

# Fine-Tuning 1.58-bit BitNet on Mali-G78 via llama.cpp and Vulkan

## 1 TABLE OF CONTENTS

---

|      |  |    |
|------|--|----|
| 1.1  | Introduction and Motivation.....   | 2  |
| 1.2  | Background on BitNet and 1.58-bit Quantization .....                                 | 2  |
| 1.3  | Overview of llama.cpp and GPU Backend Limitations .....                              | 4  |
| 1.4  | Design of Fine-Tuning Kernels for 1.58-bit Weights on Mali-G78 (Vulkan).....         | 6  |
| 1.5  | Modification Strategy for llama.cpp to Enable Mali-G78 GPU Fine-Tuning .....         | 9  |
| 1.6  | Profiling Tools and Performance Metric Collection for Mali-G78 .....                 | 12 |
| 1.7  | Required Frameworks, Libraries, and SDKs .....                                       | 13 |
| 1.8  | Optimization Strategies at the Vulkan Shader and Kernel Levels .....                 | 15 |
| 1.9  | Evaluation Benchmarks and Performance Success Criteria .....                         | 17 |
| 1.10 | Constraints of Smartphone Hardware and Low-Power Optimization Strategies .....       | 19 |
| 1.11 | Phased Implementation Plan with Milestones and Deliverables.....                     | 21 |
| 1.12 | Risk Assessment and Mitigation Strategies .....                                      | 24 |
| 1.13 | Underlying Assumptions and Critical Dependencies.....                                | 26 |
| 1.14 | Proof of Concept (PoC): 1-bit Matrix Multiplication Kernel in Vulkan for a GPT Layer | 28 |
| 1.15 | Integration Plan for the Vulkan Kernel into llama.cpp.....                           | 30 |
| 1.16 | References .....   | 32 |
| 1.17 | Sources .....  | 32 |

## 1.1 INTRODUCTION AND MOTIVATION

Recent advances in extreme quantization for large language models (LLMs), exemplified by **BitNet b1.58**, a 1.58-bit (ternary) LLM, promise dramatic reductions in model size and computation without sacrificing accuracy[1][2]. BitNet constrains each weight to only three values (-1, 0, +1), yielding an **entropy of ~1.58 bits per parameter**[3][4]. This approach slashes memory footprint and replaces costly floating-point multiplications with cheap integer addition/subtraction operations[5][6]. In theory, such ultra-low-bit models can achieve full-precision performance at a fraction of the energy and latency cost[7][8].

**Motivation:** Enabling **on-device fine-tuning** of 1.58-bit LLMs on smartphones would be revolutionary. It would allow end-users to personalize LLMs locally (for privacy and offline use) and leverage the efficiency of BitNet to overcome the usual resource constraints. The **Mali-G78 GPU**, found in modern SoCs (e.g. Kirin 9000, Exynos 2100), offers a capable mobile compute engine supporting Vulkan. Coupling BitNet’s efficiency with the Mali-G78’s GPU parallelism could make it feasible to fine-tune multi-billion-parameter LLMs on a phone within reasonable time and power budgets. The **llama.cpp** framework already demonstrated state-of-the-art LLM inference on commodity hardware via quantization[9]. Extending it for **training/fine-tuning** with a **Vulkan GPU backend** unlocks a new paradigm being local, efficient LLM adaptation.

This proposal presents a comprehensive technical plan for fine-tuning a 1.58-bit BitNet LLM on-device. We assume the reader has a strong system and machine learning background. We focus on **implementation specifics**, from custom low-bit matrix kernels to integration into llama.cpp, rather than general concepts. Key challenges addressed include GPU kernel design for 1.58-bit arithmetic on Mali’s Valhall architecture, modifications to llama.cpp’s ggml backend for training support and optimizing within smartphone power/thermal limits. The goal is to demonstrate that with careful engineering, a **ternary-weight LLM can be fine-tuned on a Mali-G78 GPU** with performance and accuracy comparable to desktop setups. This would mark an important step toward democratizing LLM deployment and personalization.

