High-Performance Sum-Product Decoding of Quasi-Cyclic LDPC Codes

Christoph Pacher[†], **Bernhard Ömer**

[†] `Christoph.Pacher@AIT.ac.at`

Safety & Security Department, AIT Austrian Institute of Technology GmbH, 1220 Vienna, Austria

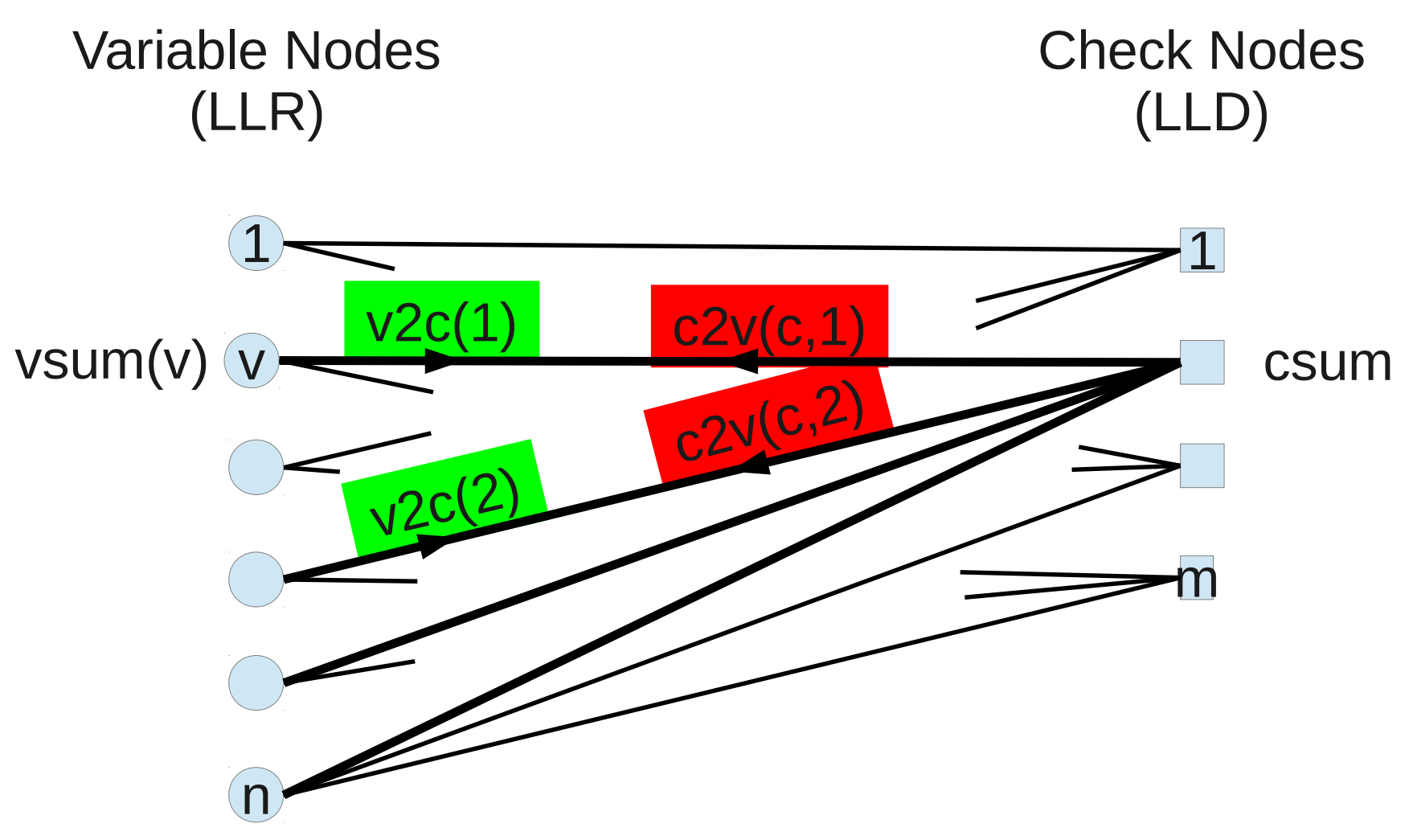
AUSTRIAN INSTITUTE
OF TECHNOLOGY

SUMMARY

We present a highly optimized modification of the Sum-Product Algorithm for LDPC decoding for CPUs which achieves the same decoding properties as the original algorithm but offers a throughput on a CPU that is comparable to GPU implementations using hundreds of GPU cores. To achieve this improvement we make use of i) quasi-cyclic LDPC codes and vectorized SSE commands, ii) interleaved variable/check node processing, and iii) concurrent use of Log-Likelihood Ratio and Log-Likelihood Difference representations and fast conversion between them. For typical parameters we can achieve a throughput of approx. 40 Mbit/s on a quad-core CPU.

1 SUM-PRODUCT DECODING OF LDPC CODES

- A Low-Density Parity-Check (LDPC) code [1] can be defined by its *sparse* parity-check matrix $\mathbb{H}$: The null-space of the parity-check matrix defines the set of all codewords: $\mathcal{C} = \left\{\boldsymbol{x} \in \{0,1\}^n : \boldsymbol{x}\mathbb{H}^T = \boldsymbol{0}\right\}$.
- The iterative sum-product algorithm (SPA) efficiently solves the NP-hard maximum-likelihood decoding problem of finding the codeword with the minimum Hamming distance to a received word in good approximation.



Iterative algorithm - Update Rules

$$LLR_{\text{v2c}}(v,c) = LLR_{ch}(v) + \sum_{c'\in\mathcal{C}_v\setminus\{c\}} LLR_{\text{c2v}}(v,c'),$$

$$LLD_{\text{c2v}}(v,c) = \prod_{v'\in\mathcal{V}_c\setminus\{v\}} \operatorname{sign} LLD_{\text{v2c}}(v',c) \sum_{v'\in\mathcal{V}_c\setminus\{v\}} \left|LLD_{\text{v2c}}(v',c)\right|$$

$\mathcal{C}_v$ = set of check-nodes adjacent to v,
$\mathcal{V}_c$ = set of variable-nodes adjacent to c.

2 OPTIMIZING CHECK NODE & VARIABLE NODE UPDATES

