# 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

## 1.1 C/C++ NAIVE ALGORITHM

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

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

Here is the code source :

```
void multCNaive(double *A, double *x, double *y, int N)
{
  int i,k;

  for(i=0;i<N;i++)
  {
    y[i]=0;
    for(k=0;k<N;k++)
    {
      y[i] += A[i*N+k]*x[k];
    }
  }

}
```

## 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 :

```
void multCUnrolled(double *A, double *x, double *y, int N)
{
  int d=4;
  int q = N/d;
  int r = N%d;
  double *pa = A;
  double *py = y;

  for(size_t i=0;i<N;i++)
  {
    pa = A+i*N;
    double *px = x;

    (*py)=0;
    for(size_t k=0;k<q;k++)
    {
      (*py) += *(pa) * *(px)
            + *(pa+1) * *(px+1)
            + *(pa+2) * *(px+2)
            + *(pa+3) * *(px+3);

      px+=d;
      pa+=d;
    }

    for(size_t rr=r;rr--;)
    {
      (*py) += *(pa) * *(px);
      px++;
      pa++;
    }

    py++;
  }

}
```

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.

```c
void multAVX1(double *A, double *x, double *y, int N)
{
  int d=4;
  int times = 2;
  int q = N/(d*times);
  int r = N%(d*times);
  double *pa = A;
  double *py = y;
  double *ytemp = (double*)aligned_alloc(64,d*sizeof(double));

  for(size_t i=0;i<N;i++)
  {
    pa = A+i*N;
    double *px = x;

    *(py)=0;

    for(size_t k=0;k<q;k++)
    {
      __m256d AA1,AA2,xx1,xx2,yy;
      AA1 = _mm256_load_pd(pa);
      xx1 = _mm256_load_pd(px);
      AA2 = _mm256_load_pd(pa+d);
      xx2 = _mm256_load_pd(px+d);

      yy = _mm256_hadd_pd( _mm256_mul_pd(AA1,xx1), _mm256_mul_pd(AA2,xx2) );
      _mm256_store_pd(ytemp, yy);

      *(py) += ytemp[0]+ytemp[1]+ytemp[2]+ytemp[3];


      px+=times*d;
      pa+=times*d;
    }

    for(size_t rr=r;rr--;)
    {
      (*py) += *(pa) * *(px);
      px++;
      pa++;
    }

    py++;
  }

  free(ytemp);
}
```

### 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.

```c
void multAVX2(double *A, double *x, double *y, int N)
{
  int d=4;
  int q = N/d;
  int r = N%d;
  double *pa = A;
  double *py = y;
  double *px = x;
  __m256d xx;
  __m256d AA;
  __m256d ty;

  for(size_t i=0;i<N;i++)
  {

    *(py)=0;
```

```
    px = x;

    ty = _mm256_set1_pd( *py );

    for(size_t k=0;k<q;k++)
    {
      AA = _mm256_load_pd(pa);
      xx = _mm256_load_pd(px);

      ty = _mm256_add_pd( ty,_mm256_mul_pd(AA,xx) );

      pa+=d;
      px+=d;
    }

    double dty[4];

    ty = _mm256_hadd_pd( ty, ty);
    _mm256_store_pd(dty, ty);

    *(py) += dty[0]+dty[1];


    for(size_t rr=r;rr--;)
    {
      *(py) += *(pa++) * *(px++);
    }

    py++;
  }

}
```

## 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.

```
void multAVX2FMA(double *A, double *x, double *y, int N)
{
  int d=4;
  int q = N/d;
  int r = N%d;
  double *pa = A;
  double *py = y;
  double *px = x;
  __m256d xx;
  __m256d AA;
  __m256d ty;

  for(size_t i=0;i<N;i++)
  {

    *(py)=0;
    px = x;

    ty = _mm256_set1_pd( *py );

    for(size_t k=0;k<q;k++)
    {
      AA = _mm256_load_pd(pa);
      xx = _mm256_load_pd(px);

      //ty = _mm256_add_pd( ty,_mm256_mul_pd(AA,xx) );
      ty = _mm256_fmadd_pd( AA, xx, ty );

      pa+=d;
      px+=d;
    }

    double dty[4];

    ty = _mm256_hadd_pd( ty, ty);
    _mm256_store_pd(dty, ty);

    *(py) += dty[0]+dty[1];


    for(size_t rr=r;rr--;)
```

