Neural Network on CUDA

Restricted Boltzmann Machine

Daniel Ly
Volodymyr Paprotski
Danny Yen

Outline

- Restricted Boltzmann Machine background
- CUDA Implementation
  - Matrix Library
  - Random Number Generator
  - Sigmoid Function
- Results
- Discussion

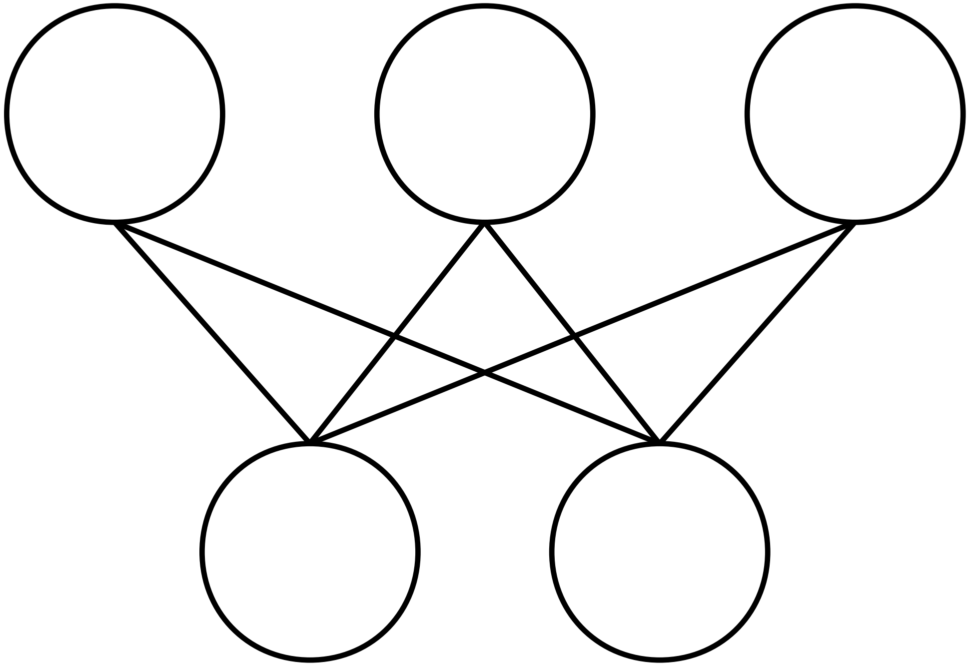
Background

- Neural networks is a computational paradigm that is inspired by biological nervous systems
- Based on the idea of having lots of simple computational units, called nodes, that maps an input to an output
- Interesting behaviour emerges from massive networks of interconnected nodes

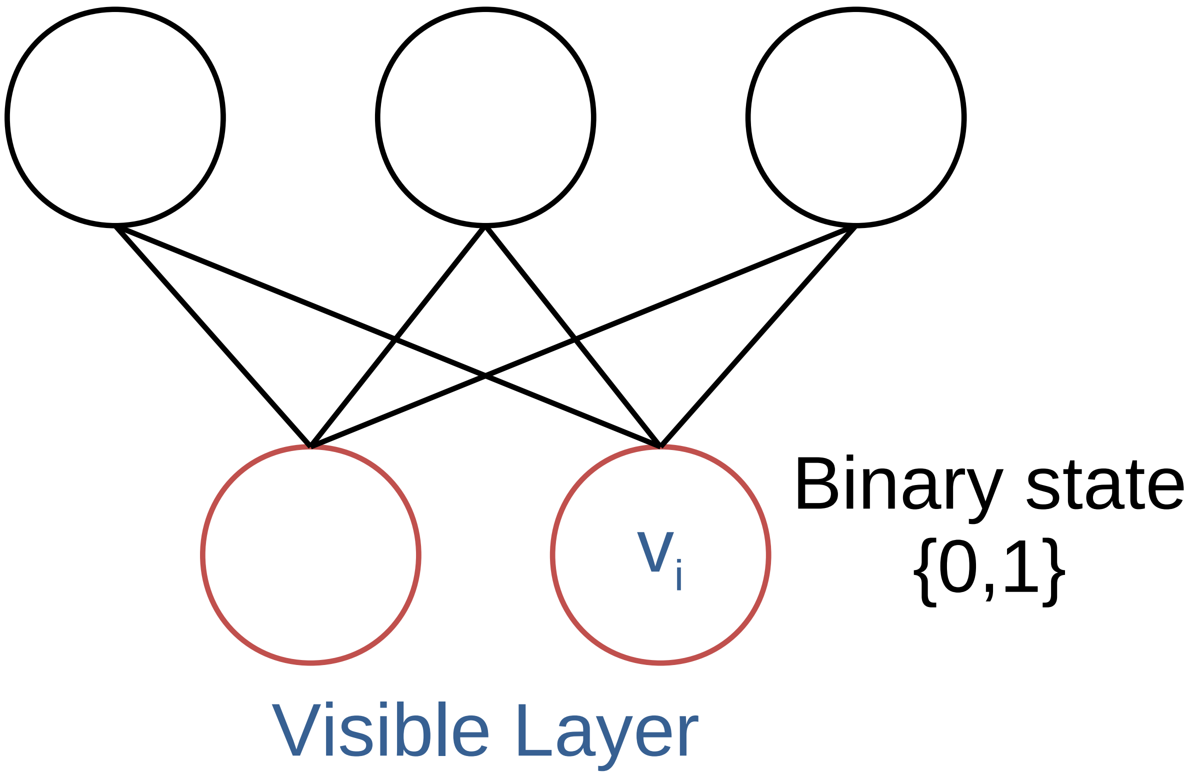
Background

- Artificial neural networks are capable of learning
  - Given a data set, automated learning rules can be applied to achieve a desired behaviour
- Artificial neural networks are applied to a growing number of applications
  - Pattern classification
  - Computer vision
  - Signal processing

