API Specification Doc

(Auto-vectorization for Xeon Phi Architecture)

Version	Date	Author	Description
2.0	14-Apr-2014	Xin Huo, Bin Ren	

Data Type

Data Type	Name	Description
Scalar Type	int, float, double	Data is shared by all the SIMD lanes. All the basic data types or temporary variables are belonged to Scalar Type
Vector Type	vint, vfloat, vdouble	It includes multiple data scaling to all the SIMD lanes.
Mask Type	mask	It helps handling control flow in vectorization

Methods

Assignment Methods		
API	**Descriptions**	**Examples (vint vi1, vi2; vfloat vf1, vf2; vdouble vd1, vd2; int si; float sf; double sd; OP represents the supported mathematic operations including +, -, *, /)**

Vec = Vec;	Assign a vector type variable to another vector type variable.	vi1 = vi2;
		vf1 = vf2;
		vd1 = vd2;

Vec = Scalar;	Assign a scalar type variable to a vector type variable. The assignment operation will replicate the scalar variable to fill the whole vector variable	vi1 = si;
		vf1 = sf;
		vd1 = sd;

Vec OP= Vec	Assignment operation can be executed with a mathematic operation. It equals to Vec = Vec OP Vec.	vi1 += vi2; vf1 += vf2; vd1 += vd2;
		vi1 -= vi2; vf1 -= vf2; vd1 -= vd2;
		vi1 *= vi2; vf1 *= vf2; vd1 *= vd2;
		vi1 /= vi2; vf1 /= vf2; vd1 /= vd2;

Vec OP= Scalar	Assignment operation can be executed with a mathematic operation taking a scalar type variable as the second parameter. The scalar type will be implicitly transferred to vector type before do the computation. So it is equals to Vec = Vec OP (Vec)Scalar.	vi1 += si; vf1 += sf; vd1 += sd;
		vi1 -= si; vf1 -= sf; vd1 -= sd;
		vi1 *= sf; vf1 *= sf; vd1 *= sd;
		vi1 /= sf; vf1 /= sf; vd1 /= sd;

Mathematic Methods		
API	**Descriptions**	**Examples** **(vint vi1, vi2, vi3; vfloat vf1, vf2, vf3; vdouble vd1, vd2, vd3; int si; float sf; double sd; OP represents the supported mathematic operations including +, -, *, /)**

Vec OP Vec	Support +, -, *, /, %, mathematic operations between two vector type variables. The returned result is also vector type.	vi1 = vi2 + vi3;
		vi1 = vi2 - vi3;
		vi1 = vi2 * vi3;
		vi1 = vi2 / vi3;
		
		vf1 = vf2 + vf3;
		vf1 = vf2 - vf3;
		vf1 = vf2 * vf3;
		vf1 = vf2 / vf3;
		
		vd1 = vd2 + vd3;
		vd1 = vd2 - vd3;
		vd1 = vd2 * vd3;
		vd1 = vd2 / vd3;

Vec OP Scalar	Support +, -, *, /, %, mathematic operations between a vector type variable and a scalar type variable. The scalar type variable will be implicitly transformed to vector type before computation. The returned result is vector type.	vi1 = vi2 + si;
		vi1 = vi2 - si;
		vi1 = vi2 * si;
		vi1 = vi2 / si;
		
		vf1 = vf2 + sf;
		vf1 = vf2 - sf;
		vf1 = vf2 * sf;
		vf1 = vf2 / sf;
		
		vd1 = vd2 + sd;
		vd1 = vd2 - sd;
		vd1 = vd2 * sd;
		vd1 = vd2 / sd;

Logic Methods		
API	**Descriptions**	**Examples** **(vint vi1, vi2, vi3; vfloat vf1, vf2, vf3; vdouble vd1, vd2, vd3; int si; float sf; doubld sd; mask m; OP represents the supported logic operations including ==, != , <, >, <=, >=)**

Vec OP Vec	Support logic operations between two vector type variables. Return type is a mask type.	m = vi1 OP vi2;
		m = vf1 OP vf2;
		m = vd1 OP vd2;

Vec OP Scalar	Support logic operations between a vector type variable and a scalar variable. Return type is a mask type. (scalar type will be implicitly transformed to vector type before further operations)	m = vi1 OP si;
		m = vf1 OP sf;
		m = vd1 OP sd;

