L10– GRAPH ALGORITHM

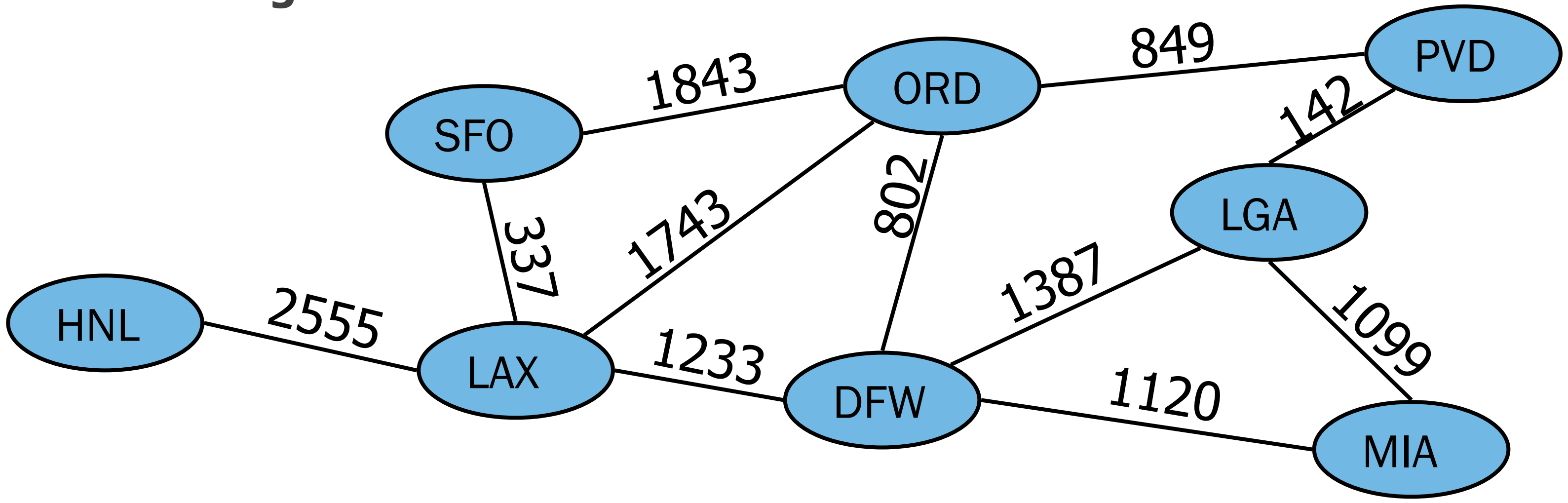
BMCS3003 DISTRIBUTED SYSTEMS AND PARALLEL COMPUTING

CONTENTS

GRAPH TERMINOLOGY

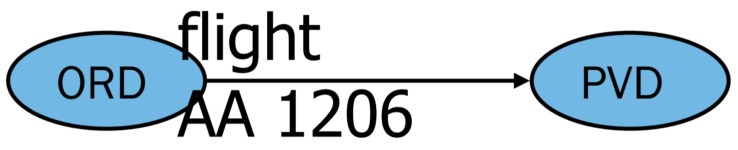
GRAPHS

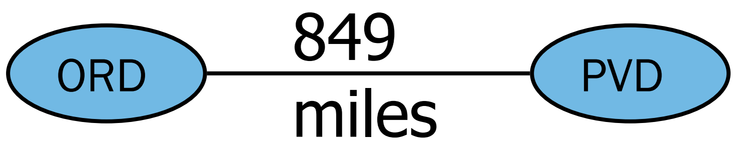
- A graph is a pair (V, E), where
  - V is a set of nodes, called **vertices**
  - E is a collection of pairs of vertices, called **edges**
  - Vertices and edges are positions and store elements
- Example:
  - A **vertex** represents an **airport** and stores the **three-letter airport code**
  - An **edge** represents a flight **route** between two airports and stores the **mileage** of the route



2 EDGE TYPES

- **Directed** edge (**ARROW**)
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- **Undirected** edge (**no arrow**)
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- **Directed** graph
  - all the edges are directed
  - e.g., route network
- **Undirected** graph
  - all the edges are undirected
  - e.g., flight network



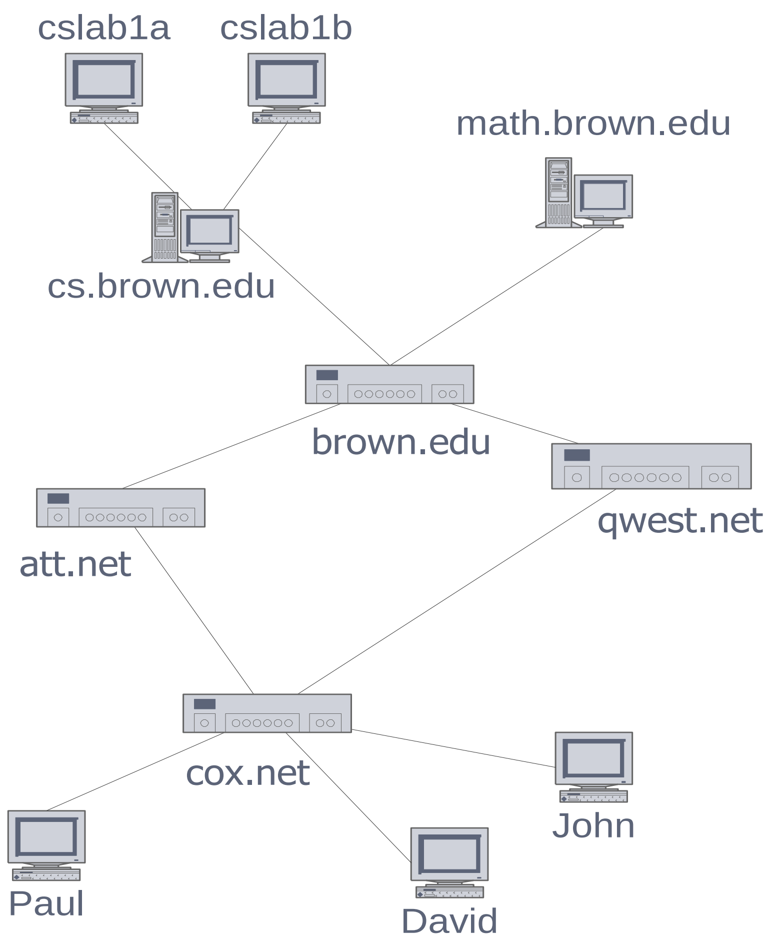


A road between two points
Could be **one way** (directed)
Or **two way** (undirected)

Could have more than one edge
(e.g. toll road and public road)

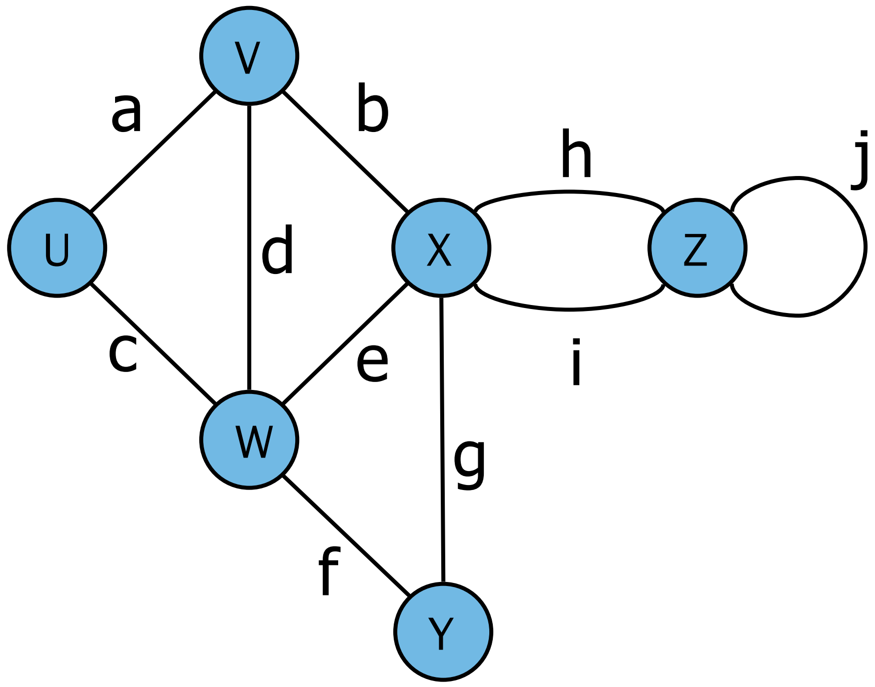
APPLICATIONS

- **Electronic circuits**
  - Printed circuit board
  - Integrated circuit
- **Transportation networks**
  - Highway network
  - Flight network
- **Computer networks**
  - Local area network
  - Internet
  - Web
- **Databases**
  - Entity-relationship diagram



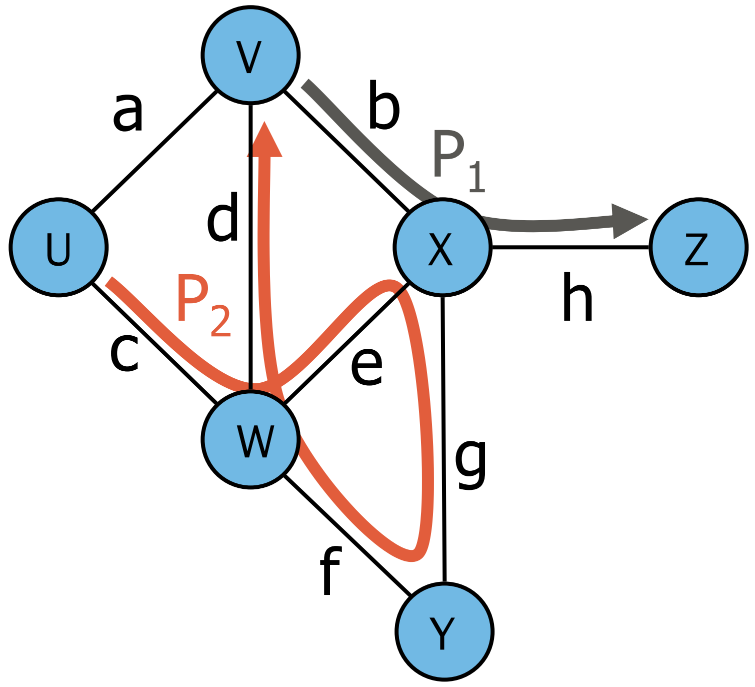
TERMINOLOGY

- End vertices (or endpoints) of an edge
  - **U and V are the endpoints of a**
- Edges incident on a vertex
  - **a, d, and b are incident on V**
- Adjacent vertices
  - **U and V are adjacent**
- Degree of a vertex
  - **X has degree 5**
- Parallel edges
  - **h and i are parallel edges**
- Self-loop
  - **j is a self-loop**



