Parallel Algorithms for GPU accelerated Probabilistic Inference

Nico Piatkowski
Department of Computer Science
Artificial Intelligence Group
TU Dortmund University
Dortmund, 44227
nico.piatkowski@cs.tu-dortmund.de

Abstract

Real world data is likely to contain an inherent structure. Those structures may be represented with graphs which encode independence assumptions within the data. Performing inference in those models is nearly intractable on mobile devices or casual workstations. This work introduces and compares two approaches for accelerating the inference in graphical models by using GPUs as parallel processing units. It is empirically showed, that in order to achieve a scaleable parallel algorithm, one has to distribute the workload equally among all processing units of a GPU. We accomplished this by introducing Thread-Cooperative message computations.

1 Introduction

Data which is extracted from ubiquitous devices like smartphones, medical devices or sensor networks is likely to contain an inherent structure. Those structures may be represented with graphs which encode independence assumptions within the data. However, performing inference on those models with an complex structure on a large amount of data is computational expensive and nearly intractable on mobile devices or casual workstations. To overcome this issue, we take advantage of recent developments in computer hardware, namely programmable Graphics Processing Units (GPUs) to accelerate the inference process.

Some machine learning algorithms are already adapted for GPUs, for instance, Support Vector Machines [1, 2], frequent sets [5] or k-Means [19]. Furthermore, special purpose variants of belief propagation e.g. for stereo vision tasks are presented by several groups [3, 7, 8, 11, 17]. There, the graphical structure is always hardcoded into the algorithm which is tailored to measure for task-specific data. In case of discriminative graphical models, there is one known distributed parallel approach by Phan et al. [15] which is restricted to linear-chain structures. The GraphLab framework [12] is a general approach for distributing the computations which arise in inference tasks among several processing units or cluster nodes. Although our algorithm is currently based on proprietary data structures, it is currently ported to work within the GraphLab framework.

In this work, we investigate the parallelization of *Loopy Belief Propagation* (LBP) with an emphasis on the actual message computation. LBP corresponds to the application of ordinary Belief Propagation (BP) [14] to graphs with loops. It is a broadly used algorithm for probabilistic inference in graphical models like markov random fields (MRFs) or conditional random fields (CRFs). For the sake of generality, we will use the term LBP, even if the underlying graphical model contains no loops.

2 Parallel LBP for GPUs

A GPU may be used as a parallel processing unit beside the usual central processing unit. Here, parallel processing means the concurrent execution of several instances of one and the same procedure. To perform such computations, a GPU is composed of several so called *Multiprocessors* (MPs). Each of those MPs consists of a fixed number P of *cores* which perform the actual arithmetic or logic computations. Cores of different MPs may execute different instructions while all cores within an MP have to execute the very same instruction simultaneously. However, the operands of all cores within an MP may differ, what they usually do. Each MP is equipped with a small, low-latency *shared memory* of size S and is furthermore connected to a large high-latency *global memory*.

An instance of a concurrently executed procedure is called *thread*. The threads are partitioned into equally sized sets, called *thread-blocks* [13]. When a parallel procedure is executed, the thread-blocks are automatically assigned to an MP with free resources and the corresponding threads to this MP's cores. A GPU may manage substantially more threads than real cores are available, since their computations are used to hide global memory's latency. To achieve the maximum throughput, there always have to exist some threads which perform computations on the cores while other threads are waiting for their memory requests. Design patterns for parallel algorithms may help to utilize all cores of a GPU. *Parallel reduction* is a pattern for the generic parallelization of functions satisfying the associative property. As mentioned later, we apply the parallel reduction algorithm as proposed by Harris [9], which is known to achieve the maximum throughput on GPUs.

