Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA)

Unit 11

MAGMA, BLAS, and Batched GPU Computing

Kwai Wong, Stan Tomov
Rocco Febbo, Julian Halloy

University of Tennessee, Knoxville

November 5, 2021

LAPENNA

The major goal of The Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA) program is to provide essential knowledge to advance literacy in AI to sustain the growth and development of the workforce in the cyberinfrastructure (CI) ecosystem on data-driven sciences. This program aims to prepare college researchers to enable, design, and direct their own in-house data-driven science programs and incorporate perspectives from their research activities into their course curricula. LAPENNA focuses to deliver algorithmic and computational techniques, numerical and programming procedures, and implementation of AI software on emergent CPU and GPU platforms.

- **Support from NSF, UTK, JICS, ICL, NICS**
- **LAPENNA, www.jics.utk.edu/lapenna, NSF award #202409**
- **www.icl.utk.edu, cfdlab.utk.edu, www.xsede.org, www.jics.utk.edu/recsem-reu,**
- **MagmaDNN is a project grown out from the RECSEM REU Summer program supported under NSF award #1659502**

Meeting Time

- ✓ **Webinar Meeting time. Friday 4:00pm ET,**
- ✓ **Tentative schedule, www.jics.utk.edu/lapenna --> Fall 2021**

Topic: LAPENNA Fall 2021 Meeting
Time: Aug 20, 2021 04:00 PM Eastern Time (US and Canada)
Every week on Fri, 15 occurrence(s)
Aug 20, 2021 04:00 PM
Aug 27, 2021 04:00 PM
Sep 3, 2021 04:00 PM
Sep 10, 2021 04:00 PM
Sep 17, 2021 04:00 PM
Sep 24, 2021 04:00 PM
Oct 1, 2021 04:00 PM
Oct 8, 2021 04:00 PM
Nov 5, 2021 04:00 PM
Please download and import the following iCalendar (.ics) files to your calendar system.
Weekly: https://tennessee.zoom.us/meeting/tJwud-isrzouHtS8Ri0ny5ysjFEWJlktQuu-/ics?icsToken=98tyKuCgrTsrHtWUtB2HRow-A4igd-jzmH5bgrduxC3sUy5KNxrlPMRnBZhzG8zh

Join from PC, Mac, Linux, iOS or Android: https://tennessee.zoom.us/j/98301411440
Password: 893603

Or iPhone one-tap (US Toll): +13126266799,98301411440# or +16468769923,98301411440#

Schedule of Fall 2021 – Friday 4:00-6:00pm

Month	Week	Date	Arrangement
August	Week 00	20	First Meeting, Logistics
	Week 01	27	Computational Ecosystem
September	Week 02	3	Linear Algebra, Statistical Learning
	Week 03	8 (4:45 - 6pm) , 10	Q&A (LA), DNN, Forward Path, Math
	Week 04	17	Backward Path, MLP, example
	Week 05	24	CNN computation
October	Week 06	Sept 29 (4:45-6pm), 1	Q&A (MLP), Backpropagation example
	Week 07	8	CNN network, Linear Algebra
	Week 08	15	Object detection, optimization
	Week 09	22	Image Segmentation
	Week 10	27 (4:45-6pm), 29	Q&A (RC vehicle), Magma, GPU CUDA, CUDNN
November	Week 11	5	Magma, CUBLAS, CUDNN
	Week 12	12	RNN, LSTM, Transformer
	Week 13	19	Recap Summary, closing
December	Workshop	10-13	Workshop

Joint Institute for Computational Sciences

INNOVATIVE COMPUTING LABORATORY

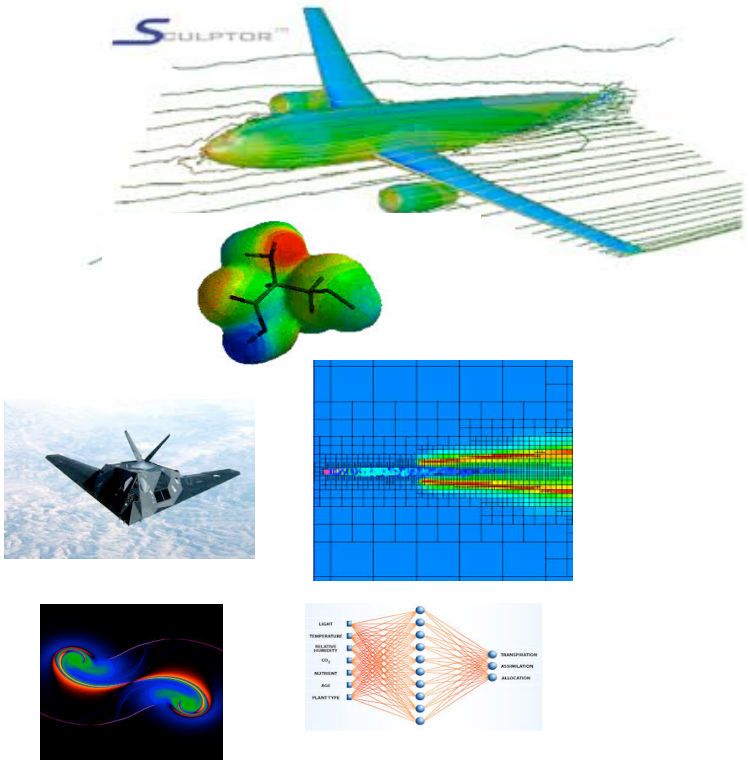
Matrix Algebra on GPU and Multicore Architectures (MAGMA)

Joint Institute for Computational Sciences

INNOVATIVE COMPUTING LABORATORY

Matrix Algebra on GPU and Multicore Architectures (MAGMA)

- Linear systems: Solve $Ax = b$
  - Computational electromagnetics, material science, applications using boundary integral equations, airflow past wings, fluid flow around ship and other offshore constructions, and many more
- Least squares: Find x to minimize $|| Ax – b ||$
  - Computational statistics (e.g., linear least squares or ordinary least squares), econometrics, control theory, signal processing, curve fitting, and many more
- Eigenproblems: Solve $Ax = \lambda x$
  - Computational chemistry, quantum mechanics, material science, face recognition, PCA, data-mining, marketing, Google Page Rank, spectral clustering, vibrational analysis, compression, and many more
- SVD: $A = U \Sigma V^*$ ($Au = \sigma v$ and $A^*v = \sigma u$)
  - Information retrieval, web search, signal processing, big data analytics, low rank matrix approximation, total least squares minimization, pseudo-inverse, and many more
- Many variations depending on structure of A
  - A can be symmetric, positive definite, tridiagonal, Hessenberg, banded, sparse with dense blocks, etc.
- Sparse solvers



Overview of Dense Numerical Linear Algebra Libraries

netlib.org

icl.utk.edu/research

BLAS Kernels for dense linear algebra

LAPACK Sequential dense linear algebra

ScaLAPACK Parallel distributed dense linear algebra

PLASMA dense linear algebra (multicore)

MAGMA Dense/batched/sparse linear algebra (accelerators)

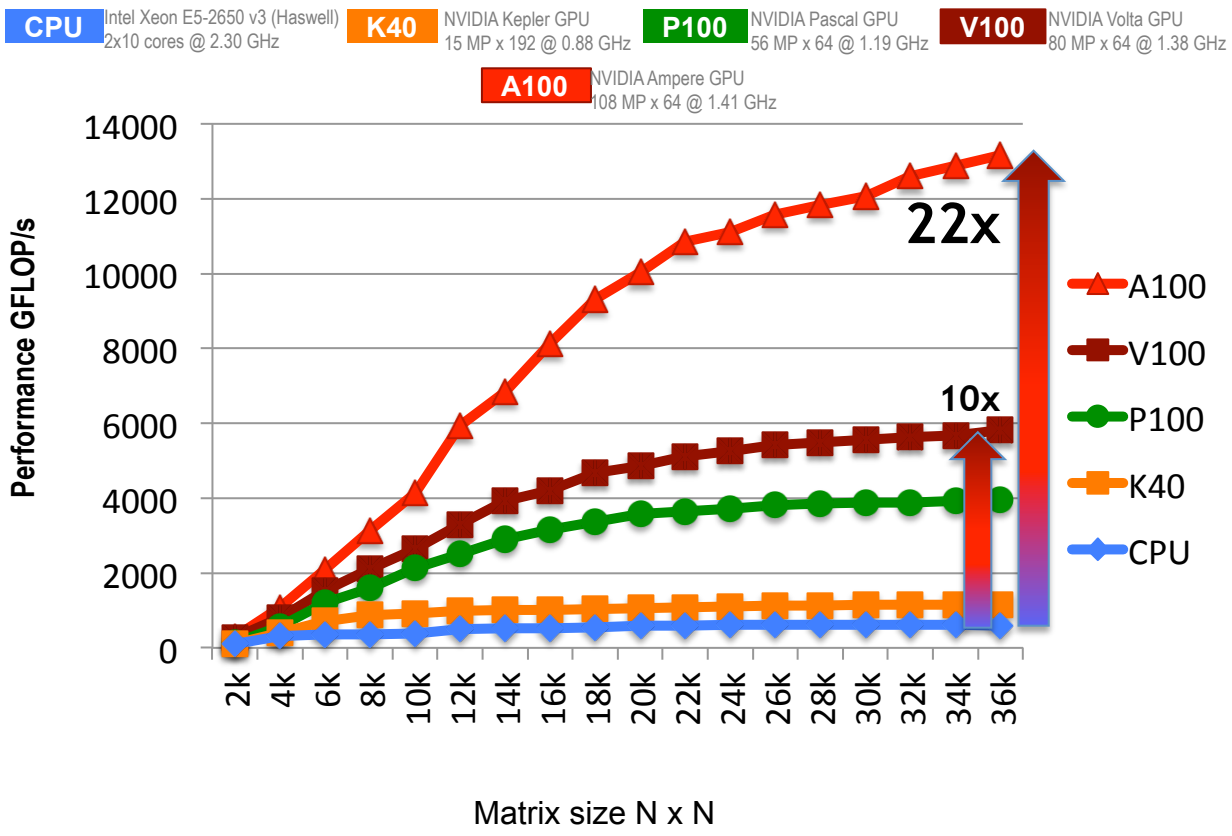
SLATE dense linear algebra (distributed memory / muticore / accelerators)

new software for multicore and accelerators

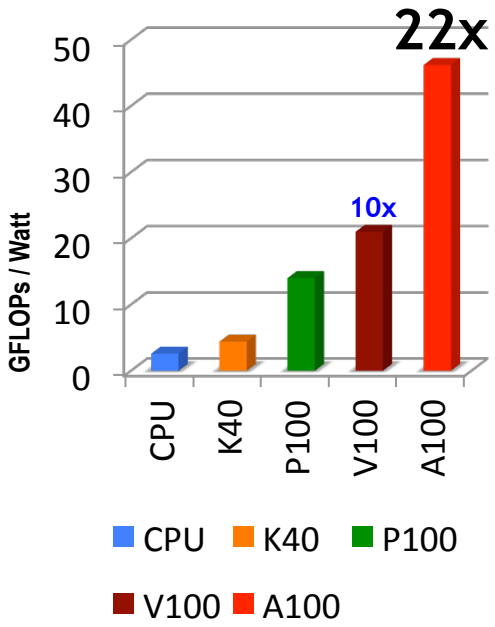
MAGMA on GPUs

PERFORMANCE & ENERGY EFFICIENCY

MAGMA 2.6.1 LU factorization in double precision arithmetic



Energy efficiency
(under ~ the same power draw)



MAGMA Today

MAGMA – provides highly optimized LA well beyond LAPACK for GPUs;
– research vehicle for LA on new architectures for a number of projects.

for architectures in
{ CPUs + Nvidia GPUs (CUDA),
CPUs + AMD GPUs (HIP & OpenCL),
CPUs + Intel Xeon Phis,
manycore (native: GPU or KNL/CPU),
embedded systems, combinations, and
software stack, e.g., since CUDA x}

for precisions in
{ s, d, c, z,
half-precision (FP16),
mixed, … }

for interfaces
{ heterogeneous CPU/GPU, native, … }

- **LAPACK**
- **BLAS**
- **Batched LAPACK**
- **Batched BLAS**
- **Sparse**
- **Tensors**
- **MAGMA-DNN**
- **Templates**
- **…**

- Collaboration and support from vendors
  NVIDIA, Intel, and AMD
- Two releases per year
  Latest MAGMA 2.6.1 released on July 13, 2021
  Number of downloads per release ~ 4K
  (MAGMA 2.5.4 – 6,111 downloads, MAGMA 2.6.x – 5,794 downloads)
- MAGMA Forum – now Google Groups
  72 members, 35 discussion topics
- MAGMA Issues
  48 issues total; 23 are still open, and 25 have been resolved
- MAGMA is incorporated/used in
  MATLAB (as of the R2010b),
  contributions in CUBLAS and MKL,
  AMD, Siemens (in NX Nastran 9.1), ArrayFire,
  ABINIT, Quantum-Espresso, R (in HiPLAR & CRAN),
  SIMULIA (Abaqus), MSC Software (Nastran and Marc),
  Cray (in LibSci for accelerators libsci_acc),
  Nano-TCAD (Gordon Bell finalist),
  Numerical Template Toolbox (Numscale), and others.
- MAGMA used in ECP – CEED, PEEKS, xSDK, ALEXA/TASMANIAN, SLATE & FFT, ExaSGD; Provides LAPACK:MAGMA and BLAS:MAGMA product

MAGMA

- Availability
  - http://icl.utk.edu/magma/ download (latest MAGMA 2.6.1) , documentation, forum
  - https://bitbucket.org/icl/magma Git repo
- Support
  - Linux, macOS, Windows
  - CUDA >= 7.0; recommend latest CUDA
  - CUDA architecture >= 2.0 (Fermi, Kepler, Maxwell, Pascal, Volta, Turing)
  - AMD GPUs through HIP
  - BLAS & LAPACK: Intel MKL, OpenBLAS, macOS Accelerate, ...
- May be pre-installed on supercomputers

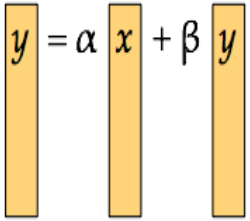
```
titan-ext1> module avail magma
---------- /sw/xk6/modulefiles ----------
magma/1.3            magma/1.6.2(default)
```

Installation options

1. Makefile
   - Edit make.inc for compilers and flags (see make.inc examples)
   - `magma> make && make install`
2. CMake
   - `magma> mkdir build && cd build`
   - `magma/build> cmake ..  or  ccmake ..`
   - Adjust settings, esp. LAPACK_LIBRARIES and GPU_TARGET
   - `magma/build> make && make install`
3. Spack
   - `spack install magma`

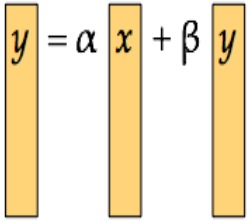
BLAS: Basic Linear Algebra Subroutines

- Level 1 BLAS — vector operations
  - $O(n)$ data and flops (floating point operations)
  - Memory bound:
    $O(1)$ flops per memory access

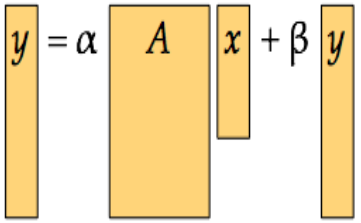


BLAS: Basic Linear Algebra Subroutines

- Level 1 BLAS — vector operations
  - $O(n)$ data and flops (floating point operations)
  - Memory bound:
    $O(1)$ flops per memory access

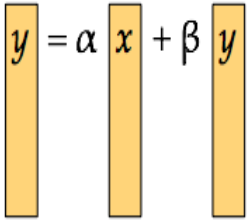


- Level 2 BLAS — matrix-vector operations
  - $O(n^2)$ data and flops
  - Memory bound:
    $O(1)$ flops per memory access

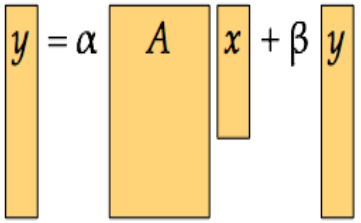


BLAS: Basic Linear Algebra Subroutines

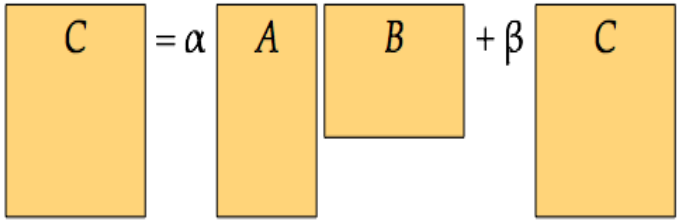
- Level 1 BLAS — vector operations
  - $O(n)$ data and flops (floating point operations)
  - Memory bound:
    $O(1)$ flops per memory access



- Level 2 BLAS — matrix-vector operations
  - $O(n^2)$ data and flops
  - Memory bound:
    $O(1)$ flops per memory access

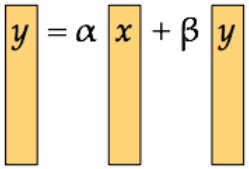


- Level 3 BLAS — matrix-matrix operations
  - $O(n^2)$ data, $O(n^3)$ flops
  - Surface-to-volume effect
  - Compute bound:
    $O(n)$ flops per memory access

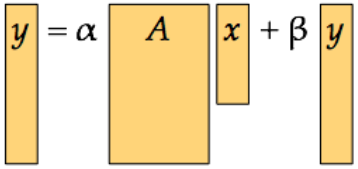


BLAS: Basic Linear Algebra Subroutines

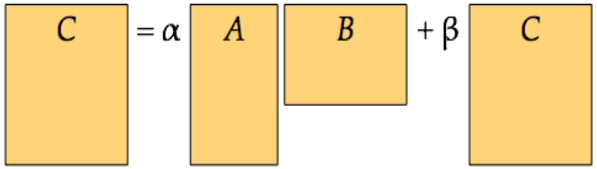
- **Level 1 BLAS**



- **Level 2 BLAS**



- **Level 3 BLAS**



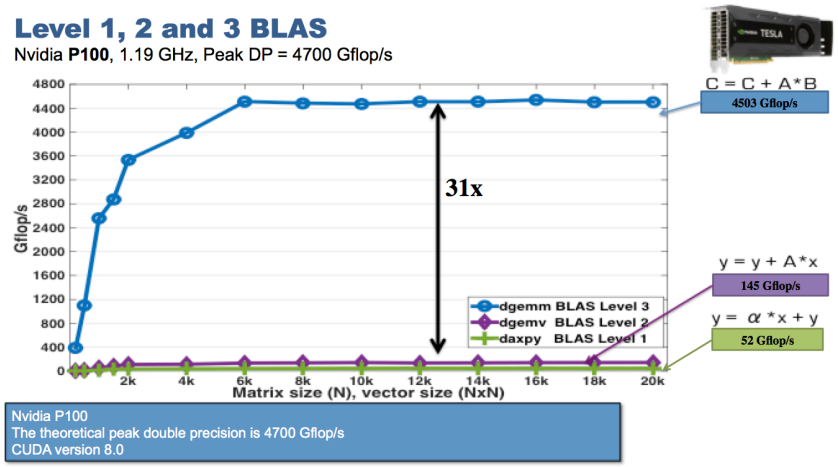
Use of BLAS for portability

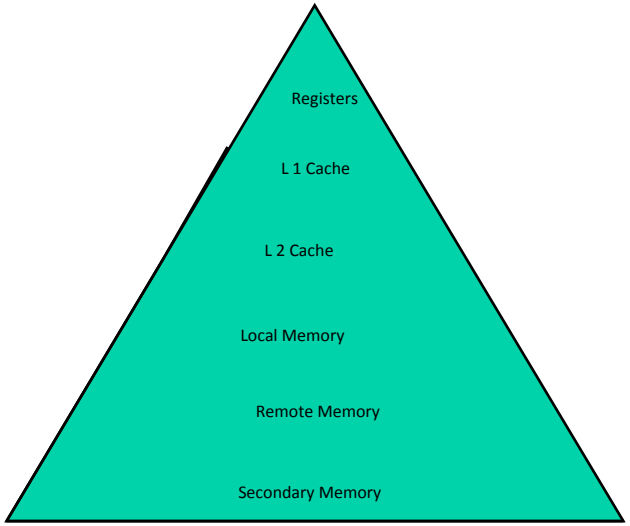
Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Level 1 **BLAS**
LAPACK (80's) (Blocking, cache friendly)		Level 3 **BLAS**
ScaLAPACK (90's) (Distributed Memory)		**PBLAS**
PLASMA (00's) New Algorithms (many-core friendly)		**BLAS** on tiles + DAG scheduling
MAGMA Hybrid Algorithms (heterogeneity friendly)		**BLAS** tasking + (CPU / GPU / Xeon Phi) hybrid scheduling

Why higher level BLAS?

