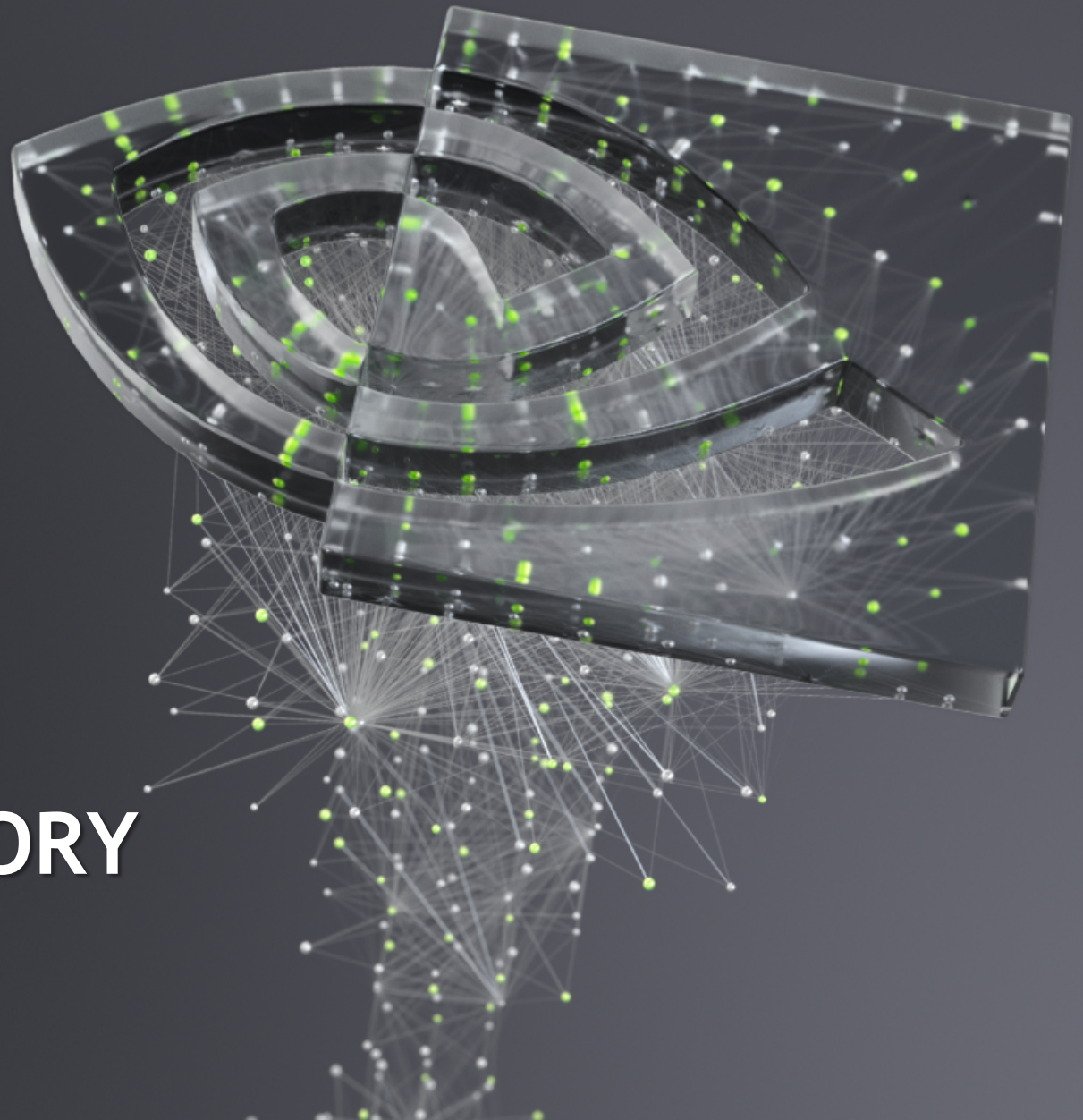


CUDA UNIFIED MEMORY

Bob Crovella, 6/18/2020

EVERYTHING YOU NEED TO KNOW ABOUT UNIFIED MEMORY

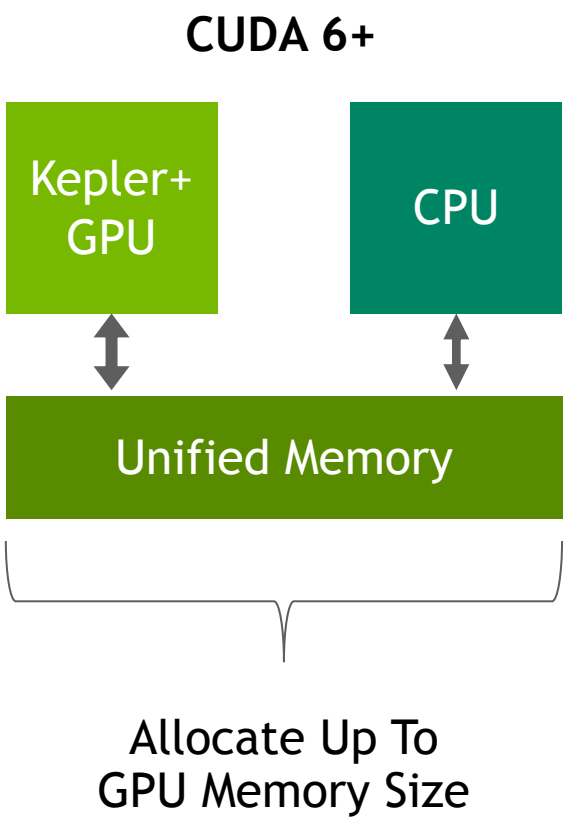
Nikolay Sakharnykh, 3/27/2018

NVIDIA


INTRODUCING UNIFIED MEMORY
WITH DEMAND PAGING

UNIFIED MEMORY

Reduce Developer Effort



Simpler Programming & Memory Model

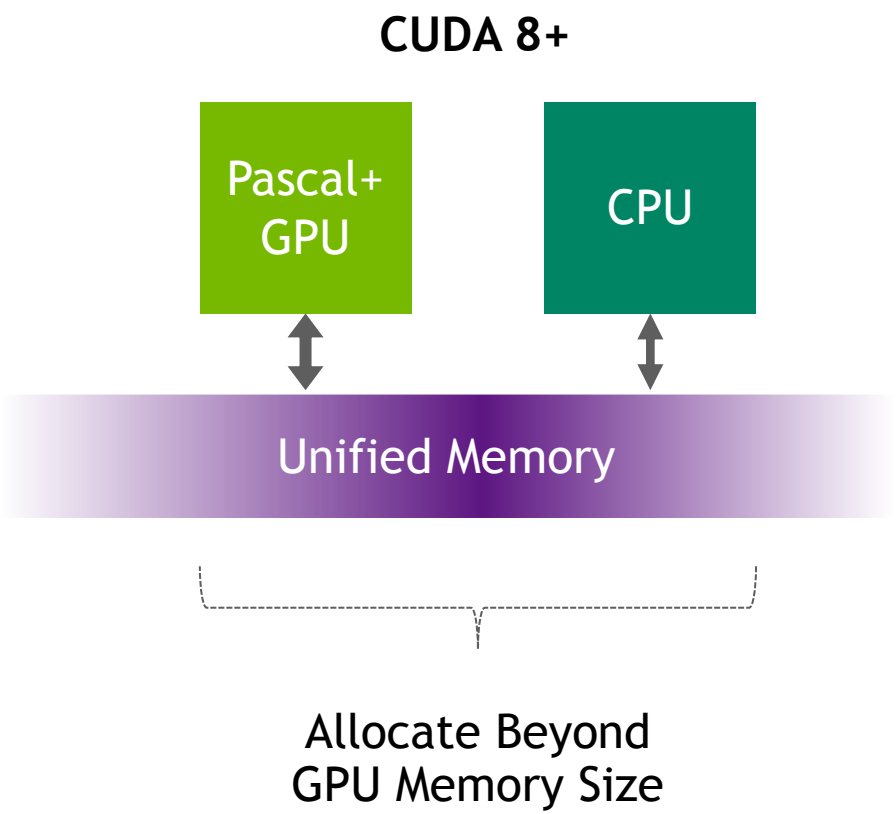
- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit* copy
- Simplifies code porting

Maintain Performance through Data Locality

- Migrate data to accessing processor
- Guarantee global coherence
- Still allows explicit hand tuning

CUDA 8+: UNIFIED MEMORY

Demand Paging For Pascal and Beyond



Enable Large Data Models	Oversubscribe GPU memory Allocate up to system memory size
Simpler Data Accesss	CPU/GPU Data coherence Unified memory atomic operations
Tune Unified Memory Performance	Usage hints via cudaMemAdvise API Explicit prefetching API

SIMPLIFIED MEMORY MANAGEMENT CODE

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

Ordinary CUDA Code

```
void sortfile(FILE *fp, int N) {
  char *data, *d_data;
  data = (char *)malloc(N);
  cudaMalloc(&d_data, N);
  fread(data, 1, N, fp);
  cudaMemcpy(d_data, data, N, …); // 1
  qsort<<<...>>>(data,N,1,compare); // 2
  cudaMemcpy(data, d_data, N, …); // 3

  use_data(data);
  cudaFree(d_data);
  free(data);
}
```

SIMPLIFIED MEMORY MANAGEMENT CODE

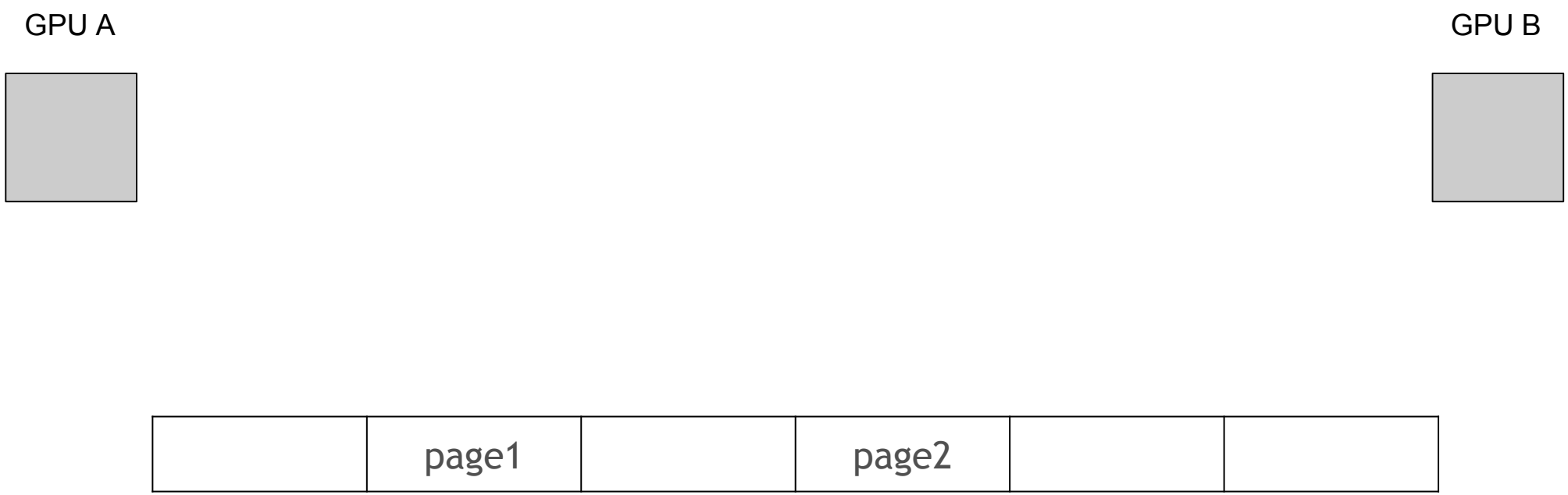
CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA Code with Unified Memory

