Binary Scan Encoding with Backward Application

DOMINIK LANGER & NIKLAS SUMM

GPU ALGORITHM DESIGN

27.02.2025

Content

- Introduction
  - Algorithm
  - Thrust Baseline
  - 3 Tree Optimizations
- Analysis
  - Setup Tree
  - Apply Tree
  - Sparsity
  - Memory
  - Profiling
- Discussion
- References
- Appendix

- Provided is a sparse bitmask
- Based on bitmask we want to pack/unpack an array
- To allow for random access, we need to save the scan
- Saving full permutation requires 32x / 64x the bitmask memory (depends on index type)
- Optimization: Only store parts of the scan, reducing the memory requirements

This Project: Map from packed to expanded index
(inverted permutation)

- Stores the full inverted permutation, memory footprint depends a lot on sparsity
- Bitmask is expanded to a boolean iterator
- Setup uses `thrust::copy_if` to create inverted permutation in device memory
- Apply re-uses this permutation with
  - `thrust::gather` for pack
  - `thrust::scatter` for unpack

- Instead of full permutation, store only a subset of the scan
- Store values in tree, summarising sections from the next lower level
- First layer after bitmask can use a smaller data type to reduce memory
- Setup stores tree once, apply can re-use tree multiple times
- Tree structure is based on expanded index, requires binary search to map packed index

Fixed Inclusive

- Two layers in addition to bitmask
- Variable reduction steps between layers (configurable)
- Values are taken from inclusive scan

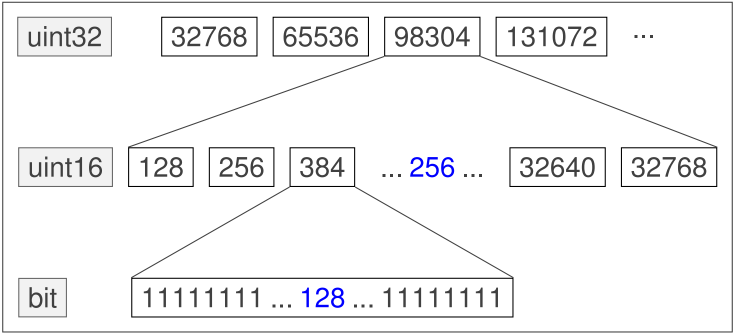

Figure 1: Structure of the fixed inclusive tree with steps of 2^7 and 2^8. Maximum values assuming dense bitmask are shown. The numbers in blue indicate the amount of elements per section.

Fixed Exclusive

- Two layers in addition to bitmask
- Variable reduction steps between layers (configurable)
- Values are taken from exclusive scan to better use 16 bit range

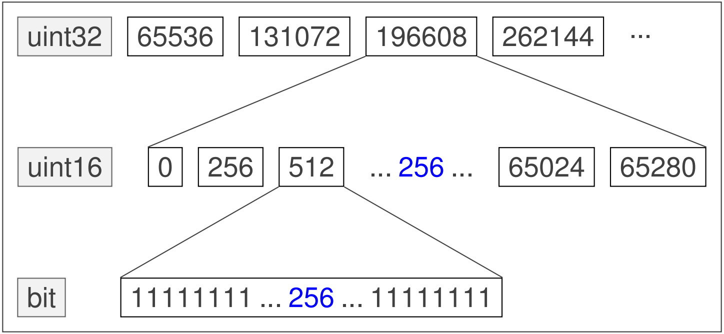

Figure 2: Structure of the fixed exclusive tree with twice 2^8 steps. Maximum values assuming dense bitmask are shown. The numbers in blue indicate the amount of elements per section.

