Funded by
the European Union

Harnessing Integrated CPU-GPU System Memory for HPC: a first look into Grace Hopper [1]

Gabin Schieffer, Ivy Peng

KTH Royal Institute of Technology, Sweden

✉: {gabins,ivybopeng}@kth.se

[1] Schieffer, G., Wahlgren, J., Ren, J., Faj, J. and Peng, I., 2024, August.
Harnessing Integrated CPU-GPU System Memory for HPC: a first look into Grace Hopper.
In *Proceedings of the 53rd International Conference on Parallel Processing*.

Jülich Supercomputing Centre (JSC), 09 October 2024 – remote presentation

Memory in GPU Applications

Need for large memory capacity in GPU applications:

- AI: large language models (LLMs)
- HPC: quantum computer simulators, particle simulations

Application	Memory
Quantum State Vector simulation, 34 qubits	275 GB
GPT3 inference	350 GB
Meta's LLM (Llama-2-70B) inference	140 GB
Plasma physics (Earth magnetosphere)	250 GB

Table: memory footprint for some GPU workloads

GPU	GPU Memory
Nvidia A100	80 GB
Nvidia H100	94 GB
Grace Hopper	96 GB

Table: memory capacity for Nvidia GPUs

GPU memory capacity: ~100 GB

One solution: use CPU memory (>256 GB) - need for large bandwidth

Explicit CPU↔GPU data movements;

or **Unified Virtual Memory (UVM): single virtual address space**

Grace Hopper Superchip (GH200)

Grace CPU

72 **Arm** Neoverse V2 cores
480 GB of DDR5 memory (500 GB/s)
(128 GB in JEDI)

NVLink-C2C

CPU-GPU interconnect ("chip-to-chip")
Cache-coherent, atomics support
450+450 GB/s

Hopper GPU

144 Streaming Multiprocessors (SM)
96 GB of HBM3 (4 TB/s)

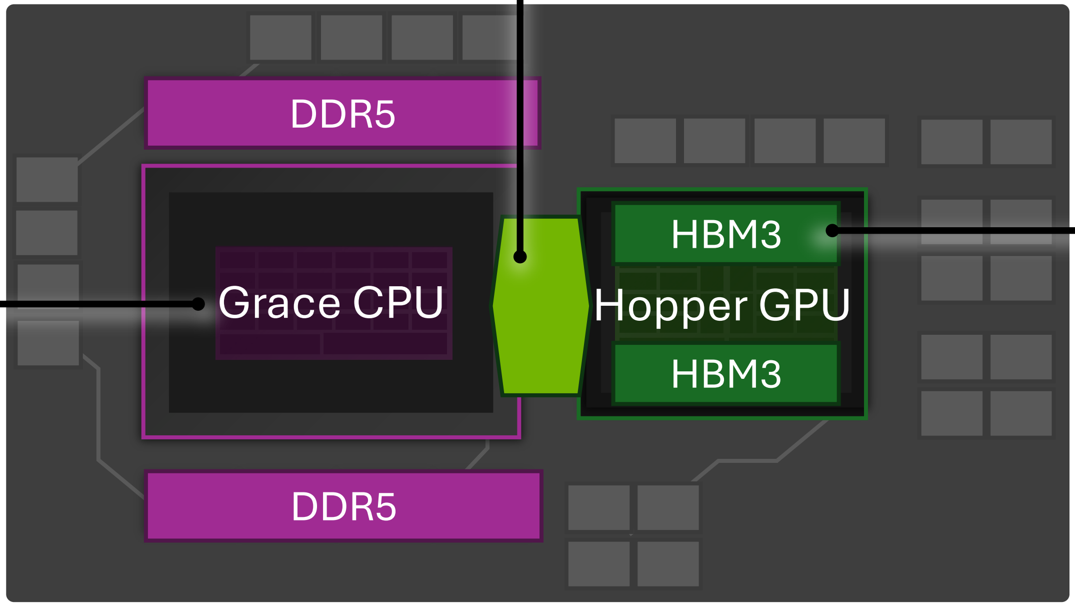
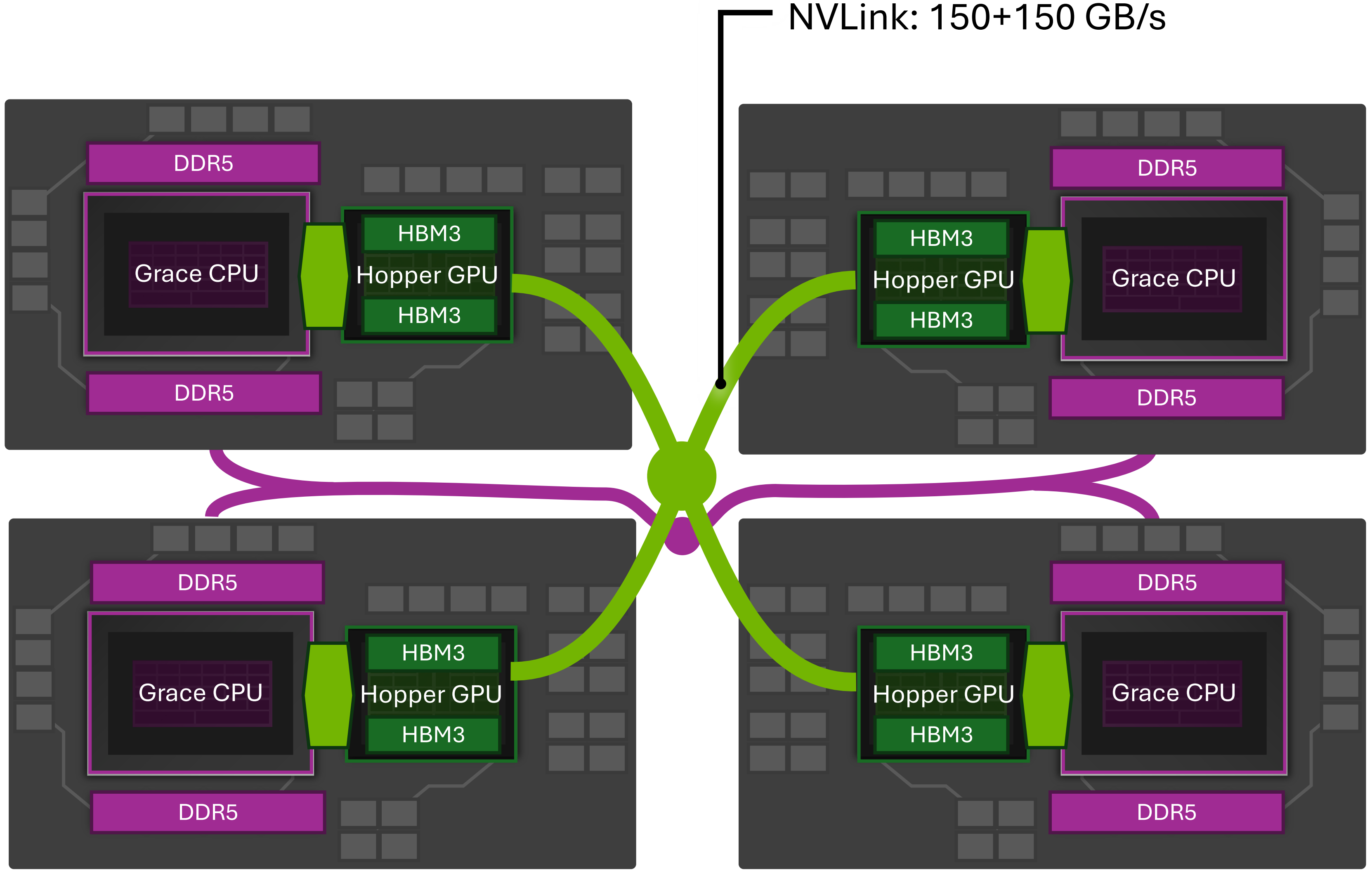


