Groq AI Workshop

ALCF AI Testbed

June 2024

Agenda - Day 1

Session	Description	Length	Speaker
Intro to ALCF	Introduction to the Argonne Leadership Computing Facility AI Testbed.	5 mins	ALCF Staff
Welcome to Groq	Introduction to the AI/ML space, who we are, and applications that can leverage Groq for inference.	5 mins	Jonathan Ross, CEO & Founder
Groq Language Processing Unit (LPU)™ Architecture	Deep dive on the Groq Language Processing Unit™ (LPU) tensor streaming architecture, including in-depth explanations on each module of the chip.	45 mins	Andrew Bitar, Principal Compiler Engineer & Manager
Accessing GroqRack™ at ALCF AI Testbed	How to access GroqRack.	30 mins	ALCF Staff
	15 MINUTE BREAK 🚀		
Porting Models with GroqFlow	Step-by-step walkthrough of model porting with GroqFlow for execution on GroqRack (including best practices).	60 mins	Sanjif Shanmugavelu, Software Engineer
Benchmarking Models with MLAgility	How to benchmark multiple models with MLAgility.	20 mins	Sanjif Shanmugavelu, Software Engineer

Welcome to Groq

Jonathan Ross
Founder & CEO

In the Past Few Months

Developer Community

15,000 + developers

https://www.discord.gg/groq

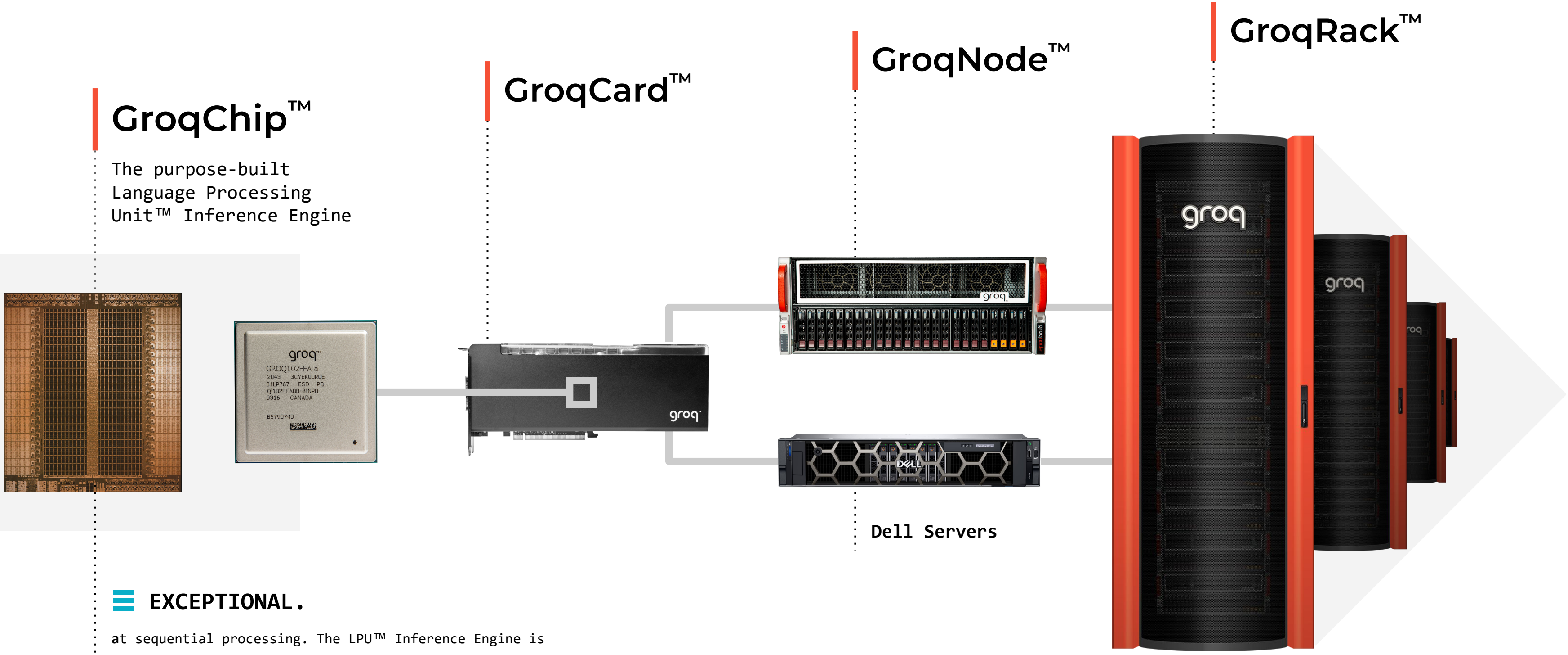
Groq Language Processing Unit™ (LPU) Architecture

Andrew Bitar
Principal Compiler Engineer & Manager

Groq LPU Architecture

AGENDA

1. Architecture Overview
2. Key Functional Units
3. Scaling to 1000s of Groq LPUs



GroqChip™

The purpose-built
Language Processing
Unit™ Inference Engine

GroqCard™

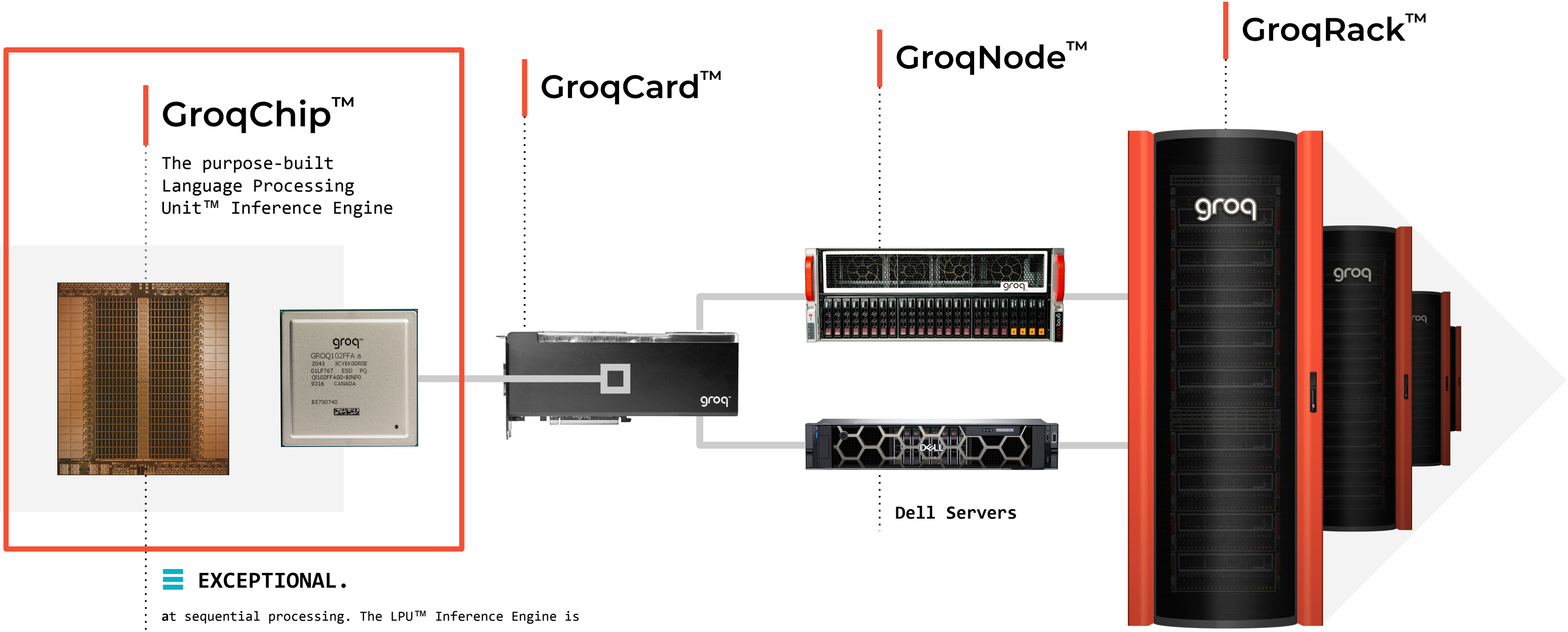
GroqNode™

GroqRack™

Dell Servers

EXCEPTIONAL.

at sequential processing. The LPU™ Inference Engine is designed to scale and is more power-efficient, with greater performance, than a GPU for AI applications like LLMs.



GroqChip™

The purpose-built Language Processing Unit™ Inference Engine

GroqCard™

GroqNode™

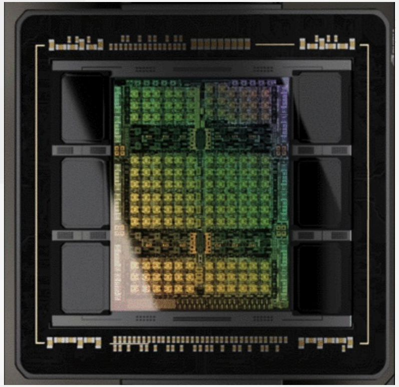
GroqRack™

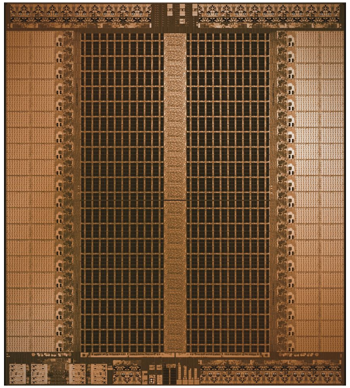
Dell Servers

EXCEPTIONAL.

at sequential processing. The LPU™ Inference Engine is designed to scale and is more power-efficient, with greater performance, than a GPU for AI applications like LLMs.

Groq Simplifies Compute





Graphics Processor (GPU)

COMPLEX

Non-deterministic execution

Manual kernel-based compilation

Higher latency

Higher costs

Language Processing Unit (LPU)

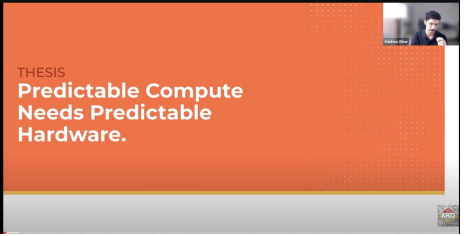
SIMPLIFIED

Deterministic & predictable execution

Automated kernel-less compilation

Lower latency

Higher efficiency at scale



THESIS

Predictable Compute Needs Predictable Hardware.

Presented at Crossroads 3D-FPGA Academic Research Center - December 2022

The Missing Middle

Algorithms

Dataflow dominated

Statically predictable set of executed operations

Highly-parallel vector operations

✔ PREDICTABLE

Compilers

Remain a challenge

Reliant on hand-tuned libraries

Fragmented front-end ecosystem

Require iterative hardware profiling

Hardware

High-density compute using SIMD

Less silicon area spent on re-ordering and speculation

More memory bandwidth

✖ UNPREDICTABLE

Groq LPU Overview

SRAM Memory

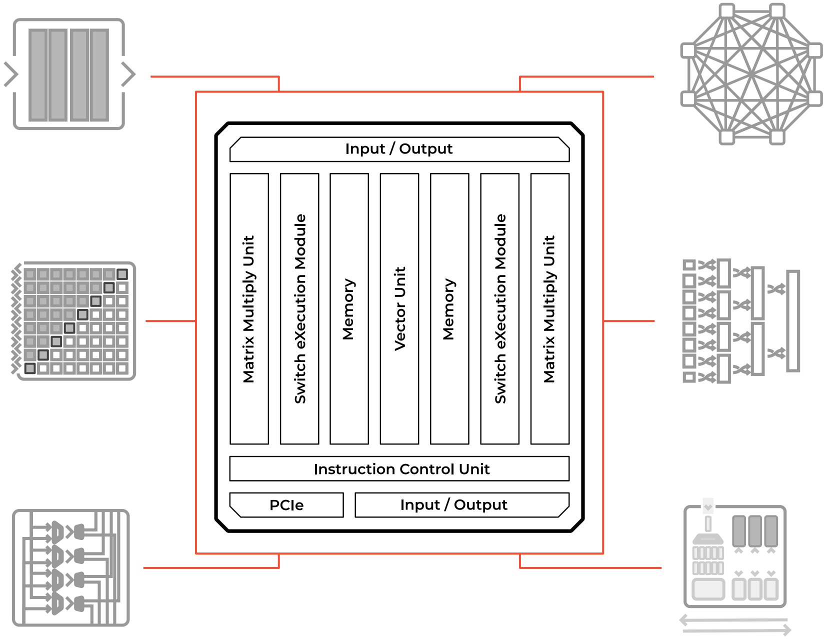
Massive concurrency
80 TB/s of BW
230MB capacity
Stride insensitive

