MATRIX OPERATIONS WITH AVX2 INSTRUCTIONS
Connor Baker, June 2017

1 Creating the Matrix

Consider the following line of C++:

```
std::vector<__m256> v(4, _mm256_setr_ps(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0));
```

this statement creates a matrix (of sorts) named v, of four rows and eight columns, where each row is an Intel AVX2 (Advanced Vector Extensions 2) datatype.

$$\text{v} = \begin{bmatrix} 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \end{bmatrix}$$

We can access any element of this matrix just like we would with a two dimensional array or vector, so the above corresponds to:

$$\text{v} = \begin{bmatrix} v[0][0] & v[0][1] & v[0][2] & v[0][3] & v[0][4] & v[0][5] & v[0][6] & v[0][7] \\ v[1][0] & v[1][1] & v[1][2] & v[1][3] & v[1][4] & v[1][5] & v[1][6] & v[1][7] \\ v[2][0] & v[2][1] & v[2][2] & v[2][3] & v[2][4] & v[2][5] & v[2][6] & v[2][7] \\ v[3][0] & v[3][1] & v[3][2] & v[3][3] & v[3][4] & v[3][5] & v[3][6] & v[3][7] \end{bmatrix}$$

and as such, the elements are addressable. If we wanted to change every value of six to an 11, we can do so:

```
for (size_t i = 0; i < 4; i++) {
  v[i][6] = 11.0;
}
```

1.1 A Little More on AVX

Intel's AVX has several datatypes of note (see next page), but for the purposes of this discussion, we will talk exclusively about the __m256 datatype, and its applications to matrix operations.

If you need an overview of how to use AVX, I highly recommend that you read Matt Scarpino's excellent article Crunching Numbers with AVX and AVX2 on Code Project.

1.2 Why Use AVX?

The benefit of using an Intel AVX2 data type is that we are then allowed to use intrinsic functions, which are essentially calls to highly optimized assembly instructions from within high-level languages. In doing this, we retain the readability and ease of use that makes high level langauges attractive, while being able to write highly-performant code in critical areas of programs.

Type	Meaning
`__m256`	256-bit as eight single-precision floating-point values, representing a YMM register or memory location
`__m256d`	256-bit as four double-precision floating-point values, representing a YMM register or memory location
`__m256i`	256-bit as integers, (bytes, words, etc.)
`__m128`	128-bit single precision floating-point (32 bits each)
`__m128d`	128-bit double precision floating-point (64 bits each)

Figure 1: Intel AVX Data Types, retrived from Introduction to IntelÂ® Advanced Vector Extensions

2 It's All About the Size (of the Row)

Returning to our earlier code sample,

```
std::vector<__m256> v(4, _mm256_setr_ps(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0));
```

we see an issue in terms of the size of the matrix that we can generate. Our matrix is stuck at a size of $n \times 8$, since the AVX vector holds eight floats. But wait! Consider the following snippet:

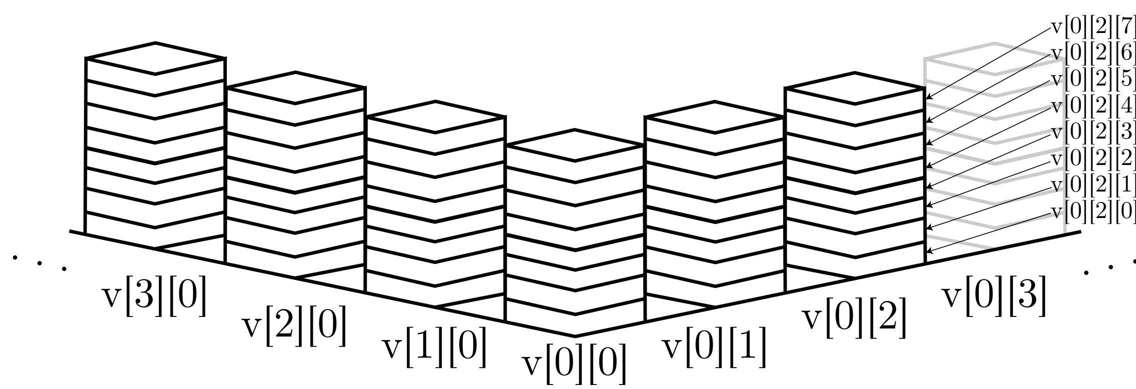
```
std::vector<std::vector<__m256>> v(SIZE, std::vector<__m256>(4,
↪  _mm256_setr_ps(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)));

for (size_t i = 0; i < v.size(); i++) {
  for (size_t j = 0; j < v[0].size(); j++) {
      for (size_t k = 0; k < 8; k++) {
          std::cout << v[i][j][k] << ' ';
      }
      std::cout << '\t';
  }
  std::cout << std::endl;
}
```

