Introduction to Computation Technologies in Deep Learning

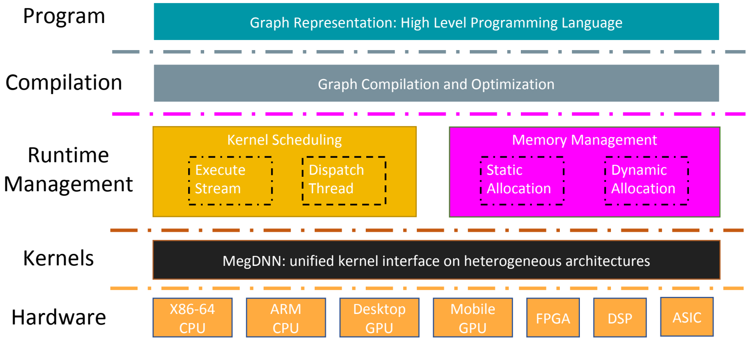
Kai Jia
jiakai@megvii.com

Megvii Inc.

March, 2019

1. Symbolic Computation
   - Representation
   - Execution & Optimization
2. Dense Numerical Computation
   - CPU Computation
   - Other Computation Devices
   - Computation & Memory Gap
3. Distributed Computation
   - System
   - Optimzation Algorithms
   - Communication Algorithms

Overview of a Deep Learning Framework
MegBrain Architecture

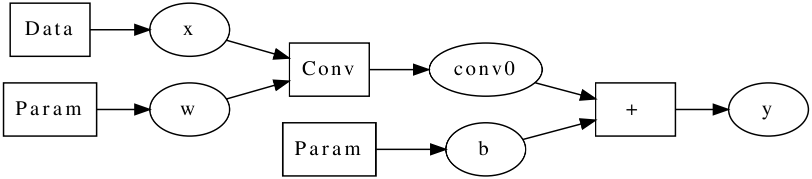


Computation Graph

$$y = \mathrm{conv}(x, w) + b$$

Computation Graph

$$y = \mathrm{conv}(x, w) + b$$



Graph Structure
Variable

- Corresponding to a tensor with concrete numerical values during graph execution

Graph Structure
Variable

- Corresponding to a tensor with concrete numerical values during graph execution
- Carrying some *attributes*:

DType Data type, like `int8` and `float32`.

Graph Structure
Variable

- Corresponding to a tensor with concrete numerical values during graph execution
- Carrying some *attributes*:

DType	Data type, like `int8` and `float32`.
Shape	Like `(5, 3)` for FC weight, `(128, 512, 7, 7)` for feature maps.

Example

Variable shape inference facilitates automatic weight initialization:

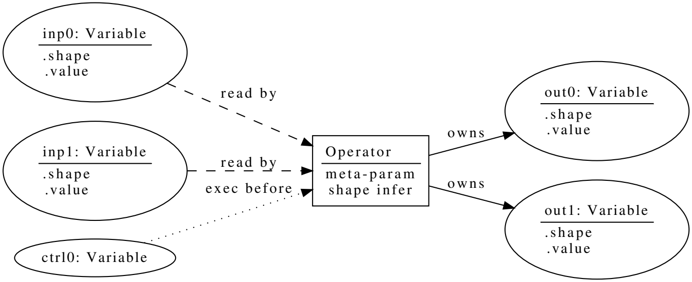
```
assert x.shape == (128, 50)
y = fully_connected(x, output_dim=100)
assert y.shape == (128, 100)
```

The weight matrix of this `FullyConnected` operator can be initialized to `np.random.normal((50, 100))`.

Graph Structure
Operators & Edges

Operator
- Operators & variables form a bipartite graph
- Operators define the computation to be applied on input variables

Edge
- Data dependency: read input data
- Control dependency: require input operator to have finished



Operator Granularity

Trade-off between flexibility and ease-to-optimize

Category	**Example**	**Advantage**	**Framework**
Coarse-grained	$y = \mathrm{BatchNorm}(x)$	Parsimony; Easy performance tuning	Caffe
Fine-grained	$y = \frac{x - \mathrm{mean}(x)}{\mathrm{std}(x)}$	Flexibility	Theano

Operator Granularity

Trade-off between flexibility and ease-to-optimize

Category	**Example**	**Advantage**	**Framework**
Coarse-grained	$y = \text{BatchNorm}(x)$	Parsimony; Easy performance tuning	Caffe
Fine-grained	$y = \frac{x - \text{mean}(x)}{\text{std}(x)}$	Flexibility	Theano

Our philosophy:

- Prefer flexibility: this can not be changed once the framework has been designed
- Utilize multi-level API for simplifying graph representation
- Speed can be continuously improved by graph optimizer

Auto Differentiation

Calculus 101: The Chain Rule

Straight-forward approach: each operator provides two methods: `fprop` and `bprop`.

```
inline Dtype Forward(const vector<Blob<Dtype>*>& bottom,
  const vector<Blob<Dtype>*>& top);

inline void Backward(const vector<Blob<Dtype>*>& top,
  const vector<bool>& propagate_down,
  const vector<Blob<Dtype>*>& bottom);
```

Auto Differentiation
Calculus 101: The Chain Rule

Straight-forward approach: each operator provides two methods: `fprop` and `bprop`.

Limitations

- Graph optimizer can not be uniformly applied on both forward and backward passes
- Difficult to implement gradient of gradient ($\frac{\partial f(\frac{\partial L}{\partial x})}{\partial y}$, in WGAN training [1]) or higher-order gradients ($\frac{\partial^2 L}{\partial x^2}$).
- Difficult to modify/manipulate gradients (e.g. for low-bit training).

[1] Ishaan Gulrajani et al. "Improved training of wasserstein gans". In: *arXiv preprint arXiv:1704.00028* (2017).

Auto Differentiation

Calculus 101: The Chain Rule

Unified approach: extending the graph with operators computing gradients of specific variables, via the chain rule.

$$y_1, \cdots, y_m = f(x_1, \cdots, x_n)$$

Gradient operator g for f:

Auto Differentiation
Calculus 101: The Chain Rule

Unified approach: extending the graph with operators computing gradients of specific variables, via the chain rule.

