TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Accelerating Scientific Applications with Automatic BLAS Offload on NVIDIA Grace-Hopper

Summary of Work
12/19/2024

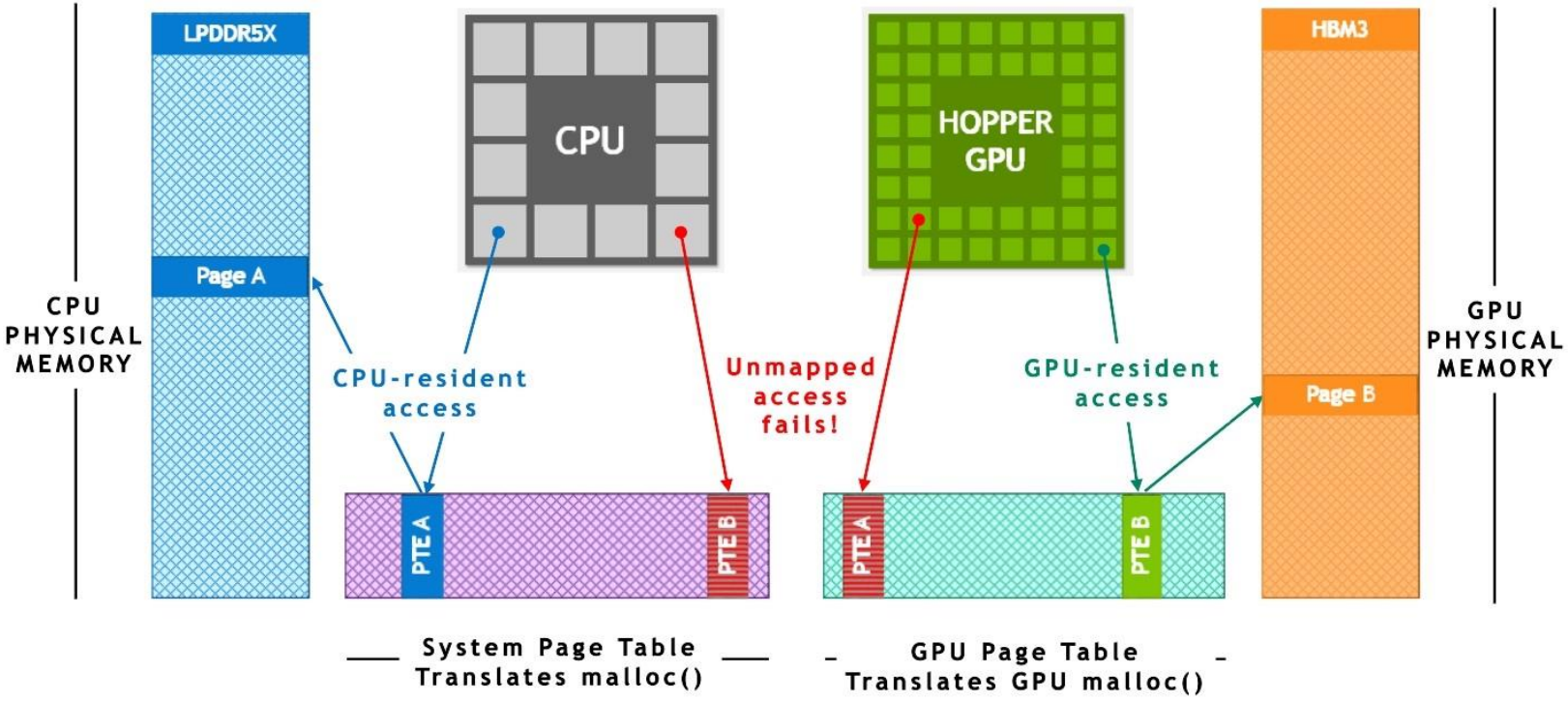
PRESENTED BY:
Junjie Li (jli@tacc.utexas.edu)
Texas Advanced Computing Center,
The University of Texas at Austin

Outline

- Quick review of GPU architecture (conventional & Grace-Hopper)
- Design philosophy of automatic BLAS offload library
- Application tests includes:
  1. PARSEC: real-space DFT
  2. LSMS/MuST: linear-scaling DFT

