Stochastic Gradient Descent on Modern Hardware: Multi-core CPU or GPU? Synchronous or Asynchronous?

Yujing Ma, **Florin Rusu**, and Martin Torres
University of California Merced

ML/AI Golden Age

- Big training data
  - Millions – billions examples
  - Thousands – millions features (dimensions)

- Highly-parallel hardware
  - Multi-core CPUs with 20+ cores
  - GPUs with 1000s of cores
  - Huge memory (100s GBs – 1s TB on a server)

Stochastic Gradient Descent (SGD)

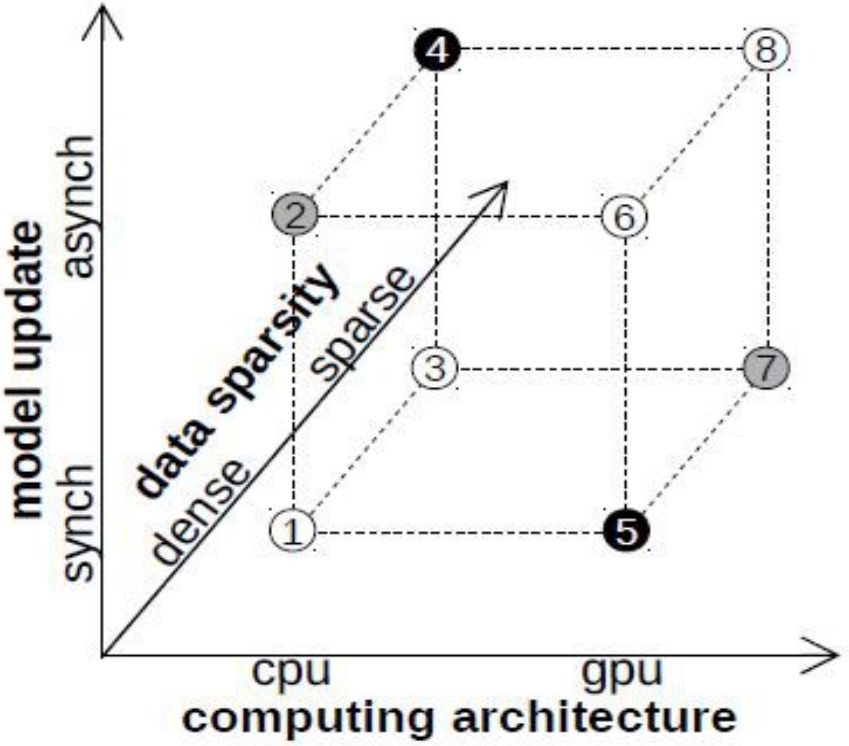
- Powers ML/AI golden age
- 100s of variants/implementations/papers
- Implemented by any single ML/AI system
  - Google Brain, Microsoft Project Adam, IBM System ML, Spark Mllib, etc.
- CPU & GPU implementations
  - Caffe, TensorFlow, MXNet, BIDMach, SINGA, Theano, Torch, etc.

SGD in Databases

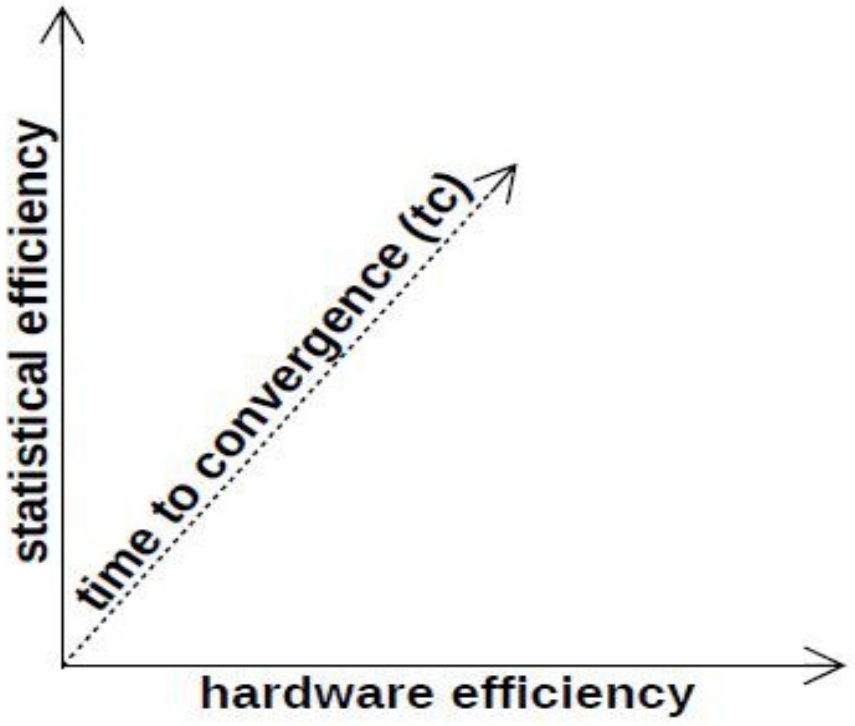
- It is not so much about deep learning
  - Regression (linear, logistic)
  - Classification (SVM)
  - Recommendation (LMF)
- Mostly about training
  - Inside DB, close to data
  - Over joins or factorized databases
  - Compressed data, (compressed) large models
- Selection of optimization algorithm and hyper-parameters
  - BGD vs. SGD vs. SCD

Contribution: SGD Study on Highly-Parallel Architectures

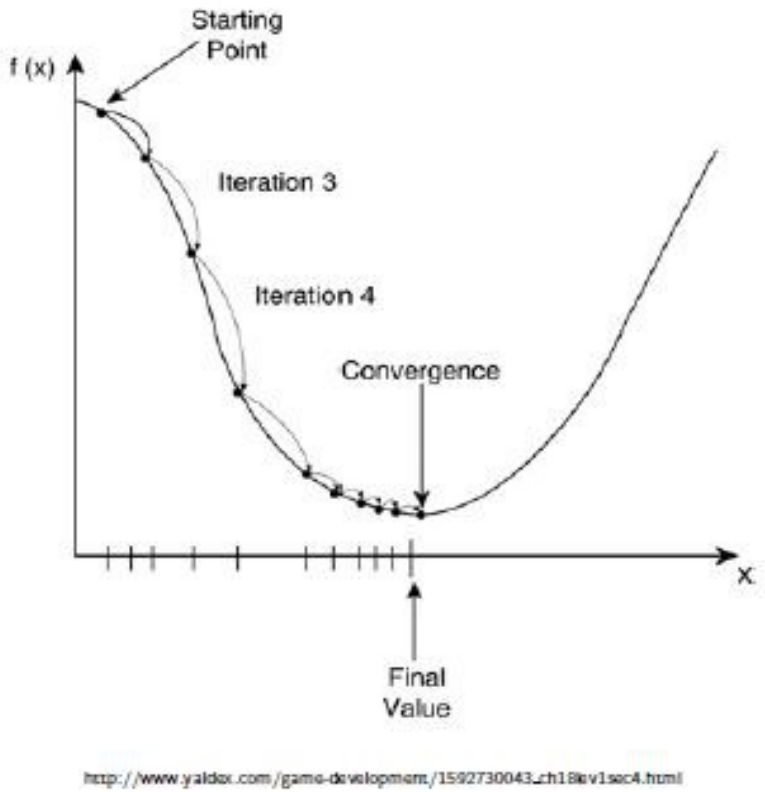
Exploratory axes



Performance axes



ML Training with Gradient Descent



http://www.yaldex.com/game-development/1592730043_ch18lev1sec4.html

$$min_{\vec{w}\in\mathbb{R}^d}\left\{\Lambda(\vec{w}) = \sum_{(\vec{x}_i,y_i)\in\text{data}} f(\vec{w},\vec{x}_i;y_i)\right\}$$

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \alpha^{(k)}\nabla\Lambda\left(\vec{w}^{(k)}\right)$$

$\Lambda(\vec{w})$ is the loss

$\nabla\Lambda(\vec{w}) = \left[\frac{\partial\Lambda(\vec{w})}{\partial w_1},\ldots,\frac{\partial\Lambda(\vec{w})}{\partial w_d}\right]$ is the gradient

$\alpha^{(k)}$ is step size or learning rate

$\vec{w}^{(0)}$ is the starting point (random)

- Convergence to minimum guaranteed for convex objective function

$$f_{LR}(\vec{w}) = \log\left(1 + e^{-y_i\vec{x}_i\cdot\vec{w}}\right)$$

$$\frac{\partial f_{LR}(\vec{w})}{\partial w_j} = x_{ij}\left(-y_i\frac{e^{-y_i\vec{x}_i\cdot\vec{w}}}{1+e^{-y_i\vec{x}_i\cdot\vec{w}}}\right)$$

$$f_{SVM}(\vec{w}) = max\,(0, 1 - y_i\vec{x}_i\cdot\vec{w})$$

$$\frac{\partial f_{SVM}(\vec{w})}{\partial w_j} = \begin{cases} -y_i x_{ij}, & \text{if } y_i\vec{x}_i\cdot\vec{w} < 1 \\ 0, & \text{otherwise} \end{cases}$$

Stochastic Gradient Descent (SGD)

Algorithm 1 Stochastic Gradient Descent (SGD)

Require:

Training examples $\vec{\vec{X}} \in \mathbb{R}^{N \times d}$ and their labels $\vec{Y} \in \mathbb{R}^N$

Loss function f and its gradient $\overrightarrow{\nabla f}$

Initial model $\vec{w} \in \mathbb{R}^d$ and step size $\alpha \in \mathbb{R}$

Number of epochs t and batch size B

1. **for** $k = 1$ **to** t **do**

OPTIMIZATION EPOCH

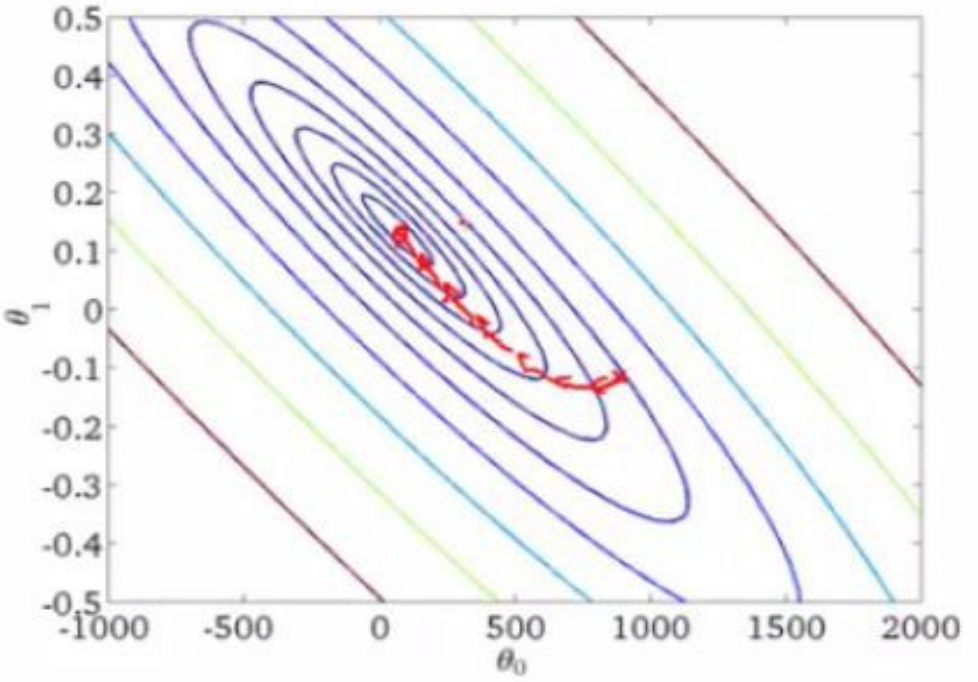
2. Select a random subset of B examples $\vec{\vec{X}}_k = \{\vec{x}_{i_1}, \ldots, \vec{x}_{i_B}\}$ and their labels $\vec{Y}_k = \{y_{i_1}, \ldots, y_{i_B}\}$
3. Compute gradient estimate: $\vec{g} \leftarrow \sum_{\vec{\vec{X}}_k, \vec{Y}_k} \overrightarrow{\nabla f}(\vec{w})$
4. Update model: $\vec{w} \leftarrow \vec{w} - \alpha \vec{g}$
5. **end for**
6. **return** $\vec{w}$

