Solving One-dimensional IHCP with Particle Swarm Optimization using Graphics Accelerators

Sándor Szénási
John von Neumann
Faculty of Informatics
Óbuda University
Budapest, Hungary
Email: szenasi.sandor@nik.uni-obuda.hu

Imre Felde
John von Neumann
Faculty of Informatics
Óbuda University
Budapest, Hungary
Email: felde.imre@nik.uni-obuda.hu

István Kovács
John von Neumann
Faculty of Informatics
Óbuda University
Budapest, Hungary
Email: kovacs.istvan.perez@outlook.com

***Abstract*—There are several implicit and explicit formulations to solve the Inverse Heat Conduction Problem. One of the most promising methods is the Particle Swarm Optimization; however, it needs a long time to find solutions for large scale problems (large swarm populations). This paper presents the implementation and the evaluation of a parallel approach using graphics accelerators. This GPU implementation is about three times faster than the original CPU based method.**

I. Introduction

Reliable simulation of heat transfer processes requires functions and/or parameters describing the real heat transfer phenomena obtained during cooling. Typical quantities characterizes the heat extraction ability of the cooling medium is the Heat Transfer Coefficient (HTC) and/or the Heat Flux (HF). During quenching in liquids generating vapour and boiling stages (ie: oils, water, polymer solutions) the heat transfer is altering significantly depending on the surface temperature. In addition to, heat transfer is not uniform at the whole surface domain of the workpiece. It follows that realistic simulation of quenching is assumed to apply the thermal boundary conditions (HTC or HF) as functions of surface temperature and local coordinates as well.

To solve the IHCP [1], there are several implicit and explicit formulations. In the first approach, the problem is formulated as a multivariable optimization problem [2]–[4], while the latter method attempts to directly determine the unknown parameters using regularization techniques to solve the resulting system of equations [5]–[7]. In this paper, we use a stochastic approach that is based on particle swarm optimization.

II. Particle swarm optimization

Particle swarm optimization (PSO) [8]–[12] is a computational method for optimization problems (it cannot guarantee the optimal solution). The main idea is to iteratively try to improve a candidate solution. From several aspects, it is similar to genetic algorithms [13], as the optimization is based on a population (swarm) of candidate solutions (particles). However, there are several differences between the two methods. In the case of PSO, the search algorithm moves these particles around in the problem specific search-space using a position and velocity value for each. The movement direction of each particle is influenced by its best local position and the best known positions of the entire swarm.

The main steps of the basic algorithm are visible in Algorithm 1. The parameters of the function are as follows:

- N : size of swarm (number of particles)
- ϵ : minimum fitness change limit
- $limit$: maximum iteration limit

Algorithm 1 Basic particle swarm optimization

```
1:  function PSO(N, ε, limit)
2:      S[ ] ← InitializeSwarm(N)
3:      bestFitness ← ∞
4:      cnt ← 1
5:      repeat
6:          for i ← 1, N do                          ▷ Move items
7:              S_i.Velocity ← CalcVelocity(S_i)
8:              S_i.Position ← S_i.Position + S_i.Velocity
9:          end for
10:         for i ← 1 → N do                         ▷ Calculate fitness
11:             S_i.Fitness ← CalcFitness(S_i.position)
12:         end for
13:         cnt ← cnt + 1
14:         lastFitness ← bestFitness
15:         bestFitness ← min_{i∈1..N} x_i.Fitness
16:     until lastFitness − bestFitness < ε ∨ cnt > limit
17:     return bestFitness
18: end function
```

III. Data parallel implementation

As it is visible in the algorithm, several parts of the PSO algorithm are well parallelizable. Both loops process the items one by one, and it is clear that there is no interference between the iterations. Furthermore, we have to do the same calculations on all particles without any communication. These are all embarrassingly parallel calculations. Therefore, these parts are executable in a data parallel fashion, which is ideal for GPU implementation [14].

The swarm creation is well parallelizable too, but this part only runs once; therefore, its execution speed is not relevant.

As it is visible in Algorithm 2, the first parallel part is the particle movement section. We should start as many threads as the number of the particles. Each thread will calculate the new velocity and (based on this) the new location of each particle.