Dynamic Exclusive

- Variable layer count, depending on bitmask size
- Fixed steps of 2^5 between all stored layers
- Values are taken from exclusive scan to better use 16 bit range

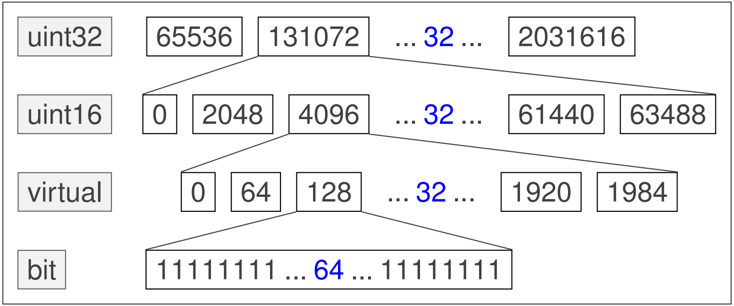

Figure 3: Structure of the dynamic exclusive tree. Maximum values assuming dense bitmask are shown. The numbers in blue indicate the amount of elements per section. Only first 2 stored layers shown.

- Warp can use collaborative descend based on ballot vote
- Reduces the instructions needed for denser bitmasks
- Divergence check switches warp to individual binary search

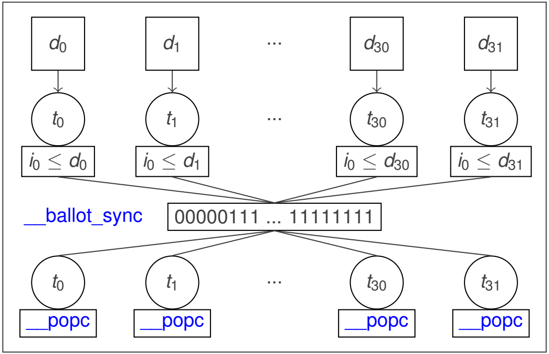

Figure 4: Collaborative tree descend within one warp using a ballot vote. Not shown is a subsequent divergence check, that switches the warp to a per-thread binary search descend.

- NVIDIA GeForce RTX 4080
  - 76 SMs
  - 48.7 FP32 TFLOPS
  - 16 GB GDDR6X global memory
  - 716.8 GB/s global memory bandwidth
  - Max thread block size: 1024
- CUDA 12.8
- Running 10 iterations
- First iteration excluded as warm-up
- L2 cache cleared between iterations
- If not stated otherwise:
  - 128 threads / block
  - 50% sparse bitmask
  - Setup bandwidth relative to bitmask size
  - Apply bandwidth relative to expanded size (using array of int)

(4) GeForce RTX 4080 Family, (5) Nvidia GeForce RTX 4080 Review

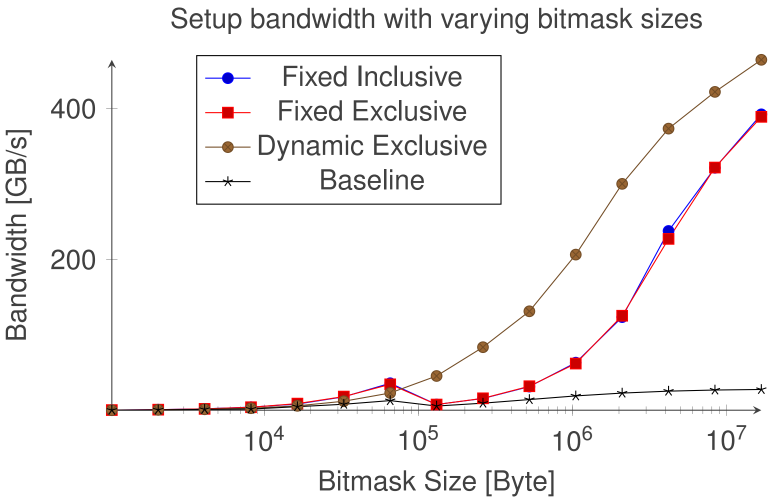

Figure 5: Setup bandwidth against varying bitmask sizes for all 3 optimizations and the baseline. Bandwidth is measured based on bitmask size, not on expanded size.

