Accelerating GenAI Workloads by Enabling RISC-V Microkernel Support in IREE

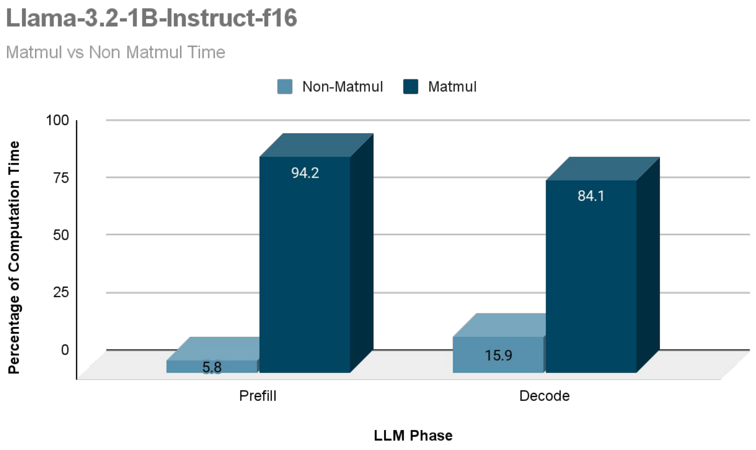
Adeel Ahmad, Ahmad Tameem, Nouman Amir, Bilal Zafar, **Saad Bin Nasir**
10xEngineers

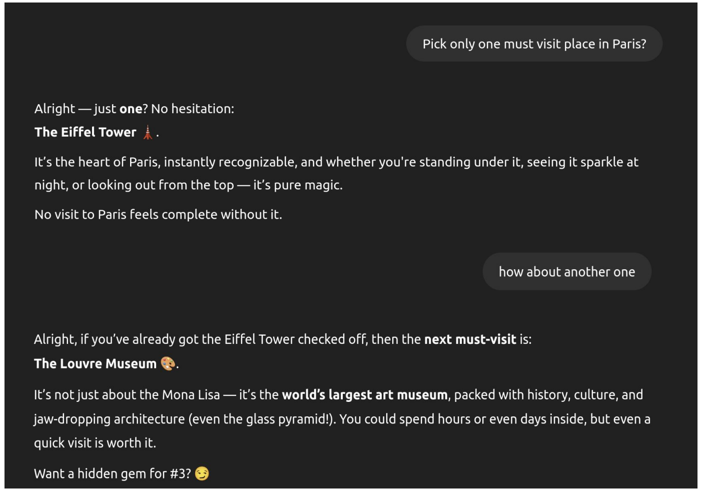
Outline

- Generative AI workloads
- IREE compilation with custom microkernels (ukernels)
- Custom RISC-V matrix multiplication ukernels - implementation
- Kernel- and model-level results
- Summary

Generative AI Workloads

- Generative AI workloads are dominated by transformer-based auto-regressive large language models (LLMs)
- text/image/code generation, chatbots, content writing, video generation and other common uses-cases heavily employ LLMs
- Matrix-matrix and matrix-vector multiplications dominate these workloads





Pick only one must visit place in Paris?

Alright — just **one**? No hesitation:

The Eiffel Tower 🗼.

It's the heart of Paris, instantly recognizable, and whether you're standing under it, seeing it sparkle at night, or looking out from the top — it's pure magic.

No visit to Paris feels complete without it.

how about another one

Alright, if you've already got the Eiffel Tower checked off, then the **next must-visit** is:

The Louvre Museum 🎨.

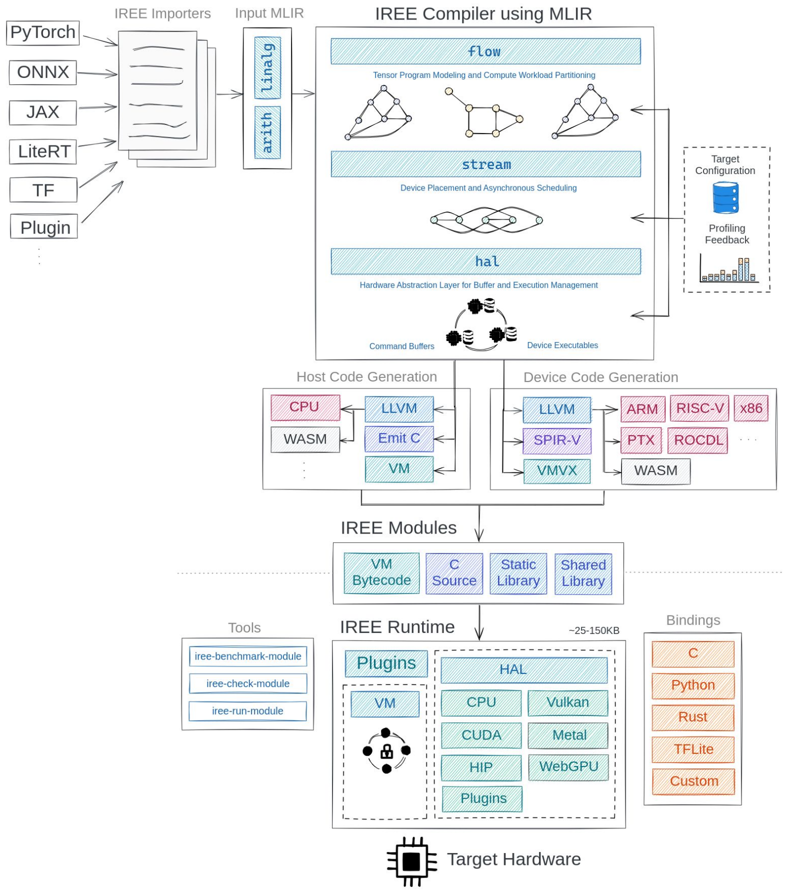
It's not just about the Mona Lisa — it's the **world's largest art museum**, packed with history, culture, and jaw-dropping architecture (even the glass pyramid!). You could spend hours or even days inside, but even a quick visit is worth it.