In the case of GPU development, the high ratio of arithmetic/memory operations is very important in regard to obtaining the maximum performance [15]. The above mentioned part does not fulfil this condition. The GPU multiprocessors have to load and save all swarm data from the device memory (for example, in the case of 1000 particles that use 10 dimensional double coordinates, it means loading/storing of 80K), meanwhile the number of arithmetic calculations is relatively low (we use a simple method to calculate the new velocity, and the calculation of the new position needs only one 10 addition operations). Nonetheless, it is worth implementing this part in the GPU, because this technique helps to avoid unnecessary memory copies from GPU device memory to host memory and vice versa.

The parallelization of the second loop is much more efficient. We should calculate the fitness values for each particle. The naïve method uses one thread for the calculation of the fitness of one particle. The memory requirement for each thread is a little lower (they only need the position of the swarm, and there is no modification and store operations), while the arithmetical complexity is significantly higher.

For the fitness calculation, we are using the explicit finite difference method (FDM) presented by Smith. In case of one-dimensional heat transfer simulation, our goal is to solve the following equations:

- For the middle line (i = 0) (Eq. 1):

$$T(n+1,0) = T(n,0) + (dt * \alpha(T)) * \frac{1}{dx^2} * 2 * (T(n,1) - T(n,0)) \quad (1)$$

- For the surface (i = N) (Eq. 2):

$$T(n+1,N) = T(n,N) + (dt * \alpha(T)) * (\frac{1}{dx^2} * 2 * (T(n,N-1) - T(n,N) - \frac{dx}{k} * (HTC(t) * (T(n,N) - T_{cm})) + \frac{1}{N * dx} * \frac{-1}{k(T)} * (HTC(t) * (T(n,N) - T_{cm}))) \quad (2)$$

- And for the inner points (i = 1..N-1) (Eq. 3):

$$T(n+1,i) = T(n,i) + (dt * \alpha(T)) * (\frac{1}{dx^2} * (T(n,i-1) + T(n,i+1) * 2 * T(n,i)) + \frac{1}{i * dx} * \frac{1}{2 * dx} * (T(n,i+1) - T(n,i-1))) \quad (3)$$

- Where
  - α - thermal diffusivity (Eq. 4)

$$\alpha(T) = \frac{k(T)}{cp(T) * \rho} \quad (4)$$

  - k - thermal conductivity
  - cp - specific heat capacity
  - ρ - density
  - htc - heat transfer coefficient
  - N - number of points

In this case, the ratio of arithmetic/memory operations is high, which leads to good GPU performance. It is difficult to theoretically estimate the speed-up of a GPU algorithm; therefore, we implemented this naive method and ran several tests to measure the execution time.

Algorithm 2 Data parallel particle swarm optimization

```
1:  function PSO(N, ε, limit)
2:      S[ ] ← InitializeSwarm(N)
3:      bestFitness ← ∞
4:      cnt ← 1
5:      repeat
6:          for all x ∈ swarm do parallel
7:              x.Velocity ← CalcVelocity(x)
8:              x.Position ← x.Position + x.Velocity
9:          end for
10:         for all x ∈ swarm do parallel
11:             x.Fitness ← CalcFitness(x.position)
12:         end for
13:         cnt ← cnt + 1
14:         lastFitness ← bestFitness
15:         bestFitness ← min_{i∈1..N} x_i.Fitness
16:     until lastFitness − bestFitness < ε ∨ cnt > limit
17:     return bestFitness
18: end function
```

IV. Test results

A. Hardware configurations

We used the following configurations for the tests:

- CPU configuration
  - Processor: Intel® Core™ i7-2600
  - Architecture: Sandy Bridge
  - Number of cores: 4
  - Memory: 16GB DDR2
- GPU configuration
  - Graphics accelerator: NVIDIA Tesla K40c
  - Architecture: Kepler (GK110B)
  - Number of shaders: 2880
  - SMX Count: 15
  - Memory: 12GB GDDR5
  - Host: the same as the CPU config

B. Runtime test

In the case of the CPU, we use the C# built-in "parallel for" statement to parallelize the calculations (using all available cores). As expected, the running time is linearly dependent on the number of cores. We need a minimum number of particles to utilize the processing power of all CPU cores, but this number is relatively small. Using swarms with 10 or more particles is enough to reach this maximum performance. Above this limit, the processing time is linearly increasing with the number of items. Thread scheduling is the responsibility of the C# compiler and the .NET environment, but in practice, we can assume that the number of threads is equal to the number of cores; therefore, every thread handles more than one particle.

