Scaling Infrastructure for Big Data Analysis

Motivation

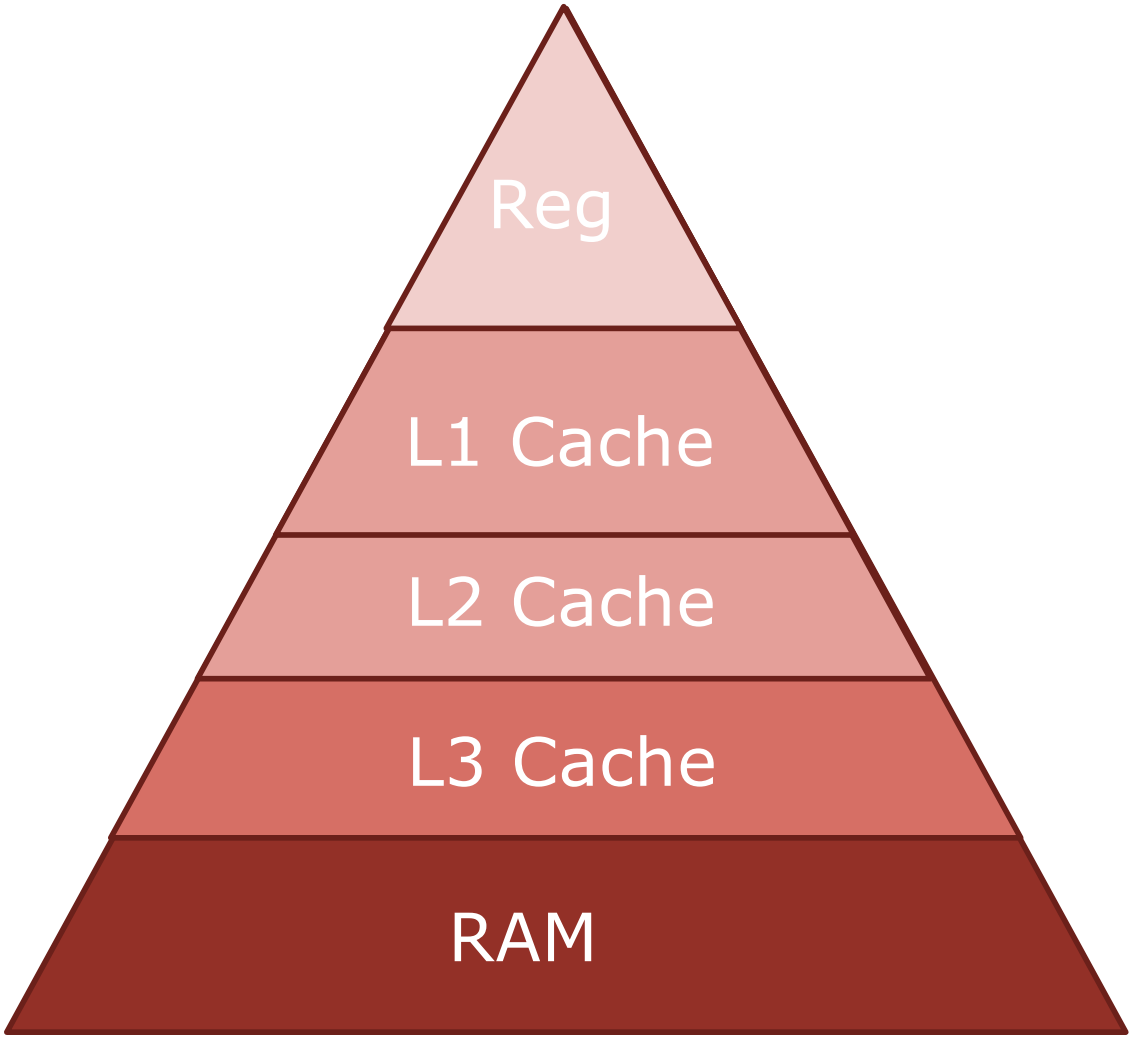
1. **Velocity**
2. **Volume**
3. Value
4. Variety
5. Veracity

Motivation

We have actual big data

We have hard real time requirements

The real problem

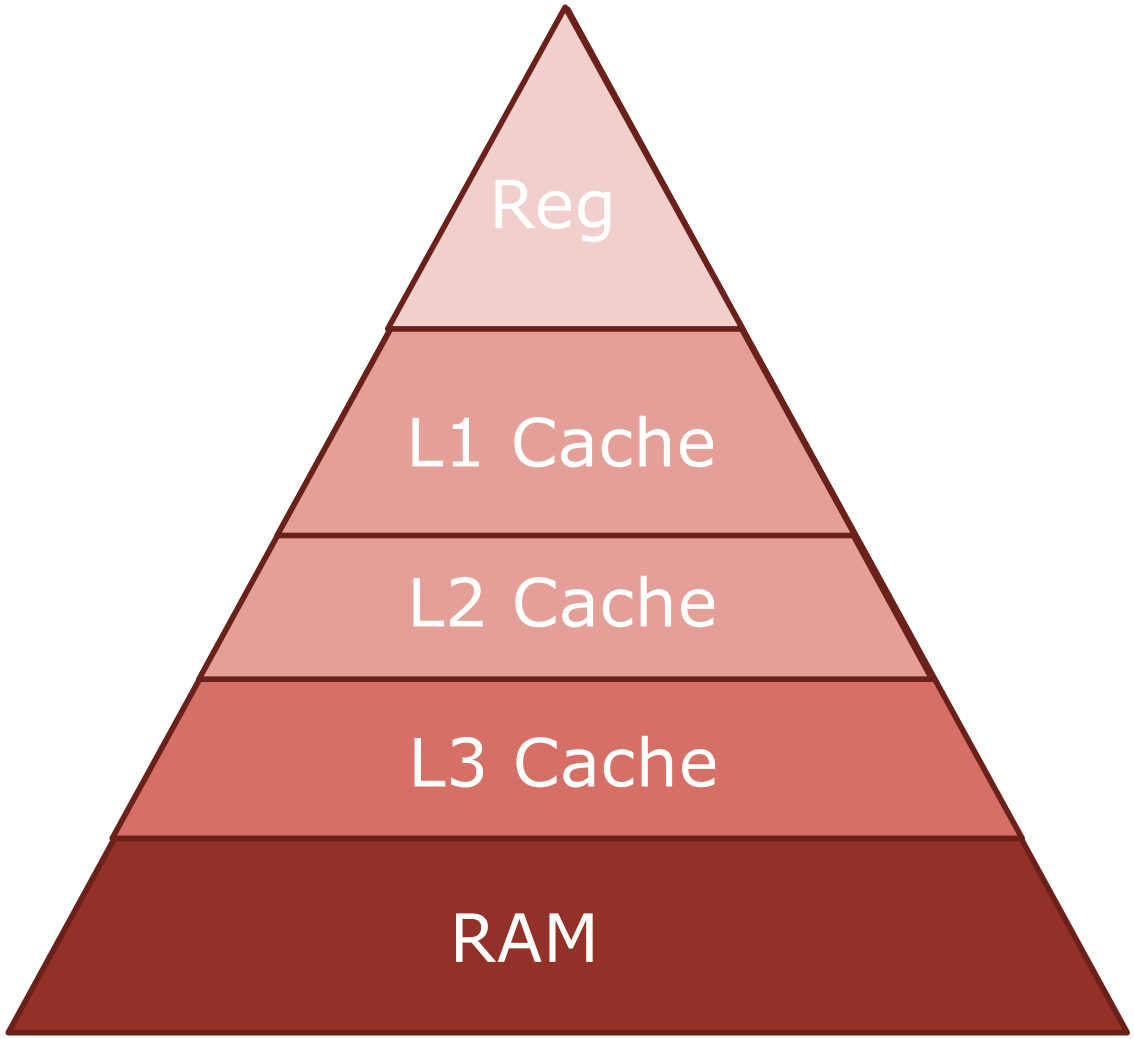


Why don’t we just get faster CPUs?

End-to-end

Processor @3.3Ghz app 0.1 ns pr instruction
L1 Data Cache Latency = 4 cycles
L2 Cache Latency = 12 cycles
L3 Cache Latency = 36 cycles (3.4 GHz i7-4770)
RAM Latency = 36 cycles + 57 ns (3.4 GHz i7-4770)

The real problem



Wasted inst

0

12

36

108

678

Reading for stable storage

SSD access is app 100 ns
HDD access is app 5 ms

Three approaches to a solution

Stream data using vectorization

Drop the cache approach and and introduce scratch memory

Drop the Von Neumann approach

Three approaches to a solution

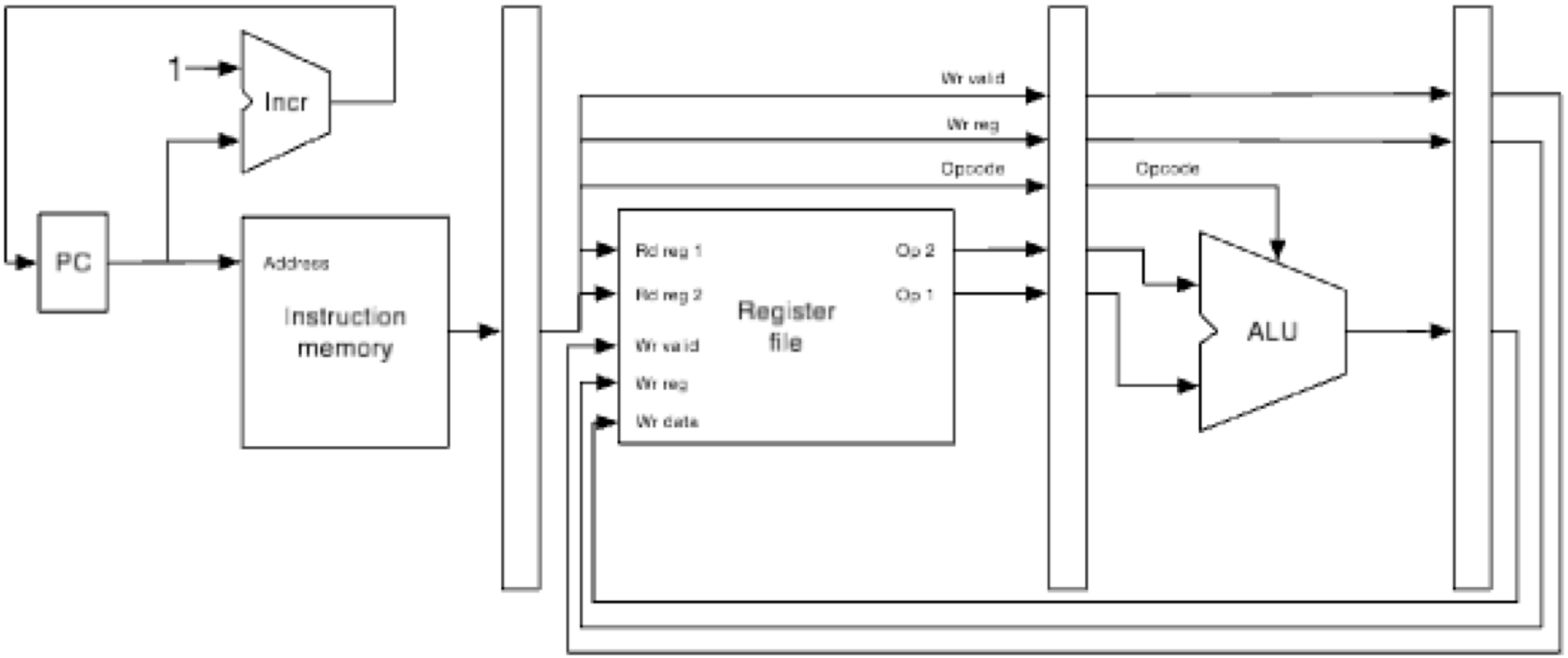
Stream data using vectorization

Drop the cache approach and and introduce scratch memory

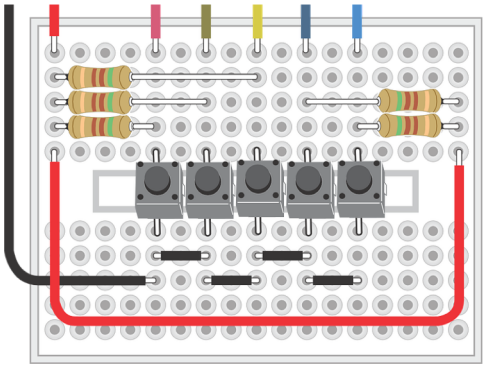
Drop the Von Neumann approach

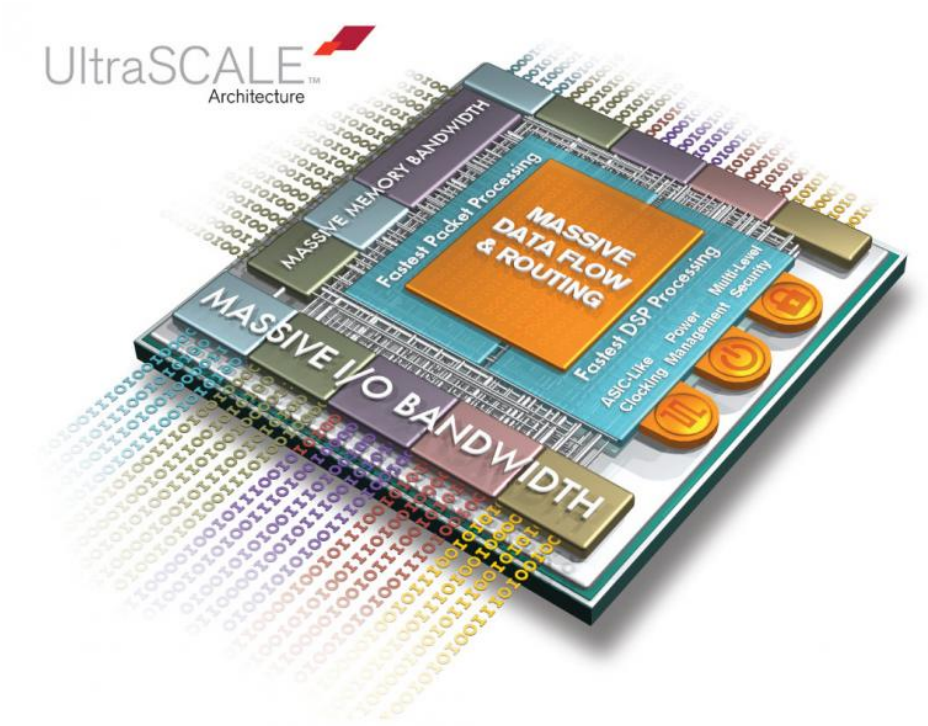
Von Neumann

```
mov eax, 0x80000000
mov ecx, 0x80000000
mov ebx, 0
mov edx, 0
add eax, ecx
adc edx, ebx
```



FPGA




UltraSCALE
Architecture
MASSIVE MEMORY BANDWIDTH
Fastest Packet Processing
MASSIVE DATA FLOW & ROUTING
Fastest DSP Processing
Multi-Level Security
Power Management
ASIC-Like Clocking
MASSIVE I/O BANDWIDTH

Why FPGA?

Performance is greater than the best GPGPUs

- But at 1/10th the electricity consumption

Superior memory bandwidth

- Current best is 460GB/sec (PC is less than 125GB/sec)

Extremely robust

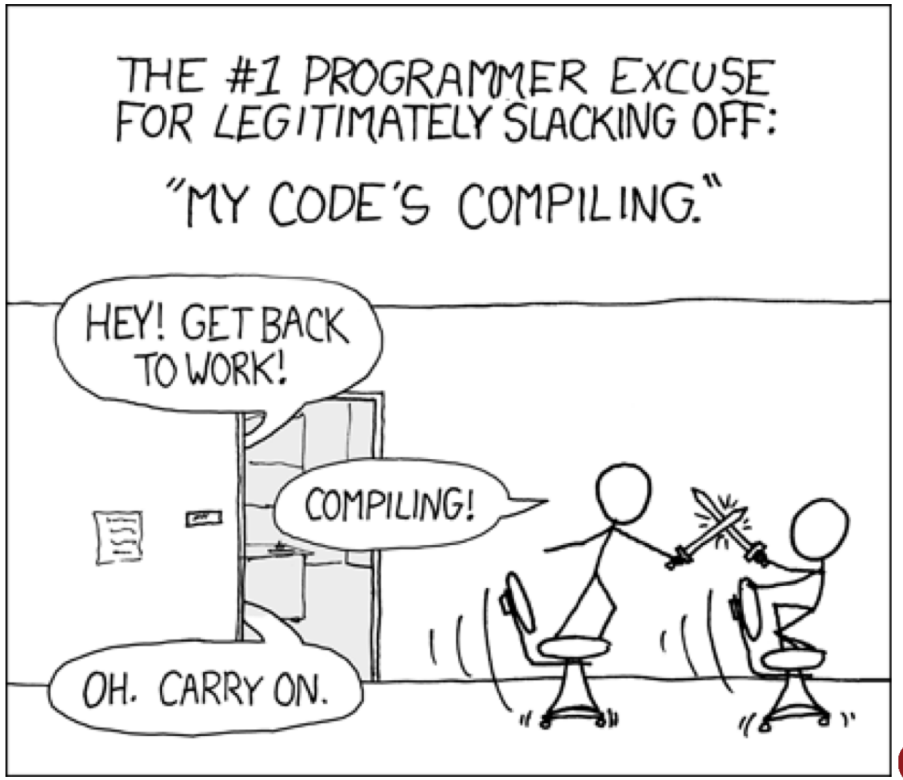
Why not FPGA?

