CWE41590 - CUDA MEMORY MANAGEMENT

UNIFIED MEMORY IN 10 MINUTES

UNIFIED MEMORY PREMISE

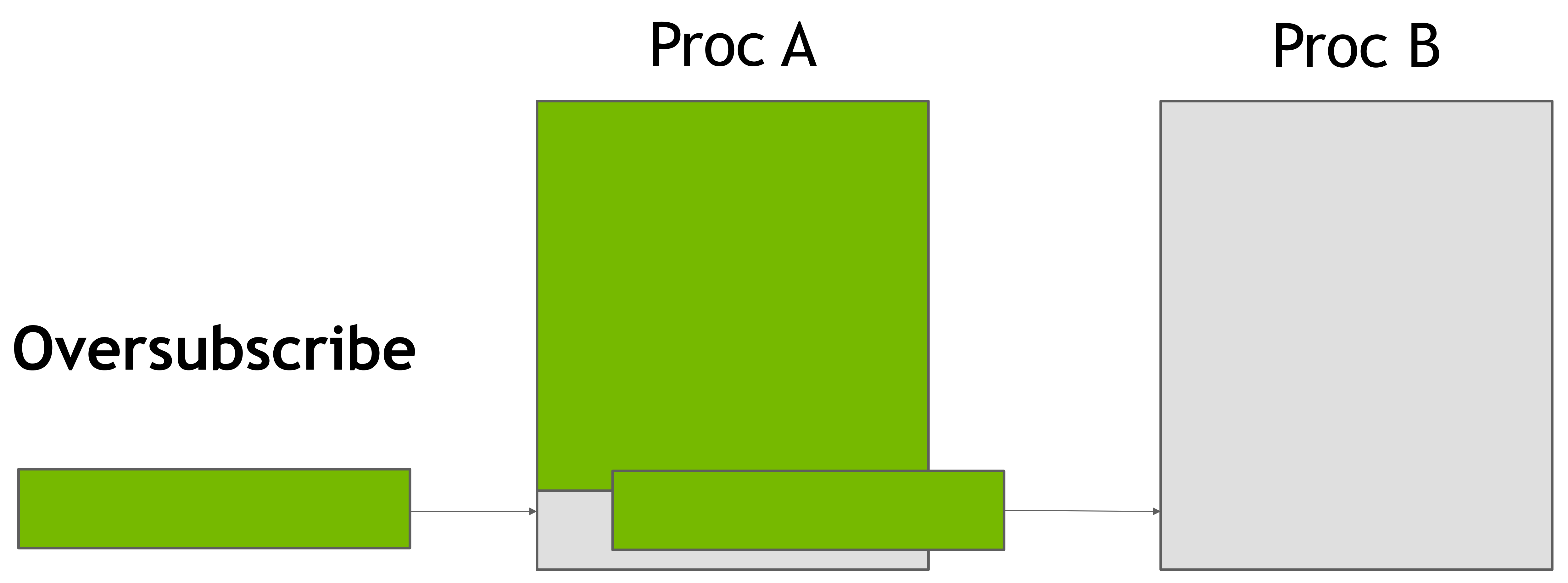
- CUDA memory allocators
  - cudaMalloc/cudaMallocAsync – accessible by GPU only, pinned to single GPU
  - cudaMallocHost/cudaHostRegister – accessible by CPU and GPU, pinned to CPU mem node
  - cuMemCreate/cuMemMap – low-level flexible access and residency controls
  - cudaMallocManaged/malloc – accessible by CPU and GPU, can change residency (“migrate”)
- What does Unified Memory bring to CUDA developers?
  - **Easier CPU/GPU memory management** and **GPU memory oversubscription**

```
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);
```

Explicit Memory Management

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

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

GPU code w/ Unified Memory



UNIFIED MEMORY OPTIMIZATIONS AKA “HINTS”

- **Default:** data *migrates* on access/prefetch

- cudaMemAdvise: advise Unified Memory about the usage pattern for the memory range
  - ReadMostly: data *duplicated* on read/prefetch
  - PreferredLocation: *resist* migrating away from it
  - AccessedBy: establish *direct mapping* / avoid faults

- cudaMemPrefetchAsync: prefetch memory to the specified device

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

init_data(data, N);

cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);

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


use_data(data, N);

