

# UNIFIED MEMORY GSP GPU COURSE 2018

8 August 2018 | Andreas Herten | Forschungszentrum Jülich

# Overview, Outline

## Overview

- Unified Memory enables easy access to GPU development
- But some tuning might be needed for best performance

## Contents

### Background on Unified Memory

History of GPU Memory

Unified Memory on Pascal

Unified Memory on Kepler

### Practical Differences

Revisiting `scale_vector_um` Example

Hints for Performance

Task

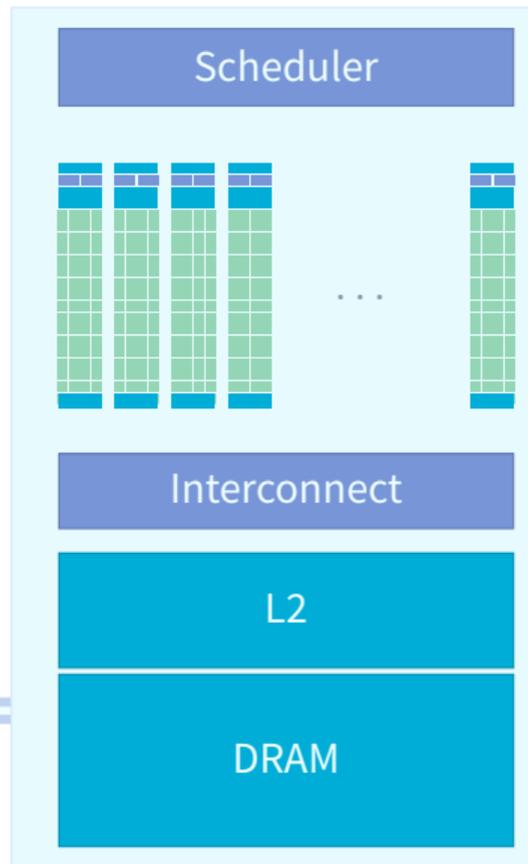
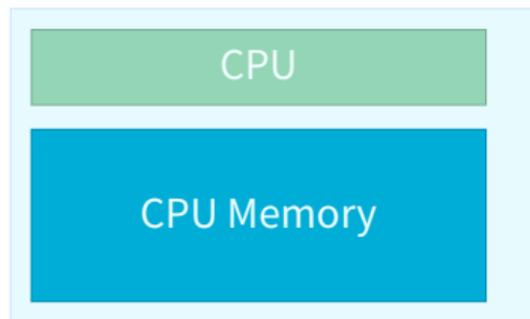
# Background on Unified Memory

## History of GPU Memory

# CPU and GPU Memory

## Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

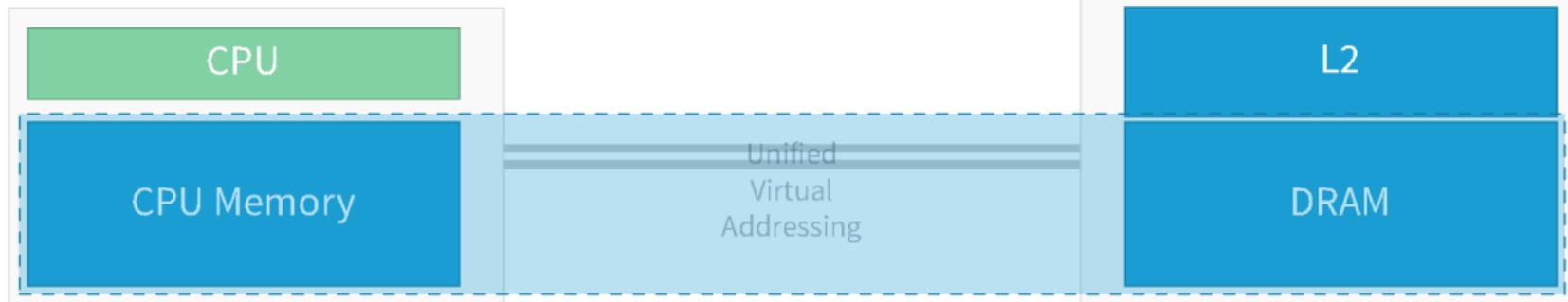


# CPU and GPU Memory

## Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual



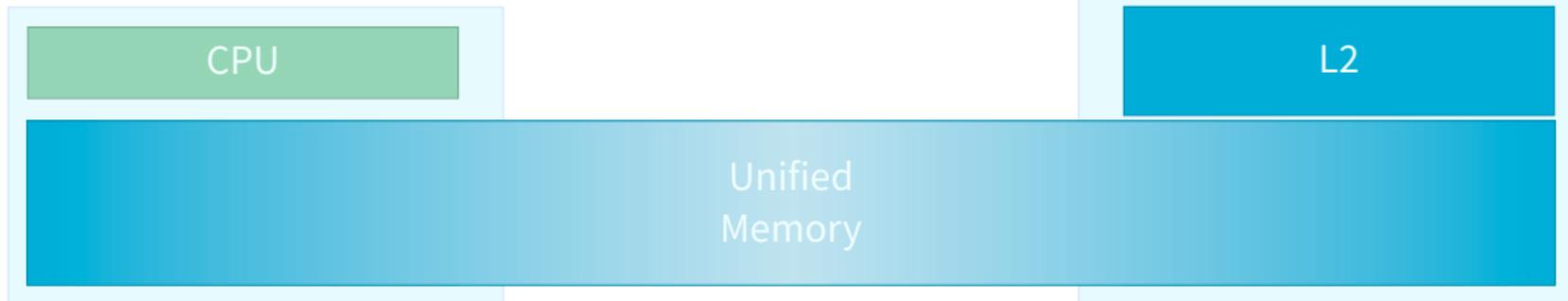
# CPU and GPU Memory

## Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual

CUDA 6.0 Unified Memory\*: Data copy by driver, but whole data at once



# CPU and GPU Memory

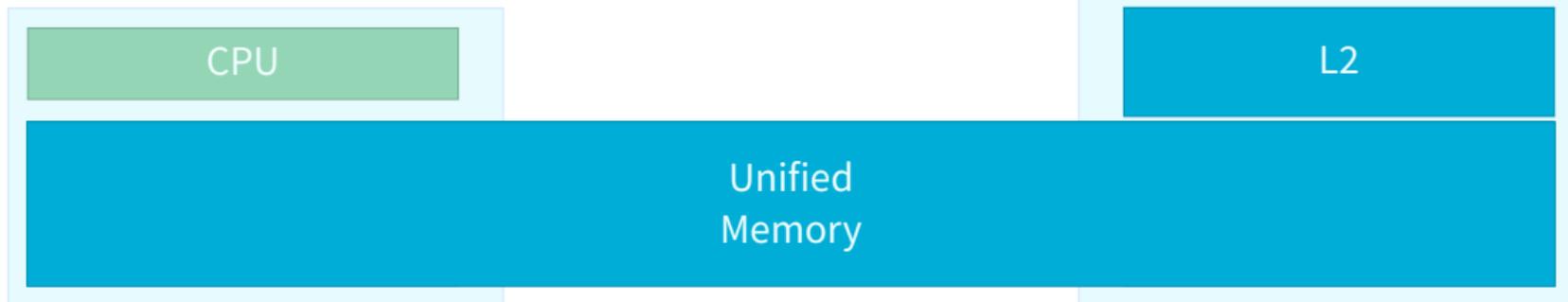
## Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual

CUDA 6.0 Unified Memory\*: Data copy by driver, but whole data at once

CUDA 8.0 Unified Memory (truly): Data copy by driver, page faults on-demand initiate data migrations (Pascal)



# CPU and GPU Memory

## Location, location, location

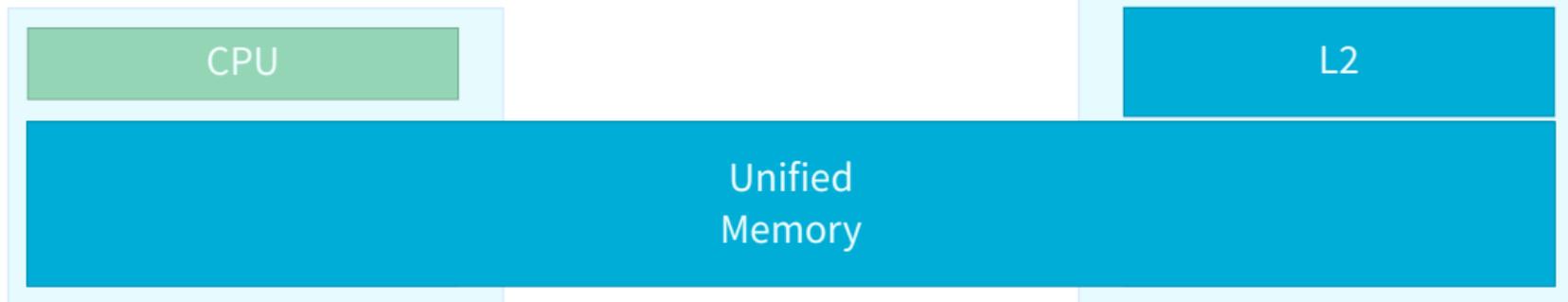
At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual

CUDA 6.0 Unified Memory\*: Data copy by driver, but whole data at once

CUDA 8.0 Unified Memory (truly): Data copy by driver, page faults on-demand initiate data migrations (Pascal)

Future Address Translation Service: Omit page faults



# Unified Memory in Code

Vojgje Nfnpsz

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    char *data_d;  
  
    data = (char *)malloc(N);  
    cudaMalloc(&data_d, N);  
  
    fread(data, 1, N, fp);  
  
    cudaMemcpy(data_d, data, N,  
        ↪ cudaMemcpyHostToDevice);  
    kernel<<<...>>>(data, N);  
  
    cudaMemcpy(data, data_d, N,  
        ↪ cudaMemcpyDeviceToHost);  
    host_func(data);  
    cudaFree(data_d); free(data); }  
}
```

```
void sortfile(FILE *fp, int N) {  
    char *data;  
  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    kernel<<<...>>>(data, N);  
    cudaDeviceSynchronize();  
  
    host_func(data);  
    cudaFree(data); }  
}
```

# Implementation Details (on Pascal)

Under the hood

```
cudaMallocManaged(&ptr, ...);
```

```
*ptr = 1;
```

```
kernel<<<...>>>(ptr);
```

# Implementation Details (on Pascal)

## Under the hood

```
cudaMallocManaged(&ptr, ...); ←● Empty! No pages anywhere yet (like malloc())
```