(Mini-)Batch SGD

Algorithm 2 Batch SGD Optimization Epoch

1. Compute gradient:

   for $i = 1$ **to** N **do** $\vec{g} \leftarrow \vec{g} + \overrightarrow{\nabla f}\,(\vec{w}; \vec{x}_i, y_i)$
2. Update model: $\vec{w} \leftarrow \vec{w} - \alpha\vec{g}$



http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html

Synchronous Parallel SGD

Algorithm 2 Batch SGD Optimization Epoch

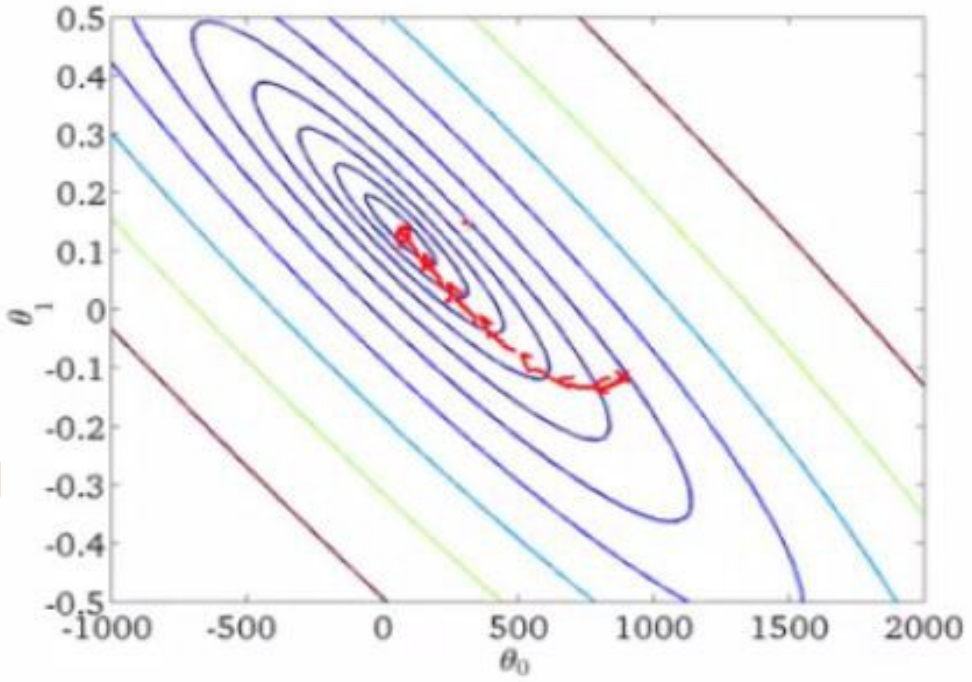
1. Compute gradient:

   for $i = 1$ **to** N **do** $\vec{g} \leftarrow \vec{g} + \overrightarrow{\nabla f}\left(\vec{w}; \vec{x}_i, y_i\right)$

2. Update model: $\vec{w} \leftarrow \vec{w} - \alpha\vec{g}$

$$\vec{g} = \vec{\vec{X}}_k^T \odot \left(-\vec{Y}_k \cdot \frac{e^{-\vec{Y}_k \cdot \vec{\vec{X}}_k \odot \vec{w}}}{1 + e^{-\vec{Y}_k \cdot \vec{\vec{X}}_k \odot \vec{w}}} \right)$$

Parallel execution on CPU and/or GPU as linear algebra kernels

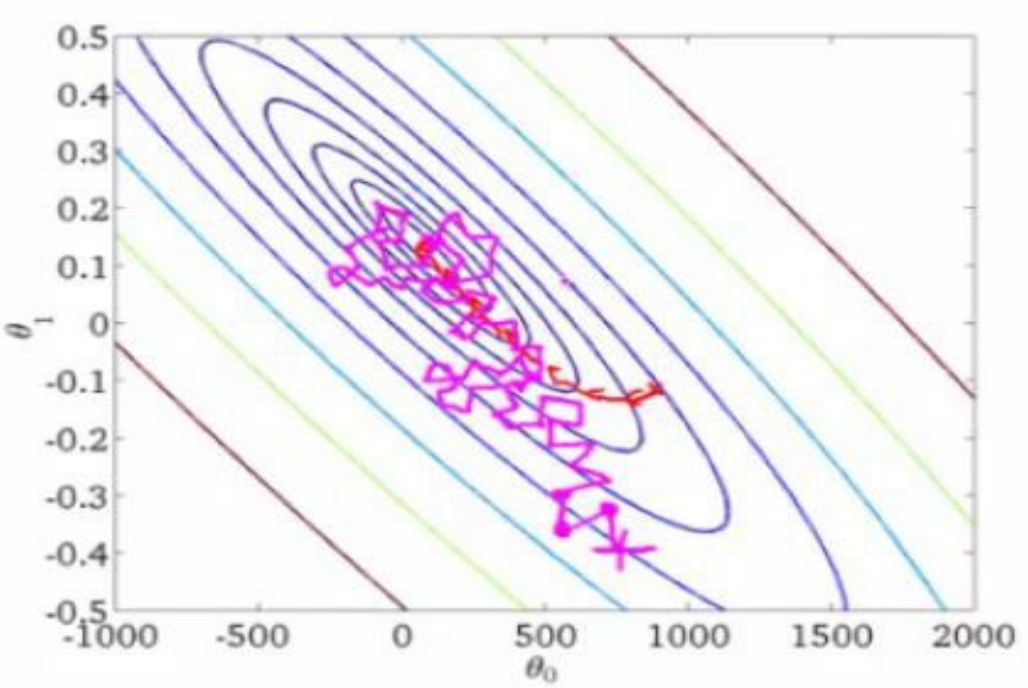


http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html

Incremental SGD

Algorithm 3 Incremental SGD Optimization Epoch

1. **for** $i = 1$ **to** N **do**
2. Compute gradient estimate: $\vec{g} \leftarrow \overrightarrow{\nabla f}\,(\vec{w}; \vec{x}_i, y_i)$
3. Update model: $\vec{w} \leftarrow \vec{w} - \alpha\vec{g}$
4. **end for**



http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html

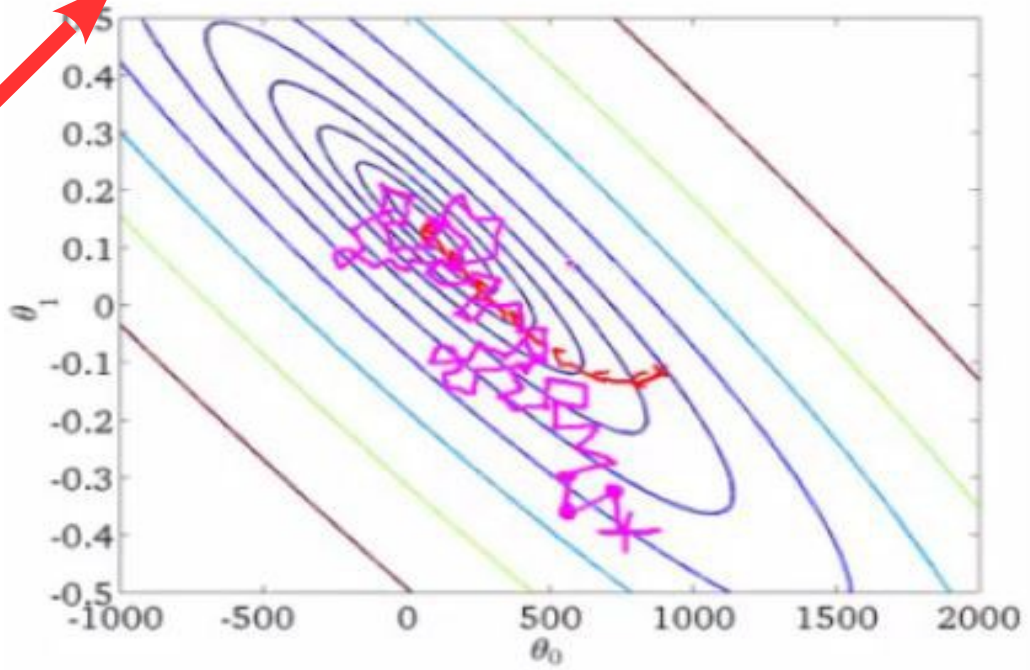
Asynchronous Parallel SGD (Hogwild!)

Algorithm 3 Incremental SGD Optimization Epoch

1. **for** $i = 1$ **to** N **do** in parallel
2. Compute gradient estimate: $\vec{g} \leftarrow \overrightarrow{\nabla f}(\vec{w}; \vec{x}_i, y_i)$
3. Update model: $\vec{w} \leftarrow \vec{w} - \alpha\vec{g}$
4. **end for**

$$\frac{\partial f_{LR}(\vec{w})}{\partial w_j} = x_{ij}\left(-y_i \frac{e^{-y_i\vec{x}_i\cdot\vec{w}}}{1+e^{-y_i\vec{x}_i\cdot\vec{w}}}\right)$$

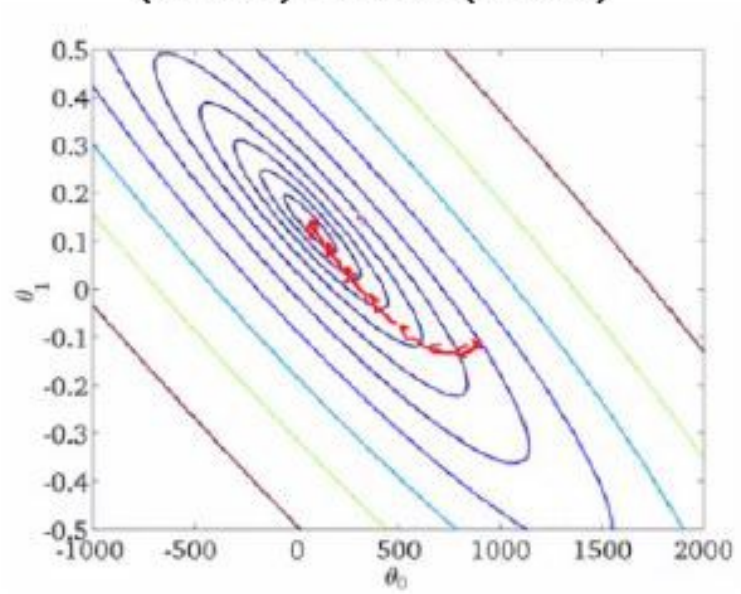
No synchronization or locks



http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html

Batch v Incremental

(Mini-)Batch (BGD)

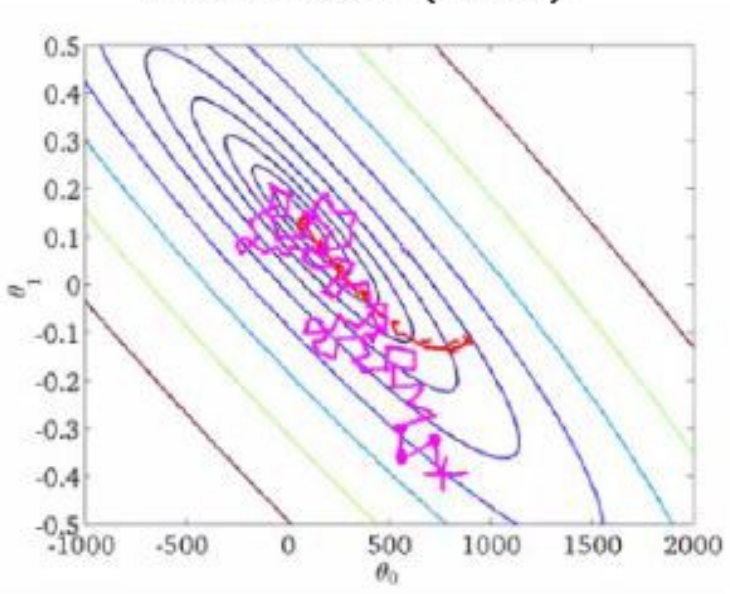


http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \alpha^{(k)} \nabla \Lambda \left(\vec{w}^{(k)} \right)$$

- Exact/accurate gradient computation
- One step for per batch/iteration
- Faster convergence close to minimum

Stochastic (SGD)

