Five Important Concepts to Consider When Using High Performance Computers

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester
Moore’s Law Reinterpreted

- Number of cores per chip doubles every 2 years, while clock speed remains fixed or decreases
- Need to deal with systems with millions of concurrent threads
  - Future generation will have billions of threads!
- Number of threads of execution doubles every 2 years
Five Important Features to Consider When Computing at Scale

• Effective Use of Many-Core and Hybrid architectures
  ▪ Dynamic Data Driven Execution
  ▪ Block Data Layout

• Exploiting Mixed Precision in the Algorithms
  ▪ Single Precision is 2X faster than Double Precision
  ▪ With GP-GPUs 10x

• Self Adapting / Auto Tuning of Software
  ▪ Too hard to do by hand

• Fault Tolerant Algorithms
  ▪ With 1,000,000’s of cores things will fail

• Communication Avoiding Algorithms
  ▪ For dense computations from $O(n \log p)$ to $O(\log p)$ communications
  ▪ GMRES s-step compute ( $x$, $Ax$, $A^2x$, ... $A^s x$ )
Major Changes 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:
Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (70’s) (Vector operations)</td>
</tr>
<tr>
<td>LAPACK (80’s) (Blocking, cache friendly)</td>
</tr>
<tr>
<td>ScaLAPACK (90’s) (Distributed Memory)</td>
</tr>
<tr>
<td>PLASMA (00’s) (New Algorithms, many-core friendly)</td>
</tr>
</tbody>
</table>

**LINPACK (70’s)**
- Vector operations

**LAPACK (80’s)**
- Level-3 BLAS operations

**ScaLAPACK (90’s)**
- PBLAS Message Passing

**PLASMA (00’s)**
- DAG/scheduler
- Block data layout
- Some extra kernels
# A New Generation of Software:
Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
</table>
| **LINPACK (70’s)**  
(Vector operations) | ![Diagram](image) | Rely on  
- Level-1 BLAS operations |
| **LAPACK (80’s)**  
(Blocking, cache friendly) | ![Diagram](image) | Rely on  
- Level-3 BLAS operations |
# A New Generation of Software:

Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

<table>
<thead>
<tr>
<th></th>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (70’s) (Vector operations)</td>
<td>Rely on - Level-1 BLAS operations</td>
</tr>
<tr>
<td>LAPACK (80’s) (Blocking, cache friendly)</td>
<td>Rely on - Level-3 BLAS operations</td>
</tr>
<tr>
<td>ScaLAPACK (90’s) (Distributed Memory)</td>
<td>Rely on - PBLAS Mess Passing</td>
</tr>
</tbody>
</table>
## A New Generation of Software: Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (70’s) (Vector operations)</td>
</tr>
<tr>
<td>LAPACK (80’s) (Blocking, cache friendly)</td>
</tr>
<tr>
<td>ScALAPACK (90’s) (Distributed Memory)</td>
</tr>
<tr>
<td>PLASMA (00’s) New Algorithms (many-core friendly)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rely on</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Level-1 BLAS operations</td>
</tr>
<tr>
<td>- Level-3 BLAS operations</td>
</tr>
<tr>
<td>- PBLAS Mess Passing</td>
</tr>
<tr>
<td>- a DAG/scheduler</td>
</tr>
<tr>
<td>- block data layout</td>
</tr>
<tr>
<td>- some extra kernels</td>
</tr>
</tbody>
</table>

Those new algorithms

- have a very low granularity, they scale very well (multicore, petascale computing, … )
- removes a lots of dependencies among the tasks, (multicore, distributed computing)
- avoid latency (distributed computing, out-of-core)
- rely on fast kernels

Those new algorithms need new kernels and rely on efficient scheduling algorithms.
LAPACK LU - Intel64 - 16 cores

DGETRF - Intel64 Xeon quad-socket quad-core (16 cores) - th. peak 153.6 Gflop/s
• Fork-join, bulk synchronous processing
Coding for an Abstract Multicore

Parallel software for multicore processors should have two characteristics:

- **Fine granularity:**
  - High level of parallelism is needed
  - Cores will probably be associated with relatively small local memories. This requires splitting an operation into tasks that operate on small portions of data in order to reduce bus traffic and improve data locality.

- **Asynchronicity:**
  - As the degree of thread level parallelism grows and granularity of the operations becomes smaller, the presence of synchronization points in a parallel execution seriously affects the efficiency of an algorithm.
Steps in the LAPACK LU

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

LAPACK
BLAS
LU Timing Profile (16 core system)

Threads – no lookahead

Time for each component

Bulk Sync Phases

- DGETF2
- DLASWP(L)
- DLASWP(R)
- DTRSM
- DGEMM
Adaptive Lookahead - Dynamic

Event Driven Multithreading

Ideas not new.

Many papers use the DAG approach.

while(1)
    fetch_task();
    switch(task.type) {
        case PANEL:
            dgetf2();
            update_progress();
        
        case COLUMN:
            dlaswp();
            dtrsm();
            dgemm();
            update_progress();
        
        case END:
            for()
                dlaswp();
            return;
    }

Reorganizing algorithms to use this approach
- 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
Achieving Fine Granularity

Fine granularity may require novel data formats to overcome the limitations of BLAS on small chunks of data.

