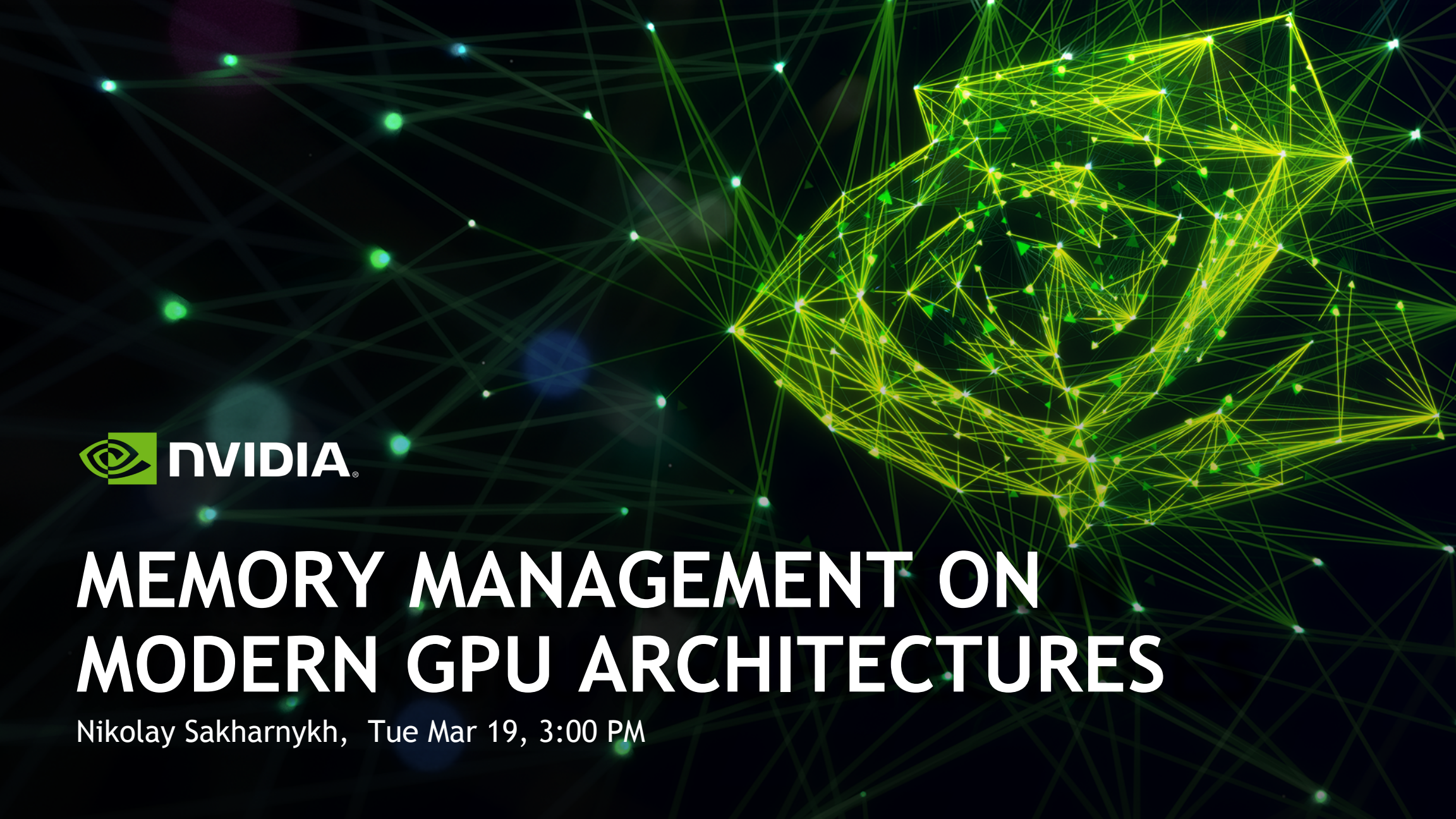
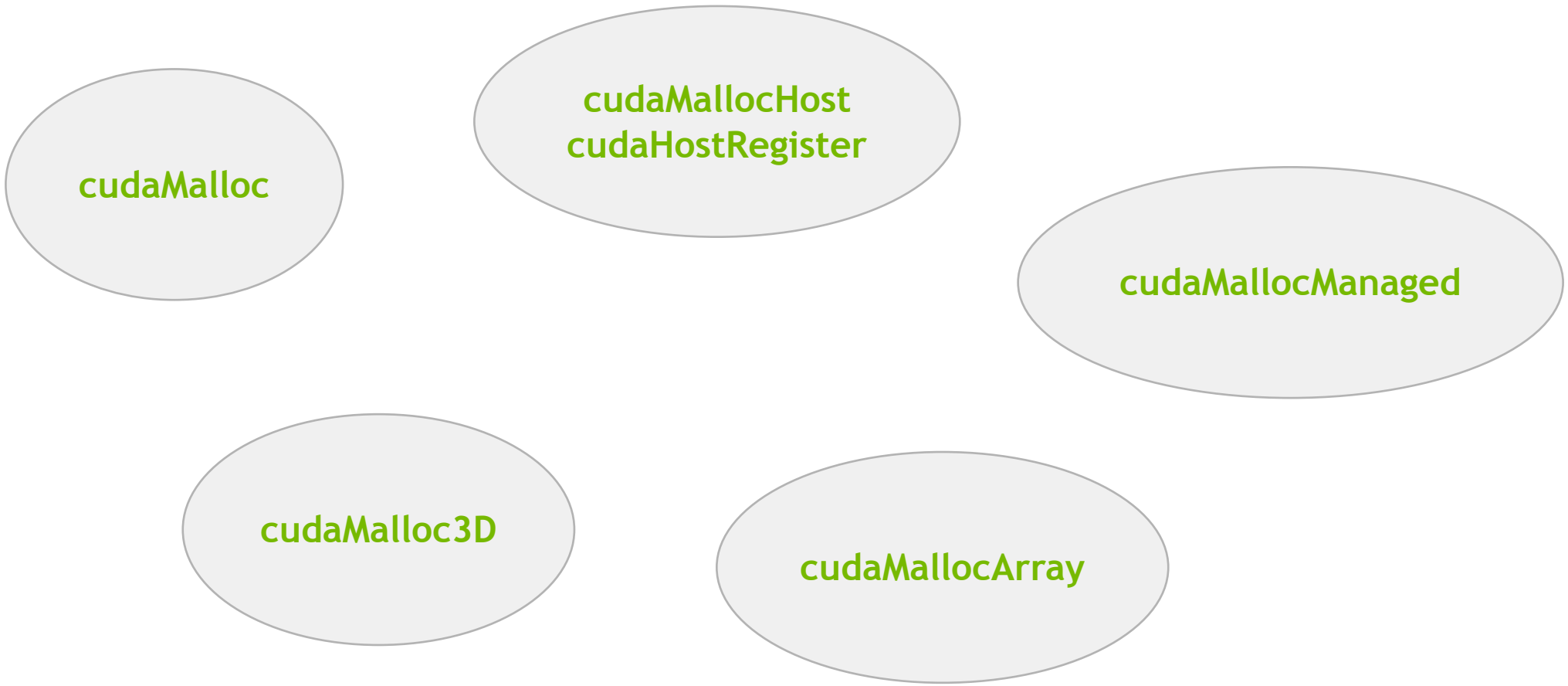


NVIDIA

MEMORY MANAGEMENT ON MODERN GPU ARCHITECTURES

Nikolay Sakharnykh, Tue Mar 19, 3:00 PM

HOW DO WE ALLOCATE MEMORY IN CUDA?



HOW DO WE ALLOCATE MEMORY IN CUDA?

cudaMalloc

- Accessible by GPU only
- Pinned to single GPU

cudaMallocHost
cudaHostRegister

- Accessible by CPU & GPU
- Pinned to CPU mem node

cudaMallocManaged

- Accessible by CPU & GPU
- Can “migrate”

AGENDA

Key principles

Performance tuning

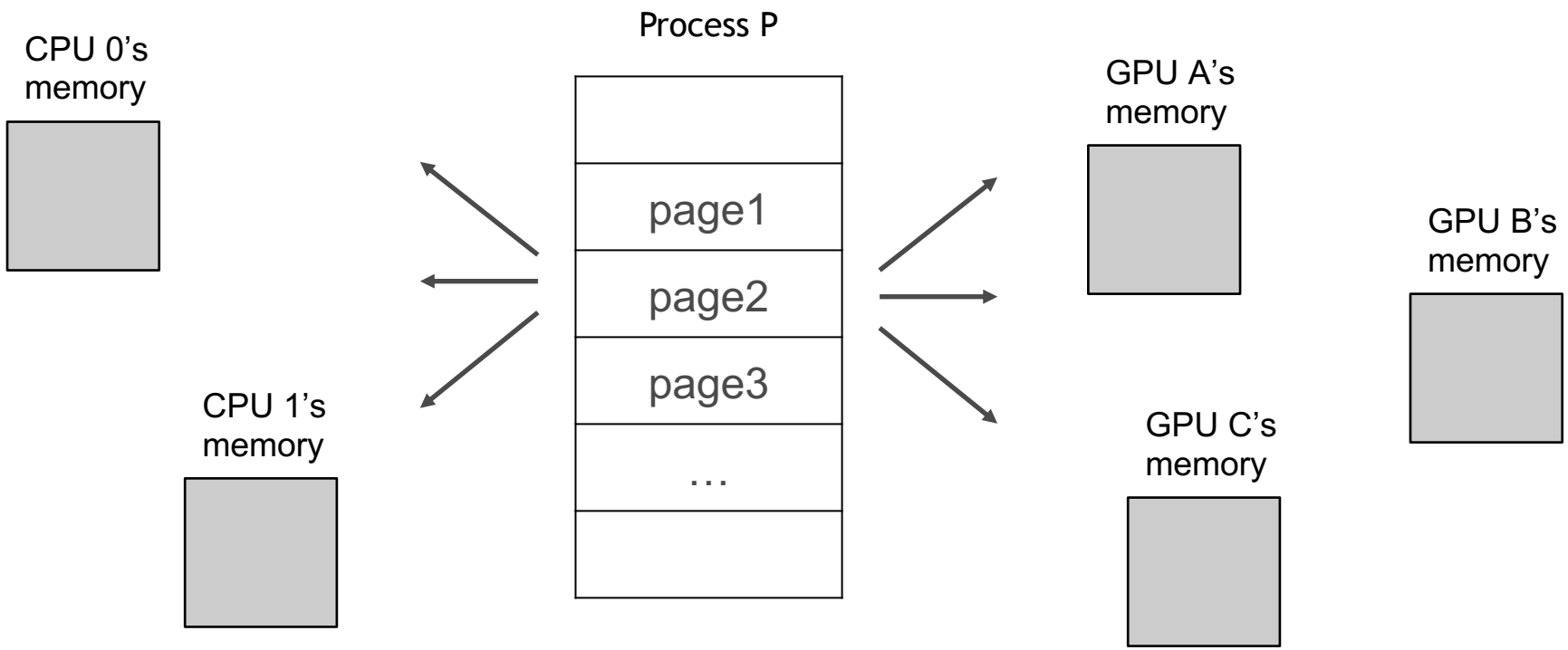
Multi-GPU systems

Summit & Sierra

OS integration

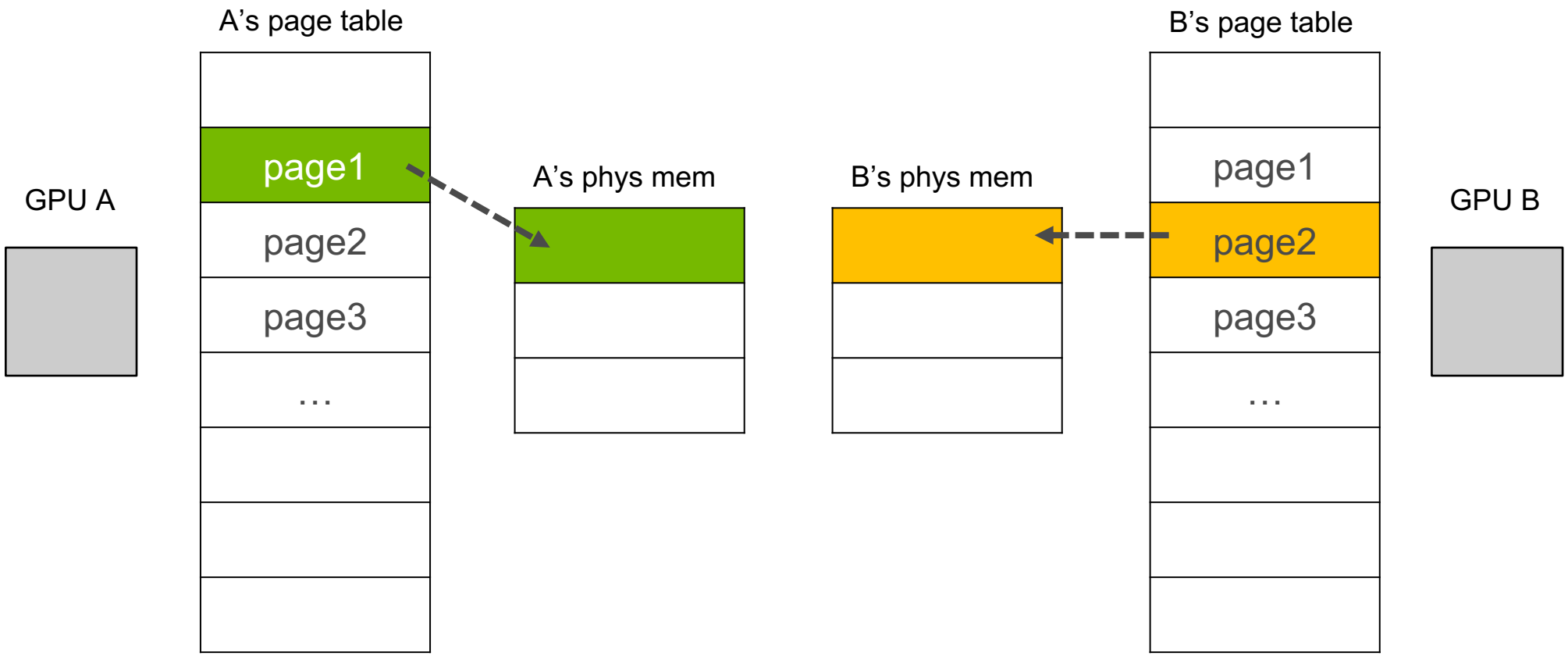
**Here is some behavior that may change in the future*

