Geometric and Algebraic Multigrid Solvers in PETSc on Many-GPU Supercomputer Architectures

Richard Tran Mills

(contributions from Hannah Morgan, Mark Adams, Karl Rupp, Hong Zhang, and Barry Smith)

SIAM Conference on Parallel Processing for Scientific Computing
February 13, 2020

PETSc: The Portable, Extensible Toolkit for Scientific Computation

- PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations; it also incorporates the Toolkit for Advanced Optimization (TAO) library for solving numerical optimization problems.
- I will talk about GPU developments in PETSc, with focus on its multigrid solver framework.
- Multigrid methods are asymptotically optimal solvers for important classes of linear systems:
  - Cycle through a hierarchy of discretizations of the problem, restricting approximate solutions onto coarser grids to obtain corrections that are prolongated back to finer grids
  - Simple iterative methods (Jacobi, SOR, Chebyshev) called *smoothers* (because they quickly reduce high-frequency components of the error) are applied on each level. (Involves SpMV or similar operations.)

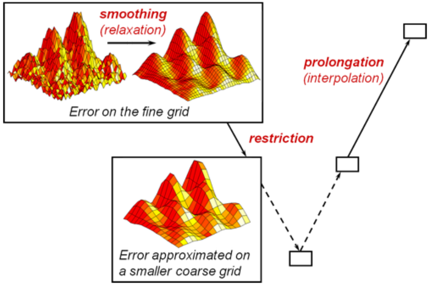


Illustration of multigrid v-cycle.

`https://fastmath-scidac.llnl.gov/research/multigrid-and-multilevel-methods.html`

What is driving current HPC trends?

Moore's Law (1965)

- Moore's Law: Transistor density doubles roughly every two years
- (Slowing down, but reports of its death have been greatly exaggerated.)
- For decades, single core performance roughly tracked Moore's law growth, because smaller transitors can switch faster.

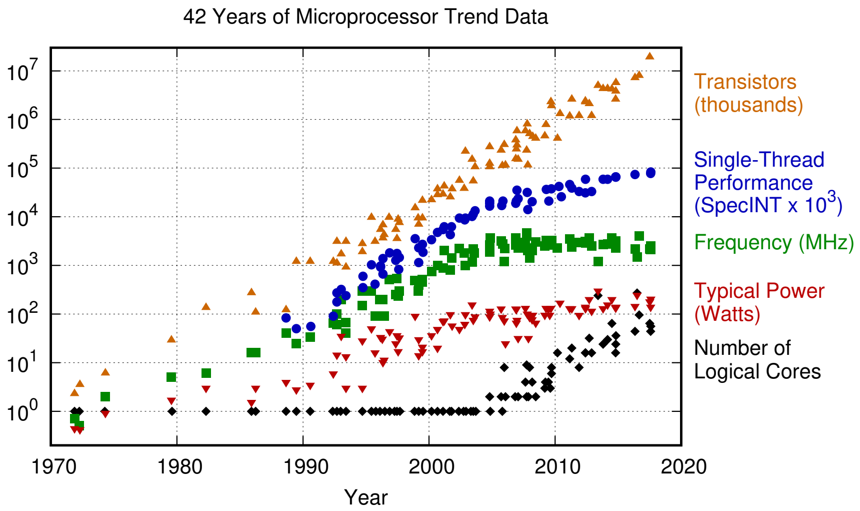
Dennard Scaling (1974)

- Dennard Scaling: Voltage and current are proportional to linear dimensions of a transistor; therefore power is proportional to the area of the transistor.
- Ignores leakage current and threshold voltage; past 65 nm feature size, Dennard scaling breaks down and power density increases, because these don't scale with feature size.

Power Considerations

- The "power wall" has limited practical processor frequencies to around 4 GHz since 2006.
- Increased parallelism (cores, hardware threads, SIMD lanes, GPU warps, etc.) is the current path forward.

Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Current trends in HPC architectures

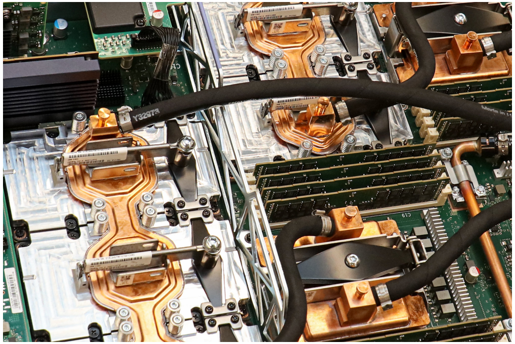
Emerging architectures are very complex...

