EVERYTHING YOU NEED TO KNOW ABOUT UNIFIED MEMORY

Nikolay Sakharnykh, 3/27/2018

NVIDIA

SINGLE POINTER

CPU vs GPU

CPU code

```
void *data;
data = malloc(N);

cpu_func1(data, N);

cpu_func2(data, N);


cpu_func3(data, N);

free(data);
```

GPU code w/ Unified Memory

```
void *data;
data = malloc(N);

cpu_func1(data, N);

gpu_func2<<<...>>>(data, N);
cudaDeviceSynchronize();

cpu_func3(data, N);

free(data);
```

SINGLE POINTER

Explicit vs Unified Memory

Explicit Memory Management

```
void *data, *d_data;
data = malloc(N);
cudaMalloc(&d_data, N);
cpu_func1(data, N);
cudaMemcpy(d_data, data, N, ...)
gpu_func2<<<...>>>(d_data, N);
cudaMemcpy(data, d_data, N, ...)
cudaFree(d_data);
cpu_func3(data, N);

free(data);
```

GPU code w/ Unified Memory

```
void *data;
data = malloc(N);

cpu_func1(data, N);

gpu_func2<<<...>>>(data, N);
cudaDeviceSynchronize();

cpu_func3(data, N);

free(data);
```

SINGLE POINTER

Full Control with Prefetching

Explicit Memory Management

```
void *data, *d_data;
data = malloc(N);
cudaMalloc(&d_data, N);
cpu_func1(data, N);
cudaMemcpy(d_data, data, N, ...)
gpu_func2<<<...>>>(d_data, N);
cudaMemcpy(data, d_data, N, ...)
cudaFree(d_data);
cpu_func3(data, N);

free(data);
```

Unified Memory + Prefetching

```
void *data;
data = malloc(N);

cpu_func1(data, N);
cudaMemPrefetchAsync(data, N, GPU)
gpu_func2<<<...>>>(data, N);
cudaMemPrefetchAsync(data, N, CPU)
cudaDeviceSynchronize();
cpu_func3(data, N);

free(data);
```

SINGLE POINTER

Deep Copy

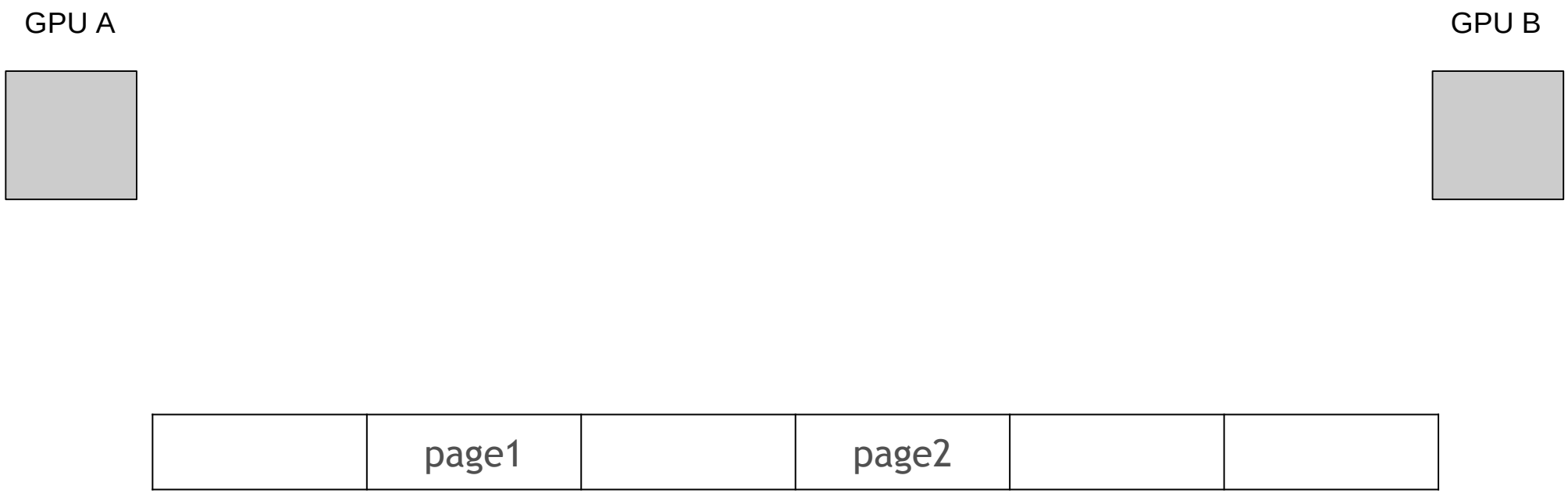
Explicit Memory Management

```
char **data;
// allocate and initialize data on the CPU

char **d_data;
char **h_data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++) {
  cudaMalloc(&h_data[i], N);
  cudaMemcpy(h_data[i], data[i], N, ...);
}
cudaMalloc(&d_data, N*sizeof(char*));
cudaMemcpy(d_data, h_data, N*sizeof(char*), ...);

gpu_func<<<...>>>(d_data, N);
```

GPU code w/ Unified Memory

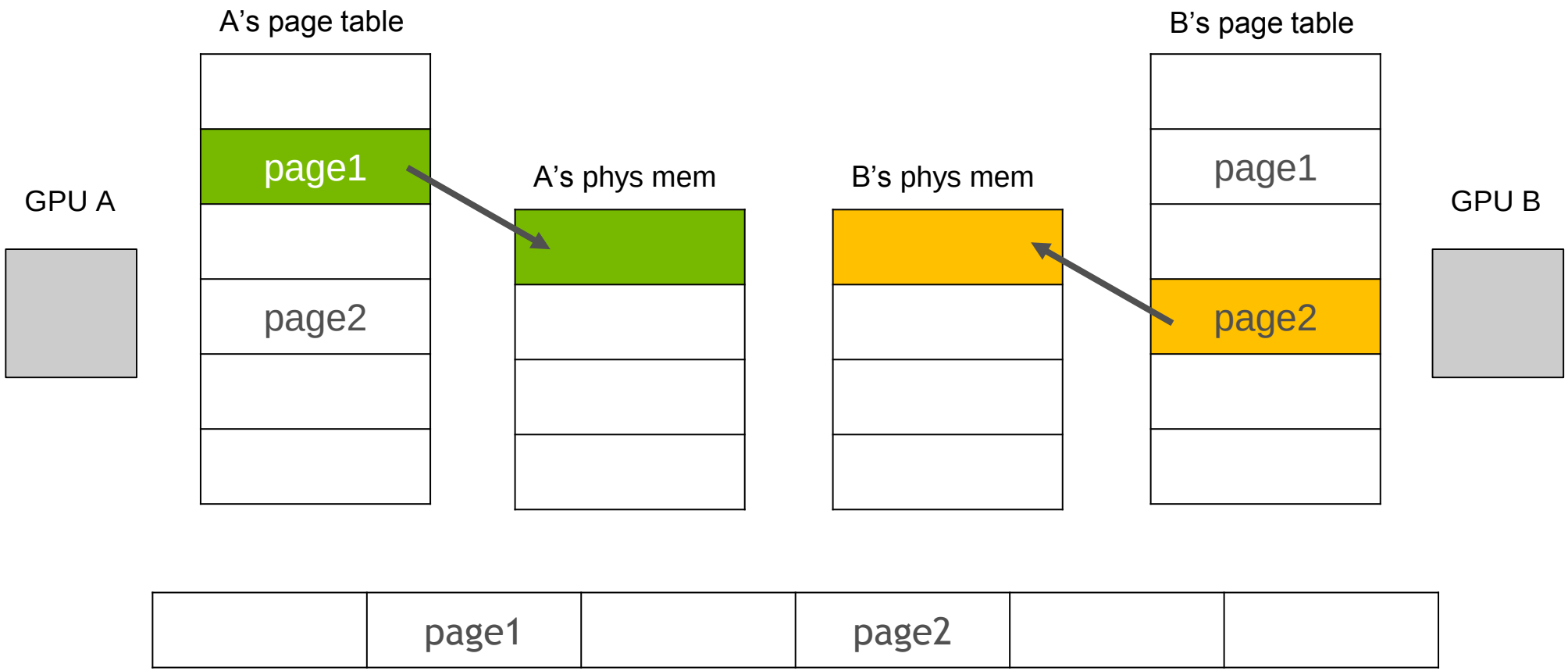
```
char **data;
// allocate and initialize data on the CPU

gpu_func<<<...>>>(data, N);
```

UNIFIED MEMORY BASICS



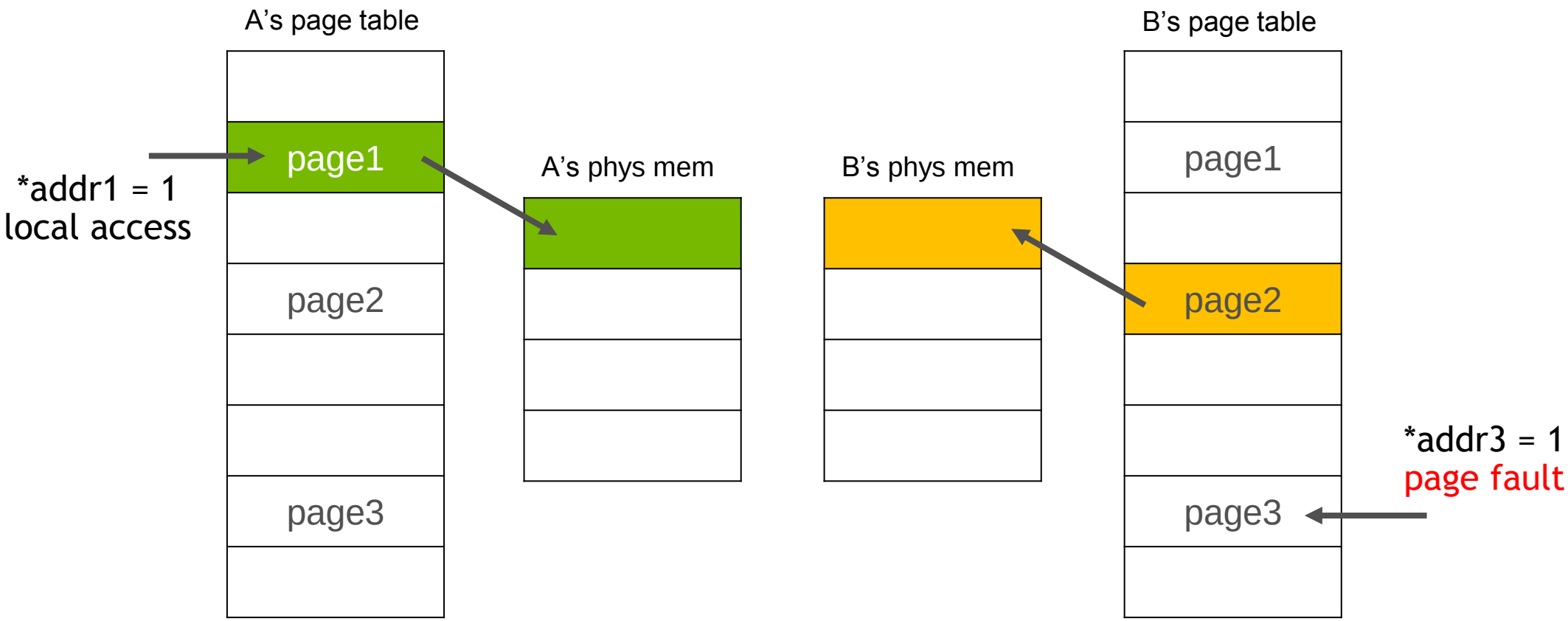
Single virtual memory shared between processors