UNIFIED MEMORY BASICS

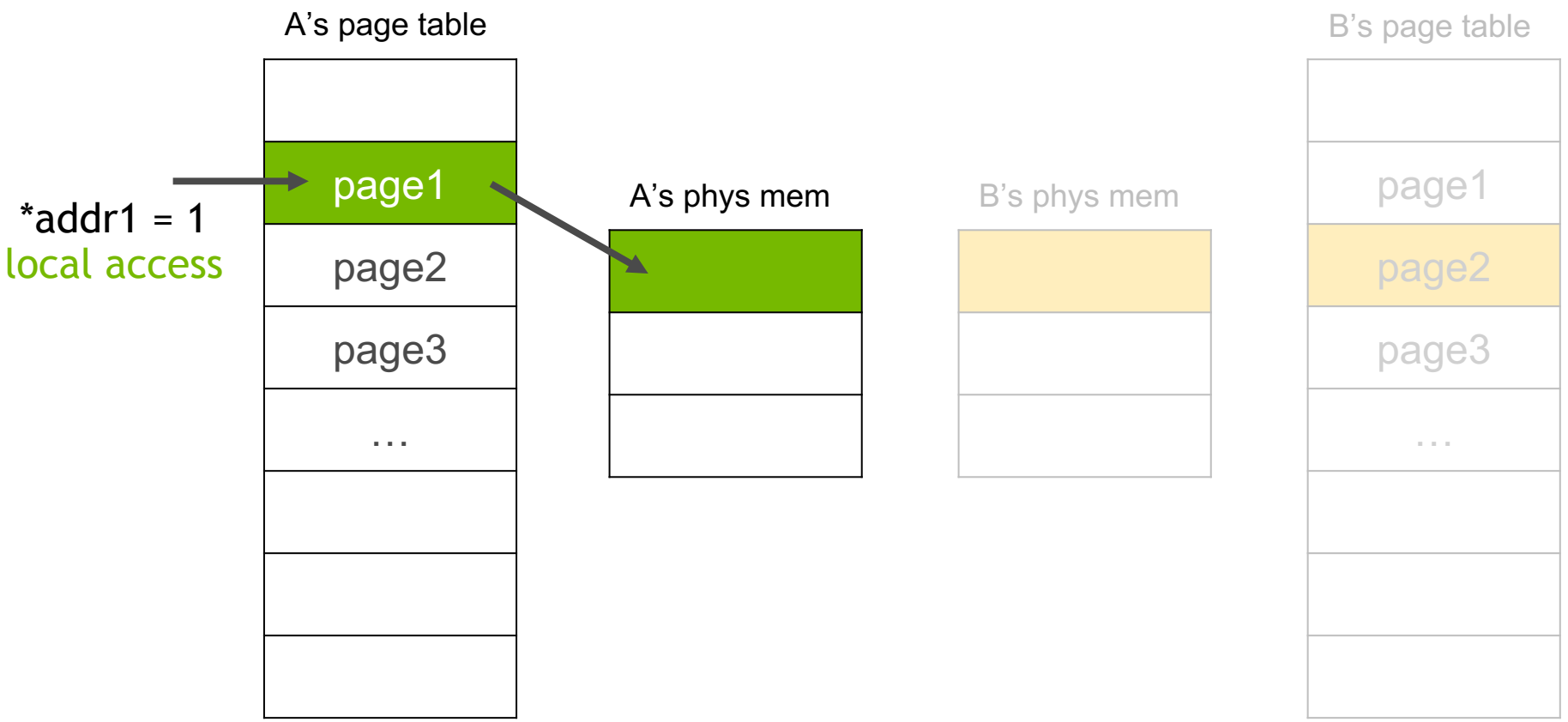


Single virtual memory shared between computing *processors*

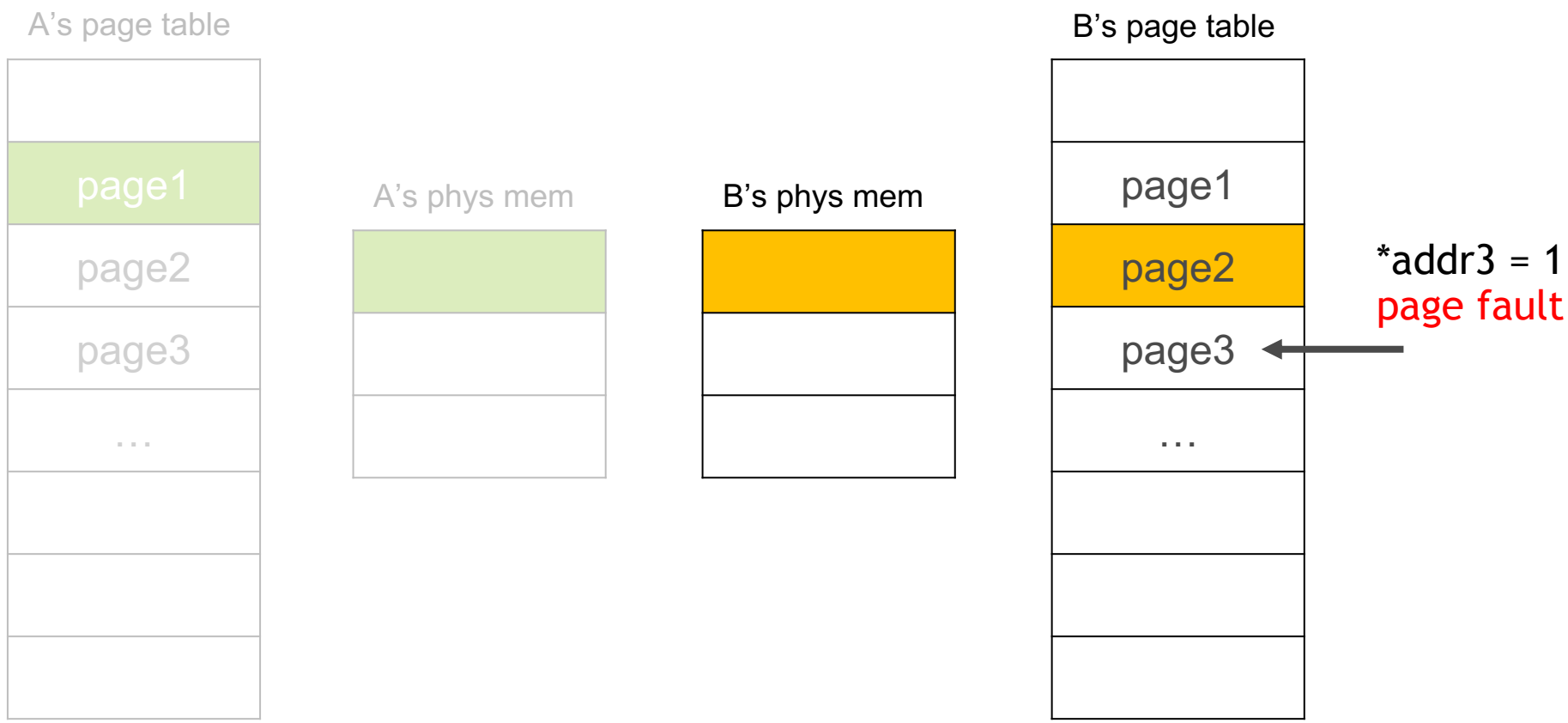
UNIFIED MEMORY BASICS



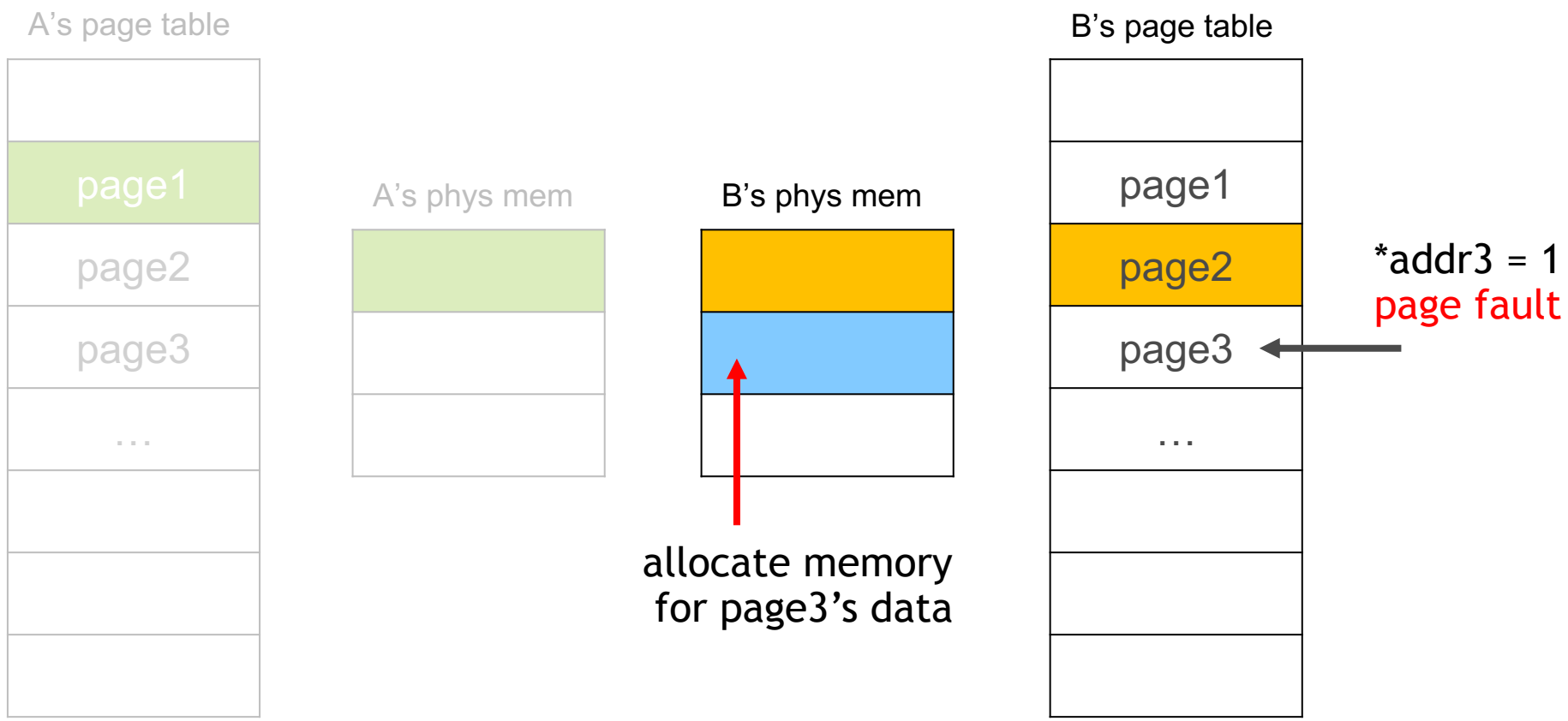
EXAMPLE: LOCAL ACCESS



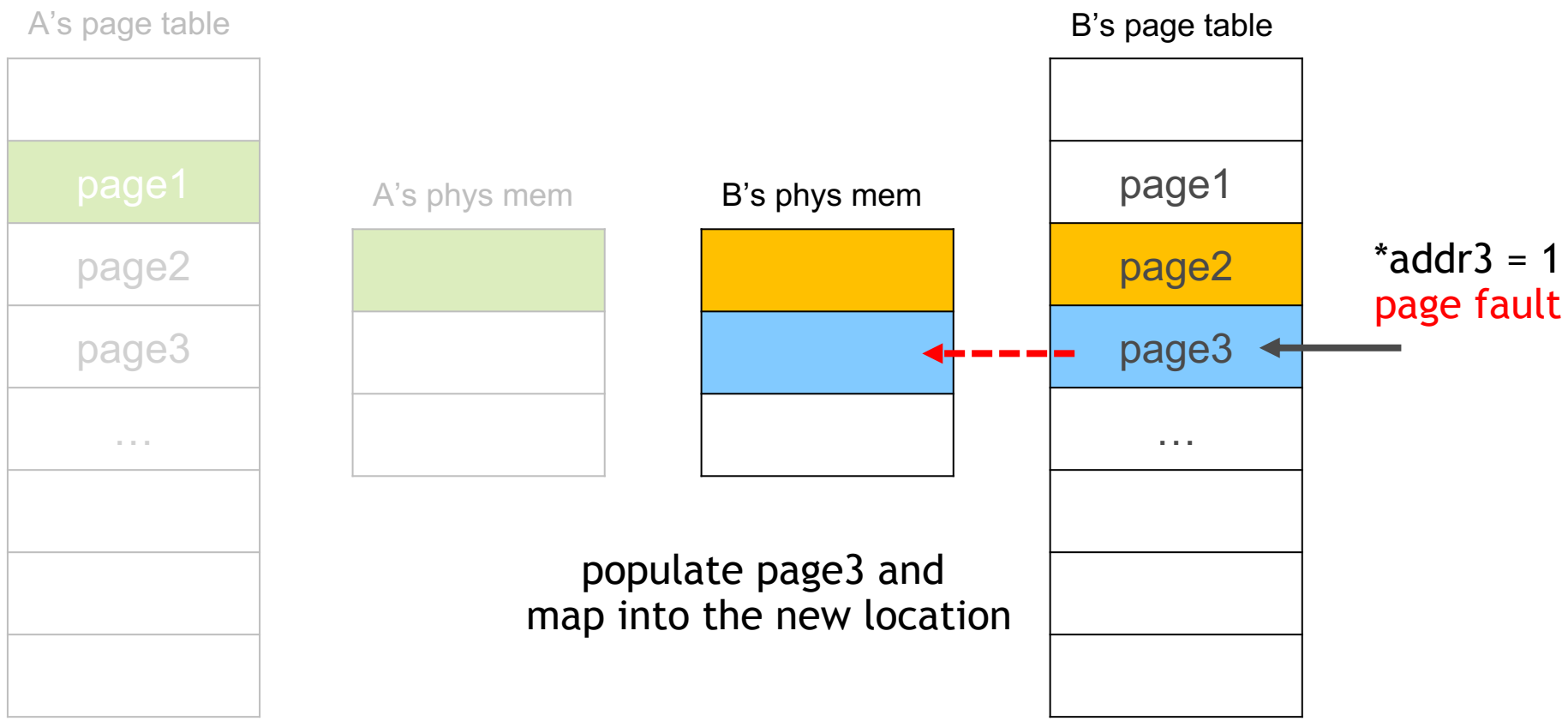
EXAMPLE: POPULATE



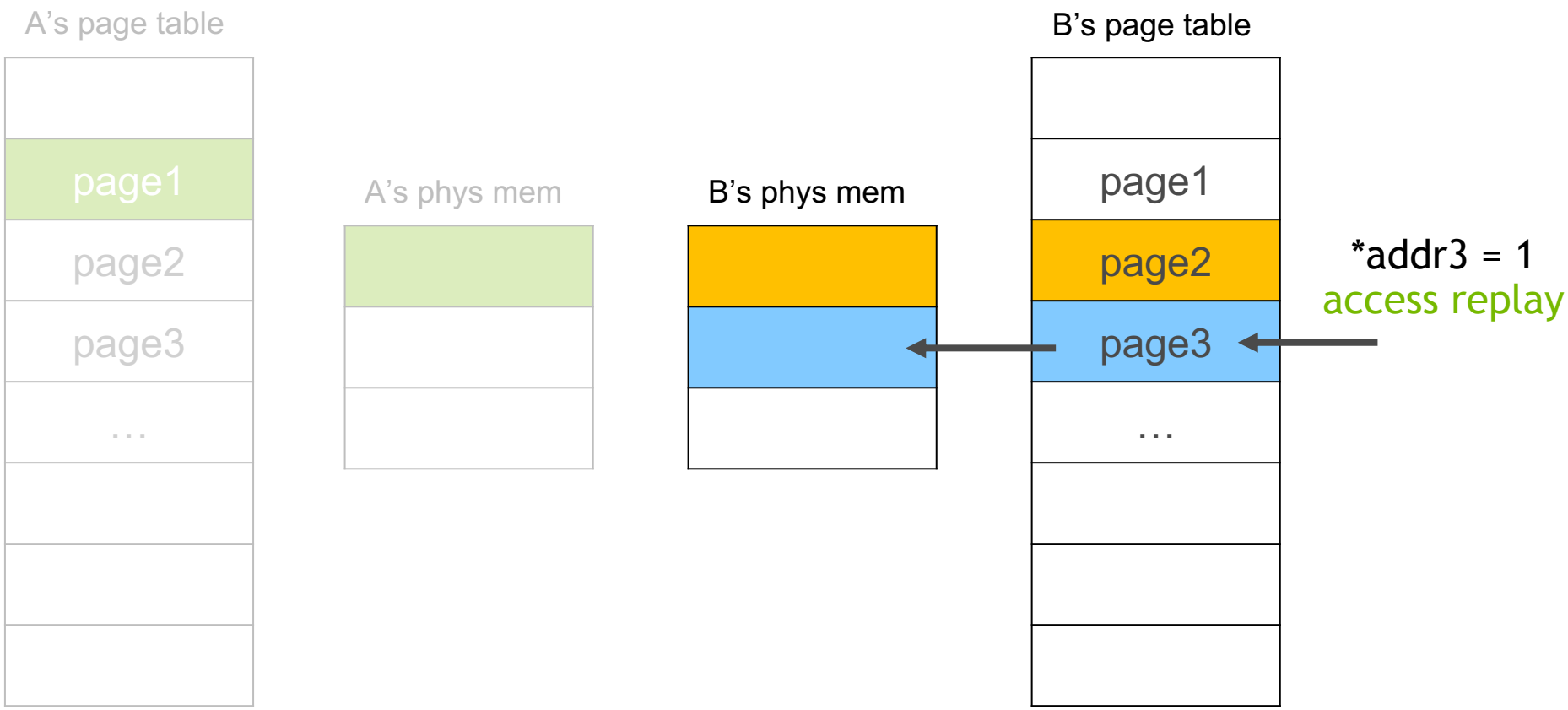
EXAMPLE: POPULATE



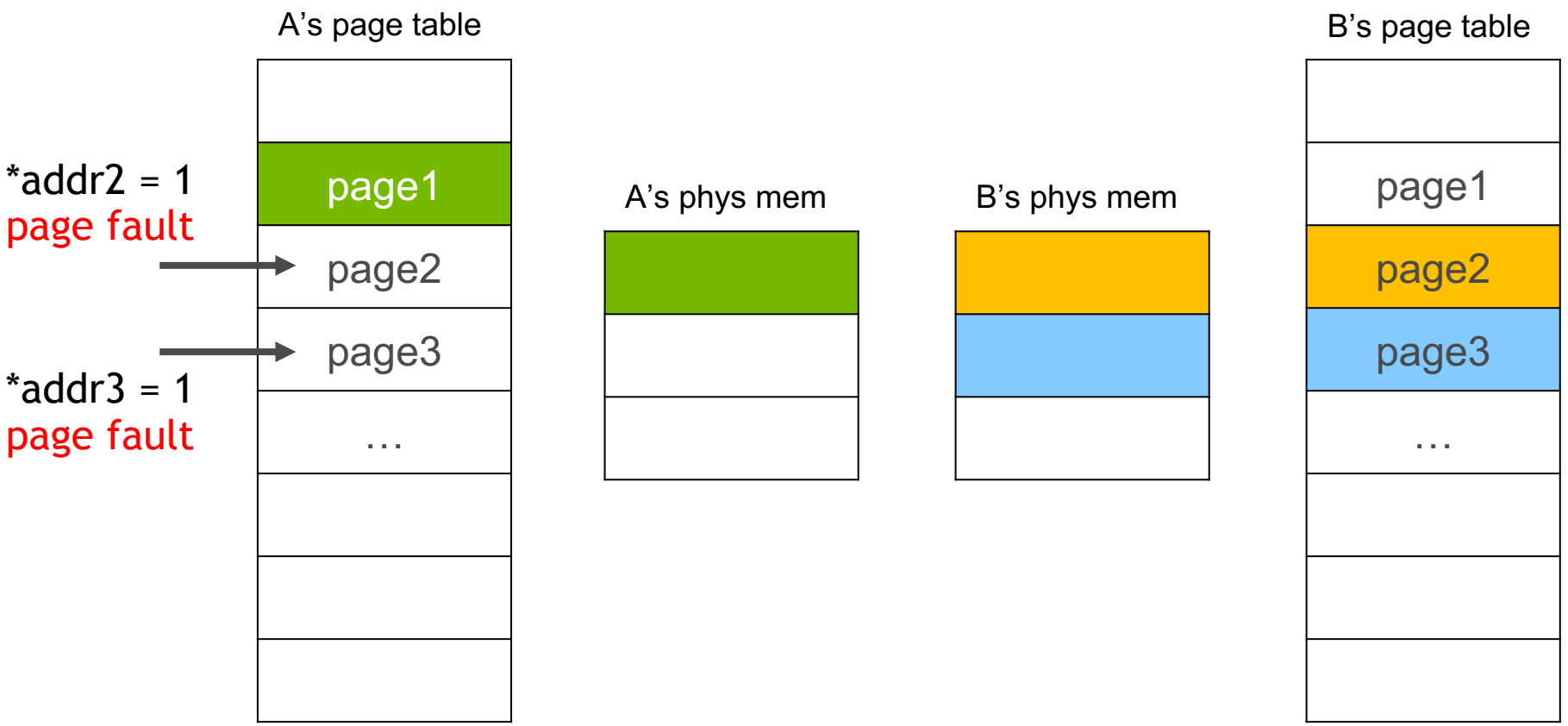
EXAMPLE: POPULATE



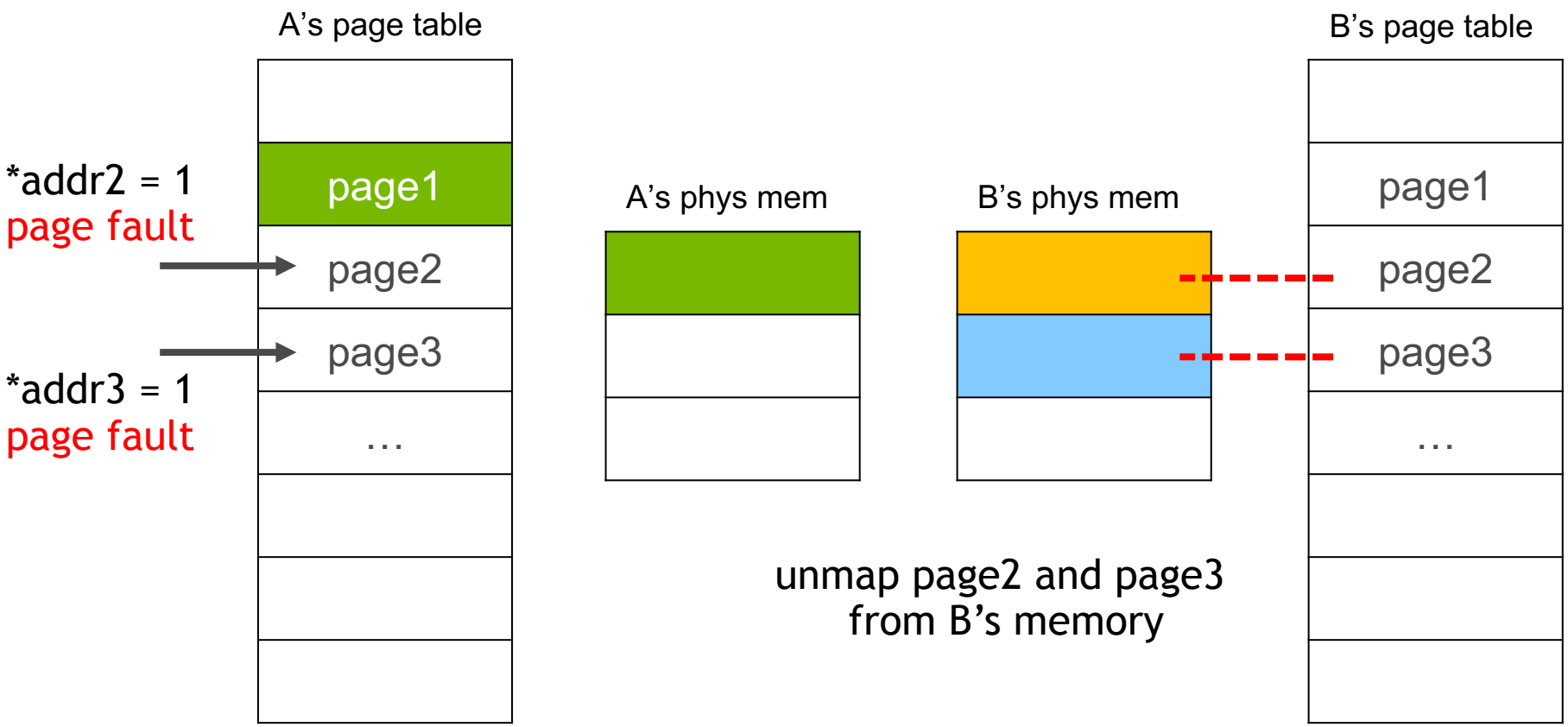
EXAMPLE: POPULATE



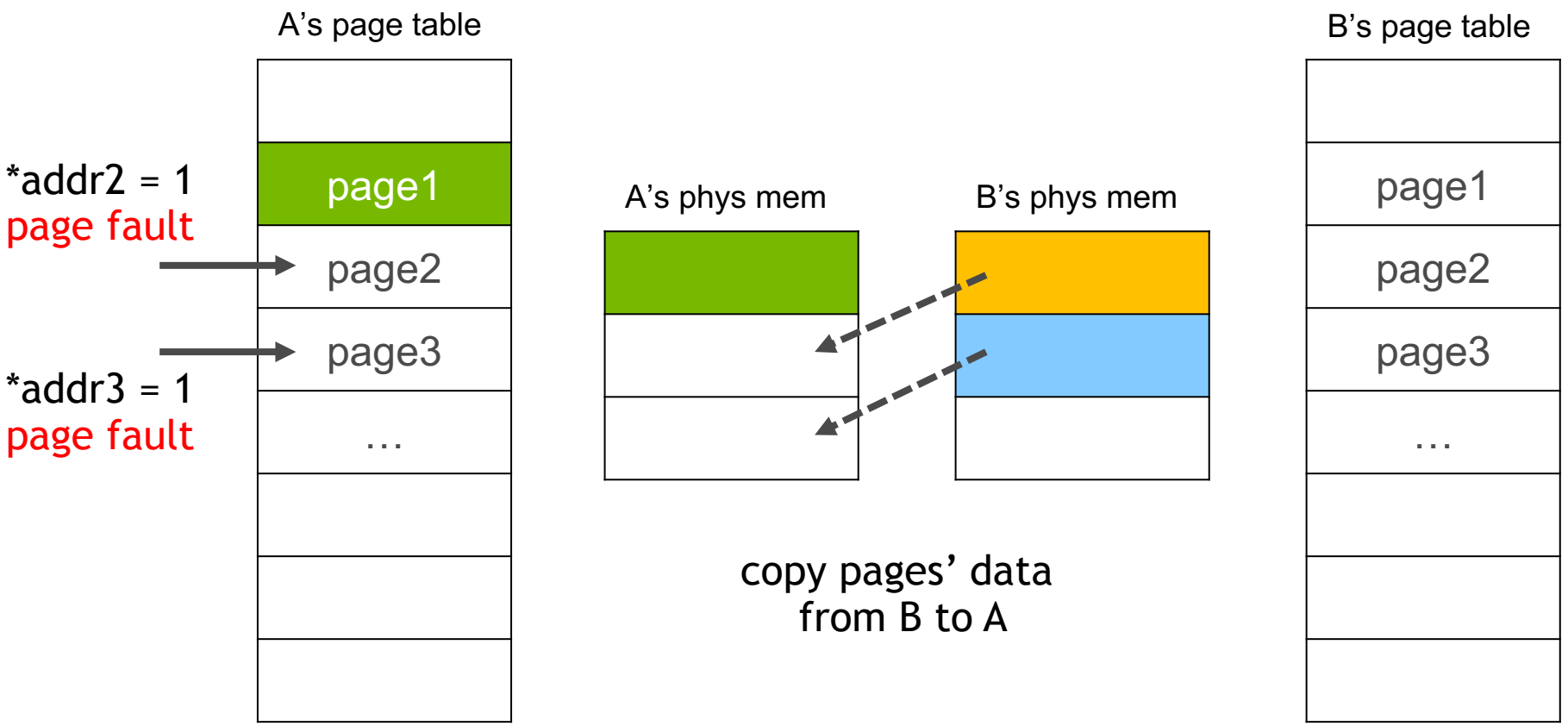
