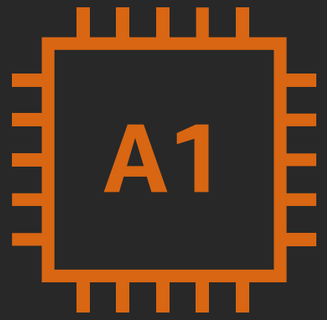
Systolic Array and Tensorization

Key Components of a Deep Learning Accelerator

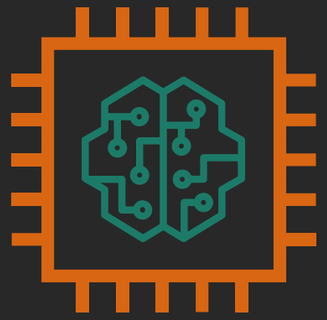
Ron & Randy

Some of AWS Custom Chips lines



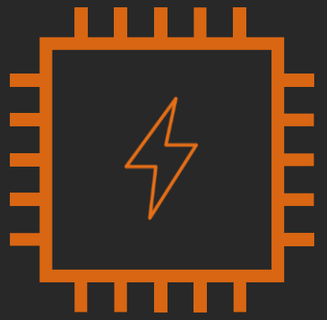
AWS Graviton

(Powerful and Efficient for Server for Modern Applications)



AWS Inferentia

(Machine Learning Hardware and Software at Scale)

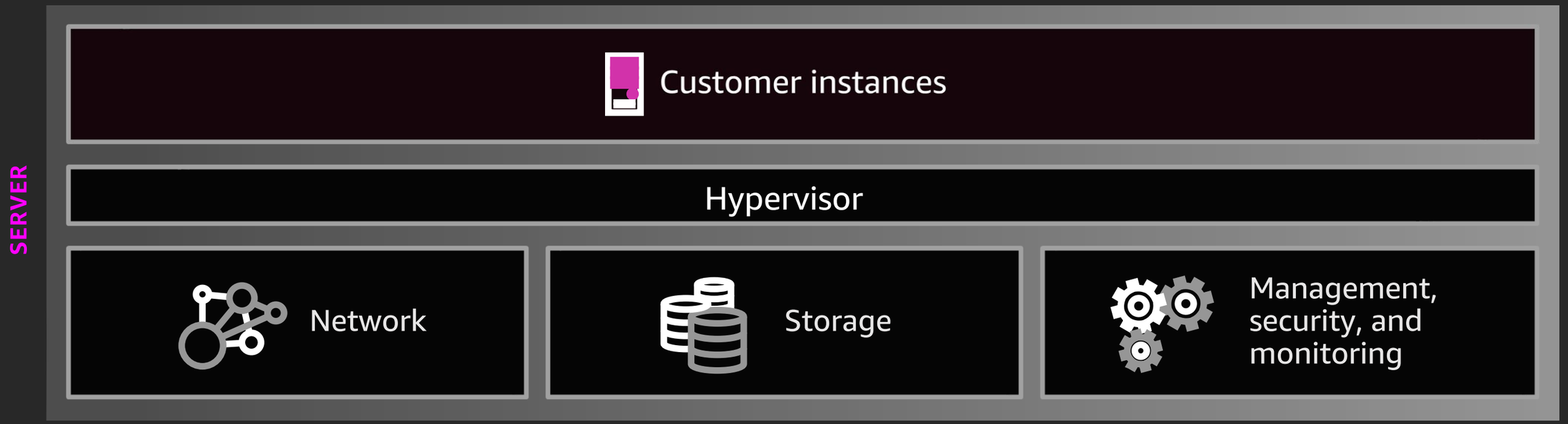


AWS Nitro System

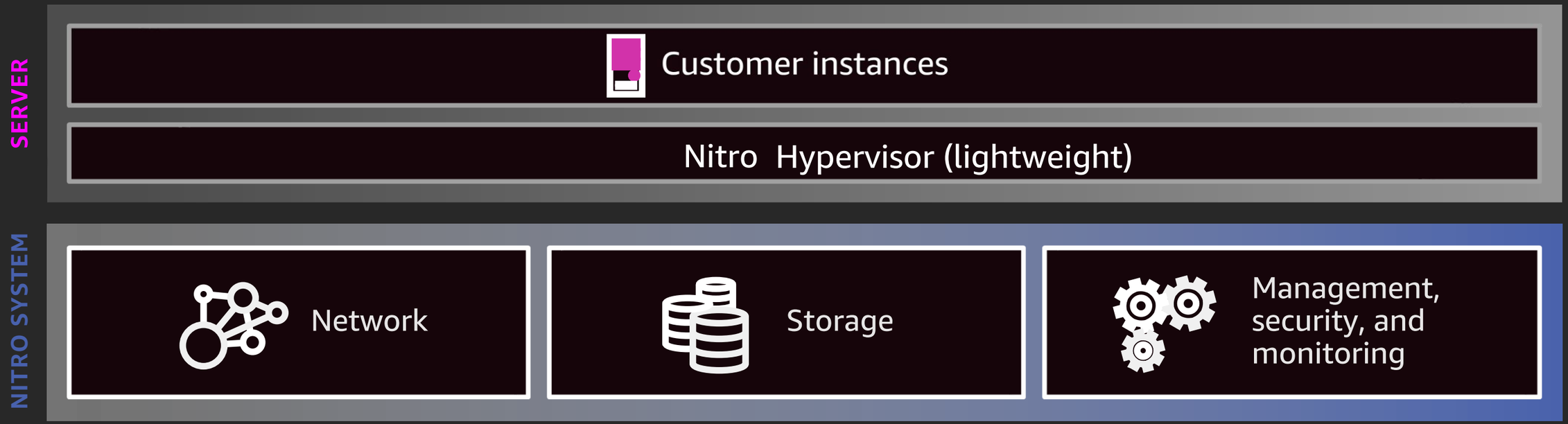
(Cloud Hypervisor, Network, Storage, and Security)

100% Developed in the Cloud: RTL → GDS2

Original EC2 host



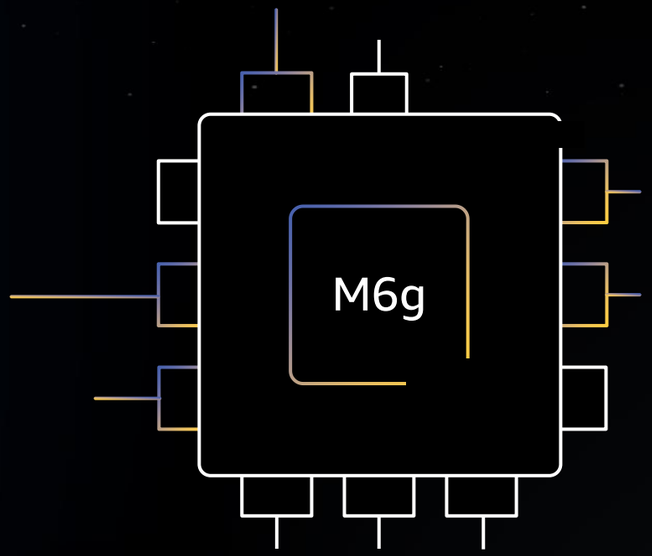
EC2 host with the Nitro System



Available Now!

M6g, R6g, C6g instances

POWERED BY AWS GRAVITON2 PROCESSOR



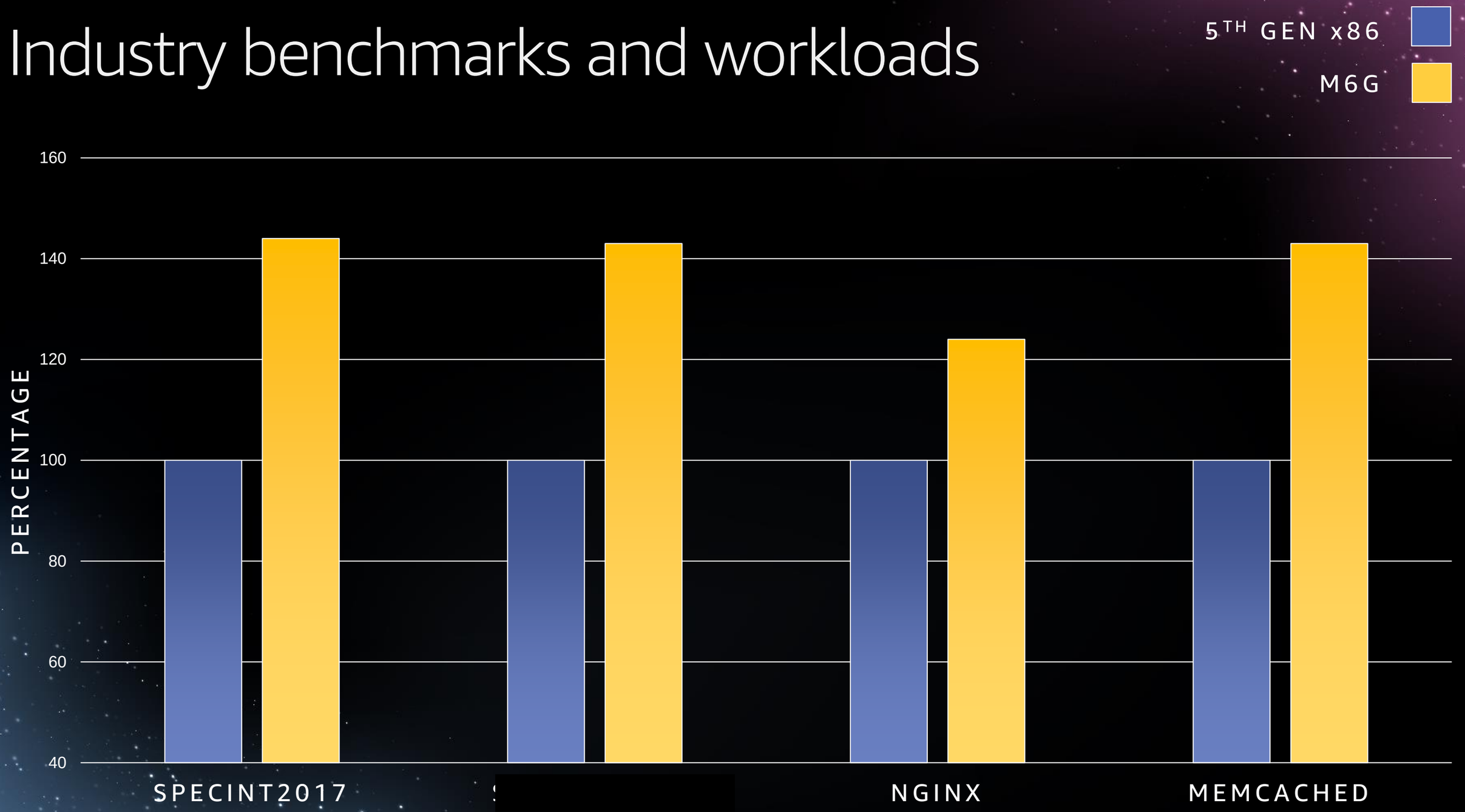
New generation of Arm-based instances powered by AWS Graviton2 processors offers 40% better price-performance than current x86-based instances

