PARALLEL CONSTRUCTION FOR CONTINUOUS GRASP OPTIMIZATION ON GPUs

Lisieux Marie Marinho dos Santos Andrade
Centro de Informática – Universidade Federal da Paraíba
Campus I, Cidade Universitária – 58059-900, João Pessoa – Paraíba
lisieuxandrade@gmail.com

Raphael Bezerra Xavier
Centro de Informática – Universidade Federal da Paraíba
Campus I, Cidade Universitária – 58059-900, João Pessoa – Paraíba
rbxavier18@gmail.com

Lucídio dos Anjos Formiga Cabral
Centro de Informática – Universidade Federal da Paraíba
Campus I, Cidade Universitária – 58059-900, João Pessoa – Paraíba
lucidio@di.ufpb.br

Andrei de Araújo Formiga
Centro de Informática – Universidade Federal da Paraíba
Campus I, Cidade Universitária – 58059-900, João Pessoa – Paraíba
andrei@ci.ufpb.br

ABSTRACT

Optimization methods, of which Continuous Greedy Adaptive Random Search Procedure (CGRASP) is an example, are useful in a great number of applications. As the problems in need of optimization techniques grow in size and dimensionality, it's increasingly necessary to harness the additional computing power provided by parallel hardware like the Graphical Processing Units in graphics cards. GPUs have hundreds of processing cores but their programming model is markedly different from sequential programming models, and require care in designing parallel versions of sequential algorithms in order to maximally utilize the additional computing power. In this paper we propose a technique that implements the construction step of CGRASP in parallel for GPUs, increasing the efficiency of the optimization process. This is a first, important step on the way to a fullyparallel version of CGRASP that is able to efficiently find the optimum in a greater number of applications.

KEYWORDS. Continuous Optimization. Continuous Greedy Adaptive Random Search Procedure. Graphical Processing Units.

Main area. OC - Combinatorial Optimization. MH Metaheuristics.

1. INTRODUCTION

Optimization techniques are widely used in a great number of applications across many areas. Optimization is necessary whenever there is a need to either maximize a measure – efficiency, for example – or minimize one – like cost. This describes a plethora of situations in applications in industrial and academic settings including automatic control, finance, data mining, logistics planning and portfolio selection (Andreasson et al, 2007).

Optimization problems, and the techniques used to solve them, are roughly divided into two main classes: discrete optimization and continuous optimization, according to the type of space over which the function to be optimized – called the *objective function* – is defined. Another way to classify optimization procedures is according to whether they guarantee that a global optimum is found even when the objective function is nonlinear and non-convex (*global optimization*) or not (*local optimization*). Given an objective function $f(x) : X \rightarrow \mathbb{R}$, the goal in global optimization is to find a specific value x^* (which may be a vector) for the parameter of the function such that $f(x^*) \leq f(x)$ for all $x \in X$. In this case x^* is called the *global minimum*, and the previous inequality would be reversed for finding the global maximum. The search for x^* may also be restricted by a set of desired constraints.

One important difficulty in solving global optimization problems is the choice of an appropriate technique considering the nature and size of the problem. Exact methods are often inefficient for most non-trivial problems, and there is no guarantee of fast convergence for finding the best solution. There are also methods that use the derivative of the objective function as an indication of the direction to continue searching for a solution, and this may be an issue for problems for which calculating the derivative of f is computationally inefficient because the function is not differentiable or not continuous.

In order to alleviate some of these problems a set of techniques commonly classified as *metaheuristics* were developed, initially for discrete optimization. Metaheuristics make few assumptions about the optimization problems being solved, and thus are widely applicable to many different problems. Greedy Randomized Adaptive Search Procedure (GRASP) (Feo and Resende, 1995) is a discrete optimization technique in the class of metaheuristics, and Continuous GRASP (CGRASP) (Hirsch et al, 2007), is a derived technique for continuous optimization. Metaheuristics like CGRASP do not guarantee, in all cases, finding the globally best solution, but they show good convergence properties for finding solutions near the global optimum.