Groq TruePoint™ Matrix

4x Engines
750 TOP/s int8
188 TFLOP/s fp16
320x320 fused dot product

Programmable Vector Units

5,120 Vector ALUs for
high performance



Networking

480 GB/s bandwidth
Extensible network scalability
Multiple topologies

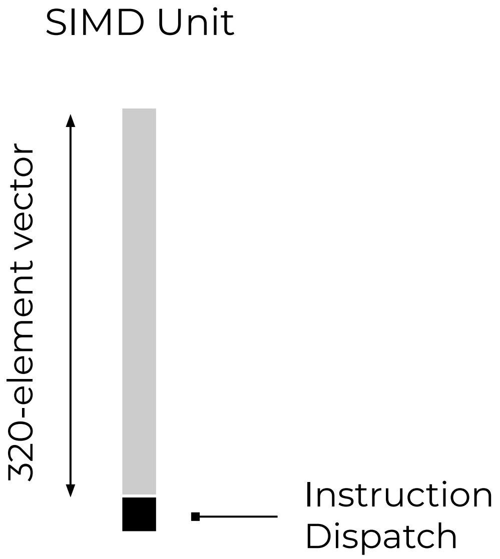
Data Switch

Shift, Transpose, Permuter for improved data movement and data reshapes

Instruction Control

Multiple instruction queues for instruction parallelism

Groq LPU Building Blocks



Groq LPU Building Blocks

Build different types of specialized SIMD units

MXM
Matrix-Vector /
Matrix-Matrix Multiply

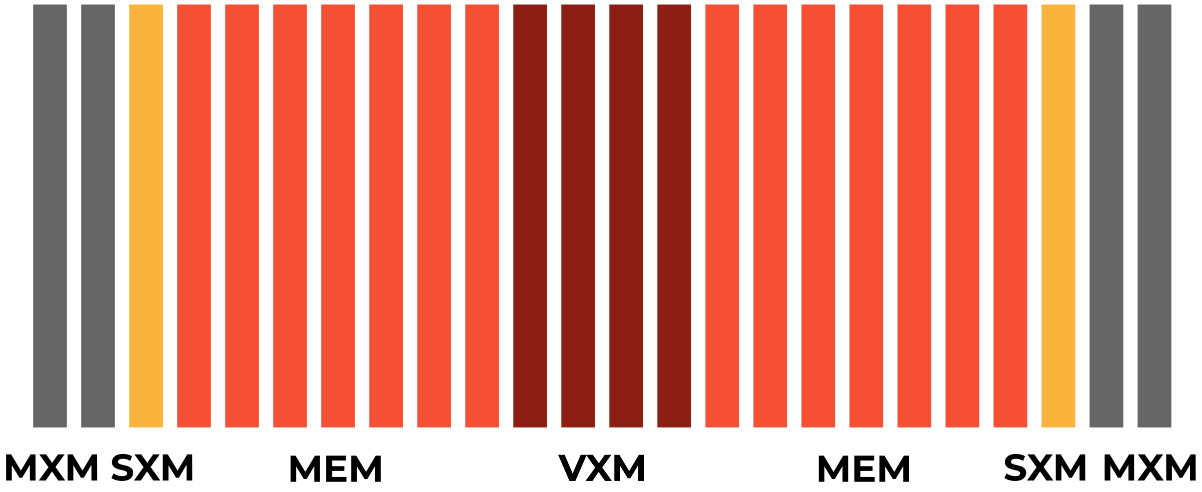
VXM
Vector-Vector
Operations

SXM
Data Reshapes

MEM
On-chip SRAM

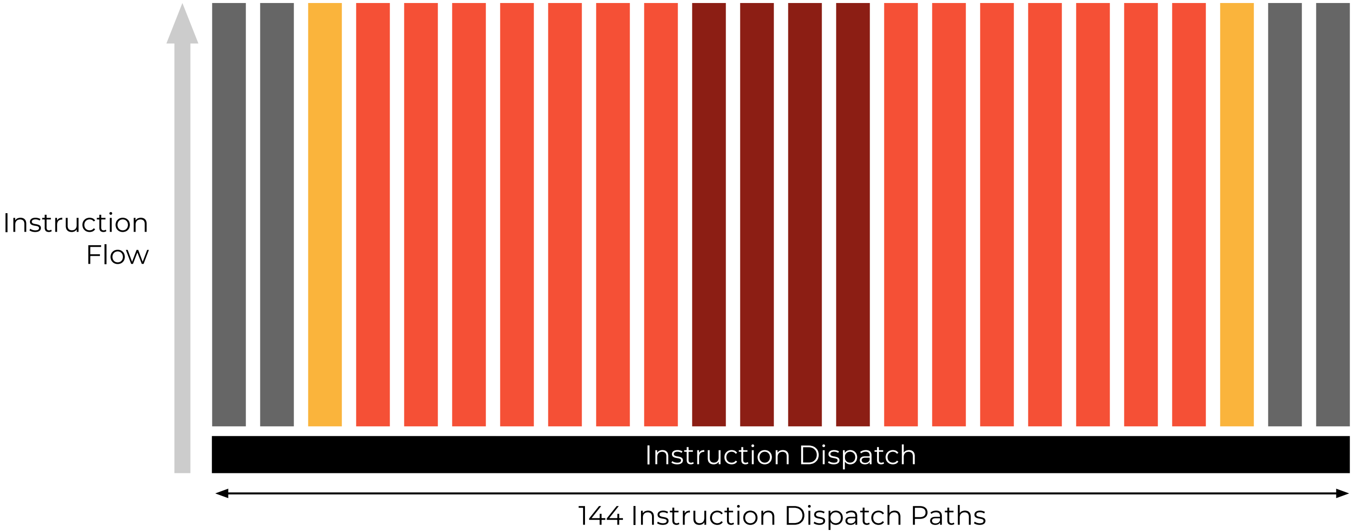
Groq LPU Building Blocks

Lay out SIMD units across chip area



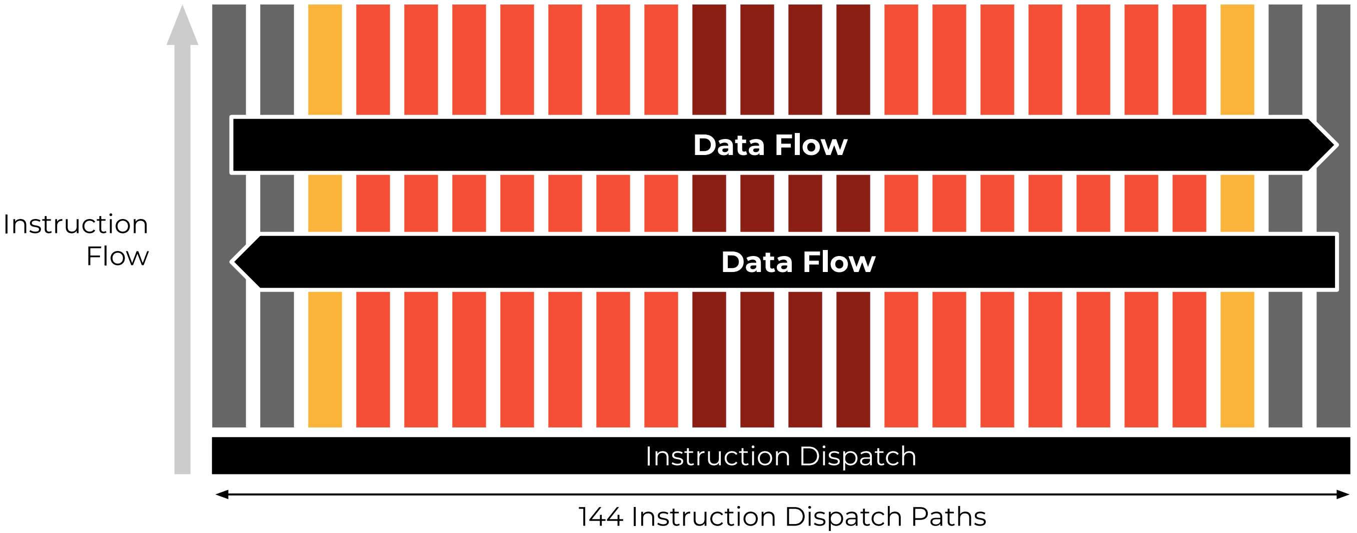
Groq LPU Building Blocks

Synchronized instruction dispatch across all SIMD units for lockstep execution



Groq LPU Building Blocks

High-bandwidth “Stream Registers” for passing data between units



Empowering Groq™ Compiler

Architecture Empowering Software

Software-controlled memory

No dynamic hardware caching

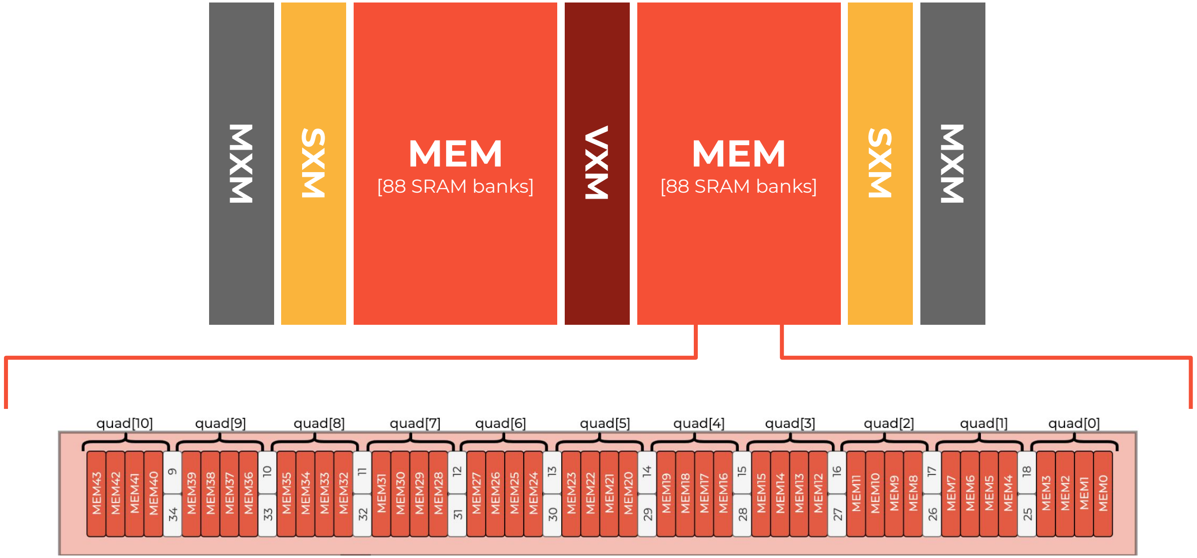
- Compiler aware of all data locations at any given point in time

Flat memory hierarchy (no L1, L2, L3, etc)

- Memory exposed to software as a set of physical banks that are directly addressed

Large on-chip memory capacity (220 MiB) at very high-bandwidth (80 TBps)

