ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

A Brief Compendium of GPU-accelerated Numerical Libraries

CSCS/USI Summer School, Riva San Vitale, Switzerland, 25.07.2018

William Sawyer, Karl Rupp, Michael Heroux, Stan Tomov, Christian Trott, H. Carter Edwards, many others

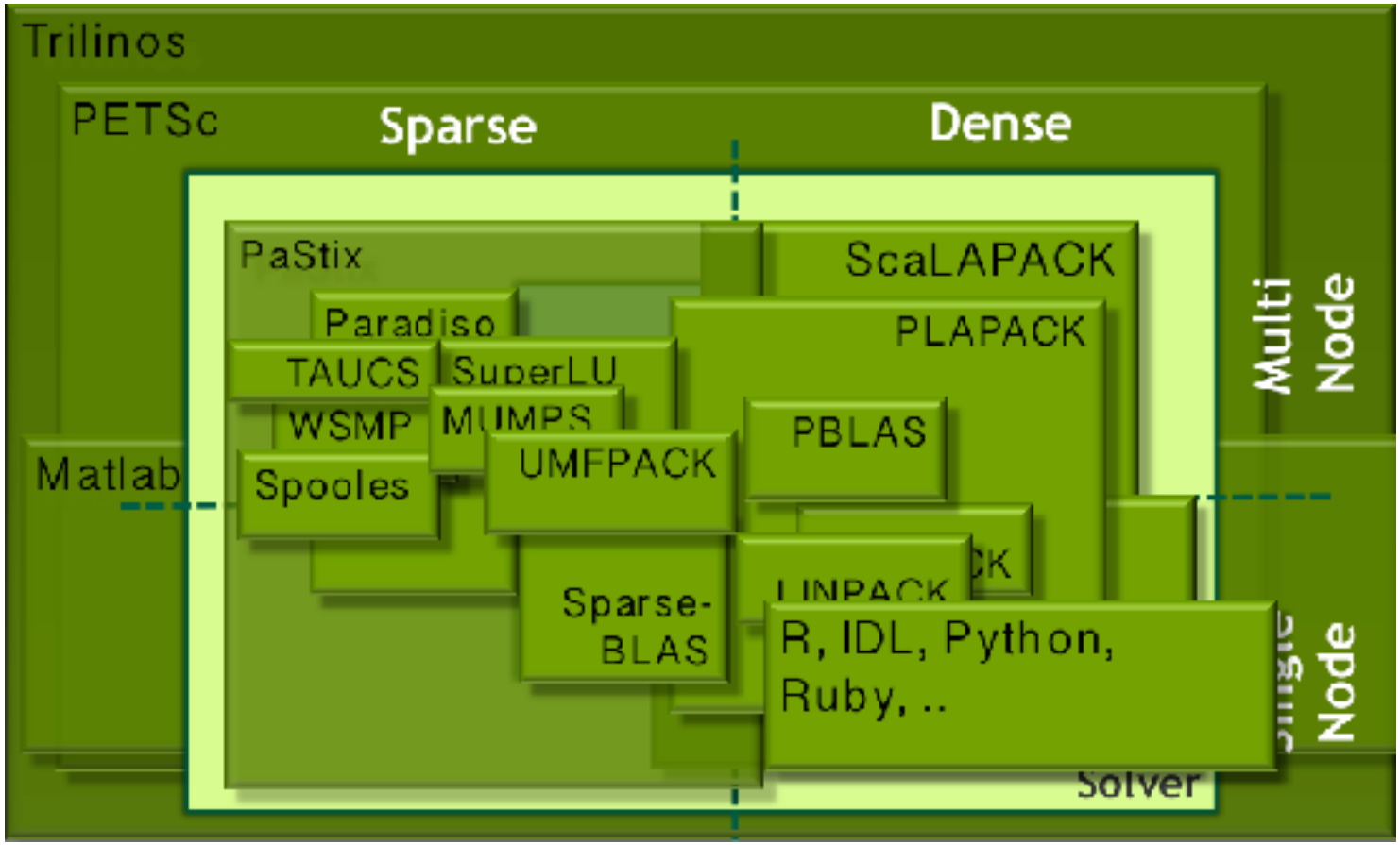
Tutorial Summary

- Introduction and library overview
  - Typical numerical and semi-numerical problems
  - Survey of some gpu-enabled libraries:
    - NVIDIA libraries: cuBLAS, cuFFT, cuRAND, cuSPARSE, AmgX
    - MAGMA
    - Thrust (+CuSP)
    - ViennaCL, Paralution
    - PETSc
    - Trilinos/Kokkos
- `https://github.com/eth-cscs/SummerSchool2018`

Objectives of this tutorial

- Awareness of the available third-party libraries
- Realization that it is not necessary for users to "recreate the wheel"
- *Subjective appraisal of the libraries to bet on*

Extensive CPU Library Ecosystem



Problems our users might like to solve

- Partial differential equations
- Dense linear algebra
- Sparse systems of linear equations
- Preconditioning of large systems
- Eigenvalue / singular value decompositions of sparse/dense matrices
- Partitioning large graphs
- Non-linear systems and optimization
- ...

For single node GPUs

Solid

Limited

Under development

NVIDIA's CUDA Libraries

- cuBLAS: simple linear algebra on vectors and matrices

  1: vectors: add, dot products, scaling, norms, rotations, ..

  2: matrix-vector: triangular/full/hermitian mat x vec, ...

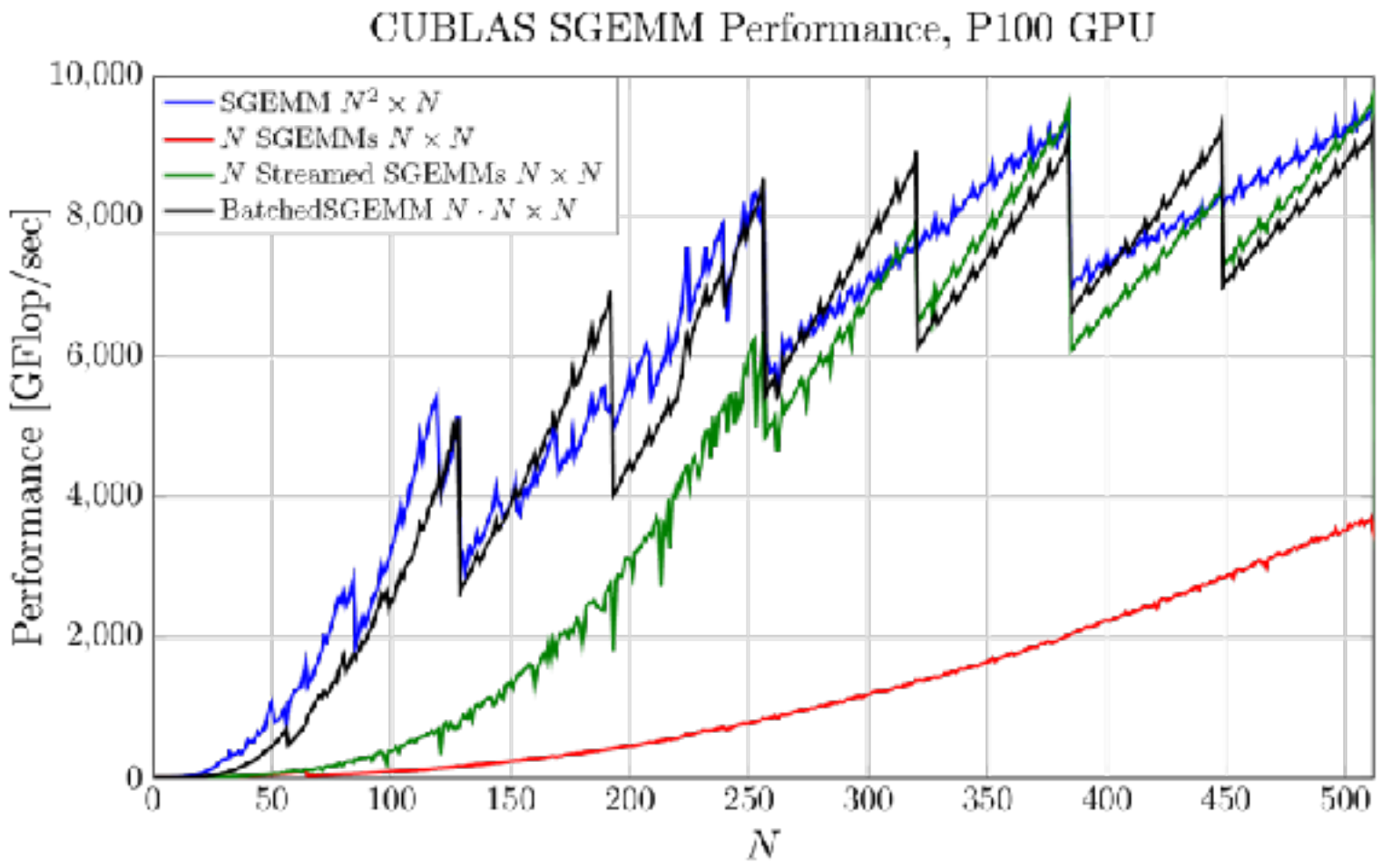
  3: matrix-matrix: products, multiple rank updates, ...
- cuSPARSE: Sparse linear algebra
  - ➡ matrix creation, various sparse matrix formats
  - ➡ indexed vector operations
  - ➡ sparse mat x vec, mat x mat operations
- cuFFT: fast Fourier transforms
- cuRAND: random number generators

➡ *offers single-GPU core support for user applications*

Example: simple SAXPY (vector add)

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void**)&d_y);

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1); // CPU -> GPU
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1); // CPU -> GPU

// Perform SAXPY: d_y[]=a*d_x[]+d_y[]

cublasSaxpy(N, 2.0, d_x, 1, d_y, 1)

