

# GPU Computing

Laurea Magistrale in Informatica - AA 2018/19

Docente **G. Grossi**

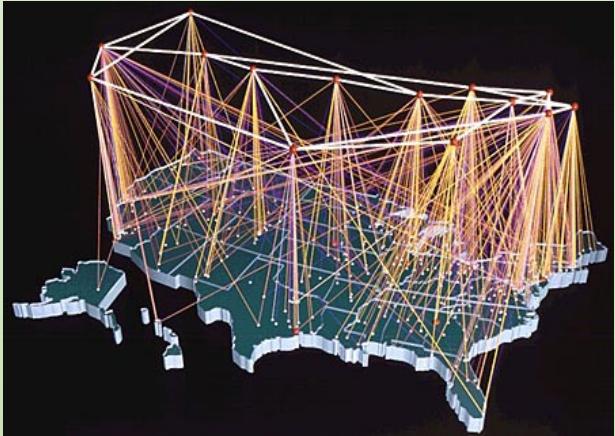
Lezione 10 – algoritmi paralleli su grafi

# Sommario

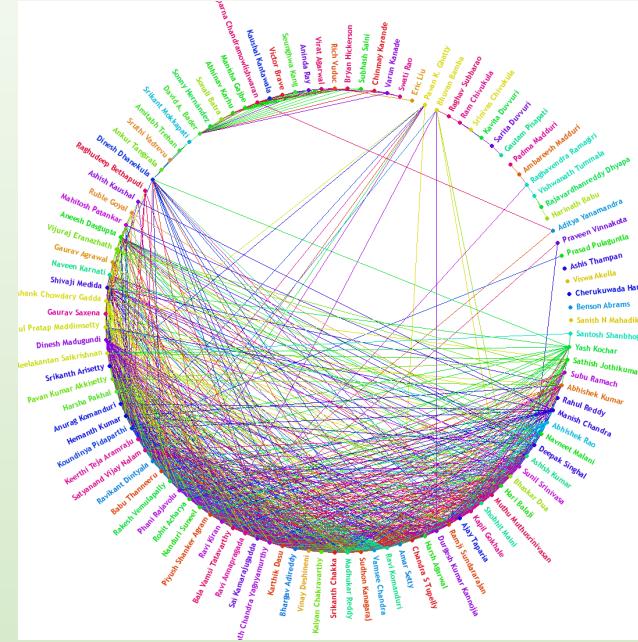
- ✓ Introduzione a problemi su grafo
- ✓ Algoritmo parallelo Breadth-First Search (BFS)
- ✓ Algoritmo parallelo di Boruvka per Min Spanning Tree (MST)
- ✓ Algoritmo greedy parallelo Luby per Graph Coloring (GC)
- ✓ Comunicazione P2P e computazioni multi-GPU distribuita

# Problemi su grafo

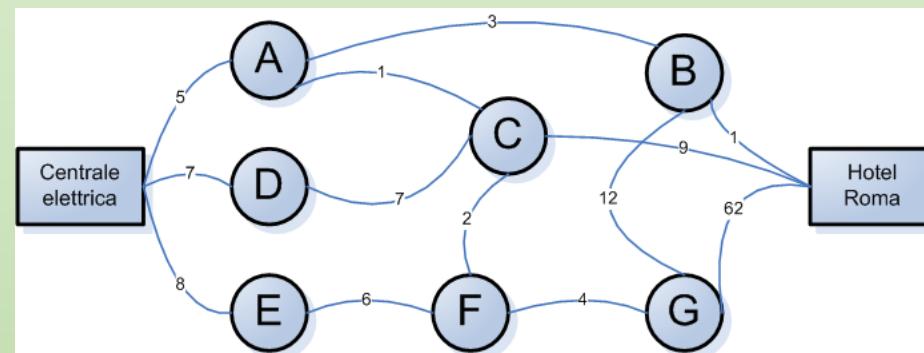
- ✓ Graph traversals
- ✓ Shortest Paths
- ✓ Ranking
- ✓ Maximal Independent Sets
- ✓ Strongly Connected Components
- ✓ Minimum spanning Tree



Internet topologies (router networks)  
are naturally modeled as graphs



clustering and centrality on internet graph



Ford-Fulkerson for MST

# Grafo

Undirected graph:

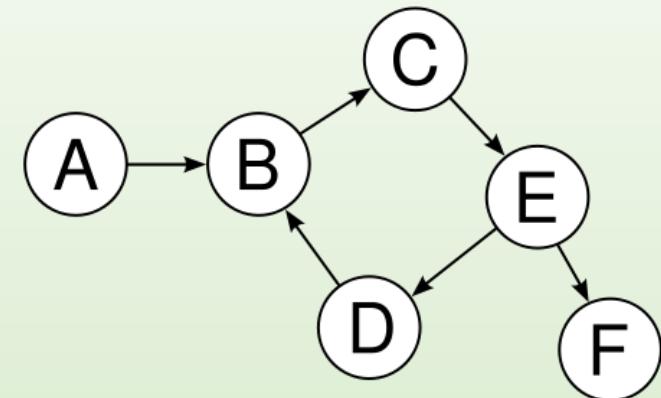
- ✓  $G(V, E)$ :  $V$  is the set of **vertices** and  $E$  is the set of **edges**
- ✓ An **edge**  $e \in E$  is an **unordered pair**  $\{u, v\}$  where  $u, v \in V$

Directed graph:

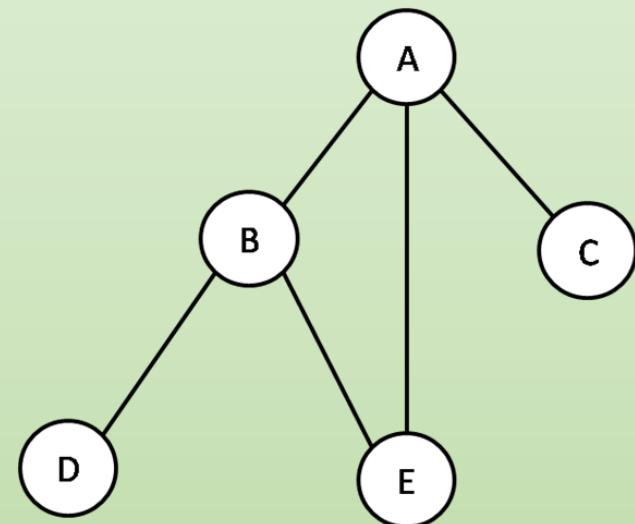
- ✓ An **arc**  $e \in E$  is an **ordered pair**  $e = (u, v) \in E$  (from  $u$  to  $v$ )  
where  $u, v \in V$

Path:

- ✓ A **path** from  $u$  to  $v$  is a **sequence**  $(u, \dots, v)$  of vertices where consecutive vertices in the sequence corresponds to an **edge (arc)** in the graph
- ✓ **Simple path:** all vertices in the path are **distinct**
- ✓ **Cycle:**  $u = v$  (acyclic: contains no cycles)



Grafo orientato



Grafo non orientato

# Proprietà dei grafi

Connection in graphs:

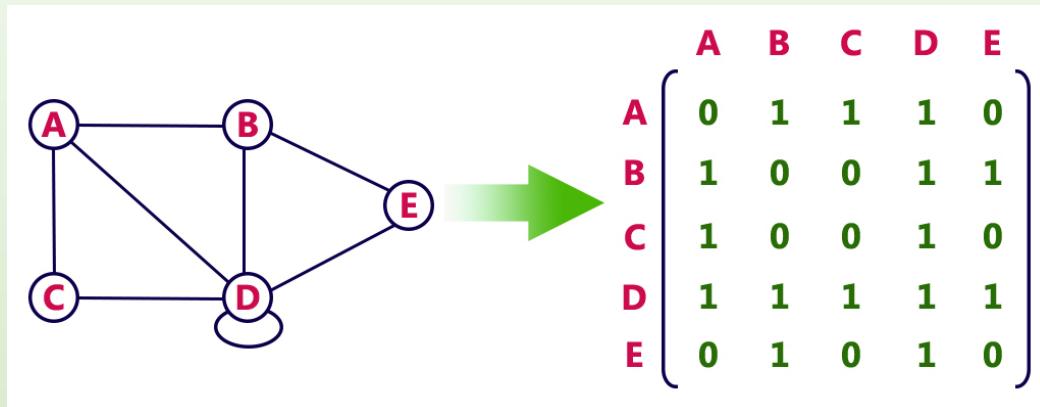
- ✓ A graph is **connected** if it exists a **path** between **every pair** of vertices
- ✓ A graph is **complete** if it exists an **edge** between **every pair** of vertices
- ✓  $G'(V', E')$  is a subgraph of  $G(V, E)$  if  $V' \in V$  and  $E' \in E$
- ✓ A **tree** is a connected acyclic graph
- ✓ A **forest** consists of several trees
- ✓ A graph  $G(V, E)$  is **sparse** if  $|E|$  is much smaller than  $O(|V|^2)$

Weighted graphs:

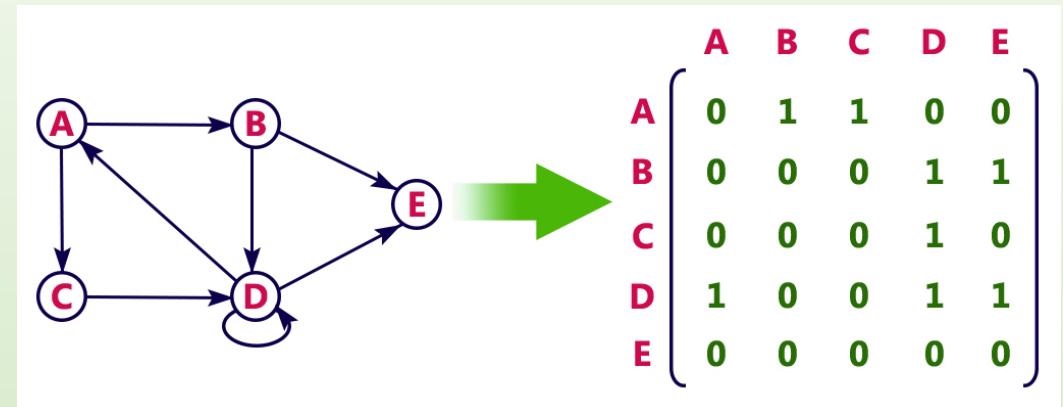
- ✓  $G(V, E, w)$  where  $w$  is a **real valued function** defined on  $E$  (every existing edge has a value)
- ✓ The **weight** of the **graph** is the **sum** of the **weights** of its edges

