Matrix Algebra on GPU and Multicore Architectures

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

Workshop on GPU-enabled Numerical Libraries
University of Basel, Switzerland
May 11-13, 2011

Outline

- PART I
  - Introduction to MAGMA
  - Methodology
  - Performance
- PART II
  - Hands-on training
  - Using and contributing to MAGMA
  - Examples

Part I: Outline

- Motivation
- MAGMA – LAPACK for GPUs
  - Overview
  - Methodology
  - MAGMA with StarPU / PLASMA / Quark
- MAGMA BLAS
- Sparse iterative linear algebra
- Current & future work directions
- Conclusions

Part I: Outline Goals

- Motivation [Hardware to Software Trends]
- MAGMA – LAPACK for GPUs
  - Overview [Learn what is available, how to use it, etc.]
  - Methodology [How to develop, e.g., hybrid algorithms]
  - MAGMA with StarPU / PLASMA / Quark [Development tools]
- MAGMA BLAS [Highly optimized CUDA kernels]
- Sparse iterative linear algebra [Methodology use in sparse LA]
- Current & future work directions
- Conclusions

About ICL

Last year ICL celebrated
20 years anniversary!

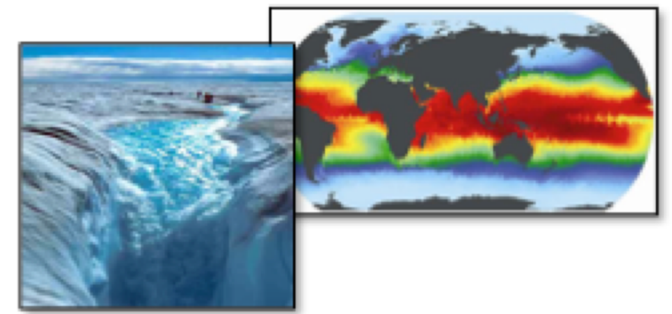
staff of more than
40 researchers, students, and administrators

Established by
Prof. Jack Dongarra

- Mission – provide leading edge tools, enable technologies and software for scientific computing, develop standards for scientific computing in general
- This includes standards and efforts such as
  PVM, MPI, LAPACK, ScaLAPACK, BLAS, ATLAS, Netlib, Top 500, PAPI, NetSolve, and the Linpack Benchmark
- ICL continues these efforts with
  PLASMA, MAGMA, HPC Challenge, BlackJack, OpenMPI, and MuMI,
  as well as other innovative computing projects

Science and Engineering Drivers

- *Climate Change*: Understanding, mitigating and adapting to the effects of global warming
  - Sea level rise
  - Severe weather
  - Regional climate change
  - Geologic carbon sequestration
- *Energy*: Reducing U.S. reliance on foreign energy sources and reducing the carbon footprint of energy production
  - Reducing time and cost of reactor design and deployment
  - Improving the efficiency of combustion energy sources
- *National Nuclear Security*: Maintaining a safe, secure and reliable nuclear stockpile
  - Stockpile certification
  - Predictive scientific challenges
  - Real-time evaluation of urban nuclear detonation

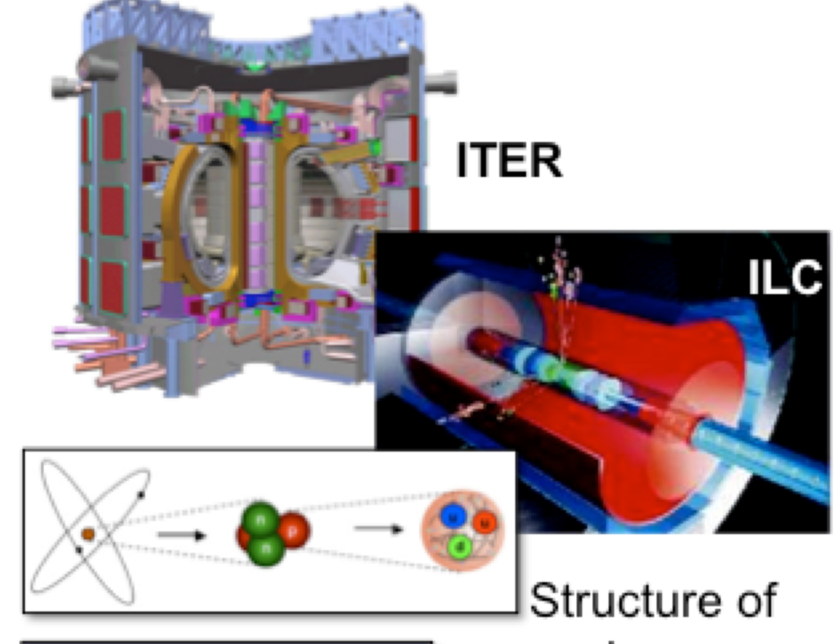


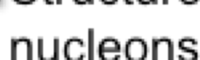
Accomplishing these missions requires exascale resources.

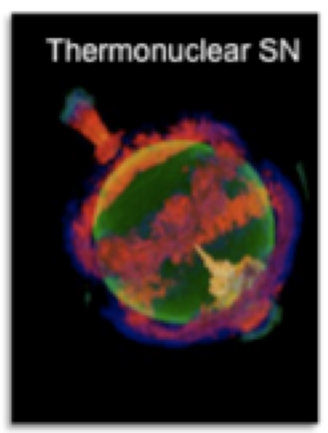
Simulation enables fundamental advances in basic science

- **Nuclear Physics**
  - Quark-gluon plasma & nucleon structure
  - Fundamentals of fission and fusion reactions
- **Facility and experimental design**
  - Effective design of accelerators
  - Probes of dark energy and dark matter
  - ITER shot planning and device control
- **Materials / Chemistry**
  - Predictive multi-scale materials modeling: observation to control
  - Effective, commercial, renewable energy technologies, catalysts and batteries
- **Life Sciences**
  - Better biofuels
  - Sequence *to* structure *to* function

These breakthrough scientific discoveries and facilities require exascale applications and resources.



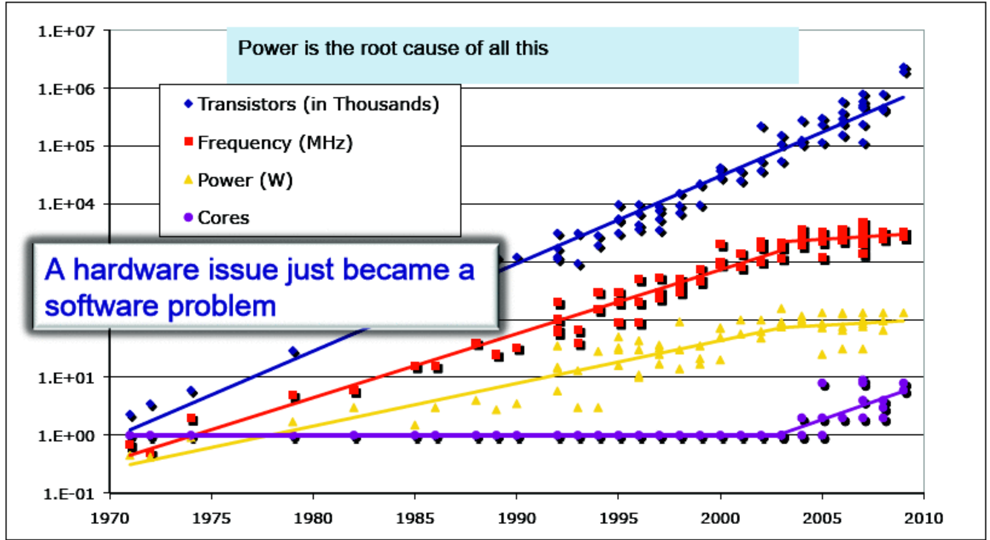




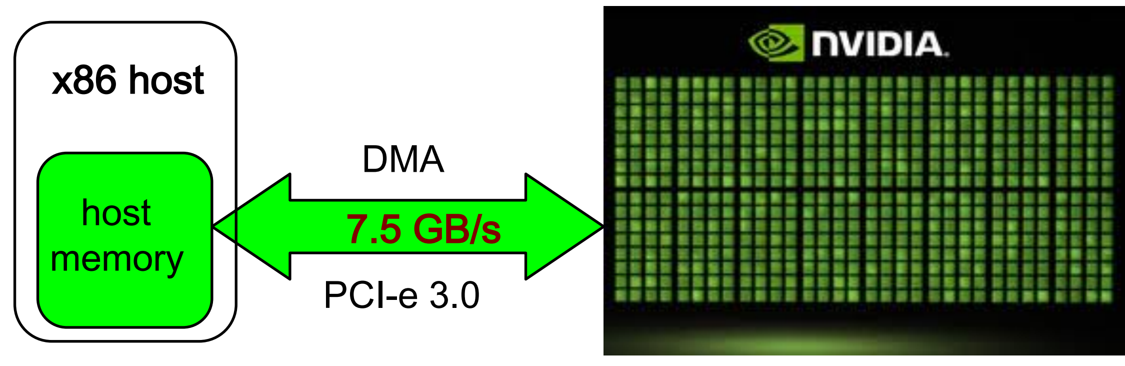
Hardware Trends

- 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)