Column-Major
Achieving Fine Granularity

Fine granularity may require novel data formats to overcome the limitations of BLAS on small chunks of data.
• Break into smaller tasks and remove dependencies
Parallel Tasks in LU

Step 1: LU of block 1,1 (w/partial pivoting)

LAPACK Style (Panel operation then MM)
Parallel Tasks in LU

Step 1: LU of block 1,1 (w/partial pivoting)

Step 2: Use $U_{1,1}$ to zero $A_{1,2}$ (w/partial pivoting)
Parallel Tasks in LU

Step 1: LU of block $1,1$ (w/partial pivoting)

Step 2: Use $U_{1,1}$ to zero $A_{1,2}$ (w/partial pivoting)
Parallel Tasks in LU

Step 1: LU of block 1,1 (w/partial pivoting)

Step 2: Use $U_{1,1}$ to zero $A_{1,2}$ (w/partial pivoting)

Step 3: Use $U_{1,1}$ to zero $A_{1,3}$ (w/partial pivoting)
Parallel Tasks in LU

Step 1: LU of block 1,1 (w/partial pivoting)

Step 2: Use U_{1,1} to zero A_{1,2} (w/partial pivoting)

Step 3: Use U_{1,1} to zero A_{1,3} (w/partial pivoting)

...
LU - Intel64 - 16 cores

DGETRF - Intel64 Xeon quad-socket quad-core (16 cores)
theoretical peak 153.6 Gflop/s
Residual from PLASMA’s Tiled LU

\[
\frac{\| Ax - b \|_\infty}{(\| A \|_\infty \| x \|_\infty + \| b \|_\infty) n \epsilon}
\]

Random Matrices

\[
\kappa(A) \cdot 10^5 - 10^8
\]

NT (Number of Tiles)
**Tile QR (\&LU) Algorithms**

FOR $k = 0..TILES-1$

\[ A[k][k], T[k][k] \leftarrow DGRQRT(A[k][k]) \]

FOR $m = k+1..TILES-1$

\[ A[k][k], A[m][k], T[m][k] \leftarrow DTSQRT(A[k][k], A[m][k], T[m][k]) \]

FOR $n = k+1..TILES-1$

\[ A[k][n] \leftarrow DLARFB(A[k][k], T[k][k], A[k][n]) \]

FOR $m = k+1..TILES-1$

\[ A[k][n], A[m][n] \leftarrow DSSRFB(A[m][k], T[m][k], A[k][n], A[m][n]) \]

- input matrix stored and processed by square tiles
- complex DAG
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
- updating the trailing submatrix
- merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
  - updating the trailing submatrix
  - merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
- updating the trailing submatrix
- merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
- updating the trailing submatrix
- merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
- updating the trailing submatrix
- merge the domains
Communication Avoiding QR Factorization

- TS matrix
  - MT=6 and NT=3
  - split into 2 domains

- 3 overlapped steps
  - panel factorization
  - updating the trailing submatrix
  - merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
- updating the trailing submatrix
- merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
- updating the trailing submatrix
- merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
- updating the trailing submatrix
- merge the domains
Communication Avoiding QR Factorization

TS matrix
  - MT=6 and NT=3
  - split into 2 domains

3 overlapped steps
  - panel factorization
  - updating the trailing submatrix
  - merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
- updating the trailing submatrix
- merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
  - updating the trailing submatrix
  - merge the domains
Communication Avoiding QR Factorization

TS matrix
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps
- panel factorization
  - updating the trailing submatrix
  - merge the domains
Communication Avoiding QR Factorization

- TS matrix
  - MT=6 and NT=3
  - split into 2 domains

- 3 overlapped steps
  - panel factorization
  - updating the trailing submatrix
  - merge the domains
Communication Avoiding QR Factorization

- **TS matrix**
  - MT=6 and NT=3
  - split into 2 domains

- **3 overlapped steps**
  - panel factorization
  - updating the trailing submatrix
  - merge the domains
  - **Final R computed**
Communication Avoiding QR Factorization

Quad-socket, quad-core machine Intel Xeon EMT64 E7340 at 2.39 GHz. Theoretical peak is 153.2 Gflop/s with 16 cores.

Matrix size 51200 by 3200
PLASMA: Parallel Linear Algebra s/w for Multicore Architectures

• Objectives
  – high utilization of each core
  – scaling to large number of cores
  – shared or distributed memory

• Methodology
  – DAG scheduling
  – explicit parallelism
  – implicit communication
  – Fine granularity / block data layout

• Arbitrary DAG with dynamic scheduling
If We Had A Small Matrix Problem

- We would generate the DAG, find the critical path and execute it.
- DAG too large to generate ahead of time
  - Not explicitly generate
  - Dynamically generate the DAG as we go
- Machines will have large number of cores in a distributed fashion
  - Will have to engage in message passing
  - Distributed management
  - Locally have a run time system
The DAGs are Large

- Here is the DAG for a factorization on a 20 x 20 matrix

- For a large matrix say $O(10^6)$ the DAG is huge
- Many challenges for the software
Execution of the DAG by a Sliding Window

- Tile LU factorization 10x10 tiles
- 300 tasks total
- 100 task window
Execution of the DAG by a Sliding Window

Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window
Execution of the DAG by a Sliding Window

Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window
Execution of the DAG by a Sliding Window

