Deploying BEV Perception Models on RISC-V: A Practical Look into Next-Gen Automotive AI on the SiFive Intelligence XM Platform

Authors: [Alex Chiang](mailto:alex.chiang@sifive.com), [Hong-Rong Hsu](mailto:hongrong.hsu@sifive.com), [Jerry Shih](mailto:jerry.shih@sifive.com), [Phoebe Chen](mailto:phoebe.chen@sifive.com)

# Introduction

Bird’s Eye View (BEV) perception has become a key technique in modern autonomous driving systems. By transforming multi-camera inputs into a unified top-down representation, BEV models support tasks such as object detection, spatial reasoning, and multi-view fusion—crucial for safe and efficient navigation.

Among these, **BEVFormer** stands out as a representative next-generation model. It extends traditional BEV pipelines with transformer-based temporal attention, enabling perception systems to reason across multiple frames and camera views for improved stability and context awareness.

In this blog, we share our experience deploying BEVFormer models on the SiFive Intelligence XM platform—a RISC-V-based AI platform built on the open RISC-V standard. Through this practical deployment journey, we aim to demonstrate the feasibility of executing complex perception workloads using an open and flexible software stack—centered on IREE—built on SiFive’s LLVM compiler and SKL high-performance library.

*(Note: We began our exploration with BEVDepth - a comparatively simpler BEV model, which served as a useful starting point for validating the overall toolchain. The experience laid the groundwork for deploying more complex models like BEVFormer, which we focus on in this blog. If you're interested in the BEVDepth deployment journey, contact us at* [*sales@sifive.com*](mailto:sales@sifive.com) *we would love to hear from you!)*

# Why RISC-V and SiFive XM Platform for Automotive AI?

Automotive AI requires compute-platforms that balance performance, flexibility, and long-term sustainability. Traditional fixed-architecture processors often fall short in customization, power efficiency, or system integration—factors that are critical for in-vehicle deployment.

RISC-V is an open-standard ISA with a modular design that allows developers to tailor processor designs to specific workloads, including vector processing, neural inference, and real-time control. This flexibility is especially valuable in automotive systems, where long lifecycles and strict functional requirements demand greater control over compute architecture.

The SiFive Intelligence XM platform builds upon this foundation with a multi-core design consisting of four X-Cores and a tightly coupled Matrix Engine. Each X-Core—such as the X390 —features a programmable RISC-V Vector (RVV) unit with a 1024-bit vector length, enabling high-throughput SIMD execution.

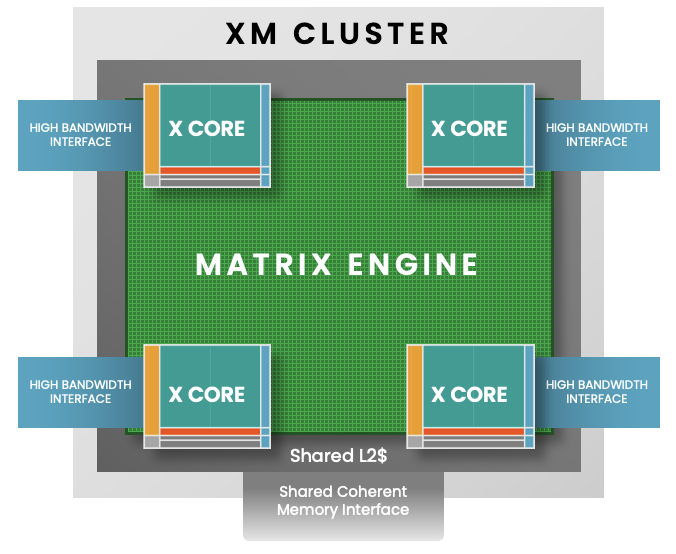


Figure 1: XM Cluster is composed of 4 X-cores with a Matrix engine

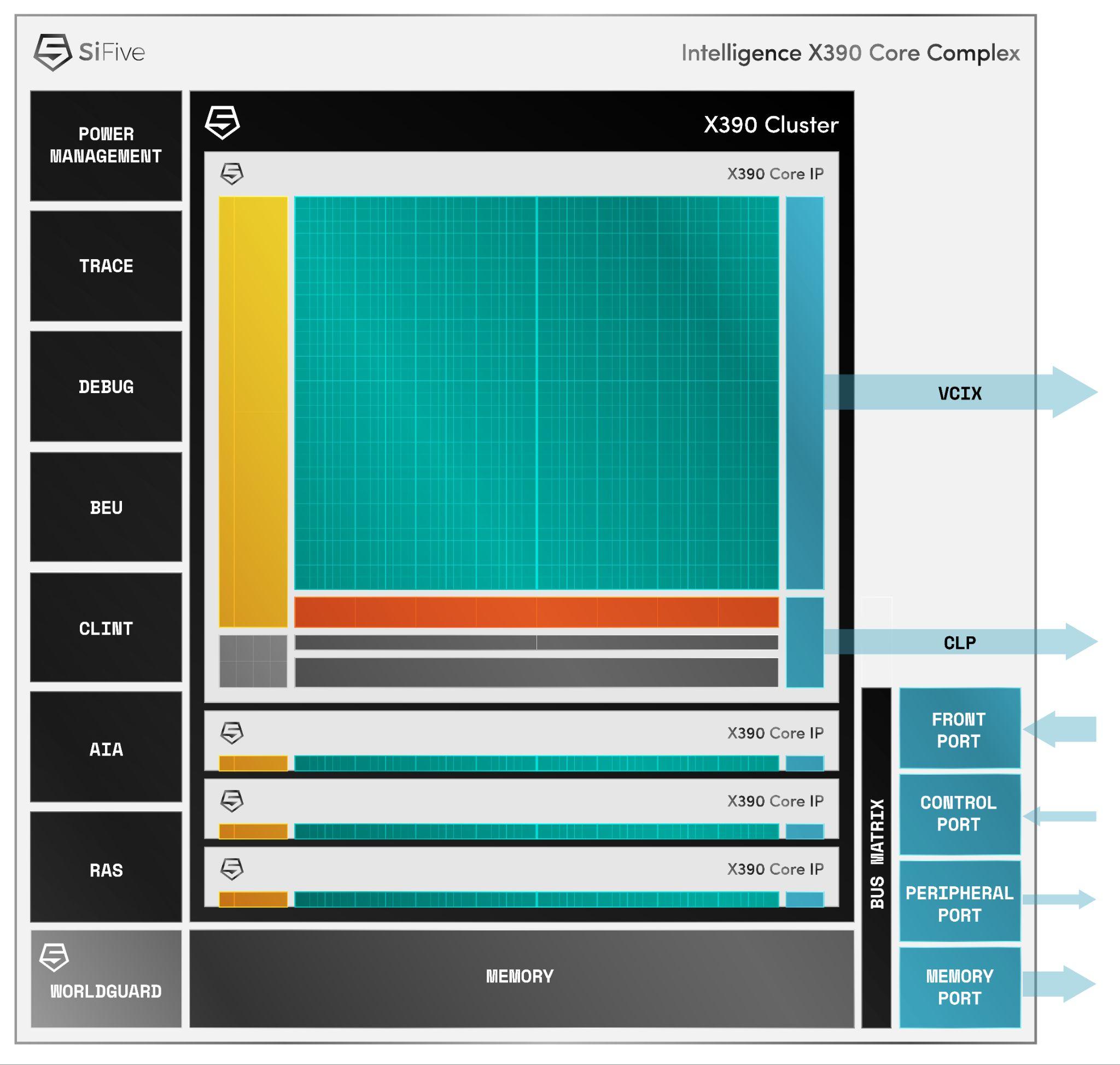


Figure 2: Diagram of X390 core.