```
*ptr = 1;
```

```
kernel<<<...>>>(ptr);
```

# Implementation Details (on Pascal)

## Under the hood

`cudaMallocManaged(&ptr, ...);` ← ● Empty! No pages anywhere yet (like `malloc()`)

`*ptr = 1;` ← ● CPU page fault: data allocates on CPU

`kernel<<<...>>>(ptr);`

# Implementation Details (on Pascal)

## Under the hood

`cudaMallocManaged(&ptr, ...);` ← ● Empty! No pages anywhere yet (like `malloc()`)

`*ptr = 1;` ← ● CPU page fault: data allocates on CPU

`kernel<<<...>>>(ptr);` ← ● GPU page fault: data migrates to GPU

# Implementation Details (on Pascal)

## Under the hood

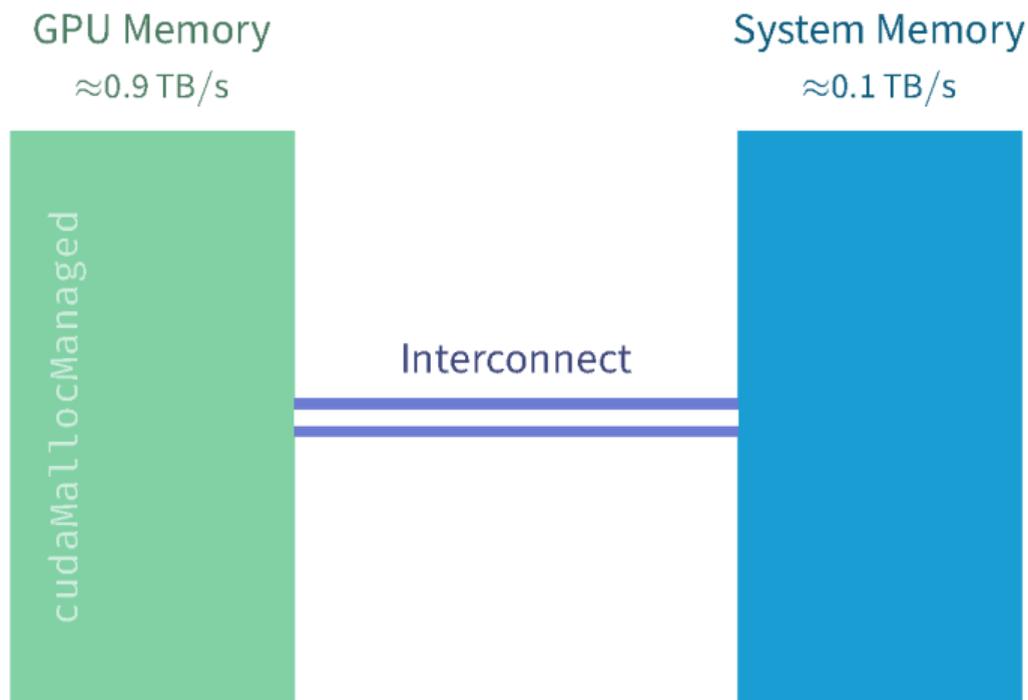
`cudaMallocManaged(&ptr, ...);` ← ● Empty! No pages anywhere yet (like `malloc()`)

`*ptr = 1;` ← ● CPU page fault: data allocates on CPU

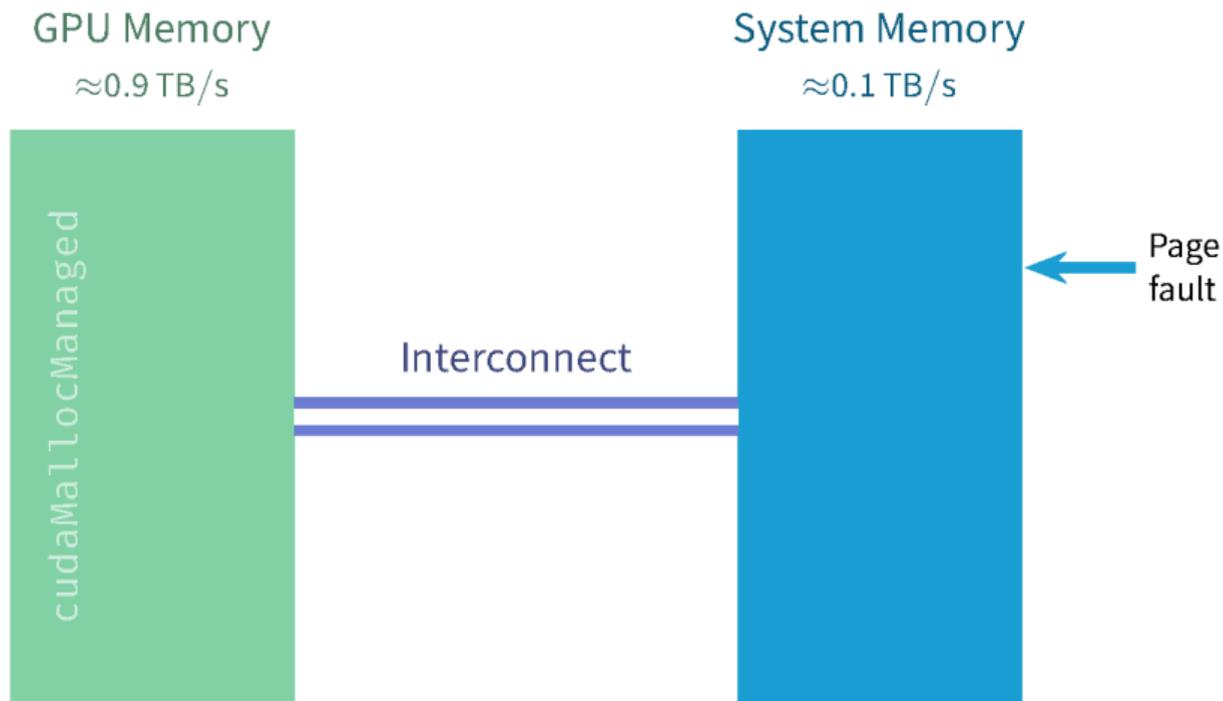
`kernel<<<...>>>(ptr);` ← ● GPU page fault: data migrates to GPU

- Pages populate on **first touch**
- Pages migrate on-demand
- GPU memory over-subscription possible
- Concurrent access from CPU and GPU to memory (page-level)

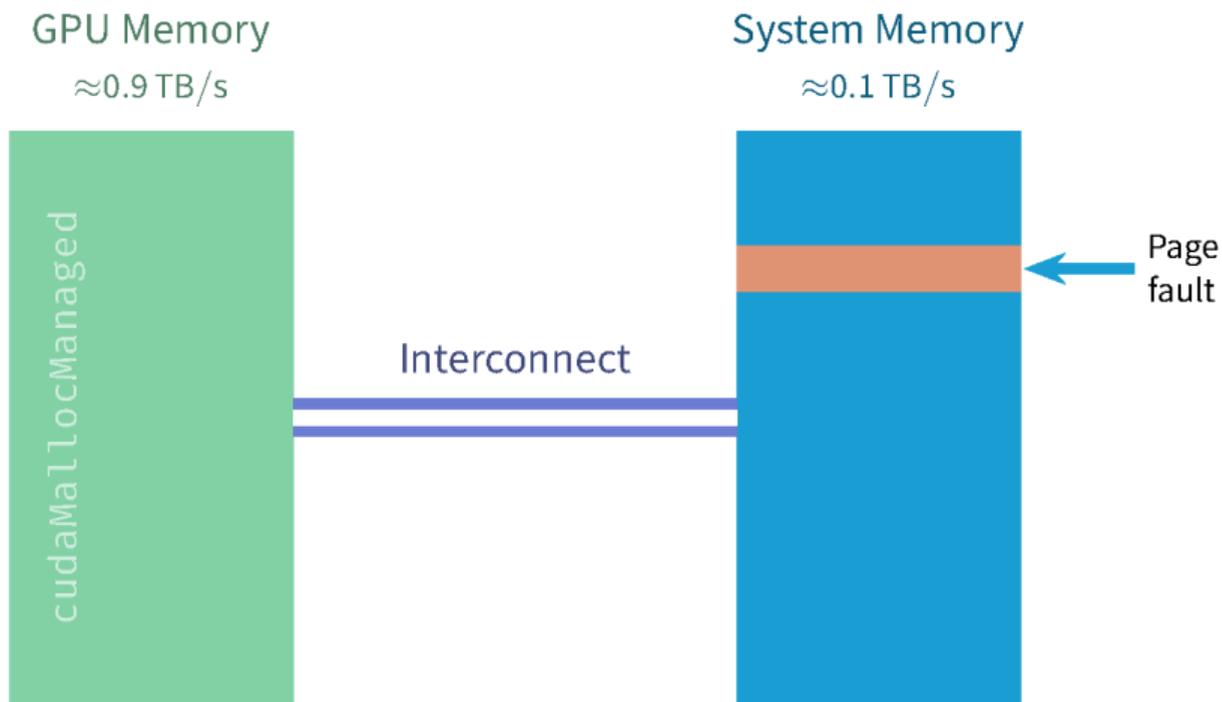
# On-Demand Migration Flow (Pascal, Volta)



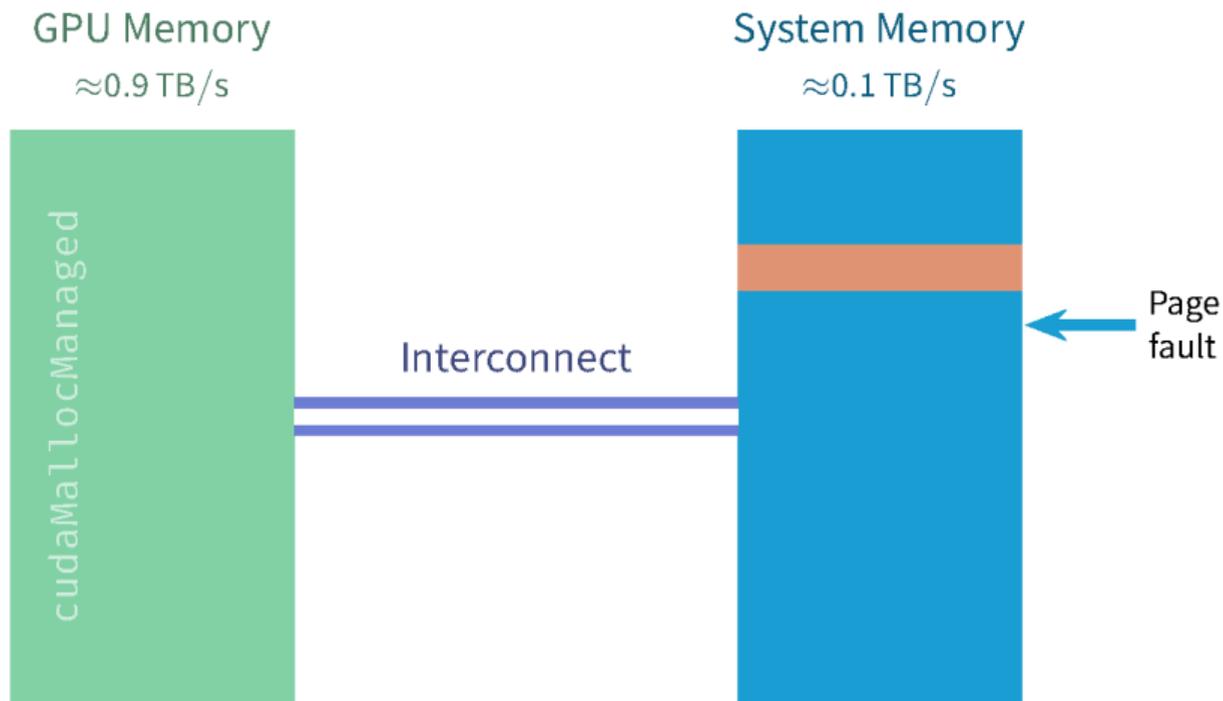
# On-Demand Migration Flow (Pascal, Volta)



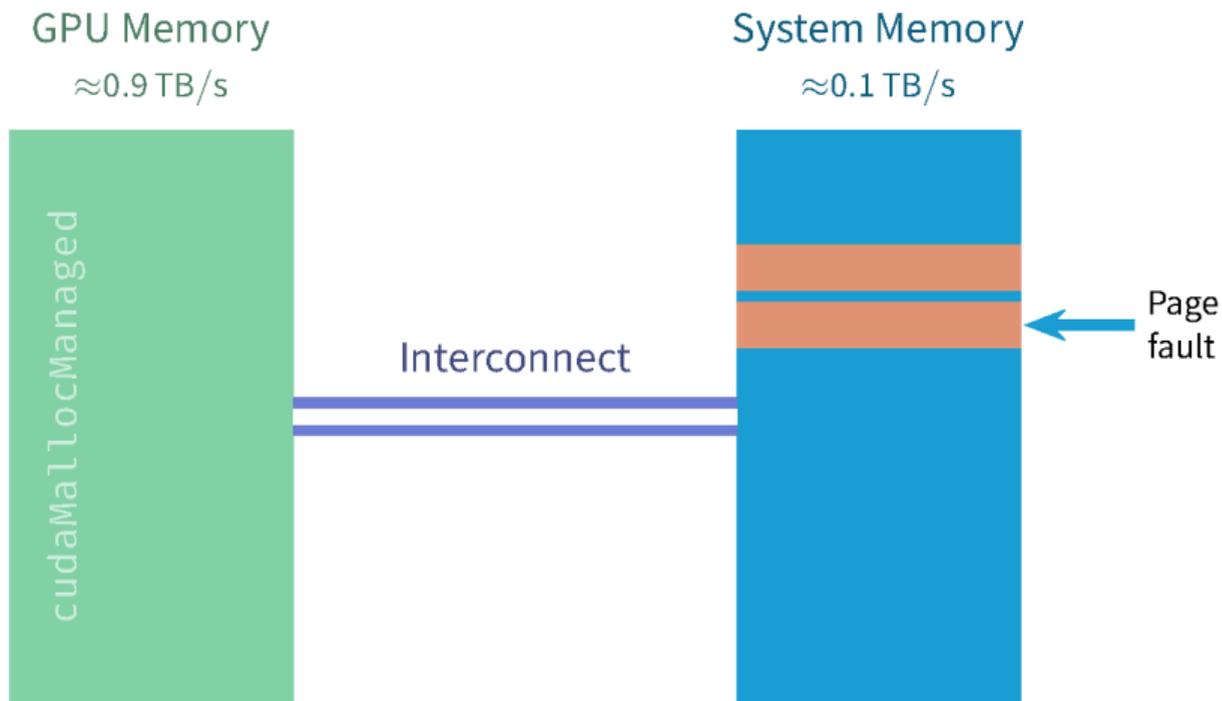
# On-Demand Migration Flow (Pascal, Volta)



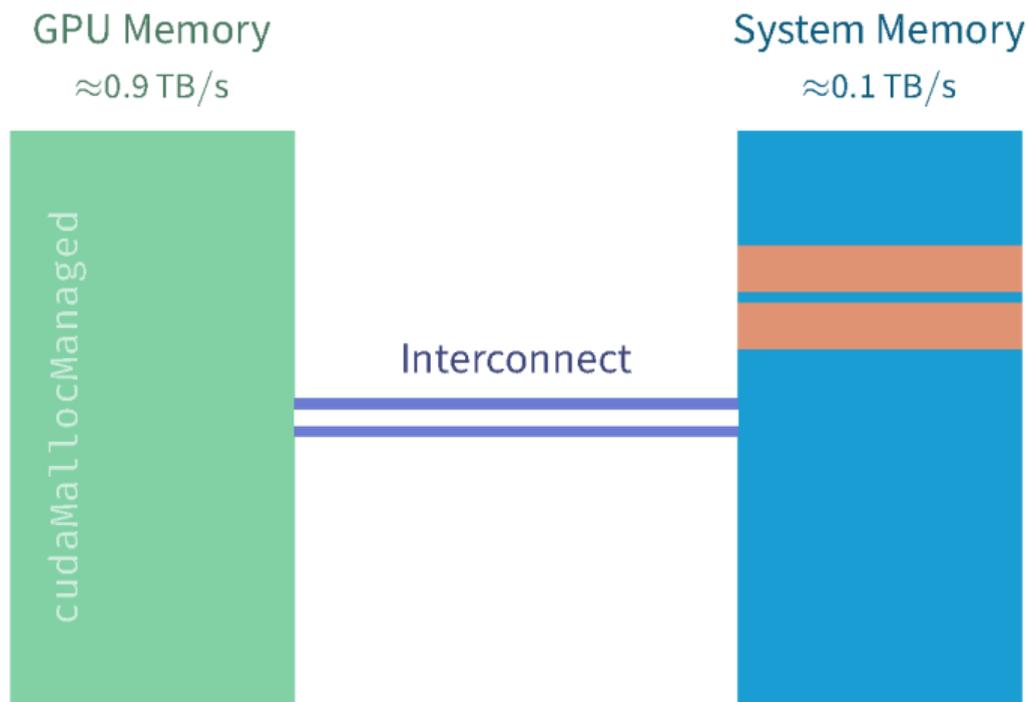
# On-Demand Migration Flow (Pascal, Volta)



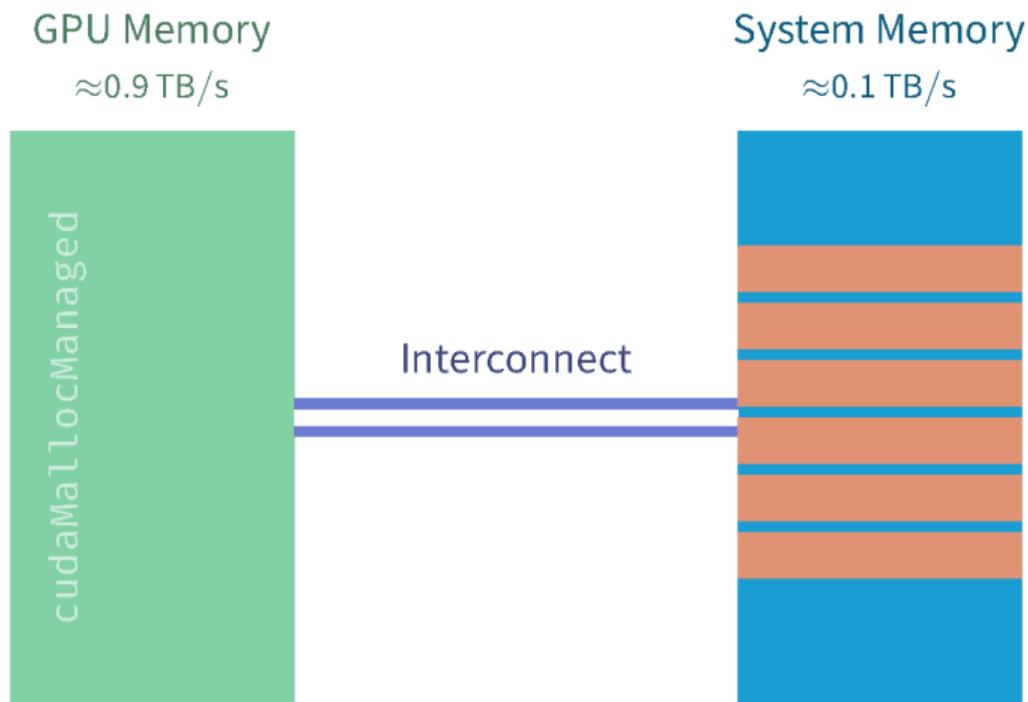
# On-Demand Migration Flow (Pascal, Volta)



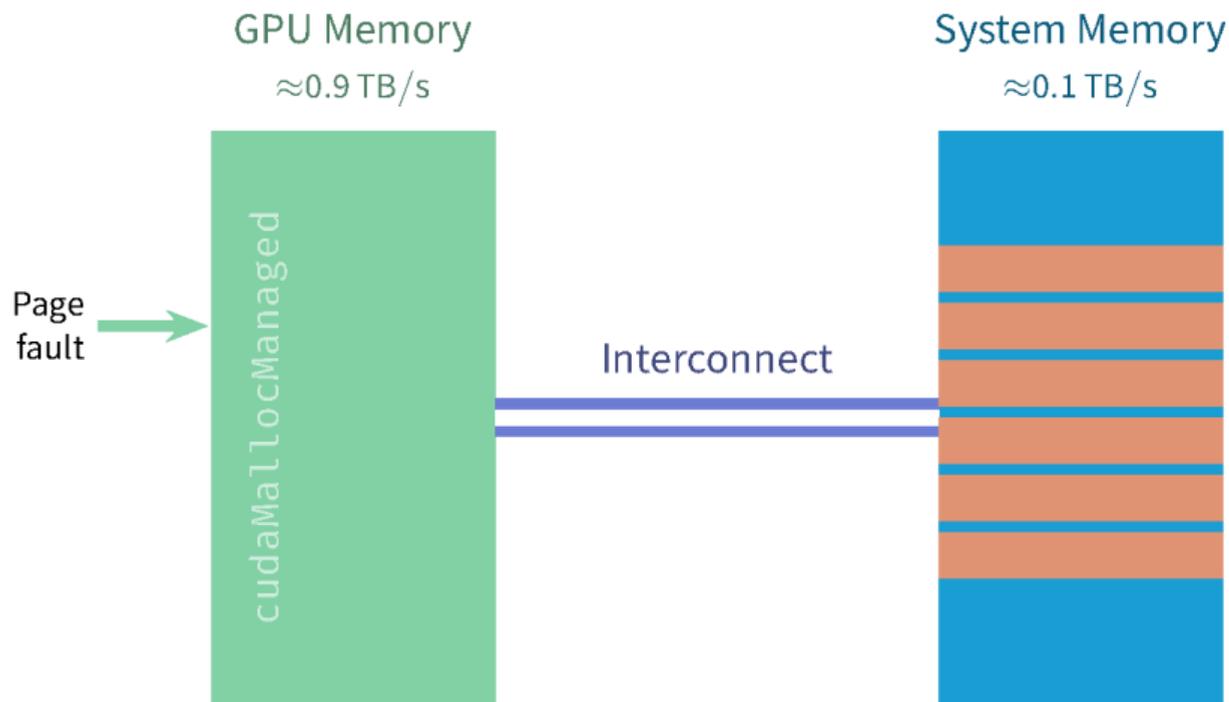
# On-Demand Migration Flow (Pascal, Volta)



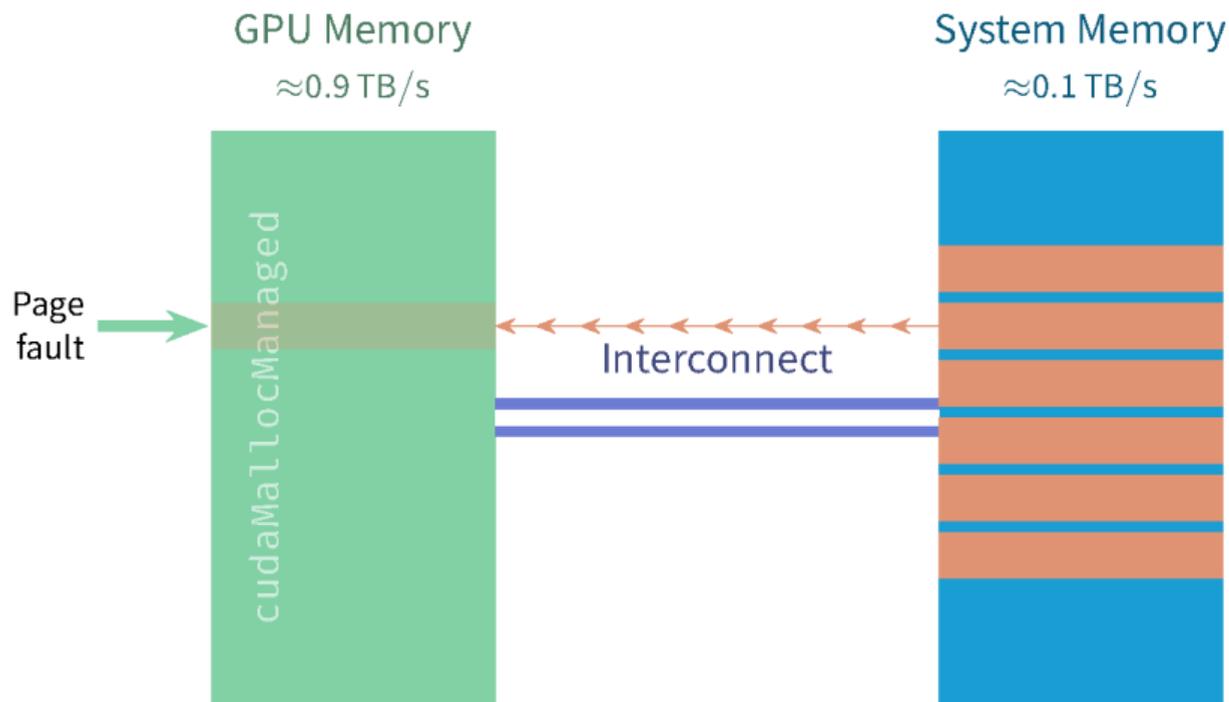
# On-Demand Migration Flow (Pascal, Volta)



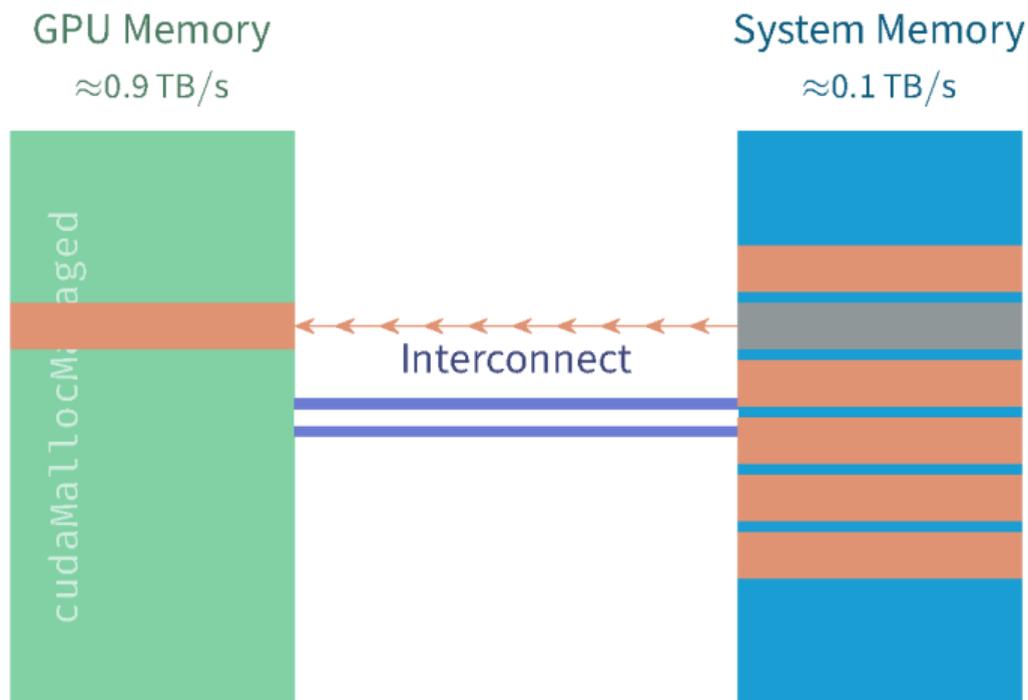
# On-Demand Migration Flow (Pascal, Volta)



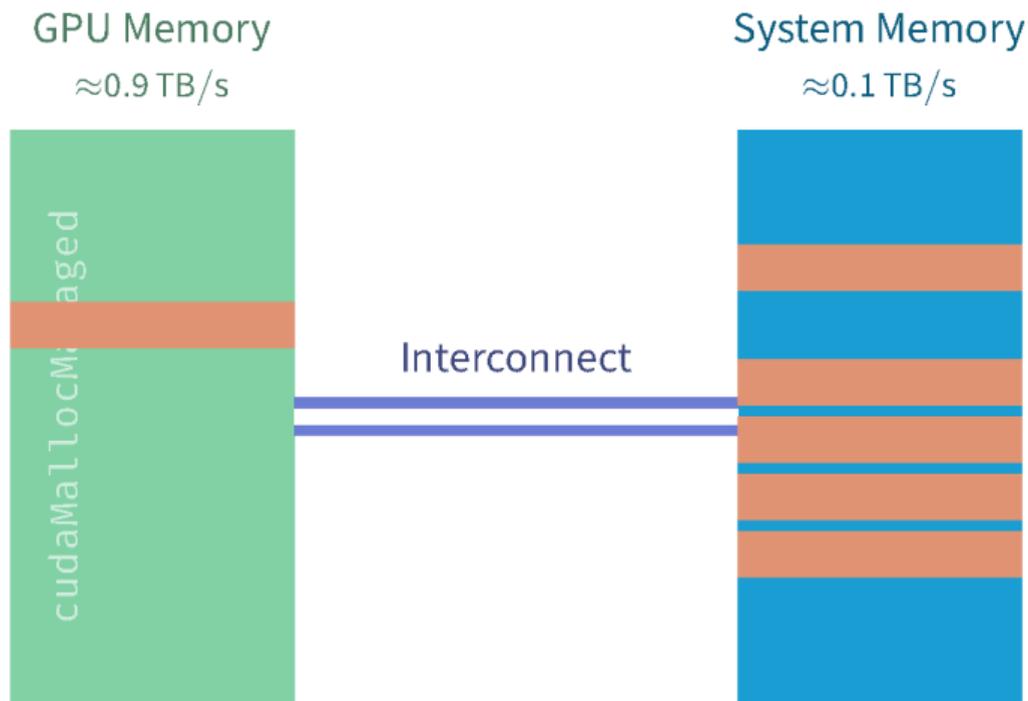
