Enabling End-to-End Accelerated Multiphysics Simulations in the Exascale Era Using PETSc

Richard Tran Mills, Argonne National Laboratory, USA (speaker)
Jed Brown, University of Colorado, Boulder, USA

11th EGU Galileo Conference: The Solid Earth and Geohazards in the Exascale Era
Barcelona, Spain, May 24, 2023

With additional contributions from Mark Adams, Satish Balay, Alp Dener, Jacob Faibussowitsch, Matthew Knepley, Scott Kruger, Hannah Morgan, Todd Munson, Karl Rupp, Patrick Sanan, Barry Smith, Stefano Zampini, Hong Zhang, and Junchao Zhang

PETSc TAO

ECP EXASCALE COMPUTING PROJECT

Who is this guy?

- May 2017–present, Computational Scientist at Argonne National Laboratory, working on PETSc development and applications using it
- January 2014–May 2017, HPC Earth System Models Architect at Intel
  - Part of the Many Integrated Core (MIC) program developing Xeon Phi
  - Hardware/software co-design in weather, climate, and Earth system models
- August 2004–January 2014, Staff scientist at Oak Ridge National Laboratory
  - Started in NCCS/OLCF (supercomputing center), moved to Computer Science and Mathematics Division, then Environmental Sciences Division
  - Joint faculty appointment (departments of Earth and Planetary Sciences; Computer Science) University of Tennessee, 2010—2014
- 2001–2004, DOE Computational Science Graduate Fellow, Dept. of Computer Science at William and Mary (PhD 2004). Practicum in Earth and Environmental Sciences Division, Los Alamos National Laboratory (initial development of subsurface flow and reactive transport code PFLOTRAN)
- 1995–1999, Undergraduate studying geology, physics, computing at University of Tennessee. Research in computational geomorphology model development, neural network processing of remote sensing data
- Earlier: Lots of photos of me standing by roadcuts, etc., ("for scale") with geology professor father

Outline

- Exascale: How did we get here? What are the trends and their implications?
- Brief overview of PETSc, the Portable, Extensible Toolkit for Scientific Computation and (mostly historical) survey of enabling extreme inter-node scalability with it
- Enabling extreme intra-node scalability (focus during DOE Exascale Computing Project)
  - Fundamental challenges posed by GPUs
  - PETSc's design for GPU computing
- Experiences with PETSc on GPUs; Practical challenges we encountered and our workarounds; Advice for library and application developers
- Exemplar of best practices for achieving end-to-end GPU acceleration with PETSc (and libCEED): Solid mechanics with Ratel
- Summary and philosophy / vision for the future

Gedanken experiment:

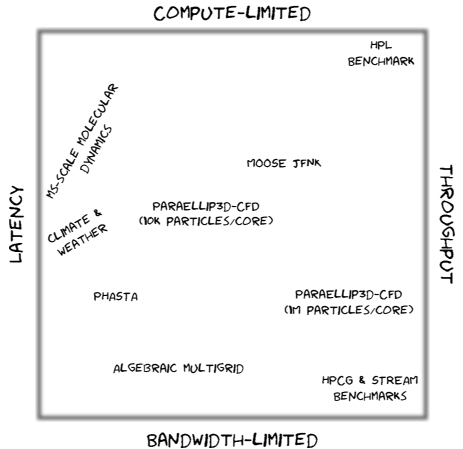
How to use a jar of peanut butter as its price slides downward?

- In 2007, at $3.20: make sandwiches
- By 2010, at $0.80: make recipe substitutions for other oils
- By 2013, at $0.20: use as feedstock for biopolymers, plastics, etc.
- By 2016, at $0.05: heat homes
- By 2019, at $0.0125: pave roads ☺

The cost of computing has been on a curve *much better than this* for two decades and promises to continue for at least one more. Like everyone else, scientists should plan increasing uses for it…

Constants Matter!

- Prof. Keyes' (quite valid) point was that the exponential growth of computing power promises to enable novel and prominent uses of modeling and simulation.
- BUT, the methods we use depend not just on scaling of available compute, but on important constants that have been changing as hardware evolves. E.g., relative cost of compute vs. memory access, CPU vs. GPU latencies



Constants Matter!

- Prof. Keyes' (quite valid) point was that the exponential growth of computing power promises to enable novel and prominent uses of modeling and simulation.
- BUT, the methods we use depend not just on scaling of available compute, but on important constants that have been changing as hardware evolves. E.g., relative cost of compute vs. memory access, CPU vs. GPU latencies
- Perhaps if peanut-butter production followed trajectories like computing in HPC centers:
  - No one would make sandwiches: peanut butter costs $0.0125 per jar's worth, but is only sold in pallets of 55-gallon drums that ship from a distribution center hundreds of miles away (high throughput, high latency)
  - Boutique peanut farmers and processors are gone; there are only a few high-volume vendors that strip-mine peanuts and each produce only one variety of peanut butter. (And be careful mixing varieties from different vendors: interactions between chemical additives cause difficulties.) 😁

What has driven HPC architecture trends as we approached the Exascale Era?

Moore's Law (1965)

- Moore's Law: Transistor density doubles roughly every two years
- For decades, single core performance roughly tracked Moore's law growth, because smaller transistors can switch faster.
- Moore's law is dead (if you ask NVIDIA) or alive and well (if you ask Intel)

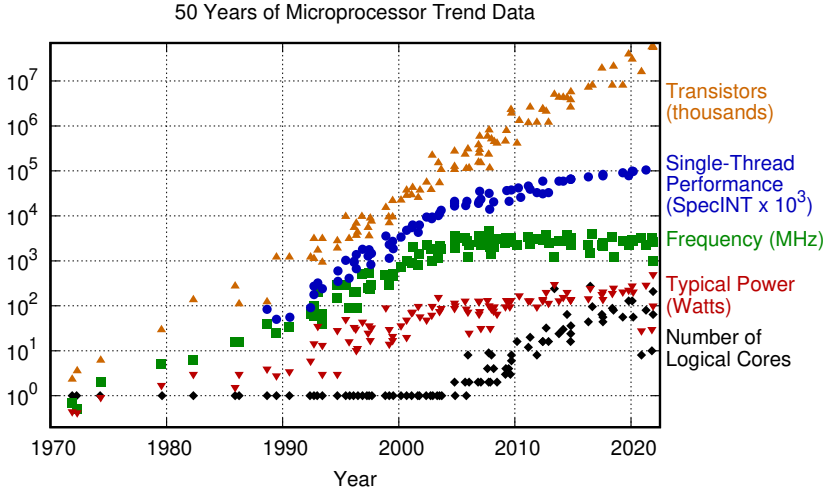
Dennard Scaling (1974)

- 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 (nodes, cores, hardware threads, SIMD lanes, GPU warps, etc.) has been the 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 2010-2021 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

Current trends in HPC architectures

Emerging architectures are very complex...

- Lots of nodes, hardware cores, hardware threads (most visible in extremely powerful GPUs)
- Reliance on wide SIMD, wider SIMT
- 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

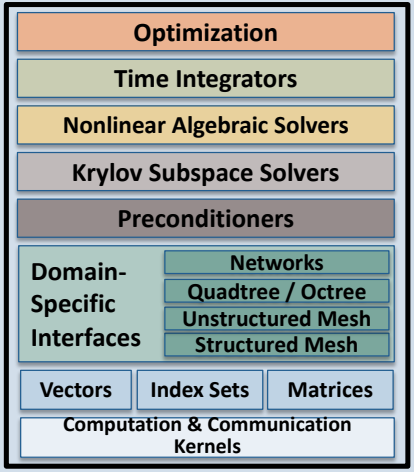
...and hard to program (at least in a performant manner)

- Growth in peak FLOP rates have greatly outpaced available memory bandwidth.
- 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

PETSc TAO

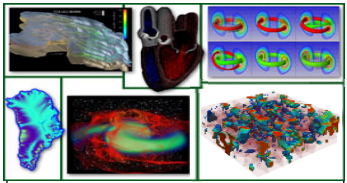
Portable, Extensible Toolkit for Scientific Computation / Toolkit for Advanced Optimization

Scalable algebraic solvers for PDEs. Encapsulate parallelism in high-level objects. Active & supported user community. Full API from Fortran, C/C++, Python.



- **Easy customization and composability of solvers at runtime**
  - Enables optimality via flexible combinations of physics, algorithmics, architectures
  - Try new algorithms by composing new/existing algorithms (multilevel, domain decomposition, splitting, etc.)
- **Portability & performance**
  - Largest DOE machines, also clusters, laptops
  - Thousands of users worldwide

Argonne NATIONAL LABORATORY



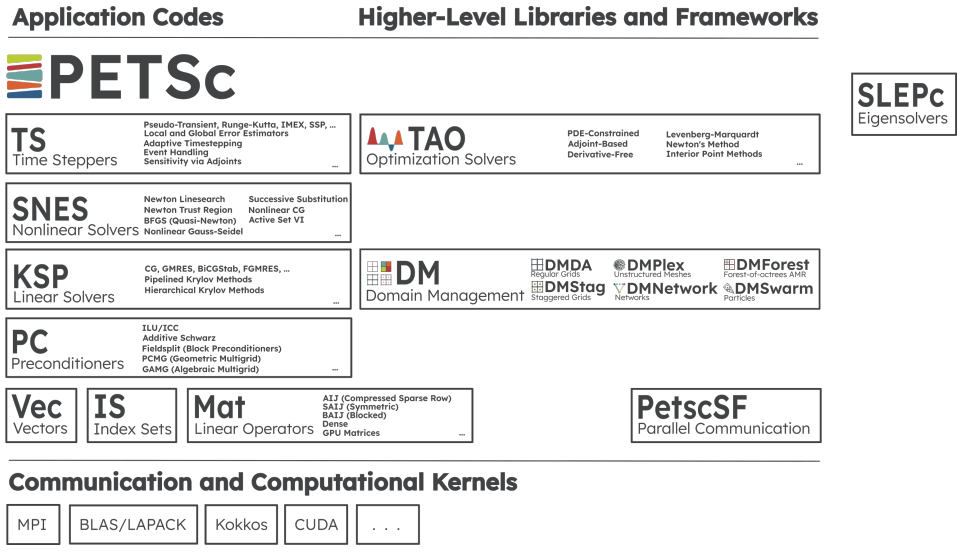
PETSc provides the backbone of diverse scientific applications. clockwise from upper left: hydrology, cardiology, fusion, multiphase steel, relativistic matter, ice sheet modeling

