Benefits of Intel® AVX For Small Matrices

Introduction

Intel® Advanced Vector Extensions (Intel® AVX) extends the capabilities of Intel® Streaming SIMD Extensions (Intel® SSE), especially for floating point data and operations. Intel® AVX essentially doubles the width of the current XMM registers and adds new extensions that can operate on the wider data width. Intel® AVX significantly increases the floating-point performance density with improved power efficiency over previous 128-bit SIMD instruction set extensions. This document specifically examines how some of the Intel® AVX architecture features such as wider 256-bit registers, new data manipulation and arithmetic primitives can benefit operations on matrices of small sizes. The operations discussed in this whitepaper include, simple operations such as matrix addition and multiplication, as well as more complex ones like computing the determinant of a matrix.

Simulation Environment

The performance speedups stated in this paper are based on simulations done on an Intel performance simulator code named Sandy Bridge. It can therefore be assumed that the test data is already in the cache. Performance comparisons are made based on the relative performance of Intel® AVX versus corresponding Intel® SSE implementations using C intrinsics, both run on the same simulator, currently code named Sandy Bridge. The code was compiled using the Intel® C Compiler 11.0.074, a version not yet released.

The paper will first discuss Matrix Addition operation.

Matrix Addition

Given two matrices A and B, where both are [I x J], matrix addition is defined by the following equation:

$$c_{ij} = a_{ij} + b_{ij}$$

Each Cij result is calculated by the sum of each corresponding element of the matrices.

The algorithm loads eight single precision floating point numbers from each input matrix using `_mm256_load_ps` intrinsic function. The algorithm then, in parallel computes the eight corresponding sums using `_mm256_add_ps` intrinsic function. The eight results are then written to the output matrix using `_mm256_store_ps` intrinsic. The use of the new 256bit Intel® AVX extensions essentially double the bandwidth compared to 128-bit architecture by executing the load, add and store operations on eight single precision floating point numbers instead of four in parallel.

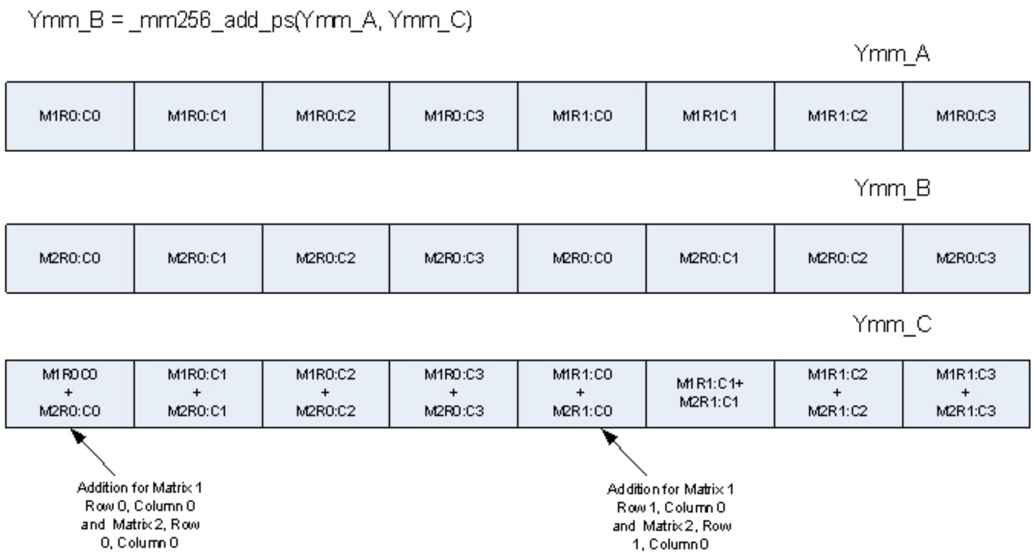

Figure 1 Addition of two 4x4 matrices

Refer to the Source Code section of this document for the matrix addition source code.

The whitepaper will next discuss the Matrix Multiplication operation.

Matrix Multiplication

Given two matrices A and B, where A is [I x N] and B is [N x J], matrix multiplication is defined by the following equation:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

Each Cij result is calculated by the dot product of each row of matrix A and a column of matrix B.

Given two 8x8 matrices, the algorithm calculates all the results for one output row through a series of parallel operations. Since the matrix multiplication requires that the each element of a row of matrix A is multiplied by the corresponding element of each column of matrix B, it follows that the same element from matrix A will be used as one of the operands in these element by element multiplications eight times. The performance can be improved by copying this element from matrix A to all locations of a YMM register. The *_mm256_broadcast_ss* instruction provides the means to broadcast, or copy, a single element to all eight single precision floating point numbers that the register contains.

The algorithm begins by reading each row of matrix B using a series of eight *_mm256_load_ps* intrinsic functions. Then the algorithm copies via *_mm256_broadcast_ss* a single element of a row from matrix A into all locations of a ymm register. Each element of this row of matrix A is copied to a ymm register in this manner.

The eight dot products are calculated in parallel by multiplying the broadcasted elements of Matrix A times a row of matrix B and then summing the results. This is achieved by a *_mm256_mul_ps* using broadcasted a11 and the first row of matrix B as operands. The multiplication is repeated using broadcasted a12 and the second row of matrix B as operands. The results of those two multiplies are added using *_mm256_add_ps* to create a partial sum. This process is repeated for a13 and the third row of Matrix B, and for a14 and the fourth row of matrix B. Two more partial sums are calculated for the remaining elements and rows. The four partial sums are then added together and stored to the output matrix C to complete the dot product calculation for this row.

This process is repeated seven more times, using a different row of Matrix A each time, to complete the matrix multiplication.

Matrix addition and multiplication are basic and simple building blocks. The paper will next discuss evaluating the determinant of a matrix which is another important matrix operation used in many applications.

