2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

# Solving the Multidimensional Knapsack Problem using a CUDA accelerated PSO

Drahoslav Zan and Jiri Jaros

*Abstract*— **The Multidimensional Knapsack Problem (MKP) represents an important model having numerous applications in combinatorial optimisation, decision-making and scheduling processes, cryptography, etc. Although the MKP is easy to define and implement, the time complexity of finding a good solution grows exponentially with the problem size. Therefore, novel software techniques and hardware platforms are being developed and employed to reduce the computation time. This paper addresses the possibility of solving the MKP using a GPU accelerated Particle Swarm Optimisation (PSO). The goal is to evaluate the attainable performance benefit when using a highly optimised GPU code instead of an efficient multi-core CPU implementation, while preserving the quality of the search process. The paper shows that a single Nvidia GTX 580 graphics card can outperform a quad-core CPU by a factor of 3.5 to 9.6, depending on the problem size. As both implementations are memory bound, these speed-ups directly correspond to the memory bandwidth ratio between the investigated GPU and CPU.**

## I. Introduction

The Particle Swarm Optimisation (PSO) is a stochastic technique initially designed for non-linear continuous function optimisation. The algorithm is inspired by the dynamic behaviour of bird flocks or fish schools while seeking for food. It has become a widely applied and adapted optimisation tool since developed by Kennedy and Eberhart in 1995 [1].

The main advantages of the PSO is an extremely simple algorithm and a low number of control parameters. On the other hand, many candidate solutions are necessary to be created and evaluated when optimising complex problems. Fortunately, the PSO is very easy to parallelise since the particles do not depend on each other while moving in the search space. Many approaches thus simulate multiple particles at a time or propose a multiple swarm versions of the PSO [2]. The trend over last few years has been to utilize Graphics Processing Units (GPUs) as general purpose co-processors. Although originally designed for rasterisation and widely used in the game industry, their raw arithmetic power has attracted a lot of researchers [3].

The PSO community has adopted this trend relatively quickly and a lot of papers have been presented in this area, e.g. [4], [5], [6]. The main drawback of these and many other papers is that they investigate the performance only on trivial numerical benchmarks with no data. Of course, in such a case the GPU performance will be stunning as no memory is needed. However, when a real problem is being optimised, many issues arise such as how to arrange the benchmark's data in memory, how to fully exploit the memory bandwidth when scatter accesses to memory are necessary, how to employ cache memories and so on. In many cases, what originally looked very promising and motivating turns out to be very difficult to achieve or even disappointing. The second downside of many performance studies is that only relative performance comparisons are provided. However, for predicting the realistic speed-up on different GPUs, the performance counters have to be investigated to gain some insight into the algorithm efficiency (what percentage of the raw performance the code can exploit while utilising a given fraction of main memory bandwidth).

Therefore, this paper investigates the attainable performance of the GPU accelerated PSO when solving a real world problem with a large dataset, the Multidimensional Knapsack Problem [7]. The performance is provided in terms of relative speed-up with respect to a parallel CPU implementation. The paper also investigates the absolute performance in terms of GFLOPS and GB/s and states what fraction of the theoretical performance can be harnessed. This easily allows to predict the performance on many other GPUs.

## II. Multidimensional Knapsack Problem

The knapsack problem is one of the famous challenges in combinatorial optimisation. The idea behind it can be explained by the following example. Suppose a hitch-hiker needs to fill a knapsack for a trip. There are many items available to select, however, the capacity of the knapsack is limited. The hitch-hiker tries to maximise the overall value (usefulness) of the items in the knapsack while not overloading it. Although sounds simple, the standard knapsack problem represents an NP-hard optimisation problem [8].

There are several generalisations of the standard knapsack problem, one of which is the Multidimensional Knapsack Problem (MKP) [9]. The MKP can formally be defined as follows. There is a set of $n$ items with profits $p_j$ and a set of $m$ resources with capacities $M_i$. Each item $j$ consumes an amount $w_{ij}$ of each resource $i$. The variable $x_j$ holds the decision whether or not the item $j$ is selected for given resources. The goal is to the maximise total profit by selecting the appropriate subset of all items (1). The necessary condition is that the selected items must not exceed resource capacities – *knapsack constraint* (2).

The typical area of MKP applications is scheduling (manufacturing lines, airports, etc.). A recent review of the MKP can be found in the literature [7].

