NOKIA

TensorCore Optimized DNN for Efficient Low Latency Inference for 5G Networks

Tero Rissa / Andrew Baldwin

GTC 2019

Goals

Results

Method

Goals

5G radio resource management L1/L2 tasks can typically benefit from use of relatively simple Multilayer Perceptron (MLP) Deep Neural Network (DNN) models

Parameters: 3.3M

Ops/Inference: 6.7M + 4K x tanh

Latency: Needs result in 50 µs to integrate the results with a 5G protocol cycle

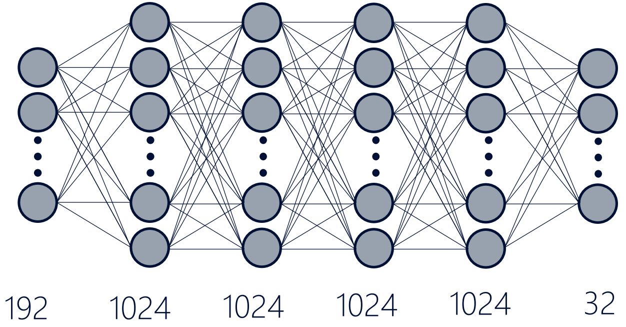
Throughput: Still need high throughput to serve maximum number of clients and reduce computation cost

Batch size: Smaller is better as combining data from multiple clients into larger batches can increase latency

Input: Data arriving over backplane to CPU DDR buffer

```
import keras
from keras.layers import Input, Dense
from keras.layers import Model

inputs = Input(shape=(192,))
x = Dense(1024, activation=”tanh”)(inputs)
x = Dense(1024, activation=”tanh”)(inputs)
x = Dense(1024, activation=”tanh”)(inputs)
x = Dense(1024, activation=”tanh”)(inputs)
predictions = Dense(32)(x)
model = Model(inputs=inputs, outputs=predictions)
```



Goals

Results

Method

Results

Keras - CPU single-core

TensorFlow 1.12 backend

Xeon 6130 @ 3.5GHz (turbo)

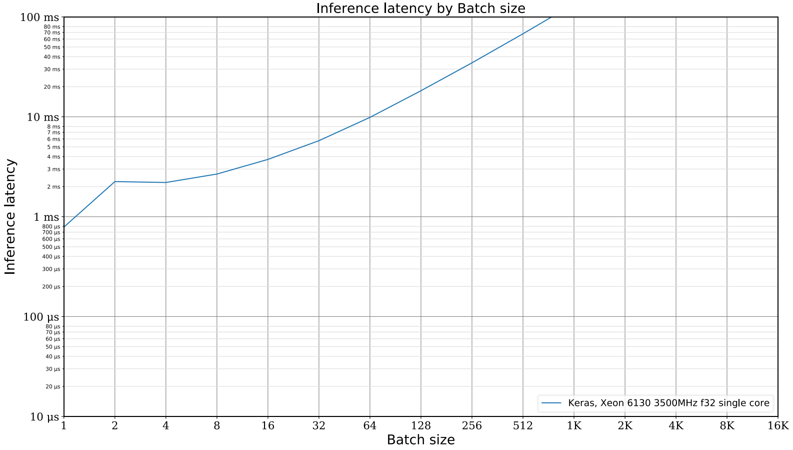
(Use "taskset 1" to constrain to single core)

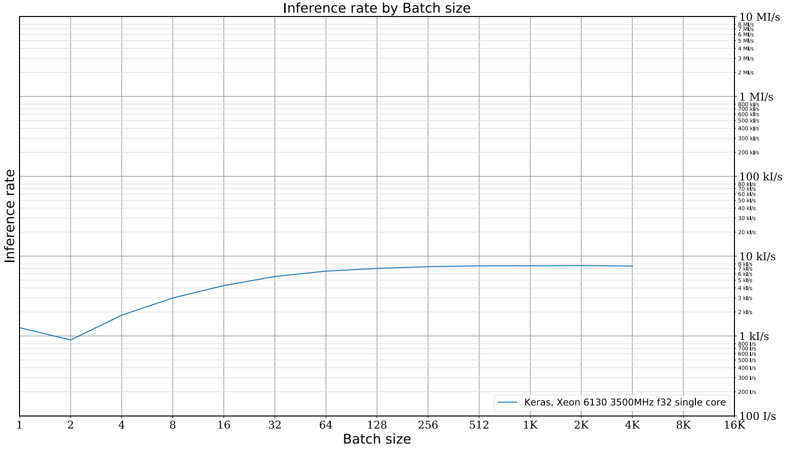
Best latency: Batch 1, 780µs, 1.3k Inf/s

Plateau: Batch 256, 35000µs, 7.5k Inf/s

Shortest latency is 16x target

17% efficiency at best latency compared to plateau





Results

Keras - CPU multi-core

TensorFlow 1.12 backend

2 x Xeon 6130 @ 2.4GHz 32 core 64 thread

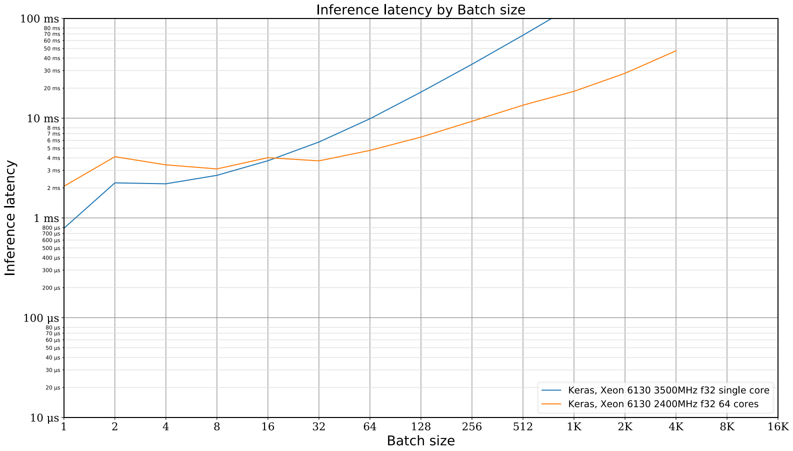
Best latency: Batch 1, 1900µs, 0.5k Inf/s

