Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning

***Zeke Wang* (王则可)**

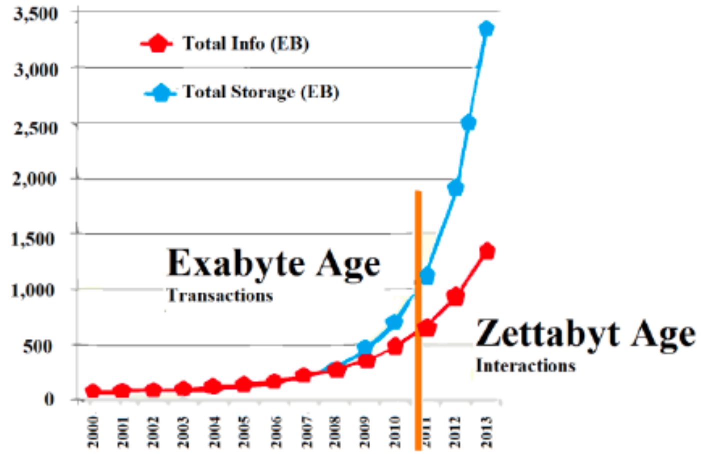
CCAI, Computer Science, Zhejiang University

The world is moving
in three directions

Motivations

Big Data

Larger and More Complicated



Machine Learning

Exciting New Techniques

Linear Model
Logistic Regression
SVM

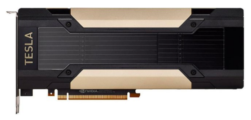
Neural Networks

Tree-based Models

…

New Hardware

The Dying of Moore's Law



GPU

TPU

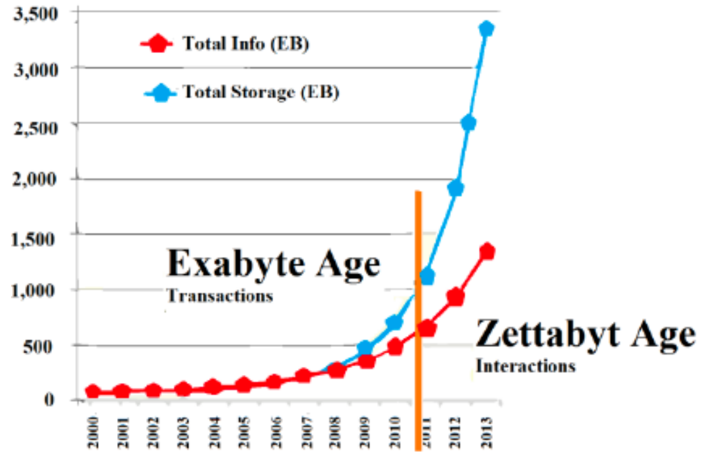
FPGA

__My work__: intersection of these three directions

Motivations

Big Data

Larger and More Complicated



Machine Learning

Exciting New Techniques

Linear Model
Logistic Regression
SVM

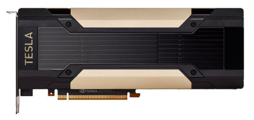
Neural Networks

Tree-based Models

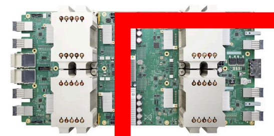
...

New Hardware

The Dying of Moore's Law



GPU



TPU

FPGA

***My work**: intersection of these three directions*

This Work

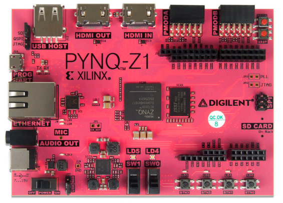
Motivations

Database

Generalized Linear Model

Linear Model, Logistic Regression, SVM

FPGA



?

Can We Use FPGAs to Accelerate GLM Training?

!

Yes, up to 11x, compared with the fastest CPU implementation we know.

=

Key Idea: Software/Hardware Co-Design

ML: Low-precision Training

+

DB: New Data Structure, optimized to bit-level

+

FPGA: Efficient Design

Outline of MLWeaving

Quick Background

Stochastic Gradient Descent (SGD)

Synchronous vs. Asynchronous

Low Precision

MLWeaving

Arbitrary-precision Training

MLWeaving Memory Layout

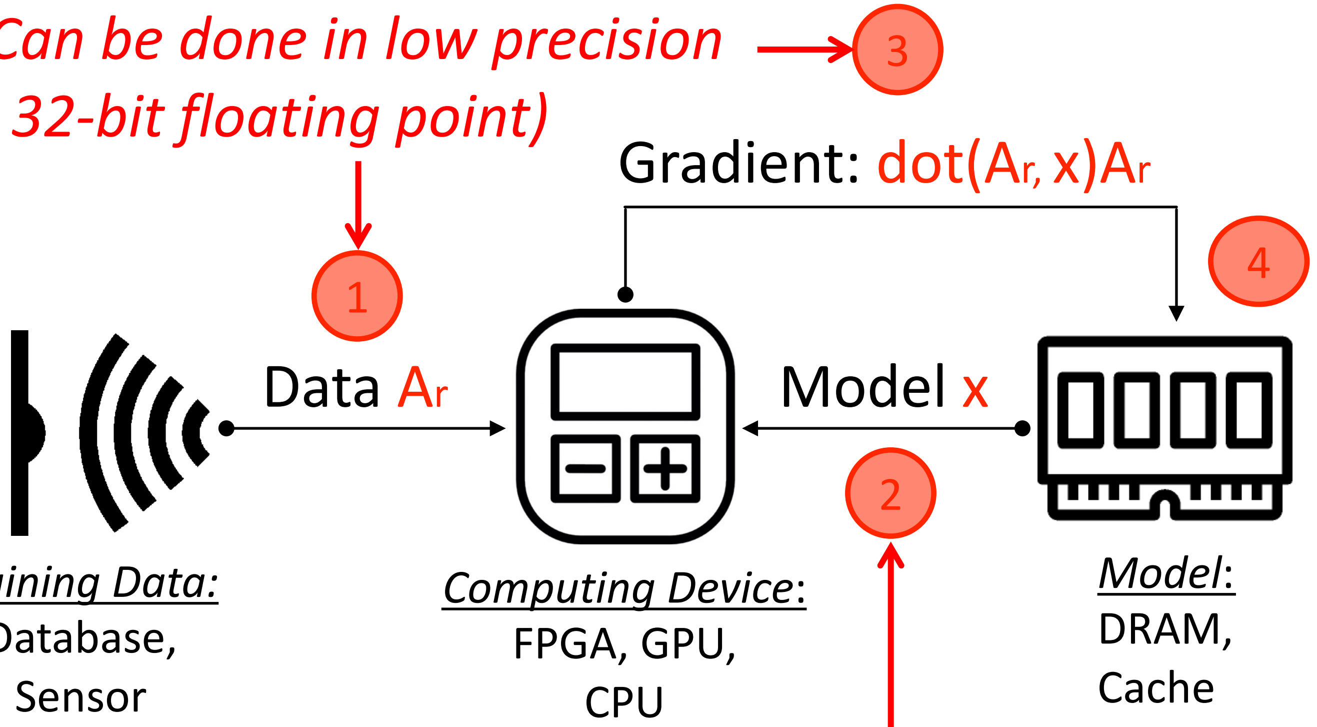
MLWeaving Hardware Design

Efficient Synchronous Design

OK, how does SGD work?

Stochastic Gradient Descent (SGD)

P2: Can be done in low precision (not 32-bit floating point)



P1: Model can be staled, especially when running on multiple cores.

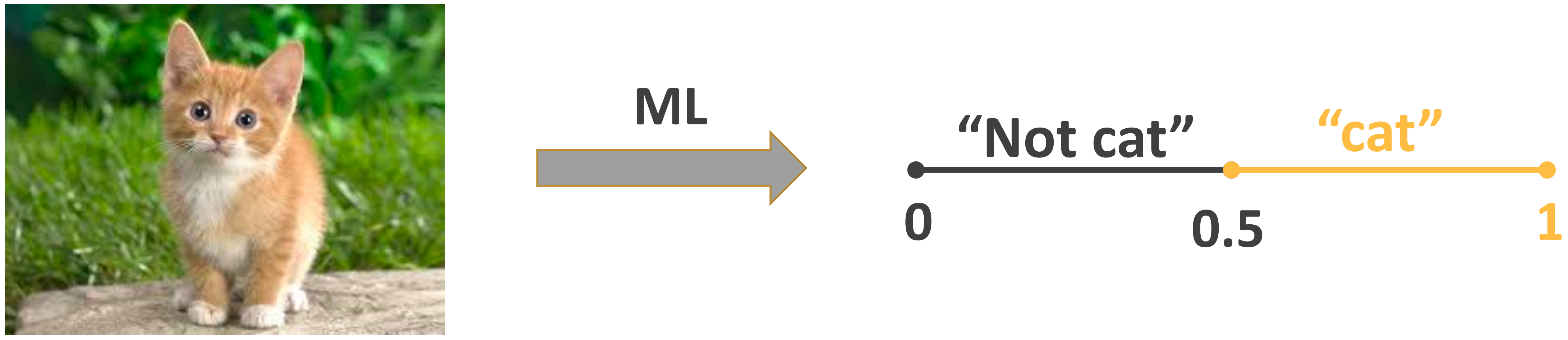
Linear Regression

$$\min_x \frac{1}{2}\sum_r (A_r x^T - b_r)^2$$

```
A_r = get_data()
x = get_model()
g = comp_grad(x,A_r)
x = x - g
set_model(x)
```

Two Interesting Properties

Intuition: Why Low Precision Works for ML



Intuition: Why Low Precision Works for ML

→ "It is a cat" (>0.5)

Full precision	*Low precision*
1.310245	about 1.3
X 0.602069	X about 0.6
0.788857897	about 0.78

Relax, It is only Machine Learning.

Different Precision Levels are Required

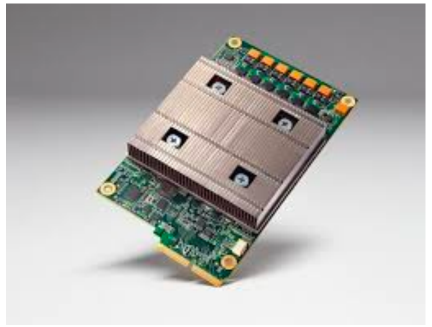
“It is a cat”

3-bit

“It is a cat”

9-bit

Current Hardware Supports Limited Precision Levels

CPU	GPU	TPU
		
Char (8-bit), Short (16-bit)	FP8 (8-bit), FP16 (16-bit)	INT8 (8-bit)

Goal of MLWeaving

?

For Generalized Linear Model training, can we enable things that cannot be well done on CPUs?

!

Any-precision Training

High-throughput Sync. Design

Outline

Quick Background

Stochastic Gradient Descent (SGD)

Synchronous vs. Asynchronous

Low Precision

MLWeaving

Arbitrary-precision Training

MLWeaving Memory Layout

MLWeaving Hardware Design

Efficient Synchronous Design

Two Goals of Arbitrary-precision Training: Using First Principles Thinking

1, One hardware design and one copy of dataset support any-precision training.

2, Our design achieves linear speedup with lower precision.

Outline

Quick Background

Stochastic Gradient Descent (SGD)

Synchronous vs. Asynchronous

Low Precision

MLWeaving

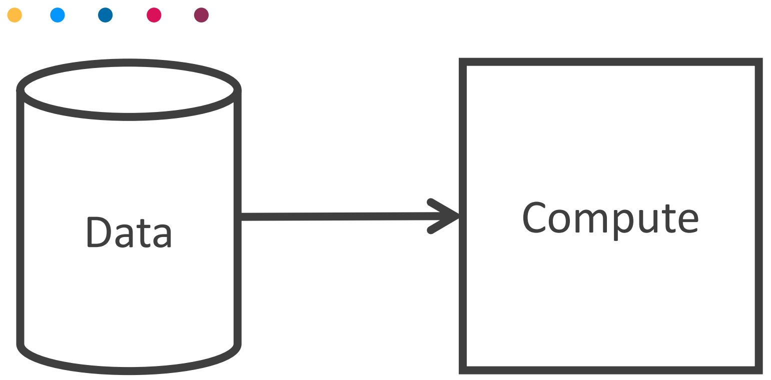
Arbitrary-precision Training

MLWeaving Memory Layout

MLWeaving Hardware Design

Efficient Synchronous Design

MLWeaving Memory Layout



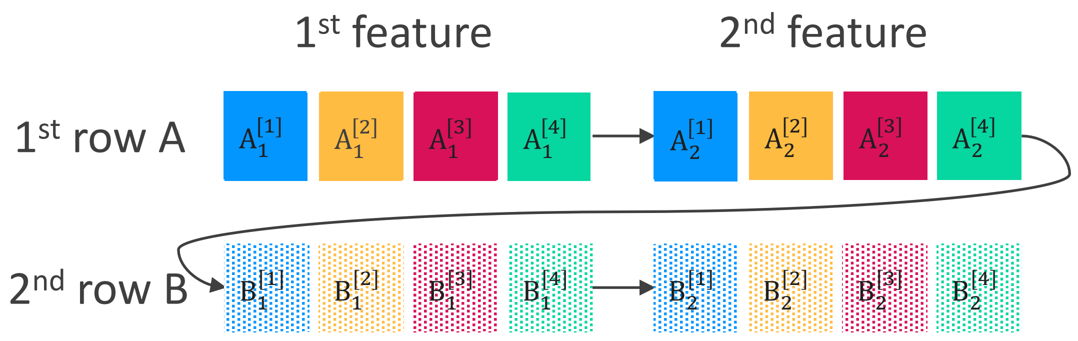
Observation 1:
Often memory bandwidth bound

Observation 2: Low precision (e.g., 8 bit fixed point) often provides reasonable quality

Observation 3: Different training task might need different precision level even on the same dataset

Can we store the data in a new data structure that efficiently supports arbitrary precision data movement?

