Argonne NATIONAL LABORATORY

Understanding and Tuning Performance in PETSc (on emerging manycore, GPGPU, **and** traditional architectures)

Richard Tran Mills
(with **major** contributions from Karl Rupp,
and also help from Matt Knepley and Jed Brown)

PETSc User Meeting 2019
June 4, 2019

Table of Contents

What is driving current HPC trends?

Moore's Law (1965)

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

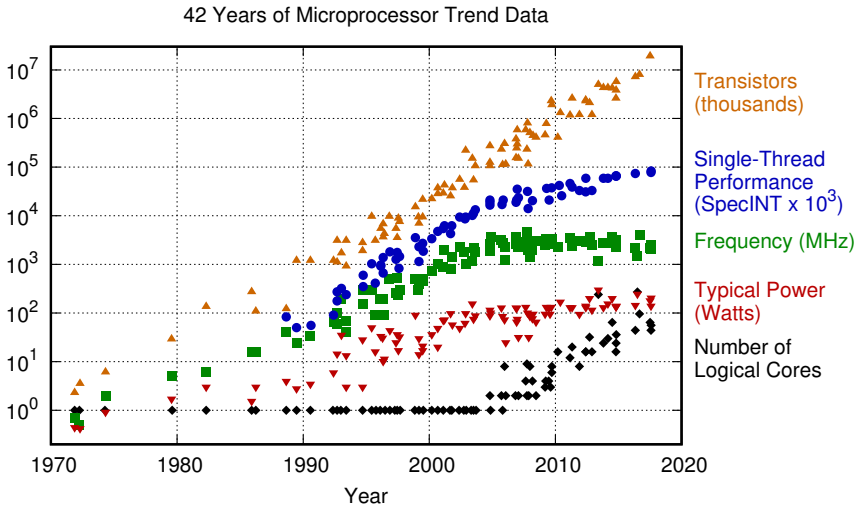
Dennard Scaling (1974)

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

Power Considerations

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

Microprocessor Trend Data



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

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

Current trends in HPC architectures

Emerging architectures are very complex...

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

...and hard to program

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

Table of Contents

FLOPS and Memory Bandwidth

Operations in PETSc tend to

- Deal with large datasets (vectors, sparse matrices)
- Perform few arithmetic operations per byte loaded/stored from main memory; this ratio, the *arithmetic intensity*, is usually below unity.

Modern CPUs support arithmetic intensity around 10 at full utilization

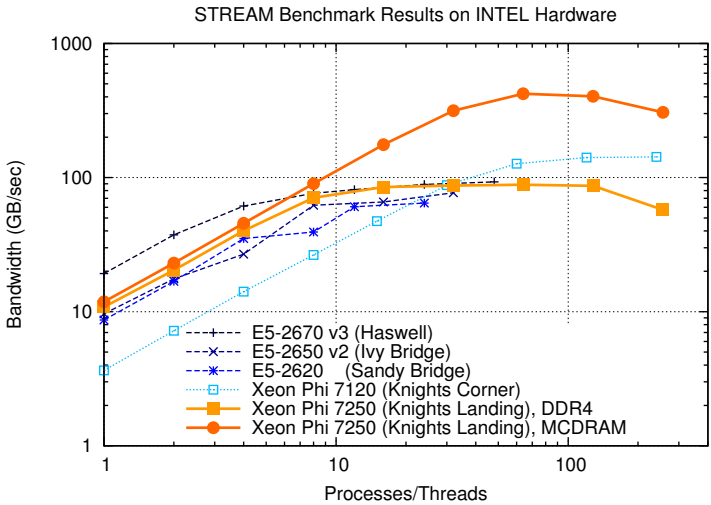
- Most operations in PETSc are limited by the rate at which data can be loaded/stored; they are *memory bandwidth limited*. (We know this from both models and measurements. More on this later.)

Maximizing use of available memory bandwidth is key!

- Process placement is critical on NUMA systems
- Read/write contiguous blocks of memory
- Avoid unordered reads whenever possible
- Vectorization doesn't matter if poor memory bandwidth utilization means VPUs cannot be kept busy!

Memory Bandwidth vs. Processes

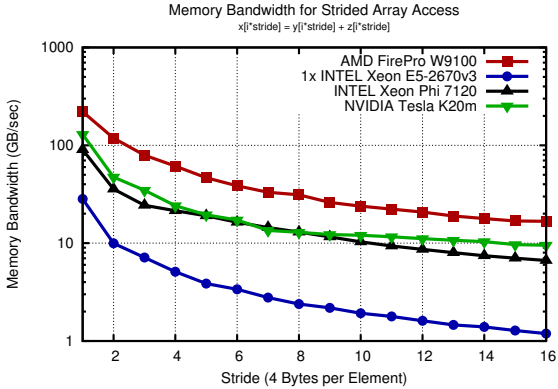
- STREAM Triad computes $\mathbf{w} = \mathbf{y} + \alpha\mathbf{x}$ for large arrays (exceeding cache size)
- Usually saturates quickly; 8-16 processes/threads sufficient for most modern server CPUs
- Little speedup to be gained after this saturation point



FLOPs and Bandwidth

Strided Memory Access

```
void work(double *x, double *y, double *z, int N, int k)
{
  for (size_t i=0; i<N; ++i)
    z[i*k] = x[i*k] + y[i*k];
}
```



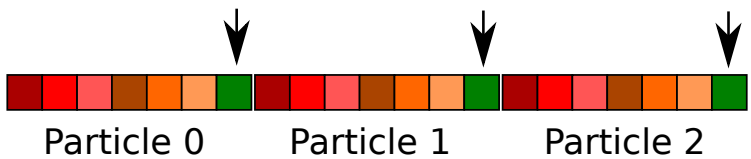
FLOPs and Bandwidth

Strided Memory Access

- Array of structs problematic

```
typedef struct particle
{
  double pos_x; double pos_y; double pos_z;
  double vel_x; double vel_y; double vel_z;
  double mass;
} Particle;

void increase_mass(Particle *particles, int N)
{
  for (int i=0; i<N; ++i)
    particles[i].mass *= 2.0;
}
```



