Code Generation and Factoring for Fast Evaluation of Low-order Spherical Harmonic Products and Squares

John Snyder
Microsoft Research
2/1/2006

Abstract
We present a method for fast evaluation of spherical harmonic (SH) products or, more generally, any binary product of vectors yielding a vector, where the product is governed by a fixed, sparse, symmetric, order 3 tensor, denoted $\boldsymbol{\Gamma}_{ijk}$. The method is given the nonzero entries of $\boldsymbol{\Gamma}$ as input (they can be computed analytically or by numerical integration for the SH basis) and makes use of an offline code generator to perform the necessary array indexing using constants rather than variables. Factoring is performed by collecting the tensor's nonzero components, represented by index triples (*i*,*j*,*k*), into groups (*i*,*j*,k_1), (*i*,*j*,k_2), …,(*i*,*j*,k_{Nij}) which share a common pair of indices (*i*,*j*) in the triple, and which vary only in the third (completion) index k_m and its corresponding coefficient $d_m = \boldsymbol{\Gamma}_{i\,j\,k_m}$ where $m \in \{1, 2, \cdots, N_{ij}\}$. The collection is done using a greedy method that successively chooses the index pair (*i*,*j*) maximizing the number N_{ij} of different k_m needed to complete the tensor's nonzero index triples. The greedy method then continues to the next best initial pair, generates its contribution, and so on, until all nonzero triples have been accounted for. The combination of "greedy pair" factoring and generating constant array indices produces code that is significantly faster than naïve evaluation methods.

Introduction
An order-*n* spherical harmonic (SH) vector has n^2 components, and represents a bandlimited scalar function defined over the sphere. Given a spherical function, *f*(*s*), we can project it to determine a vector $\boldsymbol{f}$ representing its low-frequency behavior via

$$\boldsymbol{f} = \int_S f(s)\,\boldsymbol{y}(s)\,ds \tag{1}$$

where $\boldsymbol{y}(s)$ is the vector of SH basis functions evaluated at the spherical point $s \in S$. The SH basis functions are orthogonal polynomials in $s = (x,y,z)$ restricted to the sphere. Conversely, given an SH vector $\boldsymbol{f}$, we can reconstruct a continuous spherical function *f*(*s*) via

$$f(s) = \sum_{i=0}^{n^2-1} \boldsymbol{f}_i\, \boldsymbol{y}_i(s) = \boldsymbol{f} \cdot \boldsymbol{y}(s) \tag{2}$$

Note that we use 0-based indexing of the SH vectors. Italic names represent scalar values while italic boldface is used for vectors, matrices, and tensors.

Given two order-*n* SH vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, we wish to compute $\boldsymbol{a}*\boldsymbol{b}$, the bandlimited result of multiplying the two functions represented by $\boldsymbol{a}$ and $\boldsymbol{b}$ followed by a projection of this result back to SH. This result incurs truncation error since the true product of two order-*n* vectors is order-(2*n*-1). This truncated result is given by

$$\boldsymbol{a}*\boldsymbol{b} = \int_S a(s)\,b(s)\,\boldsymbol{y}(s)\,ds = \int_S \big(\boldsymbol{a}\cdot\boldsymbol{y}(s)\big)\big(\boldsymbol{b}\cdot\boldsymbol{y}(s)\big)\,\boldsymbol{y}(s)\,ds \tag{3}$$

Using component notation, this can be written

$$\big(\boldsymbol{a}*\boldsymbol{b}\big)_i = \sum_{j=0}^{n^2-1}\sum_{k=0}^{n^2-1} \boldsymbol{\Gamma}_{ijk}\,\boldsymbol{a}_j\,\boldsymbol{b}_k \tag{4}$$

where the SH triple product tensor $\boldsymbol{\Gamma}$ is defined by

$$\boldsymbol{\Gamma}_{ijk} = \int_S \boldsymbol{y}_i(s)\,\boldsymbol{y}_j(s)\,\boldsymbol{y}_k(s)\,ds \tag{5}$$

$\boldsymbol{\Gamma}$ is a symmetric, sparse, order 3 tensor. It has $\mathbf{O}(n^5)$ nonzero entries for order-*n* SH, whereas a non-sparse tensor would be $\mathbf{O}(n^6)$. The following table records the number of non-zero entries in the tensor (bottom row) for different SH orders (top row). The middle row counts non-zero entries but without counting redundant ones from symmetry.

order (*n*)	1	2	3	4	5	6	7	8	9	10
# nonzero $\boldsymbol{\Gamma}$ comp.	1	10	83	353	1158	2907	6460	12868	23621	40418
# nonzero $\boldsymbol{\Gamma}$ comp. (not counting sym.)	1	4	25	77	238	549	1196	2300	4185	7042

In addition to equation (4), we can also compute products $\boldsymbol{a}*\boldsymbol{b}$ by first computing the product matrix, $\boldsymbol{M_a}$, associated with a vector $\boldsymbol{a}$, and then computing the matrix/vector product $\boldsymbol{M_a}\,\boldsymbol{b}$. The product matrix is defined via

$$\left(\boldsymbol{M_a}\right)_{ij} = \sum_{k=0}^{n^2-1} \boldsymbol{\Gamma}_{ijk}\,\boldsymbol{a}_k \tag{6}$$

This is typically more efficient if the same matrix $\boldsymbol{M_a}$ is to be applied to multiple different vectors $\boldsymbol{b}$. Note that the product matrix for an arbitrary vector is always symmetric.