- Lots of hardware cores, hardware threads
- Wide SIMD registers
- Increasing reliance on fused-multiply-add (FMA), with multiple execution ports, proposed quad FMA instructions
- Multiple memories to manage (multiple NUMA nodes, GPU vs. host, normal vs. high-bandwidth RAM, byte-addressable NVRAM being introduced, ...)
- Growing depth of hierarchies: in memory subsystem, interconnect topology, I/O systems
- GPU accelerators now providing bulk of computing power for most new supercomputers

...and hard to program

- Vectorization may require fighting the compiler, or entirely re-thinking algorithm.
- Must balance vectorization with cache reuse.
- Host vs. offload adds complexity; large imbalance between memory bandwidth on device vs. between host and device
- Growth in peak FLOP rates have greatly outpaced available memory bandwidth.

OLCF Summit Supercomputer



System totals

- $\sim$ 200 PFlop/s theoretical peak
  143 PFlop/s LINPACK—#1 in TOP500
- 4,608 compute nodes

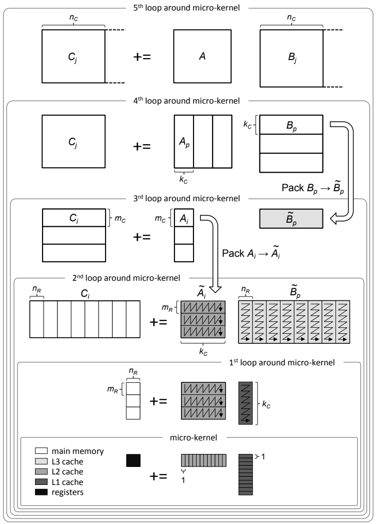
Node configuration

- Compute:
  - Two IBM Power9 CPUs, each 22 with cores, 0.5 DP TFlop/s
  - Six NVIDIA Volta V100 GPUs, each with 80 SMs–32 FP64 cores/SM, 7.8 DP TFlop/s
- Memory:
  - 512 GB DDR4 memory
  - 96 (6×16) GB high-bandwidth GPU memory
  - 1.6 TB nonvolatile RAM (I/O burst buffer)

Almost all compute power is in GPUs!

Some principles guiding our development work

- Defer algorithmic choices until execution time, and enable complex composition of multi-layered solvers via runtime options
- Strive to separate *control logic* from *computational kernels*
  - Allow injecting new hardware-specific computational kernels without having to rewrite the entire solver software library
- Hand-optimize small kernels only, and design to maximize reuse of such kernels
  - Cf. the BLIS framework, which expresses all level-3 BLAS operations in terms of one micro-kernel.
- Reuse existing, specialized libraries (e.g., MKL, cuSPARSE) when feasible


5th loop around micro-kernel
n_C
C_j
+=
A
B_j
4th loop around micro-kernel
A_p
k_C
B_p
Pack $B_p \to \tilde{B}_p$
3rd loop around micro-kernel
C_i
m_C
A_i
$\tilde{B}_p$
Pack $A_i \to \tilde{A}_i$
2nd loop around micro-kernel
n_R
m_R
$\tilde{A}_i$
1st loop around micro-kernel
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

[F. Van Zee, T. Smith, ACM TOMS 2017]

Overview of GPU Support in PETSc

Transparently use GPUs for common matrix and vector operations, via runtime options — no change of user code required.

CUDA/cuSPARSE:

- CUDA matrix and vector types:
  `-mat_type aijcusparse -vec_type cuda`
- GPU-enabled preconditioners:
  - GPU-based ILU: `-pc_type ilu -pc_factor_mat_solver_type cusparse`
  - Jacobi: `-pc_type jacobi`

ViennaCL:

- ViennaCL matrix and vector types:
  `-mat_type aijviennacl -vec_type viennacl`
- Compute backend selection (CUDA, OpenCL, or OpenMP):
  `-viennacl_backend opencl`
- Switch between CUDA, OpenCL, or OpenMP (CPU) at runtime
- GPU-enabled preconditioners:
  - Fine-grained parallel ILU: `-pc_type chowiluviennacl`
  - Smoothed aggregation AMG: `-pc_type saviennacl`

GPU Support—How Does it Work?

