Eur. Phys. J. Special Topics **194**, 87–119 (2011)

DOI: 10.1140/epjst/e2011-01398-x

THE EUROPEAN PHYSICAL JOURNAL SPECIAL TOPICS

Review

GPU-computing in econophysics and statistical physics

T. Preis[1,2,a,b]

[1] Center for Polymer Studies, Department of Physics, 590 Commonwealth Avenue, Boston, MA 02215, USA
[2] Artemis Capital Asset Management GmbH, Gartenstr. 14, 65558 Holzheim, Germany

Received 17 January 2011 / Received in final form 23 February 2011
Published online 7 April 2011

Abstract. A recent trend in computer science and related fields is general purpose computing on graphics processing units (GPUs), which can yield impressive performance. With multiple cores connected by high memory bandwidth, today's GPUs offer resources for non-graphics parallel processing. This article provides a brief introduction into the field of GPU computing and includes examples. In particular computationally expensive analyses employed in financial market context are coded on a graphics card architecture which leads to a significant reduction of computing time. In order to demonstrate the wide range of possible applications, a standard model in statistical physics – the Ising model – is ported to a graphics card architecture as well, resulting in large speedup values.

1 Introduction

In computer science applications and diverse interdisciplinary fields of science such as computational physics or quantitative finance, the computational power requirements have continuously increased with time. One prominent example is high-frequency trading (HFT) which is focused on automatic trading decisions. Decisions to buy or to sell financial assets are made by computer algorithms running on computer systems of an exchange member. Such an algorithm analyzes the flow of incoming information received from the exchange system. Information includes new transactions taking place with their corresponding transaction prices and transaction volumes, but in some systems also order submission, order modification and order deletion events of other exchange members. If a trading algorithm decides to submit a buy or sell order to the exchange system, then within a few milliseconds this information is sent from exchange member's system to the central exchange server which is responsible for matching offer and demand. The exchange server responds with a confirmation message. The time between order submission and receipt of confirmation is referred to as round-trip time (RTT). To minimize RTTs, exchange members made an effort

[a] e-mail: mail@tobiaspreis.de
[b] Author's website: http://www.tobiaspreis.de

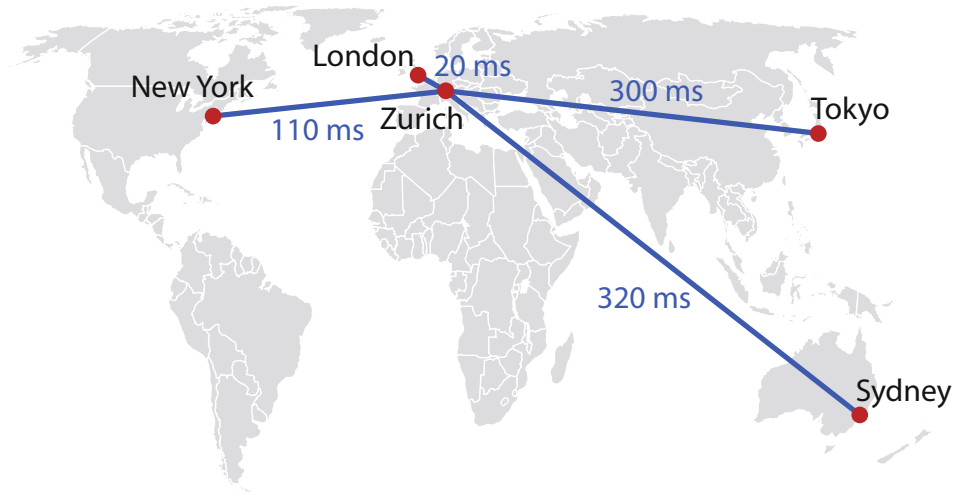

Fig. 1. (Color online) Stock market related RTT estimates for major cities in the world.

to find locations for their computer infrastructure which are as close as possible to the central server of the exchange. A longer distance causes a longer RTT of the information based on the finite speed of light and performance losses in communication components such as routers. Considering communication via public internet, rough RTTs estimates are shown in Fig. 1.

An optimized solution is given by placing the own computer system in the computer center of the exchange which is offered by a few stock and derivative exchanges. Using such a *co-location* service, RTTs of less than 5 milliseconds can be realized. However, typical time intervals between individual transactions are still significant larger – the average interval between transactions for the German DAX future (FDAX) contract traded at the European Exchange (EUREX) was roughly 1.25 seconds in December 2008 [1]. Thus, an equally important question pervades the minds of high-frequency traders: Once you are satisfied with the RTT to the exchange system, how do you use efficiently the time between transactions? That's the motivation for applying concepts from high performance computing which include also graphical processing units (GPUs), a processor that has hundreds of processing cores, compared to the typical four- to 12-core central processing units (CPUs). The GPU originally was developed to render high-resolution details for computer games and became recently available for non-graphical use satisfying huge computational needs.

Such computing requirements, which can also be found in various interdisciplinary computer sciences such as computational physics (e.g., Monte Carlo- and molecular dynamics simulations [2–4] or stochastic optimization [5]) make the use of high performance computing resources. This includes recent multi-core computing solutions based on a shared memory architecture, which are accessible by OpenMP [6] or MPI [7] and can be found in recent personal computers. Furthermore, distributed computing clusters with homogeneous or heterogeneous node structures are available in order to parallelize a given algorithm by separating it into various sub-algorithms.

In addition, general purpose computing on graphics processing units (GPGPU) is becoming an established pillar in computer science and related fields, which can yield impressive performance. Today, we can find a large community of GPU computing developers. Many of them have reported significant speedup of their applications with GPU computing – 10 to 100 times faster. Applications have already been realized in signal processing [8–10], gravitational lensing in astronomy [11], astronomy

in general [12–14], computational chemistry [15–17], chemical physics [18], computational physics [19–30], geophysics [31], biology [32,33], computational biology [34], bioinformatics [35–37], biomedicine [38–40], medical physics [41–46], medical image analysis [47], electromagnetic scattering problems [48], computational electromagnetics [49–53], magnetics [54,55], neural networks [56], pattern recognition [57], genetic programming [58], visualization [59–61], graph drawing [62], and computer science in general [63–71].

With multiple cores connected by high memory bandwidth, today's GPUs offer resources for non-graphics processing. In the beginning, GPU programs used C-like programming environments for kernel execution such as OpenGL shading language [72] or C for graphics (Cg) [73]. The common unified device architecture (CUDA) [74] is a conventional programming approach making use of the unified shader design of recent GPUs from NVIDIA corporation. The programming interface allows for implementing an algorithm using the standard C programming language without any knowledge of the native programming environment. A comparable concept "Close To the Metal" (CTM) [75] was introduced by Advanced Micro Devices Inc. for ATI graphics cards. The computational power of consumer graphics cards roughly exceeds that of a CPU by 1–2 orders of magnitude. A conventional CPU nowadays provides a peak performance of roughly 20×10^9 floating point operations per second (FLOPS) [3]. The consumer graphics card NVIDIA GeForce GTX 280 reaches a theoretical peak performance of 933×10^9 FLOPS. If one were to try to realize the computational power of one GPU with a cluster of several CPUs, a much larger amount of electrical power would be required. A GTX 280 graphics card exhibits a maximum power consumption of 236 W [76], while a recent Intel CPU consumes roughly 100 W. In the meantime, NVIDIA released already the next generation of GPUs – the NVIDIA GTX 400 series, which is roughly two times faster than the previous generation.

This article covers a short introduction into the field of GPU computing and provides introductory examples. In particular computationally expensive analysis procedures employed in financial market context are coded on a graphics card architecture which leads to a significant reduction of computing time. Applications include the analysis of general empirical features of financial market time series which are called *empirical stylized facts*. In order to demonstrate the wide range of possible applications, a standard model in statistical physics, the Ising model, is ported to GPUs as well, resulting in large speedup values.

We apply this general-purpose graphics processing unit (GPGPU) technology to methods of time series analysis, which includes determination of the Hurst exponent and equilibrium autocorrelation function – well-known empirical stylized facts of financial markets. Furthermore, we compare the recent GPU generation with the previous one. All methods are applied to a high frequency data set of the Euro-Bund futures (FGBL) contract and FDAX contract traded at EUREX. In addition, we demonstrate the wide range of possible applications for GPU computing. Thus, we explain how a standard model in statistical physics – the Ising model – can be ported to a graphics card architecture. We will cover both a single GPU as well as a multi-GPU environment.

The remaining parts of this article are organized as follows. In Sec. 2, a brief overview of key facts and properties of the GPU architecture is provided in order to clarify implementation constraints for the following sections. This section provides a very simple introductory example of code as well. GPU accelerated analyses of financial market data – the Hurst exponent estimation and the equilibrium autocorrelation function – is covered in Sec. 3. In two subsections, the performance of the GPU code as a function of parameters is first evaluated for a synthetic time series and compared to the performance on a CPU. Then the time series methods are applied to a financial market time series followed by a discussion of numerical errors. In Sec. 4, we explain

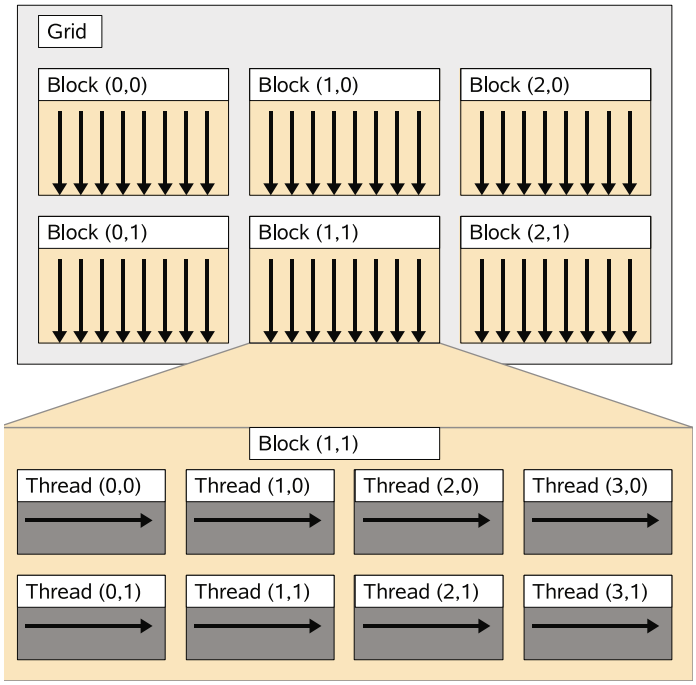

Fig. 2. (Color online) Schematic visualization of the grid of thread blocks for a two dimensional thread and block structure.

how we can use GPUs for Monte Carlo simulations of the Ising model. Two and three dimensional square lattice Ising model simulations with single-spin-flips dynamics are coded in CUDA before we demonstrate multi-spin coding and multi-GPU approaches. Section 5 summarizes our overview of GPU computing.

2 GPU device architecture

In order to provide and discuss information concerning implementation details on a GPU for time series analysis methods, key aspects of the GPU device architecture are briefly summarized in this section. As mentioned in the introduction, we use the compute unified device architecture (CUDA), which allows for implementation of algorithms using standard C with CUDA specific extensions. Thus, CUDA issues and manages computations on a GPU as a data-parallel computing device.

The graphics card architecture used in recent GPU generations is built around a scalable array of streaming multiprocessors [74]. One such multiprocessor contains, amongst others, eight scalar processor cores, a multi-threaded instruction unit, and shared memory, which is located on-chip. When a C program using CUDA extensions and running on the CPU invokes a GPU kernel, which is a synonym for a GPU function, many copies of this kernel – known as threads – are enumerated and distributed to the available multiprocessors, where their execution starts. For such an enumeration and distribution, a kernel grid is subdivided into blocks and each block is subdivided into various threads as illustrated in Fig. 2 for a two-dimensional thread and block structure. The threads of a thread block are executed concurrently in the vacant multiprocessors. In order to manage a large number of threads, a single-instruction multiple-thread (SIMT) unit is used. A multiprocessor maps each thread

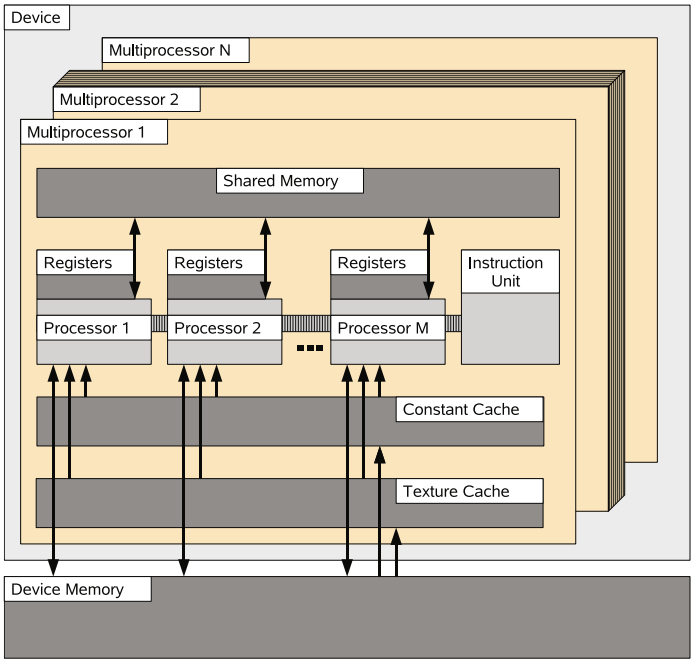

Fig. 3. (Color online) Schematic visualization of a GPU multiprocessor with on-chip shared memory.

to one scalar processor core and each scalar thread works independently of all the others. Threads are created, managed, scheduled, and executed by this SIMT unit in groups of 32 threads. Such a group of 32 threads forms a warp, which is executed on the same multiprocessor. If the threads of a given warp diverge via a data-induced conditional branch, each branch of the warp is executed serially and the processing time of this warp consists of the sum of the branches' processing times.

As shown in Fig. 3, each multiprocessor of the GPU device contains several local 32-bit registers per processor, memory which is shared by all scalar processor cores in a multiprocessor. Furthermore, constant and texture cache are available, which is also shared on the multiprocessor. In order to allow for reducing the number of involved multiprocessors, the slower global memory can be used, which is shared among all multiprocessors and is also accessible by the C function running in the CPU. Please note, that the GPU's global memory is still roughly 10 times faster than current main memory of personal computers. Detailed specifications of the consumer graphics cards 8800 GT and GTX 280 used in this study can be found in Table 1. Furthermore note that former GPU devices only support single-precision floating-point operations. Newer devices starting with the GTX 200 series also support double-precision floating-point numbers. However, each multiprocessor features only one double-precision processing core and so, the theoretical peak performance is significantly reduced for double-precision operations. Further informations about the GPU device properties and CUDA can be found in [74].