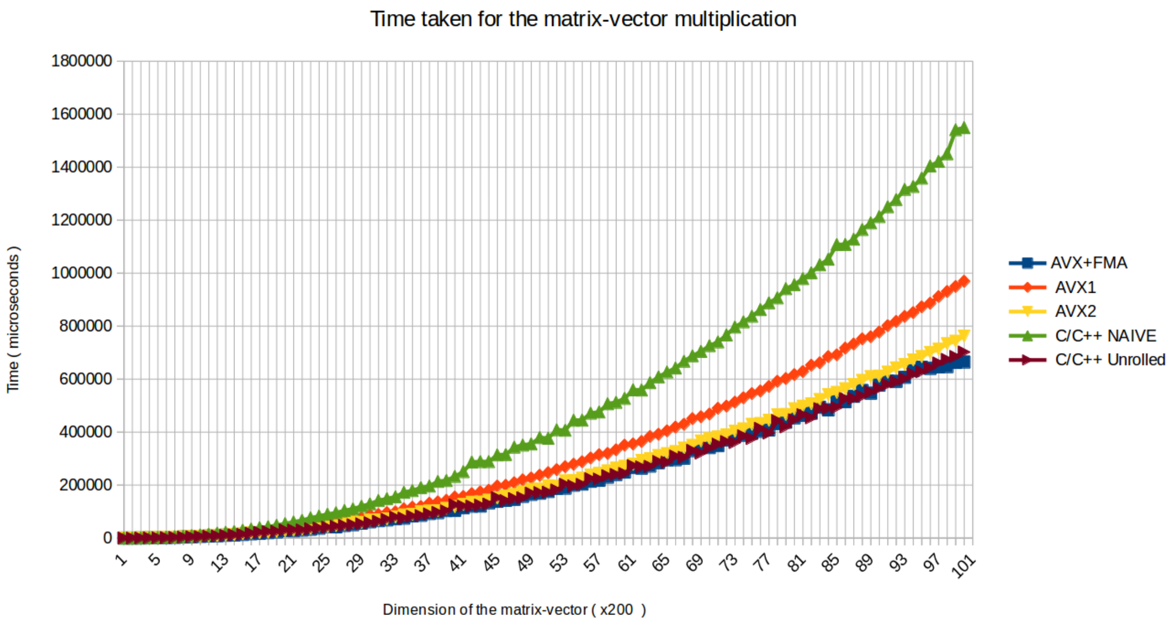
```
      {
        *(py) += *(pa++) * *(px++);
      }

      py++;
   }

}
```

# 2

# 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 :




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

# 3

# SOURCE CODE

You can find at the following the project and all its related details :
`https://github.com/Near32/AVX-Optimization.git`