Performance Has Also Slowed, Along with Power



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



Performance Development in Top500

1 Eflop/s
100 Pflop/s
10 Pflop/s
1 Pflop/s
100 Tflop/s
10 Tflop/s
1 Tflop/s
100 Gflop/s
10 Gflop/s
1 Gflop/s
100 Mflop/s

N=1

N=500

Gordon Bell Winners

1994 1996 1998 2000 2002 2004 2006 2008 2010 2012 2014 2016 2018 2020

36rd List: The TOP10

Rank	Site	Computer	Country	Cores	Rmax [Pflops]	% of Peak
1	Nat. SuperComputer Center in Tianjin	Tianhe-1A, NUDT Intel + Nvidia GPU + custom	China	186,368	2.57	55
2	DOE / OS Oak Ridge Nat Lab	Jaguar, Cray AMD + custom	USA	224,162	1.76	75
3	Nat. Supercomputer Center in Shenzhen	Nebulea, Dawning Intel + Nvidia GPU + IB	China	120,640	1.27	43
4	GSIC Center, Tokyo Institute of Technology	Tusbame 2.0, HP Intel + Nvidia GPU + IB	Japan	73,278	1.19	52
5	DOE / OS Lawrence Berkeley Nat Lab	Hopper, Cray AMD + custom	USA	153,408	1.054	82
6	Commissariat a l'Energie Atomique (CEA)	Tera-10, Bull Intel + IB	France	138,368	1.050	84
7	DOE / NNSA Los Alamos Nat Lab	Roadrunner, IBM AMD + Cell GPU + IB	USA	122,400	1.04	76
8	NSF / NICS U of Tennessee	Kraken, Cray AMD + custom	USA	98,928	.831	81
9	Forschungszentrum Juelich (FZJ)	Jugene, IBM Blue Gene + custom	Germany	294,912	.825	82
10	DOE / NNSA LANL & SNL	Cielo, Cray AMD + custom	USA	107,152	.817	79

36rd List: The TOP10

Rank	Site	Computer	Country	Cores	Rmax [Pflops]	% of Peak	Power [MW]	GFlops/ Watt
1	Nat. SuperComputer Center in Tianjin	Tianhe-1A, NUDT Intel + Nvidia GPU + custom	China	186,368	2.57	55	4.04	636
2	DOE / OS Oak Ridge Nat Lab	Jaguar, Cray AMD + custom	USA	224,162	1.76	75	7.0	251
3	Nat. Supercomputer Center in Shenzhen	Nebulea, Dawning Intel + Nvidia GPU + IB	China	120,640	1.27	43	2.58	493
4	GSIC Center, Tokyo Institute of Technology	Tusbame 2.0, HP Intel + Nvidia GPU + IB	Japan	73,278	1.19	52	1.40	850
5	DOE / OS Lawrence Berkeley Nat Lab	Hopper, Cray AMD + custom	USA	153,408	1.054	82	2.91	362
6	Commissariat a l'Energie Atomique (CEA)	Tera-10, Bull Intel + IB	France	138,368	1.050	84	4.59	229
7	DOE / NNSA Los Alamos Nat Lab	Roadrunner, IBM AMD + Cell GPU + IB	USA	122,400	1.04	76	2.35	446
8	NSF / NICS U of Tennessee	Kraken, Cray AMD + custom	USA	98,928	.831	81	3.09	269
9	Forschungszentrum Juelich (FZJ)	Jugene, IBM Blue Gene + custom	Germany	294,912	.825	82	2.26	365
10	DOE / NNSA LANL & SNL	Cielo, Cray AMD + custom	USA	107,152	.817	79	2.95	277

Commodity plus Accelerators

Commodity	Accelerator (GPU)
Intel Xeon	NVIDIA C2050 "Fermi"
8 cores	448 "CUDA cores"
3 GHz	1.15 GHz
8*4 ops/cycle	448 ops/cycle
96 Gflop/s (DP)	515 Gflop/s (DP)

X86 Host

Host Memory

Thread Execution Control Unit

Thread Processors

Special Function Unit

Local Memory

Device Memory

DMA

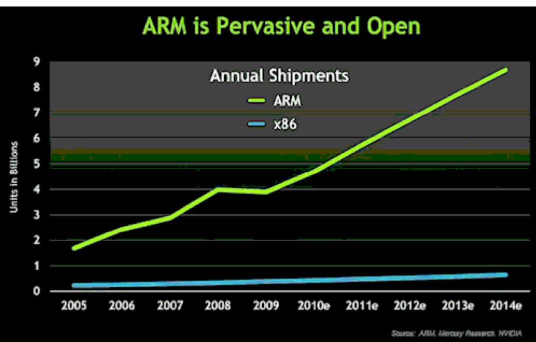
Interconnect
PCI-X 16 lane
64 Gb/s
1 GW/s

17 systems on the TOP500 use GPUs as accelerators

Future Computer Systems

- Most likely be a hybrid design
  - Think standard multicore chips and accelerator (GPUs)
- Today accelerators are attached
- Next generation more integrated
- Intel's MIC architecture "Knights Ferry" and "Knights Corner" to come.
  - 48 x86 cores
- AMD's Fusion in 2012 - 2013
  - Multicore with embedded graphics ATI
- Nvidia's Project Denver plans to develop an integrated chip using ARM architecture in 2013.

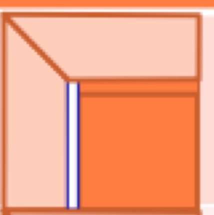
AMD
The future is fusion



Major change to Software

- **Must rethink the design of our software**
  - Another disruptive technology
    - Similar to what happened with cluster computing and message passing
  - Rethink and rewrite the applications, algorithms, and software
- **Numerical libraries for example will change**
  - For example, both LAPACK and ScaLAPACK will undergo major changes to accommodate this

A New Generation of Software

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations

A New Generation of Software

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

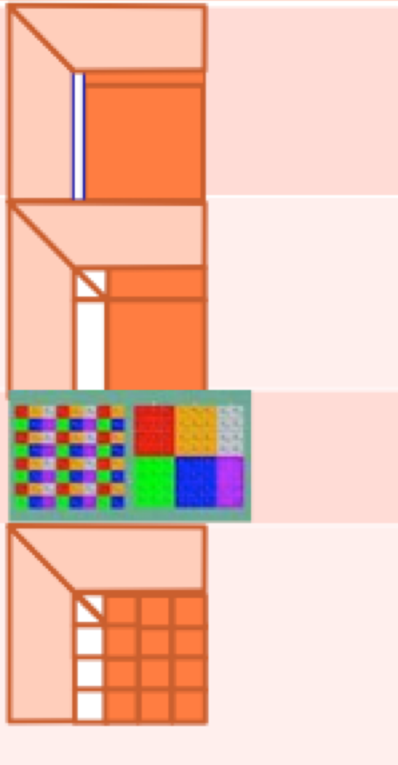
A New Generation of Software

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

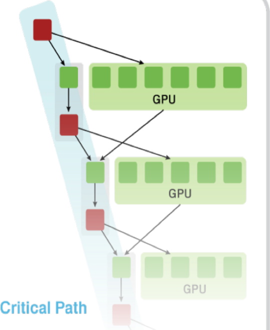
A New Generation of Software

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

A New Generation of Software

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

Challenges of using GPUs

- ## High levels of parallelism
  Many GPU cores
  [e.g. Tesla C2050 (Fermi) has 448 CUDA cores]

- ## Hybrid/heterogeneous architectures
  Match algorithmic requirements to architectural strengths
  [e.g. small, non-parallelizable tasks to run on CPU, large and parallelizable on GPU]

- ## Compute *vs* communication gap
  Exponentially growing gap; persistent challenge
  [Processor speed improves 59%, memory bandwidth 23%, latency 5.5%]
  [on all levels, e.g. a GPU Tesla C1070 (4 x C1060) has compute power of O(1,000) Gflop/s but GPUs communicate through the CPU using O(1) GB/s connection]

Matrix Algebra on GPU and Multicore Architectures (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 and extends its functionality to hybrid systems (w/ GPUs);
  - **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
  - U of Tennessee, **Knoxville**; U of California, **Berkeley**; U of Colorado, **Denver**
  - INRIA Bordeaux - Sud Ouest & INRIA Paris – Saclay, France; KAUST, Saudi Arabia
  - **Community effort [**similarly to the development of LAPACK / ScaLAPACK]

Parallel Linear Algebra Software for Multicore Architectures

PLASMA

Parallel Linear Algebra Software for Multicore Architectures

- Asychronicity
  - Avoid fork-join (Bulk sync design)
- Dynamic Scheduling
  - Out of order execution
- Fine Granularity
  - Independent block operations
- Locality of Reference
  - Data storage – Block Data Layout

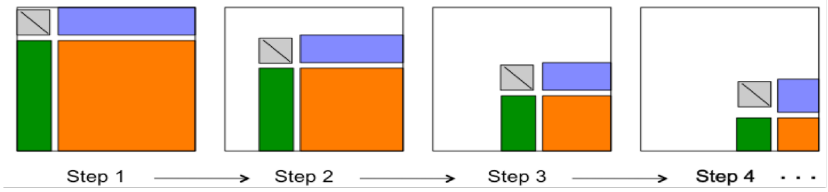
LAPACK LU

Step 1 → Step 2 → Step 3 → Step 4 . . .

