DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University
Electrical and Computer Engineering Dept.
Parallel Computer Architecture ECEC-622

TITLE: SSE Assignment: Gaussian Elimination

GROUP MEMBERS: Gregory Matthews and Mark Klobukov

INSTRUCTOR: Dr. Kandasamy

DATE SUBMITTED: 2/28/2017

DATE DUE: 2/28/2017

Assignment description:

The first PThread assignment involved parallelizing a given program that performs Gaussian elimination of an NxN system of linear equations. The algorithm is shown in the pseudo-code below:

```
procedure GAUSS_ELIMINATE($A$, $b$, $y$)
int $i$, $j$, $k$;
for $k := 0$ to $n-1$ do
    for $j := k+1$ to $n-1$ do
        $A[k,j] := A[k,j]/A[k,k]$;     /* Division step. */
    end for
    $y[k] := b[k]/A[k,k]$;
    $A[k,k] := 1$;
    for $i := k+1$ to $n-1$ do
        for $j := k+1$ to $n-1$ do
            $A[i,j] := A[i,j] - A[i,k] \times A[k,j]$;     /* Elimination step. */
        end for
        $b[i] := b[i] - A[i,k] \times y[k]$;
        $A[i,k] := 0$;
    end for
end for
```

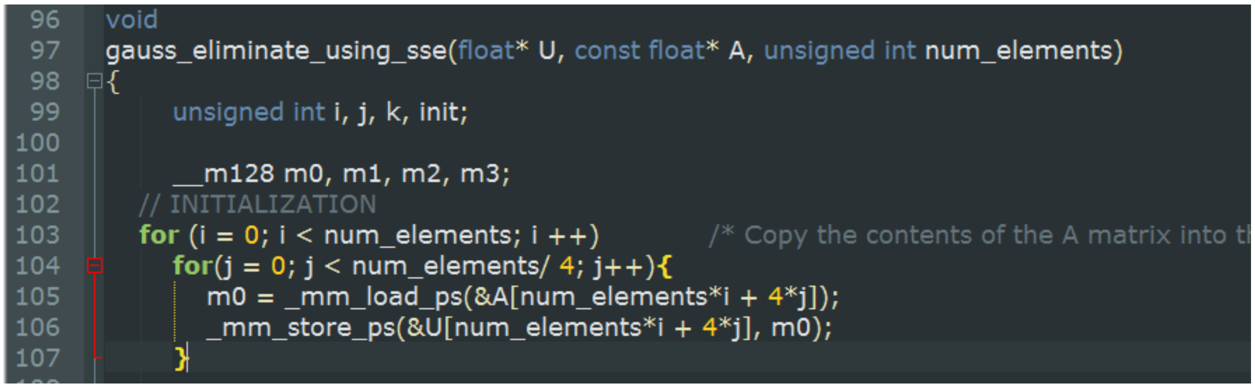
Parallelization approach:

To begin the SSE Assignment, we had to consider that the address locations for allocating space on the virtual address space has to be 16 byte aligned, as per SSE's requirements when working with a 128 bit XMM registers and attempting to store 4 floats which are 4 bytes each, or 32 bits. 4 floats stored into an XMM register will allow for the simultaneous operation of 4 floats in parallel. By creating 16 byte aligned elements in each Matrix structure for Gaussian Elimination, parallelization of each row in the division and elimination steps can be parallelized by storing 4 row elements each iteration of the j loop, and performing the necessary division and elimination operations on those stored floats held in each XMM registers, and loaded back into the Matrix upon successful operation.

Compile with **gcc -o gauss_eliminate gauss_eliminate.c compute_gold.c -std=c99 -O3 -lm**

