Scalable Gaussian Processes, GPyTorch application benchmarking, and Targeted Adaptive Design (TAD) on ThetaGPU

Mathematics and Computer Science Division
About Argonne National Laboratory
Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory’s main facility is outside Chicago, at 9700 South Cass Avenue, Lemont, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY


Reports not in digital format may be purchased by the public from the National Technical Information Service (NTIS):
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312
www.ntis.gov
Phone: (800) 553-NTIS (6847) or (703) 605-6000
Fax: (703) 605-6900
Email: orders@ntis.gov

Reports not in digital format are available to DOE and DOE contractors from the Office of Scientific and Technical Information (OSTI):
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
www.osti.gov
Phone: (865) 576-8401
Fax: (865) 576-5728
Email: reports@osti.gov

Disclaimer
This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.
Scalable Gaussian Processes, GPyTorch application benchmarking, and Targeted Adaptive Design (TAD) on ThetaGPU

prepared by
Yuyang Tian, Carlo Graziani, Marième Ngom
Mathematics and Computer Science Division, Argonne National Laboratory

Nov 30, 2022
Abstract

We aim at showcasing the scalability of a Gaussian process (GP). The naive GP implementation scales cubically with data size, which can be prohibitive, so GP has not heretofore been considered suitable for very large-scale problem settings. We take advantage of GPyTorch, a library for scalable GPs built on top of PyTorch, that incorporates GPU acceleration. With GPyTorch, one can achieve nearly linear scaling with structured kernel interpolation and constant-time predictive covariance computation with Lanczos Variance Estimates (LOVE) while preserving accuracy. We also take advantage of the computational power of ThetaGPU, a supercomputer in the Argonne Leadership Computing Facility. In addition, we implement a scalable, GPU-ready version of targeted adaptive design (TAD), a GP-based data-driven algorithm that efficiently searches the control space of an advanced manufacturing experiment for settings capable
of producing a required design within a specified tolerance, despite the poorly known mapping from control settings to design. We show our benchmarking for GPyTorch and TAD performance on a CPU vs. ThetaGPU and discuss the results and implications.

1 Introduction

Gaussian process (GP) has been popular in recent years, widely used in regression. It is a powerful tool that provides better-formulated mathematical foundations than do neural network (NN) models. However, the naive algorithm of GP has prohibitive computational cost in terms of time-to-solution and memory. Therefore, models involving GPs have been restricted to small datasets and applications. The target of this technical report is large-scale and even high-performance-computing-scale GP modeling and its applications.

To realize this, we utilize GPyTorch [1], a highly efficient and modular implementation of Gaussian processes, with graphics processing unit (GPU) acceleration, built on top of PyTorch. We demonstrate the use of GPyTorch and two of its fast algorithms. Moreover, we take advantage of the Argonne Leadership Computing Facility (ALCF) system ThetaGPU, a supercomputer for high-performance computing (HPC).

We showcase targeted adaptive design (TAD) [2] as an example application of large-scale GP modeling. TAD is a new data-driven algorithm that aims at efficiently and adaptively locating optimal control parameters that would yield a target output design in a multidimensional input and output space. On a high level, TAD is an optimization algorithm that involves GP. TAD is applied in manufacturing and design, so it is important for TAD to be scalable. We port TAD to ThetaGPU and apply our approach to enable TAD to solve much larger-scale problems than have been possible until now.

We present performance benchmarking of our GPyTorch and TAD on a CPU and ThetaGPU and discuss the results and implications.

With GPyTorch and ThetaGPU, we show that GP can be made tens and potentially hundreds of times more efficient. This work establishes the foundation for future users of Argonne’s ALCF systems with models involving GPs.

