Evaluation of NPU Acceleration for Offloading Software 5G Workloads

Giovanni Ranieri[1]

[1] School of Computer and Communication Sciences, EPFL

Abstract :

Neural Processing Units are recent hardware-specialized chips. Primarily composed of AI Engines and commonly incorporated in modern microprocessors, they serve as fast compute units for specific applications like Digital Signal Processing and Machine Learning. In this report, we explore the possibility to offload software 5G workloads into a NPU with XDNA AIE-ML architecture from AMD/Xilinx using MLIR-AIE, an open-source toolkit that targets AI Engines in both Versal and Ryzen AI products. We focus on offloading Discrete Fourier Transforms from srsRAN, an open-source 5G stack, and run its benchmarks to compete with the CPU implementation. We discuss the results and the limitations we encountered. In summary, we highlight (i) how MLIR-AIE enables developers to efficiently program the AI Engines, (ii) encountered issues while programming in this architecture and toolchain, and (iii) the necessary documentation for new researchers who would like to dig into the software paradigm of AI Engines.

Index Terms: MLIR-AIE, AI Engines, 5G Workloads, Neural Processing Units.

1 Introduction

Neural Processing Units (NPUs) have become the source of many papers concerning their efficiency for machine learning (ML) and digital signal processing (DSP) applications (Rösti et al., 2025 and Taka et al., 2023). As they require intensive computations, optimizing workloads becomes a crucial task in modern engineering, especially for reducing the energy consumption. These specialized microarchitectures-based chips are present in many devices, from mobile phones to servers, allowing developers to accelerate a wide range of applications, such as in 5G wireless or in HPC, presented in Brown et al., 2025. In 2023, AMD/Xilinx unveiled their new generation of AI Engines architecture: XDNA AIE-ML, driven towards ML workloads AMD/Xilinx, 2023a. This pushes developers to offload workloads on them, requiring a new set of skills such as understanding how the architecture works, which software tools to use, and how to program intelligently.

In this report, we used a computer equipped with a NPU with XDNA AIE-ML, integrated in a Ryzen 7 8845HS microprocessor (Ryzen AI). We run Discrete Fourier Transforms (DFTs) directly on the NPU. In addition, it enabled us to run a DFT benchmark of srsRAN, an open-source and well-supported 5G stack, comparing their CPU-based implementation with our NPU-based implementation. The main limitation we encountered is **the overhead** of the NPU's runtime library XRT when executing a DFT. The CPU-based implementation of srsRAN runs DFTs sequentially, one at a time. Doing the same with our NPU-based implementation, this overhead is introduced in every DFT, making our implementation slower by a factor or 10. Nevertheless, in addition to the solution we proposed for that, we present the steps to execute these DFTs with (i) a background summary of the XDNA AIE-ML architecture, (ii) a detailed explanation on how to use MLIR-AIE as software tool, (iii) how the DFTs were implemented and optimized and (iv) the summary of documentation with limitations encountered.

2 Background

2.1 XDNA Architecture

Ryzen AI and Versal Adaptive SoC are two platforms that integrate NPUs. While the first one is a brand of microprocessors, and the second a general software-programmable and heterogeneous compute platform, both of them can incorporate a NPU, and more specifically AI Engines. These small units are the core part of the different NPU architectures, forming an array of compute units and capable of sharing data. Microprocessors such as Ryzen AI incorporate a small amount of these AI Engines, while the second can contain hundreds of them Taka et al., 2023. Figure [1] shows a spatial architecture AMD/Xilinx, 2024, representing the XDNA. It's a high-performance architecture that uses control and data flow graphs as the computational model. It contains two different entities: Compute Tiles (CTs) and Interface/Shim Tiles (STs). This architecture reflects a flow graph, where data moves inside it in a synchronous manner. Each CT has its own vector processor unit (VPU) and an internal memory space, allowing rapid accumulation of results in the processor's registers. STs are gateways with the external memory, optimizing the transfer of data inside and outside the NPU.

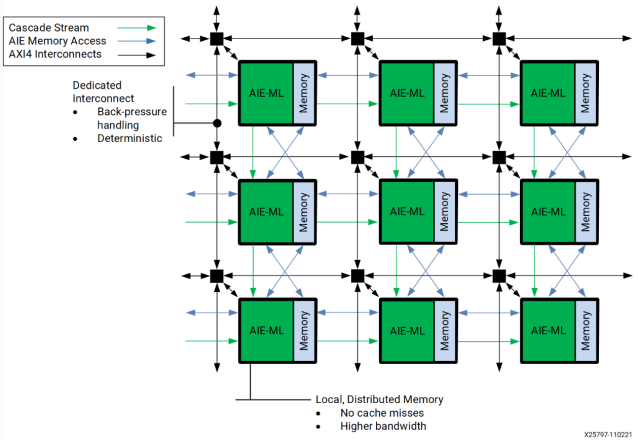

Figure 1. XDNA spatial architecture, where CTs (with the AIE-ML type of AI Engine) are shown interconnected by different communication systems.

The XDNA architecture has two forms: XDNA1 and XDNA2. The first one has again two derivations: AIE and AIE-ML, while the second comes with the code name AIE2P. The hardware changes between them but its not the purpose of this report, but keep in mind that challenges arise when it comes to software programming.

Some features can be implemented in one tool, and some not for the different derivations.

2.2 XDNA1/AIE-ML Hardware Specifications

To efficiently program our NPU, we present some important low-level specifications. AIE-ML has been developed for ML applications. Low-precision (and vector) operations, with fast data transfer across tiles are just a few examples. Memory Tiles (MTs) are introduced and increases on-chip memory capacity. In Figure [2], the importance on integer arithmetic is shown AMD/Xilinx, 2024, where the only floating point data type available is the bfloat16, a low-precision (8b mantissa) type represented on 16b. The VPU has a very-long instruction word (VLIW) architecture and allows multiple instructions at the same time (SIMD). In terms of data transfer, the spatial architecture uses two main ways for communication: (i) local memories, and (ii) AXI4-Stream (a protocol for unidirectional transfer of data). Local memories provide the most efficient data-sharing path during processing. The CT has access to the four nearest memory modules in the cardinal directions: north, south, east (local data memory in the tile itself) and west, allowing a program to use up to 4 x 64KB of local memory (close to the AI Engines). Each local memory is divided into 8 memory banks, and compilers will choose where data is stored, reducing locks when AI Engines wants to access specific data.