SH Product Code Generator

To accelerate evaluations of $\boldsymbol{c}=\boldsymbol{a}*\boldsymbol{b}$ via equation (4), we make use of a code generator which produces a C function to multiply the two SH input vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. The code generator can also produce code for squaring a single SH vector, $\boldsymbol{c}=\boldsymbol{a}*\boldsymbol{a}$. In both cases, the code is specialized to a particular SH order n. The code generator is given n as input, as well as a list of the nonzero entries in $\boldsymbol{\Gamma}$, stored as an array of index triples (i,j,k), and their corresponding coefficient $\boldsymbol{\Gamma}_{ijk}$. It assumes the tensor is symmetric, so that a single entry $(i, j, k, \boldsymbol{\Gamma}_{ijk})$ actually represents all combinations; i.e., (i,j,k), (i,k,j), (j,i,k), (j,k,i), (k,i,j), (k,j,i), excluding redundancies due to equality of indices i, j, and k.

Assume the list of sparse nonzero components of $\boldsymbol{\Gamma}$ is encoded as the array T, indexed by p, and having fields T[p].i, T[p].j, T[p].k representing the index triple and T[p].c representing its corresponding coefficient. The i, j, and k fields of the T array are arranged in increasing order, so that the k field is always the largest index of the triple.

The algorithm makes use of a 2D array N[i][j] (N[i][j]=N_{ij}), which stores the current number of different k values in nonzero triples (i,j,k). It is initialized to 0 for all (i,j). Then the entries in T are visited and used to increment entries in N, depending on the number of unique indices in the triple (T[p].i, T[p].j, T[p].k). If there is only one unique index i in this triple, we increment N[i][i]. If there are three, we increment N[i][j], N[i][k], and N[j][k], corresponding to all different pairs. If there are two, we increment the entry corresponding to the single repeated index i, N[i][i], and the entry between the repeated index i and the non-repeated index j, N[i][j].

Then we can search for the index pair (i,j) having the greatest N[i][j]. We record this index pair, its corresponding N[i][j], and the list (having N[i][j] elements) of k indices completing the tensor triples containing (i,j). Then we update the N array by reducing counts in all relevant entries corresponding to the triples (i,j,k) we just processed, and search for the next best index pair, until all tensor components have been accounted for.

The above "greedy pair" algorithm computes an array of N_T index pairs (i,j), each having its own list of N_{ij} "triple completion" indices k_m and corresponding coefficient d_m where m ranges from 1 to N_{ij}. Together, all the (i,j,k_m) from all index pairs and all completion indices in each pair's list yield all the original nonzero tensor triples (i,j,k). The greedy algorithm is an attempt to minimize N_T, by making N_{ij} as big as possible, at least initially. This provides a better factorization and so reduces the number of floating point operations required, as we will now show.

The code is then generated in (i,j) order of these index pairs. We first consider general SH products $\boldsymbol{c}=\boldsymbol{a}*\boldsymbol{b}$, and will later specialize to SH squares. If $i \neq j$, we generate the following C++ code:

```
ta = d1*a[k1] + d2*a[k2] + … + dN*a[kN];
tb = d1*b[k1] + d2*b[k2] + … + dN*b[kN];
c[i] += ta*b[j] + tb*a[j];
c[j] += ta*b[i] + tb*a[i];
t = a[i]*b[j] + a[j]*b[i];
c[k1] += d1*t;
c[k2] += d2*t;
⋮
c[kN] += dN*t;
// 3N+6 multiplies, 3N+3 additions
```

And if $i = j$, we generate the following code:

```
ta = d1*a[k1] + d2*a[k2] + … + dN*a[kN];  // km ≠ i
tb = d1*b[k1] + d2*b[k2] + … + dN*b[kN]; // km ≠ i
c[i] += ta*b[i] + tb*a[i];
t = a[i]*b[i];
c[k1] += d1*t;
c[k2] += d2*t;
⋮
c[kN] += dN*t;
// 3N+1 multiplies, 3N-2 additions
```

Note that the d_m and k_m in the above code fragments are replaced by actual floating point and integer constants, respectively, by the code generator. The above code can be optimized by keeping track of when a vector component of the output, c[i], is first assigned and using a simple assignment "=" in that initial case, followed by "+=" for later assignments. This is easily done using a array of flags for each vector component in the code generator which are initially clear and set after each assignment of the corresponding component.

These code fragments should be compared to naïve factorization code for a single triple (i,j,k). If we assume there are no repeated indices in this triple, and its corresponding coefficient is $d = \Gamma_{ijk}$ then the code is

```
c[i] += d*(a[j]*b[k]+a[k]*b[j]);
c[j] += d*(a[i]*b[k]+a[k]*b[i]);
c[k] += d*(a[i]*b[j]+a[j]*b[i]);
```

which requires 9 multiplies and 6 additions per triple. In other words, as N gets bigger, we save a factor of three in multiplies and two in additions by doing the pair factoring.

The above naive code is still better than the brute force formulation which takes no advantage of the fact that the tensor is symmetric. In that case the number of multiplies required is a factor of 2 more than the number of non-zero components in Γ shown in the table in the introduction. The brute force code is given by

```
c[i] += d*a[j]*b[k];
c[i] += d*a[k]*b[j];
c[j] += d*a[i]*b[k];
c[j] += d*a[k]*b[i];
c[k] += d*a[i]*b[j];
c[k] += d*a[j]*b[i];
```

requiring 12 multiplies and 6 additions per triple (i,j,k) with no repeated indices.