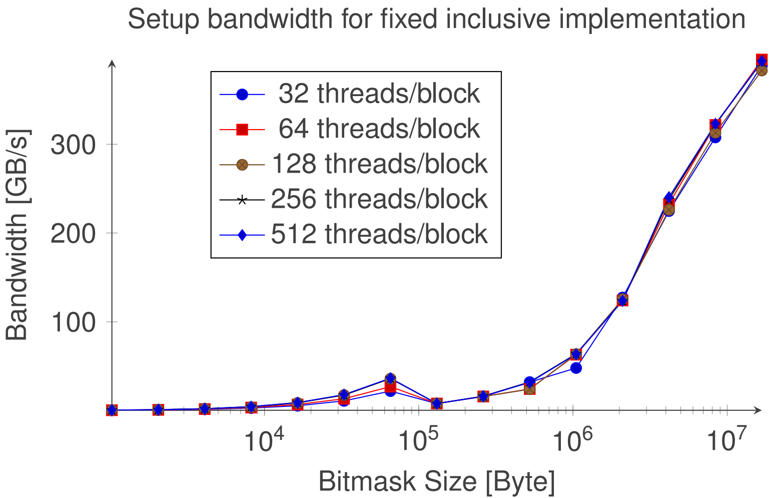

Figure 6: Setup bandwidth against varying thread block sizes for fixed inclusive implementation. Bandwidth is measured based on bitmask size, not on expanded size.

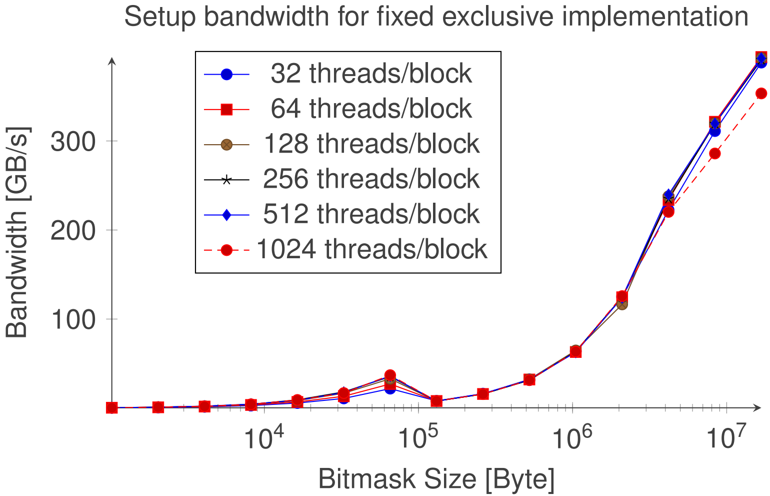

Figure 7: Setup bandwidth against varying thread block sizes for fixed exclusive implementation. Bandwidth is measured based on bitmask size, not on expanded size.

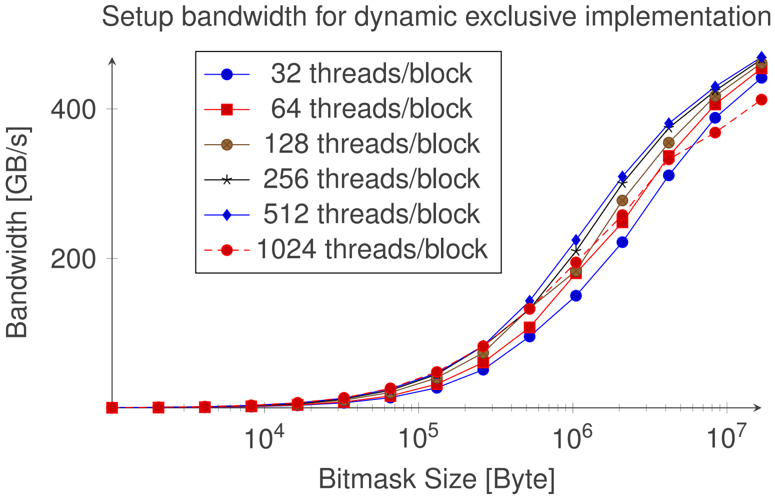

Figure 8: Setup bandwidth against varying thread block sizes for dynamic exclusive implementation. Bandwidth is measured based on bitmask size, not on expanded size.

