GPU usage within Spatial Models

Nehemias Ulloa

Iowa State University

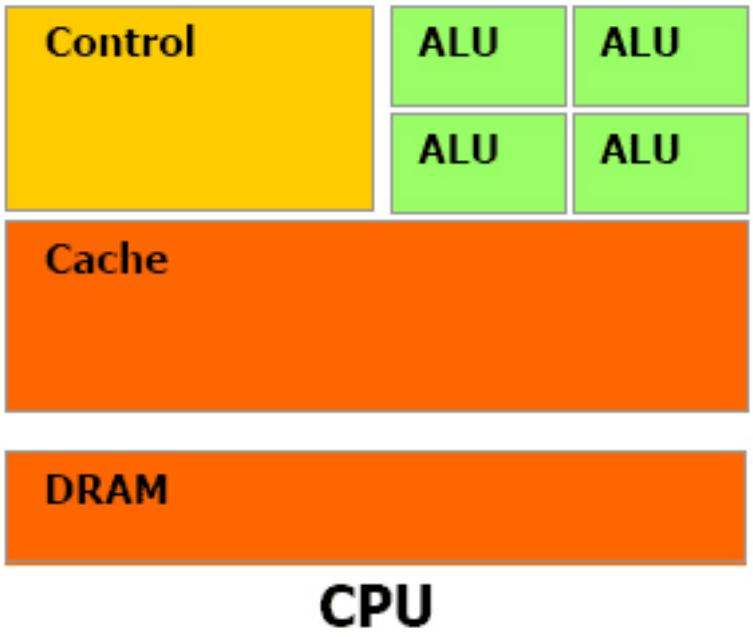
December 1, 2016

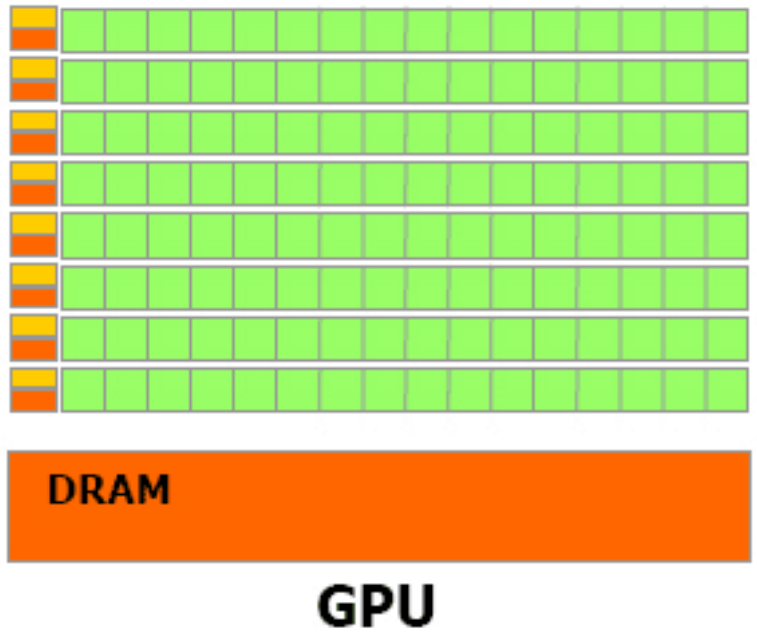
Overview

1. GPU
2. Examples
   - Spatial Models with Block Composite Likelihood
   - Approximate Gaussian Process Regression
   - Using own GPU
3. Discussion

- GPU: Graphics Processing Unit
- Chips/Processors similar to a CPU but they specialize in graphics
- First appeared in arcade games
- Motivation is largely still the same

- What's the difference from a CPU?
- Difference is in structure
- GPU has a structure that is made for parrellelization
- "GPUs are optimized for taking huge batches of data and performing the same operation over and over very quickly, unlike PC microprocessors, which tend to skip all over the place" - Nathan Brookwood