We will now introduce the necessary notation and the basic concept of LBP. Following [10], we focus on graphical models which are represented as factor graphs. A factor graph $G = (V, E, F)$ consists of a set of *variable nodes* V, a set of *factor nodes* F and an undirected edge set $E = F \times V$. The variable nodes are indentified with discrete random variables. Their joint realization is denoted with $\mathbf{v}$. Let $\mathbf{v}_U$ be the joint realization $\mathbf{v}$ restricted to nodes in an arbitrary set $U \subseteq V$. For notational convenience, let v describe the node v as well as the singleton set $\{v\}$. In the following, we assume that the variable nodes are partitioned into two distinct sets X and Y with domains $\mathcal{X}$ and $\mathcal{Y}$.The nodes $f \in F$ correspond to positive functions $f : \mathcal{Y}^{|\Delta(f)|} \times \mathcal{X}^{|\tilde{\Delta}(f)|} \to \mathbb{R}^+$, where $\Delta : F \to 2^Y$ assigns to a factor node it's adjacent nodes[1] in Y and $\tilde{\Delta}$ those in X, respectively. The computation of conditional marginal probabilities $p(\mathbf{y}_{\Delta(f)}|\mathbf{x})$ with LBP consists in repeatedly computing Eq. 1 and 2.

$$m_{f\to v}(\mathbf{y}_v|\mathbf{x}^{(i)}) = \sum_{\mathbf{y}'_{\Delta(f)-v} \in \mathcal{Y}^{|\Delta(f)|-1}} f(\mathbf{y}_v, \mathbf{y}'_{\Delta(f)-v}|\mathbf{x}^{(i)}) \prod_{u\in\Delta(f)-v} m_{u\to f}(\mathbf{y}'_u|\mathbf{x}^{(i)}) \tag{1}$$

$$m_{v\to f}(\mathbf{y}_v|\mathbf{x}^{(i)}) = \prod_{g\in\Delta^{-1}(v)-f} m_{g\to v}(\mathbf{y}_v|\mathbf{x}^{(i)}) \tag{2}$$

The *factor messages* (Eq. 1) have to be computed for all possible combinations of factor node $f \in F$, neighbor $v \in \Delta(f)$, realization $\mathbf{y}_v \in \mathcal{Y}$ and instance $\mathbf{x}^{(i)}, 1 \le i \le b$. Although the number of *variable messages* (Eq. 2) is similar to the number of factor messages, the time complexity $\mathcal{O}(\max_{v\in V} |\Delta^{-1}(v)|)$ for computing such a message from v to f is rather low if compared to the time complexity $\mathcal{O}(\max_{f\in F} |\Delta(f)||\mathcal{Y}|^{|\Delta(f)|-1})$ for computing a factor message from f to v. The constant b denotes the number of given realizations $\mathbf{x}^{(i)} \in \mathcal{X}^{|X|}$, i.e. the number of parallel processed training instances or the number of instances whose most probable realization of Y should be predicted. If the underlying graph contains no loops, LBP will converge after a number of iterations which only depends on the number of edges. Otherwise, the algorithm is not guaranteed to converge and is therefore terminated after a fixed number I of iterations. In both cases, the (approximated) single node marginals may be computed with $p(\mathbf{y}_v|\mathbf{x}^{(i)}) \propto m_{f\to v}(\mathbf{y}_v|\mathbf{x}^{(i)})m_{v\to f}(\mathbf{y}_v|\mathbf{x}^{(i)})$. We now present two appraoches for parallel LBP.

Data-Parallel LBP To compute the factor messages in a data-parallel manner, $|F| \times \max_{f\in F} |\Delta(f)| \times |\mathcal{Y}|$ concurrent blocks are instantiated, each with b threads. In this setup, thread i in block (f, v, y) computes message $m_{f\to v}(y|\mathbf{x}^{(i)})$. The time complexity per thread

[1] 2^U denotes the power set of a set $U \subseteq V$.

is $\mathcal{O}(\max_{f \in F} |\Delta(f)||\mathcal{Y}|^{|\Delta(f)|-1})$. Afterwards, the variable messages are computed likewise with $|V| \times \max_{v \in V} |\Delta^{-1}(v)| \times |\mathcal{Y}|$ concurrent blocks.

