Massively parallel multicanonical simulations on GPUs

Jonathan Gross[1]
with Johannes Zierenberg,[1] Martin Weigel[2], and Wolfhard Janke[1]

[1] Institute for Theoretical Physics – The University of Leipzig, Germany
[2] Applied Mathematics Research Centre – The University of Coventry, UK

November 25, 2016

Computational Quantum Field Theory

Outline

Timeline of generalized ensemble techniques

These advanced Monte Carlo technique are designed to estimate the density of states.

- 1986 - Replica-exchange (Swendsen, Wang)
- 1989 - Multiple histogram reweighting (Ferrenberg, Swendsen)
- 1991 - Parallel tempering (Geyer; 1996 Hukushima, Nemoto)
- 1992 - Multicanonical (MUCA) sampling (Berg, Neuhaus)
- 2001 - Wang-Landau sampling (Wang, Landau)
- 2013 - Parallel MUCA (Zierenberg, Marenz, Janke) (few cores)

Now we expand the parallel MUCA algorithm to tens of thousands of cores.

Parallel multicanonical weight iteration

The canonical partition sum can be rewritten in terms of the density of states $\Omega(E)$

$$\mathcal{Z}_{\rm can} = \sum_E \Omega(E) e^{-\beta E} \rightarrow \mathcal{Z}_{\rm muca} = \sum_E \Omega(E) W(E)$$

Parallel multicanonical weight iteration

The canonical partition sum can be rewritten in terms of the density of states $\Omega(E)$

$$\mathcal{Z}_{\mathrm{can}} = \sum_E \Omega(E) e^{-\beta E} \to \mathcal{Z}_{\mathrm{muca}} = \sum_E \Omega(E) W(E)$$

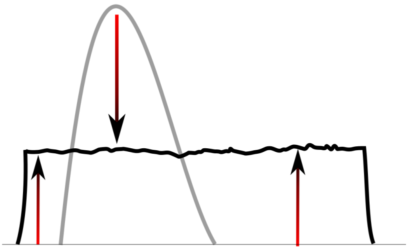


Figure: Flat histogram

For a flat histogram $W(E) \propto \Omega^{-1}(E)$. Since the density of states is usually not known in advance, the weights have to be determined iteratively.

Parallel multicanonical weight iteration

The time-consuming generation of statistics is distributed on p independent worker processes. The parallelization profits from a minimum of communication because communication happens only once per iteration.

Parallel multicanonical weight iteration

The time-consuming generation of statistics is distributed on p independent worker processes. The parallelization profits from a minimum of communication because communication happens only once per iteration.

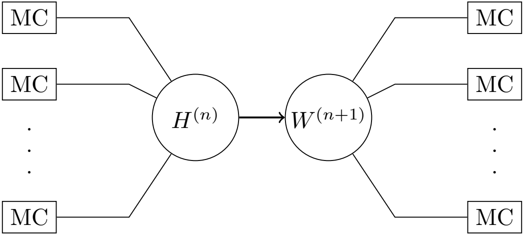


Figure: Parallel multicanonical simulation scheme

$$\sum_i H_i^{(n)}(E) = H^{(n)}(E) \rightarrow W^{(n+1)}(E) = W_i^{(n+1)}(E)$$

with

$$W^{(n+1)}(E) = \frac{W^{(n)}(E)}{H^{(n)}(E)}$$

Hardware overview

We aim to achieve a **fair** comparison of two hardware architectures.
Same parallel implementation on CPU and GPU, as opposed to highly optimized GPU code vs. serial CPU code.

Hardware overview

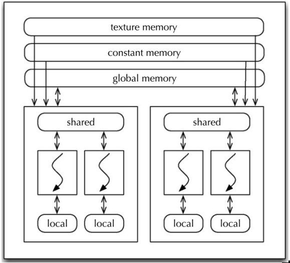
We aim to achieve a **fair** comparison of two hardware architectures.
Same parallel implementation on CPU and GPU, as opposed to highly optimized GPU code vs. serial CPU code.