How most systems store ML data today:

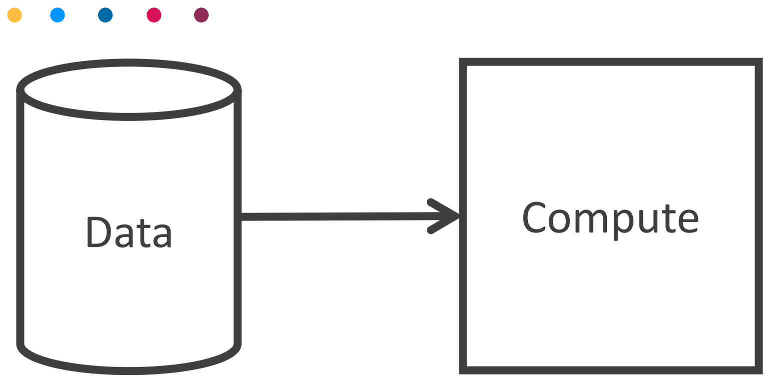


1st feature
2nd feature
1st row A
2nd row B
$A_1^{[1]}$ $A_1^{[2]}$ $A_1^{[3]}$ $A_1^{[4]}$ $A_2^{[1]}$ $A_2^{[2]}$ $A_2^{[3]}$ $A_2^{[4]}$
$B_1^{[1]}$ $B_1^{[2]}$ $B_1^{[3]}$ $B_1^{[4]}$ $B_2^{[1]}$ $B_2^{[2]}$ $B_2^{[3]}$ $B_2^{[4]}$

MLWeaving:

1st row A

MLWeaving Memory Layout



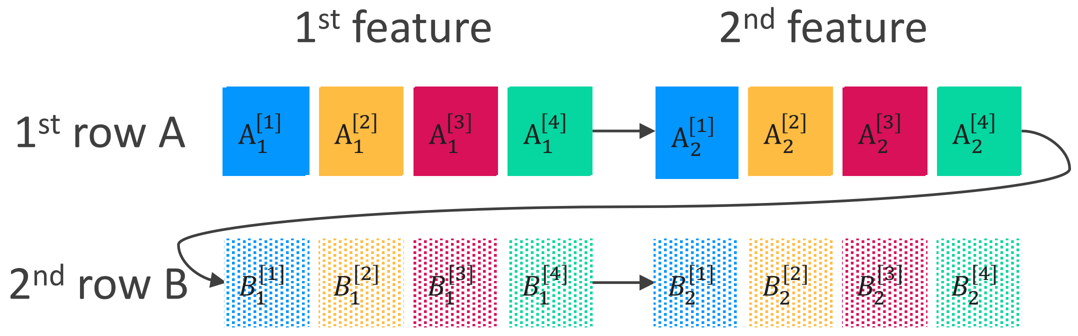
Observation 1:
Often memory bandwidth bound

Observation 2: Low precision (e.g., 8 bit fixed point) often provides reasonable quality

Observation 3: Different training task might need different precision level even on the same dataset

Can we store the data in a new data structure that efficiently supports arbitrary precision data movement?

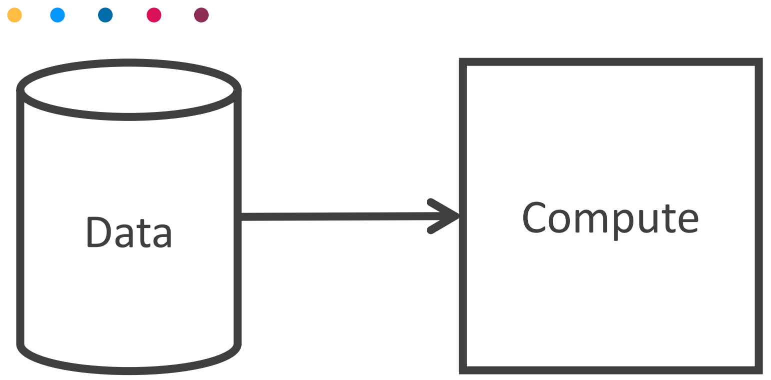
How most systems store ML data today:


1st feature
2nd feature
1st row A
2nd row B
$A_1^{[1]}$ $A_1^{[2]}$ $A_1^{[3]}$ $A_1^{[4]}$
$A_2^{[1]}$ $A_2^{[2]}$ $A_2^{[3]}$ $A_2^{[4]}$
$B_1^{[1]}$ $B_1^{[2]}$ $B_1^{[3]}$ $B_1^{[4]}$
$B_2^{[1]}$ $B_2^{[2]}$ $B_2^{[3]}$ $B_2^{[4]}$

MLWeaving:

1st row A

MLWeaving Memory Layout



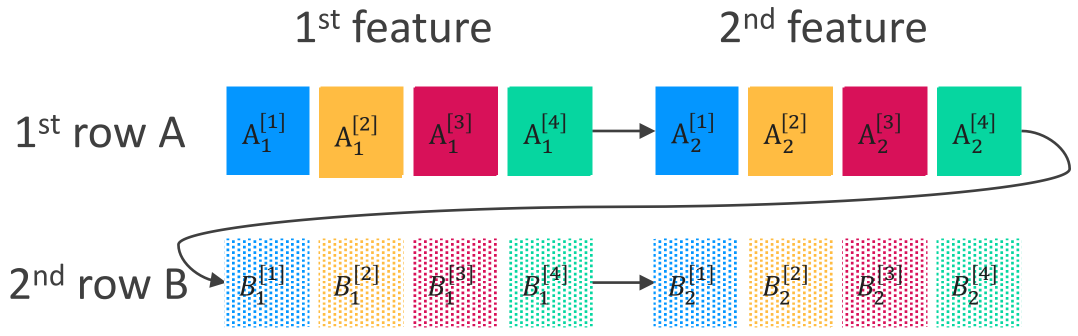
Observation 1:
Often memory bandwidth bound

Observation 2: Low precision (e.g., 8 bit fixed point) often provides reasonable quality

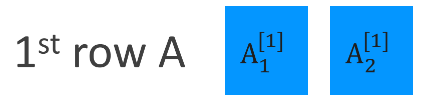
Observation 3: Different training task might need different precision level even on the same dataset

Can we store the data in a new data structure that supports arbitrary precision data movement?

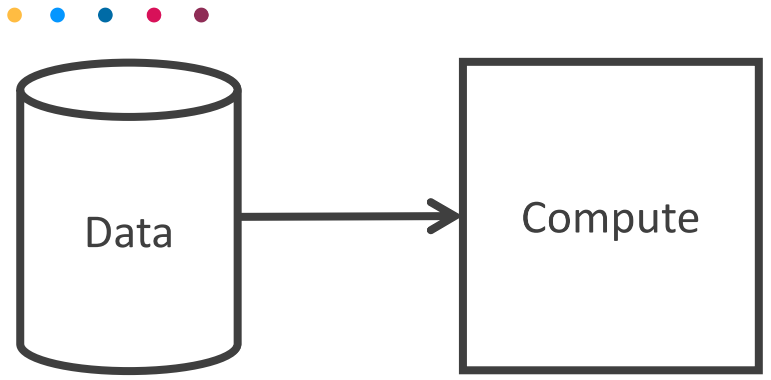
How most systems store ML data today:


1st feature
2nd feature
1st row A
$A_1^{[1]}$ $A_1^{[2]}$ $A_1^{[3]}$ $A_1^{[4]}$ $A_2^{[1]}$ $A_2^{[2]}$ $A_2^{[3]}$ $A_2^{[4]}$
2nd row B
$B_1^{[1]}$ $B_1^{[2]}$ $B_1^{[3]}$ $B_1^{[4]}$ $B_2^{[1]}$ $B_2^{[2]}$ $B_2^{[3]}$ $B_2^{[4]}$

MLWeaving:



MLWeaving Memory Layout



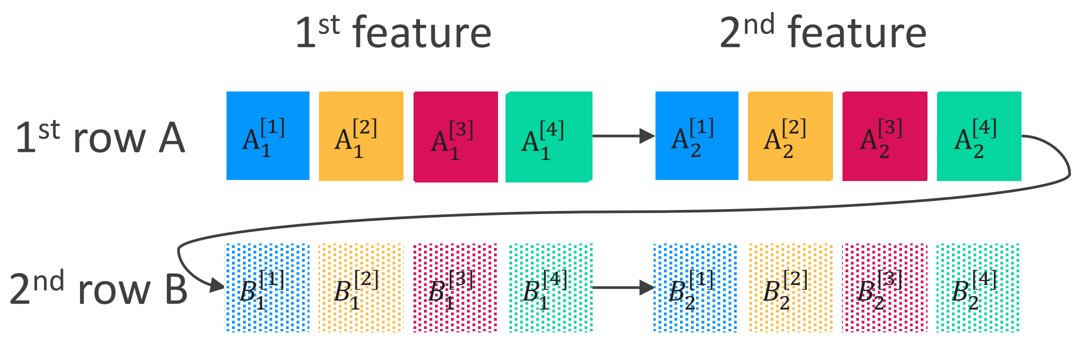
*Observation 1:
Often memory bandwidth bound*

Observation 2: Low precision (e.g., 8 bit fixed point) often provides reasonable quality

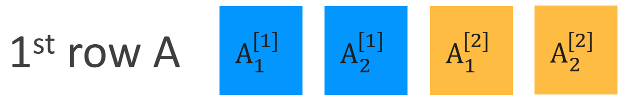
Observation 3: Different training task might need different precision level even on the same dataset

Can we store the data in a new data structure that supports arbitrary precision data movement?

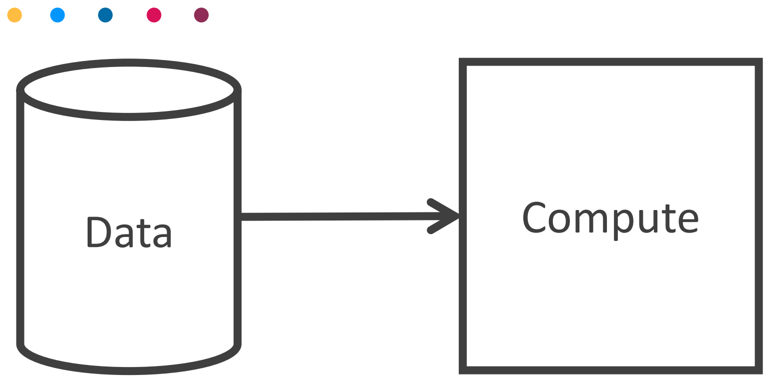
How most systems store ML data today:


1st feature
2nd feature
1st row A
2nd row B
$A_1^{[1]}$ $A_1^{[2]}$ $A_1^{[3]}$ $A_1^{[4]}$ $A_2^{[1]}$ $A_2^{[2]}$ $A_2^{[3]}$ $A_2^{[4]}$
$B_1^{[1]}$ $B_1^{[2]}$ $B_1^{[3]}$ $B_1^{[4]}$ $B_2^{[1]}$ $B_2^{[2]}$ $B_2^{[3]}$ $B_2^{[4]}$

MLWeaving:



MLWeaving Memory Layout



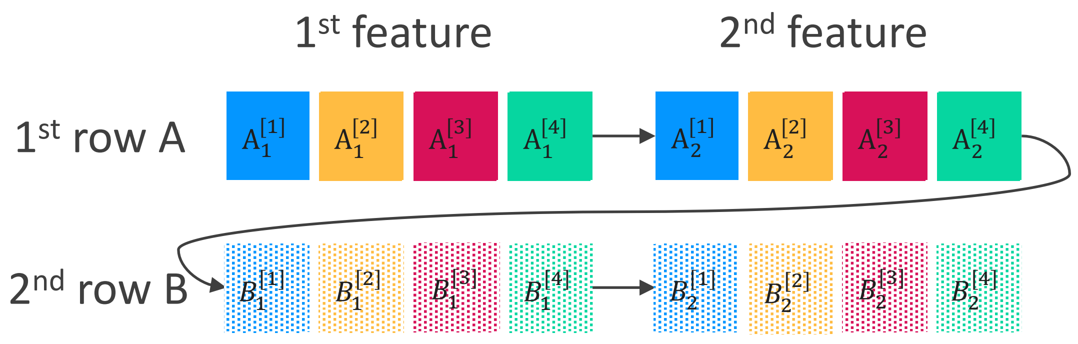
Observation 1:
Often memory bandwidth bound

Observation 2: Low precision (e.g., 8 bit fixed point) often provides reasonable quality

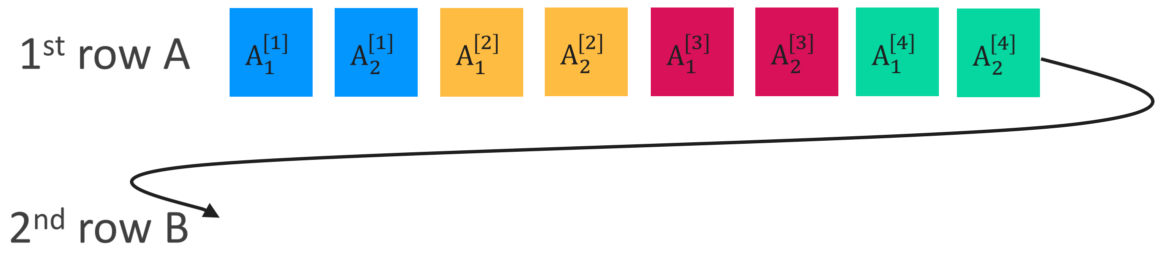
Observation 3: Different training task might need different precision level even on the same dataset

Can we store the data in a new data structure that supports arbitrary precision data movement?

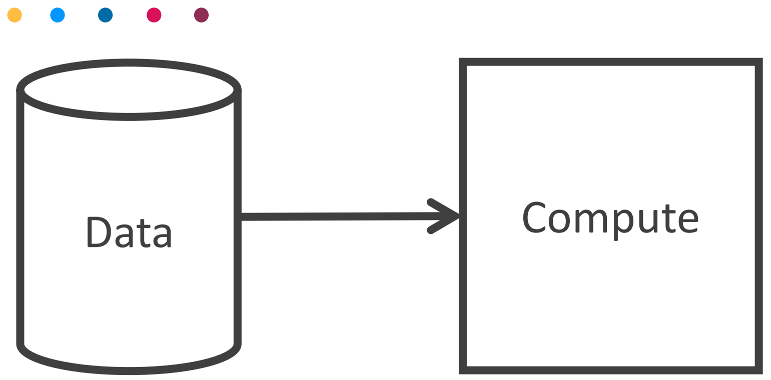
How most systems store ML data today:


1st feature
2nd feature
1st row A
$A_1^{[1]}$ $A_1^{[2]}$ $A_1^{[3]}$ $A_1^{[4]}$ $A_2^{[1]}$ $A_2^{[2]}$ $A_2^{[3]}$ $A_2^{[4]}$
2nd row B
$B_1^{[1]}$ $B_1^{[2]}$ $B_1^{[3]}$ $B_1^{[4]}$ $B_2^{[1]}$ $B_2^{[2]}$ $B_2^{[3]}$ $B_2^{[4]}$

MLWeaving:


1st row A
$A_1^{[1]}$ $A_2^{[1]}$ $A_1^{[2]}$ $A_2^{[2]}$ $A_1^{[3]}$ $A_2^{[3]}$ $A_1^{[4]}$ $A_2^{[4]}$
2nd row B

MLWeaving Memory Layout



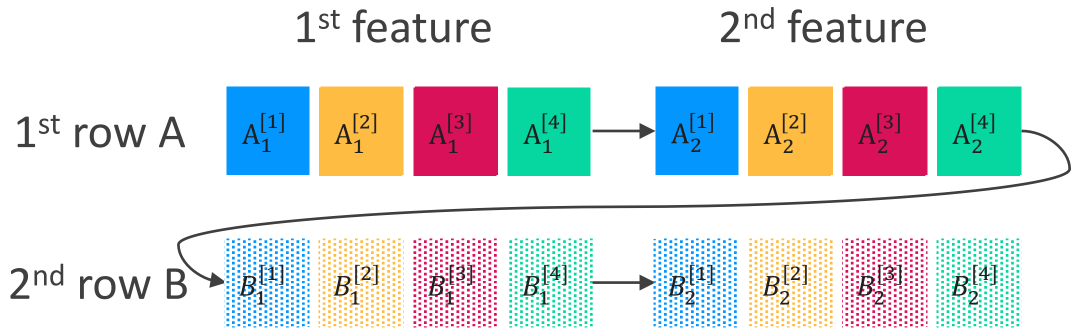
Observation 1:
Often memory bandwidth bound

Observation 2: Low precision (e.g., 8 bit fixed point) often provides reasonable quality

Observation 3: Different training task might need different precision level even on the same dataset

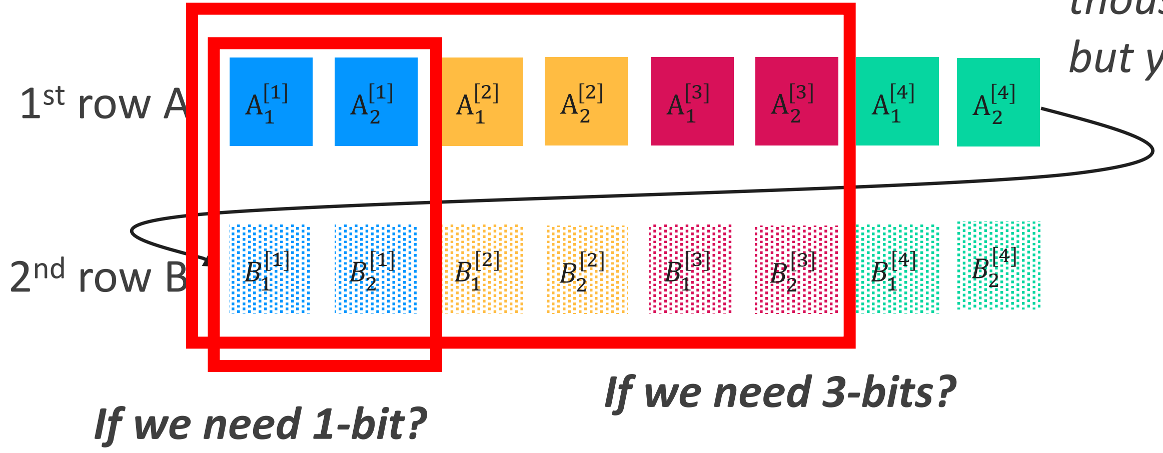
Can we store the data in a new data structure that supports arbitrary precision data movement?

