PETSc Tutorial
Profiling, Nonlinear Solvers, Unstructured Grids, Threads and GPUs

Karl Rupp
me@karlrupp.net

Freelance Computational Scientist

Seminarzentrum-Hotel Am Spiegeln, Vienna, Austria

June 28-30, 2016

Table of Contents

Debugging and Profiling

PETSc Debugging

By default, a debug build is provided

Launch the debugger

```
-start_in_debugger [gdb,dbx,noxterm]
-on_error_attach_debugger [gdb,dbx,noxterm]
```

Attach the debugger only to some parallel processes

```
-debugger_nodes 0,1
```

Set the display (often necessary on a cluster)

```
-display :0
```

Debugging Tips

Put a breakpoint in `PetscError()` to catch errors as they occur

PETSc tracks memory overwrites at both ends of arrays

- The `CHKMEMQ` macro causes a check of all allocated memory
- Track memory overwrites by bracketing them with `CHKMEMQ`

PETSc checks for leaked memory

- Use `PetscMalloc()` and `PetscFree()` for all allocation
- Print unfreed memory on `PetscFinalize()` with `-malloc_dump`

Simply the best tool today is Valgrind

- It checks memory access, cache performance, memory usage, etc.
- http://www.valgrind.org
- Pass `-malloc 0` to PETSc when running under Valgrind
- Might need `--trace-children=yes` when running under MPI
- `--track-origins=yes` handy for uninitialized memory

PETSc Profiling

Profiling

- Use `-log_summary` for a performance profile
  - Event timing
  - Event flops
  - Memory usage
  - MPI messages
- Call `PetscLogStagePush()` and `PetscLogStagePop()`
  - User can add new stages
- Call `PetscLogEventBegin()` and `PetscLogEventEnd()`
  - User can add new events
- Call `PetscLogFlops()` to include your flops

Reading -log_summary

```
                         Max       Max/Min        Avg      Total
Time (sec):           1.548e+02      1.00122   1.547e+02
Objects:              1.028e+03      1.00000   1.028e+03
Flops:                1.519e+10      1.01953   1.505e+10  1.204e+11
Flops/sec:            9.814e+07      1.01829   9.727e+07  7.782e+08
MPI Messages:         8.854e+03      1.00556   8.819e+03  7.055e+04
MPI Message Lengths:  1.936e+08      1.00950   2.185e+04  1.541e+09
MPI Reductions:       2.799e+03      1.00000
```

- Also a summary per stage
- Memory usage per stage (based on when it was allocated)
- Time, messages, reductions, balance, flops per event per stage
- Always send `-log_summary` when asking performance questions on mailing list

PETSc Profiling

```
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %F %M %L %R  %T %F %M %L %R
------------------------------------------------------------------------------------------------------------------------
--- Event Stage 1: Full solve
VecDot                43 1.0 4.8879e-02 8.3 1.77e+06 1.0 0.0e+00 0.0e+00 4.3e+01  0  0  0  0  0   0  0  0  0  1
VecMDot             1747 1.0 1.3021e+00 4.6 8.16e+07 1.0 0.0e+00 0.0e+00 1.7e+03  0  1  0  0 14   1  1  0  0 27
VecNorm             3972 1.0 1.5460e+00 2.5 8.48e+07 1.0 0.0e+00 0.0e+00 4.0e+03  0  1  0  0 31   1  1  0  0 61
VecScale            3261 1.0 1.6703e-01 1.0 3.38e+07 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0
VecScatterBegin     4503 1.0 4.0440e-01 1.0 0.00e+00 0.0 6.1e+07 2.0e+03 0.0e+00  0  0 50 26  0   0  0 96 53  0
VecScatterEnd       4503 1.0 2.8207e+00 6.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0
MatMult             3001 1.0 3.2634e+01 1.1 3.68e+09 1.1 4.9e+07 2.3e+03 0.0e+00 11 22 40 24  0  22 44 78 49  0
MatMultAdd           604 1.0 6.0195e-01 1.0 5.66e+07 1.0 3.7e+06 1.3e+02 0.0e+00  0  0  3  0  0   0  1  6  0  0
MatMultTranspose     676 1.0 1.3220e+00 1.6 6.50e+07 1.0 4.2e+06 1.4e+02 0.0e+00  0  0  3  0  0   1  1  7  0  0
MatSolve            3020 1.0 2.5957e+01 1.0 3.25e+09 1.0 0.0e+00 0.0e+00 0.0e+00  9 21  0  0  0  18 41  0  0  0
MatCholFctrSym         3 1.0 2.8324e-04 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0
MatCholFctrNum        69 1.0 5.7241e+00 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 0.0e+00  2  4  0  0  0   4  9  0  0  0
MatAssemblyBegin     119 1.0 2.8250e+00 1.5 0.00e+00 0.0 2.1e+06 5.4e+04 3.1e+02  1  0  2 24  2   2  0  3 47  5
MatAssemblyEnd       119 1.0 1.9689e+00 1.4 0.00e+00 0.0 2.8e+05 1.3e+03 6.8e+01  1  0  0  0  1   1  0  0  0  1
SNESSolve              4 1.0 1.4302e+02 1.0 8.11e+09 1.0 6.3e+07 3.8e+03 6.3e+03 51 50 52 50 50  99100 99100 97
SNESLineSearch        43 1.0 1.5116e+01 1.0 1.05e+08 1.1 2.4e+06 3.6e+03 1.8e+02  5  1  2  2  1  10  1  4  4  3
SNESFunctionEval      55 1.0 1.4930e+01 1.0 0.00e+00 0.0 1.8e+06 3.3e+03 8.0e+00  5  0  1  1  0  10  0  3  3  0
SNESJacobianEval      43 1.0 3.7077e+01 1.0 7.77e+06 1.0 4.3e+06 2.6e+04 3.0e+02 13  0  4 24  2  26  0  7 48  5
KSPGMRESOrthog      1747 1.0 1.5737e+00 2.9 1.63e+08 1.0 0.0e+00 0.0e+00 1.7e+03  1  1  0  0 14   1  2  0  0 27
KSPSetup             224 1.0 2.1040e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 3.0e+01  0  0  0  0  0   0  0  0  0  0
KSPSolve              43 1.0 8.9988e+01 1.0 7.99e+09 1.0 5.6e+07 2.0e+03 5.8e+03 32 49 46 24 46  62 99 88 48 88
PCSetUp              112 1.0 1.7354e+01 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 8.7e+01  6  4  0  0  1  12  9  0  0  1
PCSetUpOnBlocks     1208 1.0 5.8182e+00 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 8.7e+01  2  4  0  0  1   4  9  0  0  1
PCApply              276 1.0 7.1497e+01 1.0 7.14e+09 1.0 5.2e+07 1.8e+03 5.1e+03 25 44 42 20 41  49 88 81 39 79
```

