# An Implicitly Parallel EDA Based on Restricted Boltzmann Machines

Malte Probst
University of Mainz
Dept. of Information Systems and Business Administration
Mainz, Germany
probst@uni-mainz.de

Franz Rothlauf
University of Mainz
Dept. of Information Systems and Business Administration
Mainz, Germany
rothlauf@uni-mainz.de

Jörn Grahl
University of Mainz
Dept. of Information Systems and Business Administration
Mainz, Germany
grahl@uni-mainz.de

## ABSTRACT

We present a parallel version of RBM-EDA. RBM-EDA is an Estimation of Distribution Algorithm (EDA) that models dependencies between decision variables using a Restricted Boltzmann Machine (RBM). In contrast to other EDAs, RBM-EDA mainly uses matrix-matrix multiplications for model estimation and sampling. Hence, for implementation, standard libraries for linear algebra can be used. This allows an easy parallelization and leads to a high utilization of parallel architectures. The probabilistic model of the parallel version and the version on a single core are identical. We explore the speedups gained from running RBM-EDA on a Graphics Processing Unit. For problems of bounded difficulty like deceptive traps, parallel RBM-EDA is faster by several orders of magnitude (up to 750 times) in comparison to a single-threaded implementation on a CPU. As the speedup grows linearly with problem size, parallel RBM-EDA may be particularly useful for large problems.

## Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Parallel programming*; I.2.6 [**Artificial Intelligence**]: Learning—*Neural Networks*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

## Keywords

Parallelization; Restricted Boltzmann Machines; Estimation of Distribution Algorithms; Machine Learning; Combinatorial Optimization Problems; GPU; Neural Networks

## 1. INTRODUCTION

As most computing power of modern systems comes from multi-core CPUs and Graphics Processing Units (GPUs), parallelization of algorithms has become an important topic in optimization. We present a parallel version of RBM-EDA, an Estimation of Distribution Algorithm (EDA) which uses a Restricted Boltzmann Machine (RBM). RBMs are a special type of neural network. EDAs use a population of solutions and capture dependencies between problem variables in a probabilistic model. Subsequently, they use this model to sample new solutions, which compete with the existing population, based on their fitness [see e.g. 14, 10]. RBM-EDA uses an RBM as its model.

In [24], we analyze the performance of RBM-EDA for different combinatorial optimization problems such as Onemax, concatenated deceptive traps and NK-landscapes and compare it to the state-of-the-art Bayesian Optimization Algorithm (BOA [22]). RBM-EDA can solve complex combinatorial optimization problems with a number of fitness evaluations that grows polynomial in the problem size. At the same time, the computational complexity is lower that that of BOA. That is, a single-threaded RBM-EDA can build its probabilistic model for large, complex problems significantly faster, leading to lower total run times.

An RBM can capture complex multivariate dependencies. Nevertheless, the effort for implementing a parallel RBM-EDA is comparatively low. The parallel implementation is flexible and can be ported to different hardware environments (e.g. a multi core CPU system, an ARM device, a system using GPUs). The reason for the simplicity lies in the nature of an RBM. As a neural network, it is implicitly parallel. That is, most operations necessary to build the model and to sample from it are independent of each other. Formally, almost all operations are either matrix multiplications or element-wise matrix operations. Those operations can be implemented using standardized Basic Linear Algebra Subprogram (BLAS) libraries. Parallelization of the whole optimization algorithm essentially means using a parallel BLAS library.

As a consequence, the algorithm can make efficient use of parallel resources if the BLAS library is implemented efficiently. Furthermore, the sequential and parallel versions of the probabilistic model are identical and model quality is not affected by parallelization.

Our experiments indicate that the parallel version comes with massive speedups. For example, the GPU version can solve large concatenated trap problems up to 750× faster than a single-threaded CPU version. Also, the speedup grows with problem size.

The layout of the paper is as follows: Section 2 introduces EDAs and RBMs and shows how to use RBMs in EDAs. In Section 3, we discuss basic parallelization techniques for EDAs and show how RBM-EDA can be parallelized using

*GECCO'14,* July 12–16, 2014, Vancouver, BC, Canada.
Copyright 2014 ACM 978-1-4503-1964-5/14/07 ...$15.00.
http://dx.doi.org/10.1145/2576768.2598273.

**Algorithm 1** Pseudo code for main EDA loop

```
1: Initialize Population $P$
2: while not converged do
3:     $P_{parents}$ ← Select high quality solutions from $P$ based on their fitness
4:     $M$ ← Build a model estimating the (joint) probability distribution of $P_{parents}$
5:     $P$ ← Sample new candidate solutions from model $M$
6: end while
```

BLAS libraries. Section 4 introduces the benchmark problem, concatenated deceptive traps, presents our experimental setup and shows the speedups RBM-EDA achieves using BLAS on a CPU or GPU. We discuss results and implications in Section 5.

## 2. PRELIMINARIES

We review the basic concept of EDAs. We introduce RBMs, describe the relevant elements Gibbs sampling and contrastive divergence [4, 6], and show how an RBM can be used in an EDA.

### 2.1 Estimation of Distribution Algorithms

EDAs are well-established tools for solving combinatorial optimization problems [see e.g. 14, 10]. The basic structure of EDAs is given by Algorithm 1. In a nutshell, they select promising individuals from a population, build a probabilistic model of this population and then use this model to sample new individuals. These new individuals are evaluated and usually form the new population. This loop continues until the population has converged. The underlying assumption is that a model, which has captured the essence of the old population, is able to sample new, unknown individuals that possess the same high-quality structure, thereby searching the solution space efficiently.