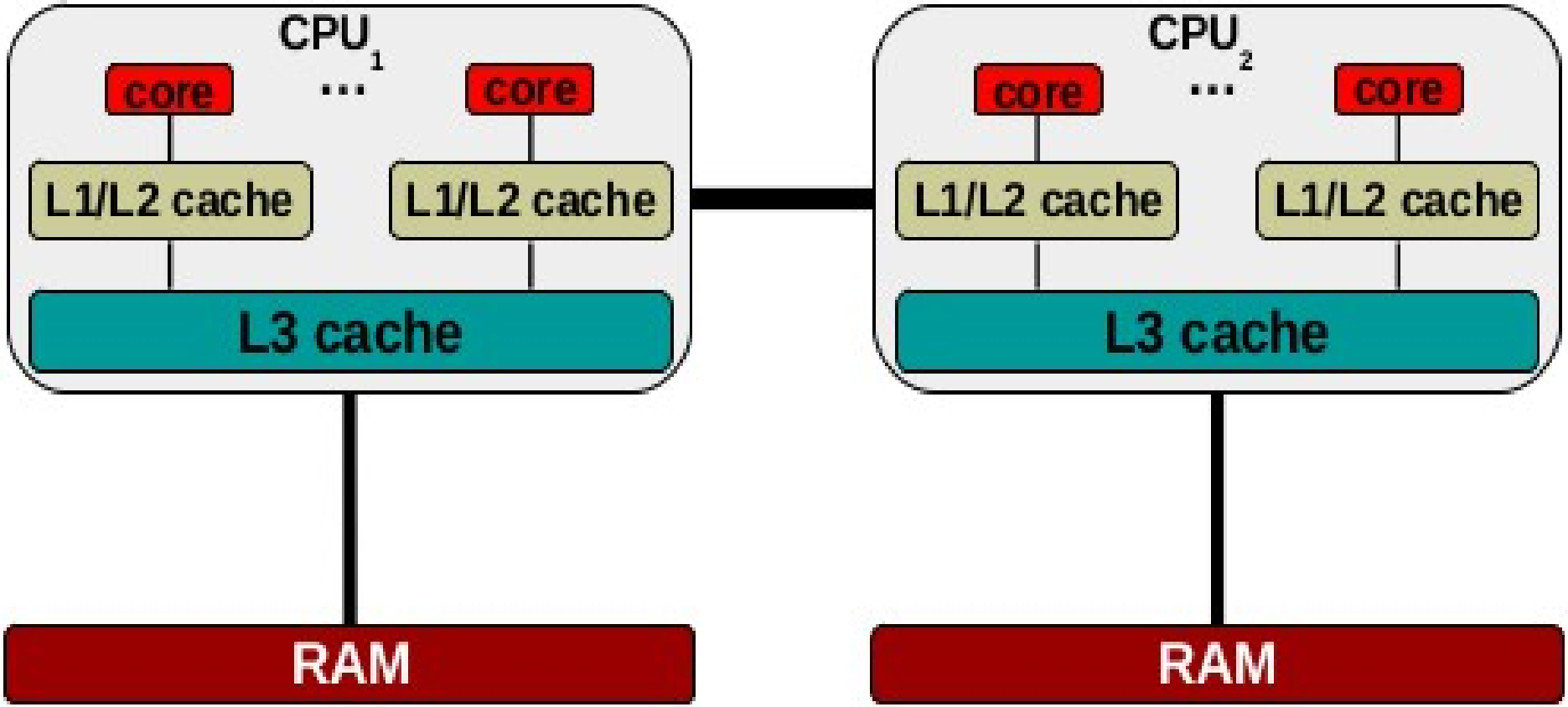


http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html

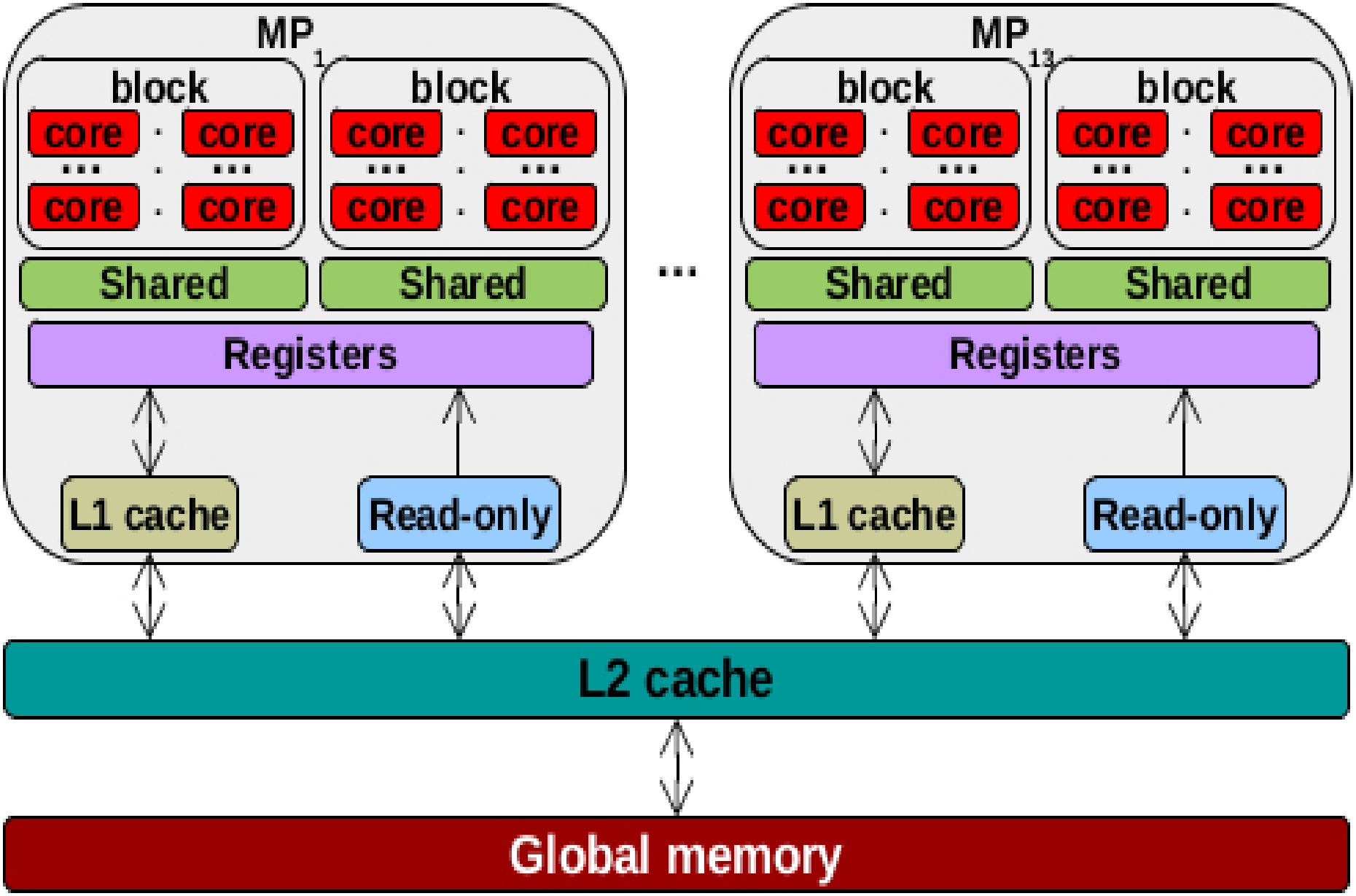
$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \beta^{(k)} \nabla f \left(\vec{w}^{(k)}, \vec{x}_{\eta^{(k)}}; y_{\eta^{(k)}} \right)$$

- Approximate gradient at each data point
- One step per data point
- Faster convergence far from minimum

NUMA CPU Architecture



GPU Architecture



NUMA CPU v GPU

	NUMA	GPU
CPU/MP	2	13
cores	14 per CPU	192 per MP
blocks	-	16 per MP
threads	28 per CPU	2048 per MP
L1 cache	32+32 KB	48 KB
L2 cache	256 KB	1.5 MB
L3 /shared	35 MB	48 KB
RAM/global	256 GB	12 GB

- CPU: 2 x Intel Xeon E5-2660 (14 cores, 28 threads)
- GPU: Tesla K80 (use only one multiprocessor, ~K40)

Datasets & Tasks

dense

dataset	#examples	#features	#nnz/exp (avg)	size (s/d)
covtype	581,012	54	54 to 54 (54)	– / 485MB
w8a	64,700	300	0 to 114 (12)	4.4MB / 155MB
real-sim	72,309	20,958	1 to 3,484 (51)	87MB / 12.1GB
rcv1	677,399	47,236	4 to 1,224 (73)	1.2GB / 256GB
news	19,996	1,355,191	1 to 16,423 (455)	134MB / 217GB

sparse

dataset	MLP architecture
covtype	54-10-5-2
w8a	300-10-5-2
real-sim	50-10-5-2
rcv1	50-10-5-2
news	300-10-5-2

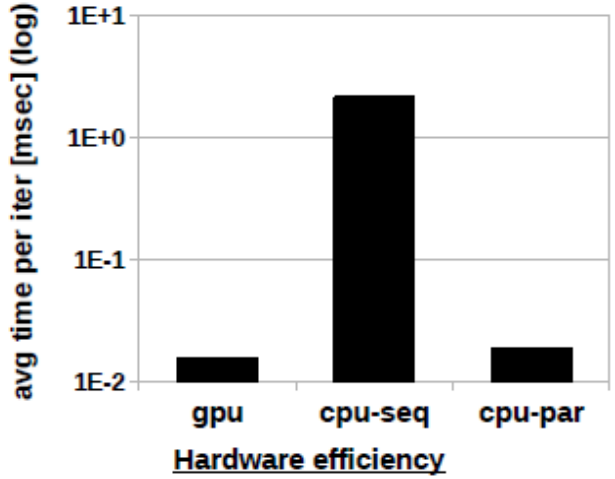
Synchronous SGD Implementation

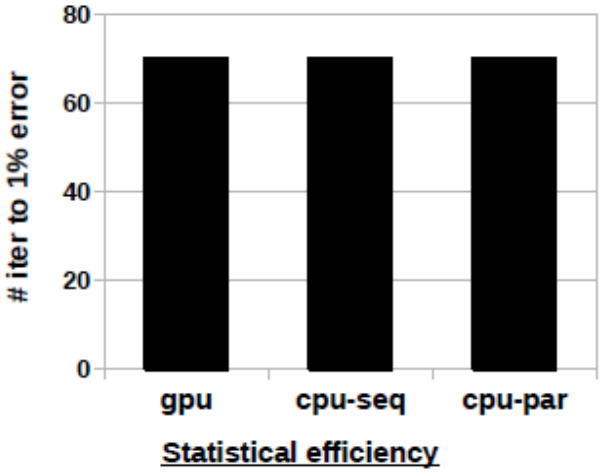
```
vector a = matrix-vector-product(data, model)
a = vector-vector-element-product(label, a)
a = vector-element-exponent(a)
vector b = vector-element-sum(1, a)
a = vector-vector-element-division(a, b)
a = vector-vector-element-product(a, -label)
gradient = matrix-vector-product(transpose(data), a)
```

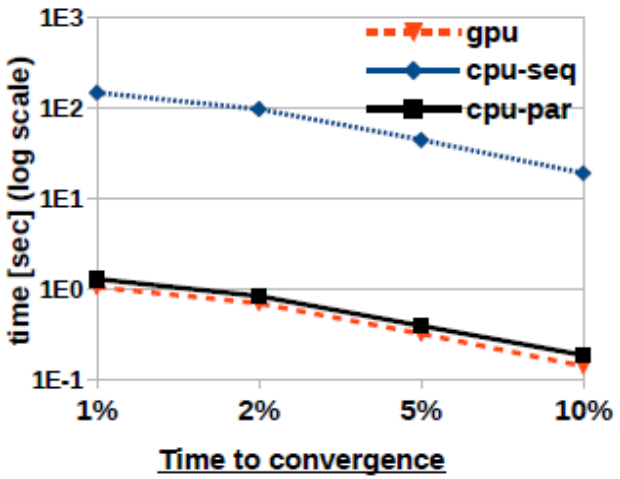
ViennaCL (1.7.1) library kernels

- Same API for CPU and GPU
- Separate compilation for each architecture

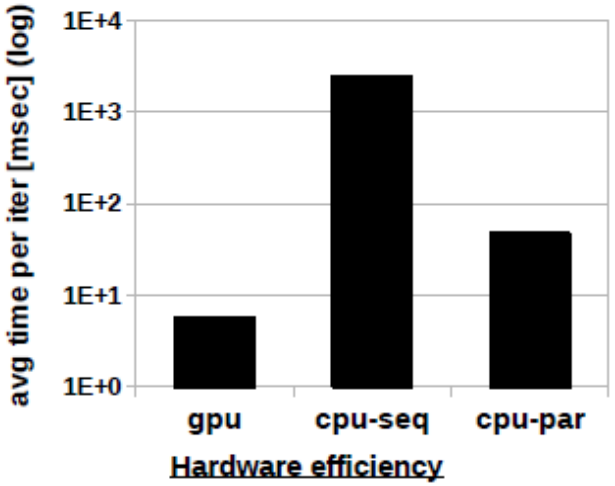
Synchronous SGD Study (LR)

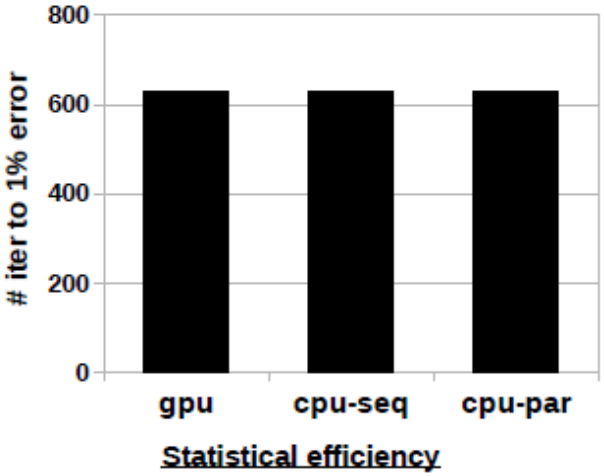


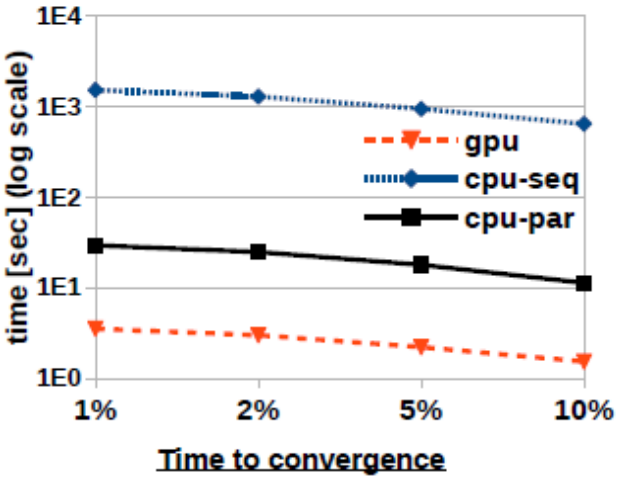




dense







sparse

Synchronous SGD Results

task	dataset	time-to-convergence (sec)			time-per-iteration (msec)			epochs	speedup	
		gpu	cpu-seq	cpu-par	gpu	cpu-seq	cpu-par		cpu-seq/cpu-par	cpu-par/gpu
LR	covtype	1.05	145.11	1.29	15	2,073	18.42	70	112.54	1.23
	w8a	0.37	148.88	0.46	4.87	1,959	6.05	76	323.80	1.24
	real-sim	3.10	1,537.90	7.67	4.43	2,197	10.96	700	200.46	2.47
	rcv1	31.69	2,227.05	48.06	44.82	3,150	67.98	707	46.34	1.52
	news	0.65	240.21	3.68	6.37	2,355	36.08	102	65.27	5.66
SVM	covtype	10.22	1,344.65	13.50	14.27	1,878	18.85	716	99.63	1.32
	w8a	0.78	342.85	0.80	4.13	1,814	4.23	189	428.84	1.02
	real-sim	0.23	75.59	0.46	6.22	2,043	12.43	37	164.36	2.00
	rcv1	1.13	111.61	2.61	29.74	2,937	68.69	38	42.76	2.31
	news	0.30	98.42	1.69	6.67	2,187	37.56	45	58.23	5.63
MLP	covtype	1,498	19,398	10,009	919	11,908	6,145	1,629	1.94	6.68
	w8a	83.57	909	388	107	1,161	495	783	2.34	4.64
	real-sim	21.99	229	93.98	130	1,365	556	168	2.46	4.26
	rcv1	48.91	1,146	241	1,193	16,960	5,880	41	2.89	4.93
	news	4.03	35.04	16.08	40.23	357	164	98	2.17	4.08

Asynchronous SGD Implementation (Hogwild!)