How most systems store ML data today:


1st feature
2nd feature
1st row A
2nd row B
$A_1^{[1]}$ $A_1^{[2]}$ $A_1^{[3]}$ $A_1^{[4]}$ $A_2^{[1]}$ $A_2^{[2]}$ $A_2^{[3]}$ $A_2^{[4]}$
$B_1^{[1]}$ $B_1^{[2]}$ $B_1^{[3]}$ $B_1^{[4]}$ $B_2^{[1]}$ $B_2^{[2]}$ $B_2^{[3]}$ $B_2^{[4]}$

More complicated when a row has thousands of features, but you get the idea.

MLWeaving:


1st row A
2nd row B
$A_1^{[1]}$ $A_2^{[1]}$ $A_1^{[2]}$ $A_2^{[2]}$ $A_1^{[3]}$ $A_2^{[3]}$ $A_1^{[4]}$ $A_2^{[4]}$
$B_1^{[1]}$ $B_2^{[1]}$ $B_1^{[2]}$ $B_2^{[2]}$ $B_1^{[3]}$ $B_2^{[3]}$ $B_1^{[4]}$ $B_2^{[4]}$
If we need 1-bit?
If we need 3-bits?

***MLWeaving does not work out on CPUs**. CPU does not have custom instruction for MLWeaving memory layout and then we have to group bits from different memory locations before the further computing.*

Outline

Quick Background

<u>Stochastic Gradient Descent (SGD)</u>

Synchronous vs. Asynchronous

Low Precision

MLWeaving

<u>Arbitrary-precision Training</u>

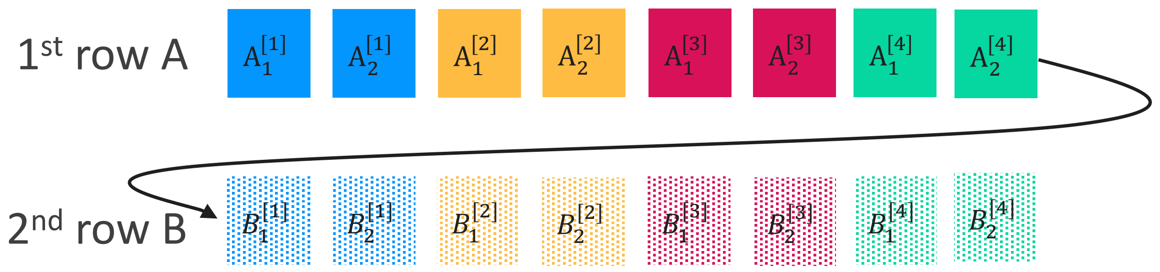
MLWeaving Memory Layout

MLWeaving Hardware Design

<u>Efficient Synchronous Design</u>

MLWeaving Hardware Design: Key Idea

MLWeaving memory layout:


1st row A
$A_1^{[1]}$ $A_2^{[1]}$ $A_1^{[2]}$ $A_2^{[2]}$ $A_1^{[3]}$ $A_2^{[3]}$ $A_1^{[4]}$ $A_2^{[4]}$
2nd row B
$B_1^{[1]}$ $B_2^{[1]}$ $B_1^{[2]}$ $B_2^{[2]}$ $B_1^{[3]}$ $B_2^{[3]}$ $B_1^{[4]}$ $B_2^{[4]}$

Key idea of MLWeaving hardware design:

To use bit-serial multiplier to enable efficient data processing from the MLWeaving memory layout.

How bit-serial multiplier works?

How Bit-serial Multiplier Deals with Low Precision?

Each bit should be binary, but we use decimal for ease of understanding.

4-bit:

$$\begin{array}{r} 4\ 3\ 2\ 1 \\ \text{X}\ 0\ 0\ 2\ 0 \\ \hline 8\ 6\ 4\ 2\ 0 \end{array}$$

3-bit:

$$\begin{array}{r} 4\ 3\ 2\ 0 \\ \text{X}\ 0\ 0\ 2\ 0 \\ \hline 8\ 6\ 4\ 0\ 0 \end{array}$$

2-bit:

$$\begin{array}{r} 4\ 3\ 0\ 0 \\ \text{X}\ 0\ 0\ 2\ 0 \\ \hline 8\ 6\ 0\ 0\ 0 \end{array}$$

1-bit:

$$\begin{array}{r} 4\ 0\ 0\ 0 \\ \text{X}\ 0\ 0\ 2\ 0 \\ \hline 8\ 0\ 0\ 0\ 0 \end{array}$$

Normal Multiplier

How Bit-serial Multiplier Deals with Low Precision?

```
          4 3 2 1
4-bit:   X 0 0 2 0
        ----------
         8 6 4 2 0

          4 3 2 0
3-bit:   X 0 0 2 0
        ----------
         8 6 4 0 0

          4 3 0 0
2-bit:   X 0 0 2 0
        ----------
         8 6 0 0 0

          4 0 0 0
1-bit:   X 0 0 2 0
        ----------
         8 0 0 0 0
```

Normal Multiplier

Initialization:

4 3 2 1

X 0020 $\boxed{\textbf{BSM}}$

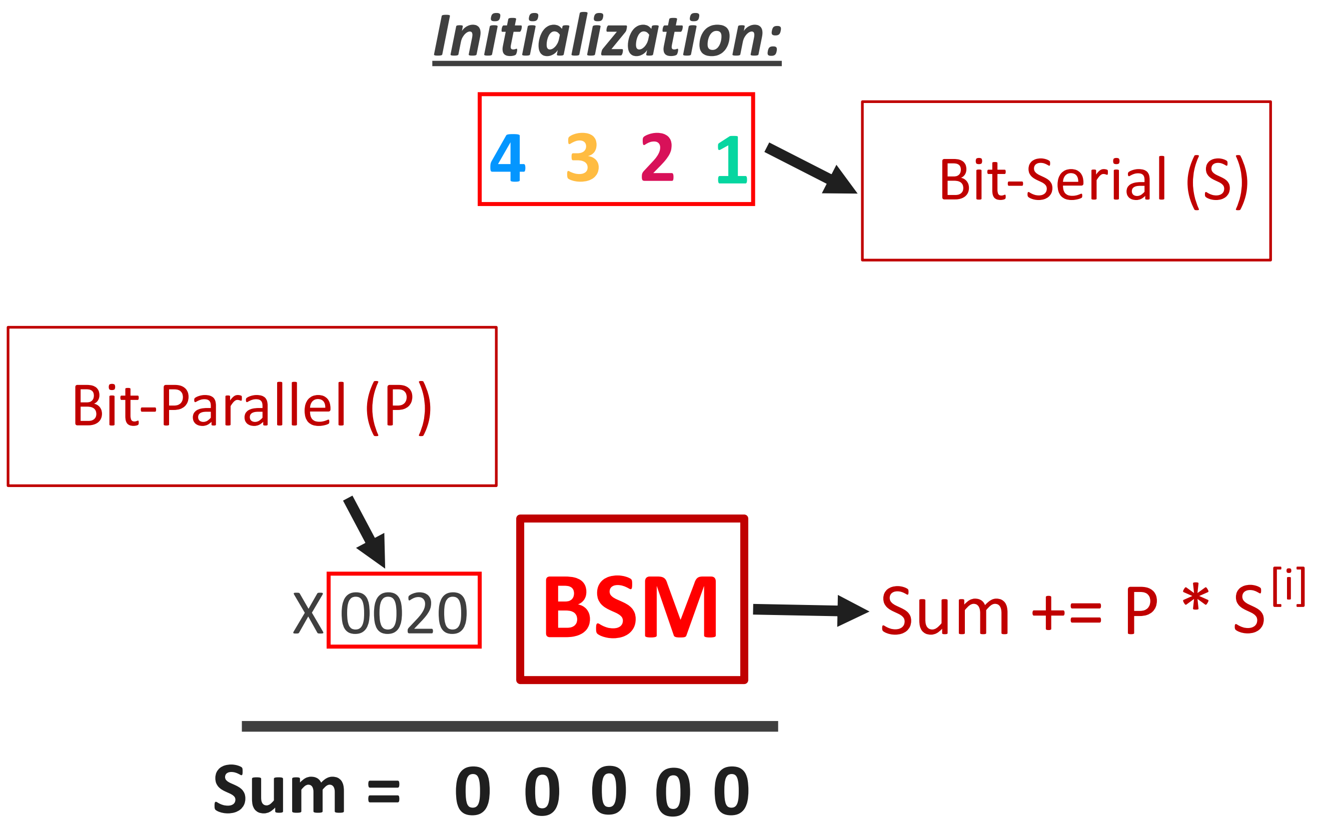
Sum = 0 0 0 0 0

Bit-serial Multiplier (BSM)

How Bit-serial Multiplier Deals with Low Precision?

```
           4 3 2 1
4-bit:   X 0 0 2 0
         ---------
         8 6 4 2 0

           4 3 2 0
3-bit:   X 0 0 2 0
         ---------
         8 6 4 0 0

           4 3 0 0
2-bit:   X 0 0 2 0
         ---------
         8 6 0 0 0

           4 0 0 0
1-bit:   X 0 0 2 0
         ---------
         8 0 0 0 0
```

Normal Multiplier

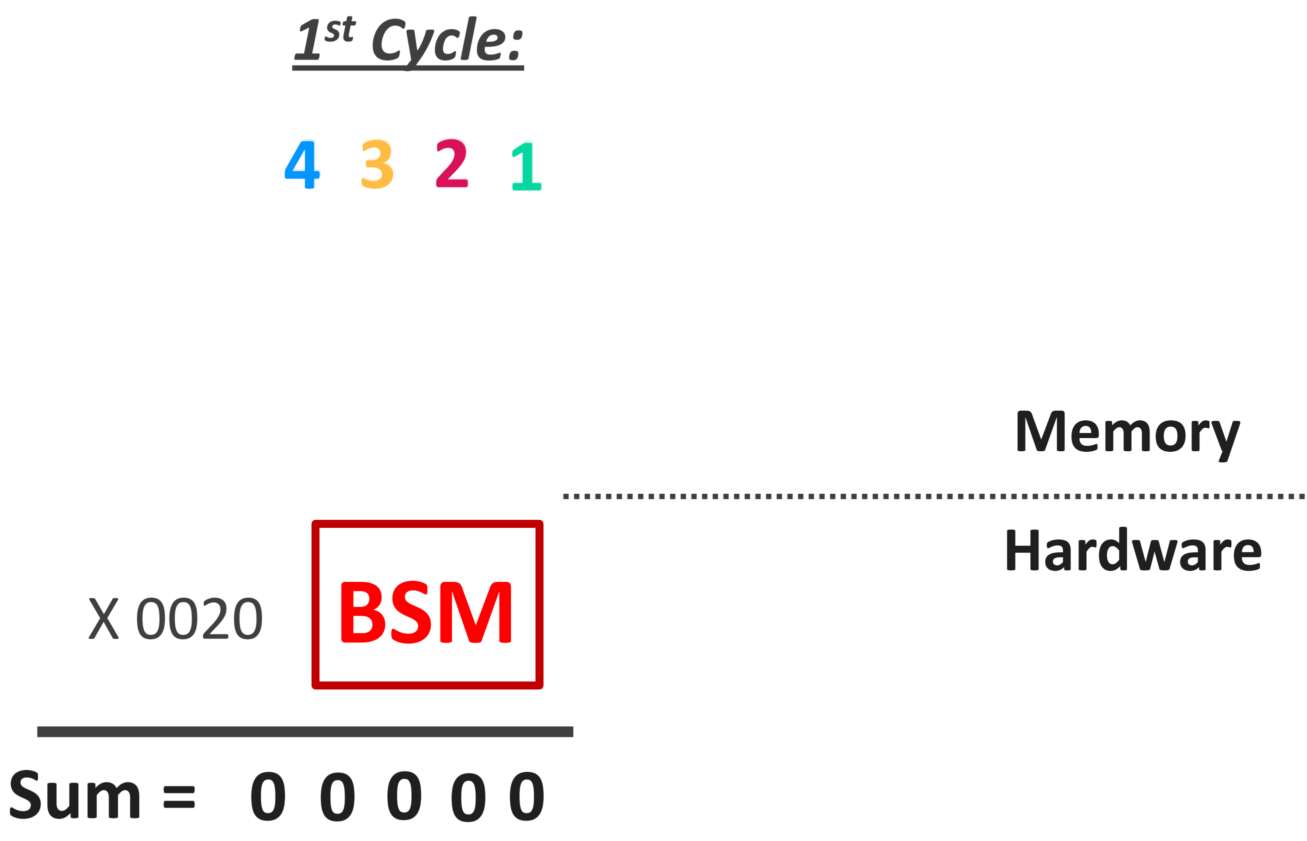


Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 1-Bit Precision

```
          4 3 2 1
4-bit:    X 0 0 2 0
          ---------
          8 6 4 2 0

          4 3 2 0
3-bit:    X 0 0 2 0
          ---------
          8 6 4 0 0

          4 3 0 0
2-bit:    X 0 0 2 0
          ---------
          8 6 0 0 0

          4 0 0 0
1-bit:    X 0 0 2 0
          ---------
          8 0 0 0 0
```

Normal Multiplier



Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 1-Bit Precision

4-bit:

```
   4 3 2 1
 X 0 0 2 0
 ---------
 8 6 4 2 0
```

3-bit:

```
   4 3 2 0
 X 0 0 2 0
 ---------
 8 6 4 0 0
```

2-bit:

```
   4 3 0 0
 X 0 0 2 0
 ---------
 8 6 0 0 0
```

1-bit:

```
   4 0 0 0
 X 0 0 2 0
 ---------
 8 0 0 0 0
```

Normal Multiplier

1st Cycle:

4 **3** **2** **1**

4 means 4000.

