GPU-accelerated k-mer analysis in MetaHipMer

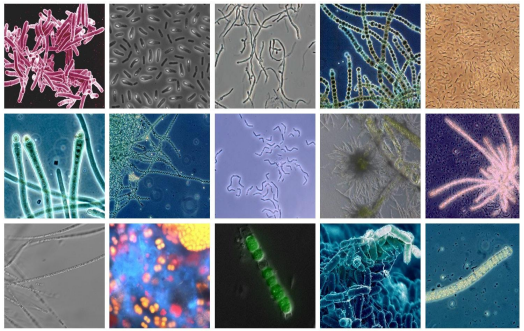
Prashant Pandey
University of Utah
September 2022

MetaHipMer2 Metagenome Assembler

- Part of ECP Exabiome project
- Original MetaHipMer (v1) was released in 2017 (written in UPC/MPI)
- Released version 2 of MHM on Sept 30th 2020
  - Entirely rewritten in UPC++
  - Runs up to 10x faster than MHM v1 and uses 2x less memory
- **Increasing use of GPUs**

Science Analysis with Metagenomics

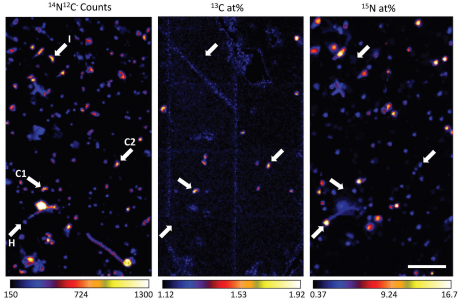
What happens to microbes after a wildfire?



What are the microbial dynamics of soil carbon cycling?

How do microbes affect disease and growth of switchgrass for biofuels

What are the seasonal fluctuations in a wetland mangrove?

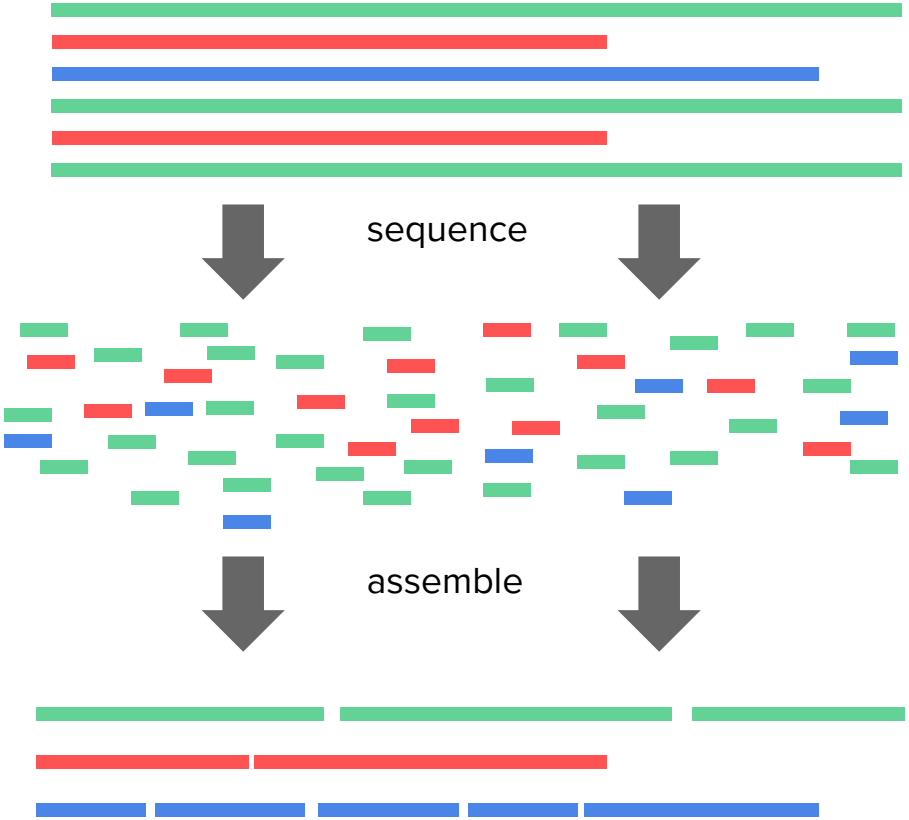


Combine genomics with isotope tracing methods for improved functional understanding

JGI JOINT GENOME INSTITUTE

JGI-NERSC-KBase FICUS (Facilities Integrating Collaborations for User Science) call called out MetaHipMer

Metagenome Assembly



microbial genomes (1000s)

- varying abundance (frequency)
- sequence depth (how many times sequenced, e.g. 50x)

