Computational Statistics and GPU Acceleration

A THESIS by

Andreas S. Bampouris

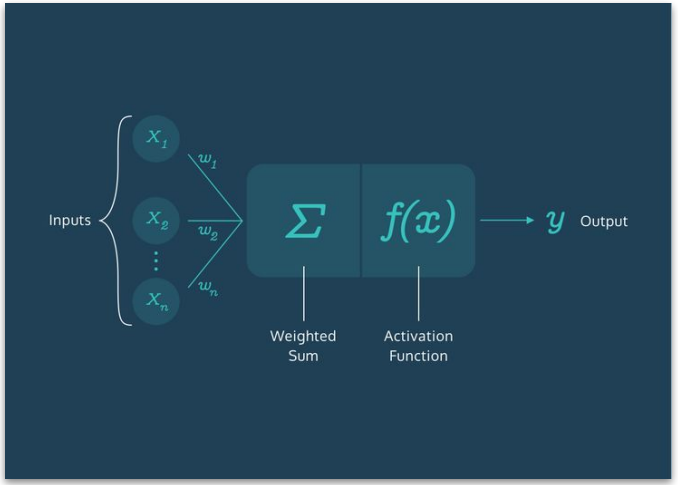
SUPERVISED by

Prof. Athanasios Yannacopoulos

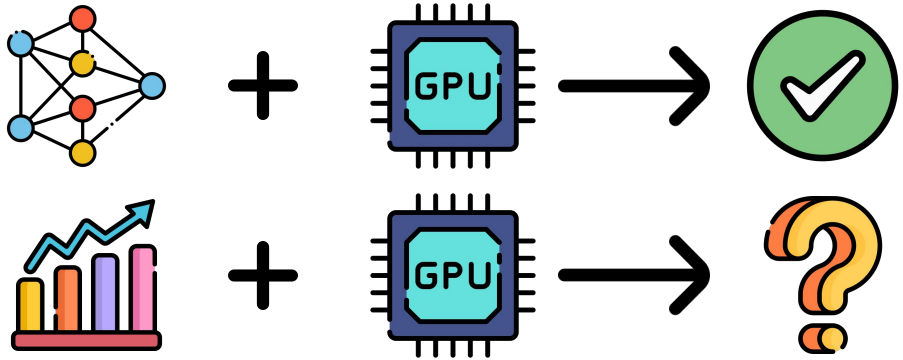
July 2025

Motivations

- The **AI** Revolution is Underway
- Built on a Foundation of **Statistics**
- Large-Scale **Computation** on **GPUs**



Core Question



*If **GPUs** transformed one area of **statistics**, what **potential** do they hold for the broader field?*

Before we start… ***What is a GPU?***

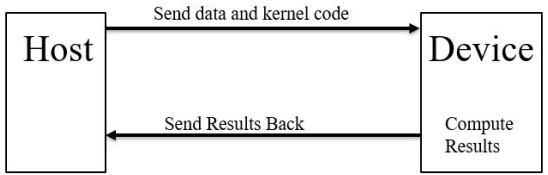
GPU Computing Model

CPU

- **Latency:** Few, powerful cores
- Serial / branch-heavy tasks
- **System RAM:** *von Neumann* bottleneck
- **Host:** Orchestrates program flow

GPU

- **Throughput:** Thousands of simple cores
- Massively parallel tasks
- **On-board VRAM:** Bandwidth-limited
- **Device:** Executes same kernel across cores

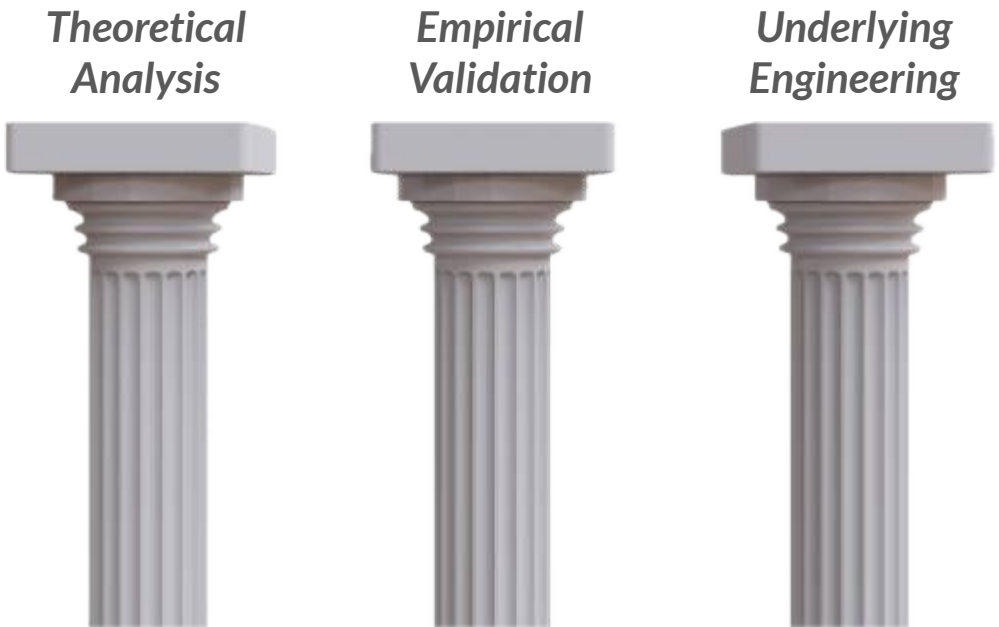


GPU Programming

- **Low-level:** CUDA, OpenCL
  - Granular Control, High Complexity
  - *Building high-performance libraries*
- **High-level:** PyTorch, TensorFlow
  - Abstracted Complexity
  - *Rapid prototyping, statistical modeling*

