UNIFIED MEMORY ON P9+V100

Jeff Larkin, December 04, 2018

UNIFIED MEMORY FUNDAMENTALS

Single pointer

On-demand migration

GPU memory oversubscription

System-wide atomics

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

cpu_func1(data, N);

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

cpu_func3(data, N);

free(data);
```

PERFORMANCE HINTS

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

Similar to cudaMemcpyAsync in CUDA or move_pages in Linux

```
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

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);

free(data);
```

P9+V100 NEW FEATURES

Available since CUDA 9.1:

- NVLINK2: increased migration throughput
- NVLINK2: enabling cache coherency
- Native atomics support for all accessible memory

Now Available:

- cudaMallocManaged will use access counters to guide migrations
- ATS*: GPU can access all system memory (malloc, stack, etc.)

NVLINK2: INCREASED THROUGHPUT

CPU-GPU peak bandwidths:

P8+P100 (2 GPUs per P8): 2xNVLINK1 = 40GB/s

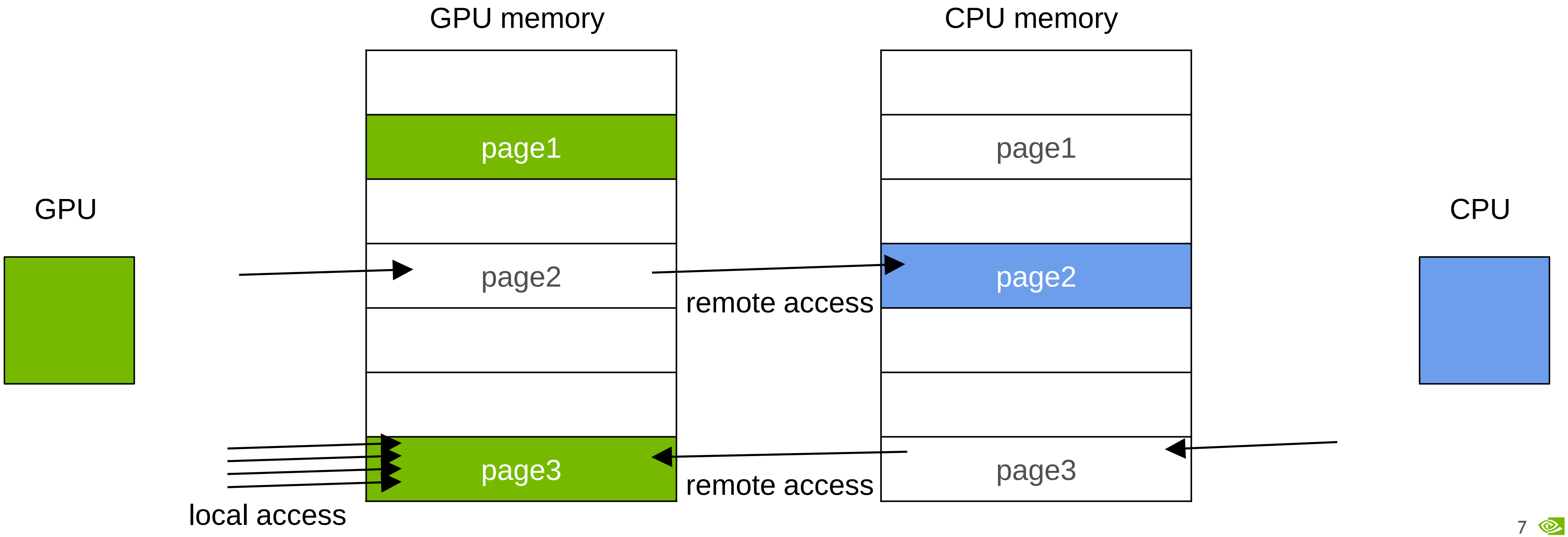
P9+V100 (3 GPUs per P9): 2xNVLINK2 = 50GB/s

