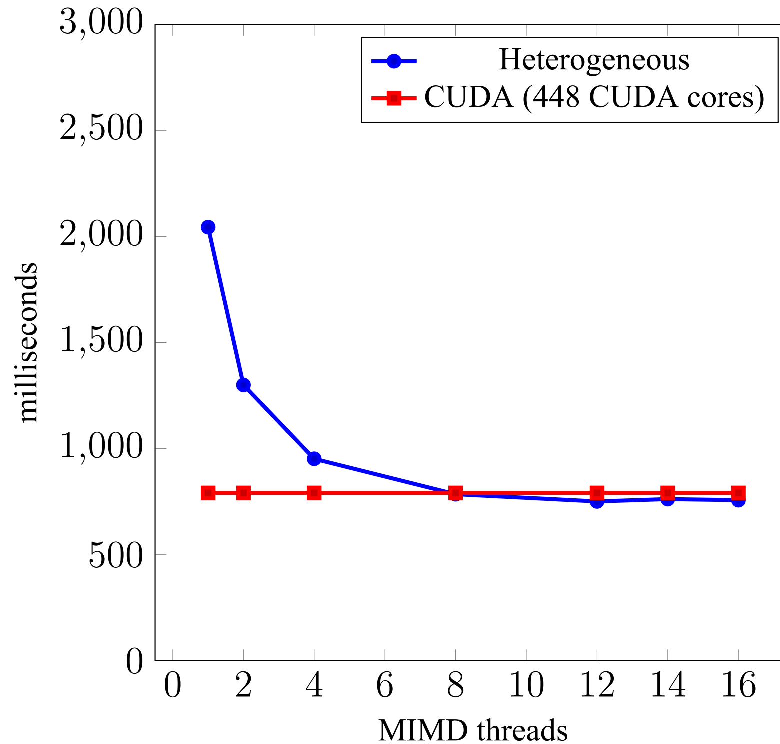

Figure 10: R-tree for 1–12 MIMD threads, with 10,000,000 points

Our testing results are plotted in Figure 10 and Figure 11. We see that our heterogeneous algorithm is approximately 105% as fast as the homogeneous GPU algorithm (751 versus 791 milliseconds with 12 threads).

Once again, we note that values this close suggest architectural changes would significantly impact the results. A faster GPU or slower CPU with fewer MIMD cores might easily make the homogeneous algorithm faster.

We note that, as with the linear quadtree results, the performance of the homogeneous CUDA algorithm in Figure 10 is not affected by the number of MIMD threads.

We also note that we expect the relative performance of the heterogeneous R-tree construction to exceed that of the quadtree, since the R-tree requires multiple passes by the

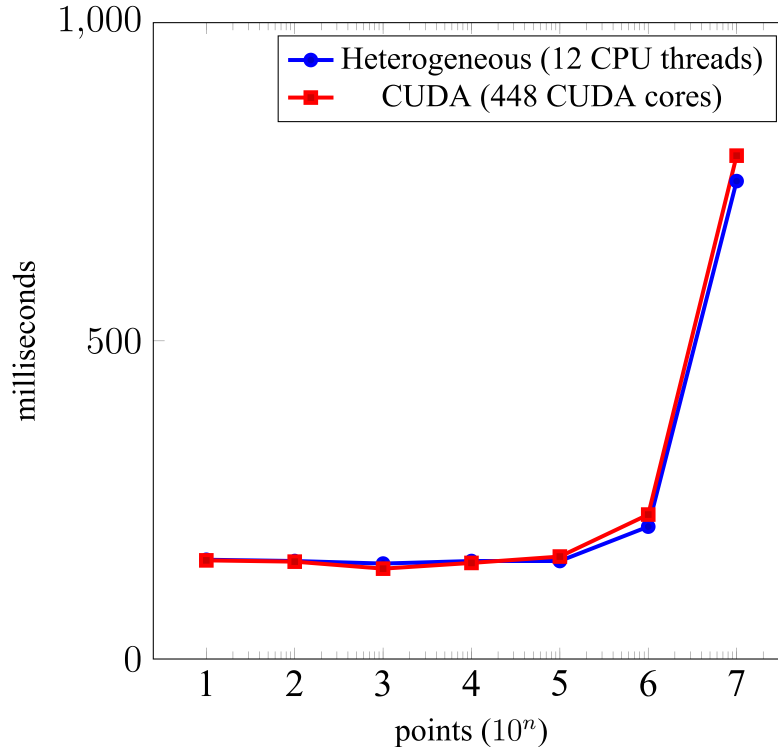

Figure 11: R-tree for 10–10,000,000 points, with 12 threads

vector processor, for each level of the tree.