# Rappresentazione dei grafi

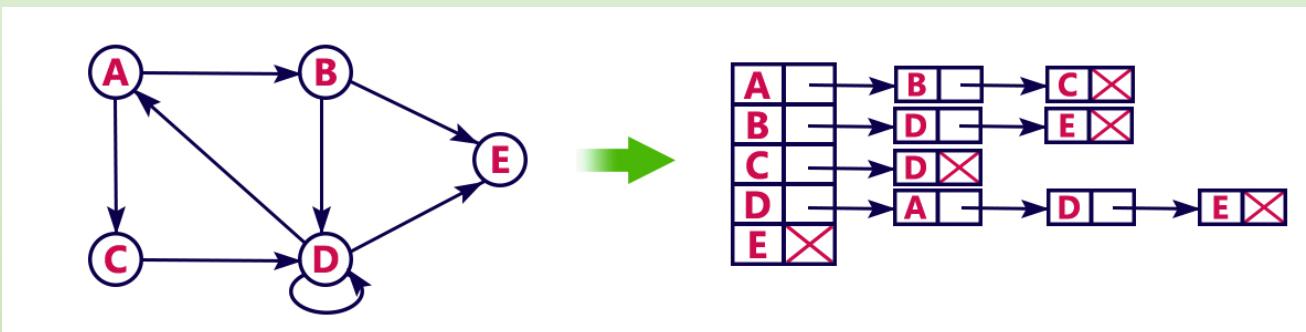
Matrice di adiacenza



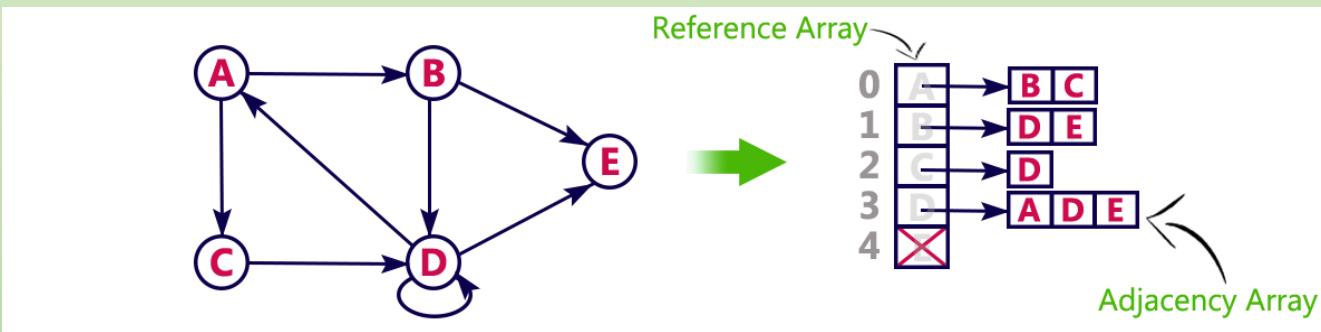
Matrice di incidenza



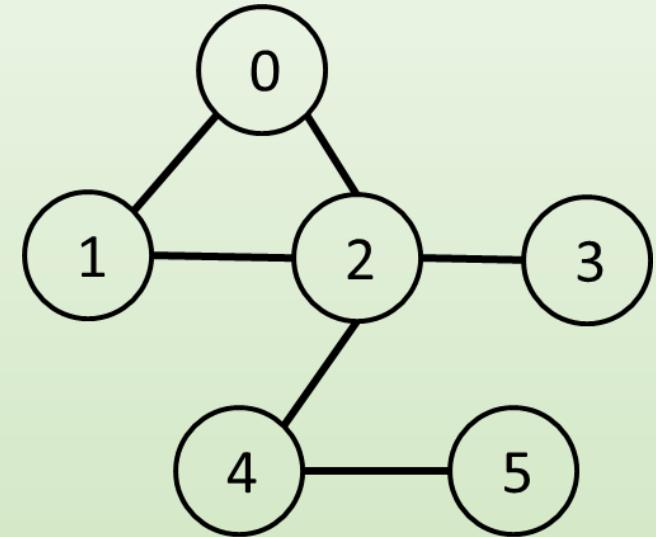
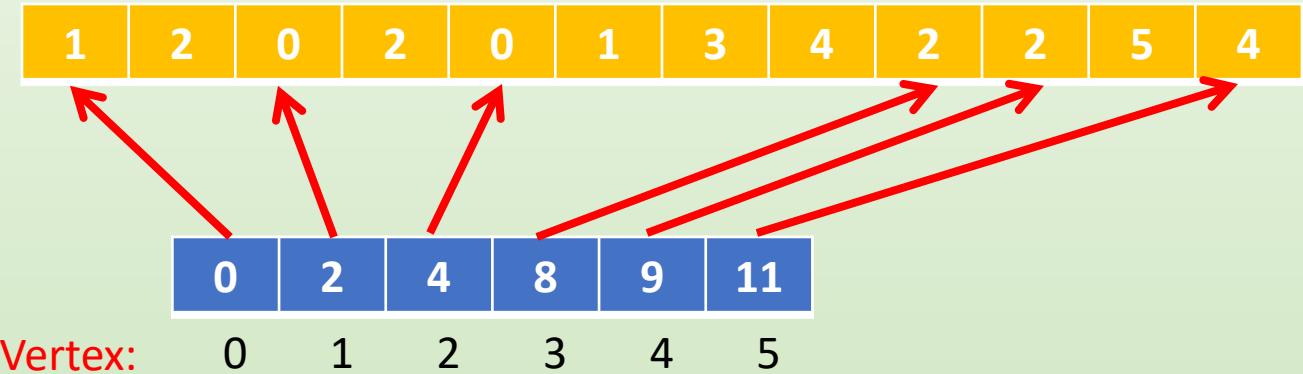
Linked list



ArrayList



# Adjacency List



- Array  $E_a$ : Adjacent vertices to vertex 0, then vertex 1, then ...  
size:  $O(E)$
- Array  $V_a$ : Delimiters for  $E_a$   
size:  $O(V)$

# Struttura di grafo a liste in C++

```
template<typename nodeW, typename edgeW> struct GraphStruct {
    node nNodes{0}; // num of graph nodes
    node_sz nEdges{0}; // num of graph edges
    node_sz* cumulDegs{ nullptr }; // cumsum of node degrees
    node* neights{ nullptr }; // list of neighbors for all nodes (edges)
    nodeW* nodeWeights{ nullptr }; // list of weights for all nodes
    edgeW* edgeWeights{ nullptr }; // list of weights for all edges
    nodeW* nodeThresholds{ nullptr };
    // return the degree of node i
    inline node_sz deg(node i) {
        return ( cumulDegs[i + 1] - cumulDegs[i] );
    }
    // check whether node i is a neighbor of node j
    bool areNeighbor(node i, node j) {
        for (unsigned k = 0; k < deg(j); k++)
            if (neights[cumulDegs[j]+k] == i)
                return true;
        return false;
    }
};
```

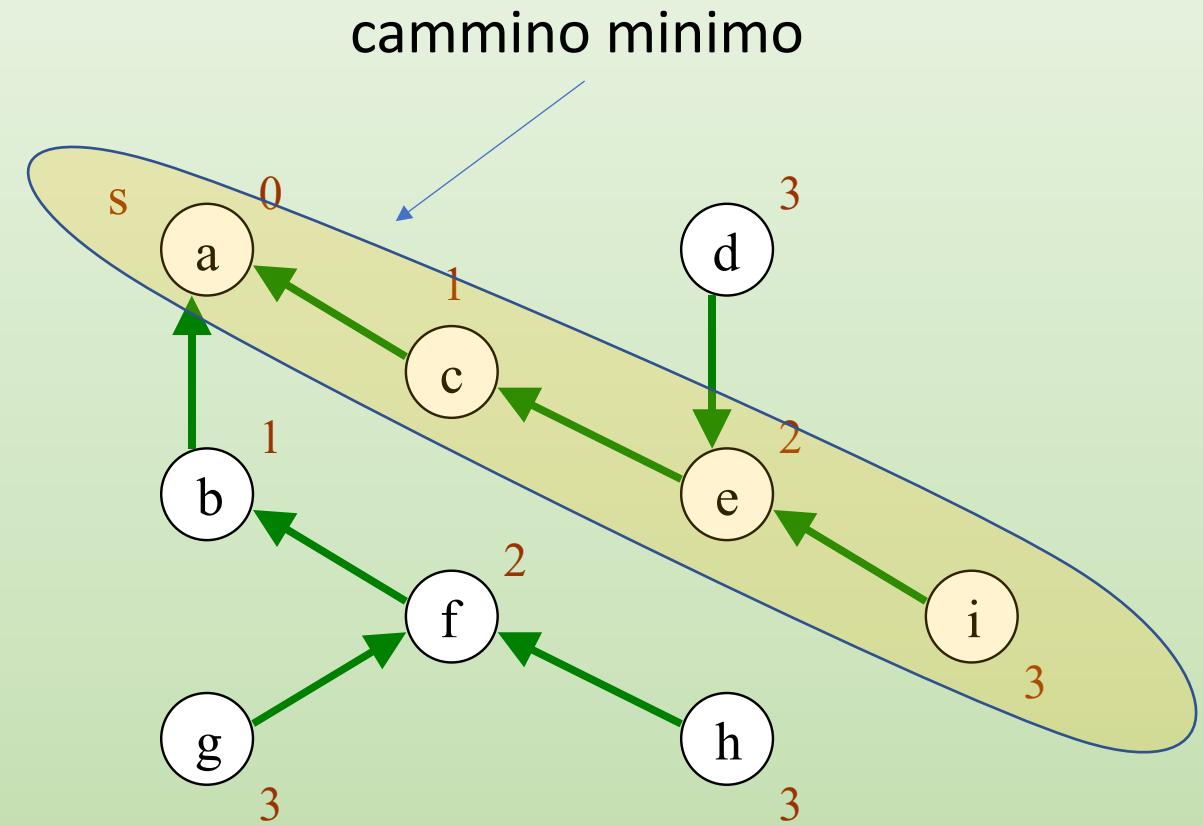
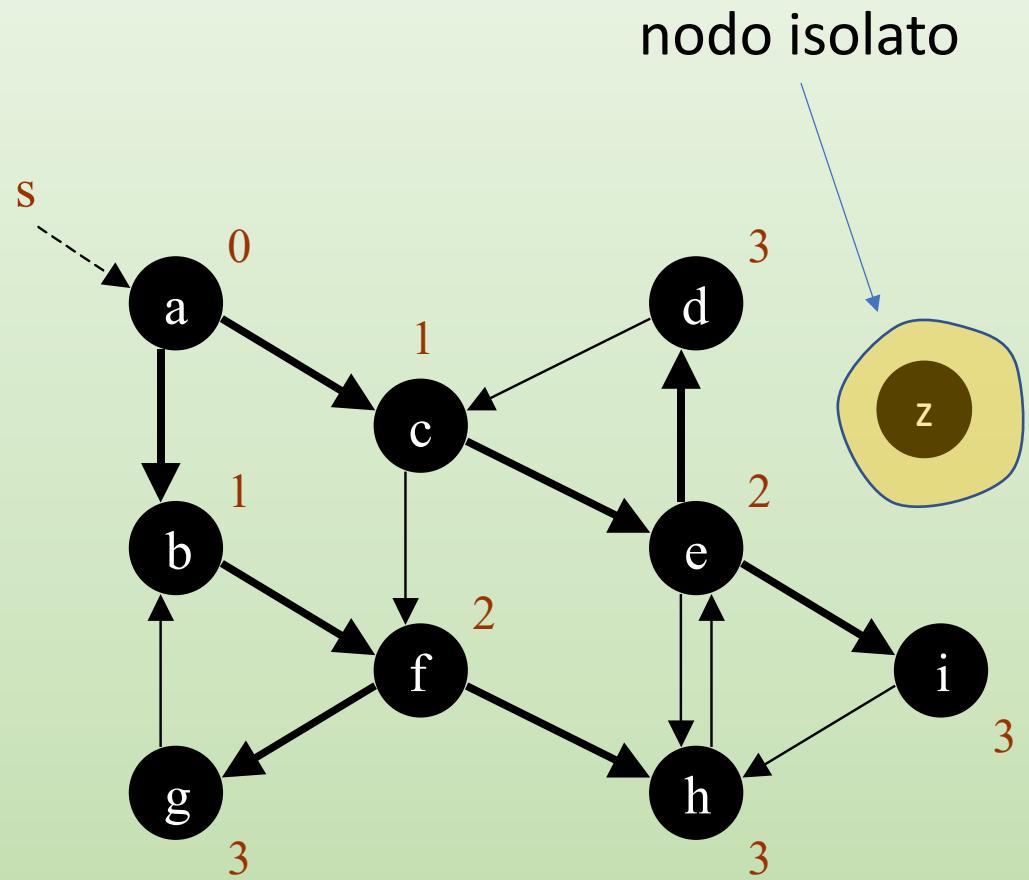
# BFS (Breadth-First Search)

Ricerca in un grafo

# Graph searching: BFS (Breadth-First Search)

- ✓ **Visita in ampiezza:** mostra la raggiungibilità dei nodi ed è la base di altri algoritmi come:
  - **single-source shortest path (SSSP)**
  - algoritmo **Ford-Fulkerson** per **min-cut/max-flow**
- ✓ **Instanza:** Grafo orientato  $\mathbf{G} = (\mathbf{V}, \mathbf{E})$  con qualche rappresentazione come la lista di adiacenze
- ✓ **Obiettivo:** esplorare sistematicamente i nodi di  $\mathbf{G}$  al fine di
  - Scoprire quali nodi sono **raggiungibili** dalla sorgente  $s$
  - Calcolare lo **shortest path** o **distanza** di ogni vertice dalla sorgente  $s$
  - Produrre un **breadth-first tree (BFT)**  $\mathbf{G}_T$  con radice  $s$  che contiene:
    - tutti i **vertici raggiungibili** da  $s$
    - lo **shortest path** da  $v$  a  $s$  (path unico da ogni vertice  $v$  a  $s$  in  $\mathbf{G}_T$ )

