NSF LEADERSHIP-CLASS COMPUTING FACILITY

Unified Memory – to use or Not to use.

Dan Stanzione

July, 2025

TACC
TEXAS ADVANCED COMPUTING CENTER
TEXAS
The University of Texas at Austin

Unified Memory

- Different Approaches
- Why use it
- How can you program it
- Which method should you use?

Unified Memory

- If you have been at this workshop all morning, you know what it is:
  - Creating a single address space between CPU and GPU (or any set of processing elements).
  - Ideally allowing both/all devices to access the same physical memory.
  - Not necessarily uniform access speeds.
- There are a lot of historic approaches to achieving this; for a short talk, the two main ones on the market now (and the drivers for this workshop) are:
  - NVIDIA Grace-Hopper (Vista, IsembardAI, Jupiter, Alps, Miyabi-G, and soon Horizon).
  - AMD MI300A (El Capitan, SDSC Cosmos).

Why would you want to do this?

- The obvious reason: Reduce Complexity
  - But a lot of code exists that assumes you don't have it.
- The hopeful reason: Increase Performance
  - In almost every HPC context, making copies/moving data around is pure overhead.
- Less obvious: Reduce total energy used.
  - Turns out, a lot of power goes into moving data around.

- Let's look at how these could work, and what we know about if they do.

Hardware Approaches

- GH200
  - The CPU has its own LP-DDR memory.
  - The GPU has its own HBM memory.
  - The Address Translation System makes access work from either device; but each has different performance and capacity.
- AMD MI300A
  - The CPU tiles and GPU tiles all have their own memories; but it is true shared memory NUMA just as in a multi-chiplet AMD CPU or across the tiles of a pure GPU. (in essence, one chiplet has been subbed out for CPU cores instead of GPU cores).
  - All tiles have the same memory speed/capacity, but there is, as always, advantages to locality – access to remote tiles has a cost.