Now we consider the simpler SH squaring rather than general SH products, $\boldsymbol{c} = \boldsymbol{a}*\boldsymbol{a}$. If $i \neq j$, we generate the following C++ code:

```
ta = (2d_1)*a[k_1] + (2d_2)*a[k_2] + … + (2d_N)*a[k_N];
c[i] += ta*a[j];
c[j] += ta*a[i];
t = a[i]*a[j];
c[k_1] += (2d_1)*t;
c[k_2] += (2d_2)*t;
⋮
c[k_N] += (2d_N)*t;
// 2N+3 multiplies, 2N+1 additions
```

Here the $(2d_m)$ factors are precomputed in the code generation and so do not involve a run-time multiplication. And if $i = j$, we generate the following code:

```
ta = (2d_1)*a[k_1] + (2d_2)*a[k_2] + … + (2d_N)*a[k_N];  // k_m ≠ i
c[i] += ta*a[i];
t = a[i]*a[i];
c[k_1] += d_1*t;
c[k_2] += d_2*t;
⋮
c[k_N] += d_N*t;
// 2N+2 multiplies, 2N additions
```

This should be compared to the naïve code, again assuming a triple (i,j,k) containing no repeated index and a coefficient $d = \Gamma_{ijk}$, given by

```
c[i] += (2d)*(a[j]*a[k]);
c[j] += (2d)*(a[i]*a[k]);
c[k] += (2d)*(a[i]*a[j]);
```

which requires 6 multiplies and 3 additions.

A further enhancement is to add a stochastic element to the processing, repeat the stochastic factorization many times, and pick the result having the fewest total number of index pairs, N_T. The algorithm is made stochastic by randomly choosing an index pair when there is more than one having the identical, largest number of completion

indices N_{ij}. In practice, we find this reduces the operation count only very slightly at the expense of much more computation in the code generation.

Output

The following table compares the number of multiplies in the factored code (first number in comma-separated pair) vs. the naive code (second number in pair) for SH squares and products of increasing SH order *n*:

order (*n*)	3	4	5	6	7	8	9
N_T	13	37	84	166	320	533	890
# mult. (square)	77,104	246,388	699,1246	1556,3034	3298,6696	6130,13176	10953,24114
# mult. (product)	120,135	399,547	1135,1781	2527,4424	5351,9808	9896,19456	17640,35678
# mult. (brute force)	166	706	2316	5814	12920	25736	47242
# adds (square)	43,43	157,178	507,598	1189,1481	2610,3299	5001,6524	9093,11976
# adds (product)	74,74	274,337	860,1133	1995,2871	4344,6411	8235,12804	14891,23540
# adds (brute force)	74	337	1133	2871	6411	12804	23540
# mult. (product matrix)	52	194	623	1517	3348	6588	12057
# adds (product matrix)	12	69	313	871	2155	4546	8788

The factored code does not realize the full theoretical benefit of a factor of 3 reduction in number of multiplies for higher order SH, and instead reaches only a factor of about 2. This indicates there are a significant number of pairs having small N_{ij}.

The bottom two lines in the table record the number of multiplies and adds required in generated code for building an order *n* product matrix given a vector, via equation (6).

The following table records timings in seconds of 1,000,000 evaluations of the generated code on a 2.8GHz Intel Xeon CPU, using double precision arithmetic:

order (*n*)	name	3	4	5	6	7
matrix/vector multiply	`SH_mvmult_n`	0.0912	0.3036	0.8823	1.9661	3.7970
product matrix	`SH_matrix_n`	0.1222	0.3298	1.0738	2.3722	5.8911
square (factored)	`SH_square_n`	0.0866	0.3313	1.3346	1.9604	4.3004
square (naïve)	`SH_square_unopt_n`	0.1361	0.5363	2.0831	4.6831	10.2842
product (factored)	`SH_product_n`	0.1220	0.4493	1.4879	3.4298	7.2551
product (naive)	`SH_product_unopt_n`	0.1492	0.6414	2.2118	5.3543	11.4159
product (interpreted)	`SH_product`	0.6561	1.8429	5.5451	13.5347	28.7929

Matrix/vector multiply code simply multiplies an $n^2 \times n^2$ matrix times and n^2-dimensional vector. Product matrix code builds the product matrix given an SH vector, using equation (6). The square and product code computes $\boldsymbol{c}=\boldsymbol{a}*\boldsymbol{a}$ and $\boldsymbol{c}=\boldsymbol{a}*\boldsymbol{b}$ respectively, using code generation and either naïve or index pair factorization. The "interpreted" product timing is for generic code not specialized to a particular SH order. It does not use code generation at all, but instead traverses through the sparse list of nonzero coefficients T[*p*] by incrementing *p*, and accumulating the result as it goes. This code does not use greedy pair factorization and indexes source and destination components of the SH vectors using the variables T[*p*].i, T[*p*].j, T[*p*].k, multiplying together appropriate components of the pair of input vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ and the tensor coefficient T[*p*].c.