Drahoslav Zan and Jiri Jaros are with Department of Computer Systems, Brno University of Technology, Bozetechova 2, 612 00 Brno, Czech Republic (email: {izan, jarosjir}@fit.vutbr.cz).

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

MKP:

$$\text{maximise} \quad f(\vec{x}) = \sum_{j=1}^{n} p_j x_j, \tag{1}$$

$$\text{subject to} \quad f(\vec{x}) = \sum_{j=1}^{n} w_{ij} x_j \leq M_i, \;\; M_i > 0, \tag{2}$$
$$i = 1, \ldots, m, \;\; x_j \in \{0, 1\},$$
$$p_j > 0, \;\; w_{ij} \geq 0, \;\; j = 1, \ldots, n.$$

## III. Background for the PSO Based MKP Solver

This section describes the background of the CPU (PSO-C) and the GPU (PSO-G) version of the proposed PSO based MKP solver. Although both implementations (see section IV) follow the same algorithm (sec. III-A), the implementations are tailored to the CPU or GPU needs.

### A. The PSO Algorithm

The Particle Swarm Optimization (PSO) [1] is a population-based stochastic technique for iterative solving of both continuous and discrete problems. The PSO solves a given problem by having a population of particles which represent candidate solutions. These particles are grouped into a swarm and moved around the problem search space. The particle movement is influenced by their local best known positions in the space and by the global best position of the swarm (which is a common knowledge amongst all particles). Using these two paradigms, the swarm is expected to efficiently explore the search space and eventually find the global optima.

The PSO algorithm can be expressed by the two following equations:

$$\vec{v}^{\,i}_{k+1} = w\vec{v}^{\,i}_{k} + \varphi_1\delta_1\left(\vec{p}^{\,i}_{k} - \vec{x}^{\,i}_{k}\right) + \varphi_2\delta_2\left(\vec{p}^{\,g}_{k} - \vec{x}^{\,i}_{k}\right) \tag{3}$$

$$\vec{x}^{\,i}_{k+1} = \vec{x}^{\,i}_{k} + \vec{v}^{\,i}_{k+1} \tag{4}$$

Eq. (3) describes the velocity of each particle $i$ in all dimensions of the search space in the next iteration of the algorithm $(k+1)$. The inertia weight factor $w$ makes the particle movement smoother and add another variability to the system apart form the stochastic parameters $\delta_{\{1,2\}} \in [0, 1)$ [10]. The actual change in the particle velocity depends on the distance from the best local $\vec{p}^{\,i}_{k}$ and the best global known position $\vec{p}^{\,g}_{k}$, where $\vec{x}^{\,i}_{k}$ is the actual position of the particle $i$. The ratio between particle nostalgia and enviousness is controlled by the cognitive and social parameters $\varphi_1$ and $\varphi_2$. Subsequently, eq. (4) defines the position of each particle $\vec{x}^{\,i}_{k+1}$ in the next iteration by adding a distance travelled in a unit time. Let us note that choosing the values of control parameter is not trivial [11] and is usually done empirically.

*1) Discrete version of the PSO:* In order to solve the MKP problem, we need to define a discrete version of the PSO [12]. Here, the particle position is defined as a vector of binary values $\vec{x}^{\,i} = (x_{i1}, x_{i2}, \ldots, x_{ij}, \ldots, x_{in})$, where $n$ is the size of the problem (the number of dimensions). The particle velocity $\vec{v}^{\,i} = (v_{i1}, v_{i2}, \ldots, v_{ij}, \ldots, v_{in})$ then holds the probability of vector $\vec{x}^{\,i}$. The particle velocity is updated by eq. (3), however, since the particle velocity is restricted to stay within an interval of $[0, 1]$, it has to be normalised afterwards by a sigmoid transformation function:

$$S(v_{ij}) = \frac{1}{1 + e^{-v_{ij}}} \tag{5}$$

The particle position is updated by the following decision function ($\delta \in [0, 1)$):

$$x_{ij} = \begin{cases} 1 & \delta \leq S(v_{ij}) \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Let us now describe the flow of the PSO algorithm. At the beginning, the particles are randomly spread over the search space, their velocities randomly generated and the local best and global best positions initialised. After initialisation, the iterative core of the PSO repeats until termination conditions are met (e.g. convergence criterion, discovery of a sufficient solution, number of iterations, etc.). For simplicity, we terminate the PSO after a predefined number of iterations. At the beginning of every iteration, the particle fitness (quality of the candidate solution) is evaluated, see section III-B. Consequently, local best positions are updated if necessary, and a new leader is voted. Finally, the particle positions and velocities are updated according to eq. (3) and (4). After a termination criterion has been met, the best solution is reported.

