Monte Carlo Methods

Johannes Gäßler

What are Monte Carlo Methods?

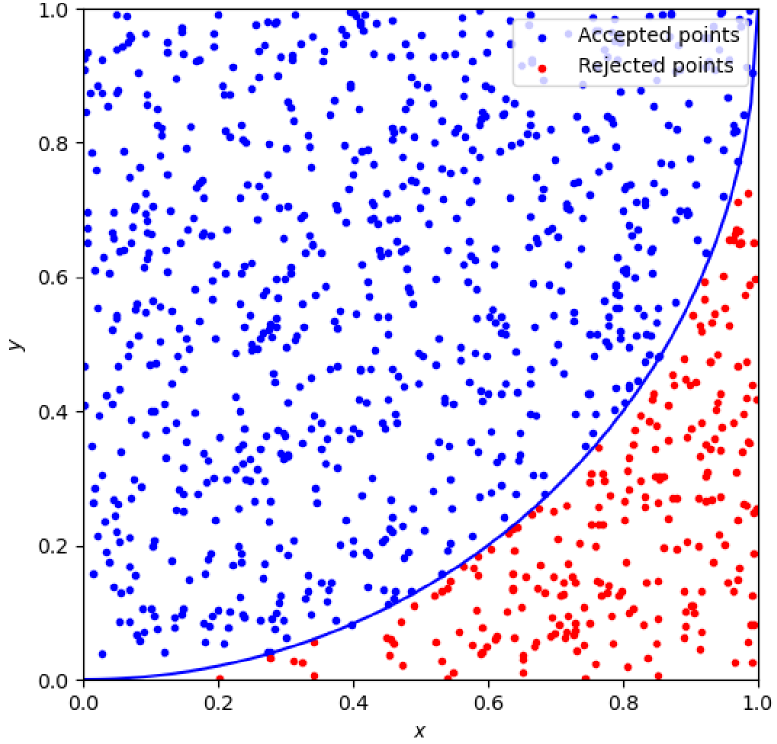
- Use random sampling for numerical result
- Can be used for both stochastic and deterministic problems
- Precision depends on number of events: $O(N^{-\frac{1}{2}})$
- “Random” numbers from random number generator

Hit-and-Miss Monte Carlo

- Determine random points
- Count points fulfilling criterion
- Accepted points follow binomial distribution
- Example: estimation of π
- Criterion: $x^2 + (y-1)^2 < 1$

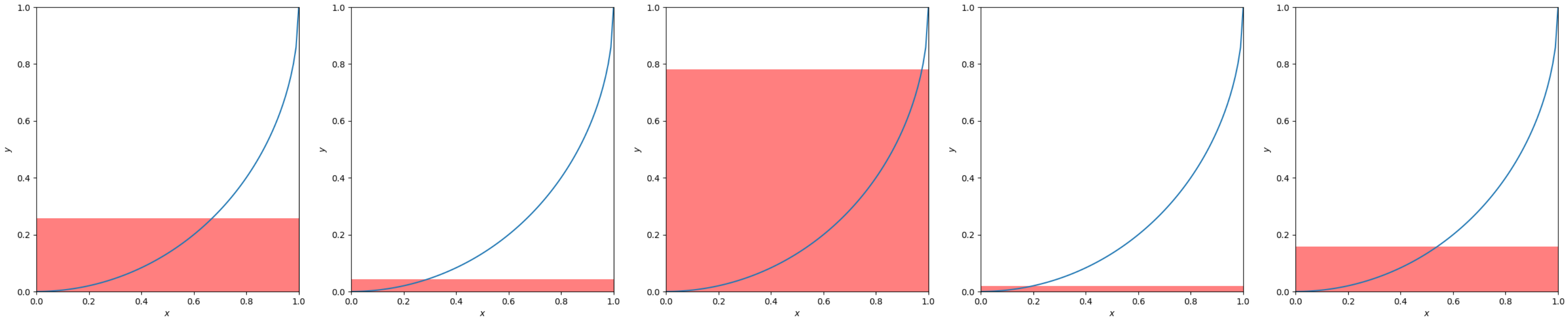
$$\pi = 4\frac{N_{\text{Acc}}}{N}, \quad s_\pi(N) = 4\sqrt{\frac{pq}{N}}$$

$$s_\pi(1000) = 1.7\%$$



Crude Monte Carlo

- Express y as function of x: $y = f(x) = 1 - \sqrt{1 - x^2}$
- Randomly sample x_i
- Estimate integral by averaging $f(x_i)$: $I = \frac{1}{N} \sum_{i=1}^{N} f(x_i)$



Crude Monte Carlo

- Calculate π from integral: $\pi = 4(1 - I)$
- Uncertainty depends on function variance $V[f(x)]$:

$$s_I(N) = \sqrt{\frac{V[f(x)]}{N}}, \quad V[f(x)] = E\left[(f(x) - E[f(x)])^2\right]$$

$$s_\pi(1000) = 0.9\%$$

- Integral converges faster for flat functions

Importance Sampling

- Variance reduction technique
- Sample x with non-uniform PDF $g(x)$
- Regions with high $g(x)$ sampled more frequently
- Scale $f(x)$ with $1/g(x)$ to compensate
- Function variance then becomes:

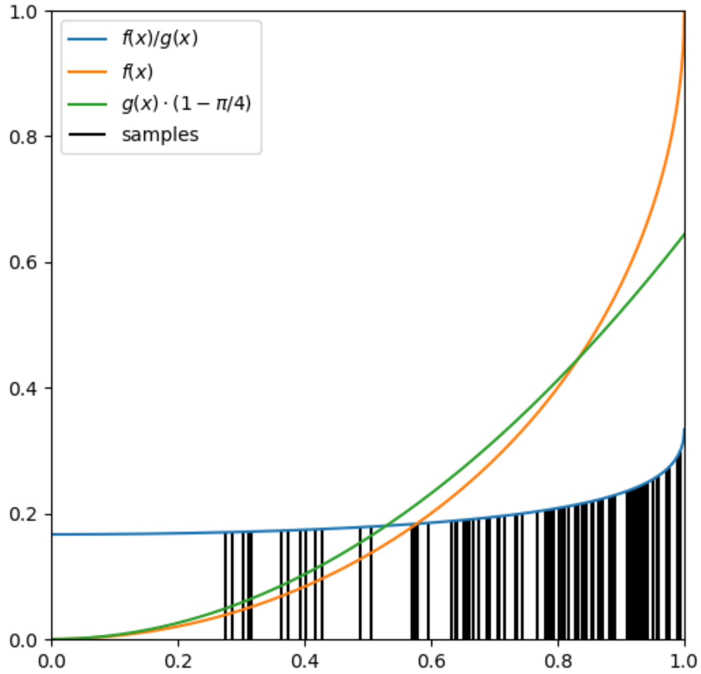
$$V\left[\frac{f(x)}{g(x)}\right]$$

Importance Sampling

- Reduce variance by choosing $\dfrac{f(x)}{g(x)} \approx \text{const}$
- Example: $g(x) = 3x^2$

$$s_I(N) = \sqrt{\frac{V\left[f(x)/g(x)\right]}{N}}$$

$$s_\pi(1000) = 0.1\%$$



VEGAS Algorithm

- Automatic importance sampling
- Split x into M equally-sized intervals:

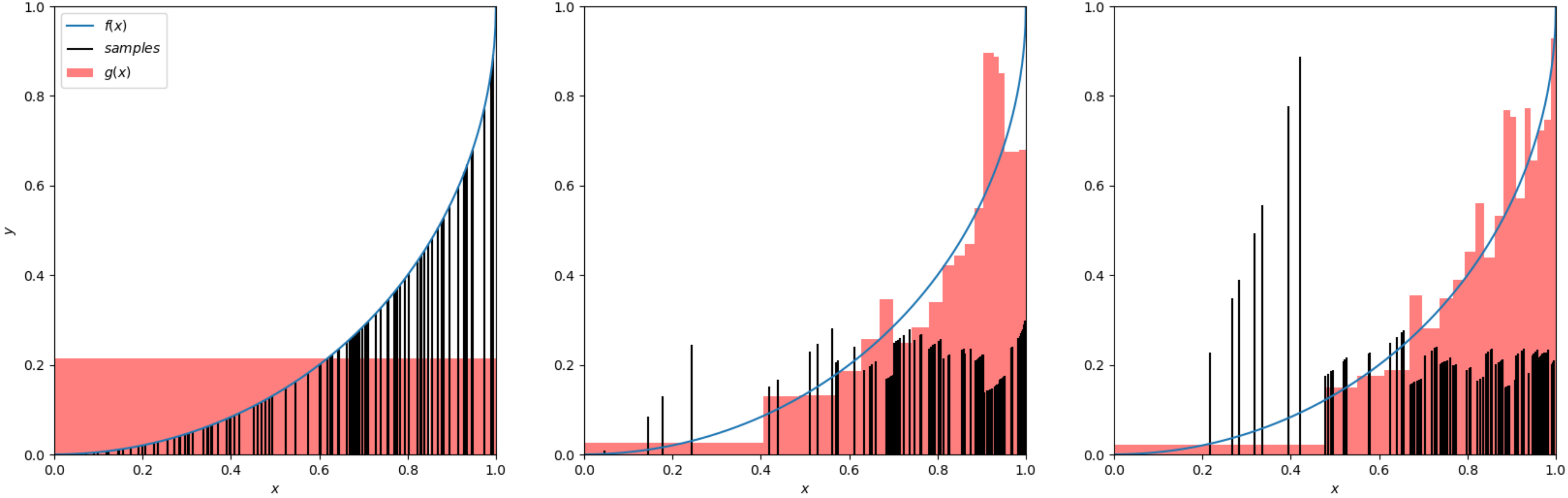
$$0 = x_1 < x_2 < \ldots < x_{M+1} = 1, \quad \Delta x_i = x_{i+1} - x_i$$

- Define step function as PDF to sample from:

$$g(x) = \frac{1}{M \Delta x_i}$$

- Adapt $g(x)$ to $f(x)$ by iteratively adjusting x_i

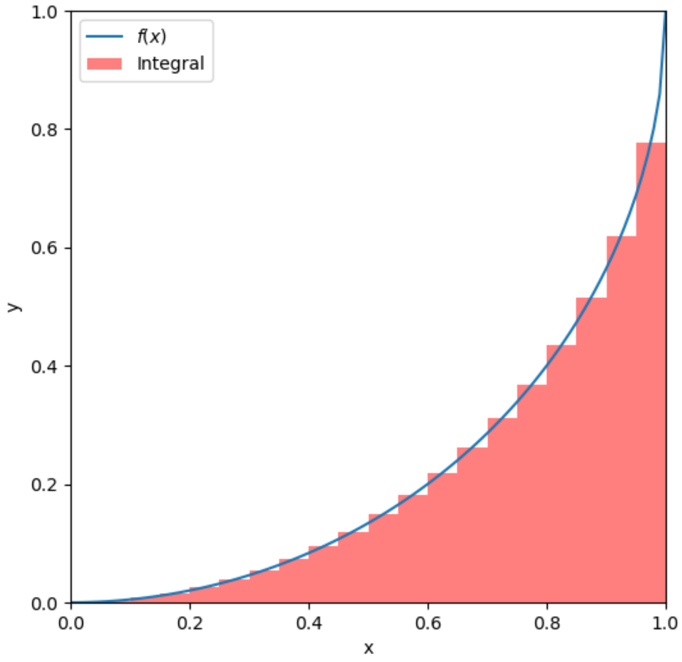
VEGAS Algorithm



$$s_\pi(1000) = 0.5\%$$

MC vs. Quadrature

- For $N \to \infty$ MC and integration are equivalent
- Compare with simple Riemann sum:
-
- $$I = \sum_{i=1}^{N} f\left(\frac{i - 1/2}{N}\right)$$
-
- Precision for $N = 1000$: 0.0003%



