Merge-like Functions and Load-Balancing Search

Sean Baxter

NVIDIA Research

NVIDIA

Modern GPU (new)

http://nvlabs.github.io/moderngpu website

https://www.github.com/nvlabs/moderngpu code

fork me!

sbaxter@nvidia.com email

@moderngpu twitter

http://www.moderngpu.com/stanford.pptx slides

Modern GPU (2)

- http://nvlabs.github.io/moderngpu

- New library and ebook for CUDA programming
  - Bulk Insert/Bulk Remove
  - Merge
  - Mergesort
  - Segmented sort
  - Vectorized sorted search
  - Load-balancing search
  - IntervalMove (vectorized memcpy)
  - Relational joins
  - Multiset operations (eg set_difference)

Irregular problems

- Show that GPU can handle irregular problems.
- Domain of GPU computing is more than embarrassingly-parallel problems.
- C++ STL array-processing functions are good starting point.

Scheduling and decomposition

- Challenge of irregular problems is scheduling or decomposition
- We map what work to which threads, and when do we run it?
- Scheduling not a real concern with sequential CPU programming. Central issue in massively-parallel programming.

Serial Merge

```
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]);  // a[ai] <= b[bi]

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

Examine two keys and output one element per iteration. O(n) work-efficiency.

Parallel Merge

- **PRAM-style merge**
  - **Low-latency when number of processors is order N.**
  - **One item per thread. Communication free.**

- **Two kernels:**

1. **KernelA assigns one thread to each item in A.**

   Thread i *lower-bound* binary searches into B for key A[i].

   Insert A[i] into dest at i + lower_bound(B, A[i]).

2. **KernelB assigns one thread to each item in B.**

   Thread i *upper-bound* binary searches into A for key B[i].

   Insert B[i] into dest at i + upper_bound(A, B[i]).

Parallel Merge (2)

```
template<int NT, typename T, typename Comp>
__global__ void ParallelMergeA(const T* a_global, int aCount,
    const T* b_global, int bCount, T* dest_global, Comp comp) {

    int gid = threadIdx.x + NT * blockIdx.x;
    if(gid < aCount) {
        T aKey = a_global[gid];
        int lb = BinarySearch<MgpuBoundsLower>(b_global, bCount,
            aKey, comp);
        dest_global[gid + lb] = aKey;
    }
}
```

ParallelMergeB is symmetric. Uses upper-bound for B into A search.

Parallel Merge (3)

- Parallel version is highly concurrent but very inefficient.
  - Serial code is O(n)
  - Parallel code is O(n log n)
- Parallel code doesn’t resemble sequential code at all
  - Hard to extend to other merge-like operations.
- Parallel code tries to solve two problems at once:
  1. Decomposition/scheduling work to parallel processors.
  2. Merge-specific logic

Two-phase decomposition

- Design implementation in two phases:
  1. PARTITIONING PHASE

     Maps work onto each CTA or thread.

     Has adjustable *grain size* parameter (VT).

     Implemented with one binary search per CTA or thread.

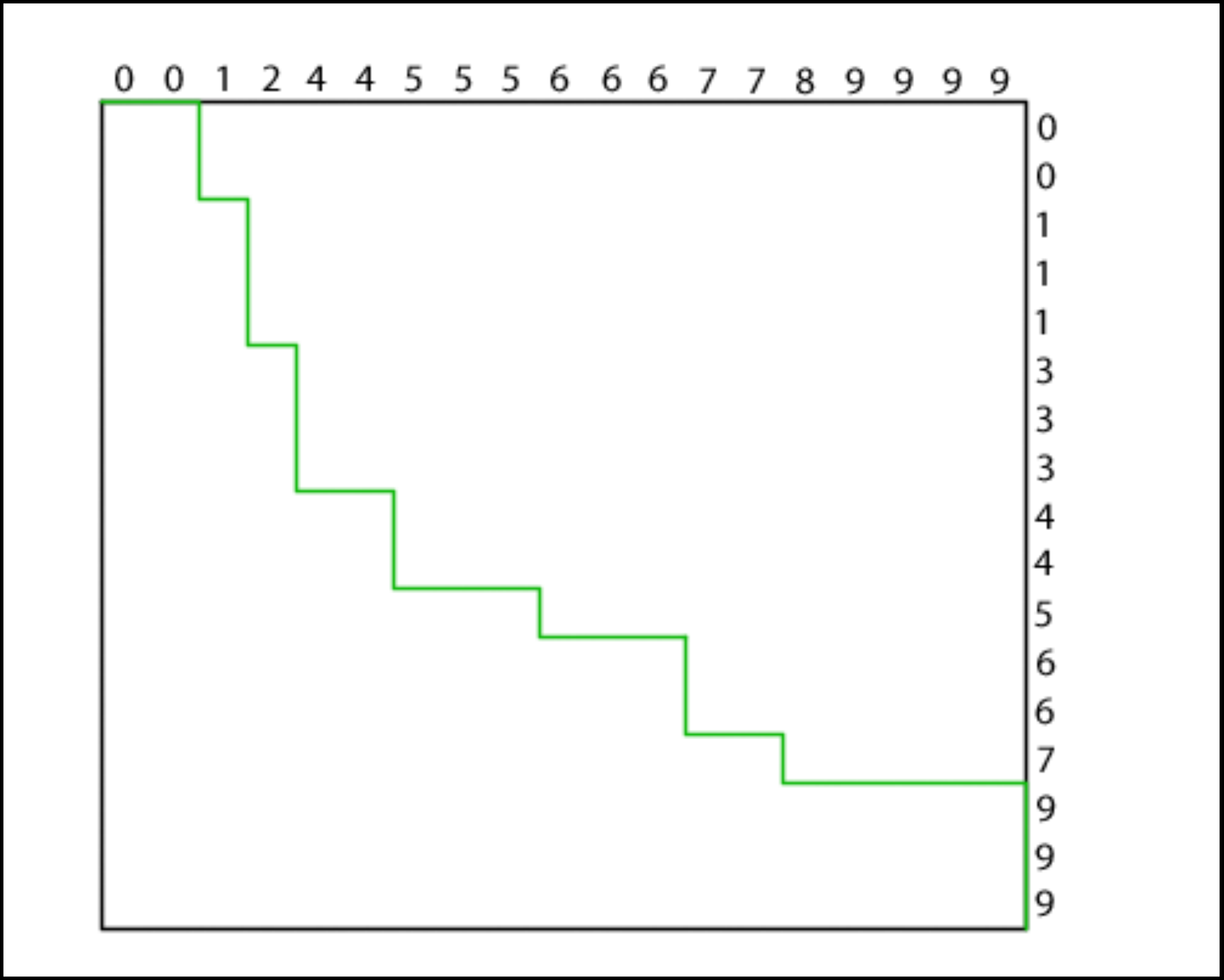
  2. WORK LOGIC

     Executes code specific for solving problem.

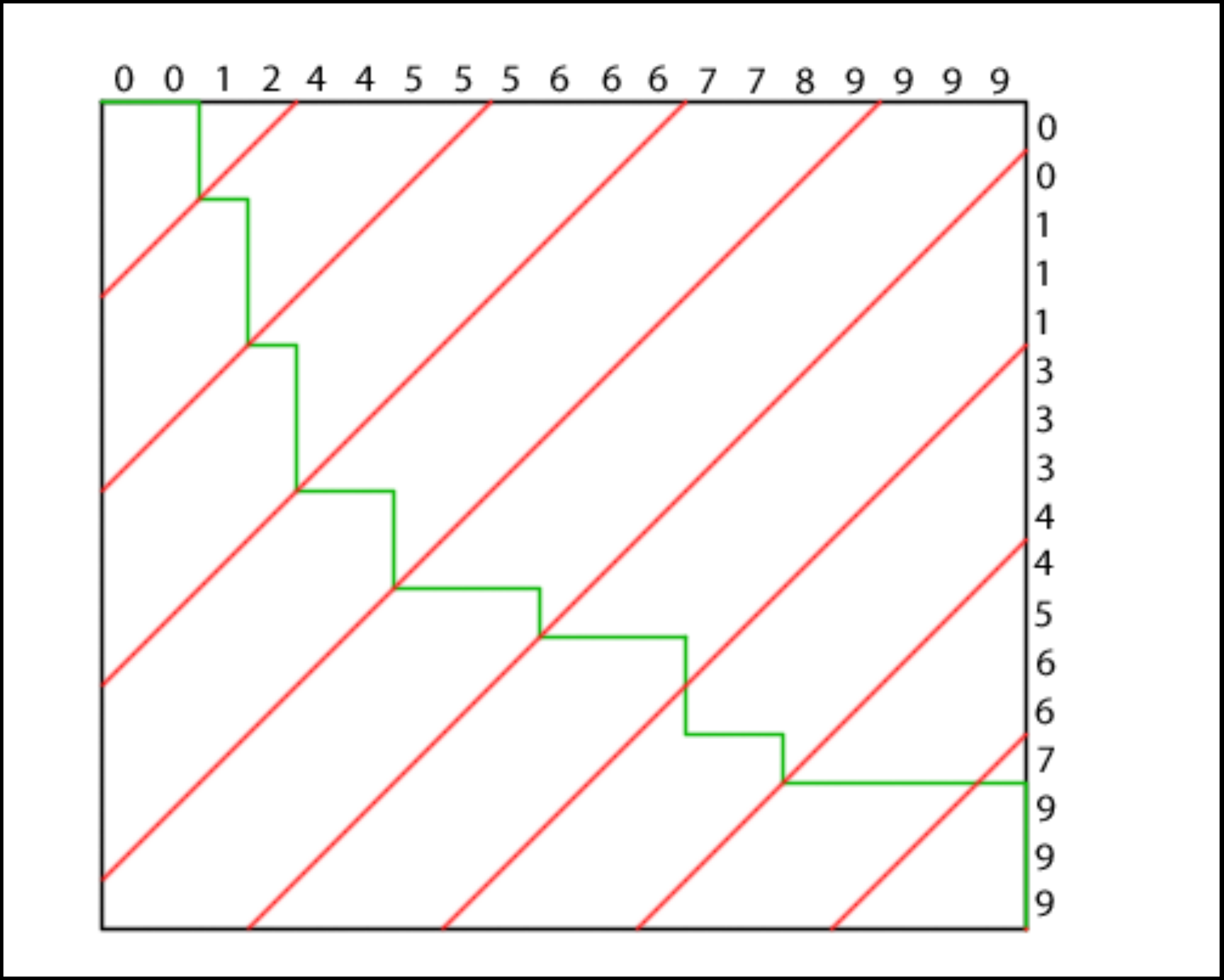
     This should resemble CPU sequential implementation.

     More work-efficient and more extensible.

Merge Path



Merge Path (2)



Binary Search