Bandwidth heatmap fixed inclusive implementation

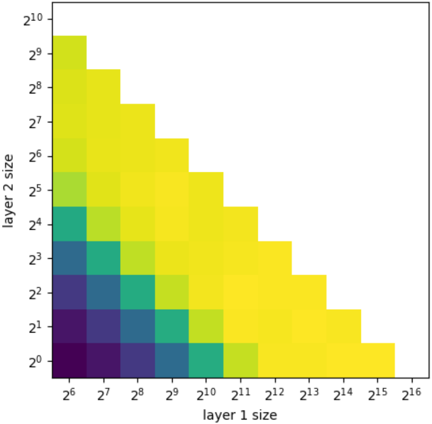

Figure 9: Setup bandwidth heatmap for varying layer sizes for fixed inclusive implementation. Blue means lower bandwidth, yellow higher bandwidth. Bitmask size 2^{28} elements.

Bandwidth heatmap fixed exclusive implementation

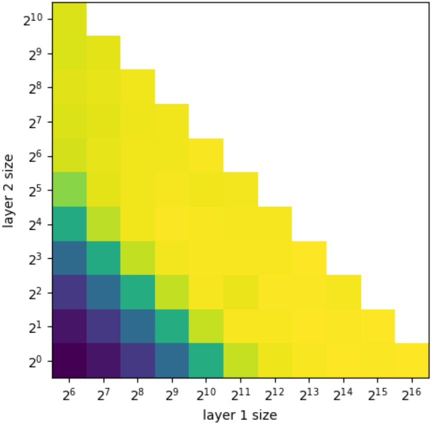

Figure 10: Setup bandwidth heatmap for varying layer sizes for fixed exclusive implementation. Blue means lower bandwidth, yellow higher bandwidth. Bitmask size 2^{28} elements.

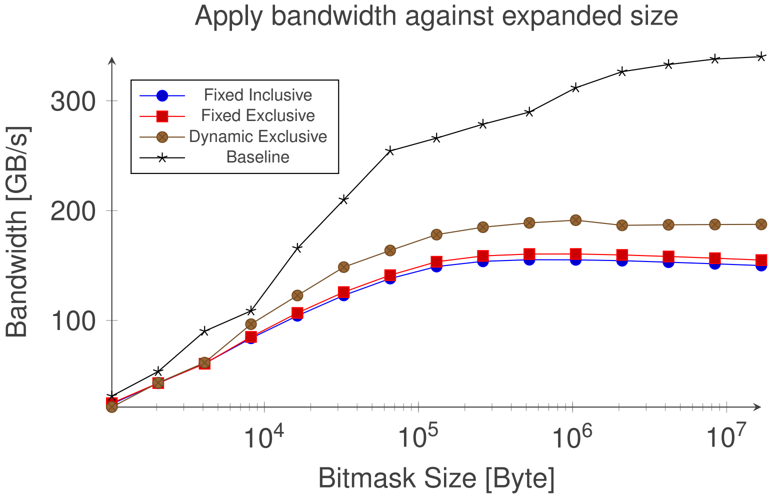

Figure 11: Apply bandwidth against varying bitmask sizes for all 3 optimizations and the baseline. Bandwidth is measured based on the expanded size with pack.

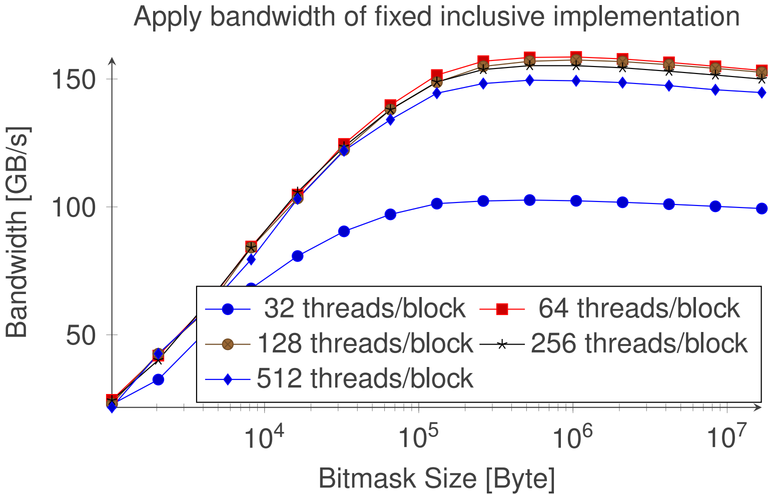

Figure 12: Apply bandwidth against varying thread block sizes for fixed inclusive implementation. Bandwidth is measured based on bitmask size, not on expanded size.