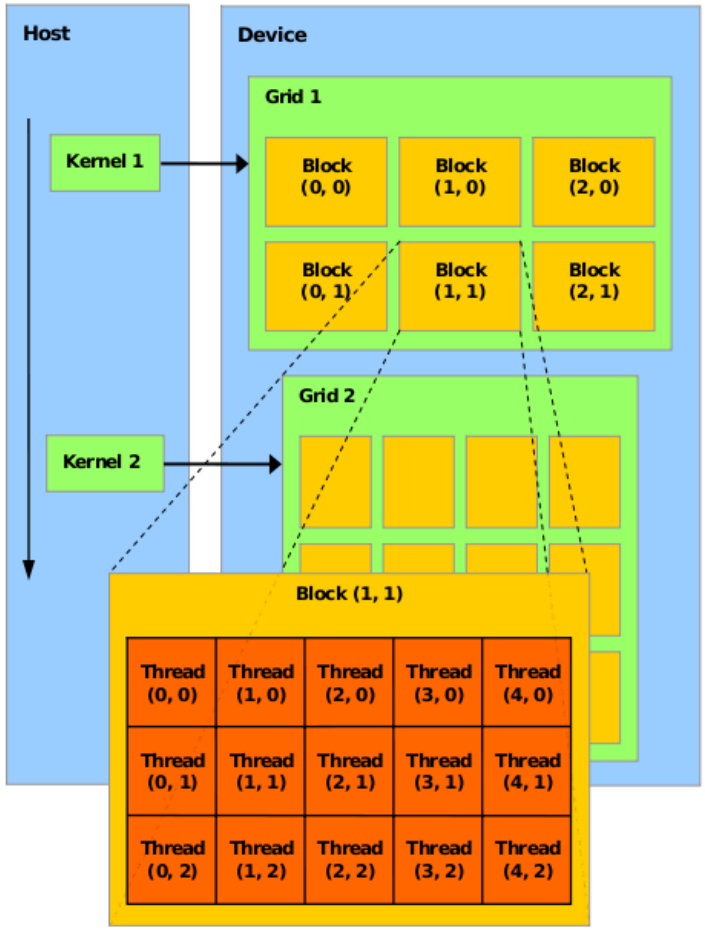


Source: NVIDIA

- Most common way to use a GPU is by using NVIDIA's `CUDA` programming model
- Send work to the GPU using a *kernal* function
- To write a good kernal function, need to understand the architecture of the GPU.

Here are the big components of the architecture:

- Threads - Smallest unit that executes a command
- Blocks - Group of 1024 threads per block
- Kernal Grid - Group of 65,533 Blocks where a kernal fn is invoked
- Warp - Groups of threads in a block that simmultaneously execute a command
- Streaming Multiprocessor(SM) - Wraps for the same block are same SM aka actually runs the CUDA kernals



Source:NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 1.1

We would like to run as many blocks as possible on the SM but that depends on two constraints:

- Amount of memory - Different for each card
- Number of registers required by each thread - We have some control

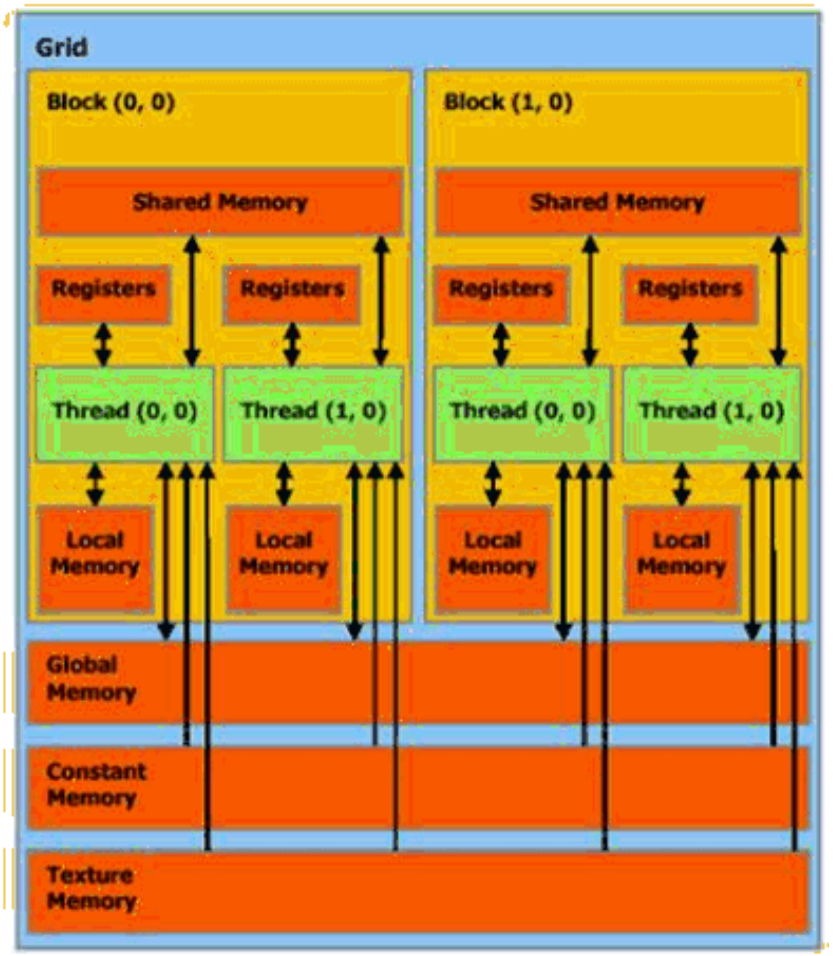
Ultimately the number of SMs and blocks per SM is defined by the hardware of the GPU. In the second paper's example their card had 16 SMs and allowed multiple blocks per SM.

