Non-steady relaxation and critical exponents at the depinning transition. Supplemental Material: GPU-based numerical implementation

E. E. Ferrero, S. Bustingorry, and A. B. Kolton
CONICET, Centro Atómico Bariloche, 8400 San Carlos de Bariloche, Río Negro, Argentina
(Dated: February 21, 2013)

We detail to some extent the numerical implementation and codes used to obtain the results presented in the manuscrip [URL will be inserted by publisher].

I. BRIEF OVERVIEW OF GPGPU COMPUTING

The acronym GPGPU stands for "General Purpouse Graphics Processing Unit". GPGPU Computing is a common denomination for the practice of using GPUs (Graphics Processing Units) as a hardware facility to run general-purpouse progarms, apart from rendering graphics (or not). The fact that the GPU is used as an accelerator collaborating with the CPU is only one example of heterogeneous computing. Heterogeneous architectures came to mitigate the technical barriers emerged in the development of faster and faster processors [1]. From a decade on, the gain of GFlops in modern computers is given more by their ability of processing applications in parallel and providing different hardware profits (as cached memory flow) than by their increase in processor clock frequency. As a consequence, software in general was pushed to be programed for parallel performance. The improvement of compilers relieves in some cases the lack of parallel implementations, but in many others, applications have to be re-formulated to fit new architectures. In this sense scientific-computing software is not an exception and we should rethink our usual codes and numerical simulation techniques.

Independently of their manufacturer, generation or model, all GPUs share the same Single Instruction Multiple Thread (SIMT) concurrency paradigm in order to exploit their high parallelism and high memory bandwidth[2]. Basically, the programming framework using SIMT paradigm consists in coding for an unlimited number of parallel threads for all practical purposes [3] (typically one thread per component of our system). A remarkable point is that, from the programmer perspective the actual number of cores in our GPU is not important at first glance. Memory issues, on the other hand, arise as the main problem to work out. We code as we were given an unlimited number of parallel processing units, the compiler and a smart thread scheduler do the rest for us [4]. By knowing better our GPU's characteristics (memory levels and sizes, number of cores, compute capabilities, etc.) we can improve the code, orientate the compiler and take real advantage of the hardware potential.

In short, the parallel simulation framework work as follows. The main stream of the program runs in a single core of the CPU (the "Host"). The GPU (or "Device") is connected to the host through a high speed IO bus slot PCI-Express. The GPU has its own device memory, up to several gigabytes. The Host runs the program's serial parts, manage data transfers between Host and Device, modifies device memory, and launch kernels that are executed concurrently by hundreds of cores on the Device. These kernels are functions that run on the Device acting exclusively on device memory.

We work with NVIDIA GPUs, as it is our available hardware. Simulations for the present work have been run on Fermi architecture (GF100) GPUs, in particular GeForce GTX 470, GeForce GTX 480 and Tesla C2075.

In the manuscript we have shown how useful can be the massive parallelism of GPUs in the framework of statistical mechanics simulations, in particular in the study of the Quenched Edwards-Wilkinson elastic line. In this Supplemental Material we present with some detail our GPU implementations of the model wich, even without being developped as a highly optimized codes, have shown to be modest powerful tools.

II. GPGPU IMPLEMENTATION OF THE QEW MODEL

In the last few years the use of GPUs to accelerate simulations has burst out in many areas of Physics and science in general. In Statistical Physics several works have been devoted to report implementations of spin models on GPU architectures (see [5–8] and references therein) and molecular dynamics [9, 10], for recent examples. In particular, in the subject of interfases and surface growth models concurrent implementations of the KPZ model dynamics were also reported [11, 12]. All these papers evidence the benefits of working with GPU's implementations in statistical physics.

For the present work, we have implemented a GPU-based parallel implementation of the QEW model. Our codes are written in C++ and C for CUDA. At it was presented in section III-B of the manuscript, we numerically solve the equation

$$\eta \partial_t u(x,t) = c\partial_x^2 u(x,t) + F_p(u,x) + f, \qquad (1)$$

discretizing the system in L segments of size $\delta x = 1$ in the x-direction (i.e., $x \to j = 0, ..., L-1$, $u(x,t) \to u_j(t)$) while keeping $u_j(t)$ as a continuous variable.

