Ryzen AI

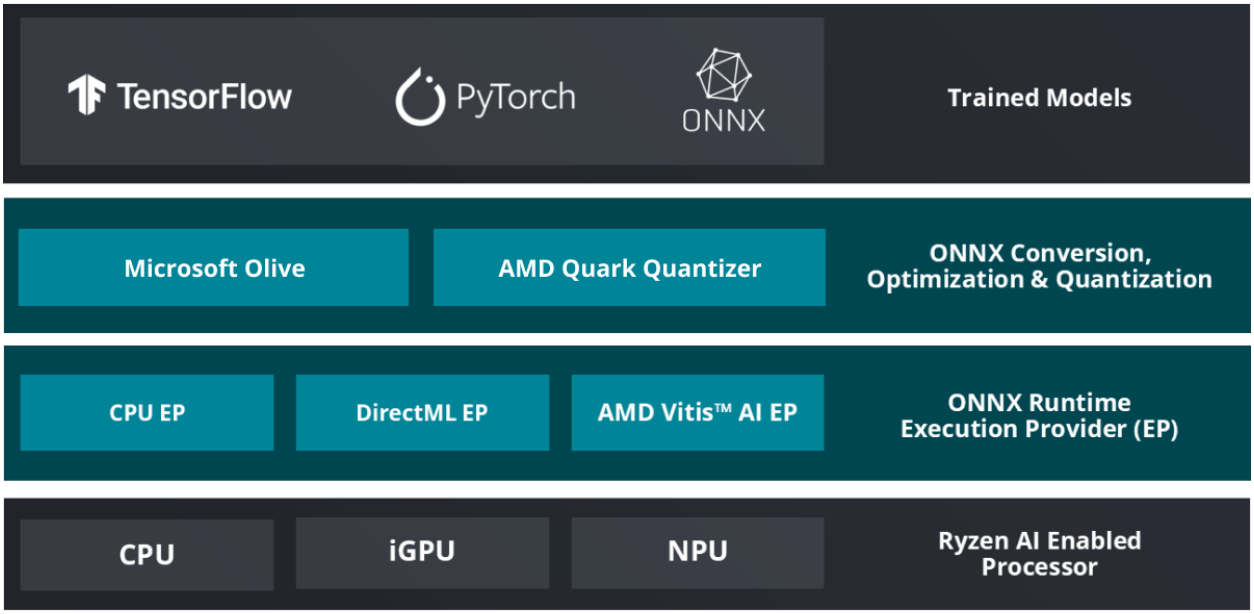
Release 1.7.1

AMD

Apr 01, 2026

CONTENTS

AMD Ryzen™ AI Software includes the tools and runtime libraries for optimizing and deploying AI inference on AMD Ryzen™ AI powered PCs. Ryzen AI software enables applications to run on the neural processing unit (NPU) built in the AMD XDNA™ architecture, as well as on the integrated GPU. This allows developers to build and deploy models trained in PyTorch or TensorFlow and run them directly on laptops powered by Ryzen AI using ONNX Runtime and the Vitis™ AI Execution Provider (EP).



CHAPTER

ONE

QUICK START

- *Supported Configurations*
- *Installation Instructions*
- *Examples, Demos, Tutorials*

CHAPTER

TWO

DEVELOPMENT FLOW OVERVIEW

The Ryzen AI development flow does not require any modifications to the existing model training processes and methods. The pre-trained model can be used as the starting point of the Ryzen AI flow.

2.1 Quantization

Quantization involves converting the AI model's parameters from floating-point to lower-precision representations, such as 8-bit integer. Quantized models are more power-efficient, utilize less memory, and offer better performance. Ryzen AI Software also supports CNN and Transformer models in floating-point 32 format as input models without quantization. These models are internally converted to bfloat16 and compiled using the bfloat16 compilation flow.

AMD Quark is a comprehensive cross-platform deep learning toolkit designed to simplify and enhance the quantization of deep learning models. Supporting both PyTorch and ONNX models, Quark empowers developers to optimize their models for deployment on a wide range of hardware backends, achieving significant performance gains without compromising accuracy.

For more details, refer to the *Model Quantization* page.

2.2 CNN/Transformer Compilation and Deployment

The AI model is deployed using the ONNX Runtime with either C++ or Python APIs. The Vitis AI Execution Provider included in the ONNX Runtime intelligently determines what portions of the AI model should run on the NPU, optimizing workloads to ensure optimal performance with lower power consumption.

For more details, refer to the *Model Compilation and Deployment* page.

CHAPTER

THREE

LLM FLOW OVERVIEW

The Ryzen AI LLM software stack is available through three development interfaces, each suited for specific use cases as outlined in the sections below. All three interfaces are built on top of native OnnxRuntime GenAI (OGA) libraries or llama.cpp libraries, as shown in the *Ryzen AI Software Stack* diagram below.

The **high-level Python APIs**, as well as the **Server Interface**, also leverage the **Lemonade SDK**, which is multi-vendor open-source software that provides everything necessary for quickly getting started with LLMs on OGA or llama.cpp.

At the bottom, **OnnxRuntime GenAI (OGA)** or llama.cpp (only supported for iGPU) API is the lowest-level API available for building LLM applications on a Ryzen AI PC.

Table 1: Ryzen AI Software Stack

Your Python Application	Your LLM Stack	Your Native Application
*Lemonade Python API**	*Lemonade Server Interface**	OGA C++ Headers **OR** llama.cpp C++ Headers
Custom AMD OnnxRuntime GenAI (OGA) **OR** llama.cpp*		
AMD Ryzen AI Driver and Hardware		

For more details, refer to the *LLM Deployment Overview* page.

3.1 Release Notes

3.1.1 Supported Configurations

Ryzen AI 1.7 Software supports AMD processors codenamed Phoenix, Hawk Point, Strix, Strix Halo, and Krackan Point. These processors can be found in the following Ryzen series:

- Ryzen 200 Series
- Ryzen 7000 Series, Ryzen PRO 7000 Series
- Ryzen 8000 Series, Ryzen PRO 8000 Series
- Ryzen AI 300 Series, Ryzen AI PRO Series, Ryzen AI Max 300 Series

For a complete list of supported devices, refer to the processor specifications page (look for the "AMD Ryzen AI" column towards the right side of the table, and select "Available" from the pull-down menu).

The rest of this document will refer to Phoenix as PHX, Hawk Point as HPT, Strix and Strix Halo as STX, and Krackan Point as KRK.

3.1.2 Model Compatibility Table

The following table lists which types of models are supported on what hardware platforms.

Model Type	PHX/HPT	STX/KRK
CNN INT8		
CNN BF16		
NLP BF16		
LLM (OGA)		

3.1.3 Version 1.7.1

- LLM
  - Performance improvements for Gemma-3-4B and GPT-OSS-20B
  - NPU support for SmolLM2-135M-Instruct and SmolLM-135M-Instruct
  - Long context support up to 16K for NPU models
- Stable Diffusion
  - Support for 10 model classes (t2i, i2i, dynamic shapes, ControlNet)
  - Performance and memory improvements for SD-2.1-v and SD-3.x

3.1.4 Version 1.7

- CNN/Transformer
  - General Performance improvements (Avg 15-20% improvements expected)
  - ~40% Compile time improvement for transformer models
  - Whisper.cpp integration enabling scalable ASR support
- LLM
  - New LLM Support: Qwen-2.5-14b-Instruct, Qwen-3-14b-Instruct, Phi-4-mini-instruct
  - First preview release of Sparse-LLM: GPT-OSS-20b NPU model
  - First preview release of VLM: Gemma-3-4b-it NPU model (functional)
  - Long context support for hybrid execution models
- Stable Diffusion
  - New model support: SD3.5-Turbo with 8x dynamic resolutions and 2x dynamic batches (Text2Image and Image2ImageControlNet)
  - New model support: Segmind-Vega 1024x1024 (Text2Image)
  - Up to 40% performance improvement for all supported models (Native-BFP16 format)
- Breaking Changes
  - The `log_level` provider option is deprecated. Use the ONNX Runtime session option `log_severity_level` instead to control Vitis AI execution logs.

3.1.5 Version 1.6

- BF16 Compiler (CNN, Transformer, ASR)
  - BF16 CNN perf improvements average 80% across release
  - BF16 perf improvements - 1.3X faster on CNN than iGPU and 2.6X faster for transformers than iGPU
  - Improved coverage and improved performance for ASR models
  - Average 3x compile time improvement
  - Smaller installation size
  - Reduction in CPU overhead by pushing data layout transformation to NPU
  - Dynamic batch size support for compilation
- New Integer Compiler (CNN)
  - Support for General Asymmetric Quantization enabling third party quantized models to run on NPU

  - Support for XINT8, A8W8, A16W8

- LLM
  - Broad Set of NPU only models optimized performance
  - New set of hybrid models with bfp16 activation
  - New architecture support in hybrid flow (Phi-4, Qwen-3)
  - Context length improvement from 2K to 4K for all models.
- Stable Diffusion Demo
  - 8x Dynamic Resolution for SD3.0/3.5 (text2image and image2imageControlNet)
  - Performance boost for SD 1.5/2.1-base/turbo/XL-turbo
  - Support Batch Size 1 for SD-turbo/SDXL-turbo
  - New model support (SD2.1-v 768x768 text2image, SDXL-base 1024x1024 text2image)
- Breaking Changes
  - For running INT8 models on STX/KRK or newer devices, the `xclbin` provider option is no longer supported and should no longer be used. See *Using INT8 Models* for full details.
  - For running INT8 models on PHX/HPT devices, the `target` option should be set to `X1`. The NPU binary should still be specified using the `xclbin` provider option. See *Using INT8 Models* for full details.
  - For BF16 models, the default configurations file requires a new `target` section. See *Config File Options* for full details.
  - LLM:
    * OGA version has been updated to v0.9.2 (Ryzen AI 1.6) from v0.7.0 (Ryzen AI 1.5). Any APIs that are obsolete must be updated to the supported equivalents as described in the Microsoft ONNX Runtime GenAI v0.9.2 documentation
    * Hybrid models published with earlier releases are not compatible with Ryzen AI 1.6. Please use the hybrid models published with the 1.6 release.

3.1.6 Version 1.5

- EoU Improvement
  - Application concurrency: improves the resource distribution across applications
  - Model Compilation time: 2x – 8x faster
  - Installation Size: 80% smaller
- Stable Diffusion demo pipelines (preview)

- 4K context length supported (on selected models)
- LLM Context cache support (on selected models)
- Bug fixes
- New LLMs released
  - Qwen/Qwen2.5-1.5B-Instruct
  - Qwen/Qwen2.5-3B-Instruct
  - Qwen/Qwen2.5-7B-Instruct
- Breaking Changes
  - The `%RYZEN_AI_INSTALLATION_PATH%\deployment` folder has been reorganized and flattened. Deployment DLLs are no longer organized in subfolders. If you use application build scripts that pull DLLs from the `deployment` folder, you need to update them based on the new paths. Refer to the *Application Packaging Requirements* section for further details.
  - The `1x4.xclbin` (PHX/HPT) and `AMD_AIE2P_Nx4_Overlay.xclbin` (STX/KRK) NPU binaries are no longer supported and should not be used. You should use the `4x4.xclbin` (PHX/HPT) and `AMD_AIE2P_4x4_Overlay.xclbin` (STX/KRK) NPU binaries instead.
  - The `XLNX_ENABLE_CACHE`, `XLNX_VART_FIRMWARE`, and `XLNX_TARGET_NAME` environment variables are no longer supported and should not be relied upon.
  - Support for VitisAI EP cache encryption is no longer available. To encrypt the compiled models, use the ONNX Runtime *EP Context Cache* feature instead.
  - For INT8 models, the VitisAI EP does not save the compiled model to disk by default. To save the compiled model, use the ONNX Runtime *EP Context Cache* feature or set the `enable_cache_file_io_in_mem` provider option to 0.
  - Generation of the `vitisai_ep_report.json` file is no longer automatic and should be manually enabled. See the *Operator Assignment Report* section for details.
  - Changes to the OGA flow for LLMs:
    - OGA Version is updated to **v0.7.0** (Ryzen AI 1.5) from v0.6.0 (Ryzen AI 1.4).
    - The `hybrid_llm` and `npu_llm` folders are consolidated into a new folder named `LLM`, which contains the `model_benchmark.exe` and `run_model.py` scripts, along with the necessary C++ headers and .lib files to support both the Hybrid LLM and NPU LLM workflows in C++ and Python.
    - For NPU LLM models, the `vaip_llm.json` file is no longer required. As a result, the `vaip_llm.json` path is removed from the `genai_config.json` for all NPU models. Ensure that you re-download the NPU models from Hugging Face when using the Ryzen AI 1.5 installer.

3.1.7 Version 1.4

- New Features:
  - New architecture support for Ryzen AI 300 series processors
  - Unified support for LLMs, INT8, and BF16 models in a single release package
  - Public release for compilation of BF16 CNN and NLP models on Windows
  - Public release of the LLM Hybrid OGA flow
  - LLM building flow for finetuned LLM
  - Support for up to 16 hardware contexts on Ryzen AI 300 series processors
  - Vitis AI EP now supports the ONNX Runtime EP context cache feature (for custom handling of pre-compiled models)
  - Ryzen AI environment variables converted to VitisAI EP session options
  - Improved exception handling and fallback to CPU
- New Hybrid execution mode LLMs:
  - DeepSeek-R1-Distill-Llama-8B
  - DeepSeek-R1-Distill-Qwen-1.5B
  - DeepSeek-R1-Distill-Qwen-7B
  - Gemma2-2B
  - Qwen2-1.5B
  - Qwen2-7B
  - AMD-OLMO-1B-SFT-DPO
  - Mistral-7B-Instruct-v0.1
  - Mistral-7B-Instruct-v0.2
  - Mistral-7B-v0.3
  - Llama3.1-8B-Instruct
  - Codellama-7B-Instruct
- *New BF16 model examples*:
  - Image classification
  - Finetuned DistilBERT for text classification
  - Text embedding model Alibaba-NLP/gte-large-en-v1.5
- New Tools:

  - Lemonade SDK
    * Lemonade Server: A server interface that uses the standard Open AI API, allowing applications in any language to integrate with Lemonade Server for local LLM deployment and compatibility with existing Open AI apps.
    * Lemonade Python API: Offers High-Level API for easy integration of Lemonade LLMs into Python applications and Low-Level API for custom experiments with specific checkpoints, devices, and tools.
    * Lemonade Command Line Interface easily benchmark, measure accuracy, prompt or gather memory usage of your LLM.
  - TurnkeyML – Open-source tool that includes low-code APIs for general ONNX workflows.
  - Digest AI – A Model Ingestion and Analysis Tool in collaboration with the Linux Foundation.
  - GAIA – An open-source application designed for the quick setup and execution of generative AI applications on local PC hardware.

- Quark-torch:
  - Added ROUGE and METEOR evaluation metrics for LLMs
  - Support for evaluating ONNX models exported using OGA
  - Support for offline evaluation (evaluation without generation) for LLMs
  - Support for Hugging Face integration
  - Support for Gemma2 quantization using the OGA flow
  - Support for Llama-3.2 quantization with FP8 (weights, activation, and KV-cache) for the vision and language components
- Quark-onnx:
  - Support compatibility with ONNX Runtime version 1.20.0, and 1.20.1
  - Support for microexponents (MX) data types, including MX4, MX6, and MX9
  - Support for BF16 data type for VAIML
  - Support for excluding pre and post-processing from quantization
  - Support for mixed precision with any data type
  - Support for Quarot rotation R1 algorithm
  - Support for microexponents and microscaling AdaQuant
  - Support for an auto-search algorithm to automatically find the best accuracy quantized model

  - Added tools for evaluating L2, PSNR, VMAF, and cosine

- ONNX Runtime EP:
  - Support for Chinese characters in the `filename/cache_dir/cache_key/xclbin`
  - Support for `int4/uint4` data type
  - Support for configurable failure handling: CPU fallback or exception
  - Update for encrypt/decrypt feature
- Known Issues:
  - Microsoft Windows Insider Program (WIP) users may see warnings or need to restart when running all applications concurrently.
    * NPU driver and workloads will continue to work.
  - Context creation may appear to be limited when some application do not close contexts quickly.

3.1.8 Version 1.3

- New Features:
  - Initial release of the Quark quantizer
  - Support for mixed precision data types
  - Compatibility with Copilot+ applications
- Improved support for *LLMs using OGA*
- New EoU Tools:
  - CNN profiling tool for VAI-ML flow
  - Idle detection and suspension of contexts
  - Rebalance feature for AIE hardware resource optimization
- NPU and Compiler:
  - New Op Support:
    * MAC
    * QResize Bilinear
    * LUT Q-Power
    * Expand
    * Q-Hsoftmax
    * A16 Q-Pad

        * Q-Reduce-Mean along H/W dimension
        * A16 Q-Global-AvgPool
        * A16 Padding with non-zero values
        * A16 Q-Sqrt
        * Support for XINT8/XINT16 MatMul and A16W16/A8W8 Q-MatMul
    - Performance Improvements:
        * Q-Conv, Q-Pool, Q-Add, Q-Mul, Q-InstanceNorm
        * Enhanced QDQ support for a range of operations
        * Enhanced the tiling algorithm
        * Improved graph-level optimization with extra transpose removal
        * Enhanced AT/MT fusion
        * Optimized memory usage and compile time
        * Improved compilation messages
- Quark for PyTorch:
    - Model Support:
        * Examples of LLM PTQ, such as Llama3.2 and Llama3.2-Vision models
        * Example of YOLO-NAS detection model PTQ/QAT
        * Example of SDXL v1.0 with weight INT8 activation INT8
    - PyTorch Quantizer Enhancements:
        * Partial model quantization by user configuration under FX mode
        * Quantization of ConvTranspose2d in Eager Mode and FX mode
        * Advanced Quantization Algorithms with auto-generated configurations
        * Optimized Configuration with DataTypeSpec for ease of use
        * Accelerated in-place replacement under Eager Mode
        * Loading configuration from file of algorithms and pre-optimizations
- Quark for ONNX:
    - New Features:
        * Compatibility with ONNX Runtime version 1.18, 1.19
        * Support for int4, uint4, Microscaling data types
        * Quantization for arbitrary specified operators

        * Quantization type alignment of element-wise operators for mixed precision
        * ONNX graph cleaning
        * Int32 bias quantization
    - ONNX Quantizer Enhancements:
        * Fast fine-tuning support for the MatMul operator, BFP data type, and GPU acceleration
        * Improved ONNX quantization of LLM models
        * Optimized quantization of FP16 models
        * Custom operator compilation process
        * Default parameters for auto mixed precision
        * Optimized Ryzen AI workflow by aligning with hardware constraints of the NPU
- ONNX Runtime EP:
    - Support for ONNX Runtime EP shared libraries
    - Python dependency removal
    - Memory optimization during the compile phase
    - Pattern API enhancement with multiple outputs and commutable arguments support
- Known Issues:
    - Extended compile time for some models with BF16/BFP16 data types
    - LLM models with 4K sequence length may revert to CPU execution
    - Accuracy drop in some Transformer models using BF16/BFP16 data types, requiring Quark intervention

3.1.9 Version 1.2

- New features:
    - Support added for Strix Point NPUs
    - Support added for integrated GPU
    - Smart installer for Ryzen AI 1.2
    - NPU DPM based on power slider
- New model support:
    - LLM flow support for multiple models in both PyTorch and ONNX flow (optimized model support will be released asynchronously)

  - SDXL-T with limited performance optimization

- New EoU tools:
  - AI Analyzer : Analysis and visualization of model compilation and inference profiling
  - Platform/NPU inspection and management tool (xrt-smi)
  - Onnx Benchmarking tool
- New Demos:
  - NPU-GPU multi-model pipeline application demo
- NPU and Compiler
  - New device support: Strix Nx4 and 4x4 Overlay
  - New Op support:
    * InstanceNorm
    * Silu
    * Floating scale quantization operators (INT8, INT16)
  - Support new rounding mode (Round to even)
  - Performance Improvement:
    * Reduced the model compilation time
    * Improved instruction loading
    * Improved synchronization in large overlay
    * Enhanced strided_slice performance
    * Enhanced convolution MT fusion
    * Enhanced convolution AT fusion
    * Enhanced data movement op performance
- ONNX Quantizer updates
  - Improved usability with various features and tools, including weights-only quantization, graph optimization, dynamic shape fixing, and format transformations.
  - Improved the accuracy of quantized models through automatic mixed precision and enhanced AdaRound and AdaQuant techniques.
  - Enhanced support for the BFP data type, including more attributes and shape inference capability.
  - Optimized the NPU workflow by aligning with the hardware constraints of the NPU.
  - Supported compilation for Windows and Linux.

  - Bugfix:
    * Fixed the problem where per-channel quantization is not compatible with onnxruntime 1.17.
    * Fixed the bug of CLE when conv with groups.
    * Fixed the bug of bias correction.
- Pytorch Quantizer updates
  - Tiny value quantization protection.
  - Higher onnx version support in quantized model exporting.
  - Relu6 hardware constrains support.
  - Support of mean operation with keepdim=True.
- Resolved issues:
  - NPU SW stack will fail to initialize when the system is out of memory. This could impact camera functionality when Microsoft Effect Pack is enabled.
  - If Microsoft Effects Pack is overloaded with other 4+ applications that use NPU to do inference, then camera functionality can be impacted. Can be fixed with a reboot. This will be fixed in the next release.

