PETSc
Portable, Extensible Toolkit for Scientific Computation

Karl Rupp
me@karlrupp.net

Freelance Computational Scientist

Workshop on Modern Scientific Computing
DTU Compute, Technical University of Denmark

March 2-3, 2016
Before we start...

Goal of this Workshop

You are here to learn new things about HPC

Ask Questions

Tell me if you do not understand
Ask for further details
Don’t be shy
Table of Contents

Debugging and Profiling

Nonlinear Solvers

Time Steppers

PETSc and GPUs
Debugging and Profiling
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.
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_view

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

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_view when asking performance questions on mailing list
<table>
<thead>
<tr>
<th>Event</th>
<th>Count</th>
<th>Time (sec)</th>
<th>Flops</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
<th>---</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max</td>
<td>Ratio</td>
<td>Max</td>
<td>Ratio</td>
<td>Mess</td>
<td>Avg len</td>
<td>Reduct</td>
<td>%T</td>
<td>%F</td>
<td>%M</td>
<td>%L</td>
<td>%R</td>
<td>Mflop/s</td>
<td></td>
</tr>
<tr>
<td>PETSc Profiling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VecDot</td>
<td>43</td>
<td>1.0</td>
<td>4.8879e-02</td>
<td>8.3</td>
<td>1.77e+06</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>4.3e+01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VecMDot</td>
<td>1747</td>
<td>1.0</td>
<td>1.3021e+00</td>
<td>4.6</td>
<td>8.16e+07</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>1.7e+03</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>VecNorm</td>
<td>3972</td>
<td>1.0</td>
<td>1.5460e+00</td>
<td>2.5</td>
<td>8.48e+07</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>4.0e+03</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>VecScale</td>
<td>3261</td>
<td>1.0</td>
<td>1.6703e+00</td>
<td>3.38e+07</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VecScatterBegin</td>
<td>4503</td>
<td>1.0</td>
<td>4.0440e+00</td>
<td>0.0</td>
<td>6.1e+07</td>
<td>2.0e+03</td>
<td>0.0e+00</td>
<td>0</td>
<td>50</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VecScatterEnd</td>
<td>4503</td>
<td>1.0</td>
<td>2.8207e+00</td>
<td>6.4</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MatMult</td>
<td>3001</td>
<td>1.1</td>
<td>3.2634e+01</td>
<td>1.1</td>
<td>4.9e+07</td>
<td>2.3e+03</td>
<td>0.0e+00</td>
<td>11</td>
<td>22</td>
<td>40</td>
<td>24</td>
<td>0</td>
<td>22</td>
<td>44</td>
</tr>
<tr>
<td>MatMultAdd</td>
<td>604</td>
<td>1.0</td>
<td>6.0195e+01</td>
<td>1.0</td>
<td>5.66e+07</td>
<td>1.3e+02</td>
<td>0.0e+00</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>MatMultTranspose</td>
<td>676</td>
<td>1.0</td>
<td>1.3220e+00</td>
<td>1.6</td>
<td>6.50e+07</td>
<td>1.4e+02</td>
<td>0.0e+00</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>MatSolve</td>