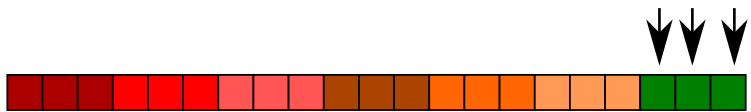
FLOPs and Bandwidth

Strided Memory Access

- Workaround: Structure of Arrays

```
typedef struct particles
{
  double *pos_x; double *pos_y; double *pos_z;
  double *vel_x; double *vel_y; double *vel_z;
  double *mass;
} Particle;

void increase_mass(Particle *particles, int N)
{
  for (int i=0; i<N; ++i)
    particles.mass[i] *= 2.0;
}
```



Check Memory Bandwidth Yourself

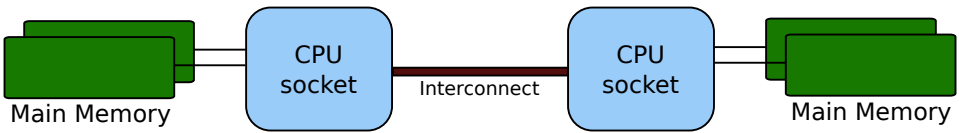
- Set $PETSC_ARCH and then `make streams` in $PETSC_DIR:

```
np  speedup
1 1.0
2 1.58
3 2.19
4 2.42
5 2.63
6 2.69
...
21 3.82
22 3.49
23 3.79
24 3.71
Estimation of possible speedup of MPI programs based on Streams benchmark.
It appears you have 1 node(s)
```

- Expect max speedup of 4X on this machine when running typical PETSc app with multiple MPI ranks on the node
- Most gains already obtained when running with 4–6 ranks.

Non-Uniform Memory Access (NUMA) and Process Placement

Modern compute nodes are typically multi-socket:



Non-uniform memory access (NUMA):

- A process running on one socket has direct access to the memory channels of its CPU...
- ...but requests for memory attached to a different socket must go through the interconnect
- To maximize memory bandwidth, processes should be distributed evenly between the sockets

Non-Uniform Memory Access (NUMA) and Process Placement

Example: 2 sockets, 6 cores per socket, 2 hardware threads per core

Processes all mapped to first socket:

```
$ mpirun -n 6 --bind-to core --map-by core ./stream
process 0 binding: 100000000000100000000000
process 1 binding: 010000000000010000000000
process 2 binding: 001000000000001000000000
process 3 binding: 000100000000000100000000
process 4 binding: 000010000000000010000000
process 5 binding: 000001000000000001000000
Triad:        25510.7507   Rate (MB/s)
```

Processes spread evenly between sockets:

```
$ mpirun -n 6 --bind-to core --map-by socket ./stream
process 0 binding: 100000000000100000000000
process 1 binding: 000000100000000000100000
process 2 binding: 010000000000010000000000
process 3 binding: 000000010000000000010000
process 4 binding: 001000000000001000000000
process 5 binding: 000000001000000000001000
Triad:        45403.1949   Rate (MB/s)
```

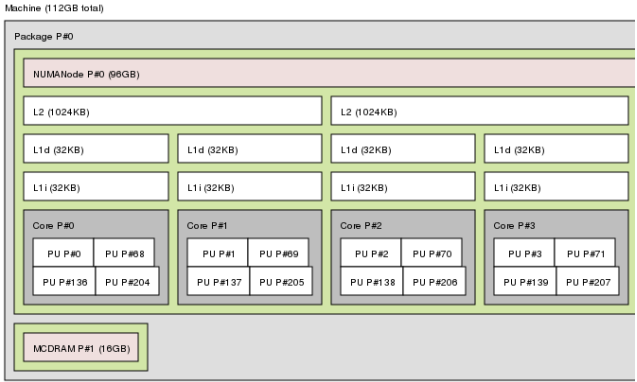
Cannot assume that mpirun defaults to sensible placement!

```
$ make streams
np  speedup
1 1.0
2 1.58
3 2.19
4 2.42
5 2.63
6 2.69
7 2.31
8 2.42
9 2.37
10 2.65
11 2.3
12 2.53
13 2.43
14 2.63
15 2.74
16 2.7
17 3.28
18 3.66
19 3.95
20 3.07
21 3.82
22 3.49
23 3.79
24 3.71
```

```
$ make streams MPI_BINDING="--bind-to core --map-by socket"
np  speedup
1 1.0
2 1.59
3 2.66
4 3.5
5 3.56
6 4.23
7 3.95
8 4.39
9 4.09
10 4.46
11 4.15
12 4.42
13 3.71
14 3.83
15 4.08
16 4.22
17 4.18
18 4.31
19 4.22
20 4.28
21 4.25
22 4.23
23 4.28
24 4.22
```

Additional Process Placement Considerations and Details

- Primary consideration: distribute MPI processes evenly distributed among sockets, thus using all available memory channels.
- Increasingly complex designs, however, mean that performance may also be sensitive to how processes are bound to the resources *within each socket*.
- Preceding examples relatively insensitive: one L3 cache is shared by all cores within a NUMA domain, and each core has its own L2 and L1 caches.
- Processors that are less “flat”, with more complex hierarchies, may be more sensitive.



A portion of the `lstopo` PNG output for an Intel Knights Landing node, showing two tiles.

Cores within a tile share the L2 cache.