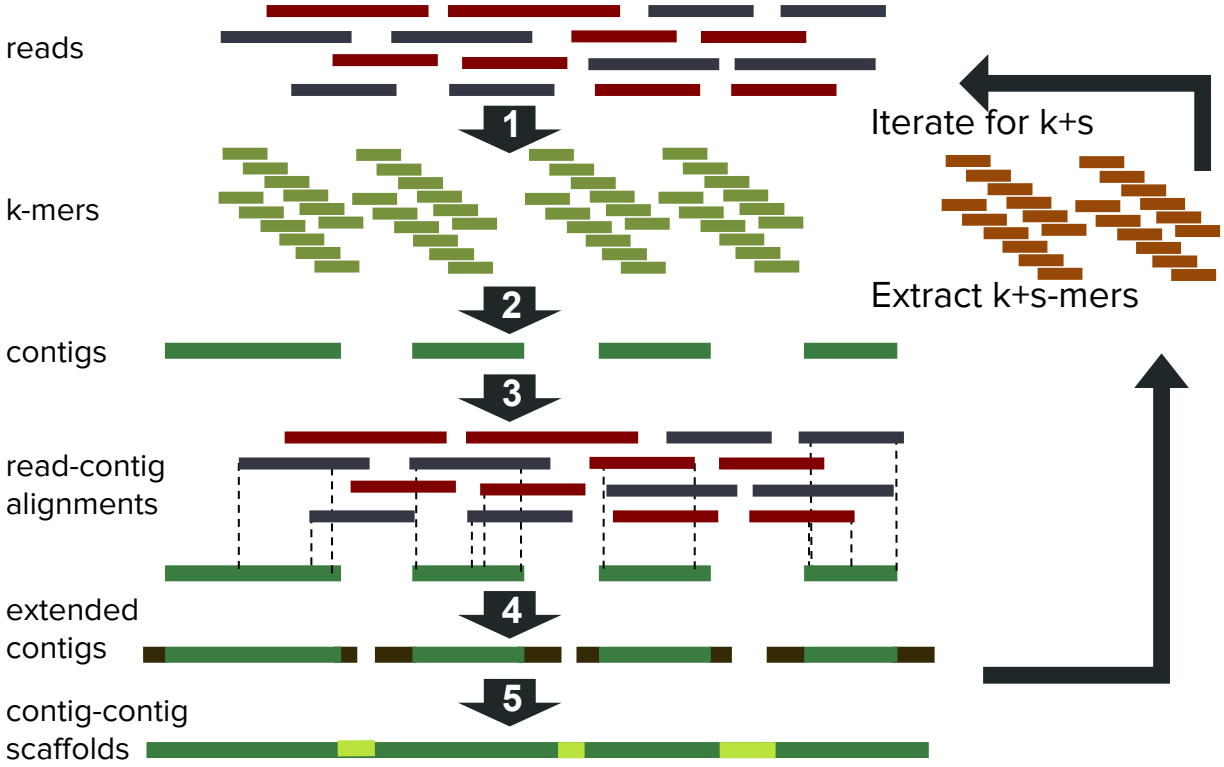
reads

- typical length for short reads 150-250
- error prone, e.g 0.24% per base
- number of reads for a genome dependant on sequence depth and abundance

contigs

- contiguous sequences
- the longer the better
- the fewer errors the better

The MHM2 pipeline



1) K-mer Analysis
Build k-mer graph

2) Contig Generation
Walk k-mer graph

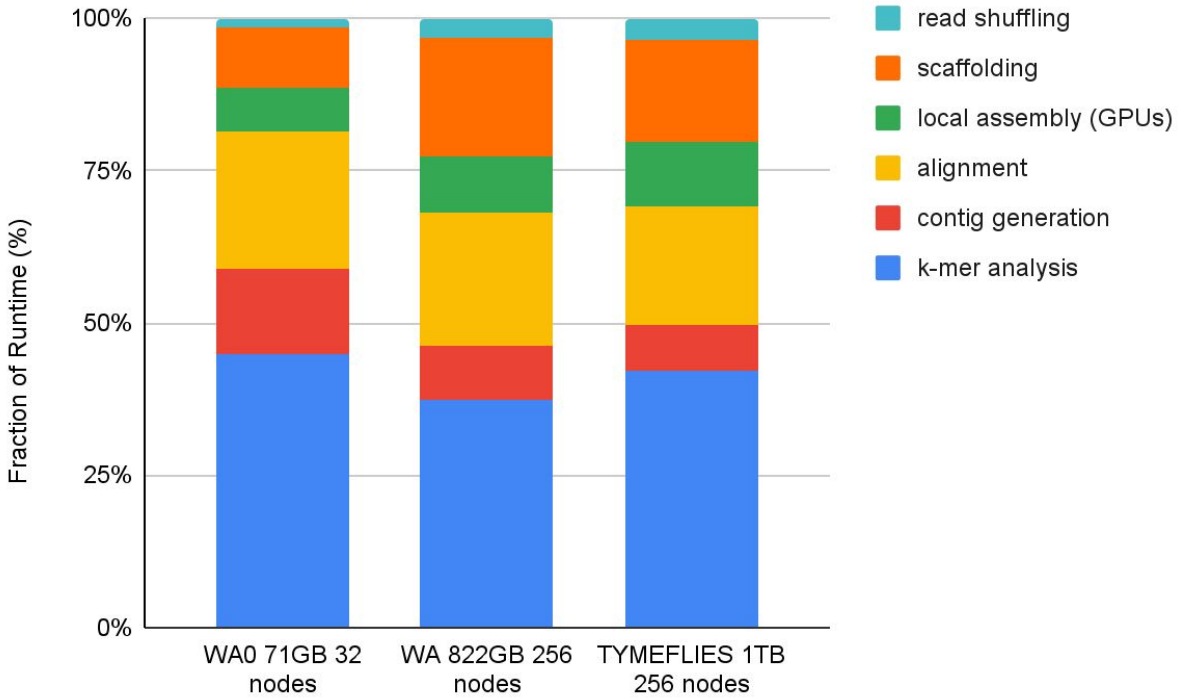
3) Alignment
Align reads to contigs

4) Local Assembly
Extend contig ends

5) Scaffolding
Walk contig graph (iterate)

MHM2 Performance on Summit

- Using GPUs for local assembly and CPUs for kmer analysis
- kmer analysis is most of runtime (up to 45%)



Can we speed up k-mer analysis phase using GPUs?

k-mer counting

Preprocessing

k-mer	Count
ACTGATG	10
GGTGCAT	20
AACTGGA	2
CCGTGAC	1000
GGTGTGC	4000000
CGTGCAC	11
GTGTCAC	9090992

***k*-mer counting**
> 30 papers

Downstream applications

1. Genome/metagenome assembly
2. Error correction
3. Metagenomic classification
4. Sequence clustering
5. Sequence search
6. Abundance study

etc.

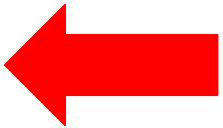
- The ***size*** of the raw sequencing data makes the problem challenging
- *k*-mer counts follow ***highly skewed distributions*** making the problem computationally intensive

Two optimizations for the GPU k-mer analysis phase

- Reducing communication volume across nodes using supermers to **speed up**
- Reducing **memory usage** on GPU nodes using mapping filters

Two optimizations for the GPU k-mer analysis phase

- Reducing communication volume across nodes using supermers to **speed up**
- Reducing **memory usage** on GPU nodes using mapping filters