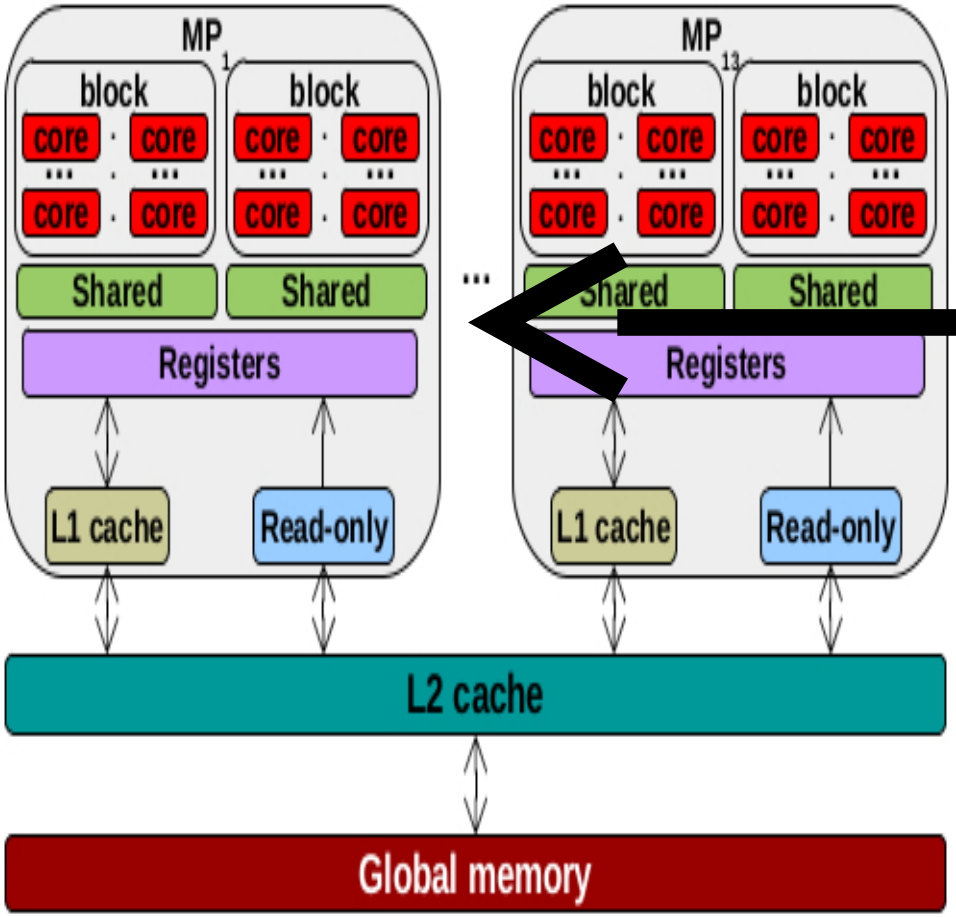
Dimension	Strategies
Data access path	row-major round-robin (row-rr) row-major chunking (row-ch) column-major round-robin (col-rr) column-major chunking (col-ch)
Model replication	kernel block thread example
Data replication	no replication (no-rep) k-wise replication (rep-2, rep-5, rep-10)

- NUMA CPU
  - Extensive study in DimmWitted by Zhang and Re (PVLDB 2014)
- GPU
  - Novel study

Map Hogwild! to GPU

Algorithm 3 Incremental SGD Optimization Epoch

1. **for** $i = 1$ **to** N **do**
2. Compute gradient estimate: $\vec{g} \leftarrow \overrightarrow{\nabla f}(\vec{w}; \vec{x}_i, y_i)$
3. Update model: $\vec{w} \leftarrow \vec{w} - \alpha\vec{g}$
4. **end for**



Algorithm

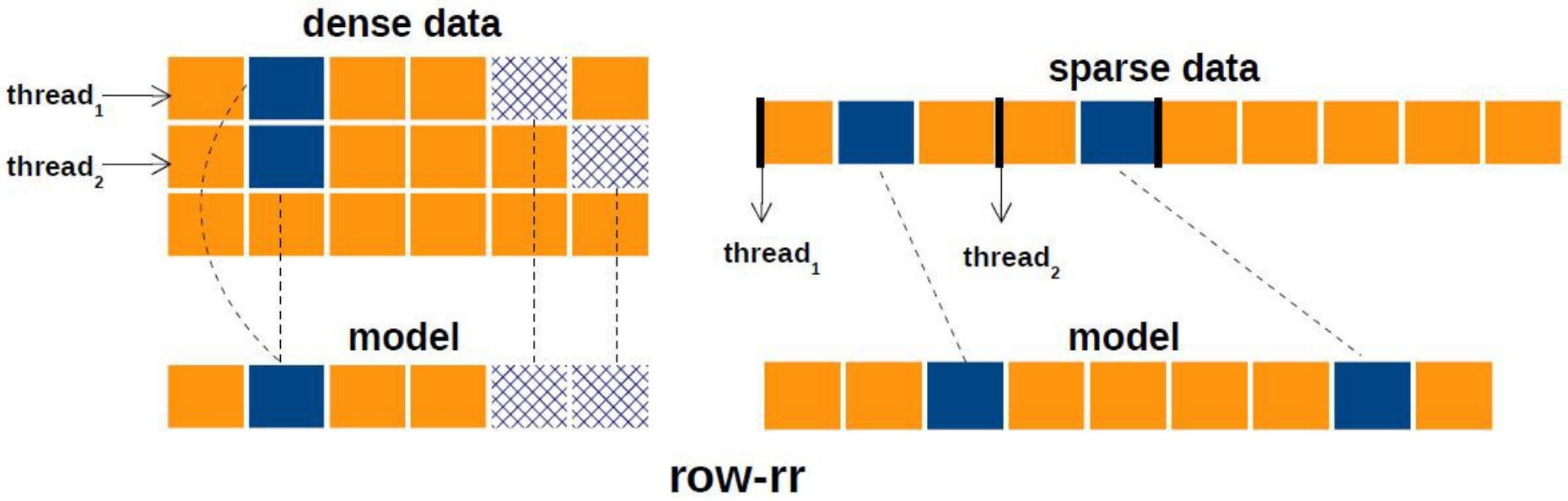
Copy data and model to GPU

While ~~not converge~~ do

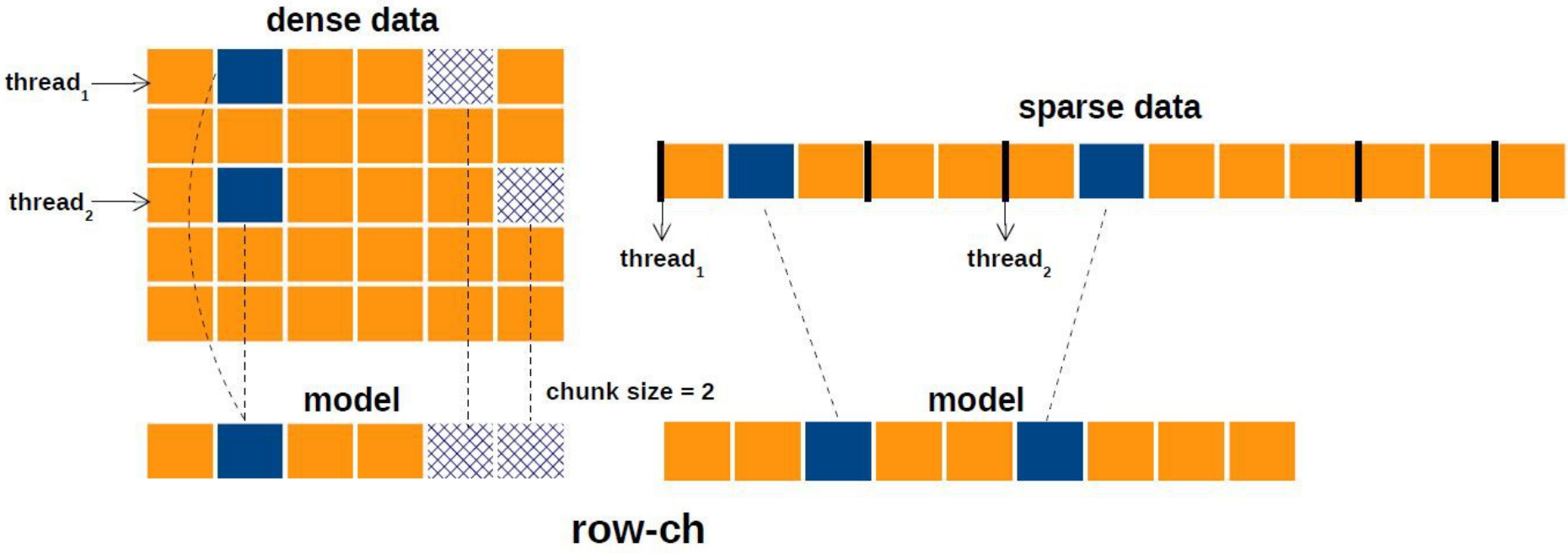
Execute Hogwild! kernel

End while

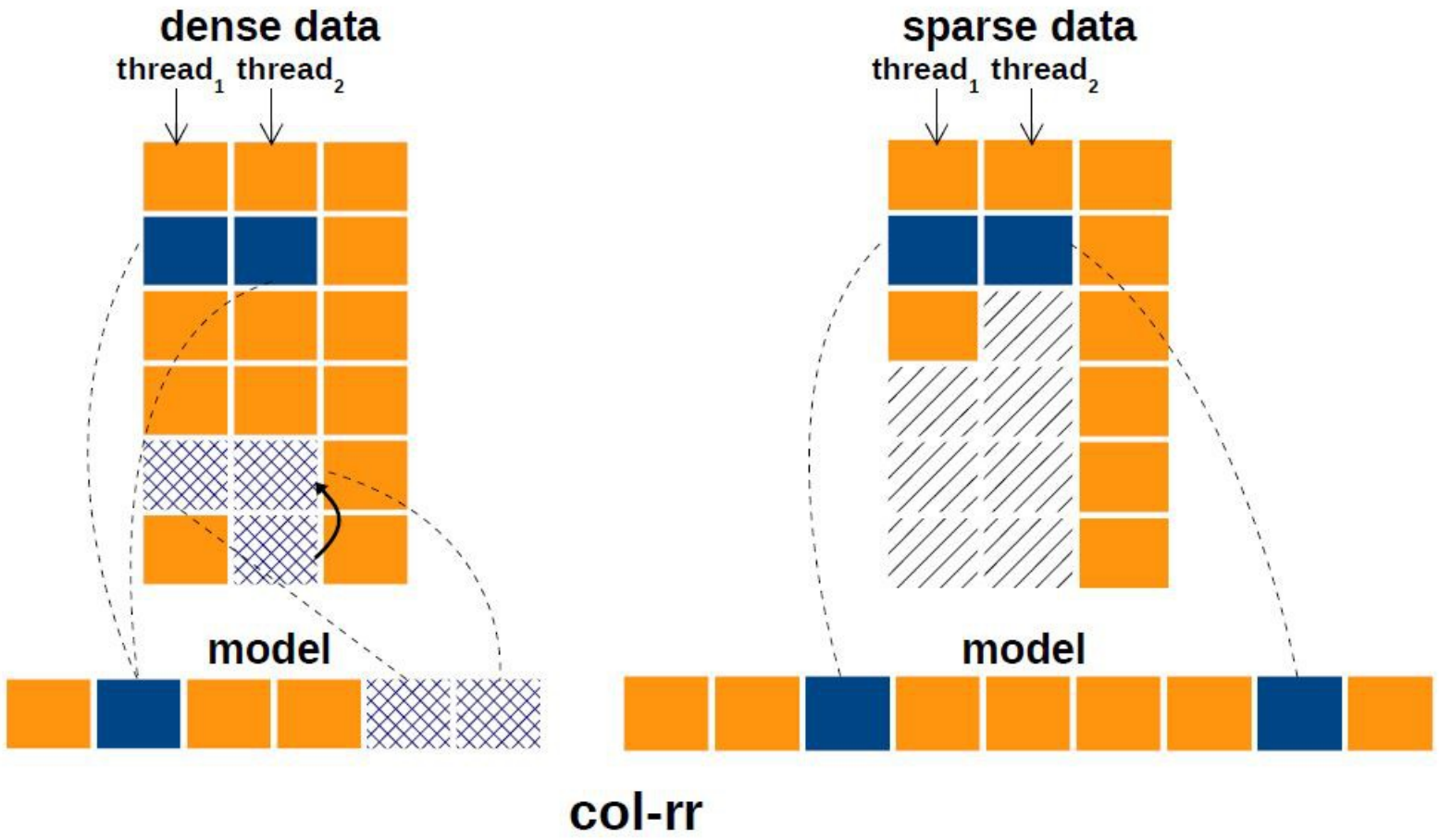
Data Access Path
row-major round-robin (row-rr)



