IEEE BIG DATA 2023
15–18 December • Sorrento, Italy

GPU Acceleration for Data Processing and Analytics

Qiong Luo

The Hong Kong University of Science and Technology

and

The Hong Kong University of Science and Technology (Guangzhou)

Data Processing and Analytics (DPA)

- Workload characteristics
  - Computation-intensive or data-intensive
  - Relatively simple or complex control flow
  - In-memory or involving multiple passes of disk IO
  - Long running time and/or large memory consumption
- An effective approach to performance improvement
  - Hardware acceleration
- This talk's focus
  - Accelerating a few DPA tasks with the GPU (Graphics Processing Unit)

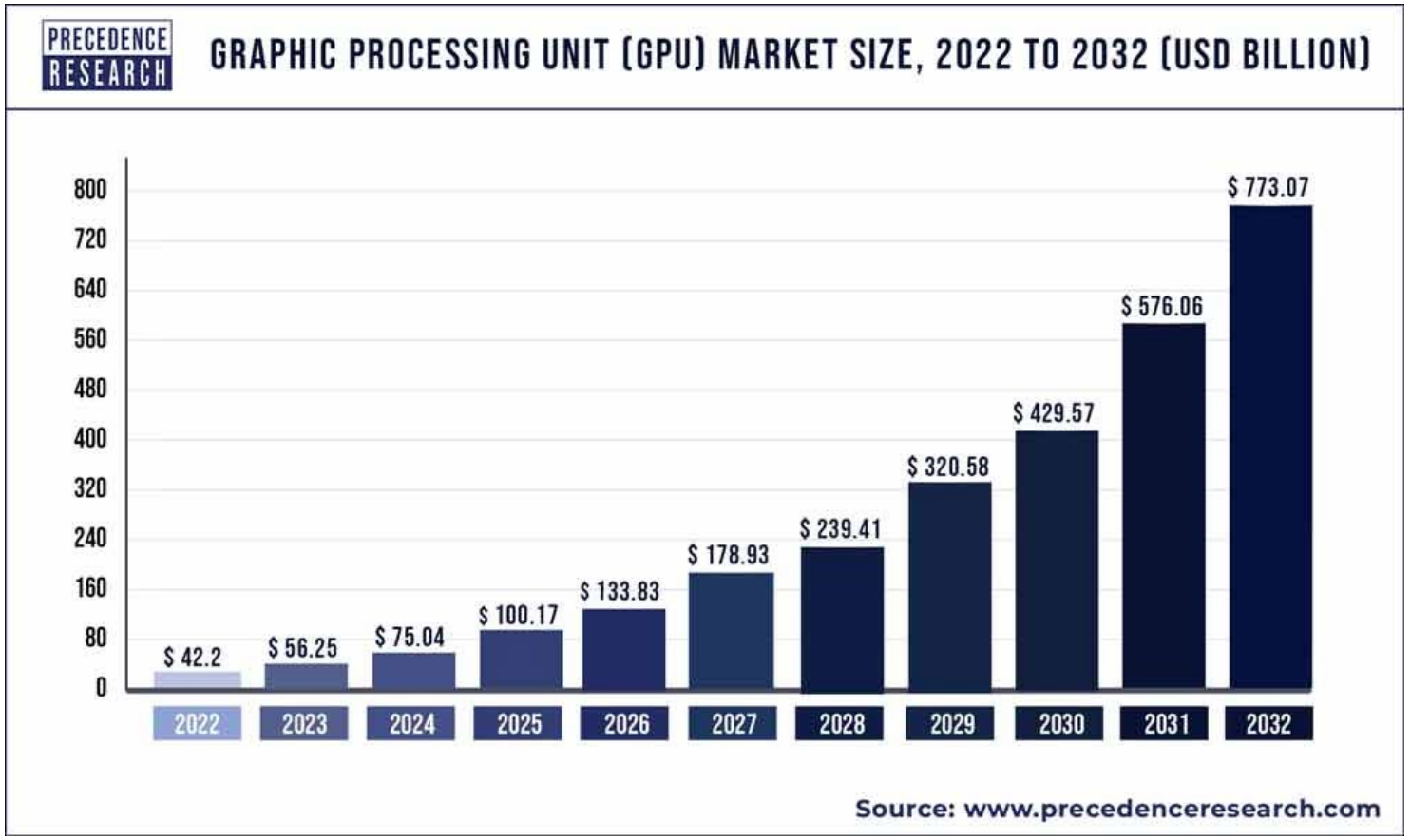
Graphics Processing Units (GPUs) on the Market

- Intel
- IBM
- Samsung
- NVIDIA
- Siemens AG
- AMD
- Qualcomm
- Google
- Dassault Systems
- Sony

- Computer
- Gaming Console
- Smartphone
- Tablet
- Television
- Others

- Integrated
- Dedicated
- Hybrid

- IT & Teleco
- Electronics
- Media & Entertainment
- Defense & Intelligence
- Others



https://www.precedenceresearch.com/graphic-processing-unit-market

NVIDIA GPUs for General-Purpose Computing

NVIDIA GPU Architectures

- Ada Lovelace Architecture (Sep 2022)
- Hopper Architecture (March 2022)
- Ampere Architecture (2020)
- Turing Architecture (2018)
- Volta Architecture (2017)
- Pascal Architecture (2016)
- Maxwell Architecture (2014)
- Kepler Architecture (2012)
- Fermi Architecture (2010)
- Tesla Architecture (2006)
- Curie Architecture 2004)
- Rankine (2003)
- Kelvin (2001)
- Celsius (1999)

Language Solutions

- CUDA Toolkit
- NVIDIA HPC SDK
- OpenACC directives
- PyCUDA
- Altimesh Hybridizer
- OpenCL
- AleaGPU for F#.

Tools & Ecosystem

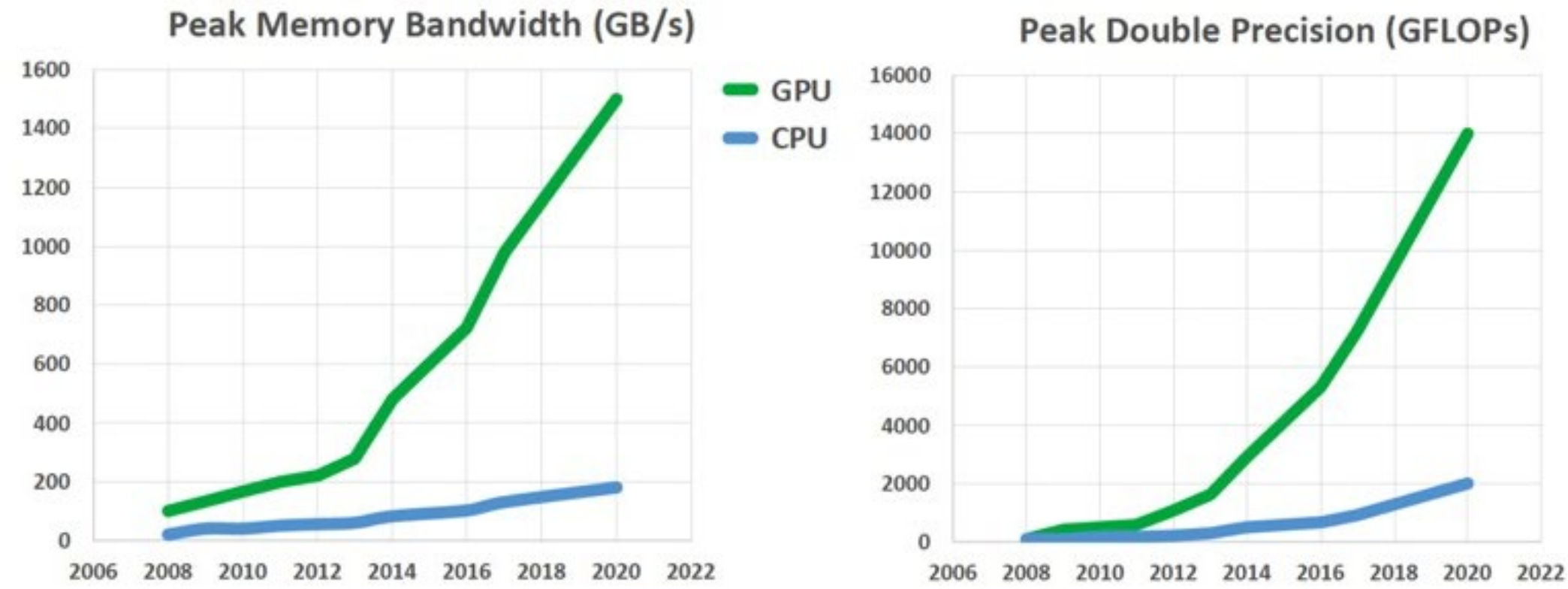
- GPU-Accelerated Libraries
- Performance Analysis Tools
- Debugging Solutions
- Data Center Tools
- Accelerated Web Services
- Cluster Management

GPU-Accelerated Libraries

- Math Libraries
  cuBLAS, cuFFT, cuSparse,...
- Image and Video Libraries
  nvJPEG, codec, optical flow...
- Deep Learning
  cuDNN, DALI, TensorRT,...
- Parallel Algorithms
  Prefix sum, sort, reduce,...
- Communication Libraries
  NVSHMEM for GPU memory
  NCCL for multi-GPU/-node
- Partner Libraries
  OpenCV for computer vision
  Gunrock for graph processing
  CVVILib for medical imaging

https://developer.nvidia.com/tools-ecosystem

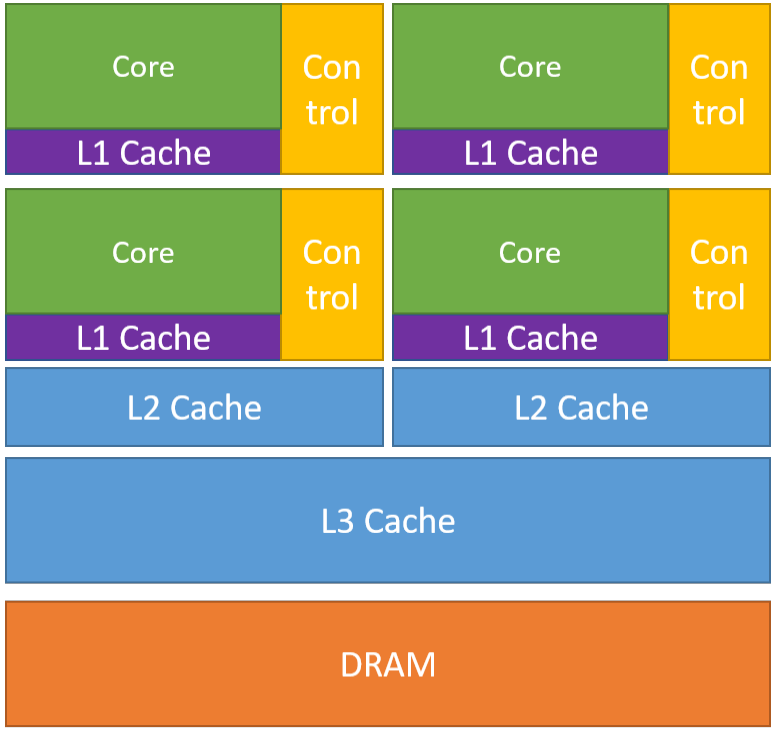
NVDIA GPU Performance Trends



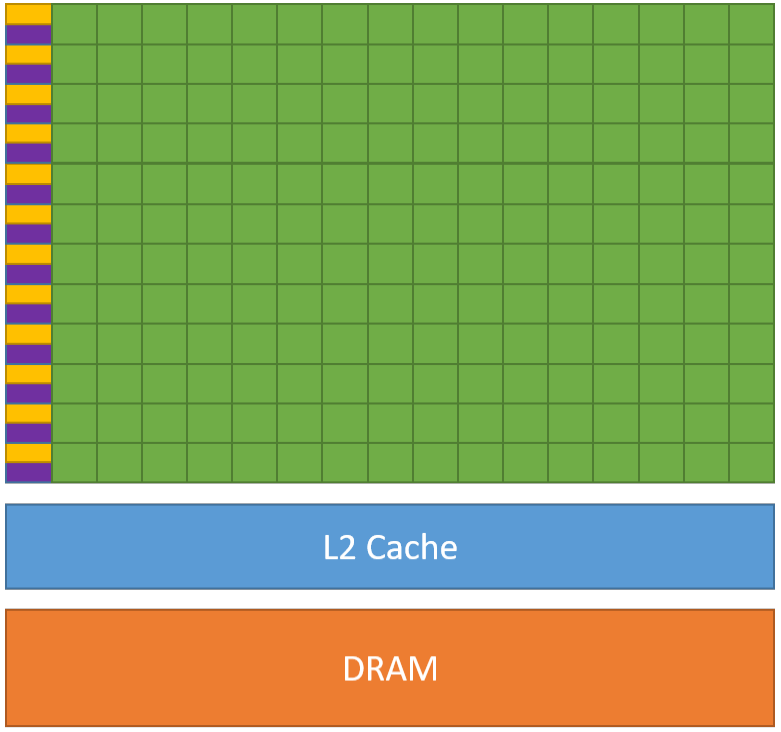
Chip to chip comparison of peak memory bandwidth in GB/s and peak double precision gigaflops for GPUs and CPUs since 2008. Data for Nvidia "Volta" V100 and Intel "Cascade Lake" Xeon SP are used for 2019 and projected into 2020.