Host and Device Data

```
struct _p_Vec {
  ...
  void              *data;        // host buffer
  void              *spptr;       // device buffer
  PetscOffloadMask offloadmask;  // indicates which copies are valid
};
```

Possible Flag States

```
  typedef enum {PETSC_OFFLOAD_UNALLOCATED,
               PETSC_OFFLOAD_GPU,
               PETSC_OFFLOAD_CPU,
               PETSC_OFFLOAD_BOTH} PetscOffloadMask;
```

GPU Support—How Does it Work?

Fallback-Operations on Host

- Data becomes valid on host (`PETSC_OFFLOAD_CPU`)

```
PetscErrorCode VecSetRandom_SeqCUDA_Private(..) {
  VecGetArray(...);
  // some operation on host memory
  VecRestoreArray(...);
}
```

Accelerated Operations on Device

- Data becomes valid on device (`PETSC_OFFLOAD_GPU`)

```
PetscErrorCode VecAYPX_SeqCUDA(..) {
  VecCUDAGetArrayReadWrite(...);
  // some operation on raw handles on device
  VecCUDARestoreArrayReadWrite(...);
}
```

Example

KSP ex12 on Host

-
  ```
  $> ./ex12
      -pc_type none -m 200 -n 200 -log_summary
  ```

  ```
  KSPGMRESOrthog      1630 1.0 4.5866e+00
  KSPSolve               1 1.0 1.6361e+01
  ```

KSP ex12 on Device

-
  ```
  $> ./ex12 -vec_type cuda -mat_type aijcusparse
      -pc_type none -m 200 -n 200 -log_summary
  ```

  ```
  MatViennaCLCopyTo          1 1.0 5.6108e-02
  KSPGMRESOrthog      1630 1.0 5.5989e-01
  KSPSolve               1 1.0 1.0202e+00
  ```

GPU Pitfalls

Pitfall: Function Pointers

- Pass CUDA function “pointers” through library boundaries?
- OpenCL: Pass kernel sources, user-data hard to pass
- Composability?

Pitfall: Repeated Host-Device Copies

- PCI-Express transfers kill performance
- Complete algorithm needs to run on device
- Problematic for explicit time-stepping, etc.

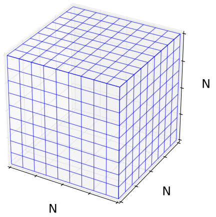
Pitfall: Wrong Data Sizes

- Data set too small: Kernel launch latencies dominate
- Data set too big: Out of memory

GPU Pitfalls

Pitfall: GPUs are too fast for PCI-Express

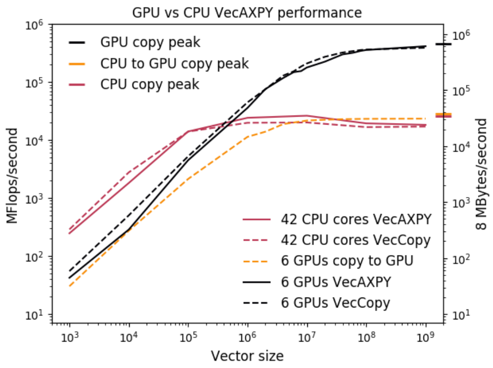
- Example system: 720 GB/sec from GPU-RAM, 16 GB/sec for PCI-Express
- 40x imbalance! (Gets better with NVLink, but large imbalance remains.)



Compute vs. Communication

- Take $N = 512$, so each field consumes 1 GB of GPU RAM
- Boundary communication: $2 \times 6 \times N^2$: 31 MB
- Time to load field: 1.4 ms
- Time to load ghost data: **1.9 ms (!!)**

AXPY operations on Summit: CPU vs. GPU



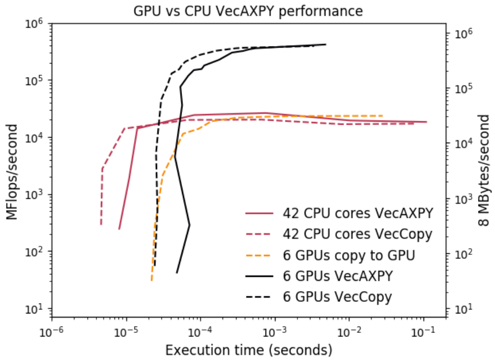


Figure: Effect of vector size on AXPY performance and memory throughput (one MPI rank per GPU) on a single node of the OLCF Summit system. As vector sizes become large, GPUs perform significantly better than the 42 Power9 CPU cores available, but the CPU cores can be significantly faster for smaller vector sizes, due to much smaller CPU latency. Figure on the right presents an alternative, *work-time spectrum* view of the data: in this view, both asymptotic bandwidth and latency of the operations can be read directly from the figure.