Load / Store Methods		
API	**Descriptions**	**Examples (vint vi1, vi2; vfloat vf1; vdouble vd1; int si; float sf; double sd; void *addr)**

void load(void *src)	load data from src address to fill a vector type variable.	vi1.load(addr);
		vf1.load(addr);
		vd1.load(addr);

void store(void *dst)	store the content in the vector type variable to the memory indicted by the dst address.	vi1.store(addr);
		vf1.store(addr);
		vd1.store(addr);

void load(void *src, const vint &index, int scale)	achieve continuous data load The address for the ith element is src+index[i]*scale. The value of scale is power of 2. (1, 2, 4, 8)	vi1.load(addr, vi2, si);
		vf1.load(addr, vi2, si);
		vd1.load(addr, vi2, si);

void store(void *dst, const vint &index, int scale)	achieve continuous data store. The address for the ith element is src+index[i]*scale. The value of scale is power of 2. (1, 2, 4, 8)	vi1.store(addr, vi2, si);
		vf1.store(addr, vi2, si);
		vd1.store(addr, vi2, si);

Reduction Methods		
API	**Descriptions**	**Examples (vint vi; vfloat vf; vdouble vd; int si, float sf; double sd; void *update; int scale; int offset; vint index)**

template<class ReducComp = reducAdd > void reduction(void *update, int scale, int offset, vint index, type value, [mask m])
 	
template<typename T>
class ReducComp{
public:
const T operator ()(T &lsh, const T &rsh)
{
return (lsh+=rsh);
}
};
	Reduction may introduce implicit data dependencies. The reduction method provide a way to reduce all the elements to the calculated address in a safe manner. The reduce address for ith element is update +index[i]*scale+offset. mask variable can be passed as a optional parameter. Our API provides some reduce operations, like add operation. Users can also defined their own reduce function, and passed it to the template.	reduction(update, scale, offset, index, vi);
		
		reduction(update, scale, offset, index, vf);
		
		reduction(update, scale, offset, index, vd);
		
		reduction(update, scale, offset, index, si);
		
		reduction(update, scale, offset, index, sf);
		
		reduction(update, scale, offset, index, sd);

Mask Class

Mask Class	
Data member	**Descriptions**
static __thread mask m;	m, which is a mask type, provides the default mask value for mask operation methods. m is static and local to each thread.
static __thread vfloat oldf;	oldf, which is a vfloat type, provides the default old value for float mask operation methods. oldf is static and local to each thread.
static __thread vint oldi;	oldi, which is a vint type, provides the default old value for integer mask operation methods. oldi is static and local to each thread.
static __thread vint oldd;	oldd, which is a vdouble type, provides the default old value for double mask operation methods. oldd is static and local to each thread.
Method member	**Descriptions**
static void clear()	Set m to ~0 (active all the vector lanes). Set oldf to 0.0. Set oldi to 0.
static void set_mask(const mask &_m, const vfloat &oldf)	Set new mask and old value for float mask operation methods.
static void set_mask(const mask &_m, const vint &oldi)	Set new mask and old value for integer mask operation methods.
static void set_mask(const mask &_m, const vdouble &oldd)	Set new mask and old value for double mask operation methods.

Mask class is used to set mask and old value for current thread. mask variable *m* represents which threads are activated (1 represents activated, 0 represents

Mask Conversion Method		
API	**Descriptions**	**Examples (vint vi1, vi2; vfloat vf1, vf2; vdouble vd1, vd2;**

Mask::vfloat& mask();
 	
Mask::vint& mask();
 	
Mask::vdouble& mask();
	Convert from general vector type to Mask vector type. After this conversion, all the operations on the converted vector variables employ current mask class state.	vi1.mask() += vi2.mask();
		vf1.mask() = vf2.mask();
		vd1.mask() -= vd2.mask();

inactivated). old variable is the default value for the inactivated threads. Once mask and old value is set, they will be used until users call *set_mask* or *clear methods*. Because the data members in Mask class are local to each thread, different threads will not interfere the states of each other.

A comparison of if-else control flow and its corresponding mask code examples are show in the following.

Comparison between serial if-else brach codes and mask SIMD codes	
Serial if-else control flow	**Mask based SIMD codes**
vint a, b, c; **if(a < b)** ** c = a;** **else** ** c = b;**	**vint a, b, c;** **mask m = a < b;** **Mask::set_mask(m, b);** **c.mask() = a.mask();**

Case Study

1. Stencil Computation (Sobel Computation)

Sobel is a stencil computation. For 2D Sobel, two 3x3 weight templates (weight_H and weight_V) are used to compute the weight on the target point.

In Figure 1, 2, and 3, we compare serial, vectorized using our API, and manually vectorized Sobel computation versions.