Your whole program must be in the FPGA

- You still have main memory for your data

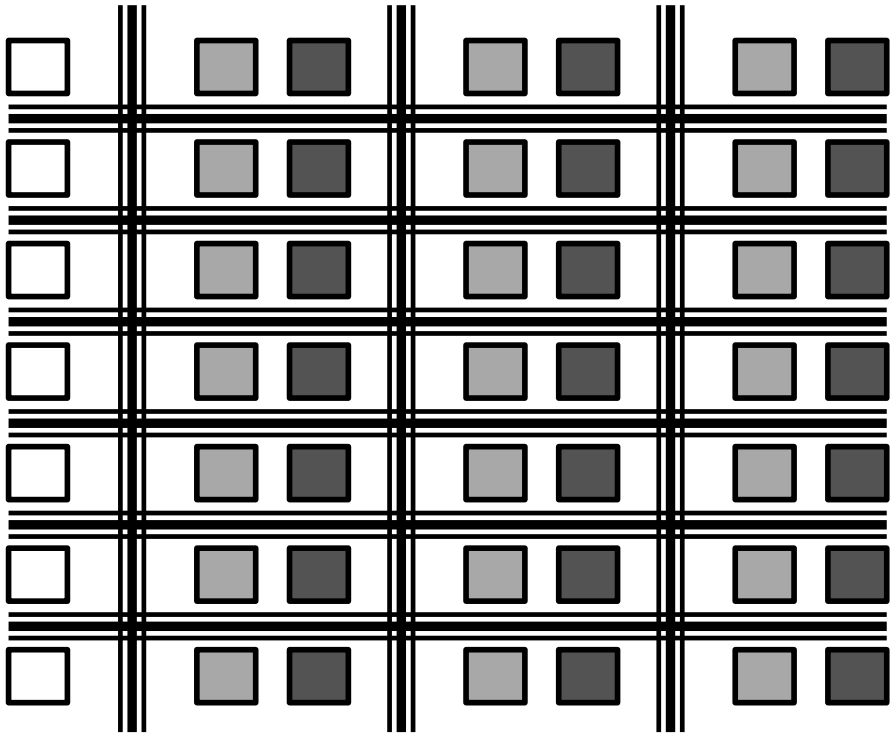
It is hard to not consider timing

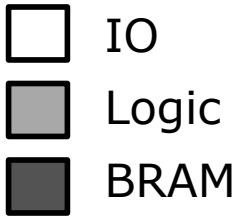
Compiling takes ~~hours~~ days



https://xkcd.com/303/

FPGA





Programming FPGA

VHDL and Verilog are the defacto standards for programming FPGA

```
type t11 is array (0 to numcols1-1) of unsigned(15 downto 0);
type t1 is array (0 to numrows1-1) of t11;
type t22 is array (0 to numcols2-1) of unsigned(15 downto 0);
type t2 is array (0 to numrows2-1) of t22;
type t33 is array (0 to numcols3-1) of unsigned(31 downto 0);
type t3 is array (0 to numrows3-1) of t33;
```

```
function  matmul  ( a : t1; b:t2 ) return t3 is
variable i,j,k : integer:=0;
variable prod : t3:=(others => (others => (others => '0')));
begin
for i in 0 to numrows1-1 loop
for j in 0 to numcols2-1 loop
for k in 0 to numcols1-1 loop
   prod(i)(j) := prod(i)(j) + (a(i)(k) * b(k)(j));
end loop;
end loop;
end loop;
return prod;
end matmul;
```

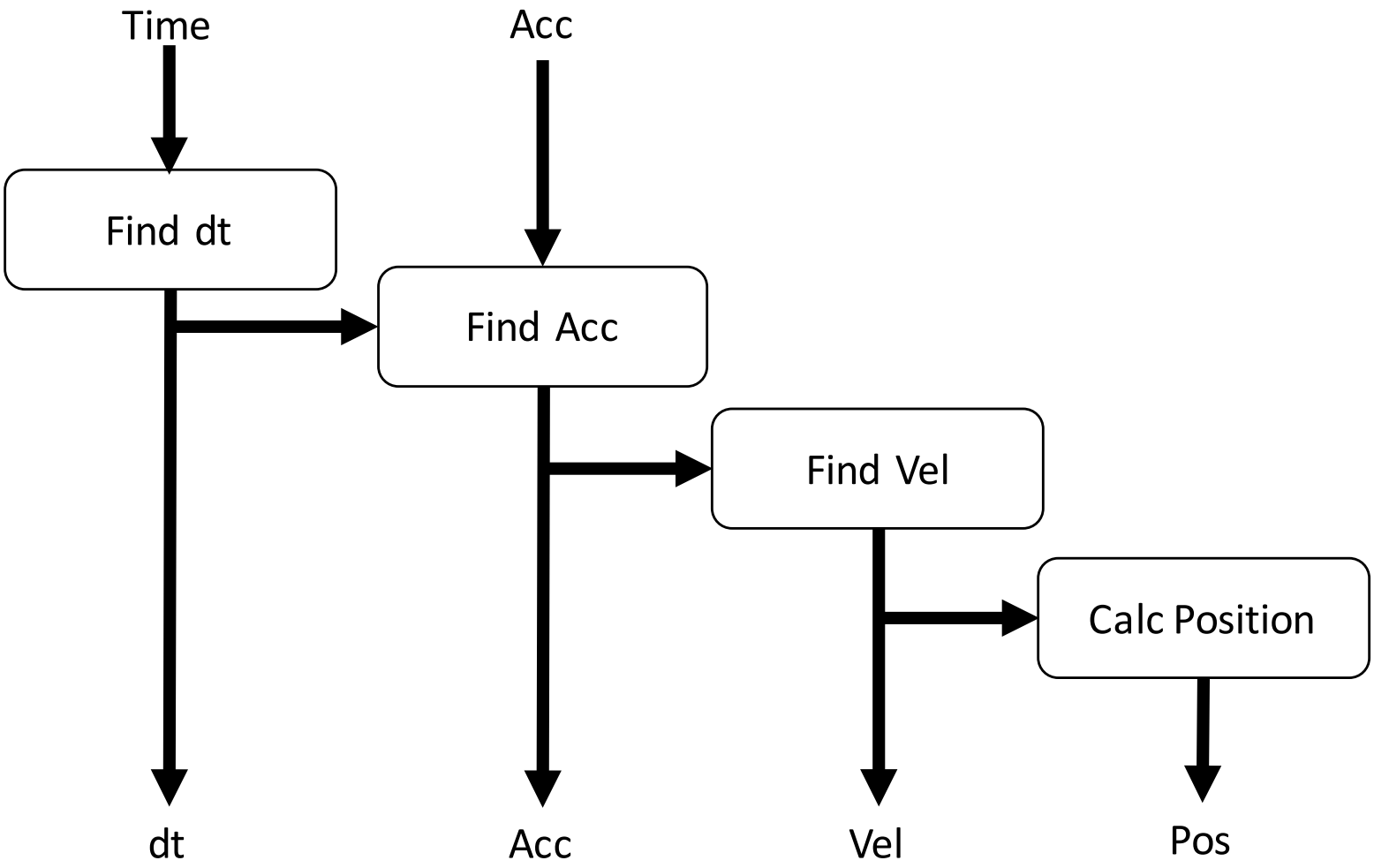
From: http://vhdlguru.blogspot.dk/2010/03/matrix-multiplication-in-vhdl.html

Example for FPGAs

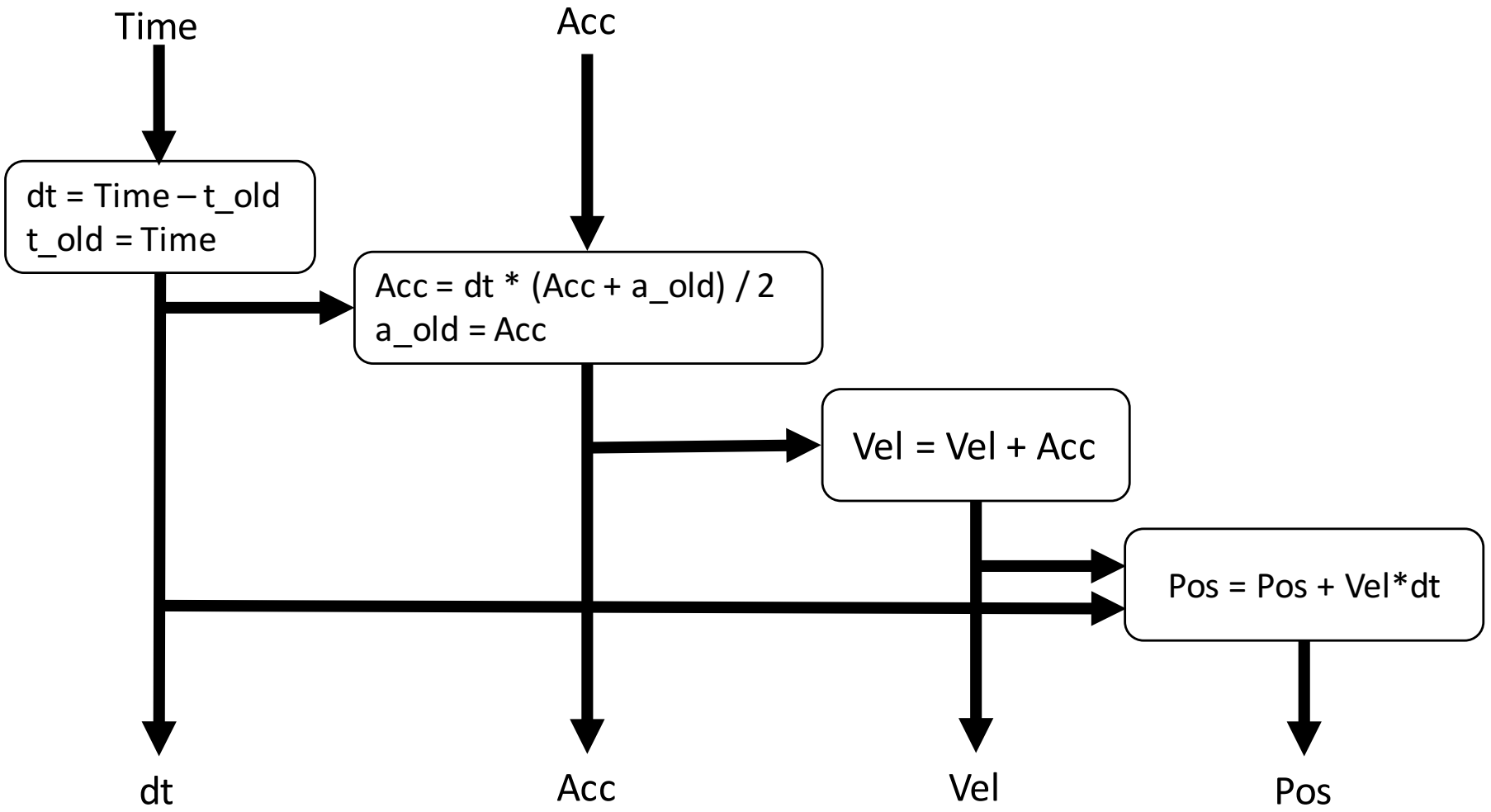
Random walk chip

The drunken sailor approach

Layout


Time
Acc
Find dt
Find Acc
Find Vel
Calc Position
dt
Acc
Vel
Pos

With Code


Time
Acc
dt = Time – t_old
t_old = Time
Acc = dt * (Acc + a_old) / 2
a_old = Acc
Vel = Vel + Acc
Pos = Pos + Vel*dt
dt
Acc
Vel
Pos

SME

```
from SME import Bus, Network, Function, External

class time_producer(External):
  def setup(self, args):
    self.output = args[0]
    self.time = 0
    self.output['time'] = self.time

  def run(self):
    import random
    self.time += random.random()
    self.output['time'] = self.time

class acc_producer(External):
  def setup(self, args):
    self.output = args[0]
    self.output['obacc'] = 0

  def run(self):
    import random
    self.output['obacc'] = random.random() - 0.5
```

SME

```
class dt(Function):
    def setup(self, args):
        self.input, self.output = args
        self.time = 0
        self.output['dt'] = self.time

    def run(self):
        self.output['dt'] = self.input['time'] - self.time
        self.time = self.input['time']

class Acc(Function):
    def setup(self, args):
        self.dt, self.input, self.output = args
        self.acc = 0
        self.output['acc'] = self.acc

    def run(self):
        self.output['acc'] = self.dt['dt'] * (self.input['obacc'] + self.acc) / 2
        self.acc = self.input['obacc']
```

SME

```
class Vel(Function):
    def setup(self, args):
        self.acc, self.output = args
        self.vel = 0
        self.output['vel'] = self.vel

    def run(self):
        self.vel += self.acc['acc']
        self.output['vel'] = self.vel

class Pos(Function):
    def setup(self, args):
        self.dt, self.vel, self.output = args
        self.pos = 0
        self.output['pos'] = self.pos

    def run(self):
        self.pos += self.dt['dt'] * self.vel['vel']
        self.output['pos'] = self.pos
```

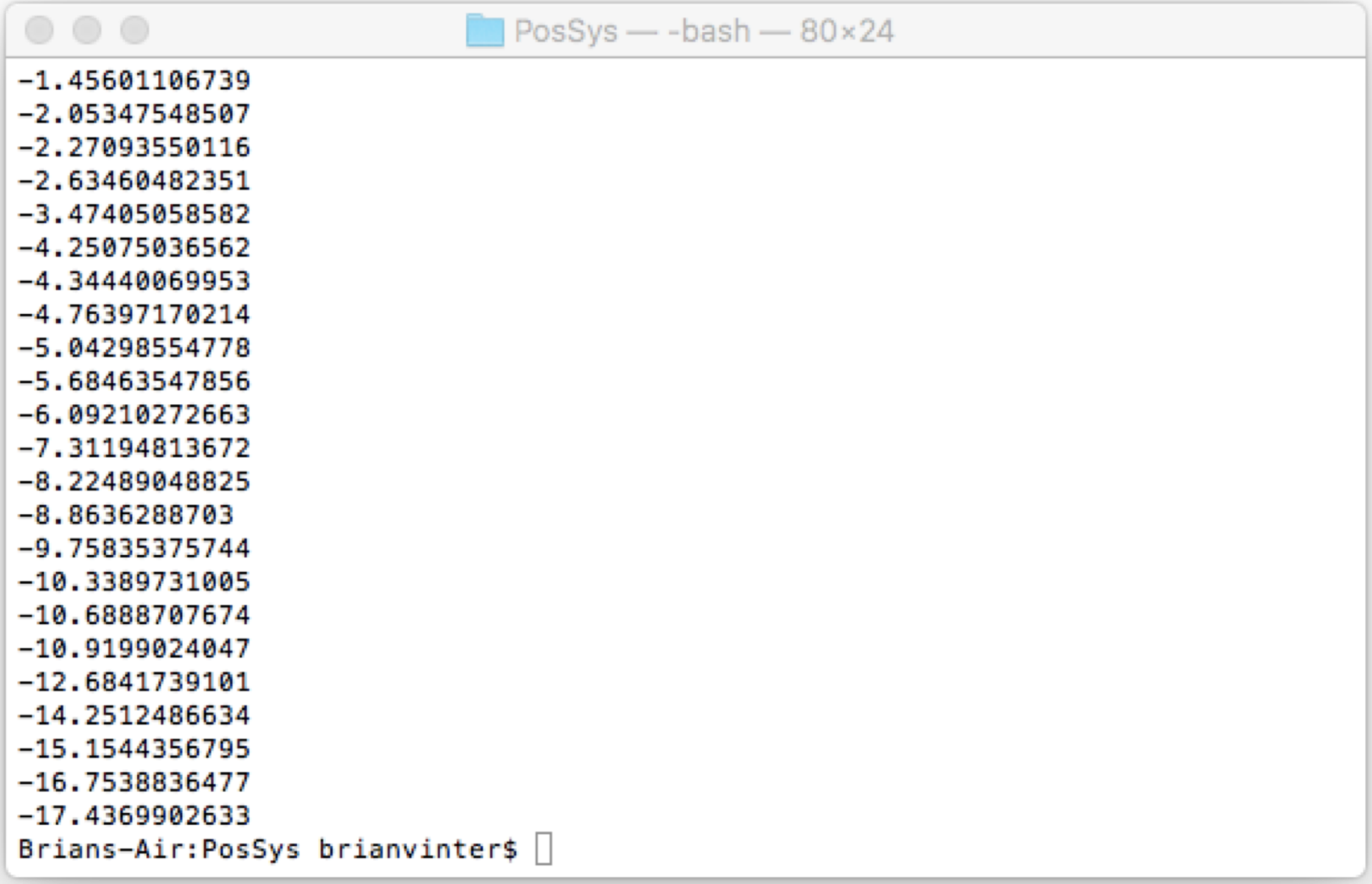
SME

```
class printer(External):
  def setup(self, args):
    self.dbus = args[0]

  def run(self):
      print(self.dbus['pos'])
```

SME

```
class Sailor(Network):
  def wire(self, args):

    self.time_bus = Bus('Time',['time',])
    self.dt_bus = Bus('dt',['dt'])
    self.ObAccbus = Bus('ObAcc',['obacc'])
    self.Accbus = Bus('Acc',['acc'])
    self.Velbus = Bus('Vel',['vel'])
    self.Posbus = Bus('Pos',['pos'])

    self.gentime = time_producer('time', [self.time_bus])
    self.getacc = acc_producer('Phone', [self.ObAccbus])
    self.dt = dt('dt', [self.time_bus, self.dt_bus])
    self.acc = Acc('Acc',[self.dt_bus, self.ObAccbus, self.Accbus])
    self.vel = Vel('Vel', [self.Accbus, self.Velbus])
    self.pos = Pos('Pos', [self.dt_bus, self.Velbus, self.Posbus])
    self.sink = printer('Print', [self.Posbus, 'pos'])

if __name__ == "__main__":

  chip = Sailor('Drunken Sailor')

  chip.clock(100)
```

Run


PosSys — -bash — 80×24
-1.45601106739
-2.05347548507
-2.27093550116
-2.63460482351
-3.47405058582
-4.25075036562
-4.34440069953
-4.76397170214
-5.04298554778
-5.68463547856
-6.09210272663
-7.31194813672
-8.22489048825
-8.8636288703
-9.75835375744
-10.3389731005
-10.6888707674
-10.9199024047
-12.6841739101
-14.2512486634
-15.1544356795
-16.7538836477
-17.4369902633
Brians-Air:PosSys brianvinter$

Other High Productivity Tools

SystemC

MyHDL

Vivado

Accelize

OpenCL for FPGA

Händel-C

Cx

Will this happen?