- Achieves high compute efficiency even at low operational intensity



Architecture Empowering Software

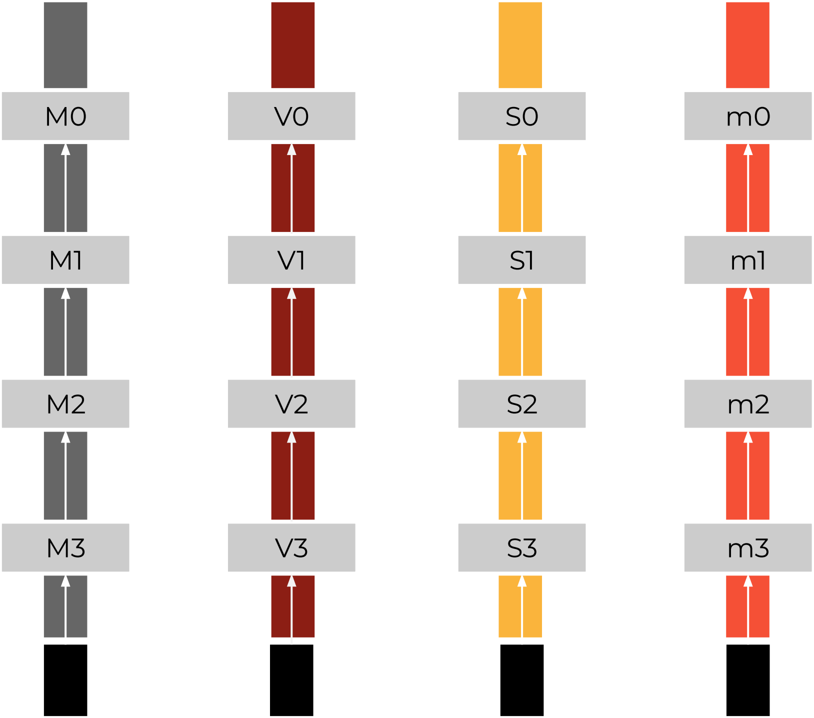
Lockstep execution of Functional Units

Compiler empowered to perform cycle-accurate instruction scheduling

- Synchronous “threads”
- One instruction issued per cycle at each dispatch path

Little hardware control needed for managing instruction execution

- < 3% area overhead for instruction dispatch logic



Architecture Empowering Software

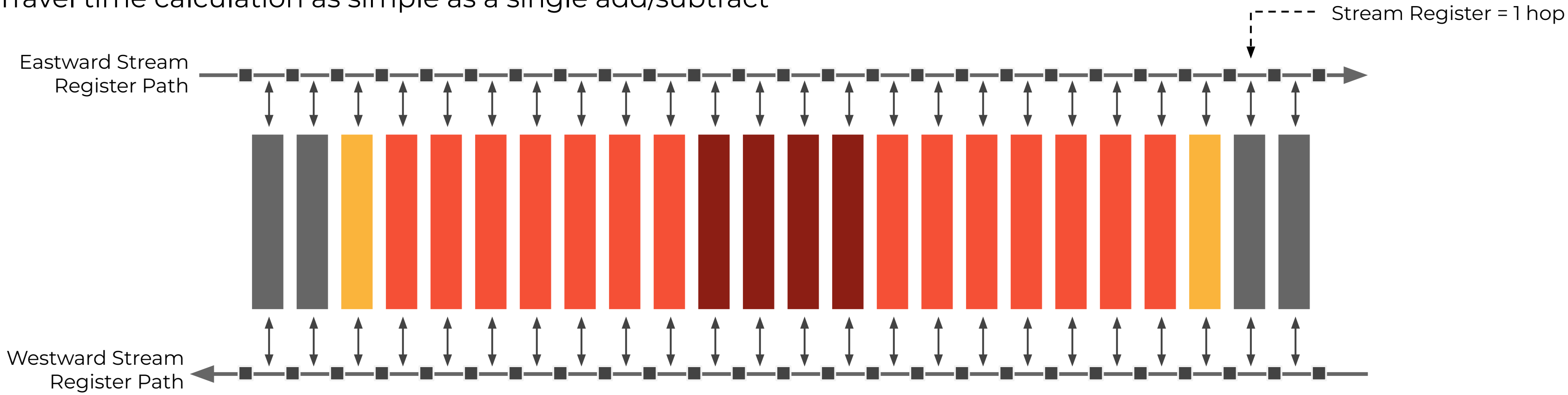
Simple, one-dimensional interconnect for inter-FU communication

Compiler can quickly reason about all data movement between FUs

- Eastward and westward paths made up of arrays of “stream registers”
- Stream register = one-cycle hop

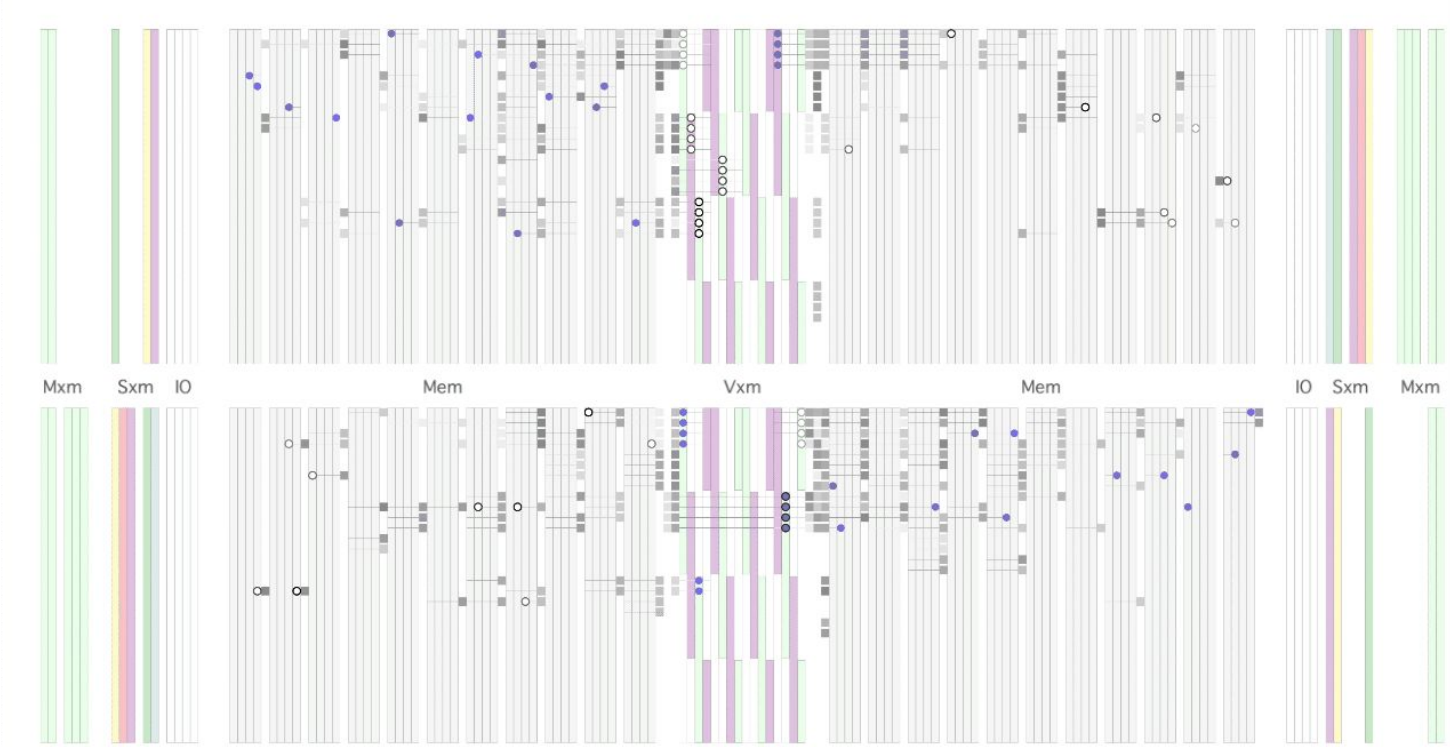
No arbiters / queues = software can easily reason about exact data movement without simulation

Travel time calculation as simple as a single add/subtract



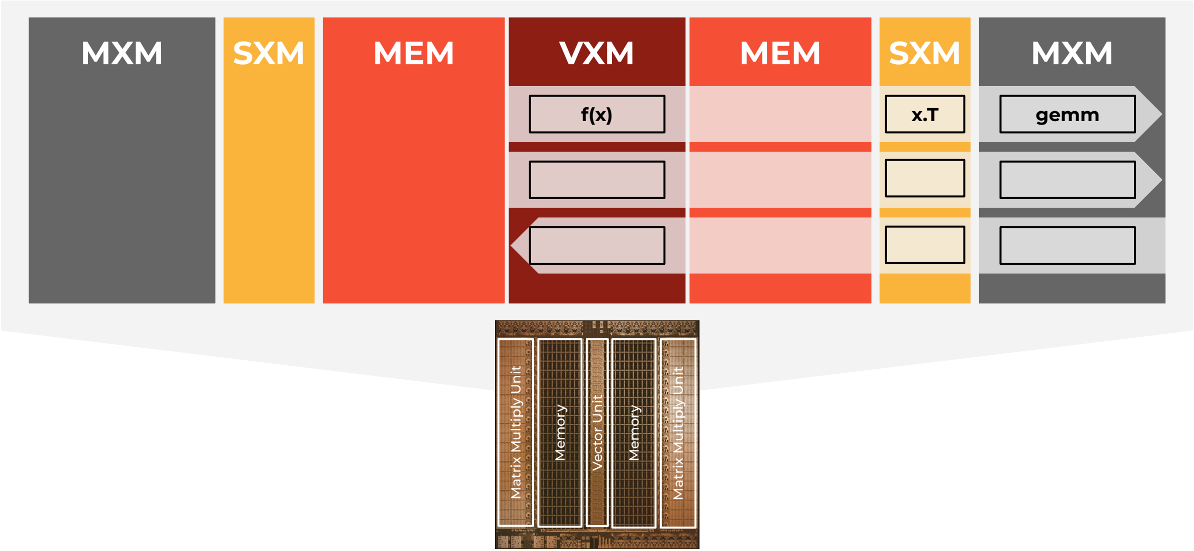
Power of Data Orchestration

Given to Groq Compiler



Groq LPU Functional Units

Tensor Streaming Dataflow



Spatial pipeline processing

Simple tensor instruction set architecture

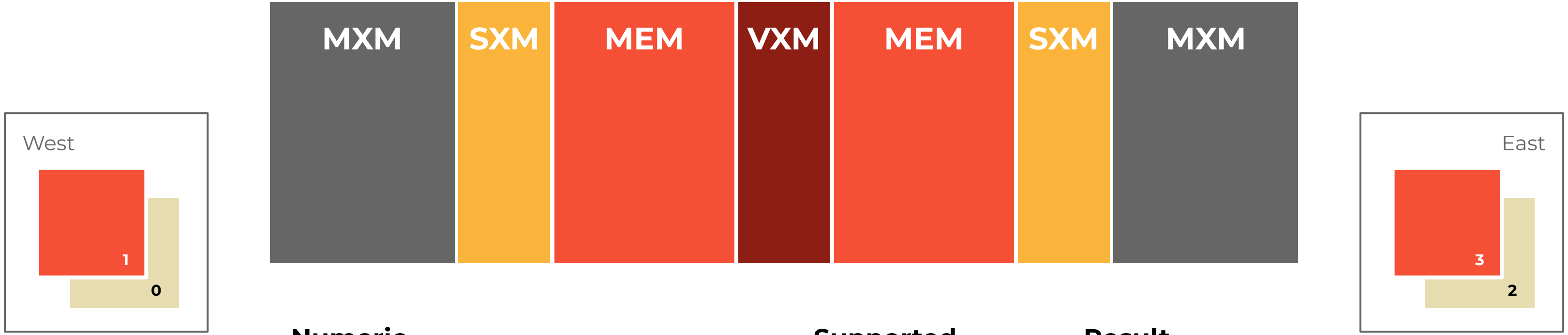
Stream programming of massive SIMD, concurrent streams

Large on-chip memory bandwidth

Deterministic, predictable performance scales to multi-chip

GroqChip™ v1

MXM: Matrix Multiply Engines



Numeric Mode	Size	Supported Density	Result Tensor
int8	[N, 320] x [320, 320]	Two per MXM	int32
float16	[N, 320] x [160, 320]	One per MXM	float32

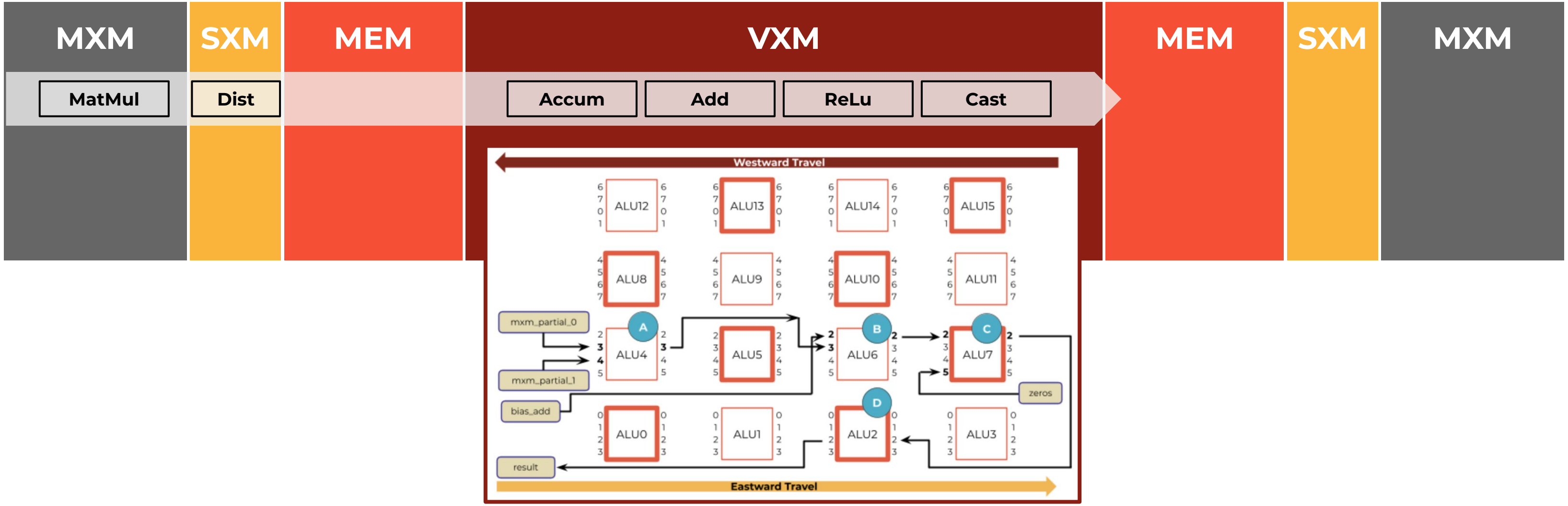
320B x 320B dot product
Loads 320B x16 in 20 cycles
20 cycle execution
Fully pipelined, N

