MAGMA
LAPACK for GPUs

Stan Tomov
Research Director
Innovative Computing Laboratory
Department of Computer Science
University of Tennessee, Knoxville

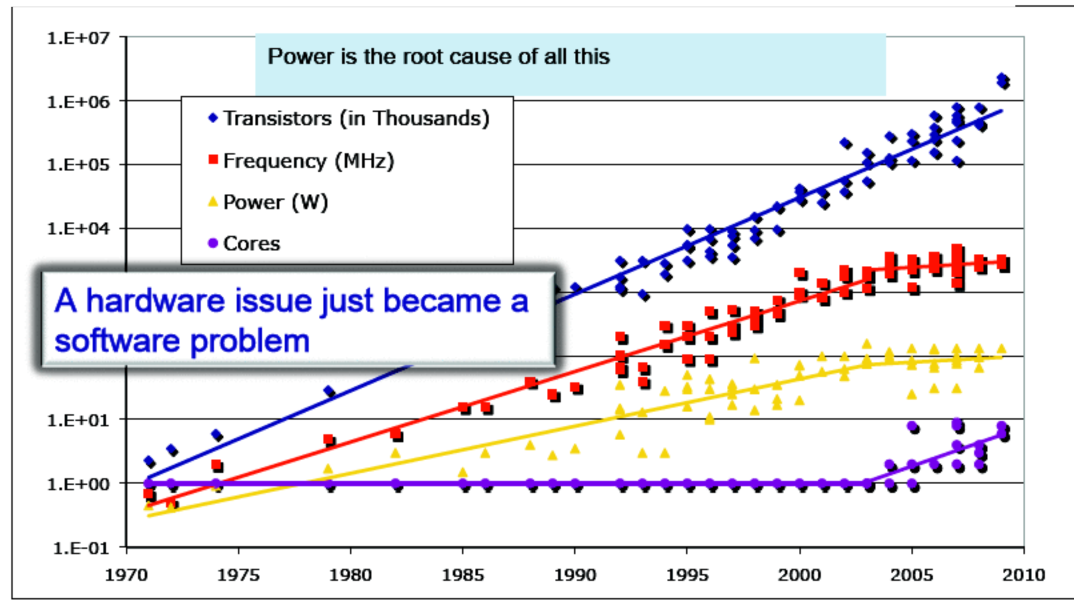
Keeneland GPU Tutorial 2011, Atlanta, GA
April 14-15, 2011

Outline

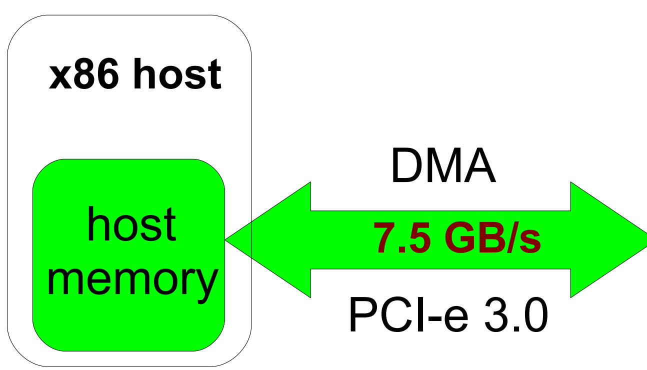
- **Motivation**
- **MAGMA 1.0 – LAPACK for GPUs**
    - **Overview**
    - **Using MAGMA**
    - **Methodology**
    - **Performance**
- **Current & future work directions**
- **Conclusions**

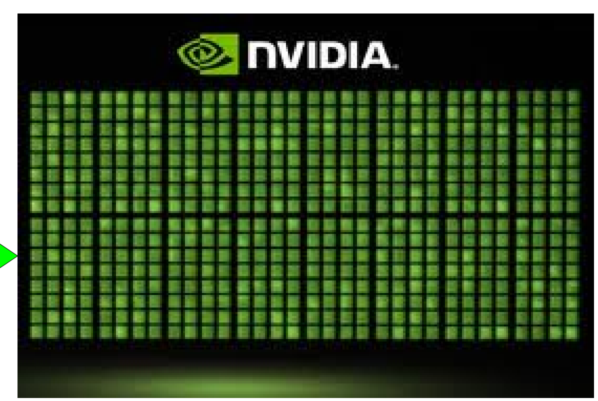
Hardware Trends

- Power consumption and the move towards multicore
- Hybrid architectures
- GPU
- Hybrid GPU-based systems
    - CPU and GPU to get integrated (NVIDIA to make ARM CPU cores alongside GPUs)

