SSE Vectorization of the EM Algorithm for Mixture of Gaussians Density Estimation

1. Introduction

The Expectation-Maximization (EM) algorithm is a popular tool for determining maximum likelihood estimates (MLE) when a closed form solution does not exist. It is often used for parametric density estimation, that is, to estimate the parameters of a density function when knowledge of the parameters is equivalent to knowledge of the density. The most famous case is the Gaussian distribution, which is fully specified by its mean and variance. The MLEs of the mean and variance of a single Gaussian are given by the sample mean and sample variance, no EM algorithm necessary. However, in real world applications, data is often generated from several unknown sources. In cases such as this, the data can be modeled as a mixture of several densities. The mixture of Gaussians (MoG) is a common model and has been used in image segmentation and saliency detection tasks for computer vision (1). To achieve real-time processing for computer vision the EM algorithm must be fast. The purpose of this paper is to demonstrate the speedup gained by a streaming SIMD extensions (SSE) formulation of the EM algorithm when compared to a straight forward implementation. The rest of the paper is organized as follows. Section 2 presents a brief overview of the EM MoG algorithm and its implementation. Section 3 discusses the SSE. Section 4 explains the SSE formulation of the EM MoG algorithm with results presented in Section 5. Section 6 concludes the paper.

2. EM Mixture of Gaussians Algorithm

The EM algorithm works recursively. At each step it calculates a conditional expectation given the current parameter estimates, and then the estimates are updated given the current expectations. Finally the likelihood of the estimate is calculated for use as a convergence criterion. The top level code for the algorithm follows.

```c
init_em(n, X, k, W, M, V);
float Ln = likelihood(n, X, k, W, M, V);
int niter = 0;

while((abs(100*(Ln-Lo)/Lo) > ltol) && (niter <= maxiter)){
    expectation(E, n, X, k, W, M, V);
    maximization(E, n, X, k, W, M, V);
    Lo = Ln;
    Ln = likelihood(n, X, k, W, M, V);
    niter++;
}
```
Before looking at the expectation (E-step) and maximization (M-step) we review the MoG formulation. A K component MoG has the form

\[
p(y|\theta) = \sum_{m=1}^{K} \alpha_m p(y|\theta_m)
\]

\[
p(y|\theta) = \frac{1}{(2\pi)^{d/2}} \sum_{m=1}^{K} \frac{\alpha_m}{|C_m|^{1/2}} \exp\left(-\frac{1}{2} (y - \mu_m)^T C_m^{-1} (y - \mu_m)\right)
\]

Where \(y\) is \(d\)-dimensional data, \(\theta\) a \(K\)-length vector of mixture parameters, and \(\theta_m = [\mu_m, C_m]\) the mean and covariance of the \(m\)th component. Given the feature space \(\mathcal{Y} = [y_1, ..., y_N]\), the goal is to estimate the mixture parameters \(\theta\) and cluster labels \(\mathcal{Z} = [z_1, ..., z_N]\), where \(z_m(i) = 1\) if pixel \(i\) belongs to component \(j\) and is zero otherwise. This estimation is performed using the EM algorithm. The E-step consists of calculating the expected value of the log likelihood function. The non-trivial component of this computation consists of calculating the conditional expectation of the cluster labels \(z_m(i)\) given the feature space data and the current (step \(t\)) mixture parameters estimate \(\hat{\theta}_t\).

\[
E[\log(p(y, \mathcal{Z}|\theta))|y, \hat{\theta}_t] = \sum_{i=1}^{N} \sum_{m=1}^{K} E[z_m(i)|y, \hat{\theta}_t] \log(\alpha_m p(y_i|\theta_m))
\]

\[
E[z_m(i)|y, \hat{\theta}_t] = \Pr[z_m(i) = 1|y_i, \hat{\theta}_t] = \frac{\hat{\alpha}_{m,t} p(y_i|\hat{\theta}_{m,t})}{\sum_{j=1}^{K} \hat{\alpha}_{j,t} p(y_i|\hat{\theta}_{j,t})}
\]