Linear *k*-d tree

Our linear *k*-d tree for testing is implemented in C++11 and CUDA, as above. MIMD parallelism is implemented via the Intel Threading Building Blocks library [Kim and Voss, 2011].

Our linear *k*-d tree does not have a corresponding vector (GPU) algorithm. However, it is relatively easy to modify to create an MIMD algorithm. We must simply perform the morton code calculations via the CPU rather than the GPU.

Thus, we have implemented a modified MIMD algorithm for performance comparisons. The MIMD morton code calculation is performed on the MIMD CPU with a high degree of parallelism using Intel Threading Building Blocks, and as previously noted, with

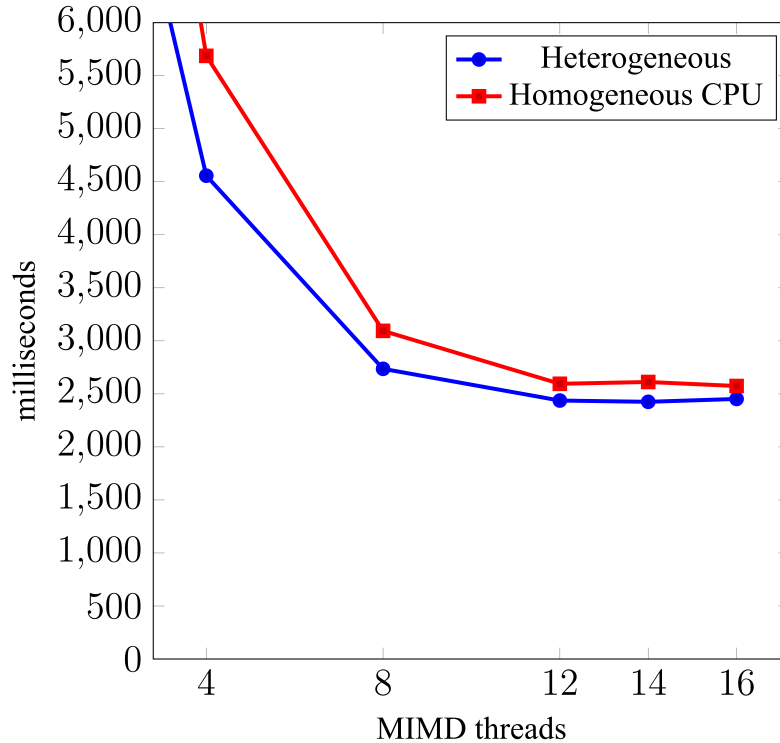

Figure 12: linear *k*-d tree for 1–12 MIMD threads, with 10,000,000 points

x86 SSE vector extensions which are automatically generated by the compiler.

Our testing results are plotted in Figure 12 and Figure 13. With an optimal number of threads for the architecture, we see our heterogeneous algorithm is 108% faster than the MIMD algorithm (2425 versus 2612 milliseconds). However, since we are comparing heterogeneous to homogeneous MIMD, we see improved relative heterogeneous performance with fewer threads. The heterogeneous algorithm is 132% faster with 2 threads (8095 versus 10686 milliseconds) and 136% faster with 1 thread (14786 versus 20244 milliseconds).

We also see that as the number of points approaches zero in Figure 13 the homogeneous CPU algorithm approaches 1 millisecond, while the heterogeneous algorithm approaches approximately 140 milliseconds. Hence, 140 milliseconds appears to be the overhead of GPU execution on our architecture with our application.