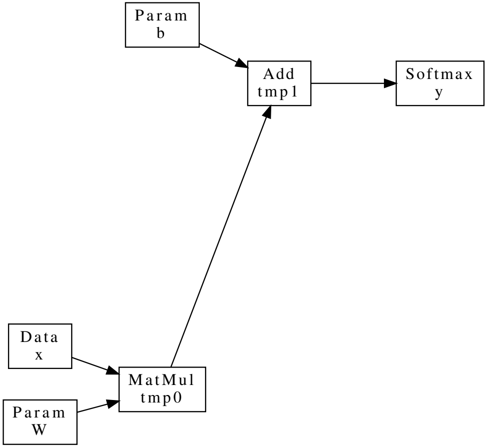
$$y_1, \cdots, y_m = f(x_1, \cdots, x_n)$$

Gradient operator g for f:

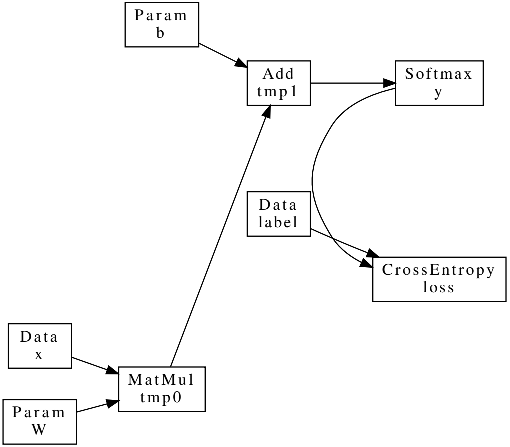
$$\frac{\partial L}{\partial x_1}, \cdots, \frac{\partial L}{\partial x_n} = g\left(\frac{\partial L}{\partial y_1}, \cdots, \frac{\partial L}{\partial y_m}, x_1, \cdots, x_n, y_1, \cdots, y_m\right)$$

Auto Differentiation

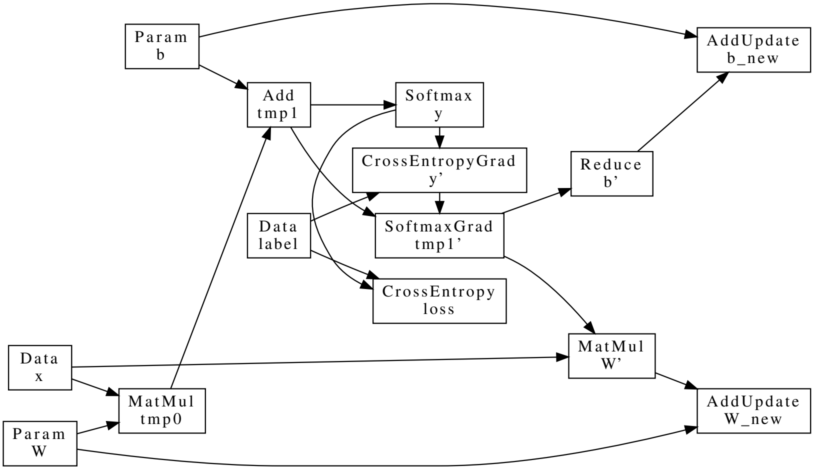
Calculus 101: The Chain Rule



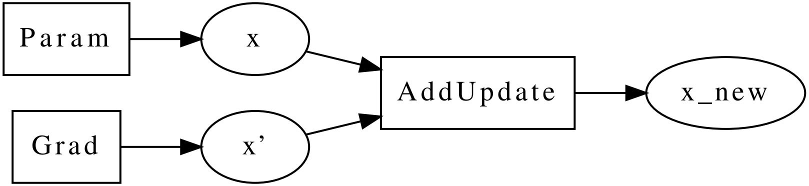
Auto Differentiation
Calculus 101: The Chain Rule



Auto Differentiation
Calculus 101: The Chain Rule

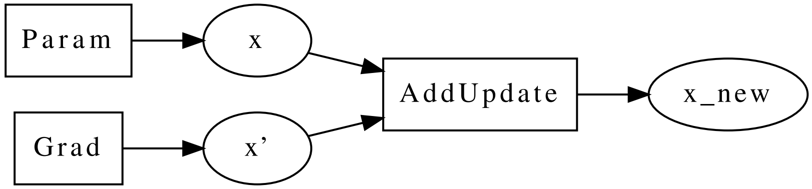


Mutable State
Enabling param updates to be expressed in graphs



Mutable State

Enabling param updates to be expressed in graphs

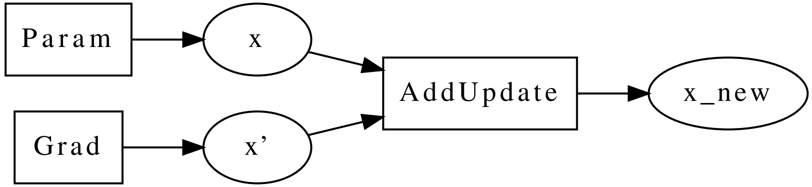


Note

- `x` and `x_new` share the underlying storage and should not be simultaneously read by one operator. Equivalently speaking, `AddUpdate` separates the graph.

Mutable State

Enabling param updates to be expressed in graphs



Note

- `x` and `x_new` share the underlying storage and should not be simultaneously read by one operator. Equivalently speaking, `AddUpdate` separates the graph.
- Readers of `x` must have finished (impl. by control dependency)

Symbolic Shape

Enabling computation involving tensor shapes

Example

Maxout[1]:

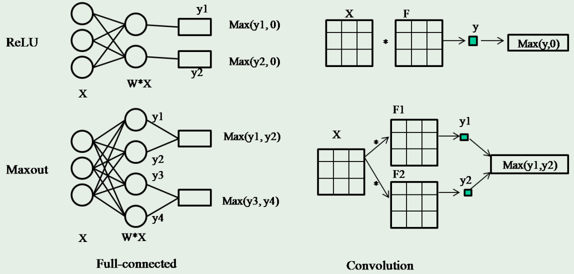


Image from [2]

[2]Hai Dai Nguyen, Anh Duc Le, and Masaki Nakagawa. "Recognition of Online Handwritten Math Symbols Using Deep Neural Networks". In: *IEICE Trans. Inf.& Syst.* 99.12 (2016), pp. 3110–3118.

[1]Ian J Goodfellow et al. "Maxout networks". In: *arXiv preprint*

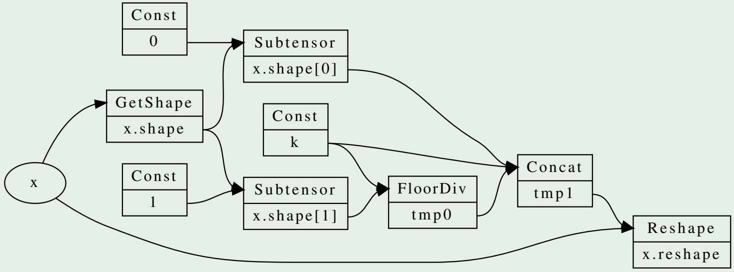
Symbolic Shape
Enabling computation involving tensor shapes

Example

Maxout[1]:

```
y = x.reshape(x.shape[0], x.shape[1] // k, k).max(axis=2)
```

where `x.shape` is also a symbol whose value is evaluated at runtime, so the computation adapts to different input shapes.