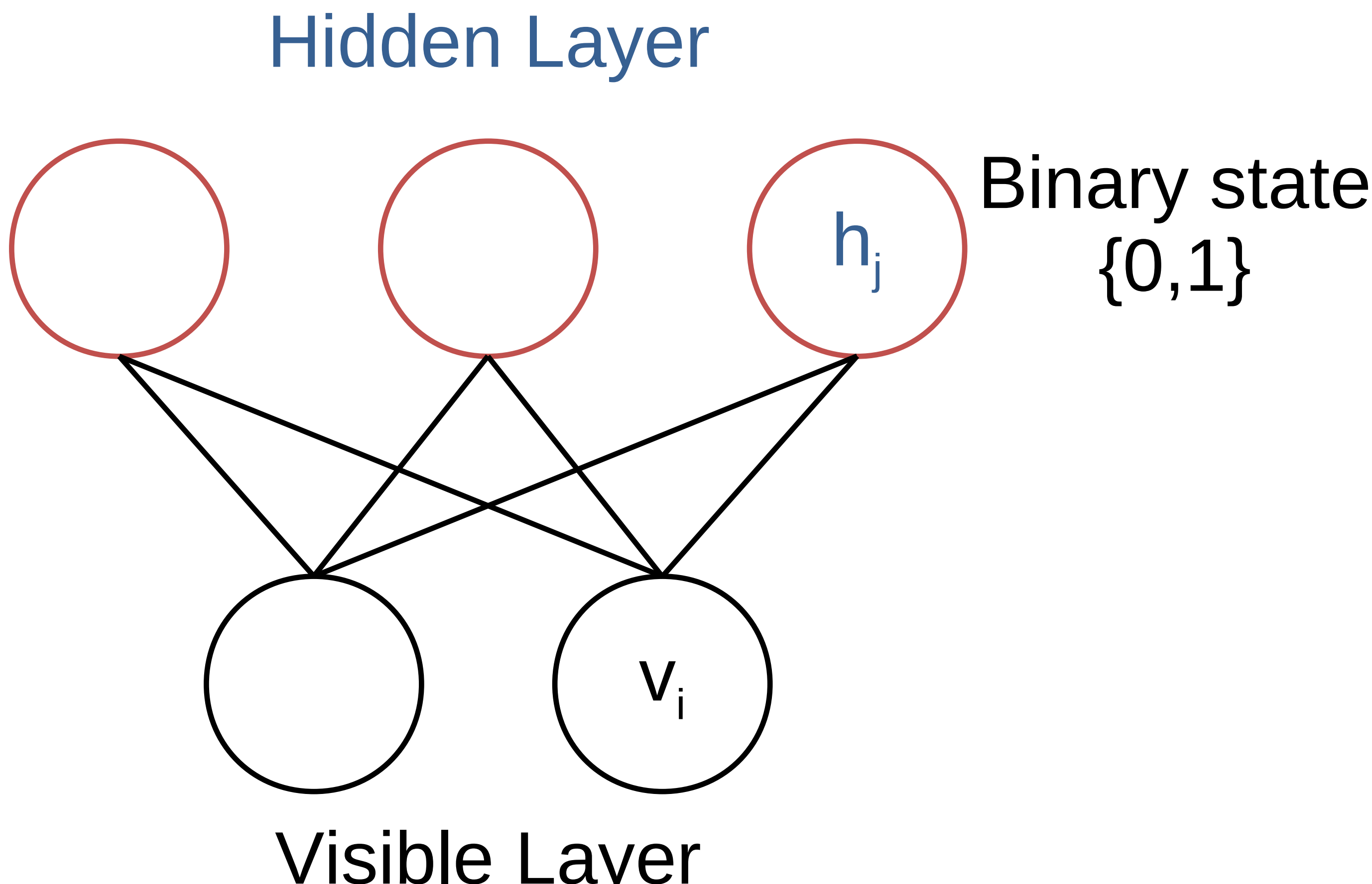
RBM Theory



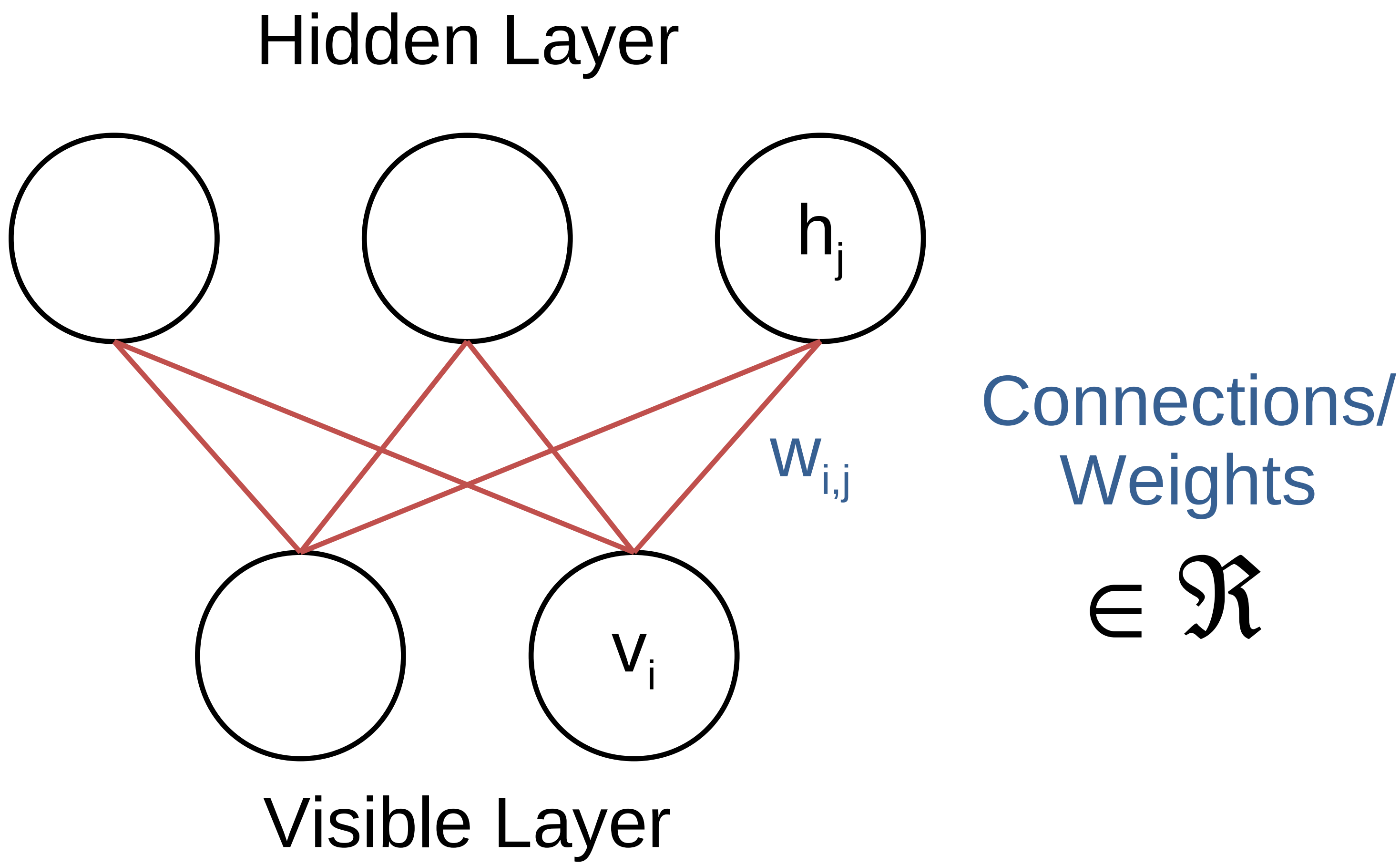
RBM Theory



RBM Theory

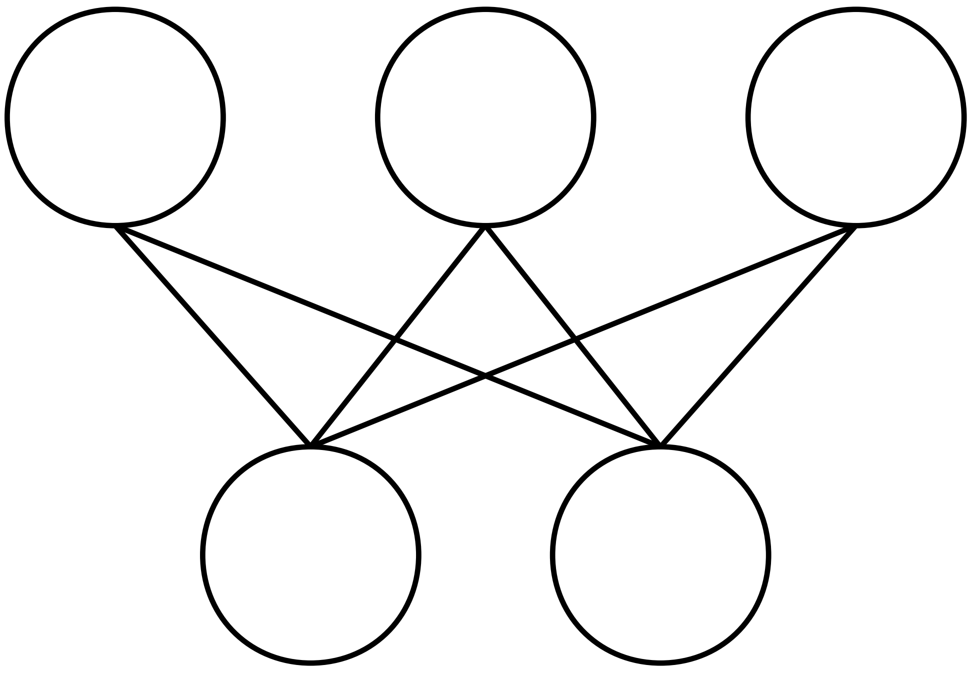


RBM Theory


Hidden Layer
h_j
$w_{i,j}$
Connections/
Weights
$\in \Re$
v_i
Visible Layer

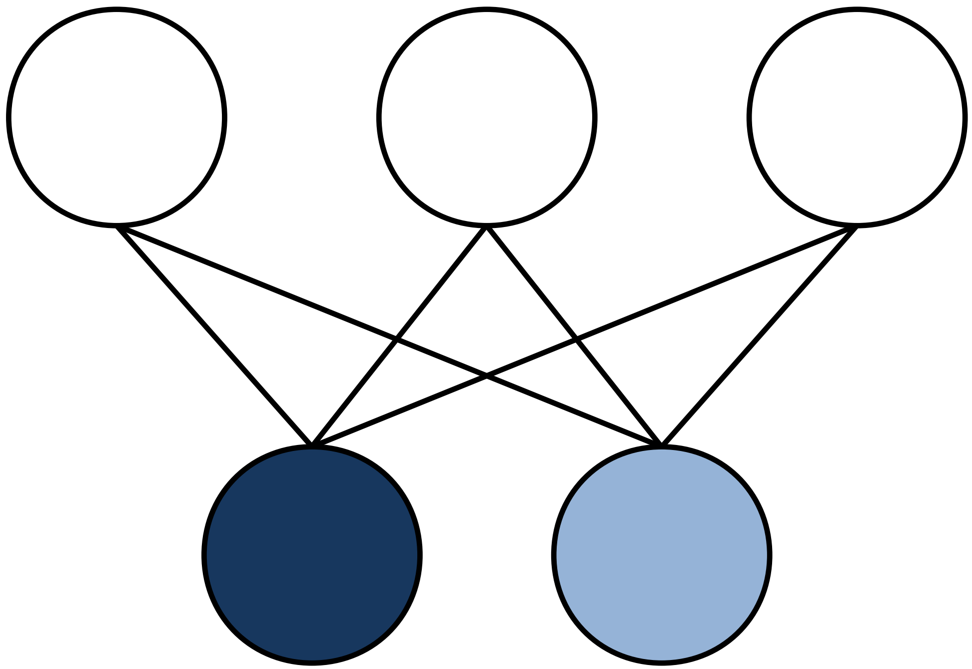
RBM Theory

Alternating Gibbs Sampling



RBM Theory

Alternating Gibbs Sampling

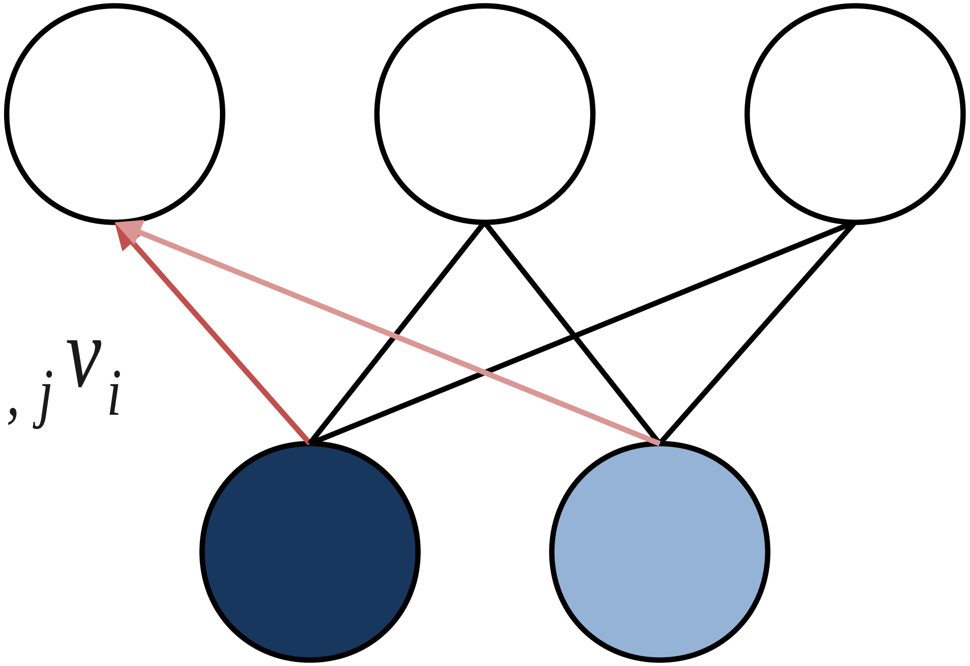


Load data vector in visible layer

RBM Theory

Alternating Gibbs Sampling

$$E_j = \sum_j w_{i,j} v_i$$

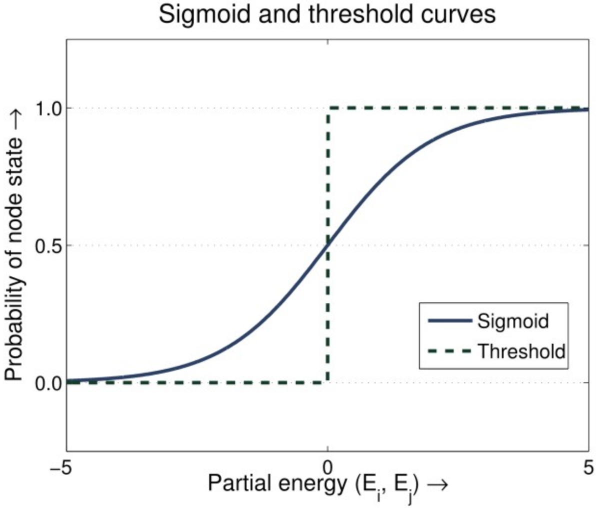


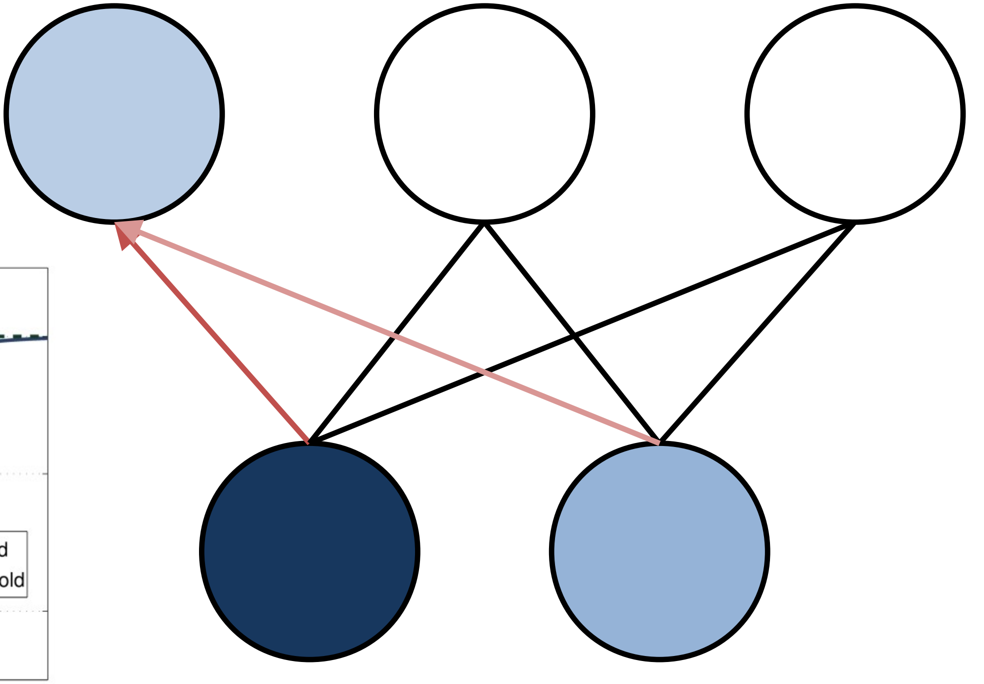
Generate energies

RBM Theory

Alternating Gibbs Sampling