```
template<typename T, typename It, typename Comp>
    int BinarySearch(It data, int count, T key, Comp comp) {
    int begin = 0;
    int end = count;
    while(begin < end) {
        int mid = (begin + end)>> 1;
        bool pred = comp(key2, key);    // key <= key2.
        if(pred) begin = mid + 1;
        else end = mid;
    }
    return begin;
}
```

Standard binary search with support for C++ comparators. Searches for a key in one array.

Merge Path (3)

```
template<typename T, typename It1, typename It2, typename Comp>
int MergePath(It1 a, int aCount, It2 b, int bCount, int diag,
    Comp comp) {
    int begin = max(0, diag - bCount);
    int end = min(diag, aCount);

    while(begin < end) {
        int mid = (begin + end)>> 1;
        bool pred = !comp(b[diag - 1 - mid], a[mid]);
        if(pred) begin = mid + 1;
        else end = mid;
    }
    return begin;
}
```

Merge Path binary search with support for C++ comparators. Simultaneously search two arrays by using constraint ai + bi = diag to make problem one dimensional.

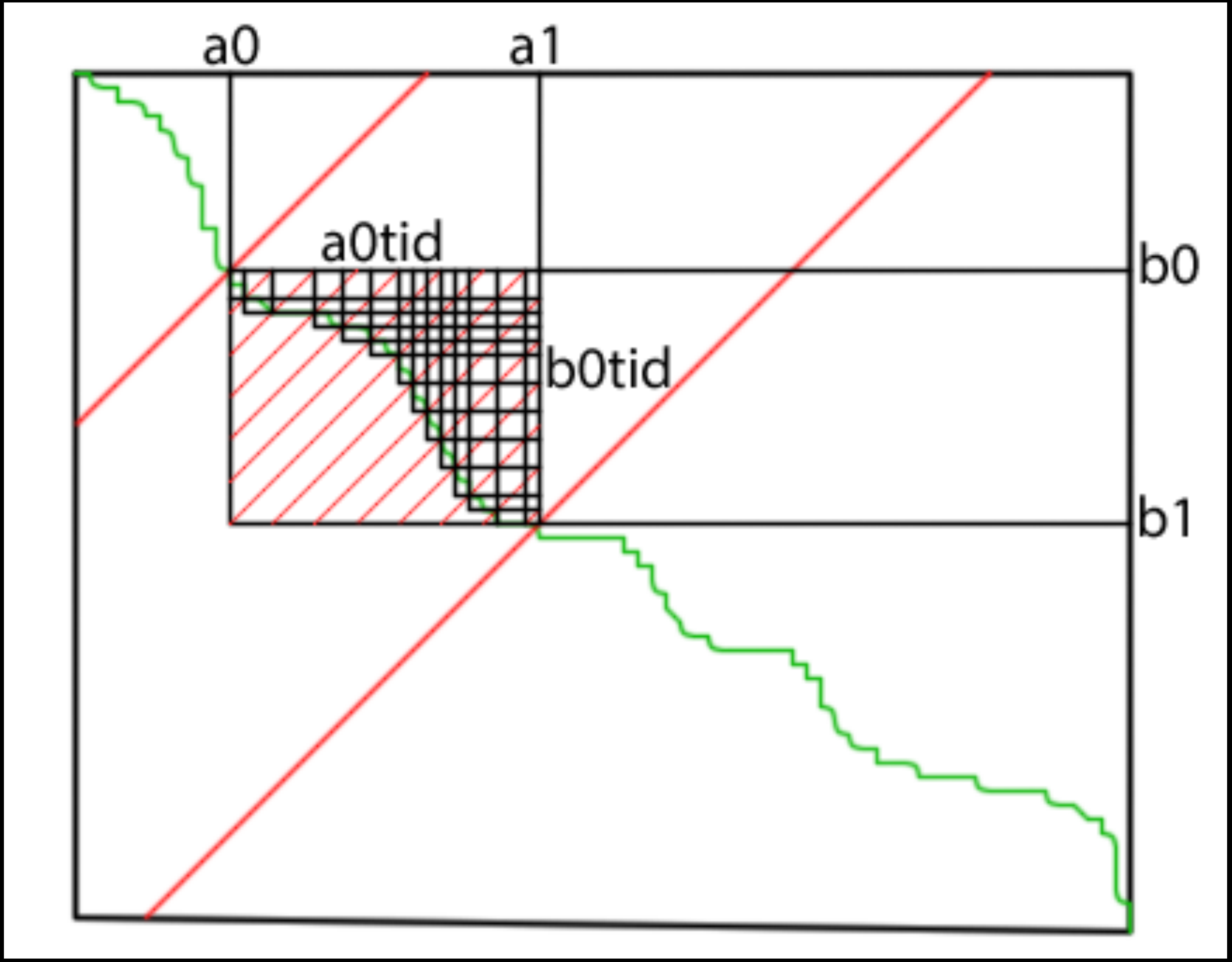
Parallel Merge (4)

1. Partitioning phase:
   1. Use MergePath search to partition input arrays into CTA-sized intervals.
   2. In CTA, load interval from A and interval from B.
   3. Run MergePath in CTA into shared memory to partition work per thread.

2. Work phase:

   Use SerialMerge to merge VT elements per thread without communication.

Parallel Merge (5)

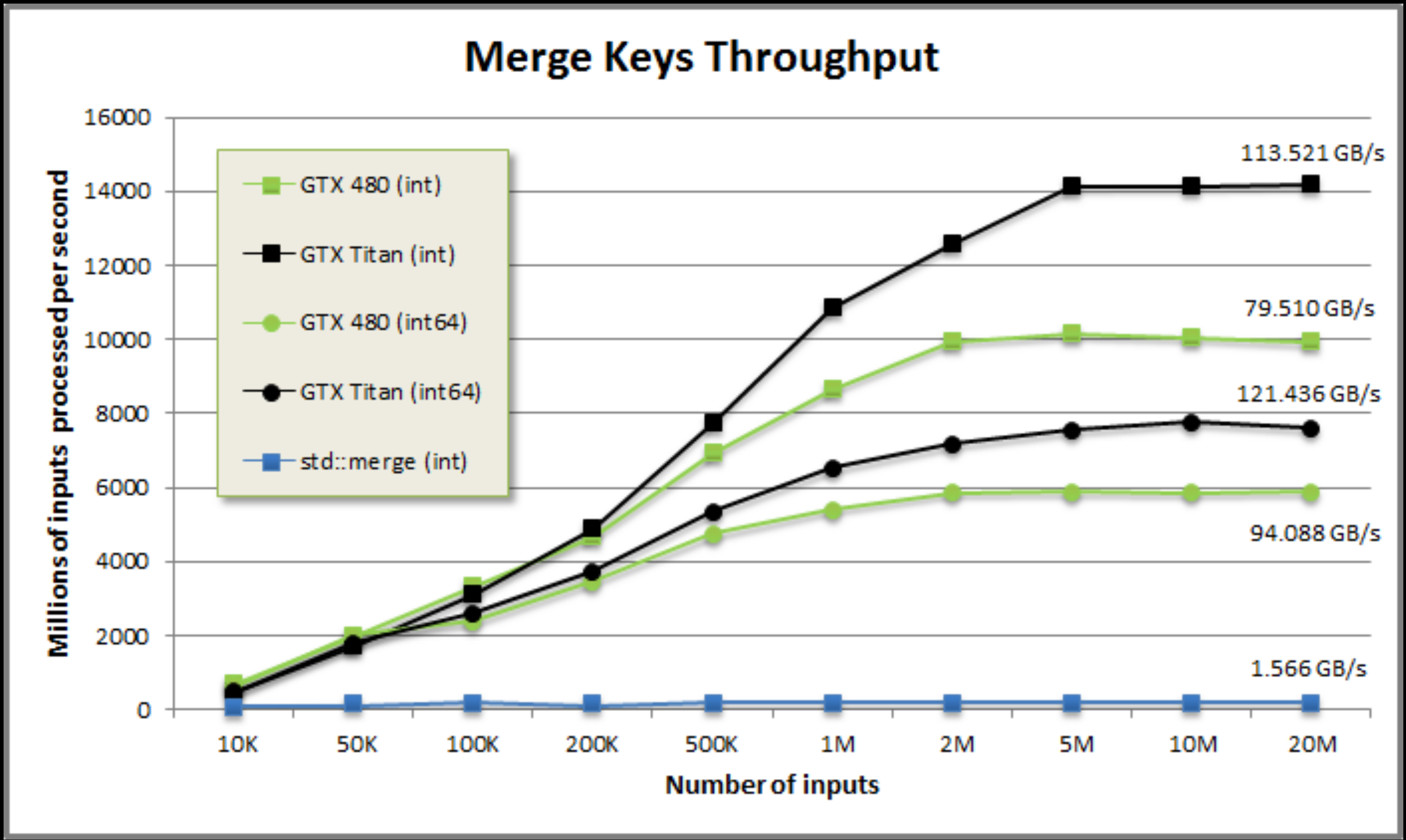


Serial Merge

```
#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 results is kept in RF, not smem!
        results[i] = p ? x1 : x2;
        if(p) ++aBegin;
        else ++bBegin;
}
```