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1); // GPU -> CPU

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

CUBLAS SGEMM Performance



Credit: http://news.developer.nvidia.com

cuSPARSE sparse linear algebra

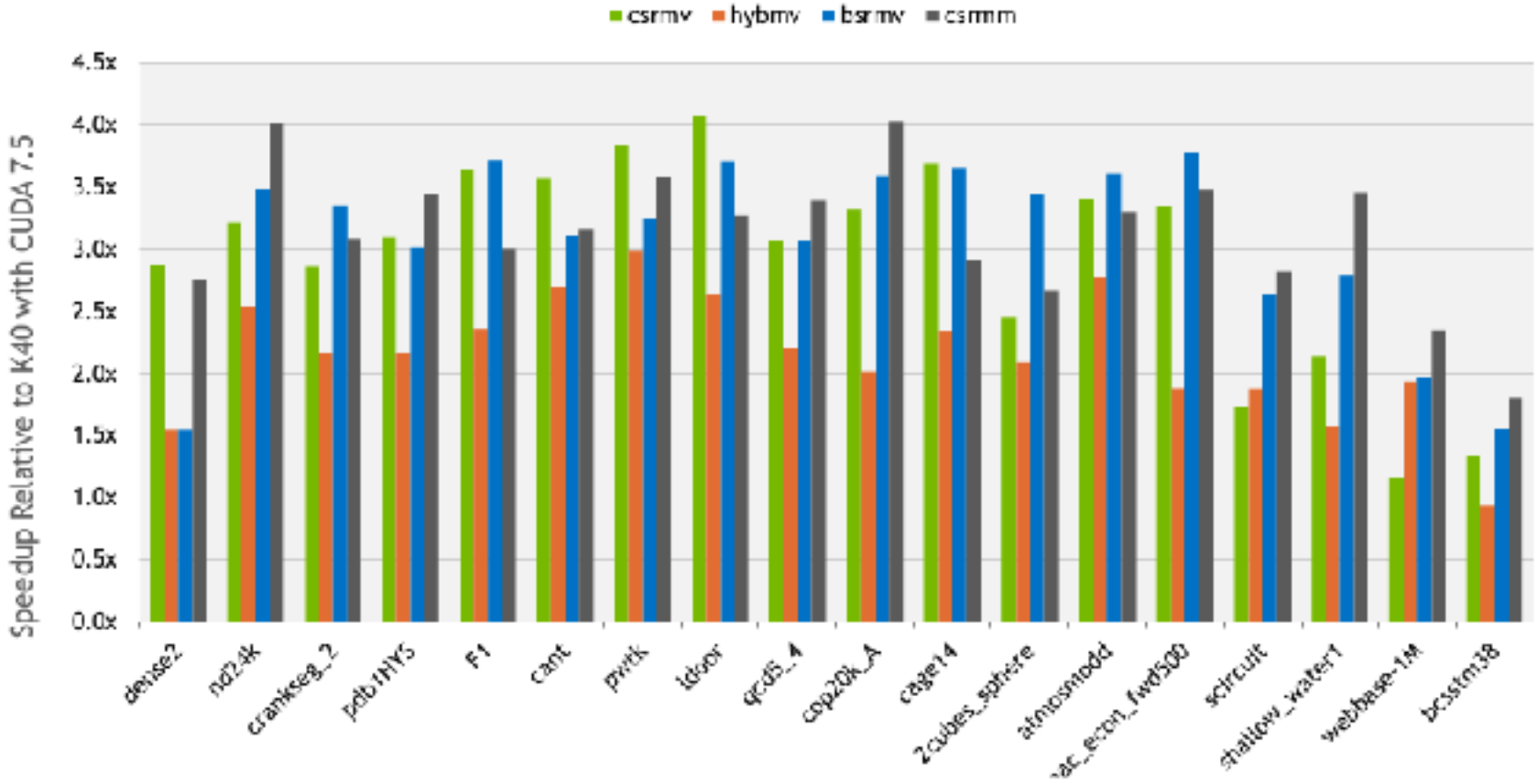
- Supports sparse (indexed) and dense formats for vectors, BLAS 1 operations for indexed vectors
- Various sparse formats for matrices (CSR, COO, ELL, hybrid ELL+COO, BSR, BSRx, ...)
- Matrix format conversions
- Sparse matrix-vector, matrix-matrix product
- Simplistic preconditioners

```
mkl_dcsrmv(transa, m, k,
alpha, descr, val, indx,
pntrb, pntre, x, beta, y);
```

```
err = cusparseDcsrmv(hdl,
transa, m, k, nnz, alpha,
desrc, val, indx, col,
x, beta, y);
```

cuSPARSE performance

CUDA8: Mat-vec and Mat-Mat, various formats, P100 vs K40



https://developer.nvidia.com/cuSPARSE

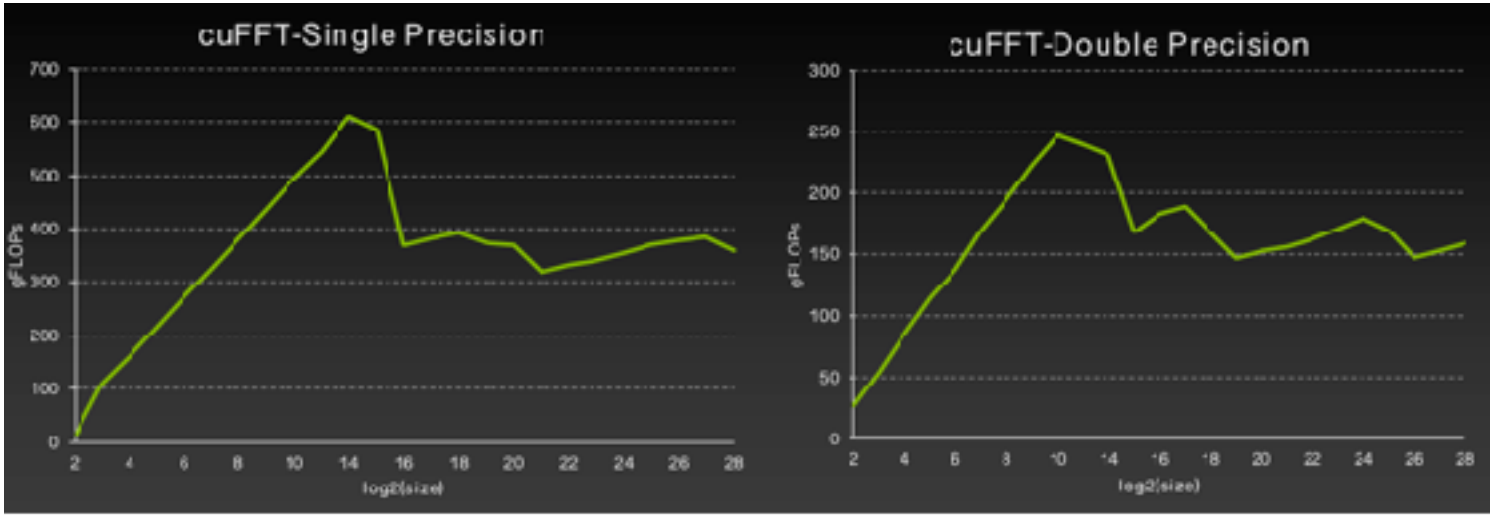
CUFFT Fast Fourier Transforms

Interface modeled after FFTW

```
fftw_plan PlanA;
fftw_plan_dft_2d(N, M, &PlanA,
data, data, FFT_FORWARD)
fftw_execute_dft(PlanA, data,
data);
```

```
cufftPlan2d PlanA;
cufftCreatePlan(N, M, &PlanA,
CUFFT_C2C);
cufftExecC2C(PlanA, d_data,
d_data, CUFFT_FORWARD);
```

1-D FFT perf. (NVIDIA)



cuRAND random numbers

- Large suite of high-quality random number generators
  - XORWOW, MRG323ka, MTGP32, scrambled Sobol
  - uniform, normal, log-normal
  - single and double precision
- Two APIs
  - Called from CPU: for large batches of random numbers

```
#include "curand.h"
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
curandGenerateUniform(gen, d_data, n);
```

  - Called from GPU: random numbers within kernels

```
#include "curand_kernel.h"
__global__ void generate_kernel(curandState *state) {
  int id = threadIdx.x + blockIdx.x * 64;
    x = curand(&state[id]);
}
```

MAGMA: Matrix Algebra on GPU and Multicore Architecture

- *MAGMA:* Matrix Algebra on GPU and Multicore Architectures
- Current version: MAGMA 2.4.0
- Essentially LAPACK functionality on hybrid platforms
- CUDA + limited availability for Xeon Phi, OpenCL
- Developers/collaborators: UTK, UC Berkeley, UC Denver, INRIA, KAUST, others (including ETH)
- Distributed memory GPU platforms — SLATE

Typical dense linear algebra algorithms

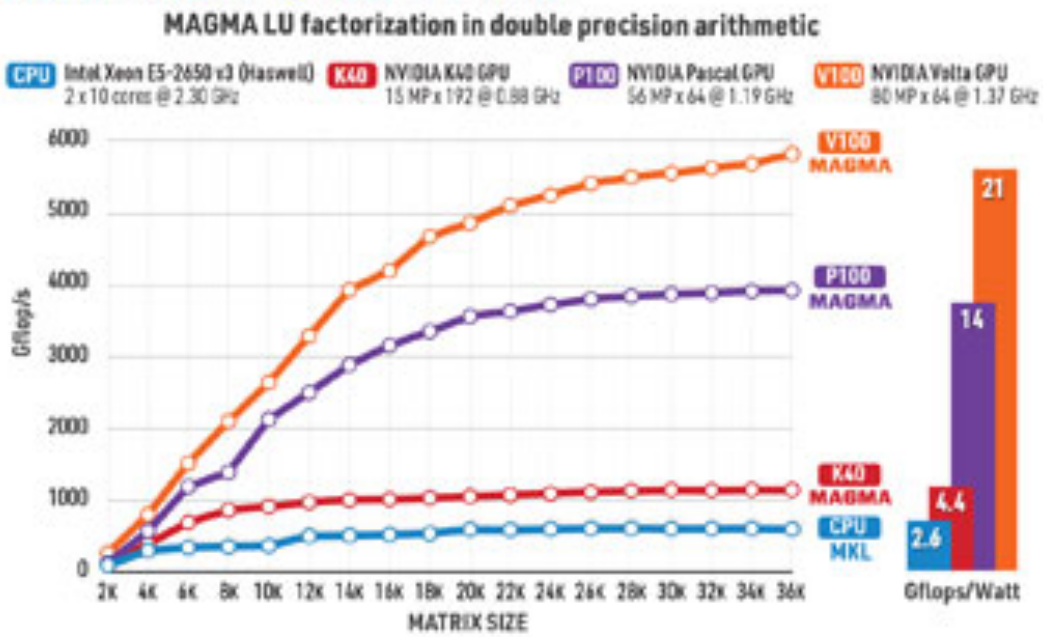
- Cholesky factorization: $A = A^T = LL^T \qquad i < j \Rightarrow L_{i,j} = 0$
- QR factorization: $A = QR \qquad Q^T Q = I \qquad i > j \Rightarrow R_{i,j} = 0$
- LU factorization: $A = P^T LU \qquad P^T P = I$
- Forward/back-substitution: $Ax = y \Rightarrow LUx = y \Rightarrow w = L^{-1}y \Rightarrow x = R^{-1}w$
- Eigenvalue decomposition: $Ax = \lambda x \Rightarrow A = QDQ^T$
- Generalized eigen-problem: $Ax = \lambda Bx$
- Singular value decomposition: $A = U\Sigma V^T \quad U^T U = I \quad V^T V = I$

MAGMA Overview (SC17 Poster)

HYBRID ALGORITHMS

MAGMA uses a hybridization methodology where algorithms of interest are split into tasks of varying granularity and their execution scheduled over the available hardware components. Scheduling can be static or dynamic. In either case, small non-parallelizable tasks, often on the critical path, are scheduled on the CPU, and larger more parallelizable ones, often Level 3 BLAS, are scheduled on the GPU.

