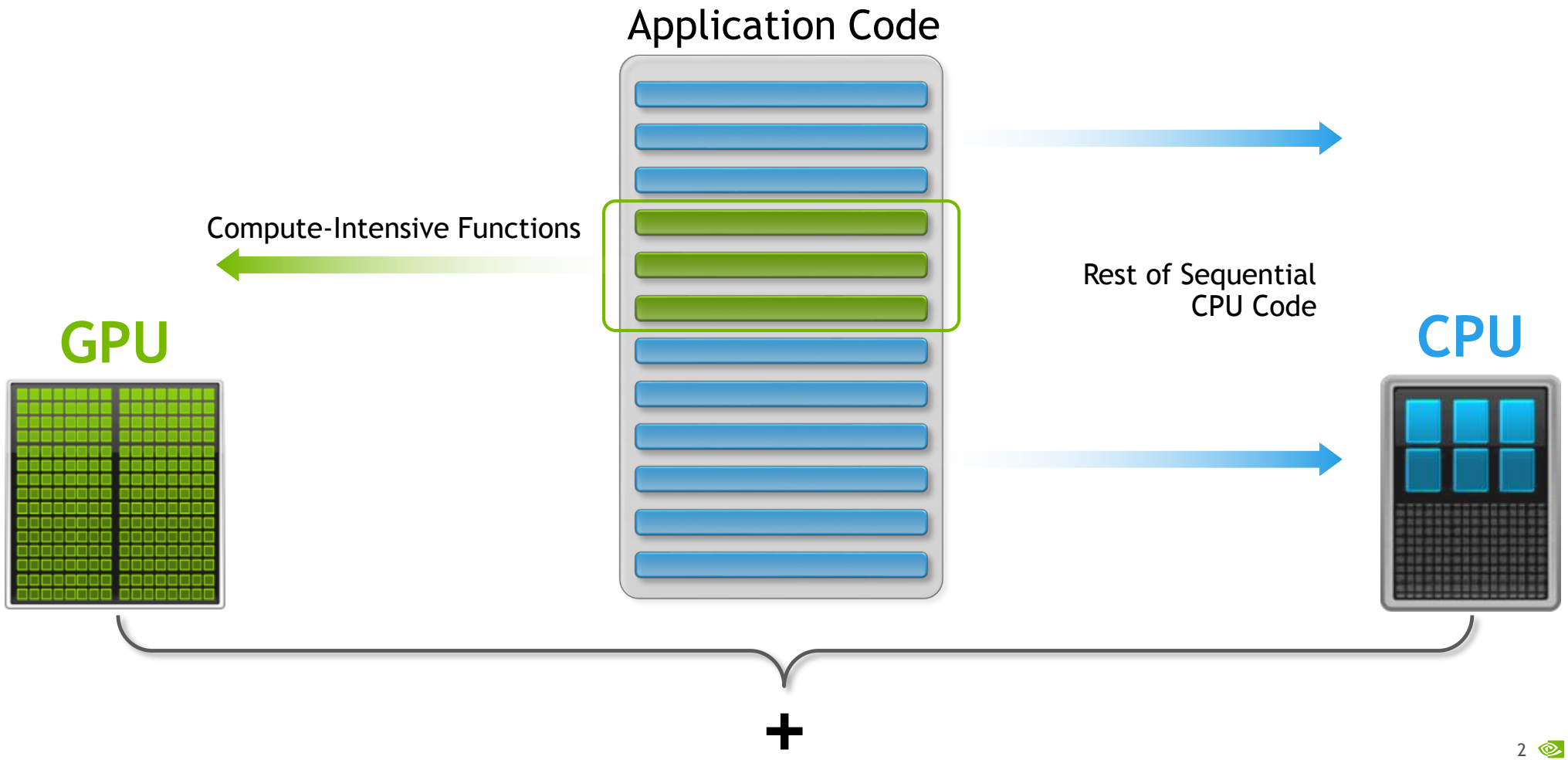
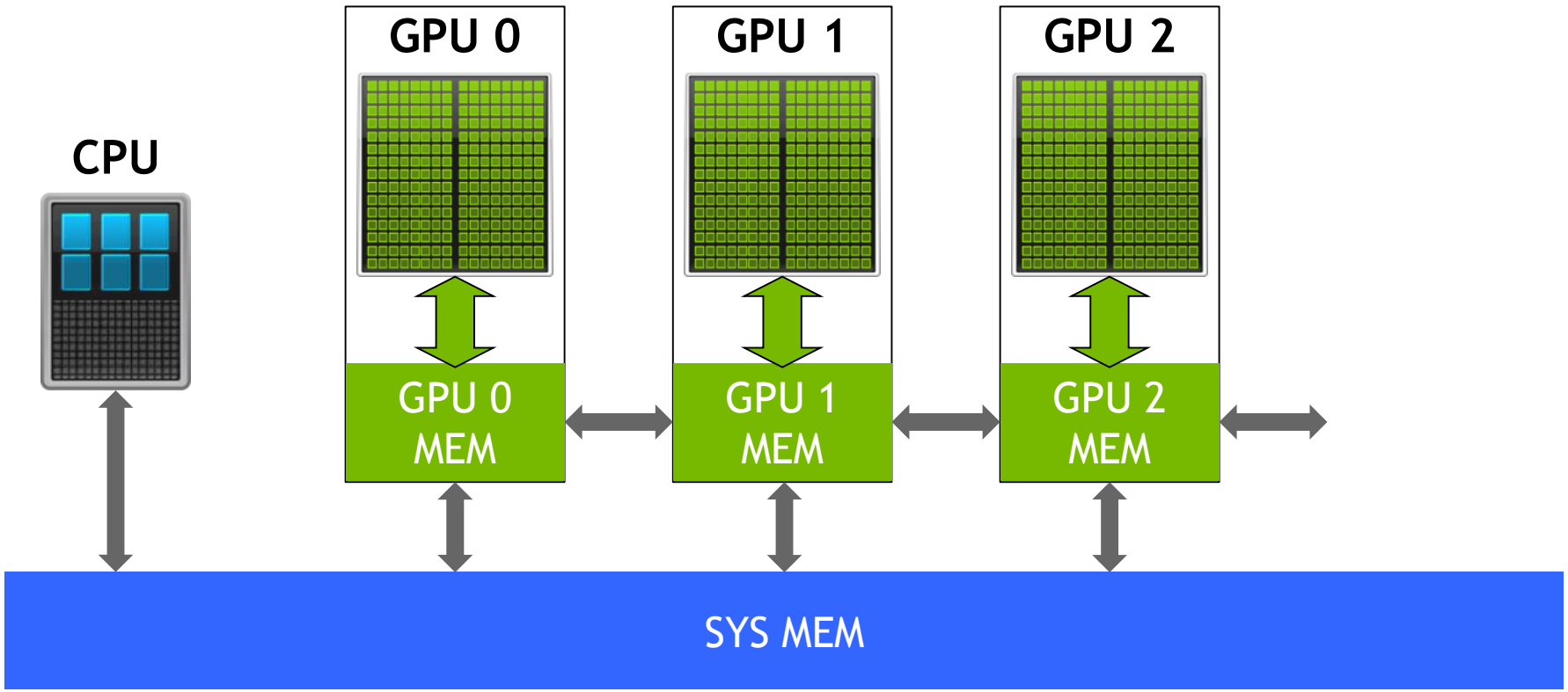
GPU ARCHITECTURES AND NEW PROGRAMMING MODEL FEATURES

Nikolay Sakharnykh, 7/31/2017

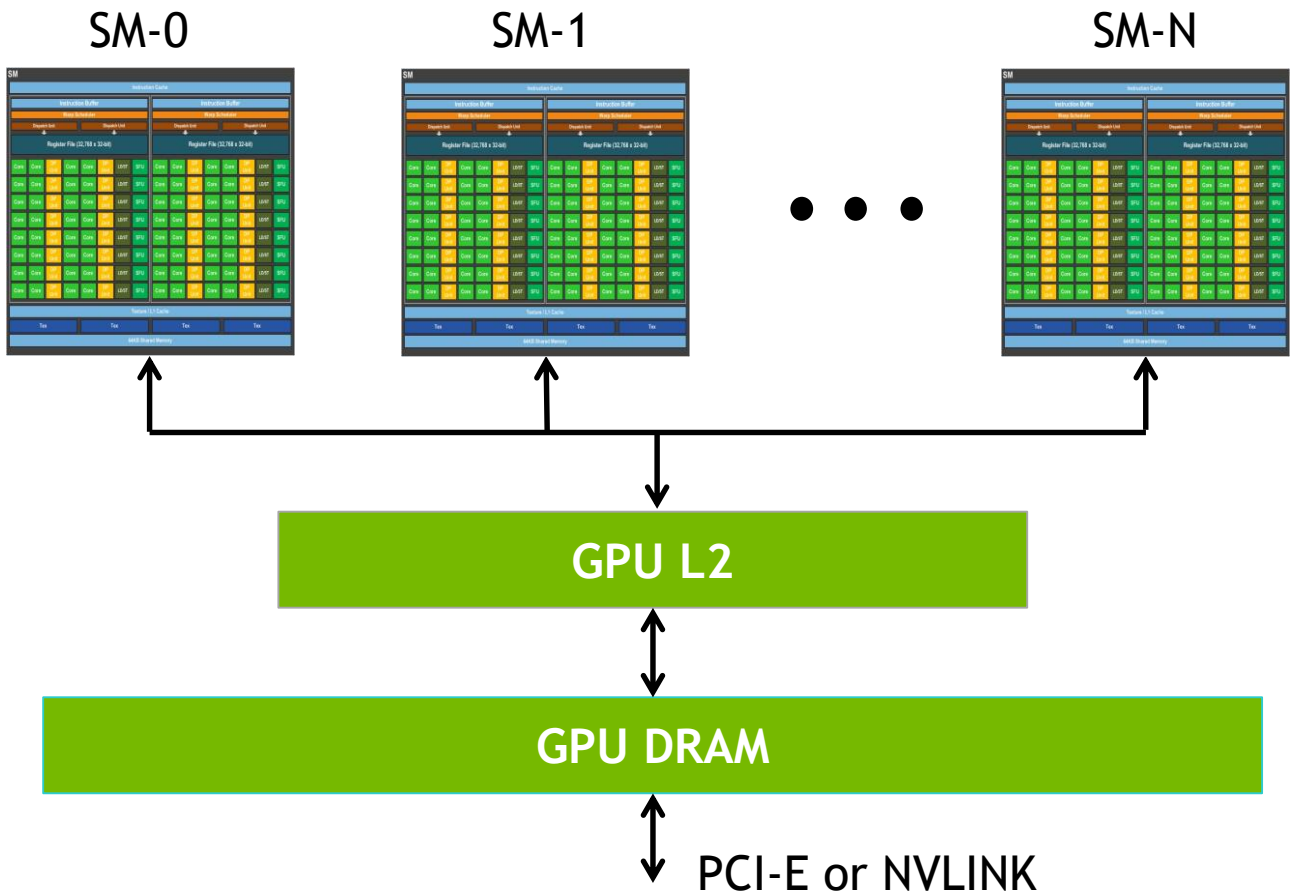
HOW GPU ACCELERATION WORKS



HETEROGENEOUS ARCHITECTURES



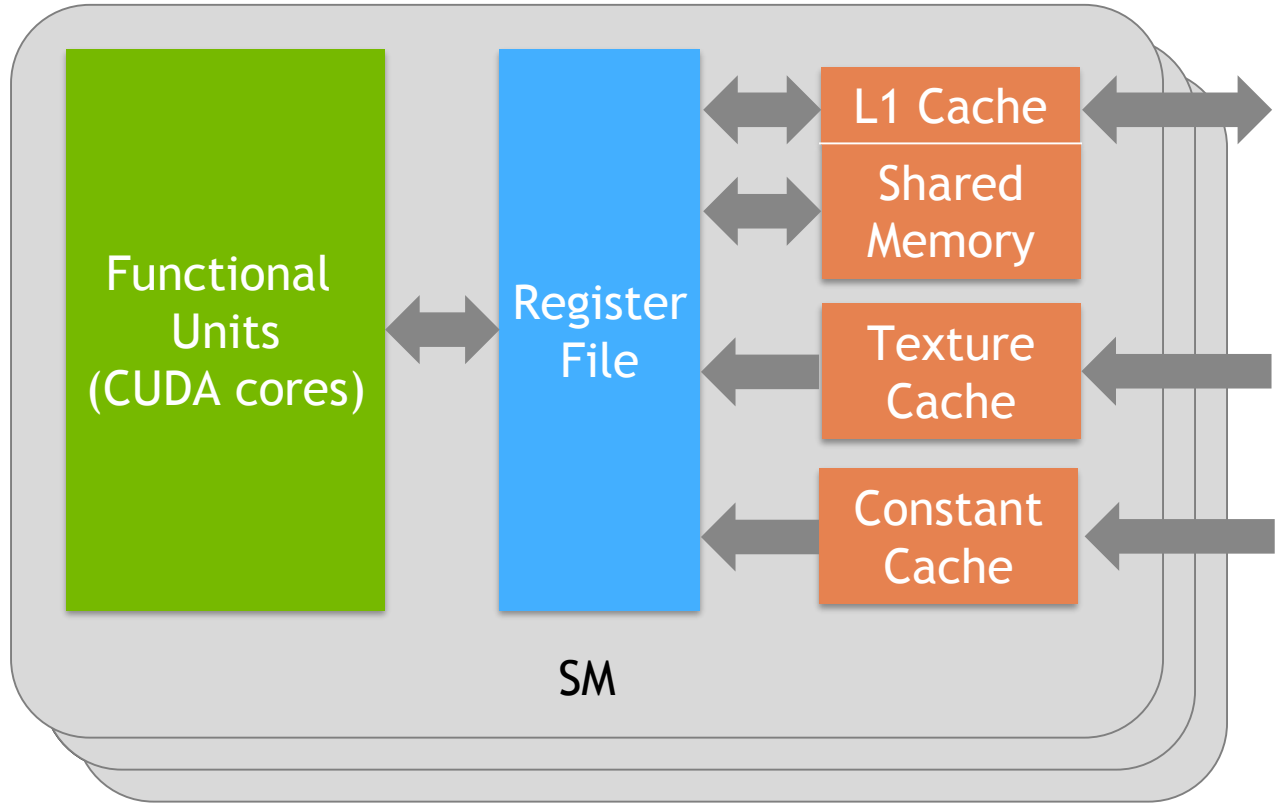
GPU ARCHITECTURE



GPU SM ARCHITECTURE

Kepler SM

	GK110
FP32 Cores	192
FP64 Cores	64
Register File	256 KB
Shared Memory	16/32/48 KB