ECP xSDK FASTMath R&D 100

https://www.mcs.anl.gov/petsc

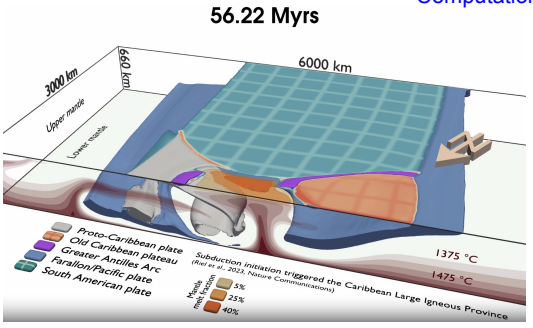
ECP EXASCALE COMPUTING PROJECT

PETSc - the Portable, Extensible Toolkit for Scientific computation

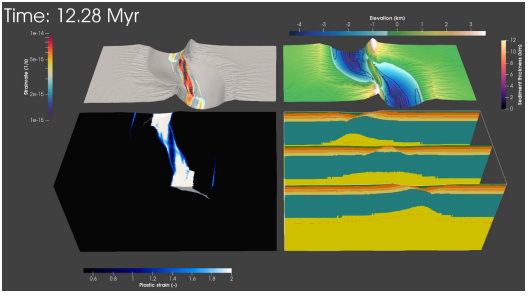


Some Geophysical Applications Enabled by PETSc

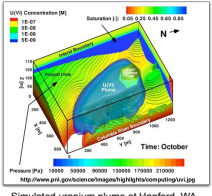
Computational Geodynamics



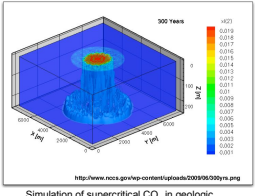
Subduction zone formation simulated with LaMEM (Riel et al. 2023, https://doi.org/10.1038/s41467-023-36419-x)



Continental rift evolution simulated with pTatin3D (Wolf et al., 2022, https://doi.org/10.1029/2022JB024687)



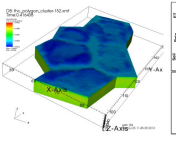
Simulated uranium plume at Hanford, WA

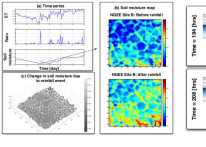


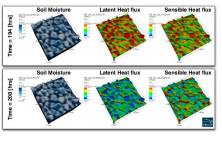
Simulation of supercritical CO_2 in geologic carbon sequestration

Computational Hydrology and Subsurface Science

PFLOTRAN







Simulations of permafrost hydrology near Barrow, AK to assess climate impacts of high-latitude warming

PETSc Design Philosophy

Optimal solvers must consider the interplay among physics, algorithms, and architectures.

Algorithmic composability

- Composable: Solvers should be easy to combine, by non-experts, to form a more powerful solver.
- Hierarchical: Outer solvers may iterate over all variables for a global problem, while inner solvers handle smaller subsets of physics, smaller physical subdomains, or coarser meshes.
- Nested: Outer solvers call nested inner solvers
- Extensible: Easily customized or extended by users

Runtime configurability

- Performance models for the most interesting problems are generally not predictive enough to enable *a priori* determination of best solver
- PETSc components should be combinable in flexible ways at runtime to **enable experimentation**

Vertical integration

- Holistic application tuning must recognize that economies of selected components are coupled
- Enable re-use of information between components: Data layout information, user-provided callbacks (e.g., between time integrators and sensitivity analysis), previously computed operators, etc.

Scaling PETSc to Extreme-Scale Systems

Parallel scalability has been a key design goal of PETSc since its inception.

Inter-node scalability

- For much of PETSc's history (first release in 1995), the primary challenge to achieving extreme-scale performance was getting good inter-node (MPI) scaling.
- As node counts increased and new barriers to inter-node scalability were encountered, many features have been introduced in response, to either
  - Decrease or hide communication, at cost of increased local work
  - Shift where collective communication occurs (trade expensive global communication for communication across smaller neighborhoods)

Decreasing/Hiding Communication via Increased Local Work

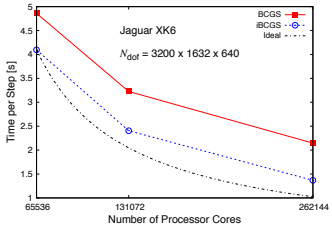


Figure: Comparison of BCGS and IBCGS on OLCF Jaguar Cray XK6 system. IBCGS is a reformulation of BCGS that requires only one global reduction, at cost of increased work

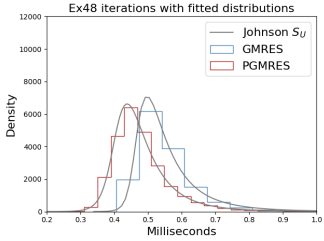
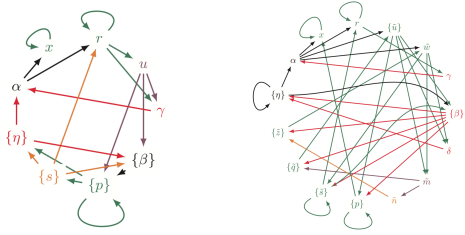
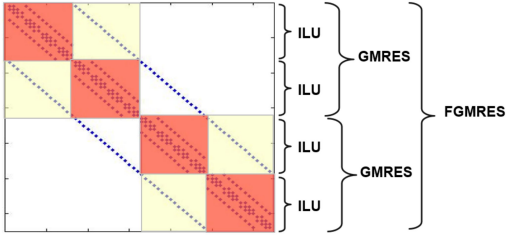


Figure: Bulk statistics for iteration times for standard and pipelined versions of GMRES on ALCF Theta Cray XC40 machine. https://doi.org/10.1177/1094342020966835



At Left: Schematics of the main loop of the Flexible Conjugate Gradient (FCG; left) and the Modified Pipelined Flexible Conjugate Gradient (PIPEFCG; right) methods. Data dependencies in FCG preclude overlapping reductions with the application of the operator or the preconditioner. At the price of increased local work and storage, PIPEFCG reductions can overlap operator and preconditioner application. https://doi.org/10.1137/15M1049130

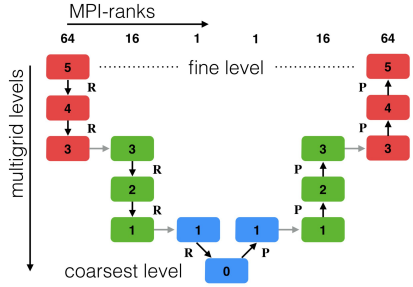
Shifting Where Collective Communication Occurs



Num. of Cores (*np*) (mesh size)	Groups of Cores (*ngp*)	Num. of Cores per Group	Timestep Cuts	% Time for Inner Products	Outer Iterations	Execution Time (sec)
512	1	512	0	28	3,853	43.9
(256x256x256)	16	32	0	17	903	45.8
4,096	1	4,096	0	39	11,810	146.5
(512x512x512)	64	64	0	23	2,405	126.7
32,768	1	32,768	1	48	35,177	640.5
(1024x1024x1024)	128	256	0	28	5,244	297.4
98,304	1	98,304	7	77	59,250	1346.0
(1024x1024x1024)	128	768	0	47	6,965	166.2
160,000	1	160,000	9	72	59,988	1384.1
(1600x1600x640)	128	1,250	0	51	8,810	232.2

Above: Hierarchical Krylov methods trade global inner products for ones over a neighborhood; generalizing additive Schwarz methods, each block is solved by Krylov methods on smaller blocks. doi:10.1016/j.parco.2013.10.001

Right: Agglomeration in parallel multigrid. Rather than incur large communication cost on coarse grid, communicate at intermediate points in the hierarchy. As we coarsen, use smaller sets of processors (MPI ranks), which allows balancing communication and computation. Implemented in `PCTELESCOPE` in PETSc.
https://doi.org/10.1145/2929908.2929913



Scaling PETSc to Extreme-Scale Systems

Parallel scalability has been a key design goal of PETSc since its inception.

Inter-node scalability

- For much of PETSc's history (first release in 1995), the primary challenge to achieving extreme-scale performance was getting good inter-node (MPI) scaling.
- As node counts increased and new barriers to inter-node scalability were encountered, many features have been introduced in response, to either
  - Decrease or hide communication, at cost of increased local work
  - Shift where collective communication occurs (trade expensive global communication for communication across smaller neighborhoods)

Intra-node scalability

- After steady increase, supercomputer node counts have plateaued and then slightly decreased (Fugaku is the exception).
- New challenge is to **leverage high levels of fine-grained, on-node parallelism**.
- Remainder of talk will focus on specific challenges posed for scientific libraries by GPU-based supercomputer architectures, and approaches adopted in PETSc to address them.

Fundamental GPU Challenges (F1–F2)

Three *fundamental* challenges arise in providing libraries that obtain *high throughput* performance on parallel GPU accelerated systems due to hardware and low-level software aspects of GPUs.
No programming model obviates or allows programmers to ignore them.

F1. Portability for application codes

- Many competing, incompatible GPU architectures and programming models
- Different vendors support different programming models and provide different mathematical libraries with different APIs and even different synchronization models

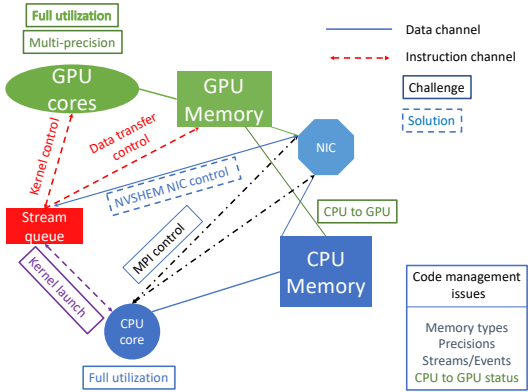
F2. Algorithms for high-throughput systems

- In addition to application porting, high-throughput systems such as GPUs require developing and implementing new solver algorithms—ones that exploit high levels of data-parallel concurrency with low levels of memory bandwidth delivered in a data-parallel fashion.
- Generally involves algorithms with higher arithmetic intensity than most traditional simulations utilize.