### B. The MKP Fitness Function

There have been proposed several fitness functions to solve the MKP problem [13]. We decided to implement the one shown in eq. (7). The fitness function here is defined as a linear composition of two components. The first evaluates the overall profit of the selected items, while the second one penalises the solution if the knapsack capacity has been exceeded. The amount of penalisation is proportional to the level of exceeding further scaled by a parameter $P$. Some strategies on how to choose parameter $P$ can be found in literature [14].

$$g(\vec{x}) = \underbrace{\sum_{j=1}^{n} p_j x_j}_{\text{profit}} - \underbrace{P \sum_{i=1}^{m} poslin\left(\sum_{j=1}^{n} w_{ij} x_j - M_i\right)}_{\text{penalty}} \tag{7}$$

The core of the penalisation is the $poslin$ function:

$$poslin(x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases} \tag{8}$$

## IV. Implementation of the Solvers

Both the PSO-C and PSO-G solvers are based on the same principles explained in sections III-A and III-B. The main difference between them lies in the implementation of the particular sections of the PSO algorithm (*initialise swarm*, *update swarm*, *calculate fitness of each particle*) as well as in the swarm encoding.

### A. The MKP Data Representation

The MKP is defined by a list of $n$ items, their profits, and their weights in $m$ different knapsacks (consumed resources). The item profits are stored in a one-dimensional array while the weights are organized in a two-dimensional array of a size of $m \times n$, where a row maintains item weights for a given knapsack. In the case of the GPU implementation, the first dimension (the number of items) is padded to the nearest power of 2.

### B. The Swarm Data Representation

A particle swarm is represented by a set of particles, each of which maintains a scalar fitness value (**fit**), a vector of current positions (**pos**), a vector of current velocities (**vel**), and a vector of the local best positions (**lbp**). Since CPU and GPU architectures require different data layouts to reach maximum efficacy, two distinct data representations were designed (see. Fig. 1).

In the case of CPU, every particle is processed sequentially dimension by dimension. Here, the best spatial locality is achieved by an array of structures representation. The data for each particle is stored together. In Fig. 1a, the size of all components are given in bits. The symbol $d$ represents the number of dimensions. Since the positions are represented as bit arrays, we need to pad the number of dimensions to a multiple of 8 (full bytes). This extended number of dimensions is denoted by $d_c$. The particle velocity and the fitness value are then represented as single precision floating point values. Finally, symbol $p$ represents the total number of particles.

On the other hand, GPU processes each particle in parallel. A block of threads cooperates on evaluating the fitness value and updating its velocity and position. Therefore, it is desirable to allow neighbour threads to access neighbour particle dimensions. This is done by storing the swarm data as a structure of arrays. The particle components are thus stored together (first all positions, then all best positions, then all velocities, and all fitness values).

Another difference is in the particle position representation. Since every thread manipulates a single dimension, using a bit array would cause a need for mutual exclusion on the byte level (threads must not modify a single byte concurrently). Therefore, a single dimension is represented as a whole byte. In order to make the fitness function calculation fast, the number of dimensions is padded to a nearest power of two.

| | **fit** | **vel** | **pos** | **lbp** |
|---|---|---|---|---|
| 1 | 32 | $32 \times d$ | $1 \ldots d_c$ | $1 \ldots d_c$ |
| 2 | 32 | $32 \times d$ | $1 \ldots d_c$ | $1 \ldots d_c$ |
| $\vdots$ | | | $\vdots$ | |
| $p$ | 32 | $32 \times d$ | $1 \ldots d_c$ | $1 \ldots d_c$ |

(a) PSO-C

| | 1 | 2 | $\ldots$ | $d_g$ | |
|---|---|---|---|---|---|
| **pos** | 8 | 8 | $8 \times (d_g - 2)$ | | $\times p$ |
| **lbp** | 8 | 8 | $8 \times (d_g - 2)$ | | $\times p$ |
| **vel** | 32 | 32 | $32 \times (d_g - 2)$ | | $\times p$ |
| **fit** | $32 \times p$ | | | | |

(b) PSO-G

Fig. 1: Memory layouts of the PSO-C and PSO-G solvers. The PSO-C uses an array of structures representation while the PSO-G benefits from a structure of arrays representation.