Table: Supported Precision Width of the Vector Data Path

Precision 1	Precision 2	Number of Accumulator Lanes	Bits per Accumulator Lane	Number of MACs
int 8	int 4	32	32	512
int 8	int 8	32	32	256
int 16	int 8	32	32	128
int 16	int 8	16	64	128
int 16	int 16	32	32	64
int 16	int 16	16	64	64
int 32 [1]	int 16	16	64	32
cint 16	cint 16	8	64	16
cint 32	cint 16	8	64	8
bfloat 16 [3]	bfloat 16	16	SPFP 32 [2]	128

Notes:

1. int32 x int32 can be emulated. The operation should have half the performance of int32 x int16 and there should be 16 multiplications per cycle.
2. Single precision floating point (SPFP) per the IEEE standard.
3. float32 x float32 can be emulated. Emulation deviates from the IEEE-754 standard. See Answer Record 34376 for more information.

Figure 2. Table summarizing the different precision width of vector data paths for AIE-ML architecture. Bfloat16 is the only floating point data type available.

The second method uses DMAs (Data Movement Accelerators), allowing transfers to any tile in the spatial architecture. What's important for a programming point of view is its a 32bits/cycle/stream transfer. A fixed number of what we call **channels** are available, depending on the tile type. Its the number of roadways in and out the tile. For example, CTs have two in and two out DMA channels.

3 Software Programming

In this section, we start our discussion on the software stack required to program and execute code on the NPU.

3.1 Introduction

NPUs are not designed for general purpose software, but for optimized code running continuously. The software methodology where you abstract the functionality of a program does not hold. Application specific programs are the target of these chips. The software architecture is composed of 3 main blocks: the host code, the AI Engine low-level configuration and the kernels.

3.1.1 Host Code The host code is the code used at runtime that start workloads on the NPU. In the context of AMD/Xilinx NPUs, it uses a binary (.xcblin) file that contains everything useful to run one or multiple computations.

3.1.2 Kernels The kernel is the actual code running on the NPU. It's a function call (potentially calling other functions internally) with arguments that respect the low-level configuration of the tiles. As explained before, kernels must be optimized to leverage the NPU's architectural features. Kernels are written in C/C++.

3.1.3 Low-level Configuration To run kernels, you need to write how the data moves to and inside the NPU, what kernel runs on each tile, among other configurations. A way of doing so is to define an Intermediate Representation (IR). A tool used by frameworks to generate these IRs is MLIR (Multi-Level Intermediate Representation). From Lattner et al., 2020, it "aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together". Creating these representations is not an easy task. Fortunately, other projects try to use high-level programming languages to create these MLIRs and use them for programming AI Engines, like MLIR-AIE where Figure [3] shows a premise[1].

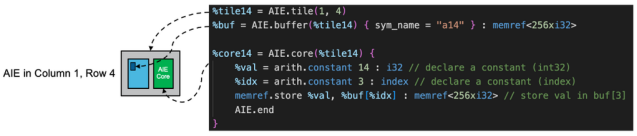

Figure 3. Example of MLIR in MLIR-AIE. We define what CT to use, allocate a buffer in the local memory and declare what's going on inside the AI Engine.

3.2 Software Tools

Multiple tools are available, and matching their requirements (programming languages, OS-specific considerations) with yours is an important step. The different stacks available are summarized in Figure [4], each developed by AMD/Xilinx as they target their AI Engines.

1. **Riallto**: It's an open-source project that aims to provide a user-friendly stack for programming AI Engines in Ryzen AI microprocessors.
2. **MLIR-AIE**: It's a MLIR-based toolchain, open-source, and provides (i) a user-friendly experience for programming AI Engines (on Ryzen AI and Versal Adaptive SoC), but also (ii) deeper tutorials and functionalities to create more advanced kernels.
3. **Ryzen AI Software**: This stack is closed-source, and it provides runtime libraries for optimizing and deploying AI infer-

[1] https://github.com/Xilinx/mlir-aie/blob/main/mlir_tutorials/tutorial-1/README.md

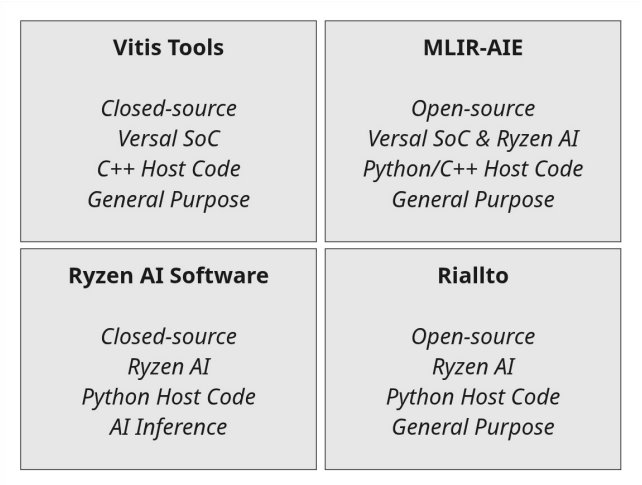

Figure 4. Summary of AMD/Xilinx Tools for XDNA NPU developement. Each one highlight the source restriction, the platform targeted, host code language support and purpose.

ence on AMD Ryzen AI based PC, great for performance and stability.

4. **Vitis Tools**: Combining a compiler (xchesscc) and multi-core AIE design tool (aiecompiler), this closed-source project is perfect for optimized kernels in DSP, but only available for Versal Adaptive SoC, not Ryzen AI.

We work with a Ryzen AI NPU (with AIE-ML architecture), making Vitis Tools useless. We selected MLIR-AIE because C++ host code was a **strict requirement** for us, as srsRAN's stack is built on C++. To efficiently write the low-level configurations, they provide an API called IRON Hunhoff et al., 2025. It offers a very high-level interface for developers. In addition, a closer-to-metal (that we will denote by "CTM") API is available, which does the same job but the syntax is different, and gives to developers a better undestanding of what's going on in the hardware (enabling tracing for example).

As explain by one of the maintainers: "The purpose of this repository and LLVM-AIE is to provide open-source toolkits that target AI Engines in both Versal and Ryzen AI products. This enables enthusiasts and researchers to build their own toolchains for AIEs using MLIR or LLVM. Additionally this repository contains the IRON API and Python libraries for building applications targeting Ryzen AI NPUs. IRON's goal is to provide a close-to-metal interface for fast and efficient programming, this is a lower level of abstraction compared to the ONNX/PyTorch AI inference interfaces in the Ryzen AI Software package mentioned above."[2]

3.3 MLIR-AIE

MLIR-AIE is primarily based on LLVM-AIE (also known as Peano). Its a fork of LLVM, a toolkit for the construction of highly optimized compilers, for targeting AIE architectures. MLIR-AIE is also a toolchain that creates MLIRs, lowering them until an optimized IR for AI Engines. The main utility tool is denoted by aiecc.py (you can search it in the code base). Its main job is to create compact terminal commands for generating ELF (Executable & Linking Format), xclbin and instruction files for executing kernels. The parameters are summarized[3] in Figure [5], and Figure [6] summarizes the steps done by the toolchain. The host code is compiled and linked to create an executable. The kernel is compiled using either Peano or xchesscc, the low-level configuration of the NPU is converted in MLIR, and at the end both of them are combined to create a XRT binary file and instructions for the NPU using aiecc.py utility tool.