EXAMPLE: MIGRATE



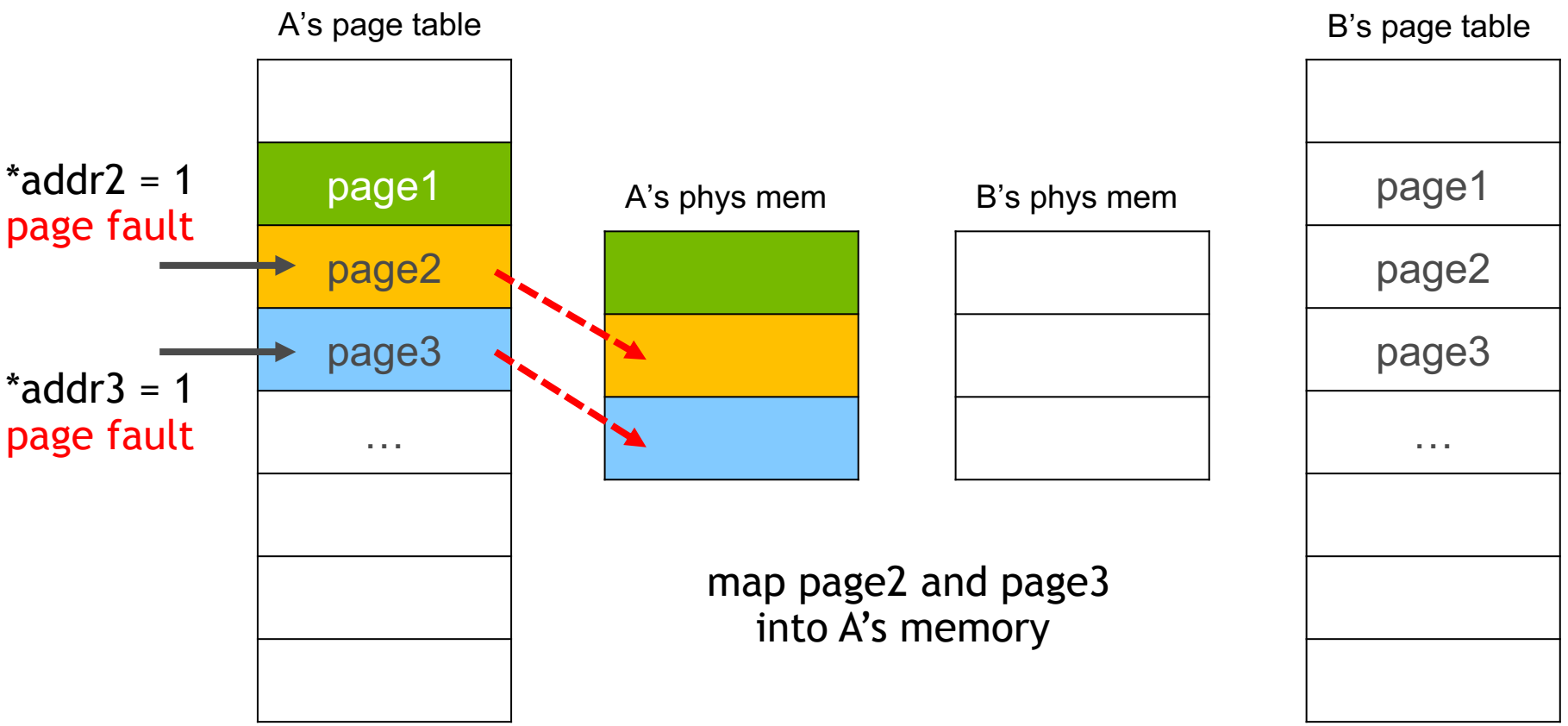
EXAMPLE: MIGRATE



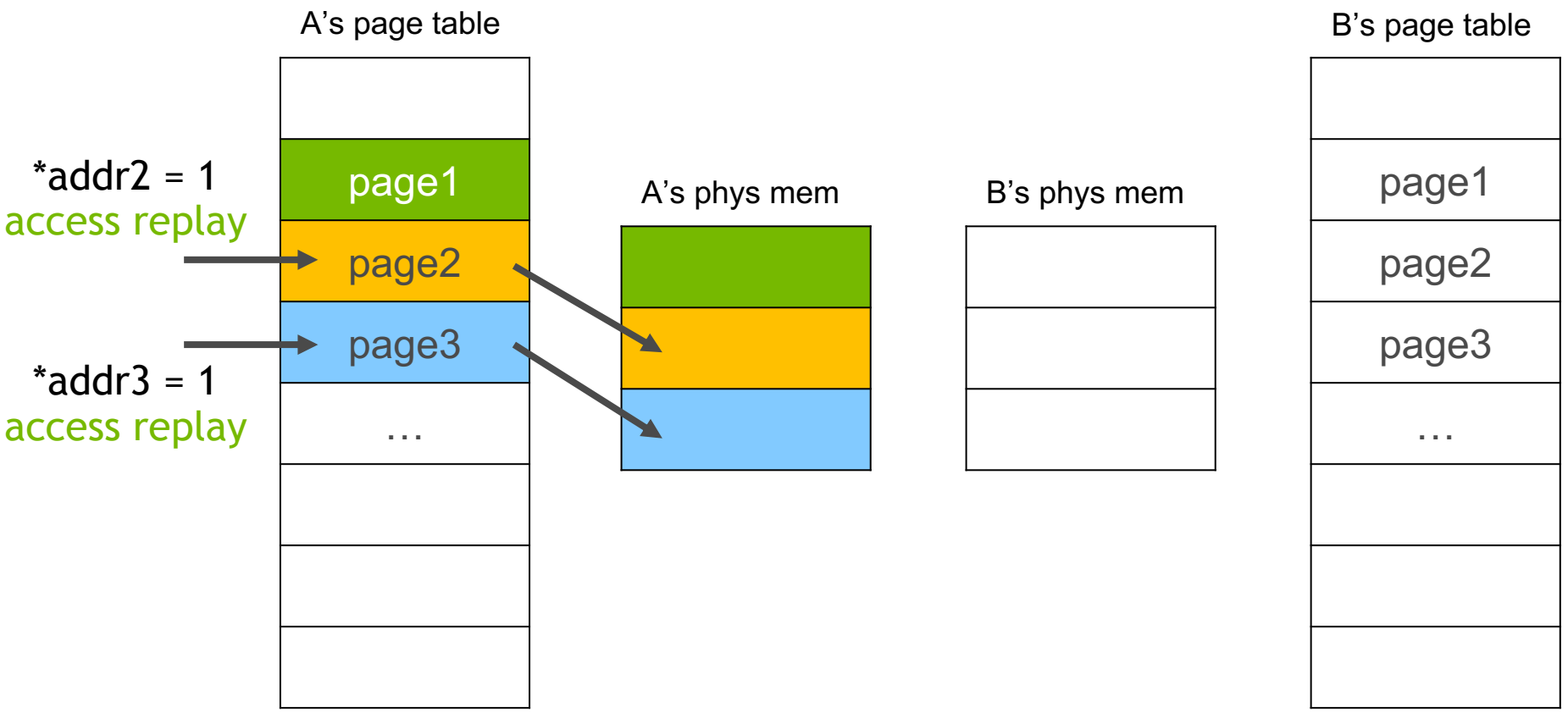
EXAMPLE: MIGRATE



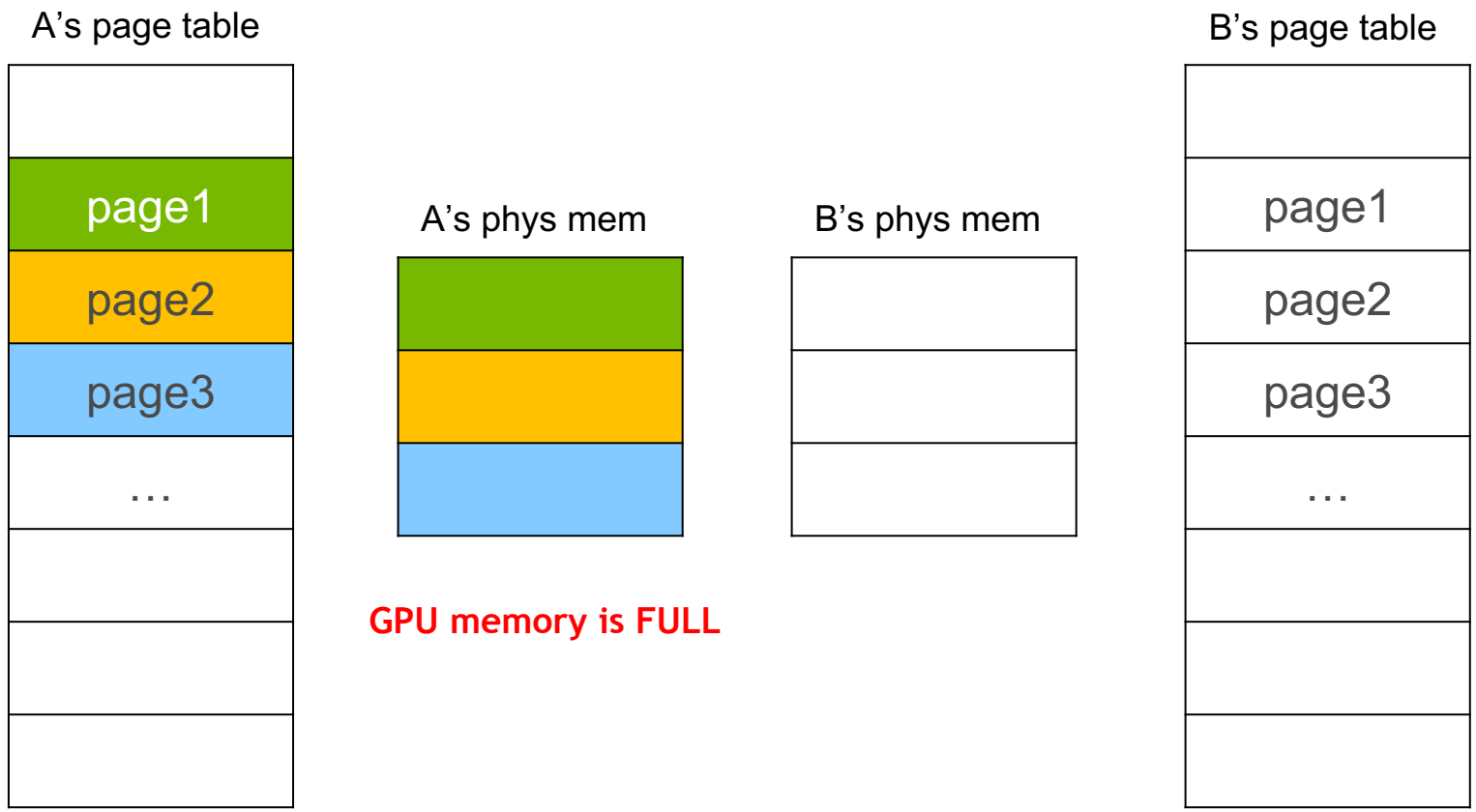
EXAMPLE: MIGRATE



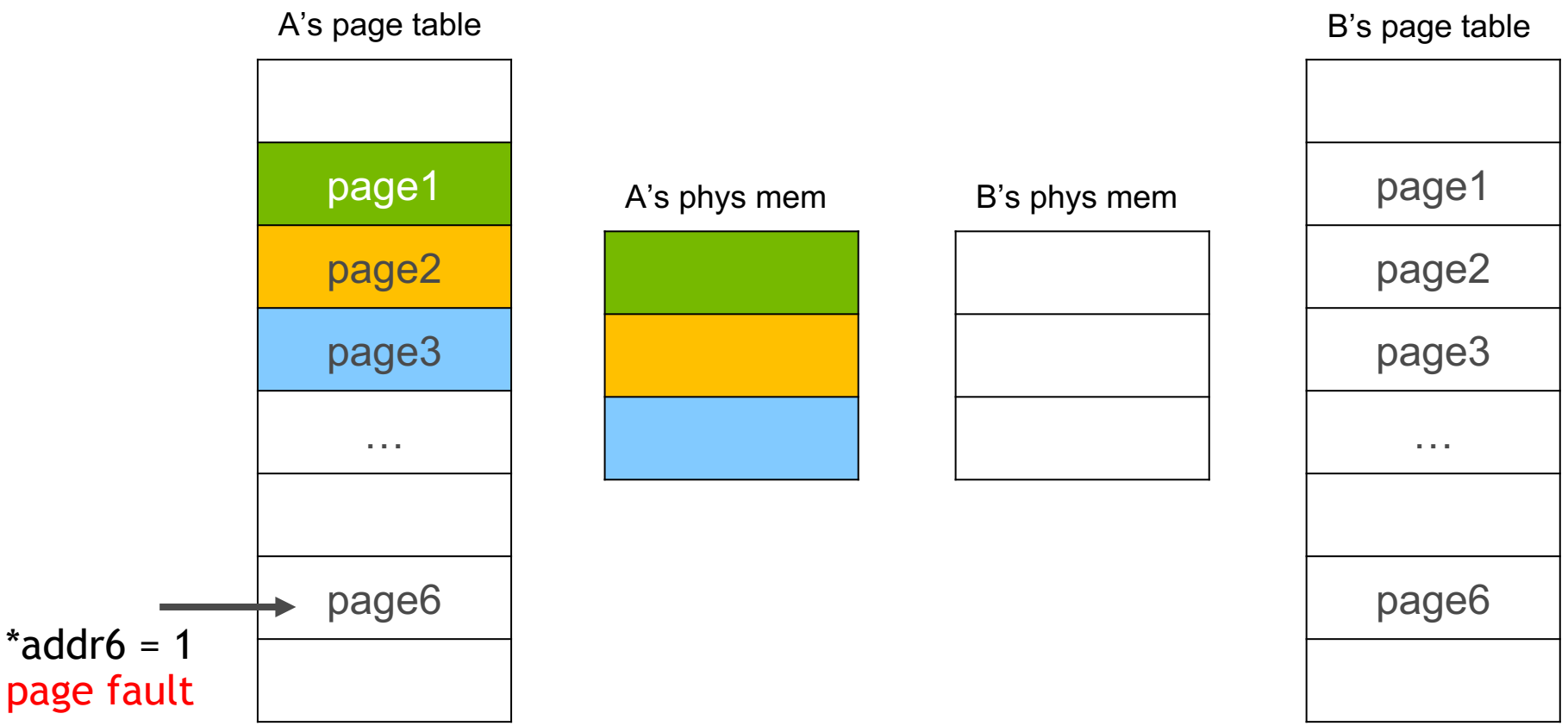
EXAMPLE: MIGRATE



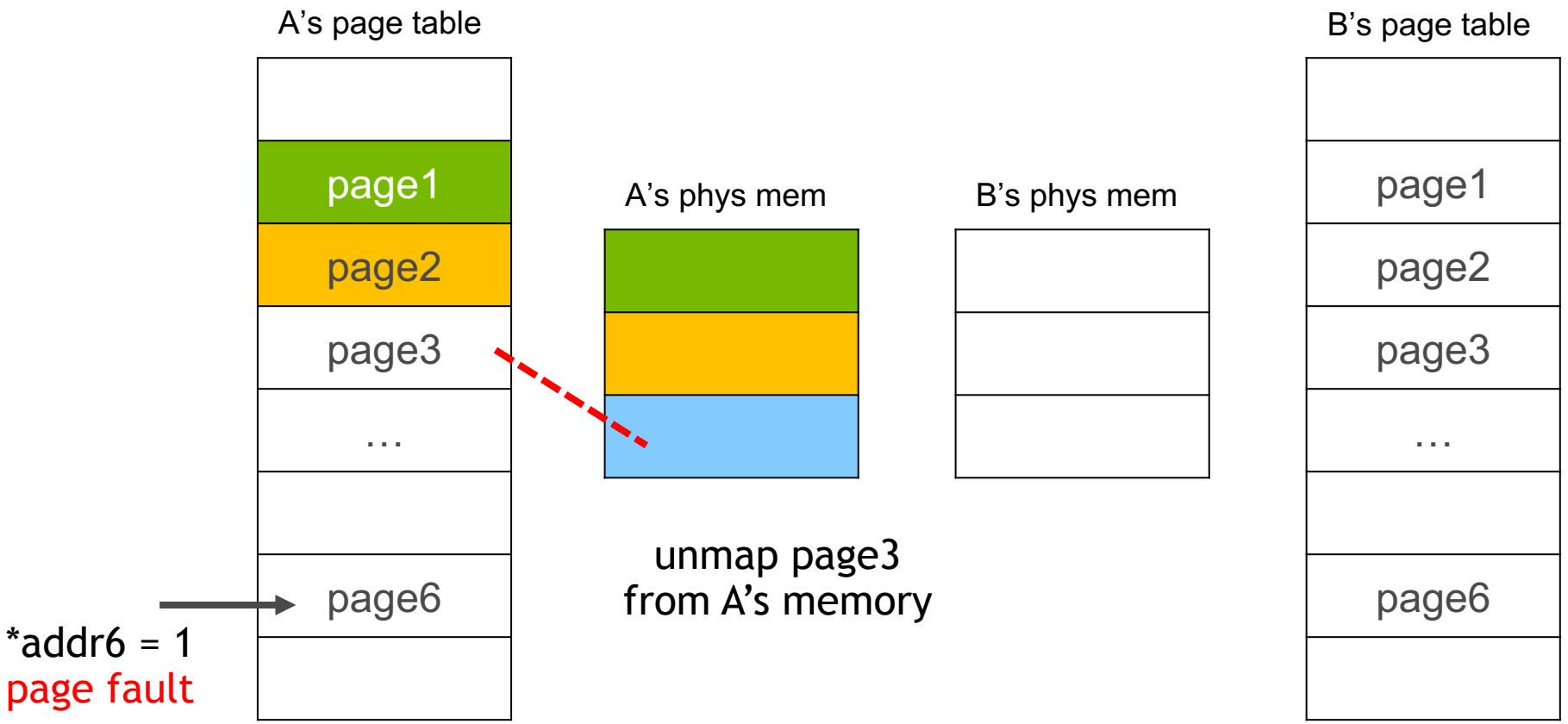
EXAMPLE: OVERSUBSCRIBE



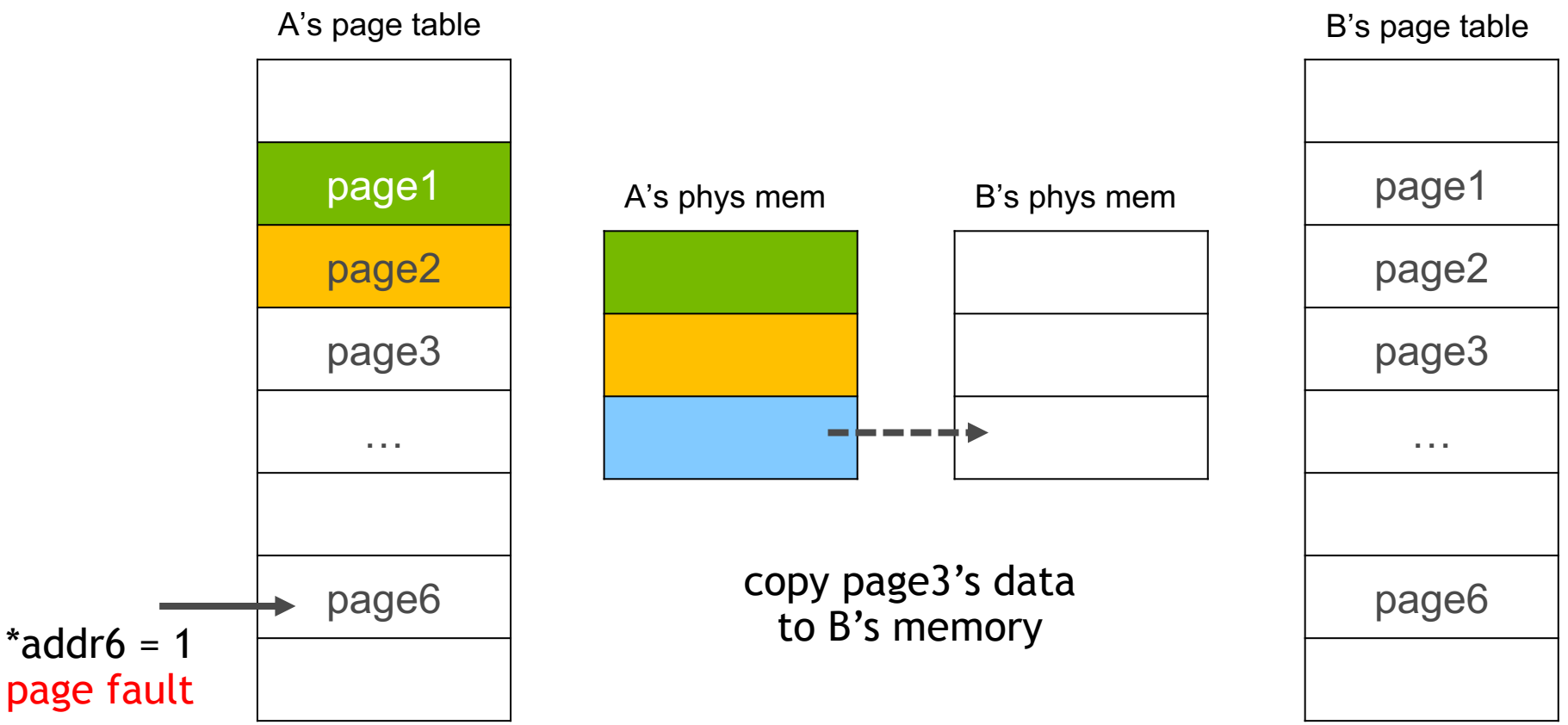
EXAMPLE: OVERSUBSCRIBE



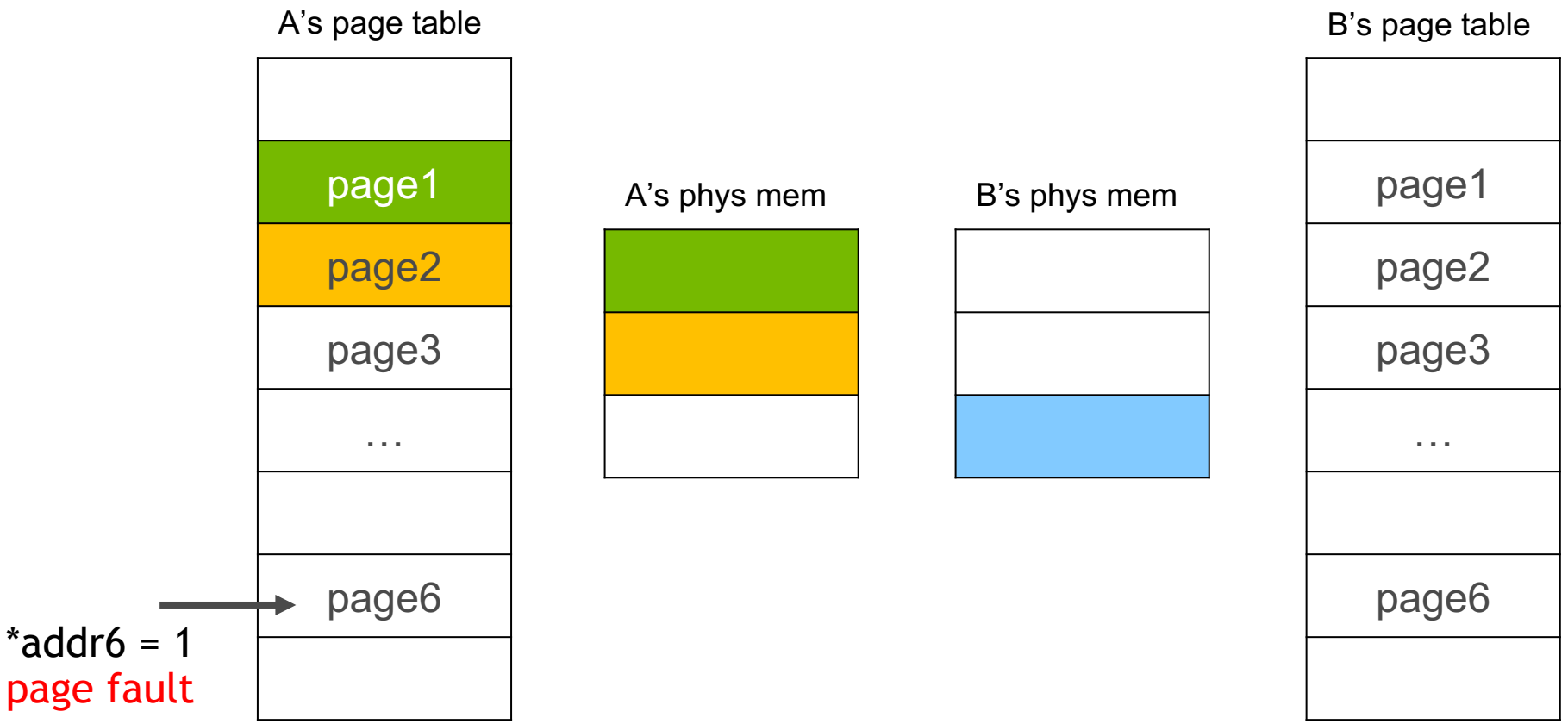
EXAMPLE: OVERSUBSCRIBE



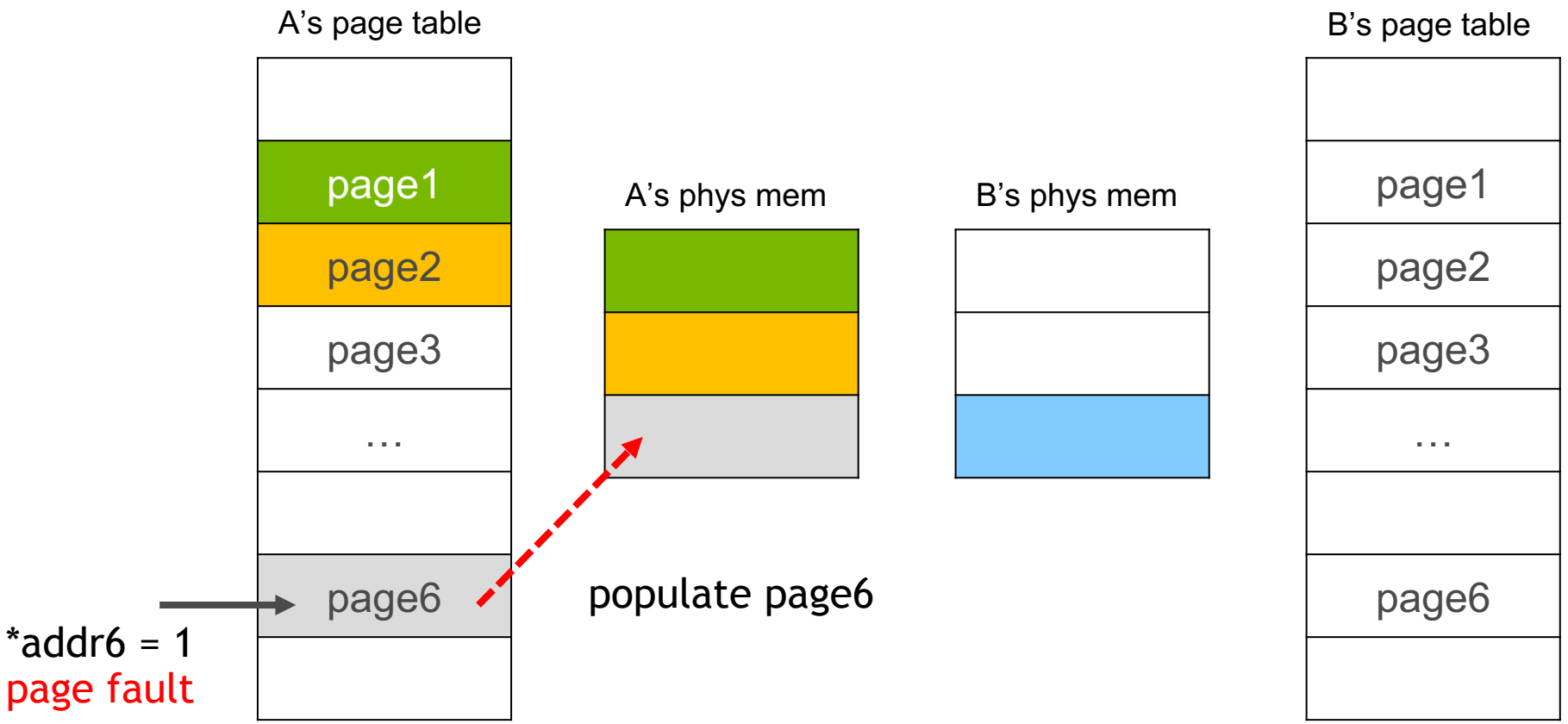
EXAMPLE: OVERSUBSCRIBE



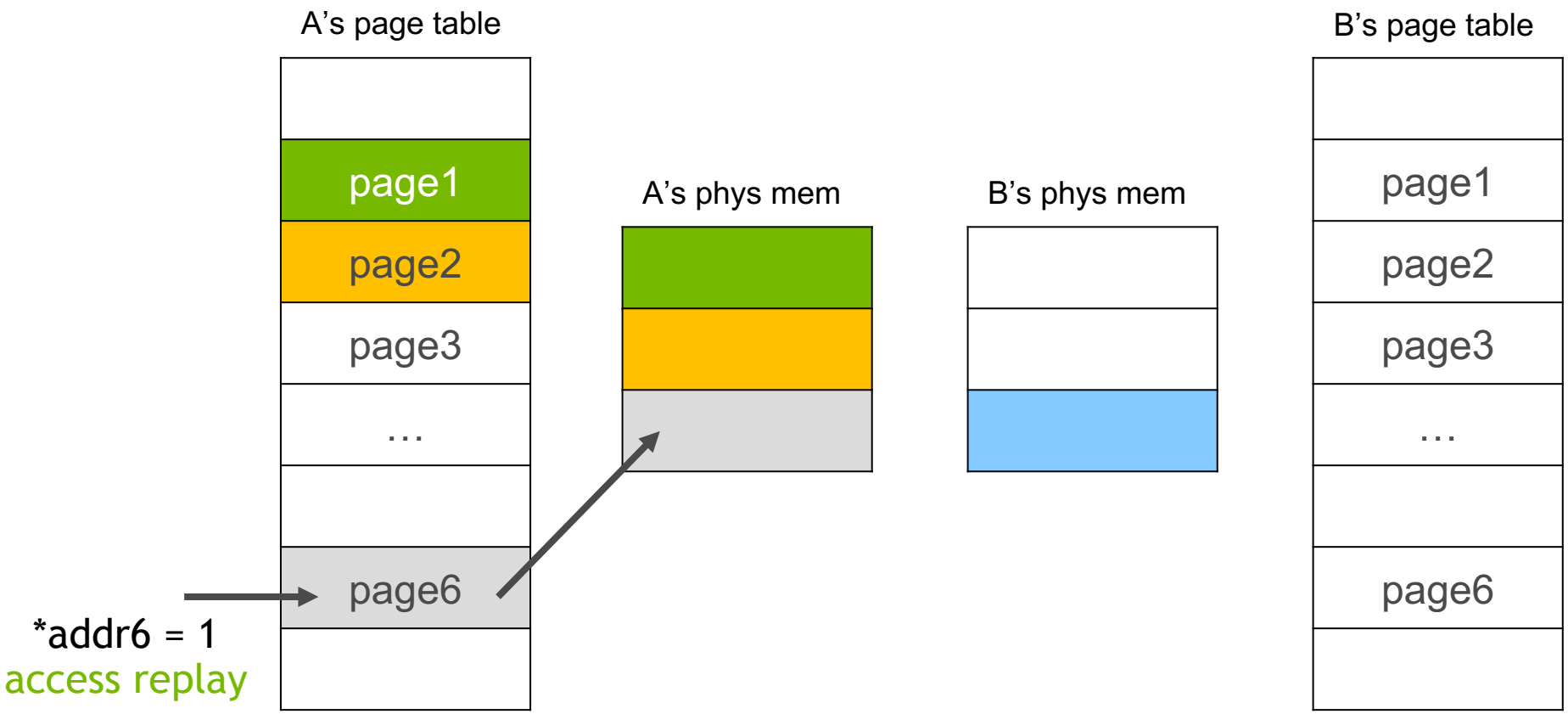