Essentially, at each step we compute *in parallel* the local forces acting on each segment u_j, adding the elastic

force given by the Laplacian, the pinning force coming from the disorder potential, the external driving force and eventually a random Langevin noise playing the role of a thermal bath. Given the instantaneous local forces, an Euler evolution step is performed also *in parallel*. Moreover, observables calculations and averages are computed also by *parallelized* routines taking full advantage of the GPU concurrence.

To model the continuous quenched random potential, we can either read or dynamically generate uncorrelated random numbers with a finite variance at the integer values of u and x and use interpolation to get $F_p(u, x)$. We have developed two different codes realizing these options. In our first approach, the disorder potential is determined from a precalculated $L \times M$ array of random numbers taken from a given distribution. This is none but the parallelized version of the code used in previous works [13–15]. Our second approach consists, instead, in generating the disorder dynamically as the string moves. This new approach allows the system size in the u direction to be virtually infinite, but a period for M can be imposed if desired. In both approaches we have used periodic boundary conditions in the longitudinal direction, so u_0 is elastically coupled with u_{L-1}.

A. Parallelized QEW line on a preset disorder potential

In our first implementation the disorder potential is constructed from a $L \times M$ matrix of real values $V(i, j)$ taken randomly from a Gaussian or Uniform distribution.

Given $V(i, j)$, we associates each value with a discrete position $i = \lfloor u \rfloor$, $j = x$ (where $\lfloor y \rfloor$ stands for the discrete part of a given continuous variable y). We can either construct a continuous spline for each x by (linearly or cubicly) interpolating the values $V(i, j)$ for fixed j, in which case we have a random bond potential and the pinning force $F_p(u, x)$ can be derived for each value of u. Alternatively, we can consider $V(\lfloor u \rfloor, j)$ as a step function force itself, in which case we have a random field.

Both the string $u(x)$ and the pinning force $F_p(u, x)$ can be optionally chosen to be `float` or `double` type variables. We have not noticed precision effects in our test runs. Nevertheless, for all results presented in this work we ensure $u(x)$ to be a double precision array, since it is an accumulative array during the string evolution.

The simulation of each sample consists of the following steps:

1) Choose an initial condition $u_j(t = 0)$ for the string.
2) Set the disorder matrix $V(i, j)$.
3) If a cubic spline is desired, calculate the potential's second derivative in each integer for interpolation.
4) Run the desired amount of updates (time steps) consisting of:
   - Calculate the total force at each point u_j of the string (including interpolation of the disorder spline when is needed).
   - Evolve $u(x, t)$ with an Euler integration.
   - At selected times, calculate and accumulate for future averages the string's center of mass displacement $u_{cm}(t)$, velocity $v(t)$ and quadratic width $w^2(t)$.
   - Eventually, calculate and accumulate the structure factor $S_q(t)$.
5) Average and print results.

Arrays of Steps 1 and 2 can be either computed on the Host and then copied to the Device or directly computed on the Device. For the case where we need cubic splines of the disorder potential, we also need to compute an array holding the second derivative of the potential $V(\lfloor u \rfloor, z)$, this is make in Step 3. Since this is done only once for each sample, we simply implement in the Host the usual Numerical Recipes [16] tri-diagonal solver to calculate it. Step 4 proceeds completely on the Device, with full parallelism, launching consecutive CUDA Kernels. For example, for the kernel which computes the total force at each point, *each* thread j ($j = 0, \ldots, L-1$) independently reads the pinning force $F_p(j)$ from a device memory array, calculates the elastic force using the instantaneous value of $u(j)$ and its neighbors sites $u(j-1)$, $u(j+1)$ (preserving periodic boundary conditions), adds the uniform driving force, calls a random number routine and adds a Langevin noise (if temperature is not zero) and finally writes the resulting total force acting at point (u, j) on a global array. With an array of forces at hand, the Euler evolution step is trivially parallel. It is worth stressing that we are not deviating from the usual (serial) dynamics at any point, since we do not modify the basic algorithm, we just replace dumb loops with CUDA kernels. Calculations of averages over x to obtain $u(t)$, $v(t)$, $w^2(t)$ are also executed in parallel taking advantage of the `Thrust` library [17] provided in the CUDA Toolkit. $S_q(t)$ is calculated using the `CUFFT` library [18], a parallel Fast Fourier Transform library for CUDA also provided in the CUDA Toolkit [19]. Copies from Device to Host only take place in Step 5, where we average over samples and print the results on a file.