P9+V100 (2 GPUs per P9): 3xNVLINK2 = 75GB/s

NVLINK2: CACHE COHERENCY

Works for cudaMallocManaged and malloc

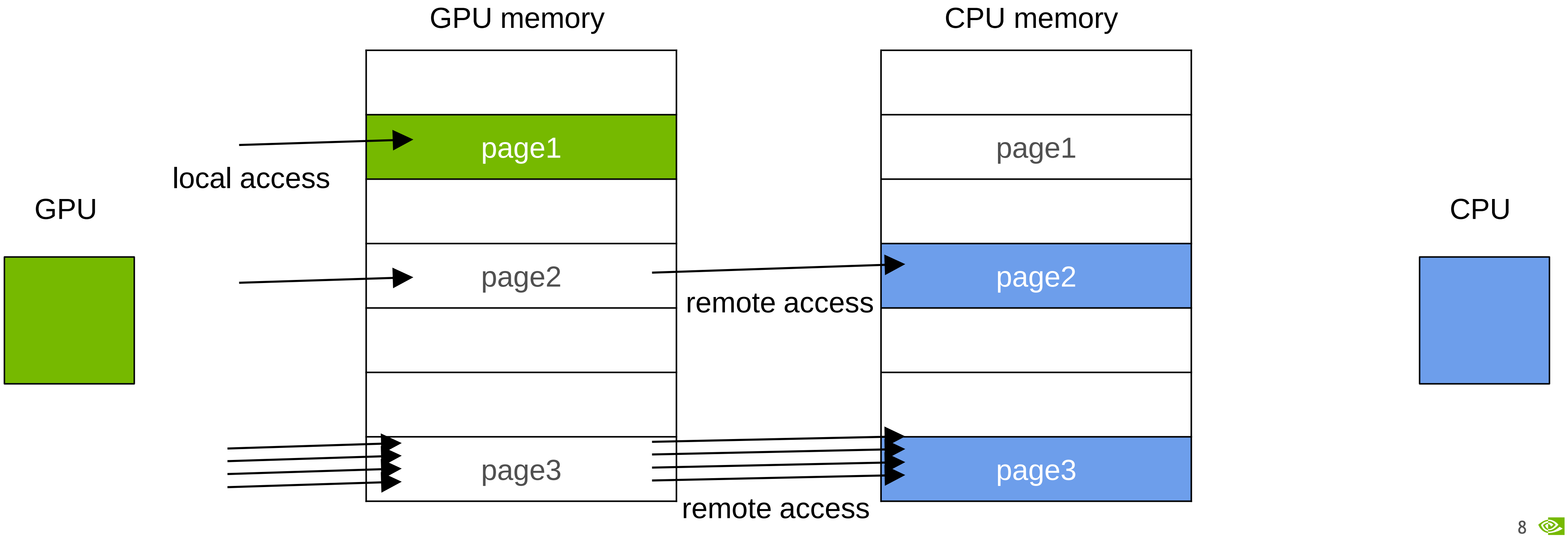
CPU can directly access and cache GPU memory; *native* CPU-GPU atomics



ACCESS COUNTERS

Works only for cudaMallocManaged

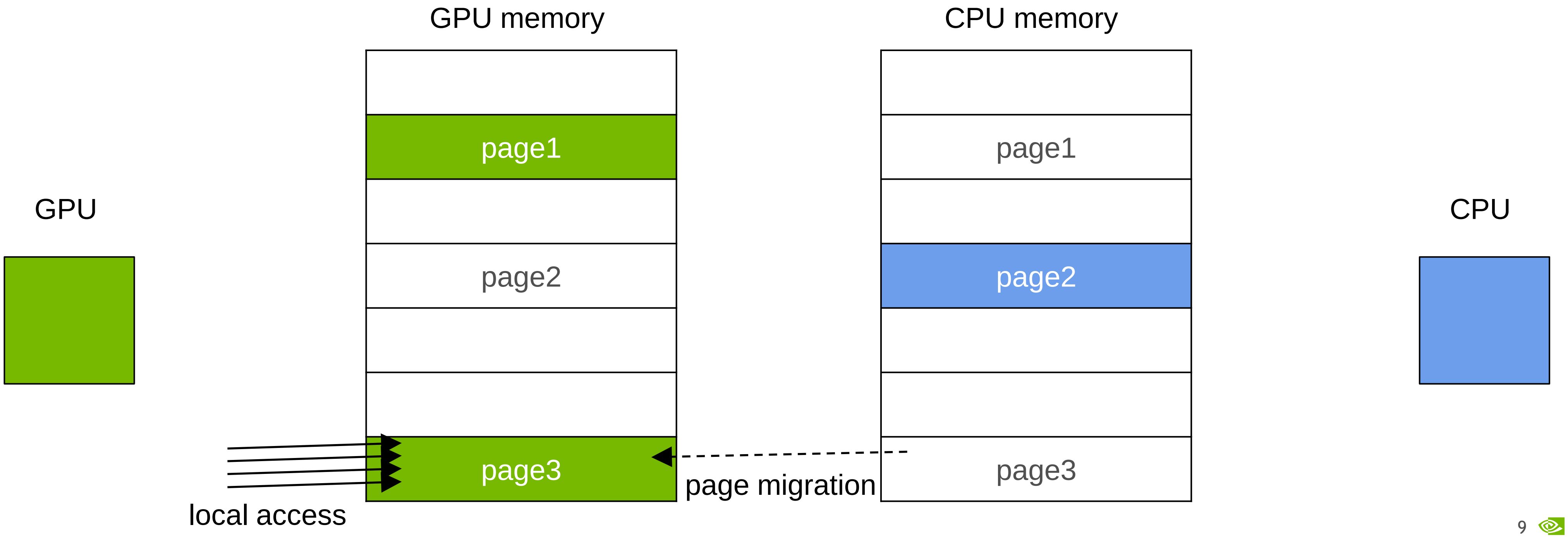
If memory is mapped to the GPU, migration can be triggered by access counters



ACCESS COUNTERS

Works only for cudaMallocManaged

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



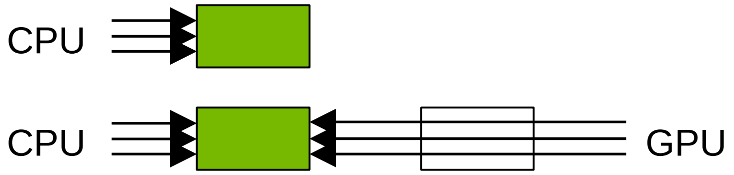
USER HINTS

Accessed By: default behavior

