# Benchmark of C++ Libraries for Sparse Matrix Computation

Georg Holzmann
http://grh.mur.at
email: grh@mur.at

August 2007

This report presents benchmarks of C++ scientific computing libraries for small and medium size sparse matrices. Be warned: these benchmarks are very specialized on a neural network like algorithm I had to implement. However, also the initialization time of sparse matrices and a matrix-vector multiplication was measured, which might be of general interest.

**WARNING**
**At the time of its writing this document did not consider the *eigen* library (http://eigen.tuxfamily.org).**
**Please evaluate also this very nice, fast and well maintained library before making any decisions!**

## Contents

# 1 Introduction

Quite a lot open-source libraries exist for scientific computing, which makes it hard to choose between them. Therefore, after communication on various mailing lists, I decided to perform some benchmarks, specialized to the kind of algorithms I had to implement.

Ideally algorithms should be implemented in an object oriented, reuseable way, but of course without a loss of performance. BLAS (Basic Linear Algebra Subprograms - see [1]) and LAPACK (Linear Algebra Package - see [3]) are the standard building blocks for efficient scientific software. Highly optimized BLAS implementations for all relevant hardware architectures are available, traditionally expressed in Fortran routines for scalar, vector and matrix operations (called Level 1, 2 and 3 BLAS).

However, some of the here presented libraries still interface to BLAS without loosing the object oriented concepts of component reusability, general programming, etc. Good performance can be achieved with expression templates, closures and other design patterns (see [4] and [5]).

I have to note that these benchmarks are very special to the algorithms I had to implement (neural network like operations) and should therefore not be seen as a general benchmark of these libraries. In particular I benchmarked a (sparse) matrix-vector multiplication (see 2.2), a neural network like operation (see 2.3) and the initialization time of the sparse matrix (see 2.1). Results to these examples can be found in section 3. A more general benchmark of non-sparse algorithms can be found at `http://projects.opencascade.org/btl/`.

# 2 Benchmarks

I used the C(++) function clock() from ctime.h for the benchmarks (see [2]). As pointed out in [6] the disadvantage with clock() is that the resolution is pretty low. Therefore one has to repeat the code many times, where the data can stay in the cache and so the code runs faster. However, I decided to use that method because in my application I also have to repeat the operations in a similar loop and therefore the resolution was high enough.

The benchmarks were made with floats and doubles with sparse and nonsparse versions of each library, where the term "connectivity" always specifies how many percent of the matrix are non-zero elements.

The y-axis in the plots is usually given in CPU ticks per element. This means the number of clock ticks elapsed between two clock() calls divided through the matrix size N (e.g. divided through 100 for a 100x100 matrix). If one would want to calculate the absolute time needed by an operation (in sec) one could use the following formula:

$$AbsTime = \frac{CPUTicksPerElement * N}{CLOCKS\_PER\_SEC}$$

where CLOCKS_PER_SEC is a macro constant expression of ctime.h, which specifies the relation between a clock tick and a second (on the benchmark system: CLOCKS_PER_SEC=1000000).

As already said, each operation was repeated in a loop for one benchmark (see Table 1 for the exact values). Additionally the whole benchmarks were repeated six times and averaged.

Table 2 shows some information about the benchmark system.

All libraries were compiled without debug mode with g++ (GCC) 4.1.2. The detailed compiler and linker flags can be found in Appendix A, the source code in Appendix B.

| Matrix Size (N) | Nr. of repetitions |
|---|---|
| 10 | 200000 |
| 30 | 100000 |
| 50 | 100000 |
| 70 | 50000 |
| 85 | 50000 |
| 100 | 10000 |
| 200 | 10000 |
| 300 | 6000 |
| 500 | 3000 |
| 1000 | 1000 |
| 2000 | 250 |
| 5000 | 50 |

Table 1: Number of repetitions for a matrix size M.

| | |
|---|---|
| OS | linux 2.6.20 |
| RAM | 512 MB |
| Processor | Intel(R) Pentium(R) 4 CPU 3.06GHz |
| CPU MHz | 3056.931 |
| Cache Size | 512 KB |

Table 2: Benchmark system information.

## 2.1 Initialization

In the initialization phase random values are assigned to a matrix and three vectors with a specific connectivity, like in the following code example:

```
for (int i=0; i<N; ++i)
{
  // vectors
  in(i) = (randval()<i_conn) ? randval() : 0;
  back(i) = (randval()<fb_conn) ? randval() : 0;
  x(i) = randval();

  // matrix
  for (int j=0; j<N; ++j)
  {
    W(i,j) = (randval()<conn) ? randval() : 0;
  }
}
```

This is mainly interesting for sparse matrix libraries, because they can need quite a lot of time in the initialization phase (which might be critical or not).

### 2.2 Matrix-Vector Multiplication

The following matrix-vector product was benchmarked:

$$x = W * x$$

where $W$ is a NxN sparse matrix with a specific connectivity and $x$ is a Nx1 dense vector. A temporary object is needed for such an operation, which results in an additional copy of the vector $x$:

$$t = x$$

$$x = W * t$$

In some libraries the temporary object is generated automatically (which is the usual C++ way, but can result in a big loss of performance), some give warnings and some even produce compilation errors and one has to code the temporary object explicitly.

### 2.3 Neural Network like Operation

The main performance critical loop of the neural network I had to implement (an Echo State Network) looks like this:

$$x = \alpha * in + W * x + \beta * back$$

where $in$ and $back$ are vectors of size Nx1 and $\alpha$, $\beta$ are scalars. Now a kernel function is applied to all the elements of vector $x$:

$$x = tanh(x)$$

And finally a dot product is calculated between vector $out$ of size 2Nx1 and two concatenated $x$ vectors, where the last one is squared:

$$y = dot(out, (x, x^2))$$

## 3 Results

In this section I want to show some illustrative results. For plots with all the tested parameters and higher resolution see http://grh.mur.at/misc/sparse_benchmark_plots/.

Initialization benchmarks can be found in Figure 1 and 2. In general sparse matrices need of course much more time than their nonsparse equivalents: in my examples FLENS took most CPU time for initialization, followed by MTL and then Gmm++.

The matrix-vector product, with doubles and floats, can be seen in Figure 3, 4, 5, 6, 7 and 8. Usually Gmm++ is a little bit faster than FLENS, followed by MTL. Newmat and Blitz++ performed worse than the rest.

Results for the neural network like operation are shown in Figure 9, 10, 11, 12, 13 and 14. They are quite similar to the matrix-vector product, because this is the most expensive operation. However, for small connectivities FLENS outperforms Gmm++, MTL is also quite similar to Gmm++.

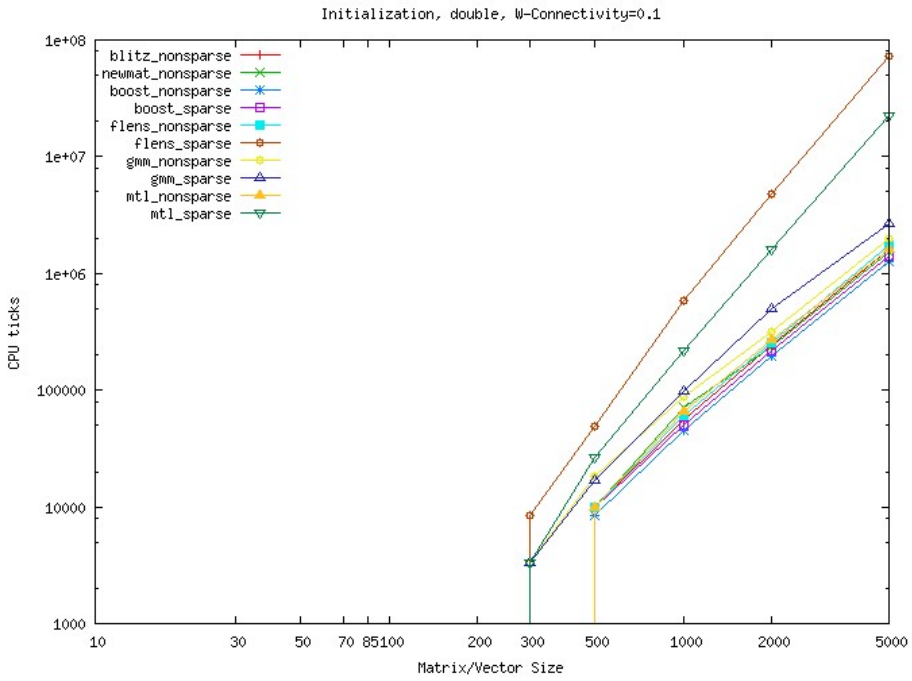


Figure 1: Initialization with doubles, connectivity 10%.

