Contemporaneous MCMC

Louis J. M. Aslett[1], Murray Pollock[2] & Gareth O. Roberts[3]
[1] Durham University
[2] Newcastle University
[3] University of Warwick

26 May 2022

Background

Contemporaneous MCMC

Asynchronus parallel adaptive population-based MCMC

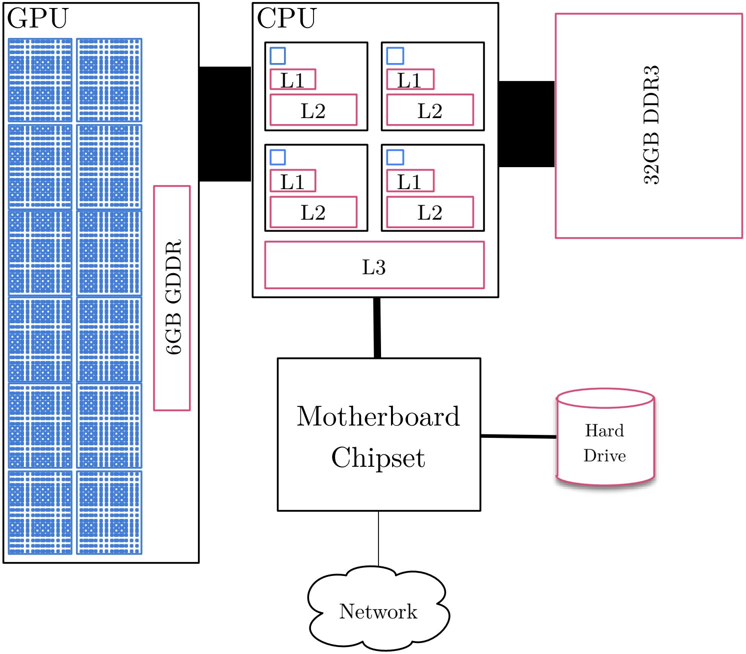
Contemporaneous MCMC

Asynchronus parallel adaptive population-based MCMC

contemporaneous

kənˌtɛmpəˈreɪnɪəs, kɒnˌtɛmpəˈreɪnɪəs

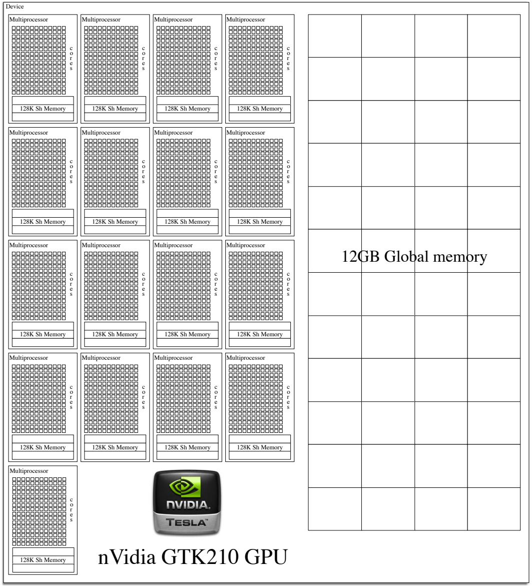
adjective existing at or occurring in the same period of time: *Pythagoras was contemporaneous with Buddha.*


GPU
6GB GDDR
CPU
L1
L2
L1
L2
L1
L2
L1
L2
L3
32GB DDR3
Motherboard Chipset
Hard Drive
Network

What are GPUs?

- GPUs are extraordinarily parallel devices (e.g. $16,896$ cores on Tesla H100 (2022))
- Usually programmed in C/C++ using CUDA
- Interfaces available in Python, R, Julia, ...
- Main mode of operation is SIMD
    - can now launch multiple independent kernels
- GPUs cannot directly access the system memory: you must copy data on and results off
    - CUDA has ‘unified memory’, but this just hides what is happening anyway

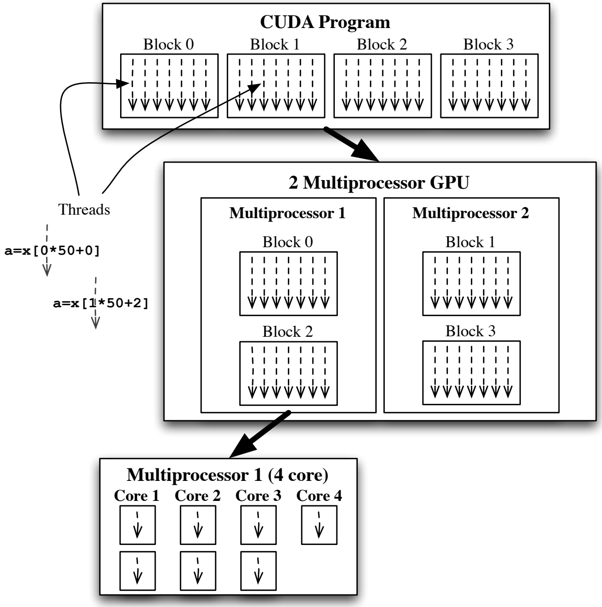
Highly simplified GPU architecture ($\sim$ 2014 to fit!!)