UNIFIED MEMORY BASICS

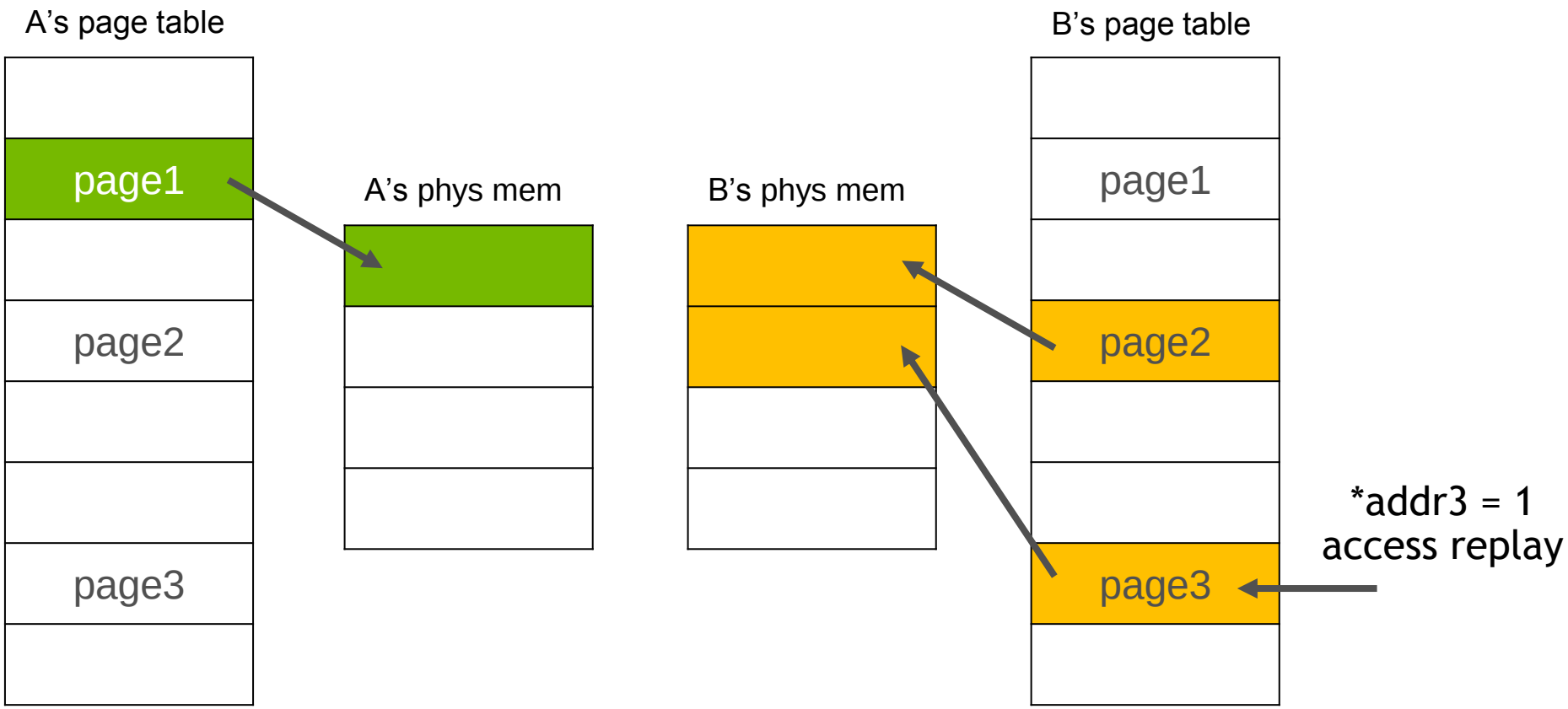


Single virtual memory shared between processors

UNIFIED MEMORY BASICS

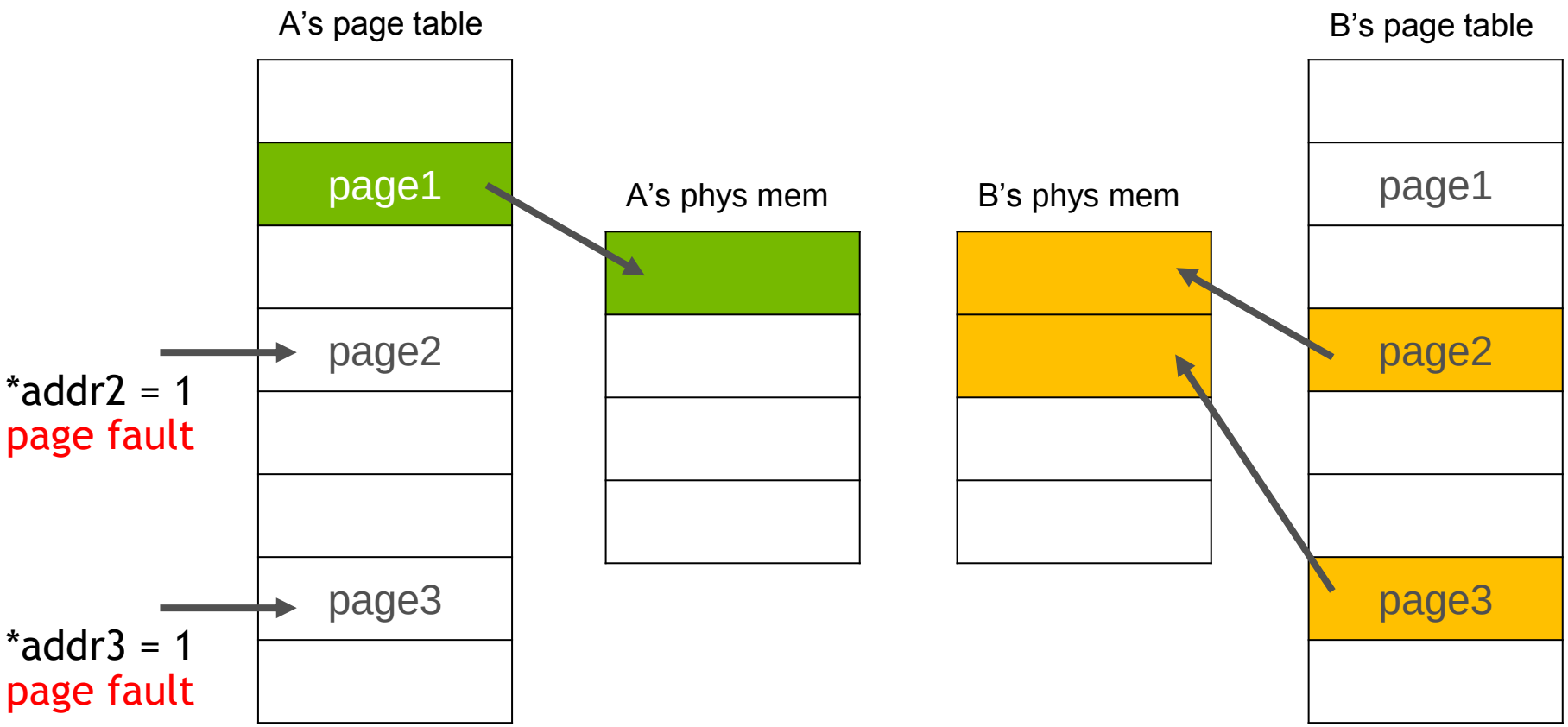


UNIFIED MEMORY BASICS

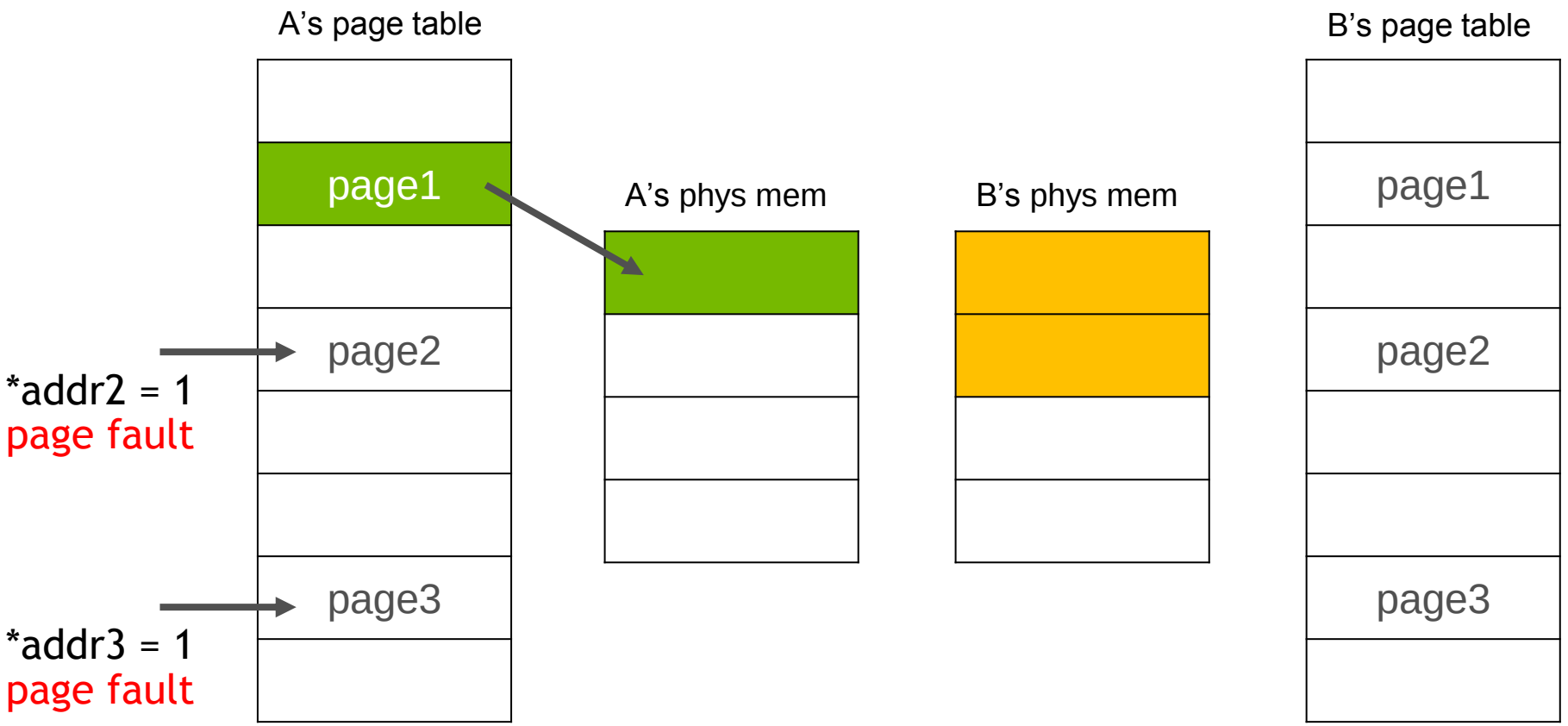


page3 populated and mapped into B's memory

UNIFIED MEMORY BASICS

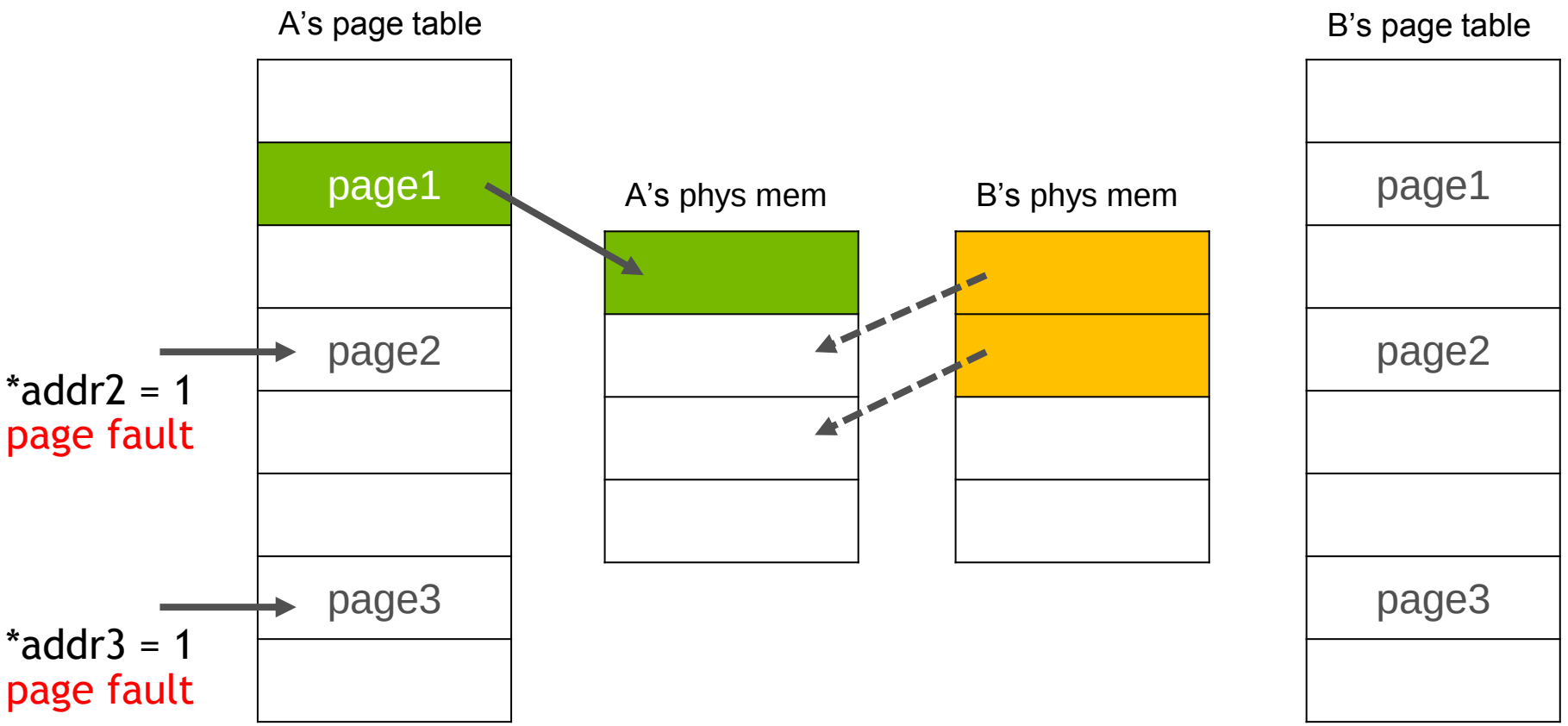


UNIFIED MEMORY BASICS



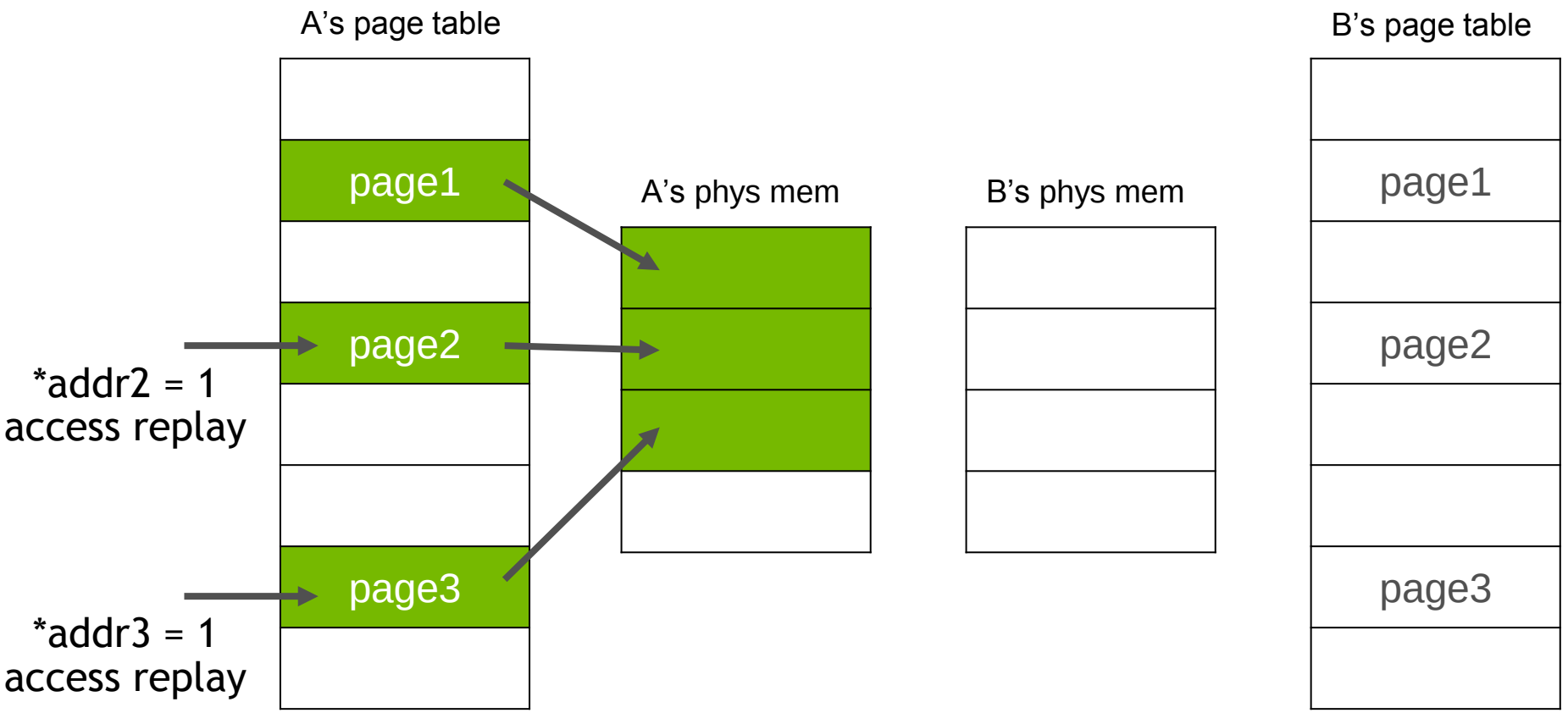
page2 and page3 unmapped from B's memory

UNIFIED MEMORY BASICS

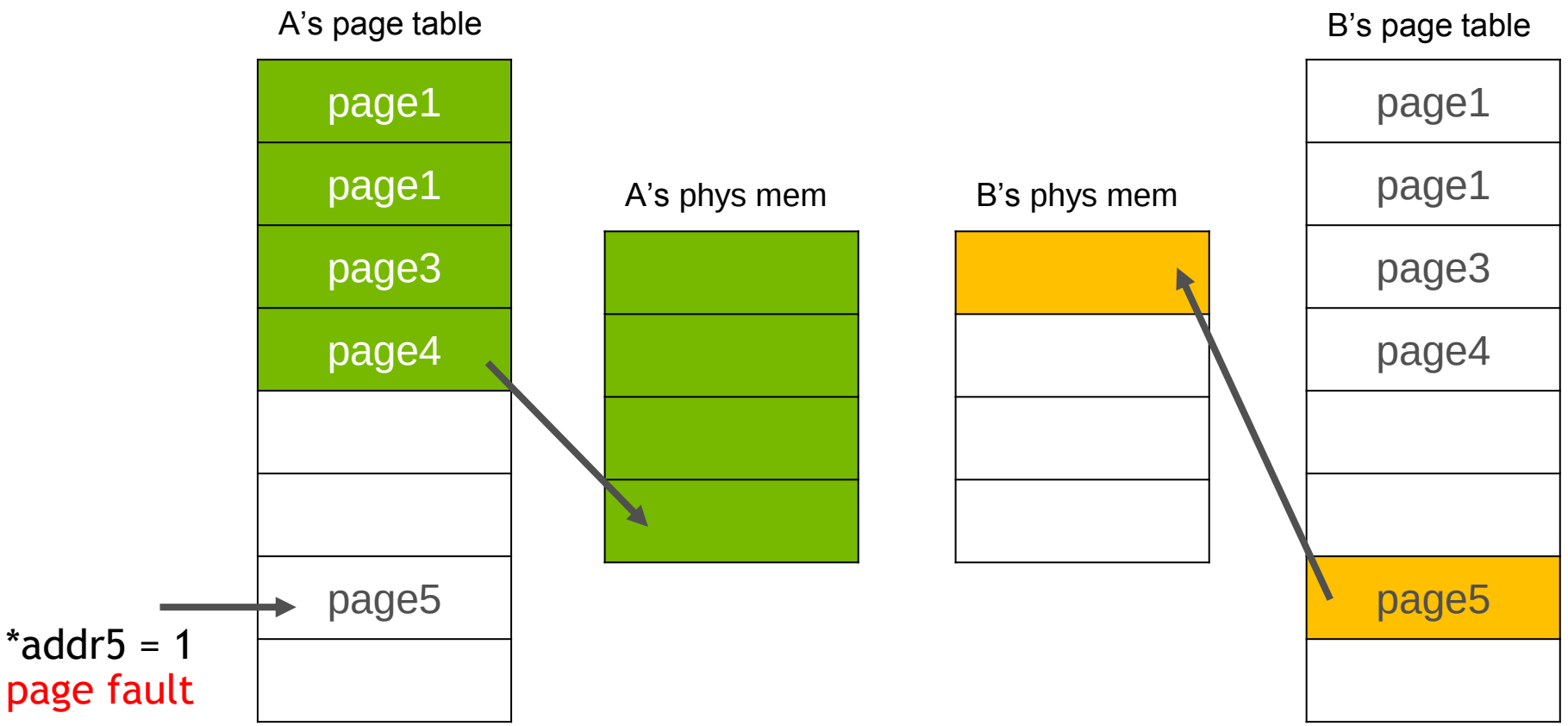


pages data migrated to A's physical memory

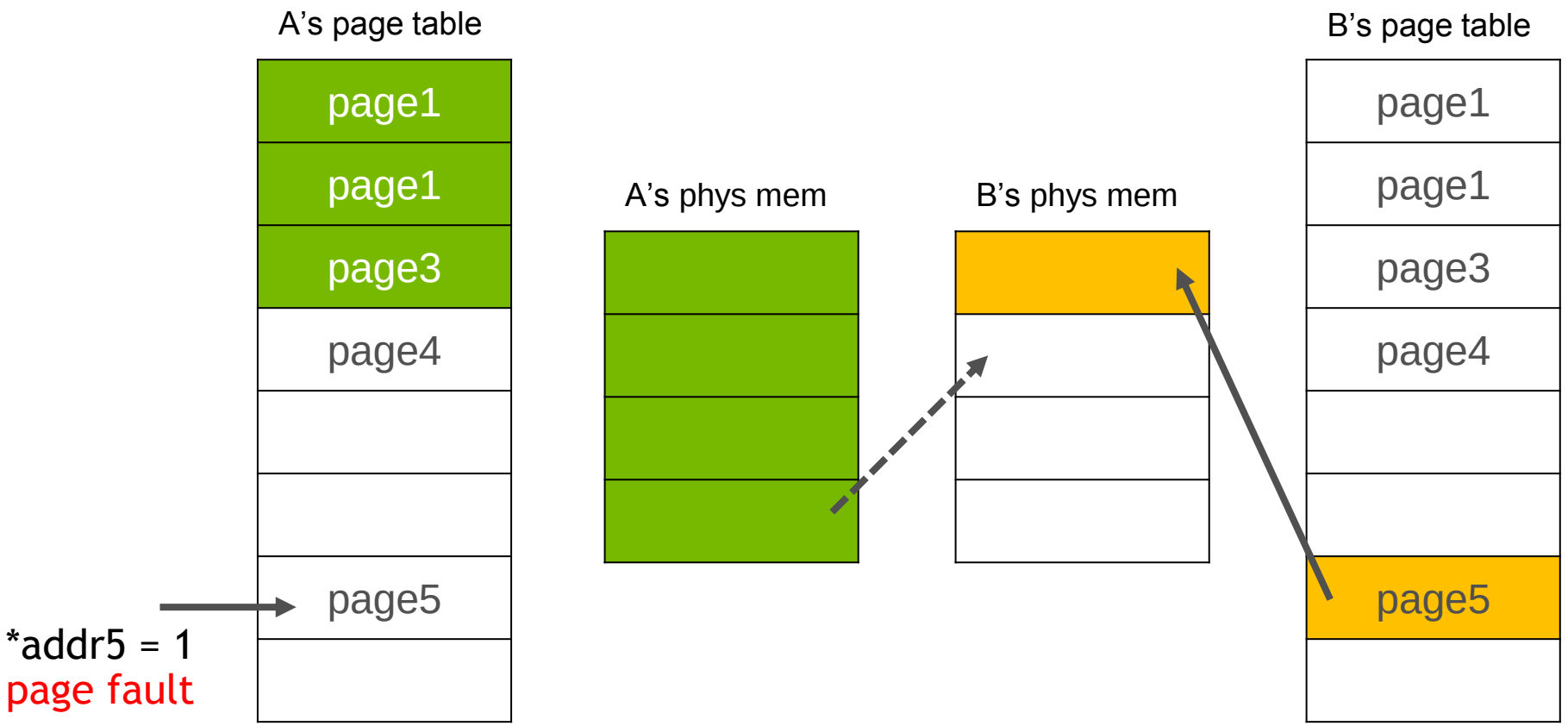
UNIFIED MEMORY BASICS



MEMORY OVERSUBSCRIPTION

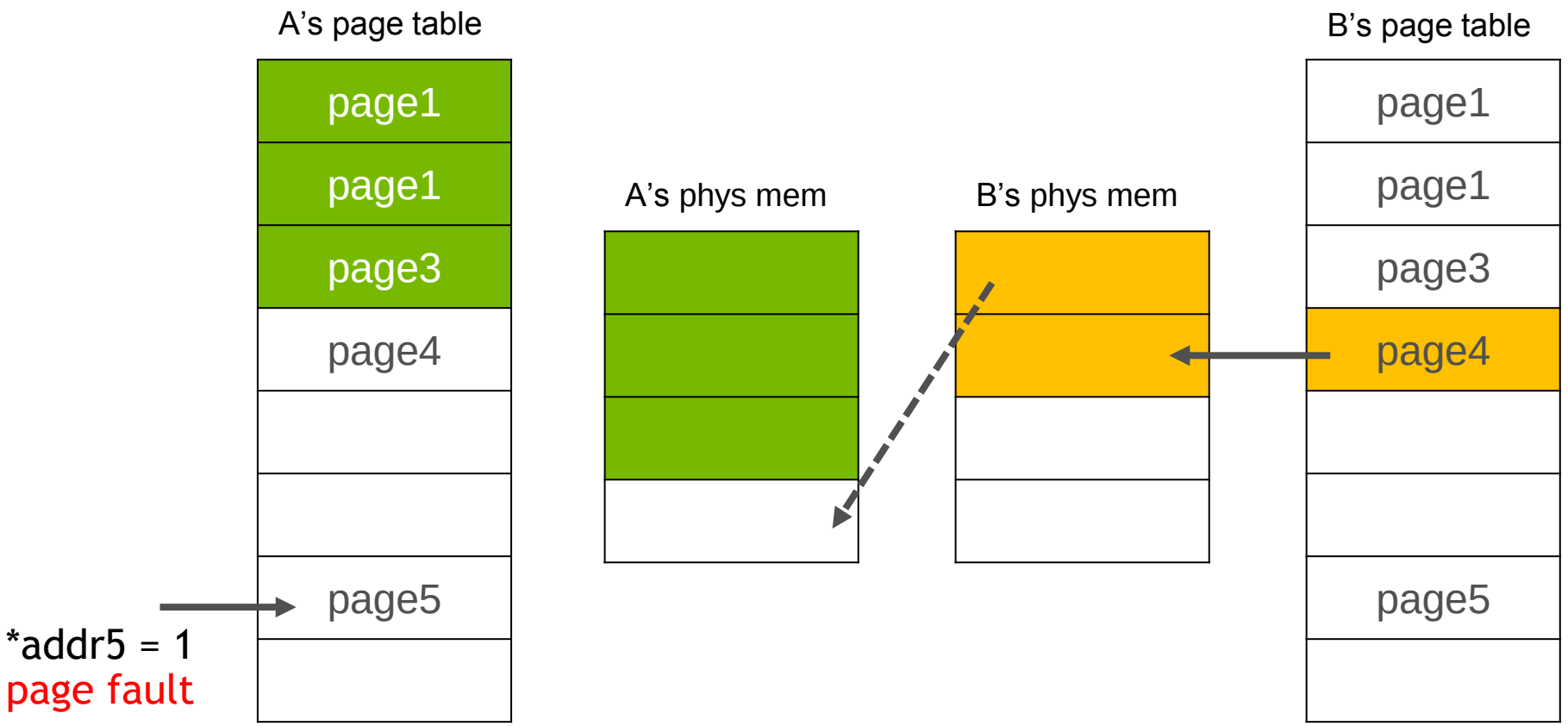


MEMORY OVERSUBSCRIPTION



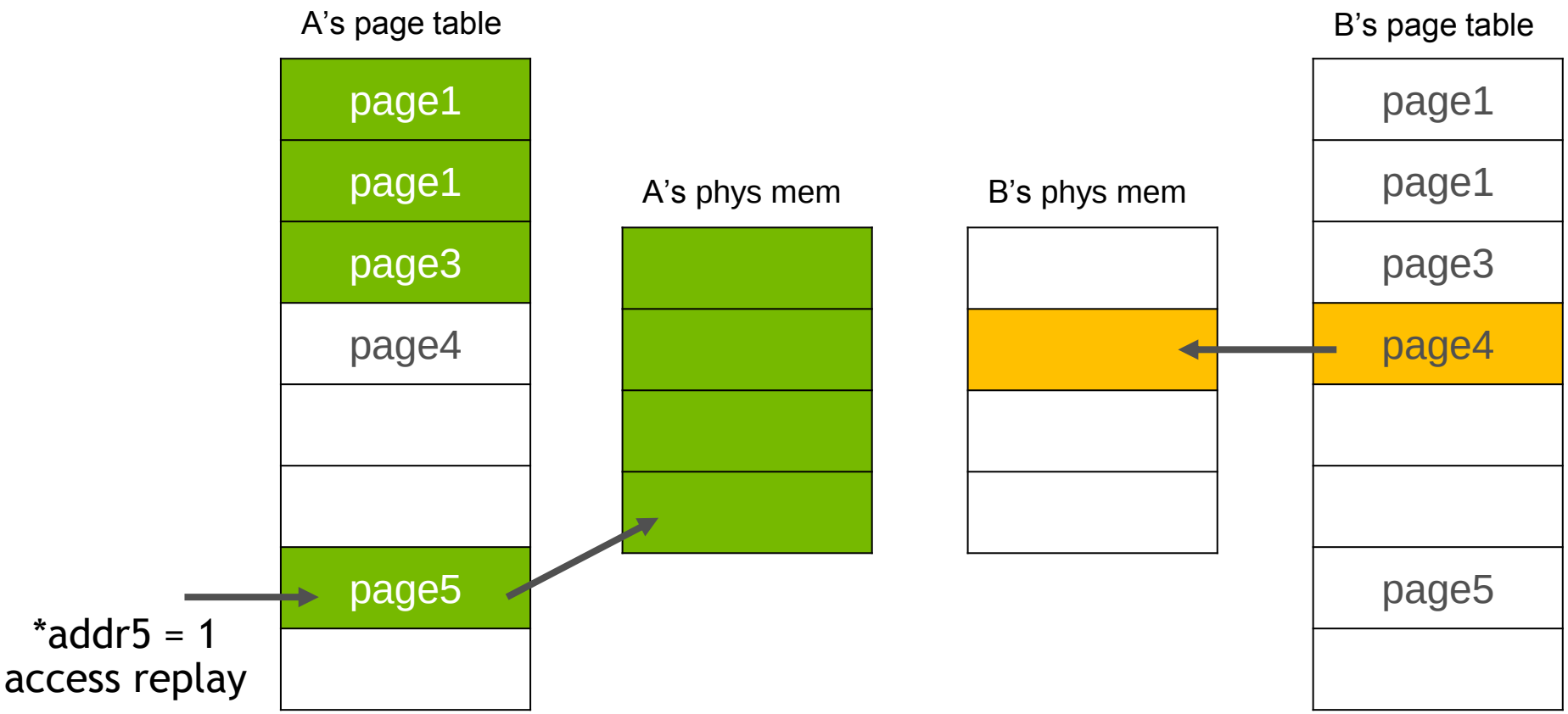
page4 unmapped from A's memory and migrated

MEMORY OVERSUBSCRIPTION



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

MEMORY OVERSUBSCRIPTION



SIMPLIFYING DL FRAMEWORK DESIGN

Eliminated **3,000 lines** of repetitive and error-prone code in Caffe

Developers can add new inherited Layer classes in a much simpler manner

The final call to a CPU function or a GPU kernel (caffe_gpu_gemm) still need to be explicit

Existing Design

```
class ConvolutionLayer
{
public:
   void cpu_data()
   void cpu_diff()
   void gpu_data()
   void gpu_diff()

   void mutable_cpu_data()
   void mutable_cpu_diff()
   void mutable_gpu_data()
   void mutable_gpu_diff()

   void Forward_cpu()
   void Forward_gpu()
   void forward_cpu_gemm()
   void forward_gpu_gemm()
   void forward_cpu_bias()
   void forward_gpu_bias()

   void Backward_cpu()
   void Backward_gpu()
   void backward_cpu_gemm()
   void backward_gpu_gemm()
   void backward_cpu_bias()
   void backward_gpu_bias()
}
```

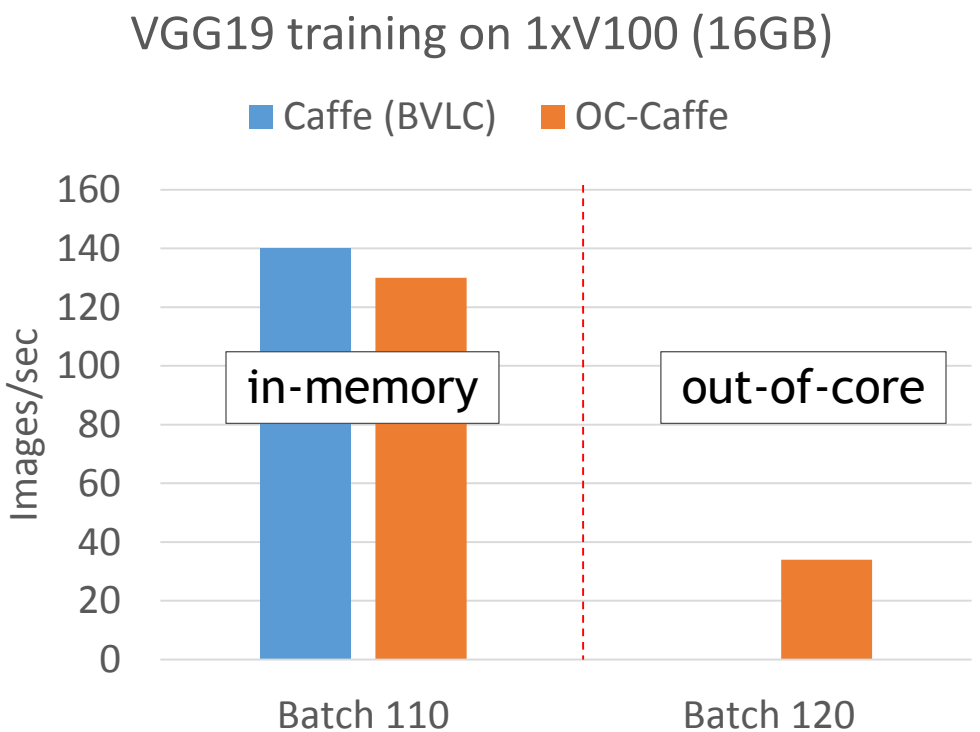
Unified Memory Design

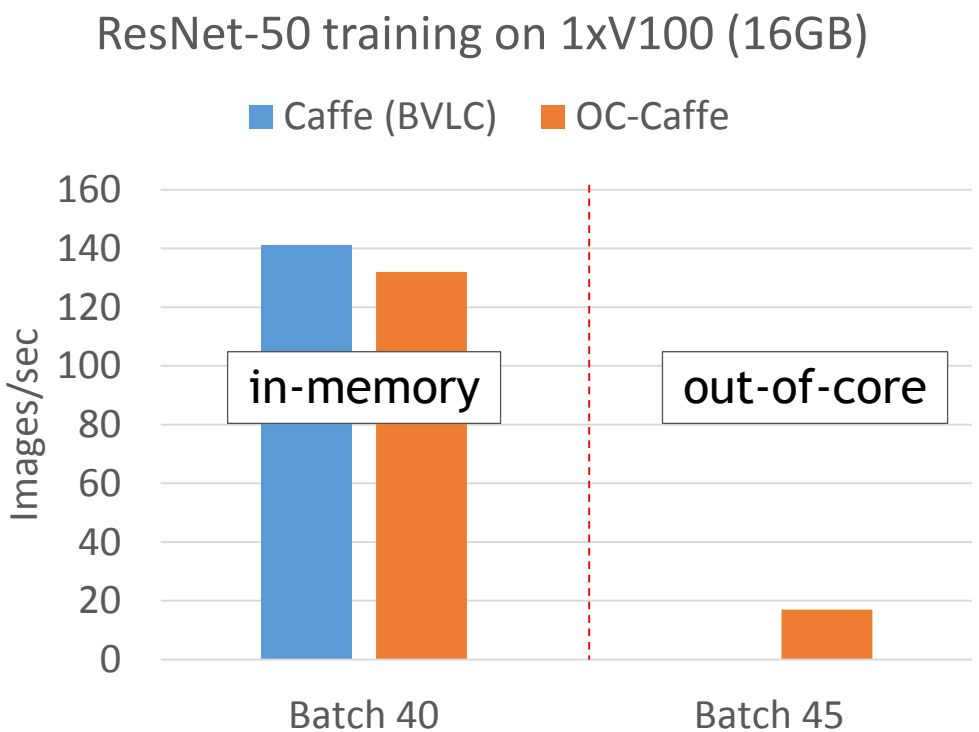
```
class ConvolutionLayer
{
public:
   void data()
   void diff()

   void mutable_data()
   void mutable_diff()

   void Forward()
   void forward_gemm()
   void forward_bias()

   void Backward()
   void backward_gemm()
   void backward_bias()
}
```

A. A. Awan, C-H Chu, H. Subramoni, X. Lu, D.K. Panda, "*OC-DNN: Designing Out-of-Core Deep Neural Network Training by Exploiting Unified Memory on Pascal and Volta GPUs*", <double-blind submission under review>

CAN THIS DESIGN OFFER GOOD PERF?