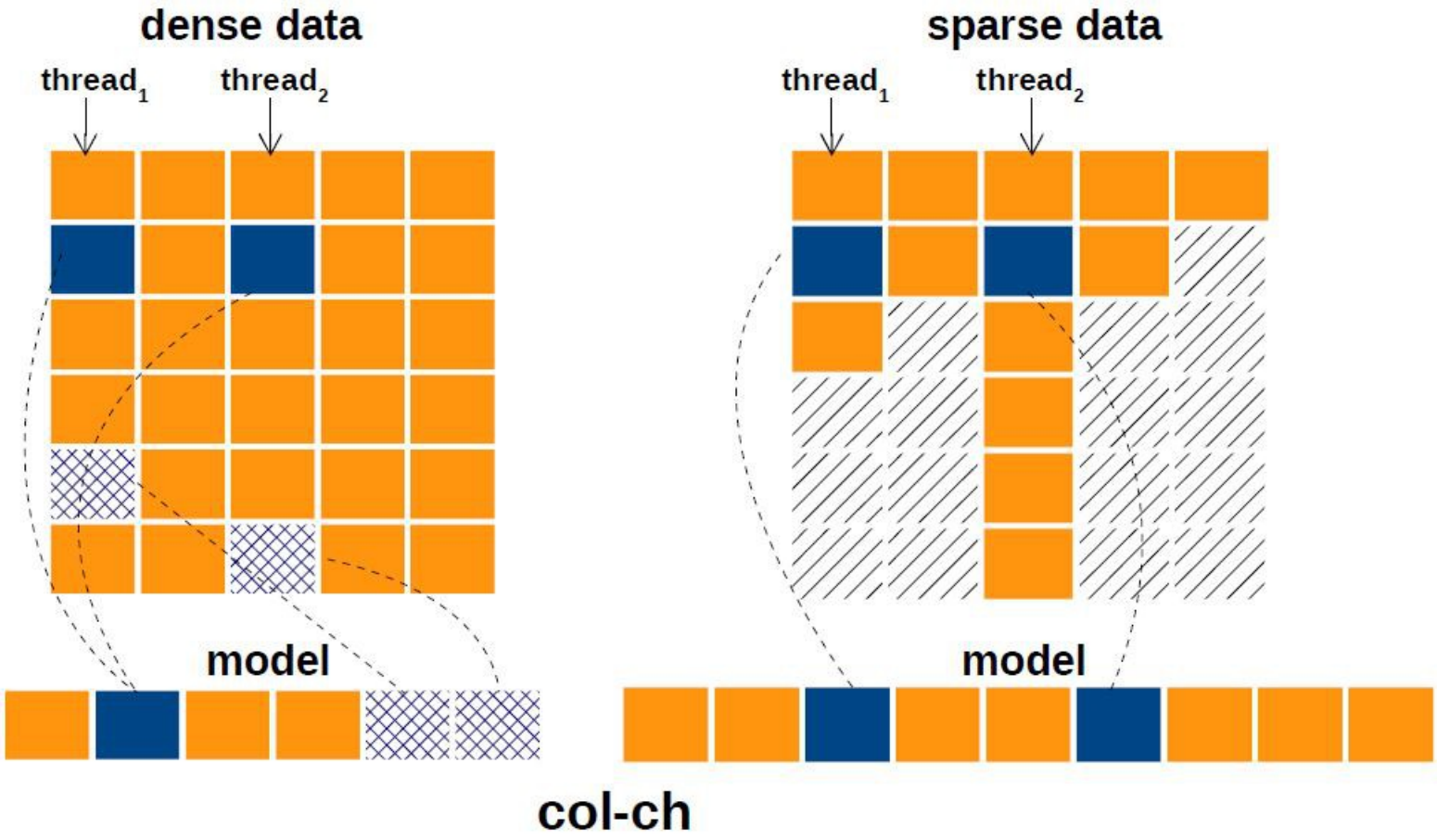
Data Access Path
row-major chunking (row-ch)



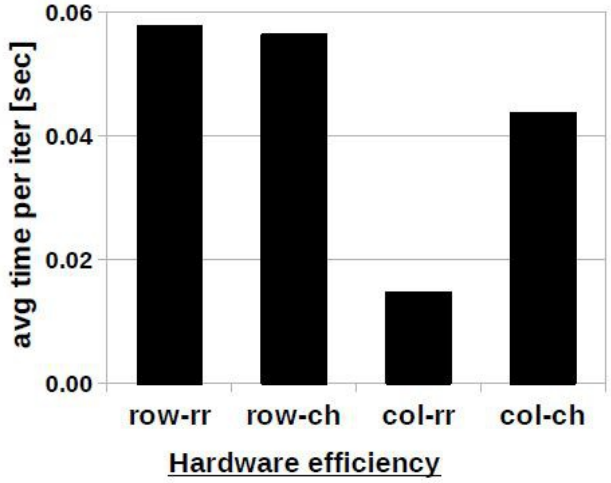
Data Access Path
column-major round-robin (col-rr)

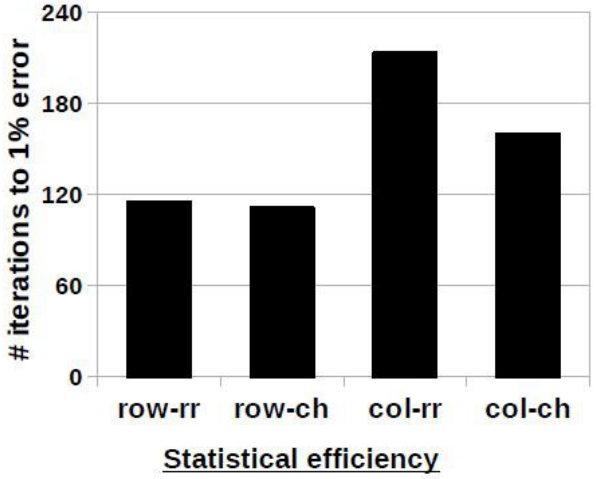


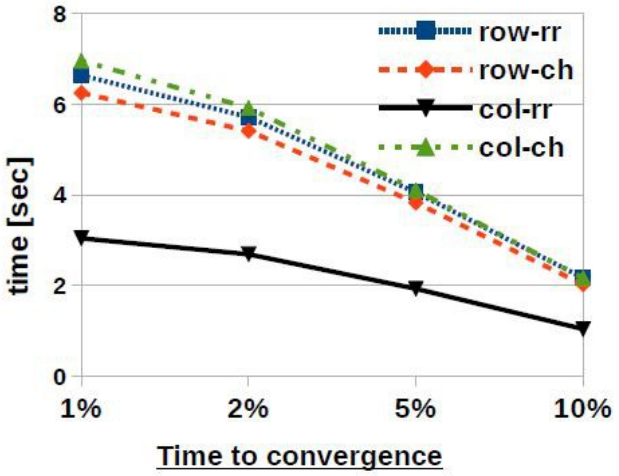
Data Access Path
column-major chunking (col-ch)