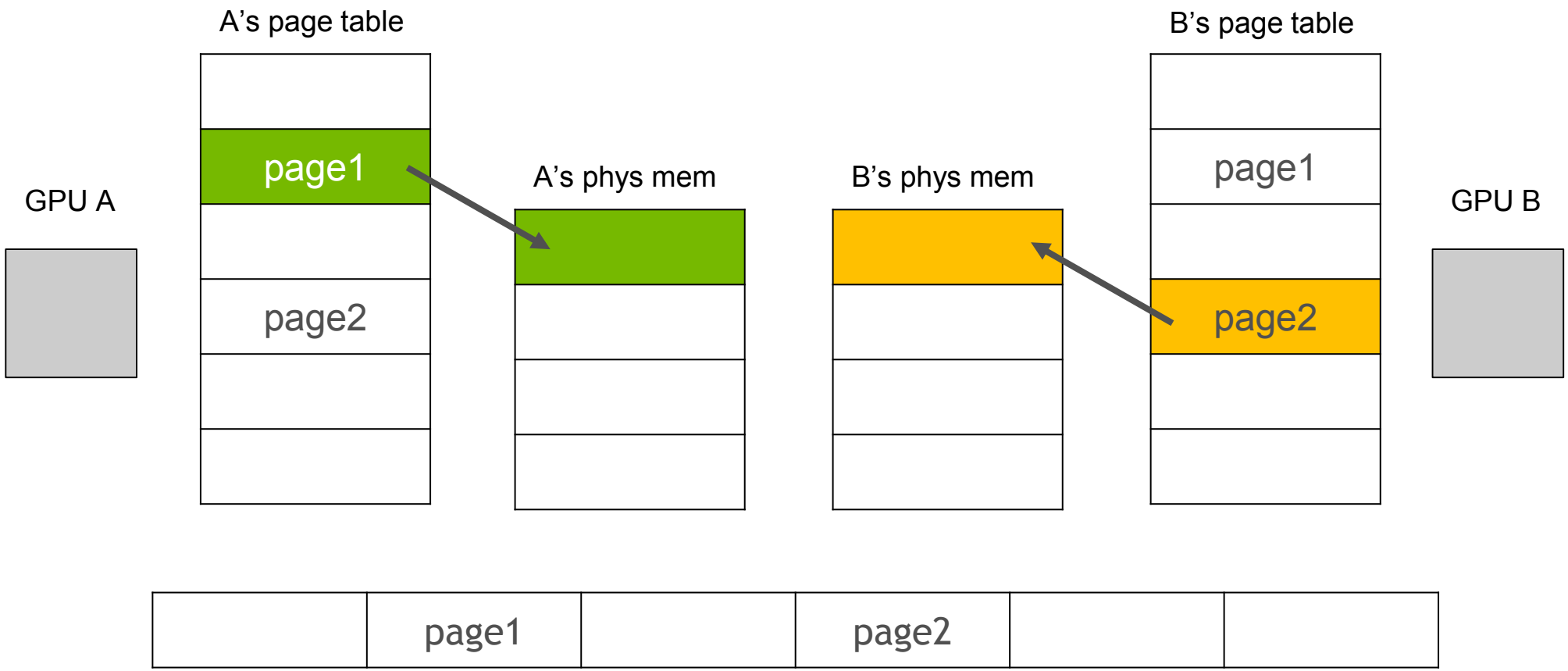
```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

UNIFIED MEMORY BASICS



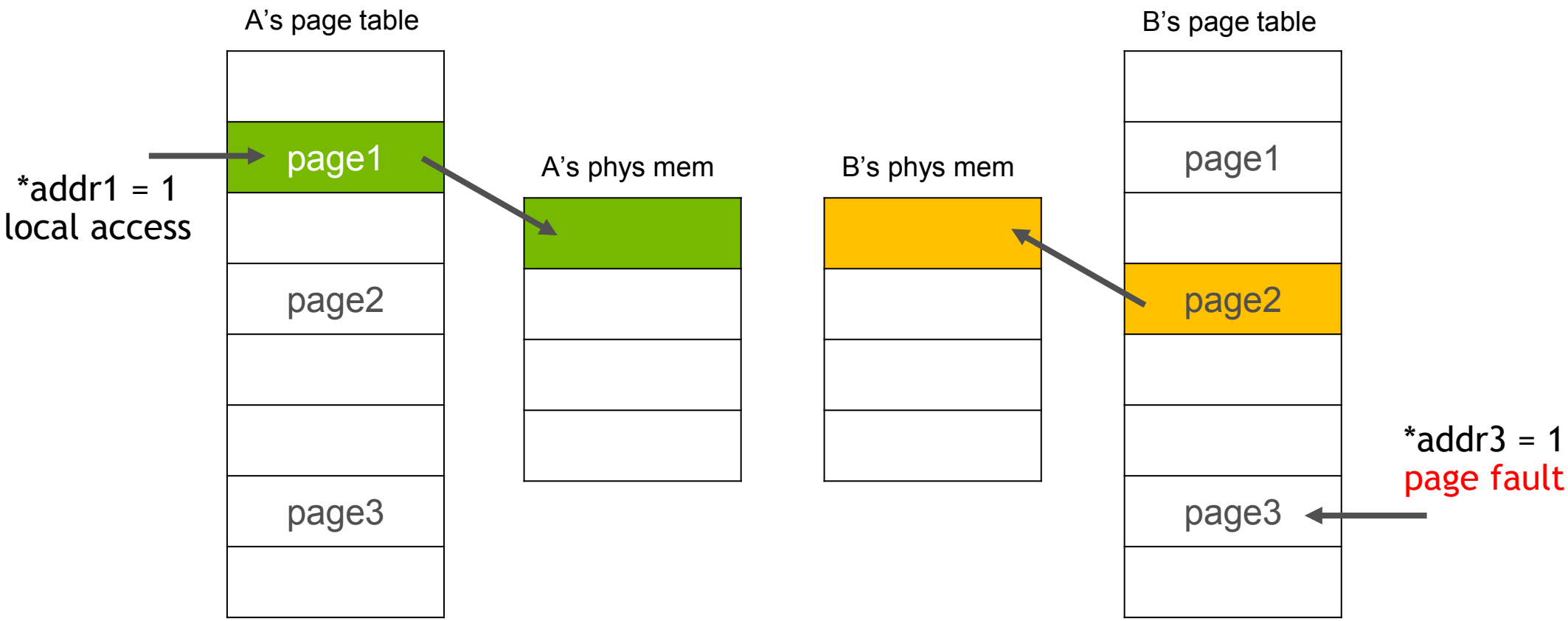
Single virtual memory shared between processors

UNIFIED MEMORY BASICS

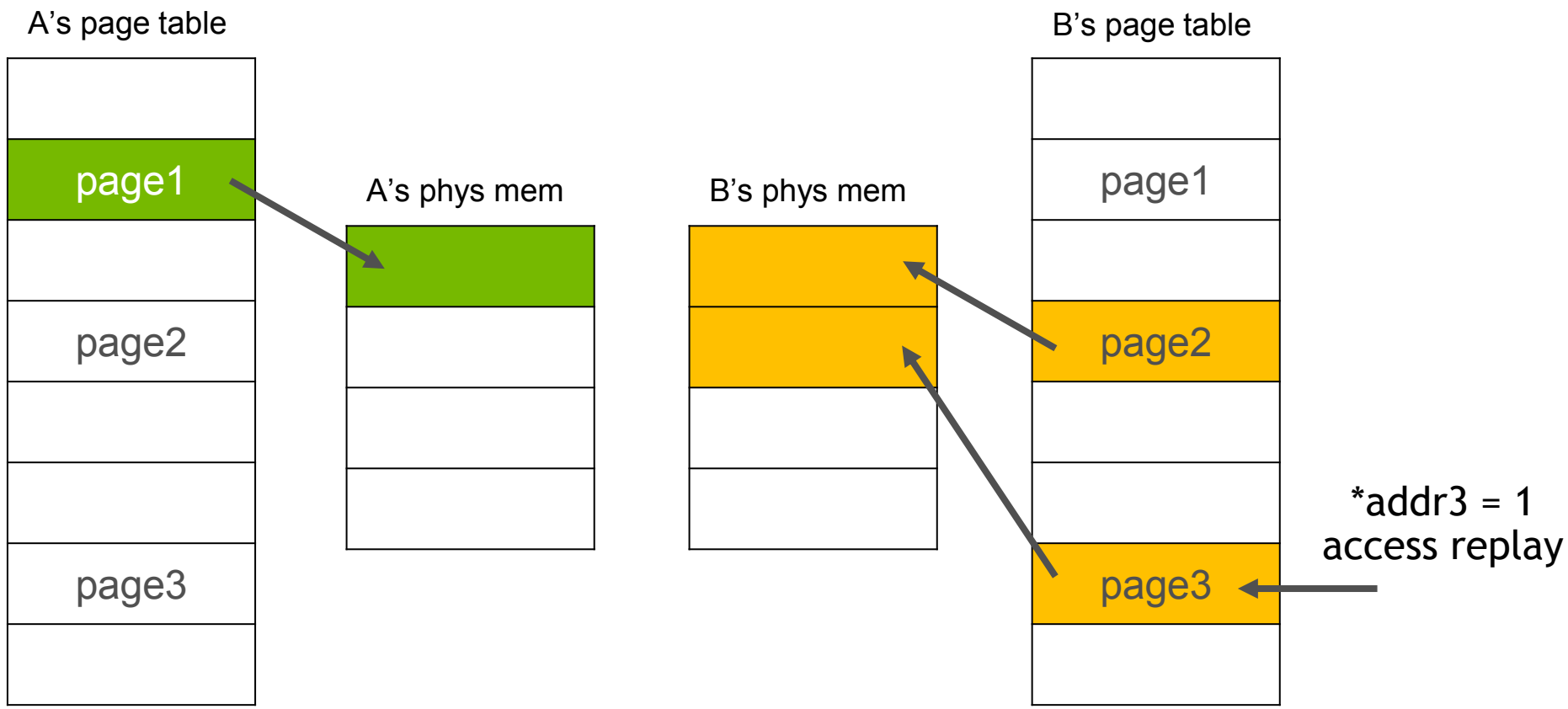


Single virtual memory shared between processors

UNIFIED MEMORY BASICS

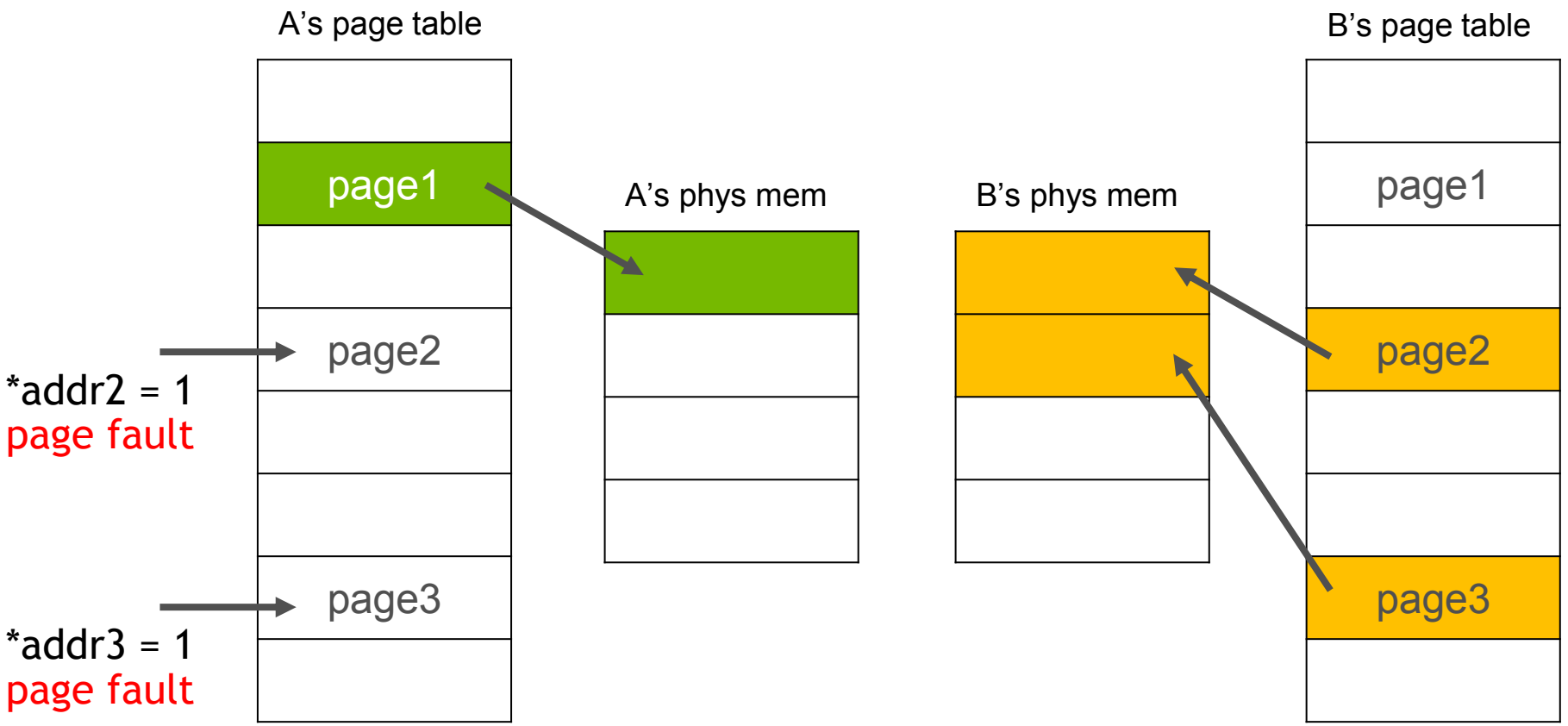


UNIFIED MEMORY BASICS

