THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

THE DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING 計算機科學及工程學系

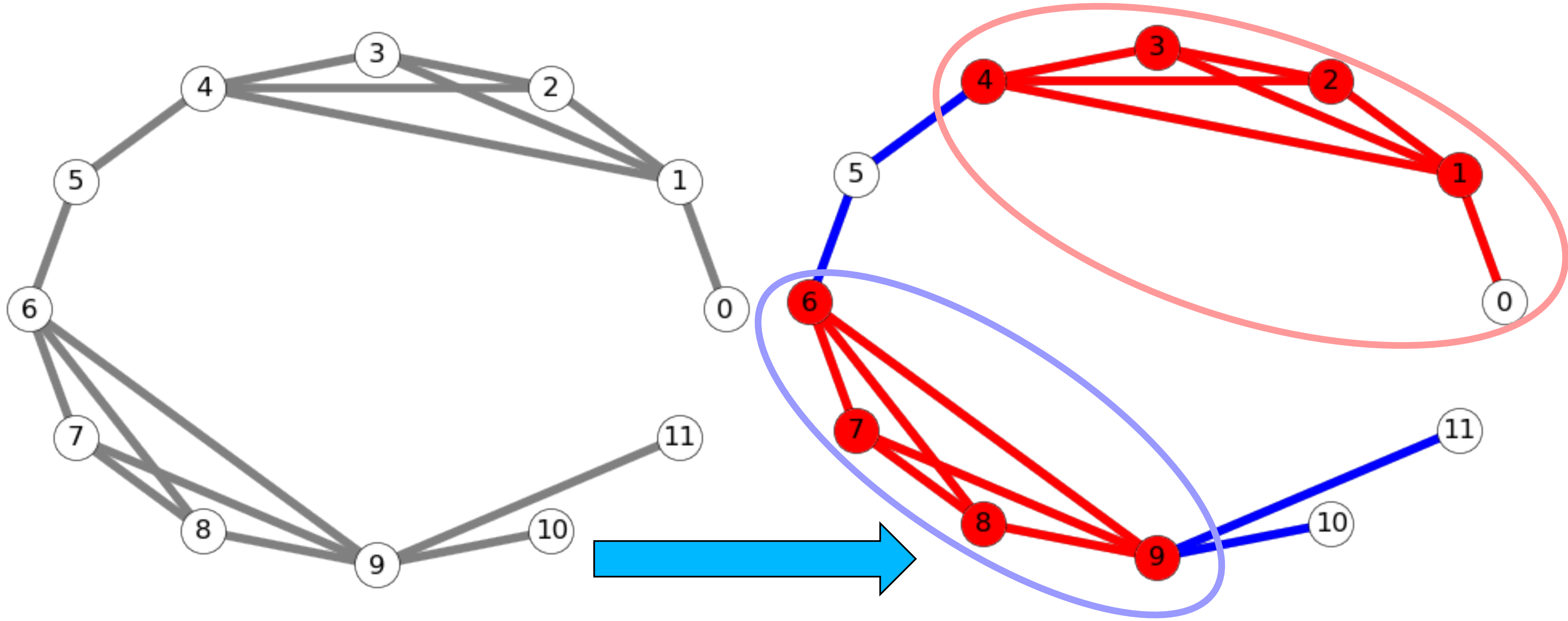
Accelerating All-Edge Common Neighbor Counting on Three Processors

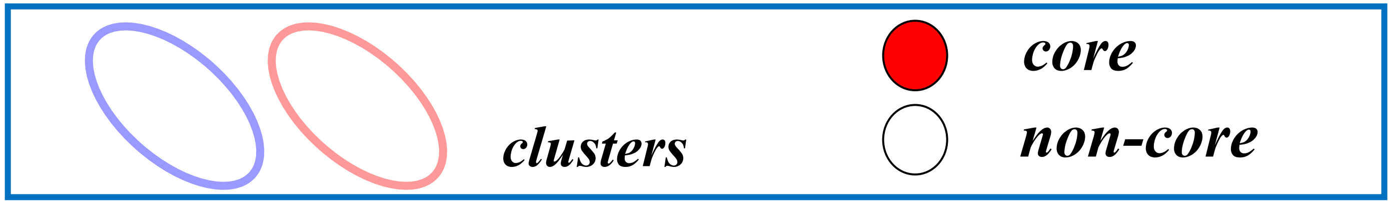
Yulin Che, Zhuohang Lai, Shixuan Sun, Qiong Luo, Yue Wang
Hong Kong University of Science and Technology

Outline

Graph Structural Clustering

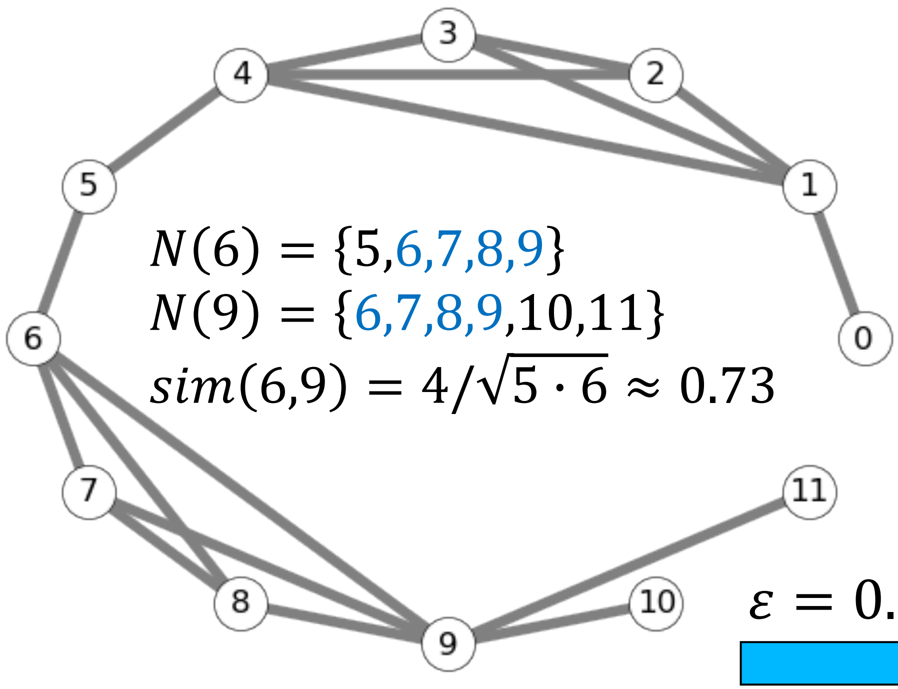
- **Basic Components**
  - utilize **structural similarity** among vertices for clustering
  - identify **clusters** and **vertex roles** (cores, non-cores)





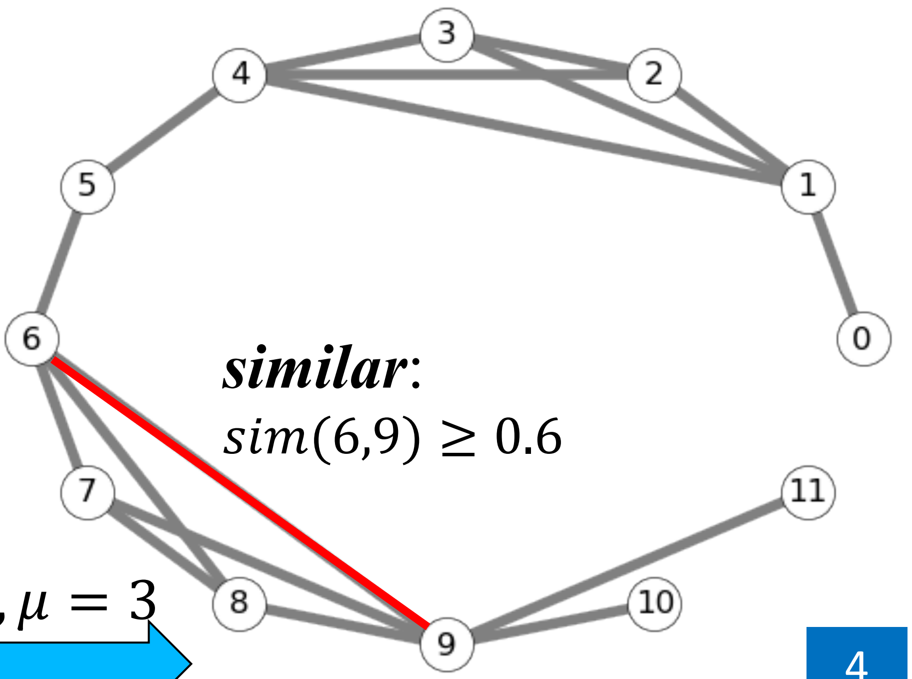
Example: SCAN [Xu+, KDD'07]

- **Structural Similarity Computation**
  - based on neighbors of two vertices u and v (cosine measure):
    - $sim(u,v) = |N(u) \cap N(v)| / \sqrt{|N(u)| \cdot |N(v)|}$
  - u and v are similar neighbors, if
    - they are connected (adjacent)
    - their structural similarity $sim(u,v) \geq \varepsilon$



$N(6) = \{5,6,7,8,9\}$
$N(9) = \{6,7,8,9,10,11\}$
$sim(6,9) = 4/\sqrt{5 \cdot 6} \approx 0.73$

$\varepsilon = 0.6, \mu = 3$



similar:
$sim(6,9) \geq 0.6$

Computation of SCAN

- **Structural Similarity Computation**
  - based on neighbors of two vertices u and v (cosine measure):
    - $sim(u,v) = |N(u) \cap N(v)| / \sqrt{|N(u)| \cdot |N(v)|}$
  - u and v are ***similar neighbors***, if
    - they are connected (adjacent)
    - their ***structural similarity*** $sim(u,v) \geq \varepsilon$

involve intensive set intersections

only intersect for adjacent vertices

Recent Work Improving SCAN

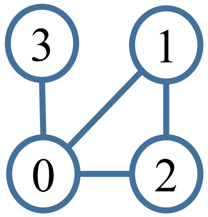
- **Reducing the number of set intersections**
  - Basic Idea: prune some set intersection computations
  - Sequential Algorithms: pSCAN [Chang+, ICDE’16]
  - Parallel Algorithms: anySCAN [Mai+, ICDE’17], SCAN-XP [Takahashi, NDA’17], ppSCAN [Che+, ICPP’18]
  - Results: workload (set intersection) reduction computation is quite trivial but helpful

Issues and Solutions in Improving SCAN

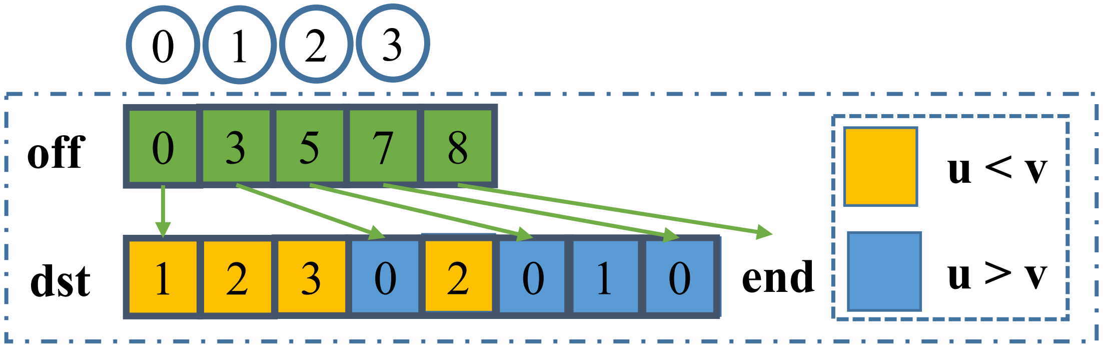
- **Issues:** for a fixed dataset, given different parameters, we need to **recompute the set intersection** for the same edge
- **Our Solution:** compute the common neighbor count **once** for all the edges

Problem: All-Edge Common Neighbor Counting

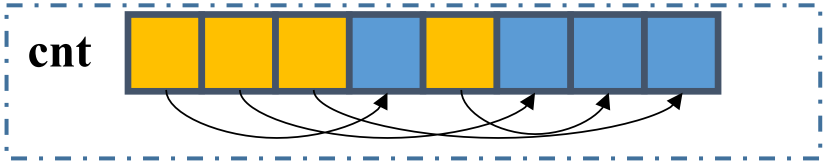
- **Problem Statement:** Given an undirected graph, compute the common neighbor counts of each adjacent vertex pairs
- **Input:** a graph in a Compressed Sparse Rows (CSR) format
- **Output**: common neighbor counts for each adjacent vertex pair in the CSR



(a) example graph



(b) CSR representation



cnt[e(u,v)] = cnt[e(v,u)]