Memory types and locations:

Type	Access	Size
Thread/Local	Thread	Small
Shared	Block	48 KB
Global	Everyone	5 GB

For "good" GPU usage we want to:

- Minimize need to access the Global Memory
- If possible 'live' in the Thread/Local Memory



Source:NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 1.1

Geostatistical model:

$$Y(\mathbf{s}) = \mathbf{x}^t(\mathbf{s})\boldsymbol{\beta} + w(\mathbf{s}) + \epsilon(\mathbf{s}) \tag{1}$$

where

- $\mathbf{s} \in \mathbf{D} \subseteq \Re^d$ are spatial locations in 2 or 3 dim space
- $Y(\mathbf{s})$ Guassian response variable
- $\mathbf{x}^t(\mathbf{s})$ is a vector of explanatory variables
- $\epsilon(\mathbf{s}) \overset{ind}{\sim} N(0, \tau^2)$
- $w(\mathbf{s})$ provides the structural dependence i.e. the covariance structure

Consider the data at n loacations then we can consider our model like this:

$$\mathbf{Y} \sim N(\mathbf{X}\beta, \Sigma)$$
$$\text{where } \Sigma = \Sigma(\theta) = \mathbf{C} + \tau^2 \mathbf{I}_n$$
$$\text{and } C(i,j) = cov(w(\mathbf{s}_i), w(\mathbf{s}_j))$$

Then we can consider the log-likelihood as:

$$\ell(\mathbf{Y}; \beta, \theta) = -\frac{1}{2} log|\Sigma| - \frac{1}{2}(\mathbf{Y} - \mathbf{X}\beta)^t \Sigma^{-1}(\mathbf{Y} - \mathbf{X}\beta) \qquad (2)$$

- As usual, the most computationally intensive parts will deal with computing $|\Sigma|$ and Σ^{-1}. One way to handle this is a composite likelihood based on pairwise data differences
- This is a natural compromise for geostatistical models; the blocks allow us a good tradeoff between computational and statistical efficiency
- The basic idea is to split our sample space ($\mathbf{D}$) into M blocks where $\bigcup_k \mathbf{D}_k = \mathbf{D}$ and $\mathbf{D}_k \bigcap \mathbf{D}_l = \emptyset$

We can get the likelihood to look like:

$$\ell(\mathbf{Y};\beta,\theta) = \sum_{k=1}^{M-1}\sum_{l>k}\ell(\mathbf{Y}_{kl};\beta,\theta)$$
$$= \sum_{k=1}^{M-1}\sum_{l>k}\Big[-\frac{1}{2}\log|\Sigma_{kl}| - \frac{1}{2}(\mathbf{Y_{kl}} - \mathbf{X_{kl}}\beta)^t\Sigma_{kl}^{-1}(\mathbf{Y_{kl}} - \mathbf{X_{kl}}\beta)\Big] \tag{3}$$

where

$$\mathbf{Y}_k = \{Y(\mathbf{s}_i) : \mathbf{s}_i \in \mathbf{D}_k\}$$
$$\mathbf{Y}_{kl} = (\mathbf{Y}_k^t, \mathbf{Y}_l^t)^t$$
$$\mathbf{X}_{kl} = (\mathbf{X}_k^t, \mathbf{X}_l^t)^t$$
$$\Sigma_{kl} = \begin{bmatrix}\Sigma_{kl}(1,1) & \Sigma_{kl}(1,2)\\ \Sigma_{kl}(2,1) & \Sigma_{kl}(2,2)\end{bmatrix}$$

$\Sigma_{kl}(1,1)$ is the $n_k \times n_k$ covariance matrix of $\mathbf{Y}_k$

In order for this to work well we need to choose blocks carefully. The goal is to minimize the correlation between observations not in a block-pair and maximize the number of blocks simultaneously.

Here are some ideas:

- Block widths equal to the effective spatial range
- If interested in computation, use equally numbered blocks
- Block according to spatial dependence

Computationally, the Block Composite Likelihood complexity is of $O(n)$.

The limit on memory is not an issue since the CL, score, and Hessian calculations are summations over independent calculations for each pair. Since these are independent calculations, they can be parrallelized.

Here are some sample results.