### C. The PSO-G Solver Implementation

At the beginning, memory for the swarm (Fig. 1b) is allocated in the global GPU memory and consequently initialized by the *init kernel*. Since the initialization involves random numbers generation, we use the Random123[1] library to do this task quickly. After the swarm has been initialized, the MKP memory is allocated and the data copied into the global GPU memory.

Afterwards, the search for an optimal solution is launched. Every iteration starts with evaluating of the particle fitness by the *fitness kernel*. All $p$ particles are evaluated in parallel, every particle is processed by a single block of threads. A given particle is then evaluated in parallel by the threads inside the block. There are as many threads in the block as there are dimensions (items) in the MKP instance ($d_g$), thus one thread processes one dimension.

Each thread first reads its item profit, multiply it by the value (0 or 1) of the particle position in a given dimension and stores the result into the shared memory. After a barrier, the partial profits are summarised using a parallel reduction[2]. In order to simplify the implementation, the number of items (dimension/threads) is ceiled to the nearest power of two. After another barrier, the threads calculate the level of excess in particular knapsacks by iterating over them. Then, the decision whether to penalise the solution fitness is made. When the new fitness value is known, the local best solution may be replaced by the new one and the fitness kernel terminates. Now, a new swarm leader must be established. This is done by a reduction routine from the library Nvidia Thrust library[3] on the **fit** array (Fig. 1b).

As soon as the new leader of the swarm has been established, the swarm can be updated by the *update kernel*. The kernel is divided into $p$ thread blocks, each of which further divided into $d$ threads. Thus, a single thread updates one dimension on a given particle using eq. (3). Since the maximum velocity has to be limited, clamping is used. Unfortunately, this introduces some thread divergence. Finally, the sigmoid function in eq. (5) is calculated. Evaluation of this function is expensive due to division and exponentiation involved.

[1] http://www.deshawresearch.com/resources_random123.html
[2] http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
[3] https://developer.nvidia.com/Thrust

### D. The PSO-C Solver Implementation

The PSO-C solver implementation is more straightforward than the implementation of the PSO-G. The GPU kernels are here replaced by simple C functions. These functions are paralellized with the use of OpenMP[4] directives to employ all available CPU cores. When iterating over the swarm, every thread processes the particles in quadruplets. This along with loop unrolling enables easy automatic vectorisation (SSE/AVX instruction generation) by the compiler.

## V. Experimental Results

This section examines the potential of the graphic processing cards in accelerating the PSO algorithm when compared to a modern multi-core system. The section investigates the level of acceleration, its stability and scaling with the problem size and puts the values in the context of the performance characteristics of the GPU architecture (compute power and memory bandwidth). The performance measurements are completed with examination of the solution quality produced by both solvers.

Both sets of experiments were mostly executed on the Chu and Beasley's benchmark set [15]. Unfortunately, the Chu and Beasley's benchmark set does not provide known optima and also does not cover the whole range of instances needed for a proper performance and quality comparison. Therefore, Weingartner and Ness [16] MKP instances were used for tests which required known optima, and a set of randomly generated MKPs was created and used for fine testing. The benchmark set used for a given comparison is specified above particular figures.

### A. Benchmark System

The performance of the proposed implementations was investigated using a single desktop equipped with a quad-core intel i7-920 (2.66 GHz, 8MB L3 cache) CPU and an NVIDIA GTX 580 GPU (512 cores, 1.5 GHz, 768 kB L2). The theoretical peak performance of the CPU is 84 GFLOPS in single precision while the memory bandwidth is about 25 GB/s. On the other hand, the GPU offers about 1.5 TFLOPS in single precision and a memory bandwidth of 192 GB/s. The attainable speed-up when exploiting the best of both architectures could sit between 8 and 18, depending on whether the application is likely to be memory or compute bound.