Falkon

Thesis Structure



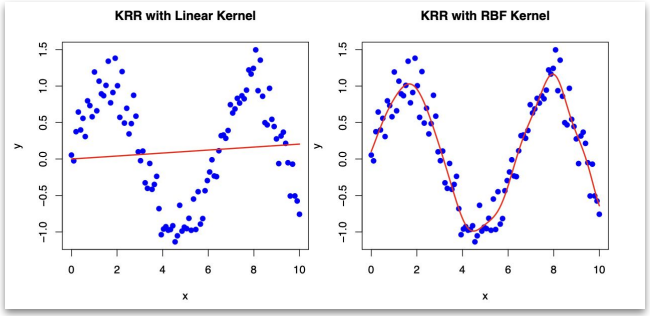
Case Study 1: Falkon

Kernel Methods at Scale

- **Kernel Trick** → Linear Separability
- RBF kernel:

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{2\sigma^2}\right)$$

- Applications: *SVM*, *GP*, ***KRR***…



KRR's Computational Wall

- Minimization problem has closed-form solution:

$$\alpha = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y}$$

- … but it depends on $\mathbf{K}$, which is $\mathrm{n} \times \mathrm{n}$.

$O(\mathrm{n}^2)$ Memory *and* $O(\mathrm{n}^3)$ Time

- Just *100,000 points* in *64-bit precision* costs **> 70 GB of RAM!**

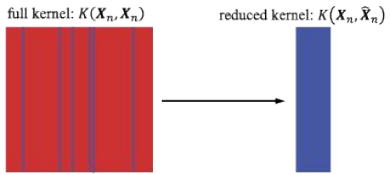
Nyström Approximation

- Approximates with m *landmark points*
- Coefficients α no longer require full $\mathbf{K}$:

$$\alpha = (\mathbf{K}_{nm}^\top \mathbf{K}_{nm} + \lambda \mathbf{K}_{mm})^{-1} \mathbf{K}_{nm}^\top \mathbf{y}$$

- For $\mathrm{m} \ll \mathrm{n}$:

$O(\mathrm{n}\,\mathrm{m})$ Memory *and* $O(\mathrm{n}\,\mathrm{m}^2)$ Time



Nyström approximation matrix

$$\boldsymbol{K}_n \approx \widehat{\boldsymbol{K}}_n \left(K(\widehat{\boldsymbol{X}}_n, \widehat{\boldsymbol{X}}_n) \right)^{-1} \widehat{\boldsymbol{K}}_n^T,$$

full kernel: $\boldsymbol{K}_n := K(\boldsymbol{X}_n, \boldsymbol{X}_n)$,

reduced kernel: $\widehat{\boldsymbol{K}}_n := K(\boldsymbol{X}_n, \widehat{\boldsymbol{X}}_n)$.

Solving Iteratively

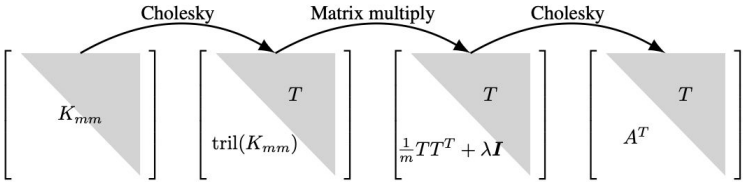
- **Conjugate gradient** method more efficient than direct solver
- Applicable for squared loss as well as other loss functions
- **Preconditioner** $\tilde{\mathbf{P}} = \frac{1}{\sqrt{n}}\mathbf{T}^{-1}\mathbf{A}^{-1}$ accelerates convergence
- **Cholesky decomposition** used to obtain $\mathbf{T} = \text{chol}(\mathbf{K}_{mm})$ and $\mathbf{A} = \text{chol}\left(\frac{1}{m}\mathbf{T}\mathbf{T}^\top + \lambda\mathbf{I}_m\right)$
- **Nyström** applied to approximate $\mathbf{K}_{nm}^\top\mathbf{K}_{nm}$ within $\mathbf{H} = (\mathbf{K}_{nm}^\top\mathbf{K}_{nm} + \lambda\mathbf{K}_{mm})^{-1}$ as $\approx \frac{n}{m}\mathbf{K}_{mm}^2$
- Coefficients $\alpha = \tilde{\mathbf{P}}\beta$ obtained by solving via CG for β:

$$\tilde{\mathbf{P}}^\top\mathbf{H}\tilde{\mathbf{P}}\beta = \tilde{\mathbf{P}}^\top\mathbf{K}_{nm}^\top\mathbf{y}$$

O (n sqrt(n) log n) Memory *and* O(n) Time

Falkon's GPU implementation

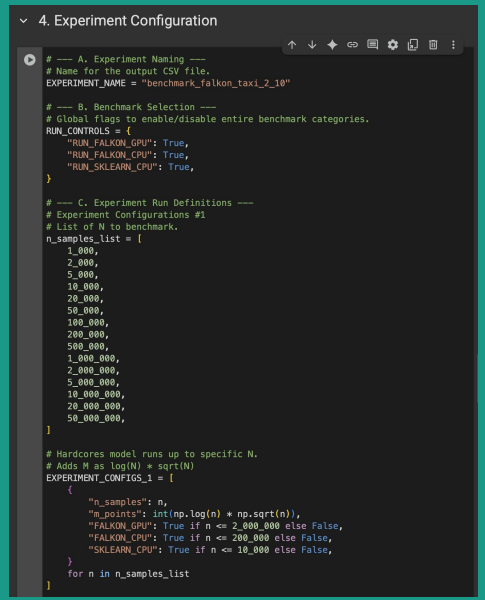
- Memory footprint minimized: $\mathrm{m} \times \mathrm{m}$ and β
- **Out-of-Core** operations if needed
- $\mathbf{K}nm$ calculated in blocks on GPU
- Mixed 32-bit / 64-bit precision