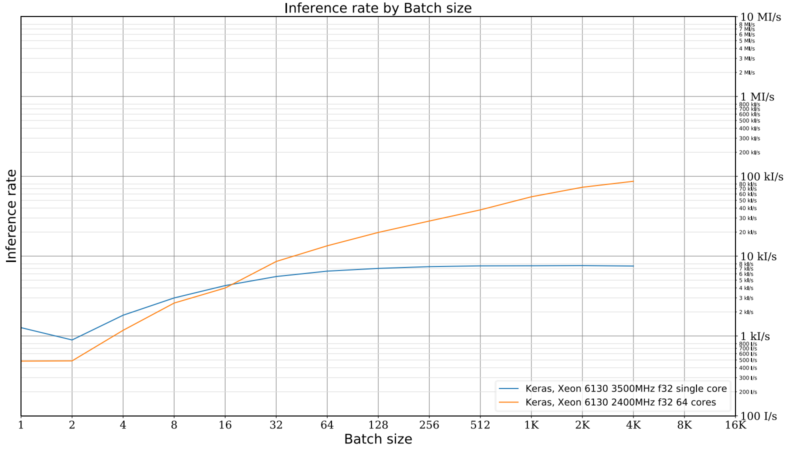
Best rate: Batch 2k, 28000µs, 80k Inf/s

Shortest latency 2x worse than single-core

Latency better than single-core above batch 16

Rate not scaling efficiently vs single (32 vs 1)





Results

Keras – GPU NVIDIA Tesla V100 PCIe

TensorFlow 1.12 backend

To allow TensorCore use:

```
keras.backend.set_floatx(“float16”)
```

Best latency: Batch 1, 670µs, 1.5k Inf/s

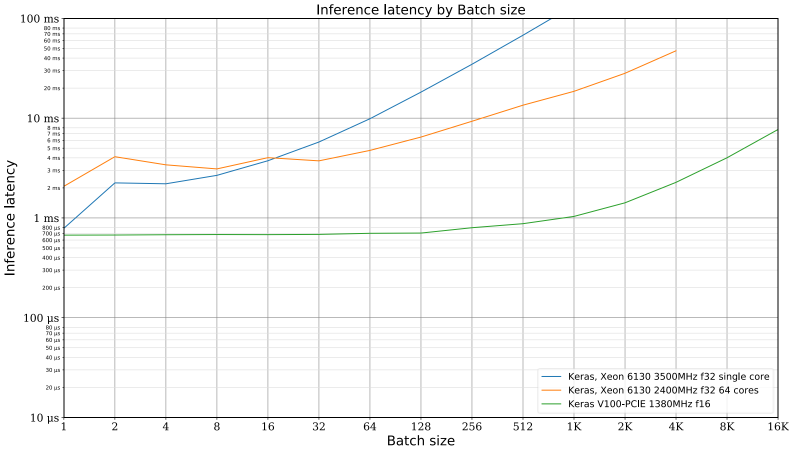
Plateau: Batch 8k, 4000µs, 2100k Inf/s

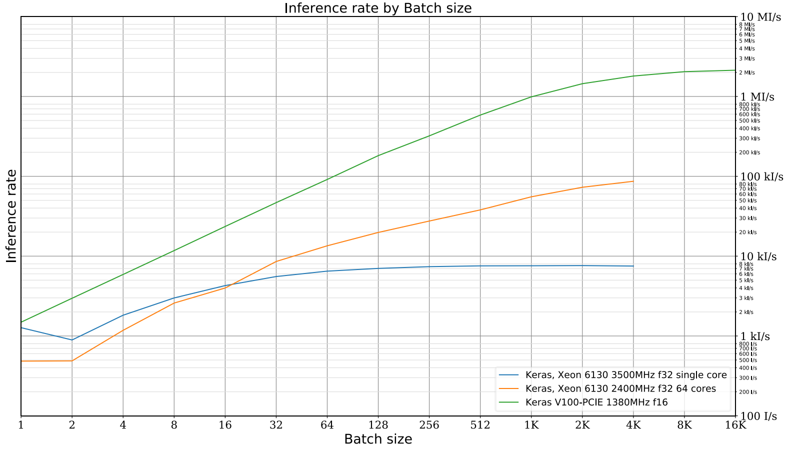
No latency improvement on Batch 1

Shortest latency is 13x target

Up to 26x rate of CPU multi-core

Latency similar between Batch 1-128





Results

TensorFlow 1.12 – GPU NVIDIA V100

Keras model converted to frozen TensorFlow graph

Aim is to see if Keras is limiting the performance

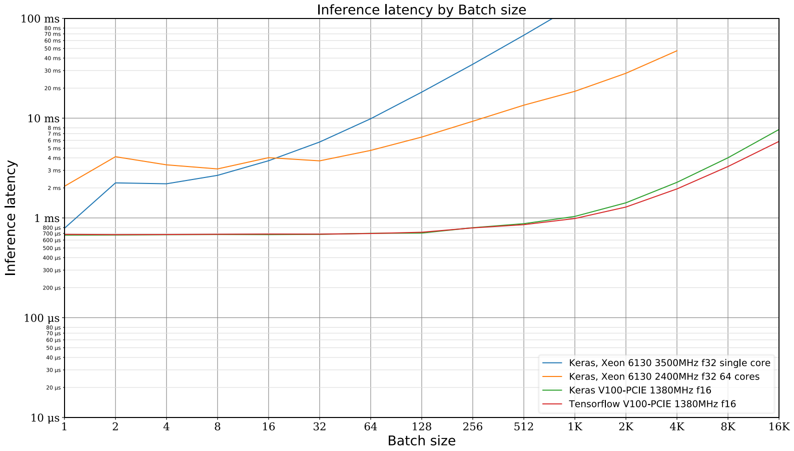
Best latency: Batch 1, 670µs, 1.5k Inf/s

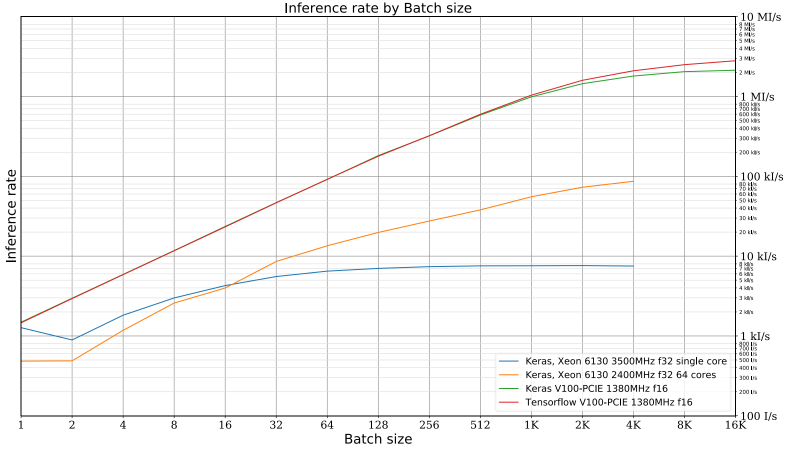
Plateau: Batch 16k, 5900µs, 2800k Inf/s

Shortest latency same as Keras on TF

30% improvement on plateau rate compared to Keras

Uncertain if or at which batch sizes TensorCores were used





Results

TensorRT v5 – GPU NVIDIA V100

TF graph converted to TRT

f16 inference enabled

Latency measurement includes on a single CUDA stream:

- Async copy from pagelocked CPU memory buffer to input device buffer
- TensorRT API Inference from input device buffer to output device buffer
- Async copy from output device buffer to pagelocked CPU memory buffer
- Stream synchronize call