EXAMPLE: OVERSUBSCRIBE



EXAMPLE: OVERSUBSCRIBE

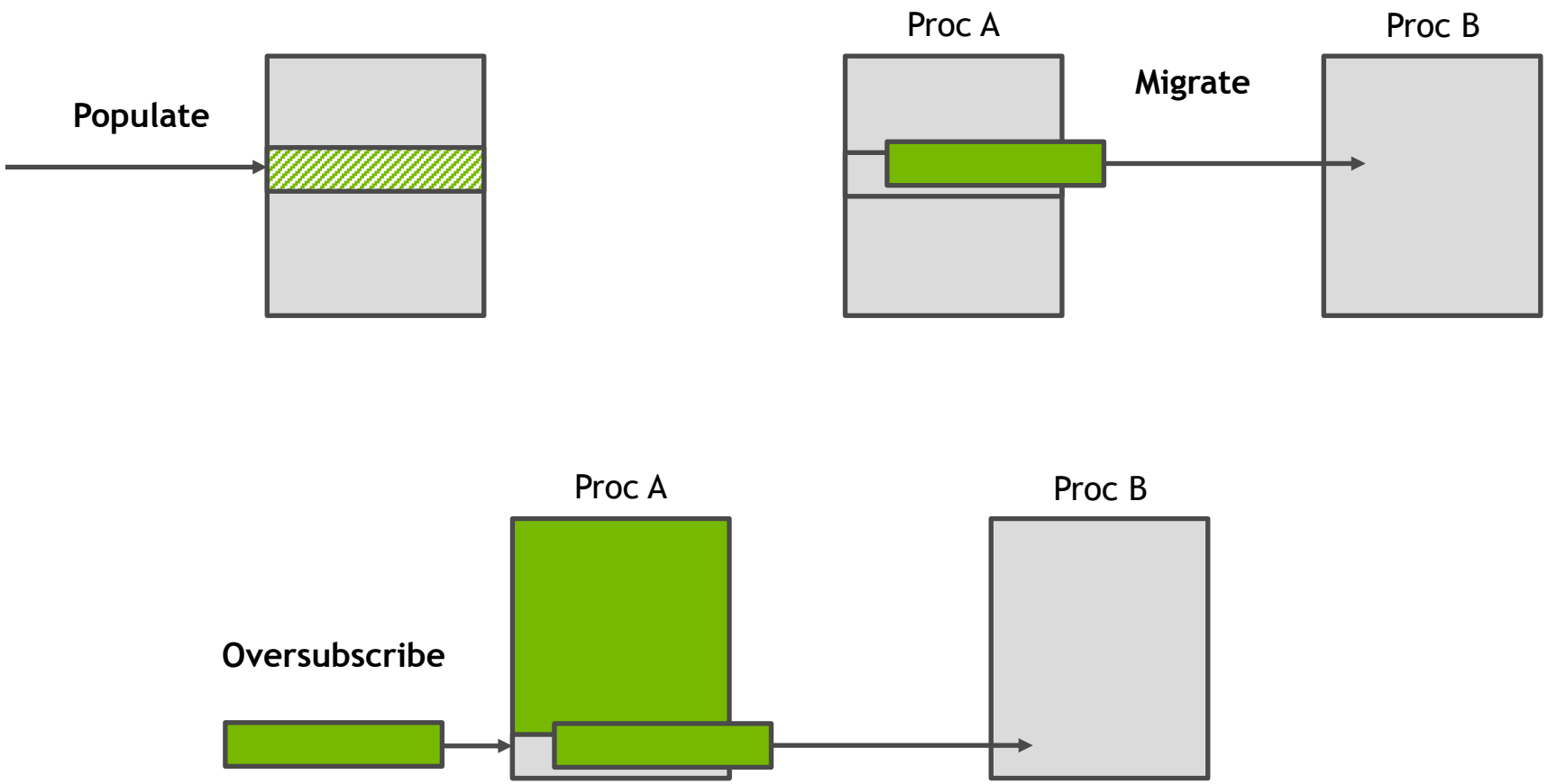


EXAMPLE: OVERSUBSCRIBE



*B cannot be a GPU in this case

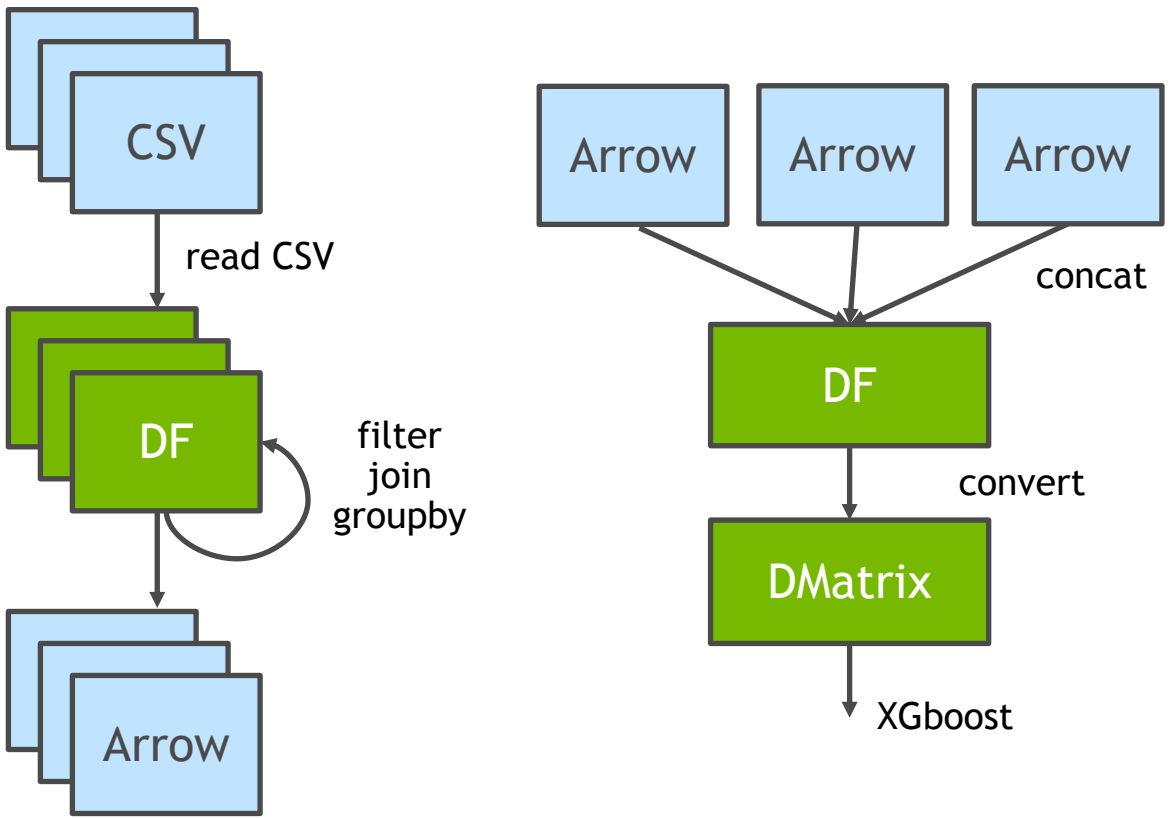
RECAP



APPLICATIONS IN ANALYTICS AND DL

S9726 - Unified Memory for Data Analytics and Deep Learning

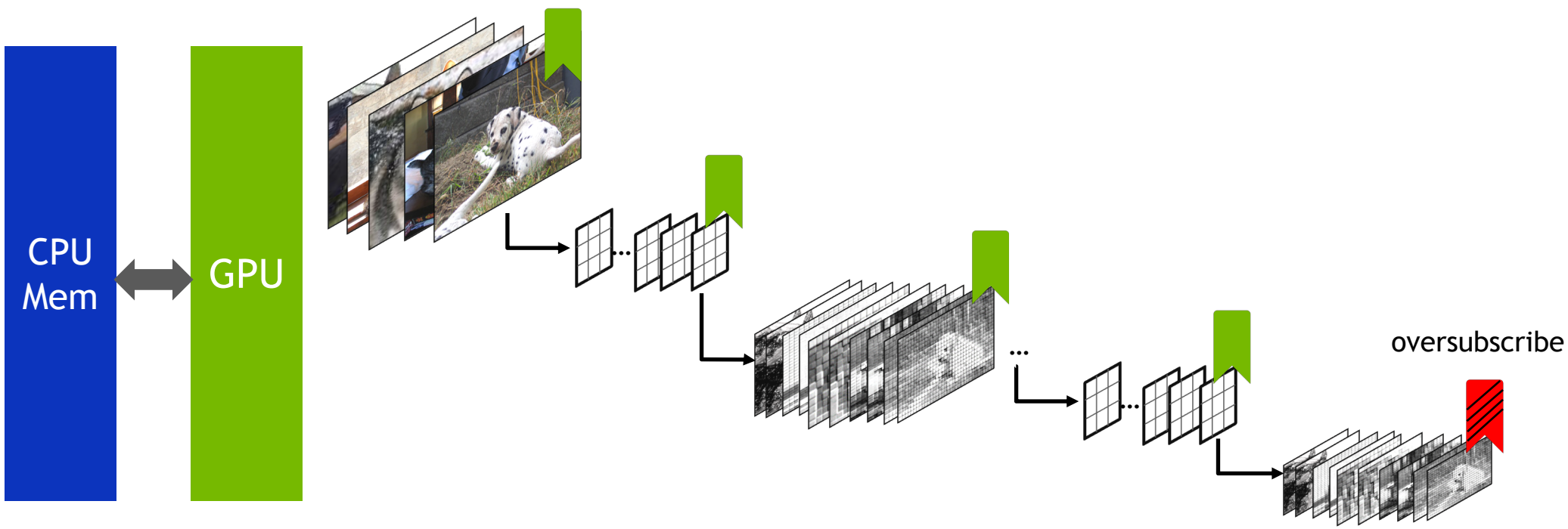
Thursday, Mar 21, 3:00 PM
– SJCC Room 211A (Concourse Level)



APPLICATIONS IN ANALYTICS AND DL

S9726 - Unified Memory for Data Analytics and Deep Learning

Thursday, Mar 21, 3:00 PM
