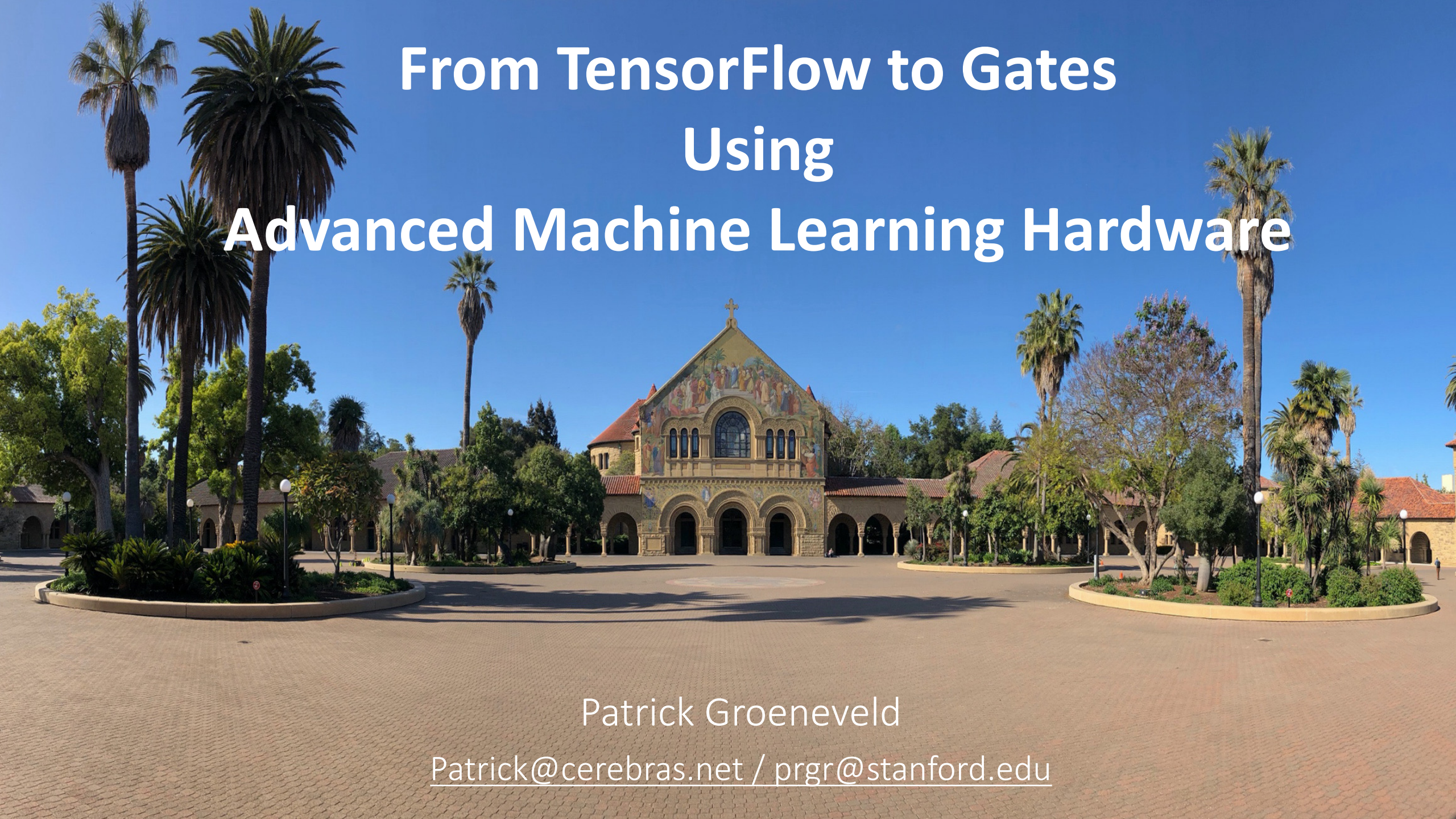


From TensorFlow to Gates Using Advanced Machine Learning Hardware

Patrick Groeneveld

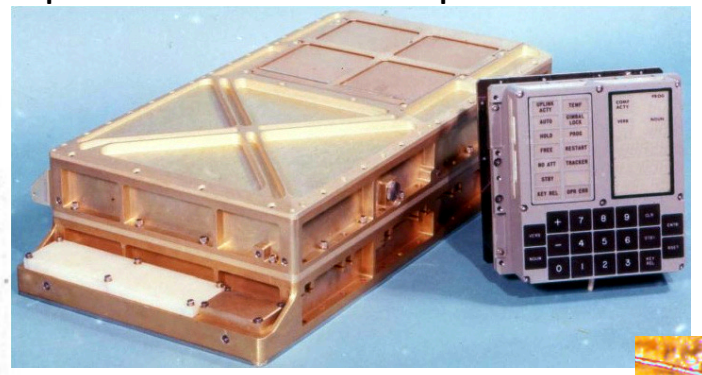
Patrick@cerebras.net / prgr@stanford.edu

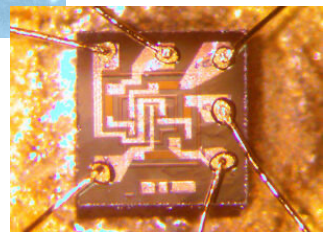
Outline: TensorFlow to Machine Learning Hardware

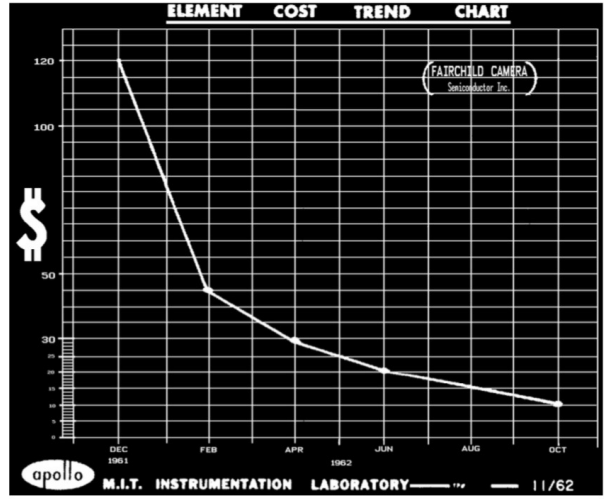
- Big picture trends in Silicon
- Computation for Machine Learning
  - Dense (Fully Connected) and convolution layers
- Cerebras: the world's largest chip
- Layer-Pipelined execution of ML
- TensorFlow to hardware flow in more detail

The Trends Leading to Moore's Law

Apollo Guidance Computer







This view graph portrays the IC cost reduction realized during the evaluation procurements.

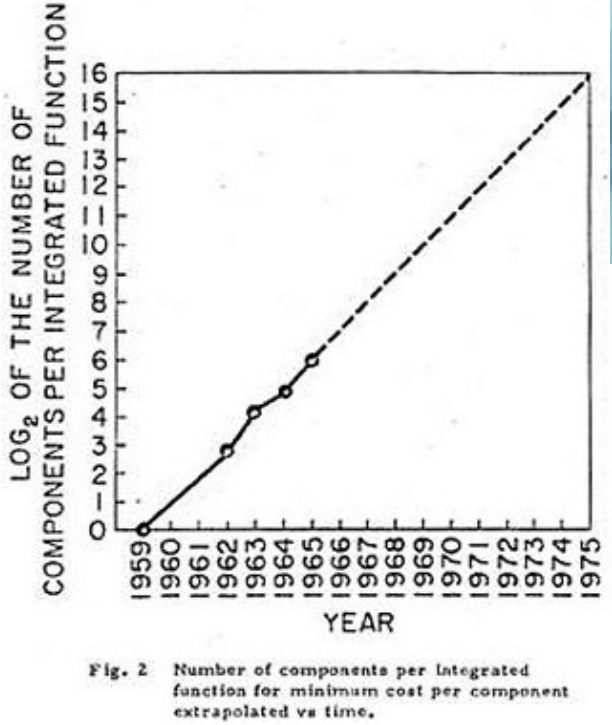
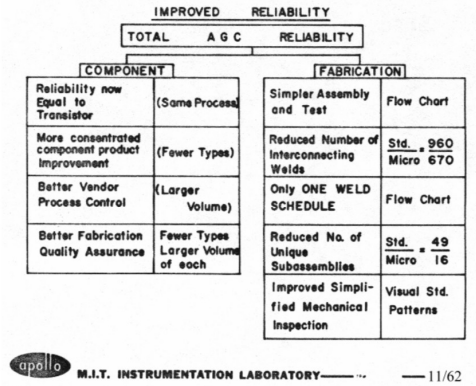
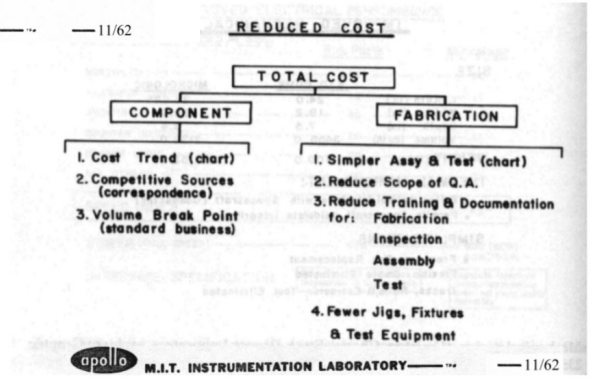
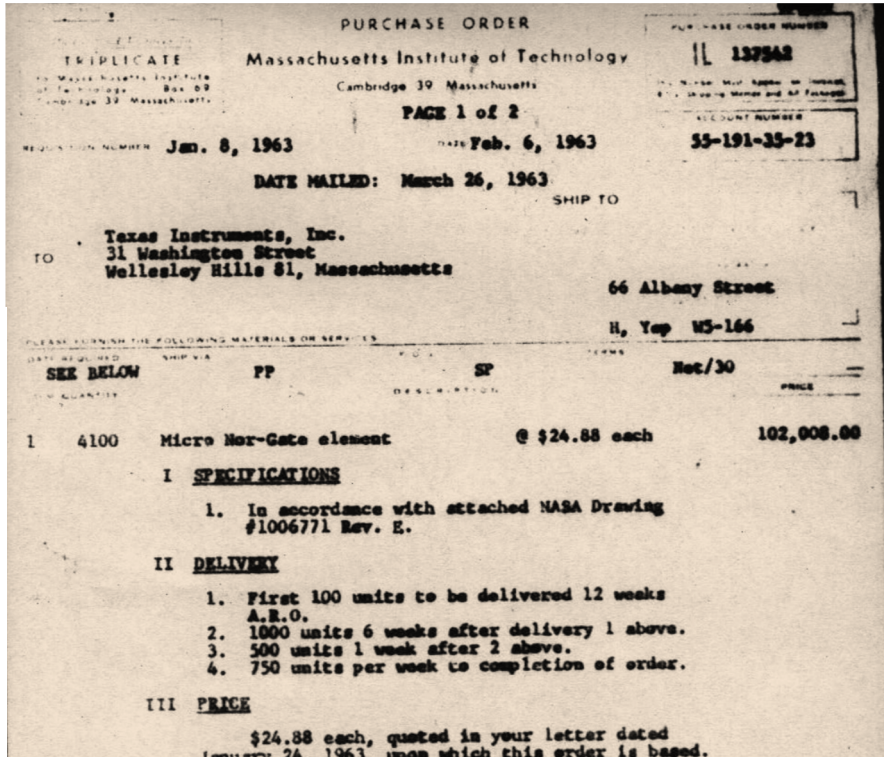


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.







