

System Engineering

Report 2 : Matrix-Vector Multiplication with AVX instructions in C/C++

DENAMGANAÏ Kevin Yandoka - 2160104105

5 février 2017

TABLE DES MATIÈRES

1	ALGORITHMS	3
1.1	C/C++ Naive Algorithm	3
1.2	C/C++ Unrolled	3
1.3	AVX - Implementation 1	4
1.4	AVX - Implementation 2	4
1.5	AVX+FMA - Implementation with Fuse Multiply and Add	5
2	RESULTS	7
3	SOURCE CODE	9

ALGORITHMS

1.1 C/C++ NAIVE ALGORITHM

This first algorithm is a naive implementation of the matrix-vector multiplication formulae for dimensions N :

$$y_i = \sum_{k=0}^{N-1} A_{i,k} x_k \quad (1)$$

Here is the code source :

```

1 void multCNaive(double *A, double *x, double *y, int N)
2 {
3     int i,k;
4
5     for(i=0;i<N;i++)
6     {
7         y[i]=0;
8         for(k=0;k<N;k++)
9         {
10             y[i] += A[i*N+k]*x[k];
11         }
12     }
13 }
14
```

1.2 C/C++ UNROLLED

This algorithm is based on the previous formulae (1), but the accumulation is not done one element at a time but four elements at the time :

```

1 void multCUnrolled(double *A, double *x, double *y, int N)
2 {
3     int d=4;
4     int q = N/d;
5     int r = N%d;
6     double *pa = A;
7     double *py = y;
8
9     for(size_t i=0;i<N;i++)
10    {
11        pa = A+i*N;
12        double *px = x;
13
14        (*py)=0;
15        for(size_t k=0;k<q;k++)
16        {
17            (*py) += *(pa) * *(px)
18                    + *(pa+1) * *(px+1)
19                    + *(pa+2) * *(px+2)
20                    + *(pa+3) * *(px+3);
21
22            px+=d;
23            pa+=d;
24        }
25
26        for(size_t rr=r;rr--;)
27        {
28            (*py) += *(pa) * *(px);
29            px++;
30            pa++;
31        }
32
33        py++;
34    }
35 }
36
```

Given the results in the following chapter, I fear that the compiler *gcc* has made automatic optimization to the code at compilation time, though...

1.3 AVX - IMPLEMENTATION 1

This algorithm is almost a direct rendition of the previous one, while using AVX instructions. Because of the possibility to perform horizontal addition, with the function *_mm256_hadd_pd()*, on two *_m256d* registers at a time, the algorithm is no longer performing the accumulation four elements at a time but eight elements at a time.

```

1 void multAVX1(double *A, double *x, double *y, int N)
2 {
3     int d=4;
4     int times = 2;
5     int q = N/(d*times);
6     int r = N%(d*times);
7     double *pa = A;
8     double *py = y;
9     double *ytemp = (double*)aligned_alloc(64,d*sizeof(double));
10
11     for(size_t i=0;i<N;i++)
12     {
13         pa = A+i*N;
14         double *px = x;
15
16         *(py)=0;
17
18         for(size_t k=0;k<q;k++)
19         {
20             __m256d AA1,AA2,xx1,xx2,yy;
21             AA1 = _mm256_load_pd(pa);
22             xx1 = _mm256_load_pd(px);
23             AA2 = _mm256_load_pd(pa+d);
24             xx2 = _mm256_load_pd(px+d);
25
26             yy = _mm256_hadd_pd(_mm256_mul_pd(AA1,xx1),_mm256_mul_pd(AA2,xx2));
27             _mm256_store_pd(ytemp,yy);
28
29             *(py) += ytemp[0]+ytemp[1]+ytemp[2]+ytemp[3];
30
31
32             px+=times*d;
33             pa+=times*d;
34         }
35
36         for(size_t rr=r;rr--;)
37         {
38             (*py) += *(pa) * *(px);
39             px++;
40             pa++;
41         }
42
43         py++;
44     }
45     free(ytemp);
46 }