3.1.10 Version 1.1

- New model support:
  - Llama 2 7B with w4abf16 (3-bit and 4-bit) quantization (Beta)
  - Whisper base (EA access)
- New EoU tools:
  - CNN Benchmarking tool on RyzenAI-SW Repo
  - Platform/NPU inspection and management tool

Quantizer

- ONNX Quantizer:
  - Improved usability with various features and tools, including diverse parameter configurations, graph optimization, shape fixing, and format transformations.
  - Improved quantization accuracy through the implementation of experimental algorithmic improvements, including AdaRound and AdaQuant.
  - Optimized the NPU workflow by distinguishing between different targets and aligning with the hardware constraints of the NPU.

  - Introduced new utilities for model conversion.
- PyTorch Quantizer:
  - Mixed data type quantization enhancement and bug fix.
  - Corner bug fixes for add, sub, and conv1d operations.
  - Tool for converting the S8S8 model to the U8S8 model.
  - Tool for converting the customized Q/DQ to onnxruntime contributed Q/DQ with the "microsoft" domain.
  - Tool for fixing a dynamic shapes model to fixed shape model.
- Bug fixes
  - Fix for incorrect logging when simulating the LeakyRelu alpha value.
  - Fix for useless initializers not being cleaned up during optimization.
  - Fix for external data cannot be found when using use_external_data_format.
  - Fix for custom Ops cannot be registered due to GLIBC version mismatch

NPU and Compiler

- New op support:
  - Support Channel-wie Prelu.
  - Gstiling with reverse = false.
- Fixed issues:
  - Fixed Transpose-convolution and concat optimization issues.
  - Fixed Conv stride 3 corner case hang issue.
- Performance improvement:
  - Updated Conv 1x1 stride 2x2 optimization.
  - Enhanced Conv 7x7 performance.
  - Improved padding performance.
  - Enhanced convolution MT fusion.
  - Improved the performance for NCHW layout model.
  - Enhanced the performance for eltwise-like op.
  - Enhanced Conv and eltwise AT fusion.
  - Improved the output convolution/transpose-convolution's performance.
  - Enhanced the logging message for EoU.

ONNX Runtime EP

- End-2-End Application support on NPU
  - Enhanced existing support: Provided high-level APIs to enable seamless incorporation of pre/post-processing operations into the model to run on NPU
  - Two examples (resnet50 and yolov8) published to demonstrate the usage of these APIs to run end-to-end models on the NPU
- Bug fixes for ONNXRT EP to support customers' models

Misc

- Contains mitigation for the following CVEs: CVE-2024-21974, CVE-2024-21975, CVE-2024-21976

3.1.11 Version 1.0.1

- Minor fix for Single click installation without given env name.
- Perform improvement in the NPU driver.
- Bug fix in elementwise subtraction in the compiler.
- Runtime stability fixes for minor corner cases.
- Quantizer update to resolve performance drop with default settings.

3.1.12 Version 1.0

Quantizer

- ONNX Quantizer
  - Support for ONNXRuntime 1.16.
  - Support for the Cross-Layer-Equalization (CLE) algorithm in quantization, which can balance the weights of consecutive Conv nodes to make it more quantize-friendly in per-tensor quantization.
  - Support for mixed precision quantization including UINT16/INT16/UINT32/INT32/FLOAT16/BFLOAT16, and support asymmetric quantization for BFLOAT16.
  - Support for the MinMSE method for INT16/UINT16/INT32/UINT32 quantization.
  - Support for quantization using the INT16 scale.
  - Support for unsigned ReLU in symmetric activation configuration.
  - Support for converting Float16 to Float32 during quantization.

  - Support for converting NCHW model to NHWC model during quantization.
  - Support for two more modes for MinMSE for better accuracy. The "All" mode computes the scales with all batches while the "MostCommon" mode computes the scale for each batch and uses the most common scales.
  - Support for the quantization of more operations:
    * PReLU, Sub, Max, DepthToSpace, SpaceToDepth, Slice, InstanceNormalization, and LpNormalization.
    * Non-4D ReduceMean.
    * Leakyrelu with arbitrary alpha.
    * Split by converting it to Slice.
  - Support for op fusing of InstanceNormalization and L2Normalization in NPU workflow.
  - Support for converting Clip to ReLU when the minimal value is 0.
  - Updated shift_bias, shift_read, and shift_write constraints in the NPU workflow and added an option "IPULimitationCheck" to disable it.
  - Support for disabling the op fusing of Conv + LeakyReLU/PReLU in the NPU workflow.
  - Support for logging for quantization configurations and summary information.
  - Support for removing initializer from input to support models converted from old version pytorch where weights are stored as inputs.
  - Added a recommended configuration for the IPU_Transformer platform.
  - New utilities:
    * Tool for converting the float16 model to the float32 model.
    * Tool for converting the NCHW model to the NHWC model.
    * Tool for quantized models with random input.
  - Three examples for quantization models from Timm, Torchvision, and ONNXRuntime modelzoo respectively.
  - Bugfixes:
    * Fix a bug that weights are quantized with the "NonOverflow" method when using the "MinMSE" method.

- Pytorch Quantizer
  - Support of some operations quantization in quantizer: inplace div, inplace sub
  - Log and document enhancement to emphasize fast-finetune
  - Timm models quantization script example

  - Bug fix for operators: clamp and prelu
  - QAT Support quantization of operations with multiple outputs
  - QAT EOU enhancements: significantly reduces the need for network modifications
  - QAT ONNX exporting enhancements: support more configurations
  - New QAT examples
- TF2 Quantizer
  - Support for Tensorflow 2.11 and 2.12.
  - Support for the 'tf.linalg.matmul' operator.
  - Updated shift_bias constraints for NPU workflow.
  - Support for dumping models containing operations with multiple outputs.
  - Added an example of a sequential model.
  - Bugfixes:
    * Fix a bug that Hardsigmoid and Hardswish are not mapped to DPU without Batch Normalization.
    * Fix a bug when both align_pool and align_concat are used simultaneously.
    * Fix a bug in the sequential model when a layer has multiple consumers.
- TF1 Quantizer
  - Update shift_bias constraints for NPU workflow.
  - Bugfixes:
    * Fix a bug in fast_finetune when the 'input_node' and 'quant_node' are inconsistent.
    * Fix a bug that AddV2 op identified as BiasAdd.
    * Fix a bug when the data type of the concat op is not float.
    * Fix a bug in split_large_kernel_pool when the stride is not equal to 1.

ONNXRuntime Execution Provider

- Support new OPs, such as PRelu, ReduceSum, LpNormlization, DepthToSpace(DCR).
- Increase the percentage of model operators performed on the NPU.
- Fixed some issues causing model operators allocation to CPU.
- Improved report summary
- Support the encryption of the VOE cache
- End-2-End Application support on NPU

  - Enable running pre/post/custom ops on NPU, utilizing ONNX feature of E2E extensions.
  - Two examples published for yolov8 and resnet50, in which preprocessing custom op is added and runs on NPU.
- Performance: latency improves by up to 18% and power savings by up to 35% by additionally running preprocessing on NPU apart from inference.
- Multiple NPU overlays support
  - VOE configuration that supports both CNN-centric and GEMM-centric NPU overlays.
  - Increases number of ops that run on NPU, especially for models which have both GEMM and CNN ops.
  - Examples published for use with some of the vision transformer models.

NPU and Compiler

- New operators support
  - Global average pooling with large spatial dimensions
  - Single Activation (no fusion with conv2d, e.g. relu/single alpha PRelu)
- Operator support enhancement
  - Enlarge the width dimension support range for depthwise-conv2d
  - Support more generic broadcast for element-wise like operator
  - Support output channel not aligned with 4B GStiling
  - Support Mul and LeakyRelu fusion
  - Concatenation's redundant input elimination
  - Channel Augmentation for conv2d (3x3, stride=2)
- Performance optimization
  - PDI partition refine to reduce the overhead for PDI swap
  - Enabled cost model for some specific models
- Fixed asynchronous error in multiple thread scenario
- Fixed known issue on tanh and transpose-conv2d hang issue

Known Issues

- Support for multiple applications is limited to up to eight
- Windows Studio Effects should be disabled when using the Latency profile. To disable Windows Studio Effects, open **Settings > Bluetooth & devices > Camera**, select your primary camera, and then disable all camera effects.

3.1.13 Version 0.9

Quantizer

- Pytorch Quantizer
  - Dict input/output support for model forward function
  - Keywords argument support for model forward function
  - Matmul subroutine quantization support
  - Support of some operations in quantizer: softmax, div, exp, clamp
  - Support quantization of some non-standard conv2d.
- ONNX Quantizer
  - Add support for Float16 and BFloat16 quantization.
  - Add C++ kernels for customized QuantizeLinear and DequantizeLinaer operations.
  - Support saving quantizer version info to the quantized models' producer field.
  - Support conversion of ReduceMean to AvgPool in NPU workflow.
  - Support conversion of BatchNorm to Conv in NPU workflow.
  - Support optimization of large kernel GlobalAvgPool and AvgPool operations in NPU workflow.
  - Supports hardware constraints check and adjustment of Gemm, Add, and Mul operations in NPU workflow.
  - Supports quantization for LayerNormalization, HardSigmoid, Erf, Div, and Tanh for NPU.

ONNXRuntime Execution Provider

- Support new OPs, such as Conv1d, LayerNorm, Clip, Abs, Unsqueeze, ConvTranspose.
- Support pad and depad based on NPU subgraph's inputs and outputs.
- Support for U8S8 models quantized by ONNX quantizer.
- Improve report summary tools.

NPU and Compiler

- Supported exp/tanh/channel-shuffle/pixel-unshuffle/space2depth
- Performance uplift of xint8 output softmax
- Improve the partition messages for CPU/DPU
- Improve the validation check for some operators
- Accelerate the speed of compiling large models
- Fix the elew/pool/dwc/reshape mismatch issue and fix the stride_slice hang issue
- Fix str_w != str_h issue in Conv

LLM

- Smoothquant for OPT1.3b, 2.7b, 6.7b, 13b models.
- Huggingface Optimum ORT Quantizer for ONNX and Pytorch dynamic quantizer for Pytorch
- Enabled Flash attention v2 for larger prompts as a custom torch.nn.Module
- Enabled all CPU ops in bfloat16 or float32 with Pytorch
- int32 accumulator in AIE (previously int16)
- DynamicQuantLinear op support in ONNX
- Support different compute primitives for prefill/prompt and token phases
- Zero copy of weights shared between different op primitives
- Model saving after quantization and loading at runtime for both Pytorch and ONNX
- Enabled profiling prefill/prompt and token time using local copy of OPT Model with additional timer instrumentation
- Added demo mode script with greedy, stochastic and contrastive search options

ASR

- Support Whipser-tiny
- All GEMMs offloaded to AIE
- Improved compile time
- Improved WER

Known issues

- Flow control OPs including “Loop”, “If”, “Reduce” not supported by VOE
- Resizing OP in ONNX opset 10 or lower is not supported by VOE
- Tensorflow 2.x quantizer supports models within tf.keras.model only
- Running quantizer docker in WSL on Ryzen AI laptops may encounter OOM (Out-of-memory) issue
- Running multiple concurrent models using temporal sharing on the 5x4 binary is not supported
- Only batch sizes of 1 are supported
- Only models with the pretrained weights setting = TRUE should be imported
- Launching multiple processes on 4 1x4 binaries can cause hangs, especially when models have many sub-graphs

3.1.14 Version 0.8

Quantizer

- Pytorch Quantizer
  - Pytorch 1.13 and 2.0 support
  - Mixed precision quantization support, supporting float32/float16/bfloat16/intx mixed quantization
  - Support of bit-wise accuracy cross check between quantizer and ONNX-runtime
  - Split and chunk operators were automatically converted to slicing
  - Add support for BFP data type quantization
  - Support of some operations in quantizer: where, less, less_equal, greater, greater_equal, not, and, or, eq, maximum, minimum, sqrt, Elu, Reduction_min, argmin
  - QAT supports training on multiple GPUs
  - QAT supports operations with multiple inputs or outputs
- ONNX Quantizer
  - Provided Python wheel file for installation

  - Support OnnxRuntime 1.15
  - Supports setting input shapes of random data reader
  - Supports random data reader in the dump model function
  - Supports saving the S8S8 model in U8S8 format for NPU
  - Supports simulation of Sigmoid, Swish, Softmax, AvgPool, GlobalAvgPool, ReduceMean and LeakyRelu for NPU
  - Supports node fusions for NPU

ONNXRuntime Execution Provider

- Supports for U8S8 quantized ONNX models
- Improve the function of falling back to CPU EP
- Improve AIE plugin framework
  - Supports LLM Demo
  - Supports Gemm ASR
  - Supports E2E AIE acceleration for Pre/Post ops
  - Improve the easy-of-use for partition and deployment
- Supports models containing subgraphs
- Supports report summary about OP assignment
- Supports report summary about DPU subgraphs falling back to CPU
- Improve log printing and troubleshooting tools.
- Upstreamed to ONNX Runtime Github repo for any data type support and bug fix

NPU and Compiler

- Extended the support range of some operators
  - Larger input size: conv2d, dwc
  - Padding mode: pad
  - Broadcast: add
  - Variant dimension (non-NHWC shape): reshape, transpose, add
- Support new operators, e.g. reducemax(min/sum/avg), argmax(min)
- Enhanced multi-level fusion
- Performance enhancement for some operators

- Add quantization information validation
- Improvement in device partition
  - User friendly message
  - Target-dependency check

Demos

- New Demos link: https://account.amd.com/en/forms/downloads/ryzen-ai-software-platform-xef.html?filename=transformers_2308.zip
  - LLM demo with OPT-1.3B/2.7B/6.7B
  - Automatic speech recognition demo with Whisper-tiny

Known issues

- Flow control OPs including “Loop”, “If”, “Reduce” not supported by VOE
- Resize OP in ONNX opset 10 or lower not supported by VOE
- Tensorflow 2.x quantizer supports models within tf.keras.model only
- Running quantizer docker in WSL on Ryzen AI laptops may encounter OOM (Out-of-memory) issue
- Run multiple concurrent models by temporal sharing on the Performance optimized overlay (5x4.xclbin) is not supported
- Support batch size 1 only for NPU

3.1.15 Version 0.7

Quantizer

- Docker Containers
  - Provided CPU dockers for Pytorch, Tensorflow 1.x, and Tensorflow 2.x quantizer
  - Provided GPU Docker files to build GPU dockers
- Pytorch Quantizer
  - Supports multiple output conversion to slicing
  - Enhanced transpose OP optimization

  - Inspector support new IP targets for NPU
- ONNX Quantizer
  - Provided Python wheel file for installation
  - Supports quantizing ONNX models for NPU as a plugin for the ONNX Runtime native quantizer
  - Supports power-of-two quantization with both QDQ and QOP format
  - Supports Non-overflow and Min-MSE quantization methods
  - Supports various quantization configurations in power-of-two quantization in both QDQ and QOP format.
    * Supports signed and unsigned configurations.
    * Supports symmetry and asymmetry configurations.
    * Supports per-tensor and per-channel configurations.
  - Supports bias quantization using int8 datatype for NPU.
  - Supports quantization parameters (scale) refinement for NPU.
  - Supports excluding certain operations from quantization for NPU.
  - Supports ONNX models larger than 2GB.
  - Supports using CUDAExecutionProvider for calibration in quantization
  - Open source and upstreamed to Microsoft Olive Github repo
- TensorFlow 2.x Quantizer
  - Added support for exporting the quantized model ONNX format.
  - Added support for the keras.layers.Activation('leaky_relu')
- TensorFlow 1.x Quantizer
  - Added support for folding Reshape and ResizeNearestNeighbor operators.
  - Added support for splitting Avgpool and Maxpool with large kernel sizes into smaller kernel sizes.
  - Added support for quantizing Sum, StridedSlice, and Maximum operators.
  - Added support for setting the input shape of the model, which is useful in deploying models with undefined input shapes.
  - Add support for setting the opset version in exporting ONNX format

ONNX Runtime Execution Provider

- Vitis ONNX Runtime Execution Provider (VOE)
  - Supports ONNX Opset version 18, ONNX Runtime 1.16.0, and ONNX version 1.13
  - Supports both C++ and Python APIs(Python version 3)
  - Supports deploy model with other EPs
  - Supports falling back to CPU EP
  - Open source and upstreamed to ONNX Runtime Github repo
  - Compiler
    - Multiple Level op fusion
    - Supports the same muti-output operator like chunk split
    - Supports split big pooling to small pooling
    - Supports 2-channel writeback feature for Hard-Sigmoid and Depthwise-Convolution
    - Supports 1-channel GStiling
    - Explicit pad-fix in CPU subgraph for 4-byte alignment
    - Tuning the performance for multiple models

NPU

- Two configurations
  - Power Optimized Overlay
    - Suitable for smaller AI models (1x4.xclbin)
    - Supports spatial sharing, up to 4 concurrent AI workloads
  - Performance Optimized Overlay (5x4.xclbin)
    - Suitable for larger AI models

Known issues

- Flow control OPs including “Loop”, “If”, “Reduce” are not supported by VOE
- Resize OP in ONNX opset 10 or lower not supported by VOE
- Tensorflow 2.x quantizer supports models within tf.keras.model only
- Running quantizer docker in WSL on Ryzen AI laptops may encounter OOM (Out-of-memory) issue

- Run multiple concurrent models by temporal sharing on the Performance optimized overlay (5x4.xclbin) is not supported

3.2 Installation Instructions

This page covers Ryzen AI installation on Windows.

3.2.1 Prerequisites

The Ryzen AI Software supports AMD processors with a Neural Processing Unit (NPU). Refer to the release notes for the full list of *supported configurations*.

The following dependencies must be installed on the system before installing the Ryzen AI Software:

Dependencies	Version Requirement
Windows 11	build >= 22621.3527
Visual Studio	2022
cmake	version >= 3.26
Python distribution (Miniforge preferred)	Latest version

IMPORTANT:

- Visual Studio 2022 Community (Optional for AMD Quark, to support custom op flow): ensure that *Desktop Development with C++* is installed
- Miniforge: ensure that the following path is set in the System PATH variable: `path\to\miniforge3\condabin` or `path\to\miniforge3\Scripts\` or `path\to\miniforge3\` (The System PATH variable should be set in the *System Variables* section of the *Environment Variables* window).

3.2.2 Install NPU Drivers

- Download and Install the NPU driver version: 32.0.203.280 or newer using the following links:
  - `NPU Driver (Version 32.0.203.280)`

    * NPU driver 32.0.203.280 is production driver for Phoenix, Hawk Point, Strix, Strix Halo, and Krackan Point.
  - `NPU Driver (Version 32.0.203.314)`
- Install the NPU drivers by following these steps:
  - Extract the downloaded ZIP file.
  - Open a terminal in administrator mode and execute the `.\npu_sw_installer.exe` file.
- Ensure that NPU driver (Version:32.0.203.280, Date:5/16/2025) is correctly installed by opening Task Manager -> Performance -> NPU0.

3.2.3 Install Ryzen AI Software

- Download the Ryzen AI Software installer `ryzen-ai-lt-1.7.1.exe`.
- Launch the EXE installer and follow the instructions on the installation wizard:
  - Accept the terms of the Licence agreement
  - Provide the destination folder for Ryzen AI installation (default: `C:\Program Files\RyzenAI\1.7.1`)
  - Specify the name for the conda environment (default: `ryzen-ai-1.7.1`)

The Ryzen AI Software packages are now installed in the conda environment created by the installer.

> **Note**
>
> NuGet package is available to download at `ryzen-ai-1.7.1-nuget.zip`.

3.2.4 Test the Installation

The Ryzen AI Software installation folder contains test to verify that the software is correctly installed. This installation test can be found in the `quicktest` subfolder which is expected to work for Strix (STX) or newer devices.

- Open a Conda command prompt (search for "Miniforge Prompt" in the Windows start menu)
- Activate the Conda environment created by the Ryzen AI installer:

```
conda activate ryzen-ai-<version>
```

- Run the test:

```
cd %RYZEN_AI_INSTALLATION_PATH%/quicktest
python quicktest.py
```

```
[I:onnxruntime:, session_state_utils.cc:243 onnxruntime::session_state_
↪utils::SaveInitializedTensors] Saving initialized tensors.
[I:onnxruntime:, session_state_utils.cc:438 onnxruntime::session_state_
↪utils::SaveInitializedTensors] Done saving initialized tensors
[I:onnxruntime:, inference_session.cc:2532␣
↪onnxruntime::InferenceSession::Initialize] Session successfully␣
↪initialized.
Test Finished
```

- Verify NPU activity by opening **Task Manager** → **Performance** → **NPU** while the test is running. You should see NPU utilization increase during model inference.