Rate is measured without async copies but with sync

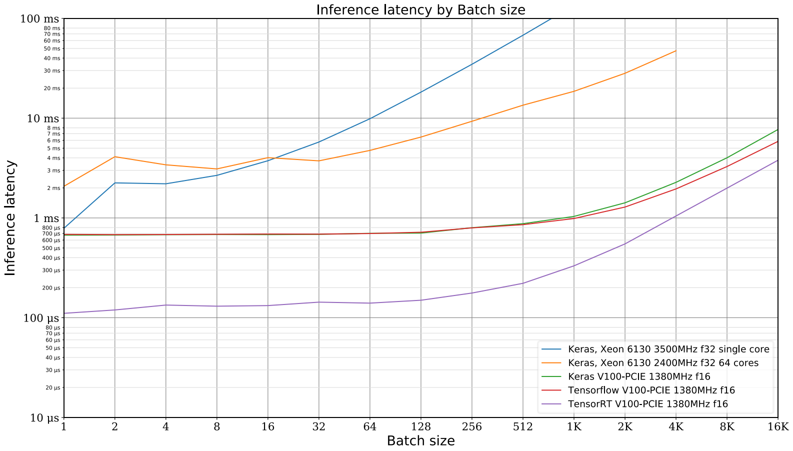
Best latency: Batch 1, 110µs, 9k Inf/s

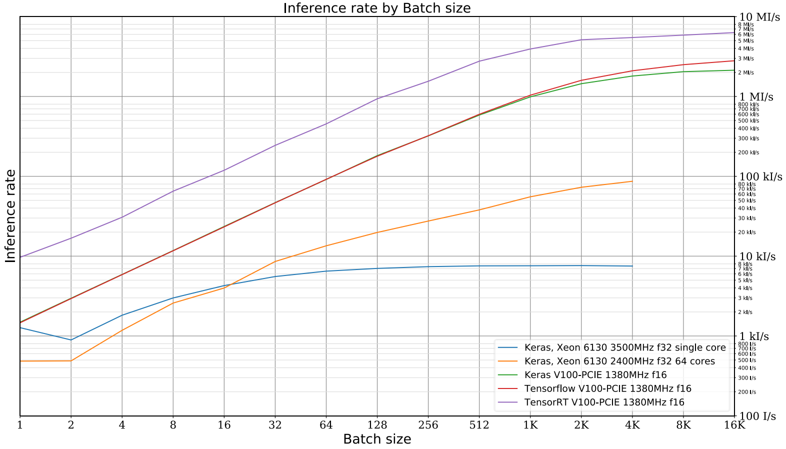
Best rate: Batch 16k, 2600µs, 6300k Inf/s

6x better latency & rate vs TensorFlow

Shortest latency 2x target

Optimal rate/latency at Batch 256, 185µs, 2800k Inf/s





Results – lowest latency

Instarence – GPU NVIDIA V100

Nokia low-latency GPU inference system

Keras model as input

Latency and Rate measured in similar way as for TRT

Parameters optimised for **lowest latency** in real use case (host to host, no pipeline)

Best latency: Batch 8, 34.8µs, 222k Inf/s

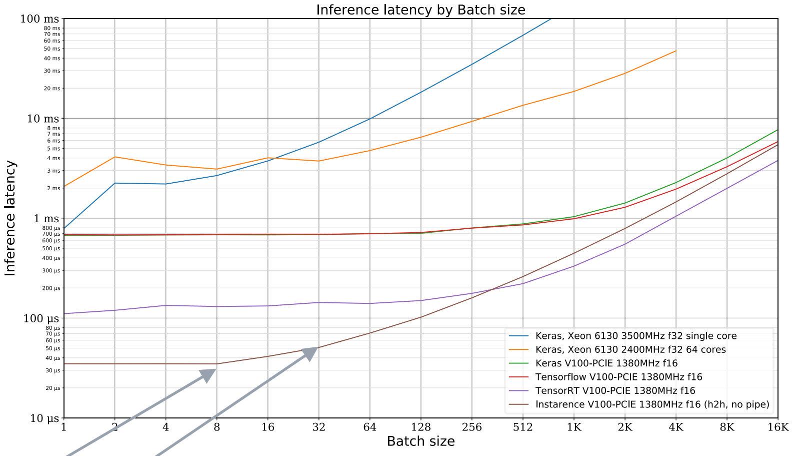
Best rate under *50µs*: Batch 32, 615k Inf/s

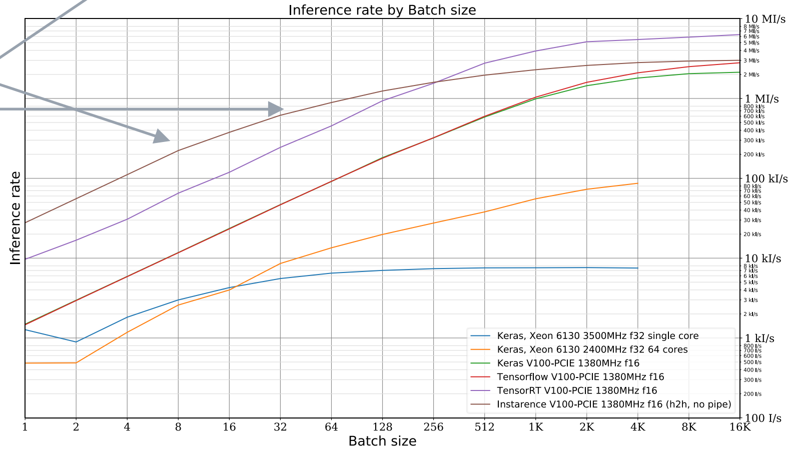
Plateau: Batch 4k, 1460µs, 2800k Inf/s

3x shorter latency than TensorRT

Latency 30% shorter than target (0.7x)

Best latency and rate up to Batch 256





Results – max throughput

Instarence – GPU NVIDIA V100

Nokia low-latency GPU inference system

Keras model as input

Latency and Rate measured in similar way as for TRT

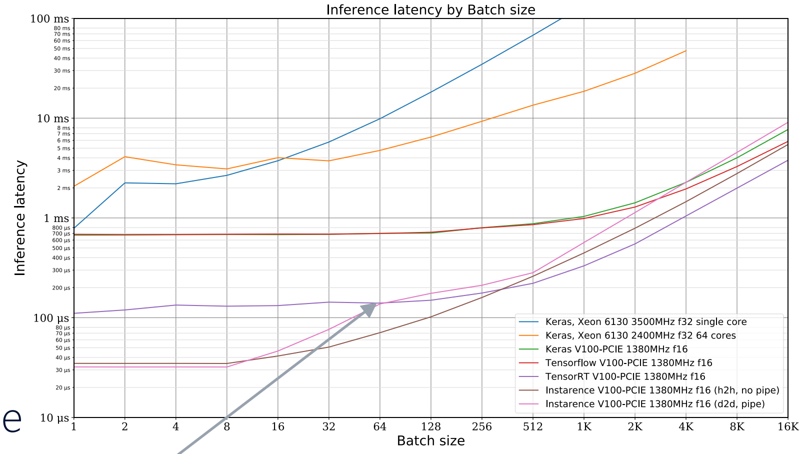
Parameters optimised for **best throughput** (device to device, full pipeline)