- By taking advantage of the principle of locality:
- Present the user with as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.
- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

BLAS	Memory Refs	Flops	Flops/ Memory Refs
Level 1 $y=y+\alpha x$	$3n$	$2n$	$2/3$
Level 2 $y=y+Ax$	n^2	$2n^2$	2
Level 3 $C=C+AB$	$4n^2$	$2n^3$	$n/2$





MAGMA Overview

- **Hybrid LAPACK-style functions**
  - Matrix factorizations: LU, Cholesky, QR, eigenvalue, SVD, ...
  - Solve linear systems and linear least squares, ...
  - Nearly all are synchronous:
    return on CPU when computation is finished
- **GPU BLAS and auxiliary functions**
  - Matrix-vector multiply, matrix norms, transpose (in-place and out-of-place), ...
  - Most are asynchronous:
    return immediately on CPU; computation proceeds on GPU
- **Wrappers around CUDA and cuBLAS**
  - BLAS routines (gemm, symm, symv, ...)
  - Copy host ⇔ device, queue (stream) support,
    GPU malloc & free, ...

Naming example

- **magma_ or magmablas_ prefix**
- **Precision (1–2 characters)**
  - **S**ingle, **D**ouble, single **C**omplex, "**Z**" double complex, **I**nteger
    Mixed precision (**DS** and **ZC**)
- **Matrix type (2 characters)**
  - **GE**neral **SY**mmetric **HE**rmetian **PO**sitive definite
    ORthogonal **UN**itary **Tr**iangular
- **Operation (2–3+ characters)**
  - **SV** solve
    TRF triangular factorization
    EV eigenvalue problem
    GV generalized eigenvalue problem
    etc.
- **_gpu suffix for interface**

Linear solvers

- Solve linear system: $AX = B$
- Solve linear least squares: minimize $\| AX - B \|_2$

Type	Routine	Mixed precision routine	Interface CPU	Interface GPU
General	dgesv	dsgesv	✓	✓
Positive definite	dposv	dsposv	✓	✓
Symmetric	dsysv		✓	
Hermitian	zhesv		✓	
Least squares	dgels		✓	✓

Selected routines; complete documentation at http://icl.utk.edu/magma/

Eigenvalue / singular value problems

- Eigenvalue problem: $\boldsymbol{Ax} = \boldsymbol{\lambda x}$
- Generalized eigenvalue problem: $\boldsymbol{Ax} = \boldsymbol{\lambda Bx}$ (and variants)
- Singular value decomposition: $\boldsymbol{A} = \boldsymbol{U\Sigma V^H}$

			Interface	
Matrix type	**Operation**	**Routine**	**CPU**	**GPU**
General	SVD	dgesvd, dgesdd	✓	
General non-symmetric	Eigenvalue	dgeev	✓	
Symmetric	Eigenvalue	dsyevd / zheevd	✓	✓
Symmetric	Generalized	dsygvd / zhegvd	✓	

Additional variants; complete documentation at http://icl.utk.edu/magma/
Fastest are divide-and-conquer (gesdd, syevd) and 2-stage versions.

Computational routines

- Computational routines solve one part of problem

			Interface	
Matrix type	**Operation**	**Routine**	**CPU**	**GPU**
General	LU	dgetrf	✓	✓
	Solve (given LU)	dgetrs		✓
	Inverse	dgetri		✓
SPD	Cholesky	dpotrf	✓	✓
	Solve (given LL^T)	dpotrs		✓
	Inverse	dpotri	✓	✓
General	QR	dgeqrf	✓	✓
	Generate Q	dorgqr / zungqr	✓	✓
	Multiply by Q	dormqr / zunmqr	✓	✓

Selected routines; complete documentation at http://icl.utk.edu/magma/

Testers/benchmarks
Every routine has tester, also used as benchmark

```
# (abbreviated output)
magma> cd testing
magma/testing> ./testing_dsymv -n 123 -n 1000:18000:1000 --lapack --check
% MAGMA 2.5.0  compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% CUDA runtime 8000, driver 9000. OpenMP threads 20. MKL 2017.0.1, MKL threads 20.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16276.2 MiB memory, capability 6.0

% uplo = Lower
%   N     MAGMA   Atomics    cuBLAS       CPU   error
%       Gflop/s   Gflop/s   Gflop/s   Gflop/s
%===================================================
  123      0.76      0.76      0.51      0.58   ok    # warmup run
 1000     27.44     34.41     29.88      8.86   ok
 2000     45.74     70.83     33.91     12.78   ok
 3000     75.30    108.51     40.09     17.76   ok
 4000    100.64    131.23     41.13     17.57   ok
 5000    118.17    162.35     41.46     16.33   ok
 6000    141.21    180.43     42.16     17.55   ok
 7000    157.81    200.44     41.94     19.32   ok
 8000    169.54    198.21     41.78     19.12   ok
 9000    188.40    216.07     42.28     21.50   ok
10000    195.92    224.50     42.36     17.44   ok
11000    214.93    237.51     45.91     21.30   ok
12000    219.33    233.44     45.76     20.81   ok
13000    217.52    241.45     42.49     21.29   ok
14000    231.26    249.06     45.84     19.71   ok
15000    232.12    255.98     45.87     19.60   ok
16000    239.26    250.89     45.58     22.61   ok
17000    240.74    257.13     45.69     23.15   ok
18000    242.45    265.05     45.75     19.09   ok
```

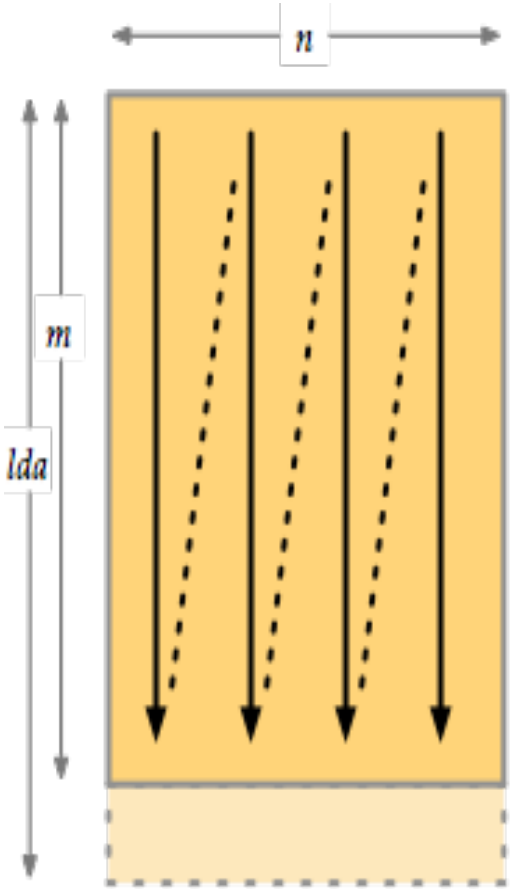
BLAS and auxiliary routines

Category	**Operation**	**Routine** (all GPU interface)
Level 1 BLAS	$y = \alpha x + y$	daxpy
	$r = x^{\mathrm{T}} y$	ddot
Level 2 BLAS	$y = \alpha Ax + \beta y$, general A	dgemv
	$y = \alpha Ax + \beta y$, symmetric A	dsymv
Level 3 BLAS	$C = \alpha AB + \beta C$	dgemm
	$C = \alpha AB + \beta C$, symmetric A	dsymm
	$C = \alpha AA^T + \beta C$, symmetric C	dsyrk
Auxiliary	$\|A\|_1$, $\|A\|_{\mathrm{inf}}$, $\|A\|_{\mathrm{fro}}$, $\|A\|_{\max}$	dlange (norm, general A)
		dlansy (norm, symmetric A)
	$B = A^{\mathrm{T}}$ (out-of-place)	dtranspose
	$A = A^{\mathrm{T}}$ (in-place, square)	dtranspose_inplace

Selected routines; complete documentation at http://icl.utk.edu/magma/

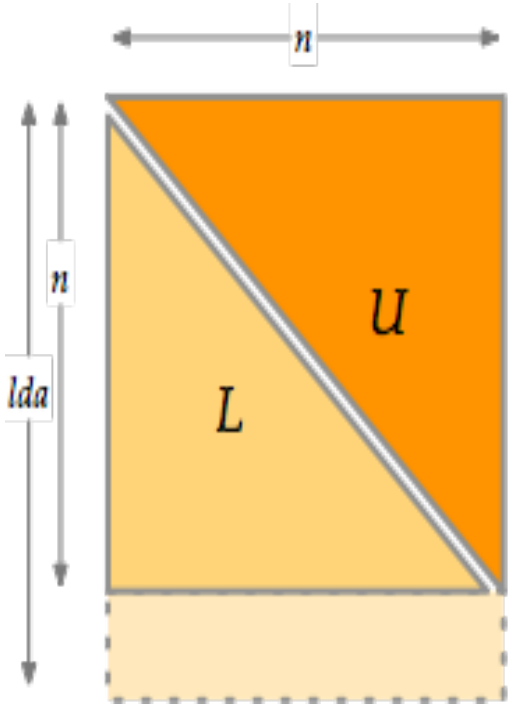
Matrix layout

General m-by-n matrix, LAPACK column-major layout



Symmetric / Hermitian / Triangular n-by-n matrix

- uplo = Lower or Upper
- Entries in opposite triangle ignored



Simple example

- Solve $AX = B$
  - **D**ouble precision, **GE**neral matrix, **S**ol**V**e (**DGESV**)
- Traditional LAPACK call
- Complete codes available at bit.ly/magma-tutorial

```
// tutorial0_lapack.cc
#include "magma_lapack.h"

int main( int argc, char** argv )
{
    int n = 100, nrhs = 10;
    int lda = n, ldx = n;
    double *A = new double[ lda*n ];
    double *X = new double[ ldx*nrhs ];
    int* ipiv = new int[ n ];

    // ... fill in A and X with your data
    // A[ i + j*lda ] = A_ij
    // X[ i + j*ldx ] = X_ij

    // solve AX = B where B is in X
    int info;
    lapackf77_dgesv( &n, &nrhs,
                     A, &lda, ipiv,
                     X, &ldx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in X

    delete[] A;
    delete[] X;
    delete[] ipiv;
}
```

Simple example

- MAGMA CPU interface
  - Input & output matrices in CPU host memory
- Add MAGMA init & finalize
- MAGMA call direct replacement for LAPACK call

```
// tutorial1_cpu_interface.cc
#include "magma_v2.h"

int main( int argc, char** argv )
{
    magma_init();

    int n = 100, nrhs = 10;
    int lda = n, ldx = n;
    double *A = new double[ lda*n ];
    double *X = new double[ ldx*nrhs ];
    int* ipiv = new int[ n ];

    // ... fill in A and X with your data
    // A[ i + j*lda ] = A_ij
    // X[ i + j*ldx ] = X_ij

    // solve AX = B where B is in X
    int info;
    magma_dgesv( n, nrhs,
                 A, lda, ipiv,
                 X, ldx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in X

    delete[] A;
    delete[] X;
    delete[] ipiv;

    magma_finalize();
}
```

Simple example

- MAGMA GPU interface
  - Add **_gpu** suffix
  - Input & output matrices in GPU device memory ("d" prefix on variables)
  - ipiv still in CPU memory
  - Set GPU stride (ldda) to multiple of 32 for better performance
  - roundup returns ceil (n / 32) * 32
- MAGMA malloc & free
  - Type-safe wrappers around cudaMalloc & cudaFree

```
// tutorial2_gpu_interface.cc
int main( int argc, char** argv )
{
    magma_init();

    int n = 100, nrhs = 10;
    int ldda = magma_roundup( n, 32 );
    int lddx = magma_roundup( n, 32 );
    int* ipiv = new int[ n ];

    double *dA, *dX;
    magma_dmalloc( &dA, ldda*n );
    magma_dmalloc( &dX, lddx*nrhs );
    assert( dA != nullptr );
    assert( dX != nullptr );

    // ... fill in dA and dX (on GPU)

    // solve AX = B where B is in X
    int info;
    magma_dgesv_gpu( n, nrhs,
                     dA, ldda, ipiv,
                     dX, lddx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in dX

    magma_free( dA );
    magma_free( dX );
    delete[] ipiv;

    magma_finalize();
}
```

BLAS example

- Matrix multiply $C = -AB + C$
  - **D**ouble-precision, **GE**neral **M**atrix **M**ultiply (**DGEMM**)
- Asynchronous
  - BLAS take queue and are async
  - Return to CPU immediately
  - Queue wraps CUDA stream and cuBLAS handle
  - Can create queue from existing CUDA stream and cuBLAS handle, if required

```
// tutorial3_blas.cc
int main( int argc, char** argv )
{
    // ... setup matrices on GPU:
    // m-by-k matrix dA,
    // k-by-n matrix dB,
    // m-by-n matrix dC.

    int device;
    magma_queue_t queue;
    magma_getdevice( &device );
    magma_queue_create( device, &queue );

    // C = -A B + C
    magma_dgemm( MagmaNoTrans,
                 MagmaNoTrans, m, n, k,
                 -1.0, dA, ldda,
                       dB, lddb,
                  1.0, dC, lddc, queue );

    // ... do concurrent work on CPU

    // wait for gemm to finish
    magma_queue_sync( queue );

    // ... use result in dC

    magma_queue_destroy( queue );

    // ... cleanup
}
```

Copy example

- Copy data host ⇔ device
  - setmatrix (host to device)
  - getmatrix (device to host)
  - copymatrix (device to device)
  - setvector (host to device)
  - getvector (device to host)
  - copyvector (device to device)
- Default is synchronous
  - Return when transfer is done
- Strides (lda, ldda) can differ on CPU and GPU
  - Set GPU stride (ldda) to multiple of 32 for better performance

```
// tutorial4_copy.cc
int main( int argc, char** argv )
{
    // ... setup A, X in CPU memory;
    // dA, dX in GPU device memory

    int device;
    magma_queue_t queue;
    magma_getdevice( &device );
    magma_queue_create( device, &queue );

    // copy A, X to dA, dX
    magma_dsetmatrix( n, n,
                      A, lda,
                      dA, ldda, queue );
    magma_dsetmatrix( n, nrhs,
                      X, ldx,
                      dX, lddx, queue );

    // ... solve AX = B

    // copy result dX to X
    magma_dgetmatrix( n, nrhs,
                      dX, lddx,
                      X, ldx, queue );

    // ... use result in X

    magma_queue_destroy( queue );

    // ... cleanup
}
```

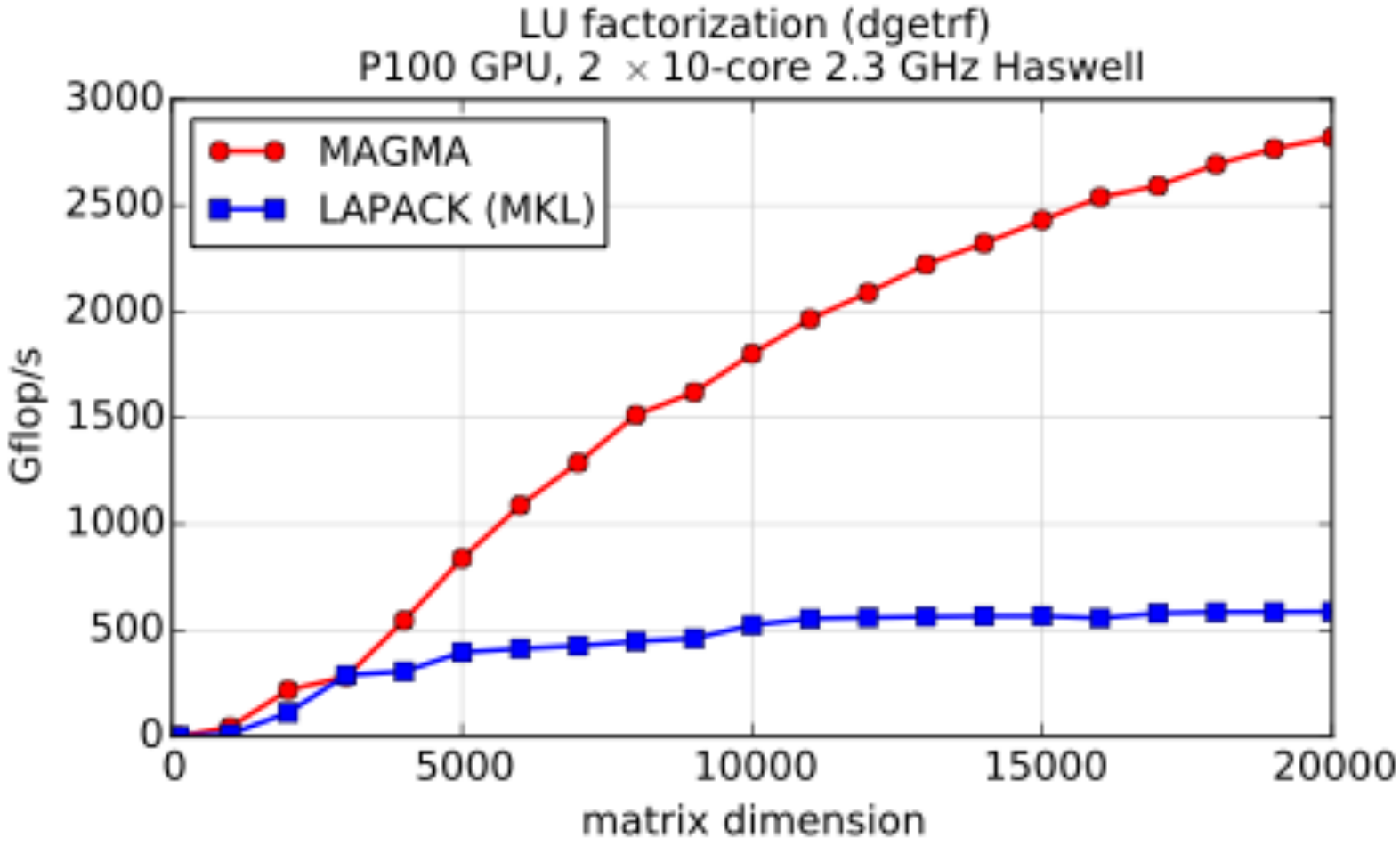
Async copy

- Add **_async** suffix
- Use pinned CPU memory
  - Page locked, so DMA can access it
  - Better performance
  - Required by CUDA for async behavior
  - But pinned memory is limited resource, and expensive to allocate
- Overlap:
  - Sending data (host to device)
  - Getting data (device to host)
  - Host computation
  - Device computation

```
// tutorial5_copy_async.cc
int main( int argc, char** argv )
{
    // ... setup dA, dX, queue

    // allocate A, X in pinned CPU memory
    double *A, *X;
    magma_dmalloc_pinned( &A, lda*n );
    magma_dmalloc_pinned( &X, ldx*nrhs );

    // ... fill in A and X

    // copy A, X to dA, dX, then wait
    magma_dsetmatrix_async( n, n,
            A, lda, dA, ldda, queue );
    magma_dsetmatrix_async( n, nrhs,
            X, ldx, dX, lddx, queue );
    magma_queue_sync( queue );

    // ... solve AX = B

    // copy result dX to X, then wait
    magma_dgetmatrix_async( n, nrhs,
            dX, ldx, X, lddx, queue );
    magma_queue_sync( queue );

    // ... use result in X

    magma_free_pinned( A );
    magma_free_pinned( X );

    // ... cleanup
}
```

