Profiling, Optimization, and Acceleration of MATLAB code

Dr Philip J Naylor

Profiling

Before you try and make any sort of program run faster you first need to find out which parts of it are running slowly. MATLAB provides a simple mechanism for seeing how much time a particular piece of code takes to run, by adding the command `tic;` at the start, and `toc;` at the end. When the `toc;` command executes it prints out the amount of time that has elapsed since the preceeding `tic;` command. However, there are two problems with this approach:

1. The time printed is the elapsed "wall clock" time – so it will be affected by whatever else your computer was doing whilst the MATLAB program was running.
2. You would need to wrap increasingly smaller and smaller sections of your program in `tic;...toc;` commands in order to find out where the slowest parts are.

A much better solution is to use MATLAB's built-in code profiler. Add the command `profile on;` at the start of your program, and `profile viewer;` at the end, then run the program. Once it has finished executing a profiler window will appear that contains the information that you need. For each function call in the top level of your program you get the following information:

Function Name - the name of the function.

Calls – the number of times that function has been called.

Total Time – the amount of CPU time spent executing that function **including** any other functions that it calls.

Self Time – the amount of CPU time spent executing that function **excluding** any other functions that it calls.

Total Time Plot – a graphical reperesentation of the total and self times.

Clicking on the column headings causes the table to be redisplayed ordered by that value (function names are sorted alphabetically, the numerical values in descending order).

Clicking on a function name will give you a more detailed break down from within that function. This culminates in a source code listing that has been annotated to include how much time has been spent executing each line of code, how many times that line has been executed, and the slowest lines are highlighted in shades of red. **N.B.** since the profiling is done line by line, you should avoid putting multiple commands on a line – otherwise you will not know which part is responsible for how much CPU time.

Sort the results by "Self Time" and work your way through the program seeing which parts of it MATLAB is spending most of it's time on. These are the areas that you should concentrate on optimizing first, as they are likely to give the most improvement for the least amount of effort. If your program is designed to run on various different sized systems, make sure that you profile it for a range of sizes (1x, 2x, 4x, etc.) as it is likely that different parts of the program will dominate at different scales – identify those parts of the program that scale least well.

N.B. Once you have finished profiling your program, remove (or at least comment out) the `profile on;` and `profile viewer;` commands – as having profiling enabled will, in itself, make your code run slower.

Standard Tips

For further information see the relevant sections in the MATLAB help facility (select Help → MATLAB Help or press F1 whilst the MATLAB window has focus). [In R2008 select Help → Product Help]

Performance

See the MATLAB → Programming → Improving Performance and Memory Usage help section for the most up-to-date information. [In R2008 select MATLAB → Programming Fundamentals → Performance → Techniques for Improving Performance]

Turn on multithreading

If your computer has multiple cores, you can get MATLAB to use separate execution threads to run those of it's low-level functions that have been suitably parallelized. To turn this on open File → Preferences and select General → Multithreading, then tick the Enable multithread computation box. Leave the other setting as Automatic unless you specifically want to reserve some cores for other purposes. N.B. MATLAB versions prior to R2008a do not support multithreading on AMD processors.

Vectorize loops

MATLAB is specifically designed to operate on vectors and matrices, so it is usually quicker to perform operations on vectors, or matrices, rather than using a loop. For example:

```
index=0;
for time=0:0.001:60;
  index=index+1;
  waveForm(index)=cos(time);
end;
```

would run considerably faster if replaced with:

```
time=0:0.001:60;
waveForm=cos(time);
```

Functions that you might find useful when using vector operations in place of loops include:

`any()` – returns `true` if any element is non-zero.

`size()` – returns a vector containing the number of elements in an array along each dimension.

`find()` – returns the indices of any non-zero elements. To get the non-zero values themselves you can use something like `a(find(a));`.

`cumsum()` – returns a vector containing the cumulative sum of the elements in its argument vector, e.g. `cumsum([0:5])` returns `[0 1 3 6 10 15]`.

`sum()` – returns the sum of all the elements in a vector, e.g. `sum([0:5])` returns `15`.