2 Technical Preliminaries

2.1 Gaussian process

Notation. We use the same set of notations as used in [3, 1]. \( X \) denotes a set of \( n \) training examples in \( d \) dimensions (i.e., \( X \) is an \( n \times d \) matrix); \( y \) denotes the training labels; \( k(\mathbf{x}, \mathbf{x}') \) denotes a kernel function; \( K_{XX} \) denotes the matrix containing all pairs of kernel entries (i.e., \( \left[K_{XX}\right]_{ij} = k(x_i, x_j) \)); \( k_{XX^*} \) denotes kernel values between training samples and a test point \( \mathbf{x}^* \) (i.e., the \( i \)th entry is \( k(x_i, \mathbf{x}^*) \)); and \( \hat{K}_{XX} = K_{XX} + \sigma^2 I \) denotes the perturbed matrix.

A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution. It defines a distribution over the function being modeled (meaning that any collection of function values \( f \) has a joint Gaussian distribution),

\[ f(\mathbf{x}) \sim \mathcal{GP}(\mu(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \]
where \( \mu(x) \) is the mean function and \( k(x, x') \) is the covariance function or kernel defined as the covariance between \( f(x) \) and \( f(x') \). The smoothness and generalization properties of the GP are encoded by the covariance kernel and its hyperparameters. Among the most common kernel choices is the RBF kernel

\[
k(x, x') = \exp\left(-\frac{\|x - x\|}{2\ell^2}\right)
\]

Rasmussen and Williams [3].

GP is commonly used for fitting a function to data (regression) or predicting function values at given points. It makes predictions with the predictive posterior distribution, \( p(f(x^*) \mid X, y) \). Given two test inputs \( x^* \) and \( x'^* \), the predictive mean for \( x^* \) and the predictive covariance between \( x^* \) and \( x'^* \) are given by

\[
\begin{align*}
\mu_{f \mid D}(x^*) &= \mu(x^*) + k_{XX}^\top \hat{K}_{XX}^{-1}y, \\
k_{f \mid D}(x^*, x'^*) &= k_{XX} - k_{XX}^\top \hat{K}_{XX}^{-1}k_{XX'^*}.
\end{align*}
\tag{2.1}
\]

Gaussian processes depend on a number of hyperparameters, including kernel length-scale and noise. These parameters are learned by minimizing the negative log marginal likelihood.

Gaussian process is a powerful tool for regression, but it is computationally heavy. Let \( N \) = data size; then the computation would take \( O(N^3) \) time and \( O(N^2) \) storage. The computational bottleneck of GP is solving \( \hat{K}_{XX}^{-1}y \) for prediction, which involves the Cholesky factorization of the \( N \times N \) matrix \( \hat{K}_{XX} \), and solving \( \log|\hat{K}_{XX}| \) for hyperparameters training. Furthermore, in the case of vector-valued GP with \( E \) = output dimension, the computation would be \( N \times E \) instead of \( N \), making the cost even worse. Hence, the powerful tool GP has generally been restricted to the smallest data size. Some efforts made to enable GP to run on larger datasets, namely, study of approximation algorithms or fast kernel methods such as in [4, 5, 6] to reduce the computational cost, as well as utilization of the power of hardware to accelerate computation.

### 2.2 Harnessing the power of hardware: GPU and ThetaGPU

**GPU.** The GPU is a microprocessor that works specifically for image computing on computers. GPUs are similar to a CPU except that GPUs are designed to perform complex mathematical and geometric calculations that are necessary for graphics rendering. With the development of artificial intelligence, today’s GPUs are no longer limited to 3D graphics processing.

Structurally, while CPUs consist of a few cores optimized for sequential serial processing, GPUs have a massively parallel computing architecture. GPUs are characterized by a large number of cores (up to several thousand) and a large amount of high-speed memory. Furthermore, GPUs have more arithmetic logic units than CPUs do.

GPUs excel at massively parallel computing, so they are well suited for deep learning and machine learning and are commonly used for computation acceleration.

Parallel computations on the GPU are usually manually implemented with GPU programming languages such as CUDA. More modern choices include Numba for Python. However,
an alternative way we take is to utilize the built-in function in PyTorch [7] to realize GPU acceleration, as we will discuss later.

Because of the nature of GPUs, they are suitable for Gaussian processes as well. In this report, we not only take advantage of GPUs to scale up our GP but go even further to utilize the supercomputer ThetaGPU.