- **SerialMerge processes a *constant* number of elements per thread.**
- **Unroll loops and store to register.**

MGPU Merge throughput

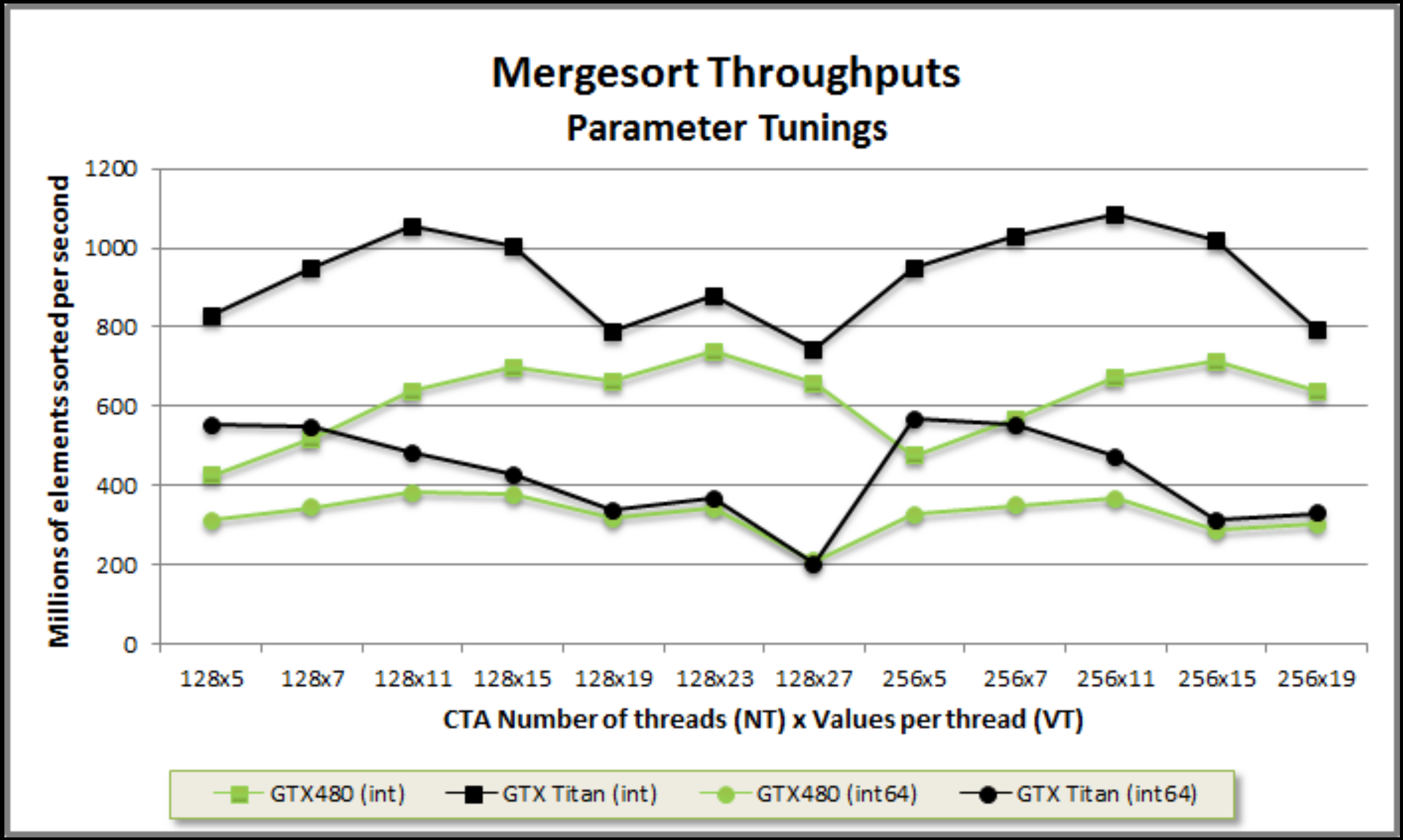


Parallel Merge (6)

- SerialMerge is looks like CPUMerge.
- It's work-efficient.
- It's easy to extend.

- Two-phase design presents opportunity for tuning:
  - Search tuning space of grain size parameter VT.
  - Increase VT to do more serial work per search and improve work-efficiency (amortizes partitioning cost).
  - Decrease VT to increase occupancy and achieve better execution efficiency.

Tuning


Mergesort Throughputs
Parameter Tunings
Millions of elements sorted per second
1200
1000
800
600
400
200
0
128x5
128x7
128x11
128x15
128x19
128x23
128x27
256x5
256x7
256x11
256x15
256x19
CTA Number of threads (NT) x Values per thread (VT)
GTX480 (int)
GTX Titan (int)
GTX480 (int64)
GTX Titan (int64)

Tuning (2)

- **Kepler has a wider SM-needs more parallelism to keep execution units fed.**
- **Fermi has more shared memory per execution unit to hide instruction latency. Choose larger grain size beneficial.**

	GTX 480 (Fermi)	GTX Titan (Kepler)
32-bit int	**128**x**23**	**256**x**11**
64-bit int	**128**x**11**	**256**x**5**

Merge-like Functions

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

Vectorized sorted search

- Unusual new primitive.
- Searches array of sorted keys into array of sorted haystack.
- Simple usage: return lower-bound (or upper-bound) of A into B.
- Power usage:
  - Lower-bound of A into B.
  - Upper-bound of B into A.
  - Flags for all matches of A into B.
  - Flags for all matches of B into A.
  - All this with a single pass over data!

Vectorized sorted search (2)