https://www.nextplatform.com/2019/07/10/a-decade-of-accelerated-computing-augurs-well-for-gpus/

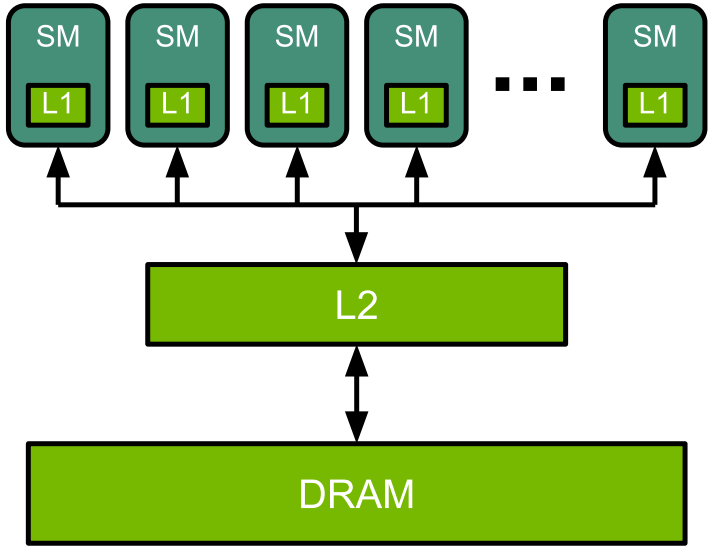
GPU Architecture in Comparison with CPU



CPU



GPU

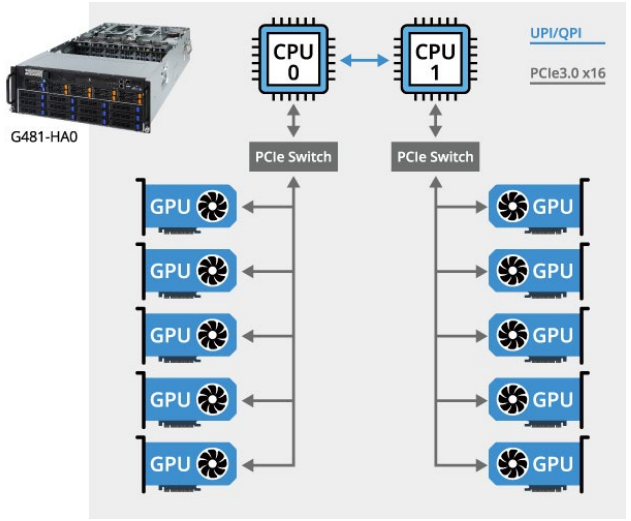


An NVIDIA A100 GPU
- 108 SM (Streaming Multiprocessor),
- a 40 MB L2 cache, and
- up to 2039 GB/s bandwidth from 80 GB of HBM2 memory

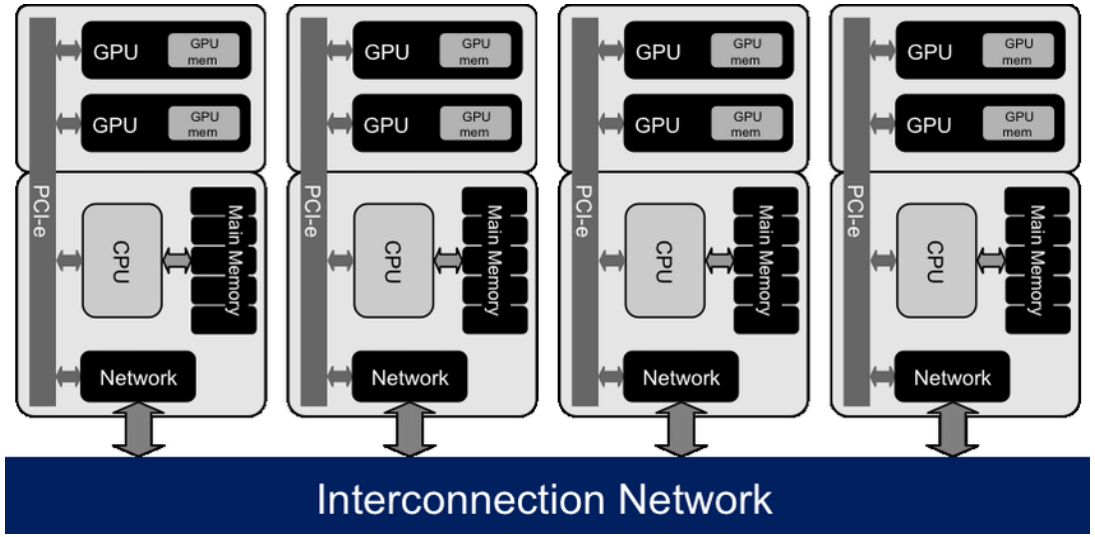
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html

Multi-GPU Computers and GPU Clusters



https://www.gigabyte.com/us/Enterprise/GPU-Server/G481-HA0-rev-200

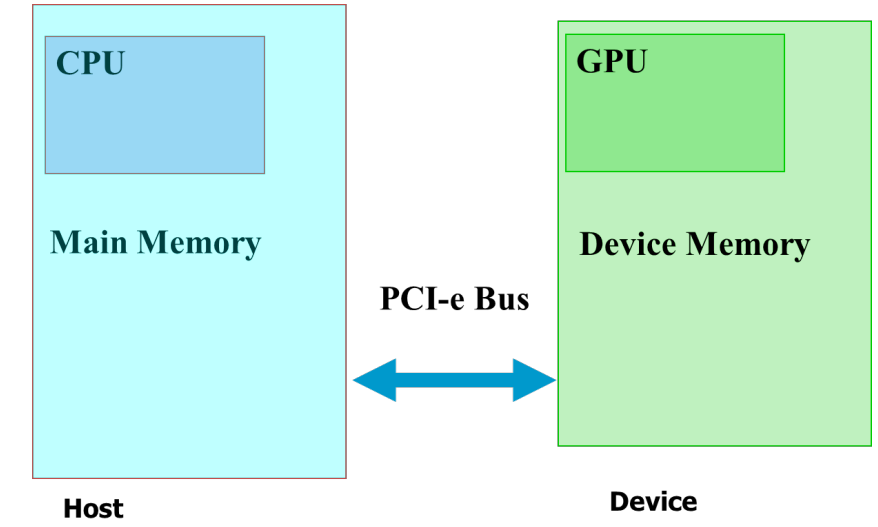


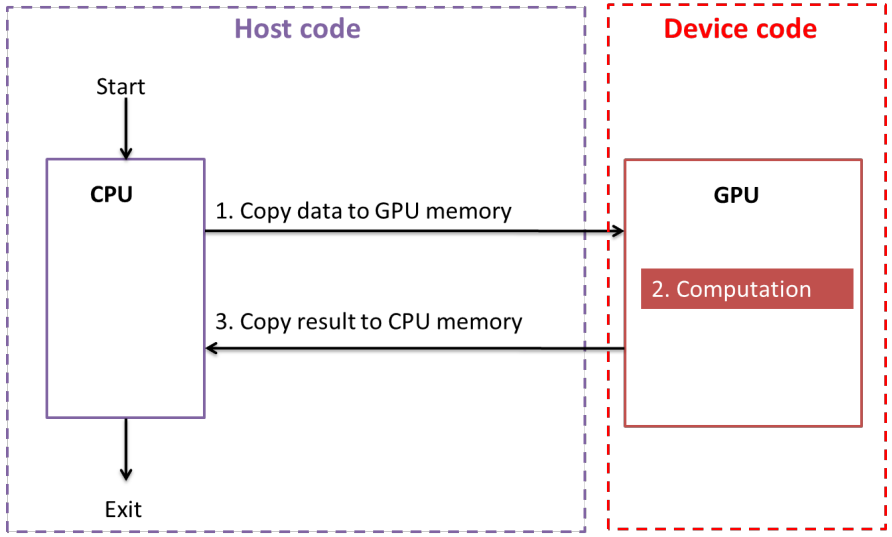
https://www.researchgate.net/figure/rCUDA-cluster-configurations_fig7_280883404

GPUs are computational devices; they require CPUs to be the host!

CUDA Programming Model

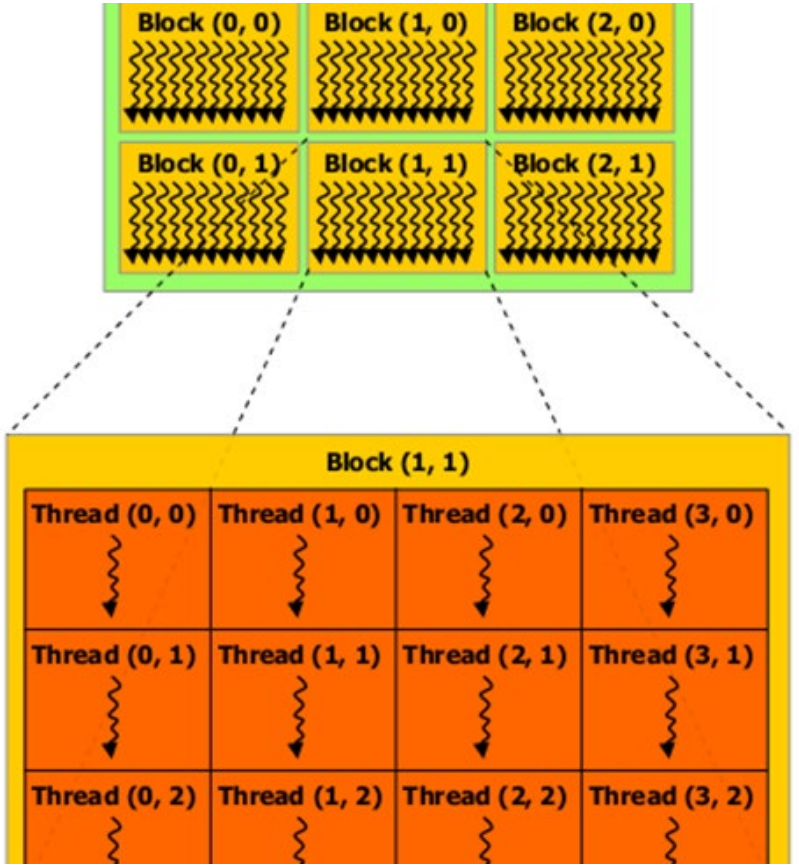
- A CUDA program consists of Host (CPU) and Device (GPU) components.
- The CPU:
  - Allocate and deallocate GPU memory
  - Transfer data between the CPU and the GPU
  - Launch GPU programs (kernels)
- The GPU:
  - Execute a kernel program with massive GPU threads
- CPU and GPU execution in parallel; explicit synchronization or through memory transfer (synchronized by default)