Preallocate arrays

MATLAB uses contiguous areas of memory to hold arrays (vectors and matrices), rather than linked lists. This means that whenever you add extra elements it has to find a section of memory big enough to hold the enlarged array, and then copy all of the existing elements across to the new location. This may be acceptable for on one-off resizing, but incrementing an array one element at a time in a loop is definitely not a good idea:

```
results=0;
for index=2:1000;
  results(index)=results(index-1)+index;
end;
```

is slower than:

```
results=zeros(1,1000);
for index=2:1000;
  results(index)=results(index-1)+index;
end;
```

N.B. when using `zeros()` to create an array of a specific type you should use its `classname` argument, rather than casting. `results=int8(zeros(1,1000));` will create a zeroed array of 1000 `double` variables, and then convert each of these to `int8`, whereas `results=zeros(1,1000,'int8');` will create 1000 `int8` variables straight off.

Don't change data types

In order for MATLAB to be able to support loose data typing (i.e. variables can store different types of information, rather than being declared as specific types) it has to store a certain amount of "header" information along with the actual data. This has implications for the amount of memory needed to store certain data types (see below), but it also means that there is an extra overhead to changing the type of the data stored in a particular variable. You may find it quicker to use a new variable instead.

Use `real...` functions for real data

Some MATLAB functions are designed to work on both real and complex numbers. If you are only going to be using them on real numbers it will be quicker to use the versions that are specifically designed for non-complex numbers, e.g. `reallog()`, `realpow()`, and `realsqrt()`.

Use "short circuit" logical operators

MATLAB's "short circuit" logical operators stop processing when they reach a sufficient condition to know the outcome of the whole operation. For example, in the condition `if (index >= 3) && (data(index) == 5);` the second part of the test will not be evaluated if the value of `index` is less than `3`.

Use function handles

Some MATLAB functions take other function names as an argument. It is tempting to specify these by using a variable to store the name as a string (e.g. `func='tan';` and then use the variable as the argument to the other function (e.g. `fzero(func,0)`). This is fine for a single function call but it there are two problems if the process is repeated inside a loop:

1. for each iteration of the loop MATLAB has to search its path for the function (e.g. `tan`) – which takes time.
2. the path may have changed between one iteration and the next, so there is no guarantee that the function that is found first this time is the same as the one found first last time – leading to inconsistent results.

The solution to both problems is to use a file handle (`func=@sin`) to return a unique identifier to the function, which can be then passed as an argument (as before).

File I/O

In general the high level input/output operations (`load()` and `save()`) have been written to be much faster than than the low level operations (`fread()` and `fwrite()`).

Memory Usage

See the MATLAB → Programming → Using Memory Efficiently help section for the most up-to-date information. [In R2008 select MATLAB → Programming Fundamentals → Memory Usage → Strategies for Effificient Use of Memory]

Also bear in mind that you can uses the `whos()` function to see how much memory a particular MATLAB variable is using.

Copying arrays

When you take a copy of an array, MATLAB initially only makes a copy of the pointer to the array data. It is only if you subsequently make a change to one of the versions that all the data gets replicated at a new location. This includes when arrays are passed as function arguments – as argument passing in MATLAB is by value, rather than by reference. Consequently, wherever possible, you might want to try and avoid making small changes to very large arrays.

Free memory when you have finished with it

When you know that you will not need a variable any more, you can delete it using `clear variableName;` – this will make that piece of memory available for re-use. N.B. MATLAB will find it much easier to re-use large chunks of freed up memory if they are contiguous, so always create large variables before small ones, and try to group them together.

Structure storage

As noted above, variables in MATLAB include headers that describe the type of data that they hold. For structures there is one header for the structure as a whole, and one for each member. You should take care when mixing arrays and structures in order to minimize your memory usage – for example:

```
pixel.red(1:600,1:400)
pixel.grn(1:600,1:400)
pixel.blu(1:600,1:400)
```

would need to store 4 headers, whereas:

```
pixel(1:600,1:400).red
pixel(1:600,1:400).grn
pixel(1:600,1:400).blu
```

would need to store 720,001 headers.