M6G
General Purpose

C6G
Compute Optimized

R6G
Memory Optimized

Industry benchmarks and workloads



MACHINE LEARNING IS HAPPENING IN COMPANIES OF EVERY SIZE AND INDUSTRY

Tens of thousands customers have chosen AWS for their ML workloads | More than twice as many customers using ML than any other cloud provider

ML inference deployment options on Amazon EC2

Custom chip
EC2 Inf1 instances

Featuring AWS Inferentia

Applications that leverage common ML frameworks

Powered by AWS Inferentia

Best price/performance for ML inferencing in the cloud

Up to 2x lower inference than EC2 G4 instances

GPU based
EC2 G4 instances

Applications that require access to CUDA, CuDNN or TensorRT libraries

Amazon EC2 G4 instances based on NVIDIA T4 GPUs

CPU based
EC2 C6g instances

Low sensitivity to inference performance

Graviton2 CPUs

Agenda

- AWS breadth and depth, ML market, inference options
- Systolic Array
- Neuron SDK
- Tensorization
- Customer story: Alexa Text-to-Speech (TTS)

Best price/performance for DL inference in the cloud

Natural language:

Instance type	Throughput (Seq/Sec)	OD Price ($/Hr)	Cost-per-1M inferences	Throughput: Inf1 vs. G4	Cost-per-inference: Inf1 vs. G4
inf1.xlarge	360	$0.368	$0.284	38% higher	49% lower
g4dn.xlarge	260	$0.526	$0.562		

Results based on running BERT-Large model end-to-end with TensorFlow, seqlen=128

Image classification:

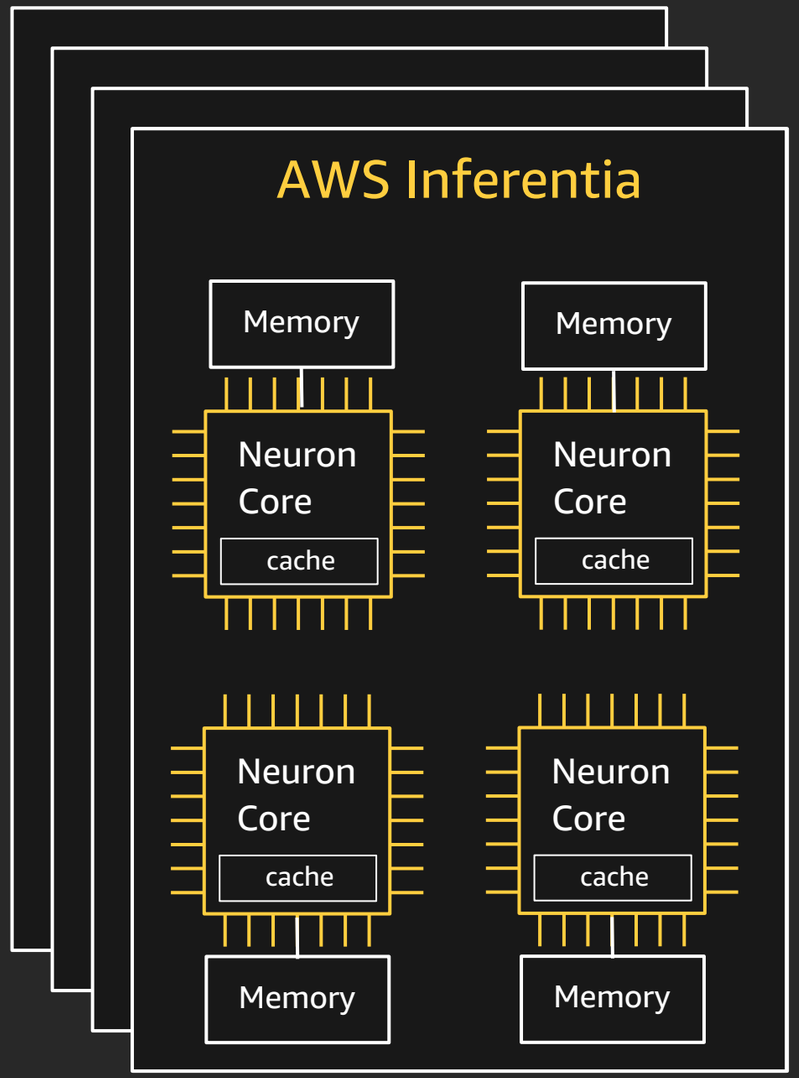
Instance type	Throughput (Img/Sec)	OD Price ($/Hr)	Cost-per-1M inferences	Throughput: Inf1 vs. G4	Cost-per-inference: Inf1 vs. G4
inf1.xlarge	2,226	$0.368	$0.045	24% higher	45% lower
g4dn.xlarge	1,792	$0.526	$0.082		

Results based on running ResNet-50 model end-to-end with TensorFlow

AWS Inferentia quick tour

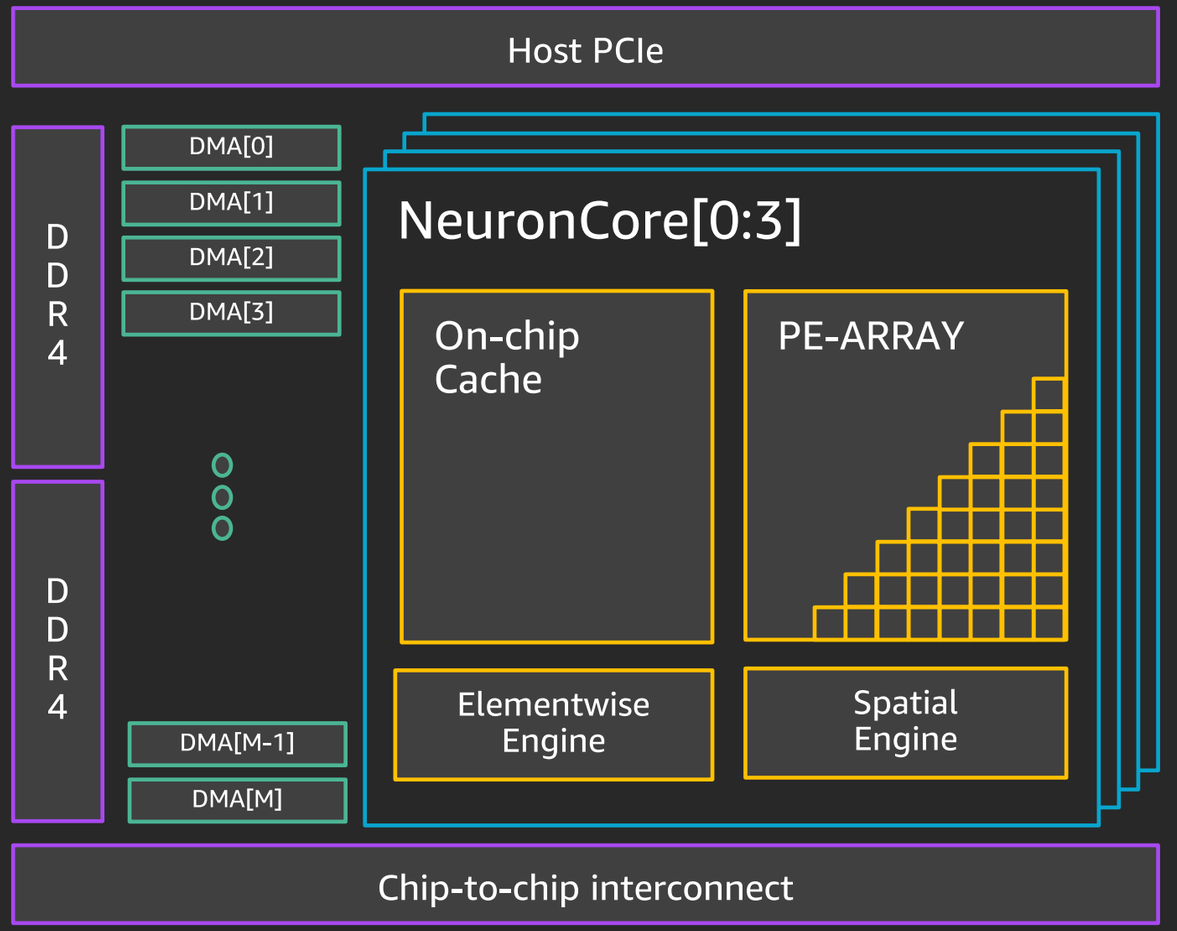
AWS custom built: chip, software, and server

- 4 NeuronCores
- Up to 128 TOPS
- 2-stage memory hierarchy
  - Large on-chip cache and commodity DRAM
- Fast chip-to-chip interconnect
- Supports FP16, BF16, INT8 data types



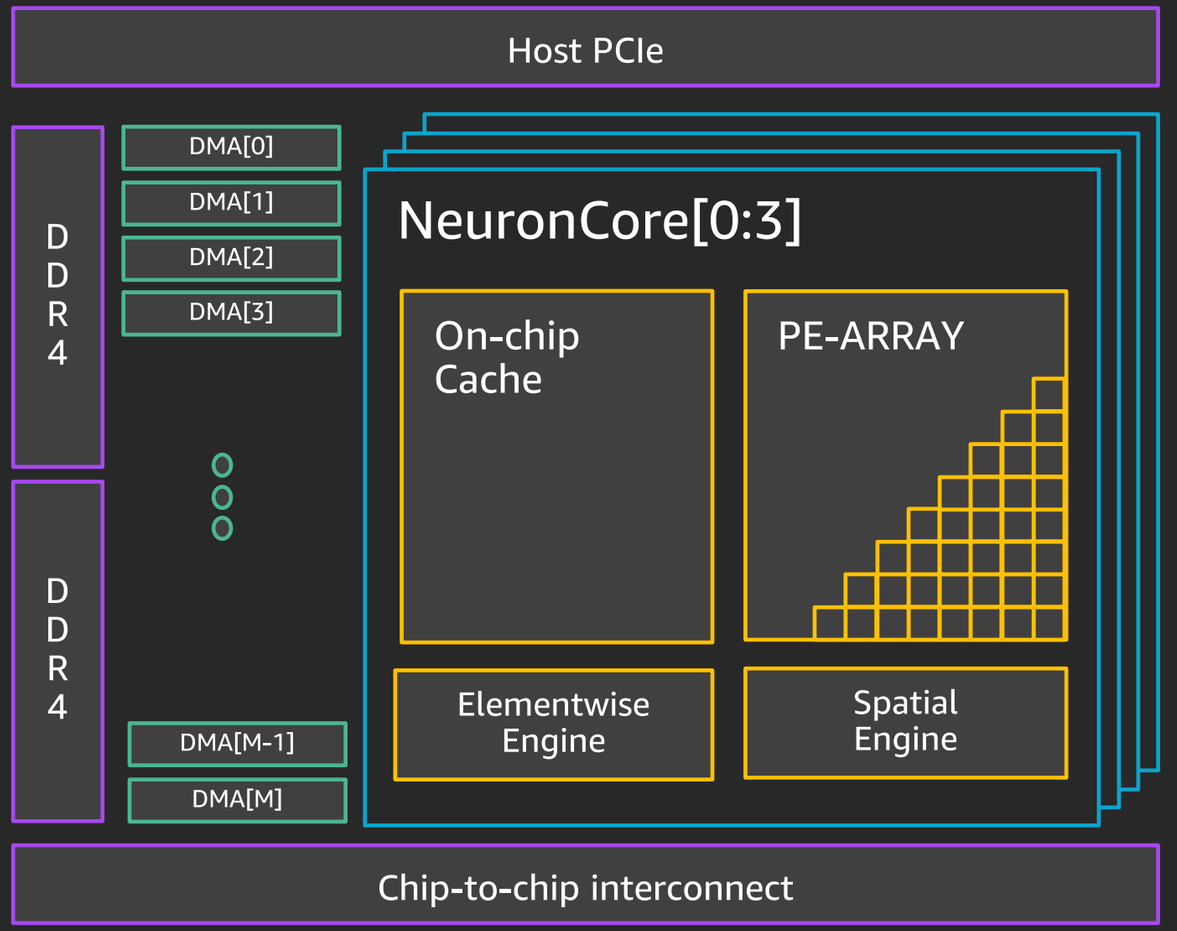
AWS Inferentia Architecture

- At the heart of a NeuronCore:
  - Systolic array
  - On-chip cache
  - Spatial/elementwise general-purpose engines



AWS Inferentia Architecture

- Why systolic array?