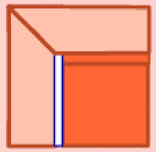
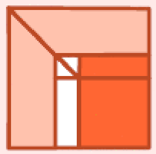
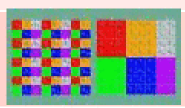
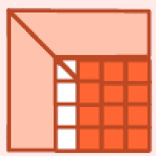
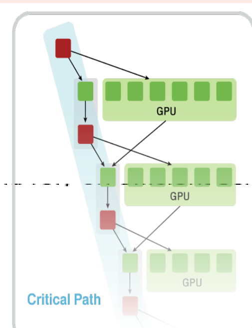


Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç
Slide from Kathy Yelick





A New Generation of Algorithms

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing
PLASMA (00's) New Algorithms (many-core friendly)		Rely on - a DAG/scheduler - block data layout - some extra kernels
MAGMA Hybrid Algorithms (heterogeneity friendly)		Rely on - hybrid scheduler (of DAGs) - hybrid kernels (for nested parallelism) - existing software infrastructure

Matrix **A**lgebra on **G**PU and **M**ulticore **A**rchitectures (**MAGMA**)

- **MAGMA**: **a new generation linear algebra (LA) libraries** to achieve the fastest possible time to an accurate solution **on hybrid/heterogeneous architectures**
  Homepage: **http://icl.cs.utk.edu/magma/**

- **MAGMA** & **LAPACK**
  - **MAGMA** uses LAPACK (on the CPUs) and extends its functionality to hybrid systems (GPU support);
  - **MAGMA** is designed to be similar to LAPACK in functionality, data storage and interface
  - **MAGMA** leverages years of experience in developing open source LA software packages like LAPACK, ScaLAPACK, BLAS, ATLAS, and PLASMA

- **MAGMA developers/collaborators**
  - University of Tennessee, **Knoxville**; University of California, **Berkeley**; University of Colorado, **Denver**
  - INRIA Bordeaux - Sud Ouest, France; INRIA Paris – Saclay, France; KAUST, Saudi Arabia

MAGMA 1.0 RC5

- 32 algorithms are developed (total – 122 routines)
  - Every algorithm is in 4 precisions (s/c/d/z, denoted by X)
  - There are 3 mixed precision algorithms (zc & ds, denoted by XX)
  - These are **hybrid algorithms**
  - Expressed in terms of BLAS
- Support is for single CUDA-enabled NVIDIA GPU, either Tesla or Fermi
- MAGMA BLAS
  - A subset of GPU BLAS, optimized for Tesla and Fermi GPUs

MAGMA 1.0

One-sided factorizations

1. Xgetrf	LU factorization; CPU interface
2. Xgetrf_gpu	LU factorization; GPU interface
3. Xgetrf_mc	LU factorization on multicore (no GPUs)
4. Xpotrf	Cholesky factorization; CPU interface
5. Xpotrf_gpu	Cholesky factorization; GPU interface
6. Xpotrf_mc	Cholesky factorization on multicore (no GPUs)
7. Xgeqrf	QR factorization; CPU interface
8. Xgeqrf_gpu	QR factorization; GPU interface; with T matrices stored
9. Xgeqrf2_gpu	QR factorization; GPU interface; without T matrices
10. Xgeqrf_mc	QR factorization on multicore (no GPUs)
11. Xgeqrf2	QR factorization; CPU interface
12. Xgeqlf	QL factorization; CPU interface
13. Xgelqf	LQ factorization; CPU interface

MAGMA 1.0

Linear solvers

14. Xgetrs_gpu	Work precision; using LU factorization; GPU interface
15. Xpotrs_gpu	Work precision; using Cholesky factorization; GPU interface
16. Xgels_gpu	Work precision LS; GPU interface
17. XXgetrs_gpu	Mixed precision iterative refinement solver; Using LU factorization; GPU interface
18. XXpotrs_gpu	Mixed precision iterative refinement solver; Using Cholesky factorization; GPU interface
19. XXgeqrsv_gpu	Mixed precision iterative refinement solver; Using QR on square matrix; GPU interface

MAGMA 1.0

Two-sided factorizations

20. Xgehrd	Reduction to upper Hessenberg form; with T matrices stored; CPU interface
21. Xgehrd2	Reduction to upper Hessenberg form; Without the T matrices stored; CPU interface
22. Xhetrd	Reduction to tridiagonal form; CPU interface
23. Xgebrd	Reduction to bidiagonal form; CPU interface