MC vs. Quadrature

- Quadrature precision depends on dimension d
- MC precision does not depend on d

Method	Precision (N points)
Monte Carlo	$O(N^{-\frac{1}{2}})$
Trapezoid rule	$O(N^{-\frac{2}{d}})$
Simpson rule	$O(N^{-\frac{4}{d}})$
Gauss rule (mth order)	$O(N^{-\frac{2m-1}{d}})$

MC Use Cases

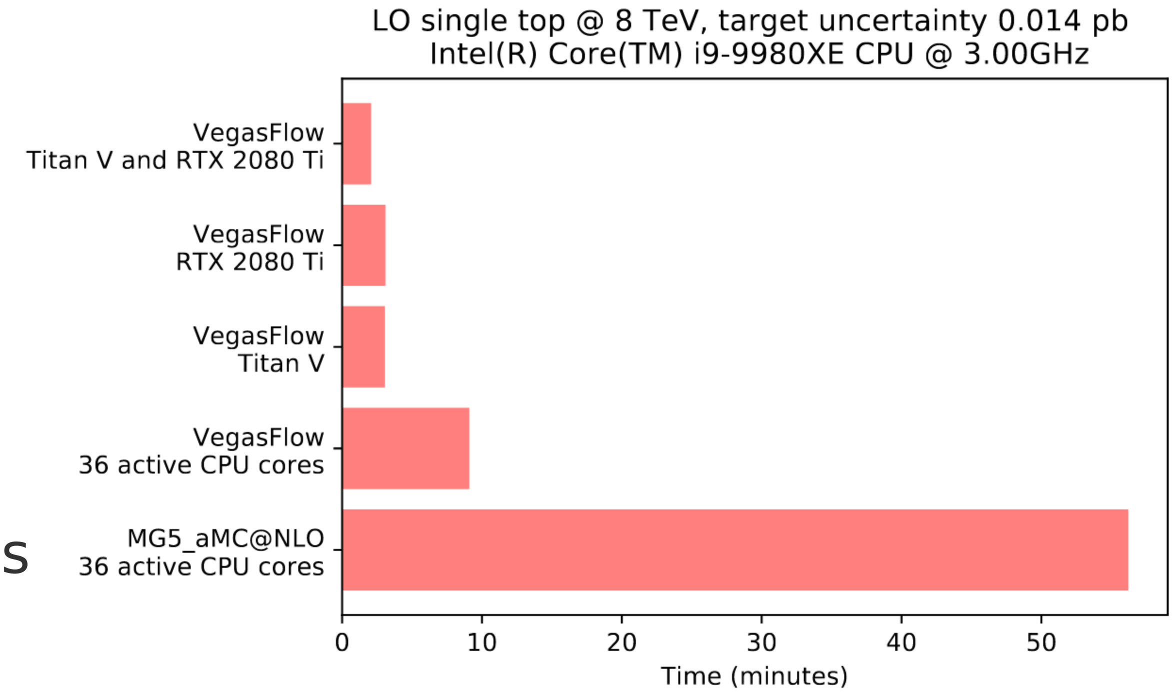
- MC suitable for problems with many coupled degrees of freedom (event generators, galaxy evolution, weather forecasts, ...)
- Need large amount of computation time
- Parallelization essential

Flynn’s Taxonomy

- Computers execute instruction streams on data streams
- Four computer architectures
- Single instruction, single data: CPU core
- Single instruction, multiple data: GPU
- Multiple instruction, single data: Space Shuttle
- Multiple instruction, multiple data: multi-core CPU