cudaDeviceSynchronize();
cudaFree(data);
```

AccessedBy example

← populates data on the CPU

← GPU creates direct mapping

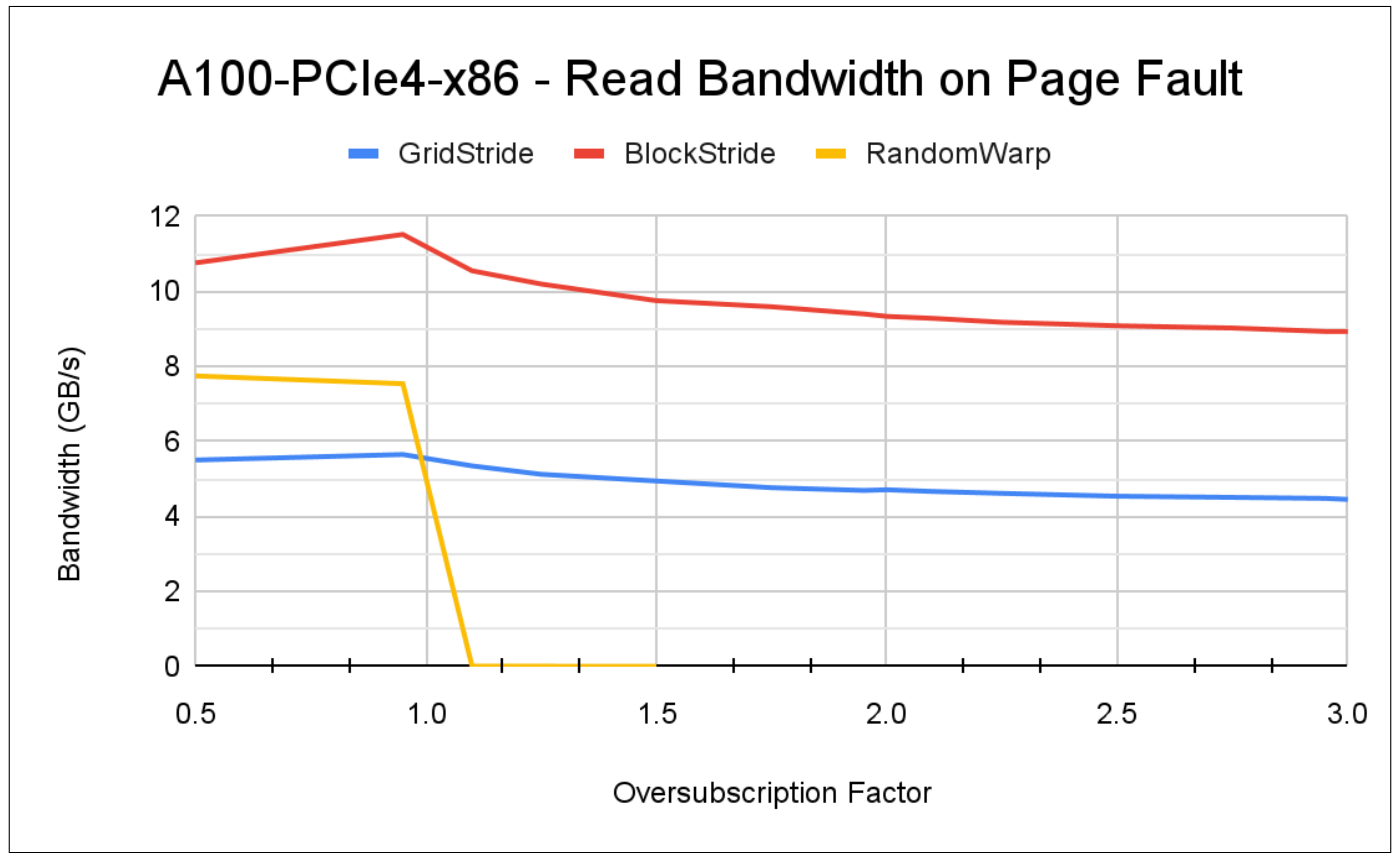
← GPU accesses data remotely without page faults

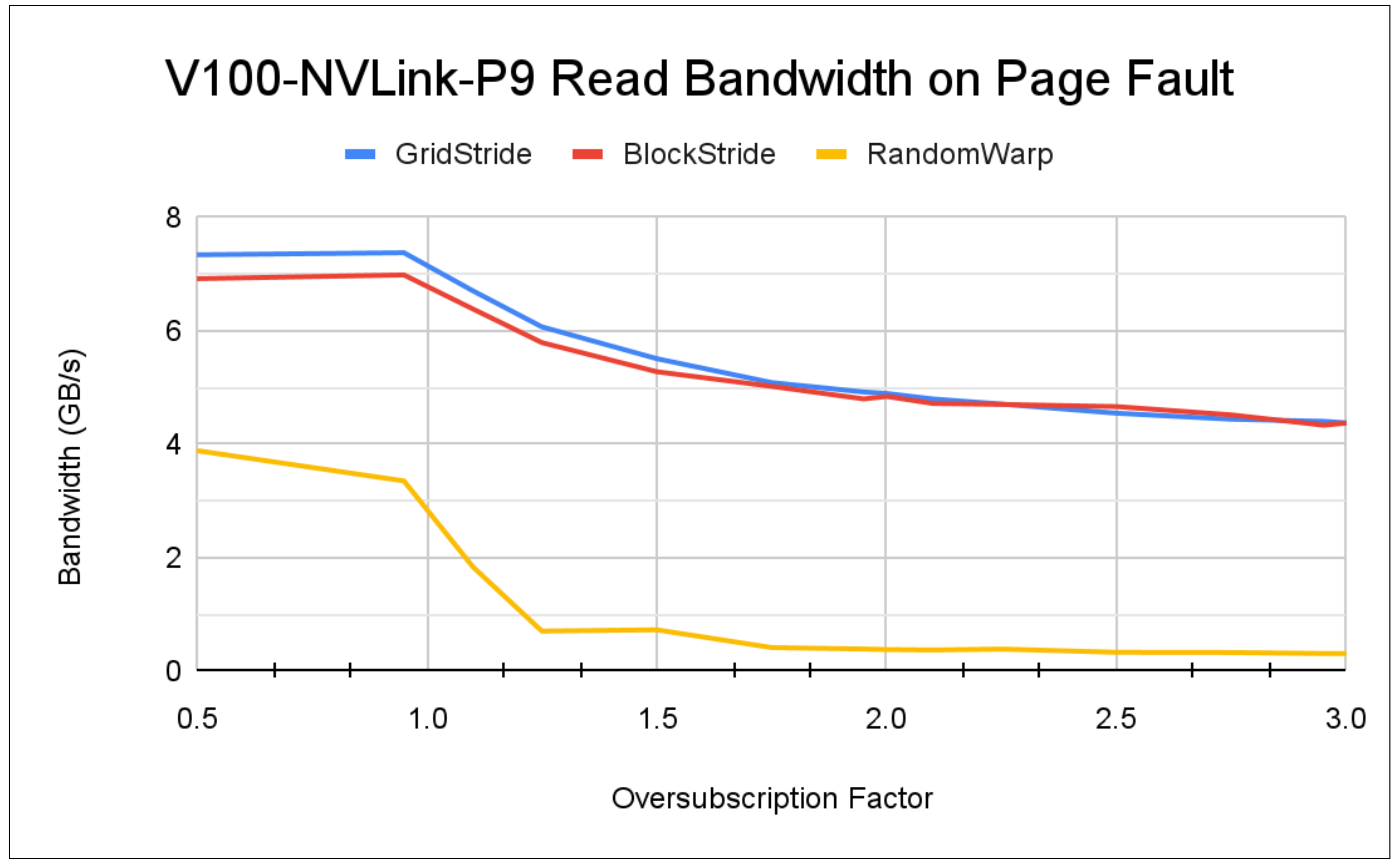
*memory can move freely to other processors and mapping will carry over

UNIFIED MEMORY OVERSUBSCRIPTION PERFORMANCE

by Chirayu Garg

- Bandwidth varies for page fault triggered memory migrations based on memory access pattern
- Produce more page faults such that PCIe and driver paging routines are utilized optimally
  - Block stride access create more page faults across CTA waves on SMs
  - Power9 has 64KB system pages therefore the performance difference between grid and block stride is marginal
- Random memory access result in page thrashing due to memory migration/eviction