PERFORMANCE & ENERGY EFFICIENCY



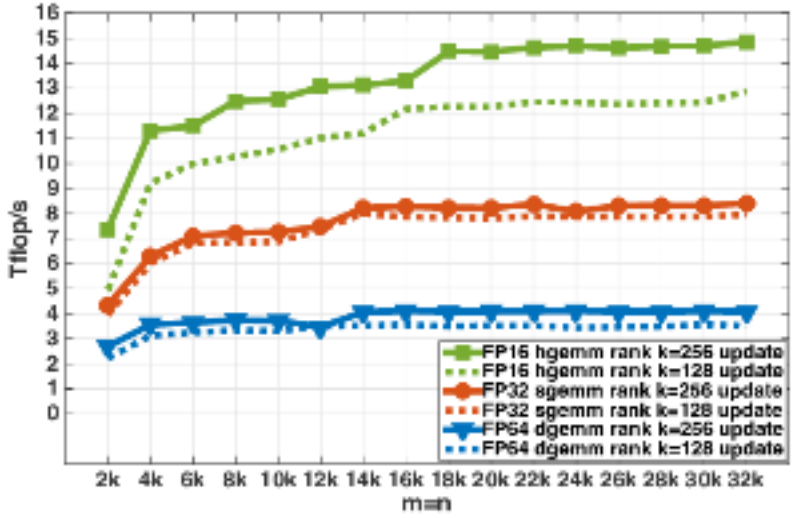
FEATURES AND SUPPORT

- **MAGMA 2.3** FOR CUDA
- **clMAGMA 1.4** FOR OpenCL
- **MAGMA MIC 1.4** FOR Intel Xeon Phi

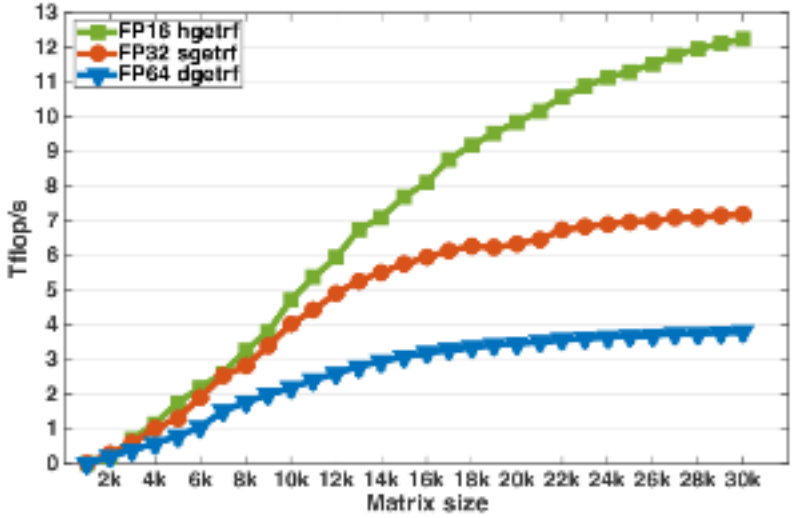
CUDA	OpenCL	Intel Xeon Phi	
●	●	●	Linear system solvers
●	●	●	Eigenvalue problem solvers
●	●		Auxiliary BLAS
●			Batched LA
●		●	Sparse LA
●	●	●	CPU/GPU Interface
●	●	●	Multiple precision support
●			Non-GPU-resident factorizations
●	●	●	Multicore and multi-GPU support
● NEW			MAGMA Analytics/DNN
●	●	●	LAPACK testing
●	●	●	Linux
●	●		Windows
●	●		Mac OS

MAGMA Performance

P100 Performance of rank-k update xGEMM (left) and tridiagonalization xGETRF (right)



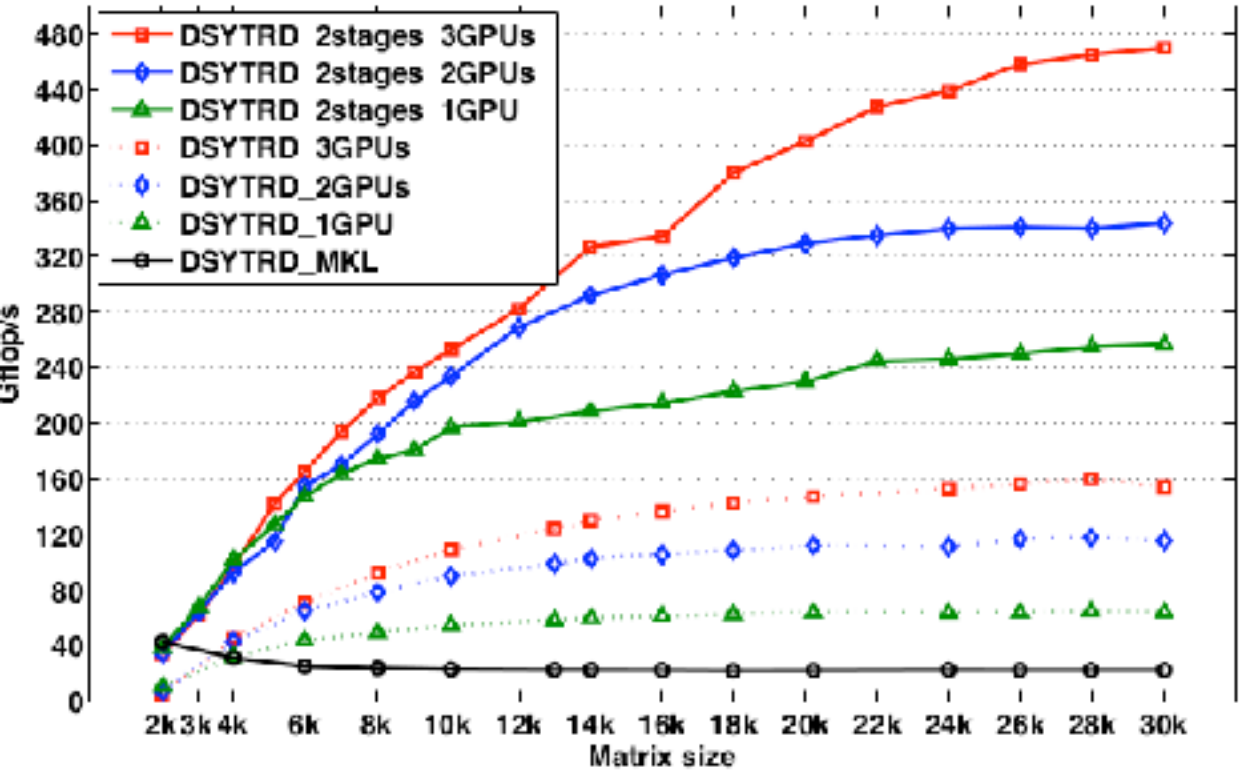
(a) Performance of the rank-k update (Xgemm function) used in Xgetrf.



(b) Performance of the Xgetrf routine.

Credit: Haidar, et al., ScalA17

MAGMA Generalized EVP $Ax = \lambda Bx$



A. Haidar, S. Tomov, J. Dongarra, T. Schulthess, and R. Solca, *A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks*, ICL Technical report, 03/2012.

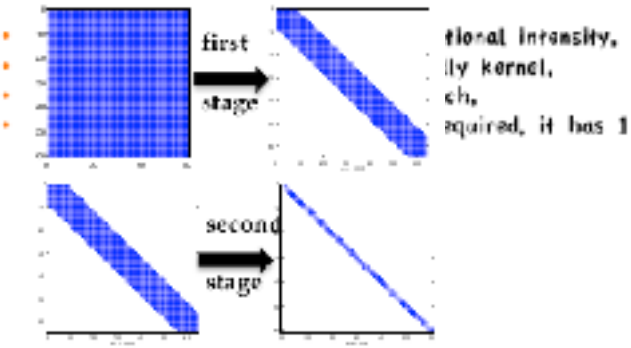
MKL Tridiagonalization

- Too many Blas-2 op.
- Relies on panel factorization,
- →Bulk sync phases,
- →Memory bound algorithm.

GPU 1-Stage

- Blas-2 GEMV moved to the GPU.
- Accelerate the algorithm by doing all BLAS-3
- →Bulk sync phases.
- →Memory bound algorithm.

