Journal of Computational Physics 366 (2018) 320–326

Contents lists available at ScienceDirect

Journal of Computational Physics

www.elsevier.com/locate/jcp

Short note

Counting defects in quantum computers with Graphics Processing Units

Bartłomiej Gardas [a,b,*], Andrzej Ptok [c,d]

[a] Marian Smoluchowski Institute of Physics, Jagiellonian University, ul. prof. S. Łojasiewicza 11, PL-30348 Kraków, Poland
[b] Institute of Physics, Uniwersytecka 4, 40-007 Katowice, University of Silesia, PL-40007 Katowice, Poland
[c] Institute of Nuclear Physics, Polish Academy of Sciences, ul. E. Radzikowskiego 152, PL-31342 Kraków, Poland
[d] Institute of Physics, Maria Curie-Skłodowska University, Plac M. Skłodowskiej-Curie 1, PL-20031 Lublin, Poland

ARTICLE INFO

Article history:
Received 10 September 2017
Accepted 6 April 2018
Available online 9 April 2018

Keywords:
Parallel computing
GPU
Quantum annealing
Ising model
D-Wave

ABSTRACT

We show how to simulate the dynamics of many body quantum systems using Graphics Processing Units or a similar heterogeneous architecture. In particular, we focus on the one dimensional quantum Ising model. Its entire time evolution is computed in *parallel*. The dynamics of this model provides a valuable information regarding defects created during quantum annealing. Counting such imperfections comprise a perfect test for assessing correctness of adiabatic quantum computing (B. Gardas et al. (2017) [11]). A fast method to calculate topological defects even when some types of decoherence are present is described.

1. Introduction

It is well known that the number of possible quantum states increases exponentially with increasing number of particles [1]. That makes the dynamics of many body quantum systems almost impossible to simulate using classical computers [2]. In principle, only a general purpose quantum computer will be capable of performing this task [3]. However to build one, some aspects of its complex behavior have to be simulated classically. To this end, extremely efficient and highly parallelizable classical algorithms need to be devised [4].

As a small step towards this objective, we describe how modern Graphics Processing Units (GPUs) can help accomplish this goal. In particular, we focus on the one dimensional (1D) quantum Ising model whose Hamiltonian [5],

$$\mathcal{H}(t) = -\sum_{n=1}^{L} h_n(t)\,\sigma_n^x - \sum_{n=1}^{L-1} J_n(t)\,\sigma_n^z\sigma_{n+1}^z, \tag{1}$$

describes the nearest neighbor interaction between L qubits [6]. Its strength is set by time dependent functions $J_n(t)$. The transverse magnetic field provides a bias and it is controlled by $h_n(t)$. Matrices σ_n^x and σ_n^z are the Pauli matrices along x and z directions respectively [7].

There are at least two good reasons why this model is important. The first has to do with D-Wave quantum computers [8]. Those machines promise to solve optimization problems encoded via even more general interaction using the so

* Corresponding author at: Marian Smoluchowski Institute of Physics, Jagiellonian University, ul. prof. S. Łojasiewicza 11, PL-30348 Kraków, Poland.
E-mail address: bartek.gardas@gmail.com (B. Gardas).

