S9422: AN AUTO-BATCHING API FOR HIGH-PERFORMANCE RNN INFERENCE

Murat Efe Guney – Developer Technology Engineer, NVIDIA

March 20, 2019

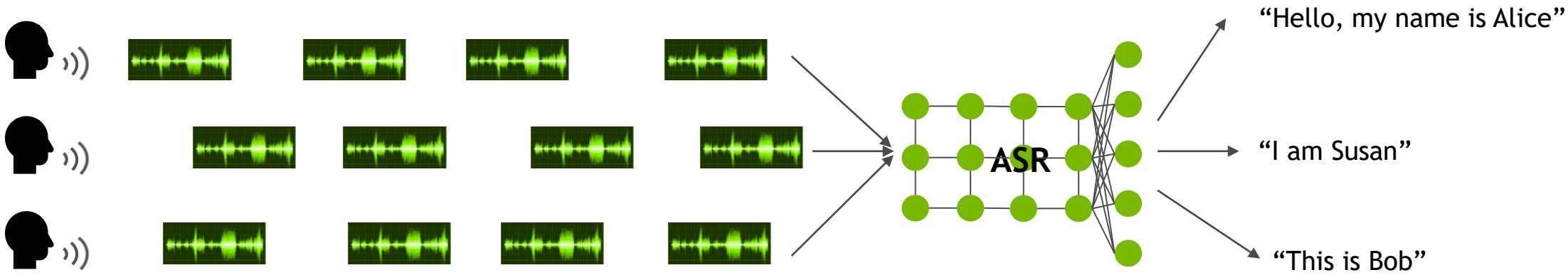
REAL-TIME INFERENCE

Sequence Models Based on RNNs

Sequence models for automatic speech recognition (ASR), translation, and speech generation

Real-time applications have a stream of inference requests from multiple users

Challenge is to perform inferencing with low latency and high throughput



BATCHING VS NON-BATCHING

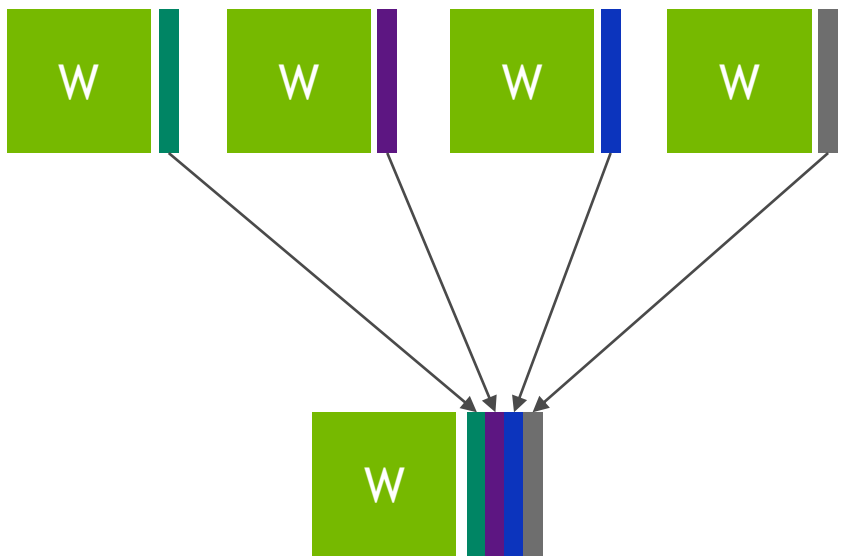
Batching: Grouping Inference Requests Together

Batch size = 1

- Run a single RNN inference task on a GPU
- Low-latency, but the GPU is underutilized

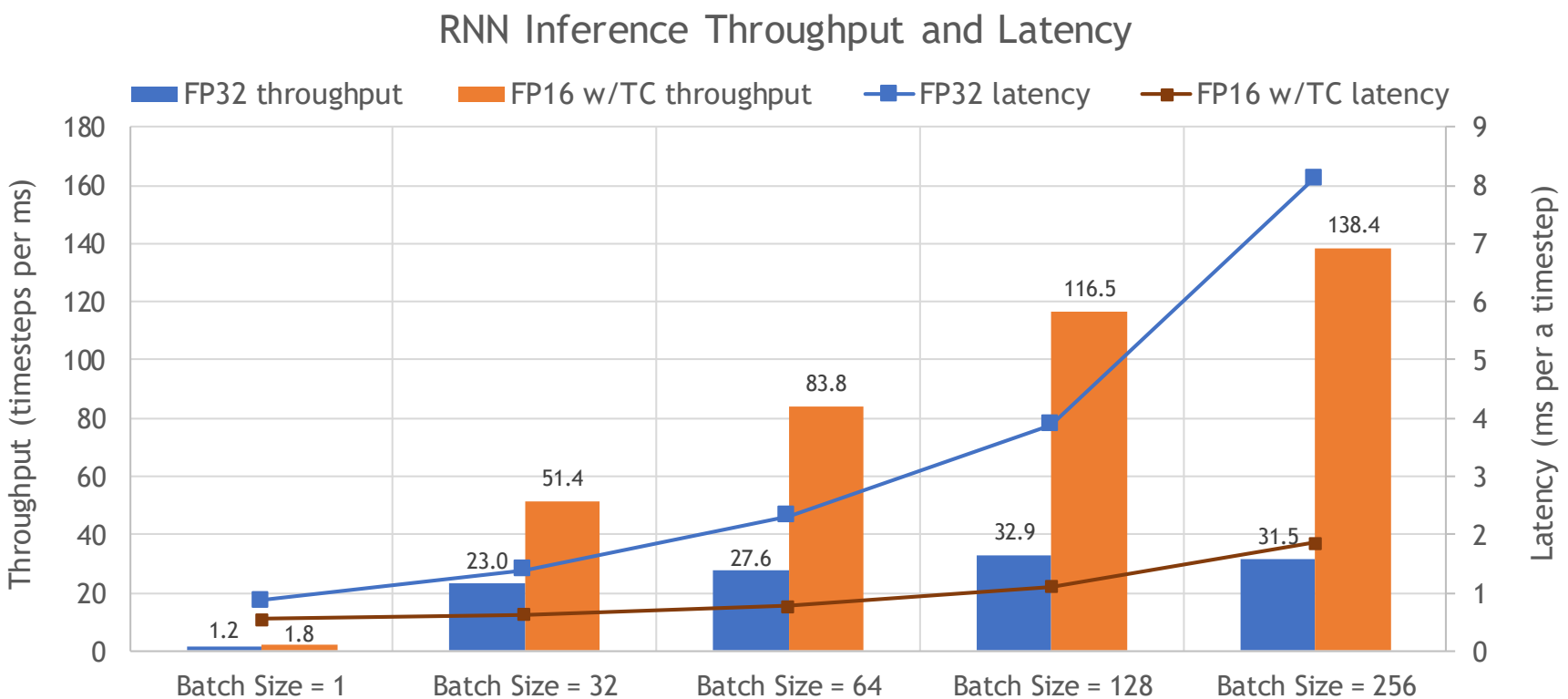
Batch size = N

- Group RNN inference instances together
- High throughput and GPU utilization
- Allows employing Tensor Cores in Volta and Turing



