Implementation of matrix library for barrier-project QP on an array processor

K.N. To    A. Burdeniuk    C.C. Lim

November 29, 2012
Outline

1 Introduction
   • Motivation
   • The array processor

2 The SVM QP algorithms
   • The SVM training problem
   • Barrier-projection QP problem
   • SVM QP using BPQP solver

3 The matrix library
   • MDML framework
   • Atomic Loops

4 Implementation
Our group has created an array processor architecture that is optimized for matrix computation.

It is a VLIW architecture.

To demonstrate it, we want to implement support vector learning.

Support vector learning requires a single linear constrained, box inequality constrained QP solver.

We believe the Barrier-Projection QP solver is suited for this architecture.

To implement it, we need a matrix library that can assist with VLIW instruction generation.
The hardware – array processor

What does the hardware look like, conceptually?
The hardware – basic concept

What does the hardware do?

- We call our architecture MatRISC.
- The MatRISC processor an attached processor for matrix computation:
  - assist a general load-store processor
  - off-loads common task in matrix algorithm – address generation, loop management, loop branching.
  - allow parallelization by providing transport link for row and column operation
  - allow data to be stored in a distributed memory architecture.
- MatRISC is VLIW controlled
- Parallelization needs to be achieved at compile-time.
- MatRISC is implemented using a matrix data management layer (MDML)
What is the MDML, conceptually?

Matrix Data Management Layer
Mesh Network Interconnection + Matrix Address Generation

K.N. To, A. Burdeniuk, C.C. Lim
Implementation of matrix library for barrier-project QP on an array processor
What is the MDML architecture?
Example: how do we map matrix multiplication?

matrix multiplication
C = AB

C = 0
for i = 0:p-1,
   for j = 0:r-1,
      for k = 0:q-1,
     
     ... in
block form
matrix multiplication algorithm 
in sub-block form

matrix multiplication algorithm in sub-block form
C = 0 for i = 0:p-1, 
   for j = 0:r-1, 
   for k = 0:q-1, 

z = i*r + j, 
x = i*q + k, 
y = k*r + j, 

C(z) = C(z) + A(x)B(y)
end
end
end

K.N. To, A. Burdeniuk, C.C. Lim
Implementation of matrix library for barrier-project QP on an
for \( i = 0 \) to \( p - 1 \),
for \( j = 0 \) to \( r - 1 \),
\( t = 0 \)
for \( k = 0 \) to \( q - 1 \),
\( x = i \times q + k \)
\( y = k \times r + j \)
\( t = t + A(x)B(y) \)
\( z = i \times r + j \)
\( C(z) = t \)

Step 1.
Step 2.
Step 3.
Step 4.
Mapping algorithms

K.N. To, A. Burdeniuk, C.C. Lim

Implementation of matrix library for barrier-project QP on an array processor

write source code

C/C++ source code

MDML compiler

MDML instructions

gcc compiler

embedded processor instructions

looplevel_begin:
set_xbar(....)
t = 0
t = t + A(x)B(y)
goto next_looplvl

looplevel_3:
set_xbar(....)
t = t + A(x)B(y)
goto next_looplvl

looplevel_2+1:
set_xbar(....)
C(z) = t
t = 0
t = t + A(x)B(y)
goto next_looplvl

looplevel_exit:
set_xbar(....)
C(z) = t
Choosing an application to demonstrate the MDML architecture

- We need dense matrix algorithms to demonstrate to show that mapping is possible.
- Another field of work which our authors are involved in provided the application – support vector machine (SVM) learning.
- The training stage of the SVM requires a dense matrix QP algorithm to be solved.
Outline

1 Introduction
   - Motivation
   - The array processor

2 The SVM QP algorithms
   - The SVM training problem
   - Barrier-projection QP problem
   - SVM QP using BPQP solver

3 The matrix library
   - MDML framework
   - Atomic Loops

4 Implementation
What is the SVM training algorithm?

- Consider a set of $k$ labeled input patterns such that
  
  $$
  z_i \in \mathbb{R}^k, \quad y_i \in \{-1, +1\}, \quad i = 1, \ldots, k
  $$

  where $z_i$ is the $i$-th input vector that belongs to the binary class $y_i$.