# On-Demand Migration Flow (Pascal, Volta)



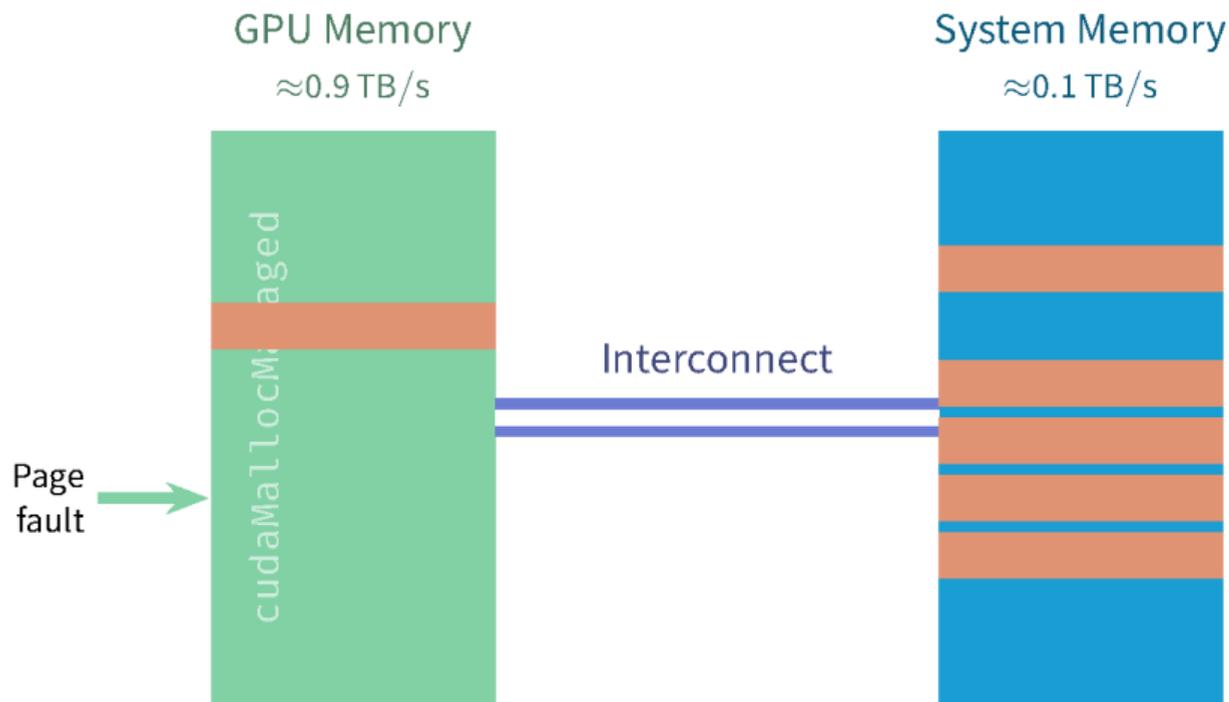
# On-Demand Migration Flow (Pascal, Volta)



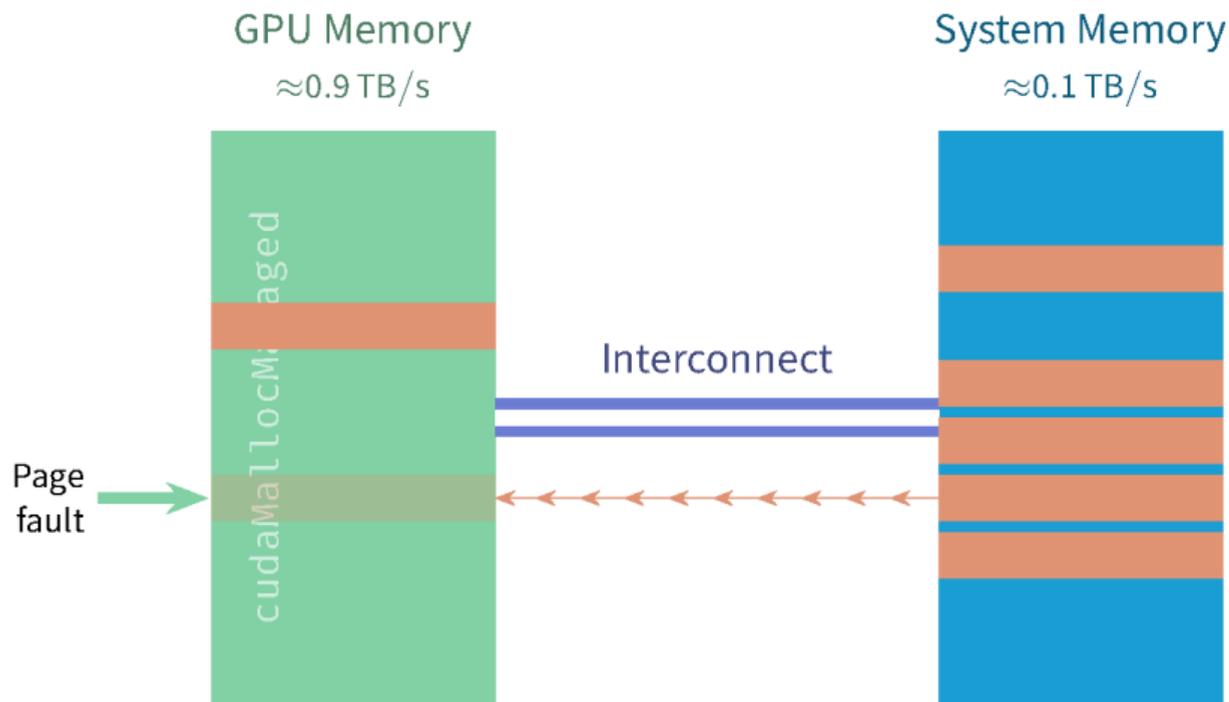
# On-Demand Migration Flow (Pascal, Volta)



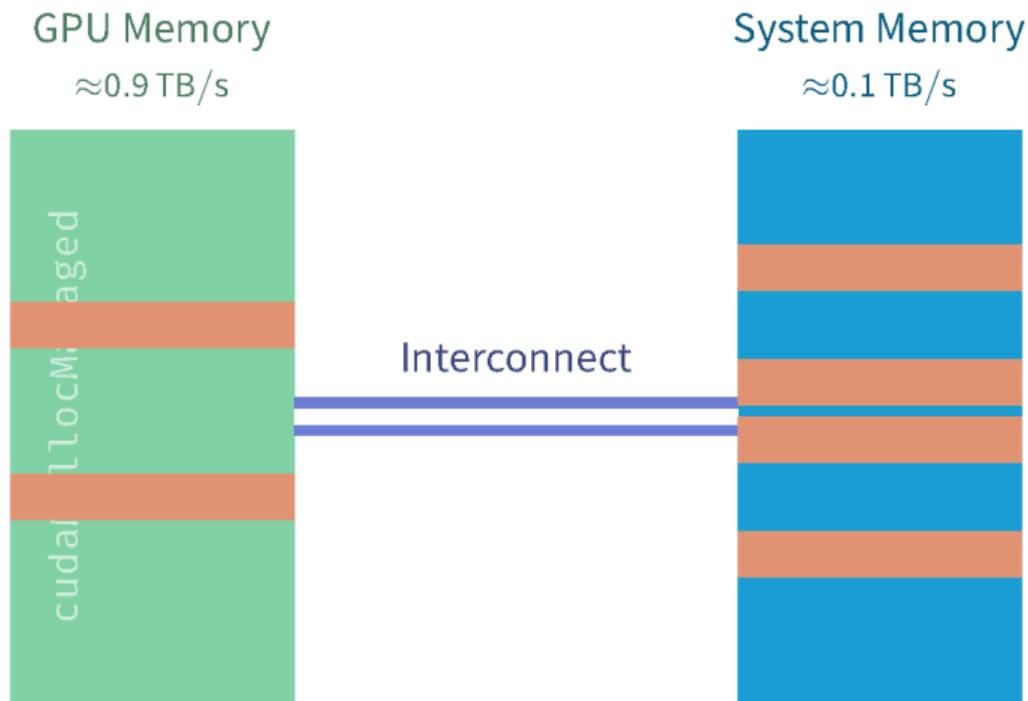
# On-Demand Migration Flow (Pascal, Volta)



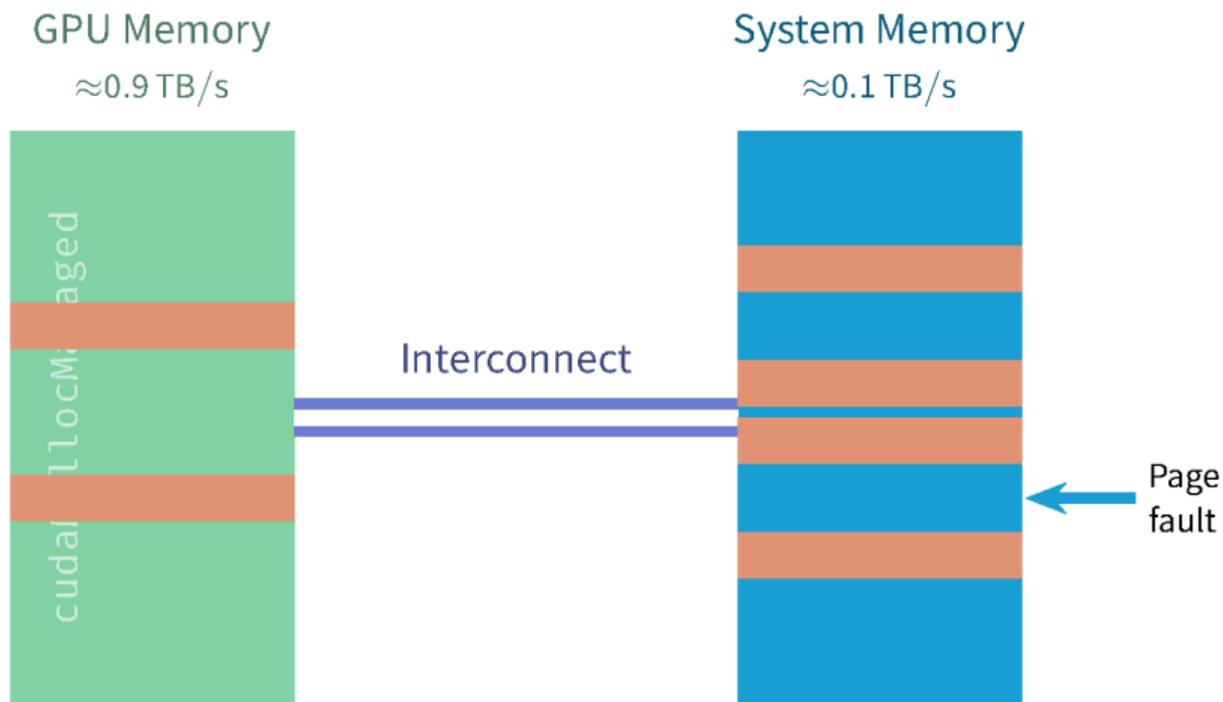
# On-Demand Migration Flow (Pascal, Volta)



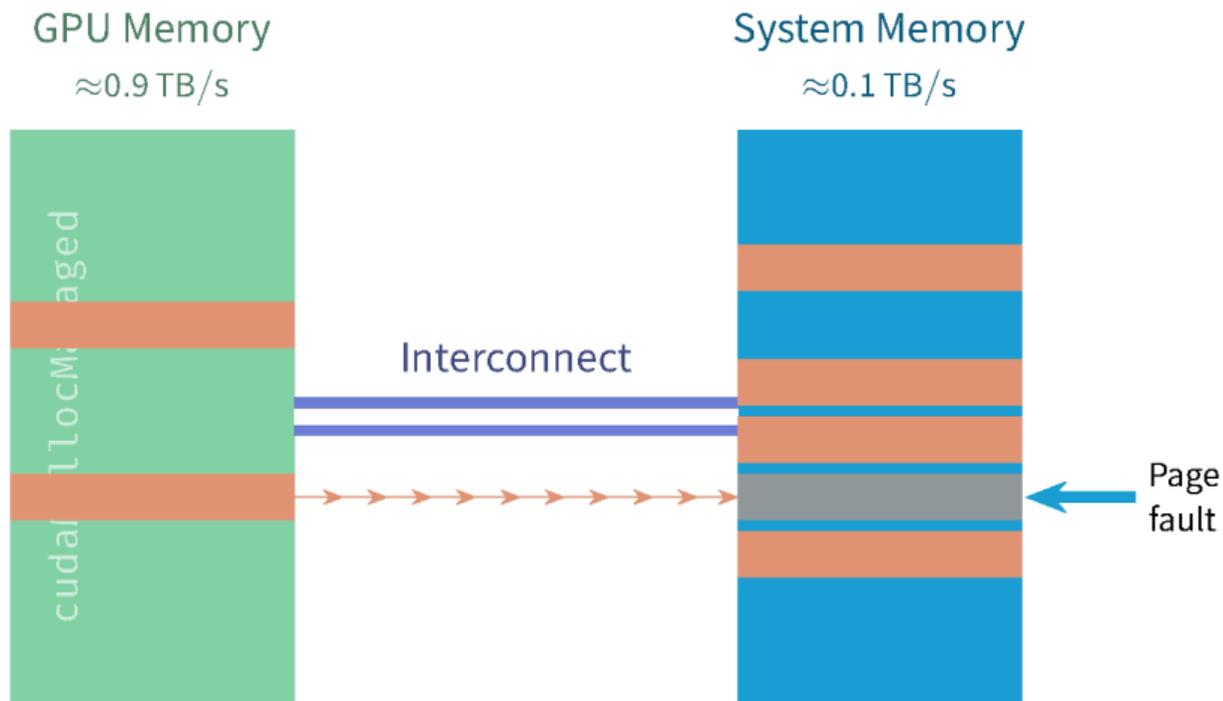
# On-Demand Migration Flow (Pascal, Volta)



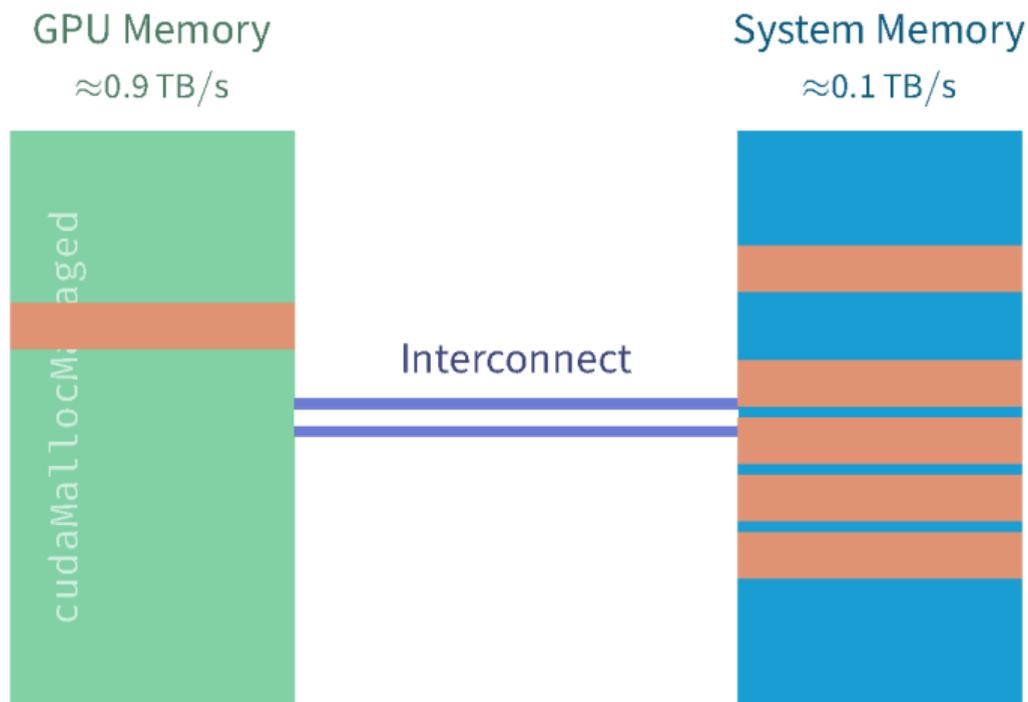
# On-Demand Migration Flow (Pascal, Volta)



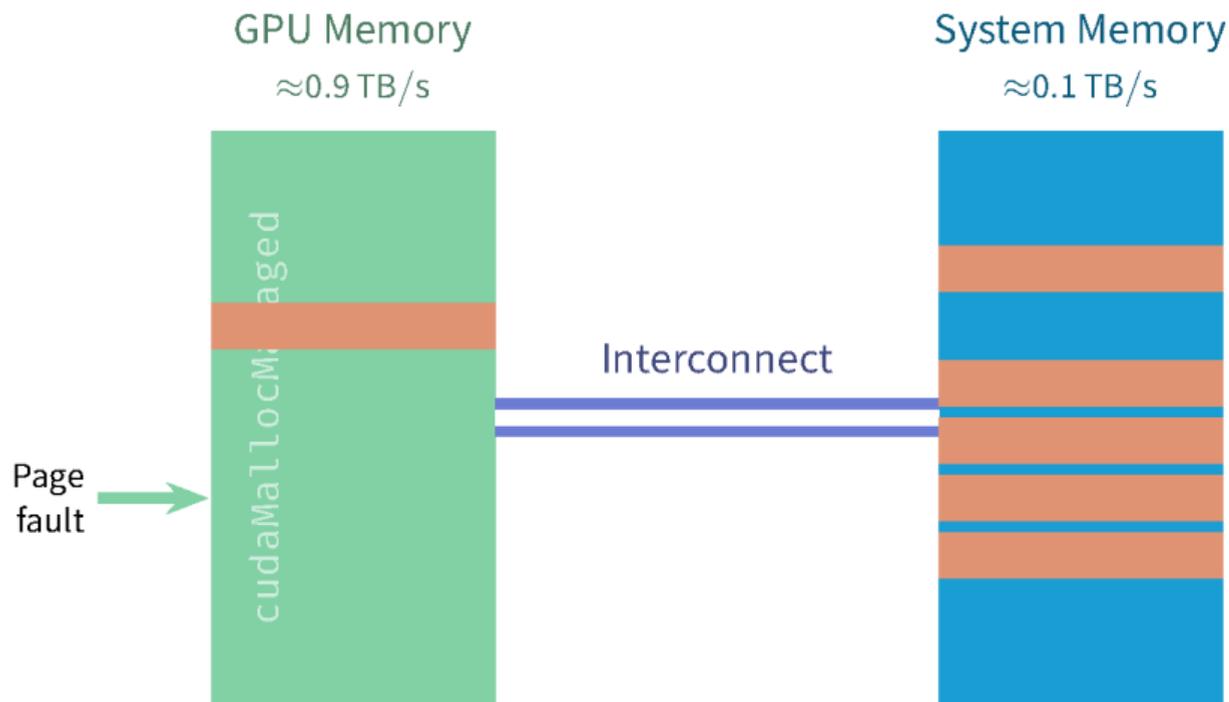
# On-Demand Migration Flow (Pascal, Volta)



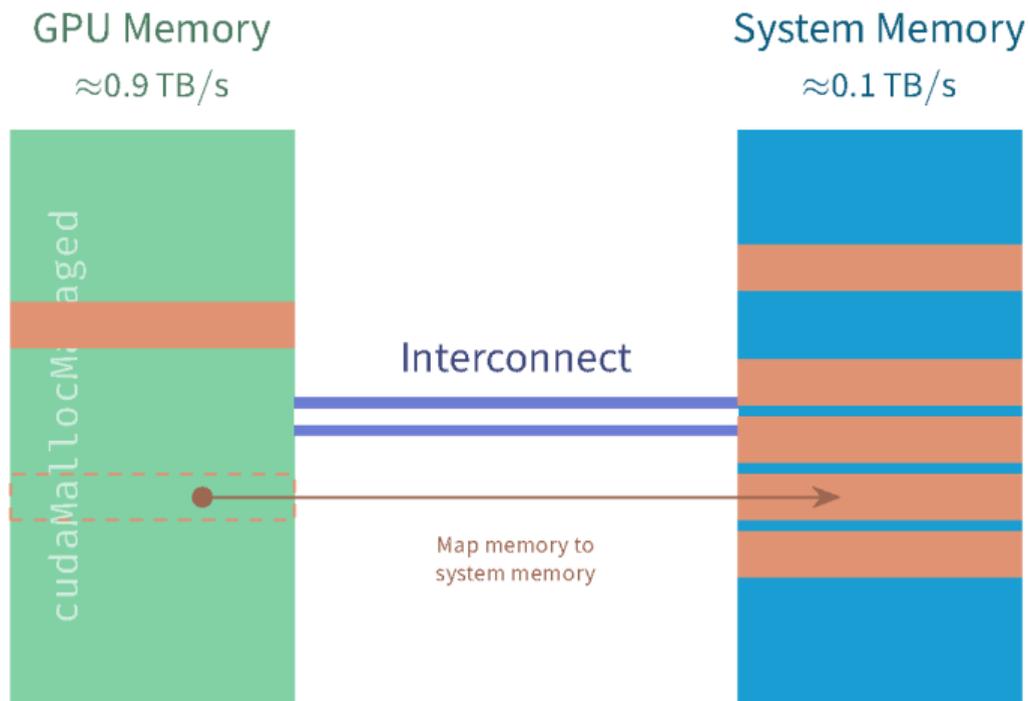
# On-Demand Migration Flow (Pascal, Volta)



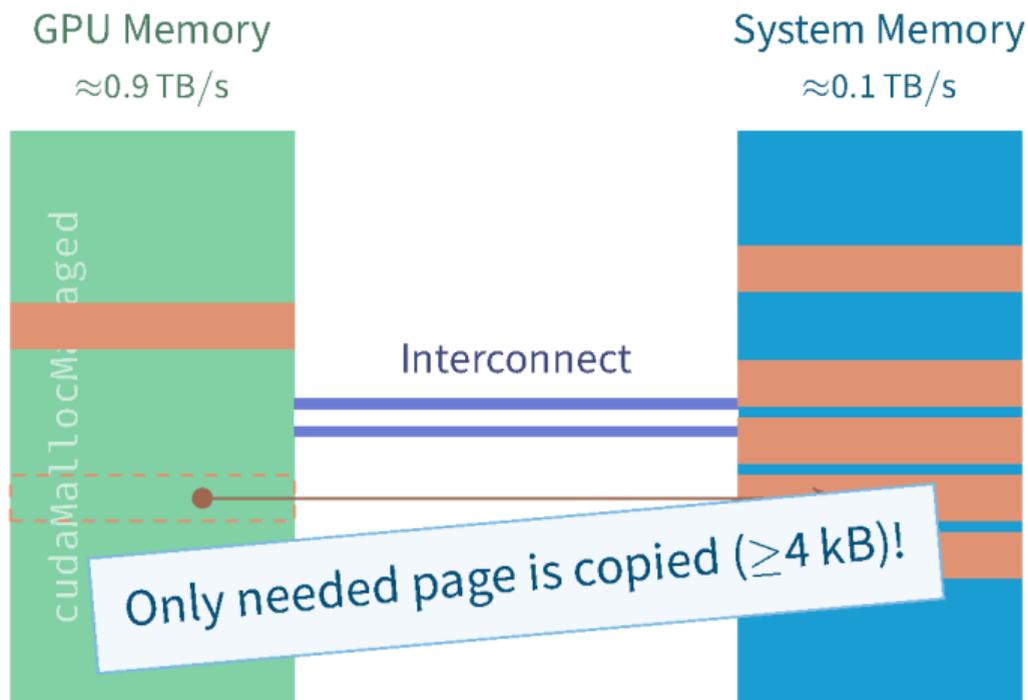
# On-Demand Migration Flow (Pascal, Volta)



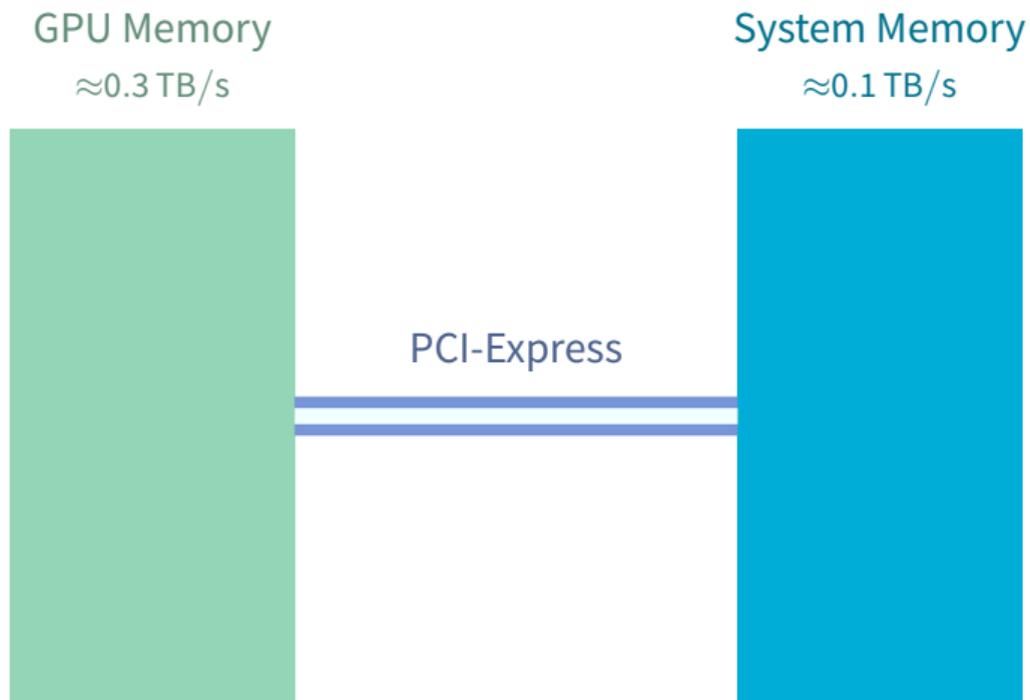
# On-Demand Migration Flow (Pascal, Volta)



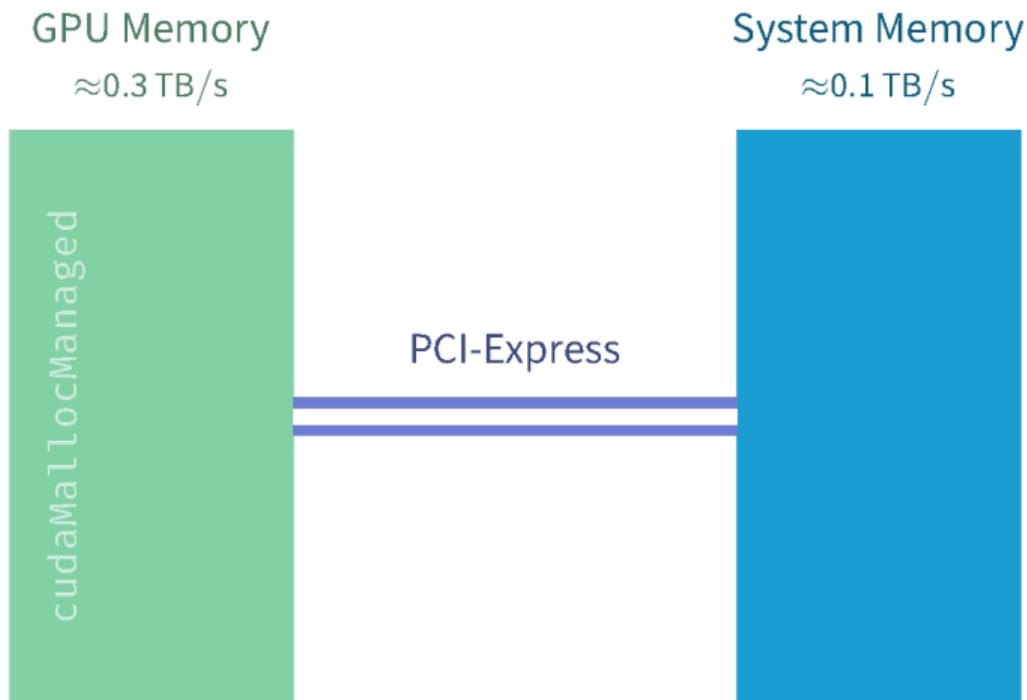
# On-Demand Migration Flow (Pascal, Volta)



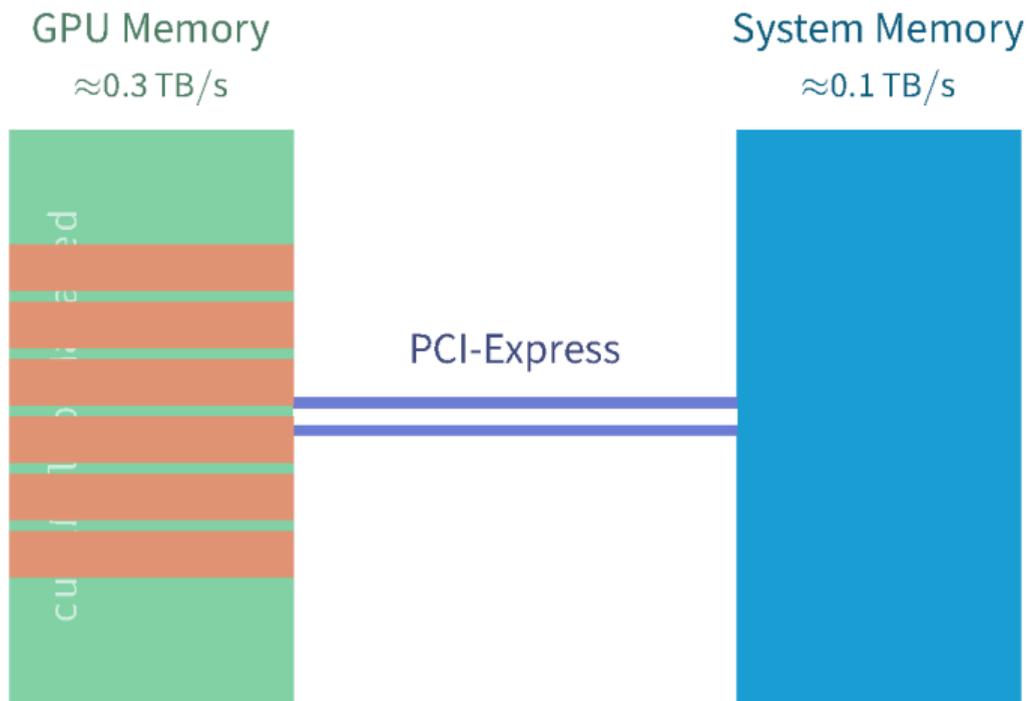
# Migration on Kepler



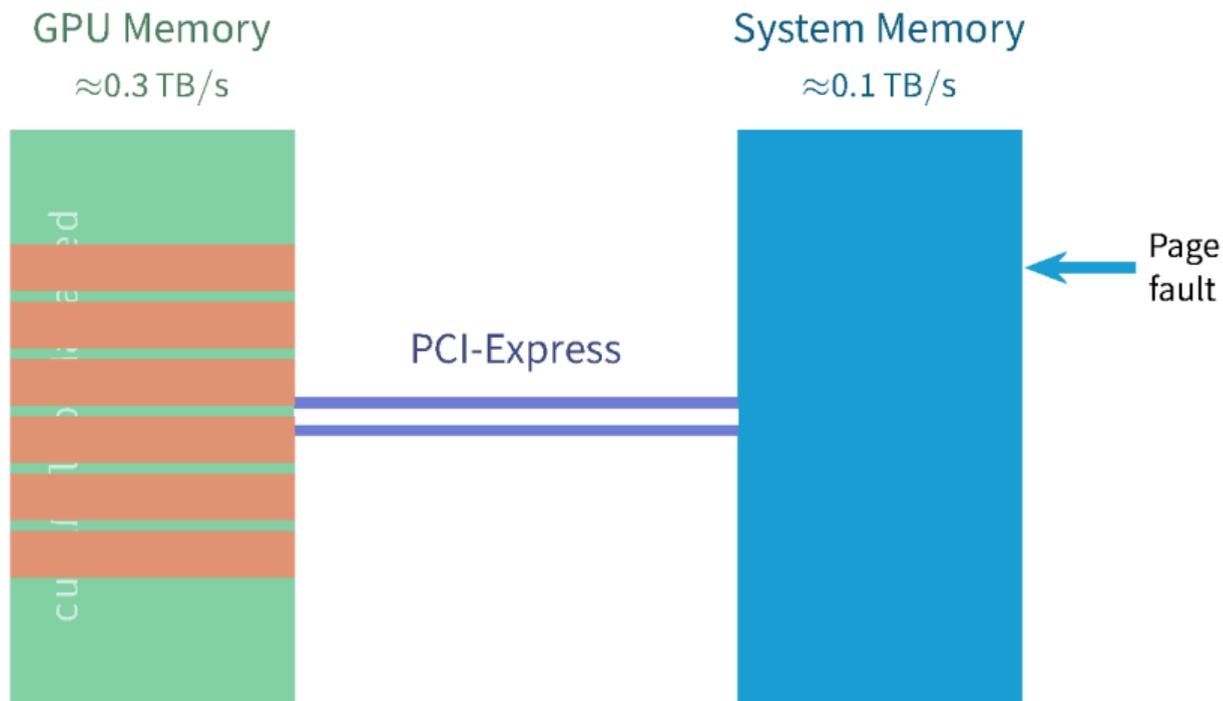
# Migration on Kepler



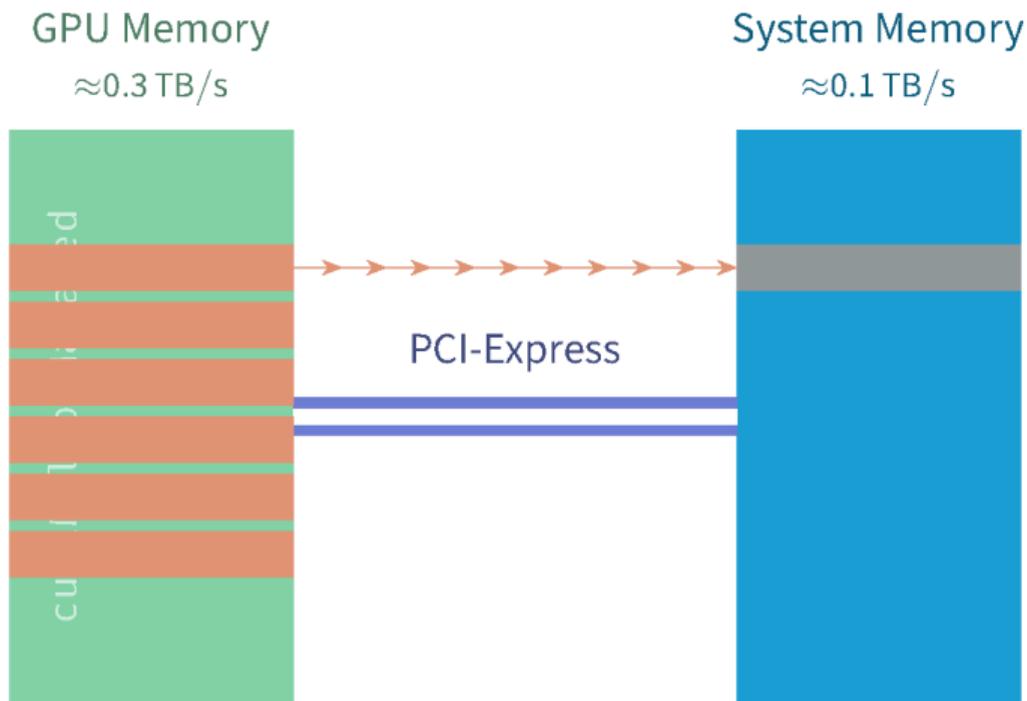
# Migration on Kepler



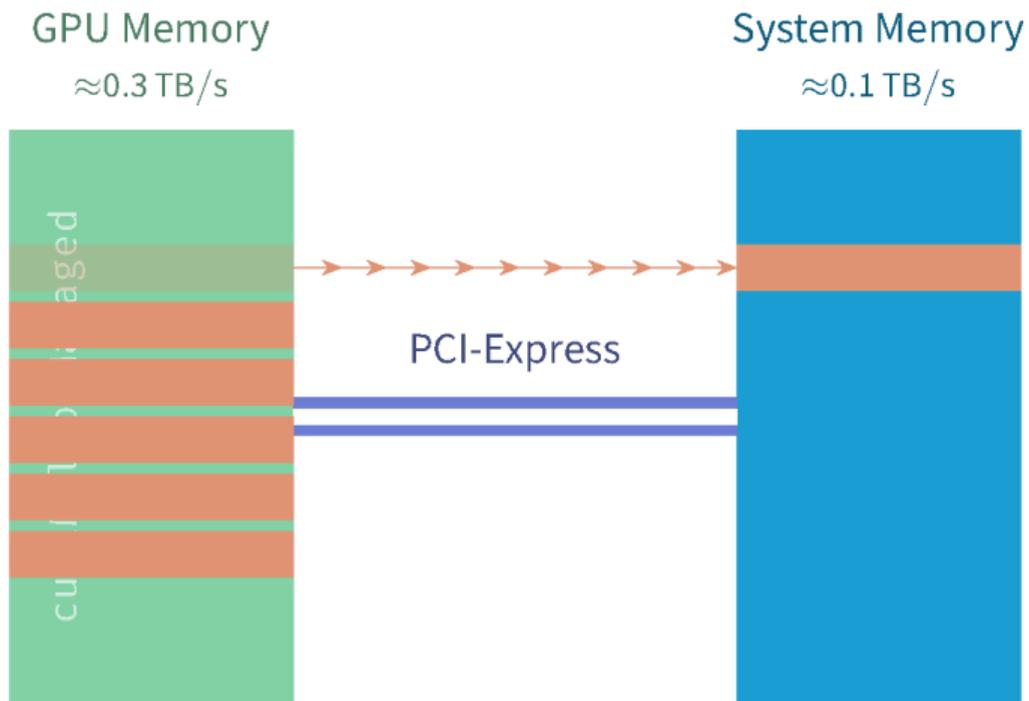
# Migration on Kepler



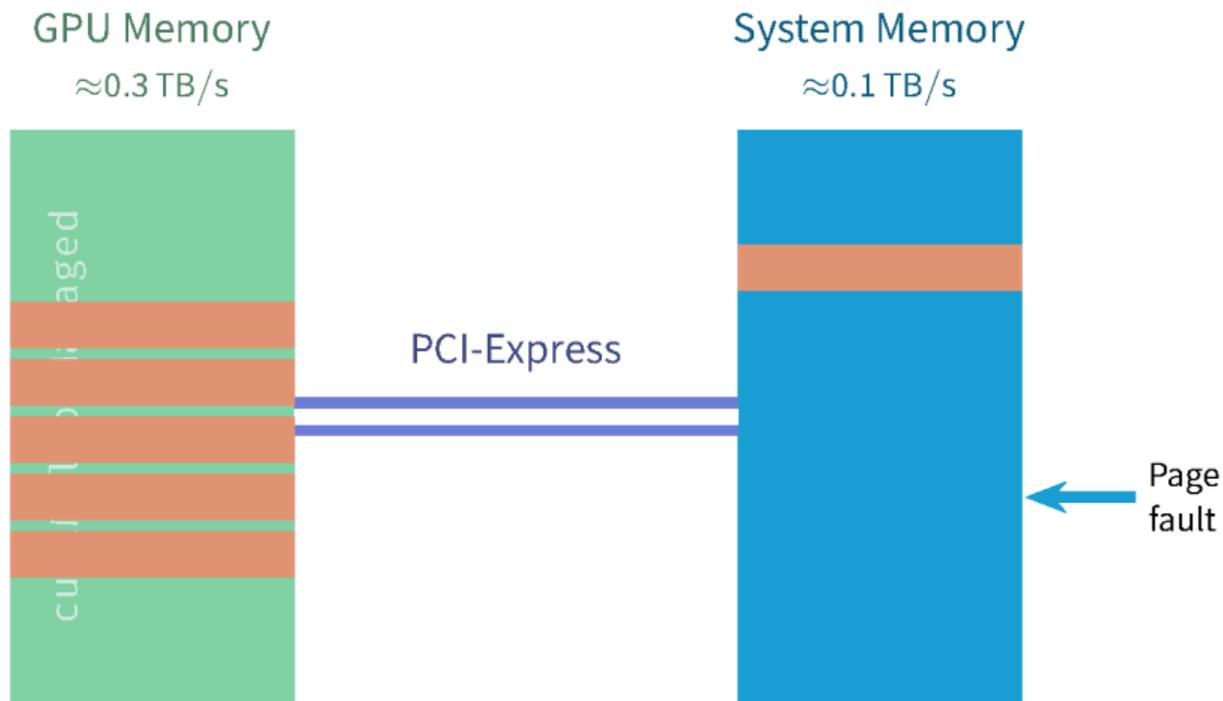
# Migration on Kepler



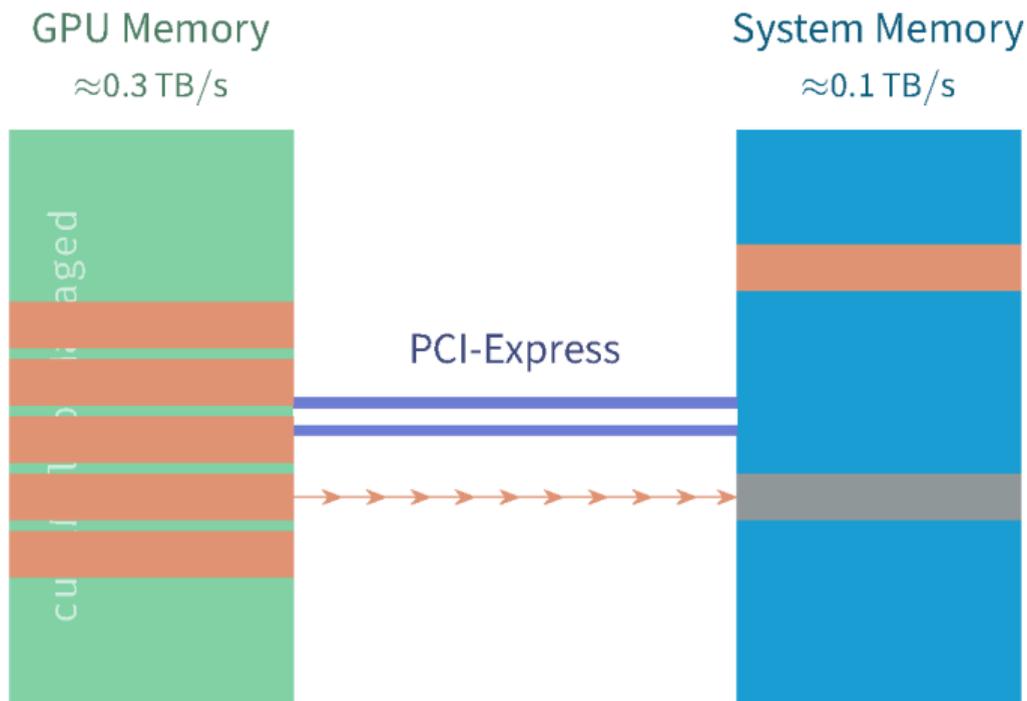
# Migration on Kepler



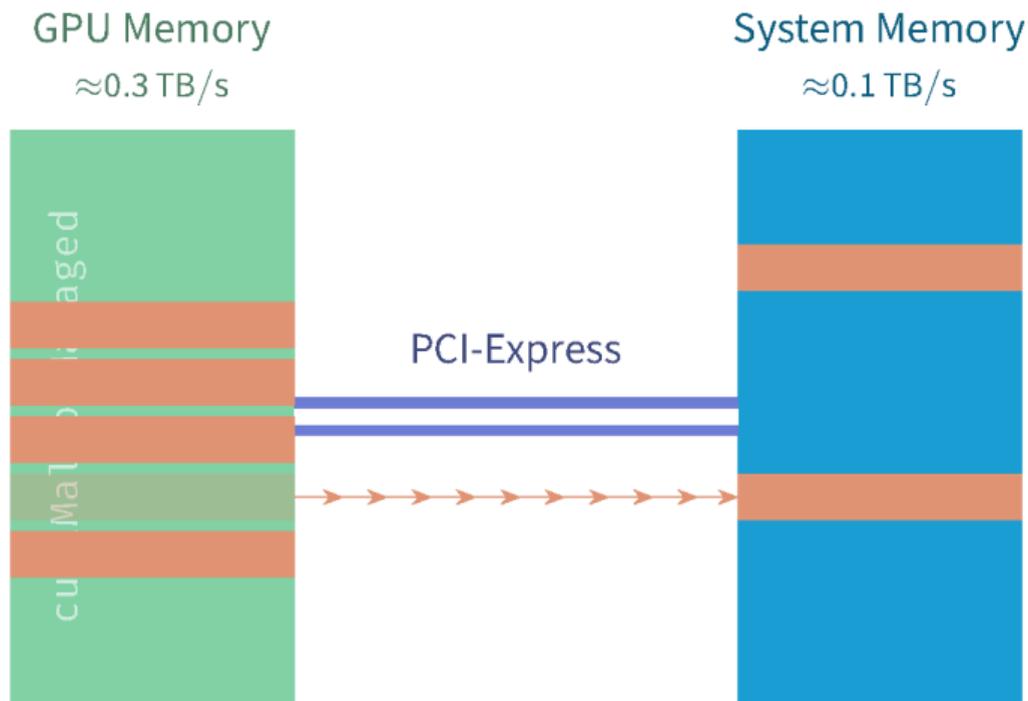
# Migration on Kepler



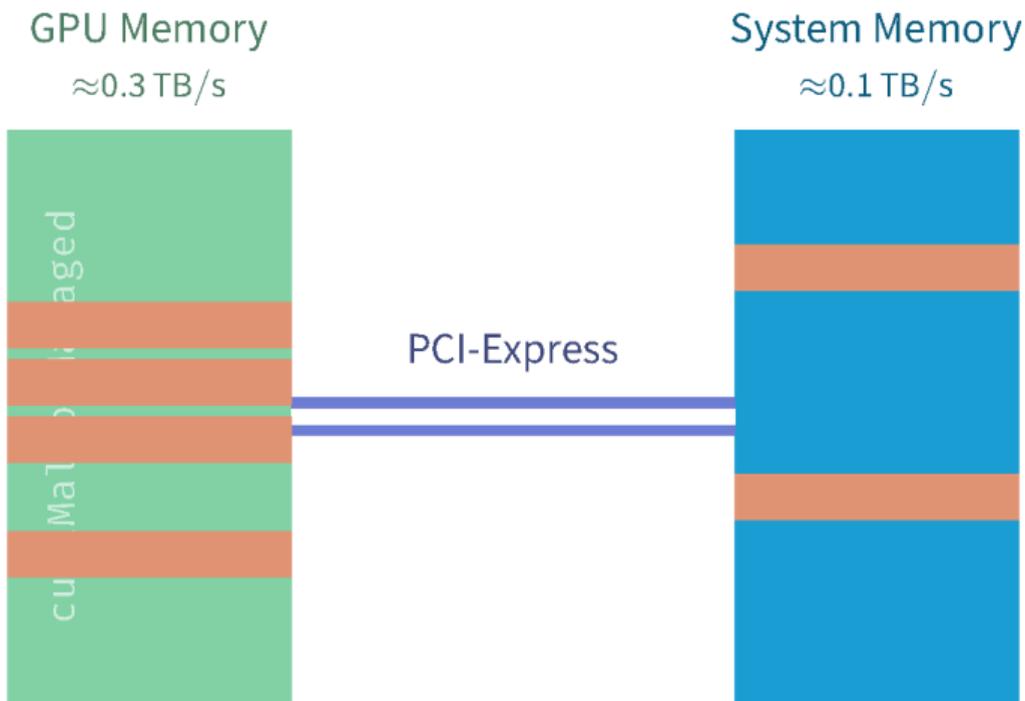
# Migration on Kepler



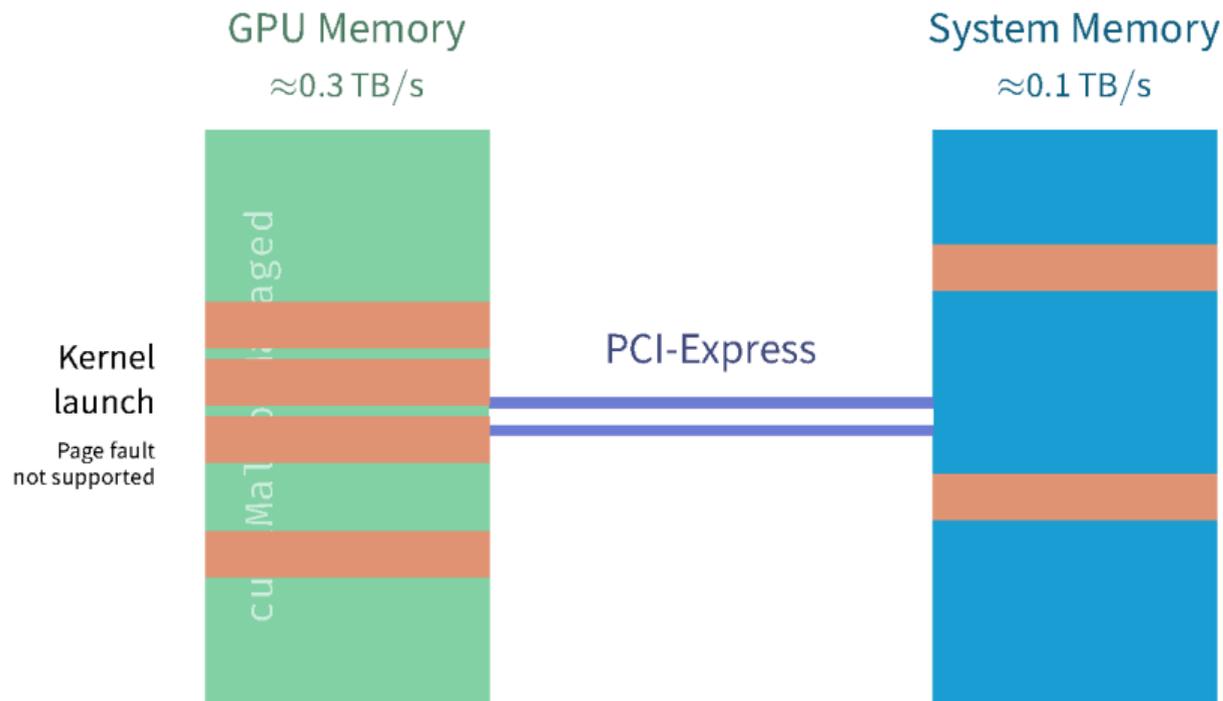
# Migration on Kepler



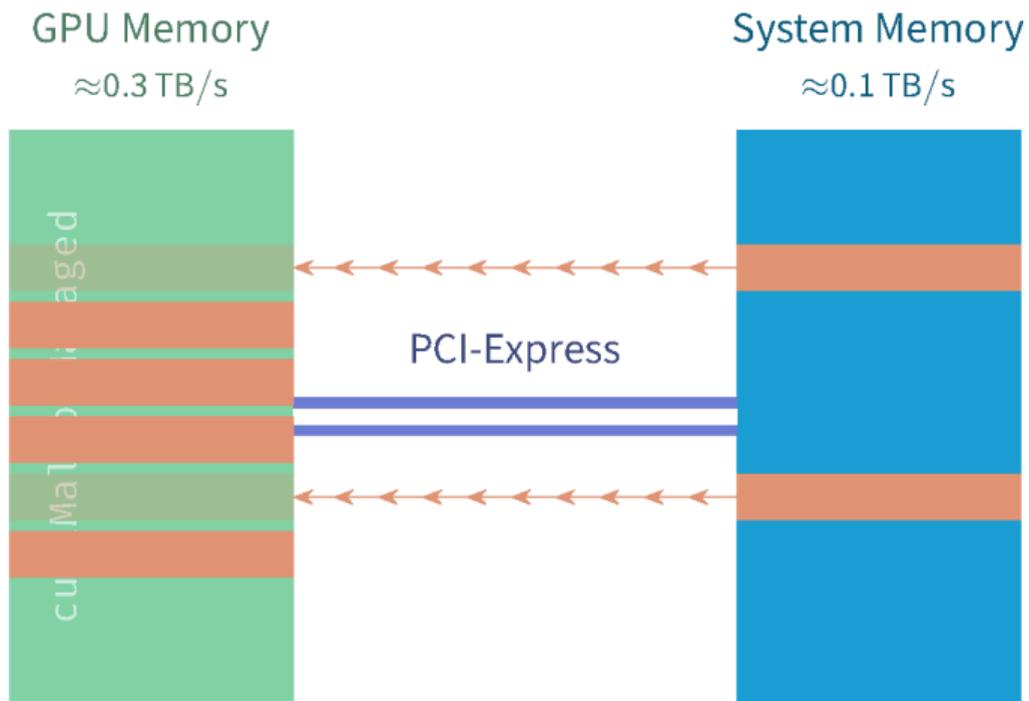
# Migration on Kepler



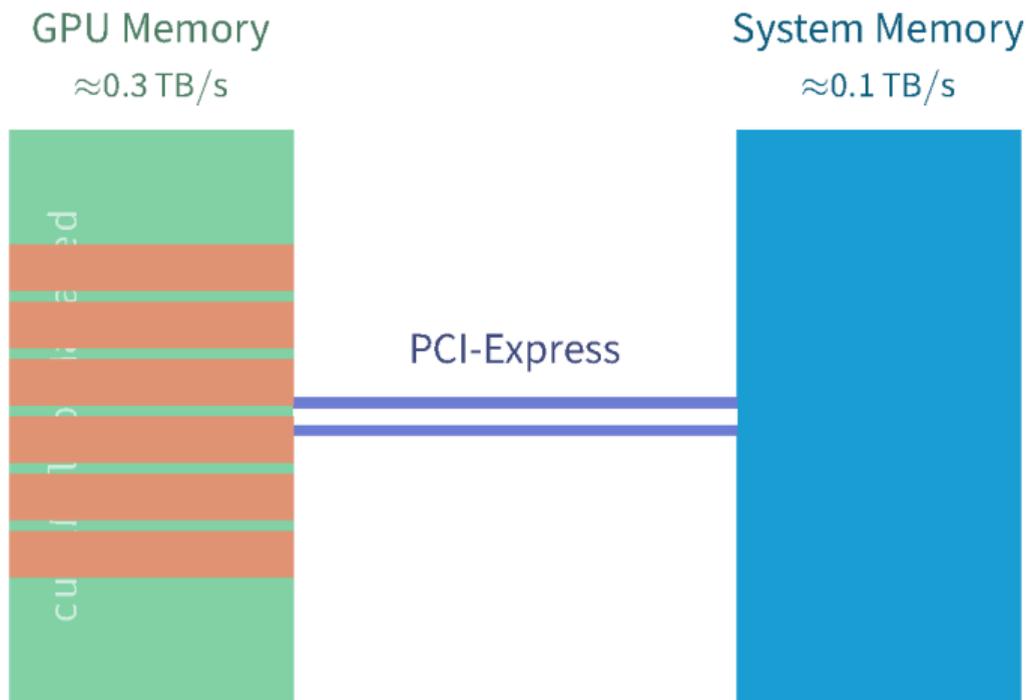