We will see FPGAs included in systems “very soon”

Microsoft and Amazon Data-centers are all fully FPGA equipped now

Quiz 2C

Looking at acceleration from an algorithm perspective: what kind of algorithms are most easily moved to non-von-Neumann architectures?

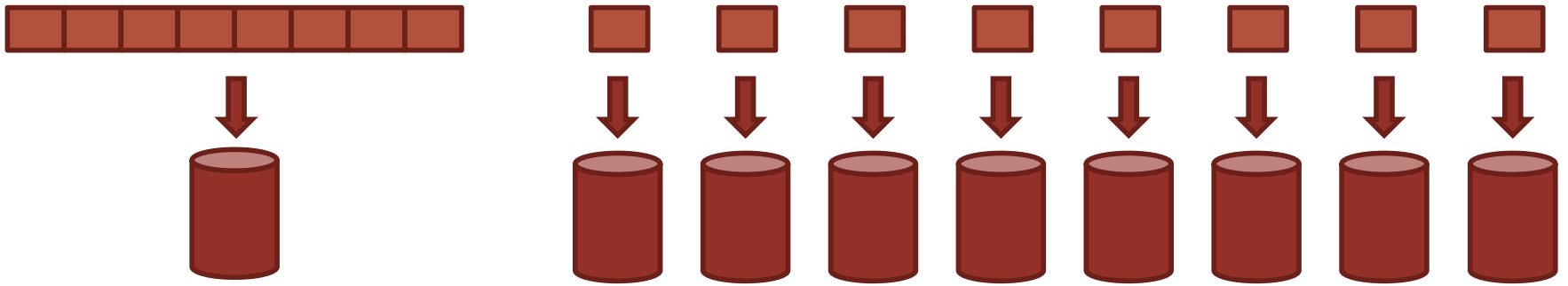
Storage is extremely slow

200-500 MB/sec

How to speed that up?

So how do we make big data go faster?

We parallelize the slowest component!



Map-Reduce

Map-Reduce is a generalized paradigm that has roots to the first functional programming languages

Many things can be done efficiently with map-reduce

- But not everything!!!

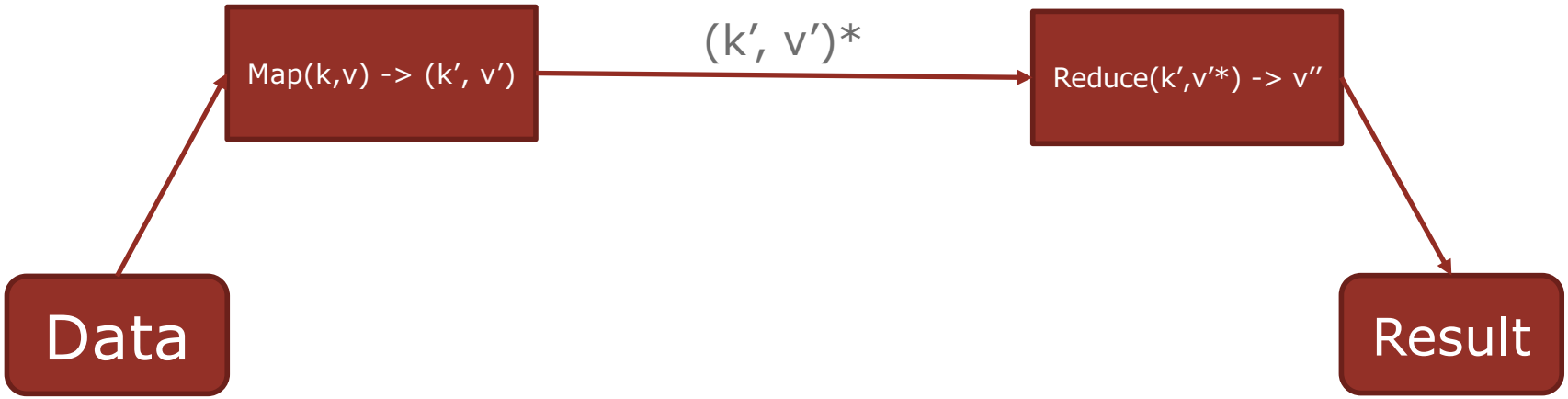
If a problem fits the Map-Reduce paradigm parallelization is mostly trivial

Underlying assumes that we work on key-value tuples

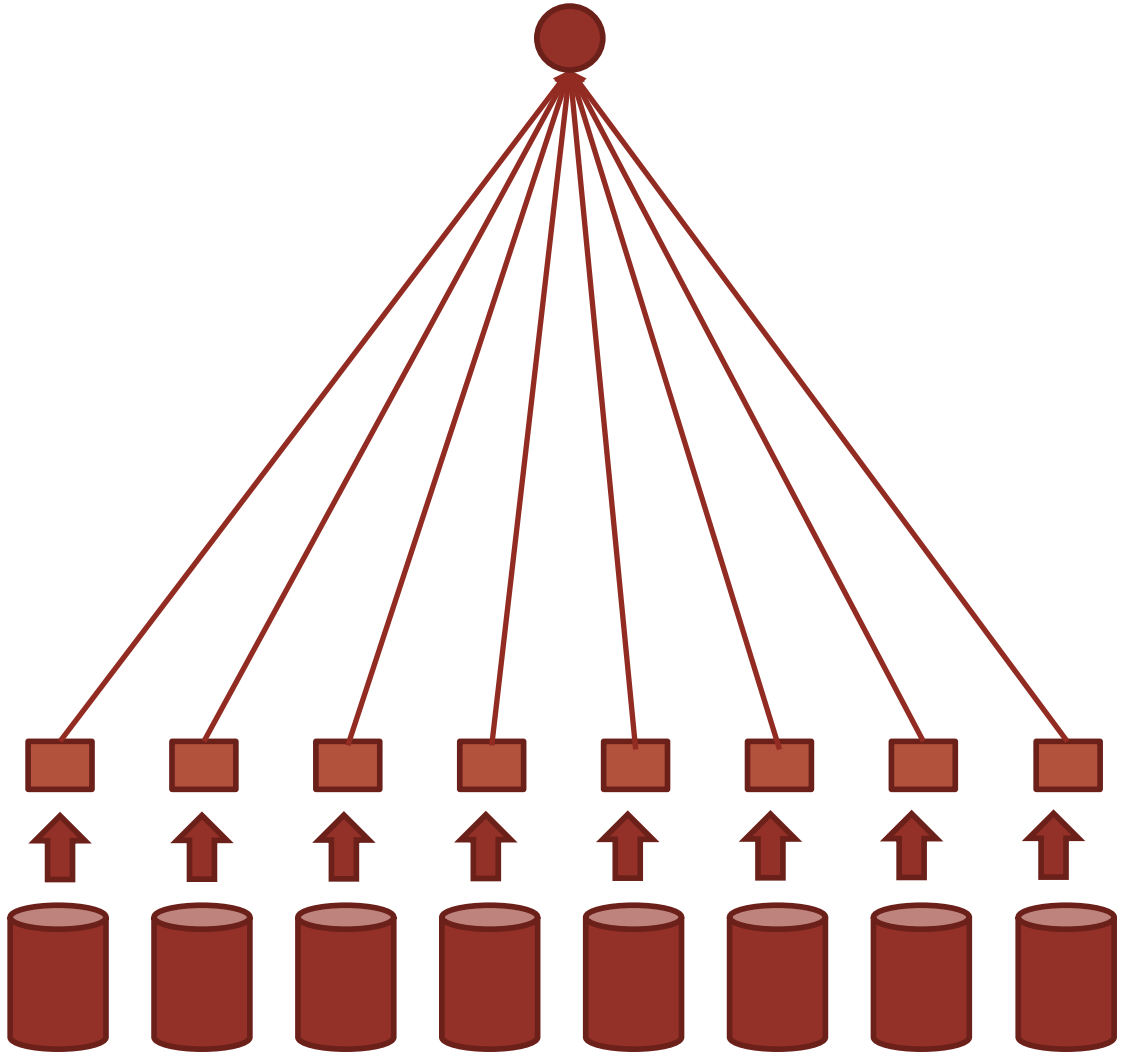
(key, value)

(zip-code, price)

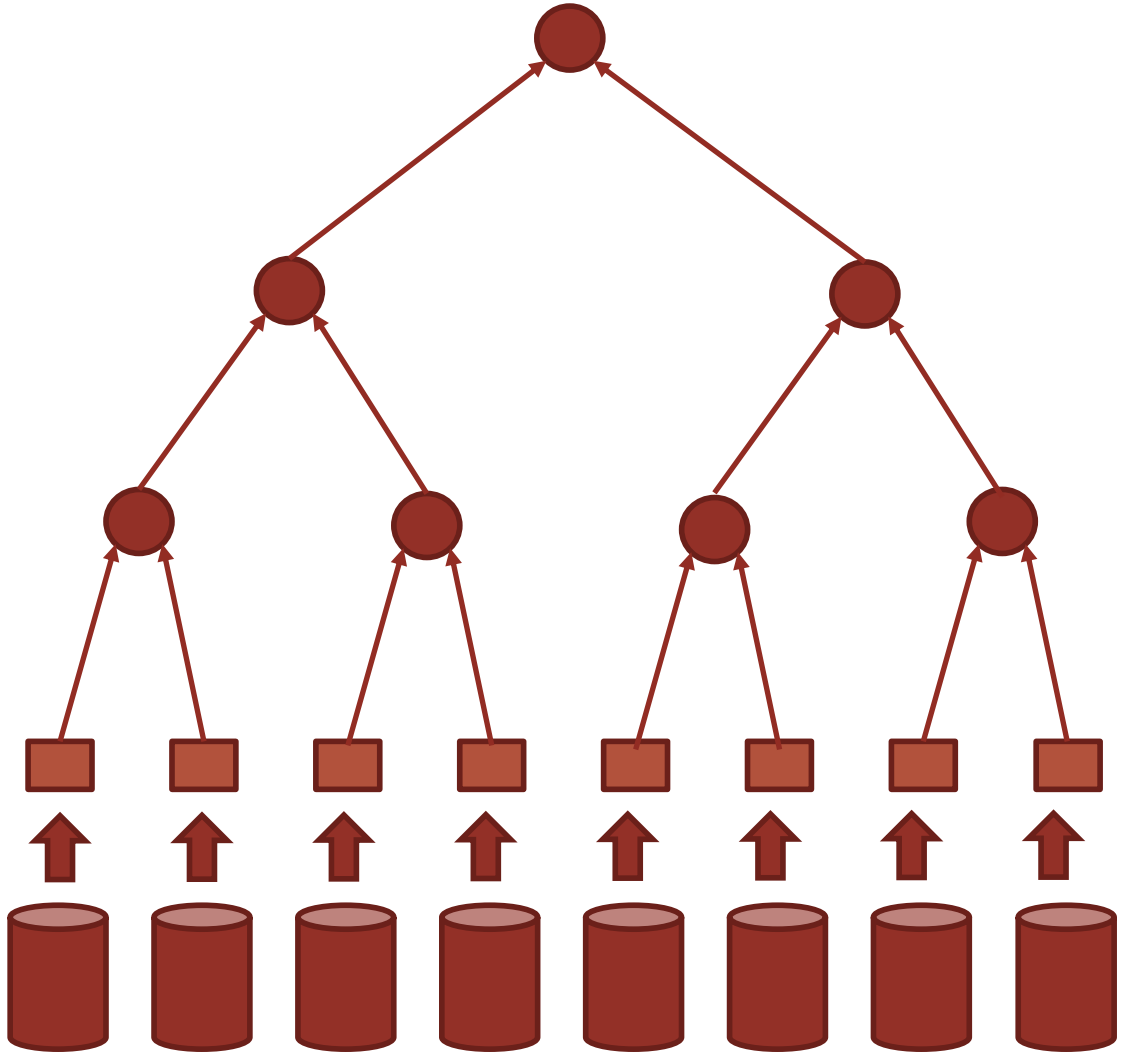
Map-Reduce



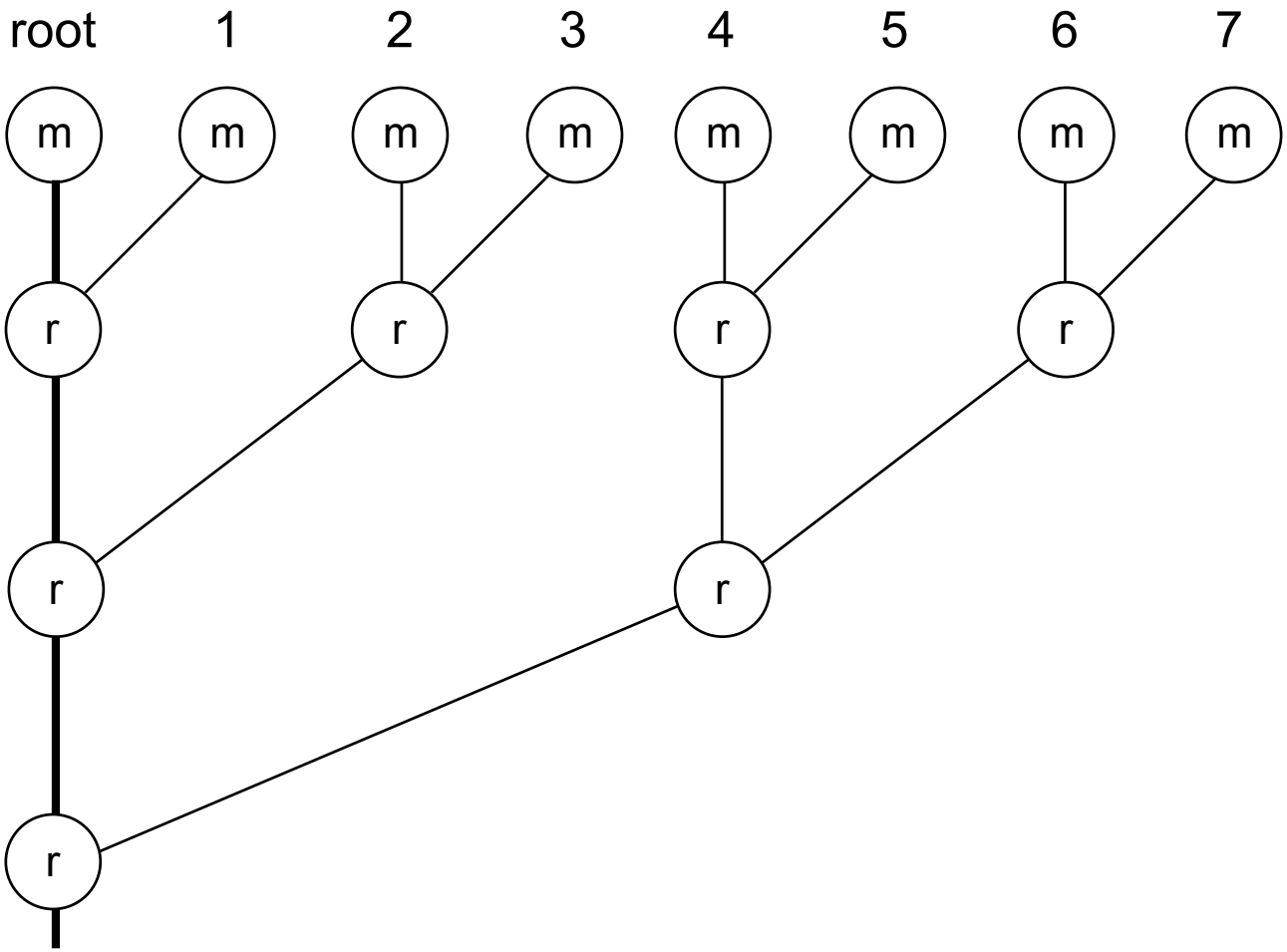
Map-Reduce



Parallel Map-Reduce



Real Reduction Tree



KNN example

satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	sales	salary
0.38	0.53	2	157	3	0	1	0	sales	low
0.8	0.86	5	262	6	0	1	0	sales	medium
0.11	0.88	7	272	4	0	1	0	sales	medium
0.72	0.87	5	223	5	0	1	0	sales	low
0.37	0.52	2	159	3	0	1	0	sales	low
0.41	0.5	2	153	3	0	1	0	sales	low
0.1	0.77	6	247	4	0	1	0	sales	low
0.92	0.85	5	259	5	0	1	0	sales	low
0.89	1	5	224	5	0	1	0	sales	low
0.42	0.53	2	142	3	0	1	0	sales	low
0.45	0.54	2	135	3	0	1	0	sales	low
0.11	0.81	6	305	4	0	1	0	sales	low
0.84	0.92	4	234	5	0	1	0	sales	low

Read data

```
def convert_to_numbers(data):
    category = tuple(set(data))
    return [category.index(key) for key in data]

def read_data():
    db = {}
    with open('HR_comma_sep.csv') as infile:
        dataset = [d for d in csv.reader(infile)]
    keys = dataset.pop(0) #first line holds the keys

    for key in keys:
        db[key] = []
    for data in dataset:
        for key in keys:
            db[key].append(data.pop(0))

    for convert in ['sales', 'salary']:
        db[convert] = convert_to_numbers(db[convert])

    data = numpy.array([numpy.array(db[key]).astype(numpy.float) for key in keys if key != 'left']).T
    data[:, :] = data / numpy.max(data, axis = 0)
    return numpy.array(db['left']).astype(numpy.float), data
```

KNN

```
def all_distances(observation, data):
    return numpy.sqrt((data[:, 0] - observation[0])**2 + (data[:, 1] - observation[1])**2)

def find_k_min(data, labels, k):
    votes = []
    for _ in range(k):
        winner = numpy.argmin(data)
        votes.append(labels[winner])
        data[winner] = 1000
    return collections.Counter(votes).most_common(1)[0][0]
```

Run

```
def knn(data, labels, point, k=5):
    distances = all_distances(data[point], data)
    return find_k_min(distances, labels, k)

labels, data = read_data()
wrong = 0

from time import time
start = time()
for i in range(len(labels)):
    if knn(data, labels, i) != labels[i]:
        wrong += 1
stop = time()

print('Got',wrong,' wrong, out of',len(labels),'. In',stop-start,'sec')
```

