REAPERS: A REAsonably PERformant Simulator for Qubit Systems

Chang Liu

Shanghai Center for Complex Physics, School of Physics and Astronomy, Shanghai Jiaotong University, Shanghai, China 200240

Abstract

A submitted program is expected to satisfy the following criteria: it must be of benefit to other physicists, or be an exemplar of good programming practice, or illustrate new or novel programming techniques which are of importance to computational physics community; it should be implemented in a language and executable on hardware that is widely available and well documented; it should meet accepted standards for scientific programming; it should be adequately documented and, where appropriate, supplied with a separate User Manual, which together with the manuscript should make clear the structure, functionality, installation, and operation of the program.

Your manuscript and figure sources should be submitted through Editorial Manager (EM) by using the online submission tool at https://www.editorialmanager.com/comphy/.

In addition to the manuscript you must supply: the program source code; a README file giving the names and a brief description of the files/directory structure that make up the package and clear instructions on the installation and execution of the program; sample input and output data for at least one comprehensive test run; and, where appropriate, a user manual.

A compressed archive program file or files, containing these items, should be uploaded at the "Attach Files" stage of the EM submission.

For files larger than 1Gb, if difficulties are encountered during upload the author should contact the Technical Editor at cpc.mendeley@gmail.com.

Keywords: keyword1; keyword2; keyword3; etc.

Email address: cl91tp@gmail.com (Chang Liu)
URL: github.com/cl91 (Chang Liu)

PROGRAM SUMMARY/NEW VERSION PROGRAM SUMMARY

Program Title:
CPC Library link to program files: (to be added by Technical Editor)
Developer's repository link: (if available)
Code Ocean capsule: (to be added by Technical Editor)
Licensing provisions: GPLv3
Programming language: C++20
Supplementary material:
*Journal reference of previous version:**
*Does the new version supersede the previous version?:**
*Reasons for the new version:**
*Summary of revisions:**
Nature of problem(approx. 50-250 words):
Solution method(approx. 50-250 words):
Additional comments including restrictions and unusual features (approx. 50-250 words):

References

[1] Reference 1

[2] Reference 2

[3] Reference 3

* Items marked with an asterisk are only required for new versions of programs previously published in the CPC Program Library.

The numerical simulations in this work use the same method described in [**?**]. We first map the SYK Majorana fermions to spin-1/2 operators using the Jordan-Wigner transformation. The OTOCs and Green functions are then computed by first constructing a Haar-random state, then evolving the state using the same Krylov subspace method described in [**?**], and finally computing the inner product of the initial state with the final state. Disorder averaging is then performed for all disorder realizations.

The main numerical effort of this work is focused on optimizing this calculation on nVidia GPUs. Previous work [**?** **?**] relied on a Python package

called `dynamite` [**?**], which is ultimately a simple frontend to the underlying C libraries that implement the basic matrix operations (`PETSc`) and on top of that, the Krylov algorithms (`SLEPc`). In the early stage of our work, we successfully reproduced the claim in [**?**] using `dynamite` that the dense SYK model with $N = 50$ fermions can be simulated (within a reasonably amount of time) on a single nVidia V100 graphics card. However, when examing the performance of these simulations, we realized that the software is sub-optimal in terms of both speed and memory: the observed power usage is consistently below half of the nominal maximum power of the graphics cards, and the video RAM usage is far above the theoretical minimum. To elaborate on the latter, for the matrix-free representation of the spin operators (called 'shell matrix' by `dynamite`), the theoretical minimum RAM usage is $(v+3)*\text{sizeof(state vector)}$, where v is the dimension of the Krylov subspace and the size of the state vector is $(\text{fp}/4) \cdot 2^{N/2-1}$ with fp being the floating point precision (fp = 32 or 64). The $v + 3$ vectors consist of $v + 1$ vectors used during the Krylov algorithm (including the initial state vector), plus a vector to store the final result of the Krylov evolution, plus a copy of the initial state vector for the purpose of computing the inner product (this last vector can be 'swapped' to host memory in order to save video memory usage, without a significant loss of performance). In our experiment, `dynamite` and its software stack consumed far more video memory than this theoretical minimum.