DGETF2 (Factor a panel)		LAPACK
DLSWP (Backward swap)		LAPACK
DLSWP (Forward swap)		LAPACK
DTRSM (Triangular solve)		BLAS
DGEMM (Matrix multiply)		BLAS

- fork join
- bulk synchronous processing

Parallel tasks in LU

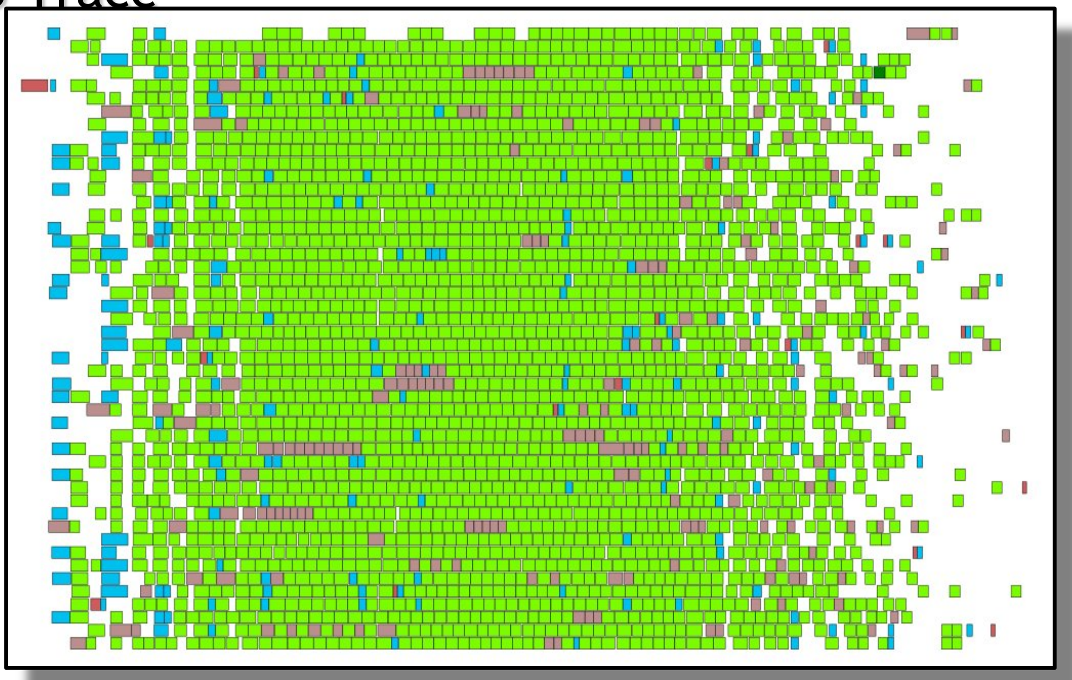


- **Idea:** break into smaller tasks and remove dependencies
- **Objectives:** high utilization of each core, scaling to large number of cores
- **Methodology:** Arbitrary DAG scheduling, Fine granularity / block data layout

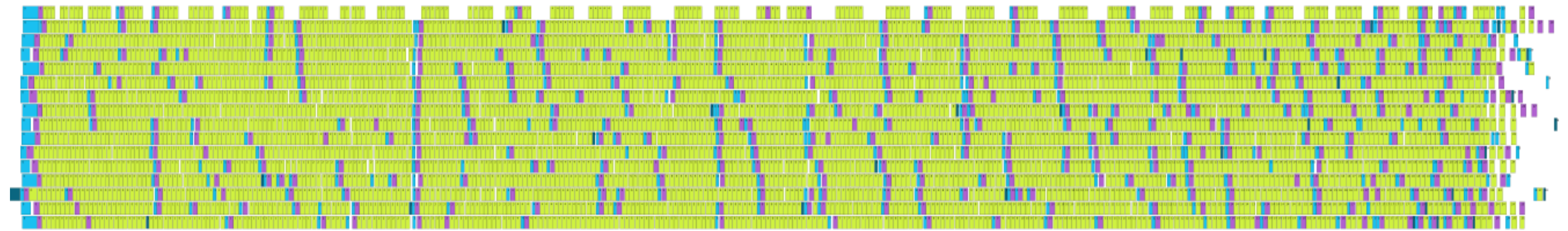
PLASMA Scheduling

Dynamic Scheduling: Tile LU Trace

- Regular trace
- Factorization steps pipelined
- Stalling only due to natural load imbalance



8-socket, 6-core (48 cores total) AMD Istanbul 2.8 GHz



quad-socket quad-core Intel Xeon 2.4 GHz

Pipelining: Cholesky Inversion

POTRF

TRTRI

LAUUM

POTRI

48 cores
POTRF, TRTRI and LAUUM.
The matrix is 4000 x 4000,tile size is 200 x 200,

Big DAGs: No Global Critical Path

- DAGs get very big, very fast
  - So windows of active tasks are used; this means no global critical path
  - Matrix of NBxNB tiles; NB^3 operation
    - NB=100 gives 1 million tasks

PLASMA Performance (QR, 48 cores)

QR Performance (double prec.)

Gflop/s

300
250
200
150
100
50
0

0
2000
4000
6000
8000
10000
12000

Size

PLASMA
MKL 11.0
LAPACK

MAGMA Software Stack

	CPU	HYBRID	GPU
distr.	Tile & LAPACK Algorithms with DAGuE		
multi	MAGNUM / Rectangular / PLASMA Tile Algorithms		
	PLASMA / Quark		StarPU
	LAPACK Algorithms and Tile Kernels		
single	MAGMA 1.0		
		MAGMA SPARSE	
		MAGMA BLAS	
	LAPACK		BLAS
	BLAS		CUDA

Linux, Windows, Mac OS X | C/C++, Fortran | Matlab, Python

MAGMA 1.0

- 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 the previous slide)**

MAGMA BLAS

- Subset of BLAS for a single NVIDIA GPU
- Optimized for MAGMA specific algorithms
- To complement CUBLAS on special cases