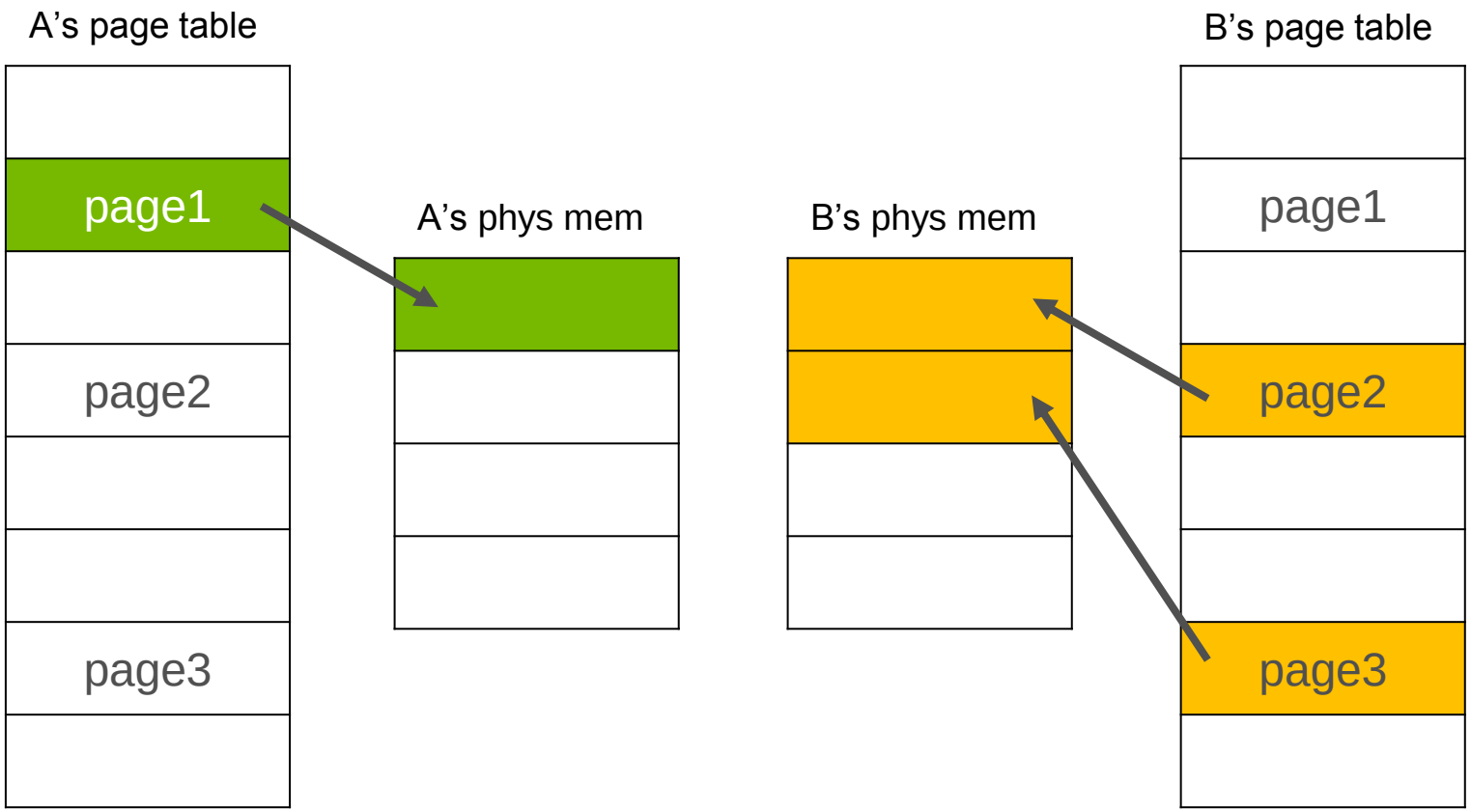
DL training with Unified Memory





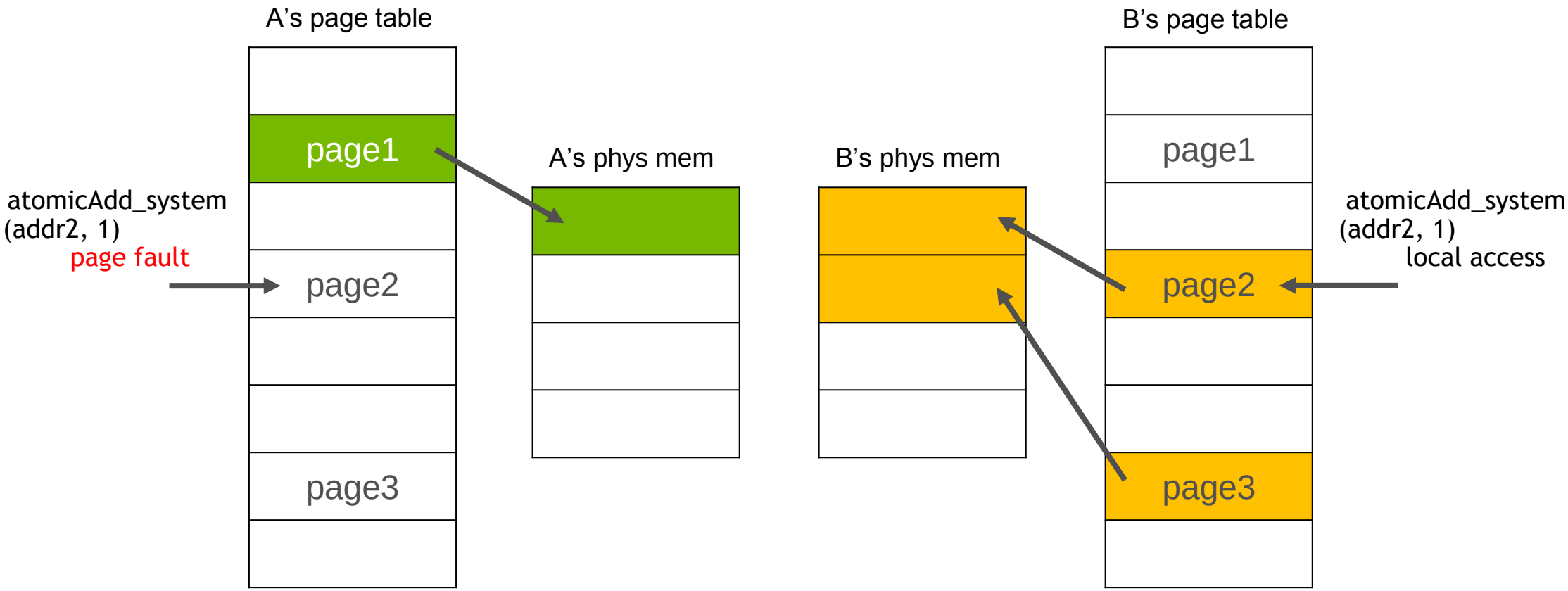
OC-Caffe will be released by the HiDL Team@OSU: hidl.cse.ohio-state.edu, mvapich.cse.ohio-state.edu