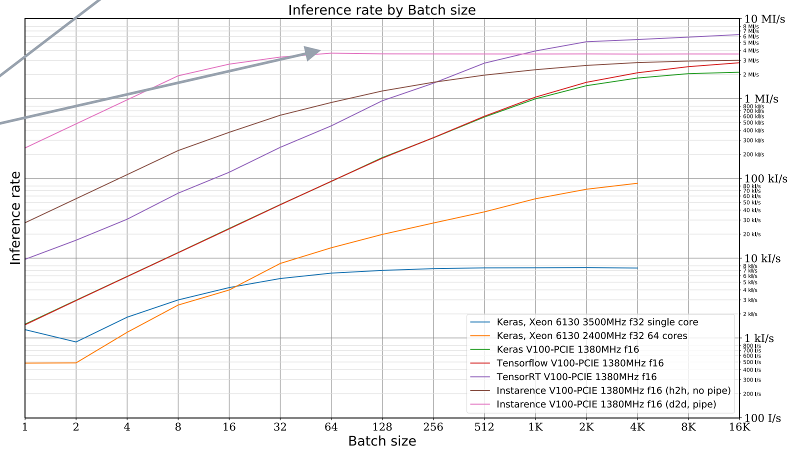
Best latency: Batch 8, 32.1µs, 1900k Inf/s