- The SVM learning problem is to find a hyper-plane
  
  $$(w \cdot z) + b = 0$$

  such that the margin hyper-planes satisfy

  $$y_i (w \cdot z_i + b) \geq 1 - \epsilon_i, \quad \forall i$$

  that maximizing the margin of separation $1/||w||_2$, where $\epsilon$ and $b$ are constants.
The dual of this problem expressed in matrix-vector form is:

\[
\min_{\alpha} \quad f(\alpha) = \frac{1}{2} \alpha^T Q \alpha - 1_k \alpha
\]  

subject to

\[
g(\alpha) = \alpha^T y = 0
\]

\[
0 \leq \alpha_i \leq C, \quad i = 1, ..., k
\]

with

\[
Q_{ij} = y_i y_j K(\alpha_i, \alpha_j)
\]

\[
y = \begin{bmatrix}
y_1 \\
\vdots \\
y_k
\end{bmatrix}
\]

where \(\alpha\) is the Lagrange multipliers, and the kernel function \(K\) allows for non-linear classification.
Choosing the QP solver

- SVM QP problem size is proportional to size of training set
- For good result, large training set is often used.
- Large QP problems will not fit in memory
- We need to use block relaxation QP algorithm to solve large problem.
- MatRISC performs optimally with no branching within loops
- We choose Barrier-Projection with Space-Transformation due to a number of useful properties:
  - inequality constrain is handled by asymptotic barrier
  - distance to barrier is stored in iterative solution
  - matrix size can be choose to be integral size of array and is fixed in size of the QP iterations
  - equality plane is handled by projector
  - starting point selection is simple (i.e. does not need to meet equality constraint)
Problem X: The SVM learning problem can be rewritten as:

$$\min_x f(x) \quad \text{subject to} \quad g(x) = 0_m, \ x \in P,$$

Problem X is formulated in $x$-space and is constrained to the positive $x$ domain $P$, if there exists another space $y \in \mathcal{P}$ and a function $\xi$ that maps to the $x$-space such that

$$\xi: x = \xi(y) \in P, \ \forall y,$$

then problem in $x$-space, if mapped to the $y$-space, can be solved without inequality constraints if $\mathcal{P}$ spans $\mathbb{R}^n$. 
Problem Y: Problem X after mapping to the $y$-space can be expressed as:

$$\min_y \tilde{f}(y) \quad \text{subject to} \quad \tilde{g}(y) = 0_m, \; y \in \mathcal{P} \in \mathbb{R}^n$$

(4)

where $\tilde{f}(y) = f(\xi(y))$ and $\tilde{g} = g(\xi(y))$.

the Lagrange form:

$$\tilde{L}(y, u) = \tilde{f}(y) + \tilde{g}^T(y)\tilde{u}(y), \quad y \in \mathcal{P}$$

(5)

where

$$\tilde{u}(y) = (\tilde{g}_y^T(y)\tilde{g}_y(y))^{-1}(\tilde{g}_y^T(y)\tilde{f}_y(y) + \tau\tilde{g}(y))$$
The solution of Problem X can be solve using the iterative expression:

\[
x_{k+1} = x_k - h_k J(x_k) \left[ \pi \left( g_x(x_k) J(x_k) \right) J^T(x_k) f_x(x_k) + \tau \left( g_x(x_k) J(x_k) \right)^+ g(x_k) \right]
\]

- the function \( \pi(W) \) is the orthogonal projector of \( W \) and is given by

\[
\pi(W) = I - W^+ W
\]

where \( W^+ \) is the pseudo-inverse of \( W \).
The algorithm – BPQP algorithms

For each iteration of $x_{k+1}$ in the steepest descent sequence, the step-size can be calculated by

$$h_k = \text{arg} \min_{h_k > 0, \ x_{k+1} \geq 0} f(x_{k+1})$$  \hspace{1cm} (6)

where

$$x_{k+1} = x_k - h_k J(x_k) L_x(x_k, u_k)$$  \hspace{1cm} (7)

and $L_x$ is the barrier projected gradient using the quadratic space transform.