```
// Return lower-bound of A into B.
template<typename T, typename Comp>
void CPUSortedSearch(const T* a, int aCount, const T* b,
    int bCount, int* indices, Comp comp) {

    int aIndex = 0, bIndex = 0;
    while(aIndex < aCount) {
        bool p;
        if(bIndex >= bCount) p = true;
        else p = !comp(b[bIndex], a[aIndex]);

        if(p) indices[aIndex++] = bIndex;
        else ++bIndex;
    }
}
```

Looks just like CPUMerge!

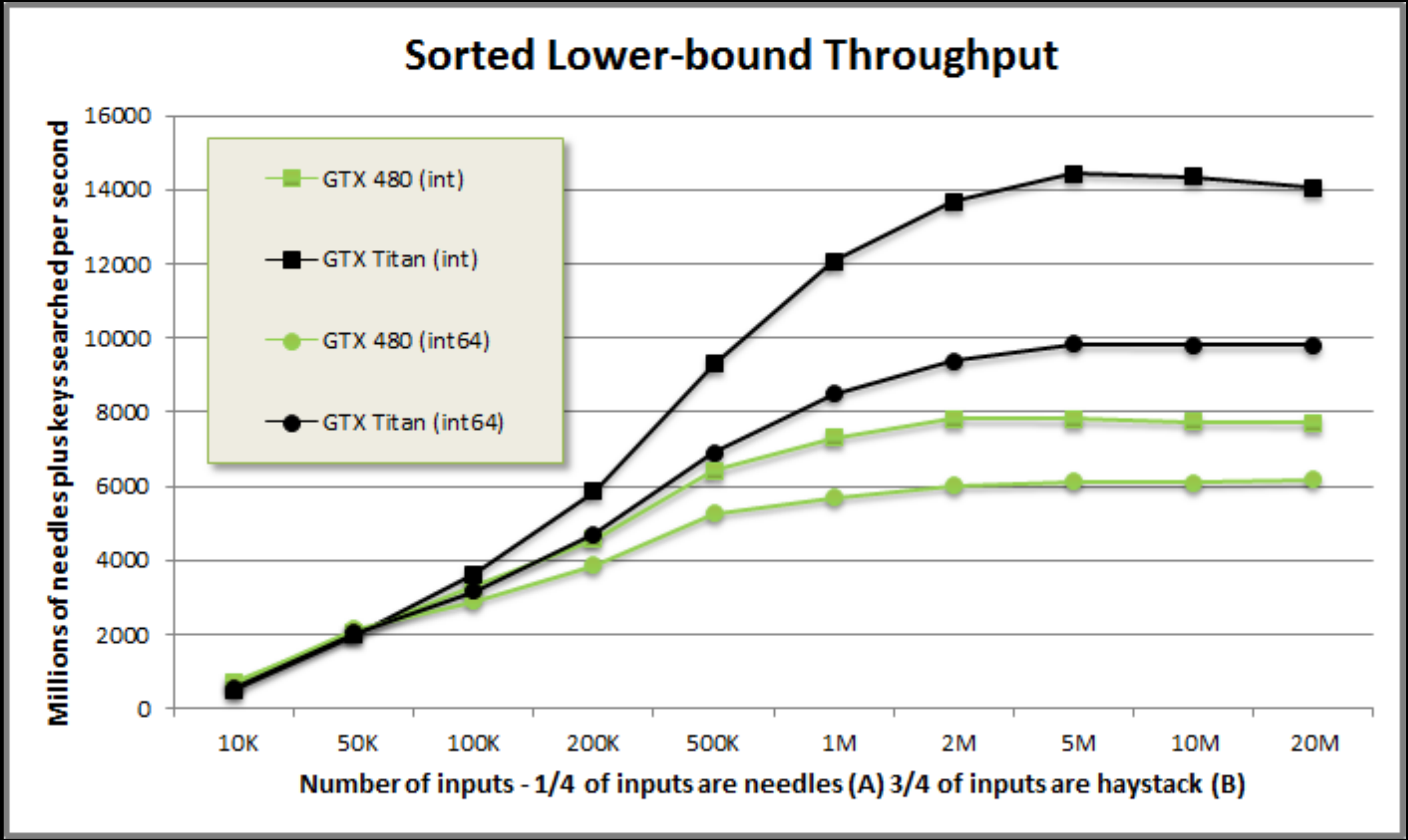
Vectorized sorted search (3)

- **Parallelize sorted search just like we parallelized merge.**
- **Use same MergePath code for partitioning problem.**
- **Use different serial work logic code to specialize for this problem.**
  - **Two-phase decomposition promotes code reuse.**
- **Coarse-grained partitioning maps a fixed amount of *needles plus haystack* into each CTA.**
- **Fine-grained partitioning maps a fixed amount of *needles plus haystack* into each thread.**

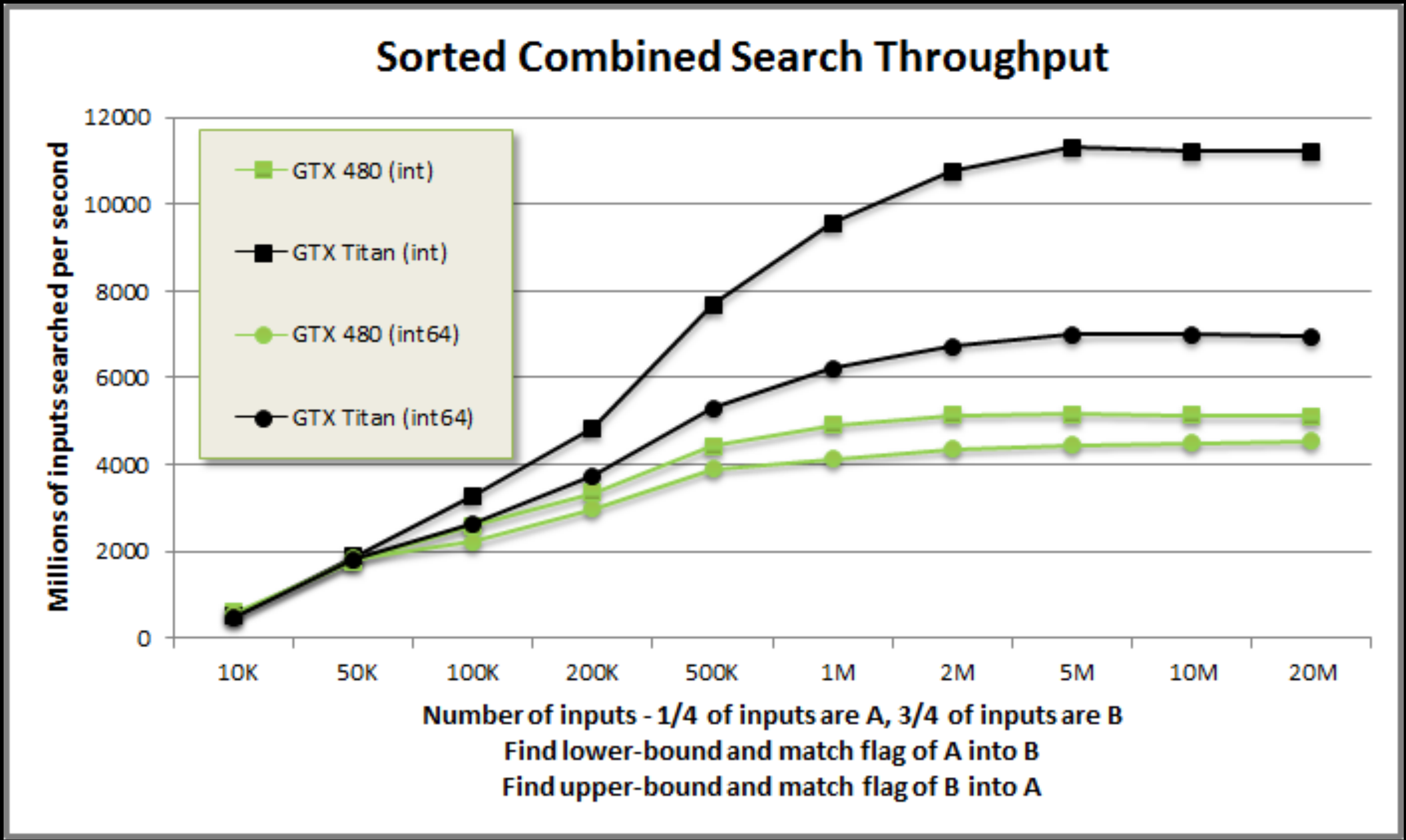
SerialSearch