### B. The MKP Solver Parameters

The PSO algorithm is known to require only a small number of control parameters. However, the proper choice of them may improve the search capabilities and the convergence speed by a great deal. Since this paper is mainly focused on the raw performance comparison, the control parameters were inspired by a proper literature [11]. More specifically, the inertia coefficient $w$ was set to 1, the cognitive and social coefficients $\varphi_1$ and $\varphi_2$ to 2, and the maximum reachable particle velocity to 10. The penalty coefficient $P$ was set to 5000.

[4]http://www.openmp.org

### C. Performance Analysis

The first set of experiments examines the proposed MKP solvers from the performance point of view. Fig. 2 compares the PSO-G and PSO-C performance on a four knapsack problem with a number of items growing from 64 up to 1024. In the figure, we can see that the speed-up offered by the PSO-G strongly depends on both the number of particles and the number of items (dimensionality of the search space).

Generally, the higher number of particles, the higher speed-up is reached. This is given by the fact that GPUs need much bigger amount of work to be fully utilised than CPUs do. It is very likely that with higher numbers of particles than 256, the speed-ups would be even higher. However, as shown in the literature [14], such high numbers of particles do not improve the algorithm convergence, yet increase compute expenses.

The speed-up curves are not smooth but contain several drops in performance. These drops are a consequence of an optimisation decision we made in order to make reduction kernels inside the fitness calculation as fast as possible (working in a single pass). However, this imposes a restriction on the number of dimension to be a power of two. Any time the number of dimensions (items) crosses a power of two, the particles must be padded to the nearest power of two. This, of course, introduces unnecessary work to be done on the GPU and deteriorates the performance benefit compared to the CPU where no reduction kernels are necessary. We limited the amount of useless work by running the update kernels only over the meaningful dimensions.

The reached speed-up is summarised in Table I. The average speed-up over all problem sizes for a fixed number of particles varies from 1.9 to 5.2 for the number of particles from 32 to 256, respectively. The peak speed-ups observed for the problems of 512 dimensions are significantly higher

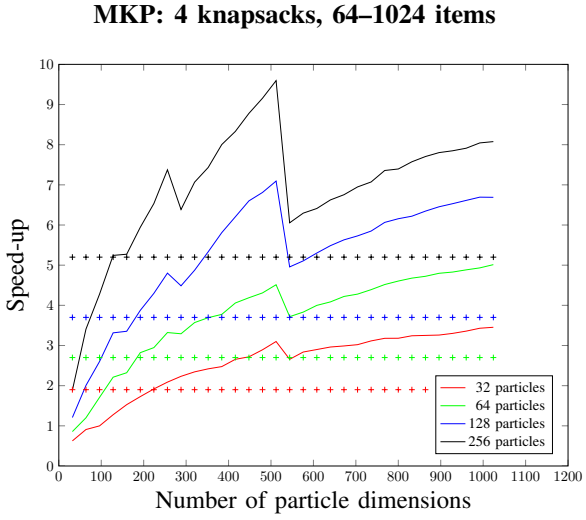


Fig. 2: The variation in speed-up reached by the PSO-G over the PSO-C with growing number of knapsack items (problem dimension) and different numbers of particles.

TABLE I: Average and peak speed-up of the PSO-G over the quad-core PSO-C solver along with the performance metrics for the PSO-G.

**MKP: 4 knapsacks, 64–1024 items**

| | GPU Speed-up | | Absolute performance | |
|---|---|---|---|---|
| Number of particles | Average | Peak | Peak GFLOPS | Global memory [GB/s] |
| 32 | 1.9 | 3.5 | 10 | 41 |
| 64 | 2.7 | 5.0 | 29 | 40 |
| 128 | 3.7 | 7.0 | 40 | 50 |
| 256 | 5.2 | 9.6 | 45 | 51 |

and reach about twice as high values as the average speed-ups. The highest speed-up of 9.6 was observed at 512 dimensions and 256 particles. Considering that both PSO and fitness evaluation of a MKP instnace have linear time complexities of $O(n)$, the proposed PSO solvers are expected to be memory bound. This thought is partially confirmed by the measured global memory bandwidth of almost 50 GB/s (25 %) of the raw memory bandwidth.

Consequently, the memory bandwidth limits the compute power of the GPU that can be harnessed to a very low 45 GFLOPS (3 % of the theoretical compute power). Although this may be seen as a very bad result, the literature [17] shows that this is a usual value for such a kind of problems.