While CGRASP does not need to use the derivative of the objective function, it often needs to compute the value of the objective function for many points in the neighborhood of a candidate solution, a process which is repeated many times over when the algorithm is being used to find a solution. This may make the method computationally inefficient to use, especially in problem spaces of high dimensionality, where there may be a large number of points in the neighborhood of a candidate solution. Parallel computation may be employed to alleviate this problem, especially with the possibility of using accesible parallel computing hardware like found on Graphics Processing Units (GPUs). A parallel version of the CGRASP algorithm will be able to solve bigger problems in a smaller time than it is possible on sequential architectures or multicore architectures, which tipically have few CPUs.

In this paper, we propose a parallel algorithm for a crucial step in the implementation of the Continuous Greedy Randomized Adaptive Search Procedure (CGRASP) (Hirsch et al, 2007), increasing the efficiency of the method by the use of highly-available and low-cost hardware of Graphical Processing Units (GPUs). The implementation uses the CUDA (Compute Unified Device Architecture) standard from nVidia (CUDA), but it can be easily adapted to other parallel computing standards like OpenCL. The algorithm with a parallel computing step is considerably faster than the purely sequential algorithm.

This paper is organized as follows. Section 2 presents the original CGRASP algorithm (Hirsch et al, 2007), and Section 3 describes the ideas behind the use of Graphical Processing Units (GPUs) for performing general computation on a massively parallel architecture. Section 4

describes the parallel construction algorithm used to speed-up CGRASP, and Section 5 shows how the parallel construction version of the algorithm performs in an experimental setting. Section 6 presents the main conclusions and results of the research reported herein.

2. CONTINUOUS GRASP (CGRASP)

Continuous Greedy Randomized Adaptative Search Procedure (CGRASP) (Hirsch et al, 2007), is a continuous version of the GRASP algorithm (Feo and Resende, 1995), which is a discrete optimization algorithm. CGRASP shares many properties with GRASP, for example both are multistart procedures and both carry out a local search from a greedy and randomly constructed initial solution. One important difference is that CGRASP repeats the construction and search phases many times, whereas GRASP executes both phases only once.

```
procedure CGRASP (n, l, u, f(.), h_s, h_e, ρ_lo)
1.   f* ← ∞ ;
2.   while stop criterion is not met do
3.        x ← random choice in [l..u];
4.        h ← h_s;
5.        while h ≥ h_s do
6.             Impr_c ← false;
7.             Impr_L ← false;
8.             x,Impr_c ←Construction (x, f(.), n, h, l, u, Impr_c);
9.             x,Impr_L ←LocalSearch(x, f(.), n, h, l, u, ρ_lo, Impr_L);
10.            if f(x) < f* then
11.                 x ← x*;
12.                 f* ← f(x);
13.            end-if
14.            if Impr_c = false and Impr_L = false then
15.                 h ← h/2; /*makes the grid denser*/
16.            end-if
17.       end-while
18.  end-while
19.return x*;
end CGRASP.
```

Figure 1. CGRASP pseudocode.

The procedure's arguments are: the dimension of the problem (n); the vectors l and u, which define the lower and upper bounds; the objective function $f(\ )$; the initial and final densities of the discretization grid, h_s and h_e, respectively; and, finally, ρlo, which defines the neighborhood area visited in a local search from the current solution (Hirsch et al, 2010).

We can see in the algorithm depicted in Figure 1 that in each iteration (lines 3 to 17), while the stop criterion is not met, a solution is defined in a random way, using an uniform distribution over the grid, whose lower and upper bounds are determined by l and u. Possible acceptance criteria are the total number of iterations and the number of objective function evaluations.

Analyzing the algorithm behavior, the variable that controls the initial grid density (h) is reset with the parameter value (h_s), and while $h \geq h_s$ the algorithm will execute the construction and local search phases sequentially. When there is improvement, ($f(x) < f^*$), the solution is updated as the current solution; otherwise ($Impr_c$ = false and $Impr_L$ = false), the grid density is made smaller to improve the search for a solution (Hirsch et al, 2007).