The Matrix Engine complements the vector units with a fat outer product-based matrix multiply accelerator, designed to work seamlessly with the RVV pipelines. It reads input matrices directly from the vector registers, performs matrix operations, and returns results either back to registers or to memory — eliminating the latency and overhead typically found in discrete accelerator designs. In the meantime, the rest of idle vector cores can still perform independent workload computing in parallel.

This tightly integrated architecture directly addresses a key challenge in heterogeneous AI systems: the high cost of falling back between compute subsystems. In conventional SoCs, when accelerators encounter unsupported operations (like non-linear functions or irregular memory access), the system often falls back to the CPU, introducing interrupt latency, data movement, synchronization delays, and added software complexity from subgraph partitioning and orchestration.

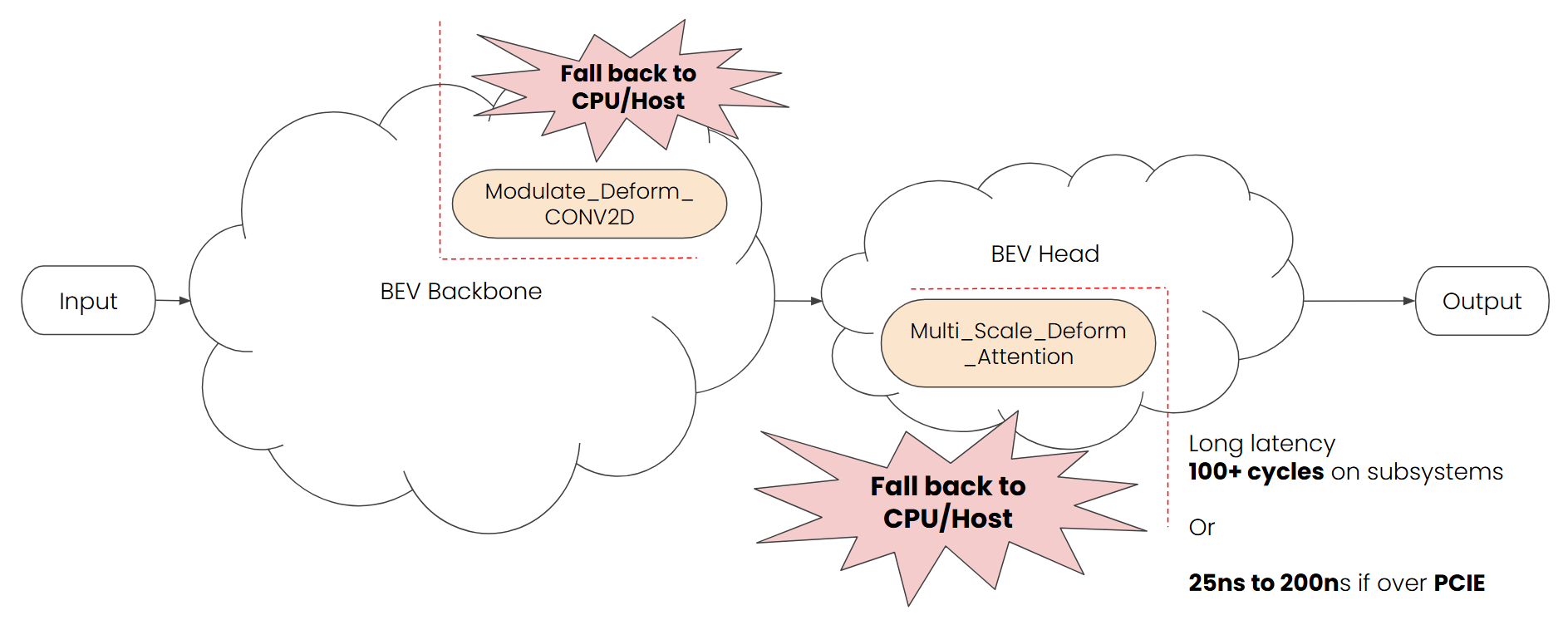


Figure 3: Split Graph for fallback in general heterogenous systems.

In contrast, the SiFive XM platform allows different workloads to run efficiently across compute units without explicit offloading or data marshaling. Taking BEVFormer as an example:

* CNN layers are executed on the Matrix Engine
* Irregular but parallelizable ops like deform\_conv2d and deformable attention are handled by X-Core using RVV
* Scalar and control-flow operations are processed by the X-Cores

All components share a unified memory space and are managed by a single compiler flow, enabling the generation of a unified binary that runs seamlessly across scalar, vector, and matrix units. This simplifies development while ensuring performance and consistency across the AI stack. Especially, significantly reduce the fall back penalty among compute subsystems in traditional heterogeneous architecture.

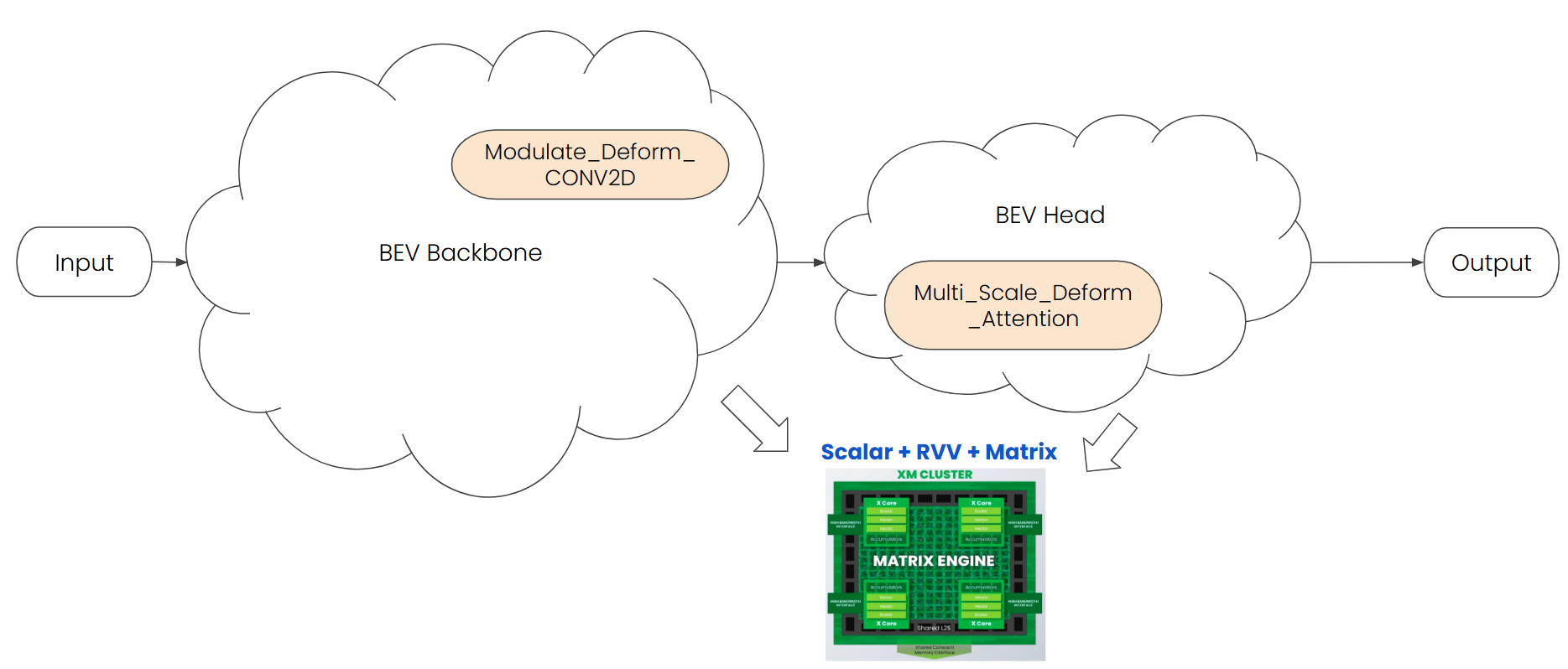


Figure 4: Unified Graph be executed in an unified platform getting lowest latency

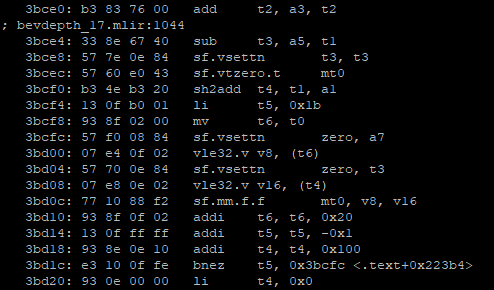


Figure 5: The unified binary of Scalar + RVV + Matrix

# Overview of BEVFormer Models

BEVFormer is a transformer-based camera perception model designed to generate a unified bird’s eye view (BEV) representation of the driving scene. It supports temporal fusion and multi-view image processing, making it highly representative of next-generation 3D perception workloads in autonomous systems.