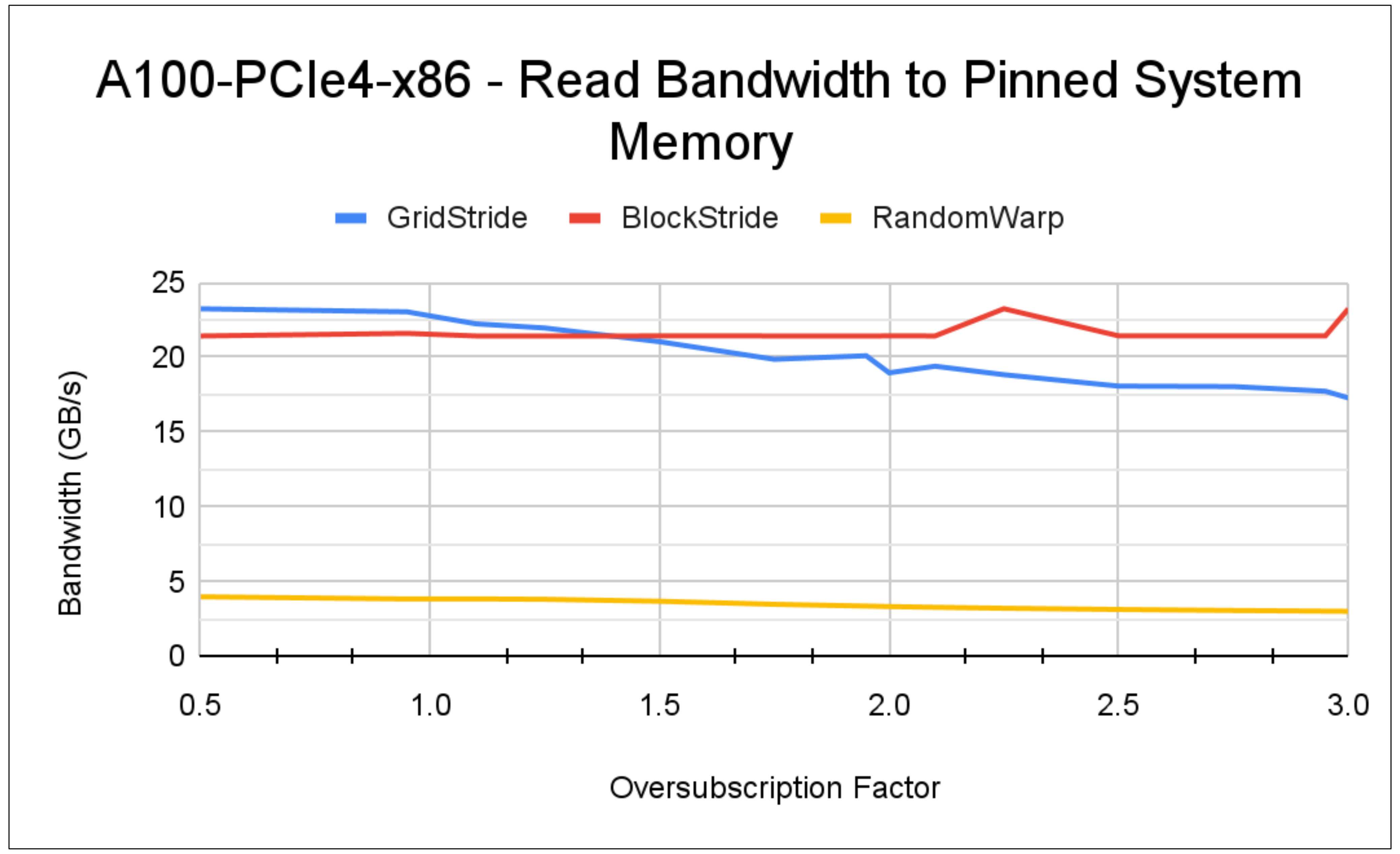


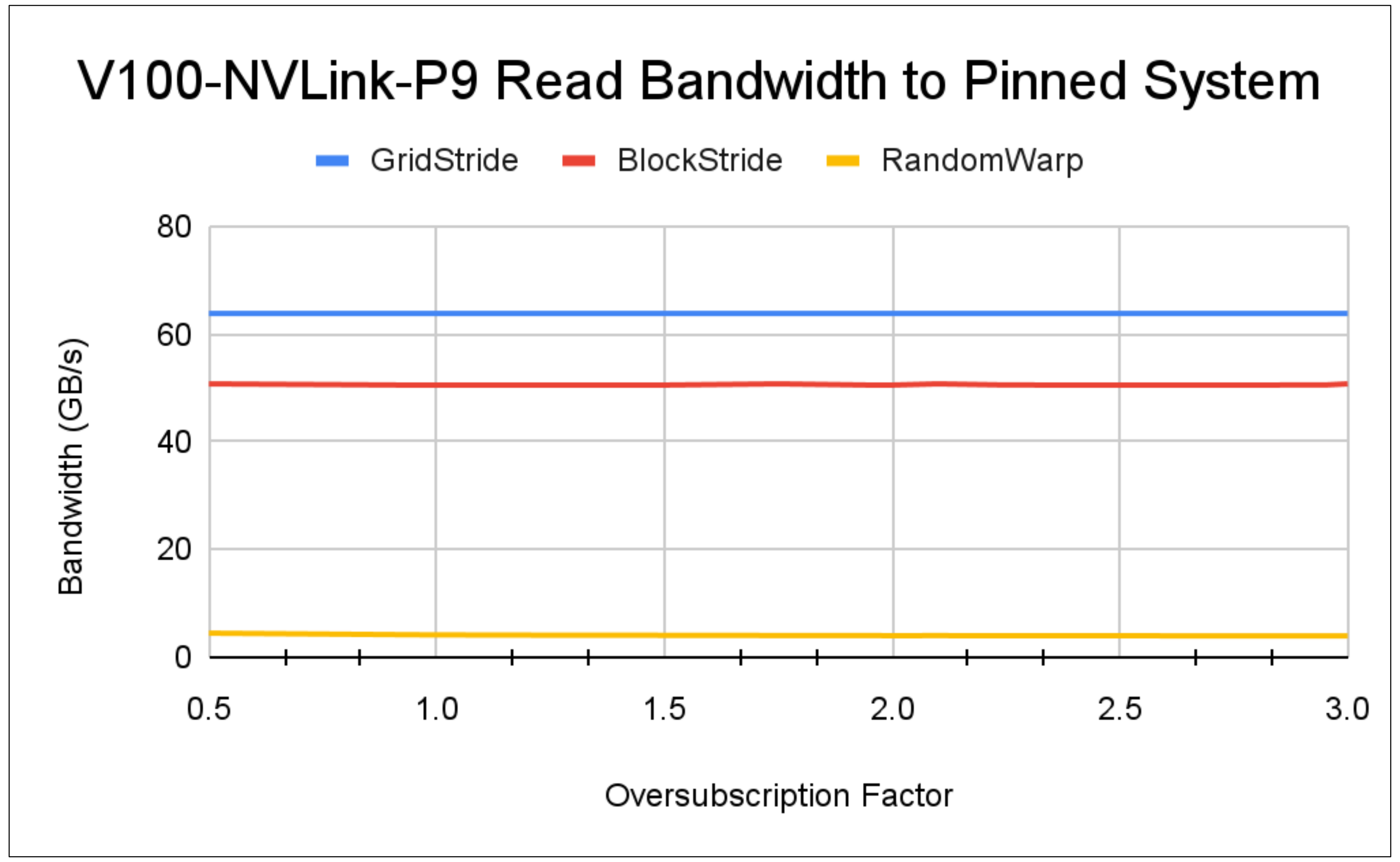


https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/

UNIFIED MEMORY OVERSUBSCRIPTION PERFORMANCE

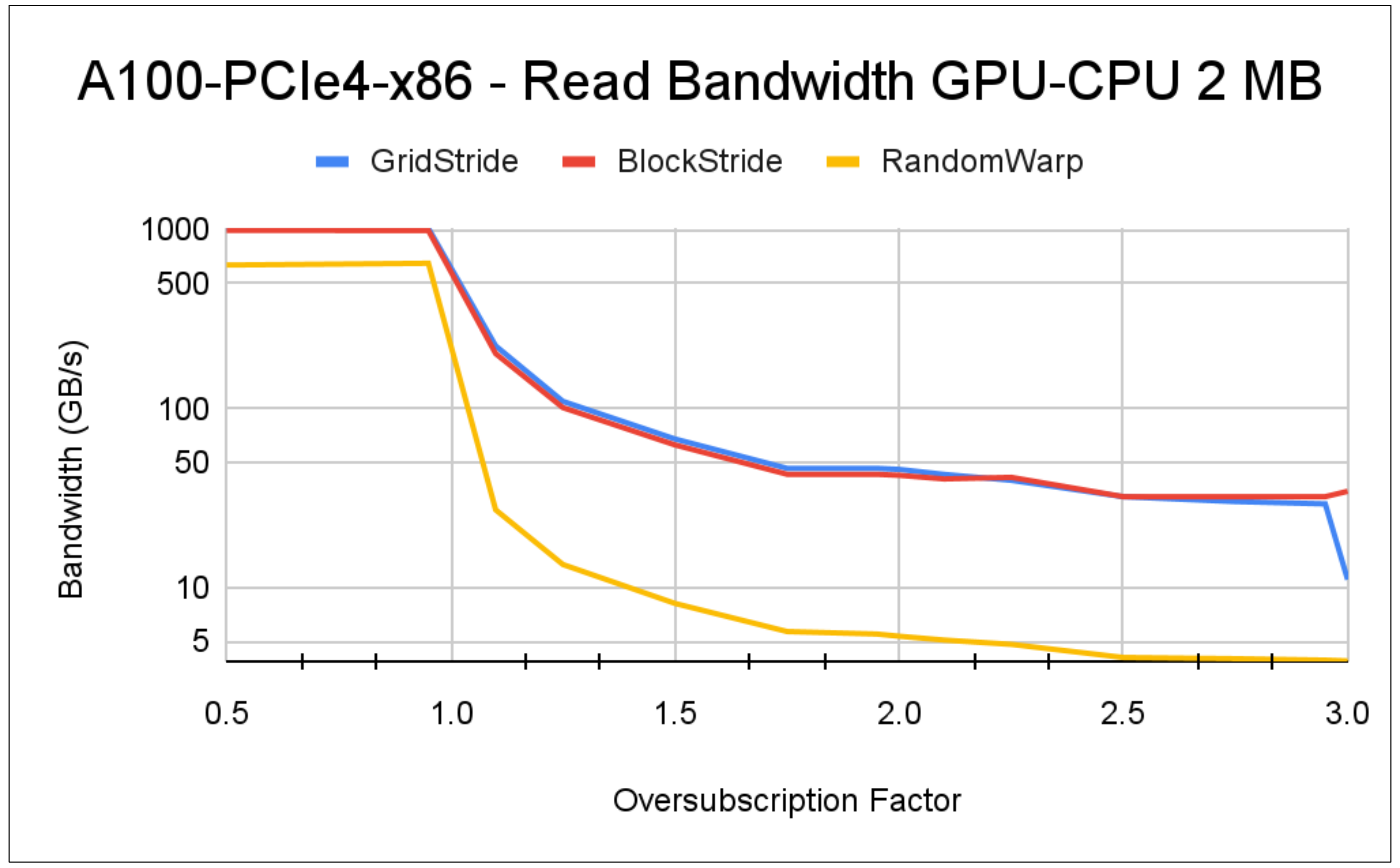
- Use `SetPreferredLocation` and `SetAccessedBy` hints to pin unified memory pages to sysmem and access directly from GPU
- Delivers consistent bandwidth throughout the range of memory allocation
- Align warp memory read request to 128-byte for efficient PCIe and DRAM page transfers

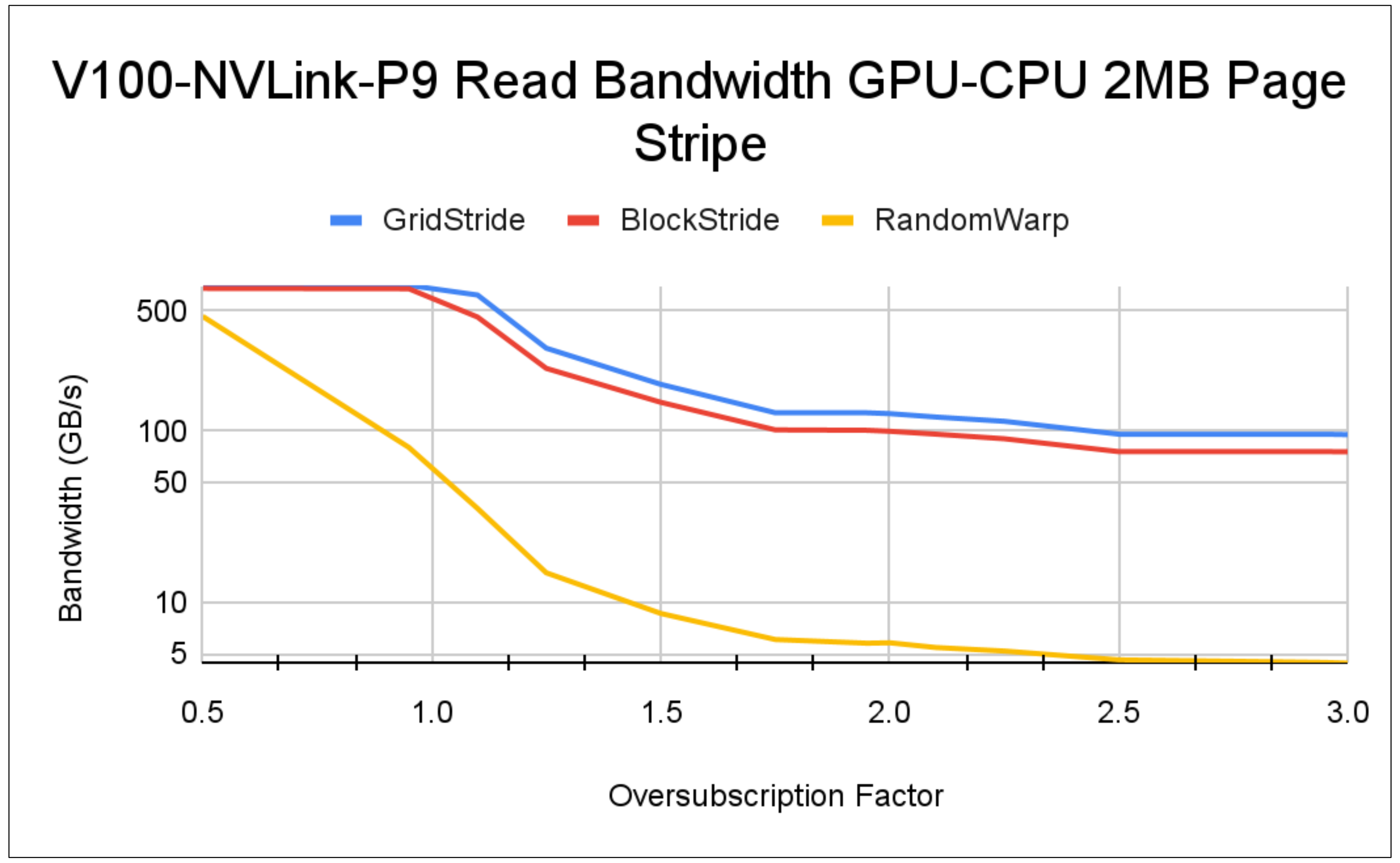




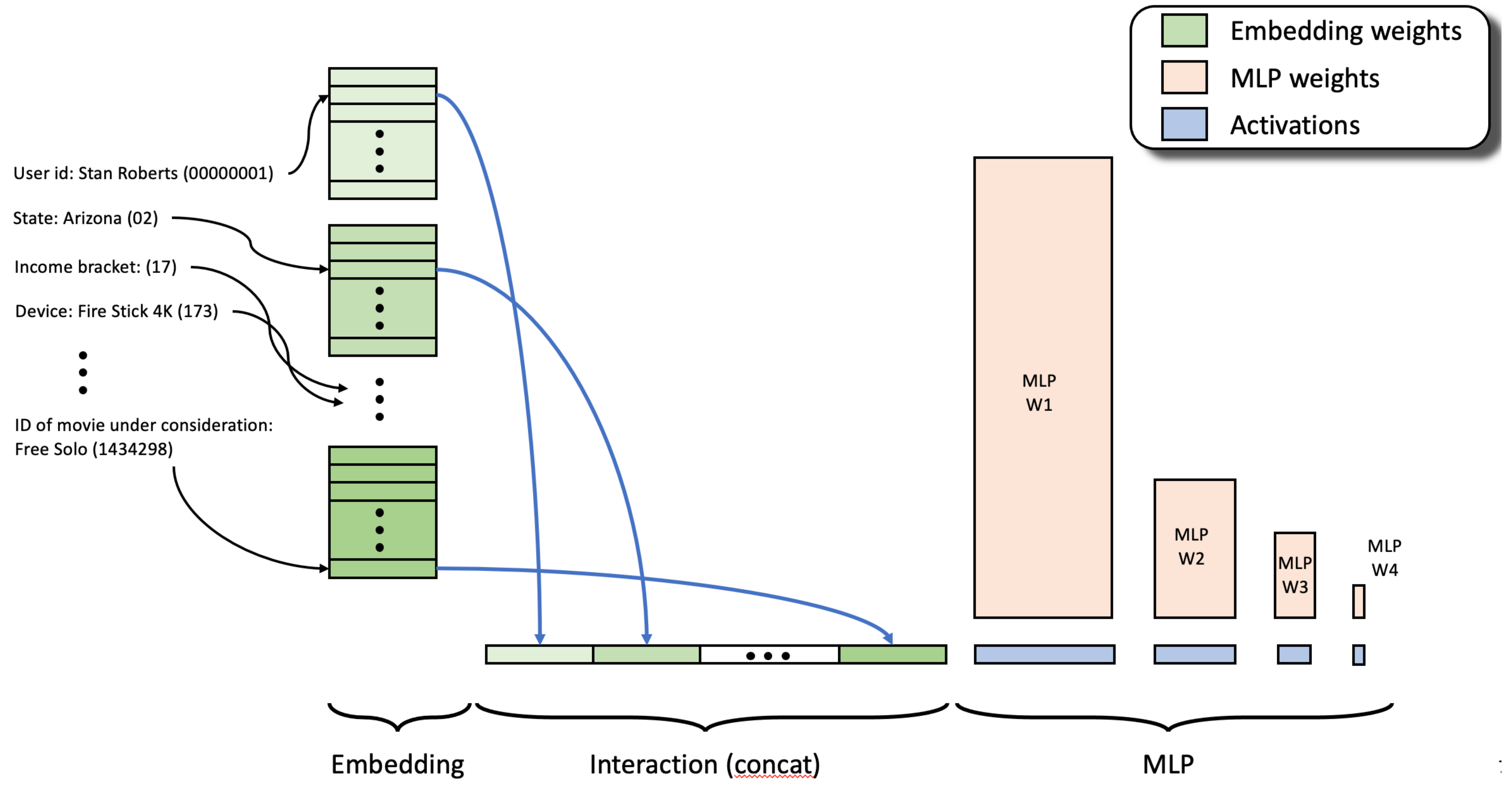
UNIFIED MEMORY OVERSUBSCRIPTION PERFORMANCE

- Distribute memory pages between GPU and CPU in round-robin order
- Oversubscription ratio determines the page location
  - At 1.5 oversubscription every third page is pinned to CPU