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

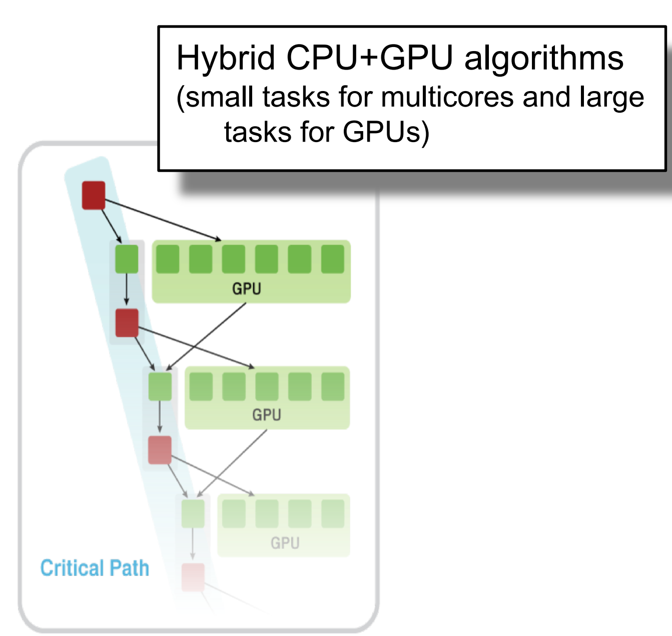
Other routines

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

Methodology overview

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



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 1.0

```
 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 in red
- Line 10 (done on CPU) is overlapped with work on the GPU (line 7)

Hybrid algorithms

QR factorization in single precision arithmetic, CPU interface

Performance of MAGMA *vs* MKL

GFlop/s: 0, 40, 80, 120, 160, 200, 240, 280, 320

Matrix size x 1000: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

MAGMA
MKL 8 cores
MKL 1 core

MAGMA QR time breakdown

Time: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%

Matrix size x 1000: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Overhead
CPU
CPU+GPU
GPU

GPU : NVIDIA GeForce GTX 280 (240 cores @ 1.30GHz)
CPU : Intel Xeon dual socket quad-core (8 cores @2.33 GHz)

GPU BLAS : CUBLAS 2.2, sgemm peak: 375 GFlop/s
CPU BLAS : MKL 10.0 , sgemm peak: 128 GFlop/s

[for more performance data, see http://icl.cs.utk.edu/magma]

Results – one sided factorizations

LU Factorization in double precision

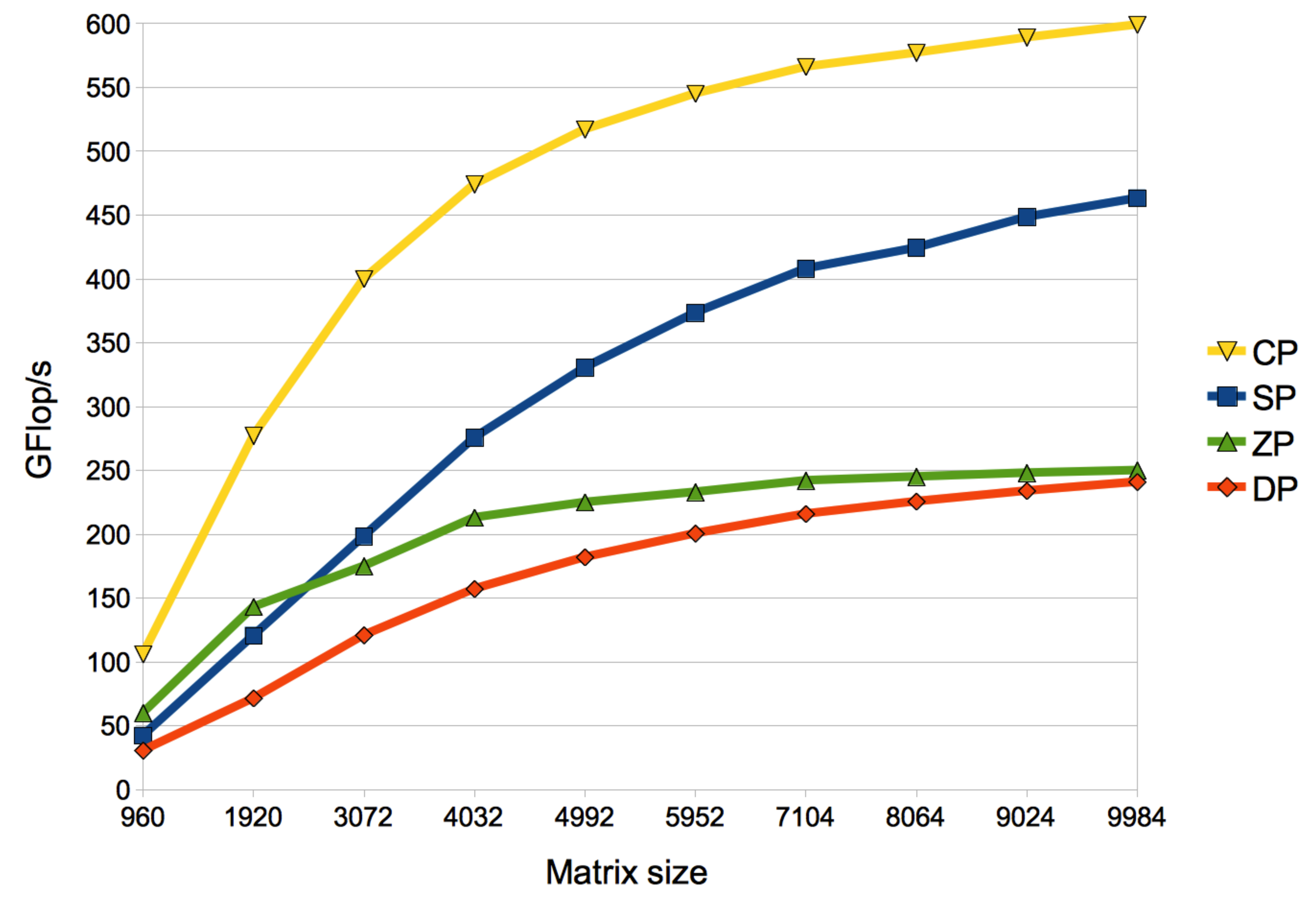
Matrix Size	FERMI MAGMA	ISTANBUL: PLASMA	MKL 11.0	LAPACK

(Chart axes: GFlop/s 0–240; Matrix Size 1024, 3072, 5184, 7040, 9088)

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
0
50
100
150
200
250
300
350
400
450
500
550
600
960
1920
3072
4032
4992
5952
7104
8064
9024
9984
Matrix size
CP
SP
ZP
DP

Mixed Precision Methods

Mixed Precision Methods

- Mixed precision, use the lowest precision required to achieve a given accuracy outcome
  - Improves runtime, reduce power consumption, lower data movement
  - Reformulate to find correction to solution, rather than solution [Δx rather than x].

Idea Goes Something Like This…

- Exploit 32 bit floating point as much as possible.
  - Especially for the bulk of the computation
- Correct or update the solution with selective use of 64 bit floating point to provide a refined results
- Intuitively:
  - Compute a 32 bit result,
  - Calculate a correction to 32 bit result using selected higher precision and,
  - Perform the update of the 32 bit results with the correction using high precision.

Mixed-Precision Iterative Refinement

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

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

- **Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.**

Mixed-Precision Iterative Refinement

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

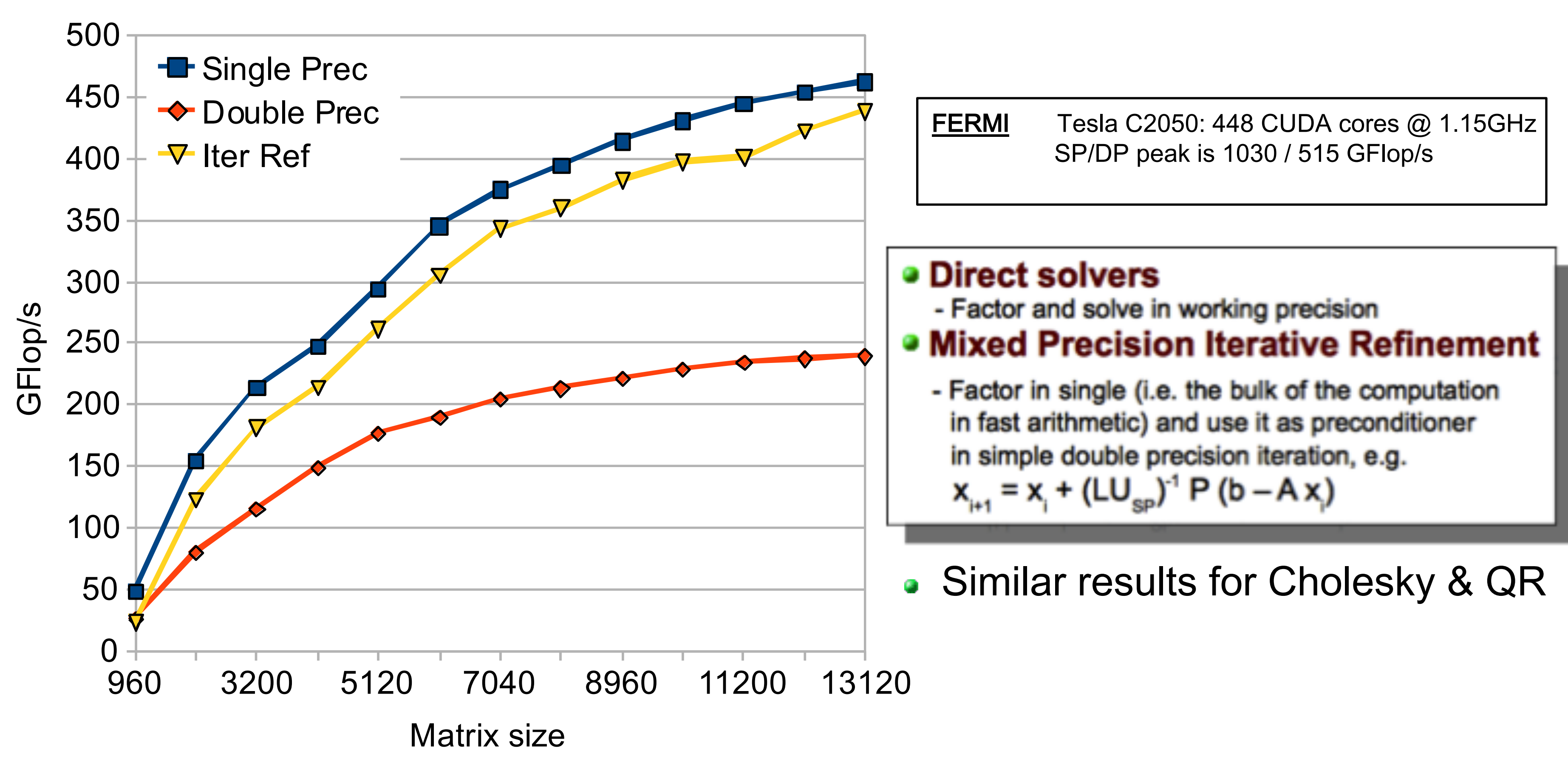
L U = lu(A)	SINGLE	$O(n^3)$
x = L\(U\b)	SINGLE	$O(n^2)$
r = b – Ax	DOUBLE	$O(n^2)$
WHILE || r || not small enough		
z = L\(U\r)	SINGLE	$O(n^2)$
x = x + z	DOUBLE	$O(n^1)$
r = b – Ax	DOUBLE	$O(n^2)$
END		

- 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; $O(10^8)$

Results – linear solvers

MAGMA LU-based solvers on Fermi (C2050)



Two-sided matrix factorizations

- Two-sided factorizations

  $Q' A Q = H$, H – upper Hessenberg / tridiagonal,

  $Q' A P = B$, B – bidiagonal

  Q and P – orthogonal similarity transformations