– SJCC Room 211A (Concourse Level)



AGENDA

Key principles

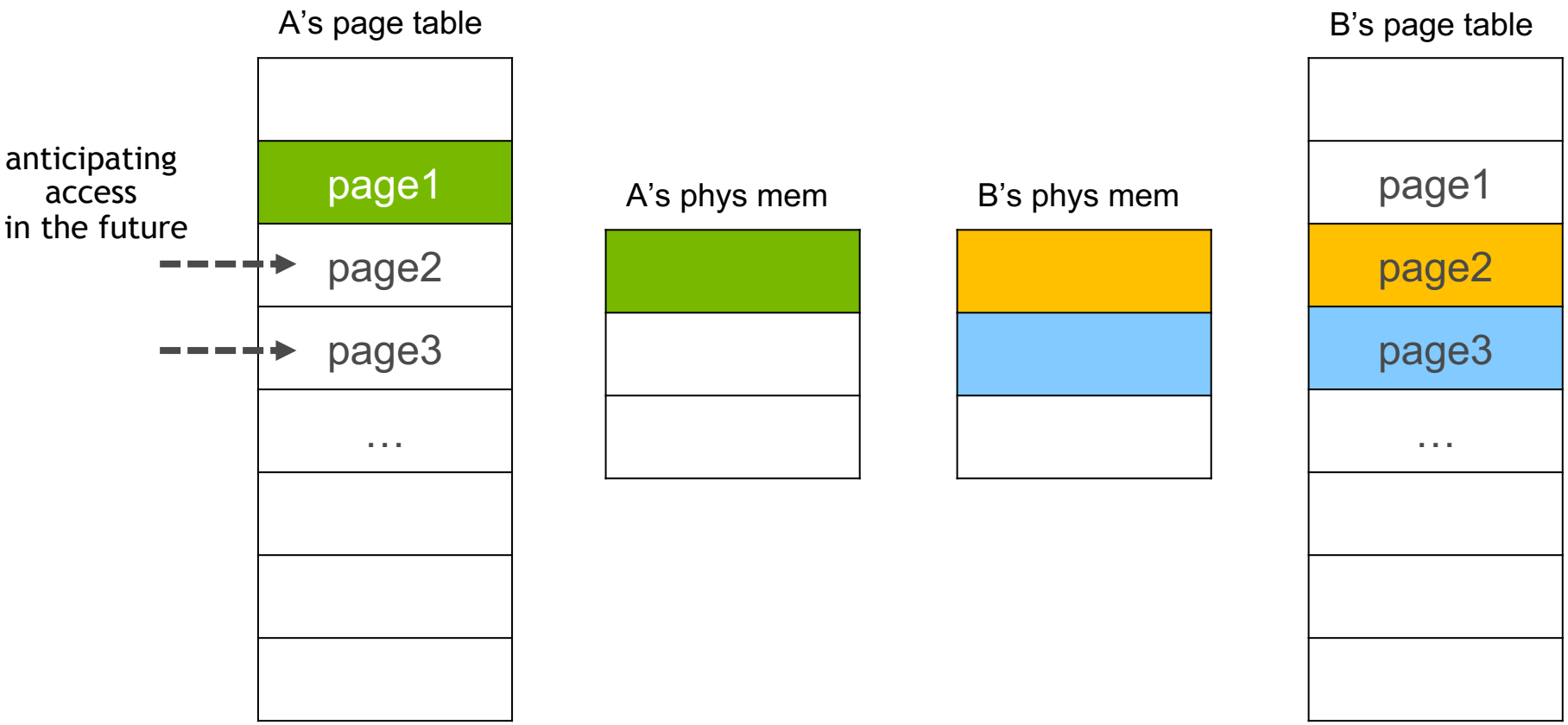
Performance tuning

Multi-GPU systems

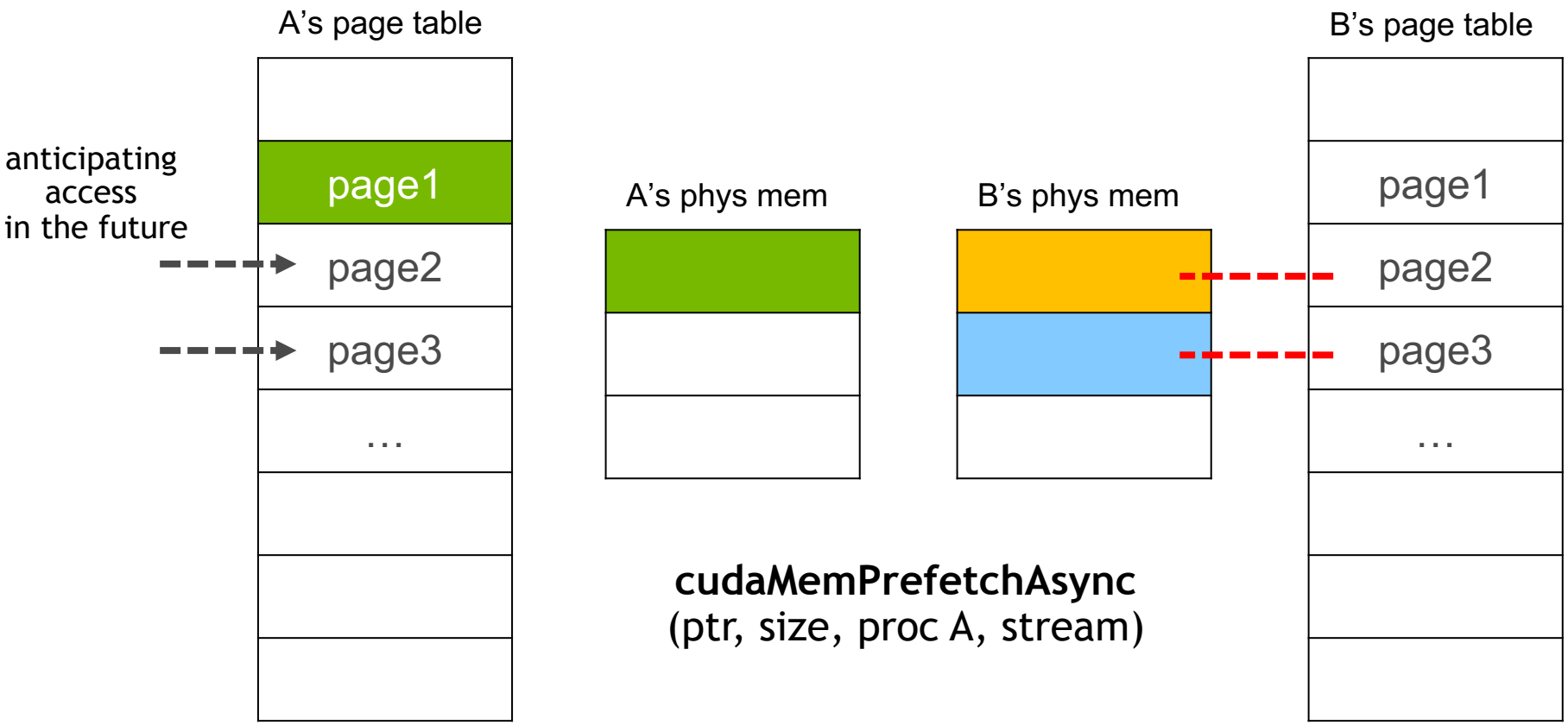
Summit & Sierra

OS integration

PREFETCH

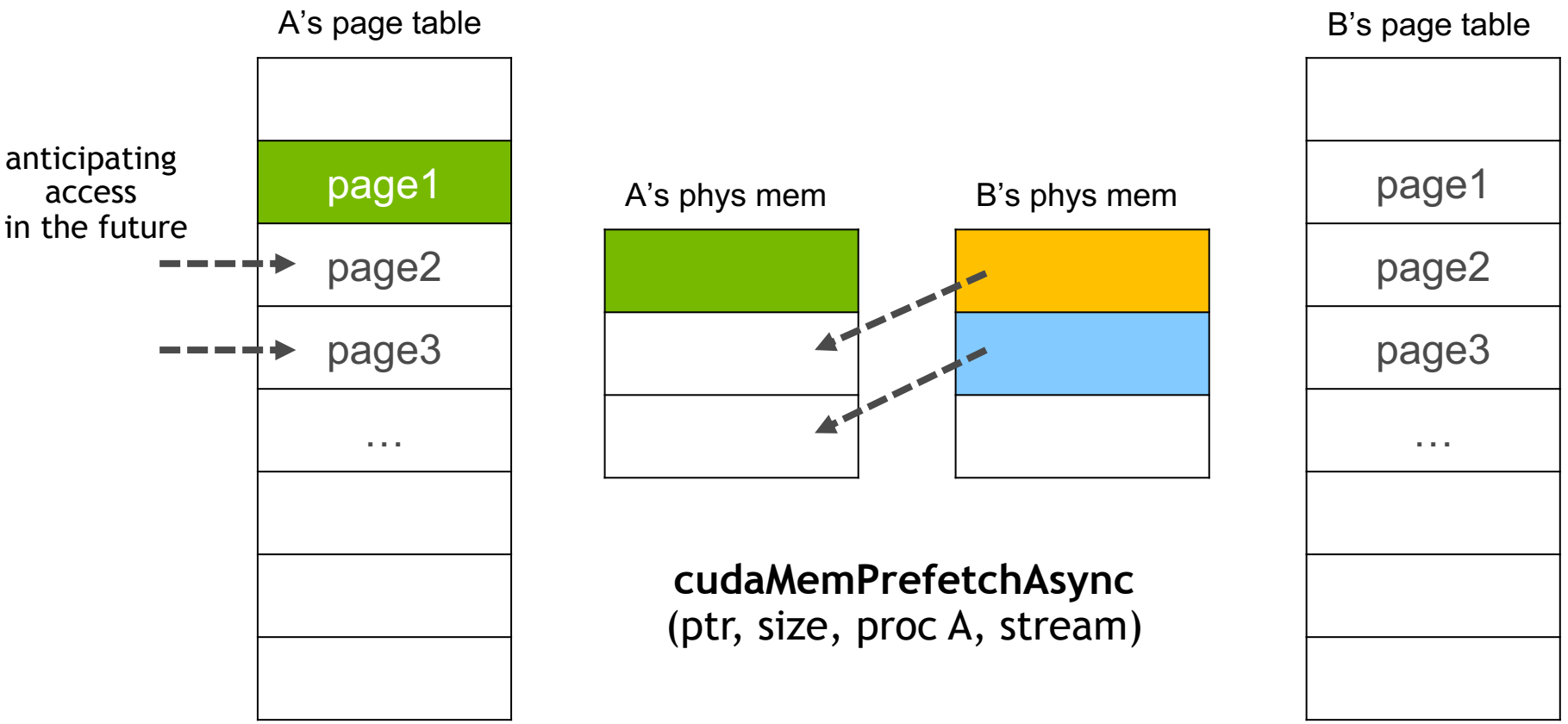

A’s page table
anticipating access in the future
page1
page2
page3
…
A’s phys mem
B’s phys mem
B’s page table
page1
page2
page3
…

PREFETCH



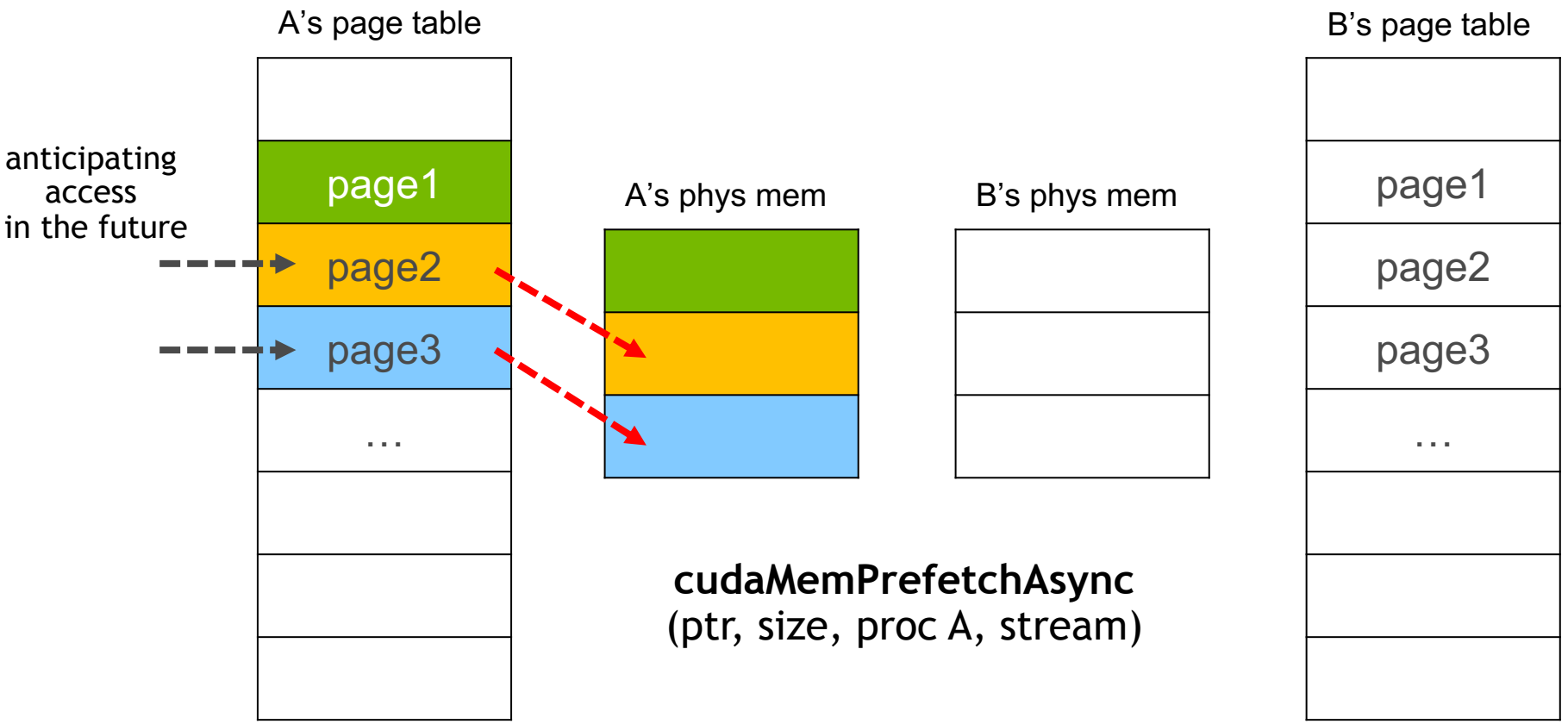
cudaMemPrefetchAsync
(ptr, size, proc A, stream)

PREFETCH



cudaMemPrefetchAsync
(ptr, size, proc A, stream)