```

1.4 AVX - IMPLEMENTATION 2

This algorithm, on the contrary of the previous one, is the direct rendition of the unrolled algorithm with AVX instructions. Horizontal addition is used only on half of the datas, this time.

```

1 void multAVX2(double *A, double *x, double *y, int N)
2 {
3     int d=4;
4     int q = N/d;
5     int r = N%d;
6     double *pa = A;
7     double *py = y;
8     double *px = x;
9     __m256d xx;
10    __m256d AA;
11    __m256d ty;
12
13    for(size_t i=0;i<N;i++)
14    {
15
16        *(py)=0;

```

```

17  px = x;
18
19  ty = _mm256_set1_pd( *py );
20
21  for( size_t k=0;k<q;k++)
22  {
23      AA = _mm256_load_pd(pa);
24      xx = _mm256_load_pd(px);
25
26      ty = _mm256_add_pd( ty ,_mm256_mul_pd(AA,xx) );
27
28      pa+=d;
29      px+=d;
30  }
31
32  double dty[4];
33
34  ty = _mm256_hadd_pd( ty , ty );
35  _mm256_store_pd(dty , ty );
36
37  *(py) += dty[0]+dty[1];
38
39
40  for( size_t rr=r;rr--;)
41  {
42      *(py) += *(pa++) * *(px++);
43  }
44
45  py++;
46  }
47
48  }

```

1.5 AVX+FMA - IMPLEMENTATION WITH FUSE MULTIPLY AND ADD