BATCHING VS NON-BATCHING

Performance Data on T4



RNN BATCHING

Challenges and Opportunities

Existing real-time codes are written for inferencing many instances with batch size = 1

Real-time batching requires extra programming effort

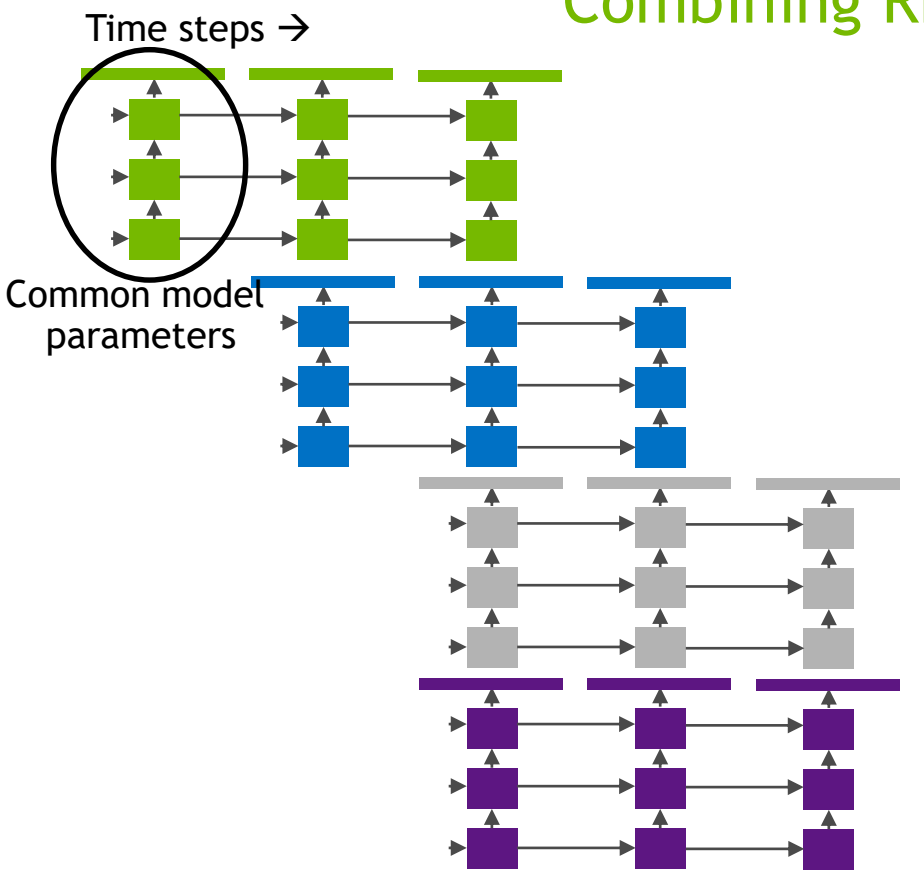
A naïve implementation can suffer from significant increase in latency

An ideal solution will allow making a tradeoff between latency and throughput

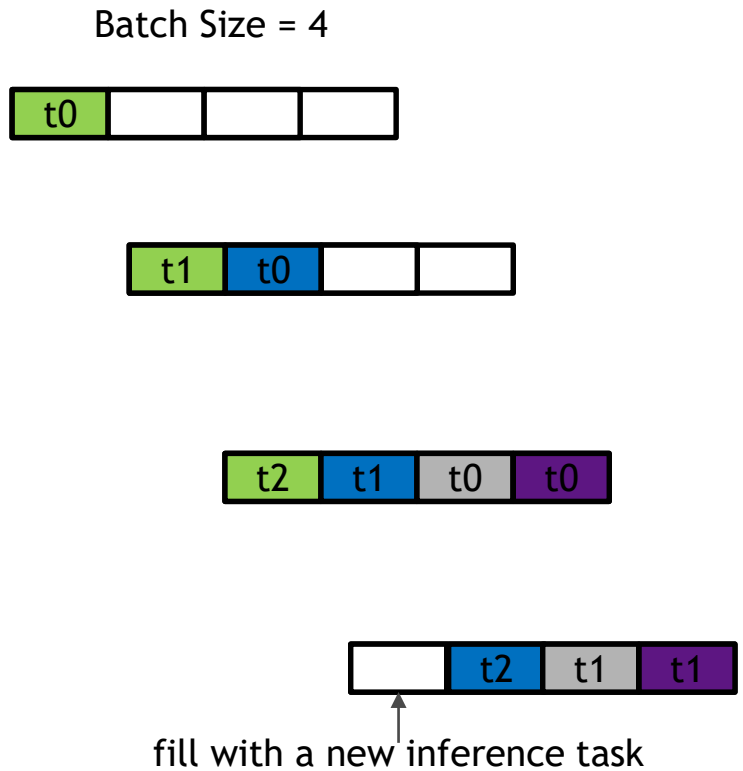
RNN cells provide an opportunity to merge inference tasks at different timesteps

RNN BATCHING

Combining RNNs at Different Timesteps



Inference Tasks Arrival Time



Batched Execution of Timesteps

RNN CELLS

RNN Cells Supported in TensorRT and cuDNN

RELU

$$h_t = \mathrm{ReLU}(W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ri})$$

TANH

$$h_t = \tanh(W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ri})$$

LSTM

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ri})$$
$$f_t = \sigma(W_f x_t + R_f h_{t-1} + b_{Wf} + b_{Rf})$$
$$o_t = \sigma(W_o x_t + R_o h_{t-1} + b_{Wo} + b_{Ro})$$
$$c'_t = \tanh(W_c x_t + R_c h_{t-1} + b_{Wc} + b_{Rc})$$
$$c_t = f_t \circ c_{t-1} + i_t \circ c'_t$$
$$h_t = o_t \circ \tanh(c_t)$$