- **Importance**

One-sided factorizations - bases for linear solvers	**Two-sided factorizations** - bases for eigen-solvers

- Block algorithm

  Q – a product of n-1 elementary reflectors

  $Q = H_1 H_2 \ldots H_{n-1}$, $H_i = I - \tau_i v_i v_i'$

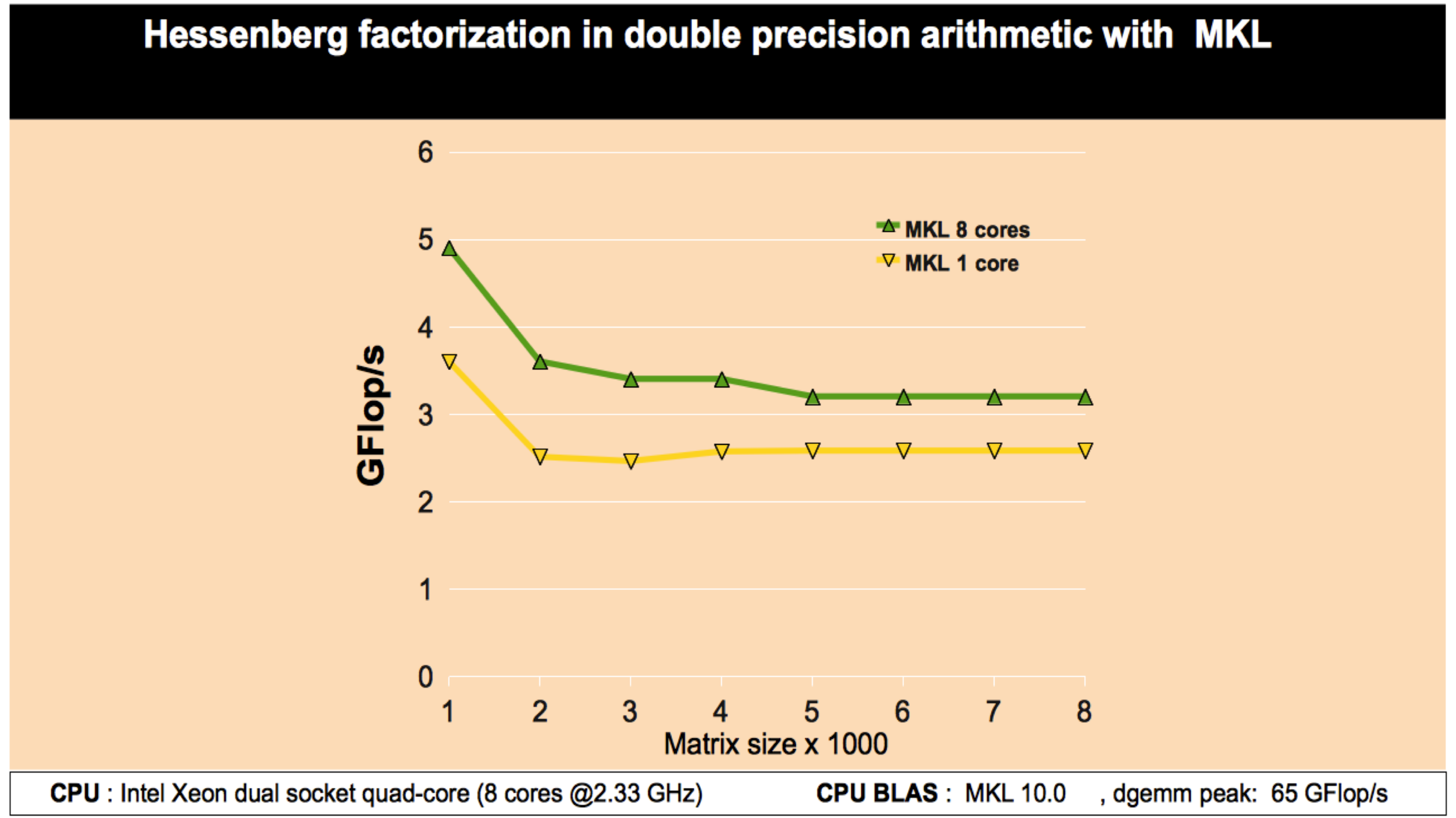
  $H_1 \ldots H_{nb} = I - V T V'$ (*WY transform*; the bases for delayed update or block algorithm)

- **Can we accelerate it ?**

  [similarly to the one-sided using hybrid GPU-based computing]

  [to see **much higher acceleration** due to a removed bottleneck]

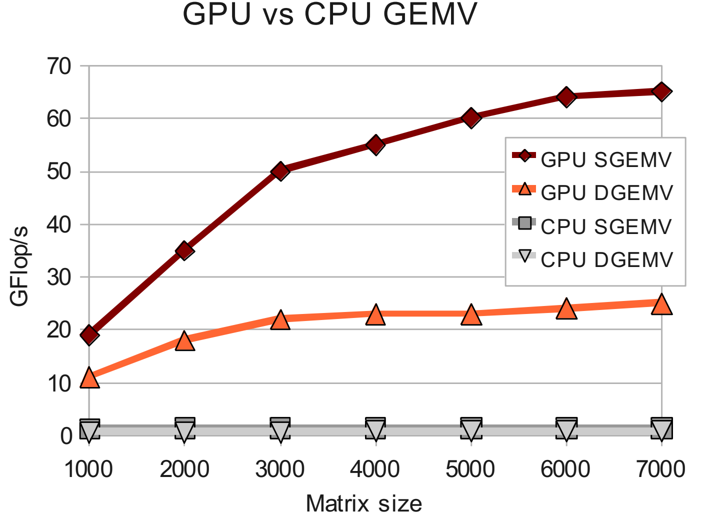
Homogeneous multicore acceleration?



- There have been difficulties in accelerating it on homogeneous multicores

Two-sided matrix factorizations

- Used in singular-value and eigen-value problems
- LAPACK-based two-sided factorizations are rich in Level 2 BLAS and therefore can not be properly accelerated on multicore CPUs
- We developed hybrid algorithms exploring GPUs' high bandwidth



GPU: GTX280 (240 cores @ 1.30GHz, 141 GB/s)
CPU: 2 x 4 cores Intel Xeon @ 2.33GHz, 10.4 GB/s)

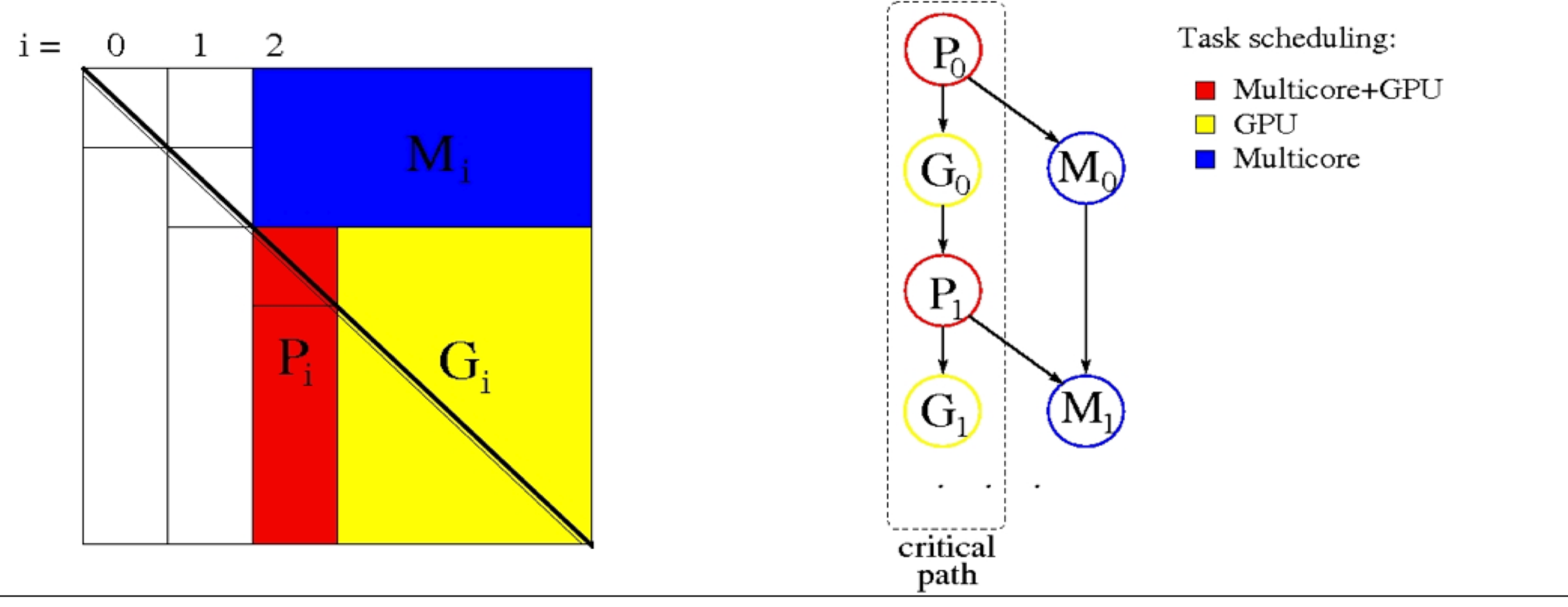
High-performance CUDA kernels were developed for various matrix-vector products
[e.g., **ssymv** reaching up to **102 Gflop/s** for the symmetric eigenvalue problem]

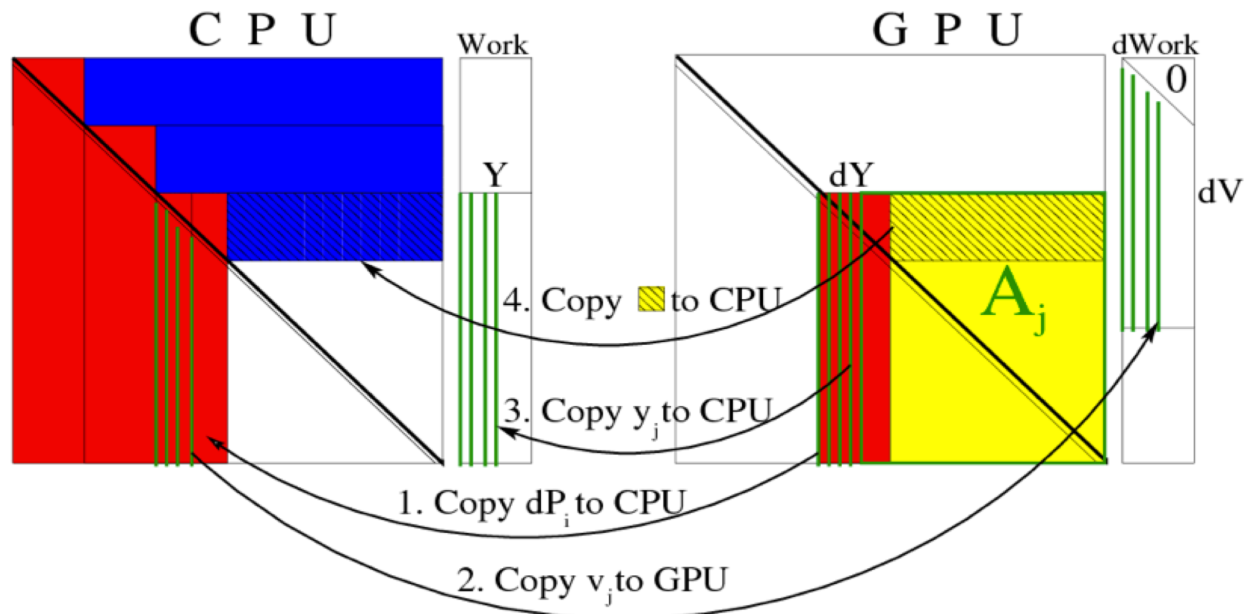
Statically Scheduled **Two-Sided Factorizations**
[Hessenber, tridiagonal, and bidiagonal reductions]

- **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

Task Splitting & Task Scheduling





Results – two sided factorizations

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

GFlop/s

90
80
70
60
50
40
30
20
10
0

FERMI MAGMA
LAPACK + GOTO BLAS

1024 2048 3072 4032 5184 6016 7040 8064

Matrix Size

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)

Results – two sided factorizations

GPU Performance in SP

26 x

12 x

22 x

GFlop/s

160
140
120
100
80
60
40
20
0

1024 4032 7040 10112 13024

Matrix size

HR
Tridiag.
Bidiag.

Multicore MKL Performance in SP

GFlop/s

50
45
40
35
30
25
20
15
10
5
0

1024 3072 5184 7040 9088

Matrix size

Hessenberg
Tridiagonalization
Bidiagonalization

GPU : NVIDIA GeForce GTX 280 (240 cores @ 1.30GHz)
CPU : Intel Xeon dual socket quad-core (8 cores @2.33 GHz)

Complete eigensolvers
Divide & Conquer

(Christof Vomel)

Phase 1: tridiagonalization

25.5 x

Matrix Size	MAGMA	MKL 8	MKL 1

(GFlop/s vs. Matrix Size: 1024, 4032, 7040, 10112, 13024; legend: MAGMA, MKL 8, MKL 1; y-axis 0–120)

Phase 2 : D & C

9.2 x

(Speedup vs. Matrix Size: 1024, 4032, 7040, 10112, 13024; legend: MAGMA, MKL; y-axis 0–10)

Phase 3 : eigenvectors

26.3 x

(GFlop/s vs. Matrix Size: 1024, 4032, 7040, 10112, 13024; legend: MAGMA, MKL 8, MKL 1; y-axis 0–400)

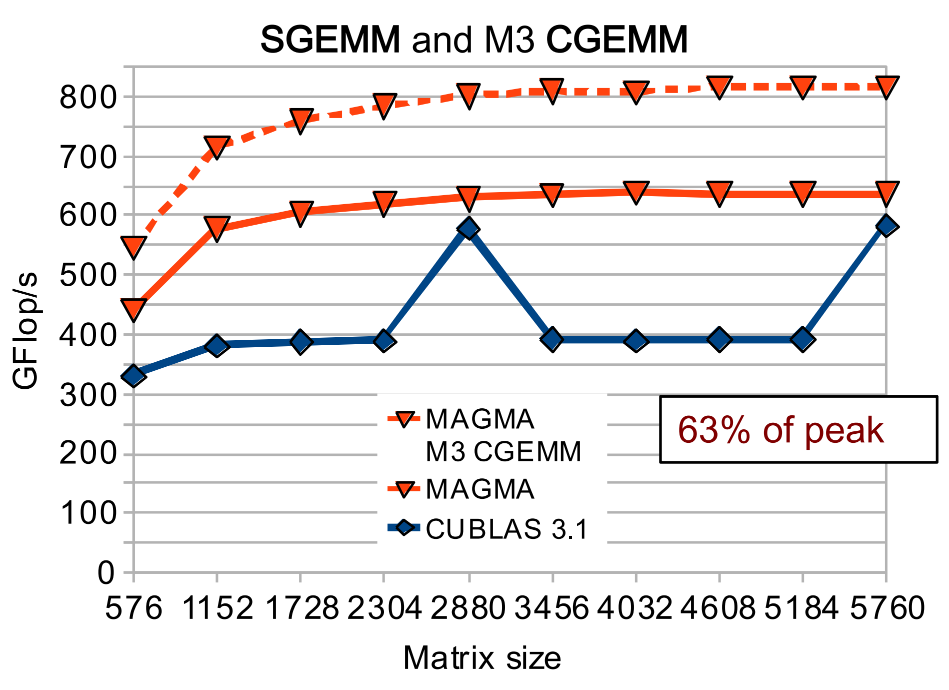
GPU : NVIDIA GeForce GTX 280 (240 cores @ 1.30GHz)
CPU : Intel Xeon dual socket quad-core (8 cores @2.33 GHz)

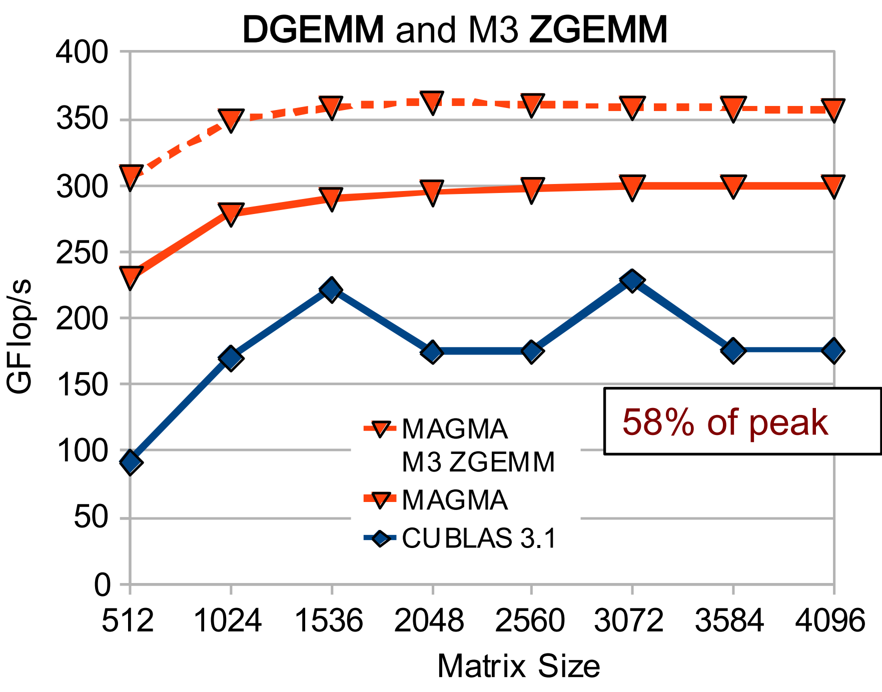
MAGMA BLAS

MAGMA BLAS

- Performance critically depend on BLAS
- How to develop fast CUDA BLAS?
- GEMM and SYMV examples

GEMM for Fermi





Tesla C2050 (Fermi): 448 CUDA cores @ 1.15GHz, theoretical SP peak is 1.03 Tflop/s, DP is 515 GFlop/s)

