CCMpred — Supplementary Information

Stefan Seemayer, Markus Gruber and Johannes Söding

1 Markov Random Field Model

The following section will outline the mathematical model in our contact prediction method that is essentially identical to the plmDCA (Ekeberg *et al.*, 2013) and GREMLIN (Kamisetty *et al.*, 2013) methods.

We eliminate transitive interactions in the observed interaction network by learning a generative model of the MSA using a Markov Random Field (MRF). Assuming we have an MSA x_i^n (n sequence index, i position index) with L columns and N sequences, we represent columns in the MSA as the vertices with single-residue emission potentials $\varepsilon_i(a)$ (with i being a column index and $a \in \{1..20\}$ representing the twenty possible amino acids) and covariation between columns as the edges with pairwise emission potentials $\varepsilon_{i,j}(a,b)$ (with i and j column indices, a and b amino acid indices). The network of co-evolution can then be learned by maximizing the likelihood of observing the sequences in the input MSA, given the model parameters $\boldsymbol{\varepsilon}$:

$$\mathcal{L}(\boldsymbol{\varepsilon}) = \frac{1}{Z} \prod_{n=1}^{N} \prod_{i=1}^{L} \left[\exp \left(\varepsilon_i(x_i^n) + \sum_{\substack{j=1 \\ j \neq i}}^{L} \varepsilon_{i,j}(x_i^n, x_j^n) \right) \right] \tag{1}$$

In order for the underlying probability distribution to be well-defined, a normalization constant Z is required which represents a sum of the unnormalized probabilities over all 21^L possible assignments for $(x_1^n, \ldots, x_L^n)$ (20 for amino acids, 1 for gaps). The runtime exponentially growing makes the computation of $\mathcal{L}(\boldsymbol{\varepsilon})$ prohibitive for relevant values of L. We instead optimize a *pseudo-likelihood*:

$$\begin{aligned}
\mathrm{pll}(\boldsymbol{\varepsilon}|\boldsymbol{X}) &= \log \prod_{n=1}^{N} \prod_{i=1}^{L} p(X_i = x_i^n | (x_1^n, \ldots, x_{i-1}^n, x_{i+1}^n, \ldots x_L^n, \boldsymbol{\varepsilon})) \\
&= \sum_{n=1}^{N} \sum_{i=1}^{L} \log \frac{\exp \left[\varepsilon_i(x_i^n) + \sum_{\substack{j=1 \\ j \neq i}}^{L} \varepsilon_{i,j}(x_i^n, x_j^n) \right]}{\sum_{c=1}^{21} \exp \left[\varepsilon_i(c) + \sum_{\substack{j=1 \\ j \neq i}}^{L} \varepsilon_{i,j}(c, x_j^n) \right]} \\
&= \sum_{n=1}^{N} \sum_{i=1}^{L} \left[\varepsilon_i(x_i^n) + \sum_{\substack{j=1 \\ j \neq i}}^{L} \varepsilon_{i,j}(x_i^n, x_j^n) - \log Z_i^n \right]
\end{aligned} \tag{2}$$

$$Z_i^n = \sum_{c=1}^{20} \exp \left[\varepsilon_i(c) + \sum_{\substack{j=1 \\ j \neq i}}^{L} \varepsilon_{i,j}(c, x_j^n) \right] \tag{3}$$

Our new pseudo-likelihood objective function has the great advantage that the normalization constants Z_i^n appearing in it involve only a sum over L terms and can therefore be computed easily.

The gradient of this pseudo-log-likelihood has components:

$$\frac{\partial pll(\boldsymbol{\varepsilon}|\boldsymbol{X})}{\partial \varepsilon_{i,j}(a,b)} = \sum_{n=1}^{N} \left\{ I\left(x_j^n = b\right) \cdot \left(I\left(x_i^n = a\right) - \frac{\exp\left[\varepsilon_i\left(a\right) + \sum\limits_{\substack{k=1\\k\neq i}}^{L} \varepsilon_{i,k}\left(a, x_k^n\right)\right]}{\sum\limits_{c=1}^{20} \exp\left(\varepsilon_i\left(c\right) + \sum\limits_{\substack{k=1\\k\neq i}}^{L} \varepsilon_{i,k}\left(c, x_k^n\right)\right)} \right) \right\}$$

$$= \sum_{n=1}^{N} \left[I(x_j^n = b) \left[I(x_i^n = a) - p(X_i = a | (x_1^n, \ldots, x_{i-1}^n, x_{i+1}^n, \ldots x_L^n, \boldsymbol{\varepsilon}) \right] \right] \tag{4}$$

In order to favor sparse solutions, we additionally add an L_2 regularization term $R(\boldsymbol{\varepsilon})$ and maximize $pll(\boldsymbol{\varepsilon}|\boldsymbol{X}) - R(\boldsymbol{\varepsilon})$ using the nonlinear conjugate gradient method:

$$R(\boldsymbol{\varepsilon}) = \lambda_{\text{single}} \sum_{i=1}^{L} \|\boldsymbol{\varepsilon}_i\|_2^2 + \lambda_{\text{pair}} \sum_{\substack{i,j=1\\j\neq i}}^{L} \|\boldsymbol{\varepsilon}_{i,j}\|_2^2 \tag{5}$$

The regularization coefficients $\lambda_{\text{single}} = 1, \lambda_{\text{pair}} = 0.2 \times (L-1)$ were set as in the GREMLIN method.