**ThetaGPU.** ThetaGPU is an extension of Theta and comprises 24 NVIDIA DGX A100 nodes. Each DGX A100 node comprises eight NVIDIA A100 Tensor Core GPUs and two AMD Rome CPUs that provide 22 nodes with 320 GB of GPU memory and two nodes with 640 GB of GPU memory (8320 GB aggregately) of GPU memory for training artificial intelligence datasets, while also enabling GPU-specific and GPU-enhanced HPC applications for modeling and simulation.

**Remark.** Throughout this report, when we say a code is running on ThetaGPU, we are really referring to running our code on the (A100) GPU of the supercomputer ThetaGPU. Running on CPU refers to running on the CPU of the supercomputer ThetaGPU, which should still be faster than the CPU of a traditional PC. When we say running on ThetaGPU vs. CPU, we are referring to comparing our code run on the GPU vs. the CPU of the supercomputer ThetaGPU.

### 2.3 GPyTorch

In this report, we use GPyTorch to scale up our GPs.

**GPyTorch** is a highly efficient and modular implementation of Gaussian processes, with GPU acceleration, built on top of PyTorch. It implements some approximation algorithms and fast kernel methods for GPs.

We chose GPyTorch because it has several key advantages.

1. It implements some important fast algorithms for both training and prediction (predictive covariance matrix computation). The two we focus on are Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP) [5] that helps with training and Lanczos Variance Estimates (LOVE) [6] that helps with computing predictive covariance matrices and operations that uses it, for example, posterior sampling.

2. It provides easy access to GPU acceleration, because GPyTorch is based on PyTorch. (In addition, GPyTorch inherits other advantages of PyTorch.)

3. PyTorch is one of the most popular deep learning frameworks. It is always part of the first software stack on every HPC machine. Therefore, GPyTorch is relatively easy to install and use on HPC machines.

   Other advantages include the modular design of GPyTorch: the GP Model, Likelihood, and Kernel are implemented as classes and are easy to use and maintain, and one can combine GPs with deep neural networks and more.

   Next we introduce more details on the two fast algorithms KISS-GP and LOVE.
KISS-GP [5] uses the structured kernel interpolation (SKI) framework with local cubic interpolation. It involves fast Fourier transform at its core. KISS-GP (1) is more scalable than inducing point alternatives, (2) enables Kronecker and Toeplitz algebra for substantial additional gains in scalability, without requiring any grid data, and (3) can be used to accelerate kernel learning.

LOVE [6] uses fast matrix vector multiplies with Krylov subspace methods. It helps with computing predictive covariance matrices and operations that use it, such as posterior sampling. Often LOVE is used in conjunction with KISS-GP to complement each other’s strengths. In order to compute a series of predictive covariance matrices at new prediction points based on a GP that was trained on $N_{\text{Train}}$ training points (say, $N_{\text{Train}} = 10,000$), the naive computations would cost a factor that scales as $O(N_{\text{Train}}^2)$ for each newly computed covariance; and even with KISS-GP the scaling is $O(N_{\text{Train}}^1)$ per covariance. With LOVE, the scaling is $O(N_{\text{Train}}^0)$. LOVE computes covariance matrices in constant time and obtains samples in linear time, without sacrificing accuracy, when used in conjunction with KISS-GP.

GPyTorch with the fast algorithms KISS-GP and LOVE and HPC machine acceleration can make GPs tens and hundreds of times more efficient, as we will show later, and potentially scale GPs to millions of points. To the best of our knowledge, this is the first time GPs have been run at these scales.

While GP modeling affords better-formulated mathematical foundations than NN models do, it has not heretofore been considered suitable for big data applications, especially at HPC scales, because of the prohibitive memory and time-to-solution scaling associated with the required linear problem solutions. The GPyTorch approach changes this and enables GP modeling to be run at massive and even HPC scales.

Moreover, GPyTorch is a software platform. Because of its access to the fast algorithms, GPU acceleration, PyTorch, and Python, it is powerful and convenient. GPyTorch will be installed in the modules of ThetaGPU and Polaris and be ready to use in the upcoming Aurora system at Argonne National Laboratory.

