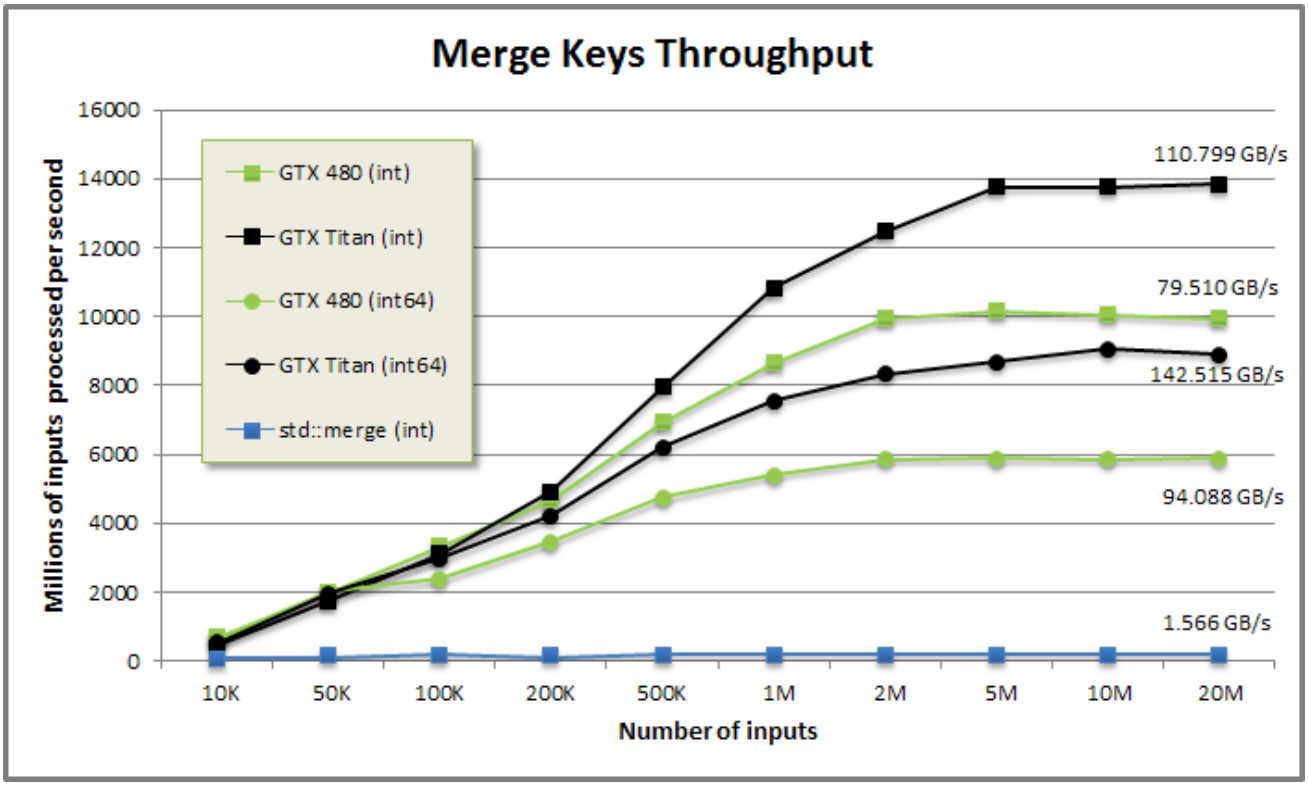
Code and text by Sean Baxter, NVIDIA Research.

« Bulk Remove and Bulk Insert Contents Mergesort

Merge

Merge two sorted sequences in parallel. This implementation supports custom iterators and comparators. It achieves throughputs greater than half peak bandwidth. MGPU's two-phase approach to scheduling is developed here.

