Overview

1. N-gram graphs overview
2. Serial Update Operation
3. GPU overview
4. Parallel Update Operation
5. Results

N-gram graphs

- General concept applicable in various fields
- An ’entity’ is broken into ‘pieces’ (called n-grams)
- n-grams are connected with edges (based on some logic)
- Capable of text representation
- Useful in many applications (e.g. topic extraction

Serial Update Operation

- Two n-gram graphs G_1 and G_2 given
- Goal : "enrich" G_1 with the information contained within G_2
- In other words, *update* G_1 using G_2
- Resulting graph : edges of both G_1 and G_2
- Edge weights dependent on parameter $l \in [0,1]$ (called learning factor)

Serial Update Operation - Algorithm

for every edge e on $G_1 \cup G_2$

$w = w_{e,1} + (w_{e,2} - w_{e,1}) * l$

if (G_1*.hasEdge(e))* /* $w_{e,1} > 0$ */

G_1*.updateEdge(e, w)*

else

G_1*.addEdge(e,w)*

Where,

- $w_{e,i}$ is the weight of edge *e* in graph G_i (*0* if it is not contained in the graph)
- l is the learning factor

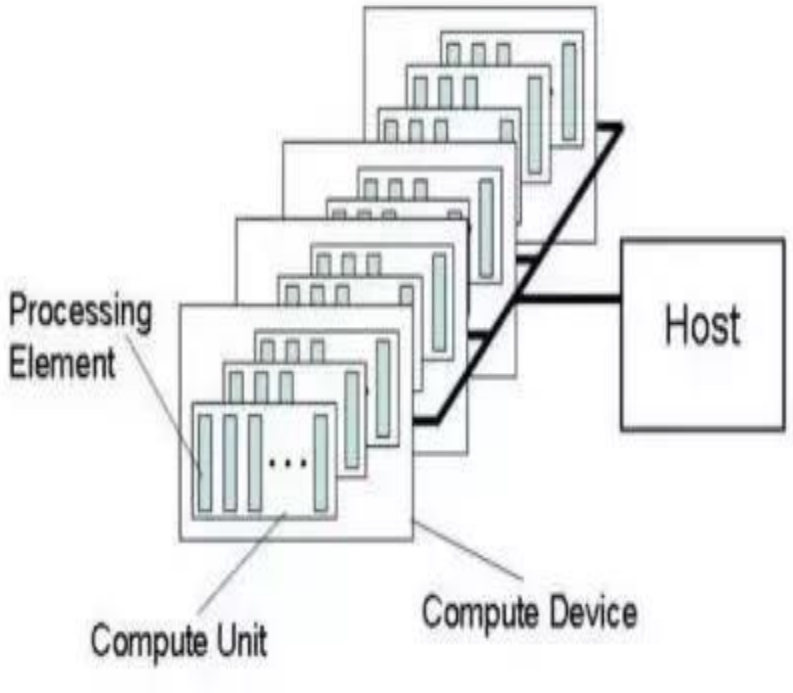
Update Operation – Execution time

- G_1 updated also with other graphs G_i, *i=3,4,…*
- G_1 can then represent a whole class of documents
- Big number of consecutive update operations → time consuming

#updates	100	200	300	400	500	600	700	800	900	1000
time(sec)	57.16	187.37	344.02	544.38	775.26	1050.9	1287.8	1591.3	1874.1	2182.9