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

```
#include <immintrin.h>
#include <time.h>
#include <iostream>
#include "./RunningStats/RunningStats.h"


void multCNaive(double *A, double *x, double *y, int N)
{
  int i,j,k;

  for(i=0;i<N;i++)
  {
    y[i]=0;
    for(k=0;k<N;k++)
    {
      y[i] += A[i*N+k]*x[k];
    }
  }

}


void multCUnrolled(double *A, double *x, double *y, int N)
{
  int d=4;
  int q = N/d;
  int r = N%d;
  double *pa = A;
  double *py = y;

  for(size_t i=0;i<N;i++)
  {
    pa = A+i*N;
    double *px = x;

    (*py)=0;
    for(size_t k=0;k<q;k++)
    {
      (*py) += *(pa) * *(px)
            + *(pa+1) * *(px+1)
            + *(pa+2) * *(px+2)
            + *(pa+3) * *(px+3);

      px+=d;
      pa+=d;
    }

    for(size_t rr=r;rr--;)
    {
      (*py) += *(pa) * *(px);
      px++;
      pa++;
    }

    py++;
  }

}


void multAVX1(double *A, double *x, double *y, int N)
```

```
{
  int d=4;
  int times = 2;
  int q = N/(d*times);
  int r = N%(d*times);
  double *pa = A;
  double *py = y;
  double *ytemp = (double*)aligned_alloc(64,d*sizeof(double));

  for(size_t i=0;i<N;i++)
  {
    pa = A+i*N;
    double *px = x;

    *(py)=0;

    for(size_t k=0;k<q;k++)
    {
      __m256d AA1,AA2,xx1,xx2,yy;
      AA1 = _mm256_load_pd(pa);
      xx1 = _mm256_load_pd(px);
      AA2 = _mm256_load_pd(pa+d);
      xx2 = _mm256_load_pd(px+d);

      yy = _mm256_hadd_pd( _mm256_mul_pd(AA1,xx1), _mm256_mul_pd(AA2,xx2) );
      _mm256_store_pd(ytemp, yy);

      *(py) += ytemp[0]+ytemp[1]+ytemp[2]+ytemp[3];


      px+=times*d;
      pa+=times*d;
    }

    for(size_t rr=r;rr--;)
    {
      (*py) += *(pa) * *(px);
      px++;
      pa++;
    }

    py++;
  }

  free(ytemp);
}

void multAVX2(double *A, double *x, double *y, int N)
{
  int d=4;
  int q = N/d;
  int r = N%d;
  double *pa = A;
  double *py = y;
  double *px = x;
  __m256d xx;
  __m256d AA;
  __m256d ty;

  for(size_t i=0;i<N;i++)
  {

    *(py)=0;
    px = x;

    ty = _mm256_set1_pd( *py );

    for(size_t k=0;k<q;k++)
    {
      AA = _mm256_load_pd(pa);
      xx = _mm256_load_pd(px);

      ty = _mm256_add_pd( ty,_mm256_mul_pd(AA,xx) );

      pa+=d;
      px+=d;
    }

    double dty[4];

    ty = _mm256_hadd_pd( ty, ty);
    _mm256_store_pd(dty, ty);

```

```
    //*(py) += dty[0]+dty[1]+dty[2]+dty[3];
    *(py) += dty[0]+dty[1];


    for(size_t rr=r;rr--;)
    {
      *(py) += *(pa++) * *(px++);
    }

    py++;
  }

}


void multAVX2FMA(double *A, double *x, double *y, int N)
{
  int d=4;
  int q = N/d;
  int r = N%d;
  double *pa = A;
  double *py = y;
  double *px = x;
  __m256d xx;
  __m256d AA;
  __m256d ty;

  for(size_t i=0;i<N;i++)
  {

    *(py)=0;
    px = x;

    ty = _mm256_set1_pd( *py );

    for(size_t k=0;k<q;k++)
    {
      AA = _mm256_load_pd(pa);
      xx = _mm256_load_pd(px);

      //ty = _mm256_add_pd( ty,_mm256_mul_pd(AA,xx) );
      ty = _mm256_fmadd_pd( AA, xx, ty );

      pa+=d;
      px+=d;
    }

    double dty[4];

    ty = _mm256_hadd_pd( ty, ty);
    _mm256_store_pd(dty, ty);

    //*(py) += dty[0]+dty[1]+dty[2]+dty[3];
    *(py) += dty[0]+dty[1];


    for(size_t rr=r;rr--;)
    {
      *(py) += *(pa++) * *(px++);
    }

    py++;
  }

}



void print(double *x, size_t N)
{
  for(size_t i=0;i<N;i++)
  {
    std::cout << x[i];

    if(i!=N-1)
    {
      std::cout << " , ";
    }

  }

  std::cout << std::endl;
}
```

```
void init(double **A, double **x, double **y, const int& N)
{
  *A = (double*)aligned_alloc(64,N*N*sizeof(double));
  *x = (double*)aligned_alloc(64,N*sizeof(double));
  *y = (double*)aligned_alloc(64,N*sizeof(double));


  int i,j;
  for(i=0;i<N;i++)
  {
    for(j=0;j<N;j++)
    {
      (*A)[i*N+j] = float(rand()%10);
    }

    (*x)[i] = float(rand()%10);
  }


}

void destroy(double *A, double *x, double *y)
{
  free(A);
  free(x);
  free(y);
}

int main(int argc, char* argv[])
{
  srand(time(NULL));

  float factor = 1e6f;
  bool verbose = false;
  if(argc>1)
    verbose = (atoi(argv[1])==1?true:false);
  else
    std::cout << " USAGE :: report.output [verbose (int) 0:false / 1:true]  [N (unsigned int)]" << std::endl;

  float duration;
  unsigned int N=4;
  if(argc>2)
    N = atoi(argv[2]);
  else
    std::cout << " USAGE :: report.output [verbose default=0 (int) 0:false / 1:true]  [N default=4 (unsigned int)]" << std::endl;

  double *A = NULL;
  double *x = NULL;
  double *y = NULL;
  clock_t time;
  std::string pathStats = "./stats";
  RunningStats<float> rs(pathStats,2);
  int nbrTimes = 100;
  int initN = 20;
  int stepN = 200;
  //------------------------------------------------------------------------

  for(int loopN=initN;loopN<=N;loopN+=stepN)
  {

    init(&A,&x,&y,loopN);

    time = clock();

    for(size_t k=nbrTimes;k--;) multCNaive(A,x,y,loopN);

    duration = float(clock()-time)*factor/CLOCKS_PER_SEC/nbrTimes;

    rs.tadd( std::string("C/C++ NAIVE"), duration );
    std::cout << " NAIVE C/C++ Mult : " << duration << " microseconds." << std::endl;
    //if(verbose) print(y,N);

    destroy(A,x,y);
  }

  //------------------------------------------------------------------------

  for(int loopN=initN;loopN<=N;loopN+=stepN)
  {