2.1 Construction Phase

CGRASP has two phases: the construction phase and the local search phase. The construction phase generates an initial solution in a random way, which provides diversification. The constructed solution, x, has a set of unfixed coordinates, whose values can vary (Hirsch et al, 2010).

```
procedure CGRASP-Construction(x, f(.), n, h, l, u, Impr_c)
1.  nFixa ← [1, n]
2.  α ← k, k ∈ [0, 1];
3.  ReUse ← false;
4.  while nFixa ≠ ∅ do
5.      g ← +∞;
6.      ḡ ← −∞;
7.      for i = 1 ... n do
8.          if i ∈ nFixa then
9.              if ReUse = false then
10.                 z_i ← LinearSearch(x, h, i, n, f(.), l, u);
11.                 g_i ← f(z_i);
12.             end-if;
13.             if g > g_i then g ← g_i;
14.             if ḡ < g_i then ḡ ← g_i;
15.         end-if
16.     end-for
17.     LCR ← ∅;
18.     threshold ← g + α(ḡ − g);
19.     for i = 1 ... n do
20.         if i ∈ nFixa and g_i ≤ threshold then
21.             LCR ← LCR ∪ i;
22.         end-if
23.     end-for
24.     j ← random choice in LCR;
25.     if x_j = z_j then ReUse ← true;
26.     else
27.         x_j ← z_j;
28.         ReUse ← false;
29.         Impr_c ← true;
30.     end-else
31.     nFixa ← nFixa \ {j};
32. end-while
33. return(x, Impr_c);
end CGRASP-Construction.
```

Figure 2. CGRASP construction phase.

After the set of unfixed coordinates (*nFixa*) is initialized, a line search procedure is called to search for a minimum in every non-fixed coordinate *i*, each time keeping the other *n-1* coordinates in their current values. After the line search, the value of the minimum found for coordinate *i* is stored in z_i and the corresponding objective function value is stored in g_i. The values of g_i are used to form the LCR, which contains the values of non-fixed coordinates with the best values of g_i. Values contained in the LCR are less than or equal to the threshold calculated in line 18 as $g + \alpha(\overline{g} - g)$, where g and $\overline{g}$ represent the maximum and minimum values of g_i, respectively. With the LCR created, one of the coordinates in it is chosen randomly to be removed from it and then fixed. Choosing a coordinate in this way ensures randomness in the construction phase (Hirsch et al, 2007).

2.2 Local search Phase

After the conclusion of the construction phase, the next step is the local search, which, according to (Hirsch et al, 2010), can be thought as an approximation function between the gradient function and the objective function. Given a solution *x*, the local search procedure inspects the neighborhood of *x* in the quest for a new solution *y*, ideally better than *x*. This new solution *y* is the initial solution in the next iteration of the local search.

procedure CGRASP-LocalSearch(x, $f(.)$, n, h, l, u, ρ_{lo}, $Impr_L$)
1. $x^* \leftarrow x$
2. $f^* \leftarrow f(x)$;
3. $GridPoints \leftarrow \prod_{i=1}^{n} \lceil (u_i - l_i/h \rceil$;
4. $MaxPoints \leftarrow \lceil \rho lo \cdot GridPoints \rceil$;
5. $PointsExamined \leftarrow 0$;
6. **while** $PointsExamined \leq MaxPoints$ **do**
7. $PointsExamined \leftarrow PointsExamined + 1$;
8. $x \leftarrow RandomlySelectElement(B_h(x^*))$;
9. **if** $l \leq x \leq u$ **and** $f(x) < f^*$ **then**
10. $x^* \leftarrow x$;
11. $f^* \leftarrow f(x)$;
12. $Impr_L \leftarrow true$;
13. $PointsExamined \leftarrow 0$;
14. **end-if**
15. **end-while**
16. **return**(x, $Impr_L$);

end CGRASP-LocalSearch.

Figure 3. CGRASP local search phase.

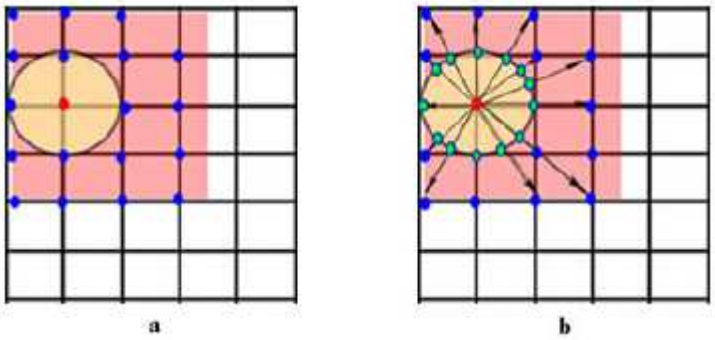
In this scenario, we can define the neighbour of the CGRASP algorithm by an actual solution $\overline{x} \in \mathbb{R}_n$ (represented in red in the Figure 4a), discretized throug *h*, where: $\mathbb{Z}$