Figure 4 provides an introductory example. This example is based on the situation that two arrays `a` and `b` which contain `n` entries are stored in the global memory of the GPU device. The code of the CPU function is executed on the CPU. The code of the GPU function is executed on the GPU device. First, we initialize the three variables `n`, `n_blocks`, and `n_threads`. Variable `n` contains the number of entries in both array. Variable `n_blocks` defines how many blocks we will start on the GPU.

Table 1. Key facts and properties of the applied consumer graphics cards. The theoretical acceleration factor between GeForce 8800 GT and GeForce GTX 280 is given by the difference in number of cores × clock rate. The GTX 480 became recently available.

	GeForce 8800 GT	GeForce GTX 280	GeForce GTX 480
Global memory	512 MB	1024 MB	1536 MB
Number of multiprocessors	14	30	15
Number of cores	112	240	480
Constant memory	64 kB	64 kB	64 kB
Shared memory per block	16 kB	16 kB	48 kB
Warp size	32	32	32
Clock rate	1.51 GHz	1.30 GHz	1.40 GHz

```
//GPU function is executed on the GPU
__global__ void gpu_function(int n, float* a, float* b) {

  //Determine global index
  int i = threadIdx.x + blockIdx.x * blockDim.x;

  //Determine array element
  if(i < n) b[i] += a[i] * a[i];
}

//CPU function is executed on the CPU
__host__ void cpu_function() {

  //Parameters
  int n = 128 * 128;
  int n_blocks = 128;
  int n_threads = 128;

  //First call of the GPU function
  gpu_function<<<n_blocks,n_threads>>>(n, a, b);

  //Control comes back to CPU

  //Second call of the GPU function with modified parameters
  gpu_function<<<n_blocks/2,n_threads*2>>>(n, a, b);
}
```

Fig. 4. Source code of an introductory example.

Variable `n_threads` defines how many threads will be started in each GPU block. The call `gpu_function` invokes the GPU kernel. In addition to the number of blocks and number of threads, we pass the number of array entries and pointers to both arrays to the GPU kernel. Thus, each process on the GPU can calculate its own global index using the inbuilt variables `threadIdx.x` (Thread ID), `blockIdx.x` (Block ID), and `blockDim.x` (number of started blocks). With this global index, each thread on the GPU updates one entry of array `b` with the squared value of the corresponding entry in array `a`. If more threads than existing array entires are started then the `if` statement ensures that only existing entries are changes. After updating all array entries, the control comes back to the CPU – the only way to set a global synchronization point. Threads within a block can be synchronized by the special command

`__syncthreads()`. The second call of the GPU function causes the change of all array entries as well. Only the organization of blocks and threads changed. Information about optimal settings can be found in [74].

3 Financial data analysis on GPUs

3.1 Hurst exponent

The Hurst exponent H [77,78] provides detailed information on the relative tendency of a stochastic process. A Hurst exponent of $H<0.5$ indicates an anti-persistent behavior of the analyzed process, which means that the process is dominated by a mean reversion tendency. $H>0.5$ mirrors a super-diffusive behavior (persistent behavior) of the underlying process. Large values tend to be followed by large values, small values by small values. If the deviations of the current values of the time series from their mean value are independent, which corresponds to a random walk behavior, a Hurst exponent of $H = 0.5$ is obtained.

The Hurst exponent H was originally introduced by Harold Edwin Hurst [79], a British government administrator. He studied records of the Nile river's volatile rain and drought conditions and noticed interesting coherences for flood periods. Harold Edwin Hurst observed in the eight centuries of records that there was a tendency for a year with good flood condition to be followed by another year with good flood conditions and vice versa. Nowadays, the Hurst exponent is well studied in context of financial markets [80–84]. Typically, an anti-persistent behavior can be found on short time scales due to the non-zero gap between offer and demand. On medium time scales, a super-diffusive behavior can be detected [82]. On long time scales, a diffusive regime is reached, due to the law of large numbers.

For a time series $p(t)$ with $t \in \{1, 2, \ldots, T\}$, the time lag dependent Hurst exponent $H_q(\Delta t)$ can be determined by the general relationship

$$\langle |p(t + \Delta t) - p(t)|^q \rangle^{1/q} \propto \Delta t^{H_q(\Delta t)} \tag{1}$$

with the time lag $\Delta t \ll T$ and $\Delta t \in \mathbb{N}$. The brackets $\langle \ldots \rangle$ denote the expectation value. Apart from Eq. (1), there are also other calculation methods, e.g., rescaled range analysis [77]. We present the Hurst exponent determination implementation on a GPU for $q = 1$ and use $H(\Delta t) \equiv H_{q=1}(\Delta t)$. The process to be analyzed is a synthetic anti-correlated random walk, which was introduced in [80]. This process emerges from the superposition of two random walk processes with different time scale characteristics. Thus, a parameter dependent negative correlation at time lag one can be observed. As a first step, one has to allocate memory on the GPU device's global memory for the time series, intermediate results, and final results. In a first approach, the time lag dependent Hurst exponent is calculated up to $\Delta t_{\max} = 256$. In order to simplify the reduction process of the partial results, the overall number of time steps T has to satisfy the condition

$$T = (2^\alpha + 1) \cdot \Delta t_{\max}, \tag{2}$$

with α being an integer number called the length parameter of the time series. The number of threads per block – known as block size – is equivalent to $\Delta t_{\max}$. The array for intermediate results has length T as well, whereas the array for the final results contains $\Delta t_{\max}$ entries. After allocation, the time series data have to be transferred from main memory to the GPU's global memory. When this step is completed, the main calculation part can start. As illustrated in Fig. 5 for block 0, each block,

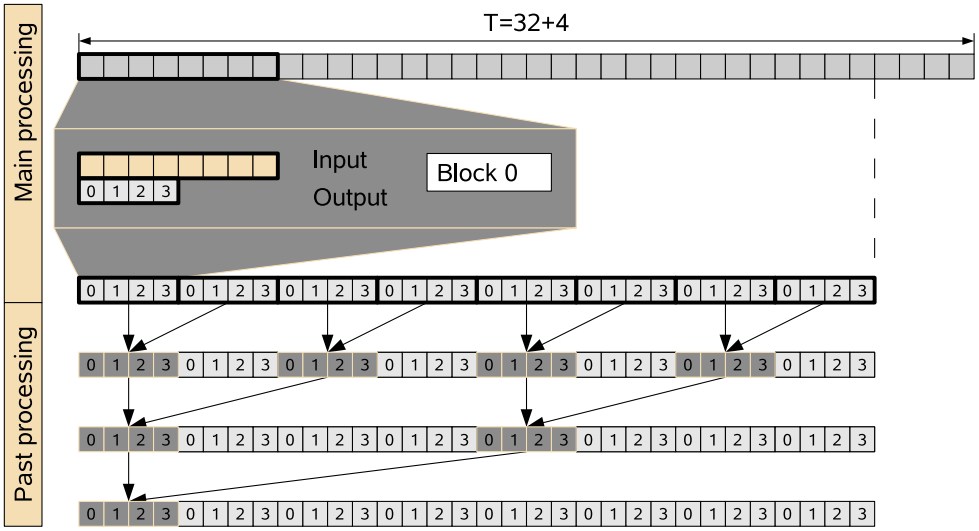

Fig. 5. (Color online) Schematic visualization of the determination of the Hurst exponent on a GPU architecture for $T = 32 + 4$ and $\Delta t_{\max} = 4$. The calculation is split into various processing steps to ensure that their predecessors are completed.

which contains $\Delta t_{\max}$ threads each, loads $\Delta t_{\max}$ data points of the time series from global memory to shared memory. In order to realize such a high-performance loading process, each thread[1] loads one value and stores this value in the array located in shared memory, which can be accessed by all the threads of a block. Analogously, each block also loads the next $\Delta t_{\max}$ entries. In the main processing step, each thread is in charge of one specific time lag. Thus, each thread is responsible for a specific value of Δt and summarizes the terms $|p(t + \Delta t) - p(t)|$ in the block subsegment of the time series. As the maximum time lag is equivalent to the maximum number of threads and as the maximum time lag is also equivalent to half of the data points loaded per block, all threads have to sum the same number of addends resulting in a uniform workload in the graphics card. However, as it is only possible to synchronize threads within a block, and native block synchronization does not exist, partial results of each block have to be stored in block-dependent areas of the array for intermediate results, as shown in Fig. 5. The termination of the GPU kernel function ensures that all blocks are executed. In a post processing step, the partial arrays have to be reduced. This is realized by a binary tree structure, as indicated in Fig. 5. After this reduction, the resulting values can be found in the first $\Delta t_{\max}$ entries of the intermediate array and a final processing kernel is responsible for normalization and gradient calculation. The source code of these GPU kernel functions can be found in Fig. 6.

For the comparison between CPU and GPU implementation, we use an Intel Core 2 Quad CPU (Q6700) with 2.66 GHz and 4096 kB cache size, of which only one core is used. The standard C source code executed on the host is compiled with the gcc compiler (version 4.2.1). The results for $\Delta t_{\max} = 256$ and the consumer graphics card 8800 GT can be found in Fig. 7. The acceleration factor β, which is shown in the inset, reaches a maximum value of roughly 40, and is determined by the relationship

$$\beta = \frac{\text{Total processing time on CPU}}{\text{Total processing time on GPU}}. \tag{3}$$

[1] Thread and block IDs are accessible in standard C language via built-in variables as we have seen in the introductory example.

```
__global__ void dev(float* in,float* out) {
  __shared__ float in_s[2*BLOCK_SIZE];
  __shared__ float out_s[BLOCK_SIZE];
  in_s[threadIdx.x]=in[blockIdx.x*BLOCK_SIZE+threadIdx.x];
  in_s[BLOCK_SIZE+threadIdx.x]=
       in[BLOCK_SIZE+blockIdx.x*BLOCK_SIZE+threadIdx.x];
  out_s[threadIdx.x]=0;
  __syncthreads();
  for(int t=0;t<BLOCK_SIZE;++t) {
    out_s[threadIdx.x]+=fabs(in_s[t+threadIdx.x]-in_s[t]);
  }
  __syncthreads();
  out[blockIdx.x*BLOCK_SIZE+threadIdx.x]=out_s[threadIdx.x];
}

__global__ void dev_postprocessing(float* in,int offset) {
  __shared__ float in_s[2*BLOCK_SIZE];
  in_s[threadIdx.x]=in[2*blockIdx.x*offset+threadIdx.x];
  in_s[BLOCK_SIZE+threadIdx.x]=
       in[2*blockIdx.x*offset+offset+threadIdx.x];
  __syncthreads();
  in_s[threadIdx.x]=in_s[threadIdx.x]+in_s[BLOCK_SIZE+threadIdx.x];
  in[2*blockIdx.x*offset+threadIdx.x]=in_s[threadIdx.x];
}

__global__ void dev_finalprocessing(float* in,float* out,
     int in_size) {
  __shared__ float in_s[BLOCK_SIZE];
  if(threadIdx.x>0) in_s[threadIdx.x]=
       log10(in[threadIdx.x]/(in_size-BLOCK_SIZE));
  __syncthreads();
  if(threadIdx.x>1) out[threadIdx.x]=
       (in_s[threadIdx.x]-in_s[threadIdx.x-1])/
       (log10((float)(threadIdx.x))-log10((float)(threadIdx.x-1)));
}
```

Fig. 6. Source code of GPU kernel functions.

A smaller speed-up factor can be measured for small values of α, as the relative fraction of allocation time and time for memory transfer is larger than the time needed for the calculation steps. The corresponding analysis for the GTX 280 yields a larger acceleration factor β of roughly 70. If we increase the maximum time lag $\Delta t_{\max}$ to 512, which is only possible for the GTX 280, a maximum speed-up factor of roughly 80 can be achieved, as shown in Fig. 8. This indicates that $\Delta t_{\max} = 512$ leads to a higher efficiency on the GTX 280.

At this point, we can also compare the ratio between the performances of 8800 GT and GTX 280 for our application to the ratio of theoretical peak performances. The latter is given as the number of cores multiplied by the clock rate, which amounts to roughly 1.84. If we compare the total processing times on these GPUs for $\alpha = 15$ and $\Delta t_{\max} = 256$, we obtain an empirical performance ratio of 1.7. If we use the acceleration factors for $\Delta t_{\max} = 256$ on the 8800 GT and for $\Delta t_{\max} = 512$ on the GTX 280 for comparison, we get a value of 2.

Following this performance analysis, we apply the GPU implementation to real financial market data in order to determine the Hurst exponent of the Euro-Bund futures contract traded on the European exchange (Eurex). We also validate the

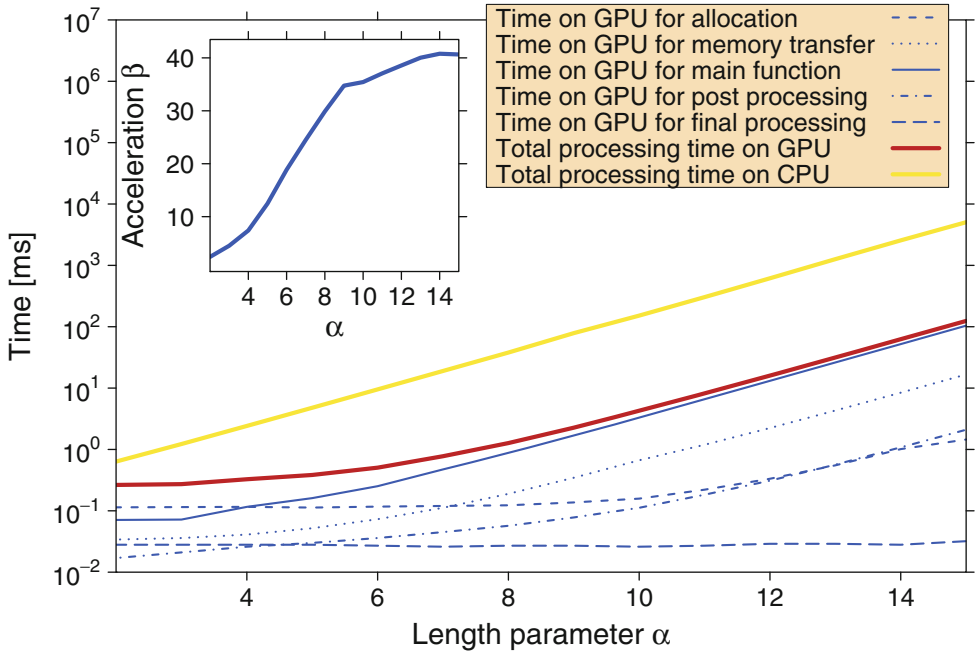

Fig. 7. (Color online) Processing times vs. length parameter α for the calculation of the Hurst exponent $H(\Delta t)$ on GPU and CPU for $\Delta t_{\max} = 256$. The time series contains $T = (2^{\alpha} + 1) \cdot \Delta t_{\max}$ data points. The consumer graphics card 8800 GT is used as GPU device. The total processing time on the GPU can be broken into allocation time, time for memory transfer, time for main processing, time for post processing, and time for final processing. The acceleration factor β is shown in the inset. A maximum acceleration factor of roughly 40 can be obtained. Furthermore, at $\alpha = 9$ there is a break of the slope of the acceleration which is influenced by cache size effects.

accuracy of the GPU calculations by quantifying deviations from the calculation on a CPU. The Euro-Bund futures contract (FGBL) is a financial derivative. As noted earlier, a futures contract is a standardized contract to buy or sell a specific underlying instrument at a proposed date in the future (called the expiration time of the futures contract) at a specified price. The underlying instruments of the FGBL contract are long-term debt instruments issued by the Federal Republic of Germany with remaining terms of 8.5 to 10.5 years and a coupon of 6 percent. We use the Euro-Bund futures contract with expiration time June 2007. The time series shown in Fig. 9 contains 1,051,982 trades, recorded from 8 March 2007 to 7 June 2007. In all presented calculations of the FGBL time series on the GPU, α is fixed to 11. Thus, the data set is limited to the first $T = 1,049,088$ trades in order to fit the data set length to the constraints of the specific GPU implementation. In Fig. 10, the time lag dependent Hurst exponent $H(\Delta t)$ is presented. On short time scales, the well-documented anti-persistent behavior is detected. On medium time scales, small evidence is observed, that the price process reaches a super-diffusive regime. For long time scales the price dynamics tend to random walk behavior ($H = 0.5$), which is also shown for comparison. The relative error

$$\epsilon = \left| \frac{H_{\mathrm{GPU}}(\Delta t) - H_{\mathrm{CPU}}(\Delta t)}{H_{\mathrm{CPU}}(\Delta t)} \right| \tag{4}$$

shown in the inset of figure 10 is smaller than one-tenth of a percent.

