Optimizing APL Matrix Indexing for Application Programmers

Eugene Ying
Software Development

Aug 8, 2011

fiserv.

Introduction

In the business world where APL is being used to perform large scale data processing, data matrices tend to be very large, with millions of rows and thousands of columns of data. When a very large matrix is manipulated thousands of times inside a loop, optimizing matrix indexing is important in reducing the function's run time. There are many cases where an application program's run time can be significantly reduced when data indexing has been optimized.

This paper talks about several techniques that can be used to optimize APL matrix indexing. The CPU time shown in the examples were based on Dyalog AIX V12.1 64-bit Classic APL running on the IBM pSeries server.

fiserv.

Indexing Optimization Techniques

The major topics to be discussed are:

Matrix Column Data Fragmentation
Progressive Indexing
Index Manipulation
Grade Up and Grade Down Indices

fiserv.

Multidimenstional Array Storage

According to Wikipedia,

"For storing multidimensional arrays in linear memory,
row-major order is used in C;
column-major order is used in Fortran and MATLAB."

We notice that APL also stores array items in row-major order.

fiserv.

Matrix Column Data Fragmentation

Matrix

1;1	1;2	1;3	1;4
2;1	2;2	2;3	2;4
3;1	3;2	3;3	3;4
4;1	4;2	4;3	4;4

Matrix[3;]

1;1	1;2	1;3	1;4	2;1	2;2	2;3	2;4	**3;1**	**3;2**	**3;3**	**3;4**	4;1	4;2	4;3	4;4

Matrix[;3]

1;1	1;2	**1;3**	1;4	2;1	2;2	**2;3**	2;4	3;1	3;2	**3;3**	3;4	4;1	4;2	**4;3**	4;4

fiserv.

A Wider Matrix

Matrix

1;1	1;2	1;3	1;4	1;5	1;6	1;7	1;8
2;1	2;2	2;3	2;4	2;5	2;6	2;7	2;8
3;1	3;2	3;3	3;4	3;5	3;6	3;7	3;8
4;1	4;2	4;3	4;4	4;5	4;6	4;7	4;8

Matrix[3;]

1;1	1;2	1;3	1;4	1;5	1;6	1;7	1;8	2;1	2;2	2;3	2;4	2;5	2;6	2;7	2;8	3;1	3;2	3;3	3;4	3;5	3;6	3;7	3;8	4;1	4;2	4;3	4;4	4;5	4;6	4;7	4;8

Matrix[;3]

1;1	1;2	1;3	1;4	1;5	1;6	1;7	1;8	2;1	2;2	2;3	2;4	2;5	2;6	2;7	2;8	3;1	3;2	3;3	3;4	3;5	3;6	3;7	3;8	4;1	4;2	4;3	4;4	4;5	4;6	4;7	4;8

The wider the matrix, the more fragmented the data columns.

fiserv.

Virtual Memory

L1
L2
L3

Seek Time (tracks)
Rotational Latency (sectors)
Transfer Time

For a very large matrix

An APL data row is concentrated in the cache, or a few pages of RAM, or perhaps a few sectors of disk.

An APL data column is scattered among a few locations in the cache, and many pages of RAM, and perhaps many sectors or tracks of disk.

fiserv.

AIX APL Run Time

```
N←10000
A←(N,N)⍴⎕AV     ⍝ 1000 by 1000 matrix
:For I :In ⍳10000
     J←N?N
     K←?N
     L←⎕AV[?256]
     A[J;K]←L   ⍝ Random items J of column K  9492 CPU ms
     A[K;J]←L   ⍝ Random items J of row K      725 CPU ms
:EndFor
```

For a 1,000 by 1,000 character matrix, random row access is on the average more than 10 times faster than random column access.

fiserv.

Another Column Data Fragmentation Example

```
Y←1000 1000⍴0.1+⍳1000
:For I :In ⍳1000
    {}+/+/Y   ⍝ Sum matrix columns first  18277 CPU ms
    {}+/+⌿Y   ⍝ Sum matrix rows first     8178 CPU ms
    {}+/,Y    ⍝ Sum vector                6445 CPU ms
:EndFor
```


fiserv.

Platform & Version Dependency

Although these observations on CPU times are generally true on all platforms, the exact performance varies depending on the hardware configuration, especially the quantity and quality of cache relative to the size of the arrays being manipulated. The performance might vary in the future versions of APL.

fiserv.