In order to simulate larger systems with less time, one of the authors (CL) initially tried to improve `dynamite` and its software stack. This soon turned out to be hopeless as the underlying `PETSc` library has a huge, ancient codebase, one that was first designed for CPU computing and later ported over to different parallel computing architectures, including GPGPU. For those unfamiliar with the codebase, any further performance work would most likely cost more engineering time than simply starting from scratch. Seeing that there is presently a lack of high-performance, GPU-enabled quantum spin model simulators, we therefore decided to write a C++ library that presents a similar `dynamite`-like interface, but with GPU computing as a first class citizen from day one. We focus on nVidia CUDA as it provides the most mature GPGPU ecosystem, but the library is designed to be portable over different parallel computing architectures. In fact, when CUDA is not available, the library will fallback to multi-CPU computing using OpenMP. The library will provide basic primitives such as spin operators and quantum state vectors, and on top of these we will reimplement the same restarted Krylov

algorithm [**?**] as in `SLEPc` for computing the state time evolution.

The design goals of the library, called `REAPERS` (short for 'a `REA`sonably `PER`formant `S`imulator for qubit systems'), are the following

1. Easy to use: both in terms of the programming interface, as well as with regard to package management.
   (a) In terms of the programming interface, we mimic that of `dynamite`, so existing users of `dynamite` have a minimal learning curve. We use advanced C++ template features to present a Python-like programming experience. The end users are NOT required to have any advanced C++ programming knowledge or know any CUDA optimization techniques. The latest C++20 features such as C++20 concepts are employed to improve the ergonomics of the library in situations where previous versions of C++ would lead to more cumbersome code or error diagnostics.
   (b) In terms of package management, we strive to have minimal external dependency. In fact, the only external library we depend on is `eigen3`, which is a header-only library by itself. This means that you do not need to build or install `eigen3`. You simply unpack the library package and point your compiler to where it is located.
   (c) Additionally, for the host code, our library is also header-only, meaning that you do not need to build or install the library. For the device code (ie. CUDA kernels), we ship a single `.cu` source file that you simply link with using `nvcc` at the final stage of the build process.
2. Fast: we strive to achieve the maximum performance offered by the hardware. More specifically:
   (a) As we will explain in detail below, we employ a low-level bit-string encoding of the spin operators in the Pauli basis, similar to the MSC (mask-sign-coefficient) representation used by `dynamite`, that allows us to reduce the mathematical operations of spin operators into simple bit manipulations that can be evaluated efficiently by hardware. In particular, for the action of spin operators on quantum states, this bit-string representation allows us to evaluate them in a 'trivially parallel' fashion, leading to highly optimized CUDA kernels.
   (b) We carefully manage memory allocations and deallocations using the language features provided by C++, in particular when it

comes to move semantics. We strive to avoid unnecessary copies and and reuse internal buffers to reduce unnecessary object allocations and deallocations. We also allow the user the freedom to choose when they want the internal buffers deallocated by providing an explicit garbage collection interface. As a result, our code can and very often do saturate the theoretical lower bound for the video memory usage, which is the key to our ability to reach $N = 62$.

3. Flexible: we provide maximum flexibility in terms of backend selection and floating point precision. More specifically,
   (a) If no CUDA is available, the library falls back to the CPU backend powered by OpenMP. If CUDA is available, the library by default will use the GPU backend but will also make the OpenMP-based CPU backend available. The user has the freedom to choose whether they want to use either CPU or GPU, or both CPU and GPU at the same time. This allows the user to maximize the utility of available computing resources.
   (b) Additionally, the library supports both single precision floating point and double precision floating point. Since most consumer grade nVidia GPUs have poor double precision performance but nonetheless decent single precision performance, this allows our library to run on consumer grade hardware without a significant loss of performance (by contrast, `dynamite` is double precision only and must run on professional compute-grade cards for acceptable performance). The user has the option to configure the default floating point precision, and can switch to a different floating point precision at runtime. Like the case of the computing backend, the user can in fact use both floating point precisions in the same program. The library uses C++ language features to safegard the mixing of floating point precisions so the user does not inadvertently use a lower (or higher) precision when they do not intend to.