PURCHASE ORDER

TRIPLICATE

Massachusetts Institute of Technology

Cambridge 39 Massachusetts

PAGE 1 of 2

IL 137542

Jan. 8, 1963 Feb. 6, 1963 55-191-35-23

DATE MAILED: March 26, 1963

SHIP TO

TO Texas Instruments, Inc.
31 Washington Street
Wellesley Hills 81, Massachusetts

66 Albany Street

H, Yep M5-166

SEE BELOW		PP	SP	Net/30
1	4100	Micro Nor-Gate element	@ $24.88 each	102,008.00

I SPECIFICATIONS

1. In accordance with attached NASA Drawing #1006771 Rev. E.

II DELIVERY

1. First 100 units to be delivered 12 weeks A.R.O.
2. 1000 units 6 weeks after delivery 1 above.
3. 500 units 1 week after 2 above.
4. 750 units per week to completion of order.

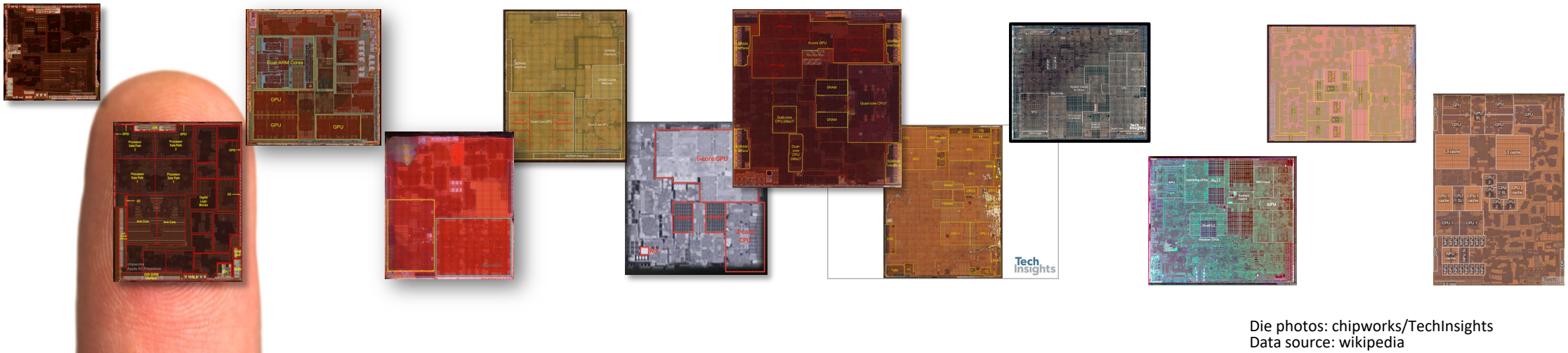
III PRICE

$24.88 each, quoted in your letter dated

Source: Gordon E. Moore,
Cramming More Components onto Integrated Circuits,
Electronics, pp. 114–117, April 19, 1965.

12 Generations of Apple Mobile System-on-Chips

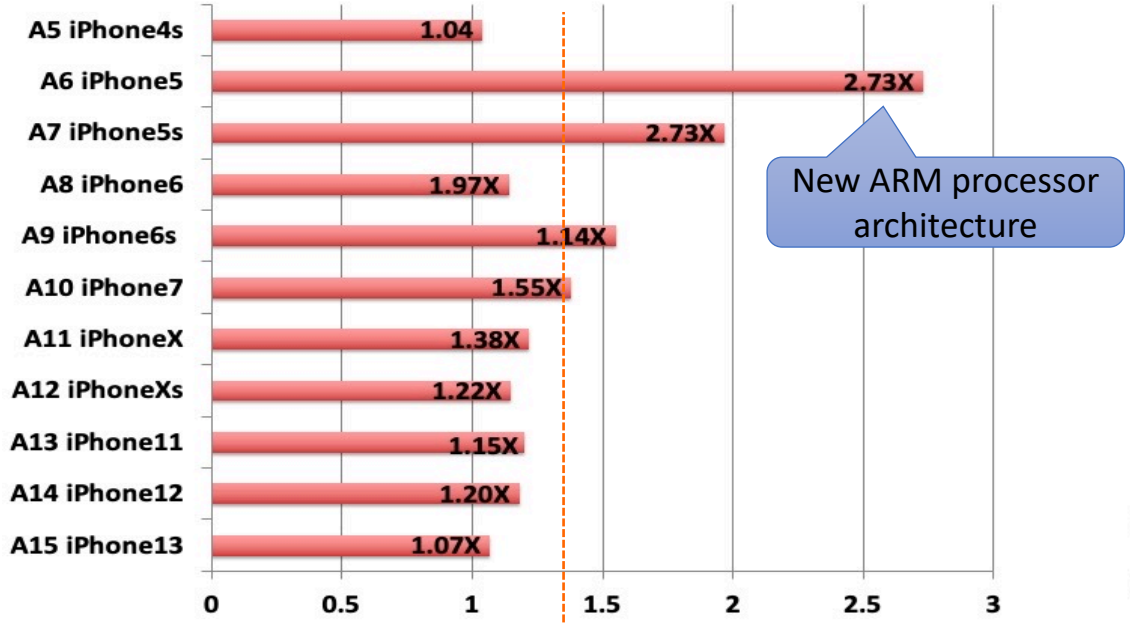
Chip	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
Year	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
Device	iPhone 4	iPhone 4s	iPhone 5	iPhone 5s	iPhone 6	iPhone 6s	iPhone 7	iPhone 8 & X	iPhone Xs	iPhone 11	iPhone 12	iPhone 13
Node	45nm Samsung	45nm Samsung	32nm Samsung	28nm Samsung	20nm TSMC	16nm TSMC	16nm TSMC	10nm TSMC	'7'nm TSMC	7nm TSMC	5nm TSMC	5nm TSMC
Area [cm2]	0.52	1.25	0.96	1.03	0.89	1.05	1.25	0.88	0.83	0.98	0.88	1.08



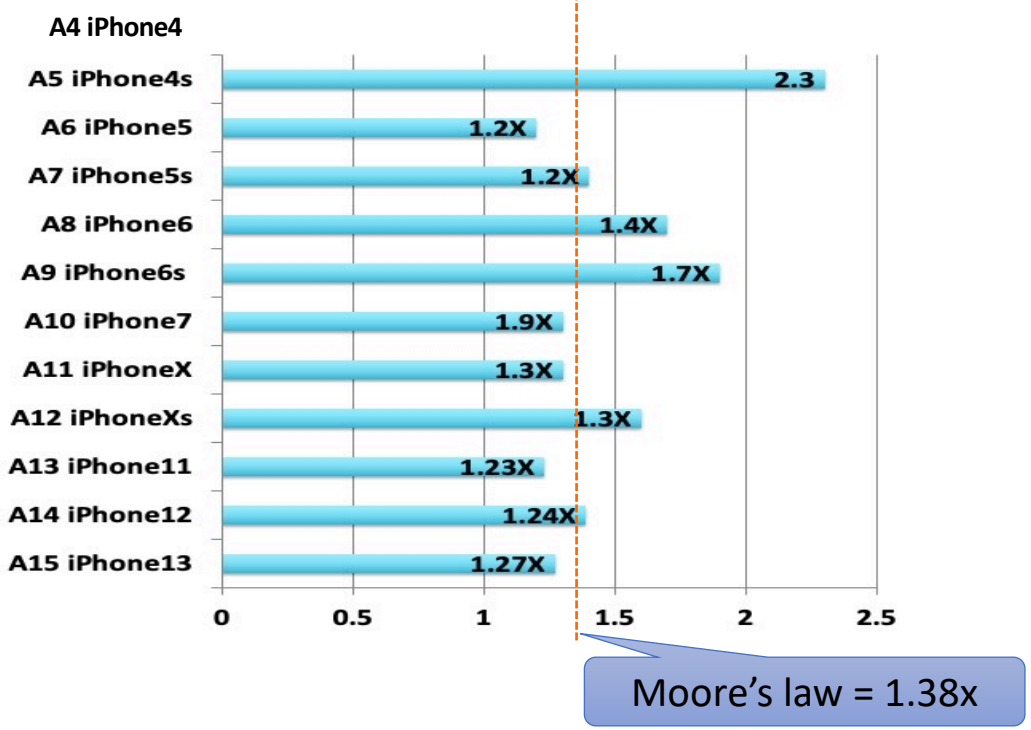
Die photos: chipworks/TechInsights
Data source: wikipedia