NPU Offloading with Session Options

This section demonstrates how to enable NPU offloading logs using ONNX Runtime session options. The code also includes changes needed in `quicktest.py` to run on Phoenix/Hawk Point devices. To view detailed logging information, update the session options in `quicktest.py` as shown below:

```
import os
import sys
import subprocess
import numpy as np
import onnxruntime as ort

def get_npu_info():
    # Run pnputil as a subprocess to enumerate PCI devices
    command = r'pnputil /enum-devices /bus PCI /deviceids '
    process = subprocess.Popen(command, shell=True, stdout=subprocess.
↪PIPE, stderr=subprocess.PIPE)
    stdout, stderr = process.communicate()
    # Check for supported Hardware IDs
    npu_type = ''
    if 'PCI\\VEN_1022&DEV_1502&REV_00' in stdout.decode(): npu_type =
↪'PHX/HPT'
    if 'PCI\\VEN_1022&DEV_17F0&REV_00' in stdout.decode(): npu_type = 'STX
↪'
    if 'PCI\\VEN_1022&DEV_17F0&REV_10' in stdout.decode(): npu_type = 'STX
↪'
    if 'PCI\\VEN_1022&DEV_17F0&REV_11' in stdout.decode(): npu_type = 'STX
↪'
    if 'PCI\\VEN_1022&DEV_17F0&REV_20' in stdout.decode(): npu_type = 'KRK
↪'
    return npu_type
```

(continues on next page)

(continued from previous page)

```
# Get APU type info: PHX/STX/HPT
npu_type = get_npu_info()
install_dir = os.environ['RYZEN_AI_INSTALLATION_PATH']
model       = os.path.join(install_dir, 'quicktest', 'test_model.onnx')
providers   = ['VitisAIExecutionProvider']
provider_options = [{}]  # Default provider options for STX/KRK and newer␣
↪devices

if npu_type == 'PHX/HPT':
    print("Setting environment for PHX/HPT")
    xclbin_file = os.path.join(install_dir, 'voe-4.0-win_amd64', 'xclbins
↪', 'phoenix', '4x4.xclbin')
    provider_options = [{
         'target': 'X1',
         'xlnx_enable_py3_round': 0,
         'xclbin': xclbin_file,
     }]

# Create session options
session_options = ort.SessionOptions()
session_options.log_severity_level = 0  # 0=Verbose, 1=Info, 2=Warning,␣
↪3=Error, 4=Fatal

try:
    session = ort.InferenceSession(model,
                               sess_options=session_options,
                               providers=providers,
                               provider_options=provider_options)
except Exception as e:
    print(f"Failed to create an InferenceSession: {e}")
    sys.exit(1)  # Exit the program with a non-zero status to indicate an␣
↪error

def preprocess_random_image():
    image_array = np.random.rand(3, 32, 32).astype(np.float32)
    return np.expand_dims(image_array, axis=0)

# inference on random image data
input_data = preprocess_random_image()
try:
    outputs = session.run(None, {'input': input_data})
except Exception as e:
```

(continues on next page)

(continued from previous page)

```
    print(f"Failed to run the InferenceSession: {e}")
    sys.exit(1)  # Exit the program with a non-zero status to indicate an␣
↪error
else:
   print("Test finished")
```

- Run the test:

```
cd %RYZEN_AI_INSTALLATION_PATH%/quicktest
python quicktest.py 2>&1 | findstr /i "VerifyEachNodeIsAssignedToAnEp |␣
↪Test"
```

- On a successful run, you will see an output similar to the one shown below. This indicates that the model is running on the NPU and that the installation of the Ryzen AI Software was successful:

```
[V:onnxruntime:, session_state.cc:1296␣
↪onnxruntime::VerifyEachNodeIsAssignedToAnEp] Node placements
[V:onnxruntime:, session_state.cc:1299␣
↪onnxruntime::VerifyEachNodeIsAssignedToAnEp]  All nodes placed on␣
↪[VitisAIExecutionProvider]. Number of nodes: 3
Test Finished
```

Note

- The full path to the Ryzen AI Software installation folder is stored in the `RYZEN_AI_INSTALLATION_PATH` environment variable.
- For Phoenix/Hawk Point hardware, set the `target` to `X1` in the provider options.

3.3 Examples, Demos, Tutorials

This page introduces various demos, examples, and tutorials currently available with the Ryzen™ AI Software.

3.3.1 Getting Started Tutorials

NPU

- Getting Started Tutorial for INT8 models - Uses a custom ResNet model to demonstrate:
  - Pretrained model conversion to ONNX
  - Model Quantization using AMD Quark quantizer

  - Deployment using ONNX Runtime C++ and Python code
- Getting Started Tutorial for BF16 models - Uses a custom ResNet model to demonstrate:
  - Preparation and compilation of BF16 models
  - Deployment using Python
  - Deployment using C++
- Hello World Jupyter Notebook Tutorial
- Additional BF16 model examples:
  - Image Classification
  - Finetuned DistilBERT for Text Classification
- Super-Resolution Models on AMD Ryzen AI NPU
- LLM examples
  - LLMs on RyzenAI with ONNX Runtime GenAI API
  - ONNX Runtime GenAI(OGA)-based RAG LLM
  - Running Vision Language Model (VLM) on RyzenAI NPU
  - Running GPT-OSS-20B with chat template

iGPU

- ResNet50 on iGPU

3.3.2 Other examples, demos, and tutorials

- Refer to RyzenAI-SW repo

3.4 Model Quantization

Model quantization is the process of mapping high-precision weights/activations to a lower precision format, such as BF16/INT8, while maintaining model accuracy. This technique enhances the computational and memory efficiency of the model for deployment on NPU devices. It can be applied post-training, allowing existing models to be optimized without the need for retraining.

The Ryzen AI compiler supports input models in the following formats:

- CNN Models
  - INT8 (quantized)
  - FP32 (automatically converted to BF16 during compilation)
- Transformer Models:

  - FP32 (automatically converted to BF16 during compilation)

Ryzen AI Software natively supports both CNN and Transformer models in floating-point (FP32) format. When FP32 models are provided as input, the VitisAI EP automatically converts them to bfloat16 (BF16) precision and processes them through the optimized BF16 compilation pipeline.

For CNN models, AMD Quark quantization enables conversion to INT8 format, delivering improved inference performance compared to higher precision alternatives. This quantization pathway provides an additional optimization option for CNN workloads requiring maximum efficiency.

The complete list of operations supported for different quantization types can be found in *Supported Operations*.

3.4.1 FP32 to BF16 Conversion

Ryzen AI provides seamless support for deploying original floating-point (FP32) models on NPU hardware through automatic conversion to BFLOAT16 (BF16) format. The conversion from FP32 to BF16 is performed when the model is compiled by the VitisAI EP. BF16 is a 16-bit floating-point format designed to have the same exponent size as FP32, allowing a wide dynamic range, but with reduced precision to save memory and speed up computations. This feature enables developers to deploy models in their native format while leveraging the Ryzen AI automatic conversion for efficient execution on NPU.

FP32 to BF16 Examples

Explore these practical examples demonstrating FP32 to BF16 conversion across different CNN, NLP model types:

- Image Classification using ResNet50 model on NPU
- Finetuned DistilBERT for Text Classification
- Advanced quantization techniques Fast Finetuning for BF16 models.

3.4.2 FP32 to INT8 Conversion

Quantization to INT8 format introduces several challenges, primarily revolving around the potential drop in model accuracy. Choosing the right quantization parameters—such as data type, bit-width, scaling factors, and the decision between per-channel or per-tensor quantization—adds layers of complexity to the design process. These decisions significantly impact both model accuracy and performance. While **AMD Quark** is the recommended quantization tool, third-party tools that support QDQ (Quantize-Dequantize) operations can also be used for model quantization.

RyzenAI supports the following INT8 datatypes:

- XINT8: uses symmetric INT8 activation and weights quantization with power-of-two scales
- A8W8: uses symmetric INT8 activation and weights quantization with float scales

- A16W8: uses symmetric INT16 activation and symmetric INT8 weights quantization with float scales

AMD Quark is the recommended quantization tool to convert FP32 models to INT8. But third-party tools that support QDQ (Quantize-Dequantize) operations can also be used for model quantization to A8W8 and A16W8.

AMD Quark

AMD Quark is a comprehensive cross-platform deep learning toolkit designed to simplify and enhance the quantization of deep learning models. Supporting both PyTorch and ONNX models, Quark empowers developers to optimize their models for deployment on a wide range of hardware backends, achieving significant performance gains without compromising accuracy.

AMD Quark provides default configurations that support INT8 quantization. For example, *XINT8* uses symmetric INT8 activation and weights quantization with power-of-two scales using the MinMSE calibration method.

For more challenging model quantization needs, **AMD Quark** supports different quantization configurations such as *A8W8*, *A16W8*, and advanced quantization techniques. For more details, refer to AMD Quark for Ryzen AI

The quantization configuration can be customized using the *QuantizationConfig* class. The following example shows how to set up the quantization configuration for INT8 quantization:

```
quant_config = QuantizationConfig(calibrate_method=PowerOfTwoMethod.
↪MinMSE,
                                  activation_type=QuantType.QUInt8,
                                  weight_type=QuantType.QInt8,
                                  enable_npu_cnn=True,
                                  extra_options={'ActivationSymmetric':␣
↪True})
config = Config(global_quant_config=quant_config)
print("The configuration of the quantization is {}".format(config))
```

The user can use the *get_default_config('XINT8')* function to get the default configuration for INT8 quantization.

FP32 to INT8 Examples

Explore practical INT8 quantization examples:

- Running INT8 model on NPU using Getting Started Tutorial
- AMD Quark Tutorial for Ryzen AI Deployment
- Advanced quantization techniques Fast Finetuning and Cross Layer Equalization for INT8 model

3.5 Model Compilation and Deployment

3.5.1 Introduction

The Ryzen AI Software supports models saved in the ONNX format and uses ONNX Runtime as the primary mechanism to load, compile and run models.

NOTE: Models with ONNX opset 17 are recommended. If your model uses a different opset version, consider converting it using the ONNX Version Converter

For a complete list of supported operators, consult this page: *Supported Operators*.

Loading Models

Models are loaded by creating an ONNX Runtime `InferenceSession` using the Vitis AI Execution Provider (VAI EP):

```
import onnxruntime

session_options = onnxruntime.SessionOptions()
vai_ep_options  = {}                          # Vitis AI EP options go␣
↪here

session = onnxruntime.InferenceSession(
    path_or_bytes = model,                    # Path to the ONNX model
    sess_options = session_options,           # Standard ORT options
    providers = ['VitisAIExecutionProvider'], # Use the Vitis AI␣
↪Execution Provider
    provider_options = [vai_ep_options]       # Pass options to the Vitis␣
↪AI Execution Provider
)
```

The `provider_options` parameter enables the configuration of the Vitis AI Execution Provider (EP). For a comprehensive list of supported provider options, refer to the *Vitis AI EP Options Reference Guide* below.

When a model is first loaded into an ONNX Runtime (ORT) inference session, it is compiled into the format required by the NPU. The resulting compiled output can be saved as an ORT EP context file or stored in the Vitis AI EP cache directory.

If a compiled version of the ONNX model is already available — either as an EP context file or within the Vitis AI EP cache — the model will not be recompiled. Instead, the precompiled version will be loaded automatically. This greatly reduces session creation time and improves overall efficiency. For more details, refer to the section on *Managing Compiled Models*.

Deploying Models

Once the ONNX Runtime inference session is initialized and the model is compiled, the model is deployed using the ONNX Runtime `run()` API:

```
input_data = {}
for input in session.get_inputs():
    input_data[input.name] = ...  # Initialize input tensors

outputs = session.run(None, input_data) # Run the model
```

The ONNX graph is automatically partitioned into multiple subgraphs by the Vitis AI Execution Provider (EP). During deployment, the subgraph(s) containing operators supported by the NPU are executed on the NPU. The remaining subgraph(s) are executed on the CPU. This graph partitioning and deployment technique across CPU and NPU is fully automated by the VAI EP and is totally transparent to the end-user.

3.5.2 Vitis AI EP Options Reference Guide

VitisAI EP Provider Options

The `provider_options` parameter of the ORT `InferenceSession` allows passing options to configure the Vitis AI EP. The following options are supported:

Table 2: Vitis AI EP Provider Options

Option	Description	Values / Type	Default
config_fi	Config file for BF16 compilation options. See *Config File Options*.	String	N/A
target	Set which Vitis AI EP backend to use for compiling/running integer model. For details see *Using INT8 Models*.	X2, X1	X2
xclbin	To be used only when running INT8 CNN models on PHX/HPT devices. For details see *Using INT8 Models*	String	None
encryptio	256-bit key for encrypting EP context model. See *EP Context Cache*.	String (64 hex)	None
opt_level	Compiler optimization level for INT8 only.	0, 1, 2, 3, 65536 (maximum effort, experimental)	0
cache_dir	VitisAI cache directory. Set enable_cache_file_io_in_mem to 0.	String	C:\temp\%USERNAME%\vaip\.cache
cache_key	Subfolder in cache for compiled model. Set enable_cache_file_io_in_mem to 0.	String	MD5 hash of model
enable_ca	Keep compiled model in memory (1) or save to disk (0). • 0: saves to disk (BF16 and INT8 models) • 1: saves in memory (INT8 models only)	0, 1	1
ai_analyz	Enable compile-time analysis data.	Boolean	False
ai_analyz	Enable inference-time analysis data.	Boolean	False

Enabling ONNX Runtime Logs

To enable detailed logging for debugging purposes, set the ONNX Runtime session log severity level using `SessionOptions.log_severity_level`:

```
import onnxruntime

session_options = onnxruntime.SessionOptions()
session_options.log_severity_level = 1  # Set log level (see table below)

vai_ep_options = {}
```

(continues on next page)

(continued from previous page)

```
session = onnxruntime.InferenceSession(
    path_or_bytes = model,
    sess_options = session_options,
    providers = ['VitisAIExecutionProvider'],
    provider_options = [vai_ep_options]
)
```

Table 3: ORT Log Severity Levels

Level	Description	Value
Verbose	All messages (most detailed)	0
Info	Informational messages and above	1
Warning (Default)	Warnings and above	2
Error	Errors and above	3
Fatal	Fatal errors only	4

NOTE: The `log_level` parameter in provider options has been deprecated. To enable logging, use `SessionOptions.log_severity_level` as shown in the example above.

Config File Options

When compiling BF16 models, a JSON configuration file can be provided to the VitisAI EP using the `config_file` provider option. This configuration file is used to specify additional options to the compiler.

The default the configuration file for compiling BF16 models contains the following:

```
{
 "passes": [
     {
         "name": "init",
         "plugin": "vaip-pass_init"
     },
     {
         "name": "vaiml_partition",
         "plugin": "vaip-pass_vaiml_partition",
         "vaiml_config": {
             "optimize_level": 1,
             "preferred_data_storage": "auto"
         }
     }
 ],
```

(continues on next page)

(continued from previous page)

```
 "target": "VAIML",
    "targets": [
        {
            "name": "VAIML",
            "pass": [
                "init",
                "vaiml_partition"
            ]
        }
    ]
}
```

The `vaiml_config` section of the configuration file contains the user options. The supported user options are described below.

Table 4: Config File Options (vaiml_config)

Option	Description	Values	Default
`optimize_leve`	Compiler optimization level.	1, 2, 3	1
`preferred_dat`	Data layout: "auto" (let compiler choose), "vectorized" (for CNNs), "unvectorized" (for Transformers).	auto, vectorized, unvectorized	auto

3.5.3 Using BF16 Models

When compiling BF16 models, an optional configuration file can be provided to the VitisAI EP. This file is specified using the `config_file` provider option. For more details, refer to *Config File Options* section.

NOTE:

- Running BF16 Models is only supported for STX/KRK or newer devices. For the model compatibility table see *Release Notes*.
- For C++ applications that need to compile BF16 models at runtime, include `${CONDA_PREFIX}/Lib/site-packages/flexml/flexml_extras/lib/vaiml.dll` along with the DLLs specified in Application Development. However, using pre-compiled BF16 models is recommended for C++ deployment.

Sample Python Code

Python example loading a configuration file called vai_ep_config.json:

```
import onnxruntime

vai_ep_options = {
    'config_file': 'vai_ep_config.json',
}

session = onnxruntime.InferenceSession(
    "resnet50.onnx",
    providers=['VitisAIExecutionProvider'],
    provider_options=[vai_ep_options]
)
```

Sample C++ Code

C++ example loading a configuration file called vai_ep_config.json:

```
#include <onnxruntime_cxx_api.h>

auto onnx_model = "resnet50.onnx"
Ort::Env env(ORT_LOGGING_LEVEL_WARNING, "resnet50_bf16");
auto session_options = Ort::SessionOptions();
auto vai_ep_options = std::unorderd_map<std::string,std::string>({});
vai_ep_options["config_file"] = "vai_ep_config.json";
session_options.AppendExecutionProvider_VitisAI(vai_ep_options);
auto session = Ort::Session(
    env,
    std::basic_string<ORTCHAR_T>(onnx_model.begin(), onnx_model.end()).c_
↪str(),
    session_options);
```

3.5.4 Using INT8 Models

Ryzen AI features a new compiler for INT8 models. This compiler is enabled by default and provides the following improvements:

- Improved ease of use and enhanced performance for models running on STX, KRK, and later devices.
- General asymmetric quantization support to enable third-party quantized models
- Support for XINT8, A8W8, and A16W8 quantization configuration providing greater flexibility for model optimization.

The `target` provider options can be used to select which backend to use when compiling the INT8 model. The option accepts the following values:

- `X2` — Default backend for integer models. Supports STX, KRK and newer devices.
- `X1` — Legacy backend for integer models. Supports PHX, HPT, STX and KRK devices. This setting should be used when running on PHX and HPT devices. It can also be used on STX and KRK devices in the cases where better results are achieved than with the default X2 setting.

Device-Specific Settings

Suitable settings for the `target` and `xclbin` provider options are dependent on the type of device. The application must perform a device detection check before configuring the Vitis AI EP. For more details on how to do this, refer to the *Application Development* page.

When compiling INT8 models on STX/KRK devices:

- The `target` provider option can be set to `X2` (default) or `X1` (legacy backend, may provide better results for some models).
- The `xclbin` provider option must not be set.

When compiling INT8 models on PHX/HPT devices:

- The `target` provider option must be set to `X1`.
- The `xclbin` provider option must be set to `%RYZEN_AI_INSTALLATION_PATH%\voe-4.0-win_amd64\xclbins\phoenix\4x4.xclbin` or to a copy of this file included in the final version of the application. The legacy "1x4" and "Nx4" xclbin files are no longer supported and should not be used.

Sample Python Code

Python example code for running an INT8 model on STX/KRK NPU (target=X2, no xclbin):

```
import os
import onnxruntime

vai_ep_options = {
    'cache_dir': str(cache_dir),
    'cache_key': 'resnet_trained_for_cifar10',
    'enable_cache_file_io_in_mem':'0',
    'target': 'X2' # Default option 'X2'
}

session = onnxruntime.InferenceSession(
    "resnet50_int8.onnx",
    providers=['VitisAIExecutionProvider'],
    provider_options=[vai_ep_options]
)
```

Sample C++ Code

C++ example code for running an INT8 model on STX/KRK NPU (target=X2, no xclbin):

```
#include <onnxruntime_cxx_api.h>

auto onnx_model = "resnet50_int8.onnx"
Ort::Env env(ORT_LOGGING_LEVEL_WARNING, "resnet50_int8");
auto session_options = Ort::SessionOptions();
auto vai_ep_options = std::unordered_map<std::string,std::string>({});
vai_ep_options["cache_dir"]   = exe_dir + "\\my_cache_dir";
vai_ep_options["cache_key"]   = "resnet_trained_for_cifar10";
vai_ep_options["enable_cache_file_io_in_mem"]   = "0";
vai_ep_options["target"]   = "X2";
session_options.AppendExecutionProvider_VitisAI(vai_ep_options);
auto session = Ort::Session(
    env,
    std::basic_string<ORTCHAR_T>(onnx_model.begin(), onnx_model.end()).c_
↪str(),
    session_options);
```

3.5.5 Managing Compiled Models

To avoid the overhead of recompiling models, it is very advantageous to save the compiled models and use these pre-compiled versions in the final application. Pre-compiled models can be loaded instantaneously and immediately executed on the NPU. This greatly improves the session creation time and overall end-user experience.

The RyzenAI Software supports two mechanisms for saving and reloading compiled models:

- VitisAI EP Cache
- ONNX Runtime EP Context Cache

TIP: The VitisAI EP Cache mechanism is most convenient to quickly iterate during the development cycle. The OnnxRuntime EP Context Cache mechanism is recommended for the final version of the application.

VitisAI EP Cache

The VitisAI EP includes a built-in caching mechanism. When a model is compiled for the first time, it is automatically saved in the VitisAI EP cache directory. Any subsequent creation of an ONNX Runtime session using the same model will load the precompiled model from the cache directory, thereby reducing session creation time.

The VitisAI EP Cache mechanism can be used to quickly iterate during the development cycle, but it is not recommended for the final version of the application.

Cache directories generated by the Vitis AI Execution Provider should not be reused across different versions of the Vitis AI EP or across different version of the NPU drivers.

If using the VitisAI EP Cache the application should check the version of the Vitis AI EP and of the NPU drivers. If the application detects a version change, it should delete the cache, or create a new cache directory with a different name.

The location of the VitisAI EP cache is specified with the `cache_dir` and `cache_key` provider options. For INT8 models, the `enable_cache_file_io_in_mem` must be set to 0 otherwise the output of the compiler is kept in memory and is not saved to disk.

Python example:

```
import onnxruntime
from pathlib import Path

vai_ep_options = {
    'cache_dir': str(Path(__file__).parent.resolve()),
    'cache_key': 'compiled_resnet50_int8',
    'enable_cache_file_io_in_mem': 0
}

session = onnxruntime.InferenceSession(
    "resnet50_int8.onnx",
    providers=['VitisAIExecutionProvider'],
    provider_options=[vai_ep_options]
)
```

In the example above, the cache directory is set to the absolute path of the folder containing the script being executed. Once the session is created, the compiled model is saved inside a subdirectory named `compiled_resnet50_int8` within the specified cache folder.

ONNX Runtime EP Context Cache

The Vitis AI EP supports the ONNX Runtime EP context cache feature. This features allows dumping and reloading a snapshot of the EP context before deployment.

The user can enable dumping of the EP context by setting the `ep.context_enable` session option to 1.

The following options can be used for additional control:

- `ep.context_file_path` – Specifies the output path for the dumped context model.
- `ep.context_embed_mode` – Embeds the EP context into the ONNX model when set to 1.