Want a hidden gem for #3? 😏

Source: Chatgpt

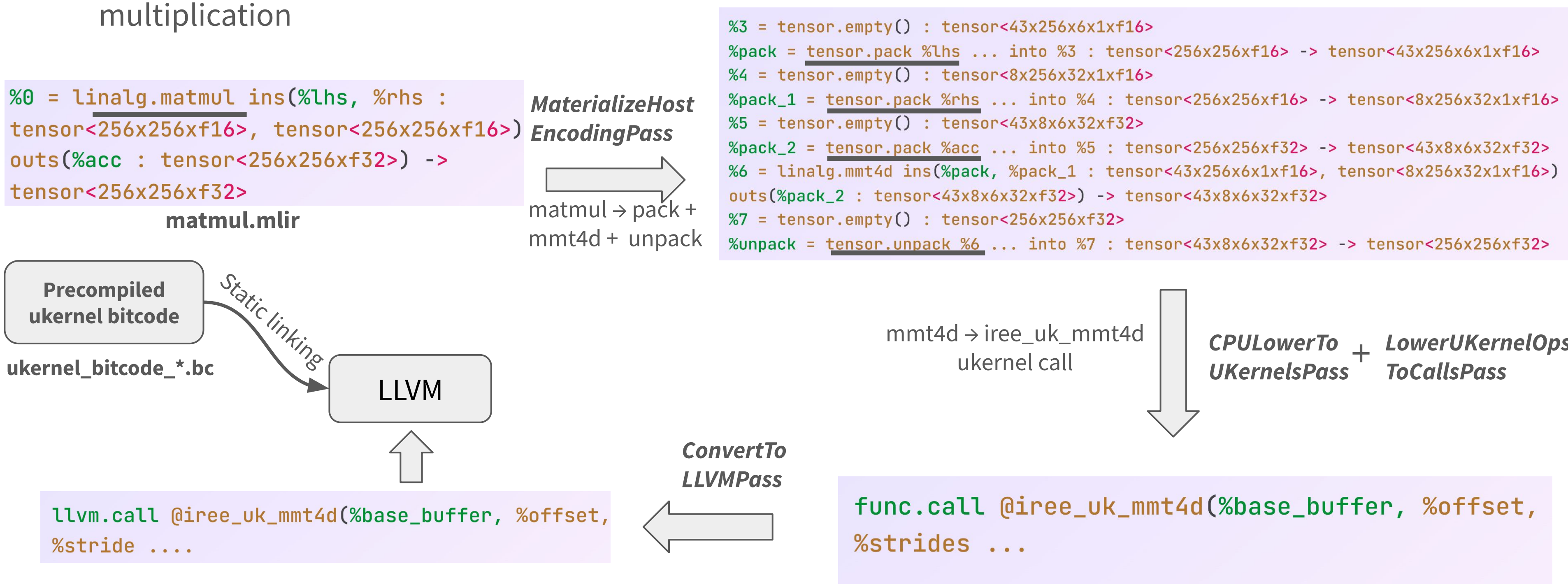
IREE Compilation with Custom Kernels

- Open-source direct code generation MLIR-based compiler and runtime
- Host/device programming model with multiple target architectures through a hardware abstraction layer (HAL) → stack is mostly architecture agnostic → step towards heterogeneous compilation
- Host does scheduling, vm-bytecode for runtime → portability
- Device-side codegen; Upstream IREE has RVV codegen through LLVM
- Microkernels
  - Intended to prevent the dichotomy between compiler and kernels
  - perform arithmetic but no memory allocation
  - standalone development and unit testing in C leads to quicker development



Matrix Multiplication ukernel (mmt4d) Compilation in IREE

- For **x86_64** and **ARM64** architectures, IREE leverages linalg dialect's **mmt4d** op for matrix multiplication



```
%0 = linalg.matmul ins(%lhs, %rhs :
tensor<256x256xf16>, tensor<256x256xf16>)
outs(%acc : tensor<256x256xf32>) ->
tensor<256x256xf32>
```

matmul.mlir

MaterializeHost EncodingPass

matmul → pack + mmt4d + unpack

```
%3 = tensor.empty() : tensor<43x256x6x1xf16>
%pack = tensor.pack %lhs ... into %3 : tensor<256x256xf16> -> tensor<43x256x6x1xf16>
%4 = tensor.empty() : tensor<8x256x32x1xf16>
%pack_1 = tensor.pack %rhs ... into %4 : tensor<256x256xf16> -> tensor<8x256x32x1xf16>
%5 = tensor.empty() : tensor<43x8x6x32xf32>
%pack_2 = tensor.pack %acc ... into %5 : tensor<256x256xf32> -> tensor<43x8x6x32xf32>
%6 = linalg.mmt4d ins(%pack, %pack_1 : tensor<43x256x6x1xf16>, tensor<8x256x32x1xf16>)
outs(%pack_2 : tensor<43x8x6x32xf32>) -> tensor<43x8x6x32xf32>
%7 = tensor.empty() : tensor<256x256xf32>
%unpack = tensor.unpack %6 ... into %7 : tensor<43x8x6x32xf32> -> tensor<256x256xf32>
```

mmt4d → iree_uk_mmt4d ukernel call

CPULowerTo UKernelsPass + *LowerUKernelOps ToCallsPass*

```
func.call @iree_uk_mmt4d(%base_buffer, %offset,
%strides ...
```