Optional arguments	Description
--sysroot sysroot	sysroot for cross-compilation
-v	Trace commands as they are executed
--vectorize	Enable MLIR vectorization
--xbridge	Link using xbridge (default)
--xchesscc	Compile using xchesscc (default)
--compile	Enable compiling of AIE code (default)
--no-compile	Disable compiling of AIE code
--host-target HOST_TARGET	Target architecture of the host program (e.g. vck190 uses aarch64-linux-gnu)
--compile-host	Enable compiling of the host program (default)
--no-compile-host	Disable compiling of the host program
--link	Enable linking of AIE code (default)
--no-link	Disable linking of AIE code
-j NTHREADS	Compile with max n-threads in the machine (default is 1). An argument of zero corresponds to the maximum number of threads on the machine.
--profile	Profile commands to find the most expensive executions.
--unified	Compile all cores together in a single process (default)
--no-unified	Compile cores independently in separate processes
-n	Disable actually executing any commands.

Figure 5. Table taken from MLIR-AIE summarizing the different compilation options with aiecc.py, enabling developers to select for example the right compiler.

3.3.1 XRT

XRT is a combination of user-space and kernel driver components. Because AI Engines are a new architecture, drivers are not implemented directly in XRT. A project called xdna-driver[4] adds this support. Following its documentation for Native APIs, we need to load the binary, instructions file and create an object holding the targeted device (the NPU in this case). After that we need to create buffer objects capable of sharing the data from our external memory to the NPU. A group ID is assigned to each buffer, and represents in a high-level way the memory bank mentioned before. When using MLIR-AIE, **group IDs 0, 1 and 2 are reserved** for optimization purpose: for the operation code, the instructions buffer and the instructions size (in kB). The bank into which a buffer falls only has performance implications, no functional implications.

3.3.2 AIE-API Intrinsics

MLIR-AIE uses in addition an API for optimizing kernels. AIE-API is a portable programming interface for AIE accelerators. It is implemented as a C++ header-only library that provides types and operations that get translated into efficient low-level intrinsics. The API also provides higher-level abstractions such as iterators and multi-dimensional arrays.

Intrinsics are well-optimized functions defined in compilers, avoiding automatically generated instructions that sometimes are not the one the more optimized. The API is well-documented, but

[2] `https://github.com/Xilinx/mlir-aie/issues/2336`

[3] `https://github.com/Xilinx/mlir-aie/blob/main/mlir_tutorials/tutorial-10/README.md`

[4] `https://github.com/amd/xdna-driver`

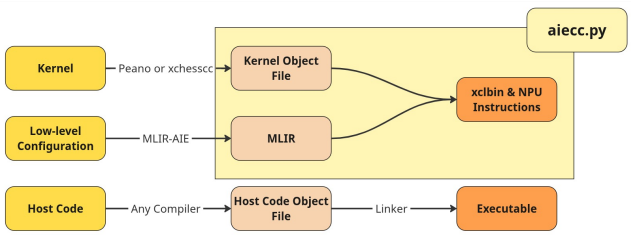

Figure 6. MLIR-AIE toolchain summary

caution to select the right version in the URL. Actually, the 2024-2 is the more stable. Loading and storing data for example from L1 memory into VPU registers can be optimized using this API. We didn't use the AIE-ML-API because at the moment of this project **DFTs where not implemented**.

4 Discrete Fourier Transforms on XDNA AIE-ML

The workloads we offload on the NPU are Discrete Fourier Transforms (DFTs). All limitations encountered are summarized in Table [1] and Figure [7] summarizes the overall structure we used. We mentioned earlier that two compilers were available. We used the xchesscc compiler and had to create an intermediate processing step to convert the output into readable instructions. Details of the graph are explained in the following subsections, and Appendix [A] improves the readability by showing the complete code.

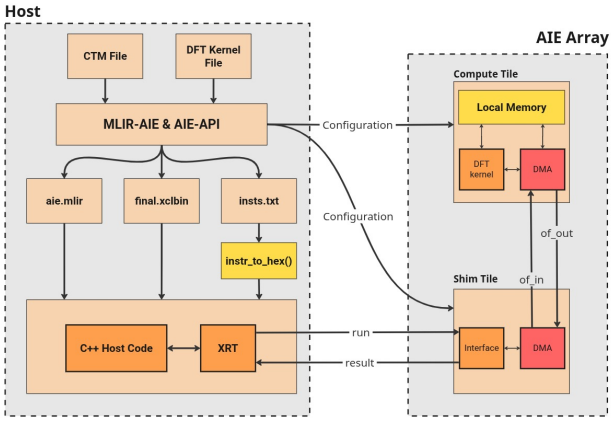

Figure 7. General overview of how MLIR-AIE and XRT have been used to run a DFT kernel. In the Host, the CTM & the kernel are processed by MLIR-AIE, resulting in multiple files. The C++ Host code using XRT synchronize the data to be processed by the kernel. The data is managed by two ObjectFIFOs *of_in* and *of_out*.

4.1 CTM Implementation

We mentioned the different communication systems inside the NPU. ObjectFIFOs abstract a first-in-first-out (FIFO) data sharing mechanism inside the NPU. An ObjectFIFO is composed by a depth, a producer tile, one of multiple consumer tiles and a data type. The producer writes data into the FIFO buffer, and consumers read from it. Each end can *acquire* an element to change it, before *releasing* it, such that others can use it. This mechanism allows **synchronization**, having now only one party accessing a specific object in the ObjectFIFO at any time, improving performance and reducing locks. An ObjectFIFO actually creates a buffer on each end, and it's these buffers that are filled when data from the CPU are synchronized with the NPU.

We created two ObjectFIFOs for the DFT kernel, one for sending the data and the other one for retrieving the results, each with bfloat16 type. If we denote $[(a_1, b_1), ..., (a_M, b_M)]$ the complex signal composed of tuples with real and imaginary parts, we literally send the data in that order. The runtime speed performance compared having two ObjectFIFOs for real and imaginary parts separately (and similarly in the NPU-to-CPU direction) drastically improved.