Communication Costs

Reductions: usually part of Krylov method, latency limited

- `VecDot`
- `VecMDot`
- `VecNorm`
- `MatAssemblyBegin`
- Change algorithm (e.g. IBCGS)

Point-to-point (nearest neighbor), latency or bandwidth

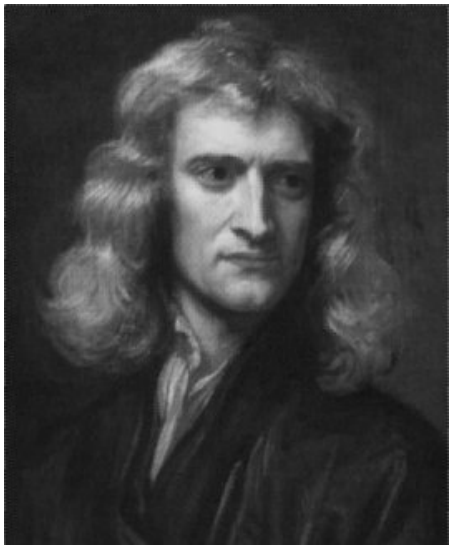
- `VecScatter`
- `MatMult`
- `PCApply`
- `MatAssembly`
- `SNESFunctionEval`
- `SNESJacobianEval`
- Compute subdomain boundary fluxes redundantly
- Ghost exchange for all fields at once
- Better partition

Nonlinear Solvers

Newton Iteration: Workhorse of SNES

Standard form of a nonlinear system

$$-\nabla \cdot \left(|\nabla u|^{\mathfrak{p}-2}\nabla u\right) - \lambda e^{u} = F(u) = 0$$



Iteration

Solve: $J(u)w = -F(u)$

Update: $u^{+} \leftarrow u + w$

Quadratically convergent near a root: $|u^{n+1} - u^{*}| \in \mathcal{O}\left(|u^{n} - u^{*}|^{2}\right)$

Picard is the same operation with a different $J(u)$

Jacobian Matrix for $\mathfrak{p}$-Bratu Equation

$$J(u)w \sim -\nabla\left[(\eta\mathbf{1} + \eta'\nabla u \otimes \nabla u)\nabla w\right] - \lambda e^{u} w$$

$$\eta' = \frac{\mathfrak{p}-2}{2}\eta/(\epsilon^{2} + \gamma)$$

SNES

Scalable Nonlinear Equation Solvers

Newton solvers: Line Search, Thrust Region

Inexact Newton-methods: Newton-Krylov

Matrix-Free Methods: With iterative linear solvers

How to get the Jacobian Matrix?

Implement it by hand

Let PETSc finite-difference it

Use Automatic Differentiation software

Nonlinear solvers in PETSc SNES

LS, TR	Newton-type with line search and trust region
NRichardson	Nonlinear Richardson, usually preconditioned
VIRS, VISS	reduced space and semi-smooth methods for variational inequalities
QN	Quasi-Newton methods like BFGS
NGMRES	Nonlinear GMRES
NCG	Nonlinear Conjugate Gradients
GS	Nonlinear Gauss-Seidel/multiplicative Schwarz sweeps
FAS	Full approximation scheme (nonlinear multigrid)
MS	Multi-stage smoothers, often used with FAS for hyperbolic problems
Shell	Your method, often used as a (nonlinear) preconditioner

SNES Paradigm

SNES Interface based upon Callback Functions

`FormFunction()`, set by `SNESSetFunction()`

`FormJacobian()`, set by `SNESSetJacobian()`

Evaluating the nonlinear residual $F(x)$

Solver calls the **user's** function

User function gets application state through the `ctx` variable

PETSc *never* sees application data

SNES Function

$$F(u) = 0$$

The user provided function which calculates the nonlinear residual has signature

```
PetscErrorCode (*func)(SNES snes,
                       Vec x,Vec r,
                       void *ctx)
```

`x` - The current solution

`r` - The residual

`ctx` - The user context passed to `SNESSetFunction()`

Use this to pass application information, e.g. physical constants

SNES Jacobian

User-provided function calculating the Jacobian Matrix

```
PetscErrorCode (*func)(SNES snes,Vec x,Mat *J,Mat *M,
                       MatStructure *flag,void *ctx)
```

`x` - The current solution

`J` - The Jacobian

`M` - The Jacobian preconditioning matrix (possibly J itself)

`ctx` - The user context passed to `SNESSetFunction()`

Use this to pass application information, e.g. physical constants

Possible `MatStructure` values are:

`SAME_NONZERO_PATTERN`
`DIFFERENT_NONZERO_PATTERN`

Alternatives

a builtin sparse finite difference approximation (“coloring”)

automatic differentiation (ADIC/ADIFOR)

Finite Difference Jacobians

PETSc can compute and explicitly store a Jacobian

- Dense
  - Activated by `-snes_fd`
  - Computed by `SNESDefaultComputeJacobian()`
- Sparse via colorings
  - Coloring is created by `MatFDColoringCreate()`
  - Computed by `SNESDefaultComputeJacobianColor()`

Also Matrix-free Newton-Krylov via 1st-order FD possible

- Activated by `-snes_mf` without preconditioning
- Activated by `-snes_mf_operator` with user-defined preconditioning
  - Uses preconditioning matrix from `SNESSetJacobian()`

DMDA and SNES

Fusing Distributed Arrays and Nonlinear Solvers

Make DM known to SNES solver

```
SNESSetDM(snes,dm);
```

Attach residual evaluation routine

```
DMDASNESSetFunctionLocal(dm,INSERT_VALUES,
       (DMDASNESFunction)FormFunctionLocal,
                         &user);
```

Ready to Roll

First solver implementation completed

Uses finite-differencing to obtain Jacobian Matrix

Rather slow, but scalable!