	CPU	GPU1	GPU2
model	2× Xeon E5-2640	Tesla K20m (ECC)	GTX Titan Black
peak clock speed	3072 MHz	706 MHz	980 MHz
# cores	12 (24 w/ HT)	2496	2880
SMX	N/A	13	15
memory bandwidth	42.6 GB/s	208 GB/s	336 GB/s
peak performance	2× 120 GFlop/s	3.5 TFlop/s	5.1 TFlop/s
thermal design power	2×95 W	225 W	250 W

Table: List of considered CPU and GPU hardware with selected properties, including the clock speed, the number of total cores, the number of streaming multiprocessors (SMX), the memory bandwidth and the power consumption (thermal design power TDP). Both GPUs are from the Kepler generation such that each SMX features 192 cores.

GPU architecture

- GPGPUs specifically designed for HPC
- but also gaming PC GPUs can be used
- features streaming multiprocessors (SMX) with multiple cores
- currently $\sim$ 2500 to 3000 cores in total
- available memory typically 6 GB to 12 GB
- programmed using CUDA, a subset of C99/C++
- you have to optimize your code for different layers of memory
- overload cores with many threads to hide memory latency (get optimal number of worker threads using CUDA occupancy calculator)



Implementation

Our test system is the 2D Ising model with periodic boundary conditions.

Implementation

Our test system is the 2D Ising model with periodic boundary conditions.

We use the same code base for MPI and CUDA. All MUCA specific code is identical, only slight architecture specific code.

Implementation

Our test system is the 2D Ising model with periodic boundary conditions.

We use the same code base for MPI and CUDA. All MUCA specific code is identical, only slight architecture specific code.
Some implementational remarks:

Implementation

Our test system is the 2D Ising model with periodic boundary conditions.

We use the same code base for MPI and CUDA. All MUCA specific code is identical, only slight architecture specific code.
Some implementational remarks:

- each worker has own copy of the lattice

Implementation

Our test system is the 2D Ising model with periodic boundary conditions.

We use the same code base for MPI and CUDA. All MUCA specific code is identical, only slight architecture specific code.
Some implementational remarks:

- each worker has own copy of the lattice
- CPU workers have individual histograms, which are combined using `MPI_Reduce`

Implementation

Our test system is the 2D Ising model with periodic boundary conditions.

We use the same code base for MPI and CUDA. All MUCA specific code is identical, only slight architecture specific code.
Some implementational remarks:

- each worker has own copy of the lattice
- CPU workers have individual histograms, which are combined using `MPI_Reduce`
- GPU workers write to same histogram using atomic operations (memory coalescence)

Implementation

Our test system is the 2D Ising model with periodic boundary conditions.

We use the same code base for MPI and CUDA. All MUCA specific code is identical, only slight architecture specific code.
Some implementational remarks:

- each worker has own copy of the lattice
- CPU workers have individual histograms, which are combined using `MPI_Reduce`
- GPU workers write to same histogram using atomic operations (memory coalescence)
- weight update on CPU identical for both

Implementation

Our test system is the 2D Ising model with periodic boundary conditions.

We use the same code base for MPI and CUDA. All MUCA specific code is identical, only slight architecture specific code.
Some implementational remarks:

- each worker has own copy of the lattice
- CPU workers have individual histograms, which are combined using `MPI_Reduce`
- GPU workers write to same histogram using atomic operations (memory coalescence)
- weight update on CPU identical for both
- 2 instances of Philox RNG per worker
  - fast RNG with good period and small memory footprint
  - selection RNG - always pick same spin in each lattice (memory coalescence)
  - acceptance RNG

Implementation

Our test system is the 2D Ising model with periodic boundary conditions.

We use the same code base for MPI and CUDA. All MUCA specific code is identical, only slight architecture specific code.
Some implementational remarks:

- each worker has own copy of the lattice
- CPU workers have individual histograms, which are combined using `MPI_Reduce`
- GPU workers write to same histogram using atomic operations (memory coalescence)
- weight update on CPU identical for both
- 2 instances of Philox RNG per worker
  - fast RNG with good period and small memory footprint
  - selection RNG - always pick same spin in each lattice (memory coalescence)
  - acceptance RNG

This implementation ensures identical timeseries and histograms for CPU and GPU, if parameters are the same (number of workers and seeds, specifically).