The first part of the Supplementary Material will focus on the specifics of the performance work that we have alluded to in the previous paragraph. Once that is explained, we present the performance benchmark of the SYK simulations using our library code. We refrain from publishing speed ups against `dynamite` as this is inherently biased and unscientific, and will sim-

ply present the measured computational times in our experiments. Due to differences of hardware configurations, compiler directives, compiler versions, simulation parameters, etc., your results may look different from these numbers.

1. Optimization Techniques

The key to a high performance implementation of quantum spin model simulators is finding an efficient encoding of spin operators in such a way that its operations map to bit-level hardware primitives effectively. `dynamite` uses what it calls the Mask-Sign-Coefficient (MSC) representation of a spin operator which uses a list of 3-tuples, namely 'mask', 'sign', and 'coefficient' to represent a spin operator expanded in the Pauli basis. Our representation is inspired by this but we only use the 'mask' and 'coefficient' and eliminate the 'sign' part of the tuple as it proves to be unnecessary. Our representation is as follows: for a spin operator defined on a spin-1/2 chain of length L, the tensor products of Pauli matrices form a basis for the space of operators. In other words, for any spin operator S we can always expand it as

$$S = \sum_{i_0,i_1,\ldots,i_{L-1},i_{L-1}} \lambda_{i_{L-1}i_{L-1}\ldots i_1 i_0}\, \sigma_{i_{L-1}} \otimes \sigma_{i_{L-2}} \otimes \cdots \otimes \sigma_{i_1} \otimes \sigma_{i_0}$$

where each $i_k = 0, 1, 2, 3$ with

$$\sigma_0 = I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad \sigma_1 = \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$
$$\sigma_2 = \sigma_y = \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix}, \quad \sigma_3 = \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

For a single term in the sum, we use a 2-tuple, consisting of a bit-string which we will call `bits` in what follows, and a complex number which we will call `coeff`. The term $\lambda_{i_{L-1}i_{L-2}\ldots i_1 i_0}\, \sigma_{i_{L-1}} \otimes \sigma_{i_{L-2}} \otimes \cdots \otimes \sigma_{i_1} \otimes \sigma_{i_0}$ will be represented as

$$\texttt{bits} = s_{i_{L-1}} s_{i_{L-2}} \cdots s_{i_1} s_{i_0}$$
$$\texttt{coeff} = \lambda_{i_{L-1}i_{L-2}\ldots i_1 i_0}\, i^{\#(\sigma_i = \sigma_y)}$$

Here $s_{i_{L-1}} s_{i_{L-2}} \cdots s_{i_1} s_{i_0}$ is a concatenation of bit strings s_i where $s_0 = 00$, $s_1 = 01$, $s_2 = 11$, and $s_3 = 10$, and $\#(\sigma_i = \sigma_y)$ is the number of σ_y's in

$\{\sigma_{i_0}, \ldots, \sigma_{i_{L-1}}\}$. Note that all the bit-strings in this paper have the most significant bit in the left-most place.