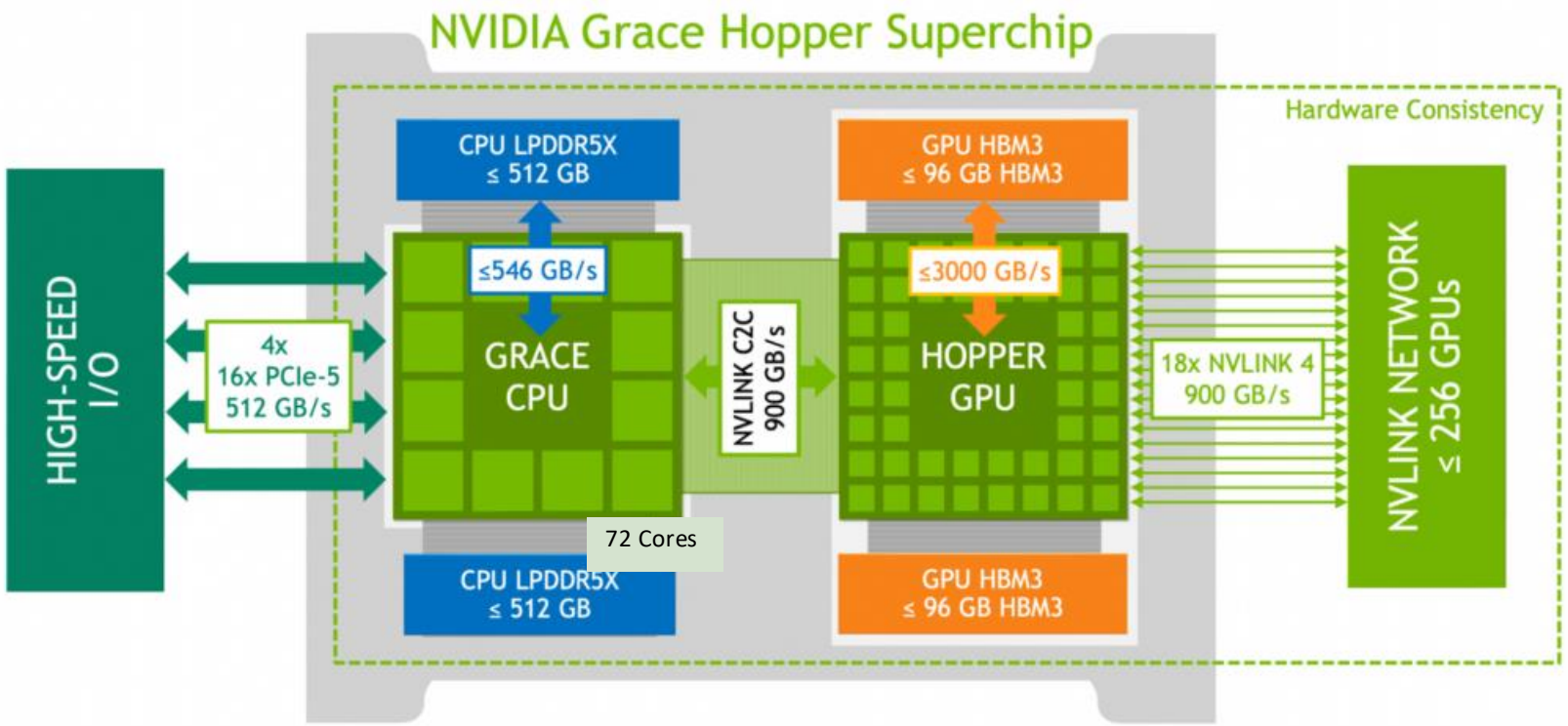
All tests in GH200 are performed with:
- CUDA driver 560.35.03
- CUDA version 12.6
- NVHPC 24.9 suite (compiler, NVPL, cuBLAS)

Conventional GPU memory architecture



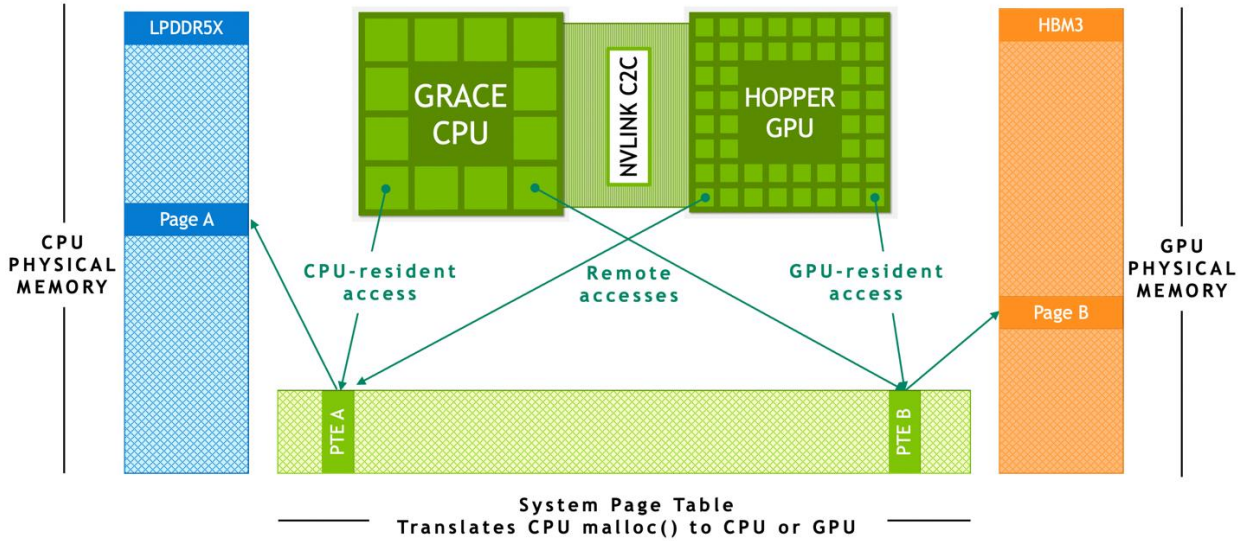
- Conventional disjoint page tables
- CPU and GPU manages their own memory
- No direct access to each other's data
- Data on main mem needs to be copied to GPU memory for use.
- GPU mem allocated by CudaMalloc

Grace-Hopper (GH200): Key features



- New architecture release in 2023
- Available on Vista @ TACC https://tacc.utexas.edu/systems/vista
- Superchip design
- Chip-to-Chip (C2C) NVLink at 450 GB/s per direction
- PCIe Gen 5 x 16 only has 64 GB/s
- Conventional memory management
- **Unified memory management with cache coherency**

Grace-Hopper (GH200): unified memory



- (supports conventional GPU memory management)
- Single system page table for LPDDR5X + HBM3
- **Coherent memory** at cache line level through C2C NVLINK
- Cache coherency is critical for Unified memory access
- Different from CUDA managed memory (used to called unified memory) which relies on page-fault mechanism and page movement.

cache coherency: data consistency of shared data that is stored in multiple local caches accessible by multiple processors

Allocator	Initial Page Table Entry	Cache Coherent
malloc	CPU	yes
cudaMallocManaged	CPU	yes
cudaMalloc	GPU	no
cudaMallocHost	CPU	no

Grace-Hopper (GH200): Stream & HPL

STREAM TRIAD Bandwidth (GB/s)

	CPU	GPU
LPDDR5X	418	610
HBM	142	3680

Astonishing bandwidth.
Data locality still matters.

