

CS 623: Application Case Studies with CUDA

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2026-27-I



Sparse Matrix-Vector Multiplication

Sparse Linear Algebra

Operations Involving Sparse Matrices

- Many real-world applications involve **sparse** matrix computation
 - ▶ Small fraction of non-zeros (NZ) relative to the matrix dimensions $M \times N$ (i.e., $NZ \ll M \times N$)
- Using a 2D array to store a sparse matrix with large values of M and N is wasteful
- Several sparse matrix representations have been proposed (e.g., CSR, ELLPACK, and CSR5)
 - + Aims to reduce memory wastage
 - + Aims to optimize the data layout according to the requirements of the compute device

COO and CSR Encodings

COO stores each non-zero element along with its row and column indices

Sparse Matrix A

$$A = \begin{bmatrix} 1 & 0 & 0 & 7 \\ 0 & 2 & 0 & 0 \\ 3 & 0 & 4 & 0 \end{bmatrix}$$

Values [1, 7, 2, 3, 4]

Row Indices [0, 0, 1, 2, 2]

Col Indices [0, 3, 1, 0, 2]

CSR compresses the row array to avoid repetitions

Sparse Matrix A

$$A = \begin{bmatrix} 1 & 0 & 0 & 7 \\ 0 & 2 & 0 & 0 \\ 3 & 0 & 4 & 0 \end{bmatrix}$$

Values [1, 7, 2, 3, 4]

Col Indices [0, 3, 1, 0, 2]

Row Pointers [0, 2, 3, 5]

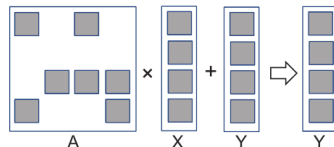
Sparse Matrix-Vector Multiply on CPUs

COO

```
for (int i = 0; i < num_elem; row++)  
    y[row_index[i]] += data[i] * x[col_index[i]];
```

CSR

```
for (int row = 0; row < num_rows; row++) {  
    float dot = 0;  
    int row_start = row_ptr[row];  
    int row_end = row_ptr[row+1];  
    // Loop can be parallelized  
    for (int elem = row_start; elem < row_end; elem++) {  
        dot += data[elem] * x[col_index[elem]];  
    }  
    y[row] += dot;  
}
```



A is a sparse matrix, and x and y are dense vectors

SpMV has irregular and indirect memory accesses and is memory-bound

Implementing SpMV GPU Kernel

```
__global__ void SpMV_CSR(int num_rows, float *data, int *col_index, int *row_ptr, float *
    x, float *y) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < num_rows) { // Number of rows may not be a multiple of the thread block size
        float dot = 0;
        int row_start = row_ptr[row];
        int row_end = row_ptr[row+1];
        for (int elem = row_start; elem < row_end; elem++) {
            dot += data[elem] * x[col_index[elem]];
        }
        y[row] += dot;
    }
}
```

Implementing SpMV GPU Kernel

```
__global__ void SpMV_CSR(int num_rows, float *data, int *col_index, int *row_ptr, float *  
    x, float *y) {  
    int row = blockIdx.x * blockDim.x + threadIdx.x;  
    if (row < num_rows) { // Number of rows may not be a multiple of the thread block size  
        float dot = 0;  
        int row_start = row_ptr[row];  
        in  
        fo  
    }  
    y[  
}  
}
```

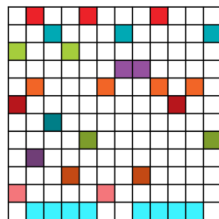
- Uncoalesced global memory accesses for data
- Possibility of significant control divergence or load imbalance

Mapping Threads to Work

CSR Scalar one thread per matrix row, i.e.,
one sparse matrix dot product

CSR Vector multiple cooperative threads
per matrix row

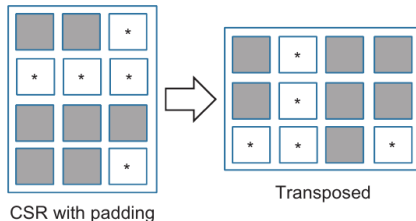
- Requires either shared memory or warp shuffle operations to reduce the partial values computed across threads



Neither CSR scalar nor CSR vector offer ideal performance

- CSR Vector will have no parallelism and reduce occupancy for short rows

Applying Padding and Transposition



```
__global__ void SpMV_ELL(int num_rows, float *data, int *col_index, int num_elem, float *  
    x, float *y) {  
    int row = blockIdx.x * blockDim.x + threadIdx.x;  
    if (row < num_rows) {  
        float dot = 0;  
        for (int i = 0; i < num_elem; i++) {  
            dot += data[row+i*num_rows] * x[col_index[row+i*num_rows]];  
        }  
        y[row] += dot;  
    }  
}
```

ELLPACK and HYB Encodings

ELLPACK is efficient for matrices with a **bounded** maximum number of non-zero elements per row

HYB combines ELLPACK and COO to reduce padding

Sparse Matrix A

$$A = \begin{bmatrix} 1 & 0 & 0 & 7 \\ 0 & 2 & 0 & 0 \\ 3 & 0 & 4 & 0 \end{bmatrix}$$

