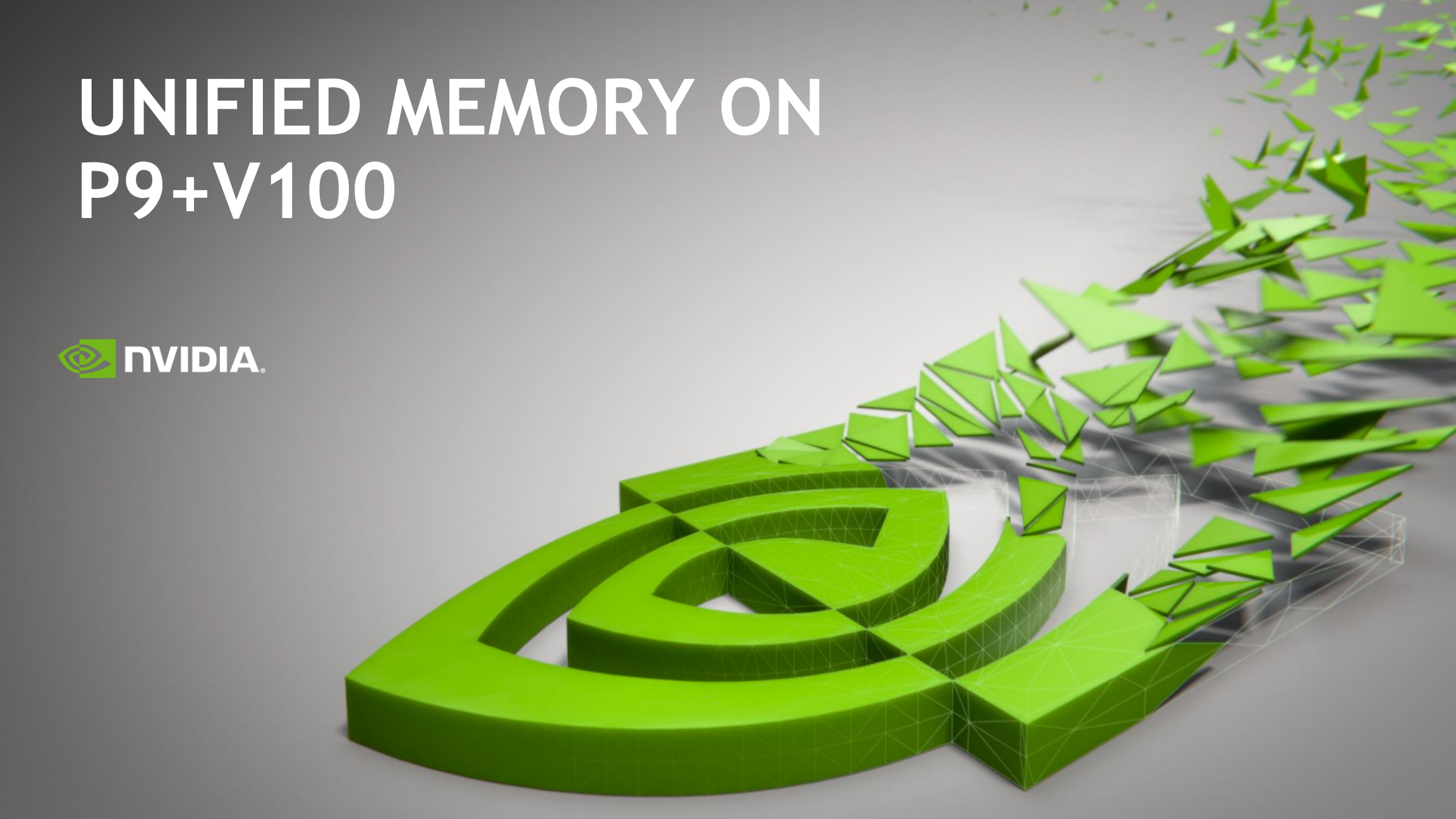
UNIFIED MEMORY ON P9+V100



UNIFIED MEMORY FUNDAMENTALS

Single pointer

On-demand migration

GPU memory oversubscription

Concurrent CPU/GPU access

System-wide atomics

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

cpu_func1(data, N);