Benchmark and usage



Merge Keys benchmark from benchmarkmerge/benchmarkmerge.cu

Merge keys demonstration from tests/demo.cu

```
142 void DemoMergeKeys(CudaContext& context) {
143     printf("\n\nMERGE KEYS DEMONSTRATION:\n\n");
144 
145     // Use CudaContext::SortRandom to generate 100 sorted random integers
146     // between 0 and 99.
147     int N = 100;
148     MGPU_MEM(int) aData = context.SortRandom<int>(N, 0, 99);
149     MGPU_MEM(int) bData = context.SortRandom<int>(N, 0, 99);
150 
151     printf("A:\n");
152     PrintArray(*aData, "%4d", 10);
153     printf("\nB:\n");
154     PrintArray(*bData, "%4d", 10);
155 
```

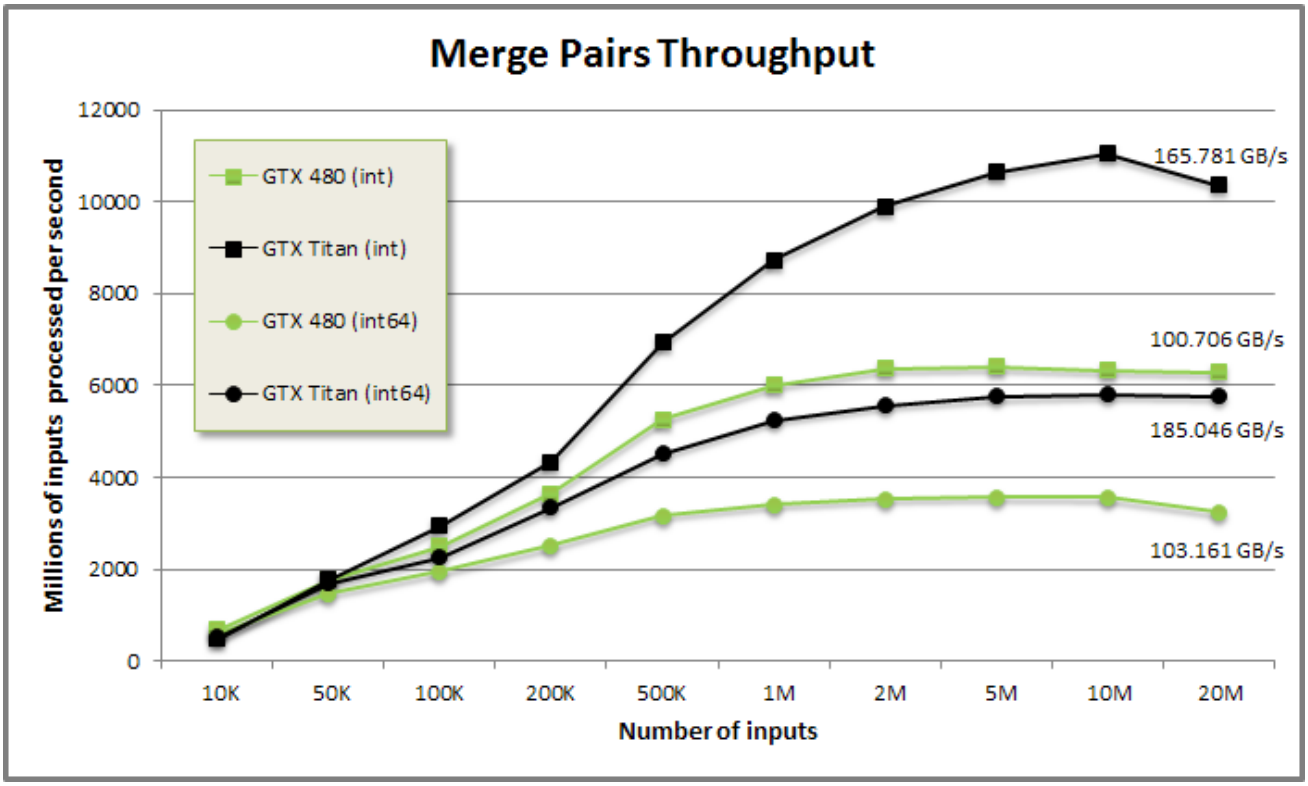
```
156     // Merge the two sorted sequences into one.
157     MGPU_MEM(int) cData = context.Malloc<int>(2 * N);
158     MergeKeys(aData->get(), N, bData->get(), N, cData->get(), mgpu::less<int>(),
159         context);
160
161     printf("\nMerged array:\n");
162     PrintArray(*cData, "%4d", 10);
163 }
```

```
MERGE KEYS DEMONSTRATION:

A:
    0:    0    0    3    4    4    7    7    7    8    8
   10:    9   10   11   12   13   13   13   14   14   15
   20:   16   16   18   18   19   22   23   23   25   25
   30:   26   26   28   31   34   34   35   36   38   39
   40:   40   43   43   43   44   44   45   46   47   49
   50:   50   50   50   51   52   52   53   53   54   54
   60:   55   57   60   60   62   62   62   65   66   67
   70:   68   68   71   72   74   74   76   77   79   80
   80:   80   81   82   82   85   85   85   86   86   86
   90:   91   91   91   92   96   97   97   98   98   99

B:
    0:    1    3    4    4    4    5    5    8    9   10
   10:   11   12   13   16   16   18   18   21   22   23
   20:   24   24   25   27   28   29   30   30   30   31
   30:   32   33   34   34   35   36   36   36   37   37
   40:   38   38   39   40   40   41   43   43   44   45
   50:   45   48   48   48   49   49   49   49   50   51
   60:   54   54   55   57   62   62   64   64   65   66
   70:   68   71   73   74   75   75   77   78   78   79
   80:   80   81   81   81   82   82   87   87   88   90
   90:   90   90   91   91   92   94   94   95   95   98

Merged array:
    0:    0    0    1    3    3    4    4    4    4    4
   10:    5    5    7    7    7    8    8    8    9    9
   20:   10   10   11   11   12   12   13   13   13   13
   30:   14   14   15   16   16   16   16   18   18   18
   40:   18   19   21   22   22   23   23   23   24   24
   50:   25   25   25   26   26   27   28   28   29   30
   60:   30   30   31   31   32   33   34   34   34   34
   70:   35   35   36   36   36   36   37   37   38   38
   80:   38   39   39   40   40   40   41   43   43   43
   90:   43   43   44   44   44   45   45   45   46   47
  100:   48   48   48   49   49   49   49   49   50   50
  110:   50   50   51   51   52   52   53   53   54   54
  120:   54   54   55   55   57   57   60   60   62   62
  130:   62   62   62   64   64   65   65   66   66   67
  140:   68   68   68   71   71   72   73   74   74   74
  150:   75   75   76   77   77   78   78   79   79   80
  160:   80   80   81   81   81   81   82   82   82   82
  170:   85   85   85   86   86   86   87   87   88   90
  180:   90   90   91   91   91   91   91   92   92   94
  190:   94   95   95   96   97   97   98   98   98   99
```