Falkon Benchmark

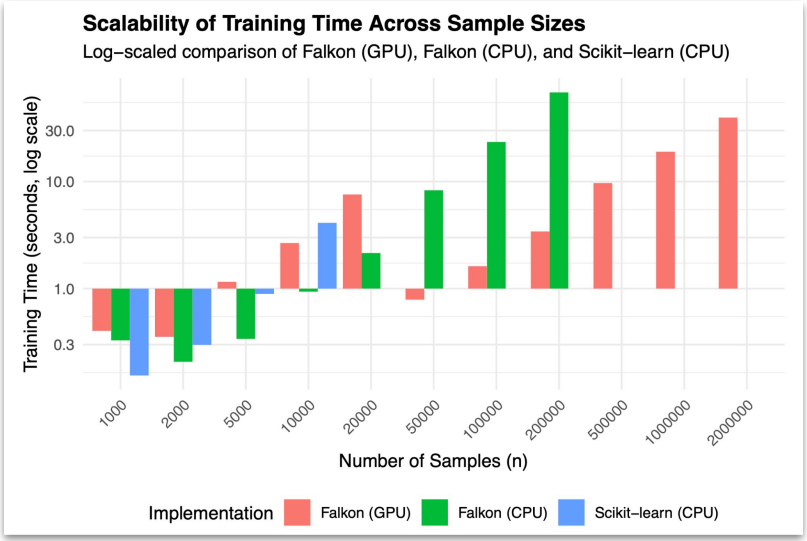
Table 3.1: Description of NYC Taxi Fare features used for model training.

Feature Name	Description	Type
`VendorID`	A code indicating the TPEP provider.	Categorical
`passenger_count`	The number of passengers in the vehicle.	Numerical
`trip_distance`	The elapsed trip distance in miles.	Numerical
`RatecodeID`	The final rate code for the trip.	Categorical
`PULocationID`	TLC Taxi Zone ID where the trip began.	Categorical
`DOLocationID`	TLC Taxi Zone ID where the trip ended.	Categorical
`payment_type`	A numeric code indicating how the passenger paid.	Categorical
`duration`	*Engineered:* Total trip duration in minutes.	Numerical
`pickup_day`	*Engineered:* Day of the week (0=Mon, 6=Sun).	Numerical
`pickup_hour`	*Engineered:* Hour of the day (0-23).	Numerical
`fare_amount`	The time-and-distance fare calculated by the meter.	**Target**



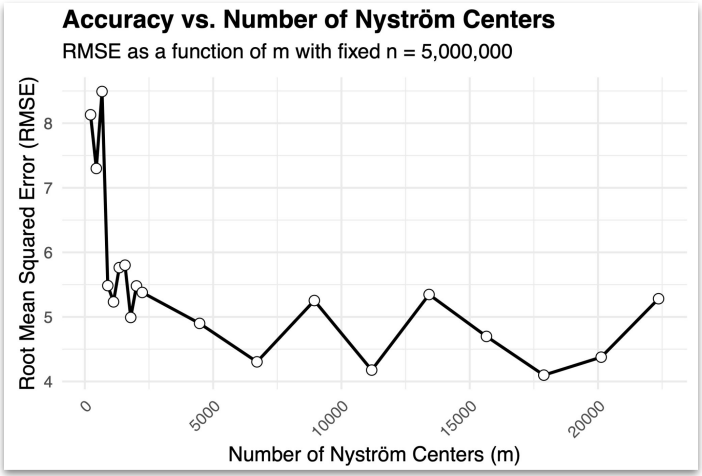
Falkon's Scalability with Sample Size (*n*)

n	m	Implementation	Train Time (s)	Pred. Time (s)	RMSE	R^2
1,000	218	**Falkon (GPU)**	0.40 ± 0.18	0.17 ± 0.02	4.6489	0.9279
1,000	218	Falkon (CPU)	0.33 ± 0.18	0.08 ± 0.01	4.1540	0.9424
1,000	NA	Scikit-learn (CPU)	0.15 ± 0.01	0.84 ± 0.01	4.9229	0.9192
2,000	339	**Falkon (GPU)**	0.35 ± 0.02	0.00 ± 0.00	4.5030	0.9324
2,000	339	Falkon (CPU)	0.21 ± 0.01	0.10 ± 0.00	3.9475	0.9480
2,000	NA	Scikit-learn (CPU)	0.30 ± 0.15	1.64 ± 0.02	4.4838	0.9329
5,000	602	**Falkon (GPU)**	1.16 ± 0.30	0.00 ± 0.00	4.3895	0.9357
5,000	602	Falkon (CPU)	0.34 ± 0.00	0.17 ± 0.00	3.6841	0.9547
5,000	NA	Scikit-learn (CPU)	0.89 ± 0.07	4.13 ± 0.03	4.2894	0.9386
10,000	921	**Falkon (GPU)**	2.67 ± 0.23	0.00 ± 0.00	4.0620	0.9450
10,000	921	Falkon (CPU)	0.94 ± 0.14	0.25 ± 0.00	3.6450	0.9557
10,000	NA	Scikit-learn (CPU)	4.12 ± 0.08	7.50 ± 0.04	4.2117	0.9408
20,000	1400	**Falkon (GPU)**	7.58 ± 0.23	0.00 ± 0.00	4.0692	0.9448
20,000	1400	Falkon (CPU)	2.15 ± 0.14	0.37 ± 0.01	3.8172	0.9514
50,000	2419	**Falkon (GPU)**	0.79 ± 0.02	0.00 ± 0.00	4.0378	0.9456
50,000	2419	Falkon (CPU)	8.31 ± 0.25	0.60 ± 0.00	3.6291	0.9561
100,000	3640	**Falkon (GPU)**	1.62 ± 0.17	0.00 ± 0.00	3.9626	0.9476
100,000	3640	Falkon (CPU)	23.48 ± 0.11	0.90 ± 0.00	3.5804	0.9572
200,000	5458	**Falkon (GPU)**	3.42 ± 0.03	0.01 ± 0.00	3.9818	0.9471
200,000	5458	Falkon (CPU)	68.19 ± 0.84	1.44 ± 0.16	3.5490	0.9580
500,000	9278	**Falkon (GPU)**	9.69 ± 0.14	0.01 ± 0.00	4.0204	0.9461
1,000,000	13815	**Falkon (GPU)**	19.02 ± 0.31	0.01 ± 0.00	3.9845	0.9470
2,000,000	20518	**Falkon (GPU)**	39.62 ± 0.27	0.01 ± 0.00	4.2097	0.9409