Algorithm 1: Interleaved Sum-Product Algorithm

```
initialize_c2v() /* check → variable */ ;
initialize_vsum();
for $iter = 1$ to $max\_iter$ do
    for $checknode = 1$ to $number\_of\_checknodes$ do
        csum=0;
        for $edge = 1$ to $degree(checknode)$ do
            variablenode = adjancency_list(checknode, edge);
            vsum(variablenode)-= c2v(checknode, edge);
            v2c(edge)=LLR2LLD(vsum(variablenode));
            csum += v2c(edge);
        for $edge = 1$ to $degree(checknode)$ do
            variablenode = adjancency_list(checknode, edge);
            c2v(checknode, edge)=LLD2LLR(csum-v2c(edge));
            vsum(variablenode)+= c2v(checknode, edge)/* restore vsum */ ;
    if converged() then
        return iter;
return ERROR;
```

We use Quasi-Cyclic codes:

- each variable and check node is replaced by a vector of nodes,
- each 1 in the parity check matrix is replaced by a rotation matrix with random offset.

Vectorized (parallel) operations are used: 4 x int32_t and 4 x float32_t on 128 bit registers (SSE, SSE2 extension) for arithmetic and shift operations.
Bit operations are bit-sliced using 64 bit all-purpose registers.

3 TRANSFORMING BETWEEN LLR AND LLD

Two domains for probabilities/likelihoods are used:

- Log-Likelihood Ratio: $LLR(p) = \log_2\left(\frac{1-p}{p}\right)$
- Signed Log-Likelihood Difference: $LLD(p) = \operatorname{sign}(2p-1)\log_2|2p-1|$.
- Transformation between $LLR(p)$ and $LLD(p)$

$$\begin{aligned} |LLR(p)| &= \gamma(|LLD(p)|), \\ |LLD(p)| &= \gamma(|LLR(p)|), \\ \operatorname{sign}(LLR(p)) &= \operatorname{sign}(LLD(p)) \end{aligned}$$

- Transform

$$\gamma(x) = -\log_2(\tanh_2(x/2)) = \log_2\left(\frac{2^x+1}{2^x-1}\right) = \log_2\left[1+\left(2^{x-1}-\frac{1}{2}\right)^{-1}\right]$$

Note, that γ is an involution, i.e. $\gamma^{-1}(x) = \gamma(x) \Rightarrow \gamma(\gamma(x)) = 1$.
We use the fast approximations (see box **4**) for $\log_2$ and 2^x (red curve).