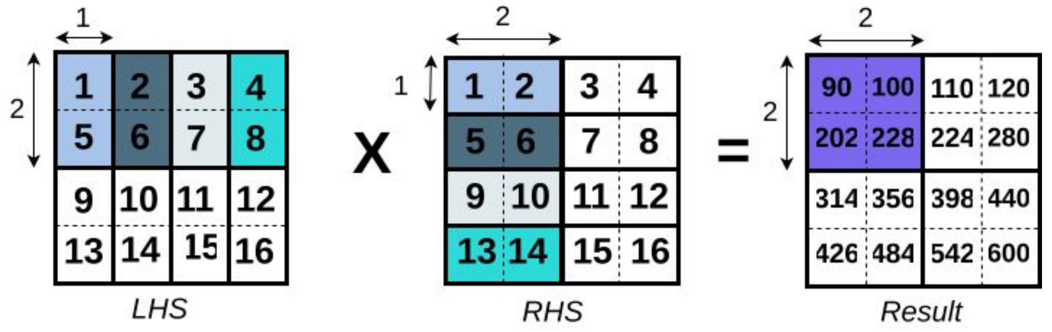
ConvertTo LLVMPass

```
llvm.call @iree_uk_mmt4d(%base_buffer, %offset,
%stride ....
```

- **mmt4d** op is meticulously optimized to exploit **hardware-specific vector instructions** and **cache hierarchies**

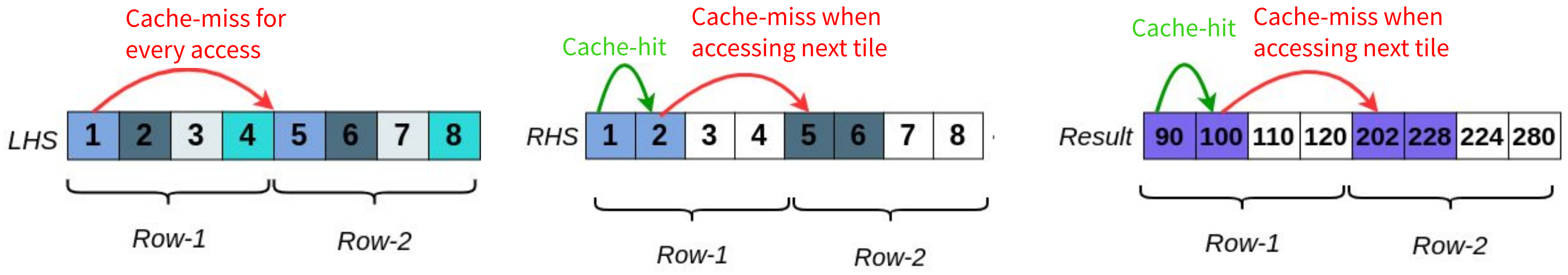
Only relevant parts of MLIR and pass pipeline are shown

Why mmt4d is better?



Tiled Matmul with Tile Size = {M,N,K} = {2,2,1}

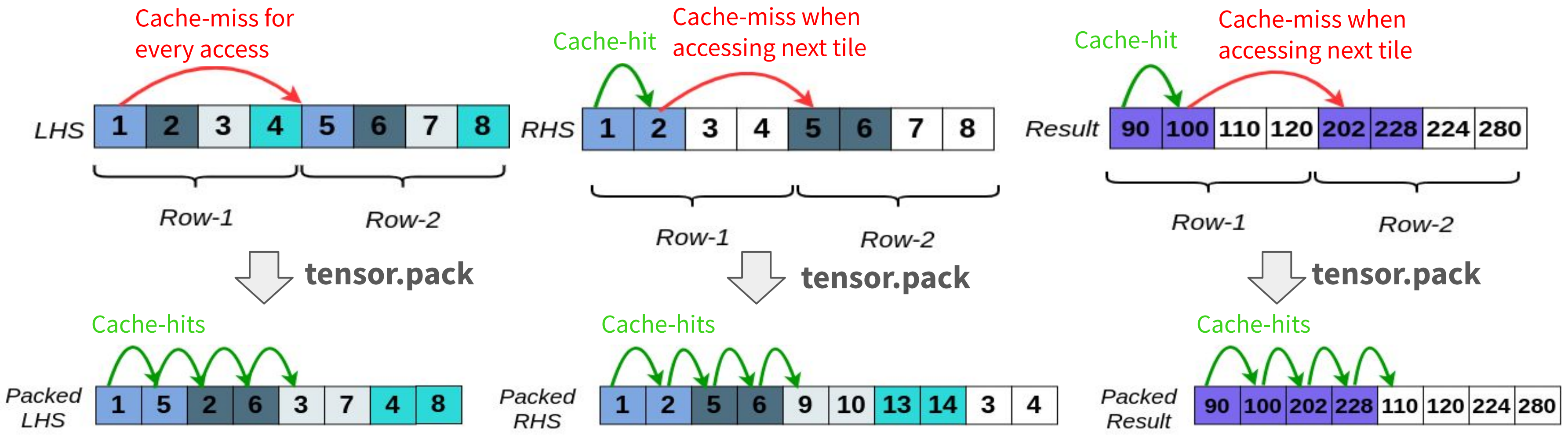
- MLIR bufferizes all tensors in **Row Major** order as shown below.