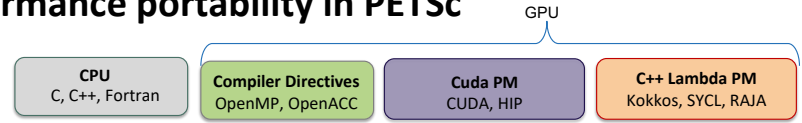
Fundamental GPU Challenges (F3)

F3. Utilizing all GPU and CPU compute power

- Achieving high computational throughput requires keeping all GPU compute units as busy as possible
- Each core must have an uninterrupted stream of instructions and a high-bandwidth stream of data within the constraints of the hardware and low-level software stack
- Difficulty arises from complex control and data flows; distributed memory further complicates picture



Performance portability in PETSc



Application code

Using PETSc API

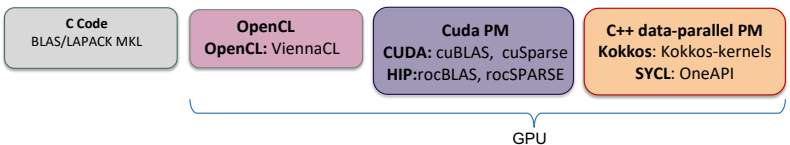
Front-end PETSc vector and matrix arrays are shared with user programming language/model

PETSc computation kernels

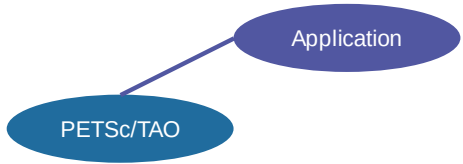
CPU: Use compiler options and vendor libraries for performance

GPU: Chosen for either speed of development or highest performance. Use GPU vendor libraries

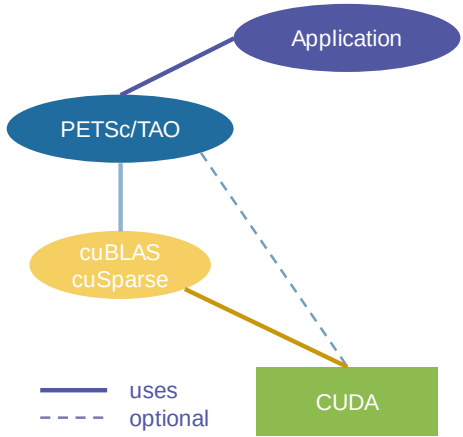
Back-end PETSc Vector and Matrix implementations



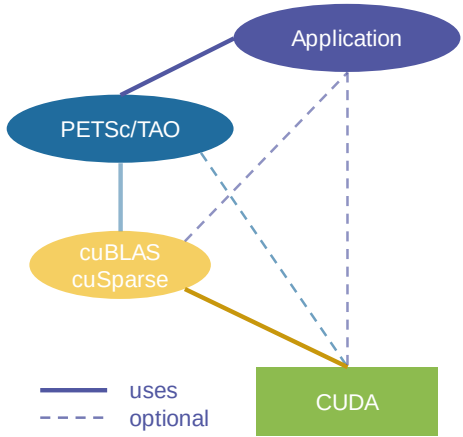
Example: PETSc and Application Use CPU only



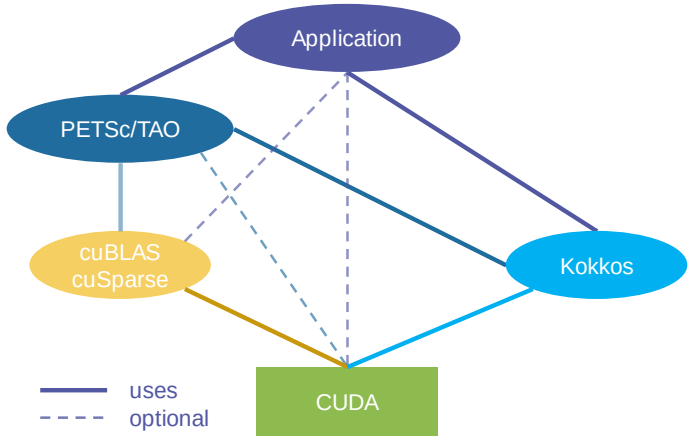
Example: PETSc Using CUDA/cuBLAS/cuSPARSE



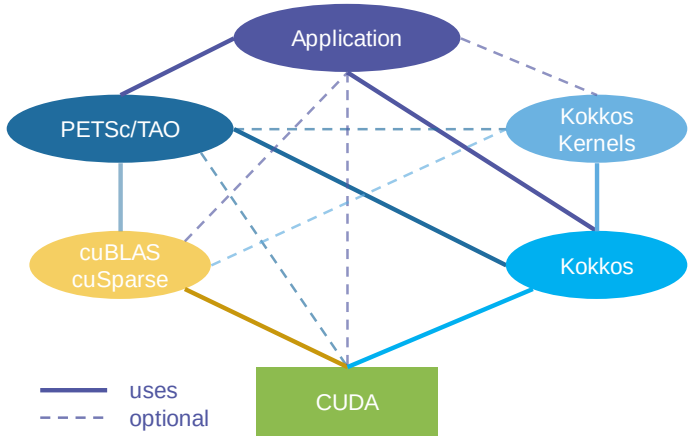
Example: PETSc and Application Using CUDA



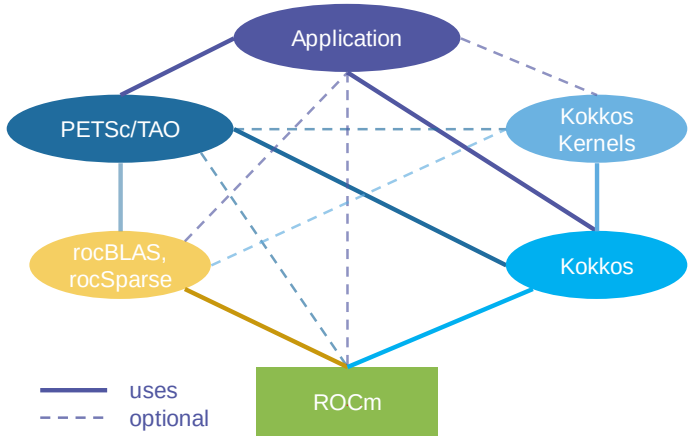
Example: PETSc and Application Using CUDA and Kokkos



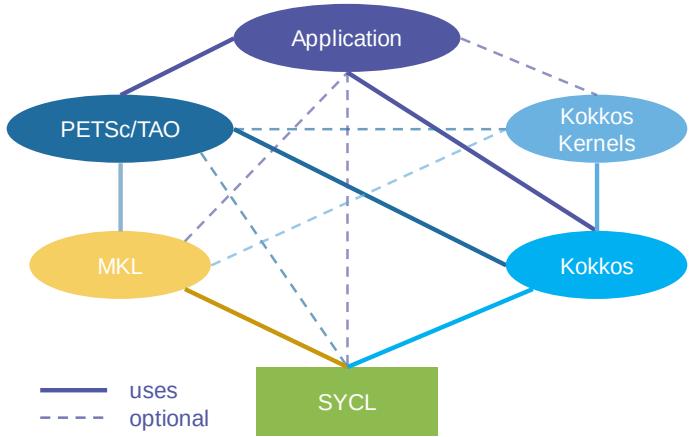
Example: PETSc and Application Using CUDA, Kokkos, and Kokkos-Kernels



Example: PETSc and Application Using AMD GPUs via ROCm, Kokkos, and Kokkos-Kernels



Example: PETSc and Application Using Intel GPUs via SYCL, Kokkos, and Kokkos-Kernels



How PETSc Uses GPUs

- Provides several new implementations of PETSc's `Vec` (distributed vector) and `Mat` (distributed matrix) classes which allow data storage and manipulation in device (GPU) memory
- Embue all `Vec` (and `Mat`) objects with the ability to track the state of a second "offloaded" copy of the data, and synchronize these two copies of the data (only) when required ("lazy-mirror" model).
- Because higher-level PETSc objects rely on `Vec` and `Mat` operations, execution occurs on GPU when appropriate delegated types for `Vec` and `Mat` are chosen.

Host and Device Data

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

Possible Flag States

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

Using GPU Back-Ends in PETSc

Transparently use GPUs for common matrix and vector operations, via runtime options.
Currently CUDA/cuSPARSE, HIP/hipSPARSE, Kokkos, and ViennaCL are supported.

CUDA/cuSPARSE usage:

- 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`

Because PETSc separates high-level control logic from optimized computational kernels, even very complicated hierarchical/multi-level/domain-decomposed/physics-based solvers can run on different architectures by simply choosing the appropriate back-end at runtime; **re-coding is not needed**.

The Common Pattern for PETSc Application Codes

PETSc application codes, regardless of whether they use time integrators, nonlinear solvers, or linear solvers, follow a common pattern:

- Compute application-specific data structures,
- Provide a `Function` computation callback,
- Provide a `Jacobian` computation callback, and
- Call the PETSc solver, possibly in a loop.

This approach does not change with the use of GPUs.
In particular, the creation of solver, matrix, and vector objects and their manipulation do not change.

Points to consider when porting an application to GPUs:

- Some data structures reside in GPU memory, either
  - constructed on the CPU and copied to the GPU or
  - constructed directly on the GPU.
- `Function` will call GPU kernels.
- `Jacobian` will call GPU kernels.

Main Application Code for CPU or GPU

Consider excerpt of a typical PETSc main application program for solving a nonlinear set of equations on a structured grid using Newton's method. It creates a solver object `SNES`, a data management object `DM`, a vector of degrees of freedom `Vec`, and a `Mat` to hold the Jacobian. Then, Function and Jacobian evaluation callbacks are passed to the `SNES` object to solve the nonlinear equations.

```
SNESCreate(PETSC_COMM_WORLD,&snes);
DMDACreate1d(PETSC_COMM_WORLD,...,&ctx.da);
DMCreateGlobalVector(ctx.da,&x);
VecDuplicate(x,&r);
DMCreateMatrix(ctx.da,&J);
if (useKokkos) {
 SNESSetFunction(snes,r,KokkosFunction,&ctx);
 SNESSetJacobian(snes,J,J,KokkosJacobian,&ctx);
} else {
 SNESSetFunction(snes,r,Function,&ctx);
 SNESSetJacobian(snes,J,J,Jacobian,&ctx);
}
SNESSolve(snes,NULL,x);
```

Listing 1: Main application code for CPU or GPU

Traditional PETSc Function and Kokkos version

```
DMGetLocalVector(da,&xl);
DMGlobalToLocal(da,x,INSERT_VALUES,xl);
DMDAVecGetArrayRead(da,xl,&X); // only read X[]
DMDAVecGetArrayWrite(da,r,&R); // only write R[]
DMDAVecGetArrayRead(da,f,&F);  // only read F[]
DMDAGetCorners(da,&xs,NULL,NULL,&xm,...);
for (i=xs; i<xs+xm; ++i)
  R[i] = d*(X[i-1]-2*X[i]+X[i+1])+X[i]*X[i]-F[i];
-----------------------------------------------------------
DMGetLocalVector(da,&xl);
DMGlobalToLocal(da,x,INSERT_VALUES,xl);
DMDAVecGetKokkosOffsetView(da,xl,&X); // no copy
DMDAVecGetKokkosOffsetView(da,r,&R,overwrite);
DMDAVecGetKokkosOffsetView(da,f,&F);
xs = R.begin(0); xm = R.end(0);
Kokkos::parallel_for(
  Kokkos::RangePolicy<>(xs,xm),KOKKOS_LAMBDA
  (int i) {
    R(i) = d*(X(i-1)-2*X(i)+X(i+1))+X(i)*X(i)-F(i);});
```

Listing 2: Traditional PETSc Function (top) and Kokkos version (bottom). `xl`, `x`, `r`, `f` are PETSc vectors. `X`, `R`, `F` at the top are `double*` or `const double*` like pointers but at the bottom are Kokkos unmanaged `OffsetViews`.