# BFT (Breadth-First Tree)

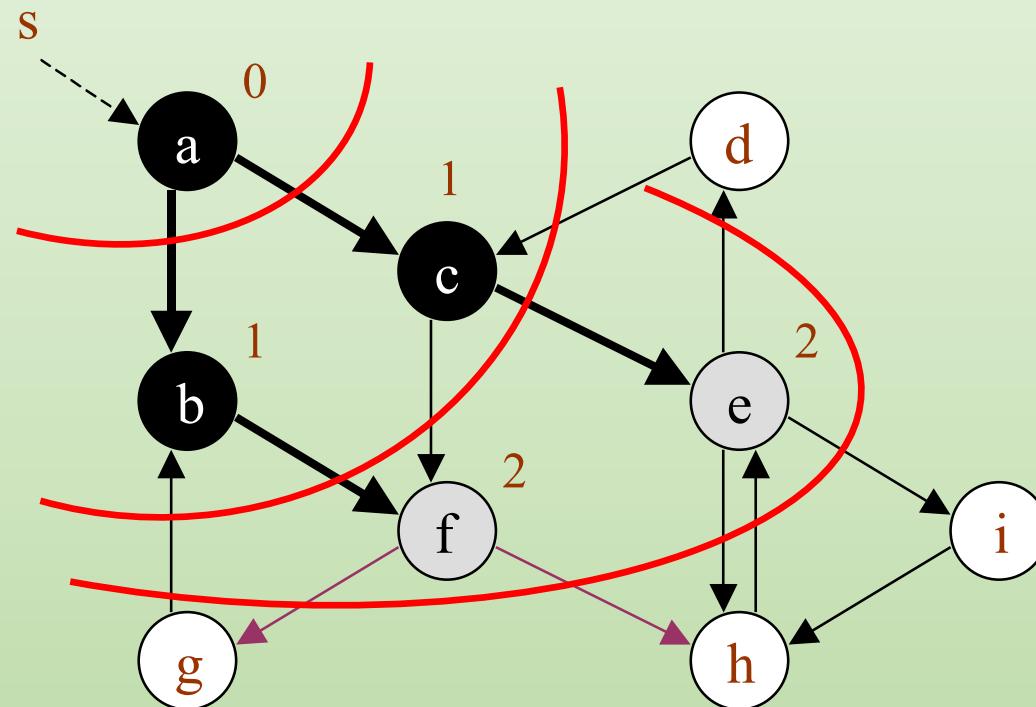


# Graph searching: BFS (Breadth-First Search)

✓ **Idea:** espandere la frontiera in ampiezza (**breadth**) in modo greedy

- propagare una ``onda'' a distanza di **arco 1**, un passo alla volta (vicini, vicini-di-vicini,...)
- Usare una coda FIFO per aggiornare i puntatori a entrambi gli estremi

✓ **Nota:** al contrario di **Depth-First Search (DFS)**, BFS è un buon candidato per la paralizzazione perché il neighborhood di un vertice può essere esplorato in parallelo



# Algoritmo BFS

Strutture dati per ogni  $u \in V$

- $\text{color}[u]$ : color of  $u$ 
  - WHITE: non ancora visitato
  - GRAY: visitato e deve essere processato
  - BLACK: visitato e processato
- $P[u]$ : genitore di  $u$  (NIL se  $u = s$  o  $u$  non ancora visitato)
- $d[u]$ : distanza di  $u$  da  $s$
- $\text{Adj}[u]$ : lista nodi adiacenti di  $u$

Elaborazione di un vertice => scansione della sua lista di adiacenza

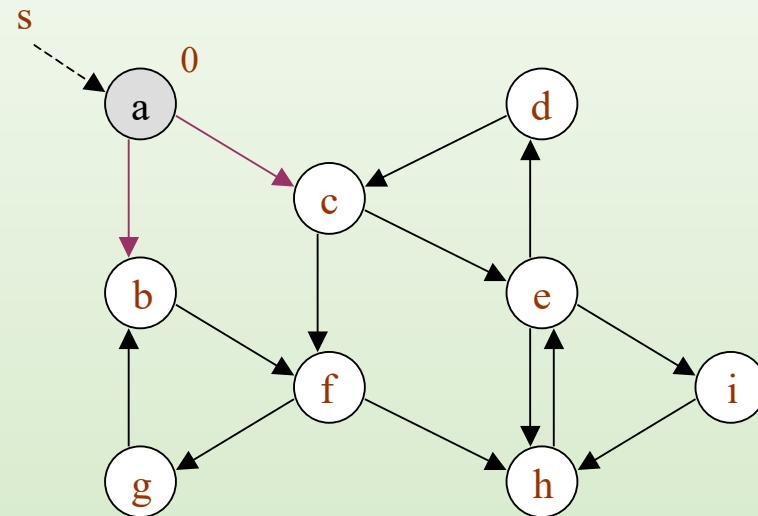
**BFS( $G, s$ )**

```
for each  $u \in V - \{s\}$  do
    color[u]  $\leftarrow$  WHITE
    P[u]  $\leftarrow$  NIL;
    d [u]  $\leftarrow$   $\infty$ 
color[s]  $\leftarrow$  GRAY
P[s]  $\leftarrow$  NIL;
d [s]  $\leftarrow$  0
Q  $\leftarrow$  {s}
while  $Q \neq \emptyset$  do
     $u \leftarrow \text{head}[Q]$ 
    for each  $v$  in  $\text{Adj}[u]$  do
        if color[v] = WHITE then
            color[v]  $\leftarrow$  GRAY
            P[v]  $\leftarrow u$ 
            d [v]  $\leftarrow d [u] + 1$ 
            ENQUEUE(Q, v)
DEQUEUE(Q)
color[u]  $\leftarrow$  BLACK
```

inizializzazione

accodamento  
dei soli  
visitati

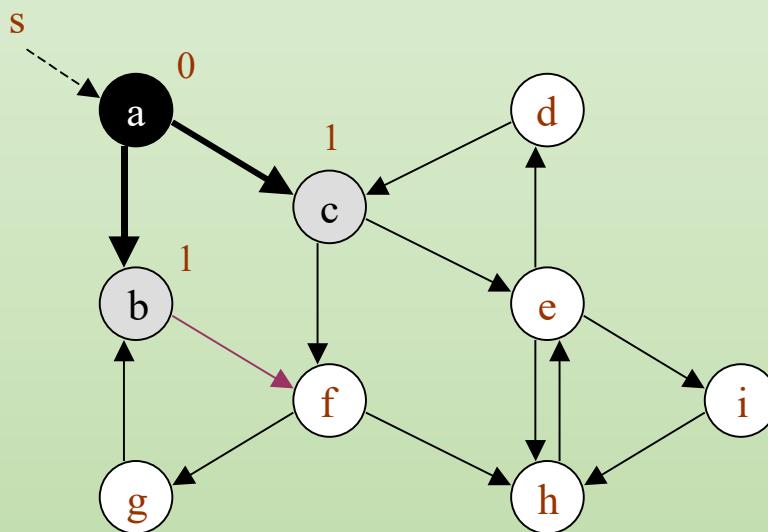
# BFS...



FIFO  
queue  $Q$  just after  
processing vertex

$\langle a \rangle$

-



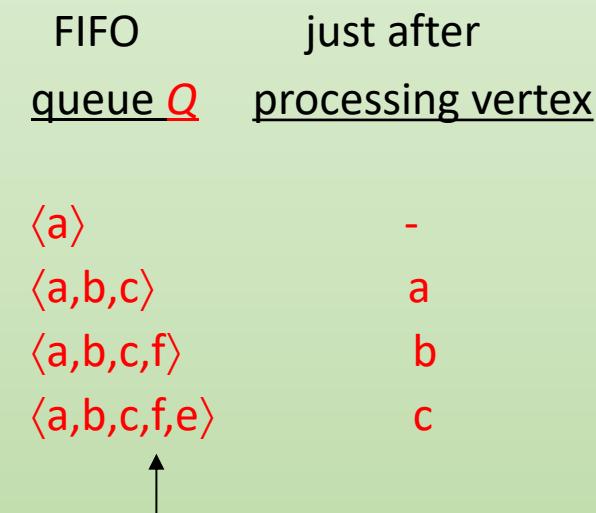
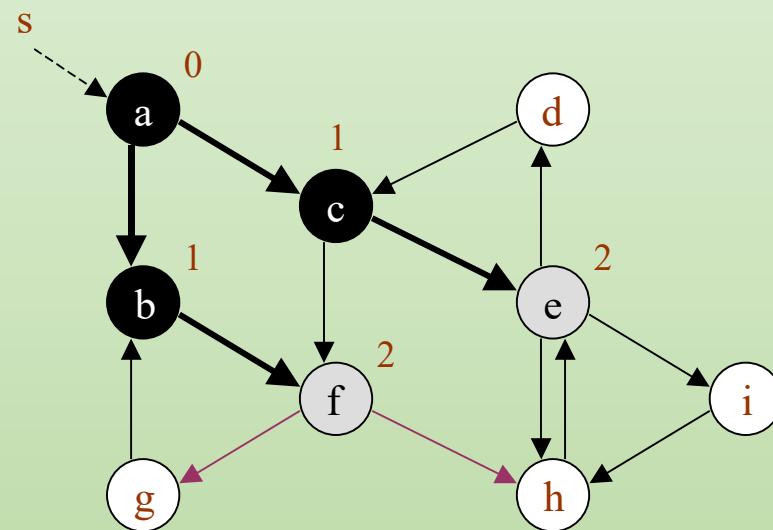
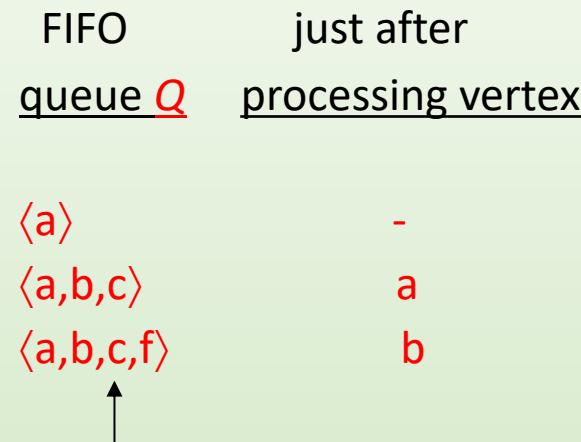
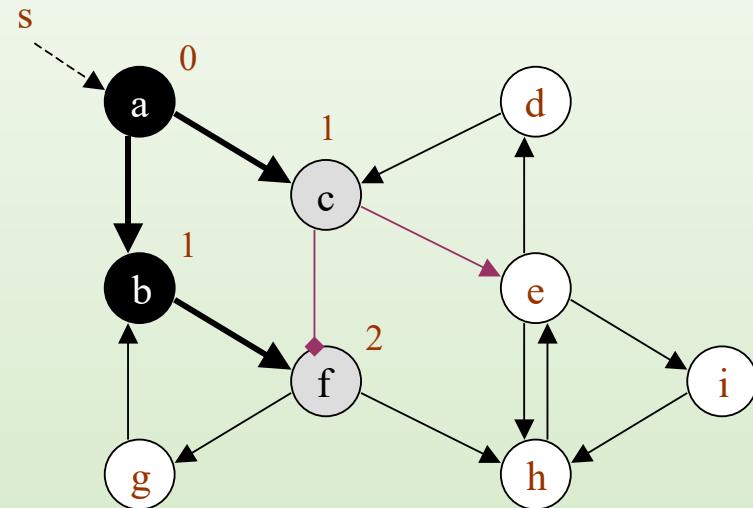
FIFO  
queue  $Q$  just after  
processing vertex

$\langle a \rangle$   
 $\langle a, b, c \rangle$

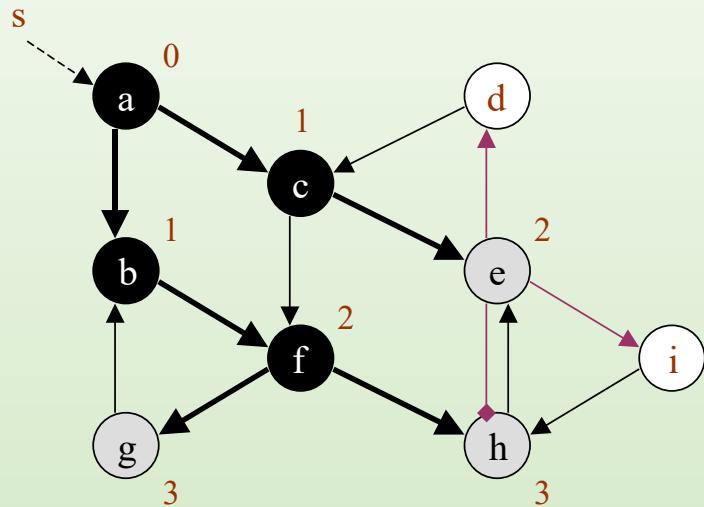
-

a

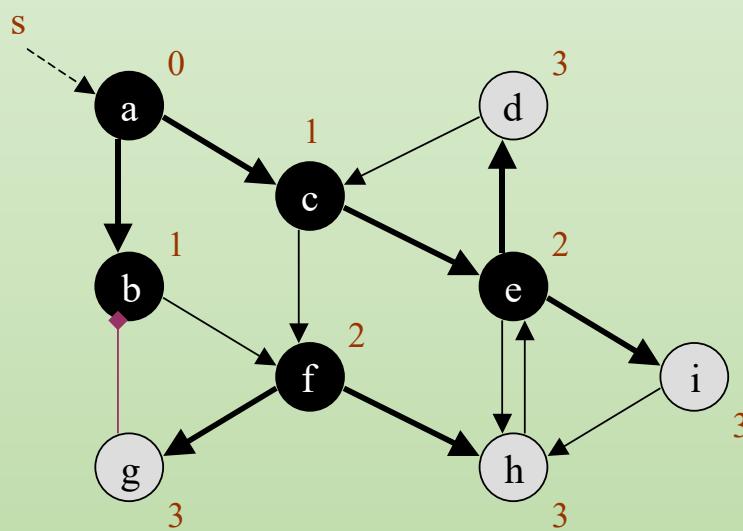
# BFS...