Grace-Hopper Architecture

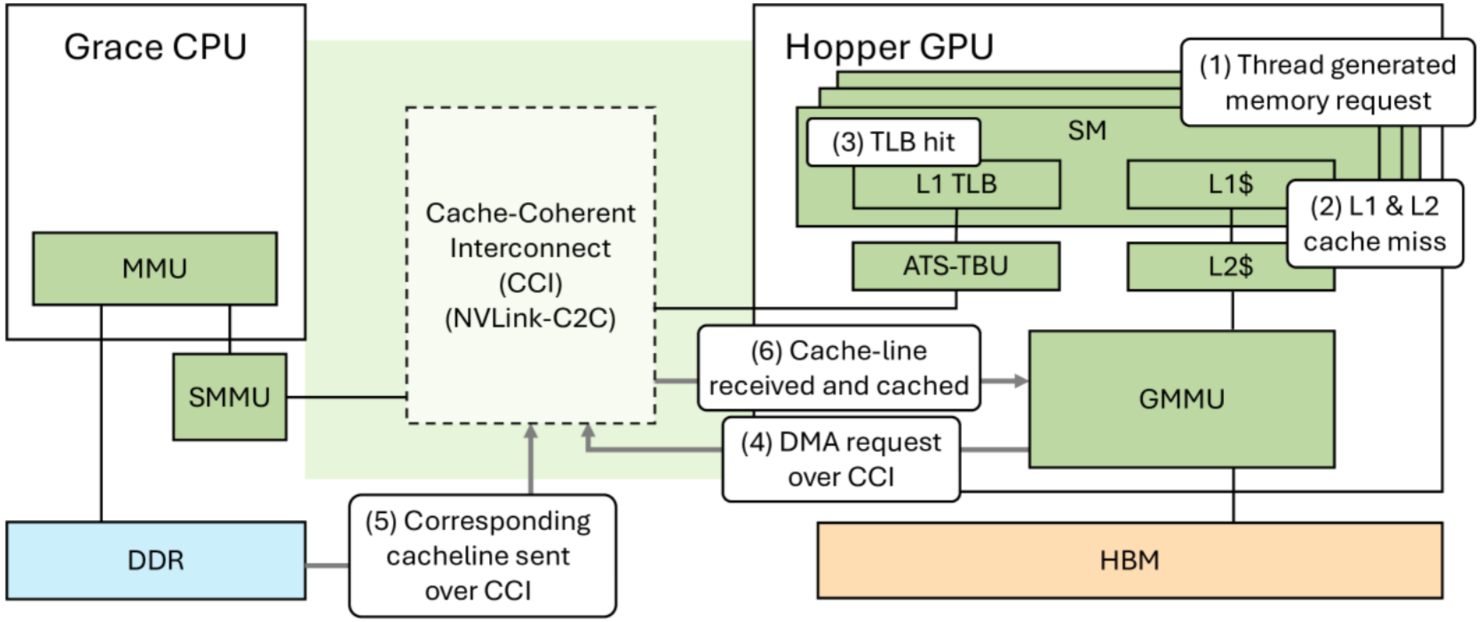


Figure 1. An overview architecture of the Grace Hopper platform that interconnects CPU and GPU with high-throughput cache-coherent NVLink-C2C.

Source: https://arxiv.org/html/2407.07850v1 "Harnessing Integrated CPU-GPU System Memory for HPC: a first look into Grace Hopper"

A Note on Energy

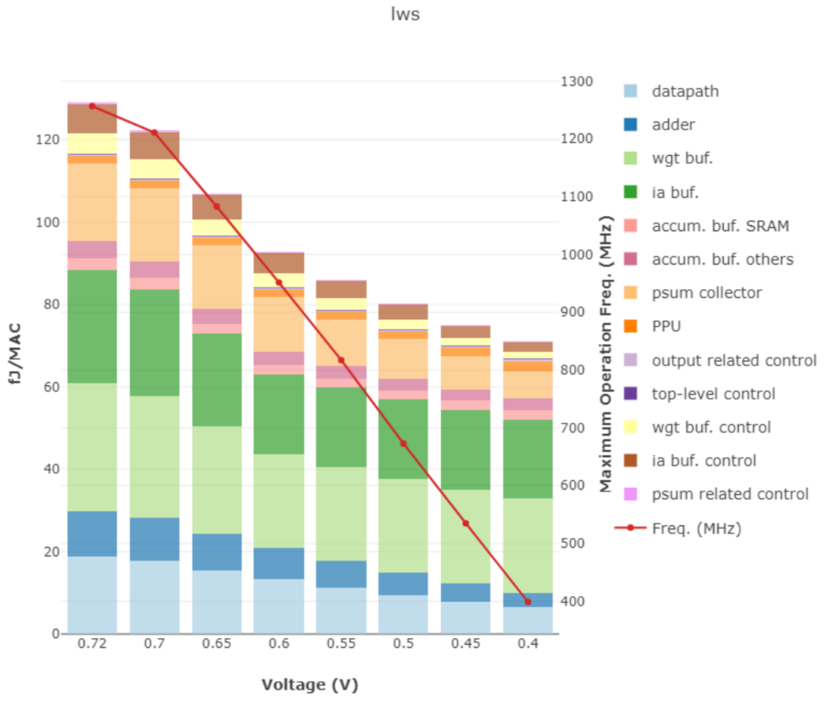
- Data Movement uses a lot of energy.
- Not just across networks, but across the chips themselves!
- In this NVIDIA examples, the actual math op is 1/3rd the power of the operation… the rest is datapath and buffers!
- Tighter integration of memory can save a lot of power.