Int8 & float16
Full precision expansion
32-bit accumulate

Used Independently
or together

GroqChip™ v1

VXM: Vector Execution Module



Dataflow begins with memory Read onto Stream Tensor

Many concurrent streams are supported in programming model

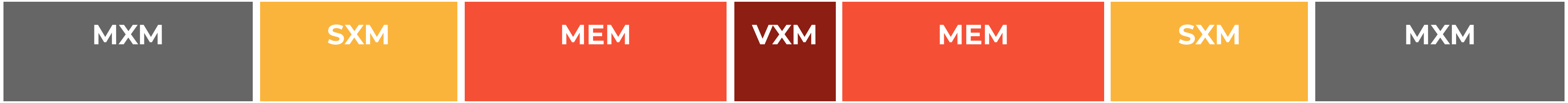
VXM provides a flexible and programmable fabric for Compute

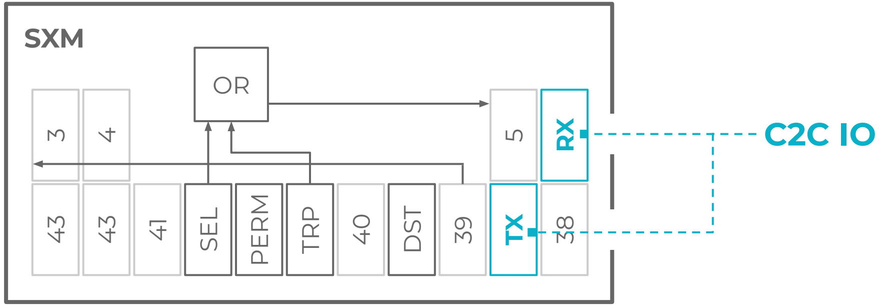
Compute occurs on data locality of passing Stream Tensor

MEM bandwidth supports high concurrency

GroqChip™ v1

SXM: Switch eXecution Module





Swiss army knife for data manipulation & Intra-vector byte operations

Distributor: 4 per hemisphere perform unto mapping of input + mask to output stream within a 16 byte superlane

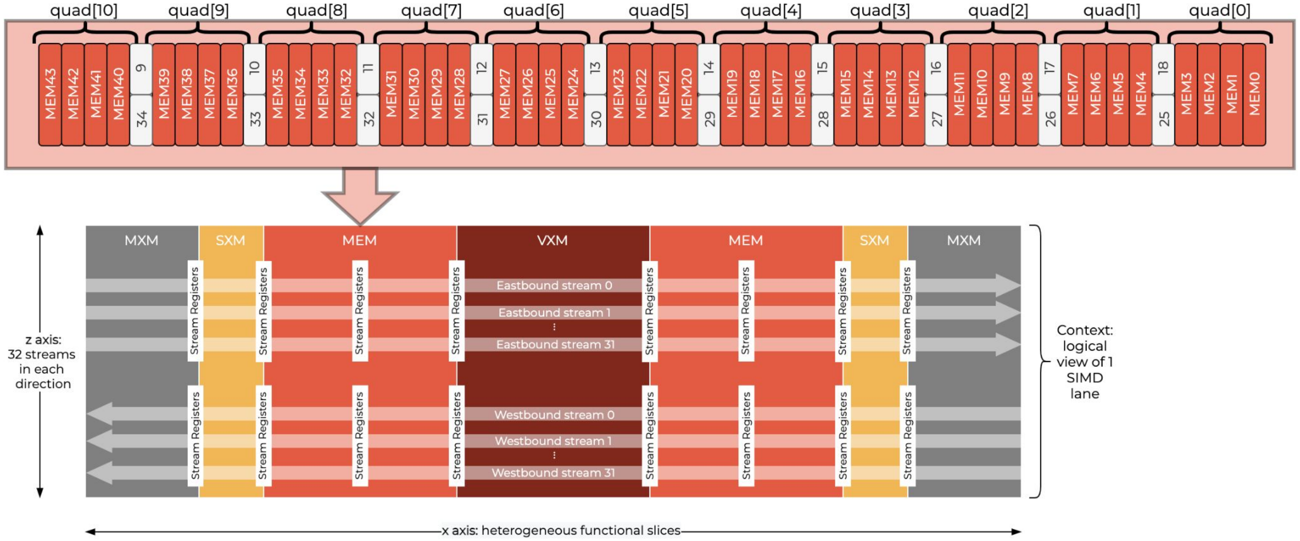
Transposer: 2 per hemisphere perform intra-superlane transpose over 16 vectors for 20 superlanes

Permuter/Shifter: arbitrary mapping of input + mask, shuffling between 320B vector elements - used for data transforms like pads/reshapes

Shift, Rotate, Distribute, Permute, Transpose, Transport to SuperLanes

GroqChip™ v1

MEM: On-Chip SRAM



88 independent MEM slices with 8192 addresses (220MiB) each arranged into quad timing groups

A read from a single MEM slice creates a 320 Byte stream; a write terminates a stream

Group MEM slices for multi-dimensional tensors or multi-byte data types

Can read and write one physical stream (vector) per cycle, from 2 banks; Interfaces the full 64 stream bandwidth @ 80 TBps

Scaling to 1000s of Groq LPUs

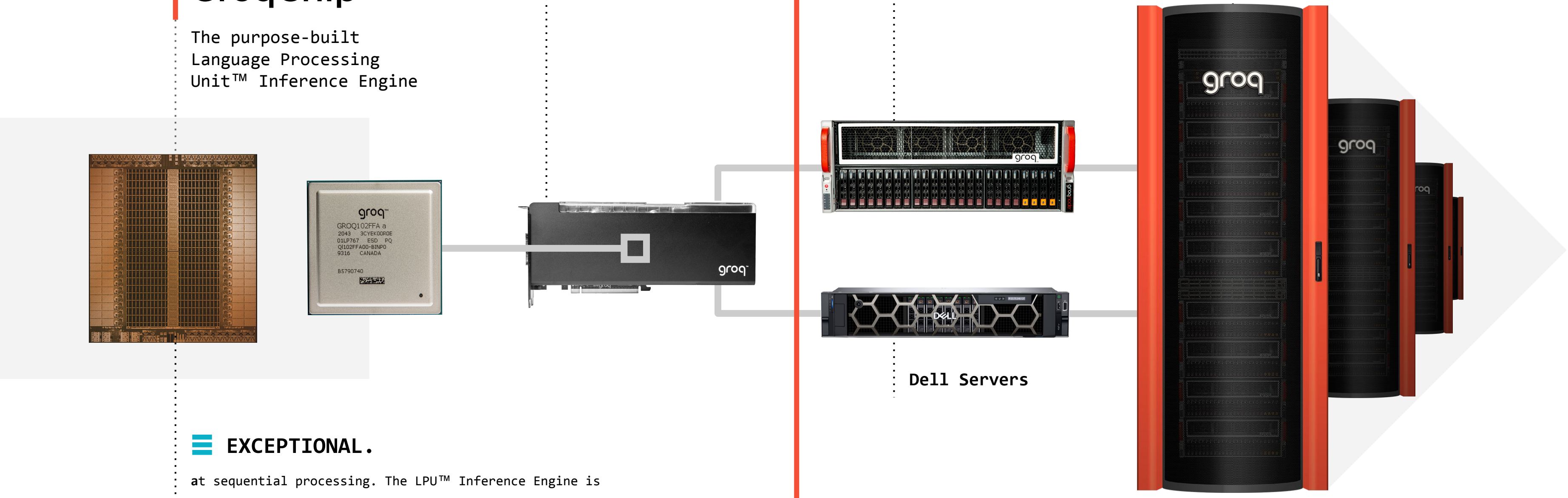
GroqChip™

The purpose-built Language Processing Unit™ Inference Engine

GroqCard™

GroqNode™

GroqRack™



Dell Servers

EXCEPTIONAL.

at sequential processing. The LPU™ Inference Engine is designed to scale and is more power-efficient, with greater performance, than a GPU for AI applications like LLMs.

Software-Scheduled Network

Synchronous Chip-to-Chip communication

Chip-to-Chip (C2C) protocol enables synchronous communication across all LPUs in a network

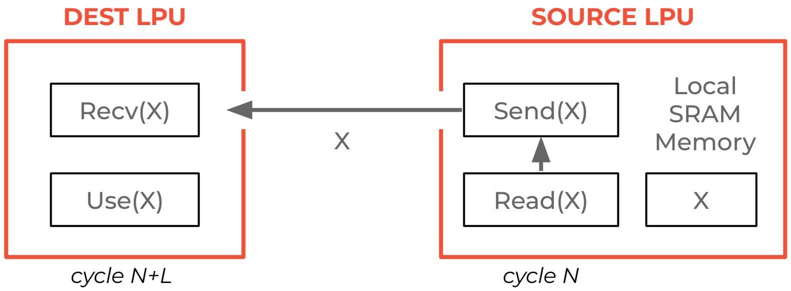
- Clock drift across LPUs is accounted for deterministically

Each LPU acts as both Processor + Router

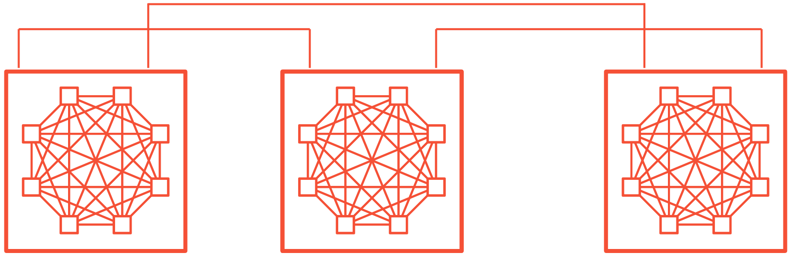
- Compiler schedules network packets as part of programs loaded onto each LPU in the system

No adaptive routing / congestion sensing needed

- Compiler knows exact cycle data should be sent from one LPU and received at another



Software-Scheduled Direct Network



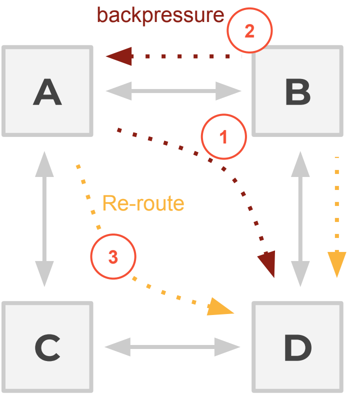
Deterministic Adaptive Routing

Conventional Network

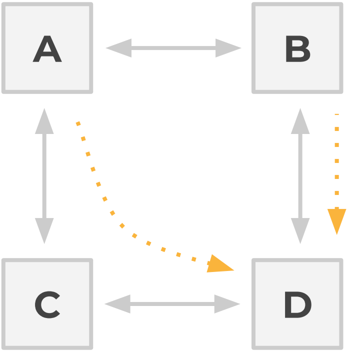
- Commonly done based on network backpressure
- Reactive approach makes the routing decision difficult, increases latency, and increases hardware complexity
- Network latency is **unpredictable**

Software-scheduled Network

- Avoids congestion
- Enables maintaining a deterministic LPU architecture to scale to a multi-node deterministic network execution