# BFS...

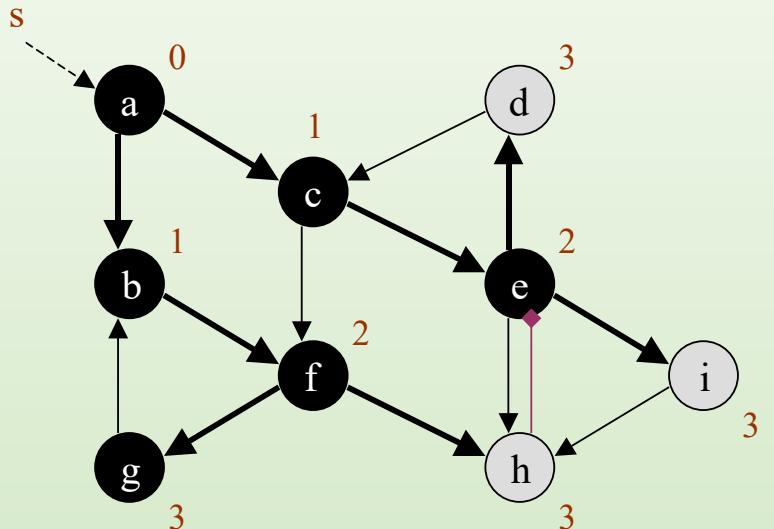


FIFO queue $Q$	just after processing vertex
$\langle a \rangle$	-
$\langle a, b, c \rangle$	a
$\langle a, b, c, f \rangle$	b
$\langle a, b, c, f, e \rangle$	c
$\langle a, b, c, f, e, g, h \rangle$	f



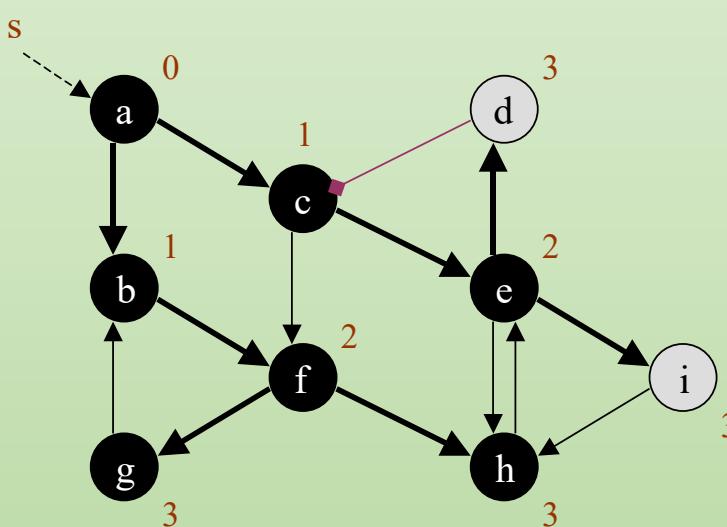
FIFO queue $Q$	just after processing vertex
$\langle a \rangle$	-
$\langle a, b, c \rangle$	a
$\langle a, b, c, f \rangle$	b
$\langle a, b, c, f, e \rangle$	c
$\langle a, b, c, f, e, g, h \rangle$	f
$\langle a, b, c, f, e, g, h, d, i \rangle$	e

# BFS...



FIFO      just after  
queue  $Q$     processing vertex

$\langle a \rangle$	-
$\langle a, b, c \rangle$	a
$\langle a, b, c, f \rangle$	b
$\langle a, b, c, f, e \rangle$	c
$\langle a, b, c, f, e, g, h \rangle$	f
$\langle a, b, c, f, e, g, h, d, i \rangle$	g

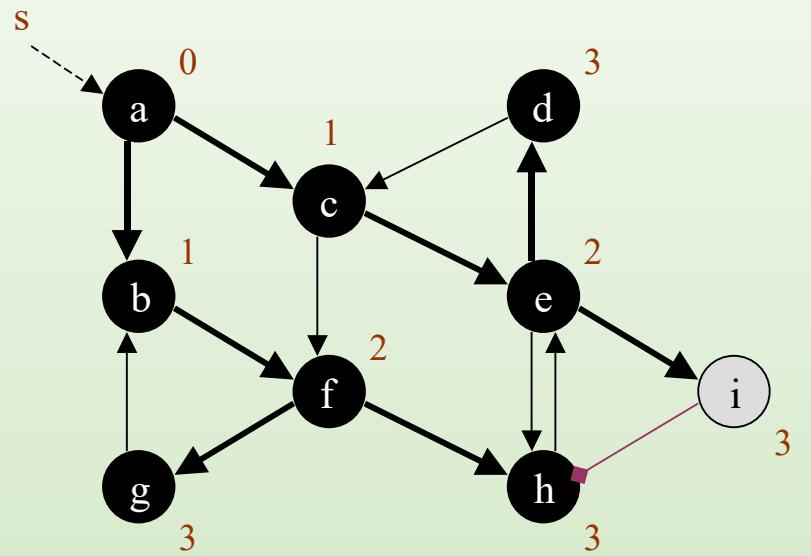


FIFO      just after  
queue  $Q$     processing vertex

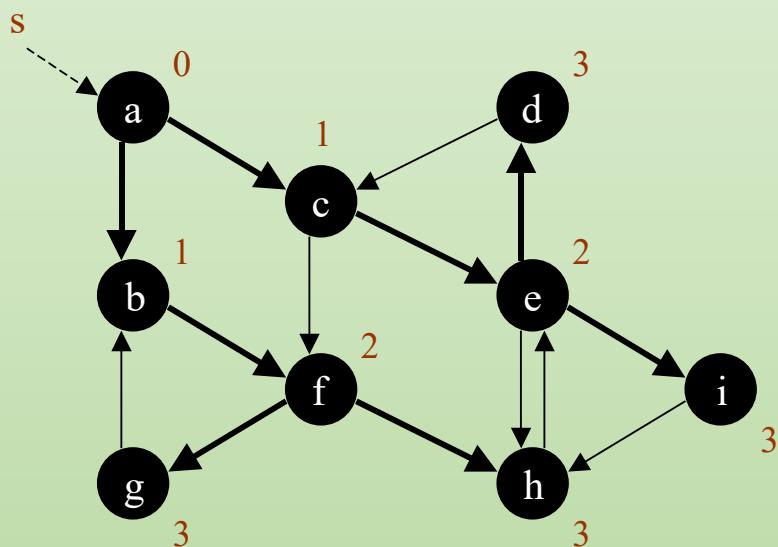
$\langle a \rangle$	-
$\langle a, b, c \rangle$	a
$\langle a, b, c, f \rangle$	b
$\langle a, b, c, f, e \rangle$	c
$\langle a, b, c, f, e, g, h \rangle$	f
$\langle a, b, c, f, e, g, h, d, i \rangle$	h



# BFS...

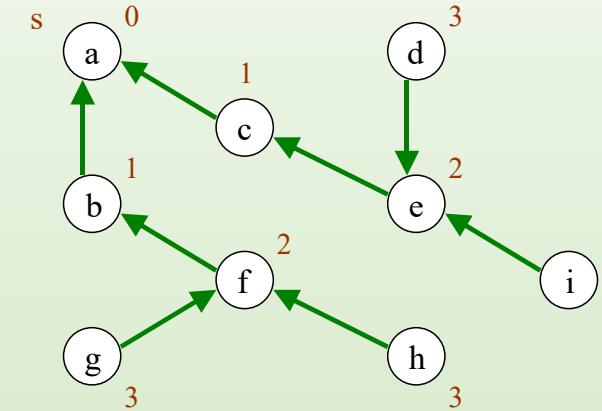
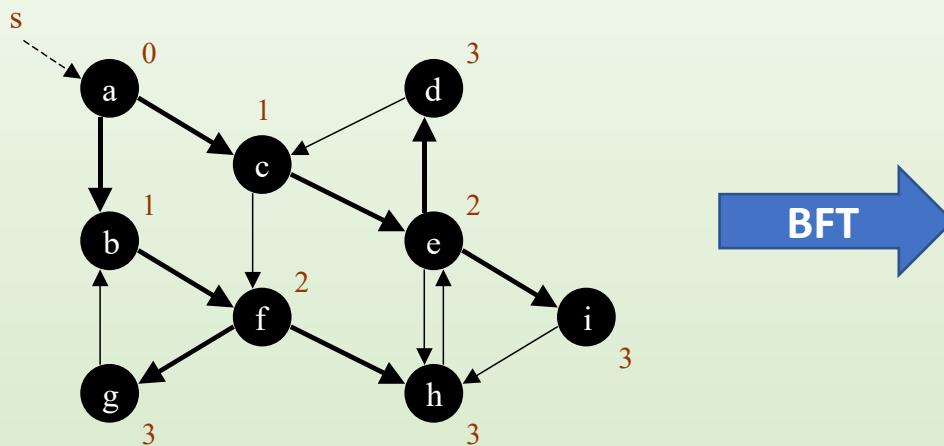


FIFO queue $Q$	just after processing vertex
$\langle a \rangle$	-
$\langle a, b, c \rangle$	a
$\langle a, b, c, f \rangle$	b
$\langle a, b, c, f, e \rangle$	c
$\langle a, b, c, f, e, g, h \rangle$	f
$\langle a, b, c, f, e, g, h, d, i \rangle$	d



FIFO queue $Q$	just after processing vertex
$\langle a \rangle$	-
$\langle a, b, c \rangle$	a
$\langle a, b, c, f \rangle$	b
$\langle a, b, c, f, e \rangle$	c
$\langle a, b, c, f, e, g, h \rangle$	f
$\langle a, b, c, f, e, g, h, d, i \rangle$	i

# BFS complexity



- ✓ **Tempo di esecuzione:**  $O(V+E)$  (considerato tempo lineare per i grafi)
  - **Inizializzazione:**  $\Theta(V)$
  - **Operazioni di accodamento:**  $O(V)$ 
    - Ogni vertice **accodato e rimuovere** almeno una volta
    - Le operazioni di **accodamento e rimozione** si fanno in tempo  $O(1)$
  - Processare i **vertici grigi**:  $O(E)$ 
    - Ogni vertice è elaborato almeno una volta:  $\sum_{u \in V} |Adj[u]| = \Theta(E)$

# Algoritmo parallelo

Strutture dati:

- **neighs**: concatenazione di tutte le liste di adiacenza di tutti i vertici (dim  $2 * |E|$ )
- **cumulDegs**: array di dim  $|V|$  per gli offset nell'array precedente l'i-mo offset fornisce l'inizio della lista di adiacenza dell'i-mo vertice
- **edgeWeights**: array di dim  $|E|$  che memorizza i pesi degli archi per la versione pesata

**BFS\_CUDA( $G, s$ )**

```
for all  $u \in V$  do  
     $d[u] \leftarrow \infty$   
 $d[s] \leftarrow 0$   
running  $\leftarrow \text{true}$   
while running do  
    running  $\leftarrow \text{false}$   
    for all  $u \in V$  in parallel do  
        for all  $v$  in  $\text{Adj}[u]$  do  
            if  $d[v] > d[u] + 1$  then  
                 $d[v] = d[u] + 1$   
running  $\leftarrow \text{true}$ 
```