The implementation of a random number generator (RNG) in GPUs is a topic of study itself (for a recent review see [20]). For this first implementation we have used the Multiply-With-Carry (MWC) RNG, which has proven to success in classical spin systems [7, 20]. A great advantage of MWC is that it allows the simultaneous generation of several independent random sequences (basically, each thread has it's own generator) and it has a good trade-out of performance and statistical quality.

B. Parallelized QEW line on dynamically generated disorder

Our second code is based on a different approach. It basically consists of generating the disorder dynamically. That is, as the line moves in the u direction we generate the underlying disorder $V(i,j)$ at each discrete position $(\lfloor u \rfloor, j)$ in a consistent manner.

For the RF case we define $F_p(u,x) \equiv V(\lfloor u \rfloor, x)$, while for the linear spline RB case we interpolate $V(u,x) = [V(\lfloor u \rfloor + 1, x) - V(\lfloor u \rfloor), x]/a$, with $a \equiv 1$, and derive the force at each value of u.

The consistence of this procedure relies on the ability to read always the same random number $V(\lfloor \widetilde{u} \rfloor, \widetilde{x})$ in a particular place $(\lfloor \widetilde{u} \rfloor, \widetilde{x})$. This can be done, of course, storing each random number as we generate them, but this option would be time expensive in terms of memory reads, which is not a good option in general, even less for a GPU implementation. A much more elegant solution come from the implementation of counter-based RNG. Let us extend on this point because it is crucial. Usual RNGs base their quality and large period on operating with linear algebra and/or logical transformations. Counter-based generators, instead, have no state. These RNGs use simple indexes or counters as an input and a well complex hash function to produce a pseudo-random number sequence. While such generators have not received much attention for their use in simulations they are common to the functions used in secret-key cryptography. These kind of generators fit perfectly in the GPGPU programing framework, since they do not need to store a state, they allow several independent sequences by setting different keys, and, the most important thing, they have been proven to show excellent quality and performance [20].

We have implemented the recently introduced PHILOX RNG [21]. In the sequence of pseudo-random numbers with key k and counter n

$$x_n = f_k(n), \tag{2}$$

we have chosen k to be a different number for each thread $j = 0, \ldots, L-1$ ($k = G(j)$, where G is some bijective function) and n to be the integer part of the instantaneous string position for that thread $n = \lfloor u(j) \rfloor$. Though, given a position $(\lfloor \widetilde{u} \rfloor, \widetilde{x})$ we have a univocal way of determining $V(\lfloor \widetilde{u} \rfloor, \widetilde{x})$ *on-the-fly*. Since the pseudo-cryptographic biyections used by PHILOX are designed to deliver essentially indistinguishable outputs for any value of k, different keys assure us independent random-number streams (up to 2^{64} independent sequences for a 64-bit key). Notice that if the string moves forward and backwards (which is actually the case when we have finite temperature), the underlying disorder already visited can be easily recovered without any memory record. While going forward and backward on a sequence is trivial for a counter-based RNG, it is an almost impossible task for usual RNGs.

The implementation of PHILOX on GPUs encouraged us to propose a different approach to the disorder representation, saving huge amount of memory storage (within this approach, our system effective size is L instead of $L \times M$) and reading, leading to an accelerated GPGPU performance. Notice as well that, if we want to preserve a finite period M for the disorder in the u direction, that can be trivially done adding a modulo operation in the definition of the counter n.

Besides this main point on the dynamical generation of the disorder, the algorithm structure is based on the first approach above. The drawback of this approach is probably that in principle it only let to simulate *linear* spline disorders. But workarounds can be find in future implementations.

The peculiarity of our second code, besides the RNG used, is that it is completely implemented using `Thrust` functions instead of setting-up and launching CUDA kernels. `Thrust` is a parallel algorithms library [17] which resembles the C++ Standard Template Library. Its high-level interface allows development without much knowledge of GPU's architecture. In addition, it enables portability between GPUs and multicore CPUs. The very same code can be compiled with an "OpenMP" flag for the backend system and it will use our multicore CPU as the Device instead of using the GPU.

C. Comparative performance

In this work, our intention was to perform a first step in the implementation on heterogeneous hardware of driven elastic systems in disordered media. We are not pretending to present a highly-optimized code, but a code that allows us to run simulations that where practically impossible to perform before in reasonable computational times. Our main objective was to obtain original physical results. The full optimization of the codes is left for further work or for other enthusiastic developers.