For further details, refer to the official ONNX Runtime documentation: https://onnxruntime.ai/docs/execution-providers/EP-Context-Design.html

EP Context Encryption

By default, the generated context model is unencrypted and can be used directly during inference. If needed, the context model can be encrypted using one of the methods described below.

User-managed encryption

After the context model is generated, the developer can encrypt the generated file using a method of choice. At runtime, the encrypted file can be loaded by the application, decrypted in memory and passed as a serialized string to the inference session.

This method gives complete control to the developer over the encryption process.

EP-managed encryption

The VitisAI EP can optionally encrypt the EP context model using AES256. This is enabled by passing an encryption key using the `encryption_key` VAI EP provider options. The key is a 256-bit value represented as a 64-digit string. At runtime, the same encryption key must be provided to decrypt and load the context model.

With this method, encryption and decryption is seamlessly managed by the VitisAI EP.

Python example:

```
import onnxruntime

vai_ep_options = {
    'encryptionKey':
↪'89703f950ed9f738d956f6769d7e45a385d3c988ca753838b5afbc569ebf35b2'
}

# Compilation session
session_options = ort.SessionOptions()
session_options.add_session_config_entry('ep.context_enable', '1')
session_options.add_session_config_entry('ep.context_file_path', 'context_
↪model.onnx')
session_options.add_session_config_entry('ep.context_embed_mode', '1')
session = ort.InferenceSession(
    path_or_bytes='resnet50_int8.onnx',  # Load the ONNX model
    sess_options=session_options,
    providers=['VitisAIExecutionProvider'],
    provider_options=[vai_ep_options]
)

# Inference session
session_options = ort.SessionOptions()
```

(continues on next page)

(continued from previous page)

```
session = ort.InferenceSession(
    path_or_bytes='context_model.onnx', # Load the EP context model
    sess_options=session_options,
    providers=['VitisAIExecutionProvider'],
    provider_options=[vai_ep_options]
)
```

C++ example:

```
Ort::Env env(ORT_LOGGING_LEVEL_WARNING, "ort");

// VAI EP Provider options
auto vai_ep_options = std::unorderd_map<std::string,std::string>({});
vai_ep_options["encryption_key"] =
 ↪"89703f950ed9f738d956f6769d7e45a385d3c988ca753838b5afbc569ebf35b2";

// Session options
auto session_options = Ort::SessionOptions();
session_options.AppendExecutionProvider_VitisAI(vai_ep_options);

// Inference session
auto onnx_model = "context_model.onnx"; // The EP context model
auto session = Ort::Session(
    env,
    std::basic_string<ORTCHAR_T>(onnx_model.begin(), onnx_model.end()).c_
 ↪str(),
    session_options);
```

NOTE: It is possible to precompile the EP context model using Python and to deploy it using a C++ program.

3.5.6 Operator Assignment Report

The compiler can optionally generate a report on operator assignments across CPU and NPU. To generate this report:

- The `enable_cache_file_io_in_mem` provider option must be set to 0
- The XLNX_ONNX_EP_REPORT_FILE environment variable must be used to specify the name of the generated report. For instance:

```
set XLNX_ONNX_EP_REPORT_FILE=vitisai_ep_report.json
```

When these conditions are satisfied, the report file is automatically generated in the cache directory. This report includes information such as the total number of nodes, the list of operator types in the model, and which nodes and operators run on the NPU or on the CPU. Additionally, the report includes node statistics, such as input to a node, the applied operation, and output from the node.

When these conditions are satisified, the report file is automatically generated in the cache directory. This report includes information such as the total number of nodes, the list of operator types in the model, and which nodes and operators runs on the NPU or on the CPU. Additionally, the report includes node statistics, such as input to a node, the applied operation, and output from the node.

```
{
 "deviceStat": [
 {
  "name": "all",
  "nodeNum": 400,
  "supportedOpType": [
   "::Add",
   "::Conv",
   ...
  ]
 },
 {
  "name": "CPU",
  "nodeNum": 2,
  "supportedOpType": [
   "::DequantizeLinear",
   "::QuantizeLinear"
  ]
 },
 {
  "name": "NPU",
  "nodeNum": 398,
  "supportedOpType": [
   "::Add",
   "::Conv",
   ...
  ]
  ...
```

To disable generation of the report, unset the XLNX_ONNX_EP_REPORT_FILE environment variable:

```
set XLNX_ONNX_EP_REPORT_FILE=
```

3.6 Application Development

This page captures requirements and recommendations for developers looking to create, package and distribute applications targeting NPU-enabled AMD processors.

3.6.1 VitisAI EP / NPU Driver Compatibility

The VitisAI EP requires a compatible version of the NPU drivers. For each version of the VitisAI EP, compatible drivers are bounded by a minimum version and a maximum release date. NPU drivers are backward compatible with VitisAI EP released up to three years. The maximum driver release date is therefore set to three years after the release date of the corresponding VitisAI EP.

The following table summarizes the driver requirements for the different versions of the VitisAI EP.

VitisAI EP version	Minimum NPU Driver version	Maximum NPU Driver release date
1.7	32.0.203.280	Jan 22nd, 2029
1.6	32.0.203.280	Oct 7th, 2028
1.5	32.0.203.280	July 1st, 2028
1.4.1	32.0.203.259	May 13th, 2028
1.4	32.0.203.257	March 25th, 2028
1.3.1	32.0.203.242	January 17th, 2028
1.3	32.0.203.237	November 26th, 2027
1.2	32.0.201.204	July 30th, 2027

The application must verify that NPU drivers compatible with the version of the Vitis AI EP in use are installed.

3.6.2 APU Types

The Ryzen AI Software supports various types of NPU-enabled APUs, referred to as PHX, HPT, STX, and KRK. To programmatically determine the type of the local APU, you can enumerate the PCI devices and look for an instance with a matching Hardware ID.

Vendor	Device	Revision	APU Type
0x1022	0x1502	0x00	PHX or HPT
0x1022	0x17F0	0x00	STX
0x1022	0x17F0	0x10	STX
0x1022	0x17F0	0x11	STX
0x1022	0x17F0	0x20	KRK

The application must verify that it is running on an AMD processor with an NPU, and that the NPU type is supported by the version of the Vitis AI EP in use.

3.6.3 NPU Utilities

When deploying applications across various NPU devices, users can determine the specific type of NPU device using Python/C++ code. Based on the detected device—such as PHX, STX, KRK, or other device—users should configure the appropriate provider options as outlined in *Model Compilation and Deployment*.

For Python, the user can get the specific NPU type using the following example *get_npu_info* function in the `%RYZEN_AI_INSTALLATION_PATH%\quicktest\quicktest.py`

For C++, a set of APIs are provided to extract information about the NPU and check driver compatibility of the VitisAI EP with the rest of the environment. For details refer to C++ NPU Utilities

3.6.4 Application Development Requirements

ONNX-RT Session

The application should only use the Vitis AI Execution Provider if the following conditions are met:

- The application is running on an AMD processor with an NPU type supported by the version of the Vitis AI EP in use. See *list*.
- NPU drivers compatible with the version of the Vitis AI EP being used are installed. See *compatibility table* .

NOTE: Sample C++ code that implements the compatibility checks to be performed before using the Vitis AI EP is available here

VitisAI EP Provider Options

For INT8 models, the application should detect the type of APU present (PHX, HPT, STX, or KRK) and set the `target` and `xclbin` provider options accordingly. Refer to the section on *using INT8 models* for more details.

For BF16 models, the application should set the `config_file` provider option to the same file that was used to precompile the BF16 model. Refer to the section on *using BF16 models* for more details.

Pre-Compiled Models

To avoid the overhead of recompiling models, it is highly recommended to save the compiled models and use these precompiled versions in the final application. Precompiled models can be loaded instantly and executed immediately on the NPU, significantly improving session creation time and overall end-user experience.

AMD recommends using the ONNXRuntime *EP Context Cache* feature for saving and reloading compiled models.

BF16 models

The deployment version of the VitisAI Execution Provider (EP) does not support the on-the-fly compilation of BF16 models. Applications utilizing BF16 models must include pre-compiled versions of these models. The VitisAI EP can then load the pre-compiled models and deploy them efficiently on the NPU.

INT8 models

Including pre-compiled versions of INT8 models is recommended but not mandatory.

3.6.5 Application Packaging Requirements

An updated version of the Ryzen AI deployment DLLs is available for download at the following link. These updated DLLs are intended to replace the ones located in the %RYZEN_AI_INSTALLATION_PATH%/deployment folder of the Ryzen AI 1.7 installation tree.

A C++ application built on the Ryzen AI ONNX Runtime must include the following components in its distribution package:

For INT8 models

- DLLs:
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\aiecompiler_client.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\DirectML.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\dyn_dispatch_core.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime_providers_shared.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime_providers_vitisai.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime_vitis_ai_custom_ops.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime_vitisai_ep.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\zlib.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\zstd.dll

- NPU Binary files (.xclbin) from the `%RYZEN_AI_INSTALLATION_PATH%\voe-4.0-win_amd64\xclbins` folder
- Recommended but not mandatory: pre-compiled models in the form of *Onnx Runtime EP context models*

For BF16 models

- DLLs:
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime_providers_shared.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime_providers_vitisai.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime_vitisai_ep.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\dyn_dispatch_core.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\ryzenai_onnx_utils.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\zlib.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\zstd.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\aiecompiler_client.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\vaiml.dll
- Pre-compiled models in the form of *Vitis AI EP cache folders*

For LLMs

- DLLs:
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime-genai.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnxruntime.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\ryzen_mm.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\onnx_custom_ops.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\libutf8_validity.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\abseil_dll.dll
  - %RYZEN_AI_INSTALLATION_PATH%\deployment\DirectML.dll

3.7 Whisper.cpp support

Ryzen AI provides NPU acceleration for whisper.cpp through an AMD-maintained fork. On AMD Ryzen AI 300 Series systems, Whisper can fully offload the encoder to the NPU, which can deliver a significant speedup versus CPU-only runs. NPU acceleration is currently supported on Windows only, with Linux support planned.

For setup steps, build flags, and NPU-optimized model guidance, refer to the GitHub README at https://github.com/amd/whisper.cpp?tab=readme-ov-file#amd-ryzen-ai-support-for-npu

3.8 LLM Deployment Overview

Large Language Models (LLMs) can be deployed on Ryzen AI PCs with NPU and GPU acceleration. NPU-only and Hybrid execution modes, which utilize both the NPU and integrated GPU (iGPU), are supported via ONNXRuntime GenAI (OGA). GPU-only acceleration is enabled through llama.cpp. See the *LLM Execution Mode Comparison* below for detailed information.

3.8.1 Execution Modes

Table 5: LLM Execution Mode Comparison

Mode	Framework(s)	Compute Allocation	Primary Use Case
NPU-Only	OnnxRuntime GenAI (OGA)	Neural Processing Unit (NPU) exclusive	Maximum NPU utilization while preserving iGPU for parallel workloads
Hybrid	OnnxRuntime GenAI (OGA)	Dynamic NPU + iGPU partitioning	Interactive inference with optimal prefill/decode performance
GPU	llama.cpp	Dedicated GPU execution	High-throughput inference on discrete/integrated GPU
CPU	OGA or llama.cpp	Traditional CPU-based inference	Baseline compatibility across all processor generations

3.8.2 Hardware Requirements

Table 6: Supported Processor Configurations

Processor Series	NPU-Only	Hybrid	GPU/CPU
Ryzen AI 300 (STX/KRK)	✓	✓	✓
Ryzen AI 7000/8000			✓

Development Interfaces

The Ryzen AI LLM software stack is available through three development interfaces, each suited for specific use cases as outlined in the sections below. All three interfaces are built on top of native OnnxRuntime GenAI (OGA) libraries or llama.cpp libraries, as shown in the *Ryzen AI Software Stack* diagram below.

The high-level Python APIs, as well as the Server Interface, also leverage the Lemonade SDK, which is multi-vendor open-source software that provides everything necessary for quickly getting started with LLMs on OGA or llama.cpp.

A key benefit of Lemonade is that software developed against their interfaces is portable to many other execution backends.

Table 7: Ryzen AI Software Stack

Your Python Application	Your LLM Stack	Your Native Application
*Lemonade Python API**	*Lemonade Server Interface**	OGA C++ Headers **OR** llama.cpp C++ Headers
Custom AMD OnnxRuntime GenAI (OGA) **OR** llama.cpp*		
AMD Ryzen AI Driver and Hardware		

* indicates open-source software (OSS).

3.8.3 Server Interface (REST API)

The Server Interface provides a convenient means to integrate with applications that:

- Already support an LLM server interface, such as the Ollama server or OpenAI API.
- Are written in any language (C++, C#, Javascript, etc.) that supports REST APIs.
- Benefits from process isolation for the LLM backend.

Lemonade Server is available in two ways:

- **Standalone Windows GUI installer**: Quick setup with a desktop shortcut for immediate use. (Recommended for end users, see *Server Interface (REST API)*)
- **Full Lemonade SDK**: Complete development toolkit with server interface included. (Recommended for developers, see *High-Level Python SDK* for Python SDK)

For example applications that have been tested with Lemonade Server, see the Lemonade Server Examples.

3.8.4 High-Level Python SDK

The high-level Python SDK, Lemonade, allows you to get started using PyPI installation in approximately 5 minutes.

This SDK allows you to:

- Experiment with models in hybrid or NPU-only execution mode on Ryzen AI hardware.
- Validate inference speed and task performance.
- Integrate with Python apps using a high-level API.

To get started in Python, follow these instructions: *High-Level Python SDK*.

3.8.5 OGA APIs for C++ Libraries and Python

Native C++ libraries for OGA are available to give full customizability for deployment into native applications. The Python bindings for OGA also provide a customizable interface for Python development.

To get started with the OGA APIs, follow these instructions: *OnnxRuntime GenAI (OGA) Flow*.

Supported LLMs

AMD provides a set of pre-optimized LLMs ready to be deployed with Ryzen AI Software and the supporting runtime for hybrid and/or NPU-only execution. These include popular architectures such as Llama-2, Llama-3, Mistral, DeepSeek Distill models, Qwen-2, Qwen-2.5, Qwen-3, Gemma-2, Phi-3, Phi-3.5, and Phi-4. For the detailed list of supported models, visit ../model_list

Hugging Face collection of hybrid models: https://huggingface.co/collections/amd/ryzen-ai-16-hybrid-llm-68d9c3ed502f871223bfa882

Hugging Face collection of NPU models: https://huggingface.co/collections/amd/ryzen-ai-16-npu-llm-68d9c927223939cb596c592b

It is also possible to run fine-tuned versions of the models listed (for example, fine-tuned versions of Llama2 or Llama3). For instructions on how to prepare a fine-tuned OGA model, refer to *Preparing OGA Models*.

End to End OGA Validation

A Jupyter Notebook example is provided to demonstrate end-to-end validation of OGA hybrid and NPU-only execution. This notebook includes:

- Installation
- Command Syntax
- Benchmarking
- Subjective Evaluation
- Objective Evaluation

To run the notebook, visit the Lemonade Tools Tutorial.

3.9 Server Interface (REST API)

The Lemonade SDK offers a server interface that allows your application to load an LLM on Ryzen AI hardware in a process, and then communicate with this process using standard `REST` APIs. This allows applications written in any language (C#, JavaScript, Python, C++, etc.) to easily integrate with Ryzen AI LLMs.

Server interfaces are used across the LLM ecosystem because they allow for no-code plug-and-play between the higher level of the application stack (GUIs, agents, RAG, etc.) with the LLM and hardware that have been abstracted by the server. For more information, see the Understanding local LLM Servers Guide.

For example, open source projects such as *Open WebUI* have out-of-box support for connecting to a variety of server interfaces, which in turn allows users to quickly start working with LLMs in a GUI.

3.9.1 Server Setup

Lemonade Server can be installed via the Lemonade Server Installer executable by following these steps:

1. Make sure your system has the recommended Ryzen AI driver installed as described in *Install NPU Drivers*.
2. Download and install `Lemonade_Server_Installer.exe` from the latest Lemonade release.
3. Launch the server by double-clicking the `lemonade_server` shortcut added to your desktop.

For a visual walkthrough of this process, watch our Lemonade Introductory Video:

See the Lemonade Server Documentation for more details.

3.9.2 Server Usage

The Lemonade Server provides the following OpenAI-compatible endpoints:

- POST `/api/v1/chat/completions` - Chat Completions (messages to completions)
- POST `/api/v1/completions` - Text Completions (prompt to completion)
- POST `/api/v1/responses` - Chat Completions (prompt|messages -> event)
- GET `/api/v1/models` - List available models

Please refer to the server specification document for details about the request and response formats for each endpoint.

The OpenAI API documentation also has code examples for integrating streaming completions into an application.

Supported Applications

The Lemonade Server supports a variety of applications that can connect to it using the OpenAI API. Some of the applications that have been tested with Lemonade Server can be found at Lemonade Server Apps.

A short list of applications that have been tested with Lemonade Server includes:

3.9.3 Next Steps

- See Lemonade Server Examples to find applications that have been tested with Lemonade Server.
- Check out the Lemonade Server specification to learn more about supported features.
- Try out your Lemonade Server install with any application that uses the OpenAI chat completions API.

3.10 High-Level Python SDK

A Python environment offers flexibility for experimenting with LLMs, profiling them, and integrating them into Python applications. We use the Lemonade SDK to get up and running quickly.

To get started, follow these instructions.

3.10.1 System-level pre-requisites

You only need to do this once per computer:

1. Make sure your system has the recommended Ryzen AI driver installed as described in *Install NPU Drivers*.
2. Download and install Miniconda for Windows or Miniforge for Windows.
3. Launch a terminal and call `conda init`.

3.10.2 Environment Setup

To create and set up an environment, run these commands in your terminal:

```
conda create -n ryzenai-llm python=3.12
conda activate ryzenai-llm
pip install lemonade-sdk[dev,oga-ryzenai] --extra-index-url=https://pypi.
↪amd.com/simple
```

3.10.3 Validation Tools

Now that you have completed installation, you can try prompting an LLM like this (where `PROMPT` is any prompt you like).

Run this command in a terminal that has your environment activated:

```
lemonade -i amd/Llama-3.2-1B-Instruct-awq-g128-int4-asym-fp16-onnx-hybrid␣
↪oga-load --device hybrid --dtype int4 llm-prompt --max-new-tokens 64 -p␣
↪PROMPT
```

For an end-to-end example demonstrating the Validation Tools, visit the Lemonade Tools Tutorial.

3.10.4 Python API

You can also run this code to try out the high-level Lemonade API in a Python script:

```
from lemonade.api import from_pretrained

model, tokenizer = from_pretrained(
    "amd/Llama-3.2-1B-Instruct-awq-g128-int4-asym-fp16-onnx-hybrid",␣
↪recipe="oga-hybrid"
)

input_ids = tokenizer("This is my prompt", return_tensors="pt").input_ids
response = model.generate(input_ids, max_new_tokens=30)
```

(continues on next page)

(continued from previous page)

```
print(tokenizer.decode(response[0]))
```

3.10.5 Next Steps

From here, you can check out the Jupyter Notebook that provides an end-to-end validation of OGA hybrid and NPU-only execution. To run the notebook, visit the Lemonade Tools Tutorial.

3.11 OnnxRuntime GenAI (OGA) Flow

Ryzen AI Software supports deploying LLMs on Ryzen AI PCs using the native ONNX Runtime Generate (OGA) C++ or Python API. The OGA API is the lowest-level API available for building LLM applications on a Ryzen AI PC. It supports the following execution modes:

- Hybrid execution mode: This mode uses both the NPU and iGPU to achieve the best TTFT and TPS during the prefill and decode phases.
- NPU-only execution mode: This mode uses the NPU exclusively for both the prefill and decode phases. Two types of NPU models are available:
  - **Token Fusion models**: Support long context up to 16K tokens with no additional configuration required.
  - **Full Fusion models**: Optimized for best performance, supporting up to 4096 total tokens (input + output).

3.11.1 Supported Configurations

The Ryzen AI OGA flow supports Strix and Krackan Point processors. Phoenix (PHX) and Hawk (HPT) processors are not supported.

3.11.2 Requirements

- Install NPU Drivers and Ryzen AI MSI installer. See *Installation Instructions* for more details.
- Install GPU device driver: Ensure GPU device driver https://www.amd.com/en/support is installed
- Install Git for Windows (needed to download models from HF): https://git-scm.com/downloads

3.11.3 Pre-optimized Models

AMD provides a set of pre-optimized LLMs ready to be deployed with Ryzen AI Software and the supporting runtime for hybrid and/or NPU-only execution. These include popular architectures such as Llama-2, Llama-3, Mistral, DeepSeek Distill models, Qwen-2, Qwen-2.5, Qwen-3, Gemma-2, Gemma-3, GPT-OSS, Phi-3, Phi-3.5, and Phi-4. For the detailed list of supported models, visit model_list

Hugging Face collection of hybrid models: https://huggingface.co/collections/amd/ryzen-ai-171-hybrid

Hugging Face collection of NPU Token Fusion models: https://huggingface.co/collections/amd/ryzen-ai-171-npu-16k

Hugging Face collection of NPU Full Fusion models: https://huggingface.co/collections/amd/ryzen-ai-171-npu-4k

> **Note**
>
> Links to be updated soon

NPU Models: Token Fusion vs Full Fusion

AMD provides two types of NPU models:

- **Token Fusion models**: These models support long context up to 16K tokens. They are pre-built and uploaded to Hugging Face — no additional configuration is required to use long context. Simply download and run the model.
- **Full Fusion models**: These models are optimized for best inference performance but do not support long context. The total token count (input + output) must not exceed 4096.