PREFETCH



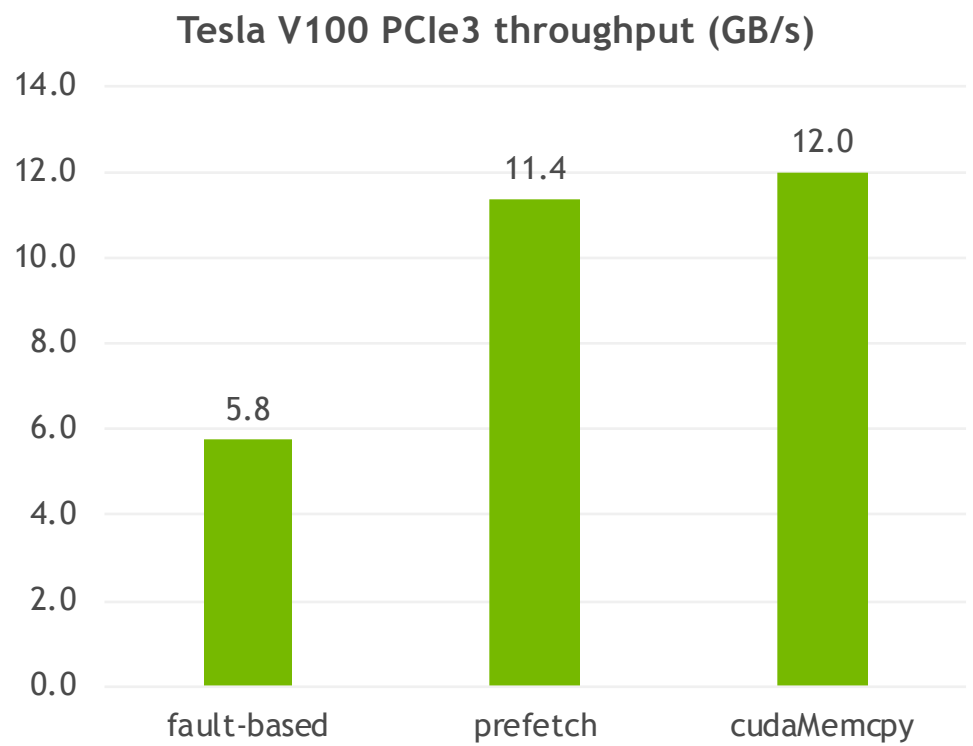
cudaMemPrefetchAsync
(ptr, size, proc A, stream)

MIGRATION PERFORMANCE

```
__global__ void kernel(int *host, int *device) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  device[i] = host[i];
}

// allocate and initialize memory
cudaMallocManaged(&host, size);
memset(host, 0, size);

// benchmark CPU->GPU migration
if (prefetch)
  cudaMemPrefetchAsync(host, size, gpuId);
kernel<<<grid, block>>>(host, device);
```



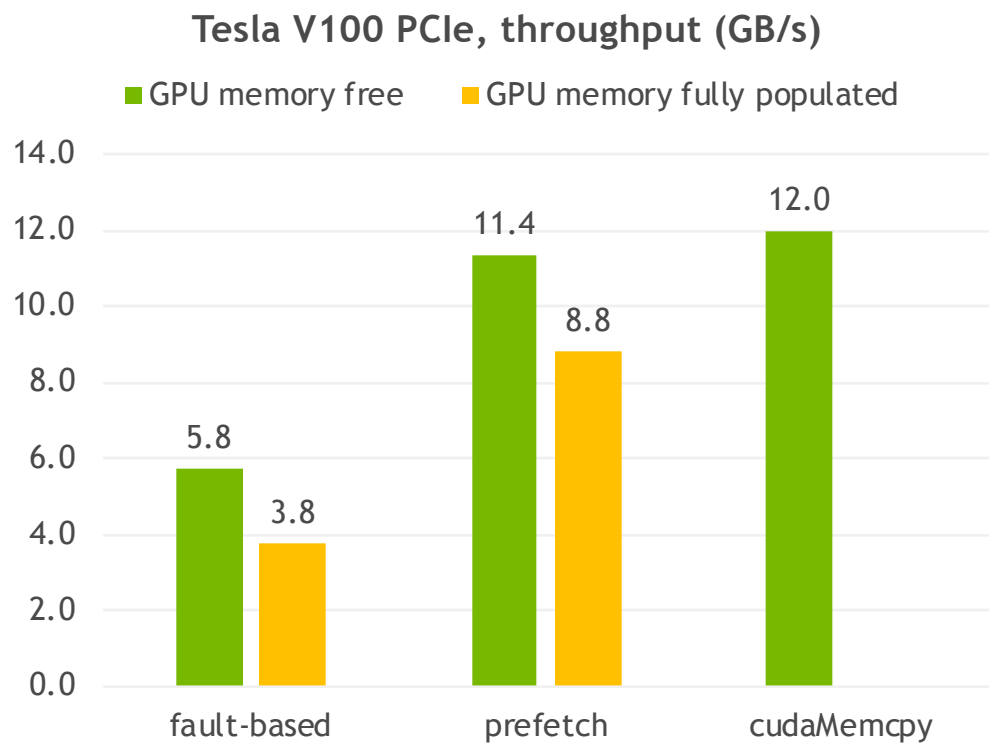
For more details see blog: https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

MIGRATION W/ OVERSUBSCRIPTION

```
// pre-populate GPU memory
cudaMallocManaged(&tmp, GPU_MEM_SIZE);
cudaMemPrefetchAsync(tmp, GPU_MEM_SIZE, gpuId);


// allocate and initialize memory
cudaMallocManaged(&host, size);
memset(host, 0, size);

// benchmark CPU->GPU migration
if (prefetch)
  cudaMemPrefetchAsync(host, size, gpuId);
kernel<<<grid, block>>>(host, device);
```

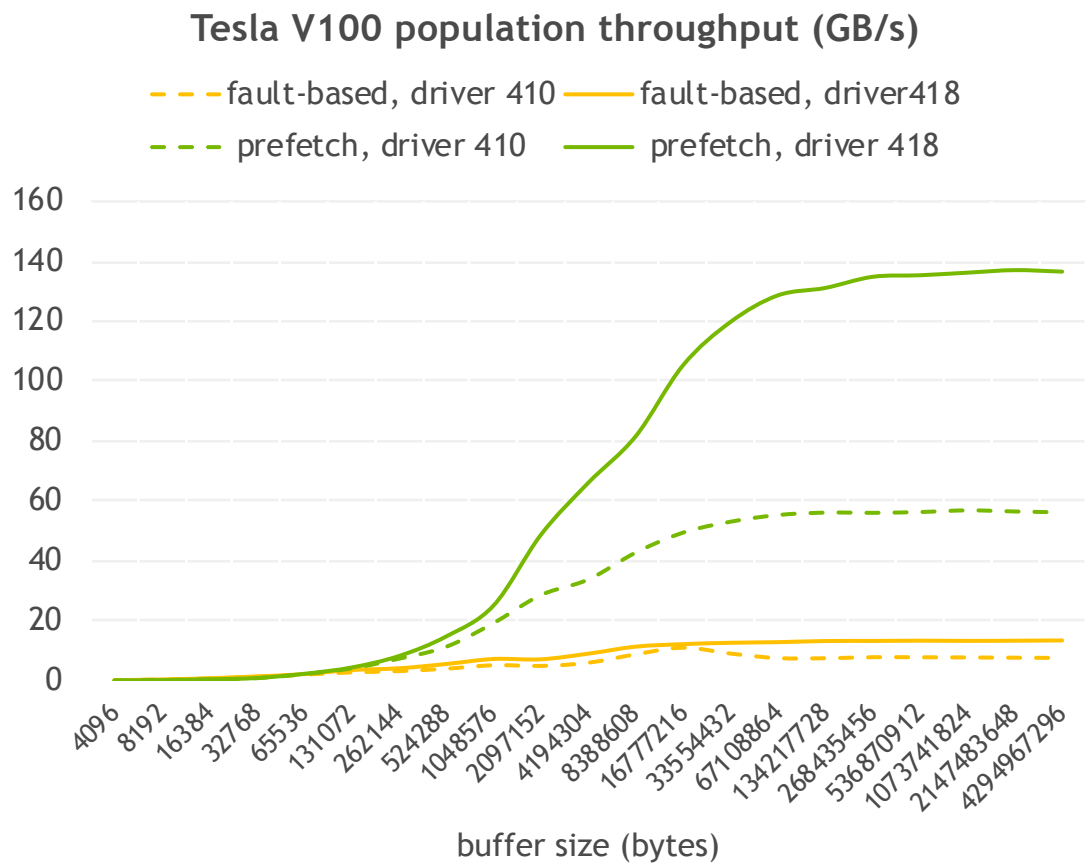


POPULATION PERFORMANCE

```
// fault-based
cudaMallocManaged(&ptr, size);
cudaMemset(ptr, 0, size);

// prefetch
cudaMallocManaged(&ptr, size);
cudaMemPrefetchAsync(ptr, size, gpuId);
cudaMemset(ptr, 0, size);
```

no migration traffic, just page population



PREFETCH GOTCHAS

CPU overhead related to updating page table mappings

Driver may defer prefetches to a background thread

How this may impact your applications:

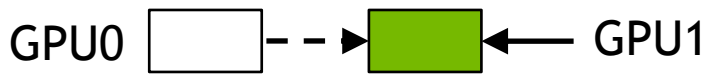
- DtoH prefetch may not return until the operation is completed
- Achieving good DtoH / HtoD overlap may be difficult in some cases

We’re actively working on improving prefetch implementation to alleviate those issues

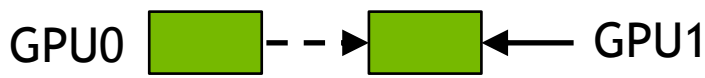
For more details see my blog: https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

USER POLICIES

Default: data *migrates* on access/prefetch

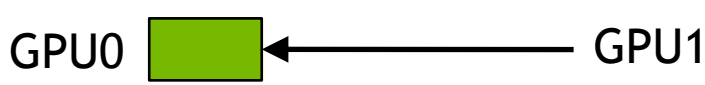


ReadMostly: data *duplicated* on read/prefetch



PreferredLocation: *resist* migrating away from it

AccessedBy: establish *direct mapping* / avoid faults



READ DUPLICATION

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..SetReadMostly, myGpuId);
cudaMemPrefetchAsync(data, N, myGpuId, s);

mykernel<<<..., s>>>(data, N);

use_data(data, N);

cudaDeviceSynchronize();
cudaFree(data);
```

populates data on the CPU

creates a copy on the GPU

both CPU and GPU can *read* data **simultaneously** without faults

writes will collapse all copies into one, subsequent reads will fault & duplicate

PREFERRED LOCATION

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..PreferredLocation, cudaCpuDeviceId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaDeviceSynchronize();
cudaFree(data);
```

← populates data on the CPU

← GPU *faults* and creates direct mapping to data

Possible reasons for migrating away:
1) Cannot access directly
2) Oversubscription
3) Prefetch

PREFERRED LOCATION

```
char *data;
cudaMallocManaged(&data, N);



cudaMemAdvise(data, N, ..PreferredLocation, cudaCpuDeviceId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaDeviceSynchronize();
cudaFree(data);
```

GPU *faults*, populates data on the CPU and creates direct mapping

*pages are populated in the preferred location if the faulting processor can access it

Note: this is true for GPUs, you can set preferred to GPU1, run kernel on GPU0 and it will populate data on GPU1

ACCESSED BY

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaDeviceSynchronize();
cudaFree(data);
```

← populates data on the CPU

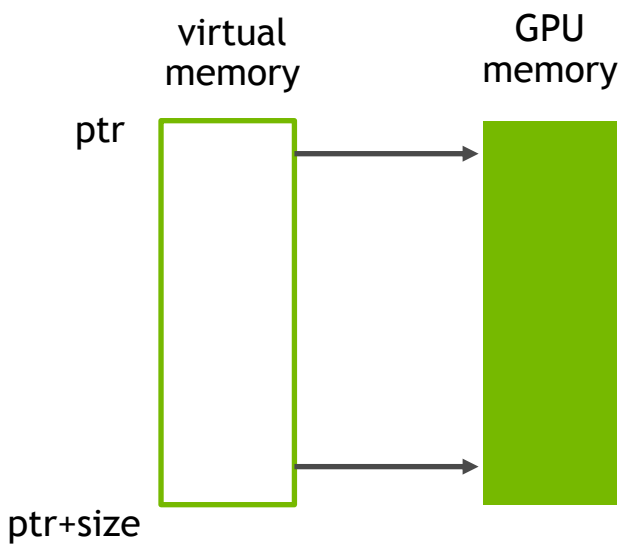
← GPU creates direct mapping

← GPU accesses data remotely without page faults

*memory can move freely to other processors and mapping will carry over

CUDAMALLOC VS UNIFIED MEMORY

```
cudaMalloc(&ptr, size);
```



=

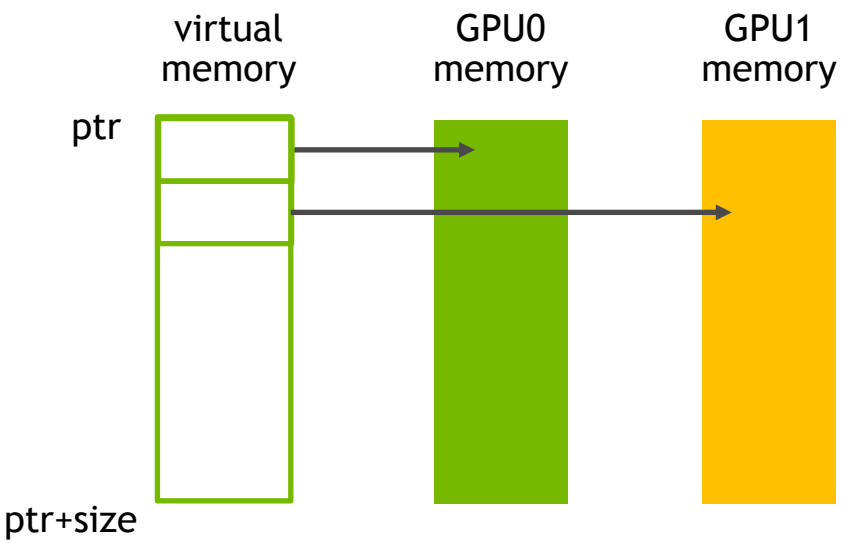
```
cudaMallocManaged(&ptr, size);

for (int i = 0; i < ngpus; i++)
  cudaMemAdvise(ptr, size, ..AccessedBy, i);

cudaMemAdvise(ptr, size, ..PreferredLocation, gpuId);

cudaSetDevice(gpuId);
cudaMemPrefetchAsync(ptr, size, gpuId);
cudaDeviceSynchronize();
```

PAGE STRIPING




```
cudaMallocManaged(&ptr, size);

for (int i = 0; i < ngpus; i++)
  cudaMemAdvise(ptr, size, ..AccessedBy, i);

for (pages) {
  cudaMemAdvise(p, page_size, ..PreferredLocation, g);
  cudaSetDevice(g);
  cudaMemPrefetchAsync(p, page_size, g);
}
for (int i = 0; i < ngpus; i++) {
  cudaSetDevice(i);
  cudaDeviceSynchronize();
}
```

AGENDA

Key principles

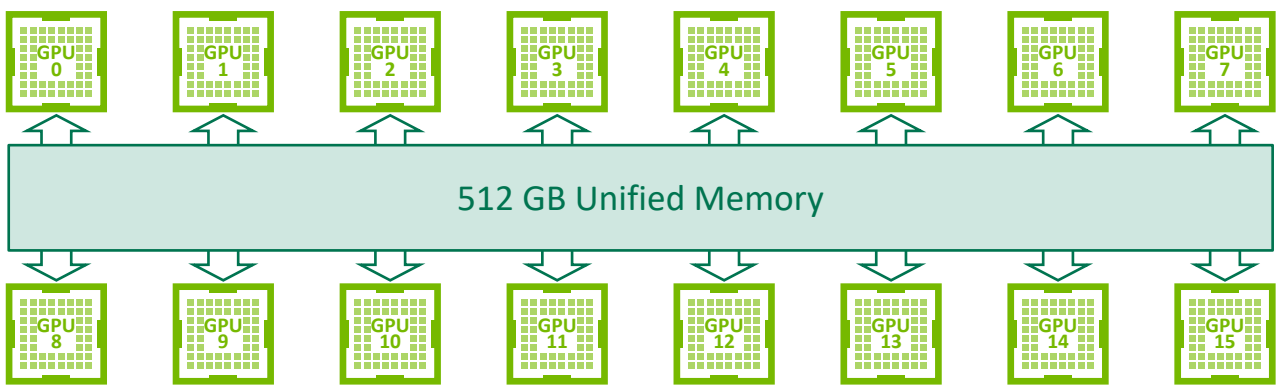
Performance tuning

Multi-GPU systems

Summit & Sierra

OS integration

UNIFIED MEMORY + DGX-2



UNIFIED MEMORY PROVIDES

Single memory view shared by all GPUs

Automatic migration of data between GPUs

User control of data locality

S9241 – All You Need To Know about Programming NVIDIA's DGX-2

Wednesday, Mar 20, 1:00 PM
– SJCC Room 220C (Concourse Level)

ENABLE MULTI-GPU WITH SINGLE PROCESS

Single-GPU

```
__global__ void kernel(int *data) {
  int idx = threadIdx.x + blockDim.x * blockIdx.x;

  doSomeStuff(idx, data, ...);
}

cudaMallocManaged(&data, N * sizeof(int));
// initialize data on the CPU

kernel<<<grid, block>>>(data);
```

Multi-GPU

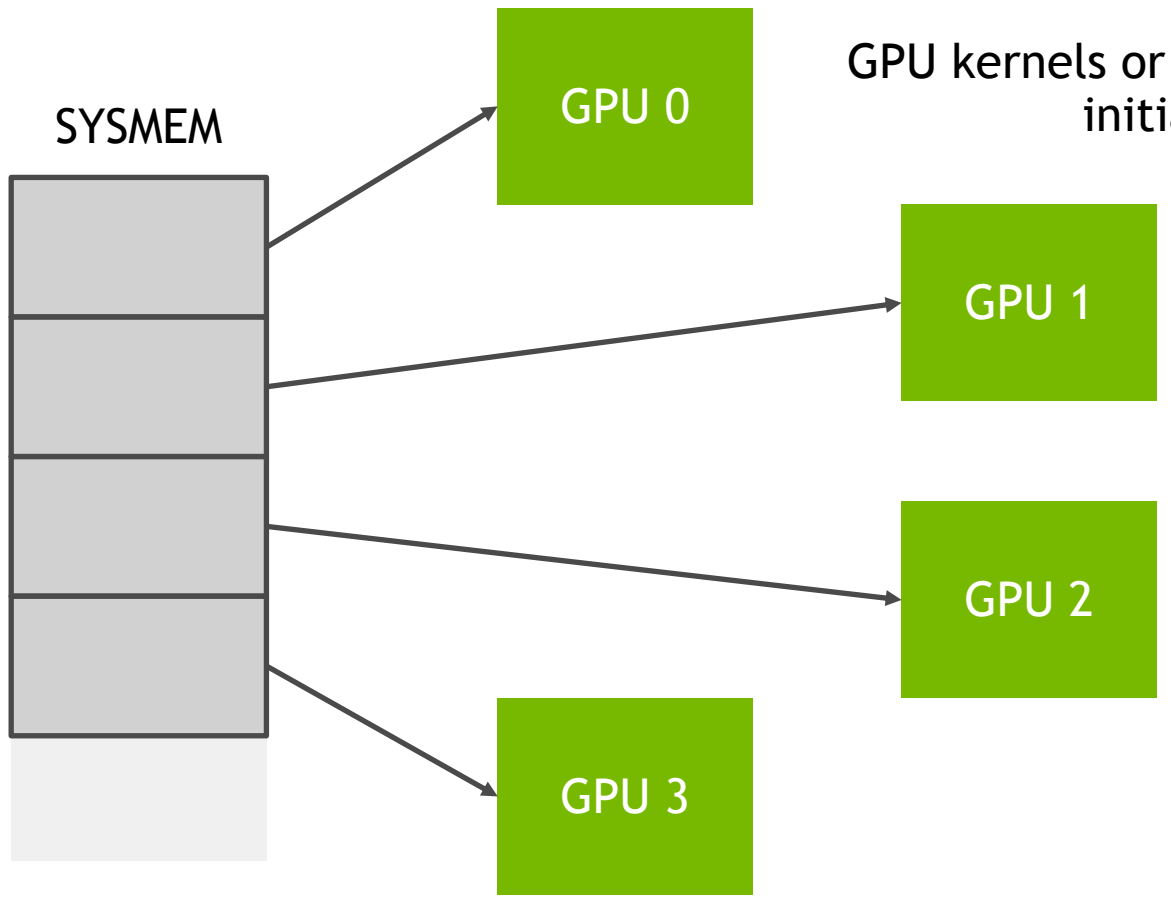
```
__global__ void kernel(int *data, int gpuId) {
  int idx = threadIdx.x + blockDim.x * (blockIdx.x
            + gpuId * gridDim.x);
  doSomeStuff(idx, data, ...);
}