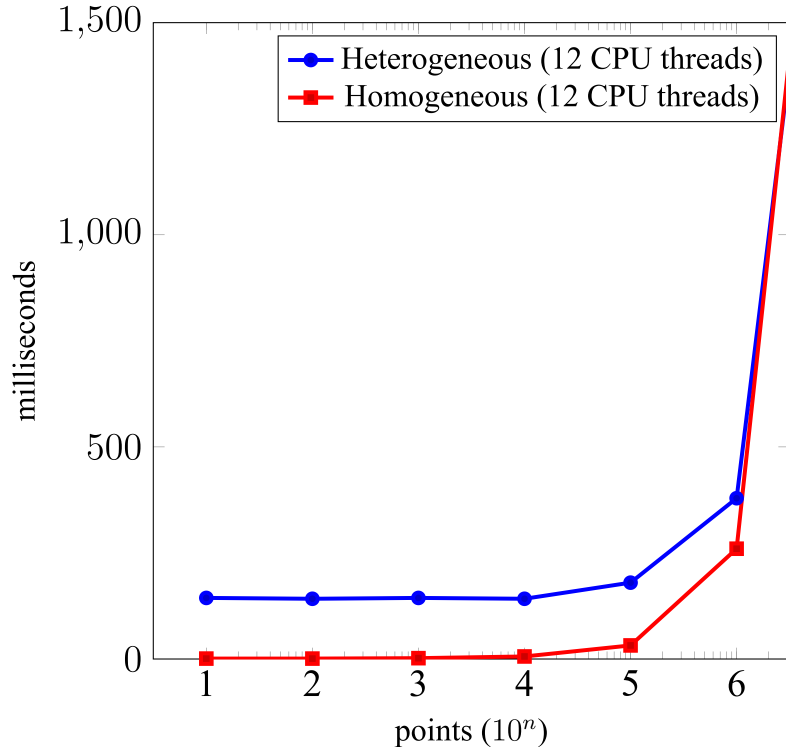

Figure 13: linear *k*-d for 10–10,000,000 points, with 12 threads

Since we are comparing our heterogeneous algorithm with a MIMD algorithm, a directly faster or slower MIMD CPU may affect the ratio of relative performance, but will not affect which algorithm is faster. However, a CPU with better vector extensions may affect the relative performance. Likewise, a slower GPU could also result in a relatively slower heterogeneous algorithm.

Pipelining

As mentioned in Section III, our heterogeneous construction algorithms may be pipelined when constructing multiple trees. We present here the results of pipelining, compared with unpipelined heterogeneous and homogeneous construction algorithms.

We see that in all cases, as expected, pipelined heterogeneous construction alogrithms are faster than unpipelined heterogeneous construction.

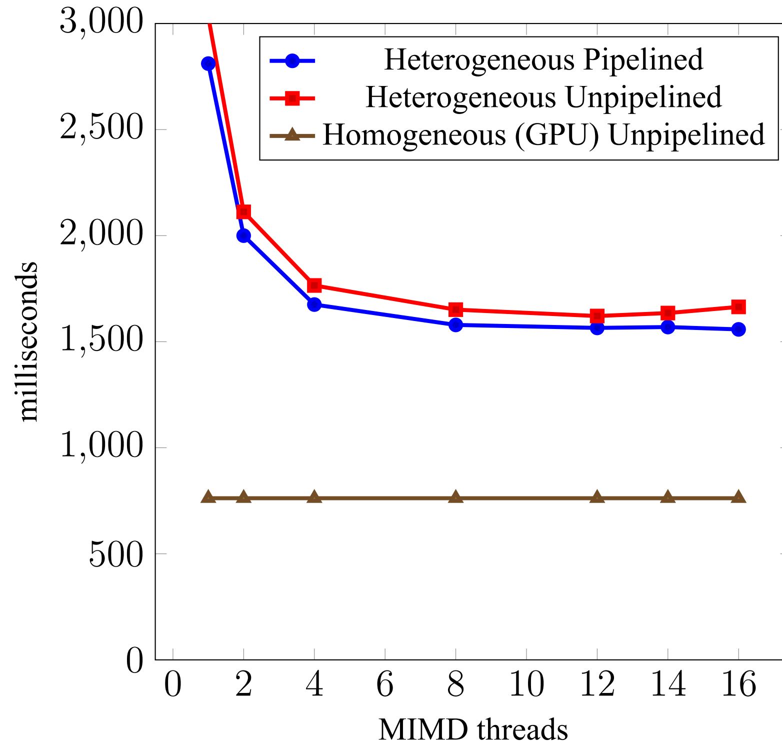

Figure 14: pipelined linear quadtree for 10 trees, using 1–16 MIMD threads, with 10,000,000 points