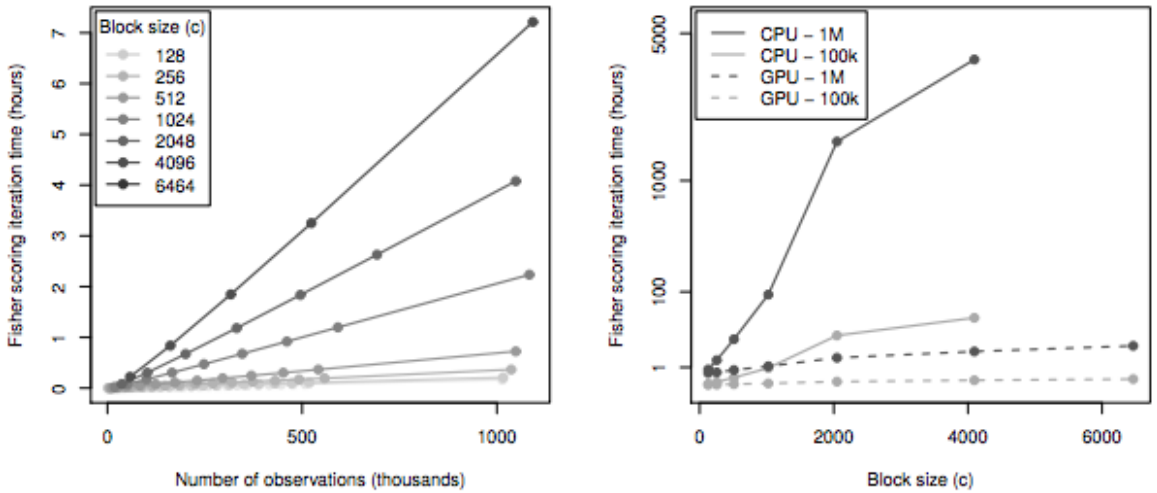


Figure : Computation times for a single Fisher-scoring iteration on NVIDIA GPU for various data sizes per block. Computation times on a CPU and GPU for a single Fisher-scoring iteration for various block sizes at two different sample sizes.

Def: Gaussian process a group of random variables where any subset of them have a joint Gaussian distribution

- Extend multivariate Gaussian distributions to infinite dimensionality e.g. $\mathbf{y} = \{y_1, \ldots, y_n\}$ as a single point from a n-dim Multivariate Normal
- Fully defined by a mean function and a covariance function
- GP Regression is often used in computer emulation i.e. a distribution $Y(x)|D_N$ for new x given simulated data D_N, $p(y(x)|D_N, K_\theta)$

We typically assume that the mean function is zero, but there are multiple options for the covariance function. Some popular options are:

- Squared exponential
- Matern
- Noise (i.e. White noise - flat spectrum)

The papers used an isotropic Gaussian correlation structure:

$$K_{\theta,\eta}(x, x') = exp\{- \parallel x - x' \parallel^2 /\theta\}$$

- Problem is that inverse and determinant computations of the correlation matrix $K(,)$ are $O(N^3)$
- One of the most common ways to handle this is to use sparsity
- Gramacy and Apley(2014) sparsity scheme for accurate and fast predictions

Here is the intuition behind the scheme:

- Focus on the prediction problem at some x
- Data far away from prediction point won't have much influence
- Use a subdesign $D_n(x) \equiv D_n(X_n(x))$
- Most common option is Nearest Neighbors(NN), but it has issues
  - Stein, Chi, Welty (2004) showed that using a few points away from the point of interest improves prediction and parameter estimation
- A criteria that cycles through possible $D_n(x)$ until optimal is found
- This is done with no extra computational cost compared to NN

We get the criterion(ALC) by minimizing the Emperical Bayes mean-square prediction error (MSPE):

$$J(x_{j+1}, x) = E\{[Y(x) - \mu_{j+1}(x|D_{j+1}, \hat{\theta}_{j+1})]^2 | D_j(x)\} \quad (4)$$

$$\mu_{j+1}(x|D_{j+1}, \hat{\theta}_{j+1}) = k^T(x)K^{-1}Y \quad (5)$$

where $j < n$, $\hat{\theta}_{j+1}$ is the est of θ on D_{j+1}, and $k^T(x)$ is the N-vector whose i_{th} component is $K_\theta(x, x_i)$, $K = K_\theta(x_i, x_j)$

$$J(x_{j+1}, x) \approx V_j(x|x_{j+1}, \hat{\theta}_j) + \left(\frac{\delta \mu_j(x;\theta)}{\delta \theta}\Big|_{\theta=\hat{\theta}_j} \right)^2 / \mathcal{G}_{j+1}(\hat{\theta}_j) \qquad (6)$$

This term is the expected future inverse Fisher's information:

$$\mathcal{G}_{j+1}(\hat{\theta}_j) = -\ell''(Y_j;\theta) + E\{-\frac{\partial^2 \ell_j(y_{j+1};\theta)}{\partial \theta^2} \mid D_j;\theta\}$$

This term is the estimate of the predictive variance at X after X_{j+1} is added into the design:

$$V_j(x|x_{j+1};\theta) = \frac{(j+1)\psi_j}{j(j-1)} \upsilon_{j+1}(x_j;\theta)$$

$$\text{where } \upsilon_{j+1}(x_j;\theta) = \left[K_{j+1}(x,x) - k_{j+1}^T(x) K_{j+1}^{-1} k_{j+1}^T(x)\right]$$

$$\psi = Y^T K^{-1} Y$$

The first part of the Eq 6 estimates the predictive variance at x after we include x_{j+1} in the design and the second part estimates the rate of change in the predictive mean at x weighted by the expectation of the future inverse information after x_{j+1} and y_{j+1} are added to the design.