- In case of plain **MLIR linalg.matmul op**, memory accesses are **non-contiguous**, which leads to a **low cache hit rate** when processing large matrices. This results in reduced performance due to **inefficient utilization of the memory hierarchy**.

Why mmt4d is better?

- To reduce number of accesses to non-contiguous memory location, **we need to reorder data**. IREE uses MLIR **tensor.pack** ops for this.



- After packing, **elements within each tile are stored contiguously** in memory, and the **tiles themselves are laid out contiguously** with respect to one another
- MLIR **linalg.mmt4d** op **operates on these packed matrices**. This results in **efficient cache utilization**, and performance acceleration

RISC-V mmt4d Kernel - Implementation

- RISC-V ukernels were implemented for **F16xF16 —>F32 case**
- **Separate** ukernels for **prefill** and **decode** phases

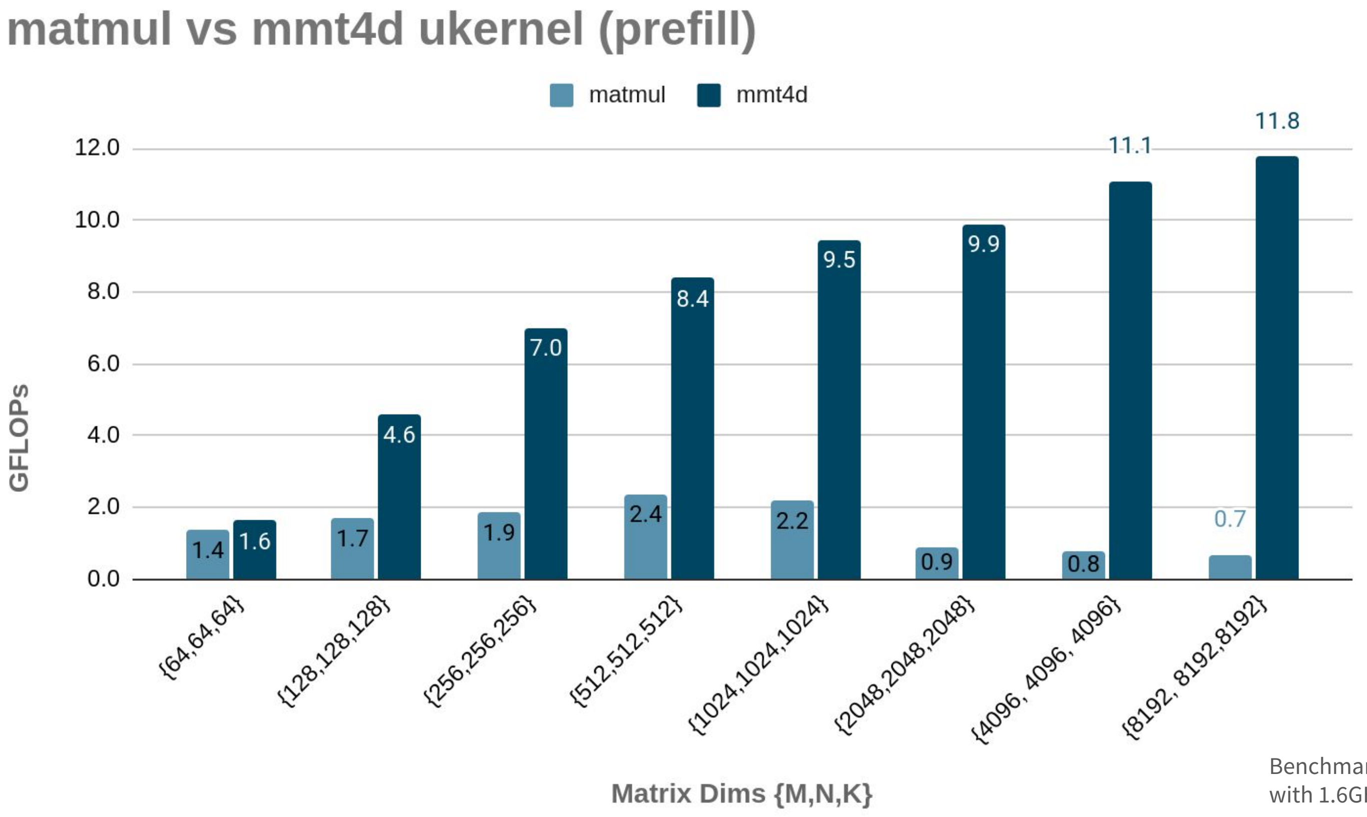
VLEN	**128**	**256**	**512**	**1024**
Tile Size (Prefill)	{6, 16, 1}	{6, 32, 1}	{6, 64, 1}	{6, 128, 1}
Tile Size (Decode)	{1, 32, 1}	{1, 64, 1}	{1, 128, 1}	{1, 256, 1}

VLEN Aware Tiling
Tile Size:

- Prefill→ **{M,N,K}={6,VLEN/8,1}**
- Decode → **{M,N,K}={1,VLEN/4,1}**
- These sizes result in **optimal register utilization** and **minimizes register spills/reloads**

- **F16** inputs were widened to **F32** using **vfcwt** instructions
- **vfmacc.v.f (ELEN=32)** instructions were used to perform **multiply-accumulation**
- uKernel testing via **fuzzing with random matrices** of **varying dimensions**; results validated against upstream IREE’s **default matmul** path

