A Framework for Efficient Execution of Matrix Computations

José Ramón Herrero

Departament d’Architectura de Computadors
Universitat Politècnica de Catalunya

Advisor: Prof. Juan J. Navarro

Doctoral Dissertation
July 7th, 2006
Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented Soft. Dev. And Tuning Env.

Conclusions and future work
Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented SofT. Dev. And Tuning Env.

Conclusions and future work
Motivation & General Goals

Motivation

- Matrix computations lie at the heart of many applications.

General Goal:

- Obtain efficient implementations of frequent matrix operations.
Specific Goals

Identify the key points for obtaining high performance.

Obtain efficient implementations . . . .

- of some frequent operations:
  - Sparse Cholesky factorization.
  - Dense Cholesky factorization.
  - Dense Matrix Multiplication.
  - Nearest Neighbor (NN) Classification.
- On different platforms.

Note:

- Focus on sequential code.
Overview of the main parts of this work.
Outline

Introduction

Compiler-optimized inner kernels
  Creation of a Small Matrix Library (SML)

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented SofT. Dev. And Tuning Env.

Conclusions and future work
Overview

In search for high performance:

- **Efficiency of inner kernel** is of paramount importance.

Usual approach:

- Ad-hoc codes written in assembler.

Our approach:

- Compiler-optimized inner kernel for operation on small matrices
Compiler-optimized inner kernels

Facts:

- Need for high performance inner kernels
- High cost in creation of such kernels by hand
- Compiler Optimization is a mature field

Approach:

- Smooth the way to the compiler
  - Collection of codes written in high level language
  - Fix as many parameters as possible at compilation time
  - Use compiler to generate optimized object code
  - Insert best code in library: Small Matrix Library (SML)

Use SML routines for general codes.
Outline

Introduction

Compiler-optimized inner kernels
Creation of a Small Matrix Library (SML)

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented Soft. Dev. And Tuning Env.

Conclusions and future work
SML: Idea

- Write several variants of code
  - Loop order
  - Loop unrolling factors

- Use the *best* compiler available
  - Try several compiler optimization flags
SML: Idea

- Fix parameters at compilation time
  - Leading dimensions
  - Loop limits

- Example: $C = C + A \cdot B^T$

```fortran
subroutine mxmt(A,B,C, lda,ldb,ldc, ui, uj, uk)
  integer A(lda,*), ... do I=1, ui ...
  ... 

subroutine mxmt_fix(A,B,C)
  integer A(8,*)
  do I=1, 8 ...
  ... 
```
Performance of $C = C - A \times B^T$
Performance of $C = C - A \times B^T$
SML: Poly-Algorithmic Approach

<table>
<thead>
<tr>
<th>Code of Algorithm</th>
<th>form</th>
<th>Matrix sizes</th>
<th>Alpha 21164</th>
<th>R10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>jik_Creg</td>
<td>4_4_4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>jik</td>
<td>4_4_8</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>kji_Breg</td>
<td>4_4_16</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>i(jk4i_BCreg)</td>
<td>4_4_32</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>i(jk4i_Breg)</td>
<td>8_8_8</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>i(jk4i)</td>
<td>8_8_16</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>ijk</td>
<td>8_8_32</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>kji</td>
<td>16_16_16</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>jki</td>
<td>16_16_32</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>jik4k_Creg</td>
<td>32_32_32</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>jik8k_Creg</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SML: Large search space

- Many combinations
  - Leading dimensions
  - Loop limits
  - Loop orders
  - Loop unrolling factors
  - Compiler flags
  - Target machines

- We need to automate the tests
  - Use a benchmarking tool
SML: Benchmarking Tool

- foreach parameter combination
  - compile
  - execute
  - store results (Mflops)

- select best combination

- add object to library

Data Base

mxmt8x8: kji,u4, -O3, -swp=on

libsml.a

POSTDATE

Conclusions and future work
Inner kernel for operations on dense matrices

\[ C = C - A \times B^T \] kernel

**Graphs:**
- **Alpha 21264A (ev67) @ 731 MHz**
- **Itanium2 @ 1,3 GHz**
Generalization

\[ C = C + \alpha \text{op}(A) \times \text{op}(B) \]

- \( \alpha \) in \{-1, 1\}
- \( \text{op}(A) \) is \( A \) or \( A^T \).