The BPQP solver using quadratic space transform is:

$$x_{k+1} = x_k - h_k J(x_k) \pi \left( J^T(x_k) g_x(x_k) \right) J^T(x_k) f_x(x_k)$$

$$- h_k \tau_k J(x_k) \left( J^T(x_k) g_x(x_k) \right)^+ g(x)$$
Steepest Descent Barrier Projection Method for QP Problem

Require: \( Q, f, A, b, x_0, C, \epsilon, \tau \)
Let pseudo-inverse \( W^+ = W^T(WW^T)^{-1} \)
Let function \( J(x_k) \leftarrow \text{diag}(x_k) \)
Let function \( \pi(W) = I - W^+ W \)
\( x_k \leftarrow x_0 \)
repeat
\( g(x_k) \leftarrow Ax_k - b \)
\( f_x(x_k) \leftarrow Qx_k - f \)
\( m_x(x_k) \leftarrow J(x_k)\pi(J(x_k)g_x(x_k))J^T(x_k)f_x(x_k) \)
\( n_x(x_k) \leftarrow J(x_k)(J(x_k)g_x(x_k))^+J^T(x_k)g(x_k) \)
\( h_{\text{steepest}} \leftarrow \frac{f_x^T(x_k)m_x(x_k) - \tau n_x^T(x_k)Qm_x(x_k)}{m_x^T(x_k)Qm_x(x_k)} \)
\( h_{\text{max}} \leftarrow \arg \min_{x_k^{(i)} > 0} \left( x_k \odot \left[ J(x_k)L_x(x_k, u) \right] \right) \)
\( h_k \leftarrow \min\{h_{\text{max}}, h_{\text{steepest}}\} \)
\( x_{k+1} \leftarrow x_k - h_k m_x(x_k) - \tau n_x(x_k) \)
until \( x_{k+1} \) satisfy KKT conditions
return \( x \)
For very large SVM problems, we use Lagrangian relaxation techniques to shrink the large SVM optimization problem. In the SVM community this is the chunking algorithm.

Iteratively solve the relaxed BPQP problem to obtain the full SVM QP problem.

We make size of the relaxed QP problem to be integer number divisible by the array size. We apply BPQP to the relaxed problem.

this maximize array utilization
Implementing the algorithm

- We need to implement the basic operators used in the barrier projection algorithm.
- We need a matrix library and a framework to represent the hardware.
Outline

1 Introduction
   • Motivation
   • The array processor

2 The SVM QP algorithms
   • The SVM training problem
   • Barrier-projection QP problem
   • SVM QP using BPQP solver

3 The matrix library
   • MDML framework
   • Atomic Loops

4 Implementation
The software – matrix library

- Use C++ to create a MatRISC class
- MatRISC class is built using component class based on hardware
  - mLoop class – used to manage loop management
  - mAgu class – used to manage address generation
  - mStorage class – used to represent full matrix array
  - mBlock class – used to represent a block slice of mStorage with block slice operators
  - Iterator class – used to access array size mBlocks in the mStorage
An algorithm will have a set of input matrix array.
Input matrix are converted to mStorage
An algorithm is built using a series of one or more mLoops
One mLoops implements a nested loop or programmable depth and size
Each loop will have a set of matrix iterators
Each iterator is paired with an Agu that generates the pattern
A dereferenced iterator returns a mBlock
// example to multiply A[9x10] and B[10x11]
void main(void) {

    const int bsize = 4; // array size
    const int tablesize = 16; // loop table size
    // matrix dimensions
    const int p = 9;
    const int q = 10;
    const int r = 11;
    // create loop table
    loopTable<> ltable(tablesize);
    // set up nested loop
    unsigned multiply_loop[4] = { 9,11,10,0 }; // add to loop table
    int mt0 = ltable.addEntry(multiply_loop);