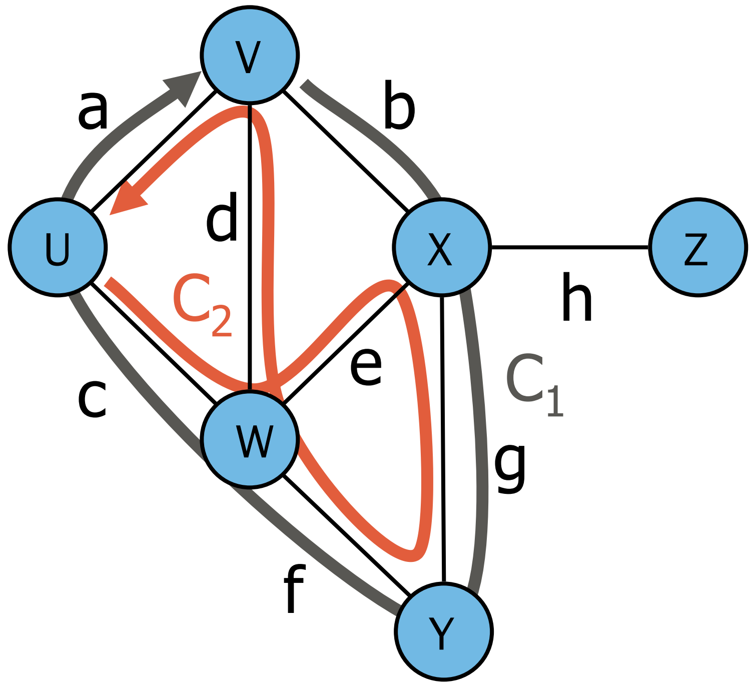
TERMINOLOGY (CONT.)

- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- **Simple path**
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a **simple** path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is **not simple**



TERMINOLOGY (CONT.)

- **Cycle**
  - **circular** sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- **Simple cycle**
  - cycle such that all its vertices and edges are **distinct**
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a **simple cycle**
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is **not simple**

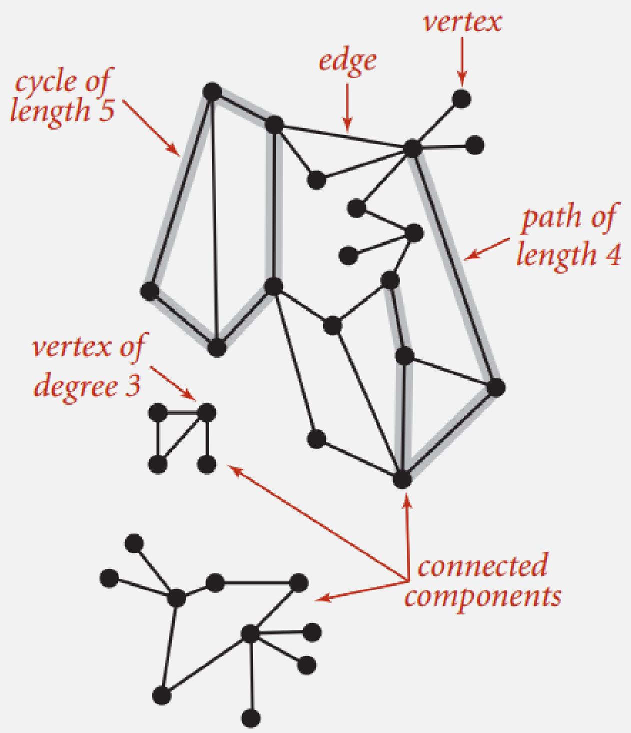


Graph terminology

Path. Sequence of vertices connected by edges.

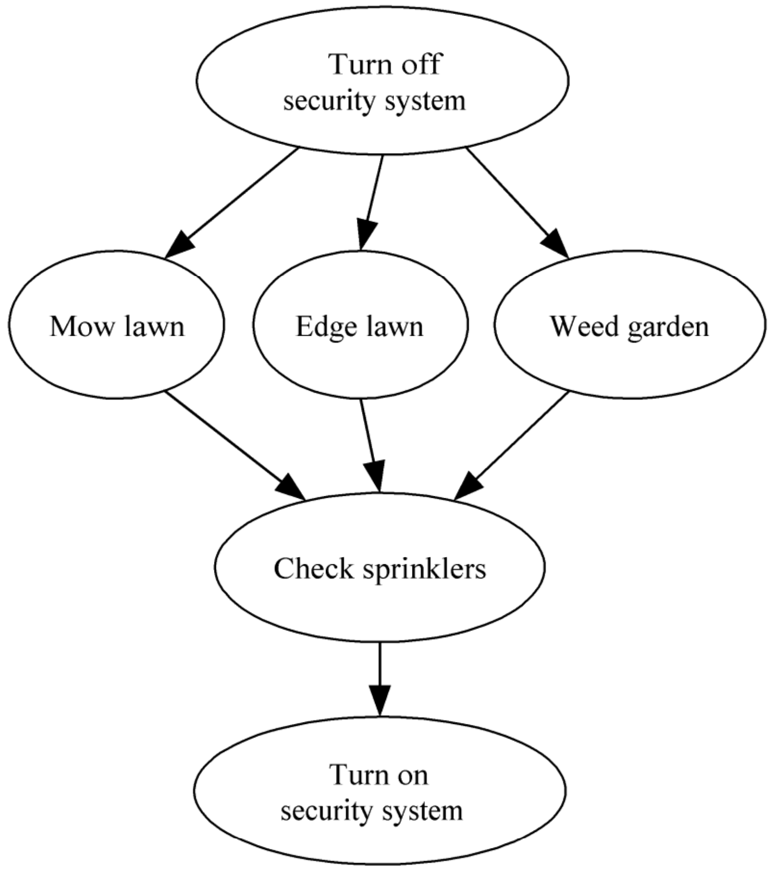
Cycle. Path whose first and last vertices are the same.

Two vertices are connected if there is a path between them.



DATA DEPENDENCE GRAPH

- Directed graph
- Vertices = tasks
- Edges = dependences
- Edge from u to v means that task u must finish before task v can start.

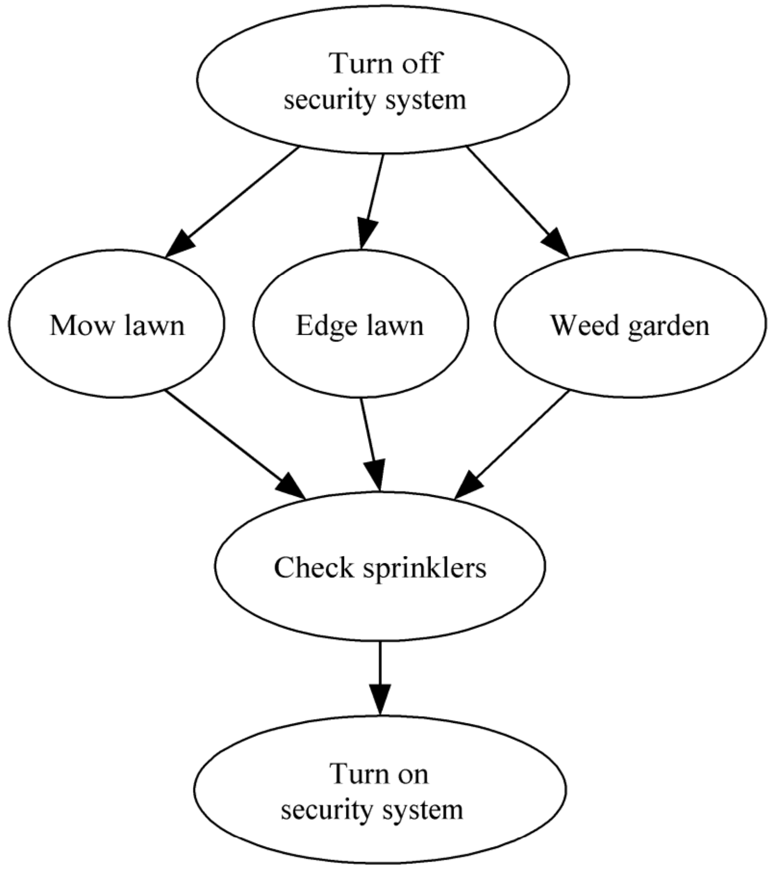


DATA DEPENDENCE GRAPH

- Can be used to identify data parallelism and job parallelism.
- Most realistic jobs contain both parallelisms
  - Can be viewed as branches in data parallel tasks

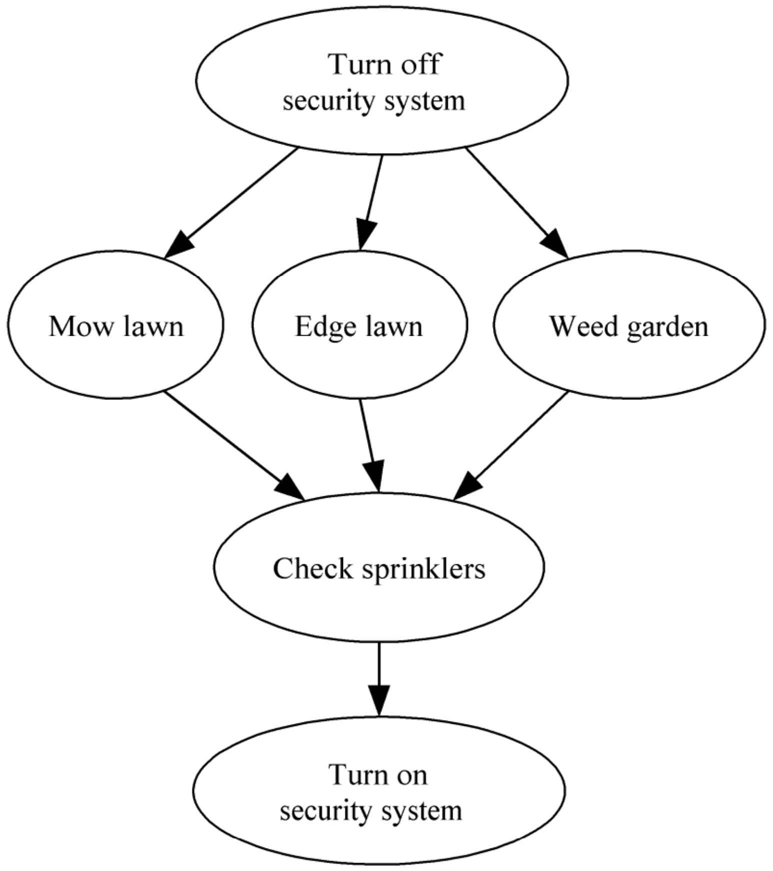
  - If no path from vertex u to vertex v, then job parallelism can be used to execute the tasks u and v concurrently.