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

cpu_func3(data, N);

free(data);
```

P9+V100 CLARIFICATIONS

cudaMallocManaged still works as before (including performance hints)

cudaMalloc still not accessible from the CPU

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

cpu_func1(data, N);

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

cpu_func3(data, N);

cudaFree(data);
```

PERFORMANCE HINTS

cudaMemPrefetchAsync(ptr, size, processor, stream)

Similar to cudaMemcpyAsync

cudaMemAdvise(ptr, size, advice, processor)

ReadMostly: duplicate pages, writes possible but expensive

PreferredLocation: “resist” migrations from the preferred location

AccessedBy: establish mappings to avoid migrations and access directly

USER HINTS

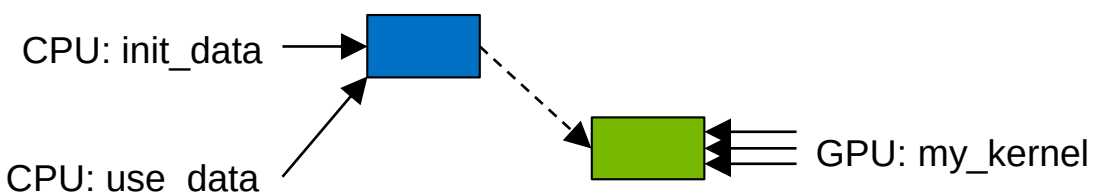
Read Mostly

```
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);
```

In this case prefetch creates a copy instead of moving data

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

Writes invalidate the copies on other processors (*expensive*)



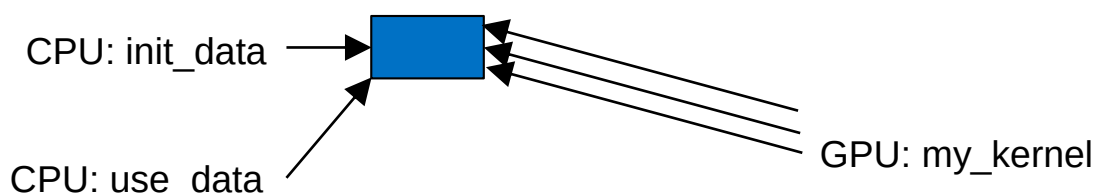
USER HINTS

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);
```

Here 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



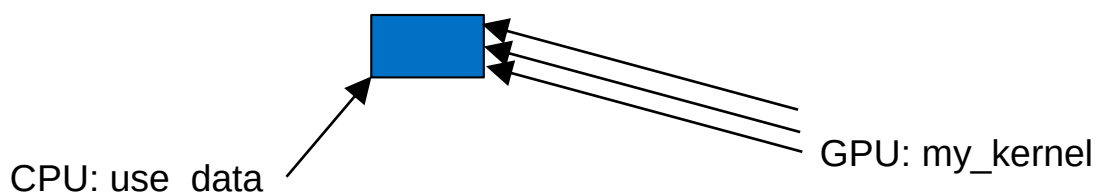
USER HINTS

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);
```

