Centro Euro-Mediterraneo sui Cambiamenti Climatici

Research Papers
Issue RP0167
February 2013

Scientific Computing and Operations Division (SCO)

TECHNICAL DOCUMENTATION L-BFGS for GPU-CUDA Reference Manual and User's Guide

By **Luisa D'Amore**
Universitá degli Studi di Napoli Federico II
luisa.damore@unina.it

Rossella Arcucci
Centro Euro-Mediterraneo sui Cambiamenti Climatici (CMCC)
rossella.arcucci@cmcc.it

Valeria Mele
Universitá degli Studi di Napoli Federico II
valeria.mele@unina.it

Giuseppe Scotti
Universitá degli Studi di Napoli Federico II
giuseppe.scotti@unina.it

and **Almerico Murli**
Centro Euro-Mediterraneo sui Cambiamenti Climatici (CMCC)
almerico.murli@cmcc.it

SUMMARY In this document is described a L-BFGS subroutine for large-scale constrained optimization algorithm implemented in GPU-CUDA. Here is provided a brief description of the L-BFGS algorithm developed and a brief tour around the main steps of the subroutine is provided. Significant efforts have been made to make the subroutine documenting, so this note should be seen as a prelude to looking at the code itself.

CMCC Research Papers

02

Centro Euro-Mediterraneo sui Cambiamenti Climatici

Contents

PURPOSE

`cuda_opt_unlp_solve` is designed to minimize an arbitrary smooth function not subject to constraints (which not include bounds on the variables) in GPU-CUDA environment [7]. The gradient of the function should be supplied by you. The Hessian matrix of second derivatives doesn't need to be evaluated directly. Each method and each necessary task into the optimization (for example the definition of the objective function and starting point) is coded into modules that can be replaced according to the application.

A list of capabilities is described below:

- **Modularity:** as mentioned before, each task into an optimization is separated one from another. It means that the code for the algorithm and the code of the objective function are different entities. For example, if the function has a new starting point, there is no need to modify the main code, but only a module related with it. If we need to change the objective function, there is no need to transform all the code, but only the module related with the function. If we want to change the line search routine with a custom version, a change in the module relating to the algorithm will suffice. This will allow the user to make minimum changes in all the codes in order to avoid programmer bugs.

- **Simplicity:** the modules are programmed in C for CUDA. Only C standards have been used, and the code will work with almost any C compiler. The code has been prepared for both an expert programmer as well as for a medium programmer.

- **Precision:** the modules can handle double precision (64-bit) just like the original sequential routine. This allows you to reach the the same accuracy of results.

- **Parallelism:** Any vector calculation procedure implemented in the sequential version of the software has been parallelized with one or more CUDA kernels in the GPU version. Linear algebra operations are performed using CUBLAS (CUDA Basic Linear Algebra Subprograms).

SPECIFICATION

```
#include "cu_lbfgs_dp.h"
#include "utils.h"
```

```
int cuda_opt_unlp_solve(int n, int m, void (*eval_fg)
(double *dev_x, double  *dev_objf, double *dev_grad, int n),
double *dev_x, double  *dev_objf, double *dev_grad,
double *dev_hess, double  *dev_workvec, double eps,
int *dev_istate)
```

Note: `dev` prefix indicates that the vectors shall be stored in the GPU memory.

DESCRIPTION

`cuda_opt_unlp_solve` is designed to solve the unconstrained minimization problem

$$\min f(x), \qquad x = (x_1, x_2, ..., x_n), \tag{1}$$

using the limited memory BFGS method [6].
The routine [2] is the CUDA version of Harwell Fortran routine VA15 [3] and it is especially effective on problems involving a large number of variables. In a typical iteration of this method an approximation H_k to the inverse of the Hessian is obtained by applying m BFGS updates to a diagonal matrix H_k^0, using information from the previous m steps.

The user specifies the number m, which determines the amount of storage required by the routine. The user may also provide the diagonal matrices H_k^0 if not satisfied with the default choice. The algorithm is described in [6] by Liu and Nocedal.
The user is required to calculate the function value f and its gradient g. In order to allow the user complete control over these computations, reverse communication is used. The routine must be called repeatedly under the control of the parameter `dev_istate`.
The steplength is determined at each iteration by means of the line search routine `MSRCH`, which is a slight modification of the routine `CSRCH` written by More' and Thuente [5].

A typical invocation would be:

```
cuda_opt_unlp_solve(n, m, evaluate_fg, d_x, d_f, d_g, d_diag, d_w,
eps, d_istate);
```

You must supply an initial estimate of the solution to (1), together with functions.

ARGUMENTS

1. **n** - `int` *(Input)*
   On entry: n, the number of variables.
   Constraint: $\mathtt{n} > 0$
2. **m** - `int` *(Input)*
   On entry: m, the number of corrections used in the BFGS update.
   It is not altered by the routine. Values of m less than 3 are not recommended; large values of m will result in excessive computing time. $3 \le m \le 7$ is recommended.
   Constraint: $\mathtt{m} > 0$
3. `eval_fg` - function, supplied by the user (*External Function*)
   `eval_fg` must be a CUDA function that calculates the value of the nonlinear function f and the gradient $g(x) = (\frac{\partial F}{\partial x})$ for a specified n element vector x.
   Its specification is: `void (`**eval_fg**`) (double `***dev_x**`, double `***dev_objf**`, double`***dev_grad**`, int `**n**`)`, where:
   (a) **dev_x** - `double *` *(Input)*
   On entry: x, the vector of variables at which the objective function and/or all available elements of its gradient are to be evaluated.
   (b) **dev_objf** - `double *` *(Output)*
   On exit: `dev_objf` must be set to the value of the objective function at x_k.

(c) **dev_grad** - `double *` (*Output*)
On exit: `dev_grad` must return the available elements of the gradient evaluated at x_k, i.e., `dev_grad[i-1]` contains the partial derivative $\frac{\partial F}{\partial x_i}$.