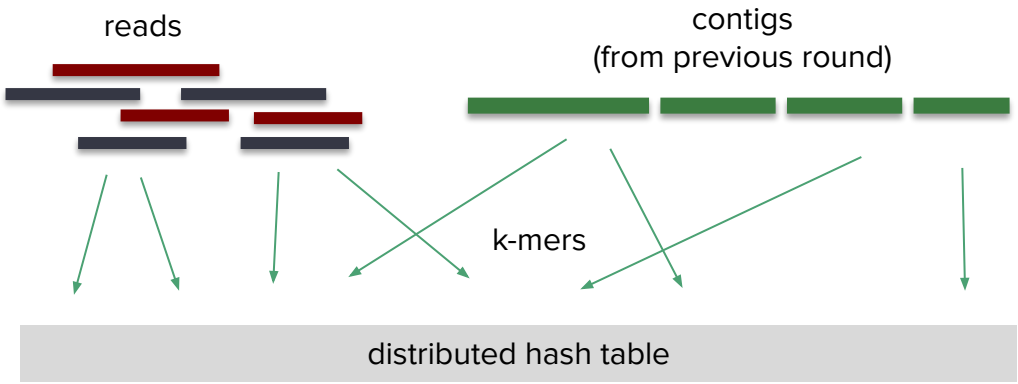


GPU hash table

- Use open addressing
  - fixed size, determined at start by using max available memory to minimize load factor/collisions
  - quadratic probing - allows high load factors
- Consists of two arrays (need to be separately initialized):
  - keys: array of uint64_t, containing k-mer packed into 4 bases per byte
  - **values: array of 8 uint16_t for the counts of each of the 4 extensions on both sides and one uint32_t for the count of the kmer**
- Empty slots indicated by last position of `KEY_EMPTY = 0xff...ff`
  - always use odd values of k, and each nucleotide is packed into 2 bits, so last two bits will never be used
- Locked slots indicated by last position of `KEY_TRANSITION = 0xff...fe`
  - also never overlaps with packed k-mer

k-mer analysis for creating distributed de Bruijn graph

- Our goal is to build a de Bruijn graph as a distributed hash table
- k-mers come from reads and from contigs in previous rounds
- **Don't use singletons (likely errors)**
- **Each vertex is a compressed k-mer with extensions and k-mer count**

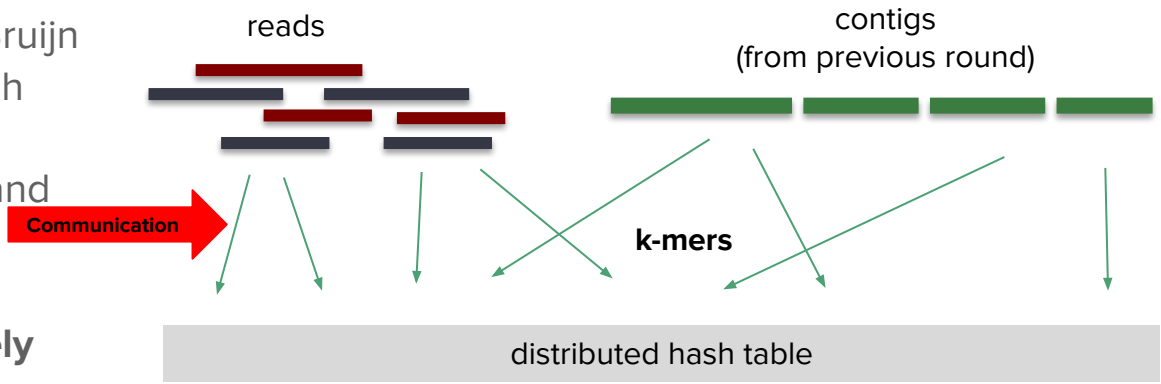


key: compressed k-mer

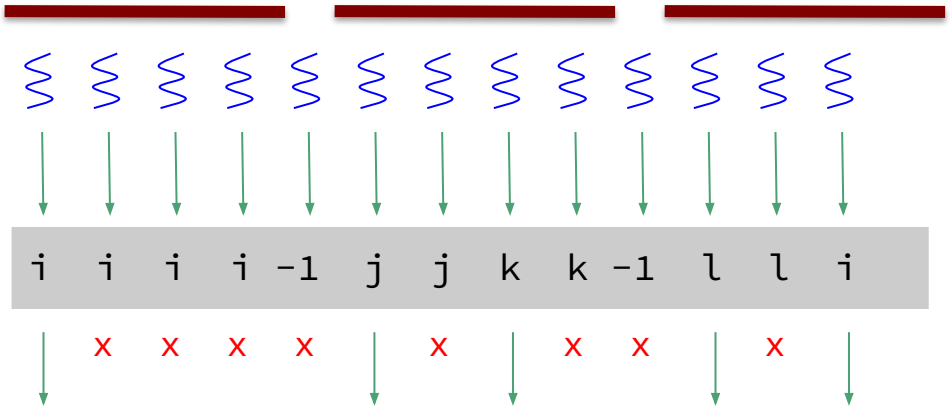
value: count: integer
left extension: A,C,G,T,X,F
right extension: A,C,G,T,X,F

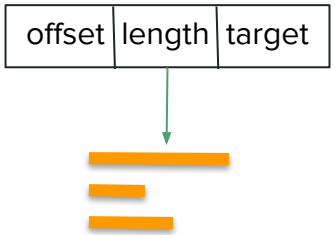
k-mer analysis for creating distributed de Bruijn graph

- Our goal is to build a de Bruijn graph as a distributed hash table
- k-mers come from reads and from contigs in previous rounds
- **Don't use singletons (likely errors)**
- **Each vertex is a compressed k-mer with extensions and k-mer count**



Splitting reads into compressed supermers





Compressed supermer sequences obtained on CPU by applying offsets and lengths to compressed string; then dispatched to targets

concatenate reads into strings separated by underscores _

each thread processes one position and computes target from (revcomp) kmer minimizer

(⬇ CPU spins on `upcxx::progress` ⬇)

output is device array of target for every kmer (-1 for spaces)