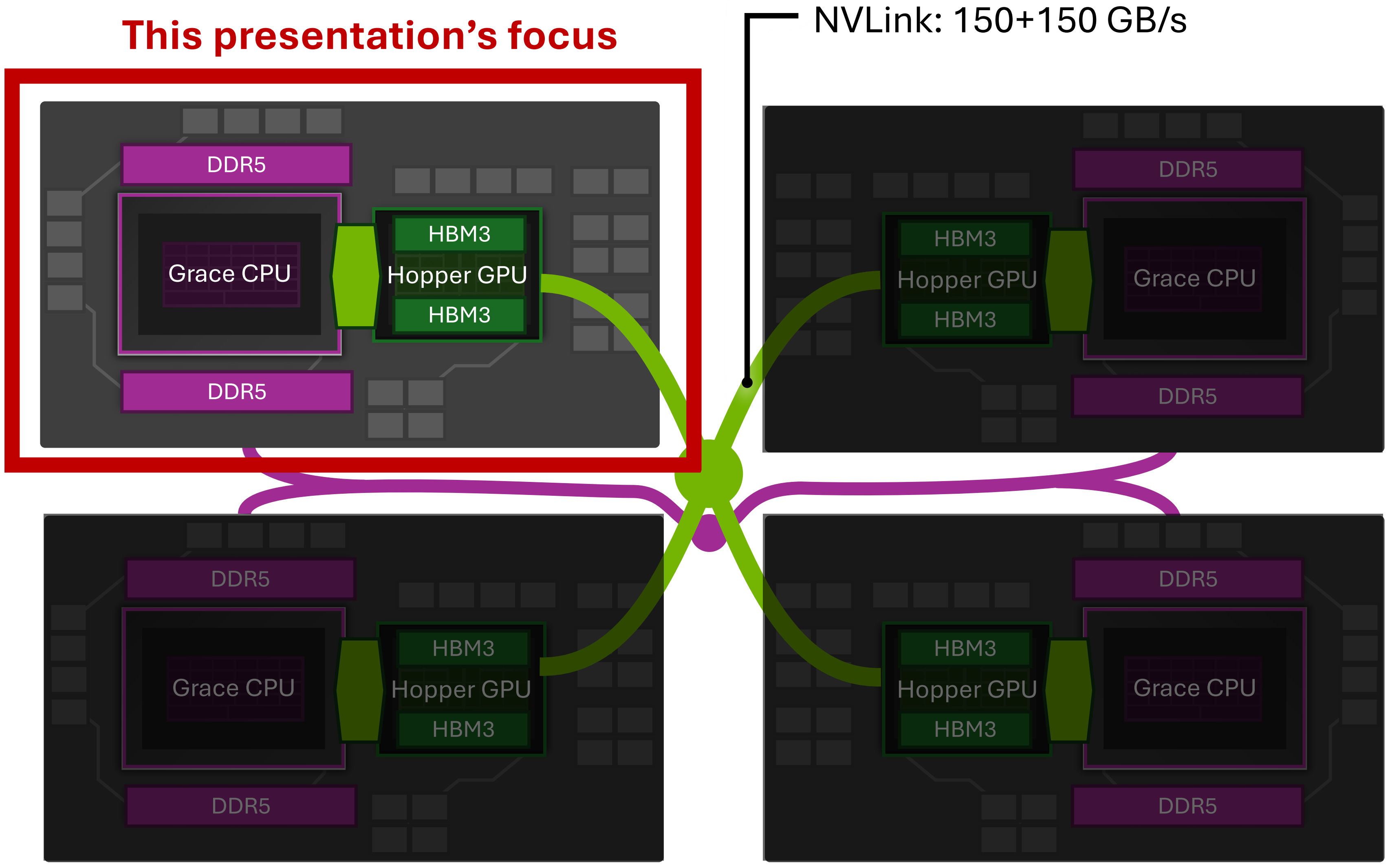
Figure: Grace Hopper Superchip

Context: 1 JEDI node = 4× GH200



+ InfiniBand, + network

Context: 1 JEDI node = 4× GH200



+ InfiniBand, + network

Unified Memory on Grace Hopper

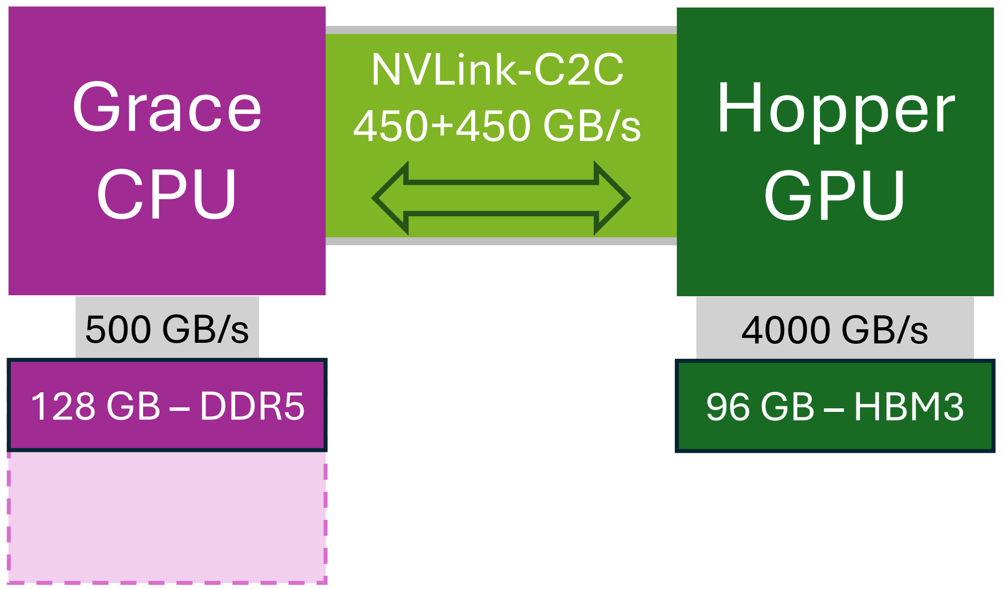
Access to both physical memory regions by any processor: CPU/GPU

⚠ Still two separate physical memory spaces

Thanks to two **hardware** features:

1. System Memory Management Unit (SMMU)
   > manages a system-level page table
   > handles translation requests

2. NVLink-C2C interconnect
   > high-bandwidth direct memory access
   > cache-coherent



Unified Memory on Grace Hopper

Interface	Memory location	Cache coherent	Access mode and granularity
malloc	CPU/GPU	yes	Direct, cacheline
cudaMallocManaged	CPU/GPU	yes	Migration, 2 MB
cudaMalloc	GPU	no	Explicit copy, 1 byte
cudaMallocHost numa_alloc_onnode	CPU	no	Explicit copy, 1 byte

Unified Memory, can be migrated

Unified Memory:

- Data located on CPU or GPU, code is location-agnostic
- The runtime ensure data can be accessed as needed

Unified Memory on Grace Hopper

New On Grace Hopper*

	Managed memory	System memory
Allocation	cudaMallocManaged	malloc numa_alloc_onnode
Page table	CPU page table GPU page table	System-wide
Page size	System (4/64 KB) or 2 MB	System (4/64 KB)
Access Granularity	Page	**Cache line**
Automatic migrations	When accessed + prefetch	Based on access counters
Supported systems	CUDA 6.0	Grace Hopper or *[x86 + Linux 6.1 + CUDA 12.2]

System memory:

- Access at cache line granularity over NVLink-C2C
- Automatic migrations meant to improve locality

Host-to-device bandwidth

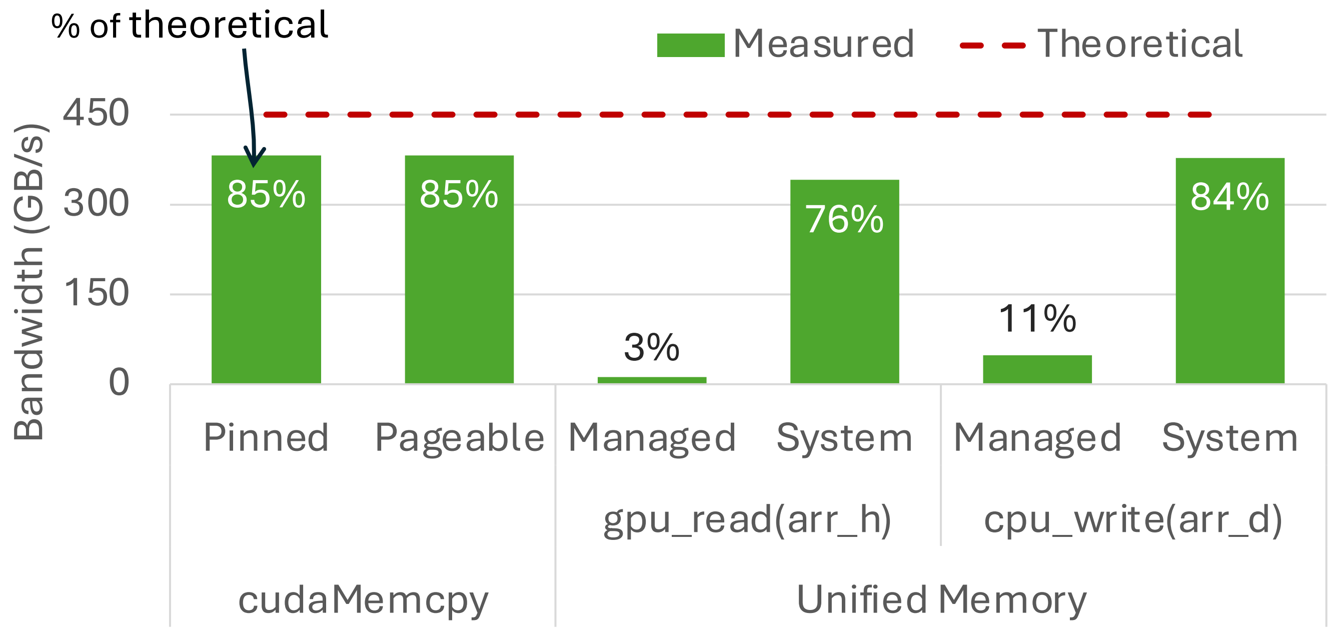


Figure: host-to-device bandwidth, baseline cudaMemcpy and unified memory kernels

cudaMemcpy:

- Pageable: malloc
- Pinned: malloc + cudaHostRegister

Unified Memory: host-to-device bandwidth, with direct access kernels:

- GPU reads from CPU memory
- CPU write to GPU memory

Host-to-device bandwidth

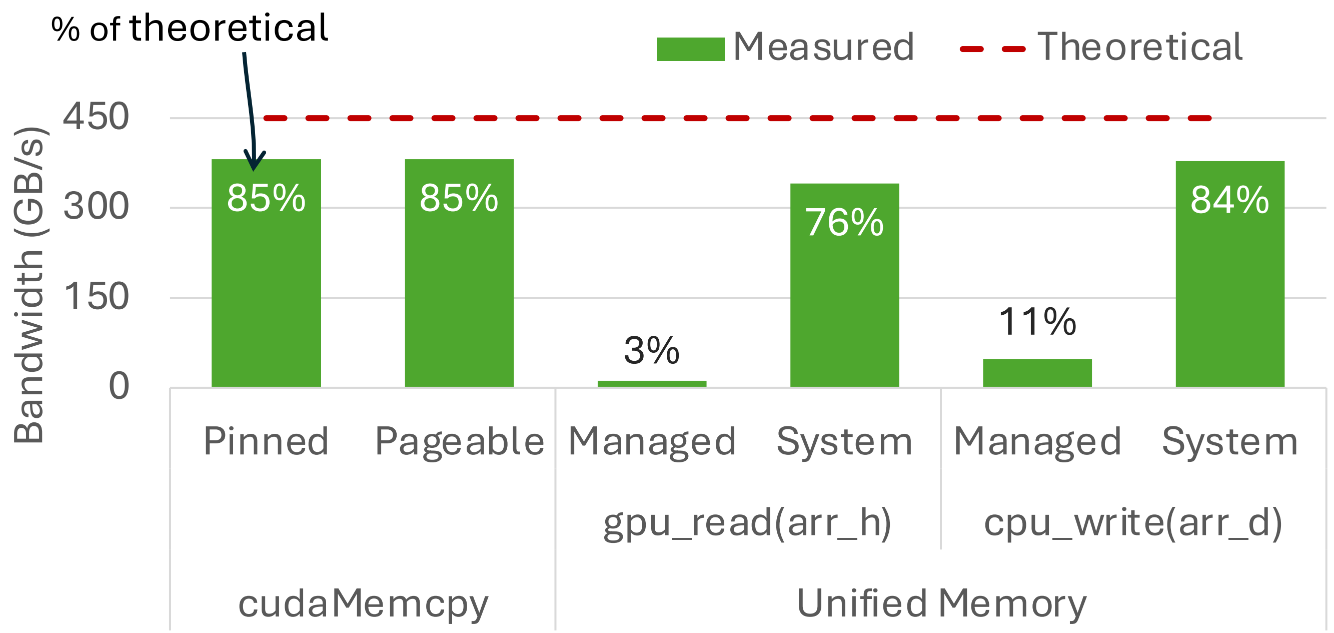


Figure: host-to-device bandwidth, baseline cudaMemcpy and unified memory kernels

① Pinned provides same performance as pageable

Pinning with `cudaHostRegister`:

- before GH: allocate physical pages + page-lock + map in GPU space
- on GH: allocate physical pages + map in GPU space

On GH, not necessary to pin memory.

Host-to-device bandwidth

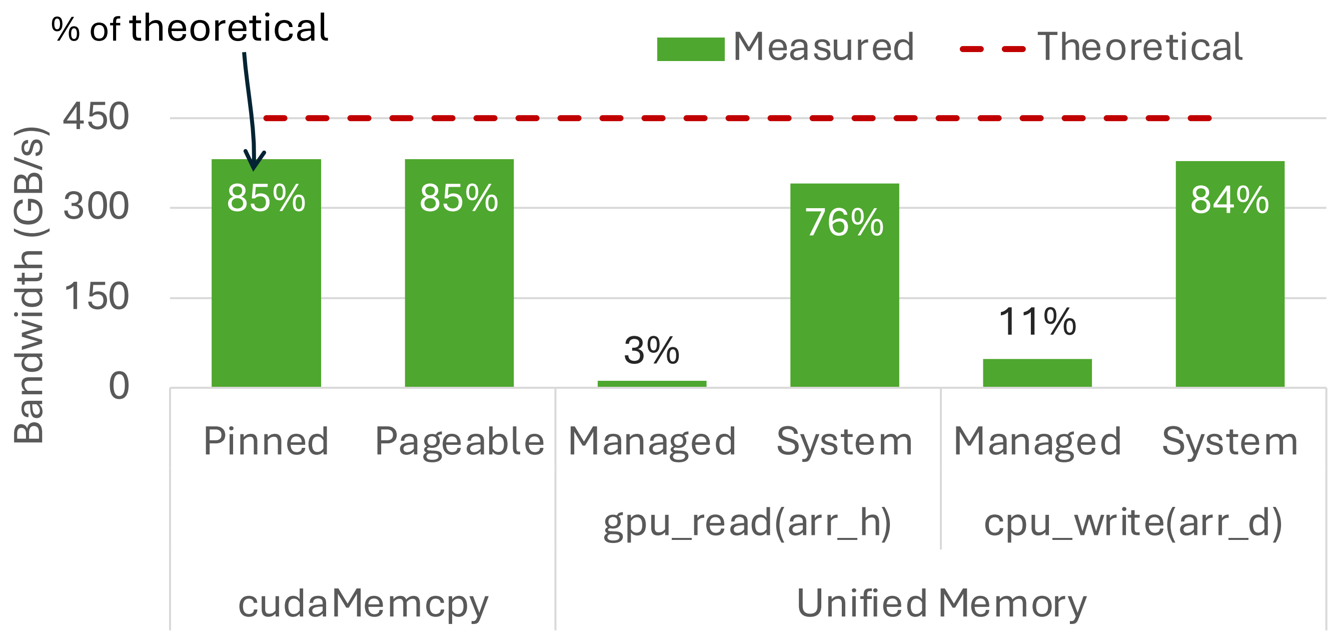


Figure: host-to-device bandwidth, baseline cudaMemcpy and unified memory kernels

② System memory achieves high bandwidth for read/writes over NVLink-C2C

System memory provides low-granularity access to unified, migratable memory.
The achievable bandwidth is on the level of cudaMemcpy.