We will showcase the scalability potential of GPs by leveraging GPyTorch’s fast methods and ThetaGPU’s computational power.

3 Implementation

3.1 Login and installation

To run GPyTorch on ThetaGPU, we first log in to ThetaGPU: specifically, this means we first log in to Theta, then log in to a service node of ThetaGPU, and then queue for a compute node of ThetaGPU. Next we load the correct modules and install GPyTorch. The installation of GPyTorch on a cluster is different from that on a PC. To run our program on ThetaGPU, we can choose to submit it as jobs, run them directly in the terminal in an interactive session, or run them with a Jupyter Notebook (which we chose for the purpose of this report). Here is our complete guide to logging in to ThetaGPU, loading environments, running jobs, installing GPyTorch, and running with a Jupyter Notebook.
The installation of GPyTorch and related modules on ThetaGPU is a prototype of installing them on other HPC machines, such as Polaris and the upcoming Aurora.

Next, we briefly demonstrate the use of GPyTorch and the two fast algorithms KISS-GP and LOVE.

3.2 Use of GPyTorch

3.2.1 Basic usage

```python
# We will use the simplest form of GP model, exact inference
class ExactGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(ExactGPModel, self).__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)

# initialize likelihood and model
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = ExactGPModel(train_x, train_y, likelihood)
```

Figure 1: Defining a GP Model, an excerpt in the GPyTorch documentation

```python
f_prets = model(test_x)
y_preds = likelihood(model(test_x))

f_mean = f_prets.mean
f_var = f_prets.variance
f_covar = f_prets.covariance_matrix
```

Figure 2: Using the trained model to make predictions, an excerpt in the GPyTorch documentation

The Model, Likelihood, and Kernel are implemented as classes in GPyTorch. To create one’s own GP model, the user writes a GP model class and specifies the mean module and kernel module. In most cases, the user defines only the kernel module, as indicated in the red box in Figure 1, and uses `gpytorch.means.ConstantMean()` for the mean module.

Then, one basically uses the standard PyTorch training code to train the model.

Finally, one makes predictions with the trained GP model as in Figure 2.

Remark (Model versus Likelihood) Many beginners of GP and GPyTorch are often confused by the difference between `model(test_x)` and `likelihood(model(test_x))`. The
former gives prediction for the latent function and does not include the noise, while the latter
gives prediction for the output values including the noise. That is, let \( X, y \) be the training
data, and \( x^*, y^* \) be a test point. Then \( \text{likelihood}(\text{model}(test_x)) \) returns the posterior
predictive distribution

\[
p(y^* | x^*, X, y)
\]

while \( \text{model}(test_x) \) returns the predictive posterior distribution

\[
p(f^* | x^*, X, y)
\]

### 3.2.2 KISS-GP

```python
class GPRegressionModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(GPRegressionModel, self).__init__(train_x, train_y, likelihood)

        # SKI requires a grid size hyperparameter. This util can help with that. Here we use
        grid_size = gpytorch.utils.grid.choose_grid_size(train_x, 1.0)

        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(gpytorch.kernels.GridInterpolationKernel()
          gpytorch.kernels.RBFKernel(), grid_size=grid_size, num_dims=1)

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)

likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = GPRegressionModel(train_x, train_y, likelihood)
```

Figure 3: Defining a KISS-GP Model, an excerpt in the GPyTorch documentation

To use the KISS-GP Model, one wraps the Kernel with the \texttt{GridInterpolationKernel} (SKI).

We note that besides \texttt{num_dims} (the dimension of one’s (preprocessed) input space), the
other important parameter of \texttt{GridInterpolationKernel} is \texttt{grid_size}. A bigger grid size
means the result would be more accurate but the computation would be slower. One may
use \texttt{grid_size = gpytorch.utils.grid.choose_grid_size(train_x, )} to determine the
optimal \texttt{grid_size}.  