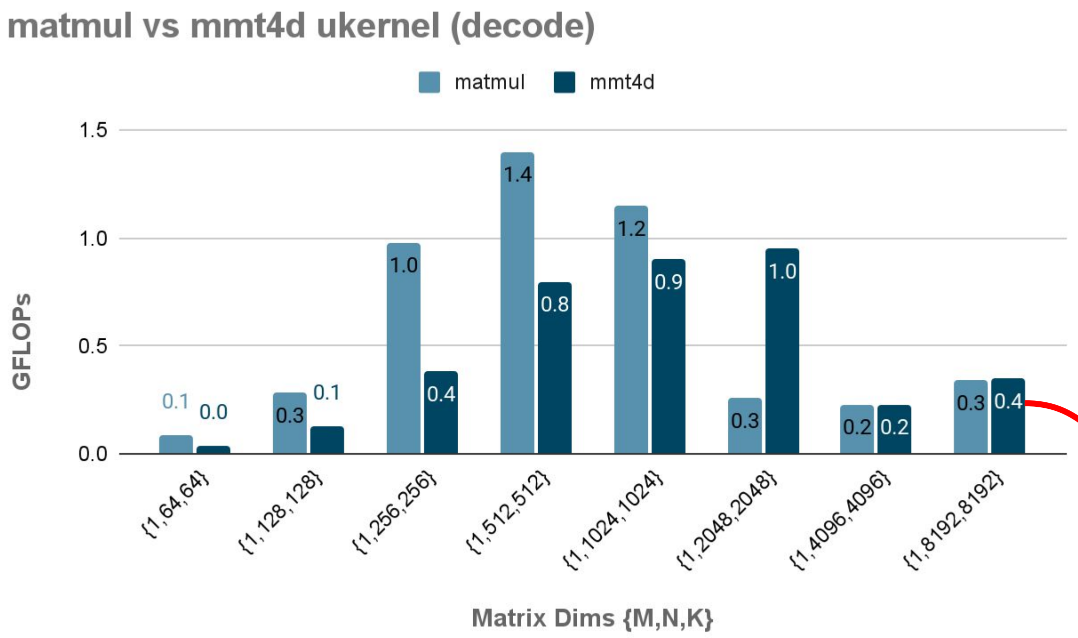
RISC-V mmt4d Kernel - Benchmarking



Mmt4d kernel **performs consistently better** over a range of matrix sizes

Benchmarking was performed on MILK-V Jupiter board with 1.6GHz x 8 cores RVV cores, VLEN=256, RVA22

RISC-V mmt4d Kernel - Benchmarking

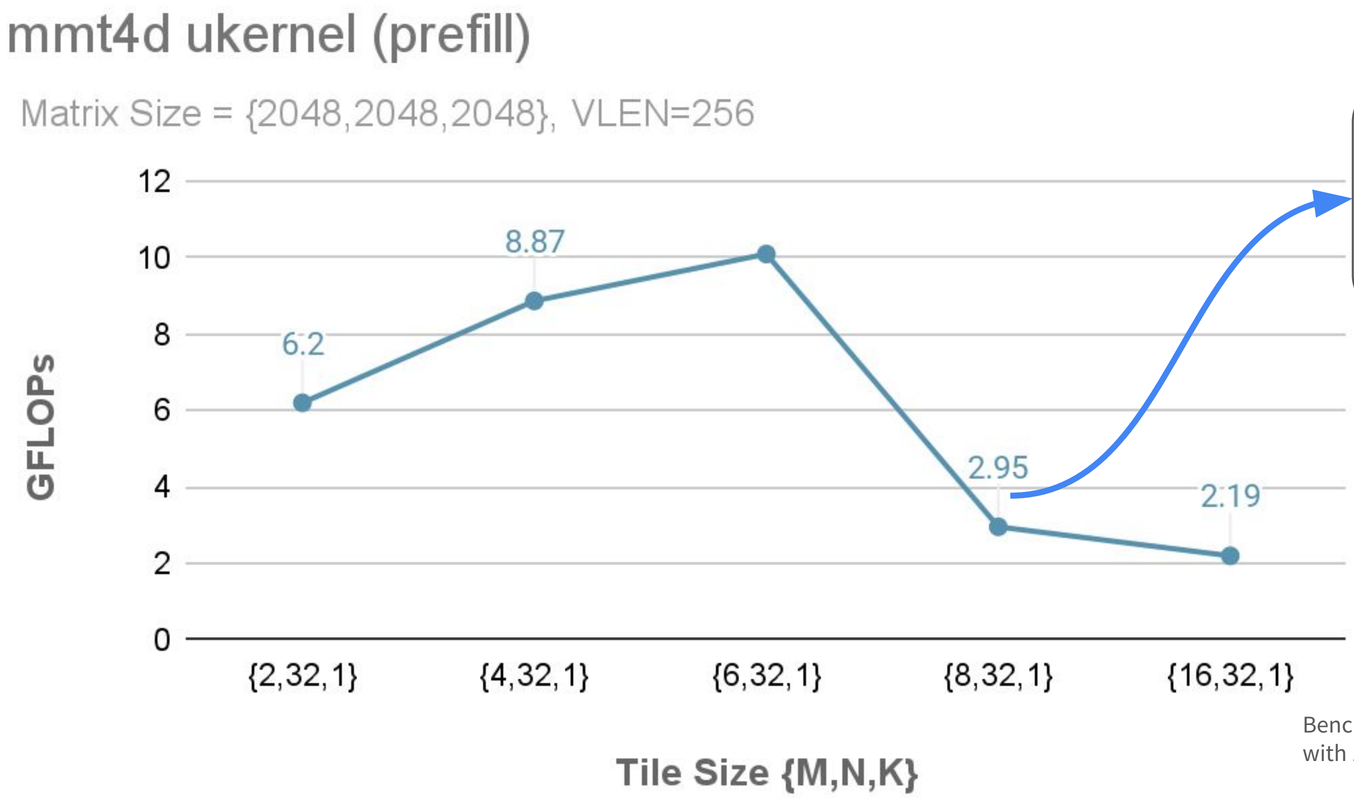


- At **kernel level**, mmt4d for decode seems to cause performance degradation
- **Smaller matrices → less impact of cache misses → pack op cost dominates in mmt4d**
- Overhead of boilerplate code more dominant