The following routines represent sample output of the code generator for order 3 SH. The routines `SH_square_unopt_3` and `SH_product_unopt_3` use the naive factorization (not the brute force formulation) while `SH_square_3` and `SH_product_3` use the greedy pair factorization algorithm described. The routine `SH_matrix_3` generates the 9×9 product matrix given an order 3 vector as input, and `SH_mvmult_3` multiplies a product matrix times a vector.

```
typedef double REAL;
#define CONSTANT(x) (x)


void SH_square_3(const REAL *a,REAL *c) {
     register REAL ta,t;

     // [0,0]: 0,
     c[0]  = CONSTANT(0.282094792935999980)*a[0]*a[0];

     // [1,1]: 0,6,8,
     ta = CONSTANT(0.564189583546000020)*a[0]+CONSTANT(-0.252313252202000020)*a[6]+CONSTANT(-0.437019372239999980)*a[8];
     c[1]  = ta*a[1];
     t = a[1]*a[1];
     c[0] += CONSTANT(0.282094791773000010)*t;
     c[6]  = CONSTANT(-0.126156626101000010)*t;
     c[8]  = CONSTANT(-0.218509686119999990)*t;

     // [1,2]: 5,
     ta = CONSTANT(0.437019372236000010)*a[5];
     c[1] += ta*a[2];
```

```
    c[2]  = ta*a[1];
    t = a[1]*a[2];
    c[5]  = CONSTANT(0.437019372236000010)*t;

    // [1,3]: 4,
    ta = CONSTANT(0.437019372229999980)*a[4];
    c[1] += ta*a[3];
    c[3]  = ta*a[1];
    t = a[1]*a[3];
    c[4]  = CONSTANT(0.437019372229999980)*t;

    // [2,2]: 0,6,
    ta = CONSTANT(0.564189590497999990)*a[0]+CONSTANT(0.504626519973999990)*a[6];
    c[2] += ta*a[2];
    t = a[2]*a[2];
    c[0] += CONSTANT(0.282094795249000000)*t;
    c[6] += CONSTANT(0.252313259986999990)*t;

    // [2,3]: 7,
    ta = CONSTANT(0.437019372236000010)*a[7];
    c[2] += ta*a[3];
    c[3] += ta*a[2];
    t = a[2]*a[3];
    c[7]  = CONSTANT(0.437019372236000010)*t;

    // [3,3]: 0,6,8,
    ta = CONSTANT(0.564189583546000020)*a[0]+CONSTANT(-0.252313252202000020)*a[6]+CONSTANT(0.437019372239999980)*a[8];
    c[3] += ta*a[3];
    t = a[3]*a[3];
    c[0] += CONSTANT(0.282094791773000010)*t;
    c[6] += CONSTANT(-0.126156626101000010)*t;
    c[8] += CONSTANT(0.218509686119999990)*t;

    // [4,4]: 0,6,
    ta = CONSTANT(0.564189583540000040)*a[0]+CONSTANT(-0.360447503152000030)*a[6];
    c[4] += ta*a[4];
    t = a[4]*a[4];
    c[0] += CONSTANT(0.282094791770000020)*t;
    c[6] += CONSTANT(-0.180223751576000010)*t;

    // [4,5]: 7,
    ta = CONSTANT(0.312156694452000010)*a[7];
    c[4] += ta*a[5];
    c[5] += ta*a[4];
    t = a[4]*a[5];
    c[7] += CONSTANT(0.312156694452000010)*t;

    // [5,5]: 0,6,8,
    ta = CONSTANT(0.564189583547999970)*a[0]+CONSTANT(0.180223751573000000)*a[6]+CONSTANT(-0.312156694455999970)*a[8];
    c[5] += ta*a[5];
    t = a[5]*a[5];
    c[0] += CONSTANT(0.282094791773999990)*t;
    c[6] += CONSTANT(0.090111875786499998)*t;
    c[8] += CONSTANT(-0.156078347227999990)*t;

    // [6,6]: 0,6,
    ta = CONSTANT(0.564189595120000000)*a[0];
    c[6] += ta*a[6];
    t = a[6]*a[6];
    c[0] += CONSTANT(0.282094797560000000)*t;
    c[6] += CONSTANT(0.180223764527000010)*t;

    // [7,7]: 0,6,8,
    ta = CONSTANT(0.564189583547999970)*a[0]+CONSTANT(0.180223751573000000)*a[6]+CONSTANT(0.312156694455999970)*a[8];
    c[7] += ta*a[7];
    t = a[7]*a[7];
    c[0] += CONSTANT(0.282094791773999990)*t;
    c[6] += CONSTANT(0.090111875786499998)*t;
    c[8] += CONSTANT(0.156078347227999990)*t;

    // [8,8]: 0,6,
    ta = CONSTANT(0.564189583540000040)*a[0]+CONSTANT(-0.360447503152000030)*a[6];
    c[8] += ta*a[8];
    t = a[8]*a[8];
    c[0] += CONSTANT(0.282094791770000020)*t;
    c[6] += CONSTANT(-0.180223751576000010)*t;

    // entry count=13
    // multiply count=77
    // addition count=43
}

void SH_product_3(const REAL *a,const REAL *b,REAL *c) {
    register REAL ta,tb,t;
    // [0,0]: 0,
    c[0]  = CONSTANT(0.282094792935999980)*a[0]*b[0];

    // [1,1]: 0,6,8,