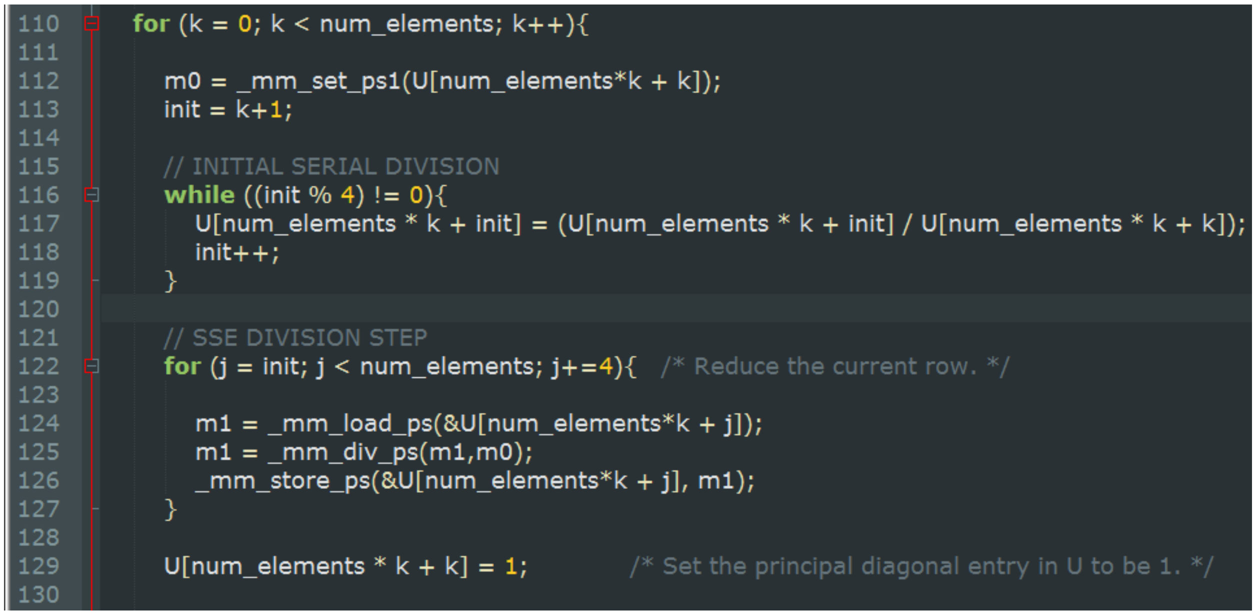
SSE code explanation:

Division and elimination step were both included in the **gauss_eliminate_using_sse()** function, the screenshots of which are included and described below.



```
 96  void
 97  gauss_eliminate_using_sse(float* U, const float* A, unsigned int num_elements)
 98  {
 99          unsigned int i, j, k, init;
100
101          __m128 m0, m1, m2, m3;
102      // INITIALIZATION
103      for (i = 0; i < num_elements; i ++)          /* Copy the contents of the A matrix into t
104          for(j = 0; j < num_elements/ 4; j++){
105              m0 = _mm_load_ps(&A[num_elements*i + 4*j]);
106              _mm_store_ps(&U[num_elements*i + 4*j], m0);
107          }
```

The above screenshot shows that the function accepts pointers to the elements of matrices U and A, as well as the number of elements. The argument types are the same as in the **compute_gold()** function provided by Dr. Kandasamy. On line 101, four __m128 variables are declared. Then, on line 103, the contents of matrix A are copied into matrix U. The matrices were created such that they are byte aligned (using posix_memalign() function), and since the number of elements in each row is divisible by 4, it is guaranteed that we can access every 4th element and utilize SSE registers to speed up the copying process.



