Lecture 8: Virtual Memory

Programming Massively Parallel Multiprocessors and Heterogeneous Systems (Understanding and programming the devices powering AI)

Jonathan Appavoo

CPU-GPU Memory Transfers

So far: explicit copying:

```
cudaMalloc(…)
cudaMemcpy( … )
```

Alternatives:

- "Zero Copy": CUDA 2.0+
- Unified Virtual Address Space: CUDA 4.0+
- Unified memory: CUDA 6.0+
- Managed memory: CUDA 8.0+
- Unified Memory + Hetrogenous Memory Management (HMM): CUDA 12.2 +

Note: confusion around terms (primarily due to marketing hype)

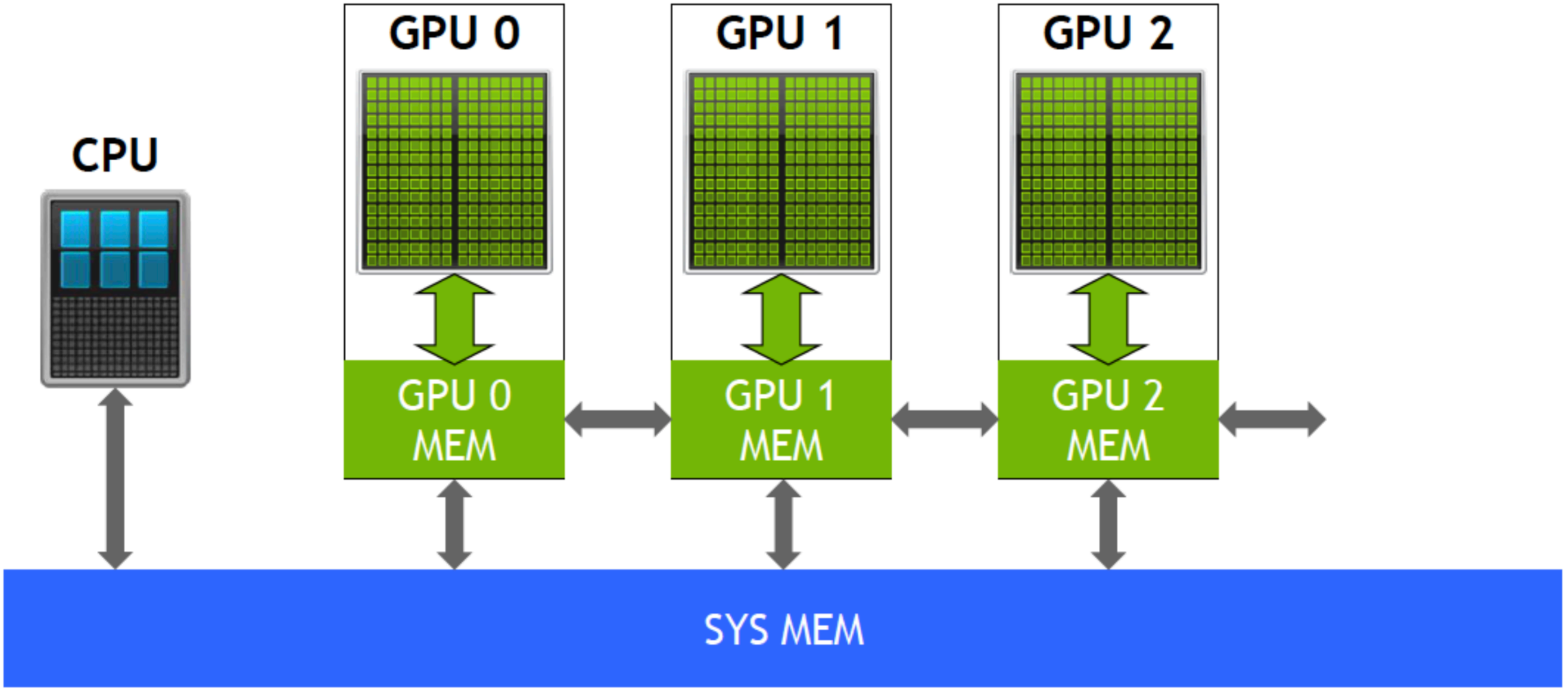
Unified Memory: the real thing (sort of)

- Allocate memory using `cudaMallocManaged()`
  - CUDA runtime takes care of transfers
  - No need to call `cudaMemcpy()` anymore
  - Behaves like you have a single memory space

- Coherence is software managed
  - ➔ can be slow
  - ➔ often better to still use `cudaMemcpy()`

- Requires CUDA 6.0+ and 64-bit OS

Unified Memory: CUDA 6.0+



For memory allocated with cudaMallocManaged()

Unified Memory: CUDA 6.0+

```cpp
__global__ void write_value(int* ptr, int v) {
  *ptr = v;
}

int main() {
  int* d_ptr = nullptr;
  // Does not require any unified memory support
  cudaMalloc(&d_ptr, sizeof(int));
  write_value<<<1, 1>>>(d_ptr, 1);
  int h_value;
  // Copy memory back to the host and synchronize
  cudaMemcpy(&h_value, d_ptr, sizeof(int),
             cudaMemcpyDefault);
  printf("value = %d\n", h_value);
  cudaFree(d_ptr);
  return 0;
}
```

```cpp
__global__ void write_value(int* ptr, int v) {
  *ptr = v;
}

int main() {
  int* ptr = nullptr;
  // Requires CUDA Managed Memory support
  cudaMallocManaged(&ptr, sizeof(int));
  write_value<<<1, 1>>>(ptr, 1);
  // Synchronize required
  // (before, cudaMemcpy was synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", *ptr);
  cudaFree(ptr);
  return 0;
}
```

Or even less code

```cpp
__global__ void write_value(int* ptr, int v) {
  *ptr = v;
}

int main() {
  int* d_ptr = nullptr;
  // Does not require any unified memory support
  cudaMalloc(&d_ptr, sizeof(int));
  write_value<<<1, 1>>>(d_ptr, 1);
  int h_value;
  // Copy memory back to the host and synchronize
  cudaMemcpy(&h_value, d_ptr, sizeof(int),
             cudaMemcpyDefault);
  printf("value = %d\n", h_value);
  cudaFree(d_ptr);
  return 0;
}
```

```cpp
__global__ void write_value(int* ptr, int v) {
  *ptr = v;
}

// Requires CUDA Managed Memory support
__managed__ int value;

int main() {
  write_value<<<1, 1>>>(&value, 1);
  // Synchronize required
  // (before, cudaMemcpy was synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", value);
  return 0;
}
```

Or even less code

```cpp
__global__ void write_value(int* ptr, int v) {
  *ptr = v;
}

int main() {
  int* d_ptr = nullptr;
  // Does not require any unified memory support
  cudaMalloc(&d_ptr, sizeof(int));
  write_value<<<1, 1>>>(d_ptr, 1);
  int h_value;
  // Copy memory back to the host and synchronize
  cudaMemcpy(&h_value, d_ptr, sizeof(int),
             cudaMemcpyDefault);
  printf("value = %d\n", h_value);
  cud
  ret
}
```

```cpp
__global__ void write_value(int* ptr, int v) {
  *ptr = v;
}

// Requires CUDA Managed Memory support
__managed__ int value;

int main() {
  write_value<<<1, 1>>>(&value, 1);
  // Synchronize required
  // (before, cudaMemcpy was synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", value);
  return 0;
```

- Data is copied to GPU automagically before kernel starts
- C data is copied back to CPU automagically when accessed on CPU
- CPU cannot access ”managed” data while kernel is running; segmentation fault otherwise.

