



Groq AI Workshop

ALCF AI Testbed



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 Tensor Streaming Processor™ 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, Sr. Staff Compiler Engineer
Intro to MLAGility™ and GroqFlow™	Introduction to the MLAGility Project's HuggingFace Space and the GroqFlow toolchain used to port models.	15 mins	Sanjif Shanmugavelu, Software Engineer
Porting Models with GroqFlow™	Step-by-step walkthrough of model porting with GroqFlow for execution on GroqRack (including best practices).	45 mins	Sanjif Shanmugavelu, Software Engineer
Benchmarking Models with MLAGility™	How to benchmark multiple models with MLAGility.	45 mins	Sanjif Shanmugavelu, Software Engineer
Accessing GroqRack™ at ALCF AI Testbed	How to access GroqRack.	20 mins	ALCF Staff

Welcome to Groq

Jonathan Ross
Founder & CEO



Groq Tensor Streaming Processor™ Architecture

Andrew Bitar

Sr. Staff Compiler Engineer

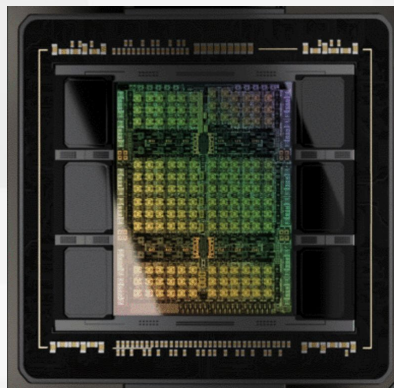
Groq Tensor Streaming Architecture

AGENDA

1. Architecture Overview
2. Key Functional Units
3. Scaling to 1000s of GroqChip™ Processors



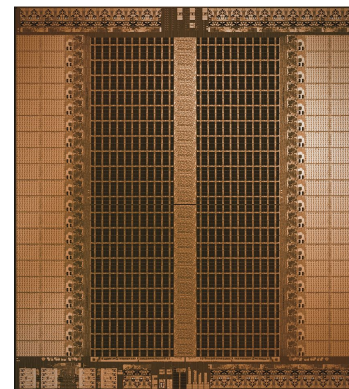
Groq Simplifies Compute



Graphics Processor (GPU)

COMPLEX

- Non-deterministic execution
- Difficult to program
- Higher latency
- Higher costs



Tensor Streaming Processor (TSP)

SIMPLIFIED

- Deterministic / Predictable execution
- Easier compilation
- Lower latency
- Higher efficiency at scale

THESIS
Predictable Compute
Needs Predictable
Hardware.

The Missing Middle

Algorithms — Compilers —> Hardware

Dataflow
dominated

Statically predictable set
of executed operations

Highly-parallel
vector operations

Remain a challenge

Reliant on hand-tuned
libraries

Fragmented front-end
ecosystem

Require iterative
hardware profiling

High-density compute
using SIMD

Less silicon area spent
on re-ordering and
speculation

More memory
bandwidth

✓ PREDICTABLE

✗ UNPREDICTABLE

GroqChip™ Overview

SRAM Memory

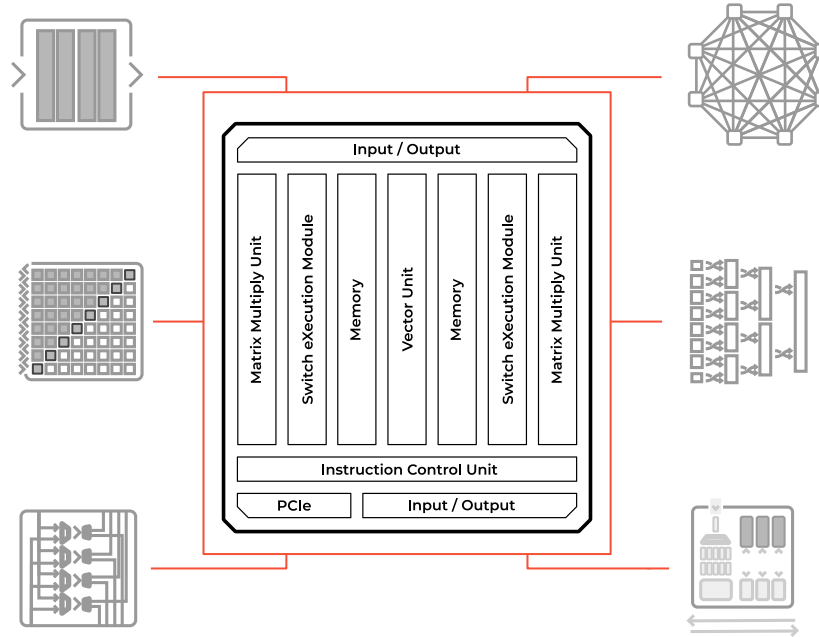
Massive concurrency
80 TB/s of BW
230MB capacity
Stride insensitive

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

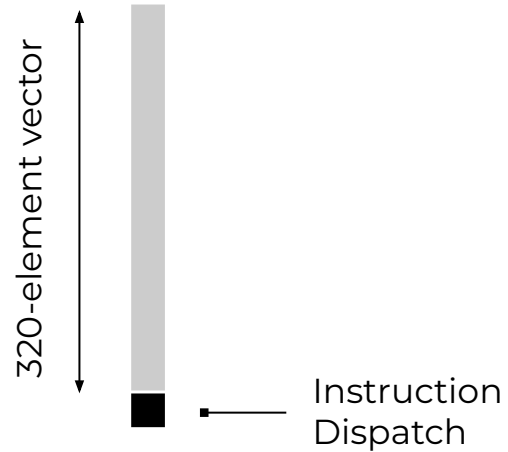
Shift, Transpose, Permuter for improved data movement and data reshapes