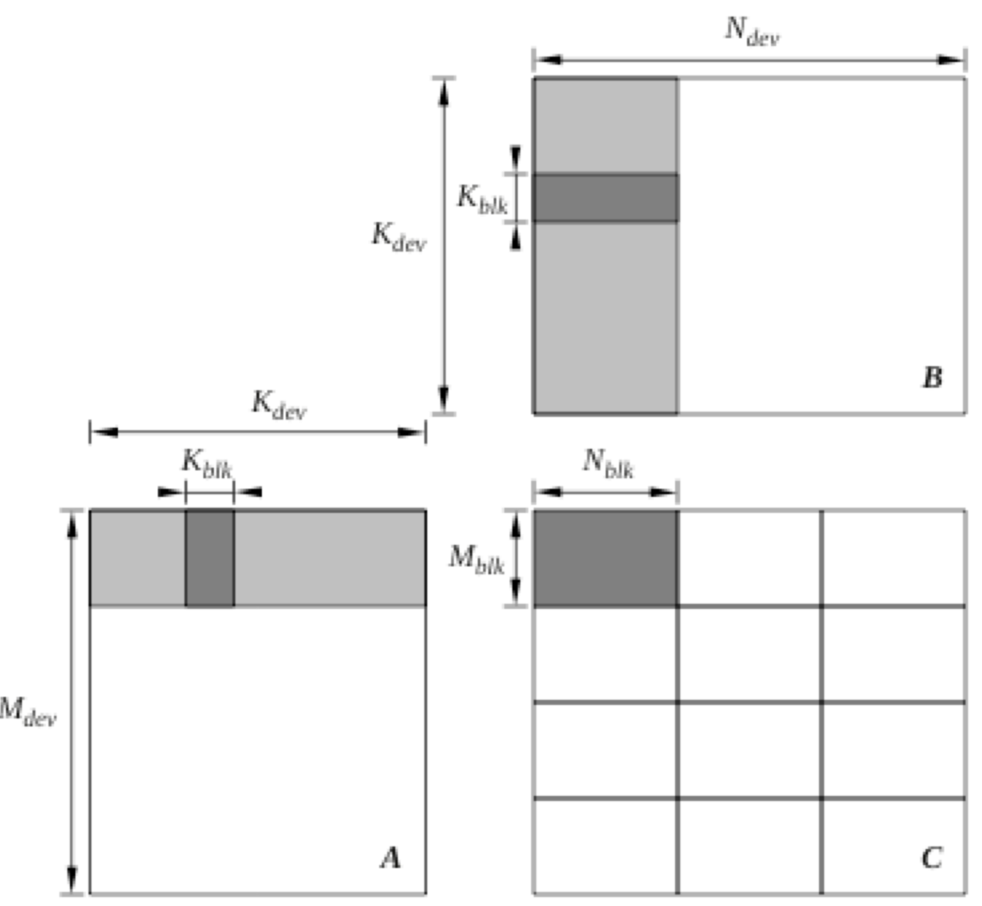
- CUBLAS 3.2 GEMM are based on these kernels
- TRSM and other Level 3 BLAS based on GEMM
- Auto-tuning has become more important
  - e.g., for BLAS, for higher-level hybrid algorithms, and for an OpenCL port

Autotuning

BACUGen – autotuning of GEMM

(J. Kurzak, UTK)

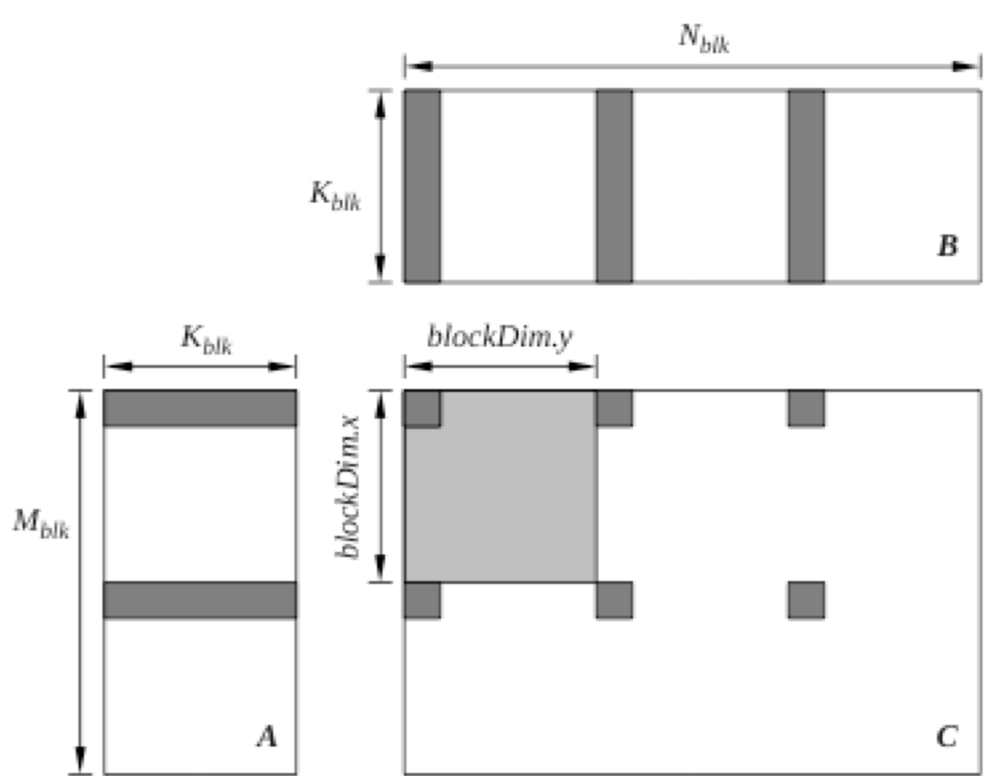
$C = \alpha A B + \beta C$



Top-level view of the algorithm

- Two levels of parallelism
  - Grid of thread blocks [coarse-level data parallelism]
  - Thread block [fine-level parallelism within a block]
- Parameterized template to generate many versions
  - including **shared memory and register blocking**
- Empirically find the "best" version

BACUGen – autotuning of GEMM



Thread-level view of the algorithm

- Parallelism in **a thread block**
  [blockDim.x x blockDim.y threads]
- A thread in this example computes 6 elements
- Register blocking
  - In this example:
    2 + 3 elements are loaded in registers (from shared memory) and reused in
    2 x 3 multiply-adds

BACUGen – autotuning of GEMM

number of variants

0 100 200 300 400 500

SGEMM: $A \times B$, $A \times B^T$, $A^T \times B^T$, $A^T \times B$

CGEMM: $A \times B$, $A \times B^T$, $A^T \times B^T$, $A^T \times B$

DGEMM: $A \times B$, $A \times B^T$, $A^T \times B^T$, $A^T \times B$

ZGEMM: $A \times B$, $A \times B^T$, $A^T \times B^T$, $A^T \times B$

- Number of variants generated and tested

BACUGen – autotuning of GEMM

CGEMM
SGEMM
ZGEMM
DGEMM
Gflop/s
900
800
700
600
500
400
300
200
100
0
0
2000
4000
6000
8000
10000
matrix size

- Performance on Fermi (C2050) in Gflop/s
- ZGEMM improved significantly compared to CUBLAS
  - from 308 to 341 Gflop/s
- Improvement up to 2x on some specific matrices (e.g., of “rectangular” shape)

SYMV example

$y = \alpha A x + \beta y$, where A is a symmetric matrix

- **Memory bound kernel**
  n^2 sizeof(data_type) B -> 2 n^2 flops
  => theoretical SSYMV peak on a 142 GB/s bus (e.g., in GTX280) is 142 Gflop/s
- **“Irregular” data accesses**
- **O(1) Gflop/s with CUBLAS**
  - **What can be done?**

SYMV example

- **Explore the symmetry**

N^2 / NB work space

A

x

y

x_1

$A'_{31} x_3$

NB

A_{31}

x_3

$A_{31}\ x_1$

= + + + + + =

1 2 3 4 5 6

1 2 3 4 5 6

SYMV example

Performance of SSYMV on GTX280

Gflop/s

100
90
80
70
60
50
40
30
20
10
0

n 64 128 192 256 320 384 448 512 576 704 832 960 1088 1216 1408 1600 1792 1984 2240 2496 2816 3136 3520 3904 4352 4800 5312 5888 6528 7232 8000 8832 9728 10752 11840 13056

Matrix size

cublas
magmablas

Multicore + multi-GPU algorithms

Multicore + multi-GPU algorithms

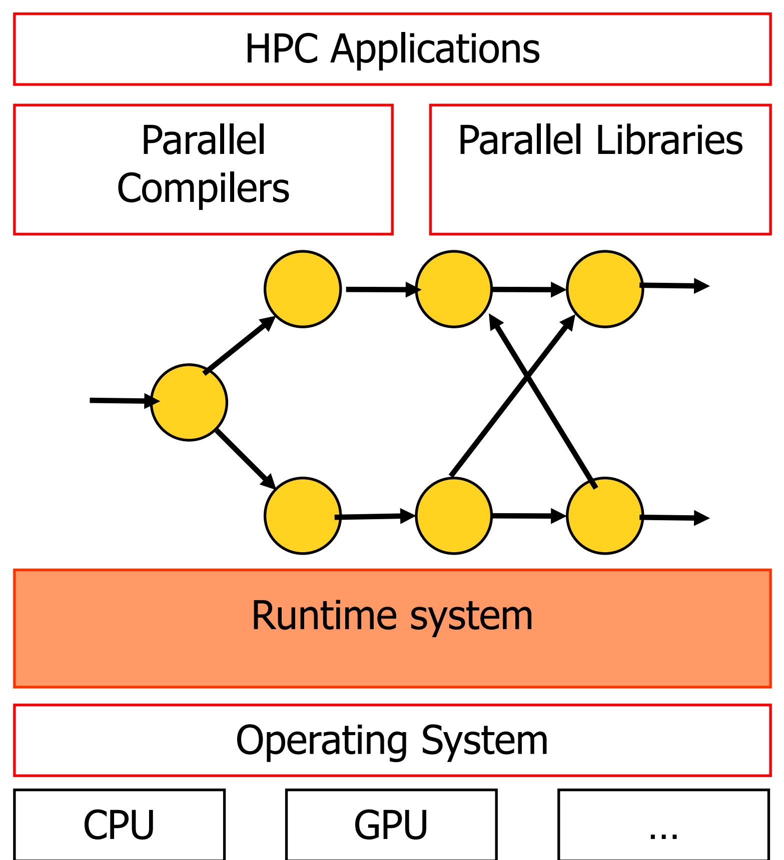
- Reuse already developed kernels
  - Hybrid MAGMA 1.0 for single GPU
  - PLASMA for multticore
- Use run time system to schedule (dynamically) the kernels' execution
  - StarPU
  - QUARK (from PLASMA)
  - …

The StarPU runtime system

The need for runtime systems

- **do dynamically what would be difficult to do statically**
- **Library that provides**
  - Task scheduling
  - Memory management

http://runtime.bordeaux.inria.fr/StarPU/



Productivity

- Develop parallel multicore + multiGPU algorithms from sequential algorithms

