Experimenting unified differential evolution

Lhassane Idoumghar, Thomas Tosch (thomas.tosch@uha.fr),

1. University of Haute Alsace, Mulhouse, France

Abstract. This article displays experimentation results for the unified differential evolution algorithm. Multiple algorithms are used to measure performances of the unified Differential Evolution method, including Sphere, Rosenbrock, Griewank and Rastrigin. Initial results displayed no superiority in terms of performances. It displayed that fine tuning is a required step. The experimentation took place on a course held by Dr. Idoumghar. During the course, we took as assignment to produce test results over the unified Differential Evolution algorithm.

Keywords: Cuda, uDE, experimentation, sequential, GPU, Massively Parallel

1 Introduction

In this article we are going to compare multiple implementations experimentation results for the unified differential evolution algorithm. We will be using multiple algorithms to measure performances of the implementations, including Sphere, Rosenbrock, Griewank and Rastrigin. From a source code made by L. Idoumghar for a course over parallel programming, we implemented the unified mutation strategy and the previously mentioned functions. First we'll be describing how the Cuda language can be used to improve performances.

2 Cuda and Differential Evolution

To improve performance of the differential evolution kernel, we are using a C++ implementation of the Cuda framework. Cuda is a GPU compiled sub-language of C++ that allows to write instructions headed for the GPU to process. In theory, this allows to improve speed of loop-based algorithms in comparison to their sequential implementations. Cuda allows easy Threads management by letting the user set manually the amount of Threads that should run the code in parallel.

Standard GPUs like the Nvidia GTX 980 used in the computer this article in written on has up to 2048 cuda cores, allowing to run in parallel many operations. With Google Colab's P100 GPUs can process even more operations with 3584 cuda cores. To manage data with operations than the amount of cuda cores, we can take advantage of Blocks.

Blocks allow to to prepare the GPU to receive the amount of data of our operations. If we had 1 million operations, we would split them over 128 or 256 threads and take the upper value of the division. See following cuda implementation.

```
1       int blockSize = 256;
2       int numBlocks = (N + blockSize - 1) / blockSize;
3       evolutionKernel<<<numBlocks, blockSize>>>(N, x, y);
```

The implementation of our algorithm will now run over 256 threads until every operations have been done. To access the current iteration from our function, we have access to multiple built-ins from cuda. 1

blockIdx give us access on the current Block iteration. With blockDim we have access to the size of our blocks, which was set to 256 in our previous exemple. The threadIdx value provides the current thread the code is currently running in. Doing a simple calculation we can get the current iteration of our parallel implementation.

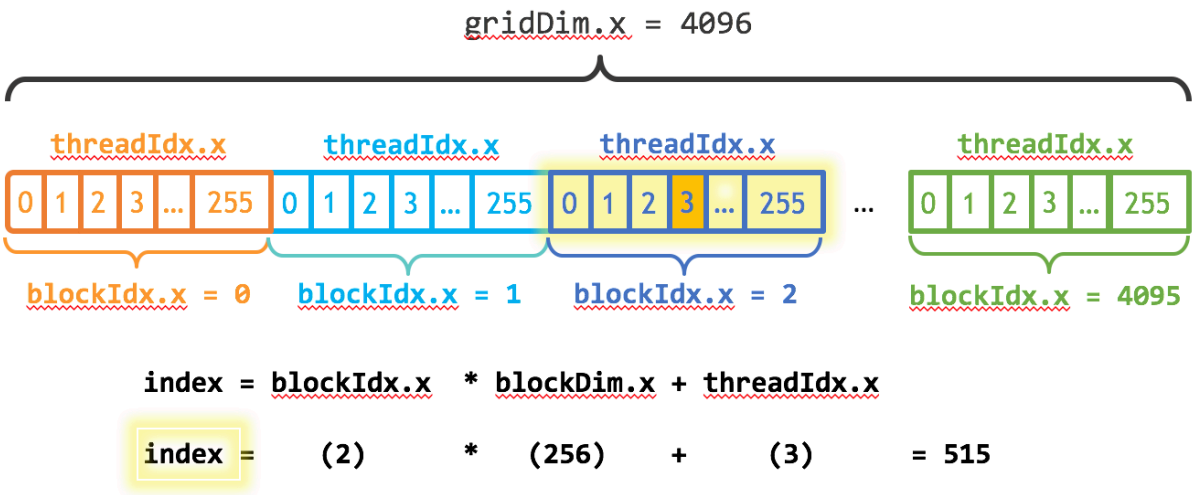

Figure 1: GPU Block distribution for a block size of 256 indexes.

