GPU TECHNOLOGY CONFERENCE

Efficient Merge, Search, and Set Operations on GPU

Sean Baxter
Duane Merrill

NVIDIA Research

Merge-like Functions

- Merge and mergesort
- Vectorized searches – sorted needles in sorted haystack
  - lower_bound, upper_bound, equal_range, counts
- Sets (C++ multisets definition)
  - set_intersection, set_union, set_difference, set_symmetric_difference
- Joins (built on sorted searches)
  - Inner, left, right, full
  - Semi-join, anti-join

Merge-like Functions

- Input: two sorted sequences.
- Output: one sequence with sorted content.

- These problems aren't *obviously* parallel.

Merge Definition

– A: $1^0\ 2^0\ 2^1\ 2^2\ 3^0\ 6^0\ 6^1\ 6^2\ 7^0\ 7^1\ 8^0\ 8^1$

– B: $2^0\ 2^1\ 3^0\ 3^1\ 4^0\ 5^0\ 6^0\ 6^1\ 6^2\ 8^0\ 8^1\ 9^0$

- Order inputs by key, then source (A or B), then rank.
  - Rank is handled by reading left-to-right.
  - This property necessary for stable merge and mergesort.

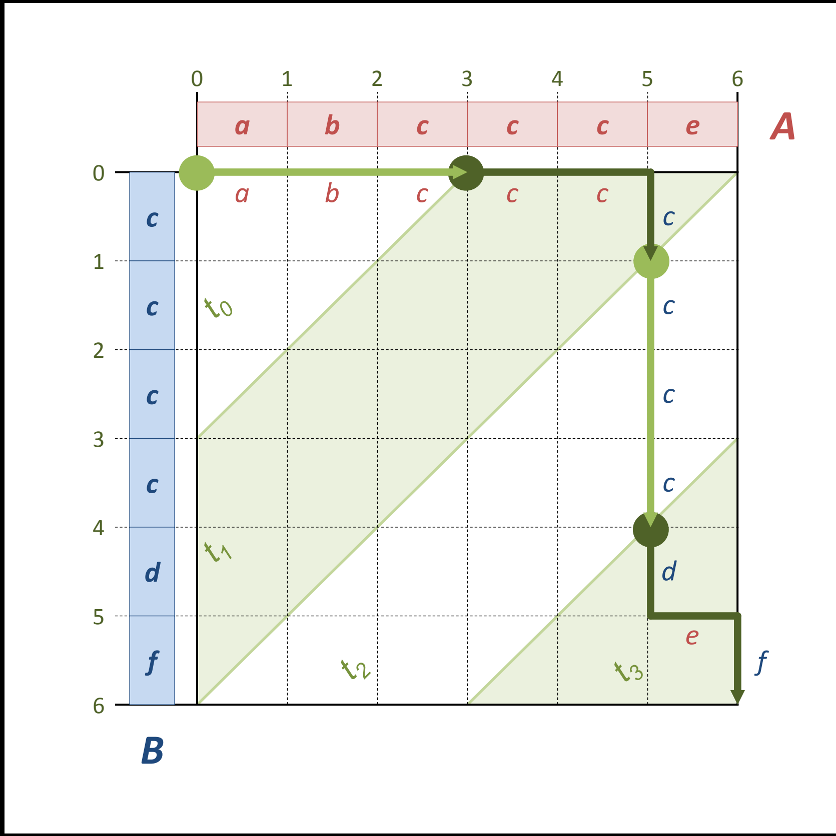
- $1_A^0\ 2_A^0\ 2_A^1\ 2_A^2\ 2_B^0\ 2_B^1\ 3_A^0\ 3_B^0\ 3_B^1\ 4_B^0\ 5_B^0 \ldots$

Solving Merge

- Two-pass solution:
  1. Parallel Partition
     Map *exactly* the necessary inputs into each tile.
  2. Serial Streaming
     Process *exactly* the grain size of inputs per thread.