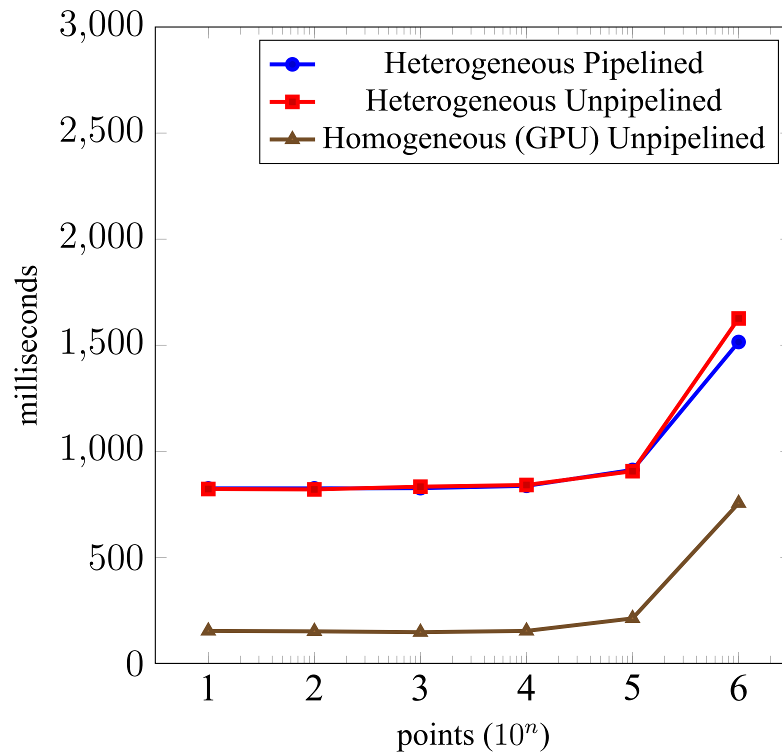

Figure 15: pipelined linear quadtree for 10 trees, with 10–10,000,000 points, with 12 threads

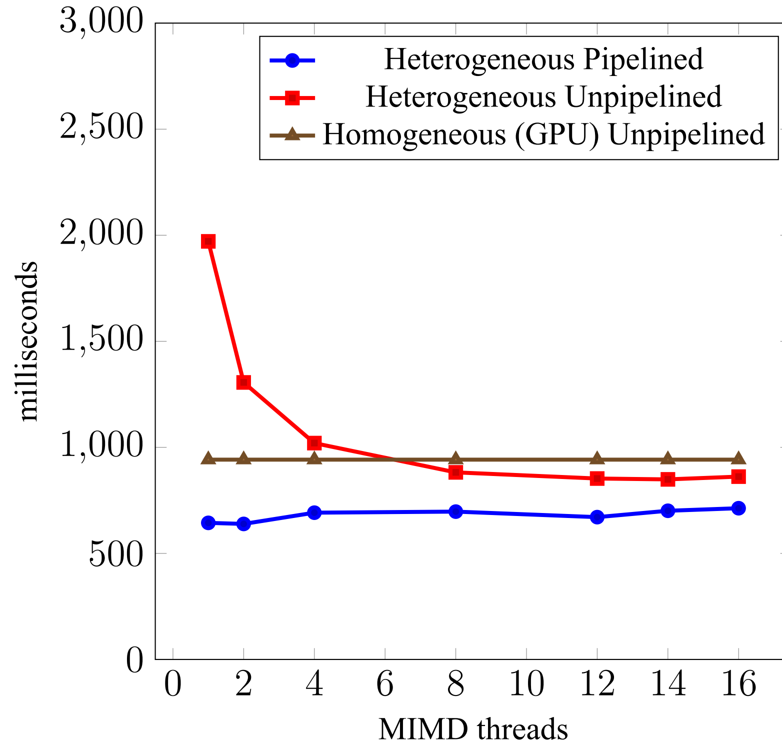

Figure 16: pipelined R-tree for 10 trees, with 10,000,000 points, using 1–16 MIMD threads