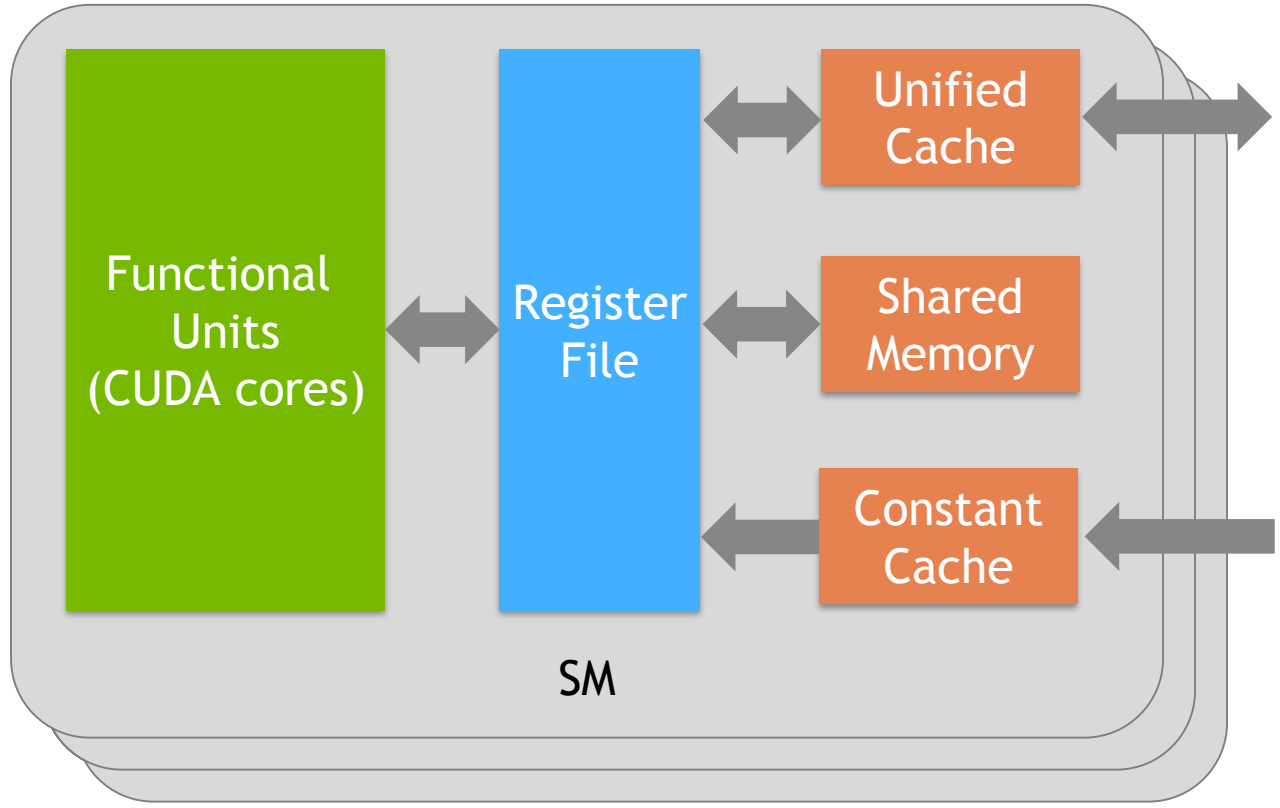


15 SMs on Tesla K40

GPU SM ARCHITECTURE

Pascal SM

	GP100
FP32 Cores	64
FP64 Cores	32
Register File	256 KB
Shared Memory	64 KB

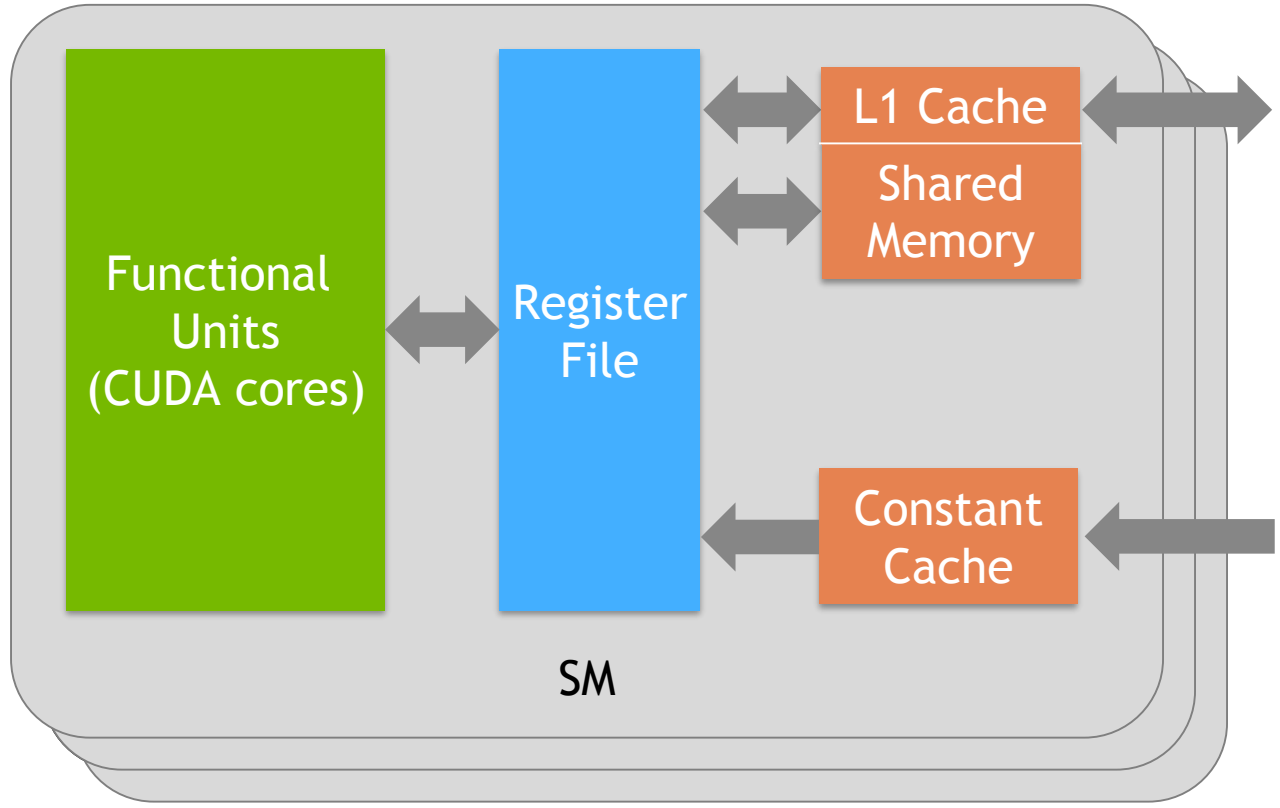


56 SMs on Tesla P100

GPU SM ARCHITECTURE

Volta SM

	GV100
FP32 Cores	64
FP64 Cores	32
Tensor Cores	8
Register File	256 KB
Shared Memory	up to 96 KB



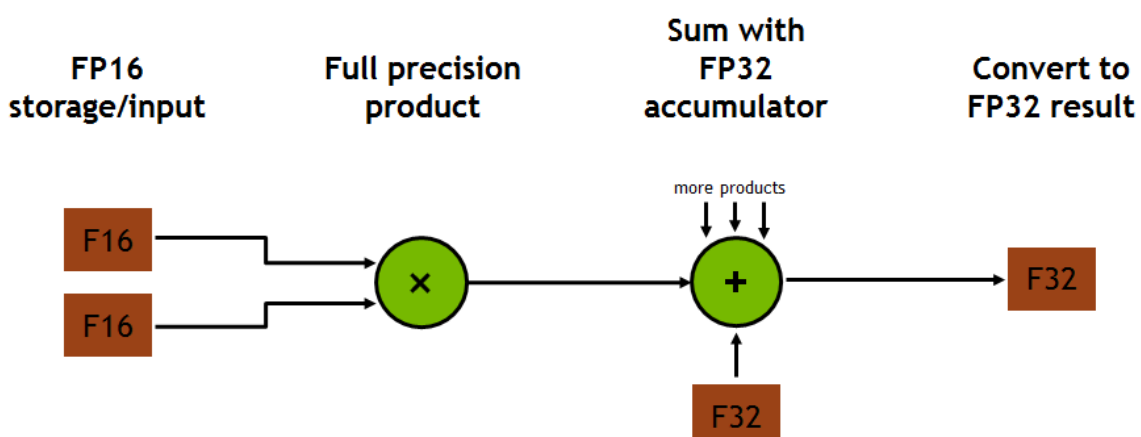
80 SMs on Tesla V100

GPU SM ARCHITECTURE

Tensor Cores

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32



TESLA FAMILY

GPU comparison (boost clocks)

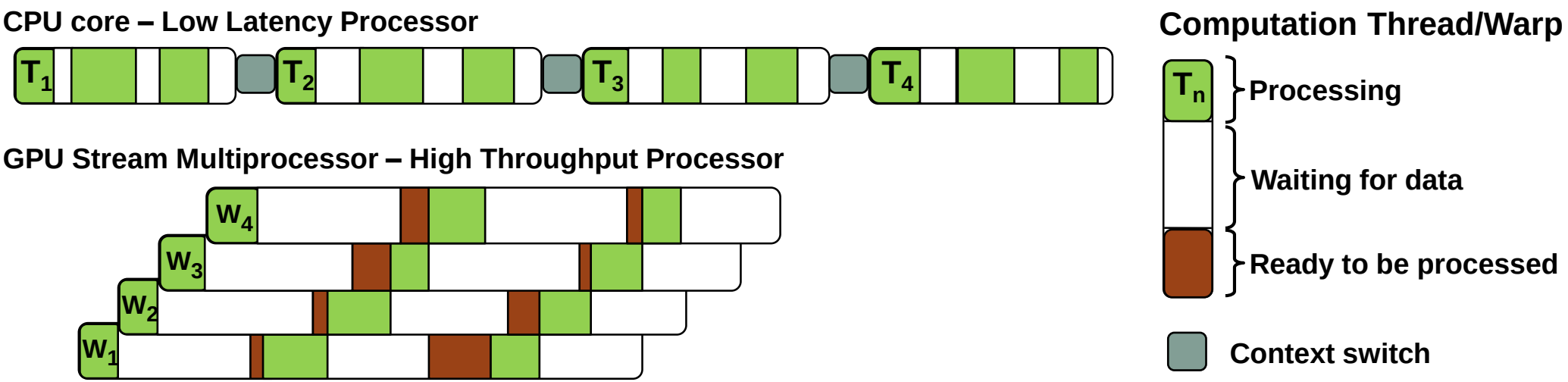
	Tesla K40	Tesla P100	Tesla V100
Peak FP32 (TFLOP/s)	5.04	10.6	15
Peak FP64 (TFLOP/s)	1.68	5.3	7.5
Peak Tensor Core (TFLOP/s)	N/A	N/A	120
Memory Size (GB)	12	16	16
Memory Bandwidth (GB/s)	288	732	900

GPU BEST PRACTICES

LOW LATENCY OF HIGH THROUGHPUT?

CPU architecture must **minimize latency** within each thread

GPU architecture **hides latency** with computation from other threads (warps)

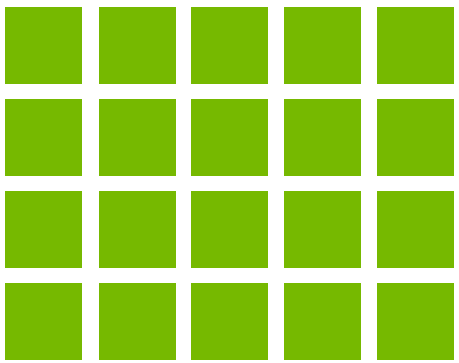


ACCELERATOR FUNDAMENTALS

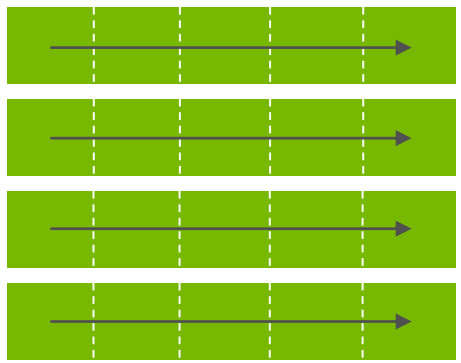
Must expose enough parallelism to saturate the GPU