Tile LU factorization 10x10 tiles
- 300 tasks total
- 100 task window
Execution of the DAG by a Sliding Window

Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window
Execution of the DAG by a Sliding Window

Tile LU factorization 10x10 tiles

- 300 tasks total
- 100 task window
PLASMA Dynamic Task Scheduler

- task – a unit of scheduling (quantum of work)
- slice – a unit of dependency resolution (quantum of data)
- Current version uses one core to manage the task pool
PLASMA on the Web

Google:
- plasma icl
- plasma dongarra
- plasma linear algebra
PLASMA Website

- Home
- Overview
- News
- Software
- Publications
- People
- Documentation
- User Forum
PLASMA Online Documentation

- README
- Installation Guide
- Users' Guide
- Routine Reference
- PLASMA TAU Guide
- Online source browser
PLASMA Online Documentation

**PLASMA_cgelqf**

**Purpose**

PLASMA_cgelqf - Computes the tile LQ factorization of a complex M-by-N matrix A: \( A = L \times Q \).

**Example: Arguments**

- \( M \) (int): The number of rows of the matrix \( A \). \( M > 0 \).
- \( N \) (int): The number of columns of the matrix \( A \). \( N > 0 \).
- \( A \) (PLASMA_Complex32_t *): Input. The M-by-N matrix \( A \).
- \( LDA \) (int): The leading dimension of the array \( A \). \( LDA \geq \max(1,M) \).
- \( T \) (PLASMA_Complex32_t *): Output. auxiliary factorization data, required by PLASMA_cgelqf to solve the system of equations.

**Example: Return Value:**

- 0: successful exit
- < 0: if -i, the i-th argument had an illegal value

**Example: C Bindings**

```c
int PLASMA_cgelqf(int M, int N, PLASMA_Complex32_t *A, int LDA, PLASMA_Complex32_t *T);
```

**Example: Fortran Bindings**

```fortran
PLASMA_CGELQF INTEGER M, INTEGER N, COMPLEX A, INTEGER LDA, INTEGER T, INTEGER INFO
```

**Online Browsing**

Dive into [PLASMA_cgelqf](https://icl.cs.utk.edu/projectsfiles/plasma/html/)

Online routine reference

Online source code browser
PLASMA Installation

-h or --help
--prefix
--cc=[CMD]
--fc=[CMD]
--ccflags=[FLAGS]
--fcflags=[FLAGS]
--ldflags_c=[flags]
--ldflags_fc=[flags]
--makecmd=[CMD]
--blaslib=[LIB]
--downblas
--nbcores
--notesting
--clean

Installation Guide Provides:
◆ list of command line options
◆ list of sample use cases

For an installation with gcc, gfortran and Reference BLAS

./setup.py --cc gcc --fc gfortran --downblas

For an installation with ifort, icc and MKL (em64t architecture)

./setup.py --cc icc --fc ifort --blaslib="-lmkl_em64t -lguide"

For an installation with xlc, xlf, essl and 8 cores

./setup.py --cc xlc --fc xlf --blaslib="-lessl" --nbcores=8

.....
PLASMA Installation

- Setup
- Checking BLAS
- Downloading source
- Compiling the library
- Building tests
- Running tests

```bash
python setup.py --cc icc --fc ifort --blaslib="-L/mnt/scr"
```

Setting up the framework

The C compiler is icc
The Fortran compiler is ifort
Checking if cc works... yes
Checking if the Fortran compiler works... yes
Setting Fortran mangling... -DADD_
Setting download command...
Checking availability of wget... available
Testing wget... working
Setting ranlib command... /usr/bin/ranlib
Detecting Fortran compiler... Intel
Detecting C compiler... Intel
Selected C compiler flags: -O2 -diag-disable vec
Selected Fortran compiler flags: -O2 -diag-disable vec
Selected loader flags (C main): -nomain
Selected loader flags (Fortran main): 
Checking loader... works
Setting environment variable... done.
Checking number of cores available on the machine: 16
Number of cores to be used for PLASMA testing: 2

BLAS installation/verification

Checking if provided BLAS works... yes

Plasma installer is starting now. Buckle up!

Downloading PLASMA from http://icl.cs.utk.edu/projectsfiles/plasma/pubs/plasma_2.0.0.tar.gz
Installing plasma_2.0.0 ...
Writing make.in... done.
Compiling PLASMA... Installation of PLASMA successful...
(log is in /home/kurzak/temp/plasma-installer/log/plasmaplot )
Compiling tests... done
Running PLASMA tests on 2 cores...(takes some time, feel free to grab a coffee!)
```
PLASMA Testing

Simple “sanity” testing
PLASMA Testing

(Insane) LAPACK testing

