UNIVERSIDADE DA CORUÑA

CUSIMANN: An optimized simulated annealing software for GPUs

AUTHORS:

Ana María Ferreiro Ferreiro

José Antonio García Rodríguez

José Germán López Salas

Carlos Vázquez Cendón

Department of Mathematics, Faculty of Informatics, Campus Elviña s/n, 15071-A Coruña (Spain)

13th April, 2012

CUSIMANN (**CU**DA **SIM**ULATED **ANN**EALING) is a free/open-source library for global optimization that provides a parallel implementation of the simulated annealing algorithm in CUDA.

1 Introduction

In this section we begin giving a general overview of the optimization problems that CUSIMANN solves.

1.1 Optimization problems

CUSIMANN addresses general nonlinear optimization problems of the form $\underset{x \in \mathbb{R}^n}{min}\ f(x)$, where f is the cost function and x represents the n optimization parameters. This problem should be subject to the bound constraints:

$$lb_i \leq x_i \leq ub_i, \quad \text{for } i = 1, \ldots, n$$

given lower bounds lb and upper bounds ub.

1.2 Global optimization

Global optimization is the problem of finding the feasible point x that minimizes the objective $f(x)$ over the entire feasible region. In general, this can be a very difficult problem, becoming exponentially harder as the number n of parameters increases. In fact, unless special information about f is known, it is not even possible to be certain whether one has found the true global optimum, because there might be a sudden dip of f hidden somewhere in the parameter space you haven't looked at yet. However, CUSIMANN is a global optimization algorithm that works well, even if the dimension n is too large.

2 CUSIMANN installation

The installation of CUSIMANN if fairly standard and straightforward, at least on Unix-like systems. It requires a C/C++ compiler, the nvcc[1] CUDA compiler and the NLopt free/open-source library for nonlinear optimization, [6]. To install the compilers and the NLopt library refer to his installation instructions.

In order to use the CUSIMANN library you only have to unpack it and include the header file `cusimann.cuh` in your code.

3 Simulated annealing

The simulated annealing algorithm works with a virtual "temperature", a variable decreasing over time. At each step, the algorithm replaces the current solution by a randomly generated nearby point if this new point results in a better solution, but also allows for "downhill" moves with a certainly probability depending on the temperature, often preventing the method from becoming trapped in a local minimum. This acceptance probability decreases with temperature. There are many ways to implement the algorithm, we are going to explain the one we propose.

[1] Version 4.0 or later, because we use the Thrust CUDA Library.

3.1 Algorithm

The annealing process can be described as follows: starting from some maximum temperature, T_0, we consider a sequence of decreasing temperatures. At each one the system is allowed to reach thermal equilibrium, in which the probability of the system to be in some state with energy E is given by the Boltzmann distribution. The main steps of the simulated annealing algorithm are the following (see [1] for details):

- **Step 1**: Start with a given temperature, T_0, and an initial point, x_0, with energy of configuration $E_0 = f(x_0)$.
- **Step 2**: Select randomly a coordinate of x_0 and a random number to modify the selected coordinate to obtain another point x_1 in a neighborhood of x_0.
- **Step 3**: Compare the function value at the two previous points, by using the Metropolis criterion as follows: let $E_1 = f(x_1)$ and select a sample of a random uniform variable $U(0, 1)$. Then, move the system to the new point if and only if $U < \exp(-(E_1 - E_0)/T)$, where T is the current temperature. Thus, $E_1 - E_0$ is compared with an exponential random variable with mean T. Note that we always move to the new point if $E_1 < E_0$, and that at any temperature there is a chance for the system to move "upwards". Note that we need three uniform random numbers: one to choose the coordinate, one to change the selected coordinate and one for the acceptance criterion.
- **Step 4**: Either the system has moved or not, repeat steps $2 - 3$. At each stage we compare the function at new points with the function at the present point until the sequence of accepted points fulfills some test of achieving an equilibrium state.
- **Step 5**: Once an equilibrium state has been achieved for a given temperature, the temperature is decreased according to the annealing schedule (in our case we update temperature with a decreasing factor ρ with $0 < \rho < 1$, usually ρ close to one) and step 2 starts again, with the last iteration of the algorithm as initial state. The iteration procedure continues until a stopping criterion considering the system has frozen is achieved.

  Since we continue steps $2 - 3$ until an equilibrium state, the starting values in step 1 have no effect on the solution. The algorithm can be implemented in numerous ways.