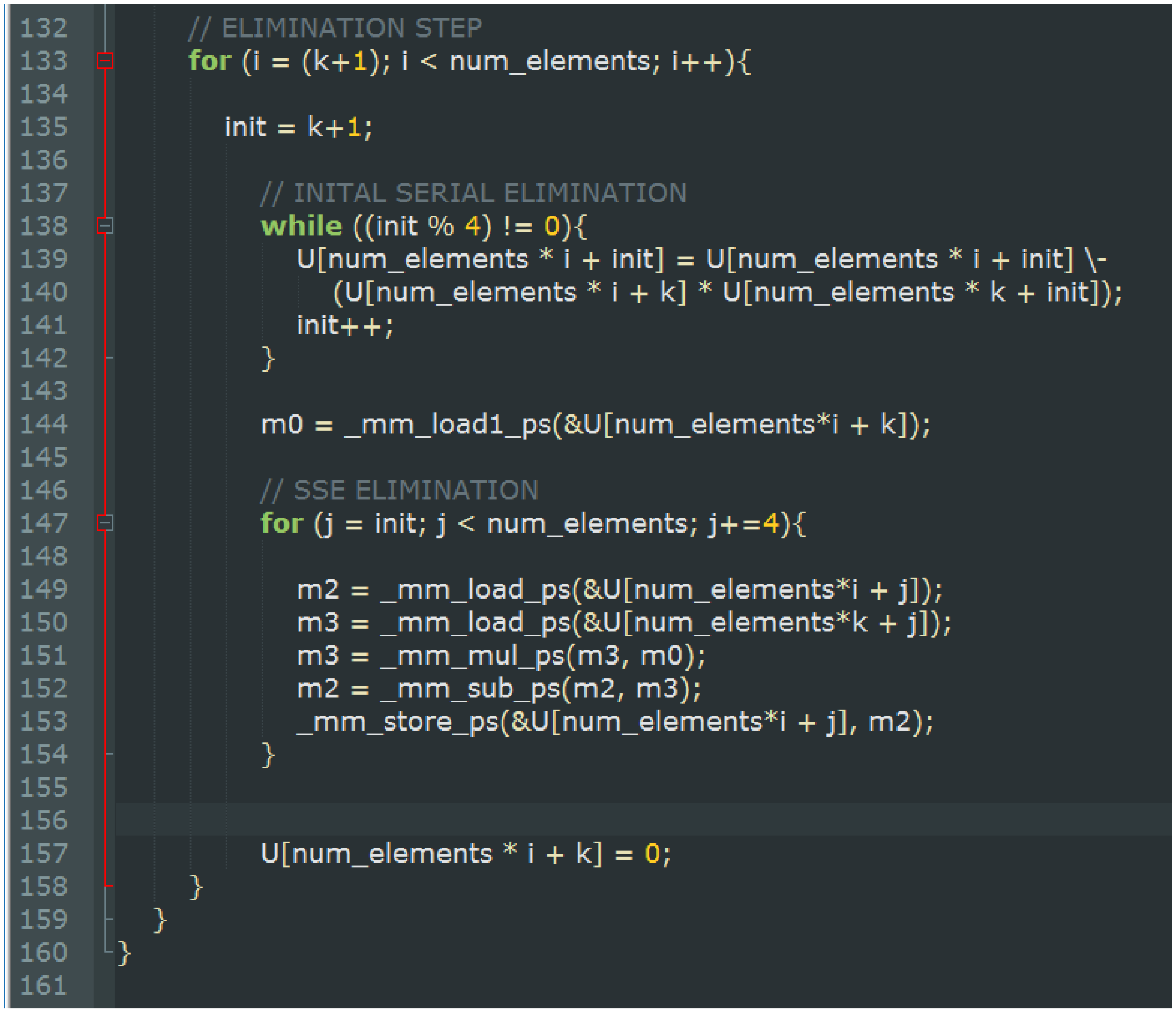
```
110      for (k = 0; k < num_elements; k++){
111
112          m0 = _mm_set_ps1(U[num_elements*k + k]);
113          init = k+1;
114
115          // INITIAL SERIAL DIVISION
116          while ((init % 4) != 0){
117              U[num_elements * k + init] = (U[num_elements * k + init] / U[num_elements * k + k]);
118              init++;
119          }
120
121          // SSE DIVISION STEP
122          for (j = init; j < num_elements; j+=4){   /* Reduce the current row. */
123
124              m1 = _mm_load_ps(&U[num_elements*k + j]);
125              m1 = _mm_div_ps(m1,m0);
126              _mm_store_ps(&U[num_elements*k + j], m1);
127          }
128
129          U[num_elements * k + k] = 1;          /* Set the principal diagonal entry in U to be 1. */
130
```

On line 110, the outermost loop with the counter **k** (like in the pseudo-code algorithm description) is started. In the division step, each element of a given row is divided by the corresponding pivot element. Divisions are performed in groups of 4 (128-bit SSE registers fit 4 floats at a time), so the pivot element needs to be loaded into all 4 words of **m0**. This is achieved with the function **_mm_set_ps1()**.