- HPL benchmark, used by Top500 ranking
- Highly FP64 capable GPU
- cuBLAS can use tensor core
- Big incentives to use GPU for FP64 calculations

HPL (FP64)

	Rmax (Tflops)
Grace (72C)	2.8
Hopper	52.9

BLAS (Basic Linear Algebra Subprograms)

- One of the fundamental building blocks of many scientific applications
- Basis of other linear algebra libraries: LAPACK, ScaLAPACK
- Linear algebra → natural language of quantum mechanics
- Many quantum physics/chemistry codes heavily relies on FP64 BLAS (well-fit for GPU)

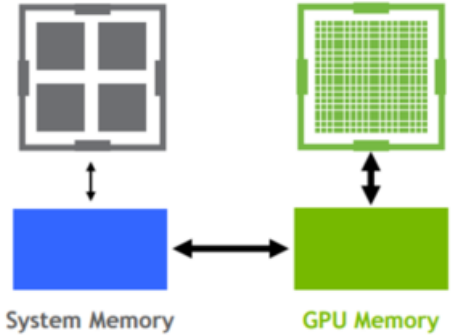
How I started: Porting PARSEC to GPU

- real-space DFT code at UT Austin by Prof. Chelikowsky
- PARSEC heavily relies on dgemm (77% runtime)
- Most dgemm comes from ScaLAPACK (pdgemm, pdtrsm)
- Never run on GPU before
- But now HPC is more GPU centric
- BLAS -> cuBLAS?
  - no way to rewrite ScaLAPACK
  - cuBLAS: different interface
  - no intent to rewrite user code
  - Something automatic for users?
  - symbols interception with BLAS wrapper for automatic offload ?

```
recorded MPI rank: 0
total runtime: 681.9s
---------- function statistics (exclusive) ----------
  exclusive call time (in seconds) and counts
-----------------------------------------------------
  | group  |  function  |  count  |  time  |
-----------------------------------------------------
 1 |   BLAS |    dgemm_ |   43481 |  524.512 |
 2 |  PBLAS |    pdtrsm_ |    750 |   25.975 |
 3 |  PBLAS |    pdgemm_ |    1505 |   13.446 |
 4 |  PBLAS |    pdsymv_ |   23840 |    4.680 |
 5 |   BLAS |    dcopy_ |  28194164 |    3.236 |
 6 |   BLAS |    dtrsm_ |    427 |    1.686 |
 7 |   BLAS |    dgemv_ |  1157095 |    1.389 |
 ...
 ...
```

Auto BLAS Offload Attempts

- Cray LIBSci (since Titan@ORNL), IBM ESSL, NVidia NVBLAS (heavy overheads)
- Some require relink, or only work for dynamically linked BLAS
- **All performs mandatory data copies to/from GPU (even with unified memory)**
  - copy data costs more time than compute in practice
- hardly useful in practice.
- Coherent unified memory in Grace-Hopper opens up new opportunities



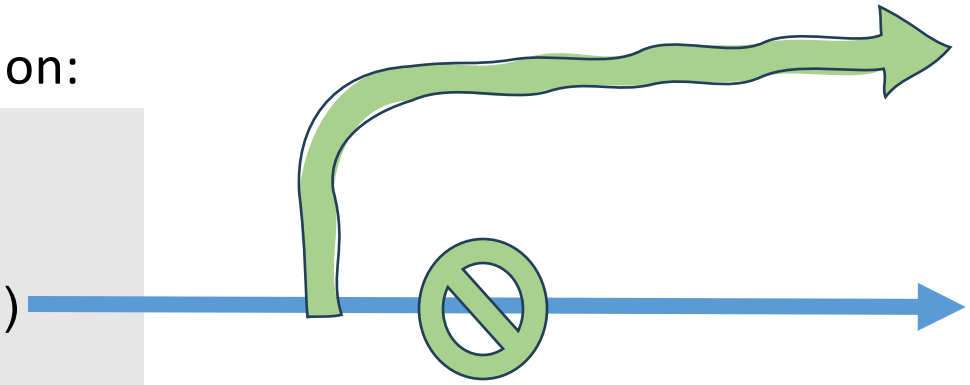
- **SCILIB-Accel: a new tool to overcome all these issues & do performant auto offload!**

SCILIB-Accel: workflow

SCILIB-Accel:
hacks user binary and redirects BLAS call to a custom-built BLAS wrapper

User application:

```
...
...
call dgemm(…)
…
```



```
mydgemm( …) {
   manage data for gpu
   call cublasDgemm(…)
   timer()
}
```