Choose the model type based on your use case: Token Fusion for long context workloads, or Full Fusion for maximum throughput on shorter sequences.

Each OGA model folder contains a `genai_config.json` file. This file contains various configuration settings for the model. The `session_option` section is where information about specific runtime dependencies is specified.

3.11.4 Changes Compared to Previous Release

- OGA version is updated to v0.11.2 (Ryzen AI 1.7) from v0.9.2.2 (Ryzen AI 1.6.1).
- For 1.7 release, a new set of hybrid and NPU models is published. Models from earlier releases are not compatible with this version. If you are using Ryzen AI 1.7, please download the updated models.
- Two types of NPU models are now available: **Token Fusion** models (long context up to 16K tokens) and **Full Fusion** models (best performance, up to 4096 tokens).

- Context length up to 4K tokens (combined input and output) is supported for Full Fusion NPU models. Extended context length up to 16K tokens is supported for Token Fusion NPU models and Hybrid models.

3.11.5 Compatible OGA APIs

Pre-optimized hybrid or NPU LLMs can be executed using the official OGA C++ and Python APIs. The current release is compatible with OGA version 0.11.2. For detailed documentation and examples, refer to the official OGA repository: https://github.com/microsoft/onnxruntime-genai/tree/rel-0.11.2

3.11.6 LLMs Test Programs

The Ryzen AI installation includes test programs (in C++ and Python) that can be used to run LLMs and understand how to integrate them in your application.

The steps for deploying the pre-optimized models using the sample programs are described in the following sections.

Steps to run C++ program and sample python script.

1. (Optional) Enable Performance Mode

To run LLMs in best performance mode, follow these steps:

- Go to `Windows` → `Settings` → `System` → `Power`, and set the power mode to **Best Performance**.
- Open a terminal and run:

```
cd C:\Windows\System32\AMD
xrt-smi configure --pmode performance
```

2. Activate the Ryzen AI Conda Environment and install `torch` library.

Run the following commands:

```
conda activate ryzen-ai-<version>
pip install torch==2.7.1
```

This step is required for running the python script.

> **Note**
>
> For the C++ program, if you choose not to activate the Conda environment, open a Windows Command Prompt and manually set the environment variable before continuing:
>
> `set RYZEN_AI_INSTALLATION_PATH=C:\\Program Files\\RyzenAI\\<version>`

C++ Program

Use the `model_benchmark.exe` executable to test LLMs and identify DLL dependencies for C++ applications.

1. Set Up a working directory and copy required Files

```
mkdir llm_run
cd llm_run

:: Copy the sample C++ executable
xcopy /Y "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\model_benchmark.exe" .

:: Copy the sample prompt file
xcopy /Y "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\amd_genai_prompt.txt" .

:: Copy required DLLs
xcopy /Y "%RYZEN_AI_INSTALLATION_PATH%\deployment\." .
```

2. Download model from Hugging Face

```
:: Install Git LFS if you haven't already: https://git-lfs.com
git lfs install

:: Clone the model repository
git clone https://huggingface.co/amd/Llama-2-7b-chat-hf-onnx-ryzenai-
↪hybrid
```

3. Run `model_benchmark.exe`

```
.\model_benchmark.exe -i <path_to_model_dir> -f <prompt_file> -l <list_of_
↪prompt_lengths>

:: Example:
.\model_benchmark.exe -i Llama-2-7b-chat-hf-onnx-ryzenai-hybrid -f amd_
↪genai_prompt.txt -l "1024"
```

Long Context Support

Ryzen AI supports long context (beyond 4096 tokens) for **Hybrid models** and **Token Fusion NPU models**.

Token Fusion NPU Models

Token Fusion NPU models are pre-built with long context support up to 16K tokens. No additional configuration is required — simply download the model from Hugging Face and run it.

```
:: Example: Clone a Token Fusion NPU model
git clone https://huggingface.co/amd/Phi-3.5-mini-instruct-onnx-ryzenai-
↪npu

:: Run with long context
.\model_benchmark.exe -i <path_to_model_dir> -f amd_genai_prompt_long.txt␣
↪-l "16000"
```

Hybrid Models

If the total number of tokens exceeds 4096 for a hybrid model, follow the steps below.

Steps to run long context:

1. Make the following changes in `genai_config.json` file.
   - Add `"hybrid_opt_chunk_context": "1"` under `model.decoder.session_options.provider_options.RyzenAI`.

```
{
"model": {
   "bos_token_id": 1,
   "context_length": 16384,
   "decoder": {
         "session_options": {
         "log_id": "onnxruntime-genai",
         "provider_options": [
         {
            "RyzenAI": {
               "external_data_file": "model_jit.pb.bin",
               "hybrid_opt_free_after_prefill": "1",
               "hybrid_opt_max_seq_length": "4096",
                                             "hybrid_opt_chunk_
↪context": "1"
            }
         }
```

(continues on next page)

(continued from previous page)

```
            ]
        },
```

- Add `"chunk_size":2048` under `search`.

```

```

“search”: {
“diversity_penalty”: 0.0, “do_sample”: false, “chunk_size”: 2048, . . .

2. Copy the `amd_genai_prompt_long.txt` into your working directory.

```
xcopy /Y "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\amd_genai_prompt_long.
↪txt" .
```

3. Run the model using `model_benchmark.exe` using the `amd_genai_prompt_long.txt` prompt file.

```
.\model_benchmark.exe -i <path_to_model_dir> -f amd_genai_prompt_long.txt␣
↪-l "16000"
```

Note

The sample test application `model_benchmark.exe` accepts `-l` for input token length and `-g` for output token length.

- **Full Fusion NPU models** support up to 4096 tokens in total (input + output). By default, `-g` is set to 128. If the input length is close to 4096, you must adjust `-g` so the sum of input and output tokens does not exceed 4096. For example, `-l 4000 -g 96` is valid (4000 + 96 4096), while `-l 4000 -g 128` will exceed the limit and result in an error.
- **Token Fusion NPU models** support long context up to 16K tokens (input + output) with no additional configuration.
- **Hybrid models**: The combined number of input and output tokens must not exceed the model’s `context_length`. You can verify the `context_length` in the `genai_config.json` file. For example, if a model’s `context_length` is 8,000, the total token count (input + output) must not exceed 8,000.

The long context feature has been tested for Token Fusion NPU models and Hybrid models up to 16,000 tokens.

Python Script

1. Navigate to your working directory and download model.

```
:: Install Git LFS if you haven't already: https://git-lfs.com
git lfs install

:: Clone the model repository
git clone https://huggingface.co/amd/Llama-2-7b-chat-hf-onnx-ryzenai-
↪hybrid
```

2. Run sample python script

```
python "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\run_model.py" -m <model_
↪folder> -l <max_length>

:: Example command
python "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\run_model.py" -m "Llama-
↪2-7b-chat-hf-onnx-ryzenai-hybrid" -l 256
```

Note

Some models may return non-printable characters in their output (for example, Qwen models), which can cause a crash while printing the output text. To avoid this, modify the provided script %RYZEN_AI_INSTALLATION_PATH%\LLM\example\run_model.py by adding a text sanitization function and updating the print statement as shown below.

Add sanitize_string function:

```
def sanitize_string(input_string):
   return input_string.encode("charmap", "ignore").decode("charmap")
```

Update line 80 to print sanitized output:

```
print("Output:", sanitize_string(output_text))
```

This sanitization fix will be included in the run_model.py script in the next release.

Python Script (with Chat Template)

For models that use chat templates, the `model_chat.py` script provides better output quality by automatically loading and applying the chat template from the model folder during inference. The script also supports single-prompt, multi-turn context cache testing, and interactive chat with timing output.

The script is included in the Ryzen AI installation:

```
:: Single prompt with timing
python "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\model_chat.py" -m <model_
↪folder> -pr amd_genai_prompt.txt --timings

:: Long context support (increase context window to e.g. 16k)
python "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\model_chat.py" -m <model_
↪folder> -pr amd_genai_prompt_long.txt -mpt 16000

:: Interactive chat
python "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\model_chat.py" -m <model_
↪folder>
```

For the full list of options including multi-turn JSON testing, guided generation, and advanced flags, refer to the RyzenAI-SW repository.

It is highly recommended to use `model_chat.py` for the GPT-OSS-20B NPU model.

3.11.7 Vision Language Model (VLM)

AMD provides a pre-optimized Gemma-3-4b-it multimodal model ready to be deployed with Ryzen AI Software. Support for this model is available starting with the Ryzen AI 1.7 release.

Model: Gemma-3-4b-it-mm-onnx-ryzenai-npu

VLM inference requires dedicated Python scripts, which are included in the Ryzen AI installation at `%RYZEN_AI_INSTALLATION_PATH%\LLM\example\vlm`.

Quick Inference

Use `vlm_run.py` to quickly test a model and see output:

```
:: Basic inference
python "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\vlm\vlm_run.py" -m
↪<model_folder> -i <image_path>

:: Custom prompt
python "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\vlm\vlm_run.py" -m
↪<model_folder> -i <image_path> -p "What's in this image?"

:: Resize image before running
python "%RYZEN_AI_INSTALLATION_PATH%\LLM\example\vlm\vlm_run.py" -m
↪<model_folder> -i <image_path> --image_size 1024 1024
```

For benchmarking scripts (`vlm_benchmark.py`, `run_all_benchmarks.py`) and detailed options, refer to the README in the `vlm` directory or the RyzenAI-SW repository.

3.11.8 Building C++ Applications

A complete example including C++ source and build instructions is available in the RyzenAI-SW repository: https://github.com/amd/RyzenAI-SW/tree/main/example/llm/oga_api

3.11.9 Using Fine-Tuned Models

It is also possible to run fine-tuned versions of the pre-optimized OGA models.

To do this, the fine-tuned models must first be prepared for execution with the OGA flow. For instructions on how to do this, refer to the page about *Preparing OGA Models*.

After a fine-tuned model has been prepared for execution, it can be deployed by following the steps described previously in this page.

3.11.10 Running LLM via pip install

In addition to the full RyzenAI software stack, we also provide standalone wheel files for the users who prefer using their own environment. To prepare an environment for running the Hybrid and NPU-only LLM independently, perform the following steps:

1. Create a new python environment and activate it.

```
conda create -n <env_name> python=3.12 -y
conda activate <env_name>
```

2. Install onnxruntime-genai wheel file.

```
pip install onnxruntime-genai-directml-ryzenai==0.11.2 --extra-index-
↪url=https://pypi.amd.com/simple
    pip install model-generate==1.7.0 --extra-index-url=https://pypi.amd.
↪com/simple
```

3. Navigate to your working directory and download the desired Hybrid/NPU model

```
cd working_directory
git clone <link_to_model>
```

4. Run the Hybrid or NPU model.

3.12 Preparing OGA Models

This section describes the process for preparing LLMs for deployment on a Ryzen AI PC using the hybrid or NPU-only execution mode. Currently, the flow supports only fine-tuned versions of the models already supported (as listed in *OnnxRuntime GenAI (OGA) Flow* page). For example, fine-tuned versions of Llama2 or Llama3 can be used. However, different model families with architectures not supported by the hybrid flow cannot be used.

For fine-tuned models that introduce architectural changes requiring new operator shapes not available in the Ryzen AI runtime, refer to the oga_op_prepare

Preparing a LLM for deployment on a Ryzen AI PC involves 2 steps:

1. **Quantization**: The pretrained model is quantized to reduce memory footprint and better map to compute resources in the hardware accelerators
2. **Postprocessing**: During the postprocessing the model is exported to OGA followed by NPU-only or Hybrid execution mode specific postprocess to obtain the final deployable model.

3.12.1 Quantization

Prerequisites

Linux machine with AMD (e.g., AMD Instinct MI Series) or Nvidia GPUs

Setup

1. Create and activate Conda Environment

```
conda create --name <conda_env_name> python=3.11
conda activate <conda_env_name>
```

2. If Using AMD GPUs, update PyTorch to use ROCm

```
pip3 install torch torchvision torchaudio --index-url https://download.
↪pytorch.org/whl/rocm6.1
python -c "import torch; print(torch.cuda.is_available())" #␣
↪Must return `True`
```

3. Download `AMD Quark 0.11` and unzip the archive
4. Install Quark:

```
cd <extracted amd quark-version>
pip install amd_quark-<version>+<>.whl
```

5. Install other dependencies

```
pip install datasets
pip install transformers
pip install accelerate
pip install evaluate
pip install nltk
```

Some models may require a specific version of `transformers`. For example, ChatGLM3 requires version 4.44.0.

Generate Quantized Model

Use following command to run Quantization. In a GPU equipped Linux machine the quantization can take about 30-60 minutes.

```
cd examples/torch/language_modeling/llm_ptq/

python quantize_quark.py \
     --no_trust_remote_code \
     --model_dir "meta-llama/Llama-2-7b-chat-hf"  \
     --output_dir <quantized safetensor output dir>  \
     --quant_scheme uint4_wo_128 \
     --num_calib_data 128 \
     --seq_len 512 \
     --quant_algo awq \
     --dataset pileval_for_awq_benchmark \
     --model_export hf_format \
     --data_type <datatype> \
     --exclude_layers []
```

- Use `--data_type bfloat16` for bf16 pretrained model. For fp32/fp16 pretrained model use `--datatype float16`
- Not using `--exclude_layers` parameter may result in model-specific defaults which may exclude certain layers like output layers.
- To specify a group size other than 128, such as 32, use `--quant_scheme uint4_wo_32` instead of `--quant_scheme uint4_wo_128`. Available group sizes are 32, 64, and 128 (e.g., uint4_wo_32, uint4_wo_64, uint4_wo_128)
- Quark supports quantizing layers with different group sizes, use `--layer_quant_scheme lm_head uint4_wo_32` to quantize the model with 32 group size for lm_head

The quantized model is generated in the <quantized safetensor output dir> folder.

Note: For the Phi-4 model, the following quantization recipe is recommended for better accuracy:

- Use `--quant_algo gptq`
- Add `--layer_quant_scheme lm_head uint4_wo_32`

Note:: Currently the following files are not copied into the quantized model folder and must be copied manually:

- For Phi-4 models: `configuration_phi3.py`
- For ChatGLM-6b models: `tokenizer.json`

3.12.2 Postprocessing

Copy the quantized model to the Windows PC with Ryzen AI installed, activate the Ryzen AI Conda environment.

```
conda activate ryzen-ai-<version>
pip install onnx_ir
pip install torch==2.7.1
```

Generate the final model for Hybrid execution mode:

```
conda activate ryzen-ai-<version>

model_generate --hybrid <output_dir> <quantized_model_path>
```

Generate the final model for NPU execution mode:

```
conda activate ryzen-ai-<version>

model_generate --npu <output_dir> <quantized_model_path>  --optimize␣
↪decode
```

Generate model for hybrid execution mode (prefill fused version)

```
conda activate ryzen-ai-<version>

model_generate --hybrid <output_dir> <quantized_model_path>  --optimize␣
↪prefill
```

- Prefill fused hybrid models are only supported for Phi-3.5-mini-instruct and Mistral-7B-Instruct-v0.2
- Edit *genai_config.json* with the following entries

```
"decoder": {
      "session_options": {
          "log_id": "onnxruntime-genai",
          "custom_ops_library": "onnx_custom_ops.dll",
          "external_data_file": "token.pb.bin",
          "custom_allocator": "ryzen_mm",
          "config_entries": {
              "dd_cache": "",
              "hybrid_opt_token_backend": "gpu",
              "hybrid_opt_max_seq_length": "4096",
              "max_length_for_kv_cache": "4096"
          },
```

(continues on next page)

(continued from previous page)

```
            "provider_options": []
        },
        "filename": "fusion.onnx",
```

Note: During the `model_generate` step, the quantized model is first converted to an OGA model using ONNX Runtime GenAI Model Builder (version 0.9.2). It is possible to use a standalone environment for exporting an OGA model, refer to the official ONNX Runtime GenAI Model Builder documentation. Once you have an exported OGA model, you can pass it directly to the `model_generate` command, which will skip the export step and perform only the post-processing.

Here are simple commands to export OGA model from quantized model using a standalone environment

```
conda create --name oga_builder_env python=3.10
conda activate oga_buider_env


pip install onnxruntime-genai==0.9.2
# pip install other necessary packages
pip install ....


python3 -m onnxruntime_genai.models.builder -m <input quantized model> -o
↪<output OGA model> -p int4 -e dml
```

3.13 DirectML Flow

3.13.1 Prerequisites

- DirectX12 capable Windows OS (Windows 11 recommended)
- Latest AMD GPU device driver installed
- Microsoft Olive for model conversion and optimization
- Latest ONNX Runtime DirectML EP

You can ensure GPU driver and DirectX version from `Windows Task Manager` -> `Performance` -> `GPU`

3.13.2 Running models on Ryzen AI GPU

Running models on the Ryzen AI GPU is accomplished in two simple steps:

Model Conversion and Optimization: After the model is trained, Microsoft Olive Optimizer can be used to convert the model to ONNX and optimize it for optimal target execution.

For additional information, refer to the Microsoft Olive Documentation

Deployment: Once the model is in the ONNX format, the ONNX Runtime DirectML EP (`DmlExecutionProvider`) is used to run the model on the AMD Ryzen AI GPU.

For additional information, refer to the ONNX Runtime documentation for the DirectML Execution Provider

3.13.3 Examples

- Optimizing and running ResNet on Ryzen AI GPU

3.13.4 Additional Resources

- Article on how AMD and Black Magic Design worked together to accelerate Davinci Resolve Studio workload on AMD hardware:
  - AI Accelerated Video Editing with DaVinci Resolve 18.6 & AMD Radeon Graphics
- Blog posts on using the Ryzen AI Software for various generative AI workloads on GPU:
  - Automatic1111 Stable Diffusion WebUI with DirectML Extension on AMD GPUs
  - Running Optimized Llama2 with Microsoft DirectML on AMD Radeon Graphics
  - AI-Assisted Mobile Workstation Workflows Powered by AMD Ryzen™ AI

3.14 Windows ML Overview

This section describes how to use **Windows ML on AMD Ryzen AI PCs** and complements the official Microsoft Windows AI documentation. It bridges Microsoft's local AI platform with Ryzen AI hardware and software.

Microsoft provides a comprehensive AI platform for Windows spanning three pillars:

- **Windows AI APIs:** Built-in system APIs (OCR, image description, super resolution, object erase, etc.) for Copilot+ PCs. Use when your scenario is covered by these APIs.
- **Foundry Local:** On-device runtime for LLMs and generative AI; auto-detects hardware and downloads compatible models. Use for LLM scenarios with minimal setup.
- **Windows ML:** Runtime for custom ONNX models with automatic execution provider (EP) management across CPU, GPU, and NPU. Use when you need to run your own models.

On **Ryzen AI PCs**, Windows ML can leverage the NPU via the VitisAI EP (Execution Provider).

3.14.1 When to Use Windows ML

Choose Windows ML when you:

- Need to run **custom ONNX models** (CNN, Transformer, or LLM) on Windows
- Want **automatic EP management** Windows downloads and registers compatible execution providers (VitisAI EP, MIGraphX EP, DirectML EP) on demand
- Prefer **C#, C++, or Python** with a shared Windows-wide ONNX Runtime (smaller app size)
- Need **hardware flexibility** select CPU, GPU, or NPU via execution policy

Use **Windows AI APIs** when built-in capabilities (OCR, image description, etc.) cover your scenario. Use **Foundry Local** when you want LLMs with minimal model preparation. Use the **Ryzen AI NPU-only flow** (*Model Compilation and Deployment*) when you need full control over ONNX Runtime without the Windows ML stack.

3.14.2 Quick Links

Topic	Page
Prerequisites and installation	*Installation*
VitisAI EP model support (CNN, Transformer, LLM)	*VitisAI EP Model Support*
Model deployment	*Model Deployment*
Model conversion and quantization (AI Toolkit)	*Model Conversion and Quantization (AI Toolkit)*
Execution providers (registration, EP policy, compilation)	*Execution Providers*
Windows ML examples (CNN, Transformer, LLM)	*Windows ML Example*
Frequently asked questions	*Frequently Asked Questions*
Troubleshooting	*Troubleshooting*

3.14.3 External Resources

- Windows AI Developer Portal
- Windows ML official documentation
- AI on Windows samples (Microsoft Learn)
- WindowsAppSDK-Samples — WindowsML
- RyzenAI-SW Windows ML examples

3.15 Installation

This page consolidates all requirements and installation steps for running Windows ML on Ryzen AI PCs.

3.15.1 Prerequisites

Requirement	Version or Notes
Windows	Windows 11 24H2 (build 26100) or greater
Ryzen AI NPU	Supported processor with NPU. See *supported configurations* in the release notes.
Visual Studio (for C++)	Visual Studio 2022, latest version. Ensure *Desktop Development with C++* is installed.
Visual Studio Code (optional)	For model conversion using AI Toolkit extension
Python (for Python examples)	3.10 to 3.12
C++ (for C++ examples)	C++20 or later

For the complete list of supported Windows versions, refer to Windows App SDK support.

3.15.2 Installation

Follow these steps in order:

1. **Install NPU drivers:** Follow the *RAI installation instructions*. Download and install the NPU driver (version 32.0.203.280 or newer) from the AMD Ryzen AI driver page.
2. **Install Windows App SDK:** Windows ML is included as part of the Windows App SDK. Install the version required by your branch/sample from Windows App SDK downloads.
3. **Verify installation:** For Python, ensure the `wasdk-microsoft-windows-ai-machinelearning` package is installed (e.g., via `pip install`) and matches the Windows App SDK version used by your sample branch. Run `conda list | findstr wasdk` to verify.

