Parallel Computing Project

May 13, 2019

1 Introduction

Many algorithms use trees as their primary data structures. For instance, many search algorithms use binary search trees. Trees are also used in structures like heaps, which are then used to implement priority queues. Some graph search algorithms also exist, such as Breadth-First Search (BFS) and Depth-First Search (DFS). The goal of this project was to find a way to implement trees as a data structure and apply them to parallel algorithms and other CUDA programs. Binary trees are not something that one would initially associate with CUDA, or using a parallel algorithm, so I thought it would be a good idea of trying to run an algorithm for it.

Here, I will describe one of the ways that I used CUDA to implement binary trees. Also, I will be describing some of the challenges and problems that we run into in implementing binary trees in CUDA and parallel coding.

2 Serial Implementation

The serial implementation of a binary tree is quite simple and was something that we learned back when we were learning C. If you want to create a binary tree in the CPU, we would create something that looks like this:

```
struct node
{
  int data;
  struct node *left;
  struct node *right;
};
```

In the CPU, a tree would be represented by a pointer to the topmost node (the root) in a tree. If a tree is empty, then the value of the root is NULL. A tree node will contain the following parts: the data, a pointer to the left child, and a pointer to the right child. If you want a tree with more than two children (a n-ary tree instead of a binary tree), then you would have to include more nodes in the structure to represent more children.

It is easy to see how one would execute all of the simple binary tree functions using a structure like this one. Leaves would be categorized by the nodes that have NULL for both of their children nodes. If you want to add a node to this tree, you would have to traverse the tree to find the node you want to add too and then modify the child node as necessary. This is nothing new, and is something that we have done various times in other classes.

Below is a way to create some simple trees using the structure that we have described above:

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

```
struct node* newNode(int data)  {
  struct node* node = (struct node*)malloc(sizeof(struct node));

  node->data = data;

  node->left = NULL;
  node->right = NULL;
  return(node);
}


int main() {
  struct node *root = newNode(1);
  /* 1 is the root of the tree */

  root->left        = newNode(2);
  root->right       = newNode(3);
  /* 2 and 3 become left and right children of 1 */

  root->left->left  = newNode(4);
  /* 4 becomes left child of 2 */
}
```

3 Challenges with Parallel Implementation

Now that we have an idea of what binary trees would look like in a serial implementation, we now want to find a way to implement binary trees in a parallel implementation. The most obvious way that one might approach this is to assign the nodes of our potential tree to threads. The various threads would then act independently and hopefully create something that resembles a binary tree. However, as we will soon see, there will be a few problems with creating a binary tree with this method.

3.1 Divergence

The biggest problem of trying to use threads as nodes and then creating a tree that way is the problem of divergence. Divergence is a measure of whether nearby threads are doing the same thing or different things. There are two main types of divergence: execution divergence which means that the threads are executing different code or making different control flow decisions, or data divergence which means that they are reading or writing disparate locations in memory.

This is not a problem in creating trees in the CPU. Since, we are individually creating each node and placing them in the structure of the tree, we don't run into issues of wanting to put multiple tree nodes in a certain spot. In other words, trees on the CPU are dynamically allocated. However, this is a bit of a problem with the GPU. The most obvious problem with a recursive implementation is high execution divergence. Each thread will be running, trying to get their value to a node in a tree. However, the decision of whether to skip a given node or recurse to its children is made independently by each thread, and there is nothing to guarantee that nearby threads will remain in sync once they have made different decisions.

4 Parallel Implementation: The Idea

So, how exactly will we run and create binary trees in parallel, or in the GPU? We do not want to be using dynamic allocation on our GPU kernels because of the issues that we stated above. The idea that we want to use is to assign each integer to a thread (like we have described above). However, the order in which the threads will execute will be beyond our control. So, we must be okay with different tree structure ultimately representing the same data.

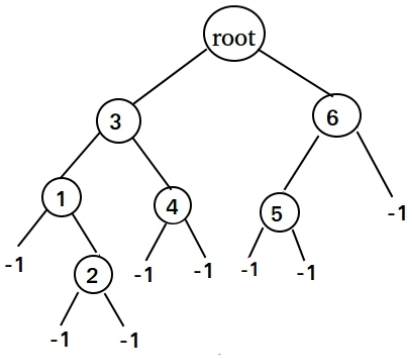
4.1 Node Insertion Algorithm

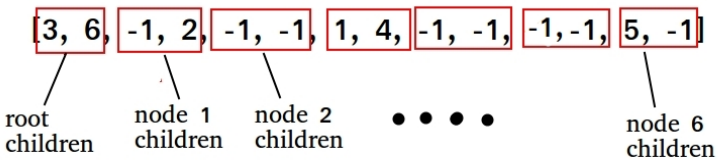
Adding a node to a tree will involve two main steps: 1. Traversing the partially constructed tree until you hit a leaf node, and 2. Adding the new node to the tree. We can do Step 1 in parallel, while we do Step 2 sequentially. So, each thread can traverse the tree in parallel, but the threads must insert their values one at a time. We want to use our known synchronization methods to ensure that our integers are added one at a time.