## 1.2 BACKGROUND ON BITNET AND 1.58-BIT QUANTIZATION

**BitNet b1.58** (introduced by Ma et al., 2024) is a Transformer-based LLM trained natively with ternary weights, each weight  $w$  in  $\{-1, 0, +1\}$ [1]. Conventional LLMs use 16-bit or 32-bit floats, but BitNet’s 3-state weights give an effective precision of  $\log_2(3) = \sim 1.585$  bits[4]. In practice, weights are stored in 2 bits each (with one unused combination) to allow 3 values[10]. The term “1.58-bit” reflects the information content per weight, highlighting the **extreme compression** achieved[11][12].

Despite this radical quantization, BitNet models up to a few billion parameters have shown comparable NLP task performance to full-precision baselines[2]. For example, a 2.7B-parameter BitNet matched 16-bit LLaMA’s perplexity and even surpassed it on certain benchmarks[13][14]. This is accomplished through architecture tweaks and training methodology:

- **BitLinear Layers:** The standard fully-connected layers are replaced by BitLinear layers that operate with 1.58-bit weights natively[15]. Rather than training a full-precision model then quantizing, BitNet is trained from scratch in ternary weight space, avoiding post-training quantization degradation[16]. During inference, multiplications by  $\{\pm 1, 0\}$  are implemented as conditional addition or skipping, which is more hardware-efficient[5].
- **Activation and Layer Normalization:** BitNet uses **RMSNorm** (root mean square layer norm) instead of traditional **LayerNorm**, which is more numerically stable for low-precision weights[17]. Activations remain in higher precision (e.g. 8-bit or 16-bit) but are dynamically quantized during the bit-linear operations[6]. Crucially, outputs of nonlinearities are centered around zero without requiring a zero-point offset[18], the activation range is symmetric, simplifying representation of 0.
- **No Bias and Custom Activation:** All bias terms are removed to reduce quantization error accumulation[17]. The activation function in feed-forward layers uses **SwiGLU**, which tends to work better with low-bit weights by avoiding extreme activation ranges[17]. Rotary positional embeddings are used in self-attention, which integrate smoothly with quantized weights[17].
- **Ternary Weight Encoding:** A simple encoding is used: for each weight, 00 (in 2-bit) represents -1, 01 represents 0, and 10 represents +1 (the combination 11 is unused or could serve as padding)[19]. By subtracting 1 from the 2-bit value, the model obtains the actual weight: e.g. 2 (10b) - 1 = +1 and 0 (00b) - 1 = -1[20]. This encoding allows 4 weights to be packed into a single byte for storage and computation efficiency[21].
- **Training Recipe:** BitNet models are trained with large token datasets (e.g. 2–4 trillion tokens for a 2B parameter model) to compensate for the reduced capacity per weight[15]. Optimizers and hyperparameters are tuned to this low-bit regime. The result is a new scaling law indicating that, for a fixed compute budget, it can be more efficient to train in low precision from the start[1][22].

Notably, **Hugging Face demonstrated fine-tuning an existing FP16 model down to 1.58-bit** using a gradual quantization strategy[23]. They fine-tuned LLaMA-3 8B models to ternary weights by slowly introducing quantization during training, achieving strong downstream task performance[24]. This suggests that even if one doesn't pre-train from scratch in 1.58-bit, it's possible to adapt a pre-trained model to BitNet format via fine-tuning[23]. In both cases, the key to good performance is ensuring the quantized weights and corresponding low-precision arithmetic are accounted for during the training process (e.g. using straight-through estimators for gradients through quantization).