https://doi.org/10.1016/j.jcp.2018.04.016
0021-9991/

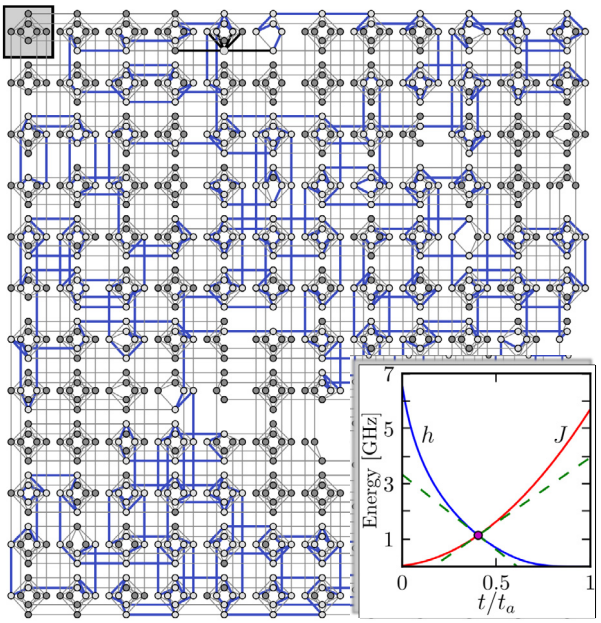

Fig. 1. 1D Ising model implemented on a D-Wave quantum computer. A random-walked implementation of the Hamiltonian (1) for $L = 600$ qubits (depicted as gray dots). Solid blue lines show active connections between sites. The inset shows a typical annealing protocol [*i.e.* functions $h(t)$ and $J(t)$], t_a is the so called annealing time – time necessary to complete one processor cycle. Green dotted lines show tangents at the critical point (depicted as a red dot). Units are such that $\hbar = 1$. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

called the Chimera graph [9]. A typical implementation of the 1D case is shown in Fig. 1. The second reason concerns adiabatic quantum computations in general [10]. As recent studies have shown, the Ising dynamics provides a valuable information regarding topological defects created during quantum computation. Counting such imperfections comprise a perfect test for assessing correctness of adiabatic quantum computing [11].

In this article we demonstrate how the 1D Ising dynamics can be simulated effectively even if some types of decoherence are present. The entire time evolution is computed in *parallel* on a GPU or similar heterogeneous architecture. Moreover, a fast method to compute topological defects is provided.

2. Quantum dynamics

The dynamic of quantum systems is determined by solving for the density operator, a positive definite [$\rho(t) > 0$], hermitian [$\rho(t)^\dagger = \rho(t)$] and trace preserving [$\mathrm{tr}\{\rho(t)\} = 1$] matrix [7]. We begin with a Lindblad master equation [12]

$$\dot{\rho}(t) = -i\left[\mathcal{H}(t), \rho(t)\right] - \frac{\gamma}{2}\sum_{n=1}^{L}\left[\sigma_n^z, \left[\sigma_n^z, \rho(t)\right]\right], \tag{2}$$

which models the Ising dynamics in the presence of a special type of local decoherence [13]. A non-negative parameter γ reflects on the coupling between the chain and its environment [14]. This particular equation attempts to model so called ghost spins. In a quantum computer some qubits may influence others (causing errors) even if the corresponding couplings between them are set to zero [11]. Note, $\gamma = 0$ implies unitary dynamics and hence the conservation of probabilities.

It is needless to say that in its current form Eq. (2) is too complicated to be solved directly. Note, here $\rho(t)$ is of size $2^L \times 2^L$ which is astronomical for long chains. To lower the complexity we first use the following Jordan–Wigner transformation [15]

$$\sigma_n^z = \left(c_n^\dagger + c_n\right)\prod_{m<n}\left(1 - 2c_m^\dagger c_m\right), \quad \sigma_n^x = 1 - 2c_n^\dagger c_n, \tag{3}$$

where c_n ($c_n^\dagger$) is a fermionic annihilation (creation) operator. In case of an open chain, the above transformation brings the Hamiltonian (1) to a quadratic form [5],

$$\mathcal{H} = 2\sum_{n=1}^{L} h_n c_n^\dagger c_n - \sum_{n=1}^{L-1} J_n\left(c_n^\dagger c_{n+1} + c_n^\dagger c_{n+1}^\dagger + \mathrm{h.c.}\right) + \mathcal{C}, \tag{4}$$

where a constant $\mathcal{C}$ does not affect equations of motion [16]. Next, we introduce the following correlation functions

$$x_{p,q} := \mathrm{tr}\left\{\rho c_p^\dagger c_q\right\}, \quad y_{p,q} := \mathrm{tr}\left\{\rho c_p^\dagger c_q^\dagger\right\}. \tag{5}$$