Comparing between Figure 1 and 2, the vectorized codes in our API are almost as same as the serial version, except new vector types (*vfloat*) are introduced to replace the original scalar types (*float*). Another difference is that the assignment from vector type to scalar type is achieved through the *store* API, because it needs to involve multiple data copies from the vector variable to the target memory locations. Also, to facilitate a possible data reorganization at runtime, a function *Dim(idx, offset1, offset2, . . .)* is provided to calculate the transformed index in each dimension by applying offsets on different dimensions. For example, in Figure 2, X(i, p, q) calculates the transformed index in the X-dimension when applying p and q offsets on the original X and Y dimensions, respectively.

```
void kernel(int i, int j){
    float Dx = 0.0, Dy = 0.0;
    //Compute the weight for a node in a 3x3 area
    for(int p = -1; p <= 1; p++){
        for(int q = -1; q <= 1; q++){
            Dx += weight_H[p+1][q+1]*b[i+p][j+q];
            Dy += weight_V[p+1][q+1]*b[i+p][j+q];
        }
    }
    float z = sqrt(Dx*Dx + Dy*Dy);
    a[i][j] = z;
}
```

Figure 1: Sobel: Stencil Computation with Serial Codes

```
void kernel(int i, int j){
    vfloat Dx = 0.0, Dy = 0.0;
    //Compute the weight for a node in a 3x3 area
    for(int p = -1; p <= 1; p++){
        for(int q = -1; q <= 1; q++){
            Dx += weight_H[p+1][q+1]*b[X(i,p,q)][Y(j,p,q)];
            Dy += weight_V[p+1][q+1]*b[X(i,p,q)][Y(j,p,q)];
        }
    }
    vfloat z = sqrt(Dx*Dx + Dy*Dy);
    z.store(&a[i][j]);
}
```

Figure 2: Sobel: Stencil Computation with SIMD API

However, the manual version, shown in Figure 3, introduced more new Intel IMCI API, an is much more complicated compared to both serial and our SIMD API versions, as about 40% extra lines are added.

```
void kernel(int i, int j){
    __m512 Dx = _mm_set1_ps(0.0), Dy = _mm_set1_ps(0.0);
    //Compute the weight for a node in a 3x3 area
    for(int p = -1; p <=1; ++p){
        for(int q = -1; q <=1; ++q){
            __m512 *tmp = (__m512*)&b[i+q][j+p*vec_width];
            __m512 tmpx = _mm512_mul_ps(*tmp,weight_H[p+1][q+1]);
            Dx = _mm512_add_ps(Dx, tmpx);
            __m512 tmpy = _mm512_mul_ps(*tmp,weight_V[p+1][q+1]);
            Dy = _mm512_add_ps(Dy, tmpy);
        }
    }
    __m512 sqDX = _mm512_mul_ps(Dx, Dx);
    __m512 sqDy = _mm512_mul_ps(Dy, Dy);
    __m512 ret = _mm512_add_ps(sqDx, sqDy);
    ret = _mm512_sqrt_ps(ret);
    _mm512_store_ps(&a[i][j], ret);
}
```

Figure 3: Sobel: Stencil Computation with Manual Vectorization

2. Generalized Reduction (K-means Computation)

```
void kernel(float *data, int i){
    float min = FLT_MAX;
    int min_index = 0;
    for(int j = 0; j < k; ++j){
        //step 1 (Computation): compute the distance
        float dis = 0.0;
        for(int m = 0; m < 3; ++m){
            dis += (data[i*3+m]-cluster[j*3+m])*
            (data[i*3+m]-cluster[j*3+m]);
        }
        dis = sqrt(dis);
        //step 2 (Control flow): update index
        if(dis < min){
            min = dis;
            min_index = j;
        }
        //the else branch can be omitted
        else{
            min = min;
            min_index = min_index;
        }
    }
    //step 3 (Reduction): reduction
    update[5*min_index]   += data[i*3];
    update[5*min_index+1] += data[i*3+1];
    update[5*min_index+2] += data[i*3+2];
    update[5*min_index+3] += 1.0;
    update[5*min_index+4] += min;
}
```