- \`SetPreferredLocation\` and \`SetAccessedBy\` hints to pin unified memory pages to sysmem and access directly from GPU
- Higher memory bandwidth is achieved for all the cases as compared to pinned system memory





RECOMMENDERS USE CASE



- **Key idea:** offload infrequently accessed embeddings to CPU memory

RECOMMENDERS USE CASE

- Offload infrequently accessed embeddings to CPU memory
- Use application managed access count (AMAC)
  - Read small portion of dataset to count access of embedding chunks (2MB by default)
  - Use cudaMemPrefetchAync and cudaMemAdvise to allocate most frequently accessed portion of embedding on GPU, the rest on CPU

Example:

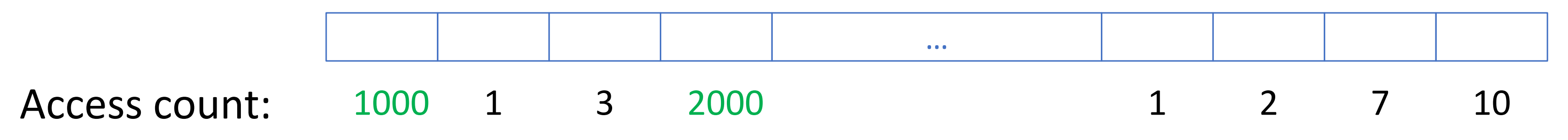
Count access of each embedding chunk (2MB) based on input data

RECOMMENDERS USE CASE

- Offload infrequently accessed embeddings to CPU memory
- Use application managed access count (AMAC)