    ta = CONSTANT(0.282094791773000010)*a[0]+CONSTANT(-0.126156626101000010)*a[6]+CONSTANT(-0.218509686119999990)*a[8];
    tb = CONSTANT(0.282094791773000010)*b[0]+CONSTANT(-0.126156626101000010)*b[6]+CONSTANT(-0.218509686119999990)*b[8];
    c[1]  = ta*b[1]+tb*a[1];
    t = a[1]*b[1];
    c[0] += CONSTANT(0.282094791773000010)*t;
    c[6]  = CONSTANT(-0.126156626101000010)*t;
    c[8]  = CONSTANT(-0.218509686119999990)*t;

    // [1,2]: 5,
    ta = CONSTANT(0.218509686118000010)*a[5];
    tb = CONSTANT(0.218509686118000010)*b[5];
    c[1] += ta*b[2]+tb*a[2];
    c[2]  = ta*b[1]+tb*a[1];
    t = a[1]*b[2]+a[2]*b[1];
    c[5]  = CONSTANT(0.218509686118000010)*t;

    // [1,3]: 4,
    ta = CONSTANT(0.218509686114999990)*a[4];
    tb = CONSTANT(0.218509686114999990)*b[4];
    c[1] += ta*b[3]+tb*a[3];
    c[3]  = ta*b[1]+tb*a[1];
    t = a[1]*b[3]+a[3]*b[1];
    c[4]  = CONSTANT(0.218509686114999990)*t;

    // [2,2]: 0,6,
    ta = CONSTANT(0.282094795249000000)*a[0]+CONSTANT(0.252313259986999990)*a[6];
```

```
    tb = CONSTANT(0.282094795249000000)*b[0]+CONSTANT(0.252313259986999990)*b[6];
    c[2] += ta*b[2]+tb*a[2];
    t = a[2]*b[2];
    c[0] += CONSTANT(0.282094795249000000)*t;
    c[6] += CONSTANT(0.252313259986999990)*t;

    // [2,3]: 7,
    ta = CONSTANT(0.218509686118000010)*a[7];
    tb = CONSTANT(0.218509686118000010)*b[7];
    c[2] += ta*b[3]+tb*a[3];
    c[3] += ta*b[2]+tb*a[2];
    t = a[2]*b[3]+a[3]*b[2];
    c[7]  = CONSTANT(0.218509686118000010)*t;

    // [3,3]: 0,6,8,
    ta = CONSTANT(0.282094791773000010)*a[0]+CONSTANT(-0.126156626101000010)*a[6]+CONSTANT(0.218509686119999990)*a[8];
    tb = CONSTANT(0.282094791773000010)*b[0]+CONSTANT(-0.126156626101000010)*b[6]+CONSTANT(0.218509686119999990)*b[8];
    c[3] += ta*b[3]+tb*a[3];
    t = a[3]*b[3];
    c[0] += CONSTANT(0.282094791773000010)*t;
    c[6] += CONSTANT(-0.126156626101000010)*t;
    c[8] += CONSTANT(0.218509686119999990)*t;

    // [4,4]: 0,6,
    ta = CONSTANT(0.282094791770000020)*a[0]+CONSTANT(-0.180223751576000010)*a[6];
    tb = CONSTANT(0.282094791770000020)*b[0]+CONSTANT(-0.180223751576000010)*b[6];
    c[4] += ta*b[4]+tb*a[4];
    t = a[4]*b[4];
    c[0] += CONSTANT(0.282094791770000020)*t;
    c[6] += CONSTANT(-0.180223751576000010)*t;

    // [4,5]: 7,
    ta = CONSTANT(0.156078347226000000)*a[7];
    tb = CONSTANT(0.156078347226000000)*b[7];
    c[4] += ta*b[5]+tb*a[5];
    c[5] += ta*b[4]+tb*a[4];
    t = a[4]*b[5]+a[5]*b[4];
    c[7] += CONSTANT(0.156078347226000000)*t;

    // [5,5]: 0,6,8,
    ta = CONSTANT(0.282094791773999990)*a[0]+CONSTANT(0.090111875786499998)*a[6]+CONSTANT(-0.156078347227999990)*a[8];
    tb = CONSTANT(0.282094791773999990)*b[0]+CONSTANT(0.090111875786499998)*b[6]+CONSTANT(-0.156078347227999990)*b[8];
    c[5] += ta*b[5]+tb*a[5];
    t = a[5]*b[5];
    c[0] += CONSTANT(0.282094791773999990)*t;
    c[6] += CONSTANT(0.090111875786499998)*t;
    c[8] += CONSTANT(-0.156078347227999990)*t;

    // [6,6]: 0,6,
    ta = CONSTANT(0.282094797560000000)*a[0];
    tb = CONSTANT(0.282094797560000000)*b[0];
    c[6] += ta*b[6]+tb*a[6];
    t = a[6]*b[6];
    c[0] += CONSTANT(0.282094797560000000)*t;
    c[6] += CONSTANT(0.180223764527000010)*t;

    // [7,7]: 0,6,8,
    ta = CONSTANT(0.282094791773999990)*a[0]+CONSTANT(0.090111875786499998)*a[6]+CONSTANT(0.156078347227999990)*a[8];
    tb = CONSTANT(0.282094791773999990)*b[0]+CONSTANT(0.090111875786499998)*b[6]+CONSTANT(0.156078347227999990)*b[8];
    c[7] += ta*b[7]+tb*a[7];
    t = a[7]*b[7];
    c[0] += CONSTANT(0.282094791773999990)*t;
    c[6] += CONSTANT(0.090111875786499998)*t;
    c[8] += CONSTANT(0.156078347227999990)*t;

    // [8,8]: 0,6,
    ta = CONSTANT(0.282094791770000020)*a[0]+CONSTANT(-0.180223751576000010)*a[6];
    tb = CONSTANT(0.282094791770000020)*b[0]+CONSTANT(-0.180223751576000010)*b[6];
    c[8] += ta*b[8]+tb*a[8];
    t = a[8]*b[8];