Device-to-host bandwidth

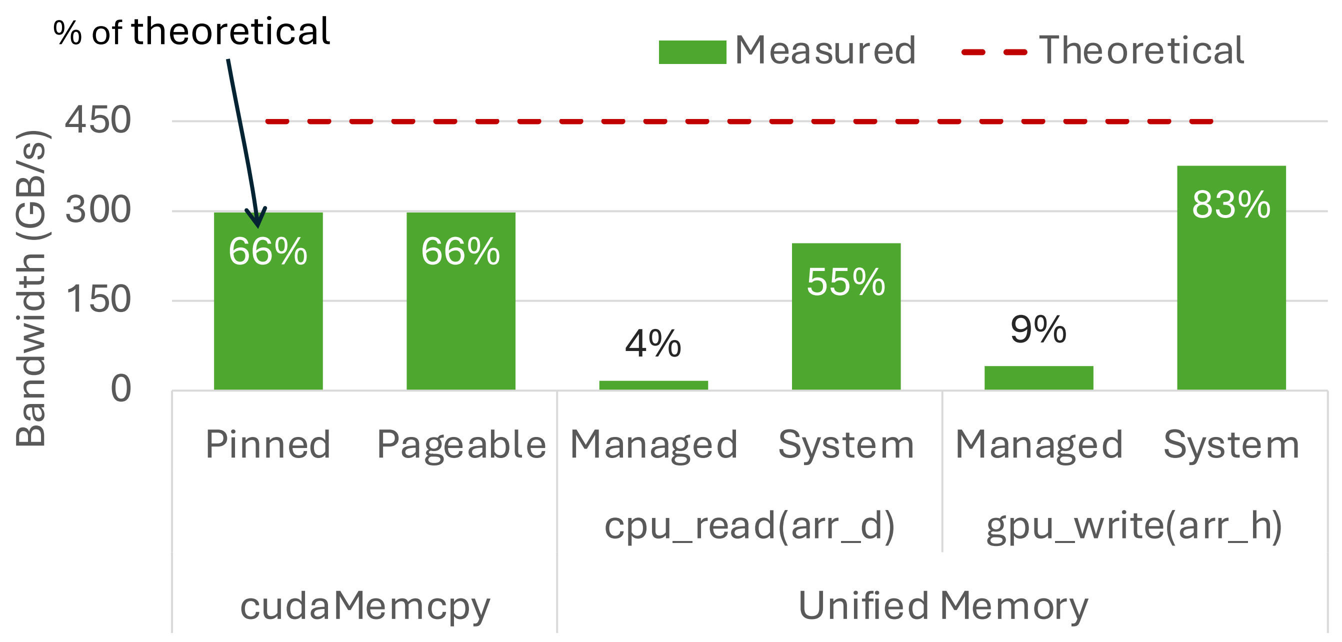


Figure: device-to-host bandwidth, baseline cudaMemcpy and unified memory kernels

Direct memory access can exceed cudaMemcpy performance

Profiling Tools

Identifying the location of physical pages (CPU/GPU)?

For system memory: regular Linux interfaces, NUMA statistics

Memory usage:

- CPU: *Resident set size* (RSS) in `/proc/<pid>/smaps_rollup`
- GPU: *Used memory* from nvidia-smi

$\}$ **regular Linux interface**

Memory traffic, in Nvidia Nsight Compute:

- Traffic to local GPU memory
- Traffic over NVLink-C2C **also available with perf**

GPU kernel Runtime:

- (system memory) data migrations should reduce kernel runtime

Porting to Unified Memory

Simple example:

```
arr_d = cudaMalloc()
arr_h = malloc()
init(arr_h)
cudaMemcpy(arr_d, arr_h, N, H2D)
kernel<<< … >>>(arr_d)
cudaDeviceSynchronize()
cudaMemcpy(arr_h, arr_d, N, D2D)
post_process(arr_h);
```

Direct porting

`arr = arr_h = arr_d = malloc()`

```
arr_d = cudaMalloc()
arr = malloc()
init(arr)
cudaMemcpy(arr_d, arr_h, N, H2D)
kernel<<< … >>>(arr)
cudaDeviceSynchronize()
cudaMemcpy(arr_h, arr_d, N, D2D)
post_process(arr)
```

More complex patterns:

```
[…]
for(int t = 0; t < T; t++)
  cudaMemcpy(arr_d, arr_h, H2D)
  kernel<<< … >>>(arr_d)
  cpu_kernel(arr_h)
```

Concurrent use of arr_d and arr_h!

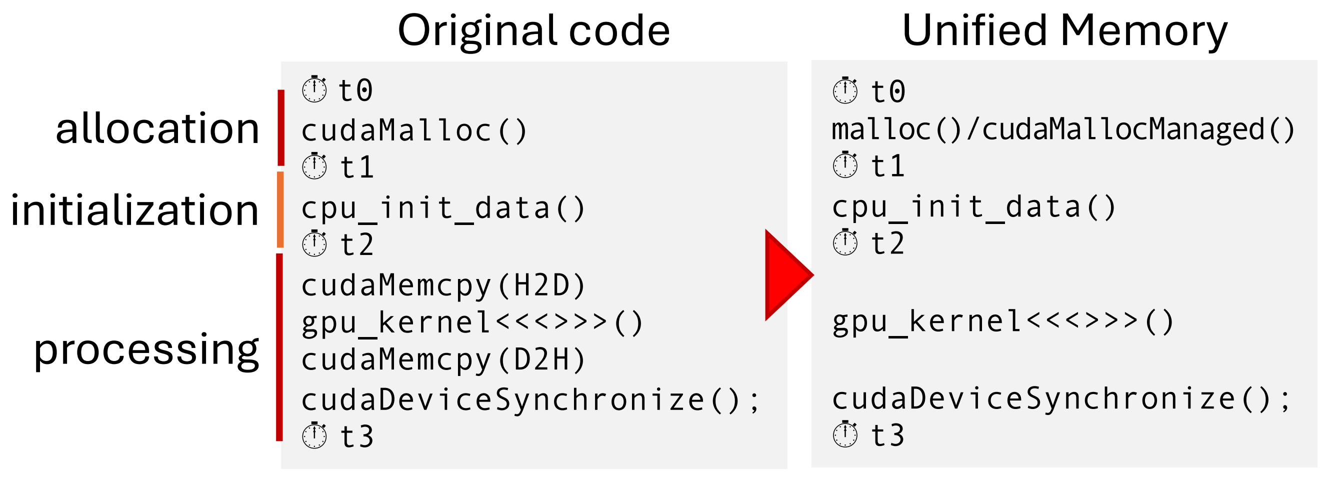
Not directly portable

Porting to Unified Memory

5 Rodinia benchmarks
1 Qiskit Quantum Computer Simulator

Two flavors for each application:

- CUDA managed memory
- System-allocated memory



Original code

allocation, initialization, processing

```
⏱ t0
cudaMalloc()
⏱ t1
cpu_init_data()
⏱ t2
cudaMemcpy(H2D)
gpu_kernel<<<>>>()
cudaMemcpy(D2H)
cudaDeviceSynchronize();
⏱ t3
```

Unified Memory

```
⏱ t0
malloc()/cudaMallocManaged()
⏱ t1
cpu_init_data()
⏱ t2

gpu_kernel<<<>>>()

cudaDeviceSynchronize();
⏱ t3
```