1.1 Post-processing

After a successful optimization, the couplings between residue positions S_{ij} are ranked by the Frobenius norms of the edge potentials $\boldsymbol{\varepsilon}_{i,j}$:

$$S_{ij} = \sqrt{\sum_{a,b=1}^{20} \varepsilon_{i,j}(a,b)^2} \tag{6}$$

Like plmDCA (Ekeberg *et al.*, 2013), GREMLIN (Kamisetty *et al.*, 2013) and PSICOV (Jones *et al.*, 2012), we apply the Average Product Correction (Dunn *et al.*, 2008) to arrive at the final score C_{ij}:

$$C_{ij} = S_{ij} - \frac{S_{i,\cdot}\ S_{\cdot,j}}{S_{\cdot,\cdot}} \tag{7}$$

where "$\cdot$" denotes averaging over the respective column/row. $S_{\cdot,\cdot}$ is the average over all elements in the matrix.

2 GPGPU Architecture

Graphics processing units (GPUs) were designed for executing small programs determining the color value of each pixel in modern video games and are thus optimized for performing massive amounts of computation in parallel where similar operations are performed over large arrays of data. Since the release of the NVIDIA Compute Unified Device Architecture (CUDA) and CUDA C in 2006, accessing the compute power of GPUs has become increasingly convenient, making GPUs a competitor to traditional CPU-based systems in appropriate scenarios.

Current NVIDIA GPUs hold many thousand cores grouped together in *streaming multiprocessors* (SMs). For example, the NVIDIA GeForce GTX 780 Ti we used for benchmarks in this paper holds 15 SMs with 192 cores each, resulting in 2880 CUDA cores, but future hardware generations will scale by adding more SMs or altering the number of cores per SM. While the number of cores is much higher than the number of cores in typical CPUs, the individual CUDA cores have slower clock speeds and cannot run completely separately. Instead, execution of programs called *kernels* is grouped in *warps* of 32 *threads* that execute the same instruction in lockstep. Warps are grouped into *blocks* where a block will always be assigned to a

single SM. The whole scheme of organizing threads in blocks is called the *grid* and the user can define the size of blocks and grids within certain architecture-defined limits. In order to ensure scalability when future hardware generations add or change SMs, the actual scheduling cannot be affected by the user and will be performed by the GPU.

Another difference between CPU and GPU is the memory hierarchy. The memories of the GPU differ mainly in size and speed and therefore their application. Global memory is comparable to the CPU's main memory. It is faster than CPU memory but by far the slowest of the GPU memories while being relatively big (several gigabytes). Data on the global memory can be accessed by the CPU as well as any SM of the GPU. Therefore it is usually utilised for data transfer between CPU and GPU and data which has to be shared between multiple SMs. Performance of global memory depends heavily on the kernel's access patterns. Maximum bandwidth can only be achieved if the memory operations of one (half-) warp are coalesced, i.e. sequentially accessing complete 128-byte-aligned memory blocks.

Contrary to global memory, shared memory is unique per SM. It can only be accessed by threads belonging to the same block. With a maximum of 48 kilobytes of shared memory per block on current GPUs, it is much smaller than global memory but also much faster. Performance of using shared memory also depends on access patterns, albeit much less so than for global memory. Shared memory is typically used for inter-thread communication and fast reordering of global memory.

The fastest available level of memory are the SM-local registers. Although they are designed for local storage within a thread without communication, the new Kepler architecture has introduced shuffle instructions that allow register exchanges between threads in the same warp. While not currently used for CCMpred, the constant and texture memories are available as additional subsets of the global memory optimized for high-bandwidth read-only access in certain access patterns.

3 Parallelization

Looking at the expressions for calculating the objective function and gradient values, it becomes apparent that certain terms appear at multiple locations and should be precomputed. By first gathering potentials for each of the nodes and then distributing those summed potentials appropriately, memory access can occur mostly independently.

In the following, we will describe the two most important computation steps, namely the pre-computation of summed potentials and the computation of the edge gradients $\partial pll(\boldsymbol{\varepsilon}|\boldsymbol{X}) / \partial \varepsilon_{st}(a,b)$, together with how to map their computation to a parallel algorithm on the GPU. While other computations have to be performed as well, the vast majority of the computation time is spent on these two kernels. Table S2 gives an overview of all kernels in CCMpred.