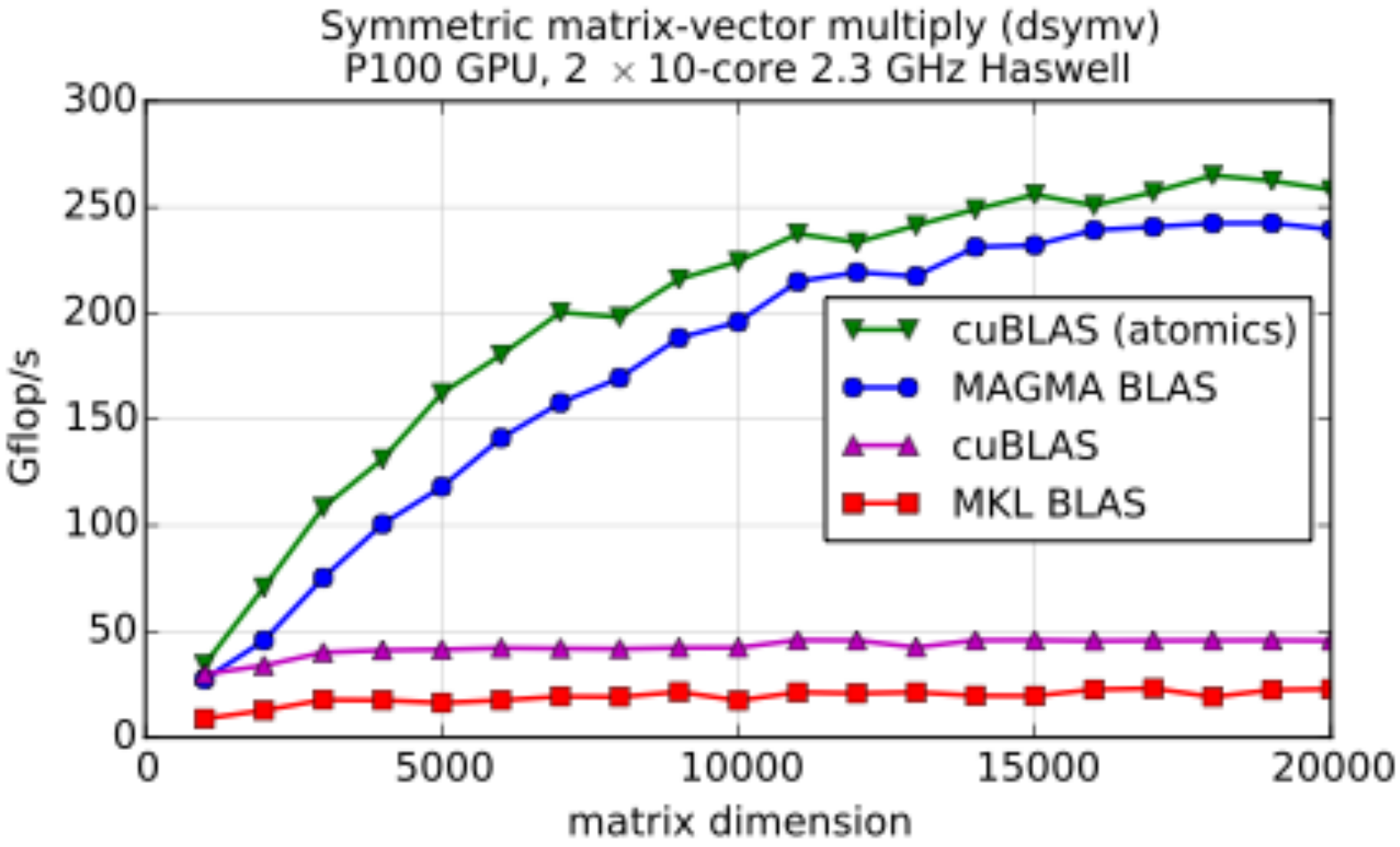
```
magma> cd testing
magma/testing> ./testing_dgetrf -n 123 -n 1000:20000:1000 --
lapack --check
% MAGMA 2.2.0  compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% CUDA runtime 8000, driver 9000. OpenMP threads 20. MKL 2017.0.1, MKL threads 20.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16276.2 MiB memory, capability 6.0

%   M     N   CPU Gflop/s (sec)   GPU Gflop/s (sec)   |PA-LU|/(N*|A|)
%===========================================================================
  123   123      0.20 (   0.01)      0.40 (   0.00)   3.59e-18   ok       # warmup run
 1000  1000     10.40 (   0.06)     43.50 (   0.02)   2.76e-18   ok
 2000  2000    111.64 (   0.05)    218.26 (   0.02)   2.68e-18   ok
 3000  3000    288.38 (   0.06)    280.28 (   0.06)   2.65e-18   ok
 4000  4000    305.58 (   0.14)    545.90 (   0.08)   2.81e-18   ok
 5000  5000    396.16 (   0.21)    838.09 (   0.10)   2.71e-18   ok
 6000  6000    413.37 (   0.35)   1088.14 (   0.13)   2.71e-18   ok
 7000  7000    426.71 (   0.54)   1288.60 (   0.18)   2.67e-18   ok
 8000  8000    447.85 (   0.76)   1514.43 (   0.23)   2.66e-18   ok
 9000  9000    461.05 (   1.05)   1621.29 (   0.30)   2.87e-18   ok
10000 10000    524.06 (   1.27)   1802.39 (   0.37)   2.84e-18   ok
11000 11000    554.16 (   1.60)   1965.85 (   0.45)   2.84e-18   ok
12000 12000    559.33 (   2.06)   2090.42 (   0.55)   2.82e-18   ok
13000 13000    563.56 (   2.60)   2223.62 (   0.66)   2.80e-18   ok
14000 14000    566.58 (   3.23)   2323.04 (   0.79)   2.78e-18   ok
15000 15000    567.17 (   3.97)   2431.59 (   0.93)   2.77e-18   ok
16000 16000    556.86 (   4.90)   2539.66 (   1.08)   2.79e-18   ok
17000 17000    579.82 (   5.65)   2593.40 (   1.26)   2.75e-18   ok
18000 18000    584.93 (   6.65)   2694.57 (   1.44)   2.76e-18   ok
19000 19000    585.78 (   7.81)   2768.67 (   1.65)   2.75e-18   ok
20000 20000    587.08 (   9.08)   2821.48 (   1.89)   2.74e-18   ok
```

Testers: LU factorization (dgetrf)



```
# (abbreviated output)
magma> cd testing
magma/testing> ./testing_dsymv -n 123 -n 1000:20000:1000 --lapack --check
% MAGMA 2.5.0  compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% CUDA runtime 8000, driver 9000. OpenMP threads 20. MKL 2017.0.1, MKL threads 20.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16276.2 MiB memory, capability 6.0

% uplo = Lower
%   N     MAGMA   Atomics    cuBLAS       CPU   error
%       Gflop/s   Gflop/s   Gflop/s   Gflop/s
%===================================================
  123      0.76      0.76      0.51      0.58   ok    # warmup run
 1000     27.44     34.41     29.88      8.86   ok
 2000     45.74     70.83     33.91     12.78   ok
 3000     75.30    108.51     40.09     17.76   ok
 4000    100.64    131.23     41.13     17.57   ok
 5000    118.17    162.35     41.46     16.33   ok
 6000    141.21    180.43     42.16     17.55   ok
 7000    157.81    200.44     41.94     19.32   ok
 8000    169.54    198.21     41.78     19.12   ok
 9000    188.40    216.07     42.28     21.50   ok
10000    195.92    224.50     42.36     17.44   ok
11000    214.93    237.51     45.91     21.30   ok
12000    219.33    233.44     45.76     20.81   ok
13000    217.52    241.45     42.49     21.29   ok
14000    231.26    249.06     45.84     19.71   ok
15000    232.12    255.98     45.87     19.60   ok
16000    239.26    250.89     45.58     22.61   ok
17000    240.74    257.13     45.69     23.15   ok
18000    242.45    265.05     45.75     19.09   ok
19000    242.53    262.48     45.81     22.42   ok
20000    239.53    258.24     45.63     22.83   ok
```

Testers: symmetric matrix-vector multiply



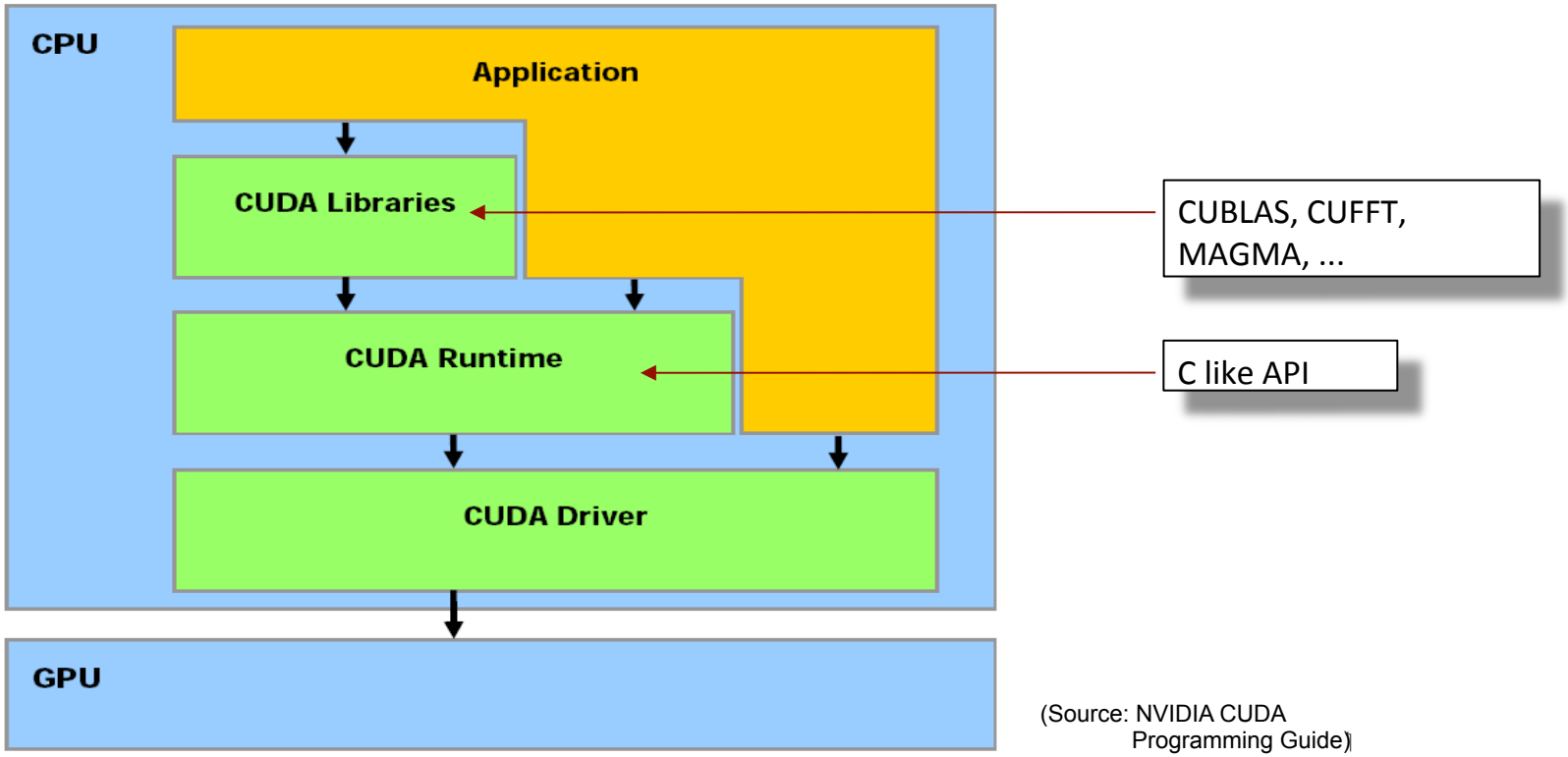
Test everything: run_tests.py

- Python script to run:
  - All testers
  - All possible options (left/right, lower/upper, ...)
  - Various size ranges (small, medium, large; square, tall, wide)
- Occasionally, tests fail innocuously
  - E.g., error = 1.1e-15 > tol = 1e-15
- Some experimental routines are known to fail
  - E.g., gegqr_gpu, geqr2x_gpu
  - See magma/BUGS.txt
- Running ALL tests can take > 24 hours

Test everything: run_tests.py

```
magma/testing> python ./run_tests.py *trsm --xsmall --small > trsm.txt
testing_strsm -SL -L -DN -c       ok  # left, lower, non-unit, [no-trans]
testing_dtrsm -SL -L -DN -c       ok
testing_ctrsm -SL -L -DN -c       ok
testing_ztrsm -SL -L -DN -c       ok
testing_strsm -SL -L -DU -c       ok  # left, lower, unit, [no-trans]
testing_dtrsm -SL -L -DU -c       ok
testing_ctrsm -SL -L -DU -c       ok
testing_ztrsm -SL -L -DU -c       ok
testing_strsm -SL -L -C -DN -c    ok  # left, lower, non-unit, conj-trans
testing_dtrsm -SL -L -C -DN -c    ok
testing_ctrsm -SL -L -C -DN -c    ok
testing_ztrsm -SL -L -C -DN -c    ok
...
testing_strsm -SR -U -T -DU -c    ok  # right, upper, unit, trans
testing_dtrsm -SR -U -T -DU -c    ok
testing_ctrsm -SR -U -T -DU -c    ok
testing_ztrsm -SR -U -T -DU -c    ok

*********************************************************************************************
summary
*********************************************************************************************
 6240 tests in 192 commands passed
   96 tests failed accuracy test
    0 errors detected (crashes, CUDA errors, etc.)
routines with failures:
    testing_ctrsm --ngpu 2 -SL -L -C -DN -c
    testing_ctrsm --ngpu 2 -SL -L -C -DU -c
...
```

CUDA Software Stack



(Source: NVIDIA CUDA Programming Guide)

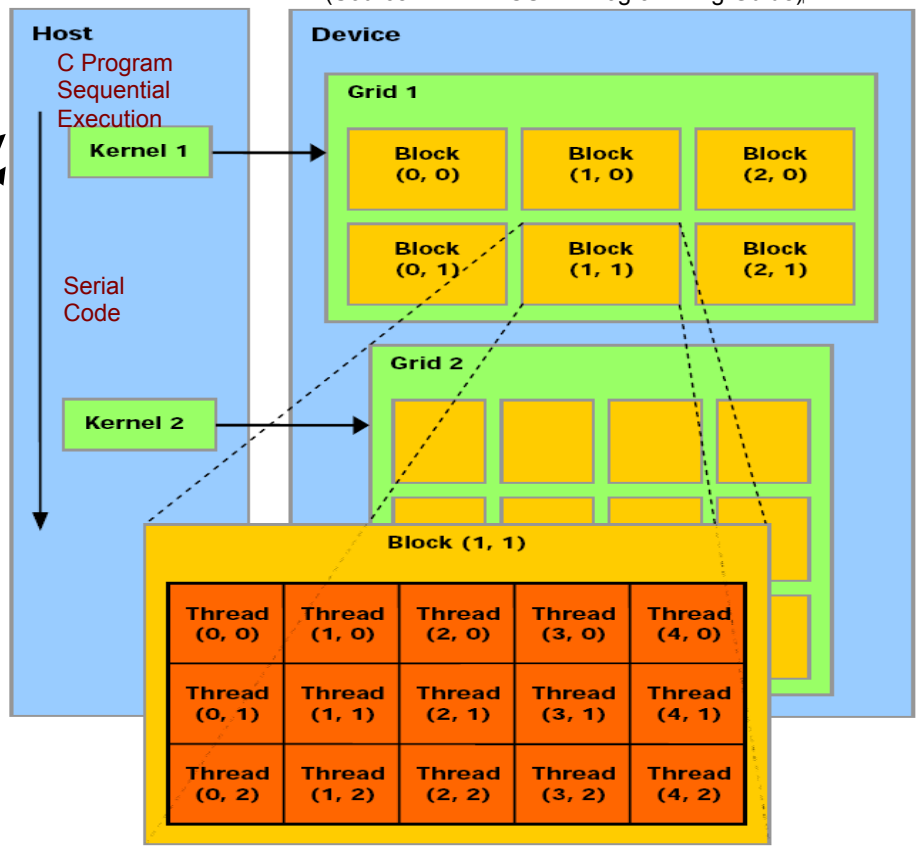
CUDA Programming Model

- Grid of thread blocks
  (blocks of the same dimension, grouped together to execute the same kernel)
- Thread block
  (a batch of threads with fast shared memory executes a kernel)
- Sequential code launches asynchronously GPU kernels

```
// set the grid and thread configuration
Dim3 dimGrid(2,3);
Dim3 dimTBlock(3,5);

// Launch the device computation
MatVec<<<dimGrid, dimTBlock>>>( . . . );
```

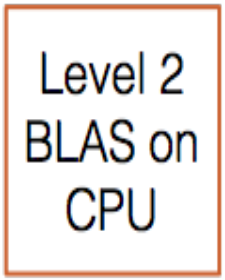
```
__global__ void MatVec( . . . ) {
  // Block index
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Thread index
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  . . .
}
```

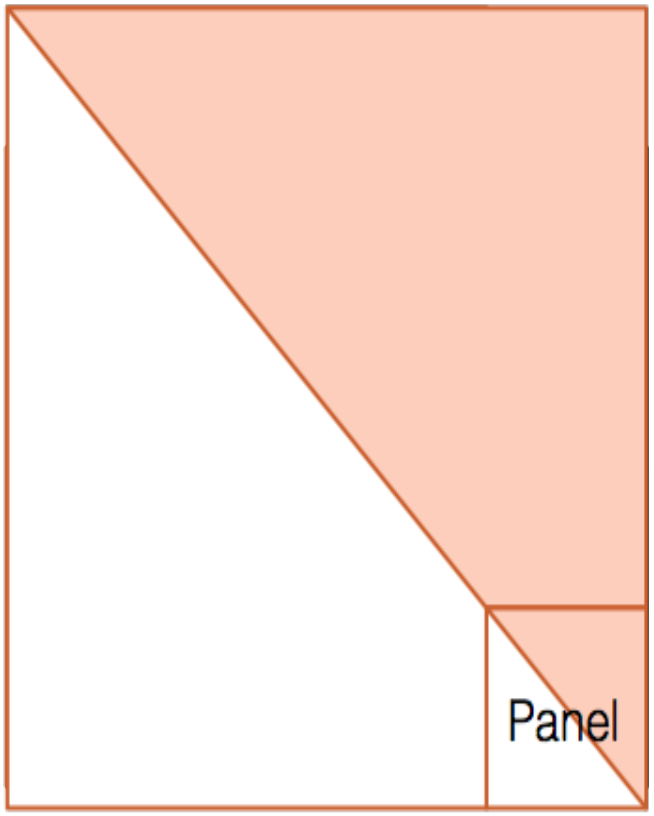
(Source: NVIDIA CUDA Programming Guide)



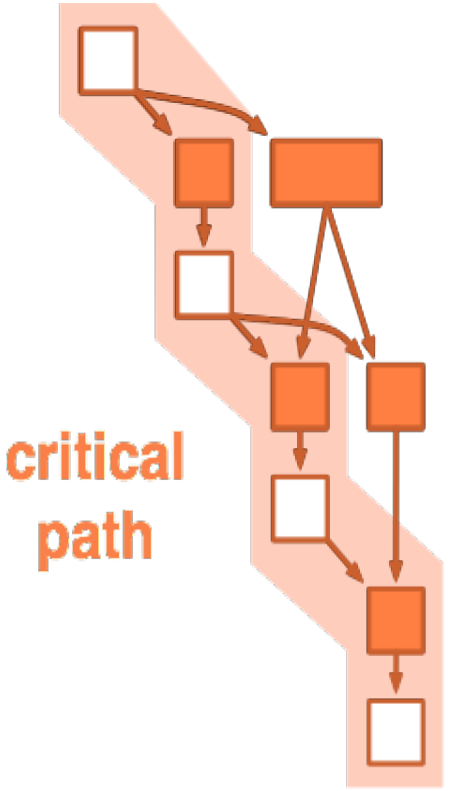
One-sided factorizations

- LU, Cholesky, QR factorizations for solving linear systems



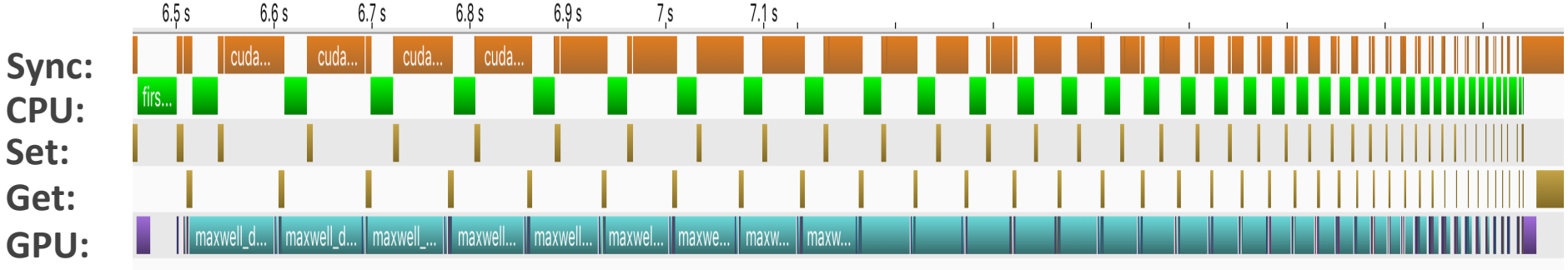


DAG



Execution trace

- **Panels** on CPU (green) and **set/get communication** (brown) overlapped with **trailing matrix updates** (teal) on GPU
- Goal to keep GPU busy all the time; CPU may idle