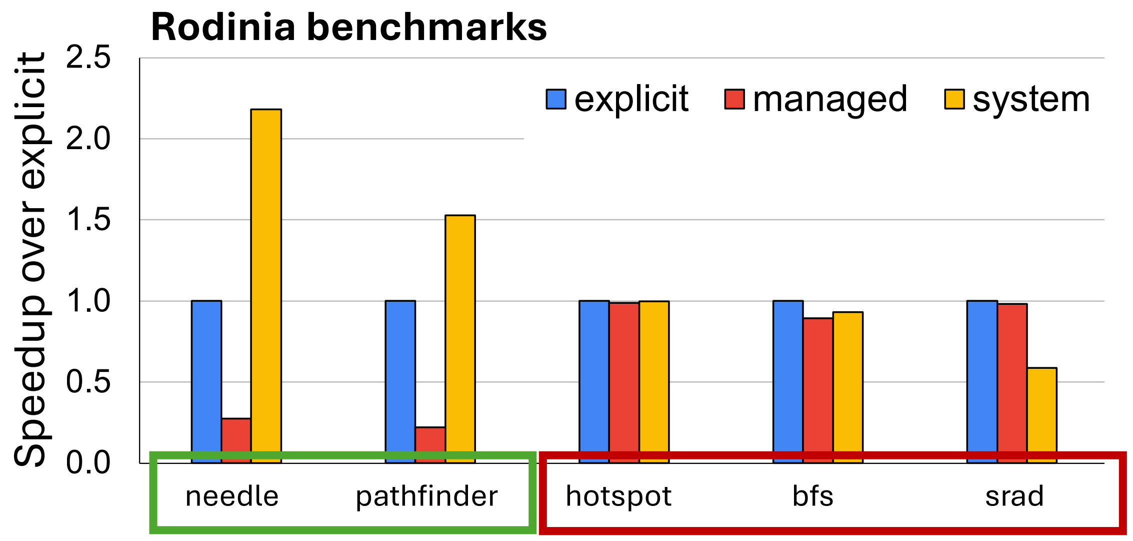
⏱: CPU-timer

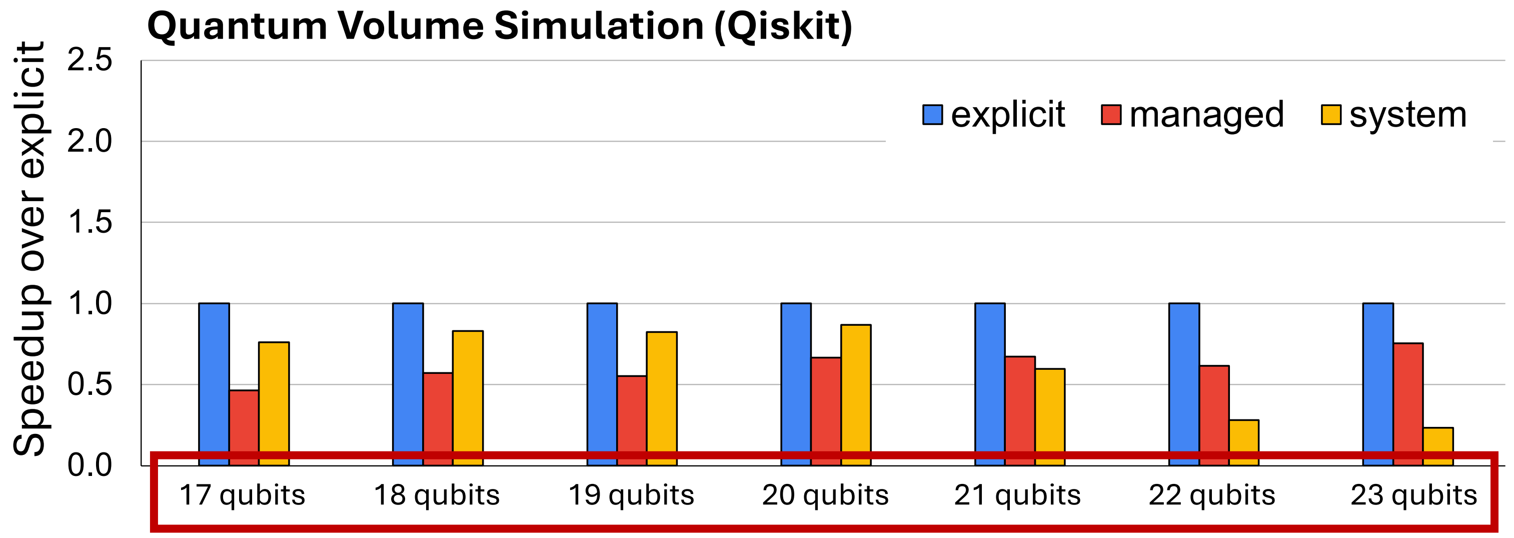
Application	Pattern	Peak Memory
Qiskit	Mixed	275 GB
Needle	Irregular	4.3 GB
Pathfinder	Regular	8.0 GB
BFS	Mixed	1.1 GB
Hotspot	Regular	3.0 GB
SRAD	Irregular	8.9 GB

Rodinia

Table: applications used for evaluation, with their memory access patterns

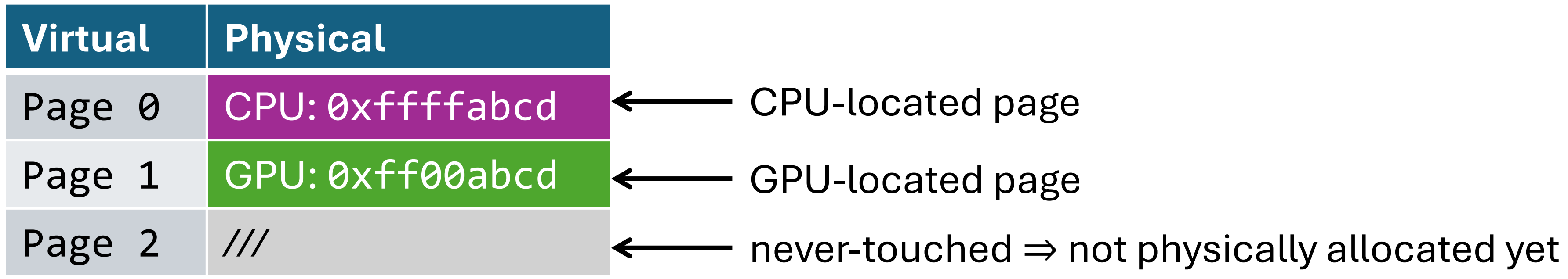
Overall Performance Comparison





First-touch Page Placement – System memory

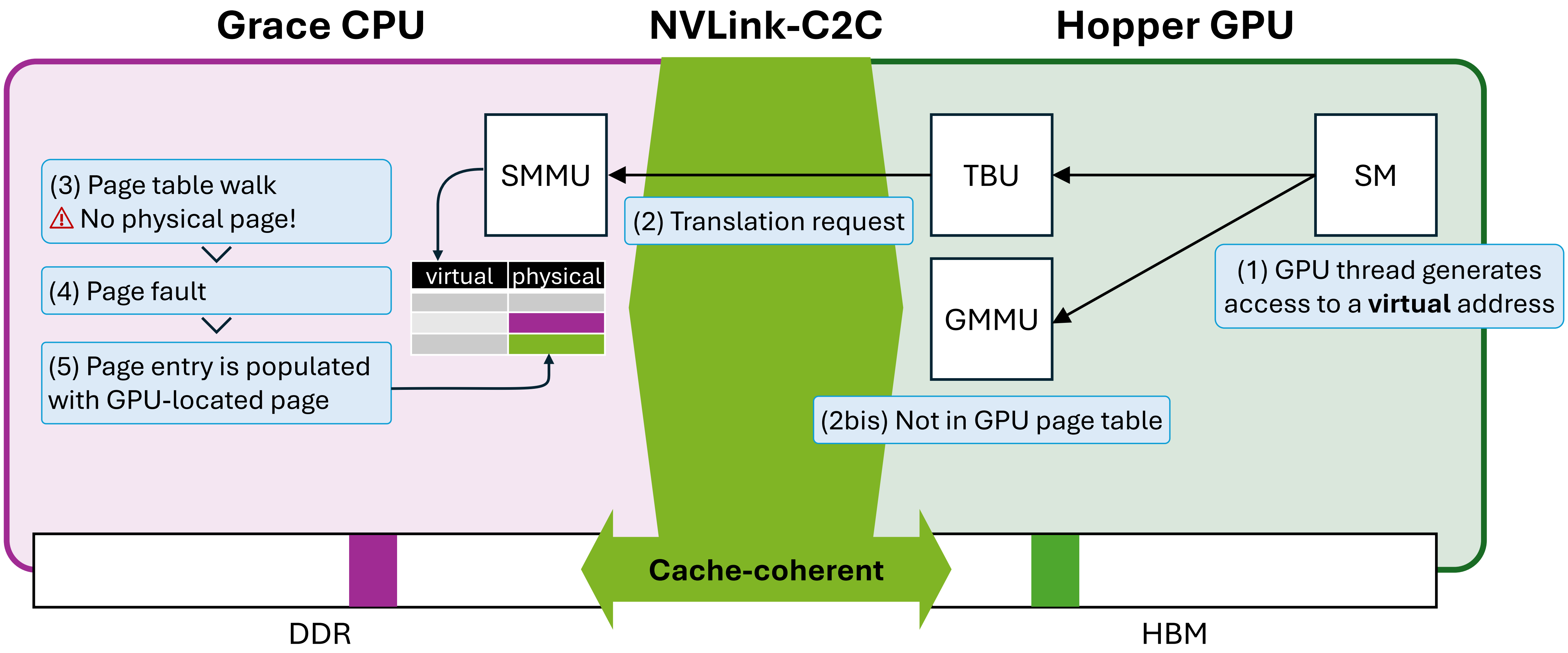
- Pages are placed on first-touch: GPU/CPU memory
- First-touch triggers page fault
- **CPU-side** initialization of page table entries



Virtual	Physical	
`Page 0`	CPU: `0xffffabcd`	← CPU-located page
`Page 1`	GPU: `0xff00abcd`	← GPU-located page
`Page 2`	///	← never-touched ⇒ not physically allocated yet

Figure: System page table (simplified)