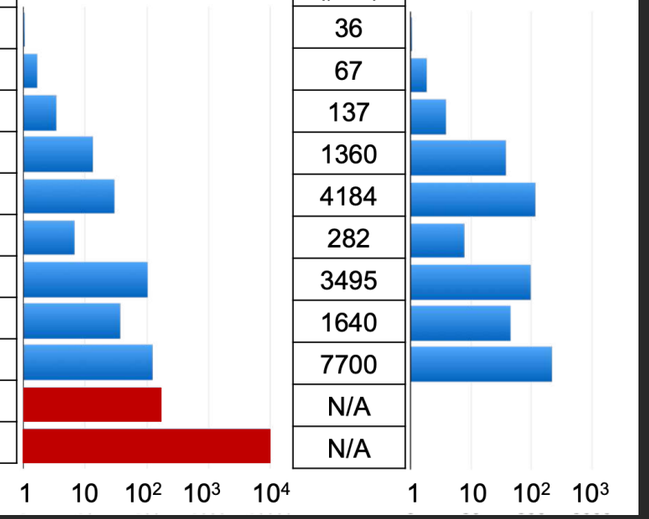


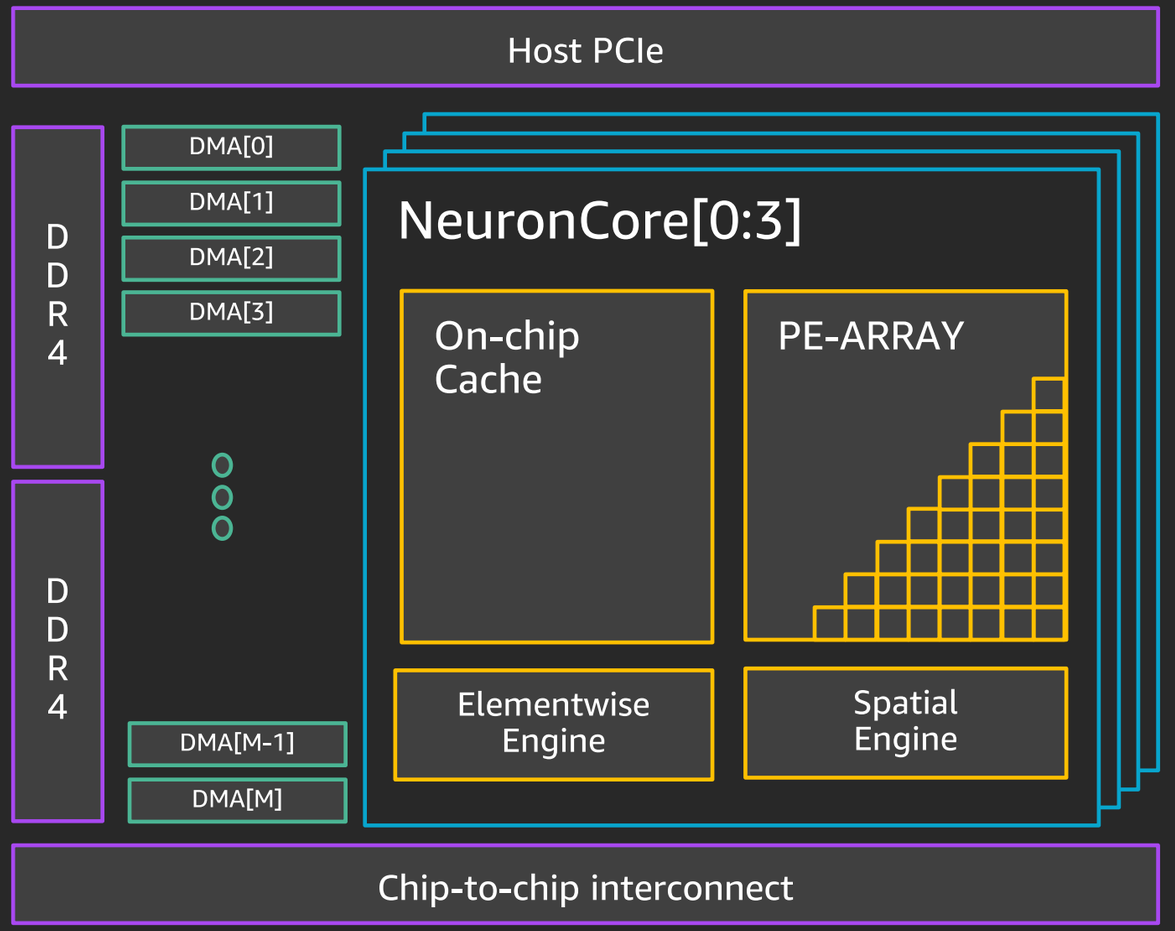
AWS Inferentia Architecture

Why systolic array?

- Data movement is expensive!

Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

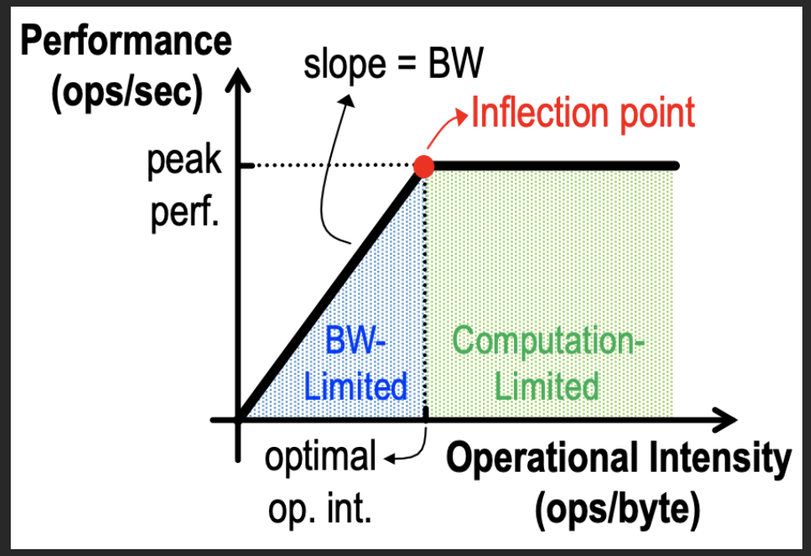




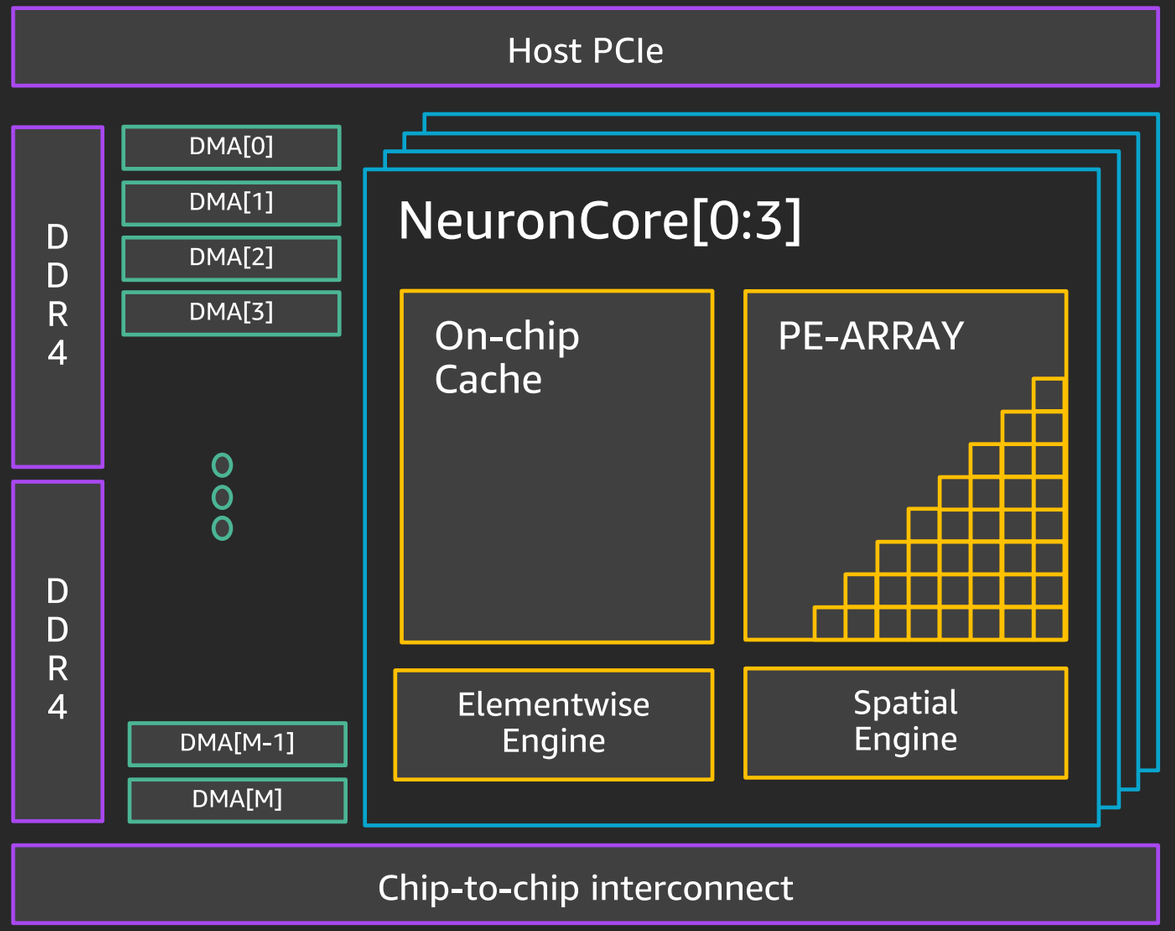
AWS Inferentia Architecture

Why systolic array?