Determinant of a Matrix

Determinants are very useful when analyzing systems of linear equations. Several algorithms to calculate a determinant exist. The method discussed here is the Laplacian expansion by minors defined by the following equation:

$$|M| = \sum_{j=1}^{N} m_{ij} c_{ij} = \sum_{j=1}^{N} m_{ij} (-1)^{i+j} \, |M^{\{ij\}}|$$

In the above equation the algorithm is recursively repeated for smaller sized minor matrices until a 2x2 matrix is reached, for which the determinant is easily computed.

$$M = \begin{vmatrix} a11 & a12 \\ a21 & a22 \end{vmatrix}$$

Figure 7 - 2x2 Matrix

The determinant of the 2x2 matrix is defined as:

$$|M| = a_{11}a_{22} - a_{12}a_{21}$$

Given a 4x4 matrix M,

$$M = \begin{vmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{vmatrix}$$

Figure 3 - 4x4 Matrix

the Laplacian equation expands to the following:

$$\begin{aligned}|M| = {} & a11a22a33a44 - a11a22a43a34 - a11a23a32a44 + a11a23a34a42 + \\ & a11a24a32a43 - a11a24a42a33 - a12a21a33a44 + a12a21a43a34 + \\ & a12a23a31a44 - a12a23a41a34 - a12a24a31a43 + a12a24a41a33 + \\ & a13a21a32a44 - a13a21a42a34 - a13a22a31a43 + a13a22a41a33 + \\ & a13a24a31a42 - a13a24a41a32 - a14a21a32a43 + a14a21a42a33 + \\ & a14a22a31a43 - a14a22a41a33 - a14a23a31a42 + a14a23a41a32\end{aligned}$$

Equation 1 - Determinant of 4x4 Matrix

The Intel® AVX implementation utilizes the expanded equation, Equation 1, and exploits the parallel SIMD architecture fully as it calculates the determinant of eight matrices simultaneously to maximize throughput. The algorithm is implemented as a two step process.

In the first step, the first two rows from each of the eight matrices are read and then swizzled (similar to the transpose operation) to gather the corresponding elements of each matrix into eight YMM registers. Then, the next two rows are read and swizzled.

Once the elements are gathered, the calculation of the determinants for eight matrices can be performed in parallel. After all the math functions are complete, the eight determinant values are held in a single YMM register which can then be stored in the output. The following code excerpt demonstrates the calculation for the determinant of a minor matrix for element a11. Similarly, determinants for elements a12, a13 and a14 are calculated according to the expanded equation.

- collapse sourceview plaincopy to clipboardprint?

```
// calculates a11a22(a33a44 - a43a34) for 8 matrices
ymm_a11a22 = _mm256_mul_ps(ymm_a11, ymm_a22);
ymm_a33a44 = _mm256_mul_ps(ymm_a33, ymm_a44);
ymm_a43a34 = _mm256_mul_ps(ymm_a43, ymm_a34);
ymm_a33a44_a43a34 = _mm256_sub_ps(ymm_a33a44, ymm_a43a34);
ymm_a11a22_D = _mm256_mul_ps(ymm_a11a22, ymm_a33a44_a43a34);

// calculates a11a23(a32a44 - a34a42) for 8 matrices
ymm_a11a23 = _mm256_mul_ps(ymm_a11, ymm_a23);
ymm_a32a44 = _mm256_mul_ps(ymm_a32, ymm_a44);
ymm_a34a42 = _mm256_mul_ps(ymm_a34, ymm_a42);
ymm_a32a44_a34a42 = _mm256_sub_ps(ymm_a32a44, ymm_a34a42);
ymm_a11a23_D = _mm256_mul_ps(ymm_a11a23, ymm_a32a44_a34a42);

// calculates a11a24(a32a43 - a42a33) for 8 matrices
ymm_a11a24 = _mm256_mul_ps(ymm_a11, ymm_a24);
ymm_a32a43 = _mm256_mul_ps(ymm_a32, ymm_a43);
ymm_a42a33 = _mm256_mul_ps(ymm_a42, ymm_a33);
ymm_a32a43_a42a33 = _mm256_sub_ps(ymm_a32a43, ymm_a42a33);
ymm_a11a24_D = _mm256_mul_ps(ymm_a11a24, ymm_a32a43_a42a33);

// calculate partial determinant: (a11a22(a33a44 - a43a34) +
//a11a23(a32a44 - a34a42) + a11a24(a32a43 - a42a33)) for all 8 matrices
ymm_a11a22_D = mm256_sub_ps(ymm_a11a22_D, ymm_a11a23_D);
ymm_a11_D = _mm256_add_ps(ymm_a11a22_D, ymm_a11a24_D);

```

Finally, the partial determinants for a11, a12, a13 and a14 added to get the final determinant value. Each of these calculations is done for eight matrices simultaneously to maximize throughput.

- collapse sourceview plaincopy to clipboardprint?

```
// Calculate final Determinant value: addition of determinants for
// a11, a12, a13, a14
ymm_a11_D = _mm256_add_ps(ymm_a11_D, ymm_a12_D);
ymm_a13_D = _mm256_add_ps(ymm_a13_D, ymm_a14_D);
ymm_a11_D = _mm256_add_ps(ymm_a11_D, ymm_a13_D);

```

Results

Results were acquired for both the 128-bit and the corresponding 256-bit code by running them on the Intel® AVX performance simulator. The speedups listed here are for 256-bit code relative to the128-bit code.

Table 1 - 256-bit to 128-bit Speedup Results

Algorithm	Speedup	Parameters
Matrix Addition	1.8x	4x4 matrices
Matrix Multiplication using multiply	1.77x	8x8 matrices
Matrix Determinant	1.56x	4x4 matrices

Conclusion

The results for both the 8x8 matrix multiply algorithm and 16x16 matrix addition algorithm shows that the Intel® AVX versions outperform their Intel® SSE implementations by 1.77x and 1.8x respectively. This is a good example of how Intel® AVX achieves higher throughput than Intel® SSE by operating on twice the number of operands per instruction when utilizing the entire register width.

The determinant algorithm achieves higher performance by performing determinant calculations on eight matrices simultaneously. The performance though, gets somewhat limited by the two 8x8 transpose performed before the determinant computations.

In general applications will obtain higher performance speedup if they are more compute-intensive than memory-intensive.

Source Code

The complete source code can be downloaded from **here**.

Partial source code listings are provided in the following sections.