$$h_j = f(E_j)$$

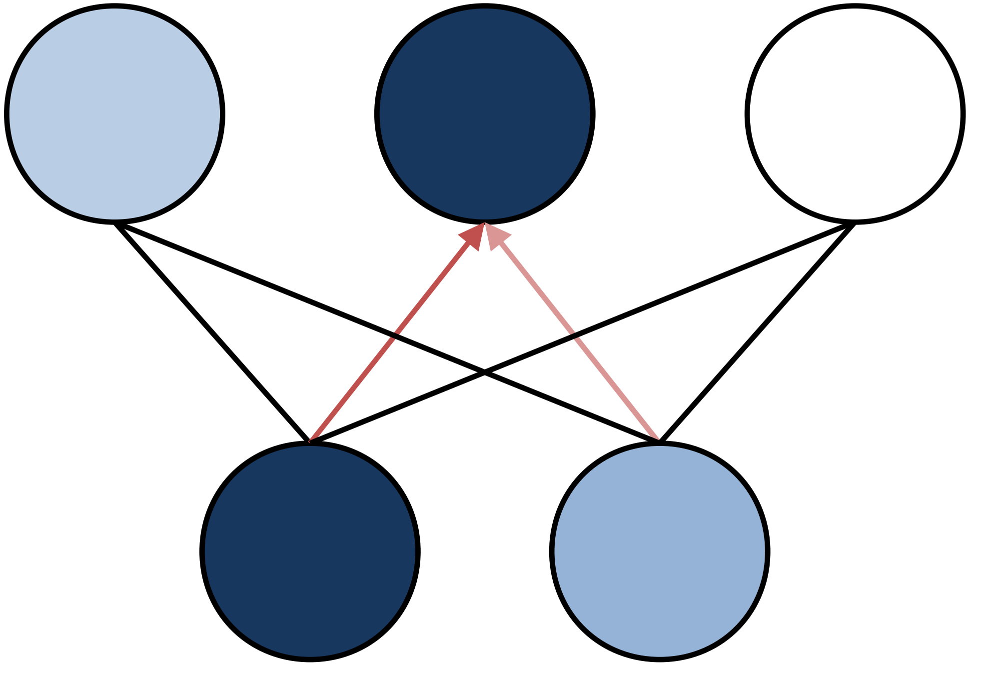




Determine node state through transfer function

RBM Theory

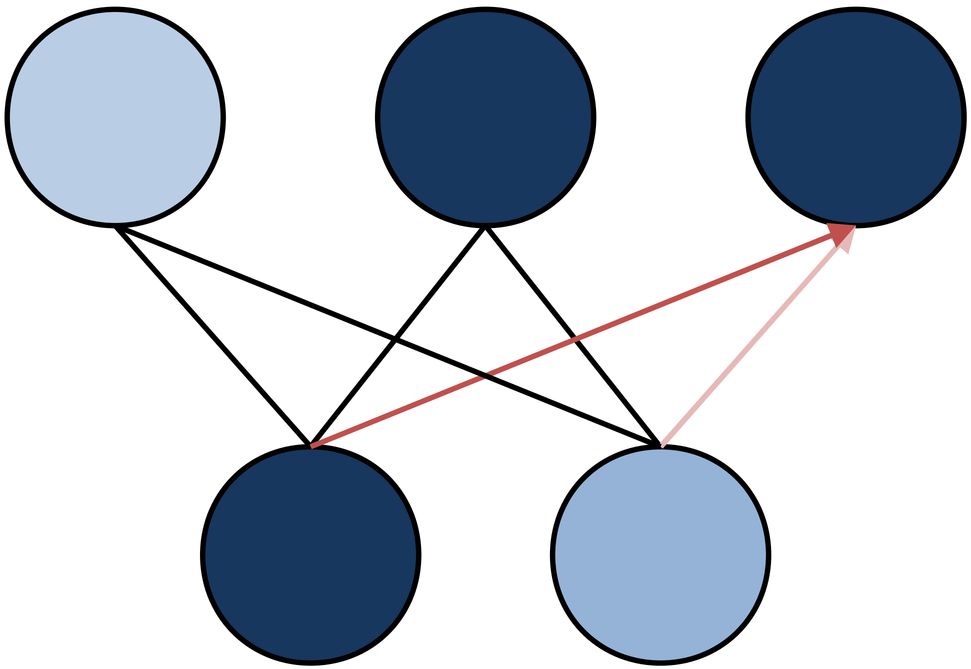
Alternating Gibbs Sampling



Determine node state through transfer function

RBM Theory

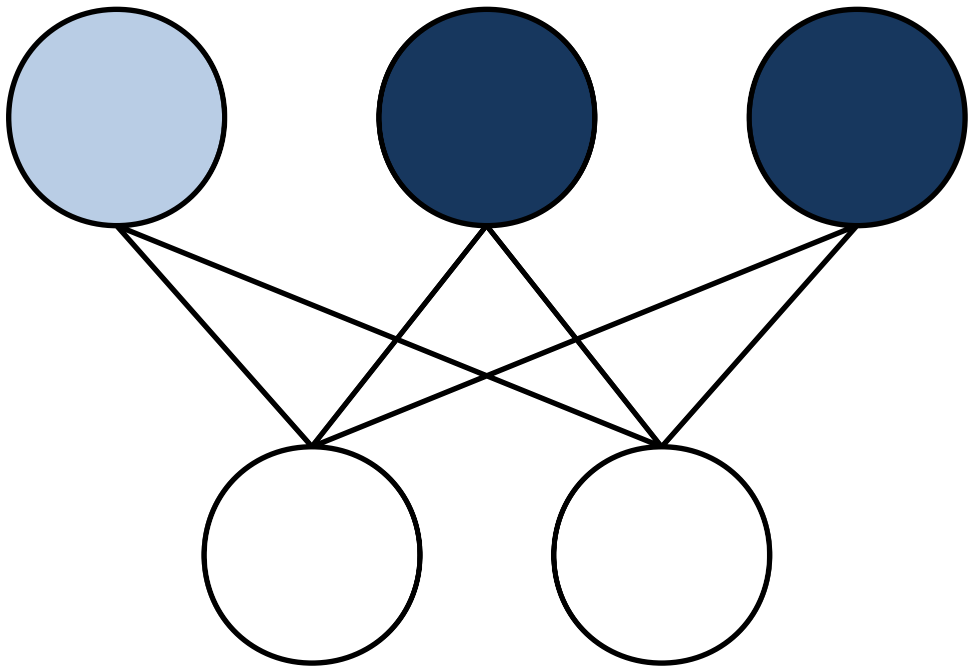
Alternating Gibbs Sampling



Determine node state through transfer function

RBM Theory

Alternating Gibbs Sampling



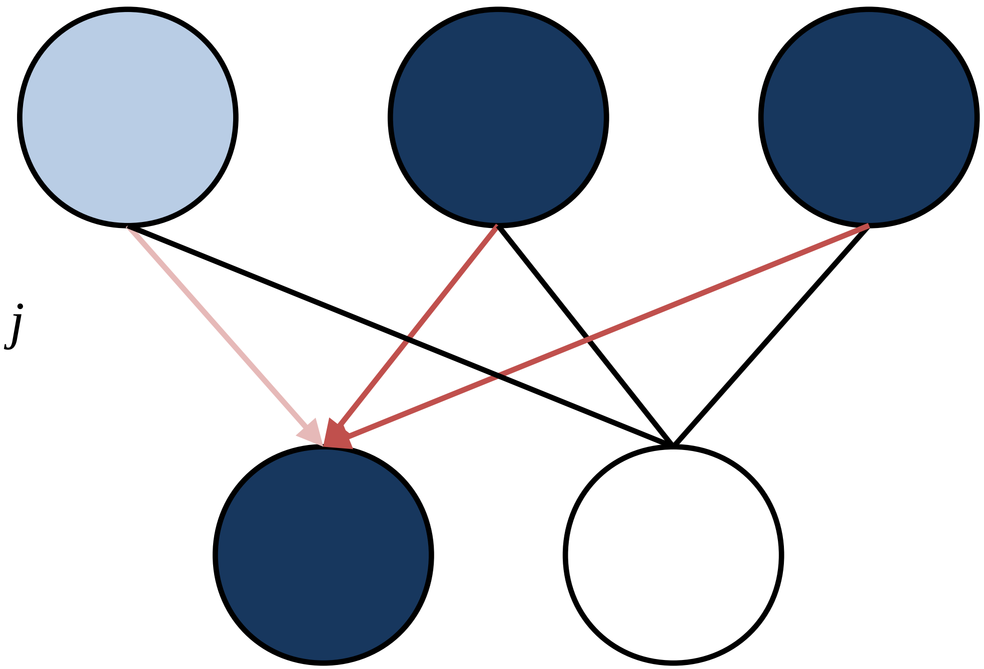
Reconstruct visible nodes

RBM Theory

Alternating Gibbs Sampling

$$E_i = \sum_i w_{i,j} h_j$$

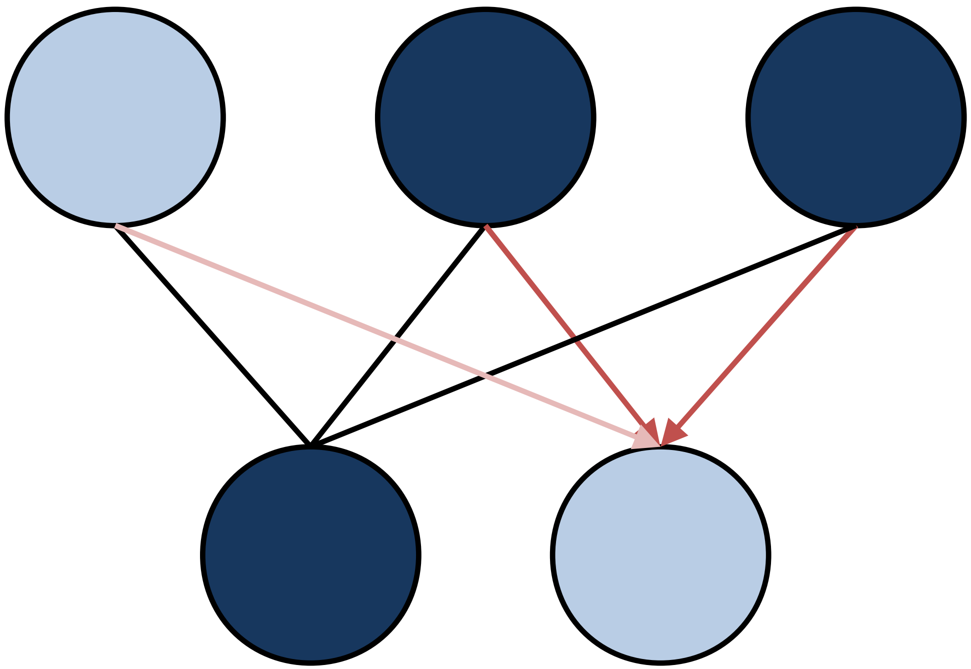
$$v_i = f(E_i)$$



Reconstruct visible nodes

RBM Theory

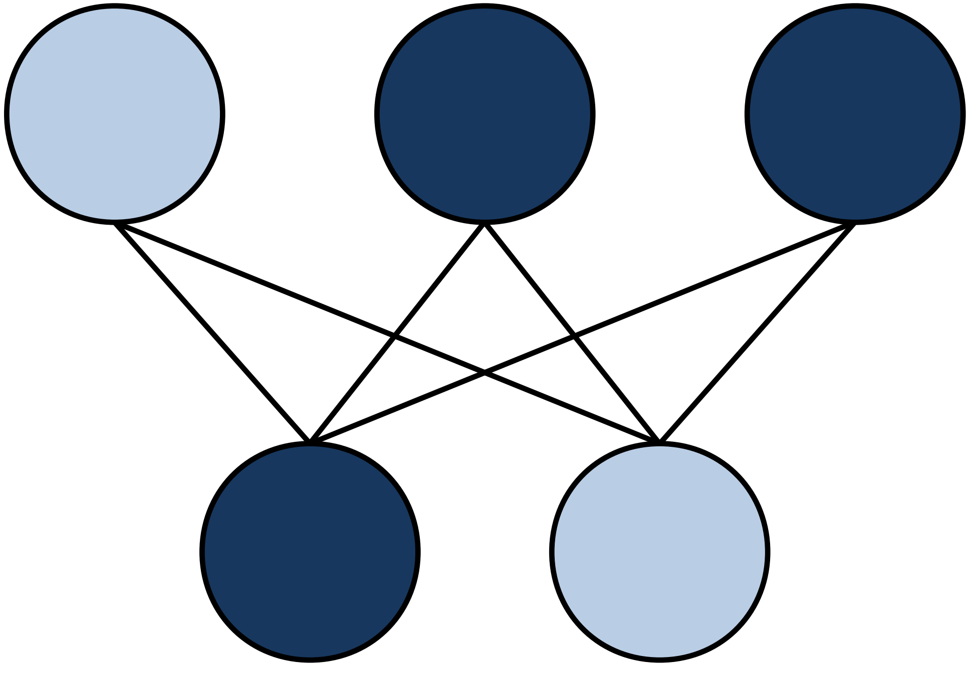