The model consists of three main modules: backbone, neck, and transformer (encoder + decoder). In this section, we describe the architecture based on the commonly used bevformer-base configuration:

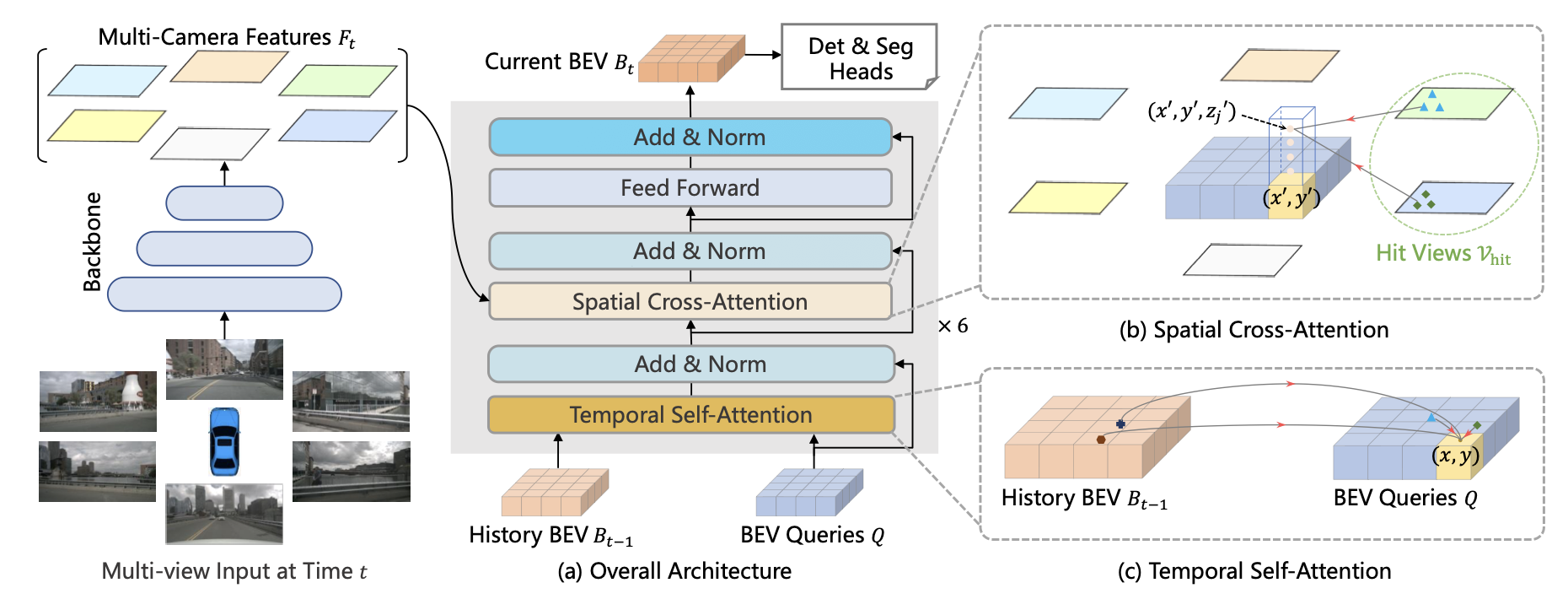
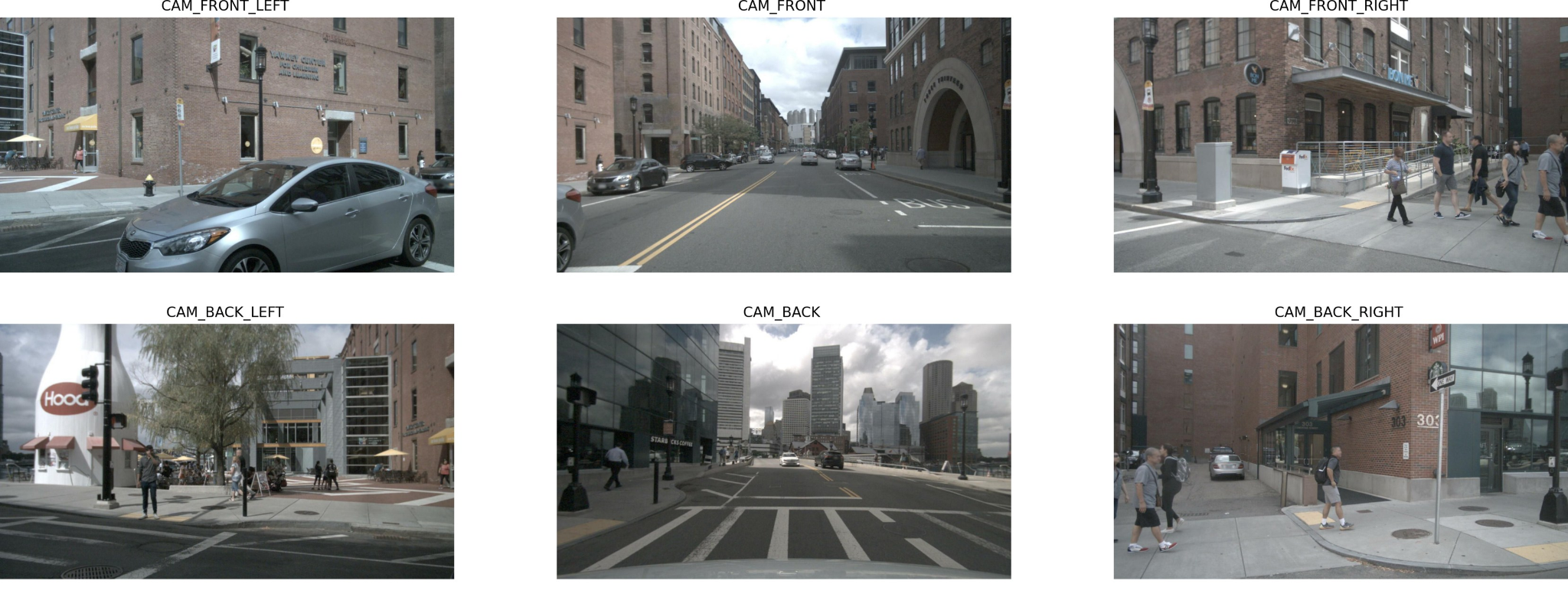
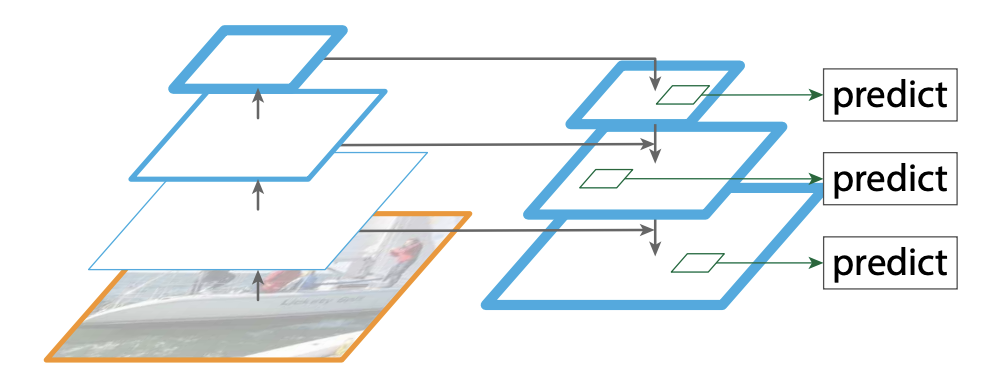


Figure 6: BEVFormer architecture (<https://github.com/fundamentalvision/BEVFormer/blob/master/figs/arch.png>)

Backbone: Extracts multi-scale image features from input 6 camera images using ResNet101 with deformable convolutions (DCN).