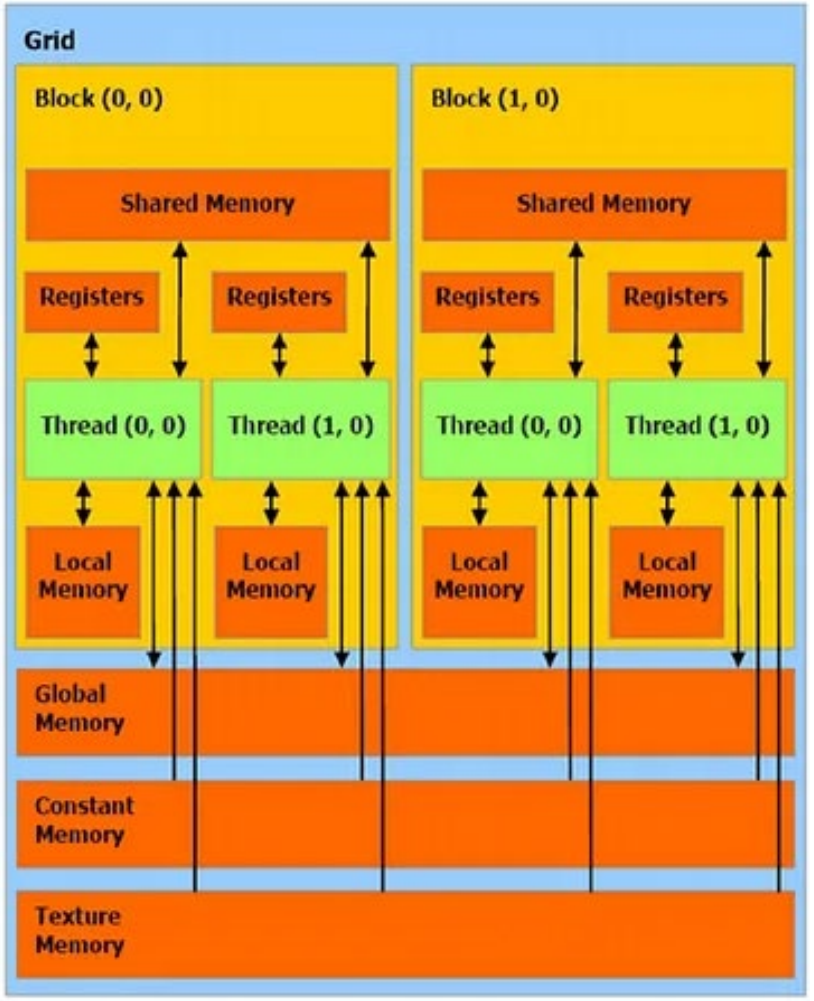
CUDA Threads

- A kernel program is executed by a thread **grid** specified by the user.
- A grid consists of 10s-1000s **thread block**s.
- A thread block contains 10s-1000s threads. Grids and blocks can be 1 to 3 dimensions.
- Each thread block runs in a single SM.
- Number of threads in a block should be set to a multiple of 32, the current warp size.
- A **warp** is the scheduling unit in the GPU, 32 threads with consecutive IDs.



CUDA Memory Hierarchy

- *Global memory*
  - Tens of gigabytes
  - High bandwidth high latency
  - Host-allocated GPU variables
  - Shared by all threads in the grid
- *Shared memory*
  - Tens of kilobytes
  - Residing in each streaming multiprocessor
  - Low access latency
  - Variables declared as “shared”
  - Shared by threads within a thread block
- *Registers*
  - Lowest latency
  - Local variables in GPU kernel programs
  - Private to each individual thread

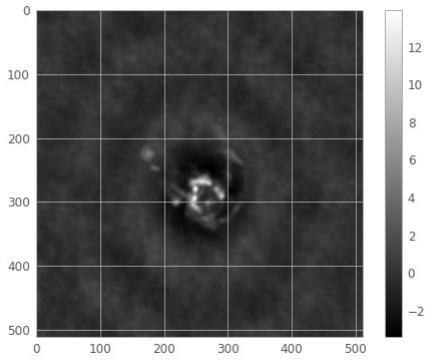


Accelerating DPA Tasks on CUDA

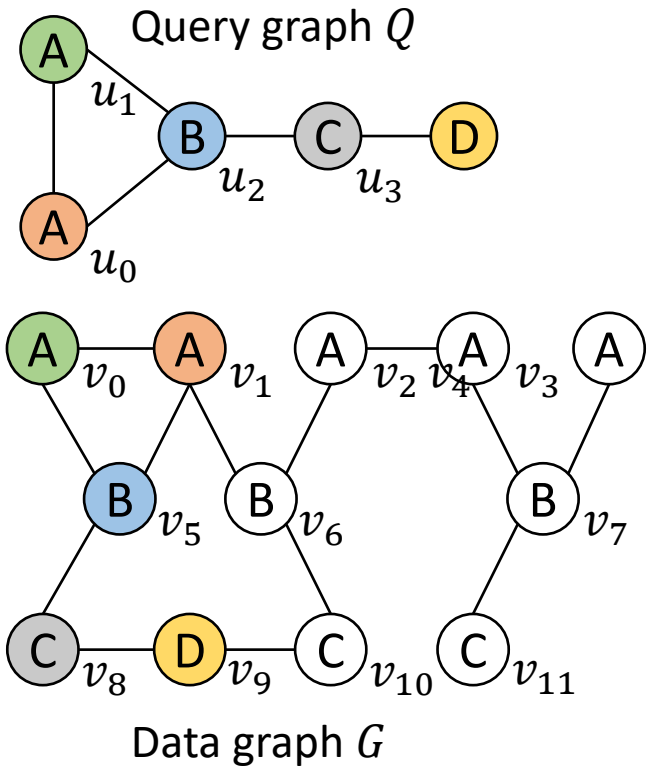
- Identify parallelisms
  - On GPU and CPU; between CPU and GPU; between processor, memory, IO.
- Design suitable data structures and algorithms
  - Various arrays for concurrent access; lock-free algorithms
- Maximize GPU occupancy
  - Increase number of threads
  - Reduce warp divergence
- Coalesced memory access for bandwidth
- Shared memory for latency

Our Recent Work as Examples

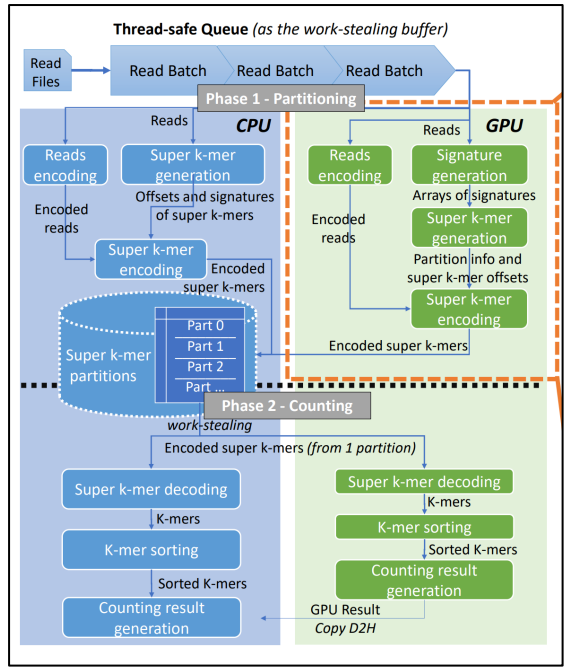
- cuGridder: Efficient Radio Interferometric Imaging on the GPU



- EGSM: Efficient GPU-Accelerated Subgraph Matching



- RapidGKC: GPU-accelerated K-mer Counting



Efficient Radio Interferometric Imaging on the GPU

eScience'22

Honghao Liu[1], Qiong Luo[1,2], and Feng Wang[3]

[1]The Hong Kong University of Science and Technology

[2]The Hong Kong University of Science and Technology (Guangzhou)

[3]Guangzhou University

Radio Interferometric Imaging

- Radio Interferometer: an array of radio antennas receiving the radio signals
- Visibility and Sky Brightness
  - V(u,v,w) – a complex function containing the information from a baseline
  - I(l,m) – the intensity of the source in the sky
- Imaging uses Fourier Transform to obtain I(l,m) from V(u,v,w)

Figure 1: Very Large Array[8]

$$V(u,v,w) = \sum\sum \frac{I(l,m)}{n} e^{-2\pi i(ul+vm+w(n-1))}$$

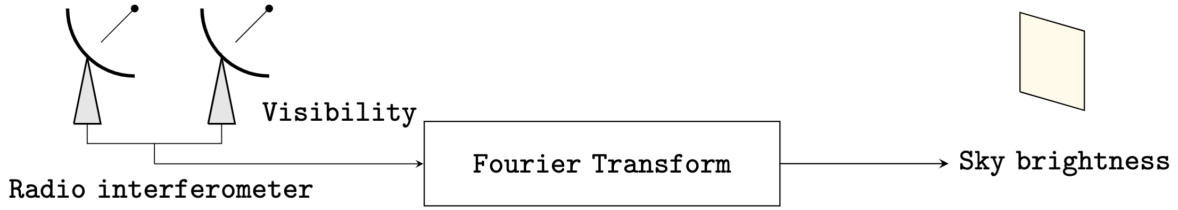


Figure 2: Visibility and Sky Brightness

State of the Art

- Previous work[1,2,3] proposed CPU-based accurate imaging algorithms
- W-gridder[4] paralellized the most accurate imaging algorithm[5] on the CPU
- FINUFFT[6] and cuFINUFFT[7]: the fastest Non-uniform Fourier Transform (NUFFT) on the CPU and the GPU respectively

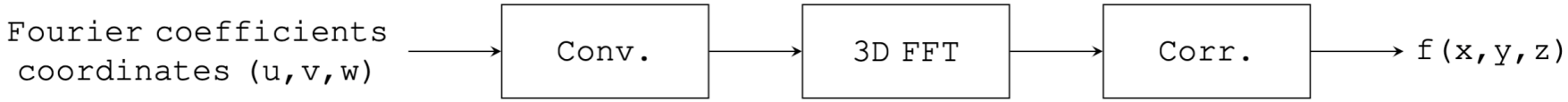


Figure 3: 3D NUFFT workflow

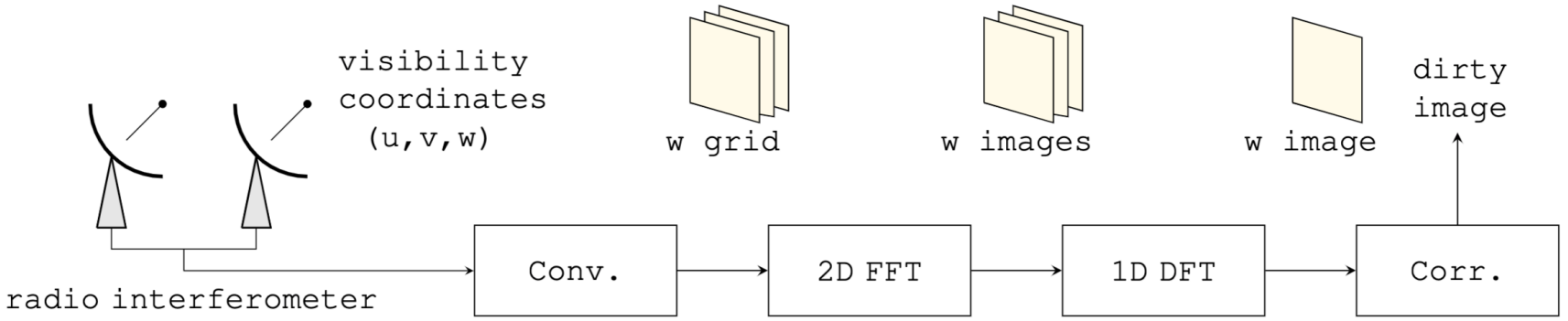


Figure 4: Gridding workflow

Our Work

- Propose **cuGridder**, a GPU-based CUDA C library for radio interferometric imaging
- Implement kernel programs for each step and optimize the memory access pattern on the GPU
- Achieve high performance
  - 5-10x faster than cuFINUFFT for the convolution
  - 3-5x faster than FINUFFT and cuFINUFFT for the NUFFT
  - 2-3x faster than the w-gridder for the entire gridding workflow
- Provide a python interface for astronomers to use the library