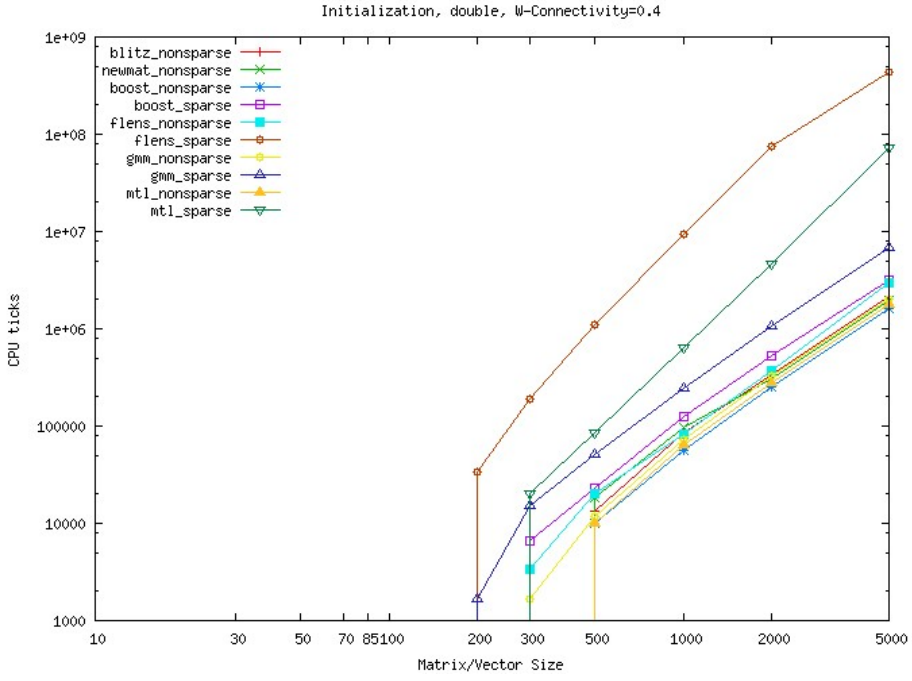


Figure 2: Initialization with doubles, connectivity 40%.

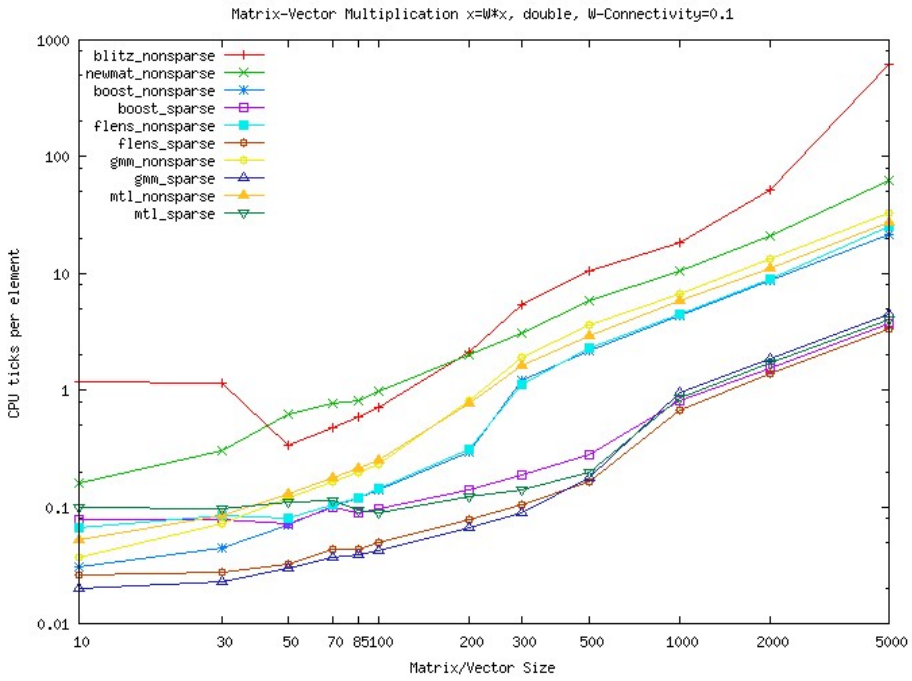


Figure 3: matrix-vector product with doubles, connectivity 10%. NOTE: logarithmic y axis !

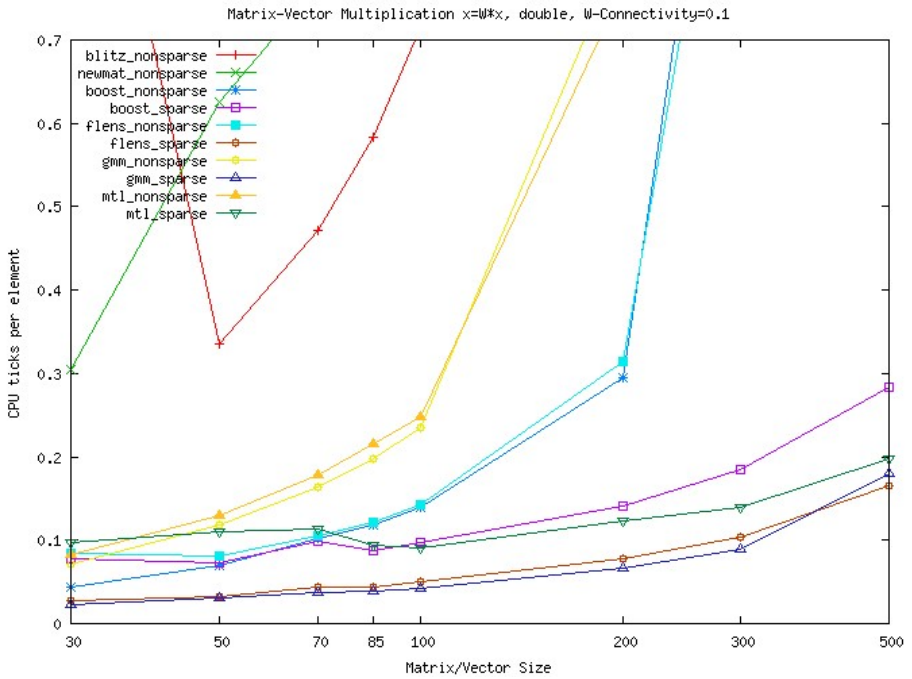


Figure 4: matrix-vector product with doubles, connectivity 10%, zoomed to lower area.