A straightforward implementation of the E-step follows.

```cpp
for(int i = 0; i < n; i++){
    row_sum = 0;
    for(int j = 0; j < k; j++){
        E[j+i*k] = W[j]/sqrt(2*PI*V[j])*exp(-1/(2*V[j])*pow(X[i]-M[j],2));
        row_sum += E[j+i*k];
    }
    for(int j = 0; j < k; j++){ E[j+i*k] = E[j+i*k]/row_sum; }
}
```

The E step is followed by a M-step that selects the parameters that best predict the cluster labels \(\mathcal{Z}\).

\[
\hat{\theta}_{t+1} = \arg \max_{\theta} E[\log(p(y, \mathcal{Z}|\theta))|y, \hat{\theta}_t]
\]

This has the following C++ formulation.

```cpp
for(int i = 0; i < k; i++){ W[i] = M[i] = V[i] = 0; }
```
for(int i = 0; i < k; i++){  
    for(int j = 0; j < n; j++){  
        W[i] = W[i] + E[i+j*k];  
        M[i] = M[i] + E[i+j*k]*X[j];  
    }  
    M[i] = M[i]/W[i];  
}

for(int i = 0; i < k; i++){  
    for(int j = 0; j < n; j++){  
        V[i] = V[i] + E[i+j*k]*pow(X[j]-M[i],2);  
    }  
    V[i] = V[i]/W[i];  
}

for(int i = 0; i < k; i++){  
    W[i] = W[i]/float(n);  
}

3. Streaming SIMD Extensions

With the introduction of the Pentium III, Intel introduced 70 new SIMD operations and additional registers. Similar to the multi-media extensions (MMX) used for integer SIMD execution, the SSE provide floating point SIMD. Unlike MMX though, the SSE registers are not shared with the floating point unit. Since there original release the number of SSE instructions has expanded several times to keep up with the current multi-media algorithms.

For the programmer, the SSE provide eight 128bit registers that can hold up to four single precision floating point numbers.

SIMD stands for single instruction multiple data. One SSE instruction can operate simultaneously on all four floating point numbers in the packed register format.
By using the SSE a maximum speedup of four, defined as the old execution time divided by the new execution time, can be obtained. In reality, due to program overhead and the fact that every instruction cannot be implemented in SSE, a speedup of less than four is expected until the data set size becomes large. When the data set size is large, data computation time far outweighs overhead computation and the speedup of 4 can be approached in the limit as the data set size grows to infinity.

Instead of working directly with SSE instructions in assembly, Intel has released a SSE intrinsics API for use with the C/C++ language. The intrinsics provide the application programmer the convenience of programming in a high level language while also providing access to the SSE registers and instructions. In most cases the intrinsics provide a one-to-one mapping from C code function call to SSE assembly operation. These are called simple intrinsics. Composite intrinsics map one C code function call to several assembly operations. The following simple intrinsics are used in the SSE reformulation of the EM MoG algorithm.

- \(_\text{mm\_add\_ps}()\) $\rightarrow$ ADDPS
- \(_\text{mm\_sub\_ps}()\) $\rightarrow$ SUBPS
- \(_\text{mm\_mul\_ps}()\) $\rightarrow$ MULPS
- \(_\text{mm\_div\_ps}()\) $\rightarrow$ DIVPS
- \(_\text{mm\_rsqrt\_ps}()\) $\rightarrow$ RSQRTPS

The only composite intrinsic used is the load or set operations.

- \(_\text{mm\_set\_ps1}()\) $\rightarrow$ MOVSS and a series of shifts

This instruction is implemented by loading a single value into the lowest doubleword of the SSE register. Then a series of shuffle operations copy the data to the upper three doublewords.

Finally, when using the SSE intrinsics, all data vectors must be aligned to a 16 byte boundary for optimal memory access.