Merge Path

- Merge Path provides useful visualization and mental model for parallel partitioning

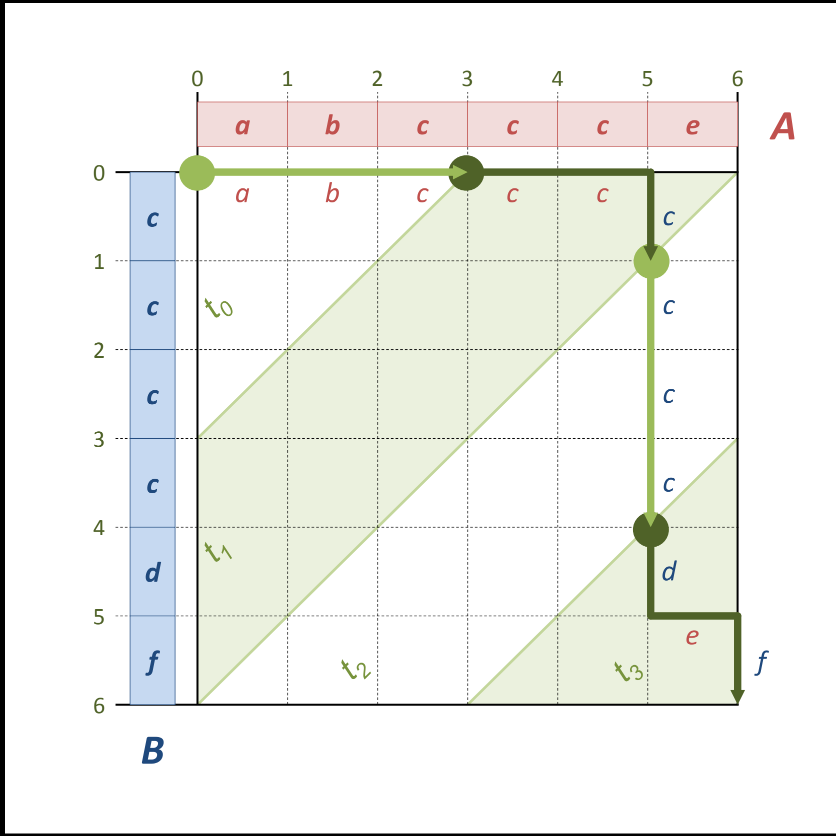


Binary search for intersection of cross-diagonals with Merge Path

Merge Path - Parallel Merging Made Simple (2012)
Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, Yitzhak Birk

GPU Merge Path – A GPU Merging Algorithm (2012)
Oded Green, Robert McColl, David A. Bader