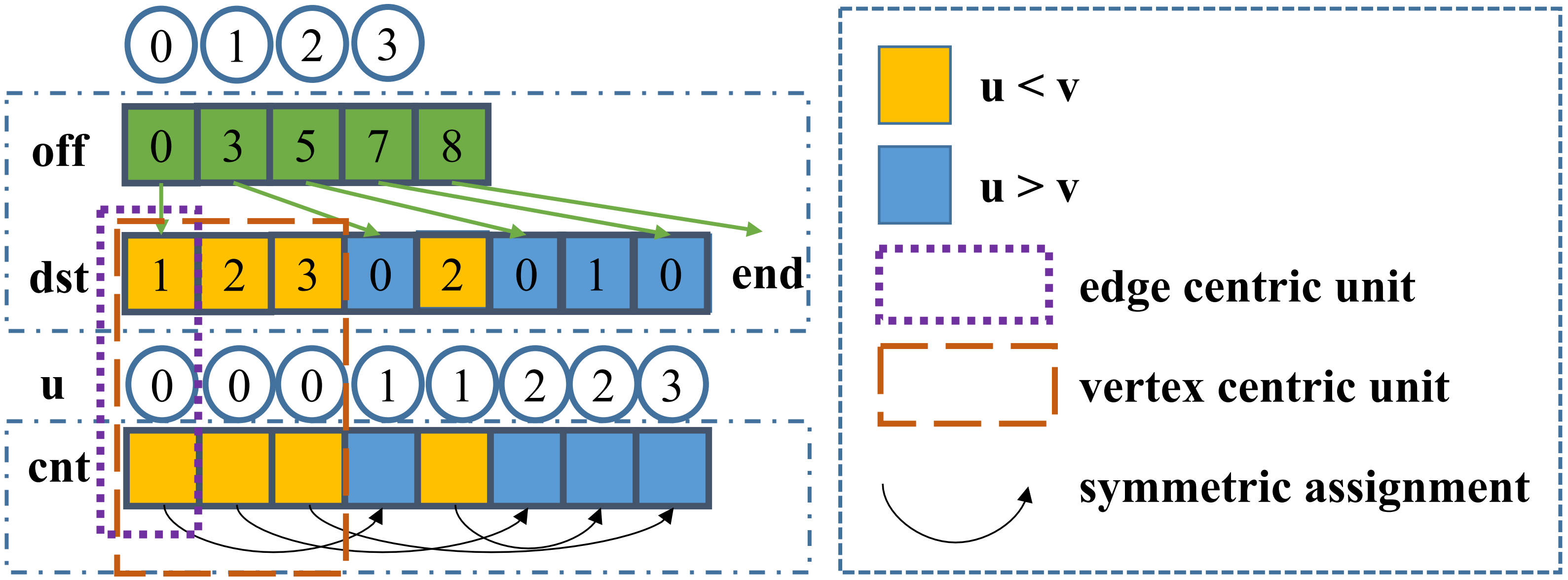
e.g., cnt[e(0,1)] = cnt[e(1,0)]

(c) output array

Outline

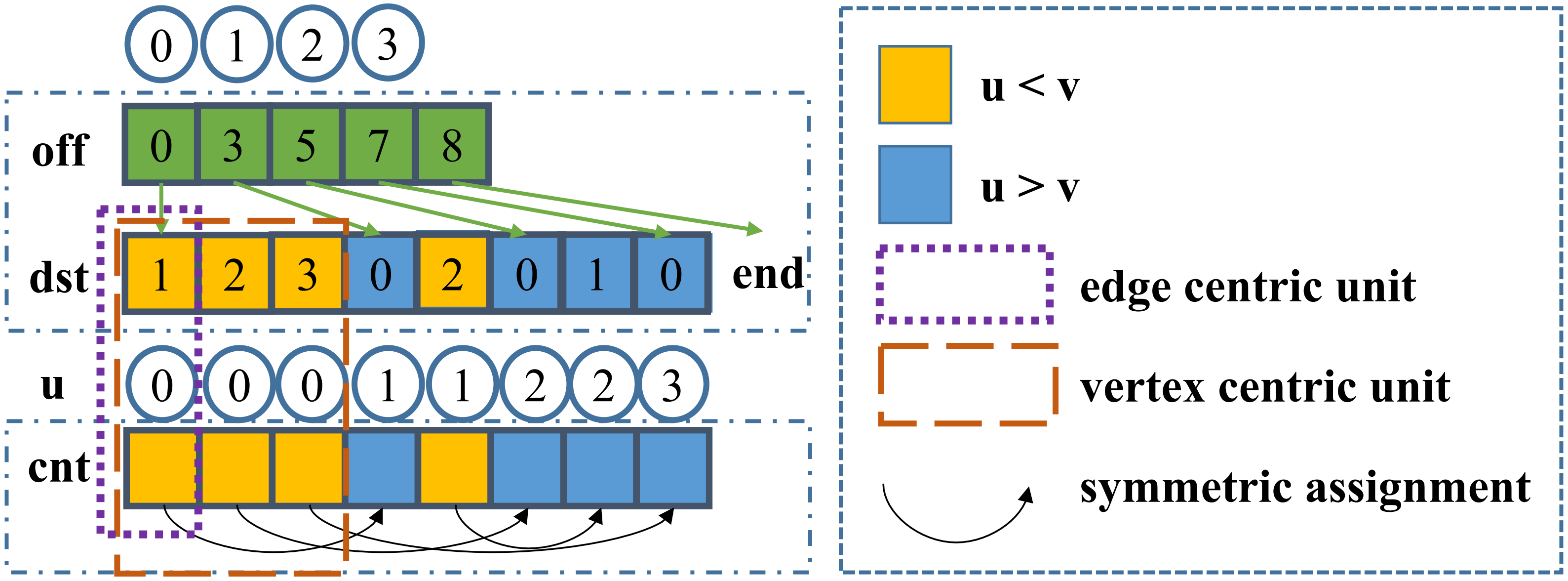
Our Solution

- **Two Algorithms**
  - **MPS**: a **M**erge based algorithm with **P**ivot **S**kip optimization on two sorted arrays to compute counts (edge-centric computation unit)
  - **BMP**: a **bitmap** index-based algorithm to dynamically construct an index on one array and loop over the other array and lookup the bitmap index to compute counts (vertex-centric computation unit)



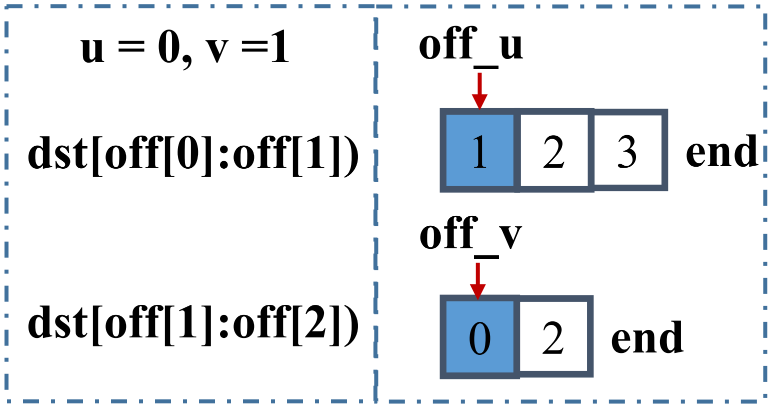
Common Optimizations

- **Symmetric Assignment**: Utilize the symmetricity (**cnt[e(u,v)] = cnt[e(v,u)]**) to avoid redundant computations
- **Degree Skew Handling**: Optimize the algorithm to make the complexity of each intersection relates to the smaller degree vertex only (**O(min(du, dv))**)

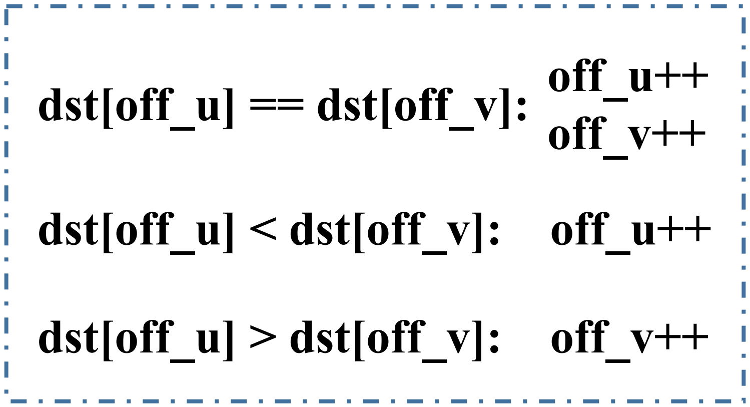


MPS’s Edge-Centric Unit Computation

- Scalar merge over two sorted arrays
    - conduct a while loop to **count the match** and **increment the offset** for the smaller value array until the end of one array is reached



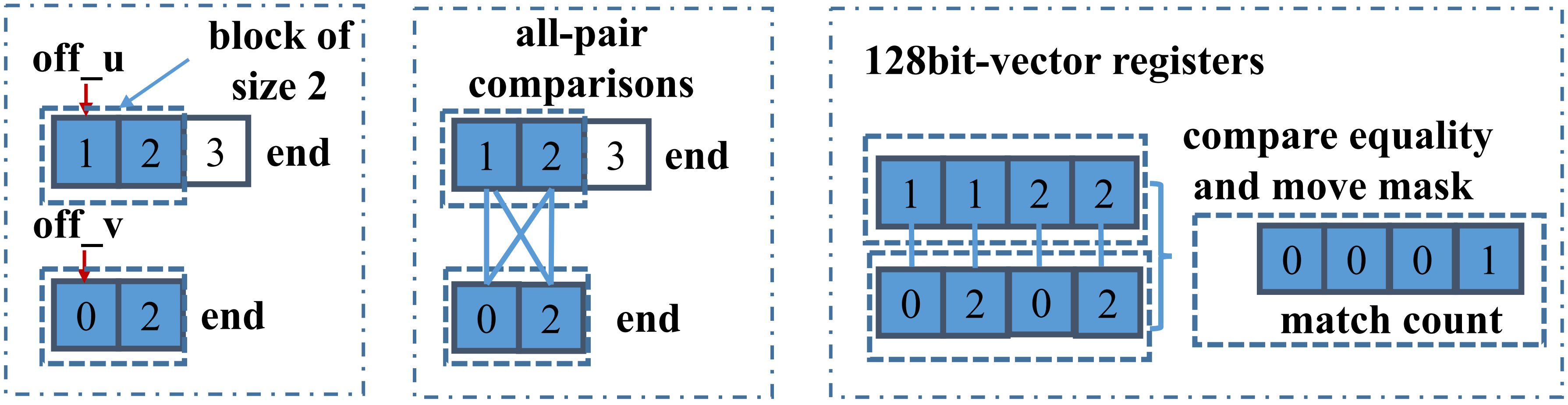
(a) initial state


dst[off_u] == dst[off_v]: off_u++ off_v++
dst[off_u] < dst[off_v]: off_u++
dst[off_u] > dst[off_v]: off_v++

(b) offset advance rules

Improving Scalar Merge

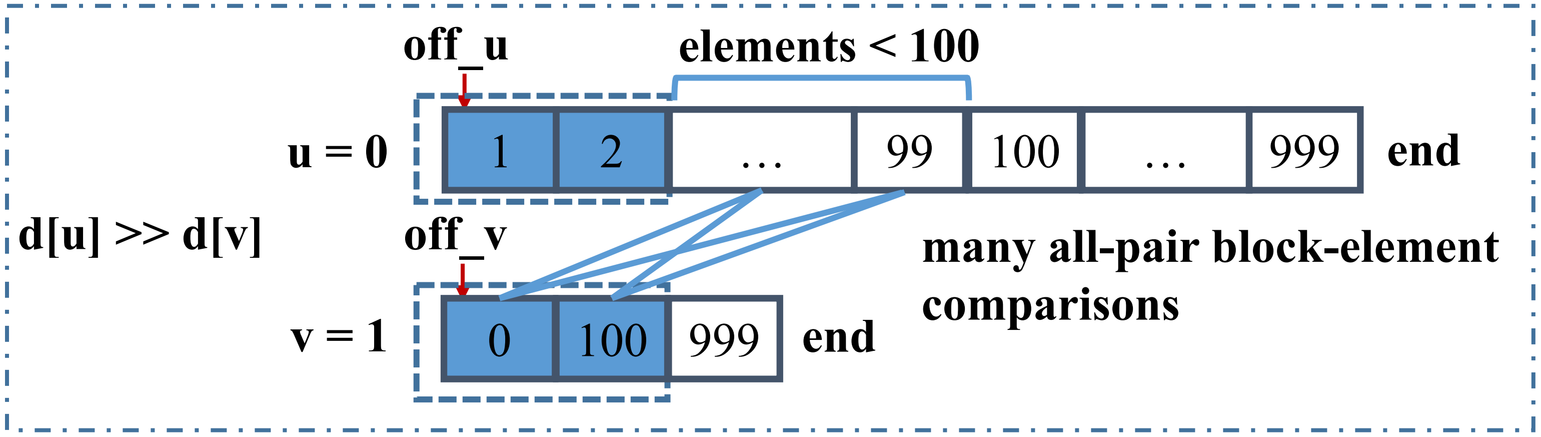
- Vectorized block-wise merge over two sorted arrays
  - do a while loop while the end is not reached
    - load two blocks of elements and shuffle the positions to conduct all-pair comparisons and count the matches
    - increment the offset of the array with smaller last block element