  - If larger tasks can be subdivided into smaller identical tasks, data parallelism can be used to execute these concurrently.



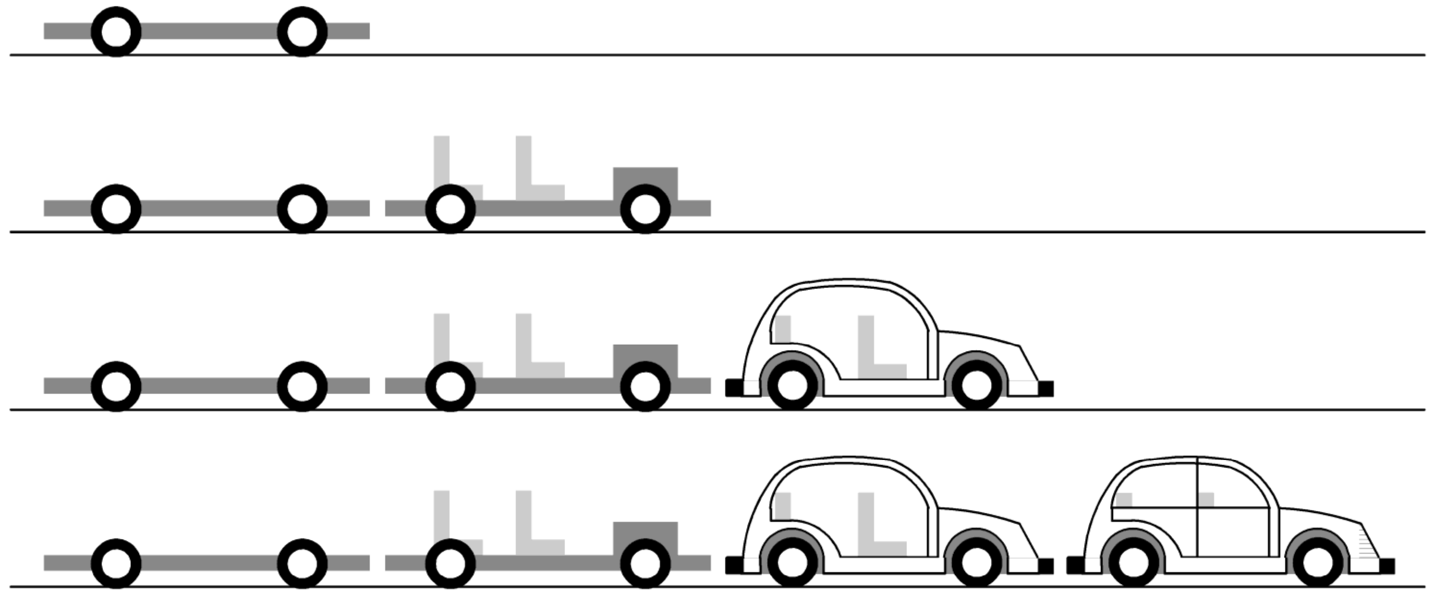
FOR EXAMPLE, “MOW LAWN” BECOMES

- Mow N lawn
- Mow S lawn
- Mow E lawn
- Mow W lawn
- If 4 people are available to mow, then data parallelism can be used to do these tasks simultaneously.
- Similarly, if several people are available to “edge lawn” and “weed garden”, then we can use data parallelism to provide more concurrency.



PIPELINING

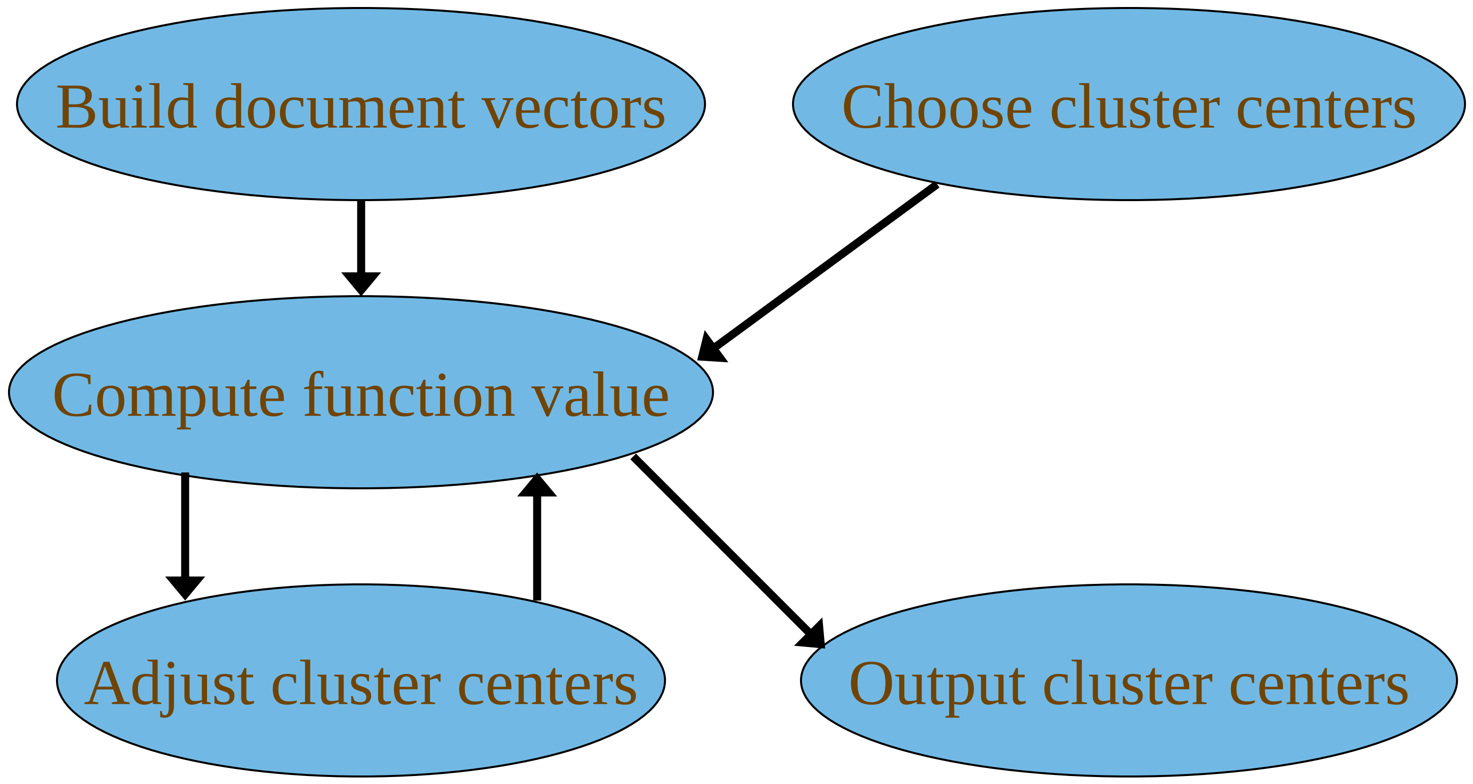
- Divide a process into stages
- Produce several items simultaneously



FUNCTIONAL PARALLELISM OPPORTUNITIES

- Draw data dependence diagram
- Look for sets of nodes such that there are no paths from one node to another

DATA DEPENDENCE DIAGRAM



FUNCTIONAL PARALLELISM TASKS

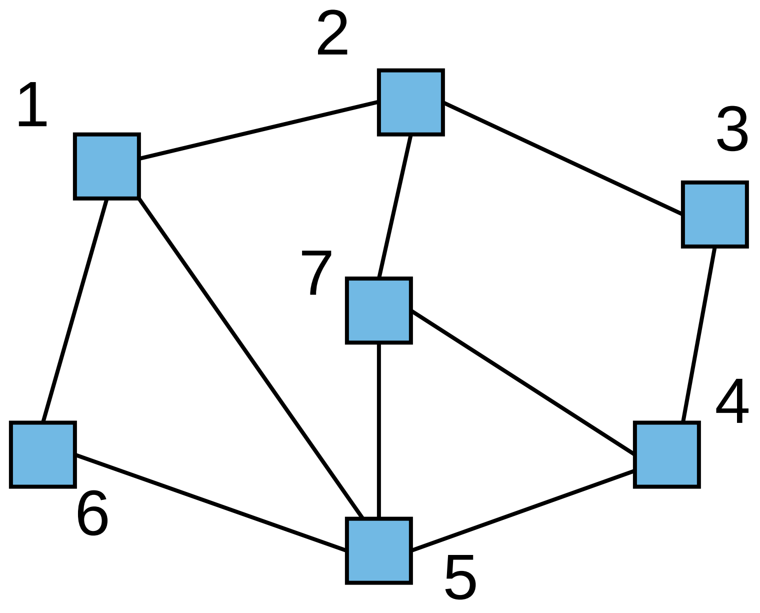
- The only independent sets of vertices are
  - Those representing generating vectors for documents and
  - Those representing initially choosing cluster centers
  - i.e. the first two in the diagram.
- These two set of tasks could be performed concurrently

