Automatic Development of Linear Algebra Libraries for the Tesla Series

Enrique S. Quintana-Ortí
Universidad Jaime I de Castellón (Spain)
### Dense Linear Algebra

<table>
<thead>
<tr>
<th>Major problems:</th>
<th>Source of large-scale cases:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Linear systems</td>
<td>• Aeronautics: BEM</td>
</tr>
</tbody>
</table>
| \[
Ax = b
\]                                              | • Computational chemistry                  |
| • Eigenvalues                                        | • Data mining                              |
| \[
Ax = \lambda x
\]                              |                                           |
| • Singular values                                    |                                           |
| \[
A = U\Sigma V^T
\]                              |                                           |
## Dense Linear Algebra

**Major problems:**

- Linear systems
  \[ Ax = b \]

- Eigenvalues
  \[ Ax = \lambda x \]

- Singular values
  \[ A = U \Sigma V^T \]

**Algorithms:**

- One-sided factorizations:
  - LU, Cholesky, QR

- Two-sided factorizations:
  - QR alg., Jacobi

- Two-sided factorizations:
  - SVD
Dense Linear Algebra Libraries

Catching up with the current high-performance architecture...

Vector instructions: BLAS 1 and 2

Cache memory: BLAS 3

Distributed memory: Message passing
Dense Linear Algebra Libraries

Catching up with the *current* high-performance architecture...
Dense Linear Algebra Libraries

Programmability is the key!

Application Programming Interfaces (APIs): Not that much of an evolution ;-(
- LAPACK and ScaLAPACK are written in F77 with C wrappers
- PLAPACK is C OO-like
- FLAME is more advanced...

Functionality:
- Libraries frequently updated with faster and/or more reliable algorithms developed by experts
Dense Linear Algebra Libraries

What if one had to design the *final* dense linear algebra library?

Compatible with unknown future...

<table>
<thead>
<tr>
<th>New languages</th>
<th>Library independent from language</th>
</tr>
</thead>
<tbody>
<tr>
<td>New functionality</td>
<td>Automatic development of new algorithms</td>
</tr>
<tr>
<td>New architectures</td>
<td>Library independent from architecture</td>
</tr>
</tbody>
</table>
Dense Linear Algebra Libraries

FLAME (Formal Linear Algebra Methods Environment)
http://www.cs.utexas.edu/users/flame

The University of Texas at Austin

Universidad Jaime I at Castellon (Spain)

Support from:
- NSF
- NEC Solutions, Inc.
- National Instruments

Support from:
- Spanish Office of Science
- NVIDIA (2008 Professor Partner Grant)
Outline

- New languages:
  - object-oriented approach
  - XML code
  - Storage and algorithm are independent

- New functionality:
  - automatic development of (dense) linear algebra algorithms

- New architectures
  - NVIDIA G80
  - NVIDIA Tesla series
New Languages

FLAME notation: $A = LL^T$

$$\begin{bmatrix} A_{21} & A_{22} \\ \alpha_{11} & 1 \end{bmatrix} = A_{22} A_{22}^{T} a_{21} a_{21}^{T}$$

Algorithm: $A := \text{CHOLL}_{UNB_{VAR3}}(A)$

Partition $A \rightarrow \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix}$

where $A_{TL}$ is $0 \times 0$

while $m(A_{TL}) < m(A)$ do

Repartition

$\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & a_{01} & A_{02} \\ a_{10}^T & a_{11} & a_{12}^T \\ A_{20} & a_{21} & A_{22} \end{pmatrix}$

where $\alpha_{11}$ is $1 \times 1$

$\alpha_{11} := \sqrt{\alpha_{11}}$

$a_{21} := a_{21} / \alpha_{11}$

$A_{22} := A_{22} - \text{TRIL} (a_{21} a_{21}^T)$

Continue with

$\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} & a_{01} & A_{02} \\ a_{10}^T & a_{11} & a_{12}^T \\ A_{20} & a_{21} & A_{22} \end{pmatrix}$

endwhile
New Languages

**FLAME notation:** \( A = LL^T \)

Object-oriented, independence of language/storage and algorithm
New Functionality