PETSc and GPUs

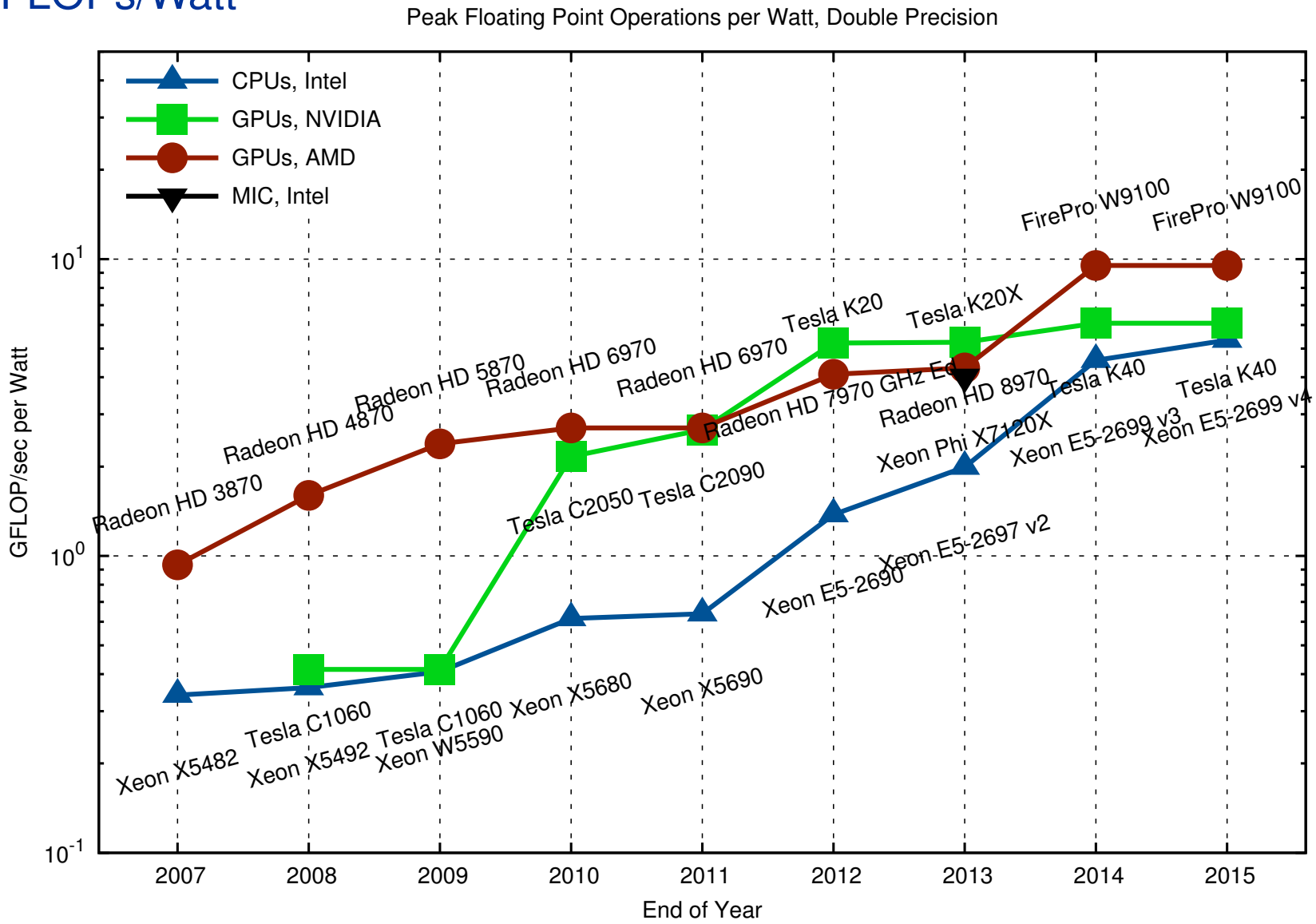
Why bother?

Don't believe anything
unless you can run it

Matt Knepley

Why bother?

GFLOPs/Watt



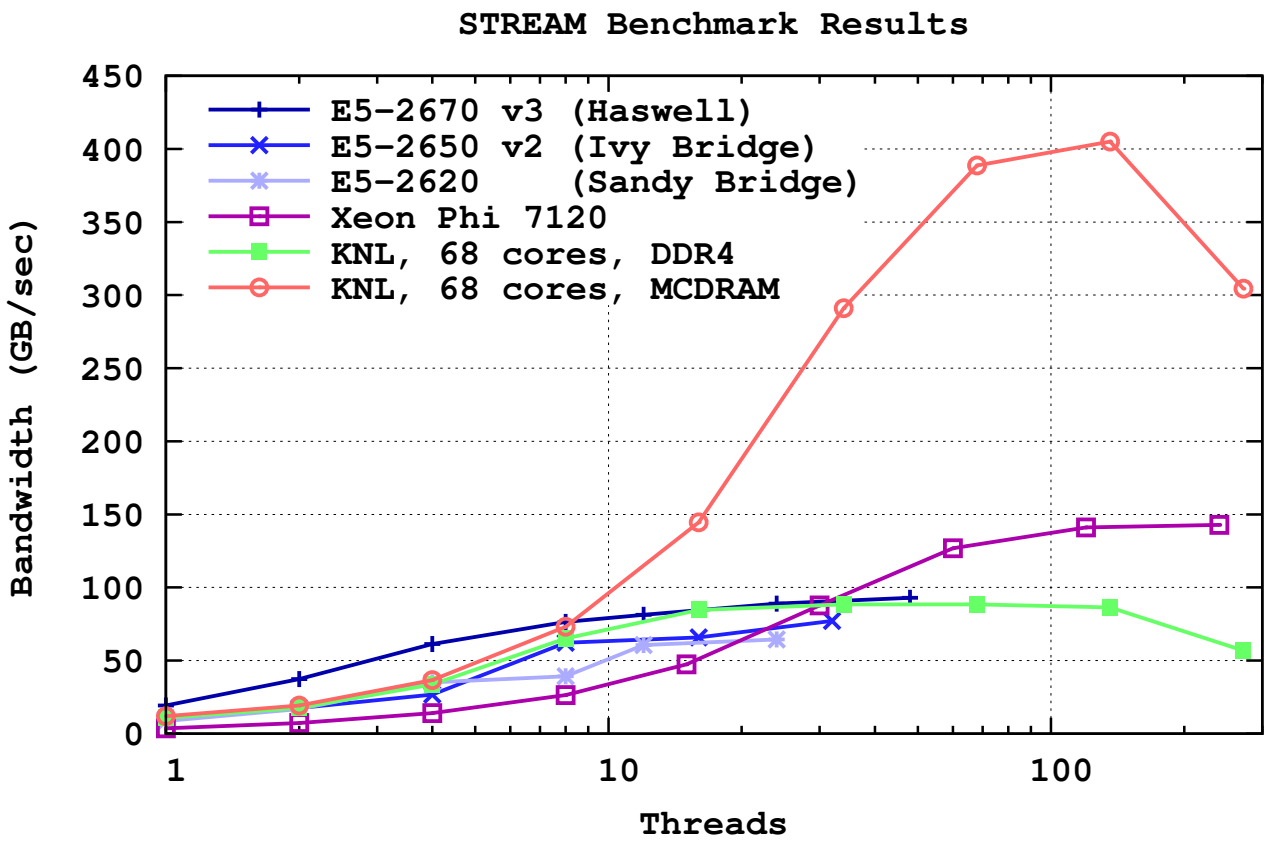
Why bother?