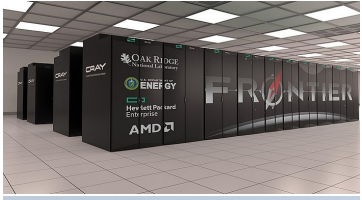
Traditional PETSc Jacobian and Kokkos version

```
DMDAVecGetArrayRead(da,x,&X);
DMDAGetCorners(da,&xs,NULL,NULL,&xm,...);
for (i=xs; i<xs+xm; i++) {
  j = {i - 1,i,i + 1}; A = {d, -2*d + 2*X[i],d};
  MatSetValues(J,1,&i,3,j,A,INSERT_VALUES);
}
-------------------------------------------------------------
DMDAVecGetKokkosOffsetView(da,x,&X);
MatKokkosGetDeviceMatWrite(J,&d_mat); // device handle that allows writing to matrix entries
xs = X.begin(0); xm = X.end(0);
Kokkos::parallel_for(
  Kokkos::RangePolicy<>(xs,xm),KOKKOS_LAMBDA
  (int i){
    j = {i-1,i,i+1}; A = {d, -2*d + 2*X(i),d};
    MatSetValuesDevice(d_mat,1,&i,3,j,A,INSERT_VALUES);});
```

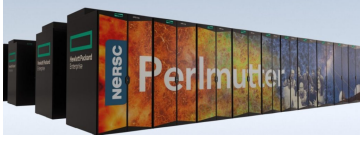
Listing 3: Traditional PETSc Jacobian (top) and Kokkos version (bottom)

Numerical Experiments: Platforms

Summit (DOE ORNL/OLCF): Current #4 on TOP500; IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband



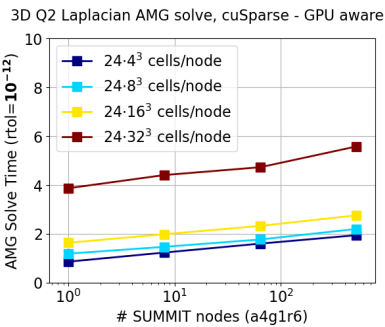
Frontier (DOE ORNL/OLCF): Current #1 on TOP500; HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11 (some results from Crusher testbed system with identical hardware)

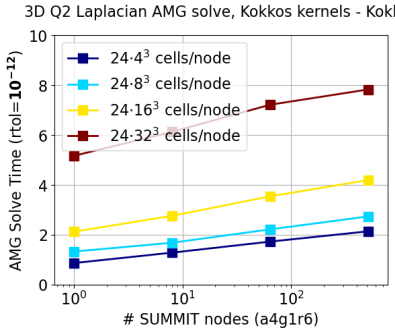


Perlmutter (DOE LBNL/NERSC): Current #7 on TOP500; HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10

(Some reported results also come from Lassen, an IBM Power System AC922 at LLNL that has the same CPUs and GPUs as Summit, but only 4 GPUs per node instead of 6.)

Summit Results: PETSc GAMG Algebraic Multigrid

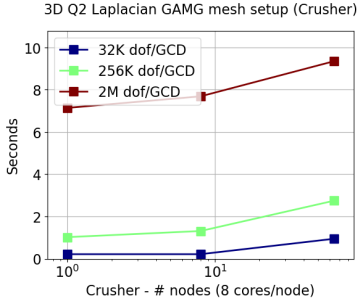


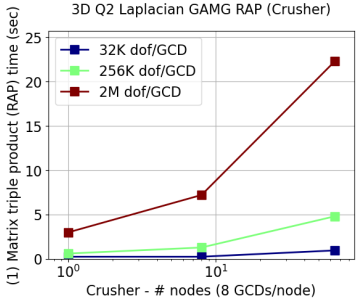


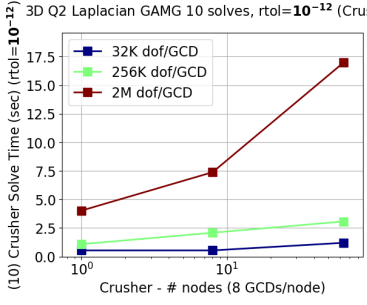
Solve time for a 3D Laplacian with second-order elements. Larger grids are generated by uniform refinement. Runs are configured with six resource sets on each Summit node, each with one GPU and 4 MPI processes.

- MPI parallel scaling is good.
- Slower performance of KokkosKernels is due to computing a transpose for matrix transpose multiply, which is not natively supported.
- Kokkos is faster when configured with cuSPARSE kernels.

Crusher (Frontier Prototype) Results: PETSc GAMG Algebraic Multigrid

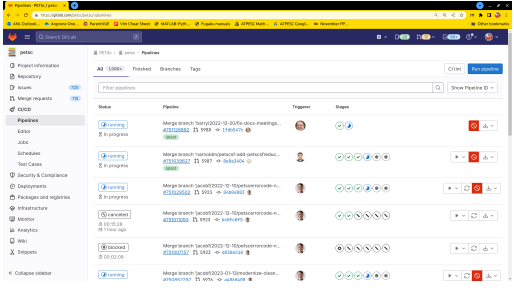


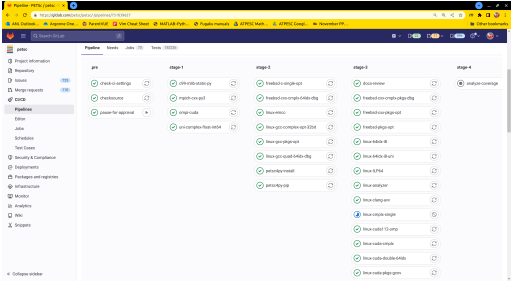




Challenge: Everything Seems Broken, All the Time

- We support four GPU back-ends (CUDA, HIP, Kokkos, ViennaCL) and may add more: there are a lot of "moving parts" and it is easy for things to break.
- GPU support in MPI implementations has been particularly problematic:
  "There's an old adage that when the impossible happens, it's probably a bug in your code, not a bug in the compiler. [Here], it's probably a bug in the vendor's GPU-aware MPI that they've known about for a year and haven't deemed worth telling you about." —Jed Brown
- Substantial investment (both money and time) in continuous integration (CI) infrastructure for GPUs has been critical!
  - Early in ECP, PETSc moved to Gitlab because it enabled much more robust CI. Investing in improved CI is one of the **most important things** we have done during ECP.





Challenge: Measuring, Understanding, and Reasoning About Performance

- Test/measure things yourself: conventional wisdom is often wrong.
- Performance measurement can be challenging on GPU nodes:
  - High GPU initialization cost can skew measurements (we support both eager and lazy initialization; former facilitates performance measurement, latter allows execution of CPU-configured binaries on non-GPU nodes).
  - (GPU initialization cost is also a substantial drag on CI infrastructure.)
  - Detailed logging of GPU performance introduces synchronization overhead; by default only highest-level events are timed (add `-log_view_gpu_time` for detailed profiling).
- Just because a vendor implementation exists does not mean you should use it: E.g., sparse triangular solves on GPUs can be 20X slower than matrix vector multiplication (on CPUs cost is roughly equal).
- It is important to establish latency and bandwidth baselines so you can reason about performance.

VecAXPY() and VecCopy() Operations on Summit: CPU vs. GPU

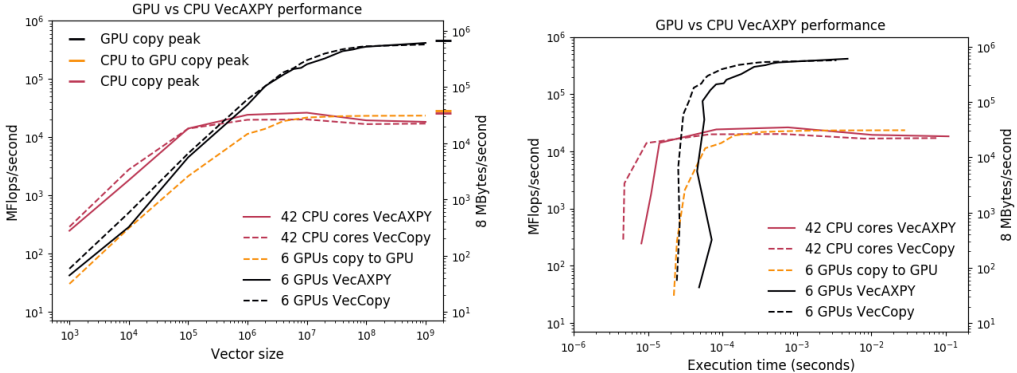
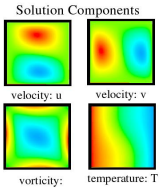


Figure: Effect of vector size on AXPY performance and memory throughput (one MPI rank per GPU) on a single node of the OLCF Summit system. As vector sizes become large, GPUs perform significantly better than the 42 Power9 CPU cores available, but the CPU cores can be significantly faster for smaller 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.