Additional Process Placement Considerations and Details

- Placing consecutive MPI ranks on cores that share the same L2 cache may benefit performance if the two ranks communicate frequently with each other, because the latency between cores sharing an L2 cache may be roughly half that of two cores not sharing one.
- There may be benefit, however, in placing consecutive ranks on cores that do not share an L2 cache, because (if there are fewer MPI ranks than cores) this increases the total L2 cache capacity and bandwidth available to the application.
- There is a trade-off to be considered between placing processes close together (in terms of shared resources) to optimize for efficient communication and synchronization vs. farther apart to maximize available resources (memory channels, caches, I/O channels, etc.)
- The best strategy will depend on the application and the software and hardware stack.
- Different process placement strategies can affect performance at least as much as some more commonly explored settings, i.e. compiler optimization levels.
- To make sense of CPU IDs in process placement info, use the Portable Hardware Locality (hwloc) software package's `lstopo` command. If not already on your system, `configure` can install it via `--download-hwloc`.

KNL Process Placement: SNES tutorial example ex19

```
$ export I_MPI_DEBUG=5 # So mappings will be printed
$ export I_MPI_PIN_DOMAIN=auto:compact
$ mpirun -n 68 numactl -p 1 ./ex19 -da_refine 7 -log_view
...
[0] MPI startup(): Rank    Pid      Node name             Pin cpu
[0] MPI startup(): 0       53095    isdp001.cels.anl.gov  {0,68,136,204}
[0] MPI startup(): 1       53096    isdp001.cels.anl.gov  {1,69,137,205}
[0] MPI startup(): 2       53097    isdp001.cels.anl.gov  {2,70,138,206}
[0] MPI startup(): 3       53098    isdp001.cels.anl.gov  {3,71,139,207}
...
                          Max       Max/Min        Avg      Total
Time (sec):           7.093e+00      1.00007   7.093e+00

$ export I_MPI_PIN_DOMAIN=auto:scatter
$ mpirun -n 68 numactl -p 1 ./ex19 -da_refine 7 -log_view
...
[0] MPI startup(): Rank    Pid      Node name             Pin cpu
[0] MPI startup(): 0       53335    isdp001.cels.anl.gov  {0,2,4,6}
[0] MPI startup(): 1       53336    isdp001.cels.anl.gov  {68,70,72,74}
[0] MPI startup(): 2       53337    isdp001.cels.anl.gov  {136,138,140,142}
[0] MPI startup(): 3       53338    isdp001.cels.anl.gov  {204,206,208,210}
...
                          Max       Max/Min        Avg      Total
Time (sec):           1.327e+01      1.00005   1.327e+01
```

Table of Contents

Importance of Performance Models and Measurements

We're almost getting ahead of ourselves: We've already discussed several strategies for dealing with the memory hierarchies and flop/byte balances trending with manycore CPUs.

Before performance tuning, we should consider two rules.

Rule 1: Don't try to tune/optimize code performance without meaningful performance measurements.

- Otherwise, you can waste tons of time "optimizing" things that will make no difference.

Rule 2: Don't think you have meaningful performance measurements if you don't have a performance model.

- Needed to help verify your implementation.
- Needed to have some understanding of why the performance is what it is, and how it can be improved. Otherwise, you are doomed to trying random things and hoping they make the code faster.
- Even a very crude, approximate model is better than no model.

Bottleneck Potpourri

Latency

- Bottleneck in strong scaling limit
- Ultimate limit for time stepping

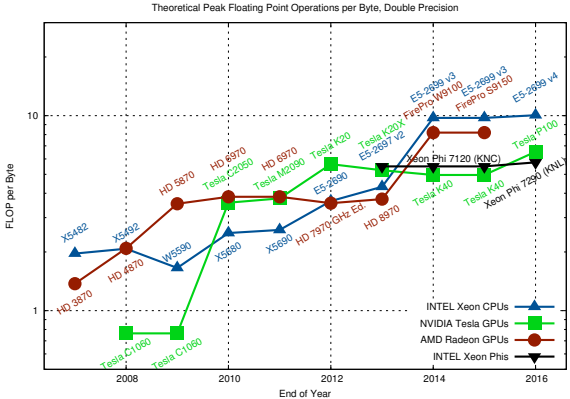
Latency - Sources

- Network latency (Ethernet $\sim 20\mu$s, Infiniband $\sim 5\mu$s)
- PCI-Express latency (Kernel launches, $\sim 10\mu$s)
- Thread synchronization (barriers, locks, $\sim 1 - 100\mu$s)
- Memory latency ($\sim$ 100ns)

Bottleneck Potpourri

Arithmetic Intensity

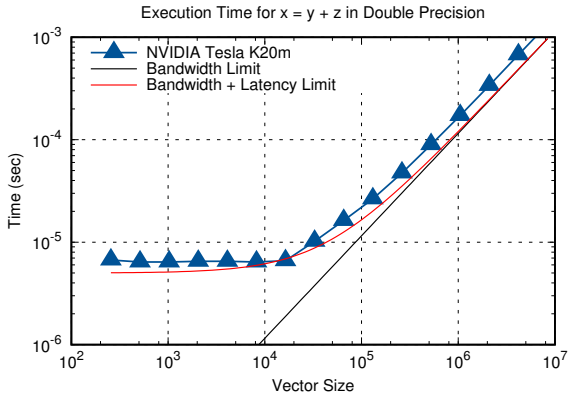
- Number of FLOPs per Byte
- FLOP-limited: Arithmetic intensity larger than ~ 10
- Memory-limited: Arithmetic intensity smaller than ~ 1



Performance Modeling: Vector Addition

Vector Addition

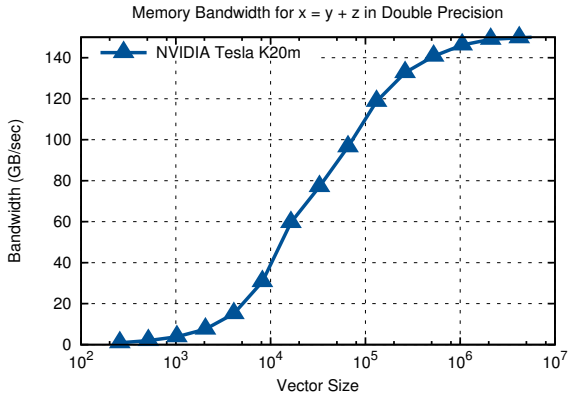
- $x = y + z$ with N elements each
- 1 FLOP per 24 byte in double precision
- Limited by memory bandwidth $\Rightarrow T_2(N) \stackrel{?}{\approx} 3 \times 8 \times N/\text{Bandwidth} + \text{Latency}$



