PYNQ-Z2 기반 최소 PYNQ.CPP 이미지 빌드: XRT 오류의 근본 원인 분석 및 해결 방안

본 보고서는 PYNQ-Z2 보드용으로 PYNQ.CPP 라이브러리만을 포함하는 경량 SD 카드 이미지(약 200MB)를 생성하는 과정에서 발생하는 XRT(Xilinx Runtime) 관련 빌드 오류의 원인을 심층적으로 분석하고, 이에 대한 명확하고 체계적인 해결책을 제시하는 것을 목표로 합니다. 사용자가 겪는 문제는 단순한 빌드 스크립트 오류가 아닌, PYNQ 생태계의 아키텍처적 특성과 빌드 프로세스의 복잡성에 기인합니다. 따라서 본 문서는 문제 해결에 필요한 기술적 배경, 근본 원인 진단, 단계별 빌드 가이드, 그리고 고급 사용자들을 위한 추가적인 전략까지 포괄적으로 다룹니다.

섹션 1: 아키텍처 심층 분석: Zynq-7000 플랫폼에서의 PYNQ 생태계 해부

사용자가 겪고 있는 빌드 실패의 본질을 이해하기 위해서는 먼저 PYNQ 프레임워크, PYNQ.CPP, 그리고 XRT 간의 기술적 관계를 명확히 정립해야 합니다. 이들 구성 요소는 특히 개발 중심의 "클래식 PYNQ"와 배포 중심의 "PYNQ.remote"라는 두 가지 패러다임 하에서 서로 다른 역할을 수행하며, 이러한 차이점을 이해하는 것이 문제 해결의 첫걸음입니다.

1.1. PYNQ 프레임워크: Python 추상화부터 임베디드 리눅스 코어까지

PYNQ(Python Productivity for Zynq)는 Zynq SoC(System-on-Chip)에서의 FPGA 개발 진입 장벽을 낮추기 위해 설계된 오픈 소스 프레임워크입니다.[1] 그 핵심 철학은 복잡한 하드웨어 설계를 "오버레이(Overlay)"라는 개념으로 추상화하고, 이를 고수준 Python API를 통해 소프트웨어 라이브러리처럼 쉽게 사용할 수 있도록 하는 것입니다.[4]

표준 PYNQ 이미지는 단순한 Python 라이브러리 모음이 아니라, 완전한 부팅 가능 운영체제입니다. 이 이미지는 Ubuntu를 기반으로 하는 리눅스 배포판으로, 웹 서버, Jupyter

Notebook 개발 환경, 그리고 NumPy, Pandas와 같은 광범위한 과학 계산용 Python 패키지를 포함하고 있습니다.[2] Zynq XC7Z020 SoC와 512MB DDR3 메모리를 탑재한 PYNQ-Z2 보드는 이러한 개발 환경의 주요 타겟 플랫폼 중 하나입니다.[3]

이러한 "클래식 PYNQ" 이미지는 보드 상에서 직접 빠르고 반복적인 프로토타이핑과 상호작용형 개발을 수행할 수 있도록 설계되었습니다. 그 결과, 이미지의 크기는 일반적으로 수 기가바이트(약 7GB)에 달합니다.[8] 사용자가 목표로 하는 약 200MB 크기의 경량 이미지와는 근본적으로 다른 목적과 구성을 가지고 있다는 점을 인지하는 것이 매우 중요합니다. 클래식 PYNQ의 빌드 프로세스는 이처럼 풍부한 개발 환경을 구축하는 데 초점이 맞춰져 있으며, 이는 필연적으로 복잡한 의존성 관계와 큰 용량을 수반합니다.

1.2. PYNQ.CPP와 PYNQ.remote 패러다임: 프로덕션 배포를 위한 전환

PYNQ 생태계는 프로토타이핑 단계를 넘어 실제 제품 배포 단계까지 지원하기 위해 PYNQ.CPP와 PYNQ.remote라는 강력한 대안을 제공합니다. 이는 사용자의 요구사항과 정확히 일치하는 솔루션입니다.

PYNQ.CPP는 PYNQ Python API의 저수준 기능을 C++로 그대로 구현한 하드웨어 추상화 라이브러리입니다.[9] 이 라이브러리를 사용하면 Python 인터프리터를 거치지 않고 C++ 애플리케이션에서 직접 FPGA 패브릭, 메모리 관리, 디바이스 제어 등 핵심 기능에 접근할 수 있습니다. 이는 성능이 중요하거나 독립 실행형(standalone) 애플리케이션을 개발할 때 필수적입니다.

PYNQ.CPP가 가장 핵심적인 역할을 하는 곳이 바로 PYNQ.remote 프레임워크입니다. 이 아키텍처는 시스템을 두 부분으로 명확히 분리합니다. 무거운 Python 개발 환경과 Jupyter Notebook은 호스트 PC에서 실행되고, PYNQ-Z2와 같은 타겟 디바이스는 최소한의 PetaLinux 이미지만을 실행합니다.[8] 호스트와 타겟 간의 통신은 gRPC(Google Remote Procedure Call) 프로토콜을 통해 효율적으로 이루어집니다.[10]

이 구조의 가장 큰 장점은 타겟 디바이스의 이미지 크기를 획기적으로 줄일 수 있다는 점입니다. PYNQ.remote 타겟 이미지는 Jupyter, 전체 Python 스택 등 불필요한 구성 요소를 모두 제거하고, PYNQ.CPP를 핵심 런타임으로 포함하는 경량 PetaLinux로 구성됩니다. 그 결과, 이미지 크기는 200MB 미만으로 줄어들어 사용자의 목표와 정확히 부합합니다.[8] 결국, 사용자가 만들고자 하는 "PYNQ.CPP만을 포함하는 경량 이미지"는 PYNQ 프로젝트 내에서

PYNQ.remote라는 이름으로 이미 정의되고 지원되는 공식적인 빌드 결과물인 것입니다.