(d) **n** - `int` (*Input*)
On entry: n, the number of variables.

4. **dev_x** - `double *` (*Input/Output*)
On entry: x, an estimate of the solution at iterate k. On exit.
On exit: x, with `dev_istate = 0`, contains the values of the variables at the best point found (usually a solution).

5. **dev_objf** - `double *` (*Input*)
On entry: Before initial entry and on a re-entry with `dev_istate = 1`, it must be set by the user to contain the value of the objective function F at the point x.

6. **dev_grad** - `double *` (*Input*)
On entry: Before initial entry and on a re-entry with `dev_istate = 1`, it must be set by the user to contain the components of the gradient g evaluated at the point x.

7. **dev_hess** - `double *` (*Input*)
On entry: If global variable `diagco=TRUE`, then on initial entry or on re-entry with `dev_istate = 2`, the array of length n, `dev_hess`, must be set by the user to contain the values of the diagonal matrix H_k^0. It needs not be initialized if the default option is used and will be set to the identity.
`void enable_first_Hessian();` Constraint: all elements of `dev_hess` must be positive.

8. **dev_workvec** - `double *` (*Input*)
On entry: `dev_workvec` is an array of length $n(2m+1)+2m$ used as workspace for `cuda_opt_unlp_solve`. This array must not be altered by the user. The work vector `dev_workvec` is divided as follows:

   (a) the first n locations are used to store the gradient and other temporary information.

   (b) locations $(n+1)...(n+m)$ store the scalars ρ.

   (c) locations $(n+m+1)...(n+2m)$ store the numbers α used in the formula that computes Hg.

   (d) locations $(n+2m+1)...(n+2m+2mn)$ store the last m search steps.

   (e) locations $(n+2m+nm+1)...(n+2m+2nm)$ store the last m gradient differences.

   The search steps and gradient differences are stored in a circular order.

9. **eps** - `double` (*Input*)
On entry: `eps` is a positive variable that must be set by the user, and determines the accuracy with which the solution is to be found. The subroutine terminates when $||g|| <$ `eps`$\cdot \max(1, ||x||)$, where $||.||$ denotes the Euclidean norm. By default `eps` = 10^{-5}. The subroutine ends even when the number of iterations is greater than 2000.

10. **dev_istate** - `int *` (*Input*)
On entry: `dev_istate` is an integer variable that must be set to 0 on initial entry to the subroutine. A return with `dev_istate` < 0 indicates an error, and `dev_istate` = 0 indicates that the routine has terminated without detecting errors.
On a return with `dev_istate` = 1, the user must evaluate the function F and gradient g. On a return with `dev_istate` = 2, the user must provide the diagonal matrix H_k^0.
The following negative values of `dev_istate`, detecting an error, are possible:

   (a) `dev_istate` = -1 The line search routine `MCSRCH` failed. The parameter `info` provides more detailed information (see also the documentation of `MCSRCH`):

i. `info` = 0 improper input parameters.
ii. `info` = 2 relative width of the interval of uncertainty is at most `XTOL`.
iii. `info` = 3 more than 20 function evaluations were required at the present iteration.
iv. `info` = 4 the step is too small.
v. `info` = 5 the step is too large.
vi. `info` = 6 rounding errors prevent further progress. there may not be a step which satisfies the sufficient decrease and curvature conditions. tolerances may be too small.

(b) `dev_istate` = -2 The $i-th$ diagonal element of the diagonal inverse Hessian approximation, given in `dev_hess`, is not positive.

(c) `dev_istate` = -3 Improper input parameters for `cuda_opt_unlp_solve` (n or m are not positive).

EXAMPLE OF CALLING PROGRAM

Here is a simple example of a `main program`, that is a driver to minimize the Extended Rosenbrock function using our function `cuda_opt_unlp_solve`.
It shows how the function is called from the main program and which parts of the code you need to edit for a specific problem. Template files included with the software allows you to easily use the routine changing and adding a few lines of code.
In this case, the minimized function is the sum of several terms, each one of the same mathematical form:

$$f(x) = \sum_{i=1}^{n/2} c(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2, \quad x_0 = [-1.2, 1, ..., -1.2, 1]. \quad c = 100. \tag{2}$$

This function is also known as Extended Rosenbrock.
In these examples some code has been omitted for clarity.

MAIN PROGRAM

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cublas.h>
#include "driver.h"
#include "functions.h"
#include "cu_lbfgs_dp.h"

int deviceID;

/* OPTIONAL: supporting for function and gradient evaluations */
int red_blocks, red_threads;
double *d_odata, *d_temp;

int main(int argc,char *argv[]){
        /* CUDA vectors */
        double *dev_x, *dev_g, *dev_diag, *dev_w;
        int *d_iflag;
        static double *d_f;

        static int m, n;
        int dim_w;
        int numThreadsPerBlock, numBlocks;

        cudaDeviceProp deviceProps;
        int deviceCount = 0;
        double eps = 1e-5;
        /* Check for the correct execution */

        n = atoi(argv[1]);
        m = atoi(argv[2]);
        dim_w = n + 2 * m + 2 * m * n;

        /* Execution configuration */
        numThreadsPerBlock = 512;
        numBlocks = n/numThreadsPerBlock + ( n % numThreadsPerBlock == 0 ? 0 : 1);

        /* Check for CUDA devices */
        cudaGetDeviceCount(&deviceCount);
        if (deviceCount == 0) { exit(0); }
        /* Selects the fastest GPU */
        deviceID = get_best_device_id();
        cudaSetDevice(1);