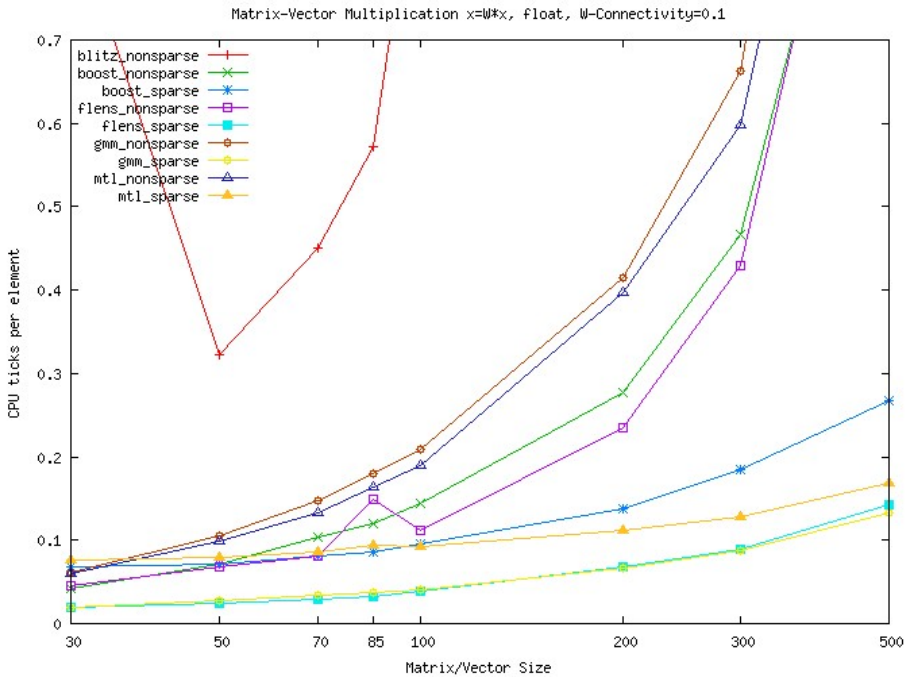


Figure 5: matrix-vector product with floats, connectivity 10%, zoomed to lower area.

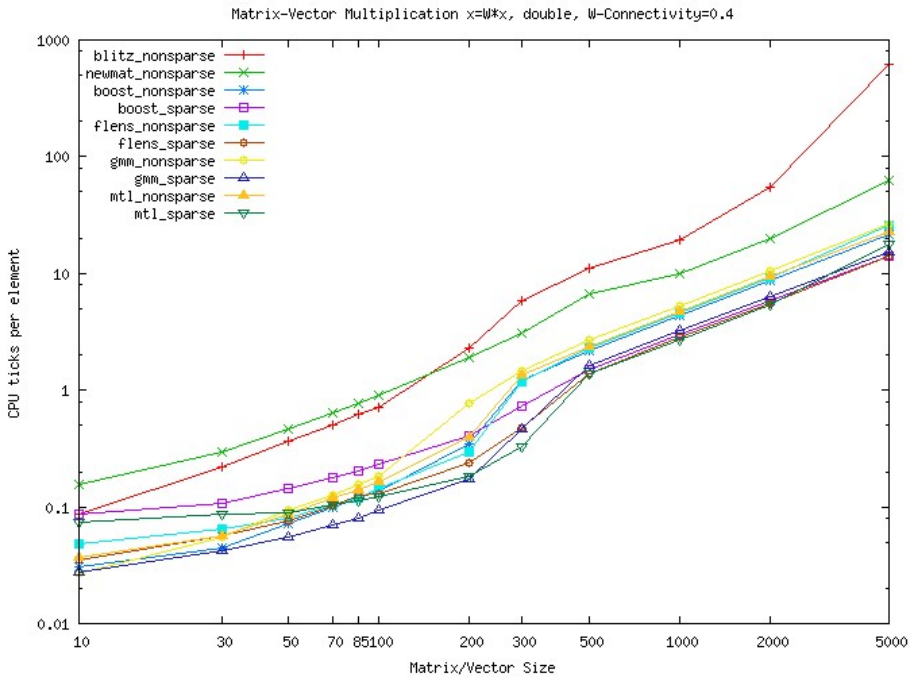


Figure 6: matrix-vector product with doubles, connectivity 40%. NOTE: logarithmic y axis !

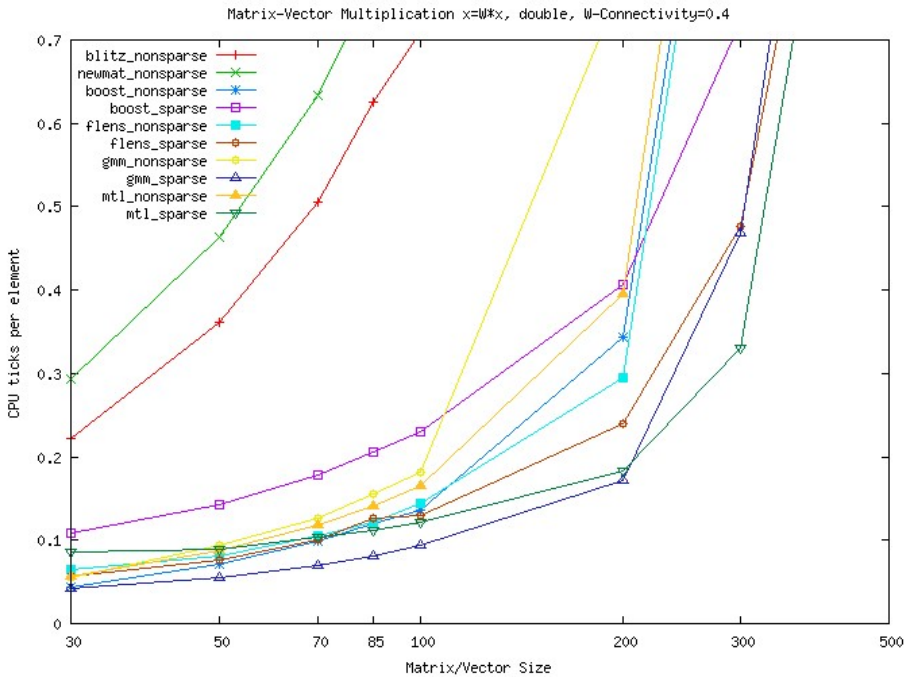


Figure 7: matrix-vector product with doubles, connectivity 40%, zoomed to lower area.

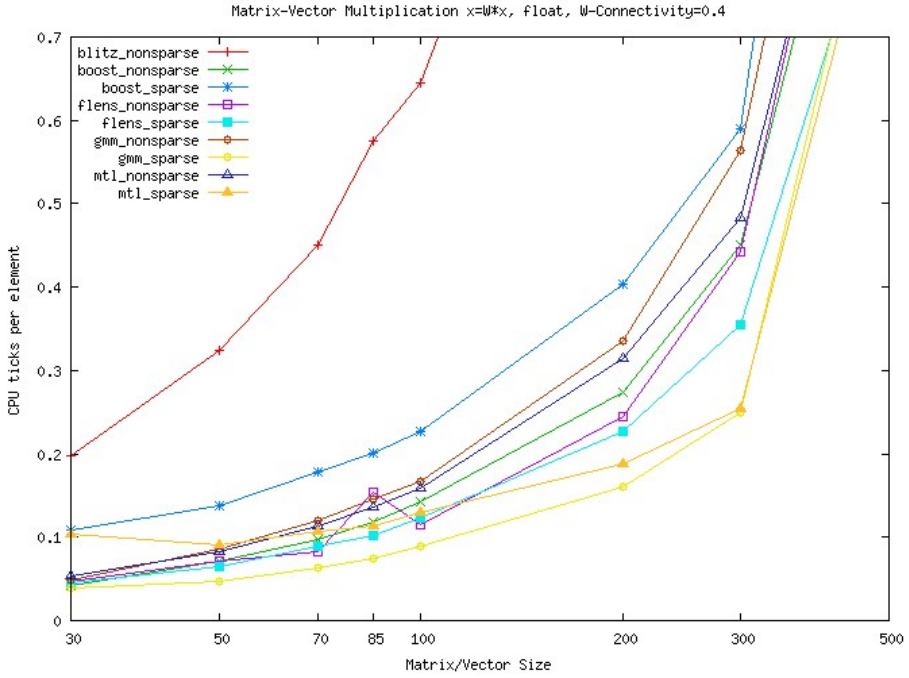


Figure 8: matrix-vector product with floats, connectivity 40%, zoomed to lower area.