Traditional Non-deterministic Network



Software-scheduled Network

Low-diameter Network

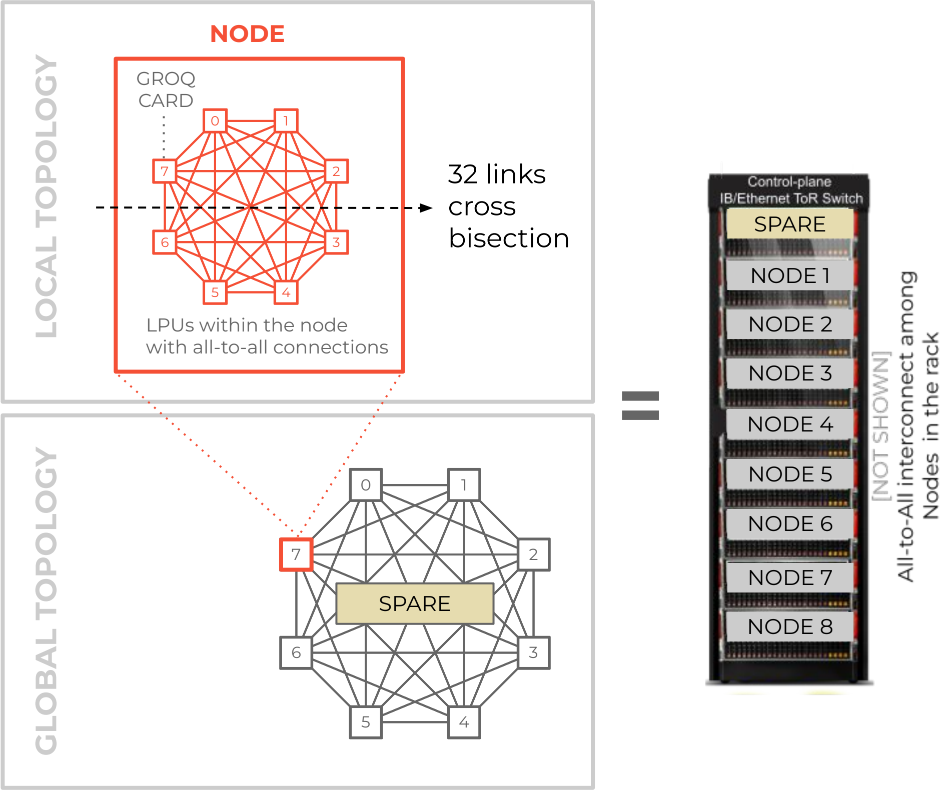
Minimize the number of hops in the network

The total observed latency and variance increases with the number of hops in the network

Dragonfly is a hierarchical topology that minimizes the number of hops taken

- Local group topology
- All-to-all global topology

Exploits packaging locality



AllReduce Comparison Results

Supercomputing Without Barriers

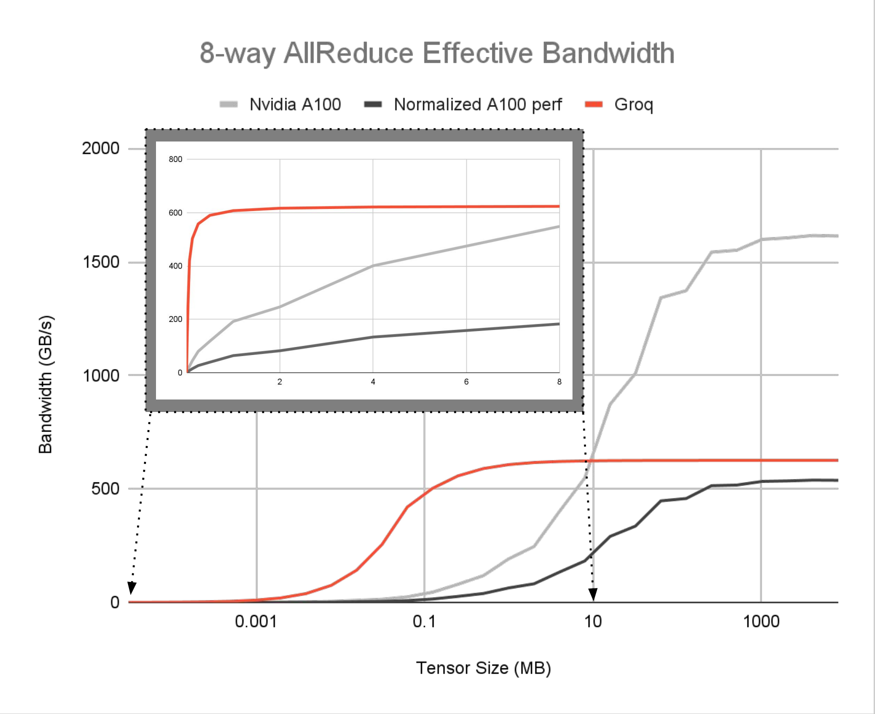
Groq collective communication outperforms state-of-the-art collective AllReduce

Groq RealScale saturates network bandwidth at common message sizes

- Eliminates the need for message aggregation

When normalized, Groq V1 card matches the bandwidth at large tensor size while significantly improving bandwidth at intermediate tensor size

- Comparison made with 8 GPU A100 system with NCCL
- A100 system has approximately 3x higher network channel bandwidth



State-of-the-art

LLM Inference Performance

GroqRack™ Compute Clusters

Recap

Architecture Overview

- Determinism, flat memory hierarchy, 1D interconnect

Key Functional Units

- MXM, VXM, SXM, MEM

Scaling to 1000s of Groq LPUs

- Plesiochronous, low-latency chip-to-chip communication

groq™

Thank You!

abitar@groq.com

Intro to MLAgility™ & GroqFlow™

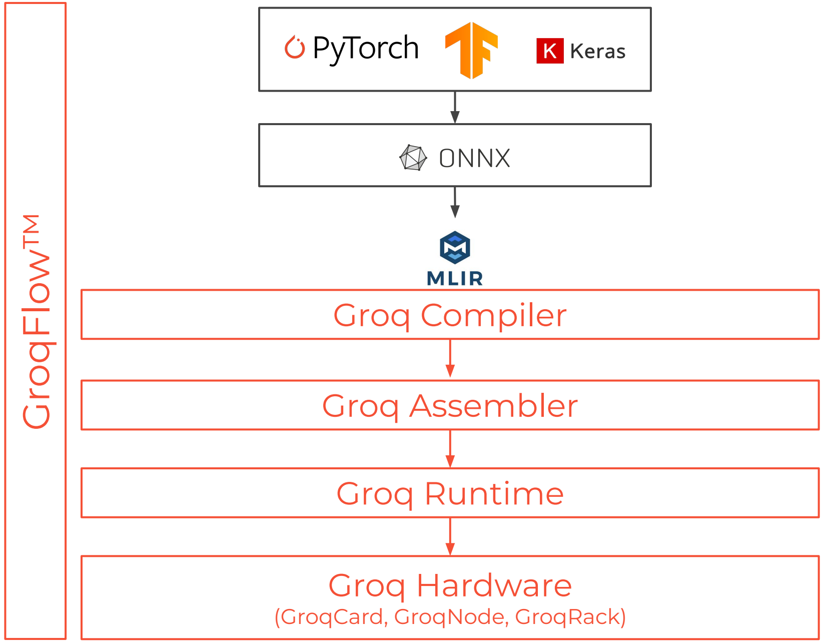
Sanjif Shanmugavelu
Software Engineer

Intro to MLAgility™ & GroqFlow™

AGENDA

1. High Level Software Stack Overview
2. GroqFlow Intro
3. MLAgility Intro

GroqWare™ Suite



DIVERSE SUITE OF DEVELOPMENT TOOLS

Out-of-Box

Groq Compiler provides out-of-box support for standard Deep Learning models

+

Productivity Tools

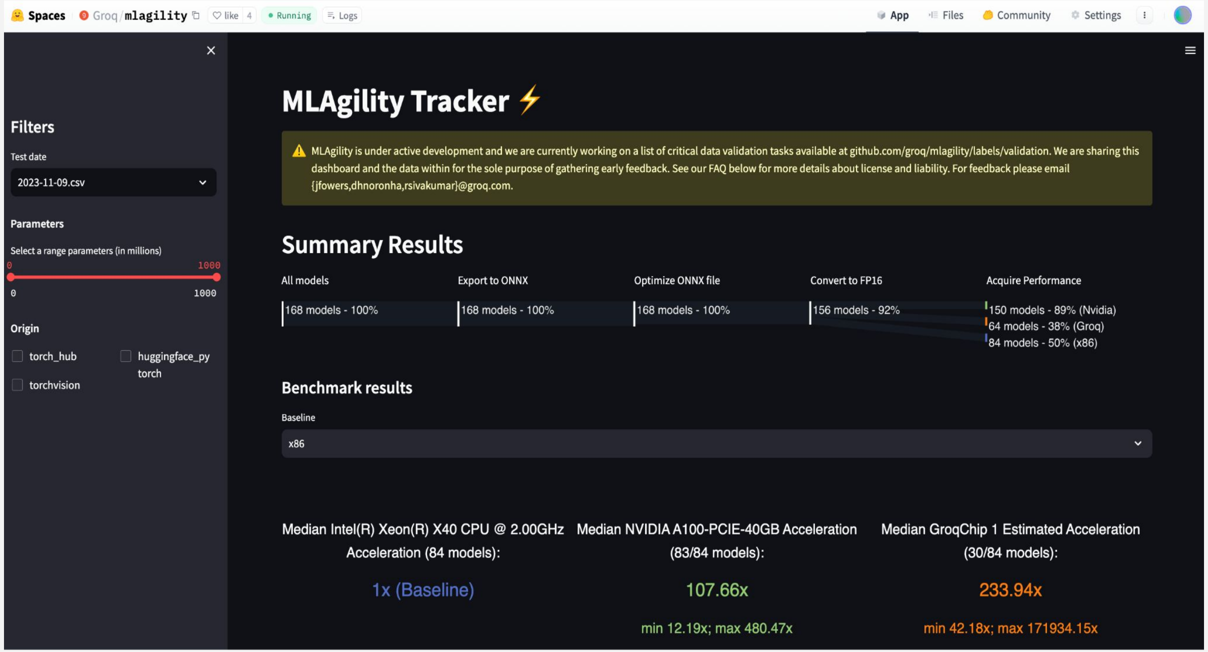
GroqView Profiler provides visualization of the chip's compute and memory usage at compile time

GroqFlow Tool Chain enables a single line of Pytorch or TensorFlow code to import and transform models through a fully automated tool chain to run on Groq hardware

MLAgility

Benchmark performance.

- The *kernelless* Groq™ Compiler supports ML models out-the-box.
- MLAgility is an open-source benchmarking tool, demonstrating model support and performance across a variety of platforms (Groq™, CPU, GPU etc.).
- You can add your own models and benchmarks.
- Groq™ performance on the MLAgility benchmark is reproducible and guaranteed.
- Models are ported to the Groq™ platform with **GroqFlow™**.



Filters

Test date: 2023-11-09.csv

Parameters

Select a range parameters (in millions): 0 – 1000

Origin

□ torch_hub
□ huggingface_pytorch
□ torchvision

MLAgility Tracker ⚡

⚠ MLAgility is under active development and we are currently working on a list of critical data validation tasks available at github.com/groq/mlagility/labels/validation. We are sharing this dashboard and the data within for the sole purpose of gathering early feedback. See our FAQ below for more details about license and liability. For feedback please email {jfowers,dhnoronha,rsivakumar}@groq.com.

Summary Results

All models	Export to ONNX	Optimize ONNX file	Convert to FP16	Acquire Performance
168 models - 100%	168 models - 100%	168 models - 100%	156 models - 92%	150 models - 89% (Nvidia) 64 models - 38% (Groq) 84 models - 50% (x86)

Benchmark results

Baseline: x86

Median Intel(R) Xeon(R) X40 CPU @ 2.00GHz Acceleration (84 models):	Median NVIDIA A100-PCIE-40GB Acceleration (83/84 models):	Median GroqChip 1 Estimated Acceleration (30/84 models):
1x (Baseline)	107.66x	233.94x
	min 12.19x; max 480.47x	min 42.18x; max 171934.15x

Figure 1: Public Groq HuggingFace space

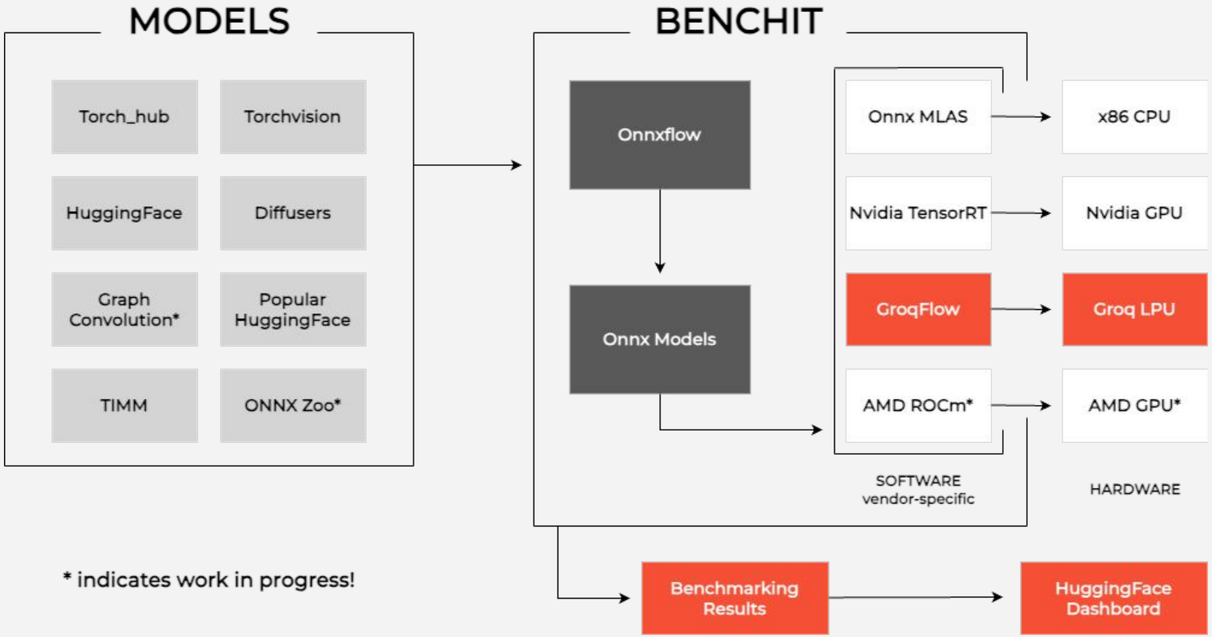
https://huggingface.co/spaces/Groq/mlagility, https://github.com/groq/mlagility

MLAgility Architecture

MLAgility Setup

The diagram illustrates the MLAgility repository structure.