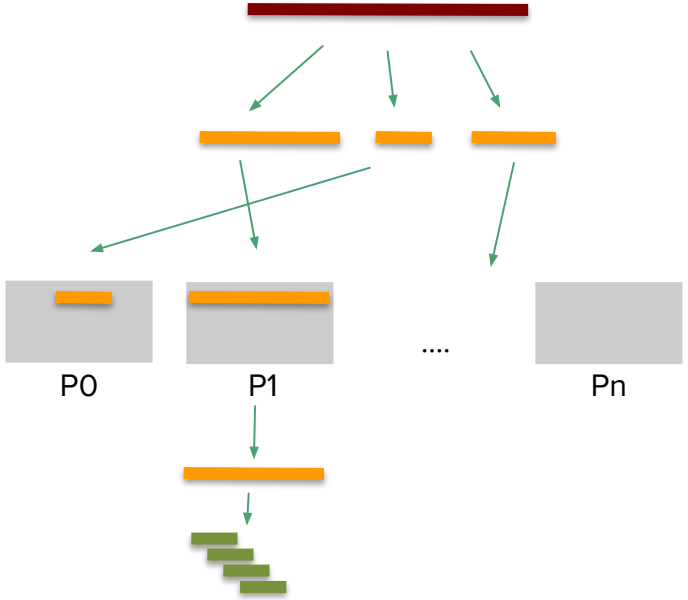
each thread processes one position and builds a supermer if the start position target is different from the left target position

array with offset, length, target for each supermer slot obtained by atomic add

input string compressed separately, one thread per 2 positions (compressing two ACGTNacgt chars into one byte (lower case = low quality)

Inserting k-mers from reads

- **Reads are split into compressed *supermers* on the GPU and then passed to the CPU for network transfer**
- The network transfer is more complicated than this (hierarchical three-tier aggregating store)
- The local hash table is not the same as the final hash table (which stores only a kmer plus left and right extension characters - A,C,G,T,X,F)



read

split into compressed supermers (GPU)

buffer for target process/node

async RPC to target

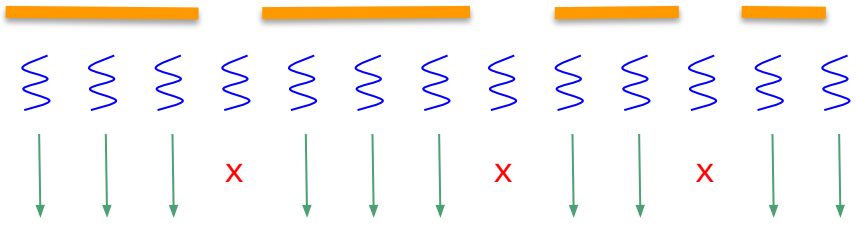
split supermer into kmers (GPU)

insert into local hash table and update counts (GPU)

key: k-mer

value: count: integer
left ext counts: array 4 ints
right ext counts: array 4 ints

Processing supermers at the receiver



left ext (A,C,G,T,0)	packed k-mer (4 bases/byte)	right ext (A,C,G,T,0)

k-mer
count: integer
left ext counts: array 4 ints
right ext counts: array 4 ints

concatenate supermers into string separated by underscores _

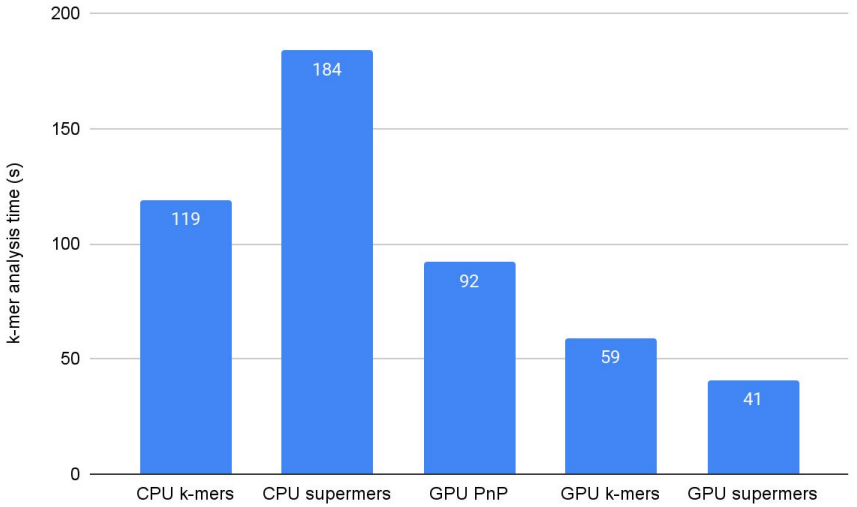
each thread processes one position and computes (revcomp) compressed kmer plus extensions

low quality extensions are set to 0

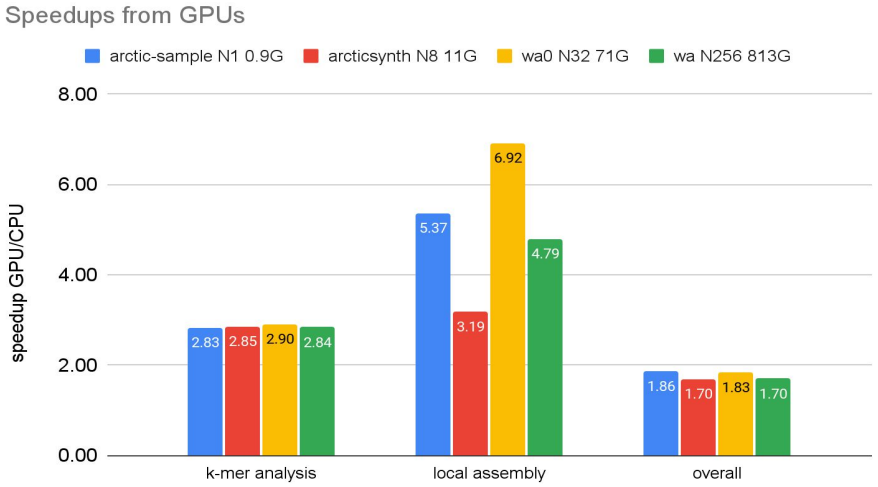
each thread inserts k-mer into hash table on GPU