7
3.2.3 LOVE

![Code Snippet]

Figure 4: Using LOVE for predictive covariance matrix computation, an excerpt in the GPyTorch documentation

To use LOVE, one simply uses the context manager `gpytorch.settings.fast_pred_var()` when computing the predictive distribution, as indicated in Figure 4

3.2.4 GPU acceleration with GPyTorch

Because GPyTorch is an extension of PyTorch, we can use PyTorch-like code to achieve GPU acceleration, which is a marvelous feature.

One moves all model, likelihood, and tensors involved onto the GPU by

```python
use_cuda = torch.cuda.is_available()
... if use_cuda:
    something.cuda(),
```

where `something` often includes model, likelihood, train_x, train_y, test_x, test_y.

One can use `something.cpu()` later if needed to move the data back to the CPU.

4 GPyTorch application benchmarking on ThetaGPU

We conducted numerical experiments for performance and accuracy of KISS-GP/LOVE vs. Exact GP on both CPU and ThetaGPU. We show that the KISS-GP Model can significantly reduce the time for training and, when combined with LOVE, reduce the time for computing predictive distributions/covariance matrix. This information can be especially useful in settings such as small-scale Bayesian optimization, where predictions need to be made at enormous numbers of candidate points.

4.1 Methodology for benchmarking

The dataset we use to train our KISS-GP Model is the elevators UCI dataset, as Gardner et al. [1] used. The data is split with the first 40% as training and the last 60% as testing. We then compare the time required for training and computing predictive distributions and the error with each model and on CPU vs. ThetaGPU.

As mentioned in Section 3.2.2, one would need to wrap the Kernel with the `GridInterpolationKernel`. This model uses a `GridInterpolationKernel` with a deep RBF base kernel. The forward
method passes the input data through the neural network feature extractor, scales the resulting features to be between 0 and 1, and then calls the kernel.

The deep RBF kernel (DKL) uses a neural network as an initial feature extractor. In this case, we use a fully connected network with the architecture \( d \rightarrow 1000 \rightarrow 500 \rightarrow 50 \rightarrow 2 \), as described in the original paper by Wilson et al. [8]. All of the code below uses standard PyTorch implementations of neural network layers.

Furthermore, as also discussed in Section 3.2.2, the \texttt{grid\_size} parameter would influence the accuracy and speed of our KISS-GP. For this numerical experiment, we fix the \texttt{grid\_size} to 100, because changing the grid size would affect the speed and accuracy, which is bad for measuring time. But, as we will see, this probably makes the accuracy for covariance matrix to be nonoptimal and fluctuate.

To conduct the experiment, we gradually increase the data size, from using only the first 200 data points in the dataset to using all 16,599 data points. More specifically, we use a for loop to loop around these sizes. In each iteration, we initialize the Exact GP model, measure the training time, and measure the time for computing predictive distributions (without LOVE). Then, we initialize the KISS-GP Model, measure the same things except this time we turn on LOVE for covariance matrix computation.

More details and specific implementations of this experiment can be found in this notebook.
4.2 Visualization of numerical results

Figure 5: Performance of KISS-GP with LOVE vs. Exact GP on CPU and GPU. The first row compares the training time on CPU and ThetaGPU with the KISS-GP model and Exact GP model; the second row compares the time for computing predictive distributions (both mean and covariance matrix) on CPU and ThetaGPU with the two models, for each input data size. The first column is the times for both models on CPU, and the second column is the times on ThetaGPU. For each plot, the x-axis is the input data size, and the y-axis is the running time. This figure shows how the running time rises with increasing input size for the two models, on the CPU and ThetaGPU.
4.3 Discussion of numerical results

We first focus on the first row: training. The plot on the left for training on the CPU shows that KISS-GP is giving a significant improvement in training time, up to 7x faster for data size 16599. The Exact GP training time grows exponentially, which is costly. The plot on the right for training on the GPU can be surprising at first, because KISS-GP is taking longer than Exact GP for training. The reason is that the data size is still too small for the GPU to fully show the power of KISS-GP. More important, this might imply that some parts of KISS-GP of GPytorch are not optimized on the GPU and take a long time. Moreover, the Exact GP training time rises exponentially, while the KISS-GP training time seems to saturate and stabilize after the data size of 2000, which might imply that we will see the same pattern as on the CPU should we switch to a much larger data size. We remark that the GPU significantly improves the speed for training, making it up to 200x times faster for Exact GP and 17x times faster for KISS-GP, and the acceleration could be much more prominent on a larger dataset, especially for KISS-GP. Numerical experiments with a much larger dataset on the GPU are needed for future study.