Accelerator threads are slower than CPU threads

Accelerators have orders of magnitude more threads



Fine-grained parallelism is good

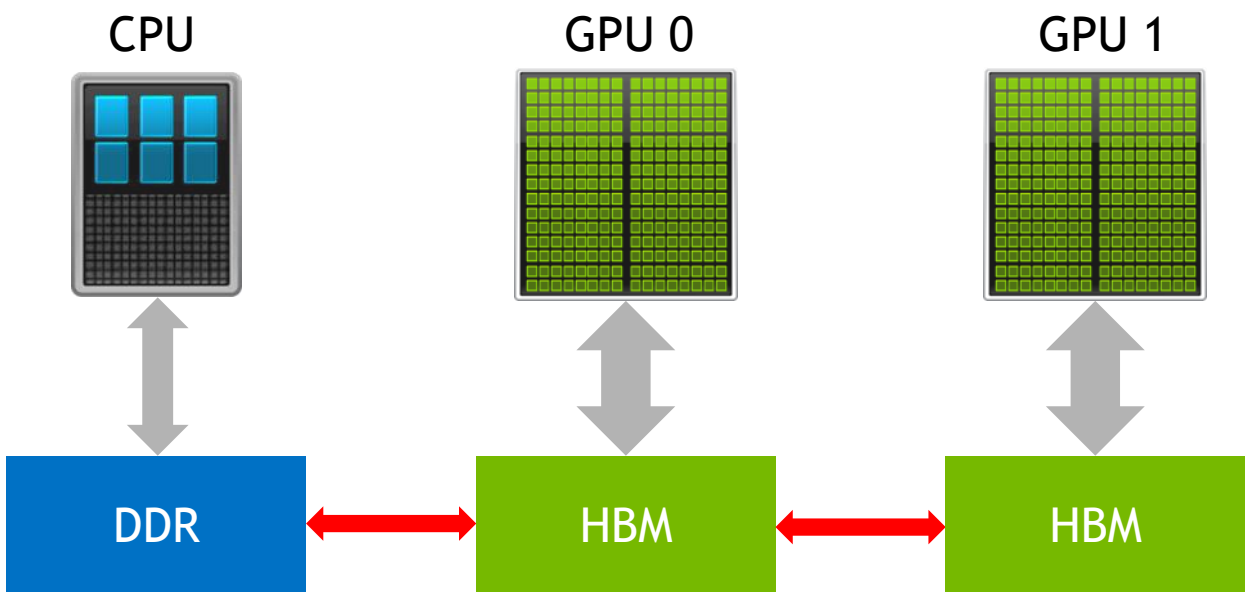


Coarse-grained parallelism is bad

BEST PRACTICES

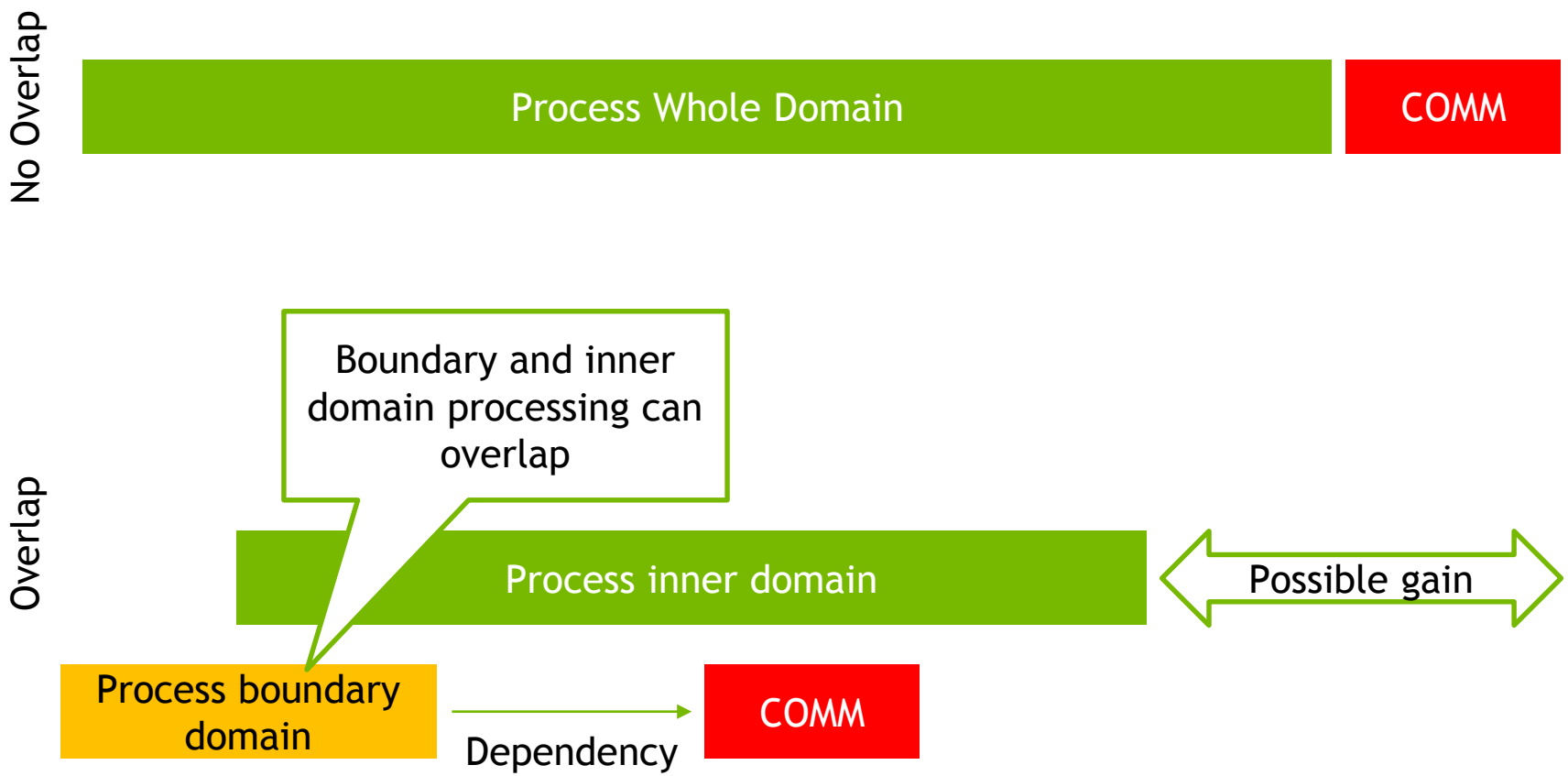
Optimize data locality for CPU and GPU

Minimize data transfers between CPU and GPU, and between peer GPUs



BEST PRACTICES

Overlap computation and communication

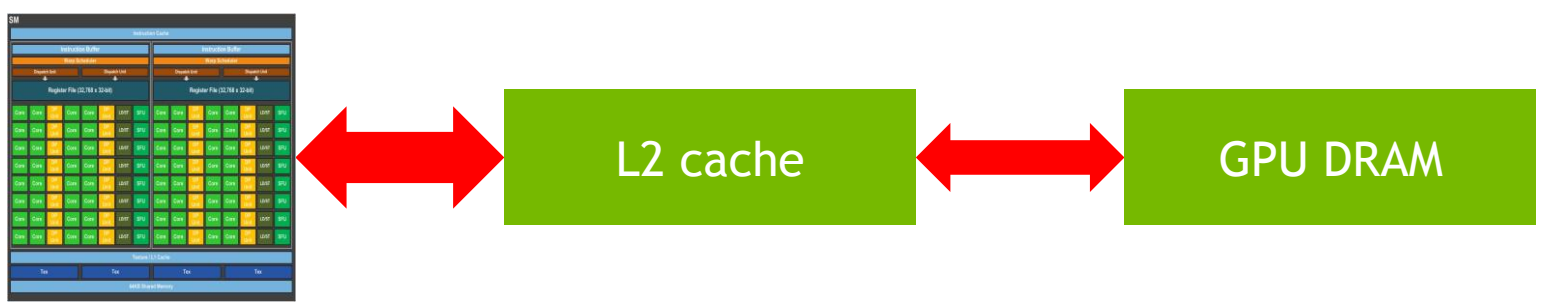


BEST PRACTICES

Optimize data locality for SM

Minimize redundant accesses to L2 and DRAM

- Store intermediate results in **registers** instead of DRAM
- Use **shared memory** for data frequently used within SM
- Use **constant** and **read-only** caches on SM



GPU SM

BEST PRACTICES

How well SM is utilized

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		3	16
Active Warps	44.81	48	64
Active Threads		1536	2048
Occupancy	70.01 %	75.00 %	100.00 %
Warps			
Threads/Block		512	1024
Warps/Block		16	32
Block Limit		4	16
Registers			
Registers/Thread		33	255
Registers/Block		20480	65536
Block Limit		3	16
Shared Memory			
Shared Memory/Block		0	49152
Block Limit		∞	16

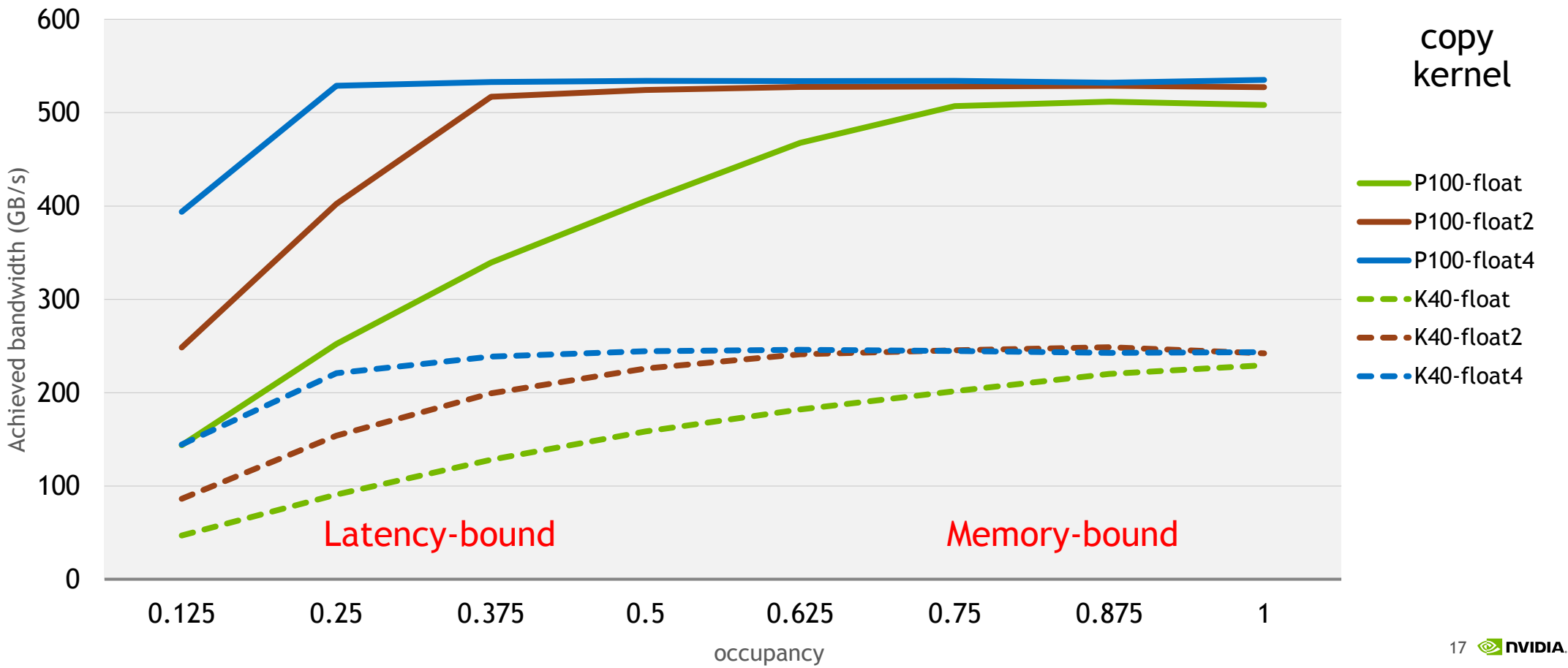
Tradeoff: run more threads per SM or use more resources per thread

Occupancy is a metric to show how effectively the SM is kept busy

But higher occupancy does not always equate to better performance

BEST PRACTICES

Achieved DRAM bandwidth

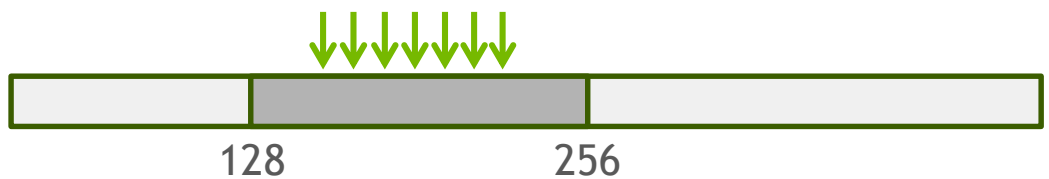


BEST PRACTICES

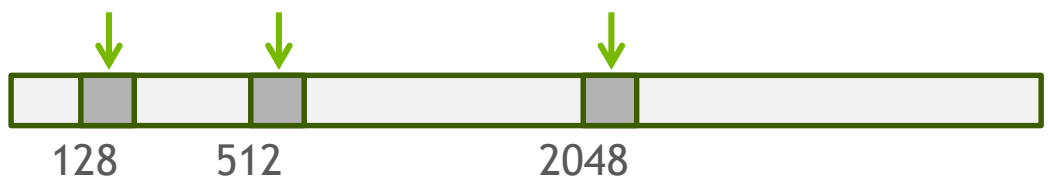
Coalesce memory requests

If addresses from a *warp* lie within the same cache line, that line is fetched once

Best case: addresses lie in a single cache line (128B), **4**x32B transactions

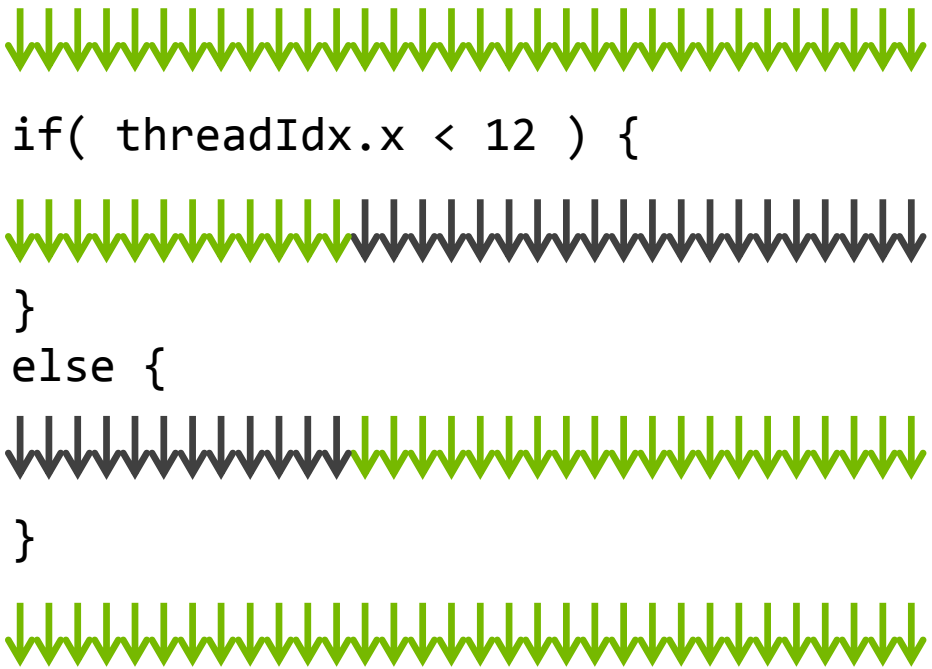


Worst case: fully scattered access, 32 allocated cache lines, **32**x32B transactions



BEST PRACTICES

Avoid warp divergence



Instructions are issued per warp

Different execution paths within a warp are serialized

Different warps can execute different code with no impact on performance

Avoid branching on thread index

BEST PRACTICES

Instruction-Level Parallelism

No ILP

```
float a = 0.0f;
for( int i = 0 ; i < N ; ++i )
  a += logf(b[i]);
```

```
c = b[0]
a += logf(c)
c = b[1]
a += logf(c)
c = b[2]
a += logf(c)
c = b[3]
a += logf(c)
```

2-way ILP (with loop unrolling)

```
float a, a0 = 0.0f, a1 = 0.0f;
for( int i = 0 ; i < N ; i += 2 )
{
  a0 += logf(b[i]);
  a1 += logf(b[i+1]);
}
a = a0 + a1
```

```
c0 = b[0]
a0 += logf(c0)
c0 = b[2]
a0 += logf(c0)
...
a = a0 + a1
```

```
c1 = b[1]
a1 += logf(c1)
c1 = b[3]
a1 += logf(c1)
```

PROGRAMMING GPUS

3 WAYS TO PROGRAM GPUS

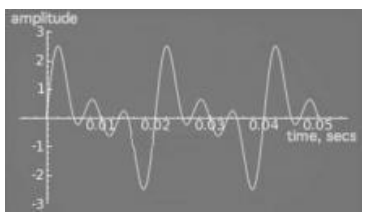
Applications

Libraries	Compiler Directives	Programming Languages
"Drop-in" Acceleration	Easy to use Portable code	Maximum Performance and Flexibility

