1.2 Computational Constraints & Model Optimization

The architecture from Chapter 1.1 sets the rules. This section is about playing inside them: what an Intel NPU will and won't accept, how to shape a model so it compiles, and how to quantize without quietly losing the quality you paid for in training. We anchor on M2M-100 throughout — partly because translation is a clean worked example, partly because M2M-100 is unforgiving enough about NPU constraints to be instructive.

The Static-Shape Mandate

The first rule of Intel NPU is that **shapes are largely set at compile time, not run time**. The compiler tiles your graph across the NCEs and SHAVE DSPs, computes the SRAM allocation, and generates a binary blob. Change the shapes and you compile a new blob — which takes seconds to tens of seconds, and which Windows Update or a driver upgrade may invalidate.

For non-LLM workloads this is an absolute constraint. The encoder of M2M-100 has to be `reshape`d to a fixed sequence length before compile:

```
encoder_model.reshape({"input_ids":      [1, 128],
                       "attention_mask": [1, 128]})
encoder_npu = core.compile_model(encoder_model, "NPU")
```

Any input shorter than 128 gets padded; anything longer either truncates or forces a new compile. Pick the sequence length once, pick it to cover your real workload's 95th percentile, and live with the padding waste on short inputs.

For LLM workloads the story has loosened. OpenVINO 2025.3 introduced **dynamic prompts on NPU by default** through the `LLMPipeline` static-shape pipeline with `PREFILL_HINT=DYNAMIC` and `NPUW_LLM_PREFILL_CHUNK_SIZE=1024`. This isn't dynamic shape in the GPU sense — it's *chunked* static prefill, where the compiler emits a fixed-shape kernel and the runtime feeds chunks until the prompt is consumed. The illusion of dynamism, paid for by a fixed chunk granularity. There's no equivalent for `OVModelForSeq2SeqLM`, which is exactly why M2M-100's decoder doesn't get the same flexibility as a Llama-3 decoder.

Intel NPU Operator Coverage

The canonical list of supported operations lives at `docs.openvino.ai/<version>/about-openvino/compatibility-and-support/supported-operations.html`, version-stamped per release. **Encoder-friendly ops are mature.** Transformer encoders compile reliably: `MatMul`, `Add`, `Multiply`, `LayerNormalization` (with decomposed fallback when the fused op isn't supported), `Softmax`, `Gelu`, `Reshape`, `Transpose`, `Concat`, `Gather` with static indices, `ScaledDotProductAttention`, `Convert`, and the `FakeQuantize`/`FakeConvert` ops for INT8/FP8 paths. OpenVINO 2025.2 explicitly added **QKV-projection and Multi-Head Attention graph-level fusions** for encoder-based LLMs, which is exactly the kind of optimization M2M-100's encoder benefits from.

Decoder pain points have specific names, and each is worth recognizing because they appear in real error messages:

- `DetectionOutput` still fails to compile on NPU and iGPU as of OpenVINO 2025.4 (Intel Community thread 1735991, Feb 2026)
- `ScatterNDUpdate` has been rejected by the VPU/NPU compiler historically (issue #13594)
- INT64 indices in `Gather` and `ScatterND` routinely cause silent CPU fallback
- Variable-length `Gather`, dynamic `Slice`, dynamic `Reshape` in autoregressive decoders are the structural reason the whole model historically had to be static

When in doubt about whether your graph compiles, the answer is to try and read the compile log. The error messages are reasonably informative; the failures are usually localizable to a specific op.

Quantization: PTQ, Not QAT

Post-training quantization (PTQ) is the default path on Intel NPU. Quantization-aware training (QAT) is technically supported by NNCF but rarely necessary — the PTQ recipes Intel has tuned for the validated NPU model list are good enough for most use cases, and they don't require retraining.

The path looks like this: export your PyTorch model to OpenVINO IR via Optimum-Intel, pick a quantization recipe (INT8 weight-only, INT4 channel-wise, INT4 group-wise, NF4 on Lunar Lake+, FP8 on Panther Lake+), and let NNCF do the work. The recipe matters because Intel NPU has strict constraints on which combinations work.

The NPU LLM quantization rule from Intel's GenAI-on-NPU guide is unambiguous: maximize the 4-bit weight ratio (`--ratio 1.0`), use `--group-size 128` for models up to ~4–5 B parameters, use `--group-size -1` (channel-wise) for larger models, and always use symmetric quantization (`--sym`). Asymmetric quantization is documented to crash the NPU LLM compile path.

The precision matrix by NPU generation:

Mode	NPU 3 (MTL)	NPU 4 (LNL)	NPU 5 (PTL)
INT8-sym weights	✅	✅	✅
INT4-sym, group-size 128	✅	✅	✅

Mode	NPU 3 (MTL)	NPU 4 (LNL)	NPU 5 (PTL)
INT4-sym, channel-wise	✅	✅	✅
NF4 (channel-wise only)	✅	✅	✅
NF4 weights + FP16 KV	✅	✅ (2025.3+)	✅
FP8 (E4M3/E5M2)	❌	❌	✅

The NF4 Lunar Lake exclusivity comes verbatim from OpenVINO's GenAI-on-NPU docs: "*The NF4 data type is only supported on Intel Core Ultra Processors Series 2 NPUs (formerly codenamed Lunar Lake) and beyond.*" The FP8 Panther Lake gating is documented in Intel's `openvino-ai-plugins-gimp` 3.2 release notes: "*FP8 model installation is now gated to NPU5000 and newer architectures.*"

