Massively Parallel Markov Chain Monte-Carlo Software for Non-Linear Optimization and Equation Solving

Leonid Benkevitch, Kazunori Akiyama
MIT Haystack Observatory, Westford MA
benkev@haystack.mit.edu

ABSTRACT

This document describes the software `cuda_mcmc` designed to greatly accelerate solving various optimization problems using the NVIDIA graphics processing units (GPU) with the CUDA computing platform. We use a version of the Markov chain Monte-Carlo (MCMC) algorithm based on the Metropolis-Hastings sampler with replica exchange (MCMC-RE). We describe in detail both the algorithm itself and its mapping on the CUDA GPU architecture. Also, we explain how to write custom optimization/equation solving applications and provide several examples.

Contents

1. Introduction and Motivation

In the problems on finding global extrema of non-linear functions, the gradient methods of optimization are powerless before multimodal functions, non-continuous functions, or functions some (or all) of whose arguments can only take discrete or integer values. Statistical methods can be more helpful for this class of problems.

The method described in this paper initially was used in a radio astronomy problem of a black hole event horizon imaging (Benkevitch et al. 2016). The sparse (and scarce) data from the VLBI radio array pointed at the Sagittarius A* (Sgr A*) object, the super massive black hole at the center of our Galaxy, did not allow to use traditional imaging tools. However, some existing models could give approximate notion of the black hole look. We decided to create a model of the black hole image composed of simple geometric forms that allowed analytical Fourier transforms into the visibility domain. The model got the name "`xringaus`": its elements were "slashed" cylinders and a Gaussian. Varying the geometry and intensities of the forms, the model was supposed to take approximate images of many of the black hole states. This gave a chance to measure some of the real black hole parameters. Finding the model whose brightness image is the closest to the one of the real black hole required fitting the model in visibility domain to the observation data. This was a problem on finding the parameters $\mathbf{p} = (p_1, p_2, ..., p_{N_p})$ of the visibility model $x^{\rm mod}(\mathbf{p}, u_i, v_i)$ that would give the absolute (or global) minimum to the χ^2 calculated as

$$\chi^2(\mathbf{p}) = \sum_{i=1}^{N} \frac{(x^{\rm mod}(\mathbf{p}, u_i, v_i) - x_i^{\rm obs})^2}{\sigma_i^2}, \qquad (1)$$

where (u_i, v_i) were the coordinates of measured visibilities in the spatial frequency space, and the variance of every datum σ_i^2 played the role of its weight in the χ^2 sum. The model had 9 parameters, so had χ^2, and the problem was in finding the global minimum of the 9-dimensional hypersurface in the 10-dimensional space. The relief of this $\chi^2(\mathbf{p})$ hypersurface was extremely complex, with innumerable creases and local minima, so the gradient methods appeared useless.

We created a massively parallel and highly efficient, problem-specific optimization code based on the CUDA computing platform to be run on the NVIDIA graphic cards (GPU). The code would take just seconds to fit our model to the observations. Subsequent modifications turned the code to a universal tool capable of fast solving general non-linear optimization problems and even solving systems of non-linear equations.

We briefly describe here the statistical method we used in our software, the Markov Chain Monte-Carlo (MCMC) with Replica Exchange. It is based on the idea of the Bayesian inference.

2. Bayesian Inference

In the Bayes paradigm, the new information, the "evidence", is used to update the "prior" guess on the probability of a hypothesis, with the use of the well known Bayes' theorem

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \qquad (2)$$

From the prior probability of event A, $P(A)$, its posterior probability, $P(A|B)$, is inferred, having the evidence $P(B)$ and the likelihood $P(B|A)$. Instead of single events A and B and their (conditional) probabilities the probability distributions defined on generally multidimensional spaces can be considered. The Bayes' theorem (Eqn. (2)) can be used to calculate the posterior distribution at any point using the values of the three other distributions. In case of the model fitting this is used to obtain the posterior distribution of the fitted model parameters from the distribution of the observation data and the "prior" model parameter distribution. Numerically it can be implemented as probing the parameter space evenly enough to get the sufficiently dense set of points to plot the histogram of posterior distribution. Its maximum (or maxima) will be at the best-fit parameter values.

The optimization problem of fitting the radio observations $x_i^{\rm obs}$ to the visibility model $x^{\rm mod}$ by varying the model parameters $\mathbf{p}$ can be formulated as minimization of Eq. (1):

$$\chi^2(\mathbf{p}) = \sum_{i=1}^{N} \frac{(x^{\rm mod}(\mathbf{p}, u_i, v_i) - x_i^{\rm obs})^2}{\sigma_i^2} \to \min_{\mathbf{p}},$$

We assume that the specific set of observed data $x_i^{\rm obs}$ presented as a vector $\mathbf{x}$ (visibility amplitudes

and closure phases) is a sample from the multidimensional random variable, $\mathbf{X}$, with the probability density distribution $P(\mathbf{X})$. For a sample $\mathbf{x}$, $P(\mathbf{x})$ is a single number, the value of $P(\mathbf{X})$ at the point $\mathbf{x}$. Further, the lower case letters are used instead of the capitals, so $P(\mathbf{x})$ actually means $P(\mathbf{X})$.

Within the Bayesian framework both the observed data set $\mathbf{x}$ and the model parameter vector $\mathbf{p}$ are considered as statistically linked multidimensional random variables with their joint probability distribution

$$P(\mathbf{p}, \mathbf{x}) = P(\mathbf{p}|\,\mathbf{x})P(\mathbf{x}) = P(\mathbf{x}|\,\mathbf{p})P(\mathbf{p}), \quad (3)$$

where "," reads "and". Relationship (3) associates the probability densities named as follows:

- $P(\mathbf{x})$ the evidence,
- $P(\mathbf{p})$ the prior or prior distribution,
- $P(\mathbf{x}|\,\mathbf{p})$ the likelihood, and
- $P(\mathbf{p}|\,\mathbf{x})$ the inference or the posterior probability distribution.

In terms of causality, the object under observation is the cause, and the observation data is the effect. The Bayes' theorem allows us to rearrange the cause and the effect: using the known data $\mathbf{x}$, compute the *posterior* probability distribution $P(\mathbf{p}|\,\mathbf{x})$ that $\mathbf{x}$ is an effect of the object represented by our model with the parameter set $\mathbf{p}$. Thus, we pose a task to find not just a single set of the "optimal" model parameters, but the probability distribution of this set over the parameter space *given* the actual set of observation data. Of course, we are interested in such distributions for every single parameter, which are the marginal distributions of $P(\mathbf{p}|\,\mathbf{x})$. Below is shown that MCMC allows direct rendering of these marginal distributions.

So far, the Bayesian distributions have been treated in terms of the visibility model fitting to the observation data. However, nothing can prevent us from formulating any optimization problem using the Bayesian approach. Consider a problem of finding the optimal solution of an overdetermined system of non-linear equations $\mathbf{F}(\mathbf{p}) = \mathbf{x}$, where $N > N_p$, $\mathbf{F} = [f_1, f_2, ...f_N]^{\mathrm{T}}$, $\mathbf{p} = [p_1, p_2, ...p_{N_p}]^{\mathrm{T}}$, and $\mathbf{x} = [x_1, x_2, ...x_N]^{\mathrm{T}}$. The equations in the system are $f_i(\mathbf{p}) = x_i$, $i = \overline{1,\ N}$. Generally, the system has no roots. However, the problem can be set to find the optimal parameters $\mathbf{p}$ that provide the minimum discrepancy between the left- and the right hand sides. This is the well-known problem of least-squares solution (LSS), and it is formulated as

$$\chi^2(\mathbf{p}) = \sum_{i=1}^{N} \frac{(f_i(\mathbf{p}) - x_i)^2}{\sigma_i^2} \to \min_{\mathbf{p}}, \quad (4)$$

Here the "data" $\mathbf{x}$ can be just a vector of zeros, $\mathbf{x} \equiv \mathbf{0}$. Moreover, the same MCMC procedure can find *all* the roots $\mathbf{p}_k$ of a non-linear system of simultaneous equations $\mathbf{F}(\mathbf{p}) = \mathbf{0}$, if it is consistent and the size of vector $\mathbf{p}$ equals the number of equations N:

$$\chi^2(\mathbf{p}) = \sum_{i=1}^{N} \frac{f_i^2(\mathbf{p})}{\sigma_i^2} \to \min_{\mathbf{p}}. \quad (5)$$

The roots will sit at the locations k where $\chi(\mathbf{p}_k)$ is close to zero.