ENERGY DOMINATED BY MEMORY AND DATA

70 fJ/MAC

35 fJ/OP

29 TOPS/W



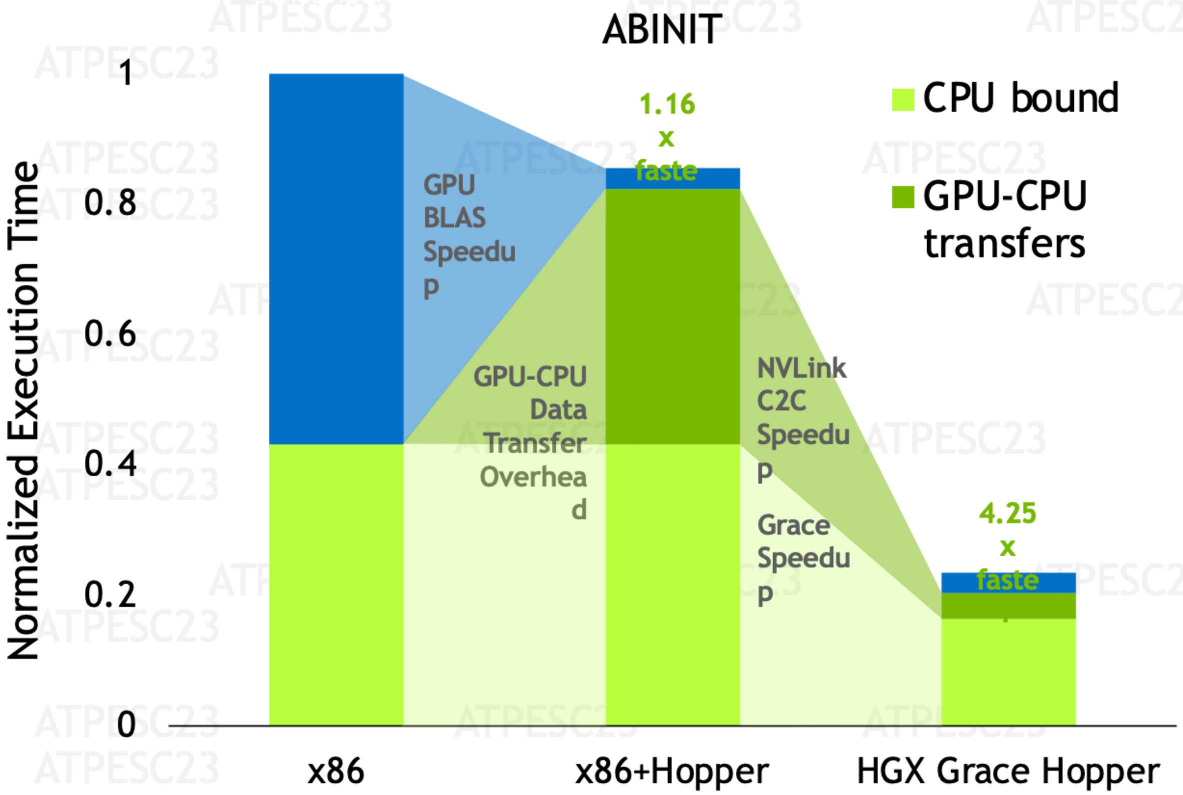
Source: Giri Chukkupali, “NVIDIA Grace Hopper Architecture” , ATPESC-2023

Let’s consider performance a bit.

- We know (and Amit has shown) we get really good performance from Grace-Hopper, vs. both CPUs and older CPU-GPU systems.
  - But, do we know where it came from?
    - Faster GPUs than A100 or MI250x?
    - Better CPUs than the old platform?
    - Removal of the PCI bus between GPU-CPU?
    - Unified/faster memory system?
  - And how would we measure the impacts of the memory system?
- To do so, we need to know a little about how to program them; let’s dive into the GH200.

GRACE+HOPPER MAKES ACCELERATION MORE ACCESSIBLE

Delivers Superior Performance and Efficiency for HPC



White Paper - Grace Hopper Superchip Architecture

A performance simulation for ABINIT with NVBLAS featuring Titanium 255 Atoms using the LOBPCG algorithm

Source: Giri Chukkupali, “NVIDIA Grace Hopper Architecture” , ATPESC-2023

Address Translation Service

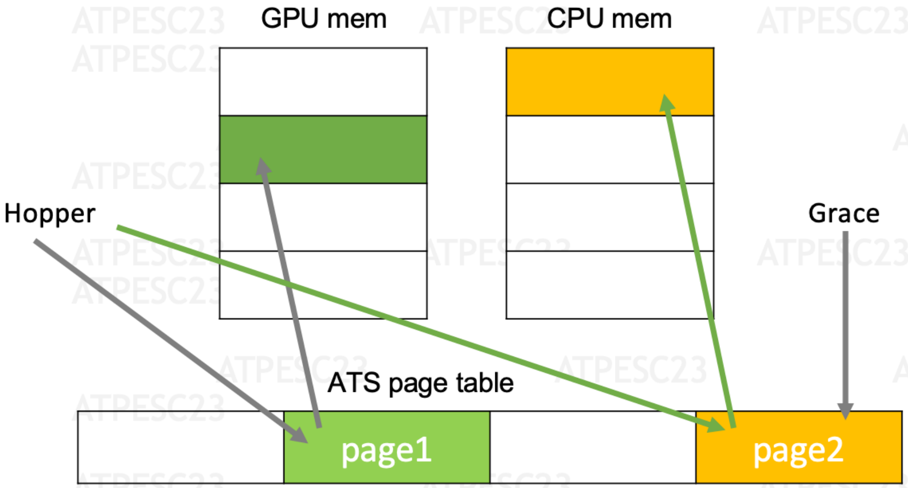
With ATS, you have multiple options for how to deal with memory:

- Explicit copies (manage as you always have).
- Managed Memory
- System Allocated

Address Translation Service

GRACE HOPPER

Unified Programming Model



- Address Translation Service (ATS) enables all CPUs and GPUs in the node to share a single page table
- System-allocated memory is accessible by all CPU and GPU threads
- Runtime system backs system-allocated memory with physical memory on first touch, either on LPDDR5 X or HBM3, depending on whether a CPU or a GPU thread accesses it first .

Source: Giri Chukkupali, “NVIDIA Grace Hopper Architecture” , ATPESC-2023

And you can do this across languages!

- Support for C, C++, Fortran, and Python.
  - Through OpenACC directives, or through native languages.
  - Python support still listed as “experimental”, but I used it and got acceleration.
- With nv compilers, add `-gpu=unified` and `-stdpar or -acc`
  - Fortran, use the do concurrent loop construct, e.g.

```
do concurrent (i = 1 : size(b))
    a(b(i)) = i
enddo
```

  - C++, use the stdpar package e.g.

```
std::for_each(std::execution::par_unseq, r.begin(), r.end(), [&](auto i) { my_array[i] = init_val; });
```

  - Python, use standard Numpy and Cupy, with managed allocation extensions.

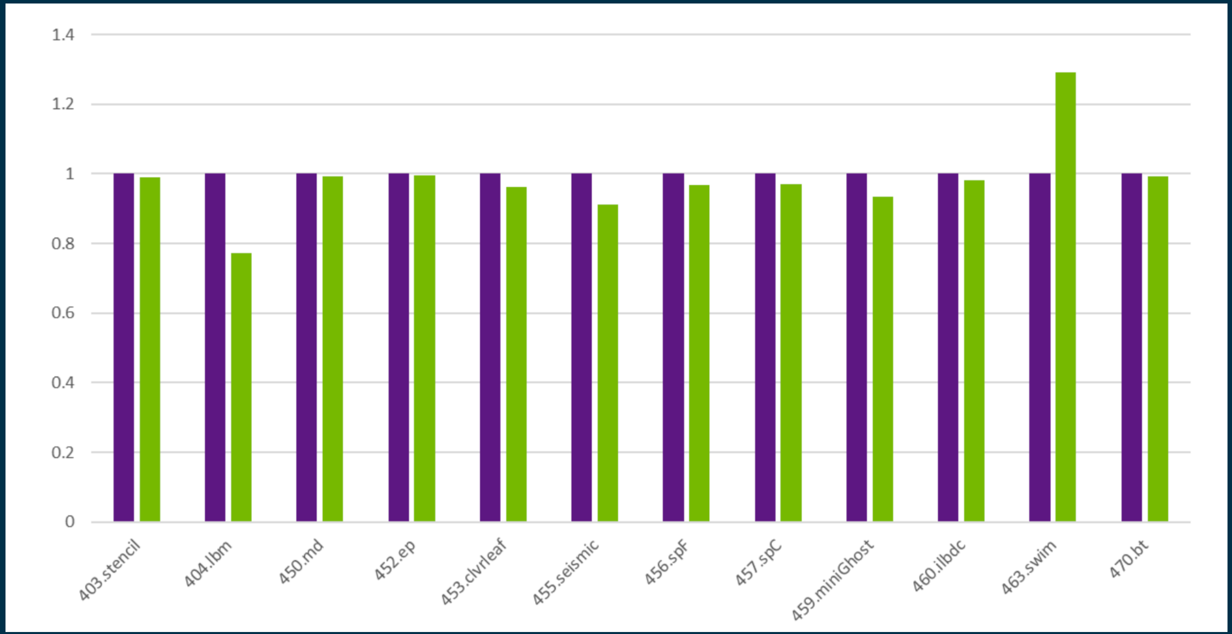
Memory Models in GH200

Memory Location	Allocation Interface	PTE Init	Cache Coherent	Migration Granularity
CPU/GPU	malloc()	CPU	Yes	transparent 128 byte 64KB
CPU/GPU	cudaMallocManaged()	CPU	Yes	transparent 2MB
GPU	cudaMalloc() cuMemCreate()	GPU	No	explicit 1 byte
CPU	numa_alloc_onnode() cudaMallocHost() cudaHostAlloc() cuMemCreate()	CPU	No	explicit 1 byte