This algorithm makes the better out of the architecture of the processors and its ability to perform accumulation and multiplication at the same time. It uses the FMA's AVX instruction instead of the two instructions for the multiplication and the addition.

```

1  void multAVX2FMA( double *A, double *x, double *y, int N)
2  {
3      int d=4;
4      int q = N/d;
5      int r = N%d;
6      double *pa = A;
7      double *py = y;
8      double *px = x;
9      __m256d xx;
10     __m256d AA;
11     __m256d ty;
12
13     for( size_t i=0;i<N;i++)
14     {
15
16         *(py)=0;
17         px = x;
18
19         ty = _mm256_set1_pd( *py );
20
21         for( size_t k=0;k<q;k++)
22         {
23             AA = _mm256_load_pd(pa);
24             xx = _mm256_load_pd(px);
25
26             //ty = _mm256_add_pd( ty ,_mm256_mul_pd(AA,xx) );
27             ty = _mm256_fmadd_pd( AA, xx, ty );
28
29             pa+=d;
30             px+=d;
31         }
32
33         double dty[4];
34
35         ty = _mm256_hadd_pd( ty , ty );
36         _mm256_store_pd(dty , ty );
37
38         *(py) += dty[0]+dty[1];
39
40
41         for( size_t rr=r;rr--;)

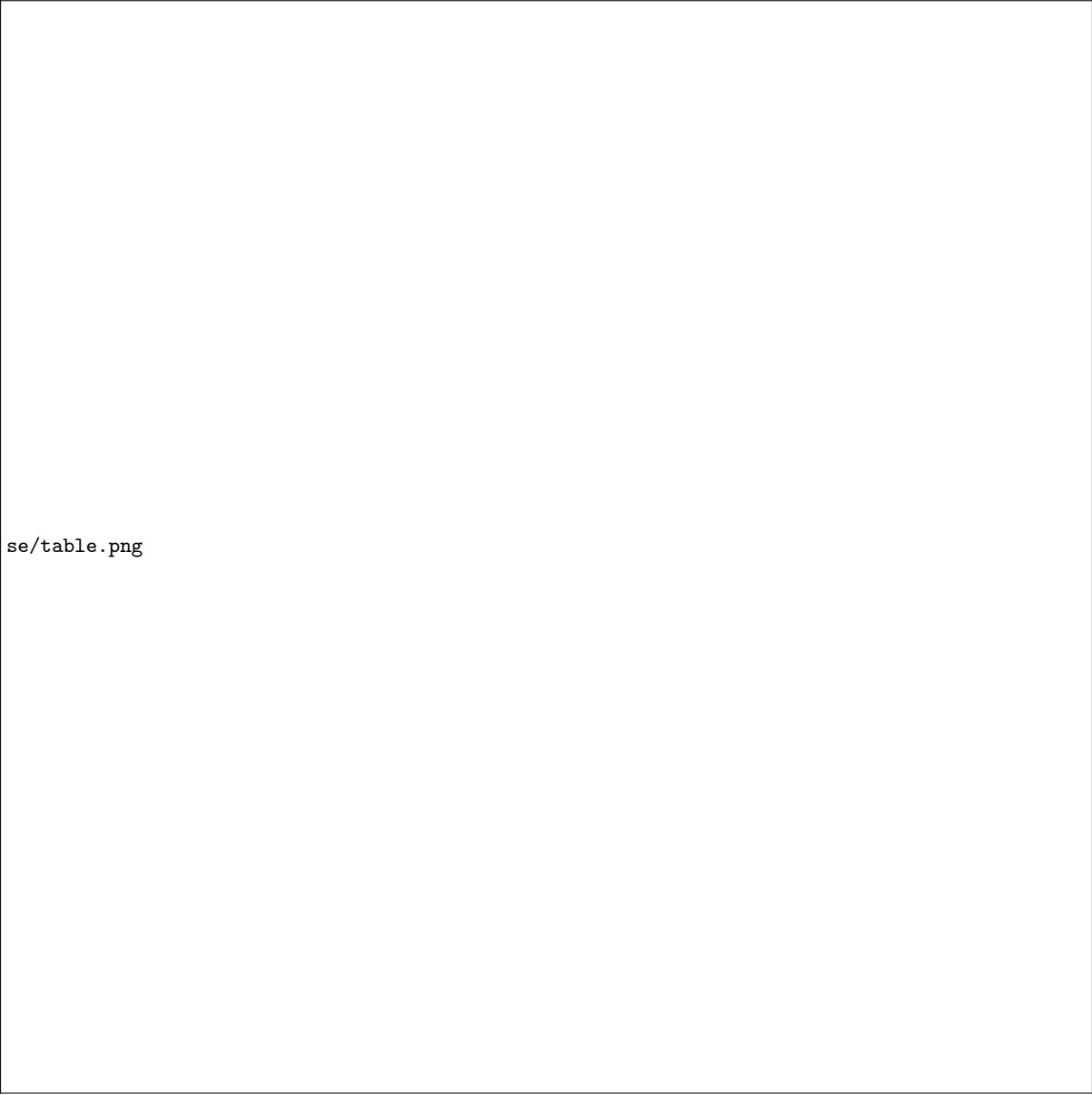
```

```
42 {  
43     *(py) += *(pa++) * *(px++);  
44 }  
45  
46 py++;  
47 }  
48  
49 }
```