Taking into account that $\mathbf{x}$ is *hermitian* (*i.e.* $\mathbf{x} = \mathbf{x}^\dagger$) and $\mathbf{y}$ *antisymmetric* (*i.e.* $\mathbf{y} = -\mathbf{y}^T$) we arrive at a closed set of equations:

$$\begin{aligned} i\dot{x}_{p,q} =& - J_p\, x_{p+1,q} - J_{p-1}\, x_{p-1,q} + J_q\, x_{p,q+1} \\ &+ J_{q-1}\, x_{p,q-1} + J_q\, y_{p,q+1} - J_{q-1}\, y_{p,q-1} \\ &+ J_p\, y^*_{p+1,q} - J_{p-1}\, y^*_{p-1,q} \\ &+ 2\left(h_p - h_q\right) x_{p,q} + \gamma\, \mathcal{D}[x_{pq}], \quad q \geq p, \end{aligned} \tag{6}$$

where the Lindblad superoperator $\mathcal{D}[x_{pq}]$ reads

$$\mathcal{D}[x_{p,q}] = \begin{cases} 1 - 2x_{p,p} & \text{if} \quad p = q, \\ 2\Re(y_{p,q}) - 2|q-p|x_{p,q} & \text{if} \quad q > p, \end{cases} \tag{7}$$

and (note $y_{p,p} = 0$ as $\mathbf{y}$ is antisymmetric)

$$\begin{aligned} i\dot{y}_{p,q} =& - J_p\, y_{p+1,q} - J_{p-1}\, y_{p-1,q} - J_q\, y_{p,q+1} \\ &- J_{q-1}\, y_{p,q-1} - J_q\, x_{p,q+1} + J_{q-1}\, x_{p,q-1} \\ &+ J_p\, x^*_{p+1,q} - J_{p-1} x^*_{p-1,q} - J_p\, \delta_{p+1,q} \\ &+ 2(h_p + h_q) y_{p,q} + \gamma\, \mathcal{D}[y_{pq}], \quad q > p, \end{aligned} \tag{8}$$

where $\mathcal{D}[y_{p,q}] = 2\Re(x_{p,q}) - 2|q-p|y_{p,q}$.

These equations are to be solved with boundary conditions. For instance, when the chain is open then $c_0 = c_{L+1} = 0$ [15]. Moreover, without any loss of generality, we assume the system to be in its ground state at the beginning. That it to say, the initial condition for (6)–(8) can be written down as

$$x_{p,q}(t_0) = \langle \mathbf{0}|\, c_p^\dagger c_q\, |\mathbf{0}\rangle, \quad y_{p,q}(t_0) = \langle \mathbf{0}|\, c_p^\dagger c_q^\dagger\, |\mathbf{0}\rangle. \tag{9}$$

Above, $|\mathbf{0}\rangle$ is the eigenvector corresponding to the smallest eigenvalue of the Hamiltonian (1) when $J > 0$. Finally, the number of topological defects reads [11]

$$\text{defects} = \frac{L-1}{2} - \sum_{p=1}^{L-1} \Re\left(x_{p,p+1} + y_{p,p+1}\right). \tag{10}$$

Note, (6)–(8) is a set of *only* $L \times L$ differential equations. In the next section we show how it can be solved in *parallel* on a GPU.

3. Parallel implementation

Among numerous programming languages Fortran is most widely used for numerical simulations [17]. From many high performance compilers for parallel computing we have chosen PGI Fortran [18]. It is the only fortran compiler that natively supports CUDA technology [19]. There are also plenty of highly optimized methods that could be used to solve ordinary differential equations, see *e.g.* [20]. Our solver consists of two *kernels* that are specifically tailored for the system (6)–(8). These two kernels embody the classical Runge–Kutta method [21,22] (RK4) in the form of a highly optimized low storage scheme proposed by Williamson [23]. Given the initial value problem,

$$\dot{\mathbf{z}} = \mathbf{F}(\mathbf{z}, t), \quad \mathbf{z}(t_0) = \mathbf{z}_0, \tag{11}$$

his method consists of the following two steps, for the 4 values of s at each stage,

$$\mathbf{w}(t + \delta t) = A(s)\mathbf{w}(t) + \delta t\, \mathbf{F}(\mathbf{z}(t), t + C(s)\, \delta t), \tag{12}$$

$$\mathbf{z}(t + \delta t) = \mathbf{z}(t) + B(s)\mathbf{w}(t + \delta t), \tag{13}$$

where A, B, C are found using the RK4 Butcher's tableau [24,25]

$$A = \begin{pmatrix} 0 \\ -11/15 \\ -5/3 \\ -1 \end{pmatrix}, \quad B = \begin{pmatrix} 1/3 \\ 5/6 \\ 3/5 \\ 1/4 \end{pmatrix}, \quad C = \begin{pmatrix} 0 \\ 3/9 \\ 5/9 \\ 8/9 \end{pmatrix}. \tag{14}$$