Matrix Addition Algorithm

// The following code adds two 4x4 matrices of single-precision floats and stores result in a third 4x4 matrix
- collapse sourceview plaincopy to clipboardprint?

```
int iBufferHeight = miMatrixWidth; // width of the matrix
int iBufferWidth = miMatrixHeight; // height of the matrix


// Initialize the input buffers to the two input matrices to be added
// and output buffer to a third matrix that will hold the added values

float* pImage1 = InputBuffer1;
float* pImage2 = InputBuffer2;
float* pOutImage = OutputBuffer;


for(int iY = 0; iY < iBufferHeight; iY+= 2)
{
 // Load 8 single precision floats from input matrix 1 and 2
 __m256 Ymm_A = _mm256_load_ps(pImage1);
 __m256 Ymm_C = _mm256_load_ps(pImage2);

 // Add 8 single precision floats in parallel
    __m256 Ymm_B = _mm256_add_ps (Ymm_A, Ymm_C);

    // Store the result of 8 additions to output matrix
    _mm256_store_ps(pOutImage, Ymm_B);

  // Advance the input and output buffers
 pOutImage+=8;
   pImage1+=8;
 pImage2+=8;
}

```

Matrix Multiply Algorithm

// The following code multiplies two 8x8 matrices of single-precision floating point values
- collapse sourceview plaincopy to clipboardprint?

```
__m256  ymm0, ymm1, ymm2, ymm3, ymm4, ymm5, ymm6, ymm7,
    ymm8, ymm9, ymm10, ymm11, ymm12, ymm13, ymm14, ymm15;










//Initialize some pointers
float * pMatrix2 = InputBuffer2;
float * pIn = InputBuffer1;
float * pOut = OutputBuffer;






//Read the eight rows of Matrix B into ymm registers
ymm8 = _mm256_load_ps((float *) (pMatrix2));
ymm9 = _mm256_load_ps((float *) (pMatrix2 + 1*8));
ymm10 = _mm256_load_ps((float *) (pMatrix2 + 2*8));
ymm11 = _mm256_load_ps((float *) (pMatrix2 + 3*8));
ymm12 = _mm256_load_ps((float *) (pMatrix2 + 4*8));
ymm13 = _mm256_load_ps((float *) (pMatrix2 + 5*8));
ymm14 = _mm256_load_ps((float *) (pMatrix2 + 6*8));
ymm15 = _mm256_load_ps((float *) (pMatrix2 + 7*8));