Next, we move our discussion to the second row: predictive distributions computation. As we see for both CPU and GPU, the time for computing exact predictive distributions scales up as $O(N^3)$, consistent with the traditional GP time complexity, while KISS-GP with LOVE shows an almost constant time complexity. (Only the predictive covariance matrix computation but not the predictive mean is accelerated by LOVE.) The GPU plot on the right once again might indicate that certain parts of KISS-GP in GPyTorch is not optimized on the GPU. Moreover, as with the case of training, the GPU significantly accelerates the predictive distributions computation, by up to 50x for Exact GP and 20x for KISS-GP for
We see that the mean absolute error (MAE) between the predictive covariance matrix of KISS-GP with LOVE and that of Exact GP is small yet fluctuating. There could be two reasons behind this: (1) poor conditioning of the matrix and numerical instability (such as there are negative entries) and (2) the fact that we are not using the optimal grid size but are fixing it instead, as we previously mentioned. The average MAE is around $6 \times 10^{-3}$. This is not big, but we also remark that this is a magnitude error while one would generally want a relative error (e.g., error on the order 0.01) when studying factors like variance. We encountered some trouble when computing the relative error again because of poor conditioning of the matrix and numerical instability. The issue is still under investigation, and we expect to solve the problem. According to Wilson and Nickisch [5], the method, in terms of scaled mean absolute error, is accurate.

In this benchmarking, we have demonstrated that both KISS-GP with LOVE and ThetaGPU give us significant acceleration without sacrificing much accuracy, and they still have a lot of potential. The accelerations will be much more significant on larger datasets, since the running time of KISS-GP is almost not rising in this dataset. Benchmarking with much larger datasets is needed and will be conducted in the future.

5 Applications of GPyTorch: Targeted Adaptive Design

Besides scaling up GPs with GPyTorch and ThetaGPU, our work included how to apply those to scale up targeted adaptive design Graziani and Ngom [2]. TAD constitutes an example application of scalable GP and GPyTorch.

TAD is a new data-driven algorithm that aims at efficiently and adaptively locating optimal control parameters that would yield a target output design in a multidimensional input and output space. An unknown mapping $f$ between D-dimensional input space and E-dimensional output space, called the objective function, is accessible only through noisy measurements. On a high level, the TAD algorithm first fits $f$ with the GP (which constitutes the bottleneck of computation for reasons mentioned in the Introduction, 2.1); then it optimizes an acquisition function $L$ to decide where to sample next, acquires those points, and updates the GP model; it repeats until the stopping criterion is met. TAD is used in modern manufacturing and design processes, which often require searches of relatively high-dimensional process control parameter spaces for settings that result in optimal structure, property, and performance parameters.

5.1 Moving TAD to ThetaGPU

The code of Graziani and Ngom [2] is intended for TAD to be applied in manufacturing and design. For this purpose, it is important for TAD to be scalable. Until last summer, however, Graziani and Ngom [2] had used TAD only in two dimensions with small data size, and the code had been running only on the CPU of a PC. Therefore, our first task for TAD was to port it to ThetaGPU.

We did so by moving all model, likelihood, and tensors involved onto the GPU by the following.
use_cuda = torch.cuda.is_available()
...
if use_cuda:
    something.cuda()

Later in each iteration of the algorithm, we need to port certain operations back to the CPU by something.cpu(), because some operations, such as converting tensors to numpy, cannot be done on the GPU. This means that we have to port the data back and forth between the CPU and GPU in each iteration of the algorithm.