Use the smallest suitable data type

To minimise memory usage, always use the smallest data type that is suitable for the job at hand. For example:

- do not use `complex` variables to store values where the imaginary part will always be zero.
- if they provide sufficient accuracy, use `single` variables rather than `double`.
- consider using `uint16` for counting operations – it will handle values between `0` and `65535` inclusive, and it takes up only a quarter the memory of the default data type (`double`).

Use sparse matrices

If a significant number of the elements in a matrix are zero, consider converting it to the sparse format (using the `sparse()` function). This will only store the values, and indices, of the non-zero elements Because of the extra overhead of storing the indices this is only worth doing if more than about 75% of the elements of a 2D matrix are zero – otherwise the sparse format will actually take up more space.

Parallel Loops

From MATLAB version R2008a onwards, the distributed computing toolbox has a much simplified interface. If you have an outer `for` loop that meets the following criteria:

- the loop counter only takes integer values.
- each iteration is independant of the others.
- it doesn't matter what order the iterations are performed in.

then you might want to look at replacing it with a `parfor` loop enclosed in `matlabpool open;` and `matlabpool close` commands. By default `matlabpool open;` creates four local worker processes – see the help facility for other options.

N.B the overhead for starting the worker pool is about 10-15 seconds, and for shutting it down about 5 seconds. Plus there are communication overheads between the main MATLAB session and the worker processes. So, this approach is only worth considering if the loop would normally take more than 30 seconds or so to execute as a non-parallel job.

MEX-files

MATLAB is an interpreted language – this means that each time it comes to a line of code it needs to figure out what you want it to do. This is a particular performance hit inside any loops that it has not been possible to vectorize. The solution is to perform these calculations using a compiled program, written in C or FORTRAN.

For more general guidance on how to do this see the help section MATLAB → External Interfaces → Calling C and FORTRAN Programs from MATLAB.