In our case, function F is defined as

$$F(z,p,q,t) = \begin{cases} \dot{x}_{p,q} & \text{if} \quad p \geq q, \\ \dot{y}_{p,q} & \text{if} \quad p < q, \end{cases} \tag{15}$$

Algorithm 1 4th order Low-Storage Runge–Kutta.

```
set: t ← t_0 and w ← 0
calculate initial state z_0
for t ≤ T_end do
  for s ≤ 4 do
    w ← A(s)w + δt F(z, t + C(s) δt)  –  Eq. (12)
    z ← z + B(s)w  –  Eq. (13)
  end for
  t ← t + δt
end for
calculate defects – Eq. (10)
```

where the respective derivatives are given in Eqs. (6)–(8). Note, a new variable $\mathbf{z}$ encapsulates both correlations functions, *i.e.*,

$$z_{p,q} = \begin{cases} x_{p,q} & \text{if} \quad p \geq q, \\ y_{p,q} & \text{if} \quad p < q. \end{cases} \tag{16}$$

Algorithm 1 sketches necessary steps to complete full time propagation in this scheme. Fortran subroutines that encode these steps are shown below. `LSRK4_predict` predicts the solution at time $t+\delta t$ according to Eq. (12) and `LSRK4_correct` tries to correct it as dictated by Eq. (13). Within these subroutines, each CUDA thread (p,q) is defined on a two dimensional grid (see Fig. 2) and assigned a separate variable to calculate, $z_{p,q}$,

```
! Kernel 1: Low Storage RK4 predictor
attributes(global) subroutine LSRK4_predict(z,w,s,t)
! system size:
integer :: L, p, q
integer, intent(in), value :: s
real(wp), intent(in), value :: t
complex(wp), intent(in), device :: z(:,:)
complex(wp), intent(inout), device :: w(:,:)
  L = size(z,1)
  p = threadIdx%x + blockDim%x * (blockIdx%x-1)
  q = threadIdx%y + blockDim%y * (blockIdx%y-1)
  if ((p>L).or.(q>L)) return
  w(p,q) = A(s) * w(p,q) + dt * F(z,p,q,t+C(s)*dt)
end subroutine LSRK4_predict
!=====================================================
! Kernel 2: Low Storage RK4 corrector
attributes(global) subroutine LSRK4_correct(z,w,s)
integer :: L, p, q
integer, intent(in), value :: s
complex(wp), intent(in), device :: w(:,:)
complex(wp), intent(inout), device :: z(:,:)
  L = SIZE(z,1)
  p = threadIdx%x + blockDim%x * (blockIdx%x-1)
  q = threadIdx%y + blockDim%y * (blockIdx%y-1)
  if ((p>L).or.(q>L)) return
  z(p,q) = z(p,q) + B(s) * w(p,q)
end subroutine LSRK4_correct
```

Note that a kernel is always called from a host program (CPU) and executed on the device (GPU). All variables declared with the `value` attribute are dynamically sent to a GPU, those having the `device` attributes resident on a GPU. We stress that only two matrices ($\mathbf{z}$ and $\mathbf{w}$), each of size $L \times L$, have to be allocated on a GPU. Moreover, there is no need to transfer them back and forth between host and device while executing the solver. In fact, only information regarding current stage, s, and time, t, have to be sent to the device at each iteration. The PGI compiler allows for seamless allocation and data transfer between GPU and CPU,

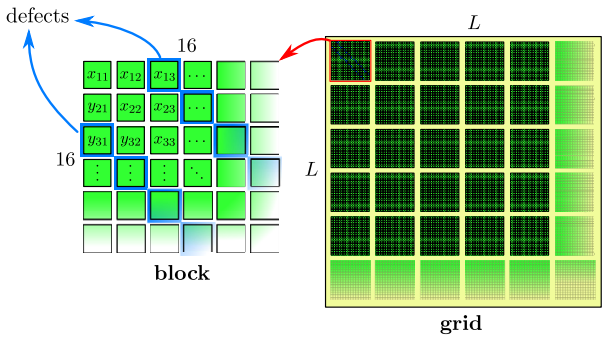

Fig. 2. CUDA grid and threading blocks setup up for Algorithm 1. Each unique CUDA thread (p, q) is defined on a $L \times L$ grid and calculates $z_{p,q}$ according to Eqs. (12)–(13). The number of defects is then computed using Eq. (10).

```
allocate(z_cpu(L,L),z(L,L)) ! both CPU and GPU
call ground_state(z_cpu) ! compute on CPU
z = z_cpu ! sent to GPU
```