The problem of minimization of a single function of several variables $x = f(\mathbf{p})$ is quite trivial:

$$\chi^2(\mathbf{p}) = \frac{f^2(\mathbf{p})}{\sigma^2} \to \min_{\mathbf{p}}. \quad (6)$$

Here the "data" x is just a zero. The χ^2 is simply f^2/σ^2. In problems like the mentioned above, all the variances σ_i^2 can be set to the same nonzero value, for example, to unity. The Bayesian inference is obviously applicable to this class of problems.

The likelihood $P(\mathbf{x}|\,\mathbf{p})$ may be any positive function that reaches its maximum when the difference between the actual data and the model data becomes zero. We use a Gaussian likelihood

$$P(\mathbf{x}|\,\mathbf{p}) = \left(\prod_i \frac{1}{\sqrt{2\pi}\sigma_i}\right) e^{-\frac{1}{2}\chi^2}, \quad (7)$$

where χ^2 and σ are from Eq. (1). The prior, $P(\mathbf{p})$, is the distribution over the parameter space that represents our preliminary knowledge about the intervals where the parameter values could be present. The prior may not be very informative (for example, a uniform value within the allowed interval and zero outside), but it must always be provided.

Dividing (3) by $P(\mathbf{x})$ yields the Bayes' formula:

$$P(\mathbf{p}|\mathbf{x}) = \frac{P(\mathbf{x}|\mathbf{p})P(\mathbf{p})}{P(\mathbf{x})}, \tag{8}$$

with the searched for posterior parameter distribution on the left hand side, and computable probabilities on the right hand side. The value of $P(\mathbf{x})$, the probability density of the given observation data sample, can be calculated using the total probability law,

$$P(\mathbf{x}) = \int P(\mathbf{x}|\mathbf{p})P(\mathbf{p})\,\mathrm{d}\,\mathbf{p}. \tag{9}$$

For a given prior and a model the evidence $P(\mathbf{x})$ is always a single constant value as long as we work with the same data set: the integration over the whole parameter space removes all the variables. The evidence value can be used to compare the quality of different models. A "better" model will have larger $P(\mathbf{x})$. The Bayes' theorem thus takes the form

$$P(\mathbf{p}|\mathbf{x}) = \frac{P(\mathbf{x}|\mathbf{p})P(\mathbf{p})}{\int P(\mathbf{x}|\mathbf{p})P(\mathbf{p})\,\mathrm{d}\,\mathbf{p}}. \tag{10}$$

As we already said, the posterior distribution of all the parameters, $P(\mathbf{p}|\mathbf{x})$, is not as interesting as that of an individual parameter, $P(p_i|\mathbf{x})$, $p_i \in \mathbf{p}$, because it can provide the information on the mean value (or values, if multi-modal) and uncertainty of the estimate of the parameter p_i. Such individual distributions for every parameter p_i are, in effect, the marginal distributions, i.e. the results of integration of the total distribution $P(\mathbf{p}|\mathbf{x})$ over the parameter subspace spanned by all the parameters but p_i:

$$P(p_i|\mathbf{x}) = \int P(\mathbf{p}|\mathbf{x})\,\mathrm{d}p_1\,\mathrm{d}p_2 \ldots \mathrm{d}p_{i-1}\,\mathrm{d}p_{i+1} \ldots \mathrm{d}p_N. \tag{11}$$

The posterior distributions, $P(p_i|\mathbf{x})$, are not required to be normalized, so the strict equations (8) or (10) can be relaxed to a mere proportionality

$$F(\mathbf{p}|\mathbf{x}) \propto P(\mathbf{x}|\mathbf{p})P(\mathbf{p}), \tag{12}$$

where $F(\mathbf{p}|\mathbf{x}) \propto P(\mathbf{p}|\mathbf{x})$. Normalization of the F function would produce the posterior distribution $P(\mathbf{p}|\mathbf{x})$ and its marginals $P(p_i|\mathbf{x})$. However, statistical parameters of $P(p_i|\mathbf{x})$ – the means and the standard deviations, or qualitative conclusions about their forms – can be found directly from $F(p_i|\mathbf{x})$ without the normalization. The Metropolis-Hastings algorithm described here utilizes this fact. It draws many samples from the $P(p_i|\mathbf{x})$ distributions, and the result of optimization, $\mathbf{p}$, is obtained from the histograms built using the saved samples.

3. Markov Chain Monte Carlo as Implementation of Bayesian Inference

The Bayesian inference is applied to the optimization problems where the target functions are non-smooth multidimensional hypersurfaces, e.g., χ^2 in case of finding the best-fit model parameters. Because of the innumerable local minima, the random exploration of the whole domain and finding the approximations of parameter distributions appears to be preferred to the gradient descent methods. We use a strong algorithm named Markov Chain Monte Carlo (MCMC) with Replica Exchange (or Parallel Tempering) based on the improved Metropolis-Hastings algorithm.

3.1. Metropolis-Hastings Algorithm

The algorithm has three stages. First, an initial set of parameters $\mathbf{p}_0$ is randomly drawn from the prior distribution $P(\mathbf{p})$. The two other stages, the burn-in and the search, are essentially the same except at the burn-in stage the optimal steps for each parameter are picked. The iterations generate the Markov chain of the parameter tuples $\mathbf{p}_i = \left(p_{i,1}, p_{i,2}, \ldots p_{i,N_p}\right)$, and the more iterations, the better the $\mathbf{p}_i$ values approximate $P(\mathbf{p}|\mathbf{x})$. The Markov property, i.e. the dependence of the i_{th} chain element on the previous $(i-1)_{\mathrm{th}}$ element only is ensured by the method of their generation. At each iteration, a *proposal* model parameter set $\mathbf{p}'$ is generated from the proposal distribution $q(\mathbf{p}_{i-1};\,\mathbf{p}')$. The new proposal set is randomly accepted or rejected with a probability α,

$$\alpha = \min\left(\frac{P(\mathbf{x}|\mathbf{p}')P(\mathbf{p}')q(\mathbf{p}_{i-1};\,\mathbf{p}')}{P(\mathbf{x}|\mathbf{p}_{i-1})P(\mathbf{p}_{i-1})q(\mathbf{p}';\,\mathbf{p}_{i-1})}, 1\right). \tag{13}$$

In the Metropolis-Hastings algorithm the proposal distribution $q(\mathbf{p}_i;\,\mathbf{p}_j)$ generally does not necessarily *have to* be symmetric. However, the question of the best choice of q is still unclear. In order to simplify the calculations, we use a symmetric proposal distribution q as in the original Metropolis

algorithm. Here it is assumed a Gaussian distribution

$$q(\mathbf{p}_i;\,\mathbf{p}_j) = \prod_k^{N_p} \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left(\frac{(p_{j,k}-p_{i,k})^2}{2\sigma_k^2}\right). \tag{14}$$

Since $q(\mathbf{p}_{i-1};\,\mathbf{p}') \equiv q(\mathbf{p}';\,\mathbf{p}_{i-1})$, the acceptance probability is simplified to

$$\alpha = \min\left(\frac{P(\mathbf{x}|\,\mathbf{p}')P(\mathbf{p}')}{P(\mathbf{x}|\,\mathbf{p}_{i-1})P(\mathbf{p}_{i-1})}, 1\right). \tag{15}$$

Obviously, the numerator and denominator in (15) are the right hand sides of (12) for the new and previous $\mathbf{p}$, respectively, which in turn are proportional to the desired probability distribution. If the probability of proposal $\mathbf{p}'$ set is greater, then $\alpha = 1$, and $\mathbf{p}'$ becomes the new parameter set unconditionally. Due to the Gaussian likelihood, i.e. uncertainty in the observations (7), the acceptance probability α becomes

$$\alpha = \min\Bigg(\exp\left\{-\frac{1}{2}\left(\chi^2(\mathbf{x};\,\mathbf{p}') - \chi^2(\mathbf{x};\,\mathbf{p}_i)\right)\right\} \times \frac{P(\mathbf{p}')}{P(\mathbf{p}_i)}, 1\Bigg). \tag{16}$$

If the proposed parameter set were accepted only in case $\alpha = 1$ when the new point is necessarily better (with lower χ^2) than the previous one, the algorithm would be the basic random Monte-Carlo search. Unfortunately, the basic random search suffers from the "curse of dimensionality": the rejection probability exponentially grows with the number of dimensions. Hence, a basic random search of many parameters will last forever. Metropolis et al. (1953) suggested a way out: accept not only χ^2-better parameter sets, but also the sets that worsen χ^2, but accept it with the probability α. This technique ensures the "random walk" of $\mathbf{p}_i$, exploring the parameter space and visiting the volumes with better posterior probability more frequently than others. If the proposal set is rejected, the previous state will be repeated in the chain.