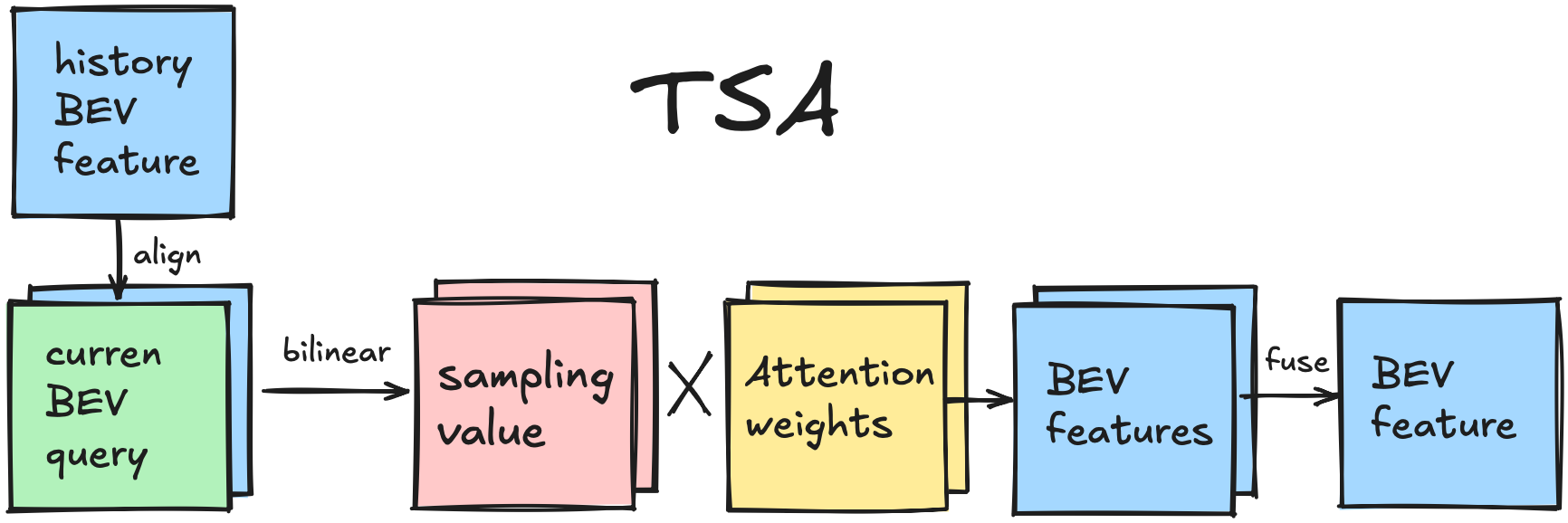
Neck: Uses a Feature Pyramid Network (FPN) to fuse features across different spatial resolutions.



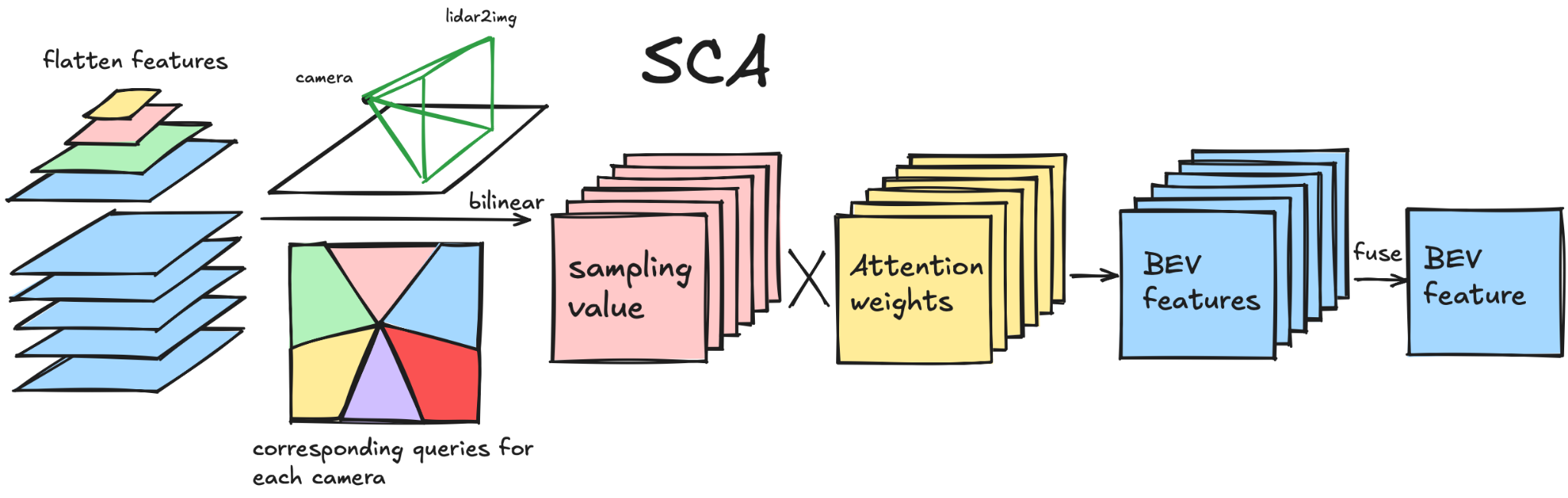
Transformer: The core module of BEVFormer. The encoder fuses spatial and temporal information, while the decoder generates object queries and refines them into final object detections.

The **encoder** includes two major components:

* Temporal self-attention (TSA): TSA aligns the historical BEV features and fuses information from the current and previous BEV features.



* Temporal self-attention (SCA): Fuses the multi-view camera images and calculates them with corresponding queries to generate BEV features.



The decoder is responsible for object-level reasoning. It takes a set of learnable object queries and performs multi-layer cross-attention with the encoded BEV features. Through iterative refinement, it generates final bounding box predictions and class scores for detected objects in the BEV space.

In this blog, we focus on mapping BEVFormer’s inference path—including its transformer components—onto the SiFive Intelligence XM platform, and show how different modules are executed across scalar cores, RVV vector units, and the Matrix Engine.