- Data movement is expensive!
  => Data-reuse

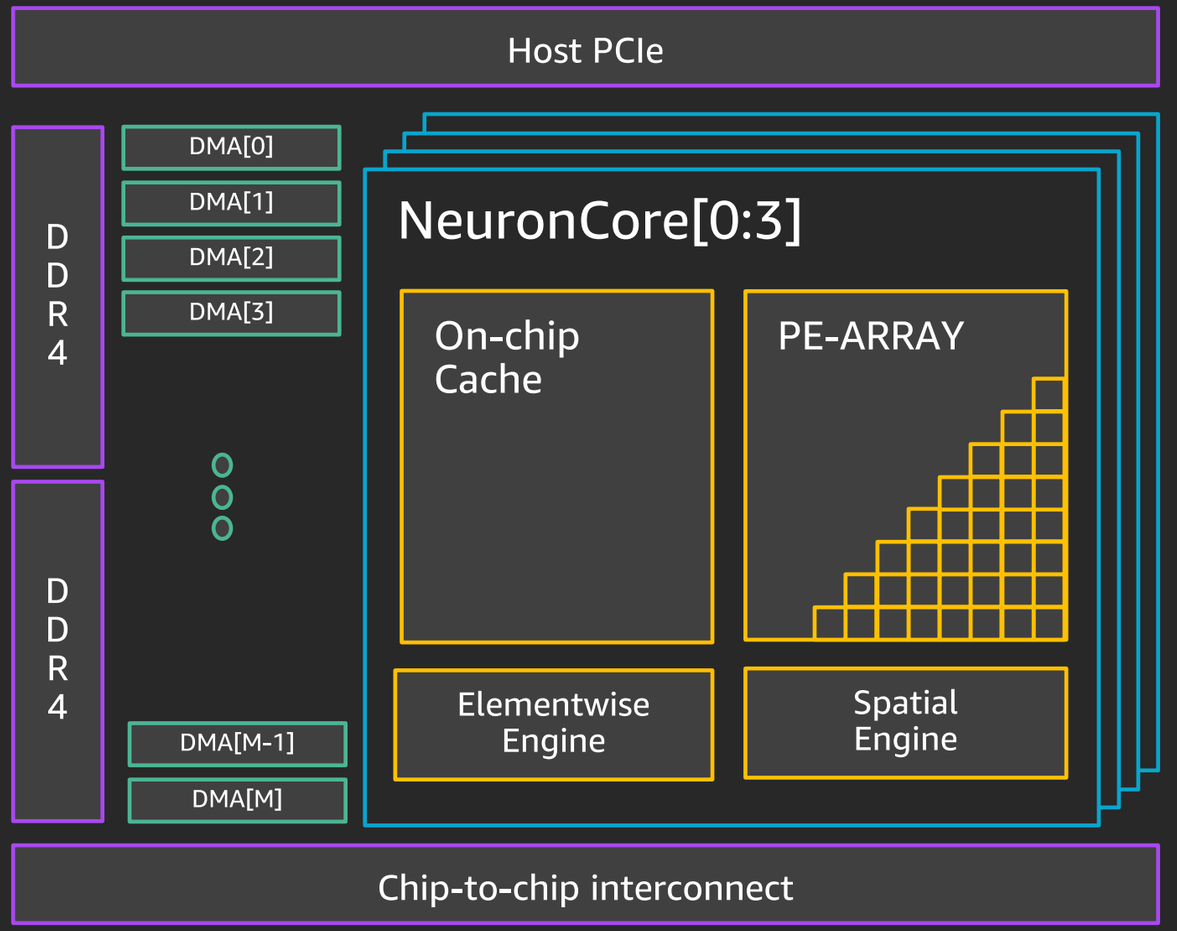


128 TOPS/sec x (2B + 2B + 4B per calc) = 1024 TB/sec!!



AWS Inferentia Architecture

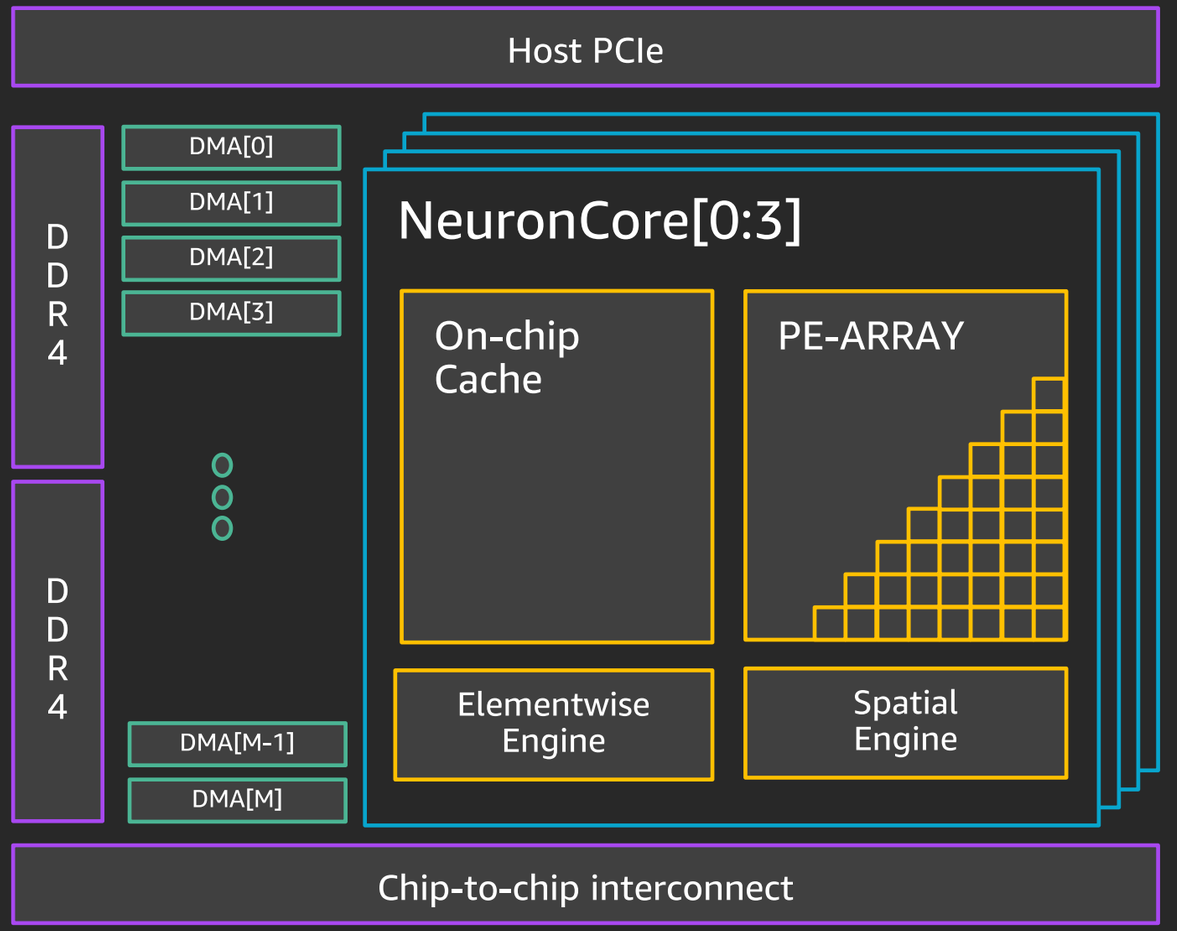
- Why on-chip cache?



AWS Inferentia Architecture

Why on-chip cache?

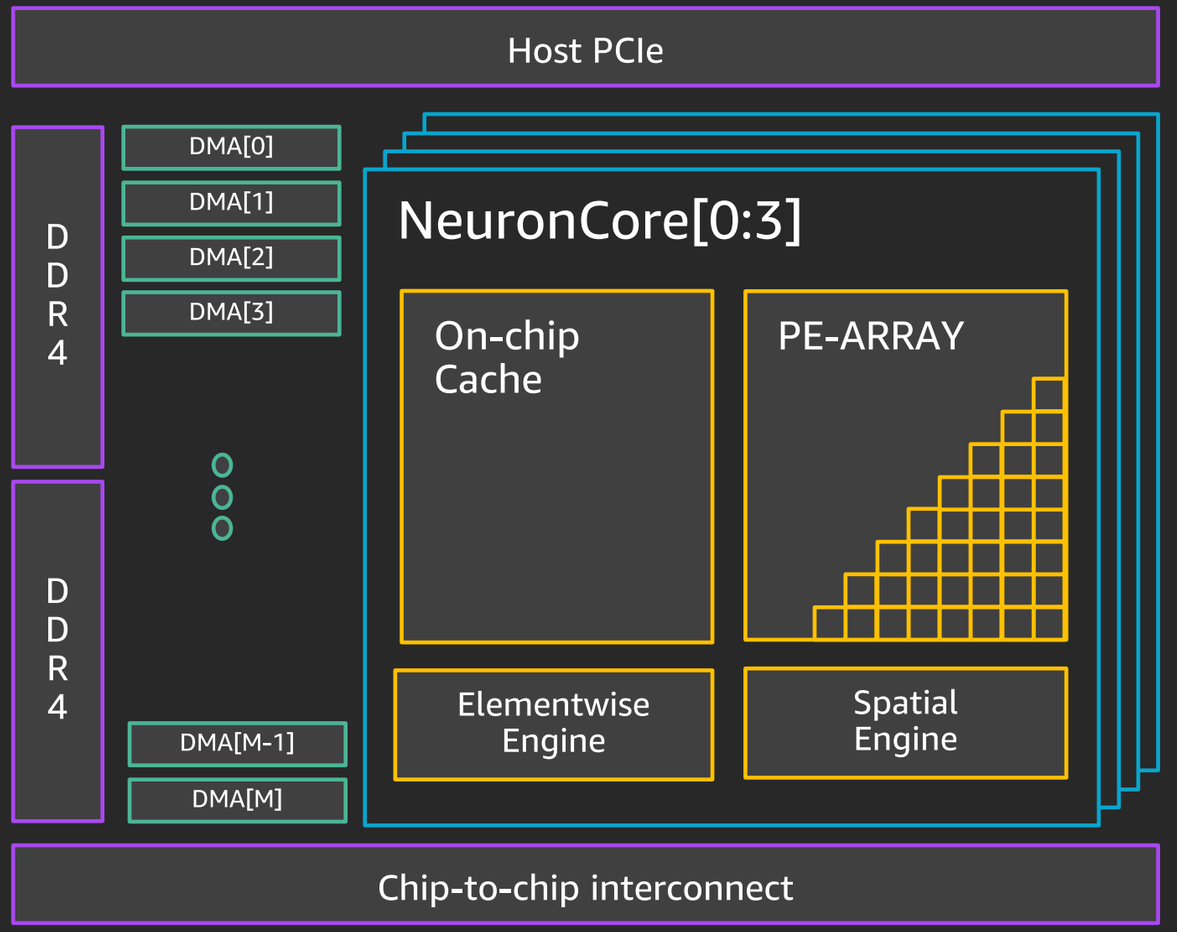
- (actually… it’s a scratch-pad RAM)



AWS Inferentia Architecture

Why on-chip cache?