Memory

Hardware

X 0020 **4** **BSM**

Sum += 20 * 4000

Sum = 0 0 0 0 0

Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 1-Bit Precision

4-bit:

```
  4 3 2 1
X 0 0 2 0
---------
8 6 4 2 0
```

3-bit:

```
  4 3 2 0
X 0 0 2 0
---------
8 6 4 0 0
```

2-bit:

```
  4 3 0 0
X 0 0 2 0
---------
8 6 0 0 0
```

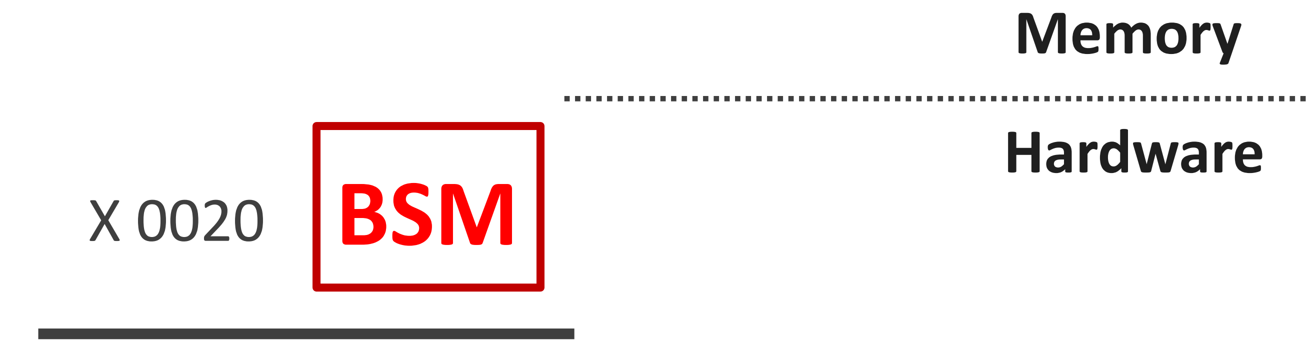
1-bit:

```
  4 0 0 0
X 0 0 2 0
---------
8 0 0 0 0
```

Normal Multiplier

1st Cycle:

4 3 2 1



Sum = 8 0 0 0 0

Done with 1-bit precision, or proceed to the next bit.

Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 2-Bit Precision

4-bit:

```
  4 3 2 1
X 0 0 2 0
---------
8 6 4 2 0
```

3-bit:

```
  4 3 2 0
X 0 0 2 0
---------
8 6 4 0 0
```

2-bit:

```
  4 3 0 0
X 0 0 2 0
---------
8 6 0 0 0
```

1-bit:

```
  4 0 0 0
X 0 0 2 0
---------
8 0 0 0 0
```

Normal Multiplier

2nd Cycle:

4 3 2 1

Memory

Hardware

X 0020 **BSM**

Sum = 8 0 0 0 0

Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 2-Bit Precision

4-bit:

```
  4 3 2 1
X 0 0 2 0
---------
8 6 4 2 0
```

3-bit:

```
  4 3 2 0
X 0 0 2 0
---------
8 6 4 0 0
```

2-bit:

```
  4 3 0 0
X 0 0 2 0
---------
8 6 0 0 0
```

1-bit:

```
  4 0 0 0
X 0 0 2 0
---------
8 0 0 0 0
```

Normal Multiplier

2nd Cycle:

4 3 **2** **1**

3 means 300.

Memory

Hardware

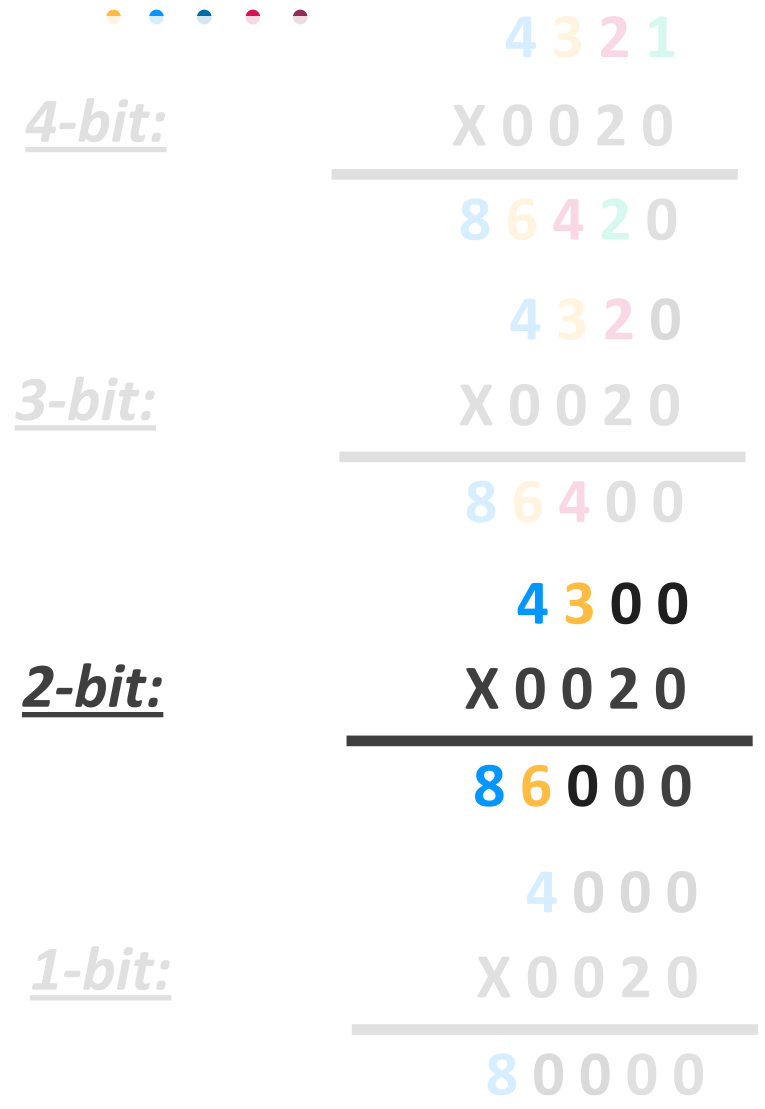
X 0020 $\boxed{\overset{3}{\textbf{BSM}}}$

Sum += 20 * 300

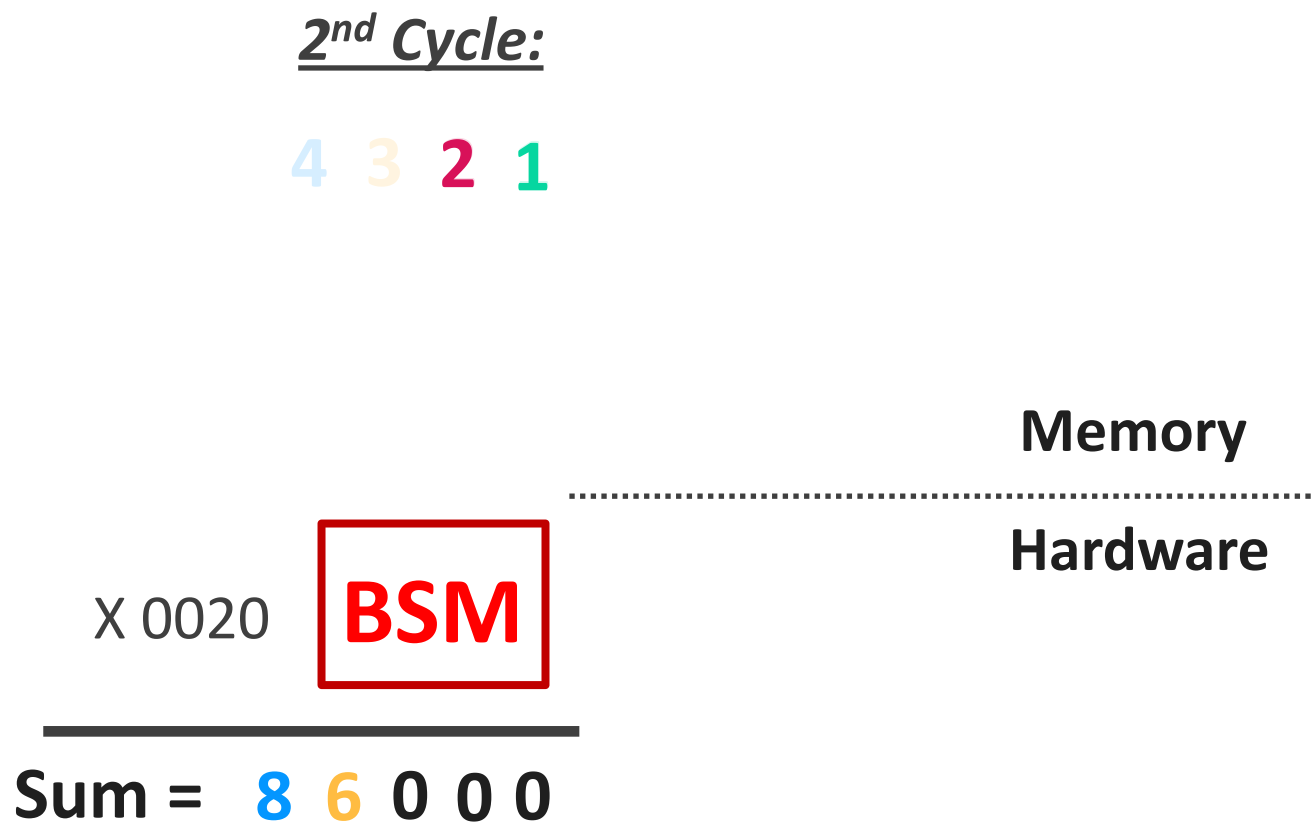
Sum = 8 0 0 0 0

Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 2-Bit Precision



Normal Multiplier



Done with 2-bit precision, or proceed to the next bit.

Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 3-Bit Precision

4-bit:

```
  4 3 2 1
X 0 0 2 0
---------
8 6 4 2 0
```

3-bit:

```
  4 3 2 0
X 0 0 2 0
---------
8 6 4 0 0
```

2-bit:

```
  4 3 0 0
X 0 0 2 0
---------
8 6 0 0 0
```

1-bit:

```
  4 0 0 0
X 0 0 2 0
---------
8 0 0 0 0
```

Normal Multiplier

3th Cycle:

4 3 **2** **1**

Memory

Hardware

X 0020 $\boxed{\textbf{BSM}}$

Sum = 8 6 0 0 0

Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 3-Bit Precision

4-bit:

```
   4 3 2 1
 X 0 0 2 0
 ---------
 8 6 4 2 0
```

3-bit:

```
   4 3 2 0
 X 0 0 2 0
 ---------
 8 6 4 0 0
```

2-bit:

```
   4 3 0 0
 X 0 0 2 0
 ---------
 8 6 0 0 0
```

1-bit:

```
   4 0 0 0
 X 0 0 2 0
 ---------
 8 0 0 0 0
```

Normal Multiplier

3th Cycle:

4 3 2 1

2 means 20.

Memory

Hardware

X 0020 $\boxed{\overset{2}{\text{BSM}}}$ Sum += 20 * 20

Sum = 8 6 0 0 0

Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 3-Bit Precision

```
        4 3 2 1
4-bit:  X 0 0 2 0
        ---------
        8 6 4 2 0

        4 3 2 0
3-bit:  X 0 0 2 0
        ---------
        8 6 4 0 0

        4 3 0 0
2-bit:  X 0 0 2 0
        ---------
        8 6 0 0 0

        4 0 0 0
1-bit:  X 0 0 2 0
        ---------
        8 0 0 0 0
```

Normal Multiplier

3th Cycle:

4 3 2 1

Memory

Hardware

X 0020 $\boxed{\text{BSM}}$

Sum = 8 6 4 0 0

Done with 3-bit precision, or proceed to the next bit.

Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 4-Bit Precision

4-bit:

```
   4 3 2 1
 X 0 0 2 0
 ---------
 8 6 4 2 0
```

3-bit:

```
   4 3 2 0
 X 0 0 2 0
 ---------
 8 6 4 0 0
```

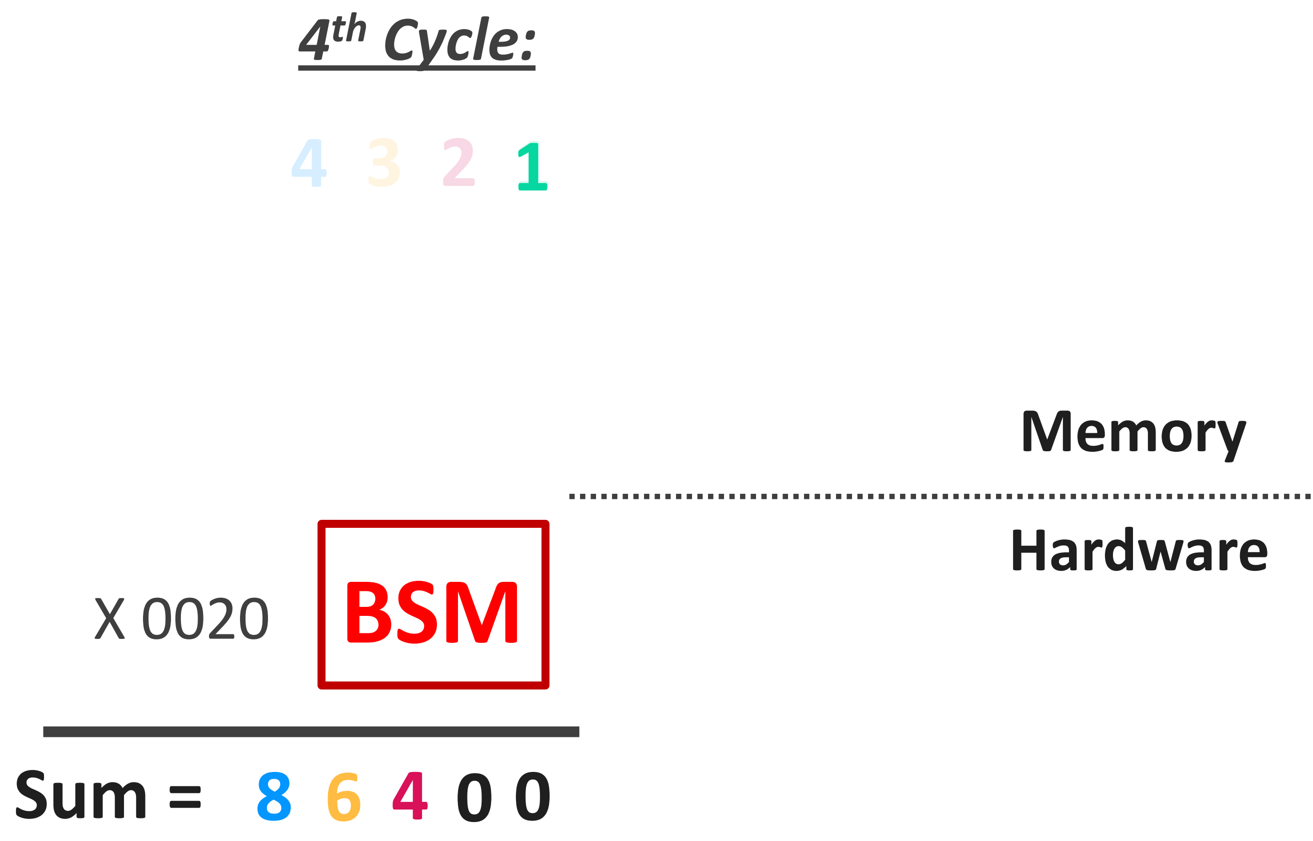
2-bit:

```
   4 3 0 0
 X 0 0 2 0
 ---------
 8 6 0 0 0
```