Workflow of cuGridder

- **Initialization**
  - Allocate host and device memory
  - Load data from the disk
- **Preprocessing** – convert matrices to 1D arrays
- **Coordinates transform** – shift and scale (u,v,w) to $[-\pi,\pi)$
- **Convolution**
  - Histogram, prefix sum and gather parallel primitives partition data based on (u,v,w)
  - Convolution primitive works on partitioned data
- **2D FFT** – computed by the NVIDIA cuFFT library
- **1D DFT** – transform along the *w* dimension
- **Correction** – remove the effect of the mask function from the convolution

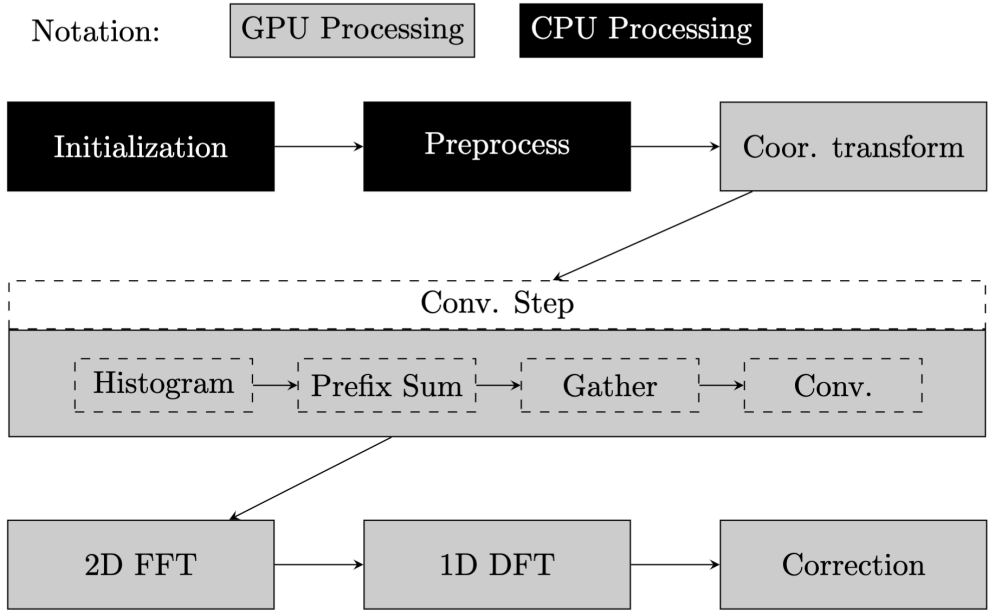


Figure 5: the gridding workflow of cuGridder

Convolution on the GPU

$$f \circ \phi = \int \phi(u - b_k) f(u) du$$

- Each thread corresponds to an output point b
- Partitioning
  - Histogram counts number of points in each bin
  - Prefix sum adds number of points of preceding bins
  - Scatter to location = in-bin index + prefix sum[Bin index]

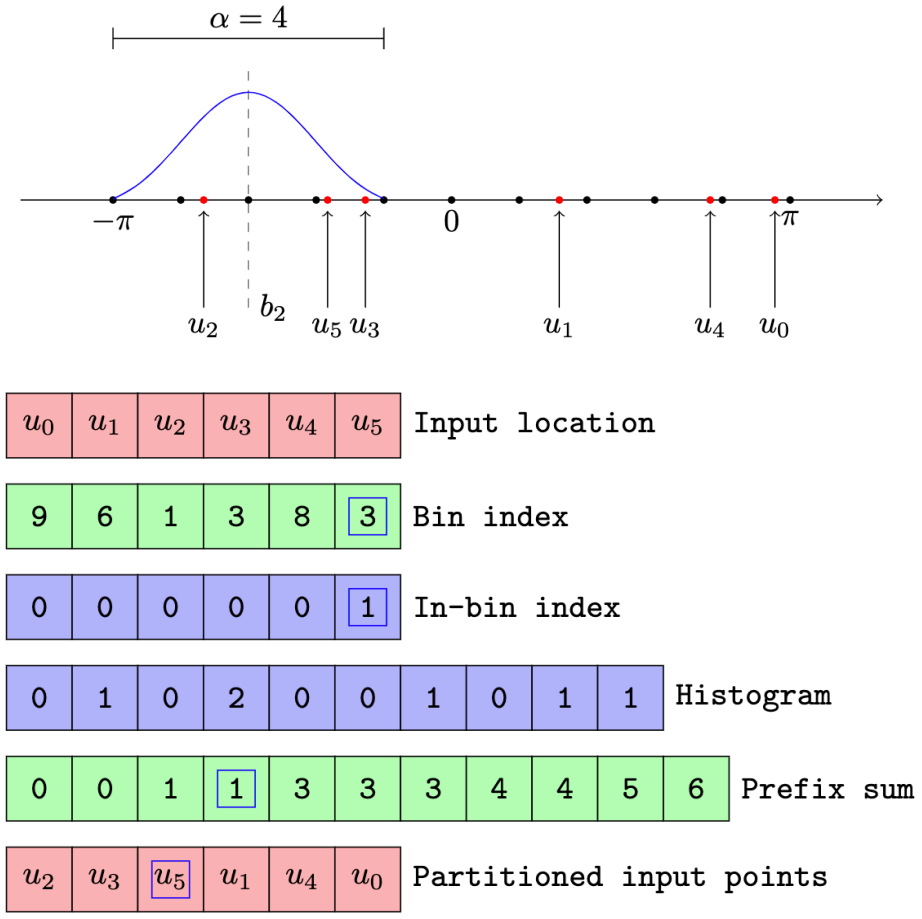
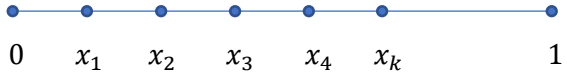


Figure 6: 1D example of mask function and partition

Mask Function Evaluation on the GPU

- The mask function evaluation is one of the heaviest computational tasks



$$\phi(x) = \begin{cases} e^{\beta(\sqrt{1-x^2}-1)} & |x| \le 1 \\ 0 & \text{otherwise} \end{cases}$$

- Taylor Series Approximation
  - Divide [0,1) into M equal segments,
  - For x in kth segment, $\phi(x)$ is evaluated by

$$\phi(x) = \underline{\phi(x_k)} + \underline{\phi'(x_k)}(x - x_k) + \cdots + \underline{\frac{\phi^{(n)}(x_k)}{n!}}(x - x_k)^n$$

$$R_n(x) = o((x - x_k)^n)$$

  - Save the coefficients into a lookup table, and load them into GPU shared memory for evaluation

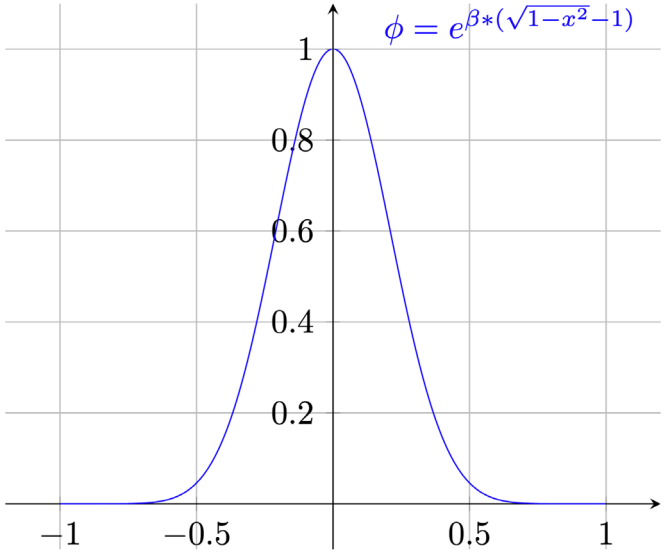


Figure 7: mask function

Summary on cuGridder

- Problem characteristics
  - Computation intensive
  - In-memory processing
  - Simple control flows and regular data access patterns
- Our method
  - Entire computation on the GPU after preprocessing
  - Massive thread parallelism to utilize the GPU
  - Data-parallel primitives to utilize memory bandwidth
  - Coefficient lookup table in the shared memory to reduce latency

Source code available at https://github.com/RapidsAtHKUST/cuGridder

Efficient GPU-Accelerated Subgraph Matching

SIGMOD'23

Xibo Sun[1], Qiong Luo[1,2]

[1]The Hong Kong University of Science and Technology

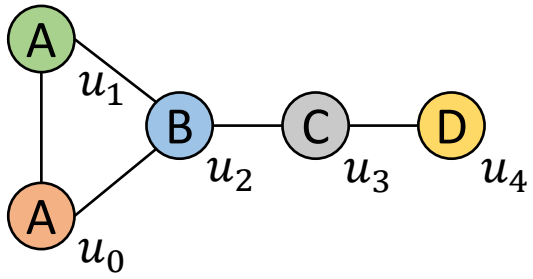
[2]The Hong Kong University of Science and Technology (Guangzhou)

Subgraph Matching

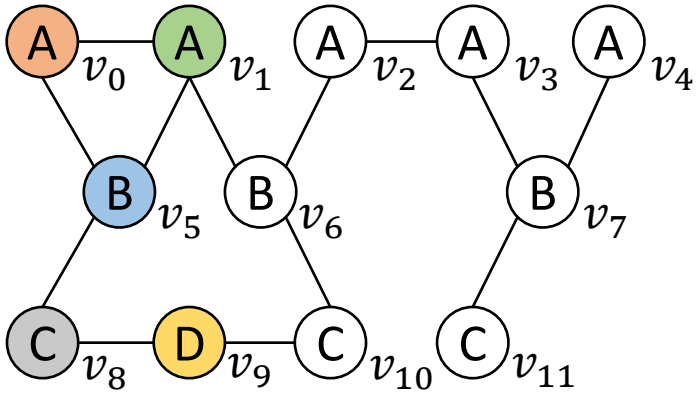
- Given vertex-labeled, undirected graphs Q (query graph) and G (data graph), find all subgraphs in G that are isomorphic to Q.
- NP-hard problem.