3.15.3 Key Features of Windows ML

- **Dynamically downloads latest EPs:** Compatible execution providers are downloaded from the Microsoft Store on demand
- **Shared Windows-wide ONNX Runtime:** Reduces application size; no need to bundle ORT
- **Smaller downloads and installs:** EPs are shared across applications
- **Broad hardware support:** Works across CPUs, GPUs, and NPUs from different vendors via ONNX Runtime

3.15.4 Windows ML setup verification

Install the required Python packages in the conda environment *winml_env*

```
conda create -n winml_env python==3.11
conda activate winml_env
git clone https://github.com/amd/RyzenAI-SW.git
cd <RyzenAI-SW>\WinML\CNN\ResNet
pip install --pre -r .\requirements.txt
```

Check the installed wasdk Python version and install same version of Windows App SDK:

```
conda list | findstr wasdk
```

Download the Windows App SDK corresponding to the wasdk version (e.g., 2.0.0.dev4) or latest and install it to ensure the Windows ML execution providers work correctly.

```
curl -L -o windowsappruntimeinstall-x86.exe "https://aka.ms/windowsappsdk/
↪2.0/2.0.0-experimental4/windowsappruntimeinstall-x86.exe"
windowsappruntimeinstall-x86.exe --quiet
```

After completing the installation, run the *check_winml_setup.py* script from the *RyzenAI-SW* repository to verify the Windows ML installation. The script is available at: https://github.com/amd/RyzenAI-SW/blob/main/WinML/check_winml_setup.py

```
cd <RyzenAI-SW>\WinML
python check_winml_setup.py
```

The script will produce output similar to the following:

```
============================================================
WinML Setup Checker
============================================================
Python: 3.11.0 (<path_to_python_installation>\python.exe)
WASDK Python Packages:
----------------------------------------
  [✓] wasdk-ML: 2.0.0.dev4
  [✓] wasdk-Bootstrap: 2.0.0.dev4
Windows App SDK Runtime:
----------------------------------------
[✓] Windows App SDK: 2.0-experimental5 (internal: 0.770.2319.0)
Installed runtimes (newest first):
    * 2.0-experimental5 (internal: 0.770.2319.0)
    - 2.0-experimental4 (internal: 0.738.2207.0)
    - 1.8 (internal: 8000.642.119.0)
    - 1.8 (internal: 8000.675.1142.0)
```

(continues on next page)

(continued from previous page)

```
    - 1.8-experimental (internal: 8000.589.1529.0)
    - 1.8-preview (internal: 8000.591.1127.0)
    * Active runtime used by this checker
Expected SDK: 2.0.0-experimental4
============================================================
Status: All components installed. Please, ensure matching Windows App SDK␣
↪version is Installed.
```

Windows App SDK version should match the wasdk Python package version. If there is a mismatch, install the correct Windows App SDK version. After installation, re-run the setup checker to verify that the correct version of Windows App SDK is installed and active.

3.15.5 Getting Started Examples

The following examples provide step-by-step instructions to help you get started with Windows ML on AMD Ryzen AI PCs. These examples cover CNN, Transformer, and LLM model deployment using both C++ and Python APIs.

- Getting Started Example for Windows ML using ResNet model:
  - Optional Model conversion to QDQ quantized ONNX model using VS Code AI Toolkit
  - Deployment using Windows ML APIs and ONNX Runtime using C++ and Python
- Additional examples:
  - Transformer based GoogleBERT example
  - Running OpenAI CLIP model on NPU
  - Running LLM models on NPU

For more details about model deployment using Windows ML, see the *Model Deployment using Windows ML documentation*.

3.16 VitisAI EP Model Support

The VitisAI EP (Execution Provider) within Windows ML supports input models in the following formats.

3.16.1 Model Support Table

Model Type	Support
CNN Models	• Original float (FP32) model with automatic BF16 conversion during compilation • Quantized QDQ model using A8W8 configuration
Transformer Models	• Original float (FP32) model with automatic BF16 conversion during compilation • Quantized QDQ model using A16W8 configuration
LLM Models (via Foundry Local)	• Quantized and pre-compiled LLM models • Support for custom models through Olive recipe

Note

- For CNN and Transformer models, you can use either the original float model (with automatic BF16 conversion) or a quantized QDQ model. Quantization can reduce model size and improve inference performance.
- For LLMs, Foundry Local provides pre-built models that auto-detect the NPU. Custom LLM deployment may require model preparation using the Olive recipe or Ryzen AI OGA workflow. See Windows ML LLM examples for details.
- For model conversion and quantization options, see *Model Conversion and Quantization (AI Toolkit)*.

3.17 Model Deployment

Windows Machine Learning (WinML) enables C#, C++, and Python developers to run ONNX AI models locally on Windows PCs through ONNX Runtime, with automatic execution provider management across hardware targets including CPUs, GPUs, and NPUs. You can use models from PyTorch, TensorFlow/Keras, TensorFlow Lite (TFLite), scikit-learn, and other frameworks by converting them to ONNX for ONNX Runtime.

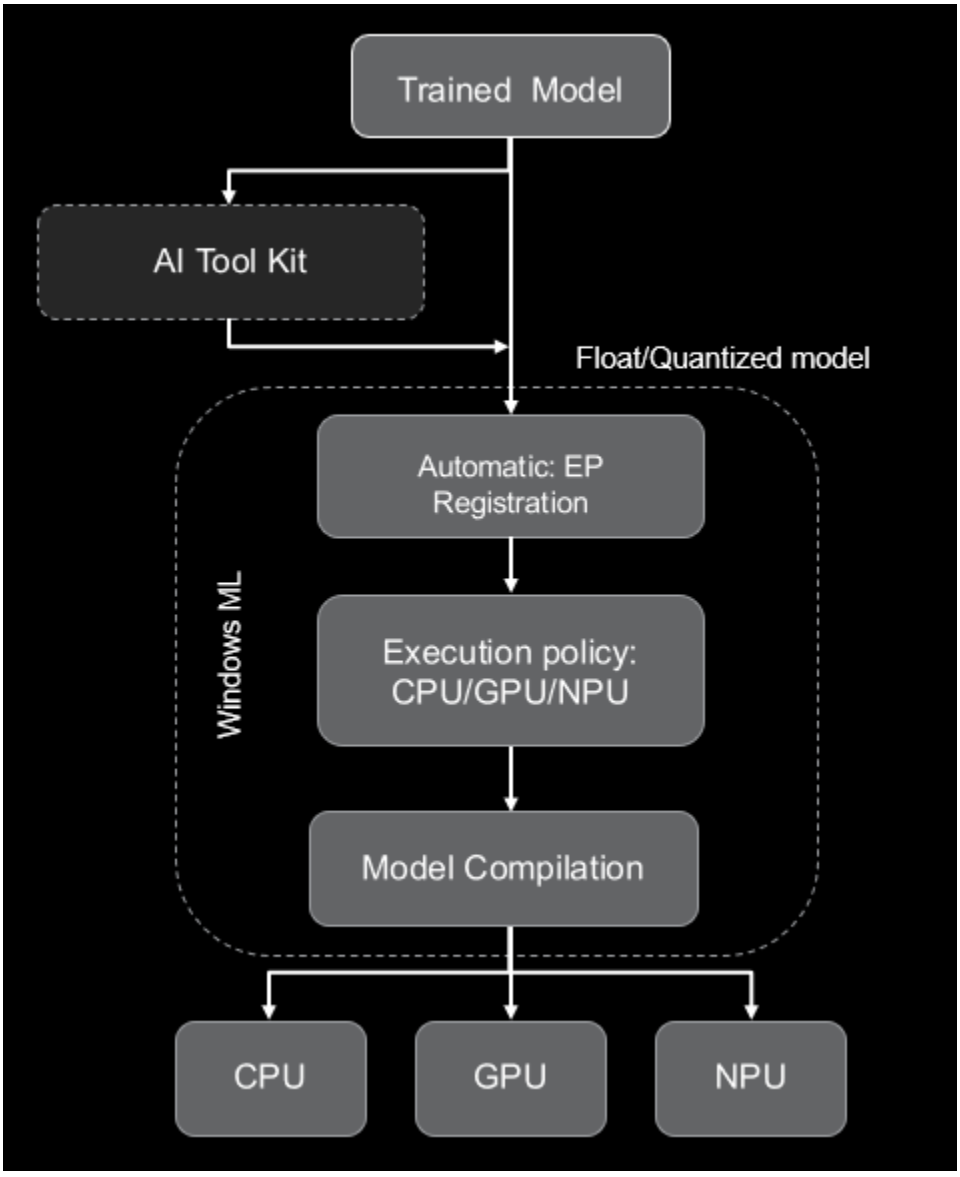
In short, Windows ML provides a shared, Windows-wide ONNX Runtime along with support for dynamically downloading execution providers (EPs).

For more details, see the Windows ML official documentation.

3.17.1 Running CNN/Transformer models on NPU

Windows ML provides a streamlined workflow for deploying CNN and Transformer models on Ryzen AI PCs. Users can either use the original float model with automatic BF16 conversion, or use AI Toolkit for model quantization (QDQ format).

Windows ML workflow



Step 1: Download the Original Float Model

Start with your pre-trained ONNX model in FP32 format. Models can be exported from PyTorch, TensorFlow, or obtained from model repositories.

Step 2: Model Quantization using VS AI Toolkit (Optional)

For improved inference performance, quantize your model using VS AI Toolkit or Olive recipe:

- **A8W8 quantization**: Recommended for CNN models (ResNet, MobileNet, etc.)
- **A16W8 quantization**: Recommended for Transformer models (BERT, CLIP etc.)

Step 3: Automatic Execution Provider Registration

Windows ML automatically downloads and registers the appropriate execution providers based on available hardware:

Execution Provider	Hardware Target
VitisAIExecutionProvider	AMD Ryzen AI NPU
MIGraphXExecutionProvider	AMD GPU (ROCm)
DmlExecutionProvider	DirectML (GPU/NPU)

Step 4: Execution Policy for device selection

Select the preferred execution target using the execution policy:

Execution Policy	First Preference EP
PREFER_CPU	CPUExecutionProvider
PREFER_GPU	DmlExecutionProvider
PREFER_NPU	VitisAIExecutionProvider

The EP selection policy can be configured to use a specific execution provider or through general execution policy. For more details, refer to the Windows ML documentation on *Execution Providers*.

Step 5: Model Compilation

The model is compiled for the target hardware:

- **Float models**: VAIML performs automatic BF16 conversion for NPU execution
- **Quantized models**: A8W8/A16W8 models are compiled using X2/X1 compiler

For more details refer to the *model compilation and deployment* documentation.

Step 6: Model Inference

Use the ONNX Runtime APIs to run inference on the compiled model. The model will execute on the selected hardware target based on the execution policy and available EPs.

3.17.2 Running LLM models on NPU

Windows ML enables support for Foundry Local models for on-device AI inference solutions that provide privacy and performance. Currently, Foundry Local is available in preview mode. It automatically detects NPU and downloads the compatible model for the NPU device.

LLM prerequisites

Make sure the following requirements are met before proceeding:

Requirement	Details
Operating System	Windows 10, Windows 11
Hardware (Minimum)	8 GB RAM, 3 GB free disk space
Hardware (Recommended)	16 GB RAM, 15 GB free disk space
Acceleration	AMD NPU

Running LLM on AMD NPU

LLM models can be run on AMD NPU using Foundry Local or Windows ML APIs. Foundry Local provides an easy-to-use interface for running LLM models on AMD NPU, while Windows ML APIs allow for more customization and control over the inference process.

Option 1: Running LLM using Foundry Local

This is the recommended option for most users as it provides a simple and efficient way to run LLM models on AMD NPU without needing to manage dependencies or optimize the model manually.

Option 2: Running a Custom LLM Model using Windows ML and OGA APIs

This option allows users to run custom LLM models on AMD NPU using Windows ML APIs. This option is suitable for users who want more control over the inference process and are comfortable managing dependencies and model optimization manually.

For detailed instructions on each option, see the Running LLM Models on NPU documentation.

3.18 Model Conversion and Quantization (AI Toolkit)

The **AI Toolkit (AITK)** for Visual Studio Code is the primary tool for model conversion and quantization when preparing models for Windows ML on Ryzen AI.

AITK supports:

- **Model conversion:** Export models from PyTorch, TensorFlow, and other frameworks to ONNX
- **Model quantization:** Convert to QDQ (Quantize-Dequantize) format for lower precision inference
- **Evaluation:** Run models on CPU, GPU, or NPU to validate accuracy and performance

3.18.1 Quantization Options

Option	Values
Activation type	INT8, UINT8, INT16, UINT16, BF16
Weight type	INT8, UINT8, INT16, UINT16, INT4, BF16

Recommended Precision Settings by Model Type:

- CNN Models: Use **A8W8** quantization (activation INT8/UINT8, weight INT8/UINT8)
- Transformer Models: Use **A16W8** quantization (activation INT16/UINT16, weight INT8/UINT8)
- LLM Models: BF16 and INT4 precision options are available

3.18.2 Device Evaluation

You can evaluate quantized models on CPU, GPU, or NPU to compare accuracy and performance before deployment.

3.18.3 Known Limitations

- **AMD GPU conversion:** Model conversion for AMD GPU may fail due to limited Olive and Quark AMD GPU support. Use NPU or CPU for conversion and evaluation when possible.
- **Windows vs Linux:** For larger LLM models, model conversion is done on Linux with GPU support, due to limited support on Windows. See the Windows ML LLM examples for details.

3.18.4 References

- VS Code AI Toolkit model conversion
- *Model quantization* (Ryzen AI Quark flow for NPU-only path)

3.19 Execution Providers

Windows ML provides a system-level execution provider (EP) management layer for ONNX Runtime on Windows PCs. It automatically discovers, downloads, and registers the best-available EPs for your hardware, whether that is a CPU, GPU, or NPU, so your application always runs on the optimal hardware accelerator.

Windows ML ships a shared, Windows-wide ONNX Runtime and exposes EP management APIs for C#, C++, and Python. Through these APIs you can:

- **Auto-register all compatible EPs** with a single API call, Windows ML handles version resolution and updates.

- **Set an execution policy** (e.g., `PREFER_NPU`) to steer workloads to a preferred device class with automatic fallback.
- **Target a specific EP and device** by enumerating available EP devices and appending the one you need — for example, `VitisAIExecutionProvider` on an AMD NPU.
- **Compile models for a specific EP** as a one-time step that optimizes the model for the target hardware, and the compiled artifact can be cached for all subsequent runs.

Models from PyTorch, TensorFlow/Keras, TensorFlow Lite (TFLite), scikit-learn, and other frameworks can be converted to ONNX and executed through this managed EP infrastructure.

For more details, see the Windows ML official documentation.

3.19.1 Automatic Execution Providers (EPs) Registration

Windows ML will automatically discover, download, and register the latest version of all compatible execution providers.

C++ Example

```
#include <winrt/Microsoft.Windows.AI.MachineLearning.h>
#include <win_onnxruntime_cxx_api.h>

// First we need to create an ORT environment
Ort::Env env(ORT_LOGGING_LEVEL_ERROR, "WinMLDemo"); // Use an ID of your␣
↪own choice

// Get the default ExecutionProviderCatalog
winrt::Microsoft::Windows::AI::MachineLearning::ExecutionProviderCatalog␣
↪catalog =
winrt::Microsoft::Windows::AI::MachineLearning::ExecutionProviderCatalog::GetDefault()
↪

// Ensure and register all compatible execution providers with ONNX␣
↪Runtime
catalog.EnsureAndRegisterAllAsync().get();
```

Python Example

```
# Known issue: import winrt.runtime will cause the TensorRTRTX execution␣
↪provider to fail registration.
# As a workaround, run pywinrt related code in a separate thread.

# winml.py
import json
```

(continues on next page)

(continued from previous page)

```
def _get_ep_paths() -> dict[str, str]:
    from winui3.microsoft.windows.applicationmodel.dynamicdependency.
↪bootstrap import (
        InitializeOptions,
        initialize
    )

    import winui3.microsoft.windows.ai.machinelearning as winml
    eps     = {}
    with initialize(options = InitializeOptions.ON_NO_MATCH_SHOW_UI):
        catalog = winml.ExecutionProviderCatalog.get_default()
        providers = catalog.find_all_providers()
        for provider in providers:
            provider.ensure_ready_async().get()
            eps[provider.name] = provider.library_path
            # DO NOT call provider.try_register in Python. That will␣
↪register to the native env.
    return eps

if __name__ == "__main__":
    eps = _get_ep_paths()
    print(json.dumps(eps))

# In your application code
import subprocess
import json
import sys
from pathlib import Path
import onnxruntime as ort

def register_execution_providers():
    worker_script = str(Path(__file__).parent / 'winml.py')
    result = subprocess.check_output([sys.executable, worker_script],␣
↪text=True)
    paths = json.loads(result)
    for item in paths.items():
        ort.register_execution_provider_library(item[0], item[1])
    _ep_registered = True

register_execution_providers()
```

The `register_execution_providers` function is used to download and register the latest ver-

sion of all compatible execution providers.

3.19.2 Execution Policy

The EP selection policy can be configured to use specific execution provider or through general execution policy. For more details, refer to the Windows ML documentation on Execution Providers.

For example, setting the execution policy to *PREFER_NPU* will prioritize the NPU execution provider if available, with a fallback to CPU execution if an NPU is not present.

C++ Example

```
// Configure the session to select an EP and device for PREFER_NPU which␣
↪typically
// will choose an NPU if available with a CPU fallback.
Ort::SessionOptions sessionOptions;
sessionOptions.SetEpSelectionPolicy(OrtExecutionProviderDevicePolicy_
↪PREFER_NPU);
```

Python Example

```
# Configure the session to select an EP and device for PREFER_NPU which␣
↪typically
# will choose an NPU if available with a CPU fallback.
options = ort.SessionOptions()
options.set_provider_selection_policy(ort.
↪OrtExecutionProviderDevicePolicy.PREFER_NPU)
assert options.has_providers()
```

Specifying the **specific execution provider** can be done through the *set_providers* API. For example, setting the execution provider to *VitisAIExecutionProvider* will only use the VitisAI EP for model execution.

C++ Example

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <stdexcept>
#include <win_onnxruntime_cxx_api.h>

// Assuming you have an Ort::Env named 'env'
// 1. Enumerate EP devices
std::vector<Ort::ConstEpDevice> ep_devices = env.GetEpDevices();
```

(continues on next page)

(continued from previous page)

```
// 2. Collect only ReplaceWithExecutionProvider NPU devices
std::vector<Ort::ConstEpDevice> selected_ep_devices;
for (const auto& d : ep_devices) {
    if (std::string(d.EpName()) == "VitisAIExecutionProvider"
        && d.HardwareDevice().Type() == OrtHardwareDeviceType_NPU) {
        selected_ep_devices.push_back(d);
    }
}
if (selected_ep_devices.empty()) {
    throw std::runtime_error("VitisAIExecutionProvider is not available␣
↪on this system.");
}

// 3. Configure provider-specific options (varies based on EP)
// and append the EP with the correct devices (varies based on EP)
Ort::SessionOptions session_options;
Ort::KeyValuePairs ep_options;
ep_options.Add("optimize_level", "1");
session_options.AppendExecutionProvider_V2(env, { selected_ep_devices.
↪front() }, ep_options);
```

Python Example

```
# 1. Enumerate and filter EP devices
ep_devices = ort.get_ep_devices()
selected_ep_devices = [
    d for d in ep_devices
    if d.ep_name == "VitisAIExecutionProvider"
    and d.device.type == ort.OrtHardwareDeviceType.NPU]

if not selected_ep_devices:
    raise RuntimeError("VitisAIExecutionProvider is not available on this␣
↪system.")

# 2. Configure provider-specific options in "vaiml_config.json" file
session_options = ort.SessionOptions()
provider_options = {'config_file':'vaiml_config.json'}
session_options.add_provider_for_devices([selected_ep_devices[0]],␣
↪provider_options)
```

For more details on the *VitisAIExecutionProvider*-specific *provider_options*, see *Model compilation and deployment*

3.19.3 Model Compilation

Models need to be compiled for specific EPs. This is a one-time process that stores the compiled model for subsequent runs:

C++ Example

```
bool isCompiledModelAvailable =␣
↪std::filesystem::exists(compiledModelPath);

if (!isCompiledModelAvailable)
{
    Ort::ModelCompilationOptions compile_options(env, sessionOptions);
    compile_options.SetInputModelPath(modelPath.c_str());
    compile_options.SetOutputModelPath(compiledModelPath.c_str());

    std::cout << "Starting compile, this may take a few moments..." <<␣
↪std::endl;
    Ort::Status compileStatus = Ort::CompileModel(env, compile_options);
    if (compileStatus.IsOK())
    {
        std::cout << "Model compiled successfully!" << std::endl;
        isCompiledModelAvailable = true;
    }
}
```

Python Example

```
input_model_path = "path_to_your_model.onnx"
output_model_path = "path_to_your_compiled_model.onnx"

model_compiler = ort.ModelCompiler(
    options,
    input_model_path,
    embed_compiled_data_into_model=True,
    external_initializers_file_path=None,
)
model_compiler.compile_to_file(output_model_path)
if not os.path.exists(output_model_path):
    # For some EPs, there might not be a compilation output.
    # In that case, use the original model directly.
    output_model_path = input_model_path
```

For more details on *VitisAIExecutionProvider* specific *provider_options* as shown in the reference documentation *Model compilation and deployment*

3.20 Windows ML Example

This example demonstrates how to deploy a ResNet model using Windows ML, covering the complete workflow for converting, quantizing, compiling, and deploying models in both Python and C++ environments.

This tutorial provides step-by-step instructions for deploying a ResNet model, demonstrating:

- Setup instructions to create the Python environment and install dependencies
- Download the ResNet ONNX model
- (Optional) Quantize the model using AI Toolkit to QDQ ONNX format for low precision inference
- Compile and run the model on NPU using ONNX Runtime with Vitis AI Execution Provider using Python/C++ code

3.20.1 Setup Instructions

The source code files can be downloaded from this link. Alternatively, you can clone the RyzenAI-SW repo and change the directory into "WinML".

