Implementation of Harmonia B+ tree

- Kshitij Deogade (CS17B104)

I am constructing a B+ tree and code is in "harmonia.cu".
Here, nodes are present in a BFS fashion in an array "nodes", and first child of each node is present in
"prefix_sum" array.

As the first children are in prefix_sum, range-query is very simple, as you can just find first node in range,
then just linearly traverse prefix_sum till you reach the last node in range.
Another reason why this is efficient is because it reduces comparison operations by making single warp use multiple
queries.

In "harmonia.cu", you can change the parameters at top to construct different trees with different throughputs.
These parameters are-
1) GS- no_of_threads_for_1_query ;
2) nk- no. of keys in a node (=fanout-1)
3) m- no of data items associated with a key

"harmonia.cu" can insert key and data into node , and also search for a key/data.
The insert happens sequentially in CPU, while the queries happen in parallel on GPU.

Note that in the 2nd code I have submitted- "harmonia_sequential_inserts.cu", range_query has also been implemented.
However, every operation is sequential here.

In "harmonia.cu", as you decrerase GS, it means no_of_threads_for_1_query in a warp is decreasing.
This leads to less comparison operations, meaning lesser warp divergence, meaning greater throughput. However, if too many
queries get into 1 warp, query divergence occurs, so throughput can increase.

I have left debugging messages in code as it is, to help in my own understanding later, too.

How to run:

There are 2 codes - "harmonia.cu" and "harmonia_sequential_inserts.cu".
The first one can be run by simple "nvcc harmonia.cu".
The second one doesn't have any parallel code, so can be run via nvcc as well as via "g++ harmonia_sequential_inserts.cpp"

Observations:-

1) From plot1.png, as no_of_keys with a single node increases, throughput first increases and then decreases. But primary observation
is that the throughput decreases with increase in no_of_keys with a node, for a given GS, no_of_queries_in_warp.

This is as you'll have to make more accesses to the keys array in a node, with all the threads assigned to a given query(=GS).
This is almost equivalent to the flip side of decreasing GS, as noted earlier. So, throughput decreases.

2) From plot2.png, as GS/no_of_threads_for_1_query increases, throughput decreases, showcasing that
indeed, including more queries in a single warp increases the throughput.

3) From plot3.png, as tree size increases, for a given GS,no_of_keys in a node and no_of_queries_in_warp,
so ideally, more nodes have to be traversed for a query, and so, throughput decreases.

Challenges:

1) The prefix_sum array is actually to be implemented in GPU's texture memory, making it fast to be read.
However, I tried with prefix_sum in global array earlier, and didn't have time later to implement on a texture memory.

2) I couldn't directly observe a lowering of throughput by decreasing GS to a very small value. Hence, I used observation 1.
If later, some method to count query divergence/warp divergence as done in paper were to be used, this would be more apparent.

3) Sequential Implementation of insert() was challenging, but it works fine, finally. Integrating this
and parallel queries was quite challenging too, as my code has reached ~1200 lines of code(double than that expected).