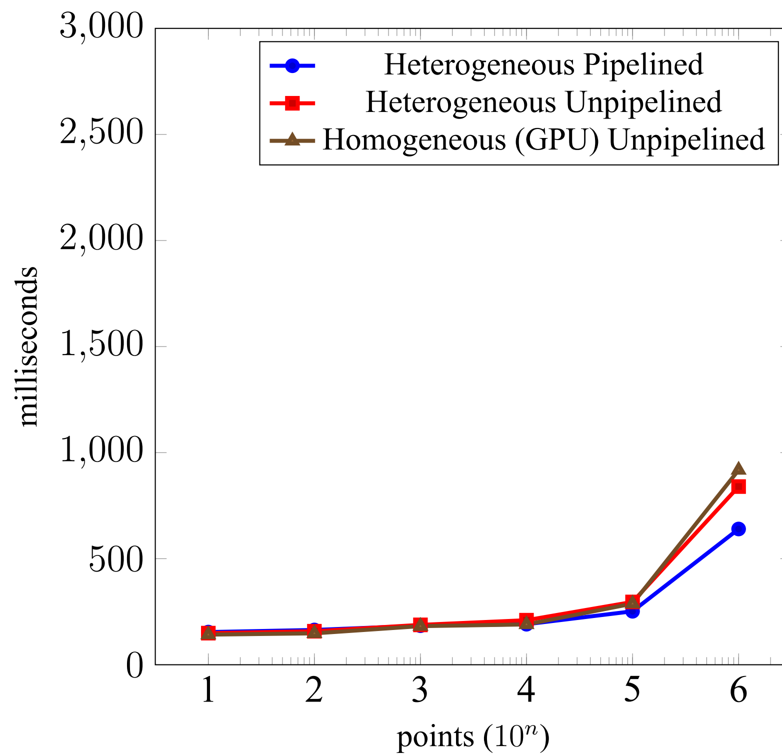

Figure 17: pipelined R-tree for 10 trees, with 10–10,000,000 points, using 12 threads

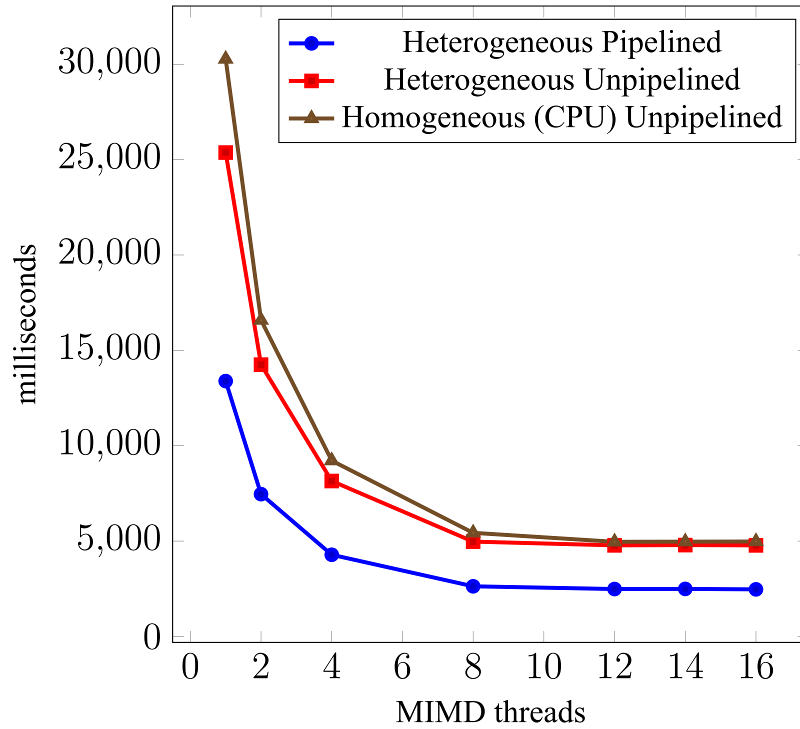

Figure 18: pipelined linear *k*-d tree for 10 trees, with 10,000,000 points, using 1–16 MIMD threads