3.1 Decomposition of Likelihood and Gradients

Certain terms in equations (2), (3) and (4) appear at multiple locations in the expressions so it makes sense to precompute these values before actual computation of the likelihood and gradient values to improve performance. We start out by defining $\mathrm{sumPot}(n,a,i)$ as the sum of potentials around position i with amino acid a at that position and all other amino acids as in sequence n:

$$\mathrm{sumPot}(n,a,i) = \varepsilon_i(a) + \sum_{\substack{j=1 \\ j \neq i}}^{L} \varepsilon_{i,j}(a, x_j^n) \tag{8}$$

With these values, the partition function $\mathrm{Z}(n,i)$ for sequence n at position i can be computed:

$$\mathrm{Z}(n,i) = \sum_{a=1}^{21} \exp\left[\varepsilon_i(a) + \sum_{\substack{j=1 \\ j \neq i}}^{L} \varepsilon_{i,j}(a, x_j^n)\right] = \sum_{a=1}^{21} \exp\left[\mathrm{sumPot}(n,a,i)\right] \tag{9}$$

Finally, we combine $\mathrm{sumPot}(n,a,i)$ and $\mathrm{Z}(n,i)$ to compute normalized probabilities $\mathrm{prob}(n,a,i)$ for observing amino acid a at position i in sequence n:

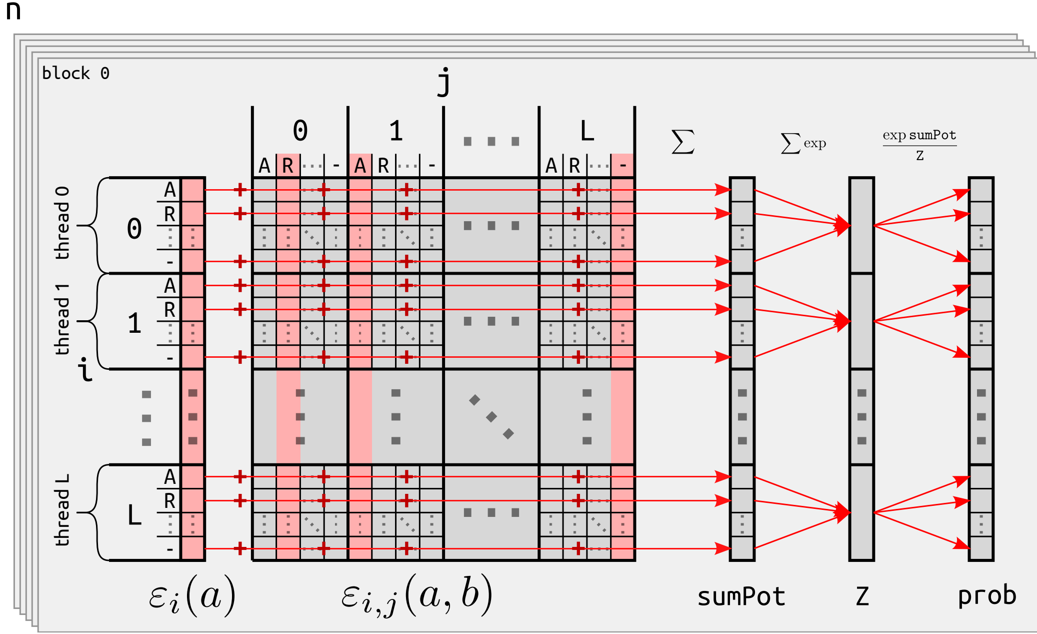

Figure S1: Pre-computation of reused values. Every sequence n in the MSA corresponds to one selection of columns $x_j^n = b$ in the $\varepsilon_{i,j}(a,b)$ matrix. We compute $\text{sumPot}(n,a,i)$ values for all n, i and a by summing over single-node emission potentials $\varepsilon_i(a)$ and selected pairwise emission potentials $\varepsilon_{i,j}(a, x_j^n)$ and then use those $\text{sumPot}(n,a,i)$ values to compute $\text{Z}(n,i)$ and $\text{prob}(n,a,i)$. We parallelize by launching one block for every n and one thread for every i, thus there are no write conflicts to sumPot, Z or prob.

$$
\begin{aligned}
\text{prob}(n,a,i) &= p(X_i = a | (x_1^n, \ldots, x_{i-1}^n, x_{i+1}^n, \ldots x_L^n, \boldsymbol{\varepsilon}) \\
&= \frac{\exp\left[\varepsilon_i(a) + \sum_{\substack{j=1 \\ j\neq i}}^{L} \varepsilon_{i,j}(a, x_j^n)\right]}{\sum_{c=1}^{21} \exp\left[\varepsilon_i(c) + \sum_{\substack{j=1 \\ j\neq i}}^{L} \varepsilon_{i,j}(c, x_j^n)\right]} \\
&= \frac{\exp\left[\text{sumPot}(n,a,i)\right]}{\text{Z}(n,i)}
\end{aligned} \tag{10}
$$

Especially in the case of prob, the precomputation is advantageous since each of the values will be used L times to compute equation (4).