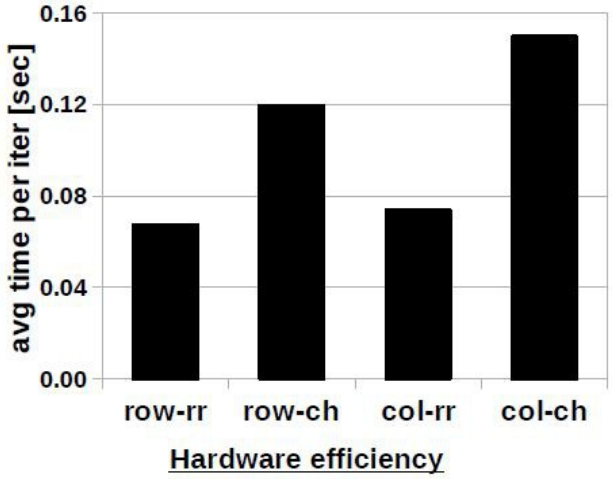
Data Access Path Study

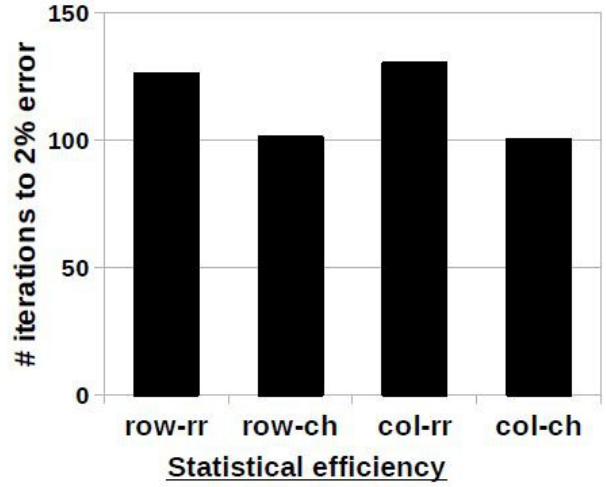


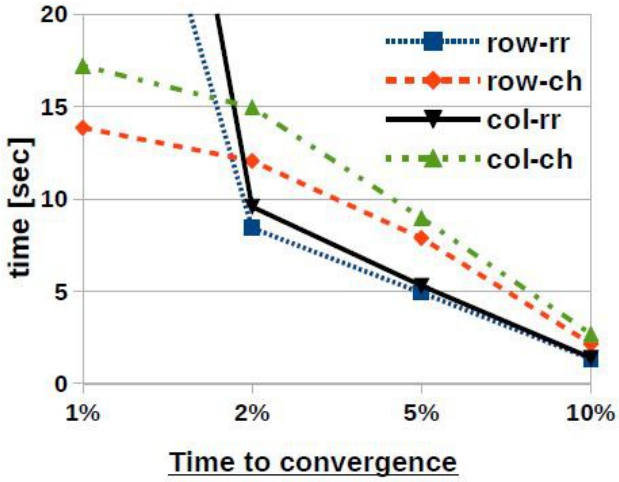




dense

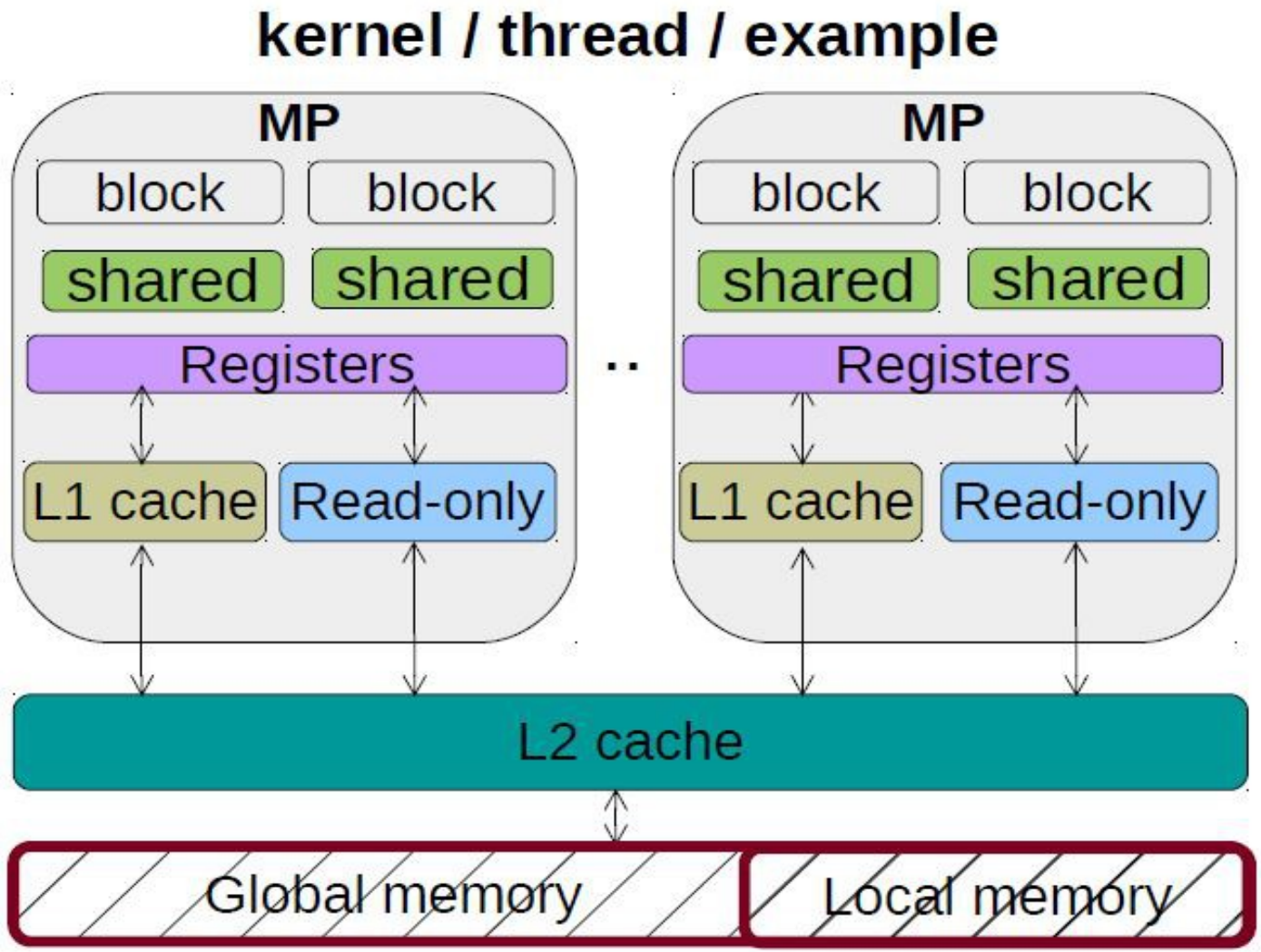




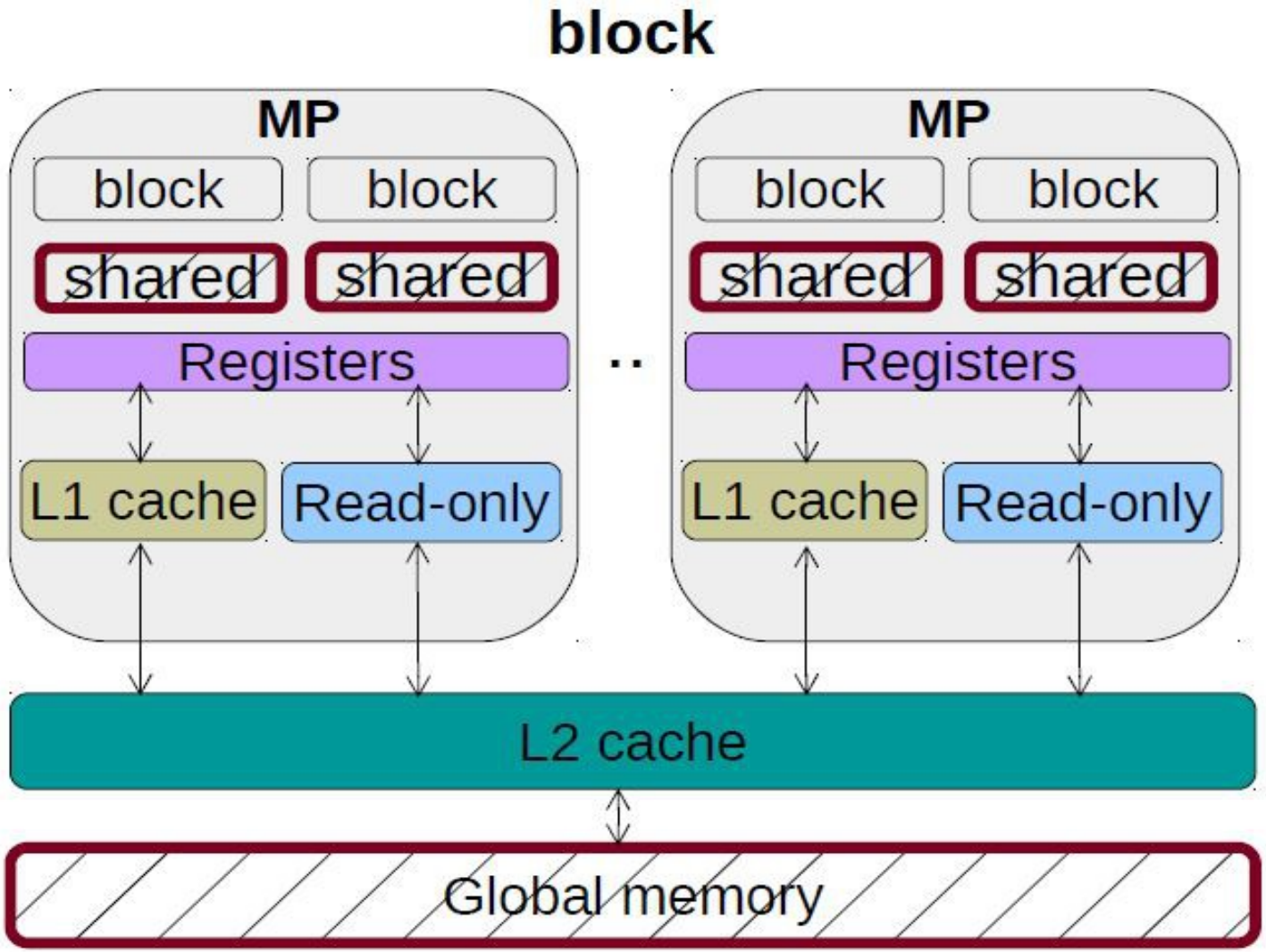


sparse

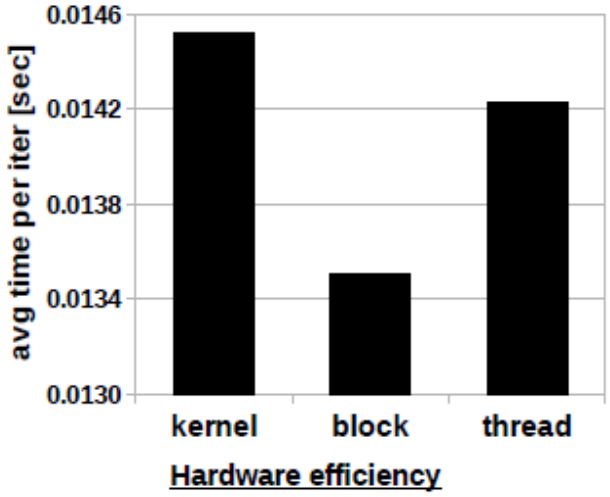
Model Replication

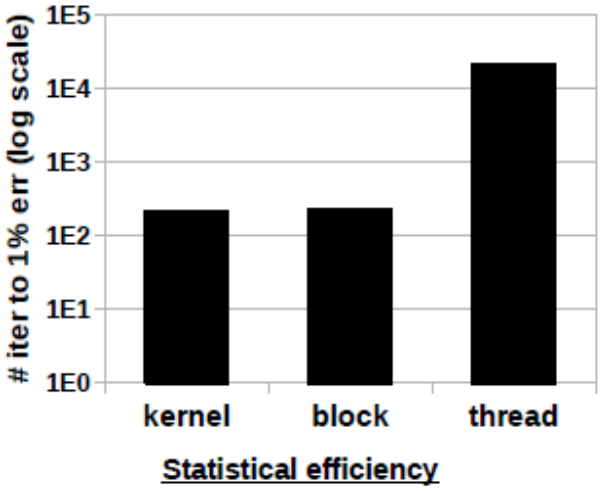


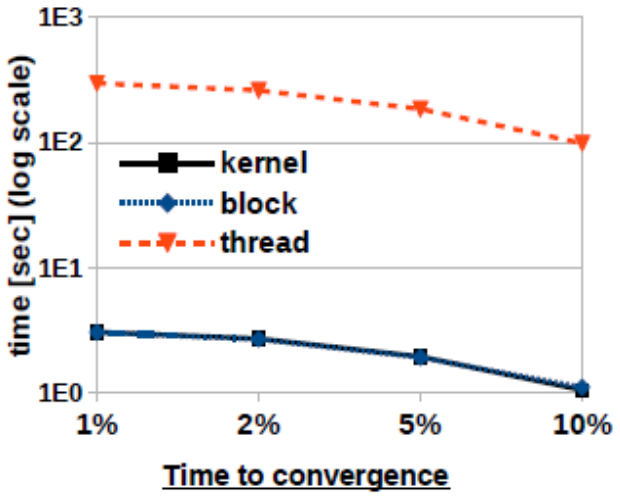
Model Replication



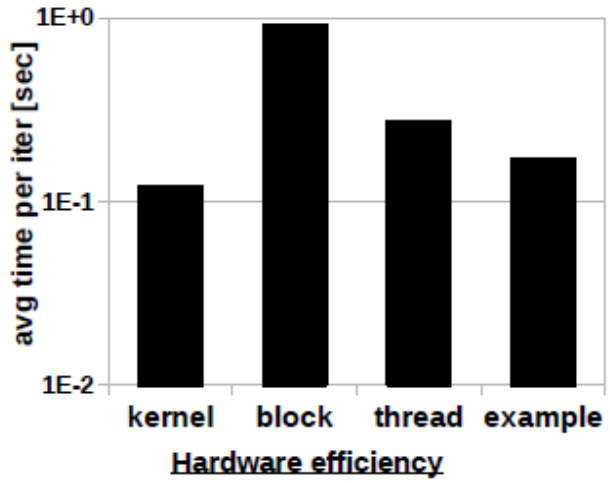
Model Replication Study

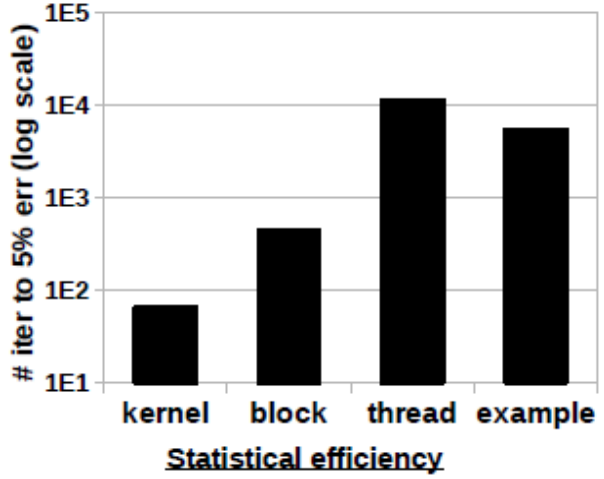


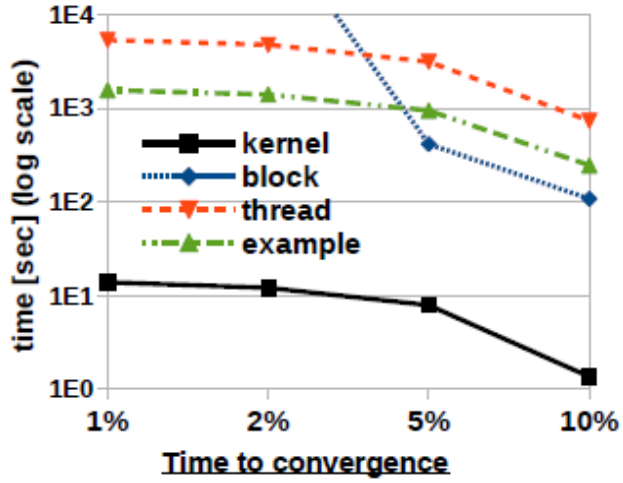




dense

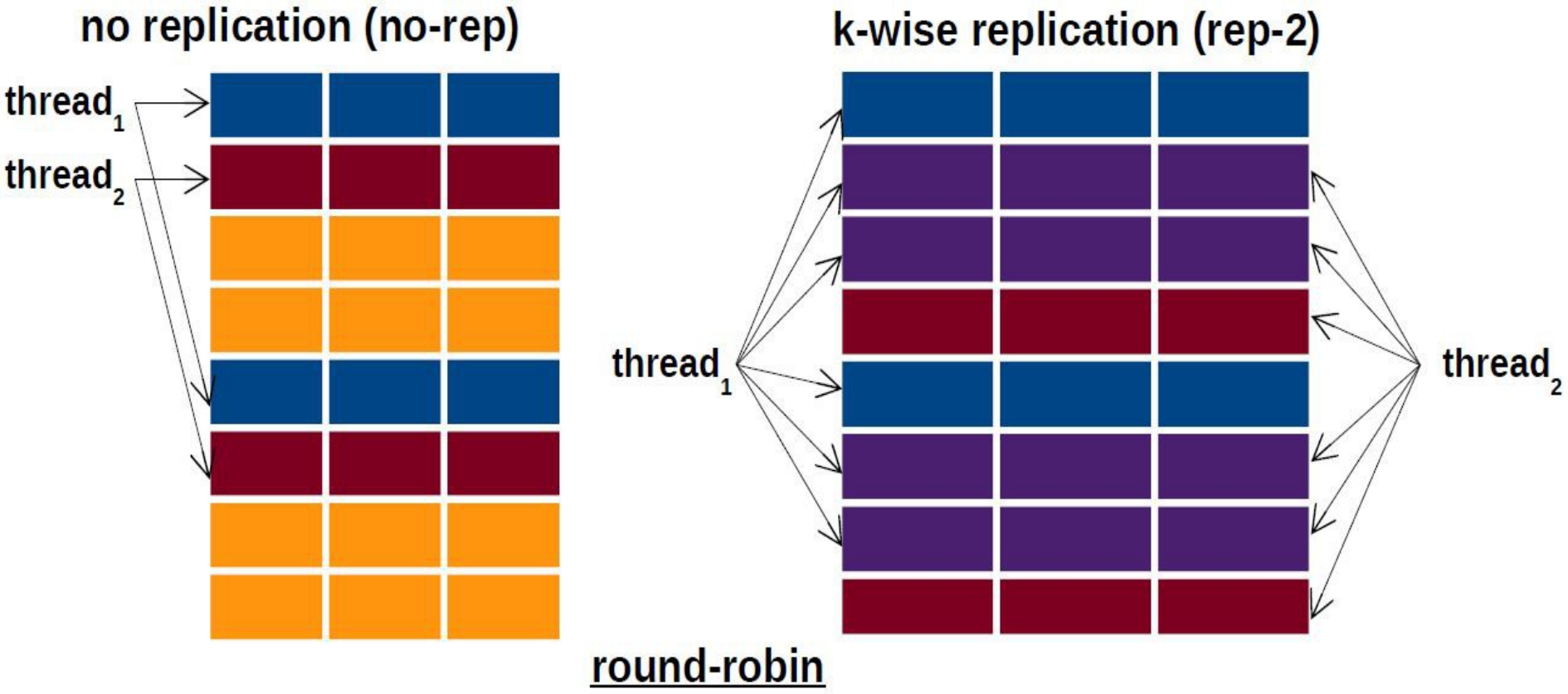




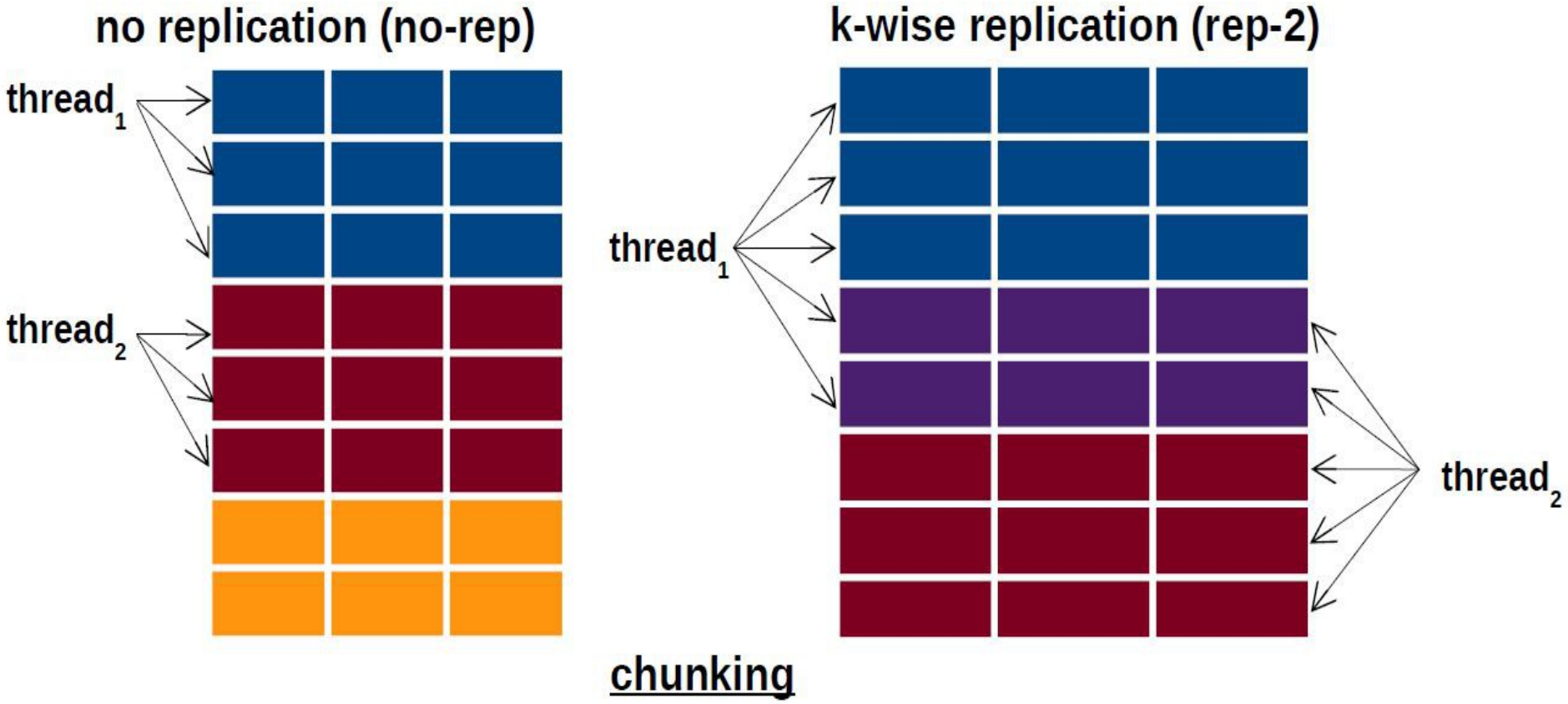


sparse

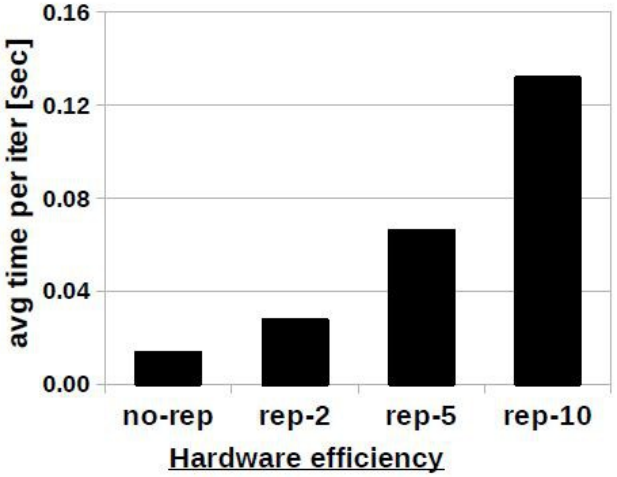
Data Replication

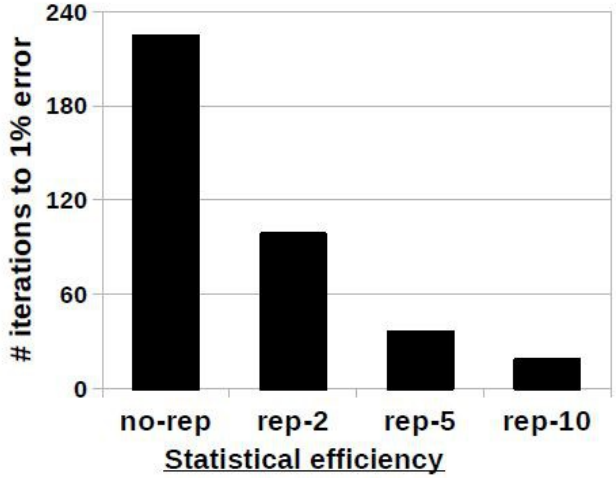


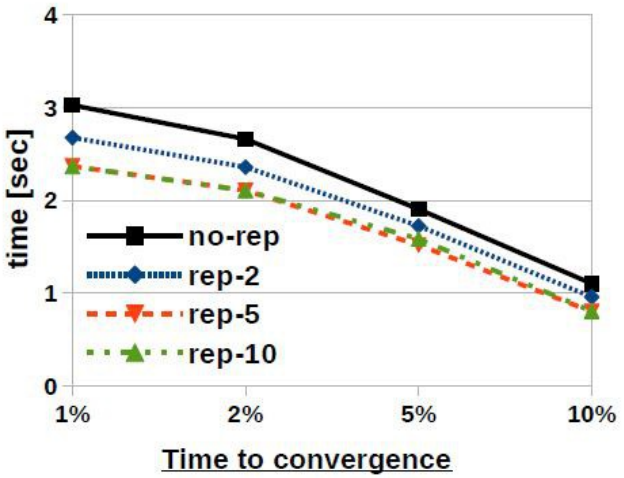
Data Replication



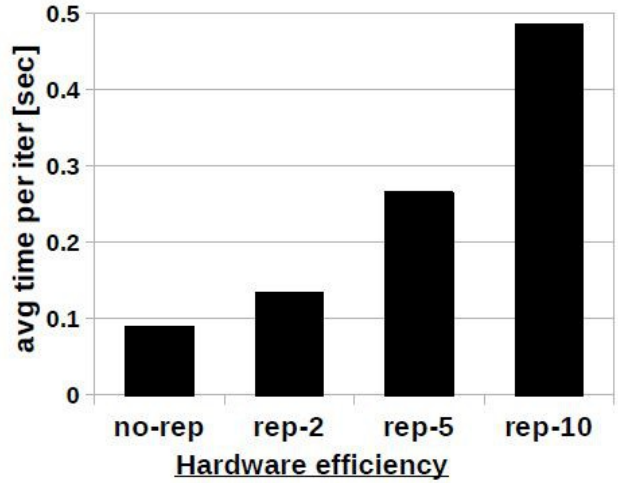
Data Replication Study

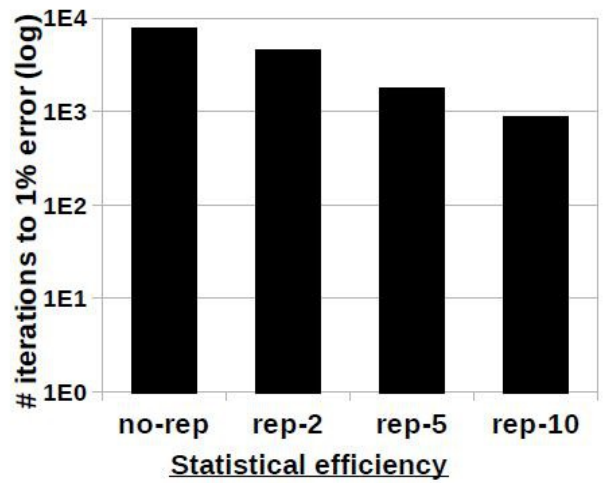


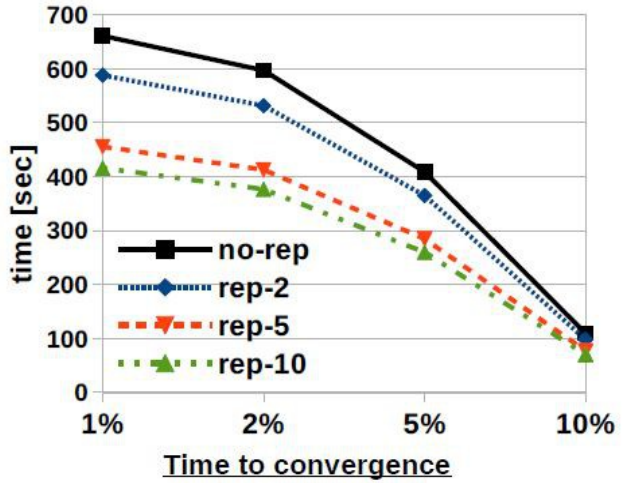




dense







sparse

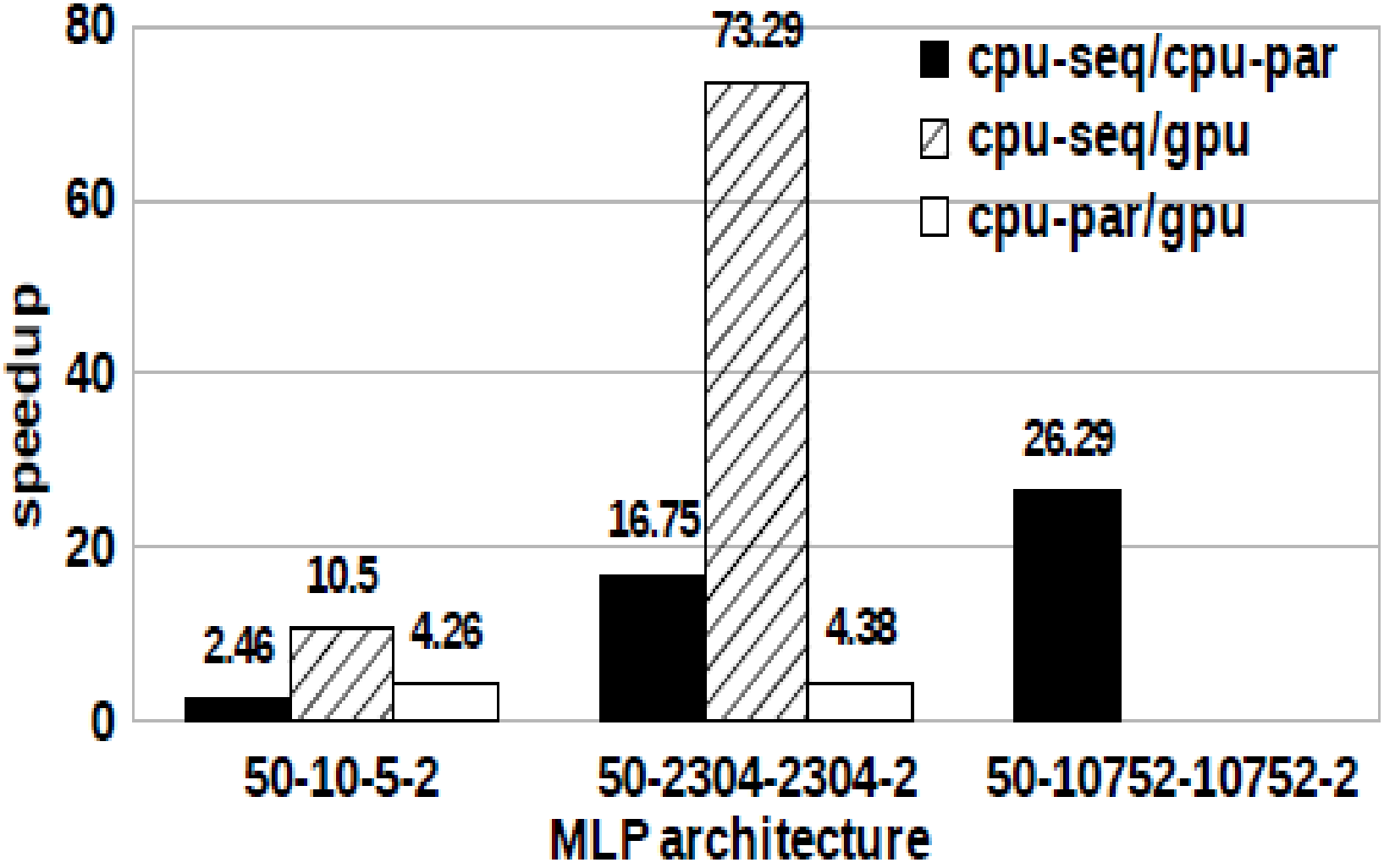
Asynchronous SGD Results