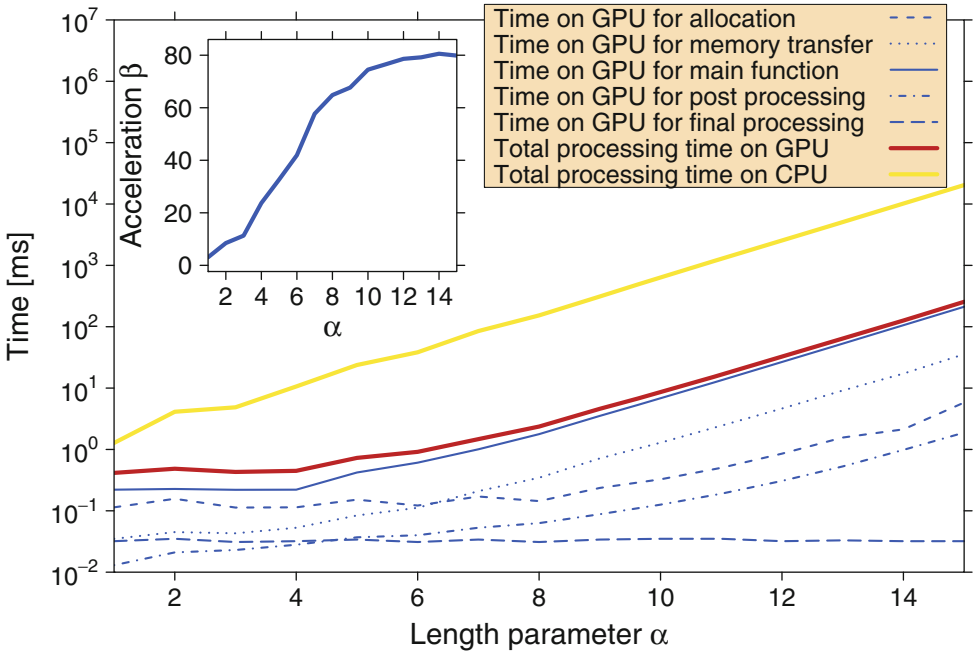

Fig. 8. (Color online) Processing times for the calculation of the Hurst exponent $H(\Delta t)$ on GPU and CPU for $\Delta t_{\max} = 512$. These results are obtained on the GTX 280. The total processing time on GPU can also be slit into allocation time, time for memory transfer, time for main processing, time for post processing, and time for final processing. A maximum acceleration factor of roughly 80 can be reached, which is shown in the inset.

3.2 Equilibrium autocorrelation

The autocorrelation function is a widely used concept for determining dependencies within a time series. The autocorrelation function is given by the correlation between the time series and the time series shifted by the time lag Δt through

$$\rho(\Delta t) = \frac{\langle p(t) \cdot p(t+\Delta t)\rangle - \langle p(t)\rangle\langle p(t+\Delta t)\rangle}{\sqrt{\langle p(t)^2\rangle - \langle p(t)\rangle^2}\sqrt{\langle p(t+\Delta t)^2\rangle - \langle p(t+\Delta t)\rangle^2}}. \tag{5}$$

For a stationary time series, Eq. (5) reduces to

$$\rho(\Delta t) = \frac{\langle p(t) \cdot p(t+\Delta t)\rangle - \langle p(t)\rangle^2}{\langle p(t)^2\rangle - \langle p(t)\rangle^2}, \tag{6}$$

as the mean value and the variance stay constant for a stationary time series, i.e., $\langle p(t)\rangle = \langle p(t+\Delta t)\rangle$ and $\langle p(t)^2\rangle = \langle p(t+\Delta t)^2\rangle$.

It can be observed that the autocorrelation function of price changes for a financial time series exhibits a significant negative value for a time lag of one tick, whereas it vanishes for time lags Δt>1. Furthermore, the autocorrelation of absolute price changes or squared price changes, which is related to the volatility of the price process, decays slowly [85,86]. In order to implement Eq. (6) on a GPU architecture, steps similar to those covered in Sec. 3.1 are necessary. The calculation of the time lag dependent part $\langle p(t) \cdot p(t+\Delta t)\rangle$ is handled analogously to the determination of the Hurst exponent on the GPU. The input time series, which is transferred to the GPU's main memory, does not contain prices but price changes. However, in addition one

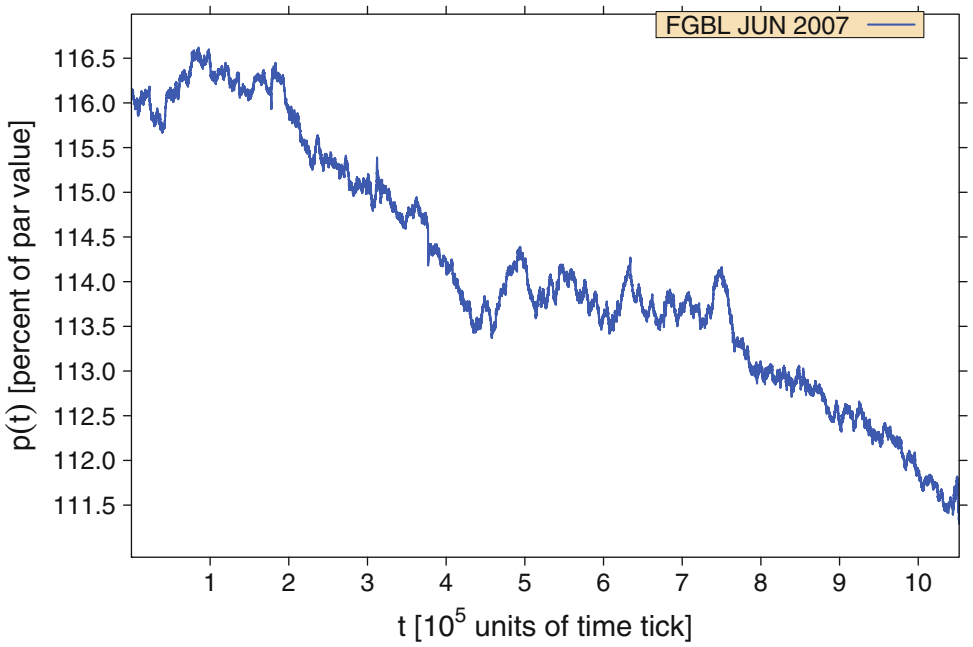

Fig. 9. (Color online) High frequency financial time series of the Euro-Bund futures contract (FGBL) with expiration time June 2007 traded at the European Exchange (Eurex). The time series contains 1,051,982 trades recorded from 8 March 2007 to 7 June 2007. The price is shown in units of percent of par value. In bond markets, the par value (as stated on the face of the bond) is the amount that the issuing firm is to pay to the bond holder at the maturity date. In calculations of the FGBL time series on a GPU presented here, α is fixed to 11. Thus, the data set is limited to the first $T = 1,049,088$ trades in order to fit the data set length to the constraints of the specific GPU implementation.

needs the results for $\langle p(t) \rangle$ and $\langle p(t)^2 \rangle$. For this purpose, an additional array of length T is allocated, in which a GPU kernel function stores the squared values of the time series. Then, time series and squared time series are decomposed with the same binary tree reduction process as in Sec. 3.1. However, as this procedure produces arrays of length $\Delta t_{\max}$, one has to sum these values in order to obtain $\langle p(t) \rangle$ and $\langle p(t)^2 \rangle$.

The processing times for determining the autocorrelation function for $\Delta t_{\max} = 256$ on a CPU and a 8800 GT can be found in Fig. 11. Here we find that allocation and memory transfer dominate the total processing time on the GPU for small values of α and thus, only a fraction of the maximum acceleration factor $\beta \approx 33$ (shown as an inset) can be reached. Using the consumer graphics card GTX 280, we obtain a maximum speed-up factor of roughly 55 for $\Delta t_{\max} = 256$ and 68 for $\Delta t_{\max} = 512$ as shown in Fig. 12. In Fig. 13, the autocorrelation function of the FGBL time series is shown. For a time lag of one, the time series exhibits a large negative autocorrelation, $\rho(\Delta t = 1) = -0.43$. In order to quantify deviations between GPU and CPU calculations, the relative error ϵ is presented in the inset of Fig. 13. Note that small absolute errors can cause relative errors of up to three percentage point because the values $\rho(\Delta t > 1)$ are close to zero.

For some applications, it is interesting to study larger maximum time lags of the autocorrelation function. To do this on the basis of our GPU implementation, the program code must be modified in the following way. So far, each thread was responsible for a specific time lag Δt. In a modified implementation, each thread would be responsible for more than one time lag in order to realize a maximum time

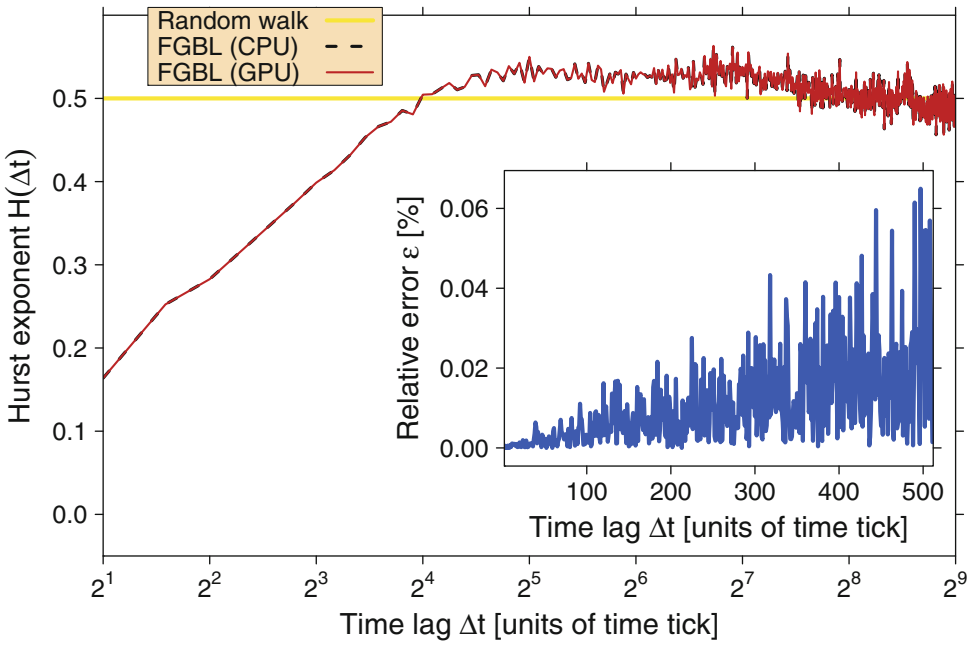

Fig. 10. (Color online) Hurst exponent $H(\Delta t)$ in dependence of time lag Δt calculated on CPU and GPU. Additionally, the theoretical Hurst exponent of a random walk process ($H = 0.5$) is included for comparison. One can clearly see the well-known anti-persistent behavior of the FGBL time series on short time scales ($\Delta t < 2^4$ time ticks). Furthermore, evidence is given that the process reaches a slightly super-diffusive region ($H \approx 0.525$) on medium time scales (2^4 time ticks $< \Delta t < 2^7$ time ticks). On long time scales, an asymptotic random walk behavior can be found. In order to quantify deviations from calculations on a CPU, the relative error ϵ (see main text) is presented for each time lag Δt in the inset. It is typically smaller than 10^{-3}.

lag, which is a multiple of the maximum number of 512 threads per block. This way, one obtains a maximum speed-up factor of roughly 84 for $\Delta t_{\max} = 1024$ using the GTX 280. Further time series analysis methods will be ported to a GPU architecture in the future including switching point analysis [1,87] and multivariate time series analysis [88] as well as multi-agent based models [89].