3.2 Calculation of Precomputed Values

All above values are precomputed by a single kernel, `d_compute_pc` as illustrated in Figure S1. We parallelize by launching a thread block per sequence n (i.e. N blocks total) and one thread per column i of the MSA (i.e. L threads per block). Each thread computes the sumPot, Z and prob values for its block's sequence and a unique column for all amino acids, i.e. $21 + 1 + 21 = 43$ values. By this partitioning, every thread will write to different memory ranges so no additional synchronization is needed.

Since each column of the sequence has to be accessed exactly L times (once by each thread), we store sequence information in shared memory. This significantly reduces the number of global memory transactions because instead of $L \times L$ global memory accesses we now have only L loads from global and $L \times L$ loads from shared memory. Using shared memory is also advantageous since every thread of a warp needs to access the same column at the same time and broadcasting a value is much more efficient from shared than from global memory.

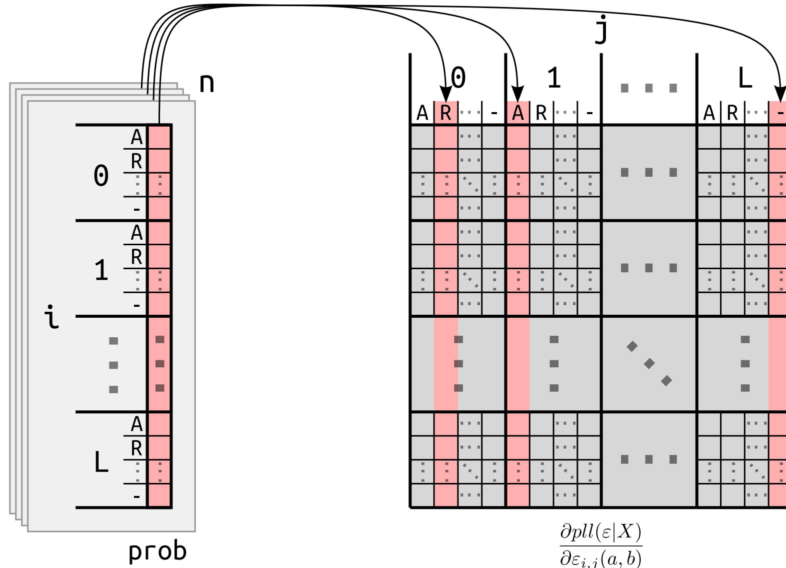

Figure S2: Computation of gradients. The prob values precomputed in the last step are added onto the gradient columns selected by the sequence in the MSA. Additionally, pre-computed pair counts are added to all gradient values (not shown here). We parallelize by launching one block per column j and one thread per unique combination of (i, a). Every thread additionally loops over all n to compute the full gradient for one cell in the gradient matrix.

We also reduced the amount of variables kept in memory. First, every thread computes $\text{sumPot}(n, a, i)$ for all 21 amino acids a. These are stored in a thread local array. In the second step, every thread computes $\text{Z}(n, i)$ as the sum of its 21 local values and stores it directly to global memory. Last, the thread-local sumPot and Z values can be used to compute the 21 different prob values which are then also stored in global memory. Finally, only the local sumPot value which has the same amino acid a as the sequence n at position i is stored to global memory because only this value will be needed for further computations.

3.3 Calculation of Edge Gradients

After pre-computing the required values, CCMpred proceeds with calculating the edge gradients as illustrated in Figure S2. Looking at equation (4), we see that the gradient computation can be split into a part dependent of the current $\boldsymbol{\varepsilon}$ and a part that only depends on the sequence data:

$$\frac{\partial pll(\boldsymbol{\varepsilon}|\boldsymbol{X})}{\partial \varepsilon_{i,j}(a,b)} = \sum_{n=1}^{N} \left\{ I\left(x_j^n = b\right) \cdot I\left(x_i^n = a\right) \right\} - \sum_{n=1}^{N} \left\{ I\left(x_j^n = b\right) \cdot \text{prob}(n, a, i) \right\} \tag{11}$$

Instead of summing over indicator functions in every iteration, we pre-compute these counts once and use them to initialize the gradients in every iteration, summing only over the changing prob values.

We parallelize by assigning a unique column j to every block and launch threads for each combination of a column i and an amino acid a. Hence, each block will compute $21 \times 21 \times L$ different gradients for all i, a and b and one fixed j.

Within each thread, we loop over all sequences n. In every iteration, b is set to the amino acid at position j in the current sequence. We load $\text{prob}(n, a, i)$ and then add it onto the gradient values. This loop corresponds to the second sum in Equation (11). Since i and a are unique per thread, j is unique per block, and b depends on j, there are no conflicting memory accesses. Furthermore, all memory accesses are coalesced because consecutive threads have consecutive combinations of i and a.

3.4 Maintaining Symmetry

In order to have favorable memory access to all pairwise emission potentials $\varepsilon_{i,j}(a,b)$, it is important that they are stored in memory in linear order. Although emission potentials are symmetric (i.e. $\varepsilon_{i,j}(a,b) = \varepsilon_{j,i}(b,a)$) and it would only be necessary to store one triangle of the symmetric matrix, we can achieve better performance by having two copies of every variable for efficiently accessing them both in horizontal and vertical stripes of the matrix. In order for the numerical optimization to be well-behaved, we therefore require a re-symmetrization step where the value of every gradient is added onto its symmetric counterpart:

$$\varepsilon_{i,j}(a,b) = \varepsilon_{i,j}(a,b) + \varepsilon_{j,i}(b,a) \tag{12}$$

This operation is achieved using a standard matrix transpose kernel that adds both values onto another instead of switching them.