Exporting M2M-100

Here are the two `optimum-cli` invocations you'll actually use:

```
# INT8 weights, stateful with KV cache (the safe default)
optimum-cli export openvino \
  --model facebook/m2m100_418M \
  --task text2text-generation-with-past \
  --weight-format int8 \
  m2m100_418M_ov_int8

# INT4 group-wise, NPU-targeted
optimum-cli export openvino \
  --model facebook/m2m100_418M \
  --task text2text-generation-with-past \
  --weight-format int4 --sym --ratio 1.0 --group-size 128 \
  m2m100_418M_ov_int4_npu
```

Two pitfalls worth calling out before you spend an afternoon debugging them. **`--task translation` does not exist** in Optimum-Intel; it lives in `optimum-neuron` for AWS Neuron, which is a different toolkit. The correct task name for M2M-100 is `text2text-generation-with-past`. And **the `--with-past` suffix is required** for a stateful, KV-cached decoder; without it the export produces a stateless decoder that re-encodes the full target prefix on every step, which destroys decode throughput.

The output is a directory containing `openvino_encoder_model.xml`, `openvino_decoder_model.xml`, `openvino_decoder_with_past_model.xml`, and the tokenizer files. Three separate models, each independently compileable to a different device — which is exactly the lever we need for the hybrid execution pattern.

Why M2M-100 Is Architecturally Expensive

Three reasons M2M-100 is harder to deploy on Intel NPU than a comparably-sized decoder-only model:

Full multi-head attention with no GQA or MQA. Look at `modeling_m2m_100.py` in HuggingFace Transformers: `self.k_proj` and `self.v_proj` both project to full `embed_dim`, and `num_heads == num_kv_heads`. The HF config has no `num_key_value_heads` field at all. A 1.2B-parameter M2M-100 decoder has the same per-token KV bandwidth as a 3.8B-parameter Phi-3-mini, because Phi-3 uses GQA with one-quarter the KV heads. We'll do the math in Chapter 2.1. The implication for NPU deployment: M2M-100's decode is bandwidth-bound at smaller parameter counts than modern models. No retrofit; switching to GQA would require retraining from scratch.

Autoregressive decoder with dynamic sequence length. The decoder generates one token at a time, with the KV cache growing on every step. The 2025.3 chunked-prefill feature relaxes this for decoder-only LLMs via `LLMPipeline`, but **no equivalent pipeline exists for `OVModelForSeq2SeqLM`**. OpenVINO 2026.0's NPU GenAI guide lists Whisper, LLM, and VLM pipelines only. M2M-100's decoder is on its own.

Encoder-decoder cross-attention. The decoder reads its own self-attention KV state *and* the encoder output every step, doubling the per-layer attention overhead relative to a decoder-only model. M2M-100's cross-attention KV cache is the same size as its self-attention KV cache for any given encoder length. This is the price of being a translation model — you keep the source sentence accessible throughout decoding — and there's no way to optimize it away.

The honest deployment recommendation that follows: **encoder on NPU** (single static prefill pass, ideal NPU fit), **decoder on CPU or iGPU** (dynamic autoregressive, where the runtime handles variable shapes well). Optimum-Intel does not expose per-component `device_map`, so this requires either subclassing `OVModelForSeq2SeqLM` or driving the IR files directly via `core.compile_model(...)`. Chapter 3.1 shows the code.

The Sizing Heuristic, Specific to Intel

For Intel NPU specifically, the rough sizing budget is: **a model whose post-quantization weight memory fits in roughly 4–8 GB will run comfortably on Lunar Lake NPU 4**. The 16 GB Copilot+ minimum spec gives you the LPDDR5X room; the static-shape constraint sets compile complexity; the LPDDR5X bandwidth ceiling sets decode throughput.

In M2M-100 sizes:

Variant	Params	FP16 weights	INT8 weights	INT4 weights	Fit

418M	418M	~840 MB	~420 MB	~210 MB	Comfortable on NPU 3+
1.2B	1.2B	~2.4 GB	~1.2 GB	~600 MB	Comfortable on NPU 4+
12B	12B	~24 GB	~12 GB	~6 GB	Infeasible on consumer NPU at FP16; tight at INT4

The 12B variant essentially doesn't fit on consumer Lunar Lake outside of pathological configurations. The 418M and 1.2B variants are the realistic deployment targets.

What This Section Bought You

You should now understand:

- **Static shapes are mandatory** for non-LLM workloads on Intel NPU; chunked prefill softens this for LLMs since 2025.3 but not for seq2seq
- **The Intel NPU operator coverage is encoder-friendly and decoder-fragile** — `DetectionOutput`, `ScatterNDUpdate`, INT64 indices, and dynamic Slice/Gather are recurring landmines
- **PTQ is the default path**; the NPU LLM quantization rule is `--sym --ratio 1.0` with group-size 128 (small) or -1 (large)
- **The precision matrix gates by generation**: NF4 needs Lunar Lake, FP8 needs Panther Lake
- **M2M-100 export goes through Optimum-Intel** with task `text2text-generation-with-past`; common mistakes are `--task translation` and missing `--with-past`
- **M2M-100 is architecturally expensive** for three structural reasons — full MHA, dynamic decode, cross-attention — none of which is fixable in post-training
- **The hybrid pattern is encoder-on-NPU, decoder-on-CPU/iGPU**, and the rest of the book builds on it

The next section turns to performance: given a model that compiles cleanly, what does its latency profile actually look like on Intel hardware, and what does that imply for agent design patterns?

Previous: *1.1 Understanding NPU Architecture* **Next:** *1.3 Latency, Throughput, and Hardware-Aware Patterns*

Revision #4
Created 2026-05-12 15:56:01 UTC by Admin
Updated 2026-05-12 19:14:31 UTC by Admin