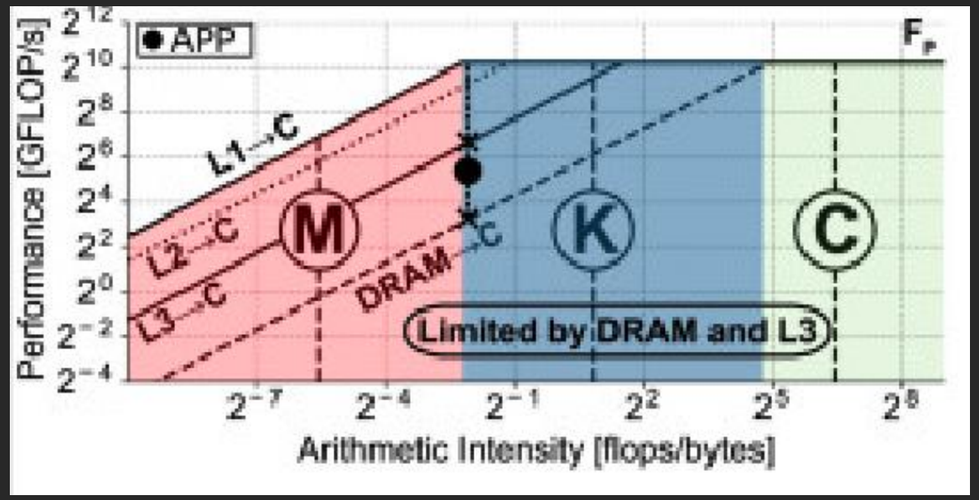
- (actually… it's a scratch-pad RAM)
- local storage for ephemeral state

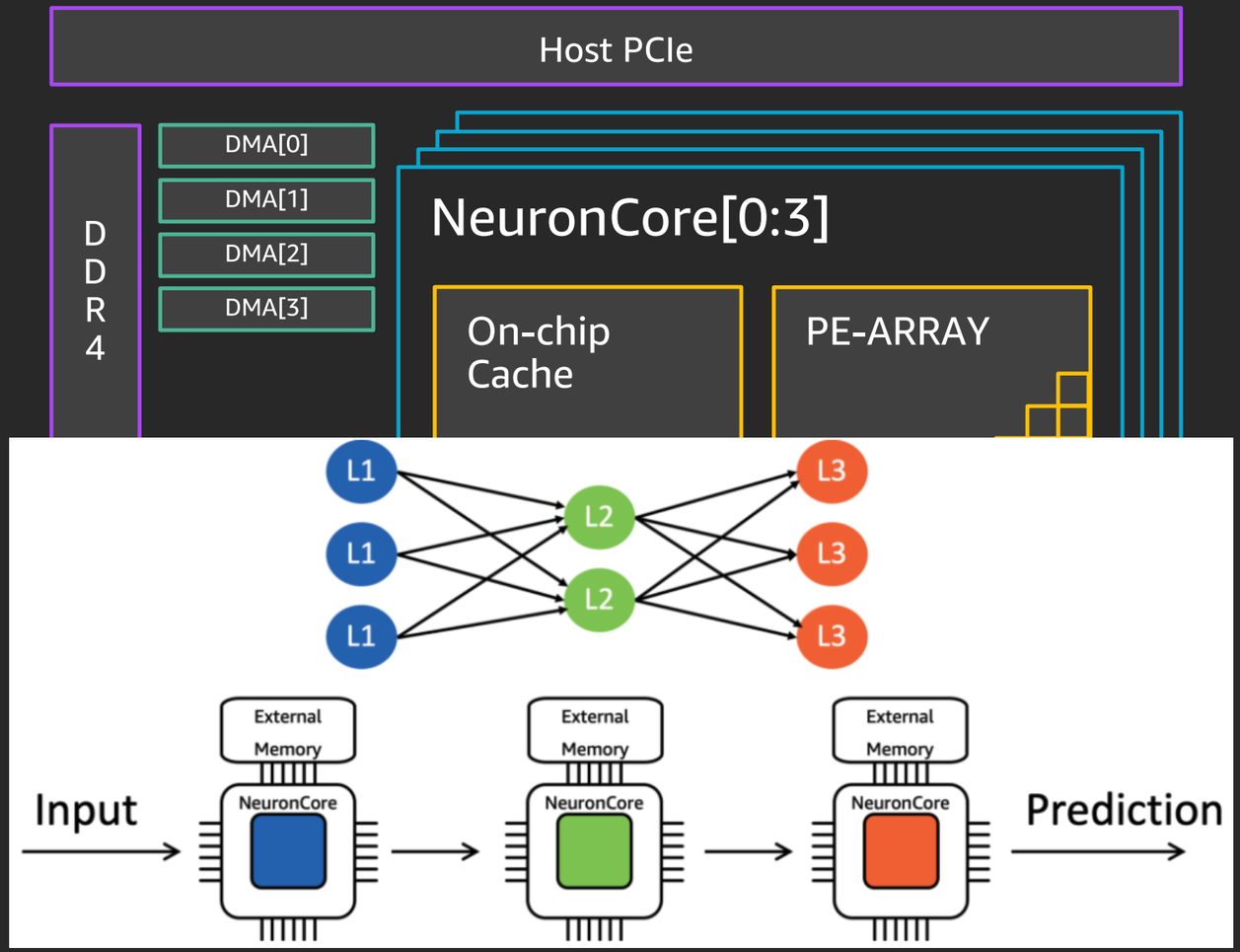


AWS Inferentia Architecture

Why on-chip cache?

- (actually… it's a scratch-pad RAM)
- local storage for ephemeral state
- model caching
  small batches
  (double-roofline model)





AWS Inferentia Architecture

How to map Convs/GEMMs?

WS/IS/OS/…

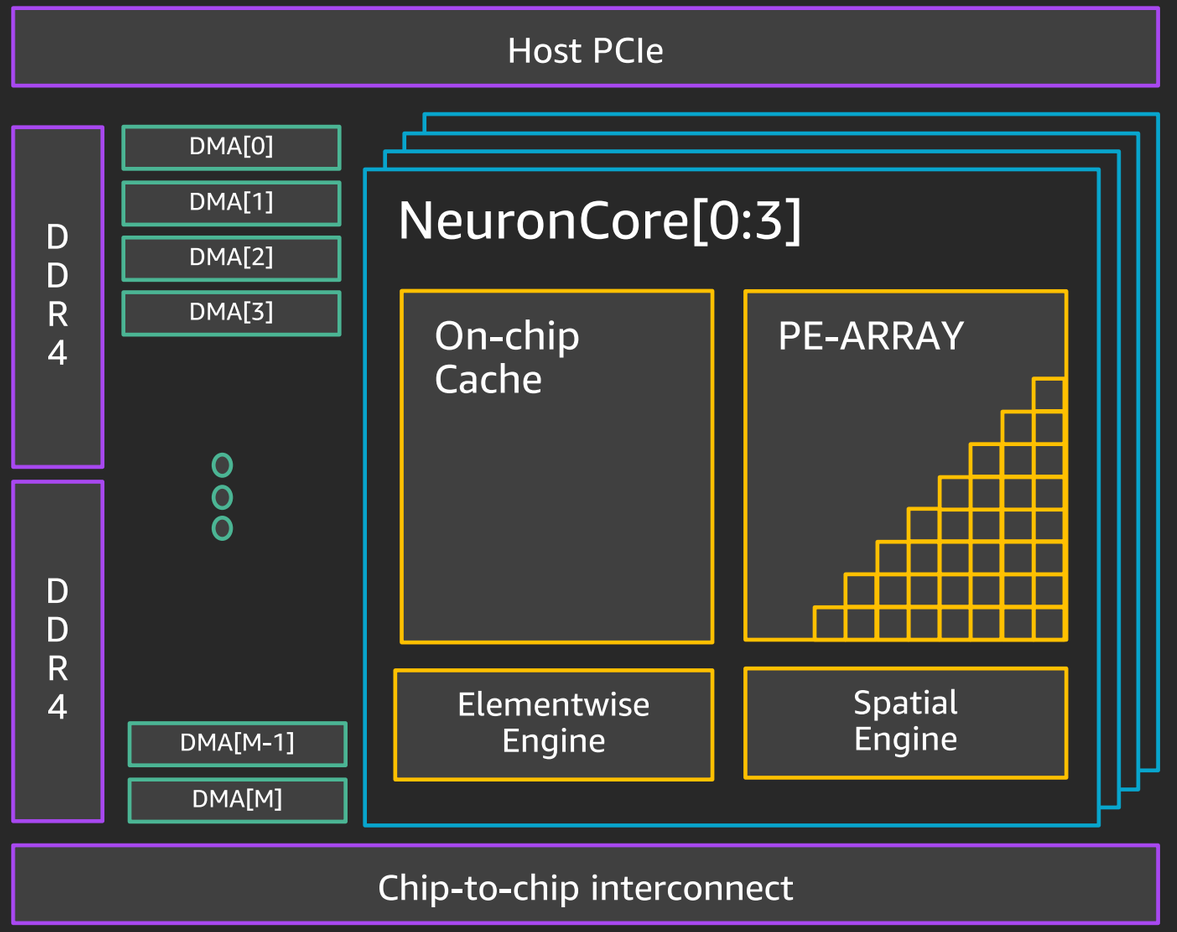
Optimize for programmability

C -> rows (IFMAPs[k])

M -> cols (OFMAPs[k])

IFMAPs[k+1] = OFMAPs[k]

=> Weight Stationary



AWS Neuron

high-performance deep learning inference SDK

Neuron Compiler

Neuron Runtime

Profiling tools

Supports all major frameworks

TensorFlow mxnet PyTorch

github.com/aws/aws-neuron-sdk

AWS Neuron support forum

https://forums.aws.amazon.com/forum.jspa?forumID=355

AWS Neuron quick tour

Compile

Neuron Compiler

(NEFF output)

Deploy

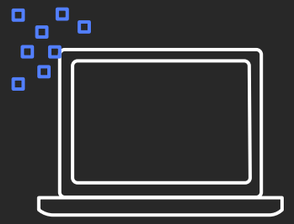
Neuron Runtime
(NRT)

Neuron Binary
(NEFF)

Profile

Neuron Tools

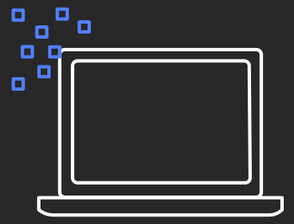
AWS Neuron Compiler highlights



Smart partitioning

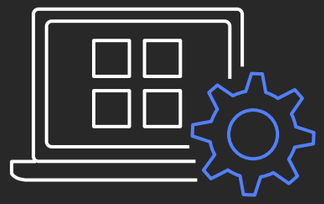
Automatically optimize neural-net compute

AWS Neuron Compiler highlights



Smart partitioning

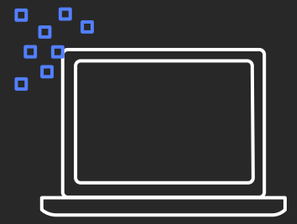
Automatically optimize neural-net compute



FP32 auto casting

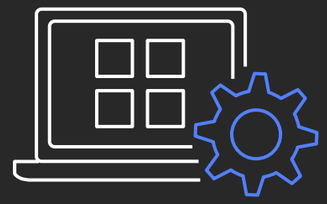
Ingest FP32 trained models, and Neuron auto casts to BF16