CUDA Concepts (Oversimplified, *cf* dynamic parallelism)

- **Kernel:** a C function which is flagged to be run on a CUDA capable device
- A kernel is executed on the core of a multiprocessor inside a *thread*. A thread can be thought of as just an index $j \in \mathbb{N}$. V Loosely: a index of cores in multiprocessors
- At any given time, a *block* of threads is executed on a multiprocessor. A block can be thought of as just an index $i \in \mathbb{N}$. V Loosely: an index of multiprocessors in devices
- Together, (i, j) corresponds to exactly one kernel running on a core of a single multiprocessor.

i.e. Very simplistically speaking, think of how to parallelize your problem by how to split it into identical chunks indexed by a pair $(i, j) \in \mathbb{N} \times \mathbb{N}$


CUDA Program
Block 0
Block 1
Block 2
Block 3
Threads
a=x[0*50+0]
a=x[1*50+2]
2 Multiprocessor GPU
Multiprocessor 1
Multiprocessor 2
Block 0
Block 1
Block 2
Block 3
Multiprocessor 1 (4 core)
Core 1
Core 2
Core 3
Core 4

Performance Considerations

Simplistically, for the purposes of today these are the key concerns when thinking of achieving high performance on a GPU:

1. Memory accesses are *slow* compared to the cores. Usually want many more total threads than cores to mask this.
2. Conditional sections of an algorithm can quickly kill performance.
3. Random or disorganised memory accesses will make a GPU under-perform a CPU!

Simple Performance Consideration #1

- Number of blocks can exceed number of multiprocessors
- Number of threads can exceed number of cores per multiprocessor

Worst case, at least both should equal the physical device sizes or else cores sit idle.

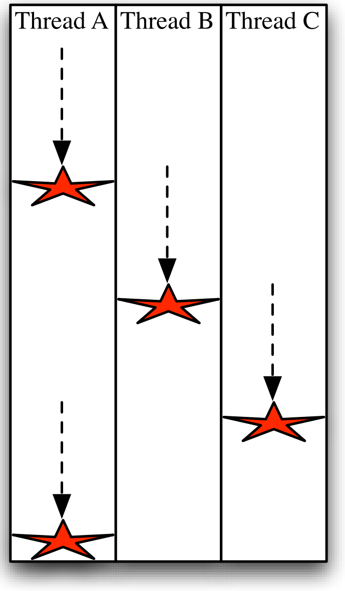
But in reality, rule of thumb is *ensure the thread figure exceeds the number of cores per multiprocessor* for performance reasons[1].

[1] this is a simplification ... $\exists$ occasions this is not true.

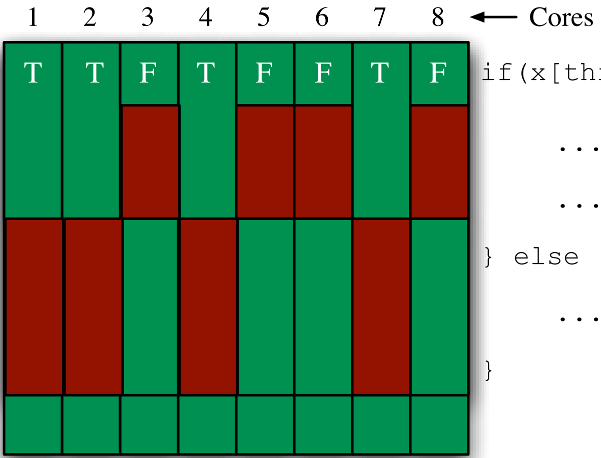
= Global memory access
=> Execution stall!

Global memory accesses are slow, so a core will stall when a request is made.

But, if # threads > # cores then another thread will be interleaved and run until the memory request is fulfilled and the first thread can run again.



Simple Performance Consideration #2



```
if(x[thrd_id]>0) {

      ...

      ...

} else {

      ...

}
```

Threads execute in lock-step on the cores of a multiprocessor, so beware of very divergent code ... best to use block indices to separate highly divergent paths.