EDAs differ in their choice of the model. Simple models use a vector with activation probabilities for each variable of the problem, while neglecting dependencies between the variables [e.g. UMDA, see 14]. Slightly more complex models use pairwise dependencies modeled as trees or forests [23]. More complex dependencies can be captured by models with multivariate interactions, like ECGA, BOA or DEUM [5, 22, 26]. Multivariate models are better suited for complex optimization problems, as univariate models can cause an exponential growth of the required number of fitness evaluations for growing problem sizes [22, 21]. Many algorithms use probabilistic graphical models with directed edges, i.e. Bayesian networks, or undirected edges, i.e. Markov random fields [9]. Hence, model building consists of finding a network structure that matches the problem structure and estimating the model's parameters. Usually, the computational effort to build the model rises with model complexity and representational power. Hence, parallelization is particularly useful for complex models.

### 2.2 Restricted Boltzmann Machines in EDAs

A Restricted Boltzmann Machine is a stochastic neural network where connections between neurons form a bipartite graph [28]. An RBM forms a Markov random field, as all connections are undirected. We can use an RBM as a model within an EDA, because it can be trained to model a probability distribution and it is possible to draw samples from this model [24]. Figure 1 illustrates the structure of an RBM. The input or visible layer $V$ holds the input data represented by $n$ binary variables $v_i, i = 1, 2, \ldots, n$. The $m$ binary neurons $h_j, j = 1, 2, \ldots, m$ of the hidden layer $H$ are called feature detectors as they are able to model patterns in the data. A weight matrix $W$ holds weights $w_{i,j} \in \mathbb{R}$ between all neurons $v_i$ and $h_j$. Each neuron in the visible as well as hidden layer makes a stochastic decision whether to be active after collecting inputs from all connected neurons. The weights $W$ determine the strengths of the influence. An RBM encodes a joint probability distribution $P(V, H)$, which is the probability of sampling a joint configuration of $V$ and $H$ [28]. In the training phase, the weights are adapted in such a way that the marginal probability $P(V)$ of the training data is maximized. Training and sampling are tractable because of the bipartite structure of an RBM.

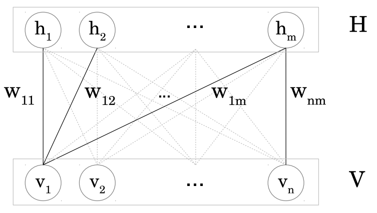


**Figure 1: Restricted Boltzmann Machine, visualized as a graph. The visible neurons $v_i$ ($i \in 1..n$) can hold a data vector of length $n$ from the training data. The hidden neurons $h_j$ ($j \in 1..m$) represent $m$ features. Weight $w_{ij}$ connects $v_i$ to $h_j$. In the EDA context, $V$ represents the decision variables**

#### 2.2.1 Sampling

Sampling new values for the visible neurons $V$ is straightforward, if the activations of the hidden layer $H$ are known. The $v_i$ are independent of each other. The conditional probability $P(v_i = 1|H)$ that the visible neuron $v_i$ is active is calculated as

$$P(v_i = 1|H) = sigm\left(\sum_j w_{ij}h_j\right), \quad (1)$$

where $sigm(x) = \frac{1}{1+e^{-x}}$ is the logistic function.[1] Analogously, for a given vector $V$, the conditional probability $P(h_j = 1|V)$ for the hidden neurons $H$ is calculated as

$$P(h_j = 1|V) = sigm\left(\sum_i w_{ij}v_i\right). \quad (2)$$

Unfortunately, sampling from the joint probability distribution $P(V, H)$ is much more difficult than sampling from the conditional distributions as it usually requires integrating over one of the joint distributions. An alternative is *Gibbs sampling* which approximates the joint distribution $P(V, H)$ from the conditional distributions $P(V|H)$ and $P(H|V)$.

[1]In addition, all neurons are connected to a special "bias" neuron, which is always active and works like an offset to the neuron's input. Bias weights are treated like normal weights during learning. Due to brevity, we omit bias terms throughout the paper. For details, see [7].

Gibbs sampling starts by assigning random values to the visible neurons. Then, it iteratively samples from $P(H|V)$ and $P(V|H)$, respectively, while assigning the result of the previous sampling step to the non-sampled variable. The iterative sampling of $V$ and $H$ ($V \rightarrow H \rightarrow V \rightarrow ...$) forms a Markov chain, whose stationary distribution is identical to the joint probability distribution $P(V, H)$ [4]. The higher the number of sampling steps, the better the approximation of the joint probability distribution $P(V, H)$. When starting Gibbs sampling with a $V$ that is close to the stationary distribution, only a few sampling steps are necessary to obtain a good approximation.

#### 2.2.2 Training

The joint probability distribution $P(V, H)$ of an RBM is encoded in its weights $W$. An effective approach for learning $P(V, H)$ by adjusting the weights of an RBM is contrastive divergence (CD) learning [6]. CD learning maximizes the log-likelihood of the training data under the model $P(V, H)$ by performing a stochastic gradient ascent. The main element of CD learning is Gibbs sampling.

For each data point $V$ in a training sample, CD learning updates $w_{ij}$ in the direction of $-\frac{\partial log(P(V))}{\partial w_{ij}}$, where $P(V)$ is the probability of $V$. It can be shown that this partial derivative is the difference of two terms usually referred to as positive and negative gradient, $\Delta_{ij}^{\text{pos}}$ and $\Delta_{ij}^{\text{neg}}$ [6]. Consequently, the total gradient $\Delta w_{ij}$ becomes

$$\begin{aligned} \Delta w_{ij} = \quad & \Delta_{ij}^{\text{pos}} & - \quad & \Delta_{ij}^{\text{neg}} \\ = & < v_i \cdot h_j >^{\text{data}} & - \quad & < v_i \cdot h_j >^{\text{model}}, \end{aligned} \tag{3}$$

where $< >$ denotes expectation. $\Delta_{ij}^{\text{pos}}$ is the expected value of $v_i h_j$ when setting the visible layer $V$ to a data vector from the training set and sampling $H$ according to (2). $\Delta_{ij}^{\text{pos}}$ increases the probability of a configuration $V$. In contrast, $\Delta_{ij}^{\text{neg}}$ is the expected value of a configuration sampled from the joint probability distribution $P(V, H)$, which is approximated by Gibbs sampling. If $P(V)$ is equal to the distribution of the training data, in the expectation, the positive and negative gradient equal each other and the total gradient becomes zero.