  - Read small portion of dataset to count access of embedding chunks (2MB by default)
  - Use cudaMemPrefetchAync and cudaMemAdvise to allocate most frequently accessed portion of embedding on GPU, the rest on CPU

Example:

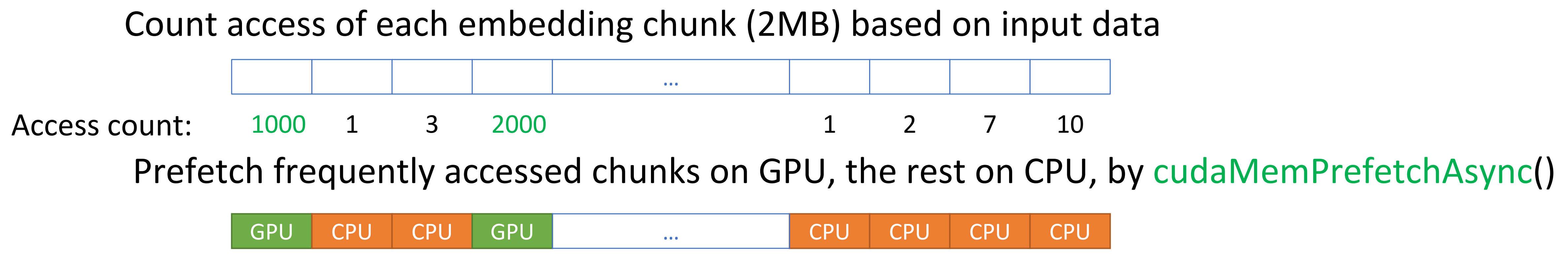
Count access of each embedding chunk (2MB) based on input data



RECOMMENDERS USE CASE

- Offload infrequently accessed embeddings to CPU memory
- Use application managed access count (AMAC)
  - Read small portion of dataset to count access of embedding chunks (2MB by default)
  - Use cudaMemPrefetchAync and cudaMemAdvise to allocate most frequently accessed portion of embedding on GPU, the rest on CPU

Example:



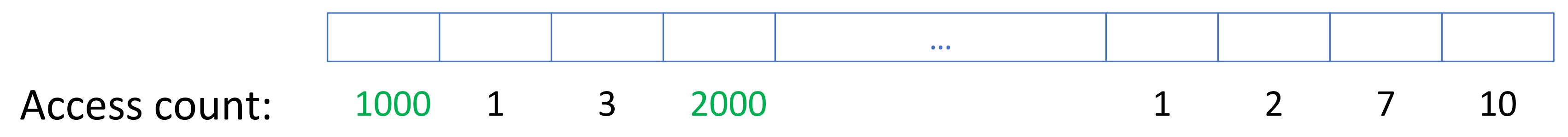
Count access of each embedding chunk (2MB) based on input data
...
Access count:
1000
1
3
2000
1
2
7
10