To take into account different boundary conditions (*e.g.* open or periodic) we define extra functions that can only be called from the device (*i.e.* device functions). For example, instead of working directly with **z** we can "hide" it behind a wrapper,

```
! Device: wraper for open boundary conditions
attributes(device) function X(z,p,q)
complex(wp) :: X=0.
integer, intent(in) :: p, q
complex(wp), intent(in), device :: z(:,:)
  if ((p==0).or.(q==SIZE(z,1)+1)) return
  X = z(p,q)
end function X
```

that takes care of an array index out of bounds error. In a similar fashion one can accommodate any extra constrain, *e.g.* $y_{p,p} = 0$.

`LSRK4_predict` and `LSRK4_correct` are invoked as follows

```
! main loop
call cpu_time(tic) ! start timer
do while (t<=time_end)
  do s=1,4 ! global sync on return
      call LSRK4_predict<<<bGrid,tBlock>>>(z,w,s,t)
      call LSRK4_correct<<<bGrid,tBlock>>>(z,w,s)
  end do
  t = t + dt
end do
istat = cudaDeviceSynchronize()
call cpu_time(toc) ! stop timer
```

The function `cudaDeviceSynchronize()` assures that all previous CUDA calls are completed. This is necessary to correctly measure the *execution* time, `toc-tic` [18]. The two variables: `bGrid` and `tBlock` specify the number of *blocks per grid* and the number of *threads per block*, respectively. They can be setup up easily using a derived data type dim3,

```
! CUDA launch parameters
tBlock = dim3(16,16,1)
bGrid = dim3(ceiling(real(L)/tBlock%x),&
           ceiling(real(L)/tBlock%y),1)
```

For the range of physical parameters that we used in our simulations blocks of 16×16 threads yielded the best performance.

At this point one may be tempted to combine `LSRK4_predict` and `LSRK4_correct` into one compact kernel, say `LSRK4`, *e.g.*

```
if ((p<=L).and.(q<=L)) then
  y(p,q) = A(s) * y(p,q) + dt * F(z,p,q,t+C(s)*dt)
   ! Global synchronization necessary
  z(p,q) = z(p,q) + B(s) * y(p,q)
end if
```

However, for this code to work a global communication between threads would need to be established. Note, a parallel update of **z** interferes with the preceding line where **w** is being modified. Unfortunately, current CUDA technology allows to synchronize threads globally from within a kernel only on its exit [26]. Hence, the need for two separate kernels for each time step. Obviously, this is a very costly solution due to the overhead accumulation. However, the next generation of graphics cards (Volta GPU architecture) will support more flexible cooperation and synchronization between threads allowing for more efficient approach [27].

It is also worth emphasizing that with only a slight modification our code could also benefit from different parallelization libraries such as OpenMP or MPI [28]. For instance, to allow `LSRK4_predict` to take advantage of multiprocessing one would only need to distribute tasks between available cores. This can be done at no extra effort. For instance,

```
! Using OpenMP to distribute work between threads
!$omp parallel do default(shared)
do p=1,L
  do q=1,L
    y(p,q) = A(s) * y(p,q) + dt * F(z,p,q,t+C(s)*dt)
  end do
end do
!$omp end parallel do
```

The number of defects, Eq. (10), can either be computed on a CPU or GPU. To use the latter, a parallel reduction is required. This does not pose a big problem as modern compilers can perform it very efficiently [29]. In fact, this is straightforward with CUDA Fortran as the entire reduction algorithm can be carried out using the function `sum()`. Reduction on the host, on the other hand, requires data transfer. Fortunately, only two vectors of size L each. Moreover, a CPU can also perform a parallel reduction operation with both OpenMP or MPI.