Alternating Gibbs Sampling



Reconstruct visible nodes

RBM Theory

Alternating Gibbs Sampling

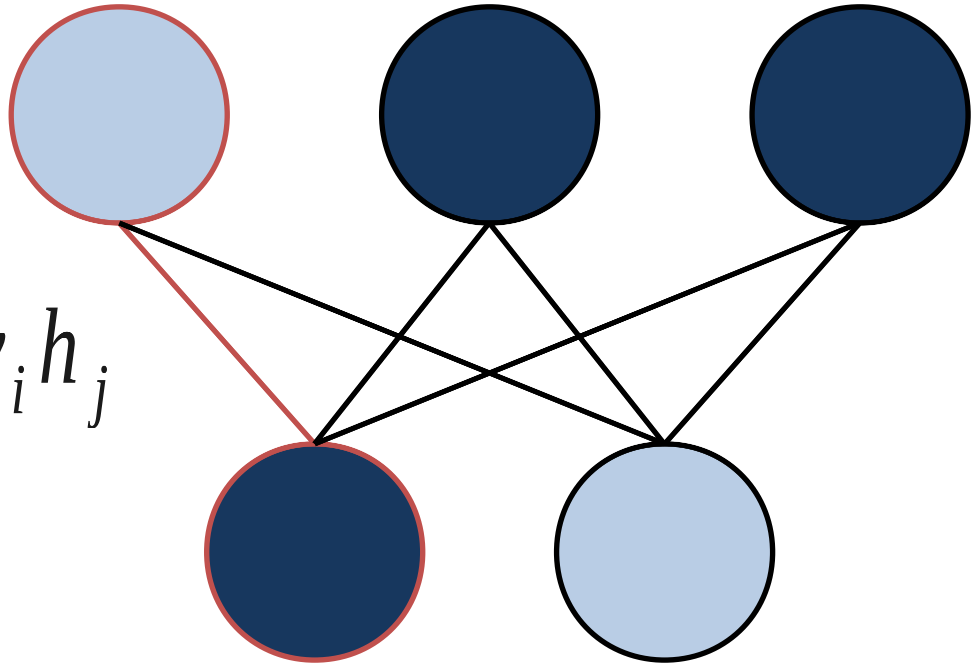


Update weights (simplified)

RBM Theory

Alternating Gibbs Sampling

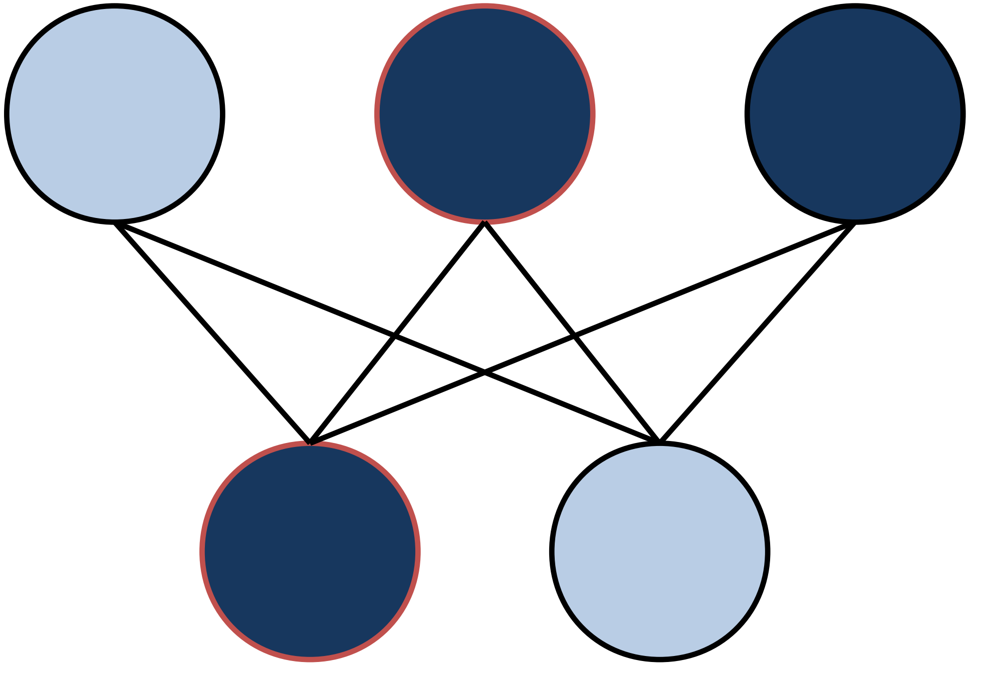
$$w_{i,j} = w_{i,j} \pm \varepsilon v_i h_j$$



Update weights (simplified)

RBM Theory

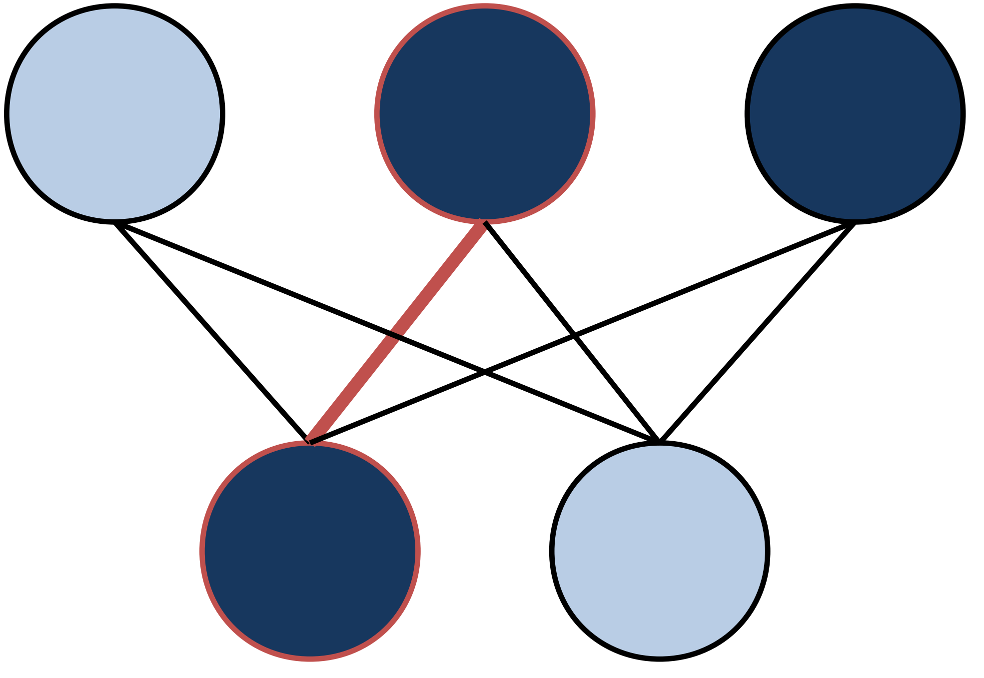
Alternating Gibbs Sampling



Update weights (simplified)

RBM Theory

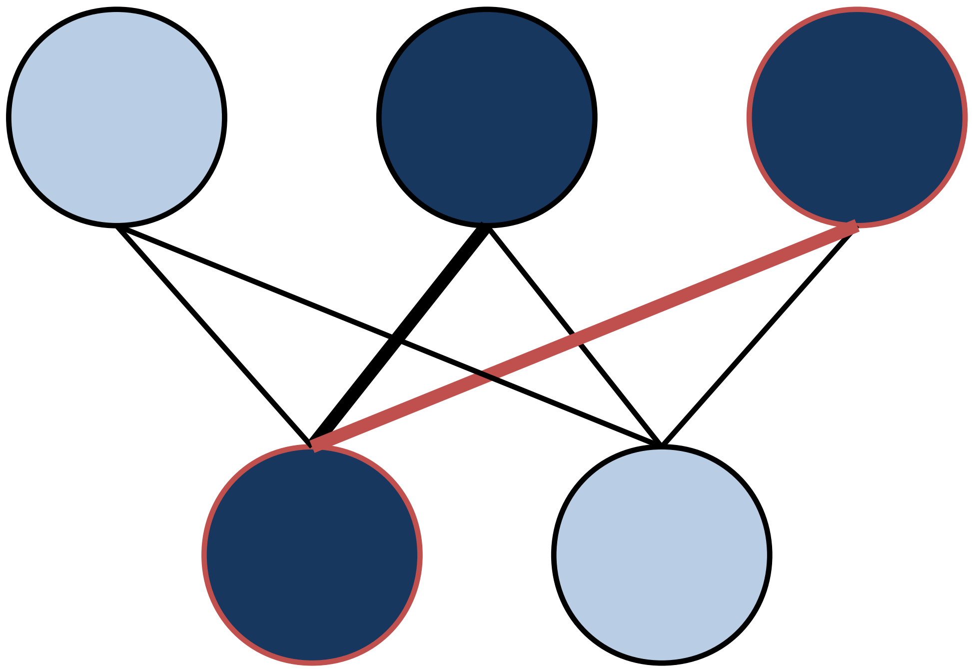
Alternating Gibbs Sampling



Update weights (simplified)

RBM Theory

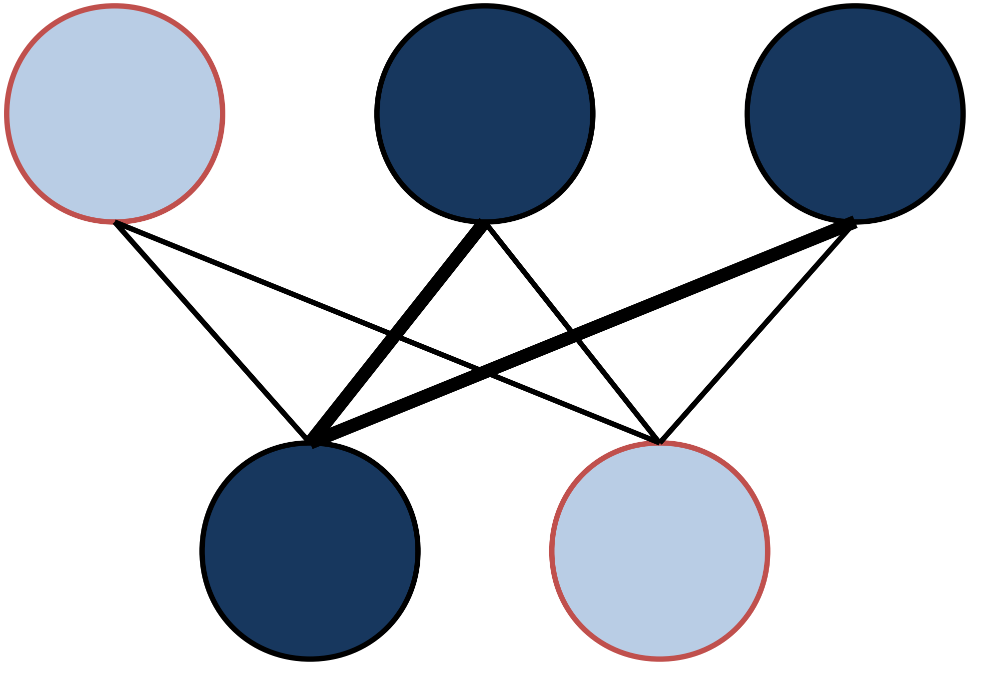
Alternating Gibbs Sampling



Update weights (simplified)