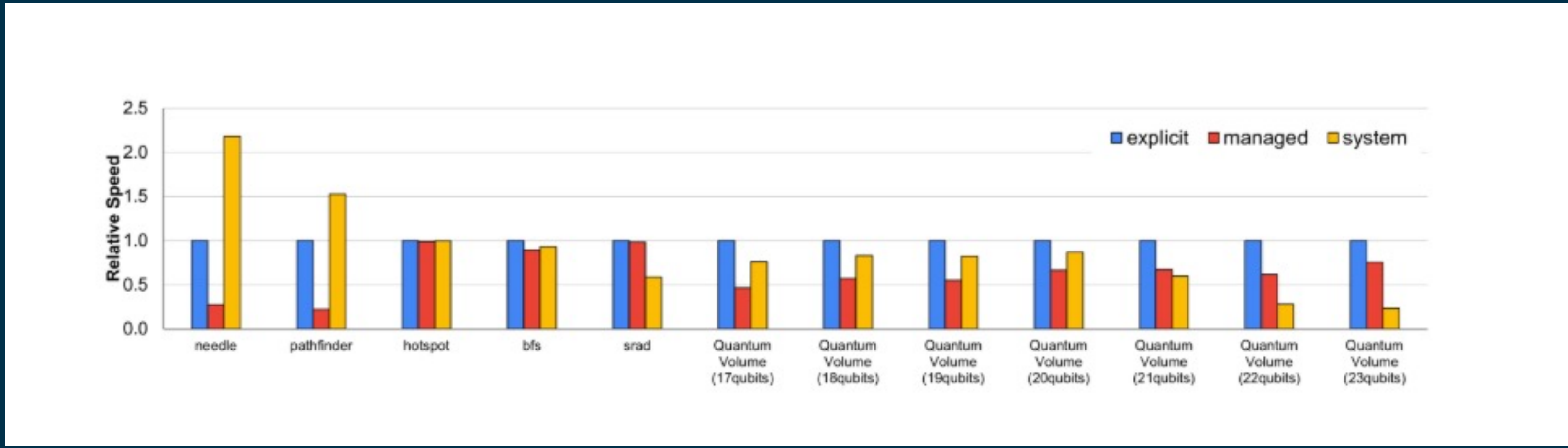
Source: https://arxiv.org/html/2407.07850v1 "Harnessing Integrated CPU-GPU System Memory for HPC: a first look into Grace Hopper"

SpecFP Performance

OpenACC implementation with/without –gpu=unified



Source: https://developer.nvidia.com/blog/simplifying-gpu-programming-for-hpc-with-the-nvidia-grace-hopper-superchip/



Various apps, explicit memcopy vs. managed buffers vs. system.

Source: https://arxiv.org/html/2407.07850v1 "Harnessing Integrated CPU-GPU System Memory for HPC: a first look into Grace Hopper"

Lulesh Performance

For Lulesh, the improvement comes from the H200 being faster and more power than the PCI version.

Not so much from the memory system.