Calculating $\Delta_{ij}^{\text{neg}}$ exactly is infeasible, as the number of required Gibbs sampling steps until the RBM is sampling from its stationary distribution is high. Hence, contrastive divergence usually uses a much lower number $N$ of sampling steps (CD-$N$). For CD-N, $\Delta_{ij}^{\text{neg}}$ is approximated by initializing the visible neurons with a data vector from the training set and performing only $N$ Gibbs sampling steps.

Algorithm 2 summarizes the functionality of CD-1 ($N = 1$). For each training vector, $V$ is initialized with the training vector and $H$ is sampled according to (2). This allows calculating $\Delta_{ij}^{\text{pos}}$ as $v_i h_j$. Then, two more sampling steps are carried out: First, we calculate the "reconstruction" $\hat{V}$ of the training vector as in (1). Subsequently, we calculate the corresponding hidden probabilities $P(\hat{h_j} = 1|\hat{V})$. Now, we can approximate $\Delta_{ij}^{\text{neg}}$ as $\hat{v}_i \cdot P(\hat{h_j}|\hat{V})$ and obtain $\Delta w_{ij}$. Finally, we update the weights as

$$w_{ij} := w_{ij} + \alpha \cdot \Delta w_{ij} \tag{4}$$

where $\alpha \in (0, \ldots, 1)$ is a learning rate defined by the user.

**Algorithm 2** Pseudo code for a training epoch using CD-1

```
 1: for all training examples do
 2:     $V$ ← set $V$ to the current training example
 3:     $H$ ← sample $H|V$, i.e. set $h_j$ to 1 with $P(h_j = 1|V)$ from (2)
 4:     $\Delta_{ij}^{\text{pos}} = v_i h_j$
 5:     $\hat{V}$ ← sample "reconstruction" $\hat{V}|H$, using (1)
 6:     $\hat{H}$ ← calculate $P(\hat{H}|\hat{V})$ as in (2)
 7:     $\Delta_{ij}^{\text{neg}} = \hat{v}_i \cdot P(\hat{h}_j|\hat{V})$
 8:     $\Delta w_{ij}$ ← calculate all $\Delta w_{ij}$ as in (3)
 9:     $w_{ij}$ ← update all weights according to (4)
10: end for
```

#### 2.2.3 Using RBMs in EDAs

RBM-EDA uses an RBM as its model to capture the probability distribution of the population. In each generation of the EDA, the individuals surviving the selection step are used as training data for the RBM. Using contrastive divergence learning, the weights of the RBM are updated until a termination criterion has been reached. A simple termination criterion is the convergence of the reconstruction error, i.e. the degree to which the RBM is able to reconstruct the training data after one Gibbs sampling step $V \rightarrow H \rightarrow \hat{V}$. After training, the RBM is used to sample new individuals, which are then incorporated into the population.

In contrast to other EDAs using Markov random fields like DEUM [26], it is not necessary to specify a network structure in advance. The bipartite nature of the RBM makes Gibbs sampling a feasible means for approximating samples from the fully connected model's equilibrium distribution.

## 3. PARALLELIZATION OF EDAS

We discuss approaches to parallelize EDAs and the major challenges. We then show how to parallelize RBM-EDA.

### 3.1 Basic concepts and challenges

Parallelization has been done for many EDAs, including the above mentioned UMDA, ECGA and BOA [see e.g. 27, 8, 17, 19, 15, 13, and others]. Parallel versions usually result in significant speedups compared to the single-core versions. Often, there are moderately diminishing returns on increasing the number of cores.

It is straightforward to parallelize fitness evaluation. Furthermore, simple EDA models like UMDA can be parallelized easily, as all parts of the model (i.e. all variables) are independent. In contrast, parallel building and sampling become more difficult for complex models with multivariate dependencies [18]. When distributing tasks of unknown, and potentially different, sizes to multiple processors, job scheduling and load balancing mechanisms are required to maximize utilization [see e.g. 20, 13].

In multivariate EDAs, interdependent parts of the model may have to exchange information. To facilitate efficient parallelization, this communication can be reduced by using sub-populations or islands [11]. Alternatively, it is sometimes possible to introduce additional assumptions to render separate parts independent of each other. For example, the parallelization approaches for BOA assume a predetermined order of nodes in the dependency graph.

Depending on the hardware platform, specific restrictions must be considered to maximize the utilization of the parallel resources. For example, when running in a cluster environment, the parallelization scheme must consider latencies between nodes. When using Graphics Processing Units (GPUs), there is only a limited amount of very fast shared memory available and memory access to slower global memory has to be done in a coalesced fashion to maximize the utilization of the memory bandwidth [see e.g. 15].

Thus, parallelization itself and subsequent tuning for maximum hardware utilization may pose a significant challenge. Sometimes, the necessary algorithmic changes also result in slightly reduced model quality. For example, the parallel version may need larger populations to match the quality of its sequential counterpart [see e.g. 27]. Hence, besides parallelization being complicated, there is often a trade-off involved: maximizing the utilization of the hardware does not always entail maximizing the model quality, and vice versa.