Result

```
Got 969  wrong, out of 14999 . In 2.4436898231506348 sec
```

Parallel distance calc

```
def all_distances(param):
    observation, data = param
    return numpy.sqrt((data[:, 0] - observation[0])**2 + (data[:, 1] - observation[1])**2)


def parallel_all_distances(point, db):
    with Pool(cores) as p:
        result =(p.map(all_distances, \
                       zip([point for _ in range(len(db))], [db[i:i + cores] for i in range(0, len(db), cores)])))
    full = []
    _ = [full.extend(i) for i in result]
    return numpy.array(full).astype(numpy.float)
```

Run

```
def knn(data, labels, point, k=5):
    distances = parallel_all_distances(data[point], data)
    return find_k_min(distances, labels, k)

labels, data = read_data()
wrong = 0
from time import time
start = time()
for _ in range(len(labels)//100):
    i = numpy.random.randint(len(labels))
    if knn(data, labels, i) != labels[i]:
        wrong += 1
stop = time()
print('Got',wrong,' wrong, out of 1%. In',stop-start,'sec')
```

Result

```
Got 10  wrong, out of 1%. In 19.565048456192017 sec
```

Parallel test

```
cores = 4
def work(id):
    wrong = 0
    chunk = len(labels)//cores
    start = id*chunk
    stop = start + chunk
    if id==cores-1:
        stop += len(labels)%cores
    for i in range(start, stop):
        if knn(data, labels, i) != labels[i]:
            wrong += 1
    return wrong

from multiprocessing import Pool

from time import time
start = time()
with Pool(cores) as p:
    result = p.map(work, range(cores))
wrong = sum(result)
stop = time()

print('Got',wrong,' wrong, out of',len(labels),'. In',stop-start,'sec')
```

Result

```
Got 969  wrong, out of 14999 . In 0.8174560070037842 sec
```

Hadoop

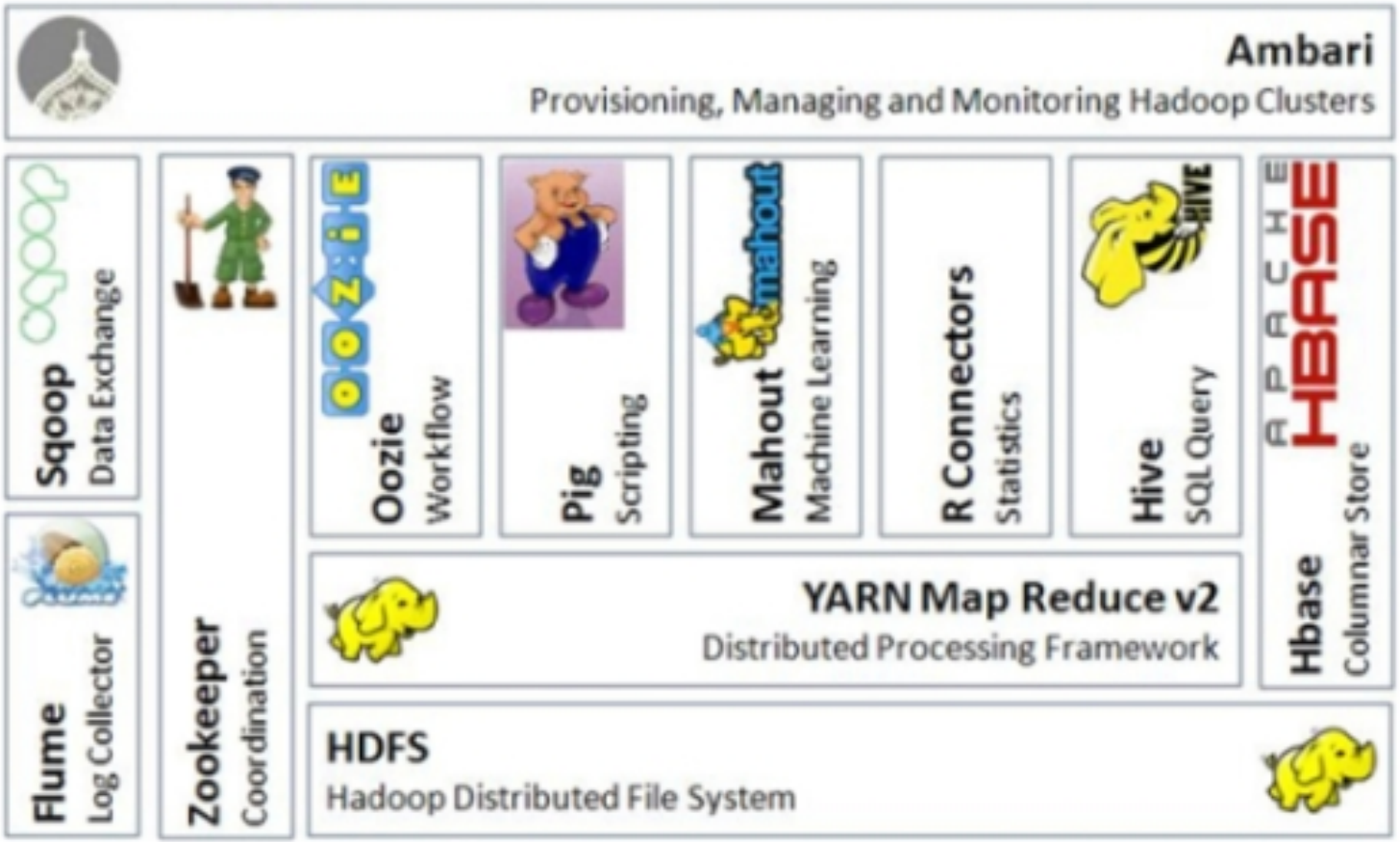
De-facto standard in Big Data Middleware

Scalable

Large Commercial support

Quite mature

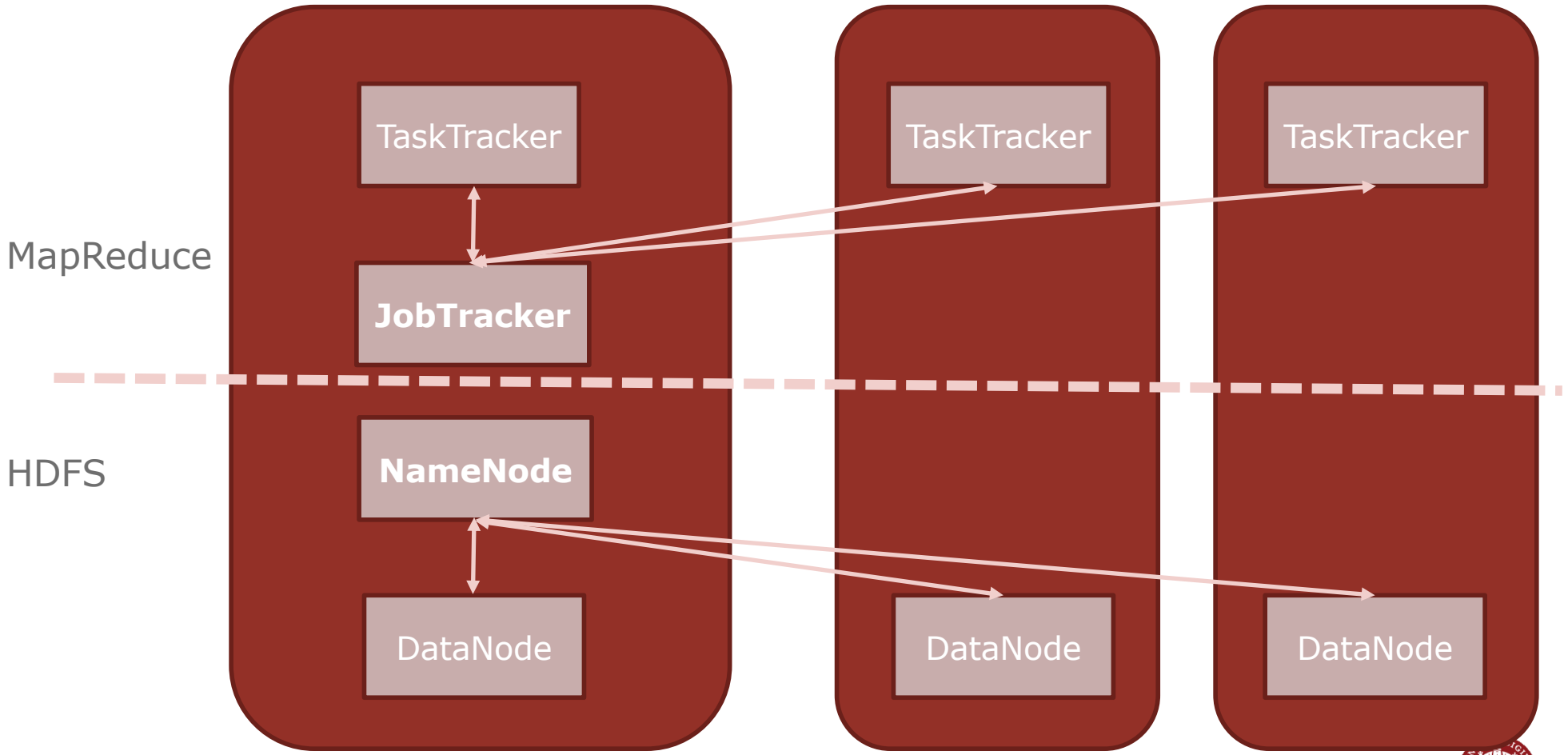
Hadoop Ecosystem



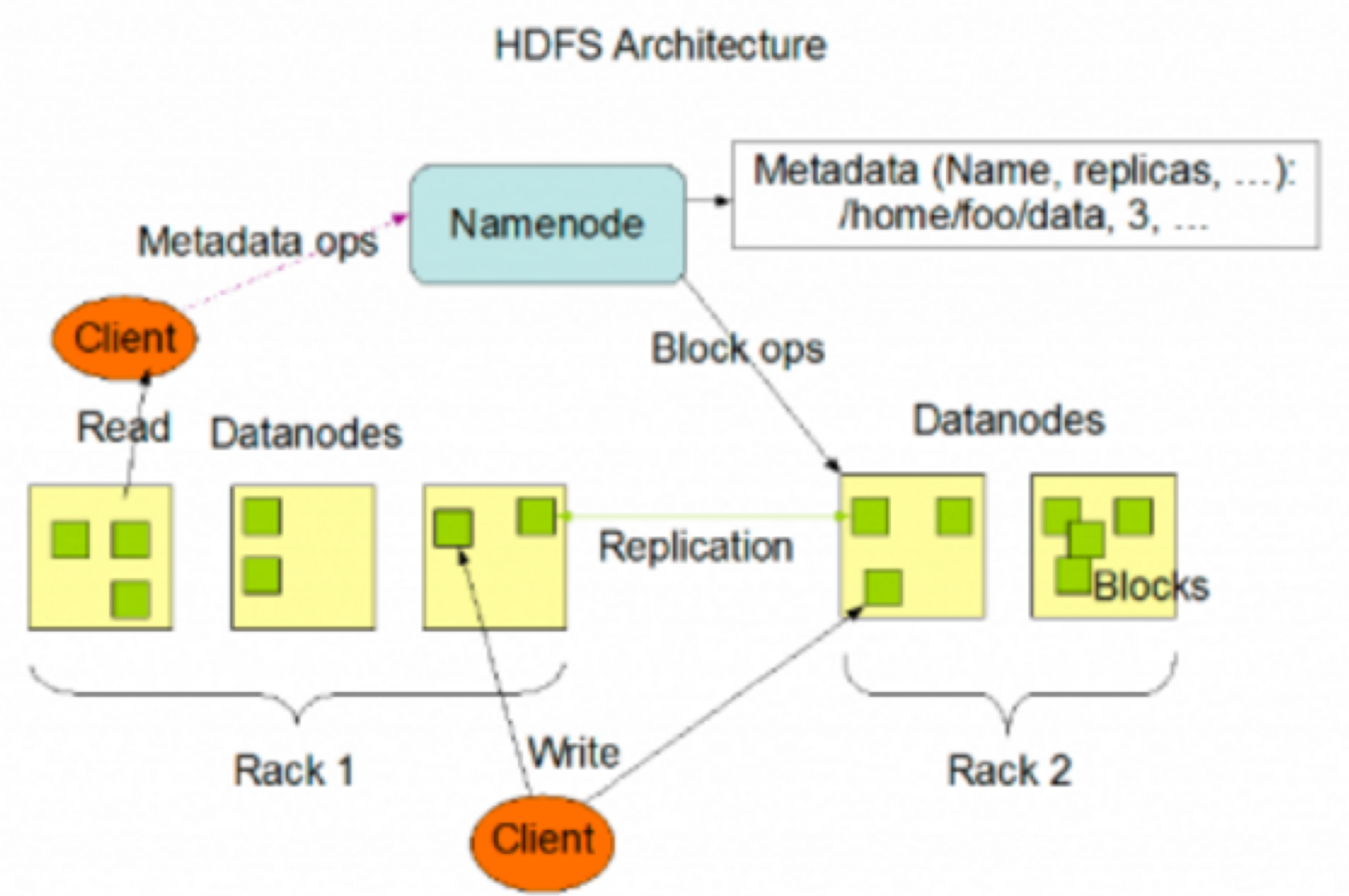
Note: This is not an exhaustive list

From: https://www.safaribooksonline.com/library/view/talend-for-big/9781782169499/ch01s02.html

Hadoop Architecture



HDFS


HDFS Architecture
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Client
Rack 2

Log statistics

```java
package com.cloudera.example;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class LogEventParseMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    enum Level { TRACE, DEBUG, INFO, WARN, ERROR, FATAL };
    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException,
            InterruptedException {

        String line = value.toString();

        // ignore empty lines
        if (line.trim().isEmpty()) {
            return;
        }
        String[] fields = line.split(" ");

        // ensure this line is not malformed
        if (fields.length <= 3) {
            return;
        }

        String levelField = fields[3];
        for (Level level : Level.values()) {
            String levelName = level.name();
            if (levelName.equalsIgnoreCase(levelField)) {
                context.write(new Text(levelName), new IntWritable(1));
            }
        }
    }
}
```

Log statistics

```java
package com.cloudera.example;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class LogEventSumReducer extends Reducer<Text, IntWritable,
        Text, IntWritable> {

    /*
     * The reduce method is invoked once for each key received from
     * the shuffle and sort phase of the MapReduce framework.
     * The method receives a key of type Text (representing the key),
     * a set of values of type IntWritable, and a Context object.
     */
    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context) throws IOException, InterruptedException {

        // used to count the number of messages for this event type
        int sum = 0;
        // increment it for each new value received
        for (IntWritable value : values) {
            sum += value.get();
        }

        // Our output is the event type (key) and the sum (value)
        context.write(key, new IntWritable(sum));
    }
}
```

Log statistics

```java
package com.cloudera.example;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Job;

// The driver is just a regular Java class with a "main" method
public class Driver {
    public static void main(String[] args) throws Exception {

        // validate commandline arguments (we require the user
        // to specify the HDFS paths to use for the job; see below)
        if (args.length != 2) {
            System.out.printf("Usage: Driver <input dir> <output dir>\n");
            System.exit(-1);
        }
        // Instantiate a Job object for our job's configuration.
        Job job = new Job();
        // configure input and output paths based on supplied arguments
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // specify the job's output key and value classes
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        // start the MapReduce job and wait for it to finish.
        // if it finishes successfully, return 0; otherwise 1.
        boolean success = job.waitForCompletion(true);
        System.exit(success ? 0 : 1);
    }
}
```

Disco

Python Centric system from Nokia

Eliminates many of the Java inconveniences from Hadoop

Reasonable mature

- But not nearly as common as Hadoop

Disco

```
from disco.core import Job, result_iterator

def map(line, params):
    for word in line.split():
        yield word, 1

def reduce(iter, params):
    from disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)

if __name__ == '__main__':
    job = Job().run(input=["http://discoproject.org/media/text/chekhov.txt"],
                    map=map,
                    reduce=reduce)
    for word, count in result_iterator(job.wait(show=True)):
        print(word, count)
```