# Deployment Strategy on SiFive XM Platform

## BEVFormer Deployment Environment

Paper: <https://arxiv.org/abs/2203.17270>

Github: <https://github.com/fundamentalvision/BEVFormer/tree/master>

Environment Requirements:

* Python: 3.9.21
* Torch: 1.9.1

Dataset & Dependencies:

* Dataset: nuScenes(<https://www.nuscenes.org/download>)
* OpenMMLab
  + mmcv-full: 1.4.0
  + mmdet: 2.14.0
  + mmdet3d: 0.17.1
  + mmsegmentation: 0.14.1

Model Config:

* Config name: BEVFormer-base
* Inputs: 6(#camera) x 900 x 1600
* BEV size: 200 x 200

Deployment:

* Software: SiFive AI/ML Software Stack
* Hardware: SiFive XM Emulator (take QEMU as the example here)

## IREE and SiFive AI/ML Software Stack

IREE is a core part of our software stack, enabling us to compile and run AI models efficiently on the SiFive XM platform. It supports multiple front-ends—including PyTorch—and provides the flexibility needed to target both RVV and the Matrix Engine. For more details on the overall stack, see our previous blog: “[LLM Optimization and Deployment on SiFive RISC-V Intelligence Products](https://www.sifive.com/blog/llm-optimization-and-deployment-on-sifive-intellig)”

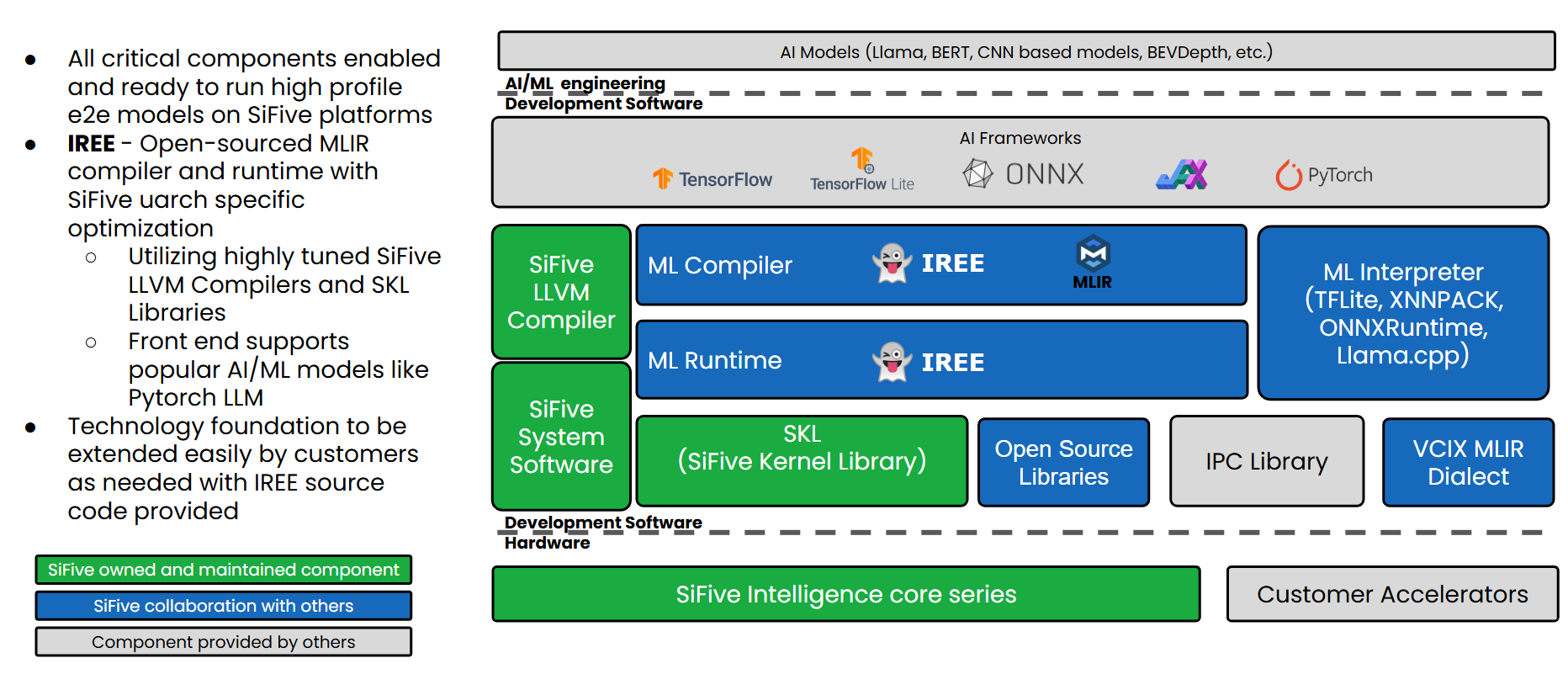


Figure 7: SiFive AI/ML Software Stack

## End-to-End Flow Overview

The first challenge is converting the BEVFormer project from PyTorch to MLIR. Since BEVFormer is based on “PyTorch 1.9.1”, it does not support newer export methods such as “torch.export + torch-mlir” or [IREE PyTorch AOT APIs](https://iree.dev/guides/ml-frameworks/pytorch/#ahead-of-time-aot-export), which require “PyTorch 2.0” or above. Fortunately, “PyTorch 1.9.1” supports ONNX export, which allows us to convert the model to ONNX format and then lower it to MLIR using IREE’s “[iree-import-onnx](https://iree.dev/guides/ml-frameworks/onnx/)” tool.

From there, the IREE compiler takes the imported MLIR and performs a series of lowering and optimization passes. The output is a “.vmfb” (IREE executable) that contains code targeting RISC-V scalar cores, RVV vector units, and the Matrix Engine, which will be executed by the IREE runtime “iree-run-module”.

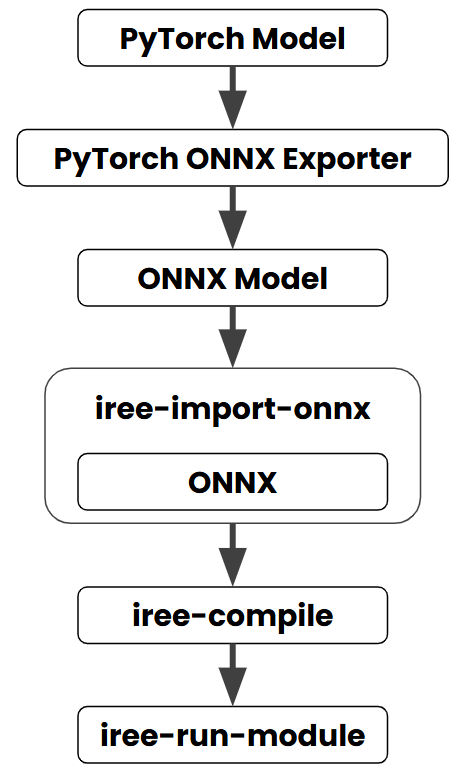


Figure 8: The e2e flow of BEVFormer

*(Note: You might wonder why we didn’t upgrade BEVFormer to PyTorch 2.x. The reason lies in* [*OpenMMLab*](https://github.com/open-mmlab) *dependencies. BEVFormer relies on specific versions of OpenMMLab modules, each of which is tied to a particular PyTorch version. This tight coupling makes upgrading PyTorch extremely difficult without breaking compatibility across the stack.)*

## Custom Operators Support

Before diving into each phase of the end-to-end flow, we’d like to highlight the challenges associated with supporting custom operators.

What are Custom Operators? Custom operators are user-defined functions or kernels that extend a framework's built-in operations, allowing support for novel algorithms or hardware-specific computations. Deploying custom operators is challenging due to limited toolchain support, backend incompatibilities, and the need for manual integration and maintenance. They often require extra effort to register, lower, and implement across different runtimes and platforms.

In our BEVFormer deployment, we implemented several custom operators as optimized microkernels to ensure compatibility and simplify future maintenance. These are seamlessly integrated into the IREE compiler and runtime stack through a modular and extensible workflow.

If you’re facing similar challenges or would like to learn more about our approach, we welcome you to contact our [sales team](https://www.sifive.com/contact-sales) to explore how SiFive can support your deployment needs.