GRU

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ru})$$
$$r_t = \sigma(W_r x_t + R_r h_{t-1} + b_{Wr} + b_{Rr})$$
$$h'_t = \tanh(W_h x_t + r_t \circ (R_h h_{t-1} + b_{Rh}) + b_{Wh})$$
$$h_t = (1 - i_t) \circ h'_t + i_t \circ h_{t-1}$$

HIGH-PERFORMANCE RNN INFERENCING

cuDNN Features

High-performance implementations of Tanh, RELU, LSTM and GRU recurrent cells

An arbitrary batch size and number of timesteps can be executed

Easy access to internal and hidden states of the RNN cells for each timestep

Persistent kernels for small minibatch and long sequence lengths (compute capability >= 6.0)

LSTMs with recurrent projections to reduce the op count

Utilize Tensor Cores for FP16 and FP32 cells (125 TFLOPs on V100 and 65 TFLOPs on T4)

UTILIZING TENSOR CORES

cuDNN, cuBLAS and TensorRT

cuDNN

```
// input, output and weight data types are FP16
cudnnSetRNNMatrixMathType(cudnnRnnDesc, CUDNN_TENSOR_OP_MATH);
```

```
// input, output and weight are FP32, which is converted internally to FP16
cudnnSetRNNMatrixMathType(cudnnRnnDesc, CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION);
```

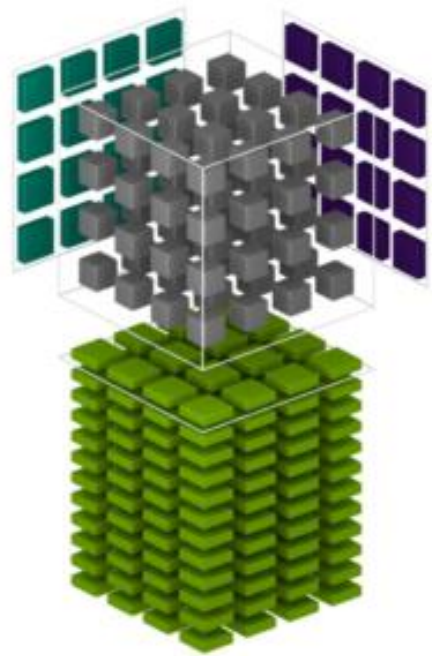
cuBLAS and cuBLASLt

```
cublasGemmEx(...);
```

```
cublasLtMatmul(...);
```

TensorRT

```
builder->setFp16Mode(true);
```



RNN INFERENCING WITH CUDNN

Key Functions

```
cudnnCreateRNNDescriptor(&rnnDesc); // creates an RNN descriptor

cudnnSetRNNDescriptor(rnnDesc, … );   // sets the RNN descriptor

cudnnGetRNNLinLayerMatrixParams(cudnnHandle, rnnDesc, …); // set weights

cudnnGetRNNLinLayerBiasParams(cudnnHandle, rnnDesc, …); // set bias

cudnnRNNForwardInference(cudnnHandle, rnnDesc, … ); // perform inferencing

cudnnDestroyRnnDescriptor(rnnDesc); // destroy the RNN descriptor
```

AUTO-BATCHING FOR HIGH THROUGHPUT

Automatically Group Inference Instances

Rely on cuDNN, cuBLAS and TensorRT for high-performance RNN implementation

Input, hidden states and outputs are tracked automatically with a new API

Exploits optimization opportunities by overlapping compute, transfer and host computations

Similar ideas explored at:

- Low-latency RNN inference using cellular batching (Jinyang Li et. al., GTC 2018)
- Deep Speech 2: End-to-End Speech Recognition in English and Mandarin (Dario Amodei et. al., CoRR, 2015)

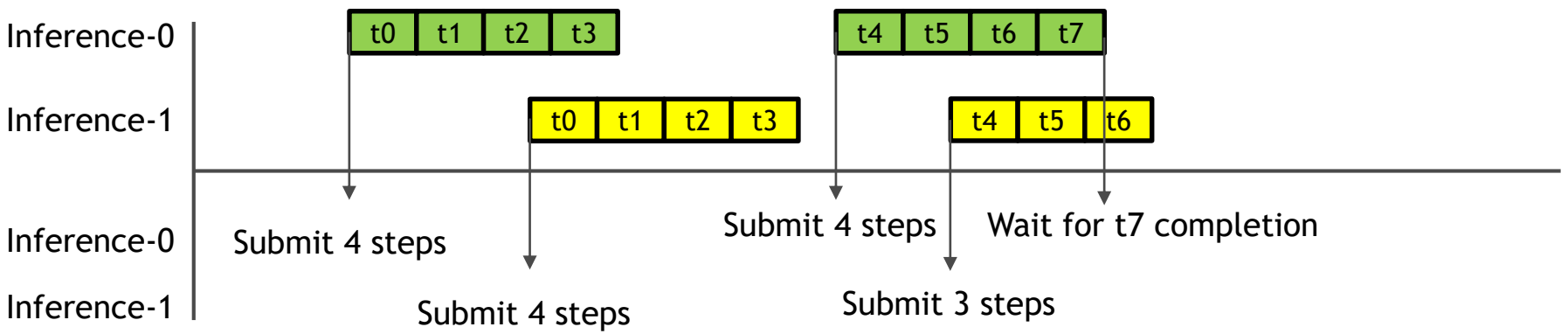
STREAMING INFERENCE API

An Auto-batching Solution

Non-blocking function calls with a mechanism to wait on completion

Inferencing can be performed in segments with multiple timesteps for real-time processing

A background thread that combines and executes the single inference tasks



STREAMING INFERENCE API

List of Functions

streamHandle = **CreateRNNInferenceStream**(modelDesc);

rnnHandle = **CreateRNNInference** (streamHandle);

RNNInference (rnnHandle, pInput, pOutput, seqLength);

timeStep = **WaitRNNInferenceTimeStep**(rnnHandle, timeStep);

timeStep = **GetRNNInferenceProgress**(rnnHandle);

DestroyRNNInference (rnnHandle);

DestroyRNNInferenceStream(streamHandle);