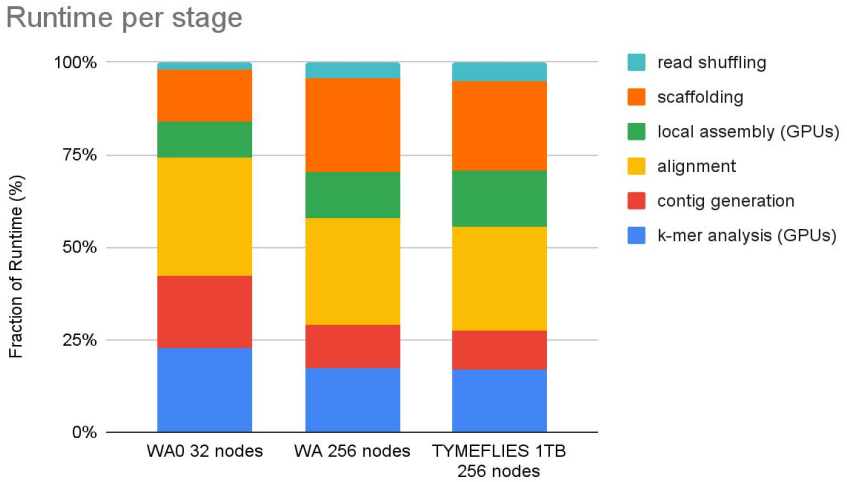
Performance of k-mer analysis

- Dataset is WA0 (71GB) on 32 Summit nodes
- Total speedup from GPUs is 2.9x
- Parse and pack (PnP) gives 29% speedup
- Adding GPU hash tables gives another 56%
- Adding supermers gives another 44%
- Supermers actually *slow down* CPU version by 55% (mostly the cost of extracting supermers in RPCs on receiving processes)



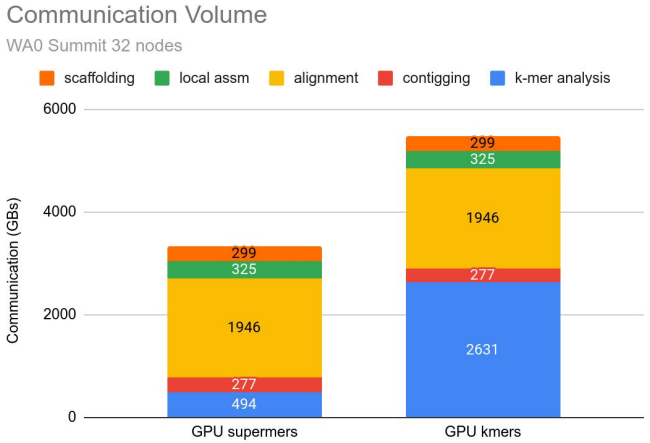
Performance of pipeline

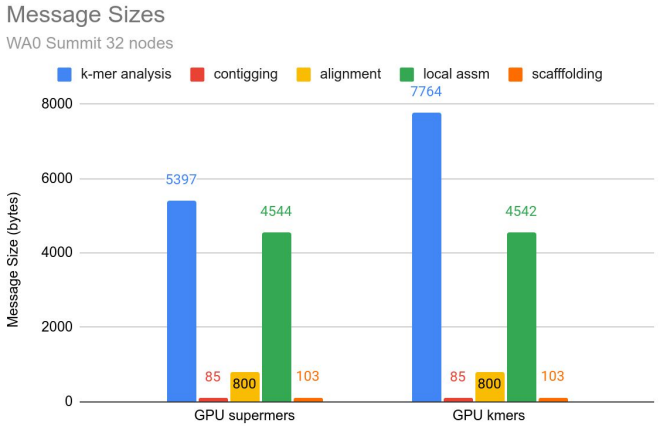




- k-mer analysis is now 2.7x to 2.9x faster
- k-mer analysis is now less than 25% of runtime (down from max 45%)
- overall speedup is 70 to 86% on up to 256 nodes
- GPU computation is a few percent of k-mer analysis time

Communication

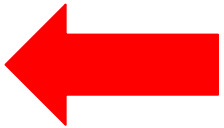




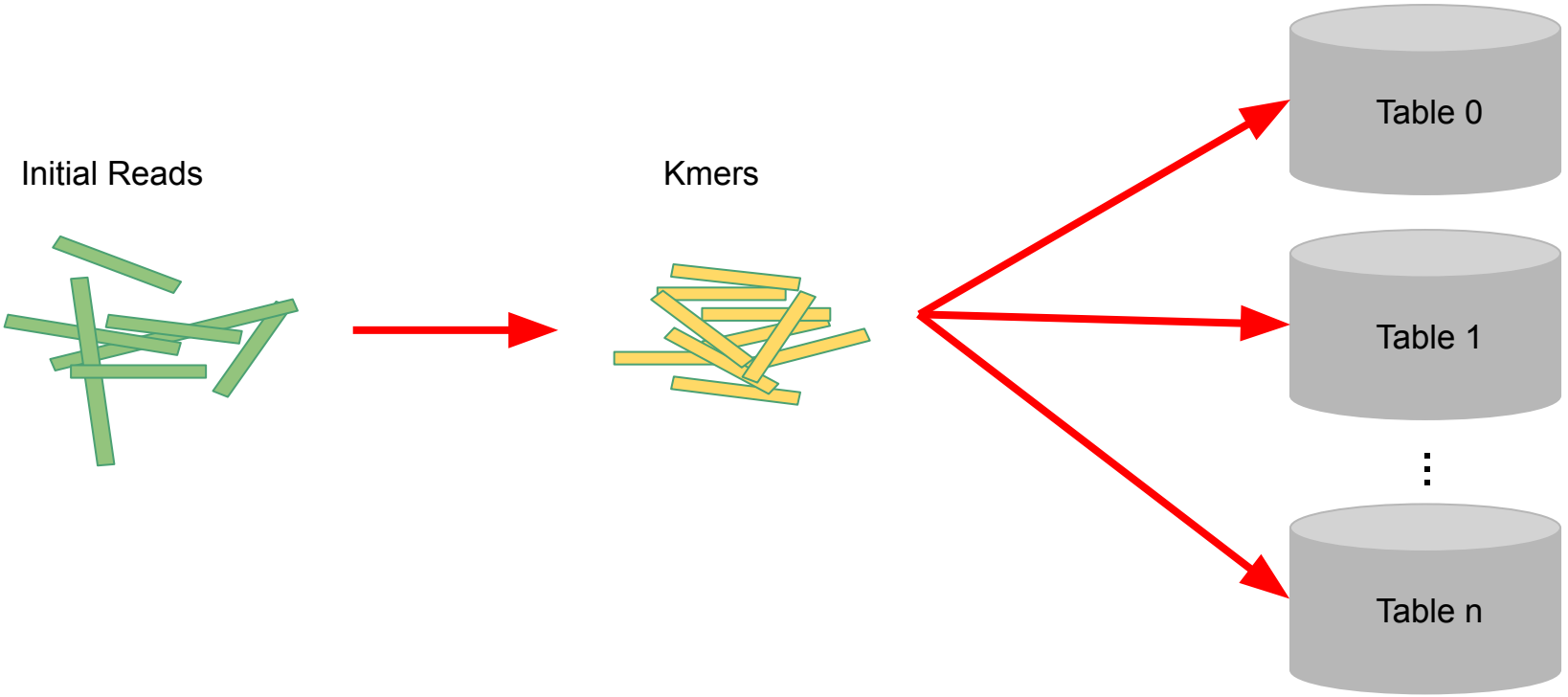
- supermers reduce communication volume by 5.3x and number of messages by 3.7x
- communication reduction results in 44% speedup for k-mer analysis
- most communication is now alignment - reduce in future through earlier read shuffling