1-bit:

```
   4 0 0 0
 X 0 0 2 0
 ---------
 8 0 0 0 0
```

Normal Multiplier



Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 4-Bit Precision

● ● ● ● ●

4-bit:

```
   4 3 2 1
 X 0 0 2 0
----------
 8 6 4 2 0
```

3-bit:

```
   4 3 2 0
 X 0 0 2 0
----------
 8 6 4 0 0
```

2-bit:

```
   4 3 0 0
 X 0 0 2 0
----------
 8 6 0 0 0
```

1-bit:

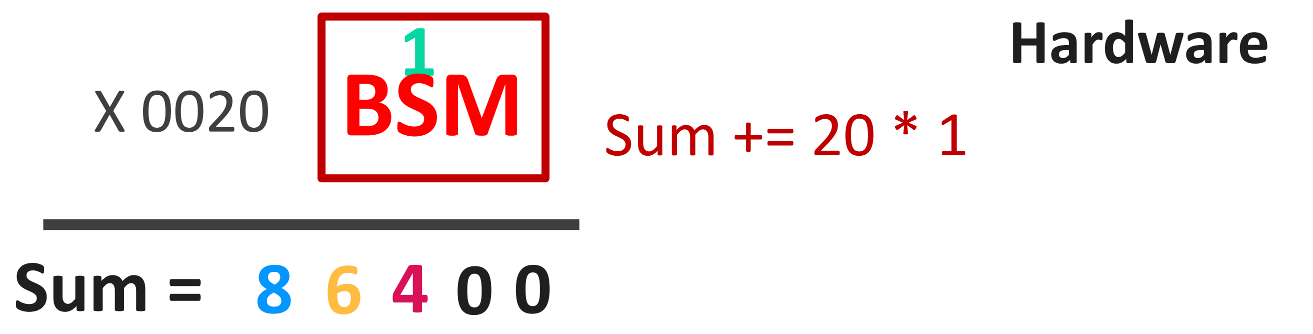
```
   4 0 0 0
 X 0 0 2 0
----------
 8 0 0 0 0
```

Normal Multiplier

4th Cycle:

4 3 2 1

1 means 1.



Sum = 8 6 4 0 0

Bit-serial Multiplier (BSM)

Bit-serial Multiplier: 4-Bit Precision

4-bit:

```
  4 3 2 1
X 0 0 2 0
---------
8 6 4 2 0
```

3-bit:

```
  4 3 2 0
X 0 0 2 0
---------
8 6 4 0 0
```

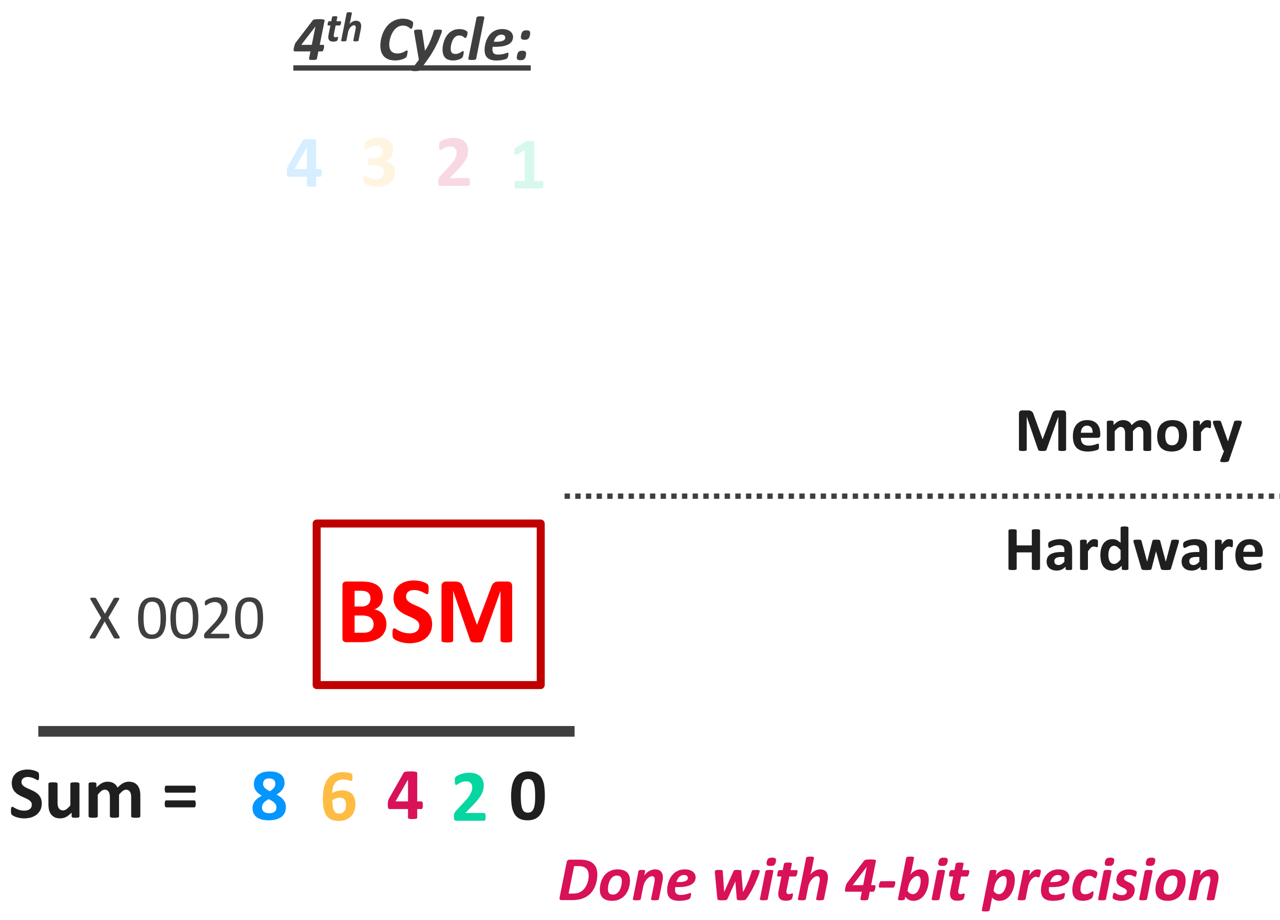
2-bit:

```
  4 3 0 0
X 0 0 2 0
---------
8 6 0 0 0
```

1-bit:

```
  4 0 0 0
X 0 0 2 0
---------
8 0 0 0 0
```

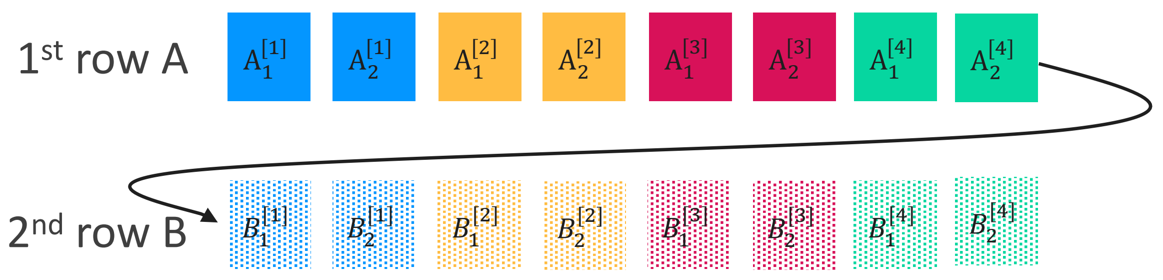
Normal Multiplier



Bit-serial Multiplier (BSM)

Custom Computation for MLWeaving

MLWeaving memory layout:

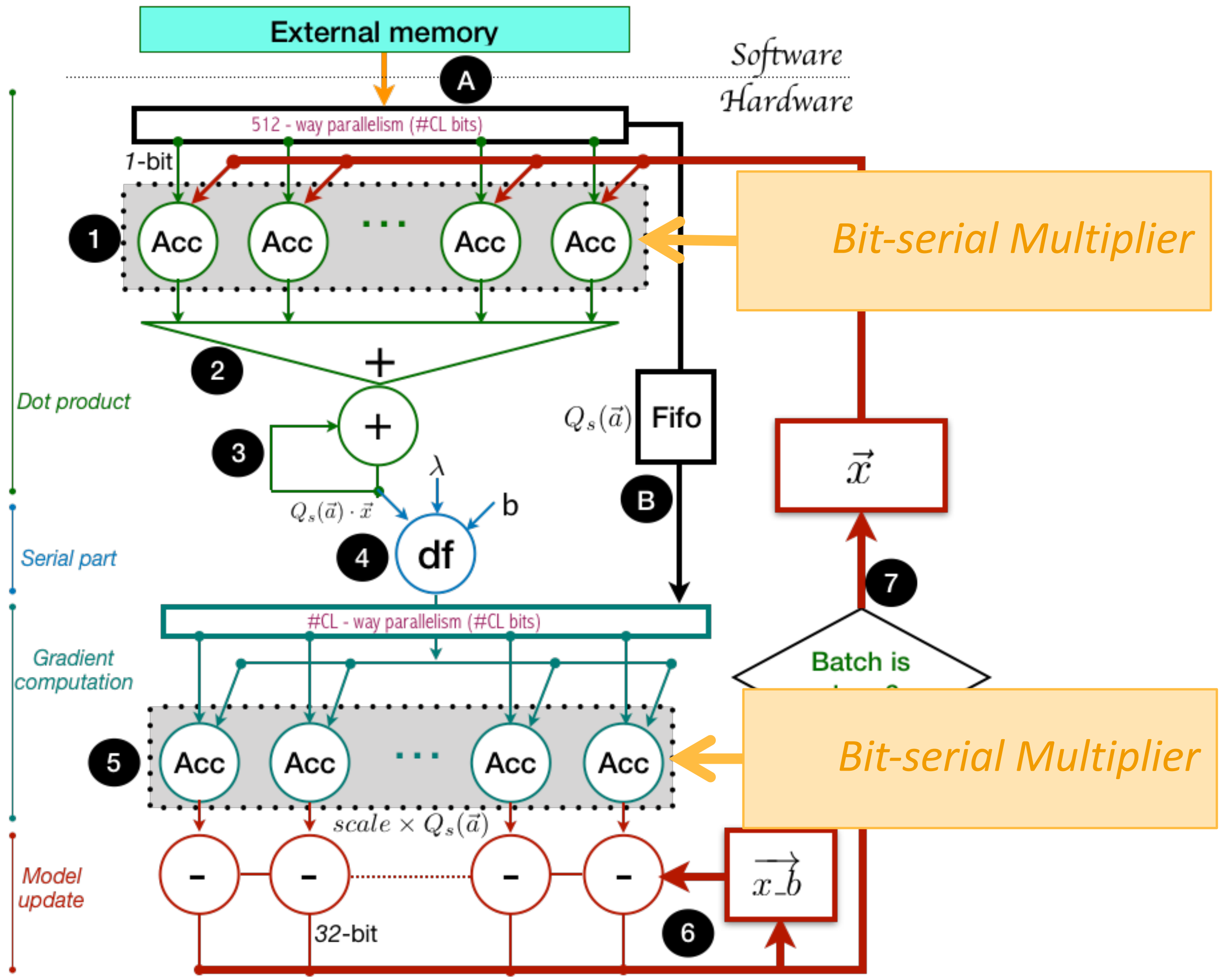

1st row A
$A_1^{[1]}$ $A_2^{[1]}$ $A_1^{[2]}$ $A_2^{[2]}$ $A_1^{[3]}$ $A_2^{[3]}$ $A_1^{[4]}$ $A_2^{[4]}$
2nd row B
$B_1^{[1]}$ $B_2^{[1]}$ $B_1^{[2]}$ $B_2^{[2]}$ $B_1^{[3]}$ $B_2^{[3]}$ $B_1^{[4]}$ $B_2^{[4]}$

Bit-serial	Bit-parallel

Dot product: $A_r * x$

Gradient: $A_r * (A_r * x - br)$

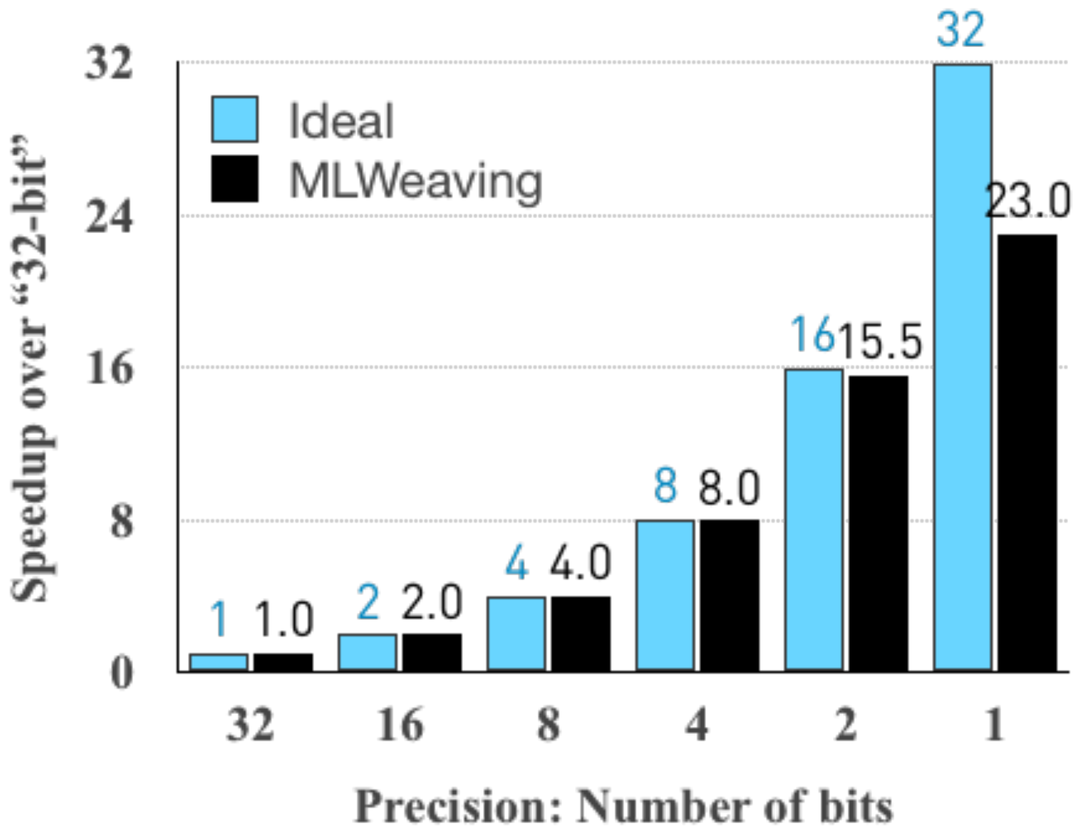
MLWeaving hardware design:



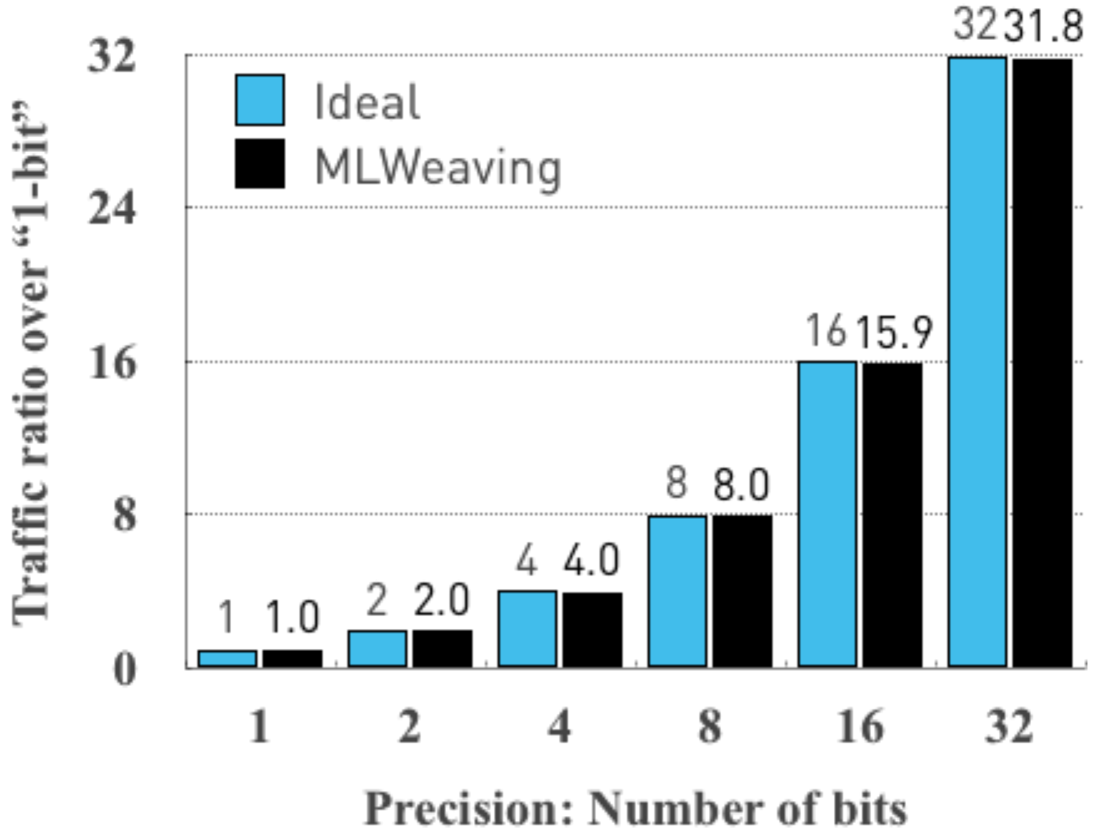
***Bit-serial multiplier* + *MLWeaving memory layout* enable *any-precision* ML training.**

MLWeaving's Performance: Almost Linear Speedup with Lower Precision

Computing time vs. Precision



Memory traffic vs. Precision



Outline

Quick Background

Stochastic Gradient Descent (SGD)

Synchronous vs. Asynchronous

Low Precision

MLWeaving

Arbitrary-precision Training

MLWeaving Memory Layout

MLWeaving Hardware Design

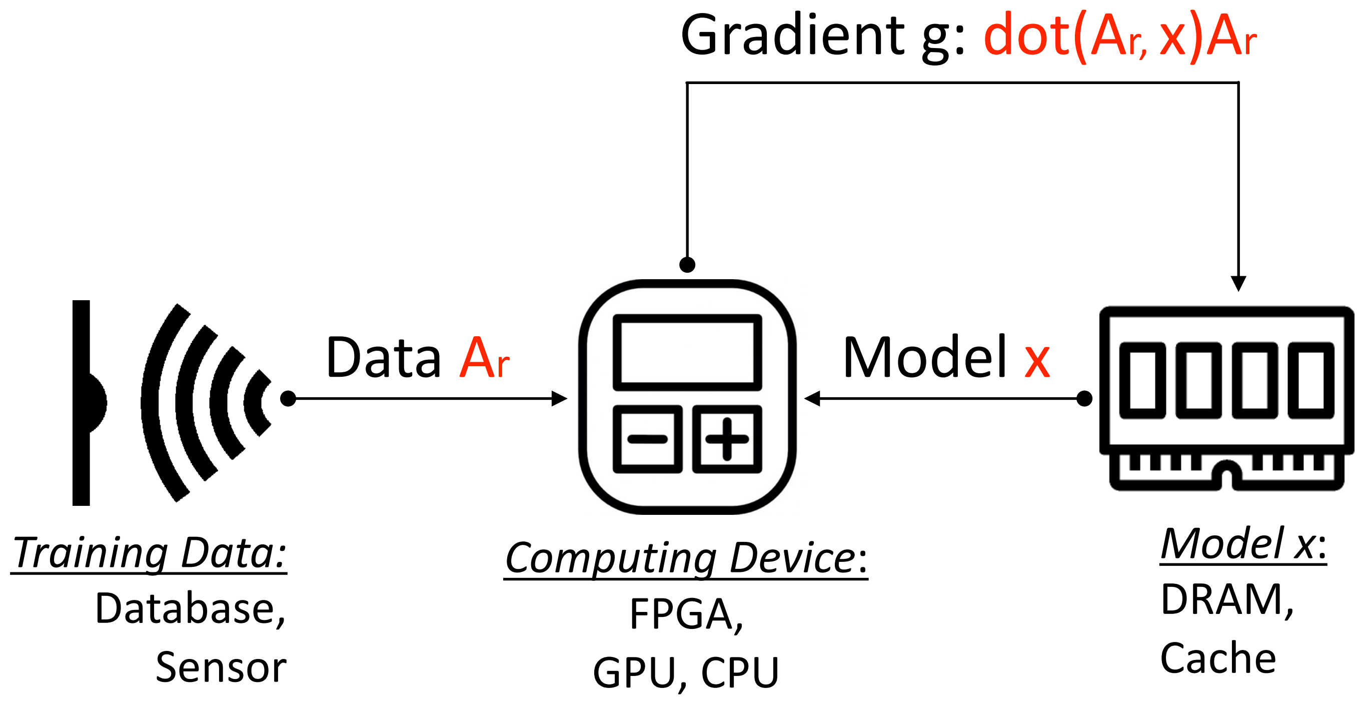
$\boxed{\text{Efficient Synchronous Design}}$

SGD on the CPU: synchronous or asynchronous?

Sync. Single-Core SGD: Low Throughput

CPU – Single Core

```
A_r = get_data()

 x  = get_model()

 g  = comp_grad(x, A_r)

 x  = x - g

 set_model(x)
```



Read After Write (RAW) Dependency Regarding the Model x

Causes Problem When Using Multiple Cores.

Async. Multi-Core SGD: High Throughput

Multi-core SGD relies on asynchrony.

HogWild! [1]

ModelAverage [2]

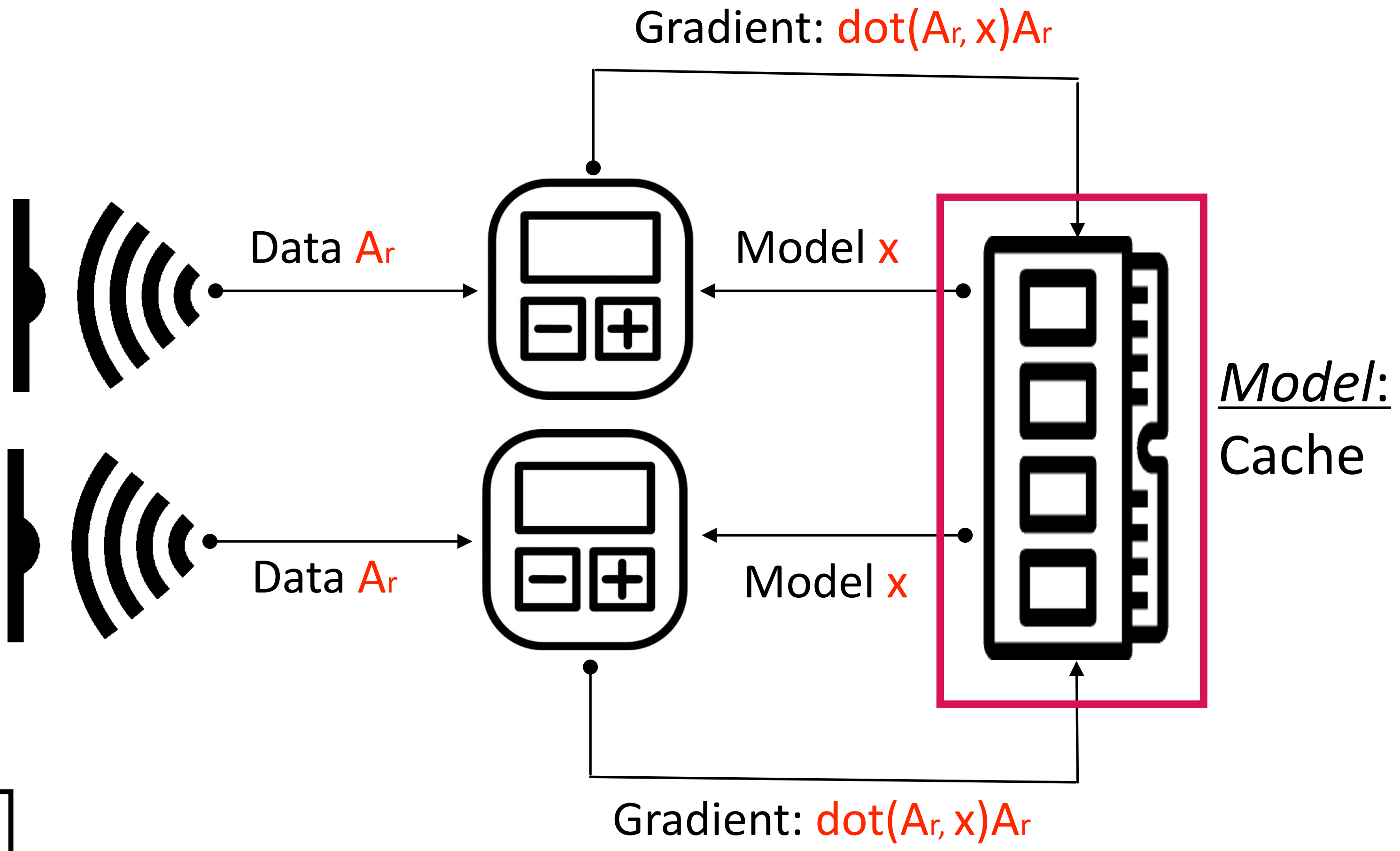
[1] Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In NIPS. 2011.

[2] Parallelized Stochastic Gradient Descent. In NIPS. 2010.

Hogwild: Asynchrony

```
A_r = get_data()

 x = get_model()

 g = comp_grad(x,A_r)

 x = x - g

 set_model(x)
```

Shared model x among cores

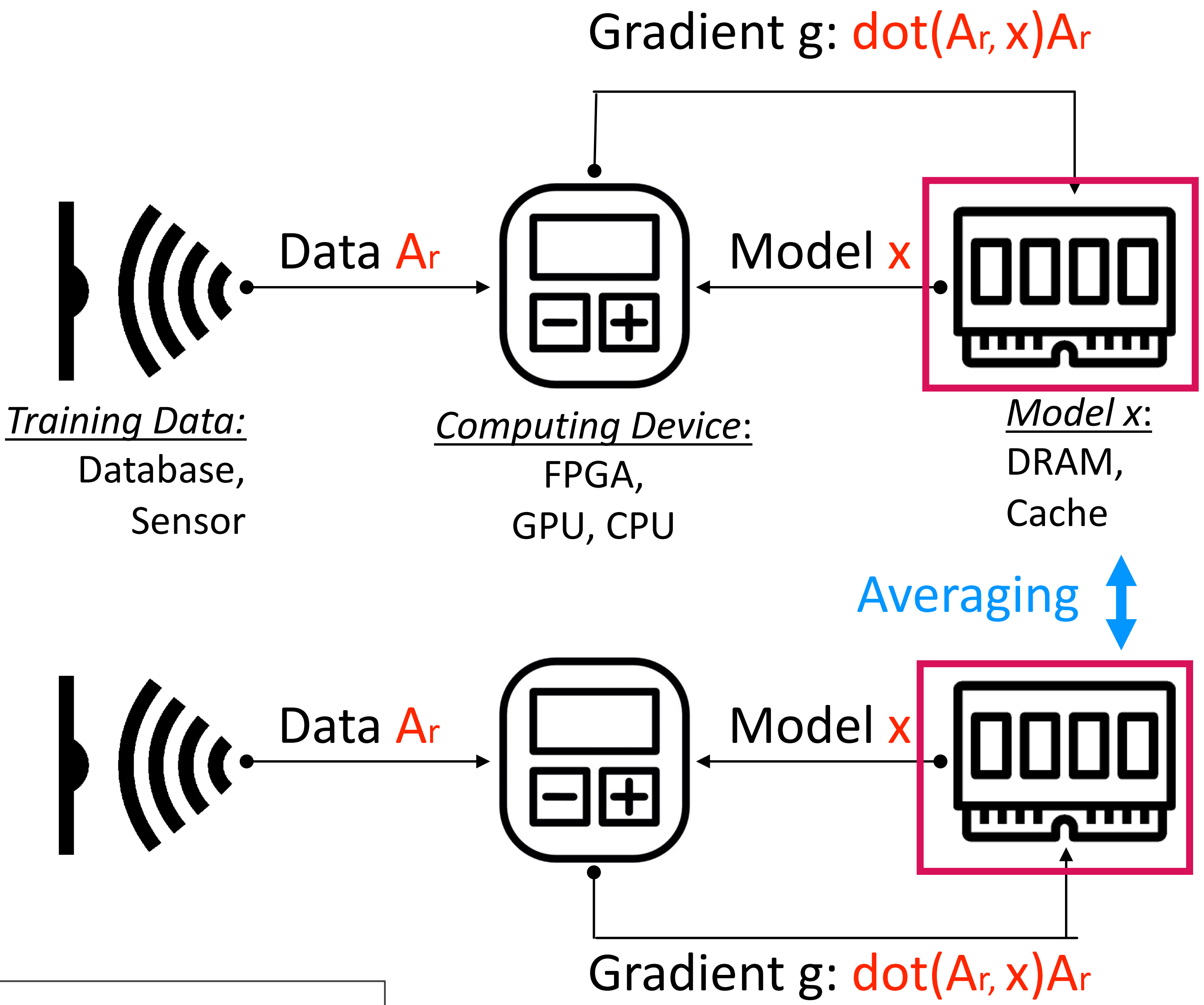


Problem? *Cache-coherence is expensive, especially for dense data!*

ModelAverage: Asynchrony

```
A_r = get_data()

 x = get_model()

 g = comp_grad(x,A_r)

 x = x - g

 set_model(x)
```