Simple Performance Consideration #3

- Multiprocessors pull memory in large blocks, not element by element
  - due to the lock-step all cores will be ready for memory access at the same time.
- When a floating point number is requested from memory, that number and the following 3 are loaded (128-bit memory bus)
  - *whether you asked for them or not!*

Simple Performance Consideration #3

- Multiprocessors pull memory in large blocks, not element by element
  - due to the lock-step all cores will be ready for memory access at the same time.
- When a floating point number is requested from memory, that number and the following 3 are loaded (128-bit memory bus)
  - *whether you asked for them or not!*
- If consecutive threads require consecutive regions of memory, $\frac{1}{4}$ as many memory transactions required: *coalesced* memory access.
- Algorithms with random or disorganised memory access hurt performance.

Population Methods

Population-based MCMC : setting

- Target measure μ on space Ω with density $\pi(x)$.
- Construct:

$$\begin{aligned} \text{Product space} \qquad & \Omega^n := \Omega \times \cdots \times \Omega \\ \text{Density of form} \qquad & \tilde{\pi}(x) := \prod_{i=1}^{n} \pi_i(x_i) \\ & x \; = (x_1, \ldots, x_n) \in \Omega^n, \; x_i \in \Omega \end{aligned}$$

  where each $\pi_i(\cdot)$ is a probability density on Ω.

- Usually, $\pi_i \equiv \pi$ for at least one i.

Population-based MCMC : moves

- Individual site update

$$K(x, y) = \prod_{i=1}^{n} K_i(x_i, y_i)$$

e.g. $K_i(x_i, \cdot)$ may be $\pi_i(\cdot)$-stationary

Population-based MCMC : moves

- Individual site update

$$K(x,y) = \prod_{i=1}^{n} K_i(x_i, y_i)$$

  e.g. $K_i(x_i, \cdot)$ may be $\pi_i(\cdot)$-stationary

- Swap, $x = (\dots, x_i, \dots, x_j, \dots), y = (\dots, x_j, \dots, x_i, \dots)$

$$\alpha(x,y) = \min\left\{1, \frac{\pi_i(x_j)\pi_j(x_i)}{\pi_i(x_i)\pi_j(x_j)}\right\}$$

Population-based MCMC : moves

- Individual site update

$$K(x, y) = \prod_{i=1}^{n} K_i(x_i, y_i)$$

e.g. $K_i(x_i, \cdot)$ may be $\pi_i(\cdot)$-stationary

- Swap, $x = (\dots, x_i, \dots, x_j, \dots), y = (\dots, x_j, \dots, x_i, \dots)$

$$\alpha(x, y) = \min\left\{1, \frac{\pi_i(x_j)\pi_j(x_i)}{\pi_i(x_i)\pi_j(x_j)}\right\}$$

- Snooker, $\pi_i \equiv \pi \ \forall i$ (Gilks et al. 1994)

$$K(x, y) = \pi(y_i \,|\, y_i \in \ell) \prod_{j \neq i} \delta(y_j = x_j)$$

for ℓ the line connecting x_i to some randomly chosen anchor $x_j, j \neq i$

Population-based MCMC : moves

- Individual site update

$$K(x,y) = \prod_{i=1}^{n} K_i(x_i, y_i)$$

e.g. $K_i(x_i, \cdot)$ may be $\pi_i(\cdot)$-stationary

- Swap, $x = (\dots, x_i, \dots, x_j, \dots), y = (\dots, x_j, \dots, x_i, \dots)$

$$\alpha(x,y) = \min\left\{1, \frac{\pi_i(x_j)\pi_j(x_i)}{\pi_i(x_i)\pi_j(x_j)}\right\}$$

- Snooker, $\pi_i \equiv \pi \; \forall i$ (Gilks et al. 1994)

$$K(x,y) = \pi(y_i \,|\, y_i \in \ell) \prod_{j \neq i} \delta(y_j = x_j)$$

for ℓ the line connecting x_i to some randomly chosen anchor $x_j, j \neq i$

- **Simultaneous** — update all components in parallel

Population-based MCMC : $\pi_i(\cdot)$

- Tempered sequence

$$\pi_i(x) \propto (\pi(x))^{\gamma_i}\ ,\ \gamma_i \in (0,1]$$
$$\pi_n(x) = \pi(x)$$

Population-based MCMC : $\pi_i(\cdot)$

- Tempered sequence

$$\pi_i(x) \propto (\pi(x))^{\gamma_i}\ ,\ \gamma_i \in (0,1]$$
$$\pi_n(x) = \pi(x)$$