[1] Ian J Goodfellow et al. "Maxout networks". In: *arXiv preprint*

Symbolic Shape

Enabling computation involving tensor shapes

Example

Maxout[1]:

```
y = x.reshape(x.shape[0], x.shape[1] // k, k).max(axis=2)
```

- Helps dealing with non-constant batch size or input image size
- Requires dynamic shape support: some shapes may remain unknown until graph execution

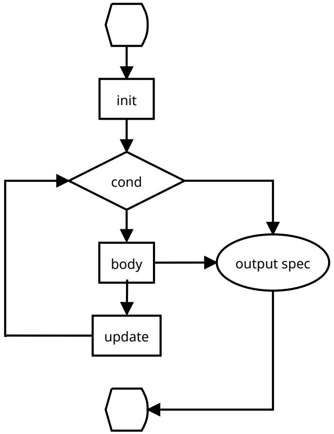
[1] Ian J Goodfellow et al. "Maxout networks". In: *arXiv preprint arXiv:1302.4389* (2013).

Control Flow Operators

Towards universal computation (in theory)

Loop operator:

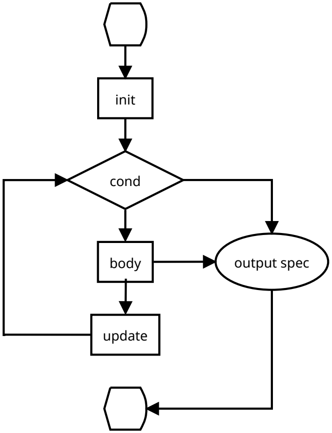
`scan` in Theano, `while_loop` in TensorFlow and `loop` in MegBrain.



Control Flow Operators

Towards universal computation (in theory)

Loop operator:
`scan` in Theano, `while_loop` in TensorFlow and `loop` in MegBrain.



- With symbolic shapes and control flow operators, a computation graph is Turing-complete!
- Useful for RNN and iterative algorithms

Dynamic Computation Graph

Ease of programming beyond Turing-completeness

Static Computation Graph

- **Unfamiliar programming model**:
  Stateless, functional: `y = x.setsub[1:3](xs)`
  rather than imperative: `x[1:3] = xs`

Dynamic Computation Graph

Ease of programming beyond Turing-completeness

Static Computation Graph

- **Unfamiliar programming model**:
  Stateless, functional: `y = x.setsub[1:3](xs)`
  rather than imperative: `x[1:3] = xs`
- **Difficult to debug**: code is written for graph contruction but tensor values can only be known during graph execution
  `y = printop(y)` rather than `print(y)`

Dynamic Computation Graph

Ease of programming beyond Turing-completeness

Dynamic Computation Graph

- Implemented by eager evaluation:

```
while (a.dot(x) - I).max().getvalue() > eps:
    x = x.dot(2 * I - a.dot(x))
print(grad(loss, x).getvalue())
```

Dynamic Computation Graph

Ease of programming beyond Turing-completeness

Dynamic Computation Graph

- Implemented by eager evaluation:

```
while (a.dot(x) - I).max().getvalue() > eps:
    x = x.dot(2 * I - a.dot(x))
print(grad(loss, x).getvalue())
```

- Auto differentiation: keep symbolic track of computation path

Dynamic Computation Graph

Ease of programming beyond Turing-completeness

Dynamic Computation Graph

- Implemented by eager evaluation:

```
while (a.dot(x) - I).max().getvalue() > eps:
    x = x.dot(2 * I - a.dot(x))
print(grad(loss, x).getvalue())
```

- Auto differentiation: keep symbolic track of computation path
- Difficulties:
  - Hard to optimize: lack of global information
  - Hard to deploy: graph depends on code

Dynamic Computation Graph
Ease of programming beyond Turing-completeness

Dynamic Computation Graph

- Implemented by eager evaluation:

```
while (a.dot(x) - I).max().getvalue() > eps:
    x = x.dot(2 * I - a.dot(x))
print(grad(loss, x).getvalue())
```

- Auto differentiation: keep symbolic track of computation path
- Difficulties:
    - Hard to optimize: lack of global information
    - Hard to deploy: graph depends on code
    - Solution: tracing, source code transformation [2]

[2] Dan Moldovan et al. "AutoGraph: Imperative-style Coding with Graph-based Performance". In: *SysML* (2019).

1. Symbolic Computation
    - Representation
    - Execution & Optimization
2. Dense Numerical Computation
    - CPU Computation
    - Other Computation Devices
    - Computation & Memory Gap
3. Distributed Computation
    - System
    - Optimzation Algorithms
    - Communication Algorithms

Graph Execution

Separation of representation and execution allows abstraction of hardware details

- Map from variables to tensor values

Graph Execution

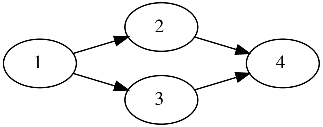
Separation of representation and execution allows abstraction of hardware details

- Map from variables to tensor values
- Map from operators to kernels on some specific architecture

Graph Execution

Separation of representation and execution allows abstraction of hardware details

- Map from variables to tensor values
- Map from operators to kernels on some specific architecture
- Kernels are scheduled according to topological order



$(1, 2, 3, 4)$ or $(1, 3, 2, 4)$

Optimizing by Graph Transformation

- Expression simplifying: $x + 1 - 2 + x \Rightarrow 2x - 1$

Optimizing by Graph Transformation

- Expression simplifying: $x + 1 - 2 + x \Rightarrow 2x - 1$
- Operation reordering according to shapes
  tensors: x and y
  scalars: a and b
  $x + a + y + b \Rightarrow a + b + x + y$

Optimizing by Graph Transformation

- Expression simplifying: $x + 1 - 2 + x \Rightarrow 2x - 1$
- Operation reordering according to shapes
  tensors: x and y
  scalars: a and b
  $x + a + y + b \Rightarrow a + b + x + y$
- Operator fusion: $x \cdot y + z \Rightarrow \text{fma}(x, y, z)$
  - static fusion: predefined fusion rules
  - dynamic fusion: Just-in-time compilation (JIT) for actual computation graph

Runtime Memory Management

- Baseline: reference counting + some classical memory allocator

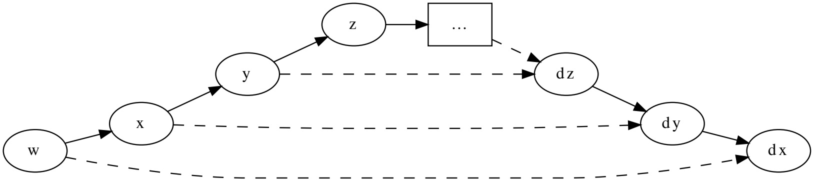
Runtime Memory Management

- Baseline: reference counting + some classical memory allocator
- Readonly forwarding: reuse input storage for operators like `reshape` and `subtensor`
- Writable forwarding (a.k.a. inplace operation): overwrite input storage
  Caution: must ensure no other readers exist (i.e. `refcnt` equals 1)

Sublinear Memory
Trade time for memory

Observation

- Long-term dependency for gradient computing consumes lots of memory.
- Assume $x_{i+1} = \mathrm{conv}(x_i, w_i)$, then x_{i+1} can only be discarded after $\frac{\partial L}{\partial w_i}$ and $\frac{\partial L}{\partial x_i}$ have been computed.