AWS Neuron Compiler highlights



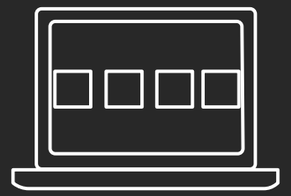
Smart partitioning

Automatically optimize neural-net compute



FP32 auto casting

Ingest FP32 trained models, and Neuron auto casts to BF16

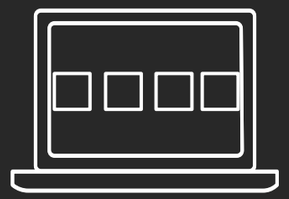


NeuronCore pipeline

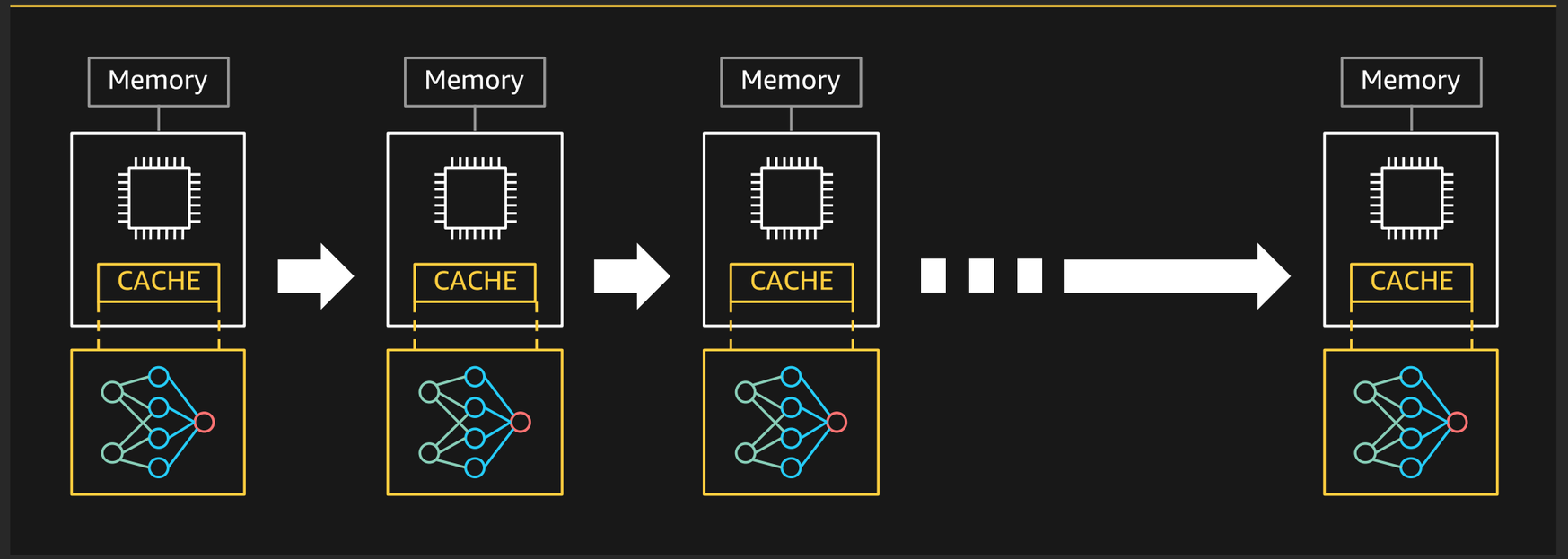
Super low-latency full bandwidth

NeuronCore Pipeline

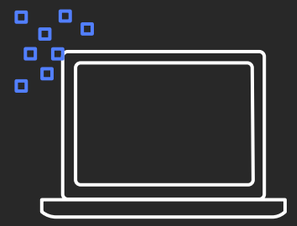
Enabling high-throughput for latency sensitive applications



NeuronCore pipeline

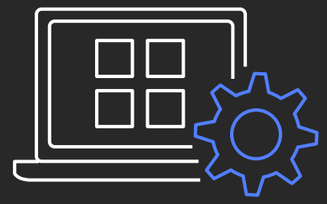


AWS Neuron Compiler highlights



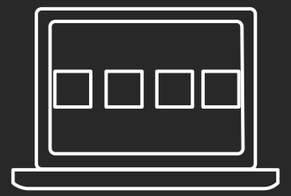
Smart partitioning

Automatically optimize neural-net compute



FP32 auto casting

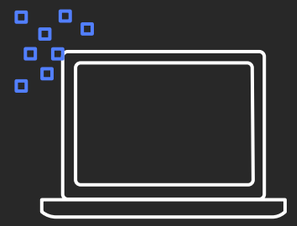
Ingest FP32 trained models, and Neuron auto casts to BF16



NeuronCore pipeline

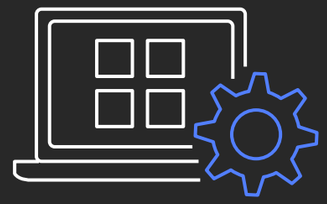
Super low-latency full bandwidth

AWS Neuron Compiler highlights



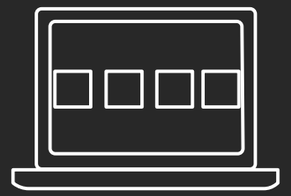
Smart partitioning

Automatically optimize neural-net compute



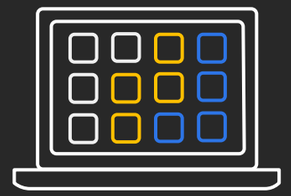
FP32 auto casting

Ingest FP32 trained models, and Neuron auto casts to BF16



NeuronCore pipeline

Super low-latency full bandwidth

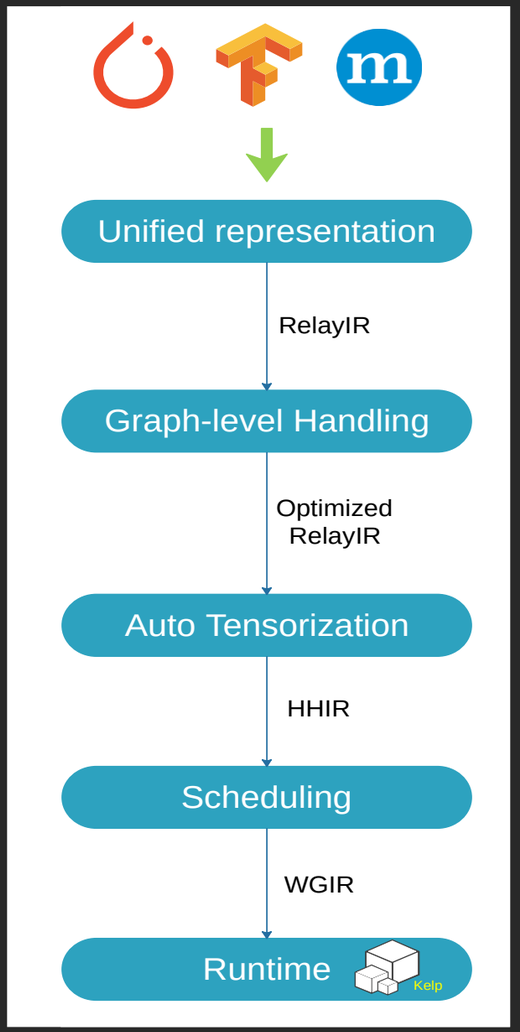


NeuronCore Groups

Concurrently run multiple models

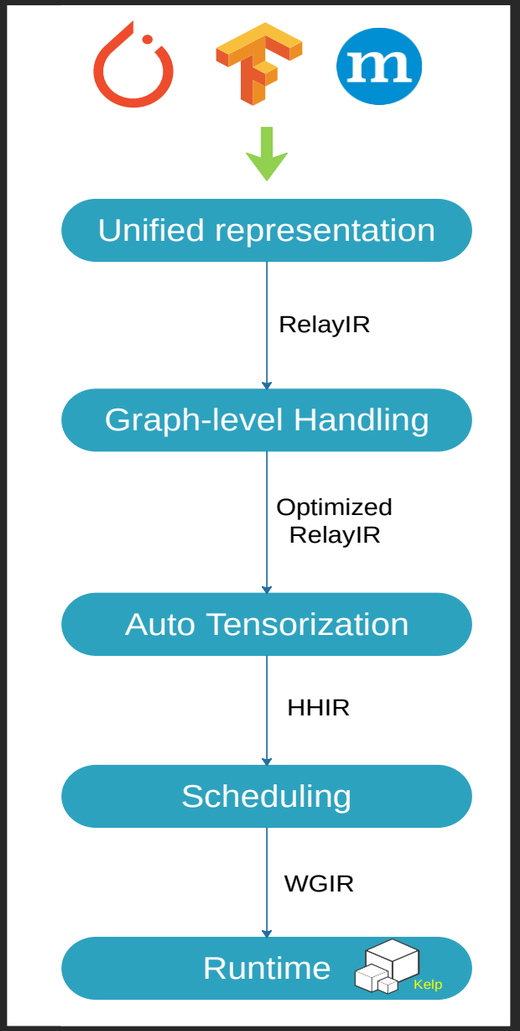
AWS Neuron Compiler Flows

- Use TVM/Relay as frontend and graph optimizer
  - Pull compute definitions from TVM for tensorization
- Optimize the entire neural network as a single acceleration kernel
  - Better manage the scratchpad and avoid being memory bound
- Optimization based on affine loopnests representation
  - Expressive: represent the fusions without predefined computes from TVM, etc.
  - Precise analyses & aggressive optimizations with polyhedral library (ISL)



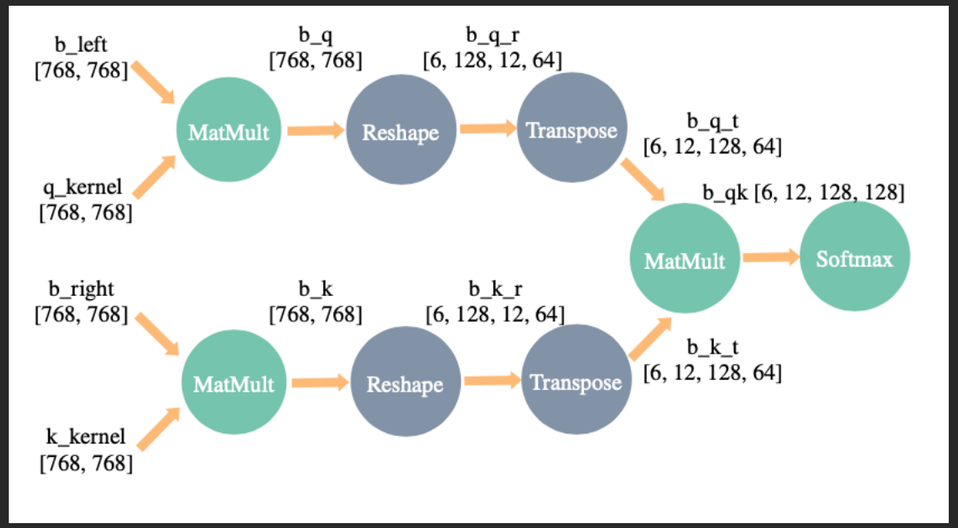
AWS Neuron Compiler: Whole-Graph Tensorization

- Basic idea: tiling and pattern matching to generate tensor intrinsics
- Pre-tensorization fusion - generate bigger loop body
  - Results tensor intrinsics share the same schedule – good temporal locality
- Data rate matching between loopnests - match producer/consumer tile size
  - Enable better fusion later in scheduling – good temporal locality