```
template<int VT, typename T, typename Comp>
MGPU_DEVICE int DeviceSerialSearch(const T* keys_shared,
    int aBegin, int aEnd, int bBegin, int bEnd, int bOffset,
    int* indices, Comp comp) {

    int decisions = 0;
    #pragma unroll
    for(int i = 0; i < VT; ++i) {
        bool p;
        if(aBegin >= aEnd) p = false;
        else if(bBegin >= bEnd) p = true;
        else p = !comp(keys_shared[bBegin], keys_shared[aBegin]);

        if(p) {
            decisions |= 1<< i;  // Set bit to indicate result
            indices[i] = bOffset + bBegin;
            ++aBegin;
        } else ++bBegin;
    }
    return decisions;
}
```

Vectorized sorted search throughput



Vectorized sorted search throughput



Load-balancing search

- Load-balancing search is a special decomposition
- … Or a change of coordinates
- … Or a kind of inverse of prefix sum

- N objects
  - Each object generates variable outputs
  - Match each output with the generating object.

Load-balancing search (2)

```
Work-item counts:
  0:     1    2    4    0    4    4    3    3    2    4
 10:     0    0    1    2    1    1    0    2    2    1
 20:     1    4    2    3    2    2    1    1    3    0
 30:     2    1    1    3    4    2    2    4    0    4

Exc-scan of counts:
  0:     0    1    3    7    7   11   15   18   21   23
 10:    27   27   27   28   30   31   32   32   34   36
 20:    37   38   42   44   47   49   51   52   53   56
 30:    56   58   59   60   63   67   69   71   75   75


Total work-items: 79
```

Load-balancing search (3)

```
Load-balancing search:
    0:     0    1    1    2    2    2    2    4    4    4
   10:     4    5    5    5    5    6    6    6    7    7
   20:     7    8    8    9    9    9    9   12   13   13
   30:    14   15   17   17   18   18   19   20   21   21
   40:    21   21   22   22   23   23   23   24   24   25
   50:    25   26   27   28   28   28   30   30   31   32
   60:    33   33   33   34   34   34   34   35   35   36
   70:    36   37   37   37   37   39   39   39   39