Values [1, 7, 2, 0, 3, 4]

Col Indices [0, 3, 1, -1, 0, 2]

Other extensions like JDS and SELLPACK have been proposed

ELL Fixed capacity K per row

COO Stores any excess entries exceeding K per row

Challenges With Sparse Algebra

- Irregular memory accesses
 - ▶ Uncoalesced global memory accesses for sparse matrix elements
 - ▶ Requires data-dependent indirect indexing of vector elements
- Load imbalance
 - ▶ Variable number of nonzeros per row leading to uneven work across threads
- Preprocessing overhead
 - ▶ Cost of format identification and conversion may offset performance benefits
- Tensor cores achieve higher arithmetic throughput, but require regular data layouts and dense operands
 - ▶ Accelerates MMA operations (e.g., $D = A \times B + C$) via the high-level **wmma** API or the low-level **mma** PTX interface, both operating at warp granularity
 - ▶ Wasted work processing zeros in sparse structures

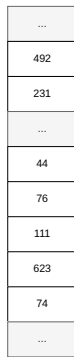
Efficient GPU Hash Table

Random-access application

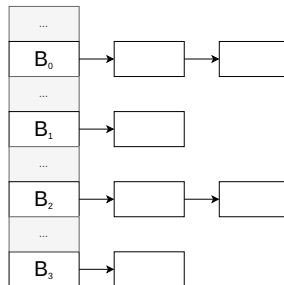
Resolving Collisions in a Hash Table

Closed addressing Each table entry refers to a set of items, called a bucket. Closed addressing is also known as chaining.

Open addressing Each table entry maps to a single item. Open addressing requires a deterministic probing scheme to search for free slots.



Open addressing



Closed Chaining

Probing Techniques

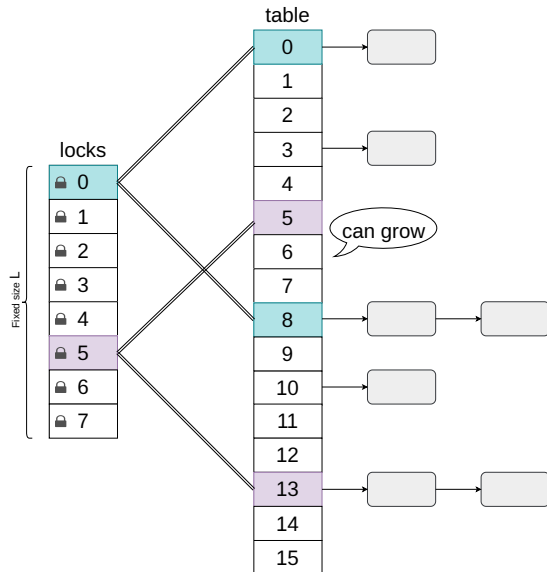
Let c be the number of entries of the hash table, l be the number of probing attempts, and $s(k, l)$ the l -th element in the probe sequence where $s(k, 0) = h(k)$

Linear probing Probing sequence is $s(k, l) = (h(k) + l) \bmod c$. Cache efficient but suffers from clustering.

Quadratic probing Probing sequence is $s(k, l) = (h(k) + l^2) \bmod c$. Incurs more cache misses but avoids primary clustering.

Chaotic probing Probing sequence is $s(k, l) = (h(k) + l \cdot g(k)) \bmod c$ where $g(k)$ is a second hash function. Incurs more cache misses but avoids clustering. Chaotic probing is also known as double hashing.

Designing Fine-Grained Concurrent Hash Table for CPU



- Each lock protects N/L buckets
- Allows more concurrency than coarse-grained locking

Resizing the Lock Array

Remember that a dynamically-sized hash table may require **resizing**

⚠ Why do we not resize the lock array?

- Associating a lock with every hash table bucket would incur large space overhead
- Accesses to the hash table are expected to be more distributed when the table is large implying low contention
- Resizing the lock array is complicated
 - Several locks may be in use, there can be concurrent resizes, need to avoid deadlocks
 - Delays threads that have initiated a concurrent operation
 - The hash index computed before resizing may no longer be valid after resizing

Designing Concurrent Hash Table on GPUs

Requirements

- Need to support high throughput for concurrent accesses to the hash table
- Need to devise nonblocking algorithms that are tuned to the GPU programming model for good performance

Challenges in designing an efficient hash table

- Lock-based synchronization will not scale to thousands of GPU threads
- Accessing linked-list-based data structures imply making random (uncoalesced) memory accesses from threads in a warp
 - ▶ Linked lists are cache-inefficient, each hash key is on a separate node with no locality
- Dynamically allocating linked list nodes for numerous GPU threads is a bottleneck

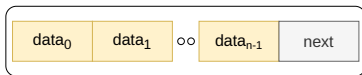
Hash Table with Closed Chaining

i Implementing horizontal chains using linked lists

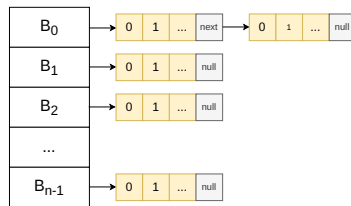
- Manipulating the list requires performing atomic updates on the next pointer