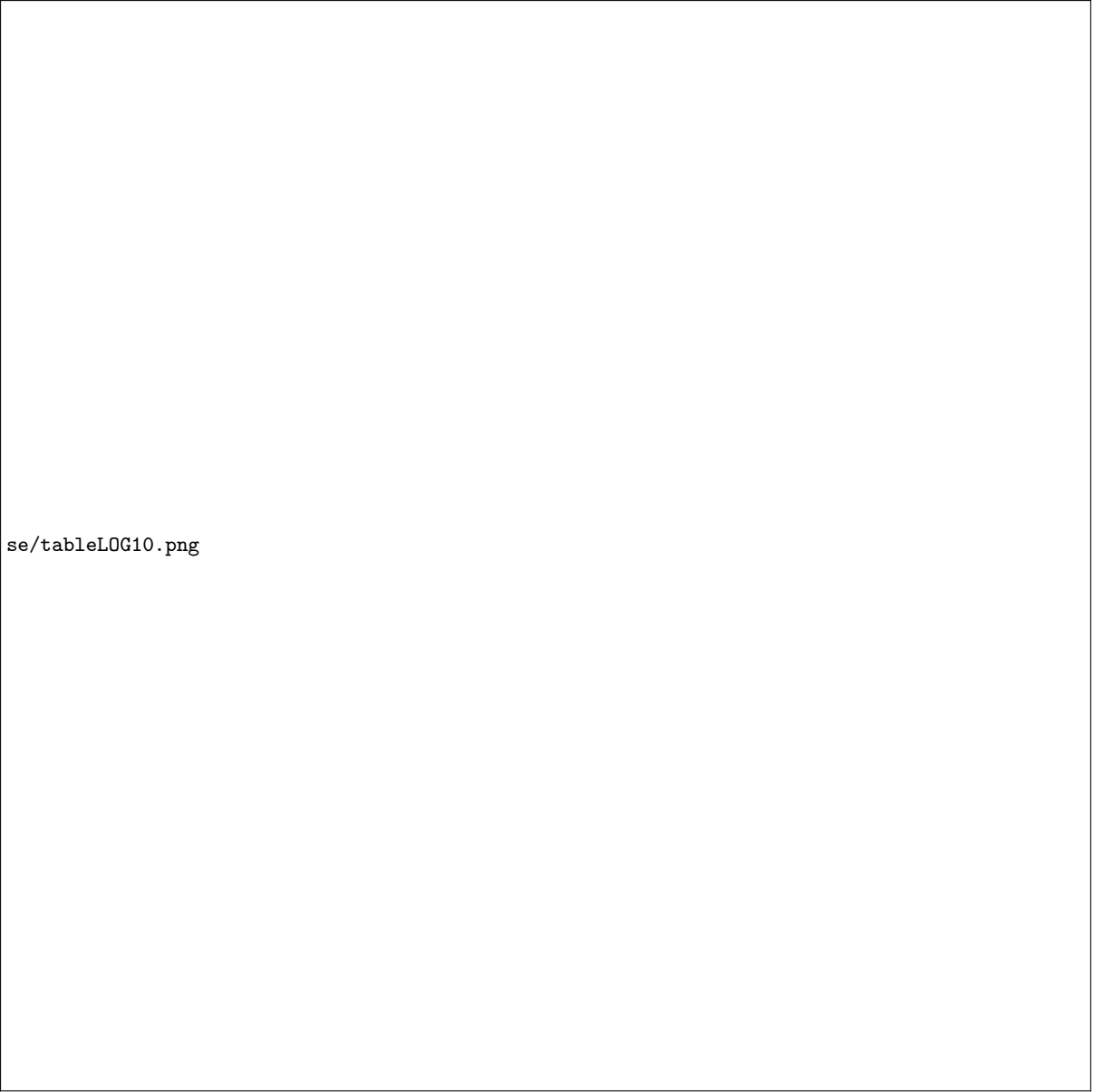
RESULTS

As explained earlier, the algorithm **C/C++ Unrolled** might not be reliable given the possibility of the compiler to have auto-optimized the code at compilation time.

So, the main comparaison has to be undertaken between the **C/C++ Naive** implementation and the others AVX-based algorithms, in the following charts :



se/table.png



se/tableLOG10.png

The main result is an improvement of the computation time of almost a ratio of four between the naive algorithm and the the AVX2 and AVX+FMA implementations.

SOURCE CODE

You can find at the following the project and all its related details :

Here is the source code of the main program to produce the charts presented earlier (after almost two hours of computation...) :

```

1 #include <immintrin.h>
2 #include <time.h>
3 #include <iostream>
4 #include "../RunningStats/RunningStats.h"
5
6
7 void multCNaive(double *A, double *x, double *y, int N)
8 {
9     int i, j, k;
10
11     for(i=0; i<N; i++)
12     {
13         y[i]=0;
14         for(k=0; k<N; k++)
15         {
16             y[i] += A[i*N+k]*x[k];
17         }
18     }
19 }
20
21
22
23 void multCUnrolled(double *A, double *x, double *y, int N)
24 {
25     int d=4;
26     int q = N/d;
27     int r = N%d;
28     double *pa = A;
29     double *py = y;
30
31     for(size_t i=0; i<N; i++)
32     {
33         pa = A+i*N;
34         double *px = x;
35
36         (*py)=0;
37         for(size_t k=0; k<q; k++)
38         {
39             (*py) += *(pa) * *(px)
40                     + *(pa+1) * *(px+1)
41                     + *(pa+2) * *(px+2)
42                     + *(pa+3) * *(px+3);
43
44             px+=d;
45             pa+=d;
46         }
47
48         for(size_t rr=r; rr--;)
49         {
50             (*py) += *(pa) * *(px);
51             px++;
52             pa++;
53         }
54
55         py++;
56     }
57 }
58
59
60 void multAVX1(double *A, double *x, double *y, int N)
61 {