The following FORTRAN program gives an idea of how the process works:

```
#include "fintrf.h"
C
C     fmatprod.F - multiply two matrices, or a scalar and a matrix
C
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------------
C declare arguments
      integer   nlhs, nrhs
      mwpointer plhs(*), prhs(*)
C-----------------------------------------------------------------------
C declare functions used
      integer   mxIsNumeric , mxIsSparse
      mwpointer mxCreateDoubleMatrix, mxGetPr
      mwsize    mxGetM, mxGetN
C-----------------------------------------------------------------------
C declare local variables
      mwsize    m(2), n(2)
      real*8    A(1000*1000), B(1000*1000), C(1000*1000)
      mwpointer A_pr, B_pr, C_pr
      logical   scalar
C-----------------------------------------------------------------------
C check for correct number of arguments
      if (nrhs .ne. 2) then
        call mexErrMsgTxt('fmatprod() - exactly two inputs required.')
      elseif (nlhs .ne. 1) then
        call mexErrMsgTxt('fmatprod() - exactly one output required.')
      endif

C check to see both inputs are numeric
      if (mxIsNumeric(prhs(1)) .ne. 1) then
        call mexErrMsgTxt('fmatprod() - input 1 is non-numeric.')
      elseif (mxIsNumeric(prhs(2)) .ne. 1) then
        call mexErrMsgTxt('fmatprod() - input 2 is non-numeric.')
      endif

C check to see if first input is a scalar
      m(1)=mxGetM(prhs(1))
      n(1)=mxGetN(prhs(1))
      scalar=((m(1) .eq. 1) .and. (n(1) .eq. 1))

C get the size of the second matrix
      m(2)=mxGetM(prhs(2))
      n(2)=mxGetN(prhs(2))

C check that the problem is not too big
      if((m(1)*n(1) .gt. 1000*1000) .or.
     +   (m(2)*n(2) .gt. 1000*1000)) then
        call mexErrMsgTxt('fmatprod() - problem is too big')
      end if

C check that the matrices are the correct shapes
      if ((.not. scalar) .and. (n(1) .ne. m(2))) then
        call mexErrMsgTxt('fmatprod() - input matrix shape mismatch')
      end if

C check that neither matrix is sparse
      if (mxIsSparse(prhs(1)) .eq. 1) then
        call mexErrMsgTxt('fmatprod() - input 1 is sparse')
      end if
      if (mxIsSparse(prhs(2)) .eq. 1) then
        call mexErrMsgTxt('fmatprod() - input 2 is sparse')
      end if

C create matrix for the return argument
```

```
      if (scalar) then
        plhs(1)=mxCreateDoubleMatrix(m(2),n(2),0)
      else
        plhs(1)=mxCreateDoubleMatrix(m(1),n(2),0)
      end if
      A_pr=mxGetPr(prhs(1))
      B_pr=mxGetPr(prhs(2))
      C_pr=mxGetPr(plhs(1))

C load the data into Fortran arrays.
      call mxCopyPtrToReal8(A_pr,A,m(1)*n(1))
      call mxCopyPtrToReal8(B_pr,B,m(2)*n(2))

C call the appropriate computational subroutine
      if (scalar) then
        call scalar_product(A,B,C,m,n)
      else
        call matrix_product(A,B,C,m,n)
      end if

C load the output into a MATLAB array
      if (scalar) then
        call mxCopyReal8ToPtr(C,C_pr,m(2)*n(2))
      else
        call mxCopyReal8ToPtr(C,C_pr,m(1)*n(2))
      end if

      return
      end

C=======================================================================

      subroutine scalar_product(A,B,C,m,n)

      mwsize m(2), n(2)
      real*8 A(m(1),n(1)), B(m(2),n(2)), C(m(2),n(2))
C
      mwsize i, j

      do 20 j=1,n(2),1
         do 10 i=1,m(2),1
            C(i,j)=A(1,1)*B(i,j)
 10      continue
 20   continue

      return
      end

C=======================================================================

      subroutine matrix_product(A,B,C,m,n)

      mwsize m(2), n(2)
      real*8 A(m(1),n(1)), B(m(2),n(2)), C(m(1),n(2))
C
      real*8 alpha, beta
      integer i,j
C
      external dgemm

      alpha=1.0D0
      beta=0.0D0

      call dgemm('N','N',m(1),n(2),m(2),alpha,A,m(1),B,m(2),beta,
     +           C,m(1))

      return
      end
```

