Binary Search Based Parallel Merge Sort Kernel

**Theoretical Basis:**

Given the sorted sequence A and B, we want to compute the sorted sequence C = merge(A, B).

* For an element *ai* in A, we need only compute *rank(ai, C)*, which is the position of element ai in the merged sequence C.
* Because both A and B are sorted, we know that *rank(ai , C) = i + rank(ai, B),* where *rank(ai, B)* is simply the count of elements ***bj*** in B with ***bj < ai*** and which we compute efficiently using **binary search**.
  + Simply put, position of ***ai*** in result sequence C, is determined by its own index ***i***, (meaning how many number in A are less than ai) and rank(ai , B) meaning how many numbers in B is less than ai.
* Elements of B can obviously be treated in the same way.
* **Each thread in a thread block can compute rank(ai , C) and rank(bj , C) independently, thus could be parallelized.**

**Carry Out:**

* Therefore, we can efficiently merge these two sequences by having each thread of the block compute the output rank of its corresponding elements in A and B,
  + subsequently writing those elements to the correct position.
* Since this can be done in on-chip memory, it will be very efficient.

// Bottom-level merge sort (binary search-based), this is

template<uint sortDir> \_\_global\_\_ void mergeSortSharedKernel(

uint \*d\_DstKey,

uint \*d\_DstVal,

uint \*d\_SrcKey,

uint \*d\_SrcVal,

uint arrayLength //actual length for this block to process in the input array.

)

{

\_\_shared\_\_ uint s\_key[SHARED\_SIZE\_LIMIT]; //SHARED\_SIZE\_LIMIT: number of elements a thread block processes

\_\_shared\_\_ uint s\_val[SHARED\_SIZE\_LIMIT]; // Actually, the number of threads per block is SHARED\_SIZE\_LIMIT/2

d\_SrcKey += blockIdx.x \* SHARED\_SIZE\_LIMIT + threadIdx.x;

d\_SrcVal += blockIdx.x \* SHARED\_SIZE\_LIMIT + threadIdx.x;

d\_DstKey += blockIdx.x \* SHARED\_SIZE\_LIMIT + threadIdx.x;

d\_DstVal += blockIdx.x \* SHARED\_SIZE\_LIMIT + threadIdx.x;

s\_key[threadIdx.x + 0] = d\_SrcKey[ 0];

s\_val[threadIdx.x + 0] = d\_SrcVal[ 0];

s\_key[threadIdx.x + (SHARED\_SIZE\_LIMIT / 2)] = d\_SrcKey[(SHARED\_SIZE\_LIMIT / 2)];

s\_val[threadIdx.x + (SHARED\_SIZE\_LIMIT / 2)] = d\_SrcVal[(SHARED\_SIZE\_LIMIT / 2)];

for (uint stride = 1; stride < arrayLength; stride <<= 1)

{

uint lPos = threadIdx.x & (stride - 1);

uint \*baseKey = s\_key + 2 \* (threadIdx.x - lPos);

uint \*baseVal = s\_val + 2 \* (threadIdx.x - lPos);

\_\_syncthreads();

uint keyA = baseKey[lPos + 0];

uint valA = baseVal[lPos + 0];

uint keyB = baseKey[lPos + stride];

uint valB = baseVal[lPos + stride];

uint posA = binarySearchExclusive<sortDir>(keyA, baseKey + stride, stride, stride) + lPos;

uint posB = binarySearchInclusive<sortDir> (keyB, baseKey + 0, stride, stride) + lPos;

//the inclusive and exclusion binary search make the sort **stable**, if there is a duplicate of Bj in A, ( A is on the left)

// This make sure the duplicate in A will output earlier than Bj, thus using inclusive search for rank(Bj, A), which

// guarantees that duplicates in the left go to the left in the resultant sorted array.

// How about a duplicate key in the same section A? how does the code deal with that?

// This is addressed by using the data mapping where each thread computes rank for a number at a unique

// position, that is, the lPos is unique for these duplicates. ( Also making sure left duplicates goes to left )

\_\_syncthreads();

baseKey[posA] = keyA;

baseVal[posA] = valA;

baseKey[posB] = keyB;

baseVal[posB] = valB;

}

\_\_syncthreads();

d\_DstKey[ 0] = s\_key[threadIdx.x + 0];

d\_DstVal[ 0] = s\_val[threadIdx.x + 0];

d\_DstKey[(SHARED\_SIZE\_LIMIT / 2)] = s\_key[threadIdx.x + (SHARED\_SIZE\_LIMIT / 2)];

d\_DstVal[(SHARED\_SIZE\_LIMIT / 2)] = s\_val[threadIdx.x + (SHARED\_SIZE\_LIMIT / 2)];

}

static void mergeSortShared(

uint \*d\_DstKey,

uint \*d\_DstVal,

uint \*d\_SrcKey,

uint \*d\_SrcVal,

uint batchSize,

uint arrayLength,

uint sortDir

)

{

if (arrayLength < 2)

{

return;

}

assert(SHARED\_SIZE\_LIMIT % arrayLength == 0);

assert(((batchSize \* arrayLength) % SHARED\_SIZE\_LIMIT) == 0);

uint blockCount = batchSize \* arrayLength / SHARED\_SIZE\_LIMIT;

uint threadCount = SHARED\_SIZE\_LIMIT / 2;

if (sortDir)

{

mergeSortSharedKernel<1U><<<blockCount, threadCount>>>(d\_DstKey, d\_DstVal, d\_SrcKey, d\_SrcVal, arrayLength);

getLastCudaError("mergeSortShared<1><<<>>> failed\n");

}

else

{

mergeSortSharedKernel<0U><<<blockCount, threadCount>>>(d\_DstKey, d\_DstVal, d\_SrcKey, d\_SrcVal, arrayLength);

getLastCudaError("mergeSortShared<0><<<>>> failed\n");

}

}