Procurements

Theta (ANL, 2016): 2nd generation INTEL Xeon Phi

Summit (ORNL, 2017), Sierra (LLNL, 2017): NVIDIA Volta GPU

Aurora (ANL, 2018): 3rd generation INTEL Xeon Phi



https://www.karlrupp.net/2016/07/knights-landing-vs-knights-corner-haswell-ivy-bridge-and-sandy-bridge-stream-benchmark-results/

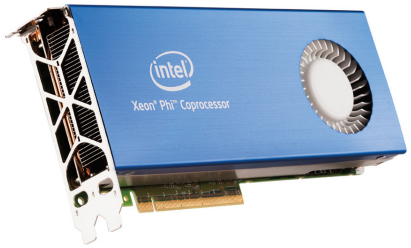
PETSc on GPUs and MIC:

Current Status

Available Options

Native on Xeon Phi

Cross-compile for Xeon Phi



CUDA

CUDA-support through CUSP as well as native

`-vec_type cusp -mat_type aijcusp`

`-vec_type cuda -mat_type aijcusparse`

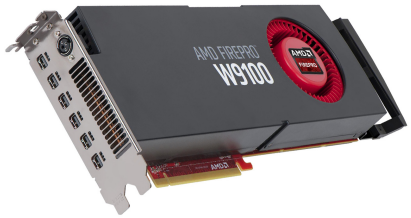
Only for NVIDIA GPUs

CUDA/OpenCL/OpenMP

CUDA/OpenCL/OpenMP-support through ViennaCL

`-vec_type viennacl -mat_type aijviennacl`

OpenCL on CPUs and MIC fairly poor



Configuration

CUDA

CUDA-enabled configuration (minimum)

```
./configure [..] --with-cuda=1
```

With CUSP:

```
  --with-cusp=1 --with-cusp-dir=/path/to/cusp
```

Customization:

```
  --with-cudac=/path/to/cuda/bin/nvcc
  --with-cuda-arch=sm_20
```

OpenCL (ViennaCL)

OpenCL-enabled configuration

```
./configure [..] --download-viennacl
   --with-opencl-include=/path/to/OpenCL/include
   --with-opencl-lib=/path/to/libOpenCL.so
```

How Does It Work?

Host and Device Data

```
struct _p_Vec {
  ...
  void           *data;             // host buffer
  PetscCUSPFlag valid_GPU_array;    // flag
  void           *spptr;            // device buffer
};
```

Possible Flag States

```
  typedef enum {PETSC_CUSP_UNALLOCATED,
                PETSC_CUSP_GPU,
                PETSC_CUSP_CPU,
                PETSC_CUSP_BOTH} PetscCUSPFlag;
```

How Does It Work?

Fallback-Operations on Host

Data becomes valid on host (`PETSC_CUSP_CPU`)

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

Accelerated Operations on Device

Data becomes valid on device (`PETSC_CUSP_GPU`)

```
PetscErrorCode VecAYPX_SeqCUSP(..) {
  VecCUSPGetArrayReadWrite(...);
  // some operation on raw handles on device
  VecCUSPRestoreArrayReadWrite(...);
}
```

Example

KSP ex12 on Host

```
$> ./ex12
    -pc_type ilu -m 200 -n 200 -log_summary
```

```
KSPGMRESOrthog       228 1.0 6.2901e-01
KSPSolve               1 1.0 2.7332e+00
```

KSP ex12 on Device

```
$> ./ex12 -vec_type cusp -mat_type aijcusp
    -pc_type ilu -m 200 -n 200 -log_summary
```

```
[0]PETSC ERROR: MatSolverPackage petsc does not support
    matrix type seqaijcusp
```