SIMD vs. MIMD

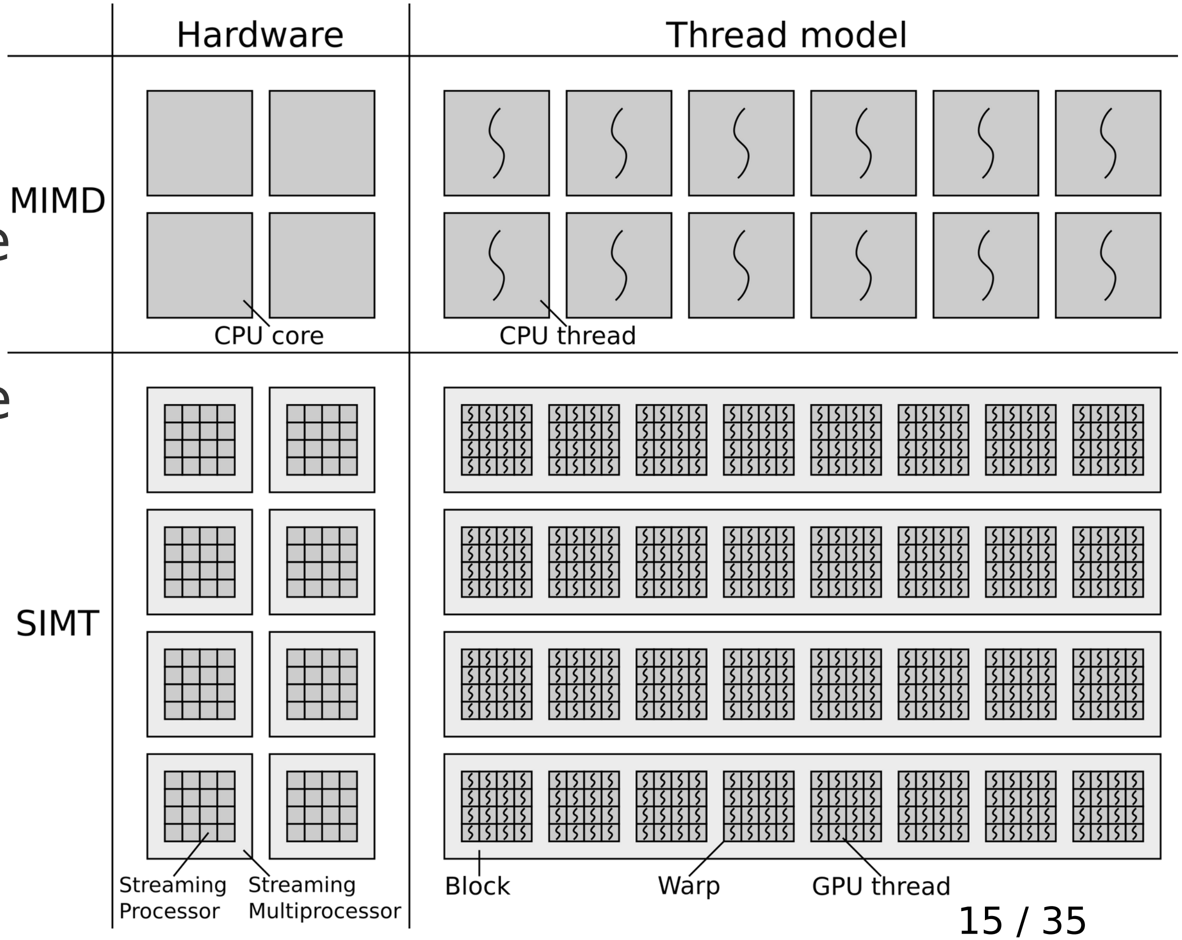
- Computing cost roughly proportional to die area
- Instruction streams and data streams both need die area
- Fewer instruction streams per data stream is more efficient
- SIMD therefore more efficient than MIMD



Stefano Carrazza, Juan M. Cruz-Martinez, 2020, VegasFlow: Accelerating Monte Carlo simulation across multiple hardware platforms

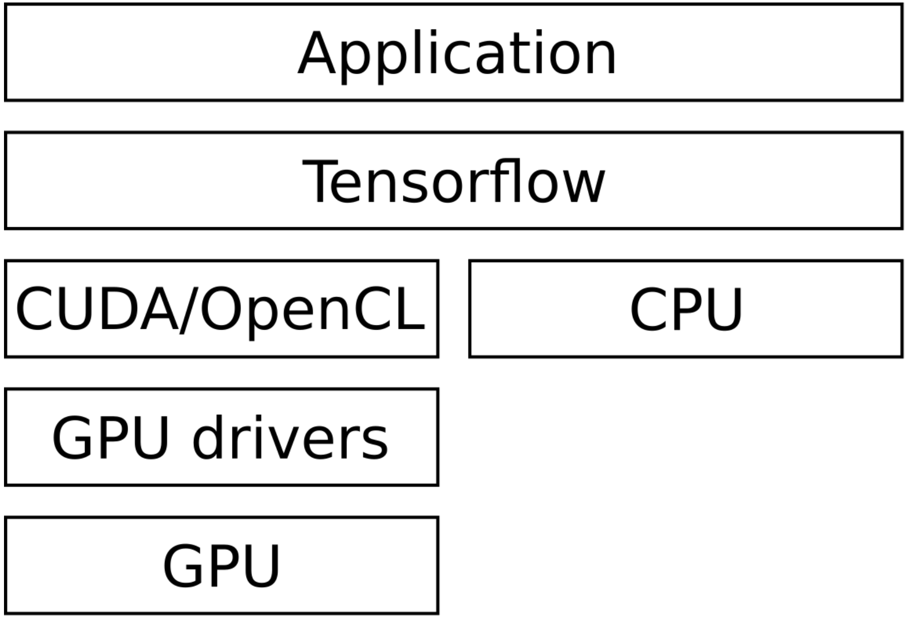
Single Instruction, Multiple Thread

- Single instruction, multiple thread: subtype of SIMD
- Multiple threads execute same instructions on different data at the same time
- GPUs are SIMT



GPU Computing Frameworks

- No framework: formulate problem as graphic primitives (e.g. triangles) and analyze framebuffer
- Low-level framework: explicitly run code on CPU/GPU
- High-level framework: perform high-level operations on vectorized data structures
- Additional layer of abstraction