Work-item rank:
    0:     0    0    1    0    1    2    3    0    1    2
   10:     3    0    1    2    3    0    1    2    0    1
   20:     2    0    1    0    1    2    3    0    0    1
   30:     0    0    0    1    0    1    0    0    0    1
   40:     2    3    0    1    0    1    2    0    1    0
   50:     1    0    0    0    1    2    0    1    0    0
   60:     0    1    2    0    1    2    3    0    1    0
   70:     1    0    1    2    3    0    1    2    3
```

Load-balancing search (4)

- Each output is paired with its generating object
- A rank for the work-item within the generating object is also returned.

- LBS is computed as the *upper-bound of the natural numbers into the scan of the work-item counts minus 1*.

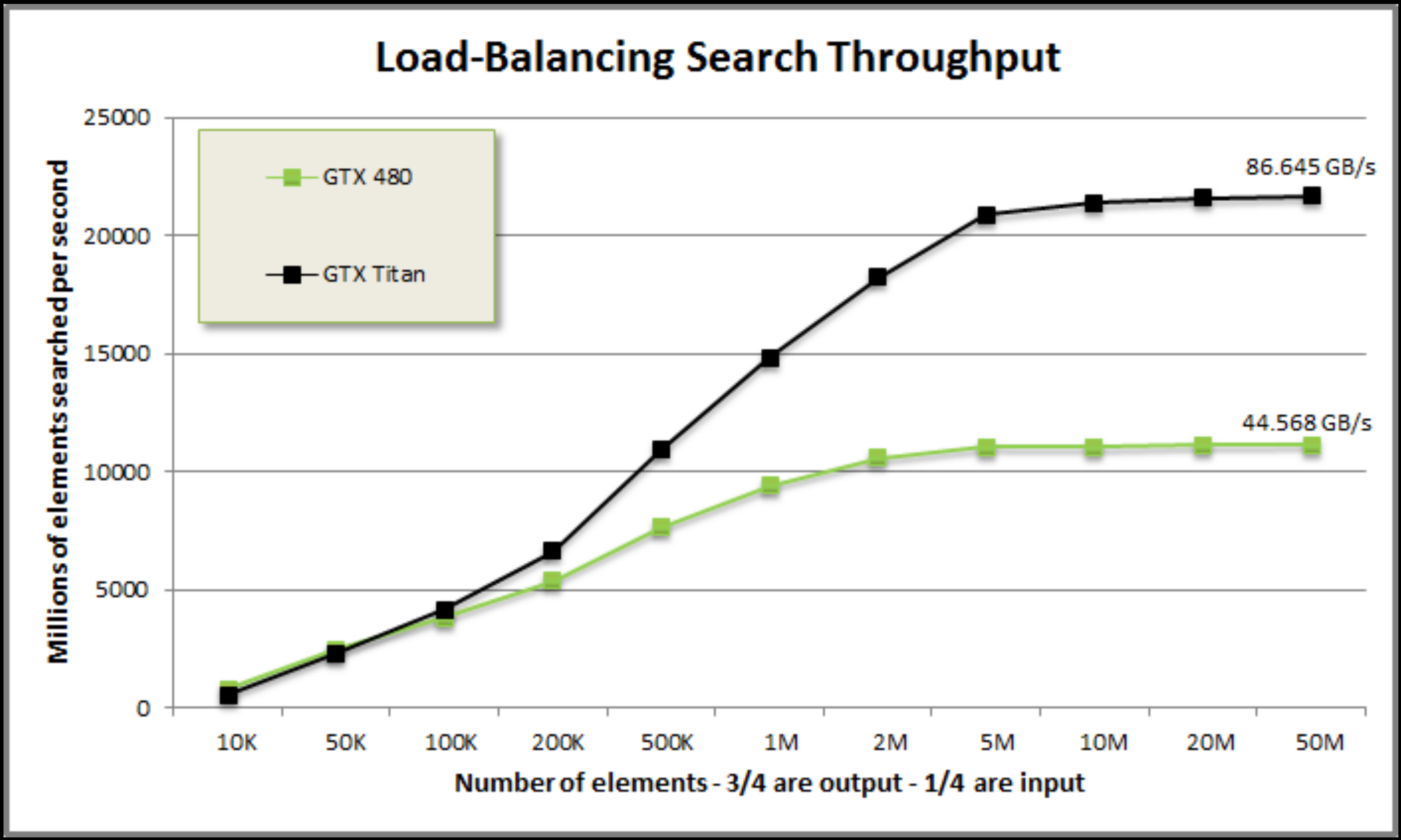
- Use vectorized sorted search (upper-bound) pattern with some optimizations.

- Use ordinary Merge Path search for coarse-grained partitioning.

Load-balancing search (5)

```
template<int VT>
MGPU_DEVICE void DeviceSerialLoadBalanceSearch(
    const int* b_shared, int aBegin, int aEnd, int bFirst,
    int bBegin, int bEnd, int* a_shared) {

    int bKey = b_shared[bBegin];

    #pragma unroll
    for(int i = 0; i < VT; ++i) {
        bool p = (aBegin < aEnd) &&
            ((bBegin >= bEnd) || (aBegin < bKey));