That said, we think that we have developed good pieces of useful code. We present their performance here and open them to the community for use, improvement and modification [22].

Testing, validation and verification are important facts on any kind of software, in particular on scientific software. Although we have not programed autonomous testings we have take carefully in mind this aspects at each step of the programming. We have tested our codes in the well known particular chooses of the parameters, e.g., without disorder, zero force, etc. Among passive security measures, we use assertions (boolean predicates) related to hardware limitations or to enforce preconditions on algorithm running. We also check every return condition of CUDA library calls and kernels.

For our benchmarking we ran the "old" single thread, CPU-based, code and the two GPU-based codes presented in the previous subsections. Each code is not totally comparable with the others in programing terms.

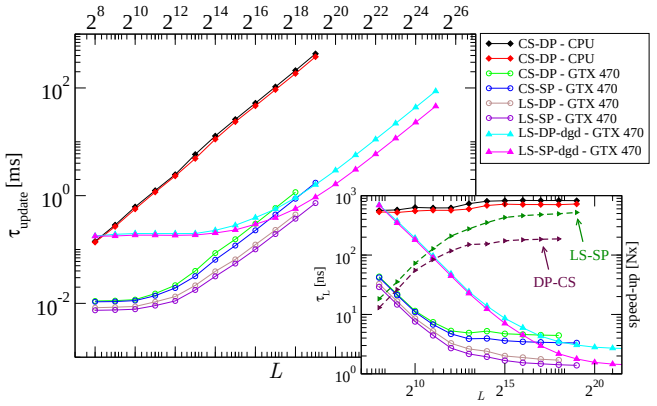

FIG. 1. (Color online) The main plot shows the update step mean computing time τ_{update} in milliseconds as a function of the system size L, from the execution of our serial and parallel codes in a CPU + NVIDIA GTX 470 GPU platform. In the inset we show $\tau_L = \tau_{update}/L$ in nanoseconds and the practical speed-up of two cases with dashed lines, showing accelerations that range between 187 and 521 for large systems. Legends are LS:= Linear Spline, CS:= Cubic Spline, SP:= Single Precision, DP:= Double Precision, dgd: dynamically generated disorder.

Therefore, we have no intentions to report speed-ups in terms of hardware, but rather, we intend to highlight the *practical speed-up* that comes out when one compare the computing times of different software-hardware (as a whole) options.

In Fig.1 we show the mean computing time in milliseconds for a single update step as a function of the system size. The values shown here where obtained running 2^{15} steps of the update procedure and dividing by that number of steps. The update procedure includes calculating for each $j = 0, \ldots, (L-1)$ all the instantaneous involved forces, and preforming an Euler integration of $u(x)$. In the rest of this Supplemental Material we briefly discuss our interpretation of this benchmark.

As can be seen from Fig.1, for large enough system sizes the computing time of all codes scale approximately as $\sim L$, as is expected since the number of arithmetic and memory operations grows linearly with the number of elementary components in the string of our model. We compare computing times in this linear scaling regime first, and the we comment about the "small" system sizes.

The CPU code is written in C/C++ using the GSL [23] library for accelerated interpolation and the Blitz++ [24] library for vector and matrix operations [25]. It is not a highly optimized code but all the commonly known good practices for a serial code have been considered.

Our CUDA codes are two, code "A", the one presented in Sec.II A where we read the disorder from an array, and code "B" presented in Sec.II B where we dynamically generate the disorder. For code A we use pure CUDA Kernels for the update steps, while for code B the update procedure uses `Thrust` functions [26]. To compare different cases, we have varied the types and operations precision (Float or Double), used different spline interpolation schemes (Cubic/Linear) when allowed, and tested the codes in different hardware platforms.

Case	single-core CPU	CPU + GTX 470		CPU + Tesla C2075	
	time[ms]	*time[ms]*	*psu*	*time[ms]*	*psu*
CS DP	54.49	0.301	181	0.396	138
CS SP	~ 54	0.227	238	0.298	181
LS DP	46.46	0.123	377	0.152	305
LS SP	~ 46	0.101	455	0.128	359

TABLE I. Mean update time execution and their related CPU vs. CPU+GPU practical speed-up for a system size $L = 65536$ taken as an example. Platform: AMD Phenom(tm) II X4 955 Processor @3.2GHz, NVIDIA Tesla C2075, NVIDIA GTX 470. CS: Cubic Spline, LS: Linear Spline, DP: Double Precision, SP: Single Precision, psu: practical speed-up. All speed-ups are approximate numbers.