First-touch Page Placement – System memory



First-touch Page Placement

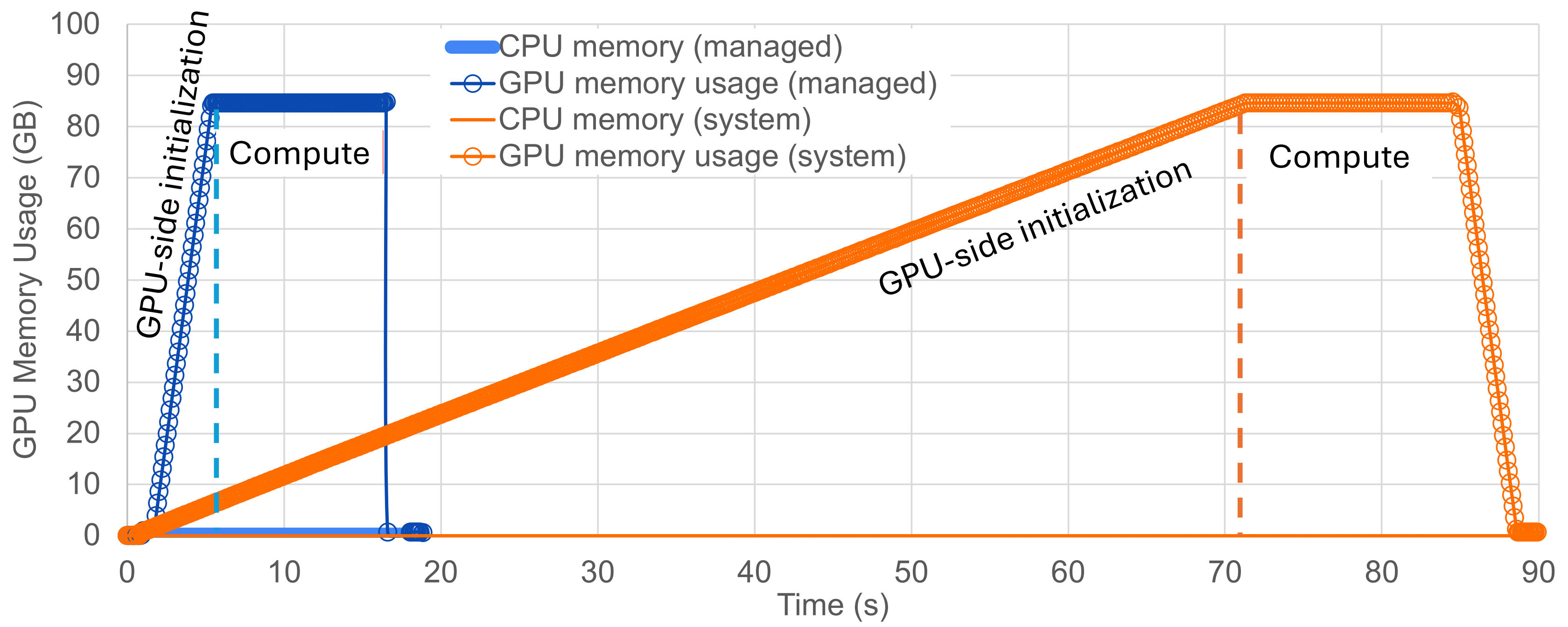


Figure: Qiskit memory usage over time
(system and managed memory, 4 KB pages)

→ In system memory, page table entries are initialized by the CPU

→ This can cause significant slowdown

System Memory – Memory Access

Transparent access from both GPU and CPU to each other’s memory

How?

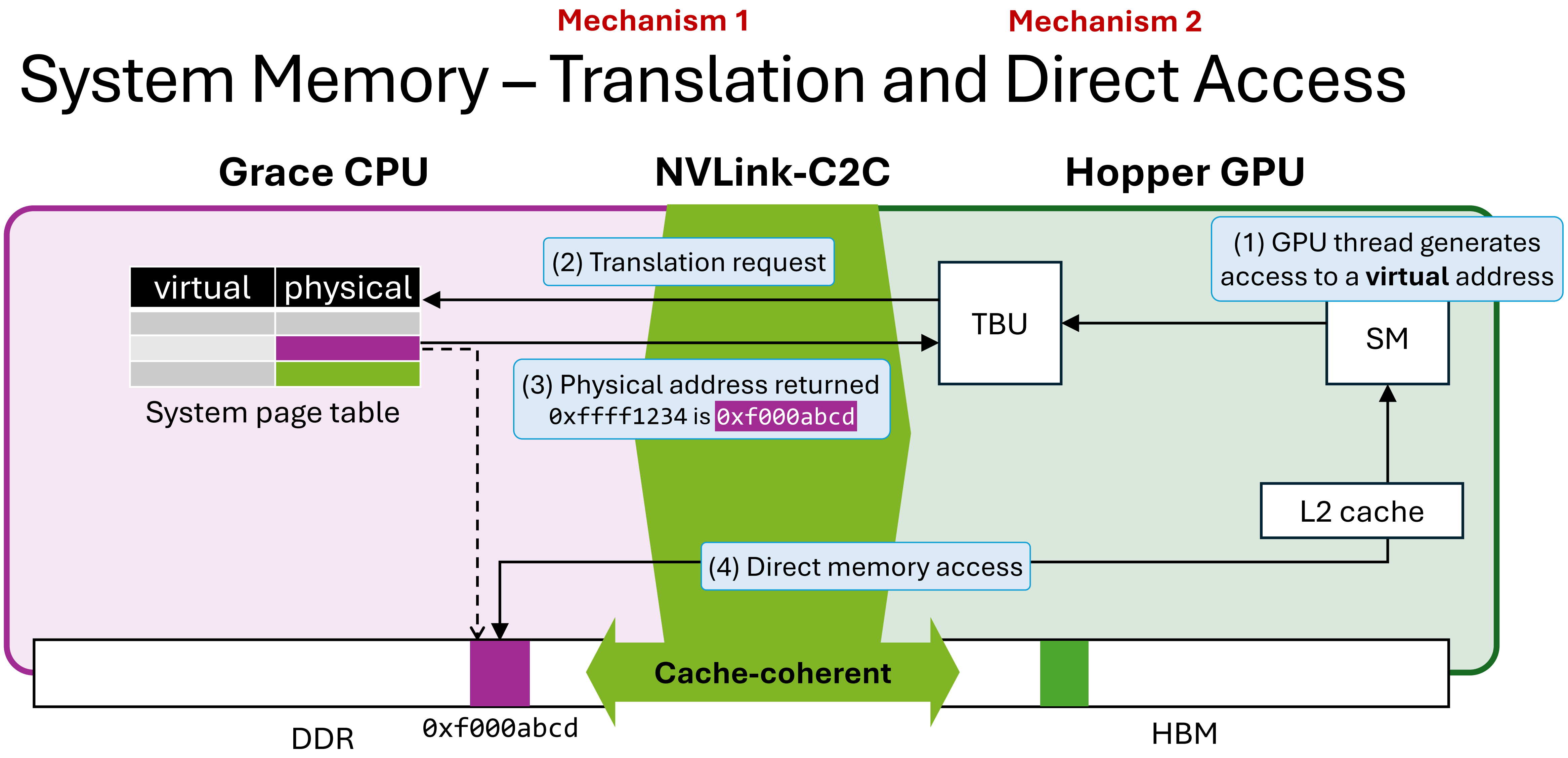
System page table: both CPU- and GPU-located pages

Mechanism 1: adress **translation**

Mechanism 2: **direct remote memory access**

(**Mechanism 3**: Automatic migrations of frequently accessed regions)