- For Bayes, $\pi(\theta) \propto L(\theta; y_{1:n})p(\theta)$, data tempering

$$\pi_i(\theta) \propto L(\theta; y_{1:i})\, p(\theta)$$

Population-based MCMC : $\pi_i(\cdot)$

- Tempered sequence

$$\pi_i(x) \propto (\pi(x))^{\gamma_i}\,,\ \gamma_i \in (0,1]$$
$$\pi_n(x) = \pi(x)$$

- For Bayes, $\pi(\theta) \propto L(\theta; y_{1:n})p(\theta)$, data tempering

$$\pi_i(\theta) \propto L(\theta; y_{1:i})\, p(\theta)$$

- Partitioned, $\Omega = E_1 \cup \cdots \cup E_{n-1},\ E_i \cap E_j = \varnothing$

$$\pi_i(x) = \pi(x)\, \mathbb{I}_{E_i}(x), \quad \pi_n(x) = \pi(x)$$

Population-based MCMC : $\pi_i(\cdot)$

- Tempered sequence

$$\pi_i(x) \propto (\pi(x))^{\gamma_i}\ ,\ \gamma_i \in (0,1]$$
$$\pi_n(x) = \pi(x)$$

- For Bayes, $\pi(\theta) \propto L(\theta; y_{1:n})p(\theta)$, data tempering

$$\pi_i(\theta) \propto L(\theta; y_{1:i})\, p(\theta)$$

- Partitioned, $\Omega = E_1 \cup \cdots \cup E_{n-1},\ E_i \cap E_j = \varnothing$

$$\pi_i(x) = \pi(x)\, \mathbb{I}_{E_i}(x), \quad \pi_n(x) = \pi(x)$$

- All equal (e.g Snooker)

$$\pi_i(x) = \pi(x)\ \forall i$$

Population-based MCMC : high-level algorithm

1. Initialise chain $(x_1, \ldots, x_n) \in \Omega^n$.
2. For $i \in \{1, \ldots, N\}$
    1. Iterate according to transition kernel $K(x, y)$, meaning π_i stationary $\forall\, i$.
    2. Perform interacting move (eg exchange) on some subset of components, $\mathcal{I}$, $\tilde{K}(x_{\mathcal{I}}, y_{\mathcal{I}})$

Population-based MCMC : parallelism

Lee et al. (2010) highlight existing population-based methods can be run on GPUs. They note:

- Individual site updates are trivially parallelisable
- Interaction moves (eg swap) are sequential in nature
    - parallelise only on disjoint subsets of variables
    - disjoint subsets must time-vary
- Not always helpful to increase n — may hinder convergence of chain

Contemporaneous MCMC

Adapting in parallel?

We want to propagate in parallel, but also adapt. Natural idea:

$$q_i(x_i, \cdot) \sim N(x_i, \Sigma_{-i})$$

where Σ_{-i} is the covariance of $\{x_j : j \in \{1, \dots, n\}, j \neq i\}$ and

$$\tilde{\pi}(x) = \prod_{i=1}^{n} \pi(x_i)$$

Adapting in parallel?

We want to propagate in parallel, but also adapt. Natural idea:

$$q_i(x_i, \cdot) \sim N(x_i, \Sigma_{-i})$$

where Σ_{-i} is the covariance of $\{x_j : j \in \{1, \ldots, n\}, j \neq i\}$ and

$$\tilde{\pi}(x) = \prod_{i=1}^{n} \pi(x_i)$$

Problems:

- inefficient to recompute n covariance matrices on every iteration
    - in practise would downdate global covariance matrix, but still expensive to do for all components.

Adapting in parallel?

We want to propagate in parallel, but also adapt. Natural idea:

$$q_i(x_i, \cdot) \sim N(x_i, \Sigma_{-i})$$

where Σ_{-i} is the covariance of $\{x_j : j \in \{1, \dots, n\}, j \neq i\}$ and

$$\tilde{\pi}(x) = \prod_{i=1}^{n} \pi(x_i)$$

Problems:

- inefficient to recompute n covariance matrices on every iteration
    - in practise would downdate global covariance matrix, but still expensive to do for all components.
- is it clear this actually a valid MCMC algorithm?

$\Longrightarrow$ transition kernel conditional on state:

$$K_i(x_i, y_i \,|\, x_{-i})$$

A more practical suggestion ...