CTM has to describe a *runtime sequence* responsible of actually mapping/filling the host buffers with the ObjectFIFO's buffers. In the case of the input signal we have (Figure [8] adds the second buffer)

1. The host buffer is passed in the host XRT code as an XRT buffer object
2. The runtime sequence assigns an identifier *x_in* for the buffer, with the data type bfloat16.
3. The runtime sequence fills *x_in* buffer into the ObjectFIFO *of_in*. This happens sequentially (first-in, first-out), **without buffering on the ST**.
4. The ObjectFIFOs are configured to connect with a CT that will consume the data. The runtime sequence sequentially fills data from *x_in* into the ObjectFIFO, and on the CT, the core can receive values out of this pipe. The DMAs then typically take this data and do indeed put this into a buffer. The ObjectFIFO allocates this buffer.

The order of the XRT's buffers when running a kernel has to match the one in the CTM.

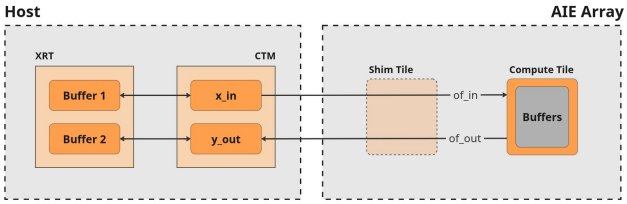

Figure 8. Links between XRT's buffers, runtime sequence and ObjectFIFOs buffers. Our two buffers on the host side are mapped using a runtime sequence, with a name, to the ObjectFIFOs buffers on the CT. The ST is "hidden" as it does not buffer data, just forwarding it.

4.2 Twiddles Generation & Radix-based FFT

Twiddle is another name for complex exponential. Developers have to evaluate these twiddles before invoking the AIE-API FFT routine. Because they do not depend on the input signal, synchronizing them using XRT is not ideal. Instead, we store them as static arrays in the CT's local memory, using a header file. The compiler allocates a limited amount of static heap space for such data. For $N = 512$, we require $N-1 = 511$ twiddles, representing around $2KB$ of memory, a small portion of the $64KB$ available per local memory.

AIE-API's FFT algorithm works with **stages**. Using radix-2

FFT, each stage splits a N-point DFT into two smaller $\frac{N}{2}$-point DFTs, storing sub-results in a temporary variable. We sequentially apply stages until a base case, following a divide-and-conquer pattern. For each stage $l \in \{\frac{N}{2}, \frac{N}{4}, ..., 1\}$, the twiddles form an array of $\frac{N}{2l}$ elements

$$\alpha_l[k] = e^{-2\pi j \frac{kl}{N}} \; k \in \{0, 1, ..., \frac{N}{2l} - 1\}. \tag{1}$$

When $N = 512$, we have 9 stages. We actually do not use the standard C library math.h but write directly their numerical values because (i) its faster than to call the exponential function and (ii) we need bfloat16 twiddles (the standard exponential function returns a double). We faced several limitations from the AIE-ML architecture ifself and the open-source Peano compiler part of LLVM-AIE. For the architecture, the lack of radix-3 and radix-5 implementation for AIE-ML architecture lets us compute N-point DFTs only for $N = 2^k$, for $k \in \mathbb{N}^*, k \geq 4$. For the compiler, it does not support FFT intrinsics of AIE-API for complex types, even if they are declared[5] (see Figure [10]). To bypass this issue, we requested Vitis Tools by a form on their website to access xchesscc, the closed-source compiler for AIE architectures supporting these intrinsics. It still can be used for Ryzen AI NPUs.

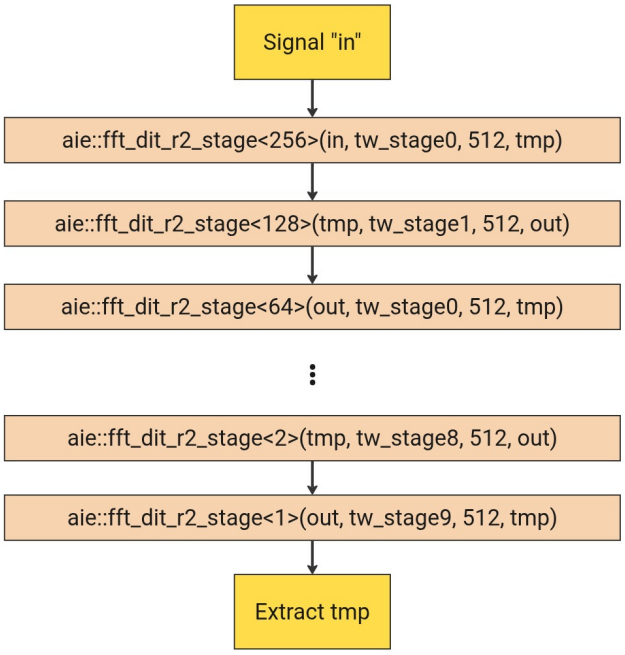

Figure 9. Functions breakdown for a radix-2 FFT using AIE-API where $N = 512$. We apply sequentially the stages on a signal "in" and extract at the end the "tmp" result.

4.3 Kernel & Optimization Steps

The term *tracing* refers for a powerful debugging and profiling technique that captures a detailed record of events happening within a system's hardware, such as memory accesses and bus transactions. MLIR-AIE enables tracing via XRT and takes advantage of the AIE architecture tracing system. The number of cycles a kernel takes to run has to be minimized, and several steps have been done for the DFT kernel.

[5] https://github.com/Xilinx/aie_api/issues/3

```
#if 0
INTRINSIC(v16cint16)
broadcast_to_v16cint16 (cint16 b) { return __builtin_aiev2_vbroadcast32_I512(b); }

INTRINSIC(v16cint16)
broadcast_to_v16cint16 (v2cint16 b) { return __builtin_aiev2_vbroadcast64_I512(b); }

INTRINSIC(v8cint32)
broadcast_to_v8cint32 (cint32 b) { return __builtin_aiev2_vbroadcast64_I512(b); }
#endif
```

Figure 10. LLVM-AIE source code (llvm-aie/clang/lib/Headers/aiev2_scl2vec.h) where we see the declaration of complex broadcast intrinsics, but commented.

Initially, the full DFT kernel required 23,985 cycles, compared to only 5,517 cycles for the FFT algorithm. We considerably decreased the cycles by optimizing loops, especially when creating the complex bfloat16 (cbfloat16) objects from the real and imaginary parts. As shown in Figure [11], we replaced a loop with a single pointer to create the *cbfloat16* objects required for the FFT. In addition, we declared every static arrays outside the kernel to initialize only one time the objects. These two optimizations reduced the number of cycles to 9600.