It was necessary to make sure that the number of elements in a given row that are divided using SSE is divisible by 4. For this reason, the while-loop on line 116 was included. It performs the division step serially until an element whose index is divisible by 4 is reached. This index is stored in the **init** variable. The rest of the division step is done with SSE in the for-loop starting on line 122.

There are three statements in the for loop. The first one on line 124 loads four floats from the current matrix row into **m1**. The next line divides these four floats by the pivot element of the given row. The last command in the for loop stores the division back into the result matrix U.

Finally, the principal diagonal entry in U is set to 1, as on line 8 of the pseudocode algorithm.

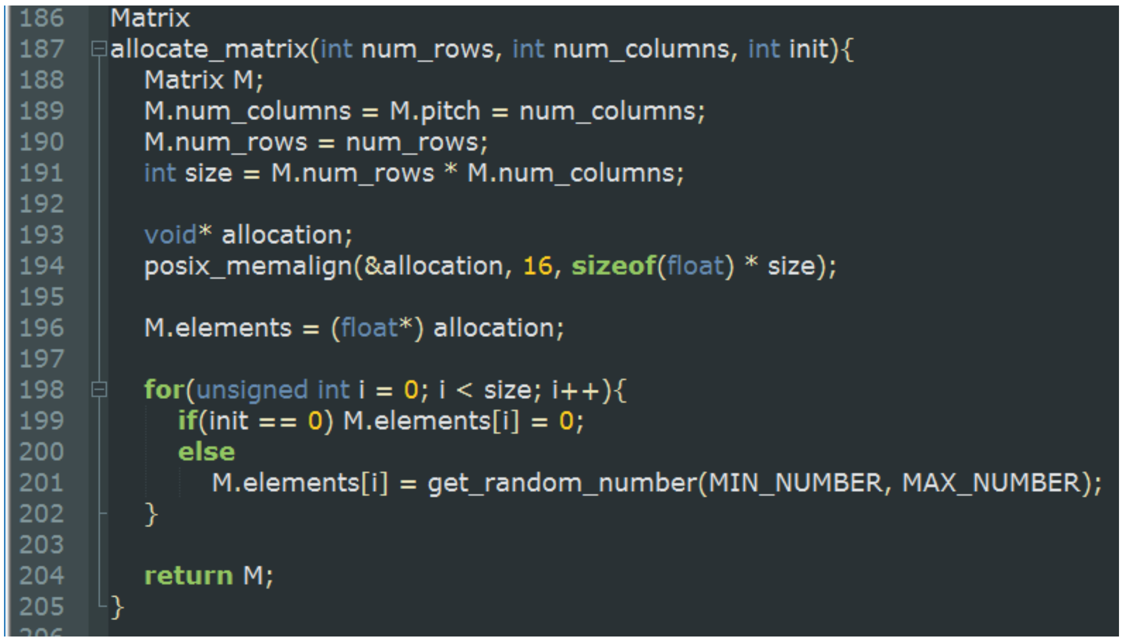


```
132         // ELIMINATION STEP
133         for (i = (k+1); i < num_elements; i++){
134 
135             init = k+1;
136 
137                 // INITAL SERIAL ELIMINATION
138                 while ((init % 4) != 0){
139                     U[num_elements * i + init] = U[num_elements * i + init] \-
140                         (U[num_elements * i + k] * U[num_elements * k + init]);
141                     init++;
142                 }
143 
144                 m0 = _mm_load1_ps(&U[num_elements*i + k]);
145 
146                 // SSE ELIMINATION
147                 for (j = init; j < num_elements; j+=4){
148 
149                     m2 = _mm_load_ps(&U[num_elements*i + j]);
150                     m3 = _mm_load_ps(&U[num_elements*k + j]);
151                     m3 = _mm_mul_ps(m3, m0);
152                     m2 = _mm_sub_ps(m2, m3);
153                     _mm_store_ps(&U[num_elements*i + j], m2);
154                 }
155 
156 
157                 U[num_elements * i + k] = 0;
158         }
159     }
160 }
161 
```