Prefetch frequently accessed chunks on GPU, the rest on CPU, by cudaMemPrefetchAsync()
GPU
CPU
CPU
GPU
...
CPU
CPU
CPU
CPU

RECOMMENDERS USE CASE

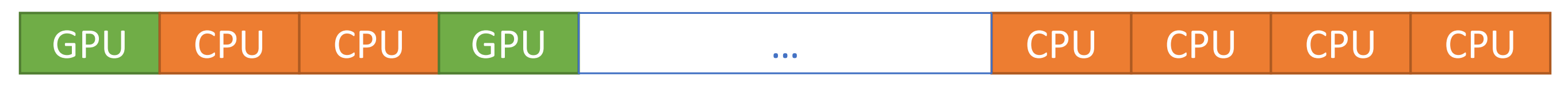
- Offload infrequently accessed embeddings to CPU memory
- Use application managed access count (AMAC)
  - Read small portion of dataset to count access of embedding chunks (2MB by default)
  - Use cudaMemPrefetchAync and cudaMemAdvise to allocate most frequently accessed portion of embedding on GPU, the rest on CPU

Example:

Count access of each embedding chunk (2MB) based on input data



Prefetch frequently accessed chunks on GPU, the rest on CPU, by cudaMemPrefetchAsync()



Set hints by cudaMemAdvise()
- Optional: set cudaMemAdviseSetPreferredLocation for each chunk
- Set cudaMemAdviseSetAccessedBy GPU on the entire region

Pytorch implementation example:

```
embedding = ConcatEmbedding(feature_sizes, FLAGS.embedding_width, uvm=True)

for i, (_, categorical_features, _) in enumerate(data_loader):
    if i < count_access_steps:
        embedding.count_access(categorical_features)
    else:
        embedding_out = embedding(categorical_features)

    if i == count_access_steps - 1:
        embedding.migrate(FLAGS.ratio)
```

RECOMMENDERS USE CASE

- Offload infrequently accessed embeddings to CPU memory
- Use application managed access count (AMAC)
  - Read small portion of dataset to count access of embedding chunks (2MB by default)
  - Use cudaMemPrefetchAync and cudaMemAdvise to allocate most frequently accessed portion of embedding on GPU, the rest on CPU

- Performance
  - MLPerf model (~100GB embedding)
    - **28 minutes on single A100** with Unified Memory. Intel’s best was **45 min** on 4 quad sockets machines.
    - **4.1 minutes on 8xA100** (v0.7 Pytorch submission. HugeCTR was 3.3 min)
  - A 1TB embedding model
    - 2 hours on one DGXA100 with Unified Memory
    - 18 minutes on 4xDGXA100 (Selene)

TAKEAWAY

- Understand Unified Memory benefits and differences vs other allocators
- Use hints to improve Unified Memory performance
- Memory oversubscription is hard, Unified Memory takes care of that for you, but beware of pitfalls
- Recommenders show how Unified Memory can make impact in important GPU workloads

VIRTUAL MEMORY MANAGEMENT IN 10 MINUTES

VIRTUAL MEMORY MANAGEMENT

Goals:

- Provide users explicit control over physical and virtual memory
- Better sharing of memory across processes
- Better interoperability with graphics APIs

VIRTUAL MEMORY MANAGEMENT

API Overview

Physical Memory

cuMemCreate — Allocate physical memory, represent it using a handle.

cuMemRelease — Release a physical memory allocation using its handle.

Virtual Memory

cuMemAddressReserve — Reserve a virtual address range, represented by a device pointer.

cuMemAddressFree — Frees a reserved virtual address range.

Physical & Virtual Memory Association

cuMemMap — Associates a physical allocation with a reserved virtual address range.

VIRTUAL MEMORY MANAGEMENT

API Overview

Set Access Permissions for Devices

cuMemSetAccess — Set the access flags (e.g. memory protection permissions) for each device.

Sharing Memory Between APIs or Processes

cuMemExportToShareableHandle — Create a sharable OS handle for a physical allocation. Can be shared to other processes or APIs.

cuMemImportFromShareableHandle — Import a shareable physical memory resource from other processes or APIs.

Querying Information

cuMemGetAccess – Get the access flags for a given device.

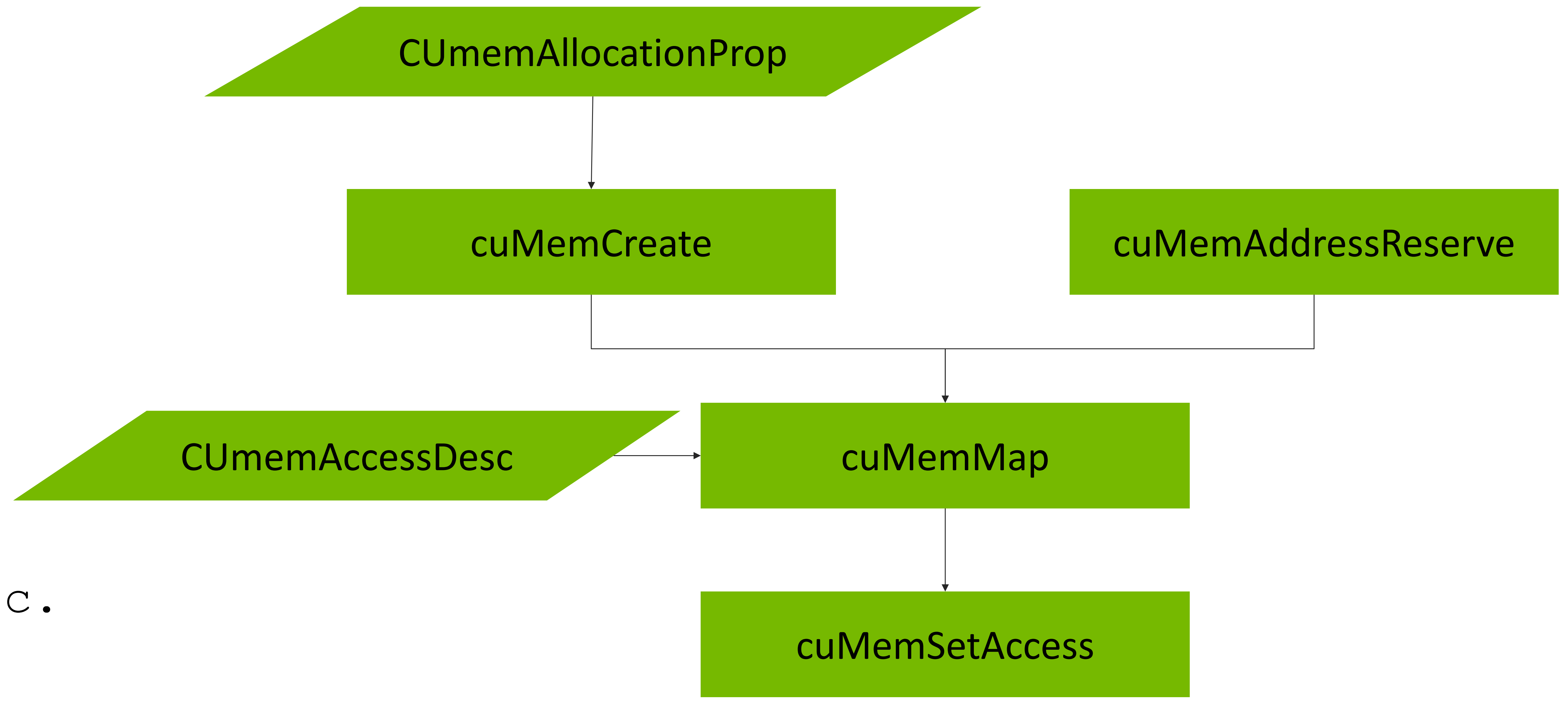
cuMemGetAllocationGranularity — Get the minimum or recommended allocation granularity.

cuMemGetAllocationPropertiesFromHandle — Get the properties that were used to create a physical allocation.

VIRTUAL MEMORY MANAGEMENT

Allocating memory