In the case of the GPU, according to the architecture of the device, the calculation of the estimated runtime is much more complex. First of all, the used graphics accelerator (Tesla K40 Active) has 2880 processing elements; therefore, it is obvious that we need a lot of threads to utilize all of them. The thread to particle assignment is manual, and our main concept was that every thread handles one particle; therefore, we need at least 2880 particles to use all the cores. In fact, a lot more than this is needed as the GPU tries to hide latency using fast context switching mechanism and thus, we need three-four times more particles to fully utilise the processing power of the GPU.

As it is expected, the evaluation of swarms containing few particles is faster in the CPU implementation (one GPU core is significantly slower than one CPU core). As it is visible in Table I, increasing the swarm size leads to a smaller difference between the measured running times. In addition, there is a limit (about 350 items) where the lead position is swapped. For larger swarms, the GPU becomes faster (Fig. 1).

As we discussed before, we have to start a lot of threads to utilize all processing elements of the GPU. It is visible in Fig. 2 that the speed-up (the ratio of the CPU and the GPU running time) increases rapidly for a smaller number of elements, and becomes a constant at about 8000-10000 or more elements. In these cases, the GPU is three times faster than the CPU, and this condition persists in the case of larger swarms.

TABLE I
RUNTIME OF THE CPU AND THE GPU IMPLEMENTATION

Test	Swarm	CPU runtime (sec)	GPU runtime (sec)	CPU/GPU
1	100	1.53	3.86	0.40
2	200	2.85	4.36	0.65
3	300	4.17	4.64	0.90
4	400	5.52	4.96	1.11
5	500	6.76	5.29	1.28
6	600	8.09	5.70	1.42
7	700	9.39	6.21	1.51
8	800	10.70	6.58	1.63
9	900	12.35	6.90	1.79
10	1000	13.44	7.44	1.81
11	1100	14.82	7.68	1.93
12	1200	16.06	8.19	1.96
13	1300	17.55	8.53	2.06
14	1400	18.83	8.92	2.11
15	1500	20.11	9.37	2.15
16	1600	21.45	9.64	2.23
17	1700	22.39	10.10	2.22
18	1800	23.95	10.52	2.28
19	1900	25.46	10.92	2.33
20	2000	26.93	11.54	2.33
21	2100	28.26	12.01	2.35
22	2200	29.69	12.41	2.39
23	2300	30.95	12.84	2.41
24	2400	32.34	13.20	2.45
25	2500	33.68	13.56	2.48
26	2600	34.87	13.97	2.49
27	2700	36.95	14.40	2.57
28	2800	37.65	14.79	2.55
29	2900	39.04	15.12	2.58
30	3000	40.44	15.63	2.59
31	3000	40.24	15.64	2.57
32	4000	53.97	19.81	2.73
33	5000	67.73	23.86	2.84
34	6000	80.31	28.01	2.87
35	7000	93.55	32.01	2.92
36	8000	106.91	36.29	2.95
37	9000	120.45	40.24	2.99
38	10000	134.37	44.61	3.01
39	11000	147.24	48.72	3.02
40	12000	160.25	53.18	3.01
41	13000	174.81	57.26	3.05
42	14000	187.32	61.84	3.03
43	15000	201.47	65.74	3.06
44	16000	215.69	73.27	2.94
45	17000	227.72	77.70	2.93
46	18000	242.07	81.79	2.96
47	19000	253.72	85.90	2.95
48	20000	268.96	89.97	2.99

C. Accuracy test

The CPU implementation of the PSO algorithm was an already tested application; therefore, we used this as a reference. To compare the results, we deactivated the random number generator functions (which were always started with the same starting seed number), and ran run several tests using this configuration. The final results of all the CPU and GPU executions were exactly the same. Both the CPU and GPU code use double precision arithmetic for the position/velocity/fitness data and all internal calculations.

V. CONCLUSIONS

We have implemented the particle swarm optimization based IHCP method for both the CPU and the GPU. The original algorithm has several well parallelizable parts; therefore, our expectation was that the GPU implementation would need significantly less time to solve the given equations. As it is clearly visible from the results, our expectations were justified; the GPU implementation is about three times faster.

This speed-up strongly depends on the size of the swarm. In the case of small swarms, it is worth using the original CPU base implementation. Though, in the case of larger swarms (350 particles or more), the GPU code becomes faster (this speed-up increase continues until the number of particles is equal to 10000). However, there are no strict rules for determining the number of elements, but in practice, it is common to use more than 350 particles.