Here, **>60% of execution time is consumed by pack op,** in real models **pack cost can be evaded leading to performance boost!**

Benchmarking was performed on MILK-V Jupiter board with 1.6GHz x 8 cores RVV cores, VLEN=256, RVA22

RISC-V mmt4d Kernel - Benchmarking



For **M>6, register pressure is high** and register spills/reload results in performance degradation

Benchmarking was performed on MILK-V Jupiter board with 1.6GHz x 8 cores RVV cores, VLEN=256, RVA22

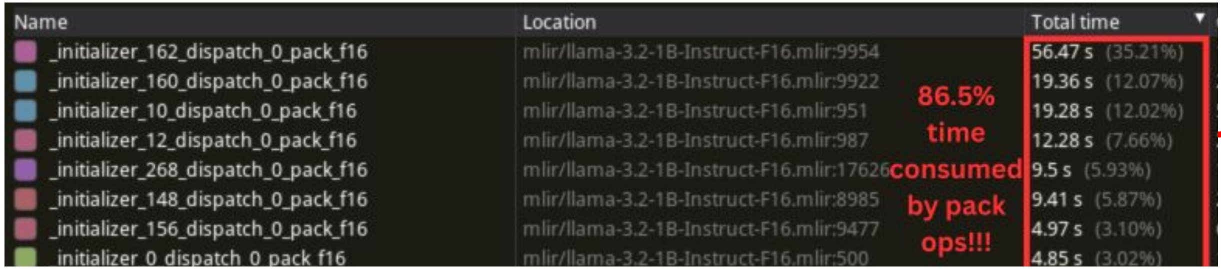
Llama-3.2-1B-Instruct-f16 Benchmarking

Test	LLaMa -1B	IREE -10x
Arc_c	59.4%	59.4%
GPQA	27.2%	27.2%

Correctness: Zero shot accuracy verified for general Q/A tests through LM Eval Harness

	threads	llama.cpp	IREE	IREE-10x
Prefill (toks/s)	1	0.04	0.14	**0.18**
	8	0.11	0.91	**1.89** ~2x
Decode (toks/s)	1	0.03	0.02	**0.99** ~50x
	8	0.07	0.12	**2.12** ~17x

86.5% computation time is consumed by pack ops operating on **weight tensors**.
These ops are executed **only once, during prefill** and **aren't executed in decode** at all, leading to higher perf gains in decode

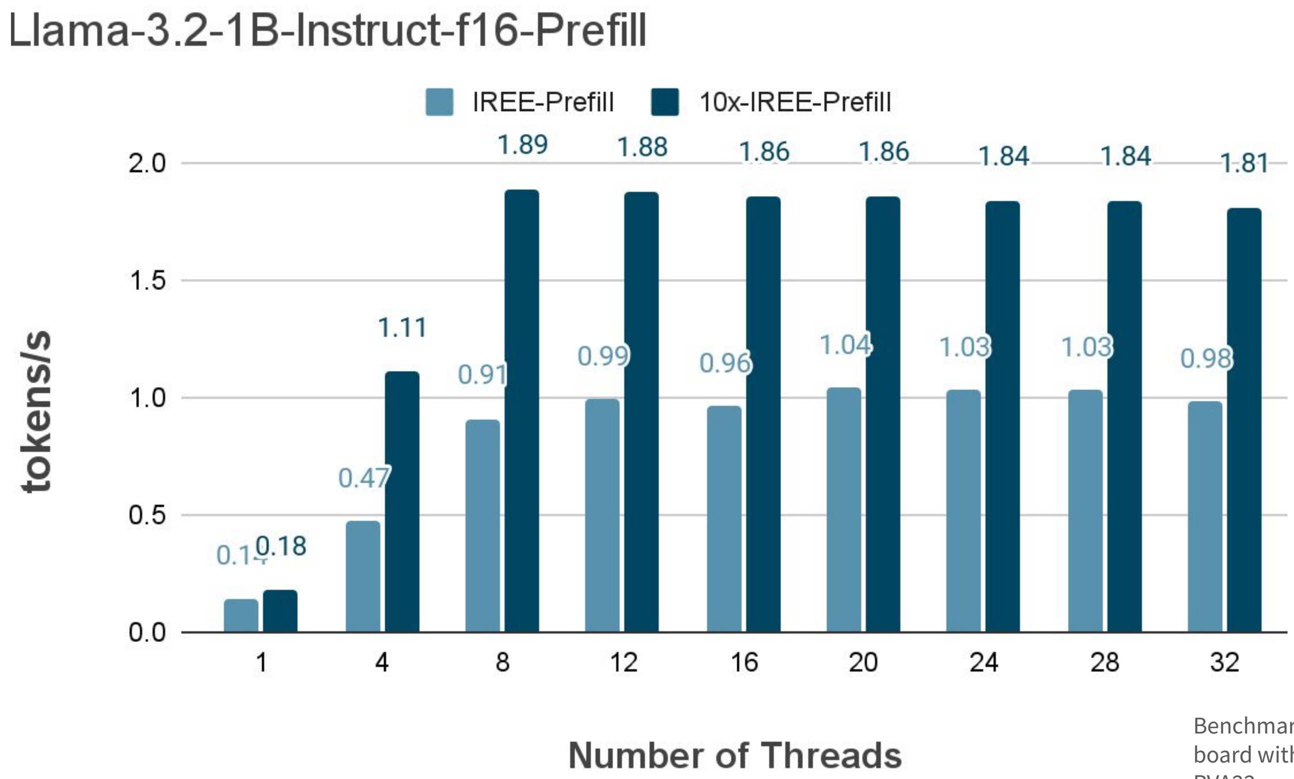


```
Name                                  Location                                       Total time
_initializer_162_dispatch_0_pack_f16  mlir/llama-3.2-1B-Instruct-F16.mlir:9954       56.47 s  (35.21%)
_initializer_160_dispatch_0_pack_f16  mlir/llama-3.2-1B-Instruct-F16.mlir:9922       19.36 s  (12.07%)
_initializer_10_dispatch_0_pack_f16   mlir/llama-3.2-1B-Instruct-F16.mlir:951        19.28 s  (12.02%)
_initializer_12_dispatch_0_pack_f16   mlir/llama-3.2-1B-Instruct-F16.mlir:987        12.28 s  (7.66%)
_initializer_268_dispatch_0_pack_f16  mlir/llama-3.2-1B-Instruct-F16.mlir:17626      9.5 s  (5.93%)
_initializer_148_dispatch_0_pack_f16  mlir/llama-3.2-1B-Instruct-F16.mlir:8985       9.41 s  (5.87%)
_initializer_156_dispatch_0_pack_f16  mlir/llama-3.2-1B-Instruct-F16.mlir:9477       4.97 s  (3.10%)
_initializer_0_dispatch_0_pack_f16    mlir/llama-3.2-1B-Instruct-F16.mlir:500        4.85 s  (3.02%)
```