MAGMA 1.0

Generating/applying orthogonal matrices

24. Xungqr	Generates Q with orthogonal columns as the product of elementary reflectors (from Xgeqrf); CPU interface
25. Xungqr_gpu	Generates Q with orthogonal columns as the product of elementary reflectors (from Xgeqrf_gpu); GPU interface
26. Xunmtr	Multiplication with the orthogonal matrix, product of elementary reflectors from Xhetrd; CPU interface
27. Xunmqr	Multiplication with orthogonal matrix, product of elementary reflectors from Xgeqrf; CPU interface
28. Xunmqr_gpu	Multiplication with orthogonal matrix, product of elementary reflectors from Xgeqrf_gpu; GPU interface
29. Xunghr	Generates Q with orthogonal columns as the product of elementary reflectors (from Xgehrd); CPU interface

MAGMA 1.0

Eigen/singular-value solvers

30. Xgeev	Solves the non-symmetric eigenvalue problem; CPU interface
31. Xheevd	Solves the Hermitian eigenvalue problem; Uses devide and conquer; CPU interface
32. Xgesvd	SVD; CPU interface

- Currently, these routines have GPU-acceleration for the
  - two-sided factorizations used and the
  - Orthogonal transformation related to them (matrix generation/application from slide 9)

MAGMA BLAS

Level 2 BLAS

1. Xgemv_tesla	General matrix-vector product for Tesla
2. Xgemv_fermi	General matrix-vector product for Fermi
3. Xsymv_ tesla	Symmetric matrix-vector product for Tesla
4. Xsymv_fermi	Symmetric matrix-vector product for Fermi

MAGMA BLAS

Level 3 BLAS

5. Xgemm_tesla	General matrix-matrix product for Tesla
6. Xgemm_fermi	General matrix-matrix product for Fermi
7. Xtrsm_ tesla	Solves a triangular matrix problem on Tesla
8. Xtrsm_fermi	Solves a triangular matrix problem on Fermi
9. Xsyrk_tesla	Symmetric rank k update for Tesla
10. Xsyr2k_tesla	Symmetric rank 2k update for Tesla

- CUBLAS GEMMs for Fermi are based on the MAGMA implementation
- Further improvements
  – BACUGen - Autotuned GEMM for Fermi (J.Kurzak)
  – ZGEMM from 308 Gflop/s is now 341 Gflop/s

MAGMA BLAS

Other routines

11. Xswap	LU factorization; CPU interface
12. Xlacpy	LU factorization; GPU interface
13. Xlange	LU factorization on multicore (no GPUs)
14. Xlanhe	Cholesky factorization; CPU interface
15. Xtranspose	Cholesky factorization; GPU interface
16. Xinplace_transpose	Cholesky factorization on multicore (no GPUs)
17. Xpermute	QR factorization; CPU interface
18. Xauxiliary	QR factorization; GPU interface; with T matrices stored

Download

- Download MAGMA 1.0 RC5

  http://icl.cs.utk.edu/magma/software/

  and get file **magma_1.0.0-rc5.tar.gz**

- Prerequisites

  LAPACK
  BLAS
  CUDA Toolkit

Compile

- Provided are Makefiles for Linux | MacOS
- Modify **make.inc** in the main MAGMA directory

specifying the GPU family and the locations of the host LAPACK, host BLAS, and CUDA, e.g.,

```
#     GPU_TARGET specifies for which GPU you want to compile MAGMA
#     0: Tesla family
#     1: Fermi Family
GPU_TARGET = 1
…
LIB           = -lmkl_em64t -lguide -lpthread -lcublas -lcudart -lm
CUDADIR   = /mnt/scratch/cuda-4.0.11rc
LIBDIR       = -L/home/tomov/intel/mkl/10.0.1.014/lib/em64t  -L$(CUDADIR)/lib64
...
```

See examples make.inc.[acml | mkl | goto | atlas | accelerate | shared]

Using MAGMA 1.0 RC5

- Documentation

  http://icl.cs.utk.edu/magma/docs/
  [generated by Doxygen]

- Examples in directory **testing**, e.g.,

  > testing_**dgeqrf_gpu**
  device 0: Tesla C2050, 1147.0 MHz clock, 3071.7 MB memory
  device 1: Quadro NVS 290, 918.0 MHz clock, 255.7 MB memory

  Usage:
  testing_dgeqrf_gpu -M 1024 -N 1024

M	N	CPU GFlop/s	GPU GFlop/s	||R||_F / ||A||_F
1024	1024	24.20	51.44	2.040039e-15
2048	2048	26.51	111.74	2.662709e-15
3072	3072	27.50	151.65	3.256163e-15
4032	4032	29.96	183.02	3.546103e-15

...