BDAE/SOFA

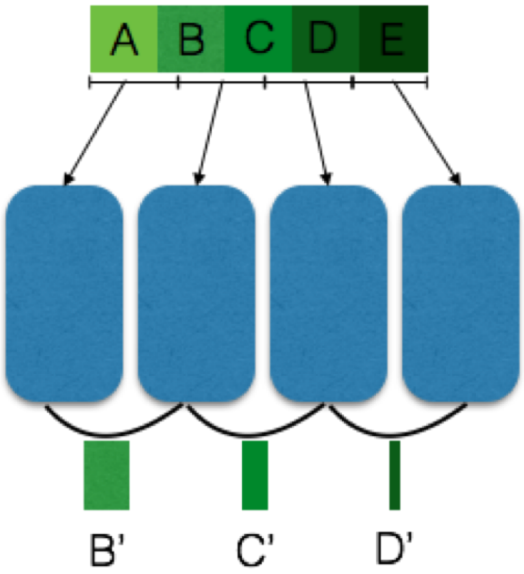
Research project from NBI

Meant to deal with very large datasets

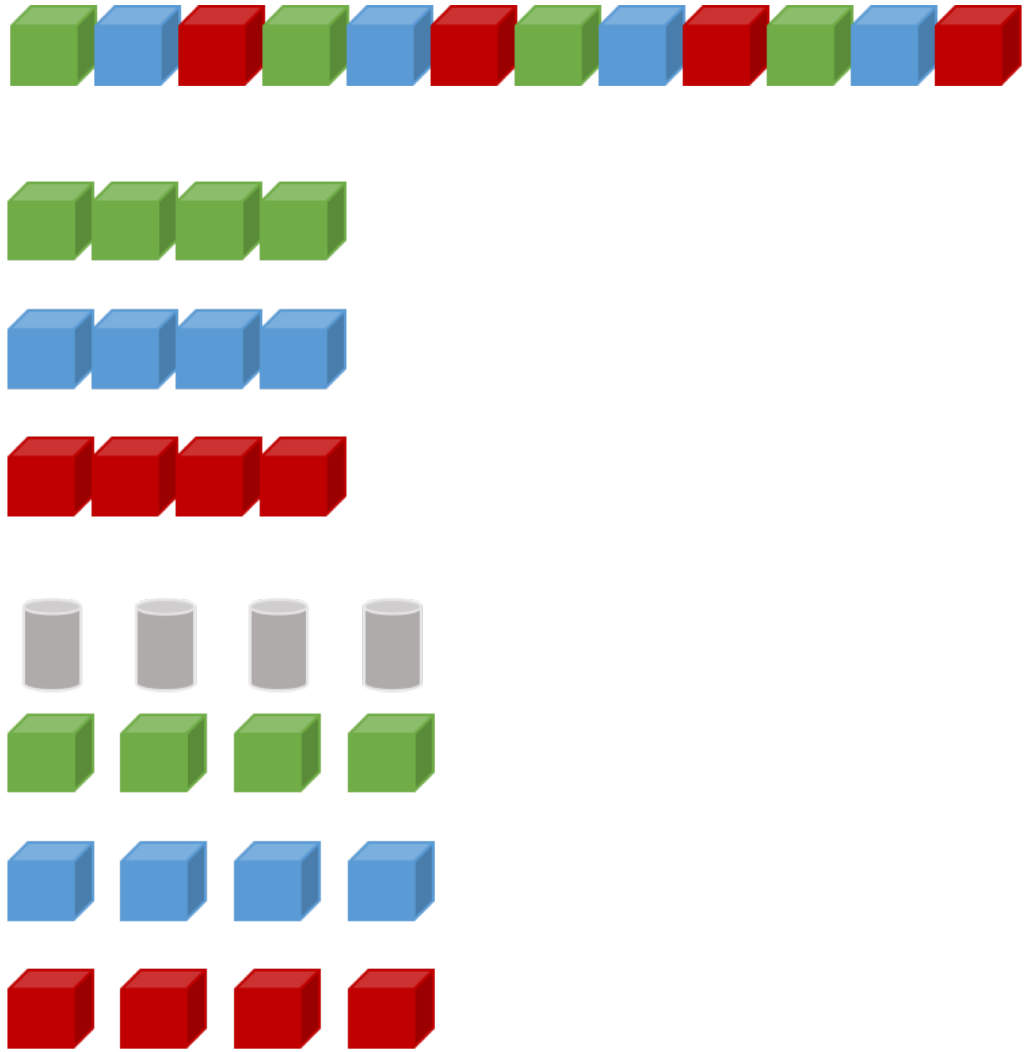
Based on a semantic understanding of the datasets

Fixes the residual problem that both Hadoop and Disco has

- Both previous examples would actually fail to produce the correct results for large enough inputs



BDAE



Log statistics

```
# Created by Steffen Karlsson on 06-13-2016
# Copyright (c) 2016 The Niels Bohr Institute at University of Copenhagen. All rights reserved.

from numpy import array, zeros

from bdae.templates.text_dataset import TextDataByLine
from sofa.foundation.base import load_data_by_path
from sofa.foundation.operation import OperationContext


class LogEventParserData(TextDataByLine):
    def preprocess(self, data_ref):
        return load_data_by_path(data_ref)

    def get_operations(self):
        return [OperationContext.by(self, 'count logs', '[log_mapper, log_reducer]')
                        .with_post_processing(post_processing)]

    def get_map_functions(self):
        return [log_mapper]

    def get_reduce_functions(self):
        return [log_reducer]
```

Log statistics

```
def log_mapper(blocks, levels):
    res = zeros((1, len(levels)))
    for entry in sum(blocks, []):
        res += array([entry.count(level) for level in levels])

    return res

def log_reducer(blocks):
    return array(blocks).sum(axis=0)
```

Error handling

All BDA systems support redundancy

Most also support automatic resubmission of failed jobs

- So most systems are very robust

Infrastructure

Three basic approaches

- Dirt cheap
- Scalability focused
- Fat nodes
- HPC

Dirt cheap

CPUs plus disk in large scale

Little disk per processor

Very high failure rate

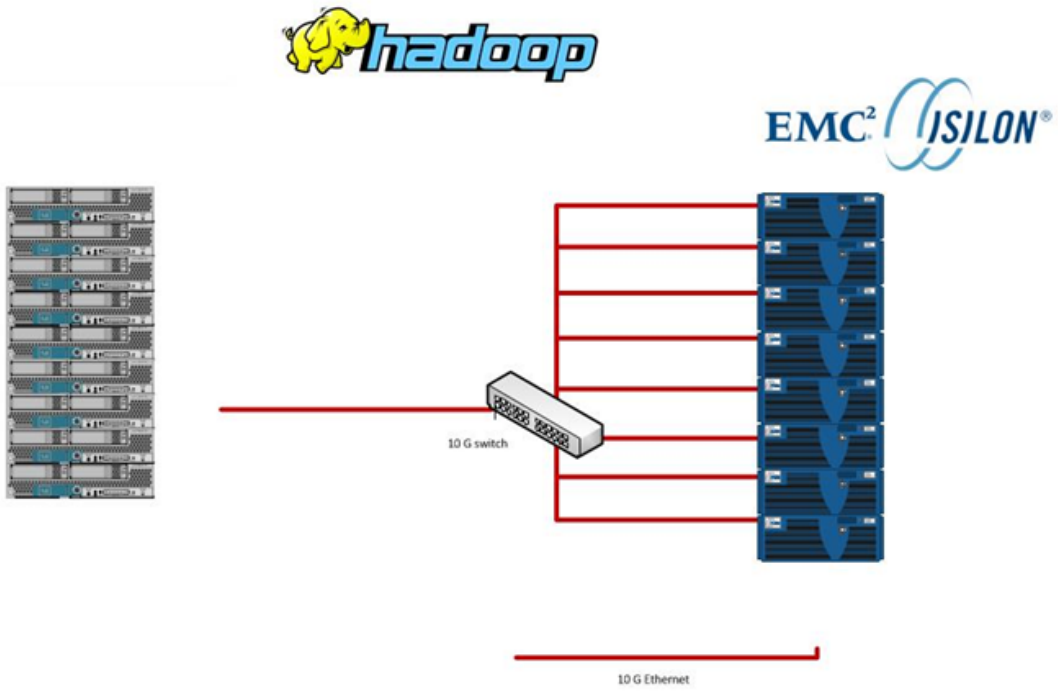
First google setup

Scalability focused

Scale IO or Processors as needed

Not suited for optimal IO utilization

Adored by many system-administrators



From: http://hsk-hwx.readthedocs.io/en/latest/hsk.html

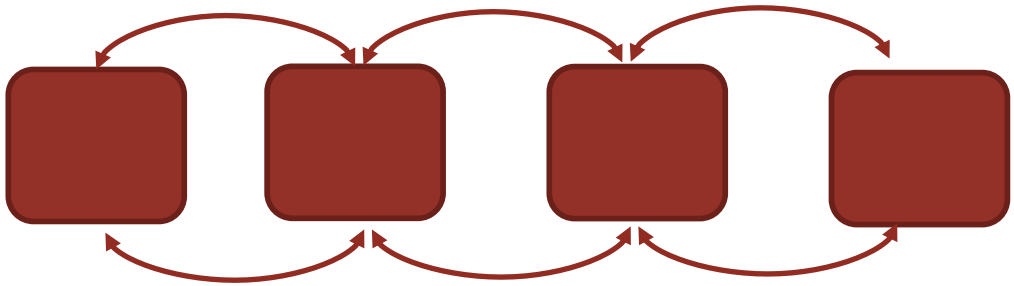
Fat nodes

Similar to dirt cheap

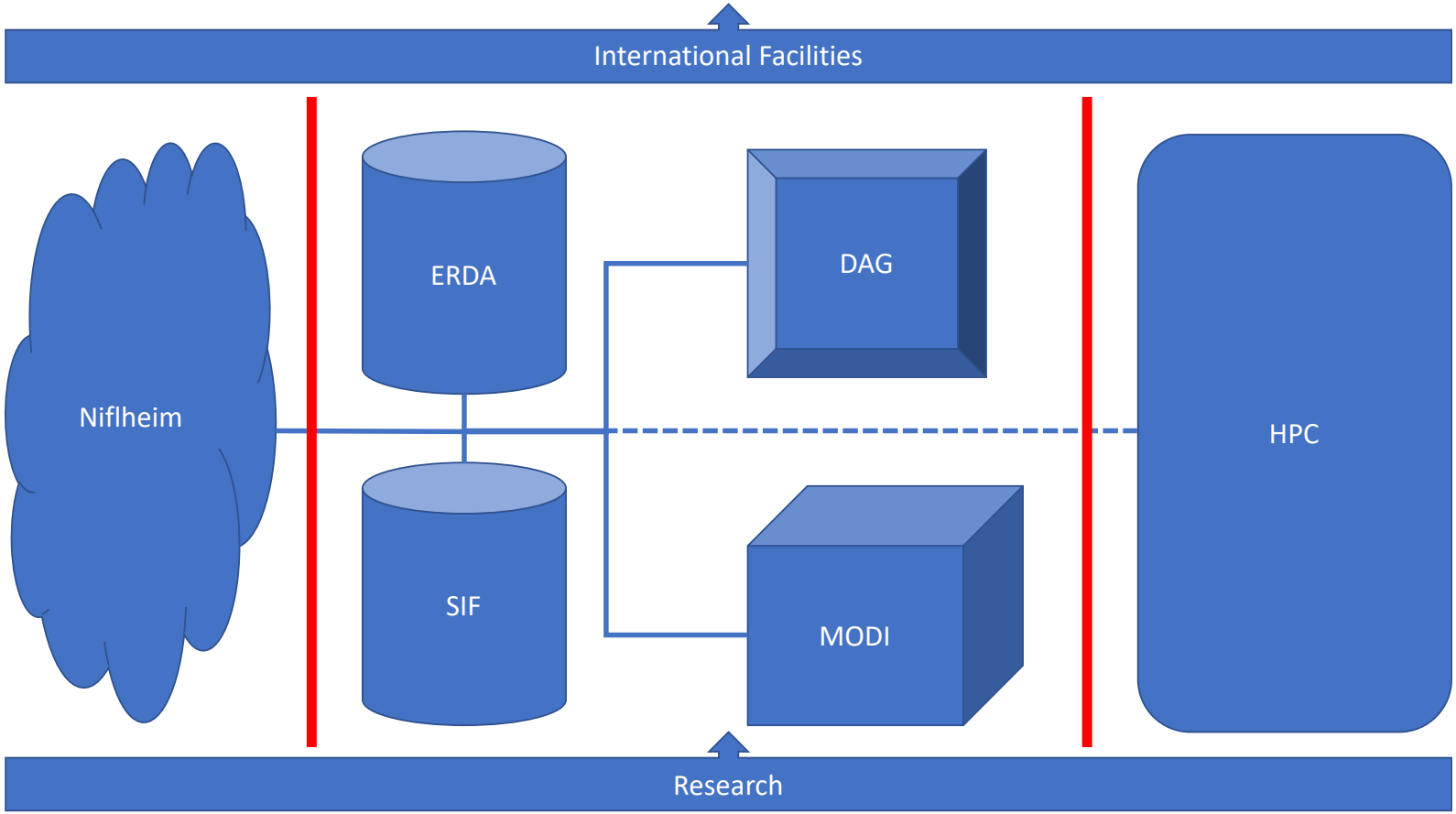
Much larger disk systems per node

Disk redundancy

Server redundancy



UCPH/SCIENCE



Quiz 2D

If we have large scientific datasets and need to apply machine learning on those; what would we like the infrastructure to do for us?

Motivation for more speed

Target

Challenge

Knots

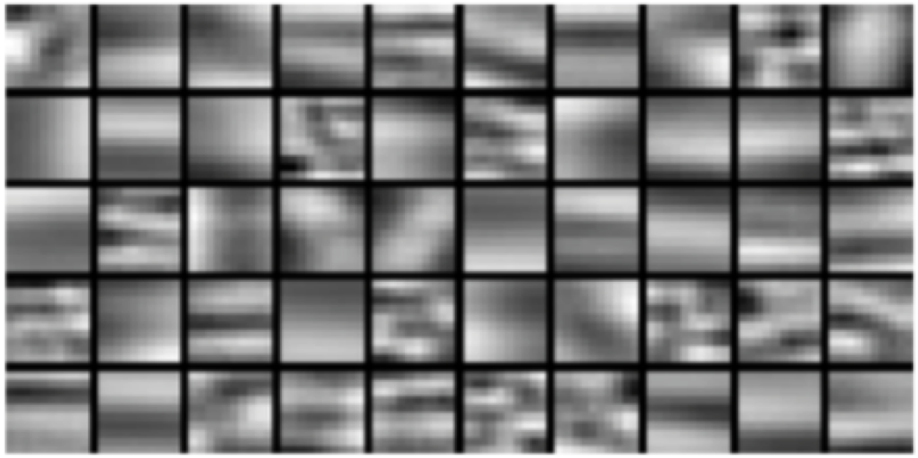
Equipment

Mechanics

Mechanics

DISCRIMINATIVE DICTIONARY LEARNING

Using training data we build a ”dictionary” or building blocks.

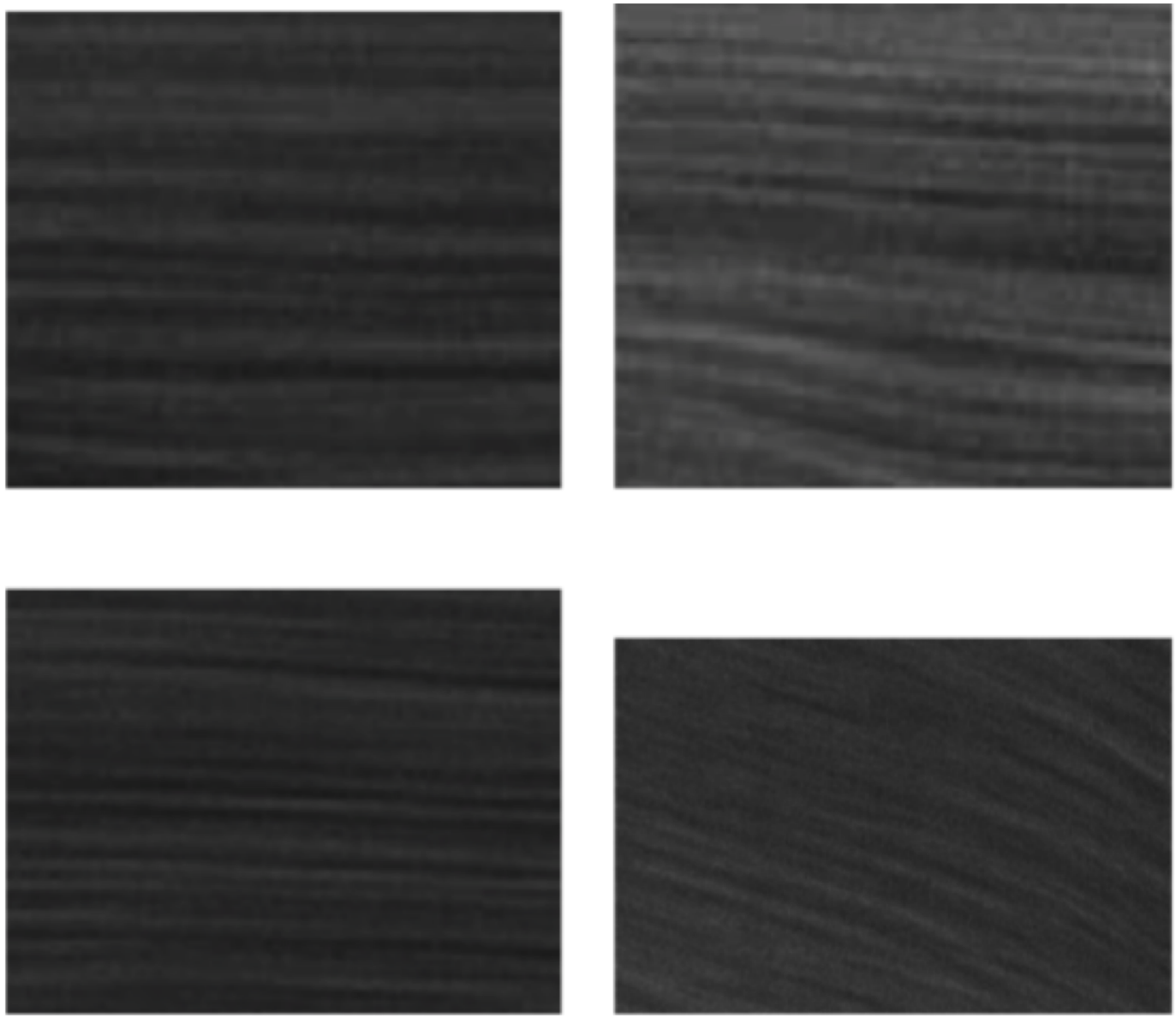


The building blocks can then be used to represent other images efficiently or *sparsely*.