Simply put, the MLAgility models are benchmarked with the benchit tool, and the results are showcased on our Hugging Face Spaces page.



* indicates work in progress!

Recap

- We port models with GroqFlow and benchmark them with MLAgility

Porting Models with GroqFlow™

Sanjif Shanmugavelu
Software Engineer

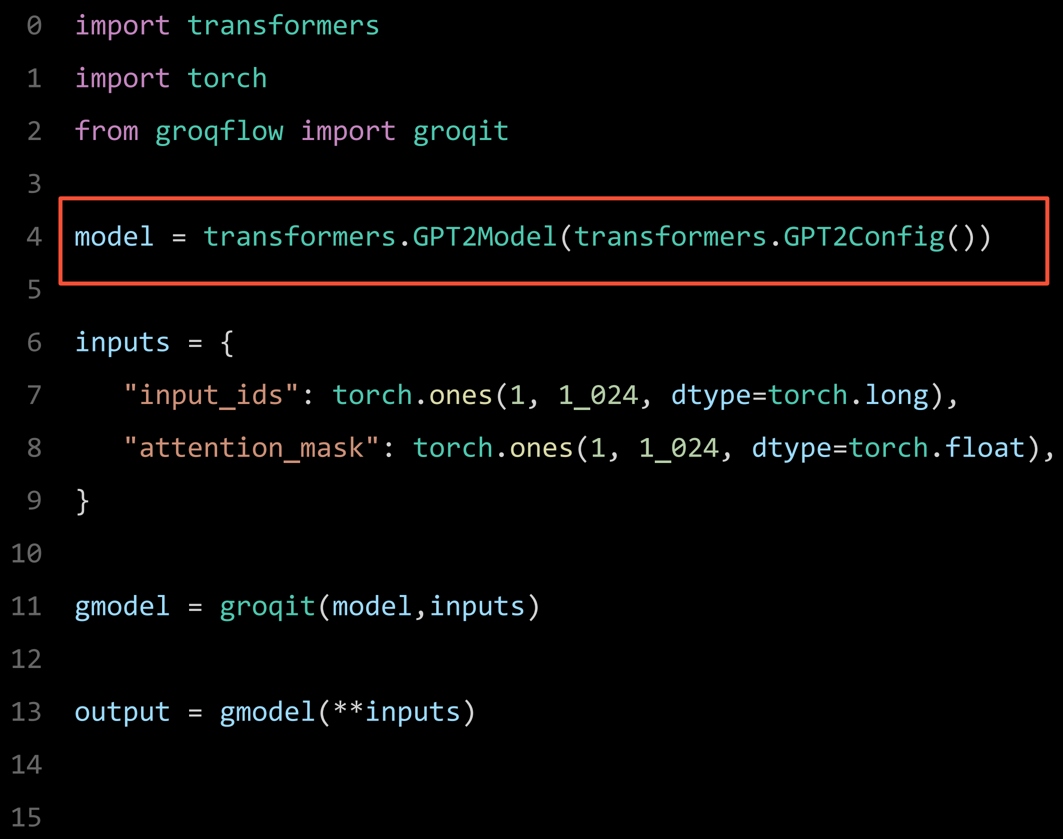
Porting Models with GroqFlow™

AGENDA

1. How To GroqFlow
2. GroqFlow Best Practices
3. GroqFlow Examples
4. Debugging GroqFlow
5. Unwrapping GroqFlow with a ResNet50 Example

Introducing GroqFlow™

Step 1: Get your model



```
import transformers
import torch
from groqflow import groqit

model = transformers.GPT2Model(transformers.GPT2Config())

inputs = {
   "input_ids": torch.ones(1, 1_024, dtype=torch.long),
   "attention_mask": torch.ones(1, 1_024, dtype=torch.float),
}

gmodel = groqit(model,inputs)

output = gmodel(**inputs)
```

*Gold standard of usability: off-the-shelf model from Huggingface.co

Introducing GroqFlow™

Step 2: Get some inputs

```python
import transformers
import torch
from groqflow import groqit

model = transformers.GPT2Model(transformers.GPT2Config())

inputs = {
    "input_ids": torch.ones(1, 1_024, dtype=torch.long),
    "attention_mask": torch.ones(1, 1_024, dtype=torch.float),
}

gmodel = groqit(model,inputs)

output = gmodel(**inputs)
```

*Gold standard of usability: off-the-shelf model from Huggingface.co

Introducing GroqFlow™

Step 3: Just Groq it!

```
import transformers
import torch
from groqflow import groqit

model = transformers.GPT2Model(transformers.GPT2Config())

inputs = {
   "input_ids": torch.ones(1, 1_024, dtype=torch.long),
   "attention_mask": torch.ones(1, 1_024, dtype=torch.float),
}

gmodel = groqit(model,inputs)

output = gmodel(**inputs)


```

```
GroqFlow is building model “bert”
      Converting to ONNX
      Optimizing ONNX file
      Checking for Op support
      Converting to FP16
      Compiling model
      Assembling model
```

*Gold standard of usability: off-the-shelf model from Huggingface.co

Introducing GroqFlow™

Inference is easy!

```python
import transformers
import torch
from groqflow import groqit

model = transformers.GPT2Model(transformers.GPT2Config())

inputs = {
    "input_ids": torch.ones(1, 1_024, dtype=torch.long),
    "attention_mask": torch.ones(1, 1_024, dtype=torch.float),
}

gmodel = groqit(model,inputs)

output = gmodel(**inputs)
```

```
tensor([ 0.3628,  0.0489,  0.2952,  0.0022,
-0.0161, 0.3451, -0.3209, 0.0021, ...
```

*Gold standard of usability: off-the-shelf model from Huggingface.co

Introducing GroqFlow™

Clear messages

What if things don’t go as planned?

Clear feedback on how to move forward

```
GroqFlow is building model “bert”
      Converting to ONNX
      Optimizing ONNX file
      Checking for Op support
      Converting to FP16
      Compiling model
      Assembling model
```

Introducing GroqFlow™

GroqIt Key Functions

PYTORCH
or
ONNX

COMPILES models into Groq programs

```
gmodel = groqit(model,inputs)
```

EXECUTES programs on Groq LPU™

```
gmodel(**inputs)
```

BENCHMARKS programs on Groq LPU

```
latency = gmodel.benchmark()
```

Quick User Guide

GroqIt Args

PYTORCH

Keras

ONNX

gmodel = **groqit(model, inputs)**

Examples:

groqit(my_pytorch_model,inputs)

Main GroqIt Args

model

- Model to be mapped to a GroqModel
- PyTorch model instance or path to an ONNX file

Quick User Guide

GroqIt Args

gmodel = **groqit(model, inputs)**

Bad Example:

```
inputs = tokenizer("I like dogs")
```

Good Example:

```
inputs = tokenizer("I like dogs", padding="max_length",
max_length=128)
```

Main GroqIt Args

model

- Model to be mapped to a GroqModel
- Can be a PyTorch model instance or a path to an ONNX file

inputs

- Dictates the maximum input size the model will support
- Same exact format as your Pytorch inputs
- Hint: Pad your inputs to the right size

Quick User Guide

GroqIt Args

gmodel = **groqit(model, inputs, num_chips)**

Example:

groqit(model, inputs, num_chips=4)

Main GroqIt Args

model

- Model to be mapped to a GroqModel
- Can be a PyTorch model instance or a path to an ONNX file

inputs

- Dictates the maximum input size the model will support
- Same exact format as your Pytorch inputs
- Hint: Pad your inputs to the right size

num_chips

- Number of Groq LPUs to be used
- Automatically selects by default
- 1, 2 or 4 chips are valid for A1.1 (1, 2, 4, 8 for A1.4)

Quick User Guide

GroqIt Args

gmodel = **groqit(model, inputs, rebuild)**

Rebuild a model every time:

groqit(model, inputs, rebuild=”always”)

Use cached model if available:

groqit(model, inputs, rebuild=”never”)

Main GroqIt Args

model

- Model to be mapped to a GroqModel
- Can be a PyTorch model instance or a path to an ONNX file

inputs

- Dictates the maximum input size the model will support
- Same exact format as your Pytorch inputs
- Hint: Pad your inputs to the right size

num_chips

- Number of Groq LPUs to be used
- Automatically selects by default
- 1, 2 or 4 chips are valid for A1.1 (1, 2, 4, 8 for A1.4)

rebuild

- GroqIt loads successfully built models by default
- Set rebuild to “always” to force GroqIt to rebuild it

Quick User Guide

GroqIt Args

gmodel = **groqit(model, inputs, build_name)**

Example:

groqit(modelA, inputsA, build_name="A") → Builds modelA
groqit(modelB, inputsB, build_name="B") → Builds modelB

Main GroqIt Args

model

- Model to be mapped to a GroqModel
- Can be a PyTorch model instance or a path to an ONNX file

inputs

- Dictates the maximum input size the model will support
- Same exact format as your Pytorch inputs
- Hint: Pad your inputs to the right size

num_chips

- Number of GroqChip processors to be used
- Automatically selects by default
- 1, 2 or 4 chips are valid for A1.1 (1, 2, 4, 8 for A1.4)

rebuild

- GroqIt loads successfully built models by default
- Set rebuild to "always" to force GroqIt to rebuild it

build_name

- Name used to cache the model
- Defaults to the name of the script

Quick User Guide

Groq Model Functions

```
gmodel = groqit(model, inputs)
gmodel(**inputs)
```

Example:

```
>>> pytorch_model(**inputs)
  tensor([0.245, 0.235, 0.235, 0.267])

>>> gmodel(**inputs)
  tensor([0.245, 0.235, 0.235, 0.267])
```

Main Groq Model Functions

inference/forward pass

- The Groq Model is callable like a Pytorch model
- Performing inference doesn't require rebuilding
- Hint: Pad your inputs to the same shape used when creating the model

Note: Not useful for timing purposes, since the entire Groq environment is setup each time

Quick User Guide

Groq Model Functions

gmodel = groqit(model, inputs)

gmodel.benchmark()

(coming soon)

Example:

```
>>> latency = gmodel.benchmark()
>>> print(f"Latency is {latency}ms")
  Latency is 0.109ms
```

Main Groq Model Functions

inference/forward pass

- The Groq Model is callable like a Pytorch model
- Performing inference doesn't require rebuilding
- Hint: Pad your inputs to the same shape used when creating the model

  Note: Not useful for timing purposes, since the entire Groq environment is setup each time

benchmark

- Returns the average latency of 100 runs in ms
- Latency includes PCIe times + on-chip compute

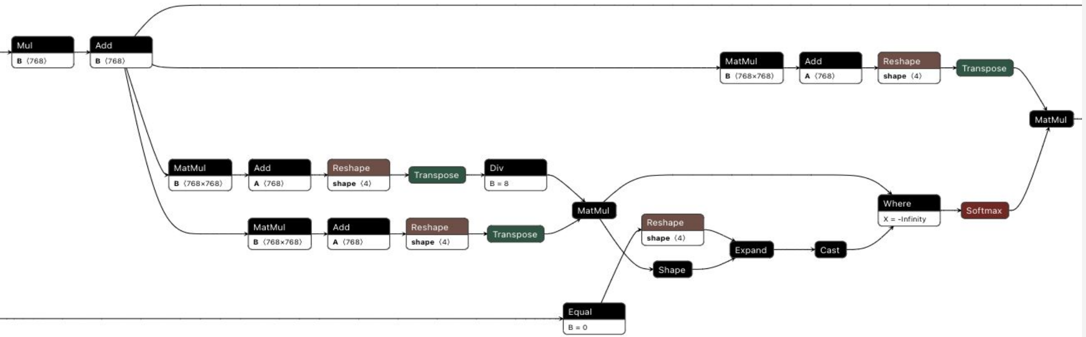
Quick User Guide

Groq Model Functions

gmodel = groqit(model, inputs)

gmodel.netron()