```
--- Detailed results are stored in testing_results.txt
---
--- Single ---
SGE routines passed the tests of the error exits (2190 tests run)
All tests for SGE routines passed the threshold (194 tests run)
SGE drivers passed the tests of the error exits (2868 tests run)
SPD routines passed the tests of the error exits (2046 tests run)
SPD drivers passed the tests of the error exits (26920 tests run)
All tests for SPD drivers passed the threshold (9006 tests run)
SRL routines passed the tests of the error exits (26920 tests run)
All tests for SRL routines passed the threshold (26920 tests run)
--- Double ---
DG SE routines passed the tests of the error exits (2190 tests run)
DG E drivers passed the tests of the error exits (194 tests run)
DGE routines passed the tests of the error exits (2868 tests run)
DPD routines passed the tests of the error exits (2046 tests run)
DPD drivers passed the tests of the error exits (26920 tests run)
SDL routines passed the tests of the error exits (9006 tests run)
All tests for SDL routines passed the threshold (26920 tests run)
DBL routines passed the tests of the error exits (26920 tests run)
All tests for DBL routines passed the threshold (26920 tests run)
Failure ===> DLQ: 1 out of 26920 tests failed to pass the threshold
--- Complex ---
CGE routines passed the tests of the error exits (2190 tests run)
CGE drivers passed the tests of the error exits (194 tests run)
CGP routines passed the tests of the error exits (2868 tests run)
Failure ===> CPD: 1 out of 2560 tests failed to pass the threshold
```
PLASMA Installation Wrap Up

done
(log is in /home/kurzak/temp/plasma-installer/log/plasmatestlog)
done. PLASMA is installed. Use it in moderation :-)

******************************************************************************
******************************************************************************

PLASMA installation completed.

Your BLAS library is:
-L/mnt/scratch/sw/intel/C-11.0.083/mkl/lib/em64t -lmkl_em64t -lguide

Your CBLAS libraries are:
/home/kurzak/temp/plasma-installer/lib/libblas.a

Your CORE LAPACK library is:
/home/kurzak/temp/plasma-installer/lib/libcorelapack.a

Your CORE BLAS library is:
/home/kurzak/temp/plasma-installer/lib/libcoreblas.a

Your PLASMA library is:
/home/kurzak/temp/plasma-installer/lib/libplasma.a

Log messages are in the
/home/kurzak/temp/plasma-installer/log
directory.

The Plasma testing programs are in:
/home/kurzak/temp/plasma-installer/build/plasma/testing

The
/home/kurzak/temp/plasma-installer/build
directory contains the source code of the libraries
that have been installed. It can be removed at this time.

******************************************************************************

kurzak:zoot ~/temp/plasma-installer> 2:59PM
## PLASMA in Numbers

<table>
<thead>
<tr>
<th>After installation:</th>
<th>Actual source to maintain:</th>
</tr>
</thead>
<tbody>
<tr>
<td>~105 MB</td>
<td>~1 MB</td>
</tr>
<tr>
<td>~1,800 files</td>
<td>~100 files</td>
</tr>
<tr>
<td>~150,000 lines</td>
<td>~15,000 lines</td>
</tr>
<tr>
<td>=160 user functions</td>
<td>=52 user functions</td>
</tr>
</tbody>
</table>

Lots of code dropped in from CBLAS, LAPACK, F2C. Precision generation.
Important Developments

- General code restructuring and cleanup
- Compliance with LAPACK error handling
- LAPACK testing
- Thread safety
- Generation of multiple precisions
- Compatibility (simple) interface and native (expert interface)
- Mixed precision algorithms
- Workspace allocation
- Windows port
- Dynamic scheduler (work in progress)
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 Larrabee in 2010
  ▪ 8, 16, 32, or 64 x86 cores
• AMD’s Fusion in 2011
  ▪ Multicore with embedded graphics ATI
• Nvidia’s plans?
Hardware to Software Trends

**The MAGMA Project** [Matrix Algebra on GPU and Multicore Architectures]
- create LA libraries for the next-generation of highly parallel, and heterogeneous processors
- to be similar to LAPACK in functionality, data storage, and interface
- to allow effortless port of LAPACK-relying software to take advantage of the new hardware
- to run on homogeneous x86-based multicores and take advantage of GPUs (if available)
Hybrid Computing

- Match algorithmic requirements to architectural strengths of the hybrid components
  - **Multicore**: small tasks/tiles
  - **Accelerator**: large data parallel tasks

  - e.g. split the computation into tasks; define critical path that “clears” the way for other large data parallel tasks; proper schedule the tasks execution
  - Design algorithms with well defined “search space” to facilitate auto-tuning
Task Splitting, Scheduling, and Data Storage

- **Task splitting**
  - Easy: the splitting itself
    - splitting BLAS
  - Difficult: task granularity
    - to be dynamic and heterogeneous
    - Ideally: scheduler to agglomerate small tasks into large tasks for GPUs

Currently:
- Multi-level blocking for the panels on the CPU
- Tiles are coarse level size (empirically tuned)
- affinity for GPUs and the sub-matrices that they correspondingly modify (to minimize communication)
NVIDIA GeForce GTX 280

