GPU-based dual-tree nearest neighbor search

Ryan R. Curtin and Chad D. Kersey

December 15, 2015

1 Introduction

Nearest neighbor search is important, for some reason. Either way, it's what a bunch of people spend time thinking about. Some people who think about it think that dual-tree algorithms [**?**] provide a good solution, especially when the dimensionality of the data is low. More recently, a formalized dual-tree algorithm has been presented [**?**], which generalizes previous dual-tree nearest neighbor search approaches [**?**, **?**, **?**].

Formally, the problem of nearest neighbor search is as follows: given a query set $S_q \in \mathcal{R}^d$ and a reference set $S_r \in \mathcal{R}^d$, find

$$\text{argmax}_{p_r \in S_r} d(p_q, p_r) \tag{1}$$

for all points $p_q \in S_q$ and for some distance metric $d(\cdot, \cdot)$. A dual-tree approach to this problem can be understood roughly as building a tree (the *query tree*) on the query set S_q, building a tree (the *reference tree*) on the reference set S_r, and then traversing both trees simultaneously, pruning away branches of the dual recursion where no descendant points of the reference node can possibly be nearest neighbors of the query node. In some sense, this strategy is a generalization of the more well-known and common *single-tree* approaches [**?**, **?**, **?**].

2 Proposed algorithm

GPUs have long been considered unsuitable for tree-based recursive algorithms because of their SIMD structure. However, dual-tree algorithms present a fascinating candidate for parallelization because of the huge fanout of each level of the recursion. While a typical single-tree *kd*-tree nearest neighbor search algorithm has a fanout of two at each level, the dual-tree *kd*-tree traversal will have a fanout of four. This means that very quickly, there will be more combinations of nodes to visit than processors. And that means that we can effectively utilize the SIMD nature of GPUs with few wasted cycles[1].

Roughly speaking, the dual-tree nearest neighbor search algorithm can be understood as the following few steps. (Here, it is not written recursively, as it usually is.)

[1] I hope.

1. Obtain an unpruned node combination $(\mathscr{N}_q, \mathscr{N}_r)$ from the set of still-to-be-visited combinations C.
2. Determine if that node combination can be pruned.
3. If the node can be pruned, return to step 1.
4. Calculate base cases between all query points held in $\mathscr{N}_q$ and all reference points held in $\mathscr{N}_r$.[2]
5. Update the bounds of $\mathscr{N}_q$ for pruning.
6. Add each child combination of $(\mathscr{N}_q, \mathscr{N}_r)$ to C.[3]
7. Return to step 1.

There are numerous strategies for how to select the next unpruned node combination [**?**, **?**, **?**]. With GPUs, though, we need not select only one node combination—we can select many at once! This allows us to outline a basic algorithm (Algorithm 2).

[2]Here we mean the points held in the node, not the descendant points. Otherwise, pruning would save us nothing since we would do all the work at the first level!

[3]For binary trees like the *kd*-tree, this is four combinations.

[6]This probably means that each node in our tree should hold an exact (or maximum) number of points, instead of the typical mlpack *kd*-tree which only holds points in the leaves.

[6]If better candidates are found, they should be inserted into the appropriate rows of N and D using a simple insertion sort procedure that can take exactly $O(k)$ time and may be done in parallel. But how do we guarantee that all threads are inserting? Shit, I didn't think about that part, or the thread safety or anything...

[6]Chad, this is where I need help. I have basic ideas, but not more than that.

Algorithm 1 Basic GPU dual-tree k-nearest neighbor algorithm outline.

Input: query tree $\mathscr{T}_q \in \mathcal{R}^{m \times d}$, reference tree $\mathscr{T}_r \in \mathcal{R}^{n \times d}$.
Output: nearest neighbors $N \in \mathcal{R}^{n \times k}$, distances $D \in \mathcal{R}^{m \times k}$.

Initialize D, filling with ∞.
Initialize combination+score list $C \leftarrow \{(\text{root}(\mathscr{T}_q), \text{root}(\mathscr{T}_r), 0)\}$.
$p \leftarrow$ number of GPU threads

{Main work loop.}
while $|C| < p$ **do**
  for all thread i **do**
    {Get a combination.}
    $c \leftarrow C[i]$

    Initialize resulting score array $s[4]$.

    Perform base case between each point $p_q \in \mathscr{N}_q$ and each point $p_r \in \mathscr{N}_r$.[45]

    $s[0] =$ score of $(\mathscr{N}_{ql}, \mathscr{N}_{rl})$.
    $s[1] =$ score of $(\mathscr{N}_{ql}, \mathscr{N}_{rr})$.
    $s[2] =$ score of $(\mathscr{N}_{qr}, \mathscr{N}_{rl})$.
    $s[3] =$ score of $(\mathscr{N}_{qr}, \mathscr{N}_{rr})$.

    {Insert results into end of $|C|$. Each thread is inserting four things...}
    $l \leftarrow |C|$
    $C[l + 4i] = (\mathscr{N}_{ql}, \mathscr{N}_{rl}, s[0])$.
    $C[l + 4i + 1] = (\mathscr{N}_{ql}, \mathscr{N}_{rr}, s[1])$.
    $C[l + 4i + 2] = (\mathscr{N}_{qr}, \mathscr{N}_{rl}, s[2])$.
    $C[l + 4i + 3] = (\mathscr{N}_{qr}, \mathscr{N}_{rr}, s[3])$.
  end for

  Sort C so that the p lowest scores are in the first p slots (in any order). Entries where $p = \infty$ should be truncated from the list.[6]
end while

return N, D