Accuracy vs. Nyström Centers (*m*)

n	m	Implementation	Train Time (s)	Pred. Time (s)	RMSE	R^2
5,000,000	223	**Falkon (GPU)**	12.20 ± 0.44	0.13 ± 0.01	8.1308	0.7795
5,000,000	447	**Falkon (GPU)**	12.12 ± 0.11	0.00 ± 0.00	7.2994	0.8223
5,000,000	670	**Falkon (GPU)**	12.03 ± 0.02	0.00 ± 0.00	8.4913	0.7595
5,000,000	894	**Falkon (GPU)**	12.12 ± 0.02	0.00 ± 0.00	5.4847	0.8996
5,000,000	1,118	**Falkon (GPU)**	12.34 ± 0.23	0.00 ± 0.00	5.2332	0.9086
5,000,000	1,341	**Falkon (GPU)**	12.40 ± 0.05	0.00 ± 0.00	5.7635	0.8892
5,000,000	1,565	**Falkon (GPU)**	12.58 ± 0.03	0.00 ± 0.00	5.8015	0.8877
5,000,000	1,788	**Falkon (GPU)**	12.71 ± 0.07	0.00 ± 0.00	4.9920	0.9169
5,000,000	2,012	**Falkon (GPU)**	12.91 ± 0.10	0.00 ± 0.00	5.4802	0.8998
5,000,000	2,236	**Falkon (GPU)**	13.02 ± 0.03	0.00 ± 0.00	5.3791	0.9035
5,000,000	4,472	**Falkon (GPU)**	14.90 ± 0.07	0.00 ± 0.00	4.8998	0.9199
5,000,000	6,708	**Falkon (GPU)**	17.48 ± 0.22	0.01 ± 0.00	4.3043	0.9382
5,000,000	8,944	**Falkon (GPU)**	21.22 ± 0.45	0.01 ± 0.00	5.2537	0.9079
5,000,000	11,180	**Falkon (GPU)**	24.78 ± 0.25	0.01 ± 0.00	4.1767	0.9418
5,000,000	13,416	**Falkon (GPU)**	29.34 ± 0.08	0.01 ± 0.00	5.3471	0.9046
5,000,000	15,652	**Falkon (GPU)**	35.11 ± 0.11	0.01 ± 0.00	4.6981	0.9264
5,000,000	17,888	**Falkon (GPU)**	42.38 ± 0.09	0.01 ± 0.00	4.0985	0.9440
5,000,000	20,124	**Falkon (GPU)**	50.88 ± 0.16	0.01 ± 0.00	4.3770	0.9361
5,000,000	22,360	**Falkon (GPU)**	61.14 ± 0.42	0.01 ± 0.00	5.2821	0.9069



Case Study 2: XGBoost

*Extreme **Gradient Boosting***

- **Tree-based** methods minimize **Gain:**

$$\Delta H = H(R_t) - \frac{|R_{t_L}|}{|R_t|} H(R_{t_L}) - \frac{|R_{t_R}|}{|R_t|} H(R_{t_R})$$

- **Boosting**: Ensemble of trees
- **Gradient Boosting**: Functional GD

Learning Objective

- Iteration m represents the mth tree of the ensemble, with objective function:

$$\mathcal{L}^{(m)} = \sum_{i=1}^{n} L(y_i, F_{m-1}(\mathbf{x}_i) + h_m(\mathbf{x}_i)) + \Omega(h_m)$$

- *L* can be squared-error or log-loss, Ω is regularization term:

$$\Omega(h_m) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

- Optimize second-order Taylor expansion:

$$\mathcal{L}^{(m)} \approx \sum_{i=1}^{n} [g_i h_m(\mathbf{x}_i) + \frac{1}{2} h_i h_m^2(\mathbf{x}_i)] + \Omega(h_m)$$

Split-Finding as a Bottleneck

- Gain can be maximized using **Exact Greedy Algorithm**, e.g. for MSE:

$$\text{Gain} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma$$

- **Sorting required**, which results in $O(p\, n_N \log n_N)$ complexity
- Linear scan and evaluation of Gain at **every unique feature value**

Approximate Algorithm

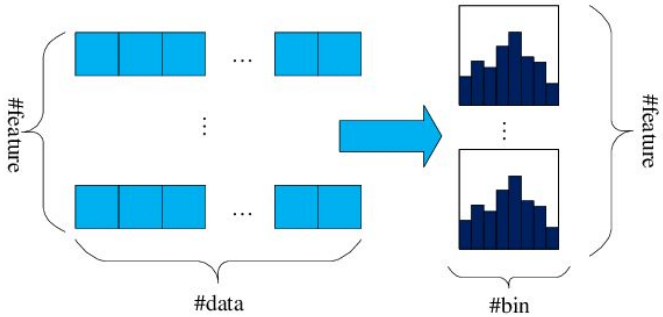
- Limited candidate split points proposed based on **percentiles** of feature's distribution
- **Weighted Quantile Sketch** → Hessian-weighted instances for proposing candidates
- Evaluation of Gain only **at candidate split points**
- Global and Local Variants
- Basis for Histogram-Based Method…