(a) initial state (b) block-wise merge (c) load, shuffle, and compare-equal

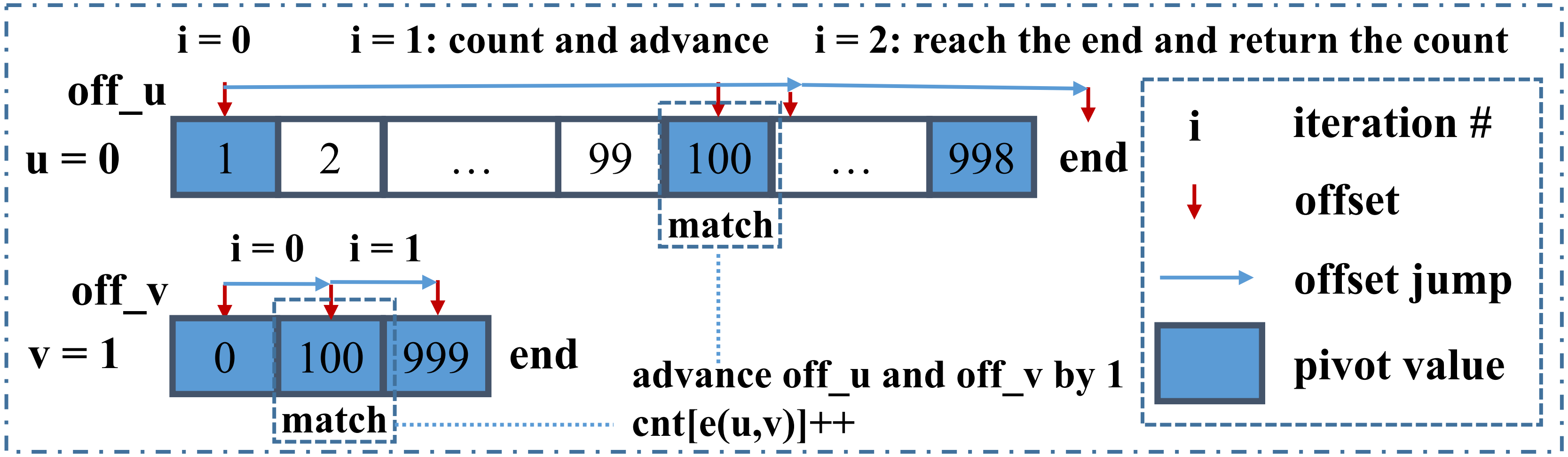
Limitation of Vectorized Block-Wise Merge

- Fail to handle the cardinality (degree) skew



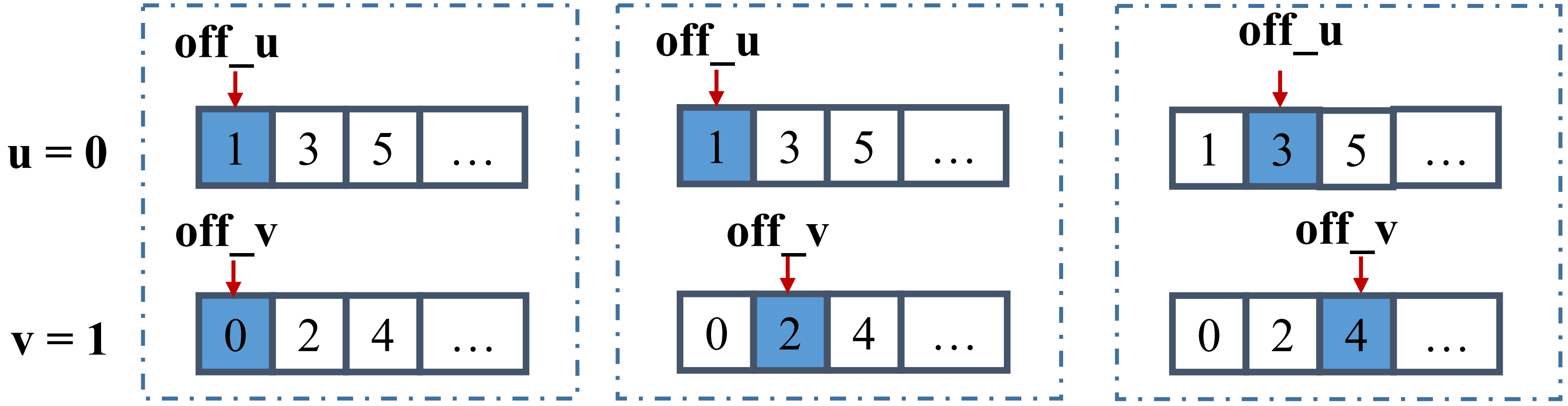
Pivot Skip Merge

- Do a while loop until we reach the end of one array
  - fix an element in one array as the **pivot**; in the other array, find the first element **not less than** the pivot value via a **galloping search**
  - do the same in the other array
  - count the matches



Limitation of Pivot Skip Merge

- Do not behave well for cases of similar degree adjacent vertices
  - Pivot Skip Merge (PS) may only advance a single step each time

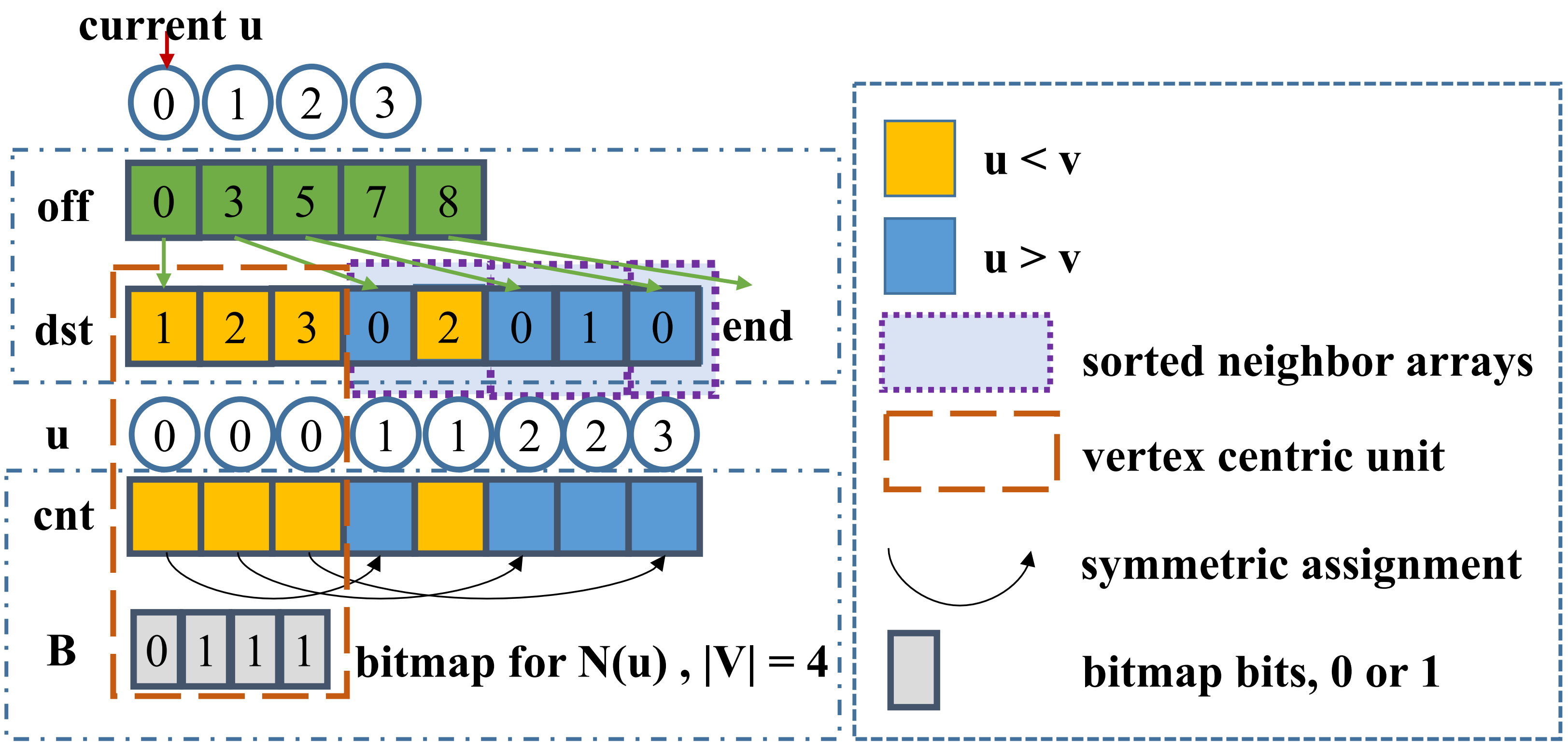


(a) initial state (b) after the 1st iteration (c) after the 2nd iteration

- Thus, we combine vectorized block-wise merge (VB) and pivot-skip merge (PS) by setting a degree-skew ratio to choose PS only when the degree-skew ratio is high (e.g. > 50)

BMP's Vertex-Centric Unit Computation

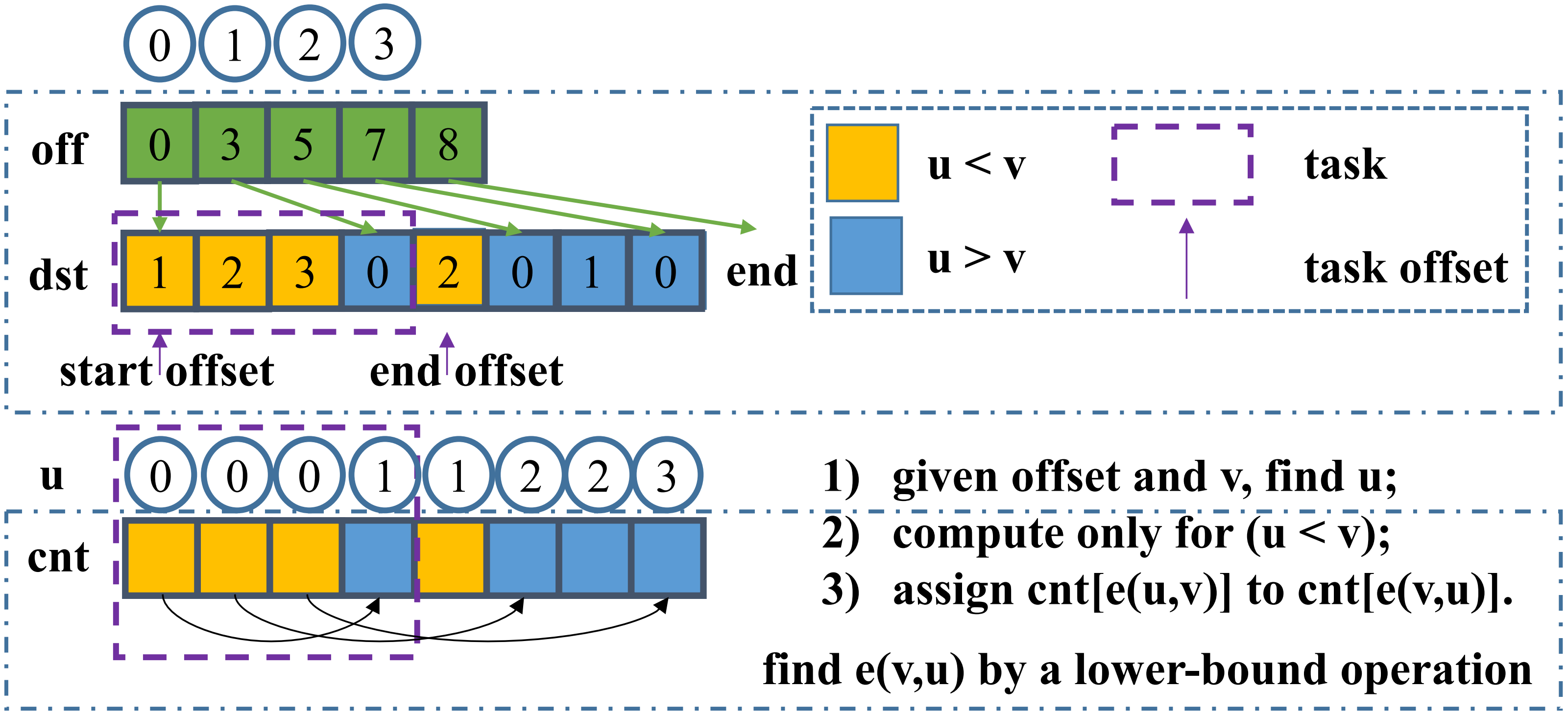
- For a vertex u, we construct a bitmap B(u)
- Then we loop over each v in N(u) satisfying the constraint **d(v) < d(u)**
  - we loop over N(v) and lookup B(u) to see if we can find a match to increment the common neighbor count
  - and then we assign the count symmetrically **from e(u,v) to e(v,u)**



Outline

Parallelization on CPUs

- Group a fixed number of edges as tasks, denoted by an offset pair



1) given offset and v, find u;
2) compute only for (u < v);
3) assign cnt[e(u,v)] to cnt[e(v,u)].

find e(v,u) by a lower-bound operation

Optimization of Finding Source Vertex

- Amortize the cost of finding a source vertex for each edge
    - use a thread local variable **u** to record current source vertex
    - check the current edge offset and compare it with **off[u]**, if it exceeds the range, we do a lower bound operation to find a **new u**

Extension for BMP

- Replace the **ComputeCntMPS** with **ComputeCntBMP** logic
  - Use a thread local bitmap for the index
  - Use thread local variable **pu** to record last indexed or constructed bitmap **B(pu)** and clear the bitmap when we process another vertex