A copy of model x for each core



Problem? *Convergence might be slower.*

Synchrony vs. Asynchrony on CPUs

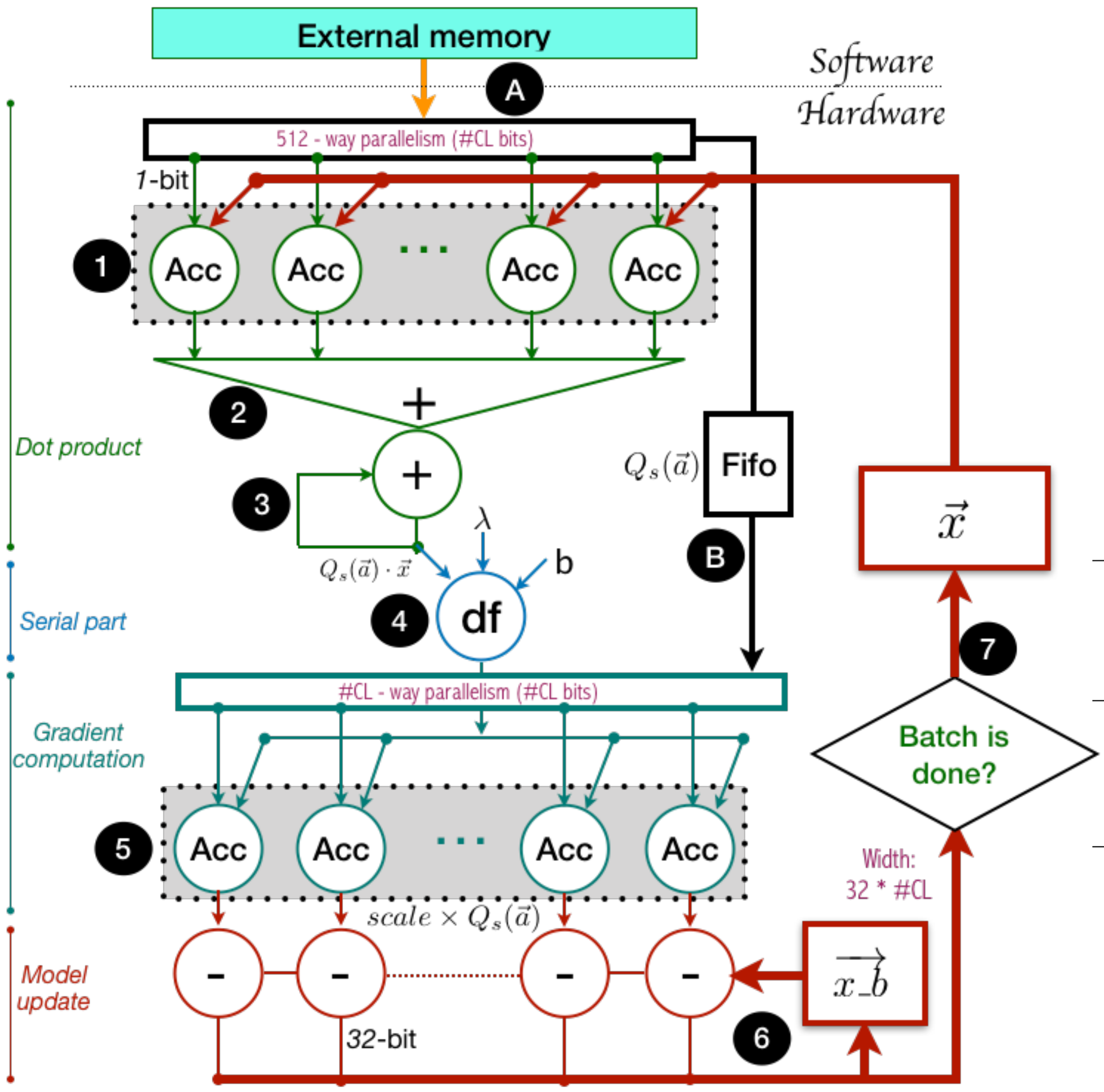
	Hardware Efficiency (Throughput)	**Statistical Efficiency (Convergence Rate)**
Single-core SGD (Synchrony)	Low 🙁	High 🙂
Multi-core SGD (Asynchrony)	High 🙂	Low 🙁

Synchronous SGD or asynchronous SGD on custom hardware?

SGD on Custom Hardware: The Best of Two Worlds

	Hardware Efficiency (Throughput)	**Statistical Efficiency (Convergence Rate)**
Single-core (Synchrony)	Low ☹	High 🙂
Multi-core (Asynchrony)	High 🙂	Low ☹
Custom hardware (Synchrony)	High 🙂	High 🙂

Original Synchronous Implementation: Compute-Bound

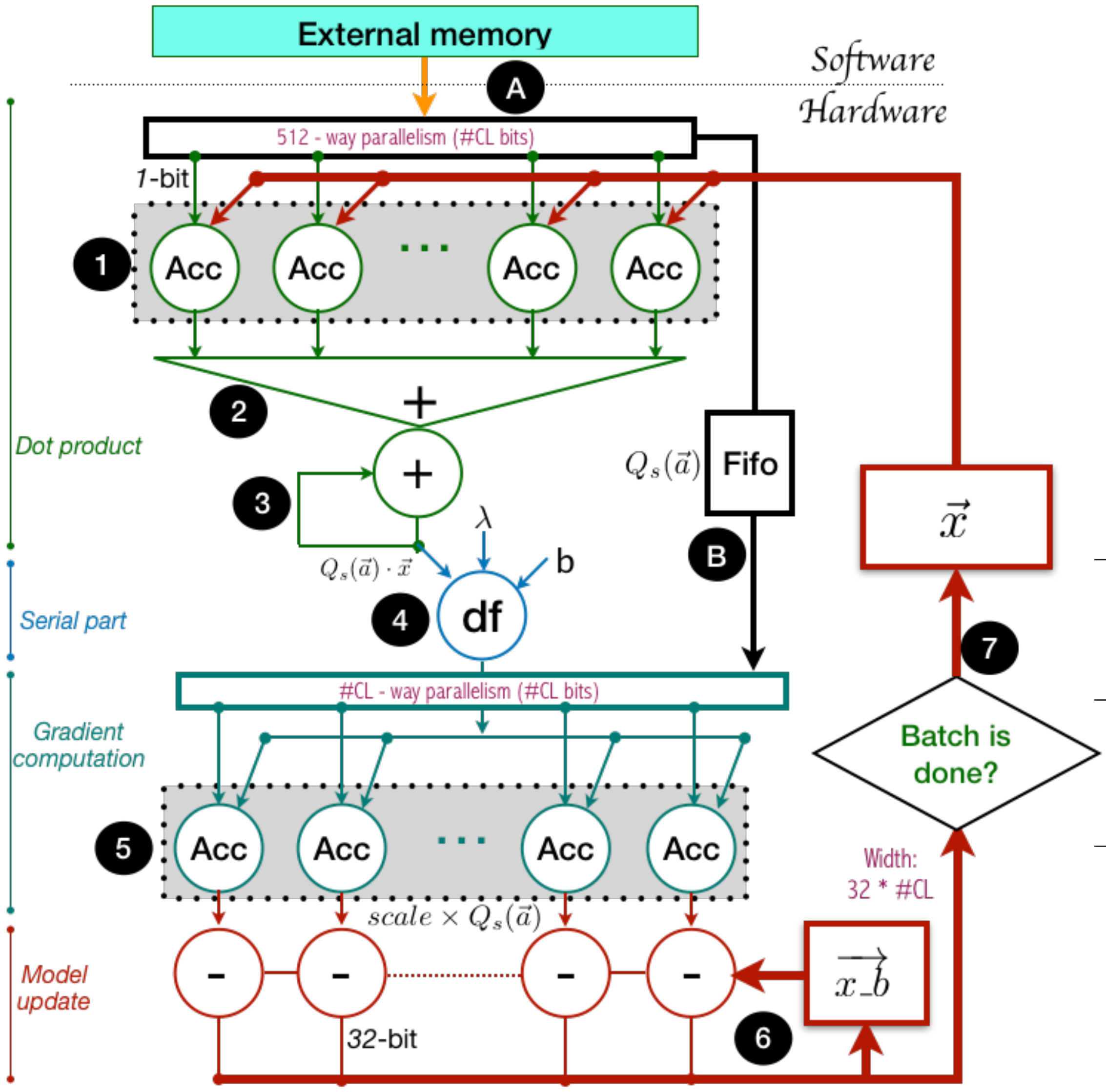


***Key idea**: to keep the RAW dependency, regarding the model x.*

Original Implementation

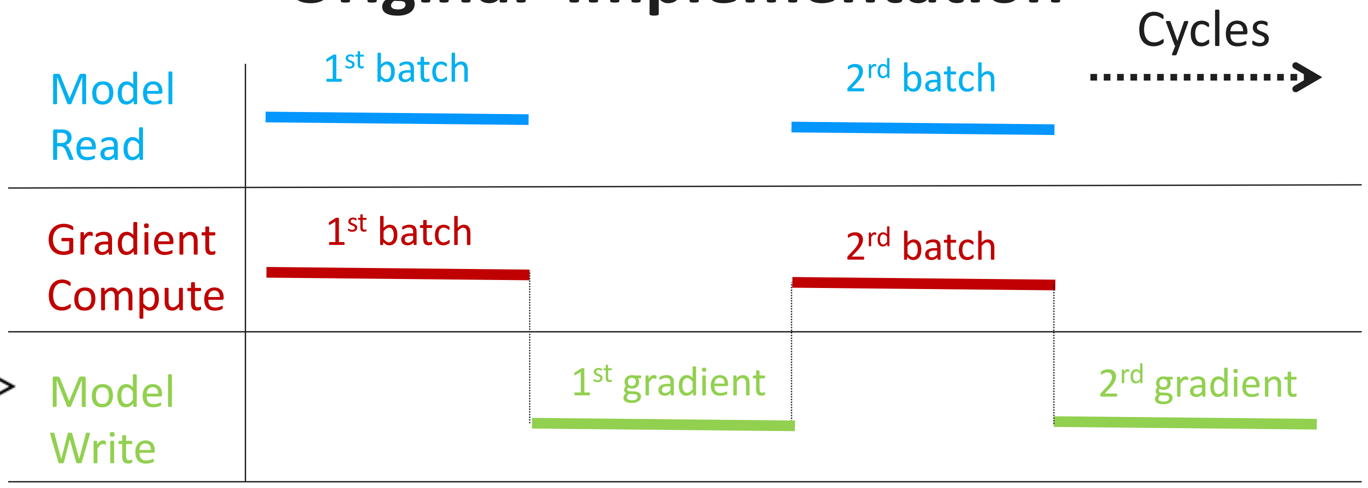
	Cycles
Model Read	
Gradient Compute	
Model Write	

Original Synchronous Implementation: Compute-Bound



***Key idea**: to keep the RAW dependency, regarding the model x.*

Original Implementation



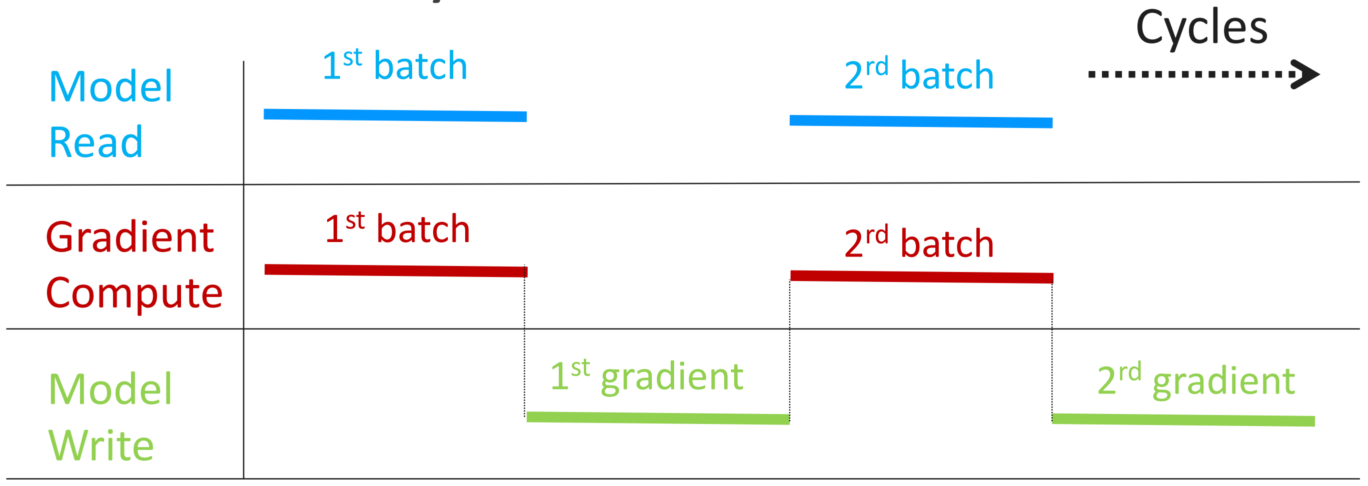
50% Utilization

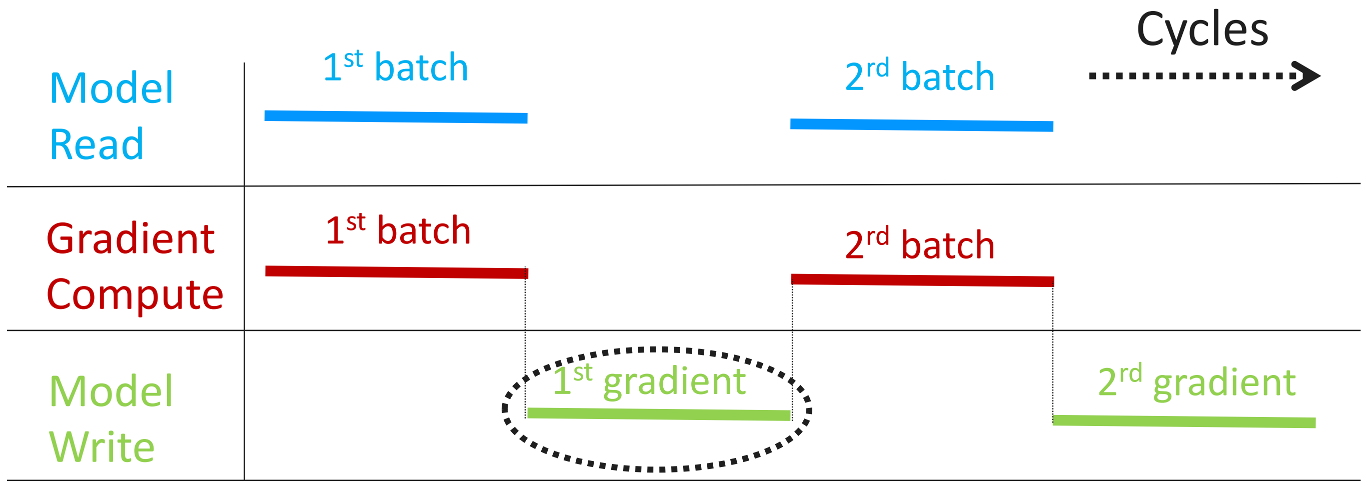
Optimal Synchronous Implementation: Memory-Bound

Original: Compute-bound

Observation: Custom hardware can update the model (thousands of weights) at the granularity level: 64 weights, not the whole model.