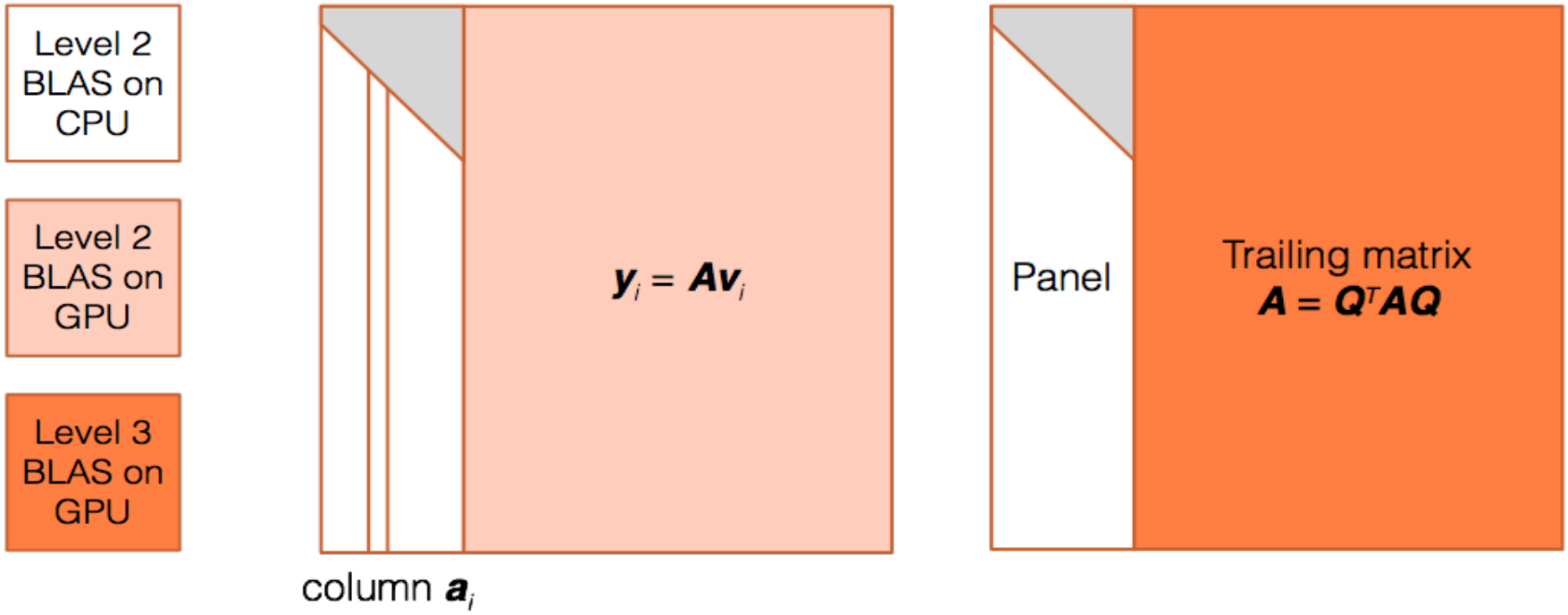


LU factorization (dgetrf), n = 20000
P100 GPU, 2 × 10-core 2.3 GHz Haswell

- Optimization: for LU, we transpose matrix on GPU so row-swaps are fast

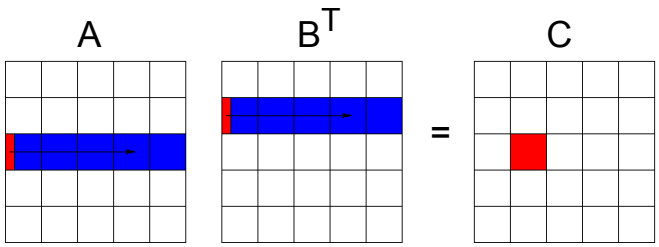
Two-sided factorizations

- Hessenberg, tridiagonal, bidiagonal factorizations for eigenvalue and singular value problems



SGEMM Example

2008. Volkov and Demmel. ***Benchmarking GPUs to tune dense linear algebra***, SC08
http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf

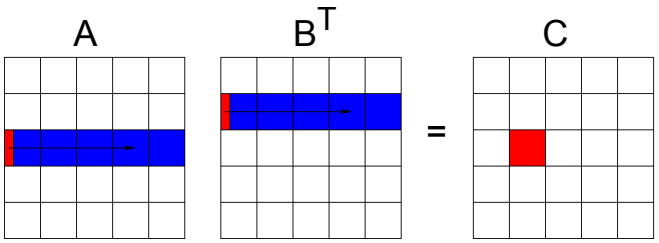


* **Small** red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads

```
// GPU kernel: compute C = alpha A B' + beta C
__global__ void sgemmNT( const float *A, int lda,
                         const float *B, int ldb,
                         float* C, int ldc, int k,
                         float alpha, float beta  )
{
    int inx = threadIdx.x;
    int iny = threadIdx.y;
    int ibx = blockIdx.x * 32;
    int iby = blockIdx.y * 32;

    A += ibx + inx + __mul24( iny, lda );
    B += iby + inx + __mul24( iny, ldb );
    C += ibx + inx + __mul24( iby + iny, ldc );

    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    for( int i = 0; i < k; i += 4 )
    {
        __syncthreads();
        __shared__ float a[4][32];
        __shared__ float b[4][32];

        a[iny][inx] = A[i*lda];
        a[iny+2][inx] = A[(i+2)*lda];

        b[iny][inx]   = B[i*ldb];
        b[iny+2][inx] = B[(i+2)*ldb];
        __syncthreads();

        for( int j = 0; j < 4; j++ )
        {
            float _a = a[j][inx];
            float *_b = &b[j][0] + iny;

            c[0] += _a*_b[0];
            c[1] += _a*_b[2];
            c[2] += _a*_b[4];
            c[3] += _a*_b[6];
            c[4] += _a*_b[8];
            c[5] += _a*_b[10];
            c[6] += _a*_b[12];
            c[7] += _a*_b[14];
            c[8] += _a*_b[16];
            c[9] += _a*_b[18];
            c[10] += _a*_b[20];
            c[11] += _a*_b[22];
            c[12] += _a*_b[24];
            c[13] += _a*_b[26];
            c[14] += _a*_b[28];
            c[15] += _a*_b[30];
        }
    }

    for( int i = 0; i < 16; i++, C += 2*ldc )
        C[0] = alpha * c[i] + beta * C[0];
}


void ourSgemm (char transa, char transb,
               int m, int n, int k,
               float alpha,
               const float *A, int lda,
               const float *B, int ldb,
               float beta,
               float *C, int ldc)
{
    assert( (transa == 'N' || transa == 'n') &&
            (transb == 'T' || transb == 't') &&
            ((m|n|k|lda|ldb)&31) == 0,
            "unsupported parameters in ourSgemm()" );

    dim3 grid( m/32, n/32, 1 );
    dim3 threads2( 32, 2, 1 );
    sgemmNT<<<grid, threads2>>>( A, lda,
                                 B, ldb,
                                 C, ldc,
                                 k, alpha, beta );
}
```

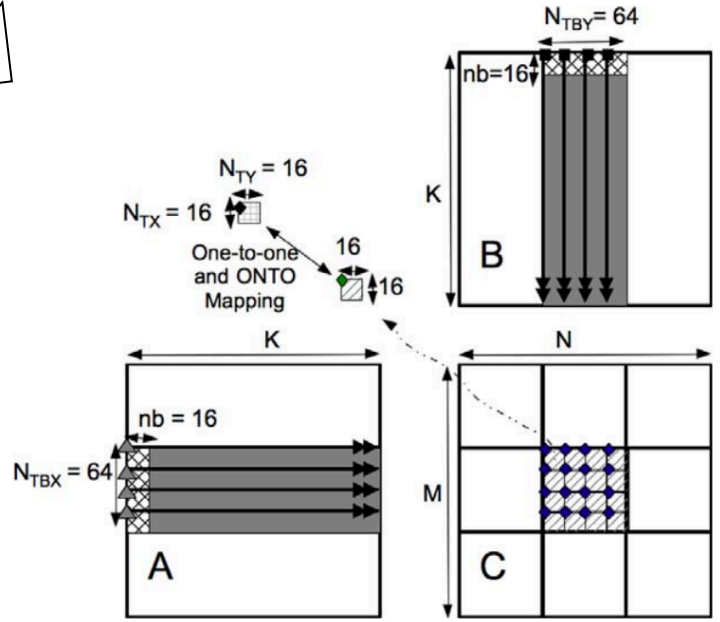
GEMM in MAGMA

2010. R. Nath, S. Tomov and J. Dongarra. ***An improved MAGMA GEMM for Fermi Graphics Processing Units***, The International Journal of High Performance Computing Applications.
http://www.netlib.org/utk/people/JackDongarra/journals/208_2010_an-improved-magma-gemm-for-fermi-gpus.pdf



* **Small** red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads

- **Add register blocking**
- **Parameterized for autotuning for particular size GEMMs and portability across GPUs**



A thread computes part of a row (16 values) of the C block

A thread computes a block of C (4 x 4 values in this case)

CUDA vs. HIP

CUDA	HIP
cudaMalloc, cudaMemcpy, ... ->	hipMalloc, hipMemcpy, ...
cuCadd, cuCsub, ... ->	hipCadd, hipCsub, ...
func<<<B, T, sz, st>>>(a, b); ->	hipLaunchKernelGGL(func, B, T, sz, st, a, b);
extern __shared__ T name; ->	HIP_DYNAMIC_SHARED(T, name); (*)

(*) Must be declared inside a function, and per function

Hipify

There is a tool supplied by AMD, called ‘hipify’ (actually, there are 2; hipify-perl & hipify-clang)

‘hipify-perl’ performs basic substitution, i.e. ‘cudaMemcpy -> hipMemcpy’, and not much else. It has some trouble with macros & nested expressions

‘hipify-clang’ is built on top of LLVM’s parser/compiler interface, so can handle any amount of parentheses/calls/etc. However, it is harder to tweak & install, as ‘hipify-perl’ is just a perl script with regex

Let’s look at an example, which is performing a 2D box filter

CUDA Box Filter (device)

```
1   __global__
2   void mykernel(int M, int N, const float* inp, float* out) {
3       int i = blockIdx.x*blockDim.x + threadIdx.x;
4       int j = blockIdx.y*blockDim.y + threadIdx.y;
5       if (i < 0 || i >= M || j < 0 || j >= N) return;
6
7       const int k = 1;
8       float sumr = 0.0f;
9
10      for (int px = max(0, i-k); px <= min(M-1, i+k); ++px)
11          for (int py = max(0, j-k); py <= min(N-1, j+k); ++py)
12              sumr += inp[py * M + px];
13
14      out[j * M + i] = sumr / ((2 * k + 1) * (2 * k + 1));
15  }
```

HIP Box Filter (device)

```
1   __global__
2   void mykernel(int M, int N, const float* inp, float* out) {
3       int i = blockIdx.x*blockDim.x + threadIdx.x;
4       int j = blockIdx.y*blockDim.y + threadIdx.y;
5       if (i < 0 || i >= M || j < 0 || j >= N) return;
6
7       const int k = 1;
8       float sumr = 0.0f;
9
10      for (int px = max(0, i-k); px <= min(M-1, i+k); ++px)
11          for (int py = max(0, j-k); py <= min(N-1, j+k); ++py)
12              sumr += inp[py * M + px];
13
14      out[j * M + i] = sumr / ((2 * k + 1) * (2 * k + 1));
15  }
```

CUDA Box Filter (host)

```
1   int main() {
2       int M = 20, N = 8;
3       float *x = (float*)malloc(M*N * sizeof(float));
4       float *y = (float*)malloc(M*N * sizeof(float));
5       for (int i = 0; i < M * N; ++i) x[i] = i;
6
7       float *d_x, *d_y;
8       cudaMalloc(&d_x, M*N * sizeof(float));
9       cudaMalloc(&d_y, M*N * sizeof(float));
10
11      cudaMemcpy(d_x, x, M*N * sizeof(float), cudaMemcpyHostToDevice);
12
13      dim3 dimBlock(16, 16);
14      dim3 dimGrid((M+dimBlock.x-1)/dimBlock.x, (N+dimBlock.y-1)/dimBlock.y);
15
16      mykernel<<< dimGrid, dimBlock >>>(M, N, d_x, d_y);
17
18      cudaMemcpy(y, d_y, M*N * sizeof(float), cudaMemcpyDeviceToHost);
19  }
```

HIP Box Filter (host)

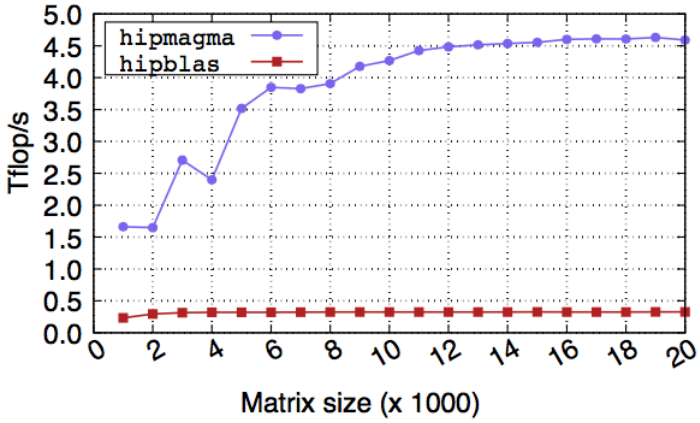
```
1   int main() {
2       int M = 20, N = 8;
3       float* x = (float*)malloc(M*N * sizeof(float));
4       float* y = (float*)malloc(M*N * sizeof(float));
5       for (int i = 0; i < M * N; ++i) x[i] = i;
6
7       float *d_x, *d_y;
8       hipMalloc(&d_x, M*N * sizeof(float));
9       hipMalloc(&d_y, M*N * sizeof(float));
10
11      hipMemcpy(d_x, x, M*N * sizeof(float), hipMemcpyHostToDevice);
12
13      dim3 dimBlock(16, 16);
14      dim3 dimGrid((M+dimBlock.x-1)/dimBlock.x, (N+dimBlock.y-1)/dimBlock.y);
15
16      hipLaunchKernelGGL(mykernel, dim3(dimGrid), dim3(dimBlock ), 0, 0, M, N, d_x, d_y);
17
18      hipMemcpy(y, d_y, M*N * sizeof(float), hipMemcpyDeviceToHost);
19  }
```

MAGMA for AMD GPUs

- Easy functional and performance portability through use of BLAS
  - Provided about 2,000 routines
    - LAPACK, BLAS, Auxiliary, Batched
  - Ported through **BLAS** and **auto-source translation tools**
  - **Added some BLAS** still missing or underperforming in hipBLAS

- MAGMA has been ported before to OpenCL and Intel Xeon Phi

DSYRK on the Mi50 GPU with ROCm 3.5



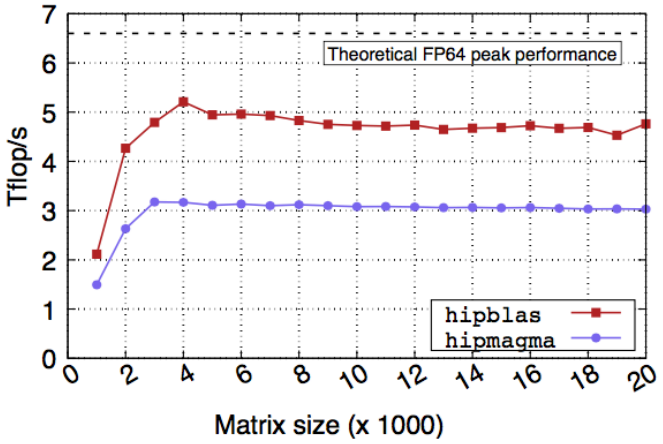
Up-coming work:

- HIP port of MAGMA Sparse
- Batched LAPACK
- GPU-only routines for LU, QR, and Cholesky
- Sync with MAGMA
- Tune multiGPU
- Improvements and autotuning for AMD GPUs

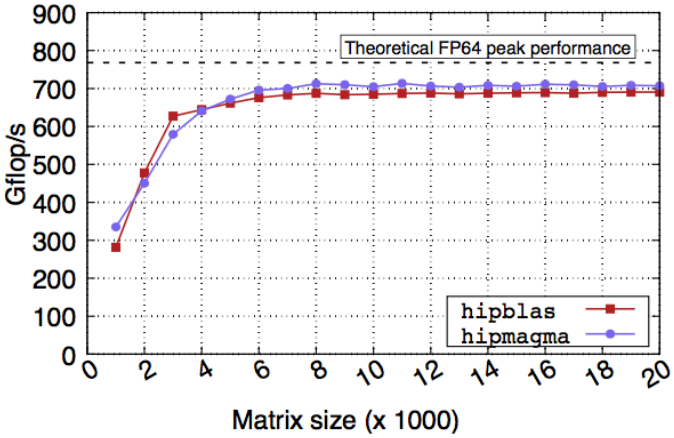
Brown, C., A. Abdelfattah, S. Tomov, and J. Dongarra, "***Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs***," Innovative Computing Laboratory Technical Report, no. ICL-UT-20-12: University of Tennessee (to appear in HPEC'20), August 2020.