task	dataset	time-to-convergence (sec)			time-per-iteration (msec)			epochs			speedup	
		gpu	cpu-seq	cpu-par	gpu	cpu-seq	cpu-par	gpu	cpu-seq	cpu-par	cpu-seq/cpu-par	gpu/cpu-par
LR	covtype	1.97	0.60	1.51	15	150	251	135	4	6	0.60	0.06
	w8a	0.22	0.27	0.18	2.8	15	5.9	80	18	27	2.54	0.47
	real-sim	2.48	1.35	0.52	27	25	8.1	92	54	61	3.09	3.33
	rcv1	18.29	20.37	4.64	226	345	71	81	59	65	4.86	3.18
	news	∞	5.47	∞	65	53	8.7	∞	103	∞	6.09	7.47
SVM	covtype	0.96	0.16	0.35	15	53	77	63	3	4	0.69	0.19
	w8a	∞	0.54	1.89	2.6	2.2	5.6	∞	239	333	0.39	1.18
	real-sim	3.46	1.82	1.28	14	11	7.6	247	164	166	1.45	1.84
	rcv1	10.25	22.71	7.57	94	216	68	109	105	111	3.18	1.38
	news	∞	20.01	1.79	50	47	8.4	∞	425	211	5.60	5.95
MLP	covtype	2,106	6,365	288	6,056	19,058	814	344	334	354	23.42	7.44
	w8a	495	1,284	986	635	1,668	92.61	776	770	10,635	18.01	6.85
	real-sim	140	317	11.14	715	1,925	107	196	165	108	18.04	6.70
	rcv1	352	724	34.47	8,326	17,234	858	42	42	40	20.08	9.70
	news	18.25	47.35	1.12	234	512	34.04	78	91	32	15.06	6.87

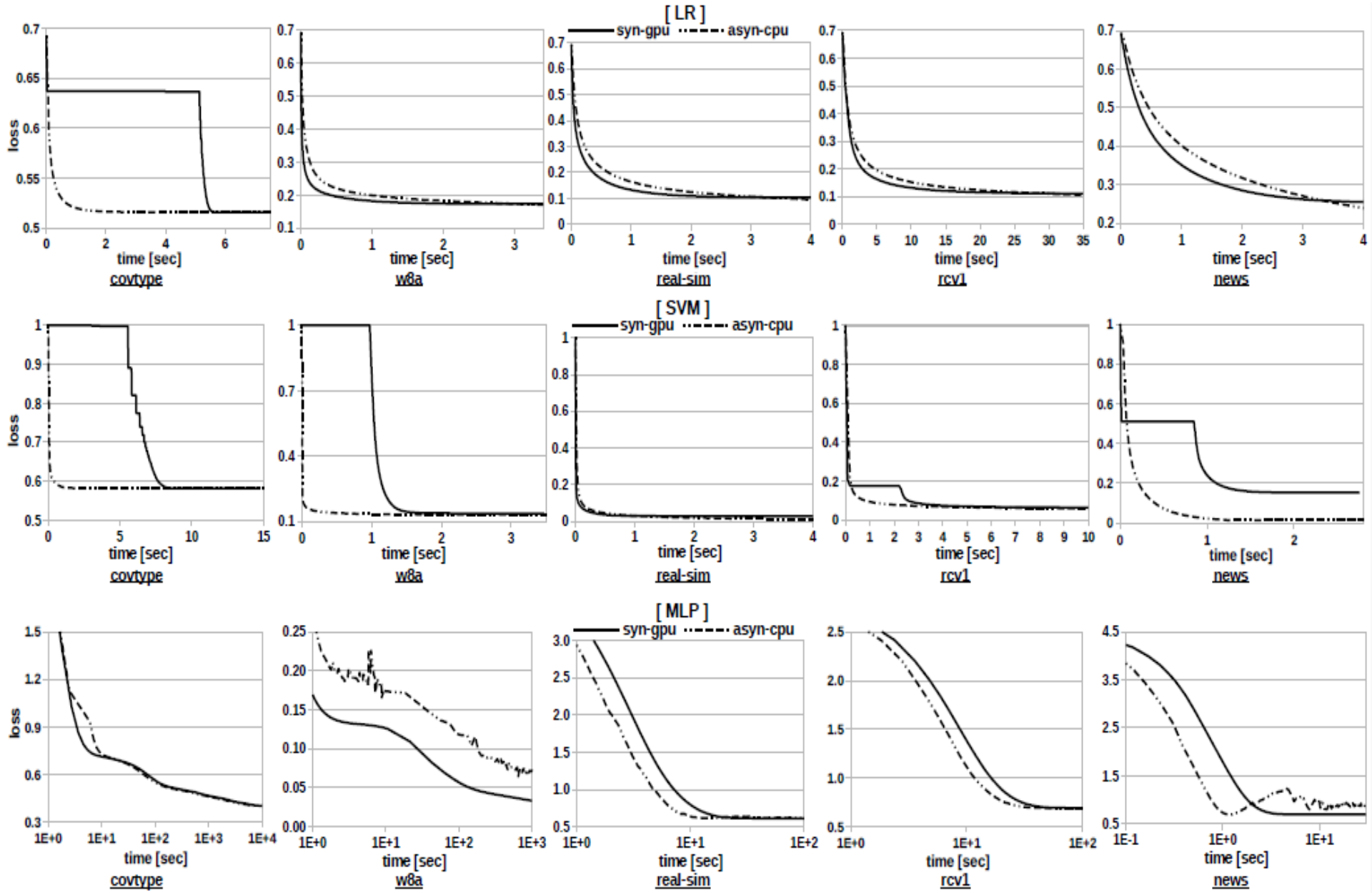
GPU Hogwild! Summary

task	dataset	optimal configuration