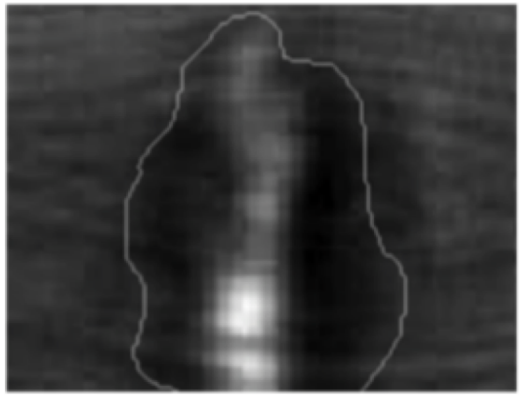
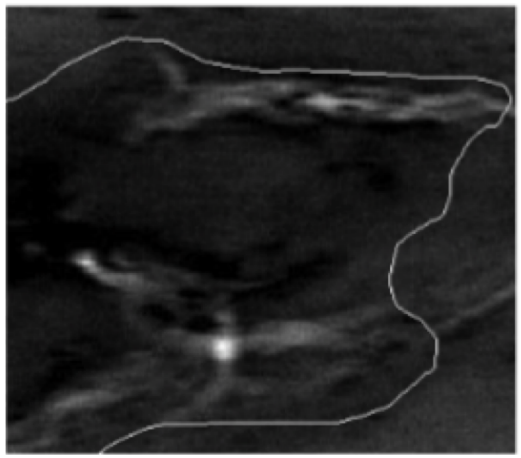
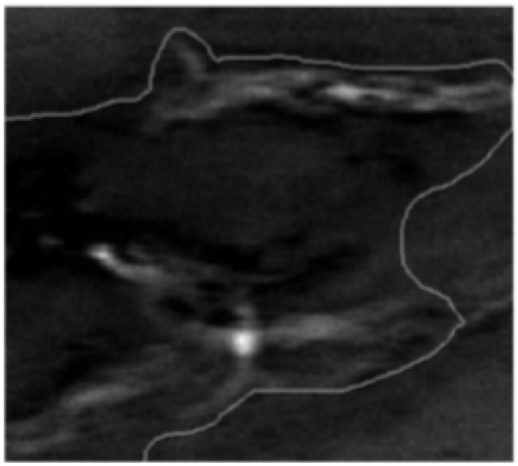
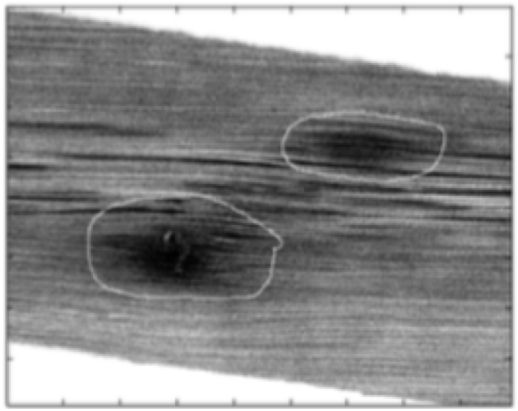
The idea is that if we have built our dictionary from knots, the knotty images will be better represented by the dictionary.

If we build so called discriminative dictionaries, then these are good at representing their target class but bad at all others.

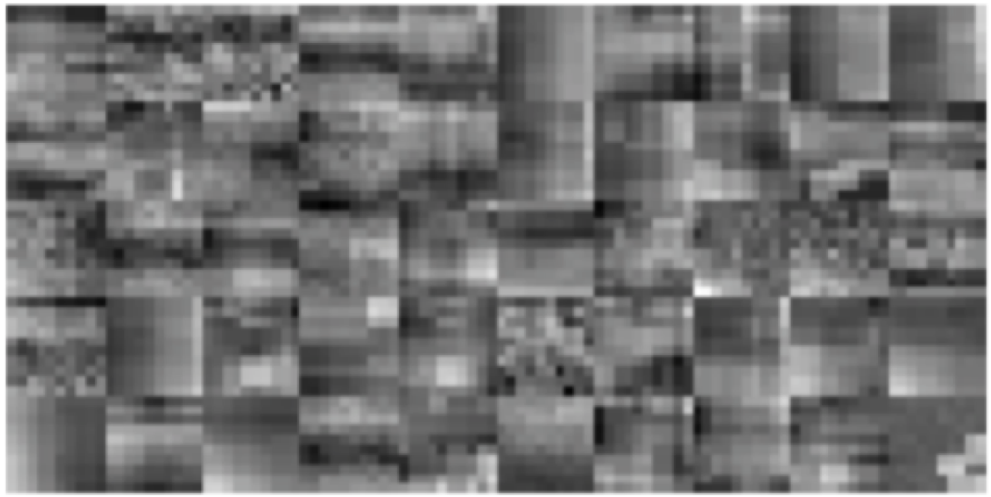
TRAINING IMAGES WITH NO KNOTS



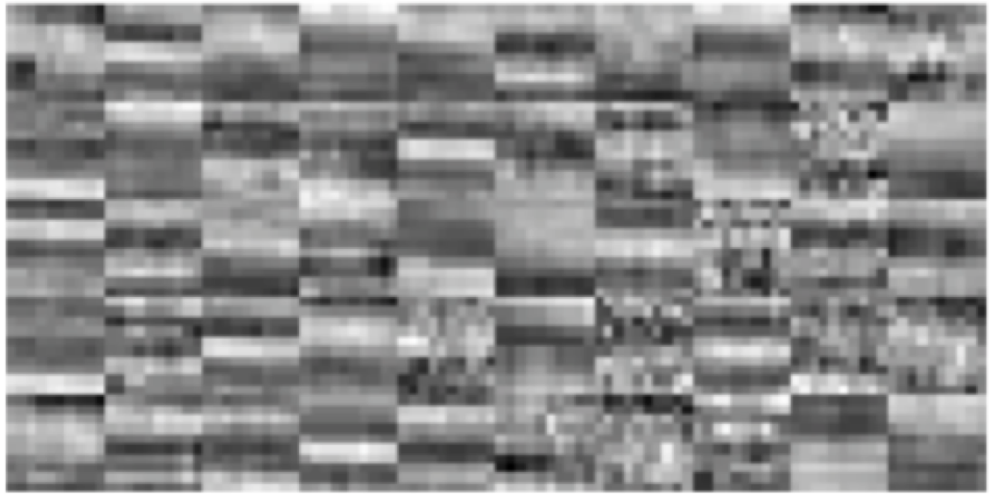
TRAINING IMAGES WITH KNOTS

DISCRIMINATIVE DICTIONARIES

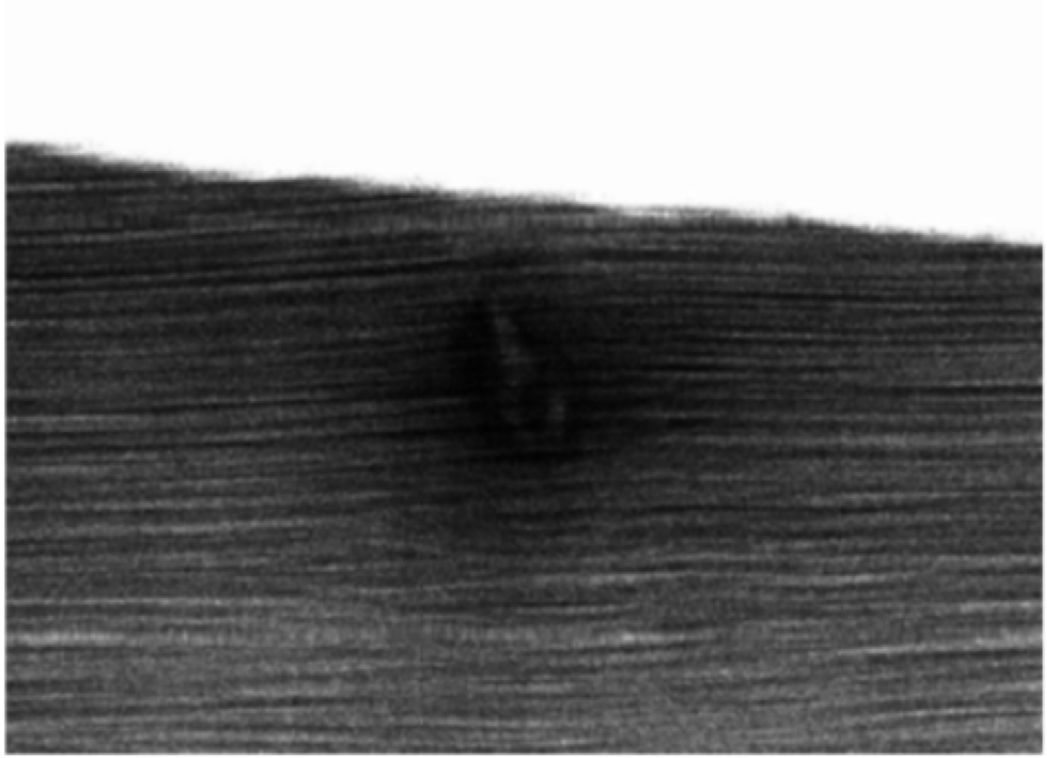


(a) Knot

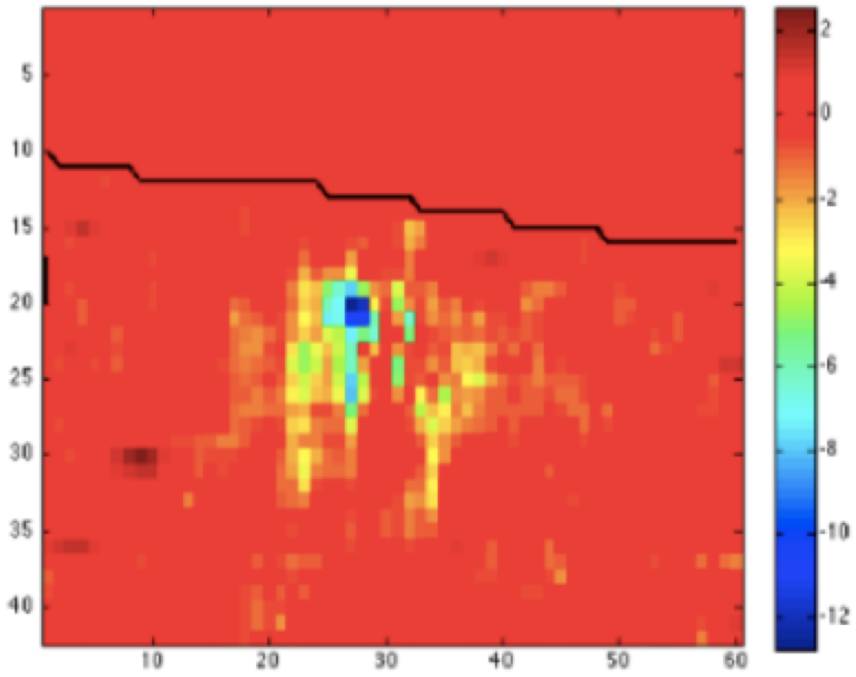


(b) No Knot

Example

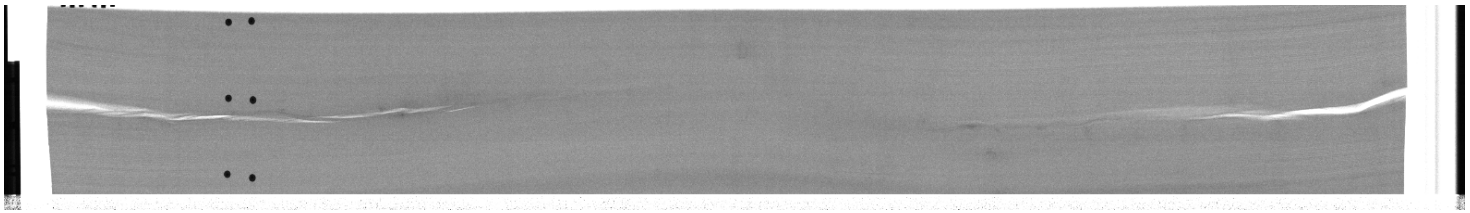


(c) Original image



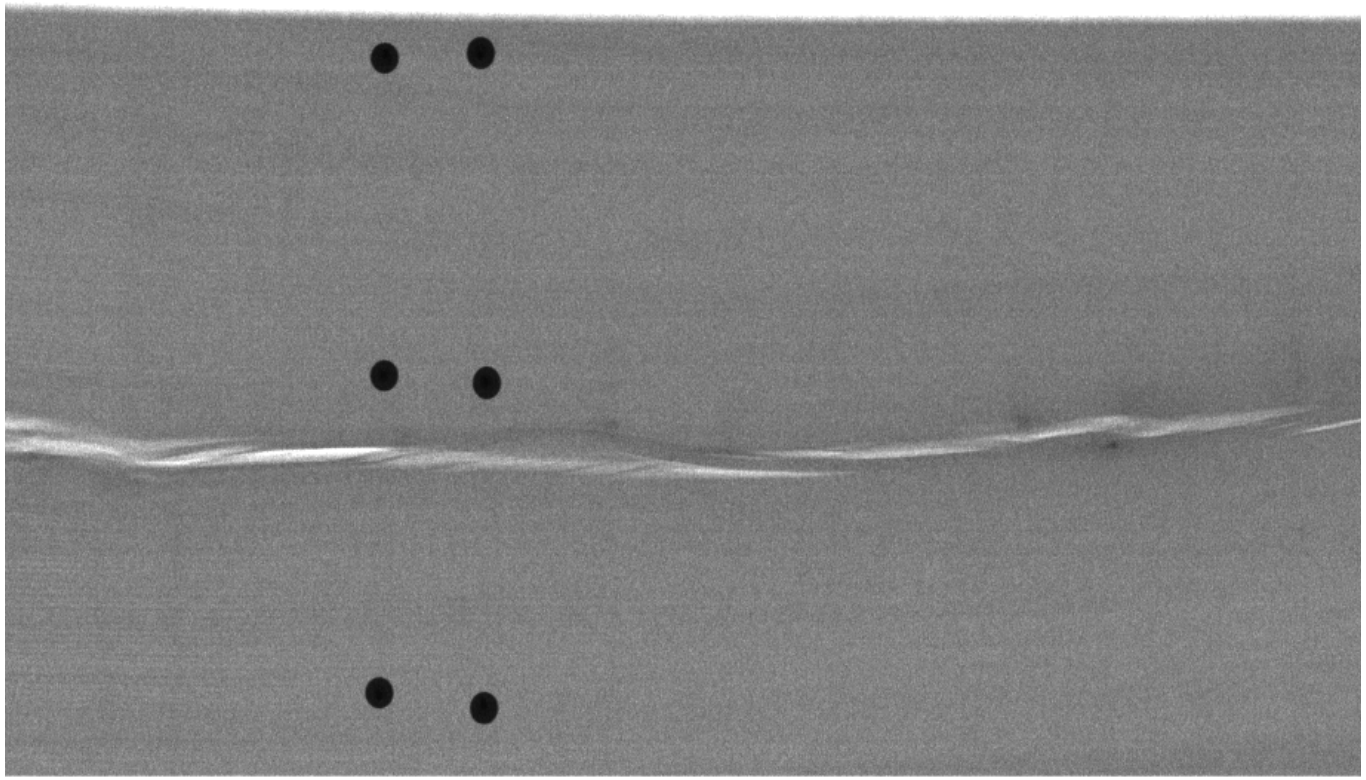
(d) Discriminative feature map

Segmented images

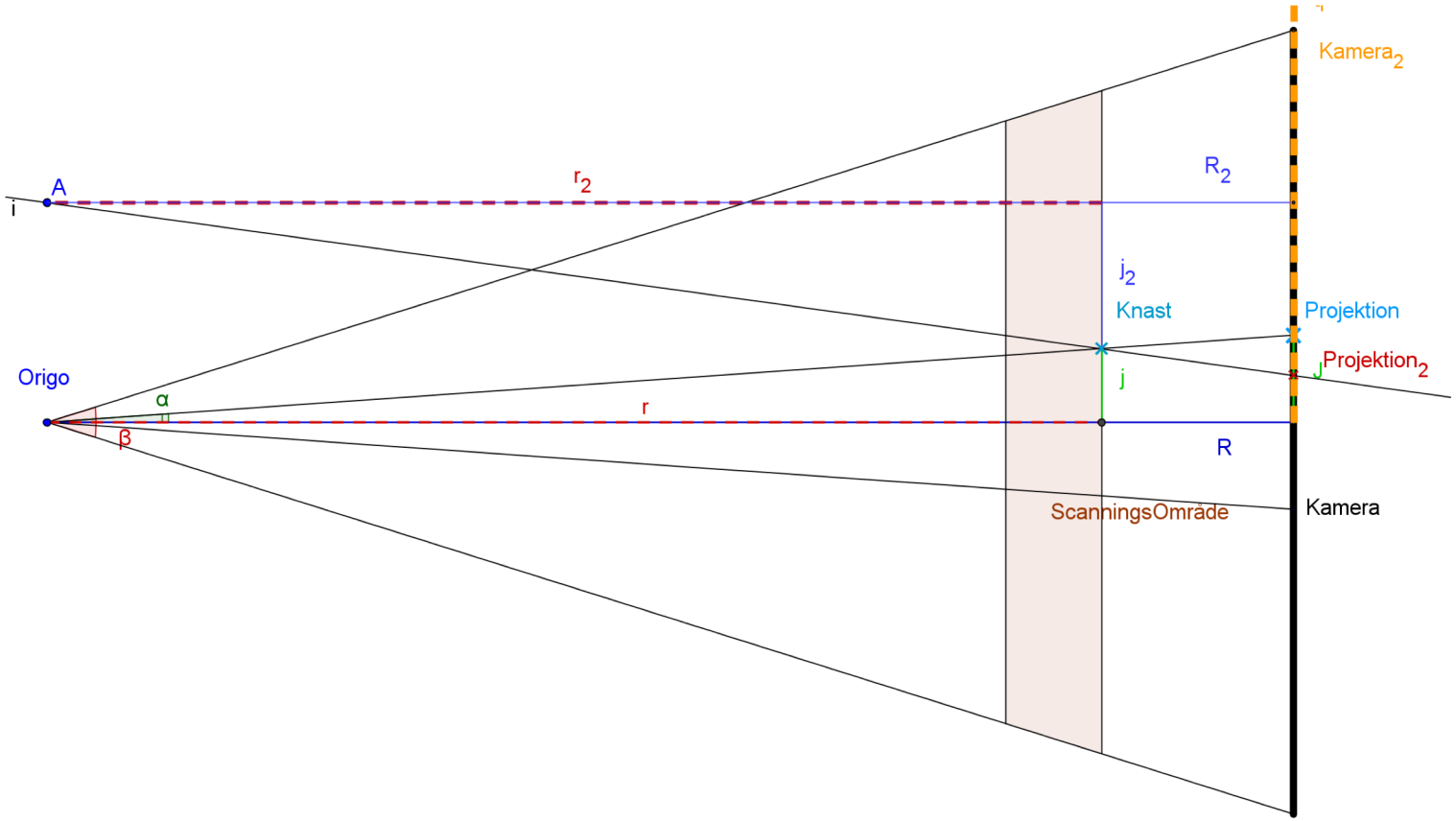


$$\begin{bmatrix} False & False & \dots & False & True \\ False & False & \dots & False & False \\ \vdots & \vdots & & \vdots & \vdots \\ False & False & \dots & False & False \\ False & False & \dots & False & False \end{bmatrix}$$