GPU Overview

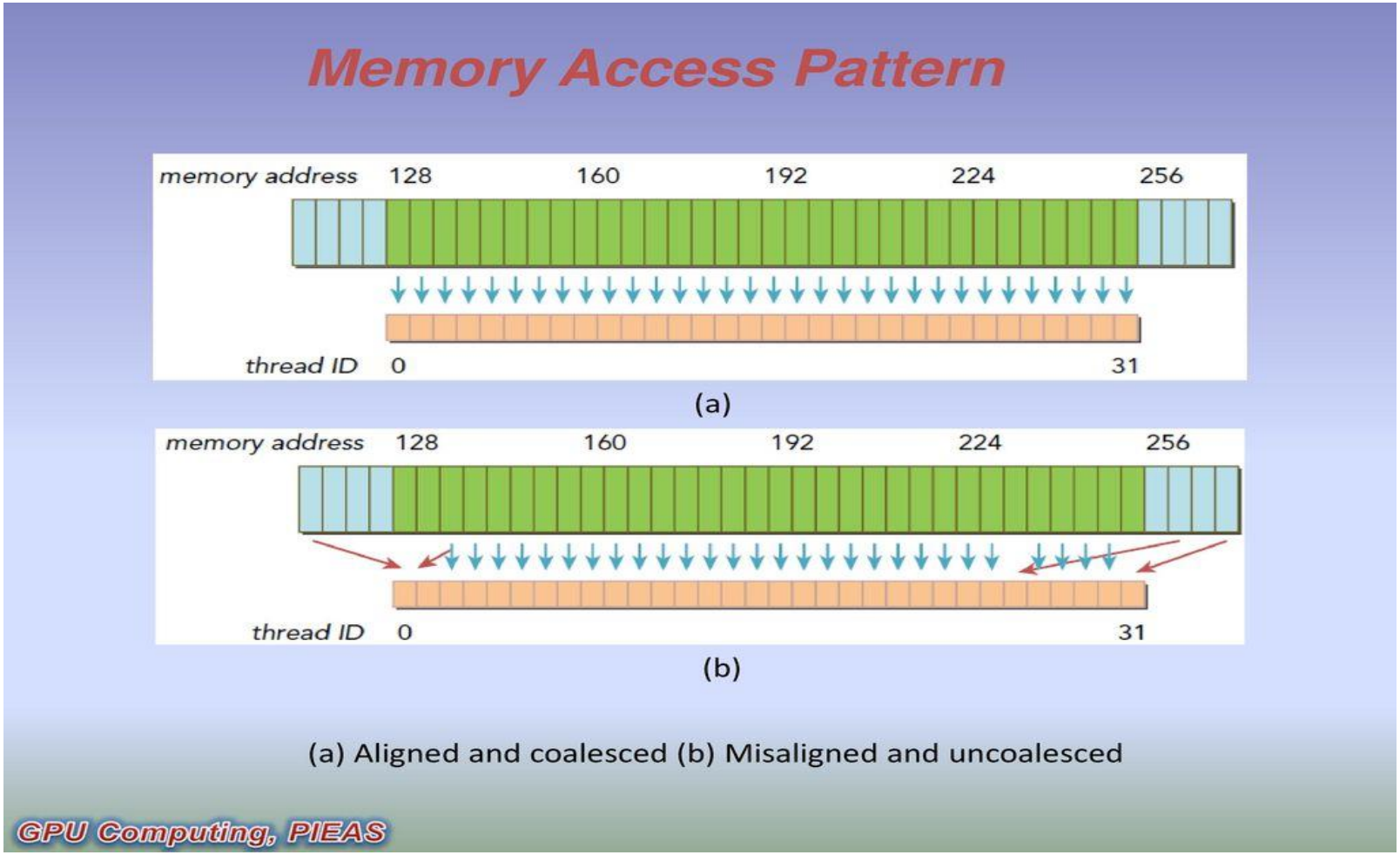
- A GPU consists of a number of Compute Units (CUs)
- Each CU consists of a number of Processing Elements (cores)
- The exact number of CUs and cores per CU is vendor-dependent
- Typical values are 8 CUs and 32 cores/CU for a total of 256 cores



Programming model

- Each core executes the same function called kernel
- Each instance of the kernel operates on different data
- Great for data-parallel applications (for example matrix multiplication)

Memory access coalescing



Parallel Update Operation (1/4)

- Update operation parallelizable
- Edge weight update : independent computation
- Individual computations assigned to threads
- Threads mapped to GPU cores