Thread-Cooperative LBP Considering the summation in Eq. 1, one may observe that all outgoing messages of one factor node have several terms in common. This observation leads to the Thread-Cooperative version of LBP. In this approach, one block for each pair of factor node f and given instance $\mathbf{x}^{(i)}$ is launched. The number T of threads per block is a free parameter of the algorithm. Each thread computes $|\Delta(f)| \times |\mathcal{Y}|$ partial outgoing messages which are eventually merged. To do so, let $J(f, \mathbf{x})$ be the set $\{j(f, \mathbf{x}, \mathbf{y}) = f(\mathbf{y}|\mathbf{x}) \prod_{v \in \Delta(f)} m_{v \to f}(\mathbf{y}_v|\mathbf{x}) : \forall \mathbf{y} \in \mathcal{Y}^{|\Delta(f)|}\}$. A thread $t, 1 \leq t \leq T$ in block $(f, \mathbf{x}^{(i)})$ computes $|J(f, \mathbf{x}^{(i)})|/T$ elements of $J(f, \mathbf{x}^{(i)})$. Once such an $j(f, \mathbf{x}^{(i)}, \mathbf{y})$ is computed, the partial outgoing messages of thread t may be updated with Eq. 3.

$$m_{f \to v}^{(t+1)}(\mathbf{y}_v|\mathbf{x}^{(i)}) \quad = \quad m_{f \to v}^{(t)}(\mathbf{y}_v|\mathbf{x}^{(i)}) + \frac{j(f, \mathbf{x}^{(i)}, \mathbf{y})}{m_{v \to f}^{(t)}(\mathbf{y}_v|\mathbf{x}^{(i)})}, \forall v \in \Delta(f) \tag{3}$$

When all $|J(f, \mathbf{x}^{(i)})|$ have been processed cooperatively by all threads in a block, the partial messages are merged by parallel reduction. This results in a time complexity of $\mathcal{O}(\max_{f \in F} |\Delta(f)||\mathcal{Y}|^{|\Delta(f)|} T^{-1} + |\Delta(f)||\mathcal{Y}| \log T)$ per thread. Here, the left term is an outcome of computing the t-th subset of $J(f, \mathbf{x}^{(i)})$ and the right term is introduced by the final reduction and may be intrepreted as parallelization overhead. Finally, one may recognize the possibility to launch $T \times R$ threads per block instead of T, in order to perform the updates in Eq. 3 concurrently for all neighbors with R threads. This introduces the factor R^{-1} to the left term of the time complexity. The just described algorithm is called *Thread-Cooperative LBP*.

The optimal values for T and R, in terms of runtime, heavily depend on the actual GPU and the actual factor graph. They may be obtained by a grid search, constrained on solutions which satisfy $((T \cdot R) \mod P) = 0$ and $B \cdot T \cdot \max_{f \in F} |\Delta(f)| \cdot |\mathcal{Y}| \leq S$, where B is the desired number of concurrently scheduled blocks per MP. The first constraint arises from the fact, that all processing units within one MP have to be used. The second constraint ensures, that the partial messages will fit into an MP's shared memory.

3 Evaluation

We now give a runtime comparison of the LBP variants described above. For the first comparsion, we used the Thread-Cooperative Forward-Backward (FB) algorithm as well, which is a specialized LBP variant for Linear-Chain structures. The messages are propagated sequentially from the left-most to the right-most node in the graph and back, which reduces the number of blocks from $|F| \times b$ to b. We used our own C++/CUDA-C code for this[2]. The GPU which was used in the following experiments is an NVIDIA GeForce GTX580 consumer GPU. The repetition of some experiments on an NVIDIA Tesla C2050 GPU, which is designed for HPC, yields similar or slightly worse results.

With the exception of FB which is exact, we approximated the marginals with $I = \sqrt{|V|}$ iterations. Thread-Cooperative LBP was configured with $T = 16$ and $R = 2$. The values where determined heuristically.