4 Accelerated simulations in statistical physics

In the previous section, a GPU approach successfully applied in order to accelerate time series analyses. We now port a standard model of statistical physics – the Ising model – to the GPU architecture. The Ising model, which Ernst Ising studied in his PhD [90], is a standard model of statistical physics and provides a simplified microscopic description of ferromagnetism. It was introduced to explain the ferromagnetic phase transition from the paramagnetic phase at high temperatures to the ferromagnetic phase below the Curie temperature T_C. A large variety of techniques and methods in statistical physics were originally formulated in terms of the Ising model and then generalized and adapted to related models and problems [91]. Supported by his results for a one dimensional spin chain, in which no phase transition occurs, Ising initially proposed in his PhD thesis that there is also no phase transition in higher dimensions which turned out to be a mistake. The Ising model on a two dimensional

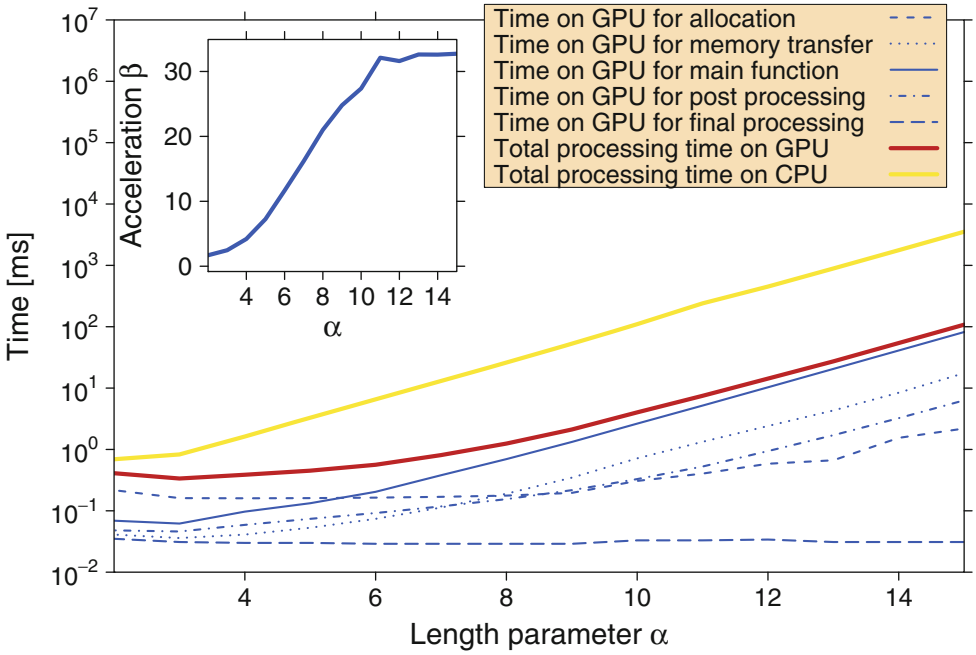

Fig. 11. (Color online) Processing times for the calculation of the equilibrium autocorrelation function $\rho(\Delta t)$ on GPU and CPU for $\Delta t_{\max} = 256$. The graphics card 8800 GT is used as GPU device. The total processing time on GPU is broken into allocation time, time for memory transfer, time for main processing, time for post processing, and time for final processing. The acceleration factor β is shown as inset. A maximum acceleration factor of roughly 33 can be obtained.

square lattice with no magnetic field was later analytically solved by Lars Onsager in 1944 [92]. The critical temperature at which a second order phase transition between an ordered and a disordered phase occurs can be determined analytically for the two dimensional model ($T_C = 2.269185$ [92]). Despite much effort, an analytic solution for the three dimensional Ising model still remains one of the great challenges in statistical physics. However, computer simulations in combination with finite-size scaling techniques [93–96] are able to determine $T_C \sim 4.5115$ [91] and the rest of the phase diagram with good accuracy. Starting in 1944, the Ising model not only became popular in main stream physics but also in various interdisciplinary fields, in particular in econophysics [1,97,98].

Critical phenomena, scaling and universality properties of Ising models have been studied via Monte Carlo simulations for decades [91] with continuously improving accuracy, these studies having directly benefited from the increasing availability of computational resources. Such computing requirements, necessary not only for Monte Carlo simulations but also for various other tasks in computational physics including, e.g., molecular dynamics simulations [3,4,17,99] need a large amount of high performance computing resources.

We employ the general purpose graphics processing unit technology for Monte Carlo simulations of a two dimensional square lattice and a three dimensional cubic lattice Ising models. Here we use a GeForce GTX 280 as GPU device if no other graphics card is specified. We are using CUDA (version 2.0) in combination with an NVIDIA graphics card driver (driver version 177.73). Early versions of the graphics card drivers for the NVIDIA Geforce GTX 200 series exhibit a bug if a large part

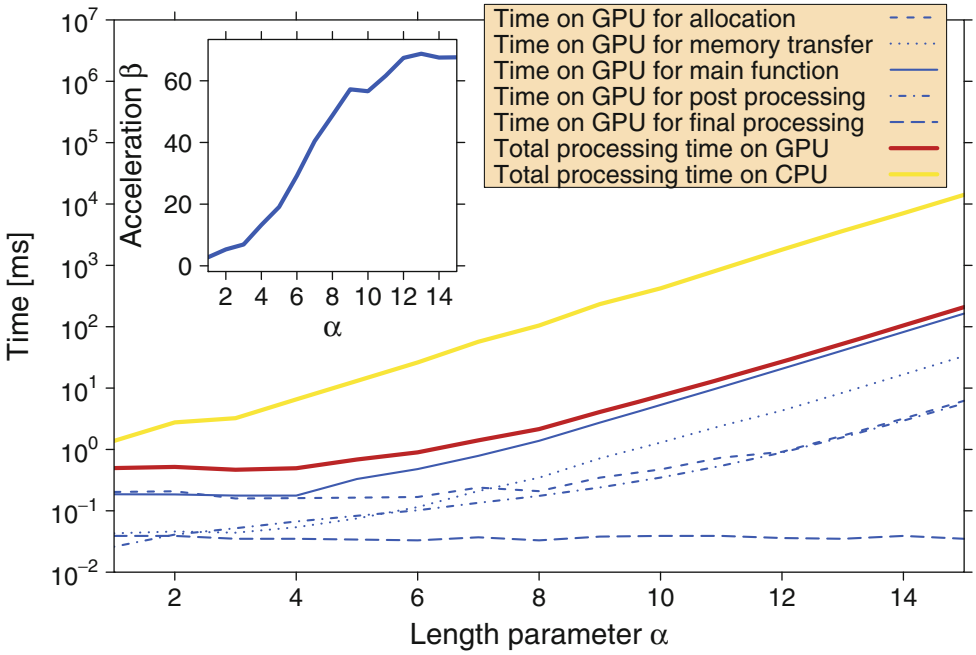

Fig. 12. (Color online) Processing times for the calculation of the equilibrium autocorrelation function $\rho(\Delta t)$ on GPU and CPU for $\Delta t_{\max} = 512$. The GTX 280 is used as GPU device. A maximum acceleration factor of roughly 68 can be obtained.

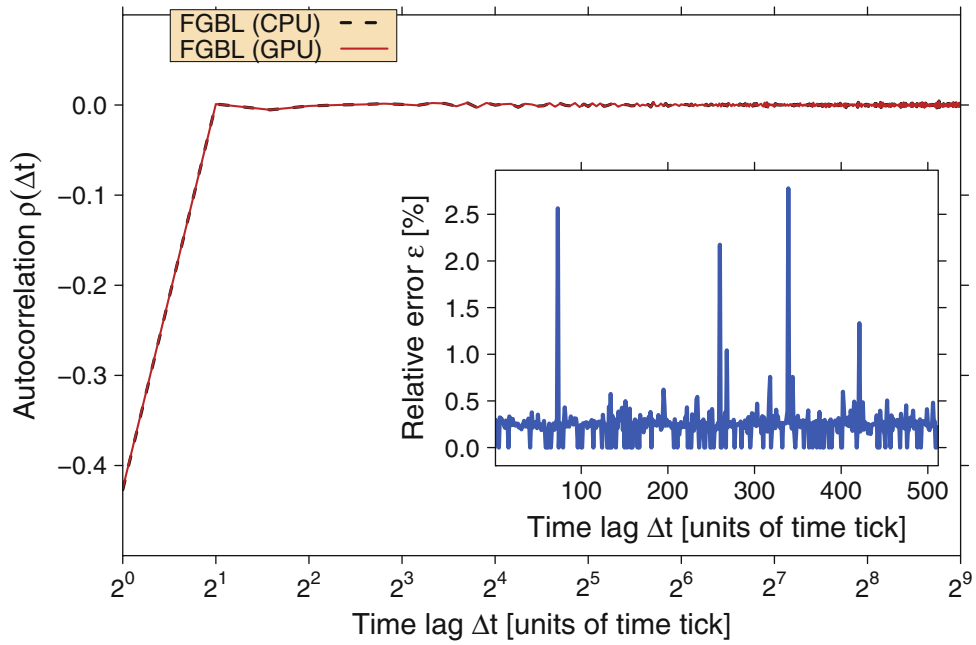

Fig. 13. (Color online) Equilibrium autocorrelation function $\rho(\Delta t)$ in dependence of time lag Δt calculated on CPU and GPU. One can clearly see the well-known negative autocorrelation of financial time series at time lag $\Delta t = 1$ also in this figure. In order to quantify deviations from calculations on a CPU, the relative error ϵ is presented for each time lag Δt in the inset. The relative error is always smaller than 3×10^{-2}.

of provided global memory is used. However, this problem has recently been fixed. For the comparison between CPU and GPU implementations, we use an Intel Core 2 Quad CPU (Q6700) with 2.66 GHz and 4096 kB cache size, of which only one core is used. The standard C source code executed on the host is compiled with the `gcc` compiler with option `-O3` (version 4.2.1). Other compilers in combination with more sophisticated compiler options lead to even better processing times on the CPU. In [19], a GPU based version of the Ising model has already been proposed. This implementation was able to accelerate the Ising model computation by a factor of three by migration to a GPU. In this article, we will demonstrate that a better acceleration factor can be obtained. We demonstrate that our implementation works by calculating the critical temperatures of the phase transitions in the two and three dimensional Ising models.

4.1 Random number generation

An efficient method for creating random numbers is essential for Monte Carlo simulations of the two dimensional ferromagnetic square lattice and the three dimensional ferromagnetic cubic lattice Ising model on a GPU in Sec. 4.2 and Sec. 4.3. For this purpose, we use an array of linear congruential random number generators (LCRNGs) applying one of the oldest and best-known algorithms for generation of pseudo random numbers [5]. Starting at a seed value $x_{0,j}$, a sequence of random numbers $x_{i,j}$ with $i \in \mathbb{N}$ of the LCRNG j can be obtained by the recurrence relation

$$x_{i+1,j} = (a \cdot x_{i,j} + c) \mod m \tag{7}$$

where a, c, and m are integer coefficients. An appropriate choice of these coefficients in Eq. (7) is responsible for the quality of the produced random numbers. We use $a = 1664525$ and $c = 1013904223$ as suggested in [5]. In order to exploit the local 32-bit architecture provided by the GPU device, the parameter of the modulo operation m is set to 2^{32} as, by construction, results on a 32-bit architecture are truncated to the endmost 32 bits. Thus, if using signed integers, pseudo random numbers $x_{i,j} \in [-2^{31}; 2^{31} - 1]$ can be obtained, which have to be normalized according to $y_{i,j} = \mathrm{abs}\left(x_{i,j}/2^{31}\right) \sim 4.656612 \cdot 10^{-10} \mathrm{abs}\left(x_{i,j}\right)$ in order to get uniformly distributed pseudo random numbers $y_{i,j}$ in the interval $[0; 1]$. As we use an entire set of linear congruential random number generators in parallel, each LCRNG j of this array is initialized by a random number obtained from a further LCRNG through

$$x_{0,j+1} = (16807 \cdot x_{0,j}) \mod m \tag{8}$$

with $x_{0,0} = 1$.

In the GPU implementation, each thread of a thread block handles its own linear congruential random number generator. Denoting s as the number of involved thread blocks and denoting σ as the number of threads per block, in a GPU kernel, values of $s \cdot \sigma$ LCRNGs are determined in parallel. Each LCRNG calculates S pseudo random numbers. Thus, a total of $S \cdot s \cdot \sigma$ pseudo random numbers are created. In Fig. 14, a comparison of processing times of the random number generation between calculation on a GPU device and on a CPU core is presented for $S = 10^4$ and $\sigma = 512$ depending on the number of involved blocks s. On the CPU, the same $S \cdot s \cdot \sigma$ pseudo random numbers are used for the sake of comparison. The acceleration factor β, which is shown in the inset, is determined by the relationship

$$\beta = \frac{\text{Total processing time on CPU}}{\text{Total processing time on GPU}} \tag{9}$$

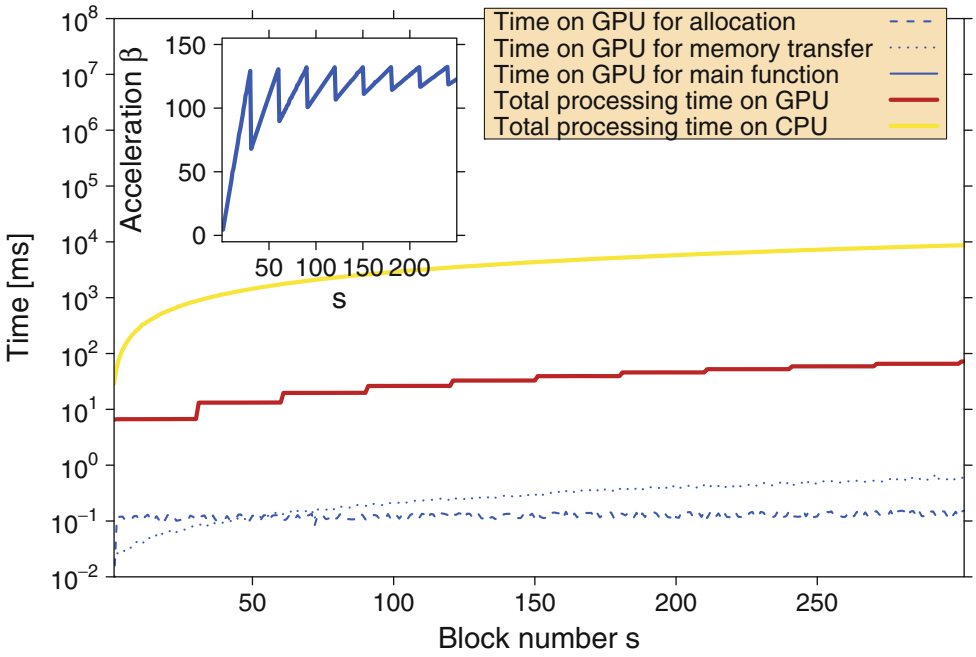

Fig. 14. (Color online) Processing times for the calculation of $S \cdot s \cdot \sigma$ pseudo random numbers on GPU and CPU for $S = 10^4$ and 512 threads per block in dependence of the number of involved thread blocks s. The total processing time on GPU is divided into allocation time, time for memory transfer, and for main processing. The acceleration factor β is shown as inset. A maximum acceleration factor of roughly 130 can be obtained for $s = 30$, which corresponds to the number of multiprocessors on our GPU device. Note that the computation on the GPU becomes inefficient if s is not a multiple of the number of multiprocessors on the gpu device because some multiprocessors are idle in the end of the calculation (see inset).

as defined in one of the previous sections. A maximum acceleration factor $\beta \sim 130$ is obtained for $s = 30$, which corresponds to the number of multiprocessors of the NVIDIA GeForce GTX 280 consumer graphics card. Up to 30 groups of $\sigma = 512$ single linear congruential random number generators can be executed concurrently at the same time on this GPU device. If s is larger than 30, the 30 available multiprocessors are able to execute only the first 30 blocks in parallel, such that the remaining groups of $\sigma = 512$ single linear congruential random number generators have to be handled in a second step. Thus, processing time on the GPU device is doubled when $s = 31$ is used instead of $s = 30$. Note that the computation on the GPU becomes inefficient if s is not a multiple of the number of multiprocessors on the gpu device because some multiprocessors are idle at the end of the calculation (see inset of Fig. 14). The fraction of GPU processing time for allocation and memory transfer of the LCRNG seed values can be neglected. Please note that there are many other possibilities for implementing an efficient creation of pseudo random numbers on a GPU device. In [3], e.g., an algorithm is presented for producing a set of pseudo random numbers in parallel on a GPU device with a single linear congruential random number generator which determines the set of numbers according to a serial rule. A CUDA based version of the Mersenne Twister random number generator [100] is able to generate pseudo random numbers in parallel as well.