GPU 2-Stage



MAGMA: conclusions

- Subset of LAPACK dense linear algebra functionality on hybrid multicore platforms (NVIDIA, Intel Xeon Phi, …)
- Some support for sparse linear algebra
- *High quality, dependable implementation*
- Proven performance on single-node each with 1 or more GPUs
- *Still no distributed memory implementation*, efforts are underway, e.g., `https://icl.utk.edu/slate` for a ScaLAPACK replacement
- *Possible funding limitations…*

Thrust: Standard Template Library for GPUs

- A library of parallel algorithms resembling the C++ STL
- Allows easy access/manipulation of vectors on both host (CPU) and device (GPU); based on data *iterators*
- Defines straightforward vector data operators, e.g., :
  - ✳ initialize vectors
  - ✳ exchange existing values
  - ✳ copy one to another
  - ✳ transform with an operator
  - ✳ perform reductions (e.g., one-dimensional to scalar)
  - ✳ sorting and other operators

Thrust: typical operations

- Declare vectors on host or device

```
thrust::host_vector<int> H(4);
thrust::device_vector<int> D = H;
thrust::device_vector<int> Z(4, 1);  // All ones
```

- Initialize vectors, use iterators

```
thrust::sequence(H.begin(), H.end()); // H = (0,1,2,3)
thrust::fill(D.begin(), D.end(), 2);  // Fill with twos
```

- Transform vectors

```
thrust::transform(D.begin(), D.end(), Z.begin(), thrust::negate<int>());
thrust::replace(H.begin(), H.end(), 2, -2);
```

- Perform a reduction

```
int sum = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());
```

- Sort vector

```
thrust::sort(H, H + 4);
```

Thrust: simple manipulations

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <iostream>
int main(void)
{
  // initialize all ten integers of a device_vector to 1
  thrust::device_vector<int> D(10, 1);
  // initialize a host_vector with the first five elements of D
  thrust::host_vector<int> H(D.begin(), D.begin() + 5);
  // set the first seven elements of a vector to 9
  thrust::fill(D.begin(), D.begin() + 7, 9);
  // copy all of H back to the beginning of D
  thrust::copy(H.begin(), H.end(), D.begin());
  // print D
  for(int i = 0; i < D.size(); i++)
  {
    std::cout << "D[" << i << "] = " << D[i] << std::endl;
  }
  return 0;
}
```

Assignment: what values are printed?

Thrust: advanced operators

- Scan operations (parse and alter vector)

```
#include <thrust/scan.h>
int data[6] = {1, 0, 2, 2, 1, 3};
thrust::inclusive_scan(data, data + 6, data); // data now {1, 1, 3, 5, 6, 9}
thrust::exclusive_scan(data, data + 6, data); // data now {0, 1, 2, 5, 10, 16}
```

- Iterator transformation -- bind an operator to an iterator.

```
#include <thrust/transform_iterator.h>
thrust::device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;
thrust::device_vector<int>::iterator first = thrust::make_transform_iterator(vec.begin(), negate<int>());
thrust::device_vector<int>::iterator last  = thrust::make_transform_iterator(vec.end(), negate<int>());
// first[0] returns -10, first[1] returns -20, first[2] returns -30
thrust::reduce(first, last);   // returns -60 (i.e. -10 + -20 + -30)
```

- Zip operator: turns multiple input arguments into tuples (more later)

Thrust: conclusions

- Attempt to extend C++ STL functionality for host/device
- Based on CUDA, thus bound to NVIDIA GPUs
- Uses template meta-programming to find correct implementation at compile time
- Is an community, open-source project, *but appears to have long-term approval from NVIDIA* (bundled in SDK releases)
- *Development intended by NVIDIA to be demand-driven by community; in reality not the case*

Linear solvers

Goal: Support the solution of linear systems,

$$Ax=b,$$

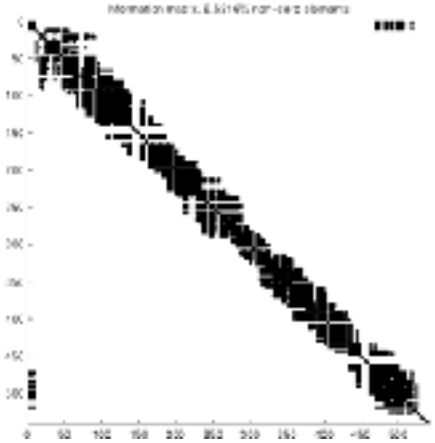
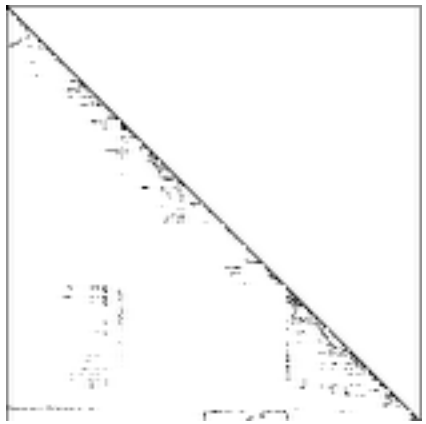
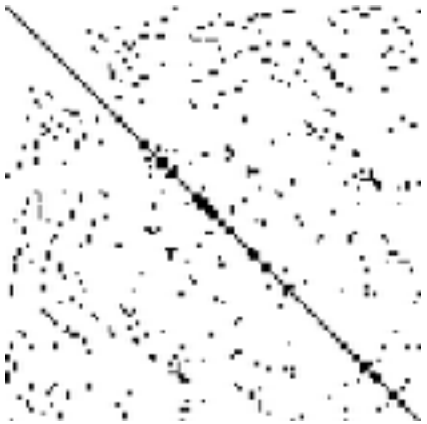
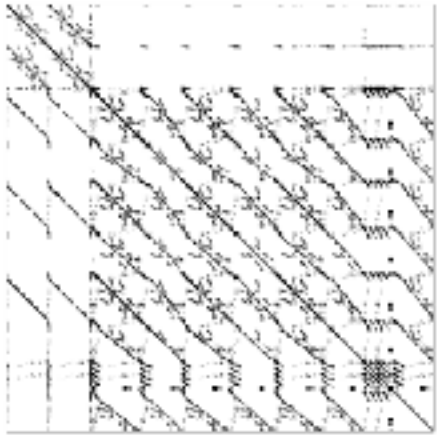
particularly for *sparse*, parallel problems, e.g., arising from PDE-based models.

User provides:

- *A (matrix or operator)*
- *b (right-hand side)*
- *u (initial guess)*

Libraries for Sparse Linear Algebra

- MAGMA limited to $m \times n$ matrices with $m,n = O(10^4)$
- Sparse matrices typically contain at least 90% zeros
- Number of non-zero (nz) elements, large: $nz = O(10^7)$
- Matrix market http://math.nist.gov/MatrixMarket/

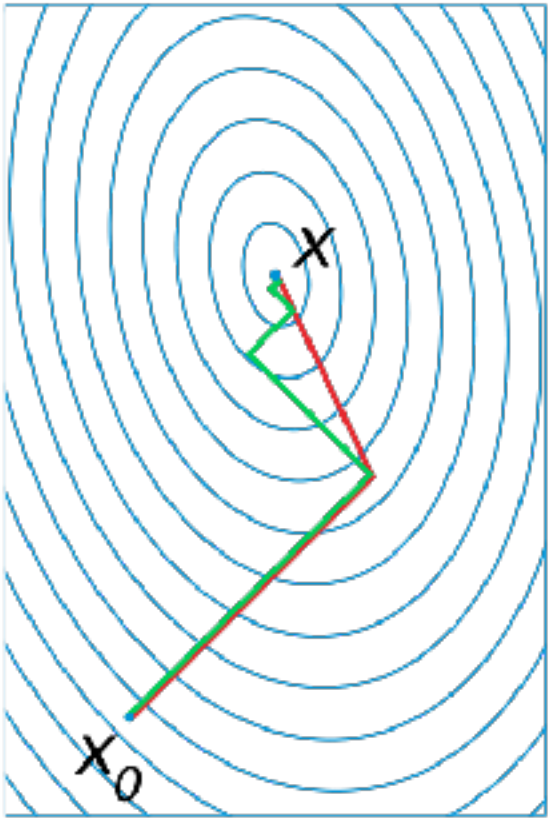





Linear System Solution: $Ax = b$

Two basic techniques:

- Direct methods, i.e. factorize matrix
  - good for multiple right hand sides
  - tend to be more robust
- Iterative methods
  - good if matrix known via operators
  - possibilities for approximate solutions

Hestenes/Stiefel, 1952: Conjugate Gradient



$$k = 0; \quad x_0 = 0; \quad r_0 = 0$$

$$while \quad r_k \neq 0 \quad \{$$

$$k = k + 1$$

$$if\,(k = 0) \Rightarrow p_1 = r_0$$

$$if\,(k > 0) \Rightarrow \beta_k = r_{k-1}^T r_{k-1} / r_{k-2}^T r_{k-2}; \quad p_k = r_{k-1} + \beta_k p_{k-1}$$

$$\alpha_k = r_{k-1}^T r_{k-1} / p_k^T A p_k$$

$$x_k = x_{k-1} + \alpha_k p_k$$

$$r_k = r_{k-1} - \alpha_k A p_k$$

$$\}$$

1980's: led to a wide class of iterative *Krylov subspace methods*

Preconditioners: KSM alone insufficient!

- CG method initially ignored due to slow convergence
  - Theoretical convergence after 2*n steps, but n is huge
  - Convergence rate related to ratio largest/smallest eigenvalue
- Easier problem: preconditioner $M \approx A$ $\quad Ax = b \Rightarrow M^{-1}Ax = M^{-1}b$
  - Find an approximation for A where $M^{-1}x$ is 'easily' calculated
  - Possibilities:
    - Approximate inverse known through physical description
    - Incomplete LU decomposition
    - Sparse approximative inverse (assume inverse also sparse)
    - Multilevel (multigrid) preconditioners
    - More…

CUSP: Sparse Lin. Alg. for GPUs

- CUda SParse: a templated library for GPUs and CPUs, providing a high-level interface that hides GPU complexities (NVIDIA, Apache license)
- Built on top of Thrust (NVIDIA)

```
#include <cusp/hyb_matrix.h>
#include <cusp/io/matrix_market.h>
#include <cusp/krylov/cg.h>
int main(void)
{
    // create an empty sparse matrix structure (HYB format)
    cusp::hyb_matrix<int, float, cusp::device_memory> A;

    // load a matrix stored in MatrixMarket format
    cusp::io::read_matrix_market_file(A, "5pt_10x10.mtx");

    // allocate storage for solution (x) and right hand side (b)
    cusp::array1d<float, cusp::device_memory> x(A.num_rows, 0);
    cusp::array1d<float, cusp::device_memory> b(A.num_rows, 1);