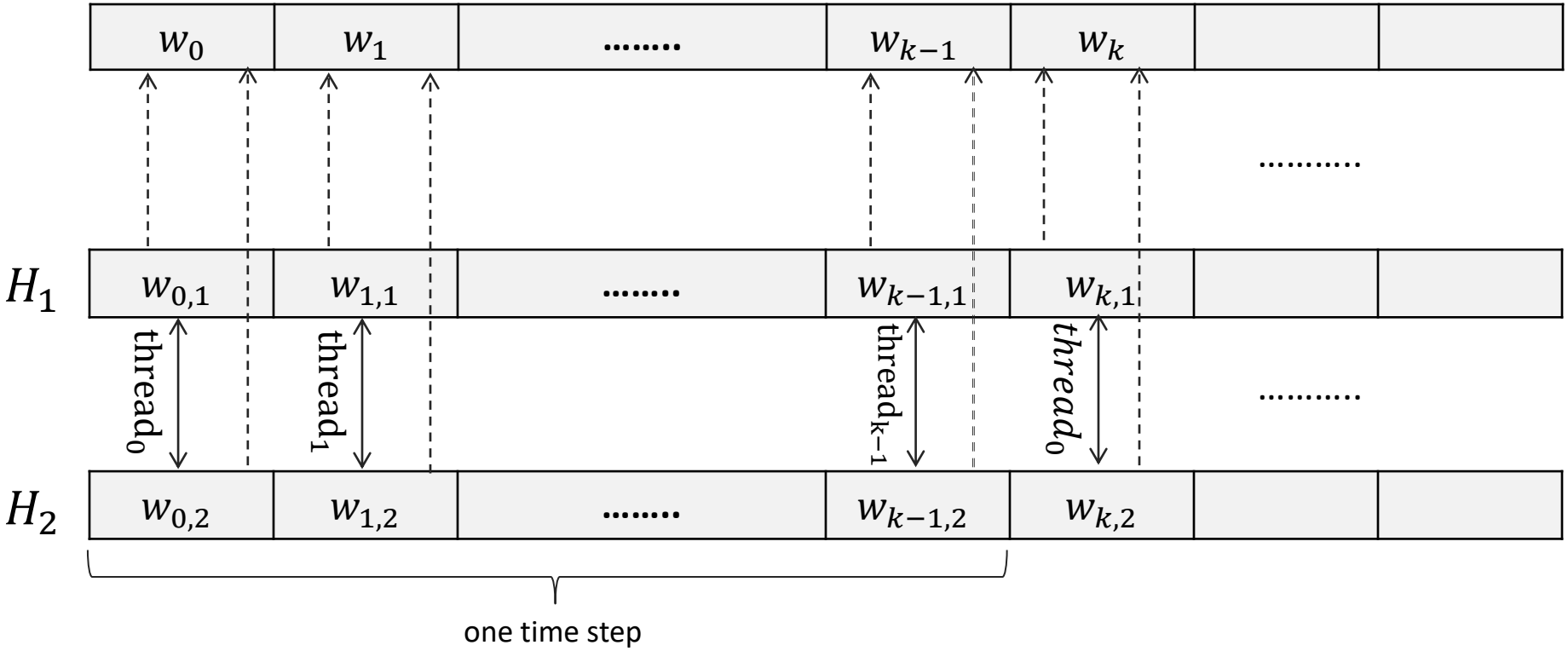
Parallel Update Operation (2/4)

- A hash table H_i for each graph G_i
- H_i : array of edge weights (floats)
- Hash function on edge labels $(hash(e) = h)$
- $H_i[h]$: edge's *e* weight on graph G_i
- (False) assumption : no edges e_1, e_2 such that $hash(e_1) = hash(e_2)$

Parallel Update Operation (3/4)

- Goal : update G_1 with G_2
- H_1, H_2 sent to GPU memory
- *k* processing elements at our disposal
- Weight computation of first *k* edges mapped to *k* available cores
- Computed in one time step
- Same for the next *k* edges…