the output of which is below.

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

By using a `std::vector` of `std::vector` of Intel's AVX `__m256` vector data type, we can create any matrix of size $n \times n * 8$ (since the `__m256` stores a vector of eight floats). With this obstacle now behind us, we are free to persue arithmetic operations on these matrices, backed by the performance inherent in the use of intrinsics.



3 Matrix Operations (when Dimensions are 'Nice')

3.1 Addition and Subtraction of Matrices

Suppose we have to matrices that we want to add:

$$A = \begin{bmatrix} 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \end{bmatrix}, \quad B = \begin{bmatrix} 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \end{bmatrix}.$$

We can quite easily implement this using intrinsics. Consider the following `C++` program.

```cpp
#include <x86intrin.h>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using std::vector;

/*
 *  Takes two vectors as arguments and sums their contents.
 *  Precondition: The matrices are of the same size.
 *  Postcondition: The function returns a matrix of the size of the inputs.
 */
vector<vector<__m256>> add_mat(vector<vector<__m256>> a, vector<vector<__m256>> b) {
    // Initialize the matrix to return, with the same row and column size as input
    vector<vector<__m256>> c(a.size(), vector<__m256>(a[0].size(), _mm256_set1_ps(0.0)));

    // Sum the the elements of the matrices
    for (size_t i = 0; i < a.size(); i++) {
        for (size_t j = 0; j < a[0].size(); j++) {
            c[i][j] = _mm256_add_ps(a[i][j], b[i][j]);
        }
    }
    return c;
}

int main() {
    // Initialize the two matrices to sum
```

```
29      vector<vector<__m256>> mat_A(8, vector<__m256>(1, _mm256_setr_ps(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)));
30      vector<vector<__m256>> mat_B(8, vector<__m256>(1, _mm256_setr_ps(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)));
31
32      // Pass the sum to a third matrix
33      vector<vector<__m256>> mat_C = add_mat(mat_A, mat_B);
34
35      // Print the result
36      for (size_t i = 0; i < mat_C.size(); i++) {
37          for (size_t j = 0; j < mat_C[0].size(); j++) {
38              for (size_t k = 0; k < 8; k++) {
39                  cout << mat_C[i][j][k] << ' ';
40              }
41              cout << '\t';
42          }
43          cout << endl;
44      }
45
46      return 0;
47  }
```

As expected, it outputs the sum, as shown below:

$$C = \begin{bmatrix} 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 10.0 & 12.0 & 14.0 \\ 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 10.0 & 12.0 & 14.0 \\ 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 10.0 & 12.0 & 14.0 \\ 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 10.0 & 12.0 & 14.0 \\ 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 10.0 & 12.0 & 14.0 \\ 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 10.0 & 12.0 & 14.0 \\ 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 10.0 & 12.0 & 14.0 \\ 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 10.0 & 12.0 & 14.0 \end{bmatrix}.$$

If we wanted to change this program so that it subtracts the matrices instead of add, one would change Line 40 to:

```
c[i][j] = _mm256_sub_ps(a[i][j], b[i][j]);
```

3.2 Multiplication of Matrices

Suppose we have to matrices that we want to multiply:

$$A = \begin{bmatrix} 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \end{bmatrix}, \quad B = \begin{bmatrix} 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \end{bmatrix}.$$