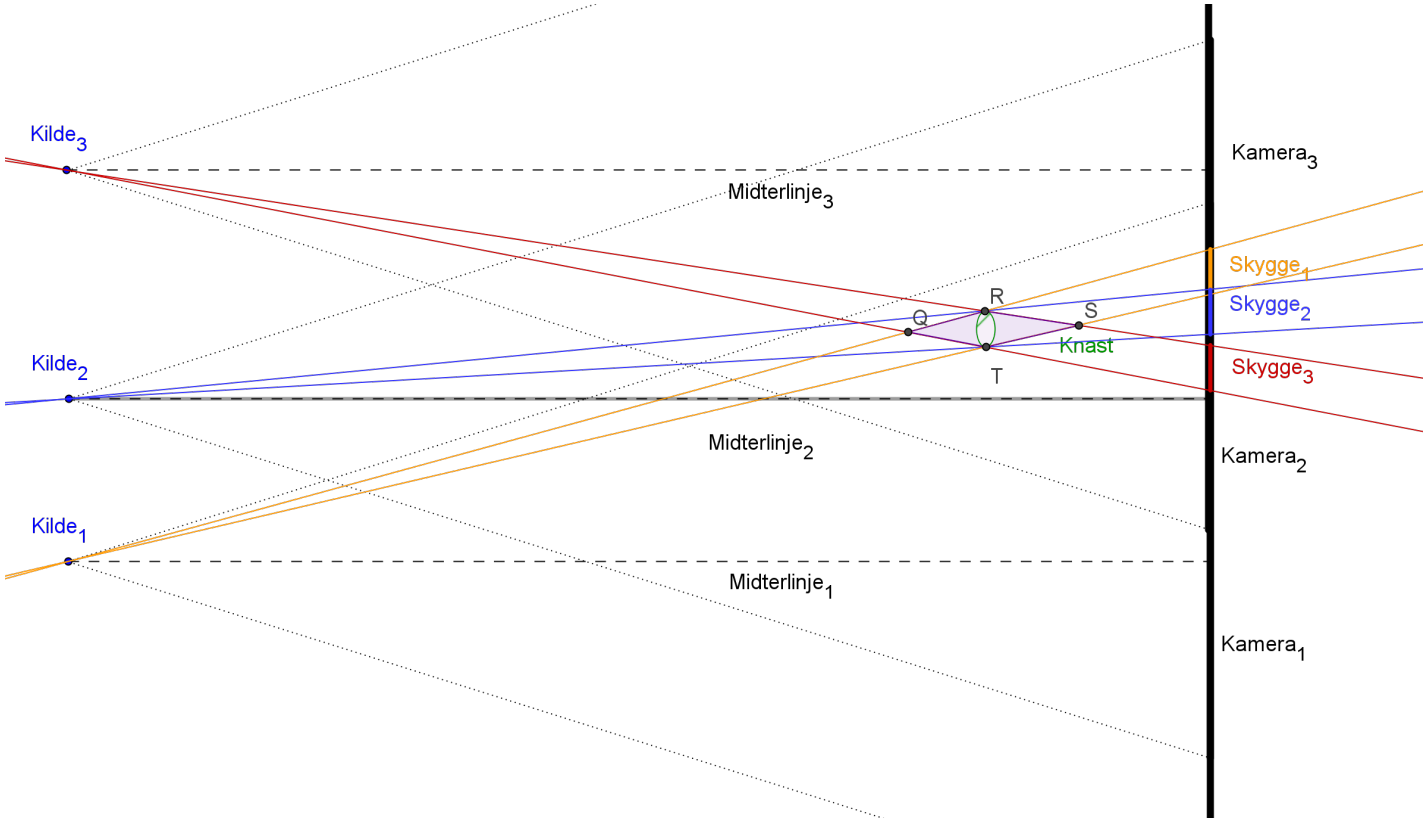
3D information



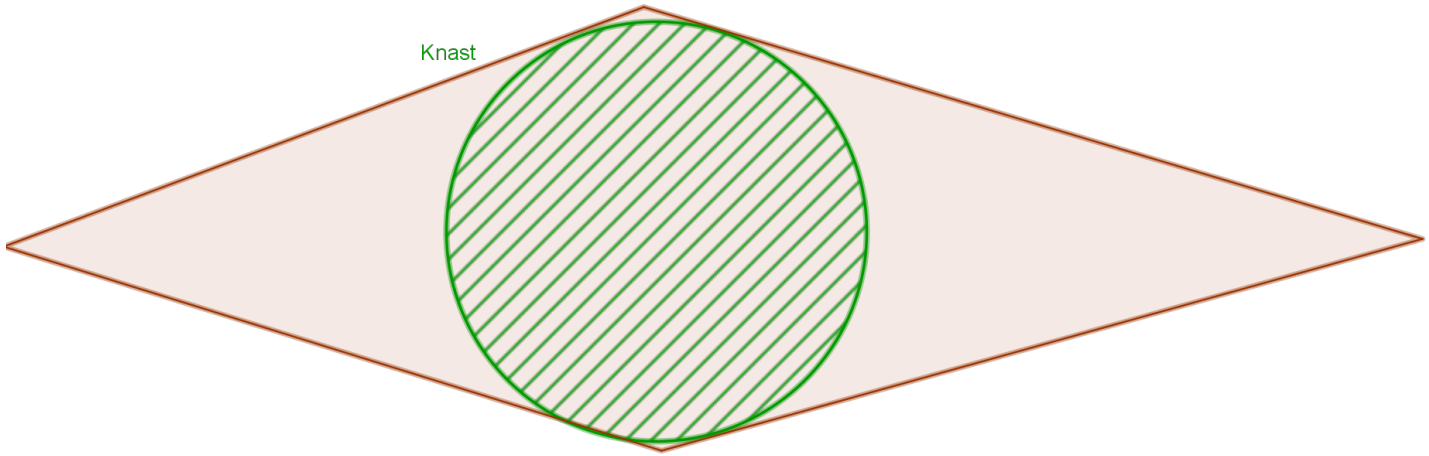
Approach


Kamera2
A
r2
R2
j2
Knast
Projektion
Origo
α
β
r
j
Projektion2
R
ScanningsOmråde
Kamera

Mapping


Kilde3
Midterlinje3
Kamera3
Skygge1
Skygge2
Skygge3
R
Q
S
Knast
T
Kilde2
Midterlinje2
Kamera2
Kilde1
Midterlinje1
Kamera1

Convex Hull



Solution Planning

$$12^{(10*20)}$$

685881690392905117434431489495385586127448264303076182128024256632771944880513797413380560779272522395272914057841494739399255775297713137770656031977268970189240419718422549044382776867564731096274305671116280037376

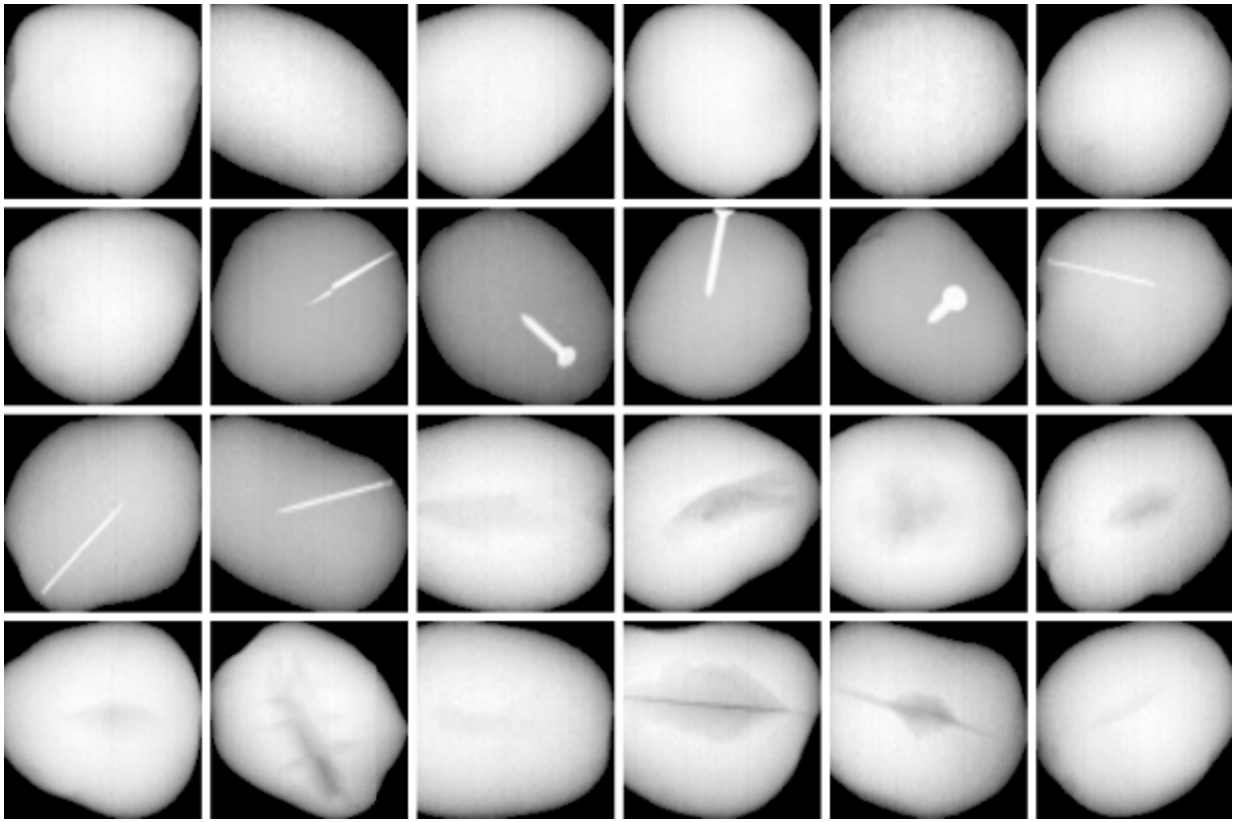
Solution Planning

$$12^{(10*20)}$$

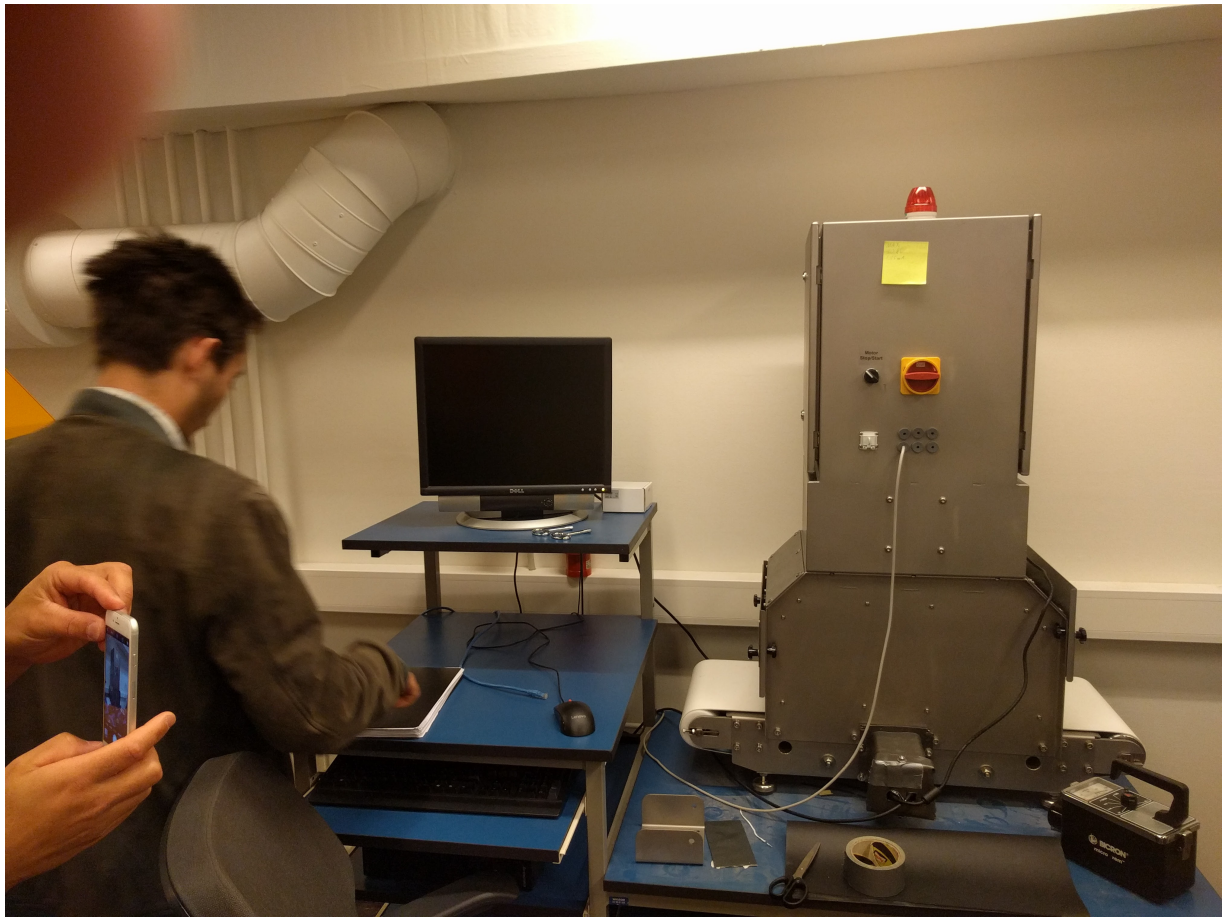
685881690392905117434431489495385586127448264303076182128024256632771944880513797413380560779272522395272914057841494739399255775297713137770656031977268970189240419718422549044382776867564731096274305671116280037376