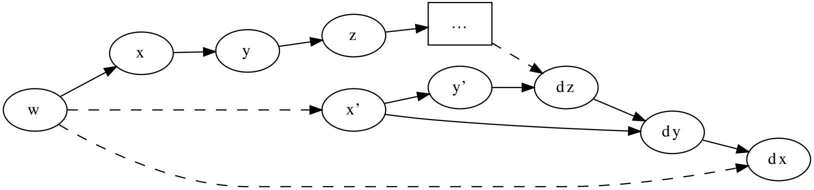


Sublinear Memory

Trade time for memory

Observation

- Long-term dependency for gradient computing consumes lots of memory.
- Assume $x_{i+1} = \text{conv}(x_i, w_i)$, then x_{i+1} can only be discarded after $\frac{\partial L}{\partial w_i}$ and $\frac{\partial L}{\partial x_i}$ have been computed.



Sublinear Memory

Trade time for memory

Observation

- Long-term dependency for gradient computing consumes lots of memory.
- Assume $x_{i+1} = \mathrm{conv}(x_i, w_i)$, then x_{i+1} can only be discarded after $\frac{\partial L}{\partial w_i}$ and $\frac{\partial L}{\partial x_i}$ have been computed.

Method

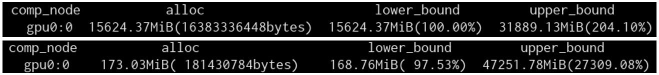
- Split the sequence into blocks consisting of consecutive operators and keep only the first variable in any block
- Recompute internal values in a block when gradient is needed
- In the above example, discard x_{km+j} for all $0 < j < m$ and recompute them when needed.

Sublinear Memory

Trade time for memory

Reduce memory usage to $O(\sqrt{n})$ with extra $O(n)$ time cost in the ideal case.

For a graph with 10000 convolutions and their gradients:



```
comp_node            alloc                       lower_bound         upper_bound
  gpu0:0  15624.37MiB(16383336448bytes)  15624.37MiB(100.00%)  31889.13MiB(204.10%)
```

```
comp_node            alloc                       lower_bound         upper_bound
  gpu0:0    173.03MiB( 181430784bytes)    168.76MiB( 97.53%)  47251.78MiB(27309.08%)
```

Note: this idea is also published in[3].

[3] Tianqi Chen et al. “Training deep nets with sublinear memory cost”. In: *arXiv preprint arXiv:1604.06174* (2016).

1. Symbolic Computation
   - Representation
   - Execution & Optimization
2. Dense Numerical Computation
   - CPU Computation
   - Other Computation Devices
   - Computation & Memory Gap
3. Distributed Computation
   - System
   - Optimzation Algorithms
   - Communication Algorithms

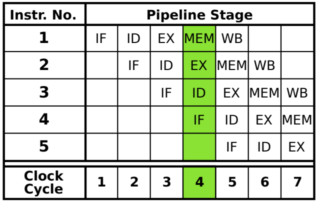
Instruction: The Hardware/Software Interface

What is a program?

```
int sum(int *x) {
    return x[0] + x[1];
}
```

```
0000000000000000 <sum>:
   0:     8b 47 04                       mov     0x4(%rdi),%eax
   3:     03 07                          add     (%rdi),%eax
   5:     c3                             retq
```

Modern CPU Technologies



Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

- **Pipeline** [4]
- **Superpipelining** increases stage number and simplifies each stage
- **Superscalar** dispatches multiple instructions to implement instruction-level parallelism
- **Out-of-order execution** executes according to availability of input data rather than original program order

[4] Image from `https://en.wikipedia.org/wiki/Instruction_pipelining`

SIMD
Single instruction, multiple data

Store multiple data items in one register and process them in a single instruction.

Calculation of theoretical FLOPS[5]

$$FLOPS = f \cdot w \cdot IPC$$

f	:	frequency
w	:	SIMD width (number of floats per register)
IPC	:	SIMD instructions per cycle

[5] floating point operations per second

SIMD
Single instruction, multiple data

Example

Intel® CPUs usually have $IPC = 2$. However if FMA is supported, IPC should be counted as 4 since 2 FMA instructions is essentially 4 floating point operations.

Intel® Xeon® Platinum 8180M[5]

# of Cores	28
Processor Base Frequency	2.50 GHz
Max Turbo Frequency	3.80 GHz
# of AVX-512 FMA Units	2

$$FLOPS = 3.8 Gcyc/s \times 4 instr/cyc \times 16 float/instr$$
$$= 243.2 GFLOPS$$
$$FLOPS\_TOT = FLOPS \times 28 = 6.8 TFLOPS$$

[5] data available at
https://ark.intel.com/products/120498/Intel-Xeon-Platinum-8180M-Processor-38_5M-Cache-2_50-GHz

Memory Hierarchy

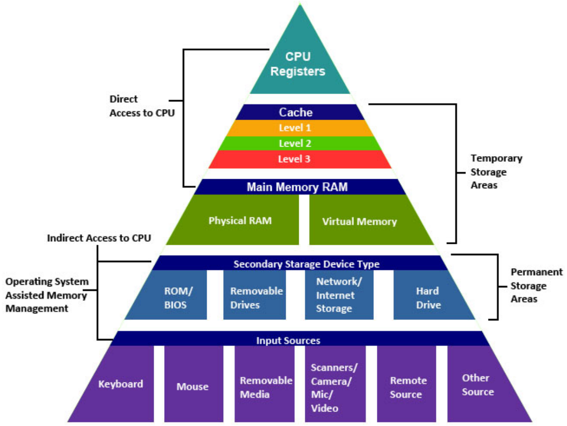


Image from `http://cse1.net/recaps/4-memory.html`

RAM Implementation

Static Random-Access Memory (SRAM)

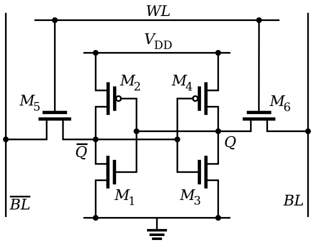
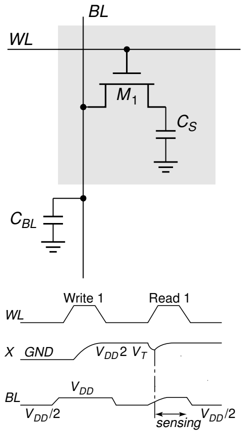


Image from `https://en.wikipedia.org/wiki/Static_random-access_memory`

Advantages:

1. Fast
2. Low power consumption
3. No refresh circuit

RAM Implementation
Dynamic Random-Access Memory (DRAM)



Advantages:

1. High density
2. Cheap

Refresh: periodically read blocks and write back.