**Table:** Peak Mflops of inner kernel on a Pentium 4 Xeon Northwood.

<table>
<thead>
<tr>
<th></th>
<th>( A \times B^T )</th>
<th>( A^T \times B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>No align</td>
<td>3334</td>
<td>3220</td>
</tr>
<tr>
<td>Align</td>
<td>3457</td>
<td>3810</td>
</tr>
</tbody>
</table>
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Creation of a Small Matrix Library (SML)

Sparse Hypermatrix Cholesky

Operation on dense matrices

Application to other fields: NN Classification

POSTDATE

Conclusions and future work

SML: current interface

- mtxms_124_124_124_124_124_124_(A,B,C)
- mtxms_96_96_96_96_96_96_(A,B,C)
- ...
- mtxmts_4_4_4_4_4_4_32_(A,B,C)
- mtxmts_4_4_4_4_4_4_(A,B,C)
- mtxmts_92_92_92_92_92_92_(A,B,C)
- ...
- mtxmts_i j kw_4_4_4_4_(A,B,C,i,j,k)
- mtxmts_i j kw_92_92_92_92_(A,B,C,i,j,k)
- mtxmts_i j w_4_4_4_32_(A,B,C,i,j)
- mtxmts_i j w_4_4_4_4_(A,B,C,i,j)
- mtxmts_i j w_92_92_92_92_92_(A,B,C,i,j)
- mtxmts_k w_4_4_4_4_4_(A,B,C,k)
- mtxmts_k w 92 92 92 92 92 (A,B,C,k)
Creation of efficient inner kernels: Conclusions

Compilers can perform efficient optimizations on regular codes. We can facilitate this by:

- Providing matrix leading dimensions and loop trip counts at compilation time;
- Trying several variants of code: different loop orders, unroll factors . . .

The resulting code can be more efficient if:

- Matrices are aligned;
- All matrices are accessed with stride one;
- Store operations are removed from the inner kernel.

$C = C + \alpha A^T \times B$ is appealing:

- access to all three matrices with stride one;
- stores to matrix $C$ can be hoisted from the inner loop
Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

SML kernels which operate on small data submatrices
Bit Vectors associated to data submatrices
Windows within data submatrices
Amalgamation
Results

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented Soft. Dev. And Tuning Env.
Hypermatrix (HM) Structure

Matrix

HyperMatrix

A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky

SML routines
Bit Vectors
Dense Windows
Amalgamation
Results

Operation on dense matrices

Application to other fields: NN Classification

Conclusions and future work
Hypermatrix (HM) Structure

Can store 0’s within data submatrices

- Storage
- Computation

Trade-off in data submatrix size

- BLAS3 efficiency
- (Useless) operation on 0’s
Reducing Overhead & Increasing Performance

- Efficient kernels which operate on small data submatrices
- Bit Vectors associated to data submatrices
- Windows within data submatrices
- Amalgamation
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

SML kernels which operate on small data submatrices
Bit Vectors associated to data submatrices
Windows within data submatrices
Amalgamation

Results

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented SofT. Dev. And Tuning Env.
Effective Mflops = \frac{\text{\#flops(excluding operations on zeros)} \cdot 10^{-6}}{\text{Time (including operations on zeros)}}
Reducing Overhead & Increasing Performance

Operation on small data submatrices . . .
still has overhead

Goal: reduce the overhead further

- Bit Vectors associated to data submatrices
- Windows within data submatrices
Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization
  SML kernels which operate on small data submatrices
  Bit Vectors associated to data submatrices
  Windows within data submatrices
  Amalgamation

Results

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented Soft. Dev. And Tuning Env.
Bit Vectors: Goal

Reduce unnecessary computation

- Avoid matrix multiplication when two submatrices produce no update upon a third one
Bit Vectors: Definition

One bit associated to a column in a data submatrix

- Value = 0 ⇔ column is full of 0’s
- Value = 1 ⇔ ∃ ≥ 1 NZ in column
Bit Vectors: Usage (I)