RBM Theory

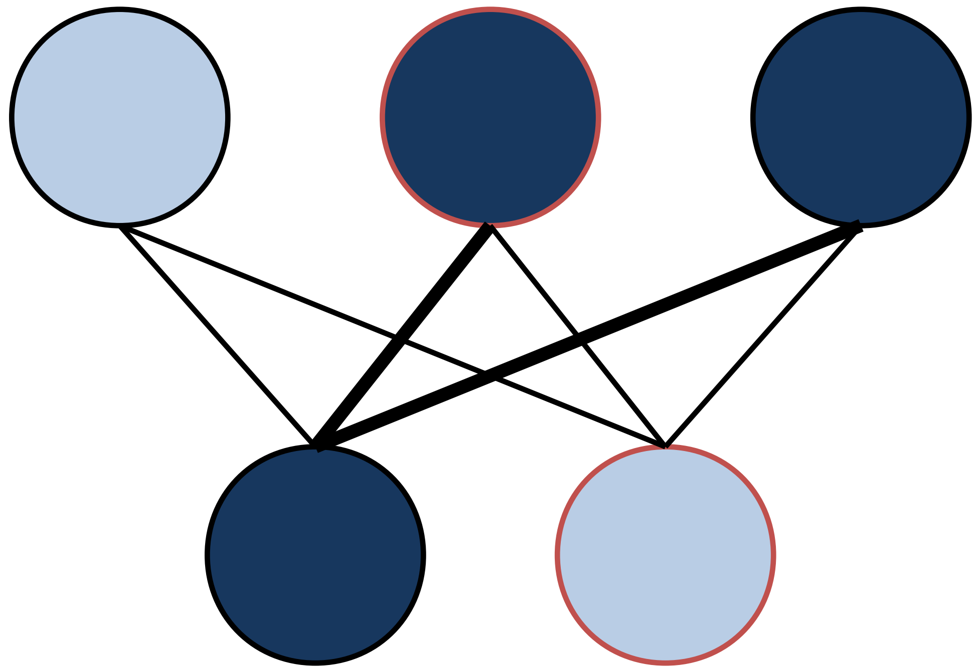
Alternating Gibbs Sampling



Update weights (simplified)

RBM Theory

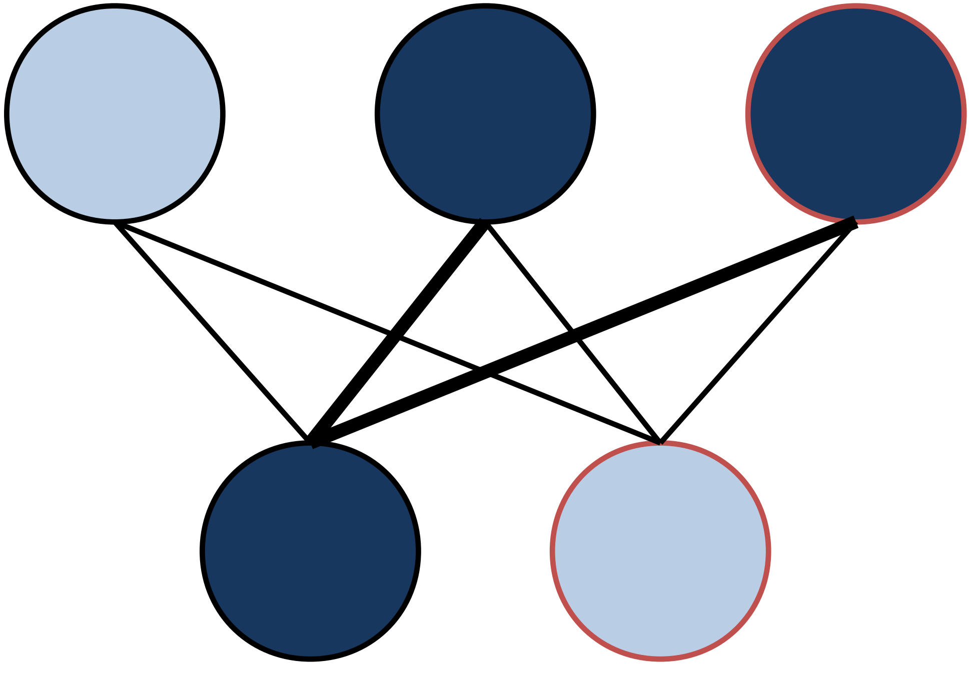
Alternating Gibbs Sampling



Update weights (simplified)

RBM Theory

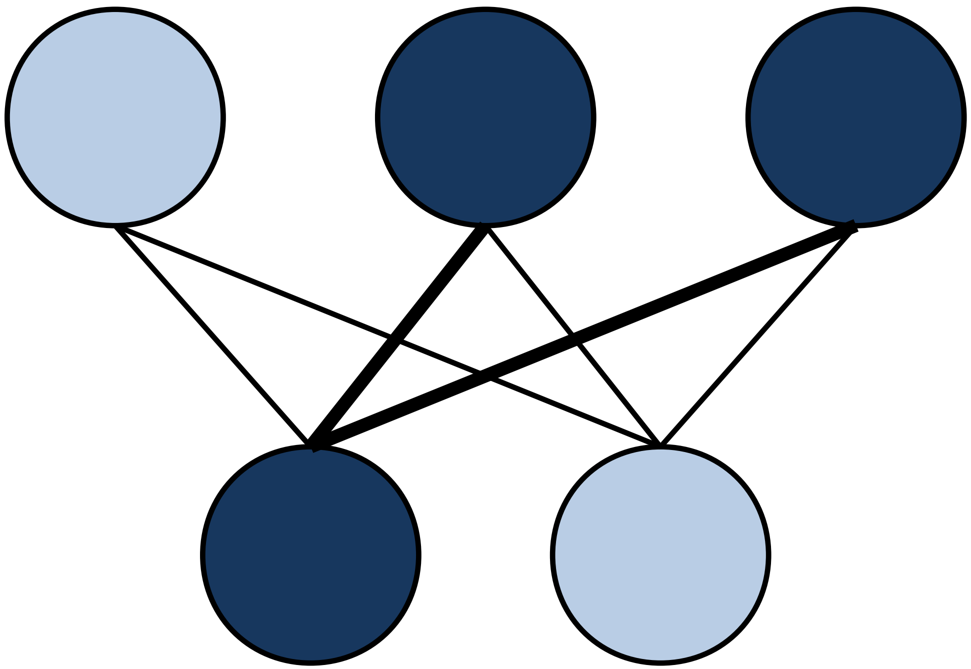
Alternating Gibbs Sampling



Update weights (simplified)

RBM Theory

Alternating Gibbs Sampling



Repeat cycle

CUDA Implementation

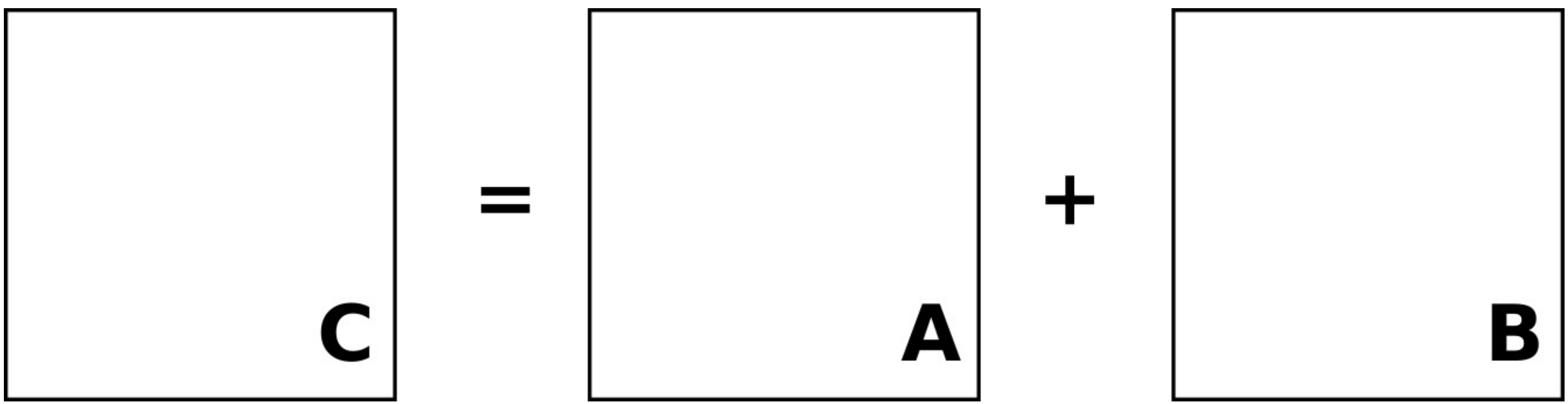
- The project was divided into three kernels
  - Matrix Operations
  - Random Number Generation
  - Sigmoid Function

Matrix Operations

- Node states, energies and weights can be represented as matrices
- Computation is dominated by matrix operations
- Matrix libraries were based on the examples in the SDK
  - Additional optimization was achieved
  - Extra care was used to ensure coalesced memory calls and bank conflicts were avoided