Here 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



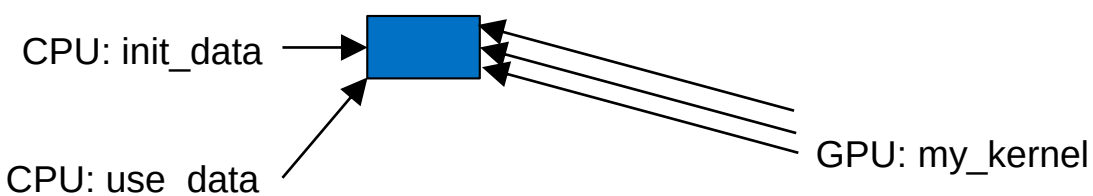
USER HINTS

Accessed By

```
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



P9+V100 NEW FEATURES

Available since CUDA 9.1:

- NVLINK2: increased migration throughput
- NVLINK2: enabling 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

Work in progress:

- cudaMallocManaged may use access counters to guide migrations (opt-in)
- ATS: GPU can access all system memory (malloc, stack, mmap files)

NVLINK2: INCREASED THROUGHPUT

	GPUS PER CPU	CPU-GPU	GPU-GPU*
2xP8+4xP100	2	40GB/s	40GB/s
2xP9+6xV100 (1)	3	50GB/s	50GB/s
2xP9+4xV100 (2)	2	75GB/s	75GB/s

Copies, prefetches and direct access can achieve roughly 80% of peak BW

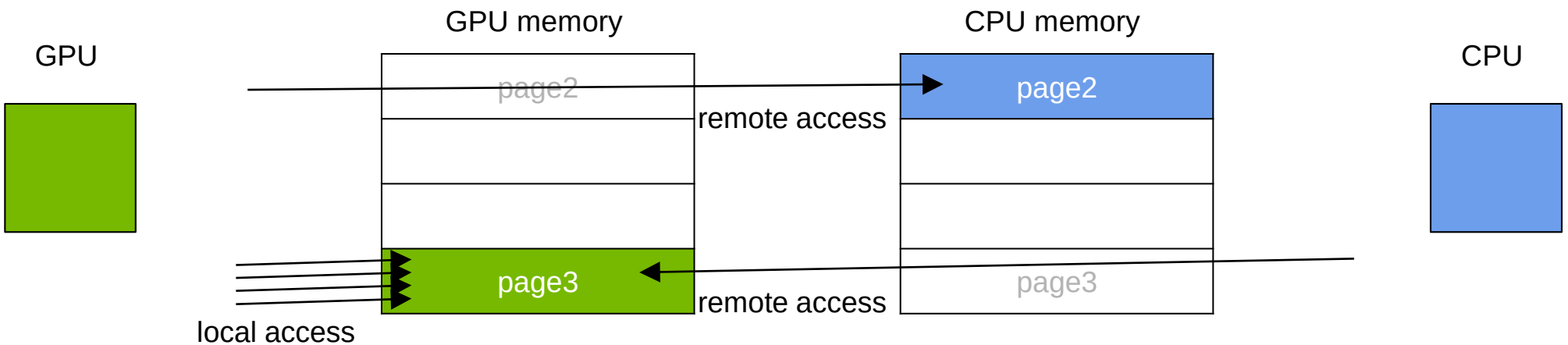
On-demand migrations have lower BW due to page fault overheads

*Indirect peers have lower BW due to P9 interconnect path

NVLINK2: COHERENCY

Works for cudaMallocManaged and malloc

CPU can directly access and cache GPU memory; *native* CPU-GPU atomics

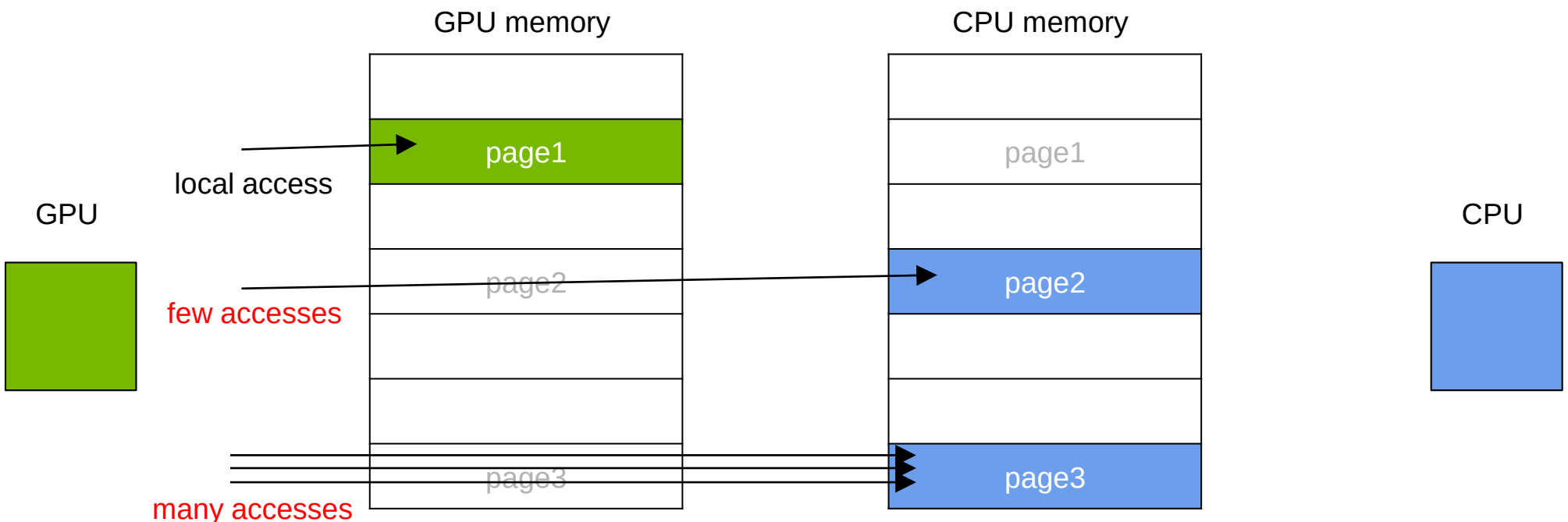


page2 maps to CPU physical memory, page3 maps to GPU physical memory

ACCESS COUNTERS

Opt-in and only for cudaMallocManaged at acceptance