System Memory – Translation and Direct Access



→ Fully hardware: CPU cores are not involved

Mechanism 3

Migrations in Unified Memory

New On Grace Hopper*

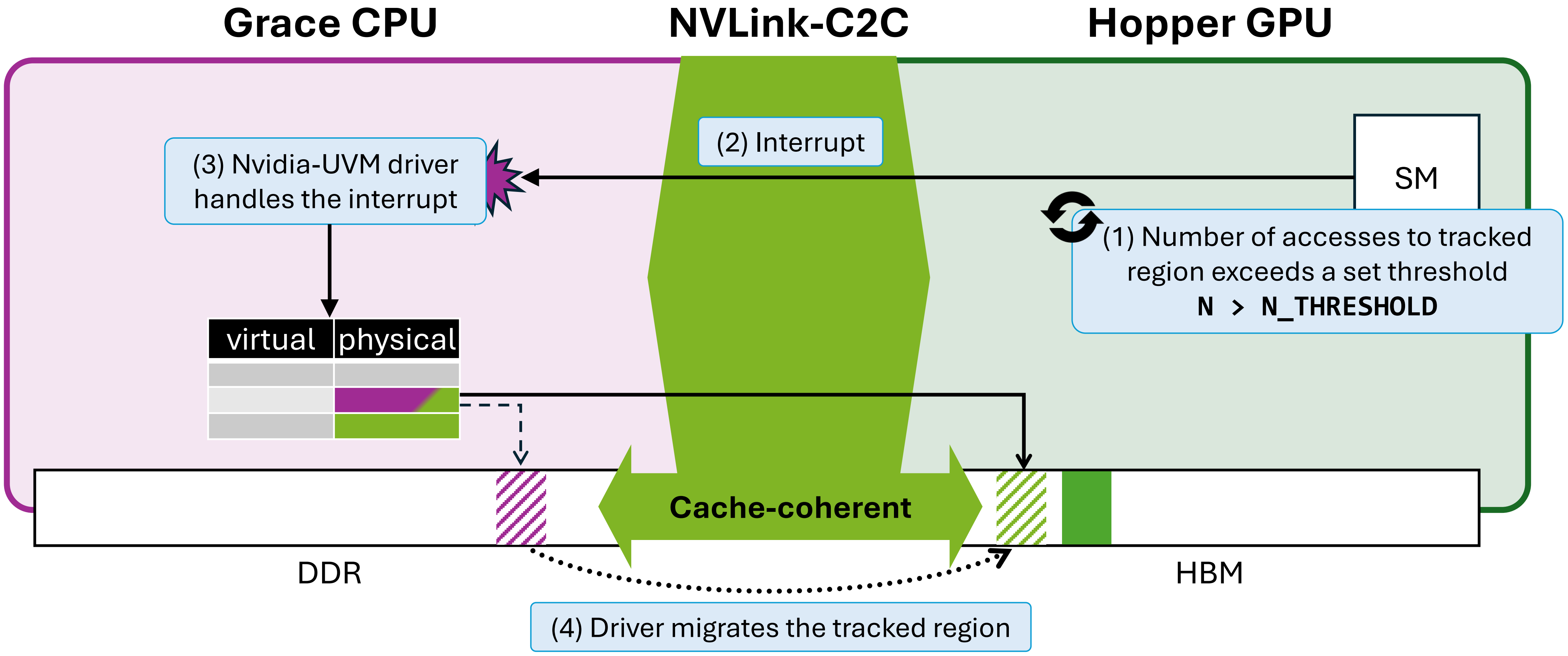
	System		Managed	
Access	Behavior	Cost	Behavior	Cost
1	zero-copy	low	stall + migrate page 💥	very high
2	zero-copy	low	local access	0
…			local access	0
N	trigger migration 💥	very high (if access to migrating page)	local access	0
$N+i$	local access	0	local access	0

+ automatic prefetch

Complex, **hardly predictable**
Cost of access **spread**

Simple, **predictable**
Cost of access paid **once**

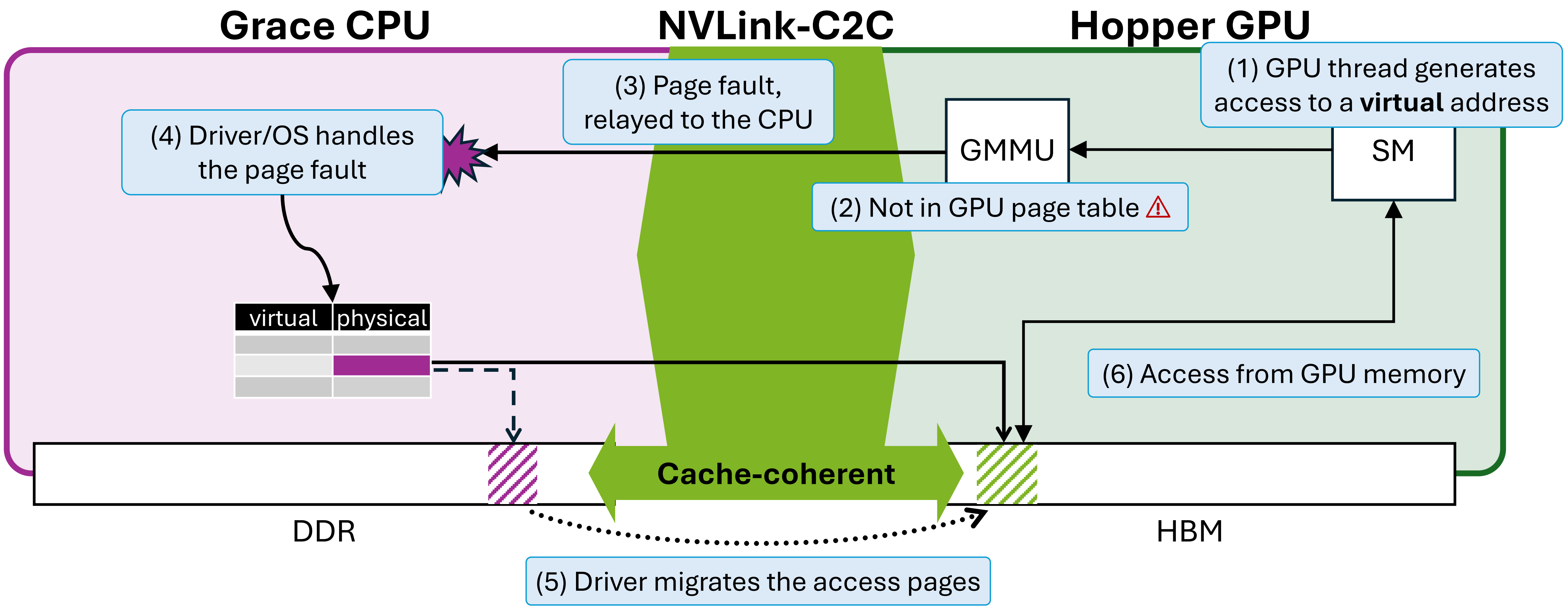
System Memory – Access-counter Based Migration


Grace CPU
NVLink-C2C
Hopper GPU
(2) Interrupt
(3) Nvidia-UVM driver handles the interrupt
SM
(1) Number of accesses to tracked region exceeds a set threshold
N > N_THRESHOLD
virtual
physical
Cache-coherent
DDR
HBM
(4) Driver migrates the tracked region

→ By default, `N_THRESHOLD = 256`, for each 2MB regions (sysadmin configuration)

→ **Goal: improve performance**

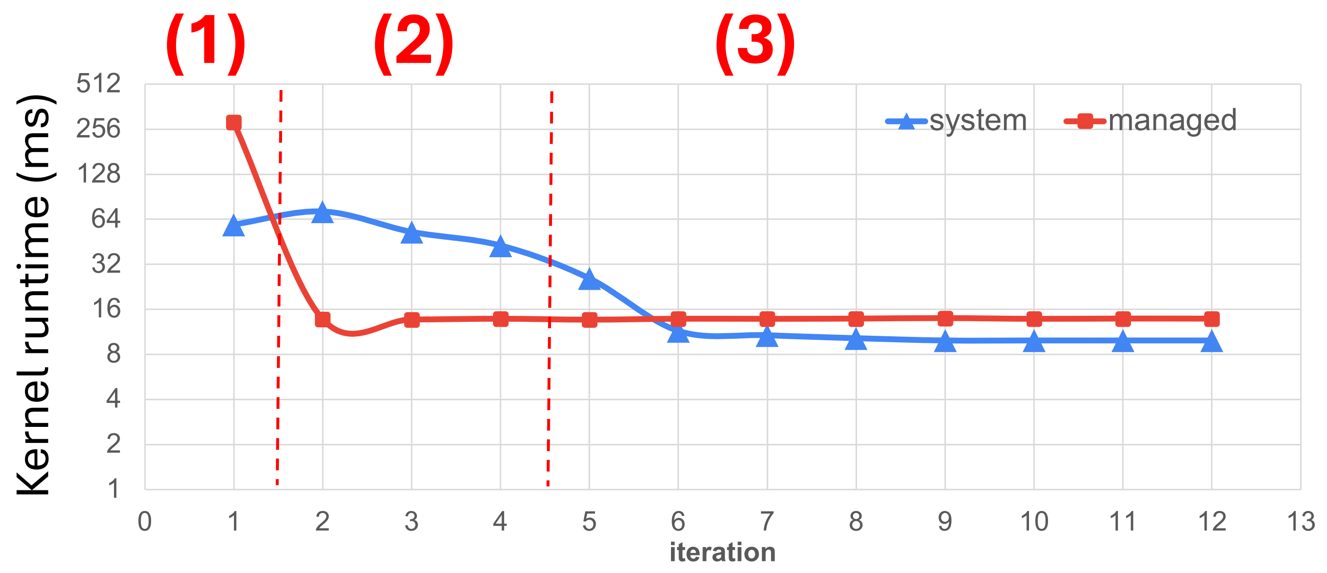
Managed Memory – On-demand Migration


Grace CPU
NVLink-C2C
Hopper GPU
(3) Page fault, relayed to the CPU
(1) GPU thread generates access to a **virtual** address
(4) Driver/OS handles the page fault
GMMU
SM
(2) Not in GPU page table ⚠
virtual
physical
(6) Access from GPU memory
Cache-coherent
DDR
HBM
(5) Driver migrates the access pages