Parallelization on GPUs

- Utilize the unified memory feature
- Design a co-processing skeleton to leave some symmetric assignment relevant workload to the CPU
  - reverse edge offset computation (**cnt[e(v,u)] = e(u,v)** where u < v)
  - symmetric assignment of **cnt[e(v,u)]** where u < v

```
$LaunchCUDAKernels(), AssignOffsetsOnCPU()$
$SynchronizeDevice()$
foreach $(u, v) \in E$ in parallel do
    if $u > v$ then $cnt[e(u, v)] \leftarrow cnt[cnt[e(u, v)]]$
Procedure $AssignOffsetsOnCPU()$
    foreach $(u, v) \in E$ in parallel do
        if $u < v$ then $cnt[e(v, u)] \leftarrow e(u, v)$
```

symmetric assignment

reverse edge offset computation

CUDA Skeleton Design

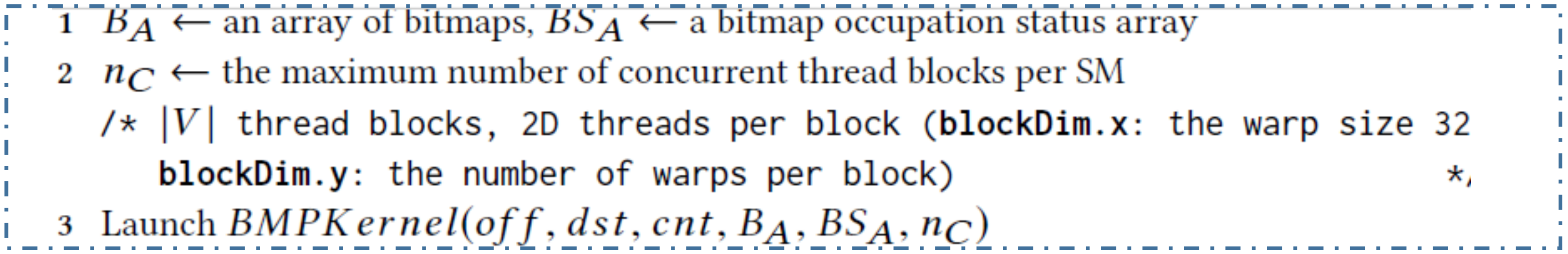
- Memory Allocation
  - Unified memory: CSR and result count arrays
  - Direct allocation on the device: a pool of bitmaps for BMP
- Thread Block Mapping
  - Each vertex related common neighbor counting tasks are mapped to a CUDA thread block in a vertex-centric manner, since GPU has an efficient hardware queue to schedule millions of thread blocks

MPS CUDA Kernels

```
/* $|V|$ thread blocks, 2D threads per block (blockDim.x: the warp size 32,
   blockDim.y: the number of warps per block)                              */
Launch $MKernel(off, dst, cnt, t)$        Each warp processes an edge
/* $|V|$ threads blocks, 1D threads per block                               */
Launch $PSKernel(off, dst, cnt, t)$       Each thread processes an edge
```

- MPS: launch two CUDA kernels (VB and PS) for the cardinality (degree) skewed and non-skewed cases respectively
- **MKernel**: block-wise merge utilizing shared memory
- **PSKernel:** pivot-skip merge kernel handling the cardinality skew

BMP CUDA Kernel



```
$B_A \leftarrow$ an array of bitmaps, $BS_A \leftarrow$ a bitmap occupation status array
$n_C \leftarrow$ the maximum number of concurrent thread blocks per SM
/* $|V|$ thread blocks, 2D threads per block (blockDim.x: the warp size 32
   blockDim.y: the number of warps per block)                              */
Launch $BMPKernel(off, dst, cnt, B_A, BS_A, n_C)$
```

- A bitmap is acquired from a pool
- Each **thread block** constructs a bitmap index B(u)
- Each **warp** processes an edge
- A bitmap is released to the pool

Summary of Optimization Techniques

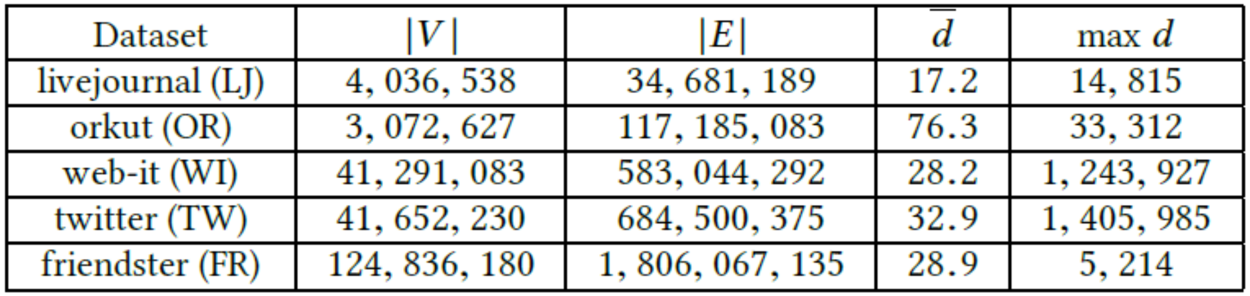
- Vectorization (discussed)
- Bitmap Range Filtering, utilizing sparsity of matches
- MCDRAM (high bandwidth memory) usage on the KNL
- Co-processing (discussed)
- Multi-pass processing to preserve memory access locality
- Block size tuning on the GPU

Outline

Experimental Setup

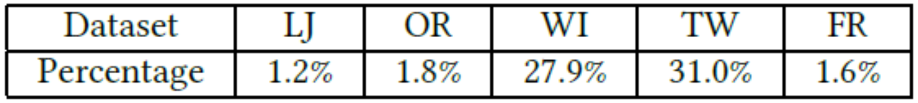
- Datasets

Table 1: Real-world Graph Statistics



Dataset	$\|V\|$	$\|E\|$	$\bar{d}$	max d
livejournal (LJ)	4, 036, 538	34, 681, 189	17.2	14, 815
orkut (OR)	3, 072, 627	117, 185, 083	76.3	33, 312
web-it (WI)	41, 291, 083	583, 044, 292	28.2	1, 243, 927
twitter (TW)	41, 652, 230	684, 500, 375	32.9	1, 405, 985
friendster (FR)	124, 836, 180	1, 806, 067, 135	28.9	5, 214

Table 2: Percentage of the highly skewed set intersections.



Dataset	LJ	OR	WI	TW	FR
Percentage	1.2%	1.8%	27.9%	31.0%	1.6%

- Platform
  - a CPU server with CPUs and Nvidia **TITAN XP** GPUs
    - The CPU server has **two 14-core** 2.4GHz Intel Xeon E5-2680 CPUs, with **35MB L3 Cache**
    - The Nvidia GPU on the CPU server has **30 SMs**, each of which **supports at most 2048 threads**
  - a KNL server with a KNL processor**, 64-core 1.3GHz** Intel Xeon Phi 7210 Processor, configured in the **quadrant mode** with **16GB MCDRAM**

Evaluation on the CPU and KNL

- Five Techniques
  1. the **d**egree-**s**kew **h**andling (**DSH**) for MPS and BMP
  2. the **v**ectorization or instruction-level parallelization (**V**) for MPS
  3. the task-level **p**arallelization (**P**) for MPS and BMP
  4. the bitmap **r**ange **f**iltering (**RF**) for BMP
  5. We further evaluate the **h**igh **b**and**w**idth memory MCDRAM usage (**HBW**) on the KNL

Summary of Results on the CPU and KNL

- **DSH** achieve **7x** (**MPS**) and **30x** (**BMP**) speedups over the baseline
- **MPS**
  - **V** and **P** bring **79x-84x** (CPU) **and 183x-186x** (KNL) speedups
  - **HBW** brings **1.6x-1.8x** speedups over the parallel vectorized MPS
- **BMP**
  - **P and RF** achieves **25x-28x** (CPU) and **47x-83x** (KNL)
  - **HBW** brings only **10-20%** improvements for the parallel **BMP-RF**, which is **random access dominant**
- CPU favors BMP whereas KNL favors MPS
  - BMP works better with CPU's **large caches** than KNL
  - KNL's **many cores** perform the computation of MPS faster than CPU

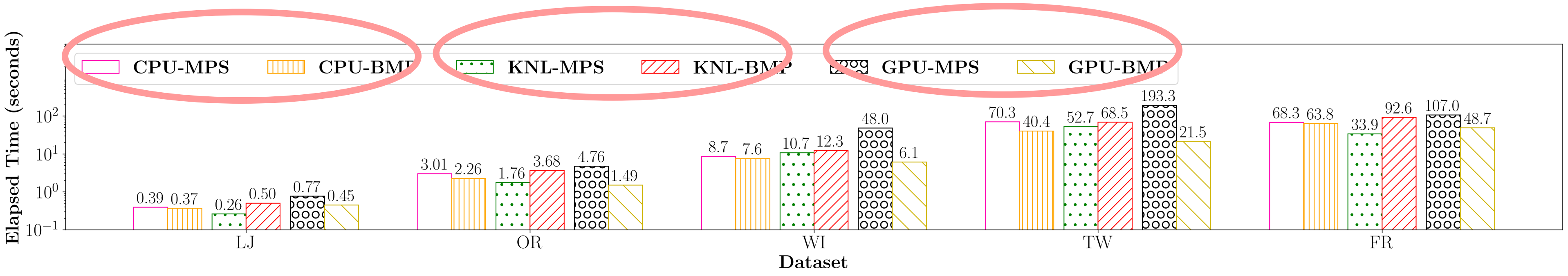
Evaluation on the GPU

- Four techniques:
  (1) the co-processing (**CP**) for MPS and BMP
  (2) the multi-pass processing (**MPP**) for MPS and BMP
  (3) the bitmap range filtering (**RF**) with the shared memory for BMP
  (4) The block size tuning (**BST**) for MPS and BMP

Summary of Results on the GPU

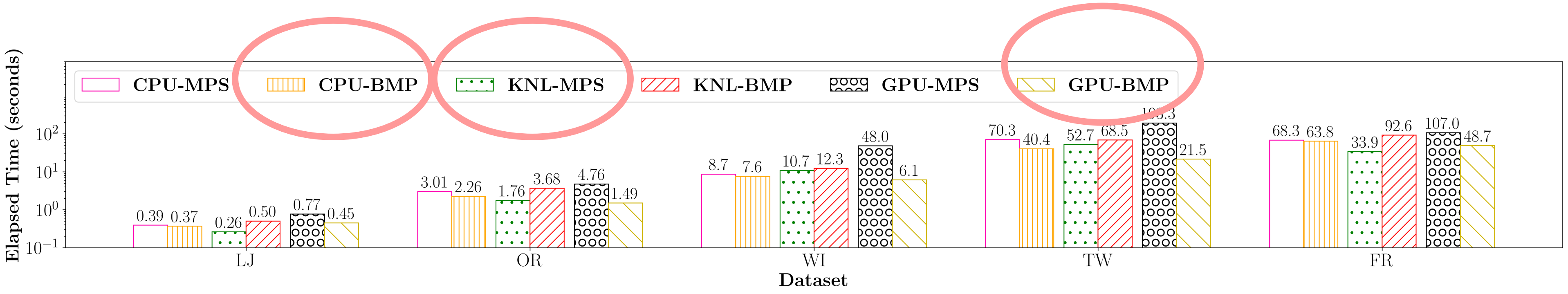
- **CP** reduces **80% of post-processing** time on the CPU
- **MPP** reduces the elapsed time **by orders of magnitude for both algorithms** on the large dataset FR (with over **14GB** CSR memory)
- **RF** reduces the time **by half**
- **BST** achieves **2.2x-3.8x**
  - fewer bitmap memory consumption and thus fewer page swaps
- Overall, the GPU favors BMP, because
  - BMP has less workload than MPS
  - exploits **warp-level parallelism**
  - **utilizes GPU resources more effectively** than MPS

Overall Comparison For Each Platform



- **CPU favors BMP**
  - less workload
  - cache-memory hierarchy
- **KNL favors MPS**
  - vectorization
  - high bandwidth memory MCDRAM
- **GPU favors BMP**
  - fewer global memory
  - warp-level parallelism
  - better occupation of GPU resources
  - PS kernel inefficient due to irregular memory gathering

Overall Comparison Among The Bests



- **Comparing CPU-BMP**, **KNL-MPS** and **GPU-BMP**
  - all of them complete within tens of seconds
  - the performance differs by a factor of **2.5x** at maximum
  - on the LJ and OR: all the algorithms complete within **2.3** seconds
  - on the WI and TW: GPU-BMP works best
  - on the FR: KNL-MPS works best
  - CPU-BMP is less competitive but has acceptable performance

Outline

Conclusion

- Two Algorithms
  - MPS: hybrid merge, scaling better to number of threads
  - BMP: amortized indexing cost, involving fewer workloads
- Parallelization and Optimization Techniques
- Algorithm Among Different Processors
- Comparison Among the Bests
- Overall Performance
  - Finish within tens of seconds on billion-edge graphs

End - Q & A

- **Source Codes / Scripts / Figures / More Results:** https://github.com/RapidsAtHKUST/AccTriCnt

- **This PPT:** https://www.dropbox.com/sh/i1r45o2ceraey8j/AAD8V3WwPElQjwJ0-QtaKAzYa?dl=0&preview=accTriCnt.pdf