3.5 Padding

Another technique for ensuring favorable memory access patterns on the GPU is to align variables in global memory to 128-byte boundaries so that 32 float variables can be accessed by the 32 threads of a warp in a single coalesced read, thus maximizing memory throughput. An easy way to ensure the 128-byte alignment in our model is to use a 32-amino-acid indexing scheme for some of the amino acid indices in our $\varepsilon_{i,j}$ values, leaving the potentials and gradients for $a \in \{21..32\}$ zero so they do not affect the model.

While somewhat less memory efficient (see section 4 for details) than a non-padded version, we observe that the padded GPU implementation is 1.37 times faster than our non-padded implementation for typical input alignment dimensions and thus use the padded indexing scheme by default. Users who want to increase the maximal size of the model computable on their graphics hardware can choose to re-compile CCMpred with padding disabled.

4 Memory Considerations

The amount of available GPU RAM limits the maximal model size that can be computed on the GPU. The $L^2 \times 21^2 + L \times 20$ variables of the model and gradients need to be stored on the GPU, together with $N \times L$ sumPot values, $N \times L$ Z values and $N \times L \times 21$ prob values. Since we also perform the optimization through nonlinear conjugate gradients on the GPU and we have to store some additional intermediate values, a total of $4 \times L^2 + 2 \times 20 \times L + 23 \times N \times L$ numbers need to be stored. We use a 32-bit floating point representation since we found the numerical accuracy to be sufficient. Adding these numbers up, we arrive at the approximate maximum model sizes for different GPU RAM sizes (using $N = 5000$) shown in Table S1. As an example, the NVIDIA GeForce GTX 780 Ti, a typical consumer-grade graphics card with 3 GBs of GPU RAM using our fast padded implementation will be able to compute models with up to 519 residues, a model size that already covers more than 98% of domains in SCOP (Figure S3 shows a histogram of SCOP domain lengths). The NVIDIA Tesla K40, the currently available GPU with the most RAM will be able to compute models for up to 1059 residues, covering 99.9% of SCOP. For models larger than what fits onto the available graphics card, users can fall back to our CPU implementation since non-GPU RAM is generally abundant.

5 Benchmarking Details

All benchmarks were run on the same computer with dual Intel Xeon E5–2620 six-core processors, 48 GBs of RAM and a NVIDIA GeForce GTX 780 Ti running Scientific Linux version 6.4. We measure the total (wall clock) time expired for each of the programs by running them with the UNIX 'time' command. All programs that supported multi-threaded computation were started with 6 threads to simulate the full usage of a single six-core processor. To demonstrate that CCMpred can also scale well to large MSAs, we run models that exceed the RAM on the NVIDIA GeForce GTX 780 Ti on an NVIDIA Tesla K40 which is somewhat slower

GPU RAM	$L_{\max}$	% SCOP	$L_{\max}^{pad}$	% SCOP
1 GB	353	90.4	291	81.8
2 GB	512	98.2	420	95.2
3 GB	635	99.4	519	98.4
5 GB	829	99.8	676	99.5
6 GB	911	99.9	743	99.7
8 GB	1057	99.9	861	99.9
12 GB	1302	99.9	1059	99.9

Table S1: Approximate maximal model sizes for different amounts of available GPU memory and percentage of domains in SCOP (Andreeva *et al.*, 2008) covered by that maximal model size. The padded version has a faster run time at the cost of a slightly higher memory requirement.

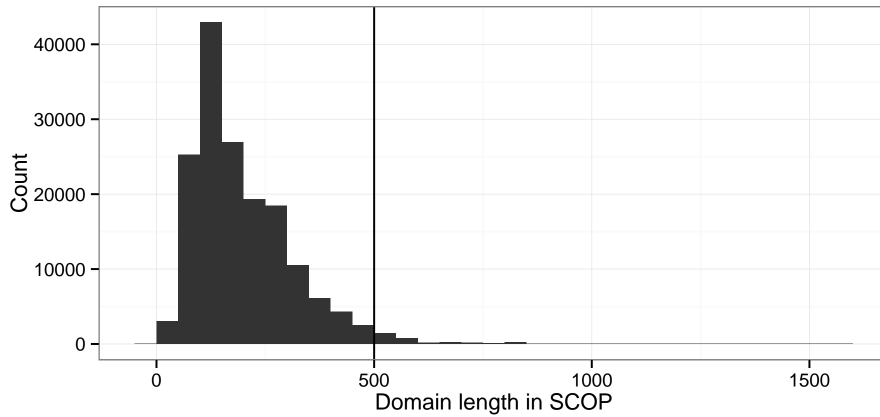

Figure S3: Domain length distribution in SCOP (Andreeva *et al.*, 2008). More than 98% of domains have a length smaller than 500.