Figure 1 shows the results on the whole CoNLL-2000 data set [18] on the left and the runtime on single instances of Chen Yanover's PROTEIN data set [20] on the right. The Data-Parallel algorithm outperforms the Thread-Cooperative approach when the number of instances per batch is greater than 8. The reason is that the Thread-Cooperative approach uses slightly more resources in terms of registers and shared memory which reduces the number of concurrently scheduled blocks per MP. Nevertheless, the Thread-Cooperative approach clearly outperforms the plain Data-Parallel approach on models with a high number of factors, as expected from the asymptotically lower time complexity. As the right plot shows, the difference in runtime becomes higher as the graphical complexity grows. Although we used $b = 1$ in this evaluation, the second plot will look similar for

[2]The code is available for download at `http://sfb876.tu-dortmund.de/crfgpu`

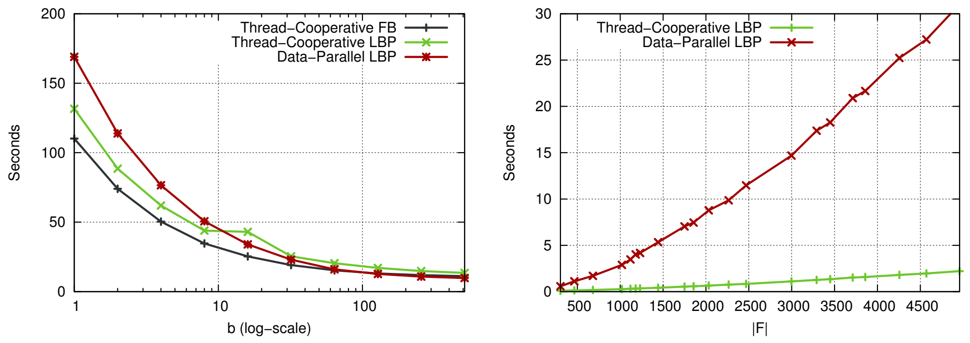

Figure 1: The left plot shows how the runtime evolves on a fixed $|F|$ and increasing b. The right plot shows how runtime evolves for fixed $b = 1$ and increasing $|F|$.

different fixed values of b. This example with $b = 1$ should show, that, in contrast to the Thread-Cooperative approach, the Data-Parallel LBP is not able to distribute the workload of one single instance among all cores of a GPU. Future experiments will involve synthetic data sets to evaluate how these algorithms perform when either $|\mathcal{Y}|$ or $\max_{f \in F} |\Delta(f)|$ is raised.

As we focus on the actual message computations, we used a plain full parallel (flooding) schedule in the above runtime evaluation. Recently, Gonzalez et al. developed the ResidualSplash (RS) message propagation algorithm [6] for parallel inference in graphical models, which is known to be optimal in terms of the number of message computations. The combination of Thread-Cooperative message computation and RS or other sophisticated schedulings like residual belief propagation [4] is ongoing work.

To finally compare our approach to sequential variants of LBP, we propagated the same number of messages with the ResidualSplash implementation[3] by Gonzales et al. and our algorithm and measured the runtime. To keep things simple, we used the round-robin scheduler and fixed the number of processed nodes. We ran both approaches on the protein `1mv8` which consists of $|V| = 436$ variable nodes, $|F| = 4476$ factors and $|E| = 12992$ edges. The algorithms propagated asymptotically $\mathcal{O}(\max_{f \in F} |\Delta(f)||F||\mathcal{Y}| + \max_{v \in V} |\Delta^{-1}(v)||V||\mathcal{Y}|)$ messages. Since we used a pairwise model, the value of $\max_{f \in F} |\Delta(f)|$ was bound by two. The number of labels was 31 for this protein. It turned out, that our algorithm propagated the messages ≈ 6 times faster than the CPU implementation.

Our comparison of different parallel algorithms for the computation of LBP messages showed empirically, that the performance heavily relies on the actual structure of the processed data. As a result, current approaches for implicitly parallel domain-specific languages for machine learning like OptiML [16] should consider an explicit description of the input data in order to tackle either highly data-parallel tasks with a high number of training instances or tasks where the high complexity of single instances is the challenging part. Future work will include a comparison with auto-generated OptiML code.