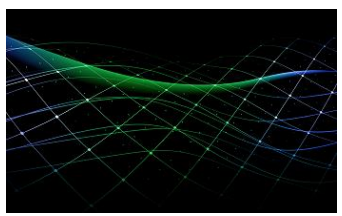
NVIDIA DEVELOPER LIBRARIES

cuBLAS
cuBLAS-XT
NVBLAS

cuFFT
cuFFT-XT

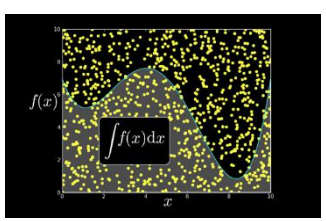


cuSPARSE
cuSOLVER
AMGX



cuDNN

cuRAND

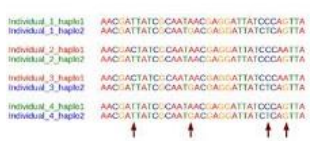


THRUST

NPP

NVENC

NVBIO



https://developer.nvidia.com/gpu-accelerated-libraries

OPENACC DIRECTIVES

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc kernels
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          z[i] = x[i] + y[i];
          ...
      }
  }
  ...
}
```

OpenACC
Directives for Accelerators

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU

CUDA

Maximum flexibility

CPU code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{


  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}


// Perform SAXPY on 1M elements
saxpy_serial(4096 * 256, 2.0, x, y);
```

GPU code

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  int i = blockIdx.x * blockDim.x +
          threadIdx.x;
  if (i < n) y[i] = a * x[i] + y[i];
}


// Perform SAXPY on 1M elements
saxpy_parallel<<<4096, 256>>>(n, 2.0, x, y);
```

CUDA + LAMBDA

No need to define explicit kernel

CPU code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{


  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}


// Perform SAXPY on 1M elements
saxpy_serial(4096 * 256, 2.0, x, y);
```

GPU code

```
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  for_each(counting_iterator<int>(0),
           counting_iterator<int>(n),
           [=] __device__ (int i)
  {
    y[i] = a * x[i] + y[i];
  });
}

// Perform SAXPY on 1M elements
saxpy_parallel(4096 * 256, 2.0, x, y);
```

MEMORY MANAGEMENT

EXPLICIT CONTROL

Custom Data Movement

CPU code

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

cpu_func1(data, N);

cpu_func2(data, N);


cpu_func3(data, N);

free(data);
```

GPU code

```
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, ...)
cpu_func3(data, N);

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

UNIFIED MEMORY

Single Pointer, Automatic Migration

CPU code

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

cpu_func1(data, N);

cpu_func2(data, N);


cpu_func3(data, N);

free(data);
```

GPU code

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

cpu_func1(data, N);

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

cpu_func3(data, N);

free(data);
```

UNIFIED MEMORY

Deep Copy Nightmare

Explicit Memory Management

```
char **data;
data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
  data[i] = (char*)malloc(N);

char **d_data;
char **h_data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++) {
  cudaMalloc(&h_data2[i], N);
  cudaMemcpy(h_data2[i], h_data[i], N, ...);
}
cudaMalloc(&d_data, N*sizeof(char*));
cudaMemcpy(d_data, h_data2, N*sizeof(char*), ...);

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

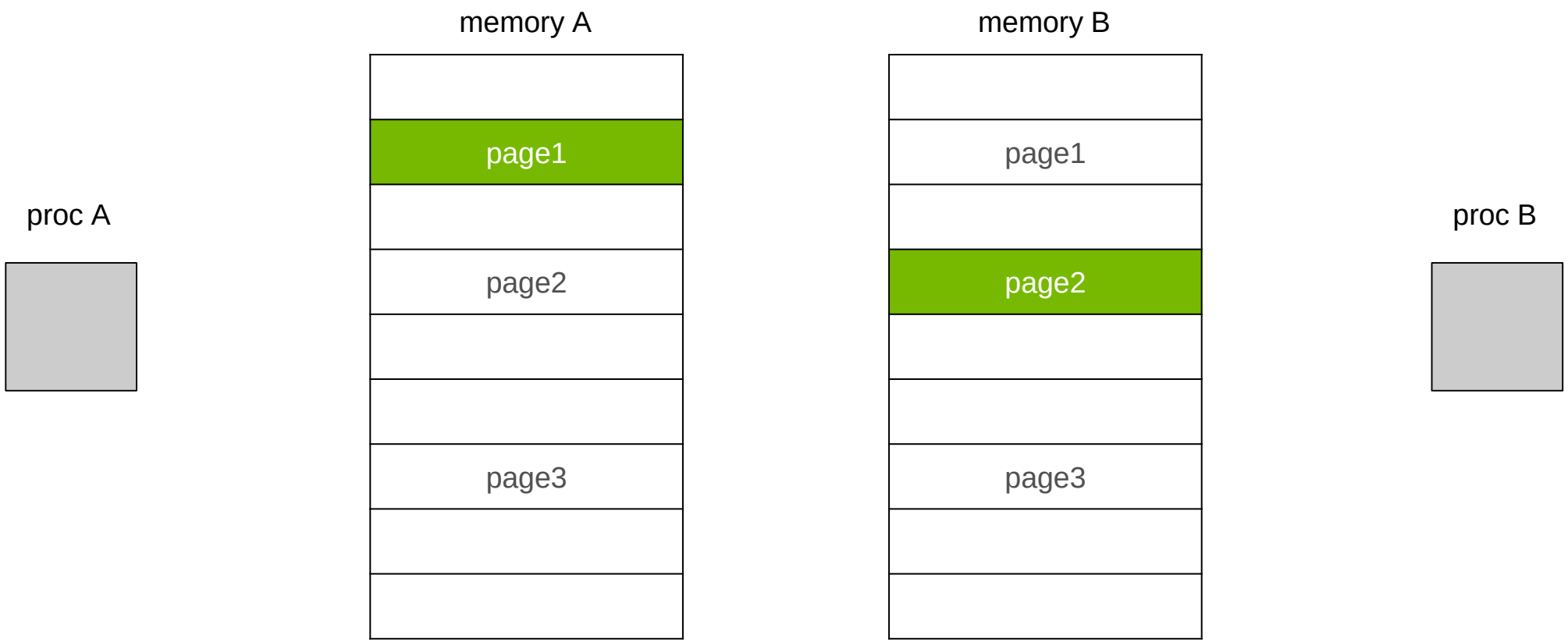
Unified Memory

```
char **data;
data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
  data[i] = (char*)malloc(N);

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

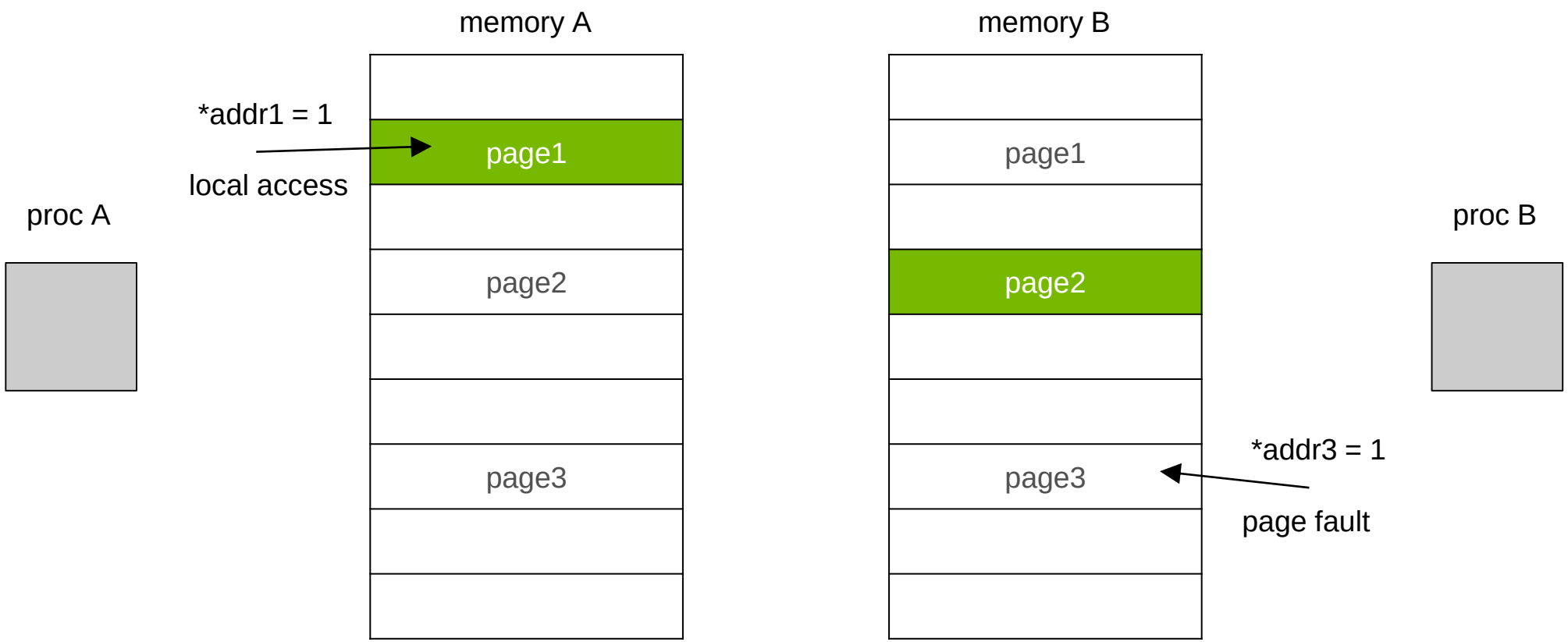
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



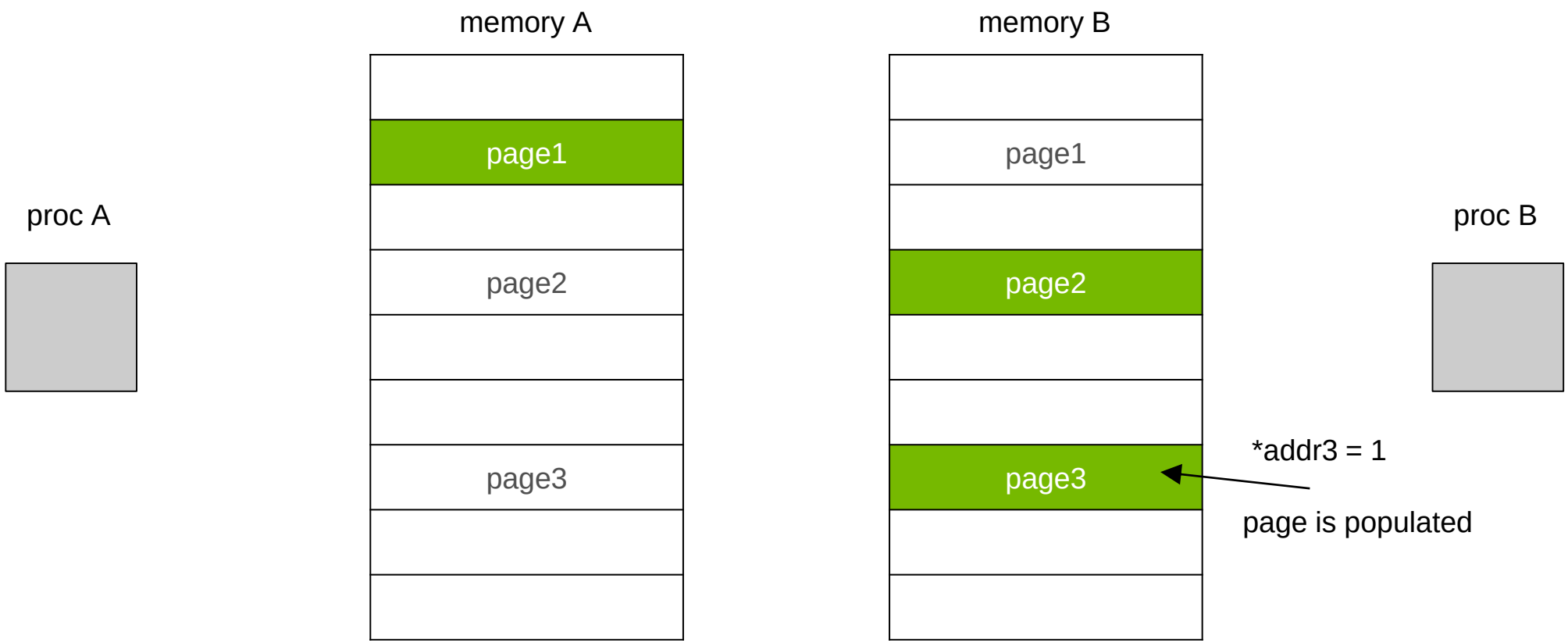
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



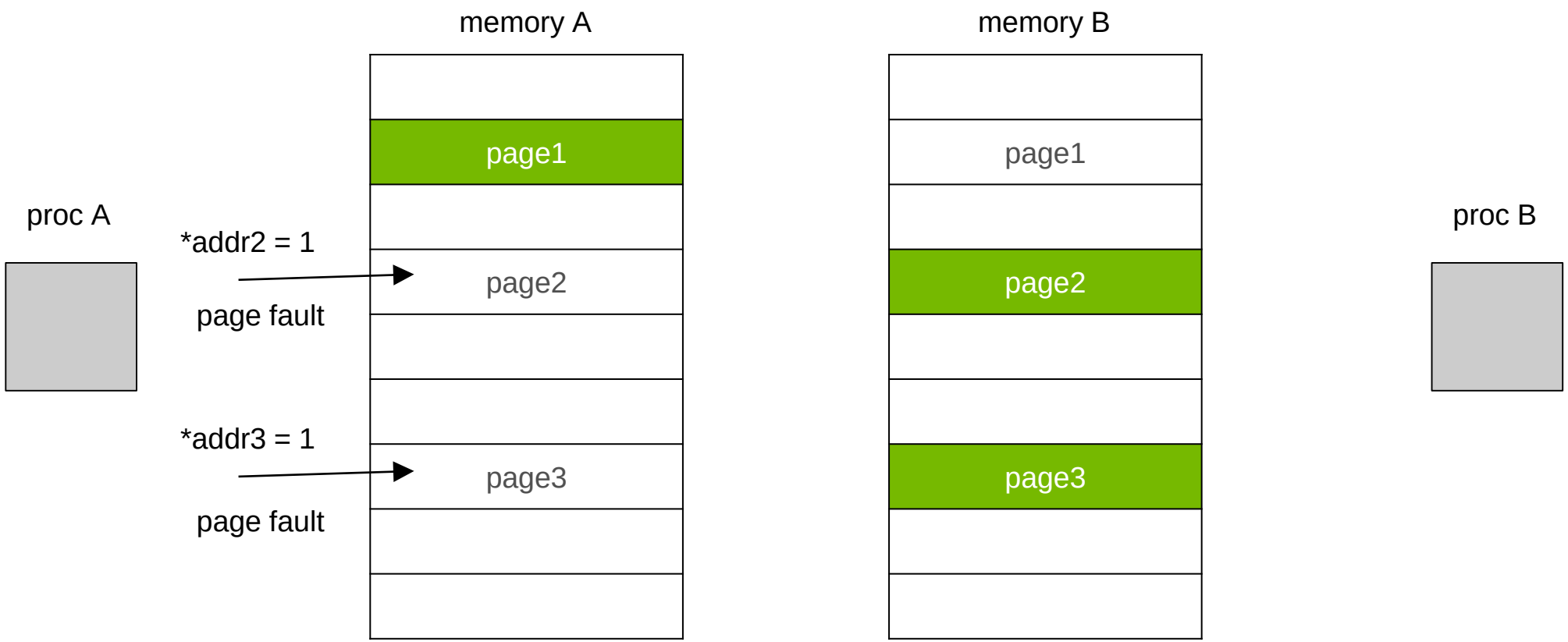
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