        if(p)       // Advance A (the needle).
            a_shared[aBegin++] = bFirst + bBegin;
        else        // Advance B (the haystack).
            bKey = b_shared[++bBegin];
    }
}
```

Load-balancing search throughput



Load-balancing search (6)

- Run CTALoadBalance inside kernel as boilerplate.
- This transforms the problem so that its dependencies are explicit.
- Great for implementing functions that *expand* or *contract* data:
  - IntervalMove (vectorized cudaMemcpy)
  - Relational joins
  - Sparse matrix * matrix (expand partials)

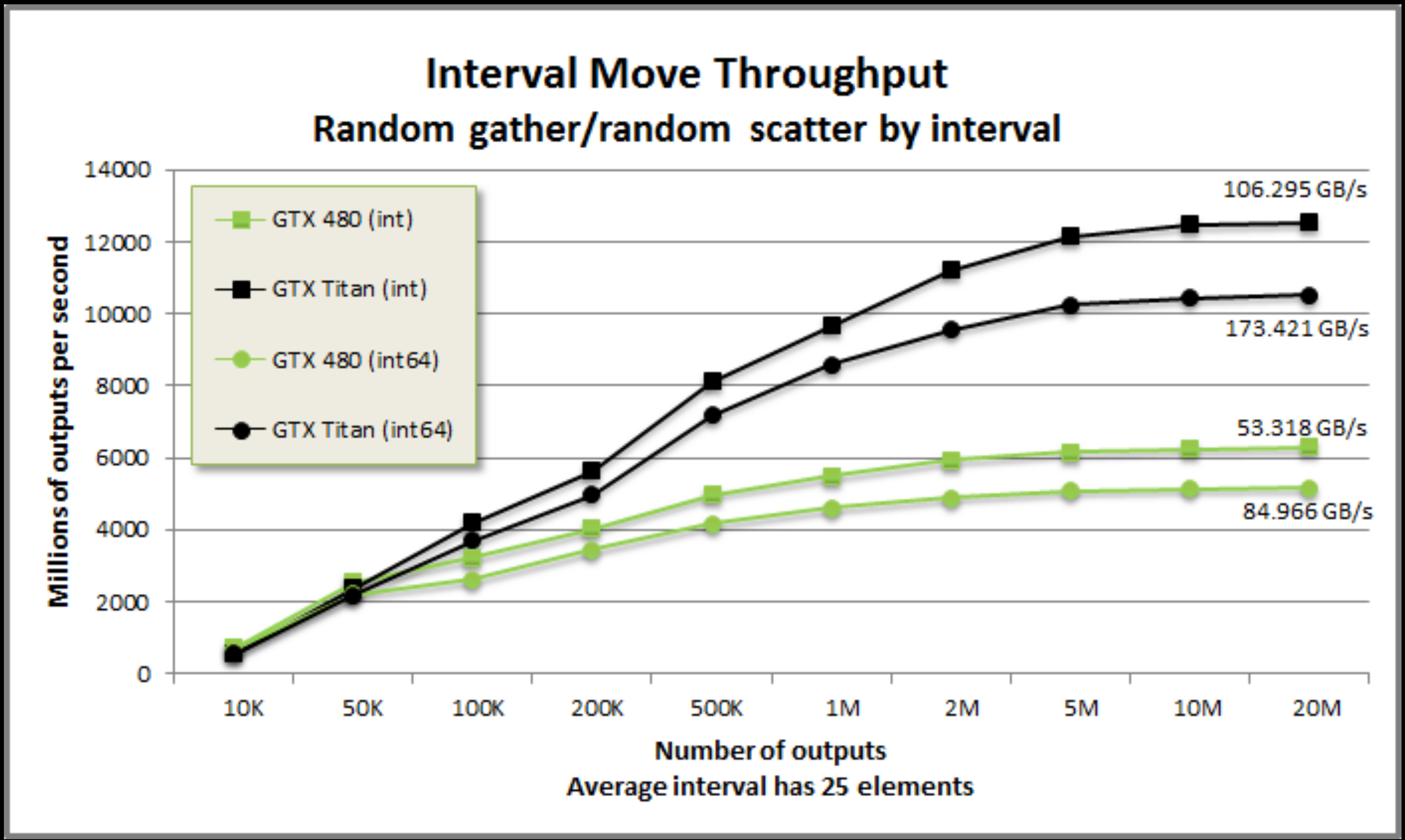
IntervalMove

- Many small cudaMemcpy calls:
  - Too slow.
- One kernel:
  - Batch up many cudaMemcpy's by gather (source), scatter (destination), and size parameters, and fire off in a single launch.
- Scan interval counts and use *load-balancing search* to map output elements into input intervals.
- Launch enough KernelIntervalMove threads to cover both the outputs *and* inputs.

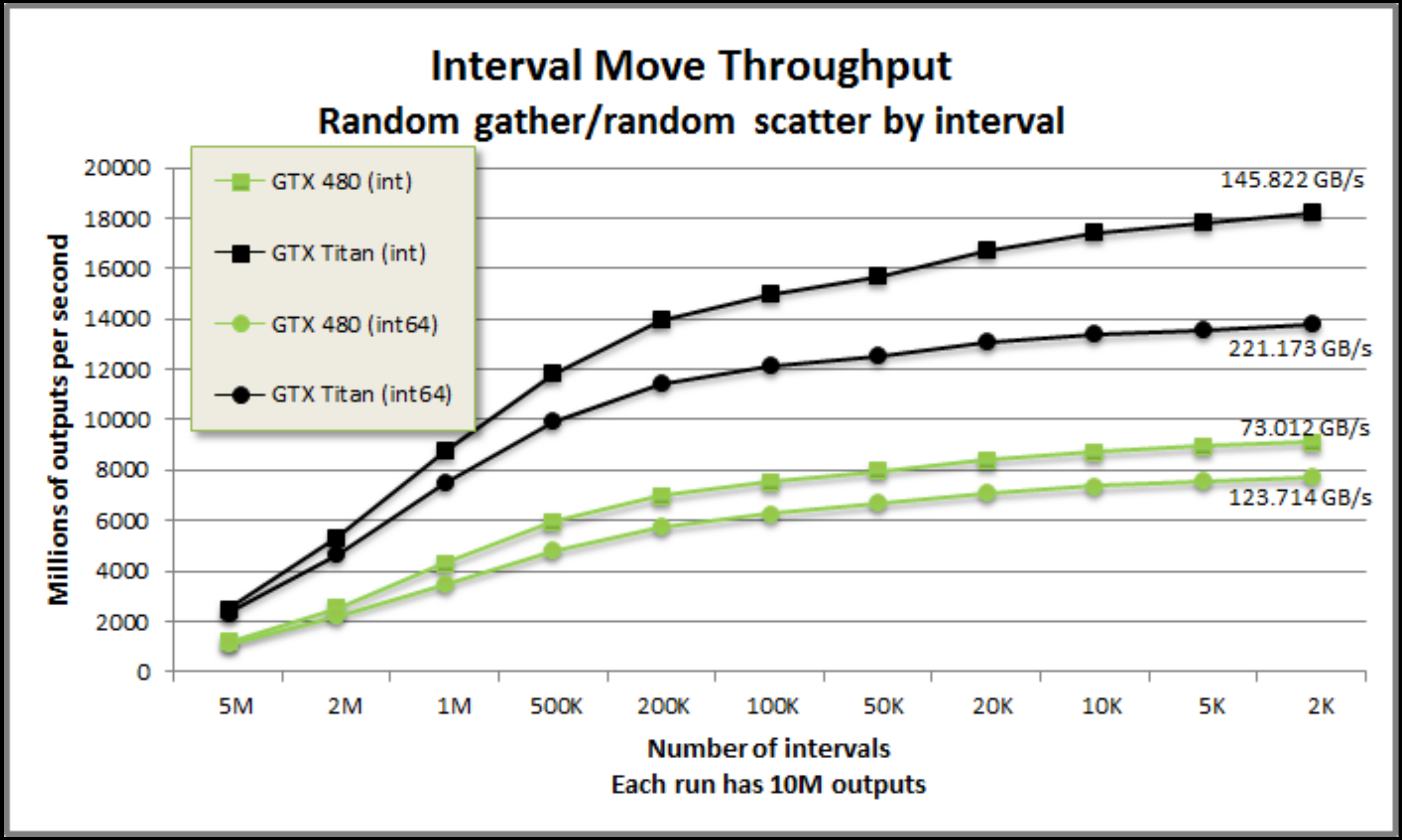
IntervalMove (2)

- CTALoadBalance inside KernelIntervalMove maps up to VT outputs per thread into the interval that generated it.
- The caller infers the rank of each output within the interval to know which element within the interval to copy.
- Because of Merge Path’s exact partitioning, each thread copies no more than VT elements, no matter the distribution of intervals.

IntervalMove throughput



IntervalMove throughput



Relational Joins

- **MGPU implements full-outer join on GPU with exact load-balancing.**

Row	A index	A key	B key	B index	Join type
0	0	A^0	A^0	0	inner
1	0	A^0	A^1	1	inner
2	1	A^1	A^0	0	inner
3	1	A^1	A^1	1	inner
4	2	B^0	B^0	2	inner
5	2	B^0	B^1	3	inner
6	2	B^0	B^2	4	inner
7	3	E^0	---	-1	left
8	4	E^1	---	-1	left
9	5	E^2	---	-1	left
10	6	E^3	---	-1	left
11	7	F^0	F^0	7	inner
12	8	F^1	F^0	7	inner
13	9	G^0	G^0	8	inner
14	9	G^0	G^1	9	inner
15	10	H^0	H^0	10	inner
16	11	H^1	H^0	10	inner
17	12	J^0	---	-1	left
18	13	J^1	---	-1	left
19	14	M^0	---	-1	left