Testers/benchmarks for AMD GPUs
dgemm

DGEMM on the Mi50 GPU with ROCm 3.5

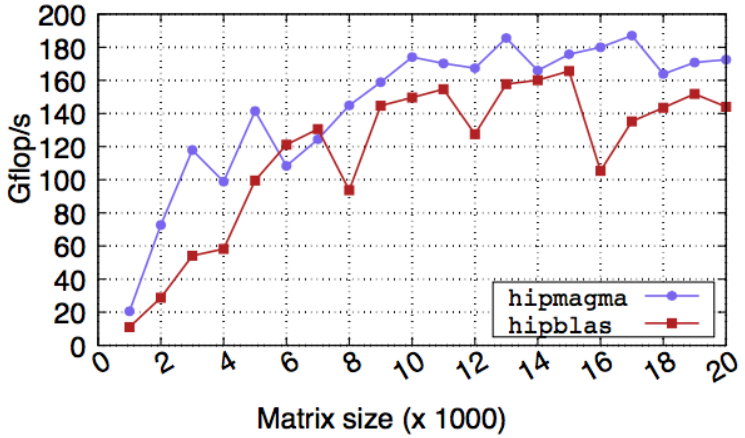


DGEMM on the Mi25 GPU with ROCm 3.5

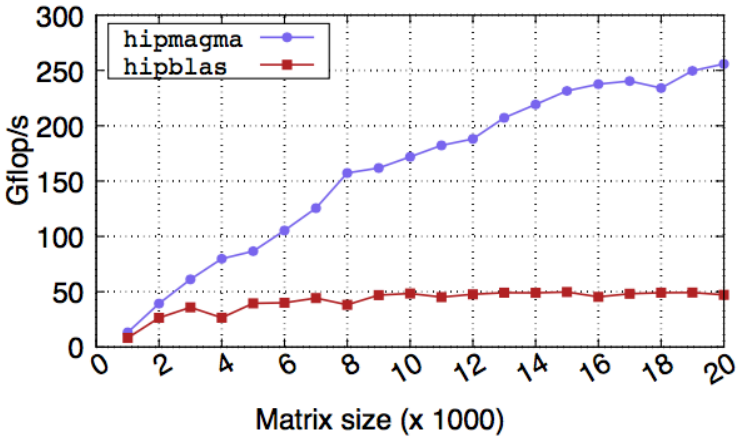


Testers/benchmarks for AMD GPUs
dgemv and dsymv

DGEMV on the Mi50 GPU with ROCm 3.5

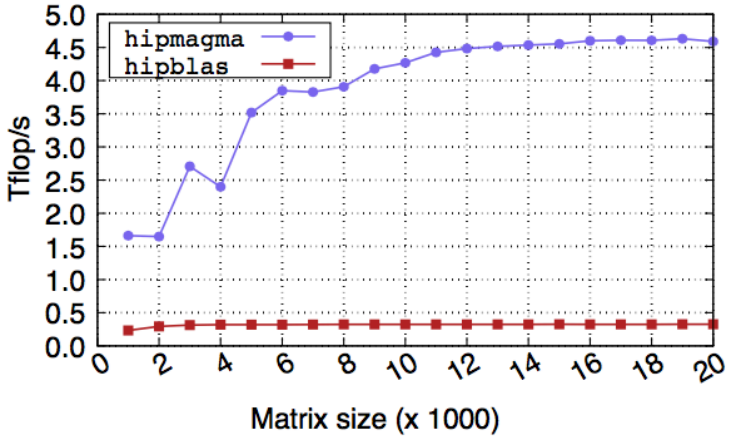


DSYMV on the Mi50 GPU with ROCm 3.5

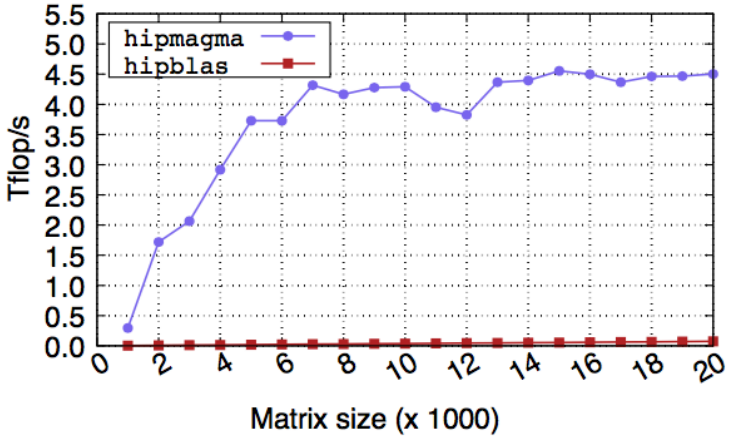


Testers/benchmarks for AMD GPUs
dgemv and dsymv

DSYRK on the Mi50 GPU with ROCm 3.5



DTRMM on the Mi50 GPU with ROCm 3.5



Mixed Precision Solvers in MAGMA

Motivation on design of mixed precision algorithms

Table 4: Parameters for the IEEE FP16, FP32, and FP64 arithmetic precisions, and their respective peak performances on NVIDIA V100 and AMD MI100 GPUs. "Range" denotes the order of magnitude of the smallest subnormal ($x_{\min,s}$), and largest and smallest positive normalized floating-point numbers. The peak 16-bit performances reported* by vendors vary depending on the instructions and special hardare acceleration used, e.g., the peak performance of 125 Tflop/s for Nvidia V100 uses FP16-TC (tensor cores) with inputs FP16, while the outputs and the computations are performed in full (FP32) precision.

Arithmetic	Size (bits)	Range			Unit roundoff	Peak Tflop/s	
		$x_{\min,s}$	$x_{\min}$	$x_{\max}$		V100	MI100
BFloat16	16	9.2×10^{-41}	1.2×10^{-38}	3.4×10^{38}	3.9×10^{-3}	N/A	92*
FP16	16	6.0×10^{-8}	6.1×10^{-5}	6.6×10^{4}	4.9×10^{-4}	125*	184*
FP32	32	1.4×10^{-45}	1.2×10^{-38}	3.4×10^{38}	6.0×10^{-8}	15.7	23
FP64	64	4.9×10^{-324}	2.2×10^{-308}	1.8×10^{308}	1.1×10^{-16}	7.8	11.5

Mixed precision iterative refinement

- Iterative refinement for dense systems, $Ax = b$, can work this way.

L U = lu(A)	SINGLE	$O(n^3)$
x = L\(U\b)	SINGLE	$O(n^2)$
r = b – Ax	DOUBLE	$O(n^2)$
WHILE || r || not small enough		
z = L\(U\r)	SINGLE	$O(n^2)$
x = x + z	DOUBLE	$O(n^1)$
r = b – Ax	DOUBLE	$O(n^2)$
END		

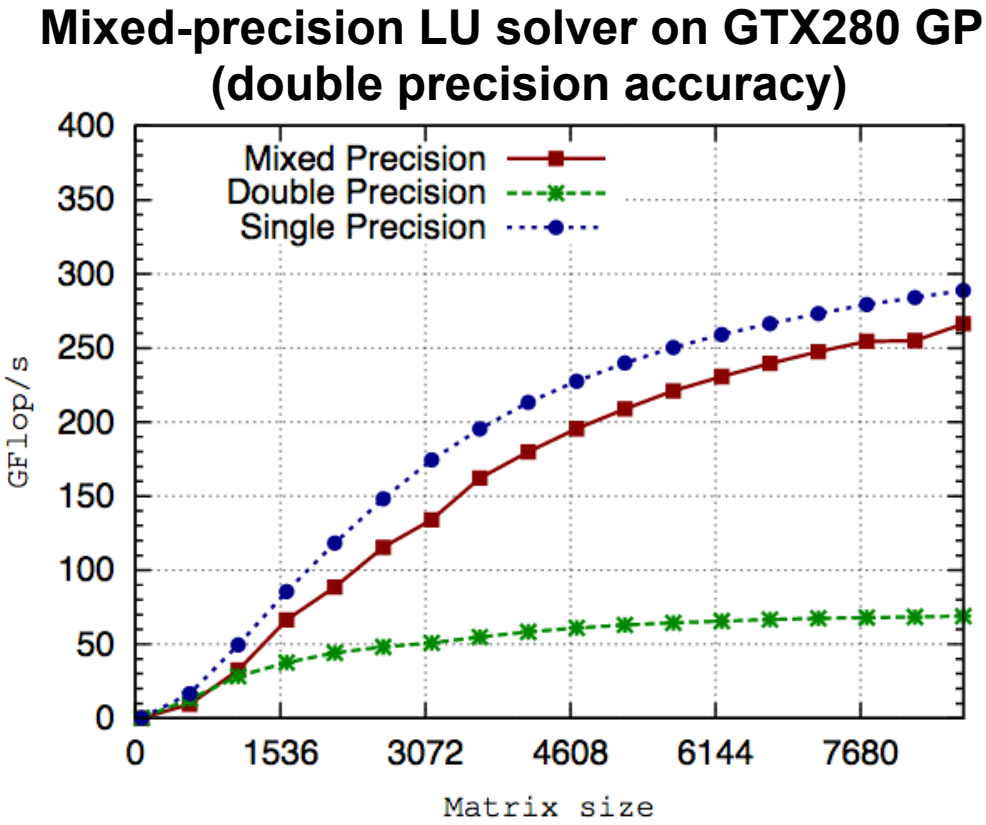
- Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.
- It can be shown that using this approach we can compute the solution to 64-bit floating point precision.

- Requires extra storage, total is 1.5 times normal;
- $O(n^3)$ work is done in lower precision
- $O(n^2)$ work is done in high precision
- Problems if the matrix is ill-conditioned in sp; $O(10^8)$

Mixed precision iterative refinement solvers

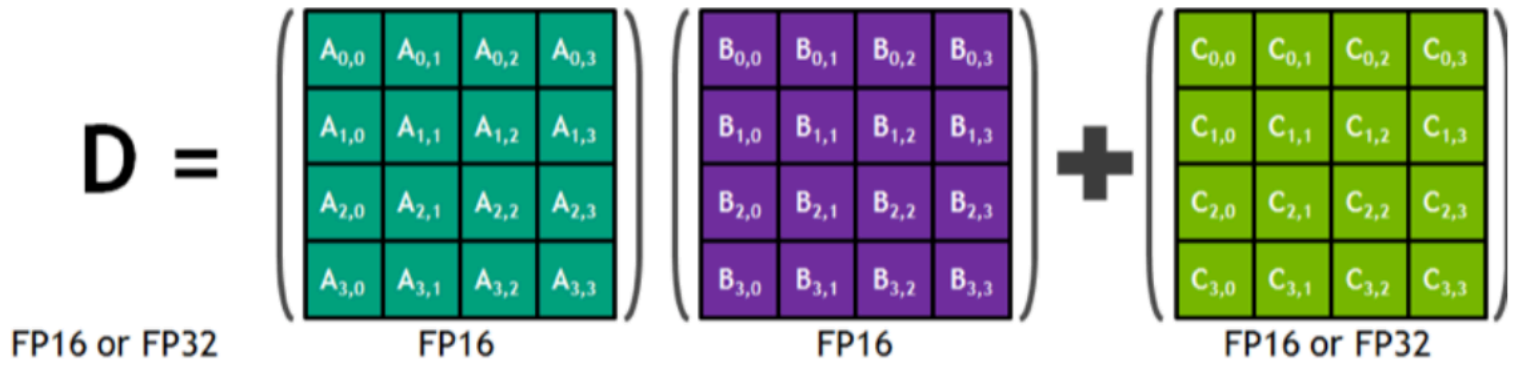
- Implemented mixed-precision iterative refinement solvers based on LU, Cholesky, and QR factorizations in MAGMA since 2010
- Algorithms and stopping criteria follow LAPACK (Langou et al. 2006)

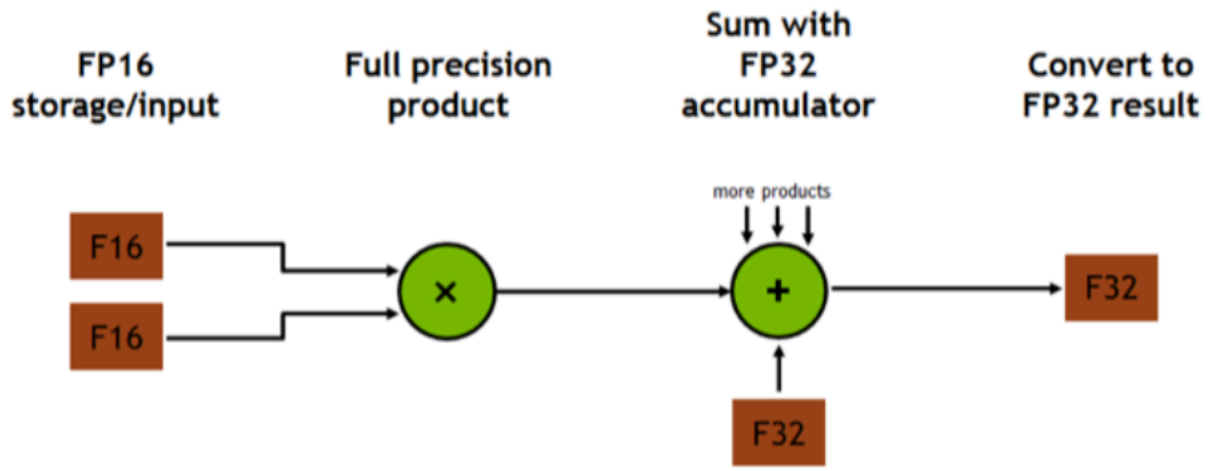
$\|Ax-b\|/(\|A\|\|x\|) < \varepsilon^w\sqrt{n}$, where ε^w is the machine precision at the working precision



GPU Tensor Cores – accelerated FP16 matrix-multiply-and-accumulate units

- Up to 120 teraFLOP/s on NVIDIA Volta V100 GPUs



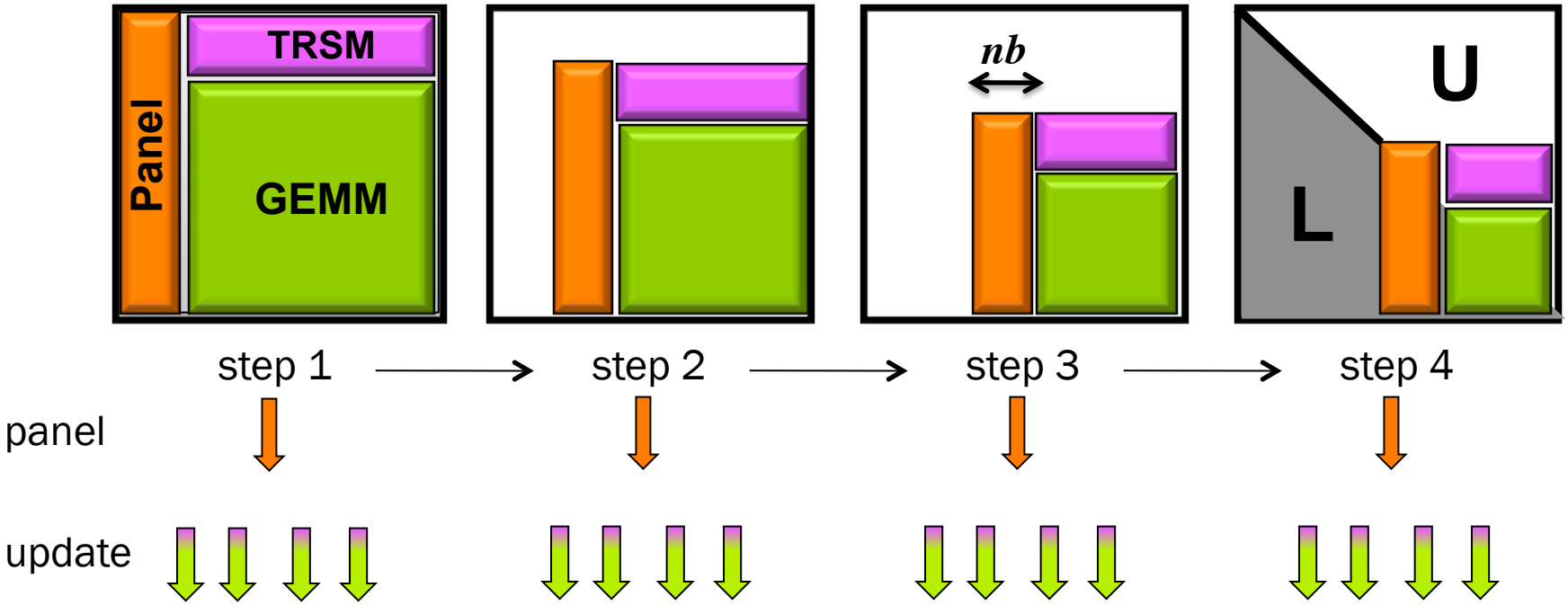


Tensor Core Accelerated IRS solving linear system Ax = b

For s = 0, nb, .. N

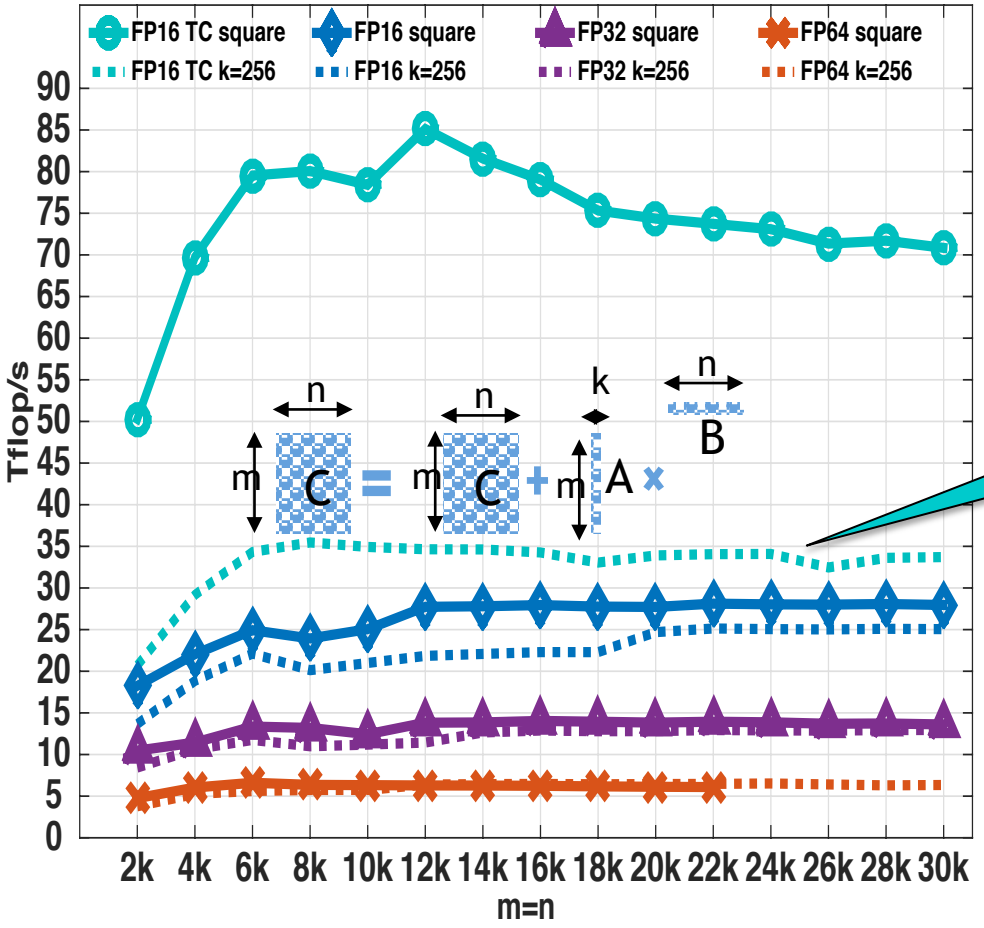
1. panel factorize
2. update trailing matrix

LU factorization requires $O(n^3)$ most of the operations are spent in GEMM



Tensor Core Accelerated IRS Motivation

Study of the Matrix Matrix multiplication kernel on Nvidia V100



- dgemm achieve about 6.4 Tflop/s
- sgemm achieve about 14 Tflop/s
- hgemm achieve about 27 Tf
- T about

Rank-k GEMM needed by LU does not perform as well as square but still OK

Matrix matrix multiplication GEMM

$C = \alpha A B + \beta C$

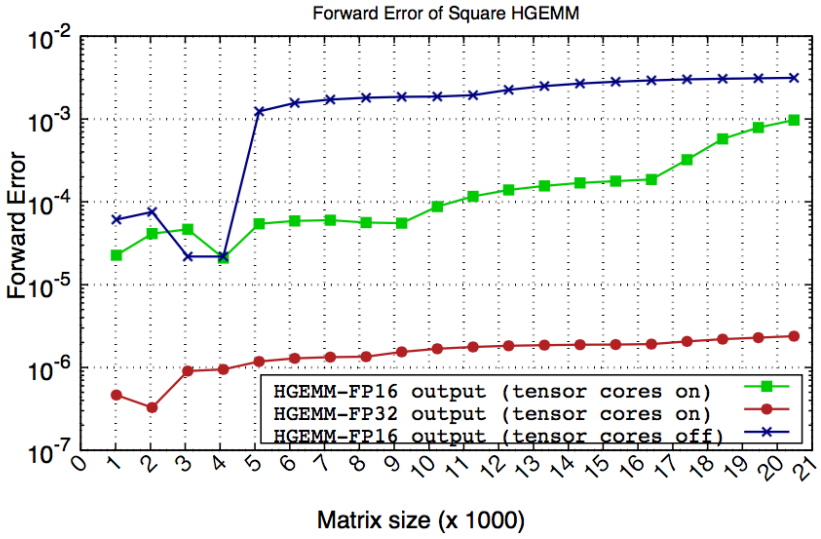
Mixed-precision factorizations

Input: A in precision $\mathbf{u}^{\mathbf{f_h}}$=FP32
$\mathbf{u}^{\mathbf{f_l}} \leq \mathbf{u}^{\mathbf{f_h}}$
for $P_i \in \{P_1, P_2, \ldots, P_n\}$ **do**
  PanelFactorize P_i in precision $\mathbf{u}^{\mathbf{f_h}}$ (e.g., FP32)
  Triangular solve T_i in precision $\mathbf{u}^{\mathbf{f_h}}$ (e.g., FP32)
  convert $P_i^{\mathbf{f_h}} \rightarrow P_i^{\mathbf{f_l}}$ (e.g., from precision FP32 to FP16)
  convert $T_i^{\mathbf{f_h}} \rightarrow T_i^{\mathbf{f_l}}$ (e.g., from precision FP32 to FP16)
  TrailingMatrixUpdate $A_i^{\mathbf{f_h}} = A_i^{\mathbf{f_h}} - P_i^{\mathbf{f_l}} T_i^{\mathbf{f_l}}$ (e.g., $A_i^{\text{FP32}} = A_i^{\text{FP32}} - P_i^{\text{FP16}} T_i^{\text{FP16}}$) using tensor cores

- A is never converted to FP16
- Sensitive for accuracy panel factorization and triangular solve are done in FP32
- For many problems of interest we get about two more digits of accuracy compared to factorizations using fixed FP16 arithmetic!