Merge Pairs benchmark from benchmarkmerge/benchmarkmerge.cu

Merge pairs demonstration from tests/demo.cu

```
167 void DemoMergePairs(CudaContext& context) {
168     printf("\n\nMERGE PAIRS DEMONSTRATION:\n\n");
169
170     int N = 100;
171     MGPU_MEM(int) aKeys = context.SortRandom<int>(N, 0, 99);
172     MGPU_MEM(int) bKeys = context.SortRandom<int>(N, 0, 99);
173     MGPU_MEM(int) aVals = context.FillAscending<int>(N, 0, 1);
174     MGPU_MEM(int) bVals = context.FillAscending<int>(N, N, 1);
175
176     printf("A:\n");
177     PrintArray(*aKeys, "%4d", 10);
178     printf("\nB:\n");
179     PrintArray(*bKeys, "%4d", 10);
180
181     // Merge the two sorted sequences into one.
182     MGPU_MEM(int) cKeys = context.Malloc<int>(2 * N);
183     MGPU_MEM(int) cVals = context.Malloc<int>(2 * N);
184     MergePairs(aKeys->get(), aVals->get(), N, bKeys->get(), bVals->get(), N,
185         cKeys->get(), cVals->get(), mgpu::less<int>(), context);
186
187     printf("\nMerged keys:\n");
188     PrintArray(*cKeys, "%4d", 10);
189     printf("\nMerged values (0-99 are A indices, 100-199 are B indices).\n");
190     PrintArray(*cVals, "%4d", 10);
191 }
```

```
MERGE PAIRS DEMONSTRATION:

A:
    0:    1    1    2    4    8    8   10   11   11   11
   10:   13   14   14   16   16   17   18   18   19   19
   20:   19   20   21   22   22   22   23   23   23   24
```

```
  30:    24   25   26   26   26   28   29   30   31   31
  40:    32   34   35   35   37   38   40   42   42   43
  50:    43   43   44   44   45   47   47   47   48   50
  60:    53   54   54   55   57   58   58   59   60   62
  70:    63   64   64   65   68   70   71   72   73   76
  80:    77   78   79   79   80   81   83   84   87   88
  90:    90   90   92   92   93   94   96   97   99   99

B:
   0:     0    1    1    2    3    3    6    9    9   10
  10:    12   13   15   16   17   18   18   19   22   23
  20:    23   23   23   24   25   26   26   28   29   29
  30:    31   31   32   32   33   33   33   35   36   38
  40:    39   40   40   41   42   47   47   47   48   48
  50:    48   49   50   50   50   50   51   51   52   54
  60:    57   58   59   60   60   61   61   62   63   65
  70:    67   67   68   69   71   71   71   72   74   74
  80:    76   76   77   79   80   84   85   88   88   88
  90:    89   90   90   91   93   95   96   96   97   98

Merged keys:
   0:     0    1    1    1    1    2    2    3    3    4
  10:     6    8    8    9    9   10   10   11   11   11
  20:    12   13   13   14   14   15   16   16   16   17
  30:    17   18   18   18   18   19   19   19   19   20
  40:    21   22   22   22   22   23   23   23   23   23
  50:    23   23   24   24   24   25   25   26   26   26
  60:    26   26   28   28   29   29   29   30   31   31
  70:    31   31   32   32   32   33   33   33   34   35
  80:    35   35   36   37   38   38   39   40   40   40
  90:    41   42   42   42   43   43   43   44   44   45
 100:    47   47   47   47   47   47   48   48   48   48
 110:    49   50   50   50   50   50   51   51   52   53
 120:    54   54   54   55   57   57   58   58   58   59
 130:    59   60   60   60   61   61   62   62   63   63
 140:    64   64   65   65   67   67   68   68   69   70
 150:    71   71   71   71   72   72   73   74   74   76
 160:    76   76   77   77   78   79   79   79   80   80