Parallel Update Operation (4/4)



$w_{i,j}$ *is the weight of edge e with* $hash(e) = i$ *on graph* G_j

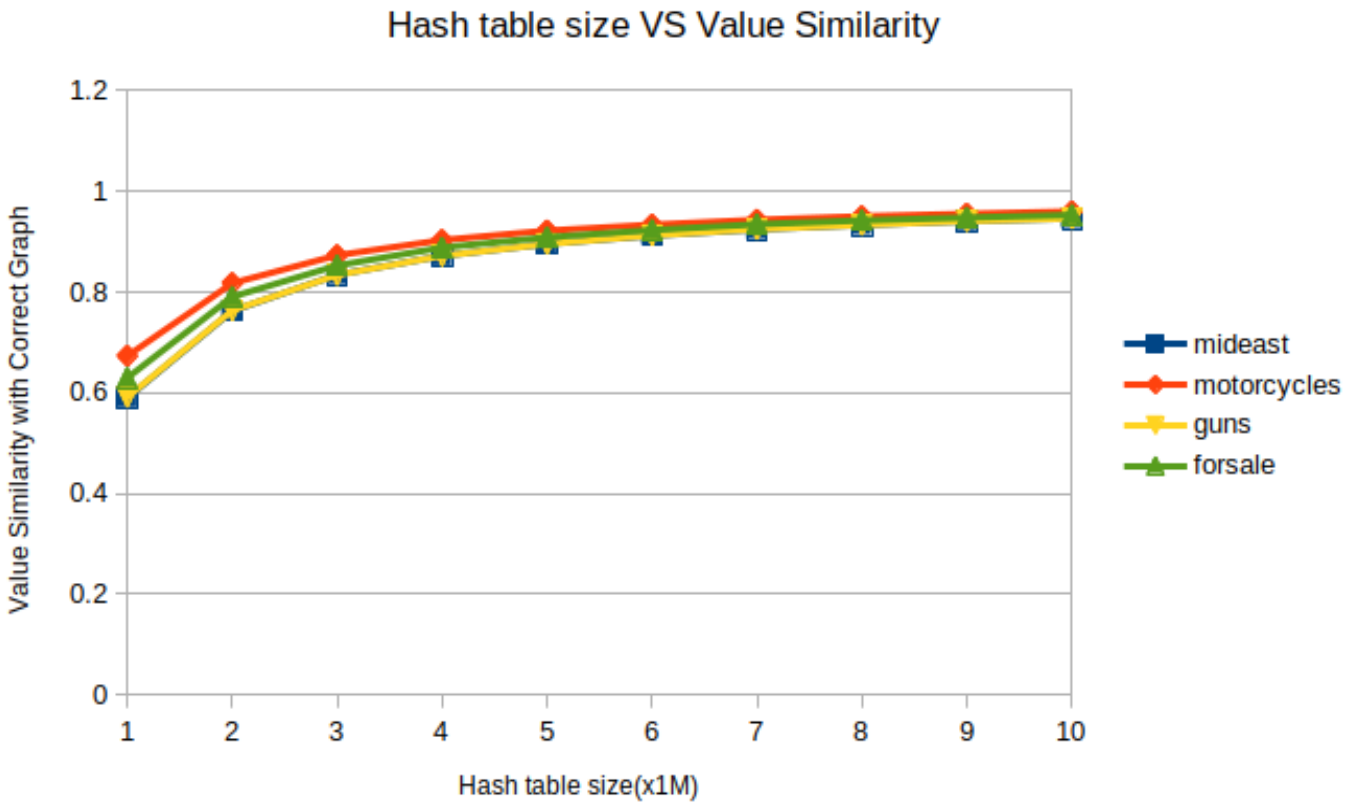
$w_i = w_{i,1} + (w_{i,2} - w_{i,1}) * l,$
is the weight of edge e with $hash(e) = i$ *on the updated graph*