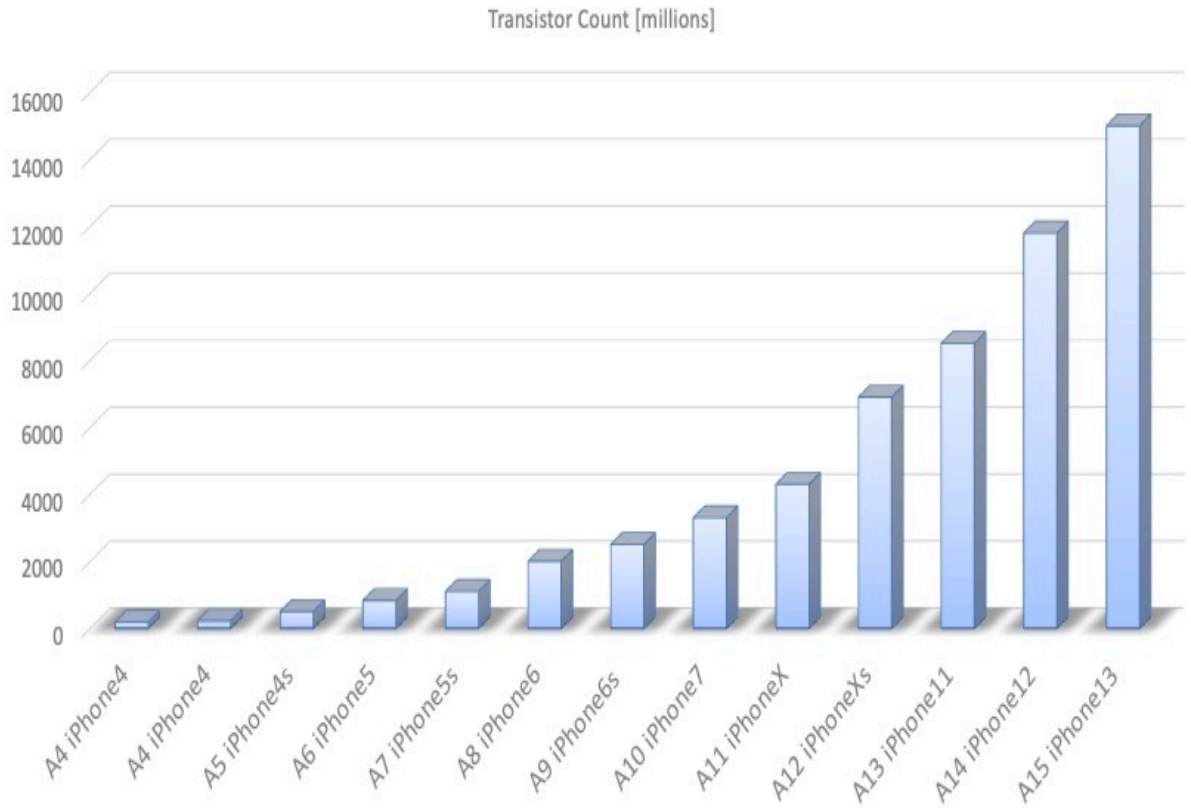
12 Years of Chip Scaling vs Predecessor

Geekbench4 Improvement (CPU, single-core)



Transistor Count increase vs previous chip

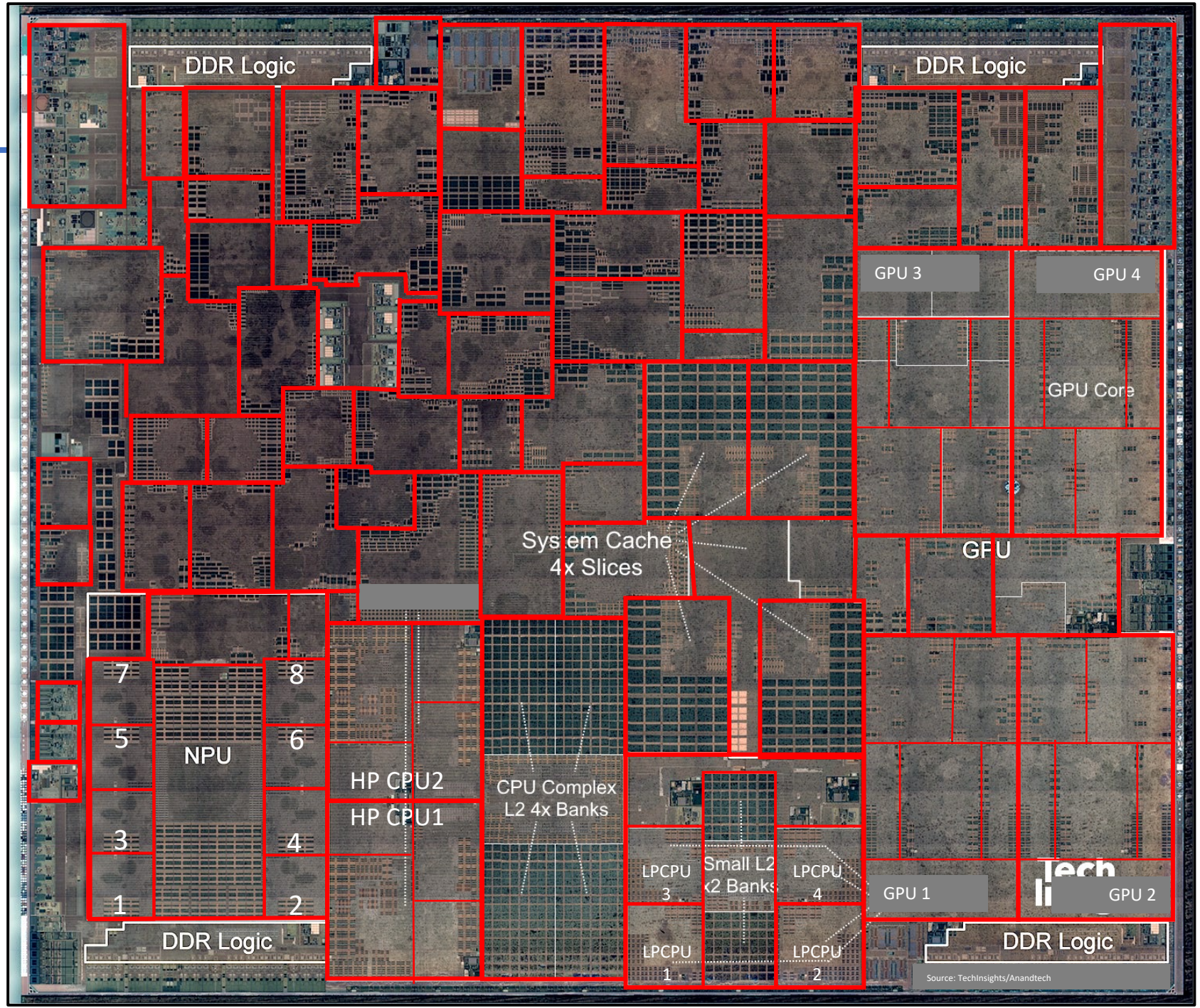




Apple A12 iPhone Xs (2018)

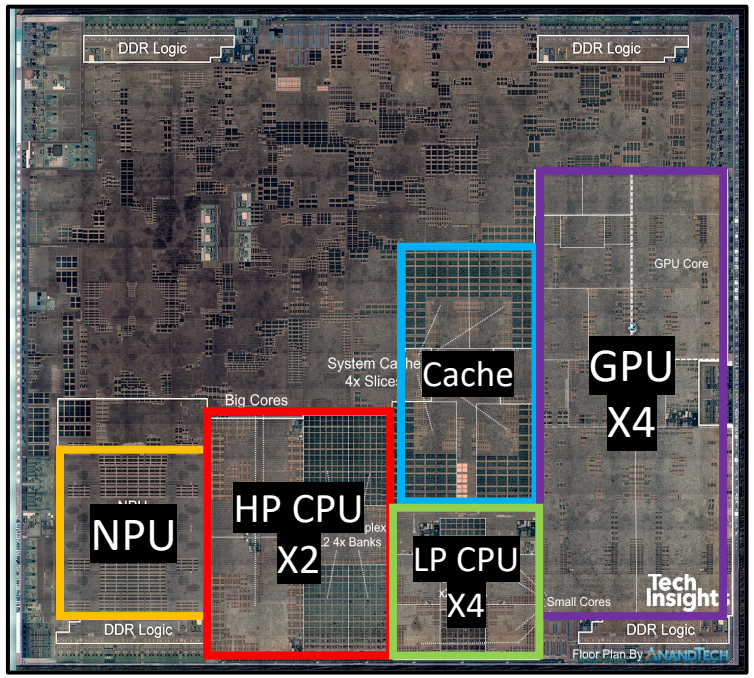
- 7nm TSMC FinFet
- 9.89 x 8.42 = 83.27 mm2
- 6.9 Billion transistors
- 4 **GPU** cores (~18%): 9 Blocks
- 6 **CPU** cores (~14%): 13 Blocks
  - 4 'Tempest' Low Power CPU cores
  - 2 'Vortex' High Performance CPU cores
  - L2 & L3 caches
- 8 **NPU/TPU** cores (~7%) 4 Blocks
- DDR (~3%) 1 block
- Misc (~57%): 50 Unique Blocks

- Total: ~75 unique blocks



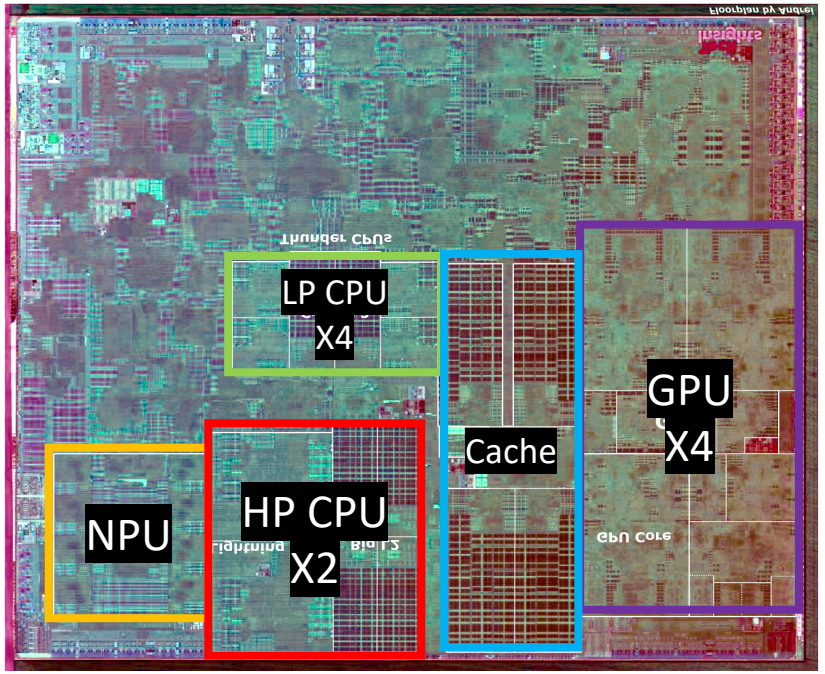
Machine Learning Hardware on a Mobile SoC

A12: iPhone Xs (2018)