UM: CUDA 12.2+ and the right HW and OS

Table 31: Overview of levels of unified memory support

Unified Memory Support Level	System device properties	Further documentation
Full CUDA Unified Memory: all memory has full support. This includes System-Allocated and CUDA Managed Memory.	Set to 1: `pageableMemoryAccess` Systems with hardware acceleration also have the following properties set to 1: `hostNativeAtomicSupported`, `pageableMemoryAccessUsesHostPageTables`, `directManagedMemAccessFromHost`	Unified Memory on devices with full CUDA Unified Memory support
Only CUDA Managed Memory has full support.	Set to 1: `concurrentManagedAccess` Set to 0: `pageableMemoryAccess`	Unified Memory on devices with only CUDA Managed Memory support
CUDA Managed Memory without full support: unified addressing but no concurrent access.	Set to 1: `managedMemory` Set to 0: `concurrentManagedAccess`	Unified Memory on Windows or devices with compute capability 5.x CUDA for Tegra Memory Management Unified Memory on Tegra
No Unified Memory support.	Set to 0: `managedMemory`	CUDA for Tegra Memory Management

UM: CUDA 12.2+ and the right HW and OS

Table 31: Overview of levels of unified memory support

Unified Memory Support Level	System device properties	Further documentation
Full CUDA Unified Memory: all memory has full support. This includes System-Allocated and CUDA Ma	Set to 1: `pageableMemoryAccess` Systems with hardware acceleration also	Unified Memory on devices with full CUDA Unified Memory support
Only CUDA Managed support.		on devices with only Memory support
CUDA Managed Mem support: unified addr concurrent access.		on Windows or devices pability 5.x Memory Management Unified Memory on Tegra
No Unified Memory support.	Set to 0: `managedMemory`	CUDA for Tegra Memory Management

CUDA has a spectrum of functionality with respect to what you can do:
Depends on your HW, CUDA version, Driver Version, OS version
Eg. V100 supports has pagetable support but our Linux Kernel version and Driver does not :-(

Let's look at what full support looks like

```
__global__ void write_value(int* ptr, int v) {
  *ptr = v;
}

int main() {
  // Requires System-Allocated Memory support
  int* ptr = (int*)malloc(sizeof(int));
  write_value<<<1, 1>>>(ptr, 1);
  // Synchronize required
  // (before, cudaMemcpy was synchronizing)
  cudaDeviceSynchronize();
  printf("value = %d\n", *ptr);
  free(ptr);
  return 0;
}
```

More examples of what is now possible

```
__global__ void kernel(const char* type, const char* data) {
  static const int n_char = 8;
  printf("%s - first %d characters: '", type, n_char);
  for (int i = 0; i < n_char; ++i) printf("%c", data[i]);
  printf("'\n");
}
```

```
void test_stack() {
  const char test_string[] = "Hello World";
  kernel<<<1, 1>>>("stack", test_string);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

More examples of what is now possible

```
__global__ void kernel(const char* type, const char* data) {
  static const int n_char = 8;
  printf("%s - first %d characters: '", type, n_char);
  for (int i = 0; i < n_char; ++i) printf("%c", data[i]);
  printf("'\n");
}
```

```
void test_static() {
  static const char test_string[] = "Hello World";
  kernel<<<1, 1>>>("static", test_string);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

More examples of what is now possible

```
__global__ void kernel(const char* type, const char* data) {
  static const int n_char = 8;
  printf("%s - first %d characters: '", type, n_char);
  for (int i = 0; i < n_char; ++i) printf("%c", data[i]);
  printf("'\n");
}
```

```
const char global_string[] = "Hello World";
```

```
void test_global() {
  kernel<<<1, 1>>>("global", global_string);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

More examples of what is now possible

```
__global__ void kernel(const char* type, const char* data) {
  static const int n_char = 8;
  printf("%s - first %d characters: '", type, n_char);
  for (int i = 0; i < n_char; ++i) printf("%c", data[i]);
  printf("'\n");
}
```

```
// declared in separate file, see below
extern char* ext_data;

void test_extern() {
  kernel<<<1, 1>>>("extern", ext_data);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

But perhaps more importantly

```
__global__ void kernel(const char* type, const char* data) {
  static const int n_char = 8;
  printf("%s - first %d characters: '", type, n_char);
  for (int i = 0; i < n_char; ++i) printf("%c", data[i]);
  printf("'\n");
}
```

```
void test_file_backed() {
  int fd = open(INPUT_FILE_NAME, O_RDONLY);
  ASSERT(fd >= 0, "Invalid file handle");
  struct stat file_stat;
  int status = fstat(fd, &file_stat);
  ASSERT(status >= 0, "Invalid file stats");
  char* mapped = (char*)mmap(0, file_stat.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
  ASSERT(mapped != MAP_FAILED, "Cannot map file into memory");
  kernel<<<1, 1>>>("file-backed", mapped);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
  ASSERT(munmap(mapped, file_stat.st_size) == 0, "Cannot unmap file");
  ASSERT(close(fd) == 0, "Cannot close file");
}
```

And even

```
__global__ void kernel(int* ptr) {
  cuda::atomic_ref{*ptr}.store(2);
}
```

```
// initialize the value in our file-backed memory
*ptr = 1;
printf("Atom value: %d\n", *ptr);

// device and host thread access ptr concurrently, using cuda::atomic_ref
kernel<<<1, 1>>>(ptr);
while (cuda::atomic_ref{*ptr}.load() != 2);
// this will always be 2
printf("Atom value: %d\n", *ptr);
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomic-accesses-synchronization-primitives https://docs.nvidia.com/cuda/cuda-c-programming-guide/#um-hw-coherency

- But be aware -- only very high end systems share a page table and syncronization is in HW
- otherwise done via pagefaults

A real benefit : no need for deep copies

Explicit Memory Mgmt

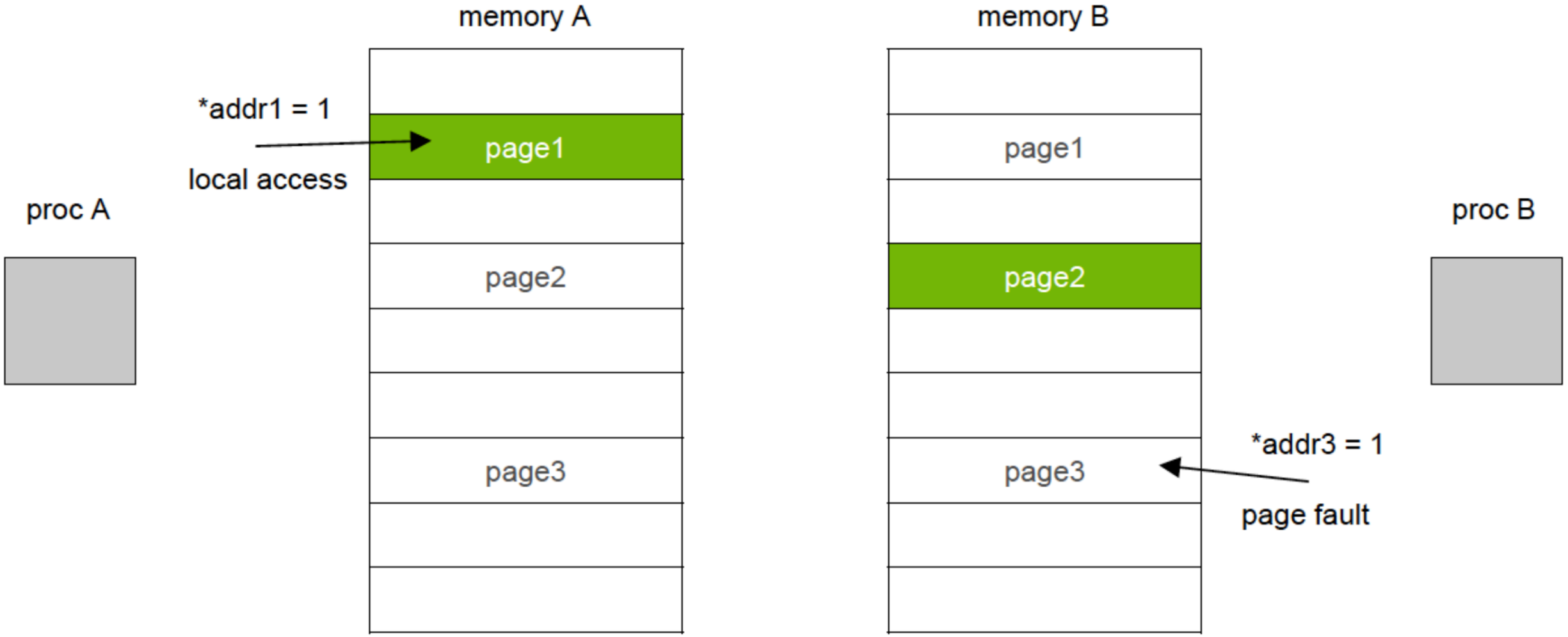
```
char **data;
data = malloc(N*sizeof(char*)) ;
for( int i=0; i<N; i++ )
   data[i] = malloc(X) ;
.. ..
char **d_data ;
cudaMalloc( &d_data, N*sizeof(char*) ;
char **h_data = malloc( N*sizeof(char*) ;
for( int i=0; i<N; i++ ) {
   cudaMalloc( &h_data[i], X ) ;
   cudaMemcpy( h_data[i], data[i], X,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