# Algoritmo BFS semplice

**Accelerating Large Graph Algorithms on the GPU Using CUDA**, Pawan Harish and P.J. Narayanan, LNCS HiPC'07 Proceedings of the 14th international conference on High performance computing, 2007

1. BFS traverses the graph in **levels**, once a level is **visited** it is **not visited again**
2. The BFS **frontier** corresponds to all the **nodes** being processed at the **current level**
3. We do **not maintain** a **queue** for **each vertex** during our BFS execution (additional overheads )
4. For our implementation we give **one thread** to every vertex
5. Two boolean arrays **frontier Fa** and **visited Xa**
6. One array cost **Ca** stores the **minimal number of edges** of each vertex from the **source vertex**

# CUDA BFS

---

**Algorithm 1.** CUDA\_BFS (Graph  $G(V,E)$ , Source Vertex  $S$ )

---

- 1: Create vertex array  $V_a$  from all vertices and edge Array  $E_a$  from all edges in  $G(V,E)$ ,
  - 2: Create frontier array  $F_a$ , visited array  $X_a$  and cost array  $C_a$  of size  $V$ .
  - 3: Initialize  $F_a$ ,  $X_a$  to false and  $C_a$  to  $\infty$
  - 4:  $F_a[S] \leftarrow \text{true}$ ,  $C_a[S] \leftarrow 0$
  - 5: **while**  $F_a$  not Empty **do**
  - 6:   **for** each vertex  $V$  in parallel **do**
  - 7:     Invoke CUDA\_BFS\_KERNEL( $V_a, E_a, F_a, X_a, C_a$ ) on the grid.
  - 8:   **end for**
  - 9: **end while**
-

# BFS Kernel

---

**Algorithm 2.** CUDA\_BFS KERNEL ( $V_a, E_a, F_a, X_a, C_a$ )

---

```
1:  $tid \leftarrow \text{getThreadID}$ 
2: if  $F_a[tid]$  then
3:    $F_a[tid] \leftarrow \text{false}$ ,  $X_a[tid] \leftarrow \text{true}$ 
4:   for all neighbors  $nid$  of  $tid$  do
5:     if NOT  $X_a[nid]$  then
6:        $C_a[nid] \leftarrow C_a[tid]+1$ 
7:        $F_a[nid] \leftarrow \text{true}$ 
8:     end if
9:   end for
10: end if
```

---

# Graph coloring

Clustering of independent tasks

# Colorazione di grafi: Luby coloring

## Algoritmo greedy sequenziale

```
n = |V|
Choose a random permutation  $p(1), \dots, p(n)$  of numbers  $1, \dots, n$ 
U := V
for  $i = 1$  to  $n$  do in sequence
     $v := p(i)$ 
     $S := \{\text{colors of all colored neighbors of } v\}$ 
     $c(v) := \text{smallest color not in } S$ 
     $U := U - \{v\}$ 
end do
```

Vedi articolo:  
*A Comparison of Parallel Graph Coloring Algorithms*

## Algoritmo greedy parallelo

```
U := V
while ( $|U| > 0$ ) do in parallel
    Choose an independent set  $I$  from  $U$ 
    Color all vertices in  $I$ 
     $U := U - I$ 
end do
```

## Algoritmo greedy x independent set

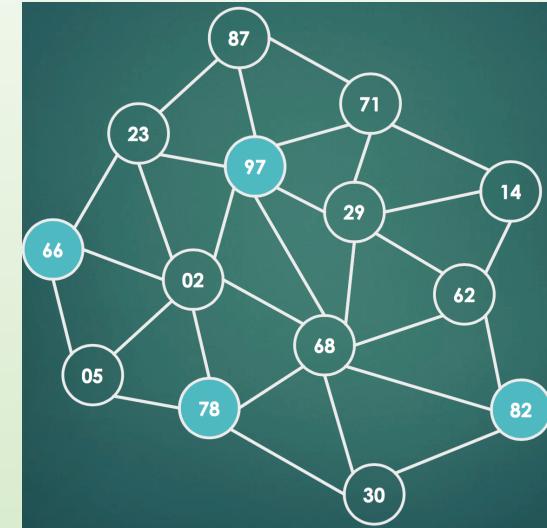
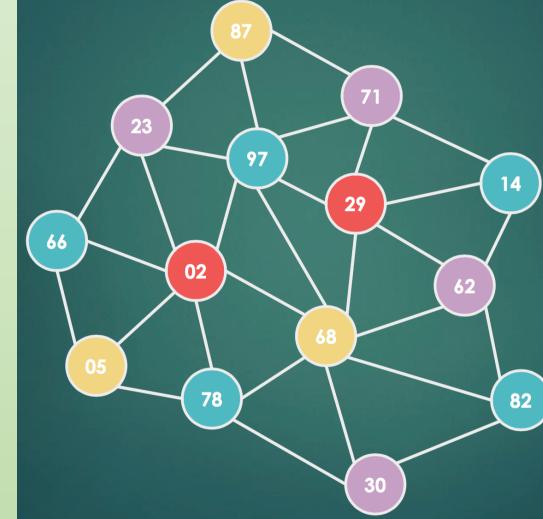
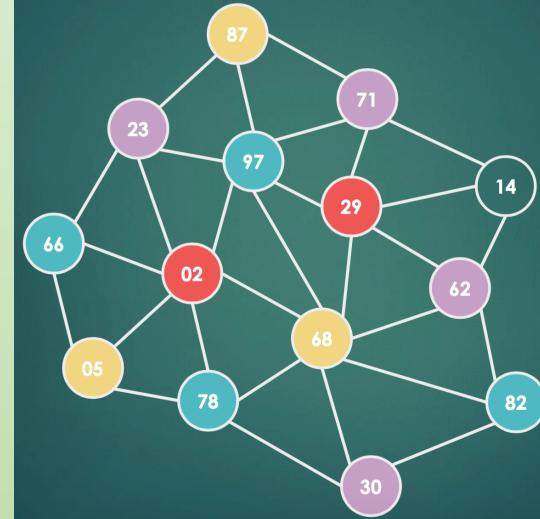
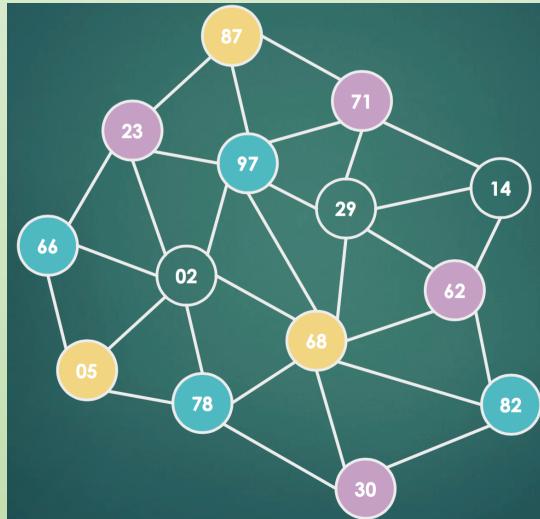
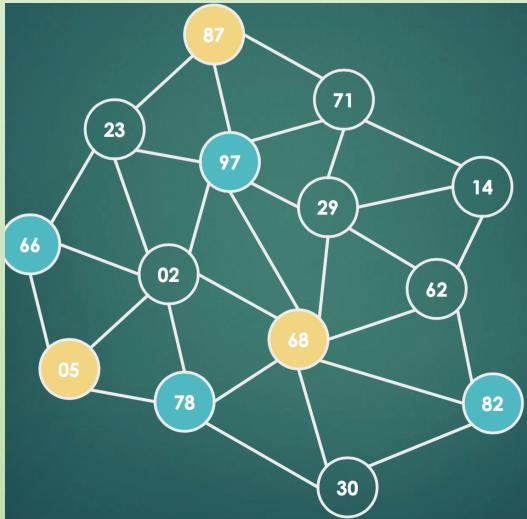
```
I := {}
V' := V
while ( $|V'| > 0$ ) do
    Choose an independent set  $I'$  from  $V'$ 
     $I := I + I'$ 
     $X := I' + N(I')$ 
     $V' := V' - X$ 
end do
```

# Jones-Plassmann Coloring

1. **Assegna** un numero a caso a tutti i vertici
2. **Scegli** localmente quelli di valore massimo
3. **Colora** e itera sui nodi non ancora colorati

**NOTA:** non sono un MIS

```
 $U := V$ 
while ( $|U| > 0$ ) do
    for all vertices  $v \in U$  do in parallel
         $I := \{v \text{ such that } w(v) > w(u) \forall \text{ neighbors } u \in U\}$ 
        for all vertices  $v' \in I$  do in parallel
             $S := \{\text{colors of all neighbors of } v'\}$ 
             $c(v') := \text{minimum color not in } S$ 
        end do
    end do
     $U := U - I$ 
end do
```



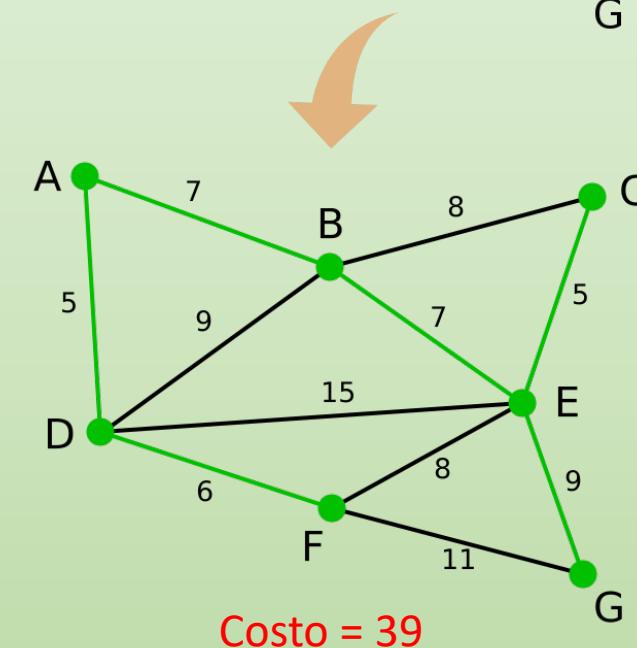
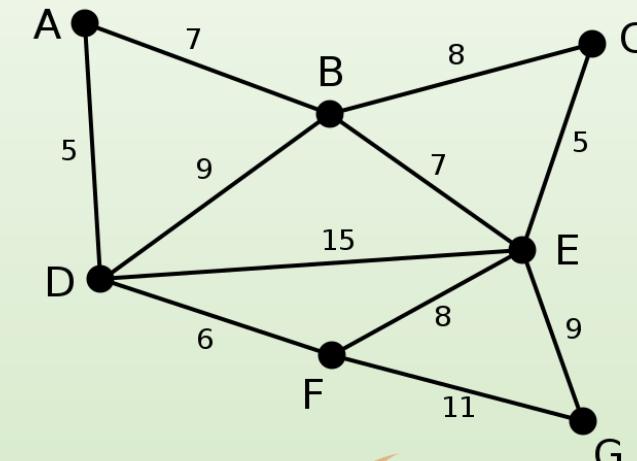
# Minimum Spanning Tree (MST)

Backbone del grafo

# Minimum Spanning Tree (MST)

- ✓ Problemi relativi ad **alberi di supporto minimo** hanno numerose applicazioni:
  - progettazione di reti (comunicazione, teleriscaldamento,...)
  - memorizzazione compatta di sequenze (DNA)
  - diffusione di messaggi segreti...
- ✓ Sono basati su un processo di colorazione che mantiene un opportuno invarianto
- ✓ A seconda del processo di colorazione si otterranno i seguenti algoritmi **greedy**
  - **Boruvka** (1926), **Kruskal** (1956), **Prim** (1957)
- ✓ **Problema:** dato un grafo  $G(V, E, w)$  con una funzione di costo sugli archi, determinare un albero di supporto di costo minimo tra tutti i possibili alberi di supporto  $X$ , cioè tale che:

$$T' = \min_{T \in X} \sum_{e \in T} w(e)$$



# Algoritmo di Borůvka per MST

**INPUT:** un grafo  $\mathbf{G} = \langle V, E \rangle$

- inizializza la foresta come in Passo 1
- finché la dimensione della foresta  $|F| > 1$ 
  - per ogni  $T \in F$ :
    - poni insieme di edge  $S_T = \emptyset$
    - per ogni nodo  $v \in T$ :
      - trova min edge  $(v, u), u \notin T$
      - unisci l'edge  $S_T = S_T \cup \{(v, u)\}$
  - aggrega gli alberi in  $F$  connessi da un edge in  $S_T$

