Matrix-vector multiplication

Comparing performance of matrix by vector multiplication in C++ and Streaming SIMD (Single Instruction Multiple Data) Extension

Homework CS342
Fall 2007

Presented by Rafal Sytek
rafal.sytek@gmail.com
Goal of the project

- The goal of this project is to demonstrate efficiency that can be gained by using SSE code in matrix-vector multiplication
Matrix-vector multiplication basics

- Matrix A m-by-n
- Vector B n-by-p
- A * B produces vector m-by-p
Functions used

- CPP_MatrixVectorMult
- SSE_MatrixVectorMult

Both functions return pointer to the resultant array; they take pointers to the matrix and vector arrays, as well as size of the matrix as the parameters

- Based on the matrix-vector multiplication optimization code from http://www.codeproject.com/tips/MatrixVector.asp
float* CPP_MatrixVectorMult(float *matrix, float *vector, int size)
{
    // return pointer to the array containing result
    float *vectorResult = new float[size];

    // use it for pointer arithmetic to traverse the arrays
    float *matrixPtr = matrix;
    float *vectorPtr;

    // store the value for each row/column multiplication
    float result;
for (int i=0; i<size; ++i) {
    // reset 'vectorPtr' to the beginning of the array
    // reset 'result' to 0
    vectorPtr = vector;
    result    = 0.0;

    // multiply each row of matrix with vector column
    // and assign this value to the output vector 'vectorResult'
    // advance pointers to next value with each iteration of 'j'
    for (int j=0; j<size; ++j) {
        result += (*matrixPtr) * (*vectorPtr);
        ++vectorPtr;
        ++matrixPtr;
    }
    vectorResult[i] = result;
}
CPP_MatrixVectorMult – Explanation

\[ A = [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p] \]
\[ B = [P, Q, R, S] \]

\[
\begin{bmatrix}
    a & b & c & d \\
    e & f & g & h \\
    i & j & k & l \\
    m & n & o & p \\
\end{bmatrix}
\begin{bmatrix}
P \\
Q \\
R \\
S \\
\end{bmatrix}
= 
\begin{bmatrix}
aP + bQ + cR + dS \\
aP + fQ + gR + hS \\
iP + jQ + kR + lS \\
mP + nQ + oR + pS \\
\end{bmatrix}
\]

matrixPtr
vectorPtr

\[ A = [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p] \]
\[ B = [P, Q, R, S] \]
CPP_MatrixVectorMult – Explanation

A = [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p]
B = [P, Q, R, S]

\[ \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} P \\ Q \\ R \\ S \end{bmatrix} = \begin{bmatrix} aP + bQ + cR + dS \\ eP + fQ + gR + hS \\ iP + jQ + kR + lS \\ nP + nQ + oR + pS \end{bmatrix} \]

i = 4, j = 0:

\[ \text{result } += (\text{matrixPtr}) \times (\text{vectorPtr}); \]

i = 4, j = 0:

A = [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p]

B = [P, Q, R, S]
Why is SSE code faster than C++?

- In C++ implementation we use SISD: **Single Instruction, Single Data**
- In SSE implementation we use SIMD: **Single Instruction, Multiple Data**
SSE_MatrixVectorMult - Introduction
SSE_MatrixVectorMult - Introduction

- SIMD advantages
  - Single data can be added/subtracted etc. to different data points at the same time
    Example: inverting RGB image to its negative

- SIMD disadvantages
  - Limited number of registers (speed vs. size)
  - Reading/writing might be an issue (technological challenges)
SSE_MatrixVectorMult - Introduction

- SIMD uses SSE (Streaming SIMD Extension) instruction set that allows parallel computations
- Developed by Intel in 1999
- Original SSE contained 70 instructions and eight 128 bit registers
- SSE5 is the most recent version (AMD), adds 170 new instructions
Sample output for SSE:

Code to determine CPU capabilities (including identifying SSE instruction set) is available from:
SSE_MatrixVectorMult - Introduction

- Elements in SSE
  - Each register (XMM0 to XMM7) has 128 bits and can store four 32-bits floating point numbers

![Diagram of SSE register bits and significand elements]

- MSB
- Most Significant Element
- LSB
- Least Significant Element
SSE_MatrixVectorMult - Introduction

- Packed instructions
  - Performed on all four data types
  - Uses ‘ps’ suffix (ADDPs – Add Packed Single-Precision Values or Parallel Scalars)

- Scalar instructions
  - Performed on the least significant element
  - Uses ‘ss’ suffix (ADDSs – Add Scalar Single-Precision Values or Single Scalar)
SSE_MatrixVectorMult - Introduction