Instruction Control

Multiple instruction queues for instruction parallelism

GroqChip™ Building Blocks

SIMD Unit



GroqChip™ Building Blocks

Build different types of specialized SIMD units



MXM

Matrix-Vector /
Matrix-Matrix Multiply



VXM

Vector-Vector
Operations



SXM

Data Reshapes

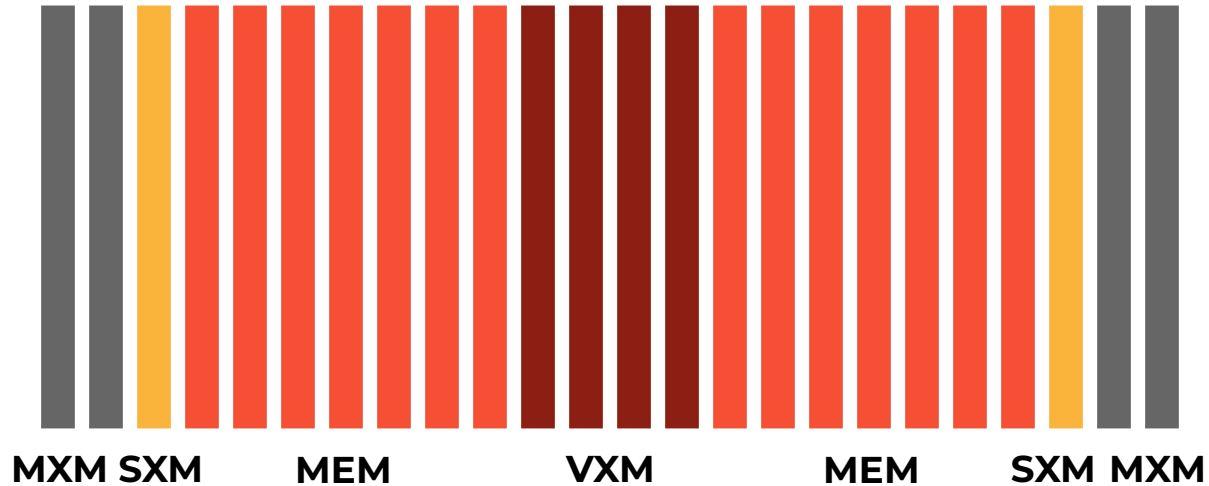


MEM

On-chip SRAM

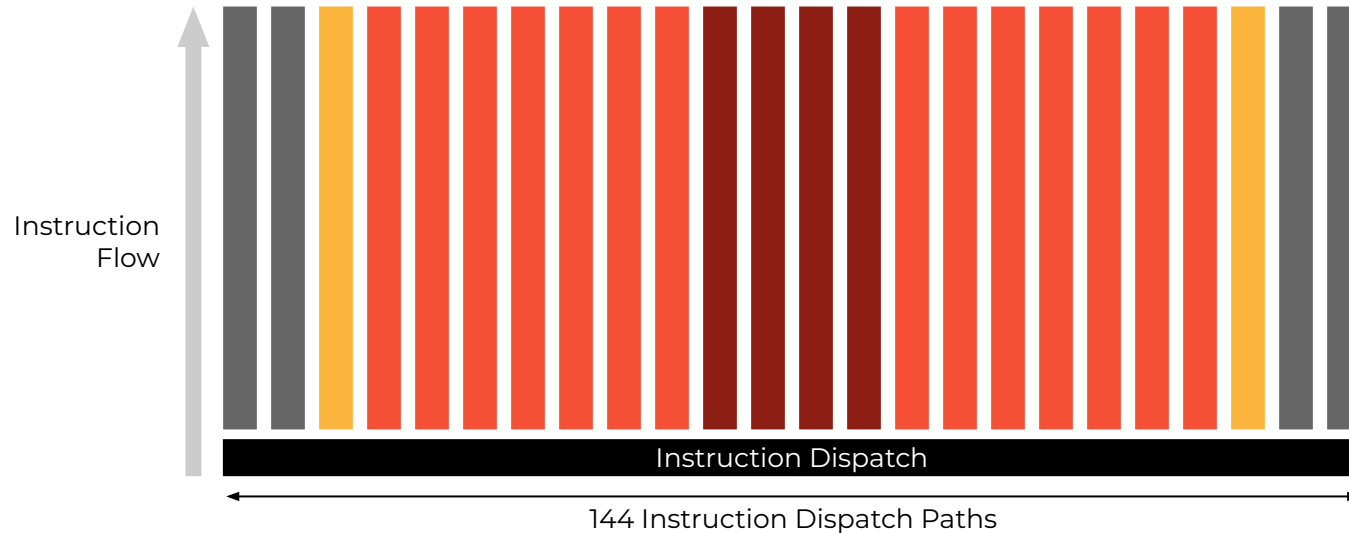
GroqChip™ Building Blocks

Lay out SIMD units across chip area



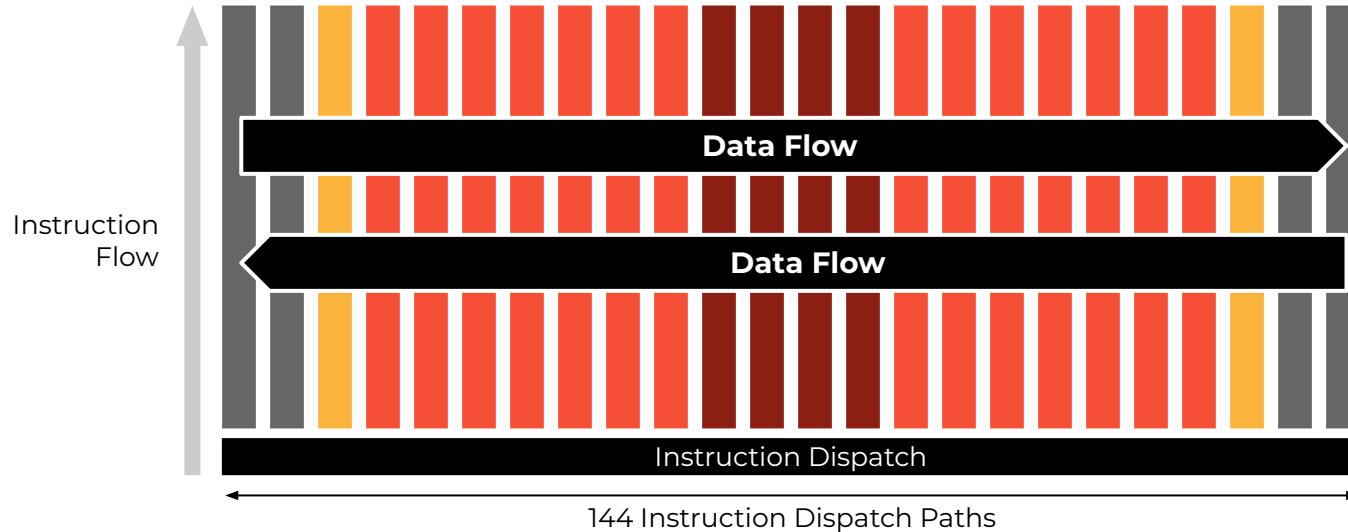
GroqChip™ Building Blocks

Synchronized instruction dispatch across all SIMD units for lockstep execution



GroqChip™ Building Blocks

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



Empowering Groq™ Compiler

Architecture Empowering Software

Software-controlled memory

No dynamic hardware caching

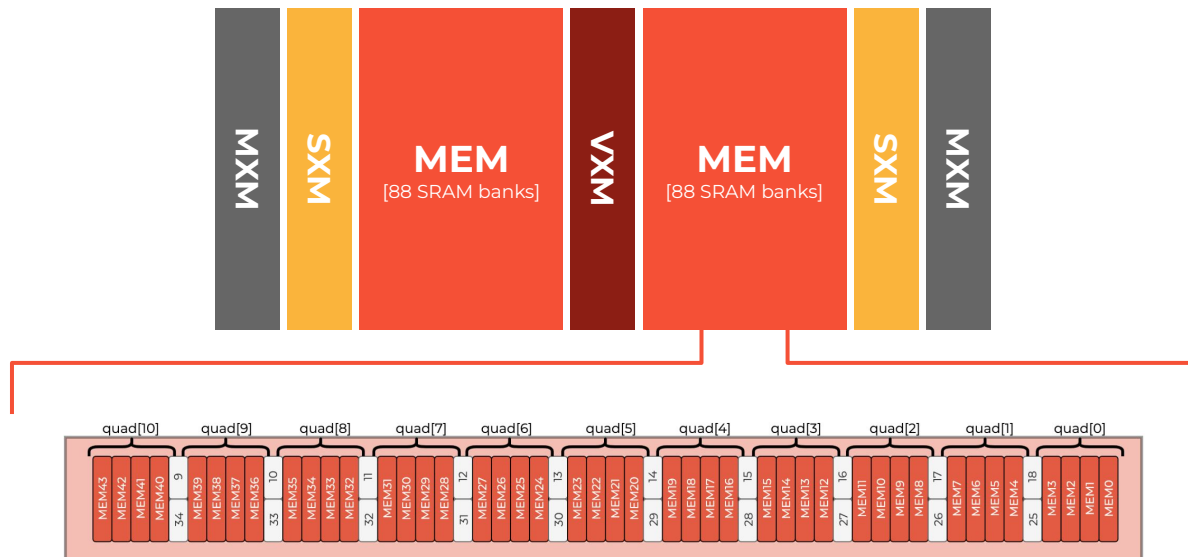
- Compiler aware of all data locations at any given point in time