<td>3200</td>
<td>1.0</td>
<td>2.5957e+01</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>9</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>18</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>MatCholFctrSym</td>
<td>3</td>
<td>1.0</td>
<td>2.8324e-04</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MatCholFctrNum</td>
<td>69</td>
<td>1.0</td>
<td>5.7241e+01</td>
<td>1.0</td>
<td>6.75e+08</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>MatAssemblyBegin</td>
<td>119</td>
<td>1.0</td>
<td>2.8250e+01</td>
<td>1.5</td>
<td>2.1e+06</td>
<td>5.4e+04</td>
<td>3.1e+02</td>
<td>1</td>
<td>0</td>
<td>22</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MatAssemblyEnd</td>
<td>119</td>
<td>1.0</td>
<td>1.9689e+01</td>
<td>1.4</td>
<td>2.8e+05</td>
<td>1.3e+03</td>
<td>6.8e+01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SNESolve</td>
<td>4</td>
<td>1.0</td>
<td>1.4302e+02</td>
<td>1.0</td>
<td>6.3e+07</td>
<td>3.8e+03</td>
<td>6.3e+03</td>
<td>5</td>
<td>31</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>SNESSolve</td>
<td>43</td>
<td>1.0</td>
<td>1.5116e+01</td>
<td>1.0</td>
<td>2.4e+06</td>
<td>3.6e+03</td>
<td>1.8e+02</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>SNESFunctionEval</td>
<td>55</td>
<td>1.0</td>
<td>1.4930e+01</td>
<td>1.0</td>
<td>1.8e+06</td>
<td>3.3e+03</td>
<td>8.0e+00</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SNESJacobianEval</td>
<td>43</td>
<td>1.0</td>
<td>3.7077e+01</td>
<td>1.0</td>
<td>4.3e+07</td>
<td>2.6e+04</td>
<td>3.0e+02</td>
<td>13</td>
<td>0</td>
<td>24</td>
<td>2</td>
<td>26</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>KSPGMRESOrthog</td>
<td>1747</td>
<td>1.0</td>
<td>1.5737e+00</td>
<td>2.9</td>
<td>1.6e+08</td>
<td>2.0e+00</td>
<td>1.7e+03</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>KSPSetUp</td>
<td>224</td>
<td>1.0</td>
<td>2.1040e+02</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>0.0e+00</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>KSPSolve</td>
<td>43</td>
<td>1.0</td>
<td>8.9988e+01</td>
<td>1.0</td>
<td>5.6e+07</td>
<td>2.0e+03</td>
<td>5.8e+03</td>
<td>32</td>
<td>49</td>
<td>46</td>
<td>24</td>
<td>46</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>PCSetUp</td>
<td>112</td>
<td>1.0</td>
<td>1.7354e+01</td>
<td>1.0</td>
<td>6.7e+08</td>
<td>0.0e+00</td>
<td>8.7e+01</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>PCSetUpOnBlocks</td>
<td>1208</td>
<td>1.0</td>
<td>5.8182e+00</td>
<td>1.0</td>
<td>6.75e+08</td>
<td>0.0e+00</td>
<td>8.7e+01</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PCApply</td>
<td>276</td>
<td>1.0</td>
<td>7.1497e+01</td>
<td>1.0</td>
<td>5.2e+07</td>
<td>1.8e+03</td>
<td>5.1e+03</td>
<td>25</td>
<td>44</td>
<td>42</td>
<td>20</td>
<td>41</td>
<td>49</td>
<td></td>
</tr>
</tbody>
</table>
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