- NVIDIA-Speak
  - 240 stream processors

- Generic speak
  - 30 processing cores
  - 8 SIMD functional units per core
  - 1 mul-add (2 flops) + 1 mul per functional unit (2 flops/clock)
  - Best case theoretically: 240 mul-adds + 240 muls per clock
    - 1.3 GHz clock
    - \( 30 \times 8 \times (2 + 1) + 1.33 = 933 \text{ Gflop/s} \)
  - Best case reality: 240 mul-adds per clock
    - Just able to do the mul-add so 2/3 or 624 Gflop/s

- All this is single precision
  - Double precision is 78 Gflop/s peak (Factor of 8)

- 141 GB/s bus
- 1 GB memory
Single Core and GTX-280

Single Precision

Double Precision

MAGMA v0.1 - Performance on GTX280

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Single Precision</th>
<th>Double Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>50</td>
<td>80</td>
</tr>
<tr>
<td>2,000</td>
<td>100</td>
<td>130</td>
</tr>
<tr>
<td>3,000</td>
<td>150</td>
<td>180</td>
</tr>
<tr>
<td>4,000</td>
<td>200</td>
<td>230</td>
</tr>
<tr>
<td>5,000</td>
<td>250</td>
<td>280</td>
</tr>
<tr>
<td>6,000</td>
<td>300</td>
<td>310</td>
</tr>
<tr>
<td>7,000</td>
<td>350</td>
<td>370</td>
</tr>
<tr>
<td>8,000</td>
<td>400</td>
<td>420</td>
</tr>
<tr>
<td>9,000</td>
<td>450</td>
<td>470</td>
</tr>
<tr>
<td>10,000</td>
<td>500</td>
<td>520</td>
</tr>
</tbody>
</table>

GPU: NVIDIA GeForce GTX 280
CPU: Intel Xeon dual socket quad-core @2.33 GHz
GPU BLAS: CUBLAS 2.2, s/d gemm peak: 375 / 75 GFlop/s
CPU BLAS: MKL 10.0, s/d gemm peak: 17.5 / 8.6 GFlop/s
**MAGMA version 0.1 performance**

**QR factorization in single precision arithmetic, CPU interface**

Performance of MAGMA vs MKL

MAGMA QR time breakdown

- **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](http://icl.cs.utk.edu/magma) ]
**MAGMA** version 0.2 performance

Hessenberg factorization in double precision arithmetic, CPU interface

Performance of MAGMA vs MKL

<table>
<thead>
<tr>
<th>Matrix size x 1000</th>
<th>GFlop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>35</td>
</tr>
</tbody>
</table>

**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, dgemm peak: 75 GFlop/s

**CPU BLAS:** MKL 10.0, dgemm peak: 65 GFlop/s

[ for more performance data, see [http://icl.cs.utk.edu/magma]](http://icl.cs.utk.edu/magma)
Available through MAGMA's homepage
http://icl.cs.utk.edu/magma/

Included are the 3 one-sided matrix factorizations

Iterative Refinement Algorithm (Mixed Precision)

Standard (LAPACK) data layout and accuracy

Two LAPACK-style interfaces

- CPU interface: both input and output are on the CPU
- GPU interface: both input and output are on the GPU

This release is intended for single GPU
**Cholesky factorization for multicore + multi-GPUs**

- **Matrix \( N \times N \) is split into \( M \times M \) Magnum-Tiles**
  - Tile reuse per GPU = \( M / \#\text{GPUs} \)
  - Optimal communications per GPU = \( N^2/2 \times (1 + 1 / \#\text{GPUs}) \)

- **Implementation**
  - Every GPU needs space for half the matrix (\( N^2/2 \) elements; can be reduced to \( N^2/4 \))
    - Reduced memory use is possible but will lead to more communications
  - One CPU core is associate with one GPU and schedules its work
    - When tiles are needed for a tile operation, only the tiles that are not locally available yet are pulled from the CPU, and the communication is overlapped with computation
    - When a tile \( T_{i,j} \) is updated/factored it is saved on GPU \( #j \) and written immediately to the CPU from where it is copied to all other GPUs (when needed for 1\(^{st} \) time)
  - At the end we have the final result on the CPU
Performance results on Tesla C1070

- Tile size = 576 for all matrix sizes $N < 18,000$
- Tile size = 960 for $N \geq 18,000$
- RAM limitations prevented larger runs [for further performance improvements]
How to Deal with Complexity?

• Many parameters in the code needs to be optimized.

• Software adaptivity is the key for applications to effectively use available resources whose complexity is exponentially increasing

• Goal:
  ▪ Automatically bridge the gap between the application and computers that are rapidly changing and getting more and more complex

• Non obvious interactions between HW/SW can effect outcome
Performance Optimization

- **Optimization cycle**

  - Code Development
  - Instrumentation
  - Measure
  - Analyze
  - Modify / Tune
  - Complete, correct and well-performing program

- **Goals**
  - Large investments in HPC systems
  - Solve larger problems
  - Solve problems faster
Automatic Performance Tuning

- Writing high performance software is hard
- Ideal: get high fraction of peak performance from one algorithm
- Reality: Best algorithm (and its implementation) can depend strongly on the problem, computer architecture, compiler, ...
  - Best choice can depend on knowing a lot of applied mathematics and computer science
  - Changes with each new hardware, compiler release
- Goal: Automation
  - Generate and search a space of algorithms
  - Past successes: PHiPAC, ATLAS, FFTW, Spiral
  - Many conferences, DOE projects, ...
Examples of Automatic Performance Tuning

- **Dense BLAS**
  - Sequential