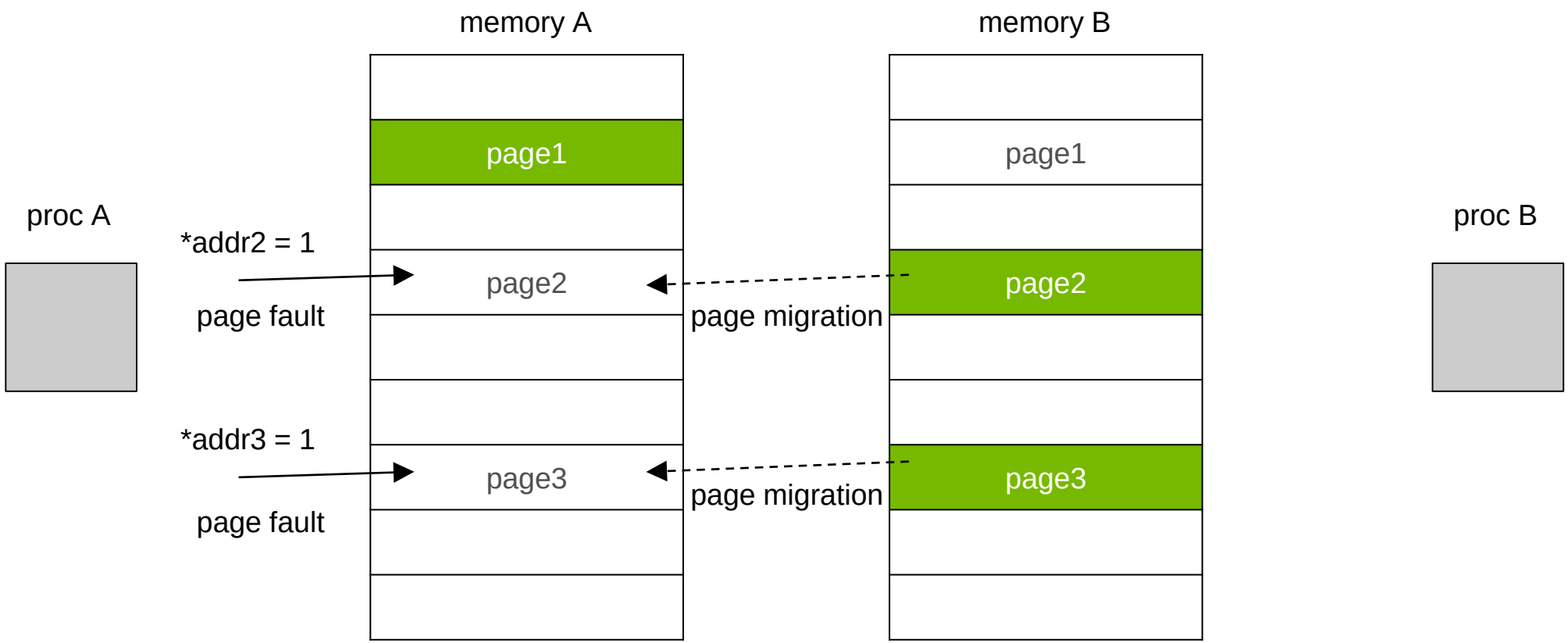
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



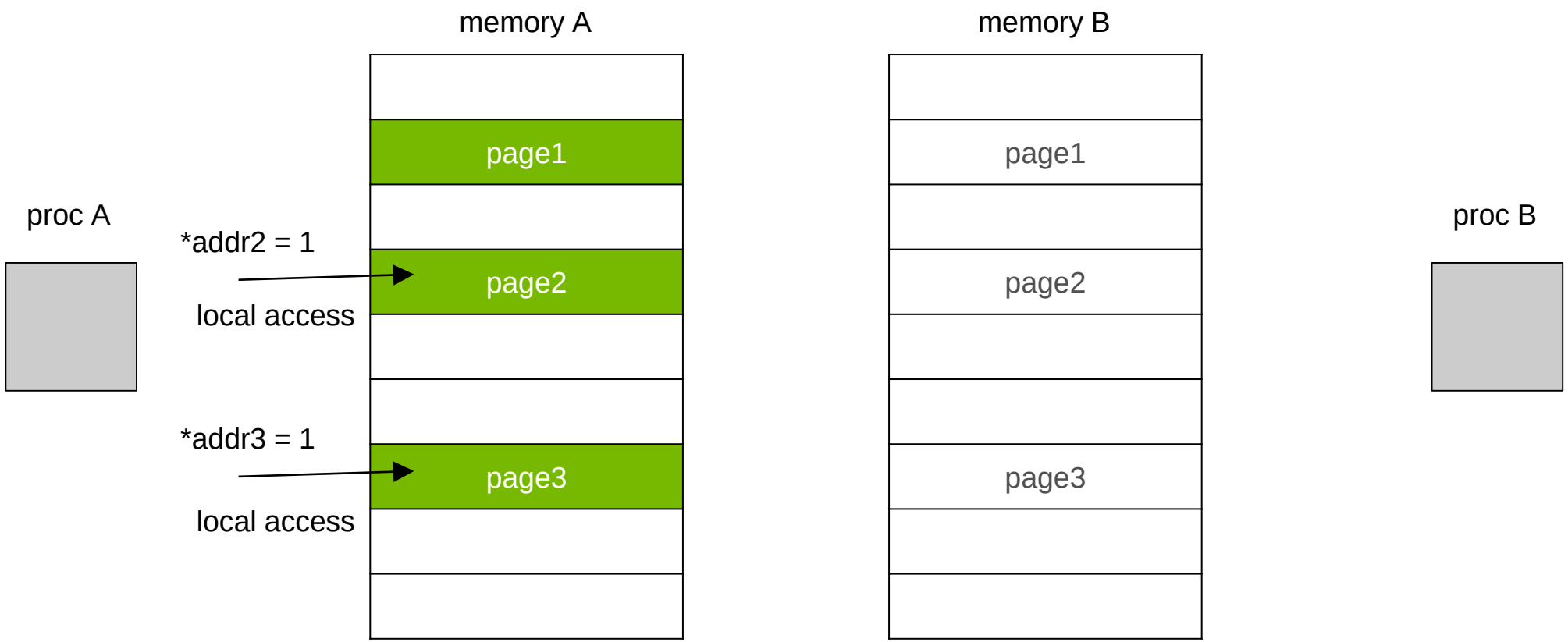
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



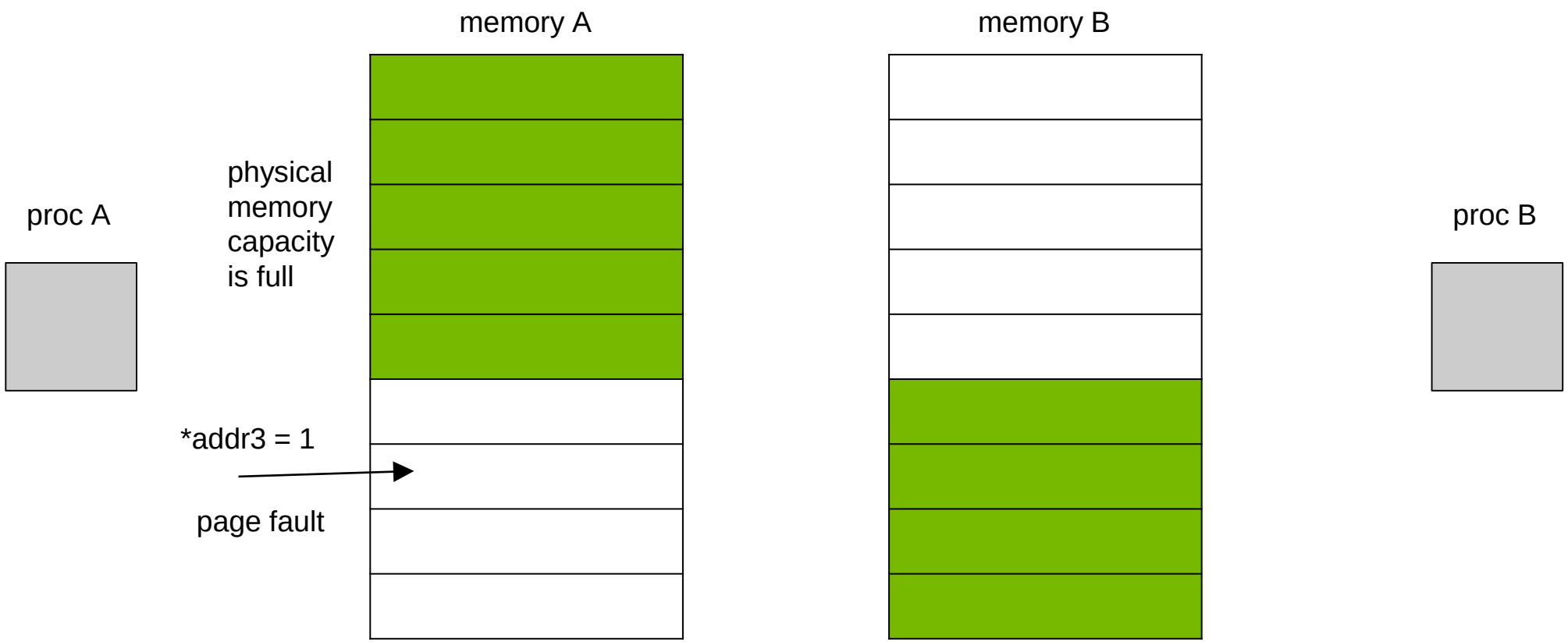
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



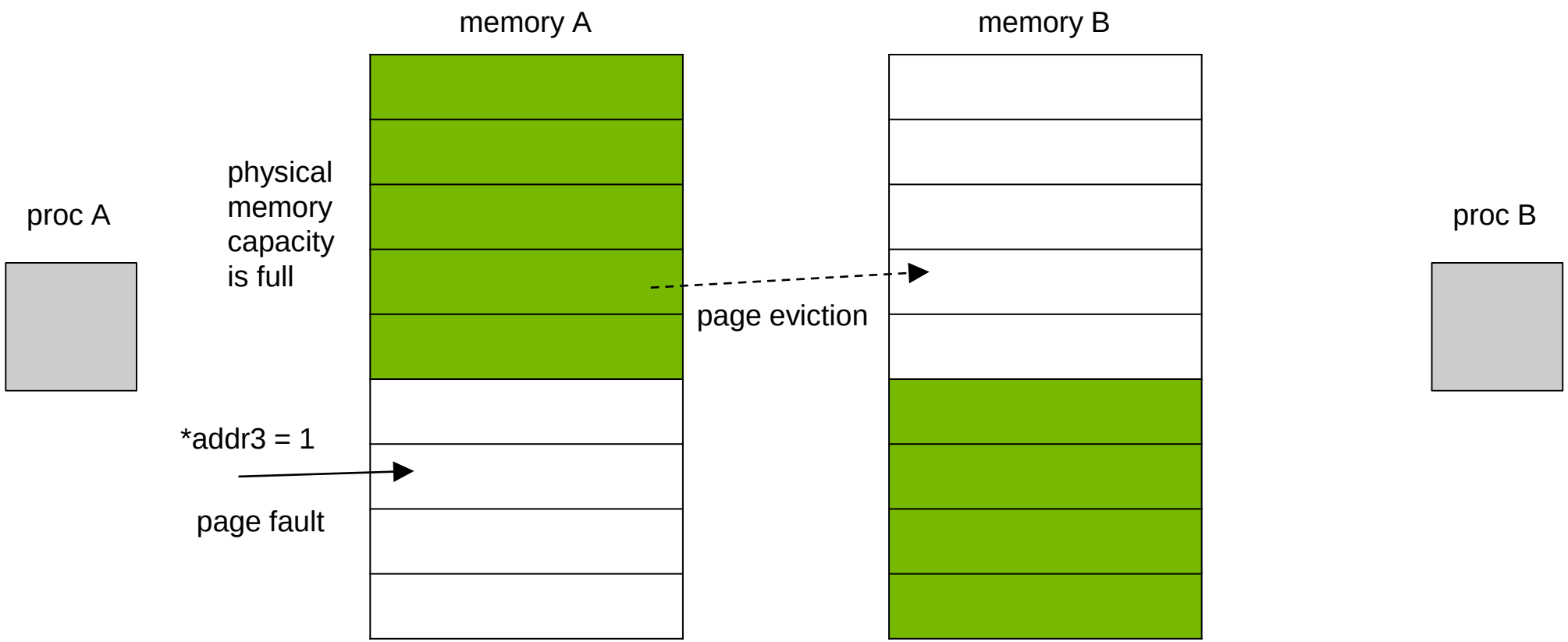
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