Image from `http://docencia.ac.upc.edu/master/MIRI/NCD/docs/04-Memory%20Structures-2.pdf`

Cache Hierarchy

A hierarchical design for better trade-off between memory capacity and latency.

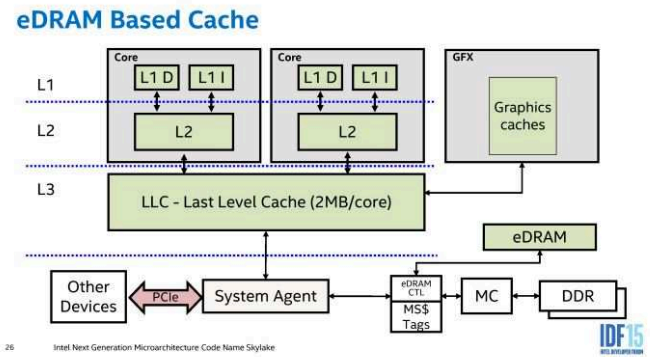


Image from

`https://www.anandtech.com/show/9582/intel-skylake-mobile-desktop-launch-architecture-analysis/5`

Cache Hierarchy

A hierarchical design for better trade-off between memory capacity and latency.

Core i7 Xeon 5500 Series

L1 hit	$\sim$ 4 cycles
L2 hit	$\sim$ 10 cycles
L3 hit line unshared	$\sim$ 40 cycles
L3 hit, shared line in another core	$\sim$ 65 cycles
L3 hit, modified in another core	$\sim$ 75 cycles
Remote L3	$\sim$ 100 – 300 cycles
Local DRAM	$\sim$ 60 ns
Remote DRAM	$\sim$ 100 ns

source:

https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

CPU Cache Structure

- Cache line

tag	data block	flag bits (valid, dirty)

- Indexing

tag (40bit)	index (6bit)	block offset (6bit)

CPU Cache Structure

- Cache line

tag	data block	flag bits (valid, dirty)

- Indexing

tag (40bit)	index (6bit)	block offset (6bit)

- Associativity
  Number of different tags to be kept under the same index

CPU Cache Structure

- Cache line

tag	data block	flag bits (valid, dirty)

- Indexing

tag (40bit)	index (6bit)	block offset (6bit)

- Associativity
  Number of different tags to be kept under the same index
- VIPT Addressing
  Virtually indexed, physically tagged (VIPT)
  - Solve aliasing problem
  - Simultaneous cache and TLB lookup

Interesting reading: `http://igoro.com/archive/gallery-of-processor-cache-effects/`

CPU Cache Structure

Example

```
$ grep -m1 name /proc/cpuinfo
model name      : Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz

$ cat /sys/devices/system/cpu/cpu0/cache/index0/{size,ways_of_associativity,coherency_line_size}
32K
8
64
```

- block offset: $\log_2 64 = 6\text{bit}$
- index: $\log_2(32\text{KiB}/64\text{B}/8) = 6\text{bit}$
- tag: $52 - 6 - 6 = 40\text{bit}$ (48-bit virtual memory and 52-bit physical memory)

Note that *block offset* and *index* together take 12 bits, which is equal to page size (4KiB), so VIPT can be easily implemented.

A MatMul Example

```
void matmul(float *a, float *b, float *c, int n) {
    for (int i = 0; i < n; ++ i) {
        for (int j = 0; j < n; ++ j) {
            float sum = 0;
            for (int k = 0; k < n; ++ k) {
                sum += a[i * n + k] * b[k * n + j];
            }
            c[i * n + j] = sum;
        }
    }
}
```

A MatMul Example

```
void matmul(float *a, float *b, float *c, int n) {
    for (int i = 0; i < n; ++ i) {
        for (int j = 0; j < n; ++ j) {
            float sum = 0;
            for (int k = 0; k < n; ++ k) {
                sum += a[i * n + k] * b[k * n + j];
            }
            c[i * n + j] = sum;
        }
    }
}
```

Swap the loops on *j* and *k*

1. Symbolic Computation
   - Representation
   - Execution & Optimization
2. Dense Numerical Computation
   - CPU Computation
   - Other Computation Devices
   - Computation & Memory Gap
3. Distributed Computation
   - System
   - Optimzation Algorithms
   - Communication Algorithms

NVIDIA GPU

A single instruction, multiple thread architecture

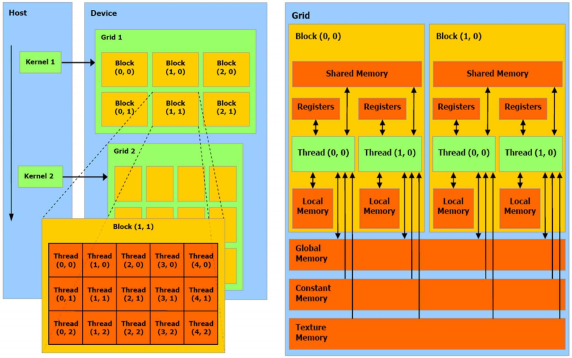


Image from[6]

[6]Marco Nobile et al. "cuTauLeaping: A GPU-Powered Tau-Leaping Stochastic Simulator for Massive Parallel Analyses of Biological Systems". In: 9 (Mar. 2014), e91963.