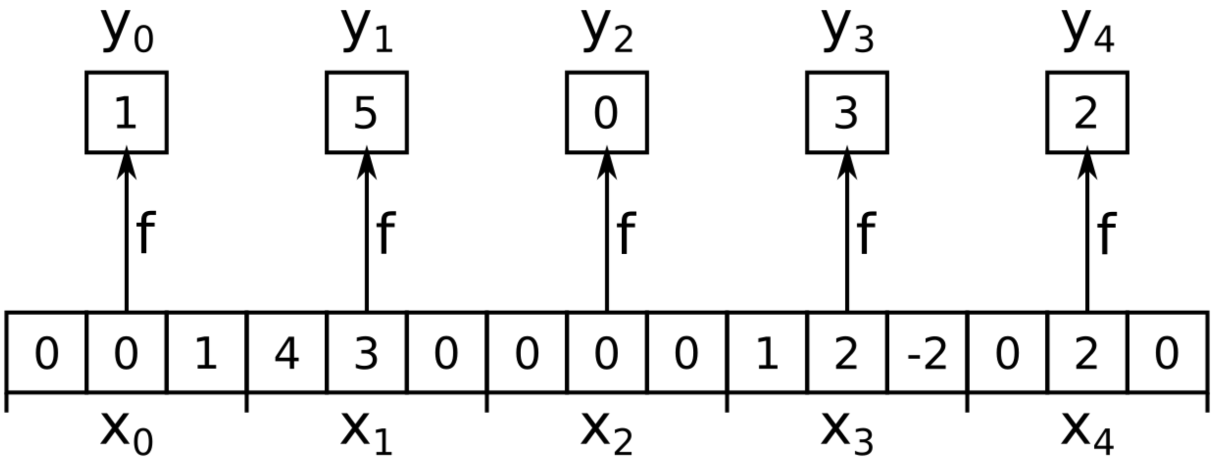


Operation: Map

- Transform array with function
- Transformation does not depend on other elements
- Example: calculate length of 3-vectors

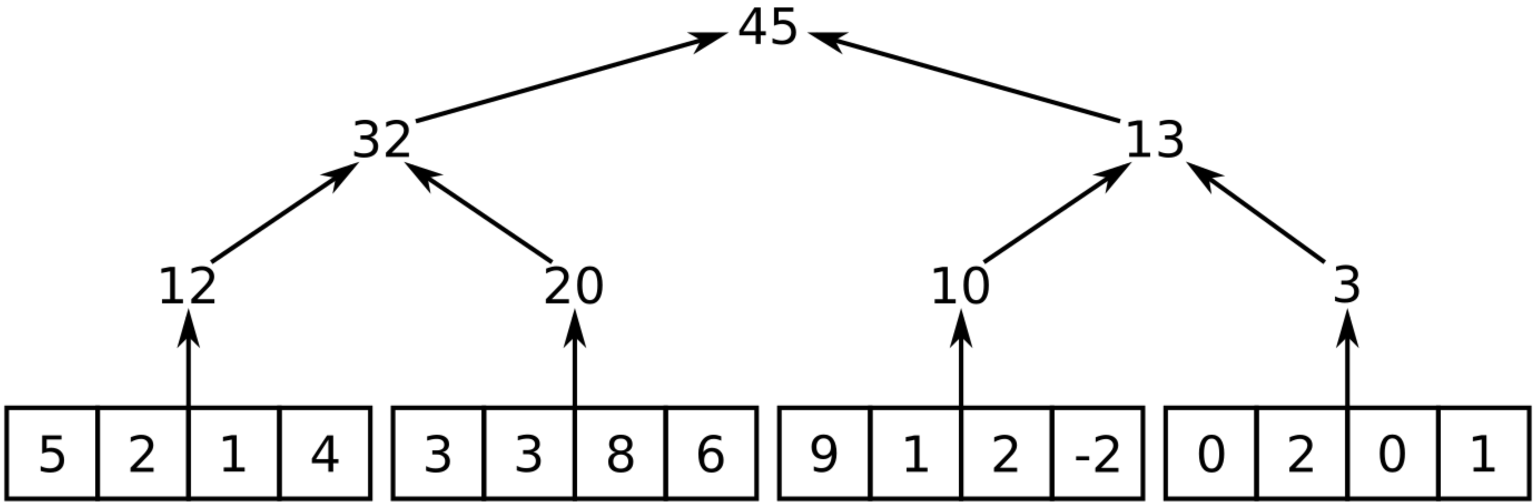
$$y_n = f(x_n) : t_x \to t_y$$

$$f(x_n) = \sqrt{x_{n,0}^2 + x_{n,1}^2 + x_{n,2}^2}$$



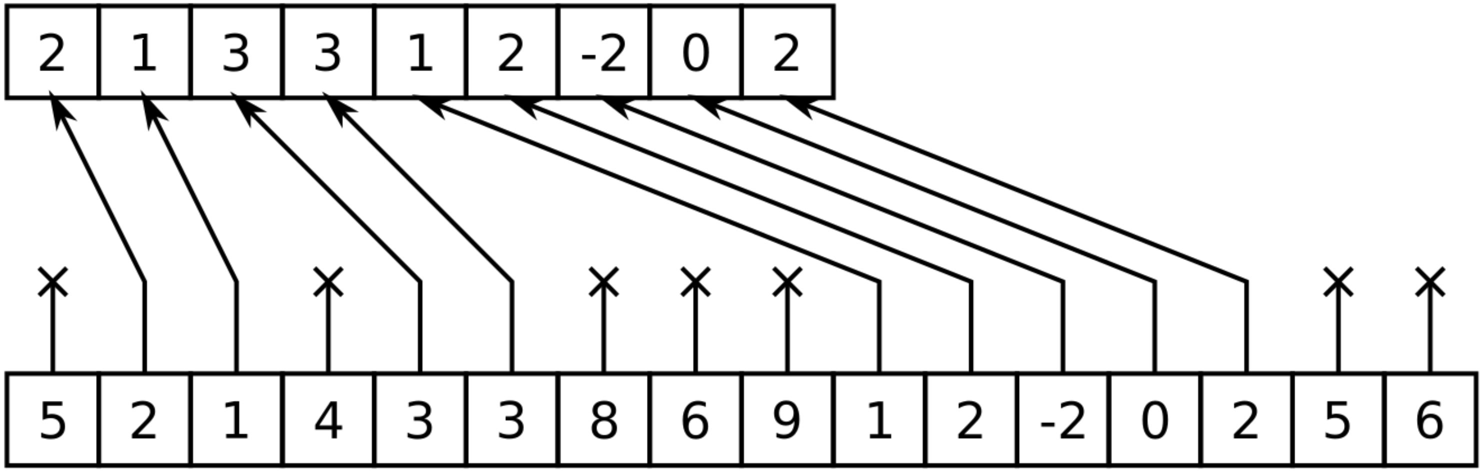
Operation: Reduce

- Collapse array into single value with associative operator $\oplus$
- Calculate partial results, then collapse partial results
- Example: summing up integers $\oplus = +$