Mixed-precision HGEMM accuracy

V100 GPU



MI100 GPU

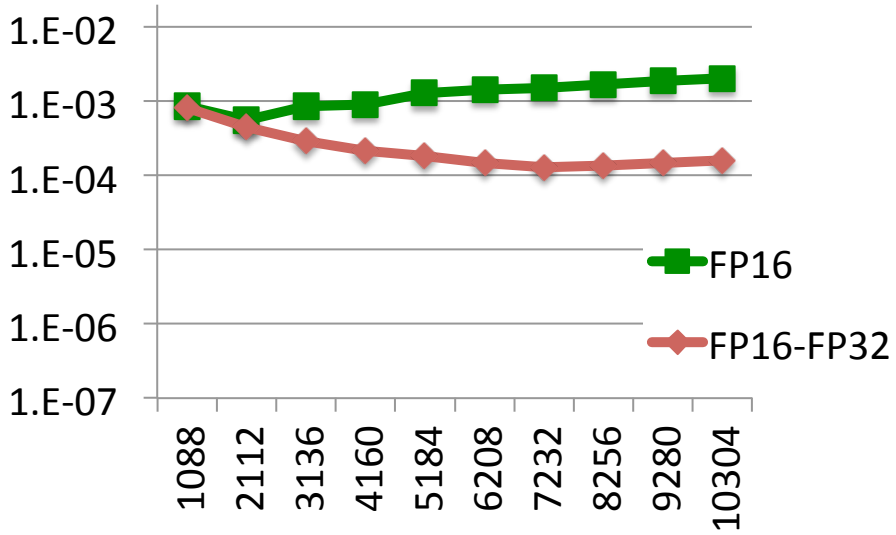
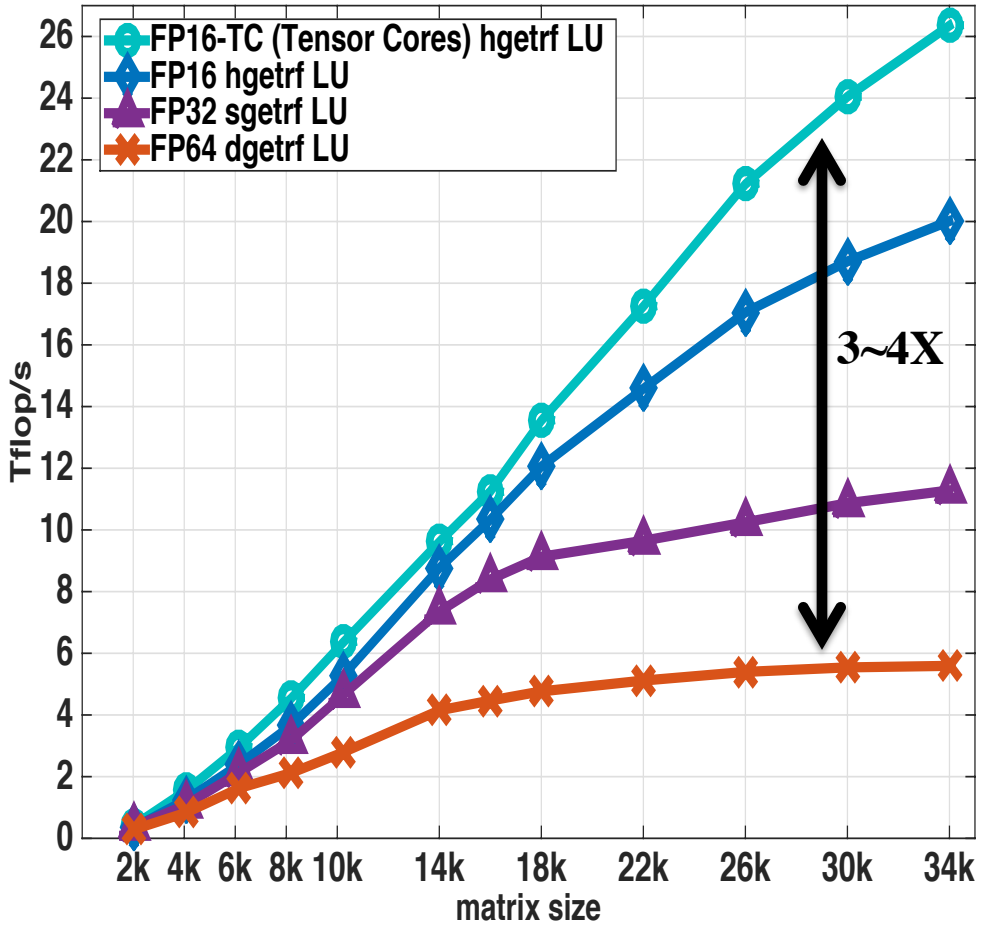


Figure 3: Forward error of HGEMM with respect to MKL SGEMM ($C = \alpha AB + \beta C$). Results are shown for square sizes using cuBLAS 9.1 and MKL 2018.1. The forward error is computed as $\frac{\|R_{cuBLAS} - R_{MKL}\|_F}{\sqrt{k+2}|\alpha|\|A\|_F\|B\|_F + 2|\beta|\|C\|_F}$, where k is equal to the matrix size.

Leveraging Half Precision in HPC on V100

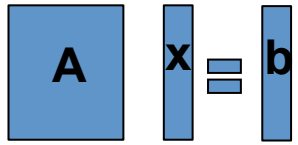
Motivation

Study of the LU factorization algorithm on Nvidia V100

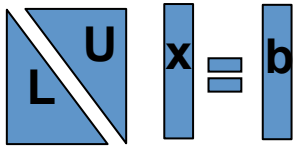


- LU factorization is used to solve a linear system Ax=b

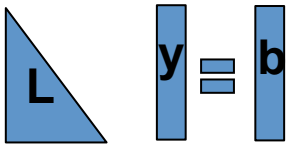
A x = b



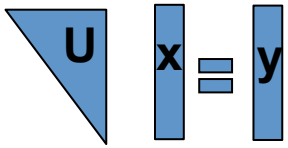
LUx = b



Ly = b

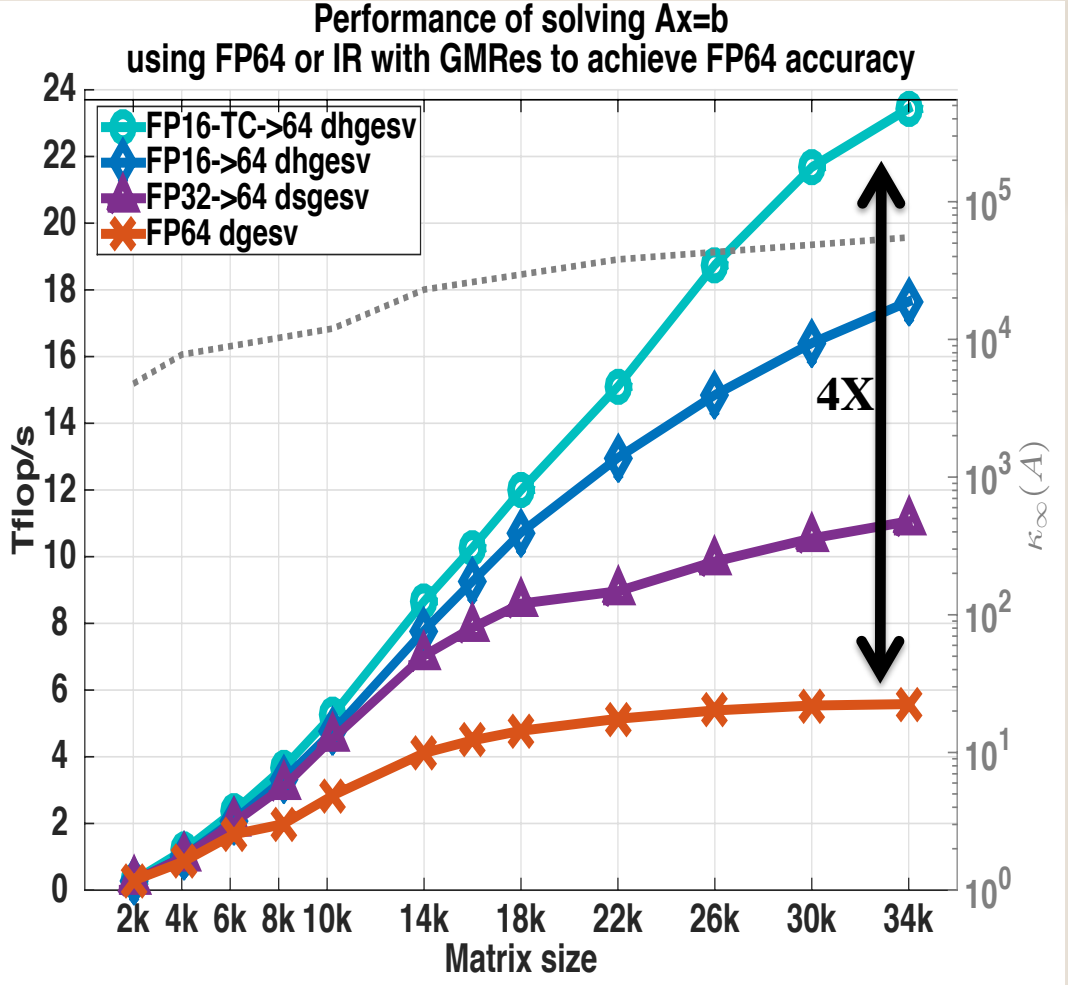


then
Ux = y



Tensor Core Accelerated IRS solving linear system Ax = b

Performance Behavior



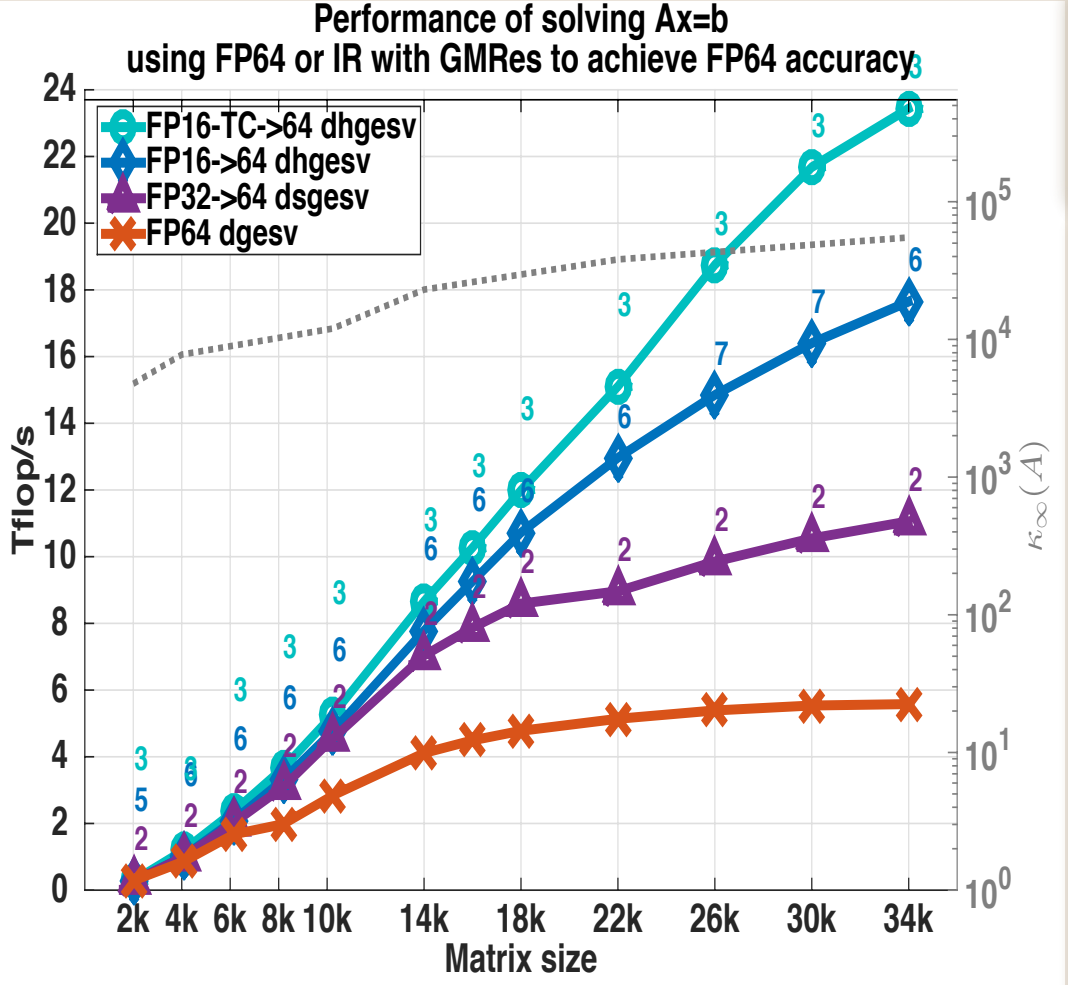
Flops = $2n^3/(3$ time)
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy

Problem generated with an arithmetic distribution of the singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and positive eigenvalues.

Tensor Core Accelerated IRS solving linear system Ax = b

Performance Behavior



Flops = $2n^3/(3 \text{ time})$
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy

Problem generated with an arithmetic distribution of the singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and positive eigenvalues.

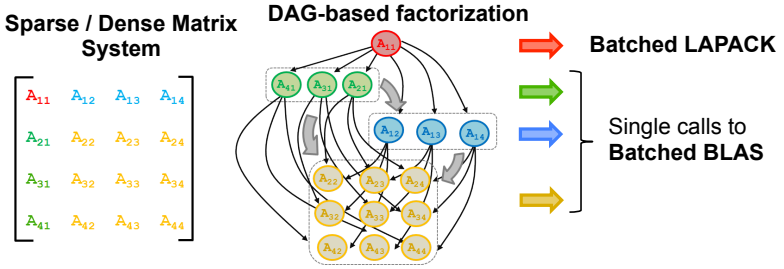
Batched Linear Algebra

Many applications need LA on many small matrices

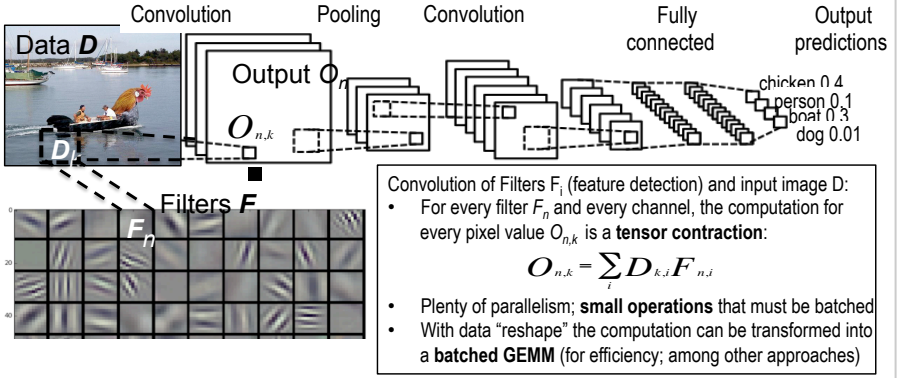
Data Analytics and associated with it Linear Algebra on small LA problems are needed in many applications:

- Machine learning,
- Data mining,
- High-order FEM,
- Numerical LA,
- Graph analysis,
- Neuroscience,
- Astrophysics,
- Quantum chemistry,
- Multi-physics problems,
- Signal processing, etc.

Sparse/Dense solvers & preconditioners



Machine learning



Convolution of Filters F_i (feature detection) and input image D:

- For every filter F_n and every channel, the computation for every pixel value $O_{n,k}$ is a **tensor contraction**:

$$O_{n,k} = \sum_i D_{k,i} F_{n,i}$$

- Plenty of parallelism; **small operations** that must be batched
- With data "reshape" the computation can be transformed into a **batched GEMM** (for efficiency; among other approaches)

Applications using high-order FEM

- Matrix-free basis evaluation needs efficient tensor contractions,

$$C_{i1,i2,i3} = \sum_k A_{k,i1} B_{k,i2,i3}$$

- **Within ECP CEED Project,** designed MAGMA batched methods to split the computation in many small high-intensity GEMMs, grouped together (batched) for efficient execution:

Batch_{ $C_{i3} = A^T B_{i3}$, for range of i3 }

Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel

```
for (int i=0; i<batchcount; i++)
  la_computation(A_i, …)
```

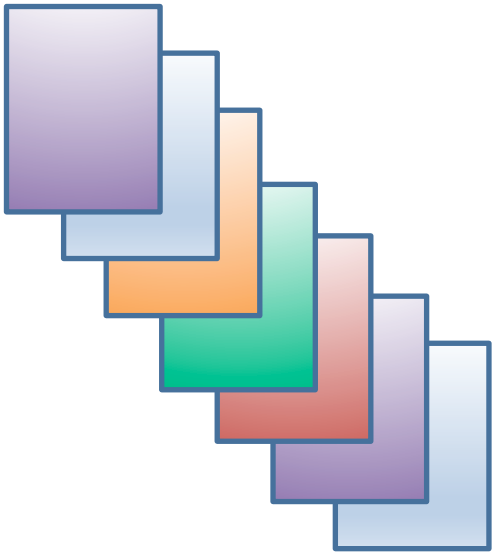
Call a numerical library (e.g., BLAS, LAPACK, etc.) to compute A_i in parallel

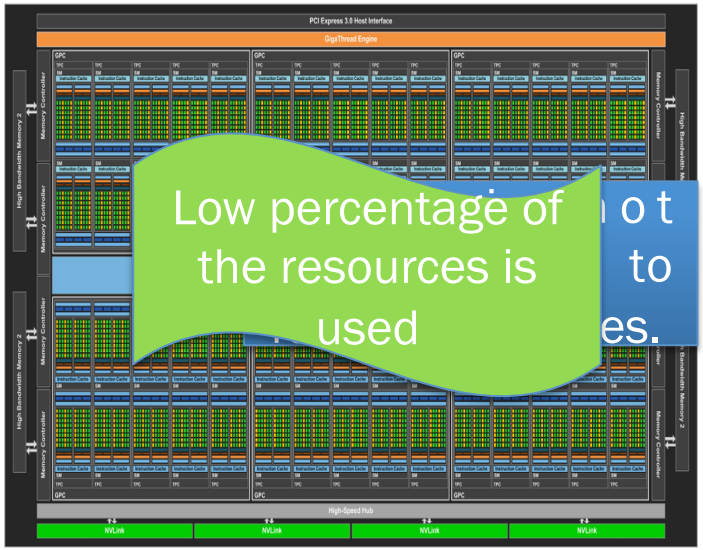
Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

```
for (int i=0; i<batchcount; i++)
  dgemm(…)
```





Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resources** by assigning a matrix to each group of core/TB to operate on it independently

```
for (int i=0; i<batchcount; i++)
  la_computation(A_i, …)
```

```
Batched_la_computation(A,batchcount,…)
```