Deeper analysis of the GPU code and execution statistics shows that the largest hindering limit is the high number of de-

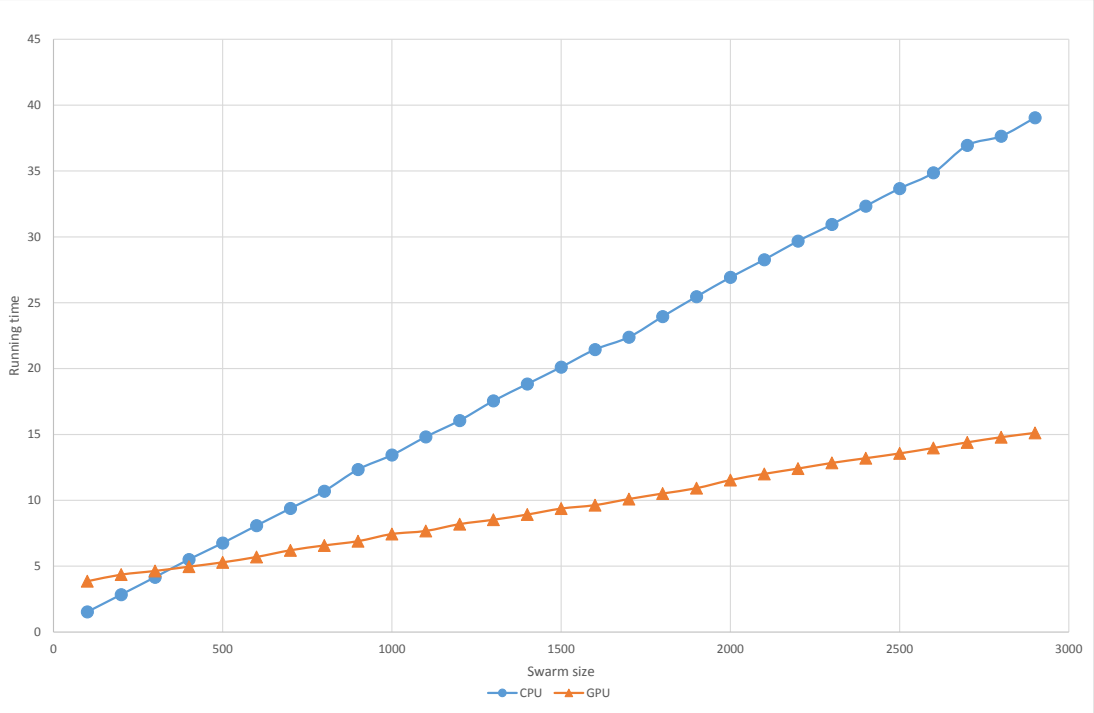

Fig. 1. Running time of the CPU and the GPU implementation of the PSO algorithm.