1.3. Xilinx Runtime (XRT)의 역할: 임베디드 Zynq와 PCIe 플랫폼의 차이

XRT는 AMD의 적응형 컴퓨팅 플랫폼을 위한 통합 소프트웨어 플랫폼으로, 드라이버, API, 그리고 다양한 유틸리티 도구를 제공합니다. XRT의 역할은 플랫폼에 따라 크게 달라지며, 이 차이점을 이해하지 못하면 빌드 과정에서 심각한 혼란과 오류를 겪게 됩니다.

XRT의 주된 사용 영역은 Alveo와 같은 데이터센터 가속기 카드를 PCIe 인터페이스를 통해 관리하는 것입니다.[11] 이러한 x86 호스트 기반 시스템에서 PYNQ는 XRT 라이브러리를 감싸는 고수준 Python 래퍼(wrapper) 역할을 합니다. 즉, PYNQ가 XRT의 API를 호출하여 FPGA를 제어하는 구조입니다.

반면, PYNQ-Z2와 같은 임베디드 Zynq 플랫폼에서 XRT의 역할은 훨씬 미묘하고 시스템에 깊숙이 통합되어 있습니다. 과거 PYNQ는 Zynq 플랫폼에서 UIO(Userspace I/O)나 FPGA Manager와 같은 상대적으로 간단한 커널 드라이버를 사용하여 PL(Programmable Logic)을 관리했습니다. 하지만 최신 PYNQ 버전에서는 플랫폼 간 API 통일성을 높이기 위해 임베디드 흐름에도 XRT를 통합하기 시작했습니다.[7]

여기서 발생하는 충돌의 근원은 바로 XRT의 통합 방식 차이입니다. 임베디드 Zynq 플랫폼에서 XRT는 PetaLinux 빌드 과정의 일부로서 커널 모듈과 라이브러리 형태로 시스템에 통합되어야 합니다. 이는 x86 호스트에서 사용자가 직접 설치하는 독립적인 런타임 환경과는 완전히 다릅니다. 사용자가 겪는 빌드 오류는 PYNQ의 sdbuild 시스템이 이 두 가지 다른 XRT 사용 모델을 혼동하여, 임베디드 타겟을 빌드하면서 Alveo 카드용 호스트 중심의 XRT 환경을 설정하거나 링크하려고 시도하기 때문에 발생할 가능성이 매우 높습니다. 즉, 빌드 프로세스가 Zynq-7000에 적합하게 통합된 임베디드 XRT 구성 요소를 사용하는 대신, 존재하지 않는 호스트 스타일의 XRT를 찾으면서 실패하는 것입니다. 이 문제는 사용자가 의도한 최소 배포용 이미지(PYNQ.remote)가 아닌, 복잡한 개발용 이미지(Classic PYNQ)를 빌드하려 할 때 더욱 두드러집니다.

섹션 **2:** 근본 원인 분석**: XRT** 빌드 실패의 원인 규명

사용자의 빌드 실패는 여러 요인이 복합적으로 작용한 결과일 가능성이 높습니다. 본 섹션에서는 PYNQ 빌드 시스템의 내부 동작을 분석하고, 커뮤니티 포럼에서 보고된 실제 실패 사례들을 바탕으로 오류의 근본 원인을 체계적으로 진단합니다.

2.1. PYNQ sdbuild 흐름의 해부**:** 스테이지**,** 패키지**,** 그리고 보드 명세

PYNQ SD 카드 이미지 생성의 핵심은 <PYNQ_repo>/sdbuild/ 디렉토리에 위치한

Makefile입니다.[17] 이

Makefile은 최종 이미지를 만들기 위해 여러 단계(Stage)로 구성된 복잡한 프로세스를 조율합니다.

- 빌드 스테이지**:** 전체 프로세스는 크게 두 부분으로 나뉩니다. 첫째, 보드에 독립적인(board-agnostic) 기본 Ubuntu 루트 파일 시스템을 구축하는 단계(Stage 1-3)입니다. 둘째, 이 기본 파일 시스템 위에 특정 보드를 위한 패키지와 설정을 추가하는 단계(Stage 4)입니다.[8]
  jupyter, python_packages_jammy, xrtlib와 같은 패키지들이 이 과정에서 시스템에 설치됩니다.
- 보드 명세 **(.spec)** 파일**:** 지원되는 각 보드는 고유한 .spec 파일을 가집니다 (예: Pynq-Z2.spec). 이 파일은 빌드에 필요한 주요 파라미터들, 즉 타겟 아키텍처(ARM), PetaLinux BSP 위치, 그리고 Stage 4에서 설치할 추가 패키지 목록 등을 정의합니다.[19] 이 파일의 설정이 잘못되면 빌드 실패로 직결될 수 있습니다.