```

```

63  int d=4;
64  int times = 2;
65  int q = N/(d*times);
66  int r = N%(d*times);
67  double *pa = A;
68  double *py = y;
69  double *ytemp = (double*)aligned_alloc(64,d*sizeof(double));
70
71  for( size_t i=0;i<N;i++)
72  {
73      pa = A+i*N;
74      double *px = x;
75
76      *(py)=0;
77
78      for( size_t k=0;k<q;k++)
79      {
80          __m256d AA1,AA2,xx1,xx2,yy;
81          AA1 = _mm256_load_pd(pa);
82          xx1 = _mm256_load_pd(px);
83          AA2 = _mm256_load_pd(pa+d);
84          xx2 = _mm256_load_pd(px+d);
85
86          yy = _mm256_hadd_pd( _mm256_mul_pd(AA1,xx1), _mm256_mul_pd(AA2,xx2) );
87          _mm256_store_pd(ytemp, yy);
88
89          *(py) += ytemp[0]+ytemp[1]+ytemp[2]+ytemp[3];
90
91
92          px+=times*d;
93          pa+=times*d;
94      }
95
96      for( size_t rr=r;rr--;)
97      {
98          (*py) += *(pa) * *(px);
99          px++;
100         pa++;
101     }
102
103     py++;
104 }
105
106 free(ytemp);
107 }
108
109 void multAVX2(double *A, double *x, double *y, int N)
110 {
111     int d=4;
112     int q = N/d;
113     int r = N%d;
114     double *pa = A;
115     double *py = y;
116     double *px = x;
117     __m256d xx;
118     __m256d AA;
119     __m256d ty;
120
121     for( size_t i=0;i<N;i++)
122     {
123
124         *(py)=0;
125         px = x;
126
127         ty = _mm256_set1_pd( *py );
128
129         for( size_t k=0;k<q;k++)
130         {
131             AA = _mm256_load_pd(pa);
132             xx = _mm256_load_pd(px);
133
134             ty = _mm256_add_pd( ty, _mm256_mul_pd(AA,xx) );
135
136             pa+=d;
137             px+=d;
138         }
139
140         double dty[4];
141
142         ty = _mm256_hadd_pd( ty, ty );
143         _mm256_store_pd(dty, ty);
144
145         //*(py) += dty[0]+dty[1]+dty[2]+dty[3];

```

```

146 *(py) += dty[0]+dty[1];
147
148 for(size_t rr=r;rr--;)
149 {
150     *(py) += *(pa++) * *(px++);
151 }
152
153 py++;
154 }
155 }
156
157 }
158
159 void multAVX2FMA(double *A, double *x, double *y, int N)
160 {
161     int d=4;
162     int q = N/d;
163     int r = N%d;
164     double *pa = A;
165     double *py = y;
166     double *px = x;
167     __m256d xx;
168     __m256d AA;
169     __m256d ty;
170
171     for(size_t i=0;i<N;i++)
172     {
173
174         *(py)=0;
175         px = x;
176
177         ty = _mm256_set1_pd( *py );
178
179         for(size_t k=0;k<q;k++)
180         {
181             AA = _mm256_load_pd(pa);
182             xx = _mm256_load_pd(px);
183
184             //ty = _mm256_add_pd( ty, _mm256_mul_pd(AA,xx) );
185             ty = _mm256_fmadd_pd( AA, xx, ty );
186
187             pa+=d;
188             px+=d;
189         }
190
191         double dty[4];
192
193         ty = _mm256_hadd_pd( ty, ty );
194         _mm256_store_pd(dty, ty);
195
196         //*(py) += dty[0]+dty[1]+dty[2]+dty[3];
197         *(py) += dty[0]+dty[1];
198
199
200         for(size_t rr=r;rr--;)
201         {
202             *(py) += *(pa++) * *(px++);
203         }
204
205         py++;
206     }
207 }
208
209 }
210
211
212 void print(double *x, size_t N)
213 {
214     for(size_t i=0;i<N;i++)
215     {
216         std::cout << x[i];
217
218         if (i!=N-1)
219         {
220             std::cout << " , ";
221         }
222     }
223
224     std::cout << std::endl;
225 }
226
227 }
228