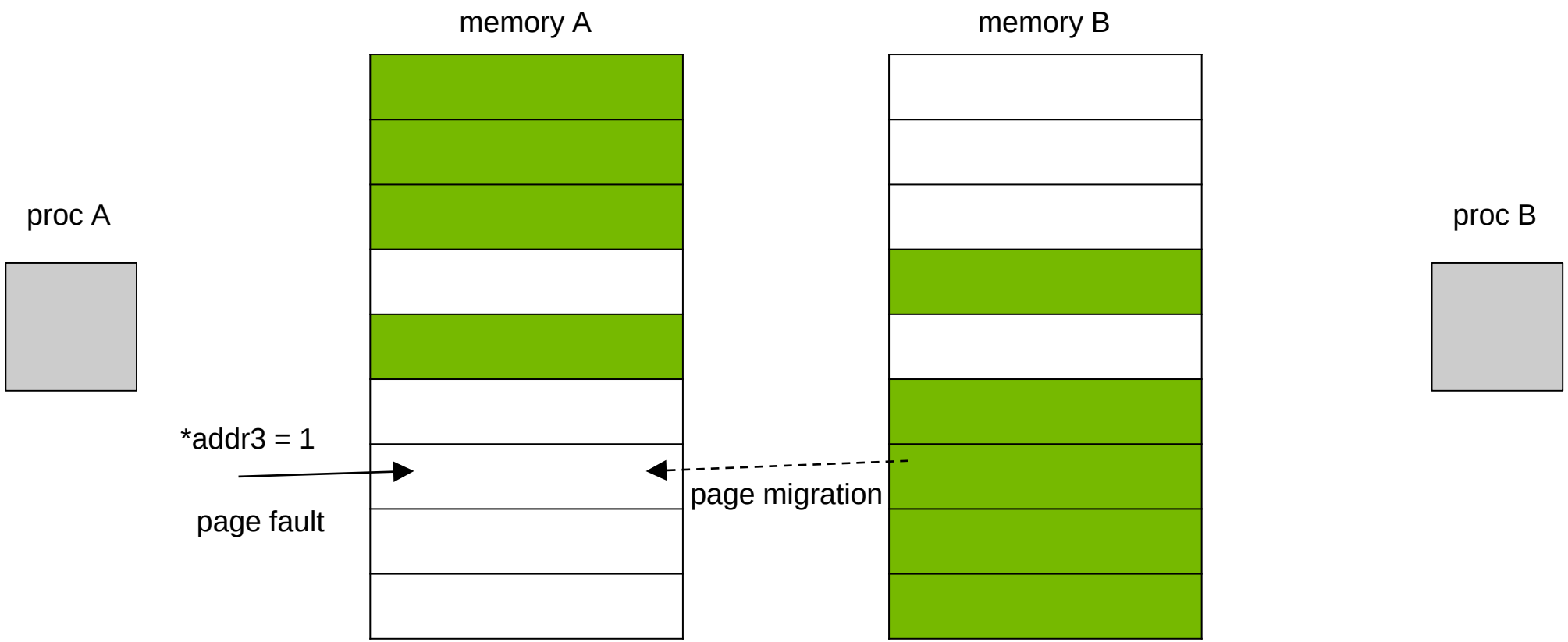
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



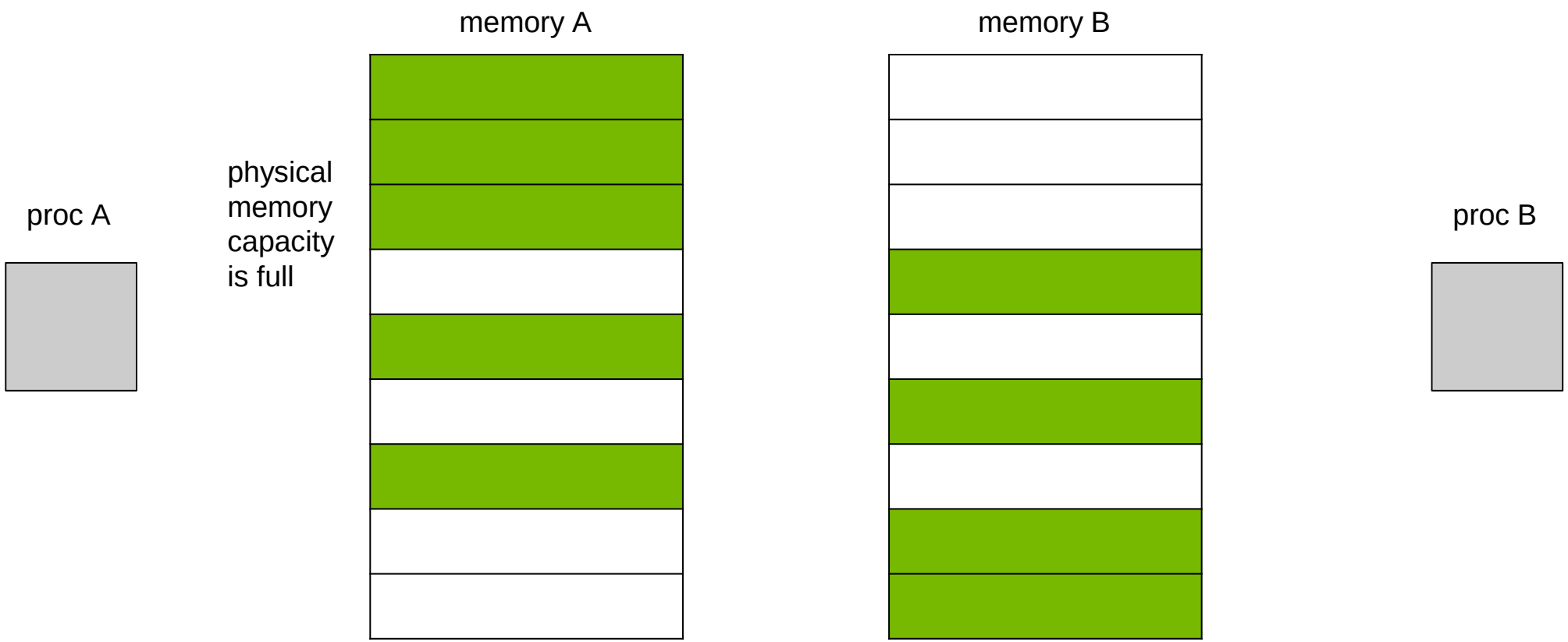
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



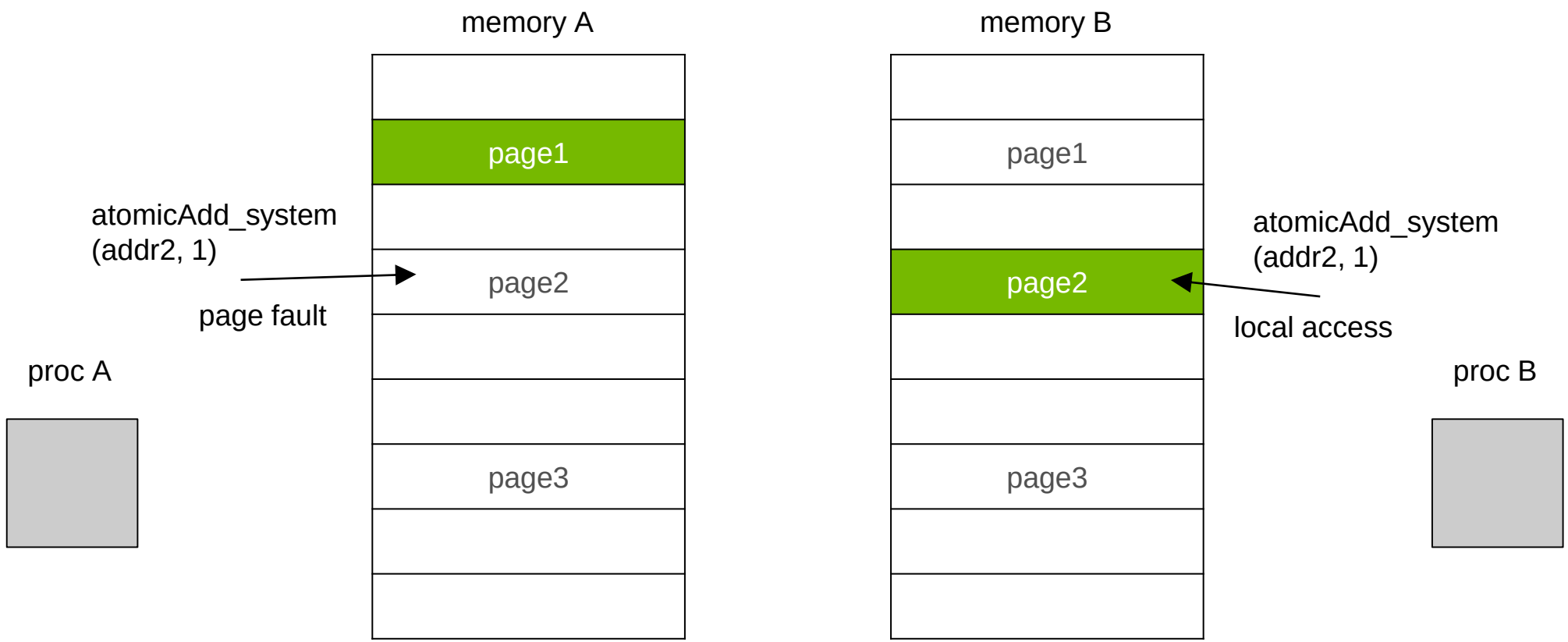
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



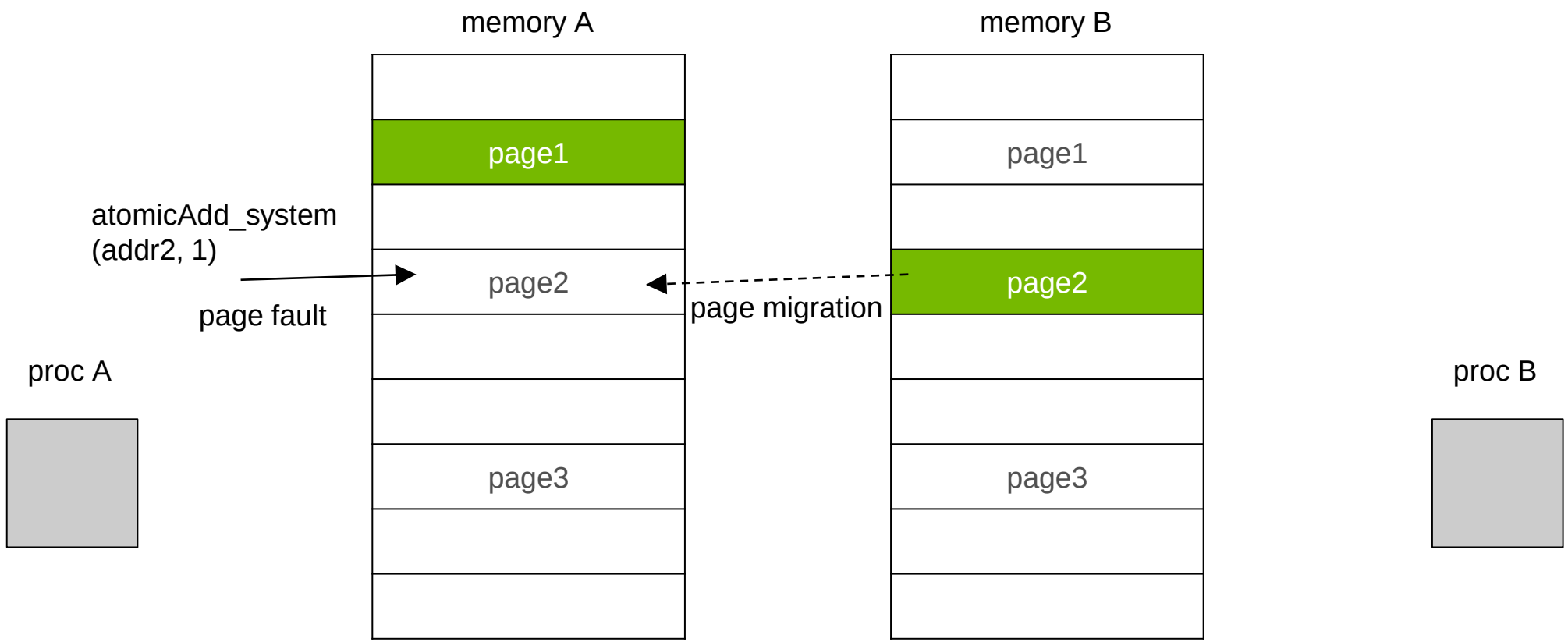
UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics with Exclusive Access



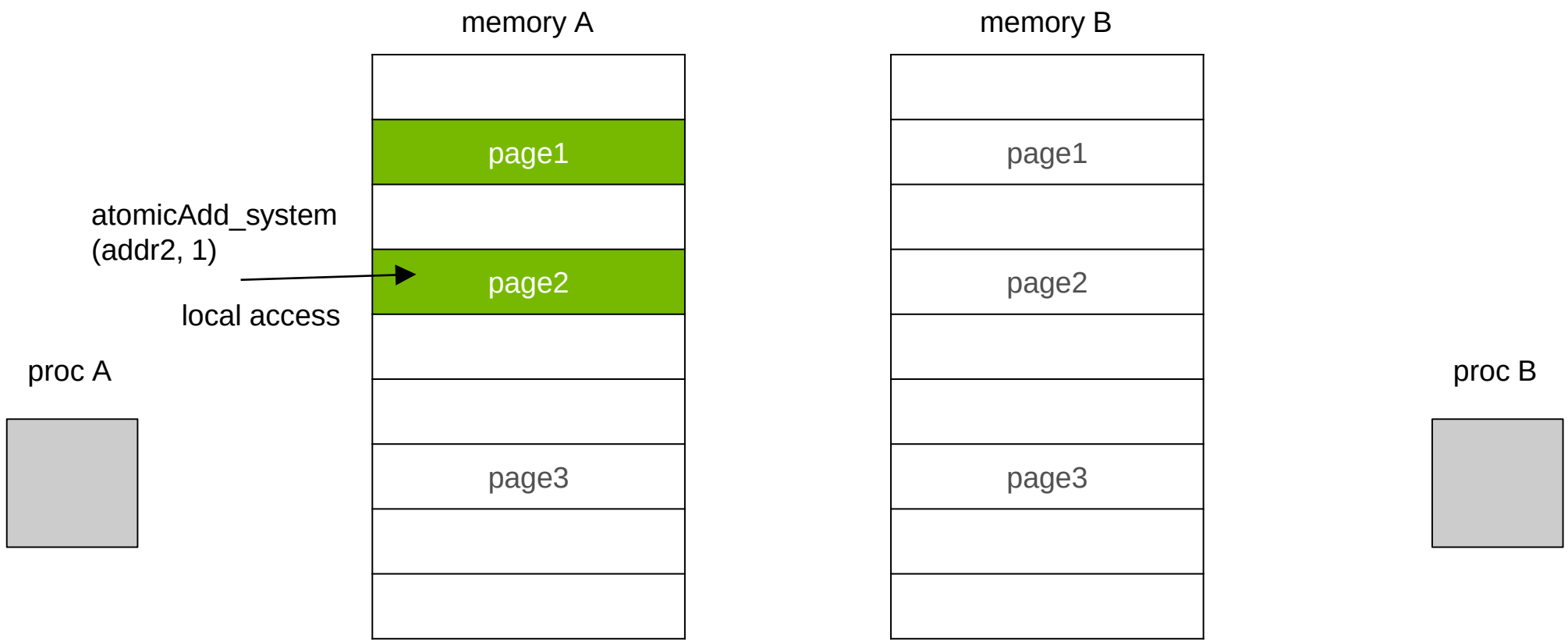
UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics with Exclusive Access