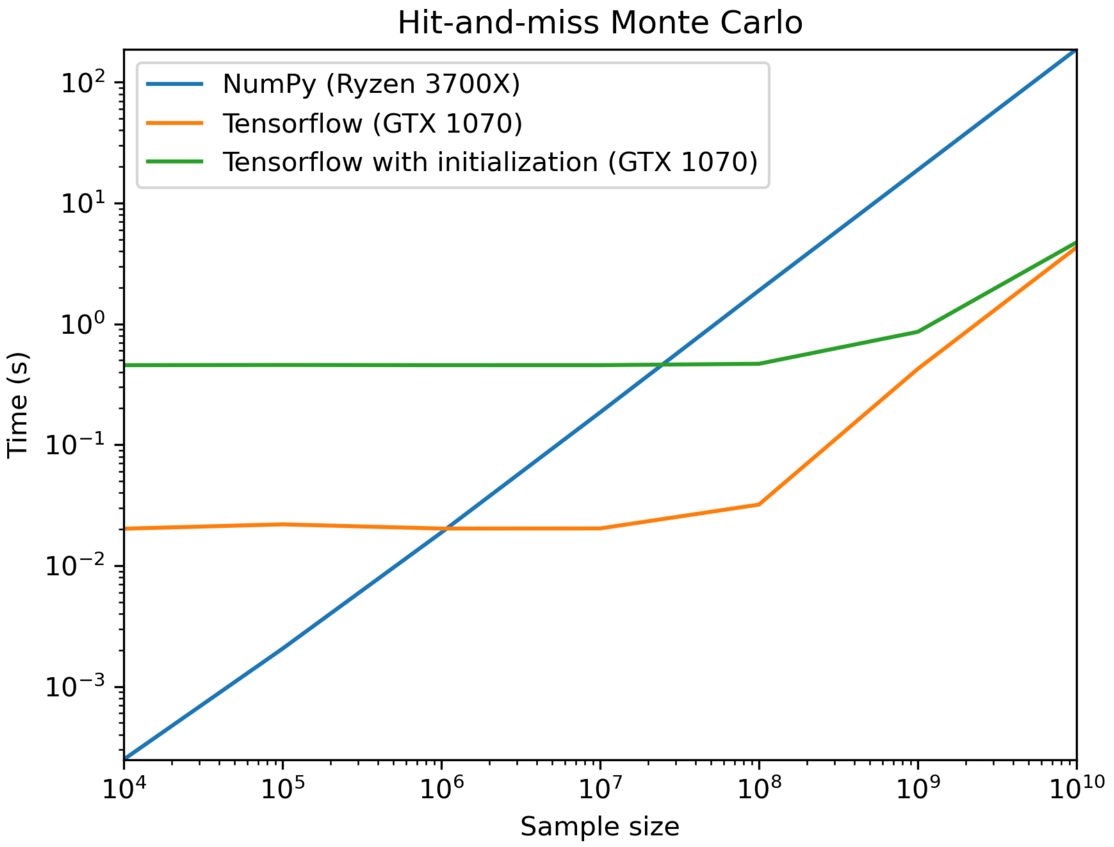
Operation: Filter

- Function that maps array values to booleans: $f : t_x \to \mathbf{bool}$
- Only keep values where $f(x) = \text{True}$
- Example: $x < 4$



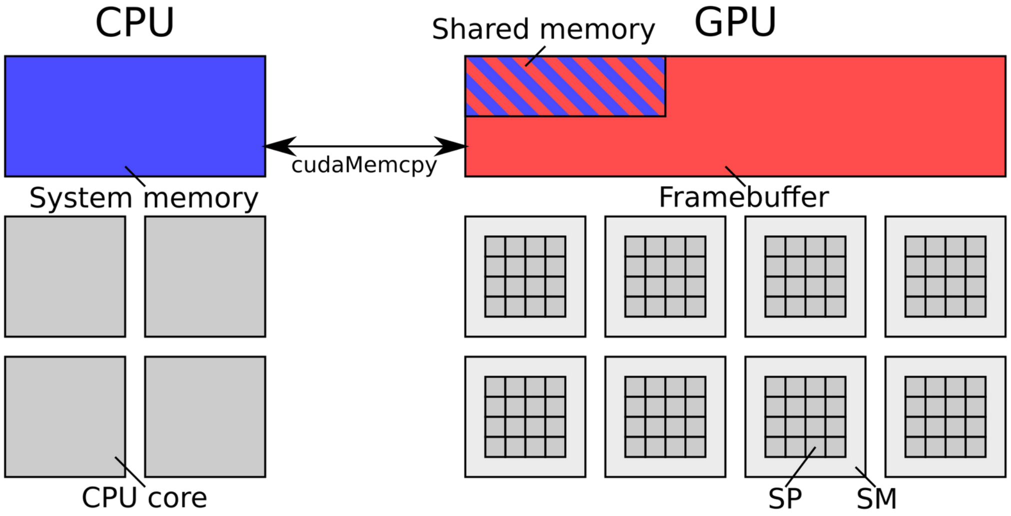
Hit-and-Miss MC with Tensorflow

```
@tf.function
def mc_tf(sample_size):
    rand_xy = tf.random.uniform((sample_size, 2))

    # Map:
    in_circle = tf.square(rand_xy[:, 0]) \
        + tf.square(rand_xy[:, 1]) < 1.0

    # Reduce:
    mc_estimate = tf.math.count_nonzero(
        in_circle) / sample_size

    return 4 * mc_estimate
```