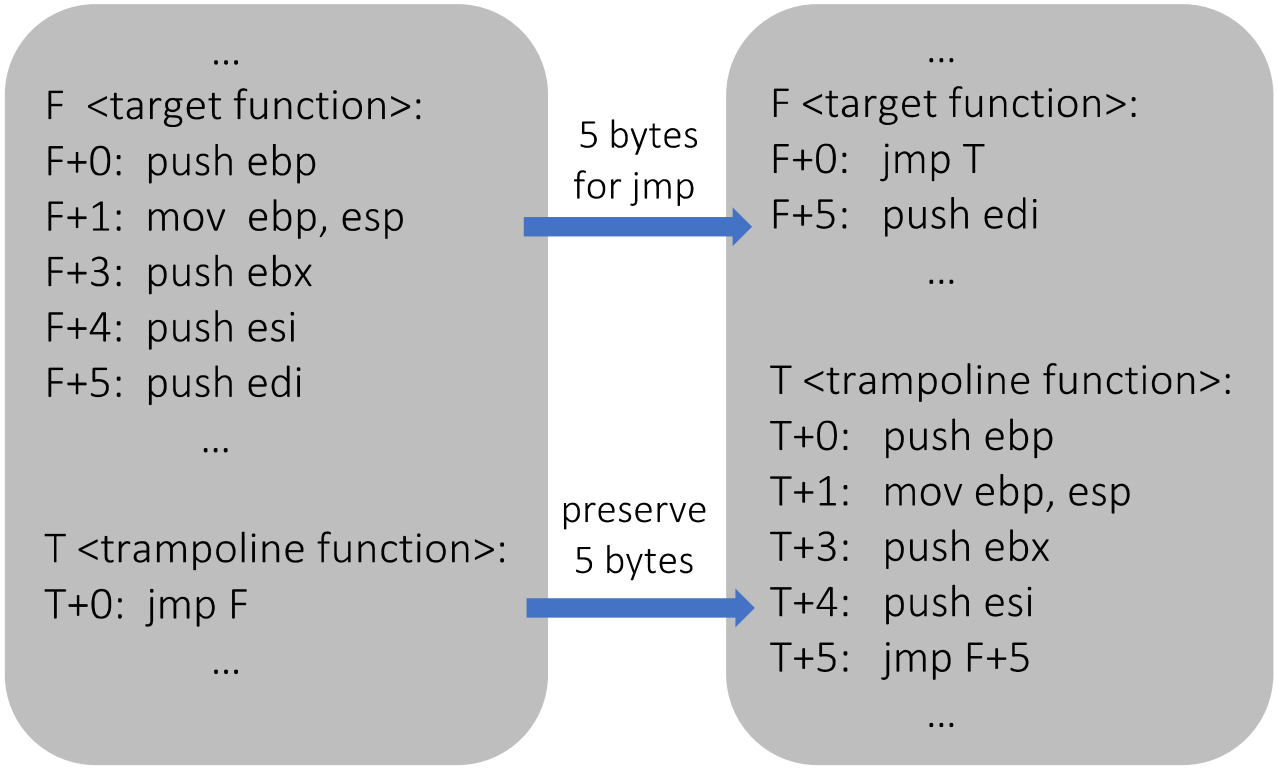
CPU BLAS Library

```
dgemm( …) {
  …
}
```

SCILIB-Accel: intercept BLAS calls

Replace BLAS calls with CUBLAS calls (all level-3 BLAS implemented)

- Symbol interception by **trampoline-based Dynamic Binary Instrumentation (DBI)**
- DBI widely used in profiling
- rewriting in-process binary image
- No need to recompile
- Effective for both dynamically and statically linked BLAS.
- Negligible overhead when replacing function with same signature.



- Yinzhi Wang and Junjie Li, "PEAK: a Light-Weight Profiler for HPC Systems", https://dl.acm.org/doi/abs/10.1145/3624062.3624143
- https://github.com/microsoft/detours/wiki/OverviewInterception

SCILIB-Accel: data movements strategy 1/3

1. **Mem-copy**: cudaMemCpy Matrices between CPU mem to GPU mem for every CUBLAS call (all other tools do this)

Pseudocode:

```
cudaMalloc  device_A, device_B, device_C

cudaMemCpy matrix A, B, C to GPU

cublasDgemm(…, device_A, .., device_B, .., device_C, ..)

cudaMemCpy matrix C back to CPU
```

Runtime of cublasDgemm w/ cudaMemCpy (transA='T', transB='N', M=32, N=2400, K=93536)

	GH200	H100-PCIe	GH200	H100-PCIe
Offload Strategy	Strategy 1	Strategy 1	NVBLAS	NVBLAS
Total time	5.50ms	32.80ms	54.8 ms	134.0ms
1. cudaMemcpy†	4.96 ms	31.79ms	-	-
2. cublasDgemm	0.52 ms	0.99ms	-	-
3. other	0.02 ms	0.02ms	-	-

† Including copying matrices A, B and C to GPU memory and C back to host memory.

- A lot more time moving data than compute
- Not useful even with NVLink
- NVBLAS is supposed to be equivalent to SCILIB-Accel Strategy 1
- but NVBLAS has ridiculous overhead, not usable at all.

SCILIB-Accel: data movements strategy 1/3

1. **Mem-copy**: cudaMemCpy Matrices between CPU mem to GPU mem for every CUBLAS call (all other tools do this)

Pseudocode:

```
cudaMalloc  device_A, device_B, device_C

cudaMemCpy matrix A, B, C to GPU

cublasDgemm(…, device_A, .., device_B, .., device_C, ..)

cudaMemCpy matrix C back to CPU
```

PARSEC Test: $Si_{1947}H_{604}$, 2 SCF iterations

Method		application total runtime	dgemm time	data movement
Grace-Grace (144 cores)		415.1s	270.1s	0
Auto offload	**S1: cudaMemcpy**	**425.7s**	**12.4s**	**220.7s**

- **This is why the existing tools are not useful**
- **Data reusability is the key to success**

SCILIB-Accel: data movements strategy 2/3

2. **Coherent Access:** just pass the CPU malloc pointer to GPU kernel
   let **counter-based migration** to move frequently used data to GPU

Pseudocode:

```
cublasDgemm(..., host_A, .., host_B, .., host_C, ..)
```

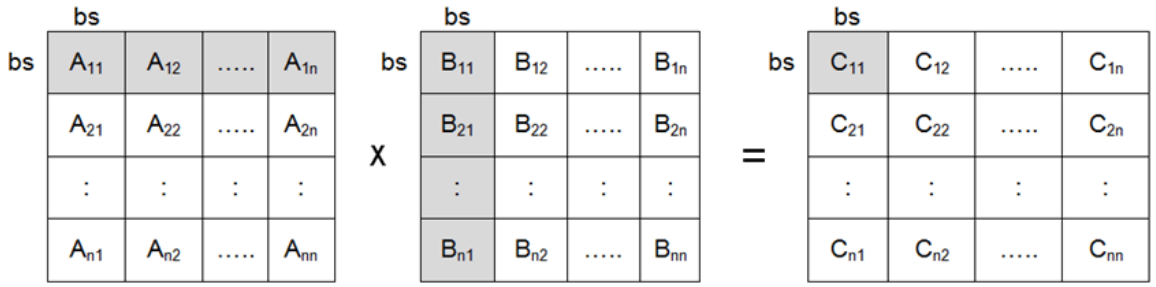
PARSEC Test: $Si_{1947}H_{604}$, 2 SCF iterations