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

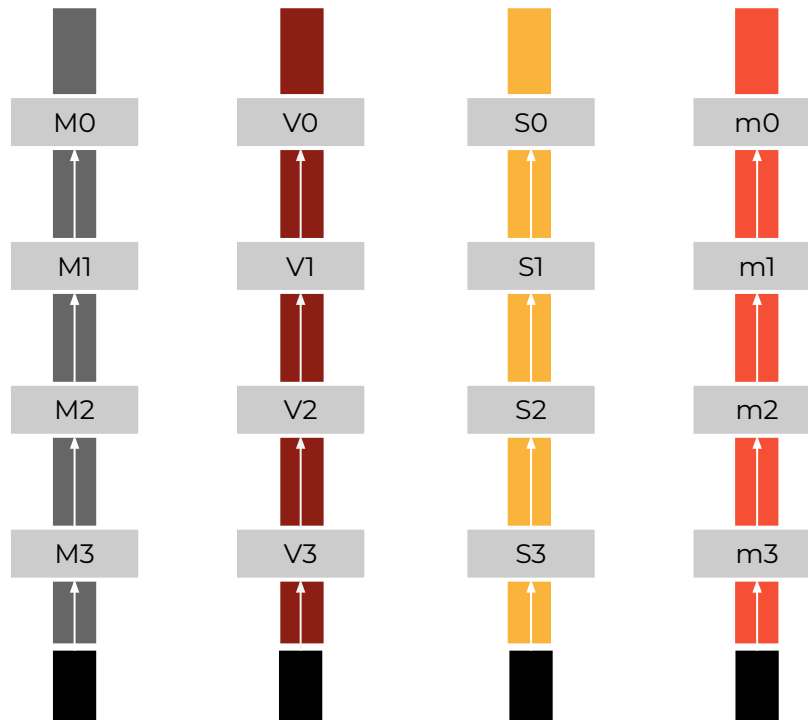
Lockstep execution of Functional Units

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

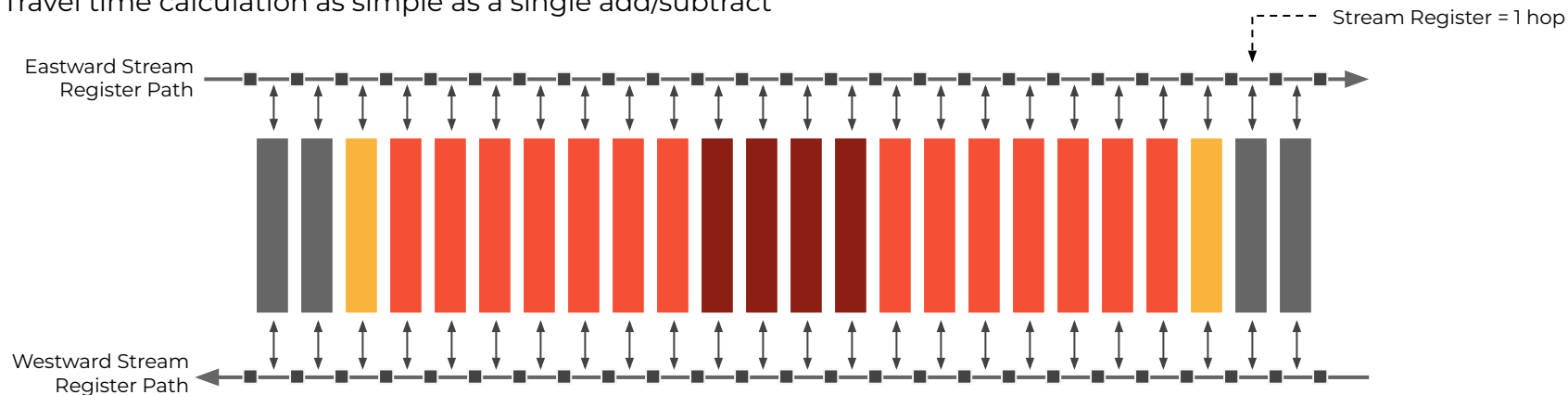
Simple, one-dimensional interconnect for inter-FU communication

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

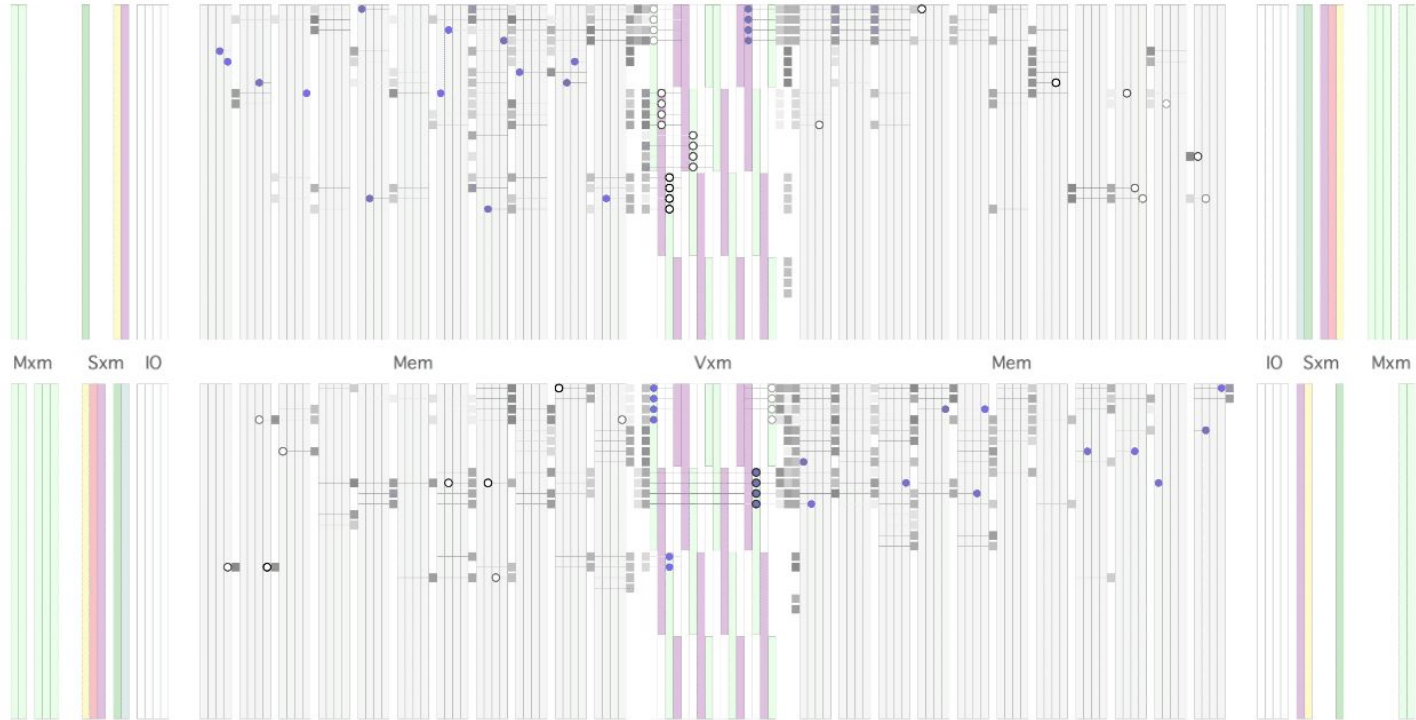
No arbiters / queues = software can easily reason about exact data movement without simulation