Simulational parameters

Each weight iteration:

- NUPDATES_THERM $= \min(100, 30 \cdot \text{width})$
- while (width $<$ N)
  NUPDATES_MEASURE $= 6 \cdot \text{width}^{2.25}$/NUM_WORKERS
- NUPDATES_MEASURE *= 1.1 **until flat**

Production run:

- thermalization with NUPDATES_THERM $= N^{2.25}$
- NUPDATES_MEASURE calculated from spinflip times to set fixed runtime (10 min)

width - width of energy range covered
$N = L^2$ - full energy range
L - lattice size
NUPDATES_THERM - number of thermalization updates (spin flips)
NUPDATES_MEASURE - number of measurement updates

J. Zierenberg et. al, CPC 184 1155 (2013).

How to tell if your histogram is flat?

- Most flat histogram methods stop the iteration when the histogram is "flat enough".
  As an example we aim for 80% flatness in our histogram, that means if $0.8 \cdot h_{\text{mean}} \leq H(E) \leq h_{\text{mean}}/0.8 \quad \forall E$, with $h_{\text{mean}} = \frac{1}{N_M} \sum H(E)$ our histogram is flat.
- Related to Chebyshev distance

But with this a flat histogram does not necessarily your weights are perfect.

How to tell if your histogram is flat?

- Most flat histogram methods stop the iteration when the histogram is "flat enough".
  As an example we aim for 80% flatness in our histogram, that means if $0.8 \cdot h_{\text{mean}} \leq H(E) \leq h_{\text{mean}}/0.8 \quad \forall E$, with $h_{\text{mean}} = \frac{1}{N_M} \sum H(E)$ our histogram is flat.
- Related to Chebyshev distance

But with this a flat histogram does not necessarily your weights are perfect.

Comparing histograms is the basis for digital image processing, so we borrowed one of the 26 methods we found.

Computational Quantum Field Theory

How to tell if your histogram is flat?

- Most flat histogram methods stop the iteration when the histogram is "flat enough".
  As an example we aim for 80% flatness in our histogram, that means
  if $0.8 \cdot h_{\text{mean}} \leq H(E) \leq h_{\text{mean}}/0.8 \quad \forall E$, with
  $h_{\text{mean}} = \frac{1}{N_M} \sum H(E)$ our histogram is flat.
- Related to Chebyshev distance

But with this a flat histogram does not necessarily your weights are perfect.

Comparing histograms is the basis for digital image processing, so we borrowed one of the 26 methods we found.

- Kullback-Leibler divergence, measure of difference between two probability distributions P and Q
- $d_k = \sum_i P(i) \log \frac{P(i)}{Q(i)}$
- in our implementation we use
  $P(i) = H(E)/\text{NUPDATES\_MEASURE}$ and $Q(i) = 1/\text{NUM\_BINS}$

Thermalization

Influence of the number of thermalization steps on the converge of the Kullback-Leibler divergence.

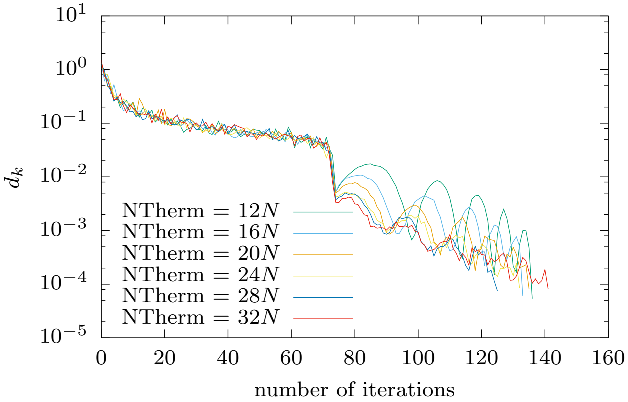


Figure: Kullback-Leibler divergence as function of iterations.