CONCURRENT ACCESS



CONCURRENT ACCESS

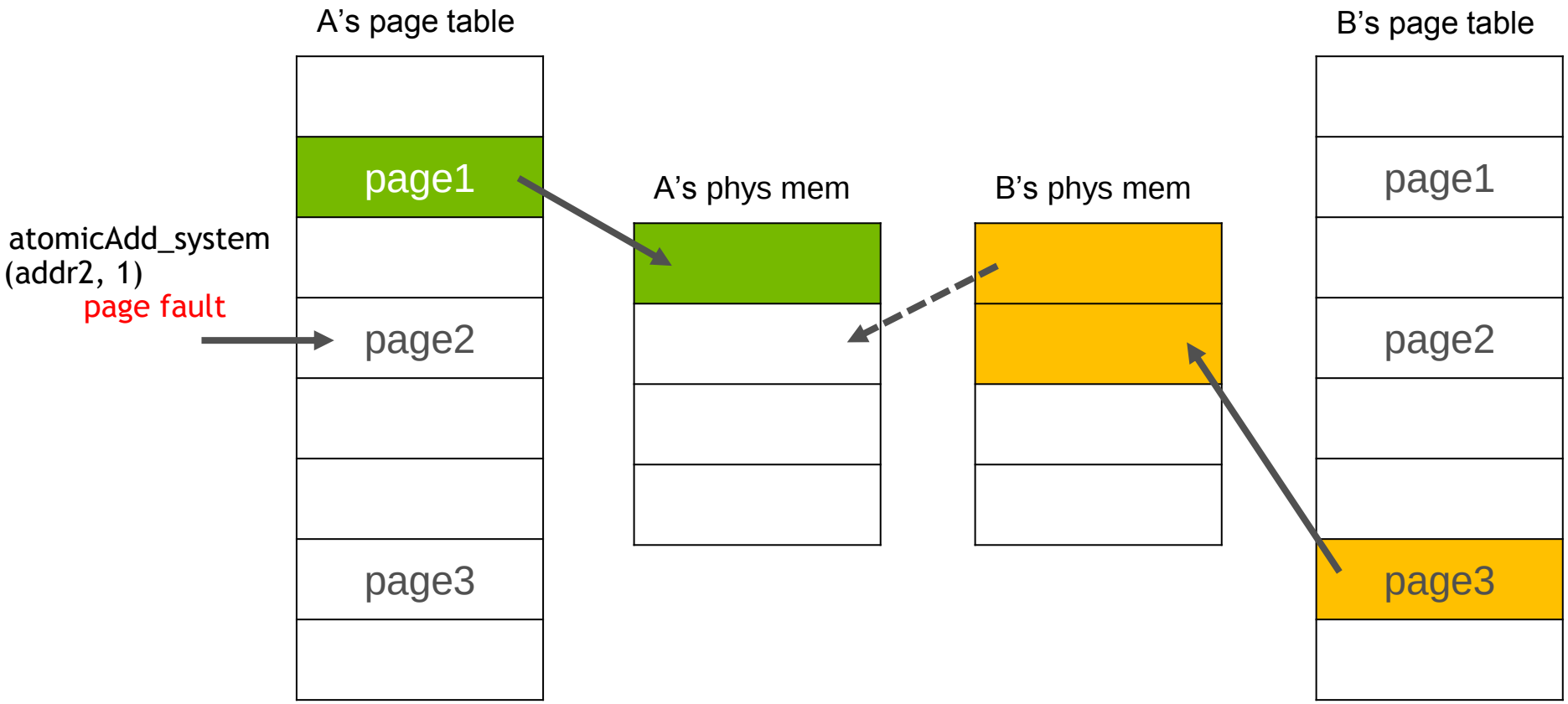
Exclusive Access*



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

CONCURRENT ACCESS

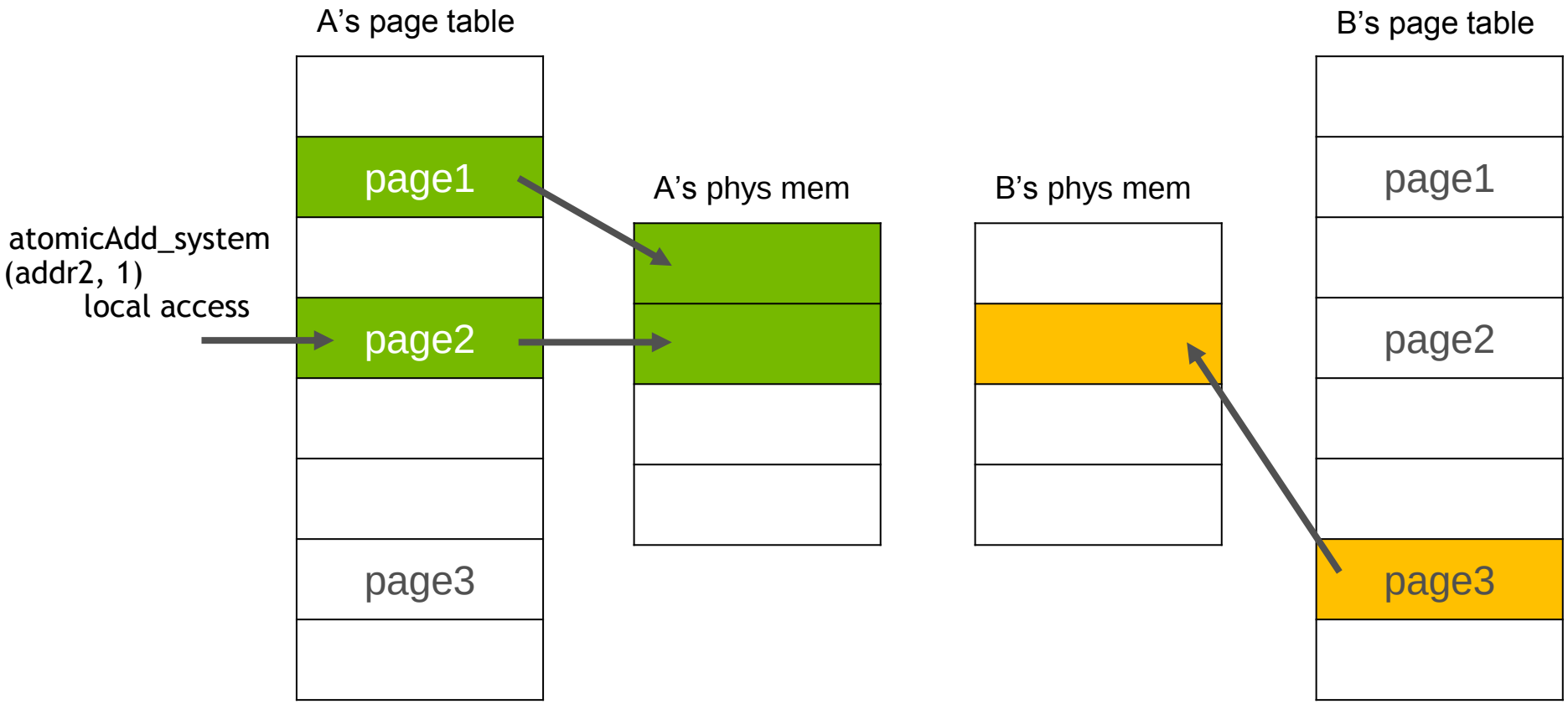
Exclusive Access



page2 unmapped in B's memory and migrated to A

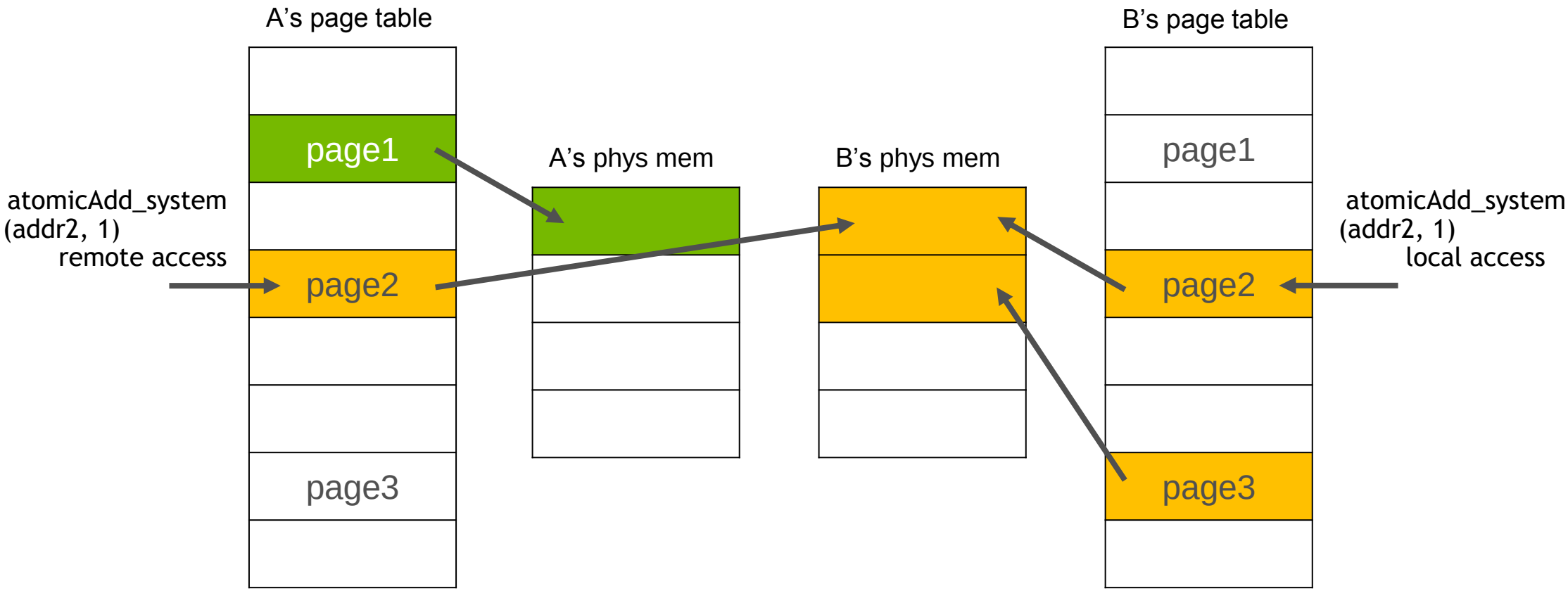
CONCURRENT ACCESS

Exclusive Access



CONCURRENT ACCESS

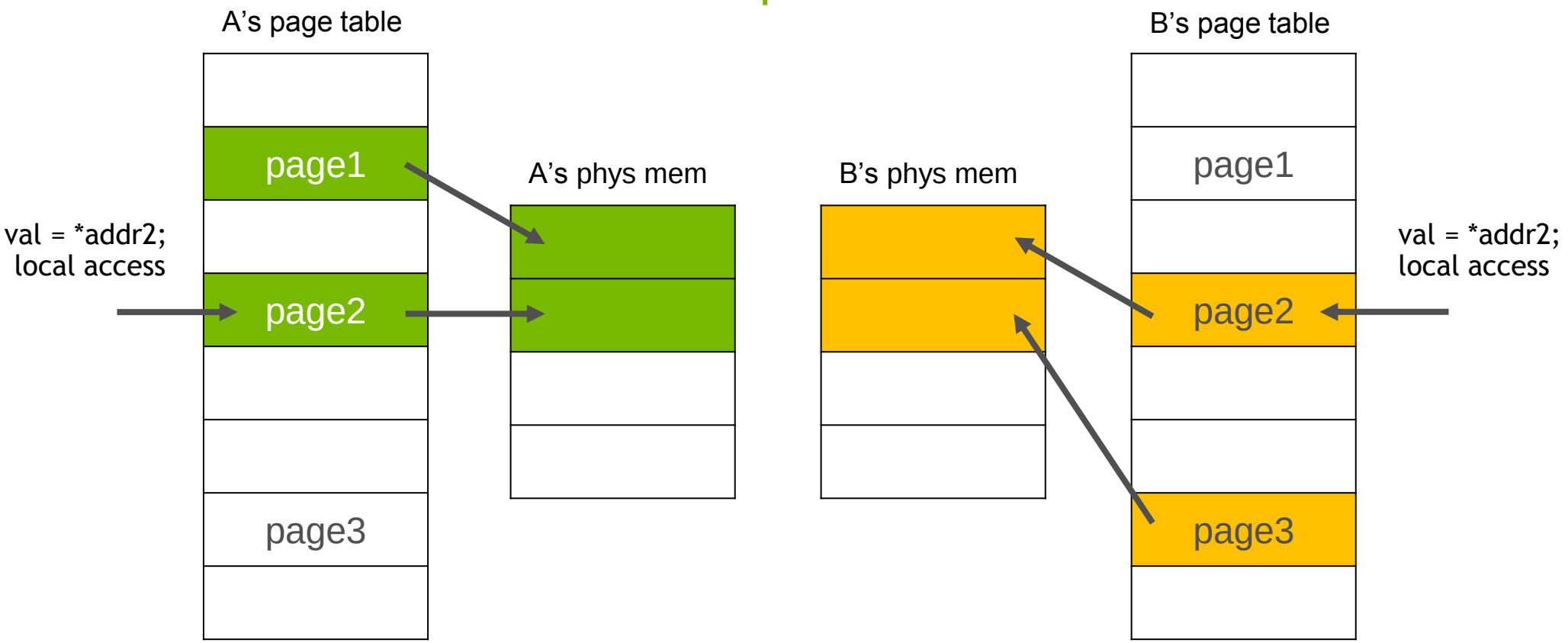
Atomics over NVLINK*



*both processors need to support atomic operations

CONCURRENT ACCESS

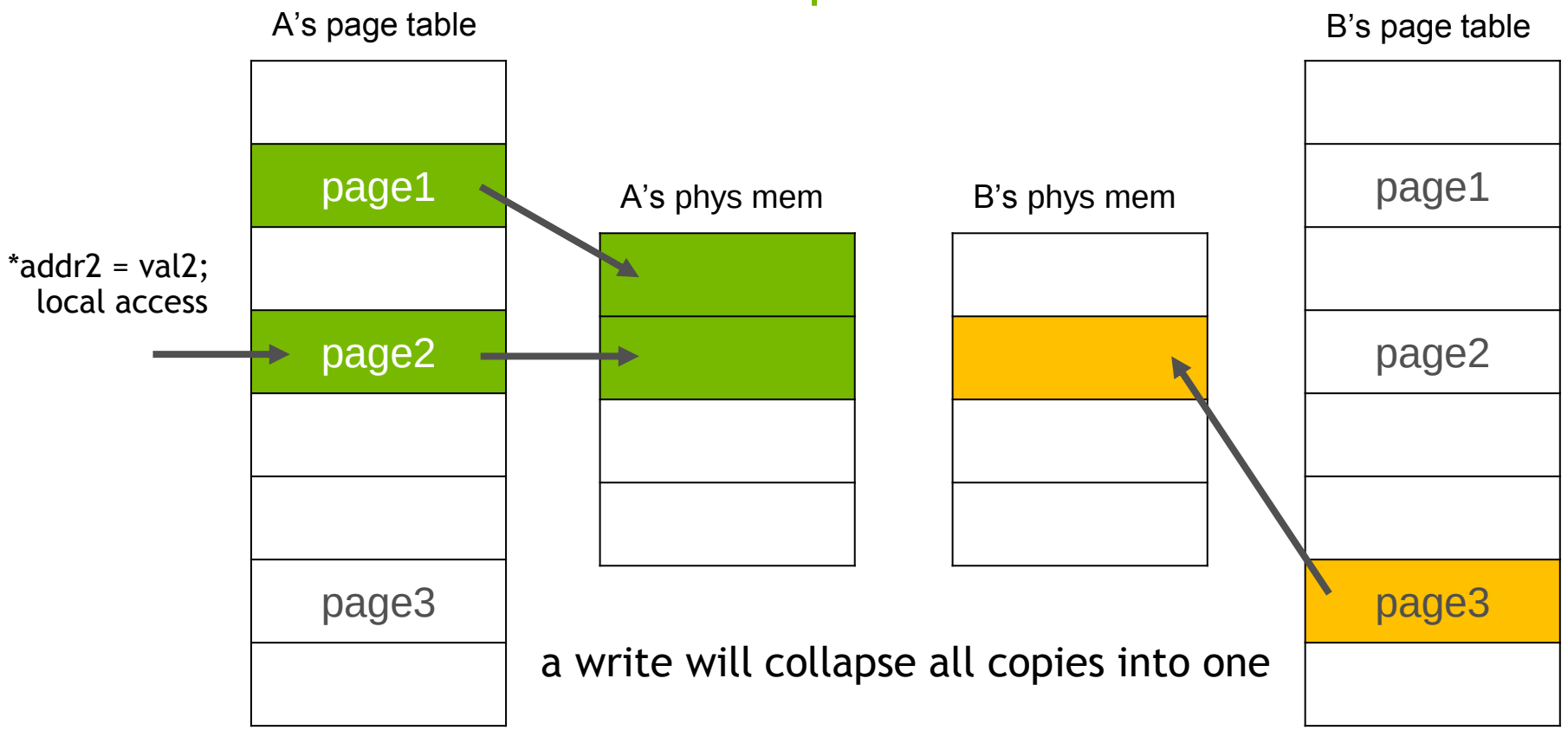
Read duplication*



*each processor must maintain its own page table

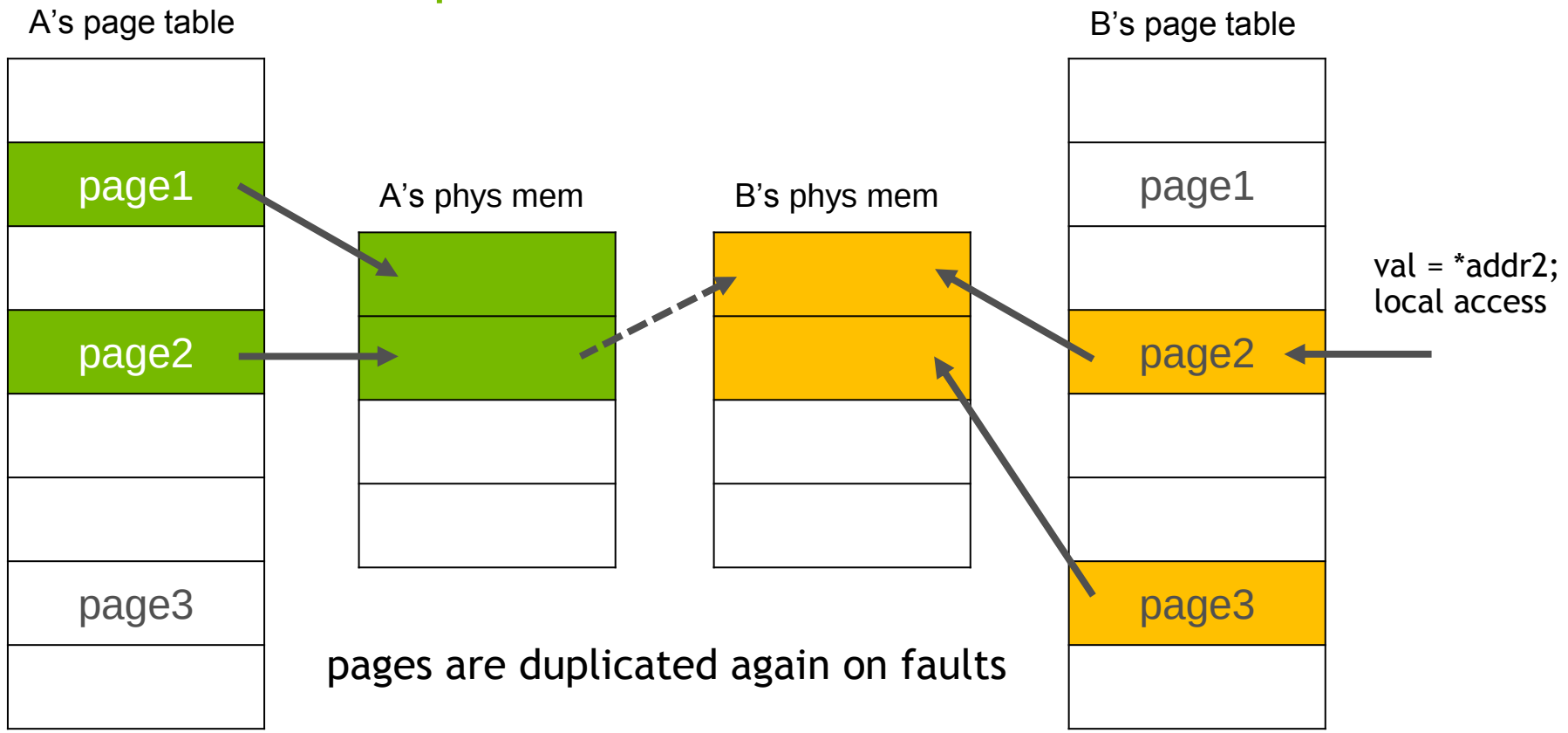
CONCURRENT ACCESS

Read duplication: write



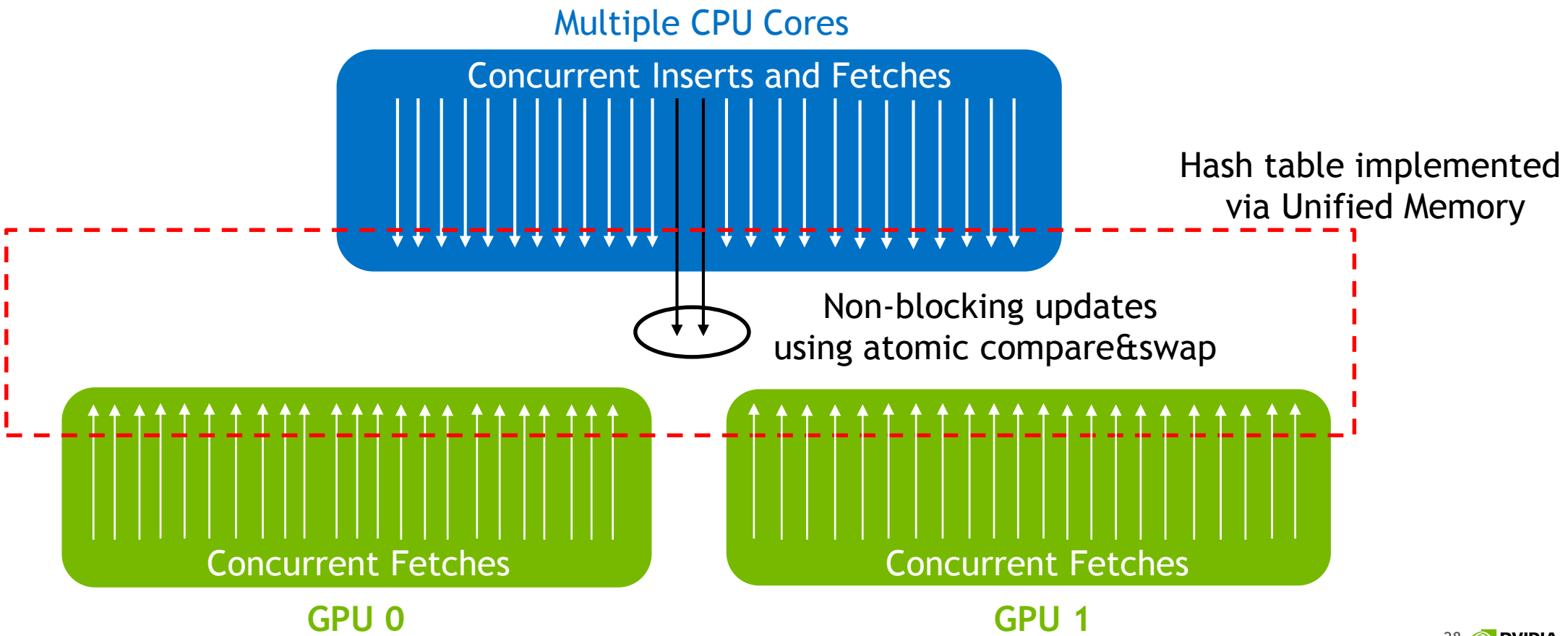
CONCURRENT ACCESS

Read duplication: read after write



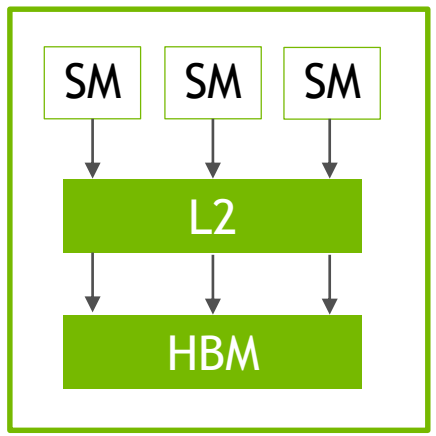
ANALYTICS USE CASE

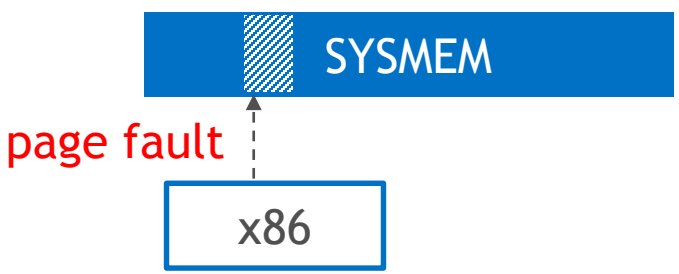
Design of a Concurrent Hybrid Hash Table



ANALYTICS USE CASE

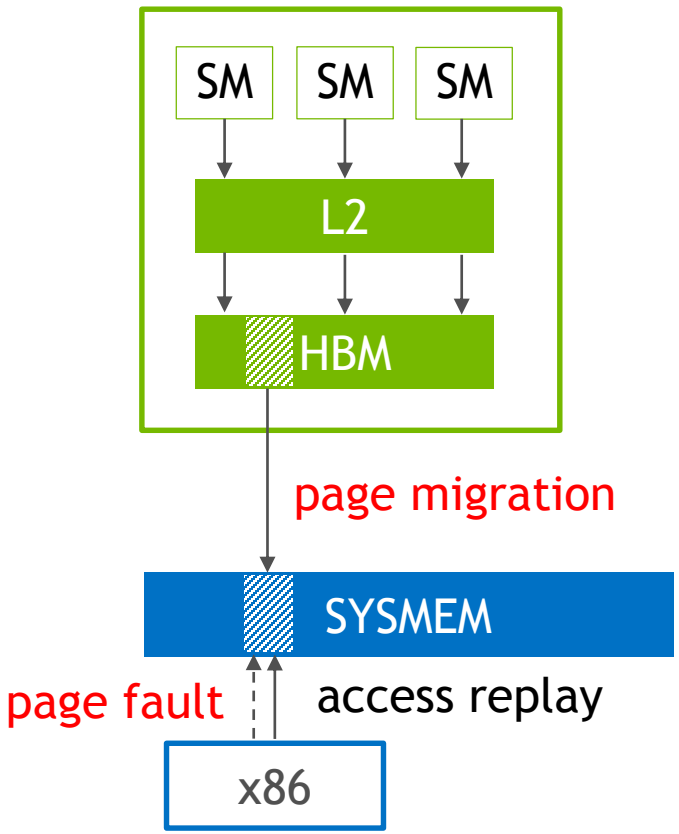
Concurrent Access To Hash Table





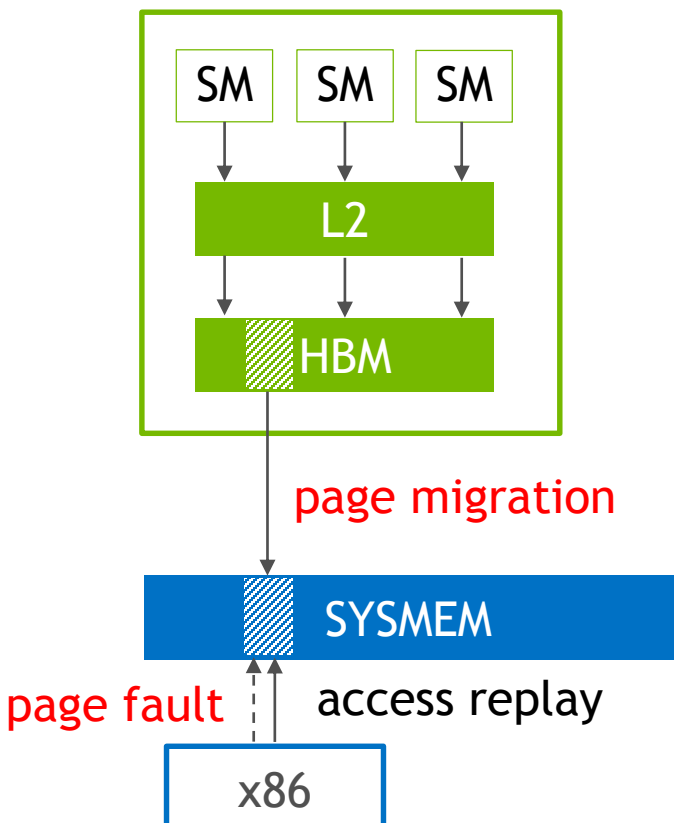
ANALYTICS USE CASE

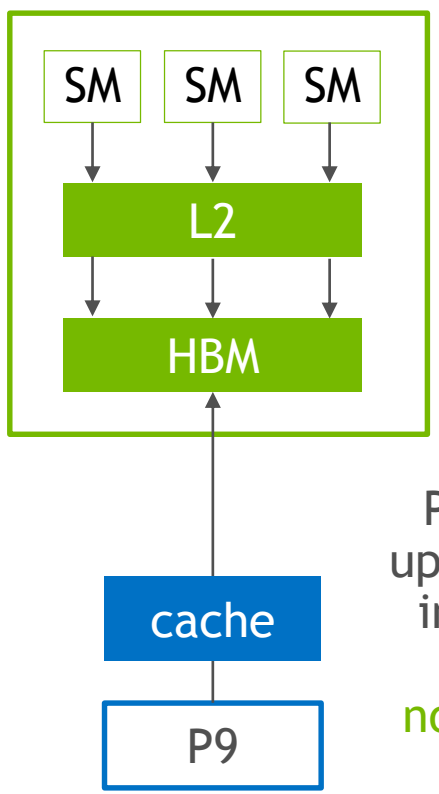
Concurrent Access To Hash Table



ANALYTICS USE CASE

Concurrent Access To Hash Table

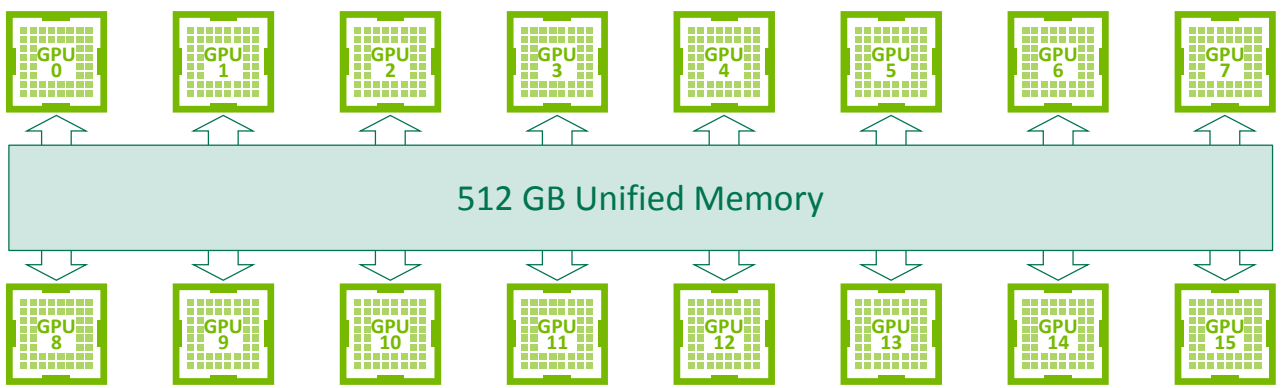




P9 can directly update hash entry in GPU memory

no page faults or migrations!

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);
```

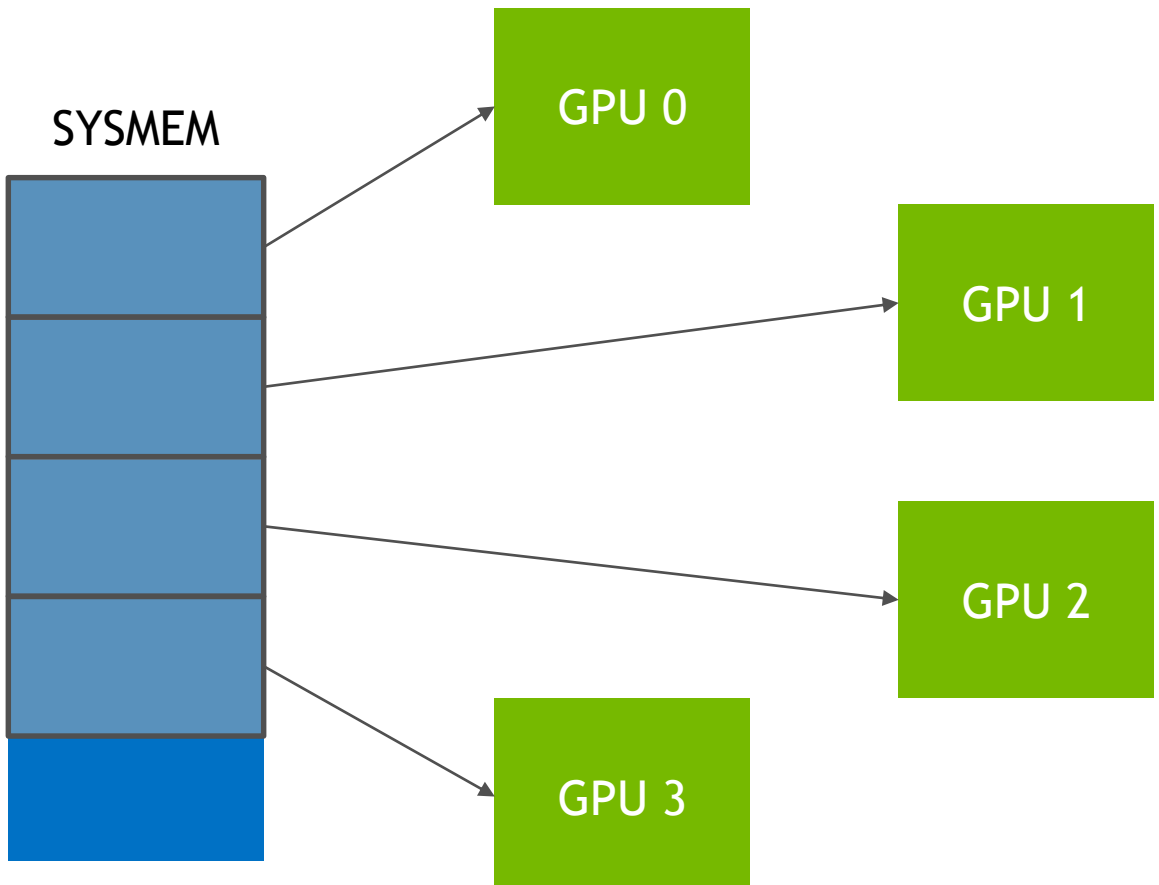
Multi-GPU