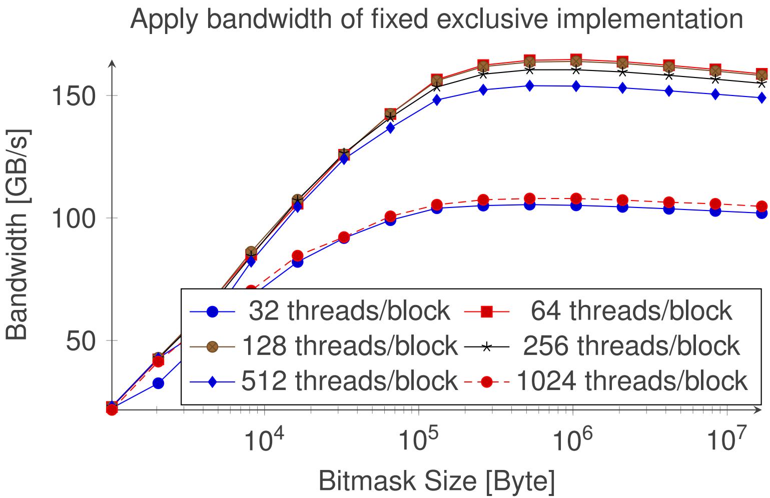

Figure 13: Apply bandwidth against varying thread block sizes for fixed exclusive implementation. Bandwidth is measured based on bitmask size, not on expanded size.

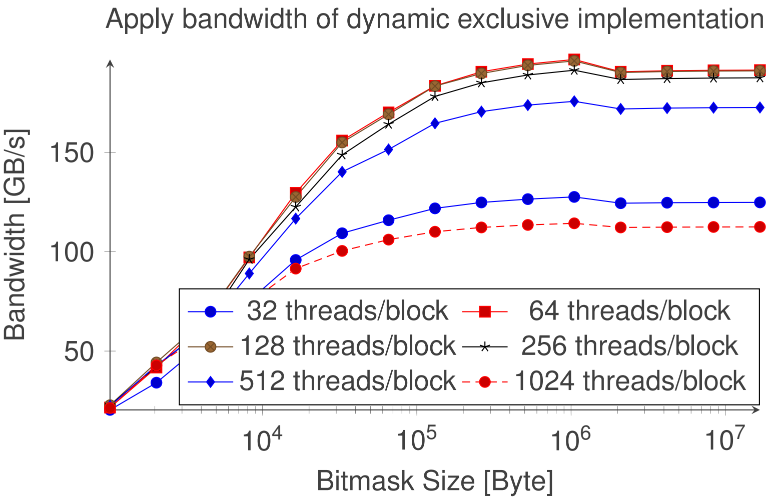

Figure 14: Apply bandwidth against varying thread block sizes for dynamic exclusive implementation. Bandwidth is measured based on bitmask size, not on expanded size.

Bandwidth heatmap fixed inclusive implementation

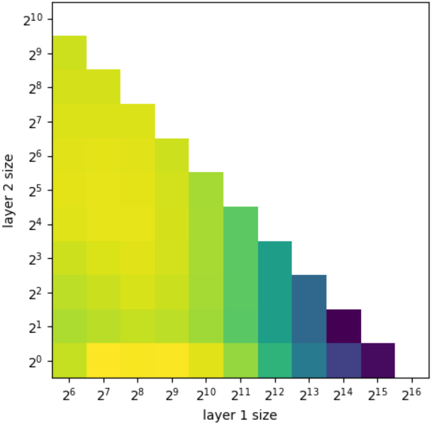

Figure 15: Apply bandwidth heatmap for varying layer sizes for fixed inclusive implementation. Blue means lower bandwidth, yellow higher bandwidth. Bitmask size 2^{28} elements.