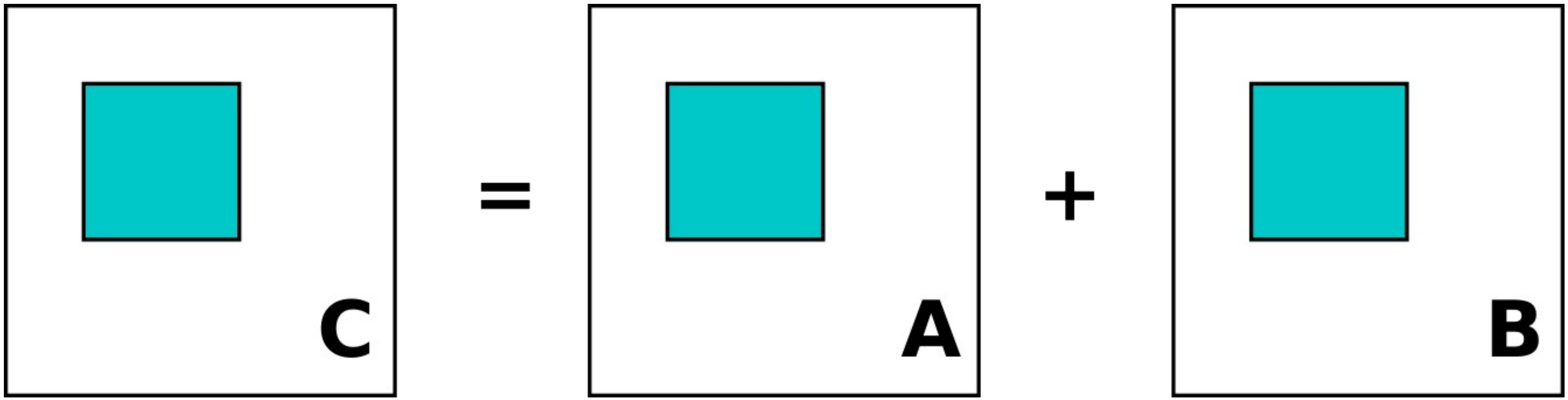
Matrix Addition

$C = \varepsilon(A + B)$



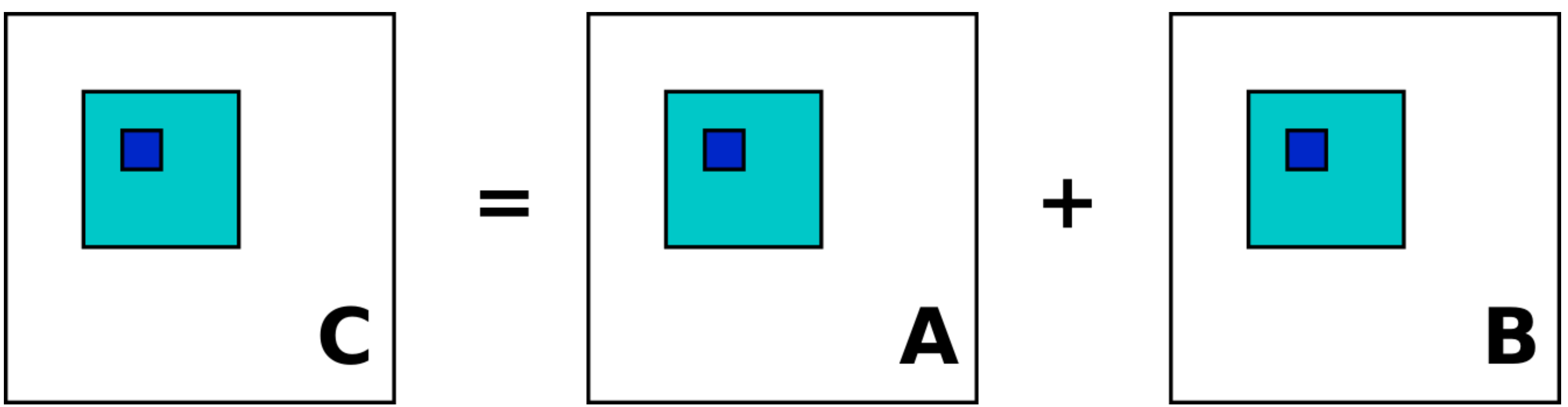
Matrix Addition

$C = \varepsilon(A + B)$



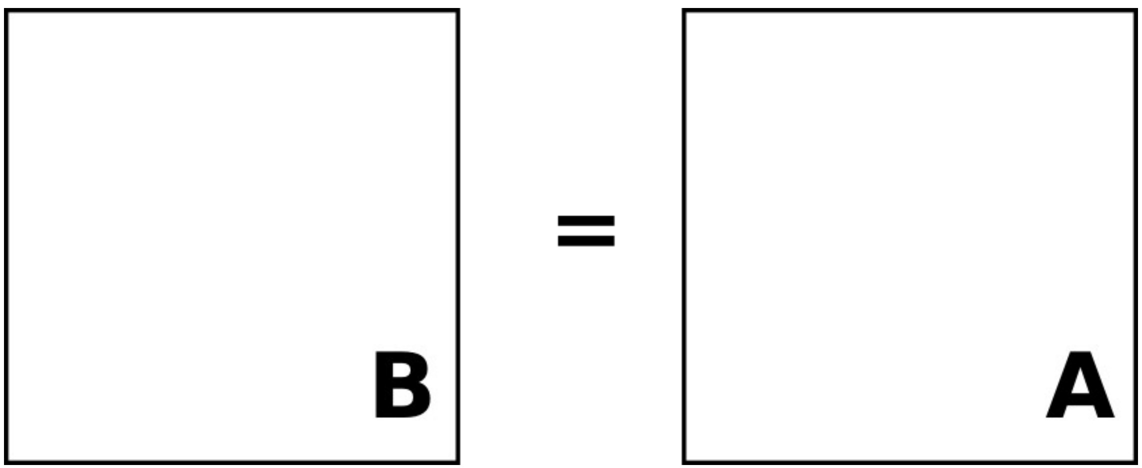
Matrix Addition

$C = \varepsilon(A + B)$



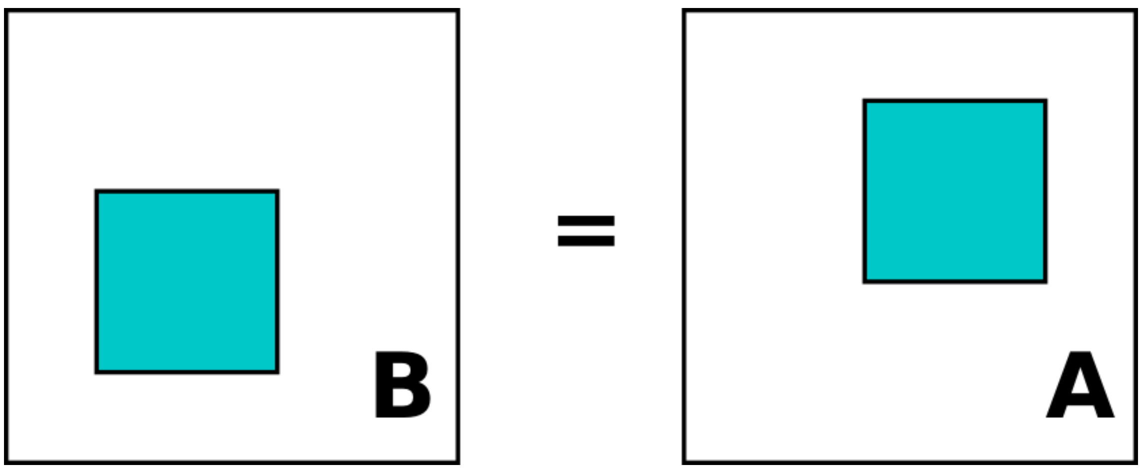
Matrix Transpose

$B = A^T$



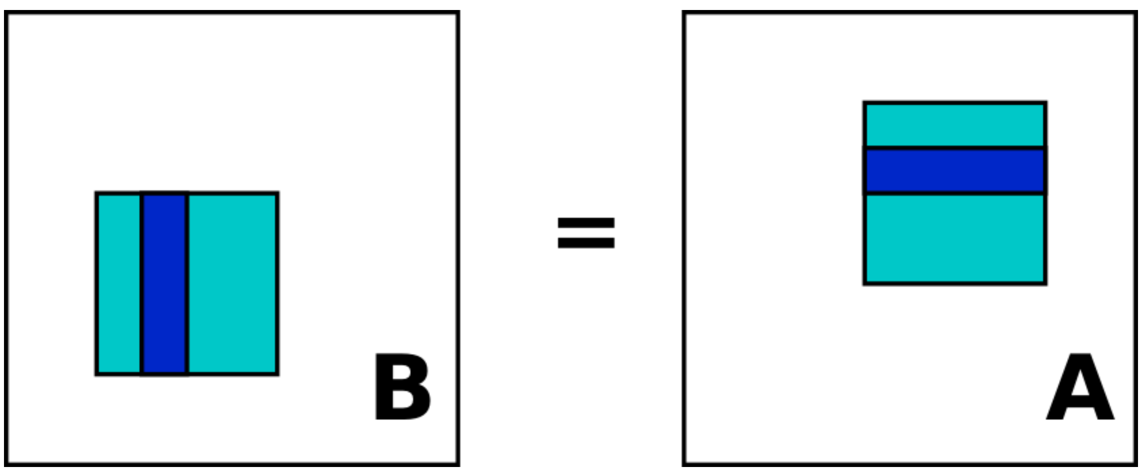
Matrix Transpose

$B = A^T$



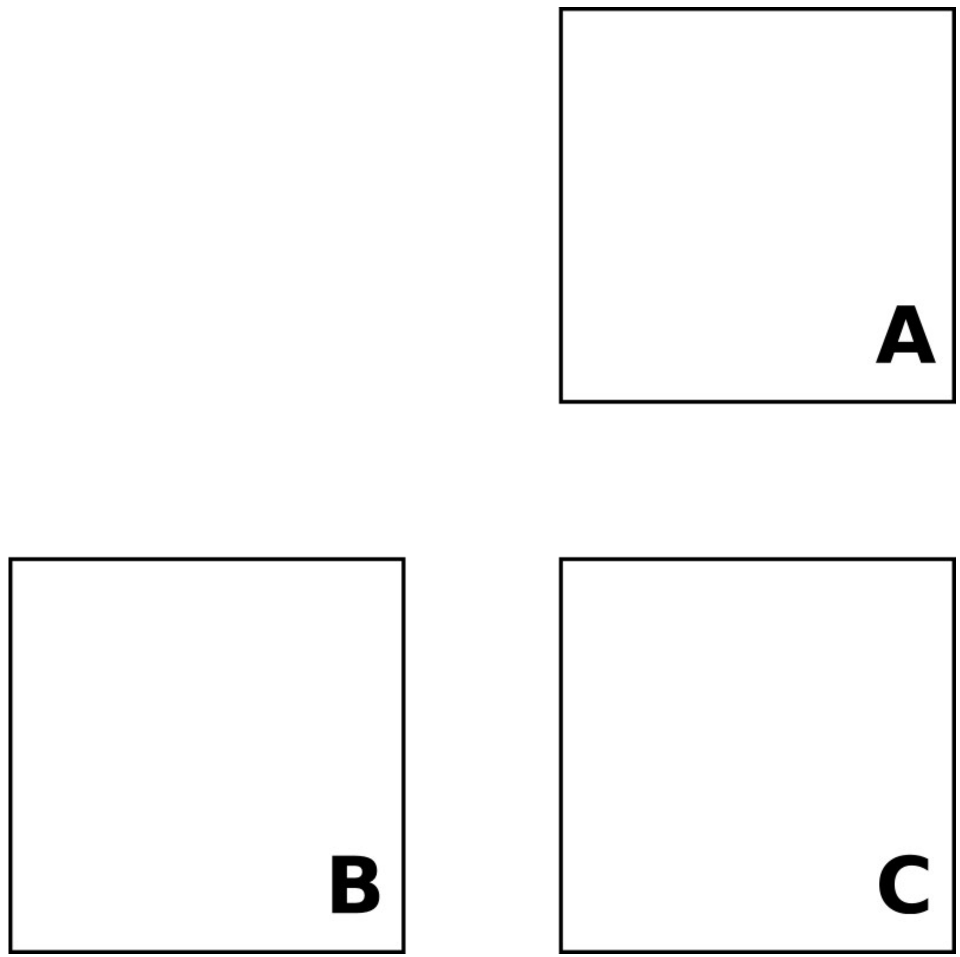
Matrix Transpose

$B = A^T$



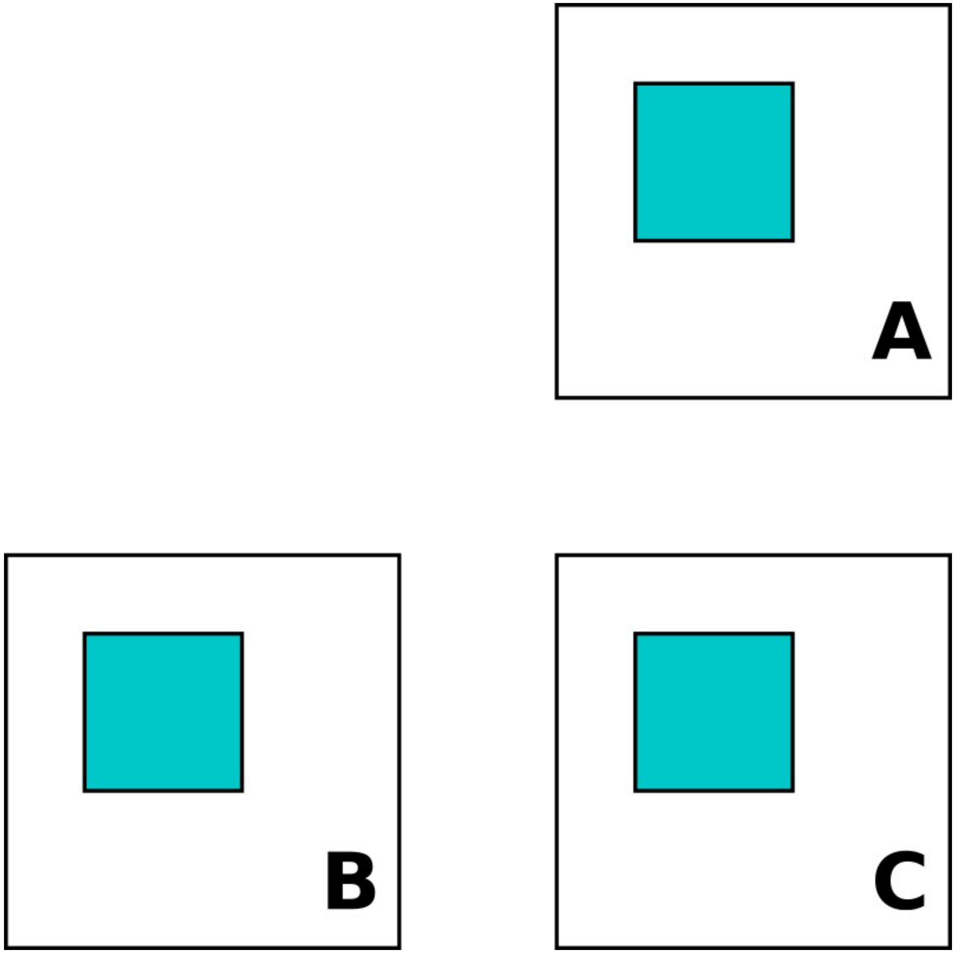
Matrix Multiply

C = BA



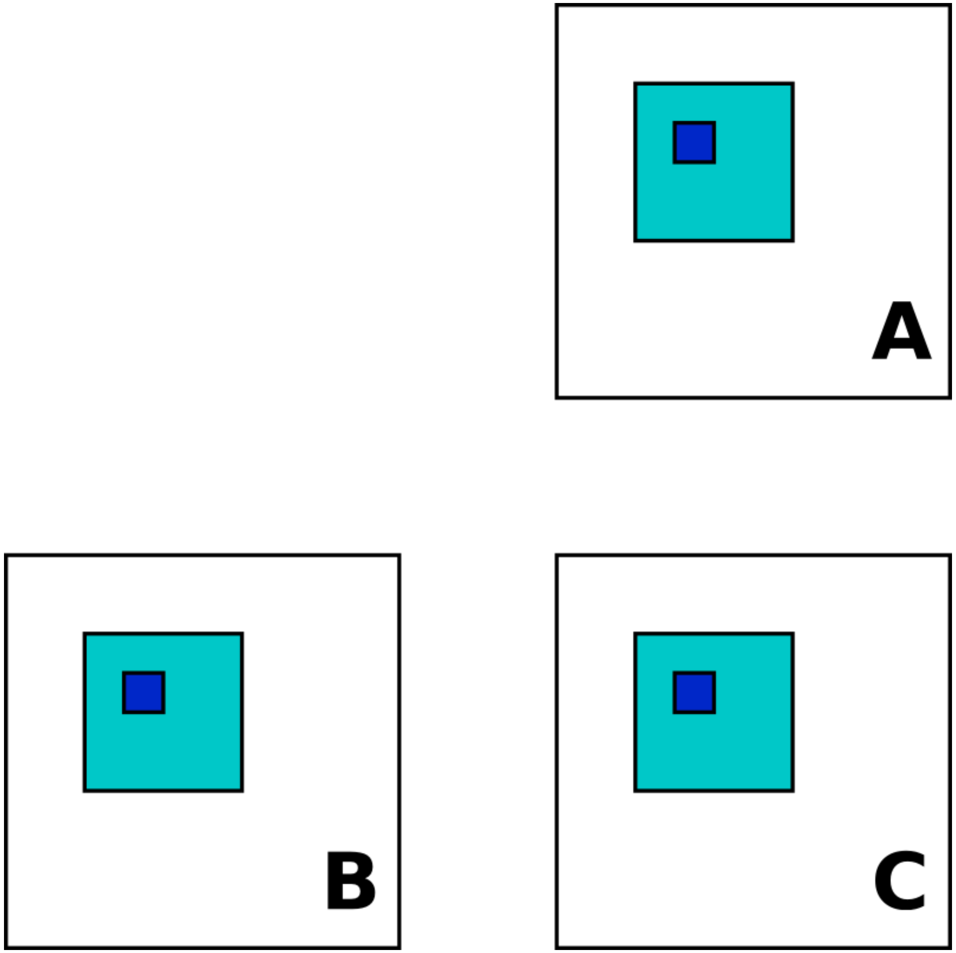
Matrix Multiply

C = BA



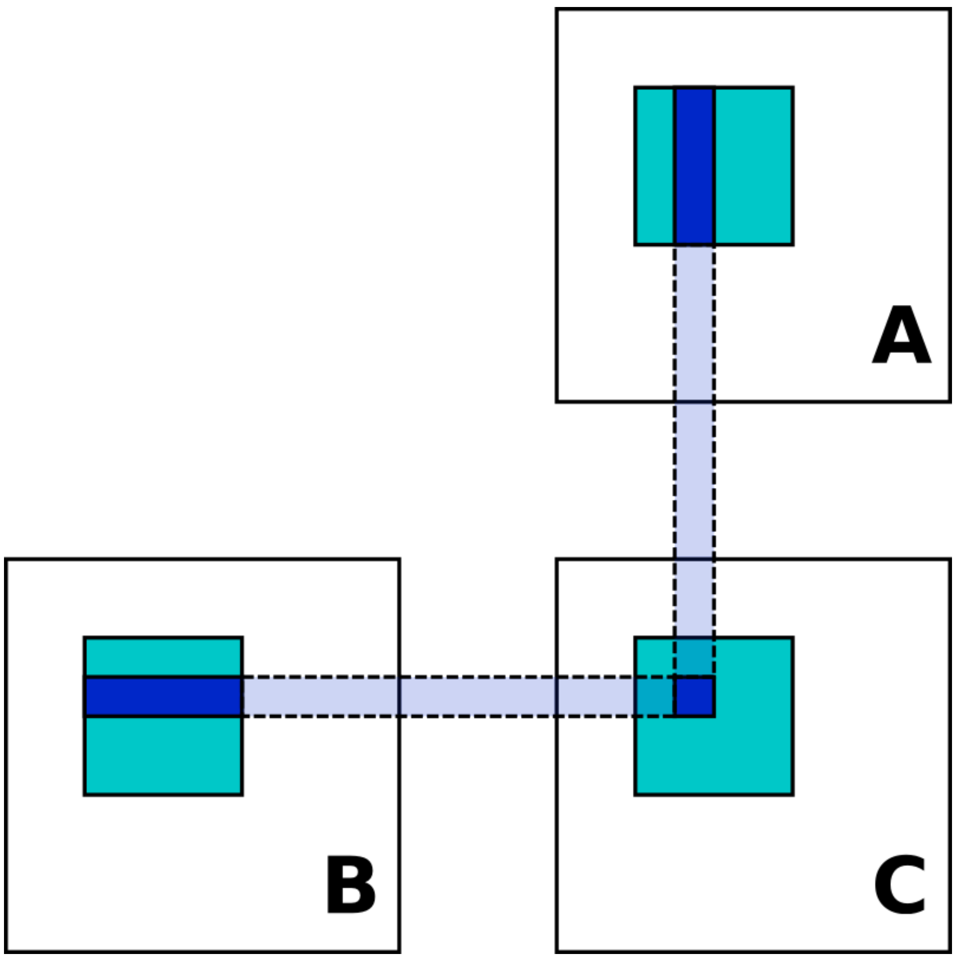