## 3.2 Parallelization of RBM-EDA

It is simple and efficient to parallelize the training and sampling of RBM-EDA. Also, parallelization can be achieved without changes to the model itself. The reason is that, due to their bipartite structure, RBMs are implicitly parallel. Furthermore, the actual implementation can use standard libraries for most of the computational steps.

### *3.2.1 Implicit parallelism*

The RBM's bipartite structure factorizes the distributions of both visible and hidden layer. That is, a visible neuron $v_i$ is independent of all other visible neurons $v_{j\neq i}$, given the activation of the $m$ neurons of the hidden layer, and vice versa. Hence, the activation of all $v_i$ can be calculated by $n$ independent threads. This holds for training and sampling.

Furthermore, contrastive divergence, which is used for training, is a stochastic gradient ascent algorithm and usually implemented in a mini-batch fashion. That is, we simultaneously calculate $\Delta w_{ij}$ in (4) for each training example in a mini-batch of size $B$, and subsequently use the average to update $w_{ij}$. This reduces sampling noise and makes the gradient more stable [1, 7]. The initial population in an EDA is usually random. Thus, the signal-to-noise ratio is low and the batch size $B$ can be even larger than in classical, well structured training tasks.

Hence, when training with mini-batches of batch size $B$, the activation of $B * m$ independent neurons has to be calculated to perform a single Gibbs sampling step $V_b \rightarrow H_b$ for all $b = 1, 2, \dots, B$ using Equation 2. Accordingly, to perform a single Gibbs sampling step $H_b \rightarrow V_b$, the activation of $B * n$ independent neurons is calculated using (1). Thus, for training, the number of independent neurons grows linearly with the problem size $n$, given that $m$ is some fixed fraction of $n$. When sampling new candidate solutions, all individuals are independent, so $B$ is equal to the number of new individuals. Hence, for sampling, the number of independent neurons grows linearly with the problem size *and* with the population size.

Recall that, unlike other EDA models based on probabilistic graphical models, RBM-EDA does not separate the search for a suitable network structure from parameter estimation. During training, the connectivity of the RBM remains unchanged. Only the strength of the connections are adjusted. Hence, there is no need for mechanisms balancing unequal workloads due to different subproblem sizes.

### *3.2.2 Usage of BLAS libraries*

Formally, calculating a single sampling step $V_b \rightarrow H_b$ consist of two steps: first, we perform one vector-matrix multiplication $V \times W$. Subsequently, we apply the sigmoid function to each element of the result vector. The same applies to the calculation of $H_b \rightarrow V_b$. When performing mini-batch learning or sampling, we combine all $B$ vector-matrix multiplications into a single matrix-matrix multiplication.

Efficient implementations of matrix-matrix operations are part of Basic Linear Algebra Subprogram (BLAS) libraries with standardized high-level interfaces [see e.g. 3]. These libraries are available for almost every platform (e.g. ATLAS, ATLAS for ARM, CUBLAS, Intel MKL, and others). Using a BLAS library comes with two major benefits: first, BLAS libraries are usually optimized to use the full potential of a single core. For example, modern BLAS libraries can make use of the processor's hardware extensions such as SSE (Streaming SIMD Extensions). Here, identical instructions that have to be be carried out many times on multiple data items can be processed simultaneously in parallel registers of a single CPU core. Second, there are numerous parallel BLAS libraries, which can make use of multiple cores or use special hardware, like GPUs. As the interface is standardized, switching to different parallelization platform can be as easy as - technically spoken - changing the "#import" statement which specifies the used library. As most of the operations to train and sample an RBM are covered by BLAS libraries, the amount of custom code for the parallelization is very low (see Section 4.2).

## 4. EXPERIMENTS

This section introduces the test problem, concatenated deceptive traps, and presents our experimental setup. We then show the speedups of the single-threaded BLAS and the GPU version of EDA-RBM in comparison to a naive serial implementation.

## 4.1 Test Problem

We evaluate the speedup on a standard benchmark problem for EDAs - concatenated deceptive traps. Concatenated deceptive traps are tunably hard, yet decomposable test problems [2]. Here, a solution vector $x$ is divided into $l$ subsets of size $k$, with each one being a deceptive trap. Within a trap, all bits are dependent on each other but independent of all other bits in $x$. Thus, the fitness contribution of the traps can be evaluated separately and the total fitness of the solution vector is the sum of these terms[2]. In particular, the assignment $a = x_{i:i+k-1}$ (i.e. the $k$ bits from $x_i$ to $x_{i+k-1}$) leads to a fitness contribution $F_l$ as

$$F_l(a) = \begin{cases} k & \text{if } \sum_i a_i = k, \\ k - (\sum_i a_i + 1) & \text{otherwise.} \end{cases}$$

The fitness of a single trap increases with the number of zeros, except for the optimum of all ones.

[2]In general, the $k$ variables assigned to trap $l$ do not have to be adjacent but can be at any position in $x$. Due to easier visualization, however, we assume in the remainder of this paper that they are adjacent. This goes without loss of generality, because the model does not assume any neighborhood relation between adjacent $x_i$.