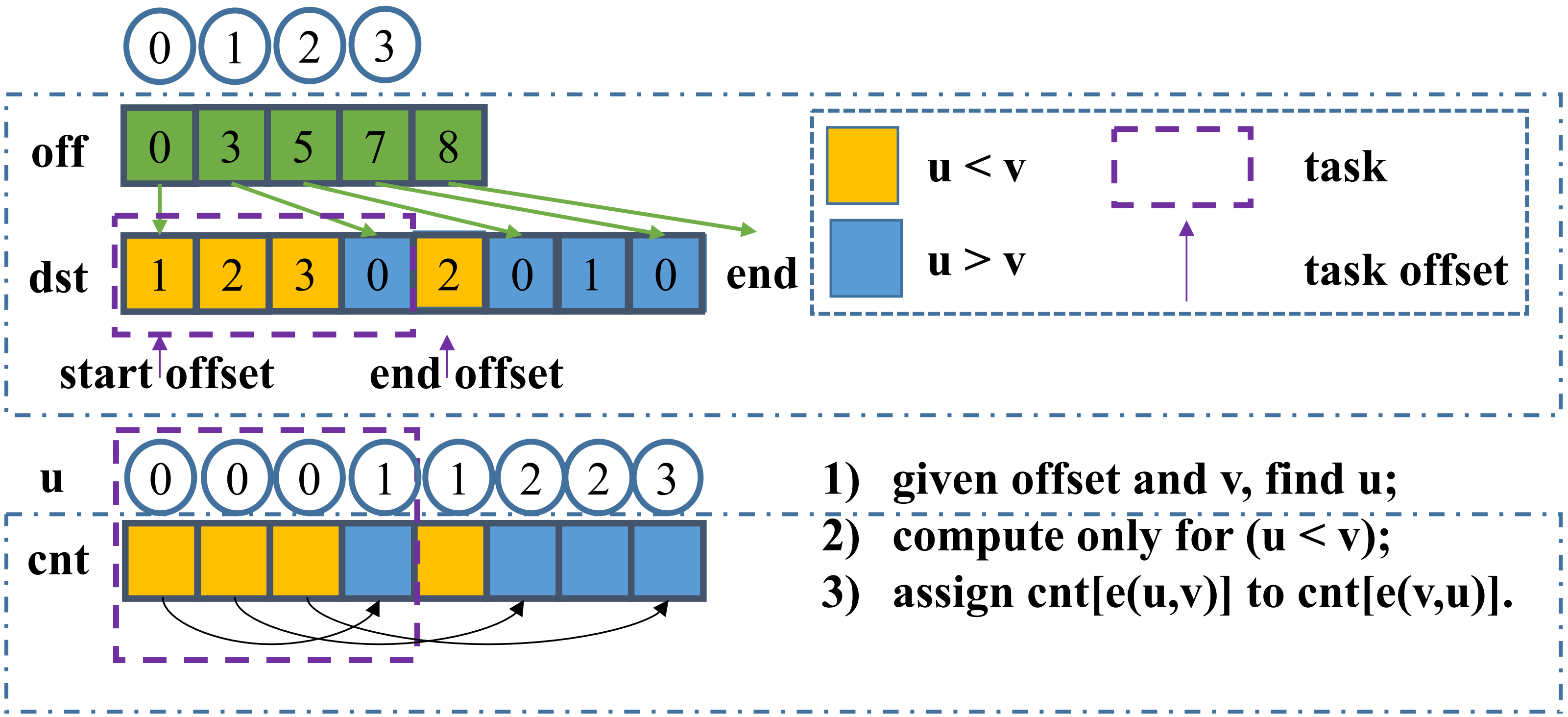
- **Our Research Group: RapidsAtHKUST** https://github.com/RapidsAtHKUST

Backup Slides: Discussion of MPS and BMP

- **MPS** utilizes the hybrid merge with a tunable switching threshold
  - PS: **O(c times min(d(u), d(v))** cost with an **average logarithm of skip size** as the parameter **c**
  - VB: exploits the utilization of **SIMD** and exploits better memory access pattern
- **BMP** has an **exact O(min(d(u), d(v))** cost
- MPS has a better memory access pattern than BMP whereas BMP involves fewer instructions

Backup Slides: OpenMP Skeleton of MPS

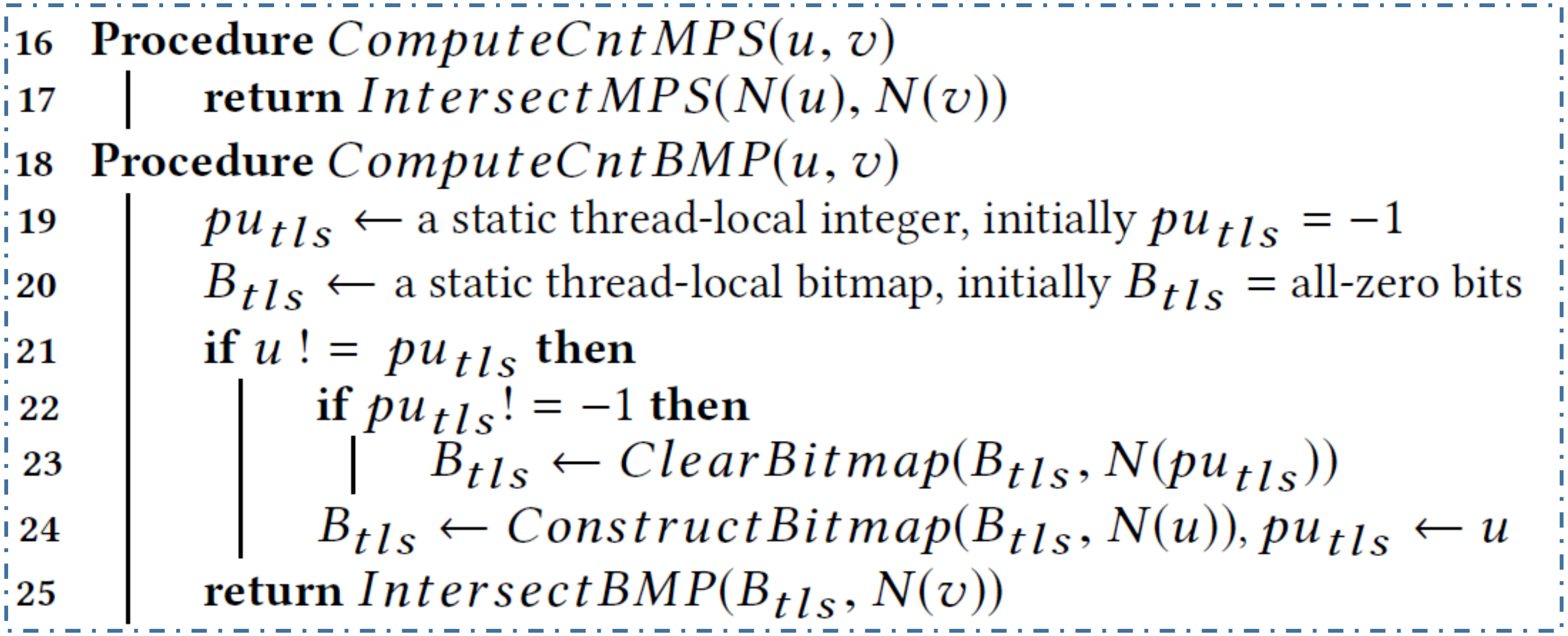
```
#pragma omp parallel for schedule(dynamic, the task size $|T|$)
foreach $e(u, v) \in [0, |E|)$ do
    $u \leftarrow FindSrc(off, e(u, v))$
    if $u < v$ then
        $cnt[e(u, v)] \leftarrow ComputeCntMPS(u, v)$
        $cnt[e(v, u)] \leftarrow cnt[e(u, v)]$
```



1) given offset and v, find u;
2) compute only for (u < v);
3) assign cnt[e(u,v)] to cnt[e(v,u)].

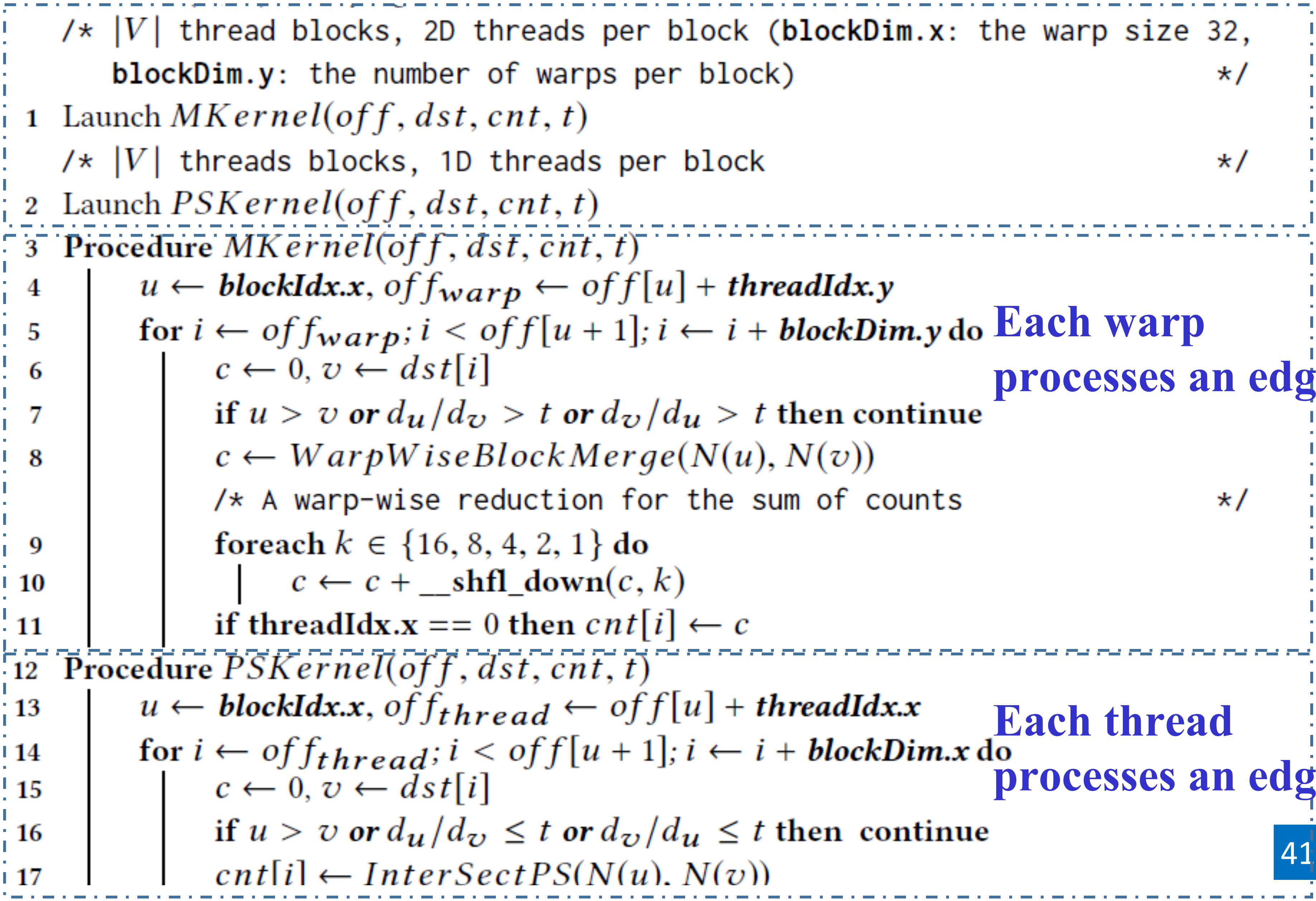
Backup Slides: Extension for BMP

- Replace the **ComputeCntMPS** with **ComputeCntBMP** logic
  - Use a thread local bitmap for the index
  - Use thread local variable **pu** to record last indexed or constructed bitmap **B(pu)** and clear the bitmap when we process another vertex



```
Procedure $ComputeCntMPS(u, v)$
    return $IntersectMPS(N(u), N(v))$
Procedure $ComputeCntBMP(u, v)$
    $pu_{tls} \leftarrow$ a static thread-local integer, initially $pu_{tls} = -1$
    $B_{tls} \leftarrow$ a static thread-local bitmap, initially $B_{tls}$ = all-zero bits
    if $u \mathrel{!}= pu_{tls}$ then
        if $pu_{tls} \mathrel{!}= -1$ then
            $B_{tls} \leftarrow ClearBitmap(B_{tls}, N(pu_{tls}))$
        $B_{tls} \leftarrow ConstructBitmap(B_{tls}, N(u)), pu_{tls} \leftarrow u$
    return $IntersectBMP(B_{tls}, N(v))$
```

Backup Slides: MPS CUDA Kernels



```
/* |V| thread blocks, 2D threads per block (blockDim.x: the warp size 32,
   blockDim.y: the number of warps per block)                          */
Launch $MKernel(off, dst, cnt, t)$
/* |V| threads blocks, 1D threads per block                            */
Launch $PSKernel(off, dst, cnt, t)$

Procedure $MKernel(off, dst, cnt, t)$
    $u \leftarrow$ blockIdx.x, $off_{warp} \leftarrow off[u] +$ threadIdx.y
    for $i \leftarrow off_{warp}; i < off[u+1]; i \leftarrow i +$ blockDim.y do
        $c \leftarrow 0, v \leftarrow dst[i]$
        if $u > v$ or $d_u/d_v > t$ or $d_v/d_u > t$ then continue
        $c \leftarrow WarpWiseBlockMerge(N(u), N(v))$
        /* A warp-wise reduction for the sum of counts                 */
        foreach $k \in \{16, 8, 4, 2, 1\}$ do
            $c \leftarrow c +$ __shfl_down$(c, k)$
        if threadIdx.x == 0 then $cnt[i] \leftarrow c$

Procedure $PSKernel(off, dst, cnt, t)$
    $u \leftarrow$ blockIdx.x, $off_{thread} \leftarrow off[u] +$ threadIdx.x
    for $i \leftarrow off_{thread}; i < off[u+1]; i \leftarrow i +$ blockDim.x do
        $c \leftarrow 0, v \leftarrow dst[i]$
        if $u > v$ or $d_u/d_v \leq t$ or $d_v/d_u \leq t$ then continue
        $cnt[i] \leftarrow InterSectPS(N(u), N(v))$
```