**Implications:** A 1.58-bit model is incredibly memory-efficient. For instance, a 2 billion parameter BitNet requires on the order of 0.4 GB for weights (2 bits/weight) versus 4+ GB in FP16. Execution can leverage integer addition: multiplying by +1 or -1 is just a sign flip, and by 0 is an ignore operation[25]. In fact, BitNet's matrix multiplications are performed with **INT8 accumulation** (adding INT8 partial sums) instead of FP16 multiply-accumulate[26]. This yields up to 71x lower energy per operation on appropriate hardware[7]. These characteristics make BitNet highly attractive for edge devices where memory and power are limited.

This proposal leverages the above: we will fine-tune a theoretical BitNet LLM for a smartphone GPU (Mali G78 based, as this is the SoC I am most familiar with). The extreme compression lets a large model fit in mobile memory, and the integer-heavy computation is well-suited to mobile GPUs or NPUs. However, **challenges** include implementing custom GPU kernels for ternary-weight matrix ops and handling backpropagation with quantized weights (likely maintaining a higher-precision shadow for weight updates, as common in binary/ternary network training). We use llama.cpp as the software backbone to orchestrate the model and quantization.

### 1.3 OVERVIEW OF LLAMA.CPP AND GPU BACKEND LIMITATIONS

**llama.cpp** is an open-source C/C++ library and runtime for LLM inference that gained popularity by enabling LLaMA and other models to run on consumer hardware. It uses the ggml library which provides tensor operations optimized for CPU (with AVX/NEON) and supports various quantization schemes down to 4-bit or even 2-bit in recent versions[27]. Notably, the latest llama.cpp supports **int4, int3, int2, and even “1.5-bit” quantization** formats for model weights[27]. (The 1.5-bit quantization refers to a variant of the 2-bit scheme that effectively encodes 3 values, similar to BitNet’s ternary encoding.) This indicates a foundation for handling 1.58-bit weights, is already present in the codebase. The project’s manifesto is to enable LLMs on a wide range of hardware with minimal dependencies[9]. To that end, llama.cpp has introduced **GPU acceleration backends** beyond just CPU:

- **CUDA backend (NVIDIA):** Custom CUDA kernels exist for certain operations to speed up inference on NVIDIA GPUs[27]. These handle dense matrix multiplications for quantized weights, offloading the heavy compute from CPU.
- **OpenCL & SYCL backends:** For broader device support (including AMD, Intel GPUs), OpenCL and SYCL implementations were added. Qualcomm even released an optimized OpenCL backend for Adreno GPUs to run llama.cpp models on Snapdragon SoCs (Adreno) efficiently[28]. This indicates a trend to support mobile GPUs, though Mali-specific optimizations have not been highlighted publicly.
- **Vulkan backend:** Vulkan support was added to target any GPU with Vulkan drivers, including integrated and older GPUs without good OpenCL/CUDA support[27]. Community projects like **VulkanLLM** built on this to achieve impressive speedups on iGPUs and older cards via Vulkan compute shaders[29][30]. For example, offloading a 1.1B model to an Intel HD iGPU gave a 33x speedup over CPU-only inference[30]. This backend is highly relevant for Mali GPUs, since Mali supports Vulkan compute and is commonly used in Android graphics.

Despite these backends, **llama.cpp currently focuses on inference**. There is no built-in support for training or fine-tuning loops, i.e., no computation of gradients or weight updates in ggml as of now. Limitations we face include:

- **No Existing Training Pipeline:** We will need to implement the backpropagation operations (gradient calculation for linear layers, optimizer updates, etc.) essentially from scratch within llama.cpp’s framework. This involves extending ggml with new operations

for the backward pass. For example, computing the gradient of a quantized matmul with respect to the weight matrix or input activations is not present and must be added.