Linked list node



Slab list node



Hash table

SlabHash: Hash Table with Closed Chaining

SlabHash builds a concurrent GPU hash table using slab lists instead of linked lists

- A slab consists of multiple words of data (i.e., unordered set) and a single next pointer, thereby reducing memory overhead
 - ▶ Slabs on NVIDIA GPUs can be of 32 words
- Can search a slab node using a single ballot instruction

Uses a universal hash function of the form $h(k; a, b) = ((ak + b) \bmod p) \bmod B$

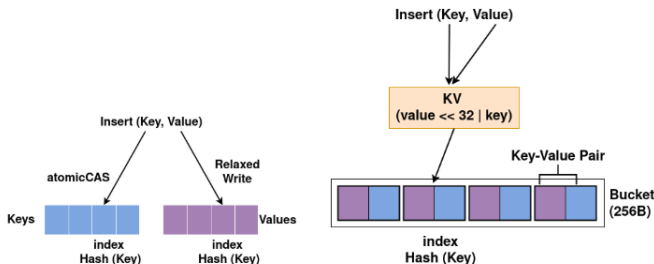
- a, b are random integers such that $1 \leq a < p$ and $0 \leq b < p$,
- p is a random prime number greater than the maximum allowed key,
- B is the number of buckets

Operations on a Slab List

- search** Search from the head of the list, load the next slab if the item cannot be found in the current slab
- insert** Start from the head of the list, use an atomic CAS to insert the new key-value pair into the first empty data slot. An unsuccessful CAS implies someone else occupied the empty slot, so restart looking for an empty slot. Load the next slab node if no empty slots are found. Allocate a new slab node at the end of the list if all slab nodes are full.
- replace** Similar to insert except that the entire slab list needs to be searched
- delete** Similar to insert except that the entire slab list needs to be searched (depending on whether duplicates are allowed)

Memory Layout of a Bucket

- A SoA layout requires an atomic CAS on the key slot and a conditional relaxed write to update the value
- With a packaged AoS, a single 64-bit CAS will suffice if both the key and value fits in 64-bits



More about SlabHash

Threads are assigned independent tasks (i.e., keys), but work is done in parallel per-warp, called warp-cooperative work sharing (WCWS)

- One-to-one mapping maps each thread to a single key, threads in a warp process their 32 keys individually
- + Advantage of WCWS is that it reduces stalls when compared to per-thread processing

- No locality across slab nodes
- Dynamic memory allocation at a massive scale is inefficient, requires a custom slab node allocator
- Lock-free insertion and deletion of nodes in linked lists is complicated because of ABA problem

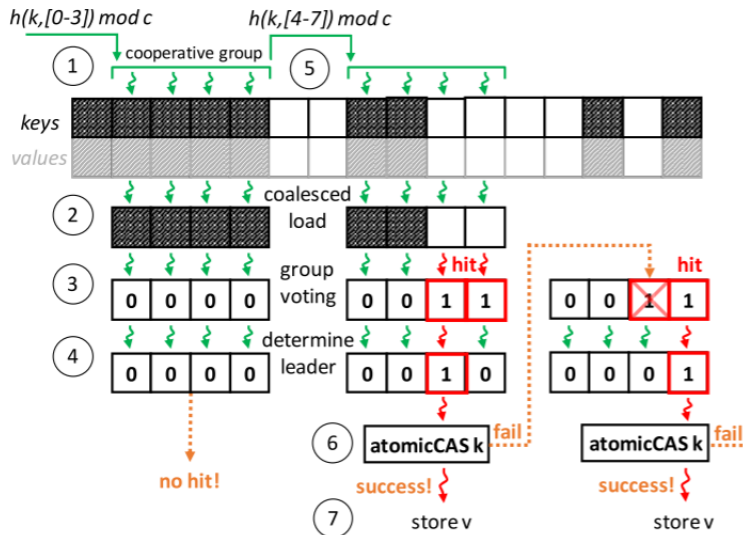
Hash Table with Open Addressing

Open addressing with parallel probing is more amenable to the SIMT execution model on GPUs

Parallel probing scheme

- Mapping each thread to a key will lead to a different probing sequence for threads in a warp, leading to non-coalesced global memory accesses
- Can use an entire warp of 32 threads per input key k , such that each thread with lane ID t probes a different hash table position $h(k, t) \bmod c$, but linear probing suffers from clustering
- WarpCore uses double hashing with an inner intra-warp linear probing, double hashing determines the starting offset for linear probing

Example of Insertion Operation with WarpCore



References



D. Kirk and W. M. Hwu. Programming Massively Parallel Processors. Chapters 1–5, 7, 10, 13, 20, 3rd edition, Morgan Kaufmann.



NVIDIA Corporation. CUDA C++ Programming Guide, Release 12.9.2.



NVIDIA Corporation. CUDA C++ Best Practices Guide, Release 12.9.