3 Experimentation between Cuda and Sequential DE

3.1 Sequential Differential Evolution

An sequential implementation is used to compare performances of cuda DE. This implementation was worked on from the code base of the cuda DE. But with lack of time and understanding of the differential evolution algorithm, ended with an incomplete sequential implementation. The code ran correctly but there was no results showing up, implying that an error was occurring in the code base.

The code used to produce the sequential results was provided by Lhassane Idoumghar and was modified to run the same setup as the Cuda implementation.

In comparison, we can see overhaul that our uDE implementation 4 beats the sequential implementation 4. For the first function (F1, Shifted Sphere), performances seam to decrease with populations size. At P50, we can see close results. The sequential implementation increase in distance from the solution faster and faster when the population increases, this could be explained by the mutation strategy difference. Our sequential implementation is not using the unified mutation strategy, which could explain the result difference.

On the contrary, it seems that the run time of the sequential implementation keeps being below the uDE implementation. This could be explained by different factors, including the power comparaison between the GPU running the Cuda implementation being weaker than the CPU running the sequential implementation. It could also be explained by the initial kernel initialisation taking an initial setup time for the uDE implementation to be running.

3.2 Comparison with non-unified Cuda DE

Optimisation of our Cuda implementation was based on an article on the topic of the DE implementation Qin et al., 2012. This article shows result from their implementation and we can see way better results across all setups.

This could be explained by further optimisations on the implementation. The cross rate which decides when to mutate a generation was set to 0.8 both in the cited article and our implementation. Moving this value proved to change the results but tweaking it to different values didn't improve enough the results to conclude that our implementation was better.

4 Conclusion

Implementing the Cuda Unified Differential Algorithm was an interesting first hand experimentation over parallel programming. We learned how to solve large problems with the help of GPU computing. In the process of learning the uDE implementation we got to face limit in term of resources. Google Colab provides access to high performance GPU at low cost. The issue for us was that Google Colab is not always available and reserved resources are not always up to our implementation requirements. Personal Computers GPU's are currently a scarce resource and are very expensive, making it very hard to run our implementation locally.

		P50	P100	P500	P1000
F1	Best EFV	-450 (1.72633e-05)	-450 (6.10352e-06)	-450 (0)	-450 (0.000156564)
	Time (sec)	107.549 (379.604)	18.3662 (0.110718)	16.0359 (0.146722)	20.705 (0.152736)
F2	Best EFV	390.066 (0.268385)	390.038 (0.0798391)	390.467 (0.153961)	397.106 (1.89901)
	Time (sec)	46.1683 (0.187814)	27.02 (0.106834)	18.7858 (0.184944)	21.9927 (0.254022)
F3	Best EFV	-180 (0.00145861)	-180 (3.00563e-05)	-179.968 (0.0114503)	-179.58 (0.0850493)
	Time (sec)	52.9977 (0.0956197)	30.5071 (0.0944979)	19.5963 (0.131352)	21.8657 (0.154719)
F4	Best EFV	-330 (2.72958e-05	-330 (1.36479e-05)	-329.751 (0.0951895)	-327.077 (0.537065)
	Time (sec)	48.8638 (0.0985188)	28.6398 (0.10667)	19.5193 (0.155161)	22.1646 (0.167134)

Table 1: Performance of cudaDEi in terms of the mean value and standard deviation (in bracket) of the best EFVs achieved when the algorithm terminates, the mean value and standard deviation of computation time (seconds) over 25 runs with respect to four population sizes (P50, P100, P500 and P1000) on four 10D test problems.