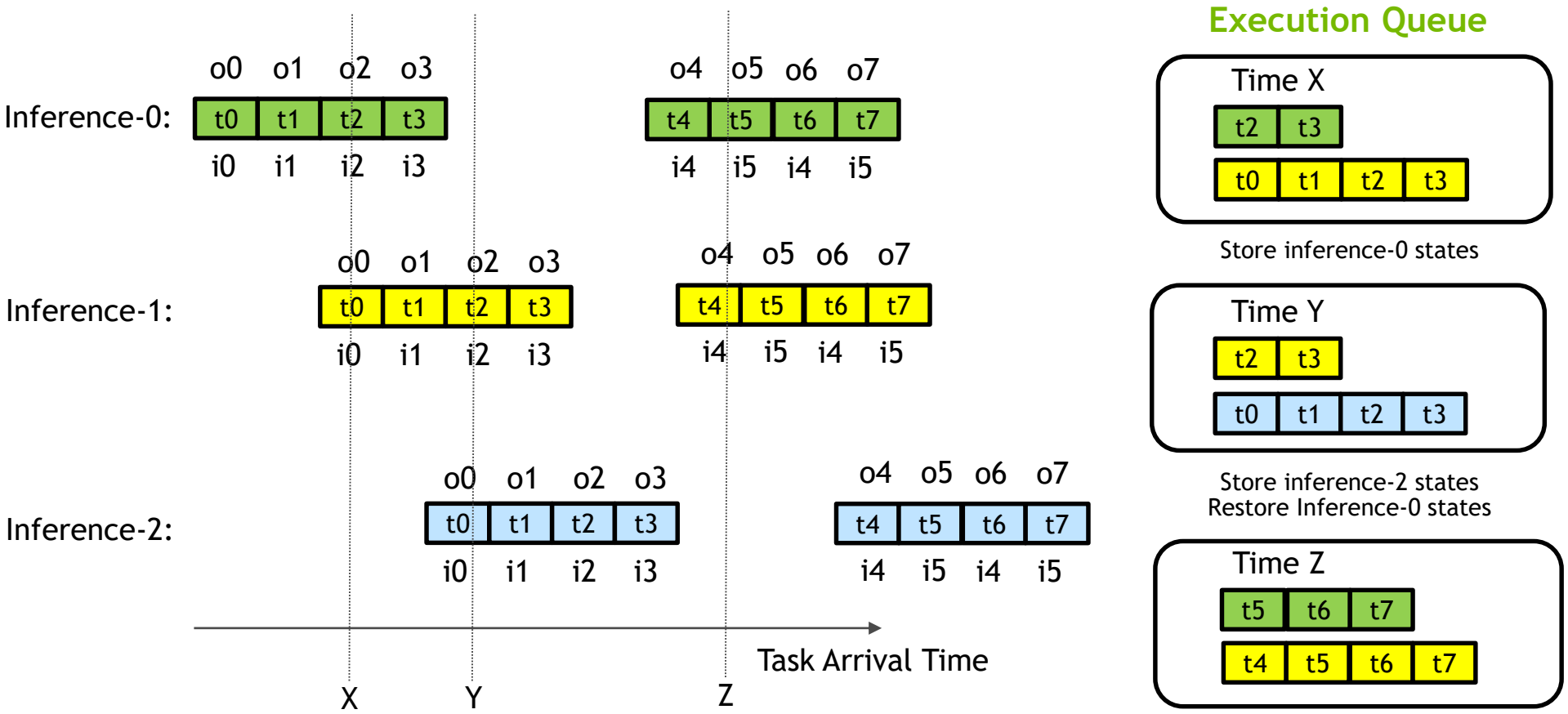
EXAMPLE USAGE

Two Inference Instances

```
// Create the inference stream with shared model parameters
streamHandle = CreateRNNInferenceStream(modelDesc);

// Create two RNN inference instances
rnnHandle[0] = CreateRNNInference(streamHandle);
rnnHandle[1] = CreateRNNInference(streamHandle);

// Request inferencing for each inference instance with 10 timesteps (non-blocking call)
RNNInference(rnnHandle[0], pInput[0], pOutput[0], 10);
RNNInference(rnnHandle[1], pInput[1], pOutput[1], 10);
// Request inferencing an additional 5 time step for the second inference instance
RNNInference(rnnHandle[1], pInput[1] + 10*inputSize, pOutput[1] + 10*outputSize, 5);

// Wait for the completion of lastly added inferencing job
WaitRNNInferenceTimeStep(rnnHandle[1], 15);

// Destroy the two inferencing tasks and the inference stream
DestroyRNNInference(rnnHandle[0]);
DestroyRNNInference(rnnHandle[1]);
DestroyRNNStream(streamHandle);
```

RNN INFERENCING WITH SEGMENTS

Execution Queue and Task Switching for Batch Size = 2



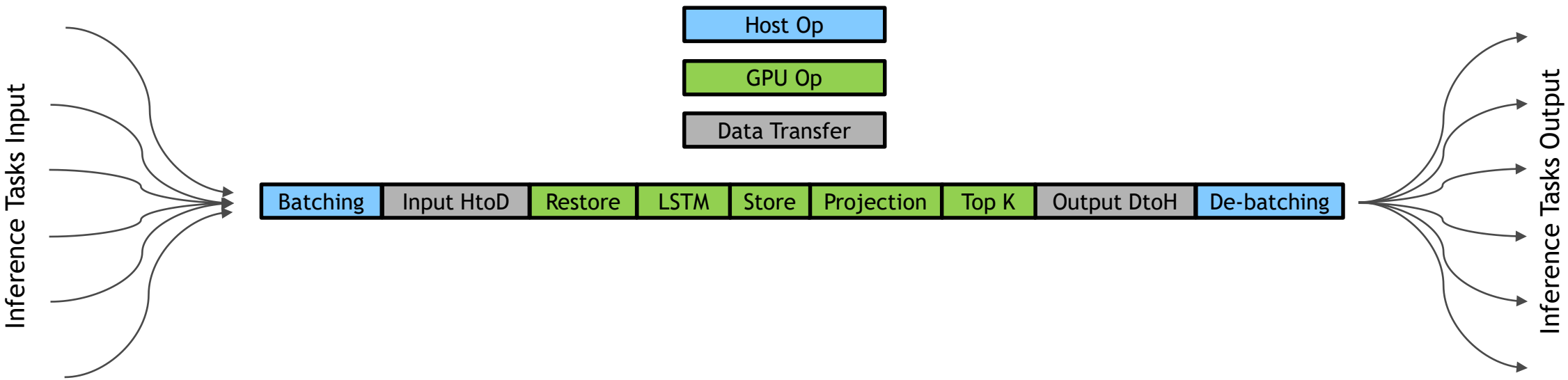
IMPLEMENTATION

Auto-batching and GPU Execution

1. Find the inference tasks ready to execute time steps
2. Determine the batch slots for each inference task
3. Send the inputs to GPU for batched processing
4. Restore hidden states as needed (+ cell states for LSTMs)
5. Batched execution on the GPU
6. Store hidden states as needed (+ cell states for LSTMs)
7. Send the batched results back to host
8. De-batch the results on the host