Matrix Multiply

C = BA



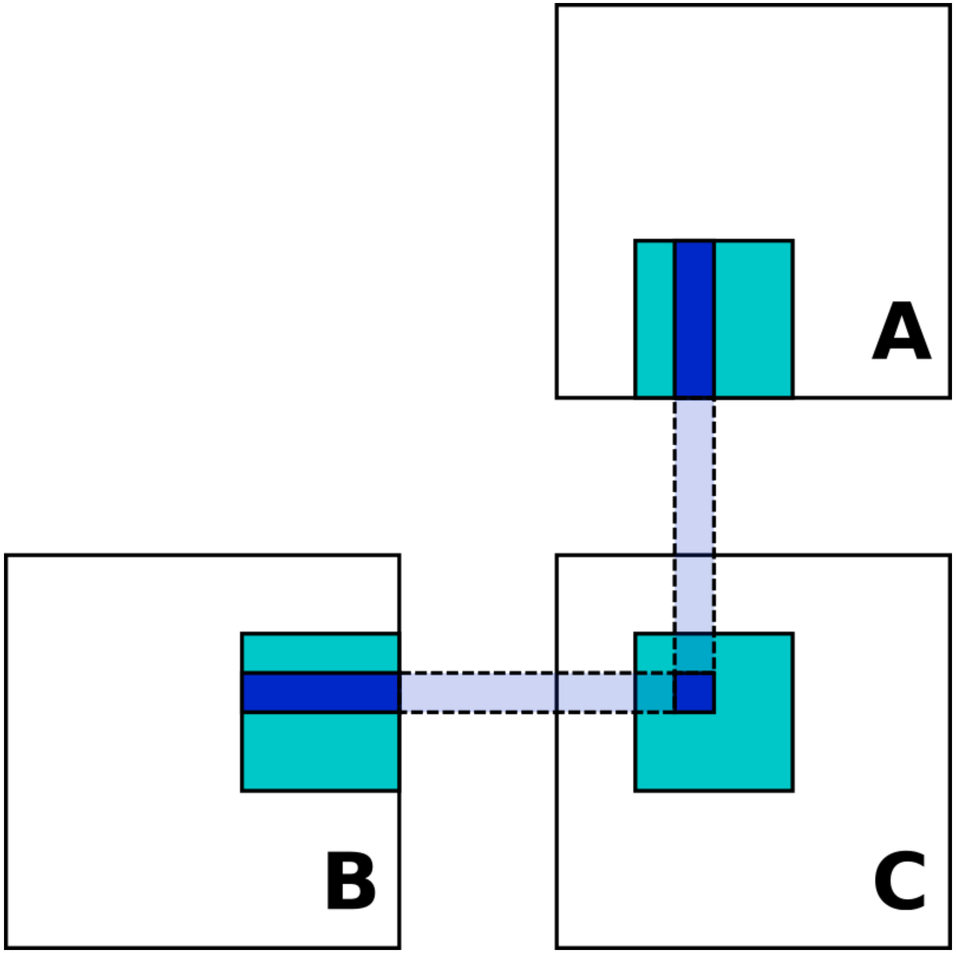
Matrix Multiply

C = BA



Matrix Multiply

C = BA



Random Number Generator

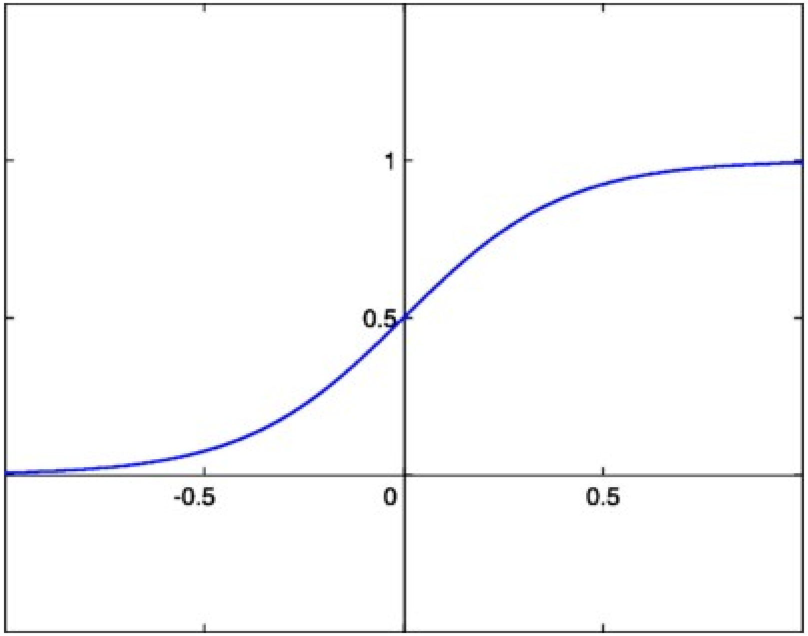
- Mersenne Twister
  - Pros
    - Uses bitwise operations
    - Long period (2^{19937}-1)
    - High dimensional equidistribution
    - Efficient memory usage
  - Cons
    - iterative
    - Insecure – after N outputs, it’s predictable

CUDA Implementation

- Launch many Mersenne twisters simultaneously
  - MT_RNG_COUNT=threads*blocks
- Initialization computes of per-thread configuration
  - dcmt0.4 library
    - Can be time consuming