Method		application total runtime	dgemm time	data movement
Grace-Grace (144 cores)		415.1s	270.1s	0
Auto offload	S1: cudaMemcpy	425.7s	12.4s	220.7s
	S2: counter-based migration	**470.0s**	**234.0s**	in dgemm

- Counter-based migration works poorly as of today, it fails to move key data to GPU

How to avoid frequent data movement?

- Case 1: block matrix multiplication, e.g. ScaLAPACK. Every submatrix of A multiplies with submatrix of B



- Case 2: Self-Consistent Field calculation, calculations are performed on the same matrix until convergence is reached.

- Case 3: a sequence of matrix multiplications. Each matrix is involved in multiply BLAS calls

A x B = C
other codes accessing C..
C x D = E
..

- In these common cases, if we move the matrices to GPU in the beginning
- They will be re-used by later BLAS calls
- Assume CPU access of the matrices are relatively trivial (trace, scale, add, etc..), matrices can be kept resident on GPU
- Sounds familiar? **OpenMP first touch**

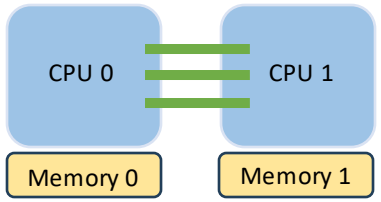
Strategy 3: **GPU first use**

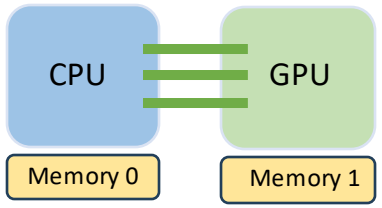
- matrices are moved to GPU mem the first time used by cuBLAS kernel.
- matrices stay on GPU throughout their lifetime

OpenMP first touch vs GPU first use

GPU first use policy resembles features of OpenMP first touch in CPU NUMA programming.

(NUMA: Non-Unified Memory Access)

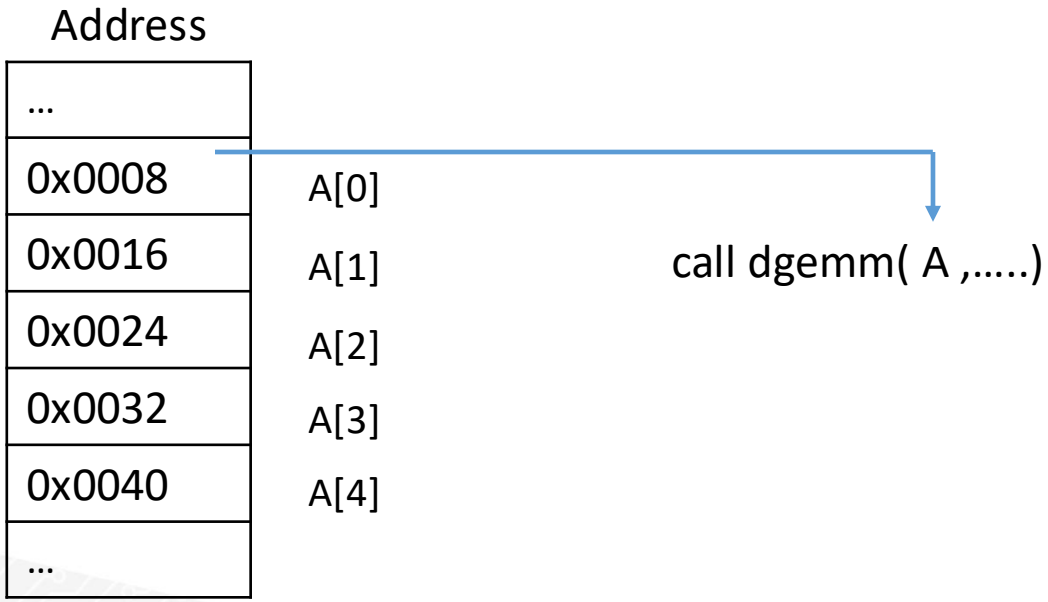




OpenMP First touch	**GPU First use**
for dual socket CPU	for CPU-GPU superchip
allocate space on local memory of OpenMP threads upon initialization	migrate data from CPU memory to local memory of GPU upon first access by GPU kernel
assume remote memory access is trivial (e.g. CPU0 accessing CPU1's memory)	assume remote memory access is trivial (CPU accessing GPU's memory)

How to migrate data CPU ⟺ GPU (1)

- At the time BLAS symbols are intercepted, data has already been allocated on CPU
- Can't reallocate as we only have array address in BLAS subroutine
- How to migrate the data to GPU?



How to migrate data CPU ⟺ GPU (2)

Review of memory management in modern operating system

Page: smallest unit of data for memory management in an operating system that uses virtual memory

When your code asks for an array, you get a contiguous memory address without worrying about physically details.