Example:



Main Groq Model Functions

inference/forward pass

- The Groq Model is callable like a Pytorch model
- Performing inference doesn't require rebuilding
- Hint: Pad your inputs to the same shape used when creating the model

  Note: Not useful for timing purposes, since the entire Groq environment is setup each time

benchmark

- Returns the average latency of 100 runs in ms
- Latency includes PCIe times + on-chip compute

netron

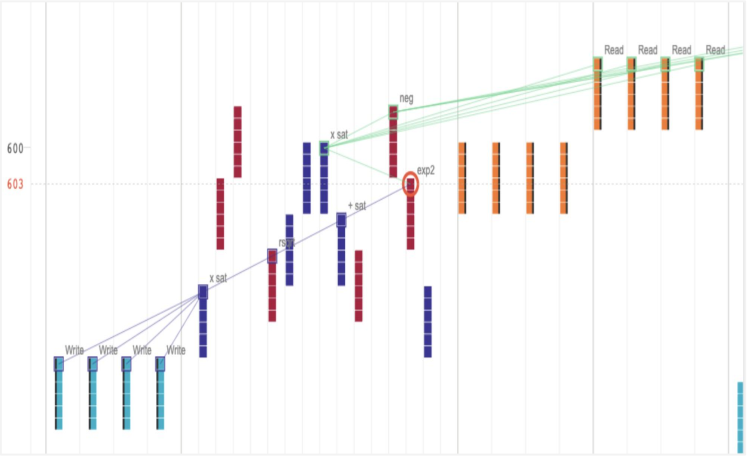
- Opens the ONNX model generated by GroqIt

Quick User Guide

Groq Model Functions

```
gmodel = groqit(model, inputs)
gmodel.groqview()
```

Example:



Main Groq Model Functions

inference/forward pass

- The Groq Model is callable like a Pytorch model
- Performing inference doesn't require rebuilding
- Hint: Pad your inputs to the same shape used when creating the model

  Note: Not useful for timing purposes, since the entire Groq environment is setup each time

benchmark

- Returns the average latency of 100 runs in ms
- Latency includes PCIe times + on-chip compute

netron

- Opens the ONNX model generated by GroqIt

groqview

- Opens the GroqView profiler generated by GroqIt

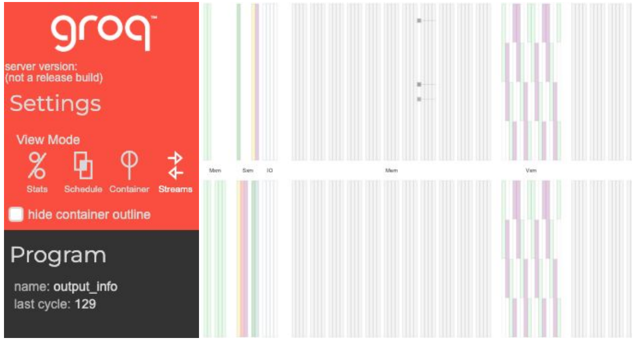
Quick User Guide

Groq Model Functions

gmodel = groqit(model, inputs,groq_view=True)

gmodel.groqview()

Example:



server version:
(not a release build)

Settings

View Mode

□ hide container outline

Program

name: output_info
last cycle: 129

Main Groq Model Functions

inference/forward pass

- The Groq Model is callable like a Pytorch model
- Performing inference doesn't require rebuilding
- Hint: Pad your inputs to the same shape used when creating the model

  Note: Not useful for timing purposes, since the entire Groq environment is setup each time

benchmark (coming soon)

- Returns the average latency of 100 runs in ms
- Latency includes PCIe times + on-chip compute

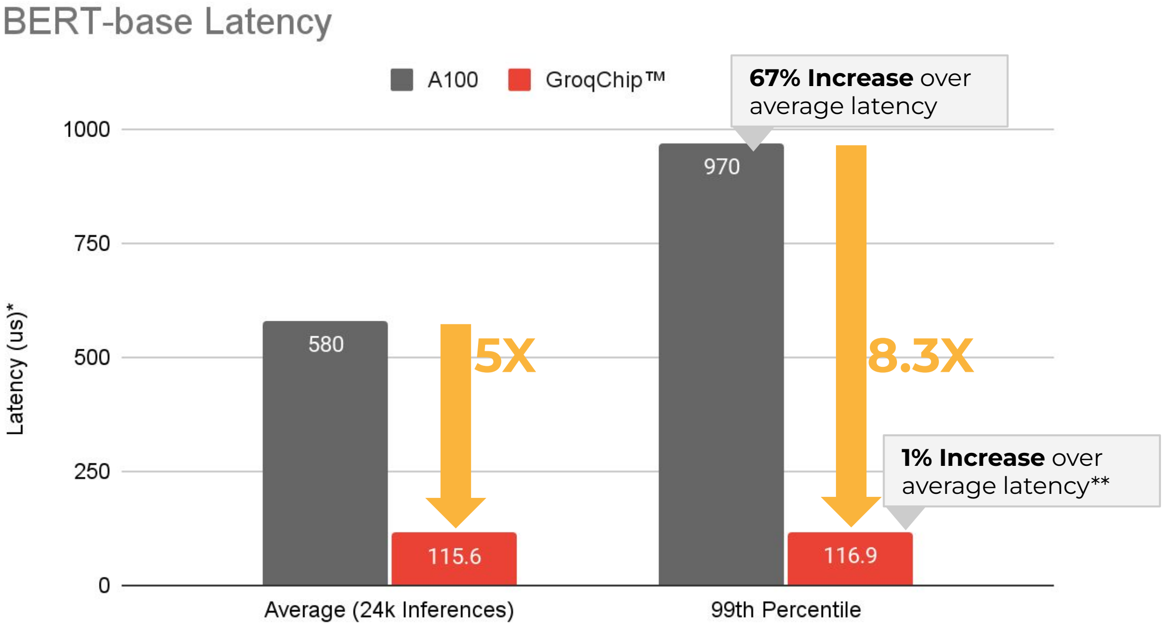
netron

- Opens the ONNX model generated by GroqIt

groqview

- Visualize data streams and execution schedule
- Requires compiling with groq_view flag

Low Latency Every Time



Groq Advantages

Determinism

Low Latency

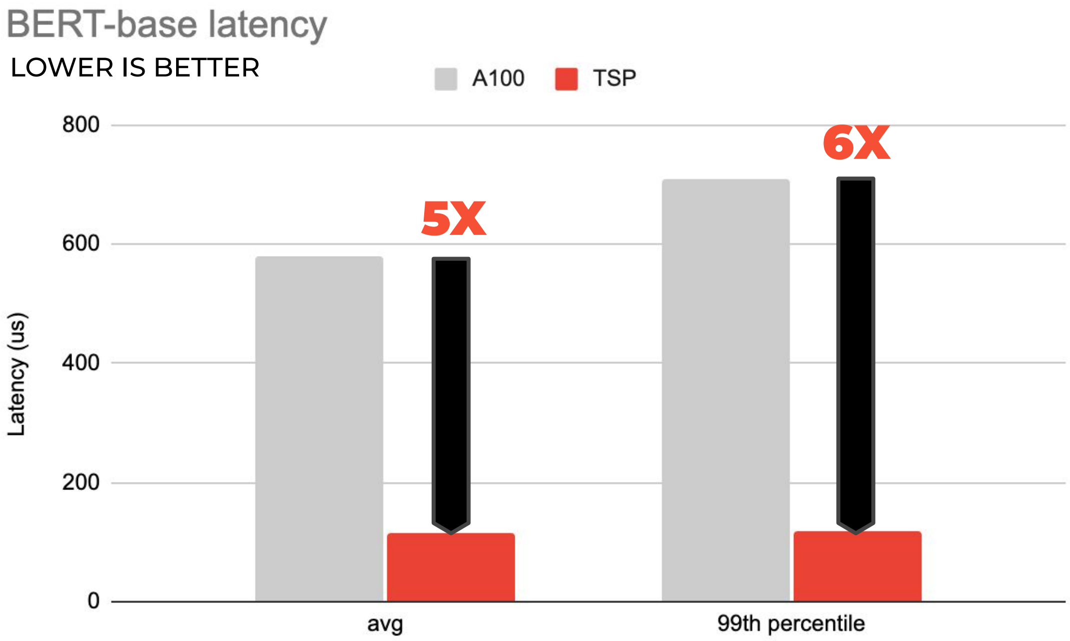
Large On-chip Memory

Groq LPU delivers up to **8.3X** better performance on the slowest inference

Nvidia results from publicly available data on github.com/NVIDIA (Batch size-1 on TensorRT v8.0.1.6)
*Lower is better
**Increase is limited to host and PCIe IO variance

BERT

Groq accelerated BERT inference to achieve a 99th percentile latency of **117 µs**



Ahmed, I., et al. "Answer Fast: Accelerating BERT on the Tensor Streaming Processor." ASAP'22.
Nvidia results from latest publicly available data (TensorRT v8.4.3)

Best Practices

- GroqFlow is a wrapper around the GroqWare™ Suite that gives you the power to quickly compile and run models.
- Pad to the maximum input dimensions
- Avoid dynamism and control flow (for now..)

Recap

- GroqFlow is a wrapper around the GroqWare™ Suite that gives you the power to quickly compile and run models.

Benchmarking Models with MLAgility™

Sanjif Shanmugavelu
Software Engineer

Benchmarking Models with MLAgility™

AGENDA

1. MLAgility Devices and Runtimes
2. MLAgility *benchit* CLI
3. Writing Scripts with MLAgility
4. MLAgility Report Generation and Visualization
5. MLAgility Future Work

MLAgility Devices and Runtimes

Benchmark setup

```
MLAgility's tools currently support the following combinations of runtimes and devices. We leverage ONNX files because of their broad
compatibility with model frameworks (PyTorch, Keras, etc.), software (ONNX Runtime, TensorRT, Groq Compiler, etc.), and devices
(CPUs, GPUs, Groq LPUs, etc.)
```

Device Type	Device arg	Runtime	Runtime arg	Specific Devices
Nvidia GPU	nvidia	TensorRT†	trt	Any Nvidia GPU supported by TensorRT
x86 CPU	x86	ONNX Runtime‡ Pytorch Eager§ Pytorch 2.x Compiled*§	ort, torch-eager, torch-compiled	Any Intel or AMD CPU supported by the runtime
Groq	Groq	GroqFlow	Groq	GroqChip1

† Requires TensorRT >= 8.5.2 ‡ Requires ONNX Runtime >= 1.13.1 * Requires Pytorch >= 2.0.0 § Only available on local backend

MLAgility CLI

Benchmark with benchit

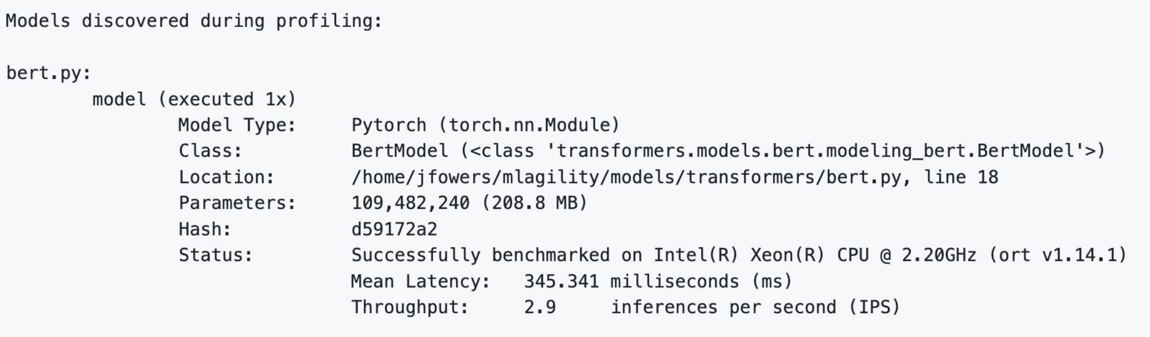
The MLAgility Benchmarking and Tools package provides a CLI, benchit, and Python API for benchmarking ML models

Let’s benchmark the popular BERT transformer model with benchit:

```
benchit models/transformers/bert.py
-device {groq, nvidia x86, }
```