# Migration on Kepler



# Migration on Kepler



# Migration on Kepler



# Implementation before Pascal

Kepler (JURECA), Maxwell, ...

- Pages populate on GPU with `cudaMallocManaged()`
- Might migrate to CPU if touched there first
- Pages migrate in bulk to GPU on kernel launch
- No over-subscription possible

# Practical Differences

## Revisiting `scale_vector_um` Example

# Comparing UM on Pascal & Kepler

Different scales

Comparing `scale_vector_um` on JURON (JUWELS) and JURECA

# Comparing UM on Pascal & Kepler

Different scales

Comparing `scale_vector_um` on JURON (JUWELS) and JURECA

JUWELS

==109924== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	4.9247ms	1	4.9247ms	4.9247ms	4.9247ms	scale(float, float*, float*, int)

JURECA

==12922== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	136.03us	1	136.03us	136.03us	136.03us	scale(float, float*, float*, int)

# Comparing UM on Pascal & Kepler

## Different scales

Comparing `scale_vector_um` on JURON (JUWELS) and JURECA

JURON

==109924== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	1.8203ms	1	1.8203ms	1.8203ms	1.8203ms	scale(float, float*, float*, int)

JURECA

==12922== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	136.03us	1	136.03us	136.03us	136.03us	scale(float, float*, float*, int)

# Comparing UM on Pascal & Kepler

## Different scales

Comparing `scale_vector_um` on JURON (JUWELS) and JURECA

JURON

==109924== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	1.8203ms	1	1.8203ms	1.8203ms	1.8203ms	scale(float, float*, float*, int)

*Why?!*

Shouldn't P100 and V100 be much faster than K80?

JURECA

==12922== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	136.03us	1	136.03us	136.03us	136.03us	scale(float, float*, float*, int)