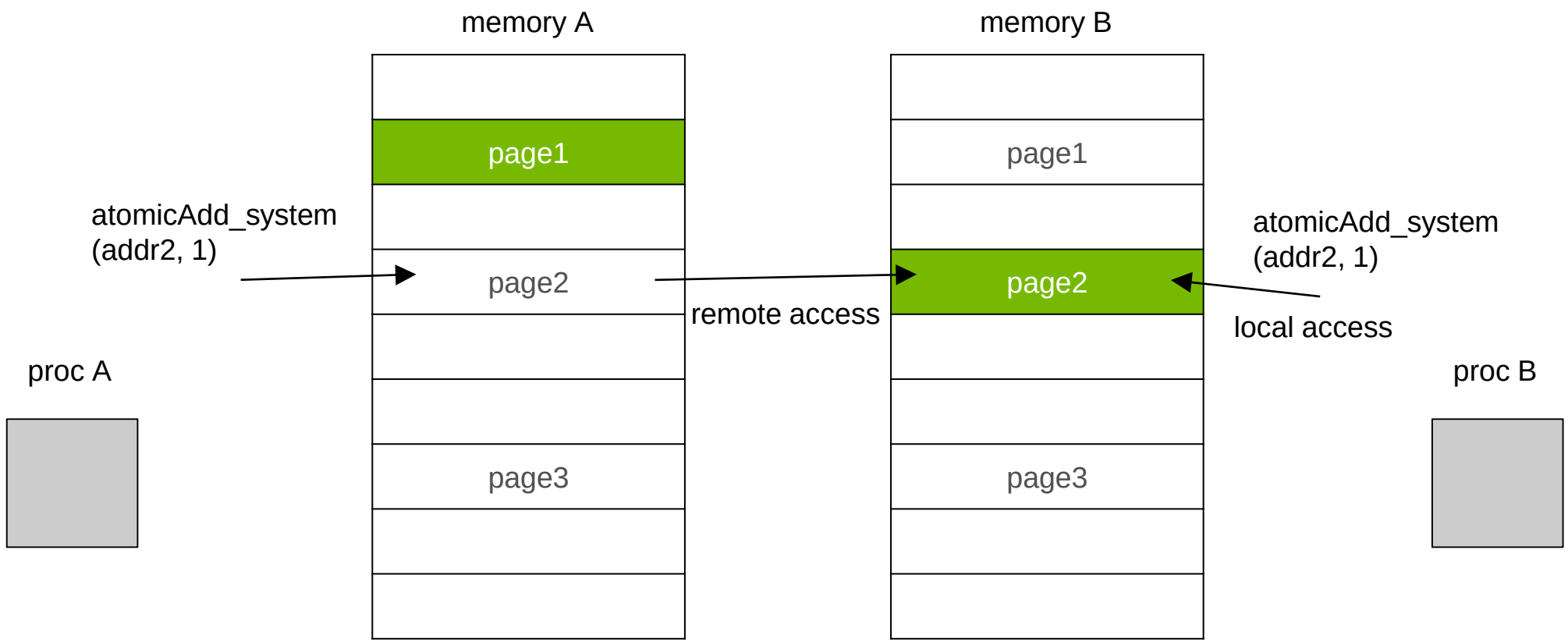
UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics with Exclusive Access



UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics over NVLINK*



*both processors need to support atomic operations

UNIFIED MEMORY ALLOCATOR

Available Options

CUDA C: **cudaMallocManaged** is your most reliable way to opt in today

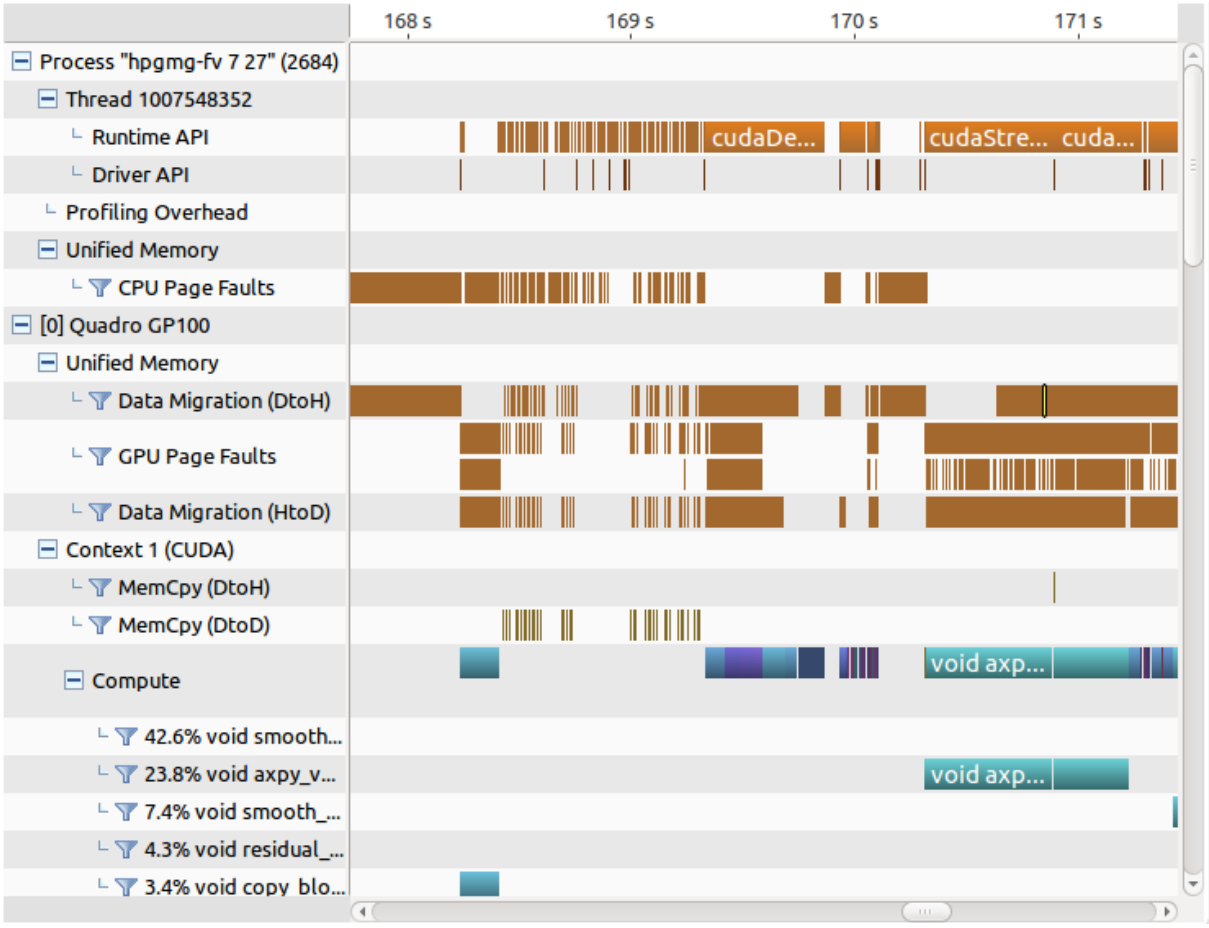
CUDA Fortran: **managed** attribute (per allocation)

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

malloc support is coming on Pascal+ architectures (Linux only)

Note: you can write your own malloc hook to use cudaMallocManaged

PROFILER: INSPECT



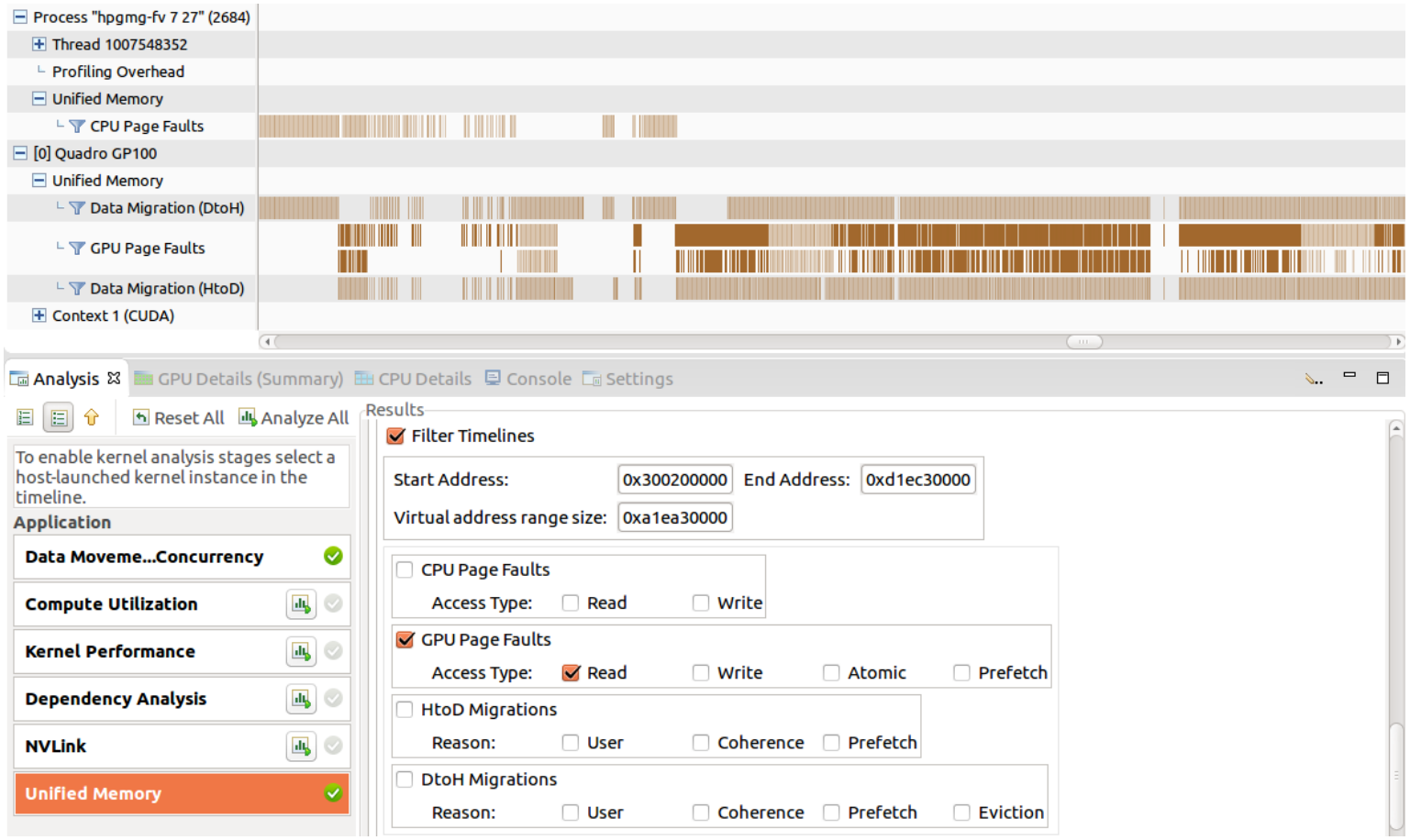
Data Migration (DtoH)

Start	170.845 s (170,844,511,496 ns)
End	170.845 s (170,844,671,976 ns)
Duration	160.48 µs
Size	2.097 MB
Migration Cause	Eviction
Throughput	13.068 GB/s
Virtual Address	0x4b2a00000
Memory operations	Browse for executable...
Process	2684

GPU Page Fault groups

Start	170.925 s (170,924,798,600 ns)
End	170.925 s (170,924,919,048 ns)
Duration	120.448 µs
Memory Acccess Type	Write
Virtual Address	0x34b680000
GPU Page Faults	2
Memory operations	Browse for executable...
Process	2684

PROFILER: FILTER



PROFILER: CORRELATE

*NewSession2

0.463 s 0.465 s

- Process "uvm_migration" (8021)
  - Thread 2816767808
    - Runtime API
    - Driver API
  - Profiling Overhead
  - Unified Memory
    - CPU Page Faults
- [0] Quadro GP100
  - Unified Memory
    - GPU Page Faults
    - Data Migration (HtoD)
  - Context 1 (CUDA)
    - Compute
      - 100.0% write(int*, u...
    - Streams
      - Default

Properties

Data Migration (HtoD)

Start	465.073 ms (465,072,848 ns)
End	465.15 ms (465,149,520 ns)
Duration	76.672 µs
Size	933.888 kB
Migration Cause	Prefetch
Throughput	12.18 GB/s
Virtual Address	0x50071c000
Memory operations	
Allocation	_Z13test_ondemandm@uvm_migration.cu:24
Free	_Z13test_ondemandm@uvm_migration.cu:40
Process	8021

uvm_migration.cu

```
  for(size_t tid = threadIdx.x + blockIdx.x * blockDim.x; tid < n;
    a[tid] = 2;
}

void test_ondemand(size_t s)
{
  int *a;
  cudaMallocManaged(&a, s);
  size_t n = s / sizeof(int);
  for (size_t i = 0; i < n; i++) a[i] = 1; // populate on the cpu

  struct timespec start, end;
  clock_gettime(CLOCK_MONOTONIC, &start);

  int block = 256;
  int grid = min((size_t)32 * 1024, (n+255)/block);
  write<<<grid, block>>>(a, n); // on-demand migration to the gpu
  cudaDeviceSynchronize();

  clock_gettime(CLOCK_MONOTONIC, &end);
  uint64_t t = get_time(start, end);
  printf("%.3f GB/s\n", (double)s/t);

  cudaFree(a);
}
```