We also include a simplified example of custom op lowering later in this blog.

### Pytorch → ONNX

We encountered several custom operators in the BEVFormer model. To convert BEVFormer from PyTorch to ONNX format, we register these ops in PyTorch using “symbolic” API. Once all required ops are registered, we use “torch.onnx.export” to export the PyTorch model to ONNX. After the export, we apply a post-processing pipeline to simplify the ONNX graph and make it more robust for downstream compilation.

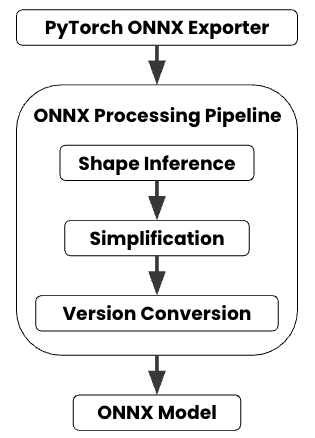


Figure 9: ONNX Processing Pipeline

As shown in the diagram above, the ONNX model passes through a series of transformations:

1. Shape Inference: Determines tensor shapes based on known input/output sizes, ensuring all operators have valid shape information.
2. Simplification: Uses tools like “onnxsim” to perform constant folding and eliminate redundant operations.
3. Version Conversion: Converts the model to ONNX IR version 17 to align with the IREE compiler.

Tools such as [Netron](https://netron.app/) can be used to inspect the exported model and verify that custom ops like “multi\_scale\_deformable\_attention” are correctly embedded.

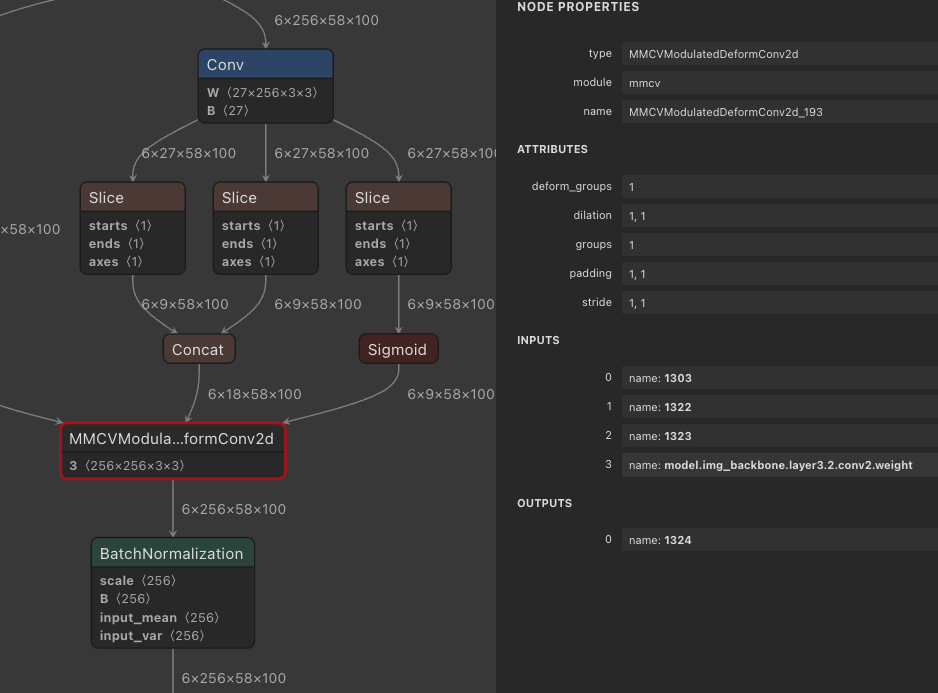


Figure 10: Custom OP - Modulated Deform Conv2d

### ONNX → Torch-MLIR

We use the IREE tool **“iree-import-onnx”** to convert the ONNX model into MLIR(Torch-MLIR) format:

|  |
| --- |
| iree-import-onnx -o bevformer.mlir bevformer.onnx |

At this point, we have an unified MLIR representation of the BEVFormer model, with all custom operators preserved and ready for downstream lowering and optimization within the IREE compiler.

### Torch-MLIR → IREE Compilation

Lowering ONNX custom ops via standard MLIR constructs is feasible but often requires significant effort and boilerplate. For maintainability and flexibility, we opt to encapsulate these ops using **IREE::Codegen::UKernelGenericOp**, which allows direct implementation of the logic in C/C++. These microkernels can then be optimized through llvm-riscv compiler auto-vectorization or handcrafted RVV intrinsics and assembly for performance tuning.

The lowering logic is integrated into the **InputConversionPass**, where we extend the pass pipeline to recognize ONNX-based custom operators. The c/c++ implementations for custom-op are placed in the **ukernel\_bitcode\_generic** library, and are automatically linked into the final IREE binary (vmfb) during compilation.

This approach provides a modular structure that encapsulates custom logic in microkernels, while the IREE compiler orchestrates the lowering, code generation, and final linking. It also makes the pipeline easier to maintain and extend as new custom ops or hardware features are introduced.

## How to Map matmul to the Matrix-Engine through IREE

We integrated the SiFive Matrix Engine into the IREE compiler via the [mmt4d](https://iree.dev/community/blog/2021-10-13-matrix-multiplication-with-mmt4d/). mmt4d provides a multi-level tiled matrix multiplication framework and allows users to customize it to match specific memory layouts.

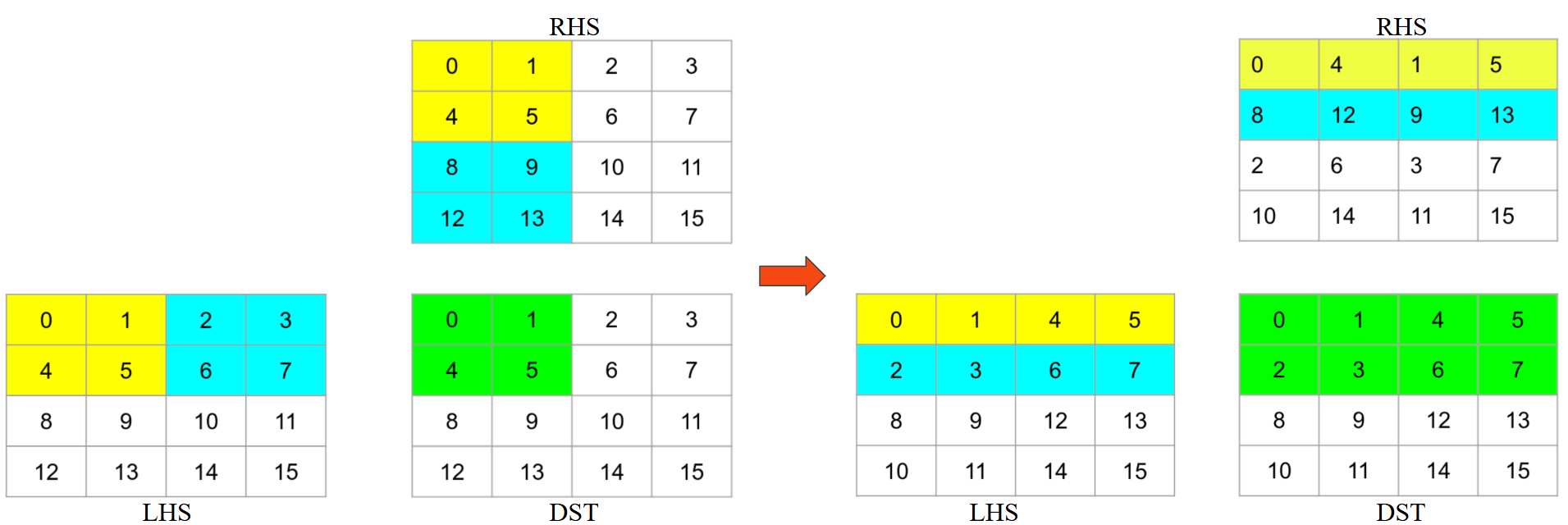


Figure 11: Original memory access and matrix-multiply → After packing for better memory access pattern.