이러한 구조는 유연성을 제공하지만, 동시에 복잡성을 증가시킵니다. 사용자가 어떤 빌드 타겟을 선택하느냐에 따라 Makefile은 서로 다른 스테이지 구성과 패키지 목록을 사용하게 되며, 이것이 바로 문제의 핵심으로 이어집니다.

2.2. 버전 호환성 매트릭스**: PYNQ, XRT,** 그리고 **AMD** 디자인 도구의 정렬

PYNQ 빌드 문제에서 가장 빈번하게 발생하는 원인 중 하나는 바로 소프트웨어 버전 불일치입니다. PYNQ의 특정 릴리즈는 특정 버전의 Vivado, Vitis, PetaLinux, 그리고 호스트 OS(예: Ubuntu 18.04, 20.04, 22.04)와 함께 사용될 것을 가정하고 설계 및 테스트되었습니다.[17]

특히 PYNQ와 XRT 간의 관계는 매우 엄격합니다. PYNQ 커뮤니티 포럼의 논의에 따르면, 특정 버전 이상의 XRT(예: 2.15 초과)는 이전 버전의 PYNQ가 의존하던 저수준 API를 제거하거나 변경하여, 빌드 시점 또는 런타임에 AttributeError나 undefined symbol과 같은 오류를 유발합니다.[23] 공식 문서는 종종 최소 요구 버전에 대해서만 언급하고 최대 호환 버전에 대해서는 명시하지 않아 사용자에게 혼란을 줄 수 있습니다.[23]

따라서 성공적인 빌드를 위해서는 아래와 같이 검증된 버전 조합을 사용하는 것이 매우 중요합니다.

표 **2.1: PYNQ, XRT,** 및 **AMD** 도구 버전 호환성 매트릭스

PYNQ 버전	필수 XRT 버전	필수 Vivado/Vitis 버전	필수 PetaLinux 버전	권장 호스트 OS

v3.1	≤ 2.15	2024.1	2024.1	Ubuntu 22.04
v3.0.1	≤ 2.14	2022.2	2022.2	Ubuntu 20.04
v2.7	≤ 2.12	2021.2	2021.2	Ubuntu 20.04
v2.6	≤ 2.8	2020.2	2020.2	Ubuntu 18.04

참고: 위 표는 공식 릴리즈 노트와 커뮤니티 논의를 종합하여 구성되었습니다.[18] 빌드 환경 구성 시 이 표를 기준으로 도구 버전을 정확히 일치시키는 것이 필수적입니다.

2.3. 주요 실패 모드 진단

분석 결과, 사용자가 겪는 XRT 관련 오류는 다음 세 가지 주요 실패 모드 중 하나 또는 그 이상의 조합으로 인해 발생했을 가능성이 높습니다.

2.3.1. 실패 모드 **1:** 빌드 타겟 불일치 **(**가장 유력한 원인**)**

앞서 1.2절에서 설명했듯이, PYNQ 빌드 시스템은 두 가지 주요 결과물, 즉 클래식 PYNQ 이미지와 PYNQ.remote 이미지를 생성할 수 있습니다. 사용자가 단순히 make 명령어를 실행했다면, 이는 기본적으로 대용량의 클래식 PYNQ 이미지를 생성하는 프로세스를 트리거합니다. 이 프로세스는 Jupyter와 전체 Python 스택을 포함하며, 이를 지원하기 위해 xrtlib 패키지를 설치하려고 시도합니다.[8] 그러나 이

xrtlib 패키지 설치 과정은 Zynq-7000과 같은 임베디드 환경에 최적화되어 있지 않으며, 호스트 스타일의 XRT 설정을 가정하는 경우가 많아 충돌을 일으킵니다.

반면, 사용자의 목표인 경량 C++ 이미지를 생성하기 위한 올바른 빌드 타겟은 make pynqremote입니다.[29] 이 타겟은

meta-pynq Yocto 레이어를 기반으로 한 별도의 PetaLinux 설정을 사용하여, Jupyter와 Python 생태계를 의도적으로 제외하고 임베디드 환경에 맞게 XRT 커널 모듈과 PYNQ.CPP 라이브러리만을 정확하게 통합합니다. 따라서 사용자의 오류는 잘못된 도구(빌드 타겟)를 사용함으로써 발생한 예견된 결과일 가능성이 매우 높습니다.

2.3.2. 실패 모드 2: 호스트 환경 오염

PYNQ 빌드 프로세스는 타겟 파일 시스템을 구축하기 위해 chroot라는 리눅스 유틸리티를 사용하여 격리된 환경을 만듭니다. 하지만 이 격리 과정이 완벽하지 않을 경우, 호스트 시스템의 환경 변수(예: XILINX_XRT, LD_LIBRARY_PATH)가 chroot 환경 내부로 "유출(leak)"될 수 있습니다.

이러한 유출이 발생하면, chroot 내부에서 실행되는 빌드 스크립트가 타겟의 파일 시스템이 아닌 호스트 시스템의 경로(예: /opt/xilinx/xrt/lib/libxrt_core.so)에서 라이브러리를 찾으려고 시도합니다. 당연히 이제 막 생성 중인 타겟 루트 파일 시스템에는 해당 경로와 파일이 존재하지 않으므로, "cannot open shared object file: No such file or directory"와 같은 치명적인 오류가 발생하며 빌드가 중단됩니다.[30] 이는 커뮤니티 포럼에서도 빈번하게 보고되는 전형적인 실패 사례입니다.