\[-\nabla \cdot (|\nabla u|^{p-2}\nabla u) - \lambda e^u = F(u) = 0\]

Iteration

Solve: \[J(u)w = -F(u)\]
Update: \[u^+ \leftarrow u + w\]

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

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

Jacobian Matrix for \(p\)-Bratu Equation

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

\[\eta' = \frac{p - 2}{2} \frac{\eta}{(\epsilon^2 + \gamma)}\]
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
- **NR**: 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 $\text{ctx}$ variable

PETSc never sees application data
The user provided function which calculates the nonlinear residual has signature

```c
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
User-provided function calculating the Jacobian Matrix

```c
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)
PETSc can compute and explicitly store a Jacobian

Dense
Activated by \(-\text{snes\_fd}\)
Computed by \text{SNESDefaultComputeJacobian()}\

Sparse via colorings
Coloring is created by \text{MatFDColoringCreate()}\nComputed by \text{SNESDefaultComputeJacobianColor()}\

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

Activated by \(-\text{snes\_mf}\) without preconditioning
Activated by \(-\text{snes\_mf\_operator}\) with user-defined preconditioning
Uses preconditioning matrix from \text{SNESSetJacobian()}
Fusing Distributed Arrays and Nonlinear Solvers

Make DM known to SNES solver

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

Attach residual evaluation routine

```c
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!
Timestepping

Check out slides 162-168 from

PETSc and GPUs
Don’t believe anything unless you can run it

Matt Knepley
Why bother?

GFLOPs/Watt

Peak Floating Point Operations per Watt, Double Precision

- CPUs, Intel
- GPUs, NVIDIA
- GPUs, AMD
- MIC, Intel

End of Year

GFLOP/sec per Watt

- Xeon X5482, Xeon X5492
- Xeon W5590
- Xeon X5680
- Xeon X5690
- Xeon E5-2690
- Xeon E5-2697 v2
- Xeon E5-2699 v3
- Xeon Phi X7120X
- Tesla C1060
- Tesla C2050, Tesla C2090
- Tesla K20, Tesla K20X
- FirePro W9100
- Radeon HD 3870
- Radeon HD 4870
- Radeon HD 5870
- Radeon HD 6970
- Radeon HD 7970 GHz Edition
- Tesla K40
- Xeon E5-2699 v4
- Xeon E5-2697 v2
- Xeon E5-2690
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
**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

STREAM Benchmark Results

- E5-2670 v3 (Haswell)
- E5-2650 v2 (Ivy Bridge)
- E5-2620 (Sandy Bridge)
- Xeon Phi 7120
PETSc on GPUs and MIC:

Current Status
Available Options

Native on Xeon Phi
  Cross-compile for Xeon Phi

CUDA
  CUDA-support through CUSP
  -vec_type cusp -mat_type aijcusp
  Only for NVIDIA GPUs

OpenCL
  OpenCL-support through ViennaCL
  -vec_type viennacl -mat_type aijviennacl
  OpenCL on Xeon Phi very poor
Configuration

CUDA (CUSP)

CUDA-enabled configuration (minimum)

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

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

Possible Flag States

```c
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**)

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

Accelerated Operations on Device

Data becomes valid on device (**PETSC_CUSP_GPU**)

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

KSP ex12 on Host

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

<table>
<thead>
<tr>
<th>KSPGMRESOrthog</th>
<th>228</th>
<th>1.0</th>
<th>6.2901e-01</th>
</tr>
</thead>
<tbody>
<tr>
<td>KSPSolve</td>
<td>1</td>
<td>1.0</td>
<td>2.7332e+00</td>
</tr>
</tbody>
</table>