- **Quantization and Backprop:** llama.cpp can load quantized models and run them, but fine-tuning requires either (a) converting weights to a differentiable form (e.g. float shadow weights) during training and re-quantizing, or (b) computing gradients in the quantized domain via straight-through estimation. The implementation strategy must carefully handle how gradients flow through the quantization function. We anticipate maintaining a parallel array of F32 or F16 weights for each quantized weight for the purposes of gradient accumulation, as is common in quantization-aware training.
- **GPU Memory and Precision Constraints:** The Mali-G78 is a mobile GPU with no discrete VRAM, it shares LPDDR memory with the CPU. This means lower memory bandwidth (~51.2 GB/s on Exynos 2100's G78 MP14) and higher latency than desktop GPUs. Also, while the G78 supports FP16 and INT8 arithmetic, it does not have specialized tensor cores. Thus, our Vulkan compute shaders must be optimized for the **Valhall architecture** (Mali G78's microarchitecture) to achieve high occupancy and throughput. We cannot rely on vendor BLAS libraries (none exists for 2-bit), so custom kernels are mandatory.
- **Threading and Work Distribution:** On CPU, llama.cpp uses multiple threads to parallelize matrix multiplications across cores. With a GPU backend, we offload these ops to the GPU, but we must ensure the CPU and GPU work asynchronously to hide latency. Also, the current Vulkan integration may primarily accelerate the forward pass. For training, the sequence of operations is more complex (forward → backward → update), requiring synchronization points. Efficient scheduling (e.g., overlapping data transfers or using multiple Vulkan command buffers) will be needed to keep the GPU busy and not stall waiting on CPU.
- **Precision vs Performance:** Using **INT8 additions** for accumulation (as BitNet does) is efficient, but care must be taken to avoid overflow in accumulation of many terms. The 1.58-bit weight matrix multiplication of dimension  $K$  adds up  $K$  terms; if  $K$  is large (say 4096), summing 8-bit values could exceed 8-bit range. The HF BitNet implementation accumulates into 32-bit integers for safety[31], then converts to FP16 for the next layer. Our kernels will likewise use 32-bit or at least 16-bit accumulators to maintain numerical fidelity. The llama.cpp GPU backend must therefore support mixing precisions (INT8 arithmetic with INT32 accumulate), luckily Vulkan's shader model and the SPIR-V shader INT8 capability allow this (we can declare 8-bit types and perform 32-bit accumulations).

In summary, llama.cpp gives us a solid starting point with model file formats, quantization support, and some GPU infrastructure. But **to enable fine-tuning** on Mali G78, we need to extend it significantly: adding training logic and ensuring the **Vulkan backend is tuned for Mali**. The upcoming sections detail the design of the fine-tuning kernels and the modifications required in llama.cpp.

## 1.4 DESIGN OF FINE-TUNING KERNELS FOR 1.58-BIT WEIGHTS ON MALI-G78 (VULKAN)

Fine-tuning an LLM involves repeatedly computing **forward passes**, **loss gradients (backward passes)**, and **updating weights**. The computational heart of both forward and backward propagation in Transformers, are **matrix–matrix multiplications** (e.g.  $Z = XW$  for a linear layer forward, or gradient  $\Delta W = X^T \delta$  for backprop). In BitNet, these multiplications involve a matrix of 1.58-bit weights. We design custom **Vulkan compute shaders** to handle these efficiently on Mali-G78:

**4.1 Ternary Weight Representation:** We store weight matrices in a **packed 2-bit format** in GPU memory. As noted, 4 ternary weights are packed into one 8-bit byte[21]. Consider a weight matrix  $W$  of shape  $(N_{out} * N_{in})$  (for a linear layer with  $N_{in}$  inputs and  $N_{out}$  outputs). We reshape this as a packed matrix  $W_{pack}$  of shape  $(N_{out} * \lceil \frac{N_{in}}{4} \rceil)$  with U8 elements. Each byte encodes 4 original weights  $w_{i,j...j+3}$  where the bitpair at position  $2 * k$  represents  $w_{i,j+k}$ . The mapping of bit pair to weight value is as described: 00 → -1, 01 → 0, 10 → +1, 11 (unused, can default to +1 or 0)[19]. This scheme allows coalesced memory access and vectorized loads of multiple weights at once.