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 cusp -mat_type aijcusp
    -pc_type none -m 200 -n 200 -log_summary
```

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

Pitfalls

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 too small: Kernel launch latencies dominate

Data too big: Out of memory

Pitfall: Function Pointers

Pass CUDA function “pointers” through library boundaries?

OpenCL: Pass kernel sources, user-data hard to pass

Composability?

Current GPU-Functionality in PETSc

	CUSP/CUDA	**ViennaCL**
Programming Model	CUDA	CUDA/OpenCL/OpenMP
Operations	Vector, MatMult	Vector, MatMult
Matrix Formats	CSR, ELL, HYB	CSR
Preconditioners	SA-AMG, BiCGStab	-
MPI-related	Scatter	-

Additional Functionality

MatMult via cuSPARSE

OpenCL residual evaluation for PetscFE

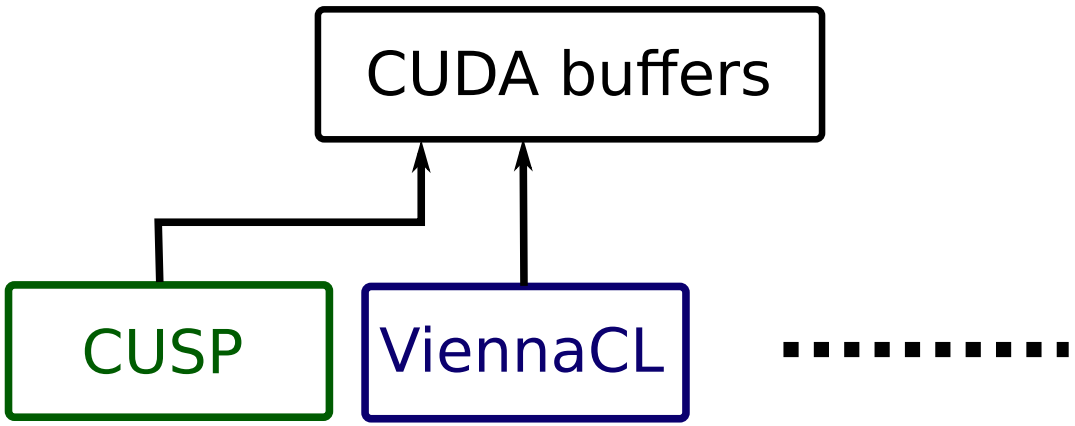
PETSc on GPUs and MIC:

Current Directions

Current: CUDA

Split CUDA-buffers from CUSP

- Vector operations by cuBLAS
- MatMult by different packages
- CUSP (and others) provides add-on functionality



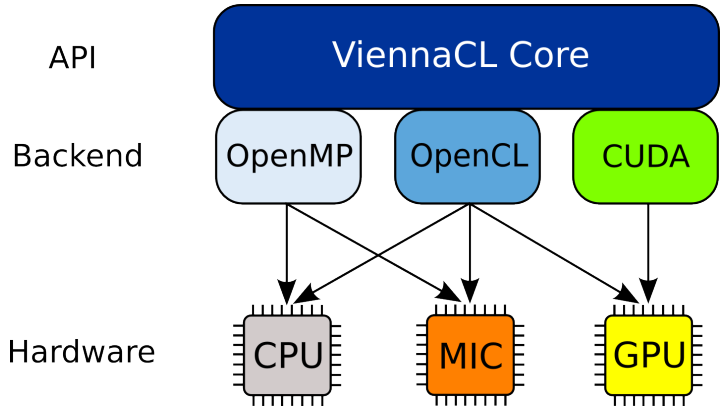
More CUSP Functionality in PETSc

- Relaxations (Gauss-Seidel, SOR)
- Polynomial preconditioners
- Approximate inverses

Current: PETSc + ViennaCL

ViennaCL

- CUDA, OpenCL, OpenMP backends
- Backend switch at **runtime**
- Only OpenCL exposed in PETSc
- Focus on shared memory machines



Recent Advances

- Pipelined Krylov solvers
- Fast sparse matrix-vector products
- Fast sparse matrix-matrix products
- Fine-grained algebraic multigrid
- Fine-grained parallel ILU

Current: PETSc + ViennaCL

Current Use of ViennaCL in PETSc

```
$> ./ex12 -vec_type viennacl -mat_type aijviennacl ...
```

Executes on OpenCL device

New Use of ViennaCL in PETSc