Subgraph Matching Example

- A, B, C, and D are vertex labels
- u_i, v_j are vertices
- Two matches
  - $\{(\boldsymbol{u_0}, \boldsymbol{v_0}), (\boldsymbol{u_1}, \boldsymbol{v_1}), (\boldsymbol{u_2}, \boldsymbol{v_5}), (\boldsymbol{u_3}, \boldsymbol{v_8}), (\boldsymbol{u_4}, \boldsymbol{v_9})\}$, and
  - $\{(u_0, v_1), (u_1, v_0), (u_2, v_5), (u_3, v_8), (u_4, v_9)\}$



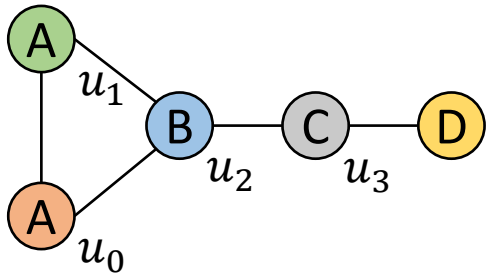
Query graph Q



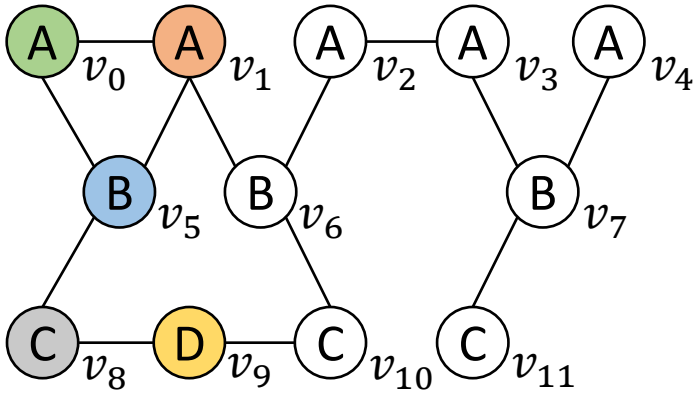
Data graph G

Subgraph Matching Example (Continued)

- A, B, C, and D are labels
- u_x, v_x are vertices
- Two matches
  - $\{(u_0, v_0), (u_1, v_1), (u_2, v_5), (u_3, v_8), (u_4, v_9)\}$, and
  - $\{(\boldsymbol{u_0, v_1}), (\boldsymbol{u_1, v_0}), (\boldsymbol{u_2, v_5}), (\boldsymbol{u_3, v_8}), (\boldsymbol{u_4, v_9})\}$



Query graph Q



Data graph G

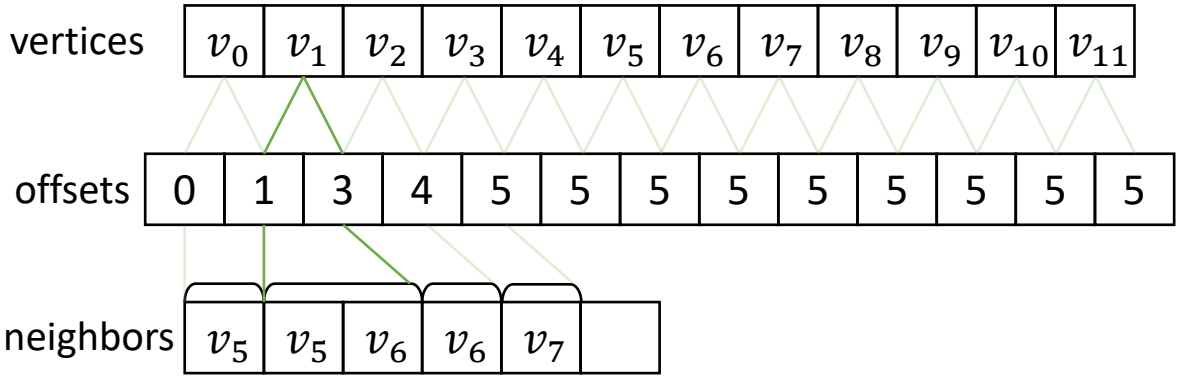
Subgraph Matching on CPU and GPU

- CPU-based algorithms:
  - **Sequential**: CFL[1], EmptyHeaded[2], DAF[3], Graphflow[4], VEQ[5], …
  - **Parallel**: PGX.ISO[6], pRI[7], CECI[8], …
  - Either graph exploration based (DFS enumeration) or join based
  - Effective heuristics for filtering out candidate vertices, ordering query vertices
  - Indices or auxiliary structures are constructed.
  - Highly optimized yet some cases still take long time
- GPU-based algorithms:
  - NEMO[9], GPSM[10], GunRockSM[11]. **Latest**: GSI[12], ALFTJ[13], CuTS[14]
  - Filtering and ordering methods less effective than CPU-based algorithms
  - BFS enumeration consumes a lot of memory

Existing Relation Storage for Graphs

- Trie (CSR) is commonly used on the CPU
  - Efficient retrieval of neighbors
  - **Many vertices and offsets for big relations**
  - **Expensive to update**

u_0	u_2
v_0	v_5
v_1	v_5
v_1	v_6
v_2	v_6
v_2	v_7

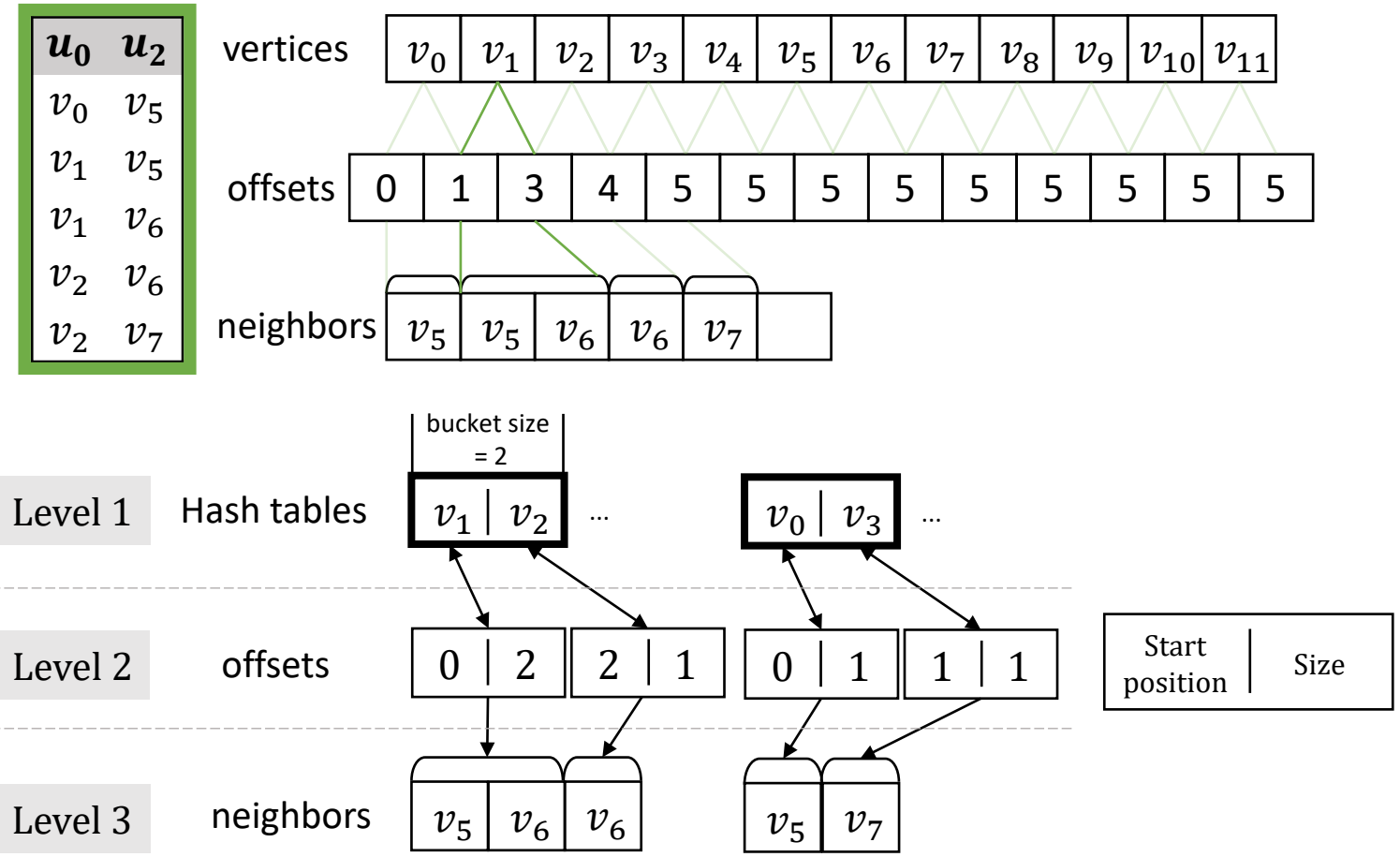


Our Relation Storage: Cuckoo Tries for G

- Trie (CSR) is commonly used on the CPU,
  - Efficient retrieval of neighbors
  - **Many vertices and offsets for big relations**
  - **Expensive to update**
- Cuckoo tries

  Level 1: Cuckoo hash tables
  - **Multiple** hash tables, $\boldsymbol{O(1)}$ search time, no warp divergence
  - **Bucket size** set to fully utilize the memory bandwidth

  Level 2: record #neighbors for each vertex

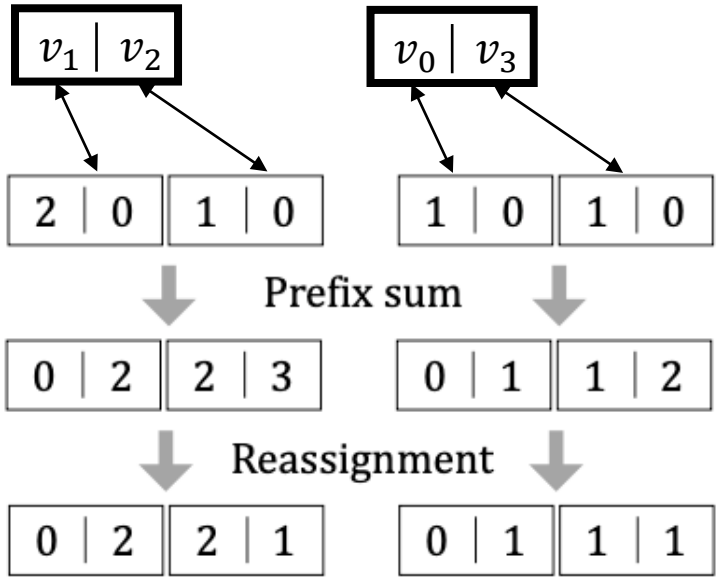


Operations on Cuckoo Tries

- Batch-insertion
  - Level 1: Push all vertices into hash tables

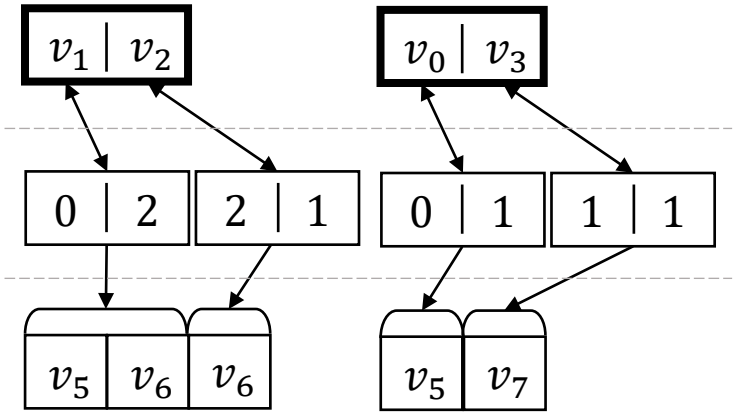
Operations on Cuckoo Tries

- Batch-insertion
  - Level 1: Push all vertices into hash tables
  - Level 2: Organize offsets
    - Count #neighbors, prefix sum, and reassignment



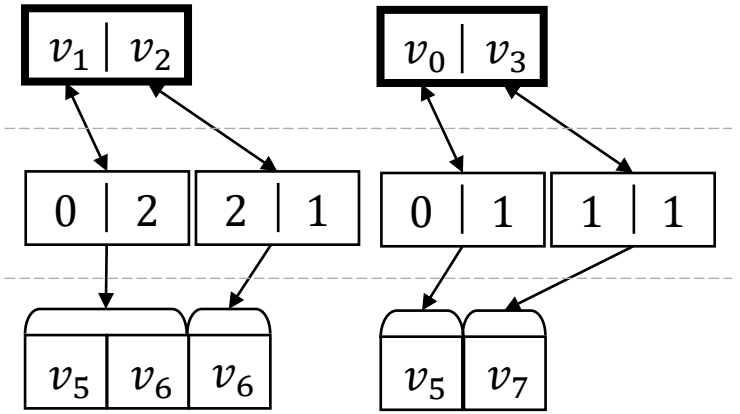
Operations on Cuckoo Tries

- Batch-insertion
  - Level 1: Push all vertices into hash tables
  - Level 2: Organize offsets
    - Count #neighbors, prefix sum, and reassignment
  - Level 3: Fill in neighbors