2.3.3. 실패 모드 3: Zynq-7000 특정 XRT 구성 문제

Zynq-7000 플랫폼에서 XRT를 사용할 때는 하드웨어 관리 방식과 관련된 특수한 고려사항이 있습니다. 기본적으로 PYNQ는 FPGA Manager라는 리눅스 커널 드라이버를 사용하여 부팅 후 동적으로 비트스트림을 재구성합니다. 그러나 인터럽트 기능을 갖춘 XRT를 활성화하려면, 이 FPGA Manager 기능을 비활성화해야 하는 경우가 있습니다. 이는 AXI 인터럽트가 동적으로 로드될 수 없는 리눅스 커널의 알려진 제약사항 때문입니다.[16]

클래식 PYNQ 이미지의 표준 빌드 프로세스는 이러한 복잡한 상충 관계를 올바르게 처리하지 못할 수 있습니다. 예를 들어, 디바이스 트리에 AXI 인터럽트를 활성화했지만 부팅 비트스트림에 해당 디자인이 없으면 커널 부팅이 멈추는 현상이 발생할 수 있습니다.[16] 또한, XRT 구성 요소가 기본 이미지에 올바르게 통합되지 않으면 런타임에 "No devices found, XRT environment sourced?"와 같은 오류가 발생하기도 합니다.[31] 반면,

pynqremote 빌드 타겟은 이러한 Zynq-7000의 특수성을 고려하여 PetaLinux 레시피가 사전에 올바르게 구성되어 있어 이러한 문제를 회피합니다.

이러한 다각적인 분석을 통해 XRT 관련 빌드 실패는 단일 원인이 아닌, 빌드 타겟 선택의 오류, 빌드 환경의 불안정성, 그리고 플랫폼 특유의 복잡성이 결합된 문제임이 명확해졌습니다. 따라서 해결책 또한 이러한 모든 요소를 종합적으로 다루는 체계적인 접근 방식을 필요로 합니다.

섹션 **3:** 최소 **PYNQ.CPP** 이미지 생성을 위한 최종 빌드 가이드

본 섹션에서는 앞선 분석을 바탕으로, 사용자가 목표하는 PYNQ.CPP 기반의 경량 이미지를 성공적으로 빌드하기 위한 구체적이고 검증된 절차를 단계별로 제시합니다. 이 가이드는 잠재적인 실패 요인을 원천적으로 차단하고 재현 가능한 성공을 보장하는 데 초점을 맞춥니다.

3.1. Docker를 이용한 일관되고 격리된 빌드 환경 구축

2.3.2절에서 진단한 '호스트 환경 오염' 문제를 근본적으로 해결하기 위해, Docker를 사용하여 완전히 격리된 빌드 환경을 구축하는 것을 강력히 권장합니다. 이는 빌드 성공의 가장 중요한 전제 조건 중 하나입니다. Docker를 사용하면 모든 도구 체인과 의존성이 명시적으로 제어되는 깨끗하고 재현 가능한 환경을 확보할 수 있습니다.[17]

단계별 지침**:**

1. **Docker** 설치**:** 호스트 머신(권장: Ubuntu 20.04/22.04)에 Docker 공식 설치 지침에 따라 Docker를 설치합니다.
2. **AMD** 도구 설치**:** 호스트 머신에 PYNQ 버전에 맞는 AMD 도구(Vivado, Vitis, PetaLinux)를 설치합니다. 설치 버전은 앞서 제시된 표 **2.1**을 반드시 참조하여 정확히 일치시켜야 합니다. 예를 들어 PYNQ v3.0.1을 빌드한다면 2022.2 버전의 도구를 설치해야 합니다.
3. **PYNQ** 저장소 복제**:** PYNQ 소스 코드와 관련 서브모듈을 함께 복제합니다. --recursive 옵션이 필수적입니다.[29]
   ```Bash
   git clone --recursive https://github.com/Xilinx/PYNQ.git
   cd PYNQ
   ```
4. **Docker** 컨테이너 실행**:** PYNQ 공식 문서에서 제공하는 docker run 명령어를 사용하여 빌드 컨테이너를 실행합니다. 이 명령어는 호스트에 설치된 AMD 도구와 PYNQ 소스 코드 디렉토리를 컨테이너 내부로 마운트합니다. 아래 명령어에서 /tools/Xilinx와 /opt/petalinux는 실제 AMD 도구가 설치된 경로로 수정해야 합니다.
   ```Bash
   docker run \
     --init \
   ```

```
  --rm \
  -it \
  -v /tools/Xilinx:/tools/Xilinx:ro \
  -v /opt/petalinux:/opt/petalinux:ro \
  -v $(pwd):/workspace \
  --name pynq-sdbuild-env \
  --privileged \
  pynqdock:latest \
  /bin/bash
```

- -v 옵션은 볼륨 마운트를 의미하며, :ro는 읽기 전용으로 마운트하여 호스트 도구의 손상을 방지합니다.
- --privileged 옵션은 빌드 과정 중 일부(예: 루프백 디바이스 마운트)에 필요합니다.