Figure 4: K-means: Generalized Reduction with Serial Codes

K-means is a data clustering kernel - in each iteration, it processes each point in the dataset, determines the closest center to this point, and computes how this center's location should be updated. In Figure 4, 5, and 6, we compare serial, vectorized using our API, and manually vectorized K-means versions. The procedures of K-means can be divided into three steps: 1) compute the distance between one node and the candidate clusters, 2) update index to the cluster with the minimum distance, 3) do reduction on the cluster found in the step 2. Thus, the step 1 is only simple arithmetic operations, whereas steps 2 and 3 involve control flow and generalized reduction, respectively.

```
void kernel(vfloat *data, int i){
    vfloat min = FLT_MAX;
    vint min_index = 0;
    for(int j = 0; j < k; ++j){
        //step 1 (Computation): compute the distance
        vfloat dis = 0.0;
        for(int m = 0; m < 3; ++m){
            dis += (data[i+m*n]-cluster[j*3+m])*
            (data[i+m*n]-cluster[j*3+m]);
        }
        dis = sqrt(dis);
        //step 2 (Control flow): update index
        mask m = dis < min;
        set_mask(m, min);
        min.mask() = dis.mask();
        set_mask(m, min_index);
        min_index.mask() = j;
    }
    //step 3 (Reduction): reduction
    reduction(update, 5, min_index, 0, data[i]);
    reduction(update, 5, min_index, 1, data[i+n]);
    reduction(update, 5, min_index, 2, data[i+n*2]);
    reduction(update, 5, min_index, 3, 1.0);
    reduction(update, 5, min_index, 4, min);
}
```

Figure 5: K-means: Generalized Reduction with SIMD API

In Figure 5, we show the main function of K-means with vectorization by using our API. In the step 1, similar to the stencil computation, the only modification is the data types of corresponding variables are changed from scalar type to the vector type. As a result, the computation is automatically vectorized by loading values from the *data* array to all vector lanes, and computing the distance between the data in each lane and the clusters. The step 2 introduces a branch, specifically, if the distance is smaller than the current minimum distance (min), we update the min and min_index, otherwise, min and min_index are not changed. Using our mask API, we represent this computation as *if-else* branch, in which the *else* branch just assigns its own value to itself. As we can see in step 2 of the Figure 5, a mask variable m is returned by the logic computation. Then, m and the default value for *else* branch are set by *set_mask* function. Next, the *mask()* function will do the conversion from unmask vector type to the mask vector type. In the

step 3, reduction is performed on the array *update* with the add operation. It is not safe to perform the reduction using the general arithmetic and assignment operations, due to the potential written conflict between different vector lane. Thus, we use the API for reduction. Here, *add* operation, which is the default reduce operation for reduction function, is used to reduce values to the array *update*.

Figure 6 shows the manual vectorization version. It is obvious that the manual version is very tedious, and introduces more efforts to users.

```
void kernel(float *data, int i){

    for(int i = start; i < end; i += vec_width)
    {
        __m512 *data_cord[DIM];
        data_cord[0] = (__m512 *)&data[i];
        data_cord[1] = (__m512 *)&data[i+n];
        data_cord[2] = (__m512 *)&data[i+2*n];
        __m512i min_index = _mm512_set1_epi32(0);
        __m512 sse_min = _mm512_set1_ps(65536*65);

        __m512 cluster_cord, dis_cord, dis;
        for(int j = 0; j < k; j++)
        {
            dis = _mm512_set1_ps(0);

            for(int t = 0; t < DIM; ++t){
                cluster_cord = _mm512_set1_ps(cluster[j*DIM+t]);
                dis_cord = _mm512_sub_ps(*data_cord[t], cluster_cord);
                dis = _mm512_add_ps(dis, _mm512_mul_ps(dis_cord,
dis_cord));
            }

            dis = _mm512_mul_ps(dis, _mm512_rsqrt23_ps(dis));

            __mmask16 mask = _mm512_cmplt_ps_mask(dis, sse_min);
            sse_min = _mm512_mask_mov_ps(sse_min, mask, dis);
            min_index = _mm512_mask_mov_epi32(min_index, mask,
_mm512_set1_epi32(j));
        }
        for(int t = 0; t < 16; t++)
            update[((int *)&min_index)[t]*5] += ((float
*)data_cord[0])[t];
        for(int t = 0; t < 16; t++)
            update[((int *)&min_index)[t]*5+1] += ((float
*)data_cord[1])[t];
        for(int t = 0; t < 16; t++)
            update[((int *)&min_index)[t]*5+2] += ((float
*)data_cord[2])[t];
        for(int t = 0; t < 16; t++)
            update[((int *)&min_index)[t]*5+3] += 1;
        for(int t = 0; t < 16; t++)
            update[((int *)&min_index)[t]*5+4] += ((float *)&sse_min)
[t];
    }
}
```

Figure 6: K-means: Generalized Reduction with Manual Vectorization