Matrix Organization Suggestion

When you design a large data matrix for an APL application, always ask this question.

"Will I be accessing the data columns more frequently than accessing the data rows?"

If the answer is yes, then you should consider redesigning your data matrix such that it is in the transposed format so that you will access consecutive memory more frequently.
If some of your virtual matrix data are on a disk, consecutive disk sectors can greatly speed up your program.

fiserv.

Marketing Data Example

Assuming we have the data of a million customers or prospects, and each customer has 100 data attributes, the first 5 of which are customer ID, country code, # of employees, industry code, and revenue.

We construct the following large random data matrix for testing.

```
ID←⍳1010000
CTY←((1000000⍴⍳80),10000⍴85)[1010000?1010000]
EMP←(1010000⍴(5⍴1000),(10⍴500),(20⍴200),50⍴100)[1010000?1010000]
IND←(1010000⍴⍳9999)[1010000?1010000]
REV←(10100000+1000000×⍳1010000)[1010000?1010000]

DAT←100↑[2]ID,CTY,EMP,IND,[1.1]REV
```


fiserv.

A Marketing Data Example

1 million rows of customers

Company ID	Country Code	# of Employees	Industry Code	Revenue	…	…	…

fiserv.

Marketing Data Selection Example

To select companies with more 1000 employees in the manufacturing industry in country 85, most APL programmers would use a one-liner Boolean logic to get the data indices.

```
MFG←2011 2012 2013 2014 2015
:For I :In ⍳100
    J←((DAT[;3]≥1000)∧(DAT[;4]∈MFG)∧DAT[;2]=85)/⍳1↑⍴DAT
    ⍝ 28733 ms CPU
:EndFor
```


fiserv.

Marketing Transposed Data Example

Let us transpose the data matrix so that instead of 1 million rows of data for the 1 million customers, we have 1 million columns of data for the 1 million customers.

1 million columns of customers

Company ID
Country Code
of Employees
Industry Code
Revenue
…
…
…

fiserv.

Marketing Data Selection Example

Data Organized by Columns

```
MFG←2011 2012 2013 2014 2015
:For I :In ⍳100
    J←((DAT[;3]≥1000)∧(DAT[;4]∈MFG)∧DAT[;2]=85)/⍳1↑⍴DAT
    ⍝ 28,733 ms CPU
:EndFor
```

Data Organized by Rows

```
:For I :In ⍳100
    J←((DAT[3;]≥1000)∧(DAT[4;]∈MFG)∧DAT[2;]=85)/⍳1↑⍴DAT
    ⍝ 12,227 ms CPU
:EndFor
```


fiserv.

Progressive Indexing

Data Organized by Rows

```
:For I :In ⍳100
     J←(DAT[3;]≥1000)/⍳¯1↑⍴DAT     ⍝  2,393 ms CPU
     J←(DAT[4;J]∈MFG)/J             ⍝    612 ms CPU
     J←(DAT[2;J]=85)/J              ⍝      5 ms CPU
:EndFor                             ⍝  3,000 ms CPU Total
```

In multi-line progressive indexing,

Expression 2: DAT[4;J]∈MFG works with fewer items than expression 1: DAT[3;]≥1000,

Expression 3: DAT[2;J]=85 works with fewer items than expression 2: DAT[4;J]∈MFG.

fiserv.

Progressive Indexing Optimization

Suppose we know that there are not too many country 85 records in the data matrix, and there are hardly any big companies in this country. A better filtering sequence would be:

```
:For I :In ⍳100
    J←(DAT[2;]=85)/⍳¯1↑⍴DAT   ⍝   1782 ms CPU
    J←(DAT[3;J]≥1000)/J       ⍝      9 ms CPU
    J←(DAT[4;J]∈MFG)/J        ⍝      4 ms CPU
:ENDFOR                       ⍝   1795 ms CPU total
```


fiserv.