Hardware performance

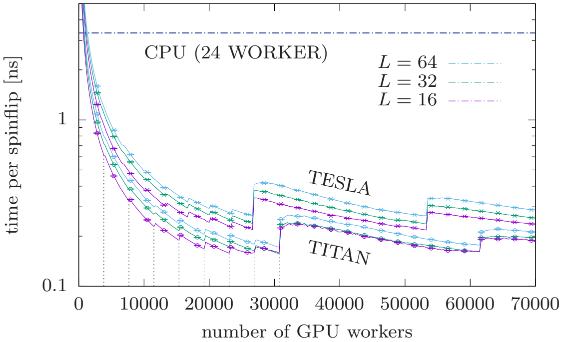
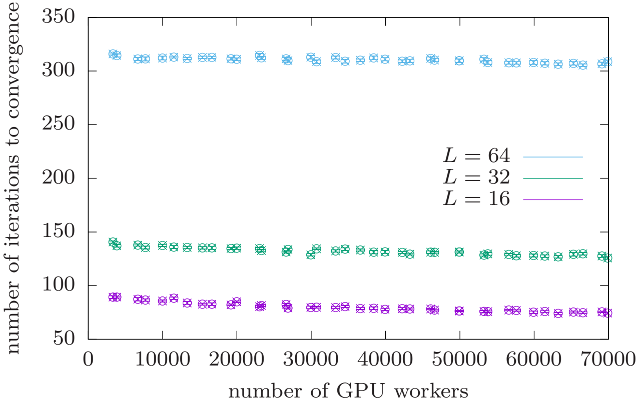


Figure: Hardware performance: spin flip times in ns as function of number of workers.

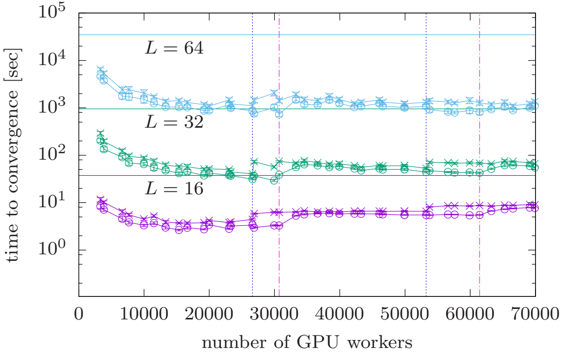
Average CPU spin flip time similar for all system sizes $t_{CPU} \approx 3.5\,\text{ns}$.
Optimal GPU spin flip times – Tesla: 0.21 ns Titan: 0.16 ns
This results in theoretical hardware speedups of factors of 16 and 21, respectively.

Iterations until convergence



Computational Quantum Field Theory

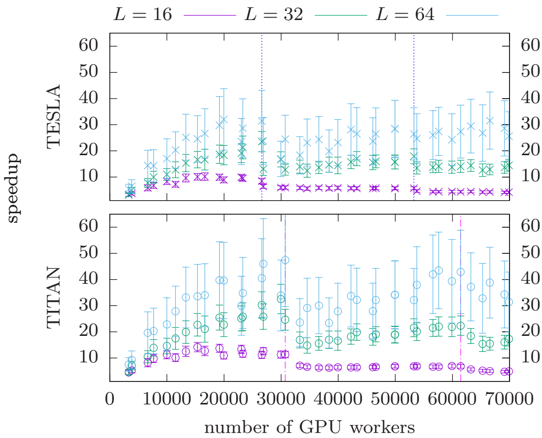
Time to convergence



- solid horizontal lines represent CPU reference times
- vertical dashed lines mark full GPU utilization

Speedup

Software speedup defined by the ratio of CPU and GPU time to convergence.



Density of states

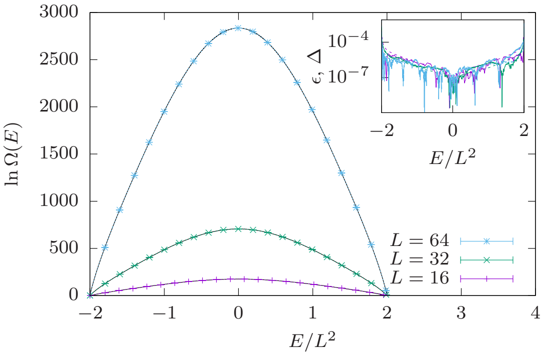


Figure: Density of states for the 2D Ising model obtained from simulation in comparison with exact solution from Beale. The inset shows the deviation of our simulational results from the Beale solution as well as Jackknife errors.