Actually, minimizing the first term of Eq 6 is sufficient to minimize the whole equation. This is because:

- Found the contribution of second term to be small
- Including it led to smaller designs
- Computationally, a lot quicker since no derivatives

So we just need to maximize:

$$\upsilon_j(x;\theta) - \upsilon_{j+1}(x;\theta) \tag{7}$$

Putting it all together, we get this algorithm:

1. Choose starting global $\theta_x = \theta_0$ for all x
2. Calculate local design $X_n(x, \theta_x)$ based on ALC, for each x (independently)
   (a) Choose NN design, $X_{n_0}(x)$ of size n_0
   (b) For $j = n_0, \ldots, n-1$, set

$$x_{j+1} = \arg \max_{x_{j+1} \in X_N \setminus X_j(x)} \upsilon_j(x; \theta) - \upsilon_{j+1}(x; \theta),$$

$$\text{then update } D_{j+1}(x, \theta_x) = D_j(x, \theta_x) \cup (x_{j+1}, y(x_{j+1}))$$

3. Independently, calculate $\hat{\theta}_n(x) | D_n(x, \theta_x)$. Set $\theta_x = \hat{\theta}_n(x)$
4. Repeat steps 2-3
5. Output predictions $Y(x) | D_n(x, \theta_x)$

From the algorithm outline, we can see that step 2.b. is the hardest to compute. This is where using the GPU comes in!

Note that each candidate's (x_{j+1}) calculations can computed independent of each other. This allows us to give each block it's own candidate. From there we give j threads their own sequence of calculations.

Here are some simulation results.