With slight modifications to our previous program, we can attempt to take the product.

```
#include <x86intrin.h>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using std::vector;

/*
 *  Takes two vectors as arguments and multiples them.
 *  Precondition: The matrices are of the same size.
 *  Postcondition: The function returns a matrix of the size of the inputs.
 */
vector<vector<__m256>> mult_mat(vector<vector<__m256>> a, vector<vector<__m256>> b) {
    // Initialize the matrix to return, with the same row and column size as input
    vector<vector<__m256>> c(a.size(), vector<__m256>(a[0].size(), _mm256_set1_ps(0.0)));

    // Sum the the elements of the matrices
    for (size_t i = 0; i < a.size(); i++) {
        for (size_t j = 0; j < a[0].size(); j++) {
            c[i][j] = _mm256_mul_ps(a[i][j], b[i][j]);
        }
    }
    return c;
}

int main() {
    // Initialize the two matrices to multiply
    vector<vector<__m256>> mat_A(8, vector<__m256>(1, _mm256_setr_ps(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)));
    vector<vector<__m256>> mat_B(8, vector<__m256>(1, _mm256_setr_ps(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)));

    // Pass the product to a third matrix
    vector<vector<__m256>> mat_C = mult_mat(mat_A, mat_B);

    // Print the result
    for (size_t i = 0; i < mat_C.size(); i++) {
        for (size_t j = 0; j < mat_C[0].size(); j++) {
            for (size_t k = 0; k < 8; k++) {
                cout << mat_C[i][j][k] << ' ';
            }
```

```
            cout << '\t';
        }
        cout << endl;
    }

    return 0;
}
```

This outputs the product:

$$C = \begin{bmatrix} 0.0 & 1.0 & 4.0 & 9.0 & 16.0 & 25.0 & 36.0 & 49.0 \\ 0.0 & 1.0 & 4.0 & 9.0 & 16.0 & 25.0 & 36.0 & 49.0 \\ 0.0 & 1.0 & 4.0 & 9.0 & 16.0 & 25.0 & 36.0 & 49.0 \\ 0.0 & 1.0 & 4.0 & 9.0 & 16.0 & 25.0 & 36.0 & 49.0 \\ 0.0 & 1.0 & 4.0 & 9.0 & 16.0 & 25.0 & 36.0 & 49.0 \\ 0.0 & 1.0 & 4.0 & 9.0 & 16.0 & 25.0 & 36.0 & 49.0 \\ 0.0 & 1.0 & 4.0 & 9.0 & 16.0 & 25.0 & 36.0 & 49.0 \\ 0.0 & 1.0 & 4.0 & 9.0 & 16.0 & 25.0 & 36.0 & 49.0 \end{bmatrix}.$$

However, this isn't correct! The product should be:

$$C = \begin{bmatrix} 0.0 & 28.0 & 56.0 & 84.0 & 112.0 & 140.0 & 168.0 & 196.0 \\ 0.0 & 28.0 & 56.0 & 84.0 & 112.0 & 140.0 & 168.0 & 196.0 \\ 0.0 & 28.0 & 56.0 & 84.0 & 112.0 & 140.0 & 168.0 & 196.0 \\ 0.0 & 28.0 & 56.0 & 84.0 & 112.0 & 140.0 & 168.0 & 196.0 \\ 0.0 & 28.0 & 56.0 & 84.0 & 112.0 & 140.0 & 168.0 & 196.0 \\ 0.0 & 28.0 & 56.0 & 84.0 & 112.0 & 140.0 & 168.0 & 196.0 \\ 0.0 & 28.0 & 56.0 & 84.0 & 112.0 & 140.0 & 168.0 & 196.0 \\ 0.0 & 28.0 & 56.0 & 84.0 & 112.0 & 140.0 & 168.0 & 196.0 \end{bmatrix}.$$

There are two choices for trying to multiply these matrices:

1. Change the algorithm so that we no longer use the intrinsic.
2. Change the first matrix to B^{T} so that we can still make use of the `_mm256_mul_ps` intrinsic function.