    c[0] += CONSTANT(0.282094791770000020)*t;
    c[6] += CONSTANT(-0.180223751576000010)*t;

    // entry count=13
    // multiply count=120
    // addition count=74
}

void SH_square_unopt_3(const REAL *a,REAL *c) {
    // 0,0,0
    c[0]  = CONSTANT(0.282094792935999980)*a[0]*a[0];

    // 0,1,1
    c[0] += CONSTANT(0.282094791773000010)*a[1]*a[1];
    c[1]  = CONSTANT(0.564189583546000020)*a[1]*a[0];

    // 0,2,2
    c[0] += CONSTANT(0.282094795249000000)*a[2]*a[2];
    c[2]  = CONSTANT(0.564189590497999990)*a[2]*a[0];

    // 0,3,3
    c[0] += CONSTANT(0.282094791773000010)*a[3]*a[3];
    c[3]  = CONSTANT(0.564189583546000020)*a[3]*a[0];

    // 1,3,4
    c[1] += CONSTANT(0.437019372229999980)*a[3]*a[4];
    c[3] += CONSTANT(0.437019372229999980)*a[4]*a[1];
    c[4]  = CONSTANT(0.437019372229999980)*a[1]*a[3];

    // 0,4,4
    c[0] += CONSTANT(0.282094791770000020)*a[4]*a[4];
    c[4] += CONSTANT(0.564189583540000040)*a[4]*a[0];

    // 1,2,5
    c[1] += CONSTANT(0.437019372236000010)*a[2]*a[5];
    c[2] += CONSTANT(0.437019372236000010)*a[5]*a[1];
    c[5]  = CONSTANT(0.437019372236000010)*a[1]*a[2];

    // 0,5,5
    c[0] += CONSTANT(0.282094791773999990)*a[5]*a[5];
    c[5] += CONSTANT(0.564189583547999970)*a[5]*a[0];

    // 1,1,6
```

```
      c[6]  = CONSTANT(-0.126156626101000010)*a[1]*a[1];
      c[1] += CONSTANT(-0.252313252202000020)*a[1]*a[6];

      // 2,2,6
      c[6] += CONSTANT(0.252313259986999990)*a[2]*a[2];
      c[2] += CONSTANT(0.504626519973999990)*a[2]*a[6];

      // 3,3,6
      c[6] += CONSTANT(-0.126156626101000010)*a[3]*a[3];
      c[3] += CONSTANT(-0.252313252202000020)*a[3]*a[6];

      // 4,4,6
      c[6] += CONSTANT(-0.180223751576000010)*a[4]*a[4];
      c[4] += CONSTANT(-0.360447503152000030)*a[4]*a[6];

      // 5,5,6
      c[6] += CONSTANT(0.090111875786499998)*a[5]*a[5];
      c[5] += CONSTANT(0.180223751573000000)*a[5]*a[6];

      // 0,6,6
      c[0] += CONSTANT(0.282094797560000000)*a[6]*a[6];
      c[6] += CONSTANT(0.564189595120000000)*a[6]*a[0];

      // 6,6,6
      c[6] += CONSTANT(0.180223764527000010)*a[6]*a[6];

      // 2,3,7
      c[2] += CONSTANT(0.437019372236000010)*a[3]*a[7];
      c[3] += CONSTANT(0.437019372236000010)*a[7]*a[2];
      c[7]  = CONSTANT(0.437019372236000010)*a[2]*a[3];

      // 4,5,7
      c[4] += CONSTANT(0.312156694452000010)*a[5]*a[7];
      c[5] += CONSTANT(0.312156694452000010)*a[7]*a[4];
      c[7] += CONSTANT(0.312156694452000010)*a[4]*a[5];

      // 0,7,7
      c[0] += CONSTANT(0.282094791773999990)*a[7]*a[7];
      c[7] += CONSTANT(0.564189583547999970)*a[7]*a[0];

      // 6,7,7
      c[6] += CONSTANT(0.090111875786499998)*a[7]*a[7];
      c[7] += CONSTANT(0.180223751573000000)*a[7]*a[6];

      // 1,1,8
      c[8]  = CONSTANT(-0.218509686119999990)*a[1]*a[1];
      c[1] += CONSTANT(-0.437019372239999980)*a[1]*a[8];

      // 3,3,8
      c[8] += CONSTANT(0.218509686119999990)*a[3]*a[3];
      c[3] += CONSTANT(0.437019372239999980)*a[3]*a[8];

      // 5,5,8
      c[8] += CONSTANT(-0.156078347227999990)*a[5]*a[5];
      c[5] += CONSTANT(-0.312156694455999970)*a[5]*a[8];

      // 7,7,8
      c[8] += CONSTANT(0.156078347227999990)*a[7]*a[7];
      c[7] += CONSTANT(0.312156694455999970)*a[7]*a[8];

      // 0,8,8
      c[0] += CONSTANT(0.282094791770000020)*a[8]*a[8];
      c[8] += CONSTANT(0.564189583540000040)*a[8]*a[0];

      // 6,8,8
      c[6] += CONSTANT(-0.180223751576000010)*a[8]*a[8];
      c[8] += CONSTANT(-0.360447503152000030)*a[8]*a[6];

      // entry count=25
      // multiply count=104
      // addition count=43
}

void SH_product_unopt_3(const REAL *a,const REAL *b,REAL *c) {
      // 0,0,0
      c[0]  = CONSTANT(0.282094792935999980)*a[0]*b[0];

      // 0,1,1
      c[0] += CONSTANT(0.282094791773000010)*a[1]*b[1];
      c[1]  = CONSTANT(0.282094791773000010)*(a[1]*b[0]+a[0]*b[1]);

      // 0,2,2