As a quick fix, let

$$
\begin{aligned}
x^{(1)} &:= \{x_1, \ldots, x_{\frac{n}{2}}\} \\
x^{(2)} &:= \{x_{\frac{n}{2}+1}, \ldots, x_n\} \\
\tilde{\pi}(x) &= \pi(x^{(1)})\pi(x^{(2)}) = \prod_{i=1}^{n} \pi(x_i)
\end{aligned}
$$

A more practical suggestion ...

As a quick fix, let

$$\begin{aligned} x^{(1)} &:= \{x_1, \ldots, x_{\frac{n}{2}}\} \\ x^{(2)} &:= \{x_{\frac{n}{2}+1}, \ldots, x_n\} \\ \tilde{\pi}(x) &= \pi(x^{(1)})\pi(x^{(2)}) = \prod_{i=1}^{n} \pi(x_i) \end{aligned}$$

Define reversible kernel $K_1(x^{(2)}, y^{(2)})$ as parallel set of $\frac{n}{2}$ MH updates with proposals

$$q_{1i}(x_i^{(2)}, \cdot) \sim N(x_i^{(2)}, \Sigma_1), \quad i \in \left\{\frac{n}{2}, \ldots, n\right\}$$

where Σ_1 is the covariance of $x^{(1)}$.

Similarly define $K_2(x^{(1)}, y^{(1)})$.

Synchronus parallel MCMC

Alternate updates:

$$\pi(y^{(1)} \mid x^{(2)}) \text{ and } \pi(y^{(2)} \mid y^{(1)})$$

Synchronus parallel MCMC

Alternate updates:

$$\pi(y^{(1)} \mid x^{(2)}) \text{ and } \pi(y^{(2)} \mid y^{(1)})$$

- Size the population n such that $\frac{n}{2}$ sites are efficiently updated in parallel on the GPU at a time.
- Store $x^{(1)}, x^{(2)}$ as separate matrices in GPU memory $\implies$ in-place GPU computation of covariance.
- Alternate GPU kernel launches updating each population between updated covariance estimates.

Can we use more than 1 GPU efficiently?

Common to have more than 1 GPU.

(p2.16xlarge has 16 nVidia K80s)

- Hold separate populations on separate GPUs?
    - Effectively pointless: only one GPU computing at a time to satisfy Gibbs updates.

Can we use more than 1 GPU efficiently?

Common to have more than 1 GPU.

(p2.16xlarge has 16 nVidia K80s)

- Hold separate populations on separate GPUs?
  - Effectively pointless: only one GPU computing at a time to satisfy Gibbs updates.
- Split each population across GPUs?
  - Must synchronise GPUs to complete iterations together
  - Latency in computing covariance jointly across GPUs

Sticking plaster fix (III)

Note also the reversible kernels

$$\pi(x^{(1)})\pi(x^{(2)})K_1(x^{(2)},y^{(2)}) = \pi(x^{(1)})\pi(y^{(2)})K_1(y^{(2)},x^{(2)})$$
$$\pi(x^{(1)})\pi(x^{(2)})K_2(x^{(1)},y^{(1)}) = \pi(y^{(1)})\pi(x^{(2)})K_2(y^{(1)},x^{(1)})$$

satisfy detailed balance

$$\begin{aligned}
&\pi(x^{(1)})\pi(x^{(2)})K_1(x^{(2)},y^{(2)})K_2(x^{(1)},y^{(1)})\\
&= \frac{\cancel{\pi(x^{(1)})}\cancel{\pi(x^{(2)})}K_1(x^{(2)},y^{(2)})\pi(y^{(1)})\pi(x^{(2)})K_2(y^{(1)},x^{(1)})}{\cancel{\pi(x^{(1)})}\cancel{\pi(x^{(2)})}}\\
&= \frac{\cancel{\pi(x^{(1)})}\pi(y^{(2)})K_1(y^{(2)},x^{(2)})\pi(y^{(1)})\cancel{\pi(x^{(2)})}K_2(y^{(1)},x^{(1)})}{\cancel{\pi(x^{(1)})}\cancel{\pi(x^{(2)})}}\\
&= \pi(y^{(1)})\pi(y^{(2)})K_2(y^{(1)},x^{(1)})K_1(y^{(2)},x^{(2)})
\end{aligned}$$

Contemporaneous updates (I)

Consider a joint density $f(x^{(1)}, x^{(2)})$ which is such that the marginals are:

$$\int_{\Omega^{\frac{n}{2}}} f(dx^{(1)}, x^{(2)}) = \pi(x^{(2)})$$
$$\int_{\Omega^{\frac{n}{2}}} f(x^{(1)}, dx^{(2)}) = \pi(x^{(1)})$$

Contemporaneous updates (I)

Consider a joint density $f(x^{(1)}, x^{(2)})$ which is such that the marginals are:

$$\int_{\Omega^{\frac{n}{2}}} f(dx^{(1)}, x^{(2)}) = \pi(x^{(2)})$$
$$\int_{\Omega^{\frac{n}{2}}} f(x^{(1)}, dx^{(2)}) = \pi(x^{(1)})$$

If we could assure:

$$\int\int\int f(x^{(1)}, x^{(2)})\pi(y^{(1)}|x^{(2)})\pi(y^{(2)}|x^{(1)})\,dx^{(1)}dx^{(2)}dy^{(1)} = \pi(y^{(2)})$$

then we would be (almost) valid in updating populations simultaneously.