These pack ops can be evaluated at **compile-time** by enabling **const-eval** optimization in IREE !!!

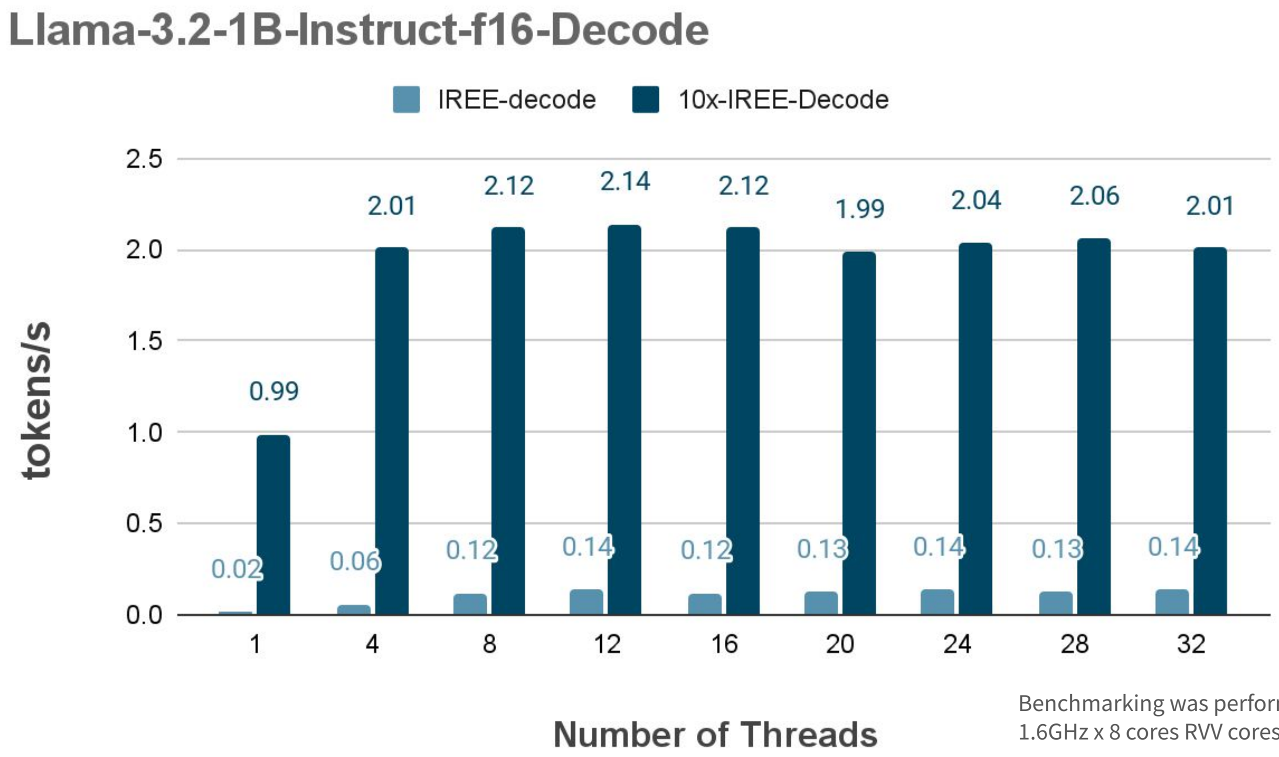
Llama.cpp commit hash:76b27d2, Nov28
Benchmarking was performed on MILK-V Jupiter board with 1.6GHz x 8 cores RVV cores, VLEN=256, RVA22