    // solve the linear system A * x = b with the Conjugate Gradient
method
    cusp::krylov::cg(A, x, b);
    return 0;
}
```

CUSP: conclusions

- Template metaprogramming: conceptually easy to specify one template for different data types, e.g., single/double precision
- Only supports single node execution (*multi-node implementation should be at a higher level, anyway*)
- Has only CUDA backends: only for NVIDIA GPUs
- *Not supported by NVIDIA! Future: uncertain*
- Community effort, driven by user demand
- Location: `https://github.com/cusplibrary`

AmgX: linear solvers

- NVIDIA's attempt at linear solvers

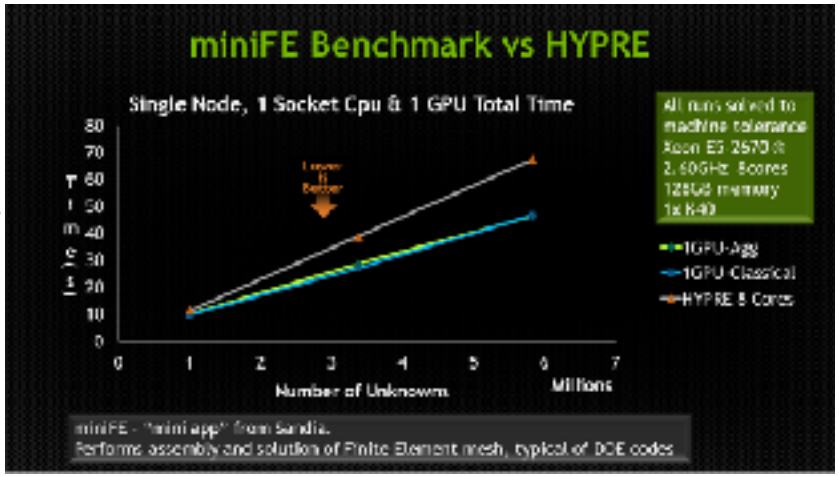
➡ `https://developer.nvidia.com/amgx`

- Free for non-commercial use
- Single node:
  - ➡ Krylov methods, smoothers, preconditioners
  - ➡ algebraic multigrid
- Requires matrix construction
- OpenMP support
- Ostensible MPI-support

Not yet installed on Piz Daint!

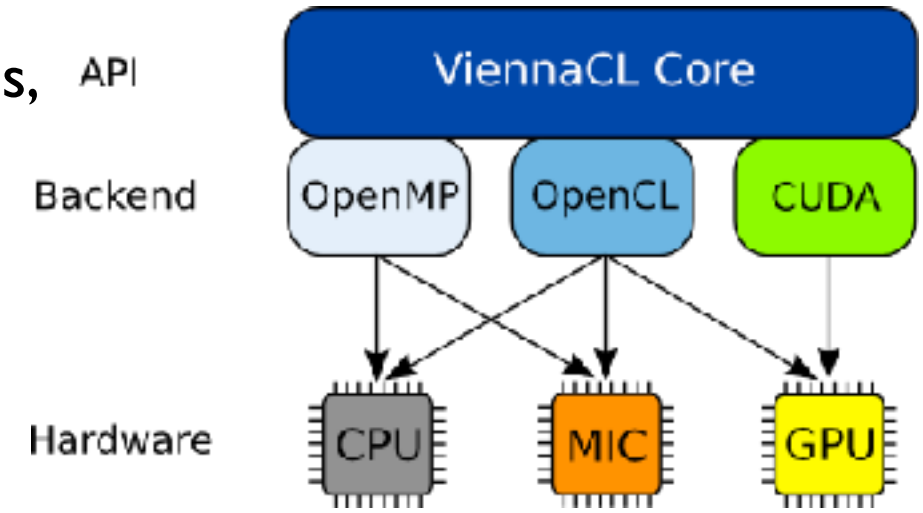
AmgX: linear solvers

```
//One header
#include “amgx_c.h”

//Read config file
AMGX_create_config(&cfg, cfgfile);

//Create resources based on config
AMGX_resources_create_simple(&res, cfg);

//Create solver object, A,x,b, set precision
AMGX_solver_create(&solver, res, mode, cfg);
AMGX_matrix_create(&A,res,mode);
AMGX_vector_create(&x,res,mode);
AMGX_vector_create(&b,res,mode);

//Read coefficients from a file
AMGX_read_system(&A,&x,&b, matrixfile);

//Setup and Solve
AMGX_solver_setup(solver,A);
AMGX_solver_solve(solver, b, x);
```



ViennaCL: Sparse Lin. Algebra on multiple platforms

- Linear algebra library for many core architectures (GPUs, CPUs, Intel Xeon Phi)
- Supports BLAS 1-3
- Iterative solvers
- Sparse row matrix-vector multiplication, solvers
- Goals:
  - ➡ Simplicity, minimal dependencies
  - ➡ Compatible with Boost.uBLAS, Eigen,...
  - ➡ Open source, header-only library



Boost: Solve linear system

```
using namespace boost::numeric::ublas;
matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);

/* Fill A, x, rhs here */

// Some operations
rhs += 2.0 * x;
double val = inner_prod(x, rhs);
matrix += val * outer_prod(x, rhs);

x = solve(A, rhs, upper_tag()); // Upper triangular solver

std::cout << " 2-norm: " << norm_2(x) << std::endl;
std::cout << "sup-norm: " << norm_inf(x) << std::endl;
```

ViennaCL: Solve linear system

```
using namespace viennacl;
using namespace viennacl::linalg;

matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);

/* Fill A, x, rhs here */

// Some operations
rhs += 2.0 * x;
double val = inner_prod(x, rhs);
matrix += val * outer_prod(x, rhs);

x = solve(A, rhs, upper_tag()); // Upper triangular solver

std::cout << " 2-norm: " << norm_2(x) << std::endl;
std::cout << "sup-norm: " << norm_inf(x) << std::endl;
```

ViennaCL: Memory Model

Memory buffers need to be managed differently for each of the compute backends (OpenMP, CUDA, OpenCL)

- Memory domain abstraction in class `viennacl::backend::mem_handle`
- Raw handles from `cuda_handle()`, `opencl_handle()` and `ram_handle()`
- backend is required to support:
  - ➡ `memory_create()`: Create a memory buffer
  - ➡ `memory_copy()`: Copy the (partial) contents of one buffer to another
  - ➡ `memory_write()`: Write from a memory location in CPU RAM to the buffer
  - ➡ `memory_read()`: Read from the buffer to a memory location in CPU RAM

ViennaCL: Interoperability

Standard C++ vectors and Boost uBLAS vectors can be passed to/from ViennaCL vectors:

```
std::vector<double> std_x(100)
ublas::vector<double> ublas_x(100);
viennacl::vector<double> vcl_x1, vcl_x2;

/* setup of std_x and ublas_x omitted */
viennacl::copy(std_x.begin(), std_x.end(), vcl_x1.begin());
viennacl::copy(ublas_x.begin(), ublas_x.end(), vcl_x2.begin());
```

ViennaCL: Solve sparse system

```
using namespace viennacl;
using namespace viennacl::linalg;
compressed_matrix<double> A(1000, 1000);  // sparse matrix format
vector<double> x(1000), rhs(1000);
/* Fill A, x, rhs here */
x = solve(A, rhs, cg_tag()); // Conjugate Gradient solver
x = solve(A, rhs, bicgstab_tag()); // BiCGStab solve
x = solve(A, rhs, gmres_tag()); // GMRES solver
```

uBLAS has no iterative solvers, but thanks to compatibility

```
using namespace boost::numeric::ublas;
using namespace viennacl::linalg;
compressed_matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);
/* Fill A, x, rhs here */
x = solve(A, rhs, cg_tag()); // Conjugate Gradient solver
x = solve(A, rhs, bicgstab_tag()); // BiCGStab solver
x = solve(A, rhs, gmres_tag()); // GMRES solver
```

ViennaCL: temporaries

Consider the expression

```
vec1 = vec2 + alpha * vec3 - beta * vec4;
```

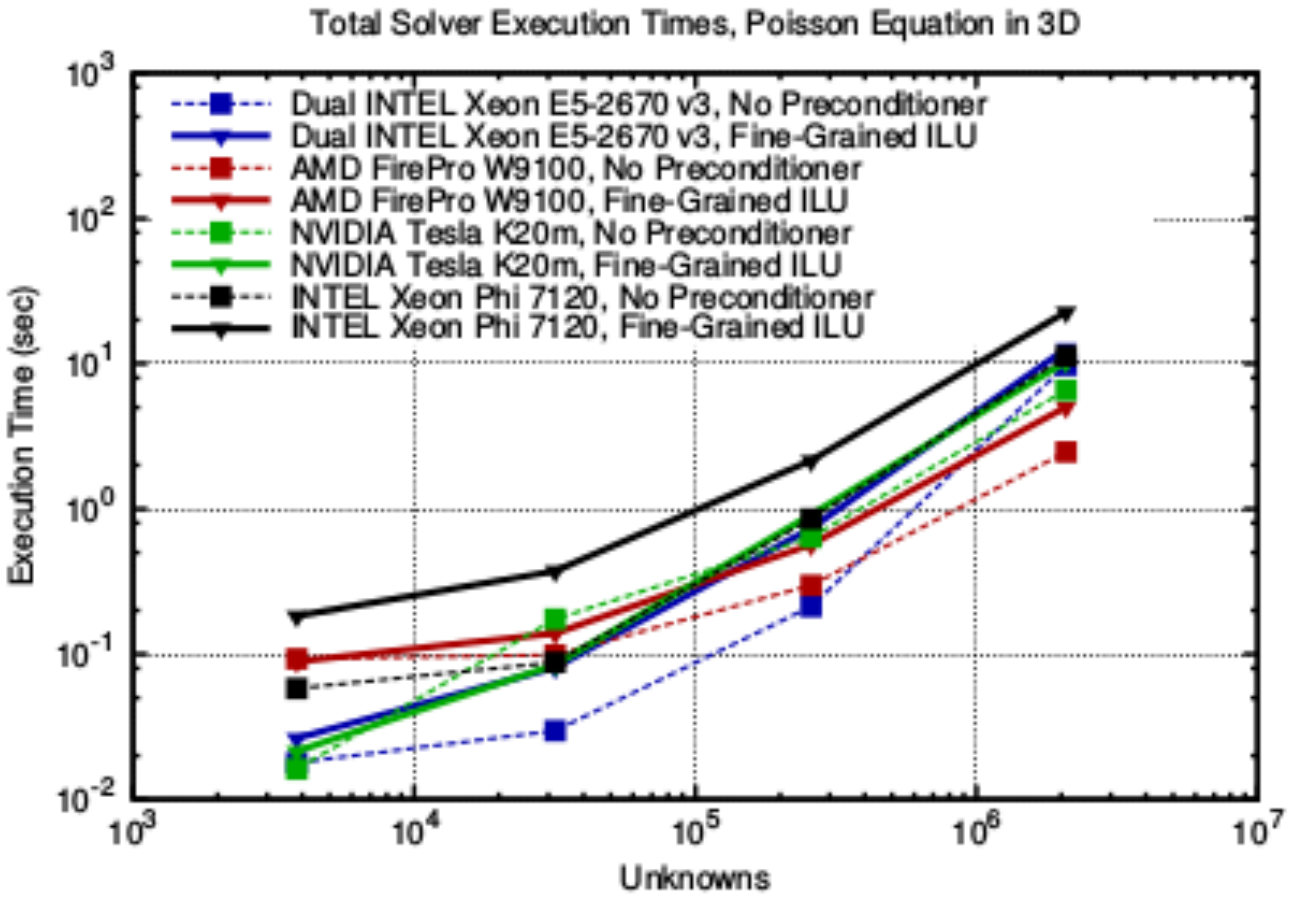
With naive C++ this could be equivalent to