```
__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

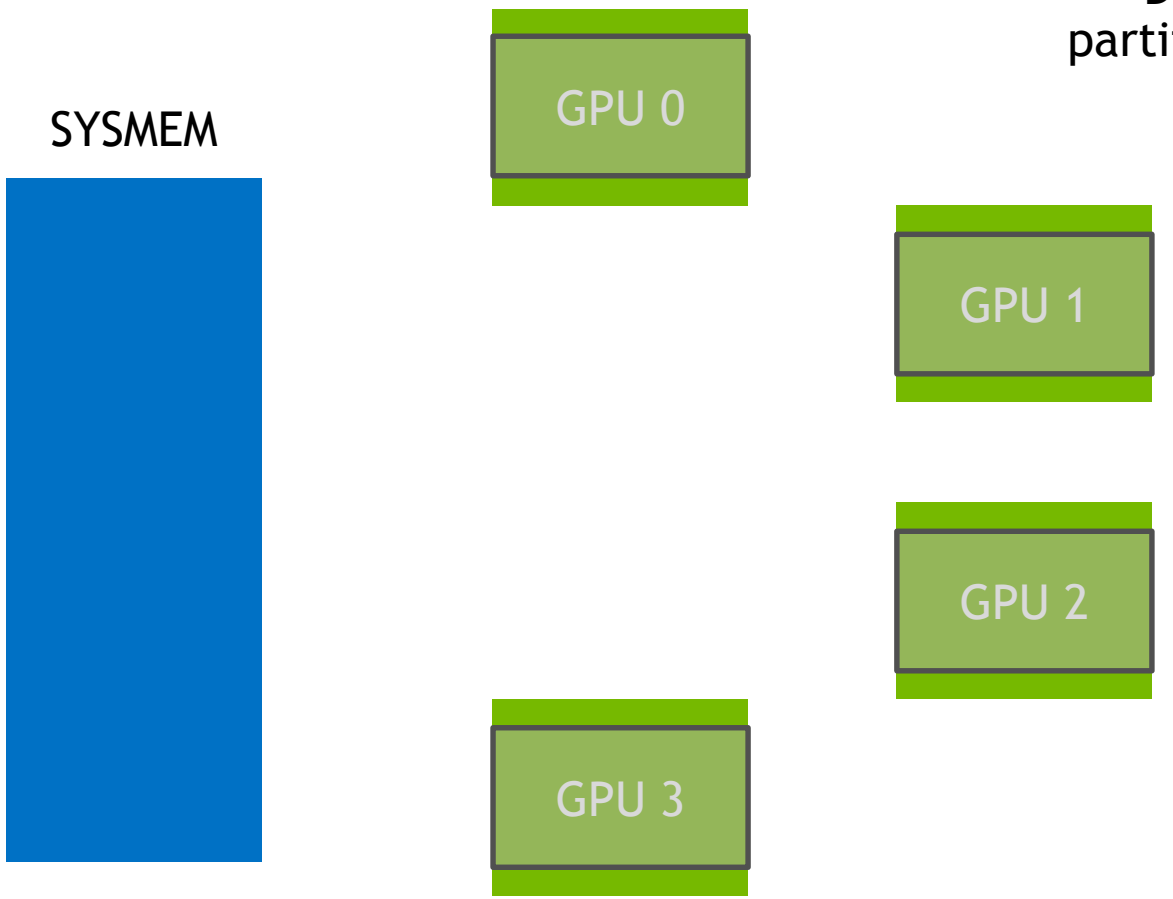
MULTI-GPU WITH UNIFIED MEMORY



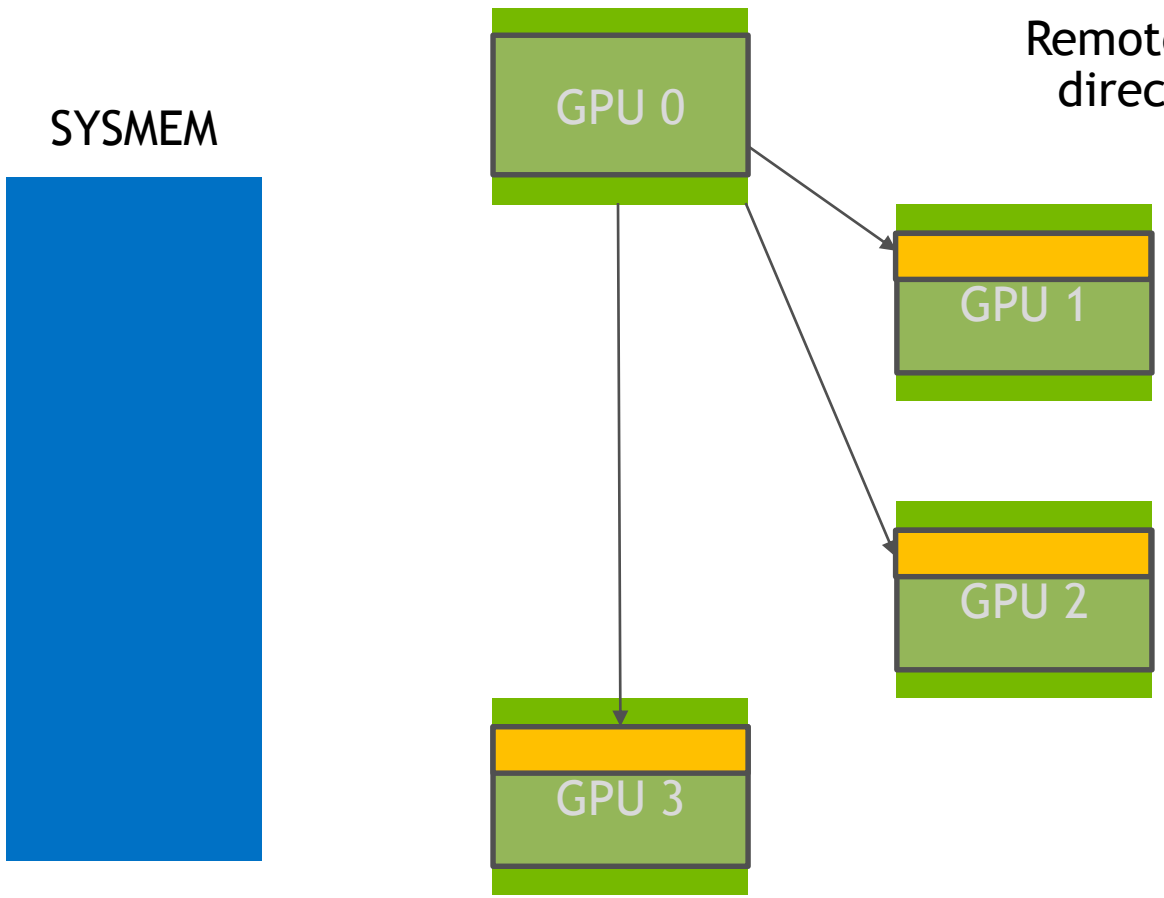
GPU kernels
initiate migrations

MULTI-GPU WITH UNIFIED MEMORY

Data *automatically* partitioned between GPUs on first-touch



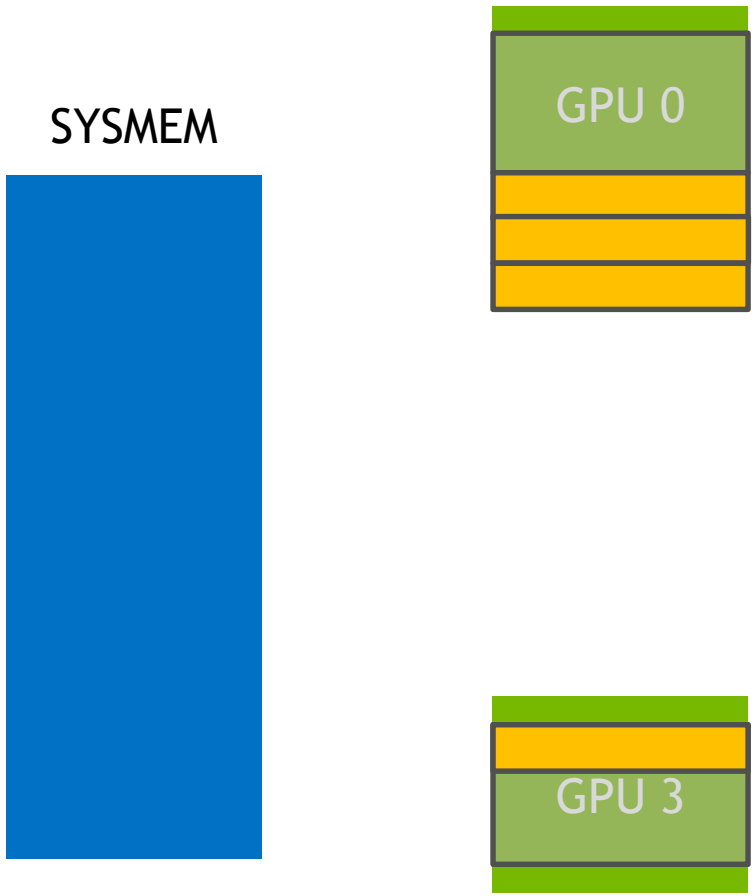
MULTI-GPU WITH UNIFIED MEMORY



With policies:
Remote data can be accessed directly without migrations

MULTI-GPU WITH UNIFIED MEMORY

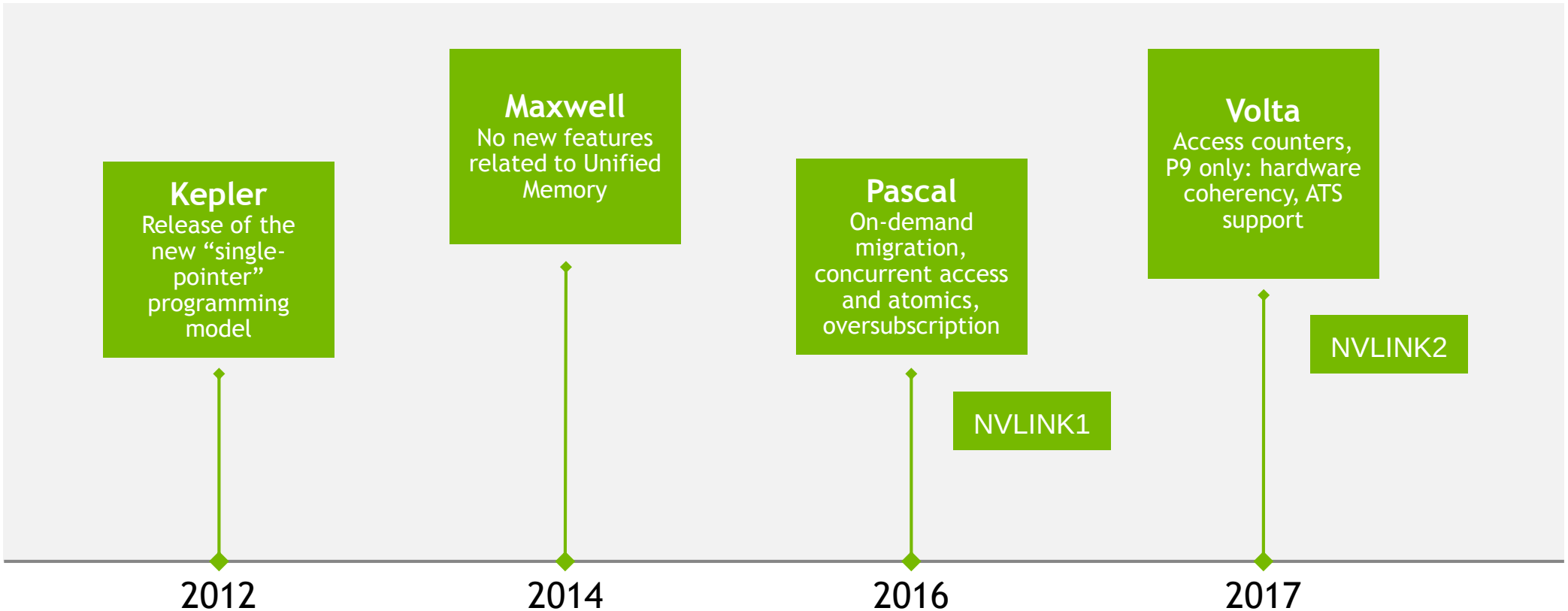
With policies:
Read-only data can be duplicated and accessed locally



GPU ARCHITECTURE AND SOFTWARE EVOLUTION

UNIFIED MEMORY

Evolution of GPU architectures



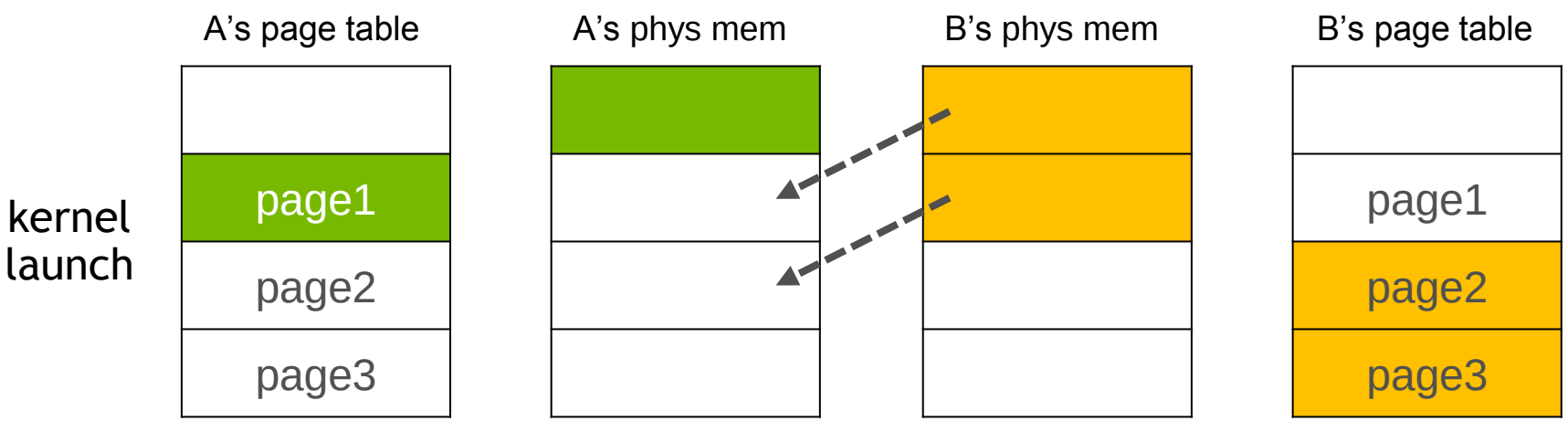
**Not all features are available on all platforms*

UNIFIED MEMORY: BEFORE PASCAL

Available since CUDA 6

No GPU page fault support: move **all dirty pages** on kernel launch

No concurrent access, no GPU memory oversubscription, no system-wide atomics

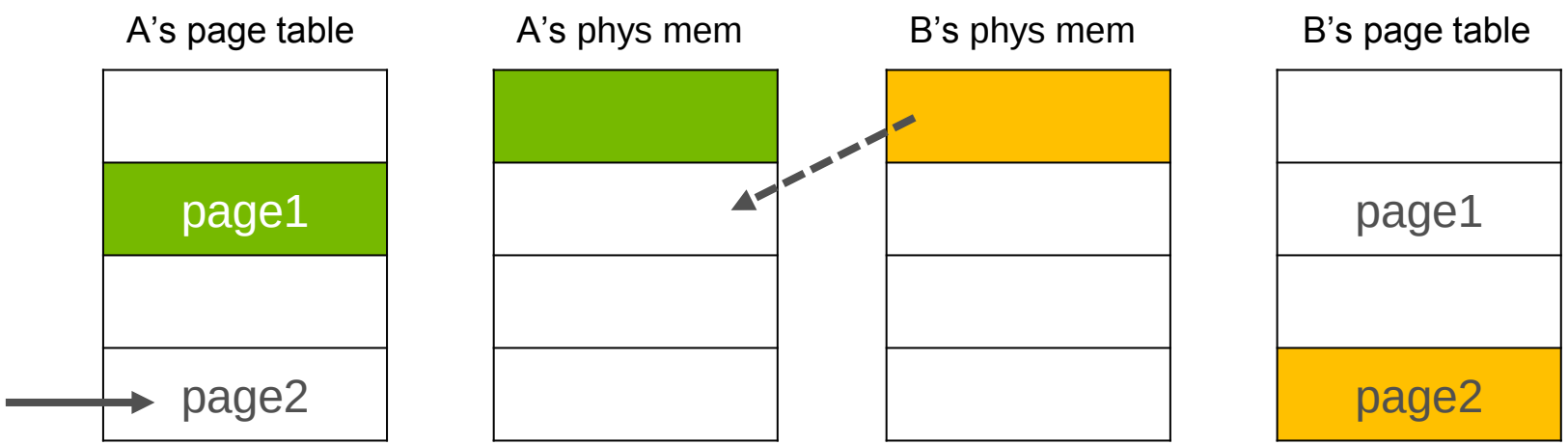


UNIFIED MEMORY: PASCAL AND VOLTA

Available since CUDA 8

GPU page fault support, concurrent access, extended VA space (48-bit)

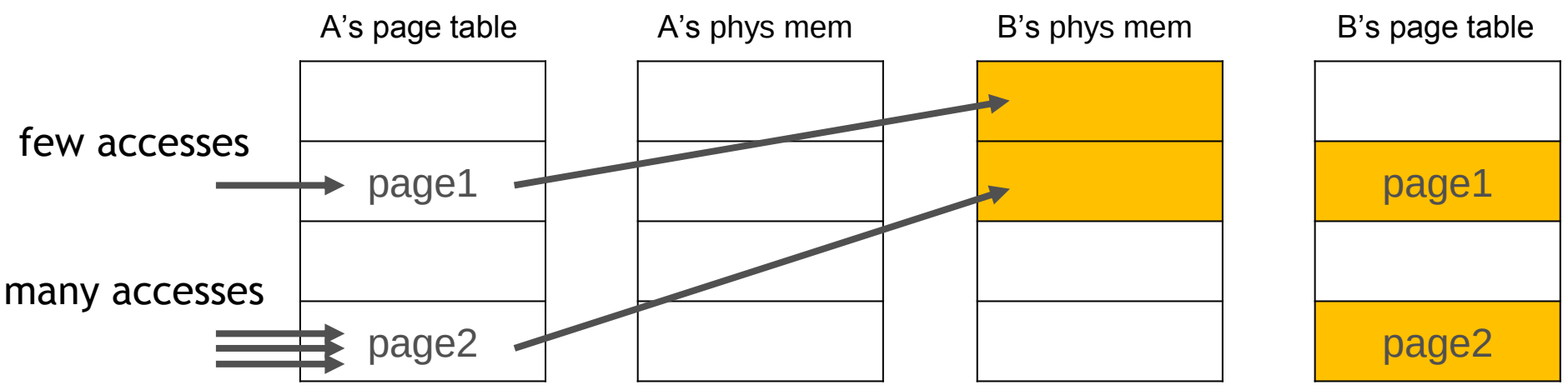
On-demand migration to accessing processor **on first touch**



UNIFIED MEMORY ON VOLTA+P9

New Feature: Access Counters

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

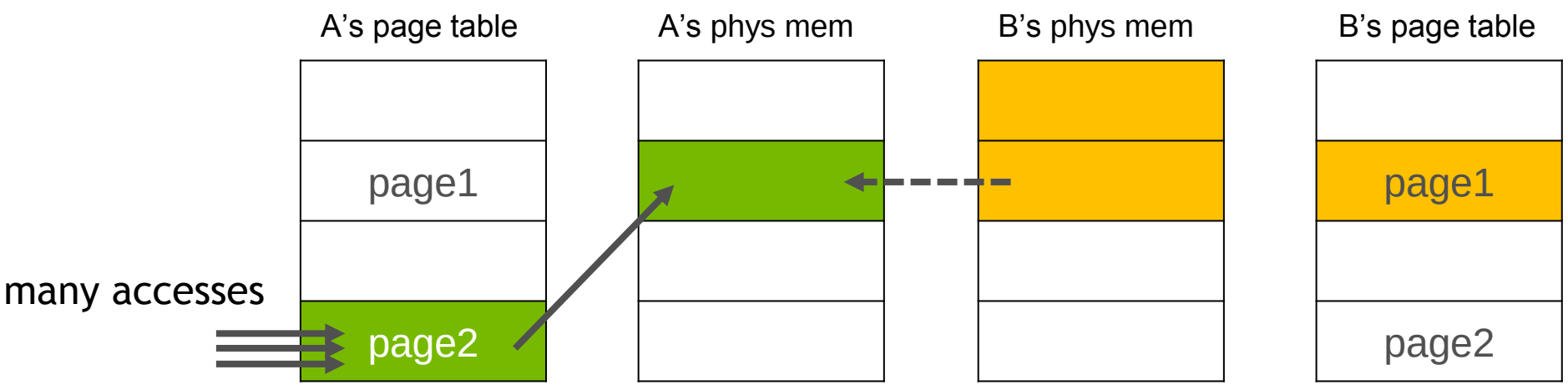


UNIFIED MEMORY ON VOLTA+P9

New Feature: Access Counters

With access counters **only hot pages** will be moved to the GPU

Migrations are *delayed* compared to the fault-based method



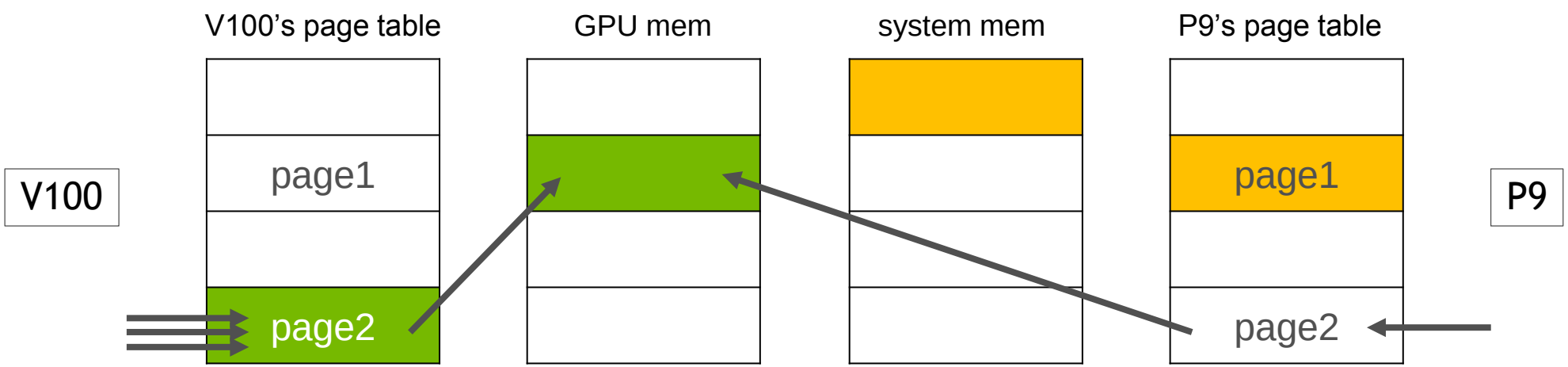
*When implemented this feature can be enabled with cudaMemAdvise policies

UNIFIED MEMORY ON VOLTA+P9

New Feature: Hardware Coherency with NVLINK2

CPU can directly access and *cache* GPU memory

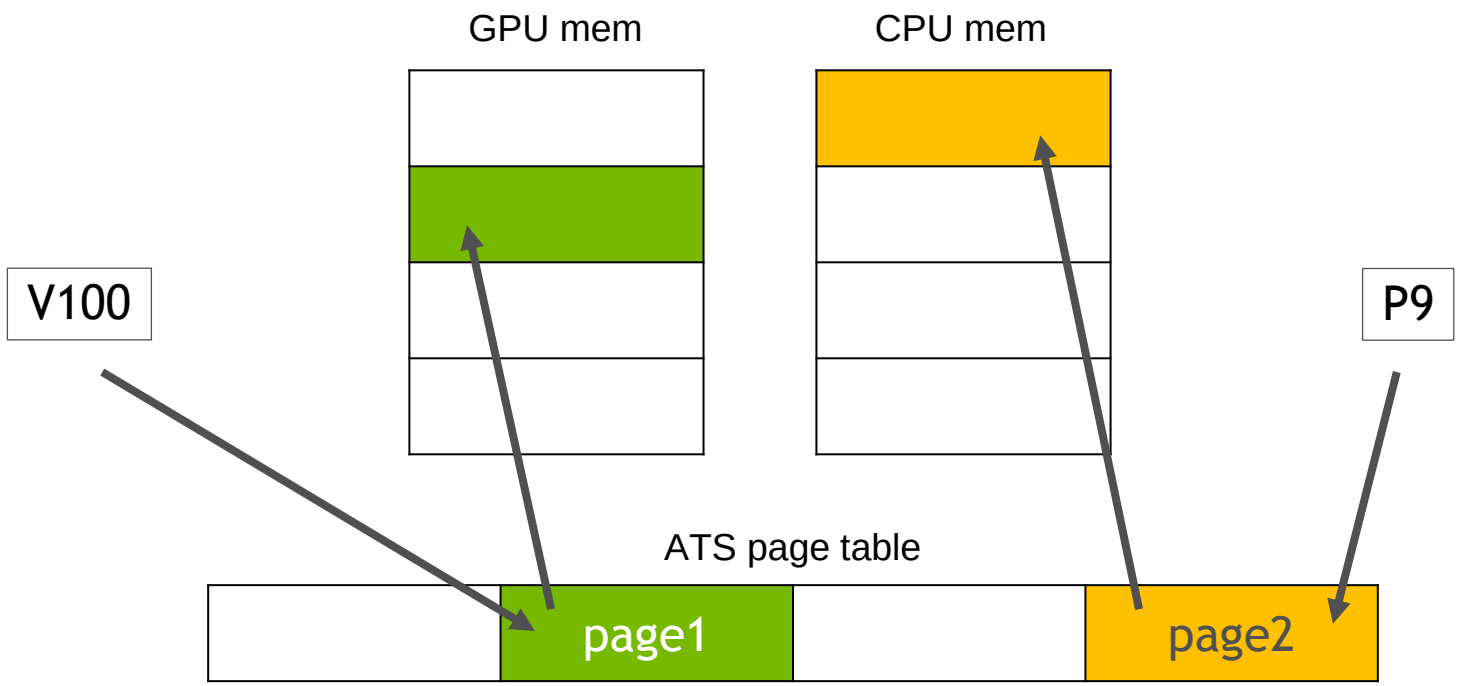
Native atomics support for all accessible memory



UNIFIED MEMORY ON VOLTA+P9

New Feature: ATS support

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



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)

Support enabled in CUDA 9.2

x86: Heterogeneous Memory Management (HMM)

Initial version of the patchset is integrated into 4.14 kernel

NVIDIA will be supporting upcoming versions of HMM

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

WHAT YOU CAN DO WITH UNIFIED MEMORY

See it in action at the end of the talk!