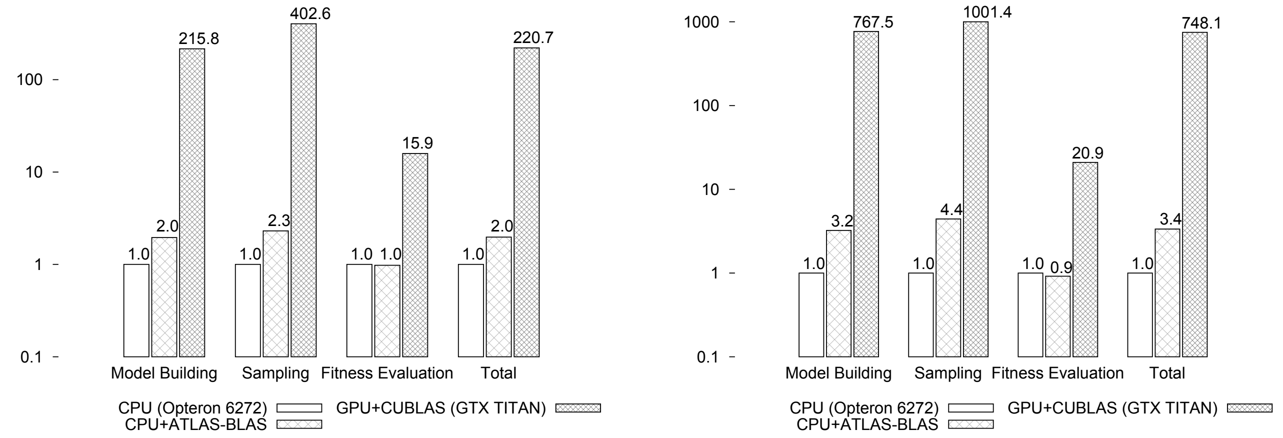


**Figure 2: Speedups for solving concatenated trap-5 problems. Left hand side: $l = 20$ concatenated traps; right hand side: $l = 70$ concatenated traps**

## 4.2 Experimental Setup

We test three versions of the EDA implemented in C++. The base version, henceforth referred to as *CPU* version, is a single threaded RBM, which performs all operations sequentially. That is, all matrix operations are carried out in nested for-loops. Second, we test a version of RBM-EDA which uses an optimized ATLAS BLAS library for all linear algebra operations, but is still restricted to one CPU core. Hence, model building and sampling operations utilize the SIMD extensions of the CPU core. The third version uses CUBLAS, the BLAS library of CUDA, the general purpose programming language for NVIDIA GPUs. We run the GPU code on an NVIDIA GeForce GTX TITAN GPU with 2,688 cores.

In the ATLAS and CUBLAS versions, all matrix multiplication operations necessary for model building and sampling are covered by the BLAS libraries. For all other operations, i.e. mainly the sigmoid function in (1) and (2), and the stochastic activations, we wrote approximately 20 lines of custom CUDA/C++ code. The first two versions do not use any parallel code for evaluating the fitness of the population. The GPU version performs parallelized model building, sampling and fitness evaluations. Here, we wrote problem-specific CUDA code. Like in other methods, device-specific restrictions have to be considered. For example, when using GPUs, the number of solutions that can be evaluated in parallel on each of the GPUs streaming multiprocessors (SM) is limited by the shared memory of the device. All together, the code for the GPU parallelization makes up for as little as 2% of the total application code.

From an algorithmic perspective, all three versions of RBM-EDA are identical. If we use the same pseudo random number generator to initialize the population and for the stochastic activation of neurons, they produce the same results, except for hardware-specific floating point rounding errors.

Also, the parametrization of all versions is identical. The number $m$ of hidden neurons is half of the number $n$ of visible neurons (i.e. half the problem size). Before training the model, we use the normalization scheme of [29]. We set the batch size $B$ to 1024 and use CD-1 as learning algorithm. When sampling the $k$'th individual of the new generation, we initialize $V$ to the $k$'th individual previous generation and perform 25 Gibbs sampling steps between $V$ and $H$. We choose the initial learning rate $\alpha$ to be 0.5 and a *momentum* of $m = 0.5$ [see 25]. As in [24], we automate the training process to decrease $\alpha$ and increase $m$ towards the end of the training, and stop training upon convergence.

## 4.3 Results

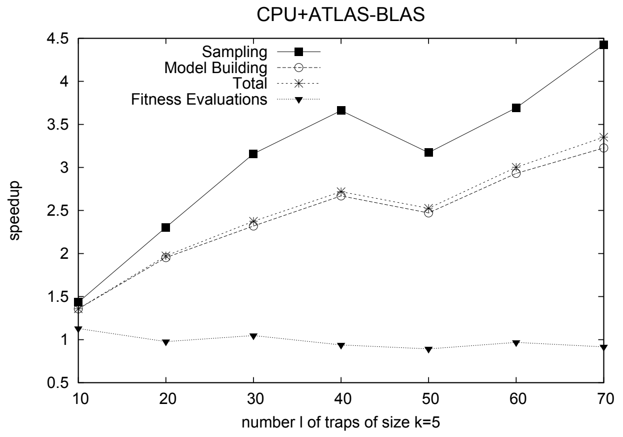
We test the different versions of RBM-EDA with concatenated deceptive traps with trap sizes of $k = 4$ and $k = 5$ for various sizes $l$. First, we report the respective speedups for model building, sampling, fitness evaluations and the total speedup when solving the respective problem to optimality. Subsequently, we analyze to what extent the parallelization can make use of the implicit parallelism of the RBM. The results for solving problems with a trap size of $k = 4$ and $k = 5$ are qualitatively equivalent. Hence, for brevity, we only report the results for $k = 5$. Initial sanity checks with other problem types such as MAXSAT also support the findings. All results are statistically significant ($p < 10^{-5}$ for BLAS results, $p < 10^{-16}$ for CUDA results).

### 4.3.1 Speedups for solving trap problems

We choose the EDA's population size such that 20 out of 20 runs find the global optimum of the respective problem.

Figure 2 (left hand side) shows the results for a trap problem with $k = 5$ and $l = 20$. That is, the length of the overall solution vector is 100. Using the ATLAS BLAS library speeds up model building, i.e. the application of the contrastive divergence training algorithm, by a factor of approx. 2.0, while still utilizing only one core. This is due to the use of the CPUs SIMD (single instruction, multiple data) units by the BLAS library.