Travel time calculation as simple as a single add/subtract



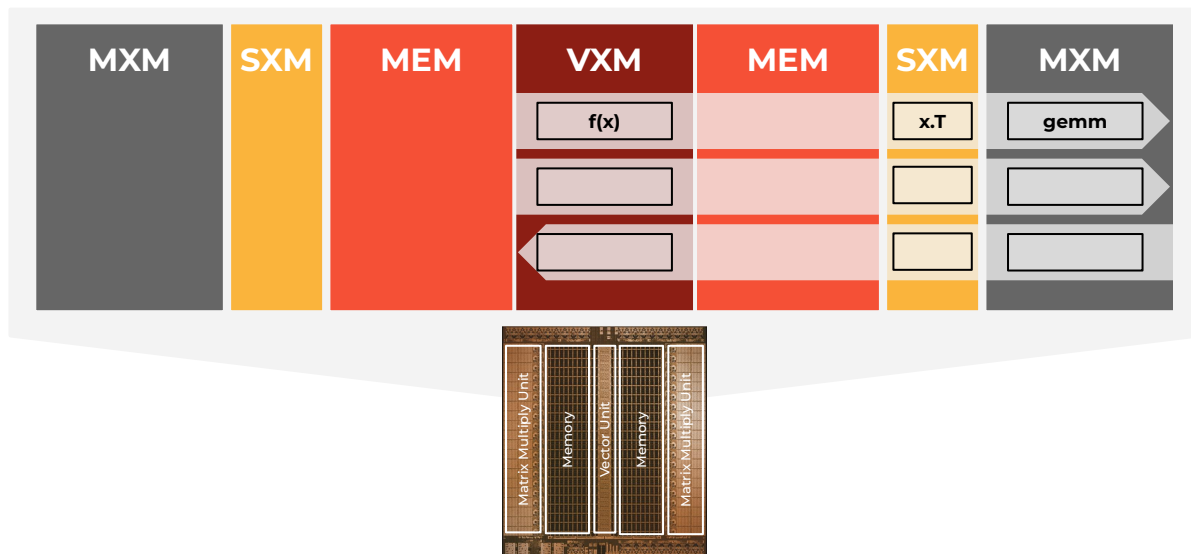
Power of Data Orchestration

Given to Groq Compiler



GroqChip™ Functional Units

Tensor Streaming Dataflow



Spatial pipeline processing

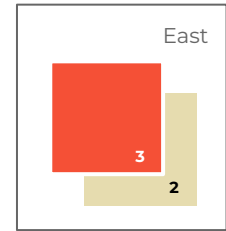
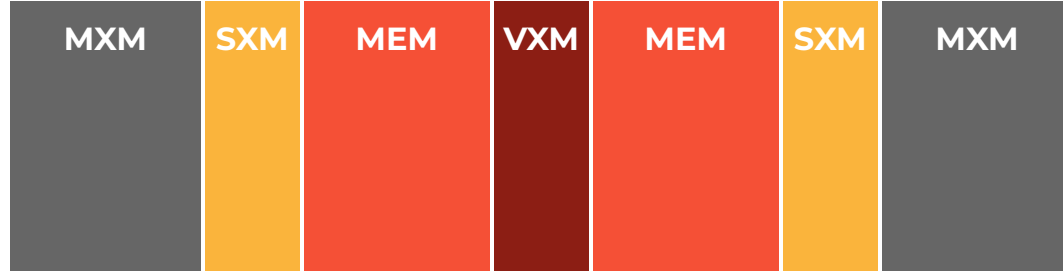
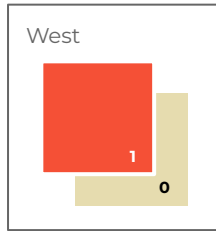
Simple tensor instruction set architecture

Stream programming of massive SIMD, concurrent streams

Large on-chip memory bandwidth

Deterministic, predictable performance scales to multi-chip

MXM: Matrix Multiply Engines



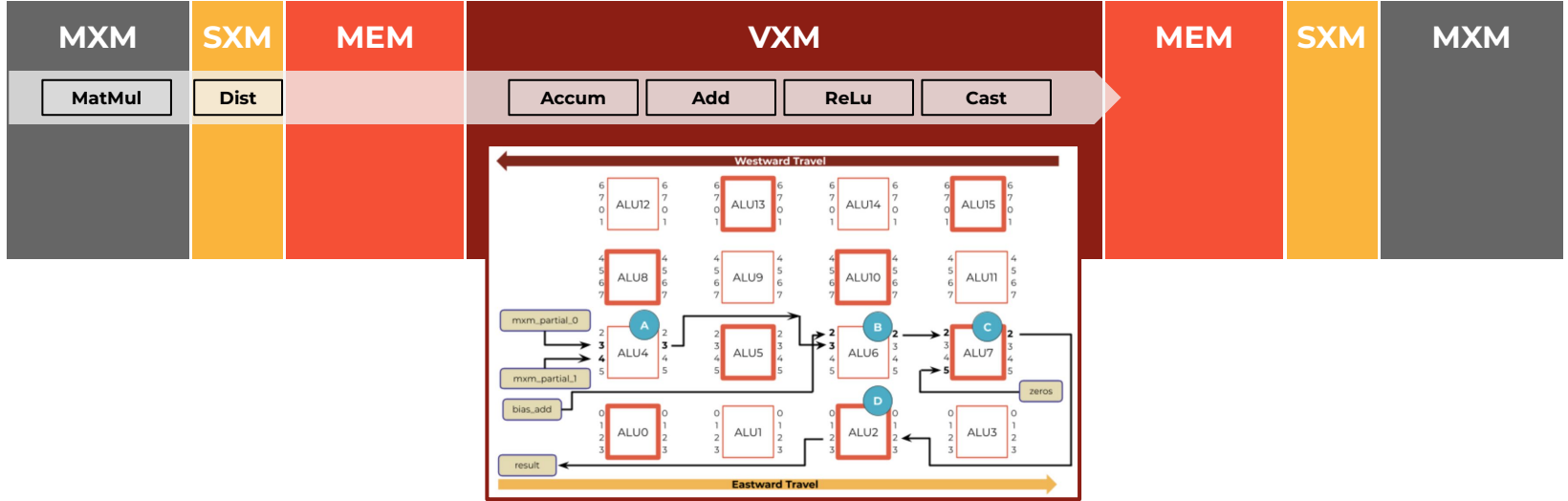
Numeric Mode	Size	Supported Density	Result Tensor
int8	[N, 320] x [320, 320]	Two per MXM	int32
float16	[N, 320] x [160, 320]	One per MXM	float32