For models with many parameters the acceptance probability α tends to become small if all the parameters are stepped simultaneously, lowering the rate of acceptance and the overall algorithm efficiency. For this reason at each iteration we step only one parameter, keeping others constant. The following pseudocode describes one MCMC algorithm iteration:

1. Randomly choose $p_{i-1,j}$ from $\mathbf{p}_{i-1}$, parameter number j uniformly distributed;
2. Generate the j-th proposal parameter $p_{i-1,j}$ from the Gaussian distribution;
3. Calculate α and accept or reject $p_{i-1,j}$ with probability α;
4. Repeat 1-3 for N_p times; Memorize the newly generated state as $\mathbf{p}_i$.

The efficiency of this algorithm is also sensitive to the step size of proposal distribution (14), which is determined by the variance of the Gaussian distribution. If it is too small, most of the trial points are accepted, but the random walk is too slow to sample all the parameter space. Conversely, if the step is too large, most of the trial points are rejected and the MCMC algorithm can get stuck at a certain point for a long time despite the ability to make large jumps. Previous empirical studies recommend optimizing the step size to make the accept rate $\sim 25\%$ in high-dimensional cases (see references in Gregory 2005). The second, burn-in stage of MCMC is intended to adaptively adjust steps for all the parameters. After updating a j^{th} parameter $p_{i,j}$, if the accept rate of newest 100 trials is more than 30%, then the variance σ_j is multiplied by 1.01. Otherwise, if the accept rate of newest 100 trials is less than 20%, the STD σ_j is divided by 1.01.

3.2. Replica Exchange MCMC Algorithm

The described Metropolis-Hastings MCMC algorithm is quite suitable for our problems where the direct sampling is complicated or impossible. However, the original Metropolis-Hastings MCMC algorithm can fail to fully explore the target probability distribution, especially if the distribution is multi-modal with widely separated peaks. The algorithm can get trapped in a local mode and miss other regions of parameter space that contain significant probability.

The replica-exchange MCMC algorithm (also known as parallel tempering) is a result of improvement of the MCMC algorithm targeted to such complex multi-modal distributions. The

replica-exchange algorithm belongs to the class of "generalized-ensemble algorithms". It has been developed mostly in the past decade and recently was applied to some astronomical problems (Gregory 2005; Varghese et al. 2011; Benneke and Seager 2012). In this method a parameter β called "temperature" is introduced as

$$\begin{aligned} P(\,\mathbf{p}|\,\mathbf{x};\beta) &= \frac{P(\,\mathbf{x}|\,\mathbf{p})^{\beta}P(\,\mathbf{p})}{\int_{\mathbf{p}}\int_{\beta}P(\,\mathbf{x}|\,\mathbf{p})^{\beta}P(\,\mathbf{p})\,\mathrm{d}\,\mathbf{p}\,\mathrm{d}\beta} \\ &\propto P(\,\mathbf{x}|\,\mathbf{p})^{\beta}P(\,\mathbf{p}). \end{aligned} \tag{17}$$

When $\beta = 1$, it becomes the target posterior distribution. For the Gaussian likelihood (7) the latter can be rendered as

$$P(\,\mathbf{p}|\,\mathbf{x};\beta) \propto \exp\left(\beta L(\,\mathbf{x}|\,\mathbf{p})\right) P(\,\mathbf{p}), \tag{18}$$

where $L(\,\mathbf{x}|\,\mathbf{p})$ is a log-likelihood. The term "temperature" is borrowed from the canonical distribution $\exp(-\beta E)$ in statistical mechanics, where the absolute temperature is expressed using the "thermodynamic β" written as

$$\beta = \frac{1}{kT}, \tag{19}$$

so β is inversely proportional to the temperature. Using this analogy one can see that in Eq. (17) the log-likelihood $L(\,\mathbf{x}|\,\mathbf{p})$ plays the role of negative energy $-E$. Higher temperatures (those with smaller β) make the likelihood function flatter and also make the Metropolis-Hastings acceptance probability α higher, because

$$\alpha = \min\left(\exp\left\{-\frac{\beta}{2}\left(\chi^2(\,\mathbf{x};\,\mathbf{p}') - \chi^2(\,\mathbf{x};\,\mathbf{p}_i)\right)\right\} \times \frac{P(\,\mathbf{p}')}{P(\,\mathbf{p}_i)}, 1\right). \tag{20}$$

Thus, the Metropolis-Hastings sampling at higher temperatures enables exploration of wider ranges of the parameter space.

In the replica exchange MCMC algorithm, multiple Markov chains with different temperatures $(\beta_1, \beta_2, ...\beta_{N_\beta})$ including a chain with the lowest temperature $\beta = 1$ and different initial conditions are generated in parallel. The specific values of β_l usually span several orders of magnitude with logarithmic steps. As an example, 40 Markov chains may have $10^{-4} \leqslant \beta_l \leqslant 1$ with $l \in \overline{1\,..\,40}$. At each MCMC iteration, when the generation of new sets of parameters $\mathbf{p}_{i,l}$ is finished in all the chains, the newly generated elements of adjacent chains at the temperatures β_l and β_{l+1} are exchanged with a probability α written as

$$\alpha = \min\left(\frac{P(\,\mathbf{p}_{i,l+1}|\,\mathbf{x};\beta_{l+1})}{P(\,\mathbf{p}_{i,l}|\,\mathbf{x};\beta_l)}, 1\right). \tag{21}$$

The exchange procedure is repeated for $N_\beta - 1$ times, after which a new parameter $\mathbf{p}_{i+1}$ generation begins. Under the Gaussian likelihood (7) and the Gaussian proposal distribution (14) it becomes

$$\begin{aligned} \alpha = \min\Bigg(&\exp\Bigg\{-\frac{1}{2}(\beta_{l+1} - \beta_l) \\ &\times\Big(\chi^2(\,\mathbf{x};\,\mathbf{p}_{i,l}) - \chi^2(\,\mathbf{x};\,\mathbf{p}_{i,l+1})\Big)\Bigg\} \\ &\times\frac{P(\,\mathbf{p}')}{P(\,\mathbf{p}_i)}, 1\Bigg). \end{aligned} \tag{22}$$

In the higher temperature distributions ($\beta \ll 1$), radically new configurations are explored, while lower temperature distributions ($\beta \approx 1$) allow for detailed exploration of new configurations and local modes. The final inference on the model parameters is based on samples drawn from the target probability distribution ($\beta = 1$) only.

4. MCMC-RE GPU Implementation Details

A single Markov chain itself cannot be computed in parallel, for each new "chain link" requires computing the previous one. However, in the Replica Exchange version of MCMC several tens of Markov chains for different temperatures are to be computed, and this can be done in the hardware parallel threads on a GPU. Moreover, to multiply chances of the algorithm convergence to the solution in shorter time, several such multi-temperature multi-thread processes can be run in parallel.

In this section we describe the implementation internals most of which are hidden from the user "under the hood". However, they can help one understand how to use the upper-level, user-friendly software.

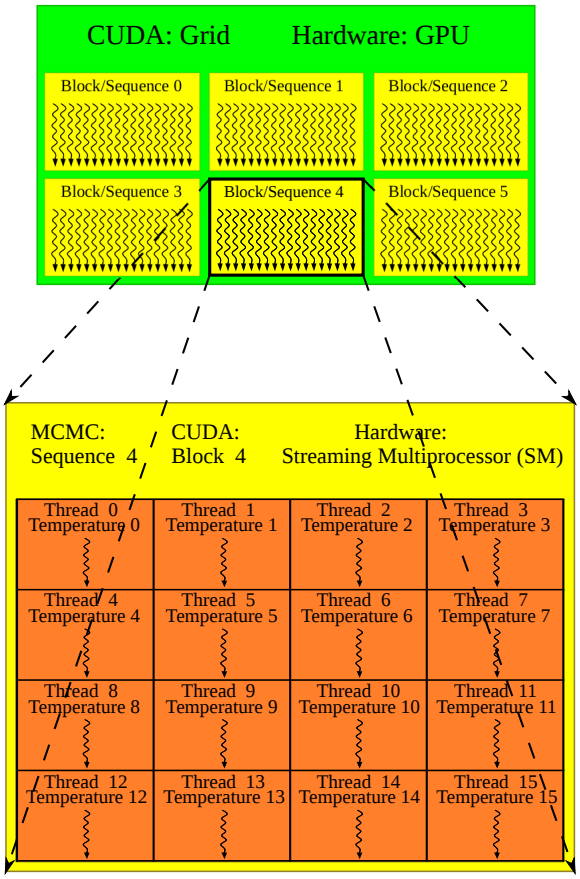

Fig. 1.— Two-level NVIDIA GPU architecture in terms of CUDA platform, GPU hardware, and MCMC-RE implementation.