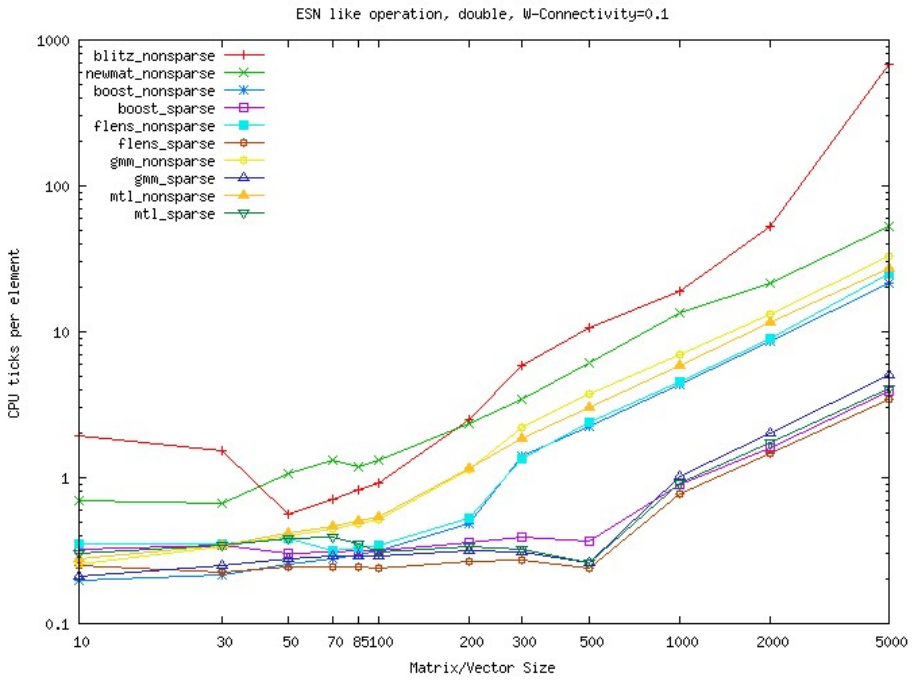


Figure 9: NN like operation with doubles, connectivity 10%. NOTE: logarithmic y axis !

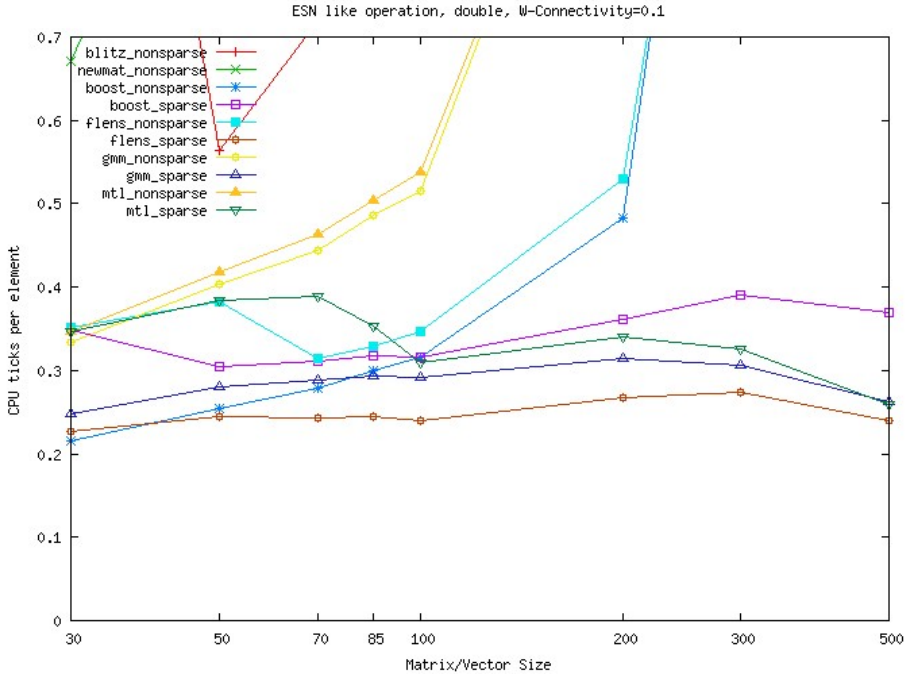


Figure 10: NN like operation with doubles, connectivity 10%, zoomed to lower area.

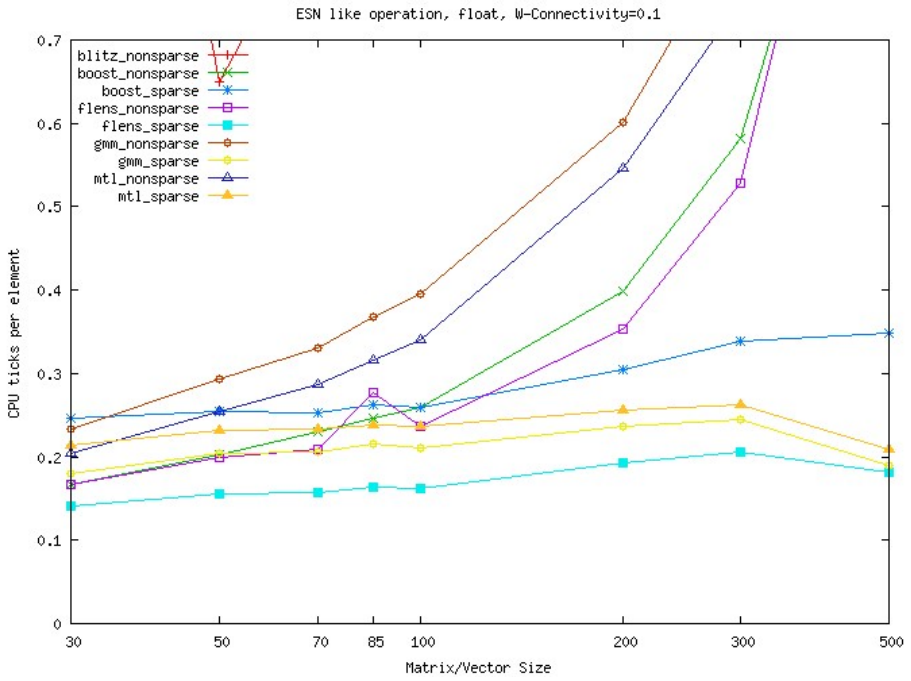


Figure 11: NN like operation with floats, connectivity 10%, zoomed to lower area.