# Comparing UM on Pascal & Kepler

## Different scales



JURON

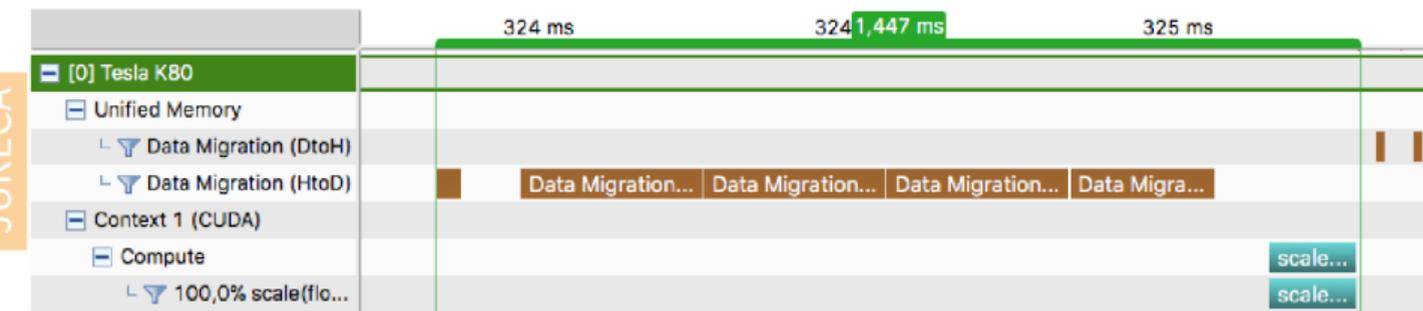
==12922== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	136.03us	1	136.03us	136.03us	136.03us	scale(float, float*, float*, int)

JURECA

# Comparing UM on Pascal & Kepler

## Different scales



# Comparing UM on Pascal & Kepler

What happens?

**JURON** Kernel is launched, data is needed by kernel, data migrates host→device  
⇒ Run time of kernel **incorporates** time for data transfers

**JURECA** Data will be needed by kernel – so data migrates host→device **before** kernel launch  
⇒ Run time of **kernel** without any transfers

# Comparing UM on Pascal & Kepler

What happens?

**JURON** Kernel is launched, data is needed by kernel, data migrates host→device  
⇒ Run time of kernel **incorporates** time for data transfers

**JURECA** Data will be needed by kernel – so data migrates host→device **before** kernel launch  
⇒ Run time of **kernel** without any transfers

- Implementation on Pascal is the more convenient one
- Total run time of whole program does not principally change  
*Except it gets shorter because of faster architecture*
- But data transfers sometimes sorted to kernel launch

# Comparing UM on Pascal & Kepler

What happens?

**JURON** Kernel is launched, data is needed by kernel, data migrates host→device  
⇒ Run time of kernel **incorporates** time for data transfers

**JURECA** Data will be needed by kernel – so data migrates host→device **before** kernel launch  
⇒ Run time of **kernel** without any transfers

- Implementation on Pascal is the more convenient one
- Total run time of whole program does not principally change  
*Except it gets shorter because of faster architecture*