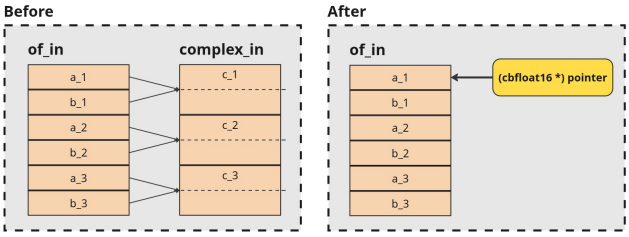

Figure 11. In memory before & after loop optimization. On the left, no optimization results in a loop to create a new array of complex bfloat16. The right block shows that we can just create a pointer in a single cycle to interprete the data in that memory region as complex bfloat16.

Another loop at the end of the kernel breaks the result of the FFT algorithm into real and imaginary components again such that XRT and MLIR-AIE can synchronize back the result to the host side. This loop takes 3600 cycles for $N = 512$, more than 30% of the 9600 cycles. Using directly the output of the last stage enabled us to remove that loop, reducing to around 6000 cycles.

4.4 Own radix-2 FFT?

The AIE-API is referred to as a single-tile API, as it provides functions for one tile only (for example the sequential stages to perform their FFT algorithm). It implies that we don't actually use the computational power of the NPU and its multiple AI Engines. To perform a DFT on multiple CTs, we manually implemented a simple radix-2 FFT for $N = 64$. Denoting the input signal by x, then the k-coefficient of the DFT can be separated into two terms Heideman et al., 1984

$$X_k = \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-2\pi j \frac{2mk}{N}}}_{E_k} + e^{-2\pi j \frac{k}{N}} \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-2\pi j \frac{2mk}{N}}}_{O_k}, \quad (2)$$

for $k = 0, ..., \frac{N}{2} - 1$. E_K is the DFT of the even-indexed samples and O_k the DFT of the odd-indexed samples. We can re-write now all coefficients with the formulas thanks to the periodicity of the complex exponentials

$$X_k = E_k + e^{-2\pi j \frac{k}{N}} O_k,$$
$$X_{k+\frac{N}{2}} = E_k - e^{-2\pi j \frac{k}{N}} O_k.$$

We first compute 32-point DFT on two CTs, and compute the final result in the last one, before synchronizing back the result. A ST is always here as gateway with the external memory.

4.5 srsRAN Benchmark

We modified the srsRAN 5G stack to benchmark DFT implementations.

1. Support for XRT and Eigen. The first one will execute the kernel and the second provides the bfloat16 type. srsRAN being with C++17, bfloat16 are only available in the C++23 standard libraries.
2. Implementation of a DFT processor for NPU side-by-side to the other implementations (CPU and FFTW library).

5 Results

The experiments ran on a GMKtec AMD Ryzen 7 Gaming Mini PC 8845HS K8 Plus, with an AMD Ryzen 7 8845HS microprocessor. The software stack was built using the XRT version 2.18.0 (2025-01-10), with support from the corresponding XDNA drivers (v2.18.0). Compilation of AI Engine kernels was handled by the xchesscc compiler (U-2023.06), part of the Vitis Tools. This setup ensures compatibility with the latest MLIR-AIE framework.

srsRAN implements benchmarks for various workloads, such as DFTs. They simply calculate the time for running one DFT, average the results, and compute the bandwidth in mega samples per second (Ms/s), with different quantiles, giving to developers a sense of average, best and worse bandwidth. With $N = 512$, the CPU implementation achieves **400 Ms/s** at the 90th percentile, whereas the NPU implementation reaches only **4 Ms/s** at that same percentile (and even worse with the own radix-2 implementation). We include a warm-up phase—launching the kernel repeatedly—to amortize pre-processing overheads (memory allocation, tracing startup, etc.).

Brown et al., 2025 in their results on offloading Fortran intrinsics on a Ryzen AI NPU highlights this overhead. AMD/Xilinx AMD/Xilinx, 2023b explains that "the overhead (of XRT) of dispatching the commands and arguments to the accelerator can be between 30μs and 60μs". To isolate overhead, we measured two empty kernels: one synchronizing DFT-sized data, and one synchronizing nothing. Both of them took around 45μs. Compared to the CPU implementation, one DFT takes around 3μs. This concludes that moving data to and from the NPU was not the issue (which in fact was taking around 600 nanoseconds). The impact of the overhead can be minimized by minimizing the number of calls for the kernel. One of the mainteners of MLIR-AIE writes that "there aren't any options to reduce the overhead to zero, but it can be "hidden" by overlapping useful work on the CPU while offloading work to an accelerator like the NPU. While the absolute overheads may be difference from accelerator to accelerator, there is always some cost in migrating work to another compute resource"[6].

Because srsRAN's benchmark runs sequentially one DFT after the other and compute its performance by averaging the time for each DFT, without any parallelism, the NPU kernel is drastically impacted when performing this task, always adding this overhead.

5.1 Documentation

Because AMD/Xilinx provides a huge set of documentation, coming with code names, we highlight in Table [2] some of them very useful while working with AI Engines and the different XDNA architectures.

6 Conclusion

In this project, we explored the feasibility and performance of offloading Discrete Fourier Transform workloads from a 5G software stack to a Neural Processing Unit (NPU) based on the XDNA AIE-ML architecture. Using the MLIR-AIE toolchain and the AIE-API, we developed and optimized a DFT kernel for a specific N, integrated it into the srsRAN benchmark suite, and analyzed the resulting performance.

While the NPU offered a programmable and potentially powerful architecture, our experiments showed that significant runtime overheads—mainly from the XRT runtime—drastically reduced its usefulness for small, sequential workloads such as single DFT computations. Despite extensive kernel-level optimizations and the use of complex vector intrinsics, the execution time per DFT was slower than the CPU-based implementation. It does not mean offloading workloads on a NPU is not useful, it just means we need another way of **measuring performance**, especially in huge stacks like srsRAN.

The project also served to highlight critical documentation and tooling gaps that future developers must be aware of when working with this architecture and software toolchain.

7 Report Notes, Observations & Acknowledgements

This research project has been very interesting from an engineering point of view. I learned a lot on hardware-specialized chips, how they are optimized, and how they can be used for offloading heavy computations, very important in DSP where bandwidth is crucial. I would like to thank Professor Al Hassanieh, Raphael Cannatà, Dan Dimitriu, from the SENS laboratory at EPFL, for the support and help from the very beginning to the end of the project.

Unfortunately, I saw that the hardware we bought for of-

[6] https://github.com/Xilinx/mlir-aie/issues/2381

Table 1
Summary of important limitations encountered while working on the XDNA AIE-ML architecture with MLIR-AIE, AIE-API and XRT.