```
tmp1 <- alpha * vec3
tmp2 <- beta * vec4;
tmp3 <- tmp1 - tmp2;
tmp4 <- vec2 + tmp3;
vec1 <- tmp4;
```

Temporaries are costly on CPUs, even more so on GPUs

➡ *Expression templates* reduce usage of temporaries

ViennaCL: 3D Poisson problem



Less is better

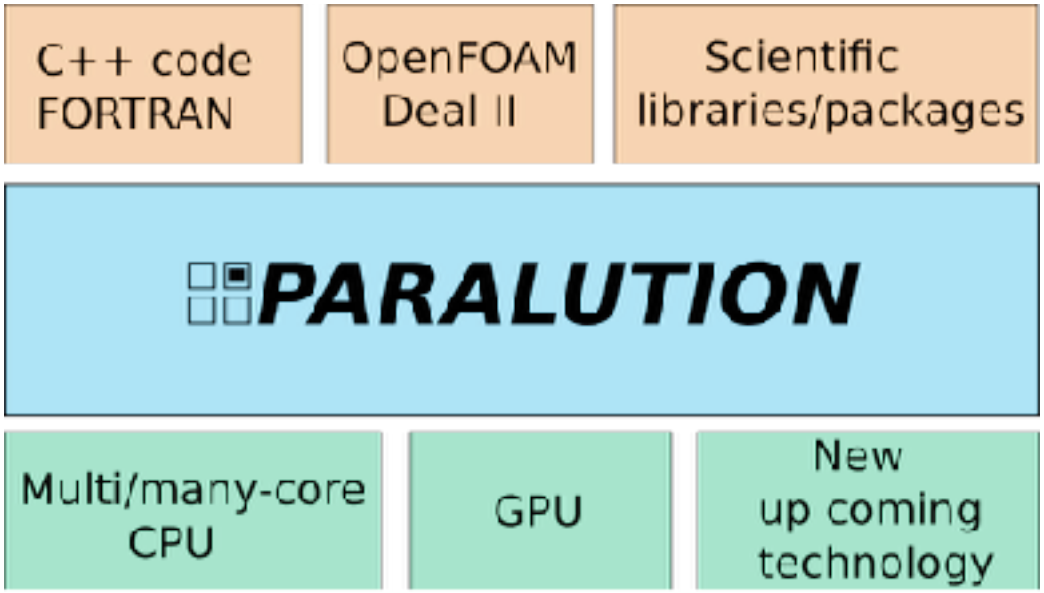
Credits: Rupp, et al., SIAM 2016

ViennaCL: conclusions

- A logical extension of Boost uBLAS template library
- Backends for OpenMP, OpenCL and CUDA
- Runs on CPUs, Intel Xeon Phi, NVIDIA + AMD GPUs
- Little activity by ViennaCL team since 2016
- Krylov-subspace solvers, minimal spectrum of preconditioners
- Interoperates with other libraries:
  - Boost
  - Eigen
  - PETSc (ViennaCL incorporated into backend)
  - others...

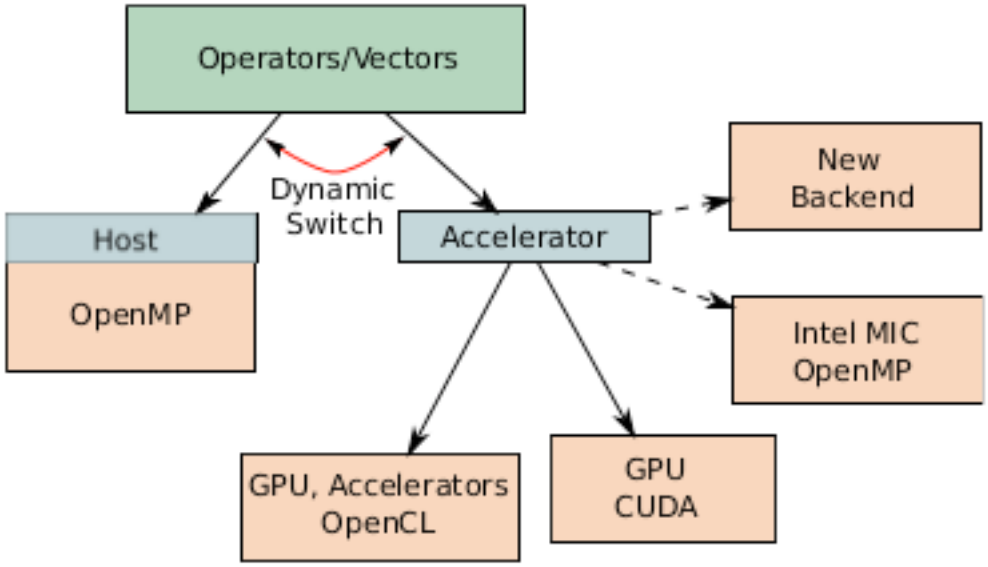
➡ *Promising library supporting high-level linear algebra*

PARALUTION: Sparse Linear Algebra on multiple platforms



- *Sparse Iterative solvers & preconditioners*
- *Targeted: CPUs + accelerators*
- *Hardware abstraction*
- *OpenMP/CUDA/OpenMP opaque to user*
- *Code portable*
- *GPL v3*
- *http://www.paralution.com*

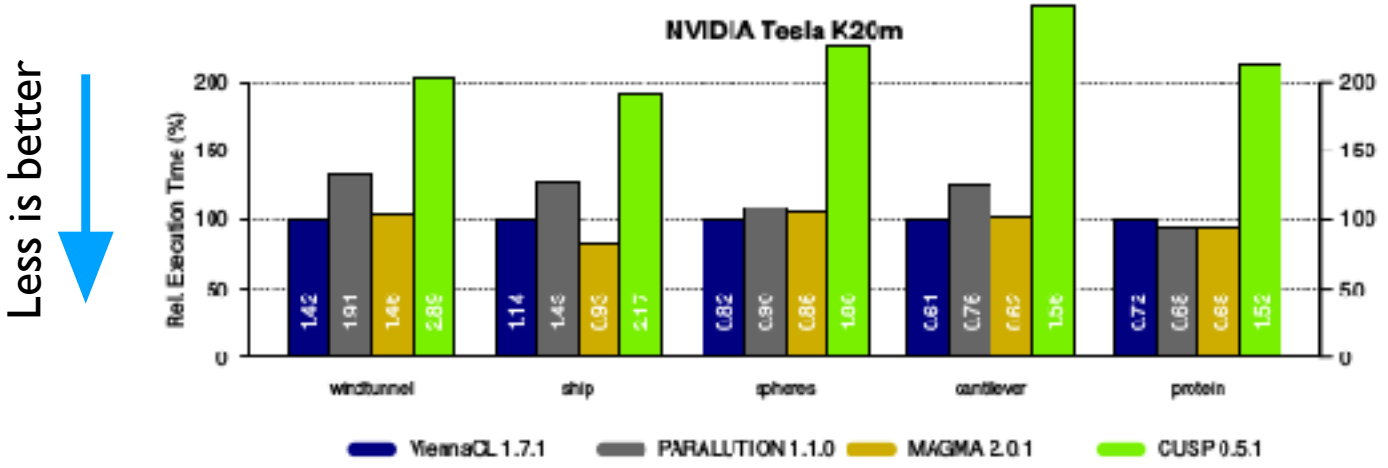
Dynamic switch to accelerator



Accelerated execution takes place on accelerator, if one found at run time

Paralution: conclusions

- Accelerator support determined at run-time
- Multiple backends for OpenMP, OpenCL and CUDA
- Freely available under GPLv3 www.paralution.com
- Krylov-subspace solvers, minimal spectrum of preconditioners
- Development effort appears to have stopped in 2016
- Performance?



Credits: Rupp, et al., SIAM 2016

Beyond single node linear algebra

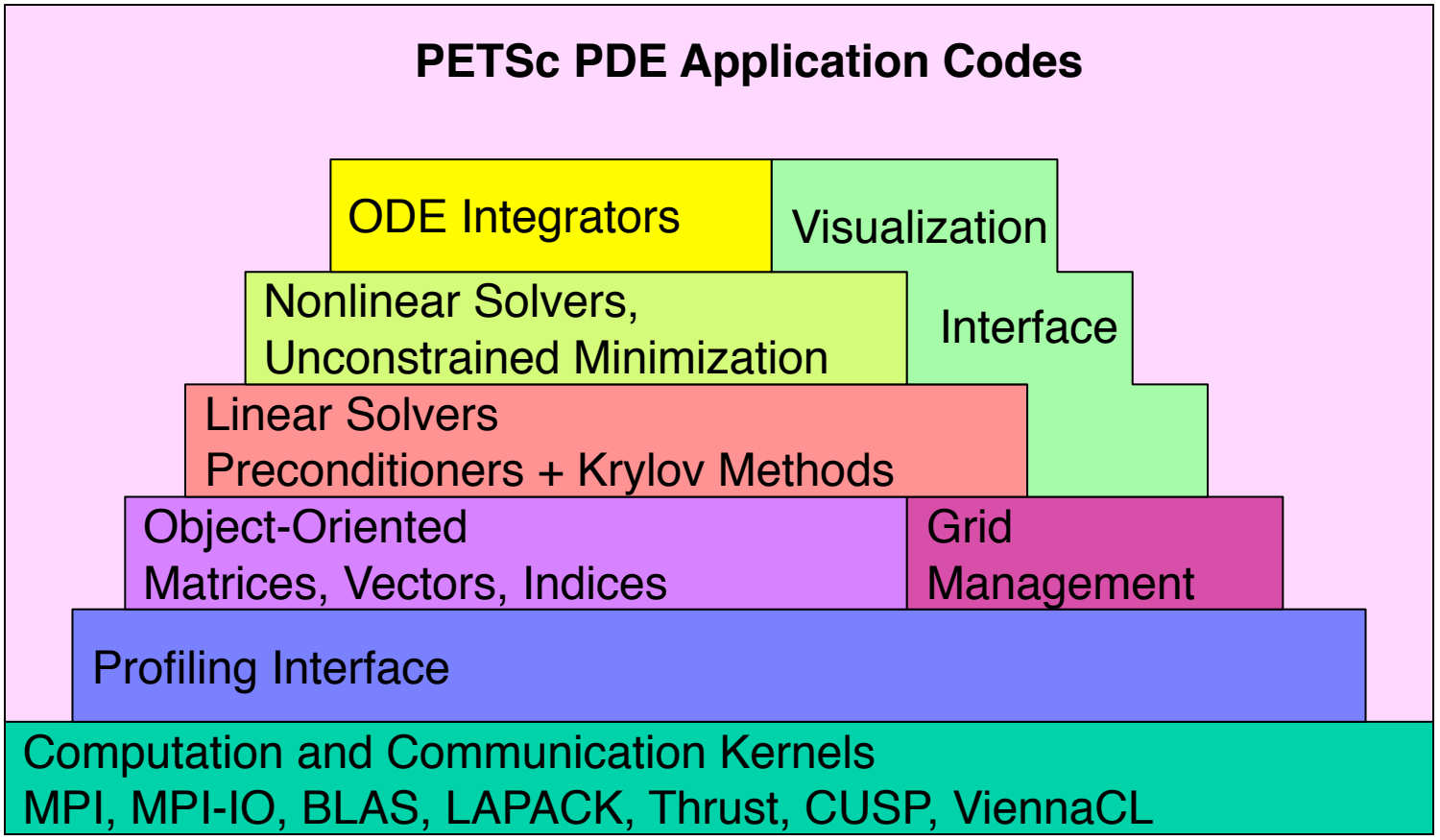
So perhaps single-node sparse/dense linear algebra is covered

- What about distributed memory parallelism?
- What about problems beyond linear algebra?
- Fact: there are solid MPI-GPU development efforts ongoing:
  - ➡ SLATE (e.g. ScaLAPACK replacement effort)
  - ➡ PETSc (next slides)
  - ➡ Trilinos (next slides)
- *Current multi-node GPU support for non-linear problems, optimizations, eigenvalue problems and others is provisional at best!*

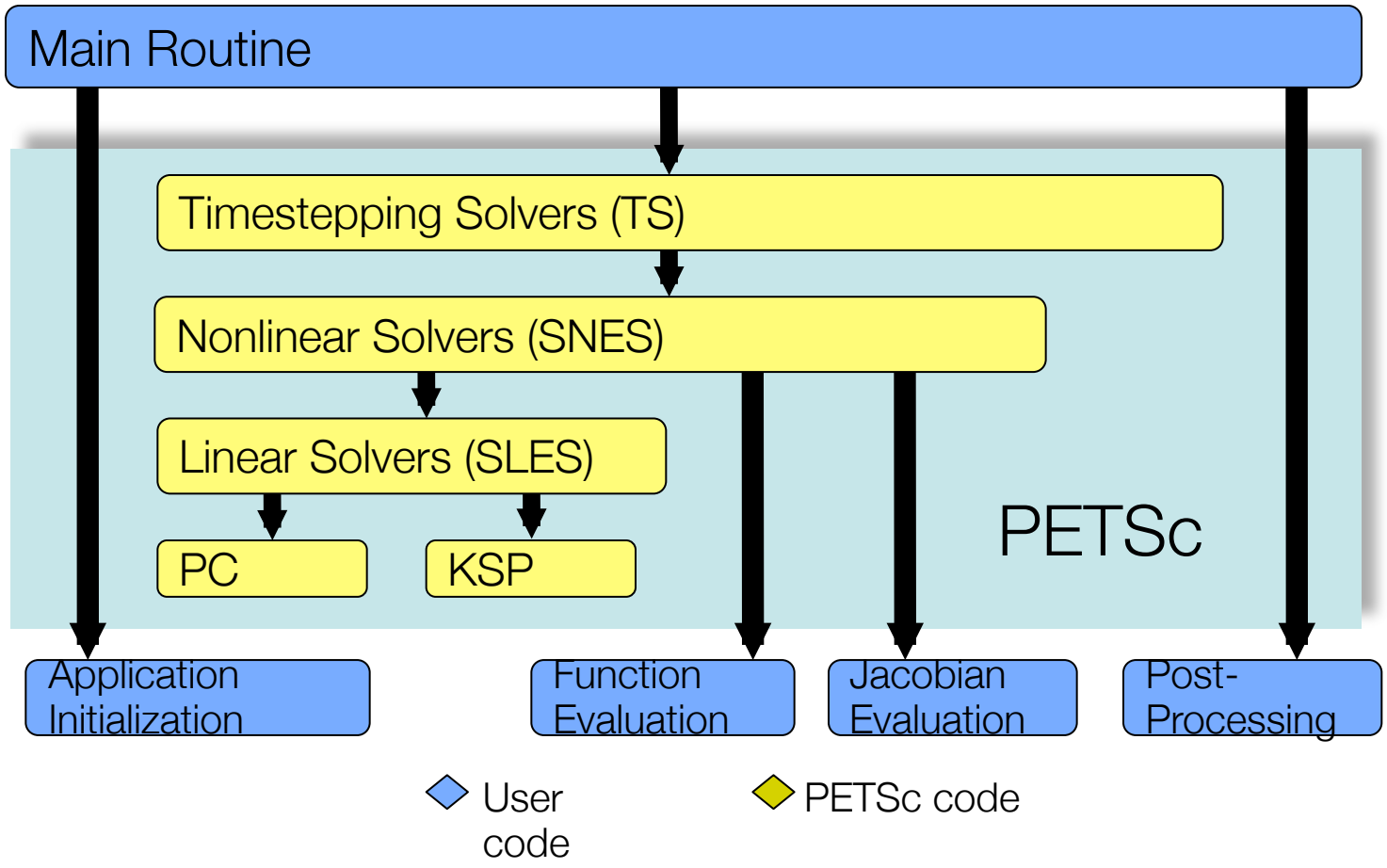
What is PETSc ?

- Effort started in 1991, funded by US DOE and NSF
- Supports non-linear PDE problems
- A freely available (and supported!) research code
  - Available via http://www.mcs.anl.gov/petsc
  - Free for everyone, including industrial users
  - Hyperlinked documentation and manual pages for all routines
  - Many tutorial-style examples
  - Support via email: petsc-maint@mcs.anl.gov
  - Current version: 3.9 (released Apr. 7, 2018)
- Portable to any parallel system supporting MPI
  - Tightly coupled systems, e.g., Cray XK7, XE6, XC30
  - Loosely coupled systems, e.g., networks of workstations, PCs

Structure of PETSc



Flow control for PDE solution

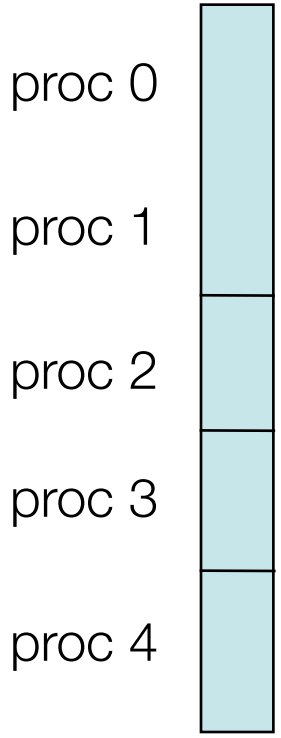


PETSc Programming Model

- **Goals**
  - Portable, runs everywhere, including heterogeneous multi-core
  - Performance
  - Scalable parallelism
- **Approach**
  - Distributed memory, “shared-nothing”
    - Access to data on remote machines or nodes through MPI
  - Can still exploit node parallelism on each node (e.g., SMP), with limitations (see PETSc home page)
  - Hide within parallel objects the details of the communication
  - User orchestrates communication at a higher abstract level than message passing
  - Additional classes added for GPU support

PETSc Data Objects

- PETSc vectors
  - Fundamental objects for storing field solutions, right-hand sides, etc.
  - Each process locally owns a subvector of contiguously numbered global indices
- Create vectors via
  - VecCreate(...,Vec *)
    - MPI_Comm - processors that share the vector
    - number of elements local to this processor
    - or total number of elements
  - VecSetType(Vec,VecType)
    - Where VecType is
      - VEC_SEQ, VEC_MPI, or VEC_SHARED



Parallel Vector and Matrix Assembly

- Processors may generate any entries in vectors and matrices
- Entries need not be generated on the processor on which they ultimately will be stored
- PETSc automatically moves data during the assembly process if necessary

PETSc Linear Solvers

Krylov Methods (KSP)

- Conjugate Gradient
- GMRES
- CG-Squared
- Bi-CG-stab
- Transpose-free QMR
- etc.

Preconditioners (PC)

- Block Jacobi
- Overlapping Additive Schwarz
- ICC, ILU via BlockSolve95
- ILU(k), LU (sequential only)
- etc.

PETSc: evolving GPU support

PETSc GPU Model

- Each MPI process has access to a single GPU, which has its own memory
- Backends for CUSP, cuSPARSE and ViennaCL available
- New implementations of **Vec** and **Mat (type at run-time)**
  - ➡ Vectors with types `VECSEQCUSP`, `VECMPICUSP`, or `VECCUSP`
  - ➡ Matrices with types `MATSEQAIJCUSP`, `MATMPIAIJCUSP`, or `MATAIJCUSP`
  - ➡ Matrices with types `MATSEQAIJCUSPARSE`, `MATMPIAIJCUSPARSE`, or `MATAIJCUSPARSE`

PETSc: GPU support

Objects support both CPU and GPU copy of data, and carry flags indicated the validity of the data

PETSC_CUDA_UNALLOCATED	Memory not allocated on GPU
PETSC_CUDA_GPU	Values on GPU are current
PETSC_CUDA_CPU	Values on CPU are current
PETSC_CUDA_BOTH	Values on both devices current

Implementations for GPU-CPU data movement

- `VecCUDACopyToGPU`
- `VecCUDACopyFromGPU`
- `...`

these are generally used internally in solvers

PETSc: GPU support

How it works, at least conceptually

- User needs to specify types of vectors and matrices at run time
- Functionality invoked from backend (CUSP, cuSPARSE, ViennaCL, if available)
- Implementation should be transparent to user
- Alternatively, user can program in CUDA and access device objects calling thrust:: and cusp:: operators directly.

PETSc GPU version not yet in cray-petsc on Daint. Access to GPU version on request

PETSc: conclusions

- PETSc library of PDE/ODE solvers
- *Extensive selection of solvers, high quality, good support, free*
- *However: much more effective for new code*
- *Saddled by design choices, e.g., not thread-safe*
- Monolithic: one package tries to solve all, though there are adapters to other libraries
- *MPI-GPU support for matrix/vector algorithms, can be invoked at run-time. Interfaces to CUSP, CUSparse, ViennaCL*

Trilinos: a ‘pearl necklace’ of packages

Object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems

- http://trilinos.sandia.gov
- http://code.google.com/p/trilinos/wiki/TrilinosHandsOnTutorial

Trilinos: parallel packages (with GPU support)

- Basic linear algebra: *Epetra/EpetraExt (C++), Tpetra (C++ templates)*
- Preconditioners: *AztecOO, Ifpack2, ML, Meros*
- Iterative linear solvers: *AztecOO, Belos*
- Direct linear solvers: *Amesos (SuperLU, UMFPACK, MUMPS, ScaLAPACK, …)*
- Non-linear / optimization solvers: *NOX, MOOCHO*
- Eigensolvers: *Anasazi*
- Mesh generation / adaptivity: *Mesquite, PAMGEN*
- Domain decomposition: *Claps*
- Partitioning / load balance: *Isorropia*, *Zoltan2*

Trilinos: *Kokkos* Compute Model

- C++ library, not new language (extension)
- supports thread-scalable parallel patterns
- one code run on many architectures (CPU, GPU, Xeon Phi)
- minimal architecture-specific implementation details
- offers multi-dimensional arrays with architecture-dependent layouts (to avoid data layout performance bottlenecks)
- Utilizes functors (like Thrust) and C++11 Lambda syntax for data parallel work

Kokkos comparison

Serial