- But data transfers sometimes sorted to kernel launch

⇒ *What can we do about this?*

# Performance Hints for UM

## General hints

- **Keep data local**

Prevent migrations at all if data is processed by close processor

# Performance Hints for UM

## General hints

- **Keep data local**  
Prevent migrations at all if data is processed by close processor
- **Minimize thrashing**  
Constant migrations hurt performance

# Performance Hints for UM

## General hints

- **Keep data local**  
Prevent migrations at all if data is processed by close processor
- **Minimize thrashing**  
Constant migrations hurt performance
- **Minimize page fault overhead**  
Fault handling costs  $\mathcal{O}(10 \mu\text{s})$ , stalls execution



# Performance Hints for UM

## New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`  
Prefetches data to device (on stream) asynchronously

# Performance Hints for UM

## New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync`(data, length, device, stream)  
Prefetches data to device (on stream) asynchronously
- `cudaMemAdvise`(data, length, advice, device)  
Advise about usage of given data, advice:

# Performance Hints for UM

## New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync`(data, length, device, stream)  
Prefetches data to device (on stream) asynchronously
- `cudaMemAdvise`(data, length, advice, device)  
Advise about usage of given data, advice:
  - `cudaMemAdviseSetReadMostly`: Data is mostly read and occasionally written to

# Performance Hints for UM

## New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync`(data, length, device, stream)  
Prefetches data to device (on stream) asynchronously
- `cudaMemAdvise`(data, length, advice, device)  