Table: Summary of PETSc vector operation performance on Summit, using a linear model in which time for data transfer is characterized by a start-up latency l and a bandwidth b (subscripts C and G denote CPU and GPU, respectively). In our linear model, the fraction of peak achieved will be $beta = \frac{n}{l \times b}$. *Small* vectors have $n = 10^3$–10^5 entries, *medium* 10^5–10^7, and *large* 10^7–10^8. Latency is in 10^{-6} seconds, bandwidth in 8,000 Mbytes/second.

		Vec size	CPU	interconnect	GPU
Sockets			2		2
Cores/GPUs			42		6
Latency					
	VecDot				
		small	17	-	70
		large	-	-	89
	VecAXPY				
		small	9	-	47
		large	-	-	89
	VecCopy				
		small	4	48	-
		large	-	43	32
Bandwidth					
	VecDot				
		medium	32		567
		large	33		667
	VecAXPY				
		medium	40		375
		large	28		627
	VecCopy				
		small	36	42	-
		medium	32	-	559
		large	24	35	593
Launch time of null kernel				10	
CPU-GPU synchronization after launch				11	
CPU-GPU synchronization when free				6	
Size for 90% of peak	VecAXPY		2.268×10^6		5.022×10^8
Nodes for the time of $10 l_G$			20.2		1
Nodes for the time of $10 l_C$			222		-

Binding an object to CPU

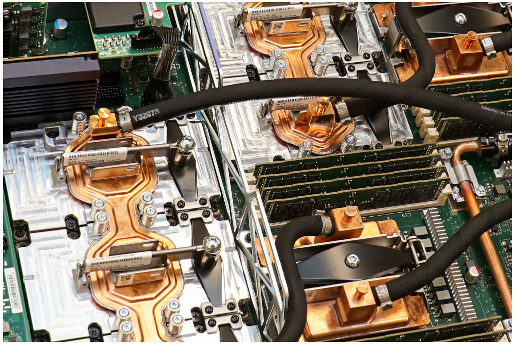
```
struct _p_Vec {
  ...
  void             *data;          // host buffer
  void             *spptr;         // device buffer
  PetscOffloadMask offloadmask;   // indicates which copies are valid
  PetscBool        boundtocpu;    // flag: perform operations on CPU only?
};
```

```
/*@
     MatBindToCPU - marks a matrix to temporarily stay on the CPU and
         perform computations on the CPU

   Input Parameters:
+    A - the matrix
-    flg - bind to the CPU if value of PETSC_TRUE

   Level: intermediate
@*/
PetscErrorCode MatBindToCPU(Mat A,PetscBool flg)
{
...
}
```

OLCF Summit Supercomputer



System totals

- $\sim$ 200 PFlop/s theoretical peak
  143 PFlop/s LINPACK—#1 in TOP500
- 4,608 compute nodes

Node configuration

- Compute:
  - Two IBM Power9 CPUs, each 22 with cores, 0.5 DP TFlop/s
  - Six NVIDIA Volta V100 GPUs, each with 80 SMs–32 FP64 cores/SM, 7.8 DP TFlop/s
- Memory:
  - 512 GB DDR4 memory
  - 96 (6×16) GB high-bandwidth GPU memory
  - 1.6 TB nonvolatile RAM (I/O burst buffer)

Almost all compute power is in GPUs!

Early Summit Results: SNES ex19 with geometric multigrid

Running SNES ex19 (velocity-vorticity formulation for nonlinear driven cavity) with 37.8 million total degrees of freedom on single Summit node.