When sampling new individuals, the ATLAS BLAS version is 2.3× faster than the nested loop version. The reason for the higher speedup is the larger matrix size: Model building is performed with a mini-batch size of 1024. Hence, the input matrix size for the calculation of Equation 2 is $1024 \times 100$ (i.e. 1024 rows holding one individual of length 100 each). During sampling, mini-batches are not necessary and all individuals are encoded in one large matrix. With larger input matrices, the underlying BLAS library is able to distribute the work to the parallel resources more efficiently.



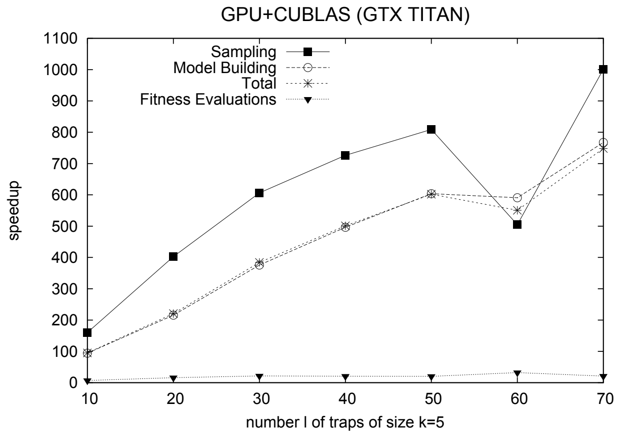


**Figure 3: Speedups for for solving concatenated trap-5 problems of various sizes $l$, compared to nested loop CPU version. Left hand side: ATLAS BLAS version; right hand side: CUBLAS version on GPU**

The CPU version and the BLAS version are identical in terms of the fitness evaluations, hence there is no speedup. As the total time is dominated by model building, the total speedup of the BLAS version is approximately 2.0.

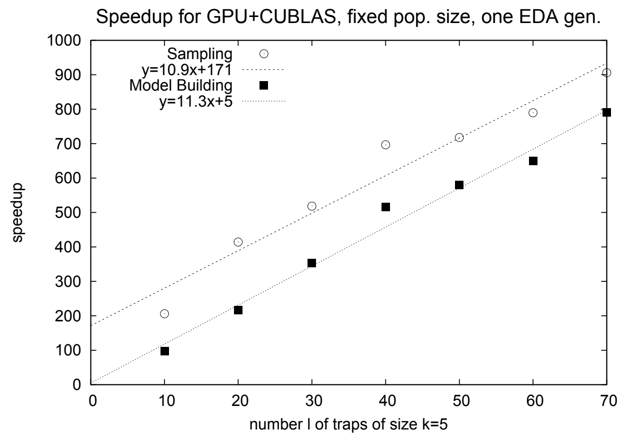
The GPU version shows a similar speedup pattern for model building and sampling, however on a much higher level. Model building and sampling are approx. 216× and 402× faster than the nested-loop version. Fitness evaluations are performed approximately 16× faster on the GPU. Compared to the model-related speedups, the gain for fitness evaluations is small. This is due to the small amount of on-chip shared memory, which limits the number of individuals, that can be processed simultaneously on the streaming multiprocessors of the GPU. However, the overall effort for fitness evaluations is low, compared to model building and sampling. Thus, a trap-5 problem of with $l = 20$ traps can be solved approximately 220× faster using a GTX TITAN GPU. Figure 2 (right hand side) shows the results for an increased problem size of $l = 70$ traps. The benefit of using the BLAS libraries increases. Due to the growing matrix sizes, the libraries utilize the parallel resources much better. Using ATLAS now yields speedups of 3.2 and 4.4 for model building and sampling, respectively. The total speedup grows to 3.4×. Using the GPU, it is possible to achieve a speedup of approximately 768 for model building and $1,001$ for sampling. The total speedup of using the GPU instead of a nested loop implementation on the CPU is approximately 748-fold.

Figure 3 summarizes the speedups for the ATLAS BLAS and GPU implementations for solving trap-5 problems of different sizes. As with other parallel implementations (e.g. [20]), the speedup grows with problem size, as the matrix sizes increase. Both the graphs for ATLAS and CUBLAS show a pronounced dent in the speedup at problem sizes of $l = 50$ and $l = 60$, respectively. Here, especially the speedup for sampling is lower than for smaller problem sizes. However, a saw tooth-like pattern is common in speedup charts for parallel matrix multiplications and most likely related to the device utilization (see e.g. [16]). On a GPU, threads working on independent subproblems are usually launched in thread blocks containing a fixed number of threads each. If there are 1024 subproblems, and the block size is 1,024, launching a single block of threads will utilize the GPU optimally. If there are 1,025 subproblems, two blocks of threads will be launched, and in one of those blocks 1,023 of 1,024 threads will be idle. The same logic applies if the number of required thread blocks does not match the number of SMs. If we launch 15 thread blocks on a GPU with 14 SMs, the 15th block will be processed on one of the SMs after the other blocks have finished. While block 15 is running, the other SMs are idle, assuming that they cannot proceed with other tasks, which is often the case if some order of execution is required.

### 4.3.2 Scalability of model building and sampling

The previous subsection reports speedups for solving trap problems with varying $l$. We now assess the scaling behavior of the model building and sampling steps in the CUBLAS version of EDA-RBM. We analyze if the CUBLAS implementation makes efficient use of the implicit parallelism of an RBM. Keep in mind that the number of independent neurons rises linearly with the *problem* size for sampling and training and, additionally, linearly with the *population* size for sampling (see Section 3.2.1). Thus, an efficient implementation should be able to distribute these independent calculations to the parallel resources without introducing significant overhead.

