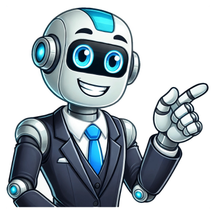
Continue



4x4 matrix multiplication example

And the result will have the same number of rows as the 1st matrix, and the same number of columns as the 2nd matrix. This may seem an odd and complicated way of multiplying, but it is necessary! I can give you a real-life example to illustrate why we multiply matrices in this way. Consider this implementation. So ... In General: To multiply an m×n matrix by an n×p matrix, the ns must be the same, and the result is an m×p matrix. struct MATRIX { union { float f[4][4]; __m128 m[4]; __m256 n[2]; }; }; MATRIX myMultiply(MATRIX M1, MATRIX M2) { // Perform a 4x4 matrix multiply by a 4x4 matrix // Be sure to run in 64 bit mode and set right flags // Properties, C/C++, Enable Enhanced Instruction, /arch:AVX // Having MATRIX on a 32 byte bundry does help performance MATRIX mResult; __m256 a0, a1, b0, b1; __m256 c0, c1, c2, c3, c4, c5, c6, c7; __m256 t0, t1, u0, u1; t0 = M1.n[0]; // t0 = a00, a01, a02, a03, a10, a11, a12, a13 t1 = M1.n[1]; // t1 = a20, a21, a22, a23, a30, a31, a32, a33 u0 = M2.n[0]; // u0 = b00, b01, b02, b03, b10, b11, b12, b13 u1 = M2.n[1]; // u1 = b20, b21, b22, b23, b30, b31, b32, b33 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(0, 0, 0, 0)); // a0 = a00, a00, a00, a00, a10, a10, a10, a10 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(0, 0, 0, 0)); // a1 = a20, a20, a20, a20, a30, a30, a30, a30 b0 = _mm256_permute2f128_ps(u0, u0, 0x00); // b0 = b00, b01, b02, b03, b00, b01, b02, b03 c0 = _mm256_mul_ps(a0, b0); // c0 = a00*b00 a00*b01 a00*b02 a00*b03 a10*b00 a10*b01 a10*b02 a10*b03 c1 = _mm256_mul_ps(a1, b0); // c1 = a20*b00 a20*b01 a20*b02 a20*b03 a30*b00 a30*b01 a30*b02 a30*b03 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(1, 1, 1, 1)); // a0 = a01, a01, a01, a01, a11, a11, a11, a11 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(1, 1, 1, 1)); // a1 = a21, a21, a21, a21, a31, a31, a31, a31 b0 = _mm256_permute2f128_ps(u0, u0, 0x11); // b0 = b10, b11, b12, b13, b10, b11, b12, b13 c2 = _mm256_mul_ps(a0, b0); // c2 = a01*b10 a01*b11 a01*b12 a01*b13 a11*b10 a11*b11 a11*b12 a11*b13 c3 = _mm256_mul_ps(a1, b0); // c3 = a21*b10 a21*b11 a21*b12 a21*b13 a31*b10 a31*b11 a31*b12 a31*b13 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(2, 2, 2, 2)); // a0 = a02, a02, a02, a02, a12, a12, a12, a12 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(2, 2, 2, 2)); // a1 = a22, a22, a22, a22, a32, a32, a32, a32 b1 = _mm256_permute2f128_ps(u1, u1, 0x00); // b0 = b20, b21, b22, b23, b20, b21, b22, b23 c4 = _mm256_mul_ps(a0, b1); // c4 = a02*b20 a02*b21 a02*b22 a02*b23 a12*b20 a12*b21 a12*b22 a12*b23 c5 = _mm256_mul_ps(a1, b1); // c5 = a22*b20 a22*b21 a22*b22 a22*b23 a32*b20 a32*b21 a32*b22 a32*b23 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(3, 3, 3, 3)); // a0 = a03, a03, a03, a03, a13, a13, a13, a13 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(3, 3, 3, 3)); // a1 = a23, a23, a23, a23, a33, a33, a33, a33 b1 = _mm256_permute2f128_ps(u1, u1, 0x11); // b0 = b30, b31, b32, b33, b30, b31, b32, b33 c6 = _mm256_mul_ps(a0, b1); // c6 = a03*b30 a03*b31 a03*b32 a03*b33 a13*b30 a13*b31 a13*b32 a13*b33 c7 = _mm256_mul_ps(a1, b1); // c7 = a23*b30 a23*b31 