Works everywhere today

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

Works on Power9 + ATS in CUDA 9.2
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 = (int*)alloca(sizeof(int) * n);
kernel<<<grid, block>>>(data);
```

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

UNIFIED MEMORY LANGUAGES

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

CUDA Fortran: **managed** attribute (per allocation)

Python: **pycuda.driver.managed_empty** (allocate numpy.ndarray)

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

UNIFIED MEMORY + OPENACC

Effortless way to run you code on GPUs

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

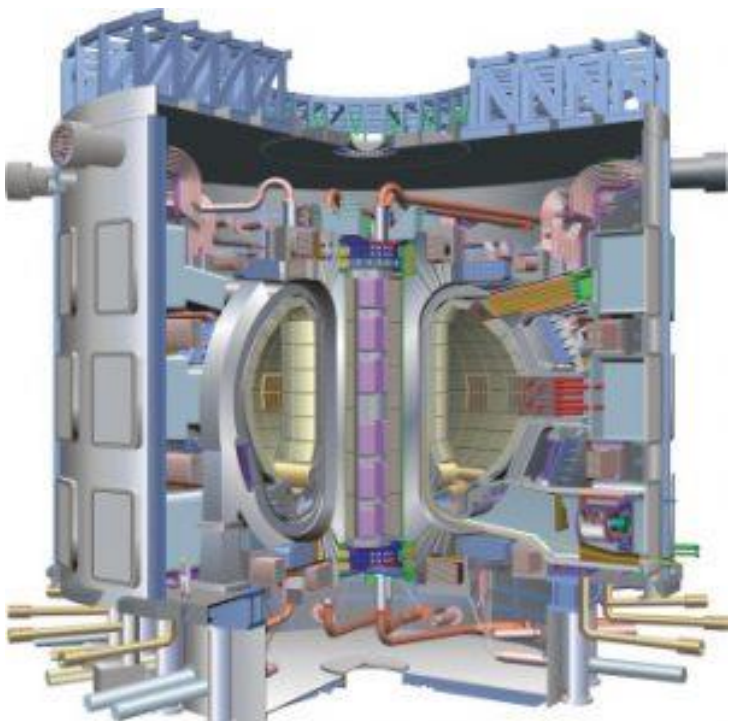
```
#pragma acc kernels
{
  for (i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
    ...
  }
}

...
```

Initiate parallel execution

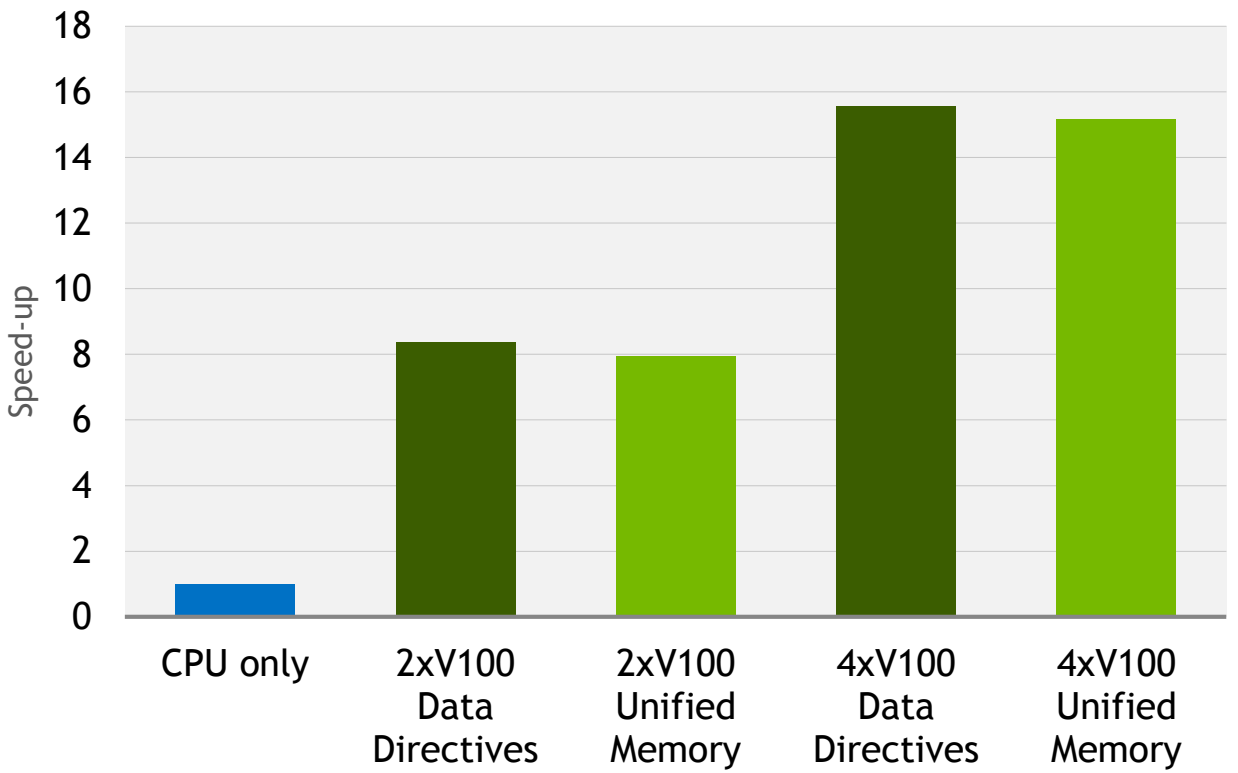
Easy to optimize later: add loop and data directives

GYROKINETIC TOROIDAL CODE

Particle-In-Cell production code



http://phoenix.ps.uci.edu/gtc_group



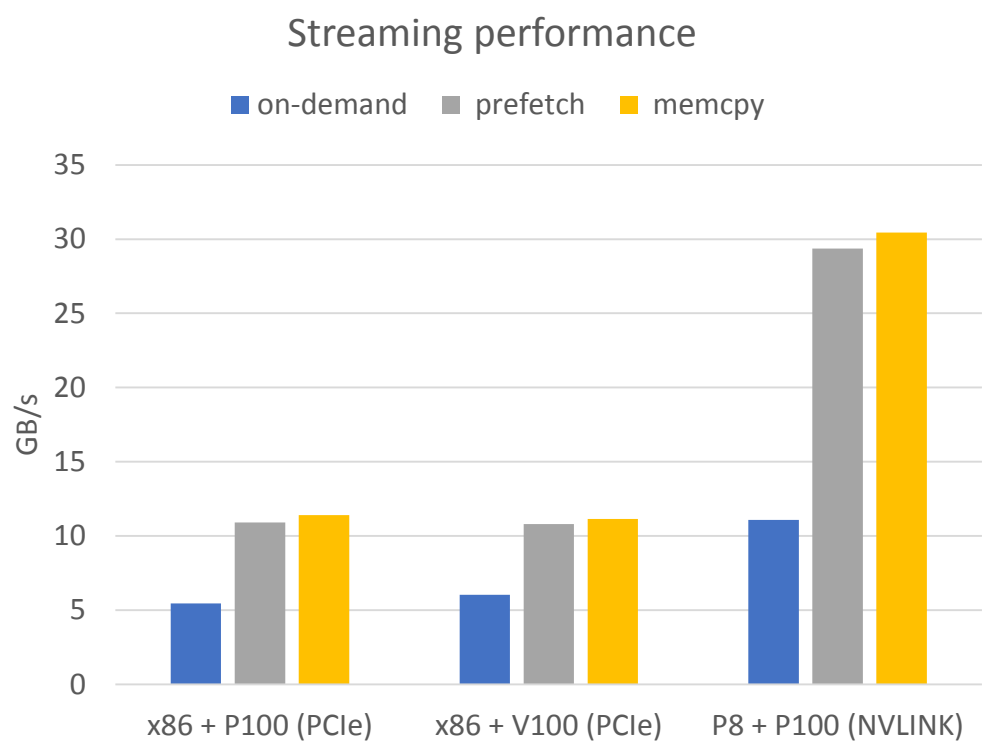
CPU: Haswell E5-2698 v3 @ 2.30GHz, dual socket 16-core; MPI: MVAPICH-GDR

PERFORMANCE DEEP DIVE

STREAMING BENCHMARK

How fast is on-demand migration?

```
__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);
cudaMalloc(&device, size);
memset(host, 0, size);

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



UNDERSTANDING PROFILER OUTPUT

```
==14487== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  23.270ms         1  23.270ms  23.270ms  23.270ms  void kernel(int*, int*)
      API calls:   79.56%  23.272ms         1  23.272ms  23.272ms  23.272ms  cudaDeviceSynchronize
                   20.42%  5.9732ms         1  5.9732ms  5.9732ms  5.9732ms  cudaLaunch
                    0.01%  2.0490us         1  2.0490us  2.0490us  2.0490us  cudaConfigureCall
                    0.01%  1.8360us         4     459ns     138ns     833ns  cudaSetupArgument

==14487== Unified Memory profiling result:
Device "Tesla V100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3012  21.758KB  4.0000KB  952.00KB  64.00000MB  13.49043ms  Host To Device
      81         -         -         -           -  23.23181ms  Gpu page fault groups
```

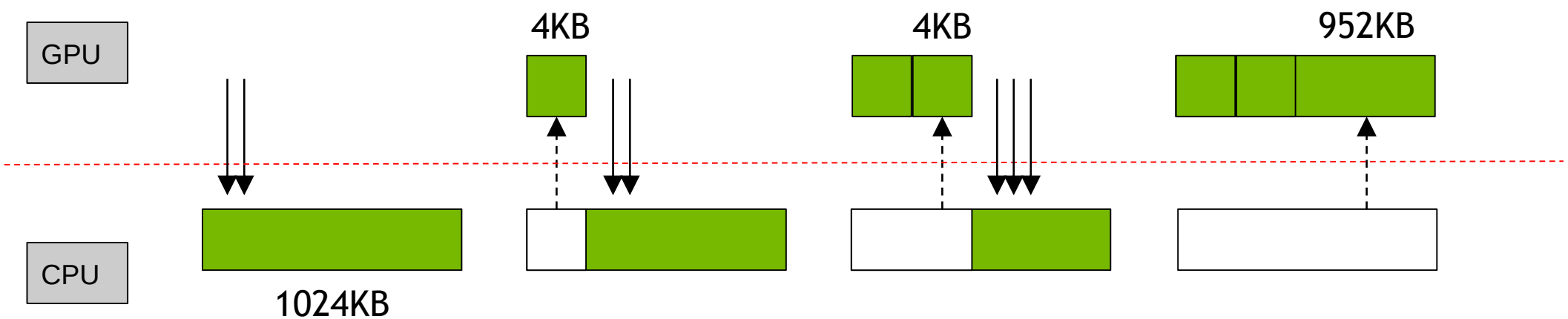
HEURISTIC PREFETCHING

Do Not Confuse with API-prefetching

GPU architecture supports different page sizes

Contiguous pages up to a large page size are promoted to the larger size

Driver prefetches whole regions if pages are accessed *densely*



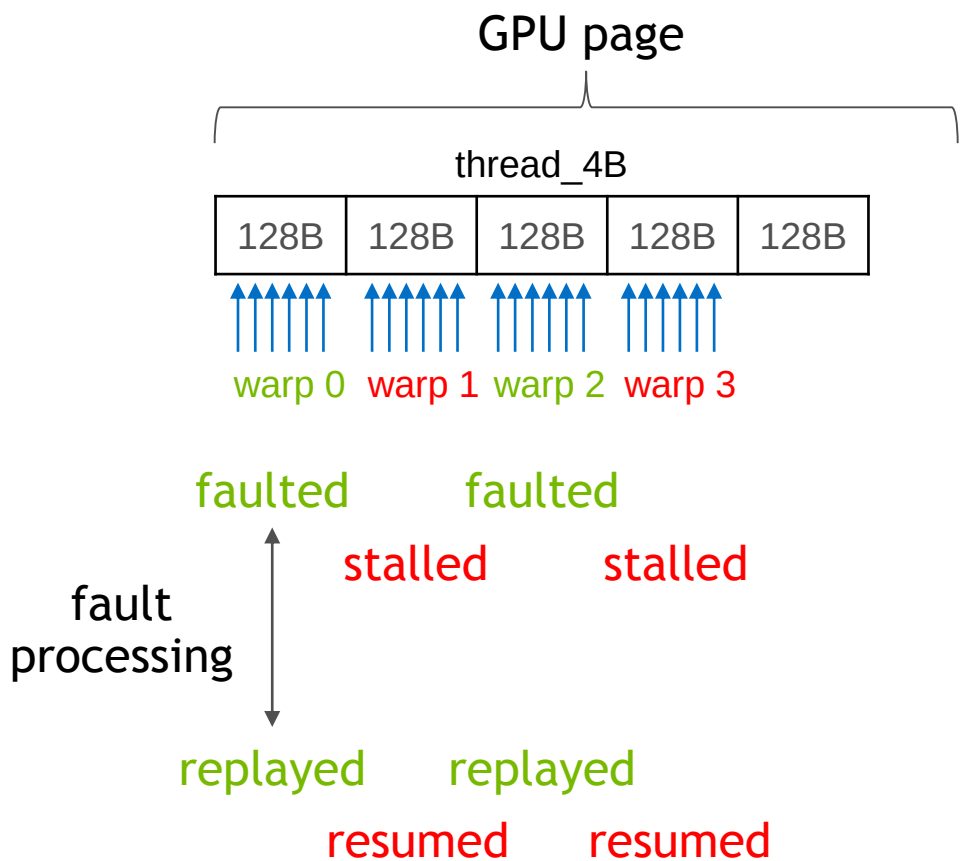
WHAT IS PAGE FAULT GROUPS?

```
==14487== Unified Memory profiling result:
Device "Tesla V100-PCIE-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3012  21.758KB  4.0000KB  952.00KB  64.00000MB  13.49043ms  Host To Device
      81         -         -         -           -  23.23181ms  Gpu page fault groups
```

nvprof --print-gpu-trace ...

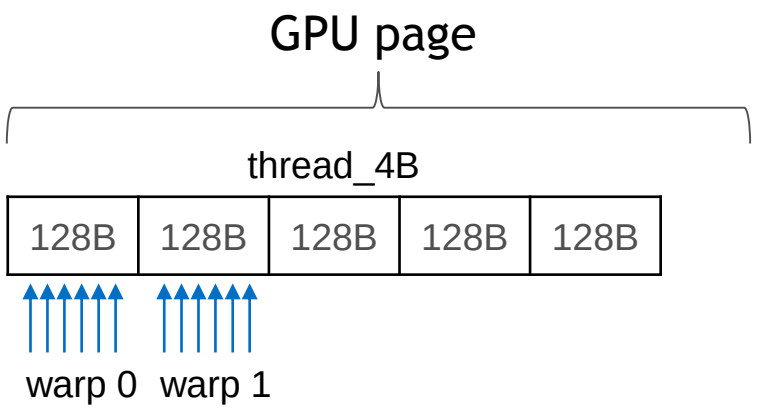
```
Unified Memory  Virtual Address  Name
             8     0x3900010000  [Unified Memory GPU page faults]
             9     0x3900040000  [Unified Memory GPU page faults]
             5     0x3900108000  [Unified Memory GPU page faults]
             5     0x3900200000  [Unified Memory GPU page faults]
```

PAGE FAULTS HANDLING

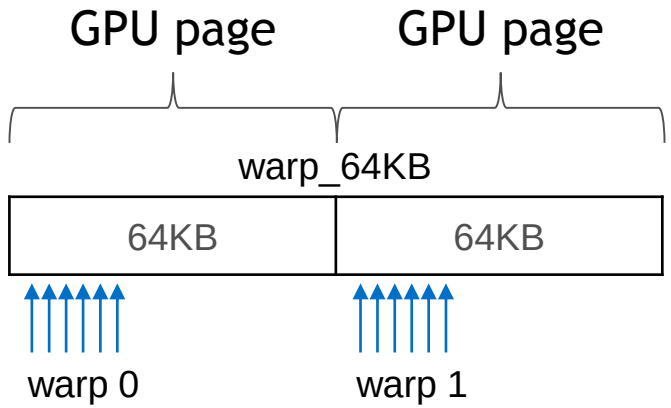


OPTIMIZING ON-DEMAND MIGRATION

Increase fault concurrency to reduce page fault stalls



multiple faults per page
warps are stalled on fault processing



fewer warps are stalled
“spread-out” pattern improves prefetching

OPTIMIZING ON-DEMAND MIGRATION

Thread/4B

```
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3012  21.758KB  4.0000KB  952.00KB  64.00000MB  13.49043ms  Host To Device
      81         -         -         -           -  23.23181ms  Gpu page fault groups

Unified Memory  Virtual Address  Name
             8     0x3900010000  [Unified Memory GPU page faults]
             9     0x3900040000  [Unified Memory GPU page faults]
             5     0x3900108000  [Unified Memory GPU page faults]
```

Warp/64KB

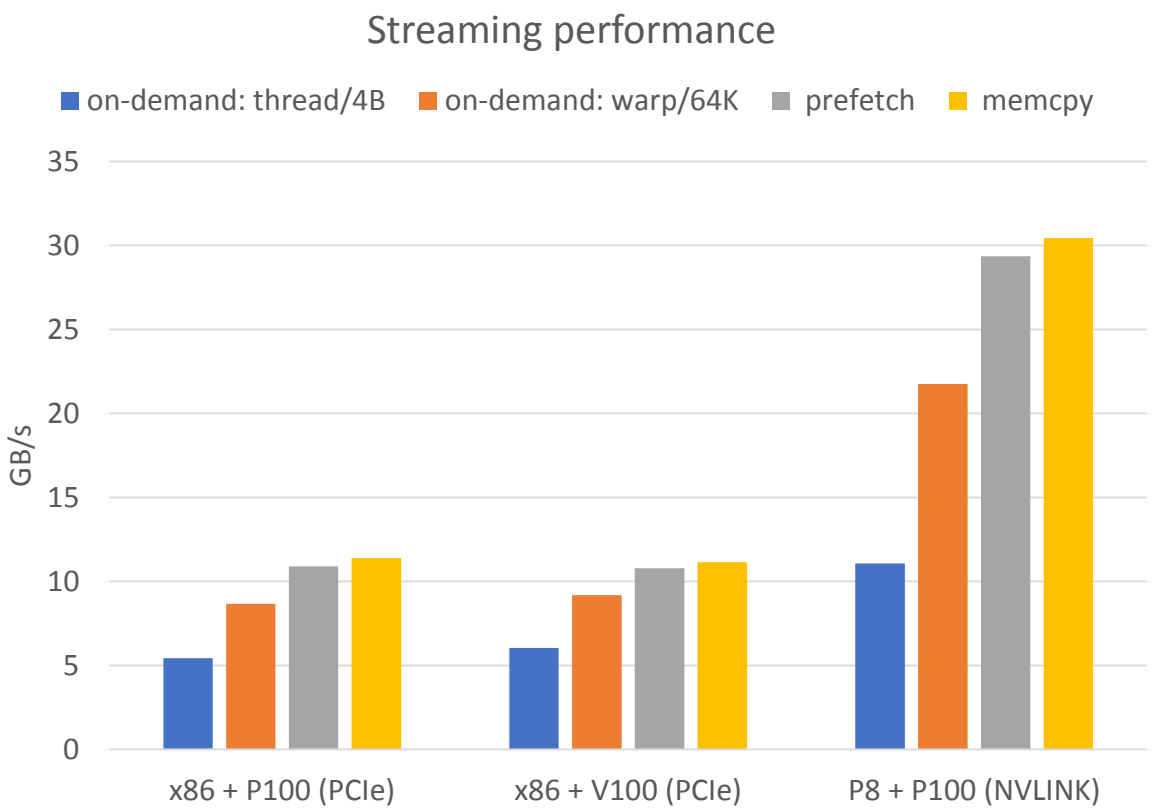
```
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     957  68.481KB  4.0000KB  576.00KB  64.00000MB  8.242080ms  Host To Device
       6         -         -         -           -  9.769984ms  Gpu page fault groups

Unified Memory  Virtual Address  Name
             1     0x39000d0000  [Unified Memory GPU page faults]
             1     0x39000c0000  [Unified Memory GPU page faults]
             1     0x3900080000  [Unified Memory GPU page faults]
```

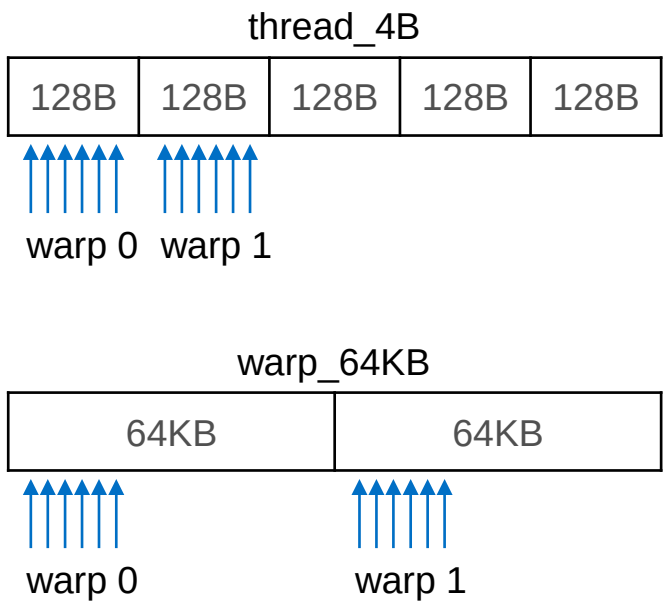
more efficient prefetching

fewer stalls

STREAMING BENCHMARK

How fast is on-demand migration?