Performance Modeling: Vector Addition

Vector Addition

- $x = y + z$ with N elements each
- 1 FLOP per 24 byte in double precision
- Limited by memory bandwidth $\Rightarrow T_2(N) \stackrel{?}{\approx} 3 \times 8 \times N/\text{Bandwidth} + \text{Latency}$



Analysis of Sparse Matvec (SpMV)

Assumptions

- No cache misses
- No waits on memory references

Notation

m	Number of matrix rows
nz	Number of nonzero matrix elements
V	Number of vectors to multiply

We can look at bandwidth needed for peak performance

$$\left(8 + \frac{2}{V}\right)\frac{m}{nz} + \frac{6}{V} \text{ byte/flop} \tag{1}$$

or achieveable performance given a bandwith BW

$$\frac{Vnz}{(8V+2)m + 6nz} BW \text{ Mflop/s} \tag{2}$$

Towards Realistic Performance Bounds for Implicit CFD Codes, Gropp, Kaushik, Keyes, and Smith.

Table of Contents

Debug vs. Optimized Builds: Choose the Right One!

PETSc's `configure` defaults to debug builds

- All development work should use a debug build!
- For maximum utility, tell compiler to generate debug sybols "`-g`" and use no optimizations or only those that do not interfere with debugging.

For performance work, must configure with `--with-debugging=no`

- Also need to ensure that compiler is generating optimized code.
- GCC defaults to no optimization, while Intel is fairly aggressive.
- Explore different levels *n* of optimization (`-On`), and consider some value-unsafe optimizations (e.g., `-ffast-math` enables several in GCC).
- Note: Compilers generally won't use advanced vector instructions by default!
  - For, e.g., KNL, need `-march=knl` with GCC, `-xMIC-AVX512` with Intel.
- Configure `--with-avx512-kernels=1` to use hand-coded AVX-512 intrinsics kernels.
- Verify correctness of your optimized executable before doing detailed performance work!

PETSc Profiling

First: Get the Math Right!

- Choose an algorithm that gives robust iteration counts
- Choose an algorithm that really converges

Profiling

- Use `-log_view` 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

- Overall 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_view` when asking performance questions on mailing list!

PETSc Profiling

```
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %F %M %L %R  %T %F %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
--- 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 73954
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 128346
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 112366
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 414021
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      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      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 220314
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 192658
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 100638
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 256792
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      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 241671
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      0
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      0
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 113626
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 13592
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      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    429
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 212399
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      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 178078
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 79715
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 237761
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 200691
```

Communication Costs

- Reductions: usually part of Krylov method, latency limited
  - `VecDot`
  - `VecMDot`
  - `VecNorm`
  - `MatAssemblyBegin`
  - Change algorithm if these are limiting factor (e.g. IBCGS, pipelined Krylov)
- 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

PETSc Profiling

Adding a Logging Event (C)

```
PetscLogEvent  USER_EVENT;
PetscClassId   classid;
PetscLogDouble user_event_flops;

PetscClassIdRegister("class name",&classid);
PetscLogEventRegister("user event",classid,&USER_EVENT);

PetscLogEventBegin(USER_EVENT,0,0,0,0);
   /* code segment to monitor */
PetscLogFlops(user_event_flops);
PetscLogEventEnd(USER_EVENT,0,0,0,0);
```

Adding a Logging Event (Python)

```
with PETSc.logEvent('Reconstruction') as recEvent:
    # All operations are timed in recEvent
    reconstruct(sol)
    # Flops are logged to recEvent
    PETSc.Log.logFlops(user_event_flops)
```

Adding a Logging Stage (C)

```c
PetscLogStage stage;

PetscLogStageRegister("name", &stage);
PetscLogStagePush(stage);

/* Code to Monitor */

PetscLogStagePop();
```