```
for (size_t i = 0; i < N; ++i) {
  /* loop body */
}
```

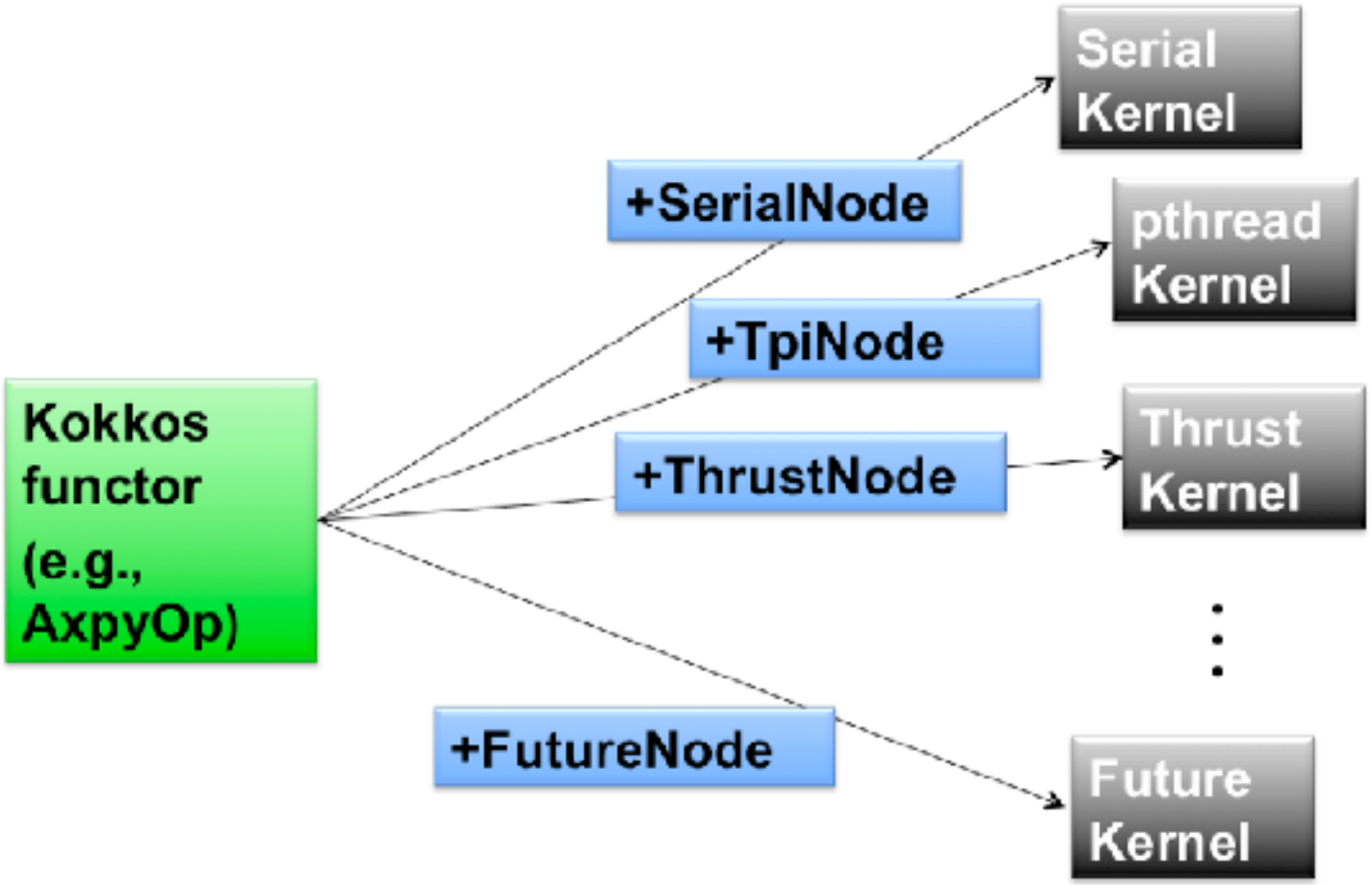
OpenMP