4.2 Two dimensional Ising model

In this section, we present an implementation of the two dimensional ferromagnetic square lattice Ising model on a GPU which was originally introduced in [27]. The simple Ising model, which is one of the simplest lattice models, consists of an arrangement of spins with are located on the sites of the lattice and which exhibit only the values $+1$ and -1 [2]. These spins interact with their nearest neighbors on the lattice with interaction constant $J > 0$. The Hamiltonian $\mathscr{H}$ for this model is given by

$$\mathscr{H} = -J \sum_{\langle i,j \rangle} S_i S_j - H \sum_i S_i \tag{10}$$

where $S_i = \pm 1$ represents a spin at site i and H denotes the coupling to an external magnetic field. We use a square lattice with n spins per row and column and periodic boundary conditions. The lattice contains $N = n^2$ spins. For the spin update, the Metropolis criterion [101] is applied which is an obvious choice for a transition probability satisfying the 'detailed balance' principle in the thermal equilibrium and leading to the fastest dynamics for single spin flip simulations. In the expression $P_a(t)W_{a\to b} = P_b(t)W_{b\to a}$ for 'detailed balance', $P_a(t)$ denotes the probability of the system being in state a at time t and $W_{a\to b}$ the transition rate for the move $a \to b$. Denoting $\Delta\mathscr{H}$ to be the energy difference between the two states a and b, which is given through $\Delta\mathscr{H} = \mathscr{H}_b - \mathscr{H}_a$, the probability for the move $a \to b$ is given by $W_{a\to b} = \exp(-\Delta\mathscr{H}/k_B T)$ if $\Delta\mathscr{H} > 0$ and by $W_{a\to b} = 1$ if $\Delta\mathscr{H} \leq 0$. As a single spin flip dynamics is applied, two successive states differ only by a single spin, such that $\Delta\mathscr{H}$ can be calculated as a local energy difference.

In the first step of the GPU implementation, one has to allocate memory on the GPU device's global memory for the two dimensional spin field and for the seed values of the LCRNGs. After a random initialization of the spin field on the CPU and initialization of seed values as described in Sec. 4.1, spins and seeds are transferred to the GPU's global memory. The transition probability $W_{a\to b}$ depends on the temperature T which has to be passed to the GPU kernel function. The Boltzmann constant k_B is fixed to 1 in all simulations. We use a zero field ($H = 0$) and $J = 1$.

It must be taken into account that for Monte Carlo trial moves to be executed in parallel, the system has to be partitioned into non-interacting domains. Thus, for the update process of the spin lattice, a checkerboard algorithm is applied, that is there is a regular scheme for updating the lattice spin by spin. First all spins on the "white" squares of the checkerboard are updated and then all spins on the "black" squares. Please note that other methods for this updating process are also available (e.g., diverse cluster algorithms [102,103], which exhibit faster convergence). However, the systematic scheme of the checkerboard algorithm is most suitable for demonstrating the migration to the GPU architecture realizing noninteracting domains where the Monte Carlo moves are performed in parallel. In fact, this partitioning for "real life" problems is a significant challenge.

On a GPU device, it is only possible to synchronize threads within a block. A native block synchronization does not exist. Therefore, only the termination of the GPU kernel function ensures that all blocks were executed. We divide the spin update process on the lattice into blocks on the GPU. In a single GPU kernel, only a semi-lattice can be changed without creating conflicts.

The spin updating process is subdivided into threads and blocks as illustrated in Fig. 15 for $n = 16$. The spin field is divided into strips of width 2. Each strip is handled by one block and each thread is responsible for a square sub-cell of 2×2 spins in order to avoid idle threads. Thus, $n/2$ threads per block are necessary. A first GPU kernel handles the update process of the first semi-lattice, i.e., in each block

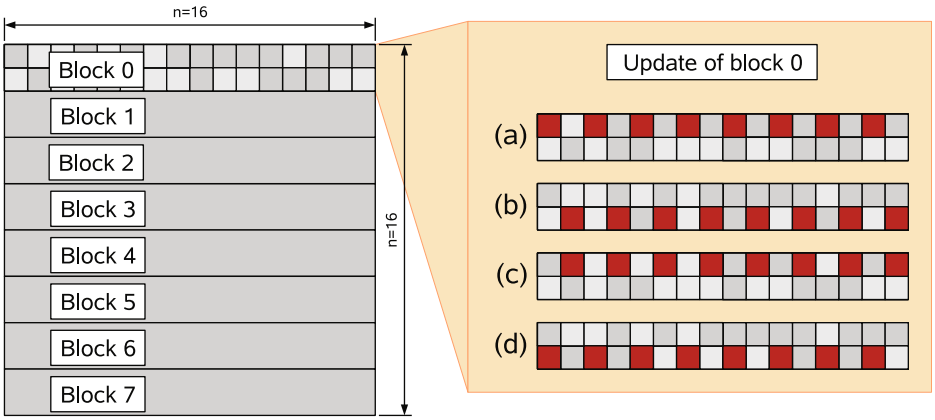

Fig. 15. (Color online) Schematic visualization of the implementation of a two dimensional Ising model on a GPU for $n = 16$ spins per row. The 2D spin field is split in strips of 16×2 spins each treated by a single block on the GPU. In each block, we use $n/2 = 8$ threads, which are used for the spin update corresponding to the scheme presented on the right side and described in the main text.

$\sigma = n/2$ threads accomplish steps (a) and (b) of Fig. 15. The termination of this GPU kernel ensures that all blocks were executed and a second kernel can start in order to accomplish steps (c) and (d). Each sub-cell (i.e., each thread in each block) has access to its own LCRNG. As each thread in one GPU kernel function needs up to two random numbers and as this array is also used after two semi-lattice updates for the reduction process of partial energies or magnetizations of the Ising lattice, the seed values or current random numbers are transferred at the beginning of a GPU kernel to a shared memory array in order to take advantage of the increased memory speed. After finishing of the updating steps (steps (b) and (d)) the current value is transferred back to global memory. As CUDA does not provide native reduction functions for treating partial results, this must be managed manually. For this purpose, the shared memory array is used after step (d). A binary tree structure ensures a fast reduction of the partial values within a block. These partial results of each block are stored at block-dependent positions in global memory and finally transferred back to the host's main memory [30]. The final summation of the results of $n/2$ blocks is carried out by the CPU. In Fig. 16, the processing times of the Ising model implementation on the GPU are compared with an Ising model implementation on one CPU core. For the largest system, $n = 1024$, an acceleration factor of roughly 60 could be achieved. For very small systems, the acceleration factor is smaller than 1 because the GPU device is not used efficiently for small thread numbers per block.

In order to verify the GPU implementation, it is insufficient to only compare values of energy and magnetization of a function of the temperature T between GPU and CPU versions. A very sensitive test is given by the determination of the critical temperature of the Ising model. For this purpose, we use finite size scaling and calculate the Binder cumulant [93], which is given in a zero field by

$$U_4(T) = 1 - \frac{\langle M(T)^4 \rangle}{3\langle M(T)^2 \rangle^2}, \tag{11}$$

with M denoting the magnetization of a configuration at temperature T and $\langle \ldots \rangle$ being the thermal average. Near a critical point, finite size scaling theory predicts the free energy and derived quantities such as magnetization being functions of linear dimension L over correlation length $\xi \simeq (T - T_c)^{-\nu}$. Therefore, moment ratios of

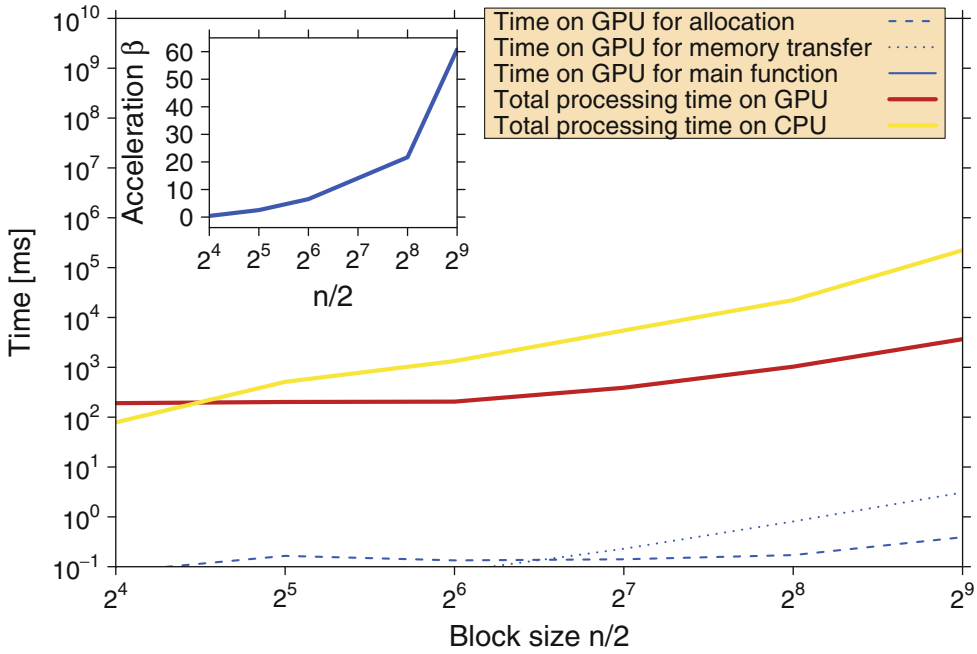

Fig. 16. (Color online) Processing times for a two dimensional ferromagnetic square lattice Ising model for a cooling down process. The temperature $T \in [2.0; 3.0]$ is stepwise reduced by the factor 0.99. In each temperature step, 100 sweeps are performed. The processing times are shown in dependence of the number of threads per block which is related to the system size by $\sigma = n/2$. The total processing time on GPU is divided into allocation time, time for memory transfer, and time for main processing. The acceleration factor β is shown as inset. A maximum acceleration factor of roughly 60 can be realized.

the magnetization like for example the Binder cumulant U_4, become independent of system size at the critical temperature. To test our implementation, we perform several simulations close to the critical point for different linear dimensions of the simulation box and determine U_4. As shown in Fig. 17, the curves of the Binder cumulants for various system sizes $N = n^2$ cross almost perfectly at the critical temperature derived by Onsager ($T_c \approx 2.269185$) [92], indicated by a dashed line. Note that single spin flips and parallelization schemes based upon them are not particularly well-suited for the determination of critical properties because of critical slowing down. Nevertheless, we are able to determine T_c with reasonable accuracy ($T_c = 2.2692 \pm 0.0002$).

4.3 Three dimensional Ising model

In this section, the GPU implementation of the two dimensional ferromagnetic square lattice Ising model is expanded to a three dimensional cubic lattice model version on the GPU. In a first step, we allocate memory analogously as in Sec. 4.2. In addition, the three dimensional spin field with $N = n^3$ spins and the random number seeds have to be transferred to the GPU device. In the three dimensional case, the spin update process is also subdivided into threads and blocks. Now, the update scheme illustrated in Fig. 18 for $n = 16$ is applied. The 3D spin field is split in cuboids of $2 \times 2 \times 16$ spins each treated by a single block on the GPU. In each block, we use $n/2 = 8$ threads, which are used for the spin update corresponding to the scheme

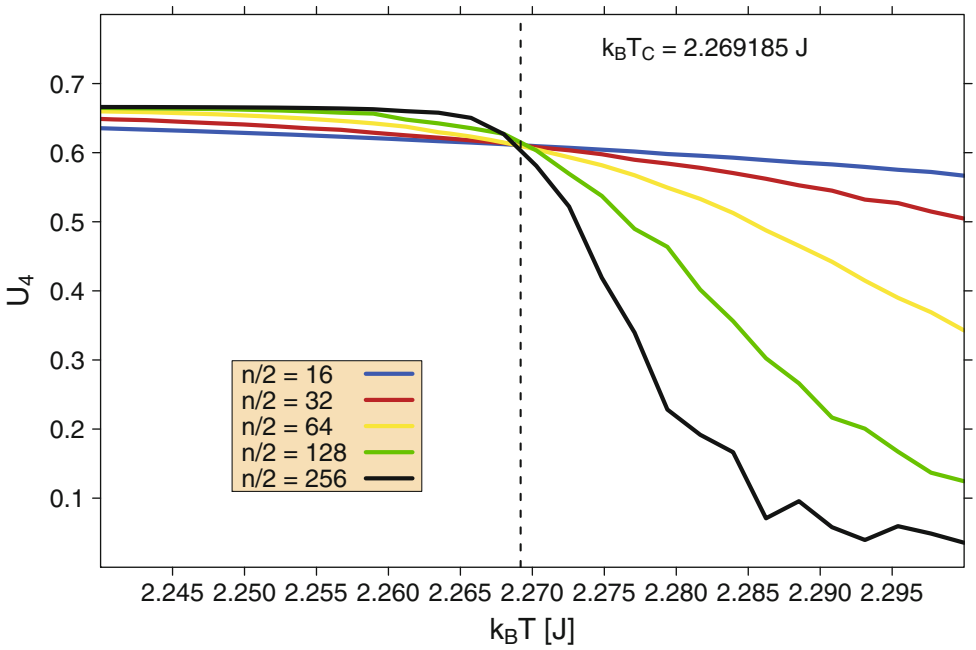

Fig. 17. (Color online) Binder cumulant U_4 in dependence of k_BT for various numbers n of spins per row and column of the two dimensional square lattice Ising model. $n/2$ corresponds to the involved number of threads per block on the GPU implementation. The curves of the Binder cumulants for various system sizes $N = n^2$ cross almost perfectly at the critical temperature derived by Onsager [92], which is shown additionally as a dashed line. In each temperature step, the average was taken over 10^7 measurements.

presented on the right side of Fig. 18. A first GPU kernel handles the update process of the first semi-lattice, i.e., in each block $\sigma = n/2$ threads accomplish steps (a), (b), (c), and (d) of Fig. 18. The termination of this GPU kernel ensures that all blocks were successfully executed and a second kernel can start in order to carry out steps (e), (f), (g), and (h). Each sub-cell (i.e., each thread in each block) which is responsible for 8 spins in the 3D case has access to its own LCRNG. In Fig. 19, the processing times of the three dimensional Ising model implementation on the GPU are compared with a corresponding Ising model implementation on a single CPU core. The largest system which can be realized on a GeForce GTX 280 in this way is $n = 256$. Global memory size limits the Ising system size to this value. Thus, a maximum acceleration factor of roughly 35 can be achieved.