Name	Description
Twiddles	Numerical values have to be computed prior to invocation of the AIE-API's FFT algorithm because of the strict requirement for bfloat16 data type.
Lack of some radix-based FFT	The lack of radix-3 and radix-5 implementation for AIE-ML architecture lets us compute N-point DFTs only for $N = 2^k$, for $k \in \mathbb{N}^*$, $k \geq 4$. Because srsRAN uses $N = 754$, advanced considerations have to be done to account that.
Floating point data type	XDNA AIE-ML architecture does not support float32, but only bfloat16, which has a lower precision. Float32 can be emulated in software, but slows down the computations.
Complex data type	LLVM-AIE does not support FFT intrinsics of AIE-API for complex types. The use of xchesscc coming with the Vitis Tools from AMD/Xilinx is mandatory.
XRT's kernel overhead	After experimentation, XRT has an overhead between 30 and 60μs when running a single kernel. The CPU takes around 3μs to run one DFT. Running one DFT at a time was necessary to benchmark against srsRAN, as they compute one DFT at a time. Several solutions are discussed in the Result section, involving OpenCL for example.
Trace in MLIR-AIE	Tracing is not implemented in IRON API. Any IRON file has to be converted in using CTM implementation.
xchesscc & Instructions file	When switching from Peano to xchesscc compiler, we highlight that the instructions file for the NPU is not hexadecimal, a requirement from XRT.
DMA channels in CTs	The number of input and output DMA channels of the tiles are an absolute hardware limitation. It limits the number of ObjectFIFOs, taking each a channel. As mentioned in the introduction, two in and two out DMA channels are available.

Table 2
Documentation summary for new developers targeting AIE architectures.

Codename	Description
AMD AM009	Complete introduction to AIE spatial architecture, with a close look to hardware specification and functionalities. We remind that Ryzen AI NPUs have only AIE-ML architecture.
AMD AM020	Complete introduction to AIE-ML spatial architecture, the evolution of AIE architecture for ML applications. Hardware specifications are also highlighted, with the main differences with the older architecture.
AMD UG1079	Complete introduction to AIE tools. Standard compilation, AIE Graphs and Utility tools are deeply explained, but its not mandatory to read everything as MLIR-AIE uses other tools. Still, important concepts are introduced such as memory stall, but also the tracing and profiling implementation. Lookup tables are also introduced.
AMD UG1076	Main software programming-related documentation for AI Engines. Presents AIE-API with examples. Developers should ready it carefully.
AIE-API	C++ heady only library triggering custom intrinsics for AIE architecture. MLIR-AIE uses also this one, while an AIEML-API exists.
XRT Native APIs	C/C++ Native APIs for runtime execution of AIE kernels, explaining every steps.

floading DSP workloads was not the best one. AIE-ML architecture, in its name, is for ML applications, not DSP. We spent a huge time on debugging for nothing because some implementations where not available for this architecture. We were forced to use MLIR-AIE with AIE-API for compatibility issues presented before. This toolchain is very recent, even more for AIE-ML architecture. The idea of implementing our own radix-2 FFT algorithm, or more generally use multiple CTs to perform computations, let us find that AMD/Xilinx proposes solutions in their Vitis Tools. Nevertheless, Ryzen AI NPUs are chips incorporated in microprocessors, limiting the number of AI Engines compared to Versal SoC platforms.

A future project would be to buy two Versal SoC with AIE architecture and optimize DFTs for 5G, creating a simple peer-to-peer network and benchmark different implementations of the DFTs. These chips have hundreds of AI Engines (not 20 as in Ryzen AI), optimized for DSP applications, and possess already a documentation as a starting point to implement these DFTs in an efficient way.

References

AMD/Xilinx (Jan. 2023a). *AMD unveiling for AIE-ML architecture*. URL: https://www.amd.com/en/blogs/2023/amd-vitis-ai-development-platform-3-0-unveiling-t.html.

— (2023b). *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)*. URL: https://docs.amd.com/r/2023.2-English/ug1393-vitis-application-acceleration/Reducing-Overhead-of-Kernel-Enqueing.

— (2024). *Versal Adaptive SoC AIE-ML Architecture Manual (AM020)*. URL: https://docs.amd.com/r/en-US/am020-versal-aie-ml/Functional-Overview.

Brown, Nick and Gabriel Rodriguez-Canal (Feb. 2025). "Seamless Acceleration of Fortran Intrinsics via AMD AI Engines". In: *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '25. ACM, pp. 185–185. DOI: 10.1145/3706628.3708854. URL: http://dx.doi.org/10.1145/3706628.3708854.

Heideman, Michael, Don Johnson, and Charles Burrus (1984). "Gauss and the history of the fast Fourier transform". In: *IEEE Assp Magazine* 1.4, pp. 14–21.

Hunhoff, Erika et al. (2025). *Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface*. arXiv: 2504.18430 [cs.SE]. URL: https://arxiv.org/abs/2504.18430.

Lattner, Chris et al. (2020). *MLIR: A Compiler Infrastructure for the End of Moore's Law*. arXiv: 2002.11054 [cs.PL]. URL: https://arxiv.org/abs/2002.11054.

Rösti, André and Michael Franz (2025). *Unlocking the AMD Neural Processing Unit for ML Training on the Client Using Bare-Metal-Programming Tools*. arXiv: 2504.03083 [cs.AR]. URL: https://arxiv.org/abs/2504.03083.

Taka, Endri et al. (2023). *MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine*. arXiv: 2311.04980 [cs.AR]. URL: https://arxiv.org/abs/2311.04980.

A Code Review

```
def dft_power_2(dev):

    @device(dev)
    def device_body():

        # Number of elements to be loaded in FIFO
        n = 512 * 2

        # Define types of Data
        data_type_tile = np.ndarray[(n,), np.dtype[bfloat16]]

        # Tile declarations
        ShimTile = tile(0, 0)
        ComputeTile1 = tile(0, 2)

        # AIE-array data movement with object fifos
        of_in = object_fifo("in_data", ShimTile, ComputeTile1, 2, data_type_tile)
        of_out = object_fifo("out_data", ComputeTile1, ShimTile, 2, data_type_tile)

        # Create a handle to an externally-defined kernel
        dft = external_func(
            "dft_power_2",
            [data_type_tile, data_type_tile],
        )

        # Compute tile
        @core(ComputeTile1, "dft_power_2.o", stack_size=0x1000)
        def core_body():
            for _ in range_(sys.maxsize):
                elem_in = of_in.acquire(ObjectFifoPort.Consume, 1)
                elem_out = of_out.acquire(ObjectFifoPort.Produce, 1)
                dft(elem_in, elem_out)
                of_in.release(ObjectFifoPort.Consume, 1)
                of_out.release(ObjectFifoPort.Produce, 1)

        # Set up a packet-switched flow from core to shim for tracing information
        tiles_to_trace = [ComputeTile1, ShimTile]
        if trace > 0:
            trace_utils.configure_packet_tracing_flow(tiles_to_trace, ShimTile)


        # To/from AIE-array data movement
        @runtime_sequence(data_type_tile, data_type_tile)
        def sequence(x_in, y_out):
            if trace > 0:
                trace_utils.configure_packet_tracing_aie2(
                    tiles_to_trace=tiles_to_trace,
                    shim=ShimTile,
                    trace_size=8129,
                    shim_burst_length=64
                )

            in_task = shim_dma_single_bd_task(
                of_in, x_in, sizes=[1, 1, 1, n], issue_token=True
            )

            out_task = shim_dma_single_bd_task(
                of_out, y_out, sizes=[1, 1, 1, n], issue_token=True
            )

            dma_start_task(in_task, out_task)
            dma_await_task(in_task, out_task)

            if trace > 0:
                trace_utils.gen_trace_done_aie2(ShimTile)