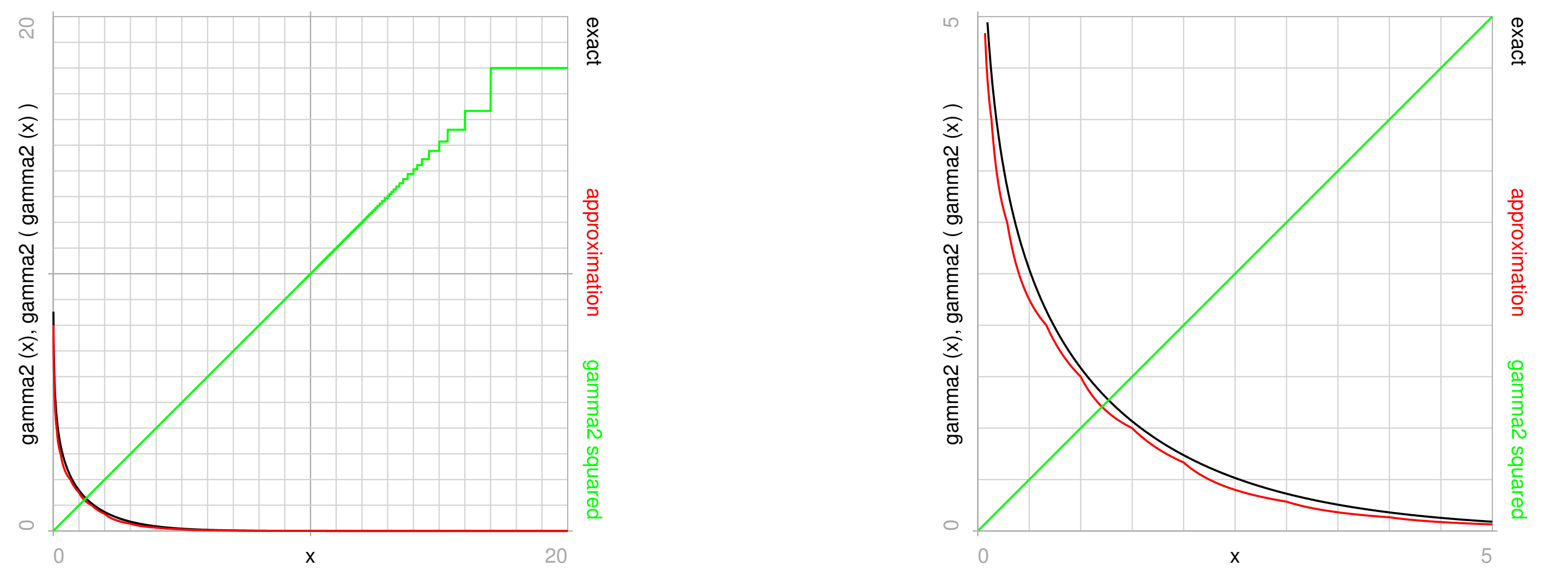


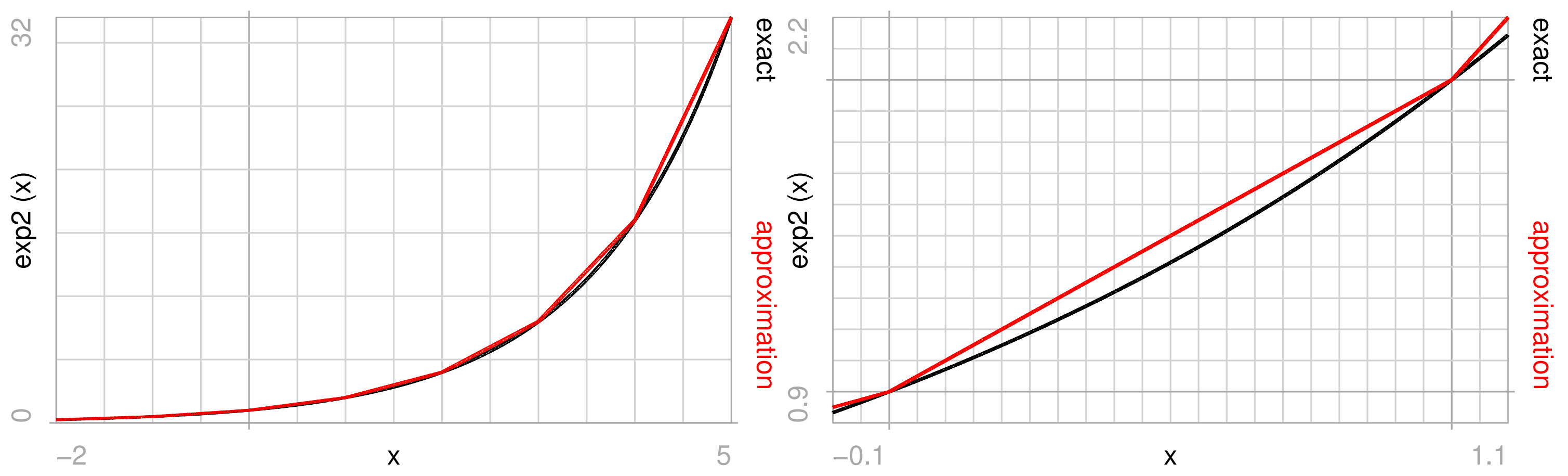
Figure: Approximated (red) and exact (black) $\gamma(x)$. $\gamma(\gamma(x))$ is shown (green) to demonstrate that the approximated $\gamma(x)$ is very close to an involution for relevant inputs.

4 FAST APPROXIMATIONS FOR $\log_2$ AND 2^X

- Fast $y' \approx y = \exp_2(x) = 2^x$ operation:
  Input: 32 bit fixed point number x (stored in `int32_t`).
  Output: 32 bit IEEE 754 floating point number $y' \approx 2^x$.
  Algorithm (based on [2]):
  - shift x to the left by 7 bit: `x <<= 7;`
  - add an offset of 127 to the exponent: `x += 0x3f800000;`
  - interpret the result as float (using a union) y'

  The last step is no actual operation, as it is just interpreting the memory content as float.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	0...7	8	9	a	b	c	d	e	f
x	s	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	f...f	f	f	f	f	f	f	f	f
	s	i	i	i	i	i	i	i	i	f	f	f	f	f	f	f	f...f	f	0	0	0	0	0	0	0