$$S_h = \{x \in S \mid l \leq x \leq u, \overline{x} + r \cdot h, r \in \mathbb{Z}^n$$

is the points set in *S*, which are measures of size *h* and from $\overline{x}$ (represented in blue in the Figure 4b):

$$B_h = \left\{ x \in S \mid x = \overline{x} + h \cdot \frac{\left(x' - \overline{x}\right)}{\left\| x - \overline{x} \right\|}, x' \in S_h(\overline{x}) \setminus \{\overline{x}\} \right\},$$

corresponds to the projection of the points $S_h(\overline{x}) \setminus \{\overline{x}\}$, inside the hiper-sphere centered in $\overline{x}$ of radius *h*. In other words, the *h-th* neighbour of $\overline{x}$ is defined by the points contained $B_h(\overline{x})$ (Figure 4b).



a b

Figure 4. Representation of neighborhood CGRASP, (a) $S_h(\overline{x})\setminus\{\overline{x}\}$. (b) Representation of the set $B_h(\overline{x})$.

In the lines 6 to 15, the algorithm select *MaxPoints* points randomly from $B_h(x^*)$,if the actual solution x is selected from $B_h(x^*)$ is feasible, so x will be defined by x ($x^* \leftarrow x$),and $Impr_L$ is set to true. $Impr_L$, as $Impr_c$ in the construction phase, hold the feature to signal whether the procedure improved or not the quality of the generated solution.

3. GENERAL COMPUTING ON GPUS

The current Graphical Processing Units (GPUs) evolved from graphics hardware intended for acceleration of 3D rendering in personal computers, mostly used for real-time 3D rendering in games (Kirk and Hwu, 2010). With time, more parts the rendering pipeline were implemented in the graphics processors, to the point where such processors became quite capable as computing devices. Eventually, the manufacturers of graphics hardware developed programmable graphics processors that could be programmed by users, via software (using the so-called *shaders* (Engel, 2004)). This led some researchers to the idea of using GPUs as general computing devices, by adapting the graphics shaders to non-intended uses (Fung and Mann, 2004). Manufacturers then established standards and tools created to make it easier for programmers to use their hardware as general computing devices: NVIDIA created CUDA (Kirk and Hwu, 2010), while AMD and other companies developed OpenCL (Aaftab Munshi et al, 2011), as an open standard.

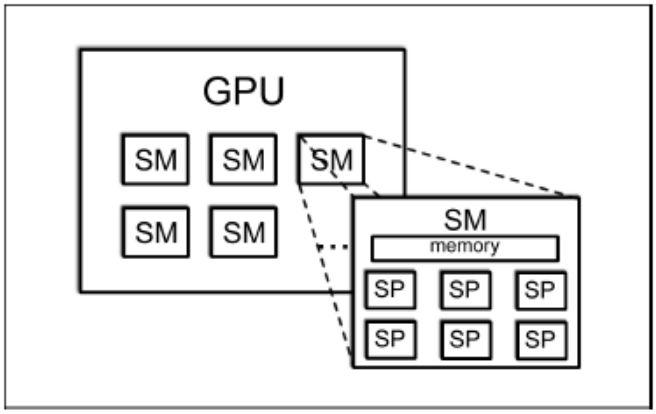


Figure 5. Overall architecture of a typical GPU. The GPU contains multiple Streaming Multiprocessors (SMs), each of which is composed of multiple Streaming Processors (SPs) or cores.

The evolution of GPU hardware from graphics accelerators resulted in an architecture that is quite distinct from the processors used as CPUs in current computers. GPUs have many simple processing cores – totaling hundreds of cores in current hardware – instead of the few complex cores in current CPUs. GPUs are massively parallel processors that are able to execute large parallel tasks much more efficiently than CPUs. However, each group of cores execute the same instructions at the same time, making GPUs a poor fit for applications that do not present data parallelism (Farber, 2011).