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

VXM: Vector Execution Module



Dataflow begins with memory
Read onto Stream Tensor

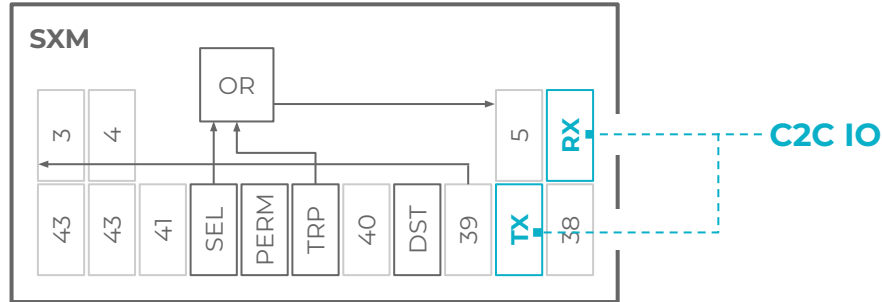
Many concurrent streams
are supported in
programming model

VXM provides a flexible
and programmable fabric
for Compute

Compute occurs on data
locality of passing Stream Tensor

MEM bandwidth supports
high concurrency

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

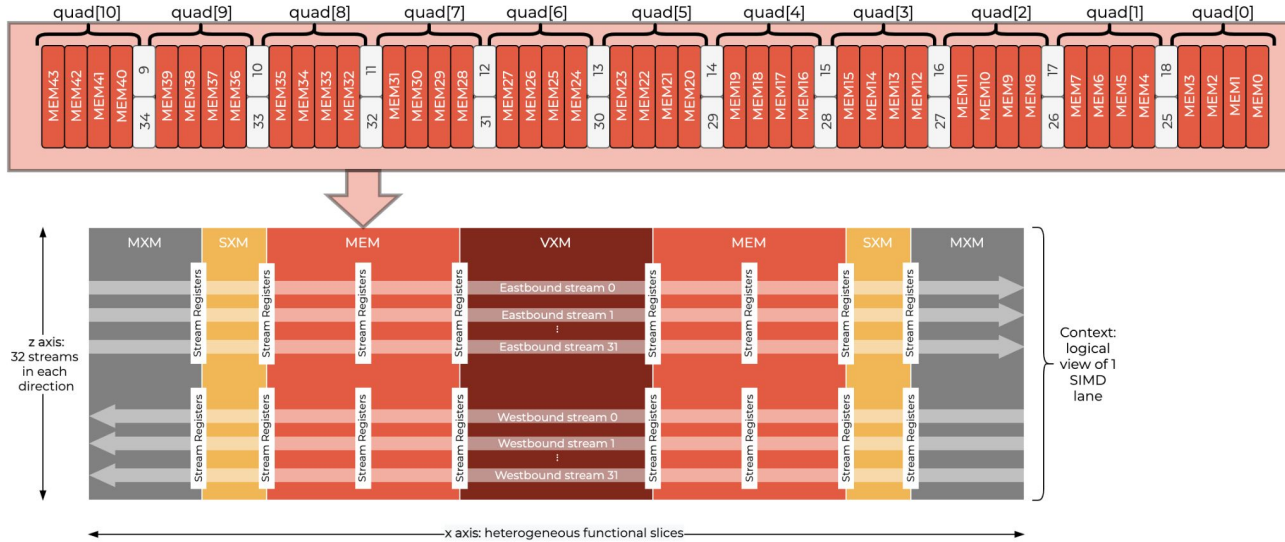
Transposer: 2 per hemisphere perform intra-superlane transpose over 16 vectors for 20 superlanes

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

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

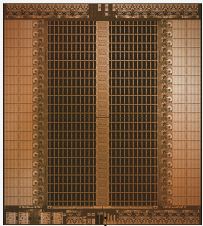
Group MEM slices for multi-dimensional tensors or multi-byte data types

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

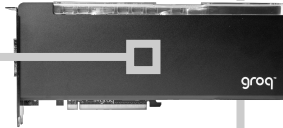
Scaling to 1000s of GroqChip™ Processors

GroqChip™

The purpose-built
Language Processing
Unit™ Inference Engine



GroqCard™



GroqNode™



Dell Servers

GroqRack™



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 TSPs in a network

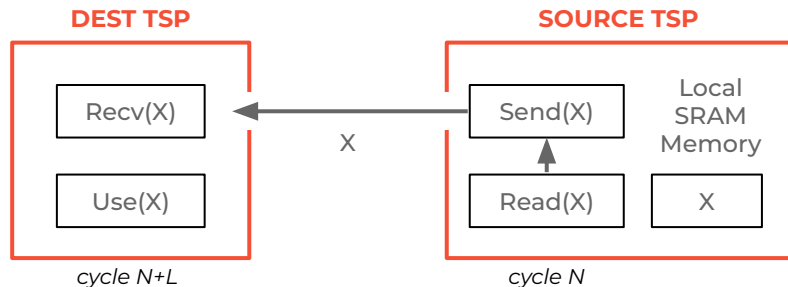
- Clock drift across TSPs is accounted for deterministically

Each TSP acts as both Processor + Router

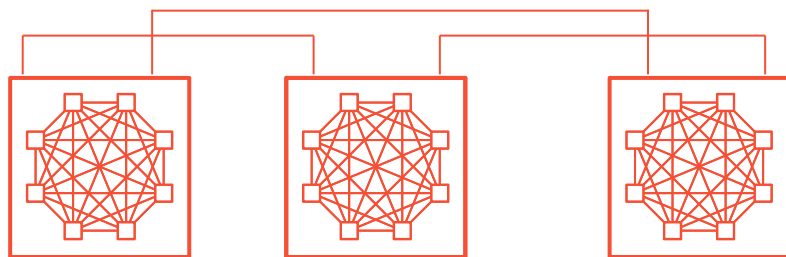
- Compiler schedules network packets as part of programs loaded onto each TSP in the system

No adaptive routing / congestion sensing needed

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



Software-Scheduled Direct Network



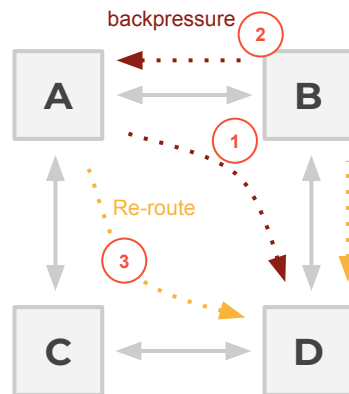
Deterministic Adaptive Routing

Conventional Network

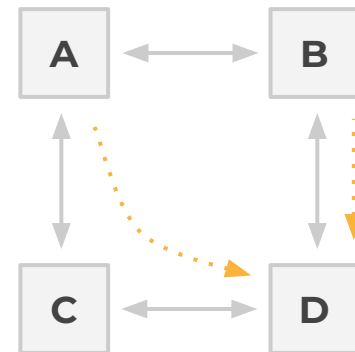
- Commonly done based on network backpressure
- Reactive approach makes the routing decision difficult, increases latency, and increases hardware complexity
- Network latency is **unpredictable**