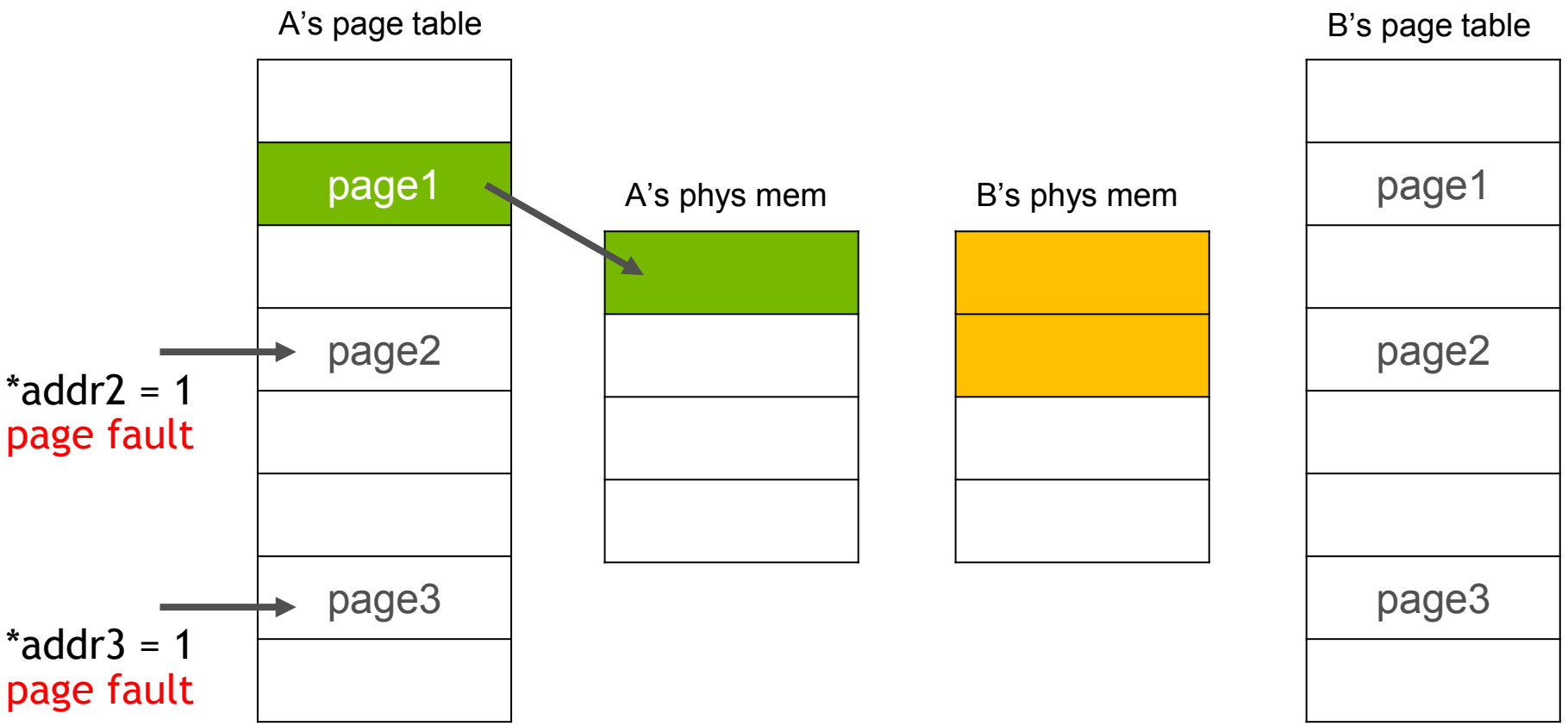


page3 populated and mapped into B's memory

UNIFIED MEMORY BASICS

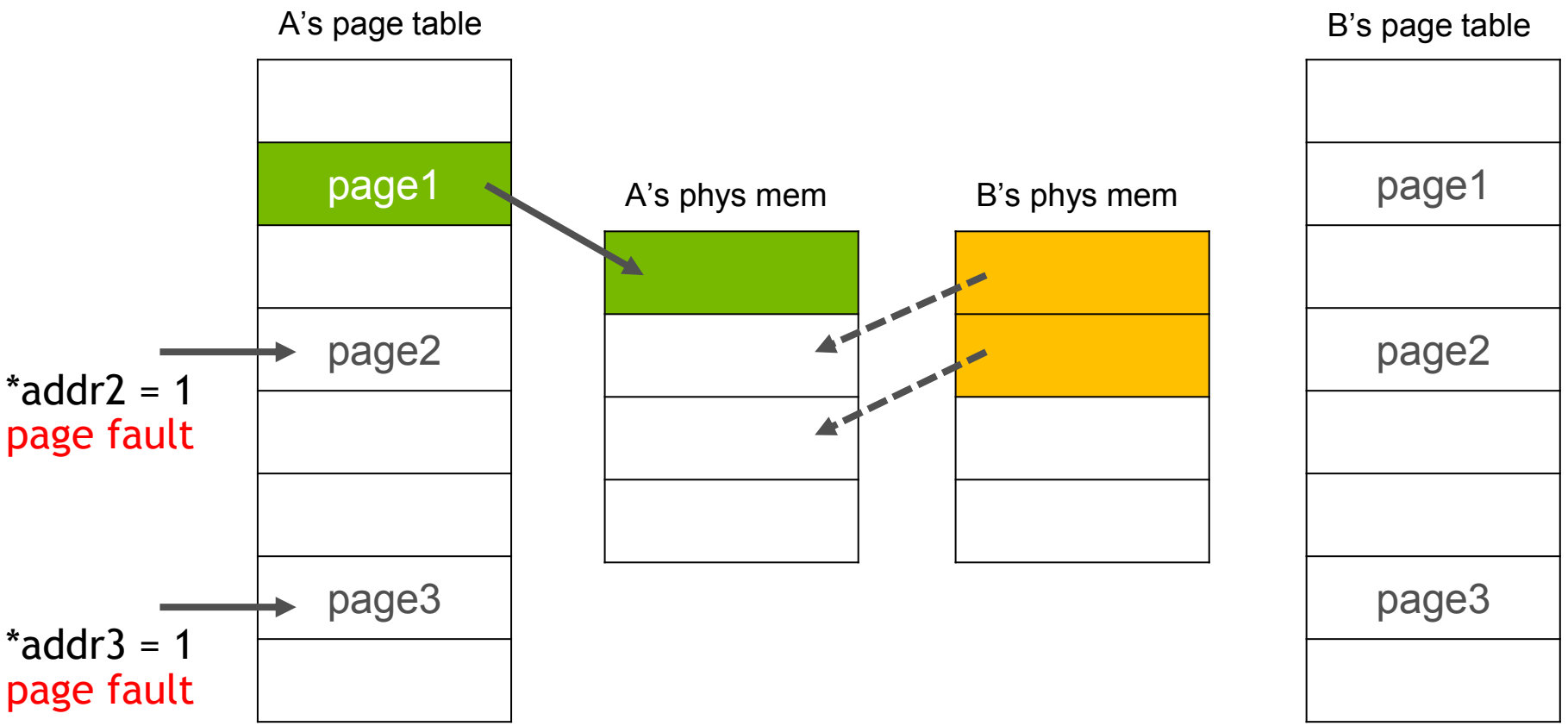


UNIFIED MEMORY BASICS



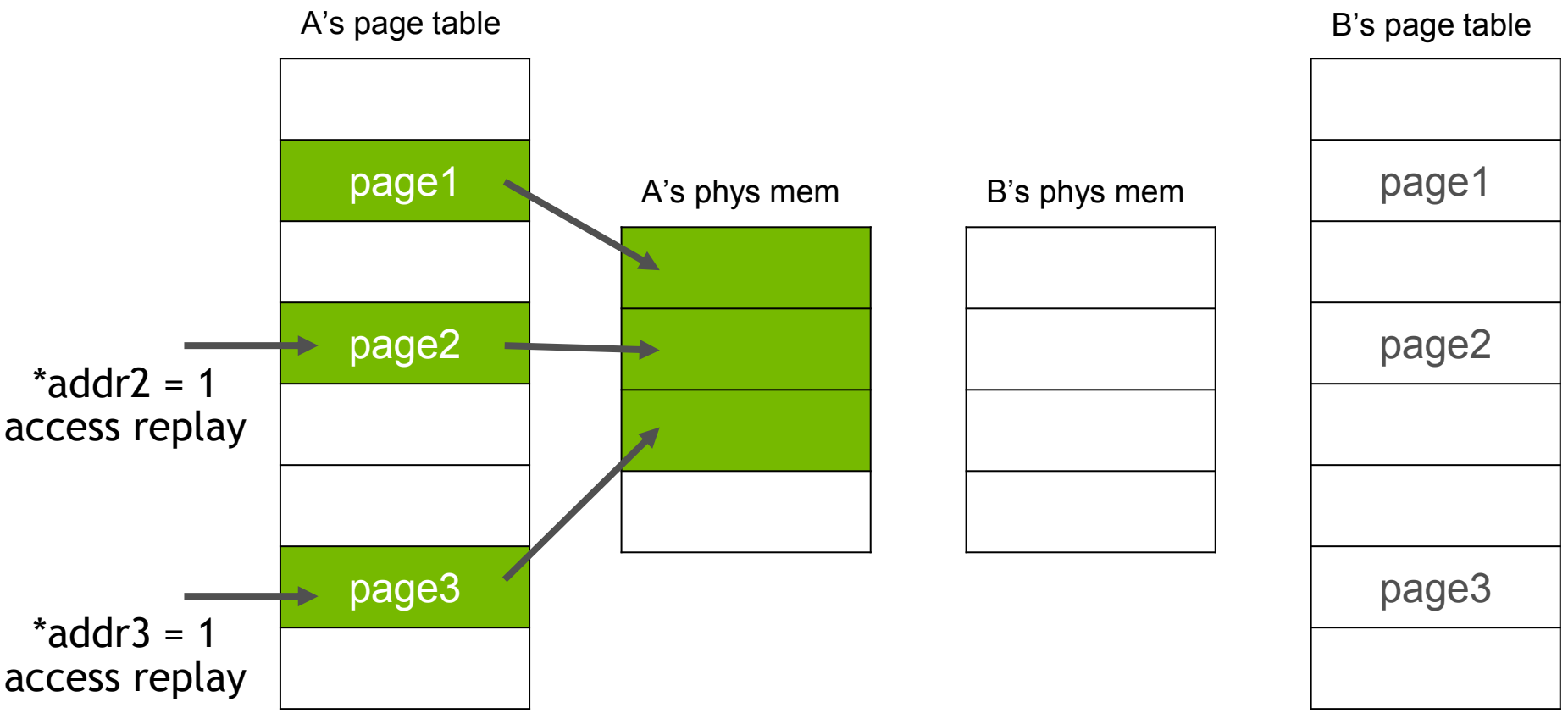
page2 and page3 unmapped from B's memory

UNIFIED MEMORY BASICS



pages data migrated to A's physical memory

UNIFIED MEMORY BASICS



UNIFIED MEMORY EXAMPLE

With On-Demand Paging