Automatic development from math. specification

\[ A = LL^T \]

Mechanical procedure
New Architectures: NVIDIA G80

Algorithms:

- BLAS:
  - MM, MV, TRSM

- One-sided factorizations:
  - LU, Cholesky, QR

- Two-sided factorizations:
  - QR alg., Jacobi, SVD

Some keys to high performance:

- CUBLAS
- Algorithms rich in matrix-matrix product
- Fast data transfer between RAM and GPU memory
- Reduce #data transfers
- Overlap communication and computation
Experimental Setup

- Two Intel QuadCore E5405 processors (8 cores) @ 2.0 GHz
- 8 Gbytes of DDR2 RAM
- Intel MKL 10.0.1

- NVIDIA Tesla S870 (4 NVIDIA G80 GPUs)
- 1.5 Gbytes of RAM per GPU (distributed-memory)
- CUBLAS 2.0

Two PCI-Express Gen2 interfaces (48 Gbits/sec.)
All experiments with real, single-precision
Performance measured in GFLOPS ($10^9$ flops/sec.)
Data and results in CPU RAM: transfer included in timings
**Matrix-Matrix Product: \( C = C + A \cdot B \)**

**BLAS (Fortran-77):**

```fortran
CALL SGEMM( 'N', 'N', m, n, k, 1.0, A, LDA, B, LDB, 1.0, C, LDC )
```

**CUBLAS (C):**

```c
cublasSgemm( 'N', 'N', m, n, k, 1.0, dA, LDA, dB, LDB, 1.0, dC, LDC );
```

Computation in GPU requires:
- Initialization of CUDA environment
- Allocation of data structures in GPU memory (handlers dA, dB, dC)
- Transfer of data (matrices A, B, C)
- Computation (cublasSgemm)
- Retrieve result (matrix C)
- Free data structures in GPU memory
- Termination of CUDA environment
Matrix-Matrix Product: \( C = C + A \cdot B \)

### CUBLAS (C):

```c
    cublasSgemm( 'N', 'N',
                m, n, k,
                1.0, dA, LDA,
                dB, LDB,
                1.0, dC, LDC );
```

### FLAME API to CUBLAS (C):

```c
    FLAG_Gemm( FLA_NO_TRANSPOSE,
               FLA_NO_TRANSPOSE,
               FLA_ONE, A,
               B,
               FLA_ONE, C );
```

**Computation with FLAME/GPU API:**
- FLAG_Gemm is a wrapper to cublasSgemm
- Similar wrappers allow creation and free of data structures in the GPU, data transfers, etc.
- A, B, C are FLAME objects that contain information on the data type, dimension, and handler (dA, dB, dC)
Matrix-Matrix Product: \( C = C + A \cdot B \)

- Timings of CUBLAS include data transfer (4 full matrices!)
- Observed peaks for 8 cores CPU/GPU are 110/160 GFLOPS
- Without data transfer CUBLAS delivers up to 200 GFLOPS
Matrix-Matrix Product: $C = C + A \cdot B$

- Impact of data transfer is important
  - Reduce by overlapping communication/computation (not possible on G80)
  - Store the matrices by blocks: contiguous access provides faster access to local data (in RAM and GPU memory) and also faster transfers

  ![Diagram](image)

  - Traditional (in C, row-wise)
  - Storage-by-blocks: 1 level

- MKL internally employs a similar repacking
Matrix-Matrix Product: $C = C + A \cdot B$
Triangular System Solve: \( A \mathbf{X} = \mathbf{B} \)

- Some kernels in CUBLAS can be further optimized
- Impact of data transfer is still important
Triangular System Solve: $A X = B$

- Observed peak performance for trsm is close to that of sgemm (160 GFLOPS)
Cholesky Factorization: \( A = LL^T \)