        /* Cuda start */
        cudaError_t cudaStat;
        cublasInit();
        cudaStat = cudaMalloc((void **) &d_f, sizeof(double));
        cudaStat = cudaMalloc((void **) &d_iflag, sizeof(int));
        cudaStat = cudaMalloc((void **) &dev_x, n * sizeof(double));
        cudaStat = cudaMalloc((void **) &dev_diag, n * sizeof(double));
        cudaStat = cudaMalloc((void **) &dev_g, n * sizeof(double));
        cudaStat = cudaMalloc((void **) &dev_w, dim_w * sizeof(double));
        if(cudaStat != cudaSuccess){
                printf(" Error: The device memory allocation failed.\n");
                return -1;
        }
        /* Function-dependent code */

        /* DEFINITION OF STARTING POINT (FUNCTION-DEPENDENT) */
        x_rosen_kernel<<<numBlocks , numThreadsPerBlock>>>(dev_x, n);
        /* Memory usage */
        gpu_mem_info();

        /* For the reduction */
        getNumBlocksAndThreads(n / 2, &red_blocks, &red_threads);
        cudaStat = cudaMalloc((void**) &d_odata, red_blocks * sizeof(double));
        cudaStat = cudaMalloc((void **) &d_temp, (n / 2) * sizeof(double));
        if(cudaStat != cudaSuccess){
                return -1;
        }

        /* -------- CALL THE SOLVER -------- */
        cuda_opt_unlp_solve(n, m, evaluate_fg, dev_x, d_f, dev_g, dev_diag, dev_w, eps, d_iflag);

        /* deallocation of device memory */
        cudaFree(dev_x);
        cudaFree(dev_g);
        cudaFree(dev_diag);
        cudaFree(dev_w);
        cudaFree(d_f);
        cudaFree(d_iflag);
        cudaFree(d_odata);
        cudaFree(d_temp);
        return 0;
}
```

PROBLEM-DEPENDENT CODE

In the code below are shown the segments which the user must define depending on the particular function to be minimized (files: `driver.cu` and `functions.cu`).
For the evaluation of the function and the gradient, you have to define the pointer of `cuda_opt_unlp_solve` to member function for the evaluation of both f and g, this allows to have more flexibility in writing code optimized for the GPU.

```
/* Custom routine for evaluate function and gradient */
void evaluate_fg(double *dev_x, double *dev_f, double *dev_g, int n){
        /*  execution configuration for the reduction kernel */
        int numThreadsPerBlock = 512;
        int numBlocks;
        /* Evaluate the function and its gradient in x */

        numBlocks = (n/2) / numThreadsPerBlock + ((n/2) % numThreadsPerBlock == 0 ? 0 : 1);
        fg_rosen_kernel<<<numBlocks , numThreadsPerBlock>>>(d_temp, dev_g, dev_x, n / 2);
        /* d_odata di utilità per la valutazione di rosenbrock è globale per non doverla riallocare
             ogni volta */
        my_reduce(n/2, red_threads, red_blocks, d_temp, d_odata, dev_f);
        cudaThreadSynchronize();
}

/** @brief KERNEL 1.1: DEFINING THE STARTING POINT (ROSENBROCK)\n To change according to the functional
      to be minimized */
__global__ void x_rosen_kernel(double *dev_x, int size){

        const int tid = blockDim.x * blockIdx.x + threadIdx.x;
        if(tid < size){
                if(tid % 2 == 0) { dev_x[tid] = -1.2; }
                else{ dev_x[tid] = 1.0; }
        }
}

/** @brief KERNEL 2.1: FOR THE CALCULATION OF THE FUNCTION AND GRADIENT (ROSENBROCK) */
__global__ void fg_rosen_kernel(double *d_tmp, double *dev_g, double *dev_x, int size){
        double c = 100.0;
        const int tid = blockDim.x * blockIdx.x + threadIdx.x + 1;
        const int k = tid * 2;
        if(tid <= size){
                d_tmp[tid - 1] = c * pow((dev_x[2 * tid - 1] - pow(dev_x[2 * tid - 2], 2)), 2) + pow
                     ((1.0 - dev_x[2 * tid - 2]), 2);
                dev_g[k - 2] = -4.0 * c * dev_x[2 * tid - 2] * (dev_x[2 * tid - 1] - pow(dev_x[2 * tid
                     - 2], 2)) - 2.0 * (1.0 - dev_x[2 * tid - 2]);
                dev_g[k - 1] = 2.0 * c * (dev_x[2 * tid - 1] - pow(dev_x[2 * tid - 2], 2));
        }
}
```

COMPILING AND RUNNING

This section explains how to compile and run a CUDA based L-BFGS application from the command line. Here are the steps you need to follow:

1. Download and install the latest Graphic Card driver and CUDA Toolkit, if you haven't already done so;
   - You can download the latest release of your GPU-CUDA driver for free from: `http://www.nvidia.com/Download/index.aspx?lang=en-us`.
     Recently, developers can download the latest CUDA Toolkit, SDK, and drivers at `https://developer.nvidia.com/cuda-toolkit`.
2. Create a program that uses parallel `cuda_opt_unlp_solve` routine;
   - Create a program that calls `cuda_opt_unlp_solve` components or make small changes to the provided driver, written to minimize the Extended Rosenbrock function.
3. Compile the program;
   - To simplify compilation on different systems the driver program comes with a Makefile. This is pre-configured for compilation of the routine for Linux systems with the NVIDIA TESLA C1060 graphic card. The main modification of the Makefile depends on the class of NVIDIA GPU architectures for which the CUDA input files must be compiled.
     For example, if you want to compile the driver for a Linux system with *GPU Nvidia GeForce GTX 560 Ti*, with 2.0 virtual architecture, you have to update the variable `GPUARCH` in the Makefile with the

Virtual Architecture Feature List (from the User Guide)	
compute_10	Basic features
compute_11	+ atomic memory operations on global memory
compute_12	+ atomic memory operations on shared memory
	+ vote instructions
compute_13	+ double precision floating point support
compute_20	+ Fermi support
compute_30	+ Kepler support

value `compute_20`.

Note: visit the NVIDIA main site to control which *virtual* GPU architecture to compile for (i.e., which version of PTX to emit).

The NVIDIA graphics driver and CUDA compiler are already installed on machines that support CUDA. However, one must set some environment variables in order to run and write CUDA enabled programs.
If you are unable to compile, make sure you are using the compiler in a recent release of the CUDA Toolkit. You can verify the version of your CUDA compilation tools using these commands:

```
nvcc -version
```

Once you've updated your CUDA Toolkit, you should be able to use the programs without changes.

4. Run the program.

   - After you compile the driver successfully, you can run it:

```
./driver_program 1000000 7
```

The parameters that the driver require to be run are two: the number of variables of the function `n` and the number of secant updates of the Hessian matrix `m`. The first two parameters are required. Whithout any change, it will minimize the Extended Rosenbrock function. User can change the problem dependent code (see section) to minimize a different one.

PROGRAM RESULTS

The following is the text output generated by running the provided driver:

```
Function to minimize:  Rosenbrock
Xtol:  2.220446e-16
Number of variables n = 1000000;
Number of corrections m = 7;
f = 1.210000e+07;
gnorm = 1.646623e+05
***********************************

     I      NFN    FUNC            GNORM           STEPLENGTH
     1      4      8.968026e+06    1.338990e+05    1.275337e-04
     2      5      2.223223e+06    1.835022e+04    1.000000e+00
     3      6      2.071674e+06    2.874521e+03    1.000000e+00
     4      7      2.066929e+06    1.253213e+03    1.000000e+00
     5     8     2.064897e+06    1.409016e+03    1.000000e+00
     6     9     2.050814e+06    3.691128e+03    1.000000e+00
           ...
     32    46    3.391364e+02    1.188979e+02    1.000000e+00
     33    47    4.085484e+01    2.639775e+02    1.000000e+00
     34    48    2.229921e+00    1.018714e+01    1.000000e+00
     35    49    1.162750e-02    1.781043e+00    1.000000e+00
     36    50    4.912961e-06    7.652340e-02    1.000000e+00
     37    51    6.752264e-10    1.013915e-03    1.000000e+00

The m̃inimization terminated without detecting errors.