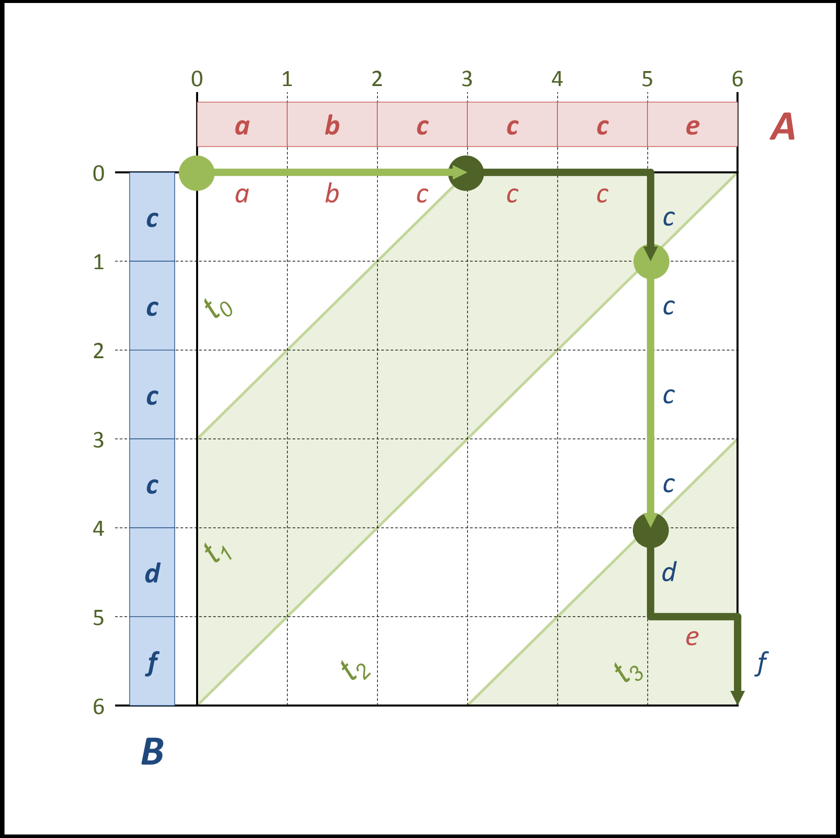
Merge Path



- Search input sequences for splitters A_d and B_d such that $A_d + B_d = d$
- Intervals (A_{begin}, B_{begin}) to (A_d, B_d) may be merged; intervals (A_d, B_d) to (A_{end}, B_{end}) may be merged.
- Don't try ad-hoc binary search.

Merge Path

We have two arrays, but choose a coordinate system so that we only have one *search*.



- Use constraint to advantage:
  - $A_d + B_d = d$ =>
    - $A_i = i$
    - $B_i = d - i$
    - Binary search over i (range i from 0 to d)
- Each CTA has 2 partitions: *begin* and *end* for defining interval to load. Load exactly enough data to merge CTASize elements.

```
// Use constraints to transform from array to cross-diagonal
// coordinates
int begin = max(0, diag - bCount);
int end = min(diag, aCount);

// Binary search in cross-diagonal coordinates
while(begin < end) {
    int mid = (begin + end) / 2;
    int aIndex = mid;
    int bIndex = diag - mid - 1;
    // p = aData[aIndex] <= bData[bIndex]
    // For std::merge-like semantics
    bool p = !comp(bData[bIndex], aData[aIndex]);
    if(p) begin = mid + 1;
    else end = mid;
}
// Transform back into array coordinates.
// As required, aIndex + bIndex == diag.
int aIndex = begin;
int bIndex = diag - begin;
```

Serial Work

- Binary search partitioning must be done to locate the starting point for each thread.
- Serial merge/search/set functions do all actual work.
- Serial routines have O(n) complexity (better than O(n log n) for most parallel operations).

Serial Merge

```
#pragma unroll
for(int i = 0; i < Count; ++i) {
    T x1 = keys[aBegin];
    T x2 = keys[bBegin];

    // If p is true, emit from A, otherwise emit from B.
    bool p;
    if(bBegin >= bEnd) p = true;
    else if(aBegin >= aEnd) p = false;
    else p = !comp(x2, x1);       // p = x1 <= x2

    // because of #pragma unroll, merged[i] is static indexing
    // so merged is kept in RF, not smem!
    merged[i] = p ? x1 : x2;
    if(p) ++aBegin;
    else ++bBegin;
}
```

Parallel sets

- A: $1^0\ 2^0\ 2^1\ 2^2\ 3^0\ 6^0\ 6^1\ 6^2\ 7^0\ 7^1\ 8^0\ 8^1$
- B: $2^0\ 2^1\ 3^0\ 3^1\ 4^0\ 5^0\ 6^0\ 6^1\ 6^2\ 8^0\ 8^1\ 9^0$

- C++ implements multisets:
  - Match elements in A and B with equal key and rank
  - A: $1^0\ 2^0\ 2^1\ 2^2\ 3^0\quad\quad\quad\quad 6^0\ 6^1\ 6^2\ 7^0\ 7^1\ 8^0\ 8^1$
  - B: $\quad\ 2^0\ 2^1\quad\ 3^0\ 3^1\ 4^0\ 5^0\ 6^0\ 6^1\ 6^2\quad\quad\quad 8^0\ 8^1\ 9^0$

- C++ implements 4 multiset operations.