The pseudo-code of the previous algorithm can be sketched as follows:

```
x = x_0 ; T = T_0;
do
   for j = 1 to N do
      x' = ComputeNeighbour(x);
      ΔE = f(x') − f(x); // Energy increment
      if (ΔE < 0 or AcceptWithProbability P(ΔE, T))
         x = x'; // The trial is accepted
   end for
   T = ρT ; // with 0 < ρ < 1
while (T > T_min);
```

The algorithm is highly dependent on the value of the parameters ρ, N, T_0 and T_{min}. As soon as T_{min} is decreased and T_0, N and ρ are increased, the accuracy and the computational cost increases. This can be allowed by the GPU technology and provides an accurate result.

The annealing algorithm appears to be very reliable, in that it always converged to within a neighborhood of the global minimum. The size of this neighborhood can be reduced by altering the parameters of the algorithm, but this can be expensive in terms of time.

In most cases we would like to find the minimizing solution up to a precision of several decimal places. It is clear that the annealing algorithm could produce results with such accuracy, but sometimes the execution time would be prohibitive. A more traditional algorithm can produce solutions to machine accuracy but can have considerable difficulty in finding the correct solution. This is the reason why we combined the annealing algorithm with a more traditional algorithm to form a hybrid algorithm which is both reliable and accurate.

3.2 Hybrid algorithm

The hybrid algorithm consists of two distinct components:

- The first component is an annealing algorithm which is used to produce a starting point for the second component.
- The second component is a local optimization algorithm. We offer to the user the possibility to use quickly and easily the Nelder-Mead algorithm implemented in the NLopt library. If you wish to use other local minimization algorithm you can implement it as well.

4 Parallelization

The parallelization of the previous algorithm is not straightforward. Figure 1 shows the sketch of the parallel algorithm. We take the following approach: at each temperature level we perform nt Markov chains of length N. These are independent processes and can be distributed among the GPU processors: each Markov process will be executed by one thread in the GPU and thus, each thread returns one final state. We then choose the best state among the nt computed ones, advance to the next temperature level and use the obtained state as starting point for the next nt Markov processes.

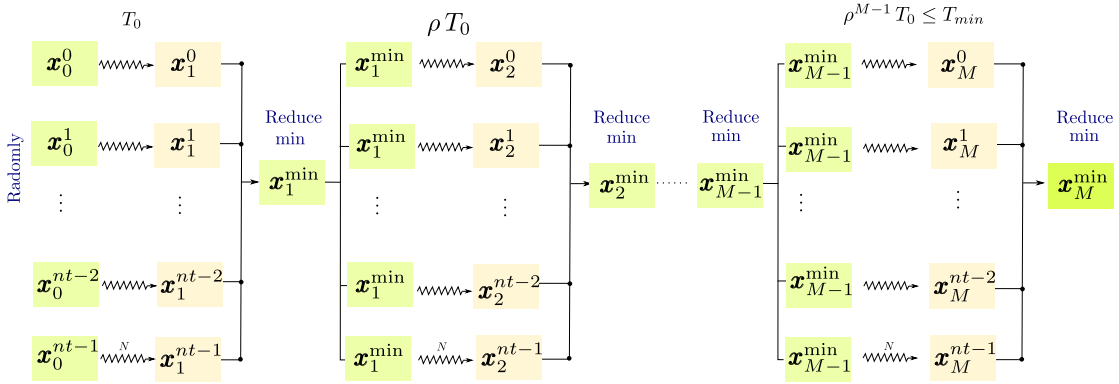

Figure 1: Sketch of the synchronous parallel algorithm.

5 Tutorial

In this tutorial we also illustrate the usage of CUSIMANN via two examples.

5.1 First example

As a first example, we consider the global minimization problem related to the Normalized Schwefel function [4]:

$$f(x) = -\frac{1}{n}\sum_{i=1}^{n} x_i \cdot \sin\left(\sqrt{|x_i|}\right), \quad -512 \leq x_i \leq 512, \qquad x = (x_1, \ldots, x_n).$$

The parameter space of f is constrained to $-512 \leq x_i \leq 512$. For any space dimension n, the global minimum is achieved at $\overline{x}_i = 420.968746$, $i = 1, \ldots, n$, and $f(\overline{x}) = -418.982887$. In order to illustrate the complexity of the function, a plot of f for $n = 2$ is shown in Figure 2.