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.



$$
\begin{aligned}
-\Delta U - \partial_y \Omega &= 0 \\
-\Delta V + \partial_x \Omega &= 0 \\
-\Delta \Omega + \nabla \cdot ([U\Omega, V\Omega]) - \mathrm{Gr}\ \partial_x T &= 0 \\
-\Delta T + \mathrm{Pr}\ \nabla \cdot ([UT, VT]) &= 0
\end{aligned}
$$

CPU only command line (V-cycles case):

```
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)

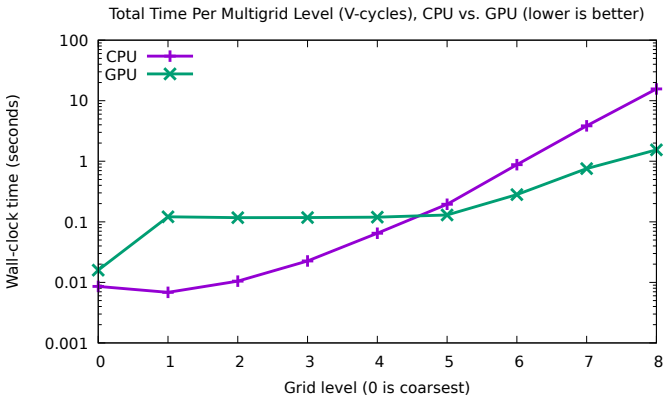
CPU + GPU hybrid command line (V-cycles case):

```
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)

SNES ex19 (Driven Cavity) on Summit with Multigrid, Times per Level

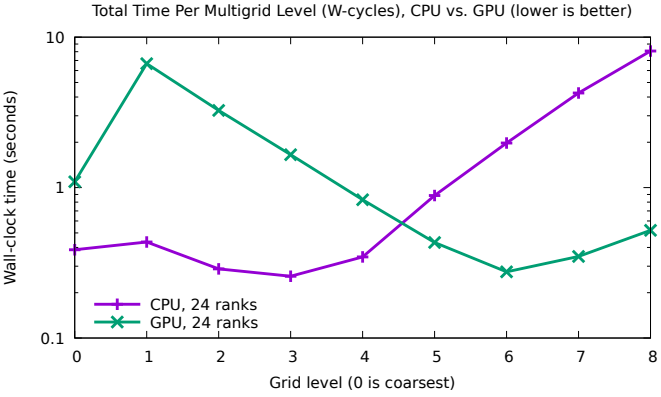
Examine time per multigrid level (use `-pc_mg_log`) for 24 ranks on CPU vs. GPU:



- Naively placing all levels on GPU results in latency dominating solve on coarse levels
- Flat line for levels 1-5 in V-cycle shows GPU latency limit

SNES ex19 (Driven Cavity) on Summit with Multigrid, Times per Level

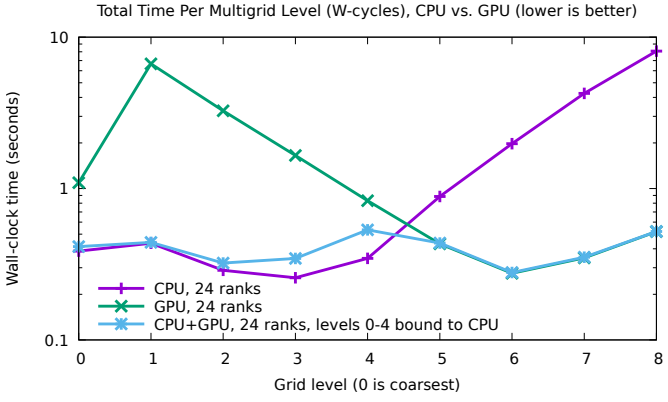
Examine time per multigrid level (use `-pc_mg_log`) for 24 ranks on CPU vs. GPU:



- Naively placing all levels on GPU results in latency dominating solve on coarse levels
- Flat line for levels 1-5 in V-cycle shows GPU latency limit
- Coarse levels are especially costly on GPU inside W-cycle

SNES ex19 (Driven Cavity) on Summit with Multigrid, Times per Level

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



- Naively placing all levels on GPU results in latency dominating solve on coarse levels
- Flat line for levels 1-5 in V-cycle shows GPU latency limit
- Coarse levels are especially costly on GPU inside W-cycle
- Implemented mechanism (see `-dm_bind_below`) to selectively bind coarse levels to CPU while using GPU for fine levels. For W-cycle case:
  - 1.15X speedup vs. CPU without binding
  - 4.3X speedup vs. CPU with binding

Challenge: (Re-)Think About Data Structures Early and Often

Consider example of long-established data structures and interfaces PETSc uses for matrix assembly.

Traditional PETSc matrix assembly paradigm—loop over elements and call `MatSetValues()`
—is problematic on GPUs:

- Assembly into compressed sparse row (CSR) matrices in PETSc involves preallocating and adding values to logically dense blocks using `MatSetValues()`, typically one block per finite element.
- Keeps memory utilization low, but is inefficient on GPUs and requires binary search to find insertion location in CSR matrix.
- Enabling full `MatSetValues()` functionality on GPU requires locks or atomics when updating local matrix stash.

As an alternative, we have introduced `MatSetValuesCOO()`:

- Uses a coordinate (COO) format for matrix entry (assembled matrix still stored in CSR format).
- Two phases: Symbolic (preallocation) followed by setting numerical values
- Requires some additional memory, but handles off-process entries without locks or atomics and delivers high performance on GPUs
- Supports identical API for both CPU and GPU

COO for assembly is the new paradigm that should be (generally) adopted in the GPU-computing era.

On-Device Matrix Assembly: Matrix Creation with COO API

```
static PetscErrorCode CreateMatrix(FEStruct *fe,Mat *A)
{
  PetscInt       *oor,*ooc,cnt = 0;

  PetscFunctionBeginUser;
  PetscCall(MatCreate(PETSC_COMM_WORLD, A));
  PetscCall(MatSetSizes(*A, fe->n, fe->n, PETSC_DECIDE, PETSC_DECIDE));
  PetscCall(MatSetFromOptions(*A));

  /* Determine for each entry in each element stiffness matrix the global row and column */
  /* The element is triangular with piecewise linear basis functions,
     so there are three degrees of freedom per element, one for each vertex */
  PetscCall(PetscMalloc2(3 * 3 * fe->Ne, &oor, 3 * 3 * fe->Ne, &ooc));
  for (PetscInt e=0; e<fe->Ne; e++) {
    for (PetscInt vi=0; vi<3; vi++) {
      for (PetscInt vj=0; vj<3; vj++) {
        oor[cnt]   = fe->vertices[3*e+vi];
        ooc[cnt++] = fe->vertices[3*e+vj];
      }
    }
  }
  PetscCall(MatSetPreallocationCOO(*A, 3 * 3 * fe->Ne, oor, ooc));
  ...
}
```

Listing 4: Matrix creation and preallocation for CPU or GPU case. See `$PETSC_DIR/src/mat/tutorials/ex18.c`

(And, for a more sophisticated example that illustrates how PETSc facilitates COO assembly for stencil operators, see `snes/tutorials/ex55`.)

On-Device Matrix Assembly: MatSetValuesCOO() API

```
/* simulation of CPU based finite assembly process with COO */
PetscScalar    *v,*s;
PetscCall(PetscMalloc1(3 * 3 * fe->Ne, &v));
for (PetscInt e=0; e<fe->Ne; e++) {
  s = v + fe->coo[e]; /* point to location in COO of current element stiffness */
  for (PetscInt vi=0; vi<3; vi++) {
    for (PetscInt vj=0; vj<3; vj++) {
      s[3*vi+vj] = vi+2*vj;
    }
  }
}
PetscCall(MatSetValuesCOO(A, v, ADD_VALUES));
```

Listing 5: Matrix assembly on CPU

```
// Simulation of GPU based finite assembly process with COO
Kokkos::View<PetscScalar*,DefaultMemorySpace> v("v",3*3*fe->Ne);
Kokkos::parallel_for(
  "AssembleElementMatrices", fe->Ne, KOKKOS_LAMBDA(PetscInt i) {
    PetscScalar *s = &v(3 * 3 * i);
    for (PetscInt vi = 0; vi < 3; vi++) {
      for (PetscInt vj = 0; vj < 3; vj++) s[vi * 3 + vj] = vi + 2 * vj;
    }
  });

PetscCall(MatSetValuesCOO(A, v.data(), ADD_VALUES));
```

Listing 6: Matrix assembly on GPU using Kokkos

OpenFOAM Miniapp

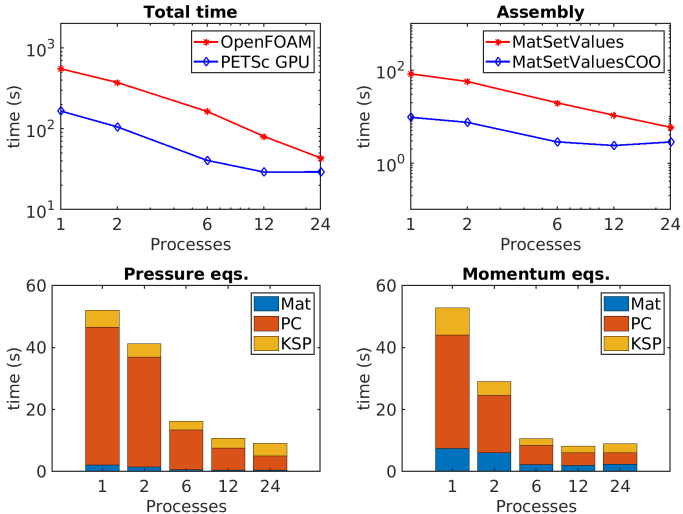


Figure: OpenFOAM miniapp timings on Summit. Upper-left panel: total time using native OpenFOAM or PETSc GPU solvers. Upper-right panel: assembly times using `MatSetValues` or `MatSetValuesCOO`. Bottom row: breakdown of PETSc solver timings for pressure (left) and momentum equations (right).

Challenge: Mismatch Between MPI and GPU Stream Programming Models

- Current MPI often prevents asynchronous GPU computation: GPU-aware MPI instructions pass directly to memory controls and do not enter the stream queue; thus, the stream queue must be empty for every MPI call, forcing synchronizations and preventing pipelined kernel launches.
- MPI must add GPU execution stream support, or we need to explore other models for distributed-memory parallelism (PetscSF communication abstraction now supports NVSHMEM).

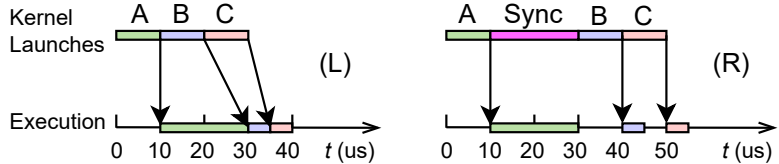


Figure: Pipelined kernel launches (L) vs. interrupted kernel launches (R). Suppose a kernel launch takes 10 µs and A, B, C are three kernels taking 20, 5, 5 µs to run respectively. (L) shows a timeline with fully-pipelined kernel launches. (R) shows a timeline with a device synchronization after kernel A. MPI communication forces synchronizations like in (R); NVSHMEM does not and allows a timeline like in (L).