      c[0] += CONSTANT(0.282094795249000000)*a[2]*b[2];
      c[2]  = CONSTANT(0.282094795249000000)*(a[2]*b[0]+a[0]*b[2]);

      // 0,3,3
      c[0] += CONSTANT(0.282094791773000010)*a[3]*b[3];
      c[3]  = CONSTANT(0.282094791773000010)*(a[3]*b[0]+a[0]*b[3]);

      // 1,3,4
      c[1] += CONSTANT(0.218509686114999990)*(a[3]*b[4]+a[4]*b[3]);
      c[3] += CONSTANT(0.218509686114999990)*(a[4]*b[1]+a[1]*b[4]);
      c[4]  = CONSTANT(0.218509686114999990)*(a[1]*b[3]+a[3]*b[1]);

      // 0,4,4
      c[0] += CONSTANT(0.282094791770000020)*a[4]*b[4];
      c[4] += CONSTANT(0.282094791770000020)*(a[4]*b[0]+a[0]*b[4]);

      // 1,2,5
      c[1] += CONSTANT(0.218509686118000010)*(a[2]*b[5]+a[5]*b[2]);
      c[2] += CONSTANT(0.218509686118000010)*(a[5]*b[1]+a[1]*b[5]);
      c[5]  = CONSTANT(0.218509686118000010)*(a[1]*b[2]+a[2]*b[1]);

      // 0,5,5
      c[0] += CONSTANT(0.282094791773999990)*a[5]*b[5];
      c[5] += CONSTANT(0.282094791773999990)*(a[5]*b[0]+a[0]*b[5]);

      // 1,1,6
      c[6]  = CONSTANT(-0.126156626101000010)*a[1]*b[1];
      c[1] += CONSTANT(-0.126156626101000010)*(a[1]*b[6]+a[6]*b[1]);

      // 2,2,6
      c[6] += CONSTANT(0.252313259986999990)*a[2]*b[2];
      c[2] += CONSTANT(0.252313259986999990)*(a[2]*b[6]+a[6]*b[2]);
```

```
      // 3,3,6
      c[6] += CONSTANT(-0.126156626101000010)*a[3]*b[3];
      c[3] += CONSTANT(-0.126156626101000010)*(a[3]*b[6]+a[6]*b[3]);

      // 4,4,6
      c[6] += CONSTANT(-0.180223751576000010)*a[4]*b[4];
      c[4] += CONSTANT(-0.180223751576000010)*(a[4]*b[6]+a[6]*b[4]);

      // 5,5,6
      c[6] += CONSTANT(0.090111875786499998)*a[5]*b[5];
      c[5] += CONSTANT(0.090111875786499998)*(a[5]*b[6]+a[6]*b[5]);

      // 0,6,6
      c[0] += CONSTANT(0.282094797560000000)*a[6]*b[6];
      c[6] += CONSTANT(0.282094797560000000)*(a[6]*b[0]+a[0]*b[6]);

      // 6,6,6
      c[6] += CONSTANT(0.180223764527000010)*a[6]*b[6];

      // 2,3,7
      c[2] += CONSTANT(0.218509686118000010)*(a[3]*b[7]+a[7]*b[3]);
      c[3] += CONSTANT(0.218509686118000010)*(a[7]*b[2]+a[2]*b[7]);
      c[7]  = CONSTANT(0.218509686118000010)*(a[2]*b[3]+a[3]*b[2]);

      // 4,5,7
      c[4] += CONSTANT(0.156078347226000000)*(a[5]*b[7]+a[7]*b[5]);
      c[5] += CONSTANT(0.156078347226000000)*(a[7]*b[4]+a[4]*b[7]);
      c[7] += CONSTANT(0.156078347226000000)*(a[4]*b[5]+a[5]*b[4]);

      // 0,7,7
      c[0] += CONSTANT(0.282094791773999990)*a[7]*b[7];
      c[7] += CONSTANT(0.282094791773999990)*(a[7]*b[0]+a[0]*b[7]);

      // 6,7,7
      c[6] += CONSTANT(0.090111875786499998)*a[7]*b[7];
      c[7] += CONSTANT(0.090111875786499998)*(a[7]*b[6]+a[6]*b[7]);

      // 1,1,8
      c[8]  = CONSTANT(-0.218509686119999990)*a[1]*b[1];
      c[1] += CONSTANT(-0.218509686119999990)*(a[1]*b[8]+a[8]*b[1]);

      // 3,3,8
      c[8] += CONSTANT(0.218509686119999990)*a[3]*b[3];
      c[3] += CONSTANT(0.218509686119999990)*(a[3]*b[8]+a[8]*b[3]);

      // 5,5,8
      c[8] += CONSTANT(-0.156078347227999990)*a[5]*b[5];
      c[5] += CONSTANT(-0.156078347227999990)*(a[5]*b[8]+a[8]*b[5]);

      // 7,7,8
      c[8] += CONSTANT(0.156078347227999990)*a[7]*b[7];
      c[7] += CONSTANT(0.156078347227999990)*(a[7]*b[8]+a[8]*b[7]);

      // 0,8,8
      c[0] += CONSTANT(0.282094791770000020)*a[8]*b[8];
      c[8] += CONSTANT(0.282094791770000020)*(a[8]*b[0]+a[0]*b[8]);

      // 6,8,8
      c[6] += CONSTANT(-0.180223751576000010)*a[8]*b[8];
      c[8] += CONSTANT(-0.180223751576000010)*(a[8]*b[6]+a[6]*b[8]);

      // entry count=25
      // multiply count=135
      // addition count=74
}

void SH_matrix_3(const REAL *a,REAL *M) {
      // compute upper triangular part of matrix
      M[0] = CONSTANT(0.282094792935999980)*a[0];
      M[1] = CONSTANT(0.282094791773000010)*a[1];
      M[2] = CONSTANT(0.282094795249000000)*a[2];
      M[3] = CONSTANT(0.282094791773000010)*a[3];