DATA STRUCTURES TO STORE GRAPHS

EDGE LIST

- **edge list**: an unordered list of all edges in the graph

1	1	1	2	2	3	5	5	5	7
2	5	6	7	3	4	6	7	4	4



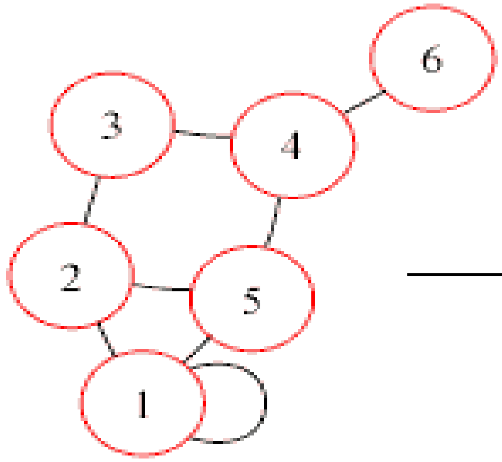
EDGE LIST: PROS AND CONS

- ***advantages***
  - easy to loop/iterate over all edges
- ***disadvantages***
  - hard to tell if an edge exists from A to B
  - hard to tell how many edges a vertex touches (its degree)

1	1	1	2	2	3	5	5	5	7
2	5	6	7	3	4	6	7	4	4

ADJACENCY MATRIX

- **adjacency matrix**: an n × n matrix where:
  - the **nondiagonal** entry a_{ij} is the number of edges joining vertex *i* and vertex *j* (or the weight of the edge joining vertex *i* and vertex *j*)
  - the **diagonal** entry a_{ii} corresponds to the number of **loops (self-connecting edges)** at vertex *i*



$$\xrightarrow{\qquad\qquad} \begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

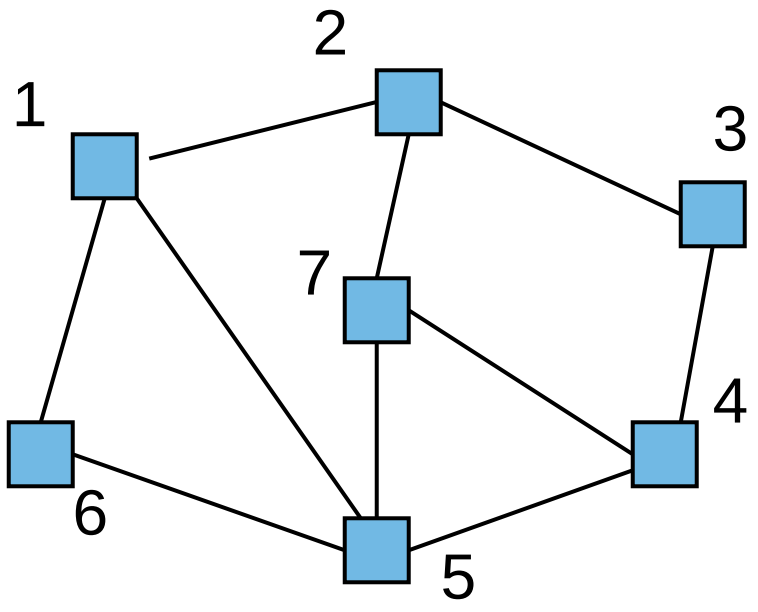
ADJACENCY MATRIX: PROS AND CONS

- ***advantages***
  - fast to tell whether edge exists between any two vertices *i* and *j* (and to get its weight)
- ***disadvantages***
  - consumes a lot of memory on **sparse** graphs (ones with few edges)
  - redundant information for undirected graphs

ADJACENCY MATRIX EXAMPLE

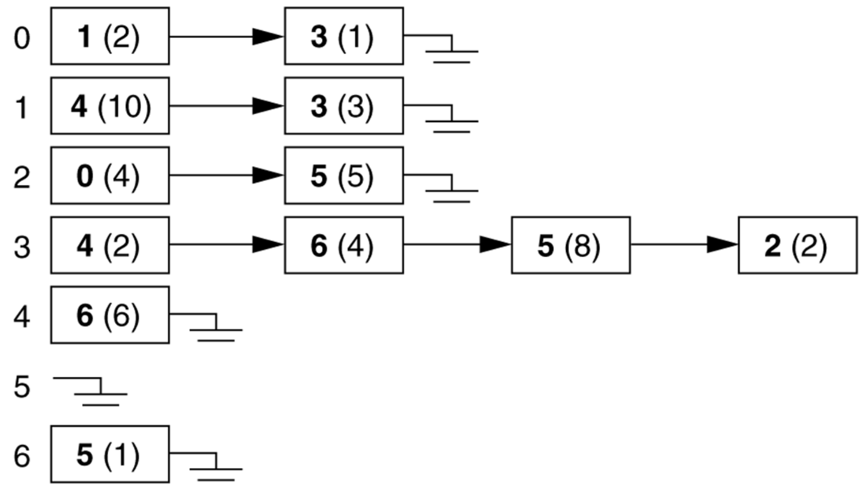
- How do we figure out the degree of a given vertex?
- How do we find out whether an edge exists from A to B?
- How could we look for loops in the graph?

	1	2	3	4	5	6	7
1	0	1	0	0	1	1	0
2	1	0	1	0	0	0	1
3	0	1	0	1	0	0	0
4	0	0	1	0	1	0	1
5	1	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	1	0	1	1	0	0



ADJACENCY LISTS

- **adjacency list**: stores edges as individual linked lists of references to each vertex's neighbors

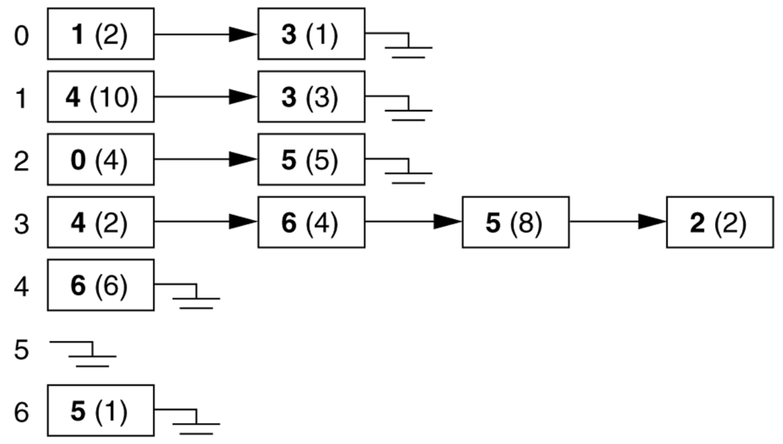


ADJACENCY LIST: PROS AND CONS

- ***advantages***:
  - new nodes can be added easily
  - new nodes can be connected with existing nodes easily
  - "who are my neighbors" easily answered
- ***disadvantages***:
  - determining whether an edge

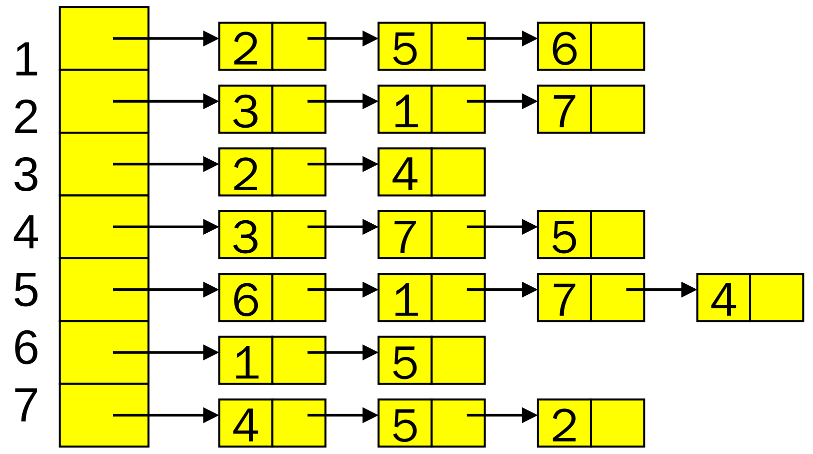
exists between two nodes:

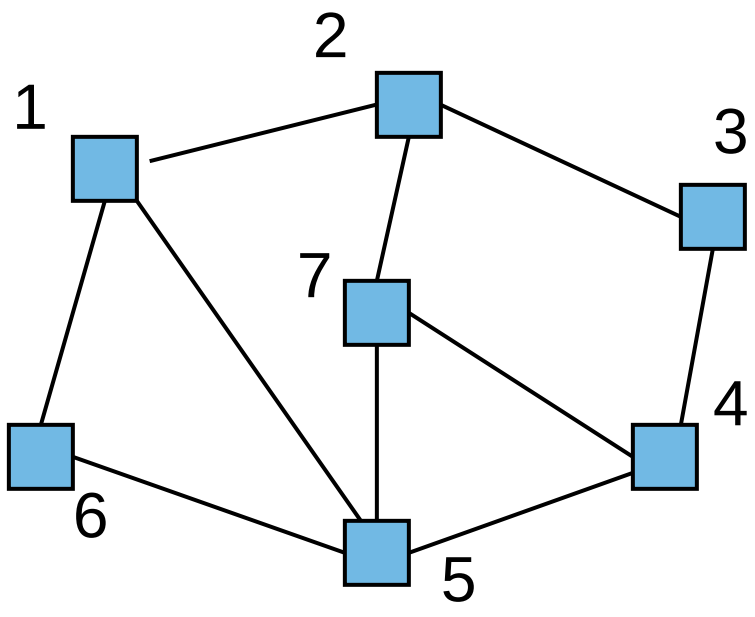
O(average degree)



ADJACENCY LIST EXAMPLE

- How do we figure out the degree of a given vertex?
- How do we find out whether an edge exists from A to B?
- How could we look for loops in the graph?