Advise about usage of given data, advice:
  - `cudaMemAdviseSetReadMostly`: Data is mostly read and occasionally written to
  - `cudaMemAdviseSetPreferredLocation`: Set preferred location to avoid migrations; first access will establish mapping

# Performance Hints for UM

## New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`  
Prefetches data to device (on stream) asynchronously
- `cudaMemAdvise(data, length, advice, device)`  
Advise about usage of given data, advice:
  - `cudaMemAdviseSetReadMostly`: Data is mostly read and occasionally written to
  - `cudaMemAdviseSetPreferredLocation`: Set preferred location to avoid migrations; first access will establish mapping
  - `cudaMemAdviseSetAccessedBy`: Data is accessed by *this* device; will pre-map data to avoid page fault

# Performance Hints for UM

## New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`  
Prefetches data to device (on stream) asynchronously
- `cudaMemAdvise(data, length, advice, device)`  
Advise about usage of given data, advice:
  - `cudaMemAdviseSetReadMostly`: Data is mostly read and occasionally written to
  - `cudaMemAdviseSetPreferredLocation`: Set preferred location to avoid migrations; first access will establish mapping
  - `cudaMemAdviseSetAccessedBy`: Data is accessed by *this* device; will pre-map data to avoid page fault
- Use `cudaCpuDeviceId` for device CPU, or use `cudaGetDevice()` as usual to retrieve current GPU device id (default: 0)

# Hints in Code

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    // ...  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    cudaMemcpyPrefetchAsync(data, N, device);  
    kernel<<<...>>>(data, N);  
    cudaDeviceSynchronize();  
  
    host_func(data);  
    cudaFree(data); }  
}
```