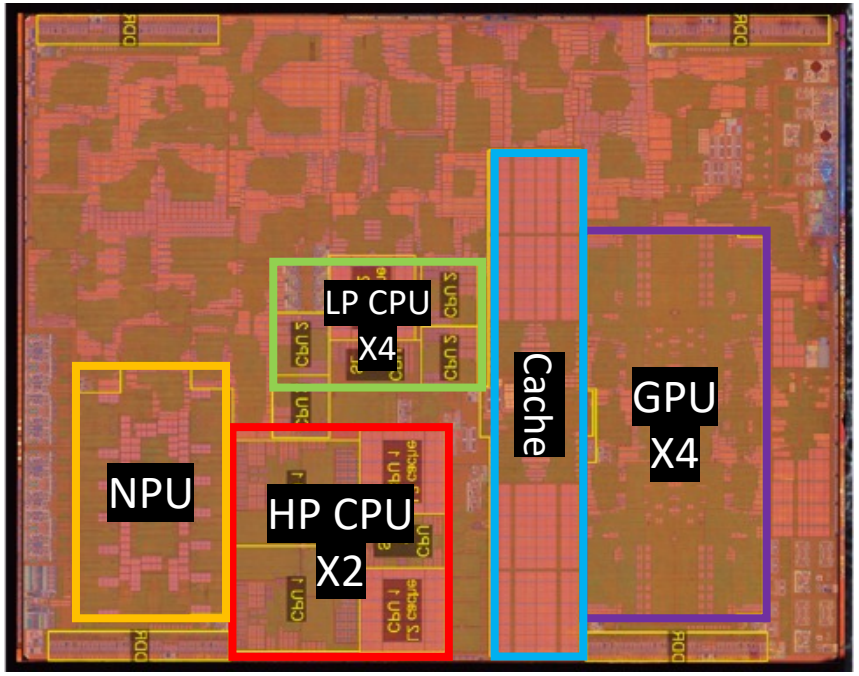
- A12: '7'nm TSMC
- Area: 83mm2
- 7.2B Transistors
- Cores: 2HP+4LP CPUs, 4 GPUs, 8NPUs

A13: iPhone 11 (2019)



- A13: 7nm TSMC
- Area: 89mm2
- 8.5B Transistors
- Cores: 2HP+4LP CPUs, 4 GPUs, 8NPUs

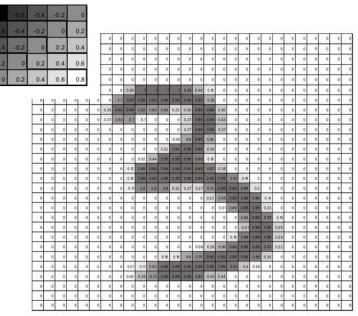
A14: iPhone 12 (2020)



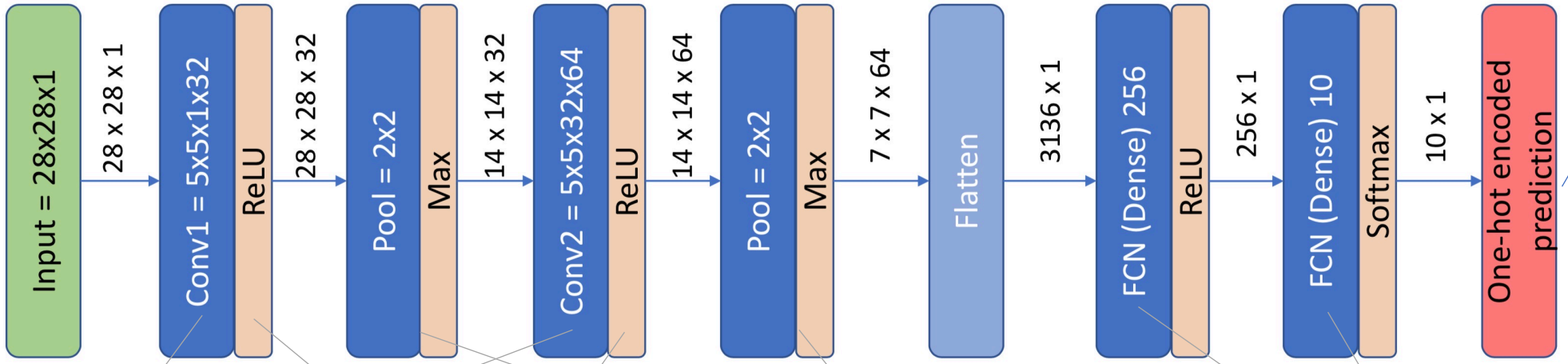
- A14: 5nm TSMC
- Area: 88mm2
- 11.8B Transistors
- Cores: 2HP+4LP CPUs, 4 GPUs, 12NPUs

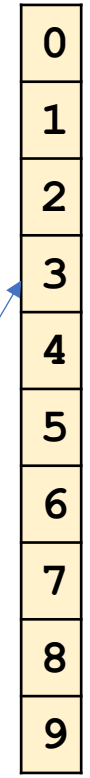
Sources: TechInsights/Anandtech

MNIST Neural Network with 7 layers

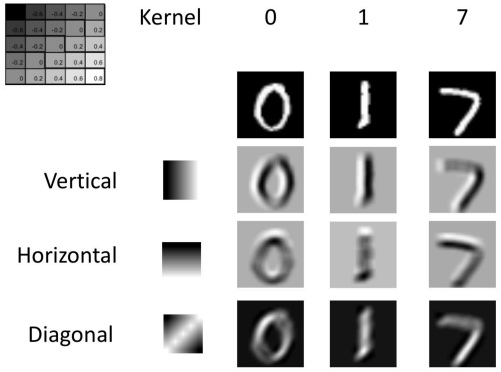


28x28 image
(MNIST database)

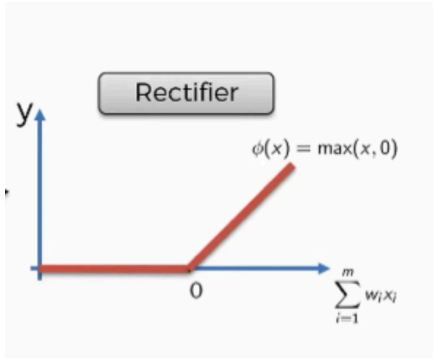




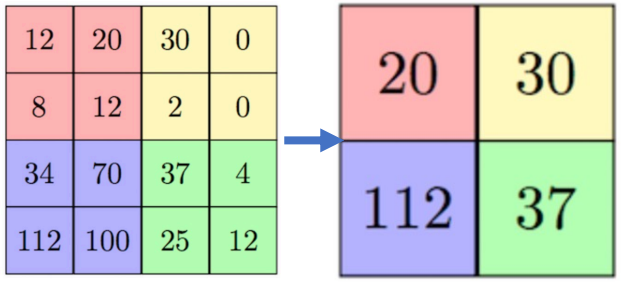
Convolutional filter



Rectified **L**inear **U**nit



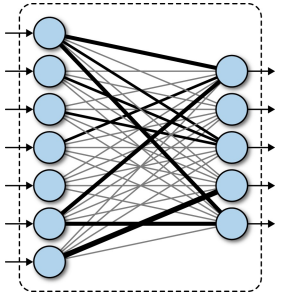
Max-Pool



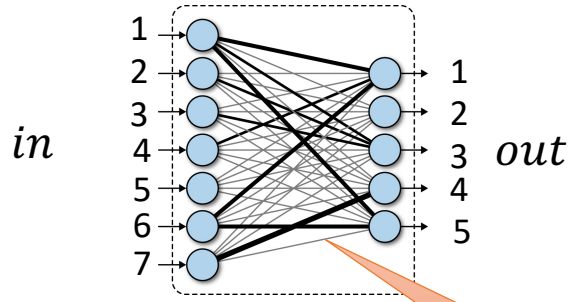
12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

20	30
112	37

Fully Connected Network



The Dense (Fully Connected) Layer



$$out_i = \sum_{j=1}^{j=7} in_j * w(i,j)$$

```
tensor dense_layer(const tensor & in,
                   const size_t outSize) {
  tensor out(outSize);
  for (int j = 1; j <= in.size(); j++) {
    for (int i = 1; i <= out.size(), i++) {
      out[i] += w[i][j] * in[j];
    }
  }
  return out;
}
```

Add

Multiply

Weight matrix is generally quite sparse

These weights are 'trainable'

IN

w(1,1)	w(2,1)	**w(3,1)**	w(4,1)	**w(5,1)**
W(1,2)	W(2,2)	**W(3,2)**	W(4,2)	W(5,2)
W(1,3)	W(2,3)	**W(3,3)**	W(4,3)	W(5,3)
W(1,4)	W(2,4)	W(3,4)	W(4,4)	W(5,4)
W(1,5)	W(2,5)	W(3,5)	W(4,5)	W(5,5)
W(1,6)	W(2,6)	W(3,6)	W(4,6)	**W(5,6)**
W(1,7)	W(2,7)	W(3,7)	**W(4,7)**	W(5,7)

OUT

Convolution Network Layer: Code Sketch

```
tensor rgb_convolution_layer(const tensor & input,
                             const convFilter & weights)
{
  tensor output(weights.size(), weights.width(), weights.height());

  for (color = red; color <= blue; color++) {
    for (int row = 0; row < input.width(); row++) {
      for (int column = 0; column < input.height(), row++) {
        for (int filter = 0; filter < weights.size(), filter++) {
          for (int i = 0; i < weights[filter].width(), i++) {
            for (int j = 0; j < weights[filter].height(), j++) {
              output[filter][row][column] +=
               weights[filter][i][j] * input[color][row+i][column+j];
          }
        }
      }
    }
  }
  return output;
}
```