- Packed instruction example

![Diagram showing the packed instruction example with matrices A and B and an operator op. The diagram illustrates the operations a3 op b3, a2 op b2, a1 op b1, and a0 op b0.]
SSE_MatrixVectorMult - Introduction

- Scalar instruction example
SSE_MatrixVectorMult - Instructions

- **XORPS** – Bitwise Exclusive-Or Parallel Scalars, returns a bitwise logical XOR between the source and destination operands

- **MOVUPS** – Move Unaligned Parallel Scalars, moves a double quadword containing four values from the source operand to the destination

- **MOVAPS** – Move Aligned Parallel Scalars, moves a double quadword containing four values from the source operand to the destination
SSE_MatrixVectorMult - Instructions

- **MOVLPS** - Move Low Pair Parallel Scalars, moves two packed values between the source and destination operands. One of the operands is a 64-bit memory location, the other is the low quadword of an XMM register.

- **MOVHPS** - Move High Pair Parallel Scalars, moves two packed values between the source and destination operands. One of the operands is a 64-bit memory location, the other is the high quadword of an XMM register.

- **MULPS** - Multiply Parallel Scalars, performs a multiplication of the packed values in both operands, and stores the results in the destination register.
SSE_MatrixVectorMult - Instructions

- **MULSS** - Multiply Single Scalar, multiplies the lowest values of both operands, and stores the result in the low doubleword of XMM register

- **MOVSS** - Move Single Scalar, moves a scalar value from the source operand to the destination operand

- **ADDPS** - Add Parallel Scalars, performs addition on each of four value pairs

- **ADDSS** - Add Single Scalars, adds the low values from the source and destination operands and stores the single-precision result in the destination operand
SSE_MatrixVectorMult - Instructions

- **SHUFPS** - Shuffle Parallel Scalars, moves two of the four packed values from first operand into the low quadword of the destination operand; moves two of the four packed values in the source operand into the high quadword of the destination operand. The third operand (select) determines which values are moved to the destination operand.

```
shufps xmmreg1, xmmreg2, mask
```

Output (xmmreg1):

```
[ ] [ ] [ ]
```
SSE_MatrixVectorMult - Instructions

Mask specifies which bits will be copied to the output:

\[
\begin{array}{c}
\text{xmmreg}2 & \text{xmmreg}1 \\
\hline
11 & 10 & 01 & 00 \\
\end{array}
\]
SSE_MatrixVectorMult - Instructions

Examples:

<table>
<thead>
<tr>
<th>XMM1</th>
<th>11</th>
<th>10</th>
<th>01</th>
<th>00</th>
<th>XMM2</th>
<th>11</th>
<th>10</th>
<th>01</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td></td>
<td>5.0</td>
<td>6.0</td>
<td>7.0</td>
<td>8.0</td>
</tr>
</tbody>
</table>

- `shufps xmm1, xmm2, 11001100b`  
  
- `shufps xmm1, xmm2, 01111001b`  
  
- `shufps xmm2, xmm1, 01111001b`  
  
- `shufps xmm1, xmm1, 10001100b`  
  
- `shufps xmm2, xmm2, 10010011b`  
  
- `shufps xmm1, xmm1, 00111001b`  

5.0  8.0  1.0  4.0
7.0  5.0  2.0  3.0
3.0  1.0  6.0  7.0
2.0  4.0  1.0  4.0
6.0  7.0  8.0  5.0
4.0  1.0  2.0  3.0
### Example for column-major matrix:

$$\begin{bmatrix}
1, & 2, & 3, & 4 \\
5, & 6, & 7, & 8 \\
9, & 10, & 11, & 12 \\
13, & 14, & 15, & 16 \\
\end{bmatrix} \begin{bmatrix}
2 \\
3 \\
4 \\
5 \\
\end{bmatrix} = \begin{bmatrix}
1*2 + 2*3 + 3*4 + 4*5 \\
5*2 + 6*3 + 7*4 + 8*5 \\
9*2 + 10*3 + 11*4 + 12*5 \\
13*2 + 14*3 + 15*4 + 16*5 \\
\end{bmatrix} = \begin{bmatrix}
40 \\
96 \\
152 \\
208 \\
\end{bmatrix}$$
void matrixMultiplication(float *matrixIn, // 4x4 matrix
                         float *vectorIn,    // 4x1 vector
                         float *vectorOut)   // 4x1 vector
{
    // pointer row points to 16 elements array (beginning of the
    // matrixIn array) containing column-major elements:
    // [1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 16]

    float *row = matrixIn;

    __asm
    {
        mov  esi,    vectorIn    // load input address
        mov  edi,    vectorOut   // load output address
    }
SSE_MatrixVectorMult - Code