gpu_fct<<<. . .>>>(d_data, N)
```

Managed Memory

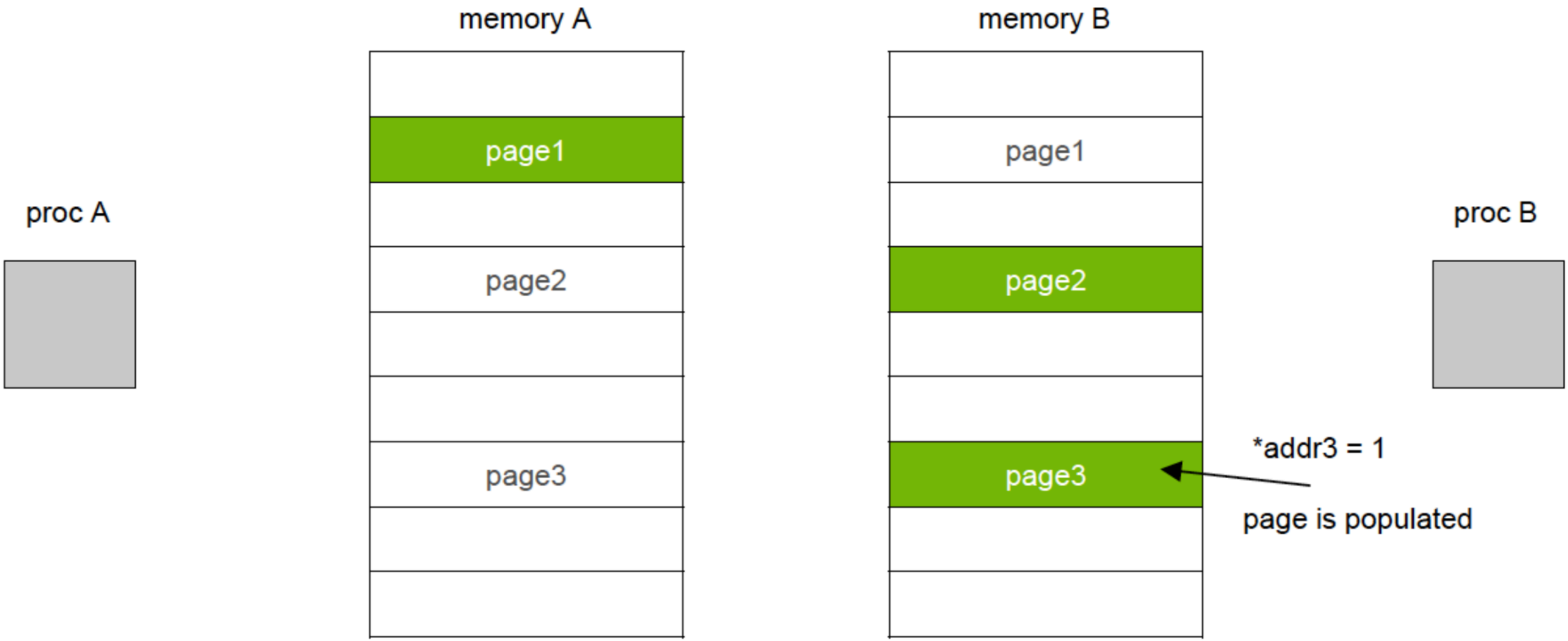
```
char **data;
data = malloc(N*sizeof(char*)) ;
for( int i=0; i<N; i++ )
   data[i] = malloc(X) ;
.. ..
gpu_fct<<<. . .>>>(data, N)
```

Under the covers: Copy on demand I



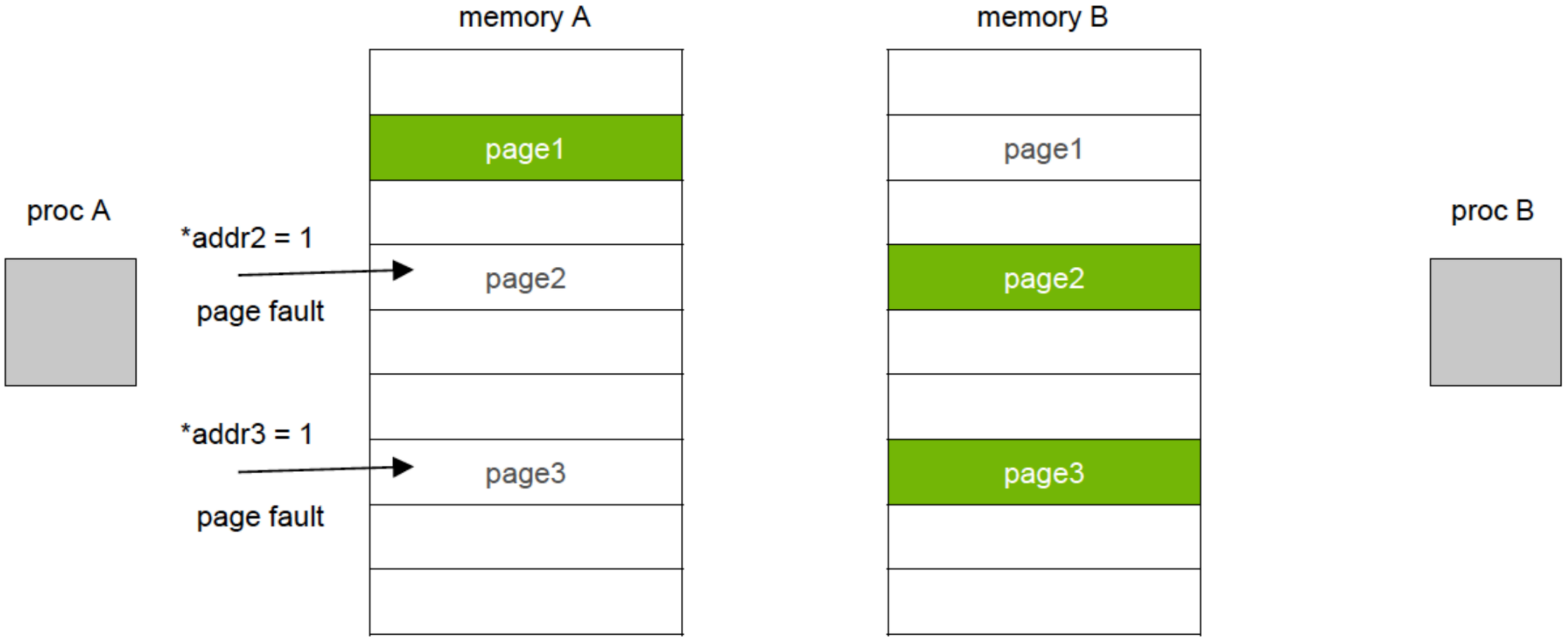
Devices without shared page tables (all but the highend eg. Grace Hopper, Frontier)

Under the Covers: Copy on demand II



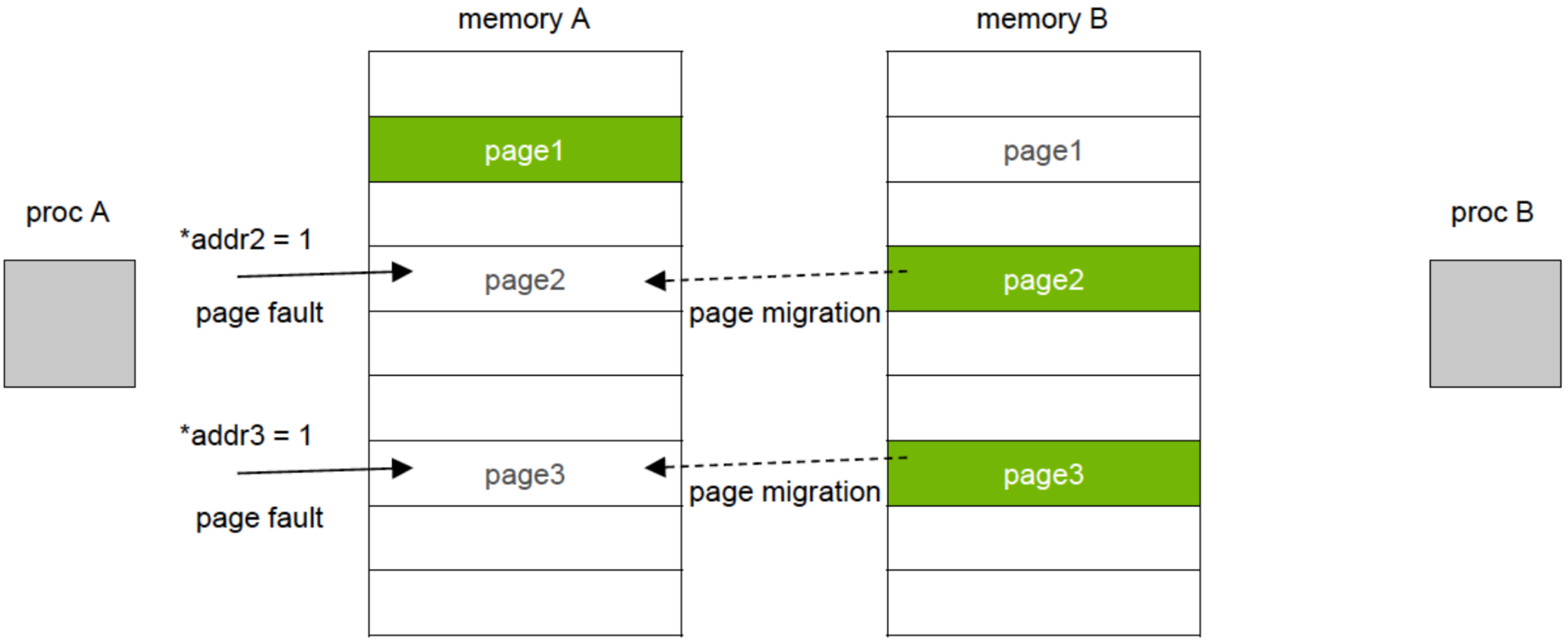
Devices without shared page tables (all but the highend eg. Grace Hopper, Frontier)

Under the Covers: Copy on demand III



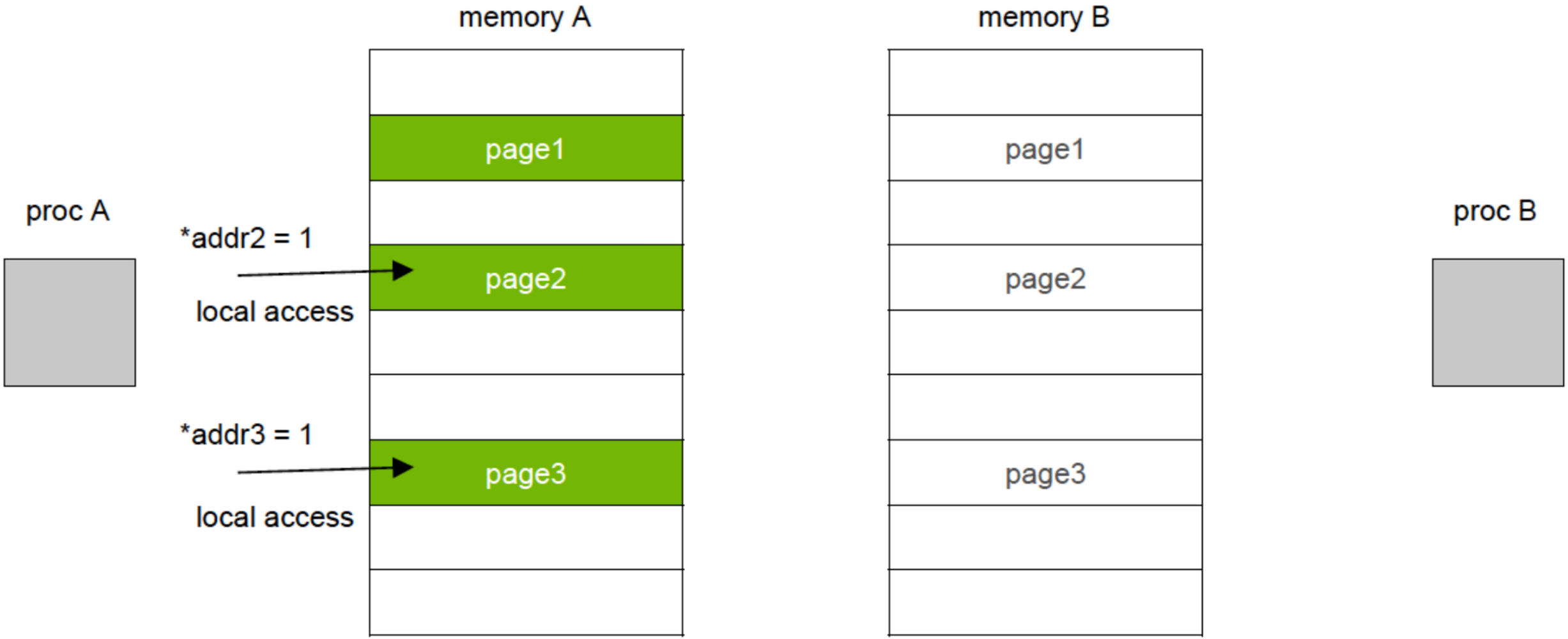
Devices without shared page tables (all but the highend eg. Grace Hopper, Frontier)

Under the Covers: Copy on demand IV