$BV_A \& BV_B \neq 0$: Operation must be performed
Bit Vectors: Usage (II)

Let's denote the matrices as follows:

- $A$: 
  - $i_1$: $x$
  - $i_2$: $x$

- $B$: 
  - $k_1, k_2, k_3, k_4, k_5$
  - $j_1, j_2, j_3$: $x$

- $C$: 
  - $x$

The Bit Vectors are:

- $BVA$: 
  - $00100110$

- $BV_B$: 
  - $01010000$

- $BV_{A\&B}$: 
  - $00000000$

Thus, $BV_{A\&B} = 0$: Operation can be skipped.
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

SML kernels which operate on small data submatrices
Bit Vectors associated to data submatrices
Windows within data submatrices
Amalgamation
Results

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented Soft. Dev. And Tuning Env.
Windows within data submatrices: Goal

Store and use only a part of a data submatrix
  ▶ Reduce unnecessary computation
  ▶ Reduce storage
Windows: Definition

Data Submatrix

Window: subset of data submatrix
Windows: Usage (I)

Operation can be reduced
Windows: Usage (II)

Operation can be skipped
Windows: Usage (III)

Unnecessary operation performed (Could be avoided with BVs)
Results: Context information

- MIPS R10000 @ 250 MHz (500 Mflops peak)
- Sequential code
- Large problems solved In-Core
- Ordered using METIS
- Post-order of Elimination Tree
- Linear Programming problems
  - NetLib
  - Multicommodity Network Flow generators
- Applications of Finite Element Method
  - NetLib
  - PERMAS
  - PARASOL
Performance: Block Size vs BVs vs Windows

HM performance

- **HM_8x8**
- **HM_8x8+BV**
- **HM_4x32**
- **HM_4x32+BV**
- **HM_4x32+win**
- **HM_4x32+win+BV**

Effective Mflops

- QAP12
- QAP15
- TRIPART2
- TRIPART4
- pds30
- pds60
- pds90

A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky

SML routines

Bit Vectors

Dense Windows

Amalgamation

Results

Operation on dense matrices

Application to other fields: NN Classification

POSTDATE

Conclusions and future work
Increase in number of floating point operations in sparse HM Cholesky w.r.t. the minimum: windows reduce the number of operations on zeros.
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky

SML routines

Bit Vectors

Dense Windows

Amalgamation

Results

Operation on dense matrices

Application to other fields: NN Classification

POSTDATE

Conclusions and future work

HM vs SN (Ng-Peyton): QAP matrix family

Effective Mflops

QAP8 QAP12 QAP15

HM_4x32+win
SN_cache2M_unr8
SN_cache1M_unr8
SN_cache512K_unr8
SN_cache32K_unr8
SN_cache32K_unr4
HM vs SN: TRIPART matrix family
HM vs SN: PDS matrix family

Effective Mflops

<table>
<thead>
<tr>
<th>pd1</th>
<th>pd10</th>
<th>pd20</th>
<th>pd30</th>
<th>pd40</th>
<th>pd50</th>
<th>pd60</th>
<th>pd70</th>
<th>pd80</th>
<th>pd90</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>104</td>
<td>154</td>
<td>204</td>
<td>254</td>
<td>304</td>
<td>354</td>
<td>404</td>
<td>454</td>
<td>504</td>
</tr>
</tbody>
</table>
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky

SML routines

Bit Vectors

Dense Windows

Amalgamation

Results

Operation on dense matrices

Application to other fields: NN Classification

POSTDATE

Conclusions and future work

HM vs SN performance: summary

SN vs HM

Effective Mflops

GRIDGEN1, QAP8, QAP12, QAP15, RMFGEN1, TRIPART1, TRIPART2, TRIPART3, TRIPART4, pds1, pds10, pds20, pds30, pds40, pds50, pds60, pds70, pds80, pds90

SN

HM
Matrix multiplication: efficiency of codes

Our sparse HM Cholesky uses 4 routines:

- SML Interface

Less efficient

Most efficient
### HM flops per $A \times B^T$ subroutine type

<table>
<thead>
<tr>
<th>Subroutine Type</th>
<th>WIN_2D</th>
<th>WIN_1DR</th>
<th>WIN_1DC</th>
<th>FULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRIDGEN1</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>QAP8</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>QAP12</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>QAP15</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>RMGEN1</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>TRIPART1</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>TRIPART2</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>TRIPART3</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>TRIPART4</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds1</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds10</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds20</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds30</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds40</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds50</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds60</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds70</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds80</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
<tr>
<td>pds90</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization
  SML kernels which operate on small data submatrices
  Bit Vectors associated to data submatrices
  Windows within data submatrices
  Amalgamation

Results

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented SofT. Dev. And Tuning Env.
Supernode Amalgamation

Definition (Amalgamation)

The mixing or blending of different elements, races, societies, etc.; also, the result of such combination or blending.
Supernode Amalgamation

Definition (Amalgamation)
The mixing or blending of different elements, races, societies, etc.; also, the result of such combination or blending.

- Hypermatrix oriented SN amalgamation
- Intra-Block amalgamation
Intra-Block Amalgamation: Original window

Data Submatrix

left column
	right column

top row

bottom row

window

X

X

X

X

50
Intra-Block Amalgamation: column-wise

Data Submatrix

left column

right column

top row

bottom row

X X X

window

[Diagram of data submatrix with columns and rows highlighted]

Operation on dense matrices

Application to other fields: NN Classification

Conclusions and future work
Intra-Block Amalgamation: row-wise

Data Submatrix

left column

right column

top row

bottom row

window

A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky

SML routines

Bit Vectors

Dense Windows

Amalgamation

Results

Operation on dense matrices

Application to other fields: NN Classification

POSTDATE

Conclusions and future work
Intra-Block Amalgamation: row and column-wise

Data Submatrix

left column

top row

bottom row

right column

window

X

X

X

X
Results: QAP8
Intra-Block Amalgamation
Results: QAP12
Intra-Block Amalgamation

![Graph showing Effective Mflops vs Columns amalgamated for different values of amr (0, 1, 2, 3).]
Results: TRIPART1

Intra-Block Amalgamation
Results: TRIPART2
Intra-Block Amalgamation

![Graph showing effective MFlops vs. columns amalgamated]

- amr=0
- amr=1
- amr=2
- amr=3

Effective MFlops

Columns amalgamated
Results: pds10
Intra-Block Amalgamation

![Graph showing Effective Mflops vs Columns amalgamated]

- amr=0
- amr=1
- amr=2
- amr=3

Conclusions and future work
Results: pds20
Intra-Block Amalgamation
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky

SML routines

Bit Vectors

Dense Windows

Amalgamation

Results

Operation on dense matrices

Application to other fields: NN Classification

POSTDATE

Conclusions and future work

Results: Original (without amalgamation) vs Intra-block amalgamation

Effective Mflops

GRIDGEN1  QAP8  QAP12  QAP15  QAP16

TRIPART1  TRIPART2  TRIPART3  TRIPART4

pds1  pds10  pds20  pds30  pds40  pds50  pds60  pds70  pds80  pds90

amr0.amc0  amr1.amc5
Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization
  SML kernels which operate on small data submatrices
  Bit Vectors associated to data submatrices
  Windows within data submatrices
  Amalgamation

Results

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented SofT. Dev. And Tuning Env.
Performance of several sparse Cholesky codes: IPM

![Graph showing performance of several sparse Cholesky codes: IPM](image-url)

- SN-LL (Ng-Peyton)
- Blik (Splash-2)
- SN-LL (Taucs)
- SN-MF (Taucs)
- HM

---

A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky

SML routines

Bit Vectors

Dense Windows

Amalgamation

Results

Operation on dense matrices

Application to other fields: NN Classification

POSTDATE

Conclusions and future work
Performance of several sparse Cholesky codes:
FEM

- SN-LL (Taucs)
- SN-MF (Taucs)
- HM

Effective Mflops

- bear
- rail
- methan
- nasasrb
- cfd1
- cdf2
- inline_1
Sparse HM Cholesky: flops per $A \times B^T$ subroutine type