Number of function and gradient evaluations for L-BFGS: 51
Total time for the function and gradient evaluations:   66.000000 ms
Average time for the function and gradient evaluations: 1.294118 ms
Number of l-bfgs calls:                                 51
Total time for the execution of l-bfgs:                 398.000000 ms
Average time for the executions of l-bfgs:              7.803922 ms
Total time for the execution of mcstep:                 0.000000 ms
Total time for the execution of CUBLAS routines:        324.000000 ms
Number of calls to mcsrch:                              87
Average time for the execution of mcsrch:               0.643678 ms
Total time for the execution of mcsrch:                 56.000000 ms
Total time for the minimization:                        464.000000 ms
```

OTHER TESTING FUNCTIONS

As previously seen, `cuda_opt_unlp_solve` comes with a driver program which shows the behavior of l-bfgs and the performance achieved with the extended Rosenbrock function (`L-BFGS\CUDA\Driver\driver.cu`). There are also other functions from CUTE collection [1] to test the behavior of the routine (`L-BFGS\CUDA\Driver\functions.cu`). For each of them the start point and the optimal solution are known.
We provide some function (objective function and problem characteristics definition) that user can change in the driver without effort.
The first function is the Extended Rosenbrock previously seen:

$$f(x) = \sum_{i=1}^{n/2} c(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2, \quad x_0 = [-1.2, 1, ..., -1.2, 1]. \quad c = 100. \tag{3}$$

The second is the Extended Beale function:

$$\begin{aligned} f(x) = \sum_{i=1}^{n/2} (1.5 - x_{2i-1}(1 - x_{2i}))^2 + (2.25 - x_{2i-1}(1 - x_{2i}^2))^2 \\ + (2.625 - x_{2i-1}(1 - x_{2i}^3))^2, \quad x_0 = [1, 0.8, ..., 1, 0.8]. \end{aligned} \tag{4}$$

The third is the Extended Powell function:

$$\begin{aligned} f(x) = \sum_{i=1}^{n/4} (x_{4i-3} + 10x_{4i-2})^2 + 5(x_{4i-1} - x_{4i})^2 + (x_{4i-2} - 2x_{4i-1})^4 \\ + 10(x_{4i-3} - x_{4i})^4, \quad x_0 = [3, -1, 0, 1, ..., 3, -1, 0, 1]. \end{aligned} \tag{5}$$

The fourth is the Extended Wood function:

$$\begin{aligned} f(x) = \sum_{i=1}^{n/4} 100(x_{4i-3}^2 - x_{4i-2})^2 + (x_{4i-3}-1)^2 + 90(x_{4i-1}^2 - x_{4i})^2 \\ +10.1\{(x_{4i-2}-1)^2 + (x_{4i}+1)^2\} + 19.8(x_{4i-2}-1)(x_{4i}-1), \\ x_0 = [-3, -1, ..., -3, -1]. \end{aligned} \tag{6}$$

The fifth is an Extended Trigonometric function:

$$\begin{aligned} f(x) = \sum_{i=1}^{n}((n - \sum_{j=1}^{n} \cos x_j) + i(1 - cosx_i) - sinx_i)^2, \\ x_0 = [0.2, 0.2, ..., 0.2]. \end{aligned} \tag{7}$$

OPTIONAL PARAMETERS CONFIGURATION

Several optional parameters in `cuda_opt_unlp_solve` define choices in the problem specification or the algorithm logic. In order to reduce the number of formal arguments of the routine these optional arguments have associated default values that are appropriate for most problems. Therefore, you need only specify those optional arguments whose values are to be different from their default values. The remainder of this section can be skipped if you wish to use the default values for all optional arguments.
Optional parameters may be specified by calling the relative function, indicated in the description of each one. A complete list of optional parameters and their default values is given:

1. **diagco** - `int` **(Default = 0)**
   `diagco` is a logical variable that must be set to 1 if the user wishes to provide the diagonal matrix H_k^0 at each iteration. Otherwise it should be set to 0, in which case `cuda_opt_unlp_solve` will use a default value. If `diagco` is set to 1 the routine will return at each iteration of the algorithm with `dev_istate = 2`, and the diagonal matrix H_k^0 must be provided in the array `dev_hess`.
   To change the default value, you must call `enable_first_Hessian()` before `cuda_opt_unlp_solve` calls.
2. **eps** - `int *` **(Default = 1e-05)**
   `eps` is the error tolerance. It determines the accuracy with which the solution is to be found.
   To change the default value, you must call `change_eps(double new_eps)` before `cuda_opt_unlp_solve` calls.
3. **iprint** - `int *` **(Default = (1,0))**
   is an integer array of length two which must be set by the user.
   `iprint[1]` specifies the frequency of the output:
   (a) `iprint[1]` $<$ 0 : no output is generated;
   (b) `iprint[1]` = 0 : output only at first and last iteration;
   (c) `iprint[1]` $>$ 0 : output every `iprint[1]` iterations.

   `iprint[2]` specifies the type of output generated:
   (a) `iprint[2]` = 0 : iteration count, number of function evaluations, function value, norm of the gradient, and steplength;
   (b) `iprint[2]` = 1 : same as `iprint[2]` = 0, plus vector of variables and gradient vector at the initial point;
   (c) `iprint[2]` = 2 : same as `iprint[2]` = 1, plus vector of variables;

(d) `iprint[2]` = 3 : same as `iprint[2]` = 2, plus gradient vector.

To change the default value of **iprint**, call `set_iprint(int i_one, int i_two)` before `cuda_opt_unlp_solve` calls.

4. **GTOL** - double **(Default = 0.9)**