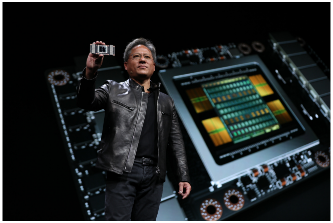
NVIDIA GPU
A single instruction, multiple thread architecture

```
__global__ void add(float *a, float *b, float *c, int n) {
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    if (id < n)
        c[id] = a[id] + b[id];
}
```

- **Memory Coalescing**
  Adjacent threads access adjacent memories simultaneously
- **Parallelism**
  Divide the total work among many tiny threads

NVIDIA GPU

A single instruction, multiple thread architecture



Tesla V100 for NVLink @ 300W

- 15.7 *TFLOPS* for single-precision
- 125 *TFLOPS* for half-precision

Tesla T4 @ 75W

- 130T INT8
- 260T INT4

Image from https://arstechnica.com/gadgets/2017/05/nvidia-tesla-v100-gpu-details/

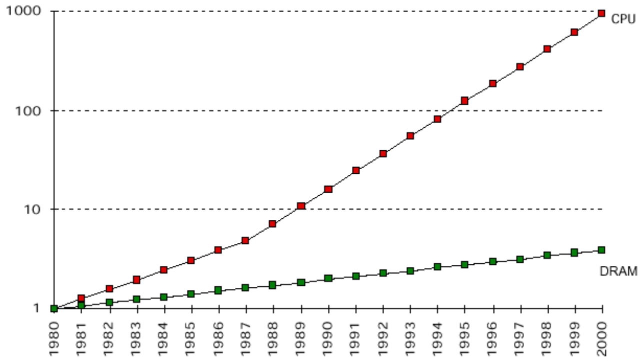
The Trend

- On cloud: high density computation
  e.g. Google TPU 3.0 pods are claimed to achieve 100PFLOPS
- On edge: low precision
  e.g. int8 in cDSP supported by Qualcomm's SNPE
- Automatic kernel tuning & generation
  An active research area. Typical projects include Halide[6], TVM[7] and TensorComprehension[8]

[6] `http://halide-lang.org/`
[7] `http://tvmlang.org/`
[8] `https://facebookresearch.github.io/TensorComprehensions/`

1. Symbolic Computation
   - Representation
   - Execution & Optimization
2. Dense Numerical Computation
   - CPU Computation
   - Other Computation Devices
   - Computation & Memory Gap
3. Distributed Computation
   - System
   - Optimzation Algorithms
   - Communication Algorithms

Unbalanced Development of Processor and Memory



Graph from[9]

[9] Carlos Carvalho. "The gap between processor and memory speeds". In: *Proc. of IEEE International Conference on Control and Automation*. 2002.

Challenges from NN Architecture

Computation-sparse structure seems to be beneficial.

Architecture	Computation	Memory
Small kernel	$\frac{k2^2}{k1^2}$	Param $\frac{k1^2}{k2^2}$
Large stride	$\frac{1}{s^2}$	Output $\frac{1}{s^2}$
Group/depthwise conv	$\frac{1}{g^2}$	Param $\frac{1}{g}$
Shuffle/concat	0	1

Roofline Model

A visualization method to characterize computation/memory

Performance P (FLOPS) is approximately a function of arithmetic intensity I (FLOP/byte) for a particular architecture[10].

Naïve Roofline

$$P = \min \begin{cases} \pi \\ \beta \times I \end{cases}$$

where π is the peak computing performance and β is the peak bandwidth.

[10] Samuel Webb Williams. *Auto-tuning performance on multicore computers.* University of California, Berkeley, 2008.

Roofline Model

A visualization method to characterize computation/memory

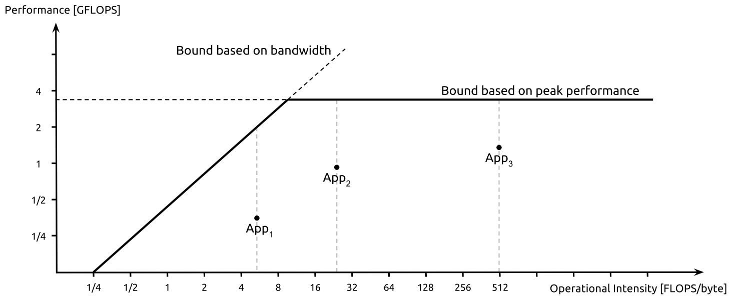


Image from `https://en.wikipedia.org/wiki/Roofline_model`

1. Symbolic Computation
   - Representation
   - Execution & Optimization
2. Dense Numerical Computation
   - CPU Computation
   - Other Computation Devices
   - Computation & Memory Gap
3. Distributed Computation
   - System
   - Optimzation Algorithms
   - Communication Algorithms

Communication System: Single Node

- PCI-e: connection between GPUs, network adaptors and others
  - Switches may be needed
  - 985 MiB/s each PCI-e 3.0 lane
  - LGA-2011 socket: 40 lanes
- NVLink: GPU interconnect by NVIDIA

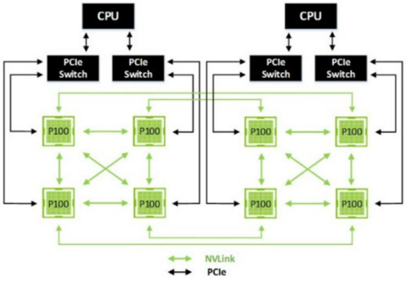


Image from https://www.nextplatform.com/2016/05/04/nvlink-takes-gpu-acceleration-next-level/

Communication System: LAN

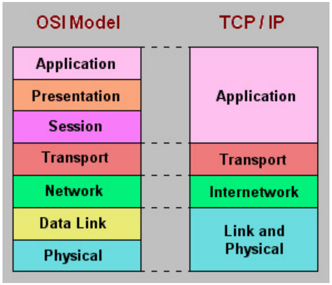


Image from `http://www.just2good.co.uk/tcpipStack.php`

- Ethernet: 10M to 100G, latency 100 - 20 μs
- InfiniBand: 2.5 to 250G, latency 5 - 0.5 μs

RDMA

Remote Direct Memory Access

Bypass the TCP/IP stack and free CPU from handling packets.

RDMA
Remote Direct Memory Access

Bypass the TCP/IP stack and free CPU from handling packets.

- RoCE: RDMA over Converged Ethernet
- InfiniBand: RDMA supported
- NVIDIA GPUDirect: RDMA between GPUs

1. Symbolic Computation
    - Representation
    - Execution & Optimization
2. Dense Numerical Computation
    - CPU Computation
    - Other Computation Devices
    - Computation & Memory Gap
3. Distributed Computation
    - System
    - Optimzation Algorithms
    - Communication Algorithms

Synchronous SGD
Basics

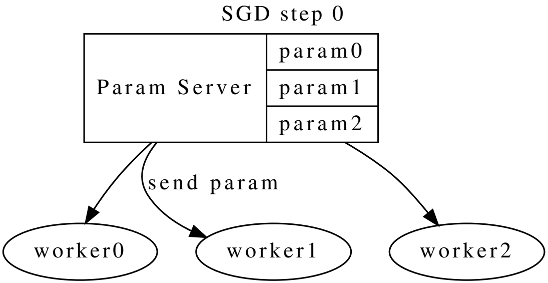
Each worker processes a part of the whole batch; params and grads must be synchronized

$$L(\{d_0, d_1\}, W) = \alpha_0 L(\{d_0\}, W) + \alpha_1 L(\{d_1\}, W)$$

Synchronous SGD
Basics

Each worker processes a part of the whole batch; params and grads must be synchronized