```

```

229 void init(double **A, double **x, double **y, const int& N)
230 {
231     *A = (double*)aligned_alloc(64,N*sizeof(double));
232     *x = (double*)aligned_alloc(64,N*sizeof(double));
233     *y = (double*)aligned_alloc(64,N*sizeof(double));
234
235     int i,j;
236     for(i=0;i<N;i++)
237     {
238         for(j=0;j<N;j++)
239         {
240             (*A)[i*N+j] = float(rand()%10);
241         }
242
243         (*x)[i] = float(rand()%10);
244     }
245 }
246
247
248
249 void destroy(double *A, double *x, double *y)
250 {
251     free(A);
252     free(x);
253     free(y);
254 }
255
256
257 int main(int argc, char* argv[])
258 {
259     srand(time(NULL));
260
261     float factor = 1e6f;
262     bool verbose = false;
263     if(argc>1)
264         verbose = (atoi(argv[1])==1?true:false);
265     else
266         std::cout << " USAGE :: report.output [verbose (int) 0:false / 1:true] [N (unsigned int)]" << std::endl;
267
268     float duration;
269     unsigned int N=4;
270     if(argc>2)
271         N = atoi(argv[2]);
272     else
273         std::cout << " USAGE :: report.output [verbose default=0 (int) 0:false / 1:true] [N default=4 (unsigned int)]" << std::endl;
274
275     double *A = NULL;
276     double *x = NULL;
277     double *y = NULL;
278     clock_t time;
279     std::string pathStats = "./stats";
280     RunningStats<float> rs(pathStats,2);
281     int nbrTimes = 100;
282     int initN = 20;
283     int stepN = 200;
284     //-----
285     for(int loopN=initN;loopN<=N;loopN+=stepN)
286     {
287
288         init(&A,&x,&y,loopN);
289
290         time = clock();
291
292         for(size_t k=nbrTimes;k--;) multCNaive(A,x,y,loopN);
293
294         duration = float(clock()-time)*factor/CLOCKS_PER_SEC/nbrTimes;
295
296         rs.tadd( std::string("C/C++ NAIVE"), duration );
297         std::cout << " NAIVE C/C++ Mult : " << duration << " microseconds." << std::endl;
298         //if(verbose) print(y,N);
299
300         destroy(A,x,y);
301     }
302
303     //-----
304
305     for(int loopN=initN;loopN<=N;loopN+=stepN)
306     {
307
308         init(&A,&x,&y,loopN);
309

```

```

310     time = clock();
311
312     for (size_t k=nrTimes; k--;) multCUnrolled(A, x, y, loopN);
313
314     duration = float(clock() - time) * factor / CLOCKS_PER_SEC / nrTimes;
315
316     rs.tadd( std::string("C/C++ Unrolled"), duration );
317     std::cout << " Unrolled C/C++ Mult : " << duration << " microseconds." << std::endl;
318     //if(verbose) print(y,N);
319
320     destroy(A, x, y);
321 }
322
323 //-----
324
325 for (int loopN=initN; loopN<=N; loopN+=stepN)
326 {
327     init(&A, &x, &y, loopN);
328
329     time = clock();
330
331     for (size_t k=nrTimes; k--;) multAVX1(A, x, y, loopN);
332
333     duration = float(clock() - time) * factor / CLOCKS_PER_SEC / nrTimes;
334
335     rs.tadd( std::string("AVX1"), duration );
336     std::cout << " AVX1 : " << duration << " microseconds." << std::endl;
337     //if(verbose) print(y,N);
338
339     destroy(A, x, y);
340 }
341
342 //-----
343
344 for (int loopN=initN; loopN<=N; loopN+=stepN)
345 {
346     init(&A, &x, &y, loopN);
347
348     time = clock();
349
350     for (size_t k=nrTimes; k--;) multAVX2(A, x, y, loopN);
351
352     duration = float(clock() - time) * factor / CLOCKS_PER_SEC / nrTimes;
353
354     rs.tadd( std::string("AVX2"), duration );
355     std::cout << " AVX2 : " << duration << " microseconds." << std::endl;
356     //if(verbose) print(y,N);
357
358     destroy(A, x, y);
359 }
360
361 //-----
362
363 for (int loopN=initN; loopN<=N; loopN+=stepN)
364 {
365     init(&A, &x, &y, loopN);
366
367     time = clock();
368
369     for (size_t k=nrTimes; k--;) multAVX2FMA(A, x, y, loopN);
370
371     duration = float(clock() - time) * factor / CLOCKS_PER_SEC / nrTimes;
372
373     rs.tadd( std::string("AVX+FMA"), duration );
374     std::cout << " AVX+FMA : " << duration << " microseconds." << std::endl;
375     //if(verbose) print(y,N);
376
377     destroy(A, x, y);
378 }
379
380 return 0;
381 }

```