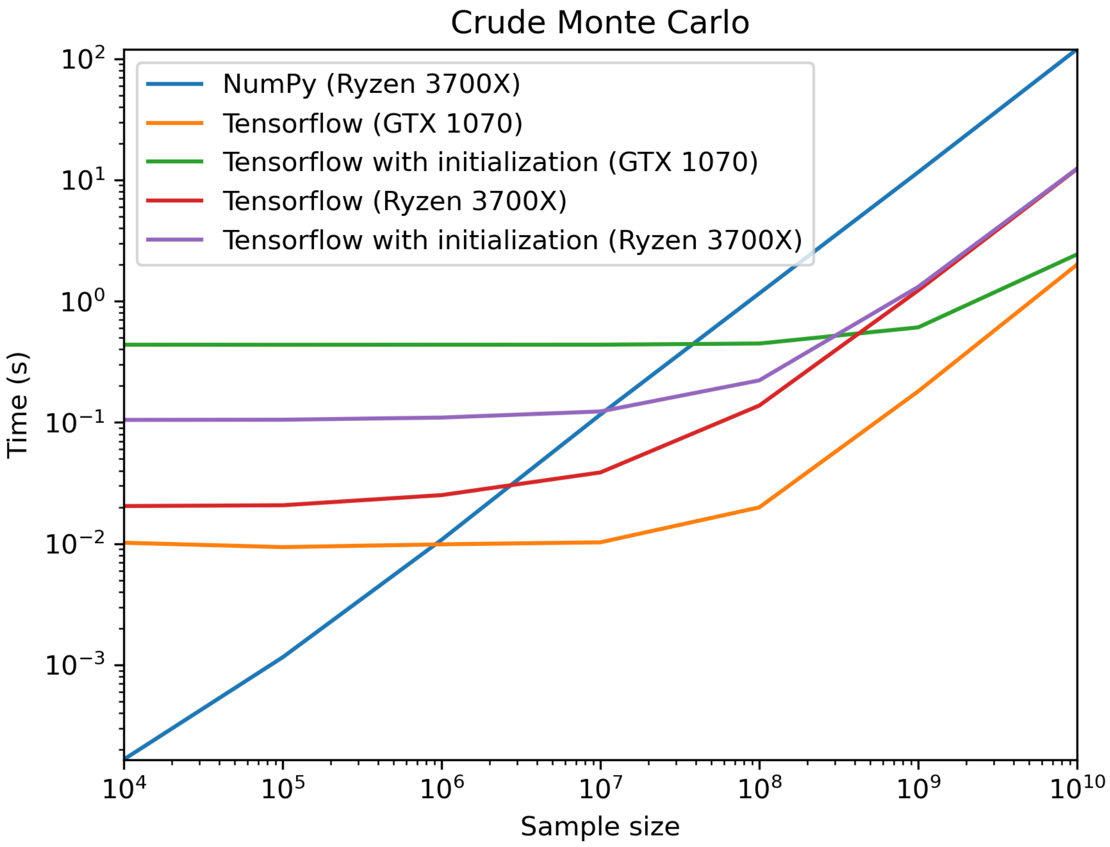
Memory Management

- GPU and CPU use separate memory
- Transfer with special functions
- Introduces latency
- Resizable bar: shared memory on framebuffer



Crude MC with Tensorflow

```
@tf.function
def mc_tf(sample_size):
    rand_x = tf.random.uniform((sample_size,))

    # Map:
    function_values = 1.0 – tf.math.sqrt(
        1.0 - tf.math.square(rand_x))

    # Reduce:
    mc_estimate = tf.reduce_mean(function_values)

    return 4 * mc_estimate
```



Summary

- MC methods solve numerical problems through random sampling
- Suitable for systems with many coupled degrees of freedom
- Can be efficiently parallelized
- Can be suitable candidate for GPU computing
- Parallelization introduces overhead
- Use design patterns (Map, Reduce, Filter, ...) for efficient parallelized code

Further Reading

- Python code used for plots and calculations: *https://github.com/JohannesGaessler/presentation_mc*
- General MC: *F. James, 1980, Monte Carlo Theory and Practice*
- VEGAS algorithm: *G. Peter Lepage, 1977, A New Algorithm for Adaptive Multidimensional Integration*
- VegasFlow: *Stefano Carrazza, Juan M. Cruz-Martinez, 2020, VegasFlow: Accelerating Monte Carlo simulation across multiple hardware platforms*

Thank you for your attention!

Appendix

- Generation of pseudo-random numbers
- SIMT programming: conditional statements, context switches, latency hiding
- VegasFlow parallelization details (annotated source code)
- SIMT parallelization of artificial neural networks

Appendix: RNG

- MC methods need large amounts of random numbers
- Problem: computers are deterministic
- Randomness from physical sources too slow for MC
- Solution: calculate pseudo-random numbers instead
- Ideally no discernable patterns in pseudo-random sequence

Appendix: RNG

- Linear congruential generator:
  $$X_{n+1} = (aX_n + b) \bmod m$$
- Produces integers
- Needs a seed X_0 to start
- Choosing good values for integers a, b, m is tricky
- Interpret integers as significand to generate float
  $$\text{float} = \text{significand} \cdot \text{base}^{\text{exponent}}$$
- Can be parallelized efficiently with SIMT