4.1. Mapping MCMC-RE Algorithm on CUDA Architecture

The CUDA software platform provides a two-level thread organization: a grid of blocks, where each block contains multiple threads. It is shown in Fig. 1. In terms of the NVIDIA GPU hardware, each block of threads is executed on one of the Streaming Multiprocessors (SM). However, the architecture is totally scalable, each SM can execute multiple blocks. The numbers differ from model to model. For example, the GPU of GeForce GTX 670 videocard has 7 SMs, with 192 processor cores per an SM. Another videocard, GeForce GTX 1050 Ti, has 6 SMs, each containing 128 processor cores.

The massively parallel MCMC-RE implementation fits in naturally with the GPU architecture. The group of Markov chains running in parallel for all the set of temperatures make up a block of threads. We call such group a *sequence*. In terms of MCMC-RE the number of treads in one block is the number of temperatures N_β, and the number of blocks is the number of independent sequences $N_{\rm seq}$. The user can set these variables to any numbers. We prefer to use $N_\beta = 32$ because the unit of parallel execution, a *warp*, consists of 32 threads. Several warps can be simultaneously run on the same Streaming Multiprocessor (SM), so multiples of 32 are preferred for N_β. However, we do not see any advantages in having more than 32 temperatures. The value of $N_{\rm seq}$ may be arbitrary. It can be found experimentally as that providing the fastest computation. Intuitively clear, though, that $N_{\rm seq}$ should not be less than the number of hardware SMs, and it would be reasonable to set it to a multiple of the SM numver. In the subsections describing the arrays used by the MCMC-RE code one can notice that many of them have two additional dimensions `[nbeta,nseq]`, which corresponds to N_β and $N_{\rm seq}$.

The CUDA C language built-in variables allow a thread to always know in which block it is running and what is its own thread number within the block: `blockIdx` and `threadIdx`. Two more variables, `blockDim` and `threadDim`, contain the values of total number of blocks and total number of threads in a block, respectively. Therefore, in the MCMC-RE CUDA code the following, where possible, is observed:
`blockDim` $= N_{\rm seq}$, `threadDim` $= N_\beta$,
`blockIdx` $= i_{\rm seq}$, the sequence index, and
`threadIdx` $= i_\beta$, the temperature index.

The MCMC algorithm was initially used for fitting parameterized models to large arrays of data located at many coordinate points, like model of a celestial radio object observed with a radio interferometer. The data there are the visibilities, and the coordinates are u and v, the locations of visibilities in the spatial frequency domain. This is why the input arrays are called `dat` and `coor`. The data are provided to the algorithm in the `dat[ndat]` array, and the coordinates – in the `coor[ncoor]` array. Sometimes the model requires integer data, passed in the integer array `idat[nidat]`, and integer coordinates (like antenna numbers) passed in the `icoor[nicoor]` array. However, in other applications the arrays

`dat, coor, idat, icoor` can be used arbitrarily. Note that the whole of `dat[ndat]` content, and only it, is used to compute χ^2. On the same reason all the functions to be optimized are located in the file `model.cuh`. It contains the only C/C++ function `model()`. In order to create an application to solve a new problem, the user must write its own `model()` function.

The `model()` function calculates only one number and stores it in a specified location of the `datm[ndatm]` array. When writing the code of `model()` function it is important to understand the general picture of what is going on. The two arrays are of special importance: `dat[ ]` and `datm[ ]`. They are immediate mappings of the data described in Sections 1 and 2: `dat[i]` is $x_i^{\rm obs}$ and `datm[nbeta,nseq,i]` is $x_i^{\rm mod}$ (or f_i in another context). We are looking for the best, optimal set of the function parameters $\mathbf{p}$ that secure the global minimum for $\chi^2(\mathbf{p})$. The parameters, for instance, can be tuples of the `xringaus` model parameters
$\mathbf{p} = (Z_{\rm sp},\, R_{\rm ex},\, R_{\rm in},\, d,\, f,\, a,\, b,\, g_q,\, \theta)$.
At the beginning of each iteration we have $N_\beta \times N_{\rm seq}$ of such parameter sets. The `gen_proposal` function randomly selects in each tuple a single parameter and adds to it a random step, all in parallel. As all the $N_\beta \times N_{\rm seq}$ parameter tuples (or sets) are "stepped", and the function (or model) being optimized needs to be called many times to create the "model" data sets, each with the same number of elements, $N_{\rm dat}$, as in the `dat[ ]` array. This model calculation occurs totally in parallel: the `model()` function is called once for each one datum in `datm[ndatm]`, or in $N_\beta \times N_{\rm seq} \times N_{\rm dat}$ threads. This number of threads is $N_{\rm dat}$ times more than $N_\beta \times N_{\rm seq}$ used by the MCMC algorithm. Therefore, the kernel `calc_chi2_terms()` that calls `model()` and, in each thread, calculates a single term of χ^2 sums, is invoked in as many of 64 thread blocks as needed. At each thread `model()` is provided with the index into `pcur[ ]` to get the correct $(j_\beta, k_{\rm seq})$-th parameter set, the index into `dat[ ]`, and the index into `datm[ ]`, where the computed value should be saved.

After that, all the $N_\beta \times N_{\rm seq}$ "model" data sets are compared with the only data set in `dat[ndat]` by computing $N_\beta \times N_{\rm seq}$ values of χ^2, (in terms of the software) as

$$\chi^2_{j_\beta,k_{\rm seq}} = \sum_{i=1}^{N_{\rm dat}} (\mathrm{datm[j}_\beta, \mathrm{k_{seq}, i]} - \mathrm{dat[i]})^2 \times \mathrm{std2r[i]}, \tag{23}$$

and some of the new parameter tuples are accepted and some rejected, according to the Metropolis algorithm in the `spmp_mcmc()` function.

To compute χ^2 the algorithm requires the variance σ^2. The `std2r[ndat]` array should contain the reciprocals $1/\sigma^2$ of the data variances. This format is chosen out of the speed considerations, since the massive divisions by σ^2 in the χ^2 calculations are several times slower than the multiplications by $1/\sigma^2$.

The MCMC with replica exchange is implemented as a function `mcmc_cuda()` (in the file `mcmcjob.cu`) called by the Cython-compiled interface extension module `mcmc_interf.so` (the source Cython script is in `mcmc_interf.pyx`). In turn, `mcmc_cuda()` creates `mcmcobj` as an instance of the class `CMcmcFit` (defined in `mcmcjob.cuh`). This object at the time of its creation transfers the arrays to GPU and calls the class method `do_mcmc_on_gpu()` (also in `mcmcjob.cu`). It, in turn, calls the `run_mcmc()`, a CUDA C function determined with the rest of CUDA MCMC code in `gpu_mcmc.cu`.

In the Python codes the types of the parameters must strictly correspond to their C and C++ analogs. Note that the implementation of CUDA MCMC-RE described here uses the single precision 32-bit floats and 32-bit integers, `float` and `int`. Their Python analogs are `np.float32` and `np.int32`, where `np` is the standard `numpy` module. For the optimization problems the single precision appeared to be sufficient. Also, the GPU works faster with the single precision numbers, and they take less of the GPU memory.

4.2. Parallel MCMC-RE software hierarchy

The MCMC software has been written in C/C++ and Python and includes several levels. Their hierarchy is depicted in Fig. 2. The top level Python module `imgpu` (file `imgpu.py`) contains two classes, a universal solver `mcgpu` and its problem-specific heir, or subclass
`mcgpu_sgra(mcgpu)`, which is intended for solving