Operations on Cuckoo Tries

- Batch-insertion
  - Level 1: Push all vertices into hash tables
  - Level 2: Organize offsets
    - Count #neighbors, prefix sum, and reassignment
  - Level 3: Fill in neighbors



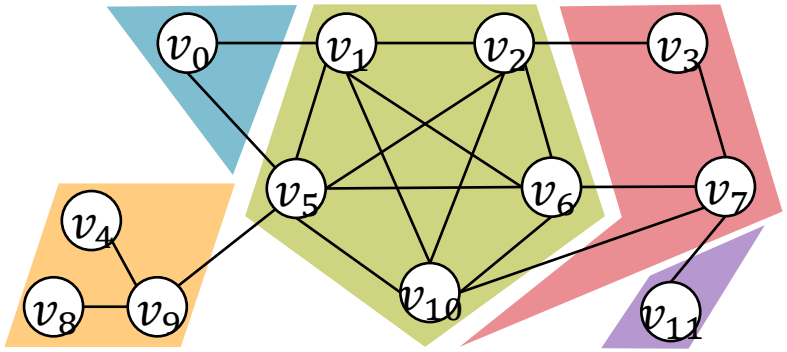
- Search of an edge $e(v, v')$
  - A thread finds v in Level 1
  - Go to Level 2 to find starting position of v's neighbor
  - Binary search of v' in the neighbor array of v
- Deletion of an edge $e(v, v')$
  - Same as searching $e(v, v')$
  - Deal with *holes* within neighbor arrays at the end of the entire construction

Enumeration: Parallel BFS vs DFS

- Parallel-BFS enumeration
  - Utilized by most GPU-based algorithms
  - In each step, all partial results are extended by one vertex concurrently
  - Large memory consumption and many memory accesses
- Parallel-DFS enumeration
  - Each thread extends one edge at a time until it finds a complete match or fails
  - Alleviate the memory consumption issue
  - Load imbalance between threads due to the irregularity of the search space

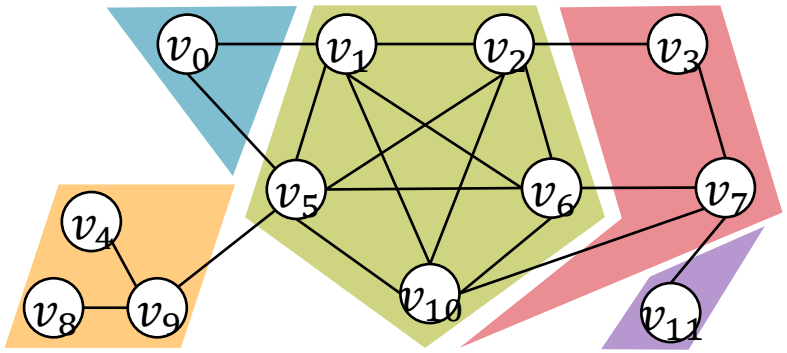
Hybrid BFS-DFS enumeration

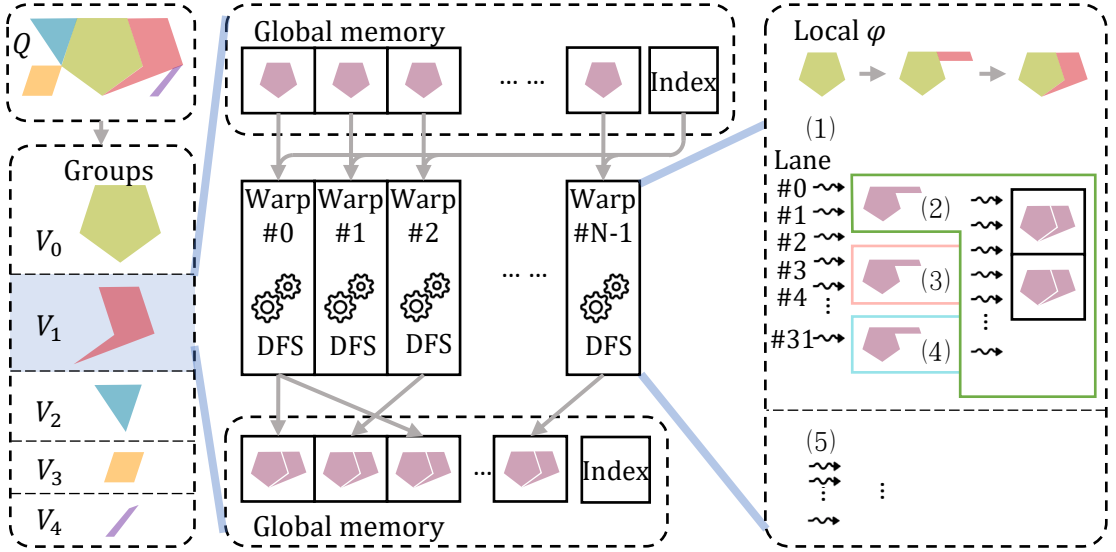
- Hybrid parallel BFS-DFS extension method
  - Organize vertices in Q into groups $(V_0, V_1, \ldots, V_n)$ based on the structure of Q
    - Dense vertex, then sparse vertices, and finally tree vertices



Hybrid BFS-DFS enumeration

- Hybrid parallel BFS-DFS extension method
  - Organize vertices in Q into groups $(V_0, V_1, \ldots, V_n)$ based on the structure of Q
    - Dense vertex, then sparse vertices, and finally tree vertices
  - Iterate over each group to extend current partial results
    - Extend vertices within the current group in DFS
    - Write partial results to global memory (BFS)





Hybrid BFS-DFS enumeration

- **Improve DFS**: A group is smaller than Q - more balanced
- **Improve BFS** by memory management
  - Store partial results in a cyclic queue
  - Before matching the group V_m, remove the partial results for V_{m-2} ($m \geq 2$).
  - If the queue is full at V_m, **roll back** to V_{m-1} results and match all remaining vertices in **DFS**.

Summary on EGSM

- Problem characteristics
  - Memory access intensive
  - Irregular memory access patterns
  - Complex control flows with branches
- Our method
  - A GPU-based data structure Cuckoo trie for storing candidate edges
  - Parallel construction and pruning routines on the Cuckoo trie
  - A BFS-DFS matching strategy with memory management

Source code available at https://github.com/RapidsAtHKUST/EGSM

RapidGKC: GPU-accelerated K-mer Counting

ICDE'24

Yiran Cheng[1], Xibo Sun[1], Qiong Luo[1,2]

[1]The Hong Kong University of Science and Technology

[2]The Hong Kong University of Science and Technology (Guangzhou)

K-mer Counting

- A ***k*-mer** refers to a length-k substring of a sequence.
- Genomic sequence fragments (strings of bases ‘A’, ‘C’, ‘G’, ‘T’) are called **reads.**
- A common routine in genomic data analysis is ***k-mer counting***, i.e., counting the number of occurrences of each unique k-mer.
- Some bioinformatics applications that require k-mer counting:

  genome assembly, genome profiling, and sequence alignment [1][2][3][4]

K-mer Counting is Expensive

- K-mer counting takes a lot of space.
- Current methods follow a **partitioning-and-counting paradigm**
  - **Phase 1 - Partitioning**: split all k-mers into multiple disjoint partitions
  - **Phase 2 - Counting**: count k-mers by partitions
  - Advantages:
    - Reduce the peak memory usage by storing temporary partition data on disk
    - Allow parallel processing among partitions
- K-mer counting is time-consuming.
  - Partitioning-and-counting incurs significant time in computation and IO

K-mer Counting (Phase 1 – Partitioning)

Phase 1 - Partitioning: **minimum substring partitioning (MSP)** [5]

- Split each read into k-mers, identify their ***minimizers,*** and generate super k-mers
  - **Minimizer**: the canonical smallest length-p substring of a k-mer
  - **Super k-mers** are generated on consecutive k-mers that have the same minimizer.
- Store generated **super k-mers** that have the same minimizer into one partition, rather than storing k-mers in partitions, for space saving.

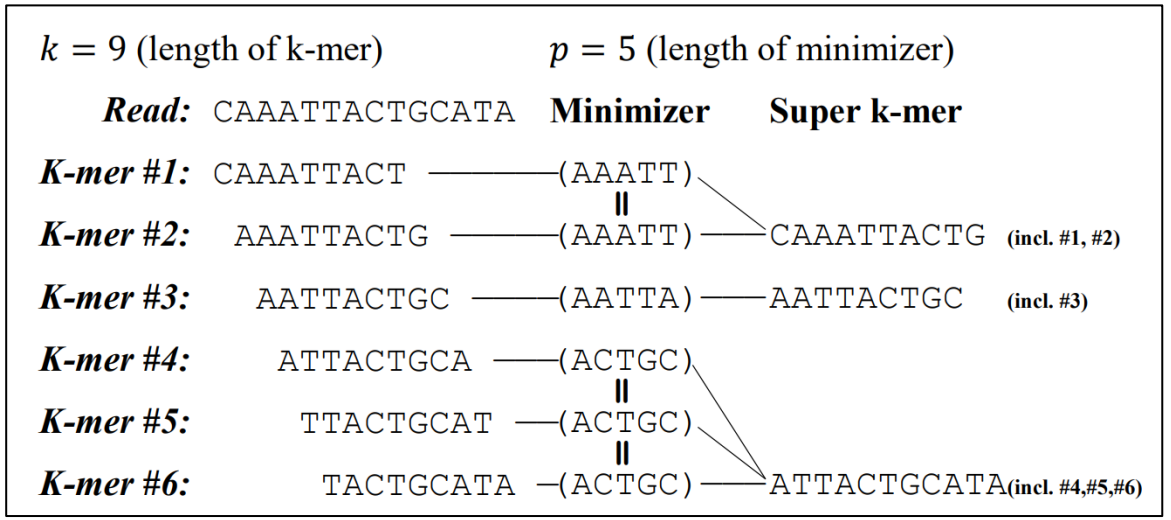


Fig. *A read, its k-mers, k-mers' minimizers, and generated super k-mers*

Storing k-mers: 6X9 = 54 bases
Storing super k-mers: 10+9+11 = 30 bases

K-mer Counting (Phase 2 – Counting)

- Phase 2 - Counting: count k-mers in each partition
  - Extract k-mers from super k-mers and then count
  - Counting approaches:
    - Sort (radix sort) – KMC2 [6], KMC3 [7] …
    - Hash table – CHTKC [8], MSPKC [9], Gerbil [10] …
    - Others (bloom filter, quotient filter, etc.) – Squeakr [11] …
  - Advantages of radix sort over hash table on GPUs:
    - Fixed memory requirement given a partition; avoid table size estimation and reallocation
    - A common parallel primitive on GPU
    - Faster counting performance than hashing in our tests
      - Over 2.5x faster than GPU-based hash table counting [12]

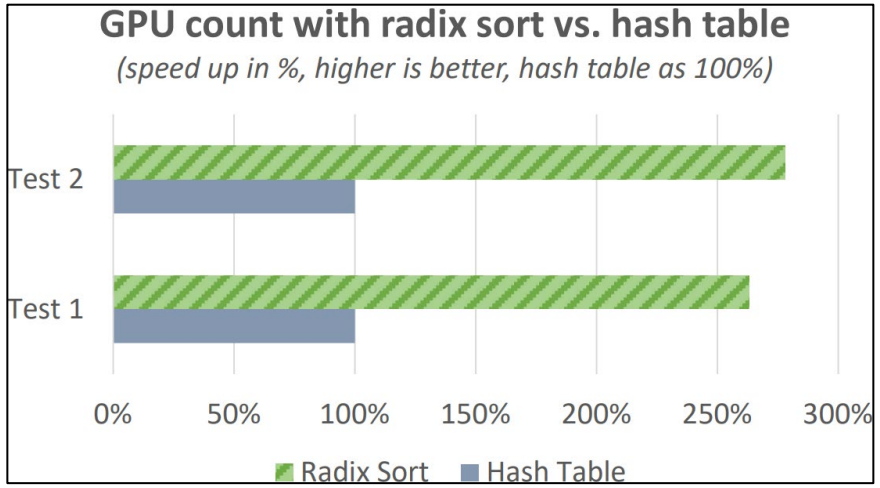


Fig. *Counting speedups of GPU radix sort vs. GPU-based hash table*

Problems in Existing Work and Our Solutions

- *Super k-mer generation took considerable time.*
  - Solution: GPU-based super k-mer generation
- *The signature rule was inefficient on the GPU due to branch divergence.*
  - Solution: a new signature rule that is as effective and has less branch divergence.
- *Super k-mer decoding was sequential.*
  - Solution: a new encoding scheme that supports fast parallel encoding and decoding.
- *Performance improvement was limited to in-memory counting.*
  - Solution: Overlapping IO and in-memory processing, GPU-CPU co-processing, and multi-GPU processing.