```
$> ./ex12 -vec_type viennacl -mat_type aijviennacl
          -viennacl_backend openmp ...
```

Pros and Cons

Use CPU + GPU simultaneously

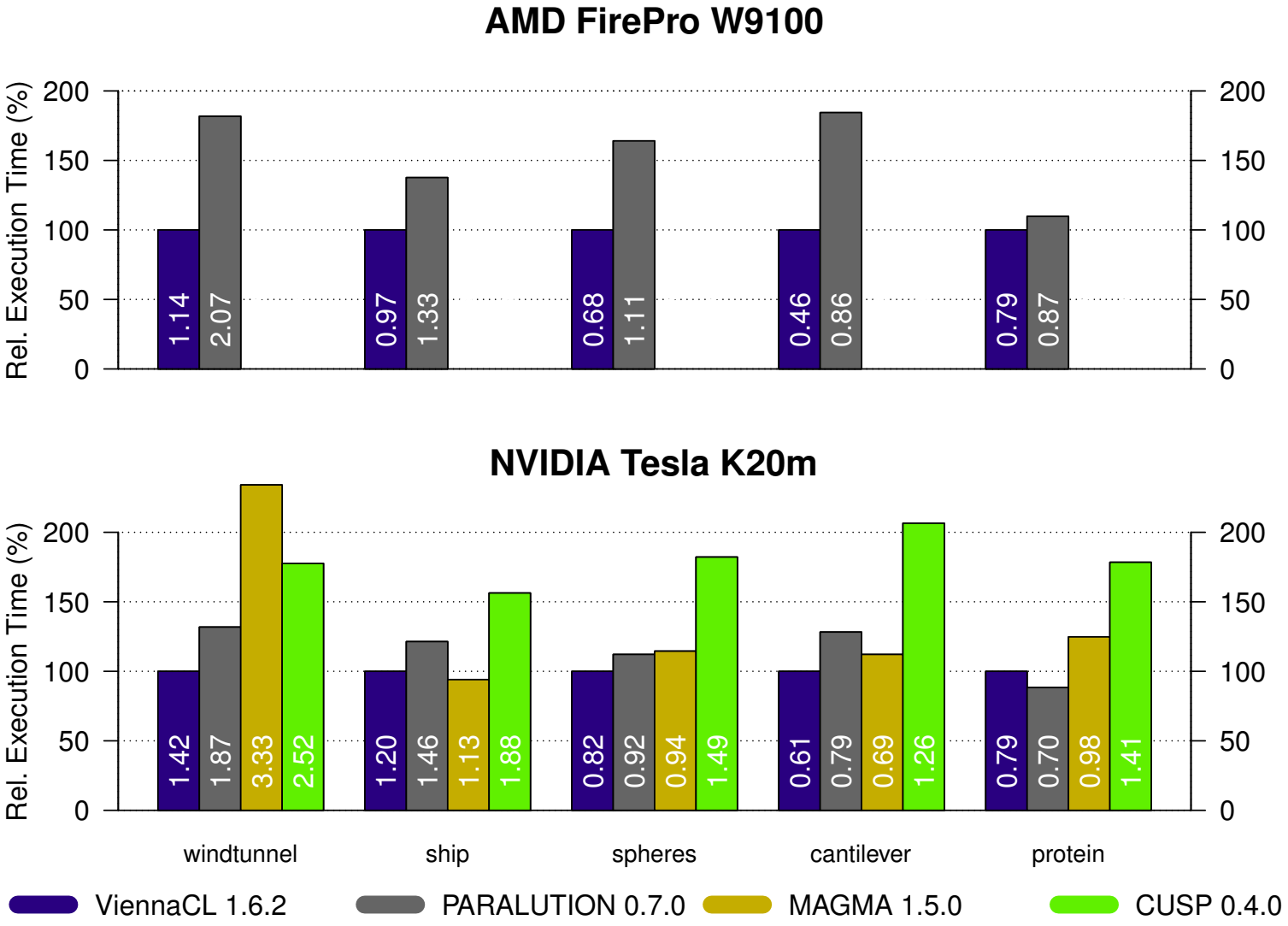
Non-intrusive, use plugin-mechanism

Non-optimal in strong-scaling limit

Gather experiences for best long-term solution

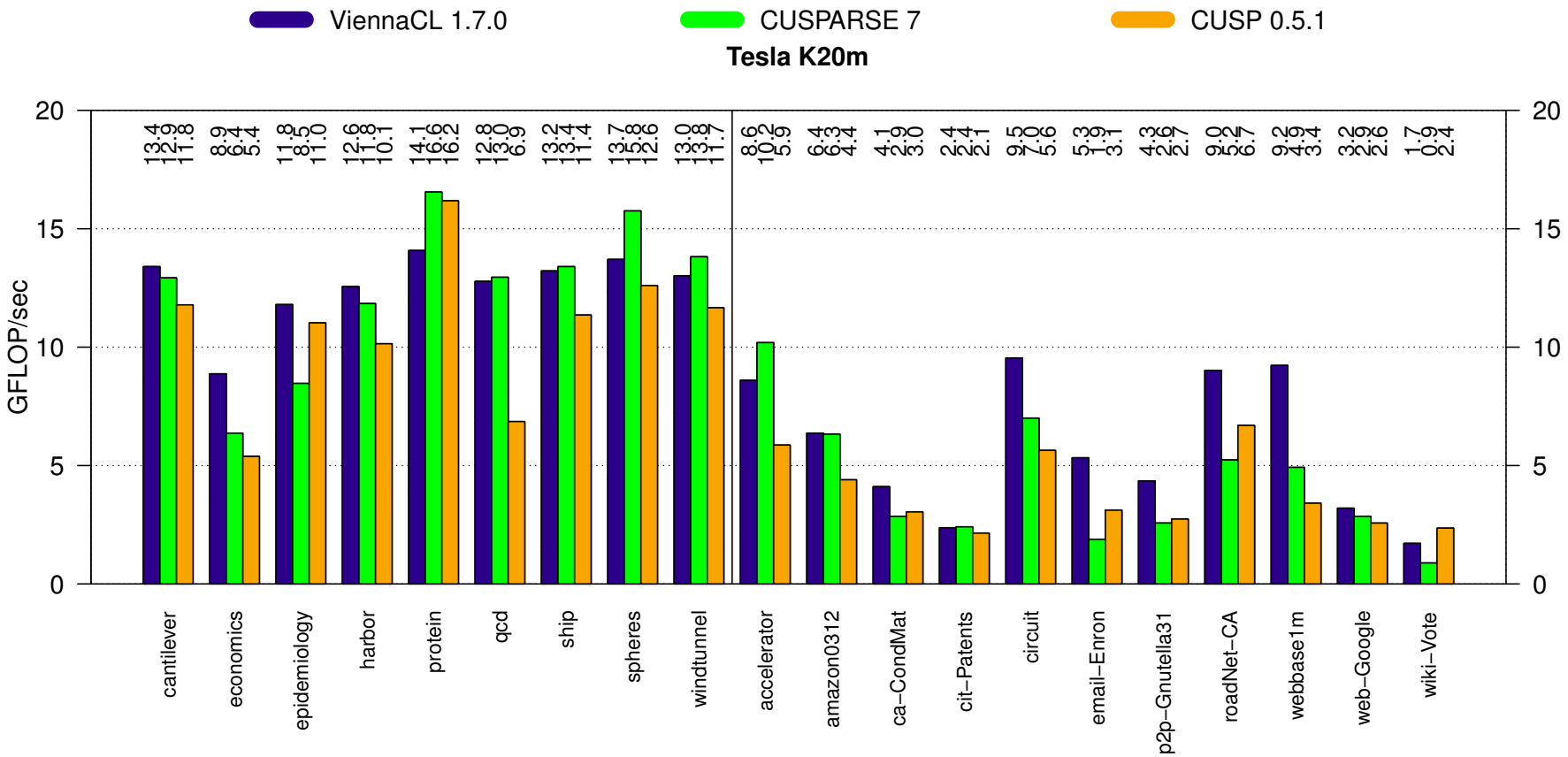
Upcoming PETSc+ViennaCL Features

Pipelined CG Method, Exec. Time per Iteration



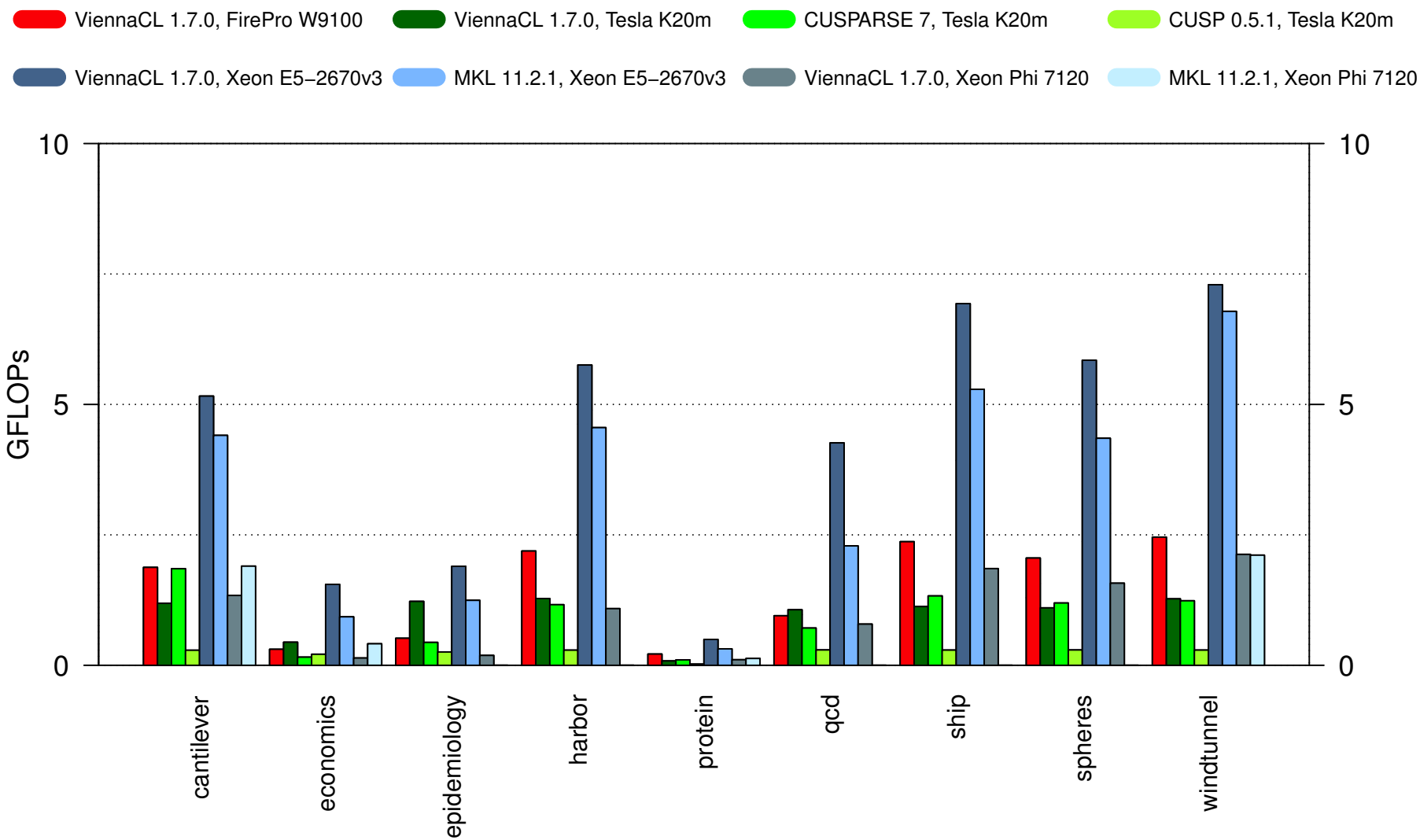
Upcoming PETSc+ViennaCL Features

Sparse Matrix-Vector Multiplication



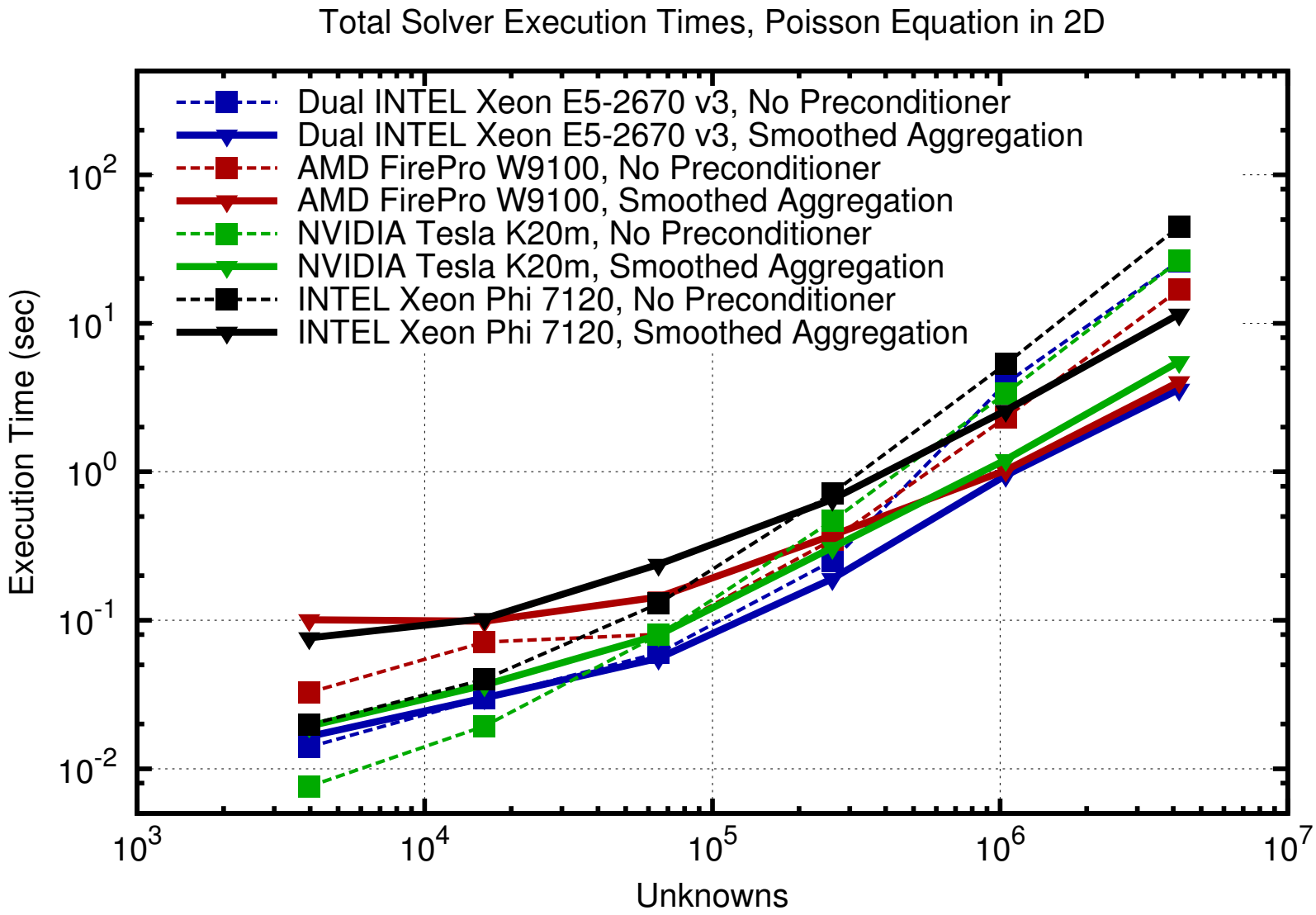