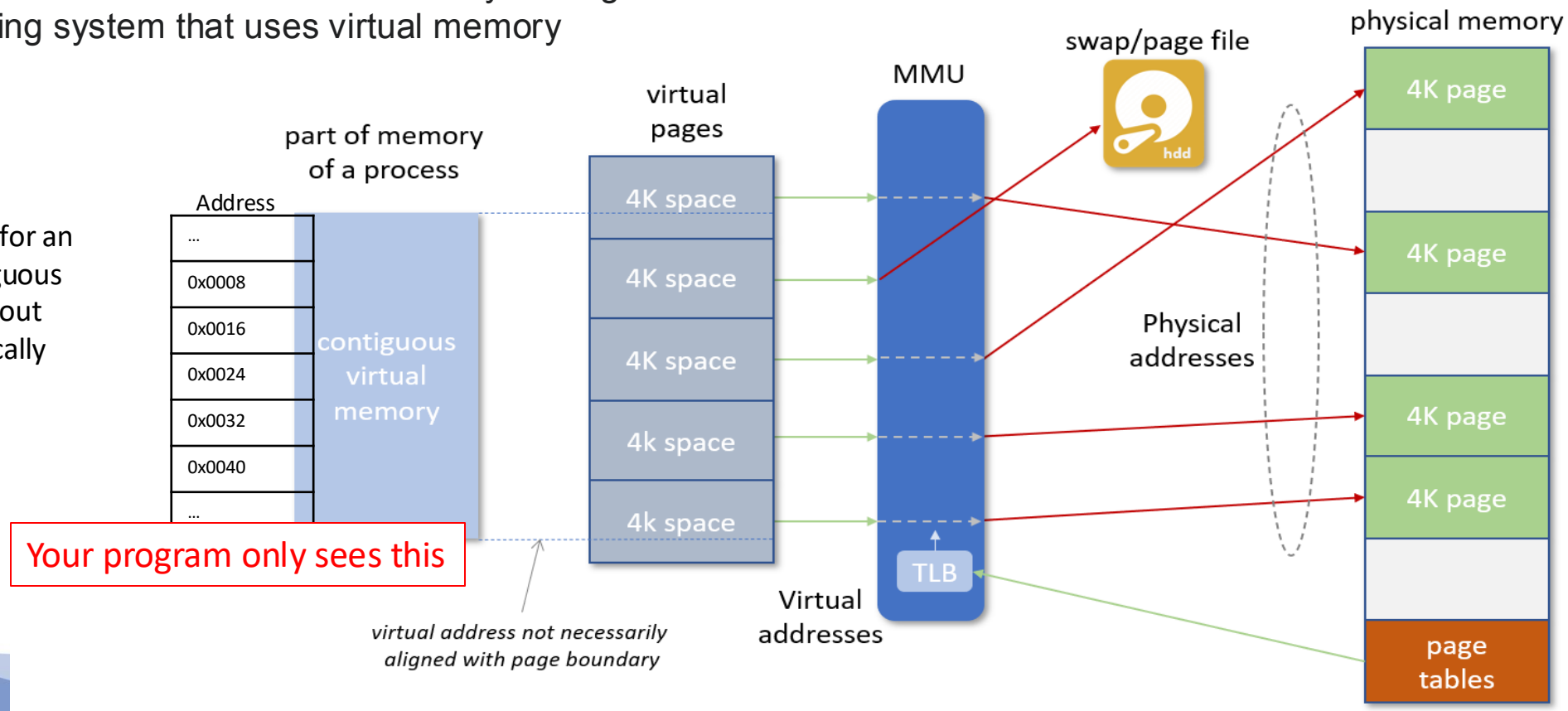
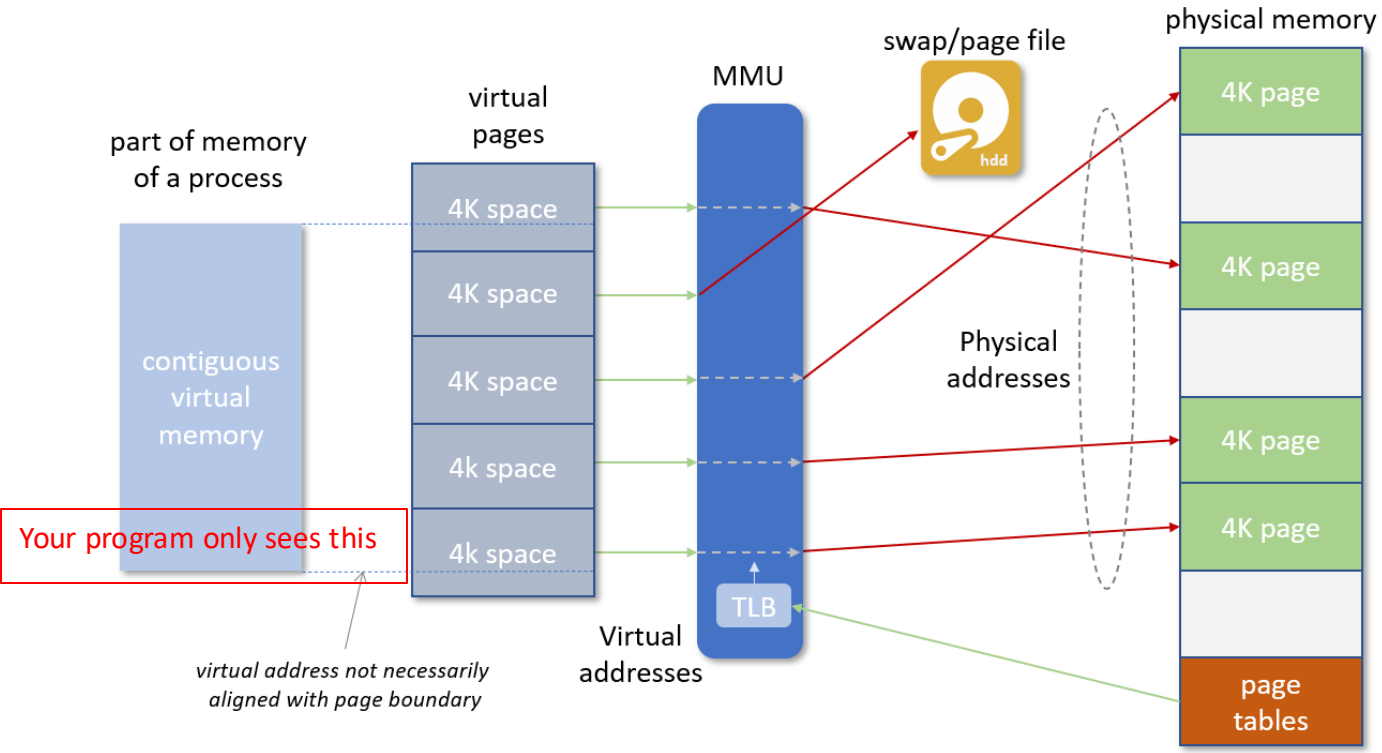