  - ATLAS (UTK) & PHiPAC (UCB)
- **Fast Fourier Transform (FFT) & variations**
  - FFTW (MIT)
  - Sequential and Parallel
  - www.fftw.org
- **Digital Signal Processing**
  - SPIRAL: www.spiral.net (CMU)
- **MPI Collectives (UCB, UTK)**
- **More projects, conferences, government reports, ...**
• Optimizing software to exploit the features of a given system has historically been an exercise in hand customization.
  ▪ Time consuming and tedious
  ▪ Hard to predict performance from source code
  ▪ Must be redone for every architecture and compiler
    • Software technology *often* lags architecture
    • Best algorithm may depend on input, so some tuning may be needed at run-time.

• There is a need for quick/dynamic deployment of optimized routines.
Performance Tuning Methodology

Software Installation
(done once per system)

Input Parameters
System specifics
User options

Hardware Probe

Parameter study of code versions

Code Generation Performance database

Installation
Performance Tuning Methodology

**Software Installation** (done once per system)

- **Input Parameters**
  - System specifics
  - User options

- **Hardware Probe**

- **Parameter study of code versions**

- **Code Generation Performance database**

**Software Execution**

- **Input Parameters**
  - Size, dim., …

- **Select best algorithm**
  - Based on input data,
  - State of hardware
  - Cluster, etc

- **Execution**
  - Data placement
  - Calculate

- **Performance Monitoring**
  - Database update

**Run-time**
Software Generation Strategy - ATLAS BLAS

- Parameter study of the hw
- Generate multiple versions of code, w/difference values of key performance parameters
- Run and measure the performance for various versions
- Pick best and generate library
- Level 1 cache multiply optimizes for:
  - TLB access
  - L1 cache reuse
  - FP unit usage
  - Memory fetch
  - Register reuse
  - Loop overhead minimization
- Similar to FFTW and Spiral

ATLAS Project
(Automatically Tuned Linear Algebra Software)

- Automatic generation of computational kernels for RISC architectures.
- Code generator takes about 1-2 hours to run.
  - Done once for a new architecture.
  - Written in ANSI C
- Extension of BLAS to Sparse, Parallel and Mixed Precision Operations.
- Extension of ATLAS to higher level operations.
  - SMPs
  - Pentium
  - SGI/Vector
  - DOD DSP
500x500 Double Precision Matrix-Matrix Multiply Across Various Systems

![Bar chart showing performance comparison between Vendor and ATLAS versions of 500x500 double precision matrix-matrix multiply across various systems.](chart.png)
Benefits:

- New libraries speed transfer of recent algorithmic technology in linear algebra, hierarchical methods, and other areas to users.
- A new division of labor between compiler writers, library writers and perhaps architects will simply the problems of all of them.
- New architectures make old solutions much less attractive (i.e. relatively slower compared to peak machine speed) unless we develop algorithms that accommodate them.
Self Adapting for Message Passing

- **Communication libraries (Part of FT-MPI)**
  - Optimize for the specifics of one’s configuration.
  - A specific MPI collective communication algorithm implementation may not give best results on all platforms.
  - Choose collective communication parameters that give best results for the system when the system is assembled.

- **Algorithm layout and implementation**
  - Look at the different ways to express implementation
• Can ATLAS-like techniques be applied to arbitrary code?
• What do we mean by ATLAS-like techniques?
  ▪ Blocking
  ▪ Loop unrolling
  ▪ Data prefetch
  ▪ Functional unit scheduling
  ▪ etc.
• Referred to as *empirical optimization*
  ▪ Generate many variations
  ▪ Pick the best implementation by measuring the performance
Applying Self Adapting Software

- Numerical and Non-numerical applications
  - BLAS like ops / message passing collectives
- Static or Dynamic determine code to be used
  - Perform at make time / every time invoked
- Independent or dependent on data presented
  - Same on each data set / depends on properties of data
Auto-Tuning

- Best algorithm implementation can depend strongly on the problem, computer architecture, compiler, ...

- There are 2 main approaches
  - Model-driven optimization
    [Analytical models for various parameters; Heavily used in the compilers community; May not give optimal results ]
  - Empirical optimization
    [ Generate large number of code versions and runs them on a given platform to determine the best performing one; Effectiveness depends on the chosen parameters to optimize and the search heuristics used ]

- Natural approach is to combine them in a hybrid approach
  [1st model-driven to limit the search space for a 2nd empirical part ]
  [ Another aspect is adaptivity - to treat cases where tuning can not be restricted to optimizations at design, installation, or compile time ]
Pruning the Search Space

- Time serial core kernels (dgemm, dssrfb, dssssm).

![Graphs showing performance comparison]

- Intel 64 - dgemm
- Power 6 - dssrfb