Workflow of RapidGKC

- Load reads into a thread-safe queue
- CPU and GPU worker threads load batch of reads from the queue
- Phase 1 – Partitioning
  - Read encoding
  - Super k-mer generation
    - Signature calculation
    - Super k-mer offset generation
  - Super k-mer encoding
  - Store super k-mers into corresponding partitions
- Phase 2 – Counting
  - Load super k-mers from a partition
  - Decode super k-mers and extract k-mers
  - Sort all k-mers
  - Count number of duplicates of each k-mer

- **underlined steps: GPU-accelerated*

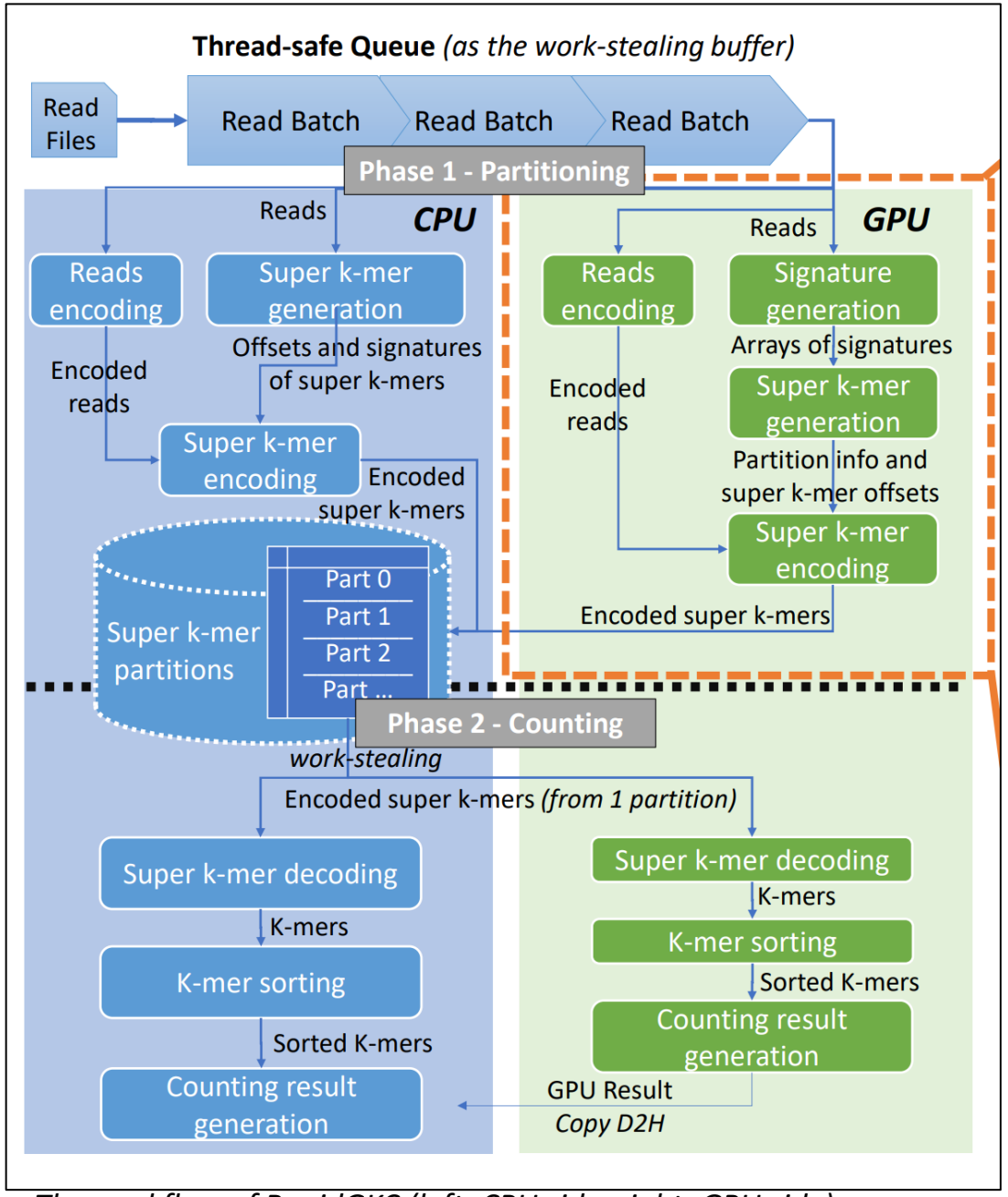


Fig. *The workflow of RapidGKC (left: CPU side, right: GPU side).*

Signature Rule

Problems with the minimizer in MSP:

- It generates relatively short super k-mers and thereby results in large temporary data sizes.
- The generated partitions are of skewed sizes because the minimizers starting with consecutive “A”s are likely to be the alphabetically smallest.

Existing solution:

- Using **signature** rather than minimizer.
- The signature proposed by KMC2 [6]:
- the canonically minimum length-p substring of a k-mer that can pass the **signature rule** that it neither starts with ”AAA” or ”ACA” nor contains ”AA” at any position except the beginning.

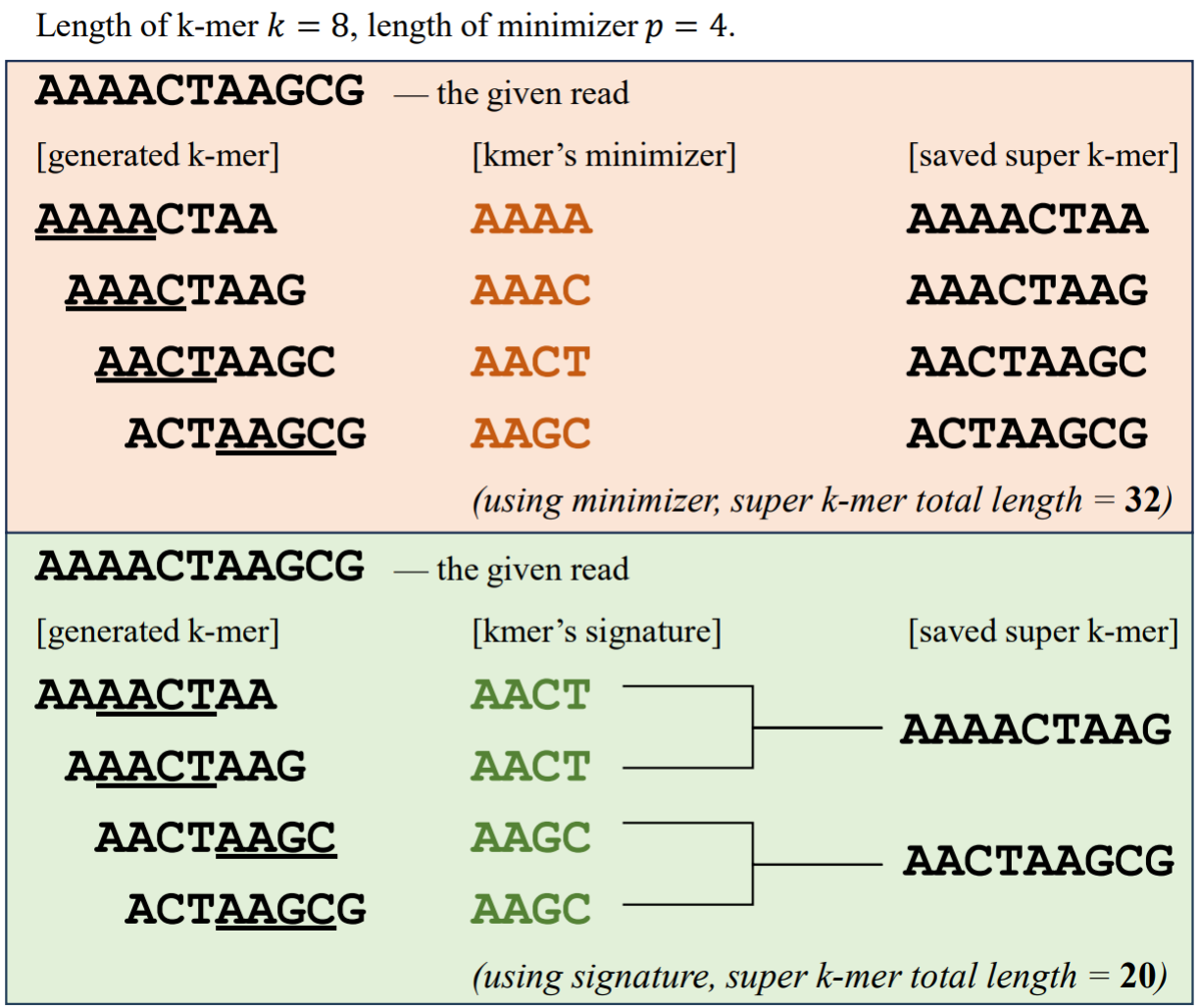


Fig. *An example comparing conventional minimizer and signature over their generated super k-mers.*

Our Improved Signature Rule

- Problem of the existing signature rule:
  - Costly to check whether “AA” appears at any position
  - Causes branch divergence and runs slow on the GPU
- Our solution in RapidGKC: a new signature rule
  - No “AAA”, “ACA”, “CAA”, or “CCA” at the beginning, and no “AAA” at the end is allowed.
  - Constant time complexity
  - Reduce branch divergence

Encoding Schemes

- Encoding in existing k-mer counting tools
  - Length-and-data encoding:
    - Each base (A, C, T, G) is encoded with two bits
    - Store the length and bases of super k-mers consecutively
  - Problems:
    - Only support sequential decoding
    - Causes a lot of decoded data to be transferred from CPU to GPU for subsequent processing on the GPU
    - Significant component(30%, in our experiments) of in-memory processing time with other components accelerated by GPU

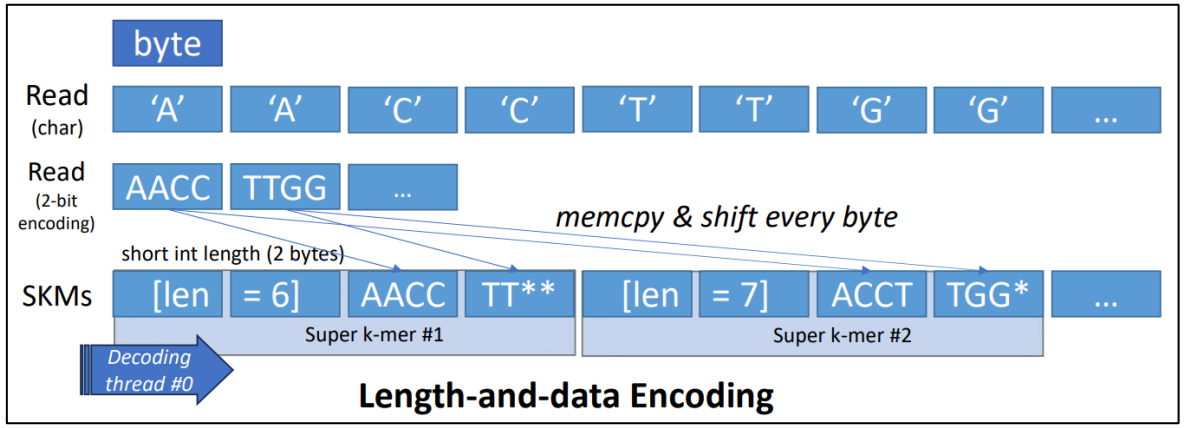


Fig. *The length-and-data super k-mer encoding in existing k-mer counting methods.*

Problems of existing parallel encoding methods

E.g., CSR, StreamVByte [13]

Lengths (offsets) and data are stored in inter-related arrays, so synchronization cost is high.