a23*b32 a23*b33 a33*b30 a33*b31 a33*b32 a33*b33 c0 = _mm256_add_ps(c0, c2); // c0 = c0 + c2 (two terms, first two rows) c4 = _mm256_add_ps(c4, c6); // c4 = c4 + c6 (the other two terms, first two rows) c1 = _mm256_add_ps(c1, c3); // c1 = c1 + c3 (two terms, second two rows) c5 = _mm256_add_ps(c5, c7); // c5 = c5 + c7 (the other two terms, second two rose) // Finally complete addition of all four terms and return the results mResult.n[0] = _mm256_add_ps(c0, c4); // n0 = a00*b00+a01*b10+a02*b20+a03*b30 a00*b01+a01*b11+a02*b21+a03*b31 a00*b02+a01*b12+a02*b22+a03*b32 a00*b03+a01*b13+a02*b23+a03*b33 // a10*b00+a11*b10+a12*b20+a13*b30 a10*b01+a11*b11+a12*b21+a13*b31 a10*b02+a11*b12+a12*b22+a13*b32 a10*b03+a11*b13+a12*b23+a13*b33 mResult.n[1] = _mm256_add_ps(c1, c5); // n1 = a20*b00+a21*b10+a22*b20+a23*b30 a20*b01+a21*b11+a22*b21+a23*b31 a20*b02+a21*b12+a22*b22+a23*b32 a20*b03+a21*b13+a22*b23+a23*b33 // a30*b00+a31*b10+a32*b20+a33*b30 a30*b01+a31*b11+a32*b21+a33*b31 a30*b02+a31*b12+a32*b22+a33*b32 a30*b03+a31*b13+a32*b23+a33*b33 return mResult; } multiplying a 1×3 by a 3×1 gets a 1×1 result: = = But multiplying a 3×1 by a 1×3 gets a 3×3 result: = 4×1 4×2 4×3 5×1 5×2 5×3 6×1 6×2 6×3 = Identity Matrix The "Identity Matrix" is the matrix equivalent of the number "1": A 3×3 Identity Matrix It is "square" (has same number of rows as columns) It can be large or small (2×2, 100×100, ... whatever) It has 1s on the main diagonal and 0s everywhere else Its symbol is the capital letter I It is a special matrix, because when we multiply by it, the original is unchanged: A × I = A I × A = A Order of Multiplication In arithmetic we are used to: 3 × 5 = 5 × 3 (The Commutative Law of Multiplication) But this is not generally true for matrices (matrix multiplication is not commutative): AB ≠ BA When we change the order of multiplication, the answer is (usually) different. Apple pies cost $3 each Cherry pies cost $4 each Blueberry pies cost $2 each And this is how many they sold in 4 days: Now think about this ... Let us see with an example: To work out the answer for the 1st row and 1st column: The "Dot Product" is where we multiply matching members, then sum up: (1, 2, 3) • (7, 9, 11) = 1×7 + 2×9 + 3×11 = 58 We match the 1st members (1 and 7), multiply them, likewise for the 2nd members (2 and 9) and the 3rd members (3 and 11), and finally sum them up. struct MATRIX { union { float f[4][4]; __m128 m[4]; __m256 n[2]; }; }; MATRIX myMultiply(MATRIX M1, MATRIX M2) { // Perform a 4x4 matrix multiply by a 4x4 matrix // Be sure to run in 64 bit mode and set right flags // Properties, C/C++, Enable Enhanced Instruction, /arch:AVX // Having MATRIX on a 32 byte bundry does help performance MATRIX mResult; __m256 a0, a1, b0, b1; __m256 c0, c1, c2, c3, c4, c5, c6, c7; __m256 t0, t1, u0, u1; t0 = M1.n[0]; // t0 = a00, a01, a02, a03, a10, a11, a12, a13 t1 = M1.n[1]; // t1 = a20, a21, a22, a23, a30, a31, a32, a33 u0 = M2.n[0]; // u0 = b00, b01, b02, b03, b10, b11, b12, b13 u1 = M2.n[1]; // u1 = b20, b21, b22, b23, b30, b31, b32, b33 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(0, 0, 0, 0)); // a0 = a00, a00, a00, a00, a10, a10, a10, a10 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(0, 0, 0, 