Two optimizations for the GPU k-mer analysis phase

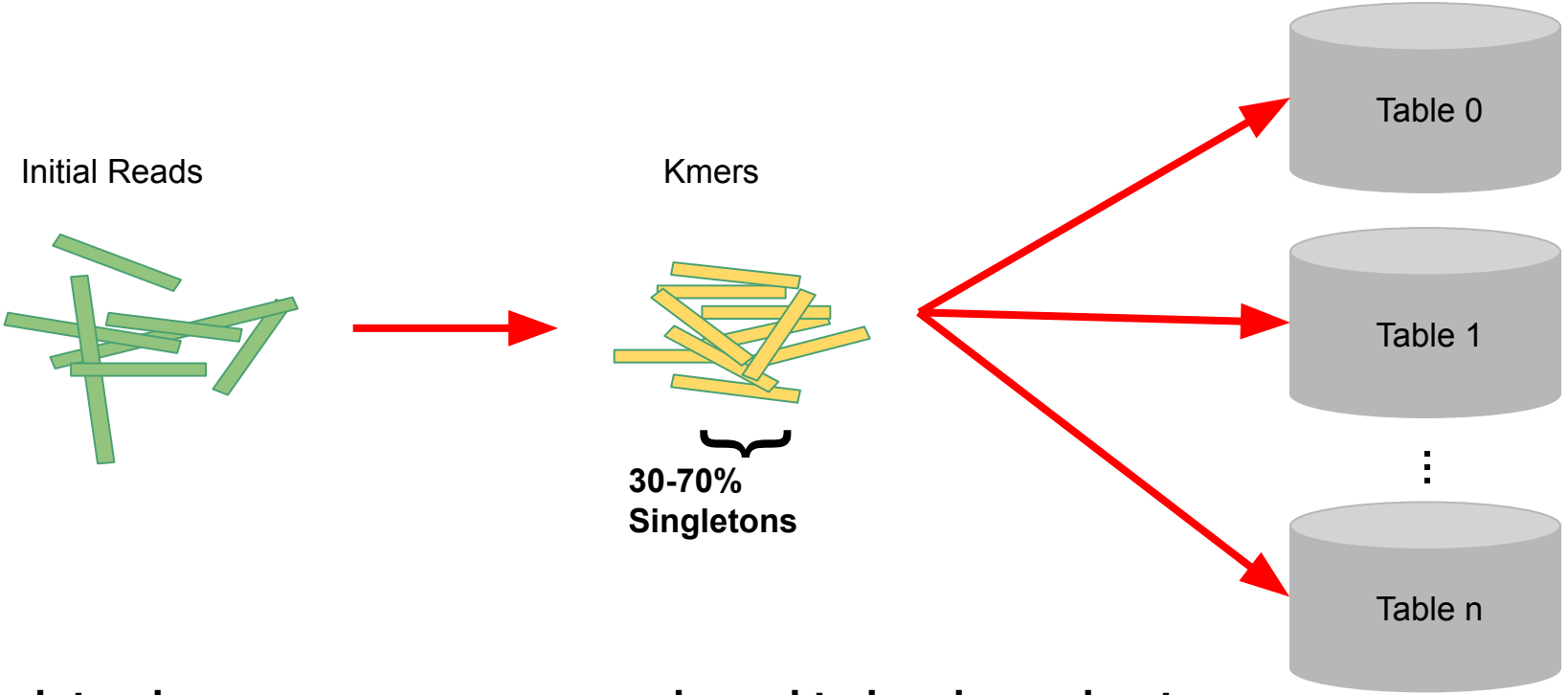
- Reducing communication volume across nodes using supermers to **speed up**
- Reducing **memory usage** on GPU nodes using mapping filters



K-mer analysis phase



K-mer analysis phase

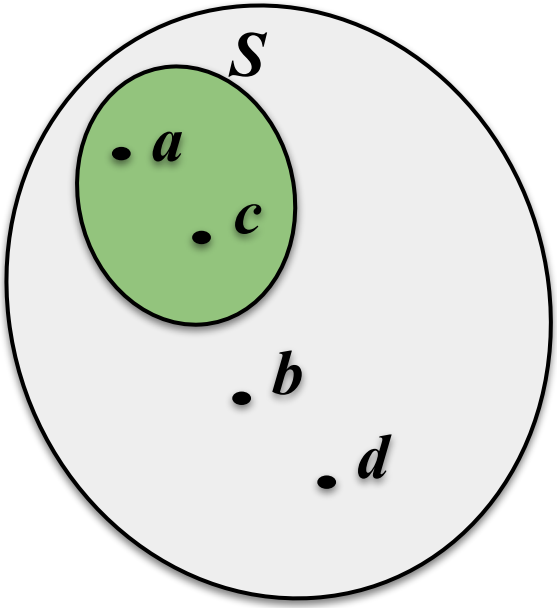


Singleton k-mers are erroneous and need to be cleaned out.

K-mer analysis phase

- Hash tables store all k-mers but discard new k-mers once they are full
  - (load factor = 1)
- GPUs have limited memory (16GBs on V100)
- Lots of wasted space due to singleton k-mers
  - Each k-mer can take up to 18 bytes
- **Non-singletons k-mers end up being dropped resulting in poor assembly quality.**

A filter is an approximate dictionary



membership(a): ✓

membership(b): ✗

membership(c): ✓ 👎 **false positive**

membership(d):

A filter supports ***approximate*** membership queries on S.