Figure source: https://www.linkedin.com/pulse/demystifying-memory-sub-systems-part-2-virtual-simon-southwell

How to migrate data CPU ⟺ GPU (3)



- On Grace-Hopper
  - CPU mem (LPDDR5x) in NUMA0
  - GPU mem (HBM) in NUMA1
  - Single page table
- Similar to dual-socket CPU system

- Move data without reallocating
  1. Copy physical page from NUMA0 to NUMA1
  2. Unmap original NUMA0 page
  3. Update page table
  4. No change to virtual page
     -> your code sees nothing different

- Linux system call move_pages()
- Move page NUMA 0 -> NUMA1
  (CPU) (GPU)
- https://docs.kernel.org/mm/page_migration.html

Figure source: https://www.linkedin.com/pulse/demystifying-memory-sub-systems-part-2-virtual-simon-southwell/

SCILIB-Accel: data movements strategy 3/3

GPU First Use Policy: matrices are moved to GPU the first time they are used by cuBLAS.

- Matrices stay on GPU memory throughout their lifetime
- Implemented by migrating memory pages from NUMA0 to NUMA1 (move_pages).

Pseudocode:

```
if (on numa 0) move_pages (host_A, to NUMA1) ;

if (on numa 0) move_pages (host_B, to NUMA1) ;

if (on numa 0) move_pages (host_C, to NUMA1) ;

cublasDgemm(…, host_A, .., host_B, .., host_C, ..)
```


J. Li, Y. Wang, X. Liang, H. Liu, "Automatic BLAS Offloading on Unified Memory Architecture: A Study on NVIDIA Grace-Hopper", https://arxiv.org/abs/2404.13195

SCILIB-Accel: data movements strategy 3

First Use Policy, best performance!

PARSEC performantly runs on GPU for the first time ever!

Matrix reuse: every matrix moved to GPU gets reused 570 times on average by subsequent dgemm calls.

Setup		application total runtime	dgemm time	data movement
Grace-Grace (144 cores)		415.1s	270.1s	0
Grace-Hopper Auto offload	S1: CudaMemCpy	425.7s	12.4s	220.7s
	S2: counter-based migration	470.0s	234.0s	in dgemm
	S3: GPU First Use	**220.3s**	**29.1s**	**1.3s**

Matrix reuse: 570

Application Tests: MuST (1)

MuST (Multiple Scattering Theory) is an ab initio electronic structure calculation software suite. It's LSMS method performs DFT calculation with linear scalability to system size (Gordon-Bell prize 1998 & 2009)

https://github.com/mstsuite/MuST

- LSMS in MuST
- 5600-atom CoCrFeMnNi alloy
- 50 Grace-Grace CPU nodes
- 5,600 CPU cores used
- >80% runtime in BLAS

```
--------------------------------------------------------------------------------
                Scientific Library Profiler
--------------------------------------------------------------------------------
for application: /scratch/07893/junjieli/must/vista-large-scale-test/mst2
total runtime (s):   2614.173, library time (s):   2207.067, lib percentage: 84.4%.

                    Exclusive Times
--------------------------------------------------------------------------------
      group    function        count [imb%]        time(s) [imb%]
--------------------------------------------------------------------------------
1      BLAS     zgemm_      35947260 [  0.0%]     1581.283 [ 14.3%]
2      BLAS     ztrsm_      86688 [  0.0%]      571.972 [ 13.6%]
3    LAPACK     zgetrf_       1968 [  0.0%]       51.125 [ 12.1%]
4    LAPACK     zlaswp_       1968 [  0.0%]        2.327 [ 19.6%]
5      BLAS     zcopy_      127381 [ 25.5%]       0.296 [105.1%]
6      BLAS     zaxpy_      316800 [  0.0%]       0.027 [ 15.8%]
7    LAPACK     zgetrs_       1968 [  0.0%]       0.017 [ 37.4%]
8      BLAS     dscal_           4 [  0.0%]       0.008 [811.2%]
9      BLAS     zscal_        8736 [  0.0%]       0.006 [ 22.9%]
10     BLAS     izamax_       1584 [ 39.4%]       0.004 [ 67.2%]
11     BLAS     zgeru_         576 [  0.0%]       0.001 [263.2%]
12     BLAS     dcopy_         152 [  0.0%]       0.000 [ 40.8%]
13   LAPACK     ilaenv_         24 [  0.0%]       0.000 [167.7%]
--------------------------------------------------------------------------------
                    total library time (s): 2207.067
--------------------------------------------------------------------------------
 * exclusive time: time excluding all children functions
 * average count and timing reported.
 * imblance metric: imb% = (max-min)/avg*100%.
```

Application Tests: MuST (2)

- Native CUDA implementation available using cuSolver.
- SCILIB-Accel auto offload performs 2x better than the native CUDA code.

Matrix reuse: every matrix moved to HBM gets reused 780 times on average by subsequent zgemm & ztrsm calls.