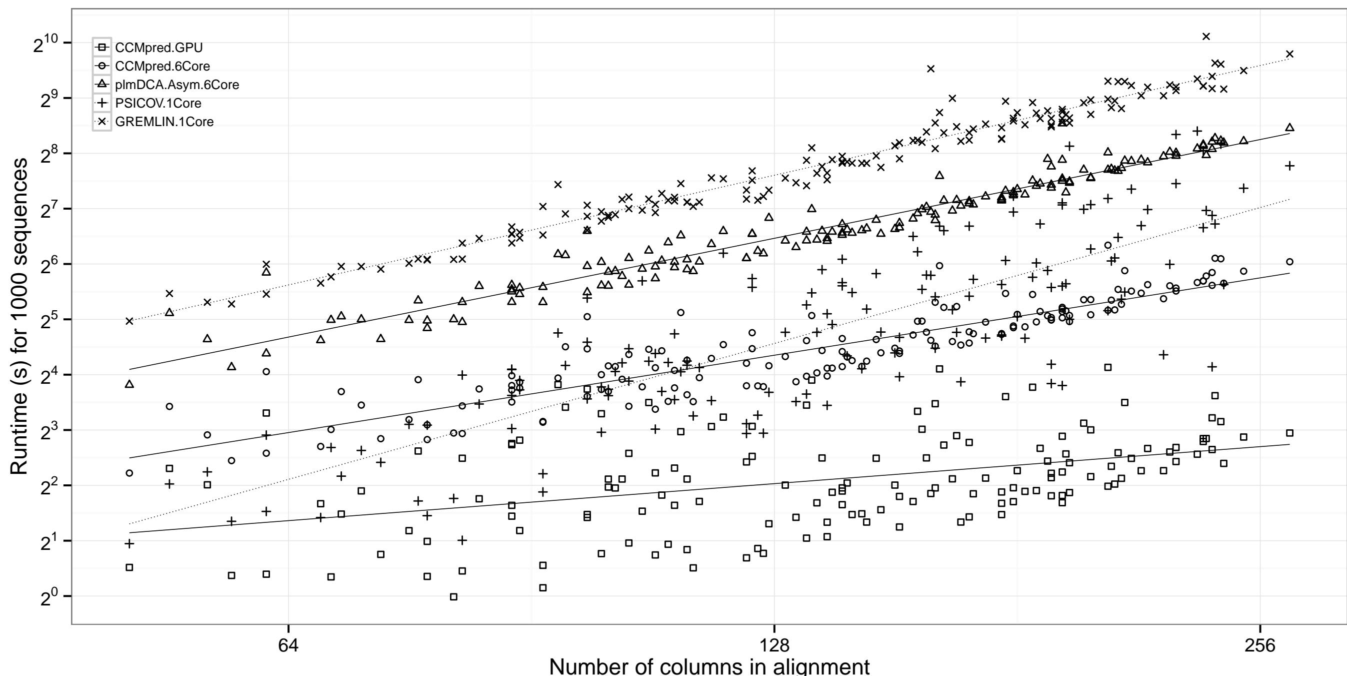

Figure S4: Runtimes on real multiple sequence alignments from the PSICOV data set. Since runtimes are linearly dependent on the number of input sequences, we normalize the runtime to $N = 1000$ sequences.

for single-precision float computations since it uses ECC memory. The corresponding points in the runtime plot of the main paper are shown greyed out.

All MATLAB code (GREMLIN and plmDCA) was compiled using MATLAB R2012b and the 'mex' compiler, PSICOV was compiled with GCC 4.7.2 and CCMpred was compiled with GCC 4.7.2 and the NVIDIA CUDA SDK Version 5.0

6 Benchmark on Biological Multiple Sequence Alignments

While real-world multiple sequence alignments show the similar overall speedups as synthetically generated alignments, their runtimes exhibit more variation due to various effects changing the number of necessary line search evaluations until the next step can be found during optimization. Figure S4 nonetheless shows the runtimes for the 150 protein families in the PSICOV data set. Table S3 gives the sums of runtimes for all families for each method. CCMpred.GPU gets a speedup of 37.8x compared to the next-fastest method plmDCA.Asymmetric, confirming our results on synthetic alignments.