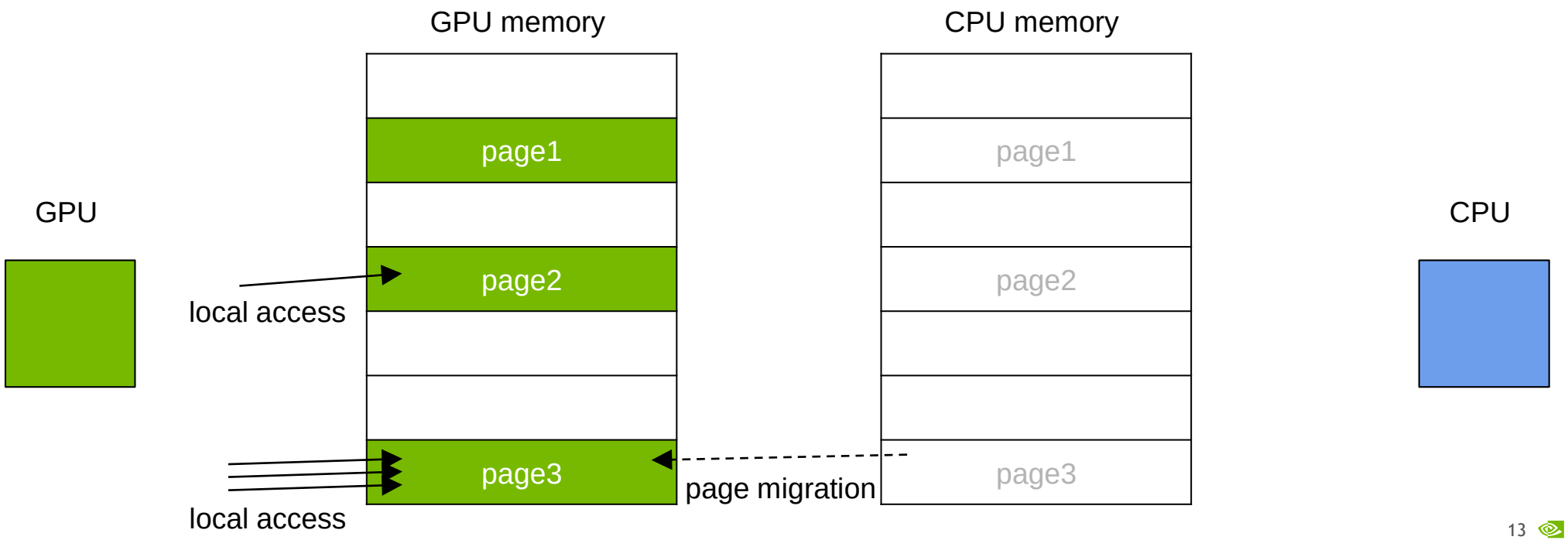
If memory has AccessedBy set, migration can be triggered by access counters (opt-in)



FAULTS VS ACCESS COUNTERS

Faults: eager migration

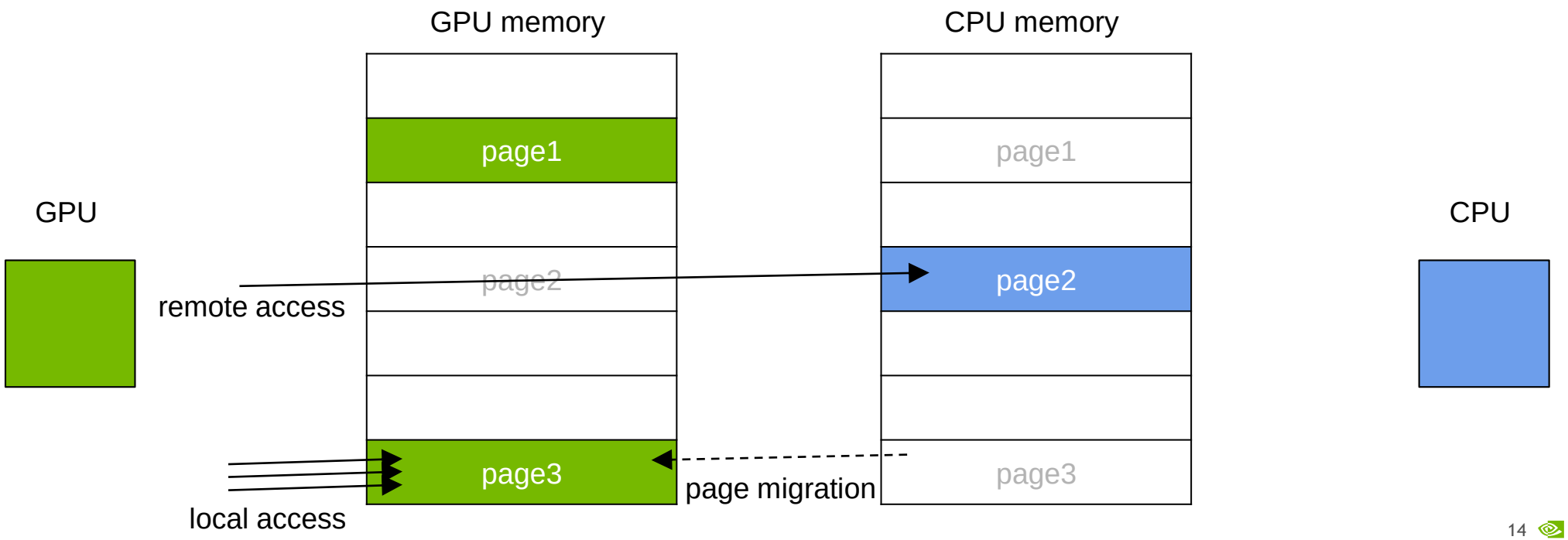
When using faults all touched pages will be moved to the GPU (*eager migration*)



FAULTS VS ACCESS COUNTERS

Access counters: delayed migration

With access counters **only hot pages** will be moved to the GPU (*delayed migration*)



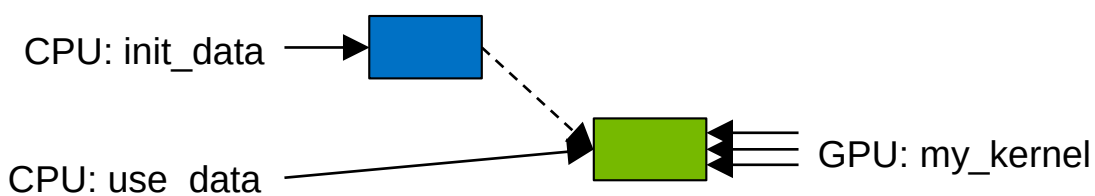
USER HINTS ON P9+V100

Preferred Location: 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);
```