USER HINTS

Why, When, and How to Use Them

If you know your application well you can optimize with hints

These are also useful to override some of the driver heuristics

`cudaMemPrefetchAsync``(ptr, size, processor, stream)`

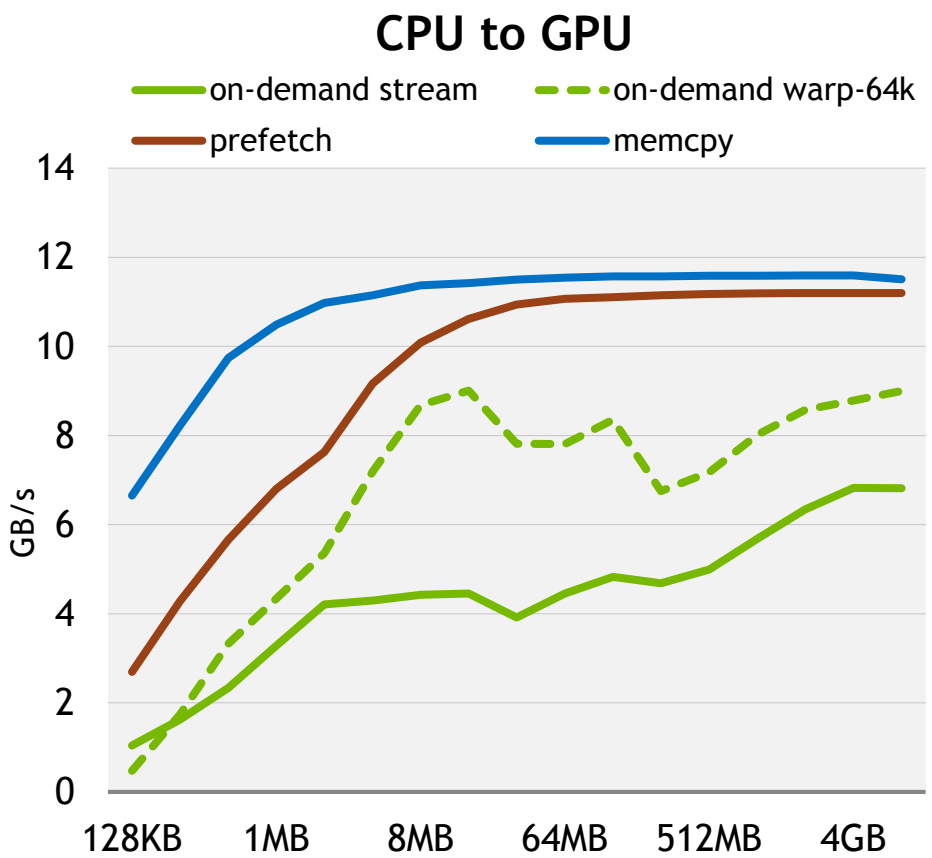
Similar to `move_pages()` in Linux

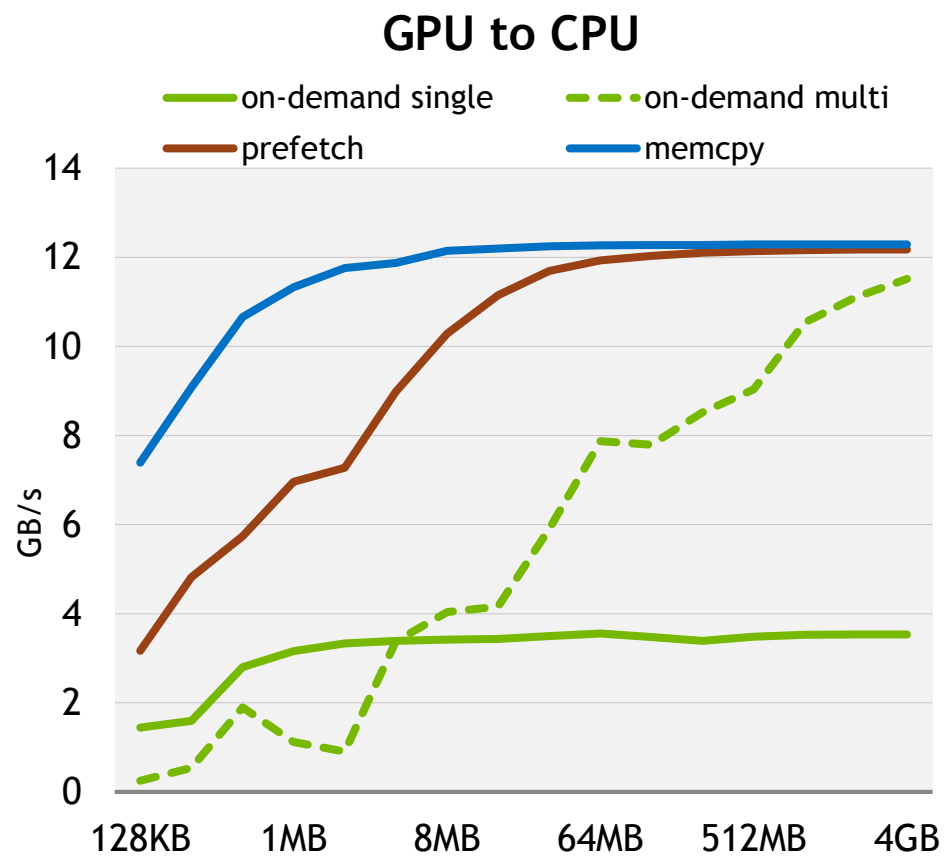
`cudaMemAdvise``(ptr, size, advice, processor)`

Similar to `madvise()` in Linux

PERFORMANCE

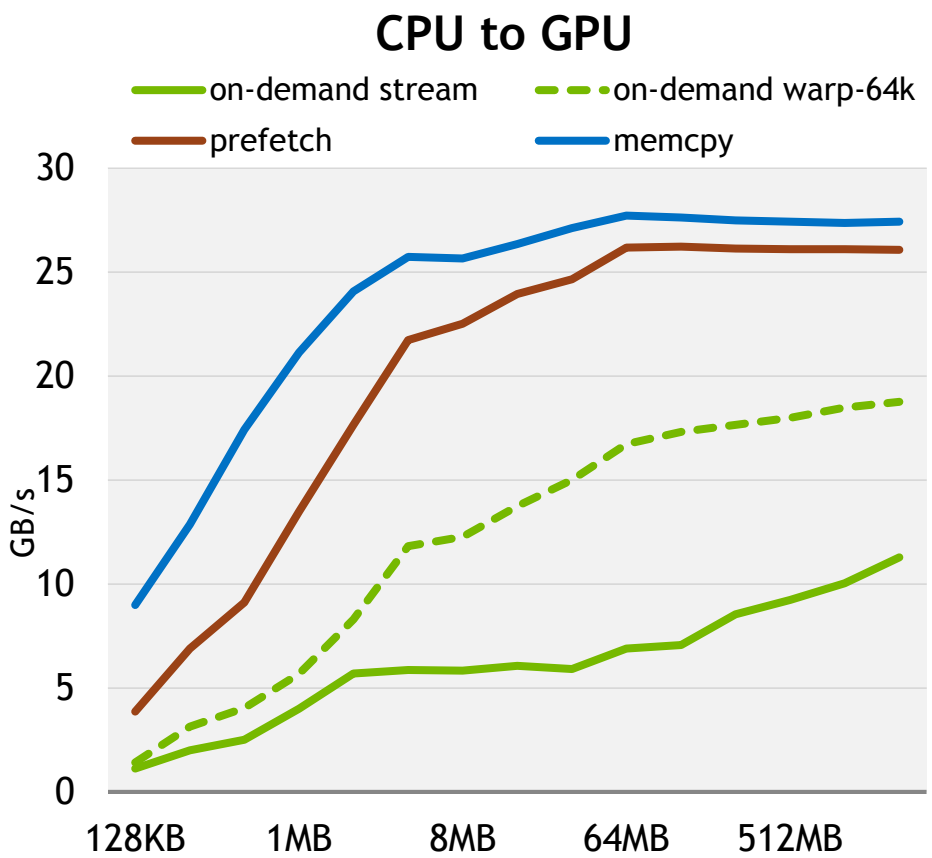
Page Migration Throughput (PCIe)

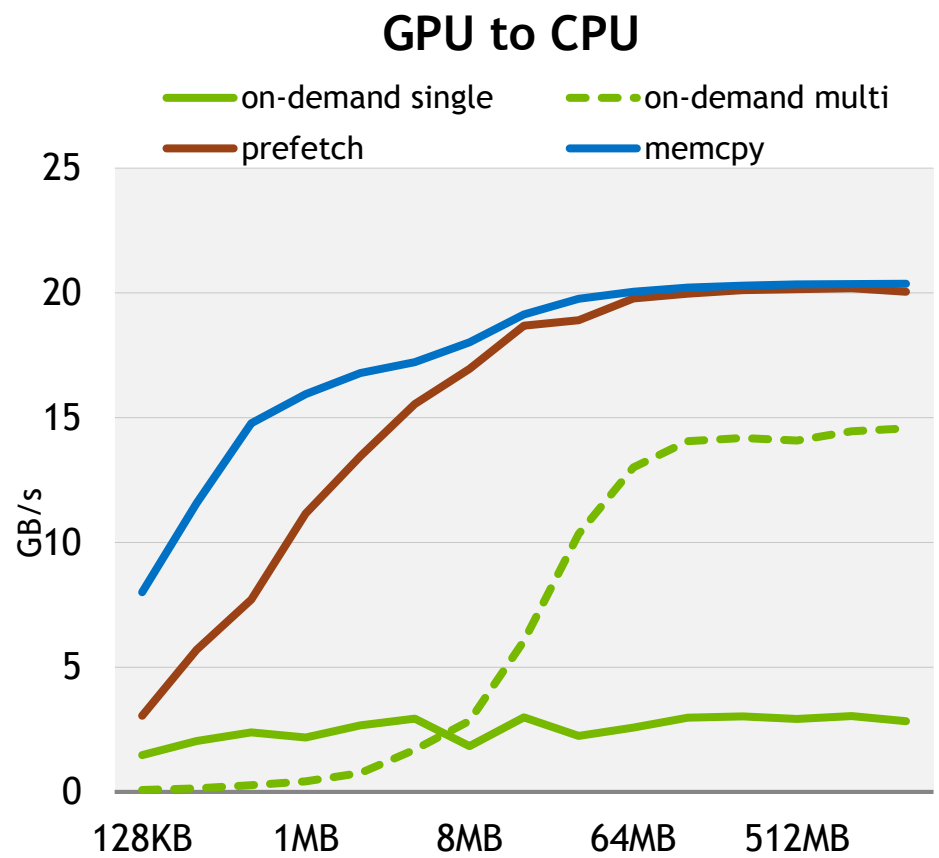




PERFORMANCE

Page Migration Throughput (2x NVLINK)





UNIFIED MEMORY OUTLOOK

Consider using Unified Memory for any new application development

- Get your code *running* on the GPU much sooner!
- Enjoy clean code and *virtually* no memory limits

Increase productivity, explore and prototype new algorithms

- Use the explicit data management only *where you need it*

RESOURCES

Learn more about GPUs

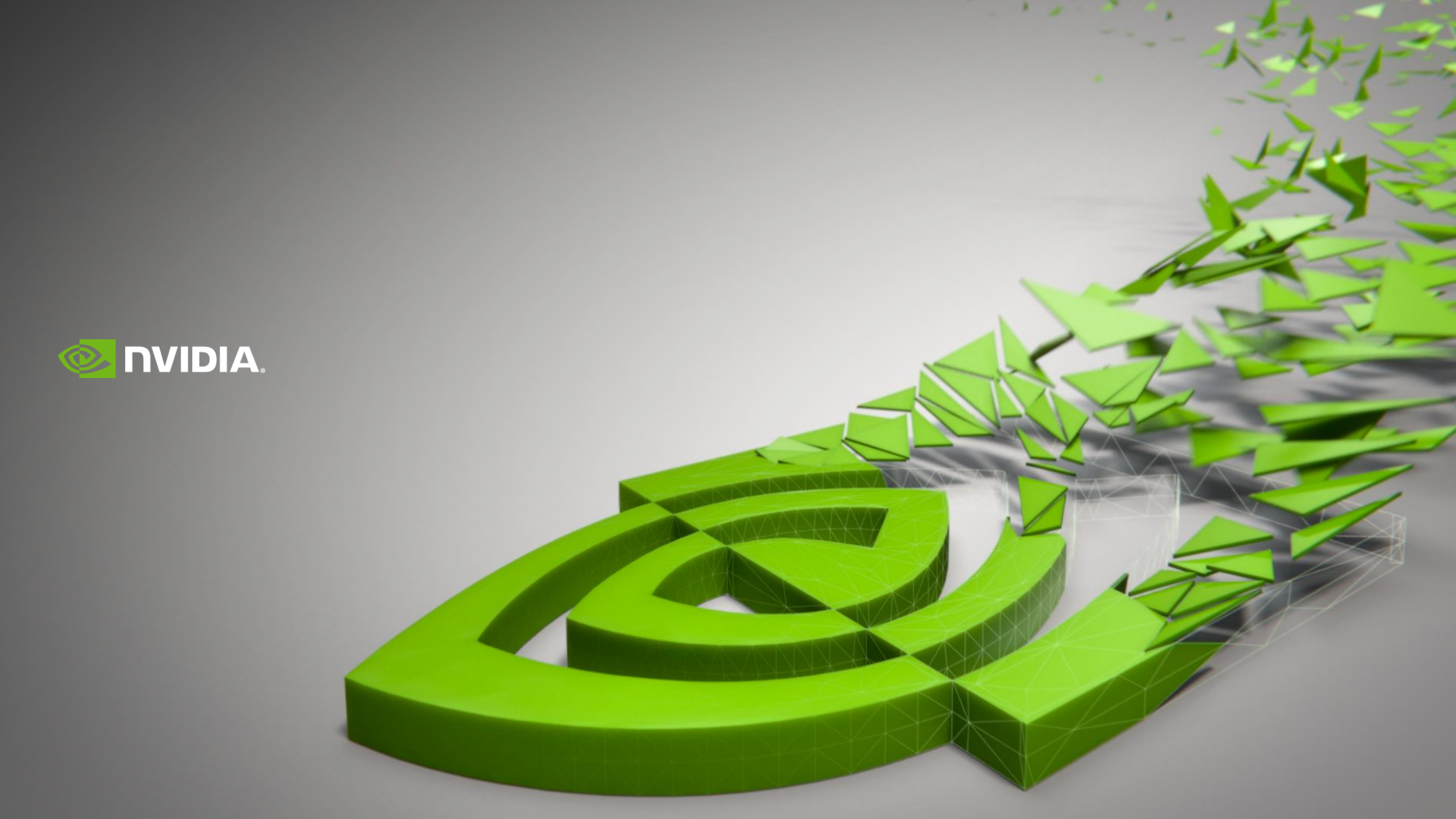
CUDA resource center: http://docs.nvidia.com/cuda

GTC on-demand: http://on-demand-gtc.gputechconf.com

Parallel Forall blog: http://devblogs.nvidia.com/parallelforall

Self-paced labs: http://nvidia.qwiklab.com

OpenACC hands-on session today 6:30pm - 9:30pm


NVIDIA

UNIFIED MEMORY FUNDAMENTALS

When Is This Helpful?

Quick and dirty algorithm prototyping, focus on the ***compute part*** first

Iterative process with lots of ***data reuse***, migration cost can be amortized

Simplify application debugging, increase your ***productivity***

Irregular or ***dynamic*** data structures, unpredictable access

Large datasets on single GPU, data partitioning between multiple GPUs

PERFORMANCE

Page Granularity Overhead

cudaMallocManaged *alignment*: 512B on Pascal/Volta, 4KB on Kepler/Maxwell

- Too many small allocations will use up many pages

cudaMallocManaged memory is moved at *system page* granularity

- For small allocations more data could be moved than necessary

Solution: use cached allocator or memory pools