In Fig. 20, the Binder cumulant for different system sizes $N = n^3$ is presented as a function of temperature T. The crossing point at $T_C \sim 4.51$ is in good agreement with previous simulation results [91], according to which the critical temperature is located at 4.5115 [95] and 4.5103 [96]. These values are shown as dashed lines in Fig. 20.

4.4 Multi-spin coded 2D CPU implementation

For our extended CPU reference implementation[2], we focus on a single spin-flip approach which performs well for large lattice sizes. Multi-spin coding refers to all

[2] This extension using multi-spin coding was developed in collaboration with Benjamin Block and Peter Virnau and is published in [24].

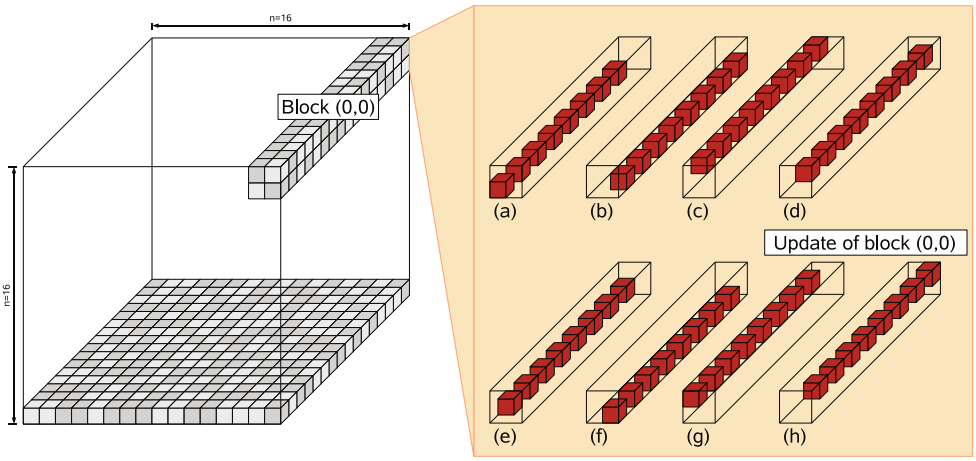

Fig. 18. (Color online) Schematic visualization of the three dimensional ferromagnetic cubic lattice Ising model implementation on a GPU for $n = 16$. The 3D spin field is split in cuboids (shown left) of $2 \times 2 \times 16$ spins each treated by a single block on the GPU. In each block, we use $n/2 = 8$ threads, which are used for the spin update corresponding to the scheme presented (shown right) and described in the main text.

techniques that store and process multiple spins in one unit of computer memory. In CPU implementations, update schemes have been developed that allow for processing more than one spin with a single operation [104–107]. We use a scheme which encodes 32 spins into one 32-bit integer in a linear fashion. The 32-bit type is chosen because register operations of current hardware perform fastest on this data type. The key ingredient for an efficient update algorithm of these 32-bit patterns is to use pre-computed bit patterns that encode the evaluations of the flip condition expression

$$r < \exp(-\Delta\mathscr{H}/k_B T) \tag{12}$$

for every single spin bit – the variate r is an independent and identically distributed random number in $[0, 1)$. Since there are only two possible energy differences $\Delta\mathscr{H}$ with $\Delta\mathscr{H} > 0$, two *boolean arrays* can encode the information of an evaluation of the flip condition. For reasonable results, N. Ito [107] suggested to use a pool of 2^{22} to 2^{24} Boltzmann patterns. We denote the two Boolean arrays "exp4" and "exp8". Our encoding is chosen such that a "1" is stored in exp4 if $\exp(-8J/k_B T) < r < \exp(-4J/k_B T)$ is satisfied and a "0" if not. A "1" is stored in exp8 if $r < \exp(-8J/k_B T)$ is satisfied and a "0" if not. For every spin update, the Monte Carlo simulation will choose a random pattern. To process a 32-bit spin pattern s_0, neighbor patterns s_1, s_2, s_3, s_4 have to be prepared which contain the neighbors of the i^{th} spin at their i^{th} digit.

Since the calculation is the same for every bit, it is convenient to look at just one bit. The first step is to transform the spin variables into "energy variables" to eliminate dependence on the initial state of s_0.

$$i_n = s_0 \hat{} s_n, \forall n \in \{1, 2, 3, 4\} \tag{13}$$

where "$\hat{}$" denotes an XOR operation. Because of the special encoding of the Boltzmann patterns, the acceptance condition for each spin can be expressed in a simple way:

$$i_1 + i_2 + i_3 + i_4 + 2 \cdot \exp8_s + \exp4_s \geq 2 \tag{14}$$

where $\exp8_s$ and $\exp4_s$ denote the sth Boltzmann patterns that encode the spin flip condition, and s is a random position in the pool. It is possible to evaluate

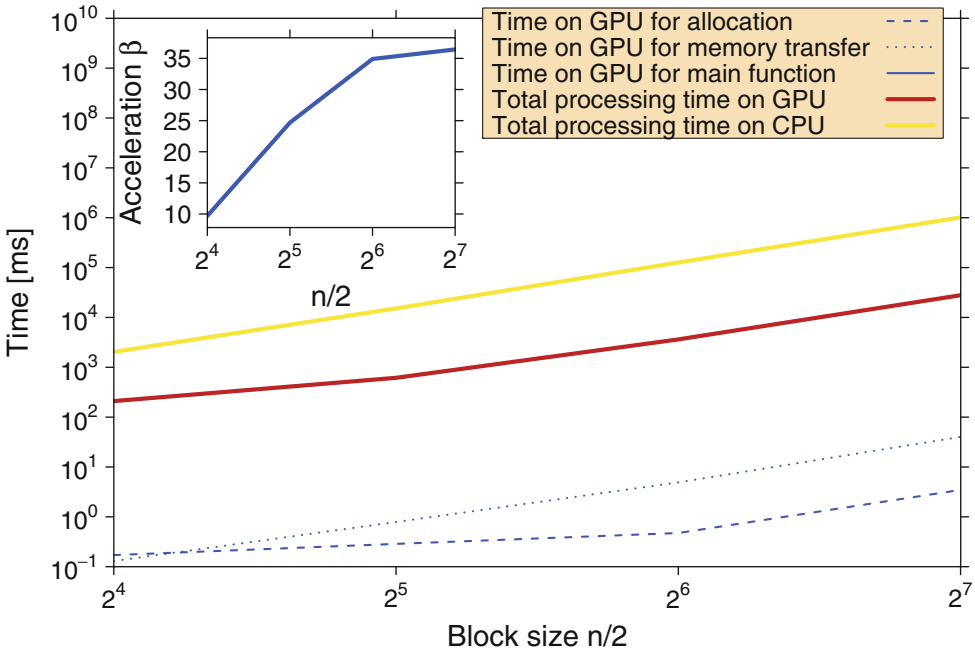

Fig. 19. (Color online) Processing times for a three dimensional ferromagnetic cubic lattice Ising model for a cooling process. The temperature $T \in [4.0; 5.0]$ is gradually reduced by the factor 0.99. In each temperature step, 100 sweeps are performed. The processing times are shown in dependence of the number of threads per block which is related to the system size by $\sigma = n/2$. The total processing time on the GPU is split into allocation time, time for memory transfer, and time for main processing. The acceleration factor β is shown as inset. A maximum acceleration factor of roughly 35 can be achieved.

this expression for all 32 bits in parallel by applying a sequence of bitwise boolean operations [107].

As parallel updates are only allowed on non-interacting domains, an additional bit mask has to be applied to the update pattern that only allows to flip each second spin in parallel.

4.5 Multi-spin coded GPU implementation

4.5.1 Problems arising from a straightforward adaption

On GPUs, memory access is very costly compared to operations on registers. The great advantage of multi-spin coding is that only one memory access is needed to load several spins in parallel. The CPU implementation could be ported to a GPU with a kernel that uses less than 16 registers. This allows optimal usage of the GPU up to a maximum block size of 512 threads. Even though the update scheme presented in Sec. 4.4 performs faster on the CPU compared to an implementation with integer representations of each spin, a straightforward GPU port of this scheme is not optimal.

The reason for the poor performance is that parallel threads in one warp have to access global memory in a random fashion which is very costly. The execution speed can be improved by choosing only one random position per block and letting all the threads in this block read the patterns linearly, starting from the chosen starting position. However, this approach reduces the quality of the flip patterns. In principle, this ansatz could be compensated for by using a significantly larger pool

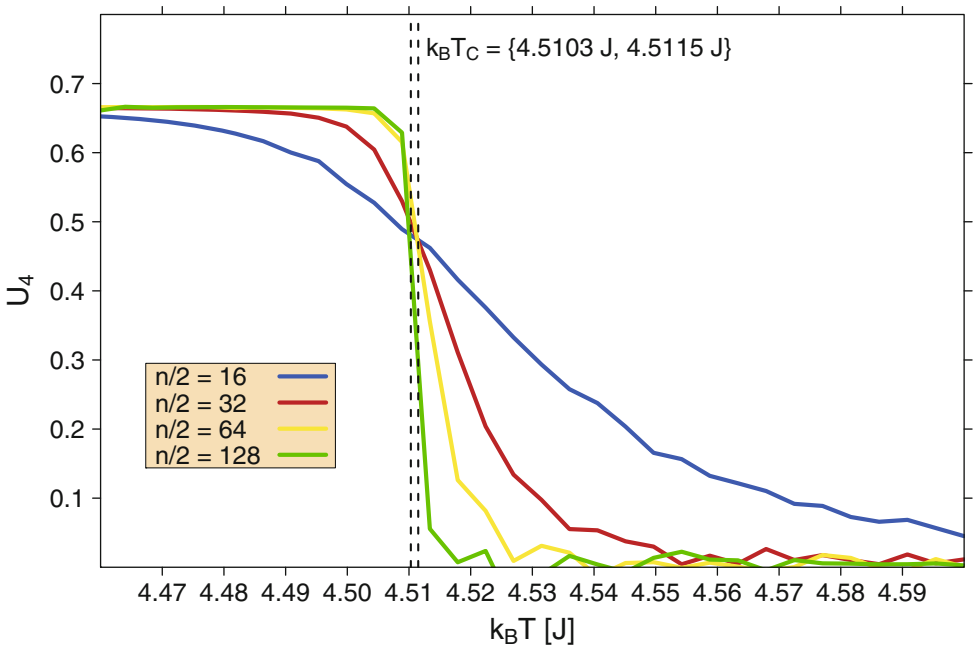

Fig. 20. (Color online) Binder cumulant U_4 in dependence of k_BT for various numbers n of spins per row and column of the three dimensional Ising model on the simple cubic lattice. $n/2$ corresponds to the involved number of threads per block on the GPU implementation. The curves of the Binder cumulant for various system sizes $N = n^2$ cross almost perfectly at the critical temperature $T_C \approx 4.51$, which is in agreement with a selection of previous simulation results which are presented as dashed lines. For each temperature step, the average was taken over 10^6 measurements.

Table 2. Key facts and properties of the Tesla C1060 GPU[108].

	Tesla C1060
Global memory	4096 MB
Streaming processor cores	240
Shared memory per block	16 KB
Clock rate	1.30 GHz
Memory clock	800 MHz
Maximal power consumption	187.8 W

of random numbers. Another option is to calculate the spin flip patterns on the fly using a random number generator on the GPU instead of looking them up from global memory. It turns out, however, that the sophisticated update scheme does not longer benefit in this case. The performance of this implementation is compared in Table 3. It should be emphasized that the quality of random numbers differs between the implementations. In the next section, we present another update scheme that works well on the GPU, and which prevents pitfalls with the quality of the random numbers.

In this section, we use a NVIDIA Tesla C1060 as our CUDA enabled device, which offers 4 GB of global memory, see Table 2. This memory can store a multi-spin coded spin field of $100,000^2$ spins on one GPU. The reference CPU used in tests in Secs. 4.4, 4.5, and 4.6 is the Intel Xeon X5560 with a clock rate of 2.80 GHz and 8192 kB cache. The purpose of the CPU implementation is to have a fast and fair

Table 3. Comparison at lattice size 4096 × 4096: *CPU simple* encodes one spin in one integer, *CPU multi-spin coding* uses the efficient "multi-spin" update scheme presented in section 4.4, *multi-spin unmodified* is a straightforward porting of this update to the GPU, *multi-spin coding on the fly* uses the same scheme but calculates the update patterns at each update step on the fly, and *multi-spin coding linear* determines one starting position in the random number pattern in the pool per block, and lets the threads read the random numbers linearly from that position on. The *shared memory* implementation (see Sec. 4.5.2) provides random numbers with a better quality.

	Spinflips per μs	Relative speed
CPU simple	26.6	0.11
CPU multi-spin coding	226.7	1.00
shared memory	4415.8	19.50
multi-spin unmodified	3307.2	14.60
multi-spin coding on the fly	5175.8	22.80
multi-spin coding linear	7977.4	35.20

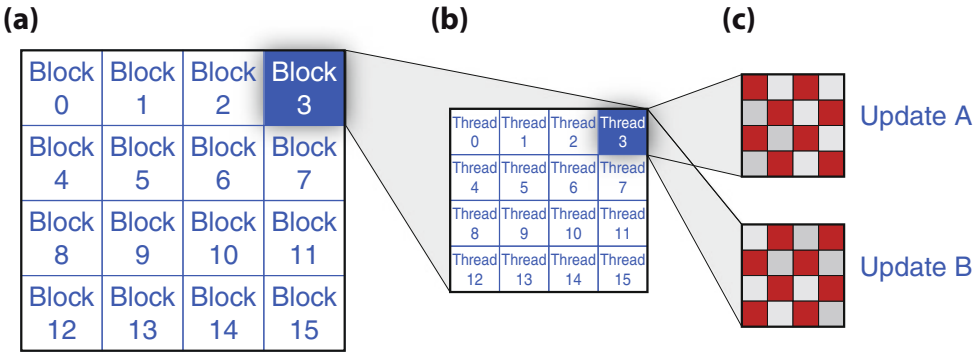

Fig. 21. (Color online) The spin lattice is processed by a variable number of blocks (a), where each block runs a variable number of threads (b). The threads update the spin lattice in two steps, A and B, using two kernel invocations (c).

non-parallel reference implementation, not to benchmark a Core i7 CPU. Therefore, only one core of the CPU is used.