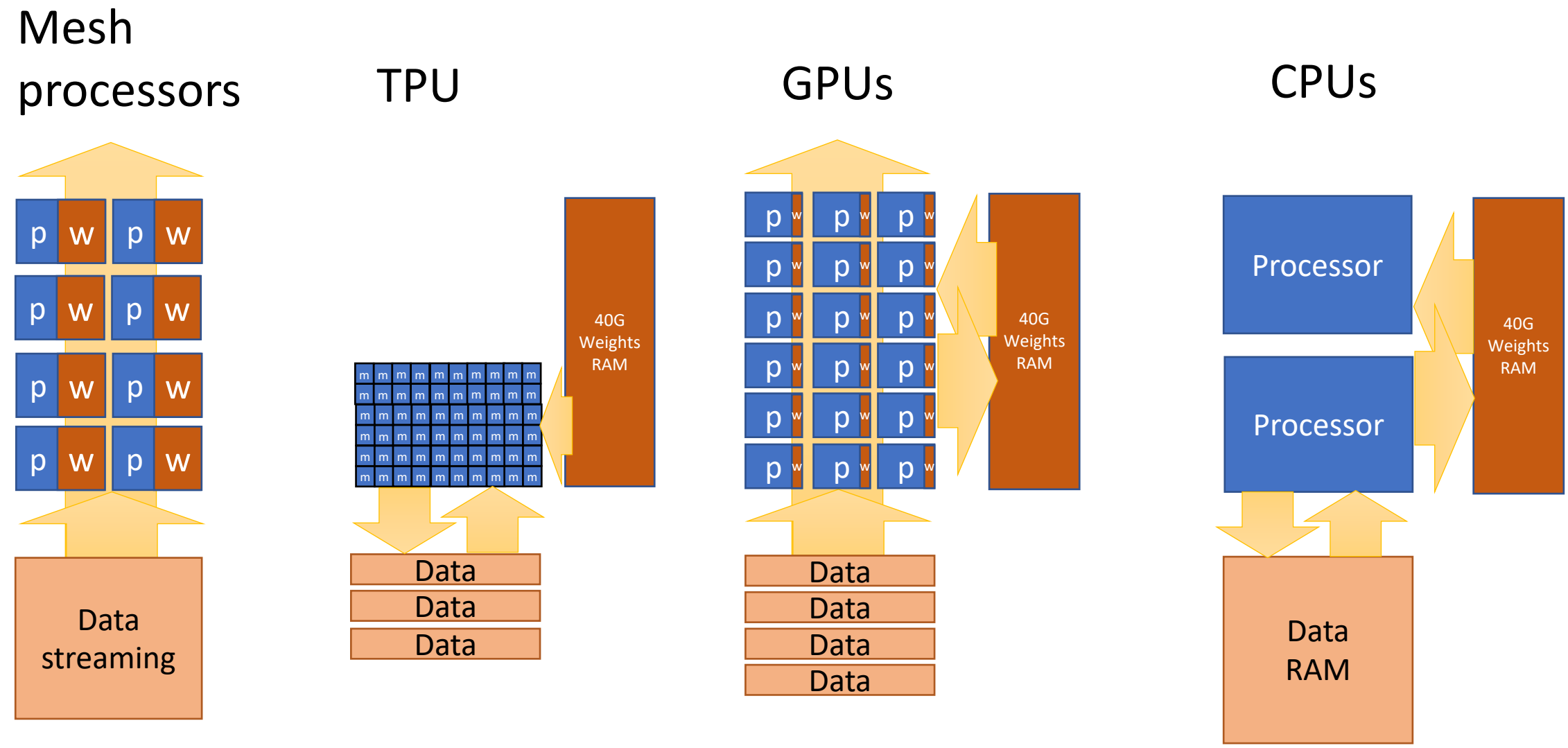
Multiply

Add

3 X
3840 X
2160 X
32 X
5 X
5
=
19,906,560,000

Typical Smartphone picture: 3X3840x2160 pixels (RGB), 32 filters or 5x5 pixels each
So, this convolution layers would require 3x3840x2160x32x5x5 = **19,906,560,000** Multiply-accumulate per layer

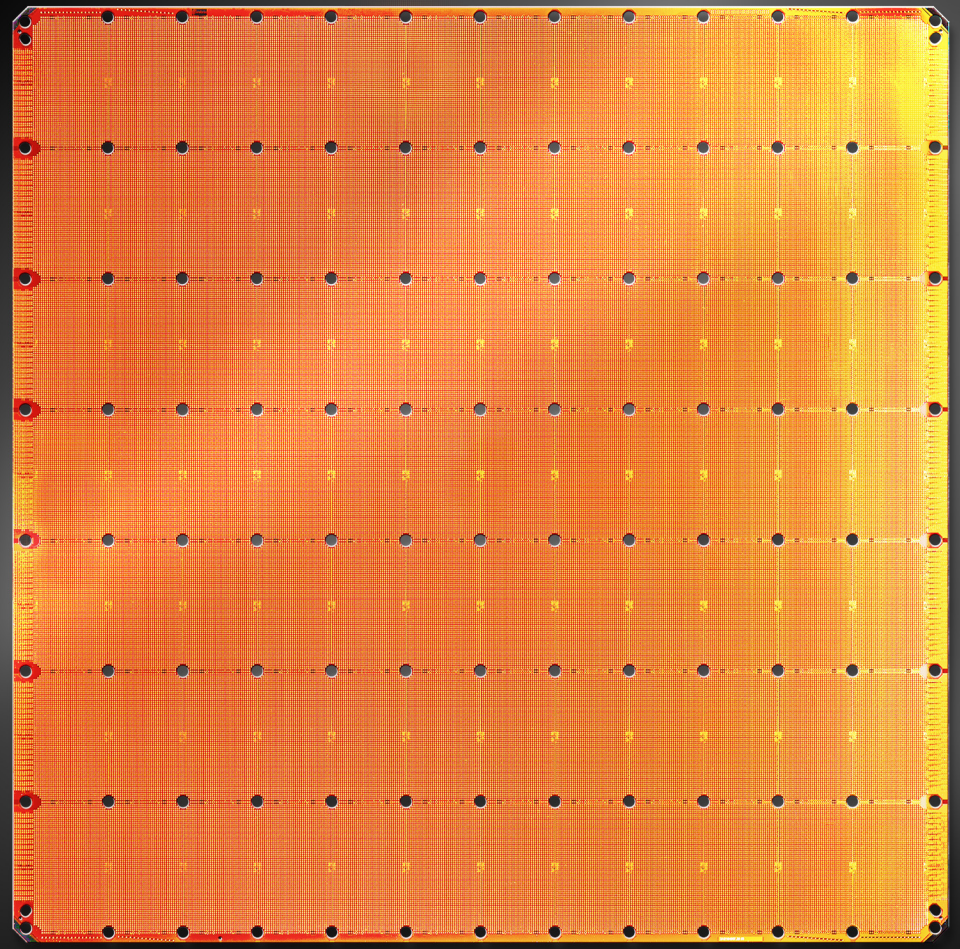
Bringing Memory and Processing Closer



Cerebras
Wafer Scale Engine
AI Supercomputer for Layer-Pipelined computing

- Startup based in ~~Los Altos~~ Sunnyvale
  - ~300 people
- Unicorn with >$300M funding
- Focus: Supercomputer for ML training
- Single 21.5cm by 21.5cm chip in 7nm
  - 2.4 Trillion transistors
- 850,000 ‘AI processor cores’
  - In a 800 by 1060 array
- Total on-chip memory:
  ~40Gb fast SRAM
- ~100 PetaByte/second fabric bandwidth