IPM & FEM

% flops performed by each mex subroutine

IPM matrices

FEM matrices

A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky

SML routines

Bit Vectors

Dense Windows

Amalgamation

Results

Operation on dense matrices

Application to other fields: NN Classification

POSTDATE

Conclusions and future work
Overhead in number of operations in sparse HM Cholesky (4x32 + windows)
Sparse HM Cholesky vs WSSMP

- HM Cholesky ported to another platform: Itanium2
- METIS options for IPM matrices:
  - 1, 3, 1, 1, 0, 3, 60, and 5
- Results include use of HM oriented SN amalgamation
Sparse HM Cholesky vs WSSMP

![Graph showing performance comparison between WSSMP LP and HM METIS LP for different datasets and problem sizes.](image-url)
Sparse HM Cholesky vs WSSMP

Performance relative to best

GRIDGEN1 QAP8 QAP12 QAP15 RMFGEN1 TRIPART1 TRIPART2 TRIPART3 TRIPART4 pds1 pds10 pds20 pds30 pds40 pds50 pds60 pds70 pds80 pds90

WSSMP WSSMP LP WSSMP METIS LP HM METIS LP
Conclusions and future work

- The most effective overhead reduction techniques have been:
  - Use of SML routines working on rectangles
  - Use of dense windows within data submatrices
  - Use of Intra-Block Amalgamation

- Future work
  - Work on $U$ instead of $L$
  - Store data submatrices as supernodes
Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts
  Operation on dense matrices: Hypermatrix Storage
  Operation on dense matrices: Square Block Storage

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented SofT. Dev. And Tuning Env.

Conclusions and future work
A bottom-up approach

In search for high performance:

- **Efficiency of inner kernel** is of paramount importance.

Different optimal block sizes for different processors
A bottom-up approach

First

- Produce inner kernel and determine best block size
A bottom-up approach

Then

- Create structure based on best block size

First

- Produce inner kernel and determine best block size
Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts

Operation on dense matrices: Hypermatrix Storage

Operation on dense matrices: Square Block Storage

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented SofT. Dev. And Tuning Env.

Conclusions and future work
Results: MIPS R10000
Dense Cholesky and Matrix Multiplication

A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction
Compiler-optimized inner kernels
Sparse Hypermatrix Cholesky
Operation on dense matrices
Application to other fields: NN Classification

POSTDATE

Results: MIPS R10000
Dense Cholesky and Matrix Multiplication

- HM Storage
- SB Storage

Conclusions and future work
Orthogonal Blocks

Constructed so that the directions of the blocks of adjacent levels are different.
Results: Size 4507 on Itanium2
HM matrix multiplication using several loop orders
Results: Itanium2
Dense Cholesky and Matrix Multiplication

- Compiler-optimized inner kernels
- Sparse Hypermatrix Cholesky
- Operation on dense matrices
  - HM Storage
  - SB Storage
- Application to other fields: NN Classification

Conclusions and future work
Results: Alpha 21264A
Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts
  Operation on dense matrices: Hypermatrix Storage
  Operation on dense matrices: Square Block Storage

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented SofT. Dev. And Tuning Env.

Conclusions and future work
Simple Square Block (SB) storage: matrices aligned and stored by submatrices.
Simple SB storage: \( C = C - A^t \times B \)

Results on Power4
Simple SB storage: $C = C - A^t \times B$

Results on Pentium4

![Graph with Mflops on the y-axis and a range from 0 to 4800, showing different performance lines for GOTO, ATLAS, nc ATLAS, SB+SML, and HM+SML.]
Simple SB storage: $C = C - A^t \times B$

Results on Itanium2

![Graph showing performance metrics for different storage methods.](image-url)
Dense codes: Conclusions

- Efficiency of inner kernel is of paramount importance.
  - Different optimal block sizes for different processors
  - Fundamental aspects to achieve high performance:
    - data accessed with stride one;
    - data properly aligned;
    - store operations removed from the innermost loop.

- Iterative+SB outperforms Recursive+HM
- Iterative+SB+SML provides competitive and stable performance
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented SofT. Dev. And Tuning Env.