Our Encoding Scheme

Two control bits and six data bits (three bases) in one byte

Control bits indicate how many bases are stored in the current byte; the first and last bytes may have "*" fillers.

Multiple threads can start decoding from any position of the data

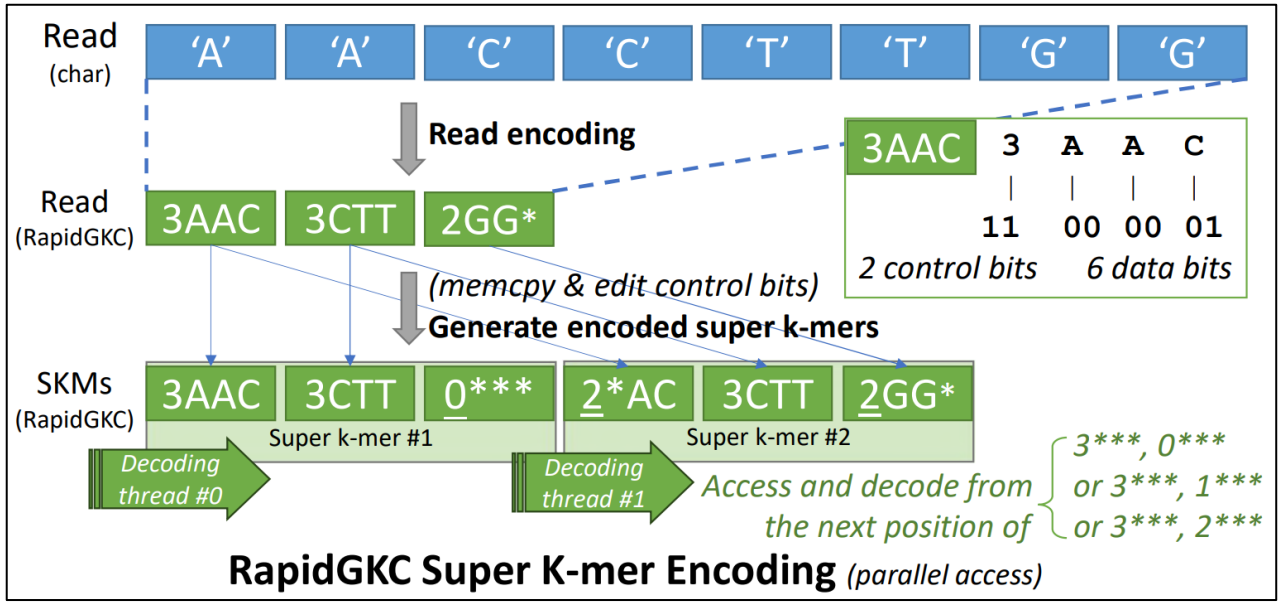


Fig. *Our proposed super k-mer encoding method supports parallel encoding and decoding.*

Advantages

- Support parallel encoding and decoding, so super k-mers can be decoded on the GPU efficiently
  - GPU parallel decoding is **40x** faster than single CPU thread decoding
- Reduce the size of data transfer from host to GPU in Phase 2
  - Transfer the super k-mers rather than all the extracted k-mers

Disadvantages

- Some wasted space in the first and last bytes of each super k-mer
- Some cost for each thread to find the first byte of a super k-mer when starting from an arbitrary point in the super k-mers

Summary on RapidGKC

- Problem characteristics
  - Memory capacity bound
  - Memory access intensive
  - Complex workflows involving IO, partitioning, and multi-step processing
  - Multicore CPUs and GPUs have comparable performance at times
- Our solution: an end-to-end GPU-accelerated k-mer counting system
  - Develop GPU kernels for partitioning and counting phases respectively
  - Employ both CPUs and GPUs as parallel workers
  - Overlap IO and in-memory processing
  - A new encoding scheme that supports fast parallel encoding and decoding
  - A new signature rule that reduces branch divergence on the GPU.

  Code available at https://github.com/cyr20040123/RapidGKC

Concluding Remarks

- GPU acceleration for DPA can be effective.
  - Understand problem characteristics
  - Design suitable data structures, algorithms, and workflows
  - Utilize data parallel primitives and shared memory
  - Reduce warp divergence
  - End-to-end system development and evaluation
- It involves considerable engineering effort due to problem diversity.
- Promising direction with technology advances and applications

Group Github: https://github.com/RapidsAtHKUST

Acknowledgment

- Yiran Cheng (PhD 2025 expected) for RapidKGC
- Xibo Sun (PhD 2024 expected) for EGSM
- Honghao Liu (MPhil 2022) for cuGridder

HKUST, Hong Kong, since 1991
https://hkust.edu.hk/

HKUST (Guangzhou), China, since 2022
https://www.hkust-gz.edu.cn/

Come and visit us!

References by cuGridder

[1] T. J. Cornwell, K. Golap, and S. Bhatnagar, "The noncoplanar baselines effect in radio interferometry: The w-projection algorithm," IEEE Journal of Selected Topics in Signal Processing, vol. 2, no. 5, pp. 647–657, 2008.

[2] T. Cornwell, M. Voronkov, and B. Humphreys, “Wide field imaging for the square kilometre array,” Proc SPIE, vol. 8500, 07 2012.

[3] A. R. Offringa and et al., "wsclean: an implementation of a fast, generic wide-field imager for radio astronomy," Monthly Notices of the Royal Astronomical Society, vol. 444, no. 1, p. 606–619, Aug 2014.

[4] Haoyang Ye, Stephen F Gull, Sze M Tan, Bojan Nikolic, "High accuracy wide-field imaging method in radio interferometry," Monthly Notices of the Royal Astronomical Society, Volume 510, Issue 3, March 2022, Pages 4110–4125, https://doi.org/10.1093/mnras/stab3548

[5] P. Arras, M. Reinecke, R. Westermann, and T. A. Enßin, "Efficient wide-field radio interferometry response," Astronomy & Astrophysics, vol. 646, p. A58, Feb 2021.

[6] Barnett, Alexander H., Jeremy Magland, and Ludvig af Klinteberg. "A parallel nonuniform fast Fourier transform library based on an “exponential of semicircle" kernel." SIAM Journal on Scientific Computing 41.5 (2019): C479-C504.

[7] Y. Shih, G. Wright, J. Anden, J. Blaschke, and A. H. Barnett, "cufinufft: a load-balanced gpu library for general-purpose nonuniform ffts," in 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

[8] https://public.nrao.edu/telescopes/vla/

[9] B. Veenboer, M. Petschow, and J. W. Romein, "Image-domain gridding on graphics processors," in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2017, pp. 545–554.

References by EGSM

[1] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Effcient Subgraph Matching by Postponing Cartesian Products. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. ACM, 1199–1214.

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. ACM, 431–446.

[3] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Ecient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019. ACM, 1429-1446

[4] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. Proc. VLDB Endow. 12, 11 (2019), 1692–1704.

[5] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021. ACM, 925–937.

[6] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. 2014. PGX.ISO: Parallel and Ecient In-Memory Engine for Subgraph Isomorphism. In Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014. CWI/ACM, 5:1–5:6.

[7] Raphael Kimmig, Henning Meyerhenke, and Darren Strash. 2017. Shared Memory Parallel Subgraph Enumeration. In IPDPSW.

[8] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019. ACM, 1447–1462.

[9] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiaoli Li. 2017. Network Motif Discovery: A GPU Approach. IEEE Trans. Knowl. Data Eng. 29, 3 (2017), 513–528.

[10] Ha Nguyen Tran, Jung-Jae Kim, and Bingsheng He. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In Database Systems for Advanced Applications - 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9049). Springer, 299–315.

[11] Leyuan Wang and John D. Owens. 2020. Fast Gunrock Subgraph Matching (GSM) on GPUs. CoRR abs/2003.01527 (2020). arXiv:2003.01527 https://arxiv.org/abs/2003.01527

[12] Li Zeng, Lei Zou, M. Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly Subgraph Isomorphism. In 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020. IEEE, 1249–1260.

[13] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. VLDB J. 31, 3 (2022), 529–553.

[14] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021. ACM, 69:1–69:14.

References by RapidGKC

[1] S. C. Manekar and S. R. Sathe, “A benchmark study of k-mer counting methods for high-throughput sequencing,” *GigaScience*, vol. 7, no. 12, p. giy125, 2018.

[2] J. Ruan and H. Li, “Fast and accurate long-read assembly with wtdbg2,” *Nature methods*, vol. 17, no. 2, pp. 155–158, 2020.

[3] T. R. Ranallo-Benavidez, K. S. Jaron, and M. C. Schatz, “Genomescope 2.0 and smudgeplot for reference-free profiling of polyploid genomes,” *Nature communications*, vol. 11, no. 1, p. 1432, 2020.

[4] D. Kleftogiannis, P. Kalnis, and V. B. Bajic, “Progress and challenges in bioinformatics approaches for enhancer identification,” *Briefings in bioinformatics*, vol. 17, no. 6, pp. 967–979, 2016.

[5] Y. Li, P. Kamousi, F. Han, S. Yang, X. Yan, and S. Suri, “Memory efficient minimum substring partitioning,” *Proceedings of the VLDB Endowment*, vol. 6, no. 3, pp. 169–180, 2013.

[6] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, “Kmc 2: fast and resource-frugal k-mer counting,” *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2015.

[7] M. Kokot, M. Długosz, and S. Deorowicz, “Kmc 3: counting and manipulating k-mer statistics,” *Bioinformatics*, vol. 33, no. 17, pp. 2759– 2761, 2017.

[8] J. Wang, S. Chen, L. Dong, and G. Wang, “Chtkc: a robust and efficient k-mer counting algorithm based on a lock-free chaining hash table,” *Briefings in Bioinformatics*, vol. 22, no. 3, p. bbaa063, 2021.

[9] Y. Li et al., “Mspkmercounter: a fast and memory efficient approach for k-mer counting,” *arXiv preprint arXiv:1505.06550*, 2015.

[10] M. Erbert, S. Rechner, and M. Muller-Hannemann, “Gerbil: a fast ¨ and memory-efficient k-mer counter with gpu-support,” *Algorithms for Molecular Biology*, vol. 12, pp. 1–12, 2017.

[11] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, “Squeakr: an exact and approximate k-mer counting system,” *Bioinformatics*, vol. 34, no. 4, pp. 568–575, 2018.

[12] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and ¨ B. Schmidt, “General-purpose gpu hashing data structures and their application in accelerated genomics,” *Journal of Parallel and Distributed Computing*, vol. 163, pp. 256–268, 2022.

[13] D. Lemire, N. Kurz, and C. Rupp, “Stream vbyte: Faster byte-oriented integer compression,” *Information Processing Letters*, vol. 130, pp. 1–6, 2018.