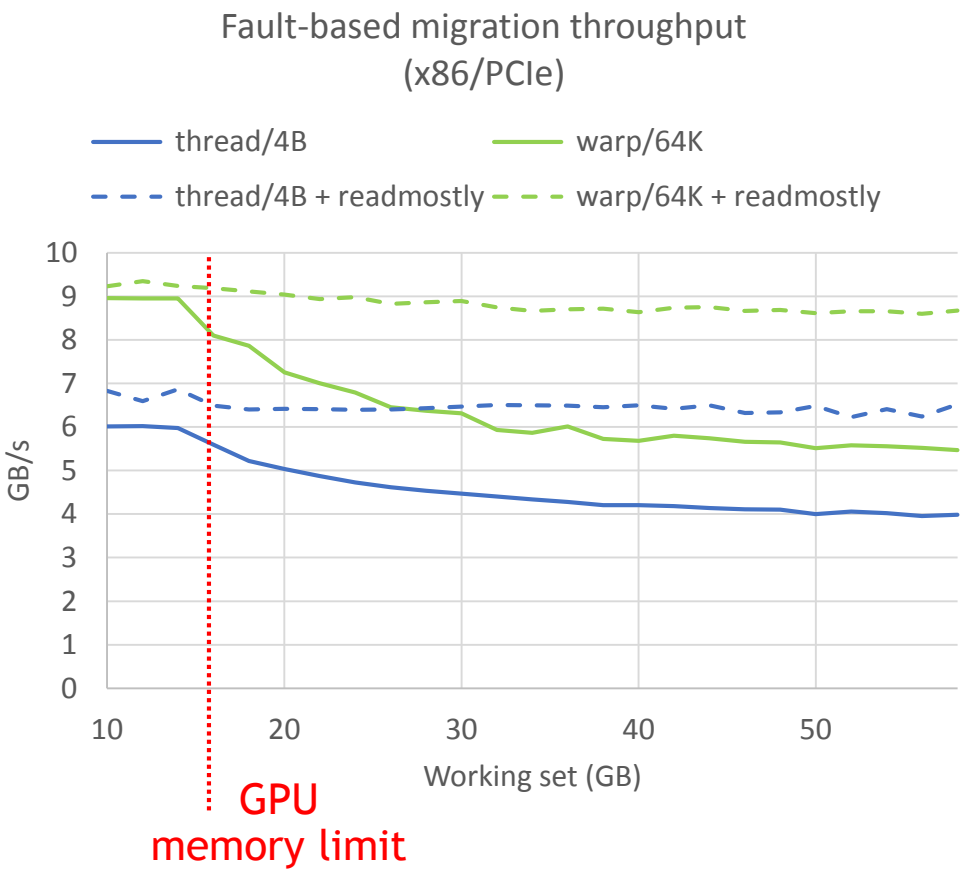


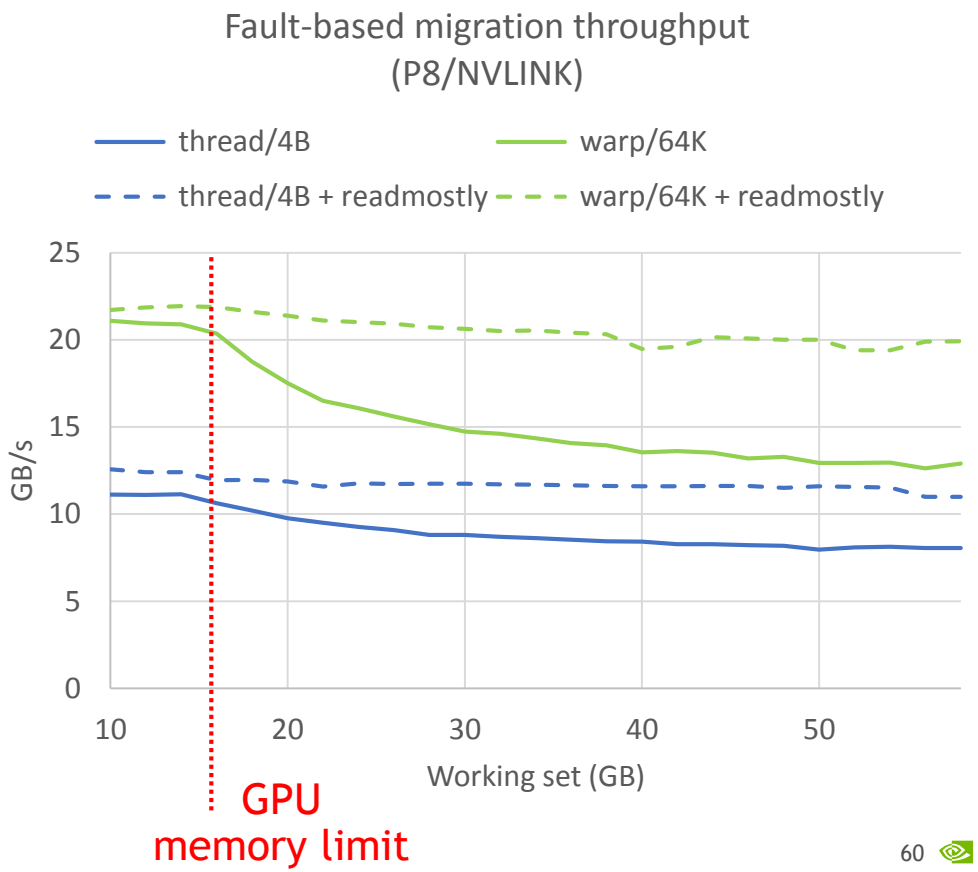


Also check the Parallel Forall blog: https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

GPU MEMORY OVERSUBSCRIPTION

Let's see how perf changes as we increase the working set





EVICTION ALGORITHM

What Pages Are Moving Out of the GPU

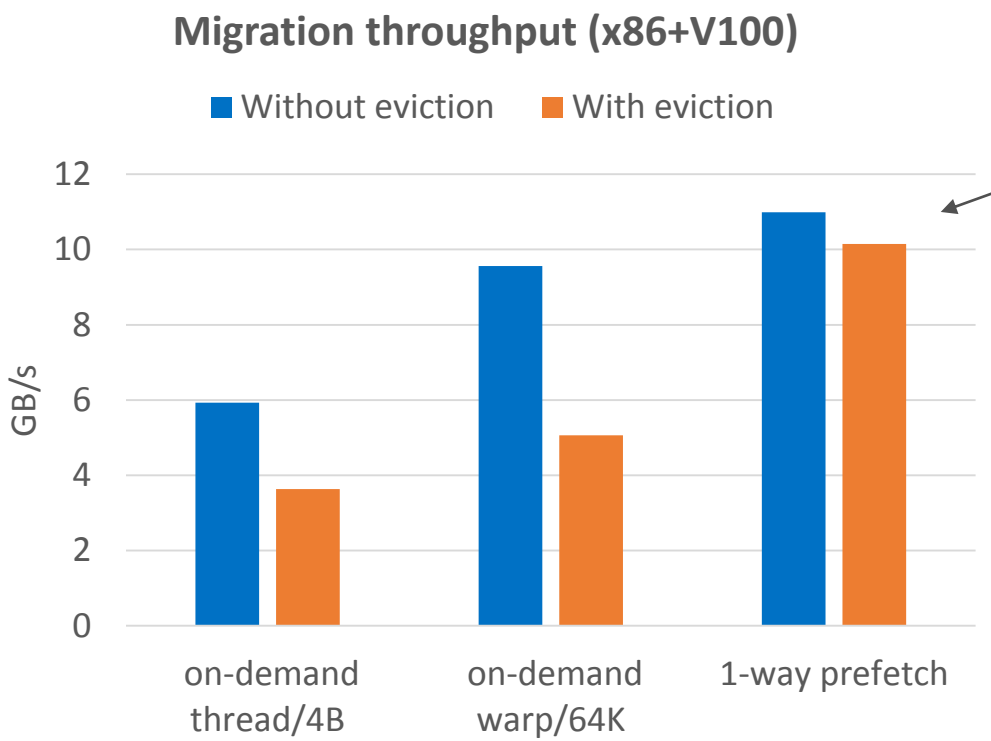
Driver keeps a list of physical chunks of GPU memory

Chunks from the front of the list are evicted first (LRU)

A chunk is considered “in use” when it is fully-populated or migrated

Prefetching and policies may impact eviction heuristic in the future

PREFETCHING AND EVICTIONS

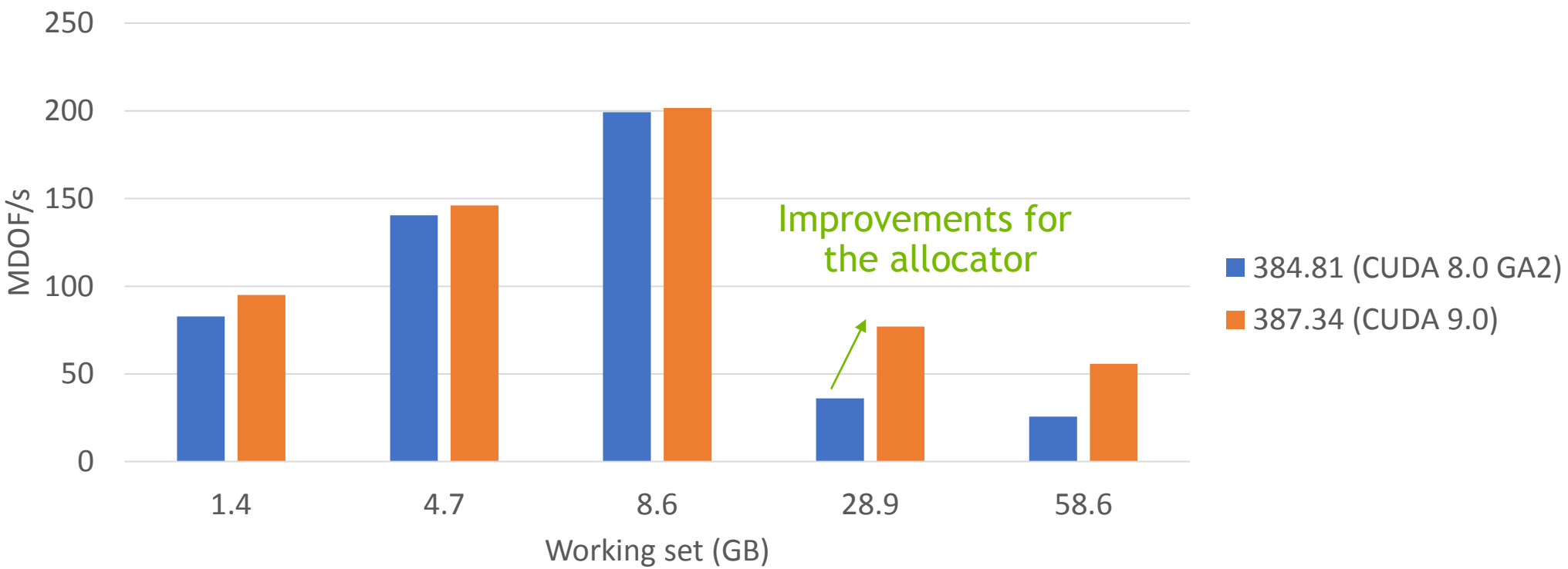


Prefetch can be overlapped with evictions
without using CUDA streams!
(enabled in CUDA 9.1)

cudaMemcpy solution requires scheduling DtoH and HtoD copies into two separate streams

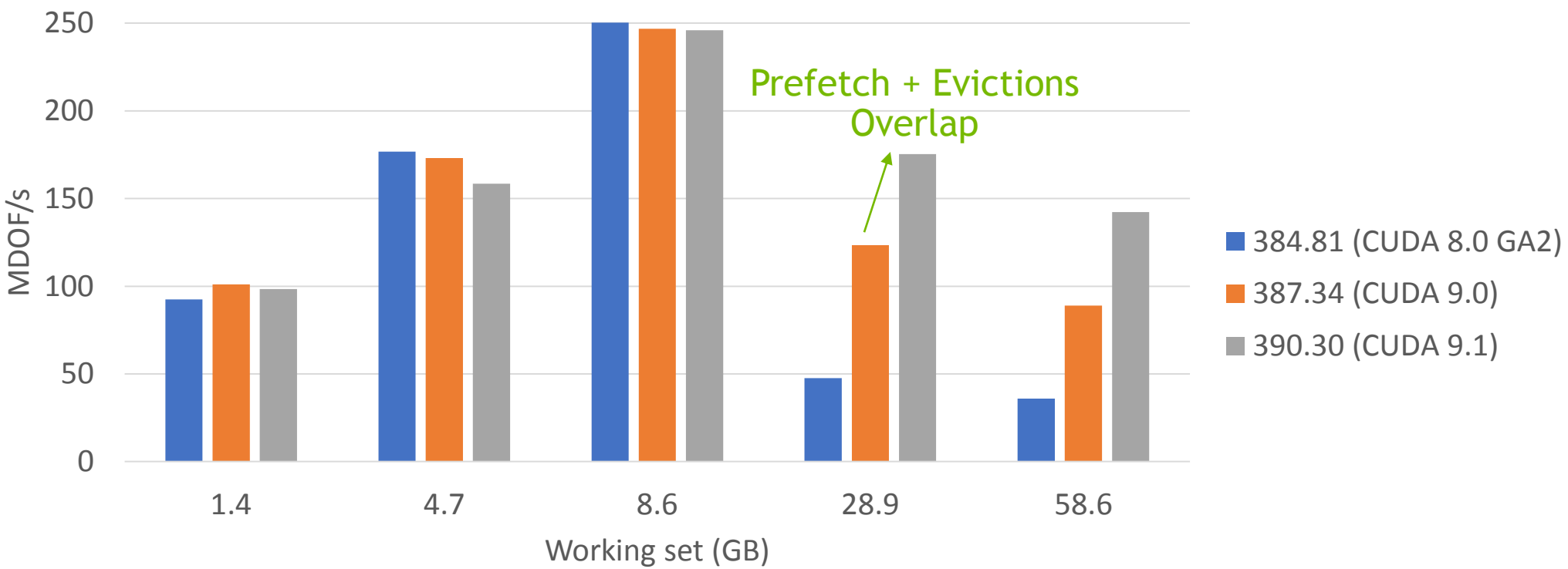
DRIVER ENHANCEMENTS

Tesla V100 + x86: HPGMG-AMR default (no hints)



DRIVER ENHANCEMENTS

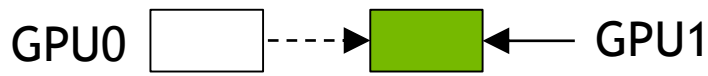
Tesla V100 + x86: HPGMG-AMR optimized (prefetches)



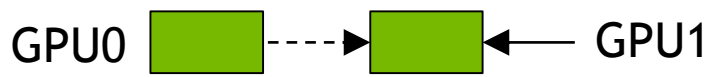
LOCALITY AND ACCESS CONTROL

cudaMemAdvise

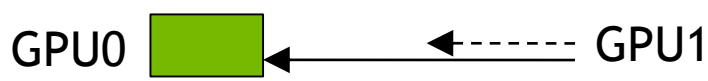
Default: data *migrates* on first touch



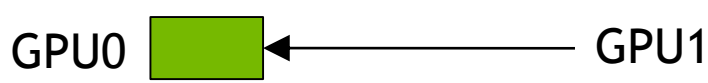
ReadMostly: data *duplicated* on first touch



PreferredLocation: *resist* migrating away from the preferred location



AccessedBy: establish *direct mapping* and avoid faults



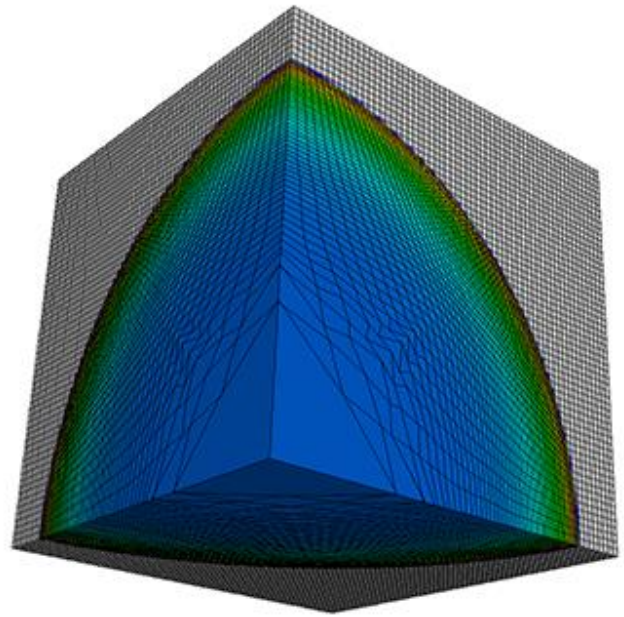
LULESH

CORAL OpenACC app

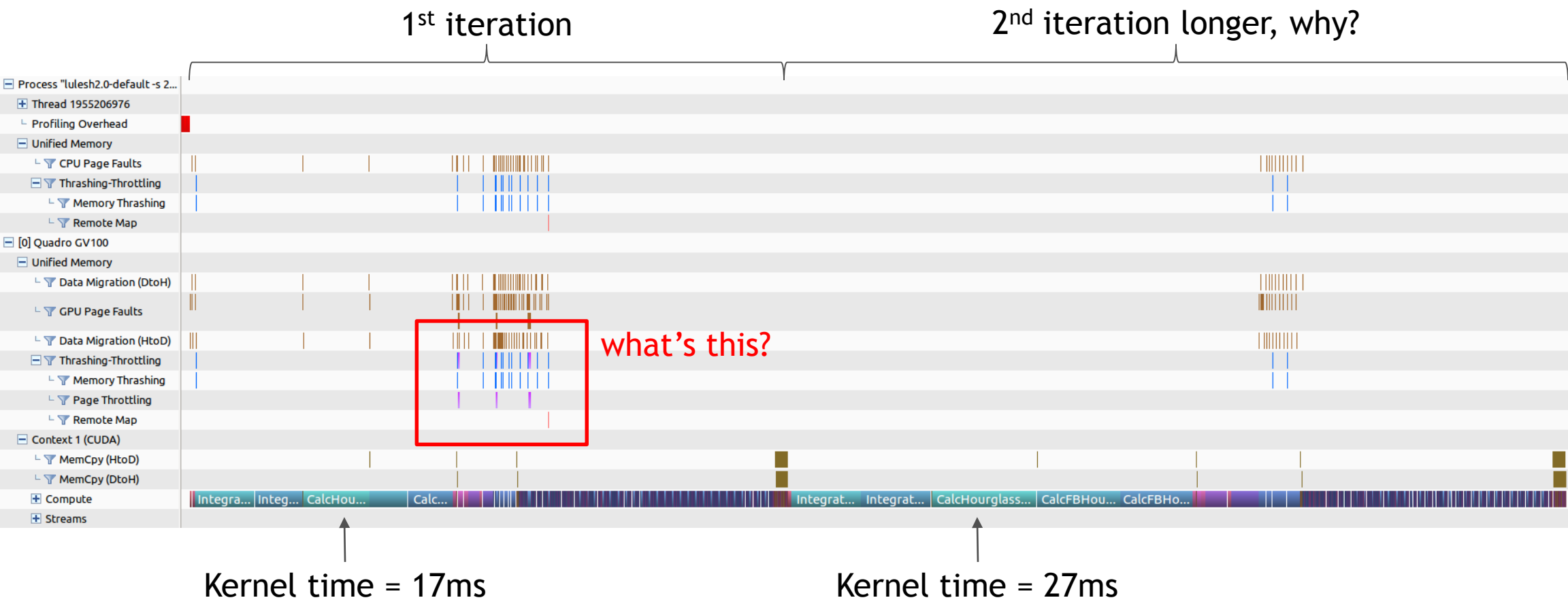
```
while not converged:
  LagrangeNodal
    CalcForceForNodes
    CalcAccelerationForNodes
    BoundaryConditionsForNodes
    CalcVelocityForNodes
    CalcPositionForNodes
  LagrangeElements
    CalcLagrangeElements
    CalcQForElems
    MaterialPropertiesForElems
  CalcTimeConstraintsForElems
    CalcCourantConstraintForElems
    CalcHydroConstraintForElems
```

few heavy GPU kernels

many small GPU kernels
many small CPU functions

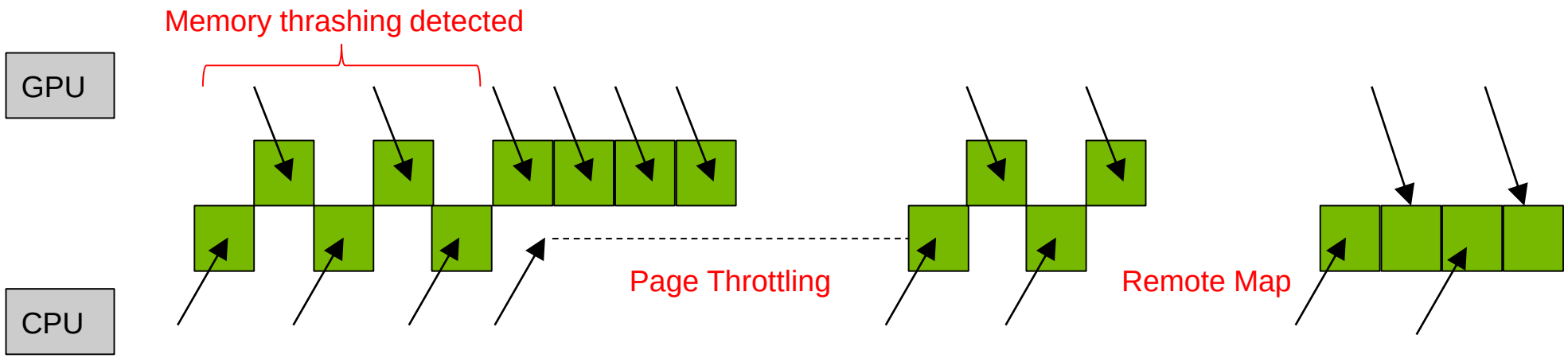


USING VISUAL PROFILER



THRASHING MITIGATION

Processors frequently read or write to the same page

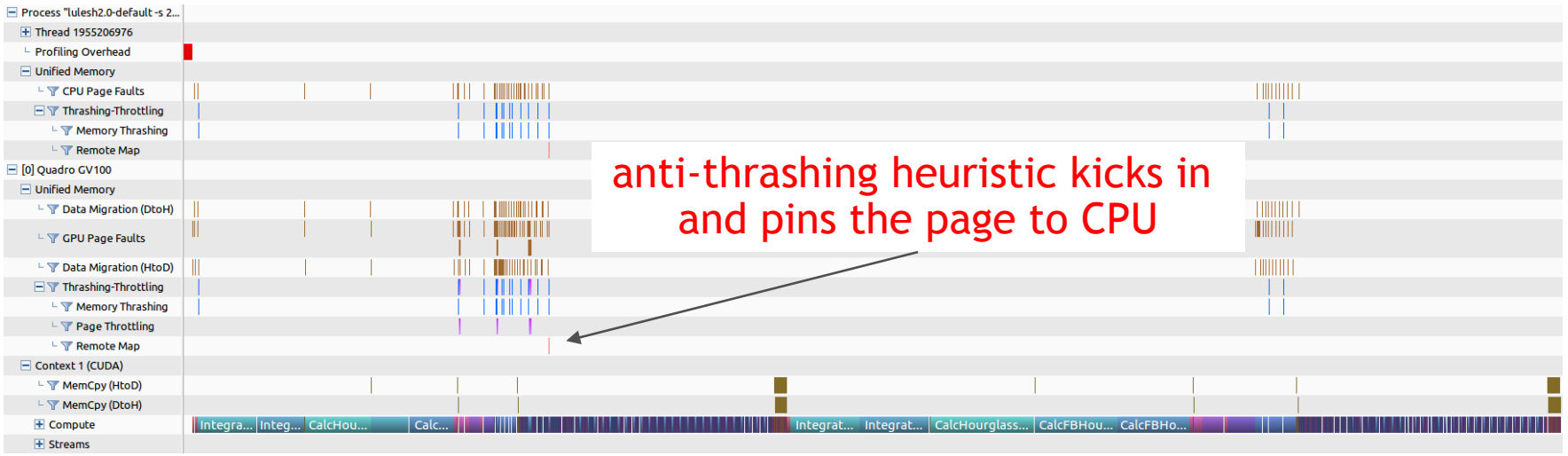


Before CUDA 9.2: when memory is pinned we lose any insight into access pattern

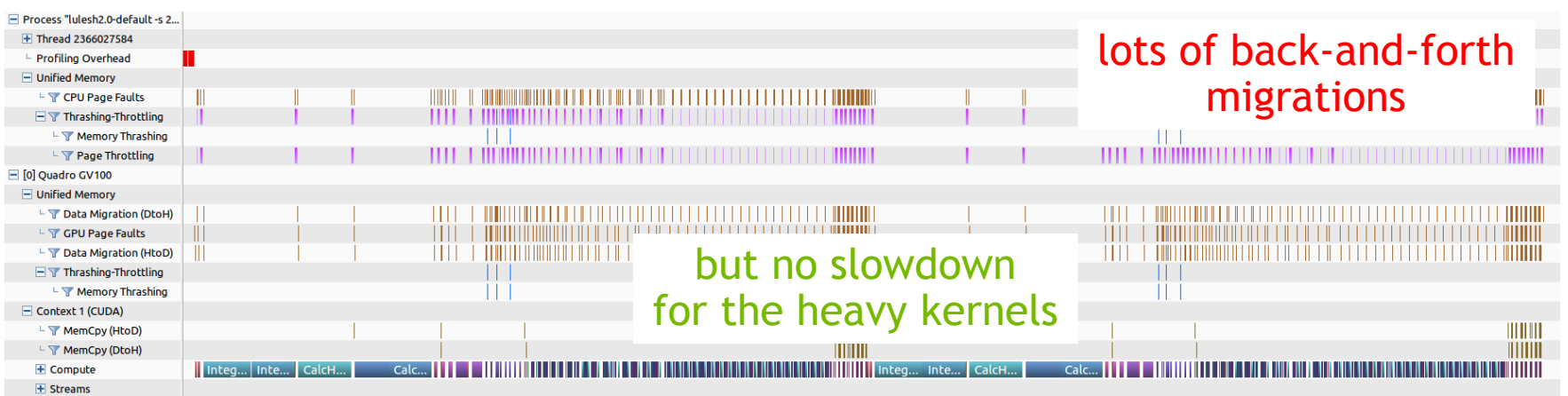
In the future we may use access counters on Volta to find a better location