Parallel sets

- std::set_intersection
  - Emit A's element if key+rank match with B:
  - *A:* $1^0\ 2^0\ 2^1\ 2^2\ 3^0\ 6^0\ 6^1\ 6^2\ 7^0\ 7^1\ 8^0\ 8^1$
  - *B:* $2^0\ 2^1\ 3^0\ 3^1\ 4^0\ 5^0\ 6^0\ 6^1\ 6^2\ 8^0\ 8^1\ 9^0$
- std::set_union
  - Emit A or B element unless key+rank match, then only emit A and advance past B:
  - *A:* $1^0\ 2^0\ 2^1\ 2^2\ 3^0\ 6^0\ 6^1\ 6^2\ 7^0\ 7^1\ 8^0\ 8^1$
  - *B:* $2^0\ 2^1\ 3^0\ 3^1\ 4^0\ 5^0\ 6^0\ 6^1\ 6^2\ 8^0\ 8^1\ 9^0$

Parallel sets

- std::set_difference
  - Emit A's element if no key+rank match with B.
  - *A:* $1^0\ 2^0\ 2^1\ 2^2\ 3^0\ 6^0\ 6^1\ 6^2\ 7^0\ 7^1\ 8^0\ 8^1$
  - *B:* $2^0\ 2^1\ 3^0\ 3^1\ 4^0\ 5^0\ 6^0\ 6^1\ 6^2\ 8^0\ 8^1\ 9^0$

- std::set_symmetric_difference
  - Emit A's element if no key+rank match with B, and emit B's element if no key+rank match with A.
  - *A:* $1^0\ 2^0\ 2^1\ 2^2\ 3^0\ 6^0\ 6^1\ 6^2\ 7^0\ 7^1\ 8^0\ 8^1$
  - *B:* $2^0\ 2^1\ 3^0\ 3^1\ 4^0\ 5^0\ 6^0\ 6^1\ 6^2\ 8^0\ 8^1\ 9^0$

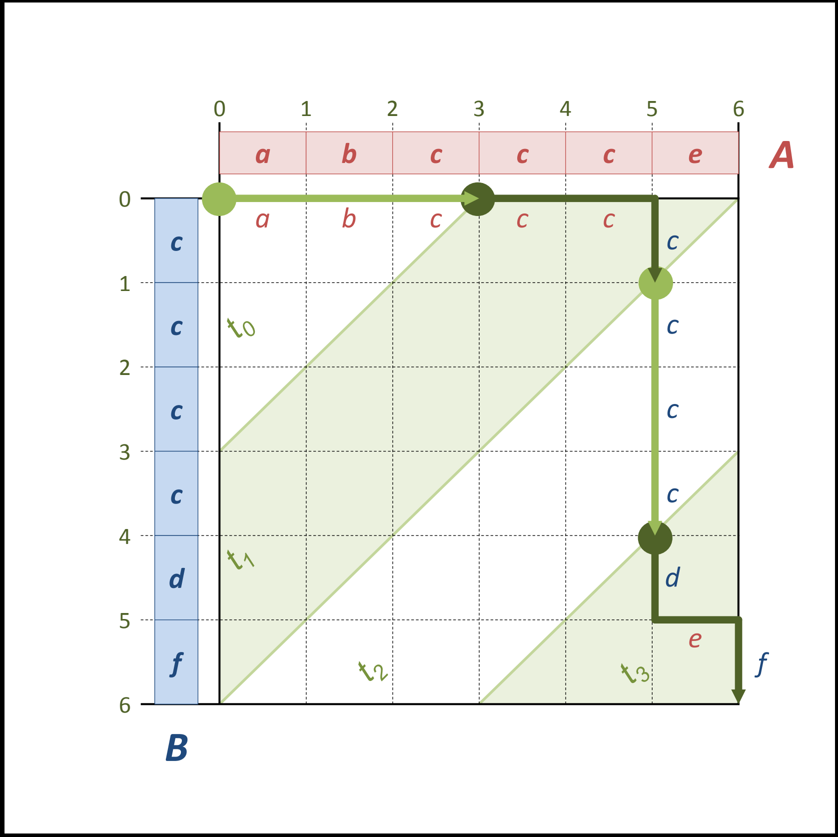
Sets Workflow

1. Partition Kernel
   - Run global Balanced Path on inputs to exactly partition over tiles.
2. Set Op Kernel
   - Load input intervals into each tile.
   - Run Balanced Path on tile data in shared memory.
   - Execute work-efficient serial multiset operations.
   - Compact outputs across threads in CTA using bitfield.
3. Compact Kernel
   - Globally compact outputs.