```
__global__
void setValue(int *ptr, int index, int val)
{
  ptr[index] = val;
}


void foo(int size) {
  char *data;
  cudaMallocManaged(&data, size);

  memset(data, 0, size);

  setValue<<<...>>>(data, size/2, 5);
  cudaDeviceSynchronize();

  useData(data);

  cudaFree(data);
}
```

Unified Memory allocation

Access all values on CPU

Access one value on GPU

ASIDE: PRE-PASCAL UM REGIME

Summary

- In effect if your device is prior to Pascal (Jetson is a special case)
- In effect if you are on windows OS (CUDA 9.x +).
- Managed data is moved en-masse at point of kernel launch (even data that your kernel may not appear to explicitly touch)
- After a kernel launch, **cudaDeviceSynchronize()** triggers the runtime to make data available to CPU code again
- No concurrent access, no on-demand migration to GPU, no oversubscription
- Just use **cudaMallocManaged()** where you would use **malloc()**, or **new**
- Use **cudaFree()** instead of **free()**, or **delete**

UNIFIED MEMORY ON PASCAL+

GPU Memory Oversubscription

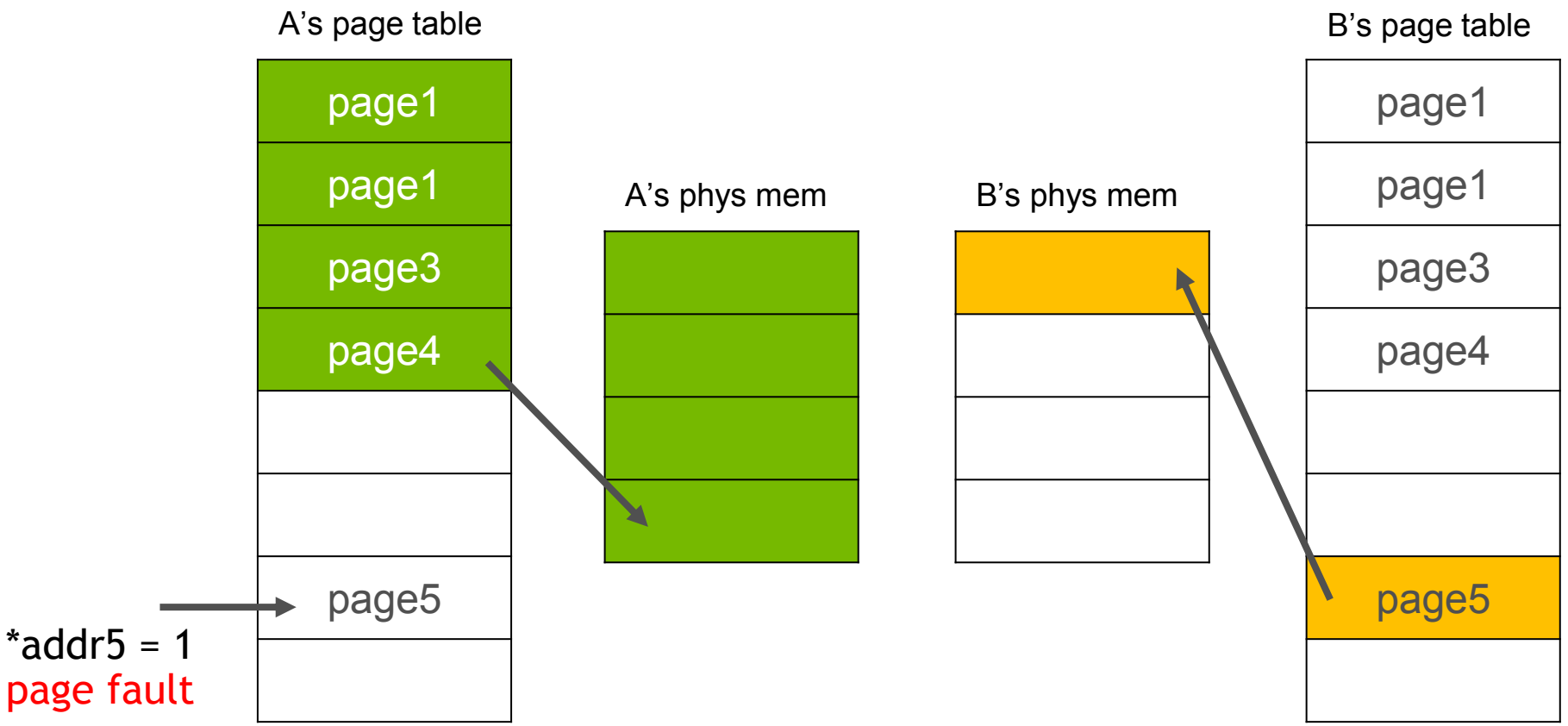
```
void foo() {
  // Assume GPU has 16 GB memory
  // Allocate 64 GB
  char *data;
  // be careful with size type:
  size_t size = 64ULL*1024*1024*1024;
  cudaMallocManaged(&data, size);
}
```

64 GB allocation

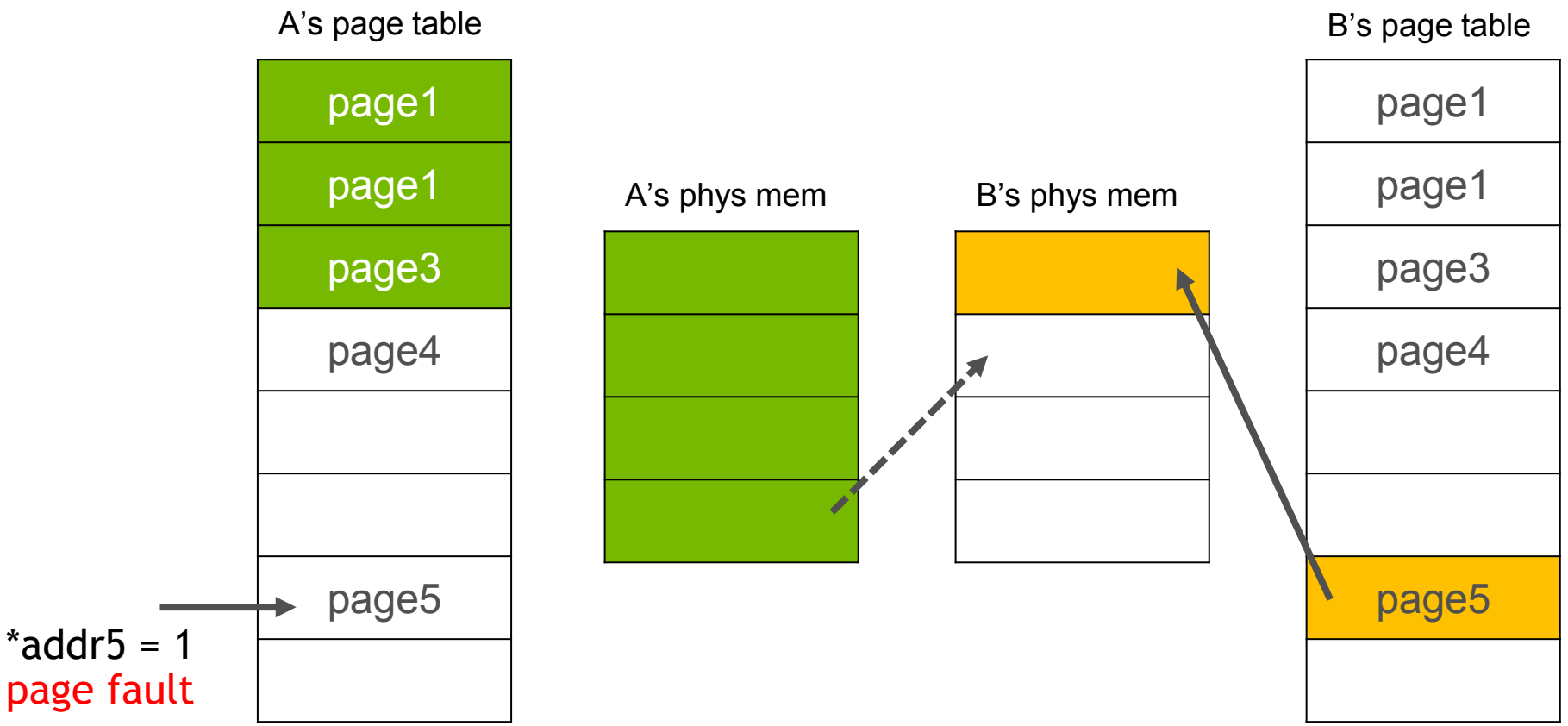
Pascal supports allocations where only a subset of pages reside on GPU. Pages can be migrated to the GPU on demand.

Fails on Kepler/Maxwell

MEMORY OVERSUBSCRIPTION

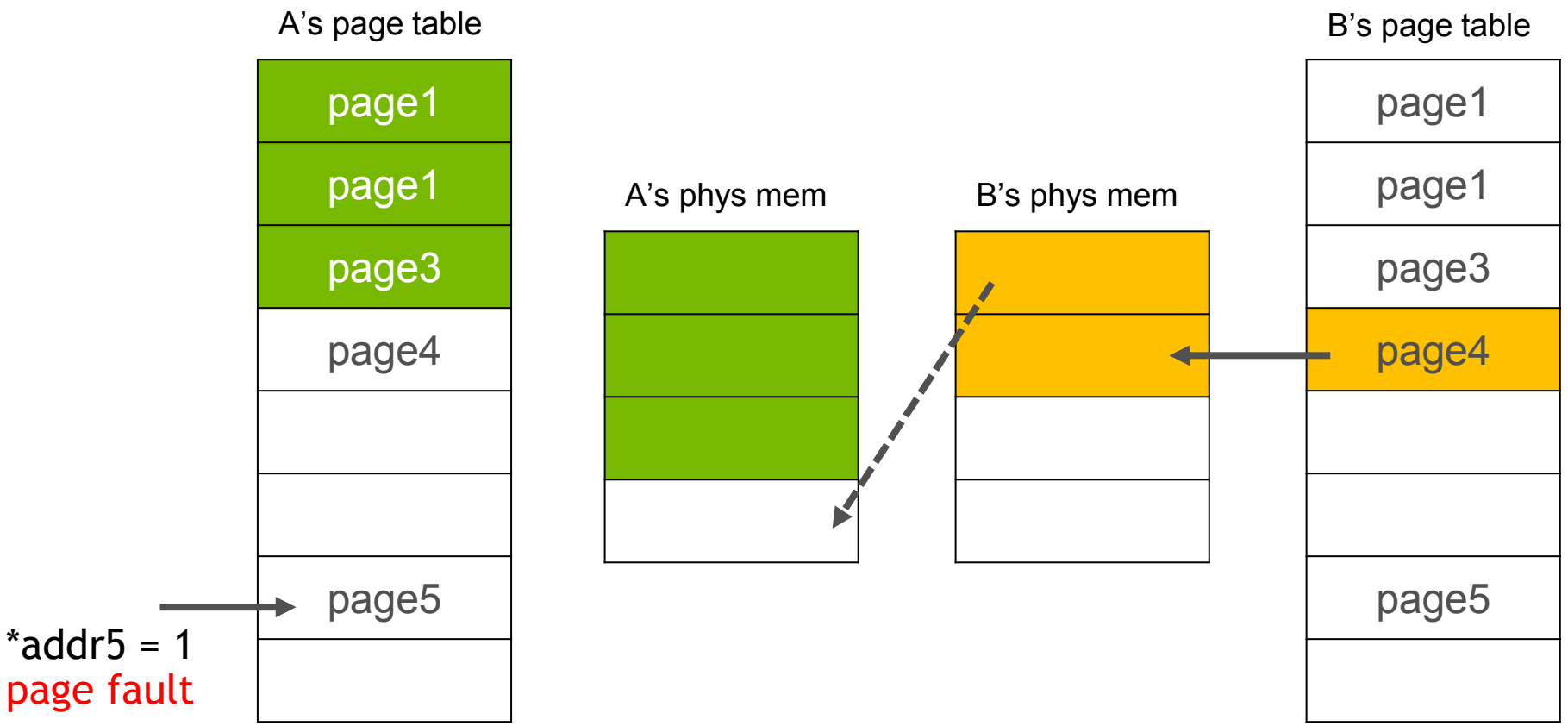


MEMORY OVERSUBSCRIPTION



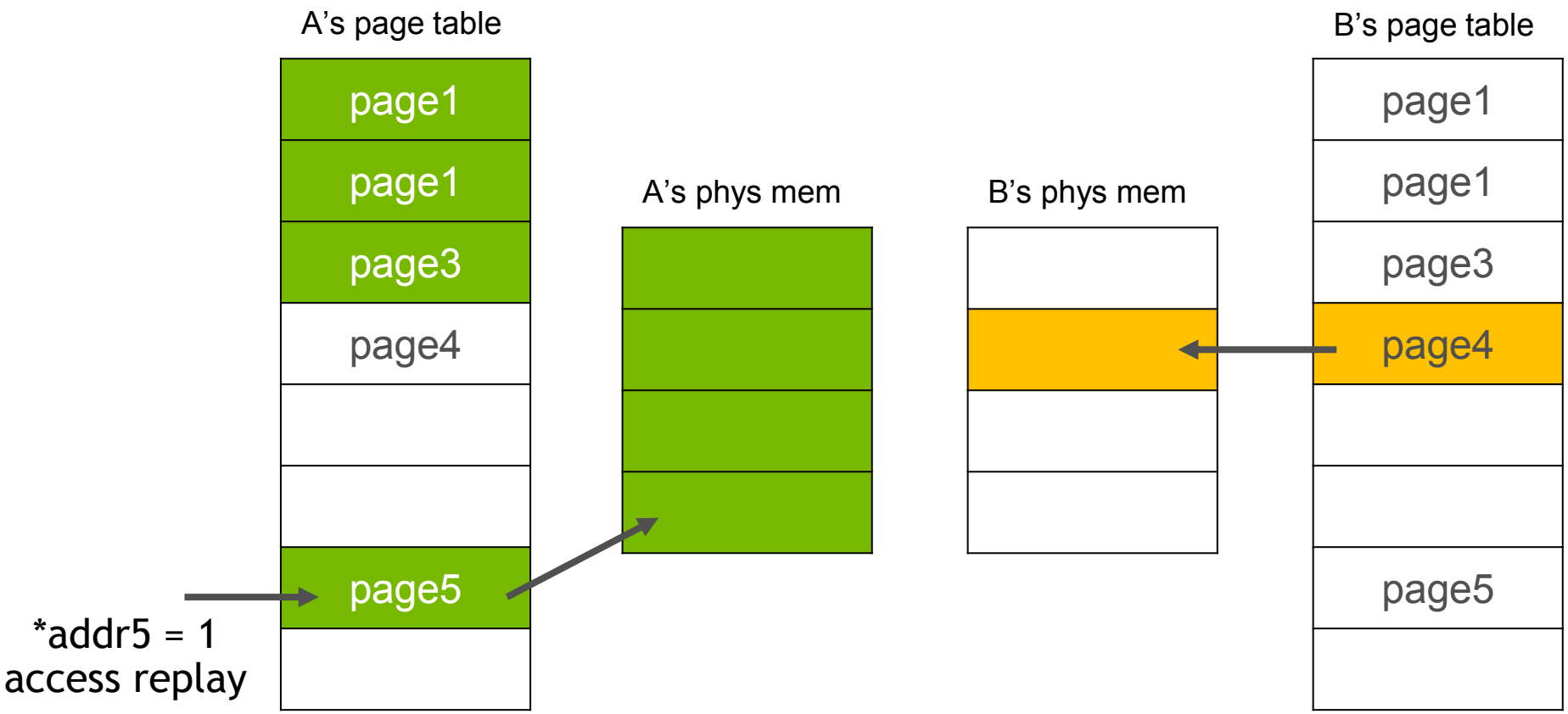
page4 unmapped from A's memory and migrated

MEMORY OVERSUBSCRIPTION



page4 mapped in B's memory, page5 unmapped and migrated to A

MEMORY OVERSUBSCRIPTION



UNIFIED MEMORY ON PASCAL+

Concurrent CPU/GPU Access to Managed Memory

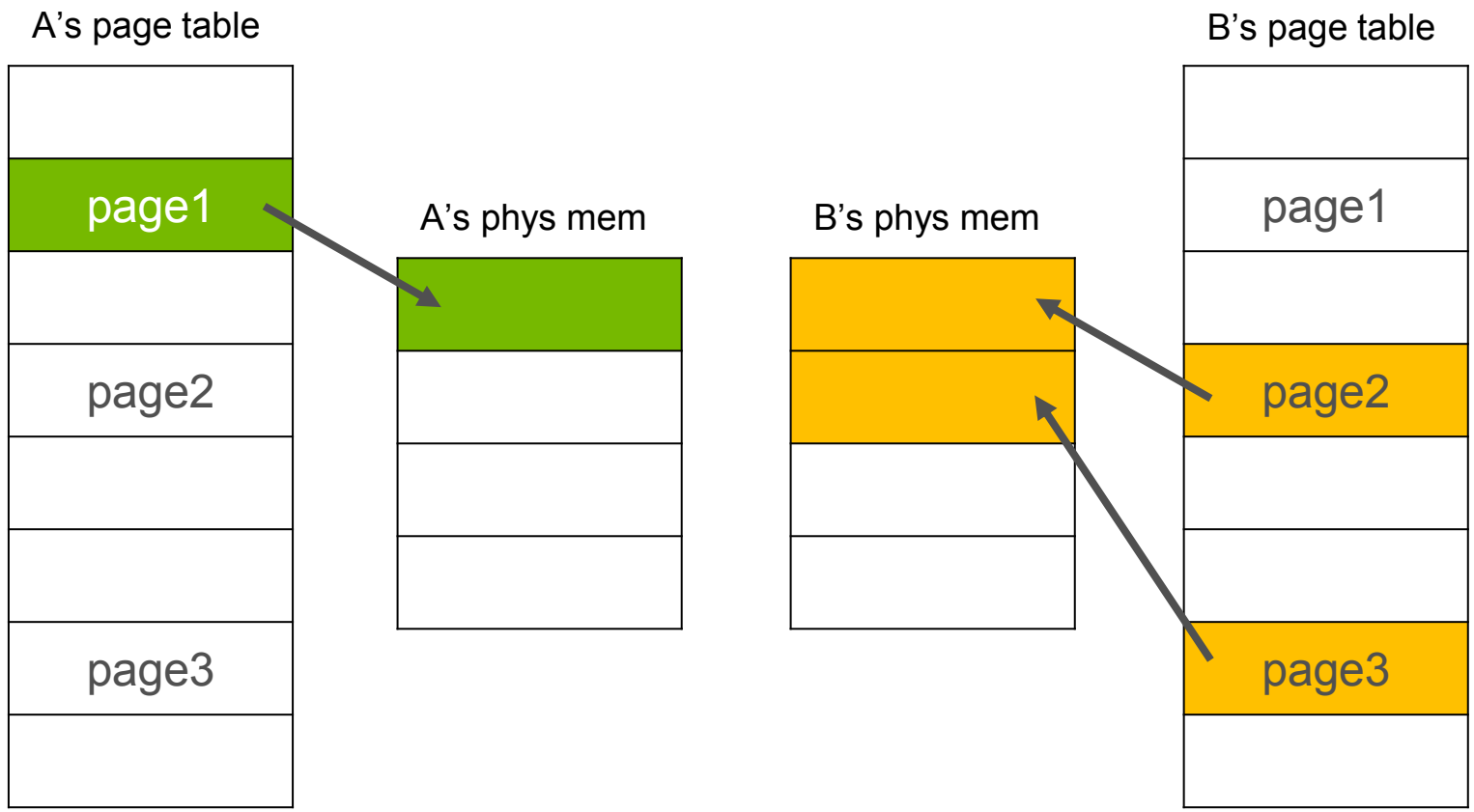
```
__global__ void mykernel(char *data) {
  data[1] = 'g';
}