PETSc Profiling

```
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %F %M %L %R  %T %F %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
--- Event Stage 0: Main Stage

MatMult              178 1.0 7.8040e+01 1.0 2.59e+11 1.0 4.4e+02 2.0e+05 0.0e+00  33 41  6 11  0  51 89 20 24  0  6648
MatPtAP               10 1.0 2.4870e+01 1.0 5.45e+09 1.0 2.1e+02 3.1e+05 1.8e+02  10  1  3  8  1  16  2  9 18  4   429
MatPtAPSymbolic       10 1.0 1.8828e+01 1.0 0.00e+00 0.0 1.2e+02 2.7e+05 8.2e+01   8  0  2  4  0  12  0  5  9  2     0
MatPtAPNumeric        10 1.0 6.0428e+00 1.0 5.45e+09 1.0 9.4e+01 3.7e+05 1.0e+02   3  1  1  4  0   4  2  4  9  2  1767
SNESSolve              2 1.0 1.9059e+02 1.0 6.22e+11 1.0 6.6e+03 9.3e+04 3.4e+03  79 99 92 75 16 123213292168 83  6509
KSPSolve               2 1.0 1.8230e+02 1.0 6.07e+11 1.0 6.5e+03 9.1e+04 3.2e+03  76 97 89 72 15 118208285161 77  6647
PCSetUp                8 1.0 1.6138e+01 1.0 4.81e+09 1.1 1.2e+03 8.1e+04 2.5e+03   7  1 17 12 11  10  2 55 28 60   582
PCApply               46 1.0 1.2586e+02 1.0 4.43e+11 1.0 6.3e+03 8.5e+04 2.7e+03  52 70 87 65 12  81152277146 64  7022
KSPSolve_FS_0         46 1.0 1.0038e+02 1.0 3.42e+11 1.0 6.2e+03 8.2e+04 2.6e+03  42 54 86 62 12  65117273138 64  6792
(...)

--- Event Stage 1: MG Apply

MatMultMFA11         296 1.0 4.3461e+01 1.0 2.82e+11 1.0 1.2e+03 3.0e+05 0.0e+00  18 45 16 43  0  51 84 24 78  0 12995
KSPSolve             230 1.0 7.2581e+01 1.0 2.87e+11 1.0 4.5e+03 8.5e+04 2.6e+02  30 46 62 47  1  85 85 91 84100  7872
PCApply              642 1.0 1.0269e+01 1.0 1.40e+10 1.1 3.0e+03 8.7e+03 1.8e+02   4  2 42  3  1  12  4 61  6 68  2645
MGSmooth Level 0      46 1.0 7.8169e+00 1.0 1.06e+10 1.1 3.0e+03 8.3e+03 1.7e+02   3  2 41  3  1   9  3 61  5 65  2621
MGSmooth Level 1      92 1.0 2.4177e+01 1.0 3.17e+10 1.0 5.0e+02 1.2e+05 4.6e+01  10  5  7  7  0  28  9 10 13 18  2569
MGResid Level 1       46 1.0 4.3231e+00 1.0 5.77e+09 1.0 9.2e+01 1.2e+05 0.0e+00   2  1  1  1  0   5  2  2  2  0  2615
MGInterp Level 1      92 1.0 3.5063e-01 1.1 1.09e+08 1.0 9.2e+01 1.5e+04 0.0e+00   0  0  1  0  0   0  0  2  0  0   612
MGSmooth Level 2      92 1.0 4.0886e+01 1.0 2.44e+11 1.0 1.0e+03 3.0e+05 4.6e+01  17 39 14 37  0  48 73 20 66 18 11954
MGResid Level 2       46 1.0 6.8277e+00 1.0 4.39e+10 1.0 1.8e+02 3.0e+05 0.0e+00   3  7  3  7  0   8 13  4 12  0 12874
MGInterp Level 2      92 1.0 1.0898e+00 1.4 8.47e+08 1.0 9.2e+01 3.8e+04 0.0e+00   0  0  1  0  0   1  0  2  1  0  1544
(...)
------------------------------------------------------------------------------------------------------------------------
```

PETSc Profiling: PFLOTRAN Copper Leaching Benchmark

```
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %F %M %L %R  %T %F %M %L %R Mflop/s
...
--- Event Stage 5: flow Stage

RResidual             15 1.0 4.3734e-01 1.4 0.00e+00 0.0 1.9e+04 2.0e+03 0.0e+00  2  0 26  9  0  33  0 50 50  0      0
RJacobian             10 1.0 4.9278e-01 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+01  2  0  0  0  5  41  0  0  0 19      0
...
SNESSolve              5 1.0 1.0988e+00 1.1 9.58e+06 1.0 3.2e+04 2.0e+03 2.0e+02  4  0 45 15 28  90100 87 87 97    558
SNESFunctionEval      15 1.0 4.3749e-01 1.4 0.00e+00 0.0 1.9e+04 2.0e+03 0.0e+00  2  0 26  9  0  33  0 50 50  0      0
SNESJacobianEval      10 1.0 4.9285e-01 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+01  2  0  0  0  5  41  0  0  0 19      0
SNESLineSearch        10 1.0 2.6465e-01 1.0 2.87e+05 1.0 1.2e+04 2.0e+03 1.0e+01  1  0 18  6  1  22  3 34 34  5     69
KSPSetUp              20 1.0 2.4080e-05 2.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0      0
KSPSolve              10 1.0 1.0688e-01 1.0 9.25e+06 1.0 1.3e+04 2.0e+03 1.2e+02  0  0 19  6 16   9 96 36 36 56   5536
PCSetUp               20 1.0 1.9406e-02 1.6 7.03e+05 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   1  7  0  0  0   2320
PCSetUpOnBlocks       10 1.0 1.9385e-02 1.6 7.03e+05 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   1  7  0  0  0   2322
PCApply               64 1.0 4.4413e-02 1.4 3.21e+06 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   3 33  0  0  0   4628

--- Event Stage 6: transport Stage

RTResidual            37 1.0 5.6747e+00 1.3 0.00e+00 0.0 9.2e+03 1.3e+04 0.0e+00 19  0 13 28  0  22  0 35 35  0      0
RTJacobian            32 1.0 7.7537e+00 1.0 9.01e+05 1.0 0.0e+00 0.0e+00 1.3e+02 31  0  0  0 17  35  0  0  0 32      7
...
SNESSolve              5 1.0 2.0480e+01 1.0 6.45e+09 1.0 2.5e+04 1.3e+04 4.0e+02 84100 36 75 54  95100 95 95 98 20140
SNESFunctionEval      37 1.0 5.6751e+00 1.3 0.00e+00 0.0 9.2e+03 1.3e+04 0.0e+00 19  0 13 28  0  22  0 35 35  0      0
SNESJacobianEval      32 1.0 7.7540e+00 1.0 9.01e+05 1.0 0.0e+00 0.0e+00 1.3e+02 31  0  0  0 17  35  0  0  0 32      7
SNESLineSearch        32 1.0 4.9337e+00 1.0 1.10e+07 1.0 7.9e+03 1.3e+04 3.2e+01 20  0 11 24  4  23  0 30 30  8    143
KSPSetUp              64 1.0 1.2088e-04 2.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0      0
KSPSolve              32 1.0 7.1024e+00 1.0 6.43e+09 1.0 1.6e+04 1.3e+04 1.6e+02 29100 23 48 22  33100 60 60 40 57963
PCSetUp               64 1.0 5.2400e+00 1.4 5.19e+09 1.0 0.0e+00 0.0e+00 0.0e+00 16 80  0  0  0  18 80  0  0  0 63338
PCSetUpOnBlocks       32 1.0 5.2395e+00 1.4 5.19e+09 1.0 0.0e+00 0.0e+00 0.0e+00 16 80  0  0  0  18 80  0  0  0 63344
PCApply               96 1.0 2.7751e-01 1.0 6.94e+08 1.0 0.0e+00 0.0e+00 0.0e+00  3 11  0  0  0   3 11  0  0  0 61020
```