# Hints in Code

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    // ...  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    cudaMemcpyPrefetchAsync(data, N, device);  
    kernel<<<...>>>(data, N);  
    cudaDeviceSynchronize();  
  
    host_func(data);  
    cudaFree(data); }  
    
```

Prefetch data to avoid expensive GPU page faults

# Hints in Code

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    // ...  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    cudaMemAdvise(data, N, cudaMemAdviseSetReadMostly, device);  
    cudaMemPrefetchAsync(data, N, device);  
    kernel<<<...>>>(data, N);  
    cudaDeviceSynchronize();  
  
    host_func(data);  
    cudaFree(data); }  
}
```

Read-only copy of data  
is created on GPU during  
prefetch  
→ CPU and GPU reads will  
not fault

Prefetch data to avoid ex-  
pensive GPU page faults

# Tuning scale\_vector\_um

## Express data movement

TASK

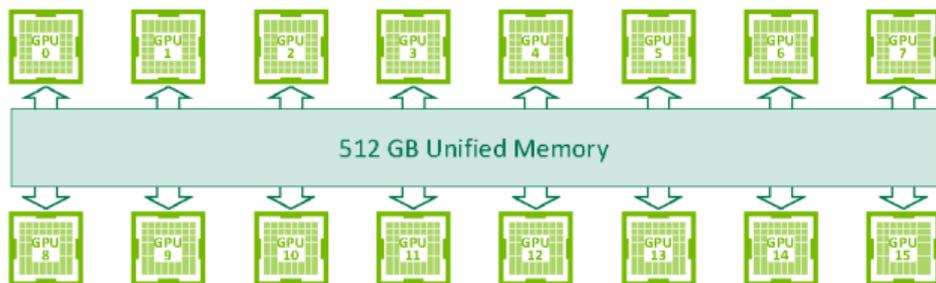
- Location of code: 3-Unified-Memory/exercises/tasks/scale/
- Look at Instructions.md for instructions
  - 1 Show runtime that data should be migrated to GPU before kernel call
  - 2 Build with make
  - 3 Run with make run  
`Or srun --gres=gpu -p gpus ./scale_vector_um`
  - 4 Generate profile to study your progress – see make profile
- See also [CUDA C programming guide](#) for details on data usage

*Finished early? There's one more task in the appendix!*

# Conclusions

## What we've learned

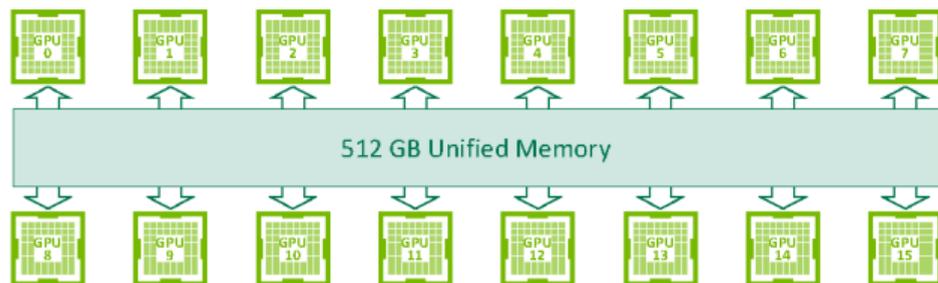
- **Unified Memory** is *productive* feature for GPU programming
- Unified Memory is implemented differently on Pascal (JURON) and Kepler (JURECA)
- With CUDA 8.0, there are new API calls to express **data locality**
- CUDA 9.x and DGX-2: `cudaMalloc()` across all GPUs, then `cudaMemAdviseSetPreferredHome`



# Conclusions

## What we've learned

- **Unified Memory** is *productive* feature for GPU programming
- Unified Memory is implemented differently on Pascal (JURON) and Kepler (JURECA)
- With CUDA 8.0, there are new API calls to express **data locality**
- CUDA 9.x and DGX-2: `cudaMalloc()` across all GPUs, then `cudaMemAdviseSetPreferredHome`



Thank you  
for your attention!  
[a.herten@fz-juelich.de](mailto:a.herten@fz-juelich.de)

Appendix  
Jacobi Task  
Glossary

# Jacobi Task

One more time...

TASK

- Location of code: `3-Unified-Memory/exercises/tasks/jacobi/`
- See Jiri Kraus' slides on Unified Memory from 2016 at `3-Unified-Memory/exercises/slides/jkraus-unified_memory-2016.pdf`
- Short instructions
  - Avoid data migrations in while loop of Jacobi solver: apply boundary conditions with provided GPU kernel; try to avoid remaining migrations
  - Build with `make` (CUDA needs to be loaded!)
  - Run with `make run`
  - Look at profile – see `make profile`

# Glossary I

**CUDA** Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 4, 5, 6, 7, 8, 71, 72, 73

**JSC** Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 76

**JURECA** A multi-purpose supercomputer with 1800 nodes at JSC. 56, 57, 58, 72, 73

**JURON** One of the two HBP pilot system in Jülich; name derived from Juelich and Neuron. 50, 51, 52, 53, 54, 55, 56, 57, 58, 72, 73

**JUWELS** Jülich's new supercomputer, the successor of JUQUEEN. 50, 51, 52, 53, 54, 55

**NVIDIA** US technology company creating GPUs. 76, 77

## Glossary II

**NVLink** NVIDIA's communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with high bandwidth. 77

**P100** A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast HBM2 memory. 53

**Pascal** GPU architecture from NVIDIA (announced 2016). 4, 5, 6, 7, 8, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 56, 57, 58, 77

**V100** A large GPU with the Volta architecture from NVIDIA. It employs NVLink 2 as its interconnect and has fast HBM2 memory. Additionally, it features Tensorcores for Deep Learning and Independent Thread Scheduling. 53

**Volta** GPU architecture from NVIDIA (announced 2017). 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 77

# Glossary III

**CPU** Central Processing Unit. 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 48, 62, 63, 64, 65, 66, 67, 70, 77

**GPU** Graphics Processing Unit. 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 48, 62, 63, 64, 65, 66, 67, 69, 70, 71, 72, 73, 75, 76, 77

**HBP** Human Brain Project. 76

# References: Images, Graphics I

- [1] Martin Oslic. *Bug*. Freely available at Unsplash. URL: <https://unsplash.com/photos/Qi93Pl5vDRw>.
- [2] Glenn Dearth and Vyes Venkataraman. *Picture: DGX-2 Memory Layout*. GTC18 Talk: S8688 – INSIDE DGX-2. 2018. URL: <http://on-demand.gputechconf.com/gtc/2018/presentation/s8688-extending-the-connectivity-and-reach-of-the-gpu.pdf>.