Contemporaneous updates (I)

Consider a joint density $f(x^{(1)}, x^{(2)})$ which is such that the marginals are:

$$\int_{\Omega^{\frac{n}{2}}} f(dx^{(1)}, x^{(2)}) = \pi(x^{(2)})$$
$$\int_{\Omega^{\frac{n}{2}}} f(x^{(1)}, dx^{(2)}) = \pi(x^{(1)})$$

If we could assure:

$$\int\int\int f(x^{(1)}, x^{(2)})\pi(y^{(1)}|x^{(2)})\pi(y^{(2)}|x^{(1)})\,dx^{(1)}dx^{(2)}dy^{(1)} = \pi(y^{(2)})$$

then we would be (almost) valid in updating populations simultaneously.

$\Longrightarrow$ **asynchronous adaptation.**

Contemporaneous updates (II)

$$
\begin{aligned}
&\int\int\int f(x^{(1)}, x^{(2)})\pi(y^{(1)} \mid x^{(2)})\pi(y^{(2)} \mid x^{(1)})\, dy^{(1)}dx^{(2)}dx^{(1)} \\
&= \int\int\int f(x^{(2)} \mid x^{(1)})\cancel{\pi(x^{(1)})}\frac{\pi(y^{(1)}, x^{(2)})}{\pi(x^{(2)})}\,\frac{\pi(x^{(1)}, y^{(2)})}{\cancel{\pi(x^{(1)})}}dy^{(1)}\,dx^{(2)}dx^{(1)} \\
&= \int\int f(x^{(2)} \mid x^{(1)})\frac{\cancel{\pi(x^{(2)})}}{\cancel{\pi(x^{(2)})}}\pi(x^{(1)}, y^{(2)})\,dx^{(2)}dx^{(1)} \\
&= \pi(y^{(2)})
\end{aligned}
$$

Thus, we can perform *simultaneous updates* in the special case where the marginals are $\pi(\cdot)$.

Contemporaneous updates (III)

In fact, (not shown) satisfied even for more groups if joint is simply product of marginals.

So:

- Infinite adaptation
  - no need for diminishing adaptation
  - no need for containment condition
- Asynchronus
- Parallel
- General framework
  - any transition kernel that can learn to adapt from other population
  - MALA, HMC, ...

Implications for GPU implementation

- Targeting $\pi_i \equiv \pi \, \forall \, i$ means we have most efficient possible SIMD execution.
  - same code paths, same memory access patterns
  - easy to arrange coalescent memory reads

Implications for GPU implementation

- Targeting $\pi_i \equiv \pi \; \forall \, i$ means we have most efficient possible SIMD execution.
  - same code paths, same memory access patterns
  - easy to arrange coalescent memory reads
- Covariance calculation is kept local to population on a given GPU.
  - zero communication
  - no synchronisation before computation can start

Implications for GPU implementation

- Targeting $\pi_i \equiv \pi \; \forall \, i$ means we have most efficient possible SIMD execution.
  - same code paths, same memory access patterns
  - easy to arrange coalescent memory reads
- Covariance calculation is kept local to population on a given GPU.
  - zero communication
  - no synchronisation before computation can start
- Additional π_i improves covariance estimation
  - can saturate GPU with work without harming statistical efficiency

Implications for GPU implementation

- Targeting $\pi_i \equiv \pi \, \forall \, i$ means we have most efficient possible SIMD execution.
  - same code paths, same memory access patterns
  - easy to arrange coalescent memory reads
- Covariance calculation is kept local to population on a given GPU.
  - zero communication
  - no synchronisation before computation can start
- Additional π_i improves covariance estimation
  - can saturate GPU with work without harming statistical efficiency
- Asynchronus updating eliminates latency
  - after covariance update immediately copy-and-continue
  - peer-to-peer memory copies in recent CUDA versions
  - ≈ 2.5 microsecond latency, > 6 GB/s throughput
  - free preload in L2 cache on target GPU
  - event dispatch for update notification