Cerebras Wafer Scale Engine



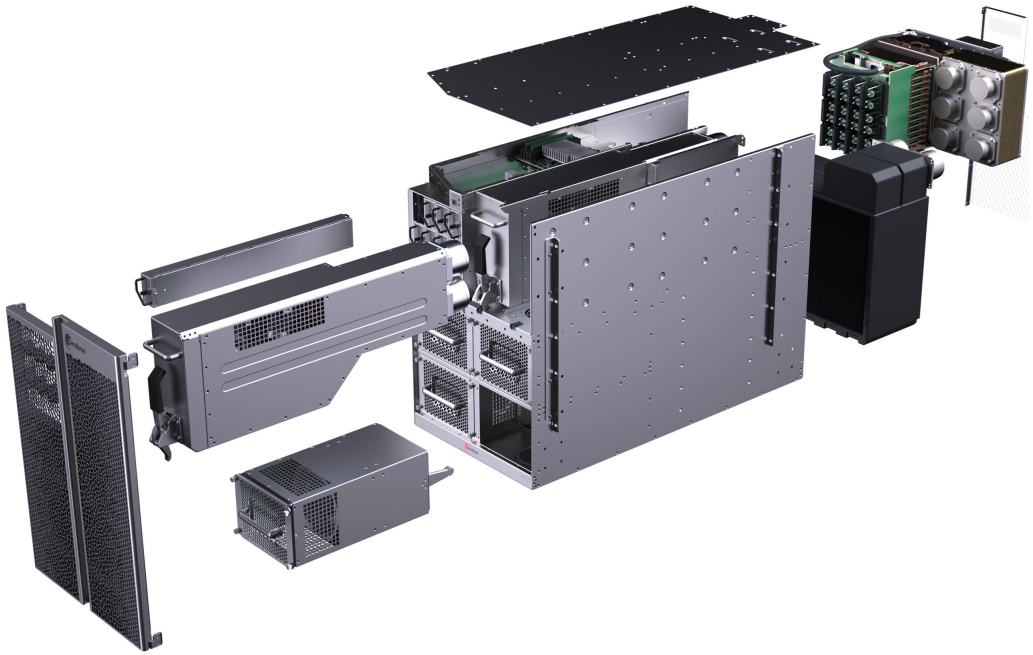
Cerebras WSE

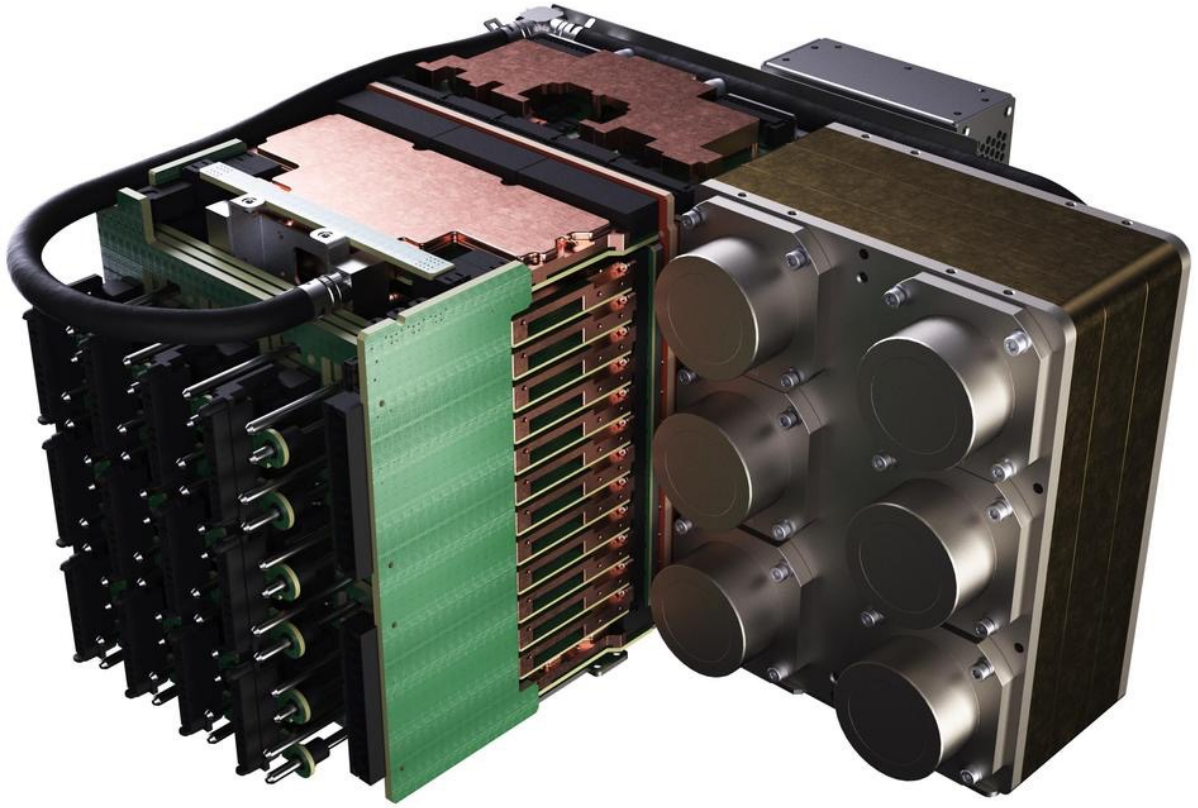
46,225 mm² Silicon

Largest GPU

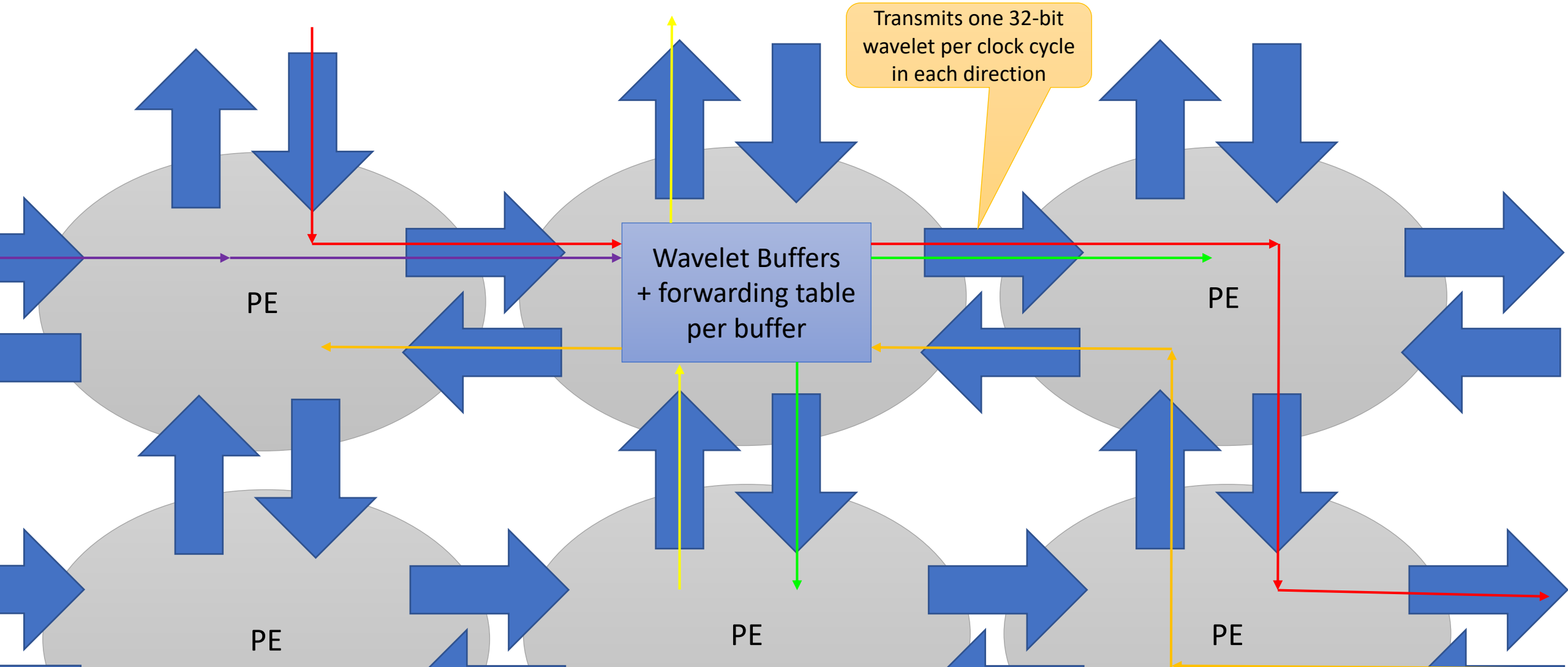
815 mm² Silicon

CS-1 Hardware: the Box



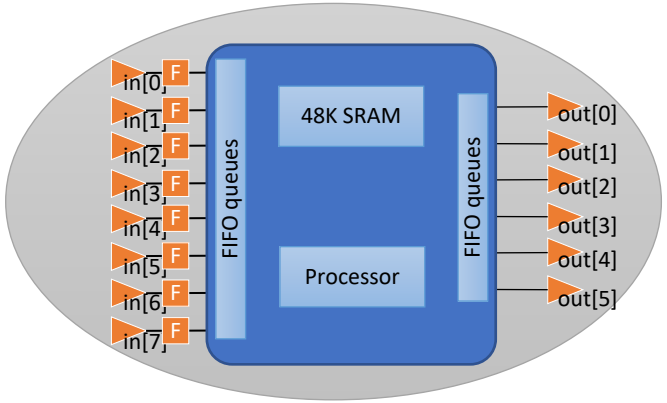


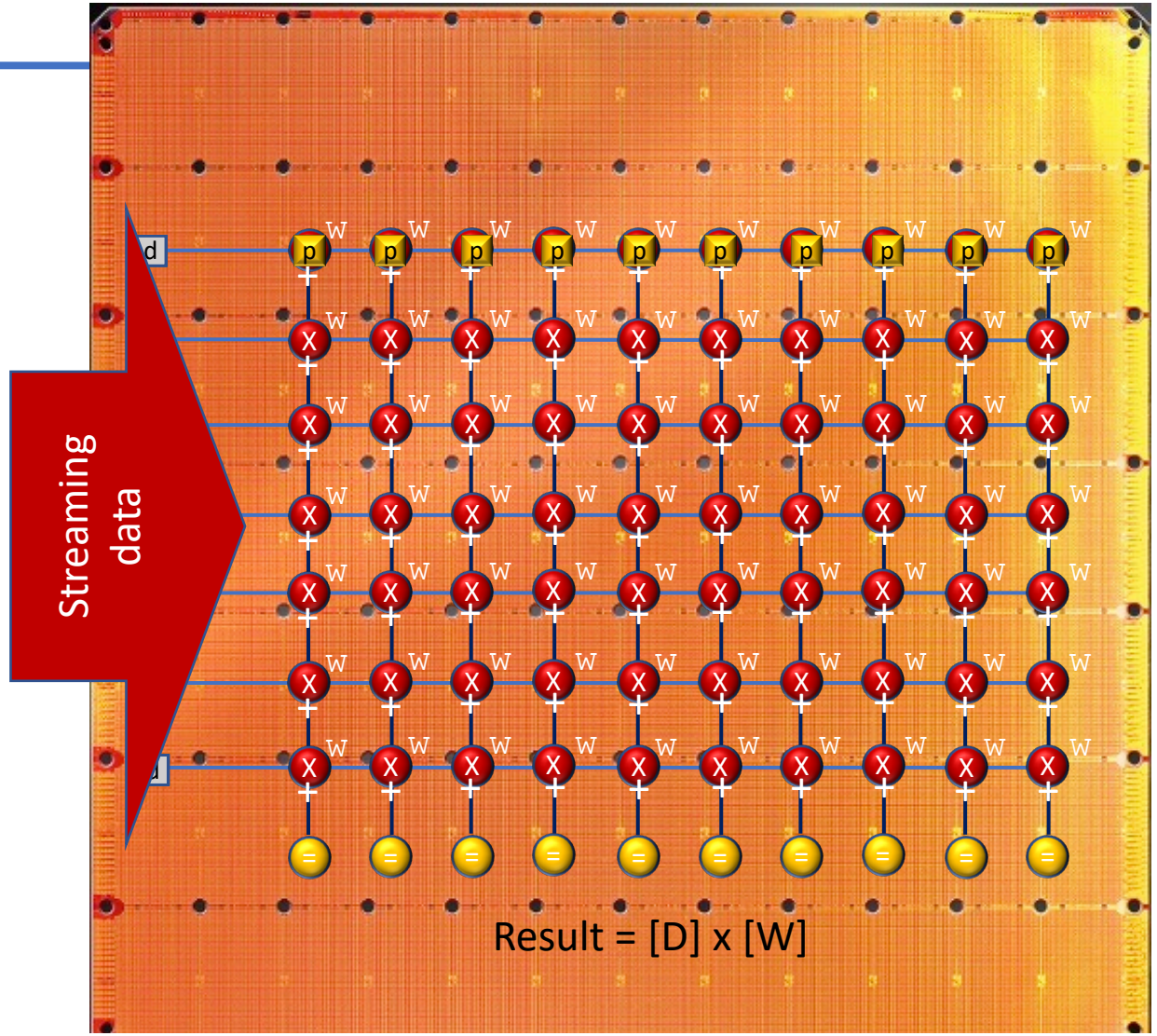
Wavelets Flowing on the Network-on-Chip



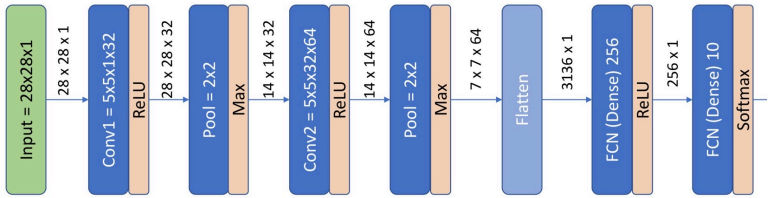
A Dense Layer Kernel in Hardware

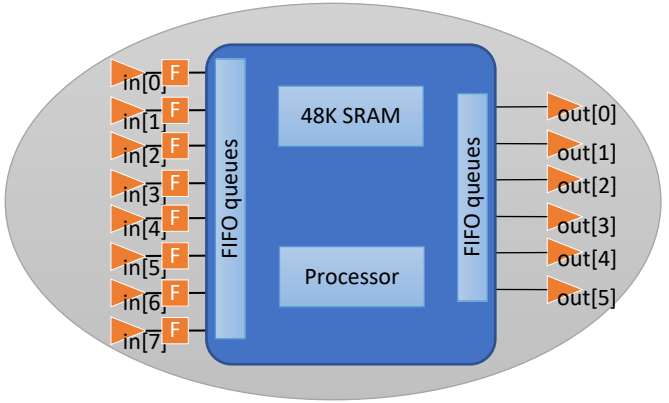
- The ML layer implementation on CS-1 is massive parallel multiply-accumulate:
  - The **data** operand is efficiently streamed in at a high rate
  - The **weight** operand is stationary in 40Gb high speed memory
  - Result (sum-of-products) is streamed out to the next layer