Software-scheduled Network

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



**Traditional
Non-deterministic
Network**



**Software-scheduled
Network**

Low-diameter Network

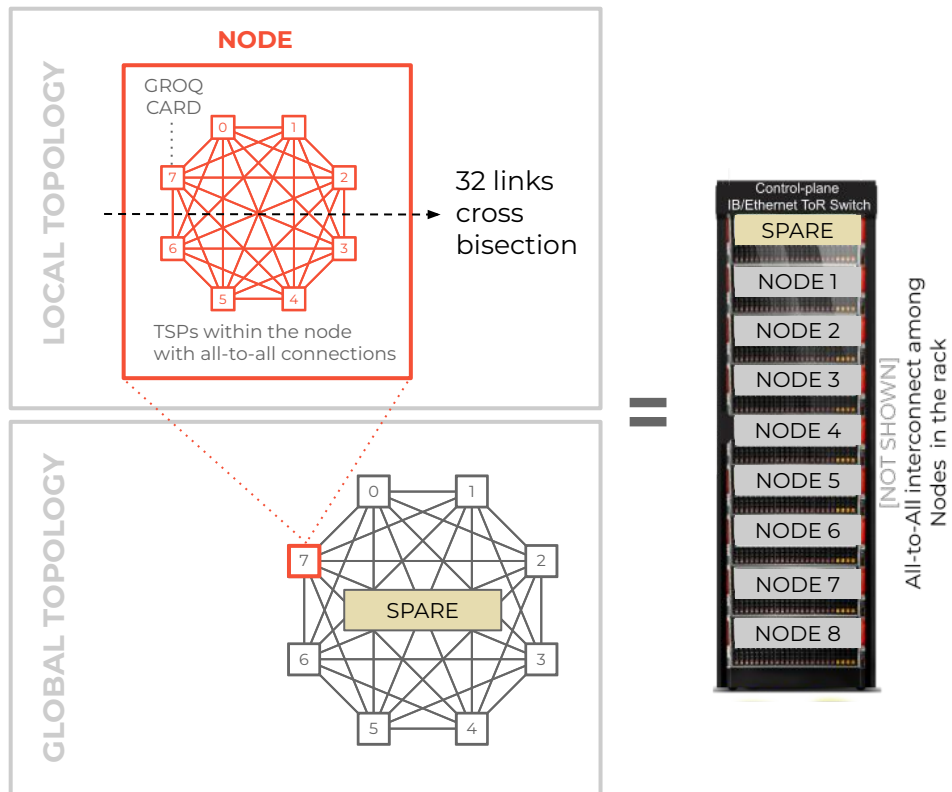
Minimize the number of hops in the network

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

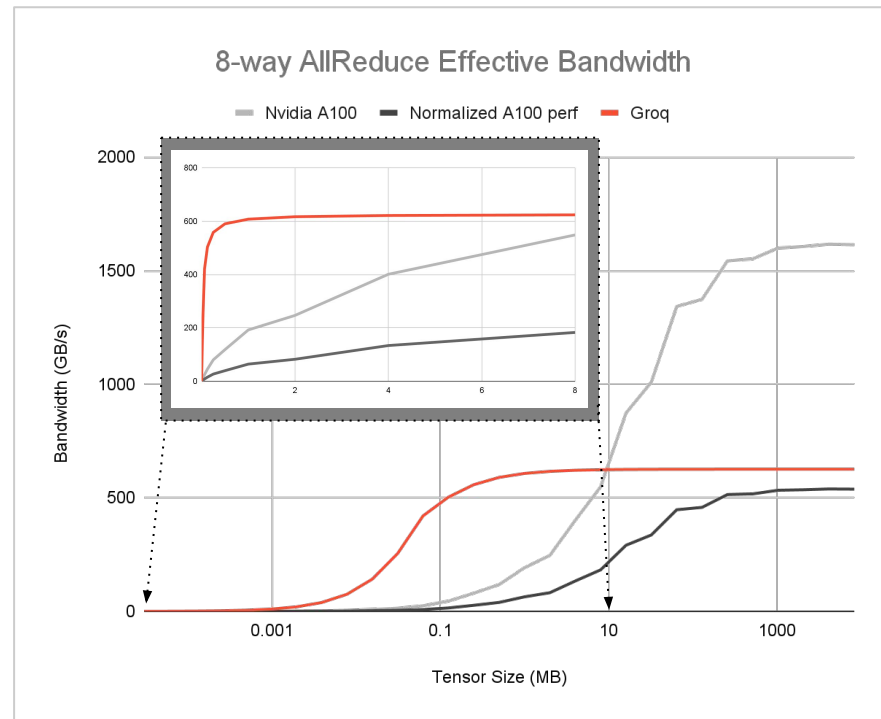
Groq collective communication outperforms state-of-the-art collective AllReduce

Groq RealScale agnostic to common message sizes

- Eliminates the need for message aggregation

When normalized, Groq TSP 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

Ten GroqRack™ Compute Clusters



Recap

Architecture Overview

- Determinism, flat memory hierarchy, 1D interconnect

Key Functional Units

- MXM, VXM, SXM, MEM

Scaling to 1000s of GroqChips

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



groq™

Thank You!

abitar@groq.com

Intro to MLAgility™ & GroqFlow™

Sanjif Shanmugavelu
Software Engineer

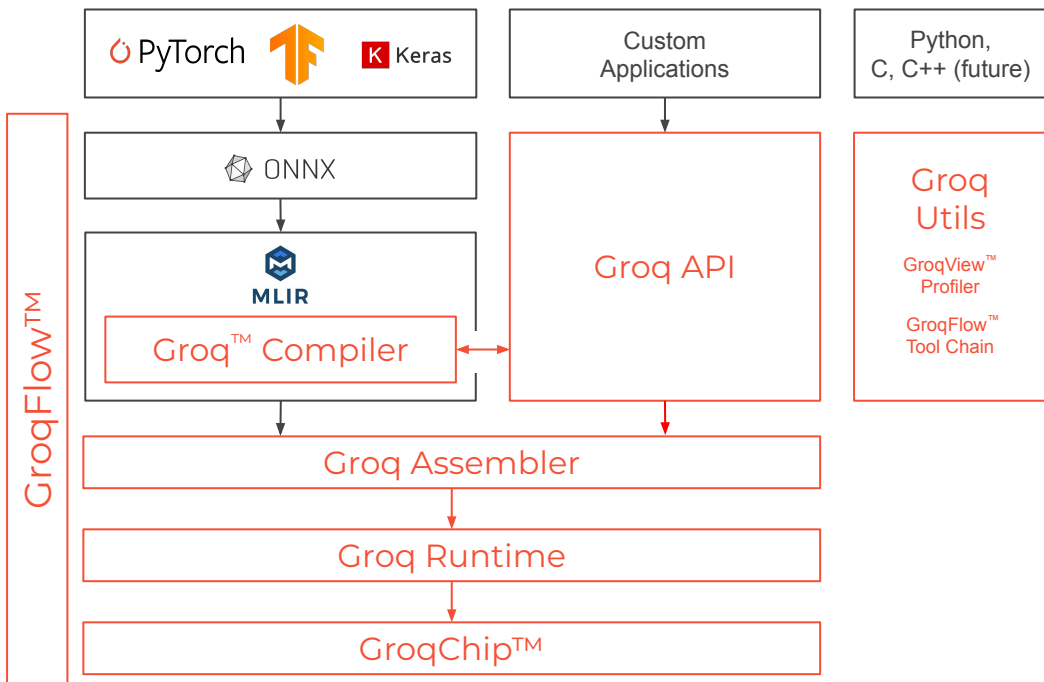
Intro to MLAGility™ & GroqFlow™

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

Fine Grained Control

Groq API provides finer grained control of GroqChip in order to support custom applications



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™**.