   `GTOL` is a variable which controls the accuracy of the line search routine `MCSRCH`. If the function and gradient evaluations are inexpensive with respect to the cost of the iteration (which is sometimes the case when solving very large problems) it may be advantageous to set `GTOL` to a small value.
   A typical small value is 0.1.
   Constraint: `GTOL` $>$ 1e-04.
5. **STPMIN** and **STPMAX** - double **(Default = 10^{-20} and 10^{20})**
   are non-negative variables which specify lower and upper bounds for the step in the line search. Their default values are 1e-20 and 1e+20, respectively. These values need not be modified unless the exponents are too large for the machine being used, or unless the problem is extremely badly scaled (in which case the exponents should be increased).
6. **XTOL** - double **(Default = machine dependent)**
   `XTOL` must be set by the user to an estimate of the machine precision. The line search routine will terminate if the relative width of the interval of uncertainty is less than `XTOL`.
7. **maxfev** - `int` **(Default = 20)**
   `maxfev` indicates the maximum number of evaluations of the function and the gradient per iteration.
8. **icall** - `int` **(Default = 2000)**
   `icall` indicates the maximum number of iterations.

UTILITY FUNCTIONS

There are some behaviors of the software that the user must take into account:

- At the end of execution, the result of the minimization is only present in the global memory of the device for future processing.
  To copy the vector of the solution from the host to the device, you can use the following function:
  `void get_solution(double *h_x, double *dev_x, int n);`
  where `h_x` is the output vector of size `n` present on the host and `dev_x` is the counterpart on the device.
- The routine in the presence of multiple GPUs with CUDA support, select the fastest.
  If you want to manually select the GPU to use, there is the function:
  `void set_device_ID(int id);`
  where `id` is the integer that represents the device ID (CUDA picks the fastest device as device 0). The function must be used before any CUDA call.
- If you want to print the first `n` values of a vector in global memory, for the purpose of debugging, you can use the function:
  `void print_vector(double *d_v, int size)` where `d_v` is the vector to print. It allocates the memory needed for printing in main memory.
- The evaluation of some particular function (as Rosenbrock) may require a reduction function (hopefully parallel). With the minimization routine described here is also provided a good parallel reduction function that comes with the NVIDIA SDK:
  ```
  void my_reduce(int n, int numThreads, int numBlocks,
  int maxThreads, int maxBlocks, double* d_idata, double *d_odata, double
  *d_result)
  ```
  For details visit: `http://www.nvidia.com/content/cudazone/cuda_sdk/Data-Parallel_Algorithms.html`

ERROR INDICATORS AND WARNINGS

All errors and warnings arising from incorrect execution of the algorithm are shown on the screen as text. These are also encoded in some variables as seen in section . If the input parameters are correct, a failure of the routine is almost always due to a failure of the subroutine `mcsrch` for linear search. Indeed, the program reports a warning or an error in the following situations:

- `GTOL` is a double precision variable with default value 0.9, which controls the accuracy of the line search routine `mcsrch`. If the function and gradient evaluations are inexpensive with respect to the cost of the iteration (which is sometimes the case when solving very large problems) it may be advantageous to set `GTOL` to a small value. A typical small value is 0.1.
  Restriction: `GTOL` should be greater than 1.E-04. If `GTOL` is less than this threshold, the program sets the variable to the default value and continue.
- The repeated execution of the subroutine `lbfgs` is kept under control with the variable `d_iflag`, which is directly related to the parameter `dev_istate` of `cuda_opt_unlp_solve`. A return with `d_iflag`=-1 indicates that the line search routine `mcsrch` failed due to errors in function, gradient, or tolerances. In this case, the value of `info` provides further information (use this list because no message is shown on the screen):
  - `info`=0, improper input parameters to the subroutine `mcsrch`.
  - `info`=-1, a return is made to compute the function and gradient.
  - `info`=1, the sufficient decrease condition and the directional derivative condition hold.
  - `info`=2, relative width of the interval of uncertainty is at most `xtol` (the machine precision).
  - `info`=3, number of function and gradient evaluations has reached `maxfev`.
  - `info`=4, the step is at the lower bound `STPMIN`.
  - `info`=5, the step is at the upper bound `STPMAX`.
  - `info`=6, rounding errors prevent further progress. there may not be a step which satisfies the sufficient decrease and curvature conditions. Tolerances may be too small.
- It may happen that, for a particular problem, the search direction is not descent. In this case, the subroutine `mcsrch` ends with an error.