**FLAME code for CPU:**

```c
while ( FLA_Obj_length(ATL) < FLA_Obj_length(A) ) {
    ...
    /*************************************************************************
    FLA_Chol_unb_var3( A11 );
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, 
               FLA_TRANSPOSE, FLA_NONUNIT_DIAG, 
               FLA_ONE, A11, A21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, 
              FLA_MINUS_ONE, A21, 
              FLA_ONE, A22 );
    /*************************************************************************/
    ...
}
```
FLAME code for GPU:

```c
while ( FLA_Obj_length(ATL) < FLA_Obj_length(A) ) {
    ...
    /*-----------------------------------------------*/
    FLAG_Chol_unb_var3( A11 );
    FLAG_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, 
               FLA_TRANSPOSE, FLA_NONUNIT_DIAG, 
               FLA_ONE, A11, A21 );
    FLAG_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, 
               FLA_MINUS_ONE, A21, 
               FLA_ONE, A22 );
    /*-----------------------------------------------*/
    ...
}
```

Factorization of diagonal block on CPU!

Cholesky Factorization: $A = LL^T$
Cholesky Factorization: $A = LL^T$

- Observed peak performance for spotrf is close to that of sgemm (160 GFLOPS)
New Architectures: NVIDIA Tesla Series

How do we deal with the multiple G80 processors in the Tesla?

- Akin distributed-memory:
  - GPU memory is distributed
  - No coherence mechanism
  - All transfer through CPU RAM

- Akin SMP:
  - GPU RAM is like cache of SMP processors
  - CPU RAM is like main memory in SMP
New Architectures: NVIDIA Tesla Series

How do we deal with the multiple G80 processors in the Tesla?

• Possible solution:
  • Program as a cluster
  • Message-passing
  • Rewrite complete library: an effort similar to that of developing ScaLAPACK/PLAPACK
How do we deal with the multiple G80 processors in the Tesla?

- FLAME solution:
  - Programmability is the key!
  - Algorithm (code) is independent from the architecture
  - Runtime system dynamically extracts the parallelism and handles data transfers
New Architectures: NVIDIA Tesla Series

First stage: symbolic execution of code by runtime
• Task decomposition
• Data dependencies identification

```c
while ( FLA_Obj_length(ATL) < FLA_Obj_length(A) ) {
    ...
    /*************************************************************************/
    FLAG_Chol_unb_var3( A11 );
    FLAG_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE, FLA_NONUNIT_DIAG, FLA_ONE, A11, A21 );
    FLAG_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_MINUS_ONE, A21, FLA_ONE, A22 );
    /*************************************************************************/
    ...
}
```
New Architectures: NVIDIA Tesla Series

Second stage: actual execution of code by runtime
- Scheduling of tasks
- Mapping of tasks and data transfers
New Architectures: NVIDIA Tesla Series

Architecture-aware runtime

- workload balance:
  - 2-D workload distribution
  - Owner-computes rule

- Reduce communication: software coherence
  - write-back
  - write-invalidate

- Distributed Shared Memory (DSM) layer
Matrix-Matrix Product: \( C = C + A \cdot B \)

- For the largest problem size, speed-ups are 3.21/5.51 w.r.t. algorithm-by-blocks/CUBLAS on a single G80 processor
Cholesky Factorization: $A = LL^T$

- For the largest problem size, speed-up is 3.25 w.r.t. a single G80 processor
Concluding Remarks

• Programmability: algorithm, data storage, and architecture are independent. Let the runtime system deal with it!

• Similar techniques can also be applied to domains other than dense linear algebra

• However, DSM layer produces little overhead due to regularity of dense linear algebra codes
Concluding Remarks

• Performance of GPUs and multi-GPUs platforms can be improved by:
  • Tune all CUBLAS kernels
  • Employ storage-by-blocks
  • Provide hardware coherence
  • Implement direct communication among GPUs
Ongoing and Future Work

• Very large-scale problems
• Two-sided factorizations for eigenvalues and singular values
• Generic approach for numeric applications not necessarily dense linear algebra

Thanks for your attention!