FORTRAN Interface

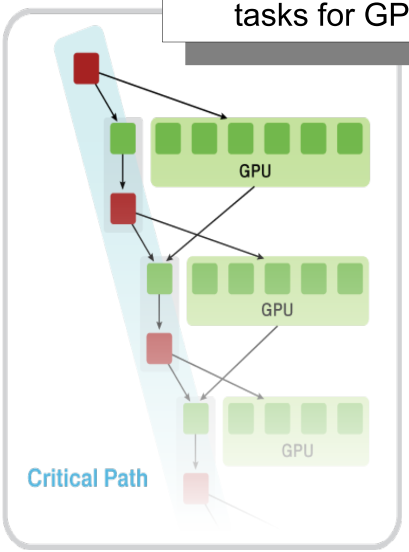
- Example in **testing_[zcds]getrf[_gpu]_f.f90**

```
        ...
        magma_devptr_t               :: devptrA, devptrB
        …
!------ Allocate GPU memory
        stat = cublas_alloc(ldda*n, sizeof_complex, devPtrA)
        …
!---- devPtrA = h_A
       call cublas_set_matrix(n, n, size_of_elt, h_A, lda, devptrA, ldda)
       …
       call magmaf_cgetrf_gpu(n, n, devptrA, ldda, ipiv, info)
```

Methodology overview

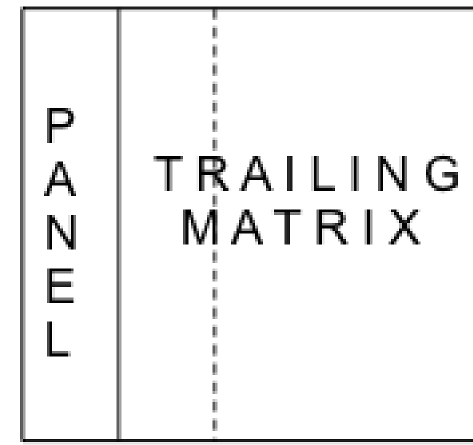
- MAGMA uses **HYBRIDIZATION** methodology based on
  - Representing linear algebra algorithms as collections of **TASKS** and **DATA DEPENDENCIES** among them
  - Properly **SCHEDULING** tasks' execution over multicore and GPU hardware components
- Successfully applied to fundamental linear algebra algorithms
  - One and two-sided factorizations and solvers
  - Iterative linear and eigen-solvers
- Productivity
  - High-level
  - Leveraging prior developments
  - Exceeding in performance homogeneous solutions

Hybrid CPU+GPU algorithms
(small tasks for multicores and large tasks for GPUs)



Statically Scheduled **One-Sided Factorizations** (LU, QR, and Cholesky)

- Hybridization
  - Panels (Level 2 BLAS) are factored on CPU using LAPACK
  - Trailing matrix updates (Level 3 BLAS) are done on the GPU using "look-ahead"



- Note
  - Panels are memory bound but are only $O(N^2)$ flops and can be overlapped with the $O(N^3)$ flops of the updates
  - In effect, the GPU is used only for the high-performance Level 3 BLAS updates, i.e., no low performance Level 2 BLAS is scheduled on the GPU