void foo() {
  char *data;
  cudaMallocManaged(&data, 2);

  mykernel<<<...>>>(data);
  // no synchronize here
  data[0] = 'c';

  cudaFree(data);
}
```

OK on Pascal+: just a page fault

Concurrent CPU access to 'data' on previous GPUs caused a fatal segmentation fault

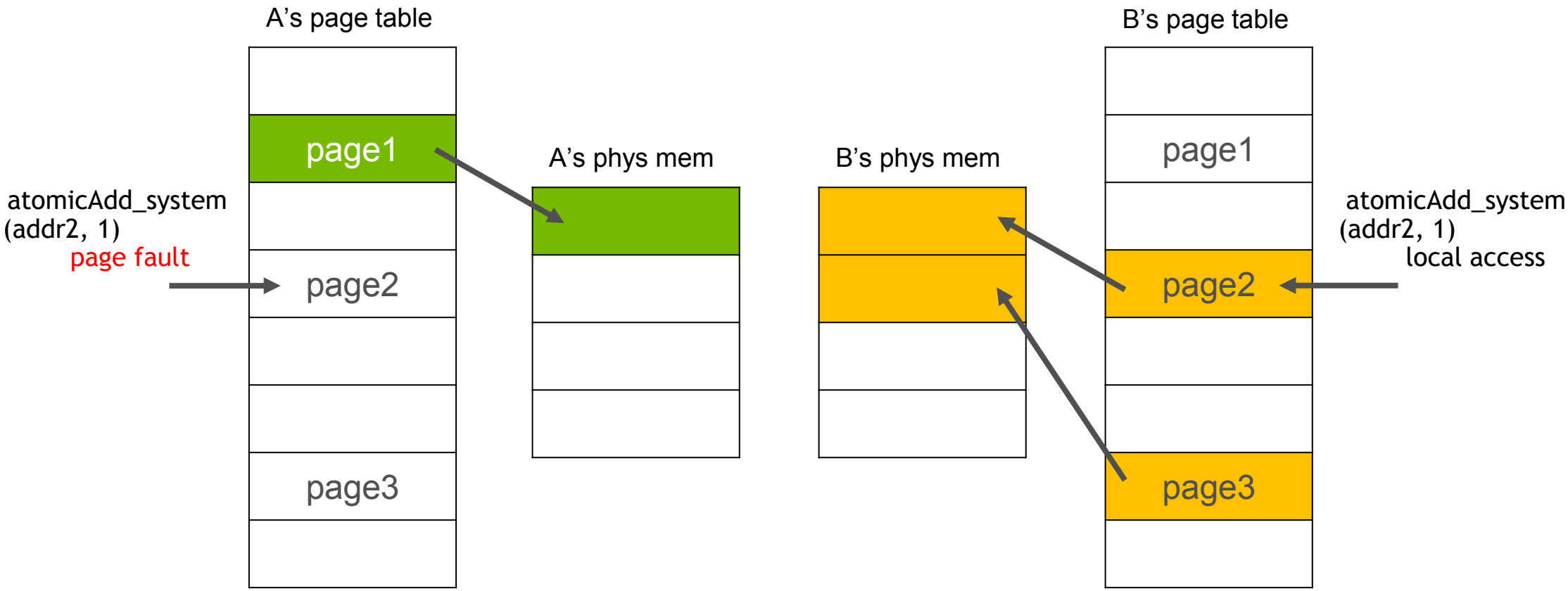
Note that there may still be ordering issues or data visibility issues; UM concurrency does not provide any ordering or visibility guarantees, but see system-wide atomics

CONCURRENT ACCESS



CONCURRENT ACCESS

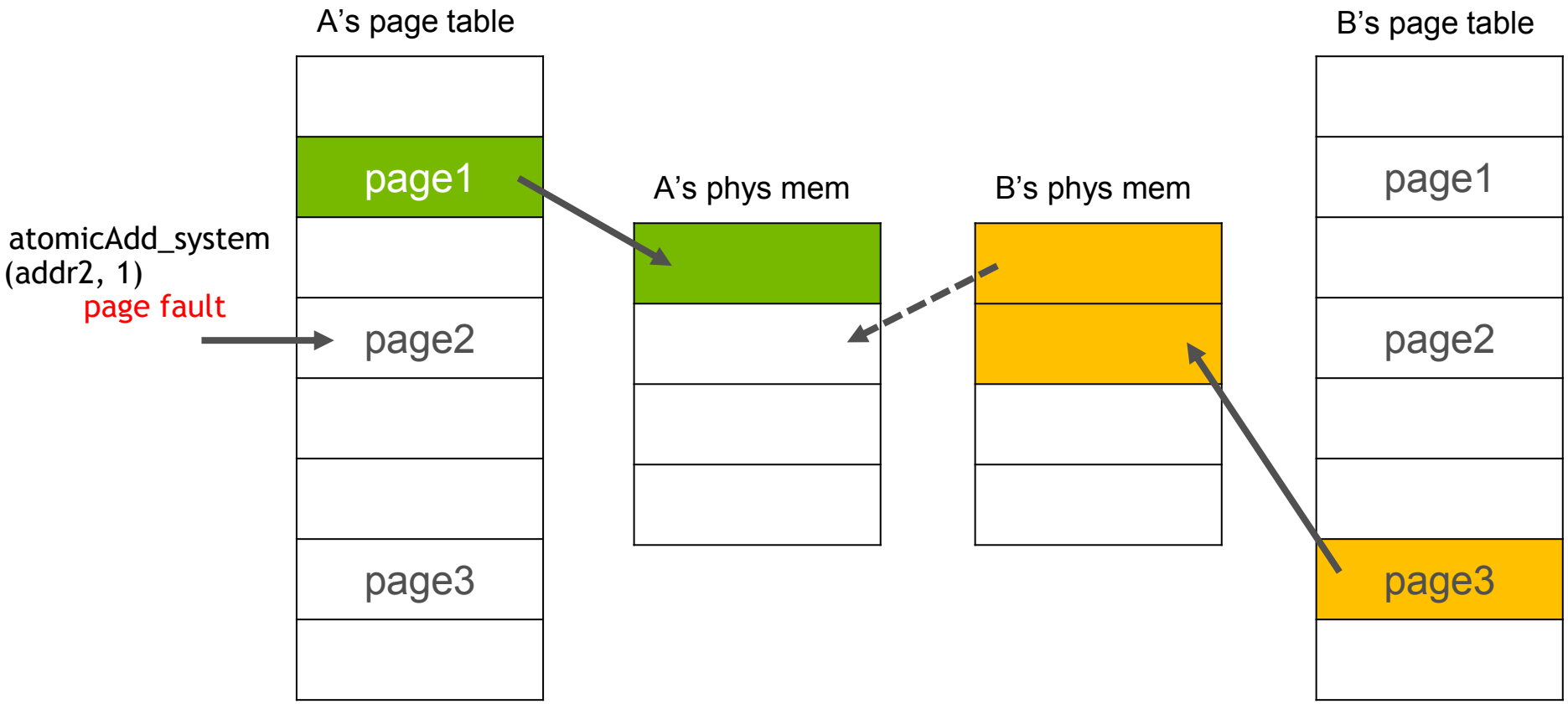
Exclusive Access*



*this is a possible implementation and to guarantee this behavior you need to use cudaMemAdvise policies

CONCURRENT ACCESS

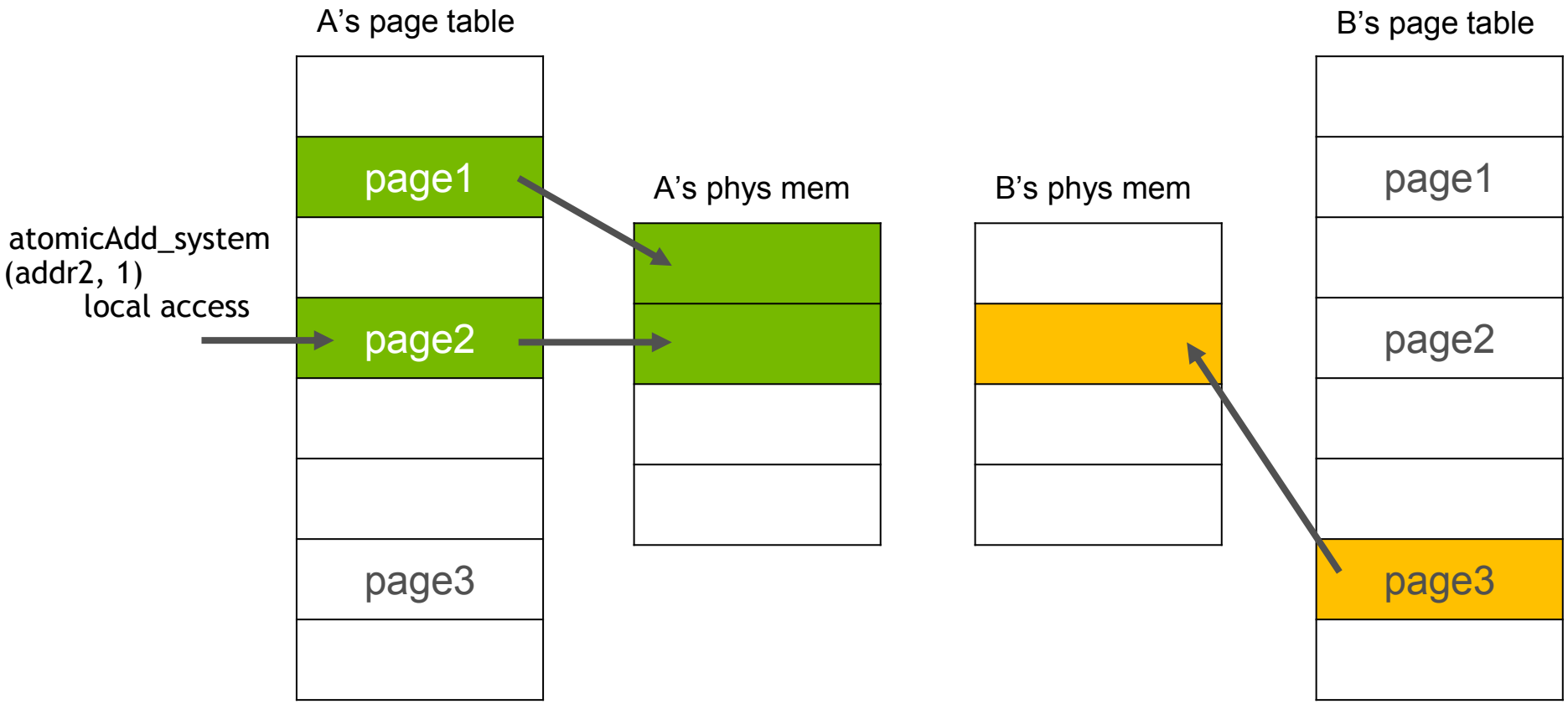
Exclusive Access



page2 unmapped in B's memory and migrated to A

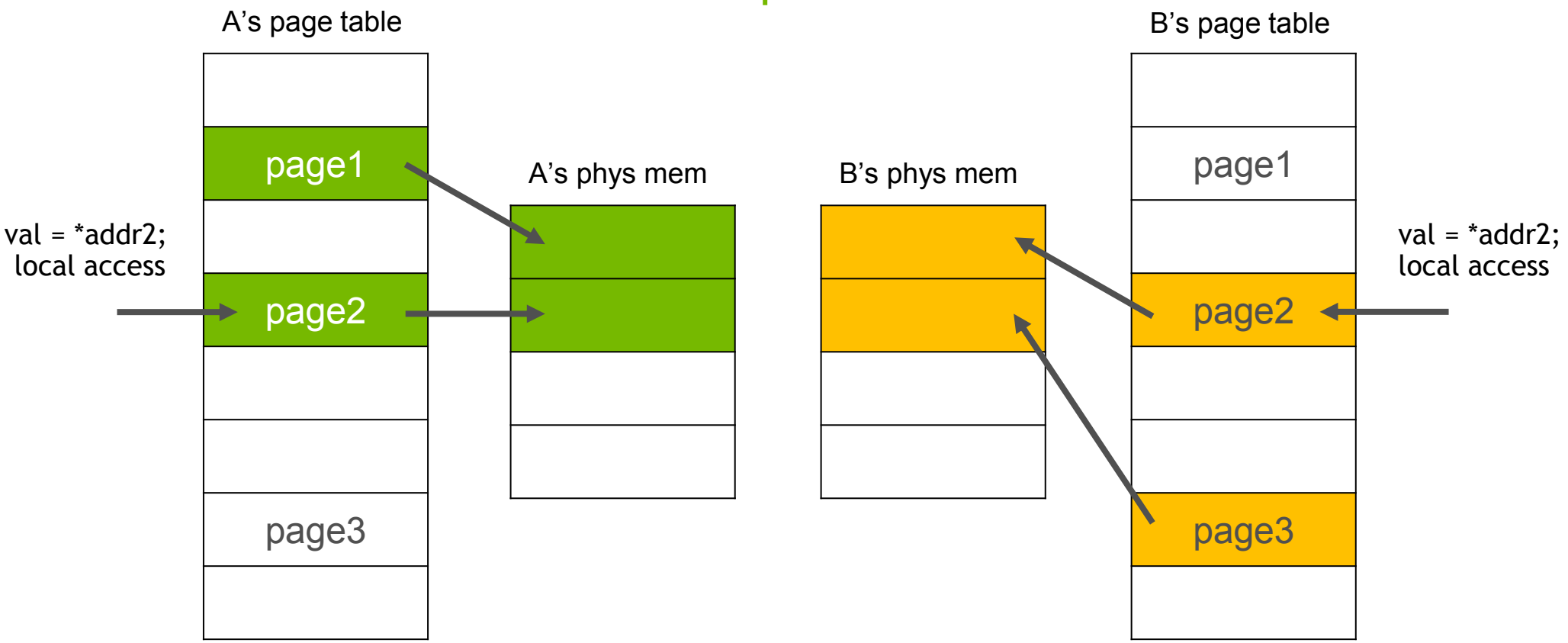
CONCURRENT ACCESS

Exclusive Access



CONCURRENT ACCESS

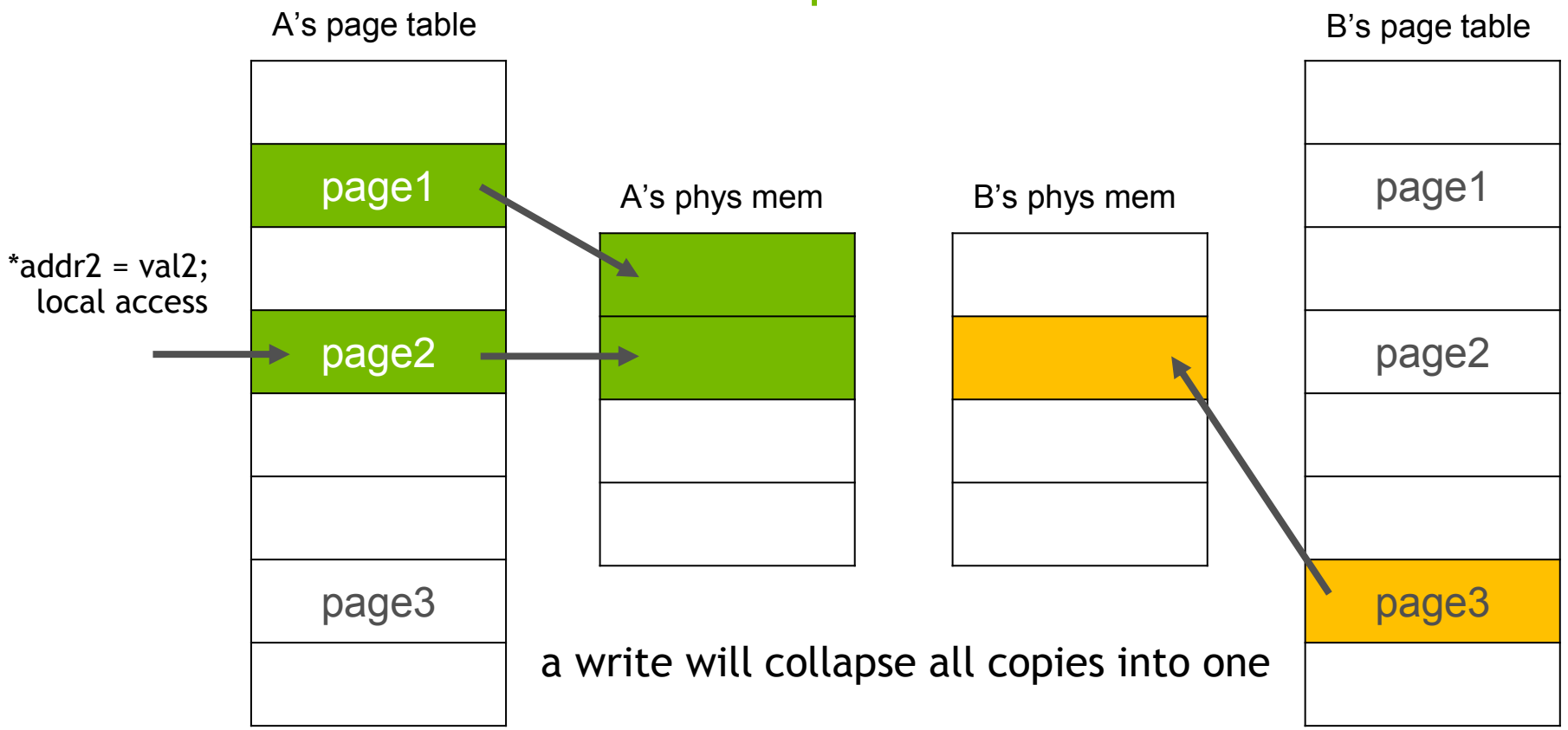
Read duplication*



*each processor must maintain its own page table

CONCURRENT ACCESS

Read duplication: write



UNIFIED MEMORY ON PASCAL+

System-Wide Atomics