3.2. 단계별 절차: PYNQ-Z2용 PYNQ.remote 이미지 빌드

이제 격리된 Docker 컨테이너 환경 내에서 실제 빌드 프로세스를 진행합니다. 이 과정의 핵심은 올바른 make 타겟을 사용하는 것입니다.

1. 빌드 환경 진입 및 도구 설정: Docker 컨테이너 내부의 셸에서 다음 명령어를 실행하여 AMD 도구 체인 환경을 설정합니다. 경로는 실제 마운트된 경로와 버전에 맞게 조정합니다.[17]
   ```
   Bash
   # Docker 컨테이너 내부에서 실행
   source /tools/Xilinx/Vitis/2022.2/settings64.sh
   source /opt/petalinux/2022.2/settings.sh
   ```

2. 올바른 빌드 타겟 실행: sdbuild 디렉토리로 이동하여 make pynqremote 명령어를 실행합니다. BOARDS 변수를 통해 타겟 보드를 Pynq-Z2로 명시해야 합니다. 이것이 본 보고서의 핵심적인 해결책입니다.[29]
   ```
   Bash
   cd /workspace/sdbuild/
   make pynqremote BOARDS=Pynq-Z2
   ```

3. **pynqremote** 타겟의 동작 원리: 이 명령어는 일반 make와 근본적으로 다르게 동작합니다.
   - **PetaLinux** 프로젝트: PYNQ.remote용으로 사전 정의된 경량 PetaLinux 설정을 사용합니다. 이 설정은 meta-pynq Yocto 레이어를 활용하여 루트 파일 시스템에 Jupyter, NumPy, Pandas 등 대용량 패키지를 포함하지 않도록 구성되어 있습니다.[8]
   - 핵심 구성 요소: 대신, 임베디드 XRT 커널 모듈, 관련 라이브러리, 그리고 PYNQ.CPP 라이브러리만을 선별적으로 설치합니다. 이는 2.3절에서 분석한 실패 모드 1(빌드 타겟

불일치)과 3(Zynq-7000 특정 구성 문제)을 직접적으로 해결합니다.
    - 결과물: 최종적으로 생성되는 이미지는 사용자가 원했던 200MB 내외의 경량 이미지입니다.
4. 빌드 과정 모니터링: 빌드 프로세스는 PetaLinux 프로젝트 생성, 커널 컴파일, 루트 파일 시스템 구축 등 여러 단계를 거치며 수 시간이 소요될 수 있습니다.[18] 터미널에 출력되는 로그를 통해 진행 상황을 확인할 수 있으며, 오류 발생 시 해당 로그가 디버깅의 첫 단서가 됩니다.

3.3. 이미지 배포, 검증 및 초기 애플리케이션 테스트

빌드가 성공적으로 완료되면, 생성된 이미지를 보드에 배포하고 정상 동작을 확인해야 합니다.

1. 결과물 확인: 빌드가 완료되면 최종 이미지 파일(예: Pynq-Z2-vX.X.X-pynqremote.img)이 /workspace/sdbuild/output/ 디렉토리 내에 생성됩니다. Docker 컨테이너는 호스트의 PYNQ 디렉토리를 마운트했으므로, 이 파일은 호스트 머신의 PYNQ/sdbuild/output/에서도 접근 가능합니다.
2. **SD** 카드 플래싱: BalenaEtcher, Win32DiskImager 또는 dd와 같은 도구를 사용하여 생성된 .img 파일을 8GB 이상의 microSD 카드에 기록합니다.[1]
3. 보드 설정 및 부팅:
    - PYNQ-Z2 보드의 부팅 점퍼(JP5)를 **SD** 위치로 설정하여 microSD 카드로 부팅하도록 합니다.[34]
    - 전원 점퍼(JP4)를 **USB** 또는 **REG**(외부 12V 어댑터 사용 시)로 설정합니다.
    - 플래싱된 microSD 카드를 보드 하단의 슬롯에 삽입합니다.
    - PC와 보드의 **PROG-UART** MicroUSB 포트를 케이블로 연결하여 시리얼 터미널 통신을 준비합니다.
    - 이더넷 케이블을 보드와 네트워크에 연결합니다.
    - 보드의 전원을 켭니다.[34]
4. 동작 검증:
    - 시리얼 터미널 연결: PuTTY, TeraTerm, 또는 minicom과 같은 터미널 프로그램을 사용하여 보드의 시리얼 포트(Baud rate: 115200)에 연결합니다. 부팅 로그가 출력되고, 잠시 후 로그인 프롬프트가 나타나는 것을 확인합니다.[34]
    - **pynq-remote** 서버 확인: 보드에 로그인한 후, ps aux | grep pynq-remote 명령어를 실행하여 pynq-remote gRPC 서버 프로세스가 자동으로 실행 중인지 확인합니다.
    - 원격 연결 테스트: 호스트 PC에서 간단한 Python 스크립트를 작성하여 보드에 원격으로 연결하고 통신을 테스트합니다. 이를 위해 호스트 PC에 pynq 라이브러리가 설치되어 있어야 합니다 (pip install pynq).