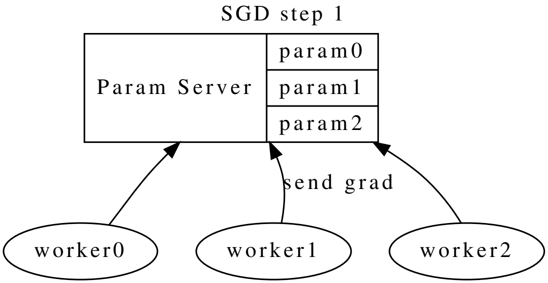
$$L(\{d_0, d_1\}, W) = \alpha_0 L(\{d_0\}, W) + \alpha_1 L(\{d_1\}, W)$$



Synchronous SGD
Basics

Each worker processes a part of the whole batch; params and grads must be synchronized

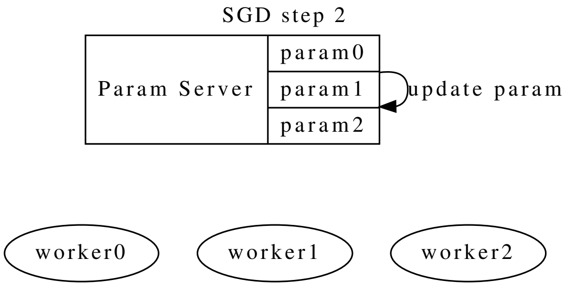
$$L(\{d_0, d_1\}, W) = \alpha_0 L(\{d_0\}, W) + \alpha_1 L(\{d_1\}, W)$$



Synchronous SGD
Basics

Each worker processes a part of the whole batch; params and grads must be synchronized

$$L(\{d_0, d_1\}, W) = \alpha_0 L(\{d_0\}, W) + \alpha_1 L(\{d_1\}, W)$$



Asynchronous SGD
Basics

Each worker has an outdated local copy of params and updates central param storage asynchronously. Friendly for parallel speedup.

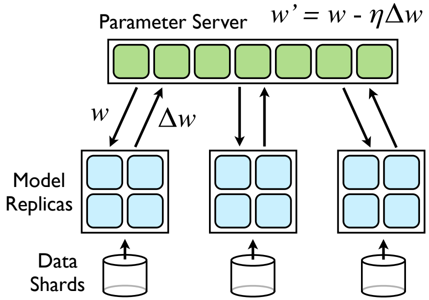


Image from[10]

[10] Jeffrey Dean et al. "Large scale distributed deep networks". In: *NIPS*. 2012, pp. 1223–1231.

Asynchronous SGD
Difficulties

ASGD is not equivalent to SGD and it is hard to tune due to noisy gradients. Many works exist on analyzing convergence and improving performance[11][12][13].

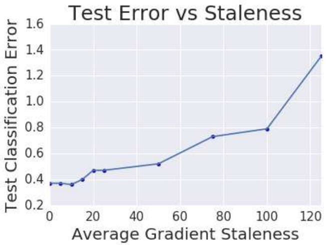


Image from[14]

[11] Xiangru Lian et al. “Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization”. In: *NIPS*. 2015, pp. 2737–2745.

[12] Wei Zhang et al. “Staleness-aware async-SGD for Distributed Deep Learning”. In: *IJCAI*. 2016, pp. 2350–2356.

[13] Sixin Zhang, Anna E Choromanska, and Yann LeCun. “Deep learning with elastic averaging SGD”. In: *NIPS*. 2015, pp. 685–693.

[14] Jianmin Chen et al. “Revisiting distributed synchronous SGD”. In: *arXiv preprint arXiv:1604.00981* (2016).

Synchronous SGD

Reduce communication by compressing gradients[15]

$$Q_s(v_i) = \|\boldsymbol{v}\|_2 \cdot \mathrm{sgn}(v_i) \cdot \xi_i(\boldsymbol{v}, s)$$

[15] Ryota Tomioka and Milan Vojnovic. "QSGD: Communication-Efficient Stochastic Gradient Descent, with Applications to Training Neural Networks". In: *arXiv preprint arXiv:1610.02132* (2016).

Synchronous SGD

Handle straggling workers by backup workers[16]

Use $N + b$ workers but only receive gradients from any N of them and do not wait for the slowest b workers.

[16] Jianmin Chen et al. "Revisiting distributed synchronous SGD". In: *arXiv preprint arXiv:1604.00981* (2016).

Synchronous SGD
Ensure accuracy by careful hyperparam tuning

8192 minibatch size on 256 GPUs[17]

$$\hat{\eta} = k\eta$$
$$m = \frac{\eta_{t+1}}{\eta_t}$$

$32K$ minibatch size with LARS[18]

$$\lambda^l = \eta \times \frac{\|w^l\|}{\|\Delta L(w^l)\|}$$

[17] Priya Goyal et al. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. In: *arXiv preprint arXiv:1706.02677* (2017).

[18] Yang You, Igor Gitman, and Boris Ginsburg. “Large batch training of convolutional networks”. In: *arXiv preprint arXiv:1708.03888* (2017).

1. Symbolic Computation
   - Representation
   - Execution & Optimization
2. Dense Numerical Computation
   - CPU Computation
   - Other Computation Devices
   - Computation & Memory Gap
3. Distributed Computation
   - System
   - Optimzation Algorithms
   - Communication Algorithms

MPI Primitives

Collective communication routines in MPI are common in distributed DL

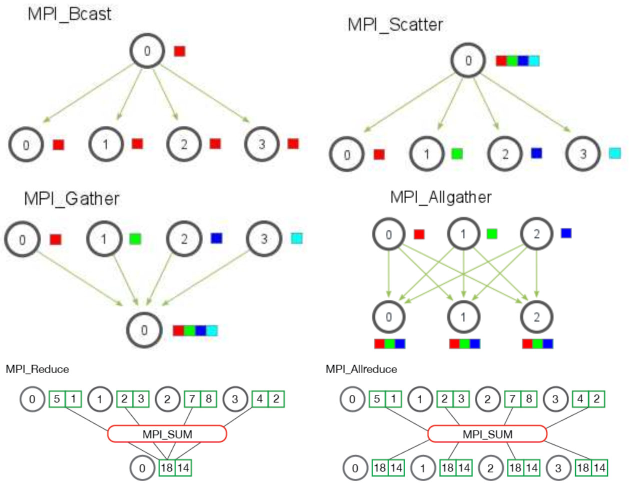


Image source: `http://mpitutorial.com`

An AllReduce Algorithm

Assume message size K and number of workers N

- Reduce to a worker (assume W_{N-1} here):
  W_i sends to W_{i+1} at step i; communication per worker is N
- Broadcast from a worker: as above

An AllReduce Algorithm

Assume message size K and number of workers N

- Reduce to a worker (assume W_{N-1} here):
  W_i sends to W_{i+1} at step i; communication per worker is N
- Broadcast from a worker: as above
- AllReduce:
  1. Split the message into N parts
  2. Reduce the ith part to W_i; all reductions run in parallel
  3. Broadcast each reduced part to all workers in parallel

  Communication cost for each worker is $2(N-1)\frac{K}{N}$, independent of N.