T

$x^{(1)}$ $x^{(2)}$

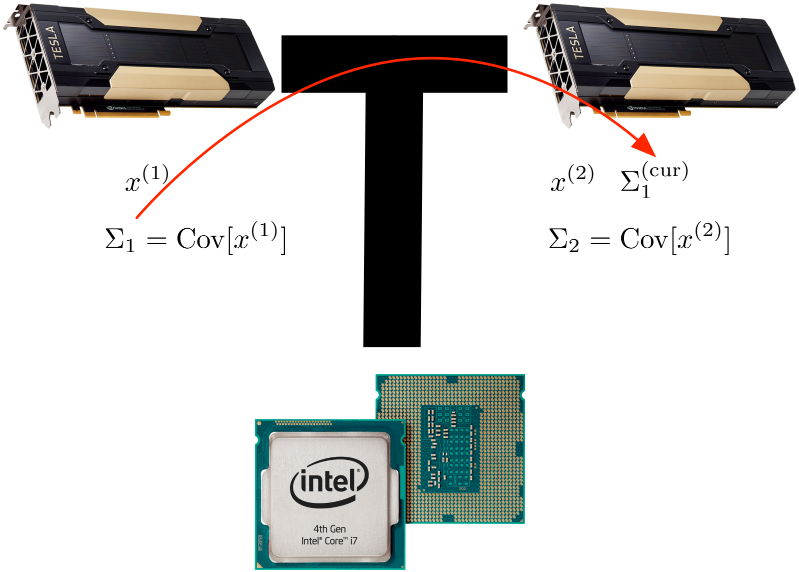
T

$x^{(1)}$

$x^{(2)}$

$$\Sigma_1 = \mathrm{Cov}[x^{(1)}]$$

$$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$$

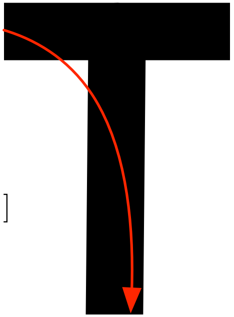


$x^{(1)}$

$\Sigma_1 = \mathrm{Cov}[x^{(1)}]$

$x^{(2)}$ $\Sigma_1^{(\mathrm{cur})}$

$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$

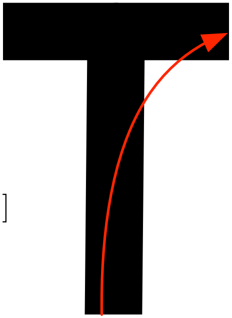


$x^{(1)}$

$$\Sigma_1 = \mathrm{Cov}[x^{(1)}]$$

$x^{(2)}$ $\Sigma_1^{(\mathrm{cur})}$

$$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$$



$x^{(1)}$

$$\Sigma_1 = \mathrm{Cov}[x^{(1)}]$$

$x^{(2)}$ $\Sigma_1^{(\mathrm{cur})}$

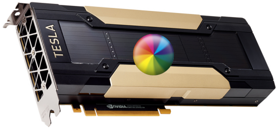
$$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$$

$\Sigma_2^{(\mathrm{cur})}$ $x^{(1)}$

$x^{(2)}$ $\Sigma_1^{(\mathrm{cur})}$

$$\Sigma_1 = \mathrm{Cov}[x^{(1)}]$$

$$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$$



$\Sigma_2^{(\text{cur})}$ $x^{(1)}$

$x^{(2)}$ $\Sigma_1^{(\text{cur})}$

$$\Sigma_1 = \text{Cov}[x^{(1)}]$$

$$\Sigma_2 = \text{Cov}[x^{(2)}]$$

$\Sigma_2^{(\mathrm{cur})}$ $x^{(1)} \to y^{(1)}$

$$\Sigma_1 = \mathrm{Cov}[x^{(1)}]$$

$x^{(2)}$ $\Sigma_1^{(\mathrm{cur})}$

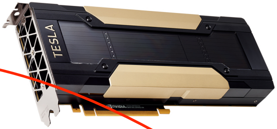
$$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$$

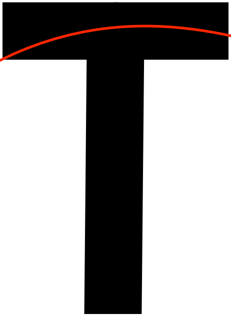
$\Sigma_2^{(\mathrm{cur})}$ $x^{(1)} \to y^{(1)}$