Both of these options have a performance penalty: either dropping the use of intrinsic functions, or taking a hit and transposing B. Since the purpose of this discussion is to use intrinsics, we will take the second option.

Below is the fixed (really stupid and hacky, which works only for this single example and not all arrays in general) code, which prints the correct output that was given above:

```
#include <x86intrin.h>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
```

```cpp
using std::vector;

/*
 *  Takes two vectors as arguments and multiples them.
 *  Precondition: The matrices are of the same size.
 *  Postcondition: The function returns a matrix of the size of the inputs.
 */
vector<vector<__m256>> mult_mat(vector<vector<__m256>> a, vector<vector<__m256>> b) {
    // Initialize the matrix to return, with the same row and column size as input
    vector<vector<__m256>> c(a.size(), vector<__m256>(a[0].size(), _mm256_set1_ps(0.0)));
    vector<vector<__m256>> c_temp(a.size(), vector<__m256>(a[0].size(), _mm256_set1_ps(0.0)));

    // Create our transpose of the second matrix
    vector<vector<__m256>> b_new(b.size(), vector<__m256>(b[0].size(), _mm256_set1_ps(0.0)));

    // Take the transpose of the second matrix
    for (size_t i = 0; i < a.size(); i++) {
        for (size_t j = 0; j < a[0].size(); j++) {
            for (size_t k = 0; k < 8; k++) {
                b_new[i][j][k] = b[k][j][i];
            }
        }
    }

    // Compute the product of the two matrices
    for (size_t i = 0; i < c.size(); i++) {
        for (size_t j = 0; j < c[0].size(); j++) {
            for (size_t k = 0; k < 8; k++) {
                c_temp[i][j] = _mm256_dp_ps(a[i][j], b_new[i][j], 0xFF);
                c[i][j][k] = c_temp[i][j][3] + c_temp[i][j][4];
            }
        }
    }

    c_temp = c;

    // Take the transpose of the product
    for (size_t i = 0; i < c.size(); i++) {
        for (size_t j = 0; j < c[0].size(); j++) {
            for (size_t k = 0; k < 8; k++) {
                c[i][j][k] = c_temp[k][j][i];
```

```
            }
        }
    }

    return c;
}

int main() {
    // Initialize the two matrices to multiply
    vector<vector<__m256>> mat_A(8, vector<__m256>(1, _mm256_setr_ps(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)));
    vector<vector<__m256>> mat_B(8, vector<__m256>(1, _mm256_setr_ps(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)));

    // Pass the product to a third matrix
    vector<vector<__m256>> mat_C = mult_mat(mat_A, mat_B);

    // Print the result
    for (size_t i = 0; i < mat_C.size(); i++) {
        for (size_t j = 0; j < mat_C[0].size(); j++) {
            for (size_t k = 0; k < 8; k++) {
                cout << mat_C[i][j][k] << ' ';
            }
            cout << '\t';
        }
        cout << endl;
    }

    return 0;
}
```

$$
\begin{bmatrix} v_{000} & \cdots & v_{007} & \vert & \cdots & \vert & v_{0m0} & \cdots & v_{0m7} \\ & \vdots & & & \vdots & & & \vdots & \\ v_{n00} & \cdots & v_{n07} & \vert & \cdots & \vert & v_{nm0} & \cdots & v_{nm7} \end{bmatrix}^T
$$

$$
\begin{bmatrix} v_{000} & \cdots & v_{700} & \vert & \cdots & \vert & v_{(n-8)00} & \cdots & v_{n00} \\ & \vdots & & & \vdots & & & \vdots & \\ v_{0m0} & \cdots & v_{0m7} & \vert & \cdots & \vert & v_{nm0} & \cdots & v_{nm7} \end{bmatrix}^T
$$

3.3 What About When the Dimensions are not 'Nice'?

I'm working on it.

4 Jacobi Method: A Nontrivial Example

Over the course of this section I explain how to implement a Jacobi Method solver using intrinsics.