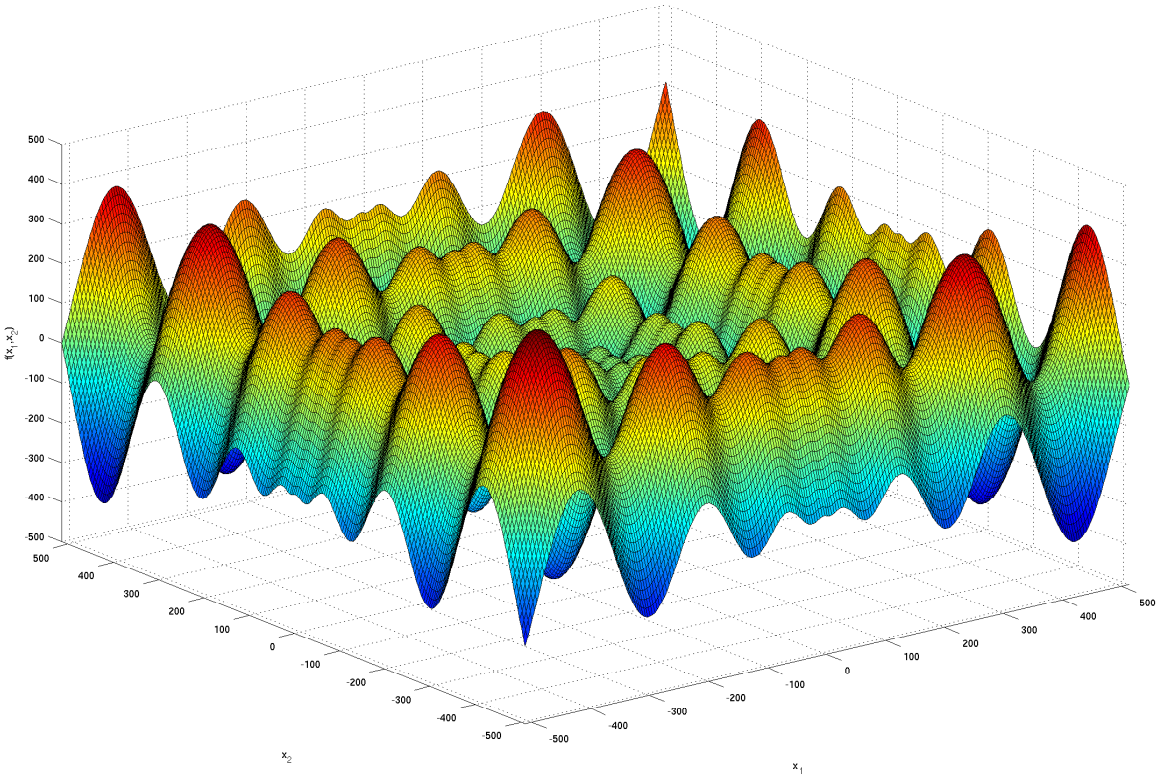

Figure 2: Plot of the Normalized Schwefel function for $n = 2$.

To implement the above example using CUSIMANN we would first do:

```
#include "cusimann.cuh"
```

to include the CUSIMANN header file.

Next we would define our objective function as:

```
template <class T>
class Schwefel {
public:
  __host__ __device__ T operator() (const T *x, unsigned int n, void *f_data) const
  {
    T f_x = 0.0f;

    int i;
    for(i=0;i<n;i++)
      f_x = f_x + x[i] * sin(sqrt(fabs(x[i])));

    f_x = f_x / n;

    return - f_x;
  }
};
```

There are several points to be noticed here. First, since in CUDA it is not allowed to take the address of a `__device__` function in host code, we use *Functor classes* to workaround this restriction. Second, the compiler will generate two functions, one for the device and one for the host; the first one can be generated in single or double precision (as chosen by the user in the configuration file `configuration.h`, although it is recommended the single-precision for reasons of speed calculation in GPUs), and the other one is generated in double precision, for reasons of accuracy. Also, the first argument of the objective function refers to the point in which we evaluate the function and `n` is the dimension of `x`. We have an extra parameter `f_data` that can be used to pass additional data to `f`, but no additional data is needed here so that this parameter is unused (see the next example for more details).