```
__global__ void mykernel(int *addr) {
  atomicAdd_system(addr, 10);
}

void foo() {
  int *addr;
  cudaMallocManaged(addr, 4);
  *addr = 0;

  mykernel<<<...>>>(addr);
  // cpu atomic:
  __sync_fetch_and_add(addr, 10);
}
```

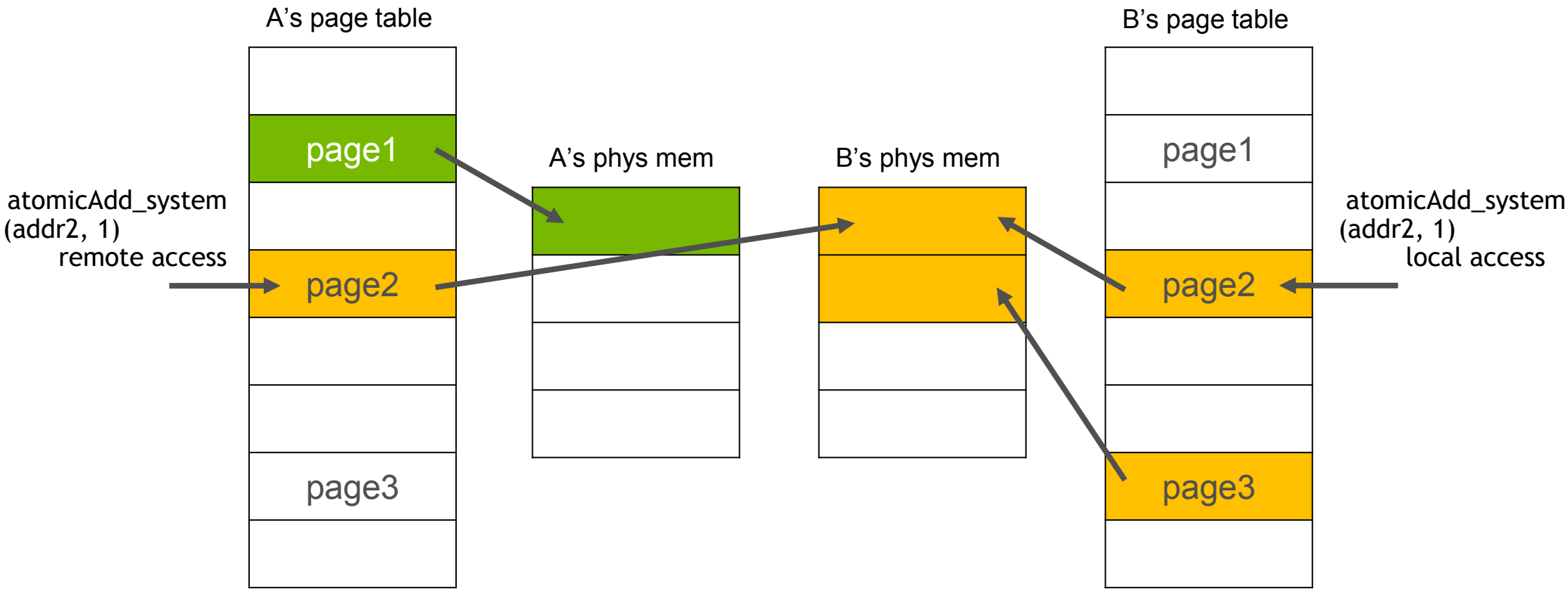
Pascal enables system-wide atomics

- Direct support of atomics over NVLink
- Software-assisted over PCIe

System-wide atomics not available on Kepler / Maxwell

CONCURRENT ACCESS

Atomics over NVLINK*



*both processors need to support atomic operations

USE CASE: DEEP COPY

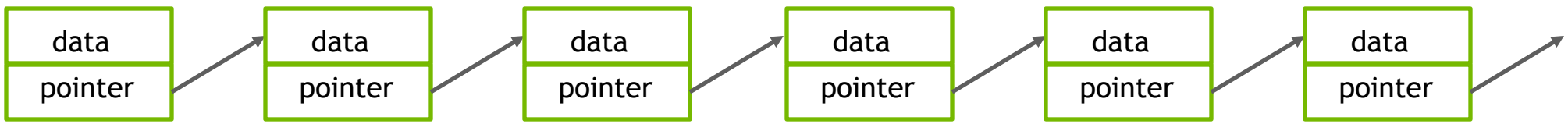
```
struct dataElem {
  int key;
  int len;
  char *name;
}
```

```
char buffer[len];
```

- Both entities (object and buffer) need to be transferred to the device
- Pointer in object needs to be “fixed” to point to new address on device for device copy of buffer

```
void launch(dataElem *elem, int N) { // an array of dataElem
  dataElem *d_elem;
  // Allocate storage for array of struct and copy array to device
  cudaMalloc(&d_elem, N*sizeof(dataElem));
  cudaMemcpy(d_elem, elem, N*sizeof(dataElem), cudaMemcpyHostToDevice);
  for (int i = 0; i < N; i++){ // allocate/fixup each buffer separately
    char *d_name;
    cudaMalloc(&d_name, elem[i].len);
    cudaMemcpy(d_name, elem[i].name, elem[i].len, cudaMemcpyHostToDevice);
    cudaMemcpy(&(d_elem[i].name), &d_name, sizeof(char *), cudaMemcpyHostToDevice);}
  // Finally we can launch our kernel
  Kernel<<< ... >>>(d_elem);}
```

USE CASE: LINKED LIST



- Similar to deep copy case
- Complex to code up the copy operation
- Unified Memory makes it trivial

USE CASE: C++ OBJECTS

Overloading **new** and **delete**

Overload new and delete in base class

```
class Managed {
public:
  void *operator new(size_t len) {
    void *ptr;
    cudaMallocManaged(&ptr, len);
    cudaDeviceSynchronize();
    return ptr;
  }

  void operator delete(void *ptr) {
    cudaDeviceSynchronize();
    cudaFree(ptr);
  }
};
```

inherit to build string class

```
// Deriving allows pass-by-reference
class umString : public Managed {
  int length;
  char *data;

public:
  // UM copy constructor allows
pass-by-value
  umString (const umString &s) {
    length = s.length;
    cudaMallocManaged(&data,
length);
    memcpy(data, s.data, length);
  }
};
```

USE CASE: C++ OBJECTS

Overloading **new** and **delete**

Inherit to build my class; embedded string

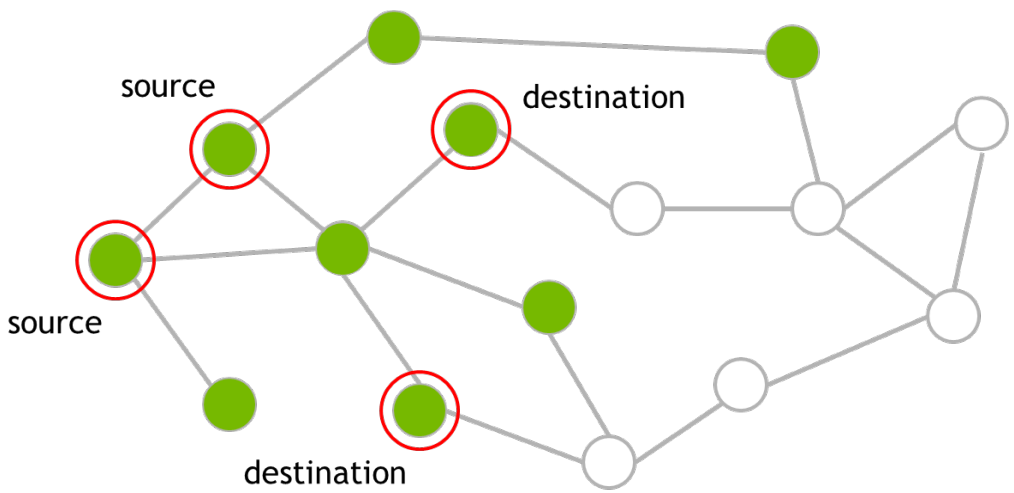
```
// Note “managed” here also

class dataElem : public Managed {
public:
  int key;
  umString name;
};
```

Profit!