Devices without shared page tables (all but the highend eg. Grace Hopper, Frontier)

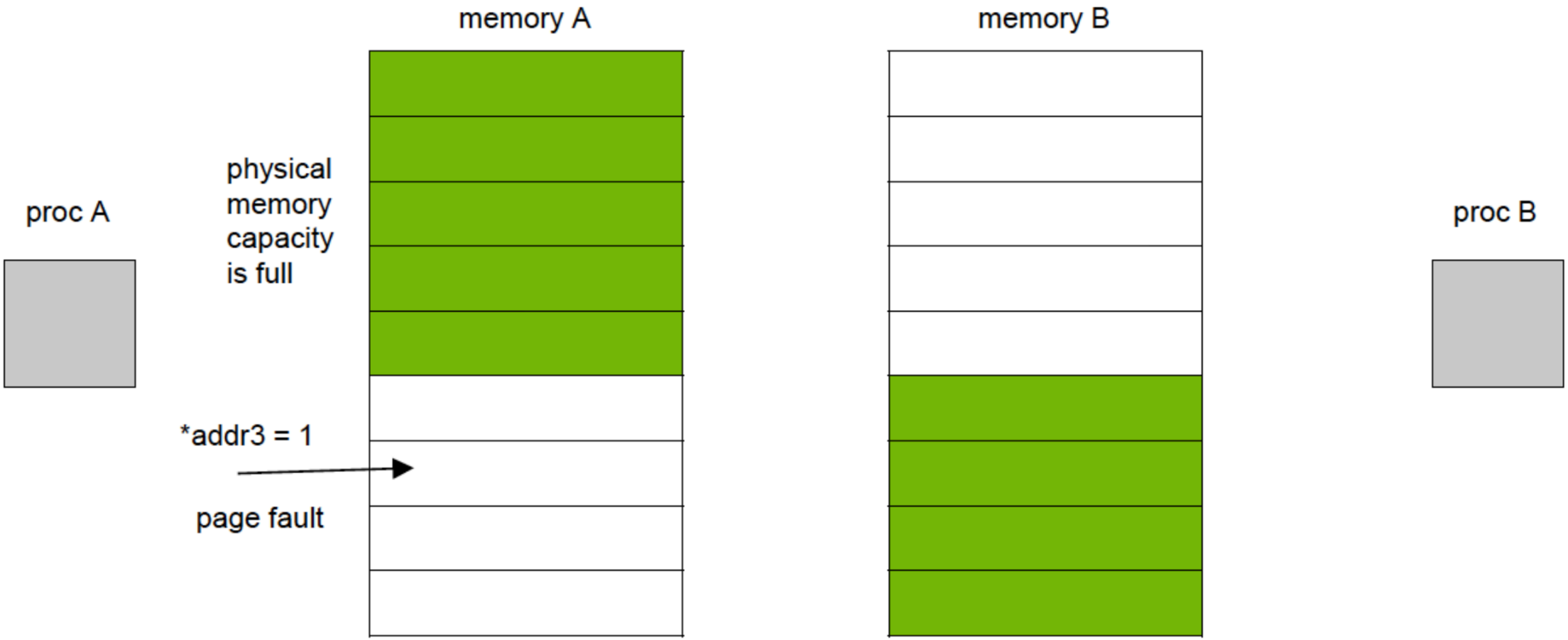
Under the Covers: Copy on demand V



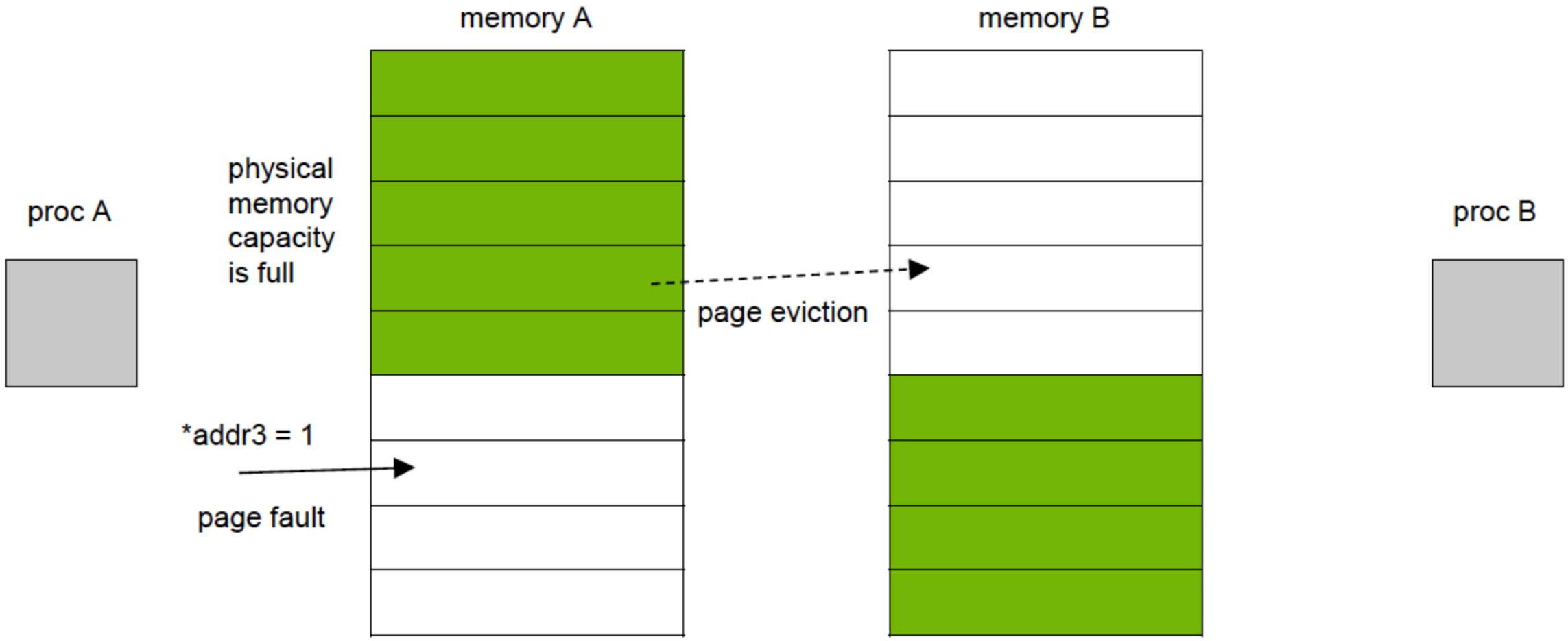
Devices without shared page tables (all but the highend eg. Grace Hopper, Frontier)

Under the Covers: Page Eviction I

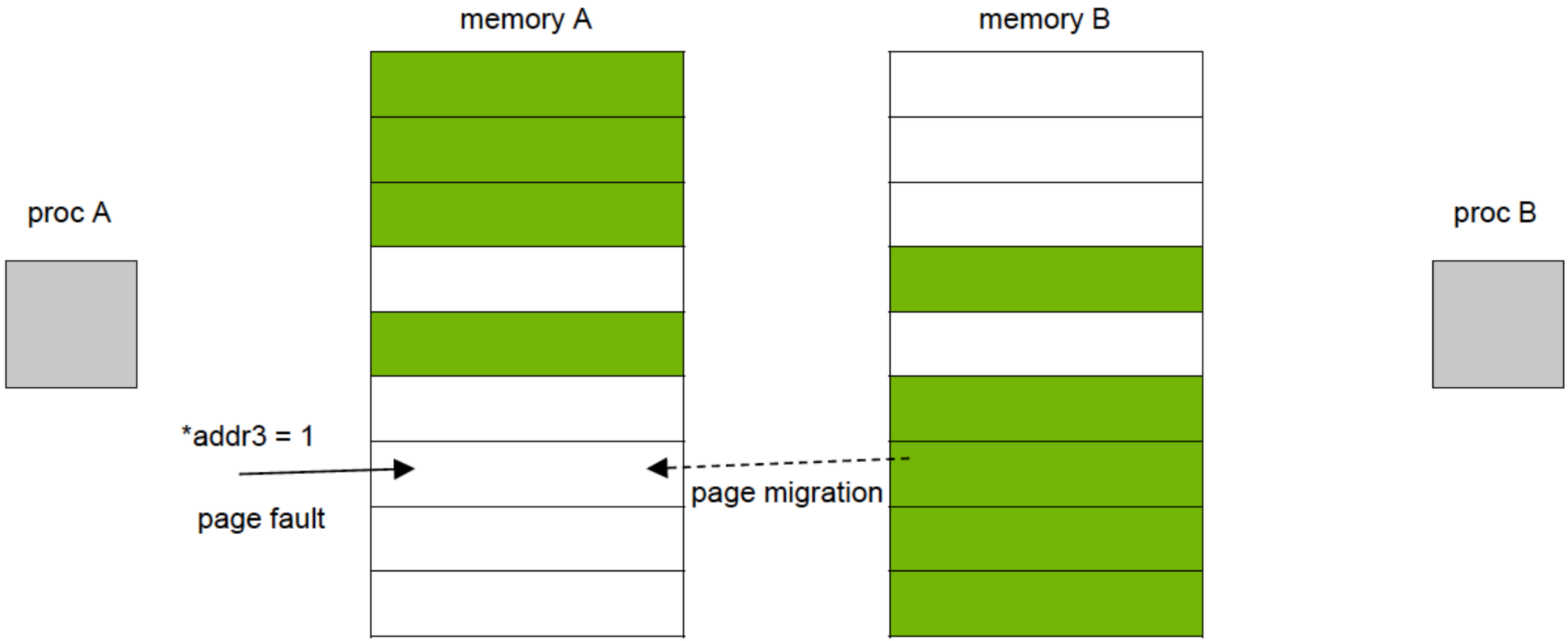
- When memory oversubscribed



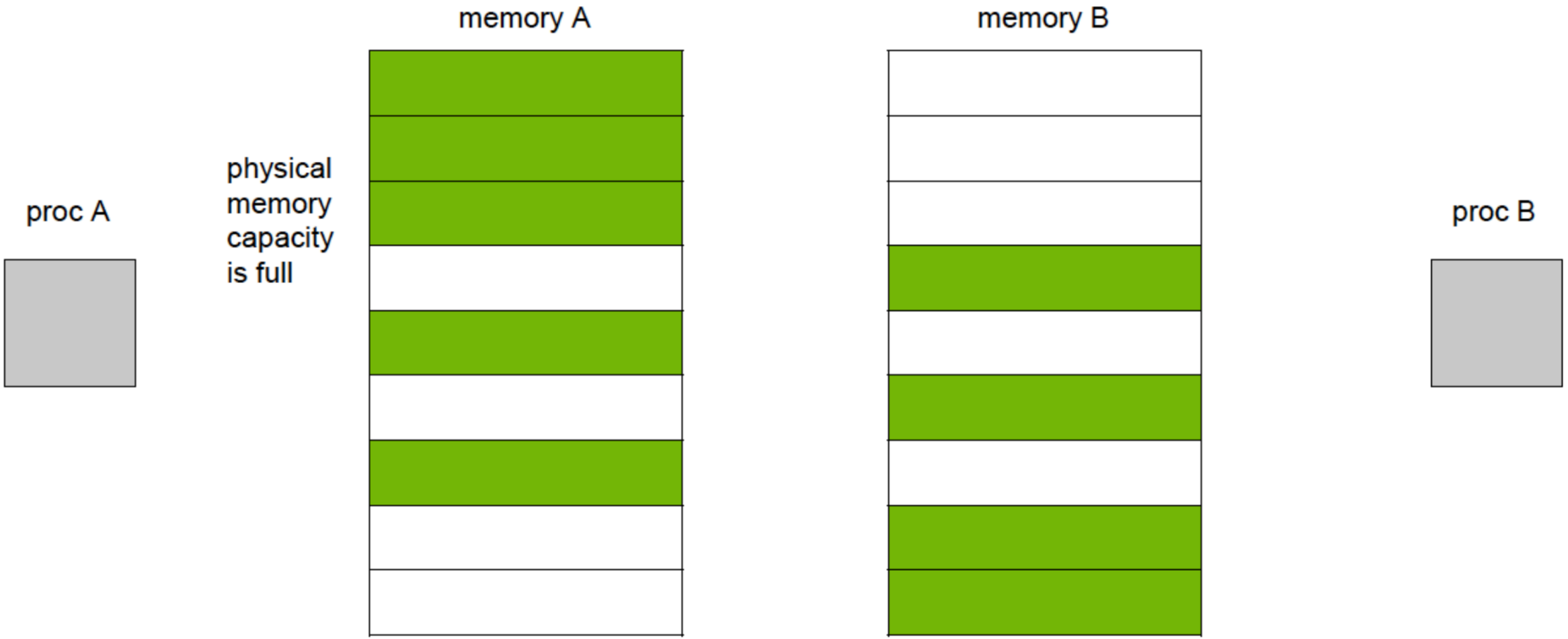
Page Eviction II



Under the Covers: Page Eviction III

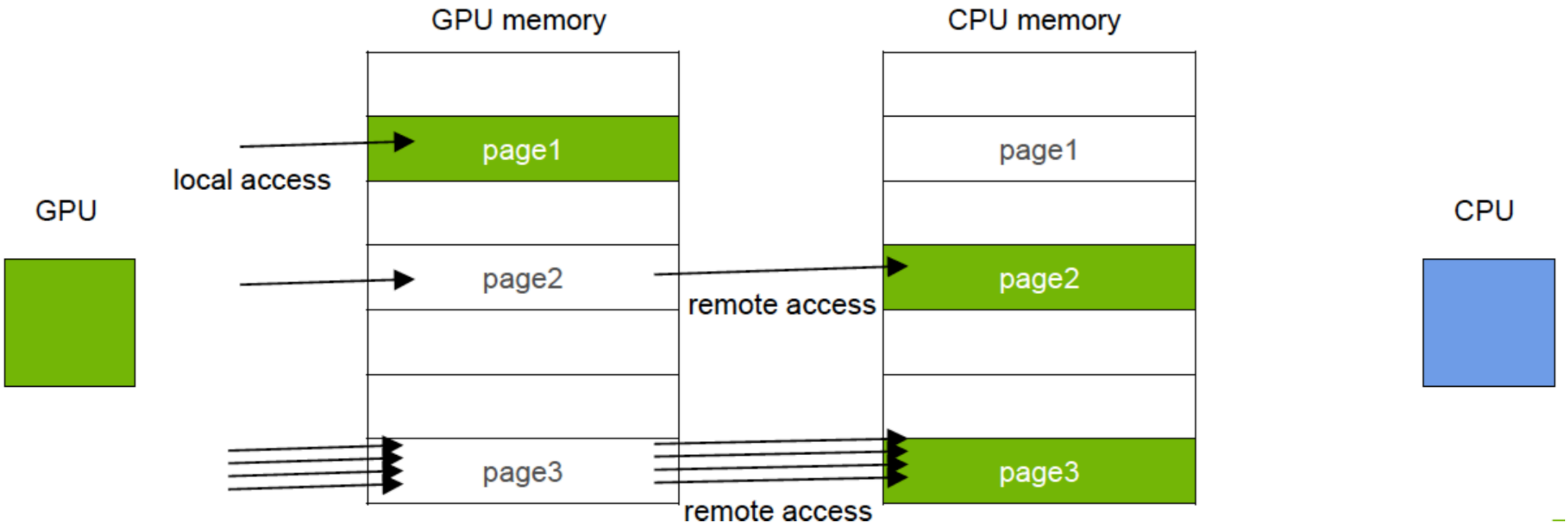


Under the Covers: Page Eviction IV

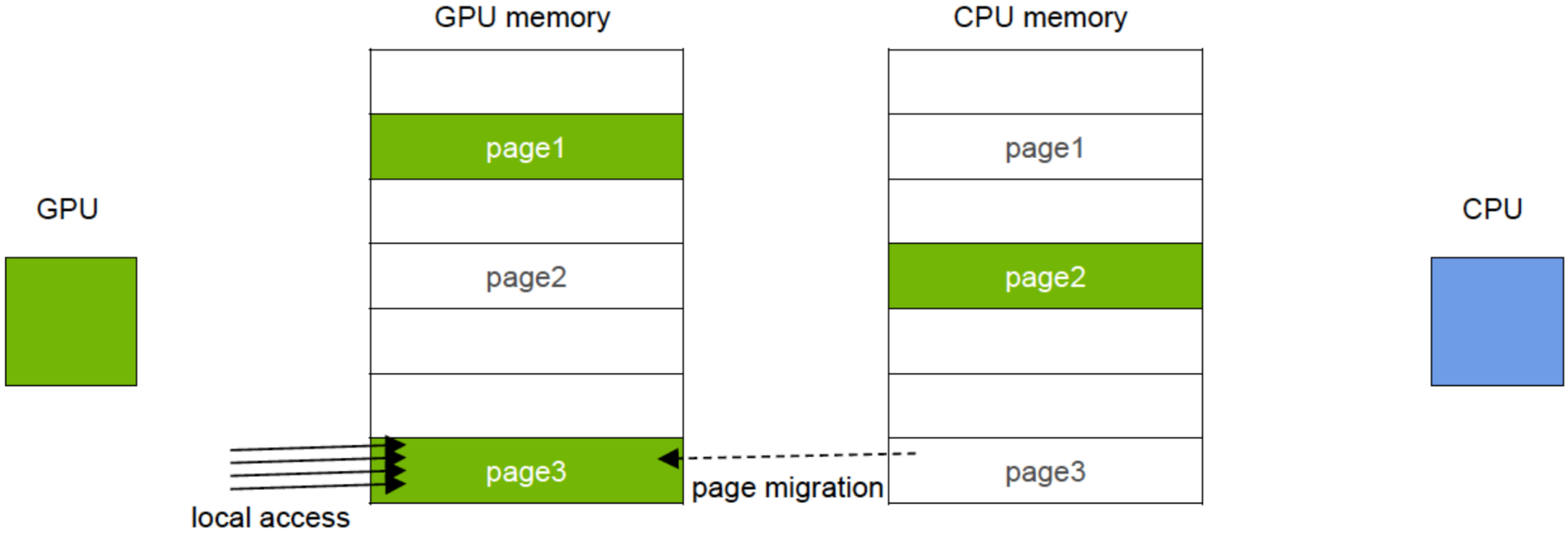


Volta Access counters

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



Volta Access counters



Under the covers

- The driver does intelligent things under the covers:
  - Prefetching to reduce # faults
  - Heuristics to avoid frequent migration
  - ”smart” evictions (LRU)
- You cannot control these,
  but can affect them through hints

- Hints:
  - `cudaMemPrefetchAsync( ptr, size, proc, stream )`
  - `cudaMemAdvise( ptr, size, advice, proc )`
    - read mostly
    - preferred location
    - etc…

Under the covers: Page Sizes

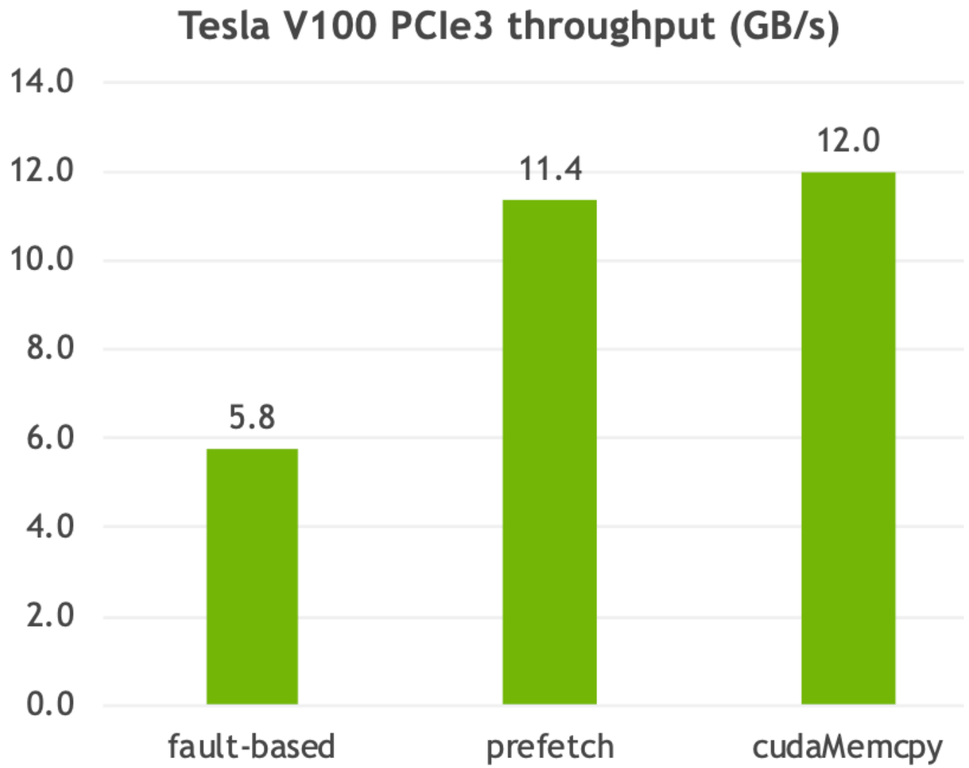
- "NVIDIA GPUs support multiple physical page sizes, but prefer 2MiB physical pages or larger. Note that these sizes are subject to change in future hardware."
- "The default page size of virtual pages usually corresponds to the physical page size,"
- GPU MMU has page table, PTE's, TLBs

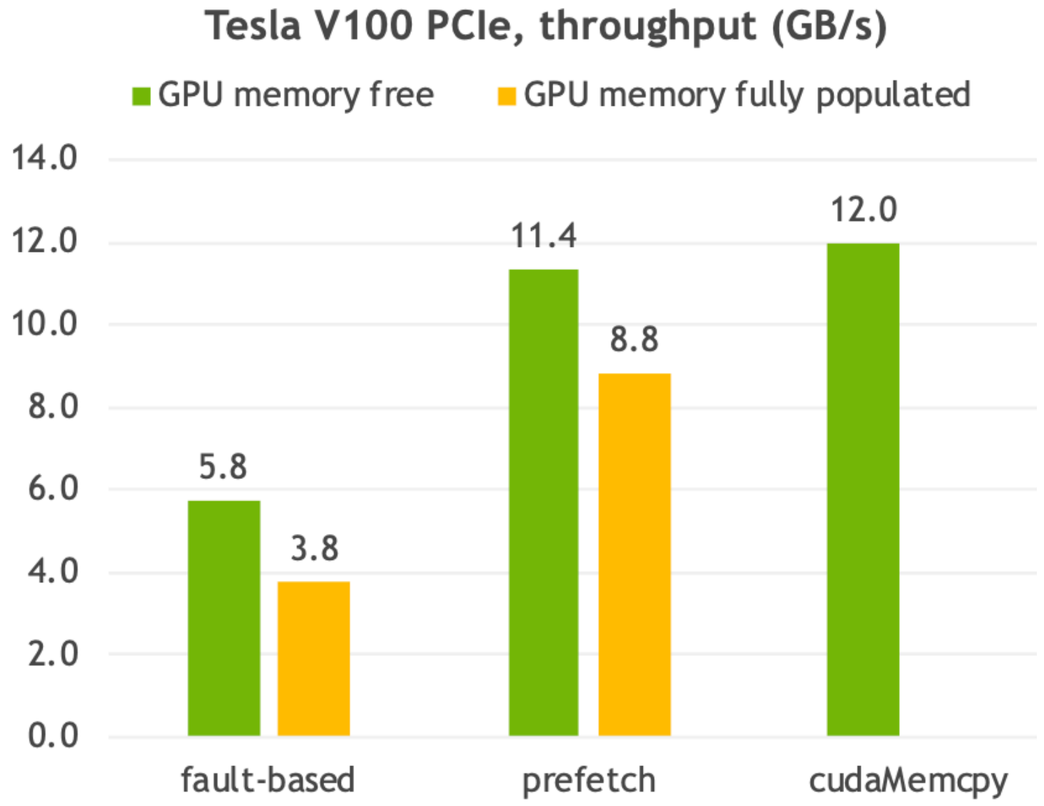