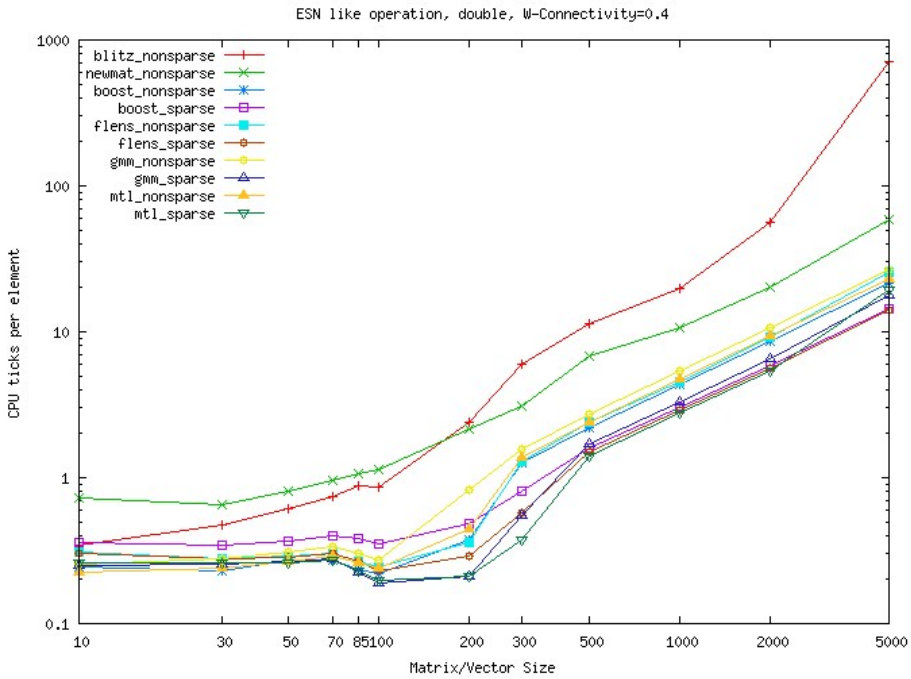


Figure 12: NN like operation with doubles, connectivity 40%. NOTE: logarithmic y axis !

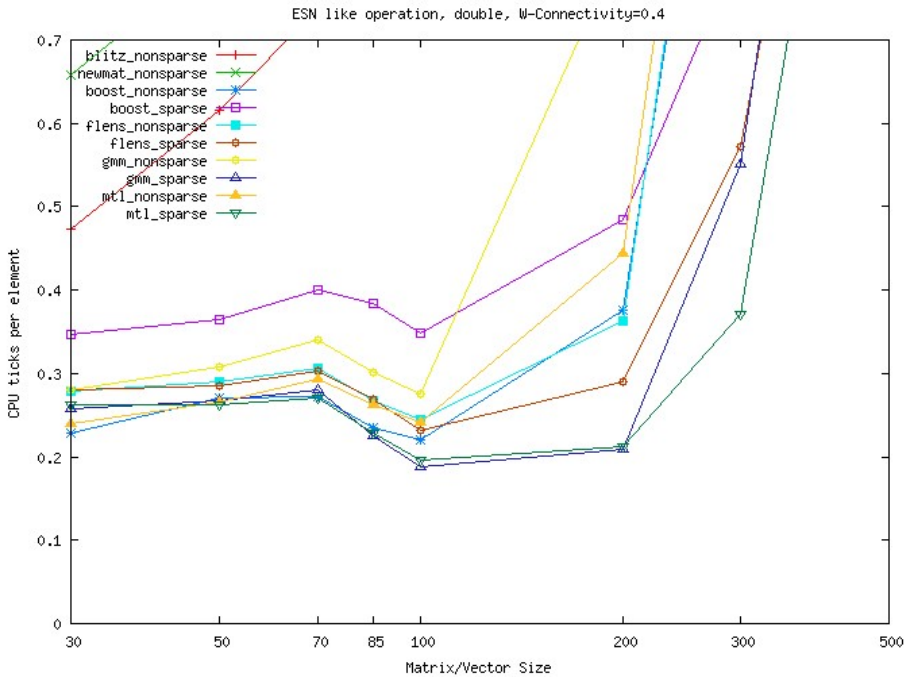


Figure 13: NN like operation with doubles, connectivity 40%, zoomed to lower area.

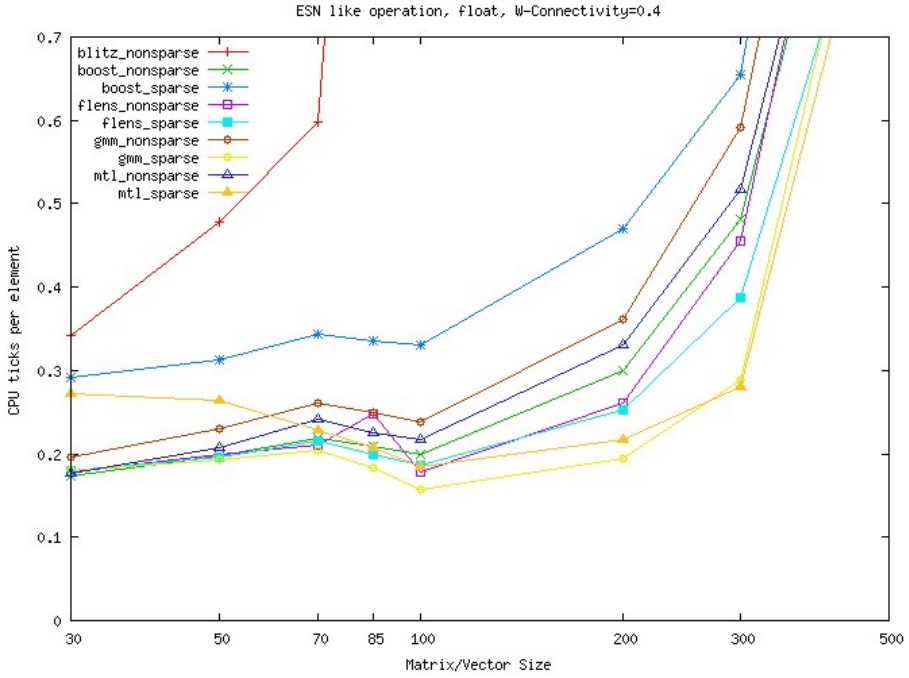


Figure 14: NN like operation with floats, connectivity 40%, zoomed to lower area.

## 4 Conclusion

In general one can say that Gmm++, FLENS and MTL have all very good performance. Also the documentation is quite good for all these libraries. However, FLENS can be used in a more intuitive way (not always), because it overloads operators +, *, -, ... without performance overhead. Gmm++ was in my humble opinion easier to use than MTL, although the commands are quite similar, but Gmm++ supports also sub-matrices for any matrix type.

Both Gmm++ and FLENS have an interface to BLAS and LAPACK, so it's possible to use vendor tuned libraries. In FLENS the bindings are activated per default, in Gmm++ one has to define them. I tried linking Gmm++ against BLAS and LAPACK, but got no performance boost also with non-sparse operations. FLENS was linked against ATLAS BLAS [1]. I tried also GotoBLAS [2] , which is said to be faster than ATLAS, but as these BLAS implementations usually come with no sparse matrix algorithms they perform worse.

According to the benchmarks for non-sparse algorithms presented at http://projects.opencascade.org/btl/ MTL is even faster than the Fortran BLAS implementation and than vendor tuned libraries like the Intel MKL [3]. Intel MKL supports also sparse matrices, which I tried to test, but I did not get a free copy of it (because of some strange errors).

## A Libraries and Flags

The following libraries were tested:

- Blitz++, see http://www.oonumerics.org/blitz/
- boost uBLAS, see www.boost.org/libs/numeric/
- FLENS, see http://flens.sourceforge.net/
- Gmm++, see http://home.gna.org/getfem/gmm_intro
- Matrix Template Library (MTL), see http://www.osl.iu.edu/research/mtl/
- newmat, see http://www.robertnz.net/nm_intro.htm

Compilation/Linking Flags were set to:

- Blitz++:
  CPPFLAGS = -msse -march=pentium4 -mfpmath=sse -funroll-loops -O3 -ftemplate-depth-50;
  LDFLAGS = -lc -lblitz -lm;
- boost uBLAS: CPPFLAGS = -DNDEBUG -ftemplate-depth-50 -msse -march=pentium4 -mfpmath=sse -funroll-loops -O3;
  LDFLAGS [4] = -lc;

[1] ATLAS: Automatic Tuned Linear Algebra Software, a portable, efficient BLAS implementation; http://math-atlas.sourceforge.net/

[2] GotoBLAS: highly optimized BLAS implementation; http://www.tacc.utexas.edu/resources/software/#blas

[3] Intel Math Kernel Library: http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm

[4] It did nont try linking boost uBLAS to BLAS and LAPACK. There exist such bindings but I guess especially for sparse matrices they do not perform better, it should be tried though.