20	15	M^1	---	-1	left
21	-1	---	C^0	5	right
22	-1	---	C^1	6	right
23	-1	---	I^0	11	right
24	-1	---	L^0	12	right
25	-1	---	L^1	13	right

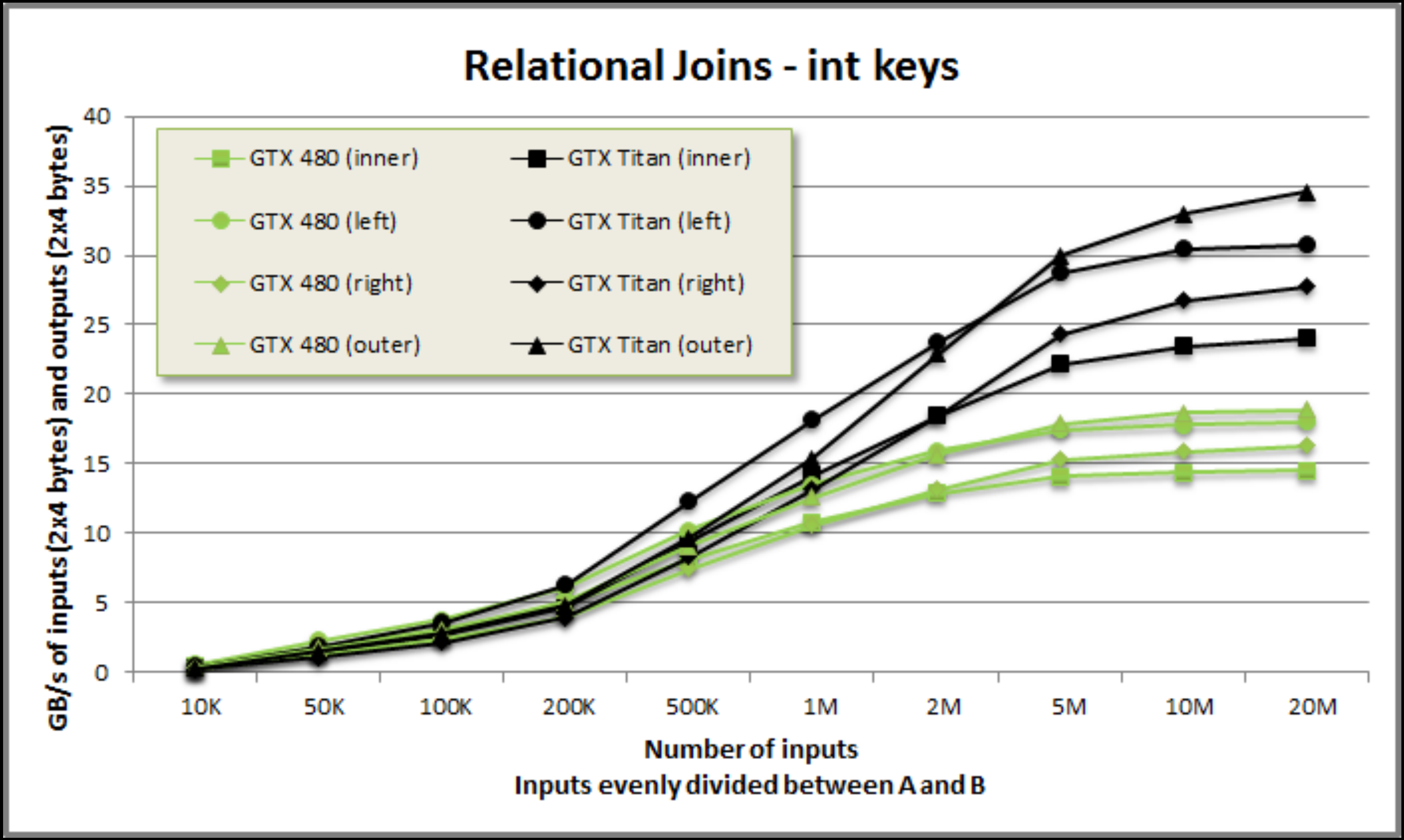
Relational Join (2)

- We join two sorted tables (sort-merge join).
- Equal keys in A and B are expanded with Cartesian product.
- Keys in A not found in B are emitted with left-join (null B key).
- Keys in B not found in A are emitted with right-join (null A key).

Relational Join (3)

- Vectorized lower-bound and upper-bound search finds set of matches for each row in A in the B table.
- Difference of upper- and lower-bound iterators are match *counts*.
- Scan match counts.
- Use load-balancing search

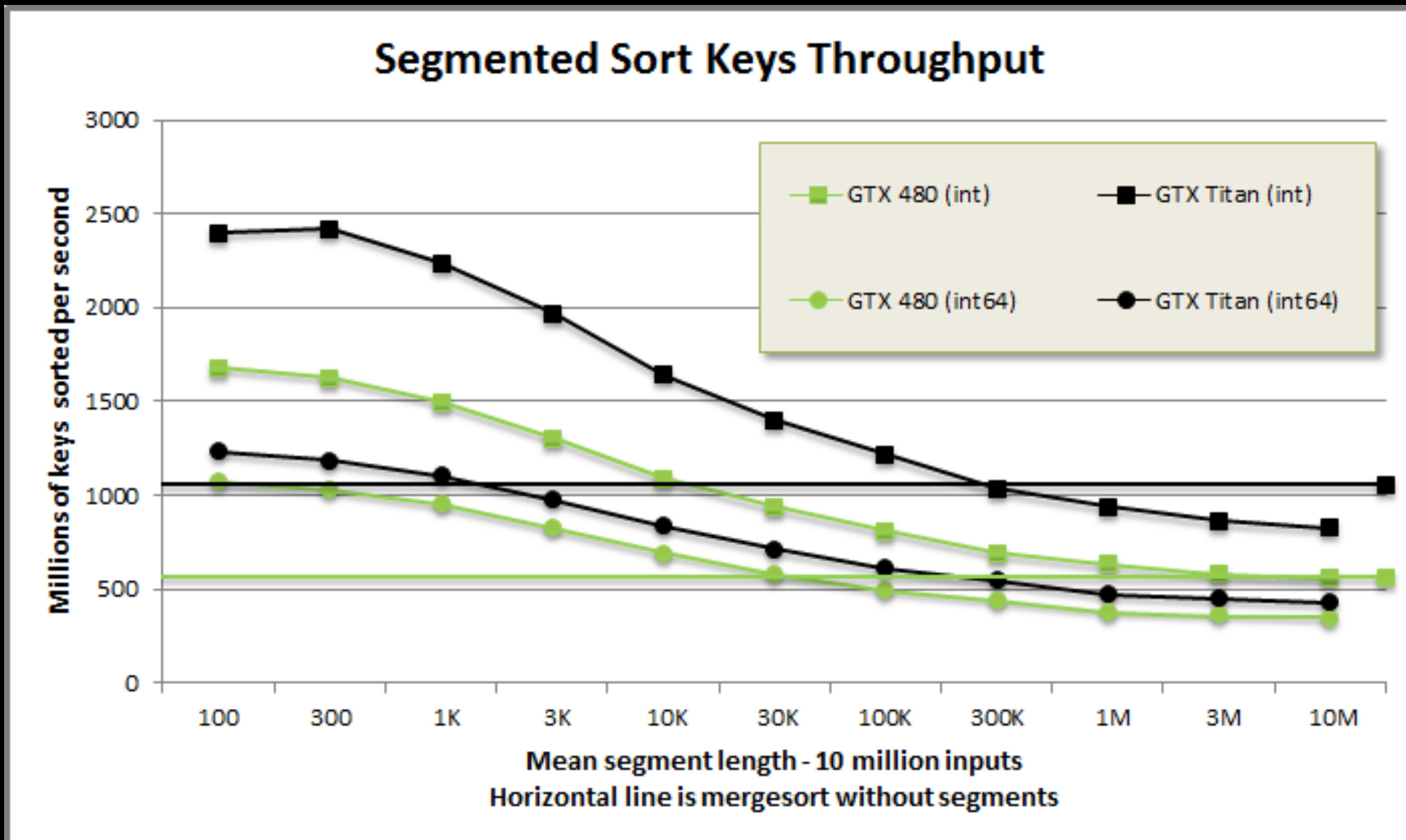
Relational join throughputs



Segmented sort

- MGPU Merge implements MGPU Mergesort.
- Extend Merge Path decomposition to segments for MGPU Segmented Sort.
- Simultaneously sort many variable-length arrays.
- Without compromising concurrency, code improves over O(n log n) complexity by detecting segment structure and early-exiting.

Segmented sort throughput



Wrap-up

- Separate partitioning from work logic.
- Expose grain size in partitioning for tuning.
- Write work-efficient, sequential, communication-free code to implement function.
- Exploit *exact partitioning*
  - Unroll all loops.
  - Read from shared memory and store to register.
  - Reduce smem consumption and boost occupancy.

Wrap-up (2)

- Use load-balancing search to accommodate expansion of contraction of data from per-object counts.
- Use CTALoadBalance to embed load-balancing search (itself a two-phase vectorized upper-bound search) as boilerplate in kernels.
- Use IntervalMove (or IntervalGather or IntervalScatter) to queue up and launch many cudaMemcpy operations at once.

Questions?

http://nvlabs.github.io/moderngpu website

https://www.github.com/nvlabs/modergpu code

fork me!

sbaxter@nvidia.com email

@moderngpu twitter

http://www.moderngpu.com/stanford.pptx slides