The SiFive Matrix Engine implements a fat outer-product operation, which requires the A matrix to be transposed before feeding into the hardware. ([The hardware specification](https://lists.riscv.org/g/tech-attached-matrix-extension/message/257?fbclid=IwZXh0bg%5B%E2%80%A6%5DKba5b5JNoybpeo_I4nC9IOqX7XIUMDXBqs_aem_IwLssbbew1R80KiynX4BAg) is publicly available, but we will not dive into those details in this blog.)

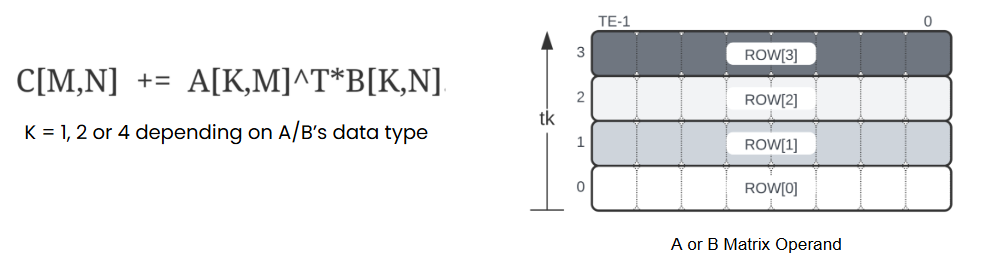


Figure 12: Matrix-Multiply of SiFive XM

Since the default memory layout used by mmt4d does not align with the Matrix Engine’s requirements, we introduced a new operator called MMT4D\_TransposeA, which reorganizes the inner tiles from [M, K] and [N, K] to **[K, M] and [K, N]**. This matches the expected layout for the fat outer-product kernel. We then integrated the **TExTE** (tile-element) microkernel, which represents the matrix compute block of the engine, into the innermost loop of mmt4d. Padding and outer-loop tiling are handled automatically by the IREE compiler.

At compile time, **contraction operations**—such as matmul, fully connected, and even some forms of conv2d—can be lowered to mmt4d and dispatched to the Matrix Engine via the registered microkernel. The compiler analyzes the problem size to determine whether to route the workload to **RVV** or the **Matrix Engine**, ensuring that the most efficient backend is used for each case.

## How to Map CUDA to SiFive XM

In many PyTorch projects—including BEVFormer, which we focus on in this blog—custom operators are often implemented in CUDA to achieve high performance. When targeting platforms like SiFive Intelligence XM, a key challenge arises: how to map CUDA implementations to portable C/C++ and RVV-based code that runs efficiently on RISC-V architectures.

When porting CUDA kernels to C/C++, one of the key considerations is maintaining parallel performance through effective vectorization. In many cases, CUDA kernels are inherently structured in a way that can be well vectorized. Modern compilers are typically able to **automatically vectorize the innermost loops**, especially when the code follows predictable access patterns.

To further improve vectorization, developers can use compiler directives, attributes, and optimization flags to guide or enforce vectorization. However, in performance-critical paths, manual tuning is often necessary. By leveraging **RISC-V Vector (RVV) intrinsics**, developers can handcraft optimized vector code that achieves near-peak hardware utilization, significantly improving throughput over default compiler-generated code.

Overall, an effective CUDA-to-RISC-V porting strategy combines automatic compiler vectorization, manual tuning with RVV, and LLM-assisted generation, offering a flexible path toward optimized execution on open and programmable hardware platforms like SiFive XM.

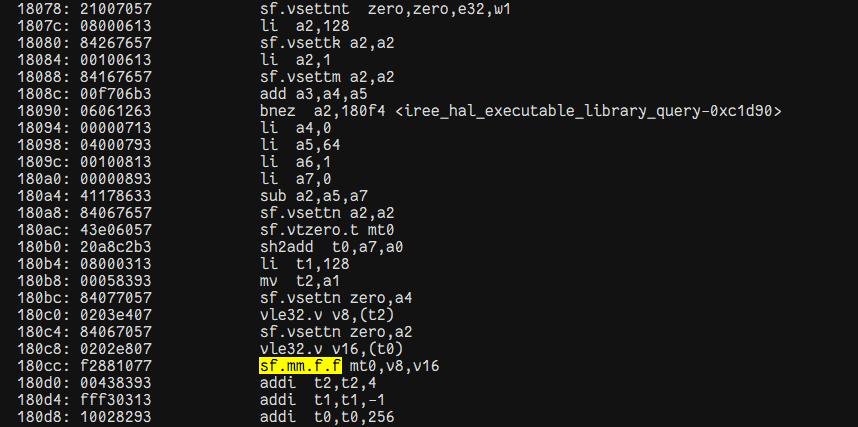
## Compiling and Running the Compiled Model on SiFive XM

To deploy models on the SiFive XM platform using IREE, we first compile the model into a VM flatbuffer (.vmfb) format using “iree-compile”. This bytecode format is optimized for execution by the IREE runtime.  
The compilation command typically looks like this:

|  |
| --- |
| **<path\_to>/iree-compile** \  --output-format=vm-bytecode \  --mlir-print-op-on-diagnostic=false \  --iree-llvmcpu-target-triple=riscv64-pc-linux-gnu \  --iree-llvmcpu-target-cpu=generic-rv64 \  --iree-llvmcpu-target-cpu-features=+m,+a,+f,+d,+zba,+zbb,+zfh,+zvl1024b,+v,+zvfh,+xsfmm32a,+xsfmm64t,+xsfvcp \  --iree-llvmcpu-target-abi=lp64d --iree-llvmcpu-riscv64-tile-size=64 \  --iree-global-opt-pad-factor=64 \  --iree-llvmcpu-stack-allocation-limit=655360 \  --riscv-v-fixed-length-vector-lmul-max=8 \  --iree-hal-target-backends=llvm-cpu \  --iree-llvmcpu-enable-ukernels=all \  --iree-preprocessing-enable-conv2d-to-img2col=true \  --iree-opt-data-tiling \  --iree-llvmcpu-ukernel-mmt4d-inner-tile-format=kmkn \  --iree-llvmcpu-matmul-narrow-n-swap=false \ **bevformer.mlir** -o **bevformer.vmfb** |

This generates an output file named “bevformer.vmfb”, which contains the executable IREE bytecode suitable for deployment.

To verify and inspect the contents of the compiled binary, you can use tools like “objdump”. When analyzing the output, you’ll notice the presence of instructions such as “sf.mm.f.f”.



These represent matrix multiplication acceleration instructions specific to the **SiFive XM** platform, and confirm that the IREE compiler has successfully targeted the hardware’s matrix engine for optimized execution.

Once the model is compiled and the .vmfb file is ready, we can execute the inference flow using QEMU to emulate the SiFive XM platform. We have seamlessly integrated IREE runtime into the BEVFormer end-to-end (e2e) pipeline, enabling direct inference and result generation.

To run evaluation with QEMU, simply use the following command:

|  |
| --- |
| ./tools/dist\_test.sh ./projects/configs/bevformer/bevformer\_base.py ./ckpts/bevformer\_r101\_dcn\_24ep.pth 1 --qemu\_test |

--qemu\_test: This flag triggers inference using the QEMU-emulated SiFive XM environment. When this flag is specified, the runtime wraps the target command with a detailed QEMU invocation that emulates a RISC-V system supporting an extensive set of SiFive-specific ISA extensions.