A filter guarantees a false-positive rate

if $q \in S$, return ✓ with probability 1 — **true positive**

if $q \notin S$, return

- ✗ with probability $\geq 1 - \varepsilon$ — **true negative**
- ✓ with probability $\leq \varepsilon$ — **false positive**

one-sided errors

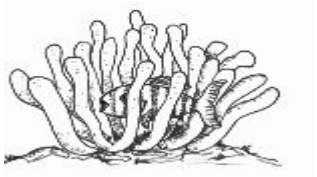
False-positive rate enables filters to be compact

$$\text{space} \geq n \log(1/\epsilon)$$

Small

$$\text{space} = \Omega(n \log |U|)$$

Large



Filter

Hash table

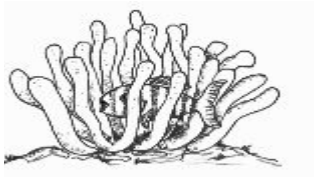
False-positive rate enables filters to be compact

$$\text{space} \geq n \log(1/\epsilon)$$

Small

$$\text{space} = \Omega(n \log |U|)$$

Large



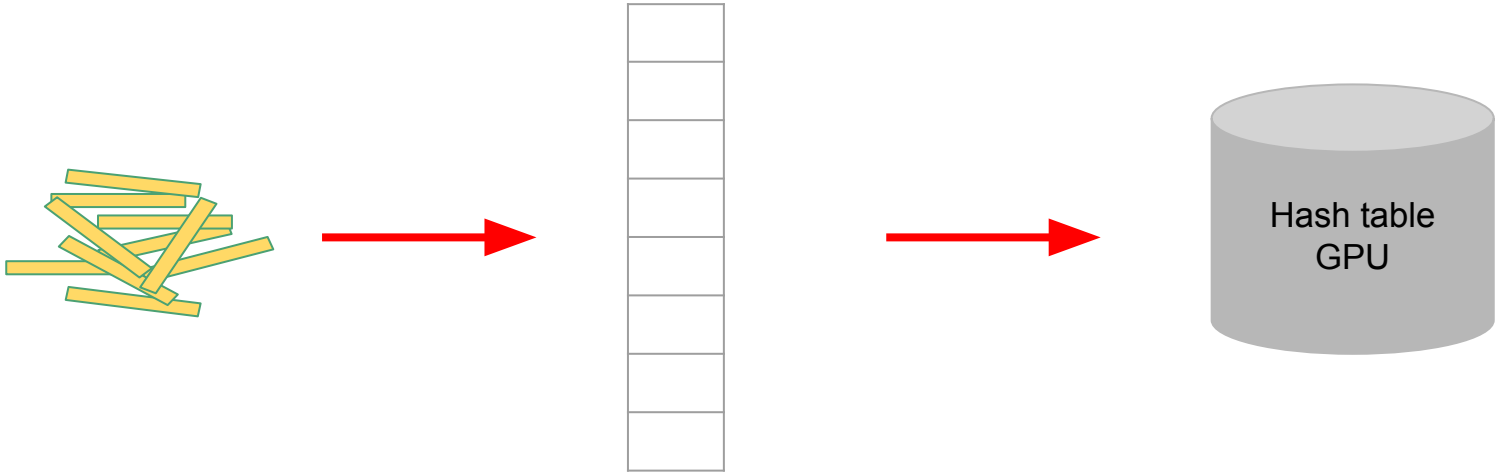
Filter

Hash table

For most practical purposes:

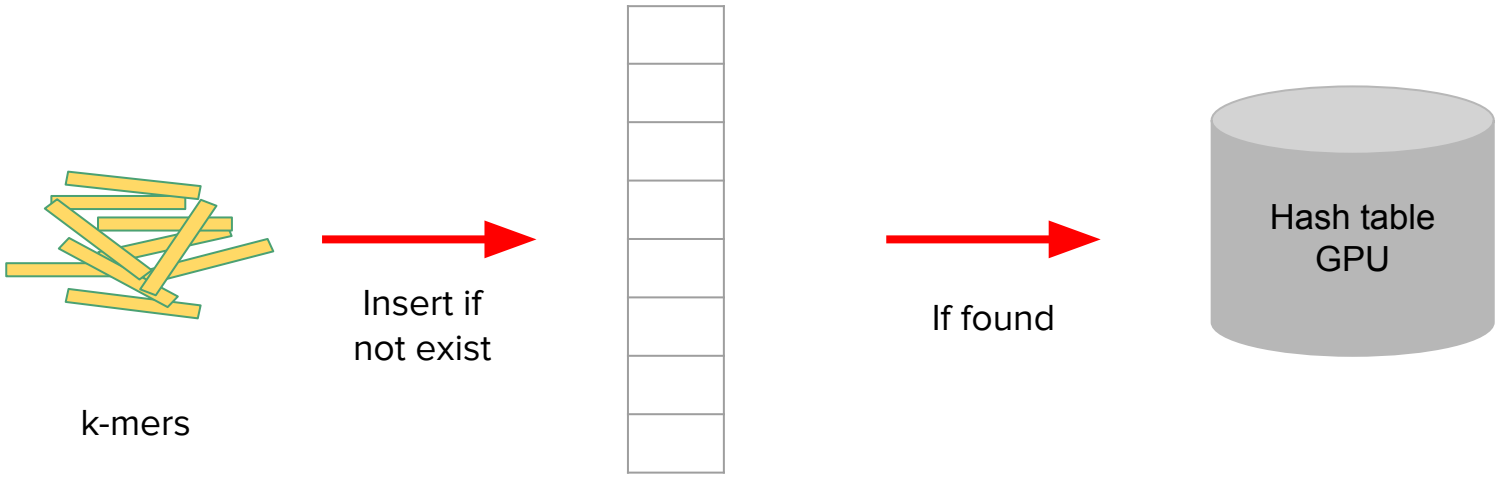
ε = 2%, a filter requires ≈ 8 bits/item

Using filters to remove singletons



Mapping filter on GPU

K-mer analysis with filter

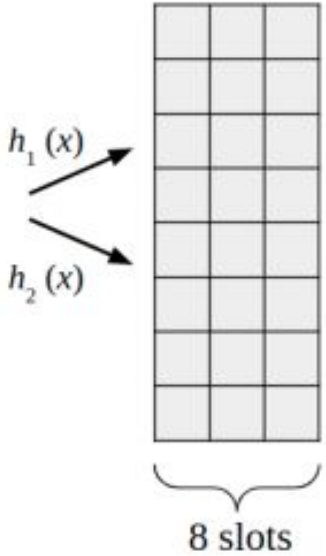


Mapping filter on GPU

We use a quotient filter[Pandey et al. SIGMOD 2017] as a mapping filter.
It can approximately map a key to a small value.
Key is k-mer and value is the extensions.