Python
```
# host_test.py
from pynq.remote import RemoteDevice
```

```
# 보드의 IP 주소 또는 호스트 이름으로 변경
board_ip = "192.168.2.99"
pynq_z2 = RemoteDevice(board_ip)

# 간단한 속성 접근으로 연결 확인
print(f"Successfully connected to {pynq_z2.name}")
print(f"Device specification dictionary: {pynq_z2.spec}")

# 연결 종료
pynq_z2.close()
```

위 스크립트를 호스트 PC에서 실행하여 성공 메시지가 출력되면, 빌드부터 배포, 원격 실행까지의 전 과정이 성공적으로 완료되었음을 의미합니다.

섹션 **4:** 고급 주제 및 전략적 권장 사항

본 보고서는 당면한 빌드 문제를 해결하는 것을 넘어, 사용자가 향후 PYNQ 생태계를 더욱 깊이 있고 효과적으로 활용할 수 있도록 고급 주제와 전략적 가이드를 제공합니다.

4.1. 대안 경로**: meta-pynq Yocto** 레이어를 이용한 심층 사용자 정의

sdbuild 흐름은 편리하지만, 고수준의 추상화를 제공하기 때문에 시스템의 모든 세부 사항을 제어하는 데는 한계가 있습니다. 예를 들어, 커스텀 커널 드라이버를 추가하거나, pynqremote 이미지에서조차 불필요하다고 판단되는 특정 패키지를 제거하는 등 극단적인 최적화가 필요한 경우, 근간이 되는 빌드 시스템인 PetaLinux/Yocto를 직접 다루어야 합니다.

PYNQ는 이를 위해 meta-pynq라는 Yocto 레이어를 제공합니다.[8] 이 레이어는 PYNQ 관련 구성 요소(라이브러리, 드라이버, 서비스 등)를 빌드하고 이미지에 설치하기 위한 레시피(recipe)들을 포함하고 있습니다. 고급 사용자는 다음의 개략적인 워크플로우를 통해 완벽한 사용자 정의 이미지를 생성할 수 있습니다.

1. **PetaLinux** 프로젝트 생성**:** PYNQ-Z2 보드용 표준 PetaLinux 프로젝트를 생성합니다.
2. **meta-pynq** 레이어 추가**:** 생성된 PetaLinux 프로젝트의 bblayers.conf 파일에 meta-pynq 레이어의 경로를 추가합니다.
3. 루트 파일 시스템 구성**:** petalinux-config -c rootfs 명령을 실행하여 메뉴 기반 설정 화면에 진입합니다. 여기서 user packages 항목으로 이동하여 pynq, pynq-cpp, libcma 등 필요한 패키지만을 명시적으로 선택하고, 원치 않는 다른 패키지들은 비활성화합니다.
4. 이미지 빌드**:** petalinux-build 명령을 통해 이미지를 빌드합니다.

이 접근 방식은 Yocto의 레시피(.bb)와 어펜드(.bbappend) 파일 형식에 대한 이해를 필요로

하지만 [37], 이미지에 포함되는 모든 소프트웨어 구성 요소를 완벽하게 제어할 수 있는 궁극적인 유연성을 제공합니다. 이는 커뮤니티에서 논의되었던 고급 최적화 시도들과 맥을 같이 합니다.[8]

4.2. sdbuild 프로세스를 위한 체계적인 문제 해결 가이드

sdbuild 과정에서 예상치 못한 오류가 발생했을 때, 다음의 체크리스트를 체계적으로 점검하면 문제의 원인을 신속하게 파악하고 해결하는 데 도움이 됩니다. 이는 커뮤니티 포럼에서 축적된 집단 지성을 정리한 것입니다.

1. 환경 **(Environment):**
   - 빌드 프로세스를 공식 PYNQ Docker 컨테이너 내부에서 실행하고 있습니까?
   - Docker 컨테이너를 실행하기 전에, 호스트 셸에서 unset XILINX_XRT, unset LD_LIBRARY_PATH 등 충돌 가능성이 있는 환경 변수를 초기화했습니까? [30]
2. 버전 **(Versions):**
   - 사용 중인 PYNQ 소스 코드 브랜치, Vivado, Vitis, PetaLinux, 그리고 호스트 OS 버전이 표 **2.1**의 호환성 매트릭스와 정확히 일치합니까? [23]
3. 빌드 상태 **(Clean State):**
   - 이전 빌드 실패 후 남은 아티팩트를 제거하기 위해 make clean 또는 make delete 명령을 실행했습니까? 마운트된 루프백 디바이스는 가장 흔한 실패 원인 중 하나입니다.[17]
4. 자원 **(Resources):**
   - 빌드 머신에 충분한 디스크 공간(최소 200GB 권장)과 RAM(최소 8GB, 16GB 이상 권장)이 확보되어 있습니까? [21]
5. 로그 **(Logs):**
   - 빌드 실패 시, 터미널에 출력된 전체 로그를 저장하고 분석했습니까? 로그의 마지막 부분에는 실패한 정확한 명령어와 오류 메시지가 포함되어 있습니다.
6. 보드 명세 **(Board Specifics):**
   - 사용자 정의 보드를 빌드하는 경우, .spec 파일 내의 BSP 또는 XSA 파일 경로가 정확합니까? [31]

4.3. 전략적 프레임워크: Full PYNQ와 Minimal PYNQ.CPP 선택 가이드

본 보고서에서 다룬 두 가지 PYNQ 패러다임은 서로 다른 목적을 가지고 있으며, 프로젝트의 생명주기 단계와 요구사항에 따라 적절한 것을 선택하는 것이 중요합니다. 이는 단순히 기술적인 선택을 넘어, 개발 효율성과 최종 제품의 성능을 좌우하는 전략적인 결정입니다.