- Pick up the 'best' NB/IB samples (pruning);
- Select one per matrix size and number of cores.
Performance of Single Precision on Conventional Processors

- Realized have the similar situation on our commodity processors.
  - That is, SP is 2X as fast as DP on many systems

- The Intel Pentium and AMD Opteron have SSE2
  - 2 flops/cycle DP
  - 4 flops/cycle SP

- IBM PowerPC has AltiVec
  - 8 flops/cycle SP
  - 4 flops/cycle DP
  - No DP on AltiVec

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>SGEMM/ DGEMM</th>
<th>Size</th>
<th>SGEMV/ DGEMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Opteron 246</td>
<td>3000</td>
<td>2.00</td>
<td>5000</td>
<td>1.70</td>
</tr>
<tr>
<td>UltraSparc-Ile</td>
<td>3000</td>
<td>1.64</td>
<td>5000</td>
<td>1.66</td>
</tr>
<tr>
<td>Intel PIII Coppermine</td>
<td>3000</td>
<td>2.03</td>
<td>5000</td>
<td>2.09</td>
</tr>
<tr>
<td>PowerPC 970</td>
<td>3000</td>
<td>2.04</td>
<td>5000</td>
<td>1.44</td>
</tr>
<tr>
<td>Intel Woodcrest</td>
<td>3000</td>
<td>1.81</td>
<td>5000</td>
<td>2.18</td>
</tr>
<tr>
<td>Intel XEON</td>
<td>3000</td>
<td>2.04</td>
<td>5000</td>
<td>1.82</td>
</tr>
<tr>
<td>Intel Centrino Duo</td>
<td>3000</td>
<td>2.71</td>
<td>5000</td>
<td>2.21</td>
</tr>
</tbody>
</table>

Single precision is faster because:
- Operations are faster
- Reduced data motion
- Larger blocks gives higher locality in cache
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.

$L U = \text{lu}(A)$ \hspace{2cm} $O(n^3)$
$x = L\backslash(U\backslash b)$ \hspace{2cm} $O(n^2)$
$r = b - Ax$ \hspace{2cm} $O(n^2)$

WHILE $\| r \| \text{ not small enough}$

$z = L\backslash(U\backslash r)$ \hspace{2cm} $O(n^2)$
$x = x + z$ \hspace{2cm} $O(n^1)$
$r = b - Ax$ \hspace{2cm} $O(n^2)$

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.

\[
\begin{align*}
L U &= \text{lu}(A) \\
x &= L\backslash(U\backslash b) \\
r &= b - Ax \\
\text{WHILE } ||r|| \text{ not small enough} \\
z &= L\backslash(U\backslash r) \\
x &= x + z \\
r &= b - Ax
\end{align*}
\]

- 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 for Mixed Precision Iterative Refinement for Dense $Ax = b$

- Single precision is faster than DP because:
  - Higher parallelism within vector units
    - 4 ops/cycle (usually) instead of 2 ops/cycle
  - Reduced data motion
    - 32 bit data instead of 64 bit data
  - Higher locality in cache
    - More data items in cache

<table>
<thead>
<tr>
<th>Architecture (BLAS)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium III Coppermine (Goto)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Intel Pentium III Katmai (Goto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sun UltraSPARC IIe (Sunperf)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Pentium IV Prescott (Goto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Pentium IV-M Northwood (Goto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD Opteron (Goto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cray X1 (libscl)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM Power PG5 (2.7 GHz)VecLib</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compaq Alpha EV6 (CXML)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM SP Power3 (ESSL)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGI Octane (ATLAS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results for Mixed Precision Iterative Refinement for Dense $Ax = b$

- Single precision is faster than DP because:
  - Higher parallelism within vector units
    - 4 ops/cycle (usually) instead of 2 ops/cycle
  - Reduced data motion
    - 32 bit data instead of 64 bit data
  - Higher locality in cache
    - More data items in cache

### Architecture (BLAS-MPI)

<table>
<thead>
<tr>
<th>Architecture (BLAS-MPI)</th>
<th># proc</th>
<th>n</th>
<th>DP Solve /SP Solve</th>
<th>DP Solve /Iter Ref</th>
<th># iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Opteron (Goto – OpenMPI MX)</td>
<td>32</td>
<td>22627</td>
<td>1.85</td>
<td>1.79</td>
<td>6</td>
</tr>
<tr>
<td>AMD Opteron (Goto – OpenMPI MX)</td>
<td>64</td>
<td>32000</td>
<td>1.90</td>
<td>1.83</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Architecture (BLAS)</th>
<th># proc</th>
<th>n</th>
<th>DP Solve /SP Solve</th>
<th>DP Solve /Iter Ref</th>
<th># iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Intel Pentium III Coppermine (Goto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Intel Pentium III Katmai (Goto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Sun UltraSPARC IIe (Sunperf)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Intel Pentium IV Prescott (Goto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 Intel Pentium IV-M Northwood (Goto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 AMD Opteron (Goto)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 Cray X1 (libsci)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 IBM Power PG5 (2.7 GHz) (VecLib)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 Compaq Alpha EV6 (CXML)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 IBM SP Power3 (ESDL)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 SGI Octane (ATLAS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sparse Direct Solver and Iterative Refinement