- FLENS:
  CPPFLAGS = -g -O3 -fPIC -DNDEBUG -msse -march=pentium4 -mfpmath=sse -funroll-loops -ftemplate-depth-50;
  LDFLAGS = -llapack -latlas -lblas;

- Gmm++:
  CPPFLAGS = -DNDEBUG -ftemplate-depth-50 -msse -march=pentium4 -mfpmath=sse -funroll-loops -O3
  LDFLAGS [5] = -lc

- MTL: CPPFLAGS = -DNDEBUG -ftemplate-depth-50 -msse -march=pentium4 -mfpmath=sse -funroll-loops -O3
  LDFLAGS = -lc

- newmat: CPPFLAGS = -ftemplate-depth-50 -msse -march=pentium4 -mfpmath=sse -funroll-loops -O3
  LDFLAGS = -lnewmat

## B Implementations

Here the exact implementations of the test algorithms in all libraries are shown. Don't be confused about the little bit cryptic pointer notation!

The whole source code can be downloaded from http://grh.mur.at/misc/sparselib_benchmarks.tar.gz.

Listing 1: Blitz++ Non-Sparse Implementation

```
// BLITZ++ NONSPARSE IMPLEMENTATION

#include <blitz/array.h>
#include <blitz/tinyvec.h>

#include "../bench-tools.h"

using namespace blitz;

typedef Array<TYPE,2> DMatrix;
typedef Array<TYPE,1> DVector;

DMatrix *W;
DVector *in;
DVector *back;
DVector *out;
DVector *x;

int main(int argc, char *argv[])
{
  run_benchmarks(argc,argv);
}

void init_matrices()
```

[5] Here Gmm++ was not linked against BLAS and LAPACK. I tried this but the performance was worse or the same for small/medium matrices.

```
{
  W = new DMatrix(N,N);
  in = new DVector(N);
  back = new DVector(N);
  out = new DVector(2*N);
  x = new DVector(N);

  for (int i=0; i<N; ++i)
  {
    (*in)(i) = (randval()<i_conn) ? randval() : 0;
    (*back)(i) = (randval()<fb_conn) ? randval() : 0;
    (*x)(i) = randval();

    for (int j=0; j<N; ++j)
    {
      (*W)(i,j) = (randval()<conn) ? randval() : 0;
    }
  }

  for (int i=0; i<2*N; ++i)
    (*out)(i) = randval();
}

TYPE compute(int iter)
{
  firstIndex i;
  secondIndex j;

  while(iter--)
  {
    // x = W* x is quite tricky
    *x = sum( (*W)(j,i) * (*x)(i), j);
    // the rest is easy
    *x += alpha * *in;
    *x += beta * *back;

    for(int i=0; i<N; ++i)
      (*x)(i) = tanh( (*x)(i) );

    // make a temporary vector (x,x^2)
    y = sum( (*out)(Range(0,N-1)) * *x);
    for(int i=0; i<N; ++i)
      y += (*out)(N+i)*pow((*x)(i),2);
  }

  return y;
}

void mtx_vec_mult(int iter)
{
  firstIndex i;
  secondIndex j;

  while(iter--)
  {
    // x = W* x was much slower as all the compute()
    // function -> why ?
```

```
    // so now I calc in = W * x

    *in = sum( (*W)(j,i) * (*x)(i), j);
  }
}
```

Listing 2: Boost Non-Sparse Implementation

```
// BOOST NONSPARSE IMPLEMENTATION

#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>

#include "../bench-tools.h"

using namespace boost::numeric::ublas;

typedef matrix<TYPE> GEMatrix;
typedef vector<TYPE> DEVector;

GEMatrix *W;
DEVector *in;
DEVector *back;
DEVector *out;
DEVector *x;

int main(int argc, char *argv[])
{
  run_benchmarks(argc,argv);
}

void init_matrices()
{
  W = new GEMatrix(N,N);
  in = new DEVector(N);
  back = new DEVector(N);
  out = new DEVector(2*N);
  x = new DEVector(N);

  for (int i=0; i<N; ++i)
  {
    (*in)(i) = (randval()<i_conn) ? randval() : 0;
    (*back)(i) = (randval()<fb_conn) ? randval() : 0;
    (*x)(i) = randval();

    for (int j=0; j<N; ++j)
    {
      (*W)(i,j) = (randval()<conn) ? randval() : 0;
    }
  }

  for (int i=0; i<2*N; ++i)
    (*out)(i) = randval();
}

TYPE compute(int iter)
```

```
{
  while(iter--)
  {
    *x = prod(*W,*x);
    *x += alpha * *in;
    *x += beta * *back;

    // activation function
    for(int i=0; i<N; ++i)
      (*x)(i) = tanh( (*x)(i) );

    // calculate y = [out] * [x; x^2]
    y = inner_prod( subrange(*out, 0, N), subrange(*x, 0, N) );
    for (int i=0; i<N; ++i)
      y += (*out)(N+i)*pow((*x)(i),2);
  }

  return y;
}

void mtx_vec_mult(int iter)
{
  while(iter--)
  {
    *x = prod(*W,*x);
  }
}
```

Listing 3: Boost Sparse Implementation

```
// BOOST SPARSE IMPLEMENTATION

#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/vector_sparse.hpp>
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>

#include "../bench-tools.h"

using namespace boost::numeric::ublas;

typedef compressed_matrix<TYPE> SMatrix;
typedef compressed_vector<TYPE> SVector;
typedef vector<TYPE> DEVector;

SMatrix *W;
SVector *in;
SVector *back;
DEVector *out;
DEVector *x;

int main(int argc, char *argv[])
{
  run_benchmarks(argc,argv);
}

void init_matrices()
```

```
{
  W = new SMatrix(N,N);
  in = new SVector(N);
  back = new SVector(N);
  out = new DEVector(2*N);
  x = new DEVector(N);

  for (int i=0; i<N; ++i)
  {
    if(randval()<i_conn) (*in)(i) = randval();
    if(randval()<fb_conn) (*back)(i) = randval();
    (*x)(i) = randval();

    for (int j=0; j<N; ++j)
    {
      if(randval()<conn) (*W)(i,j) = randval();
    }
  }

  for (int i=0; i<2*N; ++i)
    (*out)(i) = randval();
}

TYPE compute(int iter)
{
  while(iter--)
  {
    *x = prod(*W,*x);
    *x += alpha * *in;
    *x += beta * *back;

    // activation function
    for(int i=0; i<N; ++i)
      (*x)(i) = tanh( (*x)(i) );

    // calculate y = [out] * [x; x^2]
    y = inner_prod( subrange(*out, 0, N), subrange(*x, 0, N) );
    for (int i=0; i<N; ++i)
      y += (*out)(N+i)*pow((*x)(i),2);
  }

  return y;
}


void mtx_vec_mult(int iter)
{
  while(iter--)
  {
    *x = prod(*W,*x);
  }
}
```