```
git clone https://github.com/amd/RyzenAI-SW.git
cd RyzenAI-SW/WinML/CNN/ResNet/python
```

The NPU driver and Windows App SDK should be correctly installed, as described in *Windows ML Installation*.

```
conda create -n winml_resnet python==3.11
conda activate winml_resnet
cd <RyzenAI-SW>\WinML\CNN\ResNet
pip install -r .\requirements.txt
```

Check the installed *Windows App SDK* Python package using the command below.

```
conda list | findstr wasdk
```

This should print the installed version of the *Windows App SDK* Python package. Ensure that the version is 2.0.0.dev4 or later.

```
wasdk-microsoft-windows-ai-machinelearning 2.0.0.dev4                pypi_
↪0    pypi
wasdk-microsoft-windows-applicationmodel-dynamicdependency-bootstrap 2.0.
↪0.dev4               pypi_0    pypi
```

Ensure that the installed *Windows App SDK* version matches the Python package or download the specific version from Windows App SDK 2.0.0.dev4.

3.20.2 Model Conversion

Model conversion is the first step in preparing your model for deployment with Windows ML. You can use the AI Toolkit to convert models to the ONNX format and apply quantization.

- Model quantization is optional step that can help reduce model size and improve inference performance.
- Original float model can be passed through automatic BF16 conversion. For more details refer to *Model conversion*
- See the VS Code AI Toolkit model conversion page for details on model conversion using AI Toolkit.

If skipping the model quantization, you can directly download the ResNet ONNX model using the script:

```
cd <RyzenAI-SW>\WinML\CNN\ResNet\model
python download_ResNet.py
```

3.20.3 Python Deployment

This section covers how to compile and deploy ResNet ONNX model using Python script. You can choose to deploy either the original FP32 ONNX model or the quantized QDQ ONNX model.

Model Inference

Use the Python script to run inference which compiles and runs the model on NPU using ONNX Runtime with Vitis AI Execution Provider. If you are using quantized model specify the quantized model path e.g. *model\model_a8w8.onnx* and if you are using original FP32 model specify the original model path e.g. *model\resnet50.onnx*.

```
cd <RyzenAI-SW>\WinML\CNN\ResNet\python
python run_model.py --model ..\model\resnet50.onnx --image_path ..\images\
↪dog.jpg --ep_policy NPU
```

Sample Output

The following is a sample output showing the top-5 predictions from the model. You should see class indices and their associated confidence scores.

```
285, Egyptian cat with confidence of 0.904274
281, tabby with confidence of 0.0620204
282, tiger cat with confidence of 0.0223081
287, lynx with confidence of 0.00119624
761, remote control with confidence of 0.000487919
```

3.20.4 C++ Deployment

C++ deployment is recommended for production scenarios where performance and integration with native Windows applications are critical. This section shows how to compile and deploy your model using C++ APIs

Model Inference

Instructions to build the example application and run using the Visual Studio Developer Command Prompt:

```
cd <RyzenAI-SW>\WinML\CNN\cpp\CppResnetBuildDemo\
nuget.exe restore .\CppResnetBuildDemo.sln
msbuild .\CppResnetBuildDemo.sln /p:Configuration=Release /m
```

After compiling the model, you can build and run your C++ application to perform inference.

```
.\x64\Release\CppResnetBuildDemo.exe --model ..\..\model\resnet50.onnx --
↪image_path ..\..\images\dog.jpg --ep_policy NPU
```

Sample Output

The output below shows the top-5 predictions from the C++ inference application. You should see similar results as in the Python deployment section.

```
Top Predictions:
-----------------------------------------
Label                            Confidence
-----------------------------------------
207,golden retriever                 52.86%
852,tennis ball                       1.60%
805,soccer ball                       0.62%
208,Labrador retriever                0.61%
238,Greater Swiss Mountain dog        0.42%
-----------------------------------------
-----------------------------------------------
Time taken for 20 iterations: 0 seconds
Avg time per iteration : 19 milliseconds
```

3.20.5 Additional Examples

The following examples provide step-by-step instructions to help you get started with Windows ML on AMD Ryzen AI PCs. These examples cover CNN, Transformer, and LLM model deployment using both C++ and Python APIs.

- Transformer based GoogleBERT example

- Running OpenAI CLIP model on NPU
- Running LLM models on NPU

For more details about model deployment using Windows ML, see the *Model Deployment using Windows ML documentation*.

3.21 Frequently Asked Questions

This page addresses common questions about Windows ML, Windows AI APIs, and Foundry Local on Ryzen AI PCs.

Topics

- *Windows AI APIs*
- *Foundry Local*
- *Windows ML*

3.21.1 Windows AI APIs

Q: Do Windows AI APIs run on AMD NPUs?

A: Most Windows AI APIs have model options that run on CPU or GPU. A few support NPU. Check the Windows AI APIs documentation for the latest capability matrix per API.

Q: When should I use Windows AI APIs vs Windows ML?

A: Use **Windows AI APIs** when built-in capabilities cover your scenario:

- OCR (Optical Character Recognition)
- Image description
- Super resolution
- Object erase
- Background blur

Use **Windows ML** when you need to run custom ONNX models that are not covered by the built-in APIs.

3.21.2 Foundry Local

Q: Which models run with the Foundry Local CLI?

A: Run `foundry model list` to see supported models. The list is updated as new models are added.

Q: Does Foundry Local support Hybrid (NPU + iGPU) or NPU-only?

A: Currently: **NPU-only**. Hybrid execution is on the roadmap.

Q: Is there an API for Foundry Local?

A: The Foundry Local CLI and service provide an OpenAI-compatible API. See the Foundry Local documentation and microsoft/Foundry-Local for integration details.

3.21.3 Windows ML

Q: Can developers control or select EP versions?

A: No. Windows ML manages EP versions automatically; the latest compatible version is downloaded from the Microsoft Store on demand.

Q: Can developers pass EP provider options?

A: Yes. When using a specific EP (e.g., VitisAI EP), you can pass provider options via the session configuration. See *Model Deployment* for examples. For VitisAI EP options, see *Model compilation and deployment*.

Q: Is there performance overhead when using Windows ML vs native Ryzen AI?

A: Windows ML uses the same VitisAI EP and underlying ONNX Runtime as the native Ryzen AI flow. The main difference is EP management (Windows ML downloads and registers EPs automatically). For performance-critical scenarios, benchmark both paths on your specific hardware and workload.

3.22 Troubleshooting

This page addresses common issues and solutions when using Windows ML on Ryzen AI PCs.

Topics

3.22.1 Installation and Setup

Issue: Windows App SDK Version Mismatch

Symptom: Inference fails or EPs do not load; version mismatch errors.

Solution: Ensure the installed Windows App SDK Python package matches the Windows App SDK version required by your sample branch (stable or preview). Run `conda list | findstr wasdk` to verify. Download the matching version from Windows App SDK downloads.

Issue: EP Not Found or Not Registered

Symptom: Inference fails with "execution provider not found" or similar error message.

Solution:

- Ensure you have called EP registration before creating the session. See *Execution Providers*.
- Run the application with administrator privileges if the EP requires Microsoft Store download.
- Verify the NPU driver is installed. See *Installation*.

Issue: Model Compilation Fails

Symptom: Compilation step fails or times out.

Solution:

- Ensure the model is in a supported format (FP32 or QDQ). See *VitisAI EP Model Support*.
- For quantized models, verify the quantization configuration (A8W8 for CNN, A16W8 for Transformer).
- Check model opset compatibility. ONNX opset 17 is recommended. See *Model compilation and deployment*.

3.22.2 Runtime Issues

Issue: NPU Not Selected

Symptom: Model runs on CPU or GPU instead of NPU.

Solution:

- Set execution policy to `PREFER_NPU` or explicitly use `VitisAIExecutionProvider`. See *Execution Providers*.
- Verify the NPU driver is installed and the device is recognized.
- Check that the model is compatible with the Vitis AI EP. See *VitisAI EP Model Support*.

Issue: TensorRTRTX or Pywinrt Registration Failure (Python)

Symptom: Importing `winrt.runtime` causes the TensorRTRTX execution provider to fail registration.

Solution: Run pywinrt-related code in a **separate process**. Use the subprocess pattern shown in *Execution Providers* (place `winml.py` in the same directory as your application script).

3.23 NPU Management Interface

3.23.1 Introduction

The `xrt-smi` utility is a command-line interface to monitor and manage the NPU integrated AMD CPUs.

On the Windows platform, `xrt-smi` is installed in `C:\Windows\System32\AMD`. Ensure that the following path `C:\Windows\System32\AMD` is set in the System PATH variable. This will allow it to be directly invoked from within the conda environment created by the Ryzen AI Software installer.

The `xrt-smi` utility currently supports three primary commands:

- `examine` - generates reports related to the state of the AI PC and the NPU.
- `validate` - executes sanity tests on the NPU.
- `configure` - manages the performance level of the NPU.

By default, the output of the `xrt-smi examine` and `xrt-smi validate` commands goes to the terminal. It can also be written to file in JSON format as shown below:

```
xrt-smi examine -f JSON -o <path/to/output.json>
```

The utility also support the following options which can be used with any command:

- `--help` - help to use xrt-smi or one of its sub commands
- `--version` - report the version of XRT, driver and firmware
- `--verbose` - turn on verbosity
- `--batch` - enable batch mode (disables escape characters)
- `--force` - when possible, force an operation. Eg - overwrite a file in examine or validate

In Windows, the `xrt-smi` utility requires Microsoft Visual C++ Redistributable (version 2015 to 2022) to be installed.

3.23.2 Overview of Key Commands

Command	Description
examine	system config, device name
examine –report platform	performance mode, power
examine –report aie-partitions	hw contexts
validate –run latency	latency test
validate –run throughput	throughput test
validate –run gemm	INT8 GEMM test TOPS. This is a full array test and it should not be run while another workload is running. **NOTE**: This command is not supported on PHX and HPT NPUs.
configure –pmode <mode>	set performance mode

NOTE: The `examine --report aie-partition` report runtime information. These commands should be used when a model is running on the NPU. You can run these commands in a loop to see live updates of the reported data.

3.23.3 xrt-smi examine

System Information

Reports OS/system information of the AI PC and confirm the presence of the AMD NPU.

```
xrt-smi examine
```

Sample Command Line Output:

```
System Configuration
  OS Name              : Windows NT
  Release              : 26100
  Machine              : x86_64
  CPU Cores            : 20
  Memory               : 32063 MB
  Distribution         : Microsoft Windows 11 Enterprise
  Model                : HP OmniBook Ultra Laptop 14-fd0xxx
  BIOS Vendor          : HP
  BIOS Version         : W81 Ver. 01.01.14

XRT
  Version              : 2.19.0
```

(continues on next page)

(continued from previous page)

```
  Branch                : HEAD
  Hash                  : f62307ddadf65b54acbed420a9f0edc415fefafc
  Hash Date             : 2025-03-12 16:34:48
  NPU Driver Version    : 32.0.203.257
  NPU Firmware Version : 1.0.7.97

Device(s) Present
|BDF              |Name       |
|-----------------|-----------|
|[00c4:00:01.1]   |NPU Strix  |
```

Sample Command Line Output in Linux:

```
> xrt-smi examine
  System Configuration
      OS Name               : Linux
      Release               : 6.11.0-26-generic
      Machine               : x86_64
      CPU Cores             : 24
      Memory                : 31440 MB
      Distribution          : Ubuntu 24.04 LTS
      GLIBC                 : 2.39
      Model                 : BIRMANPLUS
      BIOS Vendor           : AMD
      BIOS Version          : TXB1001dB

  XRT
      Version               : 2.20.0
      Branch                : master
      Hash                  : 7a277facefecab6c87ac835916021c63d2e395dd
      Hash Date             : 2025-06-16 21:28:35
      amdxdna               : 2.20.0_20250617,␣
↪e7233301f8e4d8d1b1678f3dc3492c826290e314
      NPU Firmware Version : 255.0.1.5

  Device(s) Present
  |BDF              |Name       |
  |-----------------|-----------|
  |[0000:c5:00.1]   |NPU Strix  |
```

Sample JSON Output:

```
{
```

(continues on next page)

(continued from previous page)


```
"schema_version": {
    "schema": "JSON",
    "creation_date": "Tue Mar 18 22:43:38 2025 GMT"
},
"system": {
    "host": {
        "os": {
            "sysname": "Windows NT",
            "release": "26100",
            "machine": "x86_64",
            "distribution": "Microsoft Windows 11 Enterprise",
            "model": "HP OmniBook Ultra Laptop 14-fd0xxx",
            "hostname": "XCOUDAYD02",
            "memory_bytes": "0x7d3f62000",
            "cores": "20",
            "bios_vendor": "HP",
            "bios_version": "W81 Ver. 01.01.14"
        },
        "xrt": {
            "version": "2.19.0",
            "branch": "HEAD",
            "hash": "f62307ddadf65b54acbed420a9f0edc415fefafc",
            "build_date": "2025-03-12 16:34:48",
            "drivers": [
                {
                    "name": "NPU Driver",
                    "version": "32.0.203.257"
                }
            ]
        },
        "devices": [
            {
                "bdf": "00c4:00:01.1",
                "device_class": "Ryzen",
                "name": "NPU Strix",
                "id": "0x0",
                "firmware_version": "1.0.7.97",
                "instance": "mgmt(inst=1)",
                "is_ready": "true"
            }
        ]
    }
  }
```


(continues on next page)

(continued from previous page)

```
}
```

Platform Information

Reports more detailed information about the NPU, such as the performance mode and power consumption.

```
xrt-smi examine --report platform
```

Sample Command Line Output:

```
-------------------------
[00c5:00:01.1] : NPU Strix
-------------------------
Platform
  Name                   : NPU Strix
  Power Mode             : Default

Estimated Power          : 1.277 Watts
```

NOTE: Power reporting is not supported on PHX and HPT NPUs. Power reporting is only available on STX devices and onwards. Report "Estimated Power" is currently unavailable for Linux users.

NPU Partitions

Reports details about the NPU partition and column occupancy on the NPU.

```
xrt-smi examine --report aie-partitions
```

Sample Command Line Output:

```
-------------------------
[00c5:00:01.1] : NPU Strix
-------------------------
AIE Partitions
  Partition Index: 0
    Columns: [0, 1, 2, 3]
    HW Contexts:
      |PID    |Ctx ID  |Status  |Instr BO  |Sub  |Compl  |Migr  |Err ␣
↪|Suspensions  |Prio     |GOPS   |EGOPS  |FPS  |Latency  |
      |-------|--------|--------|----------|-----|-------|------|-----|---
↪----------|---------|-------|-------|-----|---------|
      |20696  |0       |Active  |64 KB     |57   |56     |0     |0    |0 ␣
↪          | Normal  |0      |0      |0    |0        |
```

NPU Context Bindings

Reports details about the columns to NPU HW context binding.

```
xrt-smi examine --report aie-partitions --verbose
```

Sample Command Line Output:

```
Verbose: Enabling Verbosity
Verbose: SubCommand: examine

-------------------------
[00c5:00:01.1] : NPU Strix
-------------------------
AIE Partitions
  Partition Index: 0
    Columns: [0, 1, 2, 3]
    HW Contexts:
      |PID    |Ctx ID  |Status  |Instr BO  |Sub  |Compl  |Migr  |Err ␣
↪|Prio    |GOPS  |EGOPS  |FPS  |Latency  |
      |-------|--------|--------|----------|-----|-------|------|-----|---
↪-----|------|-------|-----|---------|
      |20696  |0       |Active  |64 KB     |57   |56     |0     |0   ␣
↪|Normal  |0     |0      |0    |0        |

AIE Columns
  |Column  ||HW Context Slot  |
  |--------||-----------------|
  |0       ||[1]              |
  |1       ||[1]              |
  |2       ||[1]              |
  |3       ||[1]              |
```

3.23.4 xrt-smi validate

Executing a Sanity Check on the NPU

Runs a set of built-in NPU sanity tests which includes latency, throughput, and gemm.

Note: All tests are run in performance mode.

- `latency` - this test executes a no-op control code and measures the end-to-end latency on all columns
- `throughput` - this test loops back the input data from DDR through a MM2S Shim DMA channel back to DDR through a S2MM Shim DMA channel. The data movement within the AIE array follows the lowest latency path i.e. movement is restricted to just the Shim tile.

- `gemm` - An INT8 GeMM kernel is deployed on all 32 cores by the application. Each core is storing cycle count in the core data memory. The cycle count is read by the firmware. The TOPS application uses the "XBUTIL" tool to capture the IPUHCLK while the workload runs. Once all cores are executed, the cycle count from all cores will be synced back to the host. Finally, the application uses IPUHCLK, core cycle count, and GeMM kernel size to calculate the TOPS. This is a full array test and it should not be run while another workload is running. **NOTE**: This command is not supported on PHX and HPT NPUs.
- `all` - All applicable validate tests will be executed (default)

```
xrt-smi validate --run all
```

NOTE: Some sanity checks may fail if other applications (for example MEP, Microsoft Experience Package) are also using the NPU.

Sample Command Line Output:

```
Validate Device           : [00c4:00:01.1]
   Platform               : NPU Strix
   Power Mode             : Performance
-------------------------------------------------------------------------------
 ↪-----
Test 1 [00c4:00:01.1]     : gemm
   Details                : TOPS: 51.3
   Test Status            : [PASSED]
-------------------------------------------------------------------------------
 ↪-----
Test 2 [00c4:00:01.1]     : latency
   Details                : Average latency: 84.2 us
   Test Status            : [PASSED]
-------------------------------------------------------------------------------
 ↪-----
Test 3 [00c4:00:01.1]     : throughput
   Details                : Average throughput: 59891.0 ops
   Test Status            : [PASSED]
-------------------------------------------------------------------------------
 ↪-----
Validation completed. Please run the command '--verbose' option for more␣
 ↪details
```

3.23.5 xrt-smi configure

Managing the Performance Level of the NPU

To set the performance level of the NPU, you can choose from the following modes: powersaver, balanced, performance, or default. Use the command below:

```
xrt-smi configure --pmode <default | powersaver | balanced | performance␣
↪| turbo>
```

- `default` - adapts to the Windows Power Mode setting, which can be adjusted under System -> Power & battery -> Power mode. For finer control of the NPU settings, it is recommended to use the xrt-smi mode setting, which overrides the Windows Power mode and ensures optimal results.
- `powersaver` - configures the NPU to prioritize power saving, preserving laptop battery life.
- `balanced` - configures the NPU to provide a compromise between power saving and performance.
- `performance` - configures the NPU to prioritize performance, consuming more power.
- `turbo` - configures the NPU for maximum performance performance, requires AC power to be plugged in otherwise uses `performance` mode.

Example: Setting the NPU to high-performance mode

```
xrt-smi configure --pmode performance
```

To check the current power mode, use the following command:

```
xrt-smi examine --report platform
```

3.24 AI Analyzer

AMD AI Analyzer is a tool that supports analysis and visualization of model compilation and inference on Ryzen AI. The primary goal of the tool is to help you better understand how the models are processed by the hardware, and to identify performance bottlenecks that may be present during model inference. Using AI Analyzer, you can visualize graph and operator partitions between the NPU and CPU.

3.24.1 Installation

If you installed the Ryzen AI software using automatic installer, AI Analyzer is already installed in the conda environment.

If you manually installed the software, you need to install the AI Analyzer wheel file in your environment.