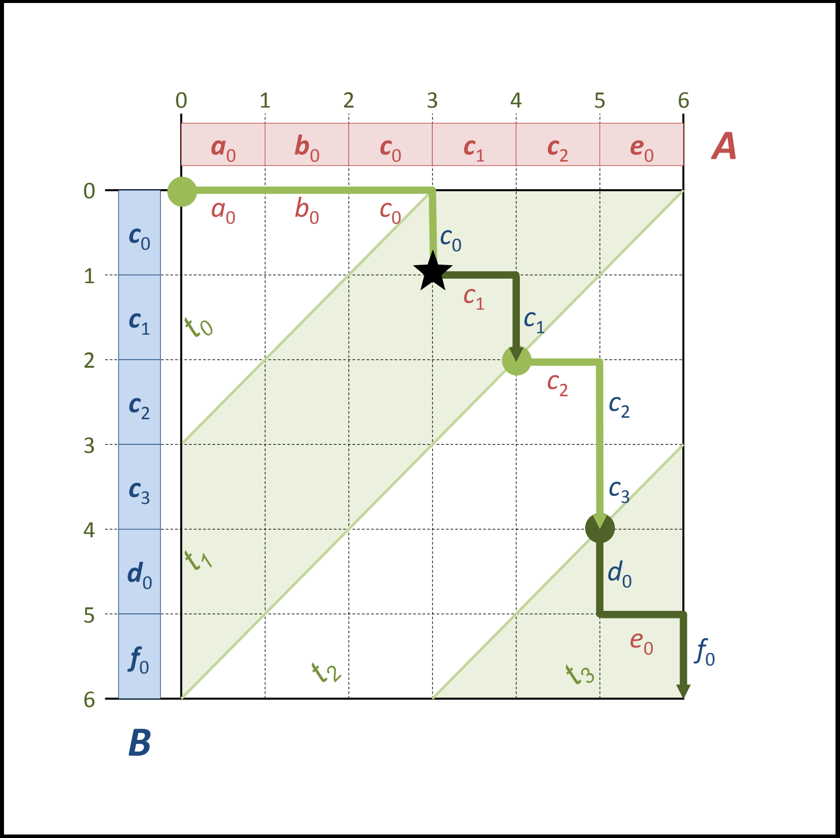
Balanced Path

We introduce Balanced Path for multiset partitioning.