Plateau: Batch 64, 137.3µs, 3700k Inf/s

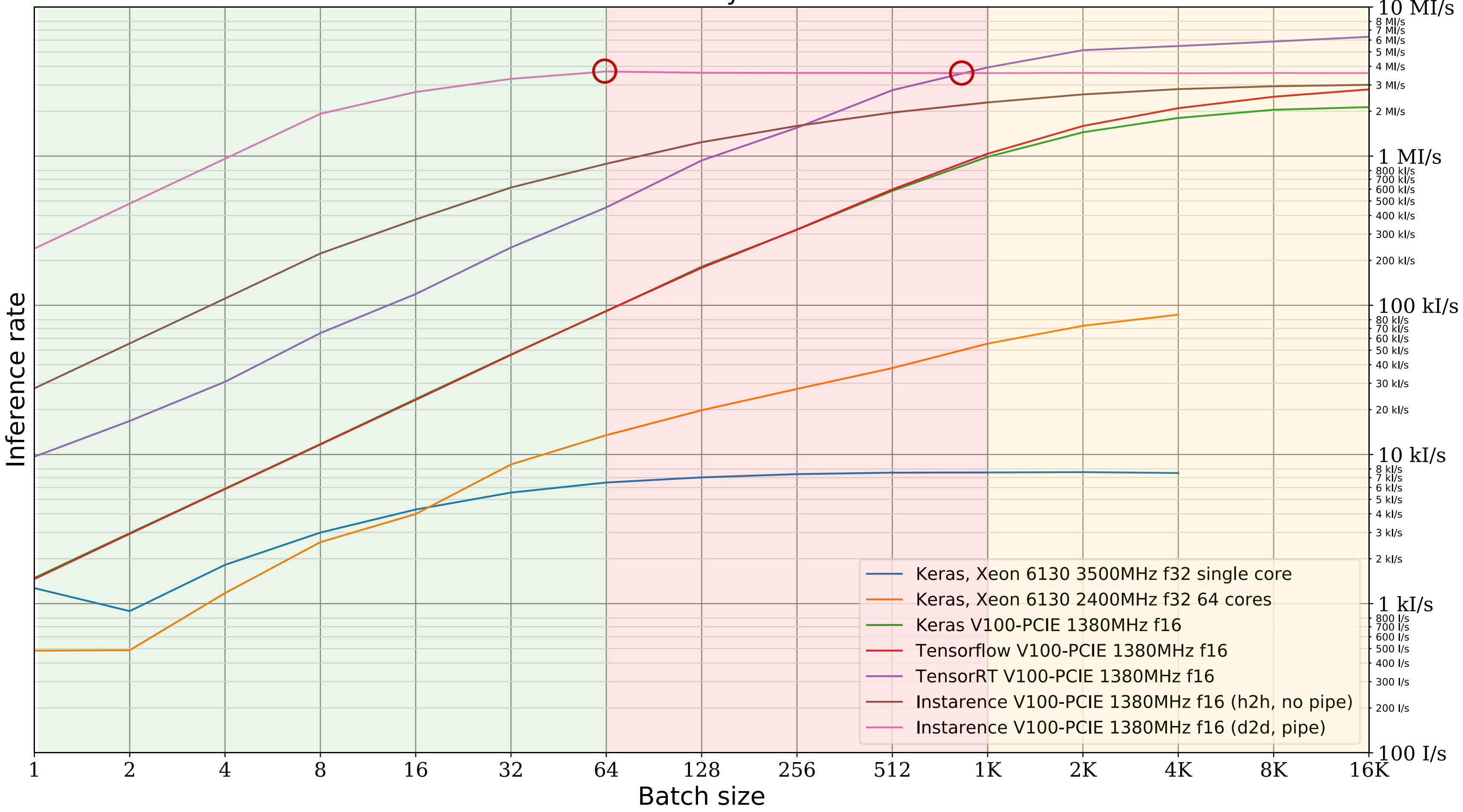
28x rate vs TRT at batch 8

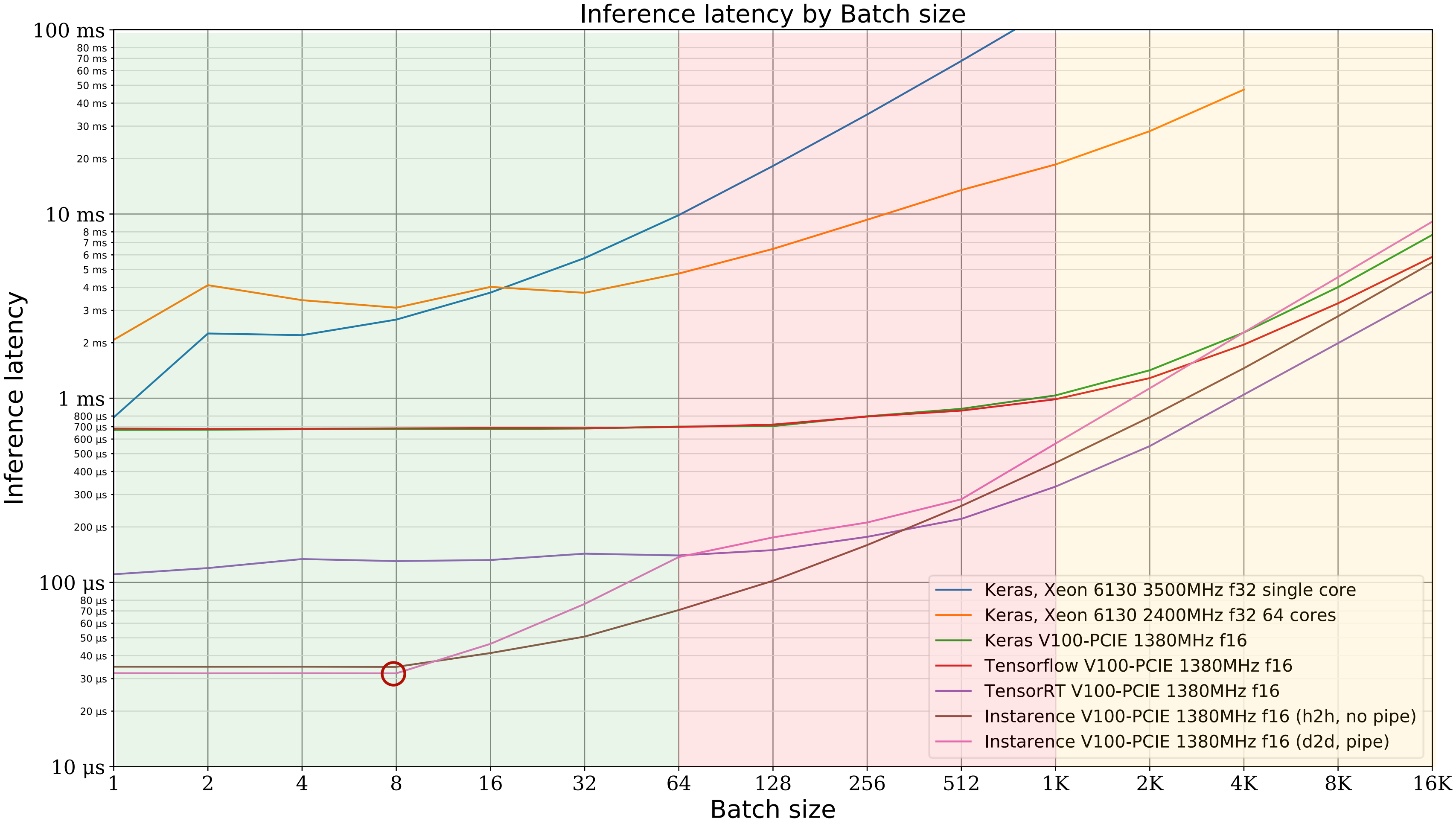
8.5x rate vs low-latency mode at batch 8





Inference rate by Batch size




Inference latency by Batch size
Inference latency
100 ms
80 ms
70 ms
60 ms
50 ms
40 ms
30 ms
20 ms
10 ms
8 ms
7 ms
6 ms
5 ms
4 ms
3 ms
2 ms
1 ms
800 µs
700 µs
600 µs
500 µs
400 µs
300 µs
200 µs
100 µs
80 µs
70 µs
60 µs
50 µs
40 µs
30 µs
20 µs
10 µs
1
2
4
8
16
32
64
128
256
512
1K
2K
4K
8K
16K
Batch size
Keras, Xeon 6130 3500MHz f32 single core
Keras, Xeon 6130 2400MHz f32 64 cores
Keras V100-PCIE 1380MHz f16
Tensorflow V100-PCIE 1380MHz f16
TensorRT V100-PCIE 1380MHz f16
Instarence V100-PCIE 1380MHz f16 (h2h, no pipe)
Instarence V100-PCIE 1380MHz f16 (d2d, pipe)

Goals

Results

Method

MLP DNN per-layer Operations

$$\begin{array}{ccccccc} Z^{[n]} & = & W^{[n]} & \times & X^{[n-1]} & + & b^{[n]} \\ (n^{[n]}, m) & & (n^{[n]}, n^{[n-1]}) & & (n^{[n-1]}, \mathrm{m}) & & (n^{[n]}, 1) \end{array}$$

$$\begin{array}{cl} A^{[n]} & = \tanh(Z^{[n]}) \\ (n^{[n]}, m) & \end{array}$$

Apply **weights** (matrix multiply):

Weight matrix: [Nodes (outputs) x Inputs (nodes in previous layer)]

x

Input matrix: [Inputs x Batch size]

→ Layer output Matrix: [Nodes x Batch size]

Add **bias** (element-wise):

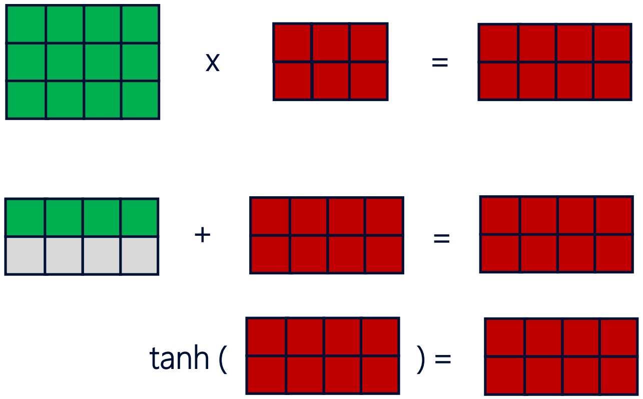
+ Bias vector: [Nodes]

→ Matrix: [Nodes x Batch size]

Apply **Activation** function (element-wise):

tanh([Nodes x Batch size])

→ Matrix: [Nodes x Batch size]



Results & Input – not reused

Constant parameters - reused (3.3M, 6.6MB as f16)

Achievable performance using cuBLAS

Measured time taken for cuBLAS Hgemm (f16) matrix multiply for a layer 1024 → 1024 with different batch sizes and TensorCores enabled

Best latency: 12µs for batch size <128

Lower limit latency for target model: 36µs (3*12)

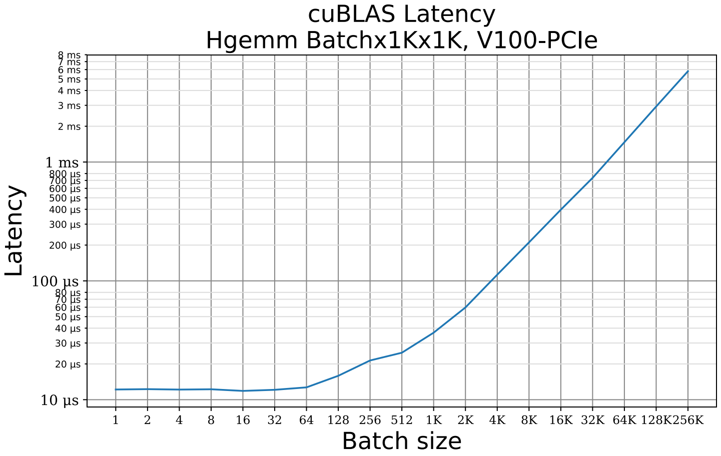
Maximum performance: 45M layers/s

Maximum achievable inference rate for target model: 15M Inf/s (45/3)