Appendix: Conditional Statements

- Evaluate condition for every thread to create mask
- *if* block: part of the threads are idling due to mask
- *else* block: invert mask
- Loops: update mask between iterations
- Idling threads bad for performance

Appendix: Thread Overhead

- Both CPU and GPU have registers/cache for threads
- GPU: registers/cache are **not** flushed on context switch
- GPU threads cheap but few registers available
- Performance optimization very different
- CPU: num threads ~ num cores
- GPU: num threads >> num cores
- Latency hiding: try to always have threads that can be worked on when data needs to be fetched from memory

Appendix: Vegasflow Sampling

```python
# Calculate x values to sample function with.

@tf.function
def digest(xn):
    # xn: random numbers between 0 and BINS_MAX.
    ind_i = tf.cast(xn, DTYPEINT)  # Map, lower bin bounds indices.
    ind_f = ind_i + ione  # Map, upper bin bounds indices.
    x_ini = tf.gather(divisions, ind_i, batch_dims=1)  # Map, lower bin bounds.
    x_fin = tf.gather(divisions, ind_f, batch_dims=1)  # Map, upper bin bounds.
    xdelta = x_fin - x_ini  # Map, bin widths.
    return ind_i, x_ini, xdelta

@tf.function
def compute_x(x_ini, xn, xdelta):
    aux_rand = xn - tf.math.floor(xn)  # Map, new uncorrelated random number.
    return x_ini + xdelta * aux_rand  # Map, final x value.
```

Appendix: Vegasflow Bin Weights

- Calculate new bin weights:

$$w_i = \left[\left(\frac{f_i \Delta x_i}{\sum_j f_j \Delta x_j} - 1 \right) \ln \left(\frac{f_i \Delta x_i}{\sum_j f_j \Delta x_j} \right)^{-1} \right]^{\alpha}$$

- Hyperparameter α dampens changes to bins

```
# smeared_tensor: bin widths times bin heights averaged with left and right neighbors.
sum_t = tf.reduce_sum(smeared_tensor)  # Reduce.
log_t = tf.math.log(smeared_tensor)  # Map.
aux_t = (1.0 - smeared_tensor / sum_t) / (tf.math.log(sum_t) - log_t)  # Map.
wei_t = tf.pow(aux_t, ALPHA)  # Map.
ave_t = tf.reduce_sum(wei_t) / BINS_MAX  # Reduce.
```

Appendix: Vegasflow New Bins

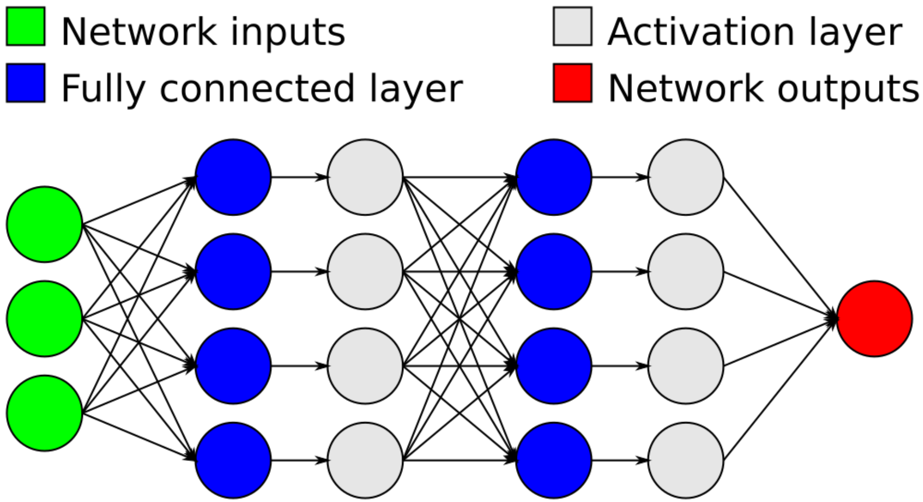
```python
@tf.function
def while_check(bin_weight, *args):
    """True if bins have at least average weight."""
    return bin_weight < ave_t

@tf.function
def while_body(bin_weight, n_bin, cur, prev):
    """Iterate bins and add up their weights."""
    n_bin += 1  # Increase index.
    bin_weight += wei_t[n_bin]
    prev = cur  # Lower bin bound.
    cur = subdivisions[n_bin + 1]  # Upper bin bound.
    return bin_weight, n_bin, cur, prev
```

```python
# Initialize varaibles before loop:
new_bins = [fzero]  # first bin always starts with 0.
bin_weight = fzero  # Running sum of bin weights.
n_bin = -1  # Bin index.
cur = fzero  # Upper bin bound.
prev = fzero  # Lower bin bound.
for _ in range(BINS_MAX - 1):
    bin_weight, n_bin, cur, prev = tf.while_loop(
        while_check,
        while_body,
        (bin_weight, n_bin, cur, prev),
        parallel_iterations=1,  # No parallelism.
    )
    bin_weight -= ave_t  # Decrease bin weight sum.
    # Decrease upper bin bound proportionally
    # to excess bin weight:
    delta = (cur - prev) * bin_weight / wei_t[n_bin]
    new_bins.append(cur - delta)
new_bins.append(fone)  # Last bin always ends with 1.
```

Appendix: ANN SIMT Implementation

- Artifical neural networks are built from layers of “neurons”
- Output of each neuron is linear combination of previous layer
- Insert non-linear activation function between layers

$$x_{i,j} = w_0 + \sum_{n=1}^{N} w_n x_{i-1,n}$$



Appendix: ANN SIMT Implementation

- Use MapReduce for fully connected layers
- Use Map for activation layers with activation function $S(x)$

$$x_{i,j} = w_0 + \sum_{n=1}^{N} w_n x_{i-1,n}$$

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$$