 170:    81   83   84   84   85   87   88   88   88   88
 180:    89   90   90   90   90   91   92   92   93   93
 190:    94   95   96   96   96   97   97   98   99   99

Merged values (0-99 are A indices, 100-199 are B indices)
   0:   100    0    1  101  102    2  103  104  105    3
  10:   106    4    5  107  108    6  109    7    8    9
  20:   110   10  111   11   12  112   13   14  113   15
  30:   114   16   17  115  116   18   19   20  117   21
  40:    22   23   24   25  118   26   27   28  119  120
  50:   121  122   29   30  123   31  124   32   33   34
  60:   125  126   35  127   36  128  129   37   38   39
  70:   130  131   40  132  133  134  135  136   41   42
  80:    43  137  138   44   45  139  140   46  141  142
  90:   143   47   48  144   49   50   51   52   53   54
 100:    55   56   57  145  146  147   58  148  149  150
 110:   151   59  152  153  154  155  156  157  158   60
 120:    61   62  159   63   64  160   65   66  161   67
 130:   162   68  163  164  165  166   69  167   70  168
 140:    71   72   73  169  170  171   74  172  173   75
 150:    76  174  175  176   77  177   78  178  179   79
 160:   180  181   80  182   81   82   83  183   84  184
 170:    85   86   87  185  186   88   89  187  188  189
 180:   190   90   91  191  192  193   92   93   94  194
 190:    95  195   96  196  197   97  198  199   98   99
```

Host functions

include/mgpuhost.cuh

```
////////////////////////////////////////////////////////////////////////////////
// kernels/merge.cuh

// MergeKeys merges two arrays of sorted inputs with C++-comparison semantics.
// aCount items from aKeys_global and bCount items from bKeys_global are merged
// into aCount + bCount items in keys_global.
// Comp is a comparator type supporting strict weak ordering.
// If !comp(b, a), then a is placed before b in the output.
template<typename KeysIt1, typename KeysIt2, typename KeysIt3, typename Comp>
MGPU_HOST void MergeKeys(KeysIt1 aKeys_global, int aCount, KeysIt2 bKeys_global,
    int bCount, KeysIt3 keys_global, Comp comp, CudaContext& context);

// MergeKeys specialized with Comp = mgpu::less<T>.
template<typename KeysIt1, typename KeysIt2, typename KeysIt3>
MGPU_HOST void MergeKeys(KeysIt1 aKeys_global, int aCount, KeysIt2 bKeys_global,
    int bCount, KeysIt3 keys_global, CudaContext& context);

// MergePairs merges two arrays of sorted inputs by key and copies values.
// If !comp(bKey, aKey), then aKey is placed before bKey in the output, and
// the corresponding aData is placed before bData. This corresponds to *_by_key
// functions in Thrust.
template<typename KeysIt1, typename KeysIt2, typename KeysIt3, typename ValsIt1,
    typename ValsIt2, typename ValsIt3, typename Comp>
MGPU_HOST void MergePairs(KeysIt1 aKeys_global, ValsIt1 aVals_global,
    int aCount, KeysIt2 bKeys_global, ValsIt2 bVals_global, int bCount,
    KeysIt3 keys_global, ValsIt3 vals_global, Comp comp, CudaContext& context);

// MergePairs specialized with Comp = mgpu::less<T>.
template<typename KeysIt1, typename KeysIt2, typename KeysIt3, typename ValsIt1,
    typename ValsIt2, typename ValsIt3>
MGPU_HOST void MergePairs(KeysIt1 aKeys_global, ValsIt1 aVals_global,
    int aCount, KeysIt2 bKeys_global, ValsIt2 bVals_global, int bCount,
    KeysIt3 keys_global, ValsIt3 vals_global, CudaContext& context);
```