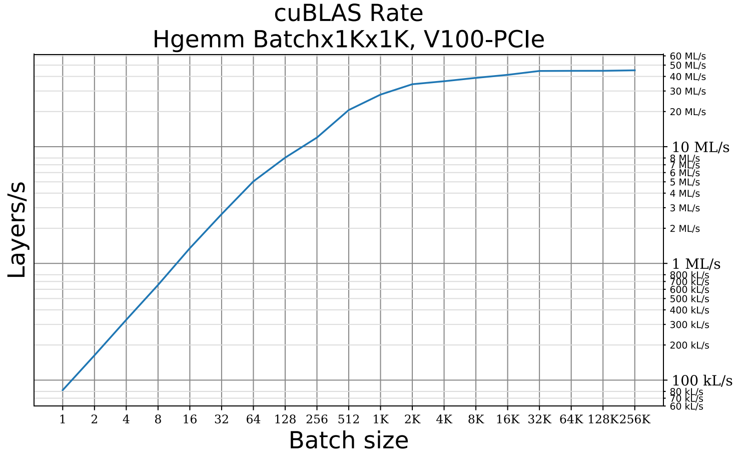
(Assuming use of cuBLAS and Ignoring small layers, bias, activation)

Hardware severely under-utilized at small batch sizes

Latency does not improve at all under batch 64



cuBLAS Latency
Hgemm Batchx1Kx1K, V100-PCIe



cuBLAS Rate
Hgemm Batchx1Kx1K, V100-PCIe

Hardware resources on NVIDIA V100 PCIe

Resource	Per SM	Per V100 (80x SM)
Max Clock Speed	1380 MHz	1380 MHz
Executing threads	128 (4x32)	10240
TensorCore count	8	640
TensorCore ops	128 (4*4*4*2) /cycle	113 TFLOPS
TensorCore BW needed	96 ([4x4]*3*2B) B/cycle	85 TB/s
Register memory	256 (64k*4B) KB	20.0 MB
Shared memory	96 KB	7.5 MB
L1/Shared memory BW	128 B/cycle (32*4)	14 TB/s
Main memory BW		0.9 TB/s

Must be in registers

Space for all 6.6MB f16 model parameters in registers

Our chosen strategies

→ **Use TensorCores directly** through **wmma** functions to achieve low latency with high rate
- With cuBLAS we would need to prioritise latency or rate.

Usually, layers are processed by sequential Cuda kernels
→ Registers need to be loaded again each time

If parameters are reloaded for each inference, rate will be limited
→ Use **persistent kernels** that can process many batches without reloading parameters
→ Create a pipeline allocating each SM to graph node, exchanging buffers with other SMs

TensorCore wmma interface currently has 3 size variants
→ Select 8 x 32 x 16 wmma operations to allow efficient **batch 8** operation

Matrix stage

1 block (1 SM) computes 256 x 256 matrix multiply using TensorCores via nvcuda::wmma API with 8 warps, 32 threads each

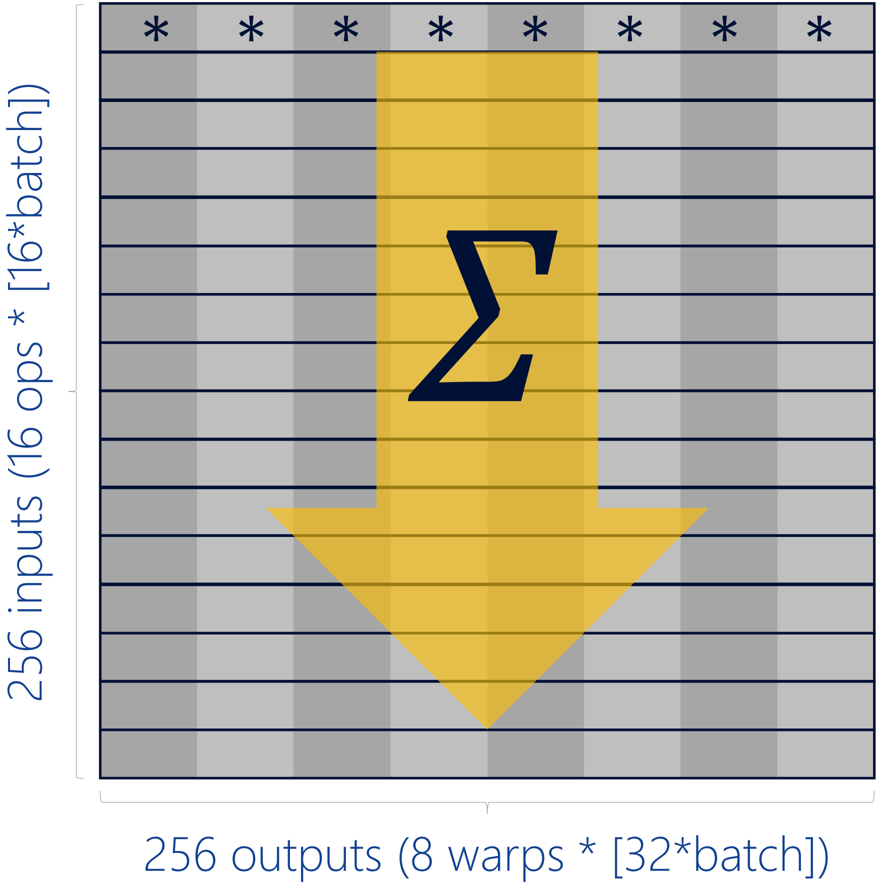
16 **preloaded weight fragments** in **registers**

Each input batch is loaded to shared memory for reuse by all warps

Accumulate 16 [8x32x16] matrix multiplies

Result is written back to global memory

Batch sizes up to 64 are processed in shared memory for increased throughput



* * * * * * * *
Σ
256 inputs (16 ops * [16*batch])
256 outputs (8 warps * [32*batch])

Reduction/bias/activation stage

For layer larger than 256x256, partial results from Matrix stage need to be reduced to final size

In case of 1024 x 1024, this means groups of 4 blocks need to be summed to single final block