Specifically, the following QEMU command is executed when --qemu\_test is used:

|  |
| --- |
| qemu-riscv64 -cpu qmp-rv64,i=true,m=true,a=true,f=true,d=true,c=true,s=true,u=true,h=true,v=true,sscofpmf=true,zicsr=true,zifencei=true,zihintntl=true,zihintpause=true,zawrs=true,zfa=true,zfh=true,sstc=true,smstateen=true,svade=true,svinval=true,svnapot=true,svpbmt=true,ssqosid=true,sdtrig=true,zba=true,zbb=true,zbs=true,zcb=true,zkr=true,zkt=true,zicbom=true,zicbop=true,zicboz=true,zvbb=true,zvkt=true,x-xsfvfbfa=true,x-xsfvfexpa=true,x-xsfvfnrclipxfqf=true,x-xsfvfwmaccqqq=true,x-xsfvqdotq=true,x-xsfvqmaccqoq=true,x-xsfmmbase=true,x-xsfmm32a=true,x-xsfmm64t=true,zicond=true,smaia=true,ssaia=true,smrnmi=true,x-ssnpm=true,x-smnpm=true,x-smmpm=true,zvfh=true,zfbfmin=true,zvfbfmin=true,zvfbfwma=true,x-xsifivecdiscarddlone=true,x-xsifivecflushdlone=true,xsfcease=true,xsfpgflushdlone=true,zimop=true,zcmop=true,vlen=1024,elen=64,priv\_spec=v1.13.0,vext\_spec=v1.0,x-xsfvcp=true,x-xsfvcp\_impl=exp  -L  <path\_to>/riscv64-unknown-linux-gnu-toolsuite-3.0.8-x86\_64-linux-redhat8/sysroot  <path\_to>/iree-run-module  --task\_worker\_stack\_size=4194304  --module\_mode=preload  --device\_allocator=caching  --device=local-task  --task\_topology\_group\_count=32  --module=bevformer.vmfb  --function="torch-jit-export" |

The accuracy of the QEMU results shows no accuracy lost compared to the PyTorch golden.

After running inference, the results can be visualized using:

|  |
| --- |
| python3.9 tools/analysis\_tools/visual.py test/$MODEL/$DATE/pts\_bbox/results\_nusc.json |

This script generates visualizations based on the inference outputs, making it easier to qualitatively assess model accuracy.

The visualized images reflect inference performed under QEMU emulation of the **SiFive XM** platform:

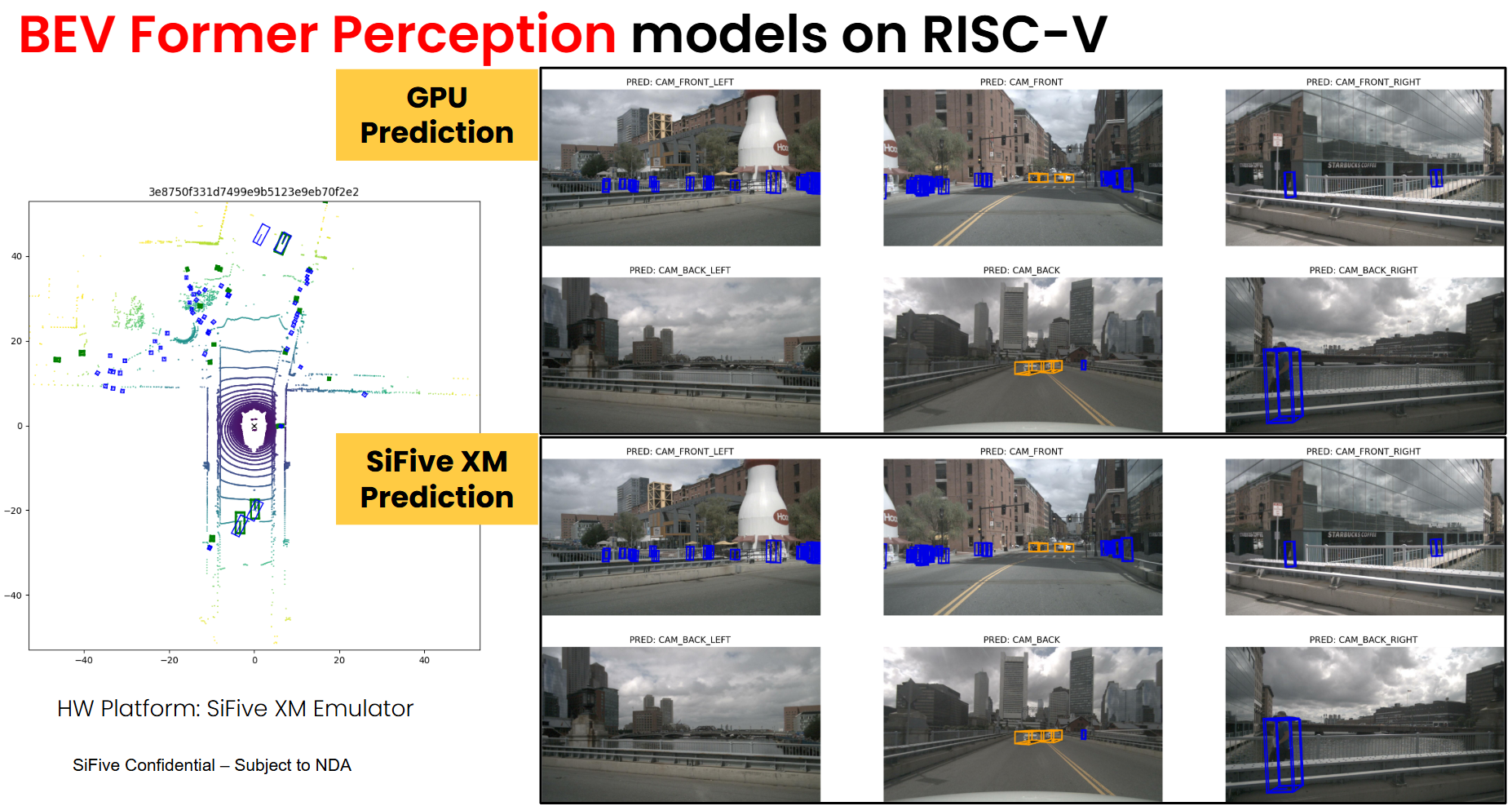


Figure 13: The inference BEV outputs on SiFive XM Emulator.

# Summary

We successfully deployed the BEVFormer model end-to-end on the SiFive XM platform. From PyTorch model export to final runtime execution, the entire flow was integrated through IREE, generating a single binary capable of running across scalar cores, RVV vector units, and the Matrix Engine. Key transformer modules, including TemporalSelfAttention and SpatialCrossAttention, were mapped to RVV and executed correctly without fallback, demonstrating the system’s ability to handle complex, memory-irregular operations.

Convolutional and matmul-heavy components were offloaded to the Matrix Engine using IREE’s mmt4d abstraction, with compiler-emitted matrix instructions confirming that hardware acceleration was properly utilized. In addition, we implemented and integrated several custom operators to support critical parts of the BEVFormer architecture. The full pipeline was validated through inference on the nuScenes dataset using QEMU emulation, and visualized outputs confirmed that results remained consistent and meaningful throughout the process.

# Looking Ahead

This deployment marks a significant step toward enabling advanced perception models for automotive applications on RISC-V. We are actively optimizing performance on the SiFive XM platform by tuning key kernels and improving code generation across our software stack.

As RISC-V adoption grows in the automotive AI domain, we welcome opportunities to collaborate with partners looking to bring real-world workloads to open and programmable platforms. If you're interested in working with us or evaluating the SiFive Intelligence Series for your next-generation automotive AI systems, please reach out through [SiFive Sales Contact](https://www.sifive.com/contact-sales).