Bandwidth heatmap fixed exclusive implementation

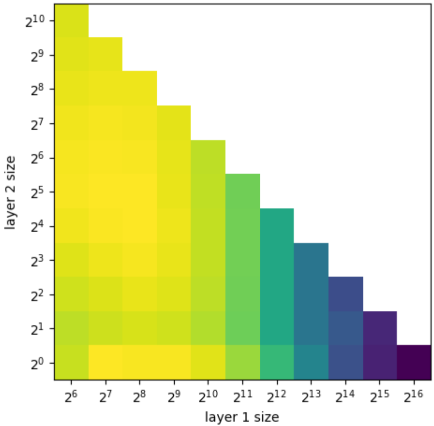

Figure 16: Apply bandwidth heatmap for varying layer sizes for fixed exclusive implementation. Blue means lower bandwidth, yellow higher bandwidth. Bitmask size 2^{28} elements.

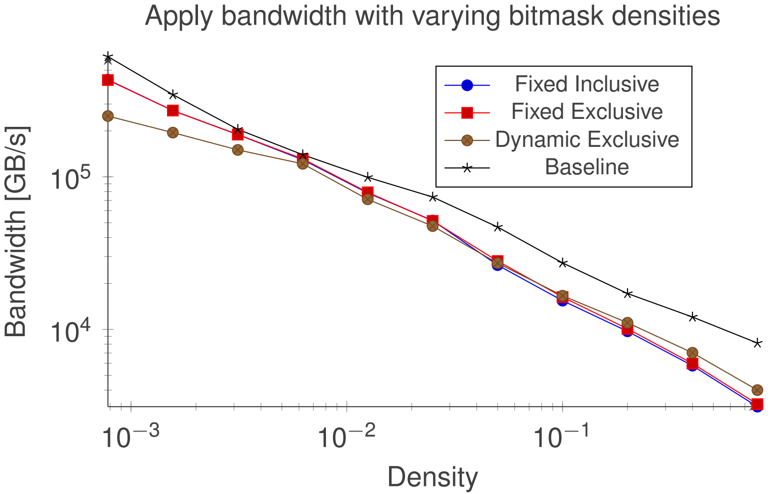

Figure 17: Apply bandwidth against varying bitmask densities for all 3 optimizations and the baseline. Bandwidth is measured based on the expanded size with pack. Bitmask size 2^{29} elements.

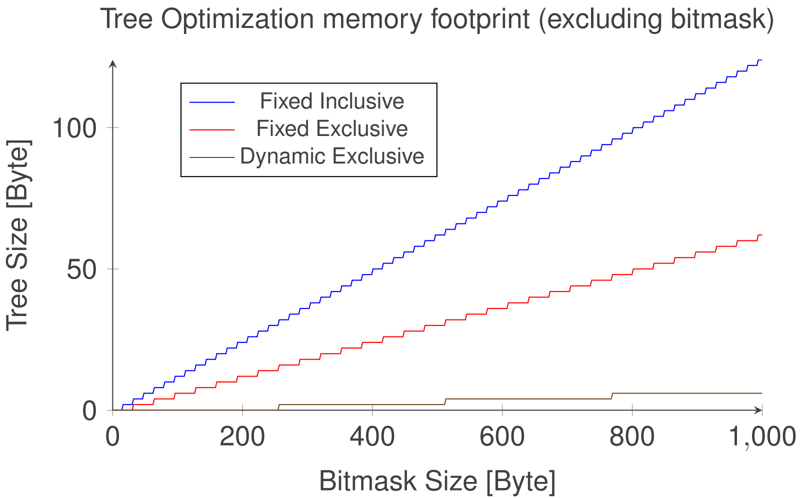

Figure 18: Memory footprint of the tree layers (excluding bitmask) for small bitmask sizes and all optimizations.