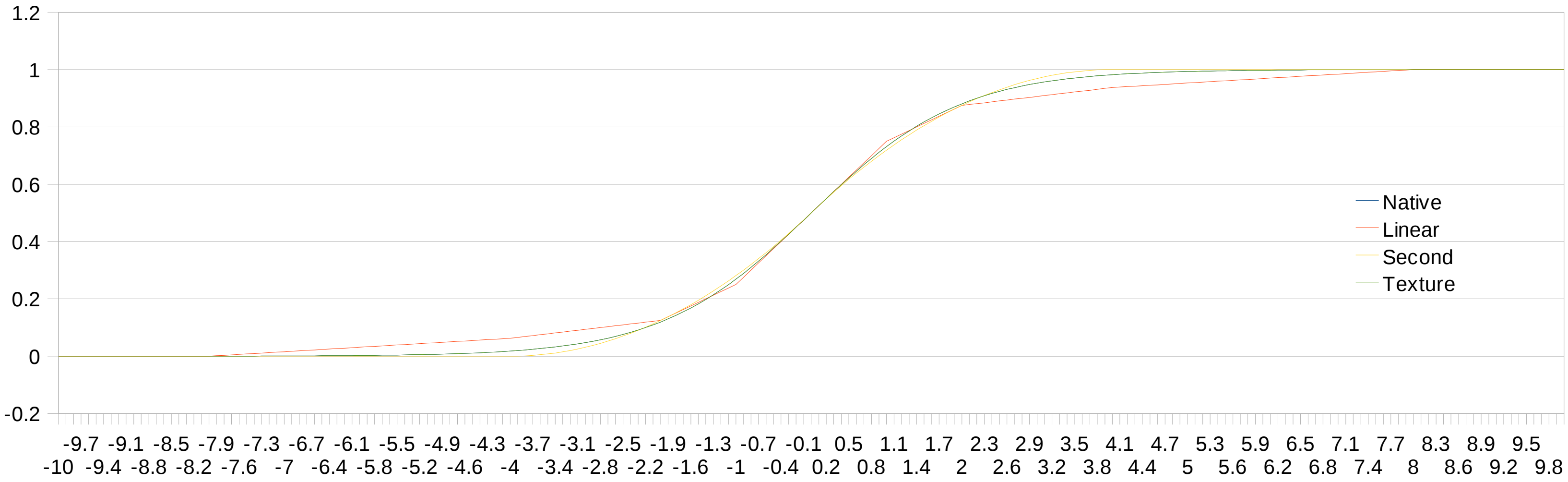
Sigmoid Function

$$P(t) = \frac{1}{1 + e^{-t}}.$$



- Road blocks
  - Number Representation
    - Fixed point
    - Floating point
  - Approximations
  - Input and Output are floats
- 100% parallel: one-to-one data-to-output map

Sigmoid Implementations



- Native floating point function
- Broken line approximation (9 segments)
- Second order approximation
- Precalculated texture (256 values)

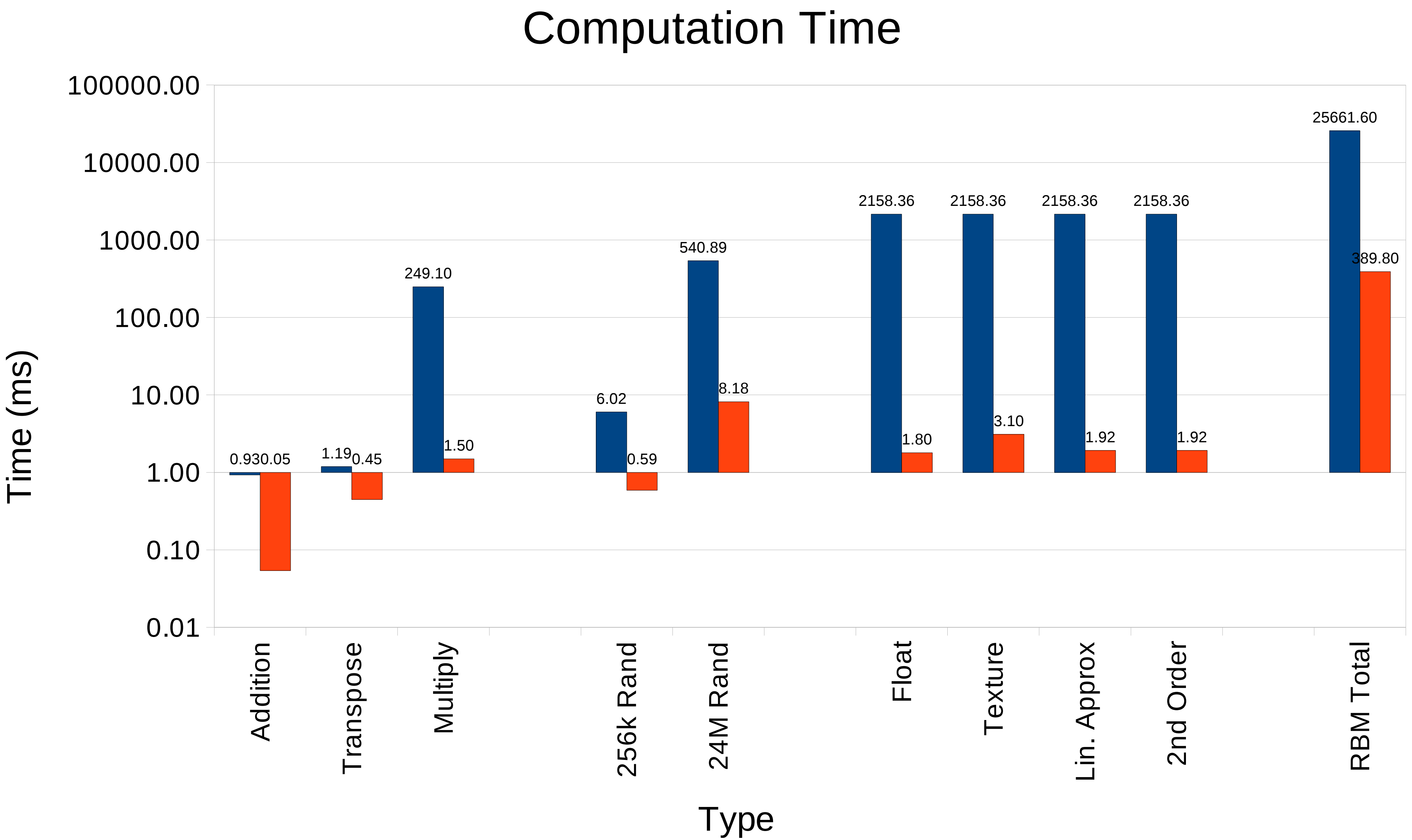
Results

- Software baseline
  - Optimized Sequential C++ code
  - Compiled with g++ version 4.1.2
    - Flags = -O3
  - 2.83GHz Intel Core2 Quad core, 6MB L2 Cache
  - 4GB DDR2 RAM
- CUDA implementation
  - GTX280

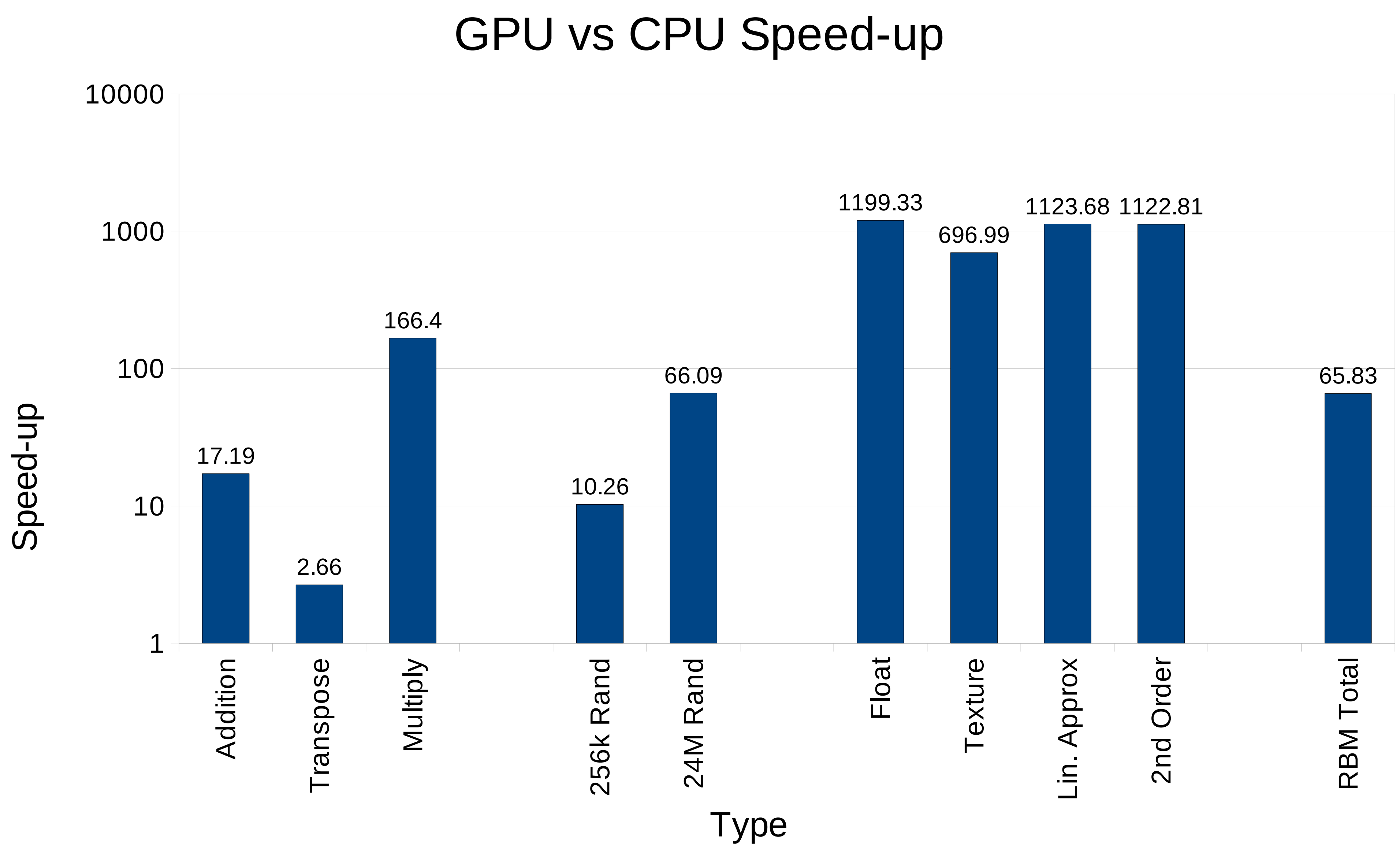
Results

- RBM Properties
  - 512 Nodes in Visible Layer
  - 512 Nodes in Hidden Layer
  - 256k Single-Precision Floating Point Weights
- Metrics
  - Performance Ratio
  - Error rate was not measured
    - Different random number implementations = different results

Results


Computation Time
Time (ms)
100000.00
10000.00
1000.00
100.00
10.00
1.00
0.10
0.01
0.93
0.05
1.19
0.45
249.10
1.50
6.02
0.59
540.89
8.18
2158.36
1.80
2158.36
3.10
2158.36
1.92
2158.36
1.92
25661.60
389.80
Addition
Transpose
Multiply
256k Rand
24M Rand
Float
Texture
Lin. Approx
2nd Order
RBM Total
Type

Results


GPU vs CPU Speed-up
10000
1000
100
10
1
Speed-up
17.19
2.66
166.4
10.26
66.09
1199.33
696.99
1123.68
1122.81
65.83
Addition
Transpose
Multiply
256k Rand
24M Rand
Float
Texture
Lin. Approx
2nd Order
RBM Total
Type

Discussion

- End Result program is much more efficient
- Programming evidence
  - >25 individual hours of debugging code
  - Memcpy in/out produces different results!
  - Adding dummy Memcpy changes behaviour!
  - One/Two consecutive prints change behaviour!
  - Removed compiler optimizations
  - Peppered code with cudeThreadSynchronize()
  - Conclussion: parallelization bugs by compiler

Conclusion

- CUDA implementations is well suited for Neural Network applications
  - 65 fold speed up was achieved for 512x512 network
- The CUDA language could be more mature and bug-free
- Further optimization still could be drawn from profiling and integrating code

Performance

- For 262144 numbers
  - Compared to CPU implementation of MT
    - ~10.26x speed up
  - Compared to optimized RNG on CPU
    - ~1.67x speed up
- For 24 million numbers
  - Compared to CPU implementation of MT
    - ~66.09x speed up
  - Compared to optimized RNG on CPU
    - ~11.19x speed up

Sigmoid Performance

	Time (CUDA)	Speed-up
CPU	2158.36	N/A
Float	1.7996	1199.33
Texture	3.0967	696.99
Linear Approximation	1.9208	1123.68
Second Order	1.9223	1122.81

- Native __expf() is the fastest method
- Texture slowest
- Normalized to 1000 kernel calls
- Sigmoid of 24M numbers