LULESH: PREFERRED LOCATION = GPU

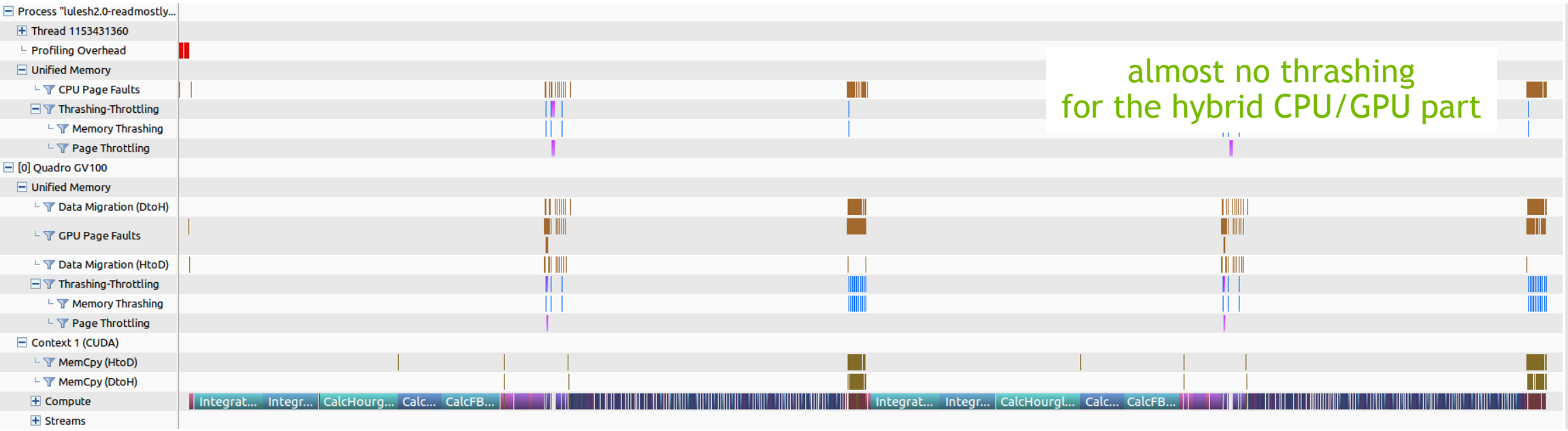
No policies



Preferred Location



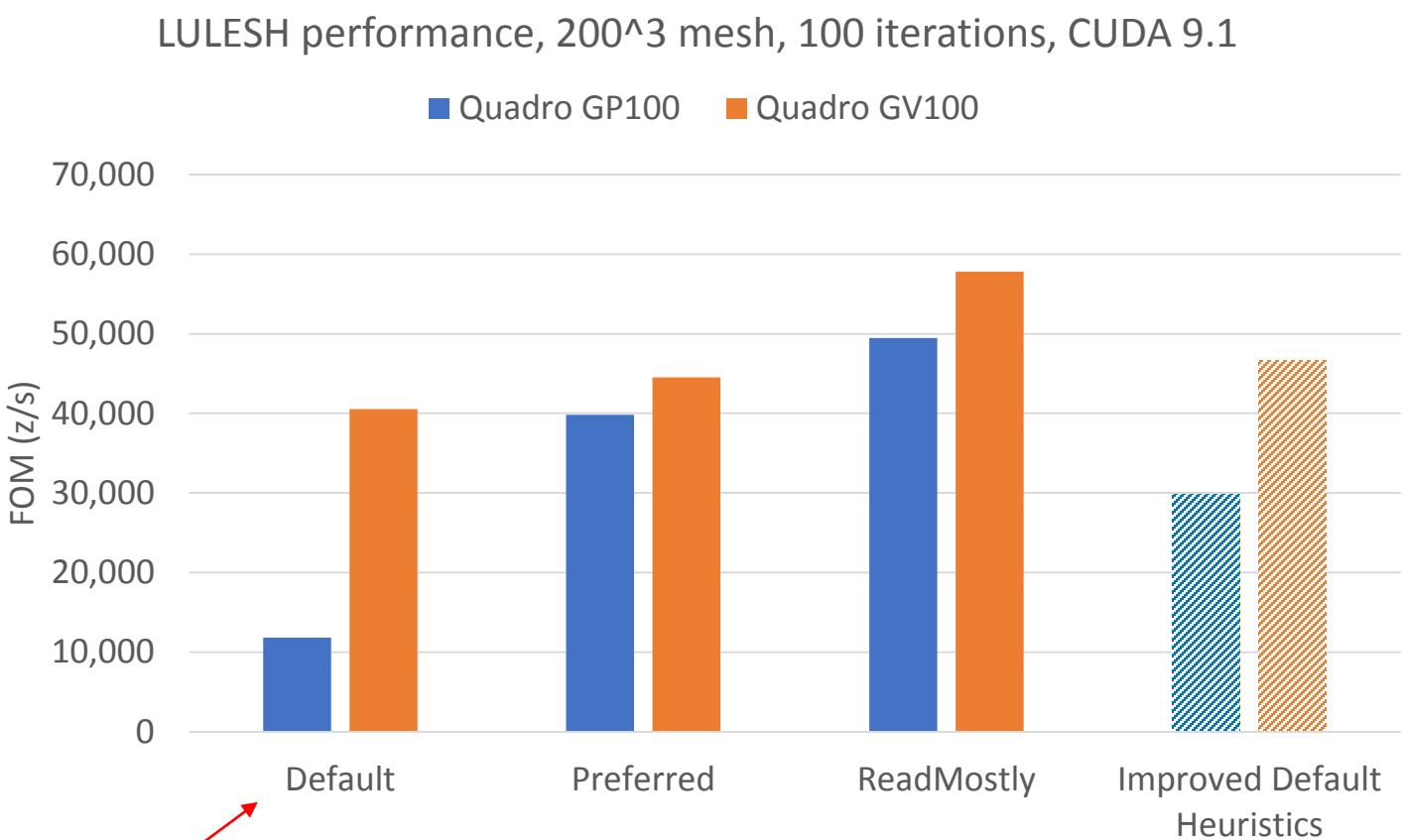
LULESH: READ MOSTLY



almost no thrashing
for the hybrid CPU/GPU part

no slowdown
for the heavy kernels

LULESH: PERF IMPROVEMENTS



Will be enabled in the future drivers

GP100 has 500x more sysmem writes than GV100

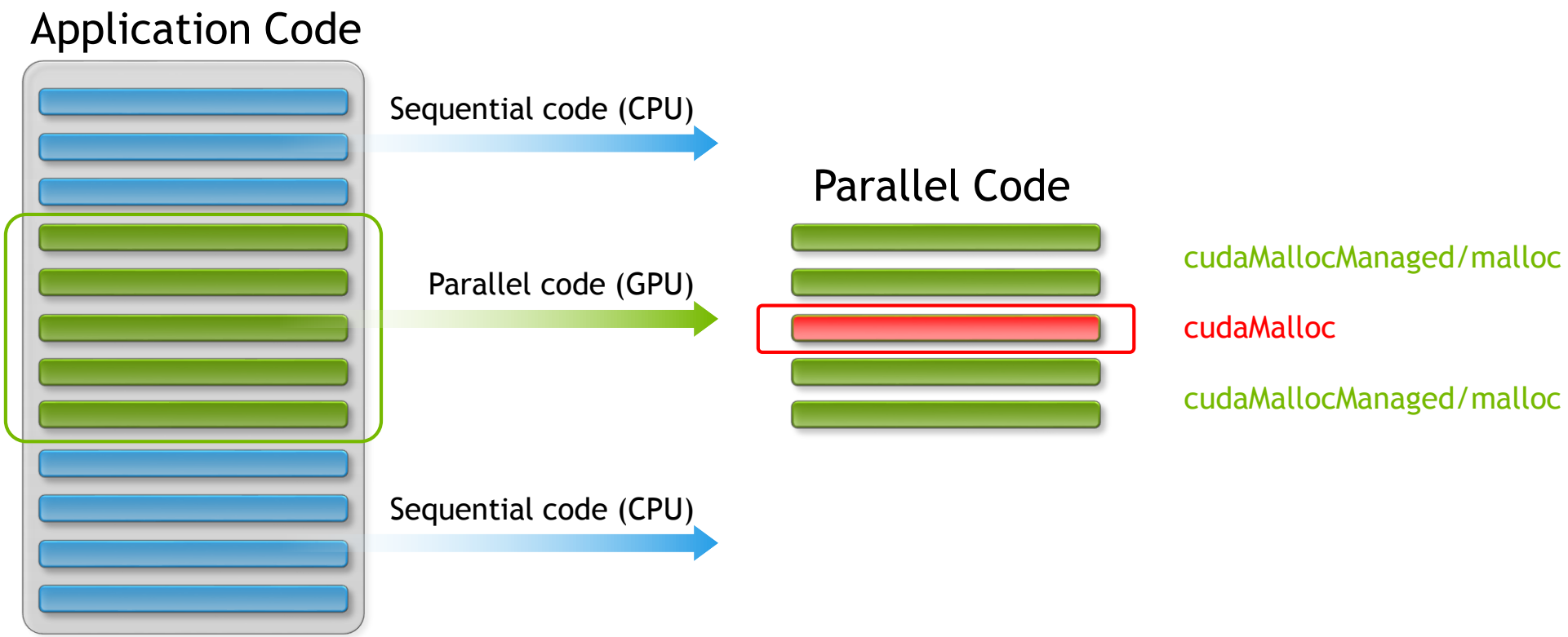
WHEN TO USE UNIFIED MEMORY

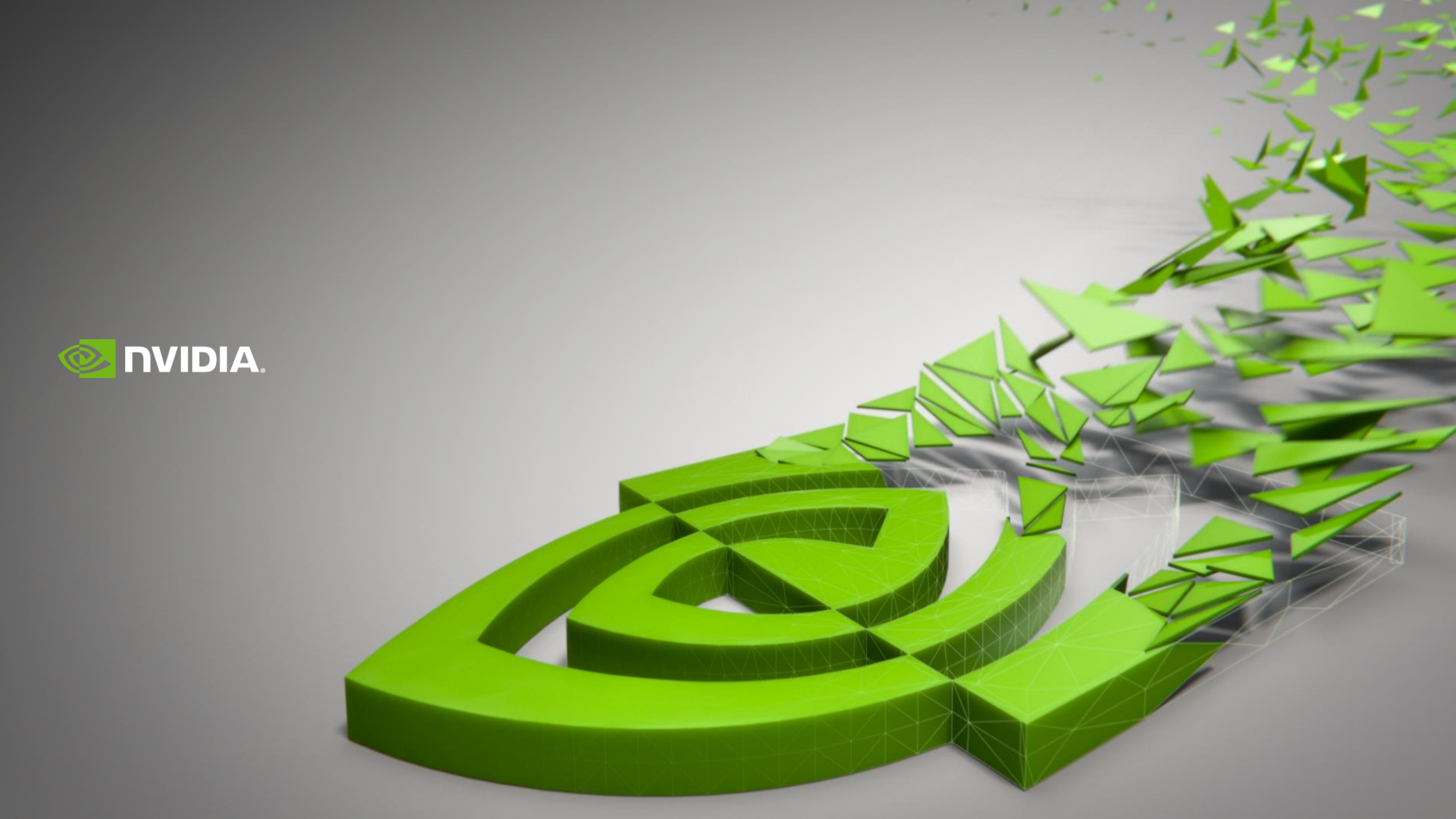
	cudaMalloc	cudaMallocManaged

Pinned allocation	`cudaMalloc`	`cudaMallocManaged`
		`PreferredLocation(GPU)`
		`SetAccessedBy(peer GPUs)`
		`cudaMemPrefetchAsync(GPU)`

cudaMemcpy: ptrA -> ptrB	Staging for non-pinned allocations or between non-P2P GPUs	Staging or a copy kernel required in all cases
Memory migration	Not possible	`cudaMemPrefetchAsync`
Debugging	Difficult	Easy
Oversubscription	No	Yes
IPC support	Yes	No

WHEN TO USE UNIFIED MEMORY




NVIDIA

UNIFIED MEMORY PLATFORMS

	KEPLER	PASCAL	VOLTA
Linux + x86	No GPU fault support No concurrent access	On-demand migration	On-demand migration
Linux + Power		On-demand migration 80GB/s CPU-GPU BW*	On-demand migration 150GB/s CPU-GPU BW** Access counters HW coherency ATS support
Windows	No GPU fault support No concurrent access		
MacOS	No GPU fault support No concurrent access		
Tegra	Cached on CPU and iGPU No concurrent access		

**IBM Minsky: 4xP100 + 2xP8, 2xNVLINK1 links between P100 and P8, bi-directional aggregate BW*
***IBM Newell: 4xV100 + 2xP9, 3xNVLINK2 links between V100 and P9, bi-directional aggregate BW*

READ DUPLICATION

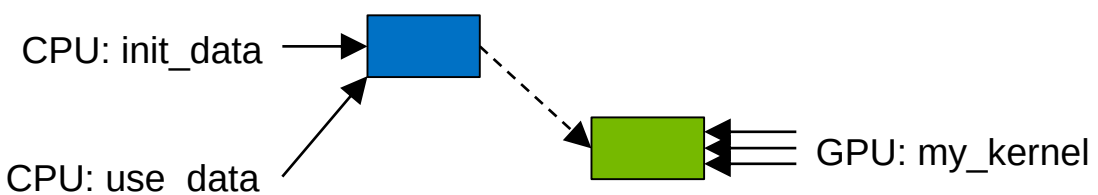
Usage example

```
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



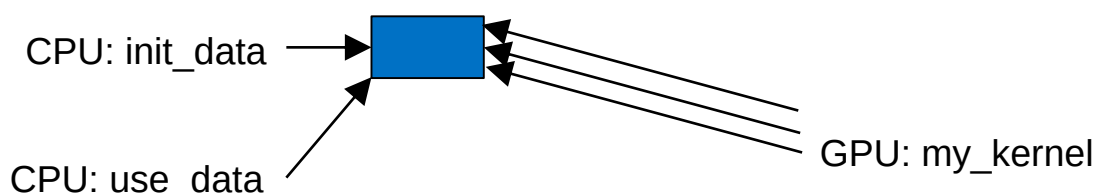
PREFERRED LOCATION

Resisting migrations

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

init_data(data, N);

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

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


use_data(data, N);

cudaFree(data);
```

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

The driver will “resist” migrating data away from the preferred location



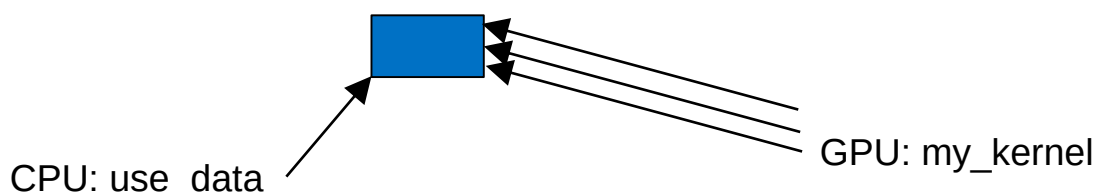
PREFERRED LOCATION

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



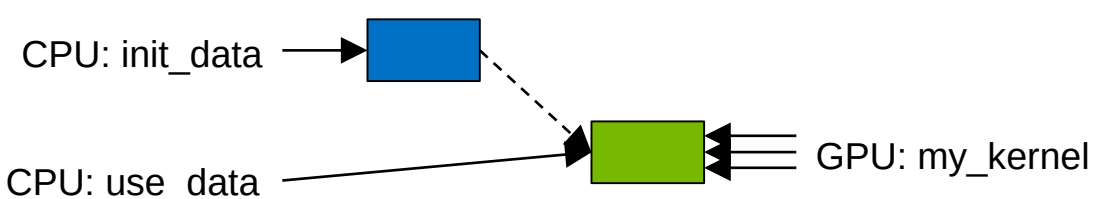
PREFERRED LOCATION ON P9+V100

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

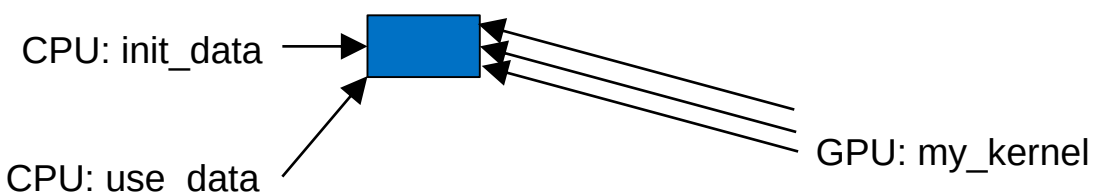
ACCESSED BY

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



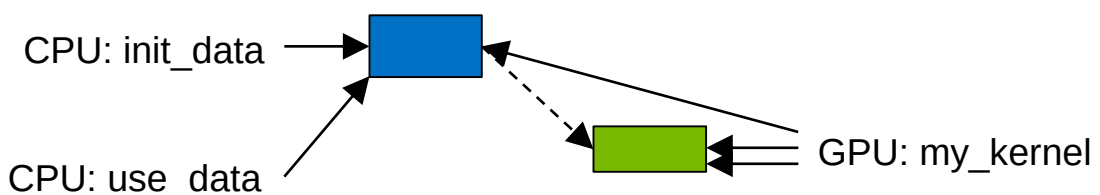
ACCESSED BY

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



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**

**You may alter this behavior by using cudaMemAdvise policies*

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);
```

GPU uses ATS to access CPU memory (no on-demand migration except cudaMemPrefetchAsync*)

CPU accesses CPU memory

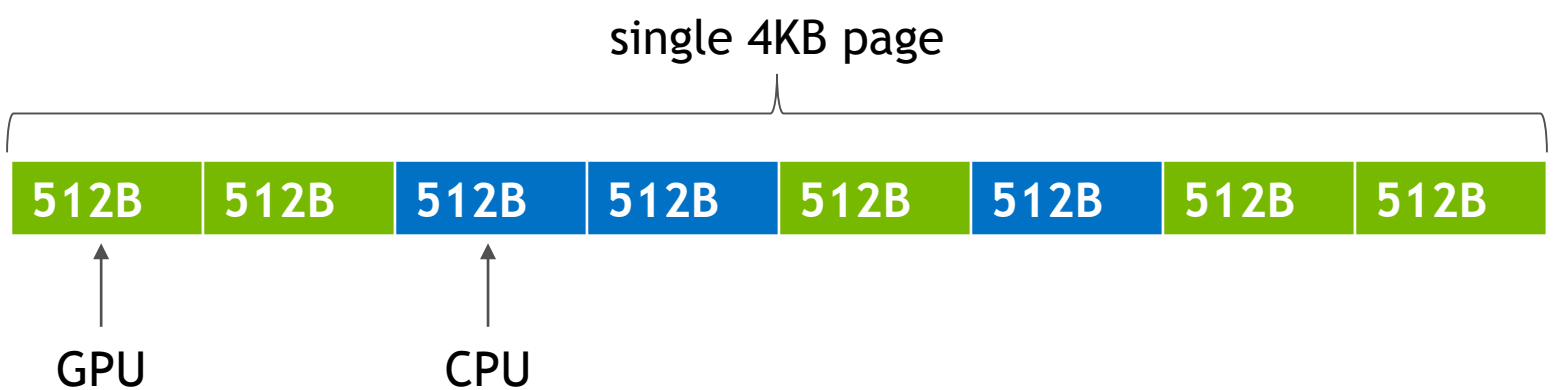
*In the future Volta access counters will be used to migrate malloc memory

HYPRE-INSPIRED USE CASE

Algebraic Multi-Grid library: https://github.com/LLNL/hypre

Lots of small allocations: multiple variables may end up on the same page

If used by different processors this will result in **false-sharing**



FALSE-SHARING

Issues with false-sharing:

- Spurious migrations, thrashing mitigation does not solve it
- Performance hints are applied on page boundaries, due to suballocation data may inherit the wrong policies

How to mitigate this:

- Use separate allocators or memory pools for CPU and GPU