Even considering the differing standards from manufacturers, the architectures of GPUs have converged to similar lines. GPUs are composed of a number of Streaming Multiprocessors (SMs). Each SM encases many processing cores or Streaming Processors (SPs), typically 32 in current hardware (Farber, 2011). SMs are independent units of execution, dispatching threads of execution to their SPs, one thread per core. The general architectural organization of GPUs is displayed in Figure 5. In most cases, the SMs can access only memory that is located in the GPU. This means that any data that must be processed in the GPU must be transferred from the system

main memory to the GPU memory.

Each process that is to be executed on the GPU is called a computational kernel, or simply a kernel. In CUDA C, the version of the C language for programming the CUDA standard from NVIDIA, each kernel is called as a function. All cores in a Streaming Multiprocessor will execute the same instructions, in lockstep, although the instructions can access different memory locations. Each SM is thus a SIMD (Single Instruction, Multiple Data) computing machine, although as the GPU contains multiple SMs, the execution model is more appropriately called Single Instruction, Multiple Threads (SIMT) (Farber, 2011). Conditional instructions (for example in *if-then-else* constructs) which force threads in the same SM to different paths of execution cause *thread divergence*. In practice, thread divergence is implemented by making some cores in a SM idle while others execute the conditional instruction, reducing overall throughput.

The architectural organization of GPUs makes it clear that efficient execution on graphical hardware requires programs which are organized very distinctly from programs designed for CPU execution. Two general principles for efficient execution on GPUs are to keep the processors busy and to avoid thread divergence. The implementation techniques for the construction phase of CGRASP on GPUs, described in the following section, were designed to adhere to these two general principles.

4. PARALLEL CONSTRUCTION ON GPUS

The construction step of CGRASP (shown in Figure 2) takes little time when the objective function is of low dimensionality. However, the construction becomes increasingly significant (in terms of time efficiency) as the dimensionality of the objective function increases. In an experiment with the sequential implementation, the construction phase when optimizing a two-dimensional objective function amounts to less than 10% of the total time spent in the CGRASP optimization algorithm, while for a 30-dimensional function the construction phase makes more than 40% of the total time of the algorithm. This indicates that the construction phase is a critical part of the algorithm to improve when trying to solve bigger problems.

Graphical Processing Units (GPUs) present a low-cost hardware solution to increase the computational power available to solve problems. Therefore, a natural idea is to parallelize optimization methods like CGRASP to take advantage of the greater number of processing units available in GPUs. However, very few algorithms can be directly translated to execute on GPUs without change, and great care is often necessary to get any efficiency gain when using this kind of parallel hardware.

A natural target for parallelization of the CGRASP algorithm is the construction phase, because of its increasing footprint in the total time as the dimensionality increases. However, to achieve good efficiency on GPUs, it is necessary to structure the algorithm and data structures to better fit the GPU architecture (as seen in Section 3).

The costly part in the standard CGRASP construction phase (shown in Figure 2) is the loop between lines 7 and 16, where a linear search is performed in each direction that is not yet fixed. The parallel algorithm could be just a parallel version of this loop, but the management of the list of unfixed dimensions from CPU to GPU would create too much overhead. Thus most of the construction process is done on the GPU. The parallel construction program, shown in Figure 6, first copies the important vectors from CPU to GPU memory, then initializes the list of unfixed dimensions on GPU memory (calling an initialization kernel) and then finally calls a GPU kernel with n threads to do the construction. At the end, the resulting vector x is copied back to CPU memory to return to the rest of the CGRASP process. Note that the bounds vectors l and u do not change during the CGRASP optimization procedure, so it can be copied only once to the GPU memory instead of every time as in line 1 of Figure 6. This is actually how it is implemented, but the high-level algorithm is simpler like described in Figure 6.

```
procedure CGRASP-Par-Const(x, f(.), n, h, l, u, Impr_c)
1.  Copy x, l and u to GPU memory
2.  Init unfixed on GPU
3.  ConstructionKernel<<<n>>>(x, f(.), n, h, l, u);
4.  Copy resulting x from GPU memory to main memory
5.  return(x, Impr_c);
end CGRASP-Par-Const.
```

Figure 6. Parallel construction algorithm.