AWS Neuron Compiler: Whole-Graph Tensorization

- BERT multi-head attention block




```
2
3  # In - b_left size [768, 768]
4  # In - q_kernel size [768, 768]
5  # Out - b_q [768, 768]
6  for (i: range(0, 768, 1))
7   for (j: range(0, 768, 1))
8    for (k: range(0, 768, 1)) {
9     %s_b = affine_load(b_left[i, k])
10    %s_q = affine_load(q_kernel[k, j])
11    %s_bq = tensor_contract_mac(
12              %s_b, %s_q,
13              contract={k})
14   }
15   b_q[i, j] = affine_store(%s_bq)
16
17
18 # Out - b_q_r [6, 128, 12, 64]
19 for (i: range(0, 6, 1))
20  for (j: range(0, 128, 1))
21   for (k: range(0, 12, 1))
22    for (l: range(0, 64, 1)) {
23      %r = affine_load(b_q[128 * i + j,64 * k + l])
24      b_q_r[i, j, k, l] = affine_store(%r)
25    }
```

AWS Neuron Compiler: Whole-Graph Tensorization

Hardware-agnostic optimization

Operator fusion, dead code elimination, constant folding, memory copy elimination

```
# In - b_left size [768, 768]
# In - q_kernel size [768, 768]
# Out - b_q [768, 768]
for (i: range(0, 768, 1))
 for (j: range(0, 768, 1))
  for (k: range(0, 768, 1)) {
   %s_b = affine_load(b_left[i, k])
   %s_q = affine_load(q_kernel[k, j])
   %s_bq = tensor_contract_mac(
            %s_b, %s_q,
            contract={k})
  }
  b_q[i, j] = affine_store(%s_bq)


# Out - b_q_r [6, 128, 12, 64]
for (i: range(0, 6, 1))
 for (j: range(0, 128, 1))
  for (k: range(0, 12, 1))
   for (l: range(0, 64, 1)) {
     %r = affine_load(b_q[128 * i + j,64 * k + l])
     b_q_r[i, j, k, l] = affine_store(%r)
   }
```

```
# Out - b_q_r [6,128,12,64]
for (i_0: range(0, 6, 1))
 for (i_1: range(0, 128, 1))
  for (j_0: range(0, 12, 1))
   for (j_1: range(0, 64, 1))
    for (k: range(0, 768, 1))
     %s_b = affine_load(b_left[i_0, i_1, k])
     %s_q = affine_load(q_kernel[k, j_0, j_1])
     %s_bq = tensor_contract_mac(
              %s_b, %s_q,
              contract={k},
    }
    b_q_r[i_0, i_1, j_0, j_1] = affine_store(%s_bq)
```

AWS Neuron Compiler: Whole-Graph Tensorization

Layout inference and transformation

Identify / insert transformation to satisfy hardware constraints.

```
# Out - b_q_r [6,128,12,64]
for (i_0: range(0, 6, 1))
 for (i_1: range(0, 128, 1))
  for (j_0: range(0, 12, 1))
   for (j_1: range(0, 64, 1))
    for (k: range(0, 768, 1))
     %s_b = affine_load(b_left[i_0, i_1, k])
     %s_q = affine_load(q_kernel[k, j_0, j_1])
     %s_bq = tensor_contract_mac(
             %s_b, %s_q,
             contract={k},
    }
    b_q_r[i_0, i_1, j_0, j_1] = affine_store(%s_bq)
```

```

# Transformed input -  b_left [768, 6, 128]
# Transformed output - b_q_r [12, 64, 6, 128]
for (i_0: range(0, 6, 1))
 for (i_1: range(0, 128, 1))
  for (j_0: range(0, 12, 1))
   for (j_1: range(0, 64, 1))
    for (k: range(0, 768, 1))
      # partition_axes = k is inferred for both input
      %s_b = affine_load(b_left[k, i_0, i_1])
      %s_q = affine_load(q_kernel[k, j_0, j_1])
      %s_bq = tensor_contract_mac(
              %s_b, %s_q,
              contract={k},
    }
    # partition_axis = j_1 is inferred for output
    b_q_r[j_0, j_1, i_0, i_1] = affine_store(%s_bq)
```

AWS Neuron Compiler: Whole-Graph Tensorization

Inferentia ISA loopnest Identification

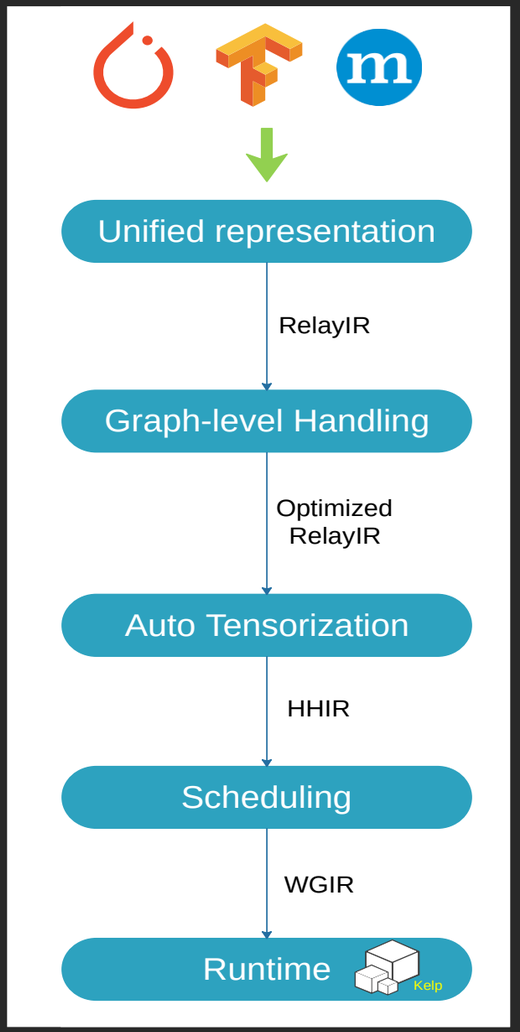
Perform loop interchange and tiling to re-organizing loops

```
# Transformed input -  b_left [768, 6, 128]
# Transformed output - b_q_r [12, 64, 6, 128]
for (i_0: range(0, 6, 1))
 for (i_1: range(0, 128, 1))
  for (j_0: range(0, 12, 1))
   for (j_1: range(0, 64, 1))
    for (k: range(0, 768, 1))
      # partition_axes = k is inferred for both input
      %s_b = affine_load(b_left[k, i_0, i_1])
      %s_q = affine_load(q_kernel[k, j_0, j_1])
      %s_bq = tensor_contract_mac(
              %s_b, %s_q,
              contract={k},
    }
    # partition_axis = j_1 is inferred for output
    b_q_r[j_0, j_1, i_0, i_1] = affine_store(%s_bq)
```

```
# Transformed input - b_left [128, 6, 6, 128]
# Transformed output - b_q_r [64, 12, 6, 128]
for (j_0: range(0, 12, 1))
 for (k_1: range(0, 6, 1))
  matmul_128x64x512{ # Marking the inner-most loop
   for (j_1: range(0, 64, 1))
    for (i_0: range(0, 6, 1))
     for (i_1: range(0, 128, 1))
      for (k_0: range(0, 128, 1))
        %s_b = affine_load(b_left[k_0,k_1,i_0,i_1])
        %s_q = affine_load(q_kernel[k_0,k_1,j_0,j_1]
        %s_bq = tensor_contract_mac(
                %s_b, %s_q,
                contract={k_0},
      }
      %acc = reduce_add(%s_bq, reduce={k_1})
      b_q_r[j_1, j_0, i_0, i_1] = affine_store(%acc)
```

AWS Neuron Compiler: Scheduling & Allocation

- Basic idea:
  - Capture schedule with loopnest structure
  - Manage/allocate the scratchpad with compiler, by scanning the whole graph
  - Modify schedule based on allocator feedback and iterate
- Modify schedule with loop transformations:
  - Overflow - Improve temporal locality with fusion/de-LICM (load weight twice)
  - Unused space - Infer more parallelism: tile/fission then infer parallel loops



Alexa usage of Inf1 instances

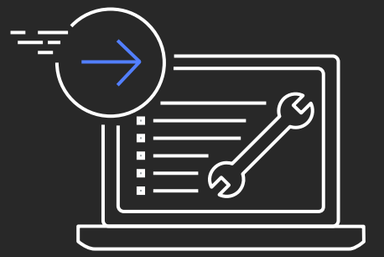
Neural text-to-speech challenges

- Low-latency requirement for dialog system
- High throughput requirement implied by streaming of the output speech
- Context generation is a sequence-to-sequence auto-regressive model
- Memory-bandwidth bound inference
- High temporal density of speech production model resulting with 90GFLOPs to generate 1s of output is Compute-bound inference
- Using EC2 GPU instances to meet requirements results in high operational cost

Alexa TTS migration to EC2 Inf1—Ease of integration

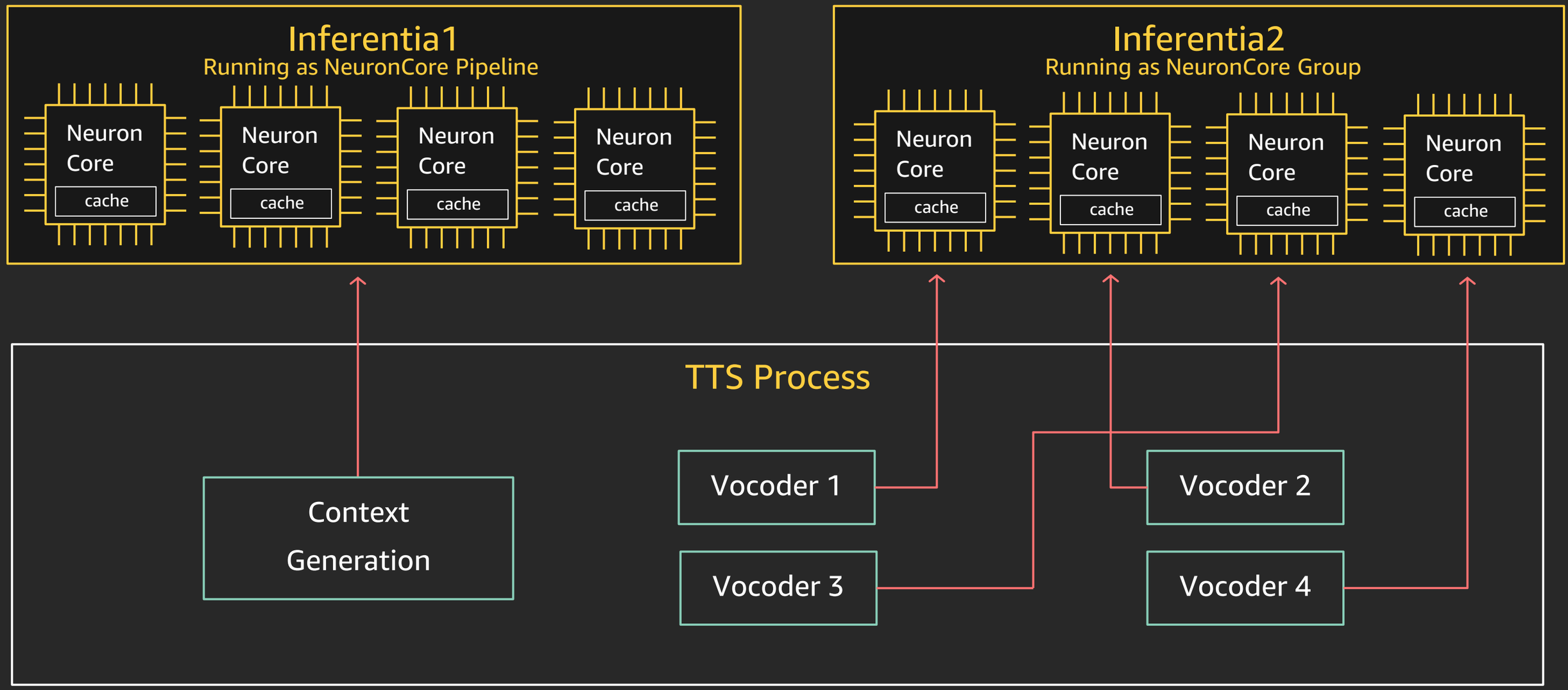
Alexa TTS uses MXNet that is supported natively by AWS Neuron