// pointer to the first 4 elements of the array:

mov edx, row  // Move Unaligned Parallel Scalars

// load the registers with matrix values:

movups xmm4, [edx]  // xmm4 contains 1,5, 9,13

movups xmm5, [edx+0x10]  // +16 (4 bytes x 4 elements)
  // xmm5 contains 2,6,10,14

movups xmm6, [edx+0x20]  // +32 (8 bytes x 4 elements)
  // xmm6 contains 3,7,11,15

movups xmm7, [edx+0x30]  // +48 (12 bytes x 4 elements)
  // xmm7 contains 4,8,12,16
SSE_MatrixVectorMult - Code

// esi contains the starting address of the vectorIn array
// [2, 3, 4, 5], so load this vector into xmm0:

movups xmm0, [esi] // xmm0 contains 2, 3, 4, 5

// we'll store the final result in xmm2, initialize it to 0:

xorps xmm2, xmm2 // xmm2 contains 4x32
   // bits with zeros

// now we need to multiply first column (xmm4)
// by the vector (xmm0) and add it to the total (xmm2)
// copy content of xmm0 into xmm1:

movups xmm1, xmm0  // xmm1 now contains [2, 3, 4, 5]

// each value in xmm1 has the following mask representation:

// mask value: 11 10 01 00
// register value: [2, 3, 4, 5]

// Shuffle Parallel Scalars

shufps xmm1, xmm1, 0xFF  // FF mask is 11 11 11 11
                        // xmm1 contains 2, 2, 2, 2
SSE_MatrixVectorMult - Code

// Multiply Parallel Scalars

mulps xmm1, xmm4 // multiply xmm1 (2,2,2,2) 
    // and xmm4 (1,5,9,13)
    // [ 2*1, 2*5, 2*9, 2*13 ]
    // save it in xmm1

// Add Parallel Scalars

addps xmm2, xmm1 // add xmm2 (0, 0, 0, 0) 
    // and xmm1 (2, 10, 18, 26)
    // save it in xmm2
    // xmm2 contains [2,10,18,26]

// we multiplied first column of the matrix by the vector,
// now we need to repeat these operations for the remaining
// columns of the matrix
SSE_MatrixVectorMult - Code

movups xmm1, xmm0  // 3, 3, 3, 3
shufps xmm1, xmm1, 0xAA  // AA -> 10 10 10 10
mulps xmm1, xmm5  // xmm5: 2, 6, 10, 14
addps xmm2, xmm1  // 2+6, 10+18, 18+30, 26+42

movups xmm1, xmm0  // 4, 4, 4, 4
shufps xmm1, xmm1, 0x55  // 55 -> 01 01 01 01
mulps xmm1, xmm6  // xmm6: 3, 7, 11, 15
addps xmm2, xmm1  // 8+12, 28+28, 48+44, 68+60

movups xmm1, xmm0  // 5, 5, 5, 5
shufps xmm1, xmm1, 0x00  // 00 -> 00 00 00 00
mulps xmm1, xmm7  // xmm7: 4, 8, 12, 16
addps xmm2, xmm1  // 20+20, 56+40, 92+60, 128+80
    // 40 ,  96 , 152 ,  208

// write the results to vectorOut
movups [edi], xmm2
SSE_MatrixVectorMult - Code

- Memory alignment
  - \texttt{__declspec(align(16))\ float\ value\_1[4];}
  - Use \texttt{__declspec} to make sure that the variables are alligned properly in the memory
  - Allows to use MOV\texttt{APS} instead of MOV\texttt{UPS}, which are faster
Results

C++ vs SSE matrix-vector multiplication

Matrix Size

Clock Ticks
Results

SSE performance relative to C++
Conclusions

- For small sizes of the matrix, C++ code performs better than SSE code
  - Caused by greater number of instructions that have to be performed in SSE code
  - Parallelism does not provide any benefits for small matrix sizes (< 16)
  - 147.3% slower (on average) than C++ code for small matrix sizes (< 16)
Conclusions

- As the size of the matrix increases, the gains in performance are high
  - Number of instructions performed by C++ code is much larger than in SSE code
  - Parallelism provides much faster computations
  - 62% faster (on average) than C++ once the matrix size is above 16 elements
References


http://developer.amd.com/sse5.jsp

http://courses.ece.uiuc.edu/ece390/books/labmanual/inst-ref-simd.html

http://www.x86.org/articles/sse_pt1/simd1.htm

http://www.intel80386.com/simd/mmx2-doc.html

http://www.cortstratton.org/articles/OptimizingForSSE.php