Now we can call `cusimann_optimize` to actually perform the optimization:

```
template<class F>
int cusimann_optimize(unsigned int n_threads_per_block, unsigned int n_blocks, real
    T_0, real T_min, unsigned int N, real rho, unsigned int n, real *lb, real *ub,
    F f, void *f_data, real *cusimann_minimum, real *f_cusimann_minimum)
```

Let's explain each one of the arguments of the minimization function:

- `n_threads_per_block`: represents the number of threads per block to be launched in the minimization kernel of simulated annealing. Reference values are 128, 256 or 512.
- `n_blocks`: similarly, represents the number of thread blocks to be launched in the minimization kernel of simulated annealing. A reference value is 64.
- `T_0`: represents the initial temperature T_0. T_0 must be sufficiently large for any point within the parameter space to have a reasonable chance of being considered as candidate to minimum, but if it is too large then too much time is spent in a "molten" state. In practice it may be necessary to try the algorithm for several values of T_0 before deciding on a suitable value. A reference value can be 1000.
- `T_min`: minimum temperature T_{min}. The algorithm stops when the reached temperature is less than or equal to T_{min}. A small T_{min} gives an accurate solution. Reference values are 0.1, 0.01, ...
- `N`: length of the Markov chain N, i.e., number of neighboring states generated for each temperature level. A large N gives an accurate solution. If possible the value of N should be selected

so that it could be reasonably assumed that the system is in equilibrium by that time. A reference value is 100.

- `rho`: parameter that defines the annealing schedule, ρ, $0 < \rho < 1$. Increasing ρ increases the reliability of the algorithm to reach the global optimum, and corresponds to a slower cooling of the system. A reference value is 0.99.
- `n`: dimension of the search space n.
- `lb`: vector of size `n` containing the lower bounds constraints lb, $lb_i \leq x_i, \forall i = 1 \ldots n$.
- `ub`: vector of size `n` containing the upper bounds constraints ub, $ub_i \geq x_i, \forall i = 1 \ldots n$. This vector, together with the above one, defines the search space. Obviously the search space should be as small as possible.
- `f`: objective function.
- `f_data`: this parameter can be used to pass additional data to the objective function `f`. It avoids the use of the "dangerous" global variables. If no additional data are needed it is set to `NULL`.
- `cusimann_minimum`: All previous parameters are inputs. However, this is an output parameter which returns the minimum found by the simulated annealing algorithm.
- `f_cusimann_minimum`: output parameter which returns the value of the objective function `f` at the obtained minimum.

Once we have performed the simulated annealing, if we want to improve the accuracy of the obtained solution, we can run the Nelder-Mead local minimization algorithm, starting from the point returned by simulated annealing. For this purpose we have to include the header file `nelderMead.h`. Next, we have to define the objective function as required by the NLopt Library (see [6] for more details):

```
double f_nelderMead(unsigned int n, const double *x, double *grad, void *f_data){
  return Schwefel<double>()(x,n,f_data);
}
```

And finally simply call the following function

```
void nelderMead_optimize(unsigned int n, real *lb, real *ub, real *startPoint,
    double (*f)(unsigned int n, const double *x, double *grad, void *f_data), void
    *f_data, double *nelderMead_minimum, double *f_nelderMead_minimum)
```

where the parameters unexplained yet represent:

- `startPoint`: first iterate for the Nelder-Mead algorithm. It must be initialized to the value returned by simulated annealing.
- `f`: function pointer to the Nelder-Mead objective function.
- `nelderMead_minimum`: output parameter which returns the minimum obtained by the Nelder-Mead algorithm.
- `f_nelderMead_minimum`: output parameter which returns the value of the objective function `f` at the obtained minimum.

For more details, the complete example code can be seen in the file `CUSIMANN/samples/minimizeSchwefel.cu`.