```

```
//Broadcast each element of Matrix A Row 1 into a ymm register
ymm0 = _mm256_broadcast_ss(pIn);
ymm1 = _mm256_broadcast_ss(pIn + 1);
ymm2 = _mm256_broadcast_ss(pIn + 2);
ymm3 = _mm256_broadcast_ss(pIn + 3);
ymm4 = _mm256_broadcast_ss(pIn + 4);
ymm5 = _mm256_broadcast_ss(pIn + 5);
ymm6 = _mm256_broadcast_ss(pIn + 6);
ymm7 = _mm256_broadcast_ss(pIn + 7);

//Multiply A11 times Row 1 of Matrix B
ymm0 = _mm256_mul_ps(ymm0, ymm8);

//Multiply A12 times Row 2 of Matrix B
ymm1 = _mm256_mul_ps(ymm1, ymm9);

//Create the first partial sum
ymm0 = _mm256_add_ps(ymm0, ymm1);

//Repeat for A13, A14, and Rows 3, 4 of Matrix B
ymm2 = _mm256_mul_ps(ymm2, ymm10);
ymm3 = _mm256_mul_ps(ymm3, ymm11);
ymm2 = _mm256_add_ps(ymm2, ymm3);

//Repeat for A15, A16, and Rows 5, 6 of Matrix B
ymm4 = _mm256_mul_ps(ymm4, ymm12);
ymm5 = _mm256_mul_ps(ymm5, ymm13);
ymm4 = _mm256_add_ps(ymm4, ymm5);

//Repeat for A17, A18, and Rows 7, 8 of Matrix B
ymm6 = _mm256_mul_ps(ymm6, ymm14);
ymm7 = _mm256_mul_ps(ymm7, ymm15);
ymm6 = _mm256_add_ps(ymm6, ymm7);

//Perform the final three adds
ymm0 = _mm256_add_ps(ymm0, ymm2);
ymm4 = _mm256_add_ps(ymm4, ymm6);
ymm0 = _mm256_add_ps(ymm0, ymm4);

//Store the result to Row 1 of Matrix C
_mm256_store_ps((float *) (pOut), ymm0);




//Repeat using Matrix A Row 2
ymm0 = _mm256_broadcast_ss(pIn + 1*8);
ymm1 = _mm256_broadcast_ss(pIn + 1*8 + 1);
ymm2 = _mm256_broadcast_ss(pIn + 1*8 + 2);
ymm3 = _mm256_broadcast_ss(pIn + 1*8 + 3);
ymm4 = _mm256_broadcast_ss(pIn + 1*8 + 4);
ymm5 = _mm256_broadcast_ss(pIn + 1*8 + 5);
ymm6 = _mm256_broadcast_ss(pIn + 1*8 + 6);
ymm7 = _mm256_broadcast_ss(pIn + 1*8 + 7);
ymm0 = _mm256_mul_ps(ymm0, ymm8);
ymm1 = _mm256_mul_ps(ymm1, ymm9);
ymm0 = _mm256_add_ps(ymm0, ymm1);
ymm2 = _mm256_mul_ps(ymm2, ymm10);
ymm3 = _mm256_mul_ps(ymm3, ymm11);
ymm2 = _mm256_add_ps(ymm2, ymm3);
ymm4 = _mm256_mul_ps(ymm4, ymm12);
ymm5 = _mm256_mul_ps(ymm5, ymm13);
ymm4 = _mm256_add_ps(ymm4, ymm5);
ymm6 = _mm256_mul_ps(ymm6, ymm14);
ymm7 = _mm256_mul_ps(ymm7, ymm15);
ymm6 = _mm256_add_ps(ymm6, ymm7);
ymm0 = _mm256_add_ps(ymm0, ymm2);
ymm4 = _mm256_add_ps(ymm4, ymm6);
ymm0 = _mm256_add_ps(ymm0, ymm4);
_mm256_store_ps((float *) (pOut + 1*8), ymm0);

//Repeat using Matrix A Row 3
ymm0 = _mm256_broadcast_ss(pIn + 2*8);
ymm1 = _mm256_broadcast_ss(pIn + 2*8 + 1);
ymm2 = _mm256_broadcast_ss(pIn + 2*8 + 2);
ymm3 = _mm256_broadcast_ss(pIn + 2*8 + 3);
ymm4 = _mm256_broadcast_ss(pIn + 2*8 + 4);
ymm5 = _mm256_broadcast_ss(pIn + 2*8 + 5);
ymm6 = _mm256_broadcast_ss(pIn + 2*8 + 6);
ymm7 = _mm256_broadcast_ss(pIn + 2*8 + 7);
ymm0 = _mm256_mul_ps(ymm0, ymm8);
ymm1 = _mm256_mul_ps(ymm1, ymm9);
ymm0 = _mm256_add_ps(ymm0, ymm1);
ymm2 = _mm256_mul_ps(ymm2, ymm10);
ymm3 = _mm256_mul_ps(ymm3, ymm11);
```

```
ymm2 = _mm256_add_ps(ymm2, ymm3);
ymm4 = _mm256_mul_ps(ymm4, ymm12);
ymm5 = _mm256_mul_ps(ymm5, ymm13);
ymm4 = _mm256_add_ps(ymm4, ymm5);
ymm6 = _mm256_mul_ps(ymm6, ymm14);
ymm7 = _mm256_mul_ps(ymm7, ymm15);
ymm6 = _mm256_add_ps(ymm6, ymm7);
ymm0 = _mm256_add_ps(ymm0, ymm2);
ymm4 = _mm256_add_ps(ymm4, ymm6);
ymm0 = _mm256_add_ps(ymm0, ymm4);
_mm256_store_ps((float *) (pOut + 2*8), ymm0);

//Repeat using Matrix A Row 4
ymm0 = _mm256_broadcast_ss(pIn + 3*8);
ymm1 = _mm256_broadcast_ss(pIn + 3*8 + 1);
ymm2 = _mm256_broadcast_ss(pIn + 3*8 + 2);
ymm3 = _mm256_broadcast_ss(pIn + 3*8 + 3);
ymm4 = _mm256_broadcast_ss(pIn + 3*8 + 4);
ymm5 = _mm256_broadcast_ss(pIn + 3*8 + 5);
ymm6 = _mm256_broadcast_ss(pIn + 3*8 + 6);
ymm7 = _mm256_broadcast_ss(pIn + 3*8 + 7);
ymm0 = _mm256_mul_ps(ymm0, ymm8);
ymm1 = _mm256_mul_ps(ymm1, ymm9);
ymm0 = _mm256_add_ps(ymm0, ymm1);
ymm2 = _mm256_mul_ps(ymm2, ymm10);
ymm3 = _mm256_mul_ps(ymm3, ymm11);
ymm2 = _mm256_add_ps(ymm2, ymm3);
ymm4 = _mm256_mul_ps(ymm4, ymm12);
ymm5 = _mm256_mul_ps(ymm5, ymm13);
ymm4 = _mm256_add_ps(ymm4, ymm5);
ymm6 = _mm256_mul_ps(ymm6, ymm14);
ymm7 = _mm256_mul_ps(ymm7, ymm15);
ymm6 = _mm256_add_ps(ymm6, ymm7);
ymm0 = _mm256_add_ps(ymm0, ymm2);
ymm4 = _mm256_add_ps(ymm4, ymm6);
ymm0 = _mm256_add_ps(ymm0, ymm4);