Batched Linear Algebra Computations

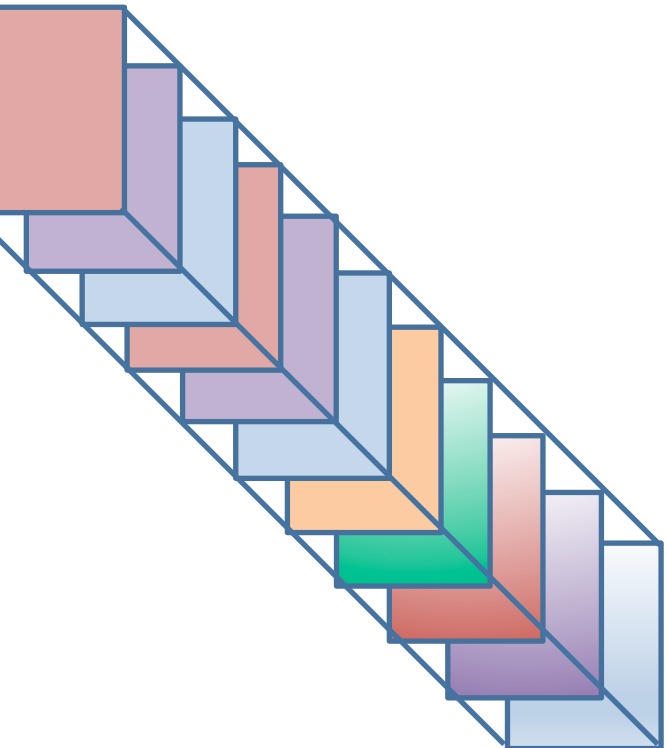
Non-batched computation

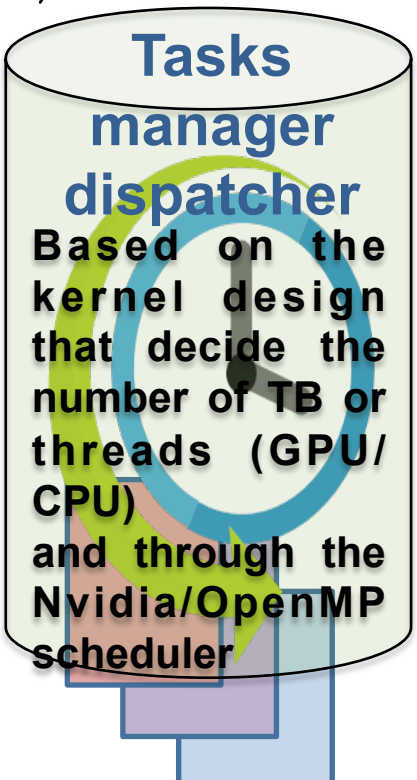
- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

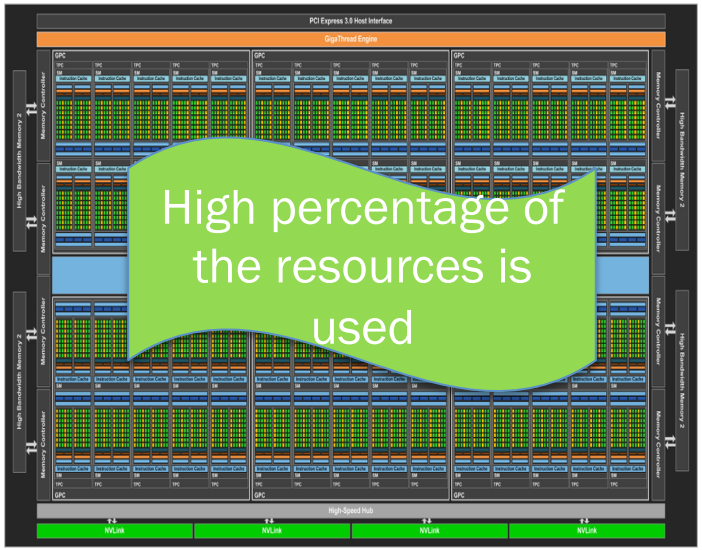
Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resources** by assigning a matrix to each group of core/TB to operate on it independently

`Batched_dgemm(...)`







Batched Linear Algebra Computations

Non-batched computation

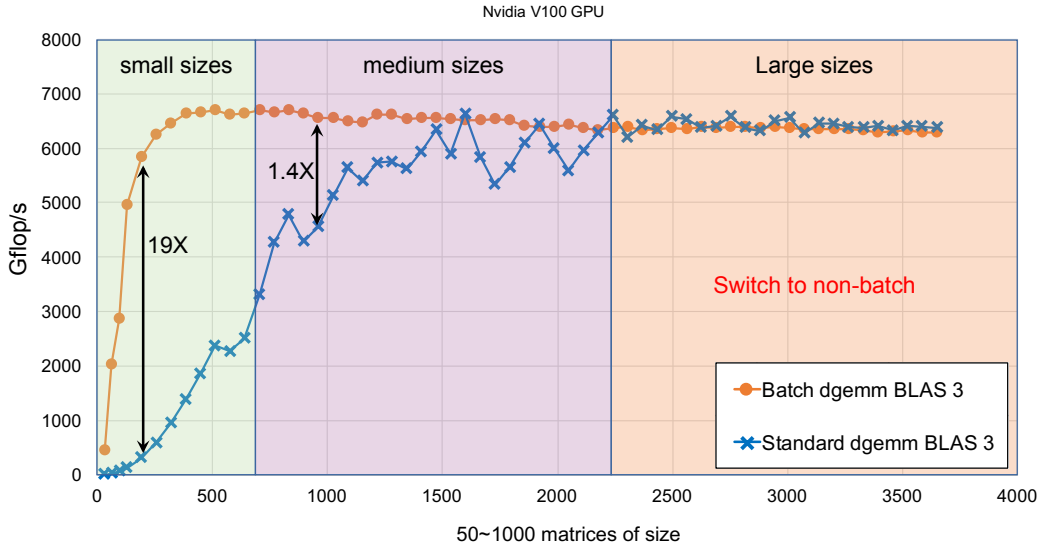
- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resources** by assigning a matrix to each group of core/TB to operate on it independently

```
for (int i=0; i<batchcount; i++)
  la_computation(A_i, …)
```

```
Batched_la_computation(A,batchcount,…)
```



Matrix sizes (fixed) in the batch

Batch size 1,000 Batch size 300 Batch size 50

Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resources** by assigning a matrix to each group of core/TB to operate on it independently

Some simple examples from BLAS

```
// Batch of adding vector entries
// dy(:) += da * dx(:)
for (int i=0; i<n; i++)
  dy[i] = dy[i] + da * dx[i];
```

```
// Level 1 BLAS daxpy routine
daxpy( n, da, dx, 1, dy, 1);
```

Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resources** by assigning a matrix to each group of core/TB to operate on it independently

Some simple examples from BLAS

```
// Batch of Level 1 BLAS dot-prod.
// y_i = alpha * A_i • x + beta y_i
for (int i=0; i<m; i++)
  double res = 0.;
  for (int j=0; j<n; j++)
     res += A(i,j) * x[j];
  y[i] = alpha * res + beta * y[i];
```

```
// Level 2 BLAS dgemv routine
dgemv('N', m, n,
      alpha, A, lda, x, 1,
      beta, y, 1);
```

Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resources** by assigning a matrix to each group of core/TB to operate on it independently

Some simple examples from BLAS

```
// Batch of Level 2 BLAS dgemv.
// C(:,i) = alpha* A B_i+ beta C(:,i)
for (int i=0; i<n; i++)
  dgemv('N', m, k,
        alpha, A, lda, B+i*ldb, 1,
        beta, C+i*ldc, 1);
```

```
// Level 3 BLAS dgemm routine
dgemm('N', 'N', m, n, k,
      alpha, A, lda, B, ldb,
      beta, C, ldc);
```

Batched Linear Algebra Computations

Non-batched computation

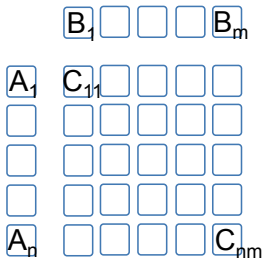
- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

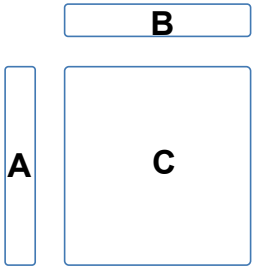
- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resources** by assigning a matrix to each group of core/TB to operate on it independently

Some simple examples from BLAS

```
// Batch of rank-k Level3 BLAS
// Cij = alpha* Ai Bj+ beta Cij
for (int i=0; i<n; i++)
   for (int j=0; j<m; j++)
      Cij = Ai Bj + beta Cij
```




```
// Level 3 BLAS dgemm routine
Batched_dgemm(...);

or, in this case, one big dgemm
dgemm( ... );
```



Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resources** by assigning a matrix to each group of core/TB to operate on it independently

Example from tensor computations

```
// Tensor computations as batches
// of LA, e.g., batched gemms
//
//
```

$$C_{i1,i2,i3} = \sum_k A_{k,i1} B_{k,i2,i3}$$

```
for (int i3=0; i<batch; i++)
   C_i3 = A^T B_i3
```

```
// Batched dgemms
Batched_dgemm('T', 'N', m, n, k,
                    1, A, lda, B, ldb, 0, C, ldc, batch);
```

Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resources** by assigning a matrix to each group of core/TB to operate on it independently

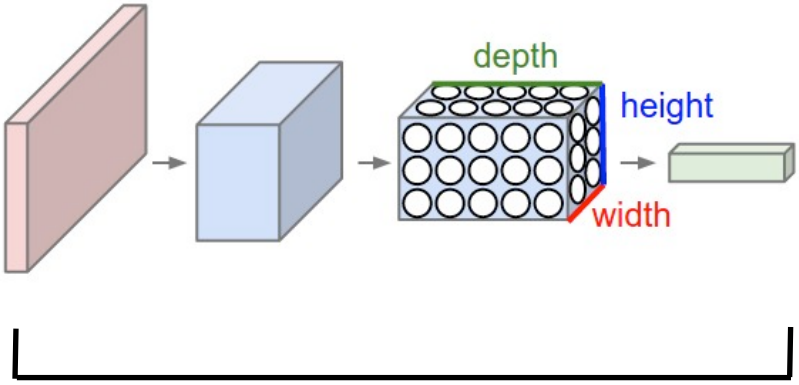
Example from CNNs

```
// Apply convolutions with filers F
// on batch of images I
for (int i=0; i<batch; i++)
  for (int x=0; x<xdim-1; x++)
     for (int y=0; y<ydim-1; y++)
       R[i][x][y] = 0;
       for (int l=0; l<m-1; l++)
          for (int k=0; k<n-1; k++)
             R[i][x][y]+=F[l][k]*I[i][x][y];
```

```
// Batched dgemms
Unfold_data(...);
Batched_dgemm(..., batch);
Put_results_back( ... );
```

Example from DNN Accelerate DNNs through GEMMs of various sizes

Convolution Network (ConvNet) example



Require matrix-matrix products of various sizes, including batched GEMMs

- **Convolutions can be accelerated in various ways:**
  - **Unfold and GEMM**
  - **FFT**
  - **Winograd minimal filtering – reduction to batched GEMMs**

Fast Convolution				
Layer	m	n	k	M
1	12544	64	3	1
2	12544	64	64	1
3	12544	128	64	4
4	12544	128	128	4
5	6272	256	128	8
6	6272	256	256	8
7	6272	256	256	8
8	3136	512	256	16
9	3136	512	512	16
10	3136	512	512	16
11	784	512	512	16
12	784	512	512	16
13	784	512	512	16

- **Use autotuning to handle complexity of tuning**

Batched LA Interface Design

Large scientific community effort (industry, national labs, academia) to define Batched BLAS and LAPACK standard

Workshop on Batched, Reproducible, and Reduced Precision BLAS 2016

Workshop on Batched, Reproducible, and Reduced Precision BLAS 2017

SC17 BOF SESSION

SC18 BOF SESSION

SIAM-CSE19

A. Abdelfattah, T. Costa, J. Dongarra, M. Gates, A. Haidar, S. Hammarling, N. Higham, J. Kurzak, P. Luszczek, S. Tomov, M. Zounon, **A Set of Batched Basic Linear Algebra Subprograms and LAPACK Routines,** TOMS, June 2021.

Ack.: James Demmel, Iain Duff, Murat Guney, Greg Henry, Jonathan Hogg, Hatem Ltaief, Sarah Knepper, Pedro Valero Lara, Srikara Pranesh, Sivasankaran Rajamanickam, Samuel Relton, Jason Riedy, and Shane Story.

Batched LA Interface Design

- Single interface and proposed API standard for **Batched BLAS and LAPACK** for **functional and performance portability** across architectures
- Main design issues include
  - **Data layout for single problem**
    (row/column major, symmetric, band, structure, sparse formats ...)
  - **Data layout interface for the batched problems**
    (groups, variable/constant size, one matrix after another or strided, etc.)
  - **Supported precisions**

Batched GEMM (general matrix-matrix multiply) C language declarations in multiple precisions

```
int BLAS_gemm_batched_r16(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Float16 * alpha, _Float16 ** A, int * A_ld, _Float16 ** B,
        int * B_ld, _Float16 * beta, _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c16(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Complex _Float16 * alpha, _Complex _Float16 ** A, int *
        A_ld, _Complex _Float16 ** B, int * B_ld, _Complex _Float16 * beta, _Complex _Float16 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_r32(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, float * alpha, float ** A, int * A_ld, float ** B, int * B_ld,
        float * beta, float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c32(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Complex float * alpha, _Complex float ** A, int * A_ld,
        _Complex float ** B, int * B_ld, _Complex float * beta, _Complex float ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_r64(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, double * alpha, double ** A, int * A_ld, double ** B, int *
        B_ld, double * beta, double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c64(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Complex double * alpha, _Complex double ** A, int *
        A_ld, _Complex double ** B, int * B_ld, _Complex double * beta, _Complex double ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_r128(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Float128 * alpha, _Float128 ** A, int * A_ld, _Float128 **
        B, int * B_ld, _Float128 * beta, _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c128(BLAS_Layout layout, BLAS_Op * A_trans, BLAS_Op * B_trans, int * m, int * n, int * k, _Complex _Float128 * alpha, _Complex _Float128 ** A,
        int * A_ld, _Complex _Float128 ** B, int * B_ld, _Complex _Float128 * beta, _Complex _Float128 ** C, int * C_ld, int group_count, int * group_sizes, int * info);
```

MAGMA Batched Computations

MAGMA BATCHED

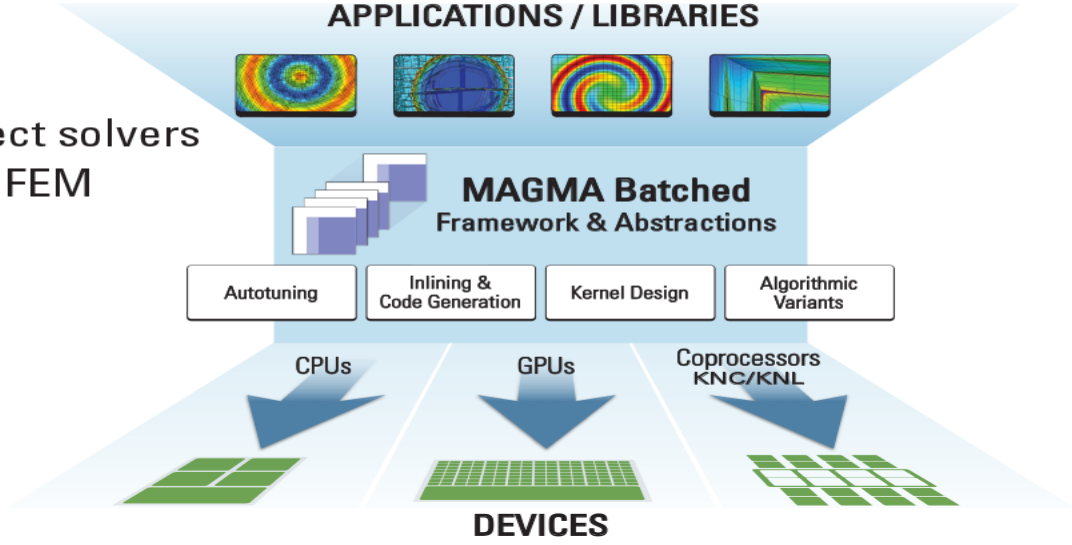
BATCHED FACTORIZATION OF A SET OF SMALL MATRICES IN PARALLEL

Numerous applications require factorization of many small matrices

- Deep learning
- Structural mechanics
- Astrophysics
- Sparse direct solvers
- High-order FEM simulations

ROUTINES

LU, QR, and Cholesky	✓
Solvers and matrix inversion	✓
All BLAS 3 (fixed + variable)	✓
SYMV, GEMV (fixed + variable)	✓



API for Batched BLAS in MAGMA

Batch of fixed-size problems:

```
extern "C" void
magmablas_dgemm_batched( magma_trans_t transA, magma_trans_t transB,
                         magma_int_t m, magma_int_t n, magma_int_t k,
                         double alpha,
                         double const * const * dA_array, magma_int_t ldda,
                         double const * const * dB_array, magma_int_t lddb,
                         double beta,
                         double **dC_array, magma_int_t lddc,
                         magma_int_t batchCount, magma_queue_t queue )
```

Batch of variable-size problems:

```
extern "C" void
magmablas_dgemm_vbatched(
    magma_trans_t transA, magma_trans_t transB,
    magma_int_t* m, magma_int_t* n, magma_int_t* k,
    double alpha,
    double const * const * dA_array, magma_int_t* ldda,
    double const * const * dB_array, magma_int_t* lddb,
    double beta,
    double **dC_array, magma_int_t* lddc,
    magma_int_t batchCount, magma_queue_t queue )
```

API for Batched LAPACK in MAGMA

Batch of fixed-size problems:

```
extern "C" magma_int_t
magma_zpotrf_batched(
    magma_uplo_t uplo, magma_int_t n,
    magmaDoubleComplex **dA_array, magma_int_t ldda,
    magma_int_t *info_array,  magma_int_t batchCount,
    magma_queue_t queue)
```

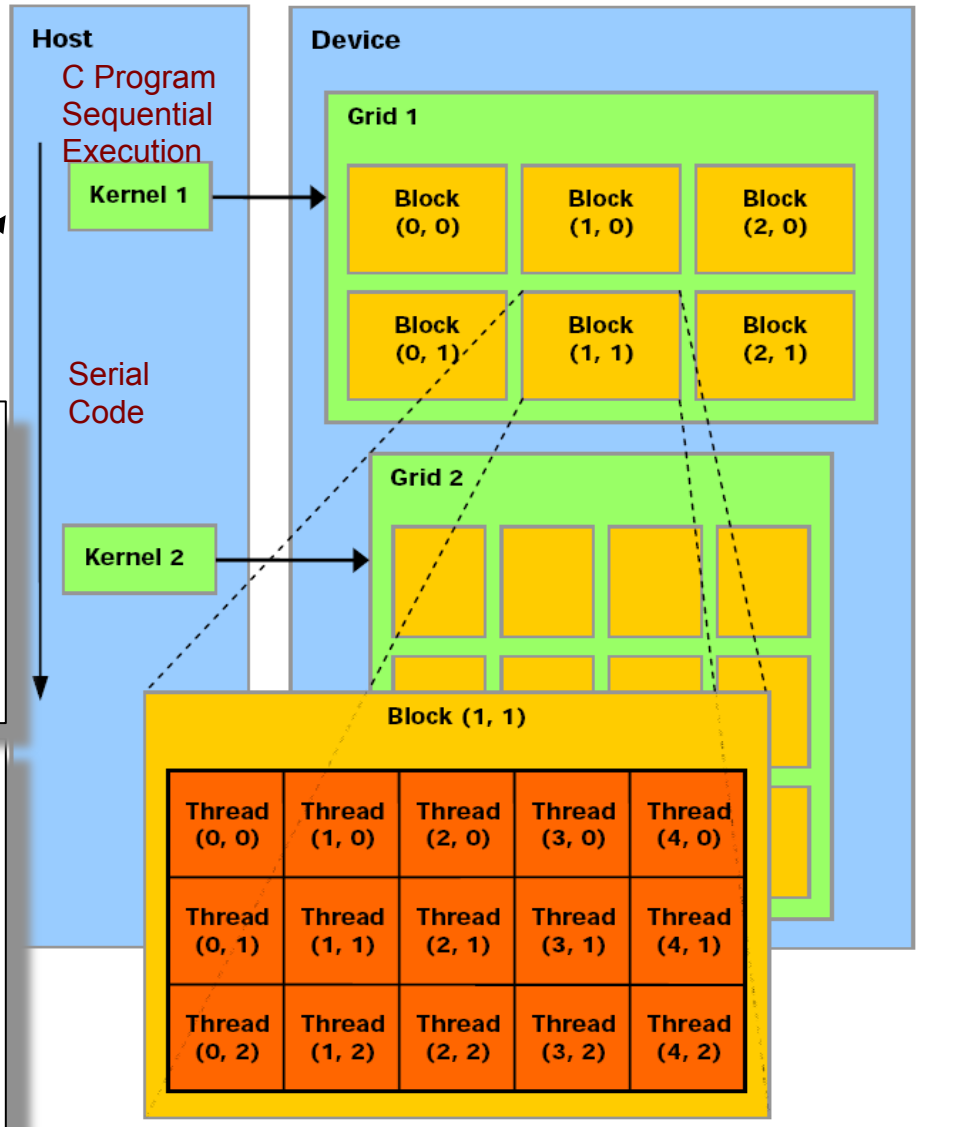
Batch of variable-size problems:

```
extern "C" magma_int_t
magma_zpotrf_vbatched(
    magma_uplo_t uplo, magma_int_t *n,
    magmaDoubleComplex **dA_array, magma_int_t *ldda,
    magma_int_t *info_array,  magma_int_t batchCount,
    magma_queue_t queue)
```