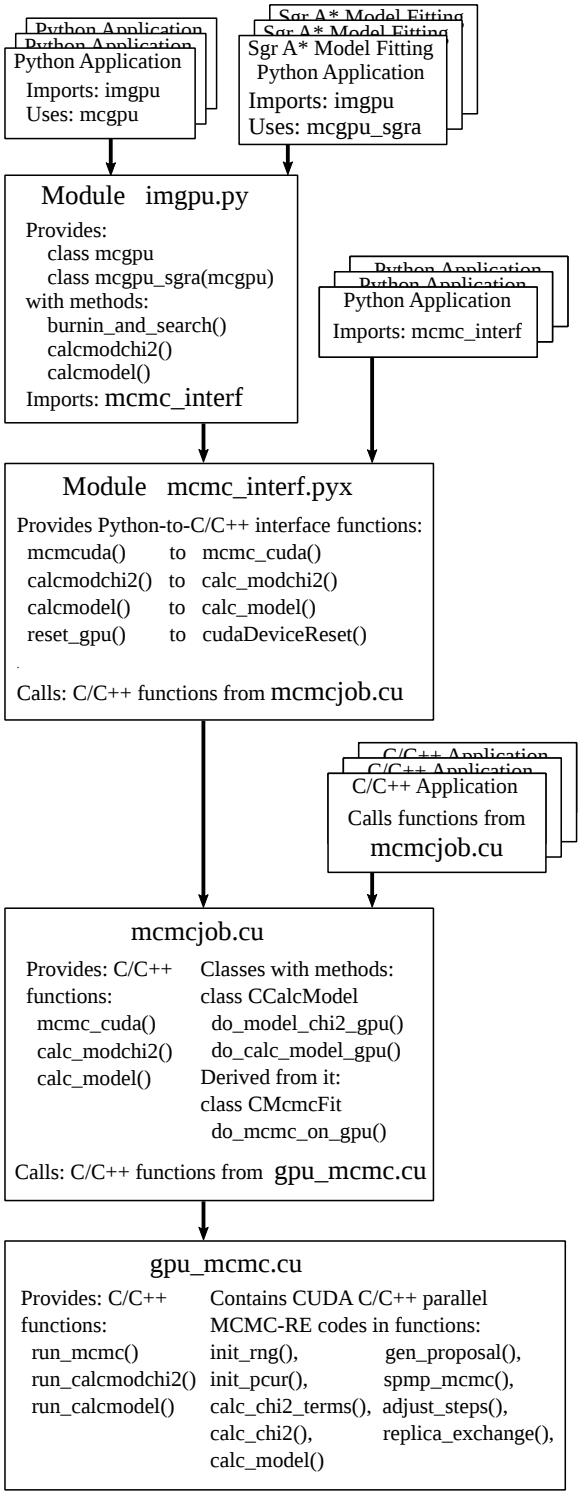

Fig. 2.— Parallel MCMC-RE software hierarchy.

a problem of fitting the Sgr A* black hole model to the observation data.

When a user creates an instance of either class, its constructor automatically creates and initializes all the variables and arrays needed for the MCMC execution. To start it, the user only needs to invoke the class method `burnin_and_search()`. The two other methods, `calcmodel` and `calcmodchi2()`, allow fast computation of the model outputs (in case of `mcgpu_sgra` – visibilities, closure phases) and χ^2 values for very large numbers of parameter sets in parallel on the GPU. The `imgpu` module provides maximum convenience and makes application programs way shorter.

The next level down in the hierarchy is presented by the `mcmc_interf` module (file `mcmc_interf.so`) built from the Cython source code `mcmc_interf.pyx`. It is a Python-C/C++ interface module whose functions, `mcmcuda()`, `calcmodel()`, `calcmodchi2()`, and `reset_gpu()` call underlying C/C++ functions. They provide convenience of programming in Python. However, in order to directly use the functions from `mcmc_interf` the user in the Python application has to correctly initialize all the variables and arrays.

The CUDA C/C++ file `mcmcjob.cu` provides functions and classes forming the underlying level used by `mcmc_interf`. The user writing the application codes in C/C++ can directly use the functions from `mcmcjob.cu` or classes from `mcmcjob.cuh`.

According to CUDA terminology, we call the CPU a "host", and the physically separate GPU unit, working as a coprocessor, a "device". We use NVIDIA graphic cards installed in the PCI-e slots. The host and the device possess their own separate memory spaces, the host memory and the device (GPU) memory. Allocation of the device memory and moving the data to and from the device is controlled by the CPU C code through the CUDA runtime API functions.

Preparing the data arrays, the device memory allocation, moving the data to the GPU device memory before the MCMC run and moving the generated data with the solution(s) back to the host memory after the MCMC run is performed by the `mcmc_cuda()` function in `mcmcjob.cu`. It uses the class `CMcmcFit` de-

rived from the class `CCalcModel`. Declaring `mcmcobj` as a class `CMcmcFit` object triggers the class constructor that allocates device memory, moves the data to the device, and launches the `do_mcmc_on_gpu()` method. The latter, in turn, calls `run_mcmc()` in `gpu_mcmc.cu`. Tho other functions, `calc_model()` and `calc_modchi2`, require only a small subset of the MCMC data. They create the `modobj` instance of the base class `CCalcModel`, whose constructor and destructor allocate/move much less data.

The last, lowest level of the hierarchy is located in the CUDA C/C++ file `gpu_mcmc.cu`. In order to launch the massively parallel MCMC processes in the GPU memory, the class method `CMcmcFit::do_mcmc_on_gpu()` calls the C program `run_mcmc()` from `gpu_mcmc.cu`. The GPU parallel processes are called *kernels* and have the attribute `__global__`. The MCMC implementation described here consists of several kernels, and they are called from the host code one after another, following the concept of "heterogeneous programming".

The MCMC "driver" program `run_mcmc()` on entry gets two arguments, both are the pointers to the `CMcmcFit` class objects. The first one, `mc_h`, points to the host memory and the object it points at has the host pointers to the host data arrays. The second argument, `mc_d`, is a pointer into the GPU device memory, and its class member pointers are into the device memory. This device pointer is further used as a parameter for the kernels: due to it the kernels running on the GPU have full access to all the data in the device global memory.

4.3. Parameters of the MCMC-RE Algorithm

Below is the list of parameters accepted by the `mcmc_cuda()` function. It resides in the `mcmcjob.cu` file and can be called directly from a C/C++ code. From Python it can be called from the module `mcmc_interf` as

4.3.1. Input Parameters

`mcmc_interf.mcmcuda()` with the same list of parameters.

`nptot`: total number of the parameters to be optimized by MCMC, or the full problem dimensionality. Part of the parameters in the `ptotal[nptot]` array can be made immutable for a particular MCMC run by setting zeros at the corresponding locations in the descriptor array `pdescr[nptot]`.

`nprm`: the number of parameters to be optimized by MCMC in a specific run. It is possible to "freeze" values of a part of the problem parameters making them immutable constants, and optimize only the rest of them. The frozen (immutable) parameters must be marked by zeros at their positions in the descriptor array `pdescr[nptot]`. Thus `nprm` is the number of non-zero elements in `pdescr[ ]`.

`ndat`: number of floating point data.

`ncoor`: number of floating point coordinates.

`nidat`: number of integer data.

`nicoor`: number of integer coordinates.

`nseq` $\equiv N_{\rm seq}$: number of independent parallel processes of model fitting. In CUDA framework it means the number of thread blocks. In the NVIDIA GPU hardware one block of threads is executed on one "Streaming Multiprocessor" (SM). For example, GTX 670 has 7 SMs.

`nbeta` $\equiv N_\beta$: number of "temperatures" in the MCMC algorithm. In CUDA framework it means the number of parallel theads per block. In NVIDIA GPU the treads in a SM are executed in units of "warps" of 32 threads each executing the same instruction at a time.

`seed`: an arbitrary 64-bit unsigned integer (np.uint64) seed for the CUDA random number generator.

`pdescr[nptot]`: array of parameter descriptors with possible values 2 - angular parameter (radians), 1 - nonangular parameter. 0 - this value excludes the corresponding parameter in `ptotal[nptot]` from optimization.

`ptotal[nptot]`: array of model parameters. The `ptotal[ ]` values at the locations where `pdescr[ ]` is nonzero are ignored. The values where `pdescr[ ]` is zero are used in the model.

`pmint[nptot]`,

`pmaxt[nptot]`: minimum and maximum values for the model parameters. The parameters are searched only inside of the nptot-dimensional rectangular parallelepiped determined by `pmint[ ]`

and `pmaxt[ ]`. This parallelepiped determines the "prior".

`ivar[nprm]`: maps [nprm] optimized parameters on ptotal[nptot]. ivar[nprm] has indices of the ptotal[nptot] parameters whose descriptors in pdescr[nptot] are non-zero.

`invar[nptot]`: maps ptotal[nptot] on the varied parameters [nprm]. If some of the parameters are immutablem invar[ ] must contain -1 values at the corresponding positions.

`beta[nbeta]`: "temperature" values in descending order from 1. Recommended are the values between 1 and 0.0001 falling off exponentially, i.e. as a geometric progression with common ratio $0.0001^{\frac{1}{\texttt{nbeta}-1}}$, from 1 down to 0.0001.

`dat[ndat]`: input floating point data (e.g. visibility amplitudes, phases etc). In the software, `dat[i]` is the same as $x_i^{\rm obs}$ described in Sections 1 and 2.

`idat[nidat]`: input integer data.

`coor[ncoor]`: input floating point coordinates (e.g. u and v pairs).

`icoor[nicoor]`: integer coordinates (e.g. antenna numbers)

`std2r[ndat]`: the reciprocals of the data variances, $1/\sigma_i^2$.

`pcur[nbeta,nseq,nptot]`: initial parameter values for all temperatures and all sequences. Can be zeros or random numbers or the coordinates of the prior center. During MCMC run is used for the current parameter values.

`tcur[nbeta,nseq]`: integer values of the "temperature" indices. On entry it should contain nseq columns of cardinals from 0 up to `nbeta-1`.