```
python -m pip install %RYZEN_AI_INSTALLATION_PATH%\aianalyzer-<version>.
↪whl
```

> **Note**
>
> AI Analyzer is currently supports BF16 Models only. INT8 model analysis is not supported in this version, but expanded support is planned for a future release.

3.24.2 Enabling Profiling and Visualization

Profiling and Visualization can be enabled by passing additional provider options to the ONNXRuntime Inference Session. Here is an example:

```
provider_options = [{
             'config_file': 'vaip_config.json',
             'cacheDir': str(cache_dir),
             'cacheKey': 'modelcachekey',
             'ai_analyzer_visualization': True,
             'ai_analyzer_profiling': True,
         }]
session = ort.InferenceSession(model.SerializeToString(),␣
↪providers=providers,
                               provider_options=provider_options)
```

The `ai_analyzer_profiling` flag enables generation of artifacts related to the inference profile. The `ai_analyzer_visualization` flag enables generation of artifacts related to graph partitions and operator fusion. These artifacts are generated as JSON files in the current run directory.

AI Analyzer also supports native ONNX Runtime profiling, which you can use to analyze the parts of the session that run on the CPU. You can enable ONNX Runtime profiling through session options and pass it along with the provider options, as shown here:

```
 # Configure session options for profiling
 sess_options = ort.SessionOptions()
 sess_options.enable_profiling = True

 provider_options = [{
             'config_file': 'vaip_config.json',
             'cacheDir': str(cache_dir),
             'cacheKey': 'modelcachekey',
             'ai_analyzer_visualization': True,
             'ai_analyzer_profiling': True,
         }]

session = ort.InferenceSession(model.SerializeToString(), sess_options,␣
↪providers=providers,
                               provider_options=provider_options)
```

3.24.3 Launching AI Analyzer

After the artifacts are generated, *aianalyzer* can be invoked through the command line as follows:

```
aianalyzer <logdir> <additional options>
```

Positional Arguments

`logdir`: Path to the folder containing generated artifacts

Additional Options

`-v`, `--version`: Show the version info and exit.

`-b ADDR`, `--bind ADDR`: Hostname or IP address on which to listen, default is 'localhost'.

`-p PORT`, `--port PORT`: TCP port on which to listen, default is '8000'.

`-n`, `--no-browser`: Prevents the opening of the default url in the browser.

`-t TOKEN`, `--token TOKEN`: Token used for authenticating first-time connections to the server. The default is to generate a new, random token. Setting to an empty string disables authentication altogether, which is not recommended.

3.24.4 Features

AI Analyzer provides visibility into how your AI model is compiled and executed on Ryzen AI hardware. Its two main use cases are:

1. Analyzing how the model was partitioned and mapped onto Ryzen AI's CPU and NPU accelerator
2. Profiling model performance as it executes inferencing workloads

When launched, the AI Analyzer server scans the folder specified with the logdir argument and detect and load all files relevant to compilation and/or inferencing per the *ai_analyzer_visualization* and *ai_anlayzer_profiling* flags.

You can instruct the AI Analyzer server to either start a browser on the same host or return an URL that you can then load into a browser on any host.

3.24.5 User Interface

AI Analyzer has the following three sections as seen in the left-panel navigator:

1. PARTITIONING - A breakdown of your model was assigned to execute inference across CPU and NPU
2. NPU INSIGHTS - A detailed look at the how your model was optimized for inference execution on NPU
3. PERFORMANCE - A breakdown of inference execution through the model

These sections are described in more detail in the following sections:

PARTITIONING

This section is comprised of two pages: Summary and Graph

Summary

The Summary page gives an overview of how the models operators have been assigned to Ryzen's CPU and NPU along with charts capturing GigaOp (GOP) offloading by operator type .

There is also table titled "CPU Because" that shows the reasons why certain operators were not offloaded to the NPU.

Graph

The graph page shows an interactive diagram of the partitioned ONNX model, showing graphically how the layers are assigned to the Ryzen hardware.

Toolbar

- You can choose to show/hide individual NPU partitions, if any, with the **Filter by Partition** button
- You can show or hide a panel that displays properties for selected objects through the **Show Properties** toggle button
- You can show or hide the model table through the **Show Table** toggle button.
- Settings
  - Show Processor separates operators that run on CPU and NPU respectively
  - Show Partition separates operators running on the NPU by their respective NPU partition, if any
  - Show Instance Name displays the full hierarchical name for the operators in the ONNX model

All objects in the graph have properties that can be viewed to the right of the graph.

Model Table

This table following the graph lists all objects in the partitioned ONNX model:

- Processor (NPU or CPU)
- Function (Layer)
- Operator
- Ports
- NPU Partitions

NPU INSIGHTS

This section is comprised of three pages: Summary, Original Graph, and Optimized Graph.

Summary

The Summary page gives an overview of how your model was mapped to the AMD Ryzen NPU. Charts are displayed showing statistics on the number of operators and total GMACs that have been mapped to the NPU (and if necessary, back to CPU via the *Failsafe CPU* mechanism). The statistics are shown per operator type and NPU partition.

Original Graph

This is an interactive graph representing your model, lowered to supported NPU primitive operators and divided into partitions if necessary. As with the PARTITIONING graph, a companion table lists all model elements and supports cross-probing with the graph view. The objects in both the graph and the table also cross-probe with the PARTITIONING graph.

Toolbar

You can choose to show/hide individual NPU partitions, if any, with the **Filter by Partition** button A panel that displays properties for selected objects can be shown or hidden using the **Show Properties** toggle button A code viewer showing the MLIR source code with cross-probing can be shown/hidden through the **Show Code View** button The following table can be shown and hidden using the **Show Table** toggle button. Display options for the graph can be accessed with the **Settings** button

Optimized Graph

This page shows the final model that is mapped to the NPU after all transformations and optimizations such as fusion and chaining. It also reports the operators that had to be moved back to the CPU through the *Failsafe CPU* mechanism. As usual, there is a companion table below that contains all of the graph's elements, and cross-selection is supported to and from the PARTITIONING graph and the Original Graph.

Toolbar

You can choose to show/hide individual NPU partitions, if any, with the **Filter by Partition** button A panel that displays properties for selected objects can be shown or hidden using the **Show Properties** toggle button The following table can be shown and hidden using the **Show Table** toggle button. Display options for the graph can be accessed with the **Settings** button

PERFORMANCE

Use this section to view the performance of your model on RyzenAI when running one or more inferences. It is comprised of two pages: Summary and Timeline.

Summary

The performance summary page displays several overall statistics for the inference(s), along with charts that break down operator runtime by operator. When the ONNX Runtime profiler is enabled,

the total inference time, including layers executed on the CPU, is shown. When NPU profiling is enabled using the *ai_analyzer_profiling* flag, additional NPU-specific statistics are displayed, including GOP and MAC efficiency, as well as a chart showing runtime per NPU operator type.

The clock frequency field shows the assumed NPU clock frequency, but it is editable. When the frequency is changed, all timestamp data—collected as clock cycles but displayed in time units—is adjusted accordingly.

Timeline

The Performance timeline shows a layer-by-layer breakdown of your model's execution. The upper section is a graphical depiction of layer execution across a timeline, while the lower section shows the same information in tabular format. It is important to note that the Timeline page shows one inference at a time, so if you have captured profiling data for two or more inferences, you can choose which one to display with the **Inferences** chooser.

Within each inference, you can examine the overall model execution or the detailed NPU execution data by using the **Partition** chooser.

Toolbar

A panel that displays properties for selected objects can be shown or hidden using the **Show Properties** toggle button The following table can be shown and hidden using the **Show Table** toggle button. The graphical timeline can be downloaded to SVG using the **Export to SVG** button

3.25 Stable Diffusion Demo

Ryzen AI 1.7.1 provides preview demos of Stable Diffusion image-generation pipelines. The demos cover Image-to-Image and Text-to-Image using SD1.5, SDXL-base-1.0, Segmind-Vega, SD-Turbo (bs1), SDXL-Turbo (bs1), SD3.0 and SD3.5. These models are available for public download from Hugging Face.

3.25.1 Installation Steps

1. Ensure the latest version of Ryzen AI and NPU drivers are installed. See *Installation Instructions*.
2. The GenAI-SD folder is located in the RyzenAI installation tree. Navigate to the folder and run the following command:

```
cd "C:\Program Files\RyzenAI\1.7.1\GenAI-SD"
```

3. Activate the Conda environment for the Stable Diffusion demo packages:

```
conda activate ryzen-ai-1.7.1
```

4. The following Stable Diffusion models will be auto-downloaded from Hugging Face when running for the first time and cached locally in the `GenAI-SD\models` folder:

- SD1.5
- SD-Turbo (bs1)
- SDXL-Turbo (bs1)
- SDXL-base-1.0
- Segmind-Vega
- SD3.0 / SD3.0-ControlNet(Canny) / SD3.0-ControlNet(Pose) / SD3.0-ControlNet(Tile) / SD3.0-ControlNet(Depth)
- SD3.5

3.25.2 Running the Demos

Activate the conda environment:

```
conda activate ryzen-ai-1.7.1
```

Optionally, set the NPU to high performance mode to maximize performance:

```
xrt-smi configure --pmode performance
```

Refer to the documentation on *xrt-smi configure* for additional information.

Image-to-Image with ControlNet

The image-to-image demo generates images based on a prompt and a control image using ControlNet (for example, Canny, pose, tile, or depth). This demo supports SD3.0 and uses 512x512 as the default resolution, which can be overridden (for example, to 1024x1024) via the `-W/-H` options in the CLI examples below.

To run the demo, navigate to the `GenAI-SD\test` directory and run the following command:

```
python run_sd3.py -C canny --model_id "stabilityai/stable-diffusion-3-
↪medium-amdnpu"
```

The demo script uses a predefined prompt and `.\ref\canny.jpg` as the control image. The output image and control image are saved in the `generated_images` folder. You can redirect the output directory to a location where your user account has write permissions, for example: `--output_path C:\Users\<username>\Documents\generated_images`.

The control image can be modified and custom prompts can be provided with the `--prompt` option. For instance:

```
python run_sd3.py -C canny --model_id "stabilityai/stable-diffusion-3-
↪medium-amdnpu" --prompt "Anime style illustration of a girl wearing a␣
```

(continues on next page)

(continued from previous page)

```
↪suit. A moon in sky. In the background we see a big rain approaching.␣
↪text 'InstantX' on image" -n 50
```

The application of ControlNet can be configured with the `-C` option. For instance:

```
python run_sd3.py -C canny --model_id "stabilityai/stable-diffusion-3-
↪medium-amdnpu" --prompt "Anime style illustration of a girl wearing a␣
↪suit. A moon in sky. In the background we see a big rain approaching.␣
↪text 'InstantX' on image" -H 1024 -W 1024 --control_image_path .\ref\
↪canny.jpg -n 50
python run_sd3.py -C pose --model_id "stabilityai/stable-diffusion-3-
↪medium-amdnpu" --prompt "Anime style illustration of a girl wearing a␣
↪suit. A moon in sky. In the background we see a big rain approaching.␣
↪text 'InstantX' on image" -H 1024 -W 1024 --control_image_path .\ref\
↪pose.jpg -n 50
python run_sd3.py -C tile --model_id "stabilityai/stable-diffusion-3-
↪medium-amdnpu" --prompt "Anime style illustration of a girl wearing a␣
↪suit. A moon in sky. In the background we see a big rain approaching.␣
↪text 'InstantX' on image" -H 1024 -W 1024 --control_image_path .\ref\
↪tile.jpg -n 50
python run_sd3.py -C depth --model_id "stabilityai/stable-diffusion-3-
↪medium-amdnpu" -H 1024 -W 1024 --control_image_path .\assets\depth.jpeg␣
↪-n 50
```

To run the image-to-image demo of Segmind-Vega model (without ControlNet applications), run the following command:

```
python .\run_sd_xl.py --model_id "amd/segmind-vega-amdnpu" --control_
↪image_path .\assets\controlimg_input_1024x1024.png --strength 0.95
```

Text-to-Image

The text-to-image demo generates images based on text prompts. It supports SD 1.5 (512x512), SDXL-base (1024x1024), SD-Turbo (512x512), SDXL-Turbo (512x512), Segmind-Vega (1024x1024), as well as SD 3.0 and SD 3.5 (see below for additional setup required for SD3.x models).

To run the demo, navigate to the `GenAI-SD\test` directory and run the following commands to run with each of the supported models:

```
python run_sd.py    --model_id "amd/stable-diffusion-1.5-amdnpu"
python run_sd.py    --model_id "amd/sd-turbo-amdnpu"
python run_sd_xl.py --model_id "amd/sdxl-turbo-amdnpu"
```

(continues on next page)

(continued from previous page)

```
python run_sd_xl.py --model_id "amd/sdxl-base-amdnpu"
python run_sd_xl.py --model_id "amd/segmind-vega-amdnpu"
```

To run the sd3/sd3.5 models, you need to set the `DD_PLUGINS_ROOT` environment variable before running the demo. For instance:

```
set "DD_PLUGINS_ROOT=C:\Program Files\RyzenAI\1.7.1\GenAI-SD\lib\
↪transaction\stx\"
```

Then run the following commands:

```
python .\run_sd3.py -C None --model_id "stabilityai/stable-diffusion-3-
↪medium-amdnpu" -n 50
python .\run_sd3.py -C None --model_id "stabilityai/stable-diffusion-3.5-
↪medium-amdnpu" -n 50
```

The demo script uses a predefined prompt for each of the models. The output images are saved in the `generated_images` folder.

Custom prompts can be provided with the `--prompt` option. For instance:

```
python run_sd.py --model_id "amd/stable-diffusion-1.5-amdnpu" --prompt
↪"Photo of a ultra realistic sailing ship, dramatic light, pale sunrise,␣
↪cinematic lighting, battered, low angle, trending on artstation, 4k,␣
↪hyper realistic, focused, extreme details"
```

3.25.3 Running with AMD Stable Diffusion Sandbox

AMD SD Sandbox is a framework for running Stable Diffusion (SD) models accelerated by AMD Ryzen AI hardware. It provides an easy-to-use interface for evaluating, comparing, and deploying multiple SD pipelines. Please go to the AMD SD Sandbox GitHub repository for more information.

3.26 Ryzen AI CVML library

The Ryzen AI CVML libraries build on top of the Ryzen AI drivers and execution infrastructure to provide powerful AI capabilities to C++ applications without having to worry about training specific AI models and integrating them to the Ryzen AI framework.

Each Ryzen AI CVML library feature offers a simple C++ application programming interface (API) that can be easily incorporated into existing applications. The following AI features are currently available,

- **Depth Estimation**: Generates a depth map to assess relative distances within a two-dimensional image.

- **Face Detection**: Identifies and locates faces within an image.
- **Face Mesh**: Constructs a mesh overlay of landmarks for a specified facial image.

The Ryzen AI CVML library is distributed through the RyzenAI-SW Github repository: https://github.com/amd/RyzenAI-SW/tree/main/Ryzen-AI-CVML-Library

3.26.1 Building sample applications

This section describes the steps to build Ryzen AI CVML library sample applications. Before starting, ensure that the following prerequisites are available in the build environment,

- CMake, version 3.18 or newer
- C++ compilation toolchain. On Windows, this may be Visual Studio's "Desktop development with C++" build tools, or a comparable C++ toolchain
- OpenCV, version 4.11 or newer

Navigate to the folder containing Ryzen AI samples

Download the Ryzen AI CVML sources, and go to the 'samples' sub-folder of the library.

On Windows,

```
git clone https://github.com/amd/RyzenAI-SW.git -b main --depth-1
cd RyzenAI-SW\Ryzen-AI-CVML-Library\samples
```

On Linux,

```
git clone https://github.com/amd/RyzenAI-SW.git -b main --depth-1
cd RyzenAI-SW/Ryzen-AI-CVML-Library/samples
```

OpenCV libraries

Ryzen AI CVML library samples make use of OpenCV, so set an environment variable to let the build scripts know where to find OpenCV.

On Windows,

```
set OPENCV_INSTALL_ROOT=<location of OpenCV libraries>
```

On Linux,

```
export OPENCV_INSTALL_ROOT=<location of OpenCV libraries>
```

Build Instructions

Create a build folder and use CMAKE to build the sample(s).

On Windows,

```
mkdir build
cmake -S %CD% -B %CD%\build -DOPENCV_INSTALL_ROOT=%OPENCV_INSTALL_ROOT%
cmake --build %CD%\build --config Release
```

On Linux,

```
mkdir build
cmake -S $PWD -B $PWD/build -DOPENCV_INSTALL_ROOT=$OPENCV_INSTALL_ROOT
cmake --build $PWD/build --config Release
```

The compiled sample application(s) will be placed in the various build<application>Release folder(s) under the 'samples' folder (or build/<application>/Release for Linux).

3.26.2 Running sample applications

This section describes how to execute Ryzen AI CVML library sample applications.

Update the console and/or system PATH

Ryzen AI CVML library applications need to be able to find the library files.

On Windows, update the PATH environment variable for both the Ryzen AI CVML library location and OpenCV

```
set PATH=%PATH%;<location of Ryzen AI CVML library package>\windows
set PATH=%PATH%;%OPENCV_INSTALL_ROOT%\x64\vc16\bin
```

On Linux, update LD_LIBRARY_PATH for the Ryzen AI CVML library location, OpenCV library location and NPU driver location,

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<location of Ryzen AI CVML␣
↪library package>/linux
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OPENCV_INSTALL_ROOT/lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/xilinx/xrt/lib
```

Adjust the aforementioned commands to match the actual location of Ryzen AI and OpenCV libraries, respectively.

Select an input source (image or video)

Ryzen AI CVML library samples can accept a variety of image and video input formats, or even open the default camera on the system if "0" is specified as an input.

In this example, a publicly available video file is used for the application's input. The following command downloads a video file and saves it locally as 'dancing.mp4'

```
curl -o dancing.mp4 https://videos.pexels.com/video-files/4540332/4540332-
↪hd_1920_1080_25fps.mp4
```

Execute the sample application

Finally, the previously built sample application may be executed with the selected input source.

On Windows,

```
build\cvml-sample-depth-estimation\Release\cvml-sample-depth-estimation.
↪exe -i dancing.mp4
```

On Linux,

```
build/cvml-sample-depth-estimation/Release/cvml-sample-depth-estimation.
↪exe -i dancing.mp4
```

3.27 Supported Operators

The following table lists the ONNX operators supported in Ryzen AI 1.7.0. The operators are categorized by their support for different quantization types: BF16, A16W8, A8W8, and XINT8. A "Y" indicates that Ryzen AI provides broad coverage for that operator and for that specific quantization type for CNN and NLP models. Some specific configurations of that operator may however not be fully supported.

Ops	BF16	A16W8	A8W8	XINT8
Abs	Y			Y
Add	Y	Y	Y	Y
And	Y			Y
ArgMax	Y			Y
ArgMin	Y			Y
AveragePool	Y	Y	Y	Y
BatchNormalization	Y	Y	Y	Y
BitShift	Y			Y
BitwiseAnd	Y			Y
BitwiseNot	Y			Y

continues on next page

Table 8 – continued from previous page

Ops	BF16	A16W8	A8W8	XINT8
BitwiseOr	Y			Y
BitwiseXor	Y			Y
Cast	Y			Y
Ceil	Y			Y
Celu				Y
Clip		Y	Y	Y
Concat	Y	Y	Y	Y
Constant	Y	Y	Y	Y
ConstantOfShape	Y	Y	Y	Y
Conv	Y	Y	Y	Y
ConvTranspose	Y	Y	Y	
Cos		Y	Y	
CumSum	Y			Y
DepthToSpace	Y	Y	Y	Y
DequantizeLinear		Y	Y	Y
Div	Y	Y	Y	Y
Einsum	Y			
Elu		Y		Y
Equal	Y			Y
Erf	Y			Y
Exp	Y	Y		
Expand	Y	Y	Y	Y
Flatten	Y	Y	Y	Y
Floor	Y			Y
Gather	Y			
GatherElements	Y	Y		
Gelu		Y	Y	Y
Gemm	Y	Y	Y	Y
GlobalAveragePool	Y	Y	Y	Y
GlobalMaxPool		Y	Y	Y
Greater	Y			Y
GreaterOrEqual				Y
GridSample	Y			
GroupConv	Y			
GroupNormalization		Y	Y	Y
HardSigmoid			Y	
HardSwish			Y	Y
Identity	Y	Y	Y	Y
InstanceNormalization	Y	Y	Y	Y
LSTM	Y			
LayerNormalization				Y

continues on next page

Table 8 – continued from previous page

Ops	BF16	A16W8	A8W8	XINT8
LeakyRelu		Y	Y	Y
Less	Y			Y
LessOrEqual				Y
Log	Y			Y
MatMul	Y	Y	Y	Y
Max	Y	Y	Y	Y
MaxPool	Y	Y	Y	Y
Min	Y	Y	Y	Y
Mish				Y
Mod	Y			
Mul	Y	Y	Y	Y
Neg	Y			Y
Not	Y			
Or	Y			Y
Pad	Y	Y	Y	Y
Pow	Y			Y
QLinearConv		Y	Y	
QLinearMatMul			Y	
QuantizeLinear		Y	Y	
Range				Y
Reciprocal	Y	Y	Y	Y
ReduceMax	Y	Y	Y	Y
ReduceMean	Y		Y	Y
ReduceMin	Y		Y	Y
ReduceSum	Y		Y	Y
Relu		Y	Y	Y
Reshape	Y	Y	Y	Y
Resize	Y		Y	Y
Round	Y			Y
STFT				Y
ScatterND	Y			
Shape	Y	Y	Y	Y
Shrink				Y
Sigmoid	Y	Y	Y	
Sign	Y			Y
Sin	Y	Y	Y	
Size	Y			
Slice	Y	Y	Y	Y
Softmax		Y	Y	Y
Softsign				Y
Split	Y	Y		

continues on next page

Table 8 – continued from previous page

Ops	BF16	A16W8	A8W8	XINT8
Sqrt	Y	Y	Y	Y
Squeeze	Y	Y	Y	Y
Sub	Y	Y	Y	Y
Tanh	Y	Y		Y
ThresholdedRelu				Y
Tile	Y			Y
TopK		Y		Y
Transpose	Y	Y	Y	Y
Unsqueeze	Y	Y	Y	Y
Upsample		Y	Y	Y
Upsample (deprecated)	Y			
Where	Y	Y		Y
Xor	Y			

3.27.1 LLM Operator support

The Ryzen AI LLM execution flow supports ONNX Runtime GenAI-based models with the following operators:

- SimplifiedLayerNormalization
- SkipSimplifiedLayerNormalization
- MatMulNBits (W4ABF16/W4ABFP16)
- Add
- RotaryEmbedding
- GroupQueryAttention
- Sigmoid
- Mul

These operators are optimized for execution on the Ryzen AI NPU via the ONNX Runtime GenAI framework.

3.28 Licensing Information

Ryzen AI is released by Advanced Micro Devices, Inc. (AMD) and is subject to the licensing terms listed below. Some components may include third-party software that is subject to additional licenses. Review the following links for more information:

- [AMD End User License Agreement](https://account.amd.com/content/dam/account/en/licenses/download/amd-end-user-license-agreement.pdf)

- [Windows - Third Party End User License Agreement](https://account.amd.com/content/dam/account/en/licenses/download/ryzen-ai-1.7.1-ga-tpn-license.pdf)