Support for C and Python APIs



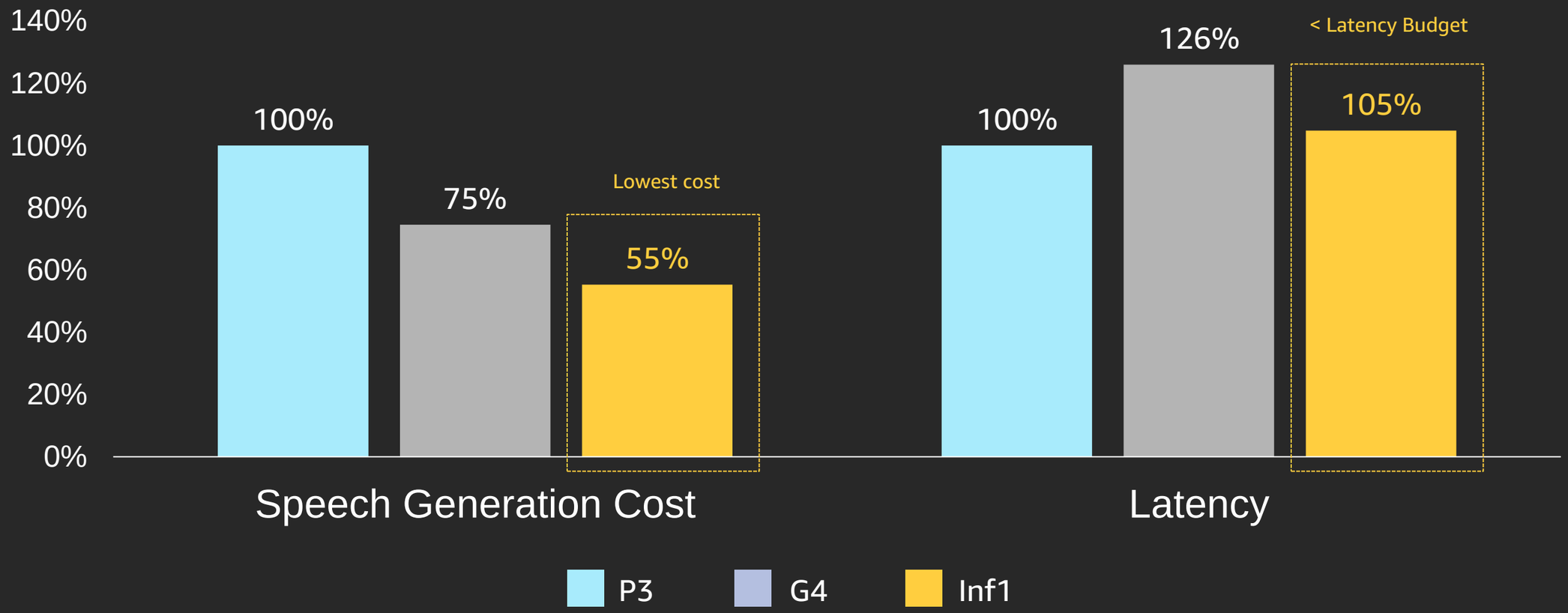
Options to migrate original FP32 models to FP16 or Bfloat16

Alexa TTS migration to EC2 Inf1—Architecture



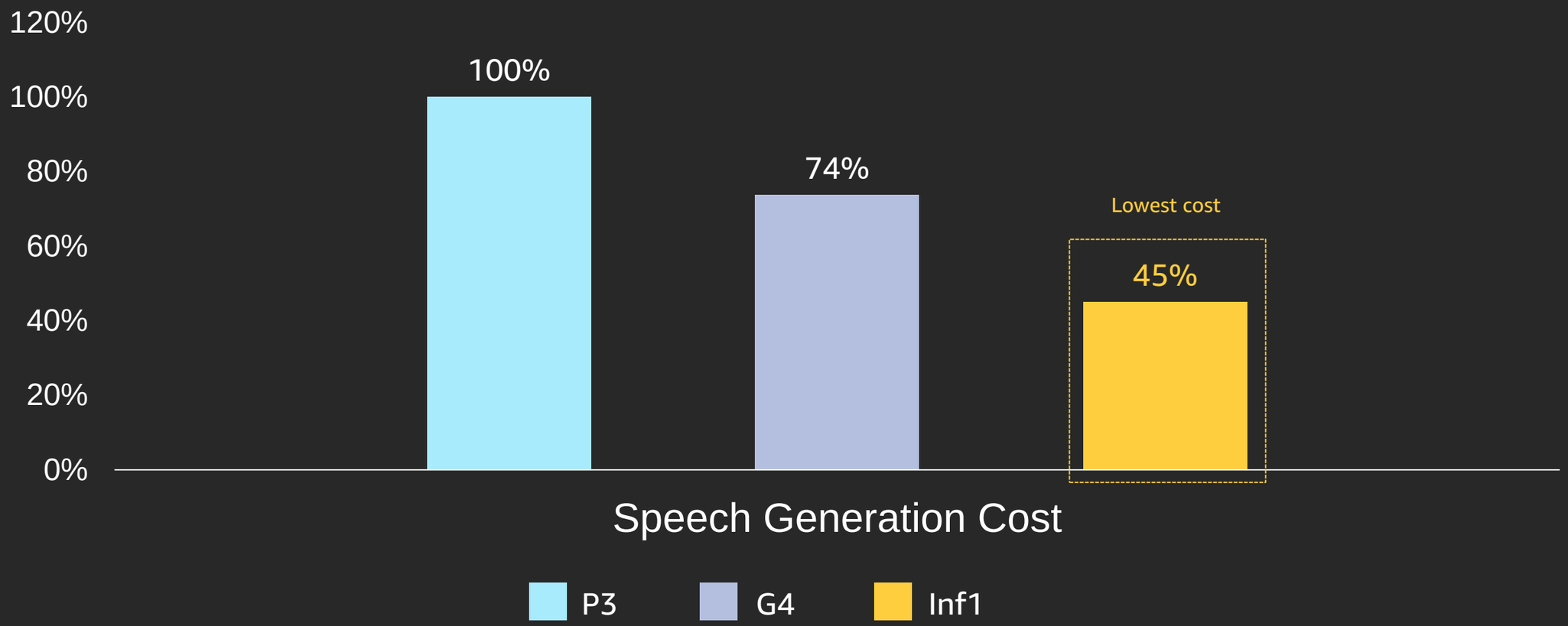
Alexa TTS migration to EC2 Inf1—Gains

Alexa generic traffic is able to get the lowest cost at the same latency budget



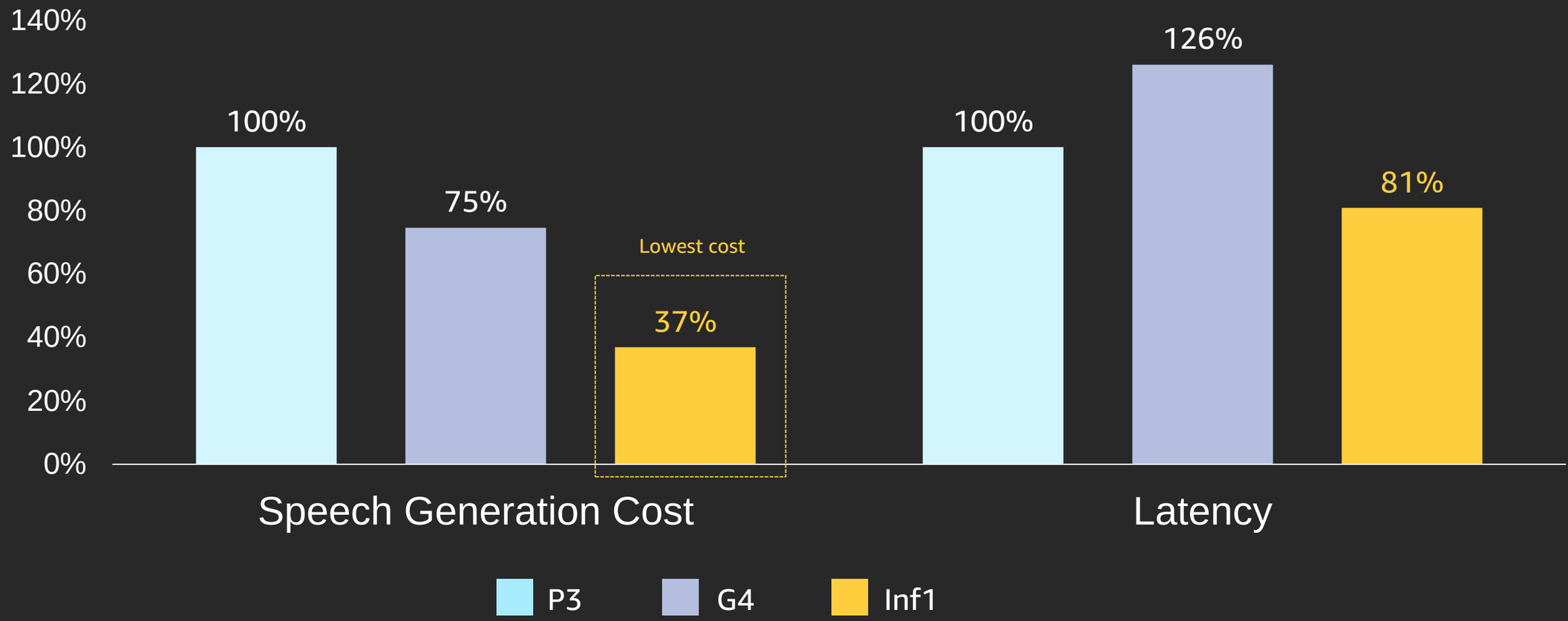
Alexa TTS migration to EC2 Inf1—Long texts

Alexa long text traffic (ex: books, news) has even higher gains



Alexa TTS migration to EC2 Inf1—Projected gains

With software optimizations Alexa can take full advantage of Inf1 for significant gains



Of course we are hiring!

Nitro, Graviton, Inferentia (and few more secret projects we wish we could tell you about)