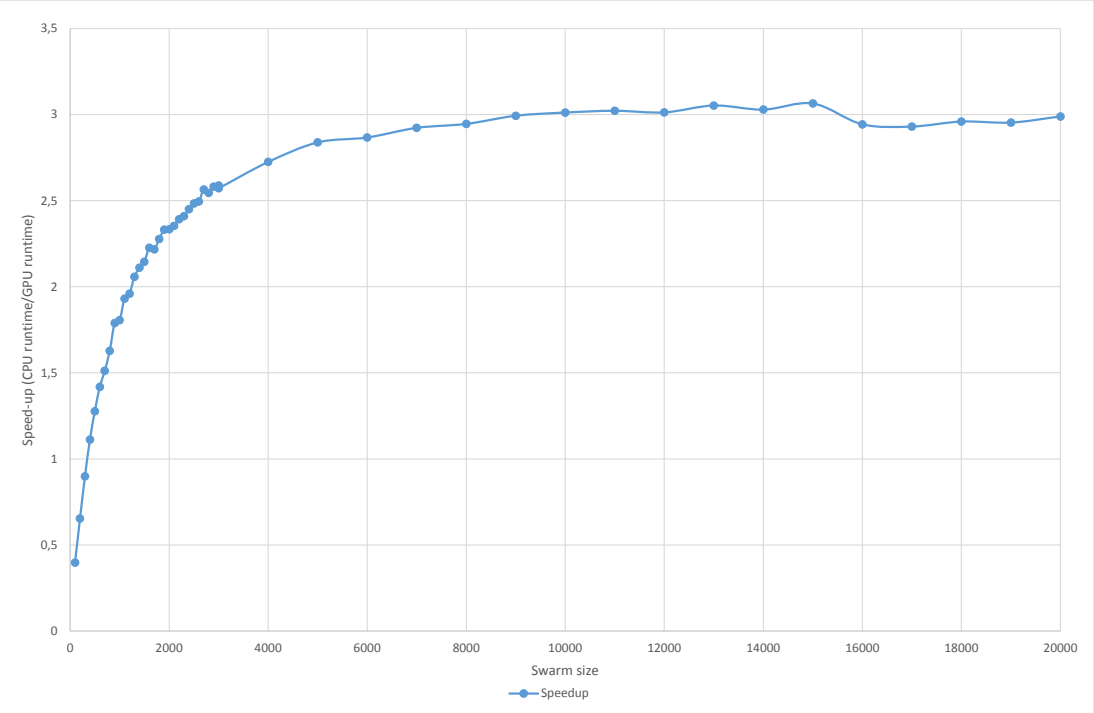

Fig. 2. Speed-up of the GPU algorithm compared to the CPU algorithm using the same swarm size and data.

vice memory access operations. Our future plan is to optimize the GPU code to use the shared memory of the multiprocessors to avoid device memory access where possible.

Acknowledgements

We acknowledge the financial support of this work by the Hungarian State and the European Union under the TÁMOP-4.2.2/A-11/1-KONV-2012-0029 and the Chinese-Hungarian S&T bilateral project TÉT_12_CN-1-2012-0027 projects. The authors would like to thank NVIDIA Corporation for providing graphics hardware for the GPU benchmarks through the CUDA Teaching Center program. The authors also wish to thank the members of the CUDA Teaching Center at Óbuda University for their constructive comments and suggestions. The content of the paper however does not necessarily express the views of these companies and persons, the authors take full responsibility for any errors or omissions.

REFERENCES

[1] I. Felde and W. Shi, "Hybrid approach for solution of inverse heat conduction problems," in *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*, Oct 2014, pp. 3896–3899.

[2] O. M. Alifanov, *Inverse Heat Transfer Problems*. Springer, 1994.

[3] M. N. Özisik and H. R. B. Orlande, *Inverse Heat Transfer: Fundamentals and Applications*. Taylor & Francis, 2000.

[4] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965.

[5] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence properties of the nelder–mead simplex method in low dimensions," *SIAM Journal on Optimization*, vol. 9, no. 1, pp. 112–147, 1998.

[6] R. Das, "A simplex search method for a conductive–convective fin with variable conductivity," *International Journal of Heat and Mass Transfer*, vol. 54, pp. 5001–5009, 2011.

[7] O. Nelles, *Nonlinear system identification*. Springer-Verlag, 2001.

[8] J. Kennedy, "Particle swarm optimization," in *Encyclopedia of Machine Learning*, C. Sammut and G. Webb, Eds. Springer US, 2010, pp. 760–766.

[9] F. Valdez, P. Melin, and O. Castillo, "An improved evolutionary method with fuzzy logic for combining Particle Swarm Optimization and Genetic Algorithms," *Applied Soft Computing*, vol. 11, pp. 2625–2632, 2011.

[10] R.-E. Precup, R.-C. David, E. Petriu, S. Preitl, and A. Paul, "Gravitational search algorithm-based tuning of fuzzy control systems with a reduced parametric sensitivity," in *Soft Computing in Industrial Applications*, ser. Advances in Intelligent and Soft Computing. Springer Berlin Heidelberg, 2011, vol. 96, pp. 141–150.

[11] A.-C. Zavoianu, G. Bramerdorfer, E. Lughofer, S. Silber, W. Amrhein, and E. P. Klement, "Hybridization of multi-objective evolutionary algorithms and artificial neural networks for optimizing the performance of electrical drives," *Engineering Applications of Artificial Intelligence*, vol. 26, no. 8, pp. 1781–1794, 2013.

[12] N. El-Hefnawy, "Solving bi-level problems using modified particle swarm optimization algorithm," *International Journal of Artificial Intelligence*, vol. 12, no. 2, pp. 88–101, 2014.

[13] S. Szénási and Z. Vámossy, "Evolutionary algorithm for optimizing parameters of GPGPU-based image segmentation," *Acta Polytechnica Hungarica*, vol. 10, no. 5, pp. 7–28, 2013.

[14] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[15] NVIDIA, *CUDA C Programming Guide*, NVIDIA Corporation, 2014.