Histogram-Based Method

- Feature Discretization via WQS
- Few bins $B \rightarrow$ compact, memory-efficient
- Histogram Aggregation, binning by feature:

$$G_b = \sum_{i \in b} g_i \qquad H_b = \sum_{i \in \text{bin}_b} h_i$$

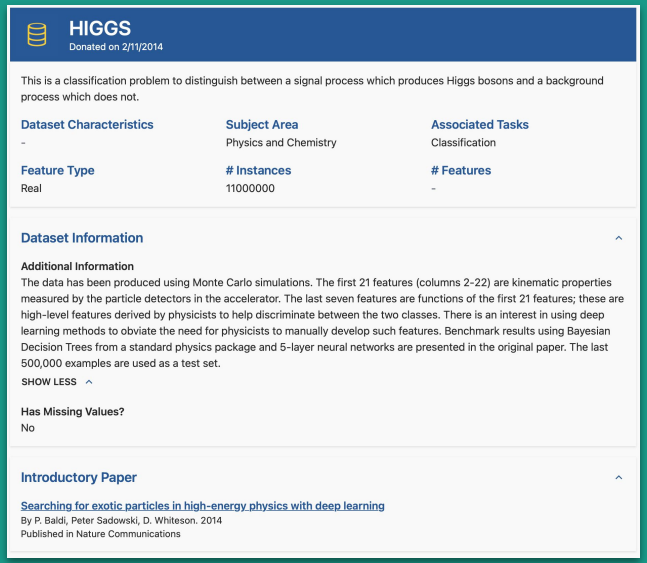
- Linear scan and evaluation of Gain only **over histogram bins**
- $O(p(n_N + B)) \approx O(p\, n_N)$



Histogram Method on GPU

- Computational profile perfect match for GPU: **Warps ≡ Tree Nodes** in gpu_hist
- Building a Histogram is a **“Parallel Reduction”**, classic SIMD pattern
  - multiscan and multireduce operations
  - **CUB** and **Thrust** for parallel primitives like scan / reduce / radix sort
- Sparse, column-major **data layout optimized for GPU**
- Single pre-sorting for *shallow trees*, data “interleaved” in memory
- GPU radix sort for *deep trees*, enabling use of segmented scan
- More system-level optimizations, like sparsity-aware splitting…

XGBoost Benchmark



HIGGS
Donated on 2/11/2014

This is a classification problem to distinguish between a signal process which produces Higgs bosons and a background process which does not.

Dataset Characteristics	Subject Area	Associated Tasks
-	Physics and Chemistry	Classification
Feature Type	**# Instances**	**# Features**
Real	11000000	-

Dataset Information

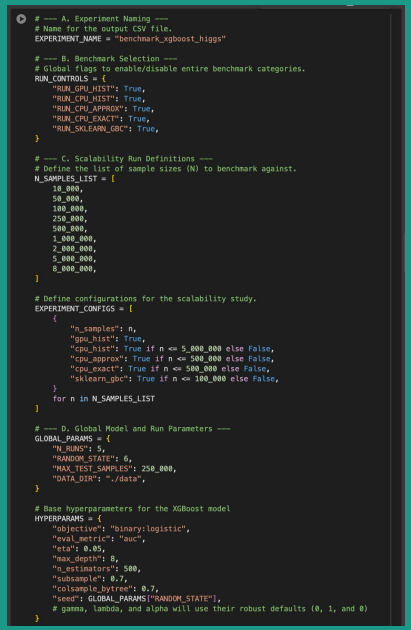
Additional Information
The data has been produced using Monte Carlo simulations. The first 21 features (columns 2-22) are kinematic properties measured by the particle detectors in the accelerator. The last seven features are functions of the first 21 features; these are high-level features derived by physicists to help discriminate between the two classes. There is an interest in using deep learning methods to obviate the need for physicists to manually develop such features. Benchmark results using Bayesian Decision Trees from a standard physics package and 5-layer neural networks are presented in the original paper. The last 500,000 examples are used as a test set.

SHOW LESS

Has Missing Values?
No

Introductory Paper

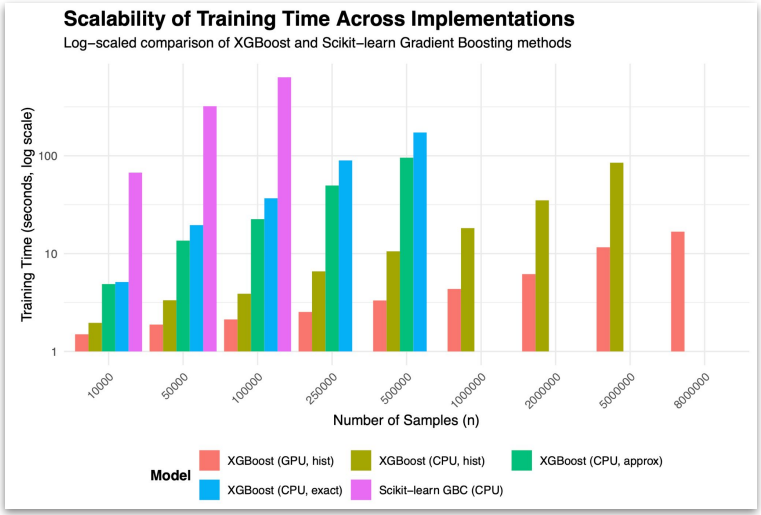
Searching for exotic particles in high-energy physics with deep learning
By P. Baldi, Peter Sadowski, D. Whiteson. 2014
Published in Nature Communications