		P50	P100	P500	P1000
F1	Best EFV	-450 (8.23409e-05)	-450 (7.22178e-05)	-450 (8.18873e-05)	-449.969 (0.00463213)
	Time (sec)	489.254 (1.35994)	291.19 (0.795224)	278.225 (0.34185)	296.061 (0.33632)
F2	Best EFV	398.396 (8.684)	391.399 (0.969263)	402.461 (2.92022)	777.548 (50.5921)
	Time (sec)	947.084 (2.2686)	517.377 (1.24267)	335.035 (0.579582)	315.189 (0.509783)
F3	Best EFV	-180 (2.76348e-05)	-180 (2.20065e-05)	-179.993 (0.00155314)	-178.9 (0.0118595)
	Time (sec)	1045.58 (2.39125)	576.956 (1.08437)	354.695 (0.278562)	337.632 (0.443143)
F4	Best EFV	-324.548 (1.44348)	-325.864 (1.29518)	-318.442 (0.859305)	-289.257 (2.89001)
	Time (sec)	974.584 (3.46262)	547.475 (1.18178)	346.441 (0.274041)	335.168 (0.464686)

Table 2: Performance of cudaDEi in terms of the mean value and standard deviation (in bracket) of the best EFVs achieved when the algorithm terminates, the mean value and standard deviation of computation time (seconds) over 25 runs with respect to four population sizes (P50, P100, P500 and P1000) on four 50D test problems.

		P50	P100	P500	P1000
F1	Best EFV	-450 (4.22864e-05)	-450 (3.76246e-05)	-450 (7.27318e-05)	-449.884 (0.0144555)
	Time (sec)	1760.79 (6.12097)	1066.28 (2.63083)	1044.45 (0.603709)	1075.89 (0.455502)
F2	Best EFV	419.009 (20.5124)	397.199 (3.52763)	454.351 (13.4546)	1976.69 (121.402)
	Time (sec)	3597.74 (8.89897)	1997.63 (3.65667)	1263.34 (0.471276)	1205.76 (0.580355)
F3	Best EFV	-180 (1.58574e-05)	-180 (1.67152e-05)	-179.99 (0.00177534)	-178.609 (0.0428756)
	Time (sec)	3927.31 (8.73395)	2212.07 (2.93232)	1340.37 (0.448757)	1275.16 (0.465865)
F4	Best EFV	-315.596 (2.1557)	-319.122 (1.445)	-298.954 (1.95028)	-227.196 (3.09385)
	Time (sec)	3624.83 (12.3424)	2056.47 (4.35575)	1301.48 (0.561792)	10244.64 (0.479688)

Table 3: Performance of cudaDEi in terms of the mean value and standard deviation (in bracket) of the best EFVs achieved when the algorithm terminates, the mean value and standard deviation of computation time (seconds) over 25 runs with respect to four population sizes (P50, P100, P500 and P1000) on four 100D test problems.

		P50	P100	P500	P1000
F1	Best EFV	-449.949 (0.03085)	-443.343 (2.50902)	-424.390 (5.85793)	-416.950 (7.65563)
	Time (sec)	40.0467 (0.54018)	19.1098 (0.08991)	3.94141 (0.02731)	4.99092 (0.06230)
F2	Best EFV	479.640 (117.265)	6501.15 (3014.45)	59715.9 (26848.9)	78228.1 (35366.5)
	Time (sec)	49.1264 (0.19695)	23.7485 (0.12731)	4.96472 (0.05372)	5.59174 (0.09675)
F3	Best EFV	-178.652 (0.55951)	-156.832 (5.55964)	-87.9838 (22.5366)	-69.5367 (24.4346)
	Time (sec)	59.2208 (0.77756)	29.0852 (0.63675)	5.65506 (0.03256)	6.24098 (0.25093)
F4	Best EFV	-316.160 (3.76779)	-275.907 (11.2746)	-224.565 (8.41156)	-219.329 (11.4247)
	Time (sec)	116.010 (2.15692)	56.3505 (0.36348)	11.3595 (0.20430)	8.97516 (0.15746)

Table 4: Performance of sequential DE in terms of the mean value and standard deviation (in bracket) of the best EFVs achieved when the algorithm terminates, the mean value and standard deviation of computation time (seconds) over 25 runs with respect to four population sizes (P50, P100, P500 and P1000) on four 10D test problems.