4. Results

To quantify the advantage gained from using CUDA we define the speedup, S, as a ratio of the CPU to GPU time spent to execute a given portion of code (*e.g.* the main loop). That is,

$$S = \frac{T_{\text{cpu}}}{T_{\text{gpu}}}. \tag{17}$$

Without any loss of generality we assume

$$J_p(t) = 1, \quad h_p(t) = 1 - t/\tau_Q\,, \quad -\tau_Q \le t \le \tau_Q \tag{18}$$

that corresponds to a typical quantum annealing [30]. As can be seen in Fig. 3 the GPU used is over two orders of magnitude faster that a single CPU, for both numerical precisions. Nevertheless, adding more CPU's closes that gap. However, given that the number of available cores on a single CPU is very limited, a GPU is a better choice from a desktop user perspective.

The speedup S depicted in Fig. 3 does not take into account neither the initial (final) data transfer nor the total time needed to compute the initial condition $\mathbf{z}_0$. For the system sizes that we have considered data transfer is negligible. In our calculations the system evolved from its ground state which was *partially* calculated on a GPU. Our analysis shows that 89% of the time a GPU spends evaluating `LSRK4_predict`, another 10% executing `LSRK4_correct`. The remaining 1% is used to compute the ground state and initial necessary variables.

5. Summary

We have devised a simple yet powerful technique to simulate the one dimensional quantum Ising model using Graphics Processing Units. Our method is based on a highly optimized low storage classical Runge–Kutta method. It runs in *parallel* on a GPU and can easily be adapted for a similar heterogeneous architecture. A computer code that we implemented with the PGI CUDA Fortran compiler shows that a typical GPU is over two orders of magnitude faster that a single CPU core.

As has been show recently, counting topological defects in the Ising chain can be used to benchmark quantum computers [11]. In this paper we have shown how the imperfection resulting from quantum annealing can be calculated using parallel reduction in CUDA.

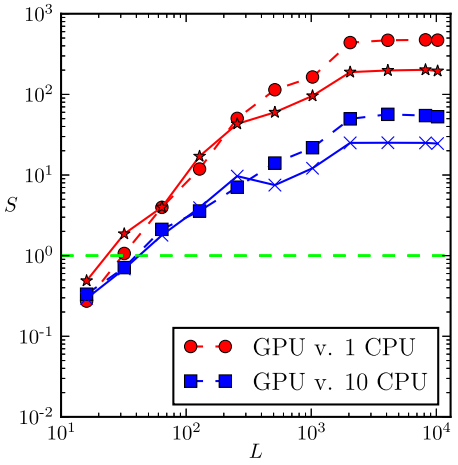

Fig. 3. CPU v. GPU. Comparison between CPU (i7-6950X with 10 cores) and GPU (GeForce GTX 1080 with 2560 CUDA cores) executing Algorithm 1 for different systems sizes L. Speedup S defined in Eq. (17) was determined using the intrinsic fortran function `cpu_time`. Red (blue) line shows the gain with respect to a singe CPU (10 CPUs running in parallel). Solid (dashed) lines refer to single (double) precision. Physical parameters used: $\tau_Q = 1.0$, $\delta t = 0.01$ and $\gamma = 0.01$ [see Eq. (18)].

Acknowledgements

Work of BG and AP was supported by National Science Centre under Project No. UMO-2016/20/S/ST2/00152 and UMO-2016/20/S/ST3/00274, respectively. This research was supported in part by PL-Grid Infrastructure. We appreciate fruitful discussions with Marek M. Rams and Jacek Dziarmaga.