Two-stage design

Further Reading: Read ***GPU Merge Path - A GPU Merging Algorithm*** by Oded Green, Robert McColl, and David A. Bader for another discussion on using Merge Path partitioning to implement merge with CUDA.

CPU Merge implementation

```
template<typename T, typename Comp>
void CPUMerge(const T* a, int aCount, const T* b, int bCount, T* dest,
    Comp comp) {

    int count = aCount + bCount;
    int ai = 0, bi = 0;
    for(int i = 0; i < count; ++i) {
        bool p;
        if(bi >= bCount) p = true;
        else if(ai >= aCount) p = false;
        else p = !comp(b[bi], a[ai]);

        dest[i] = p ? a[ai++] : b[bi++];
    }
}
```

Merge is the simplest function that is constructed in the two-phase style promoted by this project. Developing algorithms in the two-phase style begins with writing down a serial implementation. `CPUMerge` is a good point of reference because it consumes one input and emits one output per iteration. Our goal is to:

1. Divide the domain into partitions of exactly the same size. We use the Merge Path ideas covered on the previous page to assist with partitioning and scheduling. A coarse-grained search over the inputs in global memory breaks the problem into tiles with workloads of constant size. A fine-grained search over the inputs in shared memory breaks the problem into threads with workloads of constant size.

2. Develop a serial merge, like `CPUMerge` above, that is executed in parallel and in isolation by each thread to process distinct intervals of the problem. Rather than running over the entire input, as in `CPUMerge`, each thread performs exactly VT iterations, consuming VT input and emitting VT output. This strategy has the same linear work efficiency as a standard sequential merge (parallel algorithms often choose to sacrifice work efficiency to gain concurrency).

By decoupling scheduling and work, the two-phase strategy assists the programmer in developing readable and composable algorithms. We'll show in a future page how to replace the serial portion of the parallel merge to execute high-throughput vectorized sorted searches.