Table of Contents

Available GPU (or “Accelerator”) Back-Ends

CUDA

- CUDA-support through CUDA/CUSPARSE
- `-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 (CUSPARSE)

-

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

- Customization:

```
  --with-cudac=/path/to/cuda/bin/nvcc
  --with-cuda-arch=sm_60
```

ViennaCL

-

```
./configure [..] --download-viennacl
```

- Optional: CUDA/OpenCL/OpenMP

```
  --with-cuda=1
```

```
  --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
  PetscCUDAFlag valid_GPU_array;   // flag
  void          *spptr;            // device buffer
};
```

Possible Flag States

```
  typedef enum {PETSC_CUDA_UNALLOCATED,
                PETSC_CUDA_GPU,
                PETSC_CUDA_CPU,
                PETSC_CUDA_BOTH} PetscCUDAFlag;
```

How Does It Work?

Fallback-Operations on Host

- Data becomes valid on host (`PETSC_CUDA_CPU`)

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

Accelerated Operations on Device

- Data becomes valid on device (`PETSC_CUDA_GPU`)

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

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 viennacl -mat_type aijviennacl
    -pc_type ilu -m 200 -n 200 -log_summary
```

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

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

-

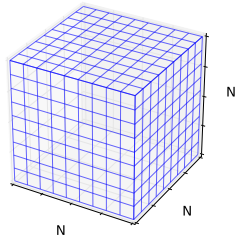
```
$ ./ex12 -vec_type viennacl -mat_type aijviennacl
    -pc_type none -m 200 -n 200 -log_summary
```

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

Pitfalls

Pitfall 1: GPUs are too fast for PCI-Express

- Latest GPU peaks: 720 GB/sec from GPU-RAM, 16 GB/sec for PCI-Express
- 40x imbalance (!)



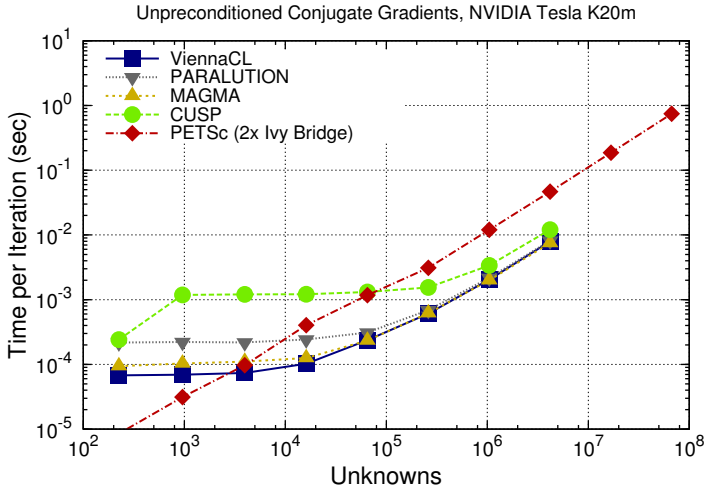
Compute vs. Communication

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

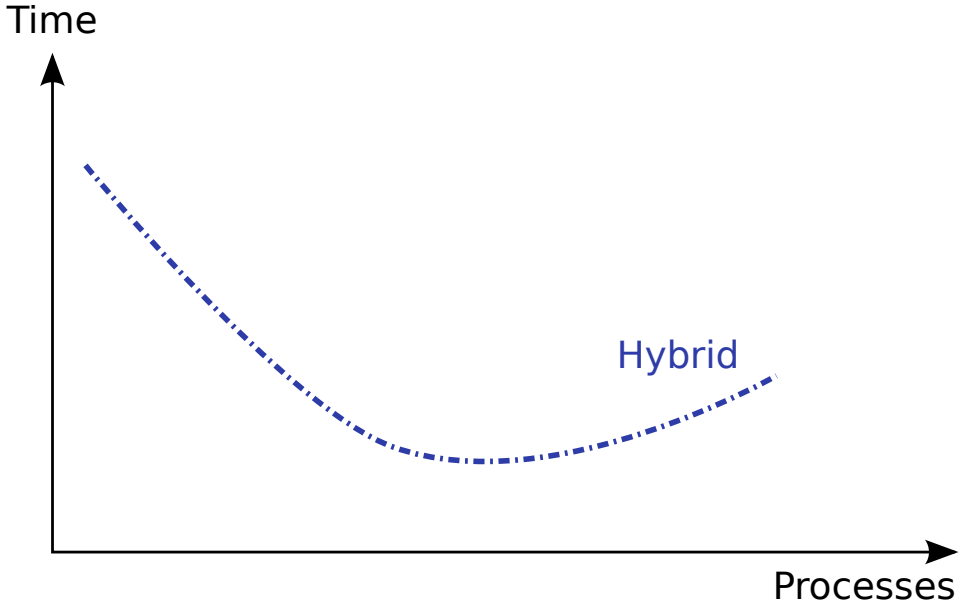
Pitfalls

Pitfall 2: Wrong Data Sizes

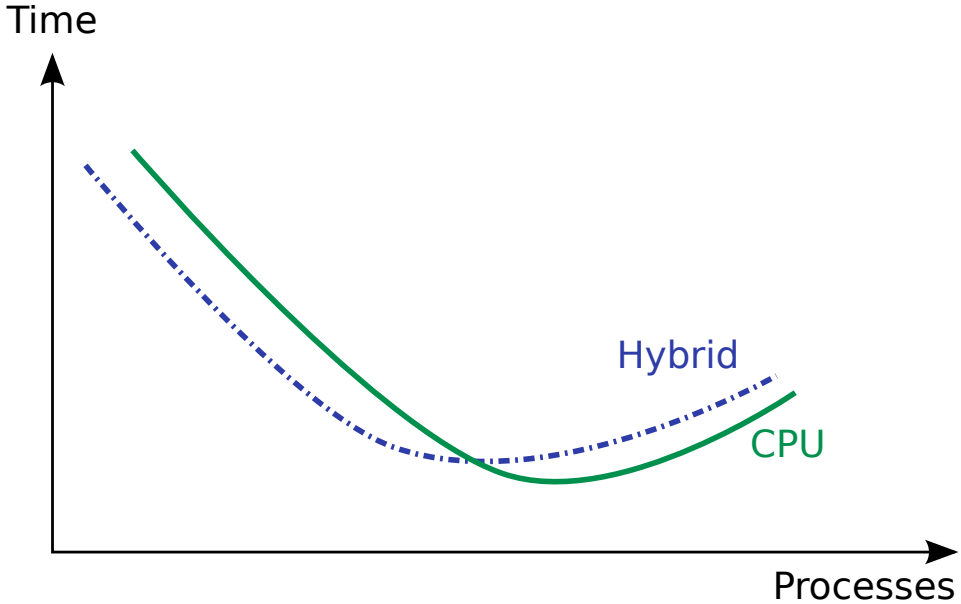
- Data too small: Kernel launch latencies dominate
- Data too big: Out of memory



Strong Scaling Implications



Strong Scaling Implications



Pitfall 3: Composability of GPU codes

- How to pass GPU pointers through library boundaries efficiently?
- High-level interfaces tend to be polluted by low-level details

Many Non-Trivial PETSc Operations

do NOT benefit from modern high-end GPUs

in a substantial way!
(OpenPower systems can be exceptions)

Current GPU-Functionality in PETSc

	CUDA/CUSPARSE	**ViennaCL**
Programming Model	CUDA	CUDA/OpenCL/OpenMP
Operations	Vector, MatMult	Vector, MatMult
Matrix Formats	CSR, ELL, HYB	CSR
Preconditioners	ILU0	SA/Agg-AMG, Par-ILU0
MPI-related	Scatter	-

Additional Functionality

- OpenCL residual evaluation for PetscFE
- GPU support for SuperLU-dist
- GPU support for SuiteSparse

Current: PETSc + ViennaCL

Previous Use of ViennaCL in PETSc

-

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

- Executes on OpenCL device

New Use of ViennaCL in PETSc

-

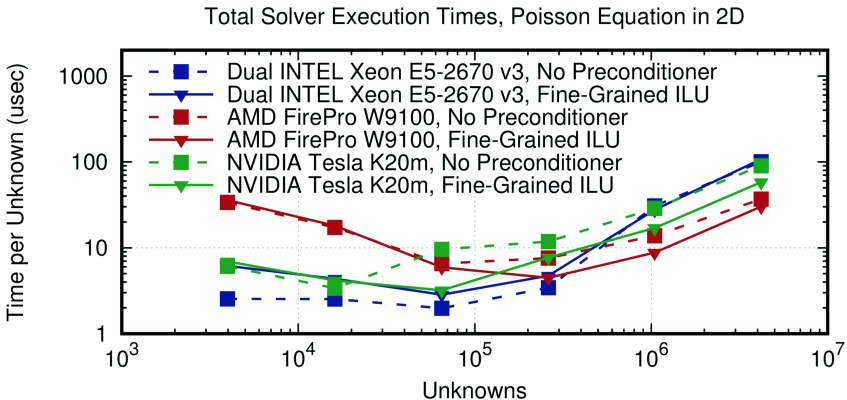
```
$ ./ex12 -vec_type viennacl -mat_type aijviennacl
         -viennacl_backend openmp ...
```

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

Chow-Patel Fine-Grained ILU in ViennaCL