```
// Sequential Tile Cholesky
FOR k = 0..TILES-1
    DPOTRF(A[k][k])
    FOR m = k+1..TILES-1
        DTRSM(A[k][k], A[m][k])
        FOR n = k+1..TILES-1
            DSYRK(A[n][k], A[n][n])
            FOR m = n+1..TILES-1
                DGEMM(A[m][k], A[n][k], A[m][n])
```

```
// Hybrid Tile Cholesky
FOR k = 0..TILES-1
    starpu_Insert_Task(DPOTRF, …)
    FOR m = k+1..TILES-1
        starpu_Insert_Task(DTRSM, …)
        FOR n = k+1..TILES-1
            starpu_Insert_Task(DSYRK, …)
            FOR m = n+1..TILES-1
                starpu_Insert_Task(DGEMM, …)
```

- Example to be given w/ QUARK scheduler (in PART II)

Performance scalability

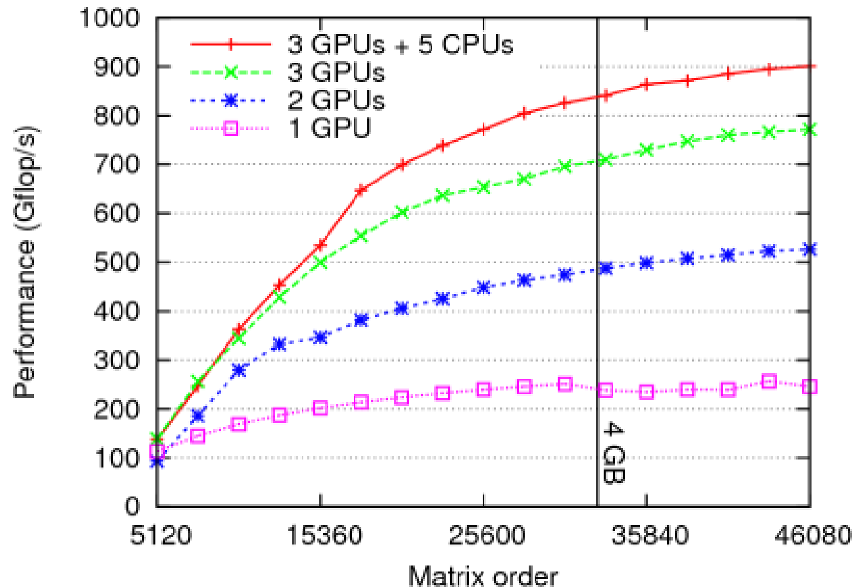
Statistics for codelet spotrf
- CUDA 0 (Quadro FX 5800) -> 3 / 36 (8.33 %)
- CUDA 1 (Quadro FX 5800) -> 1 / 36 (2.78 %)
- CUDA 2 (Quadro FX 5800) -> 3 / 36 (8.33 %)
- CPU 0 -> 6 / 36 (16.67 %)
- CPU 1 -> 9 / 36 (25.00 %)
- CPU 2 -> 4 / 36 (11.11 %)
- CPU 3 -> 6 / 36 (16.67 %)
- CPU 4 -> 4 / 36 (11.11 %)

Statistics for codelet ssyrk
- CUDA 0 (Quadro FX 5800) -> 41 / 630 (6.51 %)
- CUDA 1 (Quadro FX 5800) -> 40 / 630 (6.35 %)
- CUDA 2 (Quadro FX 5800) -> 49 / 630 (7.78 %)
- CPU 0 -> 105 / 630 (16.67 %)
- CPU 1 -> 85 / 630 (13.49 %)
- CPU 2 -> 105 / 630 (16.67 %)
- CPU 3 -> 102 / 630 (16.19 %)
- CPU 4 -> 103 / 630 (16.35 %)

Statistics for codelet strsm
- CUDA 0 (Quadro FX 5800) -> 125 / 630 (19.84 %)
- CUDA 1 (Quadro FX 5800) -> 127 / 630 (20.16 %)
- CUDA 2 (Quadro FX 5800) -> 122 / 630 (19.37 %)
- CPU 0 -> 50 / 630 (7.94 %)
- CPU 1 -> 52 / 630 (8.25 %)
- CPU 2 -> 52 / 630 (8.25 %)
- CPU 3 -> 54 / 630 (8.57 %)
- CPU 4 -> 48 / 630 (7.62 %)

Statistics for codelet sgemm
- CUDA 0 (Quadro FX 5800) -> 2258 / 7140 (31.62 %)
- CUDA 1 (Quadro FX 5800) -> 2182 / 7140 (30.56 %)
- CUDA 2 (Quadro FX 5800) -> 2261 / 7140 (31.67 %)
- CPU 0 -> 87 / 7140 (1.22 %)
- CPU 1 -> 94 / 7140 (1.32 %)
- CPU 2 -> 85 / 7140 (1.19 %)
- CPU 3 -> 85 / 7140 (1.19 %)
- CPU 4 -> 88 / 7140 (1.23 %)

Performance of Cholesky factorization in SP



SGEMM
gpu : 333.04 GFlop/s
cpu : 20.06 GFlop/s
STRSM
gpu : 59.46 GFlop/s
cpu : 18.96 GFlop/s
SSYRK
gpu : 298.74 GFlop/s
cpu : 19.50 GFlop/s
SPOTRF
gpu : 57.51 GFlop/s
cpu : 17.45 GFlop/s

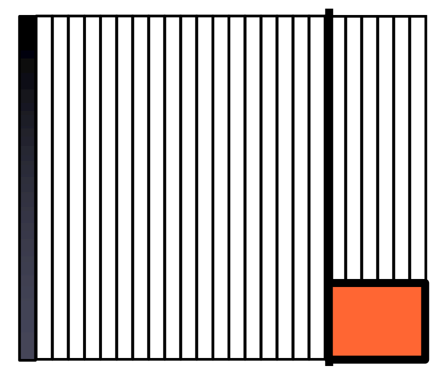
- 5 CPUs (Nehalem) + 3 GPUs (FX5800)
- Efficiency > 100%

PLASMA & MAGMA with StarPU

- **QR factorization**
  - **System: 16 CPUs (AMD) + 4 GPUs (C1060)**

ALL
4 GPUs
3 GPUs
2 GPUs
1 GPU
Speed (in Gflop/s)
Matrix order

Scheduling using QUARK



- Register tasks & dependencies in QUARK (similar to StarPU)
- Need a layer/mechanism to handle CPU-GPU communications
- Use MAGMA and LAPACK/ScaLAPACK

A QR for GPU + Multicore

- Parallel, dynamically scheduled panel factorizations (w/ QUARK) on multicore
- Parallel updates on multicore
- Parallel updates on GPU

Hybrid QR factorization **trace** for matrix of size 3360 x 3360

A QR for GPU + Multicore

Performance of MAGMA QR with 1 GPU and all Available Cores, Double Precision
Comparing Against MAGMA 1.0 and MKL
12 Cores (2 x 6-cores) 2.8 GHz X5660, 23 GB, 270 Gflop/s Peak [keeneland]
Tesla M2070, 1.1 GHz, 5.4 GB, 1.03 Tflop/s Peak
Single Node, Single GPU

Gflop/s
0
50
100
150
200
250
300

Matrix Size
0
5000
10000
15000
20000

1 GPU, All Cores, New Approach
1 GPU, 1 Socket, MAGMA 1.0
1 GPU, 1 Core, MAGMA 1.0
0 GPUs, 1 Socket, MKL

Current and future work

Sparse iterative solvers

Algorithm 1 GMRES for GPUs