The difficulty of this task lies in determining the optimal positions to put these commands and transfer the data. The number of transfers must be minimized because transferring the data is costly. By empirical evidence we have seen that optimizing in this regard can greatly improve the running time of TAD on ThetaGPU. We also optimize the structure of the original code in [2] to further improve the number of data transfers needed.

After TAD was able to run on ThetaGPU, we did some benchmarking of TAD’s performance on CPU vs. ThetaGPU.

5.2 TAD benchmarking on ThetaGPU

In this section we present our methodology for testing the running time of various components of TAD on CPU vs. ThetaGPU, and we discuss the results and implications.

5.2.1 Methodology for benchmarking

We use the same objective function as in [2]:

\[
V_1(x_1, x_2) = 3(1-x_1)^2e^{-x_1^2-(x_2+1)^2} - 10(x_1/5-x_1^3-x_2^5)e^{-x_1^2-x_2^2} - 3e^{-(x_1+2)^2-x_2^2} + 0.5(2x_1+x_2)
\]

\[
V_2(x_1, x_2) = 3(1+x_2)^2e^{-x_2^2-(x_1+1)^2} - 10(-x_2/5+x_2^3+x_1^5)e^{-x_1^2-x_2^2} - 3e^{-(2-x_2)^2-x_1^2} + 0.5(2x_1+x_2),
\]

where \((x_1, x_2) \in [-3, 3]^2\).

We first run TAD on the CPU and in every iteration record the time cost of hyperparameter optimization (\texttt{hyper_opti(.)}), TAD design optimization (\texttt{conduct_design_opti(.)}), and the whole iteration. Then we “try our best” to rerun the same thing on the GPU. Our hope is to make these two runs identical in every iteration, but it is not possible because TAD is, at its core, a probabilistic method. Therefore, we try to make the two runs as similar as possible for time testing purposes by using the same initial solution \(x_0\), initial samples \(\text{loc\_sample0}\), and training data \(x_{\text{train}}, y_{\text{train}}\).

Theoretically, the larger the data size, the more prominent the GPU acceleration is. TAD starts with four points and adds three points to its training data in each iteration. Therefore, for this experiment, we want TAD to go through a lot of iterations, instead of converging with only 10 to 20 iterations. The important trick we use here is to make the target out of the range of the objective function, so that no solution exists. In this setting, TAD would converge with typically 50 to 70 iterations and fail to find a solution.
5.2.2 Visualization of numerical results

Figure 7: Running time per iteration of different parts of TAD on CPU vs. ThetaGPU. This figure shows the running time for three different parts of TAD in each iteration of the algorithm. From left to right, the first plot shows the time for hyperparameter optimization, the second for TAD design optimization, and the third for the whole iteration (hyperparameter optimization, TAD design optimization, and other operations including checking the model and updating model complexity), on CPU vs. ThetaGPU. We note that the time is not cumulative but for each iteration. It has an upward trend because the algorithm adds three points to its training data in each iteration.

5.2.3 Discussion of numerical results

We see from the figure that ThetaGPU is giving a significant acceleration compared with that of the CPU. If we compare the 52nd iteration (the last iteration on the CPU) so that the CPU and GPU are handling the same number of data points, we see that on the GPU TAD is more than 5 times faster for hyperparameter optimization, more than 3 times faster for TAD design optimization, and more than 3 times faster for the whole iteration.

Moreover, we notice that the TAD design optimization part (which corresponds to the middle plot) contributes to the majority (more than 85%) of the running time in each iteration. Hence, optimizing the code and/or algorithm for this part is the most important. If the data size becomes much larger, however, we anticipate that “the other parts” (the parts after hyperparameter optimization and TAD design optimization) will take a decent amount of time because of their extensive use of expensive operations (e.g., torch.matmul, torch.det, and torch.log), although they take a trivial amount of time in this experiment. Therefore, putting such operations in “the other parts” in parallel on different GPUs is also a major concern when the data size gets very large.

In addition, in all (especially the second and the third) plots, the GPU running time is almost constant (with some fluctuations since TAD is a probabilistic method). This probably signals that the amount of data in this experiment is still too small for ThetaGPU
to demonstrate its advantage. The same experiment with much larger data size and higher dimensions of input and output space can be helpful and will be conducted in the future.