```
$ ./ex12 -vec_type viennacl -mat_type aijviennacl -pc_type chowiluviennacl
-m $M -n $N -log_summary
```



GPU Support in GAMG Algebraic Multigrid

New! Native PETSc GAMG algebraic multigrid support

- Specify `aijcusparse` or `aijviennacl` matrix types, then numerical setup and and solve phases (Chebyshev/Jacobi smoothing, coarse grid restriction and interpolation) will run on GPU.
  - E.g., `mpirun -n $N ./ex5 -da_refine 9 -pc_type gamg -pc_mg_levels $NLEVELS -mg_levels_pc_type jacobi -dm_mat_type aijviennacl -dm_vec_type viennacl`
- Similar approach used with AIJMKL and AIJSELL matrix types to use MKL or sliced-ELLPACK back-ends optimized for manycore/SIMD CPUs during solve phase.
  - E.g., `mpirun -n $N ./ex5 -da_refine 9 -pc_type gamg -pc_mg_levels $NLEVELS -mg_levels_pc_type jacobi -mat_seqaij_type seqaijsell`

Performance of GAMG with this usage model is largely unexplored.
We welcome feedback to help us improve its performance!

Ongoing and Future GPU Work

- Continued improvement of GPU-accelerated native multigrid preconditioner GAMG
- Plugin for NVIDIA AmgX multigrid library
- Plugin for RAPtor reduced-communication AMG library
- Efficient data exchange across MPI ranks for ViennaCL
- More examples to illustrate best practices

Not Covered Today: Reducing/Hiding Communication at Scale

This session focused on on-node optimizations, but reducing/hiding communication costs is equally important for performance on leadership-class systems. To this end, PETSc supports

Krylov methods that minimize or hide communication costs:

- Improved BiCGStab: KSPIBCGS
- Pipelined methods: KSPGROPPCG, KSPPIPECG, KSPPIPECGRR, KSPPIPELCG, KSPPIPEFGMRES, KSPPIPEFCG, KSPPGMRES, KSPPIPEBCGS, KSPPIPECR, KSPPIPEGCR

Extreme-scale multigrid:

- PCTELESCOPE
- See May et al. 2016, Extreme-Scale Multigrid Components in PETSc for details.

Vector scatter/gather with MPI-3 shared memory windows:

- `mpi3` and `mpi3node` options for VecScatterType
- `VECNODE` vector type resides in shared memory window

(Above, magenta denotes web links.)

Table of Contents

Exercise 0: Graph the topology of your system

Use `lstopo` to graph the topology of your system.

If `lstopo` is not present on your system, install `hwloc` via your package manager (e.g., `brew install hwloc` or `apt-get install hwloc`), or have PETSc `configure` download via `--download-hwloc`.

Can generate an ASCII summary by simply executing `lstopo` (or possibly `lstopo-no-graphics`, depending on how `hwloc` has been built), or get a fancier ASCII art or PNG version by doing `lstopo topo.txt` or `lstopo.png`

Exercise 1: Measure sustainable memory bandwidth via STREAM Triad

The STREAM Triad benchmark (see `https://www.cs.virginia.edu/stream/`) can provide a good idea of the possible speedup that can be obtained with a memory bandwidth-bound code on a given machine.

Execute the STREAM Triad benchmark on your machine.
Set `PETSC_ARCH` and then, in `$PETSC_DIR`, do `make streams`.

If your machine has multiple sockets (or perhaps just a complicated cache hierarchy), your results may vary depending on how MPI processes are assigned to cores. (See `https://www.mcs.anl.gov/petsc/documentation/faq.html#computers` for some notes on doing this with MPICH and OpenMP.) Example:

```
$ make streams MPI_BINDING="--bind-to core --map-by socket"
```

Exercise 2: Examine scaling of SNES ex19 with default solvers

Run SNES ex19 (nonlinear driven cavity problem) with default PETSc solvers:

```
$ cd $PETSC_DIR/src/snes/examples/tutorials; make ex19
$ mpirun -np $NP ./ex19 -da_refine $NREFINE -snes_monitor -snes_view -log_view
```

The problem size is controlled by specifying the number of times to refine the 4×4 DMDA. (Total number of degrees of freedom is $4 \times (3 \times 2^{nrefine} + 1)^2$). Work with sizes such that that runs do not complete instantly; `-da_refine 5` might be a good starting point.

How does the strong scaling (varying NP for fixed problem size) of overall execution time and events such as MatMult compare to the STREAM triad scaling? You may also want to try static scaling (varying problem size for fixed NP).

Using the block-oriented format BAIJ reduces the size of the *j* index array by a factor of the block size (4 for this example.) Try running with BAIJ matrices (`-dm_mat_type baij`) to see how the corresponding reduction in the memory bandwidth requirements affect MatMult performance.

Exercises 3–5: Run SNES ex19 with a variety of solvers on CPU and GPU

We can use the cuSPARSE/CUDA backend to run on NVIDIA GPUs:

```
$ mpirun -np $NP ./ex19 -da_refine $NREFINE -snes_monitor -snes_view -log_view
-dm_mat_type aijcusparse -dm_vec_type cuda
```

or use the ViennaCL backend to run on any type of GPU (`-dm_mat_type aijviennacl -dm_vec_type viennacl`).

Let's run SNES ex19 several ways, comparing GPU and CPU.

Aside: It may be helpful to use the nested logging capability of PETSc to understand where GPU↔CPU transfers are incurred. To do so, run with `-log_view :filename.xml:ascii_xml`. The XML file may be viewable directly in your web browser, or (my preference), do

```
$ ${PETSC_DIR}/lib/petsc/bin/petsc-performance-view filename.xml
```

to open a nicely-rendered version in your web browser.

Exercise 3: Run SNES ex19 on CPU and GPU with a very simple preconditioner, Jacobi

Use `-pc_type jacobi` with GPU and non-GPU cases, trying some different values for -da_refine. Example using cuSPARSE:

```
$ mpirun -np $NP ./ex19 -da_refine $NREFINE -snes_monitor -snes_view -log_view
-dm_mat_type aijcusparse -dm_vec_type cuda -pc_type jacobi
```

The GPU case is probably faster for most cases. What does -log_view tell us about why?

Try both the cuSPARSE and ViennaCL back-ends, if you have them both available.

Exercise 4: Run SNES ex19 with ILU on CPU and GPU

Let's run with a more complicated preconditioner: Incomplete LU factorization (ILU)

Run with `-pc_type ilu`, which will do ILU in the single-process case, or with `-pc_type bjacobi` to use block-Jacobi with ILU applied on each block, or subdomain, in the parallel case.

Compared to the Jacobi case, the CPU likely becomes more competitive with the GPU.
Can you find a -da_refine value where there is a change in which is faster?
Or identify such a point in a strong-scaling scenario?

Bonus: Can also try using ViennaCL for `-pc_type chowiluviennacl` on the GPU, which is likely faster overall but probably requires more Krylov iterations.

Exercise 5: Run SNES ex19 with multigrid on CPU and GPU

Now use what we really **ought** to be using: -pc_type mg. Example (using Jacobi smoothers):

```
$ mpirun -np $NP ./ex19 -da_refine $NREFINE -snes_monitor -snes_view -log_view
-dm_mat_type aijcusparse -dm_vec_type cuda -pc_type mg -mg_levels_pc_type jacobi
-pc_mg_levels $NLEVELS
```

`NLEVELS` is the number of multigrid levels to employ; you may need to manually set this to something less than `$NREFINE`.

The CPU may become more competitive because the small, coarse grid problems are not well suited to GPUs.

Is the CPU or GPU consistently better on your system, or is there a crossover point (in strong or static scaling scenarios) in the behavior?
Can you improve GPU performance by limiting the number of levels in the multigrid hierarchy?
Does the CPU or GPU benefit more from using a different smoother, say, SOR?
(`-mg_levels_pc_type sor`)

We can play several other games. By not always updating the Jacobian, we get worse convergence and will spend more time in events such as MatMult, but the trade-off may be advantageous if MatMult is fast. To try this, run with `-snes_lag_jacobian <lag>`.