IMPLEMENTATION

Batching, Executing, and De-batching Inference Tasks

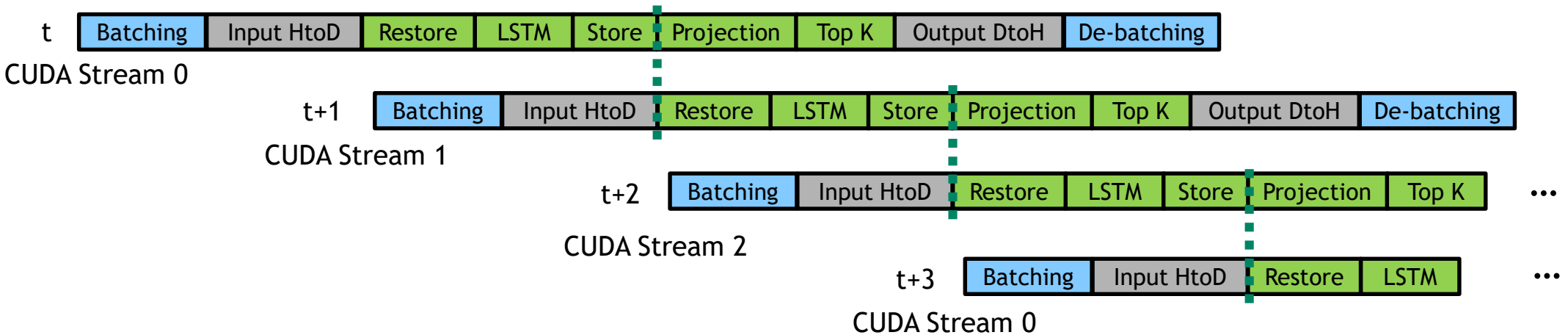


Background thread accepting inference tasks

At each timestep inference tasks are batched, executed on the GPU and de-batched

PERFORMANCE OPTIMIZATIONS

Hiding Host Processing, Data transfers, and State Management



Overlapping opportunities between timesteps for compute, batching, de-batching and transfer

Perform batching and de-batching on separate CPU threads: provides better CPU BW and GPU overlap

Employ three CUDA streams and triple-buffering of the output to better exploit concurrency

PERFORMANCE EXPERIMENTS

An Example LSTM Model

Input size = 128

7 LSTM layers with 1024 hidden cells

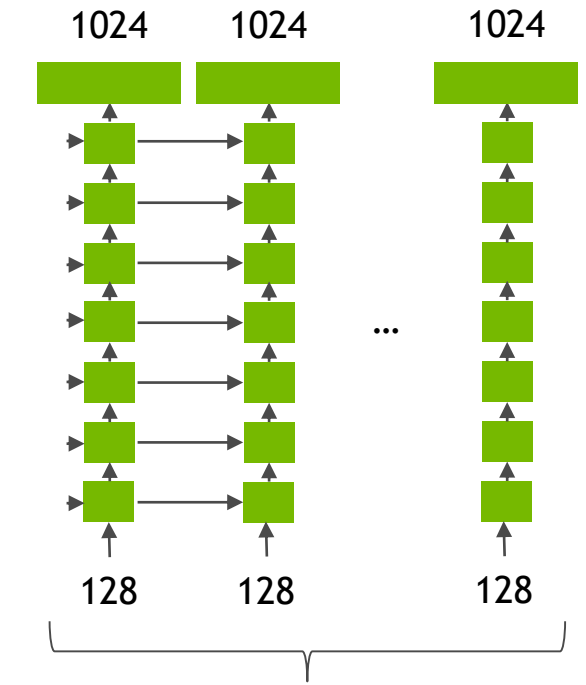
A final projection layer with 1024 output

Timestep per each inference segment = 10

Total sequence length = 1000

Experiments are performed on T4 and GV100

End-to-end time: task submission to results arriving to host



PERFORMANCE EXPERIMENTS

Benchmarking Code

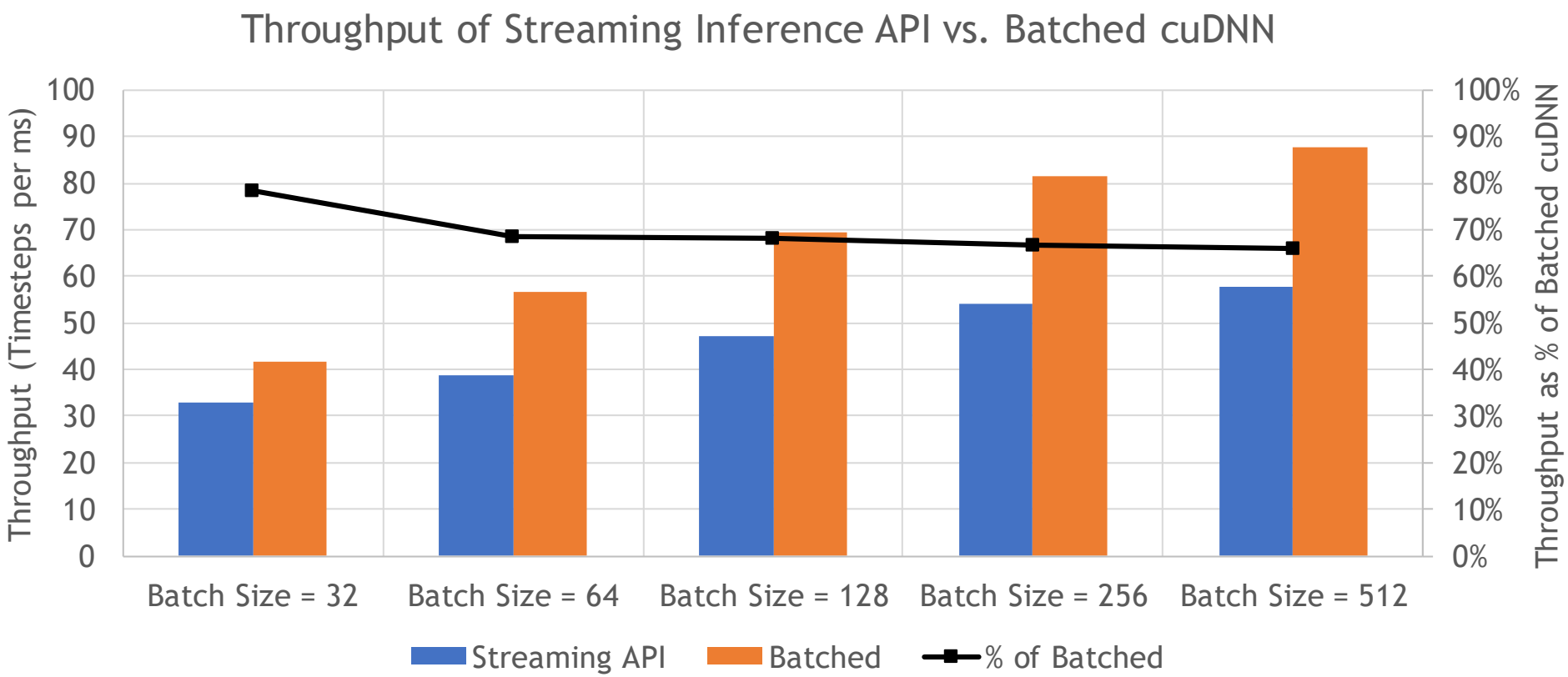
```
// Queue up inferencing tasks with 10 timesteps each
time[0] = time();
RNNInference(rnnHandle[0], pInput[0], pOutput[0], 10);
time[1] = time();
RNNInference(rnnHandle[1], pInput[1], pOutput[1], 10);
...
RNNInference(rnnHandle[N-1], pInput[N-1], pOutput[N-1], 10);

// Wait for the completion of first inferencing task
WaitRNNInferenceTimeStep(rnnHandle[0], 10);
time[0] = time[0] - time();
RNNInference(rnnHandle[N], pInput[N], pOutput[N], 10);

// Wait for the completion of second inferencing task
WaitRNNInferenceTimeStep(rnnHandle[1], 10);
time[1] = time[1] - time();
RNNInference(rnnHandle[N+1], pInput[N+1], pOutput[N+1], 10);
...
```