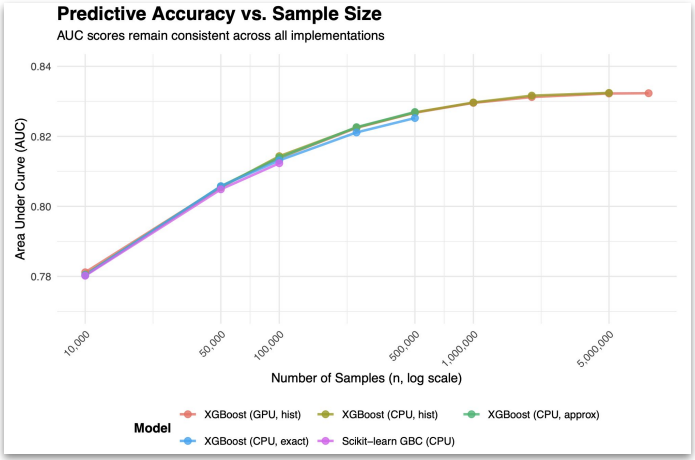
```python
# === A. Experiment Naming ===
# Name for the output CSV file.
EXPERIMENT_NAME = "benchmark_xgboost_higgs"

# === B. Benchmark Selection ===
# Global flags to enable/disable entire benchmark categories.
RUN_CONTROLS = {
    "RUN_GPU_HIST": True,
    "RUN_CPU_HIST": True,
    "RUN_CPU_APPROX": True,
    "RUN_CPU_EXACT": True,
    "RUN_SKLEARN_GBC": True,
}

# === C. Scalability Run Definitions ===
# Define the list of sample sizes (N) to benchmark against.
N_SAMPLES_LIST = [
    10_000,
    50_000,
    100_000,
    250_000,
    500_000,
    1_000_000,
    2_000_000,
    5_000_000,
    8_000_000,
]

# Define configurations for the scalability study.
EXPERIMENT_CONFIGS = [
    {
        "n_samples": n,
        "gpu_hist": True,
        "cpu_hist": True if n <= 5_000_000 else False,
        "cpu_approx": True if n <= 500_000 else False,
        "cpu_exact": True if n <= 500_000 else False,
        "sklearn_gbc": True if n <= 100_000 else False,
    }
    for n in N_SAMPLES_LIST
]

# === D. Global Model and Run Parameters ===
GLOBAL_PARAMS = {
    "N_RUNS": 5,
    "RANDOM_STATE": 6,
    "MAX_TEST_SAMPLES": 250_000,
    "DATA_DIR": "./data",
}

# Base hyperparameters for the XGBoost model
HYPERPARAMS = {
    "objective": "binary:logistic",
    "eval_metric": "auc",
    "eta": 0.05,
    "max_depth": 8,
    "n_estimators": 500,
    "subsample": 0.7,
    "colsample_bytree": 0.7,
    "seed": GLOBAL_PARAMS["RANDOM_STATE"],
    # gamma, lambda, and alpha will use their robust defaults (0, 1, and 0)
}
```

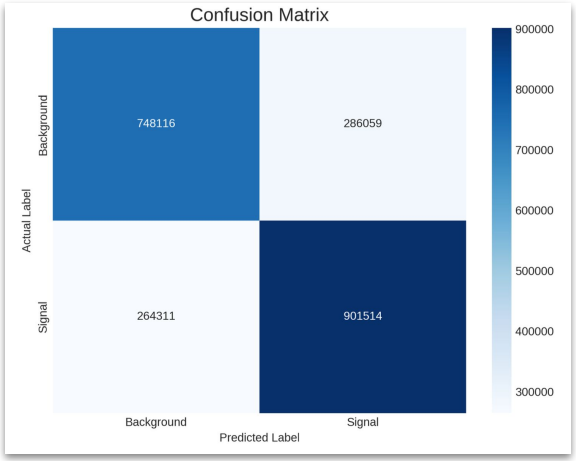
XGBoost's Scalability with Sample Size (*n*)

n	Implementation	Train Time (s)	Pred. Time (s)	AUC	Accuracy	F_1
10,000	**XGBoost (GPU, hist)**	1.50 ± 0.08	0.64 ± 0.01	0.7812	0.7079	0.7276
10,000	XGBoost (CPU, hist)	1.96 ± 0.70	5.12 ± 0.09	0.7804	0.7073	0.7269
10,000	XGBoost (CPU, approx)	4.89 ± 0.74	5.13 ± 0.14	0.7805	0.7072	0.7269
10,000	XGBoost (CPU, exact)	5.13 ± 0.81	5.10 ± 0.13	0.7803	0.7069	0.7266
10,000	Scikit-learn (CPU)	67.30 ± 0.06	47.65 ± 0.03	0.7802	0.7072	0.7265
50,000	**XGBoost (GPU, hist)**	1.88 ± 0.03	0.65 ± 0.01	0.8056	0.7269	0.7443
50,000	XGBoost (CPU, hist)	3.34 ± 0.69	5.28 ± 0.16	0.8054	0.7270	0.7444
50,000	XGBoost (CPU, approx)	13.59 ± 0.10	5.27 ± 0.11	0.8057	0.7271	0.7445
50,000	XGBoost (CPU, exact)	19.58 ± 0.68	5.19 ± 0.13	0.8058	0.7271	0.7447
50,000	Scikit-learn (CPU)	321.42 ± 0.71	45.01 ± 0.04	0.8049	0.7261	0.7432
100,000	**XGBoost (GPU, hist)**	2.12 ± 0.04	0.66 ± 0.01	0.8138	0.7336	0.7504
100,000	XGBoost (CPU, hist)	3.89 ± 0.46	5.40 ± 0.14	0.8143	0.7341	0.7511
100,000	XGBoost (CPU, approx)	22.52 ± 0.62	5.37 ± 0.10	0.8139	0.7337	0.7507
100,000	XGBoost (CPU, exact)	36.82 ± 0.29	5.26 ± 0.12	0.8131	0.7332	0.7501
100,000	Scikit-learn (CPU)	635.18 ± 1.19	44.07 ± 0.04	0.8123	0.7322	0.7492
250,000	**XGBoost (GPU, hist)**	2.53 ± 0.02	0.64 ± 0.01	0.8224	0.7409	0.7578
250,000	XGBoost (CPU, hist)	6.59 ± 0.84	5.43 ± 0.14	0.8225	0.7409	0.7579
250,000	XGBoost (CPU, approx)	49.71 ± 0.85	5.46 ± 0.16	0.8226	0.7413	0.7582
250,000	XGBoost (CPU, exact)	89.76 ± 1.18	5.28 ± 0.11	0.8211	0.7396	0.7565
500,000	**XGBoost (GPU, hist)**	3.32 ± 0.04	0.65 ± 0.01	0.8268	0.7448	0.7615
500,000	XGBoost (CPU, hist)	10.56 ± 0.83	5.52 ± 0.20	0.8268	0.7447	0.7614
500,000	XGBoost (CPU, approx)	95.76 ± 1.14	5.47 ± 0.10	0.8270	0.7448	0.7615
500,000	XGBoost (CPU, exact)	173.17 ± 0.78	5.35 ± 0.10	0.8252	0.7436	0.7603
1,000,000	**XGBoost (GPU, hist)**	4.35 ± 0.08	0.65 ± 0.02	0.8296	0.7471	0.7638
1,000,000	XGBoost (CPU, hist)	18.23 ± 0.73	5.46 ± 0.11	0.8297	0.7473	0.7639
2,000,000	**XGBoost (GPU, hist)**	6.17 ± 0.04	0.64 ± 0.01	0.8312	0.7487	0.7652
2,000,000	XGBoost (CPU, hist)	35.07 ± 0.56	5.55 ± 0.12	0.8316	0.7491	0.7656
5,000,000	**XGBoost (GPU, hist)**	11.61 ± 0.02	0.67 ± 0.02	0.8322	0.7498	0.7661
5,000,000	XGBoost (CPU, hist)	84.97 ± 0.72	5.43 ± 0.09	0.8324	0.7498	0.7661
8,000,000	**XGBoost (GPU, hist)**	16.76 ± 0.08	0.66 ± 0.02	0.8323	0.7498	0.7661