### 4. SSE Algorithm Reformulation

Using the SSE intrinsics described in Section 3, the EM MoG algorithm from Section 2 is reformulated to take advantage of SIMD execution. The new formulation of the E-step follows.

```c
//precompute the scaling factors
for(j = 0; j < k/4; j++){
    twoVSSE[j] = _mm_mul_ps(VSSE[j],two);
    scaleFactorSSE[j] = _mm_mul_ps(twoVSSE[j],pi);
    scaleFactorSSE[j] = _mm_rsqrt_ps(scaleFactorSSE[j]);
}
```
scaleFactorSSE[j] = _mm_mul_ps(WSSE[j], scaleFactorSSE[j]);
}

//for each data point
for (i = 0; i < n; i++) {

    *rsSSE = zero;
    *xnSSE = _mm_set_ps1(X[i]); //load data into SSE register

    for (j = 0; j < k/4; j++){
        //compute the argument for the exponential
        *tmpSSE = _mm_sub_ps(*xnSSE, MSSE[j]);
        *tmpSSE = _mm_mul_ps(*tmpSSE, *tmpSSE);
        *tmpSSE = _mm_div_ps(*tmpSSE, twoVSSE[j]);
        *tmpSSE = _mm_mul_ps(*tmpSSE, negone);

        //compute the exponential of each argument
        tmp[0] = exp(tmp[0]);
        tmp[1] = exp(tmp[1]);
        tmp[2] = exp(tmp[2]);
        tmp[3] = exp(tmp[3]);

        //scale, accumulate, and store the conditional expectation
        *tmpSSE = _mm_mul_ps(scaleFactorSSE[j], *tmpSSE);
        *rsSSE = _mm_add_ps(*rsSSE, *tmpSSE);
        ESSE[j+(i*k/4)] = *tmpSSE;
    }

    //accumulate row_sum for the normalizing factor
    *tmpSSE = _mm_set_ps1(normfactor);

    //normalize sums
    for (j = 0; j < k/4; j++) { ESSE[j+(i*k)/4] = _mm_div_ps(ESSE[j+(i*k)/4], *tmpSSE); }
}

The SSE formulation is coded for speed by efficient memory access and efficient register usage. In the old formulation, the same scaling factors were computed every iteration. In the new formulation all scaling factors are pre-computed, once, for speed. For efficient memory access each data is loaded from memory only once during the E-step. The main speedup comes from the look-ahead transformation of the inner loop.

The look-ahead transformation is applied three times so that four inner loops are simultaneously executed with SSE instructions. Because this is written using the SSE intrinsics, we do not directly assign data to an SSE register. To ensure efficient assembly code, SSE variables are reused as often as possible to implicitly keep the number of registers to a minimum.

The exponential function is used within the inner loop. There is no exponential SSE operation. It is a high level function call. The break from SSE instructions to normal high level function calls will affect performance. Fortunately unlike the MMX, SSE do not use the floating point
registers of the processor. Therefore during the exponential function calls the state information of the SSE registers do not need to be saved and overhead is minimized.

The result of the E-step is passed to the M-step for further computation. The design of the E-step, specifically the way the expected value is stored in memory, affects the design of the M-step. The expectation variable $E$ was declared as follows.

```c
float* E = (float*) aligned_malloc(n*k * sizeof(float), 16);
```

The expectations are stored by consecutive rows. Following the computation of the E-step each row consists of the expected value data $X[i]$ belongs to components 1 to $K$. The $E$ variable has the following form.

$$E = \begin{bmatrix} E_{11} & \ldots & E_{1K} & E_{21} & \ldots & E_{2K} & \ldots & E_{NK} \end{bmatrix}$$

The variables $E_{11}$ to $E_{NK}$ are stored in consecutive memory locations. The M-step is implemented as follows.

```c
//reset the estimates
for(i = 0; i < k/4; i++){ WSSE[i] = MSSE[i] = VSSE[i] = zero; }

//update mean and variance estimates
for(i = 0; i < k/4; i++){

    //calculate the expected value (mean) of each component
    for(j = 0; j < n; j++){
        WSSE[i] = _mm_add_ps(WSSE[i],ESSE[i+(j*k)/4]);
        *xnSSE = _mm_set_ps1(X[j]);
        *xnSSE = _mm_mul_ps(ESSE[i+(j*k)/4],*xnSSE);
        MSSE[i] = _mm_add_ps(MSSE[i],*xnSSE);
    }
    MSSE[i] = _mm_div_ps(MSSE[i],WSSE[i]);

    //estimate the variances, then normalize
    for(j = 0; j < n; j++){
        *xnSSE = _mm_set_ps1(X[j]);
        *xnSSE = _mm_sub_ps(*xnSSE,MSSE[i]);
        *xnSSE = _mm_mul_ps(*xnSSE,*xnSSE);
        *xnSSE = _mm_add_ps(VSSE[i],*xnSSE);
    }
    VSSE[i] = _mm_div_ps(VSSE[i],WSSE[i]);
}

//normalize the weights
*xnSSE = _mm_set_ps1(n);
for(i = 0; i < k/4; i++){ WSSE[i] = _mm_div_ps(WSSE[i],*xnSSE); }
```