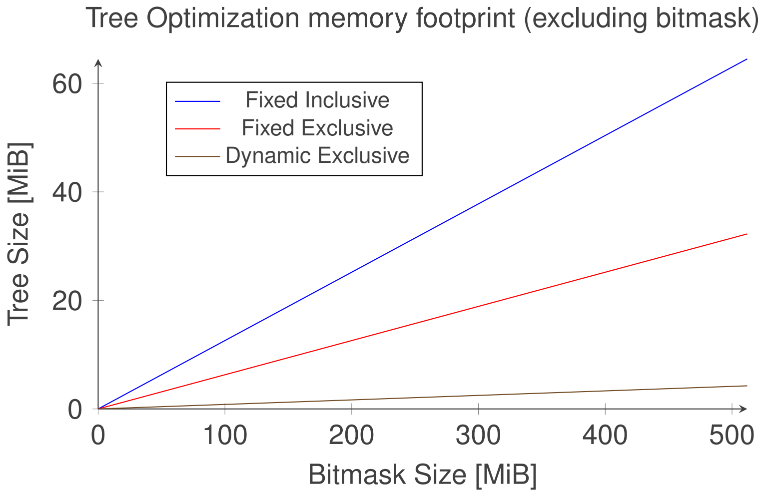

Figure 19: Memory footprint of the tree layers (excluding bitmask) for larger bitmask sizes and all optimizations.

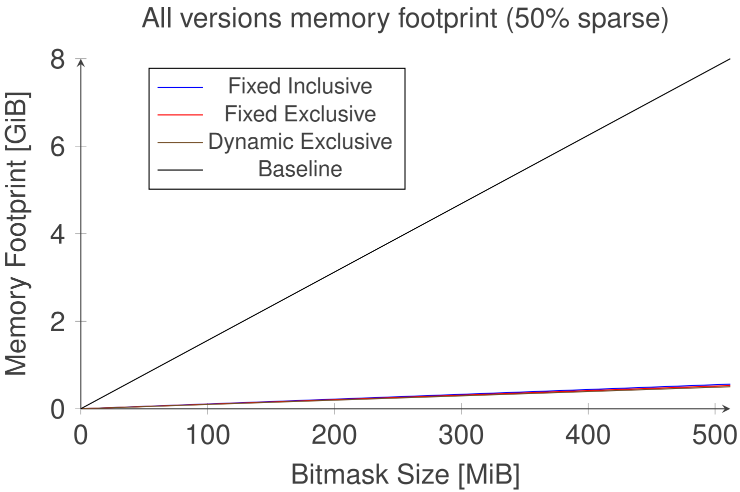

Figure 20: Full memory footprint of all implementations for larger bitmask sizes.

Nsight Compute Profiling of Apply

Metric	Fixed Inclusive	Fixed Exclusive	Dynamic Exclusive
SOL Compute	83.39%	81.77%	**83.58%**
SOL Memory	19.90%	20.87%	**24.60%**
L1/TEX Pattern	3.7 / 32	3.9 / 32	**30.6** / **32**
Avg active threads	19.4 / 32	**19.7** / **32**	17.8 / 32
Occupancy	**94.16%**	93.76%	89.56%
Instructions [M]	911.83	860.94	**731.68**

Table 1: Direct comparison of Nsight Compute metrics between our three optimizations.
Each implementation was run on the same bitmask with 2^{20} elements and a sparsity of 50%.

Discussion

- Baseline is significantly faster by a factor of 2-3x for mostly dense bitmasks and stays faster for lower densities as well
- Tree optimizations get faster for lower densities, especially starting at ~4%
- Dynamic Exclusive is fastest tree optimization for high densities by ~40%, but slower than fixed trees for low densities
- Dynamic Exclusive has lowest memory footprint of all versions
- Fixed Exclusive is in all cases faster than Fixed Inclusive, while having lower memory footprint
- Baseline has highest memory footprint for dense bitmasks, but lowest memory footprint for very sparse bitmasks

Discussion

- Thread block sizes of 256 and 512 have the highest setup bandwidth for dynamic exclusive implementation; no clear winners for fixed inclusive/exclusive
- Thread block sizes of 64 and 128 have the overall highest apply bandwidth
- Small layer sizes have negative effects on setup bandwidth for fixed inclusive/exclusive
- Large layer 1 size has negative effect on apply bandwidth

Questions and Discussion

References

(1) SIMD / GPU Friendly Branchless Binary Search. Accessed 15.01.2025. `https://blog.demofox.org/2017/06/20/simd-gpu-friendly-branchless-binary-search/`