Each warp processes an edge

Each thread processes an edge

Backup Slides: BMP CUDA Kernel

```
$B_A \leftarrow$ an array of bitmaps, $BS_A \leftarrow$ a bitmap occupation status array
$n_C \leftarrow$ the maximum number of concurrent thread blocks per SM
/* $|V|$ thread blocks, 2D threads per block (blockDim.x: the warp size 32
   blockDim.y: the number of warps per block)                          */
Launch $BMPKernel(off, dst, cnt, B_A, BS_A, n_C)$
Procedure $BMPKernel(off, dst, cnt, B_A, BS_A, n_C)$
    $u \leftarrow$ blockIdx.x, $tid \leftarrow$ threadIdx.x+ blockDim.x·threadIdx.y
    if $tid == 0$ then $B \leftarrow AcquireBitmap(B_A, BS_A, n_C)$
    __syncthreads()
    $AtomicConstrucBitmap(B, N(u))$
    __syncthreads()
    $off_{warp} \leftarrow off[u]$ + threadIdx.y
    for $i \leftarrow off_{warp}$; $i < off[u+1]$; $i \leftarrow i$ + blockDim.y do
        $c \leftarrow 0, v \leftarrow dst[i]$
        if $u > v$ then continue
        foreach $w \in N(v)$ in warp-wise parallel do
            if the $w$'s bit is a 1-bit in the bitmap $B$ then
                $c \leftarrow c + 1$
        foreach $k \in \{16, 8, 4, 2, 1\}$ do
            $c \leftarrow c +$ __shfl_down$(c, k)$
        if threadIdx.x == 0 then $cnt[i] \leftarrow c$
    __syncthreads()
    $ClearBitmap(B, N(u))$, $ReleaseBitmap(BS_A)$
```

Each thread block constructs a bitmap index B(u)

Each warp processes an edge

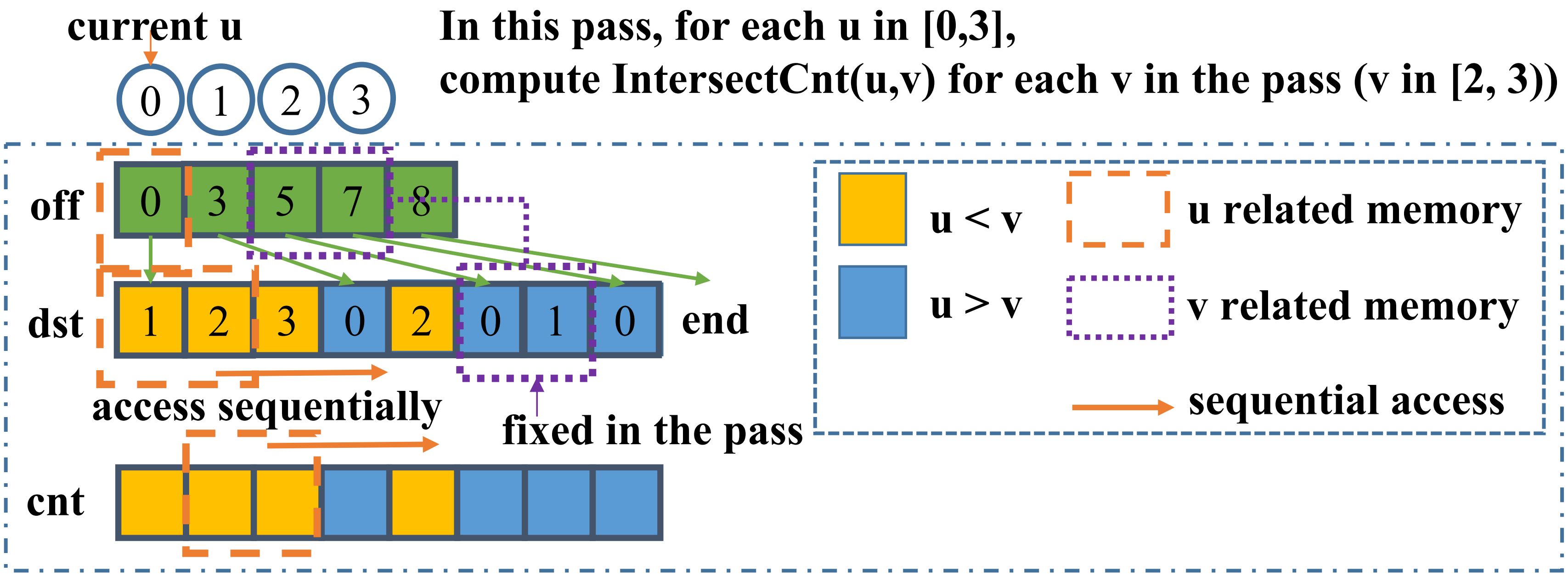
A bitmap is acquired from and released to a pool

Backup Slides: Handling Large Datasets

- Large datasets like FR occupies about 14GB memory for the CSR representation, which incur huge page swaps
- We introduce multi-pass processing to preserve the data locality (access of N(v)), avoid thrashing page swaps from the unified memory

Backup: Multi-Pass Processing Optimization

- Only compute part of the common neighbor counts of u, in our example, only for **v in the range [2,3)** for the current pass
  - Preserve the sequential memory access pattern of u
  - Limit the range of v in a single pass



Backup Slides: Optimization Techniques

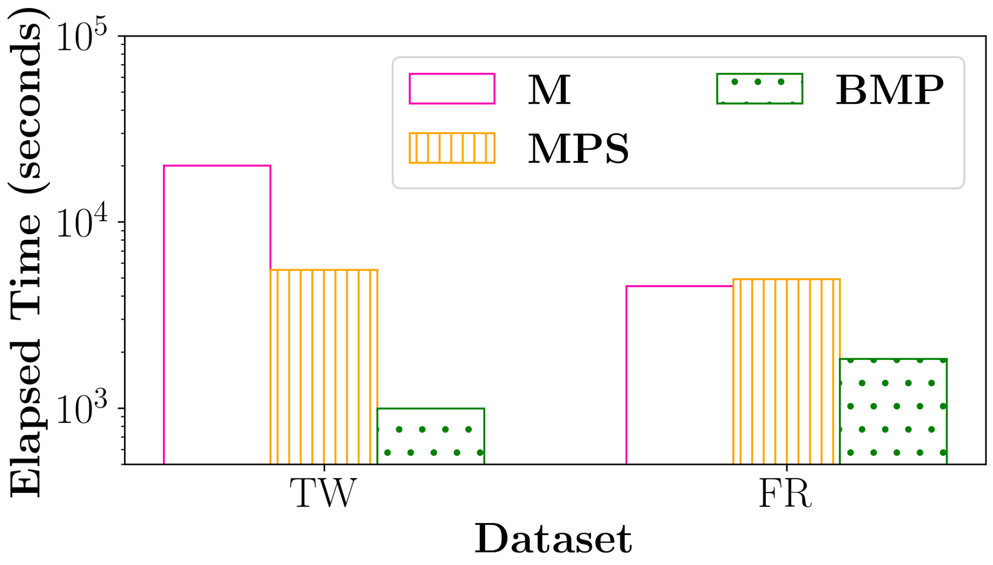
- Vectorization for MPS on the CPU and KNL (discussed)
  - AVX2
  - AVX512
- Bitmap Range Filtering
  - Utilize the sparsity of matches in real-world graphs
  - Use a small range bitmap (fit to L1 cache or shared memory) to filter underlying bitmap access
- MCDRAM (high bandwidth memory) usage on the KNL
  - Cache mode
  - Flat mode: allocation of CSR and bitmaps on the MCDRAM
- Co-processing to overlap the reversed edge offset assignment on the CPU and common neighbor counting on the GPU (discussed)
- Multi-pass processing on the GPU to improve the unified memory access locality (discussed)
- Block size tuning on the GPU (affect memory consumption for BMP and device SM occupancy for both MPS and BMP)

Backup Slides: GPU Configuration

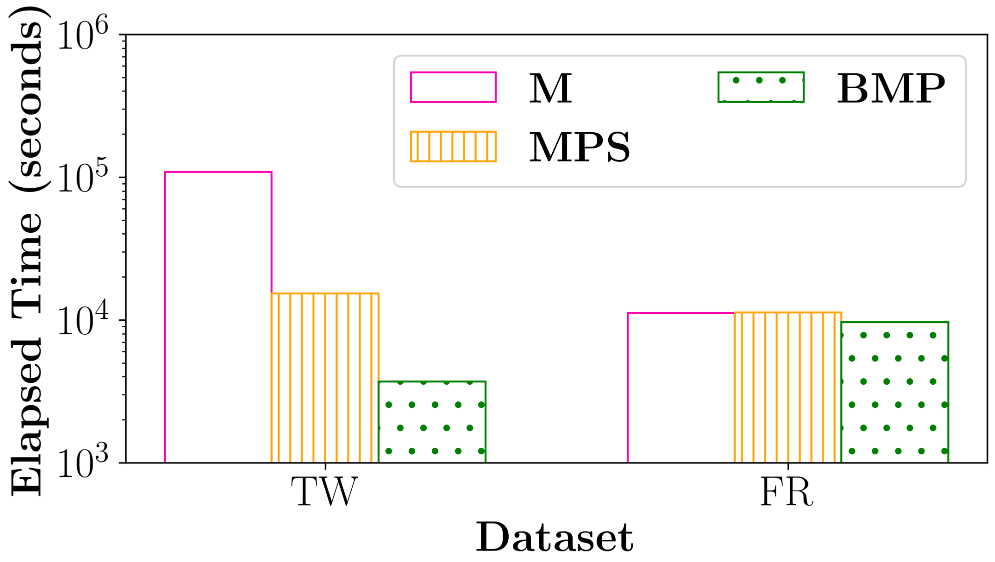
- By default, we use **4 warps per thread block**, resulting in **at most 16** (2048/128) **concurrent thread blocks per SM**. According to the Nvidia document, 16 is the maximum number of thread blocks simultaneously scheduled on a SM of the TITAN XP GPU. Our default setting targets a maximum of 100% occupancy of the GPU.