7 Utility of Fast Contact Prediction

The ultimate goal for residue-residue contact prediction methods would be to predict structural constraints for all known protein families where enough sequence information is present. We estimate that roughly a third of all families in the Pfam database (Finn *et al.*, 2014) can already be predicted using a reasonable sequence coverage threshold of $N/L \geq 3.0$ (see Figure S5). Through the rapid development of faster and cheaper DNA sequencing methods, we expect the sequence coverage of many families to increase rapidly and improvements in contact prediction methods will likely reduce the sequence coverage threshold over which predictions of reasonable accuracy can be made. Since contact prediction is therefore of relevance for more and more families, it is interesting to estimate how much processing time a prediction of all sufficiently-covered protein families would take. Since we know that the runtimes of MRF-based contact prediction methods scale in $O(NL^2)$, we estimate the expected runtimes for every protein family in Pfam using a

Time(%)	Time	Calls	Avg	Name	Shared memory per block (bytes)
61.85	14.67s	48	305.64ms	d_compute_edge_gradients	N (max 6144)
29.26	6.94s	48	144.59ms	d_compute_pc	$L \times 4$
1.87	442.66ms	208	2.13ms	vecdot_inter	$nthreads \times 4$
1.83	432.97ms	1	432.97ms	d_edge_gradients_histogram_weighted	$nthreads \times 21 \times 4$
1.07	252.70ms	1	252.70ms	d_initialize_w	-
0.88	209.71ms	48	4.37ms	d_add_transposed_g2	1056×4
0.72	171.42ms	48	3.57ms	d_compute_reg_inter_edges	$nthreads \times 4$
0.69	164.79ms	47	3.51ms	update_x	-
0.58	136.66ms	39	3.50ms	update_s	-
0.43	101.11ms	48	2.11ms	[CUDA memcpy DtoD]	-
0.34	80.81ms	305	264.95us	[CUDA memcpy DtoH]	-
0.32	75.36ms	2	37.68ms	[CUDA memcpy HtoD]	-
0.12	28.32ms	48	589.99us	d_compute_node_gradients	-
0.01	2.50ms	1	2.50ms	initialize_s	-
0.01	2.47ms	48	51.44us	d_reset_edge_gradients	-
0.01	2.10ms	3	700.22us	[CUDA memset]	-
0.01	1.93ms	48	40.16us	d_compute_fx_inter	$nthreads \times 4$
0.00	1.16ms	1	1.16ms	d_node_gradients_histogram_weighted	$nthreads \times 21 \times 4$
0.00	843.20us	304	2.77us	sum_reduction	$nthreads \times 4$
0.00	629.89us	48	13.12us	d_compute_reg_inter_nodes	$nthreads \times 4$
0.00	74.43us	1	74.43us	d_transpose_msa	1024×4
0.00	2.18us	1	2.18us	d_update_w	-

Table S2: Overview over the different kernels with runtimes for a synthetic alignment with $N = 3000$ and $L = 300$

Method M	Total runtime (s)	Runtime relative to CCMpred.GPU
CCMpred.GPU	3120	1.0
CCMpred.6Core	21702	7.0
plmDCA.Asym.6Core	117841	37.8
GREMLIN.1Core	248817	79.7
PSICOV.1Core	30236	9.7

Table S3: Runtimes on real-world protein data set of 150 PSICOV test cases.

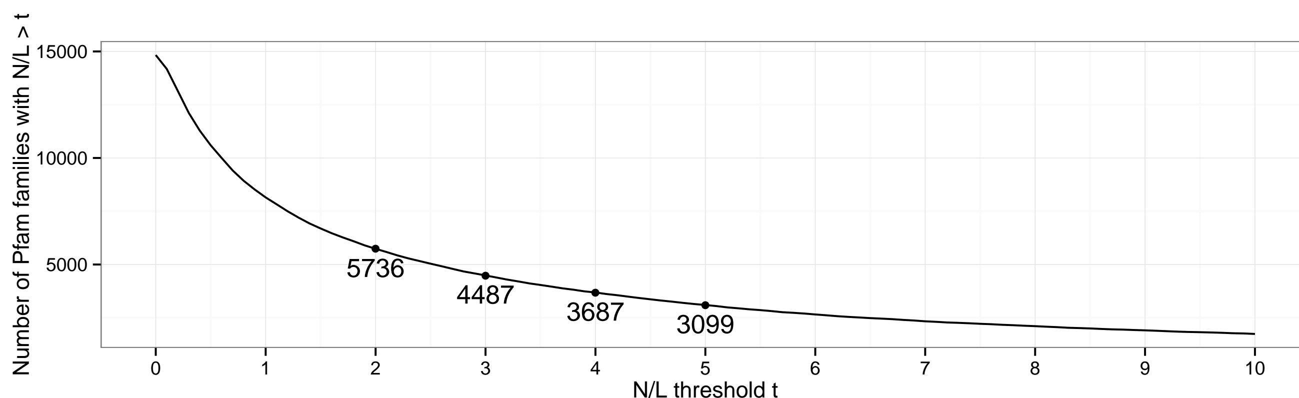

Figure S5: Number of Pfam families with sequence coverage above a given N/L threshold, out of the total 14831 Pfam families (as of June 2014)

regression model fitted to our experimentally determined runtimes.