The number of calculations required to determine all activations within the layers $V$ and $H$ grows quadratically with the problem size $n$, as each neuron collects input from all neurons of the other layer according to (1) and (2), respectively. Hence, the run time of the CPU version is quadratic in the problem size. In theory, efficient parallelization distributes the quadratic number of steps onto a linear number of independent threads, as the number of independent neurons grows linearly. This would yield a run time that grows linearly (instead of quadratically) with the problem size.
We define speedups as $\text{runtime}_{\text{CPU}}/\text{runtime}_{\text{GPU}}$. Consequently, the speedup of an efficient parallel implementation increases linearly with the problem size. This should be true until the hardware is fully utilized. We test this hypothesis as follows. We run a single EDA generation. We train the RBM for 50 epochs with contrastive divergence. Subsequently, we sample a new generation of individuals, using 25 Gibbs sampling steps. All results are averaged over 20 independent runs.



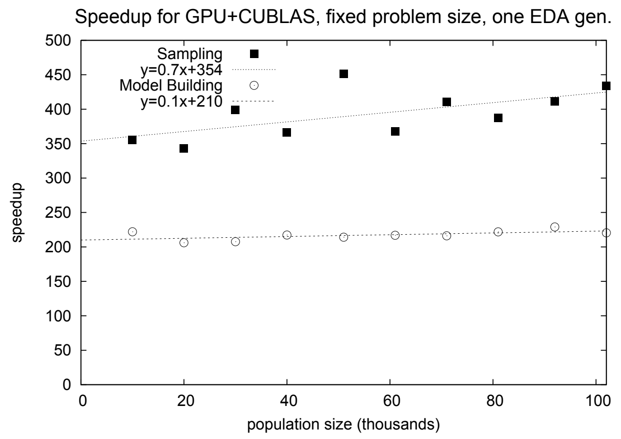


**Figure 4: Speedups for the GPU version of RBM-EDA for a single EDA generation. Left hand side: fixed population size of 25,600; right hand side: fixed problem size of 100 ($l = 20, k = 5$)**

In a first analysis we fix the population size to 25,600 and measure the speedups of the CUBLAS version compared to the single core CPU version for growing problem sizes ($l = 10, 20, ..., 70$). Figure 4, left hand side, shows the speedups for increasing problem sizes, and the corresponding approximations derived by linear regression. As predicted, the relationships for sampling and model building are about linear. For model building, the linear approximation goes almost exactly through the origin. If the input size doubles, we observe twice the speedup. In a second analysis we set the problem size to 100 ($l = 20$). Figure 4, right hand side, shows the speedups for various population sizes. The speedup for model building is unrelated to the population size for mini-batch learning. The underlying trend for sampling is linear. The vertical offset of the linear approximation is high and the familiar saw-tooth pattern is much stronger.

This indicates that the CUBLAS implementation uses the GPU resources efficiently. The speedup for the model building and sampling steps grow linearly with the problem size.

The previous results show that for *solving* trap-5 problems, the speedups for sampling already converge towards a maximum value (Figure 3, right hand side). As the problem size grows, the required population size also increases. For sampling, the utilization of the GTX TITAN is already close to the theoretical maximum. The clock speed of each GPU core is at 837 MHz, which is around 40% of the CPU clock speed of 2,100 MHz. Assuming a similar number of instructions per clock cycle, the 2,688 GPU cores should be able to process $\approx 40\% \times 2,688 = 1,070$ times as fast as a single CPU core not using its SIMD extensions. The speedups for sampling in our experiments are just slightly smaller than this upper bound. In turn, it is reasonable to assume that larger problem sizes will further increase the speedup for model building towards the upper bound.

## 5. DISCUSSION AND CONCLUSION

RBM-EDA appears to be particularly suited for parallelization. Because of the bipartite structure of the RBM most neuron-related calculations required for training and sampling are independent of each other. Hence, they can be processed in parallel. Many of these calculations can be expressed as linear algebra operations, namely matrix multiplications. They can be implemented using standardized BLAS libraries, which are available for many parallel architectures.

In practice, the parallel GPU version of RBM-EDA can use much of this theoretical potential for parallelization of RBMs. For a fixed population size, the speedups for both sampling and model building grow linearly with the problem size. This yields massive speedups when solving combinatorial optimization problems like concatenated deceptive traps. For a trap-5 problem with 70 concatenated traps, the parallel version using a NVIDIA GTX TITAN GPU is approximately 750× faster than a "plain vanilla" CPU implementation.

Furthermore, when using a BLAS library, RBM-EDA is able to utilize the parallel resources available in modern CPUs. Despite the restriction to a single CPU core, the ATLAS BLAS version can achieve significant speedups of up to 3.4× compared with the non-BLAS CPU version.

Fortunately, the large speedups come without much additional complexity of the implementation. The interface of BLAS libraries is standardized. Hence, it is easy to switch the underlying library, e.g. when porting RBM-EDA to another platform, or if a more efficient BLAS library is available. This allows us to use the library that utilizes the given hardware the best. Second, the parallelization is implicit to an RBM, i.e. the model itself is unaffected. Hence, it is not necessary to make significant changes to the code structure. Only 2% of our code is parallelization-specific. Moreover, the majority of this code is required for the parallelization of the fitness evaluations. It is not specific to the RBM.

In sum, RBM-EDA is particularly interesting for parallel architectures. The implementation effort is comparatively low, achieved speedups are large, and portability is high.