Merge Path – A before B



Balanced Path – balance by rank

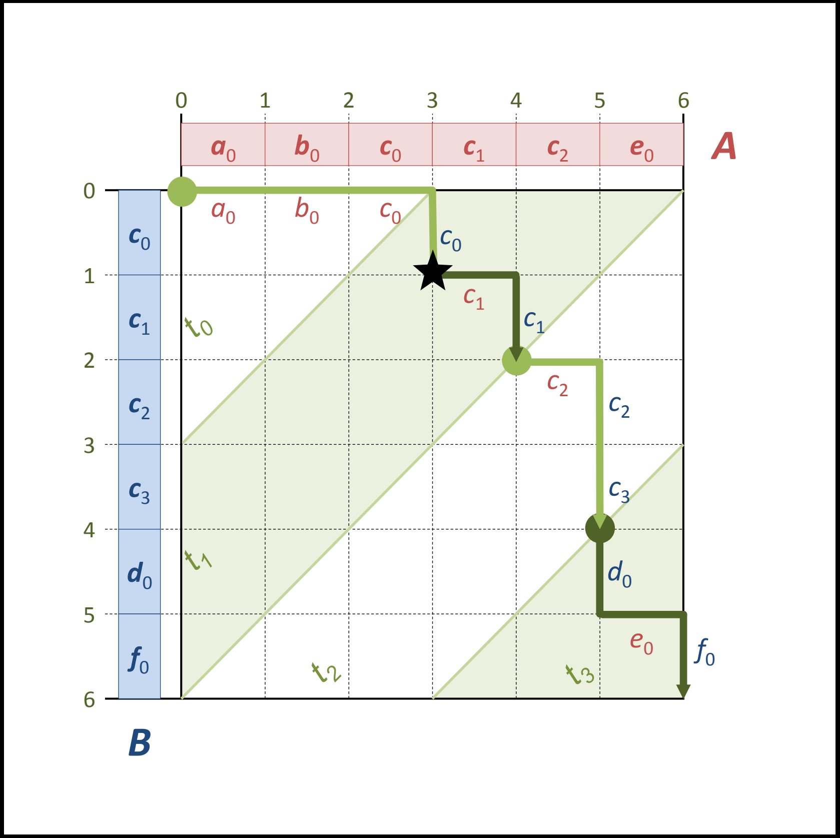


Balanced Path

Balanced Path has a ‘stair-step’ shape, following equal key-rank occurrences in A and B.

- First run Merge Path to identify the key.
- Next binary search both A and B to find the first occurrence of that key in each input array.
- Forward project to include an equal number of duplicates from each input array to the left of the cross-diagonal.

Balanced Path – balance by rank



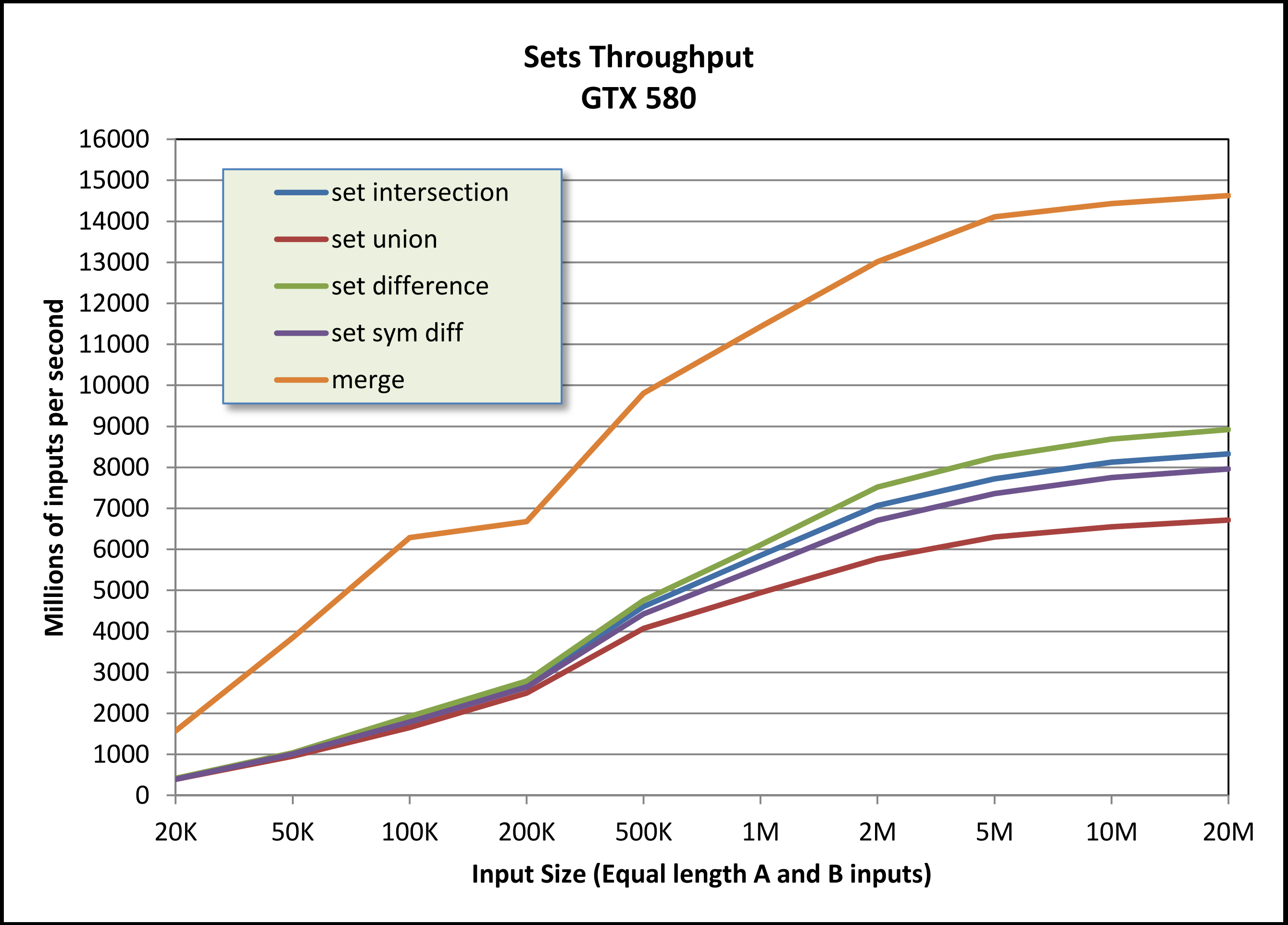
Balanced Path

```
int p = MergePath(a, aCount, b, bCount, diag);
int aIndex = p;
int bIndex = diag - p;

// A ‘starred’ Balanced Path means we steal one element
// from B to match its corresponding key-rank item in A.
// There are diag + (int)star elements to the left of the partition.
bool star = false;
if(bIndex < bCount) {
    T x = b[bIndex]; // Search for start of x duplicates in A and B.
    int aStart = LowerBound(a, aIndex, x);
    int bStart = LowerBound(b, bIndex, x);

    int aRun = aIndex – aStart;
    int bRun = bIndex – bStart;
    int xCount = aRun + bRun;
```