```
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



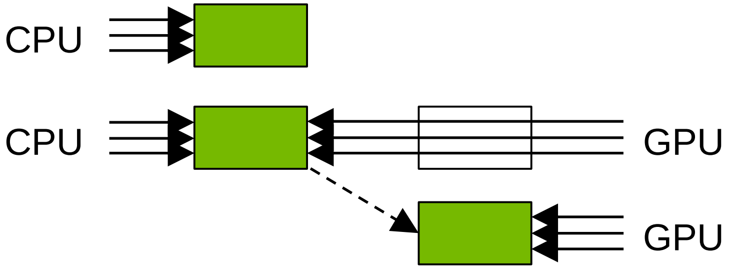
USER HINTS

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 this memory to the GPU



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

FIRST TOUCH

malloc: always allocates on the CPU

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

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

GPU uses ATS, faults
OS allocates on CPU
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

No on-demand malloc data movement except by APIs

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


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


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

USER-DIRECTED MIGRATION

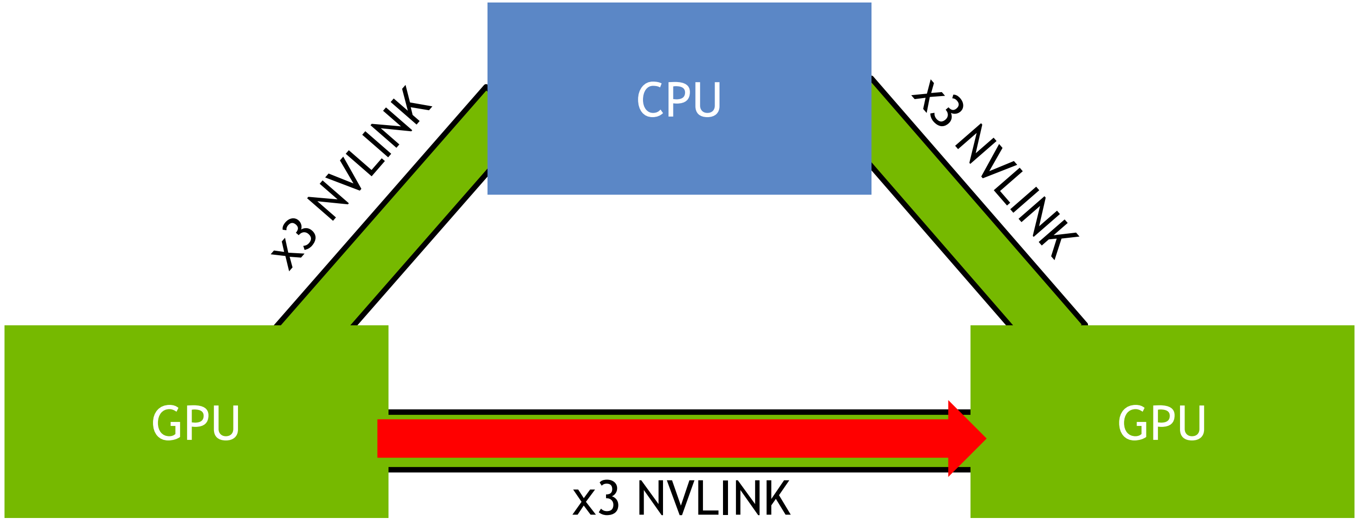
Works for all allocators

```
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)

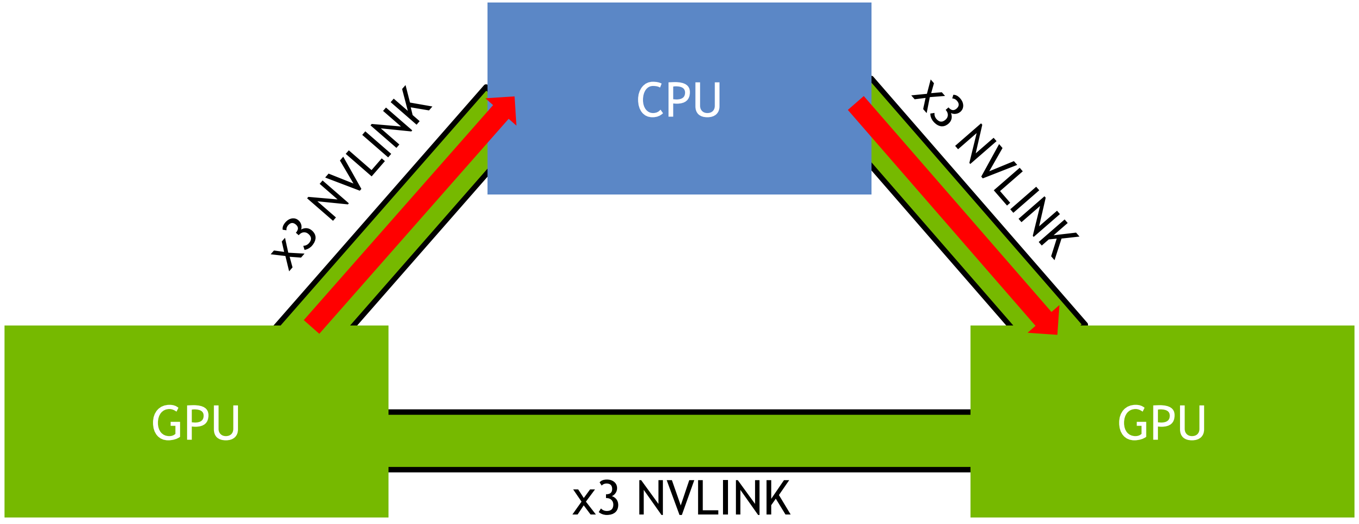


USER-DIRECTED MIGRATION

Performance considerations

malloc: migrations are CPU-directed so they are synchronous and low bandwidth

Linux NUMA migration is single-threaded and cannot saturate even one NVLINK



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

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

Use cudaMallocManaged now with possibility to replace it with system malloc later

Use pooled allocator to alleviate cost of memory allocation

Use perf hints to obtain more predictable memory access patterns and data movement behavior between CPU and GPU