4.5.2 Extraction into shared memory

The main goal of the implementation presented here is to reduce access to the global memory of the GPU, which is extremely costly. The best performance without pre-calculating flip patterns can be achieved by extracting the spins into shared memory and performing the calculations on integer registers. The spin field on the graphics card is encoded in quadratic blocks of 4×4 spins (hereafter referred to as "meta-spins") which can be stored as binary digits of one unsigned short integer (*2 bytes*), and thus can be accessed by a single memory lookup. Single spin values can be extracted from one meta-spin by using the expression

$$\mathtt{s[x,y]} = (\mathtt{meta} - \mathtt{spin}\&(1 \ll (\mathtt{y} * 4 + \mathtt{x}))) * 2 - 1$$

which returns a value of either −1 or 1. This is a slightly more complicated expression than for a linear layout, but it makes sense for a multi-GPU implementation in which border information has to be transferred between various GPUs (see Sec. 4.6). This

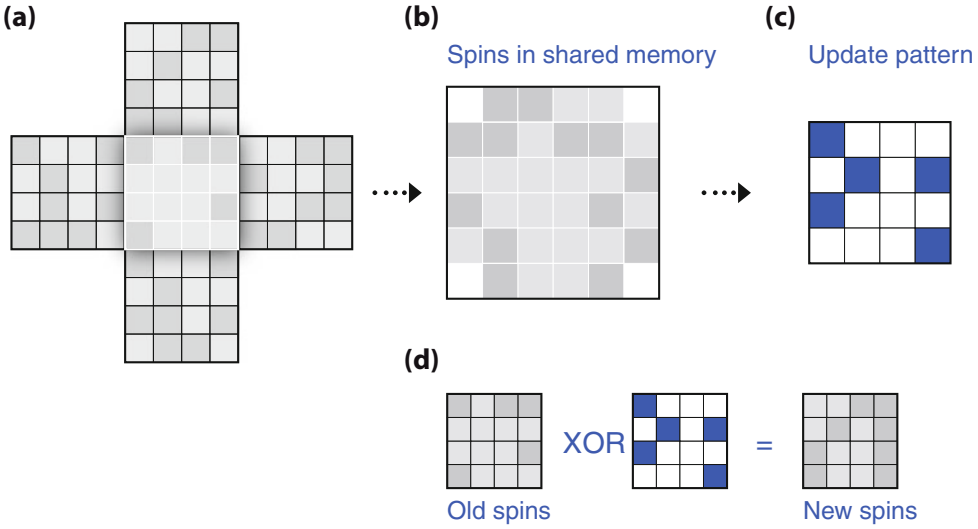

Fig. 22. (Color online) (a) The way a kernel processes a 4×4 meta-spin. (b) Spins are extracted into shared memory and an update pattern is created (c). (d) Afterwards, the new spins are obtained using the update pattern (Spins on blue sites will be flipped, spins on white sites will not be flipped), and written back to global memory.

approach accounts for the fact that each spin uses exactly one bit of memory. The spin field is stored in global memory, which is expensive to access.

To process the spin field on the GPU, the spin field is subdivided into quadratic subfields which can be processed by threads grouped into one block (see Fig. 21). Each thread of this block processes a "meta-spin" of 4×4 spins. At the beginning of a kernel, it retrieves 5 meta-spins from the global memory, namely its own and its four neighboring "meta-spins" (Fig. 22(a)). This information is used to extract the information for the 4×4 spins. Each thread will store the spin field of 4×4 spins as well as the neighboring spins in a 6×6 integer array in shared memory, which allows for fast computation of the spin flips. The spin update is performed in two steps as described before. A first kernel is needed to update the "black" sites on a checkerboard pattern, and a second processes the "white" sites. The update kernel for the "white" sites has to wait until all "black" sites have been updated. Thus, two separate kernels are needed. As discussed earlier, there is no other way to achieve global synchronization between the threads. Each kernel creates an update pattern, in which each binary digit indicates whether the associated spin has to be flipped or not. At the end of the kernel's execution, the 4×4 "meta-spins" are updated with one global memory write. In summary, each update thread executes the steps as follows: (a) Look up meta-spins from global memory. (b) Extract meta-spins into 6×6 integer array in shared memory which then contains the 4×4 "meta-spins" and the neighbors. (c) For all 8 "white"/"black" sites s_i in the 4×4 field, draw a random number and evaluate the Metropolis criterion. (d) Generate the update pattern (set the i^{th} bit to 1, if the flip of the i^{th} was accepted, otherwise to 0) (e) Update the "meta-spin" by an XOR operation with the update pattern to obtain the spins at the next time-step

Although this update scheme hardly sounds efficient, it dramatically reduces global memory access as compared to the previous implementation, which results in shorter computing times on GPU hardware. After the update is completed, the

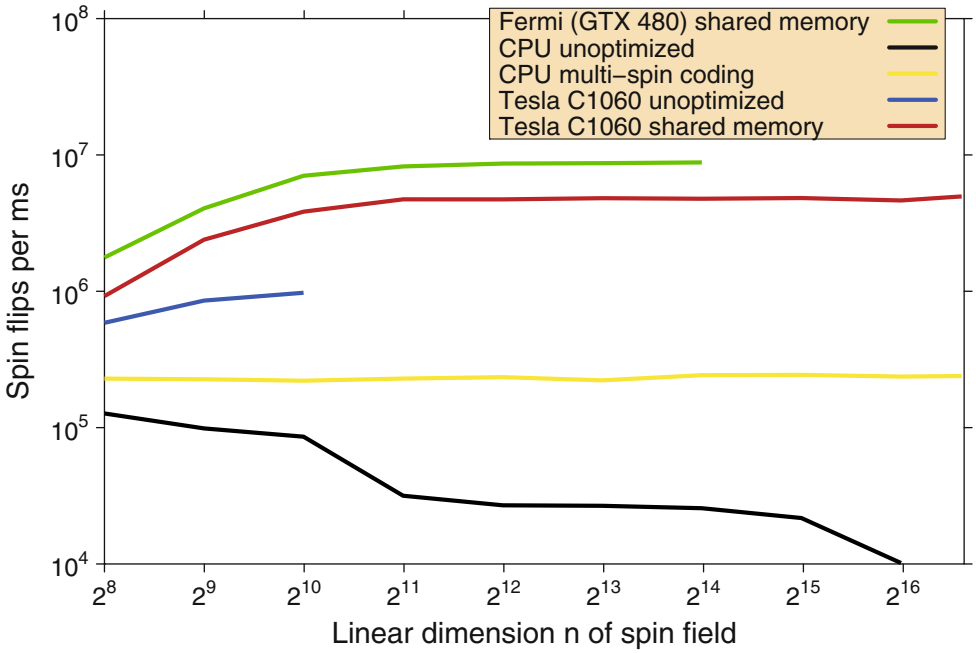

Fig. 23. (Color online) Benchmarking the implementations: The system is simulated at a constant temperature of $T = 0.99\ T_C$. The performance of the straightforward implementation varies strongly with lattice size because of the large block size of 512 threads, while the shared memory implementation offers stable performance over a wide range of sizes and offers better quality random numbers – comparable to the simple CPU implementation. For this benchmark, a GPU of NVIDIA's new Fermi generation could be used which became available in April 2010. A GeForce GTX 480 provides the following features: 1536 MB global memory, 480 streaming processor cores, 1.40 GHz processor clock, 1848 MHz memory clock, and a maximal power consumption of 250 W. The shared memory approach performs roughly two times faster than on a Tesla C1060.

magnetization per spin $m(T)$ has to be extracted from the lattice. In a first step, the magnetization of each block can be aggregated using the shared memory of each block by employing a binary tree reduction and writing the total magnetization of the slice back to main memory. The final summation of the magnetizations of the blocks can be either carried out on the CPU or on the GPU at about equal speeds.

4.5.3 Performance comparison

The "multi-spin" implementations are compared to the "one spin per integer" implementations both on a CPU and a GPU. As a measurement for the performance of an implementation, we use the number of single spin flips per second, which also allows for comparing results given different lattice sizes. The temperature is set to $0.99\ T_C$. The GPUs perform most efficient for lattice sizes of a linear dimension beyond 4096×4096. For this lattice size, a GPU is faster by a factor of about 15-35, depending on the implementation and the resulting quality of random numbers. For the implementation used in Sec. 4.2, for ranges between 1024×1024 and 2048×2048 spins, the spin field size becomes comparable to the CPU L3 cache size, leading to a higher rate of costly L3 cache misses. This is the point at which the previous implementation becomes inefficient.

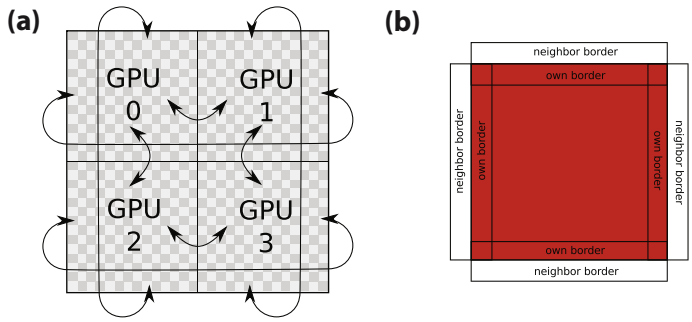

Fig. 24. (Color online) (a) Each GPU processes a "meta-spin" lattice of size $N = n^2$. The lattices are aligned on a super-lattice, and the outer borders are connected via periodic boundary conditions. In this example, 4 GPUs process a system of $2^2 \cdot N$ spins. (b) A meta-spin update needs the 4 nearest neighbor meta-spins. On the borders of a lattice, each GPU needs the spin information of the neighboring lattices. The border information has to be passed between the GPUs. In our implementation this is done by using 8 neighbor arrays.

4.6 Multi-GPU acceleration

4.6.1 Implementation details

The general idea is to extend the quadratic lattice by putting multiple quadratic "meta-spin" lattices next to each other in a super-lattice (see Fig. 24(a) for a 2×2 super-lattice) and letting each lattice be handled by one of the installed GPUs. On the border of each lattice, at least one of the neighboring sites is located in the memory of another GPU (see Fig. 24(b)). For this reason, the spins at the borders of each lattice have to be transferred from one GPU to the GPU handling the adjacent lattice. This can be realized by introducing four neighbor arrays containing the spins of the lattices' own borders, and four arrays for storing the spins of its adjacent neighbors (see Fig. 24(c)). At the beginning of execution, each MPI process initializes its own spin lattice, writes out its border spins into its own border arrays and sends them to its neighbors. In return it receives the adjacent borders from the according MPI processes. After this initialization phase, spins and random seeds are transferred to the GPU. Then, a single lattice update is performed as follows: (a) Copy neighboring borders to GPU memory. (b) Call kernel to perform update A. (c) Call kernel to extract borders from the spin array to own borders array. (d) Copy own borders to host memory. (d) Exchange borders with the other MPI processes. (d) Copy neighbor borders to GPU memory again. (e) Call kernel to perform update B. (f) Call kernel to extract borders from spin array again. (g) Transfer own borders to host memory. (h) Exchange borders with other MPI processes. (i) Retrieve processed data from GPU.

It turns out that the transfer time was not the limiting factor for our purposes but rather the latency of the memory accessed.

4.6.2 Performance on GPU clusters

For performance measurements on the GPU cluster, the shared memory implementation (see Sec. 4.5.2) was used, since it provided stable performance for various lattice sizes and because the memory layout is symmetric in the x and y directions, resulting in symmetric communication data. The tests were run on a GPU cluster with two Tesla C1060 GPUs in each node. Communication is established via Double Data Rate InfiniBand. The performance for various system sizes (see Fig. 25) provides evidence

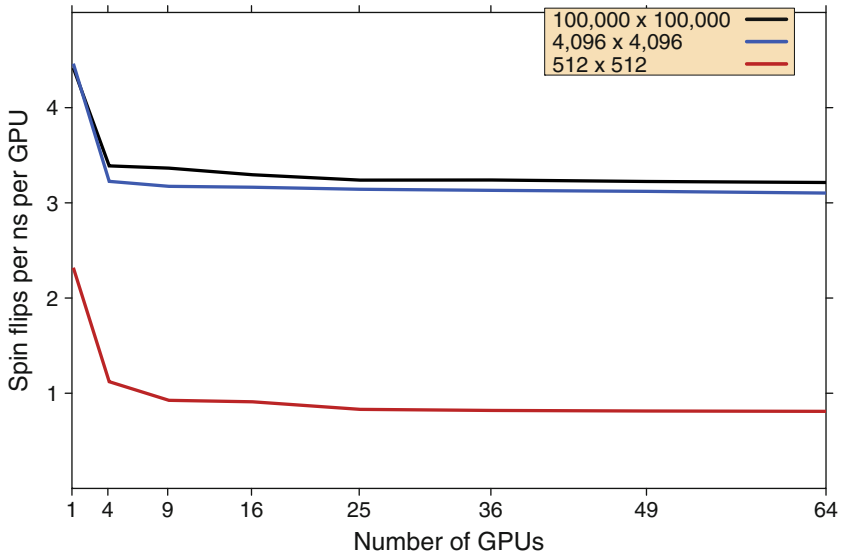

Fig. 25. (Color online) Cluster performance for various system sizes (per GPU). For more than one GPU, spin flip performance scales nearly linearly with the amount of GPUs. Again, optimal performance is reached at a lattice size of about 4096×4096 per GPU. Using 64 GPUs, a performance of 206 spin-flips per nanosecond can be achieved on a 800.000×800.000 lattice.

that for more than one GPU, spin flip performance scales nearly linearly with the number of GPUs. The drop from one GPU to four GPUs is due to the communication overhead resulting from the exchange of borders. For larger system sizes, the communication overhead per CPU/GPU remains constant. An optimal performance is reached for lattice sizes beyond 4096×4096 per GPU. For 64 GPUs – the NEC Nehalem Cluster maintained by the High Performance Computing Center Stuttgart (HLRS) provides 128 GPUs – a performance of 206 spin-flips per nanosecond can be achieved on a 800.000^2 2D Ising lattice. That is, the entire lattice can be updated in about three seconds.

5 Summary

It was shown that a graphics card architecture – a graphics processing unit (GPU) – can be very successfully used for methods of time series analysis. An accelerated determination of scaling exponents can be performed on a GPU as well as the calculation of autocorrelation coefficients. Results of the scaling behavior of a stochastic process are obtained up to 80 times faster than on a modern central processing unit (CPU) core. In addition, the relative absolute error of the results in comparison to a CPU is smaller than 10^{-3}. These methods were applied to a German Bund future (FGBL) time series of the European Exchange (EUREX), which exhibits an anti-persistence on short time scales. Evidence is found that a super-diffusive regime is reached on medium time scales. In addition, a time series of the German DAX future (FDAX) is analyzed.