5.2.4 Benchmarking conclusions and future directions on improving TAD performance on ThetaGPU

Currently when TAD is running, the GPU usage is 22%, which is not high. In this scenario, simply changing to multiple GPUs without optimizing on a more atomic level will not help in speed (unless we change to much higher dimensions or larger data size).

We suggest some future directions for improving the performance of TAD on GPU:

1. Enable multiple GPUs. This will be helpful if in the future TAD switches to much higher dimensions or makes use of much larger data sizes. To do so, one should first try to wrap the Kernel class with MultiDeviceKernel provided by GPytorch. Please visit the model definition cell of our ConvSuccess_homoskedatic_SingleGPU.ipynb and GPyTorch example use of MultiDeviceKernel for more details.

2. Try putting specific operations of TAD in parallel. As mentioned in the Discussions earlier, operations such as `torch.matmul`, `torch.det`, and `torch.log` can be costly when the data size is large. Therefore, putting such operations after hyperparameter optimization and TAD design optimization of the TAD algorithm in parallel on different GPUs is also a major concern when the data size gets very large. Many such operations are independent and can be computed in parallel. PyTorch’s built-in GPU-related functions can probably be used for this purpose.

3. Try doing GPU code optimization on a more atomic level. Probably the easiest way is to use a library called numba, but this is still demanding and may not be worthwhile depending on the situation. One should definitely consider investigating the preceding two directions first.

4. Use KISS-GP Model and enable LOVE.

6 Conclusions

Gaussian process is a powerful tool in machine learning today, commonly used in regression. GP gives us better-formulated mathematics and transparent interpretability than neural networks do. However, GP has a serious shortcoming—it is too computationally heavy, in both training and prediction, which limits GP to very small datasets. Prior to this work, no one at Argonne had been running GPs on high-performance computers at massive scale.

The key to scalable GP, we found, was taking advantage of GPyTorch [1] and the Argonne ALCF system ThetaGPU. GPyTorch is a highly efficient and modular implementation of Gaussian processes that extends PyTorch[7], with GPU acceleration. The advantages of GPyTorch include that it implements the fast algorithms KISS-GP for training and LOVE for prediction, it grants convenient GPU acceleration by inheriting PyTorch, and it is relatively easy to install and use on HPC machines.
When GPyTorch is combined with the computational power of ThetaGPU, it can make GPs tens—as we have demonstrated in our benchmarking—and potentially hundreds of times more efficient and can scale GPs to millions of points. These are scales that people have hardly seen, and this approach resolves the problem of computational cost in using GPs, enabling their use with more power and potential.

These are the reasons that make Scalable GP, GPyTorch, and our work exciting.

We demonstrated the use of GPyTorch and its two fast algorithms. We also presented our benchmarking results of KISS-GP with LOVE vs. Exact GP and on CPU vs. ThetaGPU. The results showed that GPyTorch together with ThetaGPU can improve the running time by tens of times; and, as noted in our discussions, should be able to achieve more with much larger datasets. Furthermore, we applied our scalable GP methodology and GPyTorch to targeted adaptive design: we ported the TAD code to ThetaGPU and benchmarked TAD’s performance on ThetaGPU vs. CPU. Results indicate that TAD runs multiple times faster on ThetaGPU than on the CPU, and we showed that one can expect the improvement to be greater with larger datasets and higher dimensions. Both GPyTorch and TAD benchmarking on ThetaGPU with much larger datasets and/or higher dimensions will be conducted in the future.

This work establishes the foundation for future users of GPyTorch and Argonne ALCF systems with models and algorithms involving GPs. At the time that this report was written, there are several ongoing efforts involving scalable GPs based on this work. The author of this report will also be one of the presenters of the scheduled tutorial on running GPyTorch on High Performance Computing System for Scalable GPs to invited scientists at Argonne.

Acknowledgments  This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

I thank my advisors Marieme Ngom and Carlo Graziani for their guidance over the summer.

References


Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Lemont, IL 60439

www.anl.gov