https://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-paging-and-page-sizes

"One important aspect for performance tuning is that TLB misses are generally significantly more expensive on the GPU compared to the CPU. This means that if a GPU thread frequently accesses random locations of Unified Memory mapped using a small enough page size, it might be significantly slower compared to the same accesses to Unified Memory mapped using a large enough page size. While a similar effect might occur for a CPU thread randomly accessing a large area of memory mapped using a small page size, the slowdown is less pronounced, meaning that the application might want to trade-off this slowdown with having less memory fragmentation."

https://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-paging-and-page-sizes

Under the covers: Performance be careful





Under the covers: From the horse's mouth

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

CPU: CUDA memory allocators in light of UM+ HMM

Memory allocators on systems with HMM	Placement	Migratable	Accessible from:		
			CPU	GPU	RDMA
System allocated `malloc`, `mmap`, …	First-touch GPU or CPU	Y	Y	Y	Y
CUDA managed `cudaMallocManaged`		Y	Y	Y	N
CUDA device-only `cudaMalloc`, …	GPU	N	N	Y	Y
CUDA host-pinned `cudaMallocHost`, …	CPU	N	Y	Y	Y

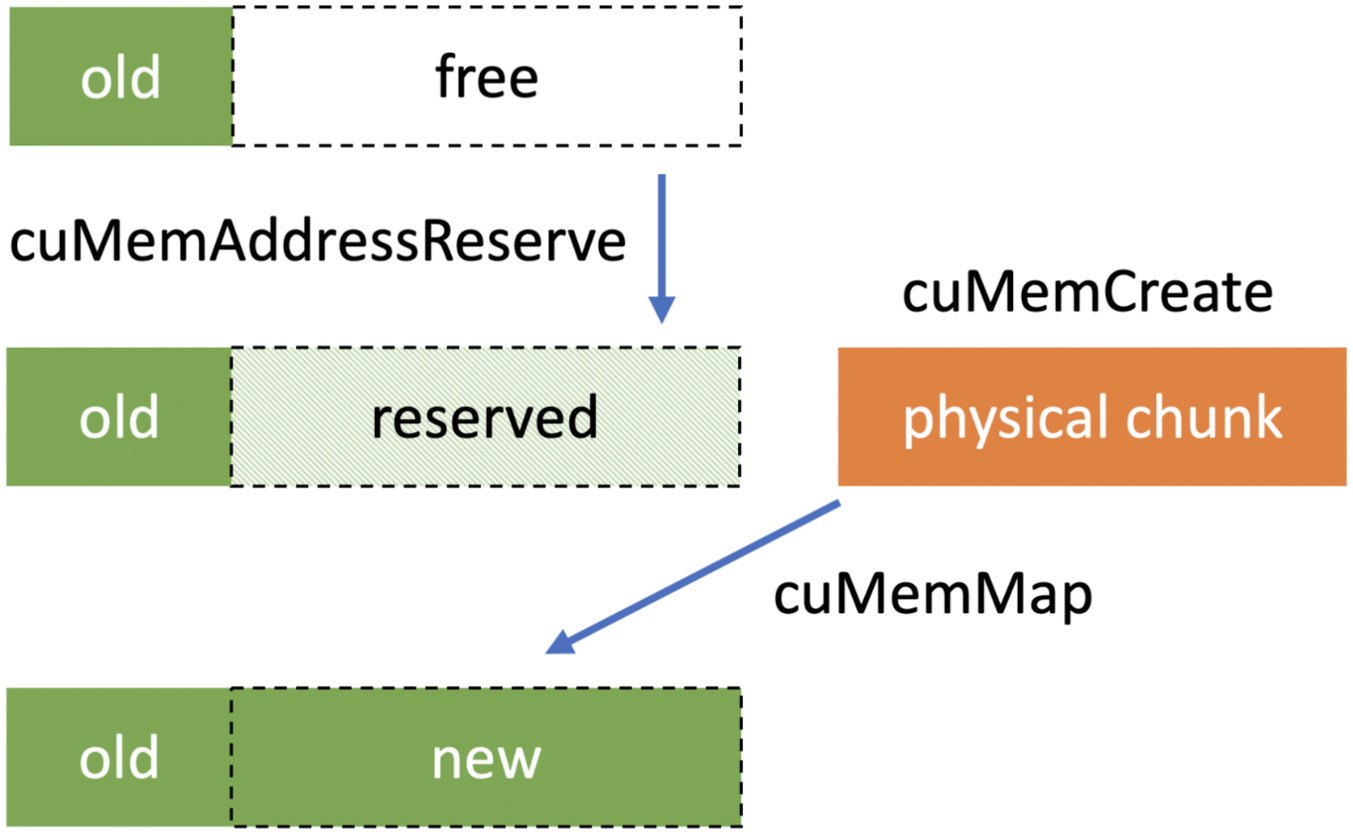
Table 1. Overview of system and CUDA memory allocators on systems with HMM

https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/

Virtual Memory API

- Allocate Physical Memory
- Reserve a VA range
- Mapping allocated memory to the VA range
- Controlling access rights on the mapped ranged

Introducing Low-Level GPU Virtual Memory Management



https://docs.nvidia.com/cuda/cuda-c-programming-guide/#virtual-memory-management

https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html

https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/

References

- https://developer.nvidia.com/gtc/2019/video/S9727/video
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/#unified-memory-programming
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/#virtual-memory-management
- https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/
- https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/
- https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/
- https://github.com/NVIDIA/open-gpu-doc