```

```
    init(&A,&x,&y,loopN);

    time = clock();

    for(size_t k=nbrTimes;k--;) multCUnrolled(A,x,y,loopN);

    duration = float(clock()-time)*factor/CLOCKS_PER_SEC/nbrTimes;

    rs.tadd( std::string("C/C++ Unrolled"), duration );
    std::cout << " Unrolled C/C++ Mult : " << duration << " microseconds." << std::endl;
    //if(verbose) print(y,N);

    destroy(A,x,y);
  }

  //--------------------------------------------------------------------------------

  for(int loopN=initN;loopN<=N;loopN+=stepN)
  {

    init(&A,&x,&y,loopN);

    time = clock();

    for(size_t k=nbrTimes;k--;) multAVX1(A,x,y,loopN);

    duration = float(clock()-time)*factor/CLOCKS_PER_SEC/nbrTimes;

    rs.tadd( std::string("AVX1"), duration );
    std::cout << " AVX1 : " << duration << " microseconds." << std::endl;
    //if(verbose) print(y,N);

    destroy(A,x,y);
  }

  //--------------------------------------------------------------------------------

  for(int loopN=initN;loopN<=N;loopN+=stepN)
  {

    init(&A,&x,&y,loopN);

    time = clock();

    for(size_t k=nbrTimes;k--;) multAVX2(A,x,y,loopN);

    duration = float(clock()-time)*factor/CLOCKS_PER_SEC/nbrTimes;

    rs.tadd( std::string("AVX2"), duration );
    std::cout << " AVX2 : " << duration << " microseconds." << std::endl;
    //if(verbose) print(y,N);

    destroy(A,x,y);
  }

  //--------------------------------------------------------------------------------

  for(int loopN=initN;loopN<=N;loopN+=stepN)
  {

    init(&A,&x,&y,loopN);

    time = clock();

    for(size_t k=nbrTimes;k--;) multAVX2FMA(A,x,y,loopN);

    duration = float(clock()-time)*factor/CLOCKS_PER_SEC/nbrTimes;

    rs.tadd( std::string("AVX+FMA"), duration );
    std::cout << " AVX+FMA : " << duration << " microseconds." << std::endl;
    //if(verbose) print(y,N);

    destroy(A,x,y);
  }



  return 0;

}
```