The same analysis for the CPU shows that the resources are used with a bit higher efficacy. Assuming the CPU performance reaches about 10% of the GPU version, we can conclude the compute performance exploited is about 4.7 GFLOPS and about 5GB/s of available memory bandwidth. This value is comparable with the performance of many HPC algorithms.

**MKP: 30 knapsacks, 500 dimensions**

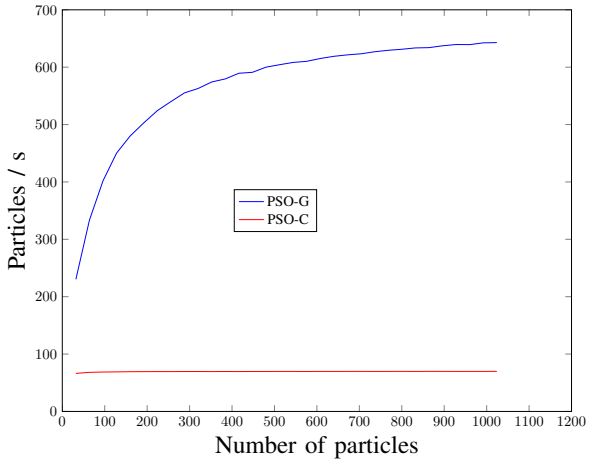


Fig. 3: The number of particles calculated per one second using PSO-C and PSO-G. The performance for PSO-G grows with increasing numbers of particles, however seems to be constant for the CPU.

Finally, we investigated the number of particles that can be calculated per second on a given platform, when the swarm size is being progressively increased. The results shown in Fig. 3 say that the CPU performance is almost independent on the swarm size, however the GPU performance saturates at 1000 particles and 500 dimensions. More interestingly, the absolute numbers say that at the peak performance we are able to evaluate over 600k particles or 3M dimensions per second using this MKP instance.

### *D. Analysis of the Solution Quality*

In order to verify that the quality of the search process was not influenced by the implementation, several tests were performed. During these tests, the number of particles, generations and different benchmarks were investigated. Generally, the differences between the solutions produced by the PSO-C and PSO-G were smaller than one percent. This differences can be attributed to the stochastic character of the PSO. We also investigated the deviation from the global known optima and the best solution produced. Although, we have not tuned the PSO parameters extensively, the solutions provided by the MKP solvers are very close to the global optima. A comparison made on the MKP instance with 2 knapsacks and 28 dimensions is summarised in Table II.

## VI. Conclusions

During the last few years, Graphical Processing Units (GPUs) have been used to accelerate a wide range of compute intensive algorithms. In this work, we focussed on the Particle Swarm Optimisation and investigated the potential performance benefit of employing a single GPU in solving the Multidimensional Knapsack Problem (MKP) on real world instances. The question we attempted to answer is how fast we could be if we tailor the MKP solver to the GPU needs, use a real world benchmark and do not harm the produced solution quality.

To make our comparison as fair as possible, we first developed a fine-tuned multi-threaded CPU version of the code and compared it with the proposed GPU code. The performance comparison were made on real test cases proposed by Chu and Beasley [15] and by Weingartner and Ness [16]. The experimental runs have shown that a single Nvidia GTX 580 can outperform a quad-core Intel CPU running at 2.6 GHz by a factor of 2 to 5 on average. The peak speed-up observed for large problems starts at 3.5 and closes at

TABLE II: Dependency of the quality of the solution on the particles count.

**MKP: 2 knapsacks, 28 dimensions**

| Particles count | Deviation from the known optimum | Deviation between PSO-C and PSO-G |
|---|---|---|
| 32 | 1.88% | 3.8% |
| 64 | 0.83% | $< 1\%$ |
| 128 | 0.40% | $< 1\%$ |
| 256 | 0.21% | $< 1\%$ |

almost 10. In other worlds, a single GPU could compete with a hypothetical 40 core CPU.

Since relative performance comparisons may be misleading due to the strong dependence on the chosen base line (the CPU code), we also investigated hardware performance counters. These showed that only 3 % of the raw theoretical performance was efficiently used while utilising almost 25 % of available memory bandwidth. As most of GPU cards on the market have very similar chip architectures and memory subsystems, the performance of the proposed code can be simply predicted. For example, having the best Kepler card on the market offering 3.95 TFLOPS and 250 GB/s of memory bandwidth, the code could reach a speed-up of 12 with respect to the quad-core CPU (the code is memory bound). Putting the results into the context of other memory bound application (search and sorting algorithms, fast Fourier transform, etc.), the code is highly competitive [17].