CPU only command line:

```
jsrun -n 6 -a 7 -c 7 -g 1 ./ex19 -cuda_view -snes_monitor -pc_type mg
    -da_refine 10 -snes_view -pc_mg_levels 9 -mg_levels_ksp_type chebyshev
    -mg_levels_pc_type jacobi -log_view
```

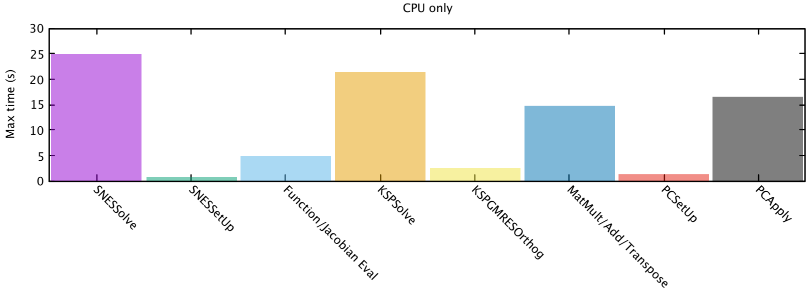
(6 resource sets, each assigned 7 MPI ranks, 7 cores, 1 GPU: 42 MPI ranks/cores, 6 GPUs total)

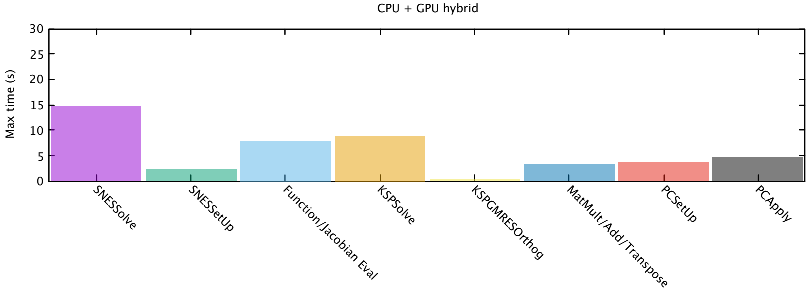
CPU + GPU hybrid command line:

```
jsrun -n 6 -a 4 -c 4 -g 1 ./ex19 -cuda_view -snes_monitor -pc_type mg
    -dm_mat_type aijcusparse -dm_vec_type cuda -da_refine 10 -snes_view
    -pc_mg_levels 9 -mg_levels_ksp_type chebyshev -mg_levels_pc_type jacobi
    -log_view
```

(6 resource sets, each assigned 4 MPI ranks, 4 cores, 1 GPU: 24 MPI ranks/cores, 6 CPUs total)

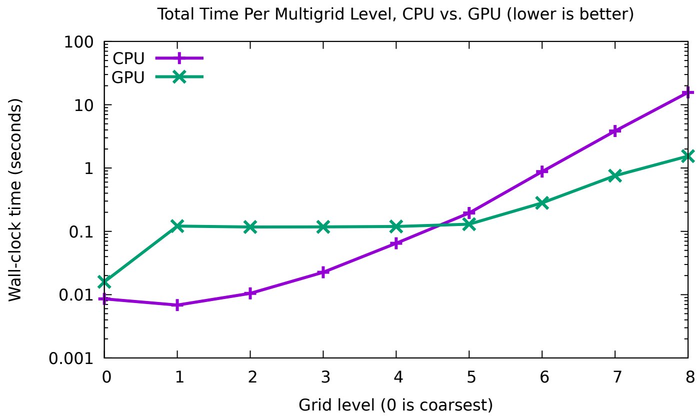
Early Summit Results: SNES ex19 with geometric multigrid





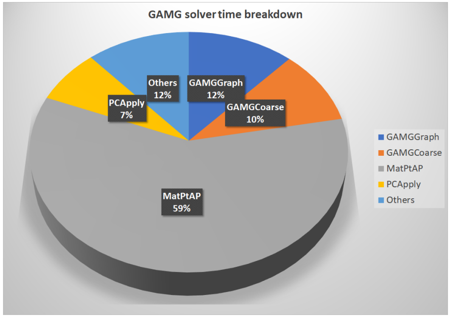
SNES ex19 with multigrid, times per level

Can examine time per multigrid level (use `-pc_mg_log`) for 24 ranks on CPU vs. GPU:



- GPU latency effects apparent in flat line for levels 1–5.
- 52 direct solves on coarsest level (0)
- 52 MGResid, 104 MGSmooth, 104 MGInterp on levels > 0
- Five coarsest levels are faster on CPU (about 4X faster in total).
- Potential 1.14X speedup in MG Apply by placing levels 0–4 on CPU

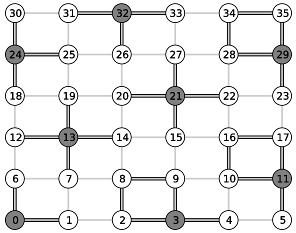
GAMG Execution Time Breakdown: SNES ex56 on Single Summit Node



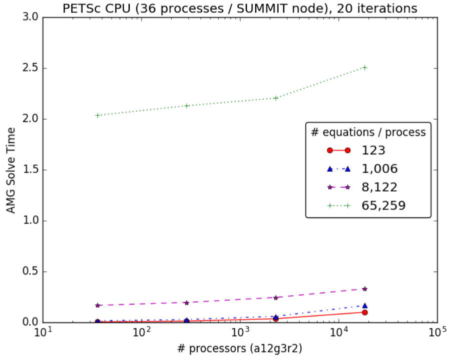
- GAMG provides native (uses PETSc PCMG framework) smoothed aggregation algebraic multigrid in PETSc
- At left, breakdown of GAMG performance on SNES ex56 linear elasticity example on single Summit node
- PCApply: Smoothers and cycling through levels. Running fairly well on GPU.
- GAMGGraph + GAMGCoarse: Mesh setup; on CPU
- MatPtAP: Coarsened operator setup; on CPU (where it benefits from large caches)

Developing GPU implementation maximal independent sets (used in GAMGCoarsen)

- GAMG CPU implementation is targeted for distributed memory and is ill-suited for conversion to CUDA routines.
- Consider reusing AmgX code, but difficult because of differences in decomposition across MPI ranks.
- ViennaCL algorithm exploits fine-grain parallelism but cannot be used directly on PETSc's parallel matrix.



Performance of GAMG on Summit at Scale



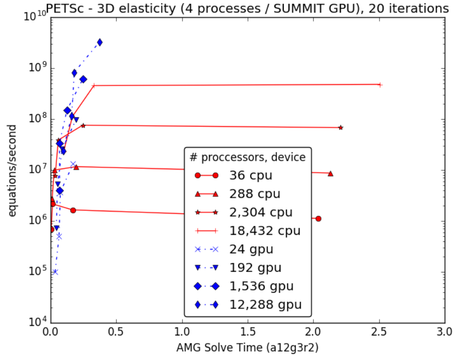


Figure: Initial performance of GAMG on Summit at scale, running the SNES ex56 3D linear elasticity benchmark. Preconditioner setup is performed on the CPU, while the multigrid solve happens on the GPU. The left plot is a weak scaling plot (flat lines are desired) for CPU only runs; the right plot shows a work-time spectrum view (GPU and CPU) that shows both latency and asymptotic throughput. **Speedup of GPU vs. CPU at scale is about 12.**

Summary and Future Directions

Current GPU support in PETSc

- GPU support is a key focus in recent releases of PETSc; NVIDIA, AMD, and Intel GPUs supported.
- Basic matrix and vector primitives supported, as well as simple multigrid smoothers (Jacobi, Chebyshev)
- Support for some complex on-node preconditioners (GPU-friendly ILU, smoothed aggregation) through ViennaCL
- PCMG multigrid framework fully supports GPUs and shows good performance on machines like Summit
- GAMG algebraic multigrid displays good performance in numerical "solve" phase on GPUs
- For NVIDIA machines, CUDA-aware MPI now supported (recent work by Junchao Zhang)

Significant work on GPGPU support in PETSc is also ongoing

- Algorithmic developments: GPU-enabled mesh setup, numerical setup for GAMG; new SpMV approaches
- Continuing to look for performance optimizations for GPU/accelerators both major (e.g., recent to use CUDA-aware MPI) and minor (e.g., identifying additional places to use XXXBindToCPU(), adding more control over use of pinned host memory, etc.)
- Adding additional GPU back-ends: Intel oneAPI MKL, Kokkos kernels, libaxb (meta back-end), etc.
- Performance engineering work for upcoming systems (Aurora 21, Frontier)
- As always, input from users should guide this work—let us hear from you!