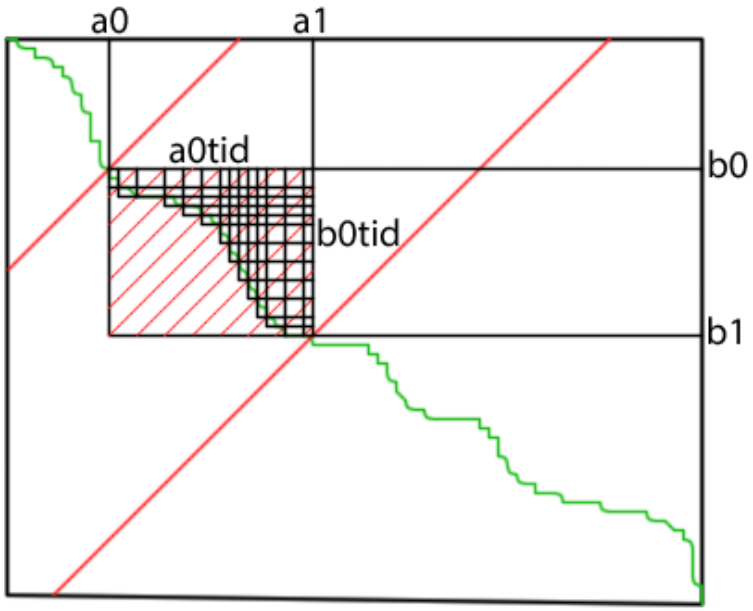
Algorithm

include/device/ctamerge.cuh

```
205 template<int NT, int VT, typename It1, typename It2, typename T, typename Comp>
206 MGPU_DEVICE void DeviceMergeKeysIndices(It1 a_global, It2 b_global, int4 range,
207     int tid, T* keys_shared, T* results, int* indices, Comp comp) {
208
209     int a0 = range.x;
210     int a1 = range.y;
211     int b0 = range.z;
212     int b1 = range.w;
213     int aCount = a1 - a0;
214     int bCount = b1 - b0;
215
216     // Load the data into shared memory.
217     DeviceLoad2ToShared<NT, VT, VT>(a_global + a0, aCount, b_global + b0,
218         bCount, tid, keys_shared);
219
220     // Run a merge path to find the start of the serial merge for each thread.
221     int diag = VT * tid;
222     int mp = MergePath<MgpuBoundsLower>(keys_shared, aCount,
223         keys_shared + aCount, bCount, diag, comp);
224
225     // Compute the ranges of the sources in shared memory.
226     int a0tid = mp;
227     int a1tid = aCount;
228     int b0tid = aCount + diag - mp;
229     int b1tid = aCount + bCount;
230
231     // Serial merge into register.
232     SerialMerge<VT, true>(keys_shared, a0tid, a1tid, b0tid, b1tid, results,
233         indices, comp);
234 }
```

MGPU Merge merges two sorted inputs with C++ std::merge ordering semantics. As in Bulk Insert, the source inputs are partitioned into equal size-interval pairs by calling `MergePathPartitions`. We double-down on this divide-and-conquer strategy by calling `MergePath` a *second time*, locally searching over the keys in a tile.

`DeviceMergeKeysIndices` is a re-usable CTA-level function that merges keys provided in shared memory. The caller secifies the tile's intervals over A and B in the `range` argument. `range` is derived by `ComputeMergeRange` using the intersections of the tile's cross-diagonals with the Merge Path, as illustrated here. `DeviceLoad2ToShared` performs an optimized, unrolled, cooperative load of a variable number of contiguous elements from two input arrays. Loaded keys are stored in shared memory: A's contributions in (0, aCount) and B's contributions in (aCount, aCount + bCount).



`MergePath` is called by all threads in parallel to find their individual partitions. This is a faster search than the global partitioning search because shared memory has much lower latency, and intra-CTA cross-diagonals are much shorter than global cross-diagonals, resulting in binary searches that converge after fewer iterations. The intra-CTA Merge Path searches are conducted in the tile's local coordinate system. Cross-diagonals are given indices VT * tid.

The starting cursor for each thread (a0tid and b0tid) is handed to `SerialMerge`, which loads keys from shared memory, merges them, and returns a fragment of the result in register.

include/device/ctamerge.cuh

```
46  template<int VT, bool RangeCheck, typename T, typename Comp>
47  MGPU_DEVICE void SerialMerge(const T* keys_shared, int aBegin, int aEnd,
48      int bBegin, int bEnd, T* results, int* indices, Comp comp) {
49
50      T aKey = keys_shared[aBegin];
51      T bKey = keys_shared[bBegin];
52
53      #pragma unroll
54      for(int i = 0; i < VT; ++i) {
55          bool p;
56          if(RangeCheck)
57              p = (bBegin >= bEnd) || ((aBegin < aEnd) && !comp(bKey, aKey));
58          else
59              p = !comp(bKey, aKey);
```

```
60
61          results[i] = p ? aKey : bKey;
62          indices[i] = p ? aBegin : bBegin;
63
64          if(p) aKey = keys_shared[++aBegin];
65          else bKey = keys_shared[++bBegin];
66      }
67      __syncthreads();
68  }
```

Partitioning doesn't really differentiate merge from similar functions, as all it does is handle scheduling. The soul of this function is `SerialMerge`. Incredible throughput is achieved because Merge Path isn't simply a very good partitioning; it's an *exact* partition. The merge kernel is tuned to a specific (odd) number of values per thread. For a CTA with 128 threads (NT) and 11 values per thread (VT), each tile loads and merges 1408 inputs (NV). These inputs aren't simply merged cooperatively, though. They are merged *independently* by the 128 threads, 11 per thread, which is far better.

Because each thread merges precisely 11 elements, the `SerialMerge` routine can unroll its loop. Accesses to the output arrays `results` and `indices` are now static (the iterator for unrolled loops is treated as a constant by the compiler). Because we're using only static indexing, the outputs can be stored in *register* rather than shared memory. RF capacity is much higher than shared memory capacity, and the performance tuning strategy of increasing grain size to amortize partitioning costs always results in underoccupied kernels. Storing outputs in register cuts the kernel's shared memory footprint in half, doubling occupancy, and boosting performance.