- **"**클래식 **PYNQ" (Full Image)**를 선택해야 하는 경우**:**

    - 신속한 프로토타이핑: 아이디어를 빠르게 구현하고 검증해야 할 때.
    - 연구 및 교육: FPGA와 파이썬을 함께 배우거나, 복잡한 알고리즘을 상호작용적으로 탐색할 때.
    - 상호작용형 디버깅: Jupyter Notebook의 시각화 및 단계별 실행 기능을 활용하여 하드웨어 동작을 디버깅할 때.
    - 자원 제약이 덜 중요한 경우: 보드의 스토리지 공간이나 부팅 시간이 핵심적인 제약 조건이 아닐 때.
- **PYNQ.remote / PYNQ.CPP (Minimal Image)**를 선택해야 하는 경우:
    - 제품 배포: 개발이 완료된 애플리케이션을 최종 제품에 탑재할 때.
    - 고성능 요구사항: Python 인터프리터의 오버헤드를 제거하고 C++의 성능을 최대한 활용해야 하는 애플리케이션.
    - 대규모 시스템 통합: 기존의 C++ 기반 호스트 시스템이나 클라우드 인프라에 FPGA 기능을 통합할 때.
    - 자원이 제한된 환경: eMMC와 같이 저장 공간이 작거나, 빠른 부팅 시간 및 낮은 메모리 사용량이 요구되는 임베디드 시스템.

결론적으로, PYNQ 생태계는 탐색과 개발을 위한 "클래식 PYNQ"와, 최적화와 배포를 위한 "PYNQ.remote"라는 두 가지의 강력하고 상호 보완적인 도구를 제공합니다. 사용자가 겪었던 빌드 오류는 이 두 가지 패러다임의 경계에서 발생한 혼란이었습니다. 올바른 도구를 올바른 목적에 사용하는 것이 성공적인 PYNQ 기반 시스템 개발의 첫걸음이자 가장 중요한 원칙이라는 점을 재확인하며 본 보고서를 마칩니다.

참고 자료

1. PYNQ Framework on TityraCore Zynq® 7000 | Numato Lab Help Center, 9월 17, 2025에 액세스, https://numato.com/blog/pynq-framework-on-tityracore-zynq-7000/
2. Xilinx/PYNQ: Python Productivity for ZYNQ - GitHub, 9월 17, 2025에 액세스, https://github.com/Xilinx/PYNQ
3. PYNQ-Z1 Reference Manual - Digilent, 9월 17, 2025에 액세스, https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual
4. Productivity of Embedded Designs with PYNQ - Avnet, 9월 17, 2025에 액세스, https://www.avnet.com/apac/products/cp/xilinx-pynq/reference-materials/productivity-of-embedded-designs-with-pynq/
5. numato.com, 9월 17, 2025에 액세스, https://numato.com/blog/pynq-framework-on-tityracore-zynq-7000/#:~:text=PYNQ%20also%20provides%20a%20web,analytics%2C%20and%20embedded%20systems%20development.
6. Python Productivity for Zynq, 9월 17, 2025에 액세스, https://www.xilinx.com/publications/events/xtd/05_Xilinx_Pynq_framework_to_support_Python_and_IIOT.pdf
7. Xilinx XRT : r/FPGA - Reddit, 9월 17, 2025에 액세스, https://www.reddit.com/r/FPGA/comments/1frloo4/xilinx_xrt/
8. How to build a minimal PYNQ 3.1 image (no Jupyter), only pynq Python library for