It is compiled with the `nvcc` compiler, and it is necessary to link properly the used libraries (cutil, curand, nlopt and math):

```
$ nvcc -I../include minimizeSchwefel.cu -lcutil -lcurand -lnlopt -lm -o minimizeSchwefel
```

Finally, we show the results of an execution with $n = 13$ and the following simulated annealing configuration parameters: $T_0 = 1000, T_{min} = 0.1, N = 100, \rho = 0.99$, `n_threads_per_block = 256`, `n_blocks = 64`:

```
cusimann_minimum = [ 420.967255 420.978210 420.984650 420.999115 420.950195
    420.955261 420.982971 420.990631 420.980896 420.968994 420.990265 420.978882
    420.987488 ]
f(cusimann_minimum) = -418.982910
nelderMead_minimum = [ 420.968746 420.968747 420.968745 420.968746 420.968745
    420.968746 420.968747 420.968747 420.968746 420.968746 420.968746 420.968747
    420.968746 ]
f(nelderMead_minimum) = -418.982887
```

Notice that, in this case, the accuracy of the simulated annealing solution is correct, in general, to only the first decimal place, and the Nelder-Mead ones is much more accurate.

5.2 Second example

As a second example we will look at the global minimization problem for the following Modified Langerman function [5]:

$$f(x) = -\sum_{i=1}^{5} c_i e^{-\frac{1}{\pi}\sum_{j=1}^{n}(x_j - a_{ij})^2} \cos\left(\pi \sum_{j=1}^{n}(x_j - a_{ij})^2\right), \quad -20 \leq x_i \leq 20, \qquad x = (x_1, \ldots, x_n),$$

where

$$A = \begin{pmatrix} 9.681 & 0.667 & 4.783 & 9.095 & 3.517 & 9.325 & 6.544 & 0.211 & 5.122 & 2.020 \\ 9.400 & 2.041 & 3.788 & 7.931 & 2.882 & 2.672 & 3.568 & 1.284 & 7.033 & 7.374 \\ 8.025 & 9.152 & 5.114 & 7.621 & 4.564 & 4.711 & 2.996 & 6.126 & 0.734 & 4.982 \\ 2.196 & 0.415 & 5.649 & 6.979 & 9.510 & 9.166 & 6.304 & 6.054 & 9.377 & 1.426 \\ 8.074 & 8.777 & 3.467 & 1.863 & 6.708 & 6.349 & 4.534 & 0.276 & 7.633 & 1.567 \end{pmatrix},$$

$$c = \begin{pmatrix} 0.806 & 0.517 & 0.100 & 0.908 & 0.965 \end{pmatrix}.$$

The global optimum in the five dimensional case ($n = 5$) is $\overline{x}_1 = 8.074000, \overline{x}_2 = 8.777001, \overline{x}_3 = 3.467004, \overline{x}_4 = 1.863013, \overline{x}_5 = 6.707995$, and $f(\overline{x}) = -0.964999$.

First of all we must define a data type to pass the constant matrix A and the vector c to the objective function:

```
#include "cusimann.cuh"
#include "nelderMead.h"

typedef struct {
  real *A, *c;
} LANGERMAN_data;
```

Next we define our objective function as:

```
template <class T>
class Langerman {
public:
  __host__ __device__ T operator() (const T *x, unsigned int n, void *f_data) const
  {
    LANGERMAN_data langerman_data;
    langerman_data = *((LANGERMAN_data*)f_data);

    real *A = langerman_data.A;
    real *c = langerman_data.c;

    T f_x = 0.0f;
    T aux;

    int i, j;
    for(i=0;i<5;i++) {

      aux = 0.0f;
      for(j=0;j<n;j++)
        aux += pow(x[j]-A[pos2Dto1D(i,j,10)],2) ;

      f_x +=  c[i] * exp(-1.0f/M_PI_CUDA * aux) * cos(M_PI_CUDA * aux);
    }

    return - f_x;
  }
};

double f_nelderMead(unsigned int n, const double *x, double *grad, void *f_data){
  return Langerman<double>()(x,n,f_data);
}
```