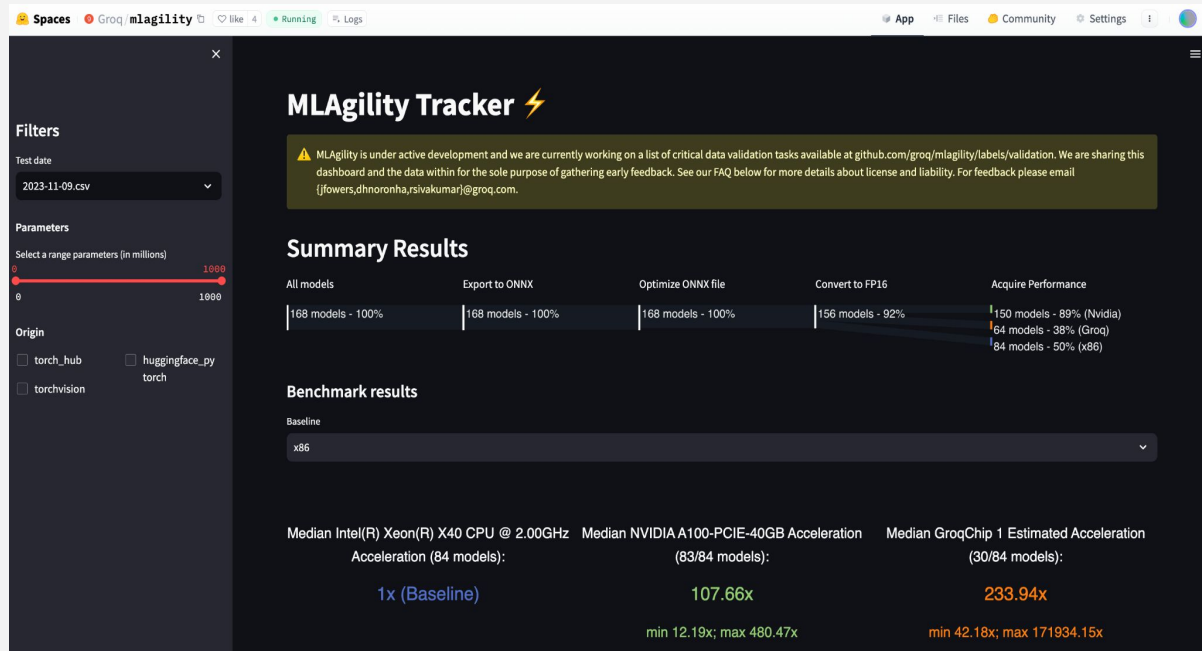


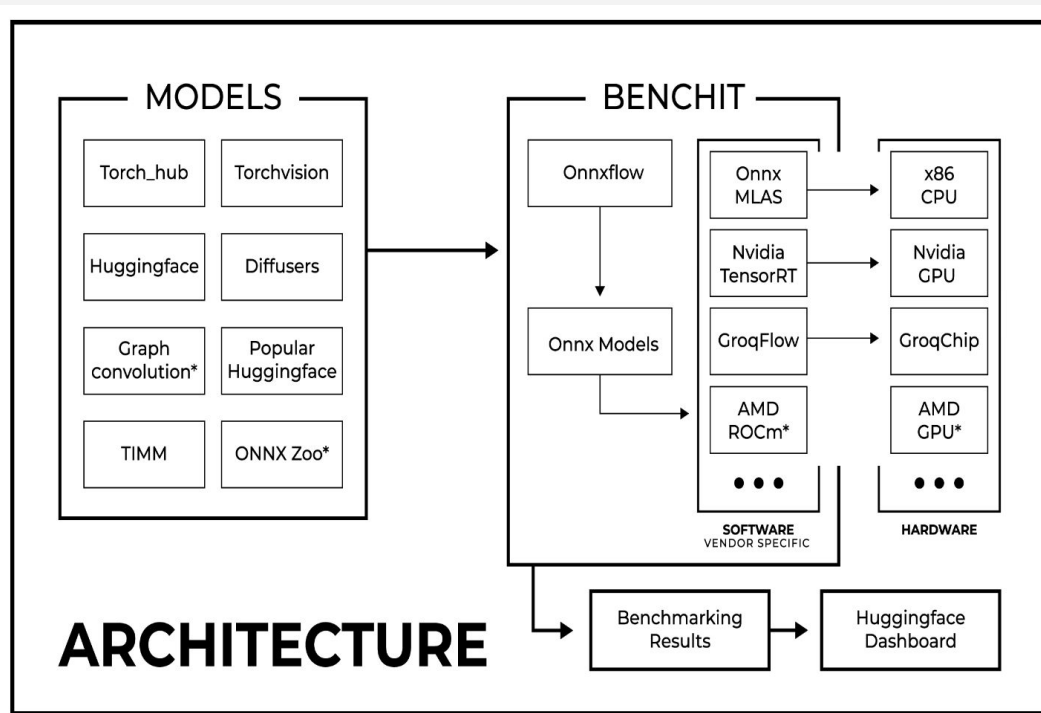
Figure 1: [Public Groq HuggingFace space](https://huggingface.co/spaces/groq/mlagility)

MLAGility Architecture

MLAGility Setup

The diagram illustrates the MLAGility repository structure.

Simply put, the MLAGility models are leveraged by our benchmarking tool, benchit, to produce benchmarking outcomes showcased on our Hugging Face Spaces page.



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



Introducing GroqFlow™

Step 1: Get your model

```
0 import transformers
1 import torch
2 from groqflow import groqit
3
4 model = transformers.GPT2Model(transformers.GPT2Config())
5
6 inputs = {
7     "input_ids": torch.ones(1, 1_024, dtype=torch.long),
8     "attention_mask": torch.ones(1, 1_024, dtype=torch.float),
9 }
10
11 gmodel = groqit(model,inputs)
12
13 output = gmodel(**inputs)
14
15
```

Introducing GroqFlow™

Step 2: Get some inputs

```
0 import transformers
1 import torch
2 from groqflow import groqit
3
4 model = transformers.GPT2Model(transformers.GPT2Config())
5
6 inputs = {
7     "input_ids": torch.ones(1, 1_024, dtype=torch.long),
8     "attention_mask": torch.ones(1, 1_024, dtype=torch.float),
9 }
10
11 gmodel = groqit(model,inputs)
12
13 output = gmodel(**inputs)
14
15
```

Introducing GroqFlow™

Step 3: Just Groq it!


```
0 import transformers
1 import torch
2 from groqflow import groqit
3
4 model = transformers.GPT2Model(transformers.GPT2Config())
5
6 inputs = {
7     "input_ids": torch.ones(1, 1_024, dtype=torch.long),
8     "attention_mask": torch.ones(1, 1_024, dtype=torch.float),
9 }
10
11 gmodel = groqit(model,inputs)
12
13 output = gmodel(**inputs)
14
15
```

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

Introducing GroqFlow™

Inference is easy!

```
0 import transformers
1 import torch
2 from groqflow import groqit
3
4 model = transformers.GPT2Model(transformers.GPT2Config())
5
6 inputs = {
7     "input_ids": torch.ones(1, 1_024, dtype=torch.long),
8     "attention_mask": torch.ones(1, 1_024, dtype=torch.float),
9 }
10
11 gmodel = groqit(model,inputs)
12
13 output = gmodel(**inputs)
14
15
```