_mm256_store_ps((float *) (pOut + 3*8), ymm0);

//Repeat using Matrix A Row 5
ymm0 = _mm256_broadcast_ss(pIn + 4*8);
ymm1 = _mm256_broadcast_ss(pIn + 4*8 + 1);
ymm2 = _mm256_broadcast_ss(pIn + 4*8 + 2);
ymm3 = _mm256_broadcast_ss(pIn + 4*8 + 3);
ymm4 = _mm256_broadcast_ss(pIn + 4*8 + 4);
ymm5 = _mm256_broadcast_ss(pIn + 4*8 + 5);
ymm6 = _mm256_broadcast_ss(pIn + 4*8 + 6);
ymm7 = _mm256_broadcast_ss(pIn + 4*8 + 7);
ymm0 = _mm256_mul_ps(ymm0, ymm8);
ymm1 = _mm256_mul_ps(ymm1, ymm9);
ymm0 = _mm256_add_ps(ymm0, ymm1);
ymm2 = _mm256_mul_ps(ymm2, ymm10);
ymm3 = _mm256_mul_ps(ymm3, ymm11);
ymm2 = _mm256_add_ps(ymm2, ymm3);
ymm4 = _mm256_mul_ps(ymm4, ymm12);
ymm5 = _mm256_mul_ps(ymm5, ymm13);
ymm4 = _mm256_add_ps(ymm4, ymm5);
ymm6 = _mm256_mul_ps(ymm6, ymm14);
ymm7 = _mm256_mul_ps(ymm7, ymm15);
ymm6 = _mm256_add_ps(ymm6, ymm7);
ymm0 = _mm256_add_ps(ymm0, ymm2);
ymm4 = _mm256_add_ps(ymm4, ymm6);
ymm0 = _mm256_add_ps(ymm0, ymm4);
_mm256_store_ps((float *) (pOut + 4*8), ymm0);

//Repeat using Matrix A Row 6
ymm0 = _mm256_broadcast_ss(pIn + 5*8);
ymm1 = _mm256_broadcast_ss(pIn + 5*8 + 1);
ymm2 = _mm256_broadcast_ss(pIn + 5*8 + 2);
ymm3 = _mm256_broadcast_ss(pIn + 5*8 + 3);
ymm4 = _mm256_broadcast_ss(pIn + 5*8 + 4);
ymm5 = _mm256_broadcast_ss(pIn + 5*8 + 5);
ymm6 = _mm256_broadcast_ss(pIn + 5*8 + 6);
ymm7 = _mm256_broadcast_ss(pIn + 5*8 + 7);
ymm0 = _mm256_mul_ps(ymm0, ymm8);
ymm1 = _mm256_mul_ps(ymm1, ymm9);
ymm0 = _mm256_add_ps(ymm0, ymm1);
ymm2 = _mm256_mul_ps(ymm2, ymm10);
ymm3 = _mm256_mul_ps(ymm3, ymm11);
ymm2 = _mm256_add_ps(ymm2, ymm3);
ymm4 = _mm256_mul_ps(ymm4, ymm12);
ymm5 = _mm256_mul_ps(ymm5, ymm13);
ymm4 = _mm256_add_ps(ymm4, ymm5);
ymm6 = _mm256_mul_ps(ymm6, ymm14);
ymm7 = _mm256_mul_ps(ymm7, ymm15);
ymm6 = _mm256_add_ps(ymm6, ymm7);
```

```
ymm0 = _mm256_add_ps(ymm0, ymm2);
ymm4 = _mm256_add_ps(ymm4, ymm6);
ymm0 = _mm256_add_ps(ymm0, ymm4);
_mm256_store_ps((float *) (pOut + 5*8), ymm0);

//Repeat using Matrix A Row 7
ymm0 = _mm256_broadcast_ss(pIn + 6*8);
ymm1 = _mm256_broadcast_ss(pIn + 6*8 + 1);
ymm2 = _mm256_broadcast_ss(pIn + 6*8 + 2);
ymm3 = _mm256_broadcast_ss(pIn + 6*8 + 3);
ymm4 = _mm256_broadcast_ss(pIn + 6*8 + 4);
ymm5 = _mm256_broadcast_ss(pIn + 6*8 + 5);
ymm6 = _mm256_broadcast_ss(pIn + 6*8 + 6);
ymm7 = _mm256_broadcast_ss(pIn + 6*8 + 7);
ymm0 = _mm256_mul_ps(ymm0, ymm8);
ymm1 = _mm256_mul_ps(ymm1, ymm9);
ymm0 = _mm256_add_ps(ymm0, ymm1);
ymm2 = _mm256_mul_ps(ymm2, ymm10);
ymm3 = _mm256_mul_ps(ymm3, ymm11);
ymm2 = _mm256_add_ps(ymm2, ymm3);
ymm4 = _mm256_mul_ps(ymm4, ymm12);
ymm5 = _mm256_mul_ps(ymm5, ymm13);
ymm4 = _mm256_add_ps(ymm4, ymm5);
ymm6 = _mm256_mul_ps(ymm6, ymm14);
ymm7 = _mm256_mul_ps(ymm7, ymm15);
ymm6 = _mm256_add_ps(ymm6, ymm7);
ymm0 = _mm256_add_ps(ymm0, ymm2);
ymm4 = _mm256_add_ps(ymm4, ymm6);
ymm0 = _mm256_add_ps(ymm0, ymm4);
_mm256_store_ps((float *) (pOut + 6*8), ymm0);

//Repeat using Matrix A Row 8
ymm0 = _mm256_broadcast_ss(pIn + 7*8);
ymm1 = _mm256_broadcast_ss(pIn + 7*8 + 1);
ymm2 = _mm256_broadcast_ss(pIn + 7*8 + 2);
ymm3 = _mm256_broadcast_ss(pIn + 7*8 + 3);
ymm4 = _mm256_broadcast_ss(pIn + 7*8 + 4);
ymm5 = _mm256_broadcast_ss(pIn + 7*8 + 5);
ymm6 = _mm256_broadcast_ss(pIn + 7*8 + 6);
ymm7 = _mm256_broadcast_ss(pIn + 7*8 + 7);
ymm0 = _mm256_mul_ps(ymm0, ymm8);
ymm1 = _mm256_mul_ps(ymm1, ymm9);
ymm0 = _mm256_add_ps(ymm0, ymm1);
ymm2 = _mm256_mul_ps(ymm2, ymm10);
ymm3 = _mm256_mul_ps(ymm3, ymm11);
ymm2 = _mm256_add_ps(ymm2, ymm3);
ymm4 = _mm256_mul_ps(ymm4, ymm12);
ymm5 = _mm256_mul_ps(ymm5, ymm13);
ymm4 = _mm256_add_ps(ymm4, ymm5);
ymm6 = _mm256_mul_ps(ymm6, ymm14);
ymm7 = _mm256_mul_ps(ymm7, ymm15);
ymm6 = _mm256_add_ps(ymm6, ymm7);
ymm0 = _mm256_add_ps(ymm0, ymm2);
ymm4 = _mm256_add_ps(ymm4, ymm6);
ymm0 = _mm256_add_ps(ymm0, ymm4);
_mm256_store_ps((float *) (pOut + 7*8), ymm0);









<h1 class="sectionHeading">Determinant of a Matrix</h1>


// The following code computes the determinant value of eight 4x4 matrices of single-precision floating point values in parallel.