There is at most N inference requests on the fly at a given time.

Measure the time required to finish each inference request including the data transfer time.

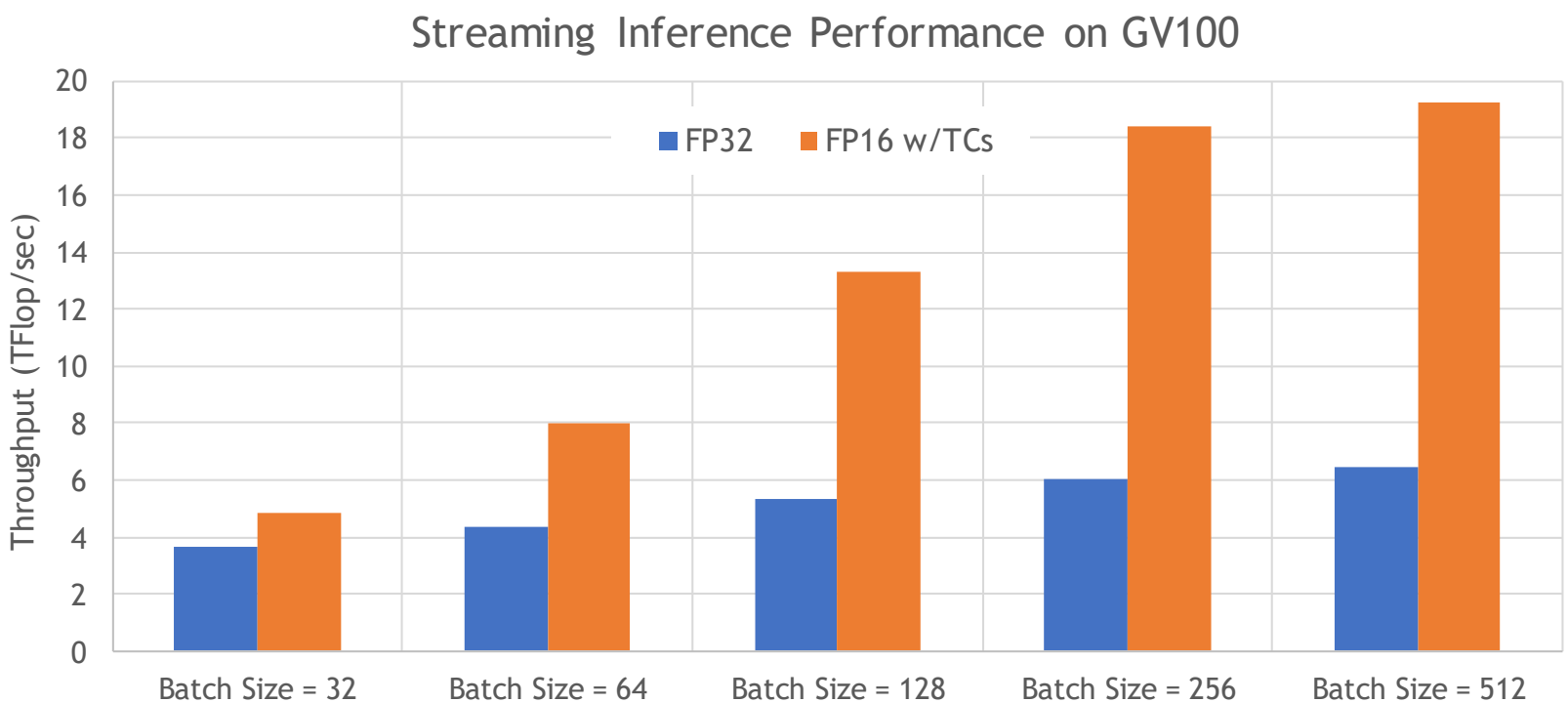
COMPARISION AGAINST BATCHED CUDNN

FP32 Model, GV100 Numbers



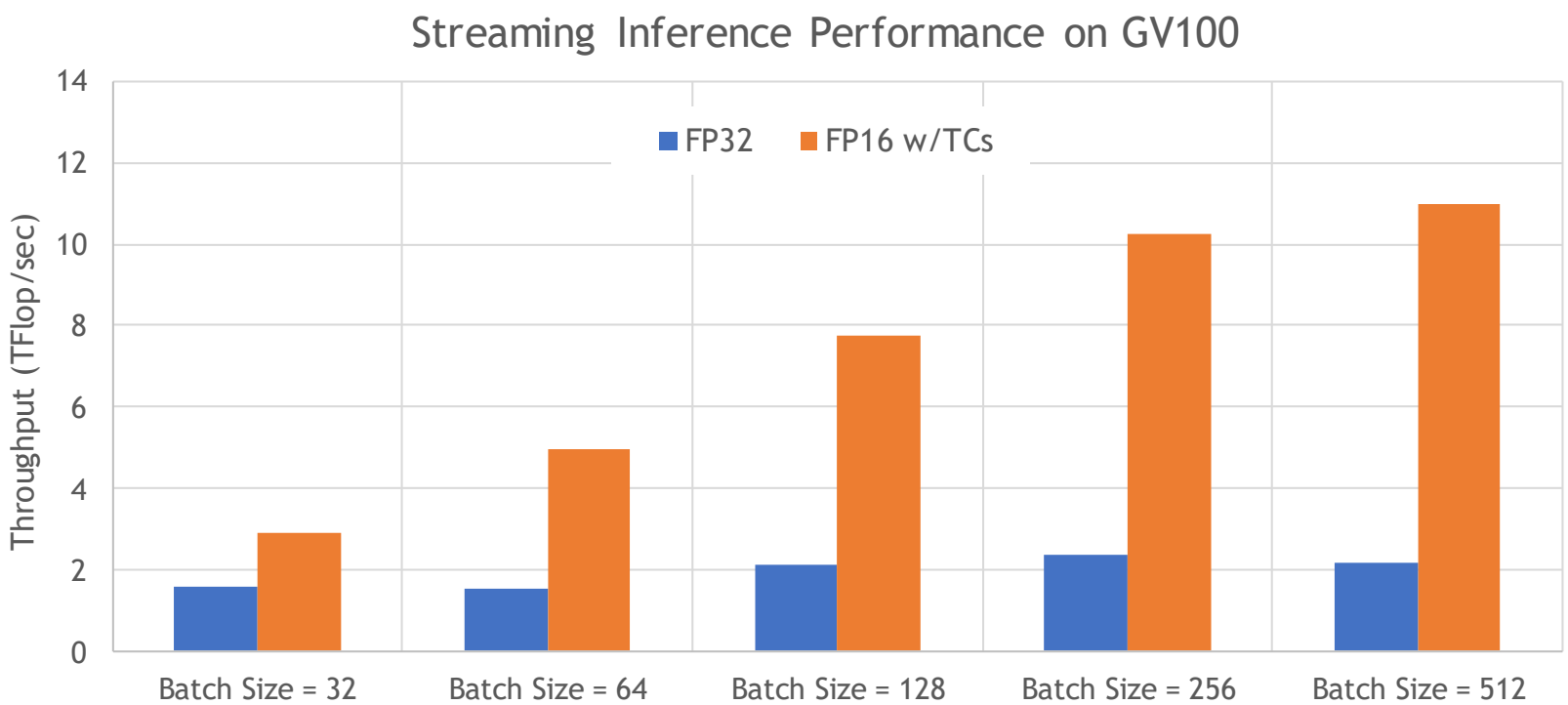
PERFORMANCE WITH TENSOR CORES

FP16 on GV100



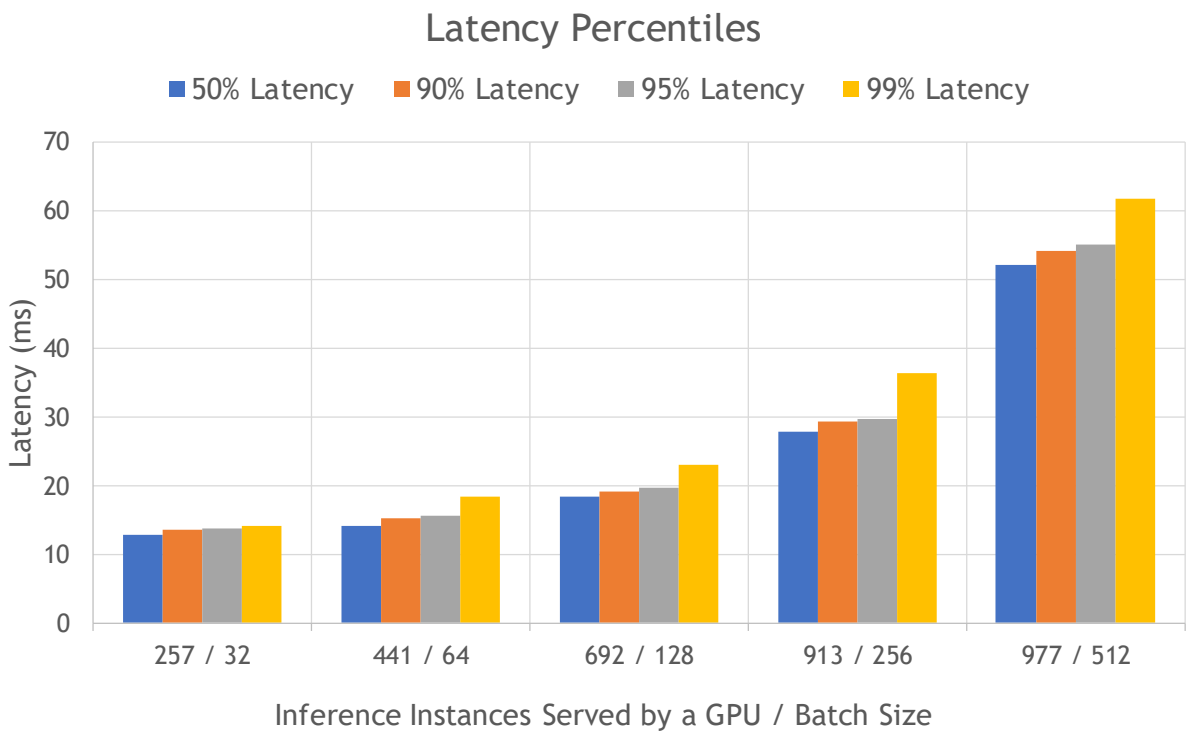
PERFORMANCE WITH TENSOR CORES

FP16 on T4



LATENCY VS THROUGHPUT TRADEOFF

FP16 on T4

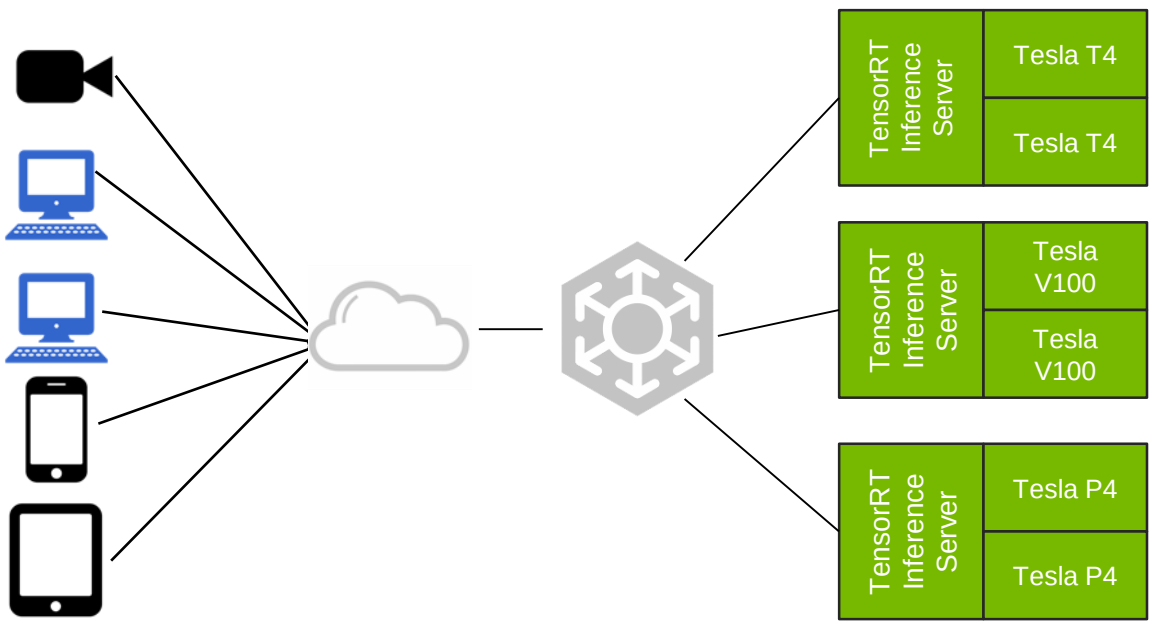


Assuming each inference segment represents100ms audio

Choose a batch size that will maximize throughput while staying within latency budget

NVIDIA TENSORRT INFERENCE SERVER

Production Data Center Inference Server



Maximize real-time inference performance of GPUs

Quickly deploy and manage multiple models per GPU per node

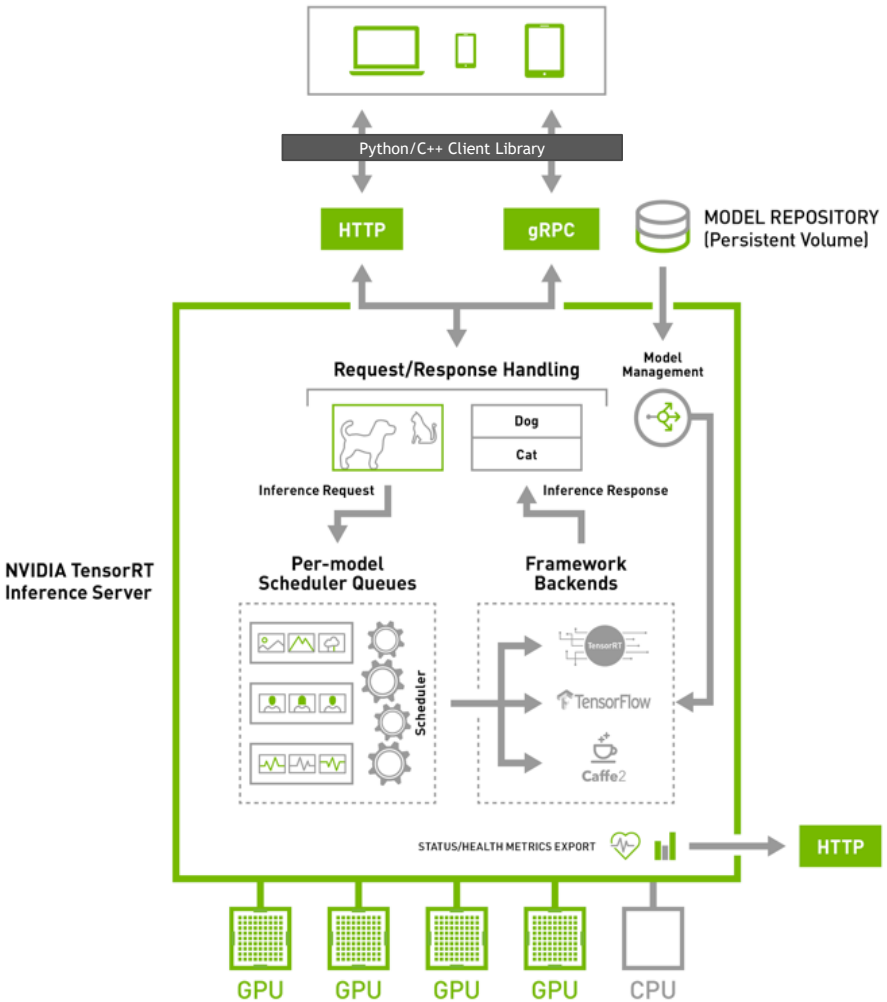