    // create matrix storage
    mStorage<double> A(p,q);
    mStorage<double> B(q,r);
    mStorage<double> C(p,r);
    // register agu parameters
    int A_access = ltable.getLoop(mt0).aguEntry(0,1,4,0);
    int B_access = ltable.getLoop(mt0).aguEntry(1,0,4,0);
    int C_access = ltable.getLoop(mt0).aguEntry(1,1,4,0);
    // create iterators
    mStorage<double>::mBlock::iterator itA(loopTable.getLoop(mt0), A_access);
    mStorage<double>::mBlock::iterator itB(loopTable.getLoop(mt0), B_access);
    mStorage<double>::mBlock::iterator itC(loopTable.getLoop(mt0), C_access);
    //...
/\ ...
/\ reset the agu
itA = A.begin();
itB = B.begin();
itC = A.begin();
/\ reset the loop
multiply_loop.begin();

/\ start he algorithm
(*itC).zero();
while( multiply_loop.looplevel() != multiply_loop.eol ) {
    switch (multiply_loop.looplevel()) {
        case 0: // loop atom 0
            for (int j=0; j<bsize; j++)
                itC = itC + itA.colcopy(j) * itB.rowcopy(j);
            itA++; itB++;
            break;
        case 1: // loop atom 1
            itA++; itB++; itC++;
            (*itC).zeros();
            break;
        case 2: // loop atom 2
            itA++; itB++; itC++;
            (*itC).zeros();
            break;
        default:
            break;
    }
    multiply_loop.next();
}
// mStorage C contains the result

K.N. To, A. Burdeniuk, C.C. Lim
Implementation of matrix library for barrier-project QP on an
There are two parts to implementing an algorithm:
- setting up the iterators
- unwrapping nested loops into loop atoms.

Loop atoms are code snippets that represent all the branch paths in a nested loop.

The MDML keeps tracks of which loop atoms need to be executed using the a single looplevel metric.

Atomizing nested loops basically converts nested loops into a single loop that selects an atom base on the loop-level value.
for(i=0;i<N_1;i++)
{
   A_b
   for(j=0;j<N_2;j++)
   {
      B_b
      for(k=0;k<N_3;k++)
      {
         C
      }
      B_e
   }
   A_e
}
State - Counter Action

S0 - \{N_i, N_j, N_k, N_l\}
S1 - \{i - 1, j, k, l\}
S2 - \{N_i, j - 1, k, l\}
S3 - \{N_i, N_j, k - 1, l\}
S4 - \{N_i, N_j, N_k, l - 1\}
S5 - \{N_i, N_j, N_k, N_l\}

State - Code Block

S0 - Execute $A_b B_b C_b D_b$
S1 - Execute $E$
S2 - Execute $D_e D_b E$
S3 - Execute $D_e C_e C_b D_b E$
S4 - Execute $D_e C_e B_e B_b C_b D_b E$
S5 - Execute $D_e C_e B_e A_e$
1 Introduction
   - Motivation
   - The array processor

2 The SVM QP algorithms
   - The SVM training problem
   - Barrier-projection QP problem
   - SVM QP using BPQP solver

3 The matrix library
   - MDML framework
   - Atomic Loops

4 Implementation
Implementation of this library is in progress.

We built a C++ library of basic algorithms such as matrix multiplication, matrix addition, LU decomposition, transposition, forward substitution, diagonalization, etc.

Matrix library has two modes:

- Test mode
  - test mode
  - pseudo-VLIW output mode

Test mode allows algorithms to be written and tested on a desktop computer.

Once tested, we can switch to output mode to generate the necessary VLIW code for the MDML engine.