Listing 4: FLENS Non-Sparse Implementation

```
// FLENS NONSPARSE IMPLEMENTATION

#include <flens/flens.h>
```

```
#include <math.h>
#include "../bench-tools.h"

using namespace flens;
using namespace std;

typedef GeMatrix<FullStorage<TYPE, ColMajor> > GEMatrix;
typedef DenseVector<Array<TYPE> > DEVector;

GEMatrix *W;
DEVector *in;
DEVector *back;
DEVector *out;
DEVector *x;

int main(int argc, char *argv[])
{
  run_benchmarks(argc,argv);
}

void init_matrices()
{
  W = new GEMatrix(N,N);
  in = new DEVector(N);
  back = new DEVector(N);
  out = new DEVector(2*N);
  x = new DEVector(N);

  for (int i=1; i<=N; ++i)
  {
    if(randval()<i_conn) (*in)(i) = randval();
    if(randval()<fb_conn) (*back)(i) = randval();
    (*x)(i) = randval();

    for (int j=1; j<=N; ++j)
    {
      if(randval()<conn) (*W)(i,j) = randval();
    }
  }

  for (int i=1; i<=2*N; ++i)
    (*out)(i) = randval();
}

TYPE compute(int iter)
{
  // temporary objects
  DEVector t;
  DEVector::View out1 = (*out)(_(1,N)), out2 = (*out)(_(N+1,2*N));

  while(iter--)
  {
    t = *x; // temporary object needed for BLAS
    *x = alpha * *in + *W * t + beta * *back;

    // activation function
    for(int i=1; i<=N; ++i)
```

```
      (*x)(i) = tanh( (*x)(i) );

    // calculate y = [out] * [x; x^2]
    y = dot(out1, *x);
    for (int i=1; i<=N; ++i)
      y += out2(i)*pow((*x)(i),2);
  }

  return y;
}

void mtx_vec_mult(int iter)
{
  DEVector::View t1 = (*x)(_(1,N));

  while(iter--)
  {
    *x = *W * t1;
  }
}
```

Listing 5: FLENS Sparse Implementation

```
// FLENS SPARSE IMPLEMENTATION

#include <flens/flens.h>
#include <math.h>
#include "../bench-tools.h"

using namespace flens;
using namespace std;

typedef SparseGeMatrix<CRS<TYPE> > SPMatrix;
typedef DenseVector<Array<TYPE> > DEVector;

SPMatrix *W;
DEVector *in;
DEVector *back;
DEVector *out;
DEVector *x;

int main(int argc, char *argv[])
{
  run_benchmarks(argc,argv);
}

void init_matrices()
{
  W = new SPMatrix(N,N);
  in = new DEVector(N);
  back = new DEVector(N);
  out = new DEVector(2*N);
  x = new DEVector(N);

  for (int i=1; i<=N; ++i)
  {
    if(randval()<i_conn) (*in)(i) = randval();
    if(randval()<fb_conn) (*back)(i) = randval();
```

```
    (*x)(i) = randval();

    for (int j=1; j<=N; ++j)
    {
      if(randval()<conn) (*W)(i,j) = randval();
    }
  }

  for (int i=1; i<=2*N; ++i)
    (*out)(i) = randval();

  // finalize sparse matrix
  W->finalize();
}

TYPE compute(int iter)
{
  // temporary objects
  DEVector t;
  DEVector::View out1 = (*out)(_(1,N)), out2 = (*out)(_(N+1,2*N));

  while(iter--)
  {
    t = *x; // temporary object needed for BLAS
    *x = alpha * *in + *W * t + beta * *back;

    // activation function
    for(int i=1; i<=N; ++i)
      (*x)(i) = tanh( (*x)(i) );

    // calculate y = [out] * [x; x^2]
    y = dot(out1, *x);
    for (int i=1; i<=N; ++i)
      y += out2(i)*pow((*x)(i),2);
  }

  return y;
}

void mtx_vec_mult(int iter)
{
  DEVector::View t1 = (*x)(_(1,N));

  while(iter--)
  {
    *x = *W * t1;
  }
}
```

Listing 6: Gmm++ Non-Sparse Implementation

```
// GMM++ NONSPARSE IMPLEMENTATION

#include "gmm/gmm.h"

#include "../bench-tools.h"

typedef gmm::row_matrix< std::vector<TYPE> > DMatrix;
```

```
typedef std::vector<TYPE> DVector;

DMatrix *W;
DVector *in;
DVector *back;
DVector *out;
DVector *x;

int main(int argc, char *argv[])
{
  run_benchmarks(argc,argv);
}

void init_matrices()
{
  W = new DMatrix(N,N);
  in = new DVector(N);
  back = new DVector(N);
  out = new DVector(2*N);
  x = new DVector(N);

  for (int i=0; i<N; ++i)
  {
    (*in)[i] = (randval()<i_conn) ? randval() : 0;
    (*back)[i] = (randval()<fb_conn) ? randval() : 0;
    (*x)[i] = randval();

    for (int j=0; j<N; ++j)
    {
      (*W)(i,j) = (randval()<conn) ? randval() : 0;
    }
  }

  for (int i=0; i<2*N; ++i)
    (*out)[i] = randval();
}

TYPE compute(int iter)
{
  // temporary objects
  DVector t1(N);
  DVector t2(2*N);

  while(iter--)
  {
    gmm::copy(*x,t1);
    gmm::mult(*W, t1, *x);
    gmm::add(*x, gmm::scaled(*in, alpha), *x);
    gmm::add(*x, gmm::scaled(*back, beta), *x);

    // activation function
    for(int i=0; i<N; ++i)
      (*x)[i] = tanh( (*x)[i] );

    // calculate y = [out] * [x; x^2]
    y = gmm::vect_sp(gmm::sub_vector(*out, gmm::sub_interval(0, N)), *x);
    for (int i=0; i<N; ++i)
```

```
        y += (*out)[N+i]*pow((*x)[i],2);
    }

    return y;
}

void mtx_vec_mult(int iter)
{
    DVector t1(N);

    while(iter--)
    {
      gmm::copy(*x,t1);
      gmm::mult(*W, t1, *x);
    }
}
```

Listing 7: Gmm++ Sparse Implementation

```
// GMM++ SPARSE IMPLEMENTATION

#include "gmm/gmm.h"

#include "../bench-tools.h"

typedef gmm::csr_matrix<TYPE,1> SPMatrix;
typedef gmm::rsvector<TYPE> SPVector;
typedef std::vector<TYPE> DVector;

SPMatrix *W;
SPVector *in;
SPVector *back;
DVector *out;
DVector *x;

int main(int argc, char *argv[])
{
    run_benchmarks(argc,argv);
}

void init_matrices()
{
   W = new SPMatrix(N,N);
    in = new SPVector(N);
    back = new SPVector(N);
    out = new DVector(2*N);
    x = new DVector(N);

    // we need temporary writeable sparse objects:
    gmm::row_matrix< gmm::wsvector<double> > W1(N,N);
    gmm::wsvector<double> in1(N);
    gmm::wsvector<double> back1(N);

    for (int i=0; i<N; ++i)
    {
      if(randval()<i_conn) in1[i] = randval();
      if(randval()<fb_conn) back1[i] = randval();
      (*x)[i] = randval();
```

```
    for (int j=0; j<N; ++j)
    {
      if (randval()<conn) W1(i,j) = randval();
    }
  }

 for (int i=0; i<2*N; ++i)
   (*out)[i] = randval();

  // now copy to the original sparse matrix
  gmm::copy(W1, *W);
  gmm::copy(in1, *in);
  gmm::copy(back1, *back);
}

TYPE compute(int iter)
{
  // temporary objects
  DVector t1(N);
  DVector t2(2*N);

  while(iter--)
  {
    gmm::copy(*x,t1);
    gmm::mult(*W, t1, *x);
    gmm::add(*x, gmm::scaled(*in, alpha), *x);
    gmm::add(*x, gmm::scaled(*back, beta), *x);

    // activation function
    for(int i=0; i<N; ++i)
      (*x)[i] = tanh( (*x)[i] );

    // calculate y = [out] * [x; x^2]
    y = gmm::vect_sp(gmm::sub_vector(*out, gmm::sub_interval(0, N)), *x);
    for (int i=0; i<N; ++i)
      y += (*out)[N+i]*pow((*x)[i],2);
  }

  return y;
}

void mtx_vec_mult(int iter)
{
  DVector t1(N);

  while(iter--)
  {
    gmm::copy(*x,t1);
    gmm::mult(*W, t1, *x);
  }
}
```