Figure S6 shows the estimated cumulative contact prediction runtimes in (single-core) CPU years for all Pfam family alignments with a sequence coverage N/L larger than different thresholds t. For the currently typical thresholds of $t = 3$ or $t = 5$, the runtime of the next-fastest method of highest accuracy is at 29-24 CPU years while CCMpred would take approximately 211-175 days on one GPU. A single contact prediction run on a 256-core cluster would consequently take competing methods at least a month of computation time at full utilization. While a single contact prediction run is still feasible using existing CPU-based methods, real-world applications will require the computation of contact predictions from multiple sequences alignments generated with different parameters. Indeed, the EVfold webserver (Marks *et al.*, 2012) will generate alignments for varying alignment E-value cutoffs before using a heuristic to choose an alignment. It might additionally be advantageous to optimize parameters such as a minimal coverage filter, sequence identity threshold for reweighting, and many others. Depending on how many such settings are tried, the total runtimes can easily increase by a factor of 10-100 in realistic scenarios. A fast contact prediction method is therefore essential to make such studies feasible. Another important benefit of fast contact prediction methods is that they can be more broadly applied, either on an experimentalist's workbench that can now run predictions in a reasonable time or by providing a webserver that will have quicker turnaround times and will require fewer hardware resources. By these speedups, contact prediction can become a new tool applicable for exploratory research. We also hope that since CCMpred is available as open source code, it might serve as a fast foundation for further methods development.

Supplementary References

Andreeva, A., Howorth, D., Chandonia, J.-M., Brenner, S. E., Hubbard, T. J. P., Chothia, C., and Murzin, A. G. (2008). Data growth and its impact on the SCOP database: new developments. *Nucleic acids research*, **36**(Database issue), D419–25.

Dunn, S. D., Wahl, L. M., and Gloor, G. B. (2008). Mutual information without the influence of phylogeny or entropy dramatically improves residue contact prediction. *Bioinformatics*, **24**(3), 333–40.

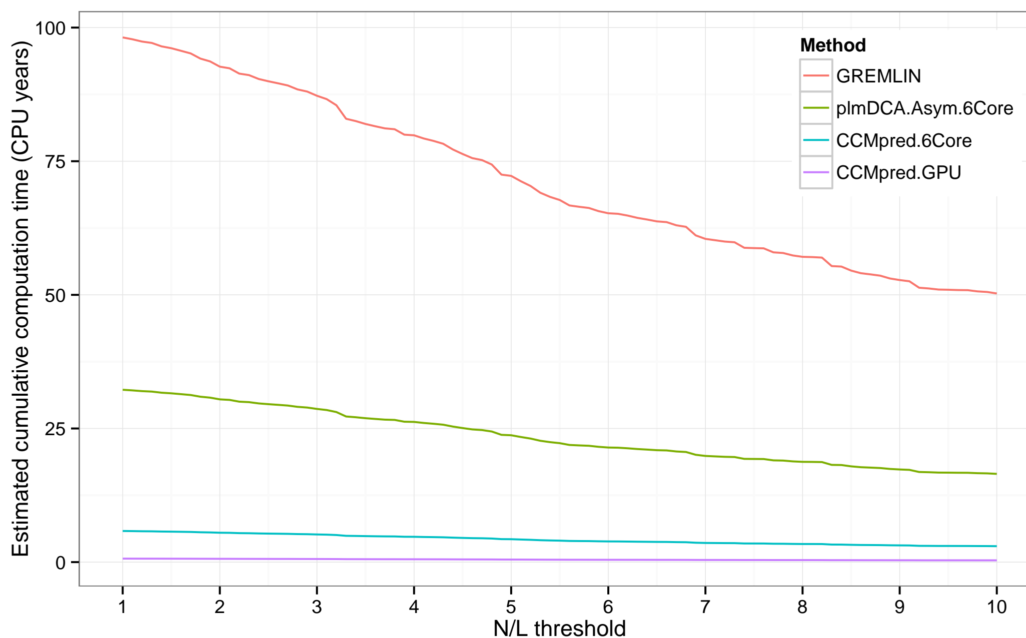

Figure S6: Total time necessary to predict contacts for all alignments in Pfam above a given N/L threshold.

Ekeberg, M., Lövkvist, C., Lan, Y., Weigt, M., and Aurell, E. (2013). Improved contact prediction in proteins: Using pseudolikelihoods to infer Potts models. *Physical Review E*, **87**(1), 012707.

Finn, R. D., Bateman, A., Clements, J., Coggill, P., Eberhardt, R. Y., Eddy, S. R., Heger, A., Hetherington, K., Holm, L., Mistry, J., Sonnhammer, E. L. L., Tate, J., and Punta, M. (2014). Pfam: the protein families database. *Nucleic acids research*, **42**(Database issue), D222–30.

Jones, D. T., Buchan, D. W., Cozzetto, D., and Pontil, M. (2012). PSICOV: precise structural contact prediction using sparse inverse covariance estimation on large multiple sequence alignments. *Bioinformatics*, **28**(2), 184–90.

Kamisetty, H., Ovchinnikov, S., and Baker, D. (2013). Assessing the utility of coevolution-based residue-residue contact predictions in a sequence- and structure-rich era. *PNAS*, **110**(39), 15674–15679.

Marks, D. S., Hopf, T. a., and Sander, C. (2012). Protein structure prediction from sequence variation. *Nature biotechnology*, **30**(11), 1072–80.