The basic principles of the parallelization could potentially be applied to other EDAs as well. Algorithms using models which factorize due to latent variables, like, for example, the growing neural gas network, have been used in EDAs and show a similar structure with implicit parallelism [12]. It may be more difficult for EDAs based on directed graphical models. This is an interesting area for future research.

## Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments on a previous version of this paper.

## References

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.

[2] K. Deb and D. E. Goldberg. *Analyzing deception in trap functions*. University of Illinois, Department of General Engineering, 1991.

[3] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *Transactions on Mathematical Software*, 16(1):1–17, Mar. 1990.

[4] S. Geman and D. Geman. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, 1984.

[5] G. Harik. Linkage Learning via Probabilistic Modeling in the ECGA. *Urbana*, 51(61):801, 1999.

[6] G. E. Hinton. Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, 14:1771–1800, 2002.

[7] G. E. Hinton, S. Osindero, and Y.-W. Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18:1527–1554, 2006.

[8] J. M. Lanza-Gutierrez, J. A. Gomez-Pulido, M. A. Vega-Rodriguez, and J. M. Sanchez-Perez. A Parallel Evolutionary Approach to Solve the Relay Node Placement Problem in Wireless Sensor Networks. In *Proceedings of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference*, GECCO'13, pages 1157–1164. ACM, 2013.

[9] P. Larrañaga, H. Karshenas, C. Bielza, and R. Santana. A Review on Probabilistic Graphical Models in Evolutionary Computation. *Journal of Heuristics*, 18(5):795–819, 2012.

[10] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Genetic Algorithms and Evolutionary Computation, 2. Kluwer Academic Pub, 2002.

[11] J. Madera, E. Alba, and A. Ochoa. A Parallel Island Model for Estimation of Distribution Algorithms. In J. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea, editors, *Towards a New Evolutionary Computation*, volume 192 of *Studies in Fuzziness and Soft Computing*, pages 159–186. Springer Berlin Heidelberg, 2006.

[12] L. Martí, J. García, A. Berlanga, and J. M. Molina. Introducing MONEDA: Scalable Multiobjective Optimization with a Neural Estimation of Distribution Algorithm. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 689–696. ACM, 2008.

[13] A. Mendiburu, J. A. Lozano, and J. Miguel-Alonso. Parallel Implementation of EDAs based on Probabilistic Graphical Models. *Evolutionary Computation, IEEE Transactions on*, 9(4):406–423, 2005.

[14] H. Mühlenbein and G. Paaß. From Recombination of Genes to the Estimation of Distributions I. Binary Parameters. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 178–187. Springer Berlin Heidelberg, 1996.

[15] A. Munawar, M. Wahib, M. Munetomo, and K. Akama. Theoretical and Empirical Analysis of a GPU-based Parallel Bayesian Optimization Algorithm. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 457–462. IEEE, 2009.

[16] NVIDIA. Speedup Charts for CUBLAS Matrix Multiplications, see https://developer.nvidia.com/cuBLAS, 2014.

[17] J. Očenášek and J. Schwarz. The Parallel Bayesian Optimization Aalgorithm. In *The State of the Art in Computational Intelligence*, pages 61–67. Springer, 2000.

[18] J. Očenášek, Jiří, E. Cantú-Paz, M. Pelikan, and J. Schwarz. Design of Parallel Estimation of Distribution Algorithms. In *Scalable Optimization via Probabilistic Modeling*, pages 187–203. Springer, 2006.

[19] J. Očenášek, Jiří and J. Schwarz. The Distributed Bayesian Optimization Algorithm for Combinatorial Optimization. *EUROGEN-Evolutionary Methods for Design, Optimisation and Control, CIMNE*, pages 115–120, 2001.

[20] J. Očenášek, Jiří, J. Schwarz, and M. Pelikan. Design of Multithreaded Estimation of Distribution Algorithms. In *Proceedings of the 2003 International Conference on Genetic and Evolutionary Computation*, GECCO'13, pages 1247–1258. Springer, 2003.

[21] M. Pelikan. Bayesian Optimization Algorithm. In *Hierarchical Bayesian Optimization Algorithm*, volume 170 of *Studies in Fuzziness and Soft Computing*, pages 31–48. Springer Berlin / Heidelberg, 2005.

[22] M. Pelikan, D. E. Goldberg, and E. Cantu-Paz. BOA: The Bayesian Optimization Algorithm. In *Genetic and Evolutionary Computation Conference*, pages 525–532, 1999.

[23] M. Pelikan and H. Mühlenbein. The Bivariate Marginal Distribution Algorithm. *Advances in Soft Computing-Engineering Design and Manufacturing*, pages 521–535, 1999.

[24] M. Probst, F. Rothlauf, and J. Grahl. Scalability of Using Restricted Boltzmann Machines to Solve Problems of Bounded Difficulty. Technical report, Information Systems and Business Administration, Universität Mainz, 2014.

[25] N. Qian. On the Momentum Term in Gradient Descent Learning Algorithms. *Neural Networks*, 12(1):145–151, 1999.

[26] S. Shakya and J. McCall. Optimization by Estimation of Distribution with DEUM Framework based on Markov Random Fields. *International Journal of Automation and Computing*, 4(3):262–272, 2007.

[27] C.-Y. Shao and T.-L. Yu. Speeding Up Model Building for ECGA on CUDA Platform. In *Proceedings of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference*, GECCO'13, pages 1197–1204. ACM, 2013.

[28] P. Smolensky. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, chapter Information Processing in Dynamical Systems: Foundations of Harmony Theory, pages 194–281. MIT Press, Cambridge, MA, USA, 1986.

[29] Y. Tang and I. Sutskever. Data Normalization in the Learning of Restricted Boltzmann Machines. Technical Report UTML-TR-11-2, Department of Computer Science, University of Toronto, 2011.