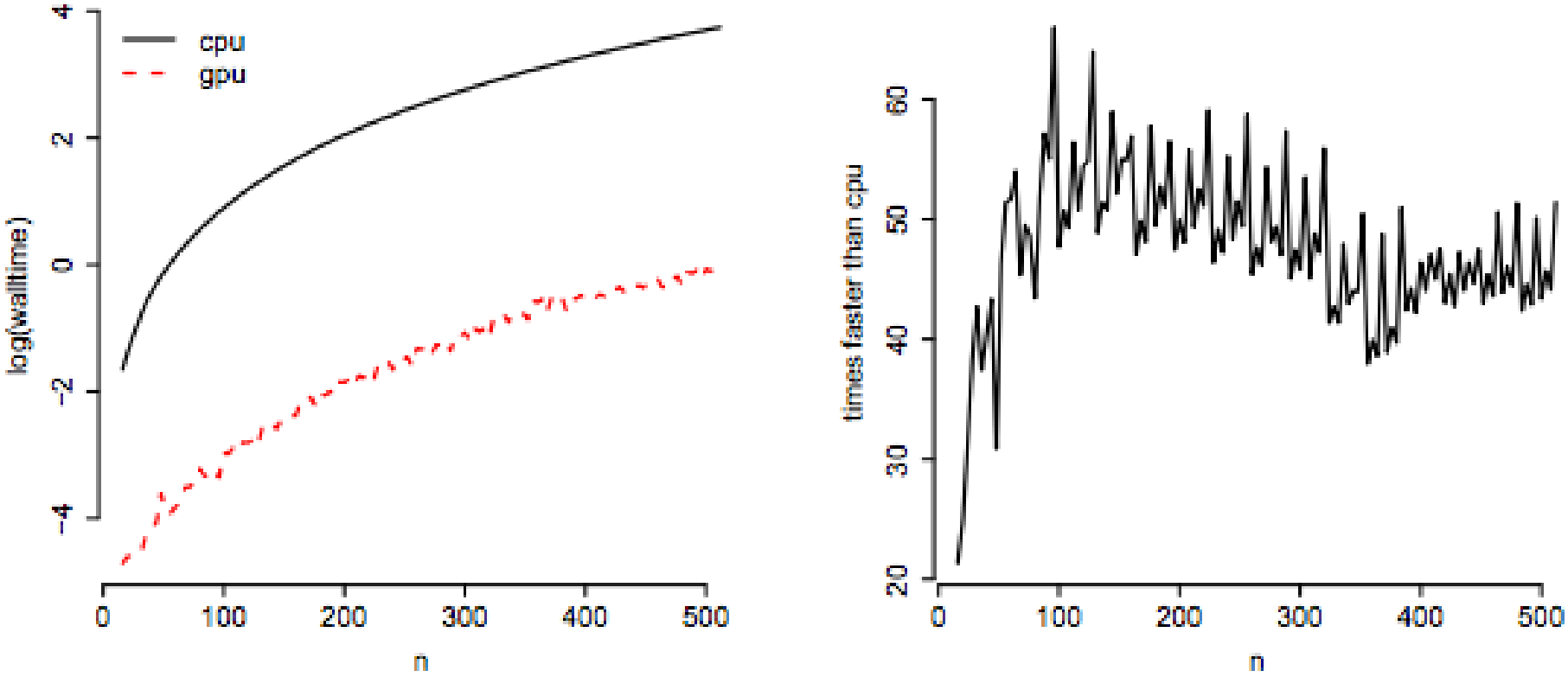


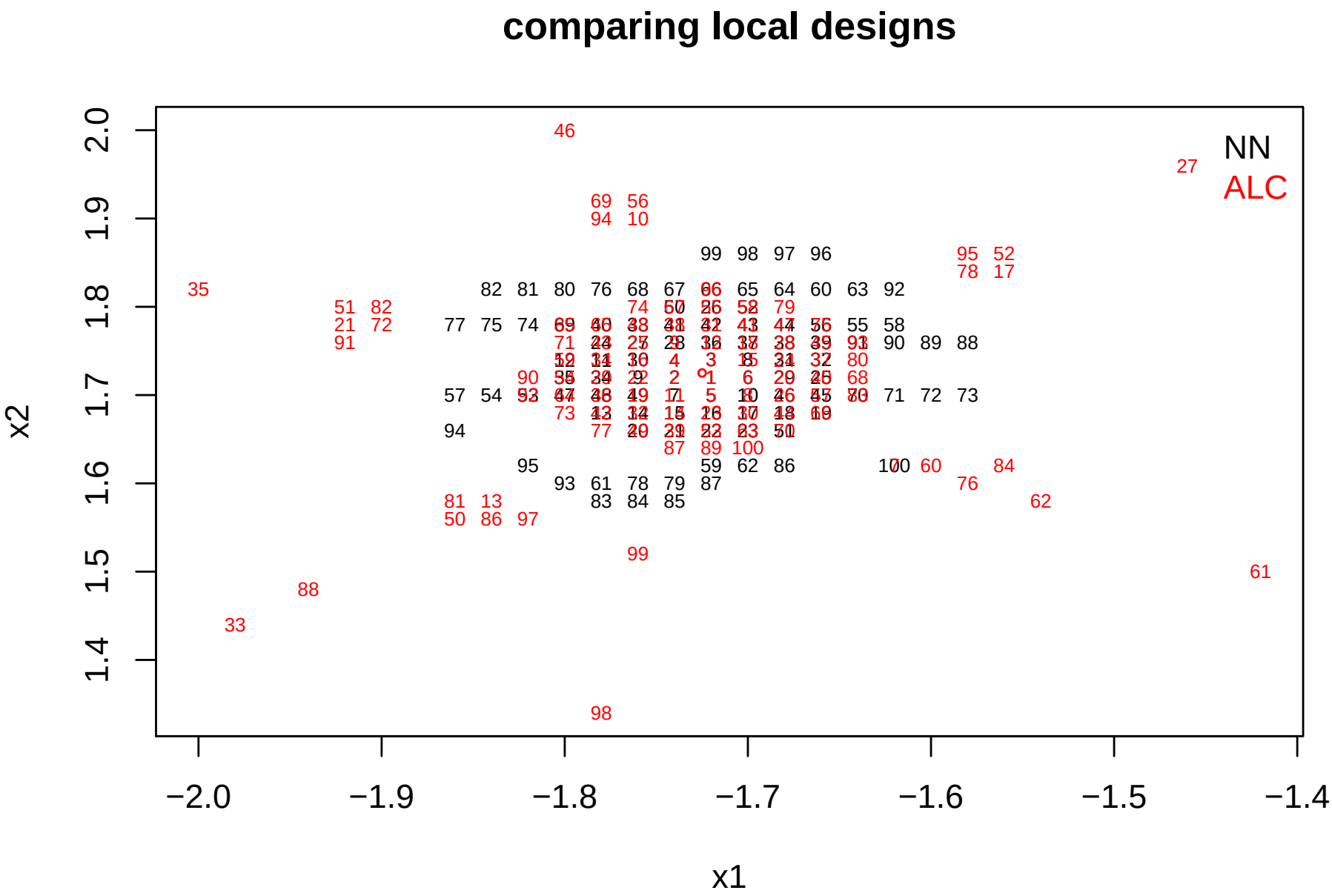
Figure : This graphics compares CPU-only & GPU-only times for ALC calculations at $n_{cand} = 60000$ ($\sim$ max number of blocks) candidate locations while changing n (sample size).

The writers of the last paper created a R package laGP in order to easily implement the CUDA, C, and R subroutines.