Assessment of Statistical Power





GPU Programming: CUDA Introduction

A Massively Parallel ***MCMC Sampler***

- Bayesian inference:
  a. *update beliefs* on parameters θ…
  b. in light of *observed data* $\mathbf{y}$

$$p(\theta|\mathbf{y}) = \frac{p(\mathbf{y}|\theta)p(\theta)}{p(\mathbf{y})} \propto p(\mathbf{y}|\theta)p(\theta)$$

- **Posterior** distribution $\mathrm{p}\,(\theta \mid \mathbf{y})$
- **Likelihood** $\mathrm{p}\,(\mathbf{y} \mid \theta)$
- **Prior** $\mathrm{p}\,(\theta)$
- **Evidence** $\mathrm{p}\,(\mathbf{y})$ often intractable
- MCMC numerical solution to $\mathrm{p}\,(\theta \mid \mathbf{y})$
- Metropolis-Hastings algorithm
  a. $\theta_t \rightarrow \theta'$ from proposal distribution
  b. Accept θ' with probability α
  c. After burn-in, MC with $\mathrm{p}\,(\theta \mid \mathbf{y})$

$$\alpha(\theta', \theta_t) = \min\left(1, \frac{p(\theta'|\mathbf{y})q(\theta_t|\theta')}{p(\theta_t|\mathbf{y})q(\theta'|\theta_t)}\right)$$

Motivating Example

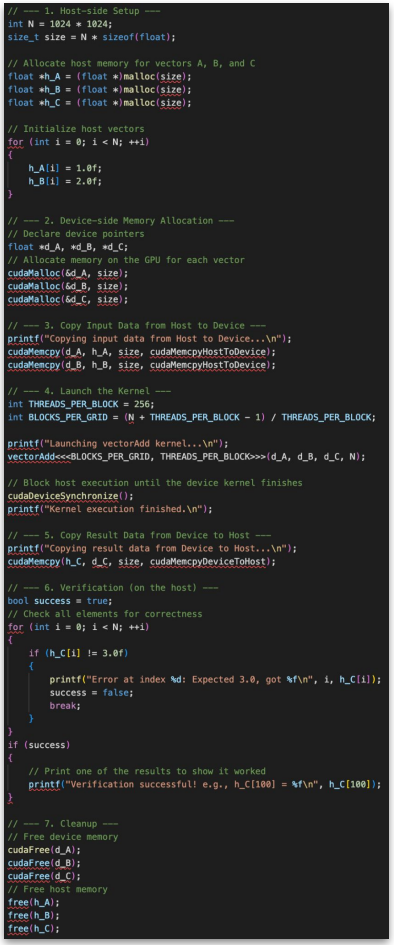
- *Trivial* sampler for inferring **scalar mean μ** of a **Normal distribution**

$$p(\mathbf{y}|\mu,\sigma^2) = \prod_{i=1}^{N} \mathcal{N}(y_i|\mu,\sigma^2) \qquad p(\mu) = \mathcal{N}(\mu|\mu_0,\tau_0^2)$$

- Markovian, thus sequential by nature
- *Key insight:* Running many **independent chains is embarrassingly parallel**

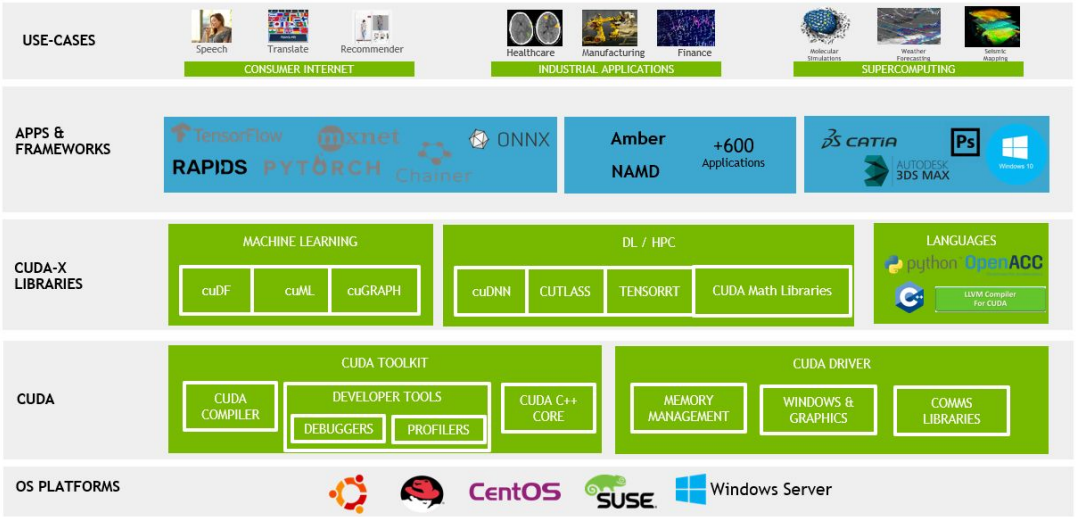
Data Pipeline

1. Host-side setup
2. Device-side setup
3. Copy data from host to device
4. Launch kernel
5. Copy data from device to host
6. Verify on host
7. Clean host and device memory