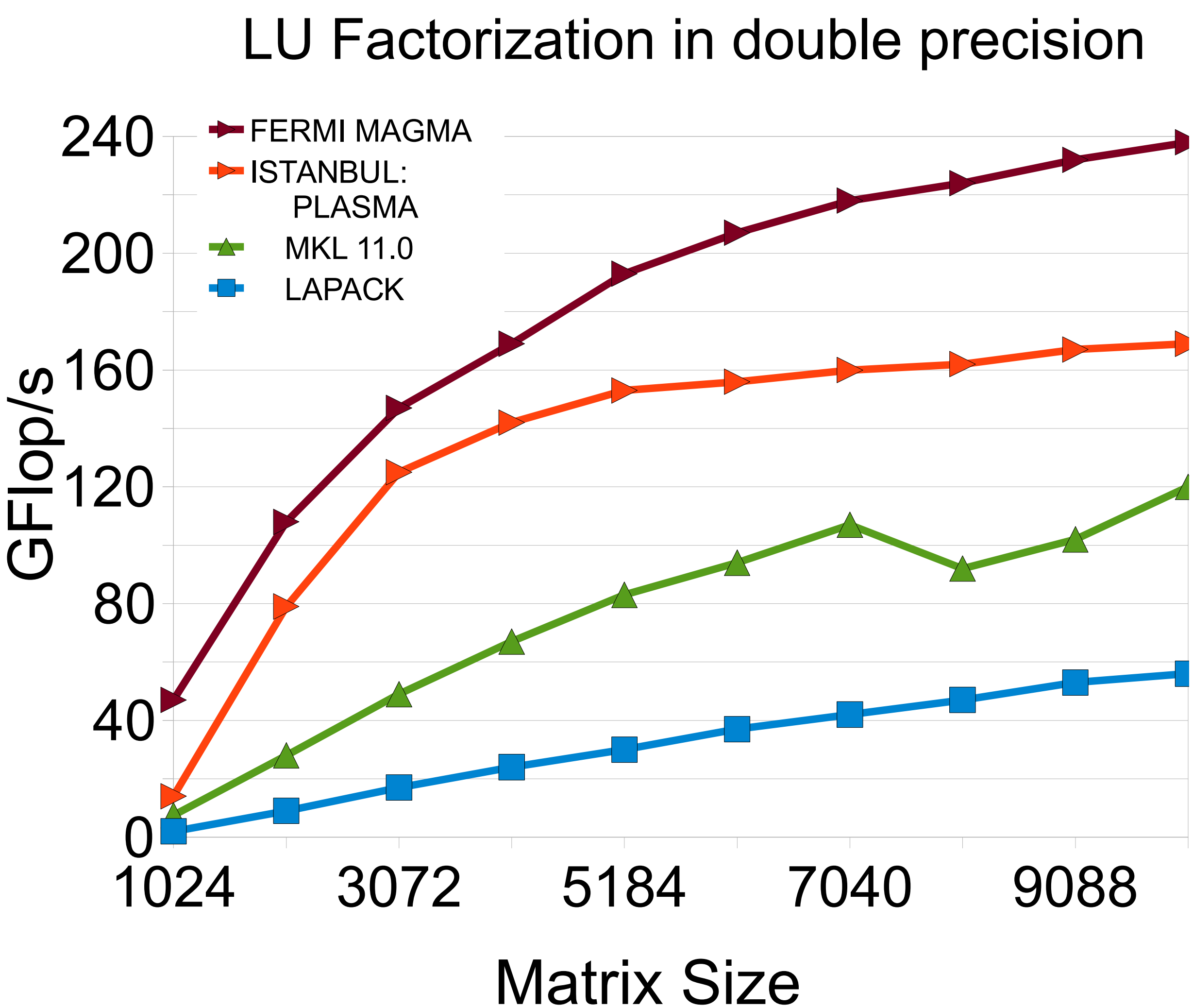
A hybrid algorithm example

- Left-looking hybrid Cholesky factorization in MAGMA

```
1  for (j = 0;  j < *n; j += nb) {
2    jb = min(nb, *n-j);
3    cublasSsyrk('l','n', jb, j, -1, da(j,0),*lda, 1, da(j,j),*lda);
4    cudaMemcpy2DAsync(work, jb*sizeof(float), da(j,j), *lda*sizeof(float),
5                      sizeof(float)*jb, jb, cudaMemcpyDeviceToHost, stream[1]);
6    if (j + jb < *n)
7       cublasSgemm('n','t', *n-j-jb, jb, j, -1, da(j+jb,0), *lda, da(j,0),
8                   *lda, 1, da(j+jb,j), *lda);
9    cudaStreamSynchronize(stream[1]);
10   spotrf_("Lower", &jb, work, &jb, info);
11   if (*info != 0)
12      *info = *info + j, break;
13   cudaMemcpy2DAsync(da(j,j), *lda*sizeof(float), work, jb*sizeof(float),
14                     sizeof(float)*jb, jb, cudaMemcpyHostToDevice, stream[0]);
15   if (j + jb < *n)
16      cublasStrsm('r','l','t','n', *n-j-jb, jb, 1, da(j,j), *lda,
17                  da(j+jb,j), *lda);
18 }
```

- The difference with LAPACK – the 3 additional lines colored in red
- Line 10 (done on CPU) is overlapped with work on the GPU (line 7)