Upcoming PETSc+ViennaCL Feature

Sparse Matrix-Matrix Products



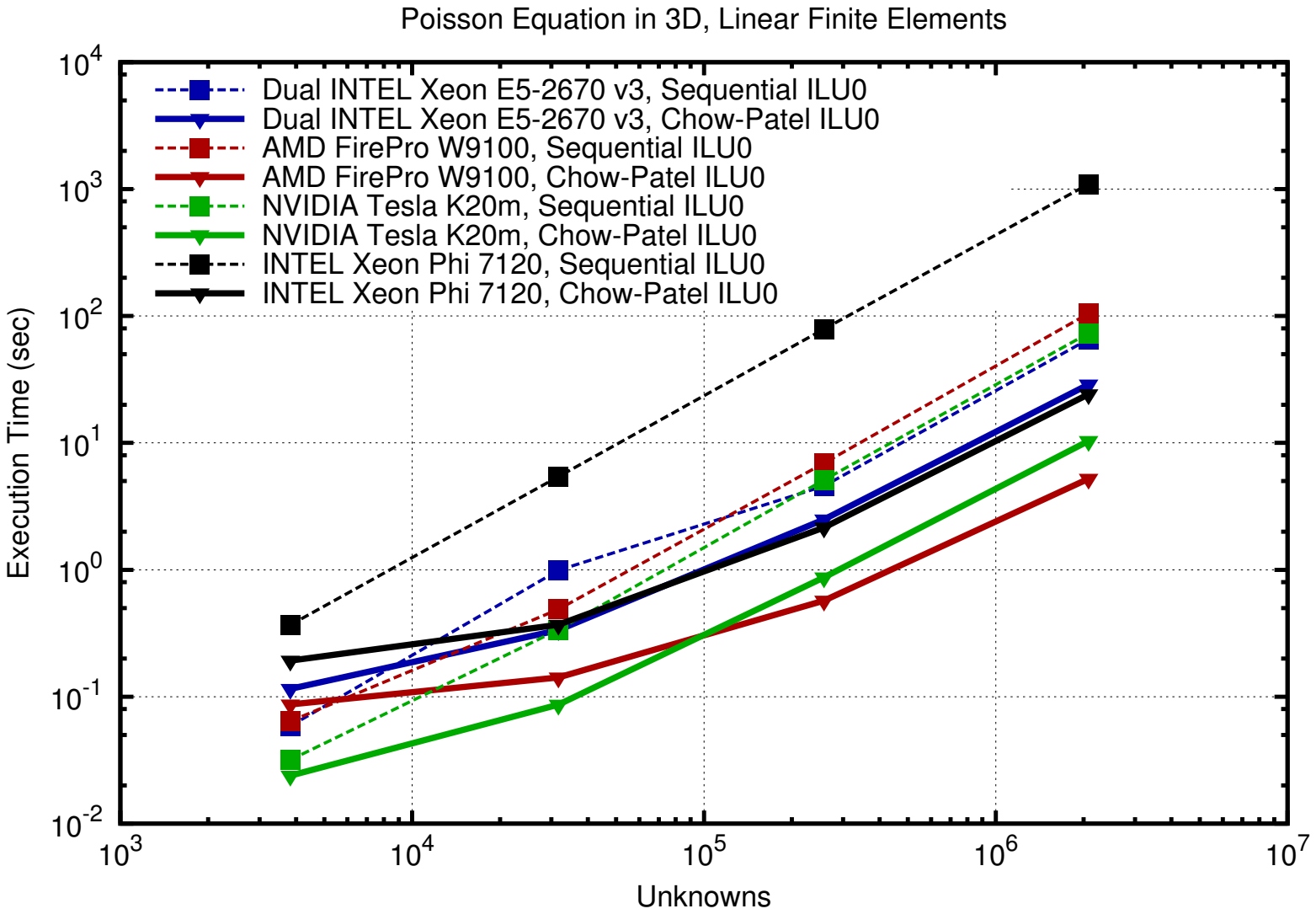
Upcoming PETSc+ViennaCL Feature

Algebraic Multigrid Preconditioners



Pipelined Solvers

Fine-Grained Parallel ILU (Chow and Patel, SISC, 2015)



GPU Summary and Conclusion

Currently Available

- CUSP for CUDA, ViennaCL for OpenCL
- Automatic use for vector operations and SpMV
- Smoothed Agg. AMG via CUSP
- ViennaCL as CUDA/OpenCL/OpenMP-hydra

Current Activities

- Use of cuBLAS and cuSPARSE
- Better support for $n > 1$ processes

Conclusions

PETSc can help You

solve algebraic and DAE problems in your application area

rapidly develop efficient parallel code, can start from examples

develop new solution methods and data structures

debug and analyze performance

advice on software design, solution algorithms, and performance

`petsc-{users,dev,maint}@mcs.anl.gov`

You can help PETSc

report bugs and inconsistencies, or if you think there is a better way

tell us if the documentation is inconsistent or unclear

consider developing new algebraic methods as plugins, contribute if your idea works