cudaMallocManaged(&data, N * sizeof(int));
// initialize data on the CPU
for (int i = 0; i < ngpus; i++) {
  cudaSetDevice(i);
  kernel<<<grid/ngpus, block>>>(data, i);
}
```

update blockIdx.x

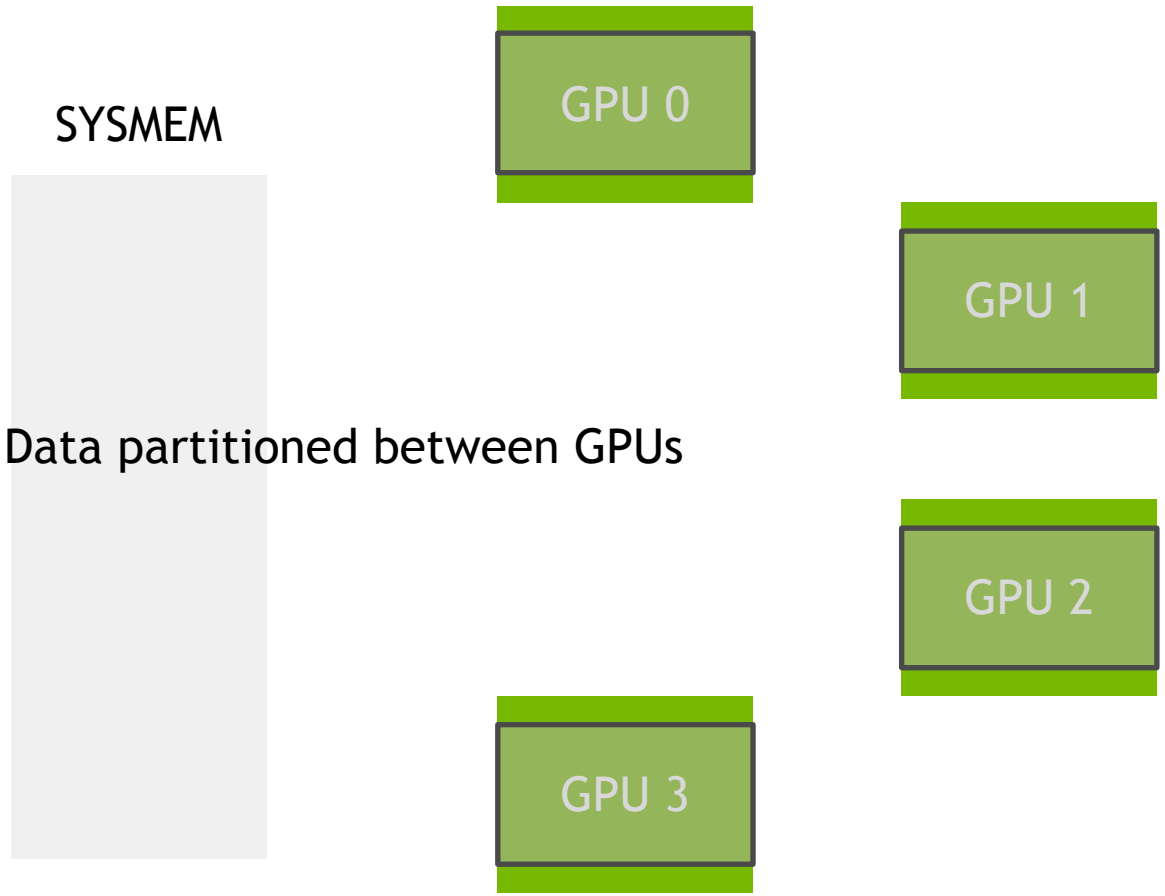
update launch config

MULTI-GPU WITH UNIFIED MEMORY

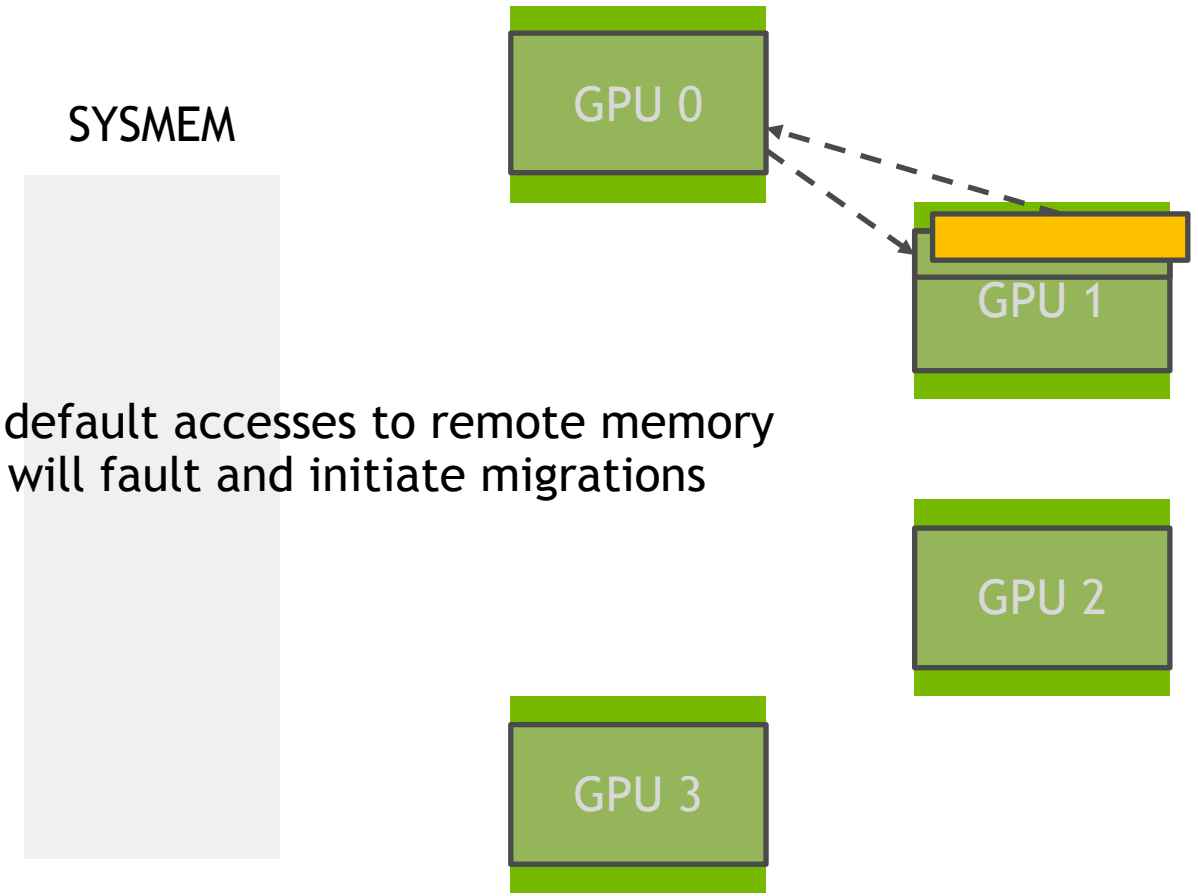


GPU kernels or **cudaMemPrefetchAsync** initiate migrations

MULTI-GPU WITH UNIFIED MEMORY



MULTI-GPU WITH UNIFIED MEMORY



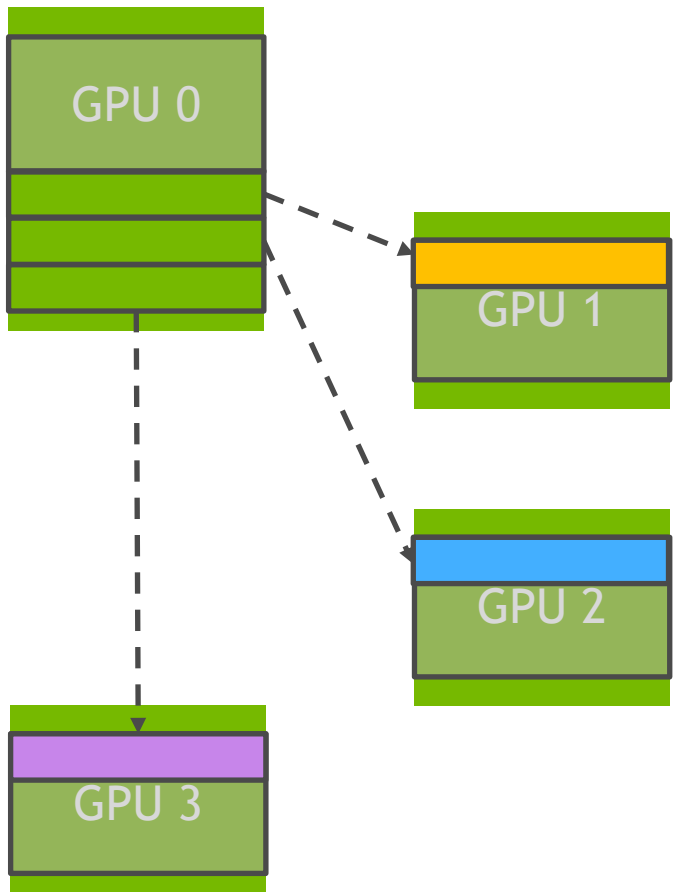
By default accesses to remote memory will fault and initiate migrations

MULTI-GPU WITH UNIFIED MEMORY

```
for (int i = 0; i < ngpus; i++)
  cudaMemAdvise(ptr, size, ..SetAccessedBy, i);

for (int i = 0; i < ngpus; i++) {
  size_t off = (size / ngpus) * i;
  cudaMemAdvise(ptr + off, ..PreferredLocation, i);
  cudaMemPrefetchAsync(ptr + off, size / ngpus, i);
}
```

Remote memory accessed
directly without faults or migrations



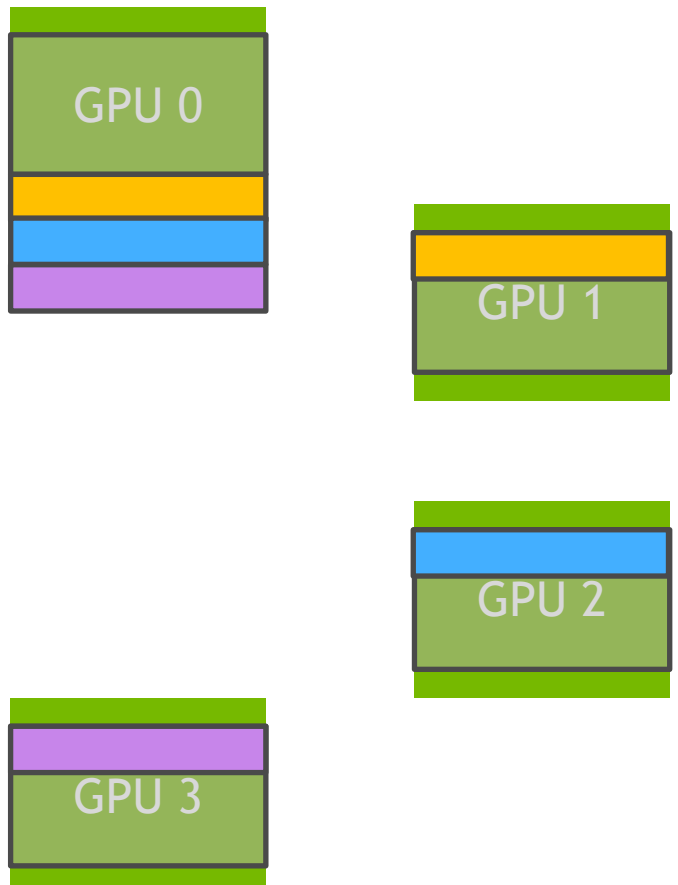
https://github.com/NVIDIA/multi-gpu-programming-models/tree/master/multi_threaded_um

MULTI-GPU WITH UNIFIED MEMORY

```
cudaMemAdvise(ptr, size, ..SetReadMostly, cudaInvalidDeviceId);

for (int i = 0; i < ngpus; i++) {
  size_t off = (size / ngpus) * i;
  cudaMemPrefetchAsync(ptr + off, size / ngpus, i);
}
```

Read-only data is
duplicated and accessed *locally*



https://github.com/NVIDIA/multi-gpu-programming-models/tree/master/multi_threaded_um

UNIFIED MEMORY: MULTIPLE PROCESSES

CUDA IPC used to exchange cudaMalloc pointers – this doesn't work for cudaMallocManaged!

All major MPI implementations support Unified Memory through staging

Multiple processes sharing single GPU memory

```
cudaSetDevice(0);
cudaMallocManaged(&ptr, GPU_MEM_SIZE);
cudaMemPrefetchAsync(ptr, GPU_MEM_SIZE, 0);
```

Process A

evicts A's data to the CPU 

```
cudaSetDevice(0);
cudaMallocManaged(&ptr, GPU_MEM_SIZE);
cudaMemPrefetchAsync(ptr, GPU_MEM_SIZE, 0);
```

Process B

AGENDA

Key principles

Performance tuning

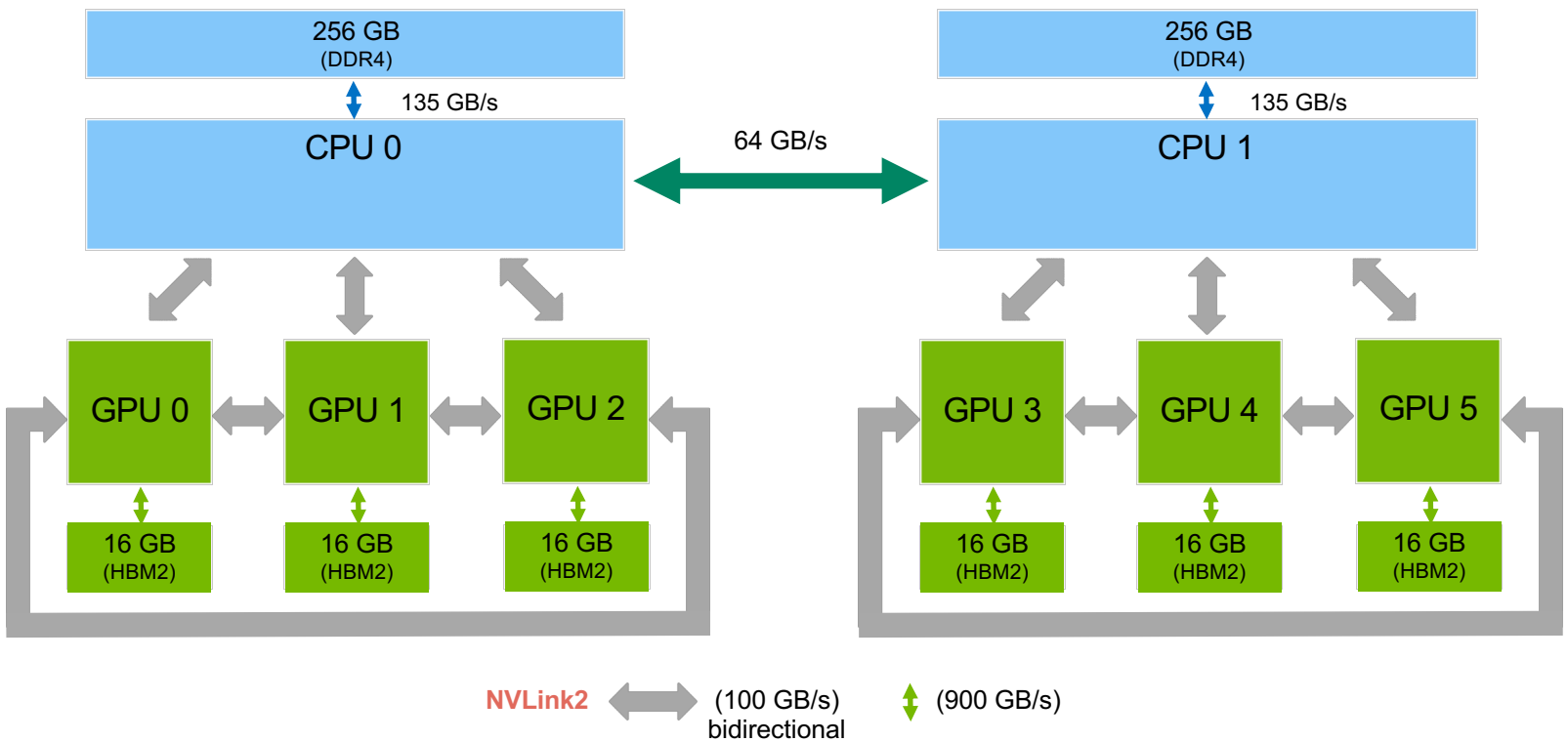
Multi-GPU systems

Summit & Sierra

OS integration

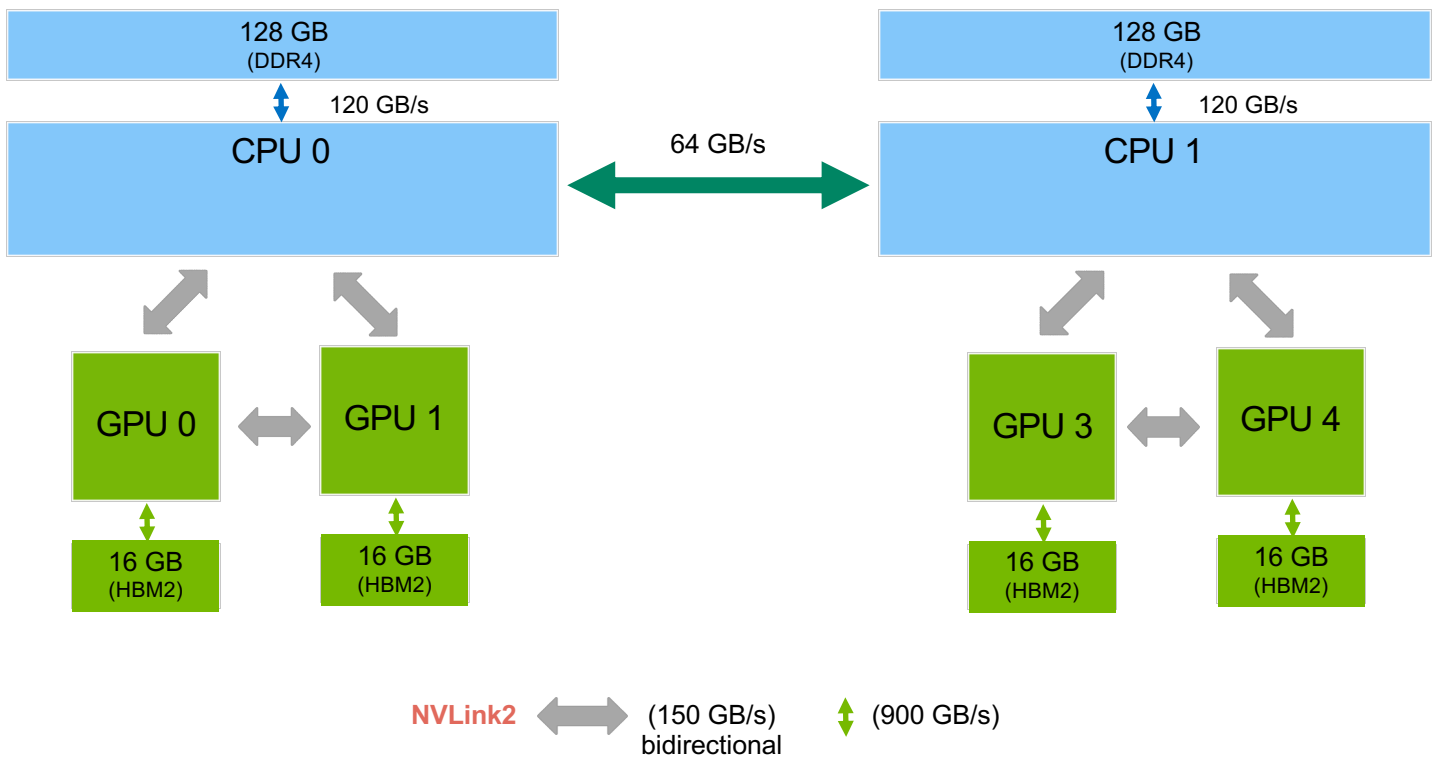
SUMMIT NODE

(2) IBM Power9 + (6) NVIDIA Volta V100



SIERRA NODE

(2) IBM Power9 + (4) NVIDIA Volta V100



VOLTA+P9 FEATURES

Volta's access counters

cudaMallocManaged may use access counters to guide migrations (opt-in)

NVLINK2 protocol

Enables HW coherency (CPU access GPU memory)

Indirect peers: GPU access memory of remote GPUs on a different socket

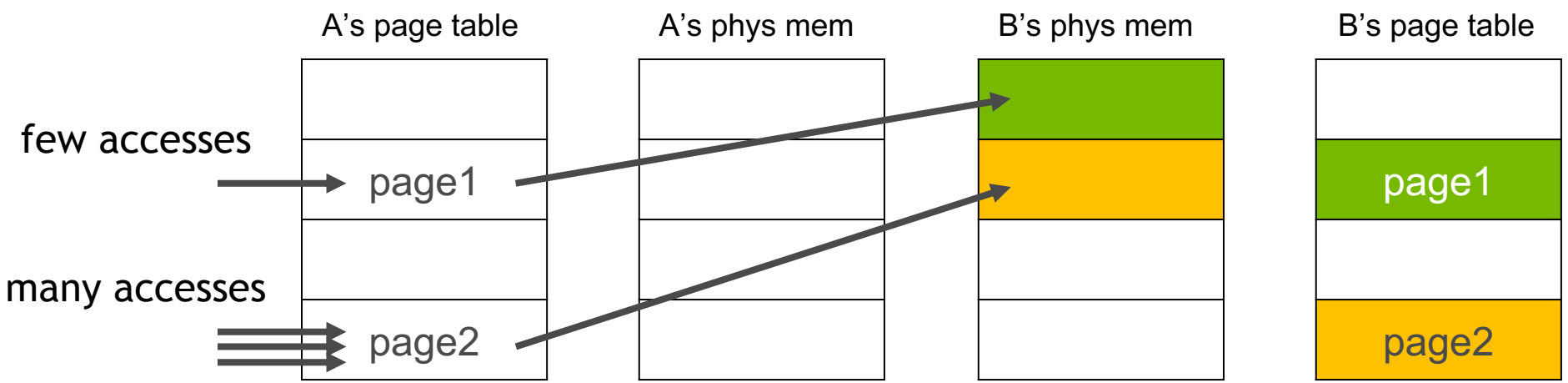
Native atomics support for all accessible memory

ATS (Address Translation Service)

GPU can access all system memory (malloc, stack, mmap files)

UNIFIED MEMORY ON VOLTA+P9

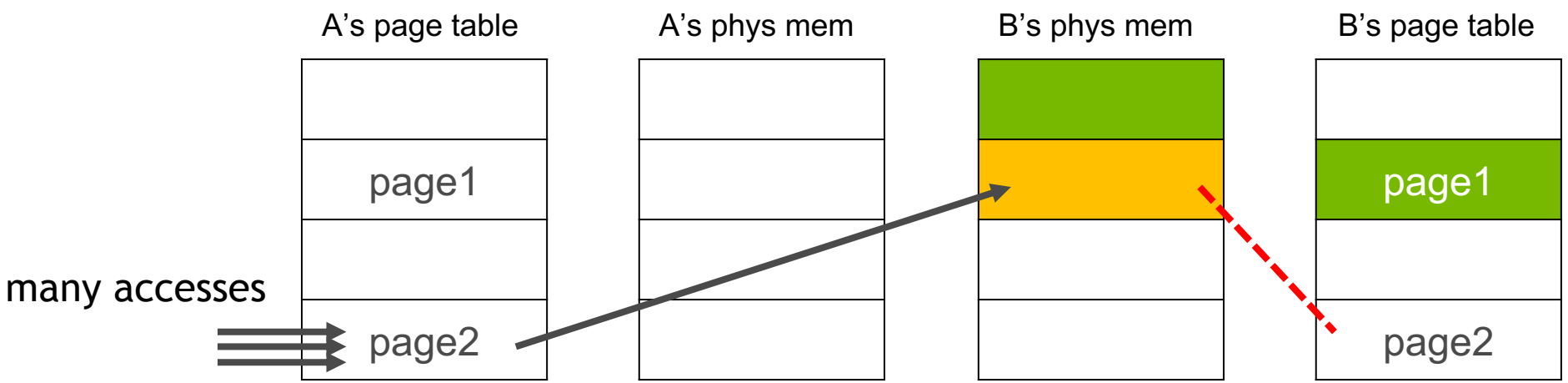
New Feature: Access Counters



If memory is *mapped* to the GPU, migration can be triggered by access counters

UNIFIED MEMORY ON VOLTA+P9

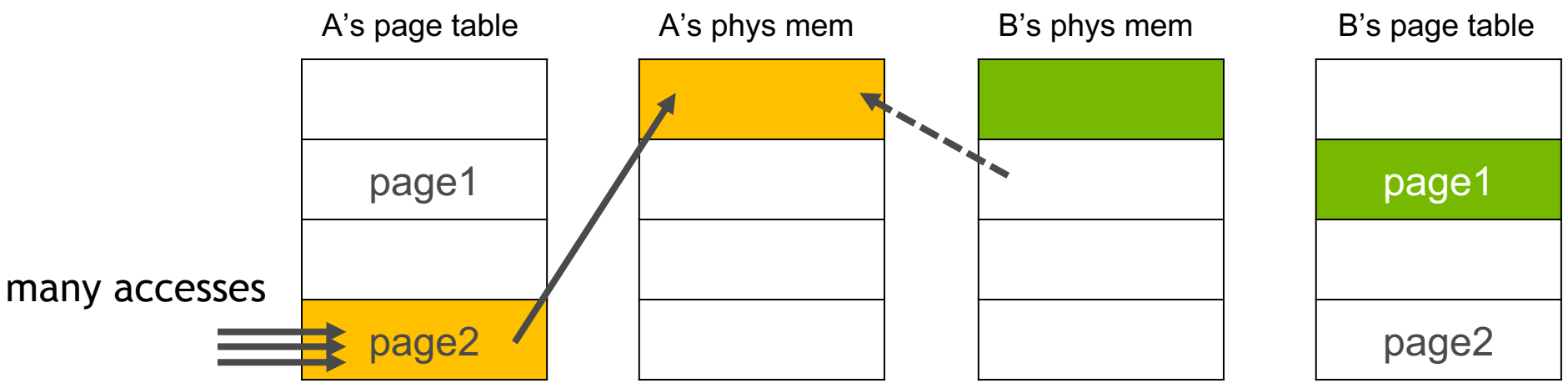
New Feature: Access Counters



page2 is unmapped from B

UNIFIED MEMORY ON VOLTA+P9

New Feature: Access Counters



Data for page2 is copied to A and mapping updated

ACCESSED BY ON VOLTA+P9