The construction kernel called by the parallel construction algorithm is shown in Figure 7. The kernel manages the unfixed list and calls the linear search kernel, one for each thread, using the thread index as a parameter. In implementation terms, LinearSearchKernel is a device kernel, to be called by global kernels like ConstructionKernel. The linear search kernel verifies the unfixed list and does a linear search for a minimum in each dimension that is not fixed. As an implementation detail, it is important to note that the unfixed list is stored as a vector of booleans on the GPU.

```
procedure ConstructionKernel<<<n>>>(x, f(.), n, h, l, u)
3.  ReUse ← false;
4.  while nFixa ≠ ∅ do
5.      g ← +∞;
6.      ḡ ← −∞;
7.      LinearSearchKernel(threadIdx, x, h, i, n, f(.), l, u, nFixa);
8.      LCR ← ∅;
9.      threshold ← g + α(ḡ − g);
10.     for i = 1 ... n do
11.         if i ∈ nFixa and g_i ≤ threshold then
12.             LCR ← LCR ∪ i;
13.         end-if
14.     end-for
15.     j ← random choice in LCR;
16.     if x_j = z_j then ReUse ← true;
17.     else
18.         x_j ← z_j;
19.         ReUse ← false;
20.         Impr_c ← true;
21.     end-else
22.     nFixa ← nFixa \ {j};
23. end-while
24. return(x, Impr_c);
end CGRASP-Par-Const.
```

Figure 7. The parallel construction kernel.

5. EXPERIMENTS AND RESULTS

The CGRASP algorithm with the parallel construction procedure shown in Figures 6 was implemented using the CUDA C language (Kirk and Hwu, 2010). We used this implementation on a set of experiments to determine the performance of the CGRASP algorithm with parallel construction. The experiments consisted on searching the optimum on a set of known objective functions of various types and dimensionalities. The version with parallel construction was compared against a fully sequential version written in C++. As this was a first experimental assessment, both versions were executed on a machine with an Intel Core i3 CPU, 4Gb of RAM and a GeForce 630M GPU with 2Gb of dedicated graphics memory. Further experiments are planned to be performed on a more recent and capable GPU, for a high-performance test.

The first phase of experimental testing was performed with objective functions of low dimensionality (less than 20 dimensions). In this case it became clear that the version with parallel construction did not achieve significant speedup over the sequential versions. When measuring the impact of the construction phase on the CGRASP algorithm for functions of low

dimensionality, we verified that the construction phase can take as little as 10% of the total execution time for the algorithm. Improving the construction phase on these kinds of problems is thus ineffective.

However, when profiling the CGRASP algorithm for objective functions of more than 20 dimensions, we verified that the construction phase could take as much as 30% or even 40% of the total execution time. This lead to the second phase of experimental testing where we used the CGRASP algorithm with parallel construction on these high dimension objective functions, achieving a significant speedup. The results are shown in Table I, which displays median times for ten executions of the algorithm on each objective function. The same results are shown graphically in Figure 8. Note that for both versions the number of objective function evaluations did not differ by much, and as such this measure is not interesting in this comparison.

Function	*Dimensions*	*CGRASP*	*CGRASP+PC*	*Speedup*
Griewank	20	12.076s	9.254s	1.3049
Rastrigin	20	9.740s	7.439s	1.3093
Powell	24	10.312s	7.524s	1.3726
Dixon-price	25	7.170s	5.181s	1.3832
Sphere	30	8.212s	5.630s	1.4582
Levy	30	13.144s	8.978s	1.4636
Piccioni	30	10.271s	8.039s	1.2776
F25aF28	40	18.722s	11.996s	1.5607

Table I. Time performance for finding optimums of objective functions using the original version of CGRASP and a version with parallel construction (CGRASP+PC).

These preliminary results clearly indicate improvement when using the version of the algorithm with parallel construction. Another trend we can observe in the data is that the speedup increases with the dimensionality of the objective function. Time performance of CGRASP does not depend only on the dimensionality of the function, because some functions are more difficult to optimize than others (the function sphere, for instance, is just a hyper-sphere in 30 dimensions, which is quite easy to optimize). However, when looking to the speedups obtained we see that functions in higher dimensions are sped up more by the parallel version, up to a maximum of 56% for the function with highest dimensionality (F25aF28). It is also important to note that the dimensionality of the test functions shown in Table I is still quite low for many application areas – like computational chemistry and machine learning – where functions in hundreds of dimensions are not unusual. Thus we can expect the CGRASP+PC algorithm to perform even better on bigger problems, which is a desirable property.