KSP ex12 on Device

```bash
$> ./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**

<table>
<thead>
<tr>
<th></th>
<th>CUSP</th>
<th>ViennaCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Model</td>
<td>CUDA</td>
<td>OpenCL</td>
</tr>
<tr>
<td>Operations</td>
<td>Vector, MatMult</td>
<td>Vector, MatMult</td>
</tr>
<tr>
<td>Matrix Formats</td>
<td>CSR, ELL, HYB</td>
<td>CSR</td>
</tr>
<tr>
<td>Preconditioners</td>
<td>SA-AMG, BiCGStab</td>
<td>-</td>
</tr>
<tr>
<td>MPI-related</td>
<td>Scatter</td>
<td>-</td>
</tr>
</tbody>
</table>

**Additional Functionality**

- MatMult via cuSPARSE
- OpenCL residual evaluation for PetscFE
PETSc on GPUs and MIC:
Future Directions
Split CUDA-buffers from CUSP

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

CUDA buffers

CUSP  ViennaCL

More CUSP Functionality in PETSc

Relaxations (Gauss-Seidel, SOR)
Polynomial preconditioners
Approximate inverses
Future: 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
Future: PETSc + ViennaCL

Current Use of ViennaCL in PETSc

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

Executes on OpenCL device

Possible Future Use of ViennaCL in PETSc

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

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

**AMD FirePro W9100**

<table>
<thead>
<tr>
<th>Data Set</th>
<th>ViennaCL 1.6.2</th>
<th>PARALUTION 0.7.0</th>
<th>MAGMA 1.5.0</th>
<th>CUSP 0.4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>windtunnel</td>
<td>1.42</td>
<td>1.87</td>
<td>3.33</td>
<td>2.52</td>
</tr>
<tr>
<td>ship</td>
<td>1.46</td>
<td>1.13</td>
<td>1.88</td>
<td>1.49</td>
</tr>
<tr>
<td>spheres</td>
<td>1.20</td>
<td>0.82</td>
<td>0.94</td>
<td>1.49</td>
</tr>
<tr>
<td>cantilever</td>
<td>0.61</td>
<td>0.69</td>
<td>0.79</td>
<td>1.26</td>
</tr>
<tr>
<td>protein</td>
<td>0.79</td>
<td>0.70</td>
<td>0.98</td>
<td>1.41</td>
</tr>
</tbody>
</table>

**NVIDIA Tesla K20m**

<table>
<thead>
<tr>
<th>Data Set</th>
<th>ViennaCL 1.6.2</th>
<th>PARALUTION 0.7.0</th>
<th>MAGMA 1.5.0</th>
<th>CUSP 0.4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>windtunnel</td>
<td>1.14</td>
<td>2.07</td>
<td>0.97</td>
<td>1.33</td>
</tr>
<tr>
<td>ship</td>
<td>1.07</td>
<td>1.11</td>
<td>0.68</td>
<td>0.86</td>
</tr>
<tr>
<td>spheres</td>
<td>1.88</td>
<td>1.88</td>
<td>0.69</td>
<td>0.79</td>
</tr>
<tr>
<td>cantilever</td>
<td>0.79</td>
<td>0.79</td>
<td>1.33</td>
<td>0.70</td>
</tr>
<tr>
<td>protein</td>
<td>1.49</td>
<td>1.49</td>
<td>1.26</td>
<td>1.41</td>
</tr>
</tbody>
</table>
Upcoming PETSc+ViennaCL Features

Sparse Matrix-Vector Multiplication

- ViennaCL 1.7.0
- CUSPARSE 7
- CUSP 0.5.1

Tesla K20m
Upcoming PETSc+ViennaCL Feature

Sparse Matrix-Matrix Products

- ViennaCL 1.7.0, FirePro W9100
- ViennaCL 1.7.0, Tesla K20m
- CUSPARSE 7, Tesla K20m
- CUSP 0.5.1, Tesla K20m
- ViennaCL 1.7.0, Xeon E5-2670v3
- MKL 11.2.1, Xeon E5-2670v3
- ViennaCL 1.7.0, Xeon Phi 7120
- MKL 11.2.1, Xeon Phi 7120

GFLOPs

- cantilever
- economics
- epidemiology
- harbor
- protein
- qcd
- ship
- spheres
- windtunnel
Algebraic Multigrid Preconditioners

Total Solver Execution Times, Poisson Equation in 2D

Execution Time (sec)

Unknowns

Total Solver Execution Times, Poisson Equation in 2D

Dual INTEL Xeon E5-2670 v3, No Preconditioner
Dual INTEL Xeon E5-2670 v3, Smoothed Aggregation
AMD FirePro W9100, No Preconditioner
AMD FirePro W9100, Smoothed Aggregation
NVIDIA Tesla K20m, No Preconditioner
NVIDIA Tesla K20m, Smoothed Aggregation
INTEL Xeon Phi 7120, No Preconditioner
INTEL Xeon Phi 7120, Smoothed Aggregation
Pipelined Solvers

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

Total Solver Execution Times, Poisson Equation in 2D

Execution Time (sec)

Unknowns

Dual INTEL Xeon E5-2670 v3, No Preconditioner
Dual INTEL Xeon E5-2670 v3, Fine-Grained ILU
AMD FirePro W9100, No Preconditioner
AMD FirePro W9100, Fine-Grained ILU
NVIDIA Tesla K20m, No Preconditioner
NVIDIA Tesla K20m, Fine-Grained ILU
INTEL Xeon Phi 7120, No Preconditioner
INTEL Xeon Phi 7120, Fine-Grained ILU
Currently Available

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

Next Steps

- Use of cuBLAS and cuSPARSE
- Better support for $n > 1$ processes
- ViennaCL as CUDA/OpenCL/OpenMP-hydra
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