```
dataElem *data = new dataElem[N];
…
// C++ now handles our deep copies
Kernel<<< ... >>>(data);}
```

USE CASE: ON-DEMAND PAGING

Graph Algorithms



PERFORMANCE TUNING ON PASCAL+

Demand Paging Impact

This kernel call runs much slower than the Pre-pascal UM 6 case, or the non-UM case.

Each page fault triggers service overhead.

Relying on page faults to move large amounts of data, page-by-page, with overhead on each page, is inefficient.

For bulk movement, a single “memcpy-like” operation is much more efficient

```
__global__ void kernel(float *data){
  int idx = …;
  data[idx] = val;}
…
int n = 256*256;
float *data;
cudaMallocManaged(&data, n*sizeof(float);
Kernel<<<256,256>>>(data);
```

PERFORMANCE TUNING ON PASCAL+

Prefetching

Explicit prefetching:

```
cudaMemPrefetchAsync(ptr, length, destDevice, stream)
```

UM alternative to cudaMemcpy(Async)

Can target any GPU and also the CPU

“Restores” performance

```
__global__ void kernel(float *data){
  int idx = …;
  data[idx] = val;}
…
int n = 256*256;
int ds = n*sizeof(float);
float *data;
cudaMallocManaged(&data, ds);
cudaMemPrefetchAsync(data, ds, 0);
Kernel<<<256,256>>>(data);
cudaMemPrefetchAsync(data, ds,
cudaCpuDeviceId); // copy back to host
```

PERFORMANCE TUNING ON PASCAL+

Explicit Memory Hints

Advise runtime on expected memory access behaviors with:

```
cudaMemAdvise(ptr, count, hint, device);
```

Hints:

- `cudaMemAdviseSetReadMostly`: Specify read duplication
- `cudaMemAdviseSetPreferredLocation`: suggest best location
- `cudaMemAdviseSetAccessedBy`: suggest mapping

Hints don’t trigger data movement by themselves

PERFORMANCE TUNING ON PASCAL+

Hints: `cudaMemAdviseSetReadMostly`

Data will usually be read-only

UM system will make a “local” copy of the data for each processor that touches it

If a processor writes to it, this invalidates all copies except the one written.

Device argument is ignored

READ DUPLICATION

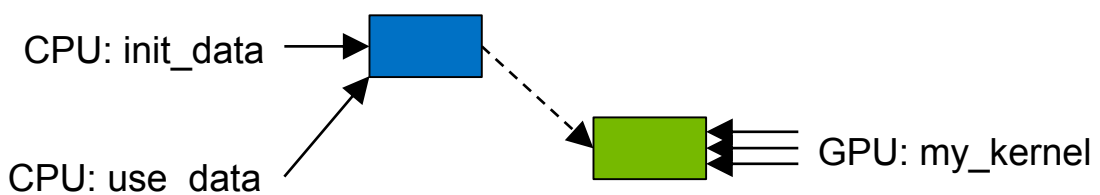
Usage example

```
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);

cudaFree(data);
```

The prefetch creates a copy instead of moving data

Both processors can read data **simultaneously** without faults

Writes will collapse all copies into one, subsequent reads will fault and duplicate



PERFORMANCE TUNING ON PASCAL+

Hints: `cudaMemAdviseSetPreferredLocation`

Suggests which processor is the best location for data

Does not automatically cause migration

Data will be migrated to the preferred processor on-demand (or if prefetched)

If possible, data (P2P) mappings will be provided when other processors touch it

If mapping is not possible, data is migrated

Volta+ adds *access counters* to help GPU make good decisions for you

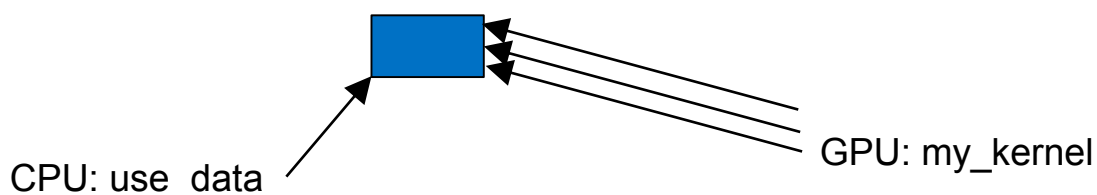
PREFERRED LOCATION

Page population on first-touch

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


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

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


use_data(data, N);

cudaFree(data);
```

The kernel will *page fault*, populate pages on the CPU and generate direct mapping to data on the CPU

Pages are populated on the preferred location if the faulting processor can access it



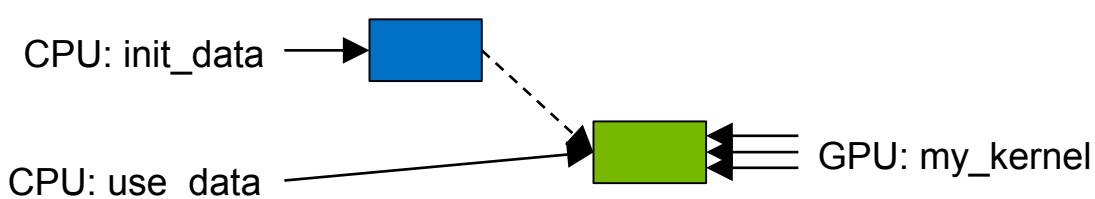
PREFERRED LOCATION ON P9+V100

CPU can directly access GPU memory

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

init_data(data, N);

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

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


use_data(data, N);

cudaFree(data);
```

The kernel will *page fault* and migrate data to the GPU

CPU will fault and access data directly instead of migrating



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

PERFORMANCE TUNING ON PASCAL+

Hints: `cudaMemAdviseSetAccessedBy`

Does not cause movement or affect location of data

Indicated processor receives a (P2P) mapping to the data

If the data is migrated, mapping is updated

Objective: provide access without incurring page faults

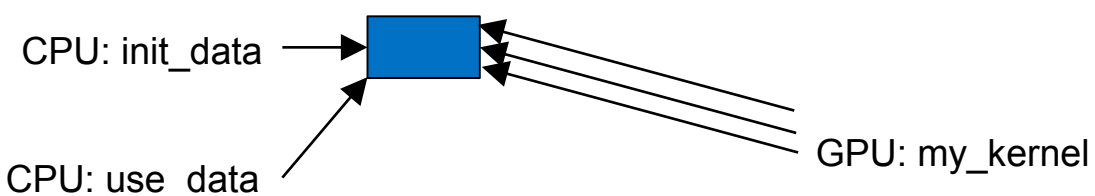
ACCESSED BY

Usage example

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

init_data(data, N);

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

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


use_data(data, N);

cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, **no page faults** will be generated

Memory can move freely to other processors and mapping will carry over



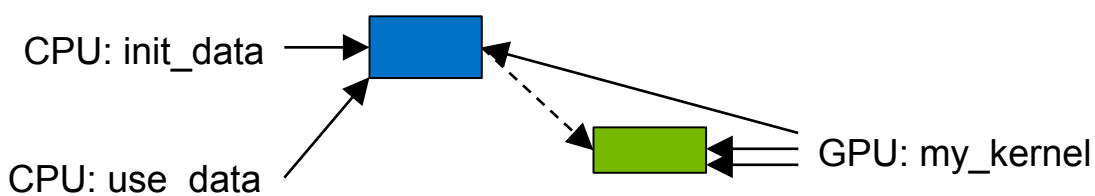
ACCESSED BY

Using access counters on Volta

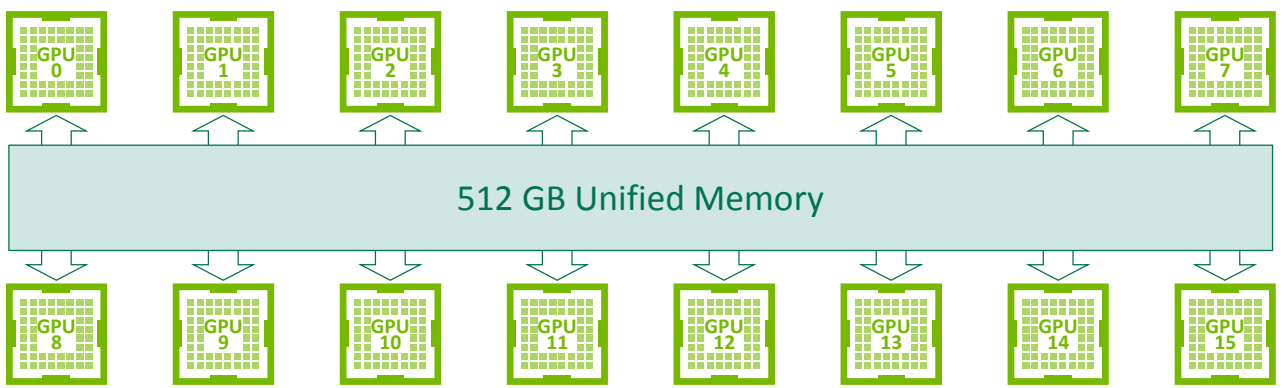
```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

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

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


use_data(data, N);

cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, **no page faults** will be generated

Access counters may eventually trigger migration of **frequently accessed pages** to the GPU



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

ENABLING MULTI-GPU

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);
```

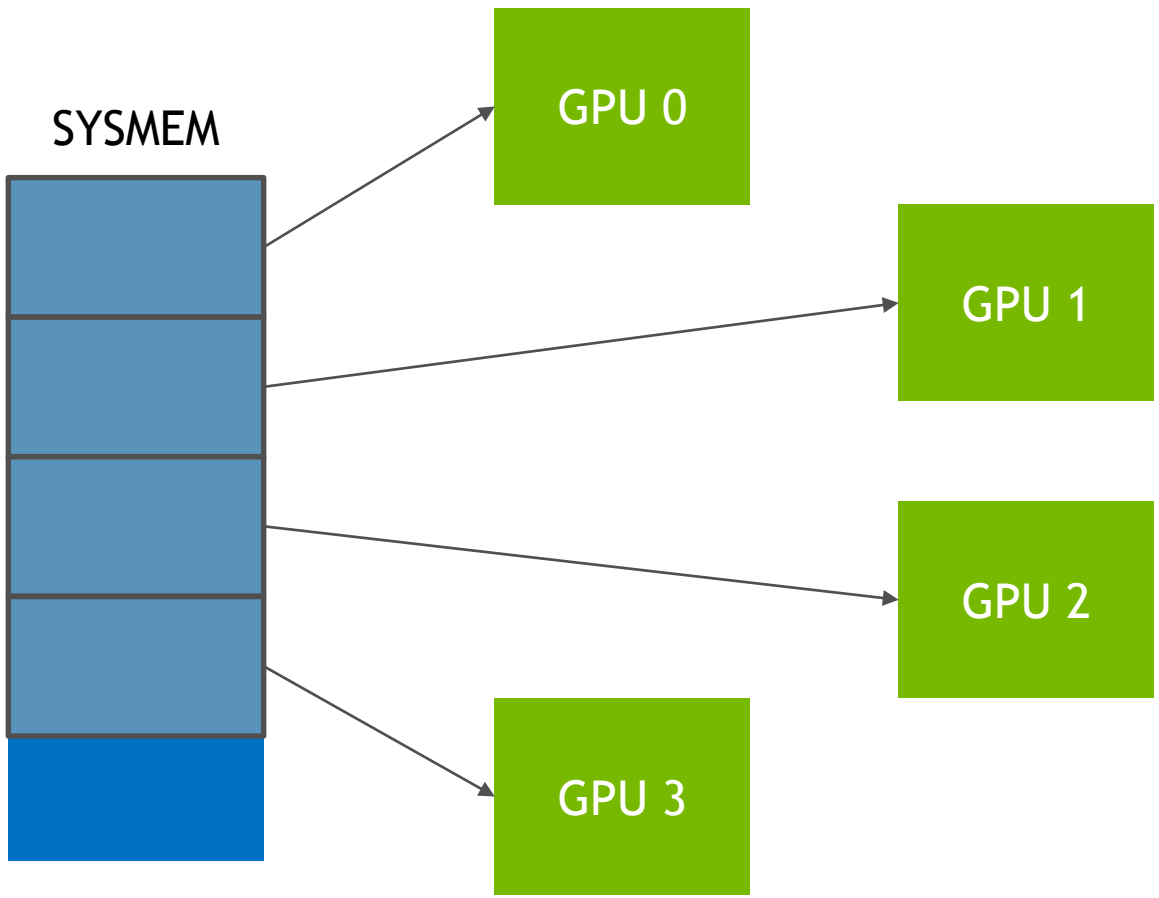
Multi-GPU