Conclusions and outlook

Conclusions

- Comparison of identical implementation of parallel MUCA using MPI and CUDA (both operating at peak performance)

Conclusions and outlook

Conclusions

- Comparison of identical implementation of parallel MUCA using MPI and CUDA (both operating at peak performance)
- speedup of 20x-30x per GPU compared to a single modern CPU node (12 core plus hyperthreading)

Conclusions and outlook

Conclusions

- Comparison of identical implementation of parallel MUCA using MPI and CUDA (both operating at peak performance)
- speedup of 20x-30x per GPU compared to a single modern CPU node (12 core plus hyperthreading)
- this is equivalent to a speedup of 300x-500x when compared to a single CPU worker

Conclusions and outlook

Conclusions

- Comparison of identical implementation of parallel MUCA using MPI and CUDA (both operating at peak performance)
- speedup of 20x-30x per GPU compared to a single modern CPU node (12 core plus hyperthreading)
- this is equivalent to a speedup of 300x-500x when compared to a single CPU worker
- Kullback-Leibler divergence is a reliable, well-defined flatness criterion

Conclusions and outlook

Conclusions

- Comparison of identical implementation of parallel MUCA using MPI and CUDA (both operating at peak performance)
- speedup of 20x-30x per GPU compared to a single modern CPU node (12 core plus hyperthreading)
- this is equivalent to a speedup of 300x-500x when compared to a single CPU worker
- Kullback-Leibler divergence is a reliable, well-defined flatness criterion
- scientific Tesla cards have very stable performance
- consumer cards can deliver even better performance with decreased "stability" for a fraction of the price

Conclusions and outlook

Conclusions

- Comparison of identical implementation of parallel MUCA using MPI and CUDA (both operating at peak performance)
- speedup of 20x-30x per GPU compared to a single modern CPU node (12 core plus hyperthreading)
- this is equivalent to a speedup of 300x-500x when compared to a single CPU worker
- Kullback-Leibler divergence is a reliable, well-defined flatness criterion
- scientific Tesla cards have very stable performance
- consumer cards can deliver even better performance with decreased "stability" for a fraction of the price

Outlook

Conclusions and outlook

Conclusions

- Comparison of identical implementation of parallel MUCA using MPI and CUDA (both operating at peak performance)
- speedup of 20x-30x per GPU compared to a single modern CPU node (12 core plus hyperthreading)
- this is equivalent to a speedup of 300x-500x when compared to a single CPU worker
- Kullback-Leibler divergence is a reliable, well-defined flatness criterion
- scientific Tesla cards have very stable performance
- consumer cards can deliver even better performance with decreased "stability" for a fraction of the price

Outlook

- Fully documented source code will be available soon

Conclusions and outlook

Conclusions

- Comparison of identical implementation of parallel MUCA using MPI and CUDA (both operating at peak performance)
- speedup of 20x-30x per GPU compared to a single modern CPU node (12 core plus hyperthreading)
- this is equivalent to a speedup of 300x-500x when compared to a single CPU worker
- Kullback-Leibler divergence is a reliable, well-defined flatness criterion
- scientific Tesla cards have very stable performance
- consumer cards can deliver even better performance with decreased "stability" for a fraction of the price

Outlook

- Fully documented source code will be available soon
- Application to aggregation of bead-spring polymers already implemented

Conclusions and outlook

Conclusions

- Comparison of identical implementation of parallel MUCA using MPI and CUDA (both operating at peak performance)
- speedup of 20x-30x per GPU compared to a single modern CPU node (12 core plus hyperthreading)
- this is equivalent to a speedup of 300x-500x when compared to a single CPU worker
- Kullback-Leibler divergence is a reliable, well-defined flatness criterion
- scientific Tesla cards have very stable performance
- consumer cards can deliver even better performance with decreased "stability" for a fraction of the price

Outlook

- Fully documented source code will be available soon
- Application to aggregation of bead-spring polymers already implemented
- Extension to multiple GPUs should be straight-forward

Thank you for your attention.

Funding:

Polymers under multiple constraints