But done in less than 10 sec

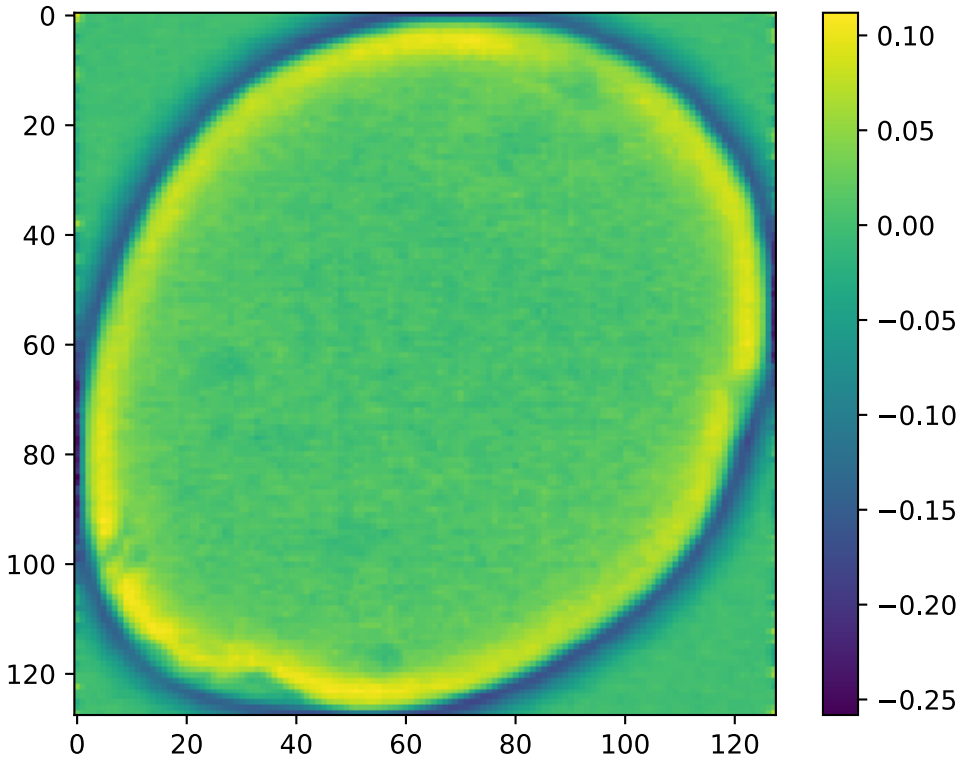
Potatoes



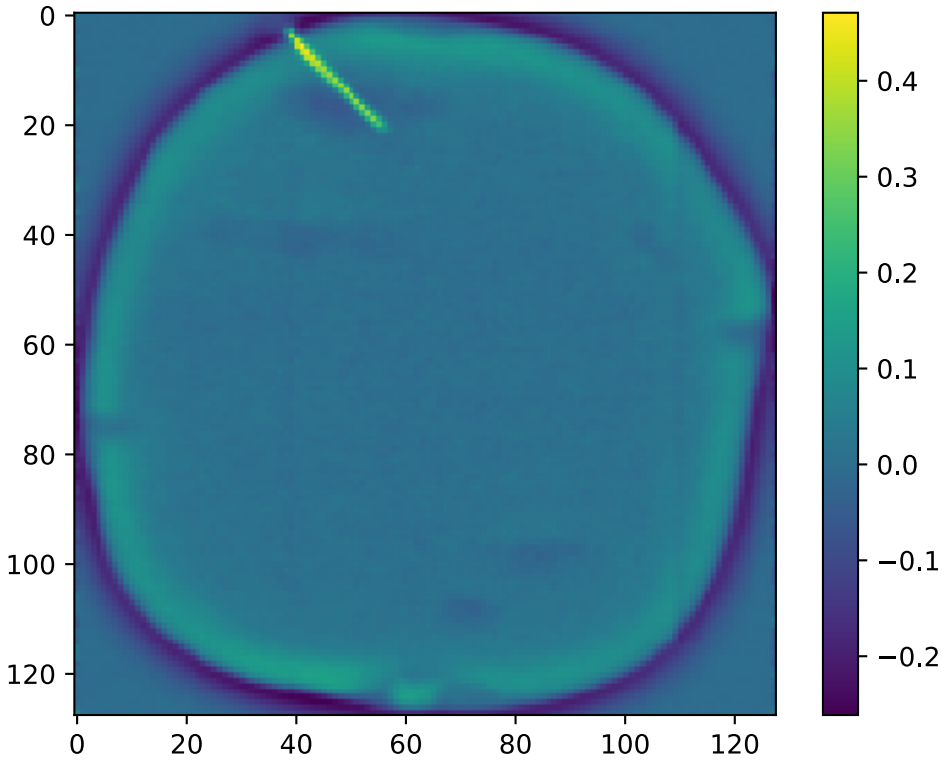
Cheap machine



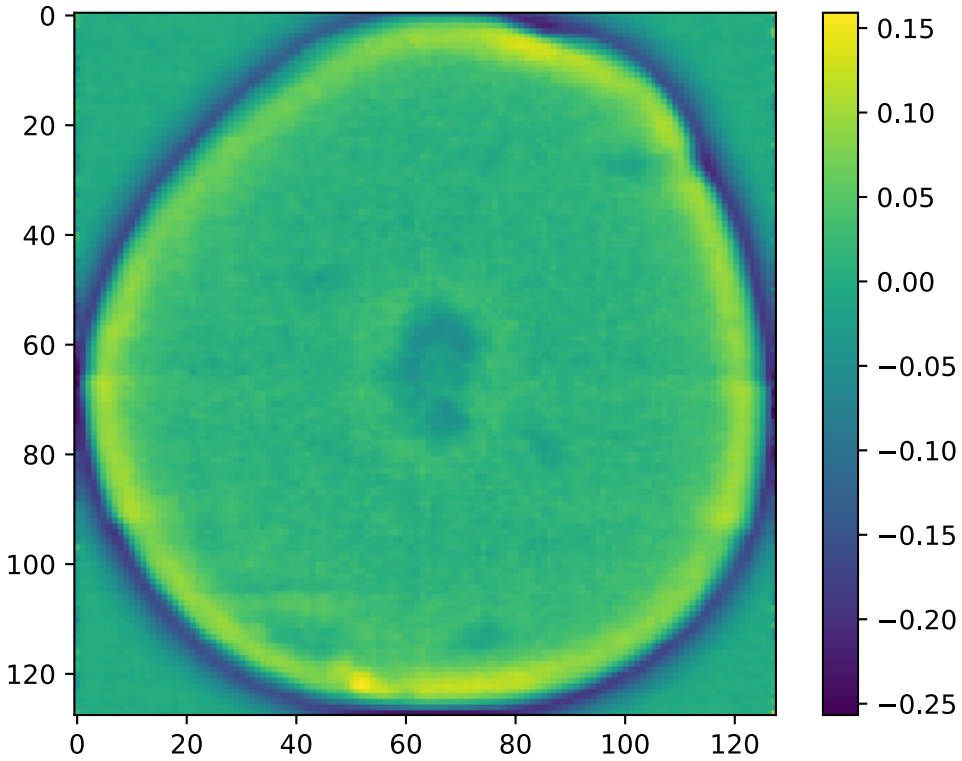
Perfect potato



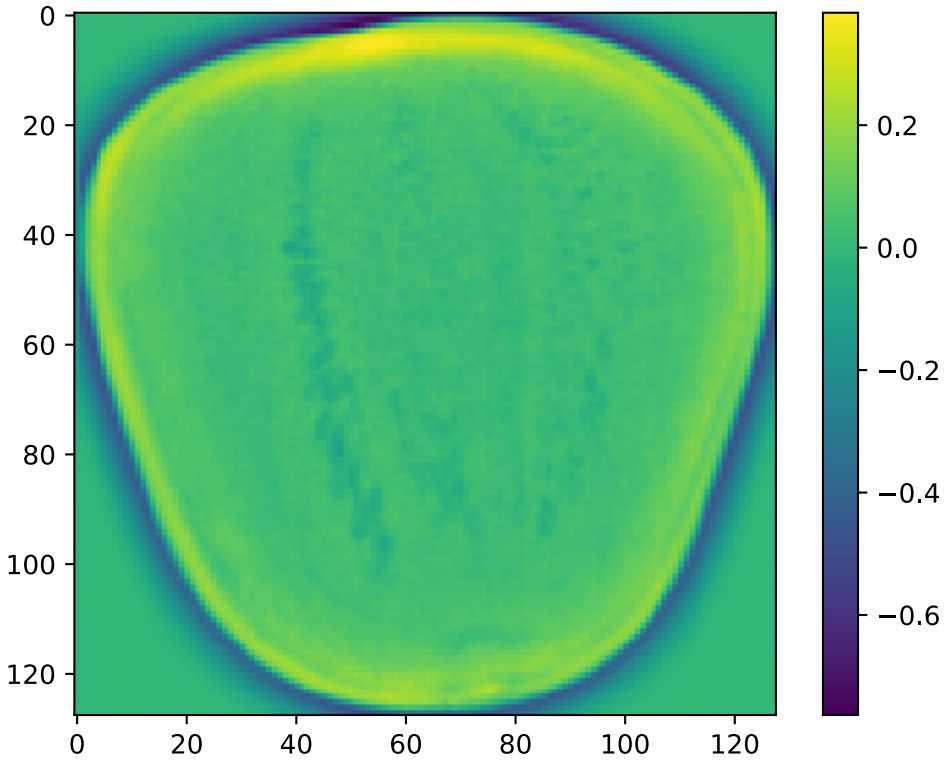
Needle



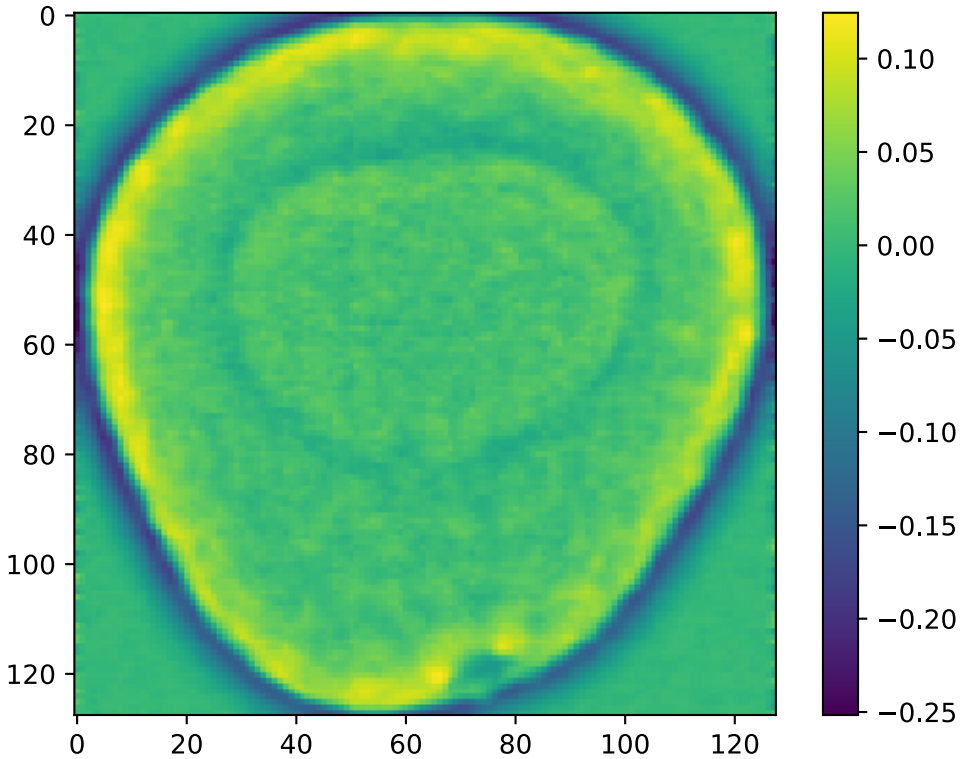
Hollow heart



Seedless melon



Avocado


0
20
40
60
80
100
120
0
20
40
60
80
100
120
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−0.25

Future Hardware

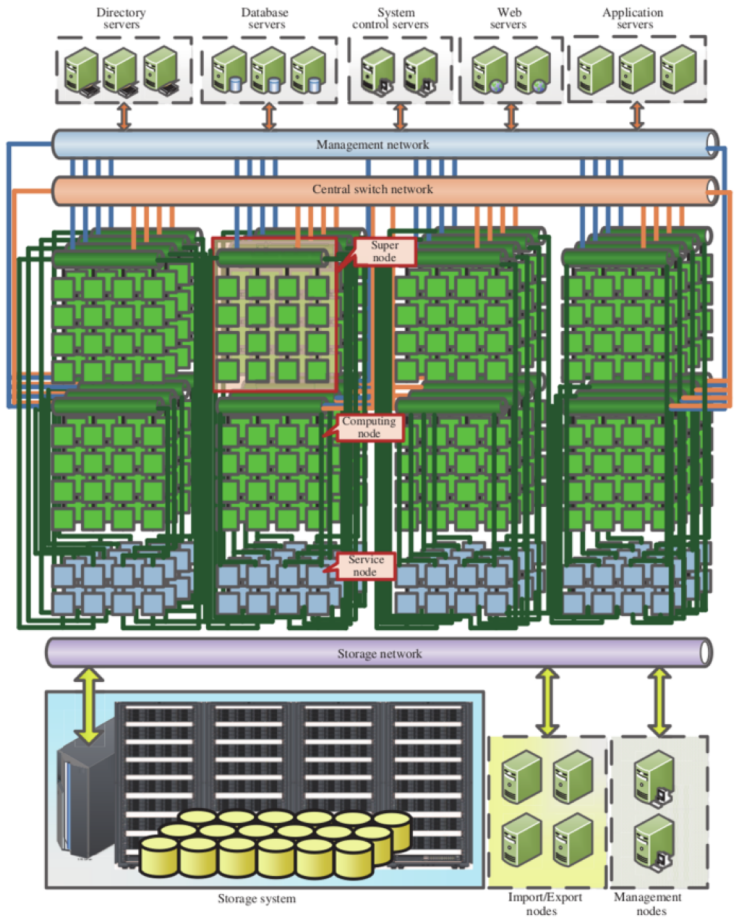
HPC

Still more groups favor HPC for big data

Has very high bandwidth and very scalable IO

Has very high and very scalable compute

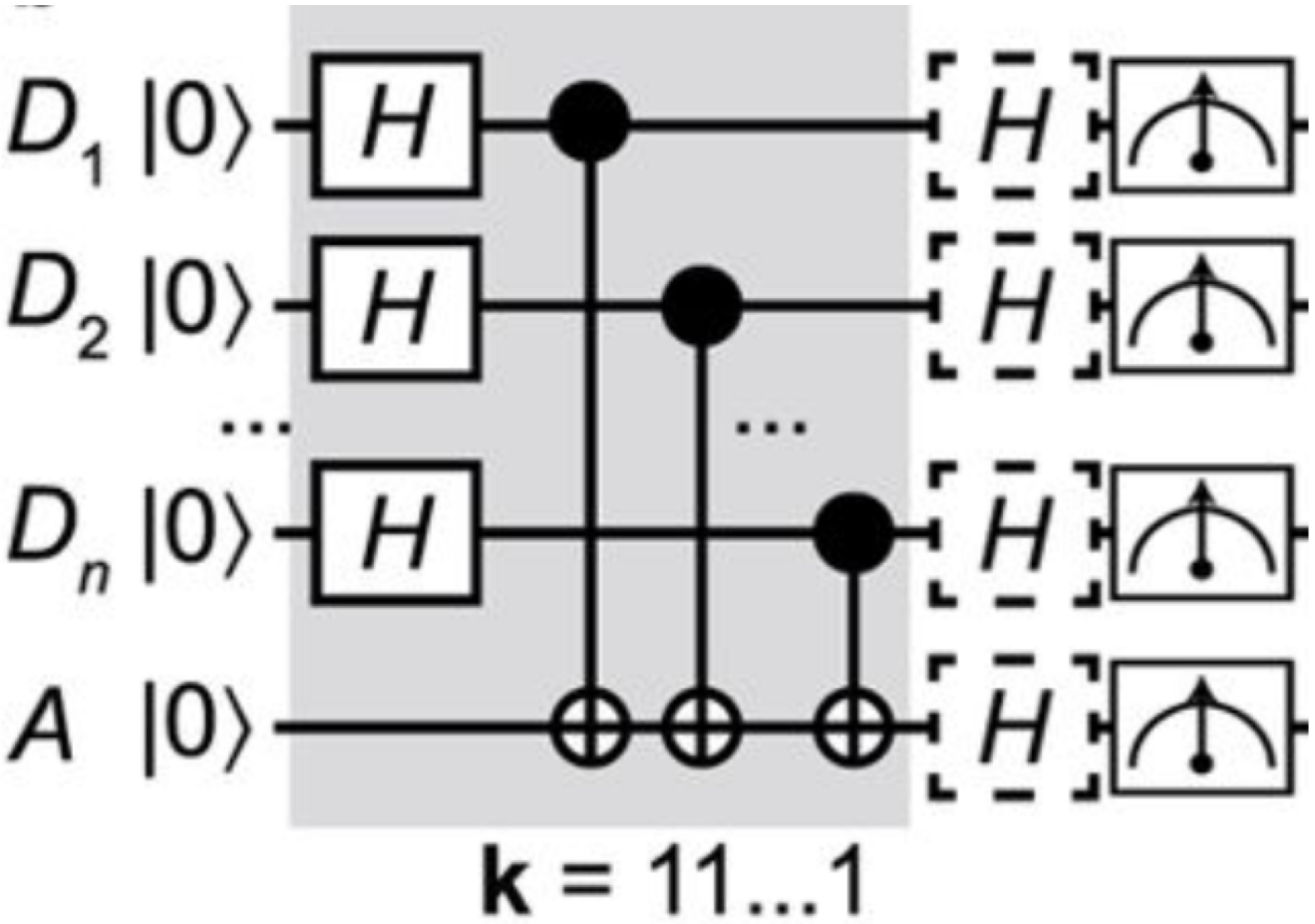
Not the cheapest solution



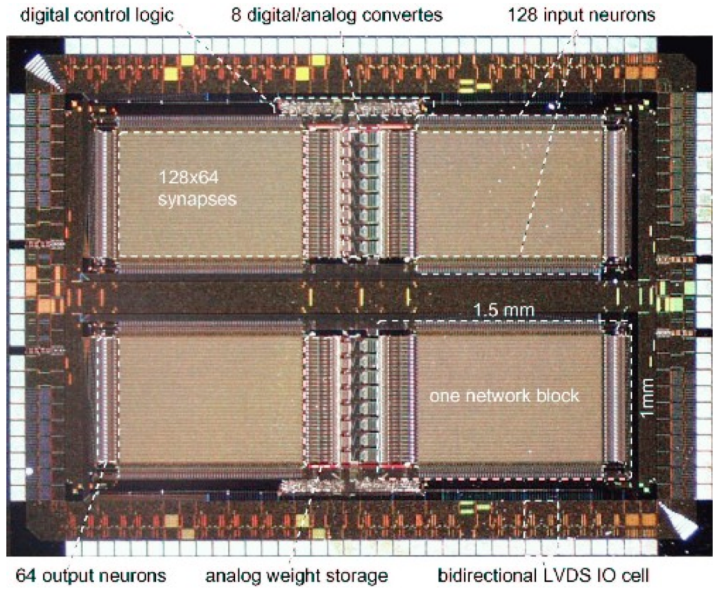
Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband

Site:	DOE/SC/Oak Ridge National Laboratory
System URL:	http://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/
Manufacturer:	IBM
Cores:	2,282,544
Memory:	2,801,664 GB
Processor:	IBM POWER9 22C 3.07GHz
Interconnect:	Dual-rail Mellanox EDR Infiniband
Performance	
Linpack Performance (Rmax)	122,300 TFlop/s
Theoretical Peak (Rpeak)	187,659 TFlop/s
Nmax	13,989,888
HPCG [TFlop/s]	2,925.75
Power Consumption	
Power:	8,805.50 kW (Submitted)
Power Measurement Level:	3
Measured Cores:	2,282,544

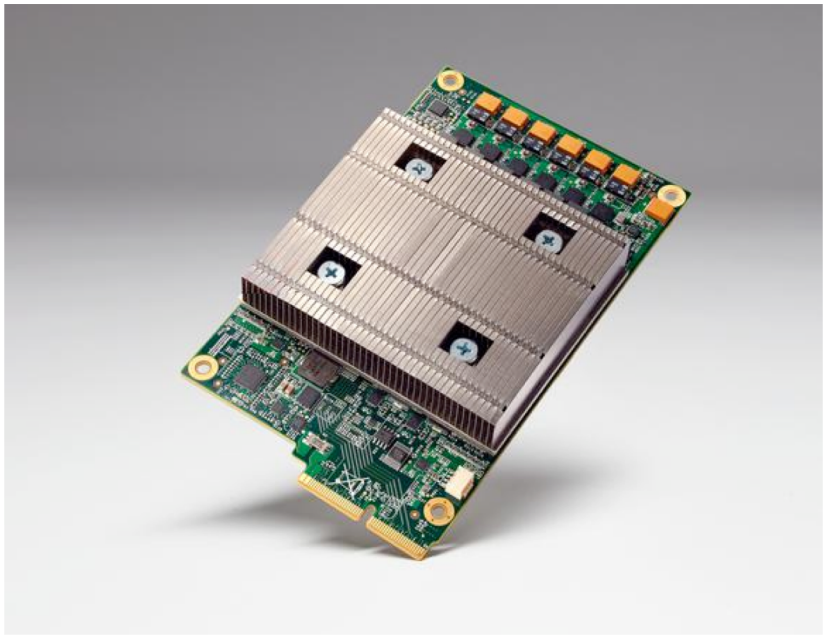
Quantum models



Analog Neural Networks



Google Tensor Flow Unit




AH, HE FINALLY
STFU
imgflip.com