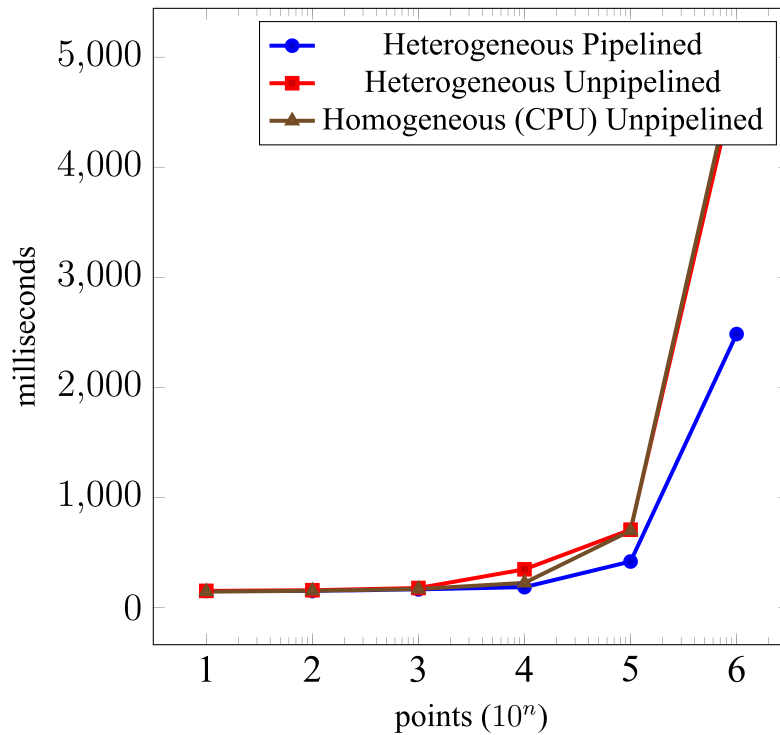

Figure 19: pipelined linear *k*-d tree for 10 trees, with 10–10,000,000 points, using 12 threads

We also see that pipelined heterogeneous algorithms widen the performance gap between homogeneous algorithms in the event the heterogeneous algorithm is faster for a single tree construction, and that pipelined heterogeneous algorithms may narrow the performance gap in the event the single heterogeneous construction algorithm is slower than the homogeneous algorithm.

We observe that the heterogeneous pipelined results in Figure 17 is almost linear in time, even with 1 and 2 threads. We hypothesize this is due to the inherent MIMD parallelization with pipelining. That is, even when only 1 thread is used for the MIMD sorting, the pipeline uses, of necessity, another CPU thread to initiate the GPU kernel, while the MIMD sorting thread may continue with another tree. This is only a hypothesis; performing a detailed analysis to determine why the pipelined construction does not slow as the unpipelined construction does is outside the scope of our research.

Summary

We have seen that our heterogeneous linear quadtree construction algorithm executes within 90% of the performance of the homogeneous SIMD (GPU) algorithm, and that our heterogeneous R-tree construction algorithm executes approximately 105% as fast as the homogeneous SIMD algorithm.

We have also seen that our novel heterogeneous linear *k*-d tree construction algorithm performs within the same order of magnitude as the linear quadtree and R-tree construction algorithms, and that it outperforms a homogeneous MIMD algorithm as expected.

Once again, We note that all our heterogeneous algorithms are heavily dependent on sorting. Hence, a faster MIMD sorting algorithm, or faster MIMD architecture, will

significantly impact their performance.

We note that even inferior heterogeneous performance may be desirable for certain environments. For example, in an environment wherein the CPU is underutilized and the GPU is fully utilized, a heterogeneous algorithm may be faster overall. Similarly, since the GPU sort is inefficient, in an environment where spatial tree construction is a significant percentage of the overall system's computation, it may save power and therefore costs to use a heterogeneous algorithm. (Additional power consumption experiments would be required to verify this conjecture, which are outside the scope of our research.)