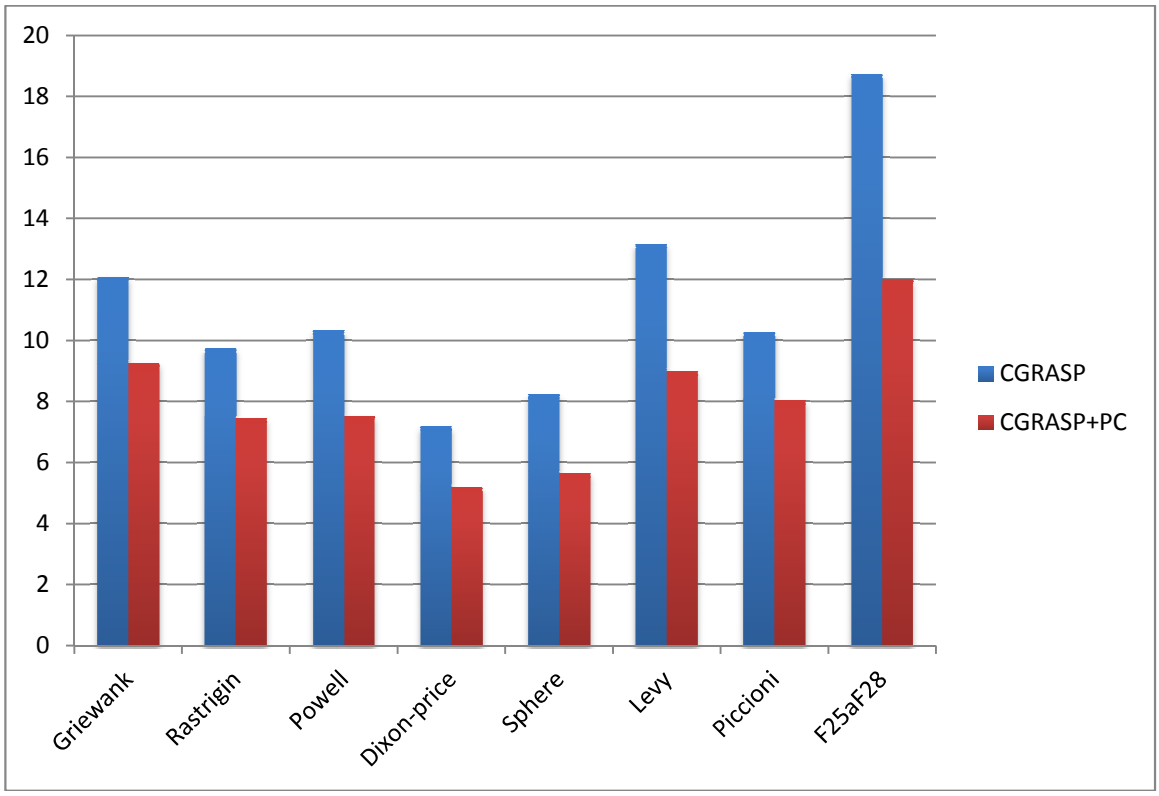


Figure 8. Median times for optimization of test functions using the original CGRASP and GRASP with parallel construction (CGRASP+GP).

6. CONCLUSION AND FUTURE WORK

In this paper we described a version of the CGRASP continuous optimization algorithm that performs the construction of candidate solutions in parallel using graphics hardware (GPUs). Although the construction phase of the algorithm is not a prime target for parallelization when objective functions of low dimensionality are optimized by the algorithm, the same phase becomes increasingly important as the dimensionality of the objective function increases. As the problems to which optimization is applied increase in size and dimensionality, it's desirable to have parallel versions of the optimization methods to cope with the bigger problems.

The construction phase was targeted as a first point for parallelization due to its increased importance as the dimensionality of the objective function increases, as mentioned. Our experiments have shown a promising speedup when construction is performed in parallel for functions of dimensionality between 20 and 30, though these are still a small number of dimensions considering problems in application areas like machine learning, for example.

As future work we intend to apply the version of CGRASP with parallel construction to bigger problems, to run more extensive experiments on bigger hardware, and to parallelize other parts of the CGRASP algorithm to achieve a parallel metaheuristic optimization procedure that fully utilize the computational power of GPUs.