      M[4] = CONSTANT(0.282094791770000020)*a[4];
      M[5] = CONSTANT(0.282094791773999990)*a[5];
      M[6] = CONSTANT(0.282094797560000000)*a[6];
      M[7] = CONSTANT(0.282094791773999990)*a[7];
      M[8] = CONSTANT(0.282094791770000020)*a[8];
      M[10] = CONSTANT(0.282094791773000010)*a[0]+CONSTANT(-0.126156626101000010)*a[6]+CONSTANT(-0.218509686119999990)*a[8];
      M[11] = CONSTANT(0.218509686118000010)*a[5];
      M[12] = CONSTANT(0.218509686114999990)*a[4];
      M[13] = CONSTANT(0.218509686114999990)*a[3];
      M[14] = CONSTANT(0.218509686118000010)*a[2];
      M[15] = CONSTANT(-0.126156626101000010)*a[1];
      M[16] = 0;
      M[17] = CONSTANT(-0.218509686119999990)*a[1];
      M[20] = CONSTANT(0.282094795249000000)*a[0]+CONSTANT(0.252313259986999990)*a[6];
      M[21] = CONSTANT(0.218509686118000010)*a[7];
      M[22] = 0;
      M[23] = CONSTANT(0.218509686118000010)*a[1];
      M[24] = CONSTANT(0.252313259986999990)*a[2];
      M[25] = CONSTANT(0.218509686118000010)*a[3];
      M[26] = 0;
      M[30] = CONSTANT(0.282094791773000010)*a[0]+CONSTANT(-0.126156626101000010)*a[6]+CONSTANT(0.218509686119999990)*a[8];
      M[31] = CONSTANT(0.218509686114999990)*a[1];
      M[32] = 0;
      M[33] = CONSTANT(-0.126156626101000010)*a[3];
      M[34] = CONSTANT(0.218509686118000010)*a[2];
      M[35] = CONSTANT(0.218509686119999990)*a[3];
      M[40] = CONSTANT(0.282094791770000020)*a[0]+CONSTANT(-0.180223751576000010)*a[6];
      M[41] = CONSTANT(0.156078347226000000)*a[7];
      M[42] = CONSTANT(-0.180223751576000010)*a[4];
      M[43] = CONSTANT(0.156078347226000000)*a[5];
      M[44] = 0;
      M[50] = CONSTANT(0.282094791773999990)*a[0]+CONSTANT(0.090111875786499998)*a[6]+CONSTANT(-0.156078347227999990)*a[8];
      M[51] = CONSTANT(0.090111875786499998)*a[5];
      M[52] = CONSTANT(0.156078347226000000)*a[4];
      M[53] = CONSTANT(-0.156078347227999990)*a[5];
      M[60] = CONSTANT(0.282094797560000000)*a[0]+CONSTANT(0.180223764527000010)*a[6];
      M[61] = CONSTANT(0.090111875786499998)*a[7];
      M[62] = CONSTANT(-0.180223751576000010)*a[8];
      M[70] = CONSTANT(0.282094791773999990)*a[0]+CONSTANT(0.090111875786499998)*a[6]+CONSTANT(0.156078347227999990)*a[8];
      M[71] = CONSTANT(0.156078347227999990)*a[7];
      M[80] = CONSTANT(0.282094791770000020)*a[0]+CONSTANT(-0.180223751576000010)*a[6];
```

```
	// entry count=40
	// multiply count=52
	// addition count=12

	// fill in lower triangular part of matrix
	M[9] = M[1]; // 1,0 = 0,1
	M[18] = M[2]; // 2,0 = 0,2
	M[19] = M[11]; // 2,1 = 1,2
	M[27] = M[3]; // 3,0 = 0,3
	M[28] = M[12]; // 3,1 = 1,3
	M[29] = M[21]; // 3,2 = 2,3
	M[36] = M[4]; // 4,0 = 0,4
	M[37] = M[13]; // 4,1 = 1,4
	M[38] = M[22]; // 4,2 = 2,4
	M[39] = M[31]; // 4,3 = 3,4
	M[45] = M[5]; // 5,0 = 0,5
	M[46] = M[14]; // 5,1 = 1,5
	M[47] = M[23]; // 5,2 = 2,5
	M[48] = M[32]; // 5,3 = 3,5
	M[49] = M[41]; // 5,4 = 4,5
	M[54] = M[6]; // 6,0 = 0,6
	M[55] = M[15]; // 6,1 = 1,6
	M[56] = M[24]; // 6,2 = 2,6
	M[57] = M[33]; // 6,3 = 3,6
	M[58] = M[42]; // 6,4 = 4,6
	M[59] = M[51]; // 6,5 = 5,6
	M[63] = M[7]; // 7,0 = 0,7
	M[64] = M[16]; // 7,1 = 1,7
	M[65] = M[25]; // 7,2 = 2,7
	M[66] = M[34]; // 7,3 = 3,7
	M[67] = M[43]; // 7,4 = 4,7
	M[68] = M[52]; // 7,5 = 5,7
	M[69] = M[61]; // 7,6 = 6,7
	M[72] = M[8]; // 8,0 = 0,8
	M[73] = M[17]; // 8,1 = 1,8
	M[74] = M[26]; // 8,2 = 2,8
	M[75] = M[35]; // 8,3 = 3,8
	M[76] = M[44]; // 8,4 = 4,8
	M[77] = M[53]; // 8,5 = 5,8
	M[78] = M[62]; // 8,6 = 6,8
	M[79] = M[71]; // 8,7 = 7,8
}

void SH_mvmult_3(REAL *M,REAL *a,REAL *c)
{
	for (int i = 0; i < 9; i++) {
		c[i] = M[0]*a[0] +
		       M[1]*a[1] +
		       M[2]*a[2] +
		       M[3]*a[3] +
		       M[4]*a[4] +
		       M[5]*a[5] +
		       M[6]*a[6] +
		       M[7]*a[7] +
		       M[8]*a[8];
		M += 9;
	}
	// multiply count=81
	// addition count=72
}
```