Case	single-core CPU	CPU + GTX 480	
	time[ms]	*time[ms]*	*psu*
CS DP	42.22	0.238	186
CS SP	~ 42	0.179	235
LS DP	36.55	0.098	373
LS SP	~ 36	0.076	477

TABLE II. Mean update time execution and their related CPU vs. CPU+GPU practical speed-up for a system size $L = 65536$ taken as an example. Platform: Intel(R) Core(TM)2 Quad CPU Q9550 @2.83GHz, NVIDIA GTX 480. CS: Cubic Spline, LS: Linear Spline, DP: Double Precision, SP: Single Precision, psu: practical speed-up. All speed-ups are approximate numbers.

The first thing that we can stress from the benchmarking is that the Double Precision (DP) Cubic Spline (CS) CUDA implementation of the QEW model (code A, DP-CS option), the one which we have first implemented to reproduce the well known results in CPU implementations, gives us a great improvement (above 138x). The conservative user can choose to work with it to obtain completely safe results and still have a great saving in simulation time.

Let us comment further options, that have been summarized in Tables I, II for the two different heterogeneous platforms we have worked with. Here we have chosen $L = 65536$ as an example in the regime where execution time grows linearly with size. For the DP-CS case the best practical speed-up (186x) is obtained with the GTX 480 GPU. On each case, we compare between C++ serial and CUDA parallel codes running on the same machine, one of them using a single CPU core and the other using a CUDA parallelized scheme and a GPU in addition to the CPU core. If we use Single Precision (SP) variables and operations, which as far as we have seen does not

affect our results, computing times drops around $\sim 30\%$, representing a boost going up from 181x to 238x on the GTX 470, for example.

Increasing M does not represent a noticeable impact in the code performance but it does represent a considerably increase in memory demand, since we need to allocate arrays of size $L \times M$. The Tesla C2075 GPU shows the smaller speed-ups in all cases, even in double precision it does not improve the GTX 470 performance, nevertheless its big global memory (5.4GB) allows us to simulate large $L \times M$ systems.

Another appreciable improvement in computing time is seen when we use a linear spline (LS) either than a cubic spline for the determination of the pinning potential. GPU computing times are in this case more than 2.4 times faster than the ones obtained for a CS. In other words, the linear spline approach take $\sim 40\%$ of the time that the cubic spline takes. This can be understand from the fact that CS requires each thread to perform two additional read operations from device global memory as compared to the LS, the memory reads of an array holding the second derivative of the disordered potential at each point. While in CPU cores (with a good cache memory) this is not expected to be a dramatically sensitive issue (we see a slowdown of $\sim 17\%$ in execution time), on GPUs this produce a more noticeable impact since performance is clearly governed by memory transfers. The averaged single update step time drops to 0.123ms for the GTX 470 case, while our CPU code time only drops to 46.46ms, representing a practical speed-up of 377x.

This very good performance of the LS case suits also to our GPU code B, the one where we generate the disorder dynamically. In this case, besides the good performance of the LS spline, we save a lot of memory storage, since we are free of disorder arrays. This allows us to increase considerably L, and was exploited in Section IV-D of the manuscript, and can be appreciated in Fig.1. We claim that with these algorithm realistic simulations of sizes $L = 10^9$ can be reached using only a single GPU (with more than 4GB of memory) in a desktop computer.

A word should be said about what happens for "small" system sizes. Bellow $L = 8192$ the computing times for our GPU codes tend to saturate to a common value for different L. This is because, in these cases, we are not taking real advantage of the GPU power. In GPGPU computing, memory latency issues are hidden under a formidable scheduling of multiple threads [3]. If threads are not enough, a lower bound imposed by the latency of global memory access dominates the computing time.

The benchmark of our code B, shows a particularity. We see that for small sizes its performance is bad as compared with the code A, and also worst than the CPU code for systems $L \lesssim 256$. This is something that it is not totally clear for us, but we guess that it is related with an overhead produced by the `Thrust` functions. Since we do not control how those kernels are launched we cannot avoid this overhead. Nevertheless, it is worth notice that for large systems this code has roughly the same performance of code A and the advantage of having much less memory limitations. In some sense, we were expecting a better performance of this code since here we do not need to read the disorder potential from global memory at each step; but it seems that the PHILOX RNG call for each thread is as time consuming as a global memory access.