```
1:  for i = 0, 1, ... do
2:      r = b − A x_i                              (magma_sspmv)
3:      β = h_{1,0} = ||r||_2                      (cublasSnrm2)
4:      check convergence and exit if done
5:      for k = 1, ..., m do
6:          v_k = r / h_{k,k−1}                    (magma_sscal)
7:          r = A v_k                              (magma_sspmv)
8:          for j=1,...,k do
9:              h_{j,k} = r^T v_j                  (cublasSdot)
10:             r = r − h_{j,k} v_j                (cublasSaxpy)
11:         end for
12:         h_{k+1,k} = ||r||_2                    (cublasSnrm2)
13:     end for
14:     Define V_k = [v_1, ..., v_k], H_k = {h_{i,j}}
15:     Find y_k that minimizes ||β e_1 − H_k y_k||_2
16:     x_{i+1} = x_i + V_k y_k                    (magma_sgemv)
17: end for
```

Algorithm 2 LOBPCG for GPUs

```
1: for i = 0, 1, ... do
2:     R = P(A X_i − Λ X_i)                        (magma_sspmv)
3:     check convergence and exit if done
4:     [X_i, Λ] = Rayleigh-Ritz on
                  span{X_i, X_{i−1}, R}            (hybrid)
5: end for
```

- The hybridization approach naturally works
  [e.g., Richardson iteration in mixed-precision
  iterative refinement solvers, Krylov space iterative solvers and eigen-solvers]
- Fast sparse matrix-vector product on Fermi
- Explore ideas to reduce communication
  [e.g., mixed precision, reduced storage for integers for the indexing, etc.]
- Need high bandwidth

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

DPLASMA (Work in progress)

- Provide a framework for distributed execution of a DAG
  - Taking in account the properties of the hardware and the network (cores and accelerators)
  - Minimizing the impact on the system (CPU and memory waste)
- Let the user describe the algorithms based on data dependencies between tasks
  - Language to be defined

DPLASMA

- Distribute the DAG analysis
  - The DAG is never completely unrolled
  - Each node only unrolls it's own portion of the DAG
- Minimize the data transfers
- Overlap communication and computations
- Many possible extensions on the 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