`n_cnt[nprm,nbeta,nseq]`: numbers of Metropolis trials for each mutable parameter, each temperature, and each sequence. Should be initialized to ones.

`nadj`: number of sets for calculating acceptance rates. Recommended value 100.

`npass`: number of times the `model()` function must be called. Sometimes the model computation requires several passes. The data generated and saved in a previous pass are used in the next one. The `model()` function can determine at which pass it has been called by its parameter `ipass` taking values from 0 to `npass-1`.

`pstp[nprm,nbeta,nseq]`: standard deviations of the parameter steps for each mutable parameter, each temperature, and each sequence. An actual step is generated as a normally distributed random number with the STD from `pstp[ ]`.

`imodel`: an integer parameter passed to the `model()` function. The usage is arbitrary. For example, it may be a model number to select between several models.

`nburn`: number of the burn-in iterations. During the burn-in phase the steps for each parameter, each temperature, and each sequence are adjusted, and the transients fall off.

`niter`: number of optimization iterations.

`ndatm`: must be equal `nbeta*nseq*ndat`.

4.3.2. Workspace Arrays

`datm[nbeta,nseq,ndat]`: computed model data for the parameter tuples $\mathbf{p}_{j_\beta,k_{\rm seq}}$. In the software `datm[ ]` represens $x_i^{\rm mod}(\mathbf{p})$ or $f_i(\mathbf{p})$ described in Sections 1 and 2. However, due to the multiple parallel threads where $\chi^2(\mathbf{p})$ is computed in many sequences $N_{\rm seq}$ for many temperatures N_β, for each particular $\chi^2_{j_\beta,k_{\rm seq}}$ calculation only a slice of `datm[ ]` array of the length $N_{\rm dat}$ is involved:
`datm[j_beta,k_seq,:]`.
Normally `datm[ ]` is used as a one-dimensional array `datm[ndatm]`, $N_{\rm datm} = N_\beta \times N_{\rm seq} \times N_{\rm dat}$.

`chi2m[nbeta,nseq,ndat]`: χ^2 terms for each model data element, each temperature, and each sequence.

`rndst[nbeta,nseq,48]`: unsigned 8-bit integer `(np.uint8)` array of states for the random number generator.

`flag[nbeta,nseq]`: 0 means the selected parameter is outside of the prior, or the parameter did not pass the alpha-test. 1 means the selected parameter is OK.

`chi2c[nbeta,nseq]`: The χ^2 values for each temperature and each sequence memorized from the previous Metropolis step to be compared with the new ones.

`ptent[nbeta,nseq]`: tentative parameter values for each temperature and each sequence.

`ptentn[nbeta,nseq]`: tentative parameter indices for each temperature and each sequence.

4.3.3. Output Parameters

`pout[nprm,nbeta,nseq,niter]`: mutable parameter values found by the MCMC for all the temperatures, sequences, and iterations. As a rule, only the coldest temperature is used, i.e. `pout[nprm,0,nseq,niter]`.
The optimum parameter set is in `pout[ ]` at the location corresponding to the minimum χ^2 in the `chi2[nseq,niter]` array. Usually these arrays reshape to 1D form:

```
po = pout[:,0,:,:].reshape((nprm,nseq*niter))
c2 = chi2.flatten()
```

and then the best parameter set is found as `po[c2.argmin()]`.
Also, the histograms of each parameter can provide important information:
`hist(po[0,:], 50, color='b') grid(1)` - for the 0-th parameter and so on.

`chi2[nseq,niter]`: χ^2 values for the highest temperature ($\beta = 1$), for all the sequences and iterations.

`n_acpt[nprm,nbeta,nseq]`: counting numbers of accepted proposal sets in the Metropolis-Hastings algorithm.

`n_exch[nbeta,nseq]`: counting numbers of exchanged adjacent temperature chanins in the replica exchange algorithm.

`n_hist[nadj,nprm,nbeta,nseq]`: counting numbers of "accept" of proposal set in Metropolis-Hastings algorithm

4.3.4. Output Parameters

4.4. Parallel MCMC-RE Algorithm in CUDA C/C++

At each call, a kernel needs two special parameters enclosed in the `<<<>>>` brackets: the number of threads in the block, and the number of blocks in the grid. We run `nseq` independent MCMC processes ("sequences"), one per a block, and each process runs `nbeta` parallel threads, one per a temperature. Below the MCMC algorithm is described as a series of kernel calls with comments. These codes are in the file `gpu_mcmc.cu`, function `run_mcmc()`.

1. Initialize the random number generators for all the $N_{\rm seq} \times N_{\rm beta}$ threads:

```
init_rng<<<nseq,nbeta>>>(mc_d);
```

2. Initialize the optimized parameters for every sequence and every temperature by assigning to them the random values uniformly distributed inside of the prior. They become the current parameters:

```
init_pcur<<<nseq,nbeta>>>(mc_d);
```

3. Compute all the χ^2 terms like
$(\mathrm{datm}[i_{\rm beta}, i_{\rm seq}, i_{\rm dat}] - \mathrm{dat}[i_{\rm dat}])^2/\sigma^2[i_{\rm dat}]$,
where `dat[ndat]` are the observation data (visibility amplitudes and closure phases), and
`datm[nbeta,nseq,ndat]` are the results of computing `nbeta x nsec` models. The χ^2 terms are saved in the array `chi2m[nbeta,nseq,ndat]`. This computation is fulfilled in parallel on `nblocks` with `blocksize` threads per a block, arbitrarily set to 128. So,

```
nblocks = nbeta*nseq*ndat/blocksize + 1.
```

The kernel is called in a loop `npass` times with `ipass` running from 0 to `npass-1`:

```
calc_chi2_terms<<<nblocks,
            blocksize>>>(mc_d, ipass);
```

4. From the terms in `chi2m[nbeta,nseq,ndat]` compute χ^2 and store the values in the `chi2m[nbeta,nseq]` array:

```
calc_chi2<<<nseq,nbeta>>>(mc_d);
```

After the initialization the two MCMC stages follow: burn-in and main. At the burn-in stage the step sizes in `pstp[nprm,nbeta,nseq]` for each parameter in each sequence, and each temperature are adjusted based on the parameter acceptance/rejection rate calculated from the history being accumulated in
`n_hist[nadj,nprm,nbeta,nseq]`. The burn-in stage requires a few hundred iterations, which is usually less than that for the main stage. Both stages are very similar. Actually, the burn-in is a "dry run" of the MCMC-RE, where the steps are adjusted and the `replica_exchange()` kernel does not save in the array
`pout[nprm,nbeta,nseq,niter]`
the new parameter sets found. At the burn-in stage the parameter `burn` of
`replica_exchange(mc_d, burn, itr)` must be non-zero. At the main stage `burn` is set to zero.

4. Burn-in stage:

```
burn = 1;
for (itr = 0; itr < nburn; itr++) {
  for (ipm = 0; ipm < nprm; ipm++) {
```

In `gen_proposal()` to a randomly chosen parameter a random "Gaussian" step with the standard deviation from `pstp[nprm,nbeta,nseq]` is added. If the parameter gets outside of the prior, it is flagged by 0 in `flag[nbeta,nseq]`. Otherwise its index is saved in `ptent[nbeta,nseq]` as "tentative":

```
gen_proposal<<<nseq,nbeta>>>(mc_d);
```

Compute χ^2 terms `npass` times for `ipass` from 0 to `npass` - 1:

```
calc_chi2_terms<<<nblocks,
       blocksize>>>(mc_d, ipass);
```

Single Parameter Metropolis MCMC. The proposed tentative parameter is accepted with a probability α calculated by the formula in Eq. (16):

```
spmp_mcmc<<<nseq,nbeta>>>(mc_d);
```

Adaptive step adjustment for all the parameters. The steps are generated as Gaussian random values with the standard deviations (STD) individual for each parameter. If for a j^{th} parameter the acceptance rate of the latest 100 trials is more than 30%, then its STD σ_j is multiplied by 1.01. Otherwise, if the accept rate of newest 100 trials is less than 20%, the variance σ_j is divided by 1.01.

```
adjust_steps<<<nseq,nbeta>>>(mc_d);
```

In `replica_exchange()` two randomly selected adjacent temperature chains exchange their parameters and χ^2 with the probability calculated as given in Eq. (22). This is repeated $N_\beta - 1$ times. Unfortunately, it cannot be parallelized into N_β temperature threads, so N_{seq} independent MCMC processes execute this simultaneously in blocks with one thread per block:

```
  replica_exchange<<<nseq,1
                  >>>(mc_d, burn, itr);
}
```

5. Main stage. It is not commented because comments to the burn-in stage explain both stages. The only two differences are:
– the calls to `adjust_steps()` are absent;
– the second parameter of `replica_exchange()`, burn, is set to zero.
This means that there is no step adjustment and that the output parameters `pout[ ]`, `chi2[ ]`, and `tout[ ]` are updated at each algorithm iteration.

```
burn = 0;

for (itr = 0; itr < niter; itr++) {
  for (ipm = 0; ipm < nprm; ipm++) {

    gen_proposal<<<nseq,nbeta>>>(mc_d);

    for (ipass = 0; ipass < npass;
                            ipass++) {
      calc_chi2_terms<<<nblocks,
            blocksize>>>(mc_d, ipass);
    }
    spmp_mcmc<<<nseq,nbeta>>>(mc_d);
  }
  replica_exchange<<<nseq,1
                  >>>(mc_d, burn, itr);
}
```

6. The results are moved to the host memory. The parameters of the model, the coordinates of the absolute minimum, or the roots found in the MCMC main stage run are in the array `pout[nprm,nbeta,nseq,niter]`, and the χ^2 values for the coldest temperature ($\beta = 1$) are in the array `chi2[nseq,niter]`. Using the latter, the best result in `pout[ ]` can be found at the location of the χ^2 minimum.