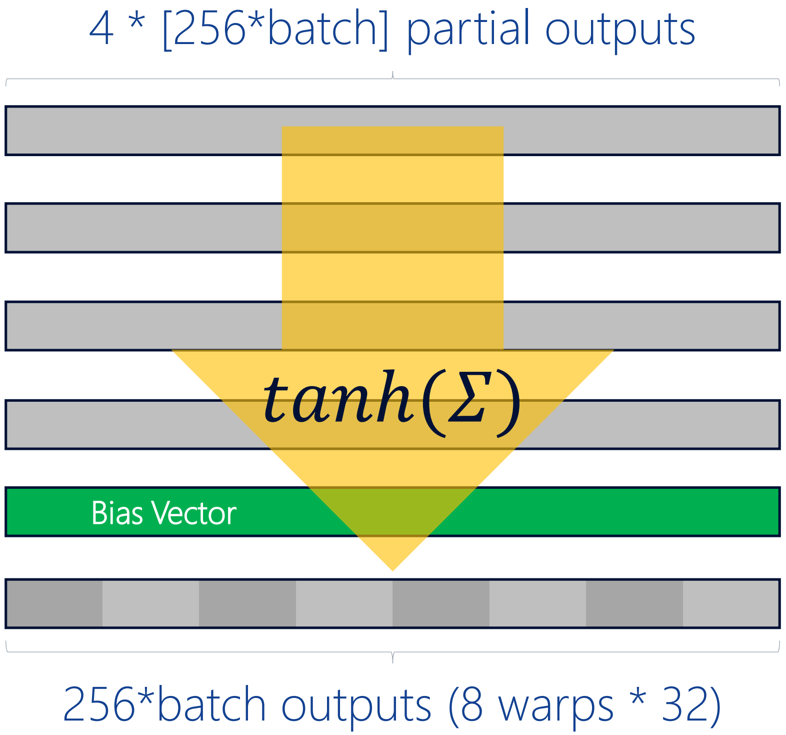
8 warps allow each thread to read and sum 4 partial values

Summed result has **bias** value **added**

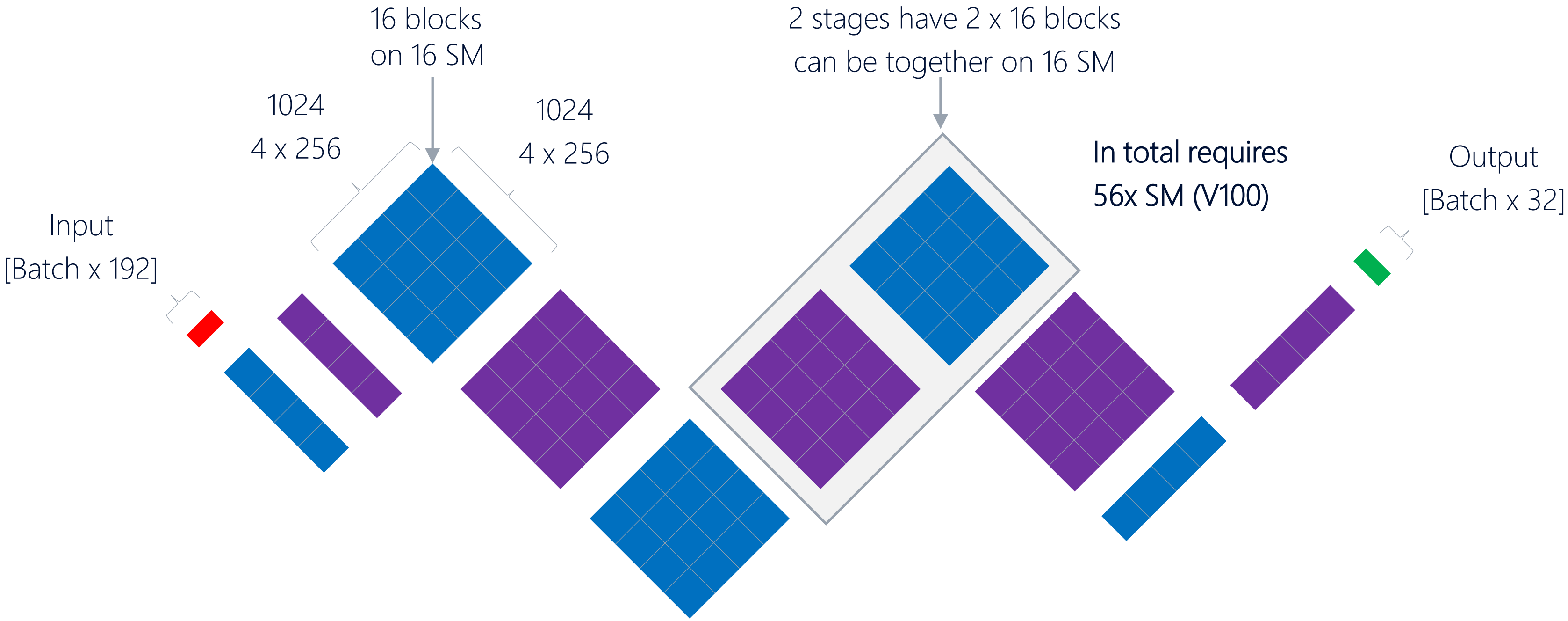
Activation function is **applied** (tanh)