Support for GPU-Aware and GPU-Centric Communication in PetscSF

PetscSF is a communication layer in PETSc that has been incrementally replacing direct use of MPI.

- It represents communications as operations over "star forests".
- A star is a simple tree consisting of a root vertex connected to zero or more leaves; a star forest is a disjoint union of stars.
- PetscSF analyzes a specified communication graph and builds communication pattern using persistent nonblocking send/receive (default), one-side operations, neighborhood collectives, or other mechanisms.

PetscSF now supports GPU-aware MPI:

- GPU buffers can be passed directly to MPI; no need to stage GPU buffers through host memory
- Used by default since PETSc 3.13; disable with `-use_gpu_aware_mpi 0`

Support for GPU-Aware and GPU-Centric Communication in PetscSF

PetscSF is not limited to using MPI as the underlying communication layer!

- We have recently added support for NVIDIA's NVSHMEM partitioned global address space runtime.
- NVSHMEM allows pipelining of communication calls and kernel launches
- Current MPI does not: GPU-aware MPI instructions pass directly to memory controls and do not enter the stream queue; thus, the stream queue must be empty for every MPI call.
- We are exploring Krylov solver implementations that can take full advantage of asynchronous GPU execution.

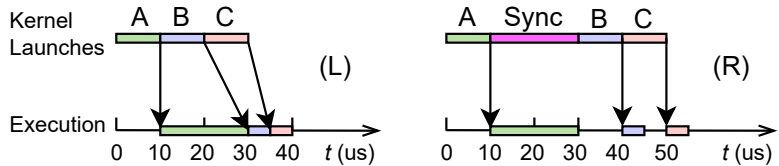


Figure: Pipelined kernel launches (L) vs. interrupted kernel launches (R). Suppose a kernel launch takes 10 μs and A, B, C are three kernels taking 20, 5, 5 μs to run respectively. (L) shows a timeline with fully-pipelined kernel launches. (R) shows a timeline with a device synchronization after kernel A. MPI communication forces synchronizations like in (R); NVSHMEM does not and allows a timeline like in (L).

Asynchronous Conjugate Gradient Solver using PetscSF + NVSHMEM

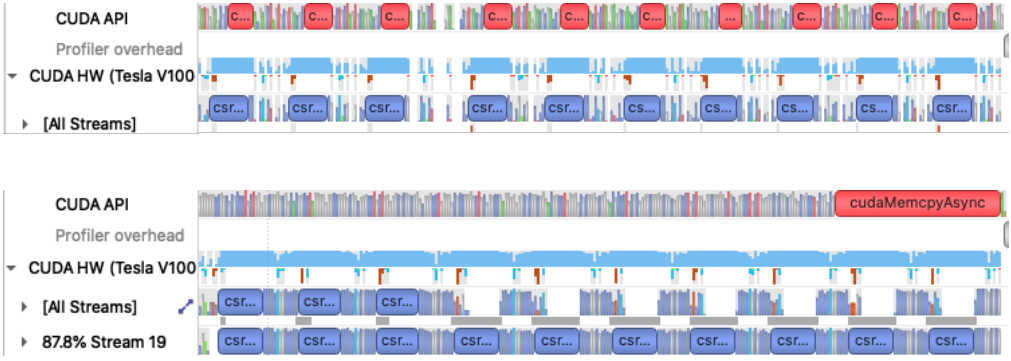


Figure: Timeline (by Nsight System) of CG with PetscSF + CUDA-aware MPI (top) and CGAsync with PetscSF + NVSHMEM (bottom) on rank 2 of a test run with 6 MPI ranks (GPUs) on a Summit compute node. Each ran ten iterations. Blue *csr...* bars are csrMV (i.e., SpMV) kernels in cuSPARSE, and the red *c...* bars are cudaMemcpyAsync() copying data from device to host. As we can see at the top, with MPI, the kernel launches (labeled with 'CUDA API') were frequently stalled and spread with kernel executions, while with NVSHMEM at the bottom, kernel launches were fully pipelined and during the 8th iteration, the host had launched all kernels of the ten iterations and began waiting for the final result on GPU.

Challenge: Managing Asynchronicity and Many Concurrent GPU Streams Is Hard

New PETSc team member Jacob Faibussowitsch is re-architecting much of how PETSc manages GPU devices and streams with `PetscDeviceContext` and `Petsc::ManagedMemory`.

`PetscDeviceContext`

- Backend agnostic wrapper over device and stream objects
- Ensures device libraries all play nice together
- Provides additional safety and ergonomics for vendor objects:
  - Simple things like "stream A wait for stream B" are *simple*
  - Improved compatibility, e.g. async allocators available for *any* CUDA/HIP version
  - Automatically serializes dependent stream operations via auto-dependency tracking system

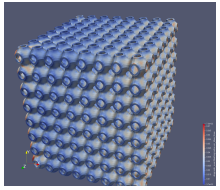
Petsc::ManagedMemory – A Sneak Peek of Work In Progress

https://gitlab.com/petsc/petsc/-/merge_requests/6178

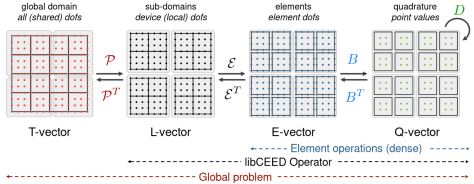
```
using namespace Petsc;
// initialization not shown
Vec                 x, y, z;
ManagedScalar       alpha, beta;
PetscDeviceContext dctx_a, dctx_b;

// Do VecAXPY() "on" dctx_a, may not complete before function returns
VecAXPYAsync(y, alpha, x, dctx_a);
// Perform this scalar calculation asynchronously on device. Support for
// arbitrary expressions
alpha = eval(dctx_a, beta + 1);
// NOTE dctx_b! Thanks to auto-dep system, this kernel knows to wait for
// scalar op to complete before starting
VecAXPBYAsync(y, alpha, beta, x, dctx_b);
// Executes concurrently with VecAXPBYAsync() (no dependency conflict)
VecScaleAsync(z, alpha, dctx_a);
```

Ratel: Efficient End-to-End GPU Simulation of Solid Mechanics

End-to-end GPU solution of non-linear solid mechanics models using Newton-Krylov with *p*-multigrid, via PETSc + libCEED + BoomerAMG.
`https://arxiv.org/abs/2204.01722`



$$A = \mathcal{P}^T \mathcal{E}^T B^T D B \mathcal{E} \mathcal{P}$$



Ratel challenges industrial state of practice and widely-held myths:

- Low order finite elements: Q1 (trilinear) hexahedra, P2 (quadratic) tetrahedra
- Assembled matrices, sparse direct and algebraic multigrid solvers
- Myth: High-order doesn't help because real problems have singularities
- Myth: Matrix-free methods are only for high-order problems

Ratel demonstrates the importance of **re-thinking data structures**: by using matrix-free *p*-multigrid and AMG coarse solvers, high-order methods become cheaper per DoF than low-order methods (assembled or not), enabling faster and cheaper simulations at engineering tolerances.

Achieves efficient end-to-end utilization of GPUs while containing no architecture-specific code: all such code exists within the libCEED back-ends and PETSc and hypre numerical kernels.

High-Order Methods Have Good Approximation Constants

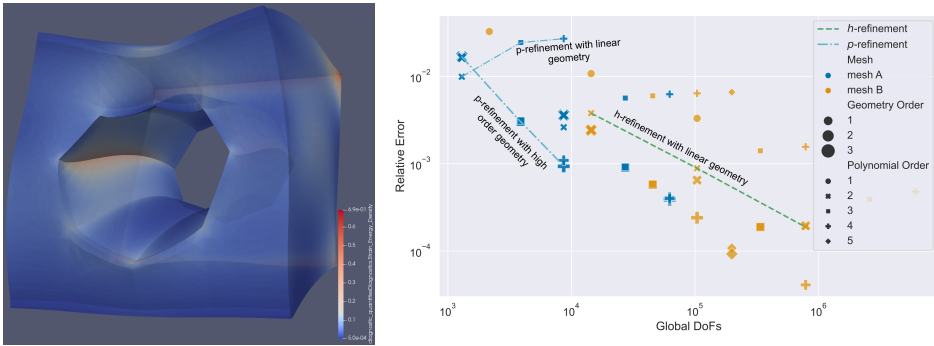
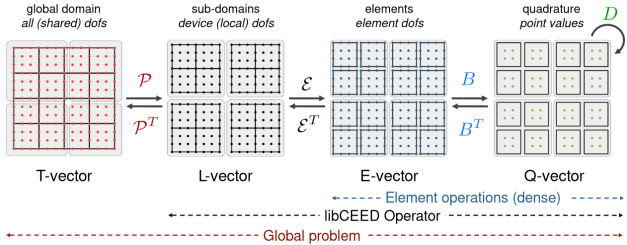


Figure: Accuracy study (right) showing relative error in total strain energy Ψ versus DoFs for a bending experiment (left) under both *h*-refinement (same shape) and *p*-refinement (same color) with low- and high-order geometry. The Pareto front is toward the lower left and we observe that *h*-refinement always moves away from optimality. The slope of *h*-refinement is the same for all meshes and solution orders. *p*-refinement is very efficient so long as the geometry is at least quadratic, but causes errors to increase when *p*-refining on linear geometry due to resolution of the non-physical singularities. **Commercial software (e.g., Abaqus and ANSYS) rules of thumb correspond to the gold circles (linear elements).**

Clear Separation of Concerns in Ratel between PETSc and libCEED

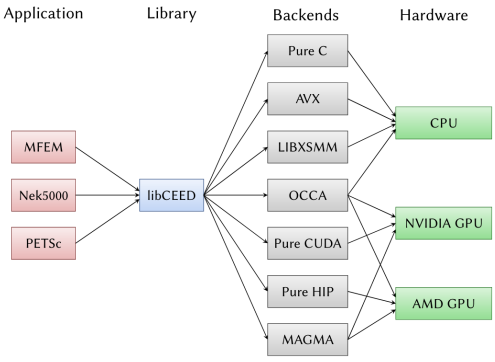
$$A = \mathcal{P}^T \mathcal{E}^T B^T D B \mathcal{E} \mathcal{P}$$



- libCEED handles operator application, vector functions, interpolations, ... (all the analytical things)
- PETSc handles mesh management/refinement/geometry (topological things) and also solvers/preconditioners (analysis again)
- PETSc represents both the mesh and the function space, allowing it to construct the "restriction" to integration domains (cells). This hand-off to libCEED allows Ratel to automate complex things.
- The way that PETSc and libCEED are used allows users flexibility in choice of GPU abstraction: GPU libraries are only linked transitively through PETSc and libCEED.