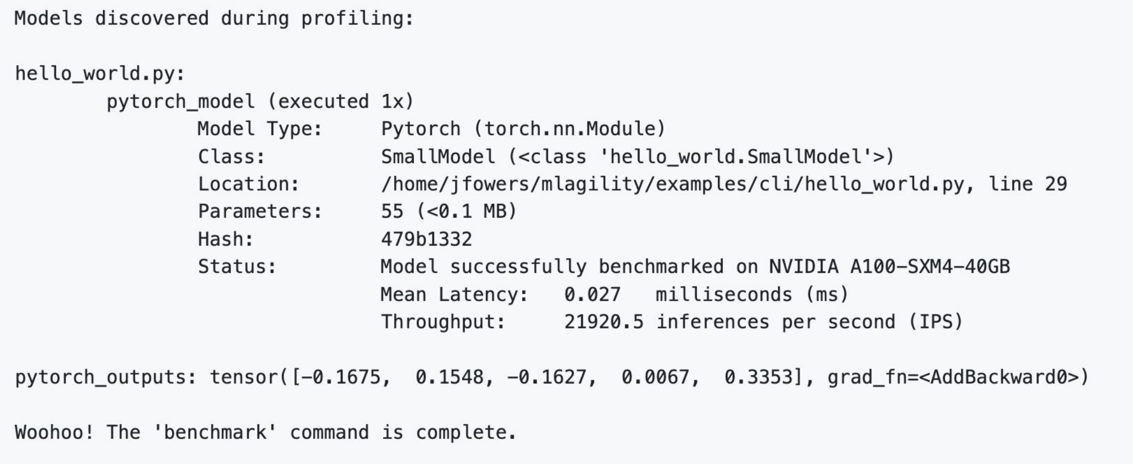
The device flag specifies the benchmark hardware. The output is saved in the user `.cache/mlagility directory`

-device x86



```
Models discovered during profiling:

bert.py:
        model (executed 1x)
                Model Type:     Pytorch (torch.nn.Module)
                Class:          BertModel (<class 'transformers.models.bert.modeling_bert.BertModel'>)
                Location:       /home/jfowers/mlagility/models/transformers/bert.py, line 18
                Parameters:     109,482,240 (208.8 MB)
                Hash:           d59172a2
                Status:         Successfully benchmarked on Intel(R) Xeon(R) CPU @ 2.20GHz (ort v1.14.1)
                                Mean Latency:   345.341 milliseconds (ms)
                                Throughput:     2.9     inferences per second (IPS)
```

-device nvidia



```
Models discovered during profiling:

hello_world.py:
        pytorch_model (executed 1x)
                Model Type:     Pytorch (torch.nn.Module)
                Class:          SmallModel (<class 'hello_world.SmallModel'>)
                Location:       /home/jfowers/mlagility/examples/cli/hello_world.py, line 29
                Parameters:     55 (<0.1 MB)
                Hash:           479b1332
                Status:         Model successfully benchmarked on NVIDIA A100-SXM4-40GB
                                Mean Latency:   0.027   milliseconds (ms)
                                Throughput:     21920.5 inferences per second (IPS)

pytorch_outputs: tensor([-0.1675,  0.1548, -0.1627,  0.0067,  0.3353], grad_fn=<AddBackward0>)

Woohoo! The 'benchmark' command is complete.
```

https://github.com/groq/mlagility/blob/main/docs/code.md

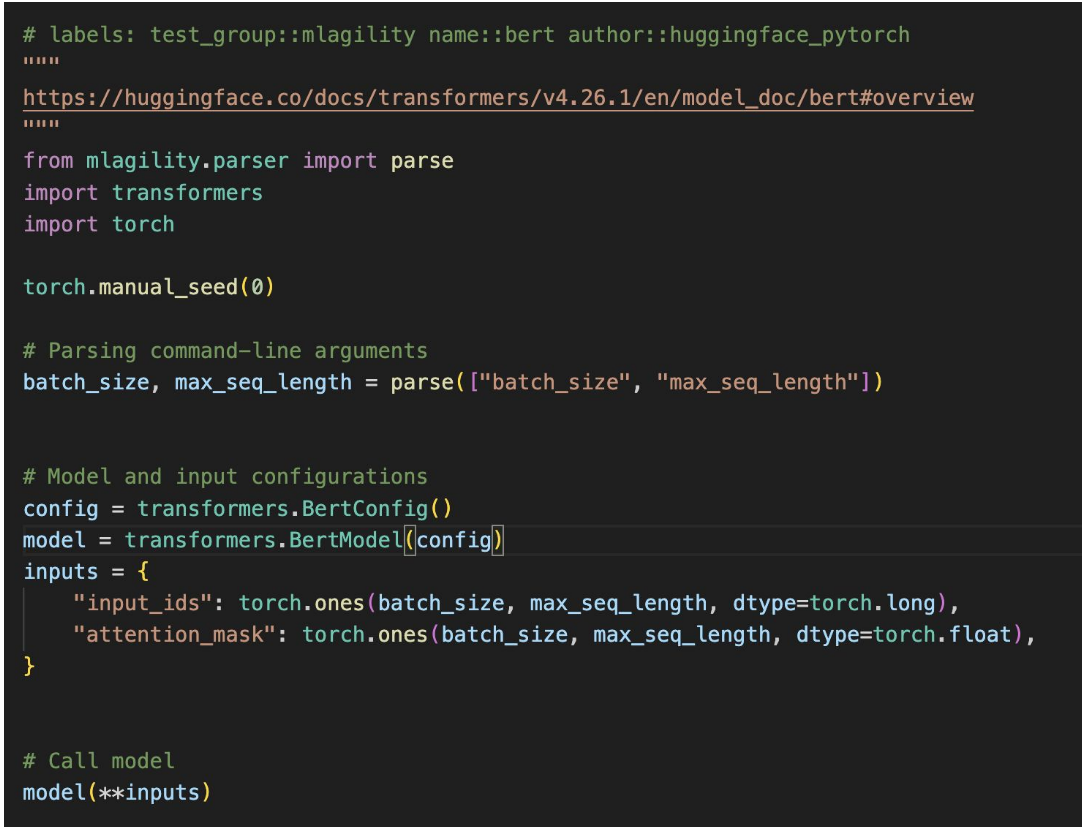
MLAgility Input

How to write a benchmark script

The following example, copied from `models/transformers/bert.py` is a sample input script for the MLAgility benchmark

It has the following properties:

- Labels in the top line of the file
- Docstring indicating where the model was sourced from
- `mlagility.parser.parse()` is used to parameterize the model
- The model is instantiated and invoked against a set of inputs



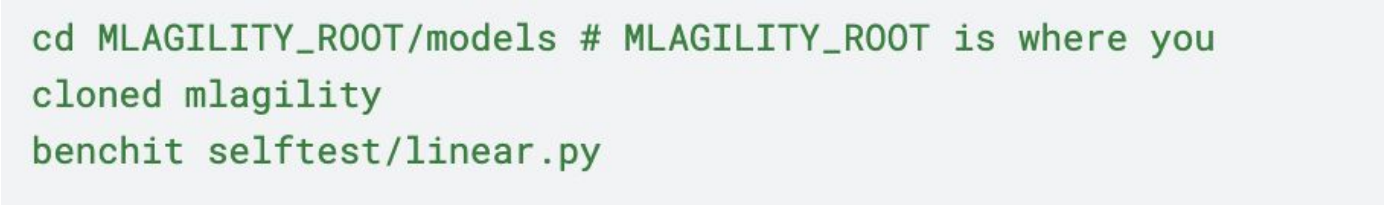
```python
# labels: test_group::mlagility name::bert author::huggingface_pytorch
"""
https://huggingface.co/docs/transformers/v4.26.1/en/model_doc/bert#overview
"""
from mlagility.parser import parse
import transformers
import torch

torch.manual_seed(0)

# Parsing command-line arguments
batch_size, max_seq_length = parse(["batch_size", "max_seq_length"])


# Model and input configurations
config = transformers.BertConfig()
model = transformers.BertModel(config)
inputs = {
    "input_ids": torch.ones(batch_size, max_seq_length, dtype=torch.long),
    "attention_mask": torch.ones(batch_size, max_seq_length, dtype=torch.float),
}


# Call model
model(**inputs)
```

https://github.com/groq/mlagility/tree/main/models#example-script

MLAgility Full Benchmark

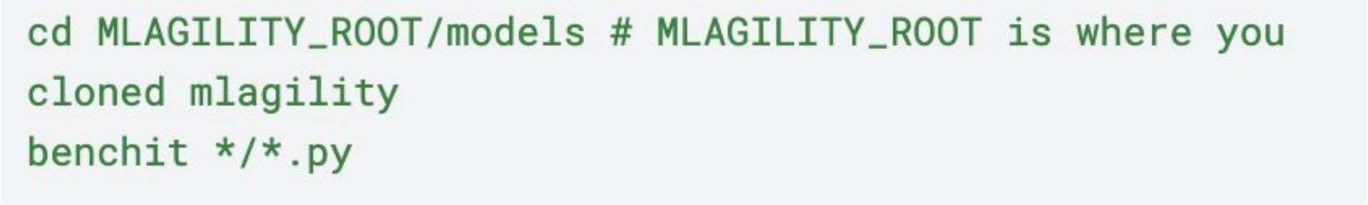
Automated push-button benchmarking

Once you have fulfilled the prerequisites, you can evaluate one model from the benchmark with a command like this:



```
cd MLAGILITY_ROOT/models # MLAGILITY_ROOT is where you
cloned mlagility
benchit selftest/linear.py
```

You can also run the entire MLAgility benchmark in one shot with:



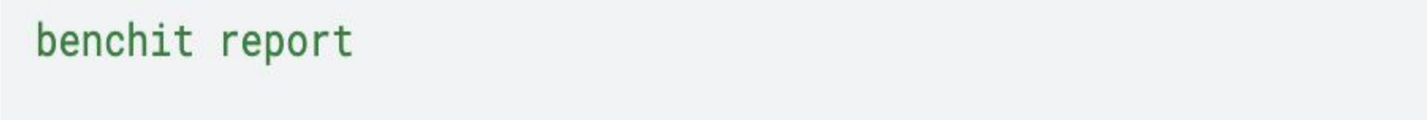
```
cd MLAGILITY_ROOT/models # MLAGILITY_ROOT is where you
cloned mlagility
benchit */*.py
```

Note: Benchmarking the entire corpora of MLAgility models might take a very long time

MLAgility Report Generation

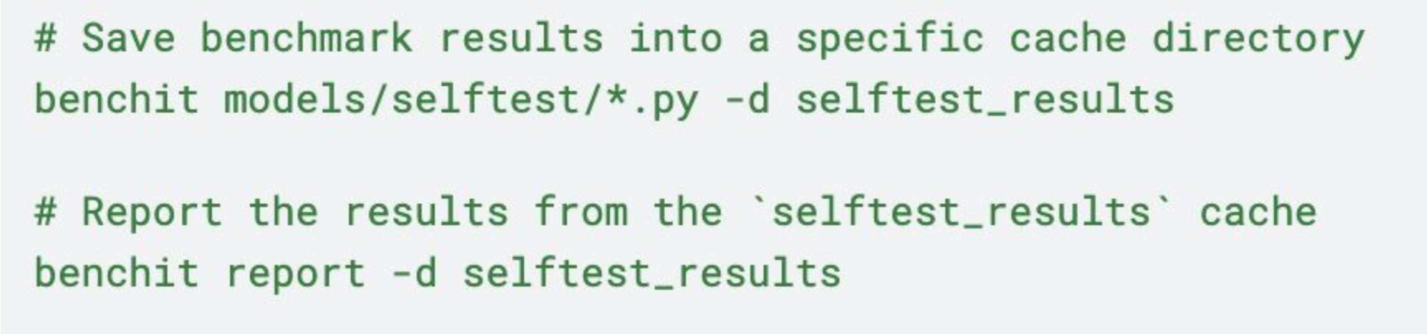
Collect and present results

You can aggregate all of the benchmarking results from your mlagility cache into a CSV file with:



```
benchit report
```

If you want to only report on a subset of models, we recommend saving the benchmarking results into a specific cache directory:



```
# Save benchmark results into a specific cache directory
benchit models/selftest/*.py -d selftest_results

# Report the results from the `selftest_results` cache
benchit report -d selftest_results
```

`By default, all results are saved in /home/{$USER}/.cache/mlagility)`

Tp present visually, use out streamlit powered huggingface directory
https://github.com/groq/mlagility/tree/main/trackers/huggingface

MLAgility Limitations and Future Work

To infinity and beyond

Current Limitations / Constraints:

Groq's latency is computed using `GroqModel.estimate_latency()`

Takes into account deterministic compute time and estimates an ideal runtime with ideal I/O time

It does not take into account runtime performance

Results currently only represent batch 1 performance

Limited number of models, devices, vendors, and runtimes

MLAgility Limitations and Future Work

To infinity and beyond

Future work:

Include additional classes of models

Experiments that include sweeps over batch and input sizes

Include operator microbenchmarks

Increase the number of devices from existing vendors

Include devices from additional vendors and number of runtimes supported

Recap

- MLAgility is a fully open-source benchmarking tool to benchmark acceleration hardware and runtimes.

groq™

Thank You!

sshanmugavelu@groq.com

groq™

Thank You!

iarsovski@groq.com