References

Aaftab Munshi, T. G. M., Gaster, B. and Fung, J., OpenCL Programming Guide. Addison-Wesley, 2011.

Andreasson, N., Evgrafov, A. and Patriksson, M., An Introduction to Continuous Optimization. Studentlitteratur AB, 2007.

Engel, W., Programming Vertex & Pixel Shaders. Charles River Media, 2004.

Farber, R., CUDA Application Design and Development. Morgan Kaufmann, 2011.

Feo, T. A. and Resende, M. G. C., Greedy randomized adaptive search procedures, Journal of Global Optimization, vol. 6, pp. 109–133, 1995.

Fung, J. and Mann S., Using multiple graphics cards as a general purpose parallel computer: Applications to computer vision, in Proceedings of the 17th International Conference on Pattern Recognition (ICPR2004). IEEE Press, 2004.

Hirsch, M. J., Meneses, C. N., Pardalos, P. M., and Resende, M. G. C., Global optimization by continuous grasp, Optimization Letters, vol. 1, pp. 201–212, 2007.

Hirsch,M. J., Pardalos, P. M., and Resende, M. G. C., Speeding up continuous grasp, European Journal of Operational Research, vol. 205, pp. 507–521, 2010.

Kirk, D. B. and Hwu, W.-M. W., Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.

Appendix A.
Function Definitions

This appendix list all the functions used in the experiments described in this paper.

Dixon-price (n = 25)

Definition: $f_x = (x_1 - 1)^2 + \sum_{i=2}^{n} i(2x_i^2 - x_{i-1})^2$

Domain: $[-10;\ 10]^n$

Global Minimum: $x^* = 2^{-\frac{2^i-2}{2^i}}, \forall i = 1, \dots, n;\ f_{x*} = 0$

F25aF28

Definition: $f_x = \left[\sum_{i=1}^{n} \frac{x_i^2}{2^{i-1}}\right] + \left[\sum_{i=1}^{n} \frac{x_i\ x_{i-1}}{2^i}\right]$

Domain: $[-20;\ 7]^n$

Global Minimum: $x^* = (0;\dots;0);\ f_{x*} = 0$

Griewank (n = 20)

Definition: $f_x = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}} + 1\right)$

Domain: $[-300;\ 600]^n$

Global Minimum: $x^* = (0;\dots;0);\ f_{x*} = 0$

Levy (n = 30)

Definition: $f_x = sin^2(\pi x_1) + \sum_{i=1}^{n-1}\left[(x_i - 1)^2\left(1 + 10\ sin^2(\pi x_{i+1})\right)\right] + (x_n - 1)^2\ (10\ sin^2(\pi x_n))$

Domain: $[-10;\ 10]^n$

Global Minimum: - 11, 50236

Piccioni (n = 30)

Definition: $f_x = 10\ sin^2(\pi x_1) - \sum_{i=1}^{n-1}[(x_i - 1)^2 + \left(1 + 10\ sin^2(\pi x_1 + 1)\right)] - (x_n - 1)^2$

Domain: $[-10;\ 10]^n$

Global Minimum: $x^* = (0;\dots;0);\ f_{x*} = 0$

Powell (n = 20)

Definition: $f_x = \sum_{i=1}^{n/4}[(x_{4i-3} + 10x_{4i-2})^2 + 5(x_{4i-1} + x_{4i})^2 + (x_{4i-2} + 2x_{4i-1})^4 + 10(x_{4i-3} - x_{4i})^4]$

Domain: $[-4;\ 5]^n$
Global Minimum: x^* = (3; -1; 0; 1; 3;…;3;-1;0;1); $f_{x*} = 0$

Rastrigin (n = 20)

Definition: $f_x = 10_n + \sum_{i=1}^{n}(x^2 - 10\ \cos(2\pi x_i))$
Domain: $[-2{,}56;\ 5{,}12]^n$
Global Minimum: x^* = (0;...;0); $f_{x*} = 0$

Sphere (n = 30)

Definition: $f_x = \sum_{i=1}^{n} x^2$
Domain: $[-2{,}56;\ 5{,}12]^n$
Global Minimum: x^* = (0;...;0); $f_{x*} = 0$