Result is written back to global memory

Batch sizes up to 64 are processed in shared memory for increased throughput

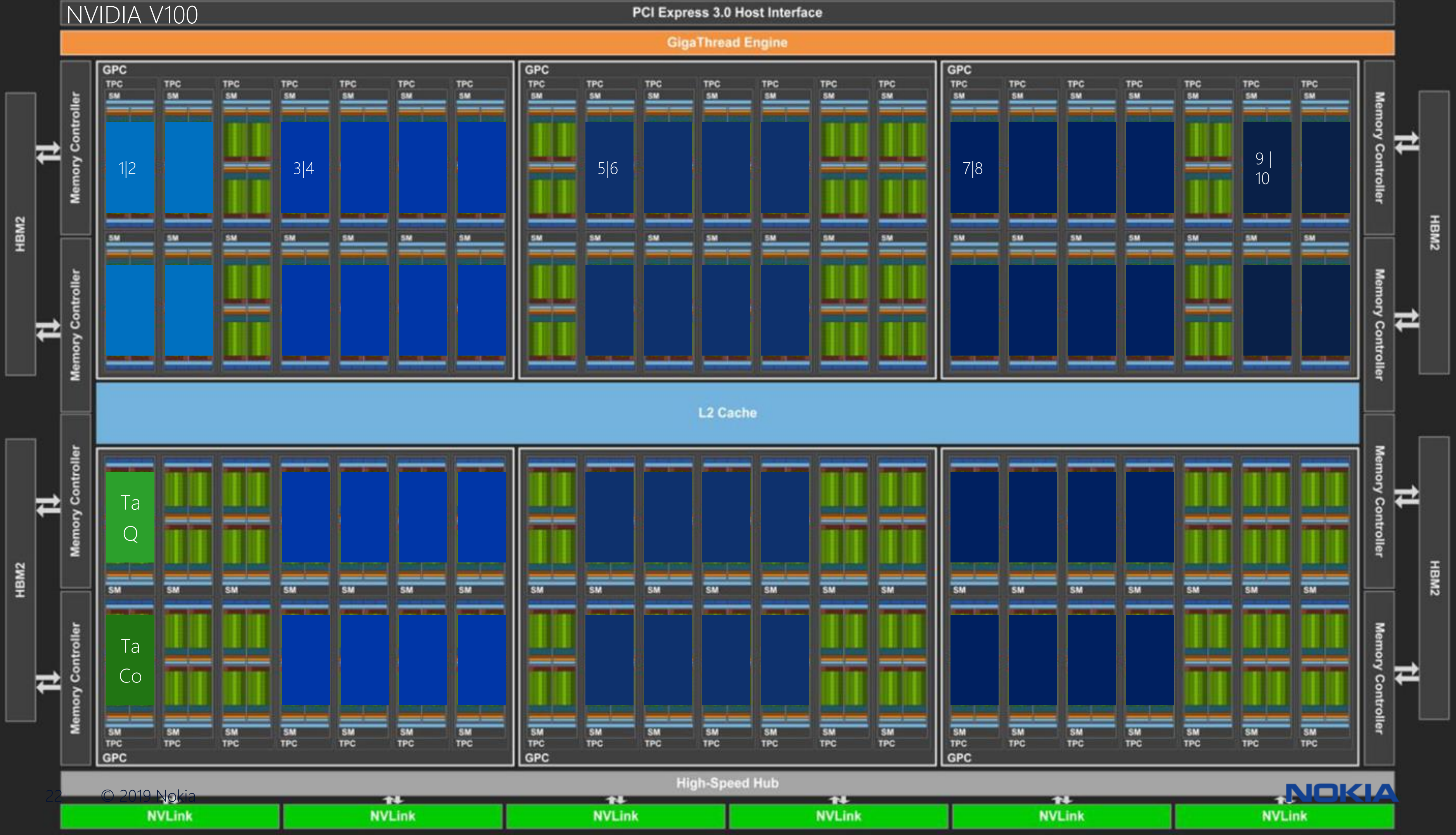

4 * [256*batch] partial outputs
$tanh(\Sigma)$
Bias Vector
256*batch outputs (8 warps * 32)

Mapping the model graph to a hardware-aware pipeline



10 stage pipeline - Alternating matrix and combined reduction/bias/activation stages
All blocks in a stage execute together when results from previous stage are available
One batch must pass through all stages, but each stage can be processing different batch

NVIDIA V100


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
1|2
3|4
5|6
7|8
9 | 10
Ta Q
Ta Co
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

Pipeline Kernel Structure

Pre-allocate batch-size buffers for exchanging data between stages

Stages notify in **both directions**:

- next stage when new work available
- previous stage when input buffer consumed to prevent overwriting

Fence to ensure visibility of results in L2 cache for different SM *before* notifying

Waiting & syncing cause unavoidable overhead when block cannot be processing

```
// Pseudocode
read_configuration()
load_parameters_to_registers()

while (*more_to_do) {
    while (!*new_input_data) {
        __nanosleep() }
    __syncthreads()
    read_input_data()
    __syncthreads()
    mark_input_data_read()
    process_input_data()
    while (!*last_output_read) {
        __nanosleep() }
    __syncthreads()
    write_output_data()
    __threadfence() // Ensure visibility
    __syncthreads()
    mark_output_data_written()
}
```

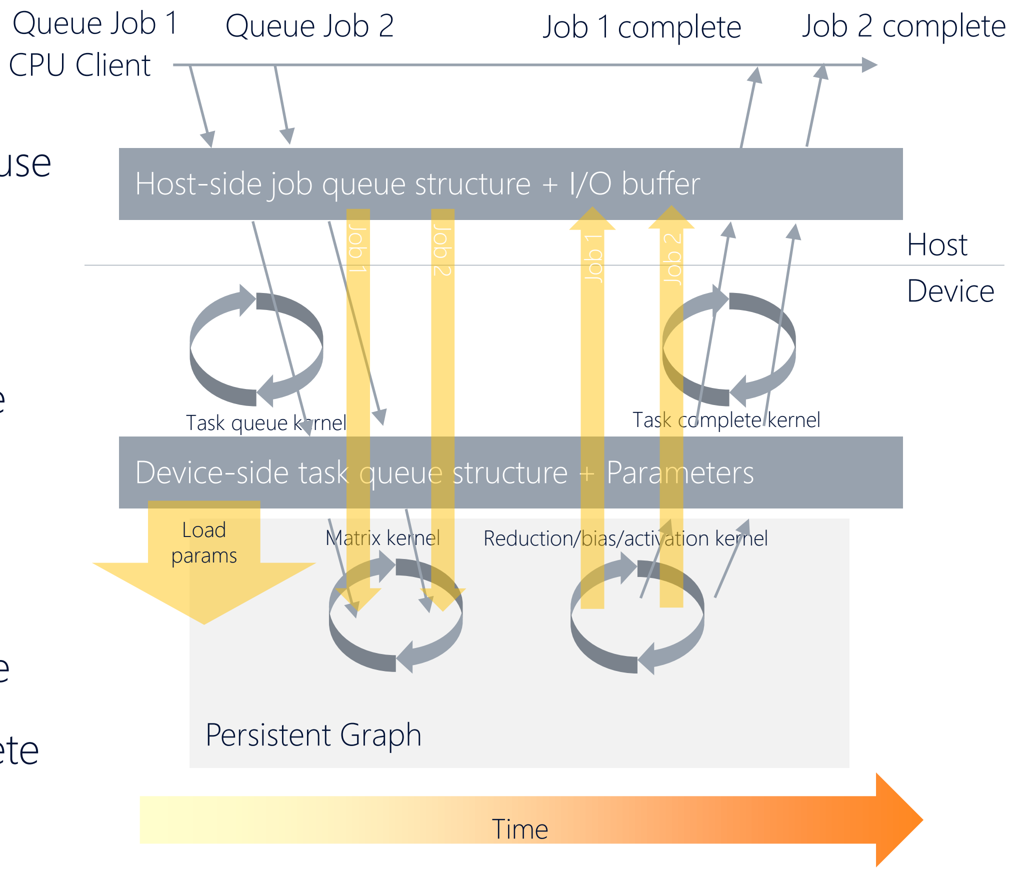
Host communication

For each task, input and output buffers can use host (unified) memory to avoid need for additional copy via device memory

A persistent kernel is used to watch for host job requests and then add to pipeline queue

A 2nd persistent kernel watches (L2) for queued tasks completion and notified host

Client can queue multiple tasks which can be processed simultaneously by the graph pipeline (one task per stage) and will complete in the order they were submitted.



Possible Future Directions

Support on T4 & INT8 inference

Convert from proof of concept to reusable framework accepting standard model formats as input

Support more node types, e.g. convolutions

Explore runtime graph reconfiguration:

- Allow multiple models to be executed with similar latency and throughput characteristics in every protocol frame cycle, increasing value of installation
- Current approach uses mainly L2 cache rather than device RAM bandwidth
- High device RAM bandwidth could allow for effective scheduled cyclic preloading of models before relevant new data availability

NOKIA