Marketing Data Selection Speed Comparisons

```
:For I :In ⍳100    ⍝ Column Data
    J←((DAT[;3]≥1000)∧(DAT[;4]∈MFG)∧DAT[;2]=85)/⍳1↑⍴DAT
:EndFor                                  ⍝ 28,733 ms


:For I :In ⍳100    ⍝ Row Data
    J←((DAT[3;]≥1000)∧(DAT[4;]∈MFG)∧DAT[2;]=85)/⍳1↑⍴DAT
                                         ⍝ 12,227 ms

    J←(DAT[3;]≥1000)/⍳¯1↑⍴DAT      ⍝  2,393 ms
    J←(DAT[4;J]∈MFG)/J              ⍝    612 ms
    J←(DAT[2;J]=85)/J               ⍝      5 ms
                                    ⍝  3,000 ms Total

    J←(DAT[2;]=85)/⍳¯1↑⍴DAT        ⍝   1782 ms
    J←(DAT[3;J]≥1000)/J             ⍝      9 ms
    J←(DAT[4;J]∈MFG)/J              ⍝      4 ms
                                    ⍝  1,795 ms total
:EndFor
```


fiserv.

Row Major Progressive Indexing

With proper arrangement of the Boolean statements such that most of the unwanted data are filtered out by the first and second statements, for very large matrices, progressive indexing can be many times faster than the single Boolean statement in performing data selection. In the previous example, we see the function is 16 times faster when progressive indexing is performed on row-major data.

fiserv.

Index Manipulaton

It is usually more efficient to manipulate a matrix inside the square brackets [ ] than outside the [ ].

```
Y←1000 100⍴⎕A
T←1000⍴1 0 0
:For I :In ⍳100000
     {} 4⌽Y[;68+⍳8]        ⍝ 6006 ms
     {} Y[;4⌽68+⍳8]        ⍝ 2110 ms

     {} 0 1↓Y[;⍳7]         ⍝ 4448 ms
     {} Y[;1↓⍳7]           ⍝ 2049 ms

     {} T≠Y[;⍳6]           ⍝ 2330 ms
     {} Y[T/⍳1↑⍴Y;⍳6]      ⍝  982 ms

     {} Y[;1 3],Y[;2 4]    ⍝ 6359 ms
     {} Y[;1 3 2 4]        ⍝ 1561 ms
:EndFor
```


fiserv.

Grade Up & Grade Down Index

It is usually more efficient to manipulate the grade up index or the grade down index than to manipulate the sorted matrix.

```
R←X ∆SUM Y;J;P
[1] X←X[⍋X[;Y];]                          ⍝ Sort X on columns Y
[2] P←{1=ρρ⍵:1,(1↓⍵)≠¯1↓⍵                  ⍝ Unique vector items
[3]      1,∨/(1↓[1]⍵)≠¯1↓[1]⍵}X[;Y]        ⍝ Unique matrix rows
[4] R←P⌿X                                 ⍝ Rows with unique columns Y
[5] ...
```

can be rewritten as

```
R←X ∆SUM2 Y;I;J;P
[1] I←⍋X[;Y]                              ⍝ Sort X on column Y
[2] P←{1=ρρ⍵:1,(1↓⍵)≠¯1↓⍵                  ⍝ Unique vector items
[3]      1,∨/(1↓[1]⍵)≠¯1↓[1]⍵}X[I;Y]       ⍝ Unique matrix rows
[4] R←X[P/I;]                             ⍝ Rows with unique columns Y
[5] ...
```


fiserv.

Speed and Space Comparisons

```
M←(10000⍴⍳100),10000 100⍴⍳1000000
:For I :In ⍳10000
     {}M ⍙SUM 1        ⍝ 20647 ms
     {}M ⍙SUM2 1       ⍝  4542 ms
:EndFor
```

```
      ⎕WA
4044960
      {}DATA ⍙SUM 1
WS FULL
⍙SUM[1] X←X[⍋⍙X[;Y];]
        ^
```

```
      ⎕WA
121960                          ⍝ Much less storage requirement
      {}DATA ⍙SUM2 1            ⍝ No WS FULL
```


fiserv.

Array Dimensions

How many columns are there in matrix M?

`ρM[1;]` should be coded as `¯1↑ρM`

How many rows are there in matrix M?

`ρM[;1]` should be coded as `1↑ρM`

How many planes and columns are there in the 3-dimensional array A?

`ρA[;1;]` should be coded as `(ρA)[1 3]`

fiserv.

Conclusion

When you perform your data selection on a large matrix using carefully constructed progressive indexing instead of a simple APL one-liner Boolean logic, the run time can be reduced significantly. If the data to be progressively indexed are arranged in rows instead of in columns, the run time can be reduced even more.

In optimizing matrix indexing, we need to pay special attention to the ones that are in the innermost loop of a function. Optimizing the matrix indexing deep inside an intensive loop would give you the maximum benefits.

fiserv.