Llama-3.2-1B-Instruct-f16 Benchmarking - Prefill



Benchmarking was performed on MILK-V Jupiter board with 1.6GHz x 8 cores RVV cores, VLEN=256, RVA22

Llama-3.2-1B-Instruct-f16 Benchmarking - Decode



Benchmarking was performed on MILK-V Jupiter board with 1.6GHz x 8 cores RVV cores, VLEN=256, RVA22

Acknowledgements

1. SiFive - LLM Optimization and Deployment on SiFive RISC-V Intelligence Products, URL: https://www.sifive.com/blog/llm-optimization-and-deployment-on-sifive-intellig
2. IREE Blogs - Matrix Multiplication with MMT4D, URL: https://iree.dev/community/blog/2021-10-13-matrix-multiplication-with-mmt4d/
3. IREE Blogs - Exploring CPU microkernels on a matmul example, URL: https://iree.dev/community/blog/2024-01-22-exploring-cpu-microkernels-on-a-matmul-example/

Summary

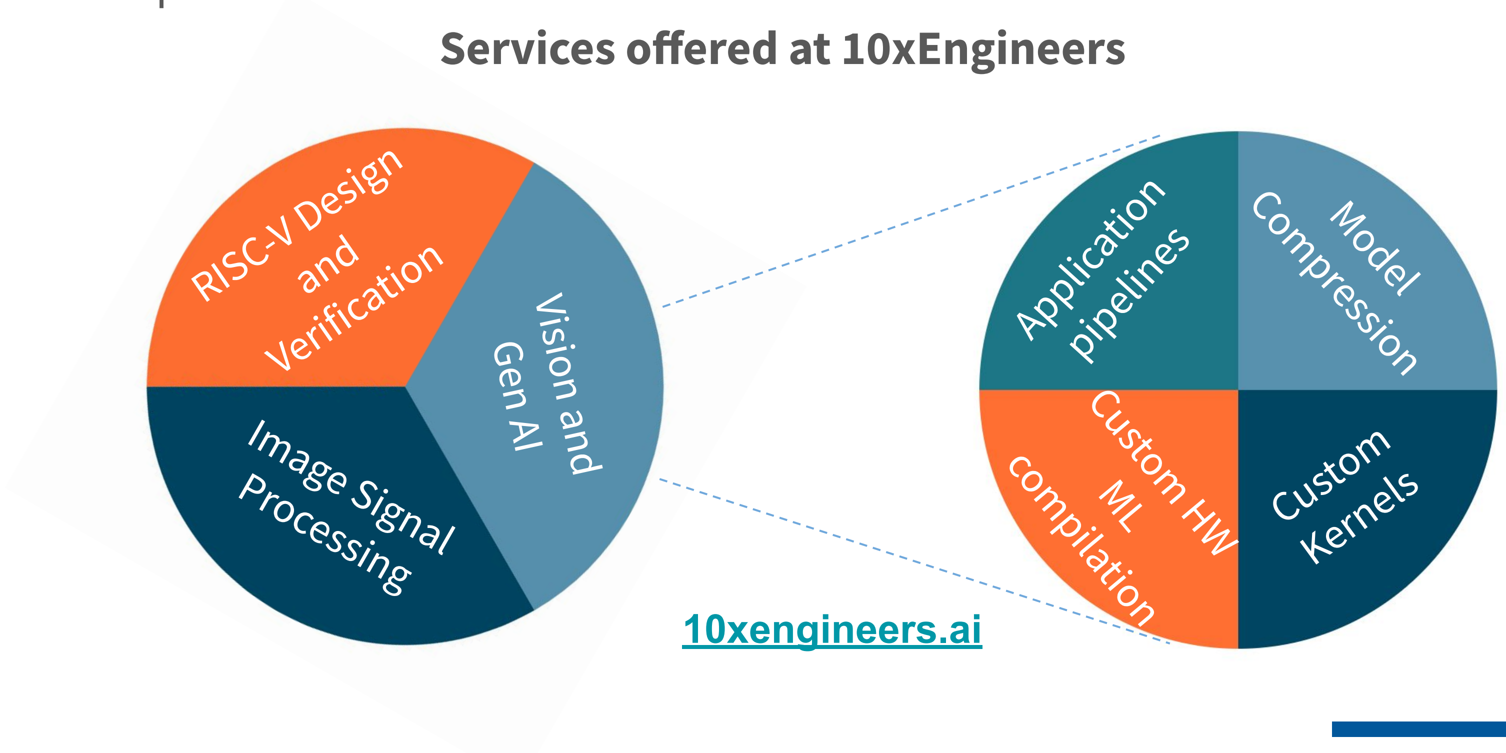
- Custom matrix multiplication microkernels were implemented in IREE producing ~2x and ~50x single-thread runtime improvements during the LLM pre-fill and decode phases; results are validated on a multi-core MILK-V Jupiter board
- One-time data prepacking cost is incurred on the first token generation i.e., during the prefill; consteval at compile-time can remove this one-time packing cost
- ukernels to be pushed to upstream IREE
- Need to improve the baselines for HPC on RISC-V
    - More open-source contributions/collaborations to improve RISC-V based ML kernels on major platforms like llama.cpp, IREE, etc serves the whole RISC-V community

Thank you

Let’s chat

Visit our posters!

Services offered at 10xEngineers



10xengineers.ai