Let us remark that, while the CPU code computing time scales with L^α with $\alpha \simeq 1$, as far as we have increase the system size all GPU codes versions are still scaling with $\alpha < 1$, thus giving greater speedups as we increase the system size. Somehow, as we saturates more and more the GPU, massively parallelism gives back us better and better performance.

[1] For a pleasant reading we refer the reader to http://herbsutter.com/welcome-to-the-jungle.

[2] Up to 2496 computing cores and 208 GB/sec of memory bandwidth in NVIDIA Tesla K20 available nowadays.

[3] D. B. Kirk and W. M. W. Hwu, *Programming massively parallel processors: a hands-on approach* (Morgan Kaufmann Publishers, 2010) p. 258.

[4] NVIDIA Corporation, *CUDA C Programming Guide Version 4.2* (2012).

[5] M. Weigel and T. Yavorskii, Physics Procedia **15**, 92 (2011), proceedings of the 24th Workshop on Computer Simulation Studies in Condensed Matter Physics (CSP2011).

[6] M. Weigel, J. Comput. Phys. **231**, 3064 (2012).

[7] E. E. Ferrero, J. P. De Francesco, N. Wolovick, and S. A. Cannas, Comput. Phys. Comm. **183**, 1578 (2012).

[8] E. E. Ferrero, F. Romá, S. Bustingorry, and P. M. Gleiser, Phys. Rev. E **86**, 031121 (Sep 2012), `http://link.aps.org/doi/10.1103/PhysRevE.86.031121`.

[9] J. van Meel, A. Arnold, D. Frenkel, S. Portegies Zwart, , and R. Belleman, Molecular Simulation **34**, 259 (2008).

[10] J. A. Anderson, C. D. Lorenz, and A. Travesset, J. Chem. Phys. **227**, 5342 (2008), ISSN 0021-9991, `http://www.sciencedirect.com/science/article/pii/S0021999108000818`.

[11] H. Schulz, G. Ódor, G. Ódor, and M. F. Nagy, Computer Physics Communications **182**, 1467 (2011), ISSN 0010-4655, `http://www.sciencedirect.com/science/article/pii/S0010465511001081`.

[12] J. Kelling and G. Ódor, Phys. Rev. E **84**, 061150 (Dec 2011), `http://link.aps.org/doi/10.1103/PhysRevE.84.061150`.

[13] A. B. Kolton, A. Rosso, E. V. Albano, and T. Giamarchi, Phys. Rev. B **74**, 140201 (2006).

[14] A. B. Kolton, A. Rosso, and T. Giamarchi, Phys. Rev. Lett. **94**, 047002 (2005).

[15] A. B. Kolton, A. Rosso, and T. Giamarchi, Phys. Rev. Lett. **95**, 180604 (2005).

[16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C (2nd ed.): the art of*

scientific computing (Cambridge University Press, New York, NY, USA, 1992) ISBN 0-521-43108-5.

[17] J. Hoberock and N. Bell, "Thrust: A parallel template library," (2010), version 1.3.0, http://www.meganewtons.com/.

[18] NVIDIA Corporation, "CUDA library for the Fast Fourier Transform, PG–05327–040_v01," (2012).

[19] We have used CUDA Toolkit 4.2 version [27].

[20] M. Manssen, M. Weigel, and A. K. Hartmann, "Random number generators for massively parallel simulations on gpu," arXiv:1204.6193 (2012).

[21] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw (ACM, 2011) pp. 16:1–16:12, ISBN 978-1-4503-0771-0, proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, http://doi.acm.org/10.1145/2063384.2063405.

[22] Source codes are available at [28].

[23] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungmana, P. Alken, M. Booth, and F. Rossi, "GNU scientific library reference manual (3rd ed.)," ISBN 0954612078, http://www.gnu.org/software/gsl/.

[24] T. Veldhuizen, "Blitz++, a high-performance vector mathematics library written in C++," (2011), http://blitz.sourceforge.net/.

[25] We have compiled it with -O3 option and gcc V4.6.3 compiler.

[26] We have used -O3 -arch=sm_20 --use_fast_math compilation options for the nvcc 4.2, V0.2.1221 compiler in all cases.

[27] https://developer.nvidia.com/cuda-toolkit-42-archive.

[28] https://bitbucket.org/ezeferrero/QEW.