```
#pragma omp parallel for
for (size_t i = 0; i < N; ++i) {
  /* loop body */
}
```

Kokkos

```
parallel_for(N, [=] (const size_t i) {
  /* loop body */
});
```

Kokkos: compile-time kernel specialization



Kokkos: simple reduction examples

Functor

```
int sum = 0;
Kokkos::parallel_reduce (n, squaresum (), sum);

struct squaresum { // Specify type of reduction value with a
"value_type"
  typedef int value_type;
  KOKKOS_INLINE_FUNCTION
  void operator () (const int i, int& lsum) const {
    lsum += i*i; // compute the sum of squares
  }
};
```

Lambda

```
Kokkos::parallel_reduce (n, [=] (const int i, int& lsum) {
  lsum += i*i;
}, sum);
```

Anasazi: Eigenvalues/vectors of large, sparse matrices

- Techniques based on Lanczos iteration (symm. A)

$$r_0 = q_1; \quad \beta_0 = 1; \quad q_0 = 0; \quad j = 0$$

$$while \quad \beta_j \neq 0 \quad \{$$

$$q_{j+1} = r_j / \beta_j; \quad j = j + 1; \quad \alpha_j = q_j^T A q_j$$

$$r_j = (A - \alpha_j I) q_j - \beta_{j-1} q_{j-1}; \quad \beta_j = \left\| r_j \right\|_2$$

$$\}$$

- Lanczos vectors: q_j
- Form tridiagonal matrix T: diagonal α_j subdiagonal β_j
- Diagonalization of T is stable iterative procedure

Anasazi: classes for $Ax = \lambda Bx$

- `Anasazi::Eigenproblem`
  - Contains components of eigen-problem
  - setOperator, SetA, SetB, setPrec, setInitVec
- `Anasazi::Eigensolution`
  - Manages the solution of the eigen-problem
- `Anasazi::Eigensolver`
  - Defines interface which must be met by any solver
  - Currently implemented solvers: BlockDavidson, BlockKrylovSchur, LOBPCG
- `Anasazi::SolverManager`
  - 'Turn-key' class to use existing eigen-solvers

NOX: non-linear equations

- Solve $F(x) = 0$ with $F(x) = \begin{pmatrix} f_1(x_1, \cdots, x_n) \\ \vdots \\ f_n(x_1, \cdots, x_n) \end{pmatrix}$ and $J_{i,j} = \frac{\partial F_i}{\partial x_j}(x)$
- User supplies:
  - Function F(x) evaluation
  - Optional: Jacobian evaluation, preconditioner
- With good guess, convergence quadratic
- Heuristics used to improve first guess
- PETSc interface available
- http://trilinos.sandia.gov/packages/nox/

Trilinos: conclusions

- Non-monolithic set of packages, some interoperating tightly, some loosely, some not at all
- Large development team, free software, technically advanced, latest solvers, following emerging technologies (e.g. GPUs)
- Solvers are opaque, hard to see internals, bugs can be hard to deal with
- GPU implementation through Kokkos abstraction, only *Tpetra, Belos, Anasazi, Ifpack2, Zoltan2, NOX*

cray-trilinos on Daint does not enable GPU backend!

GPU-enabled Libraries Summary

- The take home message is:

  Don't recreate the wheel

- A limited number of GPU-libraries available, e.g. cuXXXX (vendor), Thrust, ViennaCL, ...
- Parallel distributed memory libraries in development, e.g., SLATE, PETSc, Trilinos, within overarching message-passing framework
- *BUT: current GPU library support is sobering*
- *Please let us know your needs!*

Acknowledgments

- The PETSc team
- Trilinos team
- UT, Knoxville: Innovative Computing Laboratory
- NVIDIA
- *And thanks to you for listening!*