0)); // a1 = a20, a20, a20, a20, a30, a30, a30, a30 b0 = _mm256_permute2f128_ps(u0, u0, 0x00); // b0 = b00, b01, b02, b03, b00, b01, b02, b03 c0 = _mm256_mul_ps(a0, b0); // c0 = a00*b00 a00*b01 a00*b02 a00*b03 a10*b00 a10*b01 a10*b02 a10*b03 c1 = _mm256_mul_ps(a1, b0); // c1 = a20*b00 a20*b01 a20*b02 a20*b03 a30*b00 a30*b01 a30*b02 a30*b03 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(1, 1, 1, 1)); // a0 = a01, a01, a01, a01, a11, a11, a11, a11 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(1, 1, 1, 1)); // a1 = a21, a21, a21, a21, a31, a31, a31, a31 b0 = _mm256_permute2f128_ps(u0, u0, 0x11); // b0 = b10, b11, b12, b13, b10, b11, b12, b13 c2 = _mm256_mul_ps(a0, b0); // c2 = a01*b10 a01*b11 a01*b12 a01*b13 a11*b10 a11*b11 a11*b12 a11*b13 c3 = _mm256_mul_ps(a1, b0); // c3 = a21*b10 a21*b11 a21*b12 a21*b13 a31*b10 a31*b11 a31*b12 a31*b13 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(2, 2, 2, 2)); // a0 = a02, a02, a02, a02, a12, a12, a12, a12 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(2, 2, 2, 2)); // a1 = a22, a22, a22, a22, a32, a32, a32, a32 b1 = _mm256_permute2f128_ps(u1, u1, 0x00); // b0 = b20, b21, b22, b23, b20, b21, b22, b23 c4 = _mm256_mul_ps(a0, b1); // c4 = a02*b20 a02*b21 a02*b22 a02*b23 a12*b20 a12*b21 a12*b22 a12*b23 c5 = _mm256_mul_ps(a1, b1); // c5 = a22*b20 a22*b21 a22*b22 a22*b23 a32*b20 a32*b21 a32*b22 a32*b23 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(3, 3, 3, 3)); // a0 = a03, a03, a03, a03, a13, a13, a13, a13 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(3, 3, 3, 3)); // a1 = a23, a23, a23, a23, a33, a33, a33, a33 b1 = _mm256_permute2f128_ps(u1, u1, 0x11); // b0 = b30, b31, b32, b33, b30, b31, b32, b33 c6 = _mm256_mul_ps(a0, b1); // c6 = a03*b30 a03*b31 a03*b32 a03*b33 a13*b30 a13*b31 a13*b32 a13*b33 c7 = _mm256_mul_ps(a1, b1); // c7 = a23*b30 a23*b31 a23*b32 a23*b33 a33*b30 a33*b31 a33*b32 a33*b33 c0 = _mm256_add_ps(c0, c2); // c0 = c0 + c2 (two terms, first two rows) c4 = _mm256_add_ps(c4, c6); // c4 = c4 + c6 (the other two terms, first two rows) c1 = _mm256_add_ps(c1, c3); // c1 = c1 + c3 (two terms, second two rows) c5 = _mm256_add_ps(c5, c7); // c5 = c5 + c7 (the other two terms, second two rose) // Finally complete addition of all four terms and return the results mResult.n[0] = _mm256_add_ps(c0, c4); // n0 = a00*b00+a01*b10+a02*b20+a03*b30 a00*b01+a01*b11+a02*b21+a03*b31 a00*b02+a01*b12+a02*b22+a03*b32 a00*b03+a01*b13+a02*b23+a03*b33 // a10*b00+a11*b10+a12*b20+a13*b30 a10*b01+a11*b11+a12*b21+a13*b31 a10*b02+a11*b12+a12*b22+a13*b32 a10*b03+a11*b13+a12*b23+a13*b33 mResult.n[1] = _mm256_add_ps(c1, c5); // n1 = a20*b00+a21*b10+a22*b20+a23*b30 a20*b01+a21*b11+a22*b21+a23*b31 a20*b02+a21*b12+a22*b22+a23*b32 a20*b03+a21*b13+a22*b23+a23*b33 // a30*b00+a31*b10+a32*b20+a33*b30 a30*b01+a31*b11+a32*b21+a33*b31 a30*b02+a31*b12+a32*b22+a33*b32 a30*b03+a31*b13+a32*b23+a33*b33 return mResult; } Sandy Bridge an above extend the instruction set to support 8 element vector arithmetic. And here is the full result in Matrix form: They sold $83 worth of pies on Monday, $63 on Tuesday, etc. Example: This matrix is 2×3 (2 rows by 3 columns): When we do multiplication: The number of columns of the 1st matrix must equal the number of rows of the 2nd matrix. what does that mean? Sandy Bridge an above extend the instruction set to support 8 element vector arithmetic. Now you know why we use the "dot product". 