```

```
# Declares that subsequent code is in mlir-aie context
with mlir_mod_ctx() as ctx:
    dft_power_2(dev = AIEDevice.npu1_1col)
    res = ctx.module.operation.verify()
    if res == True:
        print(ctx.module)
    else:
        print(res)
```

Listing 1: Low-level configuration using IRON API.

```
namespace po = boost::program_options;

using DATATYPE = std::bfloat16_t;

int main(int argc, const char *argv[]) {

  // ------------------------------------------------------
  // Parse program arguments
  // ------------------------------------------------------
  po::options_description desc("Allowed options");
  po::variables_map vm;
  test_utils::add_default_options(desc);

  test_utils::parse_options(argc, argv, desc, vm);

  constexpr int32_t IN_SIZE = 512 * 2;
  constexpr int32_t OUT_SIZE = 512 * 2;

  // Load instruction sequence
  std::vector<uint32_t> instr_v =
      test_utils::load_instr_sequence(vm["instr"].as<std::string>());

  // ------------------------------------------------------
  // Get device, load the xclbin & kernel and register them
  // ------------------------------------------------------
  // Get a device handle
  unsigned int device_index = 0;
  auto device = xrt::device(device_index);

  // Load the xclbin
  auto xclbin = xrt::xclbin(vm["xclbin"].as<std::string>());

  // Load the kernel
  std::string Node = vm["kernel"].as<std::string>();

  // Get the kernel from the xclbin
  auto xkernels = xclbin.get_kernels();
  auto xkernel = *std::find_if(xkernels.begin(), xkernels.end(),
                               [Node](xrt::xclbin::kernel &k) {
                                 auto name = k.get_name();
                                 return name.rfind(Node, 0) == 0;
                               });
  auto kernelName = xkernel.get_name();

  device.register_xclbin(xclbin);
  xrt::hw_context context(device, xclbin.get_uuid());
  auto kernel = xrt::kernel(context, kernelName);

  // ------------------------------------------------------
  // Initialize input/ output buffer sizes and sync them
  // ------------------------------------------------------

  auto bo_instr = xrt::bo(device, instr_v.size() * sizeof(int),
                          XCL_BO_FLAGS_CACHEABLE, kernel.group_id(1));
  auto bo_in = xrt::bo(device, IN_SIZE * sizeof(DATATYPE),
                       XCL_BO_FLAGS_HOST_ONLY, kernel.group_id(3));

```

```
  auto bo_out = xrt::bo(device, OUT_SIZE * sizeof(DATATYPE),
                        XCL_BO_FLAGS_HOST_ONLY, kernel.group_id(5));

  void *bufInstr = bo_instr.map<void *>();
  memcpy(bufInstr, instr_v.data(), instr_v.size() * sizeof(int));

  DATATYPE *bufIn = bo_in.map<DATATYPE*>();
  bo_instr.sync(XCL_BO_SYNC_BO_TO_DEVICE);

  unsigned num_iter = 1000;
  float npu_time_total = 0;
  float npu_time_min = 9999999;
  float npu_time_max = 0;

  // ------------------------------------------------------
  // Main run loop
  // ------------------------------------------------------
  for (unsigned iter = 0; iter < num_iter; iter++) {

    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < IN_SIZE; i += 2) {
      bufIn[i] = static_cast<DATATYPE>((2.0f));
      bufIn[i + 1] = static_cast<DATATYPE>(0.0f);
    }

    bo_in.sync(XCL_BO_SYNC_BO_TO_DEVICE);

    unsigned int opcode = 3;
    auto run = kernel(opcode, bo_instr, instr_v.size(), bo_in, bo_out);
    run.wait();

    auto stop = std::chrono::high_resolution_clock::now();
    bo_out.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
    DATATYPE* bufOut = bo_out.map<DATATYPE*>();

    std::cout << "running iteration " << iter <<  std::endl;

    if(bufOut[0] != static_cast<DATATYPE>(512.0f)) {
      std::cout << "Error" << std::endl;
      return -1;
    }

    // Accumulate run times
    float npu_time =
        std::chrono::duration_cast<std::chrono::microseconds>(stop - start)
            .count();

    std::cout << "Time: " << npu_time << " microseconds." << std::endl;

    npu_time_total += npu_time;
    npu_time_min = (npu_time < npu_time_min) ? npu_time : npu_time_min;
    npu_time_max = (npu_time > npu_time_max) ? npu_time : npu_time_max;
  }

  // ------------------------------------------------------
  // Print verification and timing results
  // ------------------------------------------------------

  std::cout << std::endl
            << "Avg NPU time: " << npu_time_total / num_iter << "us."
            << std::endl;

  std::cout << std::endl
            << "Min NPU time: " << npu_time_min << "us." << std::endl;

  std::cout << std::endl
            << "Max NPU time: " << npu_time_max << "us." << std::endl;
```

```
}
```

Listing 2: Host Code using XRT

```

using DATATYPE = bfloat16;
using DATATYPE_KERNEL = cbfloat16;

#define N_DFT 512

#define DFT_STAGE(VEC, TW, INPUT, OUTPUT) \
do { \
\
    aie::fft_dit_r2_stage<VEC>(INPUT, TW, N_DFT, false, OUTPUT); \
\
} while(false)

extern "C" {

alignas(aie::vector_decl_align) static DATATYPE_KERNEL tmp[N_DFT];
alignas(aie::vector_decl_align) static DATATYPE_KERNEL y[N_DFT];

void dft_power_2(DATATYPE *x_in, DATATYPE *y_out) {

    aie::set_rounding(aie::rounding_mode::positive_inf);
    aie::set_saturation(aie::saturation_mode::saturate);

#if N_DFT == 16

#elif N_DFT == 32

#elif N_DFT == 64

#elif N_DFT == 128

#elif N_DFT == 512

    alignas(aie::vector_decl_align) DATATYPE_KERNEL* __restrict x = (DATATYPE_KERNEL *)x_in;

    DFT_STAGE(256, twiddles_stage0, x, tmp);
    DFT_STAGE(128, twiddles_stage1, tmp, y);
    DFT_STAGE(64,  twiddles_stage2, y, tmp);
    DFT_STAGE(32,  twiddles_stage3, tmp, y);
    DFT_STAGE(16,  twiddles_stage4, y, tmp);
    DFT_STAGE(8,   twiddles_stage5, tmp, y);
    DFT_STAGE(4,   twiddles_stage6, y, tmp);
    DFT_STAGE(2,   twiddles_stage7, tmp, y);
    DFT_STAGE(1,   twiddles_stage8, y, (DATATYPE_KERNEL *)y_out);

#endif

}

} // extern "C"]
```