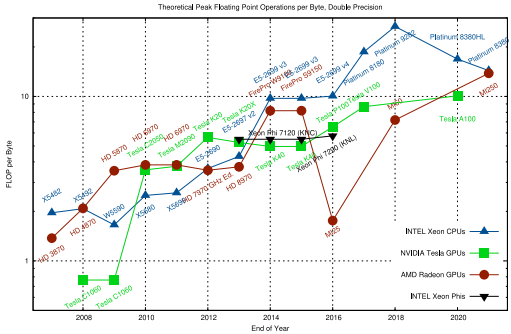
libCEED: Fast Algebra for High-Order Element-Based Discretizations

- Backend plugins with run-time selection
  - debug/memcheck, optimized
  - libxsmm, CUDA, HIP
  - MAGMA to CUDA and HIP
  - OCCA to OpenMP, OpenCL, CUDA, and HIP
  - CPU back-ends implement element action B using tensor-contractions with architecture-specific vectorization
  - GPU back-ends create fused kernel (compiled at runtime) containing entire local action of finite element operator
- Single source vanilla C for QFunctions
  - Easy to debug, understand locally, C++ optional
  - Target for DSLs, automatic differentiation
- Python, Julia, Rust bindings
- 2-clause BSD
- Available via PETSc, MFEM, Nek5000

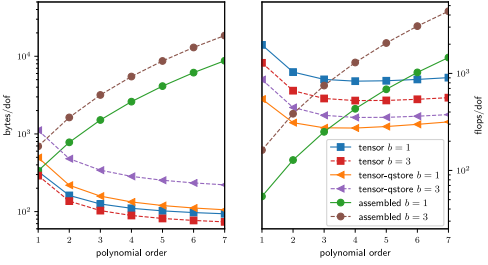


Why Matrix-Free?

Bandwidth is scarce compared to flops!



- Assembled matrices need at least 4 bytes transferred per flop. Hardware does 10 flops/byte. Matrix-free methods store and move less data, compute faster.
- High-order methods have better accuracy constants (thus favoring p-refinement over h-refinement), but are rarely used in practice, because assembly and linear algebra are much more expensive (no improvement in asymptotics).



Operator Application Efficiency: Matrix-free faster even for Q1 elements

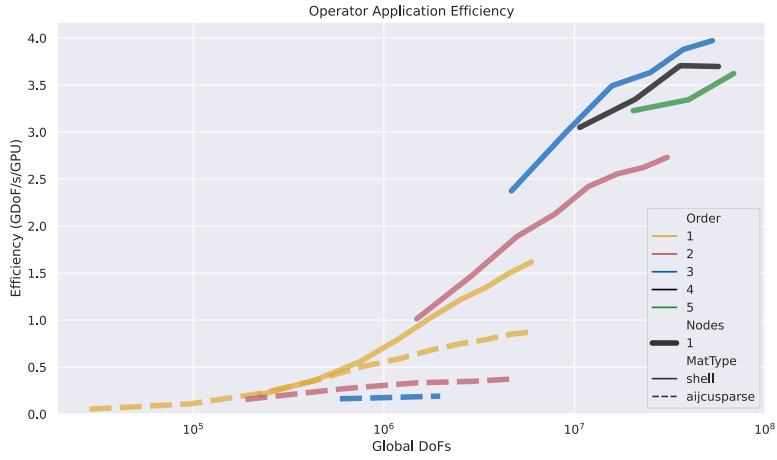
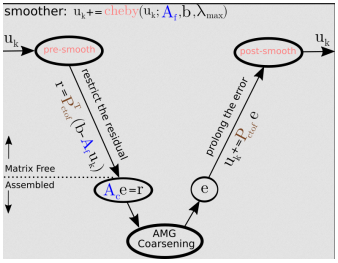


Figure: Parallel operator application efficiency (timings for matrix multiplication only) running on LLNL's Lassen with assembled `aijcusparse` and matrix-free `shell` operator representations. `shell` becomes more efficient as the order increases while `aijcusparse` becomes less so. Both are latency-limited for smaller problem sizes (left side of the figure) and plateau as memory is filled for larger sizes. `aijcusparse` cases run out of memory for smaller numbers of DoFs because high-order methods yield many nonzeros per row.

Matrix-Free p-Multigrid

- In p-multigrid, discretization is coarsened by reducing polynomial order of basis functions (vs. mesh coarsening by aggregating elements in h-multigrid)
- Natural fit for high-order elements on unstructured meshes
- Pairs naturally with matrix-free data structures; can be implemented efficiently with libCEED operators
- Approach here offers robustness of low-order AMG w/ higher efficiency per DoF and lower wall-clock time to meet engineering tolerances



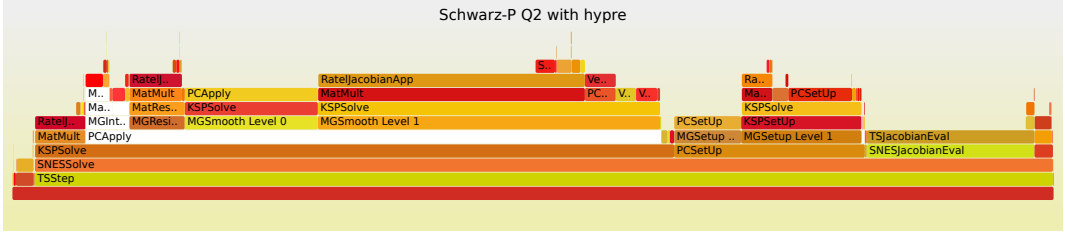


Figure: Flame graph for a typical setup and solve with Q2 elements. Dominant costs are preconditioner application (left half; part of the linear solve) and setup (center-right third). Coarse solve (Level 0 and above) takes about half the time of the fine (Level 1) and coarse setup (AMG) takes more time than fine p-MG setup, despite fine having 8x more DoF.

Nonlinear Solve Efficiency: Q2 elements

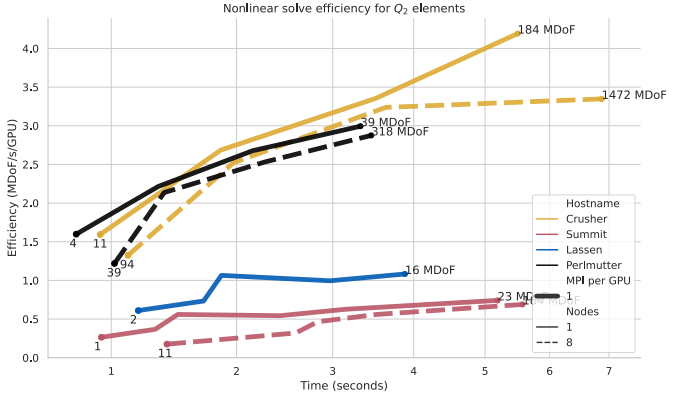


Figure: Efficiency per Newton iteration versus time for Q2 finite elements using matrix-free Newton-Krylov with p-MG preconditioning and Boomer-AMG coarse solve. In this and subsequent figures, problem sizes (in MDoF) are annotated for the minimum and maximum sizes for each host and number of nodes combination. Perfect weak scaling would have the 1-node and 8-node curves on top of each other, with strong scaling limits visible in the minimum time at which acceptable efficiency can be achieved. Impact of latency is ever-present, with memory capacity limiting right end of each curve.

Nonlinear Solve Efficiency: Q3 elements

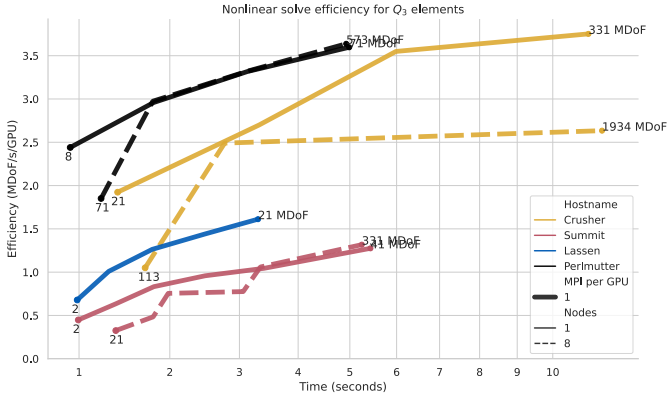


Figure: Efficiency per Newton iteration versus time for Q3 finite elements using matrix-free Newton-Krylov with p-MG preconditioning and Boomer-AMG coarse solve. Ideal weak scaling is evident on Perlmutter for Newton step time above 1.8 s where the 1-node and 8-node curves coincide, while communication latency leads to degradation at the smallest problem sizes (2 MDoF/GPU with time around 1 s). AMG requires a deeper V-cycle for the larger problem size, but this latency impact is hidden at the 2 s solve time with Q3 elements (noticeable in the Q2 case, where greater fraction of time is in AMG solve).

Linear Solve Efficiency: Q2 elements

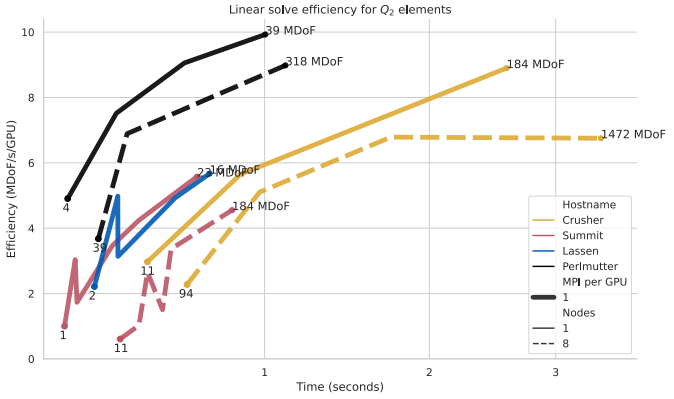


Figure: Linear solve efficiency spectrum for Q2 finite elements using matrix-free Newton-Krylov with p-MG preconditioning and BoomerAMG coarse solve. Times and efficiencies are per Newton iteration. The linear solve (this and next Figure) is communication-intensive since each preconditioner application goes through a V-cycle (with an increasing number of levels as the model gets larger).