714, 715, 716, 717, 2394, 2395, 2397, 2396, 8473, 8474, 8475, 8476 Copyright © 2023 Rod Pierce A Matrix is an array of numbers: A Matrix (This one has 2 Rows and 3 Columns) To multiply a matrix by a single number is easy: These are the calculations: 2×4=8 2×0=0 2×1=2 2×-9=-18 We call the number ("2" in this case) a scalar, so this is called "scalar multiplication". See how changing the order affects this multiplication: = 1×2+2×1 1×0+2×2 3×2+4×1 3×0+4×2 = = 2×1+0×3 2×2+0×4 1×1+2×3 1×2+2×4 = The answers are different! It can have the same result (such as when one matrix is the Identity Matrix) but not usually. Multiplying a Matrix by Another Matrix But to multiply a matrix by another matrix we need to do the "dot product" of rows and columns ... struct MATRIX { union { float f[4][4]; __m128 m[4]; __m256 n[2]; }; }; MATRIX myMultiply(MATRIX M1, MATRIX M2) { // Perform a 4x4 matrix multiply by a 4x4 matrix // Be sure to run in 64 bit mode and set right flags // Properties, C/C++, Enable Enhanced Instruction, /arch:AVX // Having MATRIX on a 32 byte bundry does help performance MATRIX mResult; __m256 a0, a1, b0, b1; __m256 c0, c1, c2, c3, c4, c5, c6, c7; __m256 t0, t1, u0, u1; t0 = M1.n[0]; // t0 = a00, a01, a02, a03, a10, a11, a12, a13 t1 = M1.n[1]; // t1 = a20, a21, a22, a23, a30, a31, a32, a33 u0 = M2.n[0]; // u0 = b00, b01, b02, b03, b10, b11, b12, b13 u1 = M2.n[1]; // u1 = b20, b21, b22, b23, b30, b31, b32, b33 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(0, 0, 0, 0)); // a0 = a00, a00, a00, a00, a10, a10, a10, a10 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(0, 0, 0, 0)); // a1 = a20, a20, a20, a20, a30, a30, a30, a30 b0 = _mm256_permute2f128_ps(u0, u0, 0x00); // b0 = b00, b01, b02, b03, b00, b01, b02, b03 c0 = _mm256_mul_ps(a0, b0); // c0 = a00*b00 a00*b01 a00*b02 a00*b03 a10*b00 a10*b01 a10*b02 a10*b03 c1 = _mm256_mul_ps(a1, b0); // c1 = a20*b00 a20*b01 a20*b02 a20*b03 a30*b00 a30*b01 a30*b02 a30*b03 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(1, 1, 1, 1)); // a0 = a01, a01, a01, a01, a11, a11, a11, a11 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(1, 1, 1, 1)); // a1 = a21, a21, a21, a21, a31, a31, a31, a31 b0 = _mm256_permute2f128_ps(u0, u0, 0x11); // b0 = b10, b11, b12, b13, b10, b11, b12, b13 c2 = _mm256_mul_ps(a0, b0); // c2 = a01*b10 a01*b11 a01*b12 a01*b13 a11*b10 a11*b11 a11*b12 a11*b13 c3 = _mm256_mul_ps(a1, b0); // c3 = a21*b10 a21*b11 a21*b12 a21*b13 a31*b10 a31*b11 a31*b12 a31*b13 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(2, 2, 2, 2)); // a0 = a02, a02, a02, a02, a12, a12, a12, a12 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(2, 2, 2, 2)); // a1 = a22, a22, a22, a22, a32, a32, a32, a32 b1 = _mm256_permute2f128_ps(u1, u1, 0x00); // b0 = b20, b21, b22, b23, b20, b21, b22, b23 c4 = _mm256_mul_ps(a0, b1); // c4 = a02*b20 a02*b21 a02*b22 a02*b23 a12*b20 a12*b21 a12*b22 a12*b23 c5 = _mm256_mul_ps(a1, b1); // c5 = a22*b20 a22*b21 a22*b22 a22*b23 a32*b20 a32*b21 a32*b22 a32*b23 a0 = _mm256_shuffle_ps(t0, t0, _MM_SHUFFLE(3, 3, 