4.2 The Binary Tree Structure in CUDA

The idea that we will use is that we will only record the children of all of the nodes. Because every node is itself a child node of its parent, this effectively means that we will store every node except the root in our structure. NULL nodes will be represented with a -1. Every pair in the structure will represent a node. The first element in the pair is the left child and the second element is the right child.

Suppose we had the array of integers $C = [5, 6, 3, 4, 2, 1]$. An example of a binary tree structure of this array can be found below:





As we can see, the array representation of the tree will have 14 elements. This is because the array C has 6 nodes, which gives us a total of 12 elements, but we will need 2 more for the children of the root node.

We can implement this idea and structure with to a tree of any number of nodes.

5 Parallel Implementation: The Code

My code will create binary trees using the structure that we have described above. It will use a parallel algorithm and the GPU to insert the nodes into the tree. We will go through the main parts of the code to explain what it is doing.

5.1 The Main

```
int *h_x;
int *d_x;
int *h_root;
int *d_root;
int *h_child;
int *d_child;

h_x = (int*)malloc(n*sizeof(int));
h_root = (int*)malloc(sizeof(int));
h_child = (int*)malloc(2*(n+1)*sizeof(int));
cudaMalloc((void**)&d_root, sizeof(int));
cudaMalloc((void**)&d_x, n*sizeof(int));
cudaMalloc((void**)&d_child, 2*(n+1)*sizeof(int));
cudaMemset(d_child, -1, 2*(n+1)*sizeof(int));
```

Here, we are allocating space for the structure of our tree. n is the number of nodes that we have. h_x is the host array and d_x is the device array.

```
for(int i=0;i<n;i++){
                int j = random() % (n-i);
                int temp = h_x[i];
                h_x[i] = h_x[i+j];
                h_x[i+j] = temp;
        }
        *h_root = h_x[0];

        for(int i=0;i<n;i++){
                printf("%d ", h_x[i]);
        }
        printf("\n");
```

This part of the code randomly shuffles the nodes that we have, representing the fact that we are randomly assigning threads to each node.

The rest of the code is call the kernel code on the inputs that we have created here.

5.2 The Kernel

```
int childPath;
int temp;
offset = 0;
while((bodyIndex + offset) < n){
```

```
        if(newBody){
                newBody = false;

                temp = 0;
                childPath = 0;
                if(x[bodyIndex + offset] > rootValue){
                        childPath = 1;
                }
        }
        int childIndex = child[temp*2 + childPath];

        // traverse tree until we hit leaf node
        while(childIndex >= 0){
                temp = childIndex;
                childPath = 0;
                if(x[bodyIndex + offset] > temp){
                        childPath = 1;
                }

                childIndex = child[2*temp + childPath];
        }

        if(childIndex != -2){
                int locked = temp*2 + childPath;
                if(atomicCAS(&child[locked], childIndex, -2) == childIndex){
                        if(childIndex == -1){
                                child[locked] = x[bodyIndex + offset];
                        }

                        offset += stride;
                        newBody = true;
                }
        }

        __syncthreads();
}
```

This is the main part of our code. The first if-statement is checking each thread and then adding to the tree depending on the root value. The next while-loop places the nodes if the thread encounters a leaf. Finally, the next if-statement accounts for the case when two threads reach the same leaf. The code then locks one of the threads, and the other threads skip the if-statement and then go to *__syncthreads*() and then waits.

6 Further Ideas and Improvements

Clearly, this implementation is not perfect, and there is a lot of room for improvement here. For instance, this implementation works best when it takes elements that are integers from 1 to n. We can potentially look to improvements in making the info of our nodes not be integers, and instead be something else, like strings.

Also, we can see that structure of the tree may be inefficient in some ways as well. For instance, suppose that our tree had an abnormal amount of NULL children nodes. Then, the mode that we constructed here would not be very efficient, and perhaps the structure could use some improvements.

Finally, we notice that binary trees are not usually the type of structures that one might implement in CUDA or with parallel computing. What would be the advantages of using a parallel algorithm in this case, other than some time being saved?

Another thing that we can potentially implement is search and traversal problems. The biggest problem that we might have is to implement these functions in a way that is strictly faster than the serial version on the CPU.

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p475-484. Print

[2] NVIDIA,
https://devblogs.nvidia.com/thinking-parallel-part-ii-tree-traversal-gpu/

[3] NVIDIA,
https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-construction-gpu/