```

- collapse sourceview plaincopy to clipboardprint?

```
#define MAT_WIDTH 4
#define MAT_SIZE 12
__m256    ymm_a11, ymm_a12, ymm_a13, ymm_a14;
__m256   ymm_a21, ymm_a22, ymm_a23, ymm_a24;
__m256   ymm_a31, ymm_a32, ymm_a33, ymm_a34;
__m256   ymm_a41, ymm_a42, ymm_a43, ymm_a44;
__m256   ymm0, ymm1, ymm2, ymm3, ymm4, ymm5, ymm6, ymm7;
__m256   ymma, ymmb, ymmc, ymmd, ymme, ymmf, ymmg, ymmh;
float * pIn = (float *) InputBuffer;
float * pOut = (float *) OutputBuffer;

for (unsigned int k = 0; k < lc; k++)
{
 // Read first 2 rows of single precision floats from 8
// 4x4 matrices into 8 ymm registers
  ymm0 = _mm256_load_ps((float *) (pIn + (0*MAT_SIZE))); //matrix 1
ymm1 = _mm256_load_ps((float *) (pIn + (1*MAT_SIZE))); //matrix 2
ymm2 = _mm256_load_ps((float *) (pIn + (2*MAT_SIZE))); //matrix 3
 ymm3 = _mm256_load_ps((float *) (pIn + (3*MAT_SIZE))); //matrix 4
 ymm4 = _mm256_load_ps((float *) (pIn + (4*MAT_SIZE))); //matrix 5
 ymm5 = _mm256_load_ps((float *) (pIn + (5*MAT_SIZE))); //matrix 6
 ymm6 = _mm256_load_ps((float *) (pIn + (6*MAT_SIZE))); //matrix 7
  ymm7 = _mm256_load_ps((float *) (pIn + (7*MAT_SIZE))); //matrix 8

  //Begin the transpose
 ymma = _mm256_unpacklo_ps(ymm0, ymm1); //11,11,12,12,21,21,22,22
  ymmb = _mm256_unpackhi_ps(ymm0, ymm1); //13,13,14,14,23,23,24,24
  ymmc = _mm256_unpacklo_ps(ymm2, ymm3); //11,11,12,12,21,21,22,22
  ymmd = _mm256_unpackhi_ps(ymm2, ymm3); //13,13,14,14,23,23,24,24
  ymme = _mm256_unpacklo_ps(ymm4, ymm5); //11,11,12,12,21,21,22,22
  ymmf = _mm256_unpackhi_ps(ymm4, ymm5); //13,13,14,14,23,23,24,24
  ymmg = _mm256_unpacklo_ps(ymm6, ymm7); //11,11,12,12,21,21,22,22
  ymmh = _mm256_unpackhi_ps(ymm6, ymm7); //13,13,14,14,23,23,24,24

  //Create output rows 0, 1, 4 and 5
//12, 12, 11, 11, 22, 22, 21, 21
  ymm7 = _mm256_shuffle_ps(ymma, ymmc, 0x4e);
  //11, 11, 11, 11, 21, 21, 21, 21    (lower half of rows 0 and 4)
  ymm5 = _mm256_blend_ps(ymm7, ymma, 0x33);
  //12, 12, 12, 12, 22, 22, 22, 22    (lower half of rows 1 and 5)
  ymm3 = _mm256_blend_ps(ymm7, ymmc, 0xcc);
  //12, 12, 11, 11, 22, 22, 21, 21
  ymm6 = _mm256_shuffle_ps(ymme, ymmg, 0x4e);
//11, 11, 11, 11, 21, 21, 21, 21    (upper half of rows 0 and 4)
  ymm4 = _mm256_blend_ps(ymm6, ymme, 0x33);
 //12, 12, 12, 12, 22, 22, 22, 22    (upper half of rows 1 and 5)
  ymm2 = _mm256_blend_ps(ymm6, ymmg, 0xcc);

  //Last step to create rows 0, 1, 4, and 5
 //Gather a11 from 8 matrices: 11,11,11,11,11,11,11,11   (Row 0)
ymm_a11 = _mm256_permute2f128_ps(ymm5, ymm4, 0x20);
  //Gather a12 from 8 matrices: 12,12,12,12,12,12,12,12   (Row 1)
ymm_a12 = _mm256_permute2f128_ps(ymm3, ymm2, 0x20);
//Gather a21 from 8 matrices: 21,21,21,21,21,21,21,21   (Row 4)
  ymm_a21 = _mm256_permute2f128_ps(ymm5, ymm4, 0x31);
//Gather a22 from 8 matrices: 22,22,22,22,22,22,22,22   (Row 5)
  ymm_a22 = _mm256_permute2f128_ps(ymm3, ymm2, 0x31);

  //Create output rows 2, 3, 6 and 7
//14, 14, 13, 13, 24, 24, 23, 23
  ymm7 = _mm256_shuffle_ps(ymmb, ymmd, 0x4e);
//13, 13, 13, 13, 23, 23, 23, 23    (lower half of rows 2 and 6)
  ymm5 = _mm256_blend_ps(ymm7, ymmb, 0x33);
 //14, 14, 14, 14, 24, 24, 24, 24    (lower half of rows 3 and 7)
  ymm3 = _mm256_blend_ps(ymm7, ymmd, 0xcc);
  //14, 14, 13, 13, 24, 24, 23, 23
  ymm6 = _mm256_shuffle_ps(ymmf, ymmh, 0x4e);
//13, 13, 13, 13, 23, 23, 23, 23    (upper half of rows 2 and 6)
  ymm4 = _mm256_blend_ps(ymm6, ymmf, 0x33);
  //14, 14, 14, 14, 24, 24, 24, 24    (upper half of rows 3 and 7)
  ymm2 = _mm256_blend_ps(ymm6, ymmh, 0xcc);

  //Last step to create rows 2, 3, 6, and 7
 //Gather a13 from 8 matrices: 13,13,13,13,13,13,13,13 (Row 2)
 ymm_a13 = _mm256_permute2f128_ps(ymm5, ymm4, 0x20);
//Gather a14 from 8 matrices: 14,14,14,14,14,14,14,14 (Row 3)
 ymm_a14 = _mm256_permute2f128_ps(ymm3, ymm2, 0x20);
//Gather a23 from 8 matrices: 23,23,23,23,23,23,23,23 (Row 6)
 ymm_a23 = _mm256_permute2f128_ps(ymm5, ymm4, 0x31);
//Gather a24 from 8 matrices: 24,24,24,24,24,24,24,24 (Row 7)
 ymm_a24 = _mm256_permute2f128_ps(ymm3, ymm2, 0x31);


  // Read rows 3 & 4 of single precision floats from 8
 // 4x4 matrices into 8 ymm registers
  ymm0 = _mm256_load_ps((float *)(pIn+(0*MAT_SIZE)+(2*MAT_WIDTH)));
 ymm1 = _mm256_load_ps((float *)(pIn+(1*MAT_SIZE)+(2*MAT_WIDTH)));
  ymm2 = _mm256_load_ps((float *)(pIn+(2*MAT_SIZE)+(2*MAT_WIDTH)));
  ymm3 = _mm256_load_ps((float *)(pIn+(3*MAT_SIZE)+(2*MAT_WIDTH)));
  ymm4 = _mm256_load_ps((float *)(pIn+(4*MAT_SIZE)+(2*MAT_WIDTH)));
  ymm5 = _mm256_load_ps((float *)(pIn+(5*MAT_SIZE)+(2*MAT_WIDTH)));