Now, before calling `cusimann_optimize`, we build up the device structure containing `A` and `c` in host memory first, and then copy that to the device. This can be done as follows:

```
  real A[5*10] = {
        9.681, 0.667, 4.783, 9.095, 3.517, 9.325, 6.544, 0.211, 5.122, 2.020,
        9.400, 2.041, 3.788, 7.931, 2.882, 2.672, 3.568, 1.284, 7.033, 7.374,
        8.025, 9.152, 5.114, 7.621, 4.564, 4.711, 2.996, 6.126, 0.734, 4.982,
        2.196, 0.415, 5.649, 6.979, 9.510, 9.166, 6.304, 6.054, 9.377, 1.426,
        8.074, 8.777, 3.467, 1.863, 6.708, 6.349, 4.534, 0.276, 7.633, 1.567 };
  real c[5] = { 0.806, 0.517, 0.100, 0.908, 0.965 };

  // struct in host memory
  LANGERMAN_data langerman_data;
  langerman_data.A = A;
  langerman_data.c = c;

  // Assemble the device structure in host memory first
  LANGERMAN_data *d_Array_langerman_data, d_langerman_data[1];

  real *d_A, *d_c;
  cutilSafeCall( cudaMalloc((void**)&d_A, 5*10 * sizeof(real)) );
  cutilSafeCall( cudaMemcpy(d_A, langerman_data.A, 5*10 * sizeof(real),
      cudaMemcpyHostToDevice ) );

  cutilSafeCall( cudaMalloc((void**)&d_c, 5 * sizeof(real)) );
```

```
  cutilSafeCall( cudaMemcpy(d_c, langerman_data.c, 5 * sizeof(real),
      cudaMemcpyHostToDevice ) );

  d_langerman_data[0].A = d_A;
  d_langerman_data[0].c = d_c;

  // Then copy that host memory version to device memory
  cutilSafeCall( cudaMalloc ( (void**) &d_Array_langerman_data, 1*sizeof(
      LANGERMAN_data) ) );
  cutilSafeCall( cudaMemcpy(d_Array_langerman_data, d_langerman_data, 1*sizeof(
      LANGERMAN_data), cudaMemcpyHostToDevice ) );
```

Now, in the same way as in the previous example, we call the `cusimann_optimize` function:

```
real lb[5] = {-10, -10, -10, -10, -10};
real ub[5] = {10, 10, 10, 10, 10};
real cusimann_minimum[5], f_cusimann_minimum;
cusimann_optimize(512, 64, 1000, 0.1, 400, 0.99, 5, lb, ub, Langerman<real>(),
    d_Array_langerman_data, cusimann_minimum, &f_cusimann_minimum);
```

Again, finally, we can invoke the Nelder-Mead algorithm:

```
  double nelderMead_minimum[5], f_nelderMead_minimum;
  nelderMead_optimize(5, lb, ub, cusimann_minimum, f_nelderMead, &langerman_data,
      nelderMead_minimum, &f_nelderMead_minimum);
```

For more details, the complete sample code can be seen in the file `CUSIMANN/samples/minimizeLangerman.cu`.

5.3 Using all available GPUs in the system

By default CUSIMANN uses only a single GPU in the minimization process. If you want to use all available GPUs in the system you must indicate it explicitly in the configuration file `configuration.h`. This is useful when trying to minimize complex functions. So the algorithm can explore more points in the function domain "without increasing" runtimes[2].

If the user needs to pass additional data to the objective function (as in the single-GPU Second example 5.2) it is his responsability to properly initialize the memory of each one of the GPUs.

Finally note that CUSIMANN uses OpenMP [7] threads to manage multiple GPUs.

6 Contact information

- Ana María Ferreiro Ferreiro (*aferreiro@udc.es*)
- José Antonio García Rodríguez (*jagrodriguez@udc.es*)
- José Germán López Salas (*jose.lsalas@udc.es*)
- Carlos Vázquez Cendón (*carlosv@udc.es*)

[2] If the user appropriately reduces the number of launched threads in each one of the GPUs, the multi-GPU version can also be used to reduce runtimes.

Department of Mathematics, Faculty of Informatics, Campus Elviña s/n, 15071-A Coruña (Spain).

This work is partially supported by I-Math Consolider Project (Reference: COMP-C6-0393), by MICINN (MTM2010-21135-C02-01) and by Xunta de Galicia (Ayuda CN2011/004 cofinanced with FEDER funds). The authors also acknowledge some ideas suggested by J.L. Fernández (Autonomous University of Madrid).

References

[1] S.P. Brooks, B.J.T. Morgan, *Optimization using simulated annealing*, The Statistician 44 (1995), 241-257.

[2] P.J.M van Laarhoven, E.H.L. Aarts, *Simulated Annealing: Theory and Applications*, Kluwer Academic Publishers (1987).

[3] *NVIDIA CUDA C Programming Guide Version 4.0*, Nvidia Cooporation (2011).

[4] `http://www.it.lut.fi/ip/evo/functions/node10.html`

[5] `http://www.it.lut.fi/ip/evo/functions/node15.html`

[6] `http://ab-initio.mit.edu/wiki/index.php/NLopt`

[7] `http://openmp.org`