Listing 8: MTL Non-Sparse Implementation

```
// MTL NONSPARSE IMPLEMENTATION

#include "mtl/matrix.h"
```

```
#include <mtl/mtl.h>
#include <mtl/utils.h>

#include "../bench-tools.h"

using namespace mtl;

typedef matrix<TYPE, rectangle<>, dense<>,
               row_major>::type  DMatrix;
typedef dense1D<TYPE> DVector;

DMatrix *W;
DVector *in;
DVector *back;
DVector *out;
DVector *x;

int main(int argc, char *argv[])
{
  run_benchmarks(argc,argv);
}

void init_matrices()
{
  W = new DMatrix(N,N);
  in = new DVector(N);
  back = new DVector(N);
  out = new DVector(2*N);
  x = new DVector(N);

  for (int i=0; i<N; ++i)
  {
    (*in)[i] = (randval()<i_conn) ? randval() : 0;
    (*back)[i] = (randval()<fb_conn) ? randval() : 0;
    (*x)[i] = randval();

    for (int j=0; j<N; ++j)
    {
      (*W)(i,j) = (randval()<conn) ? randval() : 0;
    }
  }

  for (int i=0; i<2*N; ++i)
    (*out)[i] = randval();
}

TYPE compute(int iter)
{
  // temporary object
  DVector t1(N);

  while(iter--)
  {
    copy(*x,t1);
    mult(*W, t1, *x);
    add(*x, scaled(*in, alpha), *x);
    add(*x, scaled(*back, beta), *x);
```

```
    // activation function
    for(int i=0; i<N; ++i)
      (*x)[i] = tanh( (*x)[i] );

    y = dot( (*out)(0,N), *x);
    for (int i=0; i<N; ++i)
      y += (*out)[N+i]*pow((*x)[i],2);
  }

  return y;
}

void mtx_vec_mult(int iter)
{
  DVector t1(N);

  while(iter--)
  {
    copy(*x,t1);
    mult(*W, t1, *x);
  }
}
```

Listing 9: MTL Sparse Implementation

```
// MTL SPARSE IMPLEMENTATION

#include "mtl/matrix.h"
#include <mtl/mtl.h>
#include <mtl/utils.h>

#include "../bench-tools.h"

using namespace mtl;

typedef matrix<TYPE, rectangle<>,
        compressed<>, row_major>::type SPMatrix;
typedef dense1D<TYPE> DVector;
typedef compressed1D<TYPE> SPVector;

SPMatrix *W;
DVector *in;
DVector *back;
DVector *out;
DVector *x;

int main(int argc, char *argv[])
{
  run_benchmarks(argc,argv);
}

void init_matrices()
{
  W = new SPMatrix(N,N);
  in = new DVector(N);
  back = new DVector(N);
  out = new DVector(2*N);
```

```
  x = new DVector(N);

  for (int i=0; i<N; ++i)
  {
   if (randval()<i_conn) (*in)[i] = randval();
   if (randval()<fb_conn) (*back)[i] = randval();
   (*x)[i] = randval();

    for (int j=0; j<N; ++j)
    {
      if(randval()<conn) (*W)(i,j) = randval();
    }
  }

 for (int i=0; i<2*N; ++i)
   (*out)[i] = randval();
}

TYPE compute(int iter)
{
  // temporary object
  DVector t1(N);

  while(iter--)
  {
    copy(*x,t1);
    mult(*W, t1, *x);
    add(*x, scaled(*in, alpha), *x);
    add(*x, scaled(*back, beta), *x);

    // activation function
    for(int i=0; i<N; ++i)
      (*x)[i] = tanh( (*x)[i] );

    y = dot( (*out)(0,N), *x);
    for (int i=0; i<N; ++i)
      y += (*out)[N+i]*pow((*x)[i],2);
  }

  return y;
}

void mtx_vec_mult(int iter)
{
  DVector t1(N);

  while(iter--)
  {
    copy(*x,t1);
    mult(*W, t1, *x);
  }
}
```

Listing 10: Newmat Non-Sparse Implementation

```
// NEWMAT NONSPARSE IMPLEMENTATION

#define WANT_STREAM
```

```
#define WANT_MATH

#include "newmatap.h"
#include "newmatio.h"

using namespace NEWMAT;
using namespace std;

#include "../bench-tools.h"

Matrix *W;
ColumnVector *in;
ColumnVector *back;
RowVector *out;
ColumnVector *x;

int main(int argc, char *argv[])
{
  run_benchmarks(argc,argv);
}

void init_matrices()
{
  W = new Matrix(N,N);
  in = new ColumnVector(N);
  back = new ColumnVector(N);
  out = new RowVector(2*N);
  x = new ColumnVector(N);

  for (int i=1; i<=N; ++i)
  {
    if(randval()<i_conn) (*in)(i) = randval();
    if(randval()<fb_conn) (*back)(i) = randval();
    (*x)(i) = randval();

    for (int j=1; j<=N; ++j)
    {
      if(randval()<conn) (*W)(i,j) = randval();
    }
  }

  for (int i=1; i<=2*N; ++i)
    (*out)(i) = randval();
}

double compute(int iter)
{
  ColumnVector t(2*N);

  while(iter--)
  {
    *x = *W * *x;
    *x += alpha * *in;
    *x += beta * *back;

    // activation function
   for(int i=1; i<=N; ++i)
```

```
      (*x)(i) = tanh( (*x)(i) );

    // make a temporary vector (x,x^2)
    t = *x & *x;
    for(int i=1; i<=N; ++i)
      t(N+i) = pow((*x)(i),2);

    // compute output
    y = DotProduct(*out, t);
  }

  return y;
}

void mtx_vec_mult(int iter)
{
  while(iter--)
  {
    // STRANGE: this does not work:
    // x = W * x;
    // it takes really a lot cpu
    // (but in compute() this works fast !?)
    // so for the test now this:
    *in = *W * *x;
  }
}
```

## References


[1] D. Kincaid F. T. Krogh C. L. Lawson, R. J. Hanson. Basic linear subprograms for fortran usage. 1979. ACM Transactions on Mathematical Software.

[2] cplusplus.com. clock reference. http://www.cplusplus.com/reference/clibrary/ctime/clock.html; accessed 07-08-2007.

[3] C.Bischof S.Blackford J.Demmel J.Dongarra J.Croz A.Greenbaum S.Hammarling A.McKenney D.Sorensen E.Anderson, Z.Bai. Lapack users' guide. 1999. Society for Industrial and Applied Mathematics.

[4] Michael Lehn. Everything you always wanted to know about flens, but were afraid to ask. 2007. University of Ulm, Department of Numerical Analysis, Germany.

[5] Ulisses Mello and Ildar Khabibrakhmanov. On the reusability and numeric efficiency of c++ packages in scientific computing. 2003. IBM T.J. Watson Research Center, Yorktown, NY, USA.

[6] NC State University. Serial profiling and timing. http://www.ncsu.edu/itd/hpc/Documents/sprofile.php; accessed 07-08-2007.