Parallel vs Serial – Similarity

- Scenario : ~1k consecutive updates
- Two constructed graphs (parallel/serial) not exactly the same
- Because of hash collisions on edges labels
- Value Similarity (VS) operator for graph comparison
- $VS: G \times G \rightarrow [0,1]$, where *G* is the set of n-gram graphs
- Values close to 1 → bigger similarity
- Ideally *VS=1*

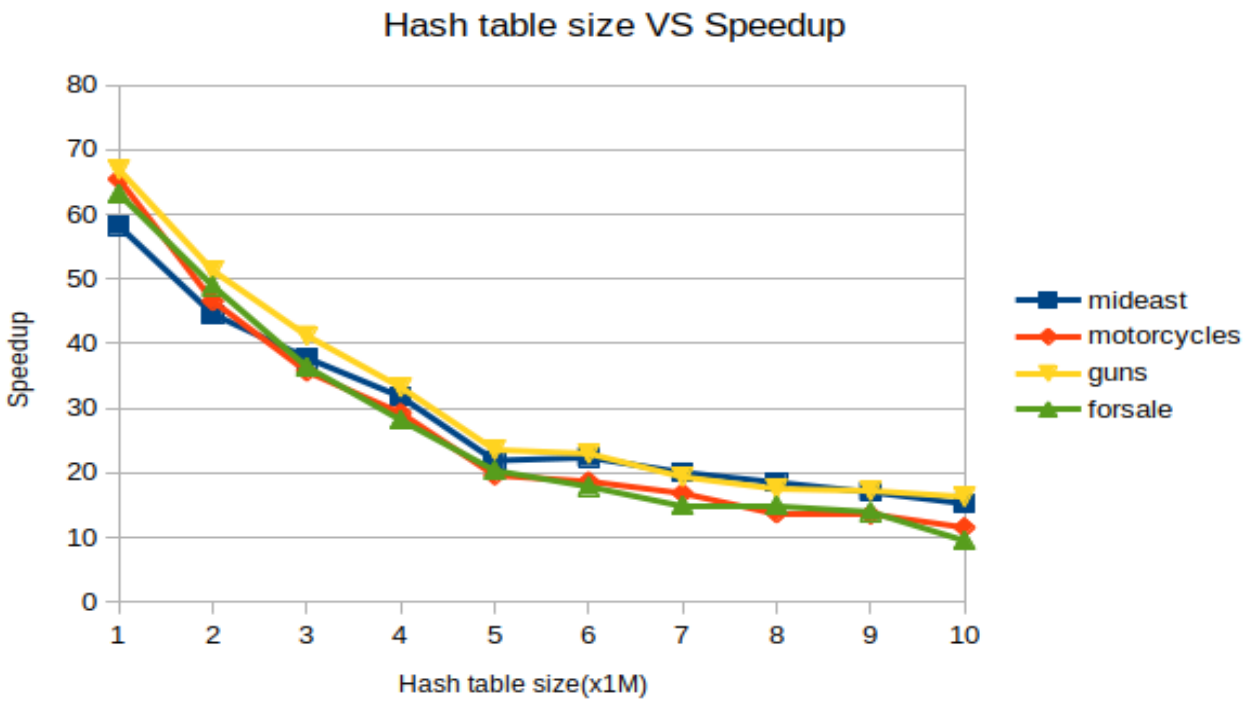
Results (1/2)

- (hash table size ↑) ⇒ (hash collisions ↓) ⇒ (VS→1)



Results (2/2)

- $speedup = \frac{serial\ time}{parallel\ time}$
- (hash table size ↑) ⇒ (parallel time ↑) ⇒ (speedup ↓)



Conclusions – Future work

- Update operation indeed parallelizable
  - Non-negligible speedup
  - Speedup – VS tradeoff
  - VS threshold should be examined
- Collision resolution for GPU program
- Similar n-gram graph operations parallelizable