Due to the implementation of the E-step, and the storage of the variable $E$ in memory, performing the look-ahead transformation on the inner loops would require reading four values of $E$ from four non-consecutive memory locations. This is terribly inefficient. Thus the only
option was to unroll the outer loops in a manner similar to the E-step. Each data X[i] is read from memory a minimum of two times. This data access rate doubles for every four additional mixture components.

The second inner loop is dependent on the scaled version of the mean produced from the first inner loop. In an attempt to improve performance, these two loops were combined and the scaling was performed afterward. After running a set of tests, the variances were no longer computed correctly. It is believed that roll-over was occurring due to the accumulation of VSSE[i] in the second inner loop. Upon further consideration, it is more likely that attempt at improvement was incorrect.

5. Results

To test the two implementations data was randomly drawn from K=4, 8, 12, and 16 component Gaussians. For each mixture size K, the data set size N was varied from N=4000, 8000, 16000, and 32000. The execution time for each (K,N) combination was averaged over 100 trials producing the following four charts.
Note that for the K=4, a Matlab test was also run to verify the correctness of the result and to compare speed. From the graphs above it is clear that the SSE formulation provides an algorithmic speedup. Unfortunately it appears that for a given K, the speedup remains approximately constant with increasing N. The following chart displays the speedup.

![Speedup realized with SSE](chart.png)

For the test cases chosen here the speedup actually becomes less as the number of mixtures increases. For each K, the speedup varies little over set size.

SSE was designed for use in data intensive applications. Upon further review of the results just presented it was determined that the data set sizes chosen above were too small to display a significant gain from SIMD execution. This can be attributed to the necessary overhead incurred when switching from regular execution to SIMD execution such as memory reads and loading registers. Until the data set size becomes large only modest gains occur. In the Intel Architecture manual there example description of SSE specifies a data set size of 40000; therefore, for a K=4 component mixture the data set size was increased and the tests were run again. The following graph shows a significantly different trend.
Clearly at larger set sizes, the SSE formulation begins to produce large speedups. The previous graphs correspond to the first four data points of this chart. At these data set sizes, less than 40000, there is negligible difference between the two formulations. For the set sizes of 64k, 128k, 256k, and 512k the speedup is approximately 2 and appears to be increasing.

6. Conclusion

In order to see considerable speedup gains from SSE, the data set size must be over 40k. At this point the overhead is negligible compared to the amount of useful computation. The intended application of this algorithm is for use in clustering for image segmentation and saliency detection. In the saliency detection application time constraints caused by slow Matlab execution have limited image sizes to a maximum of 256 by 256 = 65536 data points. The number of mixture components ranges from 3 to 12. From the results section, it is clear that larger image sizes could be used with the SSE formulation of the EM MoG.

When designing the SSE optimized algorithm, a choice existed to either perform look-ahead on the loop over the data set of size N, or the components set of size K. The look-ahead was performed on the component loop originally to reduce the memory accesses in the E-step. This caused poor memory access in the M-step. Based on the final implementation code, it appears that more calculation occurs in the M-step. Further performance gains could be possible by reformulating the M-step first and performing looking ahead on the data loop. This would reduce the number of memory reads necessary for execution. The E-step would then need to
be reformulated so that the expectation result E is saved in memory efficiently for use in the M-step.

The look-ahead transformation was also performed on the component loops because it was thought that even small data sets would benefit. Although, small datasets do see a speedup the overhead of SSE and function calls is only overcome once the dataset exceeds 40k.

7. Bibliography