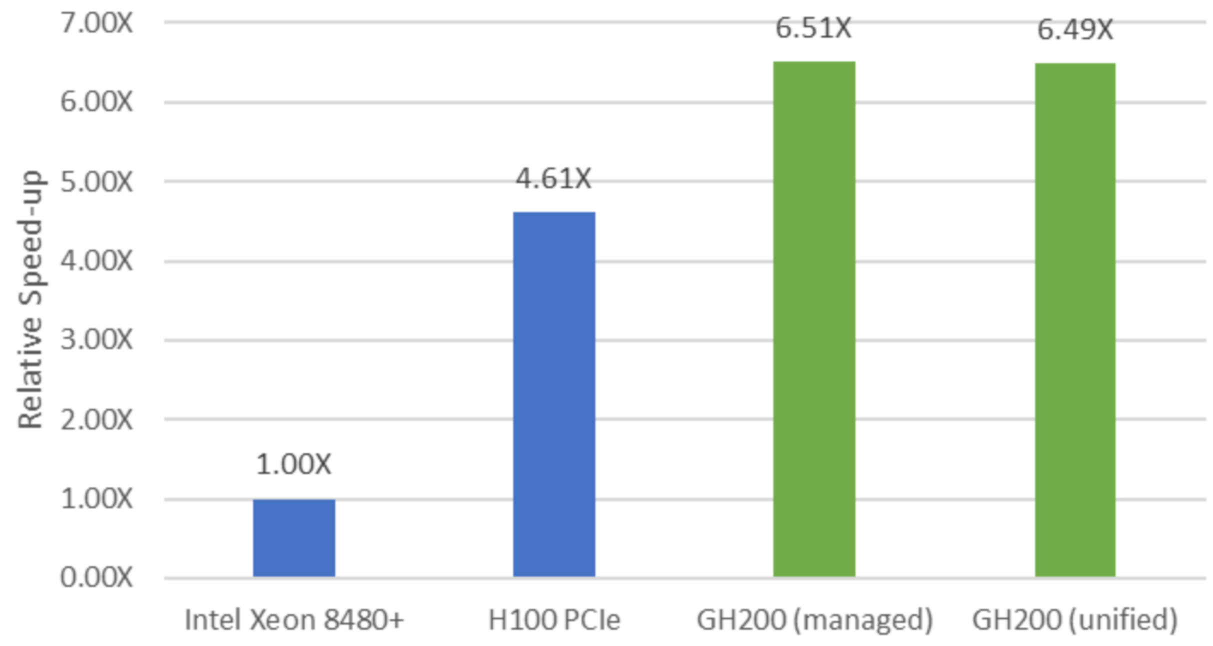


Figure 2. Comparison of LULESH performance using managed and unified memory options on NVIDIA GH200 with NVIDIA H100 PCIe and a modern CPU

Source: https://developer.nvidia.com/blog/simplifying-gpu-programming-for-hpc-with-the-nvidia-grace-hopper-superchip/

POT3D Performance

By contrast, with POT3D, unified memory makes OpenACC calls completely redundant.