Easily scale to heterogeneous GPUs and multi GPU nodes

Integrates with orchestration systems and auto scalers via latency and health metrics

Open source for thorough customization and integration

INFERENCE SERVER ARCHITECTURE

Available with Monthly Updates



Models supported

- TensorFlow GraphDef/SavedModel
- TensorFlow and TensorRT GraphDef
- TensorRT Plans
- Caffe2 NetDef (ONNX import)
- Custom backends

Multi-GPU support

Concurrent model execution

Server HTTP REST API/gRPC

Python/C++ client libraries

INFERENCE SERVER BATCHERS

Dynamic and Sequence Batching

Dynamic Batching

TensorRT Inference Server (TRTIS) groups inference requests based on customer defined metrics for optimal performance

Customer defines 1) batch size and/or 2) latency requirements

Sequence Batching

TRTIS can keep track of the inference requests belonging to a stateful model

The client application assigns a correlation ID for a stream of inferences belonging to the same sequence

Use together with a custom backend to store and restore the internal states of the model

SUMMARY

Designed and implemented the Streaming Inference API

Automatically batches the RNN inference requests together to achieve high throughput

Code written for batch size = 1 achieves ≥66% throughput of batched execution (FP32)

Allows utilizing the Tensor Cores on Volta and Turing architectures

Hit latency targets by choosing the right batch size

Generalizes to sequence models with interdependent inference streams

TRTIS sequence batcher and custom backends for high-performance real-time inferencing

S9438 - Maximizing Utilization for Data Center Inference with TensorRT Inference Server

RESOURCES

TRTIS blog post and documentation:

https://devblogs.nvidia.com/nvidia-serves-deep-learning-inference/

https://docs.nvidia.com/deeplearning/sdk/tensorrt-inference-server-guide/docs/


NVIDIA