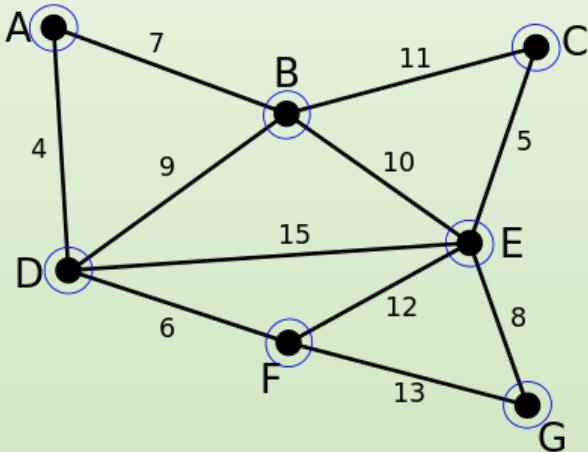
**OUTPUT:**  $T$  è il MST

# Algoritmo di Borůvka: passi

## PASSO 1:

Definisci una foresta di alberi singoletto

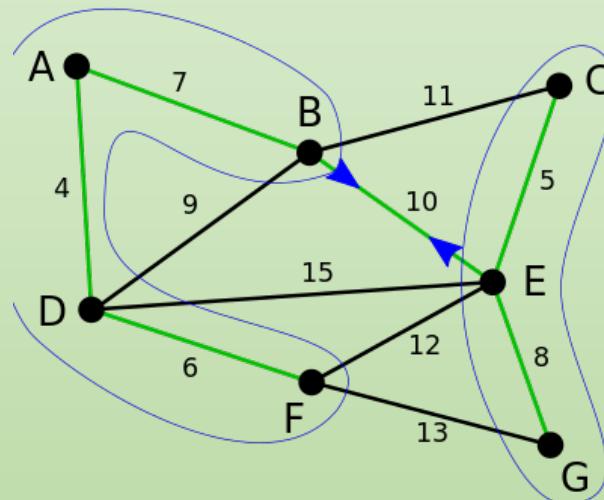
$$F = \{ [A], [B], [C], [D], [E], [F], [G] \}$$



## PASSO 2:

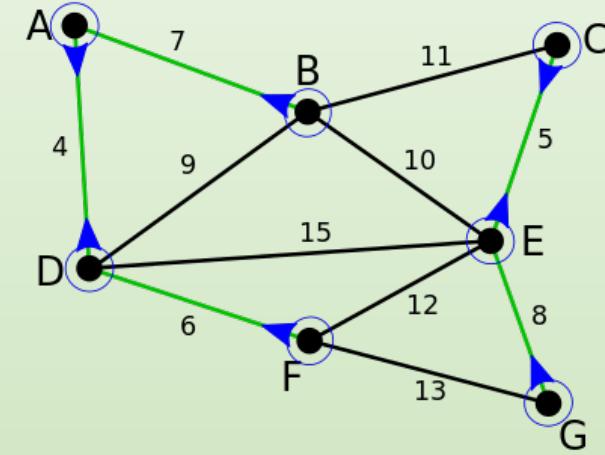
Forestà con due alberi

$$F = \{ [A, B, D, F], [C, E, G] \}$$

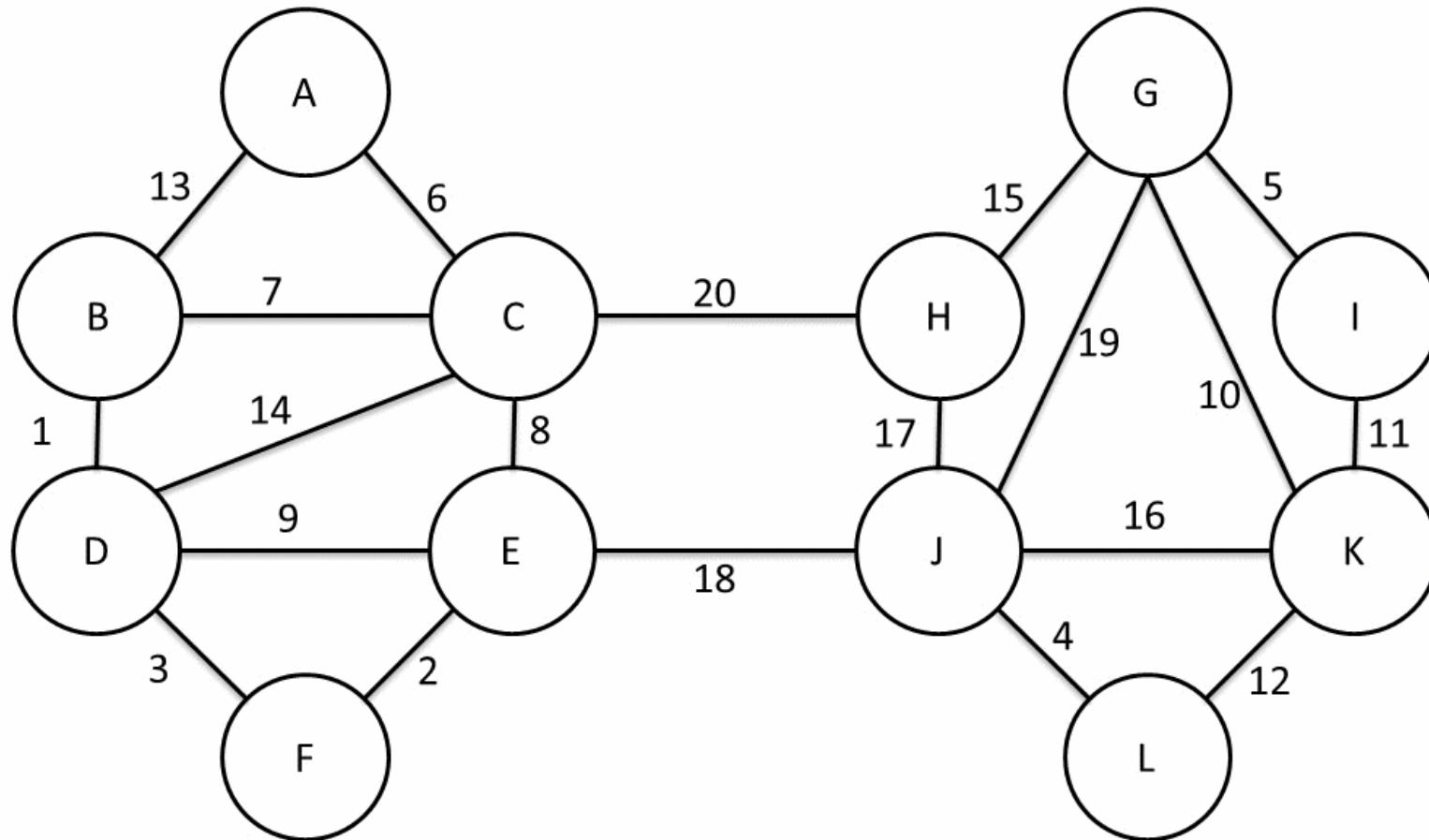


## PASSO 3:

$$\text{MST } T = \{ [A, B, C, D, E, F, G] \}$$



# Animazione...



# Esercitazione

## Algoritmi su grafi:

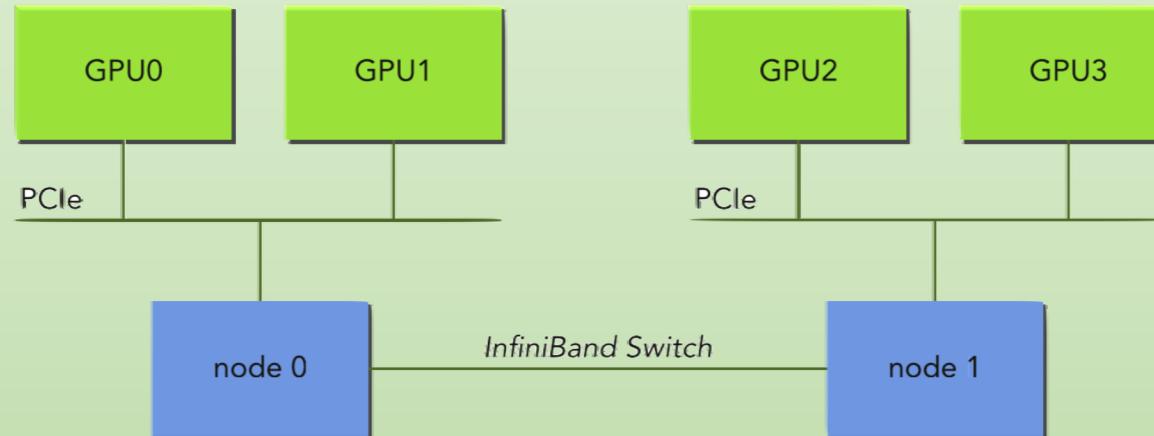
Disegnare un algoritmo distribuito su tre GPU la colorazione di grafo o MST:

1. Sviluppare secondo il metodo di Luby e conseguente clacolo di
2. Oppure considerare la sua versione semplificata secondo Jones in cui non si ha il vincolo di estrarre MST dal grafo
3. Studiare la tecnica di Boruvka e suo split sulle GPU del laboratorio seguendo lo schema P2P

# Multi-GPU programming

- ✓ La programmazione multi-GPU fornisce un ulteriore livello di parallelismo
- ✓ Permette di gestire ed **eseguire** kernel su GPU multiple tra loro interconnesse
- ✓ **Sovrapporre** computazioni e comunicazioni sulle diverse GPU
- ✓ **Sincronizzare** le esecuzioni sulle varie GPU usando stream ed eventi
- ✓ **Scalare** le applicazioni su cluster accelerati da GPU
- ✓ **Importante:** progettare correttamente la comunicazione inter-GPU!

Multiple GPUs connected over the PCIe bus in a single node



Multiple GPUs connected over the PCIe bus in a single node

Multiple GPUs connected over a network switch in a cluster

# API for device query

- ✓ Ogni thread che esegue sull'host può accedere a diversi device

Prototipo-> `cudaError_t cudaGetDeviceCount(int* count);`

- Determina il numero di CUDA-enabled device: `ngpus`

- ✓ Per determinare le proprietà del device

Prototipo-> `cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp *prop, int device);`

- Utile per le info sulla capability o particolari hw del device

```
int ngpus;
cudaGetDeviceCount(&ngpus);
for (int i = 0; i < ngpus; i++) {
    cudaDeviceProp devProp;
    cudaGetDeviceProperties(&devProp, i);
    printf("Device %d has compute capability %d.%d.\n", i, devProp.major,
    devProp.minor);
}
```

Estrarre le  
info sui  
device  
presenti nel  
nodo

# Funzione cudaSetDevice

- ✓ Per fissare quale GPU è quella destinata ad eseguire le operazioni lanciate dalla CPU (non effettua sincronizzazioni)

**Segnatura->** `cudaError_t cudaSetDevice(int id);`

- Definisce il device corrente: da **0** a **ngpus-1**

- ✓ La **GPU** corrente può essere **cambiata** anche mentre chiamate **async** (kernels, memcopies) sono in esecuzione
- ✓ E' anche possibile accodare un **gruppo** di **chiamate async** a una GPU e poi passare ad un'altra GPU
- ✓ Il codice a lato viene eseguito **concorrentemente** su entrambe le GPU
- ✓ Anche **stream** ed **eventi** creati dallo stesso CPU thread vengono eseguiti su quel device

```
cudaSetDevice( 0 );
kernel<<<...>>>( ... );
cudaMemcpyAsync( ... );
cudaSetDevice( 1 );
kernel<<<...>>>( ... );
```

# Esecuzioni da host

- ✓ Una volta scelto il device tutte le operazioni CUDA sono indirizzate a quel device:
  - Allocazioni di **memoria** dall'host saranno **fisicamente residenti sul device**
  - Allocazioni di **memoria** a **runtime** avranno lo stesso **lifetime** del device
  - Gli **stream** o gli **eventi** creati da thread dell'host saranno indirizzati al device
  - I **kernel** lanciati dall'host saranno eseguiti sul device
- ✓ Per eseguire (in modo **asincrono**) codice di un **kernel** e **trasferimenti** in memoria dal singolo thread di CPU alle GPU disponibili sul nodo (il controllo è restituito al thread di CPU dopo il lancio)