		P50	P100	P500	P1000
F1	Best EFV	-428.6508281269648 (7.31186516634)	-416.43723716835007 (7.46248125135)	-175.0586331270339 (20.2930189353)	-166.9323406377331 (13.5892907423)
	Time (sec)	398.79486084 (6.83218782717)	199.997520447 (3.23541555549)	35.9956741333 (0.884660652481)	25.1176261902 (0.484521369386)
F2	Best EFV	38584.96078865421 (31637.9479773)	84486.8229425085 (32527.4361176)	2280198.9614914614 (248689.528058)	288863.6979022934 (319602.872968)
	Time (sec)	1073.96501541 (12.1330055544)	531.776008606 (7.62798668248)	101.960744858 (1.94230040713)	56.6309928894 (1.18222359242)
F3	Best EFV	-121.09398131120842 (32.4645464267)	-67.56285417444157 (19.3275561292)	760.2936879165917 (53.886265316)	772.9666406909193 (59.8010458207)
	Time (sec)	709.627275467 (10.8141683129)	353.507890701 (5.3105627211)	63.2189464569 (1.31167483864)	38.2913208008 (0.798946033336)
F4	Best EFV	-79.623740010008924 (37.4225709068)	-28.686004559949474 (24.4262365985)	373.7213316873689 (27.6586666932)	378.27843529284206 (22.3217039518)
	Time (sec)	1895.06689072 (12.9376656385)	947.151832581 (9.19065972991)	183.579511642 (3.25836988147)	100.666484833 (2.01054807819)

Table 5: Performance of sequential DE in terms of the mean value and standard deviation (in bracket) of the best EFVs achieved when the algorithm terminates, the mean value and standard deviation of computation time (seconds) over 25 runs with respect to four population sizes (P50, P100, P500 and P1000) on four 50D test problems.

		P50	P100	P500	P1000
F1	Best EFV	-348.9110403621618 (31.4656810881)	-354.49249296454144 (17.9253908187)	152.87353824761124 (40.0193066207)	178.537593761033 (22.3186303876)
	Time (sec)	1296.62432671 (15.8790633798)	634.355812073 (5.5717792015)	122.545833588 (2.21842706967)	52.1974754333 (0.841795613206)
F2	Best EFV	449459.62218015397 (218826.968605)	385565.012507683 (132657.41586)	6239045.567351831 (640584.533488)	6217069.830197088 (680142.024476)
	Time (sec)	4128.26592445 (18.4313183884)	2086.10423088 (24.3454005862)	409.544181824 (9.39154906858)	188.517560959 (4.40024027043)
F3	Best EFV	200.77752524586992 (122.691968744)	145.09060398217264 (71.6982633971)	1911.00114045352 12 (131.522924153)	1963.54680269354 28 (96.9543675611)
	Time (sec)	2447.00567245 (27.4126138768)	1211.04588509 (12.885313635)	228.145046234 (1.70541642007)	94.4014263153 (0.962482381004)
F4	Best EFV	392.8674974204457 (62.815359619)	375.49432920232937 (57.6501011994)	1172.907069598961 (43.3628425427)	1199.8340422544854 (38.789967046)
	Time (sec)	6941.89335823 (11.9393969051)	3467.06618309 (4.62479523109)	689.184627533 (2.10331041466)	333.077316284 (1.23201251792)

Table 6: Performance of sequential DE in terms of the mean value and standard deviation (in bracket) of the best EFVs achieved when the algorithm terminates, the mean value and standard deviation of computation time (seconds) over 25 runs with respect to four population sizes (P50, P100, P500 and P1000) on four 100D test problems.

References

Qin, K., Raimondo, F., Forbes, F., & Ong, Y. (2012). An improved cuda-based implementation of differential evolution on gpu. *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation*. https://doi.org/10.1145/2330163.2330301