	s	e	e	e	e	e	e	e	e	m	m	m	m	m	m	m	m...m	m	m	m	m	m	m	m	m
y'	s	e’	e’	e’	e’	e’	e’	e’	e’	m	m	m	m	m	m	m	m...m	m	m	m	m	m	m	m	m



The exact result would be $y = 2^x$ (black curve).
Above algorithm calculates (if $x > 0$) $y' = 2^{\lfloor x \rfloor} \times (1 + \operatorname{fract}(x))$ (red curve).

- Fast $x' \approx x = log_2(y)$ operation:
  Input: float y, operations performed in opposite order.
  Output: 32 bit fixed point number x.

5 SIMULATION RESULTS

- Despite the approximations made for $\gamma(\cdot)$, the error correction performance matches that of the classical Sum-Product Algorithm.
- The throughput is approximately **100 MBit/s per decoder iteration** on a single CPU core (2.5 GHz)!
- For $iter = 10$ on a quad-core cpu, we achieve 40 Mbit/s throughput.

REFERENCES

[1] R G Gallager, *Low-density parity-check codes*, IEEE Transactions on Information Theory **8**, 21–28 (1962).

[2] P Mineiro, *fastapprox*, `http://code.google.com/p/fastapprox/`.

ACKNOWLEDGEMENTS

VIENNA SCIENCE
AND TECHNOLOGY FUND