In the future, we would like to improve the performance by utilising more memory bandwidth, remove some code limits (the highest number of dimensions to be greater than 1024) and propose a multi-GPU implementation of the PSO MKP solver. We would also like to focus on advanced PSO variants for solving the MKP problem.

## ACKNOWLEDGMENT

This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033).

This work was also supported by the grant "Architecture of parallel and embedded computer systems", FIT-S-14-2297, Brno University of Technology, Czech Republic.

## REFERENCES

1. J. Kennedy and R. Eberhart, "Particle Swarm Optimization," in *IEEE International Conference on Neural Networks*, vol. 4, 1995, pp. 1942–1948.
2. T. Hendtlass, "WoSP: A Multi-optima Particle Swarm Algorithm," in *The IEEE Congress on Evolutionary Computation*, vol. 1, 2005, pp. 727–734.
3. D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann,, 2010.
4. D. L. Souza, G. D. Monteiro, T. C. Martins, V. A. Dmitriev, and O. N. Teixeira, "PSO-GPU: accelerating particle swarm optimization in CUDA-based graphics processing units," in *Genetic and Evolutionary Computation Computation*, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 837–838. [Online]. Available: http://doi.acm.org/10.1145/2001858.2002114
5. Y. Tan and Y. Zhou, "Parallel Particle Swarm Optimization Algorithm Based on Graphic Processing Units," in *Handbook of Swarm Intelligence*, vol. 8. Springer Berlin Heidelberg, 2010, pp. 133–154. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17390-5_6
6. L. Mussi, F. Daolio, and S. Cagnoni, "Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture," *Information Sciences*, vol. 181, no. 20, pp. 4642 – 4657, 2011.
7. J. Puchinger, G. R. Raidl, and U. Pferschy, "The multidimensional knapsack problem: Structure and algorithms," *INFORMS Journal on Computing*, vol. 22, no. 2, pp. 250–265, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1287/ijoc.1090.0344
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.
9. J. Puchinger, G. Raidl, and U. Pferschy, "The core concept for the multidimensional knapsack problem," *Evolutionary Computation in Combinatorial Optimization*, 2006. [Online]. Available: http://link.springer.com/chapter/10.1007/11730095_17
10. J. Bansal, P. Singh, M. Saraswat, A. Verma, S. Jadon, and A. Abraham, "Inertia Weight strategies in Particle Swarm Optimization," in *World Congress on Nature and Biologically Inspired Computing*, 2011, pp. 633–640.
11. Z. Qingqing, H. Xingshi, and S. Na, "Convergence Analysis and Parameter Select on PSO," in *International Symposium on Information Science and Engineering*. Washington, DC, USA: IEEE CS, 2009, pp. 144–147. [Online]. Available: http://dx.doi.org/10.1109/ISISE.2009.27
12. J. Kennedy and R. Eberhart, "A discrete binary version of the particle swarm algorithm," in *IEEE International Conference on Systems, Man, and Cybernetics*, vol. 5,

1997, pp. 4104–4108 vol.5.
13. A. Olsen, "Penalty functions and the knapsack problem," in *IEEE World Congress on Computational Intelligence and Evolutionary Computation*, 1994, pp. 554–558 vol.2.
14. D. Ardagna, C. Francalanci, V. Piuri, and F. Scotti, "Evolutionary Design of Information Systems Architectures," in *Artificial Intelligence and Soft Computing*, ser. LNCS, vol. 3070. Springer, 2004, pp. 1–8.
15. P. C. Chu and J. E. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *Journal of Heuristics*, vol. 4, no. 1, pp. 63–86, Jun. 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1009642405419
16. H. M. Weingartner and D. N. Ness, "Methods for the solution of the multi-dimensional 0/1 knapsack problem," *Operations Research*, vol. 15, pp. 83–103, 1967.
17. V. W. Lee, P. Hammarlund, R. Singhal, P. Dubey, and C. Kim, "Debunking the 100X GPU vs. CPU myth," in *International symposium on Computer architecture*. New York, New York, USA: ACM Press, 2010, p. 451. [Online]. Available: http://dl.acm.org/citation.cfm?id=1816021