LR	covtype	col-rr + block + no-rep
	w8a	row-rr + kernel + rep-10
	real-sim	row-ch + kernel + rep-10
	rcv1	row-ch + kernel + no-rep
	news	row-rr + kernel + rep-10
SVM	covtype	col-rr + block + no-rep
	w8a	row-ch + kernel + rep-10
	real-sim	row-rr + kernel + rep-10
	rcv1	row-rr + kernel + rep-10
	news	row-rr + kernel + rep-10

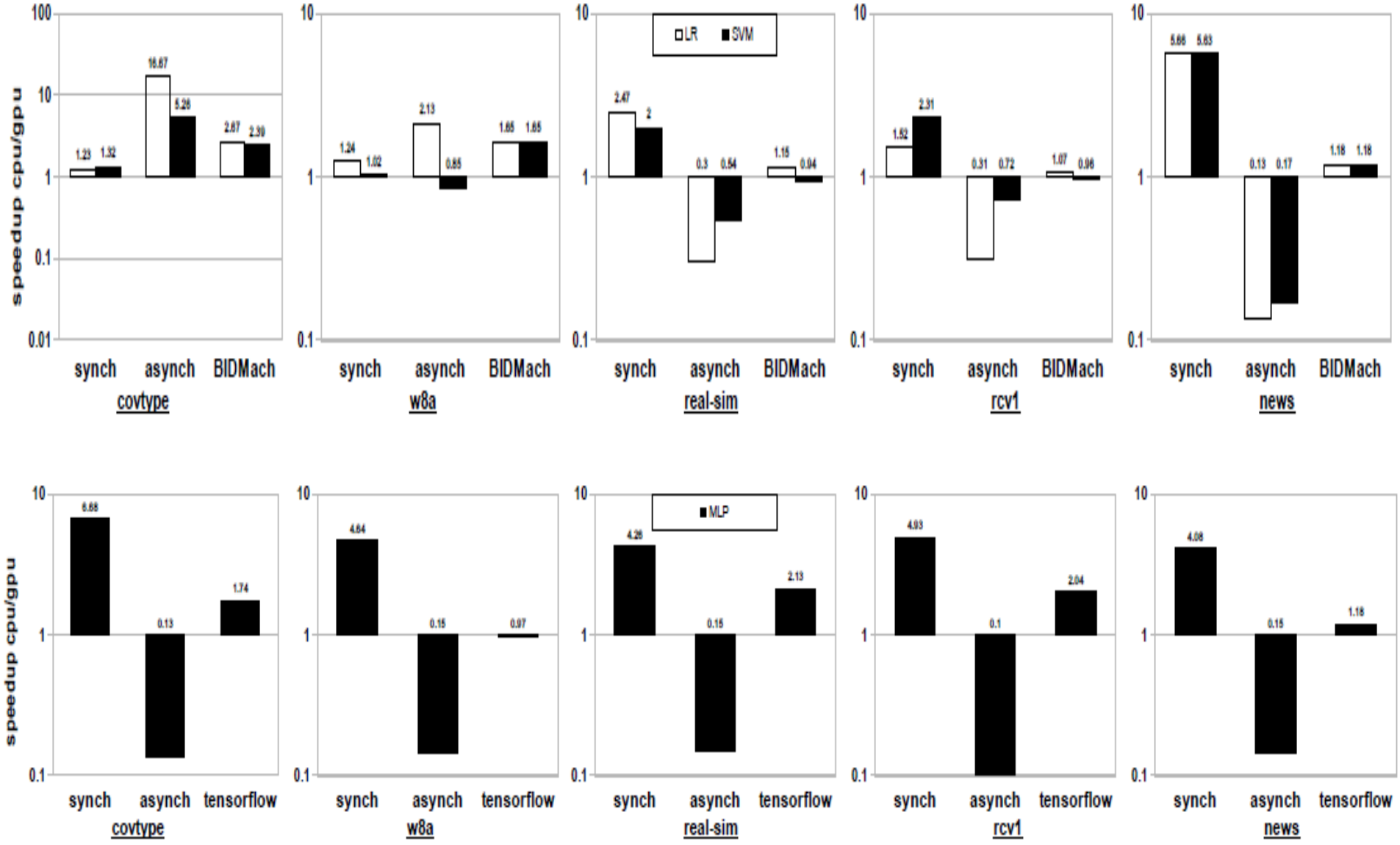
MLP Speedup (real-sim)



Synchronous GPU v Asynchronous CPU



Speedup CPU v GPU



Conclusions

- Synchronous SGD
  - GPU is always faster than parallel CPU in time to convergence
  - Gap is larger for MLP (5X on average)
  - Results are on par or better than TensorFlow and BIDMach
- Asynchronous SGD
  - CPU always outperforms GPU in time to convergence, even when GPU has a speedup larger than 10X in hardware efficiency
  - Gap is higher than 5X on sparse data and deep nets
- Synchronous GPU v Asynchronous CPU
  - The best is task- and dataset-dependent
  - CPU should not be discarded
  - GPU is more cost-effective alternative

Code:
https://github.com/YMA33/GradientDescent

Thank you.
Questions ???