CHAPTER V

FUTURE WORK

GPU Optimizations

Our algorithms do not consider GPU architecture, but rather are purely theoretical in their vector implementation. It would be valuable to research performance improvements to these algorithms which consider architecture, and optimize for modern AMD and NVidia architectures, considering block sizes, local memory, and other architectural considerations.

As our GPU sorting uses existing libraries, we do not expect these optimizations to affect whether our heterogeneous algorithms are comparable in performance, although they may affect the ratio of heterogeneous versus homogeneous performance. Regardless, architectural optimizations would be valuable, if only for practical performance improvements.

***k*-d tree Homogeneous Granularity**

Our heterogeneous *k*-d tree split point and sort calculation is entirely MIMD, while the subsequent morton code calculation is entirely SIMD. While this may be ideal for certain scenarios – for example, for pipelining – it would be ideal if the split-point and sort could be further decomposed. For example, could the split-point calculation be vectorized in parallel with the partition? Further decomposition might increase MIMD–SIMD parallelism, and increase performance of the overall algorithm.

It may also be possible to parallelize the GPU morton code calculation with the sort.

For example, if there exist $10 \cdot n$ points and n vector processing elements, if the parallel split-point is fully calculated for n points which are sorted in their final locations, could a single GPU pass be made in parallel with the ongoing sort? Is there an efficient way to determine which points are sorted? Is there a way to prioritize the parallel sort so that points are sorted depth-first, and thus more points are fully sorted before moving on to other points?

3-Dimensional Implementations

Arguably the majority of spatial partitioning applications involve 3-dimensional data. While our heterogeneous research generalizes, it would nevertheless be valuable to implement 3-dimensional versions of our heterogeneous construction algorithms, and to provide benchmarks comparing their performance with other existing 3-dimensional spatial tree construction algorithms.

Such implementations and performance analyses would be of great practical value to research and practical applications such as ray tracing and collision detection.

Investigation of *k*-d tree Splitting Algorithms

For our performance analyses, we used a simple median heuristic splitting algorithm. However, for practical applications, more intelligent splitting algorithms are generally preferred, such as the surface-area heuristic algorithm. It would be valuable to implement various splitting algorithms with our heterogeneous linear *k*-d tree construction algorithm, in order to test their respective performance compared with their other advantages.

Investigation of the Effect of Memory Bandwidth on Parallel Sorting

We note that the parallel scalability of our heterogeneous algorithms relative to their analogous homogeneous algorithms is entirely dependent on the MIMD CPU sort relative to the SIMD/SIMT GPU sort. The order of growth of parallel MIMD sorting algorithms scales well with the number of cores [McCool et al., 2012, p. 304]. However, in practice, scalability is often constrained by memory bandwidth and other architectural constraints.

Our spatial trees in particular are very memory-intensive (we were limited to testing 10,000,000 points due to GPU memory constraints). We observe that our performance results in Figure 8, Figure 10, and Figure 12 all see performance levelling off around 8 threads. Recall that our testing platform has 12 MIMD cores. This suggests that architectural constraints are becoming an issue even before 12 cores.

We hypothesize that the architectural constraint is memory bandwidth. Our algorithms are very memory-intensive, and without further evidence, memory speed and latency seems like the most likely cause, considering that the sorting algorithms theoretically scale to several orders of magnitude more processors than we are using [McCool et al., 2012, p. 304]. However, accurately measuring memory bandwidth usage can be difficult, and is outside the scope of our research.

It would certainly be valuable for future research to confirm or refute this hypothesis, and moreover to confirm the reason MIMD sorting is not scaling with our algorithms, and what would be necessary architecturally or algorithmically to scale to manycore architectures.

CHAPTER VI

CONCLUSION

We have seen that homogeneous GPU algorithms for spatial trees which involve discrete sorting and location code generation (or bounding box generation) may be easily converted to heterogeneous algorithms. We have seen that these heterogeneous algorithms perform comparably to homogeneous algorithms on our 12-core testing architecture.