3, 3)); // a0 = a03, a03, a03, a03, a13, a13, a13, a13 a1 = _mm256_shuffle_ps(t1, t1, _MM_SHUFFLE(3, 3, 3, 3)); // a1 = a23, a23, a23, a23, a33, a33, a33, a33 b1 = _mm256_permute2f128_ps(u1, u1, 0x11); // b0 = b30, b31, b32, b33, b30, b31, b32, b33 c6 = _mm256_mul_ps(a0, b1); // c6 = a03*b30 a03*b31 a03*b32 a03*b33 a13*b30 a13*b31 a13*b32 a13*b33 c7 = _mm256_mul_ps(a1, b1); // c7 = a23*b30 a23*b31 a23*b32 a23*b33 a33*b30 a33*b31 a33*b32 a33*b33 c0 = _mm256_add_ps(c0, c2); // c0 = c0 + c2 (two terms, first two rows) c4 = _mm256_add_ps(c4, c6); // c4 = c4 + c6 (the other two terms, first two rows) c1 = _mm256_add_ps(c1, c3); // c1 = c1 + c3 (two terms, second two rows) c5 = _mm256_add_ps(c5, c7); // c5 = c5 + c7 (the other two terms, second two rose) // Finally complete addition of all four terms and return the results mResult.n[0] = _mm256_add_ps(c0, c4); // n0 = a00*b00+a01*b10+a02*b20+a03*b30 a00*b01+a01*b11+a02*b21+a03*b31 a00*b02+a01*b12+a02*b22+a03*b32 a00*b03+a01*b13+a02*b23+a03*b33 // a10*b00+a11*b10+a12*b20+a13*b30 a10*b01+a11*b11+a12*b21+a13*b31 a10*b02+a11*b12+a12*b22+a13*b32 a10*b03+a11*b13+a12*b23+a13*b33 mResult.n[1] = _mm256_add_ps(c1, c5); // n1 = a20*b00+a21*b10+a22*b20+a23*b30 a20*b01+a21*b11+a22*b21+a23*b31 a20*b02+a21*b12+a22*b22+a23*b32 a20*b03+a21*b13+a22*b23+a23*b33 // a30*b00+a31*b10+a32*b20+a33*b30 a30*b01+a31*b11+a32*b21+a33*b31 a30*b02+a31*b12+a32*b22+a33*b32 a30*b03+a31*b13+a32*b23+a33*b33 return mResult; } A Matrix is an array of numbers: A Matrix (This one has 2 Rows and 3 Columns) To multiply a matrix by a single number is easy: These are the calculations: 2×4=8 2×0=0 2×1=2 2×-9=-18 We call the number ("2" in this case) a scalar, so this is called "scalar multiplication". (You can put those values into the Matrix Calculator to see if they work.) Rows and Columns To show how many rows and columns a matrix has we often write rows×columns. the value of sales for Monday is calculated this way: Apple pie value + Cherry pie value + Blueberry pie value $3×13 + $4×8 + $2×6 = $83 So it is, in fact, the "dot product" of prices and how many were sold: ($3, $4, $2) • (13, 8, 6) = $3×13 + $4×8 + $2×6 = $83 We match the price to how many sold, multiply each, then sum the result. In other words: The sales for Monday were: Apple pies: $3×13=$39, Cherry pies: $4×8=$32, and Blueberry pies: $2×6=$12. In that example we multiplied a 1×3 matrix by a 3×4 matrix (note the 3s are the same), and the result was a 1×4 matrix. 714, 715, 716, 717, 2394, 2395, 2397, 2396, 8473, 8474, 8475, 8476 Copyright © 2023 Rod Pierce Sandy Bridge an above extend the instruction set to support 8 element vector arithmetic. Here it is for the 1st row and 2nd column: (1, 2, 3) • (8, 10, 12) = 1×8 + 2×10 + 3×12 = 64 We can do the same thing for the 2nd row and 1st column: (4, 5, 6) • (7, 9, 11) = 4×7 + 5×9 + 6×11 = 139 And for the 2nd row and 2nd column: (4, 5, 6) • (8, 10, 12) = 4×8 + 5×10 + 6×12 = 154 And we get: DONE! Why Do It This Way? Together that is $39 + $32 + $12 = $83 And for Tuesday: $3×9 + $4×7 + $2×4 = $63 And for Wednesday: $3×7 + $4×4 + $2×0 = $37 And for Thursday: $3×15 + $4×6 + $2×3 = $75 So it is important to match each price to each quantity. Want to see another example?