Backup Slides: Degree Skew Handling

CPU

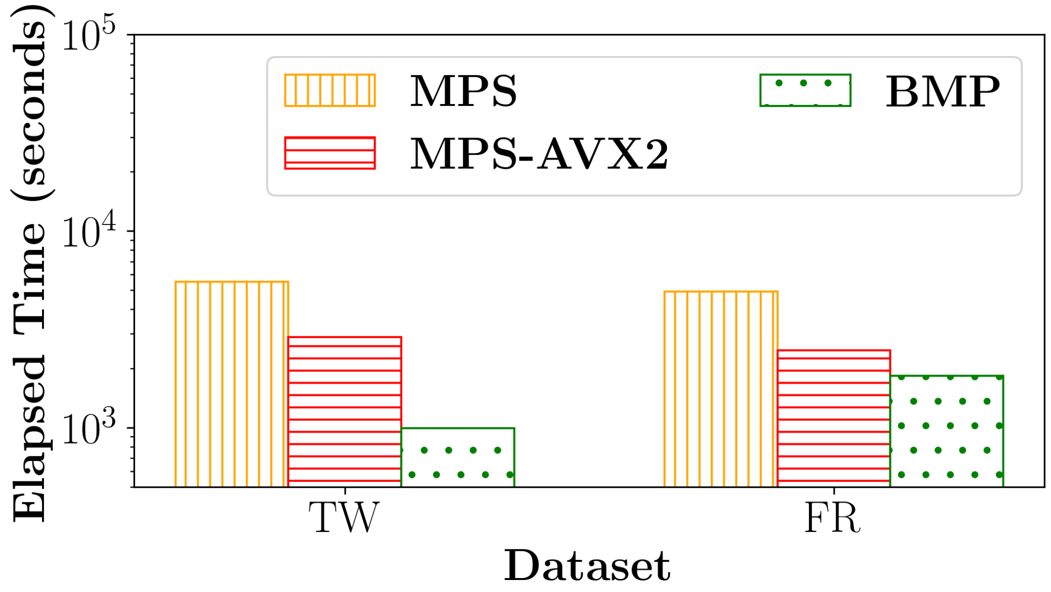


KNL

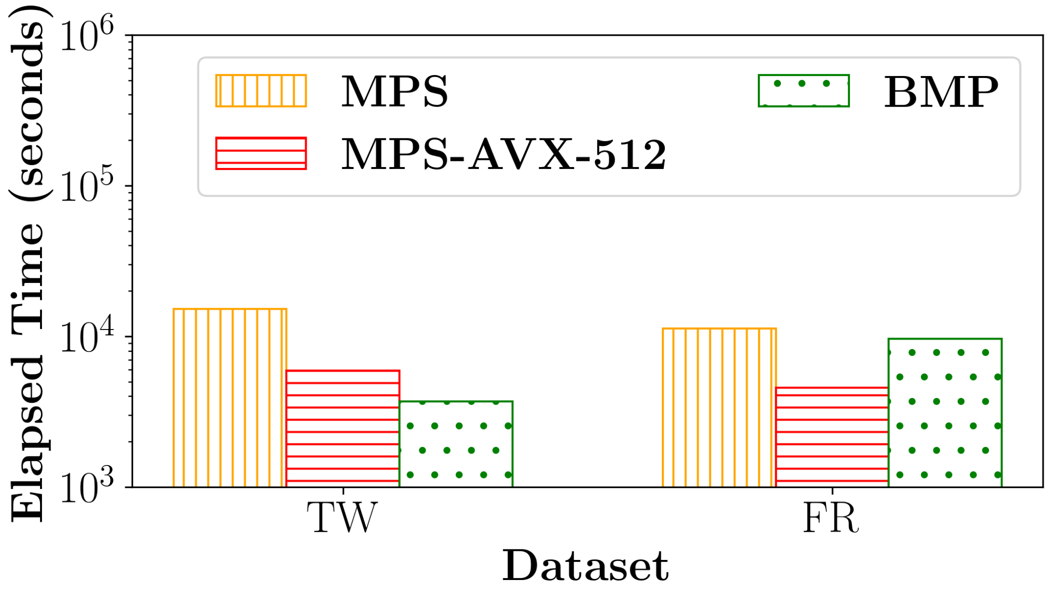


Backup Slides: Vectorization

CPU

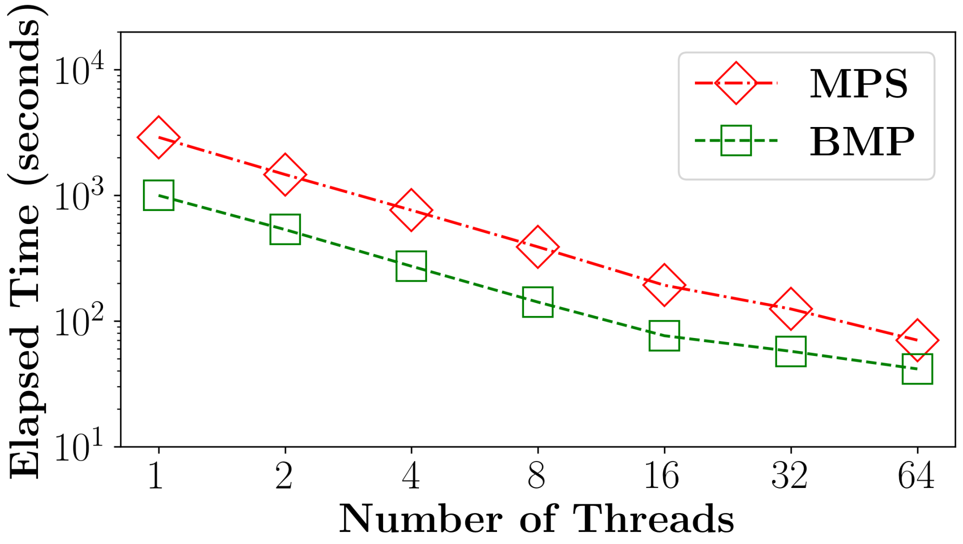


KNL

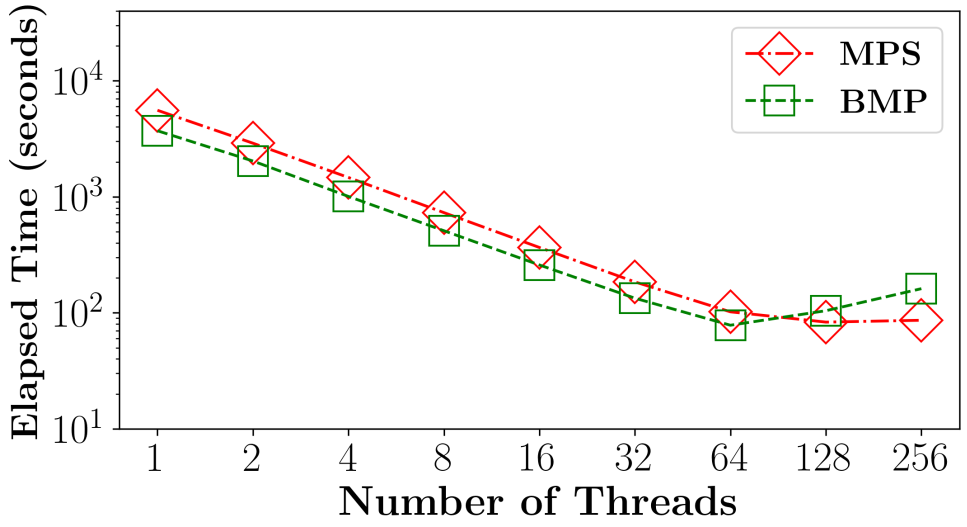


Backup Slides: Scalability (TW)

CPU, TW

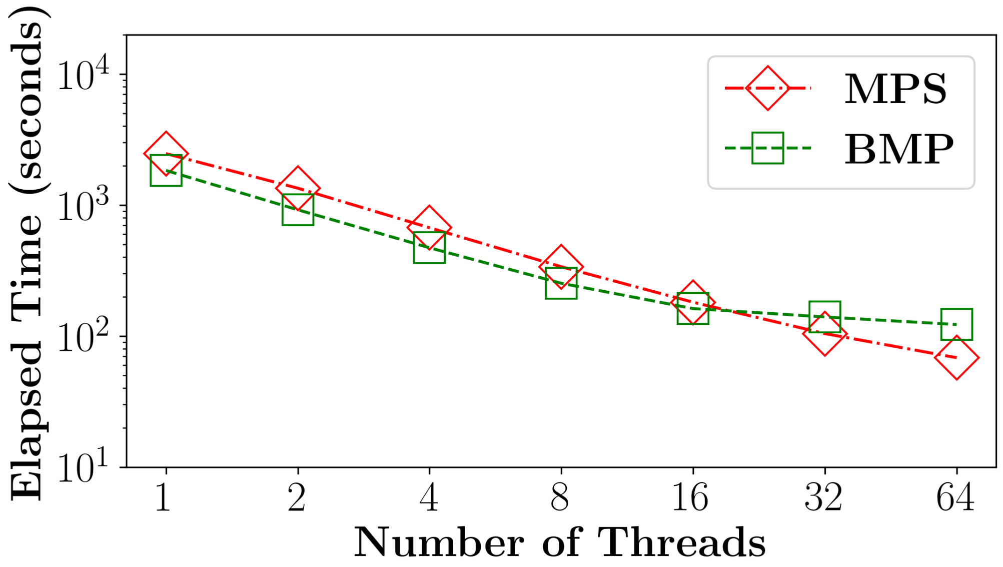


KNL, TW

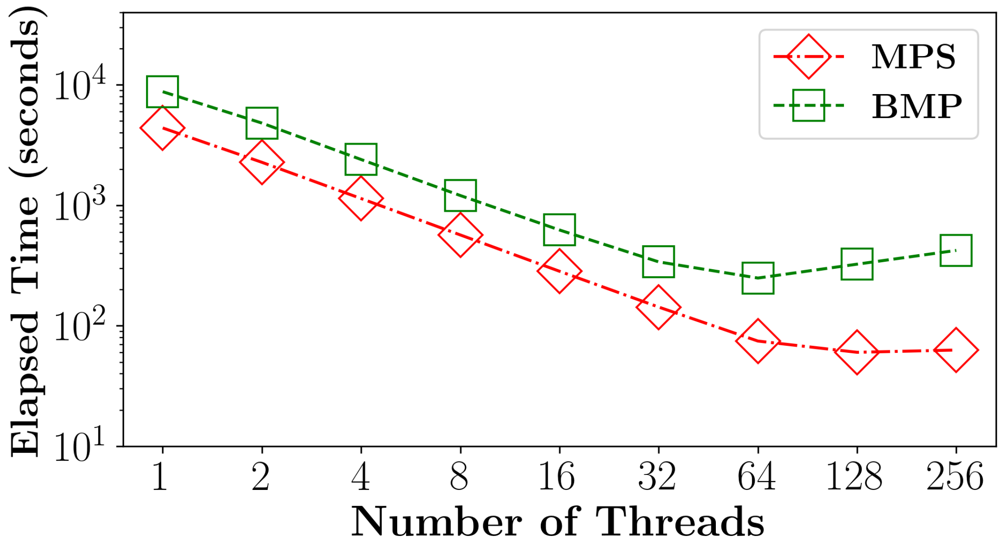


Backup Slides: Scalability (FR)