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

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



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

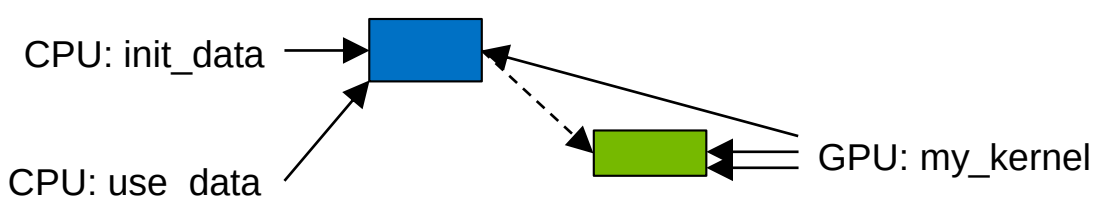
USER HINTS ON P9+V100

Accessed By: using access counters on P9+V100

```
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



on non P9+V100 systems **all** pages will stay in CPU memory

cudaMallocManaged vs malloc (on the system at acceptance)

FIRST TOUCH

cudaMallocManaged: same behavior as x86

```
ptr = cudaMallocManaged(size);
```

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

GPU page faults
Unified Memory driver allocates on GPU
GPU accesses GPU memory

Note: you may alter this behavior by using perf hints,
e.g. PreferredLocation=CPU will allocate on CPU on first touch

FIRST TOUCH

malloc: always allocate on CPU

```
ptr = malloc(size);
```

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

GPU uses ATS, faults
OS allocates on CPU (**by default**)
GPU uses ATS to access CPU memory

ON-DEMAND MIGRATION

cudaMallocManaged: same behavior as x86

```
ptr = cudaMallocManaged(size);

fillData(ptr, size);


doStuffOnGpu<<<...>>>(ptr, size);    // ptr migrates to GPU

cudaDeviceSynchronize();


doStuffOnCpu(ptr, size);             // ptr migrates to CPU
```

ON-DEMAND MIGRATION

malloc: no automatic migrations*

```
ptr = malloc(size);

fillData(ptr, size);


doStuffOnGpu<<<...>>>(ptr, size);    // GPU accesses CPU memory through ATS

cudaDeviceSynchronize();


doStuffOnCpu(ptr, size);             // CPU accesses CPU memory
```

No on-demand malloc data movement except by APIs

*In the future we'll use access counters to migrate malloc memory (**not at acceptance time**)

USER-DIRECTED MIGRATION

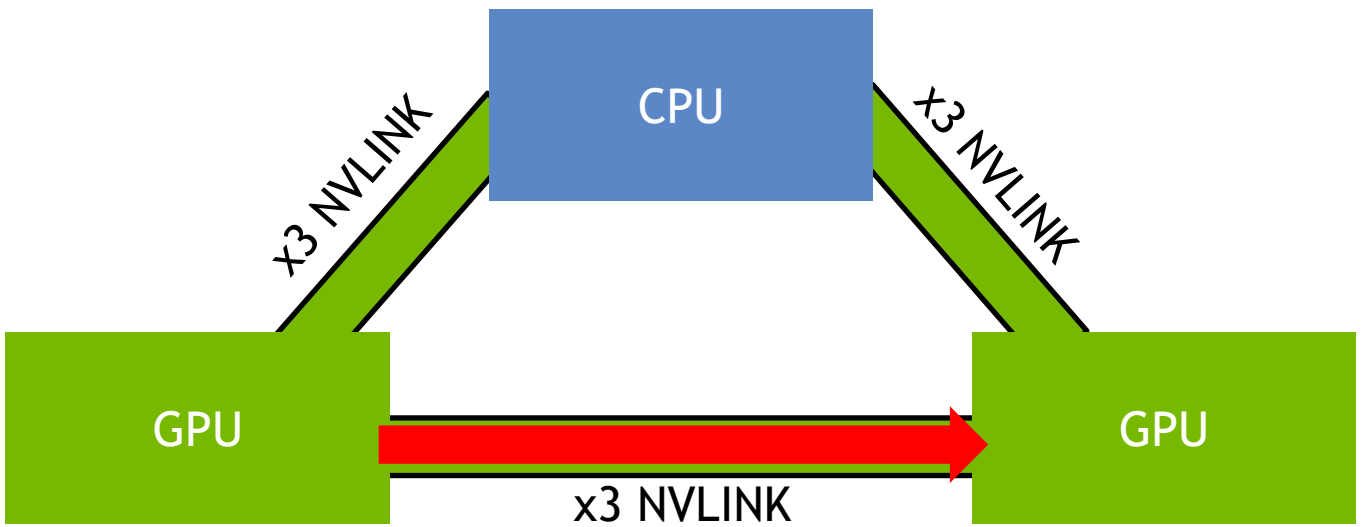
Works for all allocators