```
// Describe physical allocation in CUmemAllocationProp.
…

// Allocate physical memory.
cuMemCreate(&memHandle, &size, &allocProp,
            /*flags=*/ 0);

// Reserve a virtual address range.
cuMemAddressReserve(&ptr, size, /*alignment=*/ 0,
                    /*addr=*/ 0, /*flags=*/ 0);

// Map the physical allocation to the reserved
// virtual address range.
cuMemMap(ptr, size, /*offset=*/ 0, memHandle,
         /*flags=*/ 0);

// Describe device access descriptions in CUmemAccessDesc.
…

// Memory not accessible by any device until we grant access.
cuMemSetAccess(ptr, size, deviceAccessDescriptions,
               numDeviceAccessDescriptions)
```



VIRTUAL MEMORY MANAGEMENT
Growing allocations

Allocations with cuMemMap can grow without copying data.

If you have enough adjacent VA:

- Reserve the adjacent VA.
- Allocate the additional physical memory you need.
- Map it & make it accessible.

If you don’t have enough adjacent VA:

- Unmap the old physical allocation from its VA, free the old VA.
- Create a new larger VA reservation.
- Allocate the additional physical memory you need.
- Map the old and new physical allocation into the VA range & make it accessible.

VIRTUAL MEMORY MANAGEMENT

Sharing allocations between processes

New way to share memory allocations between processes: native OS handles.

Allows for data to be shared using traditional OS-based IPC mechanisms.

- Unix domain sockets or Windows DuplicateHandle
- CUDA can now share memory between processes on WDDM.

To share memory between processes:

- Pass your physical memory handle to cuMemExportToShareableHandle, get a native OS handle
- Share that native OS handle between processes just as you would any native OS handle.
- In the receiving process, pass the received OS native handle to cuMemImportFromShareableHandle, get a physical memory handle.
- Reserve a VA, map that physical memory handle to it, and set the access permissions using the usual cuMemMap workflow.

VIRTUAL MEMORY MANAGEMENT

Exporting allocations to graphics APIs

In the past, it was difficult to design a standalone library that would:

- Process data using CUDA
- Allow that data to be visualized outside the library via another graphics API

All allocations needed to be made via the graphics API, then imported to CUDA.

- Why would a number-crunching library want to worry about setting up Vulkan/OpenGL/DirectX?

cuMemExportToShareableHandle makes this an issue of the past.

- Takes a physical memory allocation handle generated by cuMemCreate
- Returns a handle that can be imported by graphics APIs

Allows developers to cleanly separate data processing from graphics and visualization while retaining optimal performance.

VIRTUAL MEMORY MANAGEMENT

Virtual aliasing

CUDA 11.3 introduced support for virtual aliasing

Users can map the same physical frame to multiple virtual addresses

Allows for multiple “views” of the same physical memory

Previously, writes to aliased memory are coherent and consistent with their aliases on kernel boundaries

Practically this means that readers of aliased addresses must be split and synchronized on the writing kernel *before* commencing, which can be quite cumbersome.

Now with new memory proxy PTX instructions, virtual aliases' consistency model is fully expressed without needing to synchronize on kernel boundaries!

See the PTX ISA and CUDA Programming Guide for more information.