CPU, FR

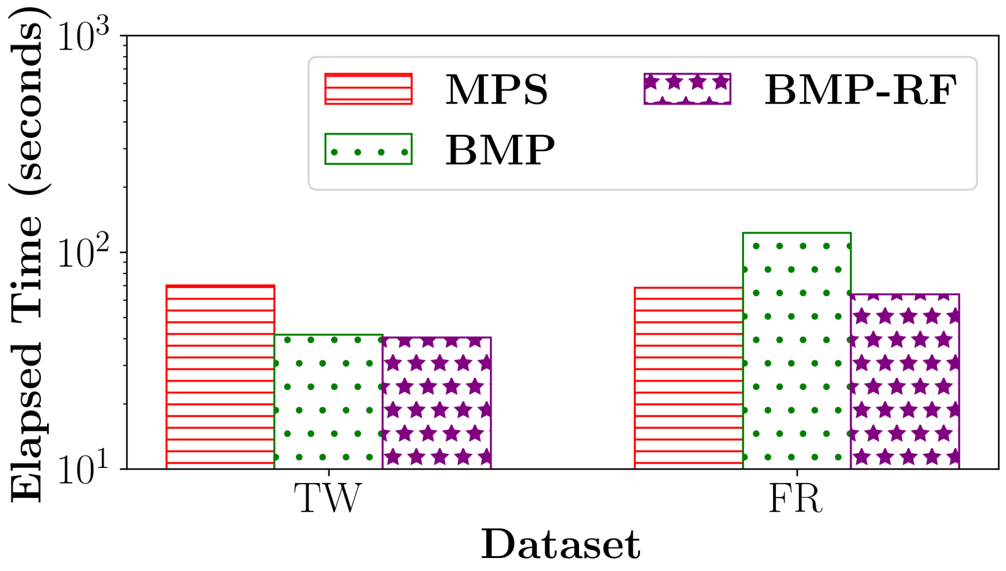


KNL, FR

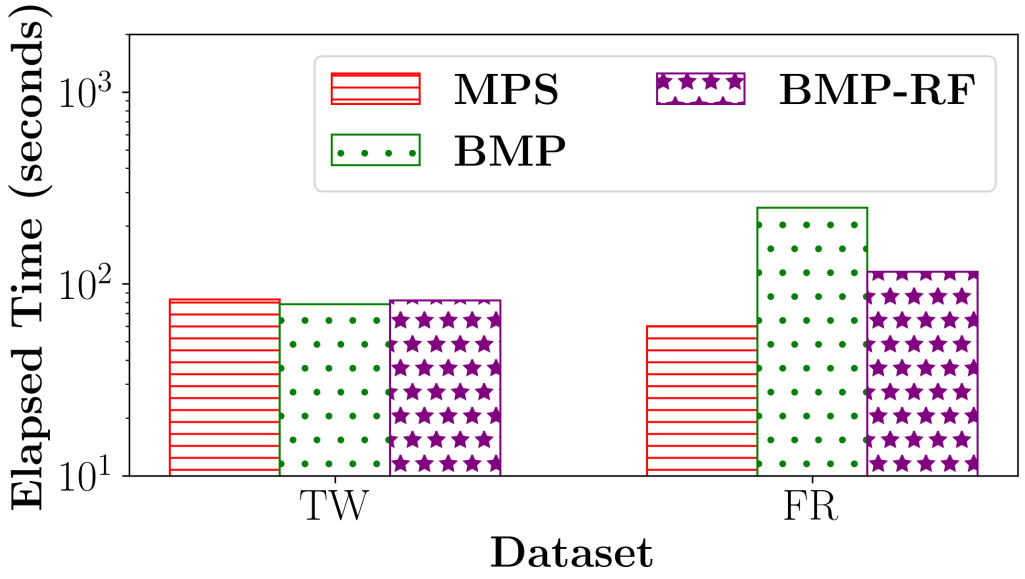


Backup Slides:　Range Filtering for BMP

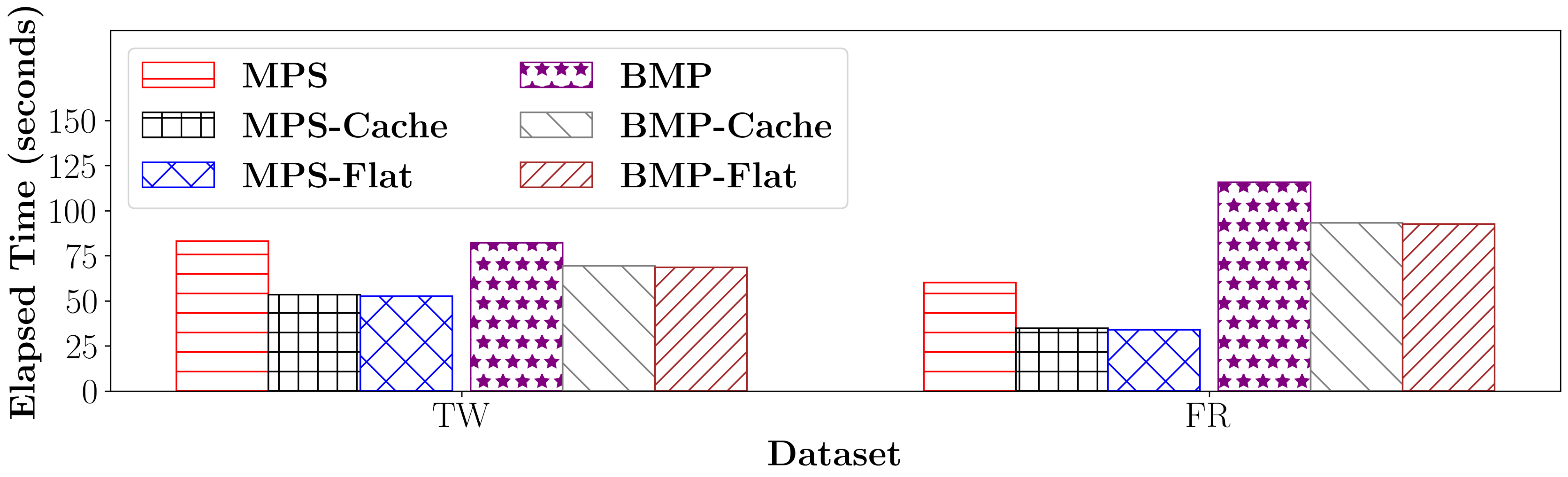
CPU



KNL

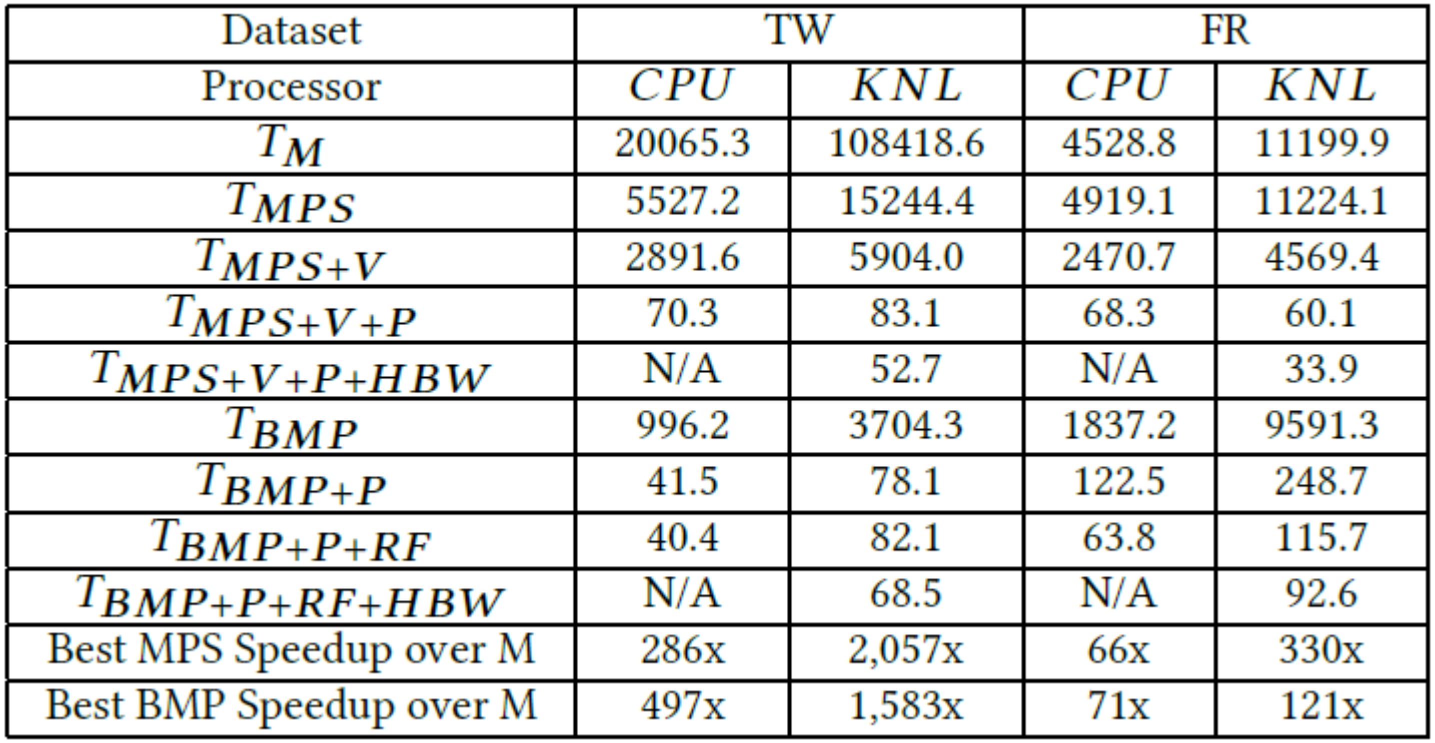


Backup Slides: MCDRAM (KNL)



Backup Slides: Summary (CPU & KNL)

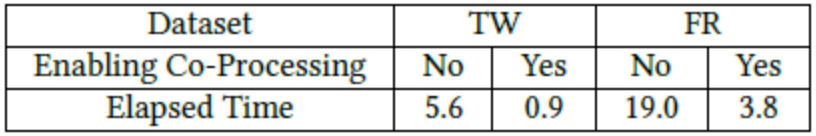
Table 4: Comparison with the baseline M (seconds).



Dataset	TW		FR	
Processor	CPU	KNL	CPU	KNL
T_M	20065.3	108418.6	4528.8	11199.9
T_{MPS}	5527.2	15244.4	4919.1	11224.1
T_{MPS+V}	2891.6	5904.0	2470.7	4569.4
$T_{MPS+V+P}$	70.3	83.1	68.3	60.1
$T_{MPS+V+P+HBW}$	N/A	52.7	N/A	33.9
T_{BMP}	996.2	3704.3	1837.2	9591.3
T_{BMP+P}	41.5	78.1	122.5	248.7
$T_{BMP+P+RF}$	40.4	82.1	63.8	115.7
$T_{BMP+P+RF+HBW}$	N/A	68.5	N/A	92.6
Best MPS Speedup over M	286x	2,057x	66x	330x
Best BMP Speedup over M	497x	1,583x	71x	121x

Backup Slides: Co-Processing

Table 5: Post-processing time on the CPU (seconds)

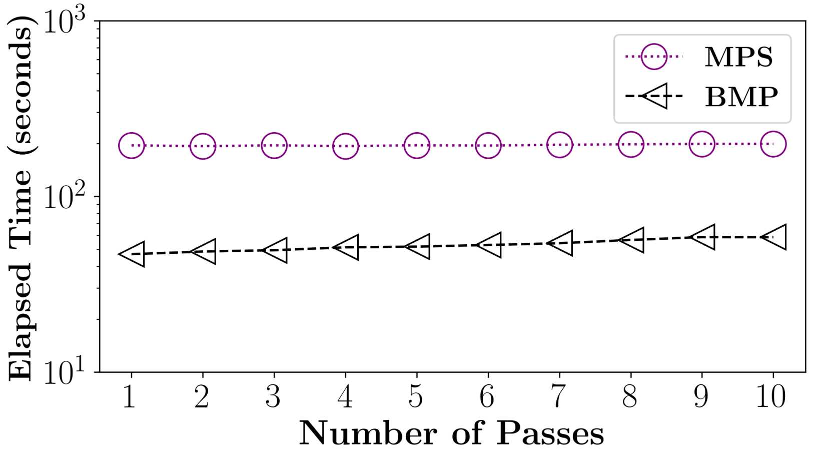


Dataset	TW		FR	
Enabling Co-Processing	No	Yes	No	Yes
Elapsed Time	5.6	0.9	19.0	3.8

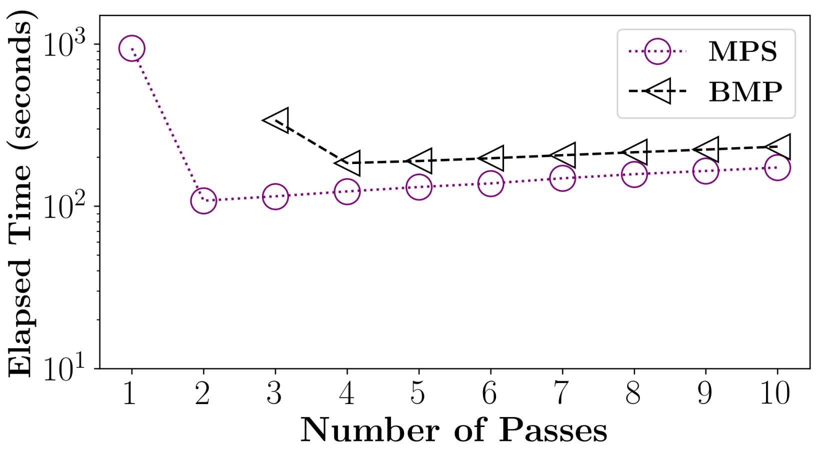
Backup Slides: Multi-Pass Processing

Table 6: Memory consumption of data structures and estimated number of passes

Dataset	TW		FR	
Algorithm	MPS	BMP	MPS	BMP
Mem_{cnt}	5.2GB		13.5GB	
Mem_{CSR}	5.3GB		13.9GB	
Mem_{B_A} (480 bitmaps)	0	2.3GB	0	7.0GB
Estimated number of passes	1	1	2	4



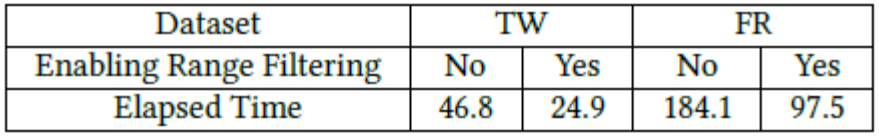
TW



FR

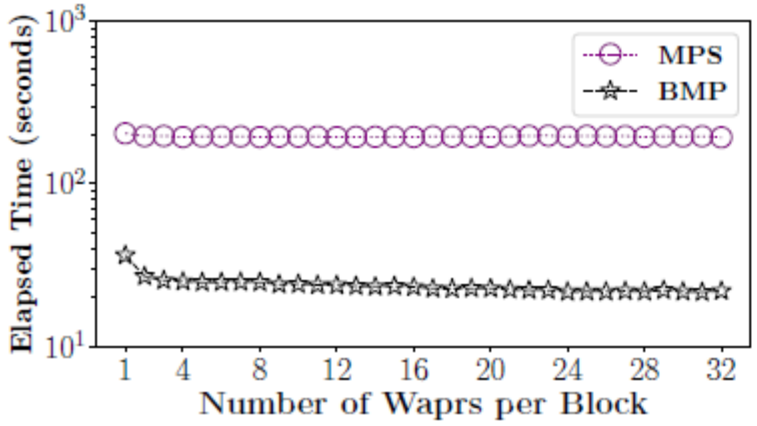
Backup Slides: Bitmap Range Filtering

Table 7: Elapsed time of BMP on the GPU (seconds)

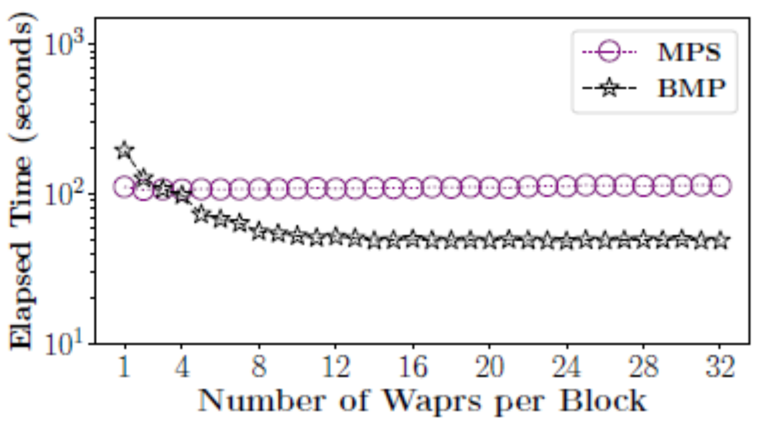


Dataset	TW		FR	
Enabling Range Filtering	No	Yes	No	Yes
Elapsed Time	46.8	24.9	184.1	97.5

Backup Slides: Effect of Block Size



(a) TW



(b) FR

Figure 9: Effect of block size

Backup Slides: Conclusion

- MPS exploits the **vectorization** technique, and **scales better to number of threads** than BMP; while BMP involves **fewer workloads** for each neighbor set intersection than that of MPS.
- The performance of the two algorithms is close on both CPU and KNL, but may differ up to an order of magnitude on the GPU. MPS works best on the KNL, because of the **128 VPUs and the high bandwidth memory MCDRAM**. CPU favors BMP, because its **L3 cache** reduces the memory access latency. GPU also favors BMP, since the **warp-level parallelism** helps utilize the resources.
- **BMP on the GPU** and **MPS on the KNL** work best respectively for degree-skewed and non-degree-skewed large graphs. The performance of both algorithms on the CPU is moderate, **at most 1.9x slower** than the best algorithms on both the KNL and the GPU. MPS on the GPU is always the slowest, followed by BMP on the KNL.
- Our optimized parallel algorithms complete the computation within **tens of seconds on billion-edge** graphs, enabling online graph analytics.