This “program” doesn’t contain a program as such, just a gateway subroutine (`mexFunction`) that allows MATLAB to interface with the FORTRAN routines that actually do the work. N.B. FORTRAN Mex-files need to have a `.F` file extension, or they will not compile. The equivalent in C is:

```
#include "mex.h"
void scalar_product(double *A_pr, double *B_pr, double *C_pr,
                                                  mwSize m[], mwSize n[]);
void matrix_product(double *A_pr, double *B_pr, double *C_pr,
                                                  mwSize m[], mwSize n[]);
/*
 * cmatprod.c - multiply two matrices, or a scalar and a matrix
 */
void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

  double  *A_pr,*B_pr,*C_pr;
  mwSize  m[2],n[2];
  int     scalar;

  /* check for correct number of arguments */
  if(nrhs!=2) {
    mexErrMsgTxt("cmatprod() - exactly two inputs required");
  }
  if(nlhs!=1) {
    mexErrMsgTxt("cmatprod() - exactly one output required");
  }

  /* check to see both inputs are numeric */
  if (!mxIsNumeric(prhs[0])) {
    mexErrMsgTxt("cmatprod() - input 1 is non-numeric");
  }
  if (!mxIsNumeric(prhs[1])) {
    mexErrMsgTxt("cmatprod() - input 2 is non-numeric");
  }

  /* check to see if first input is a scalar */
  m[0]=mxGetM(prhs[0]);
  n[0]=mxGetN(prhs[0]);
  scalar=((m[0]==1) && (n[0]==1));

  /* get the size of the second matrix */
  m[1]=mxGetM(prhs[1]);
  n[1]=mxGetN(prhs[1]);

  /* check that the matrices are the correct shapes */
  if ((!scalar) && (n[0] != m[1])) {
    mexErrMsgTxt("cmatprod() - input matrix shape mismatch");
  }

  /* check that neither matrix is sparse */
  if (mxIsSparse(prhs[0])) {
    mexErrMsgTxt("cmatprod() - input 1 is sparse");
  }
  if (mxIsSparse(prhs[1])) {
    mexErrMsgTxt("cmatprod() - input 2 is sparse");
  }

  /* get pointers to the inputs */
  A_pr=mxGetPr(prhs[0]);
  B_pr=mxGetPr(prhs[1]);

  /* create a matrix for the output  */
  if (scalar) {
    plhs[0]=mxCreateDoubleMatrix(m[1],n[1],mxREAL);
  } else {
    plhs[0]=mxCreateDoubleMatrix(m[0],n[1],mxREAL);
  }
```

```
  /* create a pointer to the output matrix */
  C_pr=mxGetPr(plhs[0]);

  /*  call the the appropriate computational subroutine */
  if (scalar) {
    scalar_product(A_pr,B_pr,C_pr,m,n);
  } else {
    matrix_product(A_pr,B_pr,C_pr,m,n);
  }

}

/*==========================================================================*/
void scalar_product(double *A_pr, double *B_pr, double *C_pr,
                                                  mwSize m[], mwSize n[]) {
  mwSize i,j;

  for (j=0; j<n[1]; j++) {
    for (i=0; i<m[1]; i++) {
      *(C_pr+(j*m[1])+i)=(*A_pr)*(*(B_pr+(j*m[1])+i));
    }
  }
}

/*==========================================================================*/
void matrix_product(double *A_pr, double *B_pr, double *C_pr,
                                                  mwSize m[], mwSize n[]) {
  mwSize i,j;
  double alpha=1.0, beta=0.0;
  char   *no="N";

  dgemm_(no,no,&m[0],&n[1],&m[1],&alpha,A_pr,&m[0],B_pr,&m[1],
         &beta,C_pr,&m[0]);

  /* dgemm(no,no,&m[0],&n[1],&m[1],&alpha,A_pr,&m[0],B_pr,&m[1],
        &beta,C_pr,&m[0]); /* for MATLAB R2008 */

}
```

In this case it has not been necessary to create copies of the arrays, they are just passed as pointers, so the C version has a smaller memory footprint (and doesn't need to be recompiled to cope with larger matrices). It is possible to construct FORTRAN MEX-files that pass the arrays around by reference (using the `%val()` function), but it's very easy to get this wrong, which will lead to segmentation violations.

Compiling MEX-files

To compile each of the above MEX-files you would type (on the command line, not in MATLAB):

```
mex fmatprod.F
```

or

```
mex cmatprod.c
```

If you get an error about the `g95` compiler not being available, you will need to change the `mex` configuration parameters. To do this type `mex -setup` and select option 2. This will create the file `mexopts.sh` in the directory `.matlab/R2007a/` under your home directory (replacing the version number with whatever you are using). If you copy this file to the directory where the source code for your MEX-files is kept, the latter version will override the former. Determine which architecture you are running on, by typing `uname -i` and seeing whether this returns `i386` (32-bit), or `x86_64` (64-bit). Then edit the `mexopts.sh` file. For 32-bit systems find the section starting `glnx86)`, and for 64-bit find `glnxa64)` then change the line `FC='g95'` to `FC='g77'`. For these particular MEX-files you would also need to specify which BLAS library to use (for the `DGEMM` routine), so add `-lacml` to the end of the `CLIBS` and `FLIBS` settings. [In R2008 you will need to use `-lmwblas` instead]

Once you get a MEX-file to compile you will find that you have created a new file with the same name, but with either a `.mexglx` (32-bit) or `.mexa64` (64-bit) extension. Now, so long as the `.mex...` file is in a directory that is in your MATLAB path, you can use the file name (without the extension) as a MATLAB command, e.g. `C=fmatprod(A,B);`.