RUNTIME TABLE

▪ ***n*** vertices, ***m*** edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	***n*** + ***m***	***n*** + ***m***	n^2
Finding all adjacent vertices to ***v***	***m***	deg(***v***)	***n***
Determining if ***v*** is adjacent to ***w***	***m***	deg(***v***)	1
adding a vertex	1	1	n^2
adding an edge to ***v***	1	1	1
removing vertex ***v***	***m***	***n***	n^2
removing an edge from ***v***	***m***	deg(***v***)	1

SOLVING PROBLEM WITH GRAPH

GRAPH TRAVERSALS - DEPTH FIRST SEARCH

The general "rule" used in searching a graph using a depth first search is to search down a path from a particular source vertex as far as you can go. When you can go to farther, "backtrack" to the last vertex from which a different path could have been taken. Continue in this fashion, attempting to go as deep as possible down each path until each node has been visited.

The most difficult part of this algorithm is keeping track of what nodes have already been visited, so that the algorithm does not run ad infinitum. We can do this by labeling each visited node and labeling "discovery" and "back" edges.

THE ALGORITHM – DEPTH FIRST SEARCH

```
DFS(Graph G,vertex v):
For  all edges e incident to the start vertex v do:
    1) If e is unexplored
        a) Let e connect v to w.
        b) If w is unexplored, then
            i) Label e as a discovery edge
            ii) Recursively call DFS(G,w)
          else
            iii) Label e as a back edge
```

THE PROOF – DEPTH FIRST SEARCH

- To prove that this algorithm visits all vertices in the connected component of the graph in which it starts, note the following:

- Let the vertex u be the first vertex on any path from the source vertex that is not visited. That means that w, which is connected to u was visited, but by the algorithm given, it's clear that if this situation occurs, u must be visited, contradicting the assumption that u was unvisited.

- Next, we must show that the algorithm terminates. If it does not, then there must exist a "search path" that never ends. But this is impossible. A search path ends when an already visited vertex is visited again. The longest path that exists without revisiting a vertex is of length V, the number of vertices in the graph.

- The running time of DFS is O(V+E). To see this, note that each edge and vertex is visited at most twice. In order to get this efficiency, an adjacency list must be used. (An adjacency matrix can not be used to complete this algorithm that quickly.)

GRAPH TRAVERSE – BREADTH FIRST SEARCH

The idea in a breadth first search is opposite to a depth first search. Instead of searching down a single path until you can go no longer, you search all paths at an uniform depth from the source before moving onto deeper paths. Once again, we'll need to mark both edges and vertices based on what has been visited.

In essence, we only want to explore one "unit" away from a searched node before we move to a different node to search from. All in all, we will be adding nodes to the back of a queue to be ones to searched from in the future.

In the implementation on the following page, a set of queues Li are maintained, each storing a list of vertices a distance of i edges from the starting vertex. One can implement this algorithm with a single queue as well.

THE ALGORITHM – BREADTH FIRST SEARCH

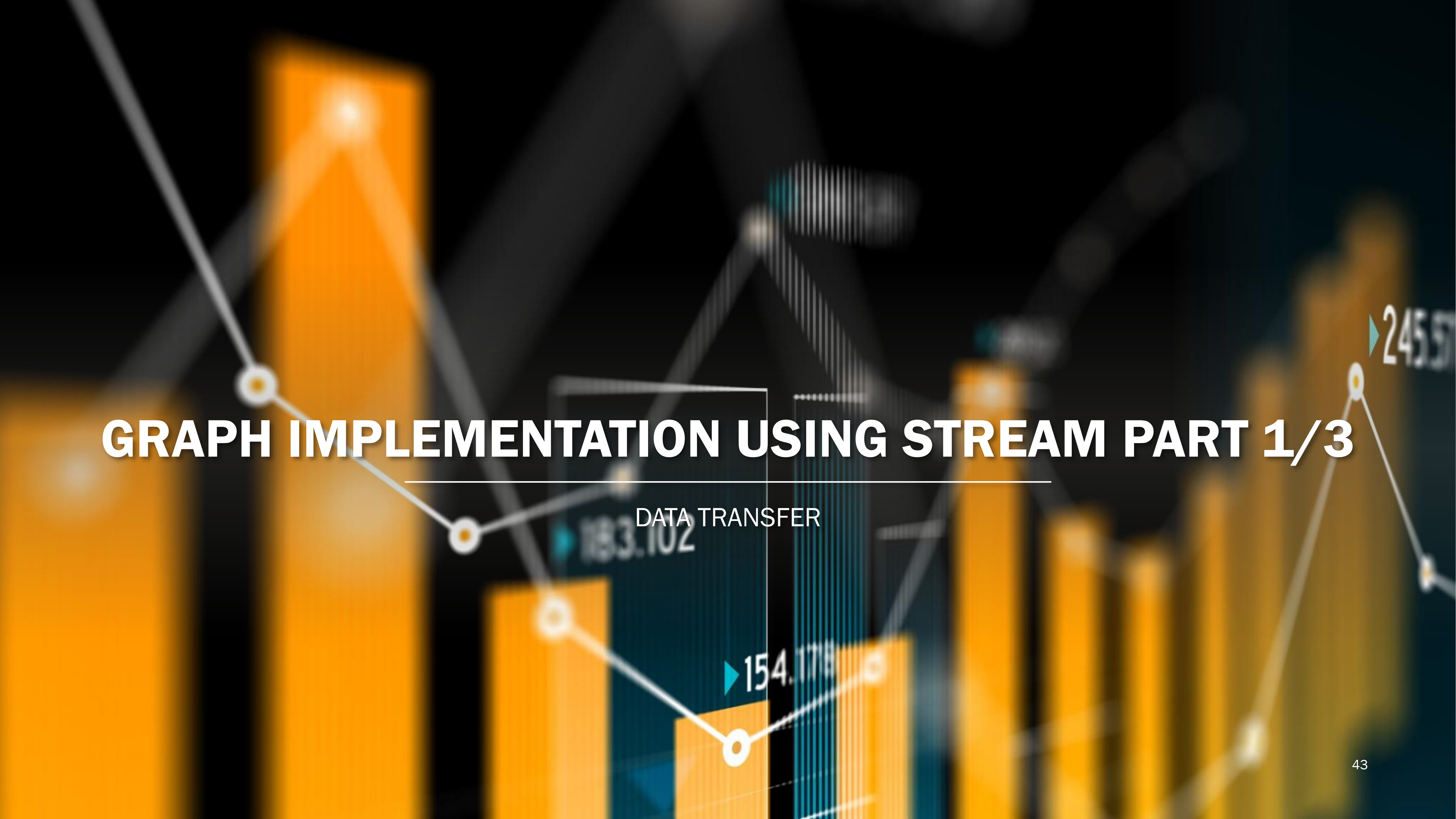
Let Li be the set of vertices visited that are a path length of i from the source vertex for the algorithm.

```
BFS(G,s):
1) Let L0 be empty
2) Insert s into L0.
3) Let i = 0
4) While Li is not empty do the following:
    A) Create an empty container Li+1.
    B) For each vertex v in Li do
        i) For all edges e incident to v
            a) if e is unexplored, mark endpoint w.
            b) if w is unexplored
                Mark it.
                Insert w into Li+1.
                Label e as a discovery edge.
              else
                Label e as a cross edge.
    C) i = i+1
```

THE PROOF – BREADTH FIRST SEARCH

- **The basic idea here is that we have successive rounds and continue with our rounds until no new vertices are visited on a round. For each round, we look at each vertex connected to the vertex we came from. And from this vertex we look at all possible connected vertices.**

- **This leaves no vertex unvisited because we continue to look for vertices until no new ones of a particular length are found. If there are no paths of length 10 to a new vertex, surely there can be no paths of length 11 to a new vertex. The algorithm also terminates since no path can be longer than the number of vertices in the graph.**



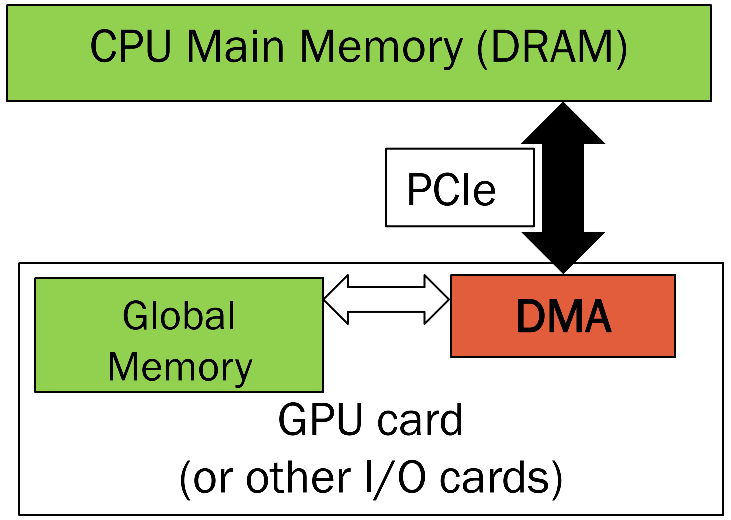
GRAPH IMPLEMENTATION USING STREAM PART 1/3

DATA TRANSFER

CPU-GPU DATA TRANSFER USING DMA

– DMA (Direct Memory Access) hardware is used by `cudaMemcpy()` for better efficiency

- Frees CPU for other tasks
- Hardware unit specialized to transfer a number of bytes requested by OS
- Between physical memory address space regions (some can be mapped I/O memory locations)
- Uses system interconnect, typically PCIe in today's systems



VIRTUAL MEMORY MANAGEMENT

- Modern computers use virtual memory management
  - Many virtual memory spaces mapped into a single physical memory
  - Virtual addresses (pointer values) are translated into physical addresses
- Not all variables and data structures are always in the physical memory
  - Each virtual address space is divided into pages that are mapped into and out of the physical memory
  - Virtual memory pages can be mapped out of the physical memory (page-out) to make room
  - Whether or not a variable is in the physical memory is checked at address translation time

DATA TRANSFER AND VIRTUAL MEMORY

- DMA uses physical addresses
  - When `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers
  - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
  - No address translation for the rest of the same DMA transfer so that high efficiency can be achieved

- The OS could accidentally page-out the data that is being read or written by a DMA and page-in another virtual page into the same physical location

PINNED MEMORY AND DMA DATA TRANSFER

– Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out

– Allocated with a special system API function call

– a.k.a. Page Locked Memory, Locked Pages, etc.

– CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

CUDA DATA TRANSFER USES PINNED MEMORY.

– The DMA used by cudaMemcpy() requires that any source or destination in the host memory is allocated as pinned memory

– If a source or destination of a `cudaMemcpy()` in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead

– `cudaMemcpy()` is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

ALLOCATE/FREE PINNED MEMORY

- `cudaHostAlloc()`, three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use `cudaHostAllocDefault` for now

- `cudaFreeHost()`, one parameter
  - Pointer to the memory to be freed

USING PINNED MEMORY IN CUDA

- Use the allocated pinned memory and its pointer the same way as those returned by `malloc();`
- The only difference is that the allocated memory cannot be paged by the OS
- The `cudaMemcpy()` function should be about 2X faster with pinned memory
- Pinned memory is a limited resource
  - over-subscription can have serious consequences

PUTTING IT TOGETHER - VECTOR ADDITION HOST CODE EXAMPLE

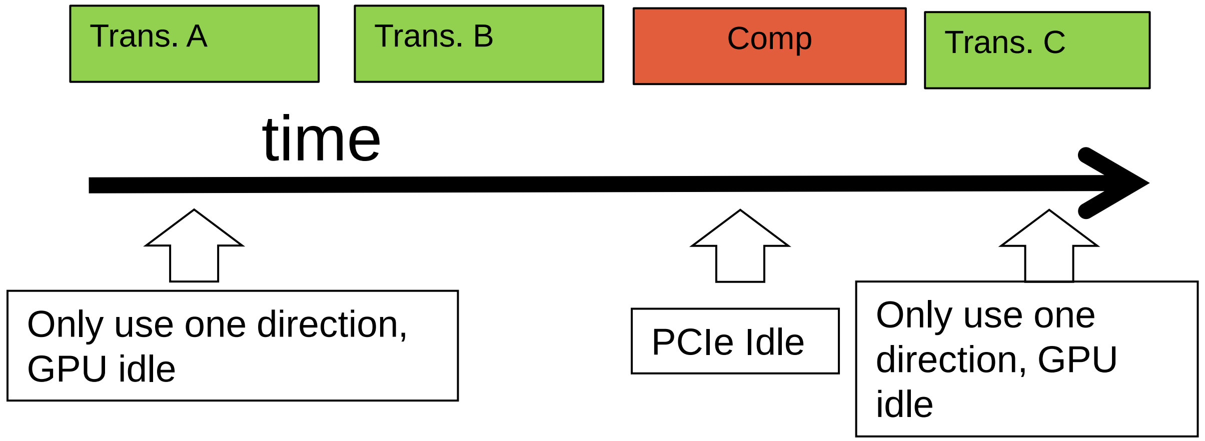
```
int main()
{
   float *h_A, *h_B, *h_C;
…
   cudaHostAlloc((void **) &h_A, N* sizeof(float),
      cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_B, N* sizeof(float),
      cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_C, N* sizeof(float),
      cudaHostAllocDefault);
…
   // cudaMemcpy() runs 2X faster
}
```

GRAPH IMPLEMENTATION USING STREAM PART 2/3

CUDA STREAMS

SERIALIZED DATA TRANSFER AND COMPUTATION

– So far, the way we use `cudaMemcpy` serializes data transfer and GPU computation for `VecAddKernel()`


Trans. A
Trans. B
Comp
Trans. C
time
Only use one direction, GPU idle
PCIe Idle
Only use one direction, GPU idle

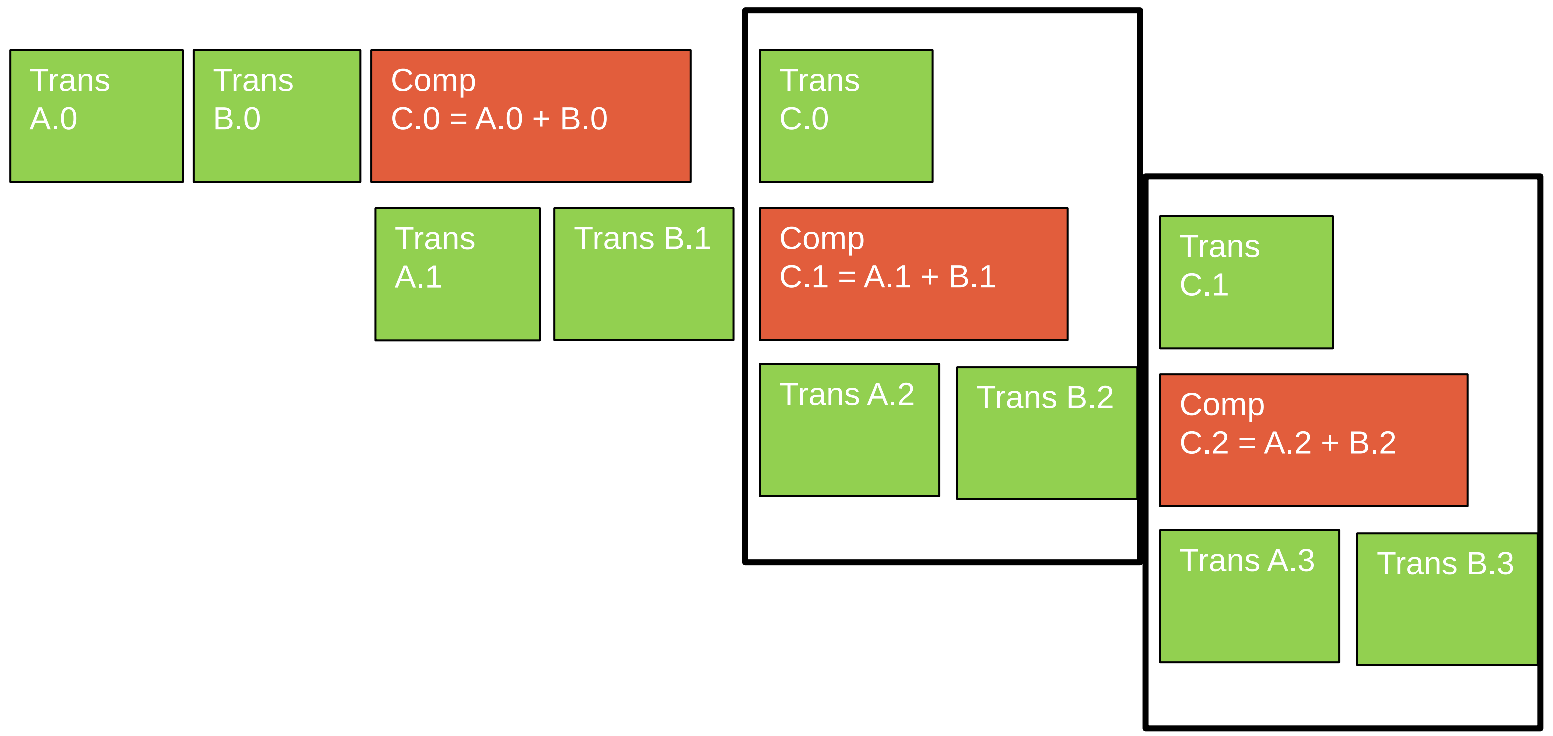
DEVICE OVERLAP

- Some CUDA devices support device overlap
  - Simultaneously execute a kernel while copying data between device and host memory

```
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount( &dev_count);
for (int i = 0; i < dev_count; i++) {
  cudaGetDeviceProperties(&prop, i);
  if (prop.deviceOverlap) …
```

IDEAL, PIPELINED TIMING

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

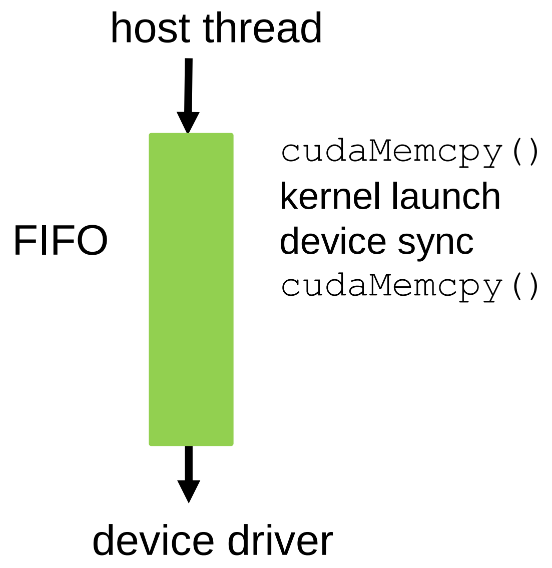


CUDA STREAMS

- CUDA supports parallel execution of kernels and `cudaMemcpy()` with “Streams”
- Each stream is a queue of operations (kernel launches and `cudaMemcpy()` calls)
- Operations (tasks) in different streams can go in parallel
  - “Task parallelism”

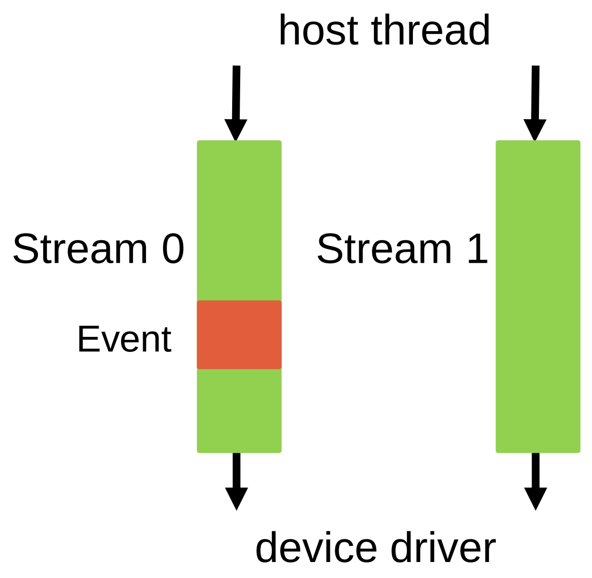
STREAMS

- Requests made from the host code are put into First-In-First-Out queues
  - Queues are read and processed asynchronously by the driver and device
  - Driver ensures that commands in a queue are processed in sequence. E.g., Memory copies end before kernel launch, etc.