Much the same ideas that were used in parallelizing the division step, the elimination step, shown in the above screenshot, also had a conditional while loop statement on line 146, which checks that the current row index is divisible by 4, and if so, continue the elimination execution serially. This index is not parallelizable, and we must wait until the starting row index is byte aligned correctly by checking until it is divisible by 4, and then continue SSE execution using the variable init, which will now be divisible by 4 when it exits the while loop. At the beginning of each i iteration, we load U[num_elements*i +k] by using **_mm_load_ps** into a 128 bit XMM register for use later inside the j loop. Inside the j loop we also load U[num_elements*i + k] and U[num_elements*k+j] into registers to perform the multiplication and subtraction operations necessary for the elimination algorithm. Once fully computed and stored into an XMM register, this value is stored back into the U Matrix at address U[num_elements*i + j].

Modification of the allocate_matrix() function:

In order to use SSE for matrix elimination, the **allocate_matrix()** function was changed such that the memory allocated to the matrices is 16 byte aligned. Instead of **malloc()**, **posix_memalign()** was used to allocate memory. This is shown on lines 193-196 of the code snipped below.



```
186  Matrix
187  allocate_matrix(int num_rows, int num_columns, int init){
188      Matrix M;
189      M.num_columns = M.pitch = num_columns;
190      M.num_rows = num_rows;
191      int size = M.num_rows * M.num_columns;
192  
193      void* allocation;
194      posix_memalign(&allocation, 16, sizeof(float) * size);
195  
196      M.elements = (float*) allocation;
197  
198      for(unsigned int i = 0; i < size; i++){
199          if(init == 0) M.elements[i] = 0;
200          else
201              M.elements[i] = get_random_number(MIN_NUMBER, MAX_NUMBER);
202      }
203  
204      return M;
205  }
```

Result:

The following table shows serial time, SSE run time, and the speedup for 4 different matrix sizes. Although the assignment instructions specified that the matrix is guaranteed to be 2048x2048 elements, we made our code more flexible so that it can handle other matrix sizes as long as they are divisible by 4.

Table 1: Run Time Comparison between Serial and SSE versions of the program

Matrix size	**Time Serial (s)**	**Time SSE (s)**	**Speedup**
512x512	0.09	0.02	4.50
1024x1024	0.57	0.16	3.56
2048x2048	3.76	1.3	2.89
4096x4096	31.86	13.05	2.44

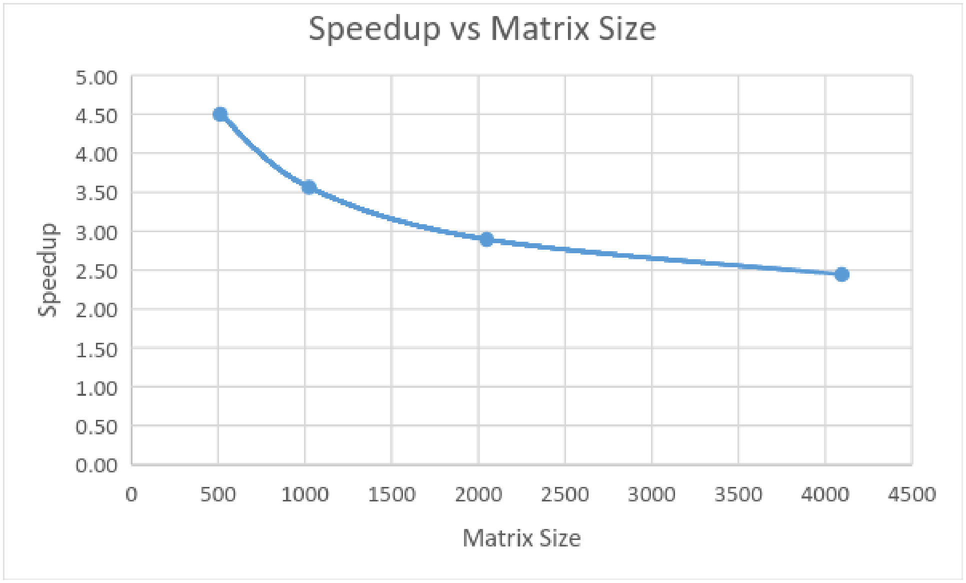


Figure 1: Speedup vs Matrix Size