Since MATLAB uses BLAS and LAPACK libraries anyway, in practice there usually isn't anything to be gained by using MEX-files to access these routines directly, however you may want to take a look at `http://math.nist.gov/spblas/` and compile a copy of the Sparse BLAS library. N.B. the way that MATLAB stores sparse arrays is not the same format that the above routines expect, so you do need to produce a function to map between the two.

Debugging MEX-files

Whilst it is possible to debug MEX-files from within MATLAB, I would not recommend it. It's much easier to keep all your MEX-files in two parts – the gateway routine, and the one that actually does the work. You can then get the main routine working under the control of a suitable C, or FORTRAN, harness program, and just interface it with the gateway routine once you're sure that it's all working.

The ClearSpeed Accelerated Maths Library (CSXL)

The ClearSpeed Accelerated Maths Library uses ClearSpeed accelerator cards to implement large scale parallelisation of some BLAS/LAPACK routines. The version currently installed on the HPC cluster supports the `DGEMM` (matrix-matrix product) BLAS routine, and the `DGETRF` (LU factorization) LAPACK routine. Unfortunately it won't currently work with MATLAB versions later than R2006b.

To force MATLAB to use CSXL you need to set the following environment variables:

`LAPACK_VERBOSITY=1` – tells MATLAB to display which BLAS/LAPACK library it is going to use when it starts up.

`BLAS_VERSION=$CSHOME/lib/libcsxl_blas.so` – tells MATLAB which BLAS/LAPACK library to use instead of its normal one.

`CS_HOST_BLAS=$MATLAB/bin/glnxa64/libacml.so` – tells CSXL where to find the normal BLAS/LAPACK library.

Where `$CSHOME` and `$MATLAB` point to the top of the CSXL and MATLAB installation trees respectively. Then just run MATLAB as normal.

N.B. Use `setenv LAPACK_VERBOSITY 1` with `csh` and `tcsh` shells, and `export LAPACK_VERBOSITY=1` with `sh` and `bash` shells.

The way the accelerated library is used is as follows:

- when a call is made to any BLAS/LAPACK routine this will be handled by CSXL in the first instance.
- if CSXL does not implement this particular routine, the call will be passed to the normal MATLAB library, for execution on the host computer.
- if CSXL does implement the routine, but the problem is too small for the accelerator card to provide any benefit (roughly anything smaller than about 450 elements square for matrix products), or too big to hold in the card's memory (roughly anything bigger than about 8050 square for matrix products), the call will be passed to the normal MATLAB library, for execution on the host computer.
- if CSXL implements the routine, and it is of a suitable size to be accelerated, the data will be transfered to the ClearSpeed card, processed, and then transfered back to the host computer.

The increases in speed for a simple matrix-matrix product, of various sizes are illustrated in figure 1. For small matrices the speed-up is about 200%, for those with the sizes, in the range 5500 square to 7100 square, performance is about 850% faster.

However, nearly all of the BLAS/LAPACK calls now have to pass through two libraries, which combined with the fact that large matrix-matrix products probably only make up a small proportion of your whole program, means that you should not expect more than about a 20% increase in performance over your program as a whole.

Using CSXL with MEX-files

Whilst CSXL does provide an accelerated LU factorization routine (`DGETRF`) this is not the LAPACK routine that MATLAB normally uses when performing "left divide" operations. If you need to solve very large systems of simultaneous equations, and you would like to make use of the ClearSpeed cards, you will need to produce a MEX-file that specifically calls `DGETSV` (which uses `DGETRF` and `DGETRS` to do the LU factorization and back substitution respectively).