```
ptr = malloc_support ? malloc(size) : cudaMallocManaged(size);

fillData(ptr, size);

cudaMemPrefetchAsync(ptr, size, dest_gpu_id);       // moves data to GPU

doStuffOnGpu<<<...>>>(ptr, size);                   // GPU accesses GPU memory

cudaMemPrefetchAsync(ptr, size, cudaCpuDeviceId);   // moves data to CPU

cudaDeviceSynchronize();

doStuffOnCpu(ptr, size);                            // CPU accesses CPU memory
```

USER-DIRECTED MIGRATION

Performance considerations

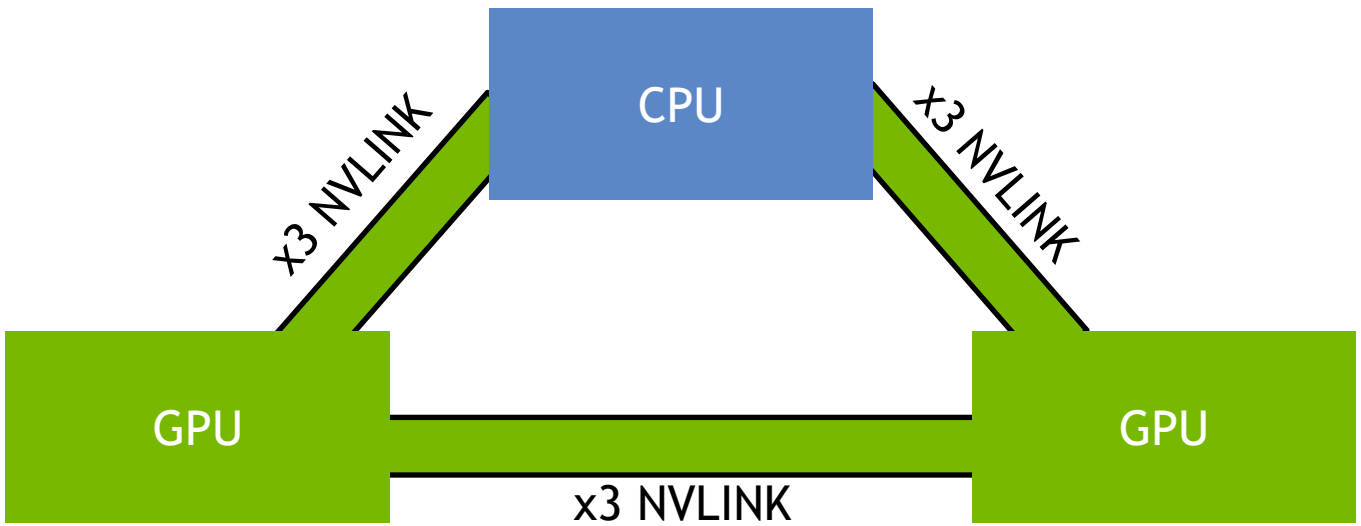
cudaMallocManaged: no performance regressions, increased bandwidth (NVLINK2)



USER-DIRECTED MIGRATION

Performance considerations

malloc: for system acceptance migrations could be implemented through CPU (*slow path*) but we're working on enabling the direct P2P transfer in the future (*fast path*)



EVICTION TABLE

Can [row] evict [column] from GPU to CPU?

	cudaMalloc	cudaMallocManaged	malloc
cudaMalloc	No	Yes	No
cudaMallocManaged	No	Yes	No
malloc	No	No	No

Green: Working as intended

Red: Want to change in future

USER HINTS: ATS

Advise hints have no impact on malloc allocations

When pages are populated they are automatically AccessedBy all GPUs in the system

PreferredLocation or ReadMostly are no-ops for malloc at acceptance

- ReadMostly does not work *by design* (ATS uses single page table)
- PreferredLocation will work in the future

GENERAL RECOMMENDATIONS

Take advantage of CPU access to GPU memory (including native atomics) so that data that is occasionally read by the CPU does not need to move from GPU

cudaMallocManaged is the most performant way to use Unified Memory *now*

Use perf hints to obtain more predictable memory access patterns and data movement behavior between CPU and GPU (only for cudaMallocManaged)

When using system malloc consider current performance implications (no on-demand migrations, possibly low prefetch bandwidth)

Use pooled allocator to alleviate cost of memory allocation

LEARN MORE AT GTC 2018

S8430 - Everything You Need to Know About Unified Memory

Tuesday, Mar 27, 4:30 PM - 5:20 PM – Room 211A

Look out for a “Connect with the Experts” session on Unified Memory