Non richiede  
sincronizzazione tra  
device (veloce!)

```
for (int i = 0; i < ngpus; i++) {  
  
    // set the current device  
    cudaSetDevice(i);  
  
    // execute kernel on current device  
    kernel<<<grid, block>>>(...);  
  
    // asynchronously transfer data between the host and current device  
    cudaMemcpyAsync(...);  
}
```

# Comunicazione peer-to-peer

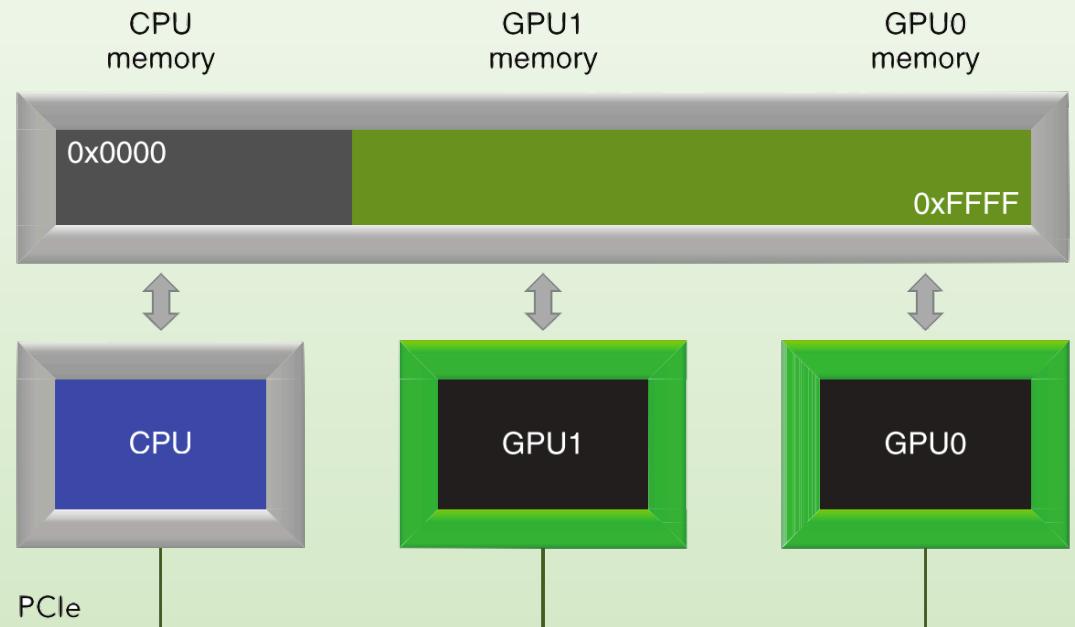
- ✓ I **kernel** che eseguono applicazioni a **64-bit** su device con **compute capability** almeno **2.0** possono accedere alla global memory di ogni GPU connessa sullo **stesso nodo** PCIe
- ✓ Dalle **CUDA API 4.0** in poi mediante la comunicazione **peer-to-peer (P2P)** è possibile instaurare la comunicazione inter-device
- ✓ Due sono le modalità di interazione P2P:
  - **Accesso Peer-to-peer:** consente alle GPU connesse allo stesso nodo PCIe di referenziare dati memorizzati su device memory di altre GPU (in maniera trasparente al kernel i dati vengono trasferiti via bus PCIe al thread richiedente)
  - **Trasferimento Peer-to-peer:** consente di copiare i dati direttamente tra GPU (se sullo stesso nodo) altrimenti la copia passa attraverso la memoria host (caso di GPU appartenenti a nodi diversi del cluster)
- ✓ L'allocazione di dati tra CPU e GPU usa **UVA (Unified Virtual Address space)**
  - ogni (CPU, GPU) gestisce il suo proprio spazio indirizzamento
  - È compito del driver/device determinare gli indirizzi ove i dati risiedono
  - Una data allocazione risiede su un solo device (un array non può essere ripartito tra diverse GPUs)

# Unified Virtual Addressing (UVA)

- ✓ Le allocazioni in host memory fatte via **cudaHostAlloc** e quelle fatte in device memory via **cudaMalloc** risiedono in uno spazio di indirizzamento virtuale unificato

- ✓ Il **device** su cui un dato risiede può essere determinato dall'indirizzo stesso

- ✓ Le applicazioni che usano **UVA** devono essere compilate su **arch a 64-bit**
- ✓ con device di **compute capability >= 2.0** e **CUDA >= 4.0**
- ✓ Se l'accesso **peer-to-peer** e **UVA** sono abilitati i **kernel** eseguiti su un device possono **dereferenziare** un **puntatore** alla memoria di un **altro device**



# Uso pratico di UVA

- ✓ UVA consente di **semplificare la gestione** separate di spazi di **memoria** host e device
- ✓ Rendere i **programmi CUDA più leggibili** e semplici da mantenere
- ✓ Per questo occorre:
  - Rimpiazzare le allocazioni host e device con memoria gestita dal device driver (automatica)
  - Eliminare la duplicazione dei puntatori
  - Rimuovere tutte le copie di memoria esplicite

Dichiarazione e allocazione  
mediante API con suffisso  
'Managed'

E' utile sincronizzare  
esplicitamente host e device  
per accedere dopo output  
del kernel

```
// allocazione di 3 array managed
float *A, *B, *gpuRef;
cudaMallocManaged((void **)&A, nBytes);
cudaMallocManaged((void **)&B, nBytes);
cudaMallocManaged((void *)&gpuRef, nBytes);
. . .
initialData(A, nxy); // inizializzazione
initialData(B, nxy); // inizializzazione
```

```
// lancio del kernel
sumMatrixGPU<<<grid, block>>>(A, B, gpuRef, nx, ny);
cudaDeviceSynchronize();
```

# Abilitare/disabilitare P2P

- ✓ Controllo esplicito se un dato device supporta il P2P ad opera di un altro device che vuole comunicare con lui

Prototipo-> `cudaError_t cudaDeviceCanAccessPeer(int* canAccessPeer, int device, int peerDevice);`

- Restituisce 1 in `canAccessPeer` se `device` è in grado di accedere alla global memory di `peerDevice`; altrimenti 0

- ✓ L'abilitazione dell'accesso P2P deve essere esplicitata ed è unidirezionale

Prototipo-> `cudaError_t cudaDeviceEnablePeerAccess(int peerDevice, unsigned int flag);`

- Abilita l'accesso peer-to-peer dal device corrente al device `peerDevice` (la flag al momento non è definita e posta a 0)

- ✓ P2P rimane abilitato finché non viene esplicitamente disabilitata

Prototipo-> `cudaError_t cudaDeviceDisablePeerAccess(int peerDevice);`

- Solo per architetture a 64-bit

# Sincronizzazione su GPUs

- ✓ Si possono usare le stesse funzioni di **sincronizzazione** viste in una singola GPU basate su **stream** a ed **eventi** in applicazioni multi-GPU, a patto di specificare il device corrente...
- ✓ Il tipico **workflow** è il seguente:
  1. **Seleziona** l'insieme di GPU che l'applicazione userà
  2. **Crea** stream ed eventi per ogni device
  3. **Allocata** le risorse su ogni device come la device memory
  4. **Lancia** i task su ogni GPU attraverso gli stream (trasferimento dati o esecuzione di kernel)
  5. **Usa** gli stream ed eventi per interrogare o sincronizzare il completamento dei task
  6. **Libera** le risorse per tutti i device
- ✓ Si può aggregare un kernel a uno stream o associare un evento ad uno stream solo se il device associato allo stream è il device corrente

# Ripartire la somma di numeri su GPUs...

Ripartizione dei dati  
tra le diverse GPU

Es. con 16M elem.  
Ripartizione equa  
dei dati

Pinned memory per  
il trasferimento  
asincrono

trasferimento  
asincrono host-  
device

Creazione stream,  
uno per ogni  
device

```
float *d_A[NGPUS], *d_B[NGPUS], *d_C[NGPUS];
float *h_A[NGPUS], *h_B[NGPUS], *hostRef[NGPUS], *gpuRef[NGPUS];
cudaStream_t stream[NGPUS];

...
int size = 1 << 24;
int iSize = size / ngpus;
size_t iBytes = iSize * sizeof(float);

for (int i = 0; i < ngpus; i++) {
    // set current device
    cudaSetDevice(i);
    // allocate device memory
    cudaMalloc((void **) &d_A[i], iBytes);
    cudaMalloc((void **) &d_B[i], iBytes);
    cudaMalloc((void **) &d_C[i], iBytes);
    // allocate page locked host memory for
    // asynchronous data transfer

    cudaMallocHost((void **) &h_A[i], iBytes);
    cudaMallocHost((void **) &h_B[i], iBytes);
    cudaMallocHost((void **) &hostRef[i], iBytes);
    cudaMallocHost((void **) &gpuRef[i], iBytes);

    // create streams for timing and synchronizing
    cudaStreamCreate(&stream[i]);
}
```

# ...suddividere sulle GPU...

Set device

```
// initialize host arrays  
for (int i = 0; i < ngpus; i++) {  
    cudaSetDevice(i);  
    initialData(h_A[i], iSize);  
    initialData(h_B[i], iSize);  
}
```

Set dati in host pinned  
memory

Set device

```
// distributing the workload across multiple devices  
for (int i = 0; i < ngpus; i++) {  
    cudaSetDevice(i);  
    cudaMemcpyAsync(d_A[i], h_A[i], iBytes, cudaMemcpyHostToDevice, stream[i]);  
    cudaMemcpyAsync(d_B[i], h_B[i], iBytes, cudaMemcpyHostToDevice, stream[i]);  
    iKernel<<<grid, block, 0, stream[i]>>> (d_A[i], d_B[i], d_C[i], iSize);  
    cudaMemcpyAsync(gpuRef[i], d_C[i], iBytes, cudaMemcpyDeviceToHost, stream[i]);  
}
```

Copia dei dati su  
stream = device

Lancio kernel su  
stream = device

# ...suddividere sulle GPU

Set device

Free device data

Free host data

Free stream resource

```
// Cleanup and shutdown
for (int i = 0; i < ngpus; i++)    {
    cudaSetDevice(i);
    cudaFree(d_A[i]);
    cudaFree(d_B[i]);
    cudaFree(d_C[i]);

    cudaFreeHost(h_A[i]);
    cudaFreeHost(h_B[i]);
    cudaFreeHost(hostRef[i]);
    cudaFreeHost(gpuRef[i]);

    cudaStreamDestroy(stream[i]);
}

cudaDeviceReset();
```

# nvprof

```
$ nvprof --print-gpu-trace ./kernels_on_MultiGPU
```

```
3 GPU timer elapsed: 29.69ms
==5350== Profiling application: Ese10-multi-GPU
==5350== Profiling result:
```

Start	Duration	Size	Throughput	Device	Context	Stream	Name
6.82334s	7.1027ms	22.370MB	3.1495GB/s	Tesla M2090 (0)	1	13	[CUDA memcpy HtoD]
6.82350s	7.2728ms	22.370MB	3.0758GB/s	Tesla M2050 (1)	2	21	[CUDA memcpy HtoD]
6.82358s	7.2102ms	22.370MB	3.1025GB/s	Tesla M2090 (2)	3	29	[CUDA memcpy HtoD]
6.83045s	7.3172ms	22.370MB	3.0571GB/s	Tesla M2090 (0)	1	13	[CUDA memcpy HtoD]
6.83077s	7.2244ms	22.370MB	3.0964GB/s	Tesla M2050 (1)	2	21	[CUDA memcpy HtoD]
6.83079s	7.1761ms	22.370MB	3.1173GB/s	Tesla M2090 (2)	3	29	[CUDA memcpy HtoD]
6.83832s	14.288ms	22.370MB	1.5657GB/s	Tesla M2090 (0)	1	13	[CUDA memcpy DtoH]
6.83853s	14.379ms	22.370MB	1.5557GB/s	Tesla M2090 (2)	3	29	[CUDA memcpy DtoH]
6.83866s	7.9121ms	22.370MB	2.8273GB/s	Tesla M2050 (1)	2	21	[CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.

SSMem: Static shared memory allocated per CUDA block.

DSMem: Dynamic shared memory allocated per CUDA block.