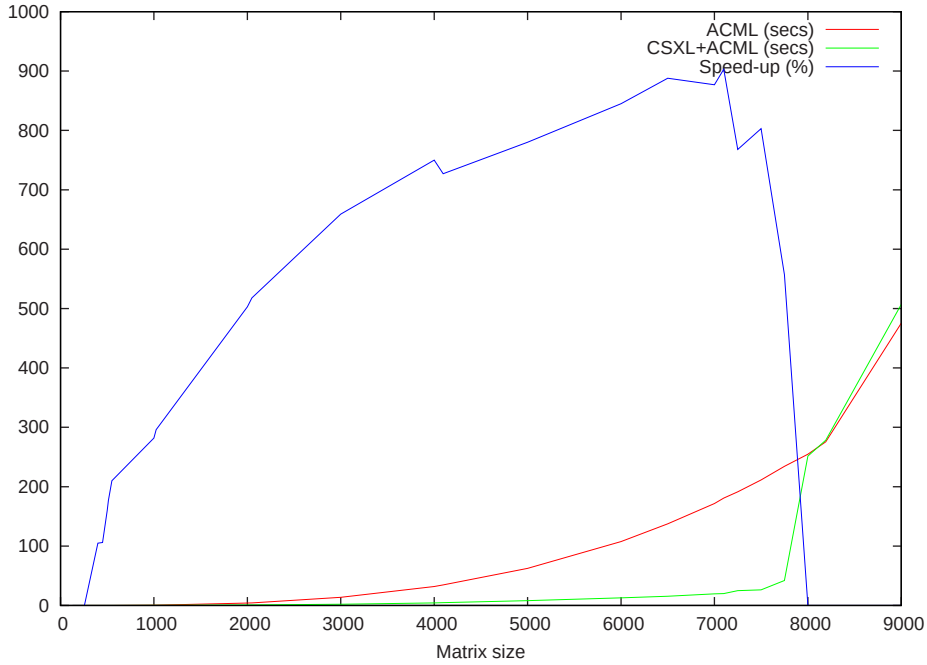

Figure 1: ClearSpeed acceleration of matrix-matrix products

In this case the ClearSpeed card appears not to be utilised for systems of less than about 1024 equations, the maximum size is determined by the amount of memory on the host system, and the peak performance improvement is about 350%, see figure 2.

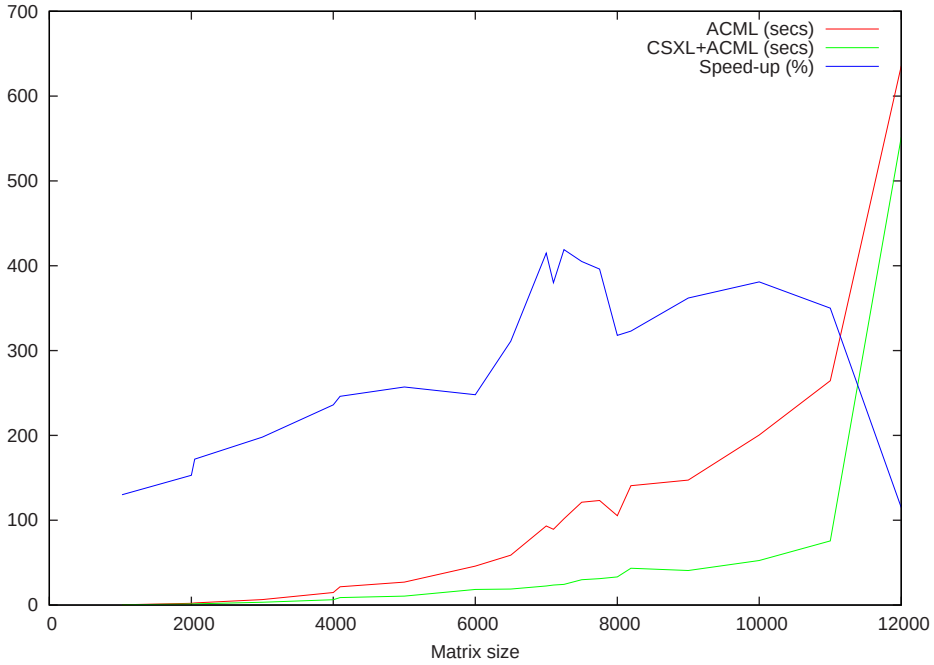

Figure 2: ClearSpeed acceleration of simultaneous equation solutions