- A common error occurs when the memory capacity of the GPU is not enough to contain the array **dev_workvec** which constitutes the space complexity of the algorithm. This error, which is managed by the CUDA environment, could be solved partially by reducing the value of m.

ALGORITHMIC DETAILS

Here we report the algorithm in detail and some considerations on the variables involved.

(1) Choose x_0, m, $0 < \beta' < 1/2$, $\beta' < \beta < 1$, and a symmetric and positive definite starting matrix H_0. Set $k = 0$,
(2) Compute

$$d_k = -H_k g_k, \tag{8}$$

$$x_{k+1} = x_k + \alpha_k d_k, \tag{9}$$

where α_k satisfies the Wolfe conditions:

$$f(x_k + \alpha_k d_k) \leq f(x_k) + \beta' \alpha_k g_k^T d_k, \tag{10}$$

$$g(x_k + \alpha_k d_k)^T d_k \geq \beta g_k^T d_k. \tag{11}$$

(The first attempt is made with steplength $\alpha = 1$)
(3) Let $m = \min k, m - 1$. Update H_0, $\hat{m} + 1$ times using the pairs $\{y_j, s_j\}_{j=k-\hat{m}}^{k}$, i.e let

$$\begin{aligned} H_{k+1} \quad &= (V_k^T ... V_{k-\hat{m}}^T) H_0 (V_{k-\hat{m}} ... V_k) \\ &+\rho_{k-\hat{m}} (V_k^T ... V_{k-\hat{m}+1}^T) s_{k-\hat{m}} s_{k-\hat{m}}^T (V_{k-\hat{m}+1} ... V_k) \\ &+\rho_{k-\hat{m}+1} (V_k^T ... V_{k-\hat{m}+2}^T) s_{k-\hat{m}+1} s_{k-\hat{m}+1}^T (V_{k-\hat{m}+2} ... V_k) \\ &... \\ &+\rho_k s_k s_k^T. \end{aligned} \tag{12}$$

(4) Set $k := k + 1$ and go to 2.

First of all the user specifies the amount of storage to be used, by giving a number m, which determines the number of matrix updates of the inverse Hessian H_k that can be stored. The more updates are stored, the more accurate will be the approximate Hessian. However, the more vectors are stored, the higher will be the cost of each iteration. The default value is likely to give a robust algorithm without significant expense, but faster convergence can sometimes be obtained with significantly fewer updates.

Regard to the Wolfe conditions, as the Harwell subroutine VA15, the line search of `cuda_opt_unlp_solve` is terminated when

$$|g(x_k + \alpha_k d_k)^T d_k| \leq \beta g_k^T d_k. \tag{13}$$

is satisfied. ((13) is stronger than (5), which is useful in practice). We use the values $\beta' = 10^{-4}$ and $\beta = 0.9$, which are recommended in [6].

The initial Hessian H_k^0 is approximated by the identity matrix, and after one iteration is completed, the methods update it with $\gamma_0 I$ instead of I, where

$$\gamma_0 = y_0^T s_0 / ||y_0||^2 \tag{14}$$

In this way is also introduced a scale in the algorithm.

STOPPING CRITERIA

As inferred from the above, the program terminates unexpectedly when an error occurs or, properly, when one of the three conditions is verified:

1. $||g|| <$ `eps`$\cdot \max(1, ||x||)$ at the point x_k
2. The number of iterations is greater than `icall`.
3. The number of f evaluations per iteration is greater than `maxfev`.

For some problems, the default values of `eps`, `icall` or `maxfev` are too restrictive, and the algorithm terminates prematurely while calculating best estimates.
Considering for example the extended Powell function implemented in the program driver: with $n = 1.0E06$, $m = 7$, `icall`=2000, `maxfev`=20 and `eps`=1.0E-5, the method terminates for the third criterion. This is a signal that the tolerance for the first stopping criterion is too low. Increase `eps` or m can help in this case.

Hardware		Tesla C1060 specifications	
Processor	Intel® Core$^{\mathrm{TM}}$i7 (3.07GHz)	Streaming Multiprocessors	30
RAM	12GB of DRR3 1333	Streaming Processor cores	240
Hard disk	500GB sata, 16MB cache, 7.200	SP core Frequency	1300 MHz
GPU	2x Tesla C1060 4GB RAM	Memory Bandwidth	102GB/s

Table 1
Computing hardware specifications.

TESTING

In this section we compare the sequential and widespread version of L-BFGS, developed by J. Nocedal, with `cuda_opt_unlp_solve` in terms of time and accuracy, for the minimization of the Rosembrock function.
The experimental results were obtained using the following computing hardware:
The GPU of C1060 card supports IEEE standard for binary double precision floating-point arithmetic (IEEE 754-1985 [4]). Figure 1 shows the relative error between the estimates obtained at each iteration by the L-BFGS Harwell routine and its GPU parallel porting. Only a slight loss in accuracy is measured between the 30-th and the 35-th iteration.

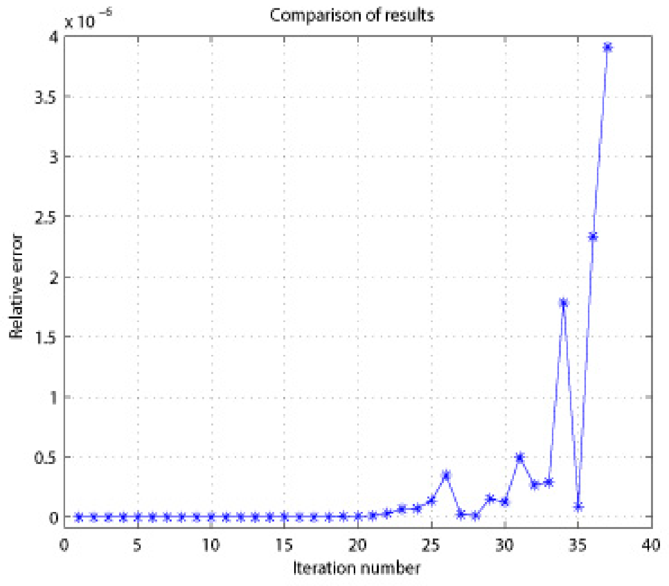

Figure 1:
Relative error between the estimates obtained at each iteration of the parallel and sequential software on Rosenbrok function with $n = 10^6$ and $m = 7$.

As stopping criterion of all runs was used the following:

$$||g_k|| < 10^{-5} \cdot \max(1, ||x_k||) \qquad (15)$$

where g_k is the projected gradient at the k-th step. The problem size was selected to be large enough so that GPU execution time could measured reliably, so the number of variables ranges from 5×10^5 to 5×10^7.
We achieved speedup of over $8\times$ in double precision and $14\times$ in single precision. For large scale problems

the performance gain is increasingly more pronounced, as is shown in Figure 2