Implementation of Batched LA in CUDA

- Grid of thread blocks
  (blocks of the same dimension, grouped together to execute the same kernel)
- Thread block
  (a batch of threads with fast shared memory executes a kernel)
- Sequential code launches asynchronously GPU kernels

(Source: NVIDIA CUDA Programming Guide)




```
void LA_computation(double *A, ... ) {
    // set the grid and thread configuration
    Dim3 dimGrid(2,3);
    Dim3 dimTBlock(3,5);

    // Launch the device computation
    LA_computation_kernel<<<dimGrid, dimTBlock>>>(A, .. );
```

```
__global__ void LA_computation_kernel(double *A,  . . . ) {
  // Get Block index that this TB is computing
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Get the Thread index that this thread is computing
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  LA_device(A, bx, by, tx, ty, ...);
```

Implementation of Batched LA in CUDA

- **Grid of thread blocks**
  (blocks of the same dimension, grouped together to execute the same kernel)
- **Thread block**
  (a batch of threads with fast shared memory executes a kernel)
- **Sequential code launches asynchronously GPU kernels**

- **Assume we want to run LA_computation on a batch of independent problems**
- **One way is to redesign LA_computation by running not just one 2 x 3 Grid but on a 2 x 3 x batch Grid where each sub-problem is done on a 2 x 3 Grid**

```
void LA_computation(double *A, ... ) {
   // set the grid and thread configuration
   Dim3 dimGrid(2,3);
   Dim3 dimTBlock(3,5);

   // Launch the device computation
   LA_computation_kernel<<<dimGrid, dimTBlock>>>(A, ... );
```

```
__global__ void LA_computation_kernel( . . . ) {
  // Get Block index that this TB is computing
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Get the Thread index that this thread is computing
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  LA_device(A, bx, by, tx, ty, ...);
```

Batched

```
void Batched_LA_computation(double **A, ..., int batchCount ) {
   // set the grid and thread configuration
   Dim3 dimGrid(2,3, batchCount);
   Dim3 dimTBlock(3,5);

   // Launch the device computation
   Batched_LA_computation_kernel<<<dimGrid,
dimTBlock>>>(A,  . . . );
```

```
__global__ void Batched_LA_computation_kernel( . . . ) {
  // Get Block index that this TB is computing
  int bx = blockIdx.x, by = blockIdx.y;
  int bz = blockIdx.z;

  // Get the Thread index that this thread is computing
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  LA_device( A[bz], bx, by, tx, ty, ...);
```

Other considerations in implementing fast batched LA

- Can not rely on single implementation to get portable high-performance
- Multiple kernels are designed for various scenarios and parameterized for subsequent autotuning
- **Fused kernels**

Optimize communications:

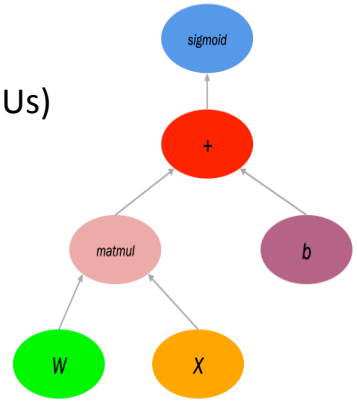
batch<e=0..nelems>{ $B_e^T D_e .* (B_e A_e B_e{}^T) B_e$ }

vs.

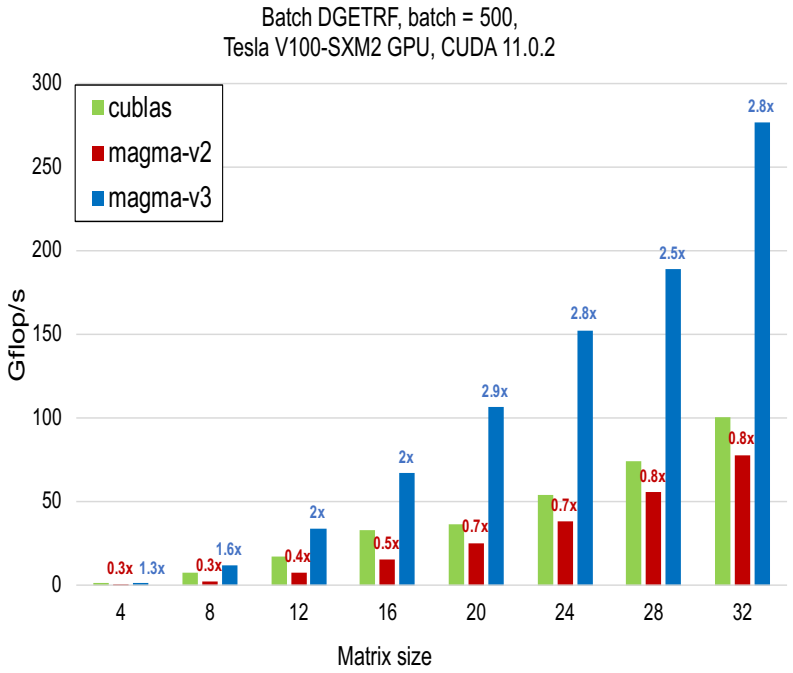
batch<e=0..nelems> { $C_e = A_e B_e{}^T$ };
batch<e=0..nelems> { $C_e = B_e C_e$ };
batch<e=0..nelems> { $C_e = D_e .* C_e$ };
batch<e=0..nelems> { $C_e = C_e B_e$ };
batch<e=0..nelems> { $C_e = B_e^T C_e$ };

How to code fused kernels:

- JIT (nvtrc for NVIDIA GPUs)
- Constructed as DAGs
- Sequence of device BLAS routines



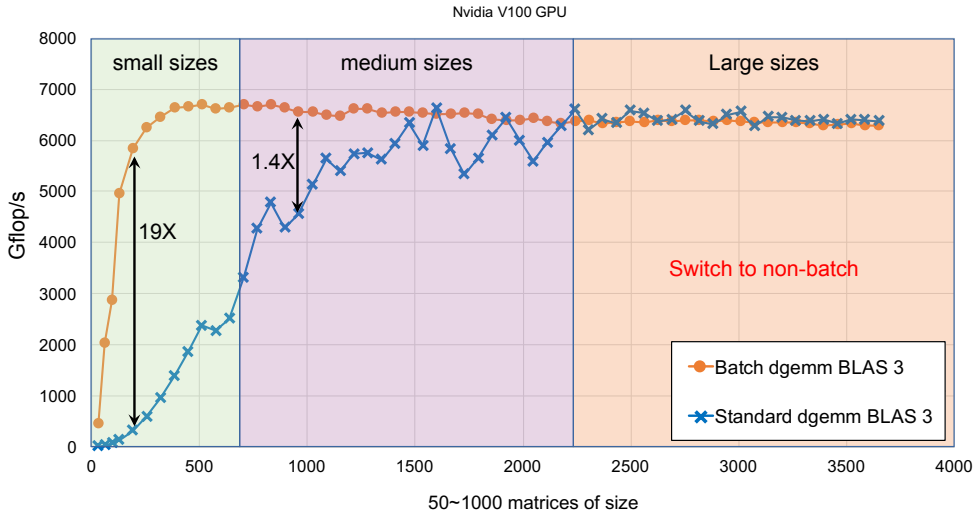
Performance improvements in Batched LU through various levels of kernel fusion
(most recent developments in MAGMA by A. Abdelfattah)



How to implement fast batched DLA?

Problem sizes influence algorithms & optimization techniques

- Can not rely on single implementation to get portable high-performance

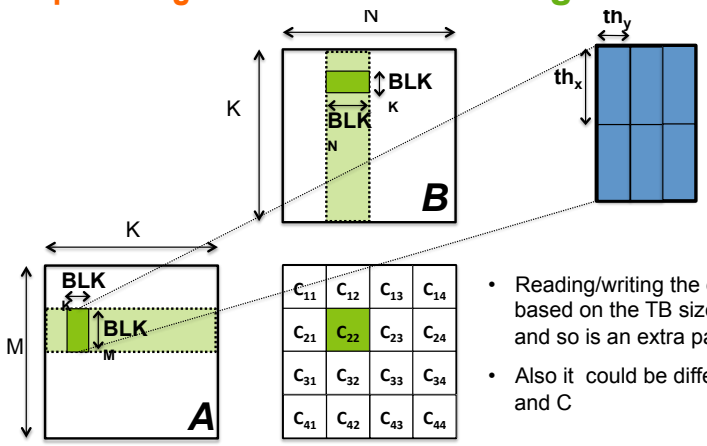


Matrix sizes (fixed) in the batch

Batch size 1,000 Batch size 300 Batch size 50

Kernels are designed various scenarios and parameterized for autotuning framework to find "best" performing kernels

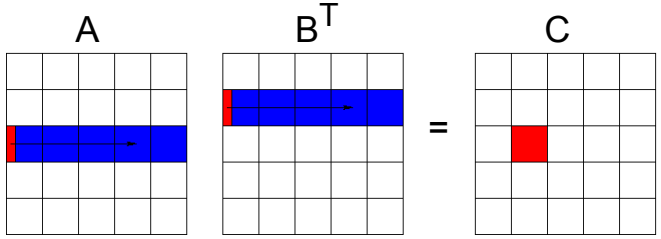
Optimizing GEMM's: Kernel design



- Reading/writing the elements is based on the TB size (# threads) and so is an extra parameter.
- Also it could be different for A, B and C

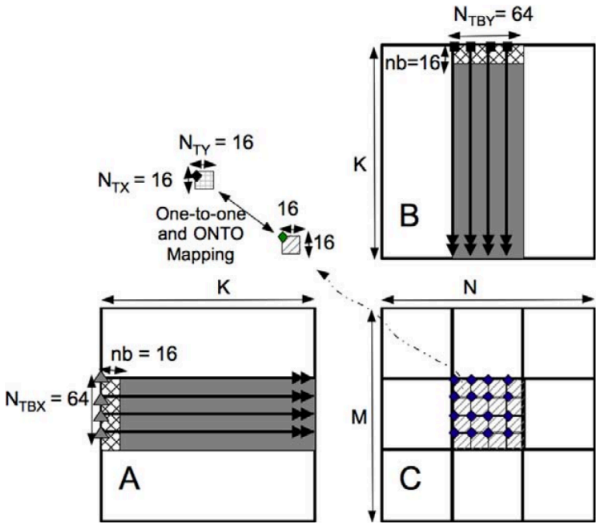
GEMM in MAGMA

2010. R. Nath, S. Tomov and J. Dongarra. ***An improved MAGMA GEMM for Fermi Graphics Processing Units***, The International Journal of High Performance Computing Applications.
http://www.netlib.org/utk/people/JackDongarra/journals/208_2010_an-improved-magma-gemm-for-fermi-gpus.pdf



* **Small** red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads

- **Add register blocking**
- **Parameterized for autotuning for particular size GEMMs and portability across GPUs**



A thread computes part of a row (16 values) of the C block

A thread computes a block of C (4 x 4 values in this case)

Autotuning

1) Kernel variants: performance parameters are exposed through a templated kernel interface

```
template< typename T,  int    DIM_X,  int   DIM_Y,
                       int    BLK_M,  int   BLK_N,  int   BLK_K,
                       int   DIM_XA,  int  DIM_YA,  int  DIM_XB,  int  DIM_YB,
                       int    THR_M,  int   THR_N,  int   CONJA,  int   CONJB >
static __device__ void  tensor_template_device_gemm_nn( int  M,  int  N,  int  K, …
```

2) CPU interfaces that call the GPU kernels as a Batched computation

```
template<typename T,  int   DIM_X,  int   DIM_Y, … >
void tensor_template_batched_gemm_nn( int m, int n, int k, … ) {
    …
  tensor_template_device_gemm _nn<T, DIM_X, DIM_Y, … ><<<dimGrid, dimBlock, 0, queue>>>(m, n, k,…);
}
```

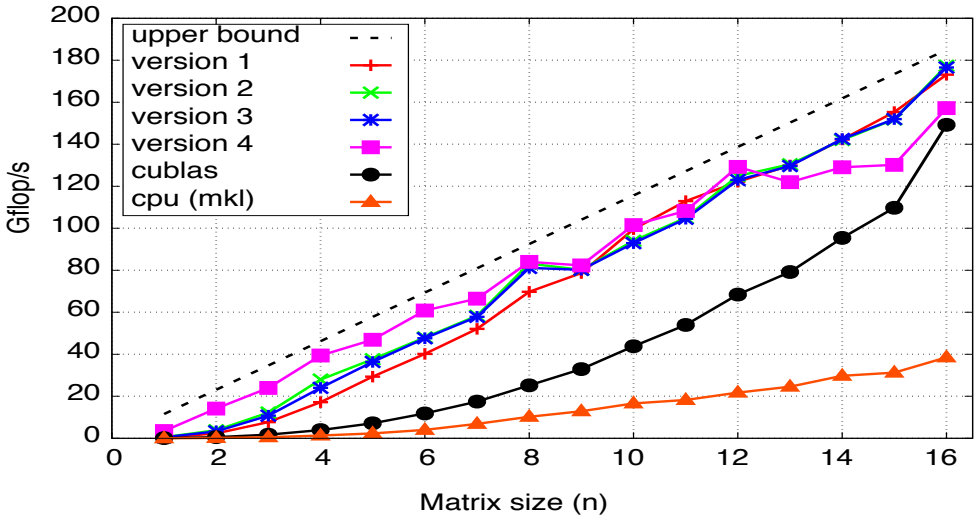
3) Python scripts that generate the search space for the parameters DIM_X, DIM_Y …

index,	DIM_X,	DIM_Y,	…						
#define NN_V_0	4,	8,	8,	24,	8,	1,	4,	8,	4,	8
#define NN_V_1	4,	8,	8,	32,	8,	1,	4,	8,	4,	8
#define NN_V_2	4,	8,	8,	40,	8,	1,	4,	8,	4,	8
…										

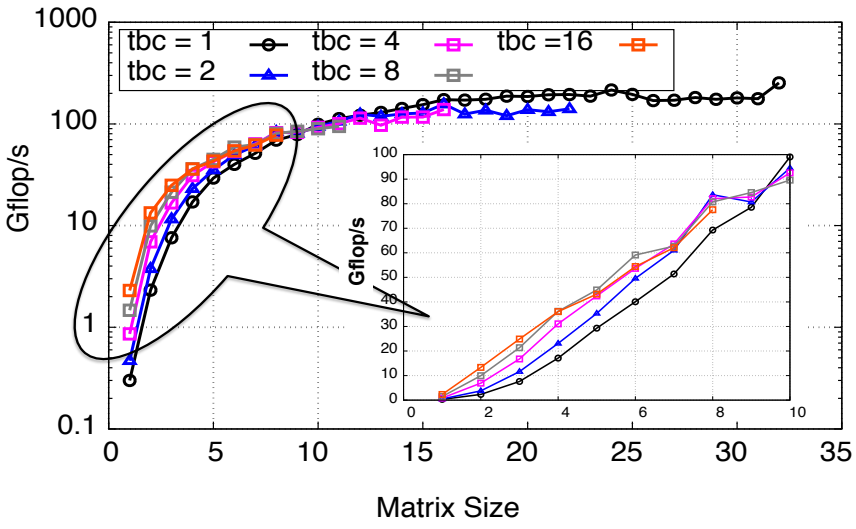
4) Scripts that run all versions in the search space, analyze the results, and return the best combination of parameters, which is stored in the library for subsequent use.

Performance results

Performance comparison of tensor contraction versions using batched C = αAB + βC on 100,000 square matrices of size n on a K40c GPU and 16 cores of Intel Xeon E5-2670, 2.60 GHz CPUs.



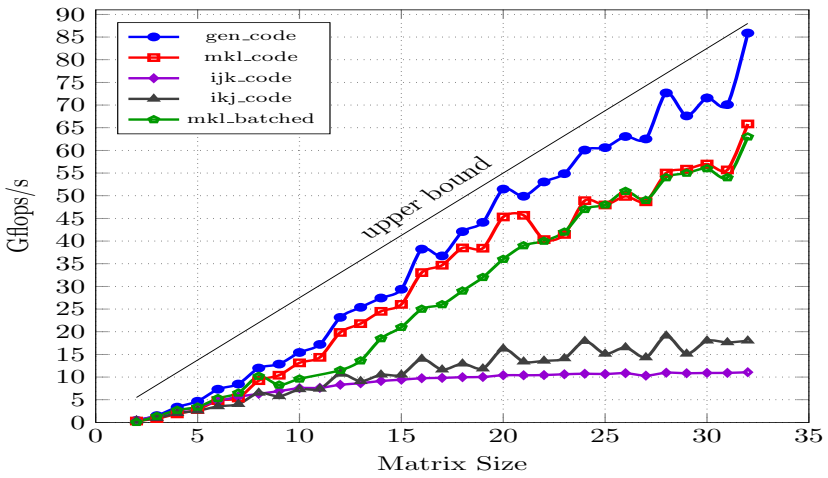
Effect of a Thread Block Concurrency (tbc) techniques where several DGEMMs are performed on one TB simultaneously



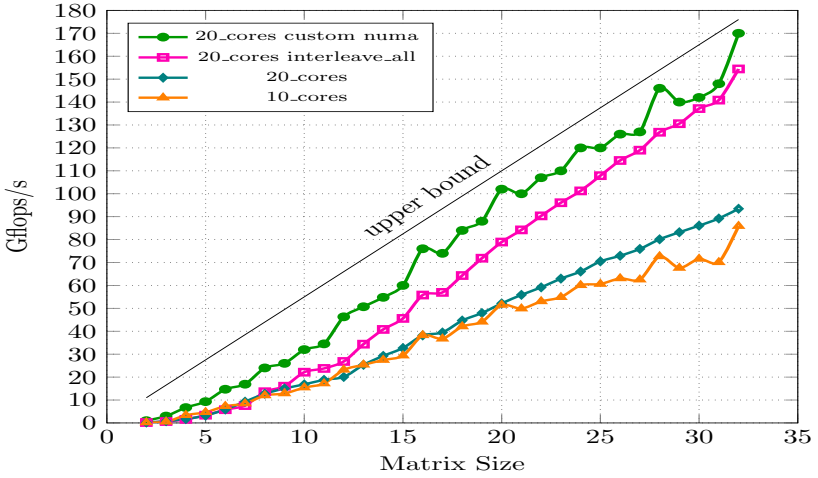
Performance …

Batched DGEMM on CPUs

Intel Xeon E5-2650 v3 (Haswell), 10 cores

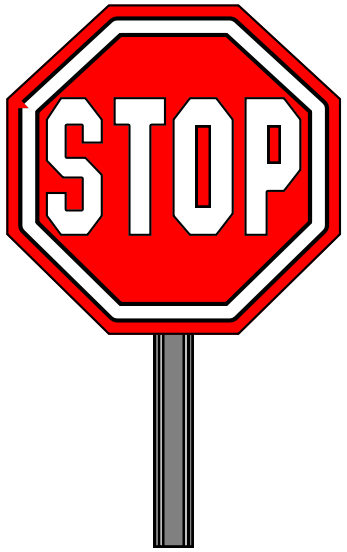


2 x Intel Xeon E5-2650 v3 (Haswell), 20 cores



I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra,
High-performance matrix-matrix multiplications of very small matrices,
Euro-Par'16, Grenoble, France, August 22-26, 2016.

The End



- The End!