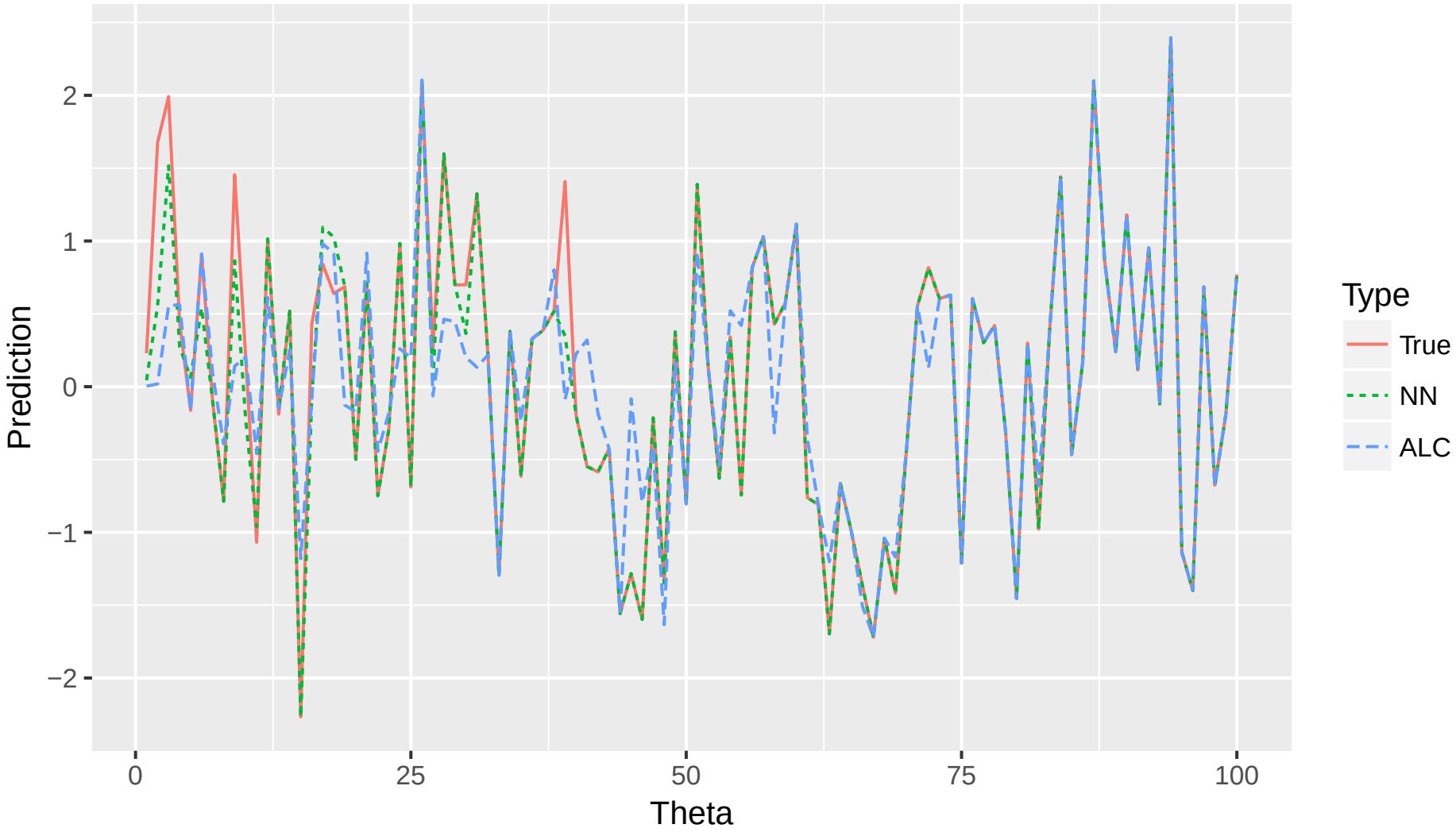
```
laGP(Xref, start, end, X, Z, d = NULL, g = 1/10000,
     method = c("alc", "alcray", "mspe", "nn", "fish"),
     Xi.ret = TRUE,
     close = min(1000*if(method == "alcray") 10 else 1, nrow(X
     alc.gpu = FALSE, numrays = ncol(X),
     rect = NULL, verb = 0)
```