$\Sigma_1 = \mathrm{Cov}[y^{(1)}]$

$x^{(2)}$ $\Sigma_1^{(\mathrm{cur})}$

$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$



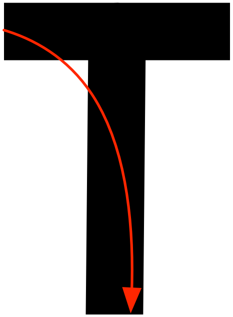


$\Sigma_2^{(\mathrm{cur})}$ $x^{(1)} \rightarrow y^{(1)}$

$\Sigma_1 = \mathrm{Cov}[y^{(1)}]$

$x^{(2)}$ $\Sigma_1^{(\mathrm{cur})}$ $\Sigma_1^{(\mathrm{nxt})}$

$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$



$\Sigma_2^{(\mathrm{cur})}$ $x^{(1)} \rightarrow y^{(1)}$

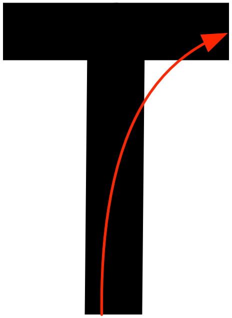
$$\Sigma_1 = \mathrm{Cov}[y^{(1)}]$$

$x^{(2)}$ $\Sigma_1^{(\mathrm{cur})}$ $\Sigma_1^{(\mathrm{nxt})}$

$$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$$

$\Sigma_2^{(\mathrm{cur})}$ $x^{(1)} \to y^{(1)}$

$\Sigma_1 = \mathrm{Cov}[y^{(1)}]$



$x^{(2)}$ $\Sigma_1^{(\mathrm{cur})} = \Sigma_1^{(\mathrm{nxt})}$

$\Sigma_2 = \mathrm{Cov}[x^{(2)}]$

Easy to code

```
float mylogdensity(float *theta, float *x) {
  // compute log density value using parameters theta[i] and
  // auxilliary info/static parameters x[i]
  return(res);
}
```

```
__device__ float mylogdensity(float *theta, float *x) {
  // compute log density value using parameters THETA[i] and
  // auxilliary info/static parameters X[i]
  return(res);
}
```

Results

Speed of adaptation

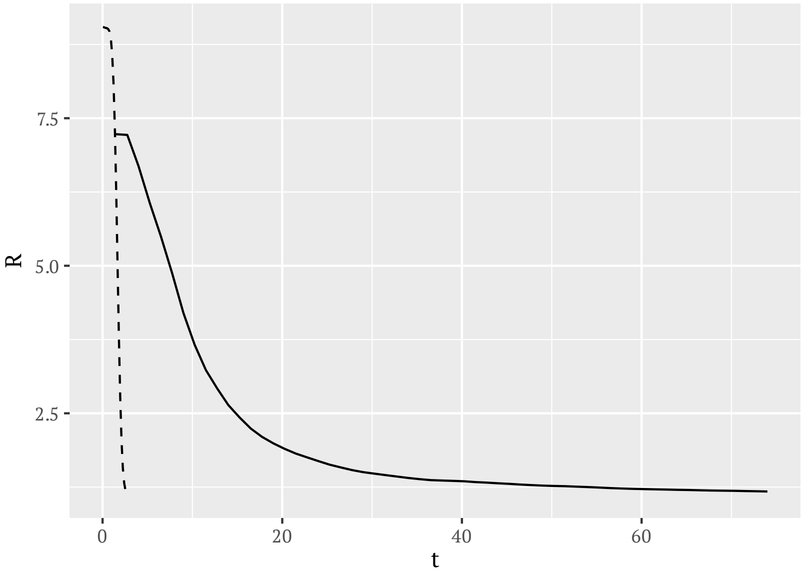
For a Gaussian target, cost of having proposal covariance V different to true covariance Σ is quantifiable as:

$$R = \frac{(\sum \lambda_i^2/d)^{1/2}}{\sum \lambda_i/d} = \frac{L2}{L1}$$

where λ_i are eigenvalues of $V^{1/2}\Sigma^{-1/2}$.

Using this measure, compare wall-clock speed to adaptation to a recent traditional adaptive scheme run on CPU.

Target π a 100-dimensional normal with $R = 9.04$ for initial covariance (diagonal $0.1^2/100$).


R
7.5
5.0
2.5
0
20
40
60
t

References

Lee, A., Yau, C., Giles, M. B., Doucet, A., & Holmes, C. C. (2010). On the utility of graphics cards to perform massively parallel simulation of advanced monte carlo methods. *Journal of Computational and Graphical Statistics*, 19/4: 769–89. DOI: 10.1198/jcgs.2010.10039