Setup:

- 5600-atom CoCrFeMnNi alloy, 32 energy points
- 50 Grace-Grace (**GG**) CPU nodes
- or 50 Grace-Hopper (**GH**) GPU nodes

Setup	Total runtime	zgemm+ztrsm	Data movement
GG, 5600 CPU cores used	2450s	~2153s	0
GH, native CUDA	1685s	NA	NA
GH, auto offload (S1: cudaMemcpy)	1098s	450s	337s
GH, auto offload (S3: GPU first-use policy)	**824s**	**600s**	**5.0s**

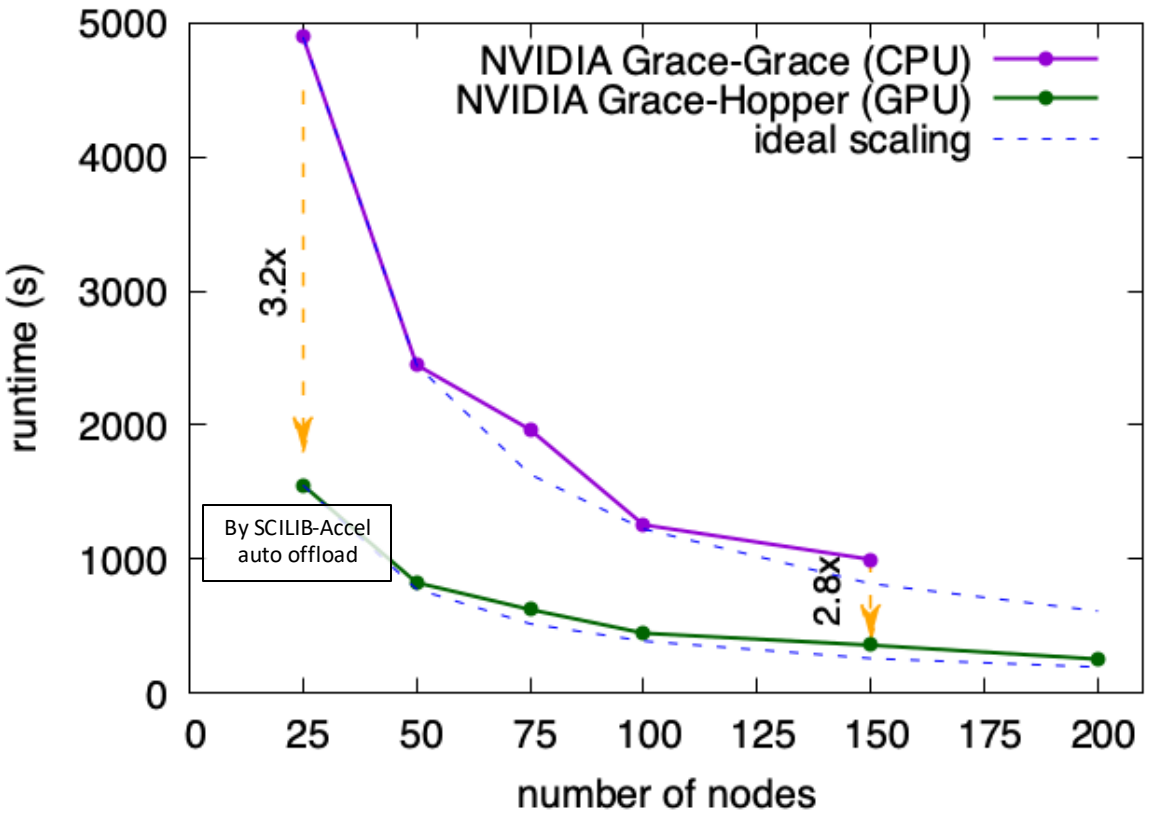
matrix reuse 780

Performance issue of GPU kernel accessing malloc HBM data

Application Tests: MuST (3)

- Using the optimal S3: GPU First Use policy
- Highly scalable with SCILIB-Accel auto BLAS offload
- Consistently 2x faster than native CUDA port

Problem Size: 5600 atoms, 32 energy points				
Node Count	Time including IO (s)			GPU/CPU speedup
	CPU: (2x Grace) 144 cores/node	GPU: Native CUDA	**GPU: SCILIB-Accel auto offload**	
25	4896.4	3223.3	1550.9	3.2x
50	2449.5	1685.2	823.8	3.0x
75	1965.8	1244.7	623.1	3.2x
100	1255.2	903.9	446.8	2.8x
150	997.0	673.6	357.5	2.8x
200	NA	493.9	253.3	NA



- TACC charges 1 SU for GH node-hour, and 0.33 SU for GG node-hour
- >3x speedup means SU profitability
- **savings in both time and cost**

SCILIB-Accel: easy to use

Just LD_PRELOAD it!
No matter if BLAS is statically or dynamically linked.
All level-3 BLAS are implemented!

```
LD_PRELOAD=$PATH_TO_LIB/scilib-dbi.so
run your cpu code.
```

get code from here
https://github.com/nicejunjie/scilib-accel

Summary

- Auto BLAS offload is made performant for the first time
- LD_PREPLOAD my library, and run CPU binary on GPU
- Easy way to explore GPU capability for BLAS heavy codes
- Get incentives to fully port CPU codes to GPU
- Performance may further improve as there is still outstanding performance issues on Grace-Hopper

https://github.com/nicejunjie/scilib-accel

Outlooks

- Similar methodology will also work on AMD GPUs
- Especially the AMD MI300A APU where CPU + GPU are packed with the same HBM memory.
  - MI300A available on El Capitain @ Lawrence Livermore National Laboratory
  - All math libs possible: BLAS, LAPACK, FFTW, etc…
- Promising to port dozens of legacy quantum chem/physics codes (in particular real-space or plane-wave based where matrices are large) to GPU with acceptable performance.

Comments and questions?

nicejunjie@gmail.com