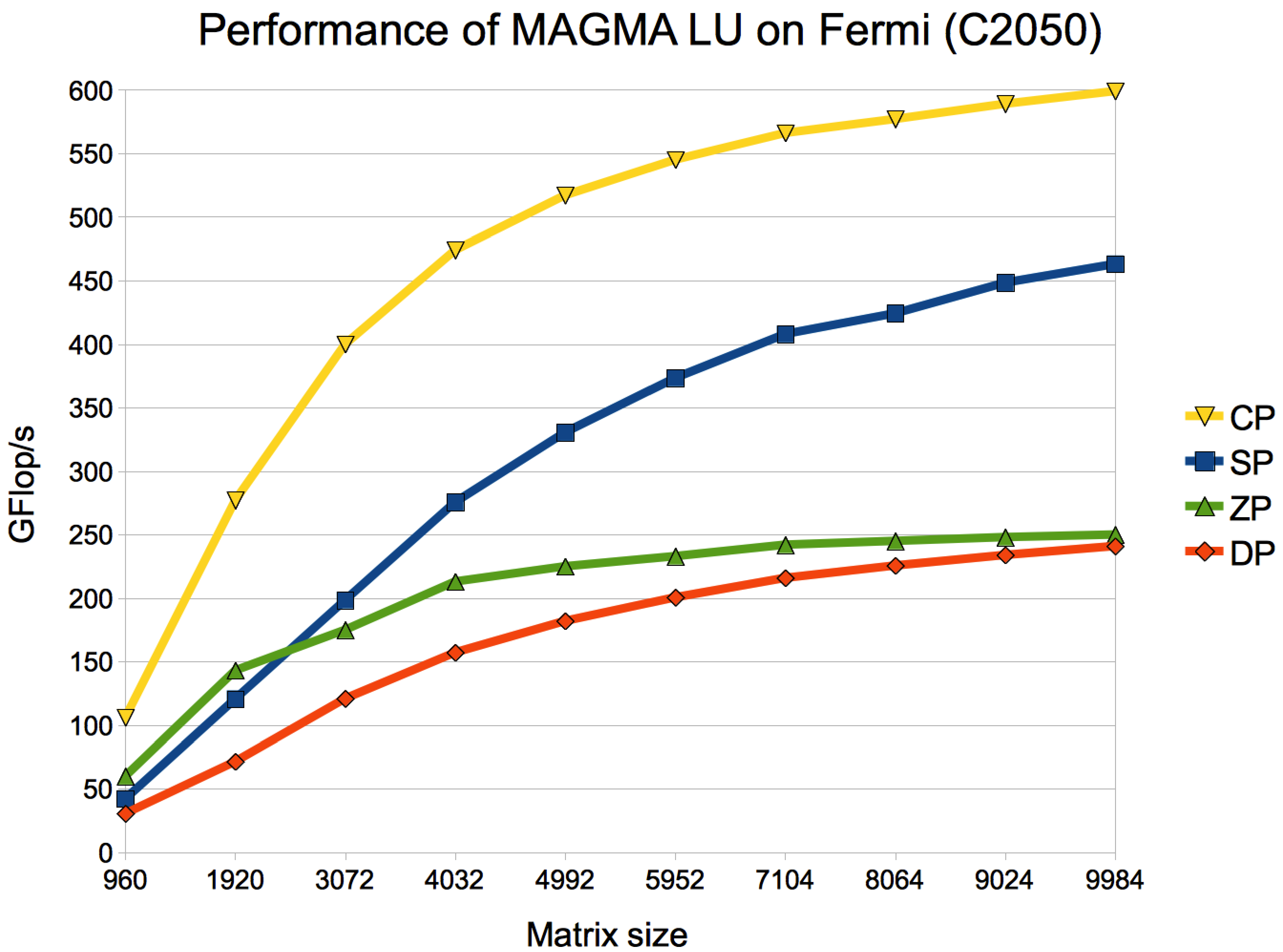
Results – one sided factorizations



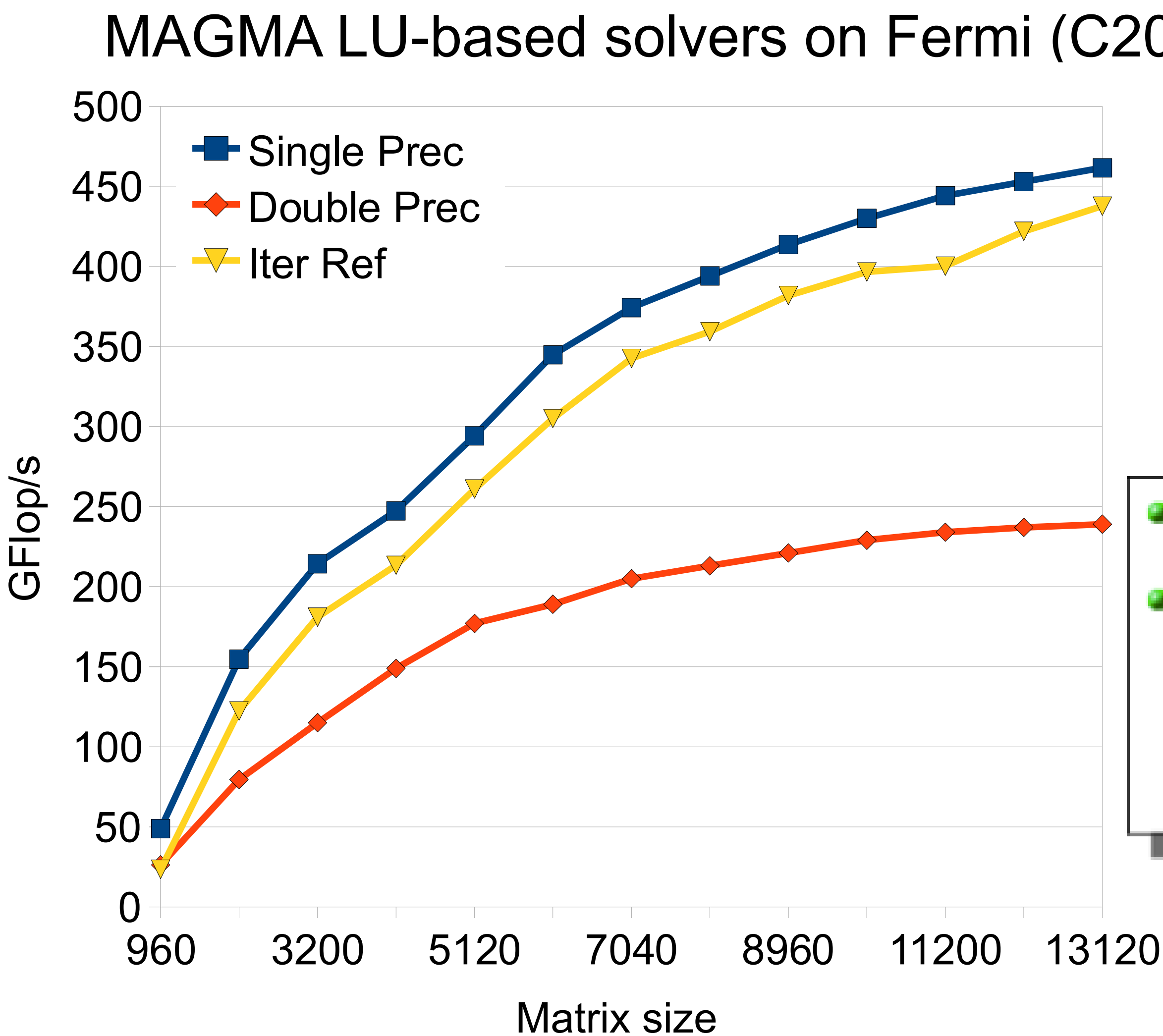
FERMI	Tesla C2050: 448 CUDA cores @ 1.15GHz SP/DP peak is 1030 / 515 GFlop/s
ISTANBUL	AMD 8 socket 6 core (48 cores) @2.8GHz SP/DP peak is 1075 / 538 GFlop/s

- Similar results for Cholesky & QR
- Fast solvers (several innovations)
  - in working precision, and
  - mixed-precision iter. refinement

  based on the one-sided factor.


Performance of MAGMA LU on Fermi (C2050)
GFlop/s
600
550
500
450
400
350
300
250
200
150
100
50
0
960
1920
3072
4032
4992
5952
7104
8064
9024
9984
Matrix size
CP
SP
ZP
DP

Results – linear solvers



FERMI Tesla C2050: 448 CUDA cores @ 1.15GHz
SP/DP peak is 1030 / 515 GFlop/s

- **Direct solvers**
  - Factor and solve in working precision
- **Mixed Precision Iterative Refinement**
  - Factor in single (i.e. the bulk of the computation in fast arithmetic) and use it as preconditioner in simple double precision iteration, e.g.
    $x_{i+1} = x_i + (LU_{SP})^{-1} P (b - A x_i)$