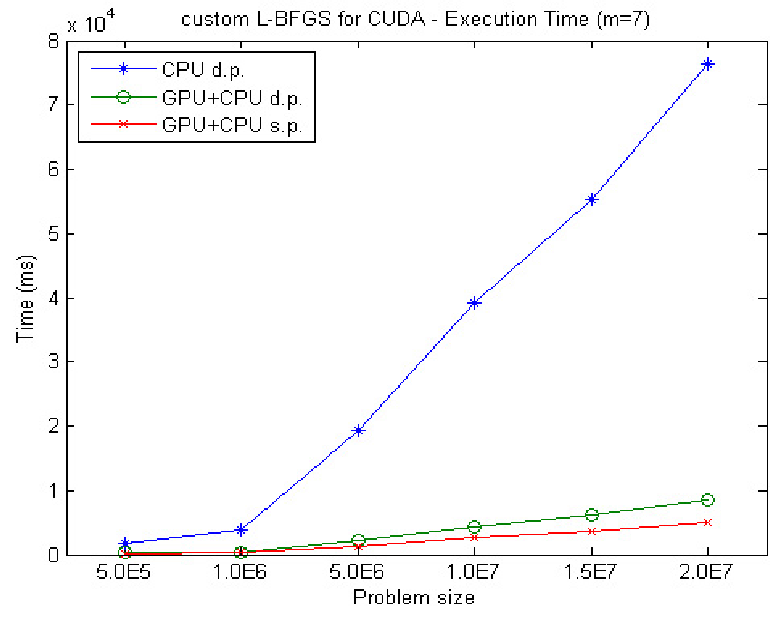

Figure 2:
Execution times for the minimization of the extended *Rosenbrock* function, using TESLA as parallel computing architecture.

`cuda_opt_unlp_solve` is the result of several optimization steps, in each of which the source code profiling was crucial. We have used the nVIDIA's proprietary CUDA Visual Profiler [9].

The graph in Figure 3 indicates the percentage by which each kernel affects the overall computation time. Observe that the most expensive part is related to routines of CUBLAS (`ddot` and `daxpy`), not due to a customization. Furthermore, the part inherently sequential affects a small part of the total time of calculation. To verify both the portability of our implementation, and the scalability of the parallel programming model we decided to execute the same parallel code on a more recent CUDA architecture. The result in Figure 4 were obtained by performing tests on a machine equipped with a FERMI 2.1 GPU.

Hardware		Tesla C1060 specifications	
Processor	Intel® Core^{TM}2 Quad (2.83GHz)	Streaming Multiprocessors	8
RAM	4GB of DRR3 1333	Streaming Processor cores	384
Hard disk	500GB sata, 16MB cache, 7.200	SP core Frequency	822 MHz
GPU	GeForce GTX 560 Ti 1GB RAM	Memory Bandwidth	128.27GB/s

Table 2
FERMI Computing hardware specifications.

Even if the code is not optimized for the last FERMI architecture and the memory constraints are more restrictive, the CUDA routine allows us to get quickly very satisfactory performance gain (20-30% s.p., 30-50% d.p.), without additional efforts.
Furthermore the first computing environment (the TESLA), though older, has a faster processor for single-threaded execution compared with the computing hardware using the FERMI architecture. The fact that the

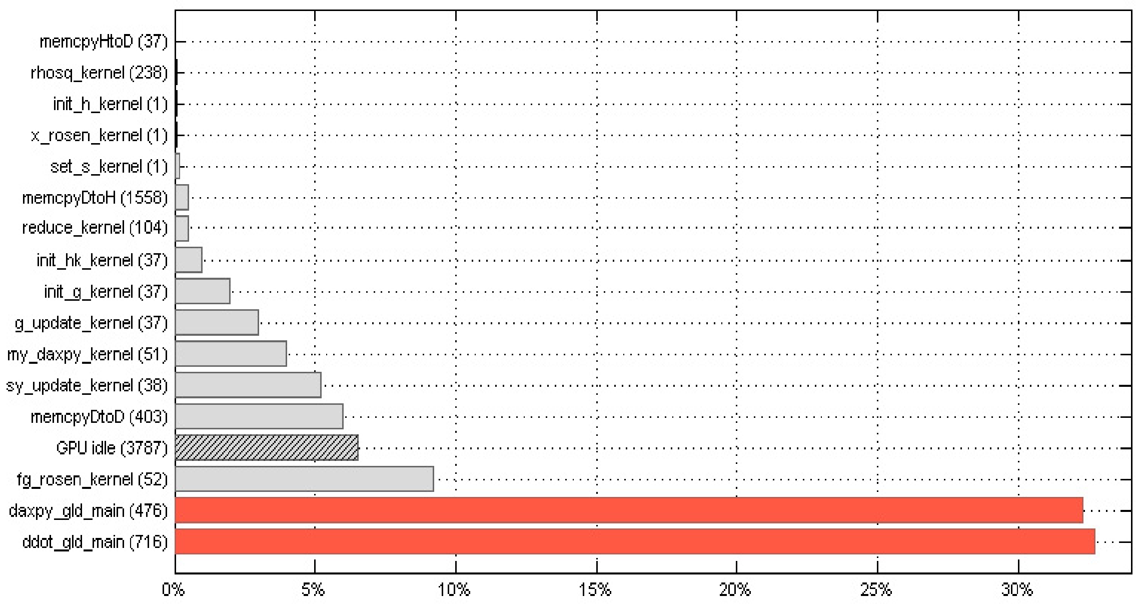

Figure 3:
Percentage of kernel execution time over total time, per kernel. Kernels using CUBLAS are the two ones on the bottom. CPU execution time is marked with stripes. In brackets the number of executions for a minimization of the *Rosenbrock* Function.

parallel software is twice as fast on the FERMI architecture shows that the sequential part of the code has been greatly reduced.

The Harwell routine VA15 provides that the entire required memory space is known and therefore allocated before any FLOP. It also provides that the space is organized in a contiguous sequence of vectors to be processed, called *Work Vector*. In Figure 5 a graphical representation is shown.

The GPU can only process the data in its global memory. For parallel processing is therefore necessary to first transfer the input data from CPU memory to the GPU.
To avoid continuous relatively slow data transfer from the host to the device, it was decided to store the entire Work Vector in the device prior to any processing. In this way, the overhead is minimized. A drawback of this kind of storage is that since the device memory is small with respect to the one on the host, the size of the problem that can be solved is limited by the size of the Work Vector that the GPU can handle locally.

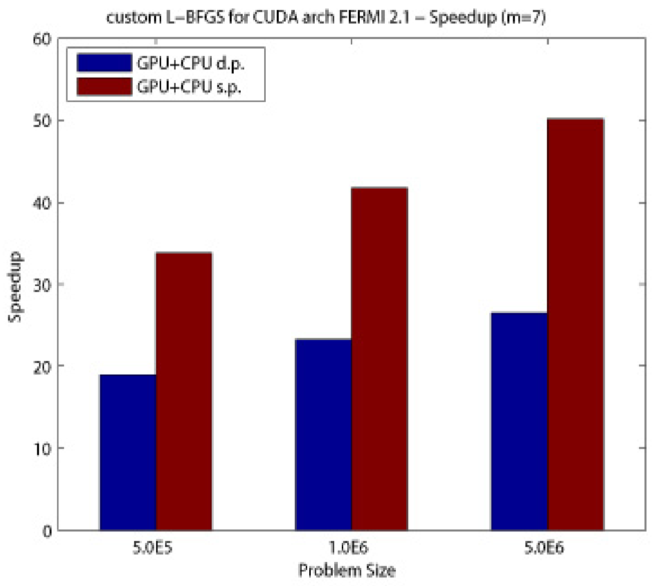

Figure 4:
Speed up achieved by parallel programs with FERMI architecture in single and double precision.

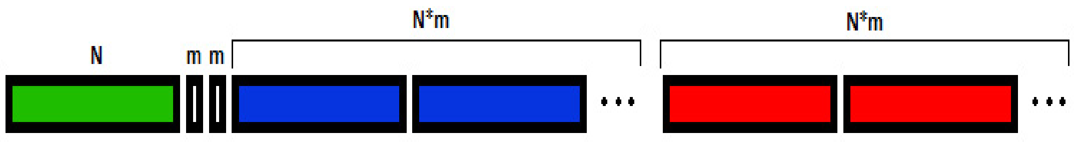

Figure 5:
Work Vector for the L-BFGS algorithm. In green are shown the first N locations used to store the gradient and other temporary information; from memory location n+1 to n+m there are the m ρ scalars; from n+m+1 to n+2m there are α numbers used in the formula for the calculation of Hg; in the following successive m*n location, colored in blue, are kept the last m search steps; in the last m*n locations, colored in red, we find the m gradient differences.

Bibliography

[1] I. Bongartz, A.R. Conn, N.I.M. Gould, and Ph.L. Toint, "CUTE: Constrained and Unconstrained Testing Environment," ACM Trans. Math. Software, Volume 21, 1995.

[2] L. D'Amore, G. Laccetti, D. Romano, G. Scotti, and A.Murli, "Towards a parallel component in a GPU-CUDA environment: a case study with the L-BFGS Harwell routine," Preprint of Dipartimento di Matematica e applicazioni. Univerity of Naples Federico II, IJCM, 2013

[3] Harwell Subroutine Library, Release 10 (1990). Advanced Computing Department, AEA Industrial Technology, Harwell Laboratory, Oxfordshire, United Kingdom.

[4] ANSI/IEEE 754-1985. American National Standard | IEEE Standard for Binary Floating-Point Arithmetic. American National Standards Institute, Inc., New York, 1985.

[5] J.J. Moré and D.J. Thuente, “Line search algorithms with guaranteed sufficient decrease,” ACM Trans. Math. Software, 1994.

[6] D.C. Liu and J. Nocedal, “On the limited memory BFGS method for large scale optimization, Math. Programming,” Volume 45, 1989.

[7] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 1.1, 2007.

[8] NVIDIA. CUBLAS Library. NVIDIA Corporation, 2009. `http://www.nvidia.com/`

[9] NVIDIA. CUDA Visual Profiler, 2009. http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/ docs/cudaprof_1.2_readme.html.