5. Using the MCMC-RE Software

In order to use the software for fast numerical solving a specific optimization problem, the user should write two programs: a script in Python and

a CUDA C/C++ function `model()`. The Python script is intended for running the MCMC-RE software, which, in turn, calls `model()` in a highly parallel mode to compute the function(s) to be optimized or the equations to be solved.

5.1. Installation

The prerequisites include:

- an NVIDIA video card of the compute capability 2.0 or greater
- the NVIDIA CUDA Toolkit and NVIDIA driver installed
- gcc, g++
- Python
- Cython, the C/C++ extension to Python

The current version of the CUDA MCMC-RE software works in Linux only. Its installation does not require the root privileges. The software is in `cuda_mcmc.tar.gz` archive. Unpack it with the command

```
$ tar xvfz cuda_mcmc.tar.gz
```

This creates the directory `cuda_mcmc` with the following structure:

```
cuda_mcmc
 |-- doc
 |-- src
 +-- template
 +-- examples
      |-- search_Rosenbrock
      |-- sgra_model_fit
      +-- solvsys
```

Enter the `cuda_mcmc` directory and run `make` command:

```
$ cd cuda_mcmc
$ make
```

The command creates two directories if they have not existed yet in the user's home directory: `~/lib64/python` and `~/bin`. They will be used to install Python modules and binary codes, respectively. These directories must be appended to the `PYTHONPATH` and `PATH` environment variables. One more environment variable, `CUDA_MCMC`, must point at the directory `cuda_mcmc` from which `make` has been invoked. The following three lines are added to the end of user's `~/.bashrc` file:

```
export PATH=$PATH:~/bin
export PYTHONPATH=$PYTHONPATH:~/lib64/python
export CUDA_MCMC=<path-to-mcmc_cuda>
```

When `make` exits, it asks the user to run the command

```
$ source ~/.bashrc
```

5.2. Working on a Project

Just create a directory for your project in any location of your directory tree and copy there the contents of `$CUDA_MCMC/template`. Along with the `Makefile` and `README.txt` it contains two other files: `model.cuh` and `template.py`.

The user is responsible for writing two programs: the backend and the frontend. The backend is the `model()` function contained in the `model.cuh` file. This is a CUDA kernel that runs inside the GPU and includes the codes for computing the user's function(s) to be optimized. The frontend is a Python or C/C++ program that uses the primitives of one of the MCMC-RE software layers to run the parallel optimization process and to present the results in printed and graphic forms: plots, histograms etc.

Rename `template.py` into a desired name. This is a frontend script that contains the general scheme of running the MCMC-RE software with the use of the high-level Python class `Mcgpu`. The backend, the `model.cuh` file, should be also edited to include the codes for user's function(s).

When both Python script and `model.cuh` are ready, just run `make`. The CUDA software in `$CUDA_MCMC/src` will be compiled with `model.cuh` included from your project directory and the Python extension module `mcmc_interf.so` will be built and copied to `~/lib64/python`. Running the Python script will start the MCMC-RE algorithm with your project's `model.cuh`.

5.3. Writing the model() function to be optimized

For each individual fitting, optimization, or equation solving problem the user needs to write the `model()` function, compile it and link it with the rest of the code. This function computes the optimized expression(s) working in parallel in all the threads of all the blocks. This function should be determined in a CUDA header file `model.cuh`.

The `model()` function is only called from the GPU device, from the kernel `calc_chi2_terms()`, so it has the attribute `__device__` (i.e. callable from the GPU device only). Initially this function was intended to compute the model visibilities and closure phases of the supermassive black hole at our Galaxy center observed with the VLBI array, hence the function name. However, it can be written to compute the left hand side of a vector or scalar expression like $\mathbf{F}(\mathbf{x}, \mathbf{v}) = 0$. The MCMC is able to search for the absolute minimum of the expression, or search for its multiple minima in the space determined by the parameter vector $\mathbf{x}$, while $\mathbf{v}$ are some arbitrary parameters set by the user. If $\mathbf{F}(\mathbf{x}, \mathbf{v}) = 0$ is a consistent system of equations, MCMC is able to find all of its roots.

Each `model()` call computes only one `datm[ ]` element. The function parameters:

```
iret = model(mc, idat, ipt, idatm, ipass);
```

`CCalcModel *mc`: pointer to the `mc` object whose data structure contains all the working data for all the blocks and all the threads. All the mentioned arrays and variables other than the immediate `model()` parameters are the `mc` members and are accessed using the "`->`" operator, for example `mc->dat`. We will omit "`->`" before the arrays assuming it will be used in the real code.

`int id`: index into `dat[ ]`.

`int ipt`: index of the first parameter in a set in `pcur[nbeta,nseq,:]` for the specific temperature `ibeta == threadIdx.x` and specific sequence `iseq == blockIdx.x`. The first model parameter is `pcur[ipt]`, and `pcur[ipt+mc->nptot-1]` is the last parameter. If `i` is a parameter index, from 0 to `mc->nptot-1`, then `pcur[ipt+i]` is the `i`-th parameter, or `pcur[ibeta,iseq,i]` in the Python code.

`imd`: a "through index" that points at the location in `datm[ ]` that corresponds to the location in `dat[id]`, i.e. `datm[ibeta,isec,id]`. However, `imd` treats it as a one-dimensional array and must be used as `datm[imd]`. This array contains the calculated model data that will be compared with the user-provided data in `dat[ ]`. The `imd` index is only used to save the result at `datm[imd]` before the return.

`ipass`: pass number, starting from 0. The function can be called multiple times, or in many passes. For example, at pass 0 the visibility amplitudes and phases are calculated. At pass 1 the closure phases from already available phases are calculated.

The result should be stored at the `datm[imd]` location.

Also, `iret = model()` returns are treated as follows.
`iret == 0`: no results.
`iret == 1`: success.
`iret == 2`: the computed `datm[imd]` is an angle in radians.

The two arrays are of special importance: `dat[ ]` and `datm[ ]`. They are immediate mappings of the data described in Sections 1 and 2: `dat[i]` is $x_i^{\rm obs}$ and `datm[nbeta,nseq,i]` is $x_i^{\rm mod}$ (or f_i in another context). These arrays are used beyond `model()` to compute χ^2 values in `chi2c[nbeta,nseq]` according to Eq. (23).

Below is an example of `model()` written for solving a quite elementary (but hard for the gradient methods) optimization problem: finding the minimum of the "Rosenbrock's valley" or Rosenbrock's banana function

$z = (1 - x)^2 + 100(y - x^2)^2$.

This is the code:

```
#define X (mc->pcur[ipt])
#define Y (mc->pcur[ipt+1])
#define Z (mc->datm[imd])

__device__ int
model(CCalcModel *mc, int id, int ipt,
                  int imd, int ipass) {
  if (id == 0)
    Z = pow(1.f - X, 2)
```

```
      + 100.f*pow(Y - pow(X, 2), 2);
  return 1;
}
```

In this specific problem we have only two parameters and only one data element, set to zero. The z value is calculated in many sequences and many temperatures, but it is always compared with zero. The χ^2 here is actually z^2, but it is what we need: minimize z varying x and y.

5.4. Writing Python Script: Class Mcgpu

As the CUDA MCMC-RE software has a multilayer architecture shown in Fig. 2, the functions provided by each of the layers can be used to run the optimization code linked with the custom `model()` function.

1. The high-level Python class `Mcgpu` and its methods, module `imgpu`.
2. Python API functions, module `mcmc_interf`.
3. C++ classes CCalcModel, CMcmcFit and their methods, files `mcmcjob.cuh`, `mcmcjob.cu`.
4. CUDA C/C++ kernels (see Subsection 4.4)

We do not consider using the last two layers. The Python API requires preliminary initialization of a significant number of arrays and parameters (see Subsection 4.3 "Parameters of the MCMC-RE Algorithm") and is not recommended. The scripts using API are provided in the directories `example/search_Rosenbrock/` and `example/solvsys/`. Both have the "`_api.py`" endings.