→ By default, performed at access

→ **Goal: ensure locality**

Observing Automatic Migrations



Application: **srad** (iterative process)

```
cpu_init(data)
for(int i = 0; i < n; i++)
    kernel<<<…>>>(data)
```

Three phases:

(1) Initialization:
high runtime, mostly remote reads

(2) Transition:
varying runtime, distributed remote/local reads

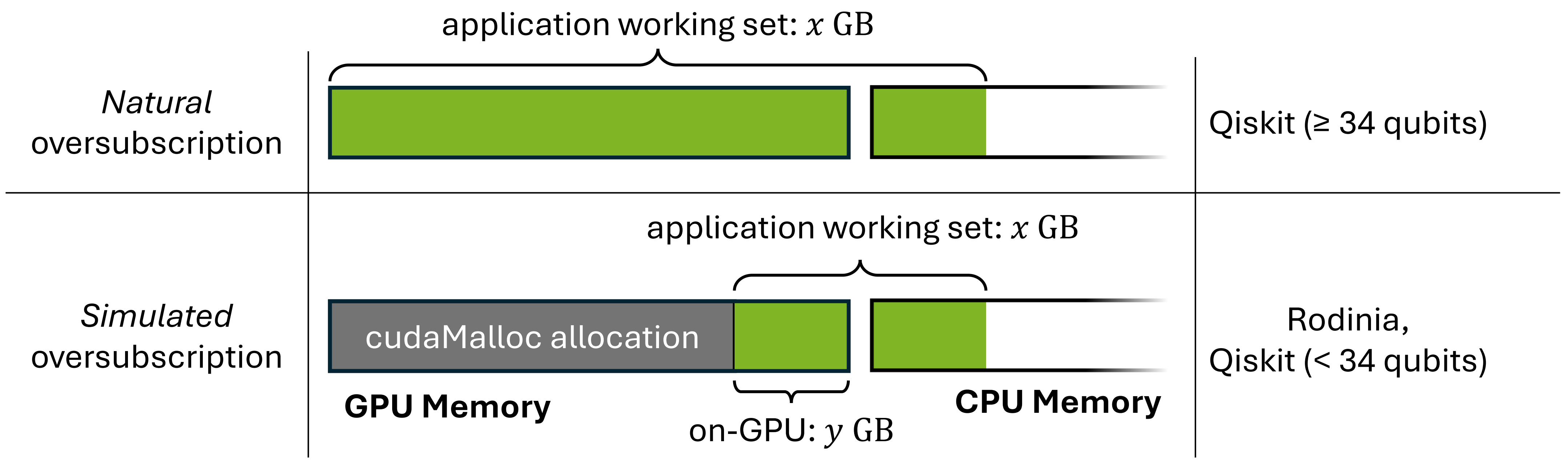
(3) Stable phase:
all data has been migrated, local GPU reads

Access-counter based migrations:

→ Temporary latency increases

→ Runtime affected, in a non-predictable way (migrations are complex)

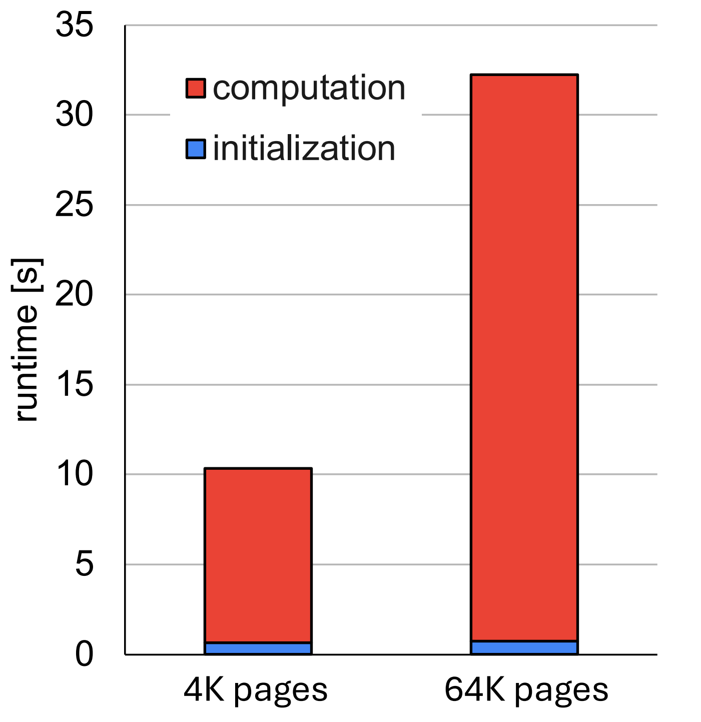
Oversubscription



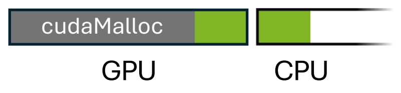
Natural oversubscription
application working set: x GB
Qiskit (≥ 34 qubits)
Simulated oversubscription
application working set: x GB
cudaMalloc allocation
GPU Memory
on-GPU: y GB
CPU Memory
Rodinia, Qiskit (< 34 qubits)

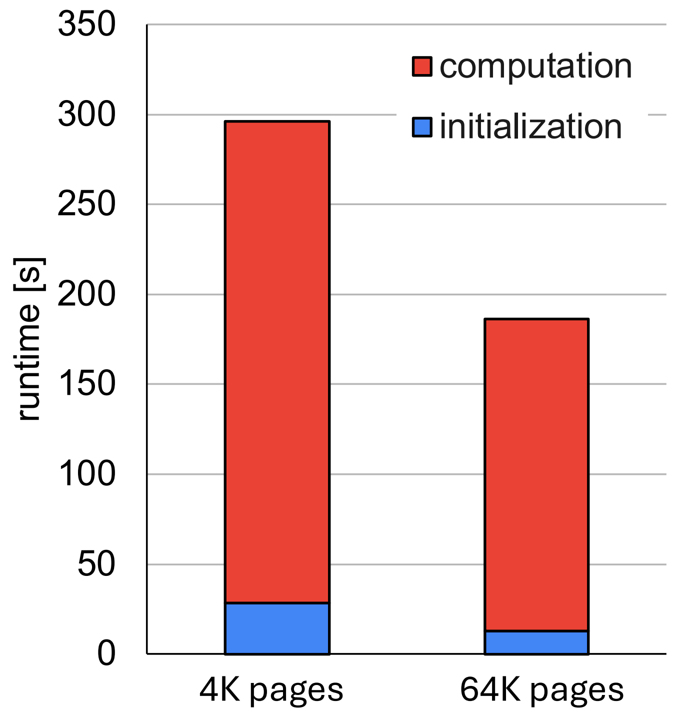
Oversubscription ratio: $\frac{x}{y}$

Oversubscription

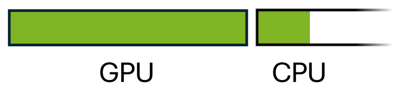


Qiskit, 30 qubits
(200% oversubscription, simulated)





Qiskit, 34 qubits
(125% oversubscription)



Some Key Takeaways

- Page migration: managed *vs*. system

  Managed migrated when accessed **first accessed** → **one-time cost**

  System: migrated when accessed **enough times** → **spread cost**

- cudaHostRegister is still usefull:

  No pinning, but populates page table

  Avoid page faults when accessing system-allocated memory.

- Does it replace cudaMalloc? No

  cudaMalloc: no cache coherency, no adress translation → higher performance

  on-GPU initialization of system memory is very expensive

Future – Scaling to JEDI node (4×GH200)?

More heterogeneous hardware…

- 4× CPU memory, 4× GPU memory
- Two tiers of NVLink interconnect: NVLink-C2C + NVLink-P2P

… but similar software!

- 8× NUMA nodes

Limited oversubscription: CPU memory ≈ GPU memory

Conclusions

- Grace Hopper Superchip: **hardware** support for Unified Memory
- System memory can **improve performance**, induced by
  1. low access granularity (cache line)
  2. automatic access-counter based migrations
- **Simpler** programming interfaces, native profiling tools

- Use case of system memory:

❌ when access pattern is well-known and already finely-tuned

✅ when access pattern is unknown/unpredictable/involves both CPU+GPU

Thanks!

KTH
VETENSKAP
OCH KONST

Funded by
the European Union