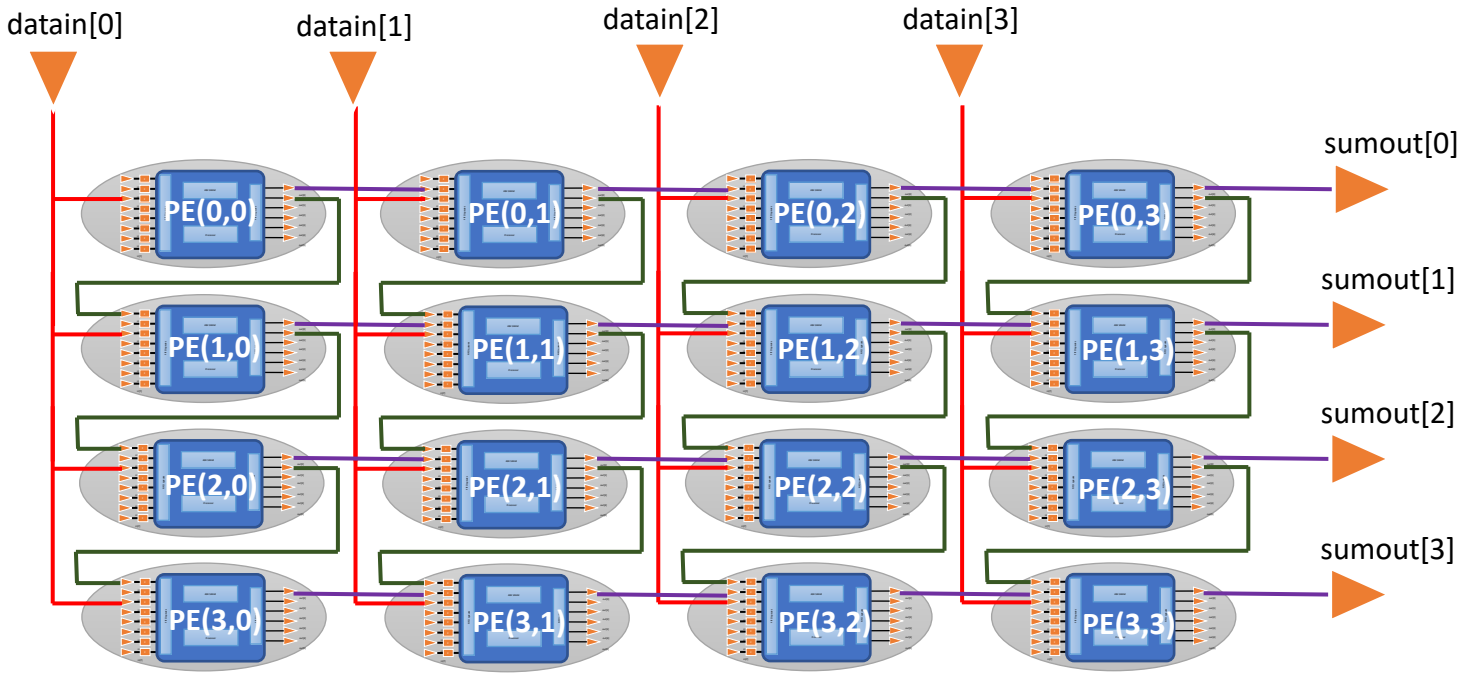




Netlist and Routing of Processor Elements

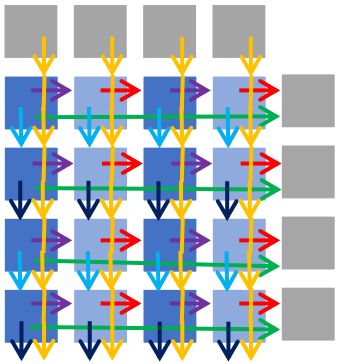






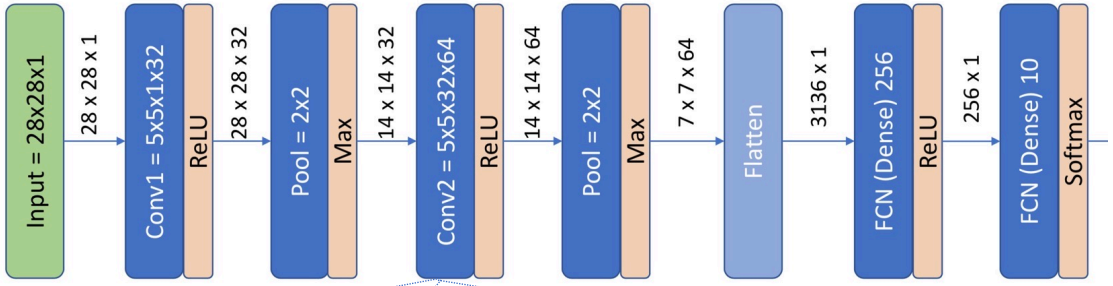
Fully Connected Network Kernel Netlist

route

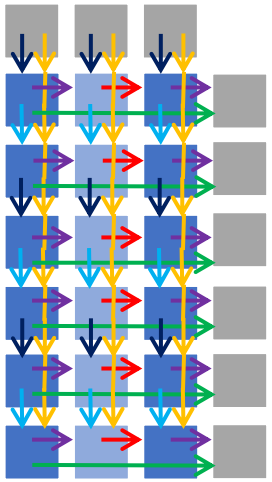


Programmed Fabric Interconnect as (4+1)X(4+1) PEs

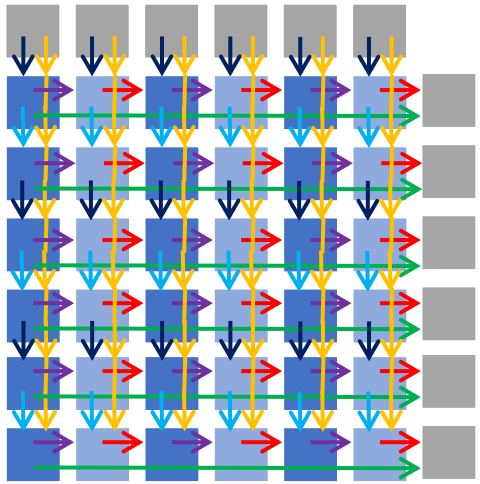
Area vs Performance of a Kernel



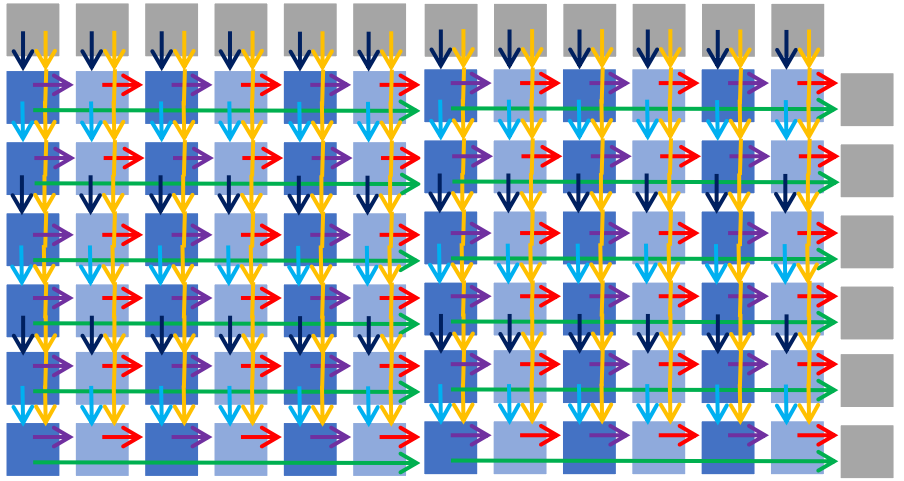
2X base task per PE as 3X6 core
Expect ~2X slower, 2X memory