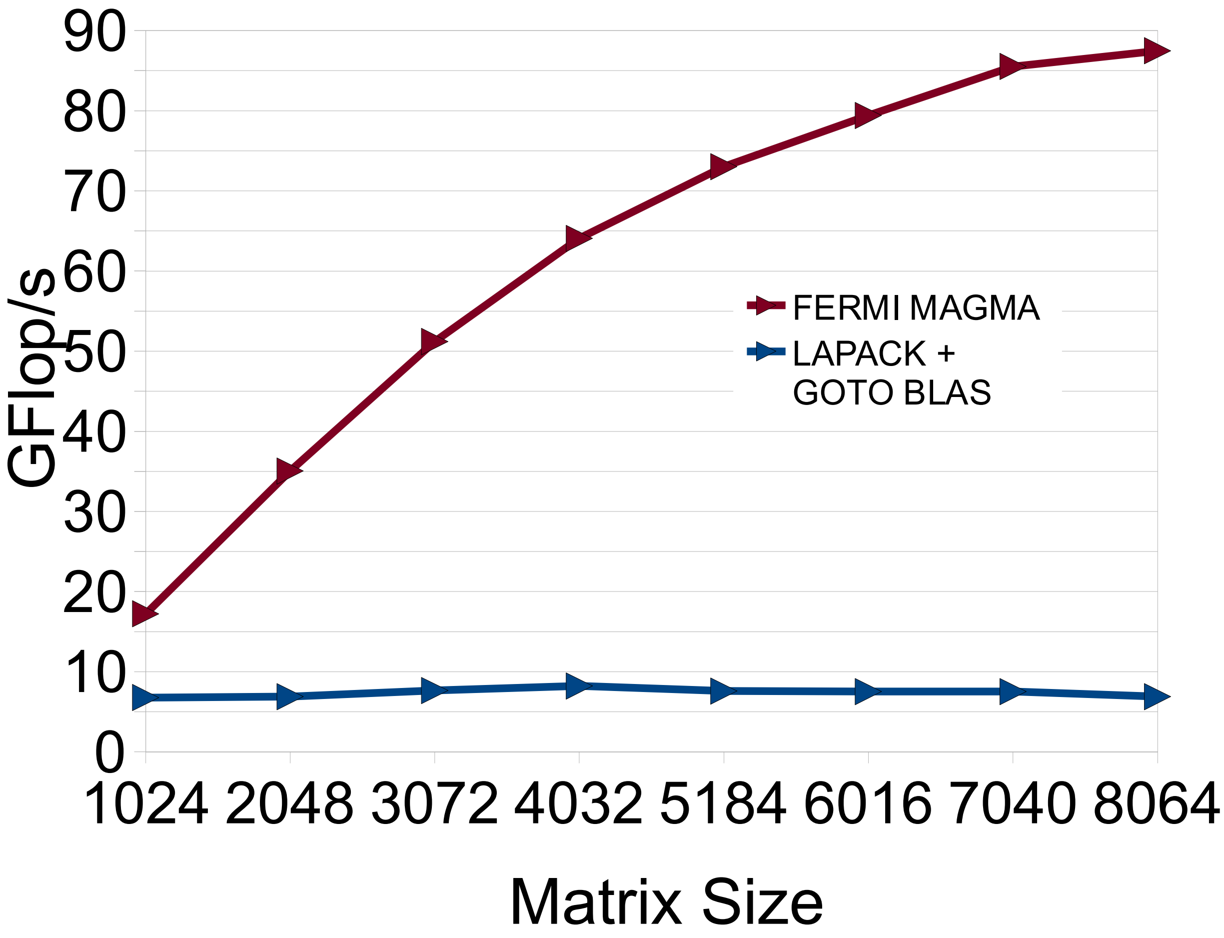
Statically Scheduled **Two-Sided Factorizations**
[Hessenber, tridiagonal, and bidiagonal reductions]

- Hybridization
  - Trailing matrix updates (Level 3 BLAS) are done on the GPU (similar to the one-sided factorizations)
  - Panels (Level 2 BLAS) are hybrid
    – operations with memory footprint restricted to the panel are done on CPU
    – The time consuming matrix-vector products involving the entire trailing matrix are done on the GPU

- Note
  - CPU-to-GPU communications and subsequent computations always stay in surface-to-volume ratio

Results – two sided factorizations

Hessenberg Factorization in double precision
[for the general eigenvalue problem]



FERMI	Tesla C2050: 448 CUDA cores @ 1.15GHz SP/DP peak is 1030 / 515 Gflop/s [system cost ~ $3,000]
ISTANBUL	AMD 8 socket 6 core (48 cores) @2.8GHz SP/DP peak is 1075 / 538 Gflop/s [system cost ~ $30,000]

- Similar accelerations for the bidiagonal factorization [for SVD] & tridiagonal factorization [for the symmetric eigenvalue problem]
- Similar acceleration (exceeding 10x) compared to other top-of-the-line multicore systems (including Nehalem-based) and libraries (including MKL, ACML)

Current and future work

- Hybrid algorithms
  - Further expend functionality
  - New highly parallel algorithms of optimized communication and synchronization
- OpenCL support
  - To be derived from OpenCL BLAS
- Autotuning framework
  - On both high level algorithms & BLAS
- Multi-GPU algorithms
  - StarPU scheduling

Conclusions

- ***Linear and eigenvalue solvers can be significantly accelerated on systems of multicore and GPU architectures***
- Many-core architectures with accelerators (e.g., GPUs) are the future of high performance scientific computing
- Challenge: Fundamental libraries will need to be redesigned/rewritten to take advantage of the emerging many-core architectures

Collaborators / Support

- **MAGMA** [Matrix Algebra on GPU and Multicore Architectures] team
  http://icl.cs.utk.edu/magma/

- **PLASMA** [Parallel Linear Algebra for Scalable Multicore Architectures] team
  http://icl.cs.utk.edu/plasma

- Collaborating partners

  University of Tennessee, Knoxville
  University of California, Berkeley
  University of Colorado, Denver

  INRIA, France
  KAUST, Saudi Arabia