More details and discussions are given in[19] [20].

[19] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH". In: *The International Journal of High Performance Computing Applications* 19.1 (2005), pp. 49–66.

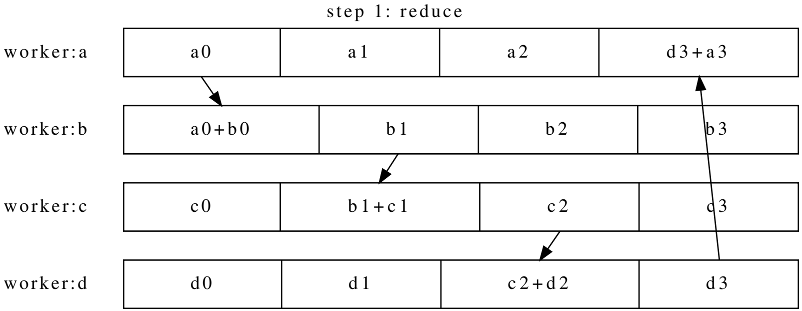
[20] `http://research.baidu.com/bringing-hpc-techniques-deep-learning`

An AllReduce Algorithm

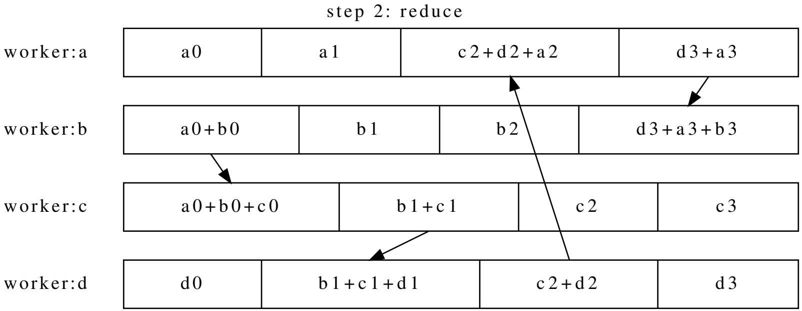
step 0: init

worker:a	a0	a1	a2	a3
worker:b	b0	b1	b2	b3
worker:c	c0	c1	c2	c3
worker:d	d0	d1	d2	d3

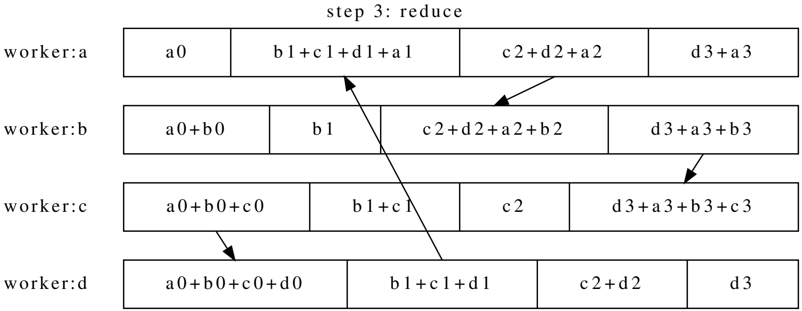
An AllReduce Algorithm



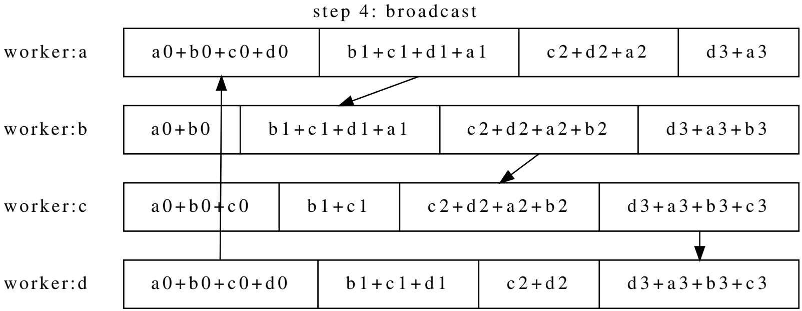
An AllReduce Algorithm



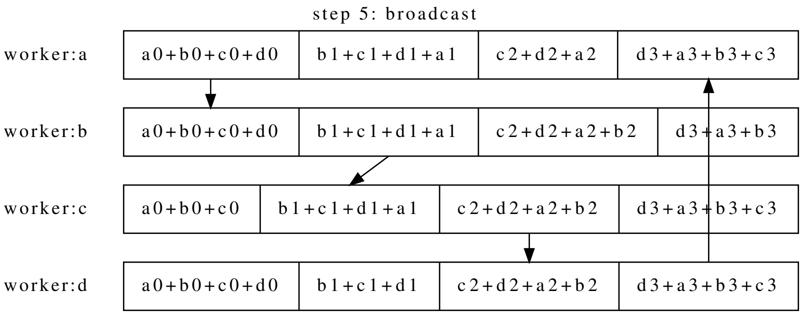
An AllReduce Algorithm



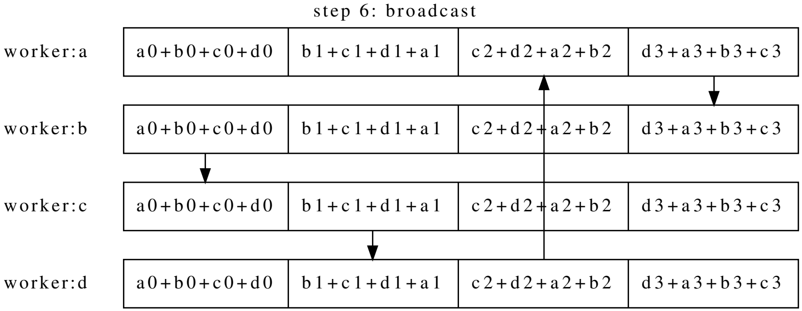
An AllReduce Algorithm



An AllReduce Algorithm



An AllReduce Algorithm



Thanks!

Questions and feedback are welcome :)