Conclusions and future work
Applications in other fields can benefit from the use of techniques widely used in linear algebra codes:

- use floating-point operations instead of integer arithmetic;
- apply tiling;
- loop unrolling;
- software pipelining.

Conclusions:

- A simple code can sometimes outperform complex codes which can be more difficult to implement efficiently.
- It can pay off to do more operations, as long as they are performed faster.
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented Soft. Dev. And Tuning Env.

Conclusions and future work
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky

Operation on dense matrices

Application to other fields: NN Classification

POSTDATE

Conclusions and future work

POSTDATE

Performance Oriented SofTware Development And Tuning Environment

- **maker**: A front-end to *make* to ease the build process.
  - Handle compilation on different platforms.
  - Avoid problems in a Networked File System

- **ACME**: A framework to ensure **accurate** measurements.
  - Avoid problems due to lack of precision of timers.

- **BMT**: A **benchmarking** tool.
  A framework to handle the tuning process automatically.
  - Handle compile time and run time parameters automatically.
  - Manage DB
  - Create library from optimum combination of parameters.
A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Outline

Introduction

Compiler-optimized inner kernels

Sparse Hypermatrix Cholesky Factorization

Operation on dense matrices: Nonlinear array layouts

Application to other fields: Nearest Neighbor Classification

POSTDATE: Performance Oriented Soft. Dev. And Tuning Env.

Conclusions and future work
Conclusions

- Current compilers can generate very efficient codes when working on simple and regular codes.

- There is a trade-off between the speed of an algorithm and:
  - computation of non productive operations.
  - more regular codes

- Fundamental aspects to achieve high performance:
  - data accessed with stride one;
  - data properly aligned;
  - store operations removed from the innermost loop.
Contributions

- Generation of efficient compiler-optimized inner kernels for different:
  - Matrix sizes
  - Platforms

- Highly competitive codes for some important algorithms working on matrices:
  - sparse Cholesky factorization using a hypermatrix data structure.
  - dense Cholesky factorization and matrix multiplication using a square block data layout.
  - Nearest Neighbor classification.

- Environment for creation of high performance codes: aids in the process of building the code, obtaining accurate measurements, and benchmarking different variants of code.
Future Work

- Improve creation of SML routines using more realistic access patterns;
- Implement HM Cholesky on matrix $U$ to increase number of accesses with stride one;
- Allow for storage of data submatrices as supernodes;
- Experiment with an Incomplete Cholesky factorization;
- ...
A Framework for Efficient Execution of Matrix Computations