Overlay(), fitting in 8 GB eMMC, 9월 17, 2025에 액세스, https://discuss.pynq.io/t/how-to-build-a-minimal-pynq-3-1-image-no-jupyter-only-pynq-python-library-for-overlay-fitting-in-8-gb-emmc/8712
9. pynq.readthedocs.io, 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/latest/pynq_remote/cpp_index.html#:~:text=cpp%20is%20a%20C%2B%2B%20hardware,memory%20management%2C%20and%20device%20operations.
10. PYNQ.cpp — Python productivity for Zynq (Pynq), 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/latest/pynq_remote/cpp_index.html
11. PYNQ on XRT Platforms — Python productivity for Zynq (Pynq), 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/v2.7.0/getting_started/pynq_alveo.html
12. Frameworks | HACC Resources, 9월 17, 2025에 액세스, https://www.amd-haccs.io/frameworks.html
13. pynq - PyPI, 9월 17, 2025에 액세스, https://pypi.org/project/pynq/
14. PYNQ on XRT Platforms — Python productivity for Zynq (Pynq) - Read the Docs, 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/v2.5.1/pynq_alveo.html
15. pynq.pl_server.xrt_device Module — Python productivity for Zynq (Pynq) - Read the Docs, 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/v2.5.1/pynq_package/pynq.pl_server/pynq.pl_server.xrt_device.html
16. Pynq 2.6.1 XRT - Zynq - Support, 9월 17, 2025에 액세스, https://discuss.pynq.io/t/pynq-2-6-1-xrt-zynq/1965
17. PYNQ SD Card image — Python productivity for Zynq (Pynq) - Read the Docs, 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/latest/pynq_sd_card.html
18. How to build PYNQ image in Ubuntu 20.04 - Support, 9월 17, 2025에 액세스, https://discuss.pynq.io/t/how-to-build-pynq-image-in-ubuntu-20-04/3157
19. PYNQ Edition! Building PYNQ Images - Hackster.io, 9월 17, 2025에 액세스, https://www.hackster.io/news/pynq-edition-building-pynq-images-4f0ec30e172f
20. PYNQ v3.01 - Adaptive Support - AMD, 9월 17, 2025에 액세스, https://adaptivesupport.amd.com/s/question/0D54U00008kl1G5SAK/pynq-v301?language=en_US
21. PYNQ 3.0.1 VirtualBox install and SD build - Learn, 9월 17, 2025에 액세스, https://discuss.pynq.io/t/pynq-3-0-1-virtualbox-install-and-sd-build/4859
22. XRT Operating System Support - 2025.1 English - UG1451, 9월 17, 2025에 액세스, https://docs.amd.com/r/en-US/ug1451-xrt-release-notes/XRT-Operating-System-Support
23. Pynq / XRT version compatibility - Support - PYNQ, 9월 17, 2025에 액세스, https://discuss.pynq.io/t/pynq-xrt-version-compatibility/7832
24. Support for XRT 2.17 and above · Issue #1465 · Xilinx/PYNQ - GitHub, 9월 17, 2025에 액세스, https://github.com/Xilinx/PYNQ/issues/1465
25. Alveo Getting Started Guide — Python productivity for Zynq (Pynq), 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/v3.0.0/getting_started/alveo_getting_started.html
26. HACC-mlcluster Tools & Storage | AMD-Xilinx Center of Excellence, 9월 17, 2025에 액세스,

https://xilinx-center.csl.illinois.edu/xacc-cluster/hacc-mlcluster/tools-storage/
27. 000034458 - XRT - Xilinx Runtime version 2022.2, 2.14 Tested Operating Systems, 9월 17, 2025에 액세스, https://adaptivesupport.amd.com/s/article/000034458
28. xclmgmt seems to failed only when using v2.12 and later · Issue #7691 · Xilinx/XRT - GitHub, 9월 17, 2025에 액세스, https://github.com/Xilinx/XRT/issues/7691
29. Remote Image Build Guide — Python productivity for Zynq (Pynq) - Read the Docs, 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/pynq_remote/image_build.html
30. PYNQ build failed for custom board - Support - PYNQ, 9월 17, 2025에 액세스, https://discuss.pynq.io/t/pynq-build-failed-for-custom-board/2977
31. "No devices found, XRT environment sourced?" Error - Page 2 - Support - PYNQ, 9월 17, 2025에 액세스, https://discuss.pynq.io/t/no-devices-found-xrt-environment-sourced-error/4022?page=2
32. "No devices found, XRT environment sourced?" Error - Support - PYNQ, 9월 17, 2025에 액세스, https://discuss.pynq.io/t/no-devices-found-xrt-environment-sourced-error/4022
33. Define Custom Pynq-Z2 Board with SoC Blockset - MATLAB & Simulink - MathWorks, 9월 17, 2025에 액세스, https://www.mathworks.com/help/soc/ug/define-custom-pynq-z2-board-with-socb.html
34. PYNQ-Z2 Setup Guide, 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/v3.1/getting_started/pynq_z2_setup.html
35. PYNQ-Z2 Board: Getting Started Guide - YouTube, 9월 17, 2025에 액세스, https://www.youtube.com/watch?v=GqK4o9A4HHg&vl=en-US
36. PYNQ-Z2 - Pre-workshop setup - element14 Community, 9월 17, 2025에 액세스, https://community.element14.com/members-area/personalblogs/b/blog/posts/pynq-z2---pre-workshop-setup
37. Modifying Component Recipes - 2025.1 English - UG1137, 9월 17, 2025에 액세스, https://docs.amd.com/r/en-US/ug1137-zynq-ultrascale-mpsoc-swdev/Modifying-Component-Recipes
38. 5 Writing a New Recipe — The Yocto Project ® 5.2.3 documentation, 9월 17, 2025에 액세스, https://docs.yoctoproject.org/dev-manual/new-recipe.html
39. Create a Custom Meta-Layer — phyCORE-AM64x - PHYTEC's documentation!, 9월 17, 2025에 액세스, https://docs.phytec.com/projects/yocto-phycore-am64x/en/latest/developingwithyocto/createCustomMetaLayer.html
40. meta-xilinx-pynq - Yocto Project, 9월 17, 2025에 액세스, https://lists.yoctoproject.org/g/meta-xilinx/topic/meta_xilinx_pynq/88655368
41. meta-xilinx-pynq for ZCU102 - Yocto Project, 9월 17, 2025에 액세스, https://lists.yoctoproject.org/g/meta-xilinx/topic/meta_xilinx_pynq_for_zcu102/81733994
42. PYNQ SD Card image — Python productivity for Zynq (Pynq) - Read the Docs, 9월 17, 2025에 액세스, https://pynq.readthedocs.io/en/v2.7.0/pynq_sd_card.html
43. Rebuild Pynq image failed - Page 2 - Support, 9월 17, 2025에 액세스,

https://discuss.pynq.io/t/rebuild-pynq-image-failed/120?page=2