```
char *data;
cudaMallocManaged(&data, N);

cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);

init_data(data, N);

mykernel<<<..., s>>>(data, N);

use_data(data, N);

cudaDeviceSynchronize();
cudaFree(data);
```

← GPU creates direct mapping

← populates data on the CPU

← Volta's access counters may eventually trigger migration of **frequently accessed pages** to GPU

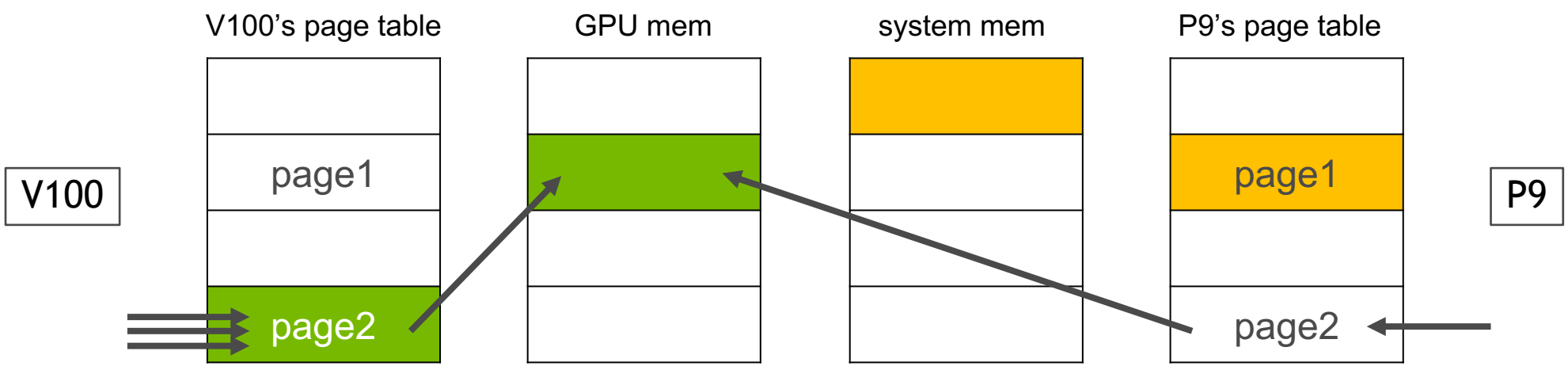
on non P9/V100 systems the data will stay in CPU memory

UNIFIED MEMORY ON VOLTA+P9

New Feature: Hardware Coherency with NVLINK2

CPU can directly access and *cache* GPU memory

Native atomics support for all accessible memory



PREFERRED LOCATION ON VOLTA+P9

```
char *data;
cudaMallocManaged(&data, N);
cudaMemAdvise(data, N, ..PreferredLocation, gpuId);

init_data(data, N);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

CPU will *page fault* and populate data on the GPU

The driver will “resist” migrating data away from GPU

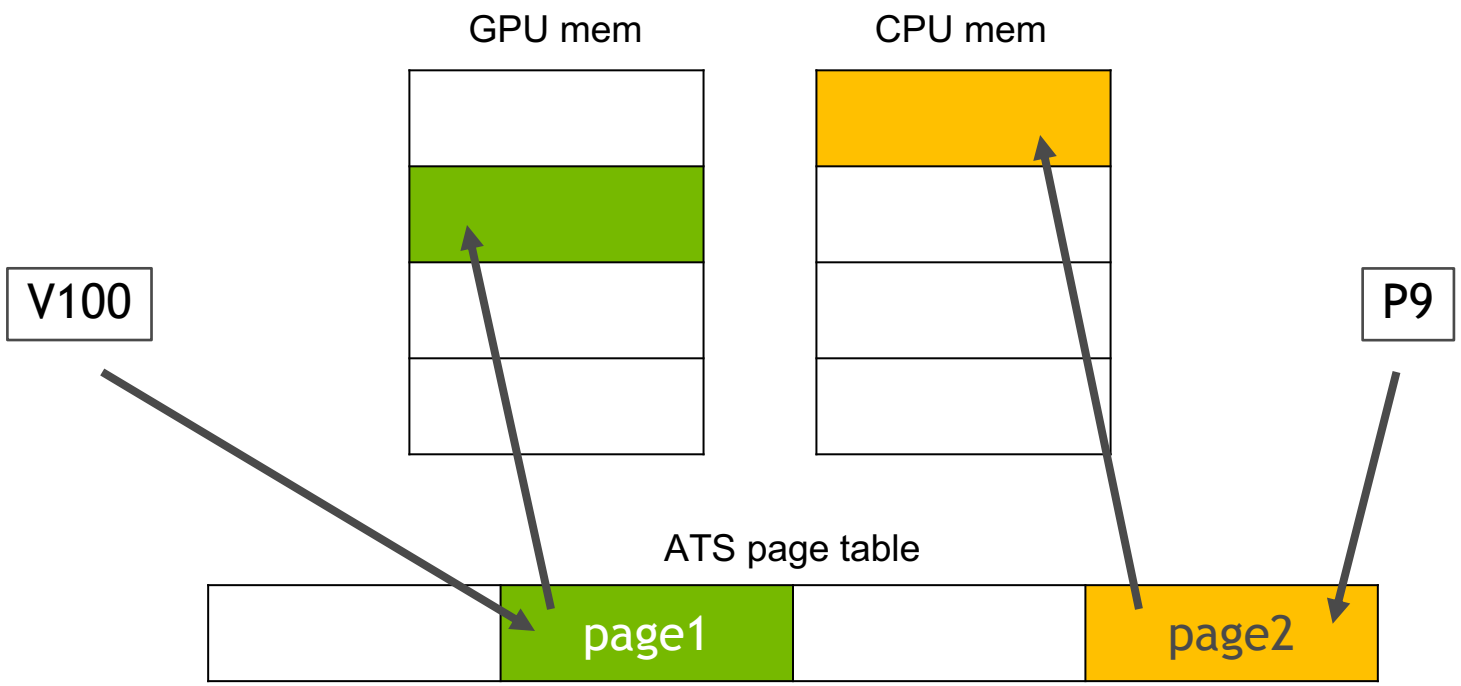
CPU accesses GPU memory *directly*

on non P9/V100 systems the driver will migrate back to the CPU

UNIFIED MEMORY ON VOLTA+P9

New Feature: ATS support

ATS: address translation service; CPU and GPU can share a *single* page table



MANAGED VS MALLOC ON VOLTA+P9

First touch allocation policy

```
ptr = cudaMallocManaged(size);

doStuffOnGpu<<<...>>>(ptr, size);
```

GPU page faults

Unified Memory driver allocates on GPU

GPU accesses **GPU memory**

```
ptr = malloc(size);

doStuffOnGpu<<<...>>>(ptr, size);
```

GPU uses ATS, faults

OS allocates on CPU (by default)

GPU uses ATS to access **CPU memory**

MANAGED VS MALLOC ON P9

cudaMallocManaged: same behavior as x86

```
ptr = cudaMallocManaged(size);

fillData(ptr, size);

doStuffOnGpu<<<...>>>(ptr, size);

cudaDeviceSynchronize();

doStuffOnCpu(ptr, size);
```

GPU page faults
ptr migrated to GPU

CPU page faults
ptr migrated to CPU

MANAGED VS MALLOC ON P9

malloc: no on-demand migrations*

```
ptr = malloc(size);

fillData(ptr, size);

doStuffOnGpu<<<...>>>(ptr, size);

cudaDeviceSynchronize();

doStuffOnCpu(ptr, size);
```

*no on-demand migration except cudaMemPrefetchAsync**

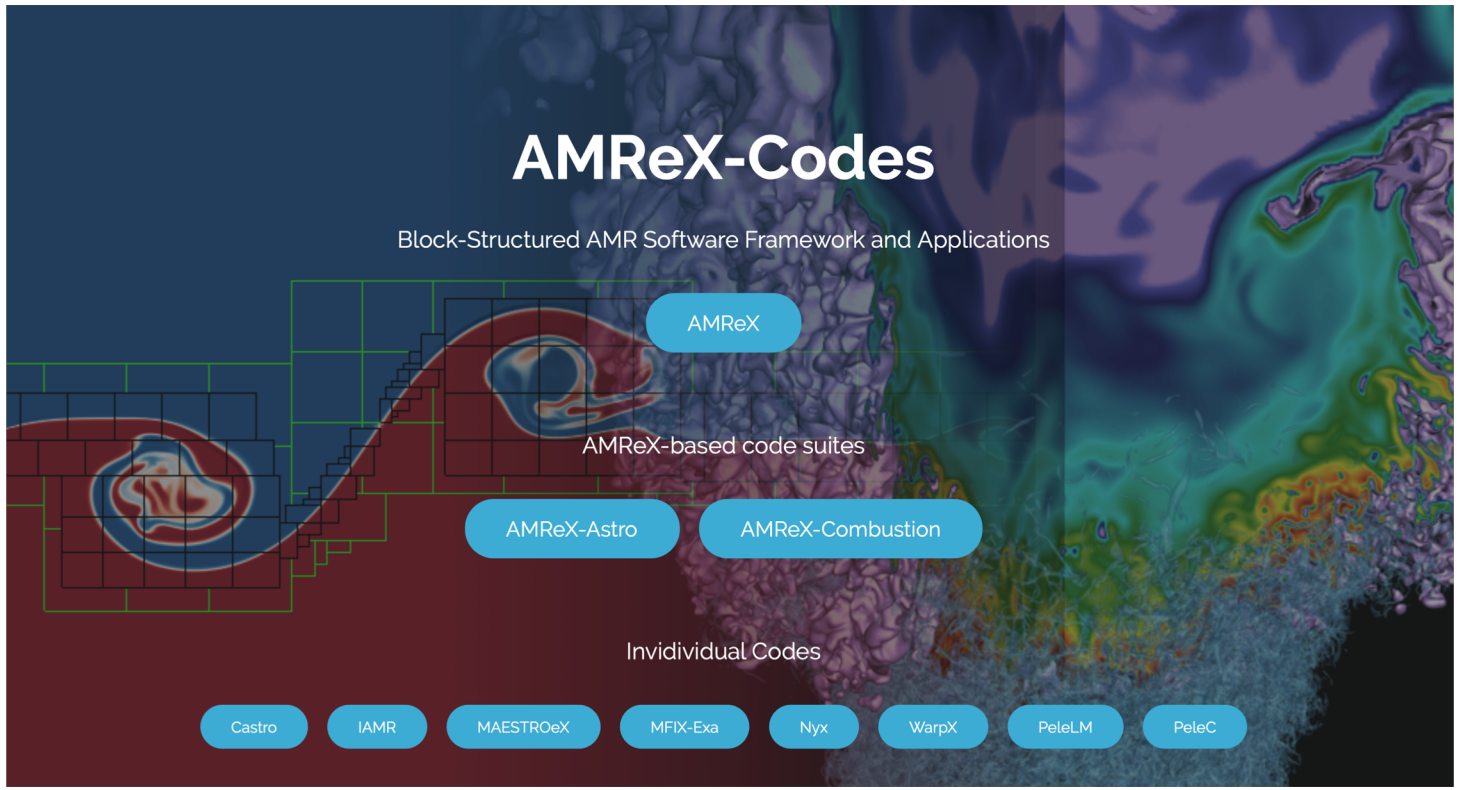
GPU uses ATS to access CPU memory

CPU accesses CPU memory

**In the future access counters may be used to migrate malloc memory*

POWER9+V100 APPLICATIONS

https://www.exascaleproject.org/project/amrex-co-design-center-block-structured-amr/



AGENDA

Key principles

Performance tuning

Multi-GPU systems

Summit & Sierra

OS integration

UNIFIED MEMORY IN MANY LANGUAGES

CUDA C/C++: **cudaMallocManaged**

CUDA Fortran: **managed** attribute

Python: **pycuda.driver.managed_empty**

OpenACC: **-ta=managed** compiler option (all dynamic allocations)

Available as opt-in in GPU caching allocators and memory managers (CNMEM, RMM)

UNIFIED MEMORY + OPENACC

Effortless way to run you code on GPUs

Literally adding a single line will get your code running on the GPU

```
#pragma acc kernels
{
  for (i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
    ...
  }
}

...
```

Initiate parallel execution

Easy to optimize later: add loop and data directives

UNIFIED MEMORY WITH SYSTEM ALLOCATOR

System allocator support allows GPU to access **all** system memory

malloc, stack, global, file system

P9: Address Translation Service (ATS) - enabled since CUDA 9.2

x86: Heterogeneous Memory Management (HMM)

Initial version of the patchset integrated into 4.14 kernel

Still in the kernel - 5.0 ☺

NVIDIA will be supporting upcoming versions of HMM

https://lkml.org/lkml/2017/6/23/443

WHAT YOU CAN DO WITH UNIFIED MEMORY

Works everywhere today

```
int *data;
cudaMallocManaged(&data, sizeof(int) * n);
kernel<<<grid, block>>>(data);
```

Works today on Power9 + Volta
Will work in the future on x86 + HMM

```
int *data = (int*)malloc(sizeof(int) * n);
kernel<<<grid, block>>>(data);
```

```
int data[1024];
kernel<<<grid, block>>>(data);
```

```
int *data = mmap(0, size, .. , fd, 0);
kernel<<<grid, block>>>(data);
```

```
extern int *data;
kernel<<<grid, block>>>(data);
```


DEMO TIME!

TAKEAWAY

Unified Memory enables new, **more productive** ways of managing CPU/GPU memory

It's easy to start with bare metal Unified Memory, then **tune performance** with hints

ATS and HMM provide easier **integration with OS** and legacy CPU libraries

Connect with the Experts session on memory management

Hall 3 Pod C – 4:00pm, *right after this talk!*


NVIDIA