MUMPS package based on multifrontal approach which generates small dense matrix multiplies

Tim Davis’s Collection, n=100K - 3M
Cray XD-1 (OctigaBay Systems)

Experiments with Field Programmable Gate Array
Specify arithmetic

Six Xilinx Virtex-4 Field Programmable Gate Arrays (FPGAs) per chassis
## Mixed Precision Iterative Refinement

### FPGA Performance Test

**Junqing Sun et al**

### Characteristics of multiplier on an FPGA* (using DSP48)

<table>
<thead>
<tr>
<th>Data Formats</th>
<th>DSP48s</th>
<th>Frequency (MHz)</th>
<th>GFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>s52e11 (double)</td>
<td>16/96</td>
<td>237</td>
<td>1.42</td>
</tr>
<tr>
<td>s51e11</td>
<td>16/96</td>
<td>238</td>
<td>1.43</td>
</tr>
<tr>
<td>s50e11</td>
<td>9/96</td>
<td>245</td>
<td>2.61</td>
</tr>
<tr>
<td>s34e8</td>
<td>9/96</td>
<td>289</td>
<td>3.08</td>
</tr>
<tr>
<td>s33e8</td>
<td>4/96</td>
<td>292</td>
<td>7.01</td>
</tr>
<tr>
<td>s23e8 (single)</td>
<td>4/96</td>
<td>339</td>
<td>8.14</td>
</tr>
<tr>
<td>s17e8</td>
<td>4/96</td>
<td>370</td>
<td>8.88</td>
</tr>
<tr>
<td>s16e8</td>
<td>1/96</td>
<td>331</td>
<td>31.78</td>
</tr>
<tr>
<td>s16e7</td>
<td>1/96</td>
<td>352</td>
<td>33.79</td>
</tr>
<tr>
<td>s13e7</td>
<td>1/96</td>
<td>336</td>
<td>32.26</td>
</tr>
</tbody>
</table>

* XC4LX160-10
Mixed Precision Iterative Refinement
- Random Matrix Test - Junqing Sun et al

Refinement iterations for customized formats (sXXe11).
Random matrices

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Mantissa Bits</th>
<th>12</th>
<th>16</th>
<th>23</th>
<th>31</th>
<th>48</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td></td>
<td>8.9</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>11.1</td>
<td>5.1</td>
<td>2.1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>19.7</td>
<td>6.1</td>
<td>2.5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>28</td>
<td>6.3</td>
<td>2.6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td></td>
<td>9.3</td>
<td>3</td>
<td>1.3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
<td>13.3</td>
<td>2.1</td>
<td>1.42</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Mixed Precision Hybrid Direct Solver

- Profiled Time* on Cray-XD1 - Junqing Sun et al

LU w Partial Pivoting using variable precision on an FPGA

High Performance Mixed-Precision Linear Solver for FPGAs,
Junqing Sun, Gregory D. Peterson, Olaf Storaasli, IEEE TPDC, 2008
**Sparse Iterative Methods (PCG)**

- **Outer/Inner Iteration**
  
  Outer iterations using 64 bit floating point

  Compute \( r^{(0)} = b - Ax^{(0)} \) for some initial guess \( x^{(0)} \)

  for \( i = 1, 2, \ldots \)
  
  solve \( Mz^{(i-1)} = r^{(i-1)} \)
  
  \( \rho_{i-1} = r^{(i-1)T} z^{(i-1)} \)
  
  if \( i = 1 \)
  
  \( p^{(1)} = z^{(0)} \)
  
  else
  
  \( \beta_{i-1} = \rho_{i-1}/\rho_{i-2} \)
  
  \( p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)} \)
  
  endif
  
  \( q^{(i)} = Ap^{(i)} \)
  
  \( \alpha_{i} = \rho_{i-1}/p^{(i)T} q^{(i)} \)
  
  \( x^{(i)} = x^{(i-1)} + \alpha_{i}p^{(i)} \)
  
  \( r^{(i)} = r^{(i-1)} - \alpha_{i}q^{(i)} \)
  
  check convergence; continue if necessary

  end

- **Inner iteration**
  
  In 32 bit floating point

  Compute \( r^{(0)} = b - Ax^{(0)} \) for some initial guess \( x^{(0)} \)

  for \( i = 1, 2, \ldots \)
  
  solve \( Mz^{(i-1)} = r^{(i-1)} \)
  
  \( \rho_{i-1} = r^{(i-1)T} z^{(i-1)} \)
  
  if \( i = 1 \)
  
  \( p^{(1)} = z^{(0)} \)
  
  else
  
  \( \beta_{i-1} = \rho_{i-1}/\rho_{i-2} \)
  
  \( p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)} \)
  
  endif
  
  \( q^{(i)} = Ap^{(i)} \)
  
  \( \alpha_{i} = \rho_{i-1}/p^{(i)T} q^{(i)} \)
  
  \( x^{(i)} = x^{(i-1)} + \alpha_{i}p^{(i)} \)
  
  \( r^{(i)} = r^{(i-1)} - \alpha_{i}q^{(i)} \)
  
  check convergence; continue if necessary

  end

- **Outer iteration in 64 bit floating point and inner iteration in 32 bit floating point**
Mixed Precision Computations for Sparse Inner/Outer-type Iterative Solvers

**Speedups** for mixed precision
Inner SP/Outer DP (SP/DP) iter. methods vs DP/DP
(CG², GMRES², PCG², and PGMRES² with diagonal prec.)
*(Higher is better)*

**Iterations** for mixed precision
SP/DP iterative methods vs DP/DP
*(Lower is better)*

**Machine:**
Intel Woodcrest (3GHz, 1333MHz bus)

**Stopping criteria:**
Relative to $r_0$ residual reduction ($10^{-12}$)
Intriguing Potential

• Exploit lower precision as much as possible
  ▪ Payoff in performance
    • Faster floating point
    • Less data to move

• Automatically switch between SP and DP to match the desired accuracy
  ▪ Compute solution in SP and then a correction to the solution in DP

• Potential for GPU, FPGA, special purpose processors
  ▪ Use as little you can get away with and improve the accuracy

• Applies to sparse direct and iterative linear systems and Eigenvalue, optimization problems, where Newton’s method is used.

\[
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}
\]

Correction = - A\((b - Ax)\)