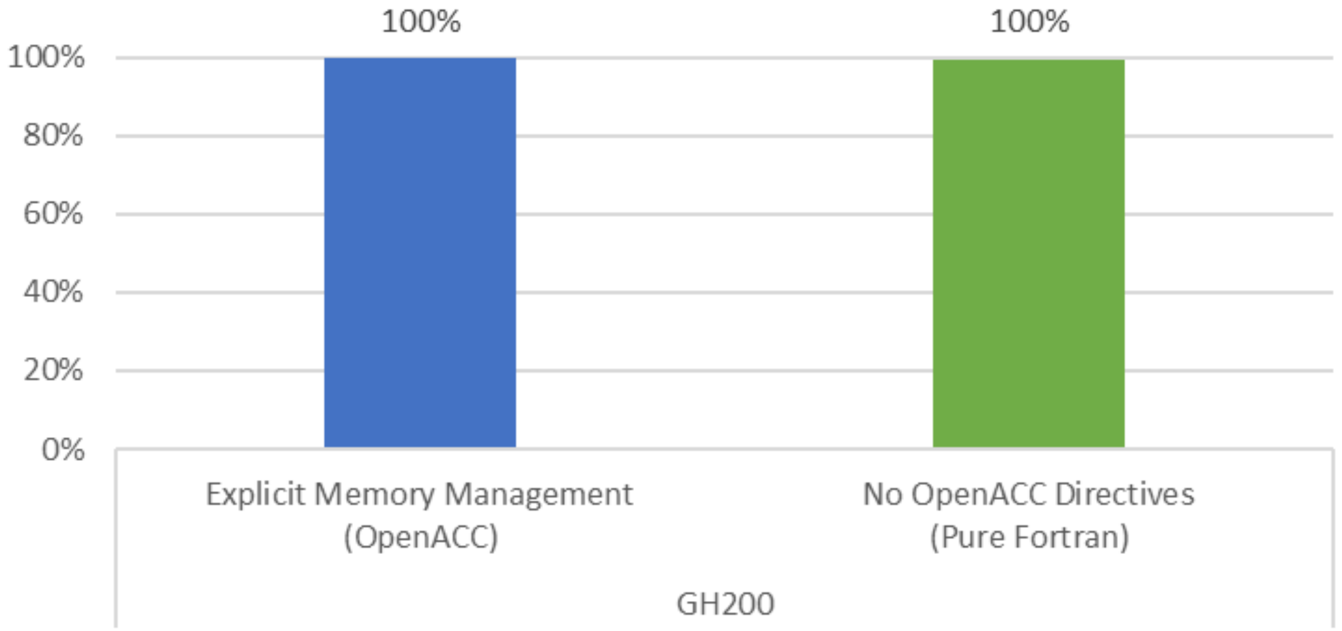


Figure 3. POT3D performance using OpenACC data directives compared to Grace Hopper unified memory

Source: https://developer.nvidia.com/blog/simplifying-gpu-programming-for-hpc-with-the-nvidia-grace-hopper-superchip/

Performance for the same Matrix Multiply, across 3 ways of allocating memory in Python.

- Let's compare GH200 to GH200, so we can only look at performance of the memory system!
- "Traditional" – allocate with numpy on the CPU, multiply on the GPU.
  - Old code on the new platform, in essence.
- "Managed" – allocate on the CPU, use the shared memory system.
- "GPU Local" – allocate on the GPU, use on the GPU (the old way if it fit).
- Multiply a 64kx16k matrix with a 16kx8k matrix to generate a 64kx8k matrix.
  - Data to be moved: ~10GB (and about a second of FLOPS).

The setup:

Vista Modules: gcc/15.1.0 cuda/12.9 python3

Python libraries:

```
import nvmath as nv
import cupy as cp
import numpy as np
import cupy._core.numpy_allocator as ac
import numpy_allocator
import ctypes
import time
```

The allocator package is experimental last I checked

My simple kernel for these measurements:

```
m, n, k = 65536, 32768, 8192
A = np.random.rand(m, n)
B = np.random.rand(n, k)
C = np.random.rand(m, k)
start = time.time()
D = cp.matmul(cp.asarray(A), cp.asarray(B) )
elapsed = time.time() - start
print(f"MatMul:  Time: {elapsed:.2f}s")
```

To use managed memory:

This is the experimental part:

```
cp.cuda.set_allocator(cp.cuda.MemoryPool(cp.cuda.malloc_managed).malloc)
```

```
lib = ctypes.CDLL(ac.__file__)
class my_allocator(metaclass=numpy_allocator.type):
    _calloc_ = ctypes.addressof(lib._calloc)
    _malloc_ = ctypes.addressof(lib._malloc)
    _realloc_ = ctypes.addressof(lib._realloc)
    _free_ = ctypes.addressof(lib._free)
my_allocator.__enter__()
```

All GPU Version:

```
A = cp.random.rand(m, n)
B = cp.random.rand(n, k)
C = cp.random.rand(m, k)
start = time.time()
D = cp.matmul(cp.asarray(A), cp.asarray(B) )
cp.cuda.Stream.null.synchronize()
elapsed = time.time() - start
print(f"MatMul:  Time: {elapsed:.2f}s")
```

With Managed Memory on, NumPy and CuPy references is just the domain of where to run the command.

Results

5 runs on Vista dev node, GH200:

- Traditional copy: 1.99 seconds
- ATS/Managed: 1.30 seconds
- GPU Native: 1.01 seconds (should be the upper bound).

Unified memory is a huge improvement on the traditional copy, execution time reduced by 1/3rd. GPU native is the best you can get, but assumes your data fits in GPU memory, and didn't ever have to get there.

  - Another way to look at this: With traditional copy, the runtime is 50% overhead; with unified memory, it's only 25% overhead.
  - As usual, the code matters, not just the chip!

Takeaways

- Well implemented managed memory systems *can* help performance, power, and open up different programming models.
- You can run code without changes… but it may not be optimal.
- You can run code without managing data movement… but that may not be optimal either!
- As always, good implementation helps; hopefully this type of system will be closer to universal in the future!
  - Certainly, you will see a lot of it on upcoming NVIDIA machines – future AMD systems TBD.

NSF LEADERSHIP-CLASS COMPUTING FACILITY

Thanks!

dan@tacc.utexas.edu

TACC
TEXAS ADVANCED COMPUTING CENTER

TEXAS
The University of Texas at Austin