(2) CUDA C++ Programming Guide, v12.8. Accessed 16.02.2025. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

(3) Giles, M. Lecture 4: warp shuffles, and reduction / scan operations. Accessed 22.02.2025. `https://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec4.pdf`

(4) GeForce RTX 4080 Family. Accessed 26.02.2025. `https://www.nvidia.com/de-de/geforce/graphics-cards/40-series/rtx-4080-family/`

(5) Nvidia GeForce RTX 4080 Review: More Efficient, Still Expensive. Accessed 26.02.2025. `https://www.tomshardware.com/reviews/nvidia-geforce-rtx-4080-review`

References

(6) cub::BlockScan. Accessed 24.02.2025. https://nvidia.github.io/cccl/cub/api/classcub_1_1BlockScan.html/

Appendix / Optimized Divergence Detection

- Fetch tree element of left thread with __shfl_up_sync()
- Each thread determines the range it covers
- Subsequent threads map subsequent packed indices
- Each thread can determine if others are covered by own range
- Need for multicast of own lane to all threads covered

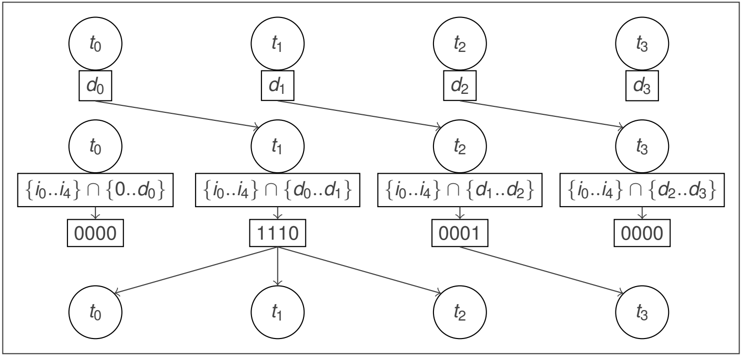

Figure 21: Constant time divergence detection after collaborative tree descend.
Only theoretical, since CUDA doesn't support push based multicast.

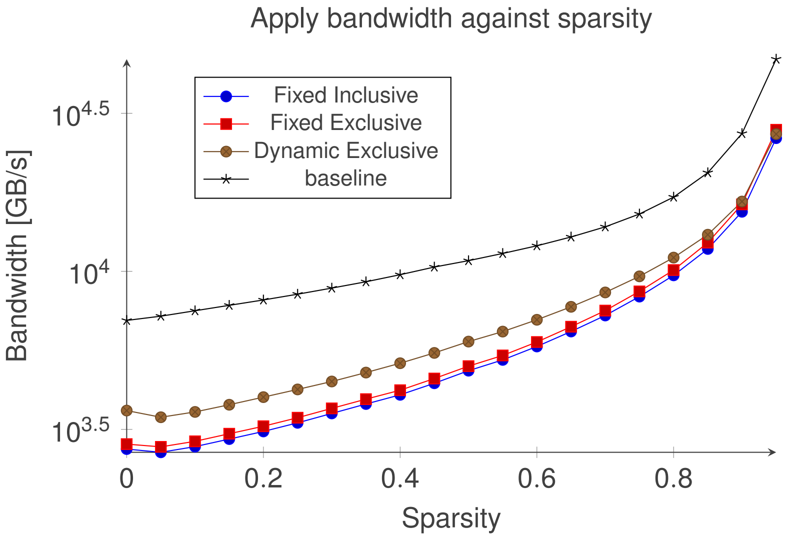

Figure 22: Apply bandwidth for different bitmask sparsities up to 95% sparsity.

Appendix / Baseline Benchmark Improvement

- Thrust copy_if requires temporary global memory
- Malloc and free count towards our benchmarks
- Improvement: Custom cached allocator, malloc once in warmup

Figure 23: Nsight systems profile of setup benchmark with 3 iterations for 2^{20} bitmask elements. Malloc in dark red, memcpy in light red. First iteration is excluded from measurement.