  ymm6 = _mm256_load_ps((float *)(pIn+(6*MAT_SIZE)+(2*MAT_WIDTH)));
  ymm7 = _mm256_load_ps((float *)(pIn+(7*MAT_SIZE)+(2*MAT_WIDTH)));

  //Begin the transpose
 ymma = _mm256_unpacklo_ps(ymm0, ymm1); //31,31,32,32,41,41,42,42
ymmb = _mm256_unpackhi_ps(ymm0, ymm1); //33,33,34,34,43,43,44,44
  ymmc = _mm256_unpacklo_ps(ymm2, ymm3); //31,31,32,32,41,41,42,42
  ymmd = _mm256_unpackhi_ps(ymm2, ymm3); //33,33,34,34,43,43,44,44
```

```
ymme = _mm256_unpacklo_ps(ymm4, ymm5); //31,31,32,32,41,41,42,42
ymmf = _mm256_unpackhi_ps(ymm4, ymm5); //33,33,34,34,43,43,44,44
ymmg = _mm256_unpacklo_ps(ymm6, ymm7); //31,31,32,32,41,41,42,42
ymmh = _mm256_unpackhi_ps(ymm6, ymm7); //33,33,34,34,43,43,44,44

//Create output rows 0, 1, 4 and 5
//32, 32, 31, 31, 42, 42, 41, 41
ymm7 = _mm256_shuffle_ps(ymma, ymmc, 0x4e);
//31, 31, 31, 31, 41, 41, 41, 41    (lower half of rows 0 and 4)
ymm5 = _mm256_blend_ps(ymm7, ymma, 0x33);
//32, 32, 32, 32, 42, 42, 42, 42    (lower half of rows 1 and 5)
ymm3 = _mm256_blend_ps(ymm7, ymmc, 0xcc);
//32, 32, 31, 31, 42, 42, 41, 41
ymm6 = _mm256_shuffle_ps(ymme, ymmg, 0x4e);
//31, 31, 31, 31, 41, 41, 41, 41    (upper half of rows 0 and 4)
ymm4 = _mm256_blend_ps(ymm6, ymme, 0x33);
//32, 32, 32, 32, 42, 42, 42, 42    (upper half of rows 1 and 5)
ymm2 = _mm256_blend_ps(ymm6, ymmg, 0xcc);

//Last step to create rows 0, 1, 4, and 5
//Gather a31 from 8 matrices: 31,31,31,31,31,31,31,31   (Row 0)
ymm_a31 = _mm256_permute2f128_ps(ymm5, ymm4, 0x20);
//Gather a32 from 8 matrices: 32,32,32,32,32,32,32,32   (Row 1)
ymm_a32 = _mm256_permute2f128_ps(ymm3, ymm2, 0x20);
//Gather a41 from 8 matrices: 41,41,41,41,41,41,41,41   (Row 4)
ymm_a41 = _mm256_permute2f128_ps(ymm5, ymm4, 0x31);
//Gather a42 from 8 matrices: 42,42,42,42,42,42,42,42 (Row 5)
ymm_a42 = _mm256_permute2f128_ps(ymm3, ymm2, 0x31);

//Create output rows 2, 3, 6 and 7
//34, 34, 33, 33, 44, 44, 43, 43
ymm7 = _mm256_shuffle_ps(ymmb, ymmd, 0x4e);
//33, 33, 33, 33, 43, 43, 43, 43    (lower half of rows 2 and 6)
ymm5 = _mm256_blend_ps(ymm7, ymmb, 0x33);
//34, 34, 34, 34, 44, 44, 44, 44    (lower half of rows 3 and 7)
ymm3 = _mm256_blend_ps(ymm7, ymmd, 0xcc);
//34, 34, 33, 33, 44, 44, 43, 43
ymm6 = _mm256_shuffle_ps(ymmf, ymmh, 0x4e);
//33, 33, 33, 33, 43, 43, 43, 43    (upper half of rows 2 and 6)
ymm4 = _mm256_blend_ps(ymm6, ymmf, 0x33);
//34, 34, 34, 34, 44, 44, 44, 44    (upper half of rows 3 and 7)
ymm2 = _mm256_blend_ps(ymm6, ymmh, 0xcc);

//Last step to create rows 2, 3, 6, and 7
//Gather a33 from 8 matrices: 33,33,33,33,33,33,33,33   (Row 2)
ymm_a33 = _mm256_permute2f128_ps(ymm5, ymm4, 0x20);
//Gather a34 from 8 matrices: 34,34,34,34,34,34,34,34 (Row 3)
ymm_a34 = _mm256_permute2f128_ps(ymm3, ymm2, 0x20);
//Gather a43 from 8 matrices: 43,43,43,43,43,43,43,43 (Row 6)
ymm_a43 = _mm256_permute2f128_ps(ymm5, ymm4, 0x31);
//Gather a44 from 8 matrices: 44,44,44,44,44,44,44,44   (Row 7)
ymm_a44 = _mm256_permute2f128_ps(ymm3, ymm2, 0x31);

// ******Determinants for a11*****************
__m256 ymm_a11a22, ymm_a33a44, ymm_a43a34, ymm_a11a23;
__m256 ymm_a32a44, ymm_a34a42, ymm_a11a24, ymm_a32a43;
__m256 ymm_a42a33;
__m256 ymm_a11a22_D, ymm_a11a23_D, ymm_a11a24_D, ymm_a11_D;
__m256 ymm_a33a44_a43a34, ymm_a32a44_a34a42, ymm_a32a43_a42a33;

// calculate a11a22(a33a44 - a43a34) for 8 matrices
ymm_a11a22 = _mm256_mul_ps(ymm_a11, ymm_a22);
ymm_a33a44 = _mm256_mul_ps(ymm_a33, ymm_a44);
ymm_a43a34 = _mm256_mul_ps(ymm_a43, ymm_a34);
ymm_a33a44_a43a34 = _mm256_sub_ps(ymm_a33a44, ymm_a43a34);
ymm_a11a22_D = _mm256_mul_ps(ymm_a11a22, ymm_a33a44_a43a34);

// calculate a11a23(a32a44 - a34a42) for 8 matrices
ymm_a11a23 = _mm256_mul_ps(ymm_a11, ymm_a23);
ymm_a32a44 = _mm256_mul_ps(ymm_a32, ymm_a44);
ymm_a34a42 = _mm256_mul_ps(ymm_a34, ymm_a42);
ymm_a32a44_a34a42 = _mm256_sub_ps(ymm_a32a44, ymm_a34a42);
ymm_a11a23_D = _mm256_mul_ps(ymm_a11a23, ymm_a32a44_a34a42);

// calculate a11a24(a32a43 - a42a33) for 8 matrices
ymm_a11a24 = _mm256_mul_ps(ymm_a11, ymm_a24);
ymm_a32a43 = _mm256_mul_ps(ymm_a32, ymm_a43);
ymm_a42a33 = _mm256_mul_ps(ymm_a42, ymm_a33);
ymm_a32a43_a42a33 = _mm256_sub_ps(ymm_a32a43, ymm_a42a33);
ymm_a11a24_D = _mm256_mul_ps(ymm_a11a24, ymm_a32a43_a42a33);

// calculate partial determinant for 8 matrices:
// (a11a22(a33a44 - a43a34) + a11a23(a32a44 - a34a42) +
//  a11a24(a32a43 - a42a33))
ymm_a11a22_D = _mm256_sub_ps(ymm_a11a22_D, ymm_a11a23_D);
```

```
  ymm_a11_D = _mm256_add_ps(ymm_a11a22_, ymm_a11a24_D);

   // ******Determinants for a12*****************//
    __m256 ymm_a12a21, ymm_a12a23, ymm_a31a44, ymm_a41a34,