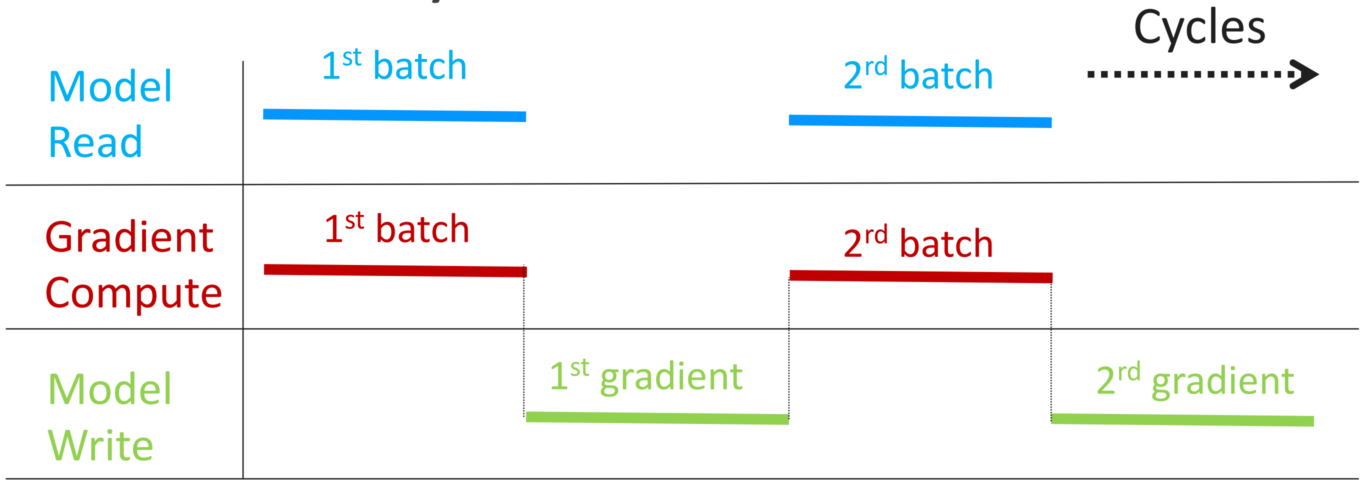
With Chaining: Memory-bound





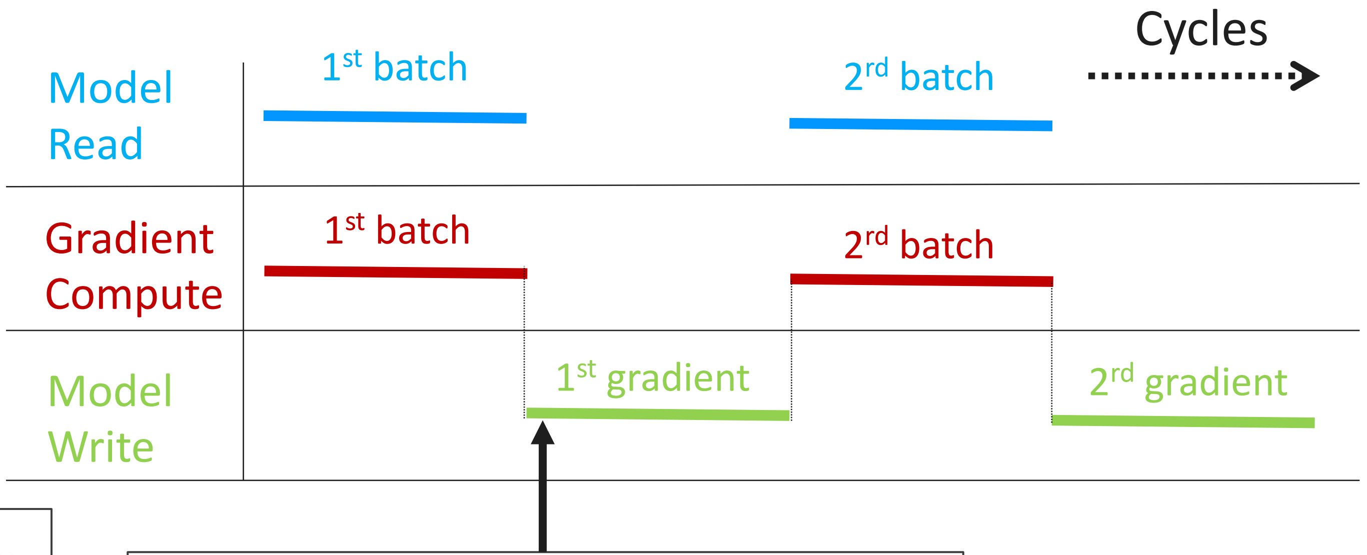
Optimal Synchronous Implementation: Memory-Bound

Original: Compute-bound



Observation: Custom hardware can update the model (thousands of weights) at the granularity level: 64 weights, not the whole model.

With Chaining: Memory-bound

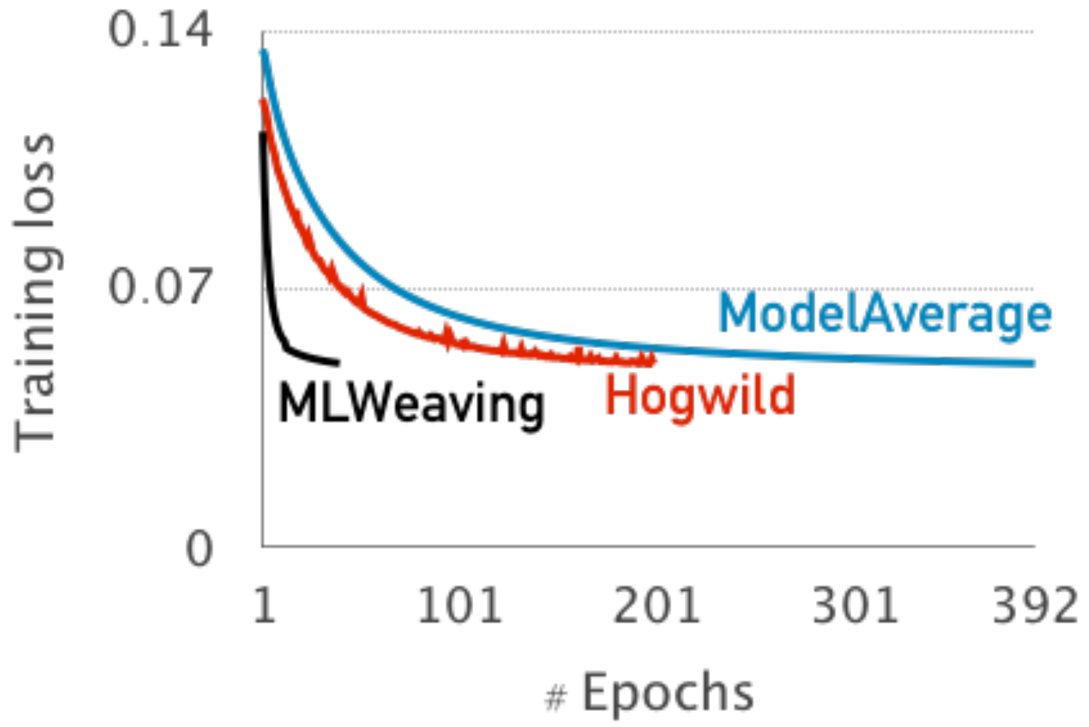


High throughput: "sync" is as fast as "async".

***Gap**: gradient from 64 weights*

Effect of Sync. Design

Training loss vs. Number of Epochs



Our sync. design needs fewer epochs to converge.

ModelAverage *and* ***Hogwild*** on a multi-core CPU: **Async.**

MLWeaving on the *custom hardware*: **Sync.**

Outline

● ● ● ● ●

Quick Background

Stochastic Gradient Descent (SGD)

- Synchronous vs. Asynchronous
- Low Precision

MLWeaving

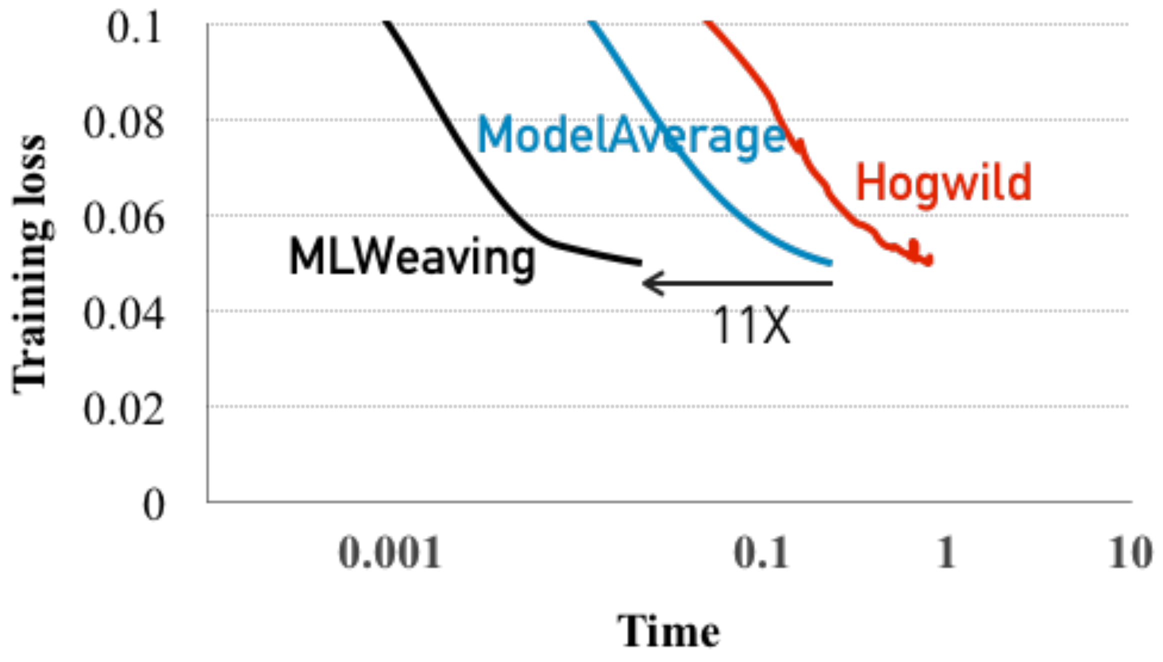
Arbitrary-precision Training

- MLWeaving Memory Layout
- MLWeaving Hardware Design

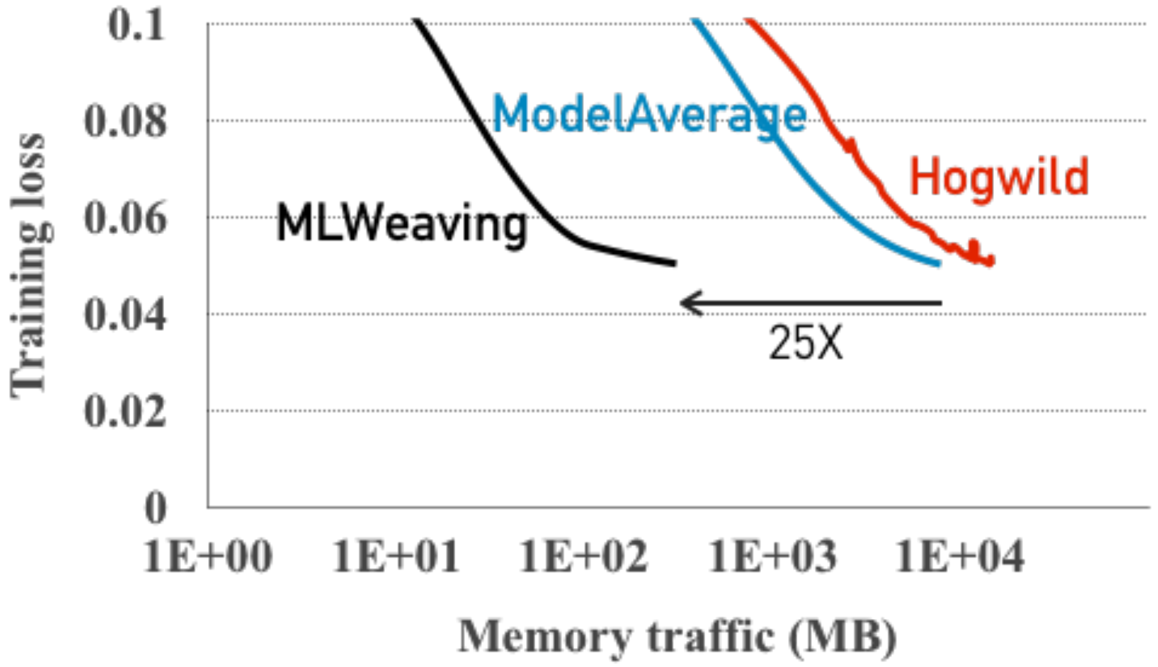
Efficient Synchronous Design

End-to-End Performance: MLWeaving

Training loss vs. Time



Training loss vs. Memory



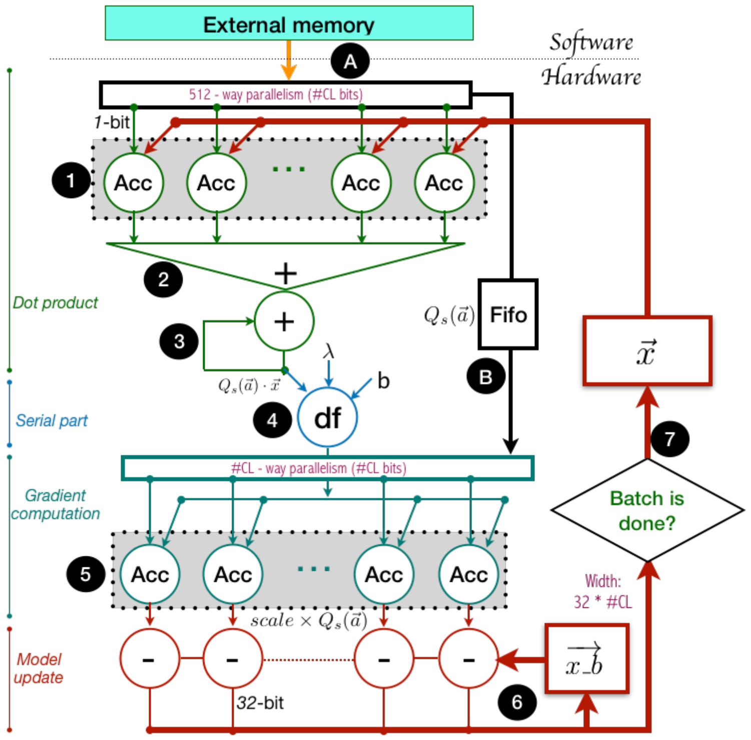
ModelAverage and Hogwild **on an Intel CPU: 14 cores, AVX2-enhanced, 8-bit dataset.**

MLWeaving **on an FPGA: 3-bit dataset.**

Big Data

Machine Learning

New Hardware



Thanks!