Appendix: Parallel Primitives

Although naming conventions might differ slightly under different contexts and software implementations, since our implementation is based on the Thrust library, we next introduce the primitives that we have used in our design using the Thrust terminology.

(1) *Reduce* and *Reduce by key*. *Reduce* is used to simplify a vector/array to a scalar value. For example, reduce([3,2,4])→11. While the summation is frequently used in reductions, Thrust allows using a user defined associative binary function for tailored summation, such as determining the maximum entry or computing bounding boxes of points. *Reduce by key* is a generalization of *Reduce* to key-value pairs based on groups where consecutive keys in the groups are the same. For example, reduce[1,3,3,2],[2,1,3,4])→([1,3,2],[2,4,6]). In this research, *Reduce by key* has been extensively used to compute numbers of points and quadrant that have the same keys based on Morton codes.

(2) *Scan* and *Scan by Key*. The Scan primitive computes the cumulative sum of a vector/array. The Scan primitive can also take a user defined associative binary function. Both the inclusive and exclusive scans are available. For example, exclusive_scan([3,2,4])→([0,3,5]) while inclusive_scan ([3,2,4])→([3,5,9]). Similarly, *Scan by Key* works on consecutive key groups instead of a whole vector/array. In this research, *Scan by Key* is extensively used to compute the positions of entries in a vector after applying *Reduce by key* which outputs numbers of entries with same keys.

(3) *Copy* and *Copy_if*. The functionality of the two primitives is self-evident. In this research, we use *Copy* to move groups of entries from one location to another, mostly within a same vector. The *Copy_if* primitive is mostly used for identifying points and keys (point quadrants) that satisfy certain criteria and output the identified entries to a new vector for further processing.

(3) *Transform*. The basic form of *Transform* applies a unary function to each entry of an input sequence and stores the result in the corresponding position in an output sequence. Transform is more general than Copy as it allows a user defined operation to be applied to entries rather than simply copying. Similar to *Copy_if*, there is also a *Transform_if* primitive which is essentially the combination of *Transform* and *Copy_if*. The combination usually results better performance. In this research, *Transform* has been extensively used to convert points into Morton codes.

(4) *Gather* and *Scatter*. *Gather* copies elements from a source array into a destination range according to a map and *Scatter* copies elements from a source range into an output array according to a map. For example, Gather([3,0,2],[4,7,8,12,15])→([12,4,8]) and Scatter([3,0,2],[12,4,8],[*,*,*,*,*,*])→([4,*,8,*12,*]). Note * values are those unchanged in the third input vector. In this research, we have used the combination of *Gather* and *Scatter* to locate individual points fall within quadrants that have fewer than K points so that they can be moved to proper locations.

(5) *Sort* and *Sort by Key*. *Sort* is probably among the most popular primitives in parallel libraries. In fact, our design aims at utilizing the power of parallel sorting on GPGPUs to speed up generating point quadrants. The current implementations of the sorting algorithms in Thrust are based on a combined radix sort and merge sort which has been proven to be memory bandwidth friendly and practically efficient. Our design facilitates reducing memory traffic and further improves sorting efficiency in the following sense. First, rather than sorting coordinates directly, we sort Z-order transformed Morton codes. The transformation preserves spatial adjacency and requires less data movement. Second, we sort the increasingly longer Morton codes level-by-level and the data movement overheads are amortized among multiple steps since keys and points with the same values do not need to be moved during sorting. Third, keys and points that are identified as those that should be associated with identified quadrants do not need to be sorted any more in the subsequent levels. The last two points have been quantified in Section 4.3. We are also in the process of combining our application semantics and Thrust sorting code to develop a tailored sort primitive implementation to further improve the overall efficacy. This is important as the sort costs are more than half of the end-to-end computing costs in generating point quadrants (see details in Section 4.3).

(6) *Remove_if*. *Remove_if* marks elements in a vector that satisfy a predicate and compact the unmarked elements to the beginning of the vector so that the marked elements are removed. For example, *Remove_if* ([1, 4, 2, 8, 5, 7,is_even])→[1,5,7]. Remove_if is functionally equivalent to *Copy_if* but it allows in-place operation in the Thrust library. In contrast, using *Copy_if* would require a temporary vector and *Remove_if* is more convenient in this case.

(7) *Unique*. *Unique* moves unique elements to the front of a range for each group of consecutive elements. For example, unique([1, 3, 3, 3, 2, 2, 1])→[1,3,2,1]. *Unique* needs to work with *sort* to obtain globally unique elements.

(8) *Binary Search* and *lower_bound*. *Binary Search* searches for values in sorted ranges and needs to work with *sort* for correct searching. When *Binary Search* tells whether the searching elements are in the vector being searched, *lower_bound* tells the position of the searched element. Thrust has provided a vertorized form of both *Binary Search* and *lower_bound.* There is a shuttle implementation issue that requires using *Binary Search* and *lower_bound* together. For example, assuming A=[0,2,5,7,8] and B=[0,1,2,3,8,9], when searching all elements of B in A, conceptually the results should be [0,-1,1,-1,4,-1] where -1 indicates not found. However, *lower_bound* (A,B)→[0,**1**,1,**2**,4,**5**] where the numbers indicate the index of first position where the search value could be inserted without violating the ordering. The numbers are the same as the matching positions of elements if there are matches but meaningless if the searching elements are not in the vector being searched. Fortunately, binary_search(A,B)→[T,F,T,F,T,F] which serves the exact purpose. As such, *Binary Search* and *lower_bound* need to be used together.