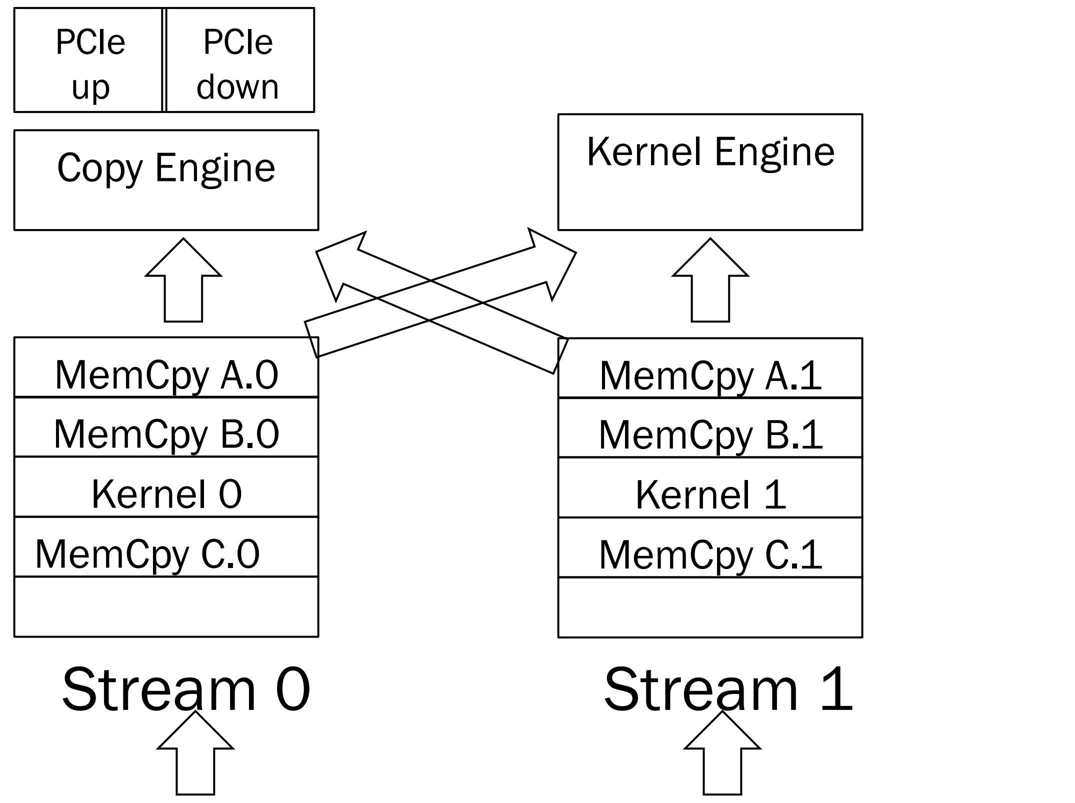


STREAMS CONT.

- To allow concurrent copying and kernel execution, use multiple queues, called “streams”
  - CUDA “events” allow the host thread to query and synchronize with individual queues (i.e. streams).



CONCEPTUAL VIEW OF STREAMS



Operations (Kernel launches, `cudaMemcpy()` calls)

GRAPH IMPLEMENTATION USING STREAM PART 3/3

OVERLAPPING DATA TRANSFER WITH COMPUTATION

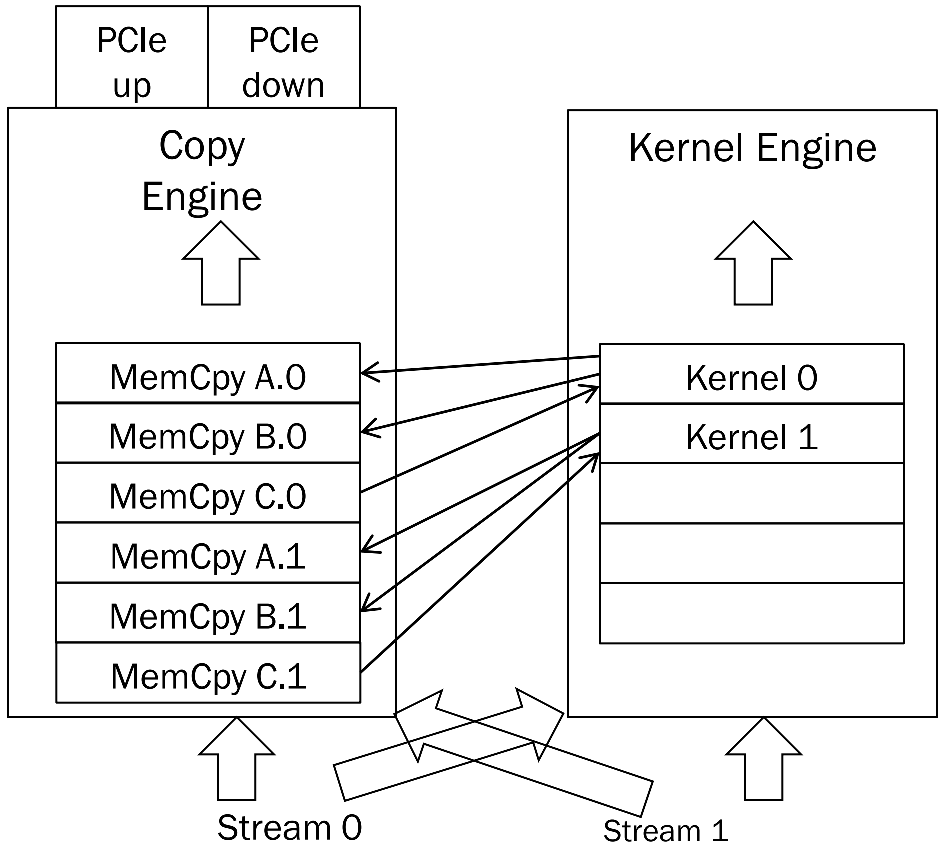
SIMPLE MULTI-STREAM HOST CODE

```
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

float *d_A0, *d_B0, *d_C0; // device memory for stream 0
float *d_A1, *d_B1, *d_C1; // device memory for stream 1

// cudaMalloc() calls for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here
```

SIMPLE MULTI-STREAM HOST CODE (CONT.)

```
for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),…, stream0);
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),…, stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0,…);
    cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),…, stream0);
    cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),…, stream1);
    cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),…,       stream1);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);
    cudaMemcpyAsync(d_C1, h_C+i+SegSize, SegSize*sizeof(float),…,       stream1);
}
```

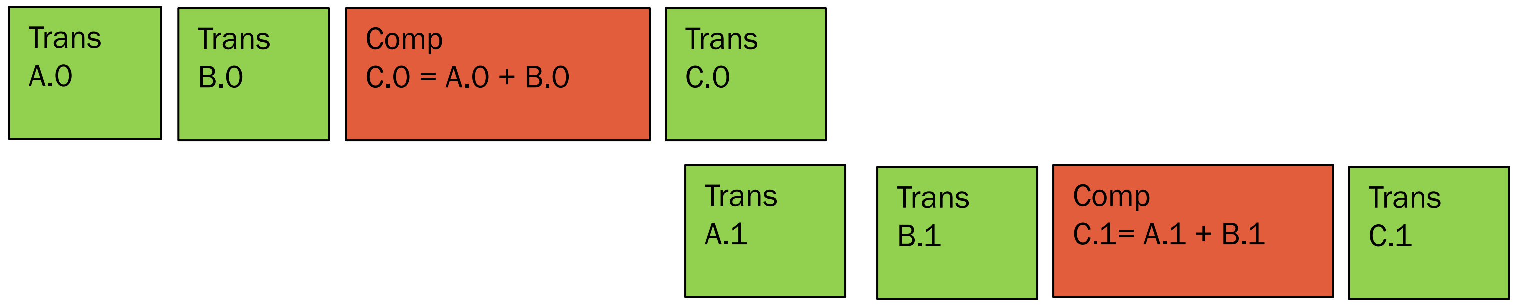
A VIEW CLOSER TO REALITY IN PREVIOUS GPUS



Operations (Kernel launches, `cudaMemcpy()` calls)

NOT QUITE THE OVERLAP WE WANT IN SOME GPUS

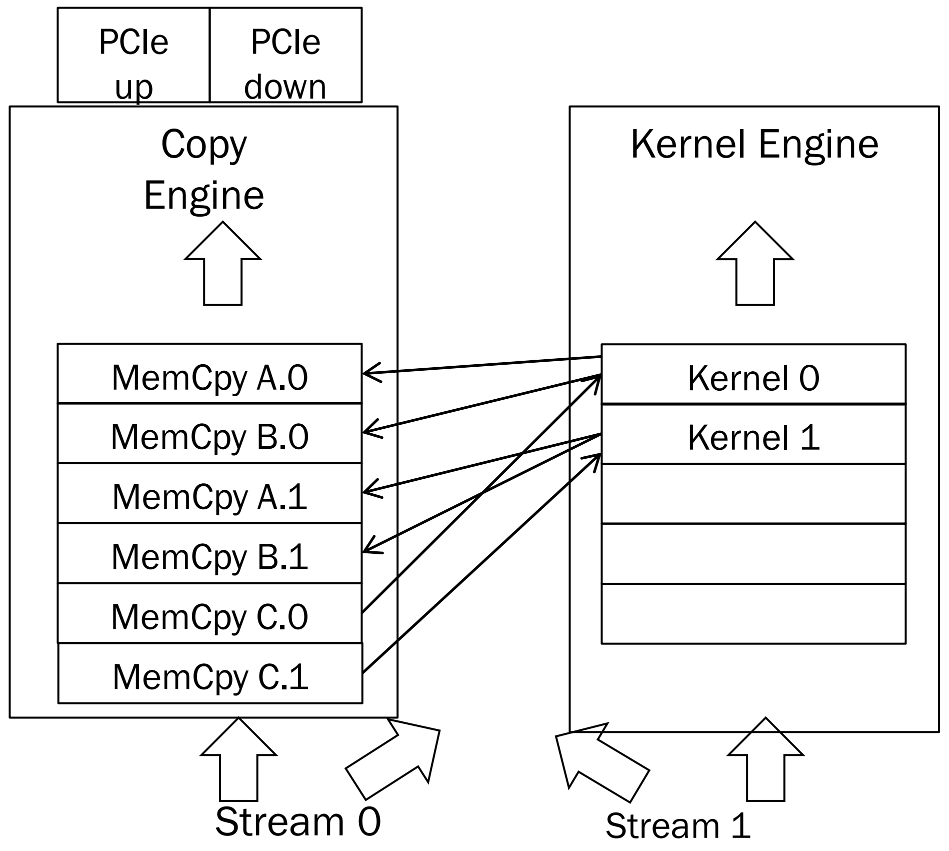
– C.0 blocks A.1 and B.1 in the copy engine queue