> **Important:** Structure your code to only dynamically index either the sources or the destinations (not both). Use loop unrolling to statically index the complementary operations in register, then synchronize and swap. Exact partitioning facilitates this pattern, which doubles occupancy to improve latency-hiding.

Keys are returned into `results`. `indices` (the locations of keys in shared memory) are also returned to facilitate a value gather for sort-by-key. For key-only merge, operations involving `indices` should be eliminated by the compiler.

Note that the next item in each sequence is fetched *prior* to the start of the next iteration. This reduces two shared loads per thread to just one, which reduces bank conflicts across the warp. Unfortunately it may also cause us to read off the end of the B array. To prevent an illegal access failure in the kernel, *allocate at leats one extra slot in shared memory*. This doesn't compromise occupancy at all, because we use odd numbered VT parameters—we can reserve up to a full additional slot per thread before the extra provisioning reduces the number of concurrent CTAs per SM.

> **Important:** If fetching the next iteration's data at the end of the loop body, allocate an extra slot in shared memory to prevent illegal access violations.

include/device/ctamerge.cuh

```
241 template<int NT, int VT, bool HasValues, typename KeysIt1, typename KeysIt2,
242     typename KeysIt3, typename ValsIt1, typename ValsIt2, typename KeyType,
243     typename ValsIt3, typename Comp>
244 MGPU_DEVICE void DeviceMerge(KeysIt1 aKeys_global, ValsIt1 aVals_global,
245     KeysIt2 bKeys_global, ValsIt2 bVals_global, int tid, int block, int4 range,
246     KeyType* keys_shared, int* indices_shared, KeysIt3 keys_global,
247     ValsIt3 vals_global, Comp comp) {
248
249     KeyType results[VT];
250     int indices[VT];
```

```
    DeviceMergeKeysIndices<NT, VT>(aKeys_global, bKeys_global, range, tid,
        keys_shared, results, indices, comp);

    // Store merge results back to shared memory.
    DeviceThreadToShared<VT>(results, tid, keys_shared);

    // Store merged keys to global memory.
    int aCount = range.y - range.x;
    int bCount = range.w - range.z;
    DeviceSharedToGlobal<NT, VT>(aCount + bCount, keys_shared, tid,
        keys_global + NT * VT * block);

    // Copy the values.
    if(HasValues) {
        DeviceThreadToShared<VT>(indices, tid, indices_shared);
        DeviceTransferMergeValues<NT, VT>(aCount + bCount,
          aVals_global + range.x, bVals_global + range.z, aCount,
          indices_shared, tid, vals_global + NT * VT * block);
    }
}
```

`DeviceMerge`, one level closer to the kernel, invokes `DeviceMergeKeysIndices` and receives the merged results and indices in register. Each thread uses `DeviceThreadtoShared` to store its merged keys to shared memory at VT * tid + i, synchronizes, and calls `DeviceSharedToGlobal` to cooperatively make coalesced stores to the destination array. `DeviceTransferMergeValues` (discussed here) uses the indices to gather values from global memory and store them back, coalesced, to `vals_global`.

`DeviceMerge` does the heavy lifting for both MGPU's merge and mergesort kernels.

To recap merge:

1. Prior to the merge kernel, use `MergePathPartitions` for coarse-grained exact partitioning.
2. At the start of the kernel, `ComputeMergeRange` determines the intervals to load from arrays A and B. `DeviceLoad2ToShared` loads these into shared memory; first A, then B.
3. `MergePath` searches keys in shared memory to find a fine-grained partitioning of data, with VT items per thread.
4. Each thread makes VT trips through an unrolled loop, dynamically indexing into shared memory retrieving keys, comparing them, and emitting the smaller key to an array in register, using the static loop iterator.
5. After synchronization each thread writes its values back at VT * tid + i (thread order). The values are cooperatively transferred to the destination in global memory using coalesced stores.
6. Indices are stored to shared memory (writing from thread order into strided order). `DeviceTransferMergeValues` uses these to gather merged values from the input. It makes coalesced stores to the destination.

Much of the MGPU Merge implementation is shared with Mergesort—these portions are covered on the next page.

« **Bulk Remove and Bulk Insert** **Contents** **Mergesort** »