A comparison on a larger set of proteins, different data and other message passing schedules will be done in future. It is also planned to integrate our approach into GraphLab [12] to have a broader platform for comparison with other inference methods and to benefit from the several already implemented scheduling algorithms.

Acknowledgments

This work has been supported by the DFG, Collaborative Research Center SFB876, project A1.

[3]The code is available for download at `http://select.cs.cmu.edu/code`

References

[1] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris. GPU acceleration for support vector machines. In *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*, 2011.

[2] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *Procs. of the 25th international conference on Machine learning*, pages 104–111. ACM, 2008.

[3] Cheng Chao-Chung, Li Chung-Te, Liang Chia-Kai, Lai Yen-Chieh, and Chen Liang-Gee. *Architecture design of stereo matching using belief propagation*, pages 4109–4112. 2010.

[4] G. Elidan, I. Mcgraw, and D. Koller. Residual Belief Propagation: Informed Scheduling for Asynchronous Message Passing. In *Proceedings of the Twenty-second Conference on Uncertainty in AI (UAI)*, Boston, Massachussetts, 2006.

[5] Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He, and Qiong Luo. Frequent itemset mining on graphics processors. In *DaMoN '09: Procs. of the Fifth International Workshop on Data Management on New Hardware*, pages 34–42, New York, NY, USA, 2009. ACM.

[6] Joseph E. Gonzalez, Yucheng Low, and Carlos Guestrin. Residual splash for optimally parallelizing belief propagation. In *Arti. Intell. and Stat. (AISTATS*, pages 177–184, 2009.

[7] S. Grauer-Gray and C. Kambhamettu. Hierarchical belief propagation to reduce search space using cuda for stereo and motion estimation. In *Applications of Computer Vision (WACV), 2009 Workshop on*, pages 1–8, 2009.

[8] S. Grauer-Gray, C. Kambhamettu, and K. Palaniappan. Gpu implementation of belief propagation using cuda for cloud tracking and reconstruction. In *Pattern Recognition in Remote Sensing (PRRS 2008), 2008 IAPR Workshop on*, pages 1–4, 2008.

[9] Mark Harris. Optimizing Parallel Reduction in CUDA. NVIDIA Corporation, 2008.

[10] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. on Infor. Theory*, 47(2):498–519, 2001.

[11] Yen-Chieh Lai, Chao-Chung Cheng, Chia-Kai Liang, and Liang-Gee Chen. Efficient message reduction algorithm for stereo matching using belief propagation. *Img.Pr.*, 1(2):2977–2980, 2010.

[12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Arti. Intell. (UAI)*, California, July 2010.

[13] NVIDIA Corporation. *CUDA Programming Guide 4.0.* June 2011.

[14] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[15] Xuan-Hieu Phan, Le-Minh Nguyen, Yasushi Inoguchi, and Susumu Horiguchi. High-performance training of conditional random fields for large-scale applications of labeling sequence data. *IEICE - Trans. Inf. Syst.*, E90-D:13–21, January 2007.

[16] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 609–616, New York, NY, USA, June 2011. ACM.

[17] Jian Sun, Nan-Ning Zheng, and Senior Member. Stereo matching using belief propagation. *IEEE Transactions on Pattern Analysis and Machine Intell.*, 25(7):787–800, 2003.

[18] Erik F. Tjong Kim Sang and Sabine Buchholz. Introduction to the CoNLL-2000 shared task: chunking. NJ, USA, 2000. Assoc. for Comp. Linguistics.

[19] Han Xiao. Towards Parallel and Distributed Computing in Large-Scale Data Mining: A Survey. Technical report, Technical University of Munich, Germany, 2010.

[20] Chen Yanover, Ora Schueler-Furman, and Yair Weiss. Minimizing and learning energy functions for Side-Chain prediction. Berlin, 2007. Springer.