Listing 3: DFT Kernel for N equal to 512. Twiddles are defined in another header file

```
import struct

with open("build/insts.txt", "rb") as f:
    data = f.read()

# Unpack as 32-bit unsigned integers
words = struct.iter_unpack("I", data)

with open("build/insts_hex.txt", "w") as out:
    for (word,) in words:
        out.write(f"{word:08X}\n")
```

Listing 4: Output of xchesscc converted in readable instructions.

```
srcdir := $(shell dirname $(realpath $(firstword $(MAKEFILE_LIST))))

include ${srcdir}/../../makefile-common

VPATH := ${srcdir}/../../../aie_kernels/aie2

device = npu
targetname = dft_power_2
CHESS?=true

aie_py_src=${targetname}.py

all: build/final.xclbin build/insts.txt

kristof: build/insts.txt

build/%.o: %.cc
  mkdir -p ${@D}
ifeq ($(device),npu)
ifeq ($(CHESS), true)
  cd ${@D} && xchesscc_wrapper ${CHESSCCWRAP2_FLAGS} -g -c $< -o ${@F};
else
  cd ${@D} && ${PEANO_INSTALL_DIR}/bin/clang++ ${PEANOWRAP2_FLAGS} -c $< -o ${@F};
endif
else ifeq ($(device),npu2)
  cd ${@D} && xchesscc_wrapper ${CHESSCCWRAP2P_FLAGS} -DBIT_WIDTH=8 -c $< -o ${@F};
else
  echo "Device type not supported"
endif

build/aie.mlir: ${srcdir}/${aie_py_src}
  mkdir -p ${@D}
  python3 $< ${device} > $@

build/aie_trace.mlir: ${srcdir}/${aie_py_src}
  mkdir -p ${@D}
  python3 $< ${device} > $@

build/final.xclbin: build/aie.mlir build/dft_power_2.o
  mkdir -p ${@D}
ifeq ($(CHESS), true)
  cd ${@D} && aiecc.py -v --aie-generate-xclbin --aie-generate-cdo --no-compile-host --xclbin-name=${@F}
      \
        --aie-generate-npu-insts --npu-insts-name=insts.txt $(<:%=../%)
else
  cd ${@D} && aiecc.py --aie-generate-cdo --no-compile-host --xclbin-name=${@F} \
        --no-xchesscc --no-xbridge \
        --aie-generate-npu --npu-insts-name=insts.txt $(<:%=../%)
endif

build/final_trace.xclbin: build/aie_trace.mlir build/dft_power_2.o
  mkdir -p ${@D}
ifeq ($(CHESS), true)
  cd ${@D} && aiecc.py -v --aie-generate-xclbin --no-compile-host --xclbin-name=${@F} \
        --aie-generate-npu-insts --npu-insts-name=insts.txt $(<:%=../%)
else
  cd ${@D} && aiecc.py --aie-generate-cdo --no-compile-host --xclbin-name=${@F} \
        --no-xchesscc --no-xbridge \
        --aie-generate-npu --npu-insts-name=insts.txt $(<:%=../%)
endif

${targetname}.exe: ${srcdir}/test.cpp
  rm -rf _build
  mkdir -p _build
  cd _build && ${powershell} cmake ${srcdir} -DCMAKE_POLICY_VERSION_MINIMUM=3.5 -DTARGET_NAME=${
    targetname}
  cd _build && ${powershell} cmake --build . --config Release
```

```
ifeq "${powershell}" "powershell.exe"
  cp _build/${targetname}.exe $@
else
  cp _build/${targetname} $@
endif

run: ${targetname}.exe build/final.xclbin build/insts.txt
  ${powershell} ./$< -x build/final.xclbin -i build/insts_hex.txt -k MLIR_AIE

run_debug: ${targetname}.exe build/final.xclbin build/insts.txt
  gdb ./$< core

trace: ${targetname}.exe build/final_trace.xclbin build/insts.txt
  python3 instr_to_hex.py
  ${powershell} ./$< -x build/final_trace.xclbin -i build/insts_hex.txt -k MLIR_AIE -t 8192
  ${srcdir}/../../utils/parse_trace.py --filename trace.txt --mlir build/aie_trace.mlir --colshift 1 >
    trace_vs.json
  ${srcdir}/../../utils/get_trace_summary.py --filename trace_vs.json

clean_trace:
  rm -rf tmpTrace trace.txt parse*json trace*json

clean: clean_trace
  rm -rf build _build ${targetname}*.exe
```

Listing 5: Makefile modified for xchesscc compiler.

```
import numpy as np

def tw(n, radix, vec):
    n_stage = n / vec
    points = n_stage / radix
    return np.exp(-2j * np.pi * np.arange(1, radix).reshape(-1, 1) * np.arange(0, points) / n_stage)

def generate_twiddles_combined(N):
    tmp = N // 2
    stage = 0

    with open("twiddles.hpp", "w") as f:
        f.write("#pragma once\n\n")
        f.write("#ifndef __TWIDDLES_HPP__\n")
        f.write("#define __TWIDDLES_HPP__\n\n")
        f.write("#include <stdfloat>\n")
        f.write("using DATATYPE = std::bfloat16_t;\n")
        f.write("const DATATYPE twiddles_512[] = {\n")

        while tmp >= 1:
            table = tw(N, 2, tmp)
            twiddle_pairs = [(val.real, val.imag) for val in table[0]]

            for real, imag in twiddle_pairs:
                f.write(f"  static_cast<DATATYPE>({real:.5f}f), static_cast<DATATYPE>({imag:.5f}f), //
    stage {stage}\n")

            tmp = tmp // 2
            stage += 1

        f.write("};\n")
        f.write("#endif // __TWIDDLES_HPP__\n")

generate_twiddles_combined(512)
```

Listing 6: Script that generated the twiddles for each stage