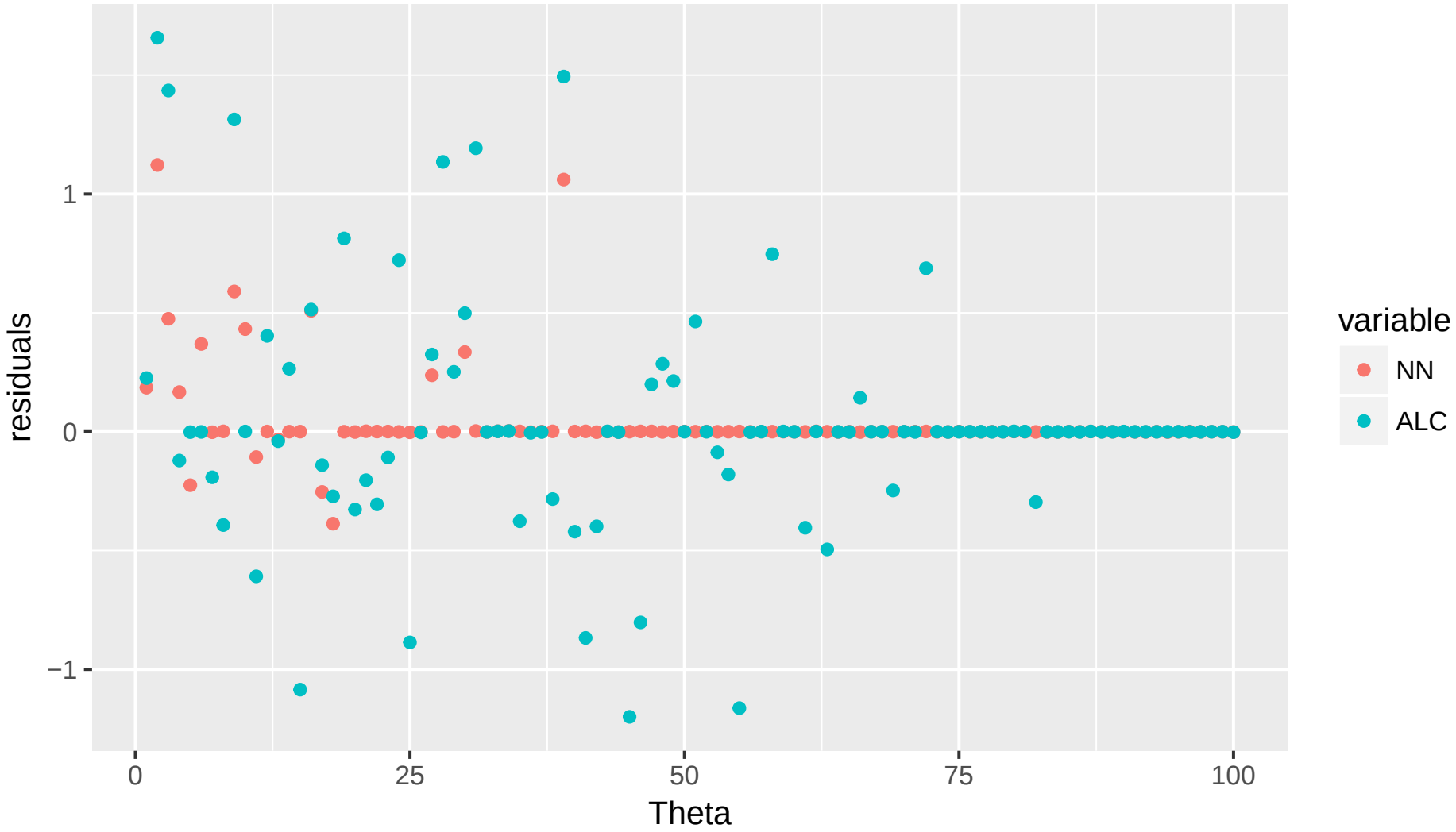
```
library(laGP)
## build up a design with N=~40K locations
x <- seq(-2, 2, by=0.02)
X <- as.matrix(expand.grid(x, x))
Z <- f2d(X)
## local analysis, first pass
Xref <- matrix(c(-1.725, 1.725), nrow=TRUE)
out <- laGP(Xref, 6, 100, X, Z, method="nn")
## second and pass via ALC, MSPE, and ALC-ray respectively
out2 <- laGP(Xref, 6, 100, X, Z, d=out$mle$d, method="alc", al
out2.alccpu <- laGP.R(Xref, 6, 100, X, Z, d=out2$mle$d, metho
out2.alcgpu <- laGP.R(Xref, 6, 100, X, Z, d=out2$mle$d, metho
```

	nn	alc	alcCPU	alcGPU
elapsed	0.23	0.59	1.46	0.62


comparing local designs
NN
ALC
x1
x2
−2.0
−1.9
−1.8
−1.7
−1.6
−1.5
−1.4
2.0
1.9
1.8
1.7
1.6
1.5
1.4

Show GIF


Prediction
2
1
0
−1
−2
0
25
50
75
100
Theta
Type
True
NN
ALC


residuals
1
0
−1
0
25
50
75
100
Theta
variable
NN
ALC

- ALC seems decent for prediction, but NN seems better
- It is not clear how much better it is to use ALC vs NN

- These two examples of GPU usage have contradicting ideas
  - One method says that points nearest are most important
  - One method says that you need to include some points outside of the nearest points
- Although it is important to note they set out for different purposes: estimation and prediction