```
__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 < numGPUs; i++) {
  cudaSetDevice(i);
  kernel<<<grid/numGPUs, block>>>(data, i);
}
```

update blockIdx.x

update launch config

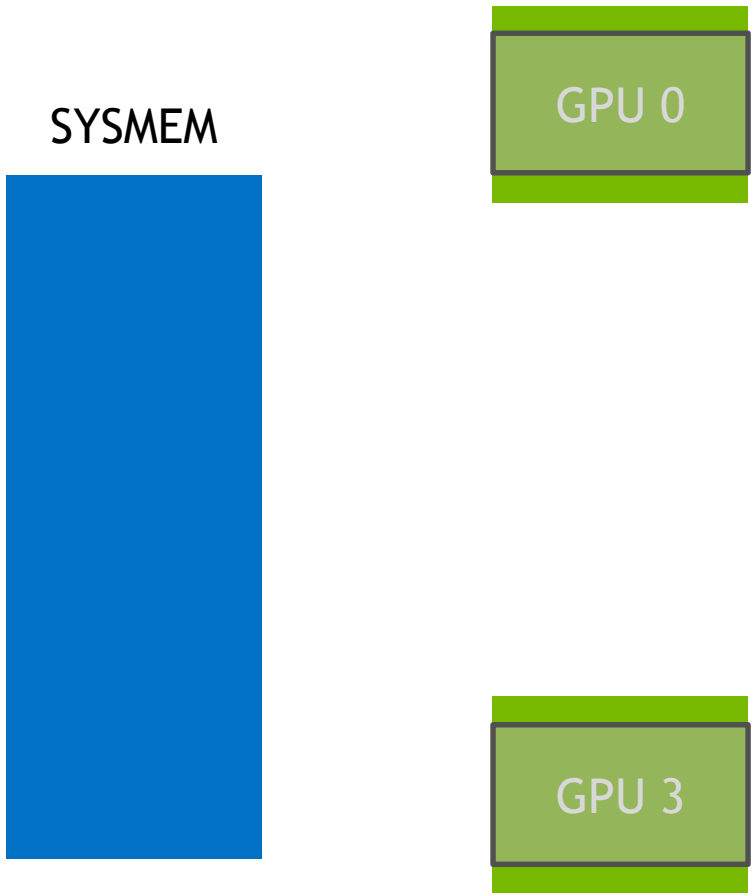
MULTI-GPU WITH UNIFIED MEMORY



GPU kernels
initiate migrations

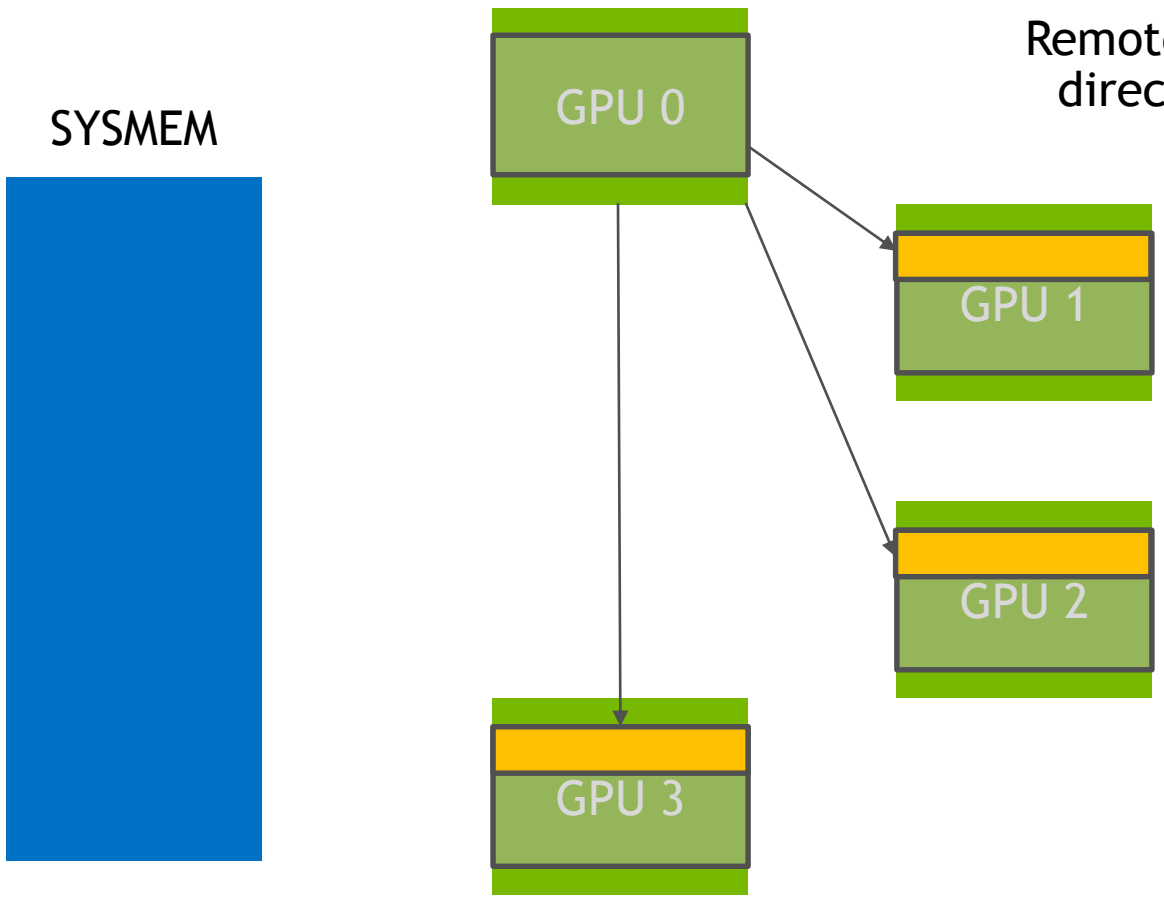
MULTI-GPU WITH UNIFIED MEMORY

Data *automatically* partitioned between GPUs on first-touch

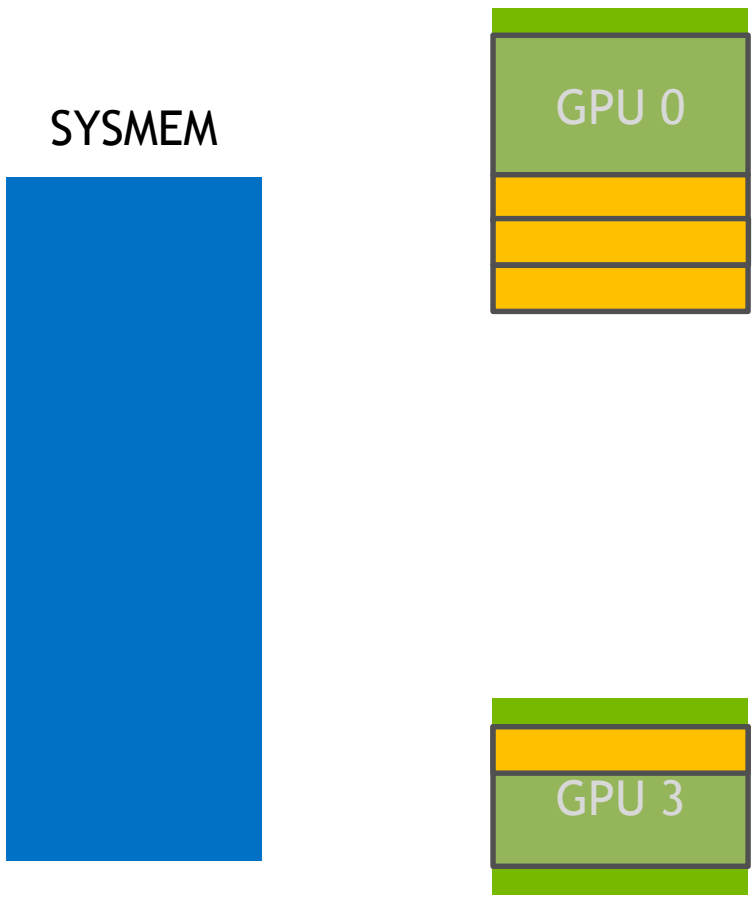


MULTI-GPU WITH UNIFIED MEMORY

With policies:
Remote data can be accessed directly without migrations



MULTI-GPU WITH UNIFIED MEMORY

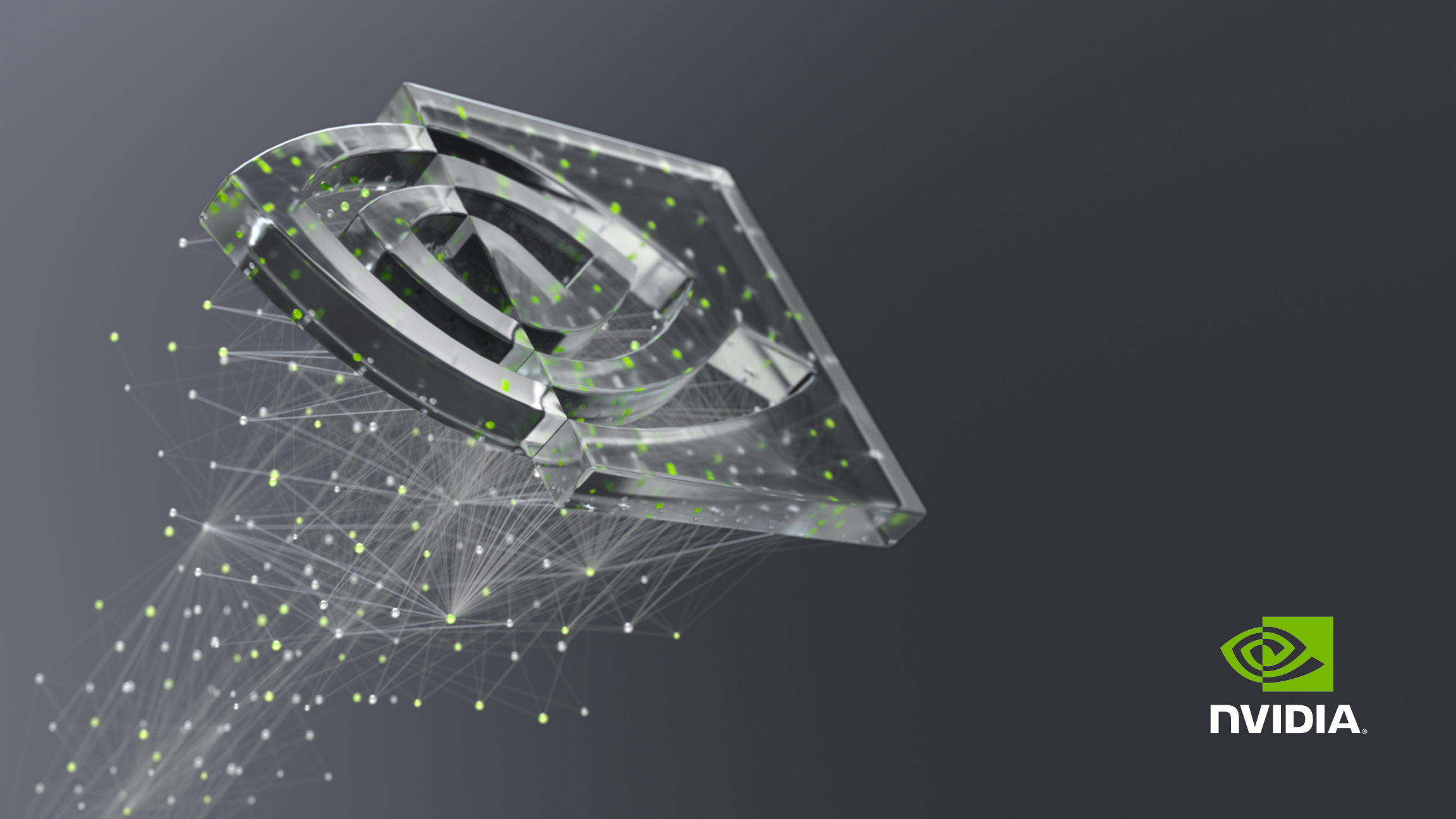


With policies:
Read-only data can be duplicated and accessed locally

PERFORMANCE

Final Words

- UM is first and foremost about ease of programming and programmer productivity
- UM is not primarily a technique to make well-written CUDA codes run faster
- UM cannot do better than expertly written manual data movement, in most cases
- It can be harder to achieve expected concurrency behavior with UM.
- Misuse of UM can slow a code down dramatically
- There are scenarios where UM may enable a design pattern (e.g. graph traversal).
- Oversubscription does not easily/magically give you GPU-type performance on arbitrary datasets/algorithms
- For codes that tend to use many different libraries, each of which makes some demand on GPU memory with no regard for what other libraries are doing, UM can sometimes be a primary way to tackle this challenge (via use of oversubscription), rather than an entire rewrite of the codebase


NVIDIA