 __m256 ymm_a12a24, ymm_a31a43, ymm_a41a33;
  __m256 ymm_a12a21_D, ymm_a12a23_D, ymm_a12a24_D, ymm_a12_D;
 __m256 ymm_a31a44_a41a34, ymm_a31a43_a41a33;

    // calculate a12a21(a33a44 - a43a34) for 8 matrices
 ymm_a12a21 = _mm256_mul_ps(ymm_a12, ymm_a21);
   ymm_a12a21_D = _mm256_mul_ps(ymm_a12a21, ymm_a33a44_a43a34);

    // calculate a12a23(a31a44 - a41a34) for 8 matrices
 ymm_a12a23 = _mm256_mul_ps(ymm_a12, ymm_a23);
   ymm_a31a44 = _mm256_mul_ps(ymm_a31, ymm_a44);
   ymm_a41a34 = _mm256_mul_ps(ymm_a41, ymm_a34);
   ymm_a31a44_a41a34 = _mm256_sub_ps(ymm_a31a44, ymm_a41a34);
  ymm_a12a23_D = _mm256_mul_ps(ymm_a12a23, ymm_a31a44_a41a34);

    // calculate a12a24(a31a43 - a41a33) for 8 matrices
 ymm_a12a24 = _mm256_mul_ps(ymm_a12, ymm_a24);
   ymm_a31a43 = _mm256_mul_ps(ymm_a31, ymm_a43);
   ymm_a41a33 = _mm256_mul_ps(ymm_a41, ymm_a33);
   ymm_a31a43_a41a33 = _mm256_sub_ps(ymm_a31a43, ymm_a41a33);
  ymm_a12a24_D = _mm256_mul_ps(ymm_a12a24, ymm_a31a43_a41a33);

   // calculate partial determinant for 8 matrices:
   // (a12a21(a33a44 - a43a34) + a12a23(a31a44 - a41a34)
   // + a12a24(a31a43 - a41a33))
  ymm_a12a21_D = _mm256_sub_ps(ymm_a12a23_D, ymm_a12a21_D);
   ymm_a12_D = _mm256_sub_ps(ymm_a12a21_D, ymm_a12a24_D);

 // ******Determinants for a13*****************//
    __m256 ymm_a13a21, ymm_a13a22, ymm_a13a24, ymm_a31a42;
  __m256  ymm_a41a32;
 __m256 ymm_a13a21_D, ymm_a13a22_D, ymm_a13a24_D, ymm_a13_D;
 __m256 ymm_a31a42_a41a32;

   // calculate a13a21(a32a44 - a42a34) for 8 matrices
 ymm_a13a21 = _mm256_mul_ps(ymm_a13, ymm_a21);
   ymm_a13a21_D = _mm256_mul_ps(ymm_a13a21, ymm_a32a44_a34a42);

    // calculate a13a22(a31a43 - a41a33) for 8 matrices
 ymm_a13a22 = _mm256_mul_ps(ymm_a13, ymm_a22);
   ymm_a13a22_D = _mm256_mul_ps(ymm_a13a22, ymm_a31a43_a41a33);

    // calculate a13a24(a31a42 - a41a32) for 8 matrices
 ymm_a13a24 = _mm256_mul_ps(ymm_a13, ymm_a24);
   ymm_a31a42 = _mm256_mul_ps(ymm_a31, ymm_a42);
   ymm_a41a32 = _mm256_mul_ps(ymm_a41, ymm_a32);
   ymm_a31a42_a41a32 = _mm256_sub_ps(ymm_a31a42, ymm_a41a32);
  ymm_a13a24_D = _mm256_mul_ps(ymm_a13a24, ymm_a31a42_a41a32);

   // calculate partial determinant for 8 matrices:
   // (a13a21(a32a44 - a42a34) + a13a22(a31a43 - a41a33)
   // + a13a24(a31a42 - a41a32))
  ymm_a13a21_D = _mm256_sub_ps(ymm_a13a21_D, ymm_a13a22_D);
   ymm_a13_D = _mm256_add_ps(ymm_a13a21_D, ymm_a13a24_D);

 // ******Determinants for a14*****************//
    __m256 ymm_a14a21, ymm_a14a22, ymm_a14a23;
  __m256 ymm_a14a21_D, ymm_a14a22_D, ymm_a14a23_D, ymm_a14_D;

 // calculate a14a21(a32a43 - a42a33) for 8 matrices
 ymm_a14a21 = _mm256_mul_ps(ymm_a14, ymm_a21);
   ymm_a14a21_D = _mm256_mul_ps(ymm_a14a21, ymm_a32a43_a42a33);

    // calculate a14a22(a31a43 - a41a33) for 8 matrices
 ymm_a14a22 = _mm256_mul_ps(ymm_a14, ymm_a22);
   ymm_a14a22_D = _mm256_mul_ps(ymm_a14a22, ymm_a31a43_a41a33);

    // calculate a14a23(a31a42 - a41a32) for 8 matrices
 ymm_a14a23 = _mm256_mul_ps(ymm_a14, ymm_a23);
   ymm_a14a23_D = _mm256_mul_ps(ymm_a14a23, ymm_a31a42_a41a32);

    // calculate partial determinant for 8 matrices:
   // (a14a21(a32a43 - a42a33)) + a14a22(a31a43 - a41a33)
  // + a14a23(a31a42 - a41a32))
   ymm_a14a21_D = _mm256_sub_ps(ymm_a14a22_D, ymm_a14a21_D);
  ymm_a14_D = _mm256_sub_ps(ymm_a14a21_D, ymm_a14a23_D);

 // Calculate final Determinant value for 8 matrices: addition of
   // determinants for a11, a12, a13, a14
  ymm_a11_D = _mm256_add_ps(ymm_a11_D, ymm_a12_D);
    ymm_a13_D = _mm256_add_ps(ymm_a13_D, ymm_a14_D);
    ymm_a11_D = _mm256_add_ps(ymm_a11_D, ymm_a13_D);

   // Need to scatter/extract the individual matrix D values to dest
  _mm256_store_ps((float *)pOut, ymm_a11_D);
}
```

About the Authors

Pallavi Mehrotra - Pallavi joined Intel as a Senior Software Engineer in 2006 after working as a CDMA Network Infrasturcture software developer at Motorola. She is currently member of the SSG Apple enabling team, working on optimizing Mac OS X applications for power and performance. Pallavi holds a Masters degree in Computer Science from Arizona State University and Bachelors in Electrical Engineering from India.

Richard Hubbard – Richard is a Senior Software Engineer and member of the SSG Apple enabling team, working on optimizing Mac OS X applications for power and performance. Richard holds a Masters degree in Electrical Engineering from Stevens Institute of Technology and Bachelors in Computer Engineering from New Jersey Institute of Technology.

***Trademarks**

©Intel Corporation