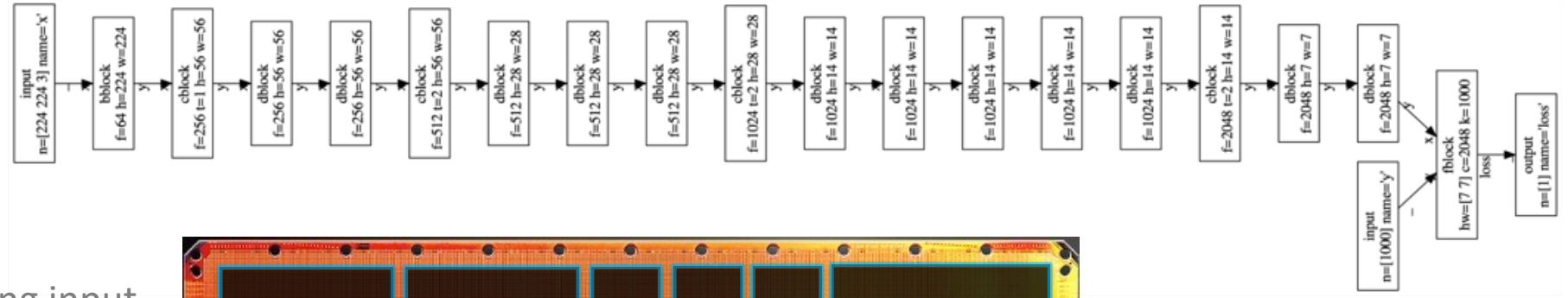
Base task, 6X6 core



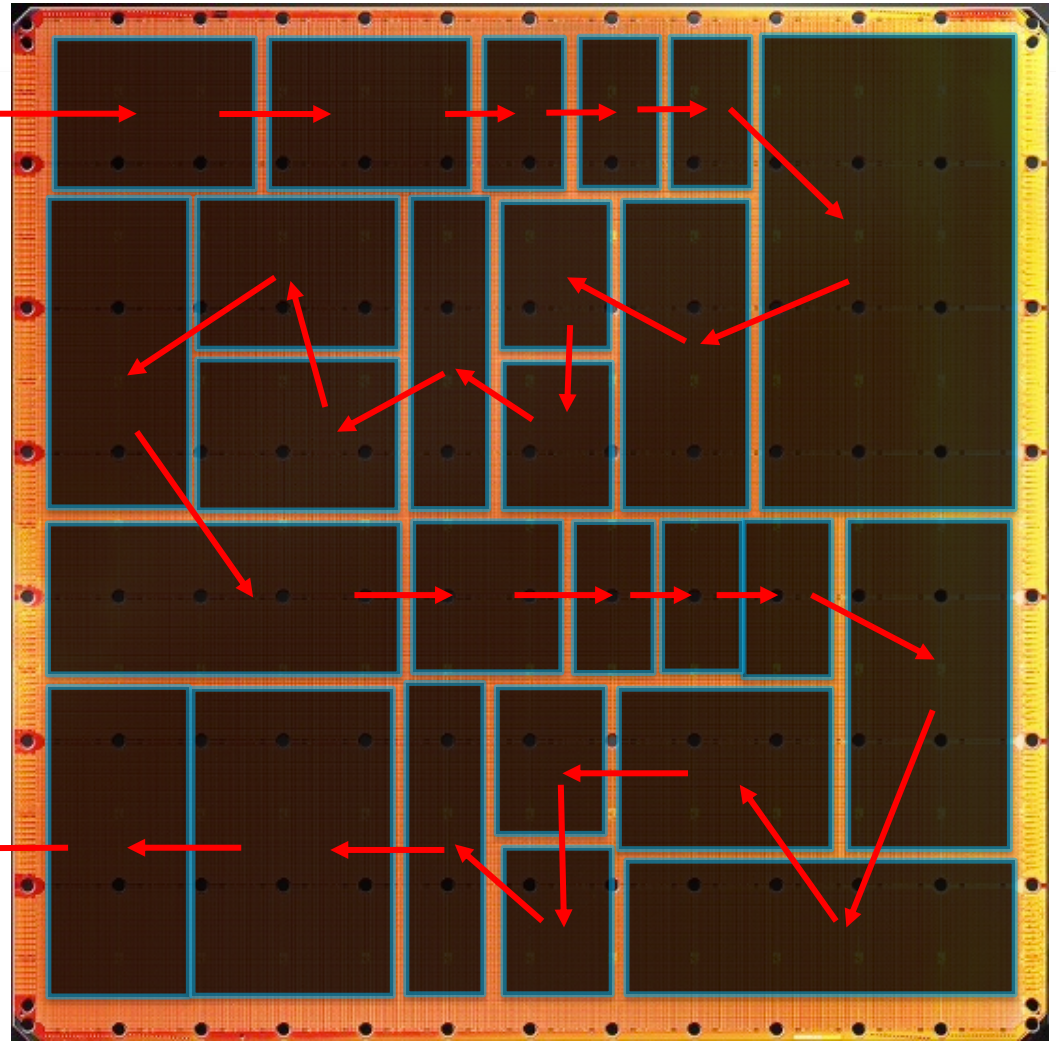
½ X base task per PE as 12X6 core
Expect ~2X faster, ½ memory



Layer-pipelined execution



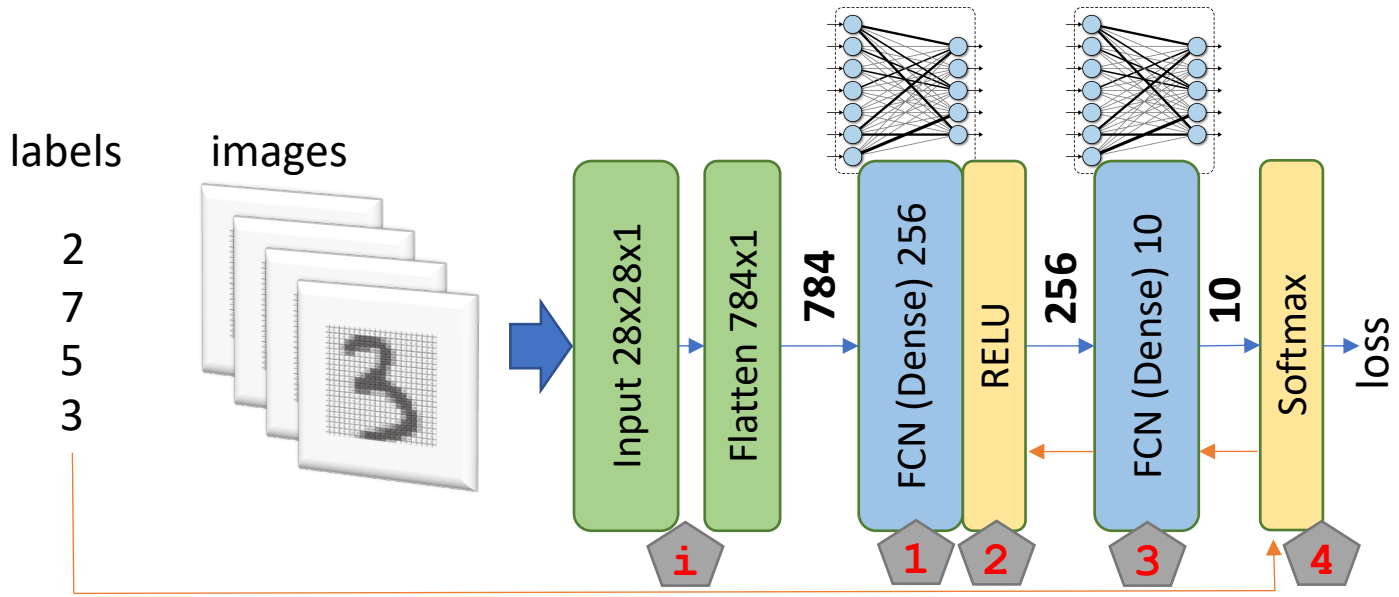
Continuously Streaming input



cat

Continuously Streaming output

Training a Simplified MNIST TensorFlow Model



File **data.py**

```
def mnist_input_fn(
    mode=tf.estimator.ModeKeys.TRAIN
):
    mnist_loader = MnistLoader(
        img_dtype=tf.float32,
        label_dtype=tf.float32,
        flatten=True,
    )
    dset = mnist_loader.get_dataset(
        'train',
         batch_size=1, num_batches=20
    ).repeat()

    return dset
```

File **train.py**

```
import tensorflow as tf
from data.py  import mnist_input_fn
from model.py import mnist_model_fn

estimator = tf.Estimator(
    model_fn,
)

estimator.train(
    input_fn=input_fn,
)
```

File **model.py**

```
def mnist_model_fn(
    features, labels, mode=tf.estimator.ModeKeys.TRAIN
):
    dtype = tf.keras.mixed_precision.experimental.Policy(
            'mixed_float16', loss_scale=None)
    tf.keras.mixed_precision.experimental.set_policy(dtype)

    network = tf.keras.layers.Dense(
        256,
        activation=tf.nn.relu,
    )(features)

    logits = tf.keras.layers.Dense(
        10,
    )(network)

    loss_op = tf.reduce_mean(
        input_tensor=tf.nn.softmax_cross_entropy_with_logits(
            labels=tf.stop_gradient(labels),
            logits=logits,
        )
    )

    train_op = tf.compat.v1.train.GradientDescentOptimizer(
        learning_rate=0.01).minimize(loss_op)

    return tf.estimator.EstimatorSpec(
        mode=mode, loss=loss_op, train_op=train_op
    )
```

Logits = unnormalized predictions

Synthesis Flow From TensorFlow to Training Hardware

File **model.py**

```
def mnist_model_fn(
    features, labels, mode=tf.estimator.ModeKeys.TRAIN
):
    dtype = tf.keras.mixed_precision.experimental.Policy(
            'mixed_float16', loss_scale=None)
    tf.keras.mixed_precision.experimental.set_policy(dtype)

    network = tf.keras.layers.Dense(
        256,
        activation=tf.nn.relu,
    )(features)

    logits = tf.keras.layers.Dense(
        10,
    )(network)

    loss_op = tf.reduce_mean(
        input_tensor=tf.nn.softmax_cross_entropy_with_logits(
            labels=tf.stop_gradient(labels),
            logits=logits,
        )
    )

    train_op = tf.compat.v1.train.GradientDescentOptimizer(
        learning_rate=0.01).minimize(loss_op)

    return tf.estimator.EstimatorSpec(
        mode=mode, loss=loss_op, train_op=train_op
    )
```

Compile

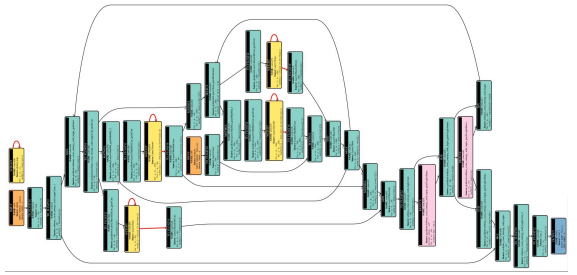
TensorFlow model

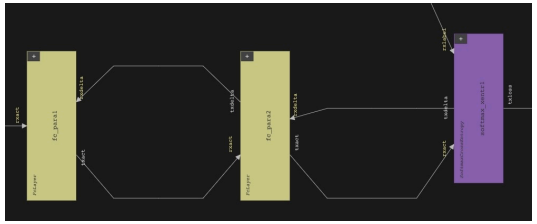
Intermediate **R**epresentation
MLIR, XLA

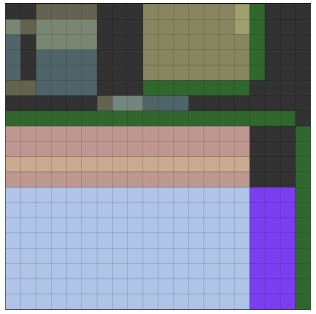
Match to Computation Kernels

Shape, Place and Route

Program Hardware





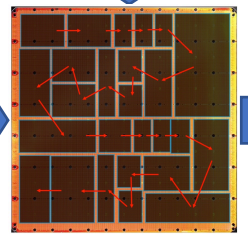


File **data.py**

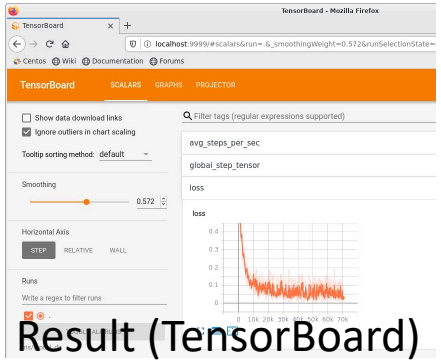
```
def mnist_input_fn(
    mode=tf.estimator.ModeKeys.TRAIN
):
    mnist_loader = MnistLoader(
        img_dtype=tf.float32,
        label_dtype=tf.float32,
        flatten=True,
    )
    dset = mnist_loader.get_dataset(
        'train',
         batch_size=1, num_batches=20
    ).repeat()

    return dset
```

Stream data



Weights, losses



Result (TensorBoard)

Summary: Mapping TensorFlow to Hardware

- Complex process with many steps
  - Much like the familiar EDA flow
- System requires many abstraction levels:
  - TensorFlow graph
  - Intermediate Representation
  - Matched kernels
  - Place & Route
  - Machine code for the Network-on-Chip between Processing Elements
  - Machine code for the computation in each Processing Element
  - Actual CS-1 hardware, WSE + several embedded processors
  - Programming hardware, orchestration of data streaming processes