Start	Duration	Grid Size	Block Size	Regs*	SSMem*	DSMem*	Device	Context	Stream	Name	
6.83777s	544.50us	(10923 1 1)	(512 1 1)	8	0B	0B	Tesla M2090 (0)	1	13	kernel(float*, float*, float*, int) [303]	
6.83797s	543.96us	(10923 1 1)	(512 1 1)	8	0B	0B	Tesla M2090 (2)	3	29	kernel(float*, float*, float*, int) [323]	
6.83800s	652.82us	(10923 1 1)	(512 1 1)	8	0B	0B	Tesla M2050 (1)	2	21	kernel(float*, float*, float*, int) [313]	

# Comunicazione P2P su GPU multiple

**Trasferimento dati tra 2 GPU... tre casi:**

1. Copia di memoria **unidirezionale** tra le 2 GPU
2. Copia di memoria **bidirezionale** tra le 2 GPU
3. **Accesso** alla memoria di un device da parte di un kernel

# Abilitare il P2P: schema

Verifica se P2P è possibile tra la GPU i e la GPU j

Abilita la copia dei dati P2P tra la GPU i e la GPU j (il flag 0 non ha un significato al momento)

```
inline void enableP2P (int ngpus) {  
    for( int i = 0; i < ngpus; i++ ) {  
        cudaSetDevice(i);  
        for(int j = 0; j < ngpus; j++) {  
            if(i == j) continue;  
            int peer_access_available = 0;  
            cudaDeviceCanAccessPeer(&peer_access_available, i, j);  
            if (peer_access_available) {  
                cudaDeviceEnablePeerAccess(j, 0);  
                printf("> GPU%d enabled direct access to GPU%d\n", i, j);  
            }  
            else  
                printf("> GPU%d direct access to GPU%d not enabled\n", i, j);  
        }  
    }  
}
```

- ✓ Se l'accesso P2P tra due GPU non è consentito la copia di memoria P2P passa attraverso la host memory con relativa perdita di performance (dipendente da quanto tempo il kernel spende in computazione, se elevato potrebbe nascondere la latenza aggiuntiva)

# Allocare i buffers

array di doppi puntatori... uno per ogni GPU

```
float **d_src = (float **)malloc(sizeof(float) * ngpus);
float **d_rcv = (float **)malloc(sizeof(float) * ngpus);
float **h_src = (float **)malloc(sizeof(float) * ngpus);
cudaStream_t *stream = (cudaStream_t *)malloc(sizeof(cudaStream_t)*ngpus);

// Create CUDA event handles
cudaEvent_t start, stop;
CHECK(cudaSetDevice(0));
CHECK(cudaEventCreate(&start));
CHECK(cudaEventCreate(&stop));

for (int i = 0; i < ngpus; i++) {
    CHECK(cudaSetDevice(i));
    CHECK(cudaMalloc(&d_src[i], iBytes));
    CHECK(cudaMalloc(&d_rcv[i], iBytes));
    CHECK(cudaMallocHost((void **) &h_src[i], iBytes));
    CHECK(cudaStreamCreate(&stream[i]));
}
```

allocazione su GPU mediante UVA

# Copia (a)sincrona in memoria via P2P

- ✓ Il seg. esempio effettua 100 volte (**ping-pong**) la copia di memoria tra dev0 e dev1
- ✓ Poiché il bus PCIe supporta la comunicazione **full-duplex** tra due device, è possibile eseguire **trasferimenti bidirezionali** di memoria P2P usando le funzioni di copia **asincrone**

Asincrona...

```
// ping pong unidirectional gmem copy
cudaEventRecord(start, 0);
for (int i = 0; i < 100; i++) {
    cudaMemcpy(d_src[1], d_src[0], iBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(d_src[0], d_src[1], iBytes, cudaMemcpyDeviceToDevice);
}
```

Sincrona...

```
// bidirectional asynchronous gmem copy
for (int i = 0; i < 100; i++) {
    CHECK(cudaMemcpyAsync(d_src[1], d_src[0], iBytes, cudaMemcpyDeviceToDevice, stream[0]));
    CHECK(cudaMemcpyAsync(d_rcv[0], d_rcv[1], iBytes, cudaMemcpyDeviceToDevice, stream[1]));
}
```

# Copia sincrona in memoria via P2P

- ✓ Con **P2P** abilitato è possibile copiare direttamente i dati tra due device
- ✓ Se l'accesso **non** è supportato la copia continua senza errori usando la **host memory** come ponte
- ✓ Il seg. esempio effettua 100 volte (ping-pong) la copia di memoria tra dev0 e dev1 in modo sincrono:

```
// ping pong unidirectional gmem copy
cudaEventRecord(start, 0);
for (int i = 0; i < 100; i++) {
    cudaMemcpy(d_src[1], d_src[0], iBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(d_src[0], d_src[1], iBytes, cudaMemcpyDeviceToDevice);
}
```

Per ricavare il tempo di esecuzione si procede con gli eventi start e stop sullo stream NULL come d'uso

```
CHECK(cudaSetDevice(0));
CHECK(cudaEventRecord(start, 0));
for (int i = 0; i < 100; i++) {
    ...
}
CHECK(cudaSetDevice(0));
CHECK(cudaEventRecord(stop, 0));
CHECK(cudaEventSynchronize(stop));
float elapsed_time_ms;
CHECK(cudaEventElapsedTime(&elapsed_time_ms, start, stop));
```

# Esecuzione su Lagrange

- ✓ Copia unidirezionale e bidirezionale tra GPU 0 e GPU 1 su Lagrange

```
CUDA-capable device count: 3
GPU0: Tesla M2090 is capable of Peer-to-Peer access
GPU1: Tesla M2050 is capable of Peer-to-Peer access
GPU2: Tesla M2090 is capable of Peer-to-Peer access
GPU0 -> GPU1 (direct access NOT enabled)
GPU0 -> GPU2 (direct access enabled)
GPU1 -> GPU0 (direct access NOT enabled)
GPU1 -> GPU2 (direct access NOT enabled)
GPU2 -> GPU0 (direct access enabled)
GPU2 -> GPU1 (direct access NOT enabled)
```

Allocating buffers (64MB on each GPU and CPU Host)...

Ping-pong unidirectional cudaMemcpy:	25.30 ms performance:	2.55 GB/s
Ping-pong bidirectional cudaMemcpyAsync:	13.19ms performance:	9.98 GB/s
> GPU0 disabled direct access to GPU2		
> GPU2 disabled direct access to GPU0		

- ✓ Copia unidirezionale e bidirezionale tra GPU 0 e GPU 2 su Lagrange

Allocating buffers (64MB on each GPU and CPU Host)...

Ping-pong unidirectional cudaMemcpy:	25.45 ms performance:	2.64 GB/s
Ping-pong bidirectional cudaMemcpyAsync:	13.47ms performance:	9.96 GB/s
> GPU0 disabled direct access to GPU2		
> GPU2 disabled direct access to GPU0		

# P2P & UVA in sintesi...

- ✓ Combinando **peer-to-peer** CUDA APIs con **UVA** si hanno **accessi trasparenti** alla memoria di qualsiasi device
- ✓ Non si devono **gestire separatamente** i buffer di memoria o effettuare copie esplicite dalla memoria host
- ✓ Il **sistema lo rende possibile** evitando di esplicitare le operazioni e dunque **semplificando il codice**
- ✓ Non occorre usare **troppto** intensamente **UVA** in combinazione con **accessi peer-to-peer** perché le prestazioni possono deteriorare a causa di continui piccoli trasferimenti attraverso il bus PCIe
- ✓ se **peer-to-peer** e **UVA** sono abilitati entrambi, l'esecuzione di kernel su un device può **dereferenziare un puntatore** a un buffer di **memoria** di un **altro device**

# Esempio P2P & UVA

```
// UVA check  
  
int deviceId = 0;  
  
cudaDeviceProp prop;  
  
cudaGetDeviceProperties(&prop, deviceId);  
  
printf("GPU%d: %s unified addressing\n", deviceId, prop.unifiedAddressing ? "supports" :  
"does not support");
```

- ✓ Esempio:

attiva il **dev0**, il kernel che  
legge da gmem di **dev1**  
usando puntatore  
**d\_src[1]** e scrive in  
**d\_rcv[0]** del **dev0**

```
__global__ void kernel(float *src, float *dst) {  
  
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;  
  
    dst[idx] = src[idx] * 2.0f;  
}  
  
...  
  
cudaSetDevice(0);  
  
kernel<<<grid, block>>>(d_rcv[0], d_src[1]);
```

# P2P memcpy senza UVA

- ✓ Effettuare **trasferimento** dati **asincrono** (rispetto host e altri device) da dev **mem sorgente** verso dev **mem destinazione** del device selezionato **senza** uso di **UVA**

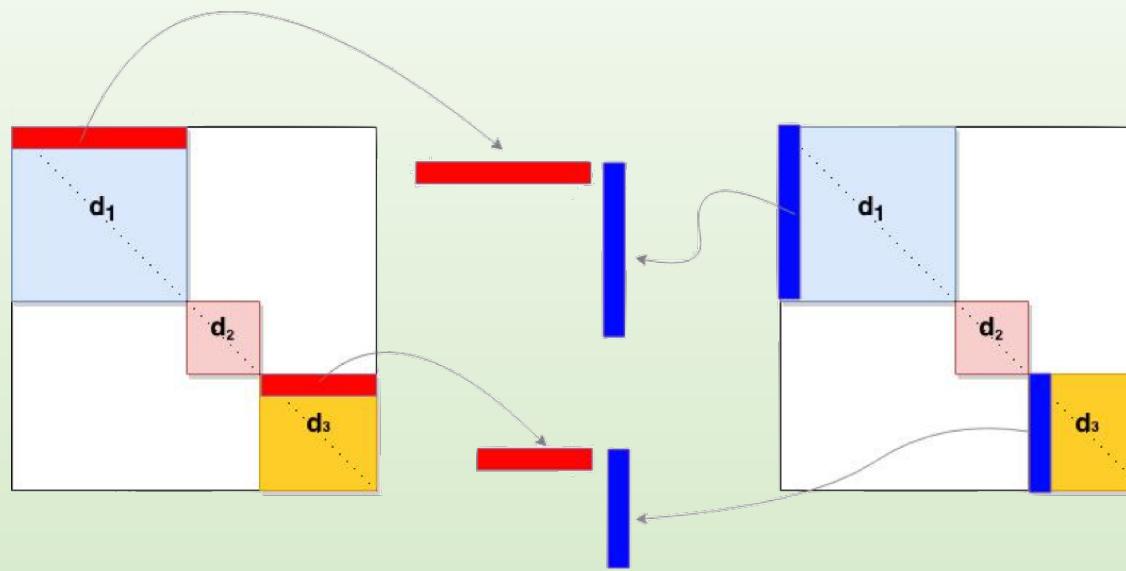
Prototipo->

```
cudaError_t cudaMemcpyPeerAsync( void* dst, int dstDev,  
                               void* src, int srcDev,  
                               nBytes, cudaStream_t stream);
```

➤ Trasferisce dati dalla device memory di **srcDev** alla device memory di **dstDev**

- ✓ Le performance sono massimizzate quando lo **stream** appartiene alla **GPU sorgente**
- ✓ Esiste anche la modalità **blocking** (opposto a Async)
- ✓ Se peer-access è **abilitato**:
  - I byte vengono trasferiti lungo il cammino PCIe più breve (se appartengono allo stesso nodo)
  - Non si usa passaggio attraverso memoria host
- ✓ Se peer-access non è **disponibile**
  - Il CUDA driver effettua il trasferimento attraverso la memoria CPU
  - Non occorre gestire manualmente buffer per lo scambio

# Esercitazione



## Prodotti su matrici a blocchi multi-GPU

Disegnare un kernel per il prodotto tra matrici a blocchi con le seguenti specifiche

- ✓ Seguire il package MQDB e il partire da precedente esercitazione che usa stream e SMEM
- ✓ Suddividere gli stream per il calcolo del prodotto tra le 3 GPU di lagrange
- ✓ Verifica dei risultati finali e sincronizzazione del device
- ✓ Analisi di prestazioni usando i tempi ricavati con CUDA event

# Esercitazione

## Prodotto di matrici distribuito:

Disegnare un algoritmo distribuito su tre GPU per i seguenti prodotti di matrici tra loro dipendenti:

1. Le matrici quadrate **B** e **C** sono date
2. Calcolare i prodotti:

$$A = B \times C \quad \& \quad D = A \times B \quad \& \quad E = D \times C$$

3. Usare il meccanismo di accesso P2P tra le tre schede Tesla di Lagrange e distribuire le computazioni su ciascuna per ottimizzare il tempo