Linear Solve Efficiency: Q3 elements

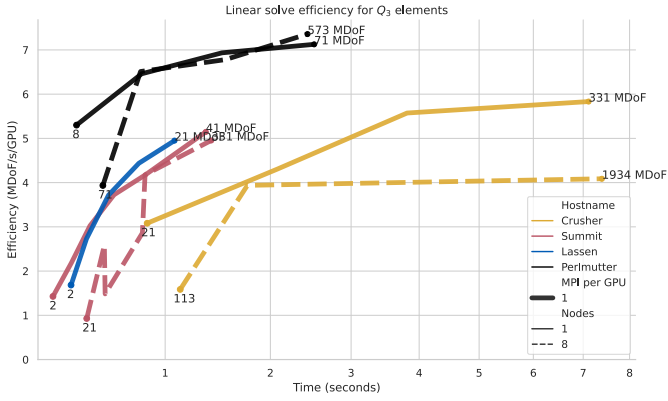


Figure: Linear solve efficiency spectrum for Q3 finite elements using matrix-free Newton-Krylov with p-MG preconditioning and BoomerAMG coarse solve. Times and efficiencies are per Newton iteration.

Preconditioner Setup Efficiency: Q2 elements

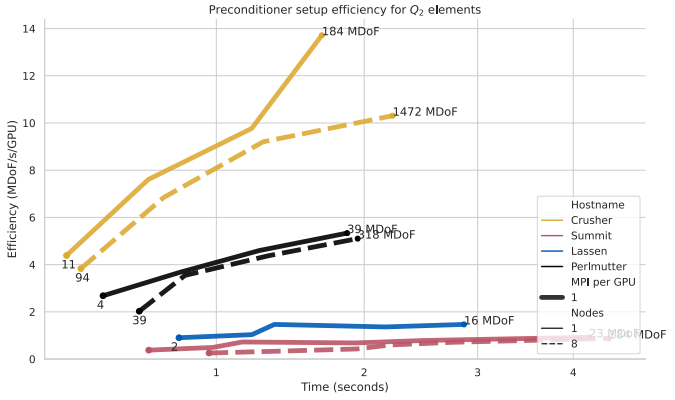


Figure: Preconditioner setup efficiency spectrum for Q2 finite elements using matrix-free Newton-Krylov with p-MG preconditioning and BoomerAMG coarse solve. Times and efficiencies are per Newton iteration. Setup consists of algebraic multigrid analysis and Galerkin products, as well as a few Krylov iterations to calibrate the smoothers.

Preconditioner Setup Efficiency: Q3 elements

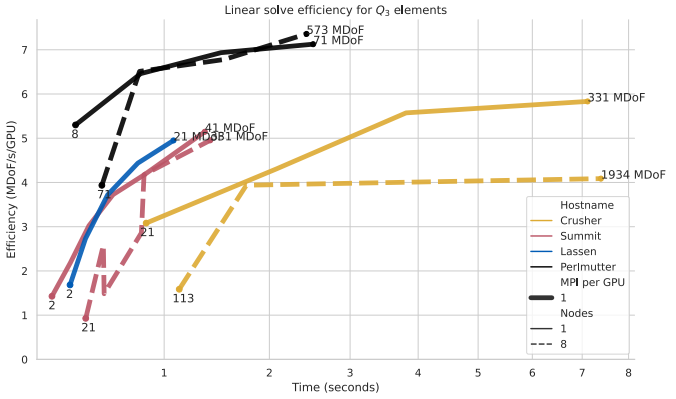
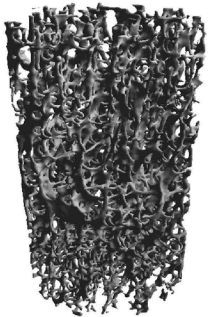


Figure: Preconditioner setup efficiency spectrum for Q3 finite elements using matrix-free Newton-Krylov with p-MG preconditioning and BoomerAMG coarse solve. Times and efficiencies are per Newton iteration. Relative cost of Jacobian assembly and preconditioner setup is decreased for Q3 elements because the coarse problem is a smaller fraction of the fine problem size.

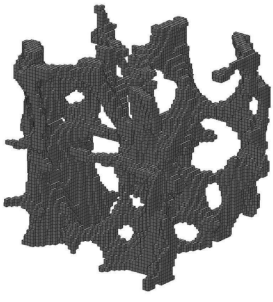
General Observations about Preceding Efficiency Plots

- Tend to see greater volatility in the "strong scaling" regime at left edge of the Figures.
- Most configurations reach high efficiency weak scaling (solid and dotted lines very close) as the problem size per GPU increases, leading to Newton solve times increasing to around 2 s and higher.
- Efficient weak scaling is usually realized at smaller (faster) solves than where performance plateaus, indicating that single-node architectural latencies are a more insidious performance obstacle than multi-node communication.
- Crusher (the Frontier testbed) exhibits a regime of efficient scaling, but efficiency degrades at the largest problem sizes. This effect is not present on other machines; profiling points to network degradation not identifiable in microbenchmarks (or smaller problem sizes) that will hopefully be resolved in the MPI implementation or tuning.

2004 Gordon Bell Winner (Adams et al.): Implicit FEM for Solid Mechanics with > 0.5B DoFs



micro-CT @ 22 μm resolution
Ø:8 mm H:15 mm



micro-FE mesh
2.5 x 2.5 x 2.5 mm^3
44 μm hexahedral elements

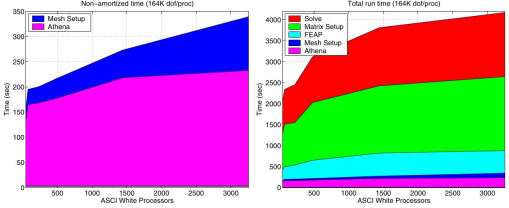


FIGURE 5.6. *Total end-to-end solution times - 164K dof per processor*

2004 Gordon Bell Winner: ASCI White vs. Single Crusher/Frontier Node

130 nodes of

vs. $\frac{1}{2}$ of

Metric	Adams et al. 2004	Ratel	Ratel
Discretization	linear	quadratic	cubic
Machine	ASCI White 130 nodes	Crusher 1 node	Crusher 1 node
Peak Bandwidth	1.56 TB/s	12 TB/s	12 TB/s
Degrees of Freedom	237 M	184 M	331 M
kDoF/GB	460	400	700
Load Step Strain	0.5%	12%	12%
kDoF/s per Load Step	600	6000	5500

Implicit Solid Mechanics: Old Performance Model

Iterative Solvers: Bandwidth

- SpMV arithmetic intensity of 1/6 flop/byte
- Preconditioners also mostly bandwidth
  - Architectural latency a big problem on GPUs, especially for sparse triangular solves.
  - Sparse matrix-matrix products for AMG setup

Direct Solvers: Bandwidth and Dense Compute

- Leaf work in sparse direct solves
- Dense factorization of supernodes
  - Fundamentally nonscalable, granularity on GPUs is already too big to apply on subdomains
- Research on H-matrix approximations (e.g., in STRUMPACK)

Implicit Solid Mechanics: New Performance Model

Still Mostly Bandwidth

- Reduce storage needed at quadrature points
  - Half the cost of a sparse matrix already for linear elements
  - Big efficiency gains for high order
- Assembled coarse levels are much smaller.

Compute

- Kernel fusion is necessary
- Balance vectorization with cache/occupancy
- $O(n)$, but benefits from BLIS-like (Van Zee and van de Geijn, 2015) abstractions

```
| BLIS | libCEED | |------|---------| | packing | batched element restriction |
| microkernel | basis action | | ? | user-provided qfunctions |
```

Summary

The foundation that PETSc has built (and continues to improve) over the years for scaling across many nodes is ready for exascale-class node counts.

Intensive work on GPU support (for multiple programming models) over the last few years has added the missing piece for upcoming exascale installations:

- Full support for AMD, Intel, and NVIDIA GPUs
- Support for efficient on-device matrix assembly
- Continuing to add helper routines to facilitate writing user callbacks running on GPUs

Ongoing efforts to explore and optimize performance on exascale and other GPU-based systems:

- Improving communication performance and leveraging asynchronicity (only briefly discussed)
- Many activities not discussed here: Adding batched solvers that aggregate many small solves onto GPU; algorithmic improvements for better GPU utilization by quasi-Newton methods, etc.

Philosophy / Vision for the Future

Enable rapid experimentation

"*Compilers are error reporting tools with a code generation side gig*" —Esteban Küber
"*PETSc is a library for diagnostics about differential algebraic systems that moonlights as a solver*" —Jed Brown

- It's critical to be able to experiment rapidly (generating said diagnostics) to peel away the mysticism in solver design for emerging architectures:
  - Use late binding whenever possible: Composition of solvers, data structures, communication mechanisms, execution target (host vs. device), specified at execution time.
  - Part of rapid experimenting is having easy access to cool algorithms/implementations others have developed, hence emphasis on having interfaces to many other packages (hypre, Kokkos, STRUMPACK, etc.)

Strive for configuration simplicity and strong encapsulation

- Make it possible for application developers to write code that achieves state of the art performance without directly knowing about GPU compilers or libraries.
- Users can and should choose a GPU abstraction for the nonlinear operations in their problem domain. That choice is independent of what PETSc uses.

Further Information

R. T. Mills, M. F. Adams, S. Balay, J. Brown, A. Dener, M. Knepley, S. E. Kruger, H. Morgan, T. Munson, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, J. Zhang, 2021. "Toward Performance-Portable PETSc for GPU-based Exascale Systems". Parallel Computing. doi:10.1016/j.parco.2021.102831. Preprint available as arXiv:2011.00715 [cs.MS].

J. Zhang, J. Brown, S. Balay, J. Faibussowitsch, M. Knepley, O. Marin, R. T. Mills, T. Munson, B. F. Smith, S. Zampini, 2021. "The PetscSF Scalable Communication Layer". IEEE Transactions on Parallel and Distributed Systems. doi: 10.1109/TPDS.2021.3084070. Preprint available as arXiv:2102.13018 [cs.DC].

(Also see "Rethinking MPI for GPU Accelerated Supercomputers" by Nicole Hemsoth, which appeared on *The Next Platform* in March 2021, for a non-technical summary of the PetscSF paper.)

J. Brown, V. Barra, N. Beams, L. Ghaffari, M. Knepley, W. Moses, R. Shakeri, K. Stengel, J. L. Thompson, J. Zhang, 2022. "Performance Portable Solid Mechanics via Matrix-Free *p*-Multigrid". arXiv:2204.01722 [cs.MS].

Questions? Want help using PETSc to run your code at extreme scales?
Write to `petsc-users@mcs.anl.gov` (public forum)
or
`petsc-maint@mcs.anl.gov` (only goes to PETSc maintainers).