Two choice filter on the GPU

- The two choice filter is based on **power-of-two-choice** hashing.
- We can perform insert/query/delete operations using only **2 cache line probes** and **1 cache line write.**
- Cooperative groups is used to perform operations inside each block.
- Both point and batch operations supported.
    - Batching operations helps to speed up the throughput



K-mer analysis results using filters

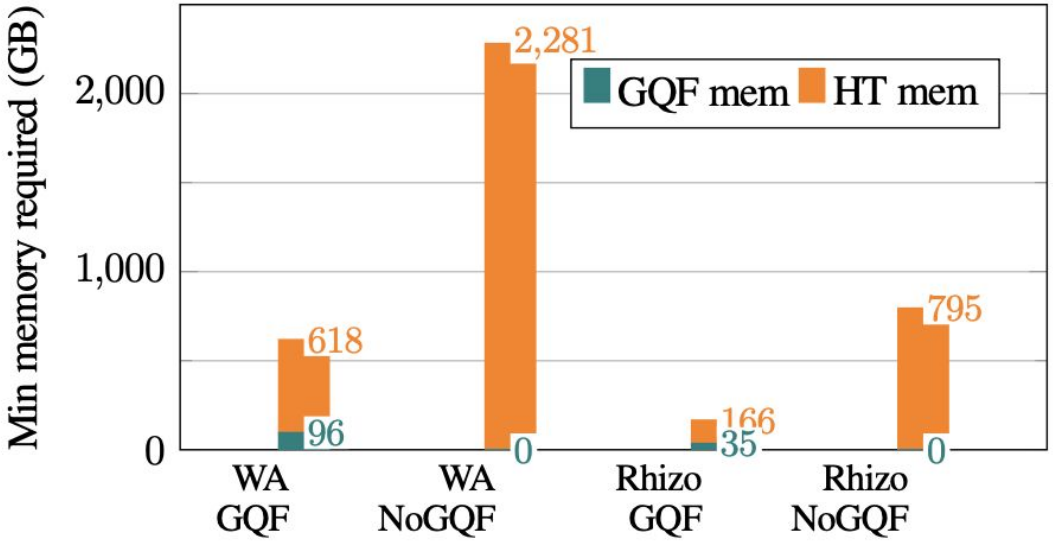


Fig. 6: Memory usage of the GQF and hash table in MHM for WA and Rhizo datasets.

K-mer analysis results using filters

Dataset	Method	Nodes	GQF mem	HT mem mem	GPU insert time	k-mer analysis time	N50 quality
WA	GQF	32	96	522	13.5	236.5	-
	GQF	64	96	522	6.2	125	1336
	No GQF	64	0	2281	4	123	1336
Rhizo	GQF	14	35	131	13.7	151	901
	GQF	24	35	131	8.4	76	910
	No GQF	24	0	795	3.9	72	910

TABLE VI: GQF and Hash table memory usage and running time during MetaHipMer runs across two datasets. Note: the N50 quality score for the WA dataset is not available as the assembly process failed at a later stage after the k-mer analysis phase. **Memory is in GB and time is in seconds.**

Summary

- Accelerating k-mer analysis with GPUs is complicated
- Gives a **2.5x to 2.9x speedup**
- Can **reduce the memory usage by 2x** using advanced filter data structures
- Open questions:
  - Is there a better GPU hash table that can offer consistent high performance?
  - Can we use a better partitioning scheme to perform efficient distributed join during contiging phase?
  - Can we rewrite the whole pipeline using batch parallel model?

Hash table insertions

```
// kmer is array of uint64_t of length N
// all key slots initially set to KEY_EMPTY
int slot = hash(kmer) % capacity; // starting slot
for (i = 0; i < MAX_PROBE; i++) { // maximum number of probes
  do {  // acquire "lock" on key
    key = atomicCAS(keys[slot][N - 1], KEY_EMPTY, KEY_TRANSITION)
  } while (key != KEY_TRANSITION);
  if (key == KEY_EMPTY) {  // lock was acquired on an empty slot, set key
    for (l = 0; l < N - 1; l++) { key = atomicExch(keys[slot][l], kmer[l]);}
    found_key = true; break;
  } else if (key == kmer[N - 1]) { // this could be the same key
    found_key = kmers_equal(kmer, keys[slot]);  // check for same key
    if (found_key) break;
  }
  slot = (start_slot + (i + 1) * (i + 1)) % capacity;  // quadratic probe
}
if (found_slot)
  atomicADD(values[slot][...]...;  // inc k-mer count and extension counts
```

k-mers from contigs

- k-mers from contigs inserted into a separate, smaller *ctg-kmers* GPU hash table
- processed similarly to k-mers from reads, except:
  - count for k-mer is min of all inserts (set with atomicMin, after atomicCAS to check for zero)
  - after all insertions are done, a new GPU kernel purges conflicts (2 or more valid extensions on a side) - marked KEY_EMPTY
- after *ctg-kmers* completed, insert each entry into *read-kmers* hash table:
  - if the k-mer already exists (from a read) but it doesn't have good extensions, add the extension counts from the contig k-mer
  - if the k-mer doesn't exist, add the ctg k-mer

Final stages

- purge invalid entries, one GPU thread per hash table slot:
  - purge (set to KEY_EMPTY) k-mers with a count of 1 or without valid extensions on both sides
- compute final extensions, one GPU thread per hash table slot:
  - for each side, convert array of counts for A,C,G,T into a single base extension or X or F
- each GPU thread inserts (kmer, count, left ext, right ext) into new compact GPU hash table
- copy compact GPU hash table into host memory
- iterate through array on CPU, inserting each (kmer, count, left ext, right ext) element into local CPU hash table for later use in deBruijn graph traverse