References

[1] A. Montina, Exponential complexity and ontological theories of quantum mechanics, Phys. Rev. A 77 (2008) 022104.
[2] A. Chenu, M. Beau, J. Cao, A. del Campo, Quantum simulation of generic many-body open system dynamics using classical noise, Phys. Rev. Lett. 118 (2017) 140403.
[3] T.D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, J.L. O'Brien, Quantum computers, Nature 464 (2010) 45.
[4] M. Januszewski, M. Kostur, Accelerating numerical solution of stochastic differential equations with CUDA, Comput. Phys. Commun. 181 (2010) 183.
[5] J. Dziarmaga, Dynamics of a quantum phase transition and relaxation to a steady state, Adv. Phys. 59 (2010) 1063.
[6] J. Dziarmaga, Dynamics of a quantum phase transition: exact solution of the quantum Ising model, Phys. Rev. Lett. 95 (2005) 245701.
[7] J. Sakurai, J. Napolitano, Modern Quantum Mechanics, Pearson new international edition, Pearson, 2013.
[8] T. Lanting, A.J. Przybysz, A.Y. Smirnov, F.M. Spedalieri, M.H. Amin, A.J. Berkley, R. Harris, F. Altomare, S. Boixo, P. Bunyk, N. Dickson, C. Enderud, J.P. Hilton, E. Hoskinson, M.W. Johnson, E. Ladizinsky, N. Ladizinsky, R. Neufeld, T. Oh, I. Perminov, C. Rich, M.C. Thom, E. Tolkacheva, S. Uchaikin, A.B. Wilson, G. Rose, Entanglement in a quantum annealing processor, Phys. Rev. X 4 (2014) 021041.
[9] R. Santana, Z. Zhu, H.G. Katzgraber, Evolutionary approaches to optimization problems in Chimera topologies, arXiv:1608.05105, 2016.
[10] T. Kadowaki, H. Nishimori, Quantum annealing in the transverse Ising model, Phys. Rev. E 58 (1998) 5355.
[11] B. Gardas, J. Dziarmaga, W.H. Zurek, Defects in quantum computers, arXiv:1707.09463, 2017.
[12] J. Dziarmaga, W.H. Zurek, M. Zwolak, Non-local quantum superpositions of topological defects, Nat. Phys. 8 (2012) 49.
[13] J.E. Avron, M. Fraas, G.M. Graf, P. Grech, Landau–Zener tunneling for dephasing Lindblad evolutions, Commun. Math. Phys. 305 (2011) 633.
[14] W.H. Zurek, Decoherence, einselection, and the quantum origins of the classical, Rev. Mod. Phys. 75 (2003) 715.
[15] E. Lieb, T. Schultz, D. Mattis, Two soluble models of an antiferromagnetic chain, Ann. Phys. 16 (1961) 407.
[16] M.L. Wall, L.D. Carr, Out of equilibrium dynamics with matrix product states, New J. Phys. 14 (2012) 125015.
[17] S. Arabas, D. Jarecka, A. Jaruga, M. Fijałkowski, Formula translation in Blitz++, NumPy and modern Fortran: a case study of the language choice tradeoffs, Sci. Program. 3 (2014) 201.
[18] G. Ruetsch, M. Fatica, CUDA Fortran for Scientists and Engineers, vol. 2701, 2011.
[19] Volta GPU Architecture, http://www.pgroup.com/resources/cudafortran.htm. (Accessed 23May2017).
[20] K. Ahnert, D. Demidov, M. Mulansky, Solving ordinary differential equations on GPUs, in: V. Kindratenko (Ed.), Numerical Computations with GPUs, Springer International Publishing, Cham, 2014, pp. 125–157.
[21] C. Runge, Über die numerische Auflösung von Differentialgleichungen, Math. Ann. 46 (1895) 167.
[22] W. Kutta, Beitrag zur naherungsweisen Integration von Differentialgleichungen, Z. Math. Phys. 46 (1901) 435.
[23] J. Williamson, Low-storage Runge–Kutta schemes, J. Comput. Phys. 35 (1980) 48.
[24] J.C. Butcher, Coefficients for the study of Runge–Kutta integration processes, J. Aust. Math. Soc. 3 (1963) 185–201.
[25] M.H. Carpenter, C.A. Kennedy, Fourth-order 2N-storage Runge-Kutta schemes, NASA-TM-109112 (1994) 1–18.
[26] CUDA Toolkit Documentation, http://docs.nvidia.com/cuda/. (Accessed 25May2017).
[27] PGI CUDA Fortran Compiler, https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/. (Accessed 24August2017).
[28] M. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw–Hill, 2004.
[29] Faster Parallel Reductions on Kepler, https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/. (Accessed 25May2017).
[30] W.H. Zurek, U. Dorner, P. Zoller, Dynamics of a quantum phase transition, Phys. Rev. Lett. 95 (2005) 105701.