José Ramón Herrero

Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya

Advisor: Prof. Juan J. Navarro

Doctoral Dissertation
July 7th, 2006
Outline

Sparse Hypermatrix Cholesky Factorization
### IPM: Matrix Characteristics

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Dimension</th>
<th>NZs</th>
<th>NZs in L&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Density</th>
<th>Flops to factor&lt;sup&gt;b&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRIDGEN1</td>
<td>330430</td>
<td>3162757</td>
<td>130586943</td>
<td>0.002</td>
<td>278891</td>
</tr>
<tr>
<td>QAP8</td>
<td>912</td>
<td>14864</td>
<td>193228</td>
<td>0.463</td>
<td>63</td>
</tr>
<tr>
<td>QAP12</td>
<td>3192</td>
<td>77784</td>
<td>2091706</td>
<td>0.410</td>
<td>2228</td>
</tr>
<tr>
<td>QAP15</td>
<td>6330</td>
<td>192405</td>
<td>8755465</td>
<td>0.436</td>
<td>20454</td>
</tr>
<tr>
<td>RMFGEN1</td>
<td>28077</td>
<td>151557</td>
<td>6469394</td>
<td>0.016</td>
<td>6323</td>
</tr>
<tr>
<td>TRIPART1</td>
<td>4238</td>
<td>80846</td>
<td>1147857</td>
<td>0.127</td>
<td>511</td>
</tr>
<tr>
<td>TRIPART2</td>
<td>19781</td>
<td>400229</td>
<td>5917820</td>
<td>0.030</td>
<td>2926</td>
</tr>
<tr>
<td>TRIPART3</td>
<td>38881</td>
<td>973881</td>
<td>17806642</td>
<td>0.023</td>
<td>14058</td>
</tr>
<tr>
<td>TRIPART4</td>
<td>56869</td>
<td>2407504</td>
<td>76805463</td>
<td>0.047</td>
<td>187168</td>
</tr>
<tr>
<td>pds1</td>
<td>1561</td>
<td>12165</td>
<td>37339</td>
<td>0.030</td>
<td>1</td>
</tr>
<tr>
<td>pds10</td>
<td>18612</td>
<td>148038</td>
<td>3384640</td>
<td>0.019</td>
<td>2519</td>
</tr>
<tr>
<td>pds20</td>
<td>38726</td>
<td>319041</td>
<td>10739539</td>
<td>0.014</td>
<td>13128</td>
</tr>
<tr>
<td>pds30</td>
<td>57193</td>
<td>463732</td>
<td>18216426</td>
<td>0.011</td>
<td>26262</td>
</tr>
<tr>
<td>pds40</td>
<td>76771</td>
<td>629851</td>
<td>27672127</td>
<td>0.009</td>
<td>43807</td>
</tr>
<tr>
<td>pds50</td>
<td>95936</td>
<td>791087</td>
<td>36321636</td>
<td>0.007</td>
<td>61180</td>
</tr>
<tr>
<td>pds60</td>
<td>115312</td>
<td>956906</td>
<td>46377926</td>
<td>0.006</td>
<td>81447</td>
</tr>
<tr>
<td>pds70</td>
<td>133326</td>
<td>1100254</td>
<td>54795729</td>
<td>0.006</td>
<td>100023</td>
</tr>
<tr>
<td>pds80</td>
<td>149558</td>
<td>1216223</td>
<td>64148298</td>
<td>0.005</td>
<td>125002</td>
</tr>
<tr>
<td>pds90</td>
<td>164944</td>
<td>1320298</td>
<td>70140993</td>
<td>0.005</td>
<td>138765</td>
</tr>
</tbody>
</table>

<sup>a</sup>Number of non-zeros in factor L (matrix ordered using METIS).

<sup>b</sup>Number of floating point operations (in Millions) necessary to obtain L from the original matrix (ordered with METIS).
### FEM: Matrix Characteristics

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Dimension</th>
<th>NZs</th>
<th>NZs in L(^a)</th>
<th>Density</th>
<th>Flops to factor(^b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bear</td>
<td>25906</td>
<td>412447</td>
<td>3278225</td>
<td>0.009</td>
<td>847</td>
</tr>
<tr>
<td>rail</td>
<td>11783</td>
<td>799545</td>
<td>3768886</td>
<td>0.054</td>
<td>1594</td>
</tr>
<tr>
<td>methan</td>
<td>48162</td>
<td>1234332</td>
<td>16631801</td>
<td>0.014</td>
<td>10493</td>
</tr>
<tr>
<td>nasasrb</td>
<td>54870</td>
<td>1366097</td>
<td>10489476</td>
<td>0.007</td>
<td>3496</td>
</tr>
<tr>
<td>cfd1</td>
<td>70656</td>
<td>949510</td>
<td>20910296</td>
<td>0.008</td>
<td>13523</td>
</tr>
<tr>
<td>cfd2</td>
<td>123440</td>
<td>1605669</td>
<td>37696869</td>
<td>0.005</td>
<td>31218</td>
</tr>
<tr>
<td>inline_1</td>
<td>503712</td>
<td>18660027</td>
<td>174608135</td>
<td>0.001</td>
<td>150974</td>
</tr>
</tbody>
</table>

\(^a\)Number of non-zeros in factor L (matrix ordered using METIS).

\(^b\)Number of floating point operations (in Millions) necessary to obtain L from the original matrix (ordered with METIS).
Calls, flops & time per $A \times B^T$ subroutine type
QAP8 and QAP12

A Framework for Efficient Execution of Matrix Computations

J.R. Herrero

Appendix

Sparse Hypermatrix Cholesky

IPM matrices