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

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



COMPILES models
into Groq programs

gmodel = groqit(model,inputs)

EXECUTES programs
on GroqChip™

gmodel(**inputs)

BENCHMARKS programs
on GroqChip

latency = gmodel.benchmark()

Quick User Guide

GroqIt Args



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

Examples:

```
groqit(my_pytorch_model,inputs)
```

Main GroqIt Args

model

- Model to be mapped to a GroqModel
- PyTorch model instance

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 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)

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 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

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 (coming soon)

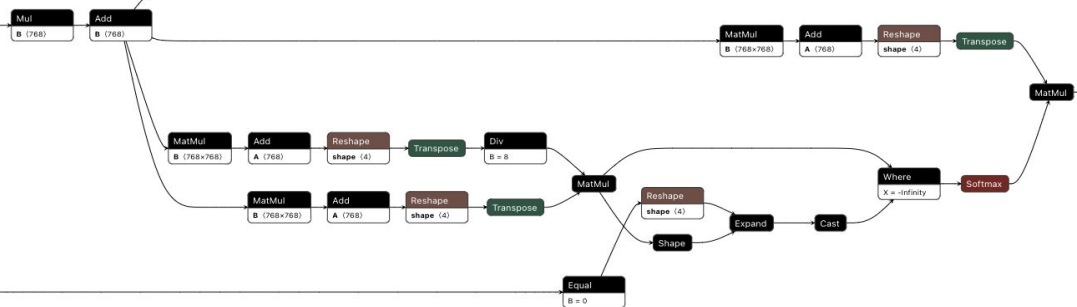
- Returns the average latency of 100 runs in ms
- Latency includes PCIe times + on-chip compute

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 (coming soon)

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

netron

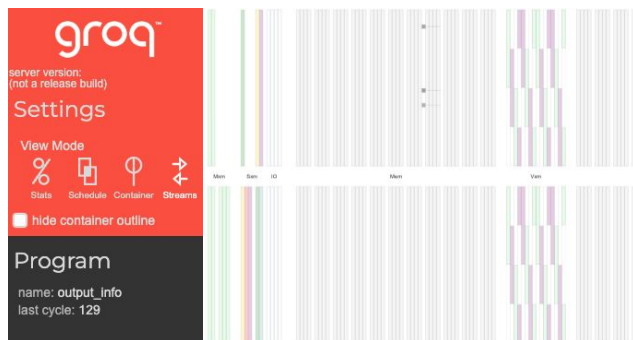
- Opens the ONNX model generated by GroqIt

Quick User Guide

Groq Model Functions

```
gmodel = groqit(model, inputs, groq_view=True)
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 (coming soon)

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

netron

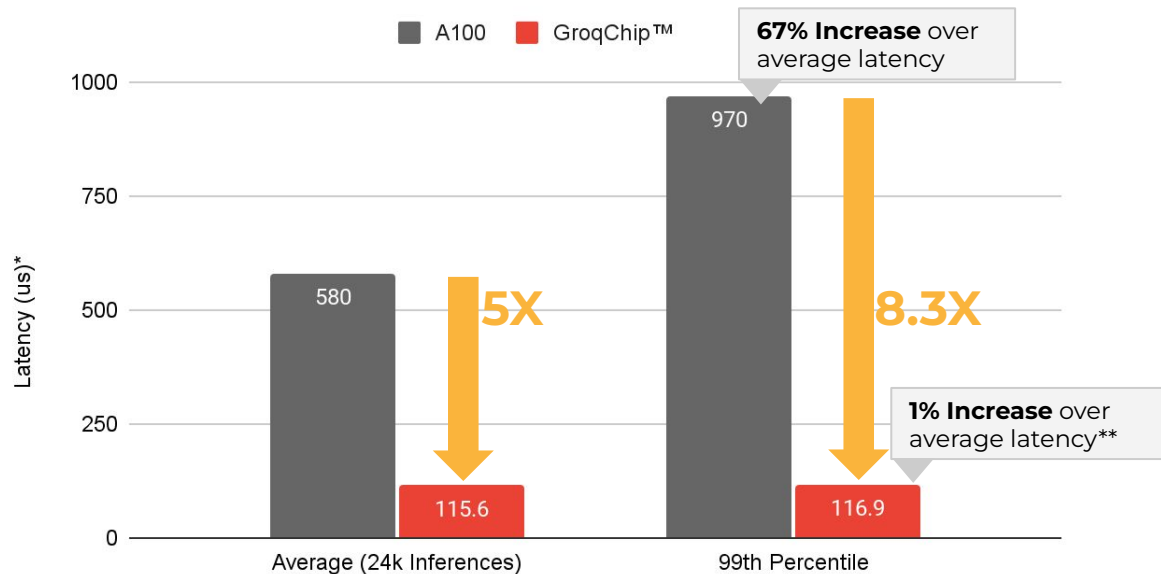
- Opens the ONNX model generated by GroqIt

groqview

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

Low Latency Every Time

BERT-base Latency



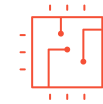
Groq Advantages



Determinism



Low Latency



Large On-chip
Memory

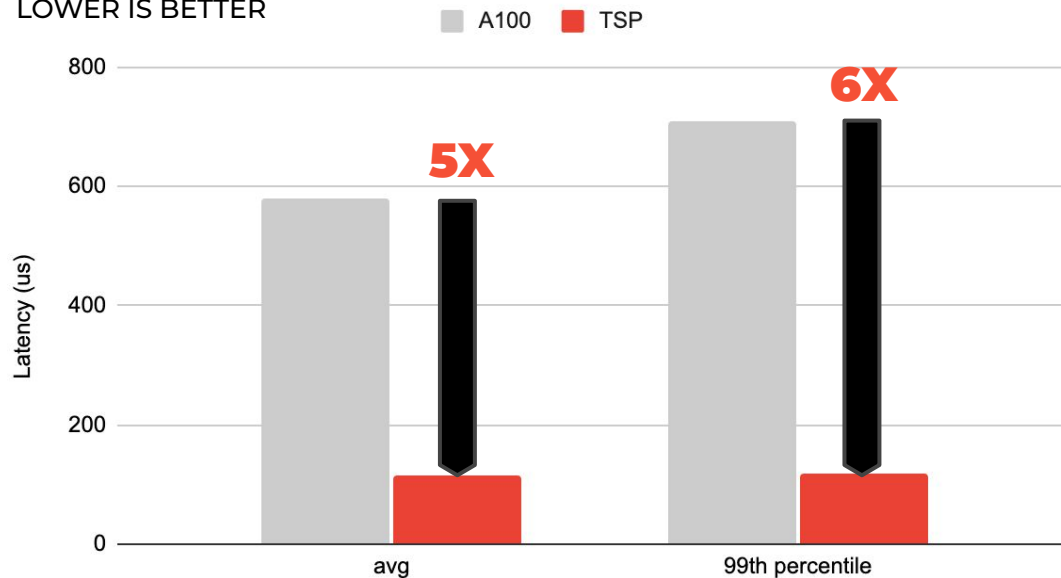
GroqCard delivers up to **8.3X** better performance on the slowest inference

BERT

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

BERT-base latency

LOWER IS BETTER



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, GroqChip processors, 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

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.
```

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

```
# 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)
```

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

MLOpacity Report Generation

Collect and present results

You can aggregate all of the benchmarking results from your mlogility 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
```

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

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

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