Balanced Path

```
    // Attempt to advance b and regress a.
    int bAdvance = max(xCount / 2, xCount - aRun);
    int bEnd = min(bCount, bStart + bAdvance + 1);
    int bRunEnd = UpperBound(b + bIndex, bEnd - bIndex, x) + bIndex;
    bRun = bRunEnd - bStart;
    bAdvance = min(bAdvance, bRun);
    int aAdvance = xCount - bAdvance;

    // If we're adding an odd number of duplicates, and there's another
    // duplicate in B, star the search to pair it with the last included
    // duplicate in A.
    bool roundUp = (aAdvance == bAdvance + 1) && (bAdvance < bRun);
    aIndex = aStart + aAdvance;
    if(roundUp) star = true;
}
return (aIndex, star);
```

Results



Single-threaded
C++ STL functions
2.8GHz Sandy i7

Merge: 191M/s
Int: 195M/s
Union: 195M/s
Diff: 190M/s
Sym Diff: 199M/s

GPU Merge is 75x faster than single CPU thread.
GPU Sets are 35x-47x faster than single CPU thread.

Tuning Pattern

- Balanced Path for partitioning is run once per thread, no matter grain size.
- Amortize partitioning cost by increasing grain size (do more work per search)
- Bigger grain size = more smem usage.
- More smem = lower occupancy.
- Lower occupancy = worse performance.

- Autotune for good performance based on device, datatype, distribution, and input sizes.

Questions?

- sbaxter@nvidia.com
- dumerrill@nvidia.com

- (Merge Path merge and Balanced Path set operations already in CUDA thrust 1.7)

Optimizing Shared Memory

- Principles for all merge-like functions:
- Store values in shared memory – this is exact fit thanks to Merge Path/Balanced Path partitioning.
- Use #pragma unroll to merge from shared memory (we need gather) into register.
- __syncthreads then store back all results.
- Only need space to store elements once in shared memory.

Serial set_symmetric_difference

Each iteration emits 0 or 1 outputs. Static indexing requires building bitfield of valid outputs and then compacting with scan.

```
// For simplicity, assume we're not at end of array.
int available = 0;    // bitfield indicating which outputs are valid.
#pragma unroll
for(int i = 0; i < Count; ++i) {
    // NOTE: continue; if already advanced Count pointers.
    T aKey = data[aBegin];
    T bKey = data[bBegin];
    bool pA = comp(aKey, bKey);   // aKey < bKey
    bool pB = comp(bKey, aKey);   // bKey < aKey
    results[i] = pA ? aKey : bKey;

    if(pA != pB) available |= 1<< i;
    if(!pA) ++aBegin;
    if(!pB) ++bBegin;
}
return available;
```