We have also seen a novel heterogeneous linear *k*-d tree construction algorithm based around a parallel quicksort, which allows for the use of any desirable splitting hyperplane function, and which performs within the same order of magnitude as our observed quadtree and R-tree algorithms on our testing architecture. We also observe that the heterogeneous algorithm is primarily MIMD in nature, while using the SIMD GPU for the vectorizable task of calculating Morton codes. Hence, our linear *k*-d tree construction algorithm is also easily adaptable to a homogeneous parallel MIMD CPU algorithm.

We have unfortunately seen evidence suggesting our heterogeneous algorithms, specifically their usage of MIMD sorting, may not scale to manycore architectures (Chapter V). The sorting algorithms scale in theory, and thus, manycore scalability may be achievable with architectural or algorithmic improvements.

On a heterogeneous architecture, heterogeneous algorithms allow for fully using the architecture, as well as for balancing usage to reduce the load of an overused processor type (SIMD, MIMD), and to increase the load of an underused processor type. Heterogeneous

algorithms also allow for using the optimal processor type for each procedure (that is, using a vector processor [GPU] for procedures which are well-vectorizable, and using a parallel MIMD processor [CPU] for procedures which are poorly vectorizable). Such ideal usage for respective processors may lead to greater efficiency and performance of both individual procedures and algorithms, as well as overall systems.

REFERENCES

Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4):13–27, July 2008.

Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, February 2008. ACM.

John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460, New York, NY, June 2010. ACM.

Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10, Rome, May 2009a. IEEE.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2009.

Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming*. Elsevier, Waltham, MA, 2012.

Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 351–362, New York, NY, June 2010. ACM.

Hanan Samet. The quadtree and replated hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, March 1974.

Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, New York, NY, June 1984. ACM.

D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of the Fifth International conference on Data Engineering*, pages 606–615, Los Angeles, CA, February 1989. IEEE.

Hans-Peter Kriegel, Matthias Renz, and Kunath Peter. R-tree. In Shashi Shekhar and Hui Xiong, editors, *Encyclopedia of GIS*, pages 987–992. Springer, New York, NY, 2008.

Lijuan Luo, Martin D.F. Wong, and Lance Leong. Parallel implementation of R-trees on the GPU. In *17th Asia and South Pacific Design Automation Conference*, pages 353–358, Sidney, NSW, January 2012. IEEE.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer: International Journal of Computer Graphics*, 6(3):153–166, 1990.

Min Feng. Efficiency of k-d tree ray-traversal algorithms. In *Fifth International Conference on Computational and Information Sciences*, pages 984–987, Shiyang, June 2013. IEEE.

Vlastimil Havran and Jiri Bittner. On improving kd-trees for ray shooting. In *In Proc. of WSCG 2002 Conference*, pages 209–217, 2002.

Mohammed Otair. Approximate k-nearest neighbour based spatial clustering using k-d tree. *International journal of database management systems*, 5(1):97–108, 2013.

Irene Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

Y. M. Kim and S. B. Park. New method for representing linear quadtree. *Electronics Letters*, 25(2):137–139, January 1989.

G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, 1966.

C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009.

Philippas Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN Enterprise 10000. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 372–381, Genova, Italy, February 2003. IEEE.

Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

Nvidia. NVIDIA's next generation CUDA compute architecture. Technical report, Nvidia, 2009.

The ANSI C Standard. ISO, 1999.

Programming Languages — C++. The C++ Standards Committee, 2003.

Fortran 90 Standard. J3, 1991.

Using GNU Fortran. FSF, 2007.

Tom Altman and Yoshihide Igarashi. Roughly sorting: sequential and parallel approach. *Journal of Information Processing*, 12(2):154–158, January 1989.

Tom Altman and Bogdan S. Chlebus. Sorting roughly sorted sequences in parallel. *Information Processing Letters*, 33(6):297–300, February 1990.

Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Pearson, Boston, MA, 2010.

Wooyoung Kim and Michael Voss. Multicore desktop programming with Intel Threading Building Blocks. *IEEE Software*, 28(1):23–31, 2011.

AMD64 Architecture Programmer's Manual Volume 3. AMD, 2005.

Using the GNU Compiler Collection. FSF, 2013.

Nvidia. CUB. https://nvlabs.github.io/cub/, 2014. version 1.3.2.

Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10, Rome, May 2009b. IEEE.