BETTER MULTI-STREAM HOST CODE

```
for (int i=0; i<n; i+=SegSize*2) {
  cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),..., stream0);
  cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),..., stream0);
  cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),..., stream1);
  cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),..., stream1);

  vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
  vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);

  cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),..., stream0);
  cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),..., stream1);
}
```

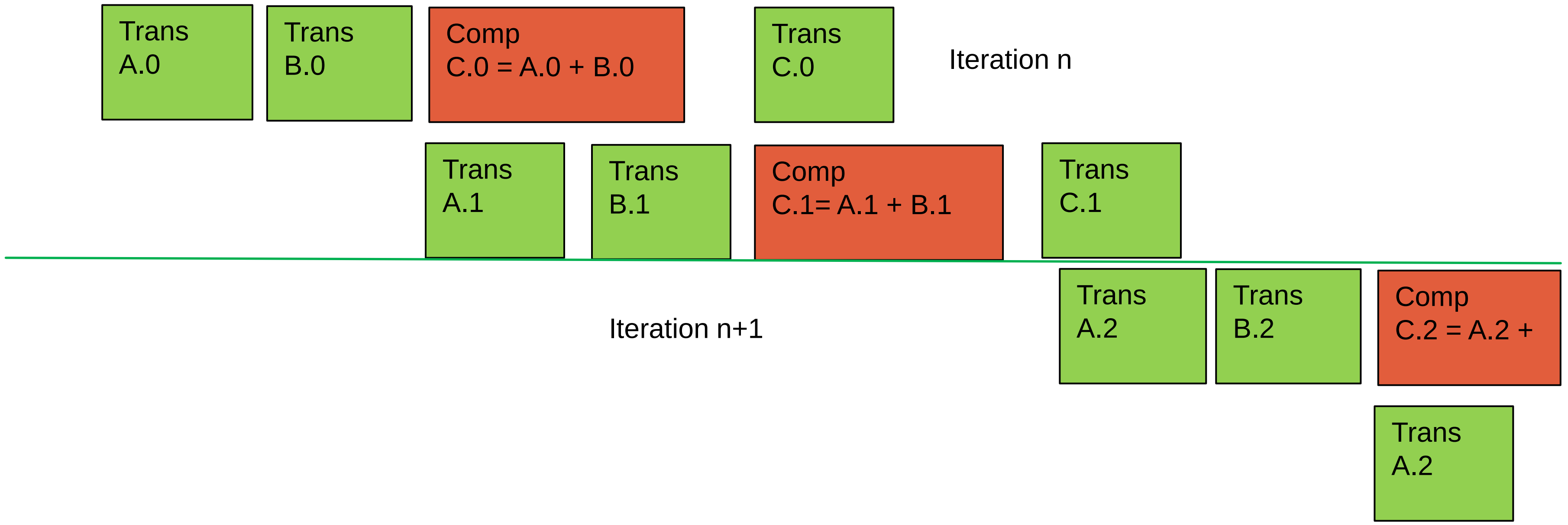
C.0 NO LONGER BLOCKS A.1 AND B.1



Operations (Kernel launches, cudaMemcpy() calls)

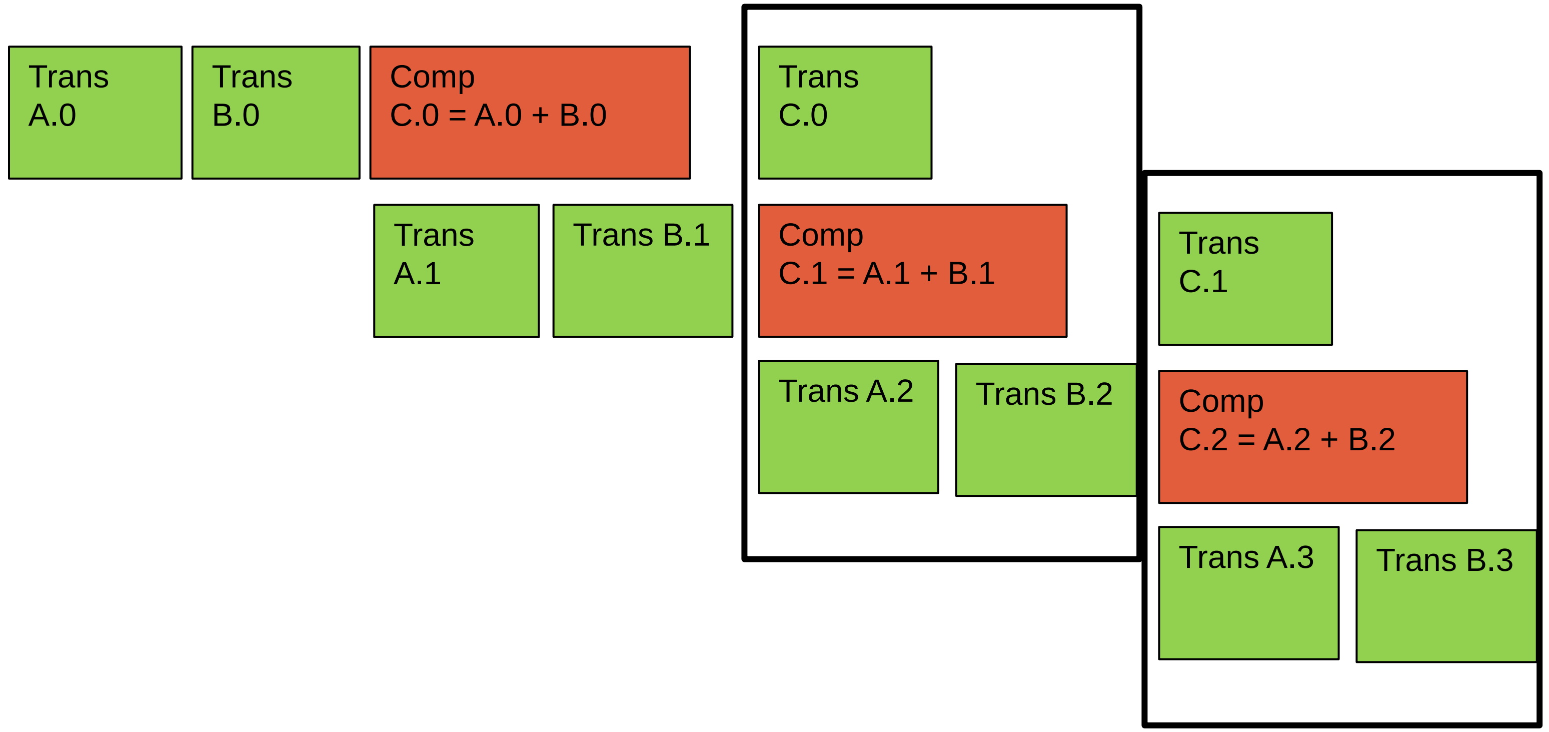
BETTER, NOT QUITE THE BEST OVERLAP

– C.1 blocks next iteration A.0 and B.0 in the copy engine queue



IDEAL, PIPELINED TIMING

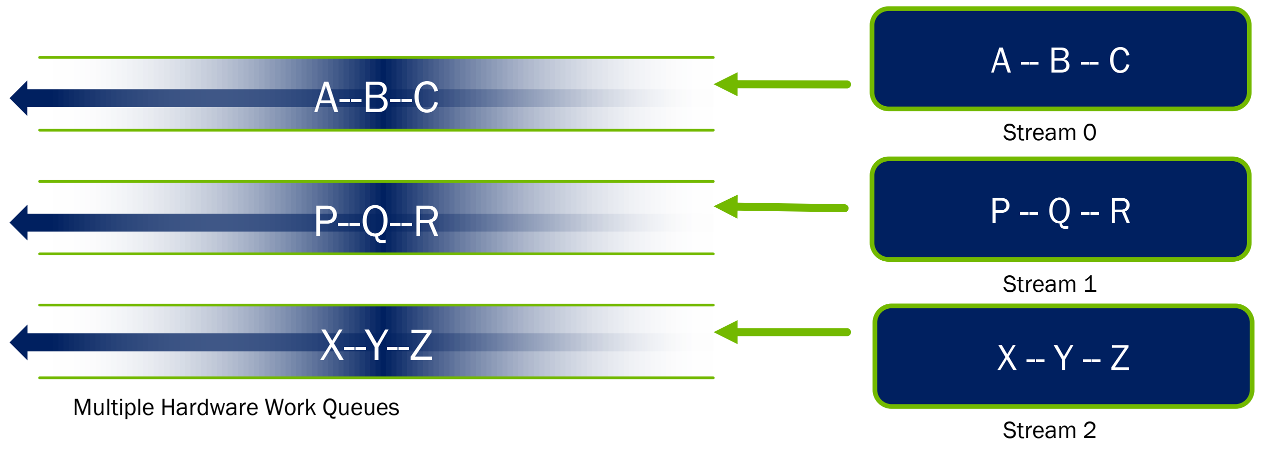
– Will need at least three buffers for each original A, B, and C, code is more complicated



HYPER QUEUES

– Provide multiple queues for each engine

– Allow more concurrency by allowing some streams to make progress for an engine while others are blocked



WAIT UNTIL ALL TASKS HAVE COMPLETED

- `cudaStreamSynchronize(stream_id)`
  - Used in host code
  - Takes one parameter – stream identifier
  - Wait until all tasks in a stream have completed
  - **`E.g., cudaStreamSynchronize(stream0)`** in host code ensures that all tasks in the queues of stream0 have completed

- This is different from **`cudaDeviceSynchronize()`**
  - Also used in host code
  - No parameter
  - **`cudaDeviceSynchronize()`** waits until all tasks in all streams have completed for the current device