As data-driven methods were able to be significantly accelerated on a GPU device, we used the concept of GPU computing for a standard model in statistical physics in order to demonstrate the wide range of applicability. Thus, we presented a GPU accelerated version of the two dimensional ferromagnetic square lattice Ising model

and the three dimensional ferromagnetic cubic lattice Ising model. This model is often adapted and interpreted in interdisciplinary contexts. For example, in the context of financial markets, spins correspond to market participants. The two possible states model the decision to buy or to sell. For our GPU based Monte Carlo simulations of the Ising model, we use a set of linear congruential random number generators on the GPU device. With the GPU implementation of a checkerboard algorithm of the two dimensional Ising model, results on the GPU can be obtained up to 60 times faster than on a recent CPU core. An implementation of a three dimensional Ising model on a GPU is able to generate results up to 35 times faster than on a modern CPU core. As proof of our conceptual method for the GPU implementation, the critical temperatures of the 2D and 3D Ising models were determined successfully by finite size scaling. Both the theoretical result for the 2D Ising model and previous simulation results for the 3D Ising model can be reproduced. The limitation of such an approach as well must also be mentioned. On a single graphics card, the simulation is limited to the amount of global memory. Algorithms simulating larger lattices have to communicate with the main memory which is the main bottleneck. Facing this problem, two major improvements could be realized as well. For the simple approach, one integer number was used for one single Ising spin. By using multi-spin coding techniques – one bit is used for one Ising spin – we improved the computation to memory access ratio of our calculations dramatically. On one GPU, up to 7.9 spin-flips per nanosecond are attainable this way. This update time is roughly 15 to 35 times faster than a multi-spin coded CPU version. The exact speed increase depends on the implementation and the quality of random numbers. A second major step was to overcome the memory limitation of the GPU global memory. It was possible to distribute a huge Ising lattice on many GPUs – each GPU updating a sub-lattice. This is very efficient for the Ising model as the spin configurations of sub-lattices can be stored on the GPUs. Only the boundaries of each sub-lattice have to be shared with neighboring GPUs hosting the neighboring lattices. This extension of the GPU based simulation of the Ising model is very attractive for GPU clusters. It is shown that the implementation scales nearly linearly with the number of GPUs.

References

1. T. Preis, H.E. Stanley, J. Stat. Phys. **138**, 431 (2010)
2. D.P. Landau, K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics* (Cambridge University Press, 2005)
3. J.A. van Meel, A. Arnold, D. Frenkel, S.F. Portegies Zwart, R.G. Belleman, Mol. Simul. **34**, 259 (2008)
4. H. Köstler, R. Schmid, U. Rüde, C. Scheit, Comput. Visual. Sci. **11**, 115 (2008)
5. J.J. Schneider, S. Kirkpatrick, *Stochastic Optimization* (Springer, 2006)
6. L. Dagum, R. Menon, IEEE Comput. Sci. Eng. **5**, 46 (1998)
7. E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, T.S. Woodall, *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 97 (2004)
8. S.J. Park, J.A. Ross, D.R. Shires, D.A. Richie, B.J. Henz, L.H. Nguyen, IEEE Trans. Parallel Distrib. Syst. **22**, 46 (2011)
9. A. Ruiz, M. Ujaldon, L. Cooper, K. Huang, J. Sig. Proc. Syst. Signal Image Video **55**, 229 (2009)
10. O.M. Lozano, K. Otsuka, J. Sig. Proc. Syst. Signal Image Video **57**, 285 (2009)
11. A.C. Thompson, C.J. Fluke, D.G. Barnes, B.R. Barsdell, New Astron. **15**, 16 (2010)
12. E.B. Ford, New Astron. **14**, 406 (2009)
13. R.B. Wayth, L.J. Greenhill, F.H. Briggs, Publ. Astron. Soc. Pac. **121**, 857 (2009)

14. R.G. Belleman, J. Bedorf, S.F.P. Zwart, New Astron. **13**, 103 (2008)
15. I.S. Haque, V.S. Pande, J. Comput. Chem. **31**, 117 (2010)
16. N. Schmid, M. Botschi, W.F. Van Gunsteren, J. Comput. Chem. **31**, 1636 (2010)
17. J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, J. Comput. Chem. **28**, 2618 (2007)
18. V.B. Putz, J. Dunkel, J.M. Yeomans, Chem. Phys. **375**, 557 (2010)
19. S. Tomov, M. McGuigan, R. Bennett, G. Smith, J. Spiletic, Comput. Graph. **29**, 71 (2005)
20. E. Gutierrez, S. Romero, M.A. Trenas, E.L. Zapata, Comput. Phys. Commun. **181**, 283 (2010)
21. F. Molnar, T. Szakaly, R. Meszaros, I. Lagzi, Comput. Phys. Commun. **181**, 105 (2010)
22. K. Hagiwara, J. Kanzaki, N. Okamura, D. Rainwater, T. Stelzer, Eur. Phys. J. C **66**, 477 (2010)
23. S. Bianchi, R. Di Leonardo, Comput. Phys. Commun. **181**, 1442 (2010)
24. B. Block, P. Virnau, T. Preis, Comput. Phys. Commun. **181**, 1549 (2010)
25. D. Komatitsch, G. Erlebacher, D. Goddeke, D. Michea, J. Comput. Phys. **229**, 7692 (2010)
26. S. Rostrup, H. De Sterck, Comput. Phys. Commun. **181**, 2164 (2010)
27. T. Preis, P. Virnau, W. Paul, J.J. Schneider, J. Comput. Phys. **228**, 4468 (2009)
28. N. Sanna, I. Baccarelli, G. Morelli, Comput. Phys. Commun. **180**, 2544 (2009)
29. J.A. Anderson, C.D. Lorenz, A. Travesset, J. Comput. Phys. **227**, 5342 (2008)
30. T. Preis, P. Virnau, W. Paul, J.J. Schneider, New J. Phys. **11**, 093024 (2009)
31. D. Michea, D. Komatitsch, Geophys. J. Int. **182**, 389 (2010)
32. X.K. Zhang, X. Zhang, Z.H. Zhou, J. Struct. Biol. **172**, 400 (2010)
33. D. Castano-Diez, D. Moser, A. Schoenegger, S. Pruggnaller, A.S. Frangakis, J. Struct. Biol. **164**, 153 (2008)
34. D. Dynerman, E. Butzlaff, J.C. Mitchell, J. Comput. Biol. **16**, 523 (2009)
35. J. Buckner, J. Wilson, M. Seligman, B. Athey, S. Watson, F. Meng, Bioinformatics **26**, 134 (2010)
36. J.M. Cecilia, J.M. Garcia, G.D. Guerrero, M.A. Martinez-del Amor, I. Perez-Hurtado, M.J. Perez-Jimenez, Brief. Bioinform. **11**, 313 (2010)
37. L. Dematte, D. Prandi, Brief. Bioinform. **11**, 323 (2010)
38. P.B. Noel, A.M. Walczak, J.H. Xu, J.J. Corso, K.R. Hoffmann, S. Schafer, Comp. Meth. Progr. Biomed. **98**, 271 (2010)
39. W.F. Shen, D.M. Wei, W.M. Xu, X. Zhu, S.Z. Yuan, Comp. Meth. Progr. Biomed. **100**, 87 (2010)
40. J.C. Phillips, J.E. Stone, Comm. ACM **52**, 34 (2009)
41. S. Hissoiny, B. Ozell, P. Despres, Med. Phys. **37**, 1029 (2010)
42. C. Rohkohl, B. Keck, H.G. Hofmann, J. Hornegger, Med. Phys. **36**, 3940 (2009)
43. M. de Greef, J. Crezee, J.C. van Eijk, R. Pool, A. Bel, Med. Phys. **36**, 4095 (2009)
44. C.H. Men, X.J. Gu, D.J. Choi, A. Majumdar, Z.Y. Zheng, K. Mueller, S.B. Jiang, Phys. Med. Biol. **54**, 6565 (2009)
45. A. Badal, A. Badano, Med. Phys. **36**, 4878 (2009)
46. S.S. Samant, J.Y. Xia, P. Muyan-Ozcelilk, J.D. Owens, Med. Phys. **35**, 3546 (2008)
47. Z.A. Taylor, O. Comas, M. Cheng, J. Passenger, D.J. Hawkes, D. Atkinson, S. Ourselin, Med. Image Anal. **13**, 234 (2009)
48. K. Xu, D.Z. Ding, Z.H. Fan, R.S. Chen, Microw. Opt. Technol. Lett. **52**, 502 (2010)
49. V. Demir, A.Z. Elsherbeni, Appl. Comput. Electrom. Soc. J. **25**, 303 (2010)
50. V. Demir, Appl. Comput. Electromagn. Soc. J. **25**, 323 (2010)
51. N. Godel, N. Nunn, T. Warburton, M. Clemens, Appl. Comput. Electromagn. Soc. J. **25**, 331 (2010)
52. F. Rossi, C. McQuay, P. So, Appl. Comput. Electromagn. Soc. J. **25**, 348 (2010)
53. A. Capozzoli, C. Curcio, G. DÉlia, A. Liseno, P. Vinetti, Appl. Comput. Electromagn. Soc. J. **25**, 355 (2010)

54. N. Godel, S. Schomann, T. Warburton, M. Clemens, IEEE Trans. Magn. **46**, 2735 (2010)
55. N. Godel, N. Nunn, T. Warburton, M. Clemens, IEEE Trans. Magn. **46**, 3469 (2010)
56. J.M. Nageswaran, N. Dutt, J.L. Krichmar, A. Nicolau, A.V. Veidenbaum, Neural Networks **22**, 791 (2009)
57. Y.C. Liu, B. Schmidt, W.G. Liu, D.L. Maskell, Patt. Recogn. Lett. **31**, 2170 (2010)
58. A. Munawar, M. Wahib, M. Munetomo, K. Akama, Genetic Program. Evolvable Mach. **10**, 391 (2009)
59. C. Muller, S. Frey, M. Strengert, C. Dachsbacher, T. Ertl, IEEE Trans. Visualiz. Comp. Grap. **15**, 605 (2009)
60. C. Wang, Y.J. Chiang, IEEE Trans. Visualiz. Comp. Graph. **15**, 1367 (2009)
61. D.M. Hughes, I.S. Lim, IEEE Trans. Visualiz. Comp. Graph. **15**, 1555 (2009)
62. A. Godiyal, J. Hoberock, M. Garland, J.C. Hart, I.G. E.D. Tollis, M. Patrignani, Graph Drawing **5417**, 90 (2009)
63. D. Goddeke, R. Strzodka, IEEE Trans. Parall. Distrib. Syst. **22**, 22 (2011)
64. J. Nickolls, W.J. Dally, IEEE Micro **30**, 56 (2010)
65. A. Benso, S. Di Carlo, G. Politano, A. Savino, A. Scionti, Control Eng. Appl. Inform. **12**, 34 (2010)
66. K. Jang, S. Han, S. Han, S. Moon, K. Park, Comp. Comm. Rev. **40**, 437 (2010)
67. A. Akoglu, G.M. Striemer, Cluster Computing. The J. Networks Software Tools Appl. **12**, 341 (2009)
68. A. Leist, D.P. Playne, K.A. Hawick, Concurr. Comp.-Pract. Exper. **21**, 2400 (2009)
69. M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov, IEEE Micro **28**, 13 (2008)
70. S.S. Stone, J.P. Haldar, S.C. Tsao, W.M.W. Hwu, B.P. Sutton, Z.P. Liang, J. Parallel Distrib. Comp. **68**, 1307 (2008)
71. S. Che, M. Boyer, J.Y. Meng, D. Tarjan, J.W. Sheaffer, K. Skadron, J. Parallel Distrib. Comp. **68**, 1370 (2008)
72. R.J. Rost, *OpenGL Shading Language* (2004)
73. R. Fernando, M.J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics* (2003)
74. NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture*, Programming Guide Version 2.0 (2008)
75. ATI Technologies Inc., *ATI CTM Guide*, Technical Reference Manual Version 1.01 (2006)
76. NVIDIA Corporation, *NVIDIA GeForce GTX 280 Specifications* (2008)
77. B.B. Mandelbrot, R.L. Hudson, *The (Mis)behavior of Markets: A Fractal View of Risk, Ruin and Reward* (Basic Books, 2004)
78. T. Preis, Eur. Phys. J. Special Topics **194**, 5 (2011)
79. H.E. Hurst, Trans. Amer. Soc. Civil Eng. **116**, 770 (1951)
80. T. Preis, W. Paul, J.J. Schneider, Europhys. Lett. **82**, 68005 (2008)
81. G.A. Darbellay, D. Wuertz, Physica A **287**, 429 (2000)
82. M. Ausloos, Physica A **285**, 48 (2000)
83. A. Carbone, G. Castelli, H.E. Stanley, Physica A **344**, 267 (2004)
84. G.-F. Gu, W.-X. Zhou, Eur. Phys. J. B **67**, 585 (2009)
85. T. Preis, S. Golke, W. Paul, J.J. Schneider, Phys. Rev. E **76**, 016108 (2007)
86. T. Preis, S. Golke, W. Paul, J.J. Schneider, Eur. Lett. **75**, 510 (2006)
87. H.E. Stanley, S.V. Buldyrev, G. Franzese, S. Havlin, F. Mallamace, P. Kumar, V. Plerou, T. Preis, Physica A **389**, 2880 (2010)
88. T. Preis, D. Reith, H.E. Stanley, Philosoph. Trans. Royal Soc. A **368**, 5707 (2010)
89. T. Preis, J. Phys.: Conf. Ser. **221**, 012019 (2010)
90. E. Ising, Z. Phys. **31**, 253 (1925)
91. K. Binder, E. Luijten, Phys. Rep. **344**, 179 (2001)
92. L. Onsager, Phys. Rev. **65**, 117 (1944)
93. K. Binder, Z. Phys. B Cond. Matt. **43**, 119 (1981)

94. B. Fierro, F. Bachmann, E.E. Vogel, Phys. B: Cond. Matt. **384**, 215 (2006)
95. H.-O. Heuer, J. Phys. A: General Phys. **26**, L333 (1993)
96. M.E. Fisher, Reports Progr. Phys. **30**, 615 (1967)
97. R.N. Mantegna, H.E. Stanley, *An Introduction to Econophysics: Correlations and Complexity in Finance* (Cambridge University Press, 2000)
98. W. Paul, J. Baschnagel, *Stochastic Processes: From Physics to Finance* (Springer, 2000)
99. W. Liu, B. Schmidt, G. Voss, W. Müller-Wittig, Comp. Phys. Comm. **179**, 634 (2008)
100. M. Matsumoto, T. Nishimura, ACM Trans. Model. Comp. Simul. **8**, 3 (1998)
101. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, J. Chem. Phys. **21**, 1087 (1953)
102. R.H. Swendsen, J.-S. Wang, Phys. Rev. Lett. **58**, 86 (1987)
103. U. Wolff, Phys. Rev. Lett. **62**, 361 (1989)
104. S. Wansleben, J.G. Zabolitzky, C. Kalle, J. Stat. Phys. **37**, 271 (1984)
105. R. Zorn, H.J. Herrmann, C. Rebbi, Comput. Phys. Commun. **23**, 337 (1981)
106. N. Ito, Y. Kanada, Supercomputer **7**, 29 (1990)
107. N. Ito, Y. Kanada, Supercomputer **5**, 31 (1988)
108. NVIDIA Corporation, NVIDIA Tesla C1060 Specifications (2009)