The reason for encoding the σ matrices in this fashion should be apparent once the reader realizes that the action of σ_x on a spin site simply flips the qubit (ie. exchange the amplitudes for $|0\rangle$ and $|1\rangle$), and the action of σ_z simply negates the amplitude for $|1\rangle$. Ignoring the factor of i, the action of σ_y is simply that of σ_x followed by σ_z. The factors of i can be absorbed into the overall coefficient $\lambda_{i_{L-1}i_{L-2}\ldots i_1 i_0}$, because tensor products commute with scalar multiplications. Therefore to compute the action of a single term $\lambda_{i_{L-1}i_{L-2}\ldots i_1 i_0}\, \sigma_{i_{L-1}} \otimes \sigma_{i_{L-2}} \otimes \cdots \otimes \sigma_{i_1} \otimes \sigma_{i_0}$ on a state vector $|\Psi\rangle = \sum_{n\in\{0,1\}^L} \Psi_n |n\rangle$, we simply scan the bit-string `bits`, starting from the least-significant bit, and whenever we see a 1 in an even-numbered position, we flips the corresponding qubit, and whenever we see a 1 in an odd-numbered position, we negate the corresponding amplitude Ψ_n, and finally we multiply the resulting state vector by the overall coefficient `coeff`.

This leads to Algorithm 1 for computing the action of a generic spin operator S on a state $|\Psi\rangle$. Note that for this algorithm, due to the commutativity of additions (subject to the loss of precision of floating point arithmetic) and a lack of explicit data dependency, all three for-loops, including the inner-most loop, can be evaluated without regard to the order in which their bodies are executed. This algorithm is therefore "trivially parallel", in the sense that one can simply evaluate each iteration of a given loop in multiple computing units simultaneously, and then aggregate the results. Furthermore, the compiler is free to employ loop unrolling (especially for the inner-most loop), loop interchange (for the two outer loops), vectorization (including vectorization of bit operations), etc., for better cache locality and to maximize the utility of hardware units.

The advantage of the bit-level encoding of σ matrices is made even more evident when we compute the product of two spin operators. Note that up to an overall minus sign, the product of two σ matrices happens to coincide with the XOR (exclusive-OR) of their corresponding bit strings. Therefore, to compute the product of two spin operators expressed as a list of $\langle$`bits`, `coeff`$\rangle$ tuples, we first simply XOR the `bits` from either list and multiply the corresponding `coeff`'s. The overall minus sign is taken care of by scanning the `bits` and pick out those cases that produce a minus sign (this can be done either via simple pattern matching or through a lookup table). The results are then aggregated to construct the final answer. Finally, for spin operator addition, we simply merge the two lists, taking into account

Algorithm 1: Action of spin operator S on state $|\Psi\rangle$

input : Dimension of the spin chain length $L > 0$
Spin operator S, expressed as a list $\hat{S}$ of $\langle \texttt{bits}, \texttt{coeff} \rangle$ tuples.
Initial state vector $|\Psi\rangle$, expressed as a 2^L dimensional array $\Psi[n]$.
output: Final state vector $|\Phi\rangle = S|\Psi\rangle$.

```
for 0 ≤ m < 2^L do
    for each ⟨bits, coeff⟩ tuple in list Ŝ do
        n ← m;
        minus ← false;
        for each i-th non-zero bit of bits do
            if i is even then
                flip the i-th bit of n
            else if the i-th bit of n is set then
                flip minus
            end
        end
        if minus is false then
            Φ[m] += coeff * Ψ[n]
        else
            Φ[m] -= coeff * Ψ[n]
        end
    end
end
```

possible duplications of the `bits` field.

Building on Algorithm 1, we reimplemented the restarted Krylov subspace method [**?**] for computing the state time evolution $e^{-iHt}|\Psi\rangle$. This algorithm is the exact same one implemented in `SLEPc` and invoked by `dynamite`, but we take care to manage (video) memory allocations and deallocations in order to minimize both the amount of video memories used as well as the number of memory allocation and deallocation calls. This is enabled by the move semantics of the C++ language, which allows one to eliminate unnecessary copies of objects. We are therefore able to achieve a much larger N than the previous state of the art, due to the significantly higher performance and lower memory usage than `dynamite`. We will present the performance benchmarks in the next section.