The high-level class `Mcgpu` from the module `imgpu`, on the contrary, is easy to use. When instantiated, its constructor creates all the arrays and completes the necessary initializations "under the hood". To start the MCMC-RE optimization, the user simply calls the class method `burnin_and_search()` without parameters. Here we show the steps coded in `template/template.py`:

```
import imgpu

# Create the 'solver' object of
# the 'Mcgpu' class:
solver = imgpu.Mcgpu(pdescr=pdescr1, \
         pmint=pmint1, pmaxt=pmaxt1)

# Start parallel MCMC-RE on the GPU:

solver.burnin_and_search()
```

The results are among the `solver` object attributes. For example, the optimal parameters found by the algorithm are in the array `solver.pout` at the location(s) determined by the minimum values in another array, `solver.chi2`.

All the instantiation parameters as well as the class attributes (or "class members" in C++ terminology) have *exactly* the same names and meanings as those of the MCMC-RE parameters described in Subsection 4.3. The only two exceptions are `Mcgpu.pout_4d` and `Mcgpu.chi2_2d`: they correspond to `pout` and `chi2` of the algorithm. This is because the method `Mcgpu.burnin_and_search()` transfers their contents into more convenient 2-dimensional `Mcgpu.pout[nprm,nseq*niter]` and 1-dimensional `Mcgpu.chi2[nseq*niter]`.

The class `Mcgpu` instantiation only requires three mandatory parameters:

`pdescr[nptot]`: The array of parameter descriptors. Its size, `nptot`, determines the model dimensionality inside of `Mcgpu`. The possible values are 2 - angular parameter (radians), 1 - nonangular parameter, 0 - this value excludes the corresponding parameter in `ptotal` from optimization.

`pmint[nptot]` and `pmaxt[nptot]`: for any i-th of the `nptot` dimensions `pmint[i]` and `pmaxt[i]` determine the lower and upper limits between which the i-th parameter will be searched.

Here is the full list of other parameters that can be used at `Mcgpu` instantiation.

`ptotal[nptot]`: The array of model parameters. The `ptotal[ ]` values at the locations where `pdescr[ ]` is nonzero are ignored. The values where `pdescr[ ]` is zero are used in the model as constants and not optimized.

`dat[ndat]`: The array of `np.float32` data to compare with the model output to `datm[ ]` via computing χ^2. In case of a function minimum search `dat[ ]` can have any single value. If not specified, 1-element array with zero value is created by default.

`coor`: Array of arbitrary size of the `np.float32` data the user wants `model()` to have access to. If not specified, 1-element array with zero value is created by default.

`idat`: Array of arbitrary size of the `np.int32` data the user wants `model()` to have access to. If not specified, 1-element array with zero value is created by default.

`icoor`: Array of arbitrary size of the `np.int32` data the user wants `model()` to have access to. If not specified, 1-element array with zero value is created by default.

`std2r[ndat]`: the reciprocals of the data variances, $1/\sigma_i^2$. If not specified, filled in with ones by default.

`beta[nbeta]`: the array of temperatures. If not specified, by default is filled in with a falling off geometric progression with common ratio `(betan/beta1)`$^{\frac{1}{\texttt{nbeta}-1}}$.

`beta1`, `betan`: First and last values in `beta[nbeta]`. If not specified, the defaults are `beta1 = 1.` and `betan = 0.0001`. The temperature corresponds to $\beta = 1/kT$. The lowest temperature, `beta[0]`, is usually set to `beta1 = 1.`, the highest, `beta[nbeta-1]`, can be set to several orders of magnitude less, something like `betan = 0.0001`.

`seed`: The `np.uint64` seed for the CUDA random number generator. If not specified, it is internally picked dependent on the PC time. If the user wants the random number sequence to be reproducible, seed must be specified as a constant at the `Mcgpu` instantiation.

`imodel=0`: If there are several alternative models programmed in `model()`, this parameter tells which one should be computed.

`npass=1`: If `model()` requires several sequential calls to compute, this parameter tells how many times it should be called. For example, in `examples/sgra_model_fit` at the first pass the amplitudes and phases model are calculated, and the closure phases are calculated at the second pass from the phases saved from the first pass.

`nadj`: number of sets for calculating acceptance rates. Default is 100.

`nbeta`: Number of temperatures (and number of threads per block). Default is 32.

`nseq`: Number of "sequences" or the 32-temperature processes (and the number of CUDA blocks). Default is 14.

`nburn`: Number of iterations in the burn-in phase. Default is 300.

`niter`: Number of iterations in the main phase. Default is 500.

5.5. Examples of Optimization and Solving Nonlinear Equation Systems

Three examples of using the MCMC-SE CUDA software are given in the directory `examples/`. Two of them are very simple "toys": `search_Rosenbrock` and `solvsys`. A third one, `sgra_model_fit`, is quite comprehensive.

5.5.1. Solving a Nonlinear System of Equations

$$\begin{cases} x^2 + y^2 &= 1 \\ -0.25x + y &= 0.5. \end{cases} \tag{24}$$

Its exact roots are $x_1, y_1 = (-0.966, 0.2585)$ and $x_2, y_2 = (0.7307, 0.6827)$. The solutions are found in `pout` at several locations where `chi2` values are the smallest. See the printout.

```
$ cd $CUDA_MCMC/examples/solvsys/
$ make
$ python2 solvsys.py
or
$ python2 solvsys_api.py
or
$ ipython2 --pylab
[ ]: %run solvsys.py
```

5.5.2. Finding the Rosenbrock Function Minimum

$$z = (1 - x)^2 + 100(y - x^2)^2 \tag{25}$$

The minimum is inside a long, narrow, parabolic shaped flat valley at $x, y = (1, 1)$.

The best solution is in `pout` at the location where `chi2` is minimum. See the printout.

```
$ cd $CUDA_MCMC/examples/search_Rosenbrock/
$ make
$ python2 search_Rosenbrock.py
```

or

```
$ python2 search_Rosenbrock_api.py
```

or

```
$ ipython2 --pylab
[ ]: %run search_Rosenbrock.py
```

5.5.3. Fitting Model to SgrA Black Hole Image*

This example uses the Python class `Mcgpu_Sgra` derived from the class `Mcgpu` described above in Subsection 5.4. Both classes are defined in the module `imgpu`.

There are two models defined in `model()`, 9-parameter and 13-parameter. Of them, the first one is useful, although both can be tested.

The directory `examples/sgra_model_fit/` has several fits files with the SgrA* images and `*_uvdata.txt` files with the results of their simulated observations with the use of MAPS system. The script `model_obsfit.py` runs MCMC-RE to fit the SgrA* image model to the observation data. The `model_obsfit.py` script has minimum two parameters: the model identifier, 9 or 13, and the name of observation data file ending with `_uvdata.txt`. In order to compare the model images with the original fits image in the same color map, use `plot_fits.py`. Below are shown a couple of examples.

```
$ cd $CUDA_MCMC/examples/sgra_model_fit/
$ make
$ ipython2 --pylab
[ ]: %run model_obsfit.py 9 \
[ ]:     000008_1_1_000snd_uvdata.txt
[ ]: %run plot_fits.py 000008_1_1_000.fits

[ ]: %run model_obsfit.py 9 \
[ ]:     000508_1_1_000snd_uvdata.txt
[ ]: %run plot_fits.py 000508_1_1_000.fits
```

One can try the same with the 13-parameter model:

```
[ ]: %run model_obsfit.py 13 \
[ ]:     000508_1_1_000snd_uvdata.txt
```

REFERENCES

Benkevitch, L., Akiyama, K., Lu, R., Doeleman, S., and Fish, V. (2016). Reconstruction of Static Black Hole Images Using Simple Geometric Forms. *arXiv*, 1609:00055.

Benneke, B. and Seager, S. (2012). Atmospheric Retrieval for Super-Earths: Uniquely Constraining the Atmospheric Composition with Transmission Spectroscopy. *ApJ*, 753:100.

Gregory, P. C. (2005). A Bayesian Analysis of Extrasolar Planet Data for HD 73526. *ApJ*, 631:1198–1214.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation of State Calculations by Fast Computing Machines. *J. Chem. Phys.*, 21:1087–1092.

Varghese, A., Ibata, R., and Lewis, G. F. (2011). Stellar streams as probes of dark halo mass and morphology: a Bayesian reconstruction. *MNRAS*, 417:198–215.

This 2-column preprint was prepared with the AAS LaTeX macros v5.2.