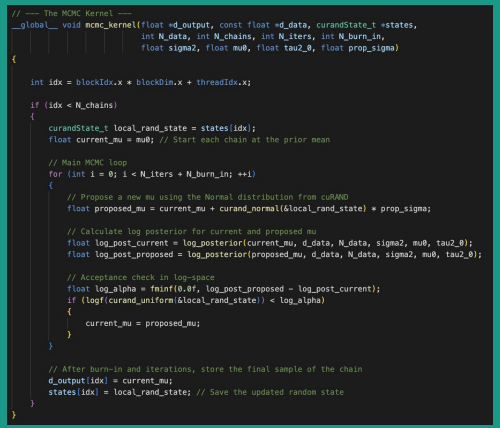


```
// --- 1. Host-side Setup ---
int N = 1024 * 1024;
size_t size = N * sizeof(float);

// Allocate host memory for vectors A, B, and C
float *h_A = (float *)malloc(size);
float *h_B = (float *)malloc(size);
float *h_C = (float *)malloc(size);

// Initialize host vectors
for (int i = 0; i < N; ++i)
{
    h_A[i] = 1.0f;
    h_B[i] = 2.0f;
}

// --- 2. Device-side Memory Allocation ---
// Declare device pointers
float *d_A, *d_B, *d_C;
// Allocate memory on the GPU for each vector
cudaMalloc(&d_A, size);
cudaMalloc(&d_B, size);
cudaMalloc(&d_C, size);

// --- 3. Copy Input Data from Host to Device ---
printf("Copying input data from Host to Device...\n");
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// --- 4. Launch the Kernel ---
int THREADS_PER_BLOCK = 256;
int BLOCKS_PER_GRID = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

printf("Launching vectorAdd kernel...\n");
vectorAdd<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>(d_A, d_B, d_C, N);

// Block host execution until the device kernel finishes
cudaDeviceSynchronize();
printf("Kernel execution finished.\n");

// --- 5. Copy Result Data from Device to Host ---
printf("Copying result data from Device to Host...\n");
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// --- 6. Verification (on the host) ---
bool success = true;
// Check all elements for correctness
for (int i = 0; i < N; ++i)
{
    if (h_C[i] != 3.0f)
    {
        printf("Error at index %d: Expected 3.0, got %f\n", i, h_C[i]);
        success = false;
        break;
    }
}
if (success)
{
    // Print one of the results to show it worked
    printf("Verification successful! e.g., h_C[100] = %f\n", h_C[100]);
}

// --- 7. Cleanup ---
// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
// Free host memory
free(h_A);
free(h_B);
free(h_C);
```

CUDA Ecosystem



MCMC Sampler
Example CUDA Implementation



```cuda
// --- The MCMC Kernel ---
__global__ void mcmc_kernel(float *d_output, const float *d_data, curandState_t *states,
                            int N_data, int N_chains, int N_iters, int N_burn_in,
                            float sigma2, float mu0, float tau2_0, float prop_sigma)
{

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N_chains)
    {
        curandState_t local_rand_state = states[idx];
        float current_mu = mu0; // Start each chain at the prior mean

        // Main MCMC loop
        for (int i = 0; i < N_iters + N_burn_in; ++i)
        {
            // Propose a new mu using the Normal distribution from cuRAND
            float proposed_mu = current_mu + curand_normal(&local_rand_state) * prop_sigma;

            // Calculate log posterior for current and proposed mu
            float log_post_current = log_posterior(current_mu, d_data, N_data, sigma2, mu0, tau2_0);
            float log_post_proposed = log_posterior(proposed_mu, d_data, N_data, sigma2, mu0, tau2_0);

            // Acceptance check in log-space
            float log_alpha = fminf(0.0f, log_post_proposed - log_post_current);
            if (logf(curand_uniform(&local_rand_state)) < log_alpha)
            {
                current_mu = proposed_mu;
            }
        }

        // After burn-in and iterations, store the final sample of the chain
        d_output[idx] = current_mu;
        states[idx] = local_rand_state; // Save the updated random state
    }
}
```

(Log) Posterior Device Function

- __device__ functions are reusable helpers run only on the device
- Use of log-space for **numerical stability** (*Bayes' Theorem* product → sum)
- **Log-Prior:** μ assumed to be from Normal distribution
- **Log-Likelihood:** Sums log-probabilities of the data given μ

Main MCMC kernel

- `__global__` function that is run independently across all parallel threads
- Each GPU thread is assigned **one MCMC chain**
- Each chain maintains **own RNG state**
- Metropolis-Hastings runs for burn-in + N iterations:
  - Generates new μ proposal
  - Calculates log-posterior for current and proposed μ
  - Uses those log-posteriors to calculate acceptance probability α
  - Accepts or rejects new proposal
- Only final sample of the chain is stored

Host-Side Orchestration

- **Problem Setup:** *16,386* chains, *2000* iterations, *500* burn-in
- **Generated Data**: *1000* data points, true $\mu = 10.0$, known $\sigma^2 = 4.0$
- **Weak Prior:** $\mu_0 = 0.0$, known $\tau^2 = 100.0$ *(high variance)*
- Launches 65 blocks of 256 threads each…
- Standard CUDA program **data pipeline** followed
- Derives **posterior mean** from retrieved samples

<2 seconds

When compared to a (*vectorized*) implementation on **Intel Xeon CPU,** running on an **NVIDIA A100 GPU** is almost **instantaneous**!

Conclusions

- Deliberate algorithmic co-design essential in taking advantage of GPU
- Data-parallel tasks fit for high-throughput design
- Wide range of computationally intensive methods in stats well-suited:
  - Large-scale linear algebra operations
  - Resampling techniques
  - Likelihood calculations
  - Distance computations
  - *and more…*
- Thesis repo: https://github.com/andbamp/stats-gpu

Computational power is not external to statistics!