**4.2 Forward Pass Kernel (MatMul with 1.58-bit W):** We implement the operation  $Y = X * W^T$  (for forward propagation of a fully connected layer or the projection in self-attention) where:

- $X$ : input activation matrix of shape  $(B * N_{in})$  for batch size  $B$  (in training,  $B$  could be >1 for efficiency; for inference  $B = 1$ ).
- $W^T$ : weight matrix of shape  $(N_{in} * N_{out})$  (we prefer  $W^T$  in shape to align memory contiguity with how we pack  $W$ ).
- $Y$ : output matrix  $(B * N_{out})$ .

We allocate  $W_{pack}$  as described (dimensions  $N_{out} * (\frac{N_{in}}{4})$  bytes). The compute shader uses a **tiling approach** to multiply  $X$  by  $W$  in chunks:

- **Workgroup Tiling:** Each workgroup computes a tile of the  $Y$  output, say of size  $M * N$  ( $M$  rows,  $N$  columns). For example, a tile might be  $(16 * 16)$  outputs. Each thread in the workgroup computes one element or a small block of this tile. The total number of workgroups launched is  $\lceil \frac{B}{M} \rceil * \lceil \frac{N_{out}}{N} \rceil$  to cover the whole output matrix.
- **Shared Memory (LDS) Usage:** Within a workgroup, we load portions of  $X$  and  $W_{pack}$  from global memory into faster shared local memory. This is analogous to block matrix multiplication: we divide the inner dimension  $N_{in}$  (and its packed form length  $\frac{N_{in}}{4}$ ) into chunks of size  $K_{tile}$ . For each chunk:

- Load an  $M * K_{tile}$  sub-tile of  $X'$  into shared memory (these  $X$  values are standard activations, stored as e.g. FP16 or INT8 depending on our quantization of activations).
  - Load an  $(N_{tile, pack} * K_{tile, pack})$  sub-tile of  $W'_{pack}$  into shared (where  $K_{tile, pack} = \frac{K_{tile}}{4}$  in bytes, and  $(N_{tile, pack} = N)$  in this iteration since we focus on one output tile).
- **Decode weights:** Each thread then decodes the packed bytes to actual weight values as needed for computation. We vectorize this decode e.g., load 32-bit chunks (= 4 bytes) which contain 16 weight values. Using bit operations in SPIR-V, we extract bit fields. In our Triton prototype for reference, we used a loop over 4 sub-bytes (because  $4*2$  bits = 8 bits per weight)[10]:
  - For each 2-bit position in the byte, apply a mask and shift to get values 0,1,2,3[19].
  - Subtract 1 to map to -1,0,1,2[20] (with 2 representing the unused code if it ever appears).
  - Convert to INT8 or INT32 for arithmetic.
- **Compute:** Now we have a tile of  $X$  values (say FP16) and corresponding weights (as  $INT8 \in \{-1,0,1\}$ ). We perform a matrix multiply on these small tiles. Each thread accumulates into a 32-bit accumulator for its output element. For example, thread  $(m, n)$  in the tile does:  $acc[m, n] += sum_{\{k = 1..K_{tile}\}} (X_{shared}[m, k] * W_{val, shared}[k, n])$ . Since  $W_{val}$  is -1,0, or 1, this effectively adds or subtracts  $X_{shared}[m, k]$  or does nothing. We leverage the GPU's integer ALUs for these additions. The Mali-G78 can handle integer ops efficiently and even has specialized instructions (SIMD) for vector integer arithmetic; by using INT8 and INT32, the JIT compiler for Valhall will likely utilize the I8 dot product capabilities if available. (Arm's recent architecture has an I8 dot instruction, but it may be mostly in CPUs; for Mali we rely on standard integer pipelines.)
- We loop over enough chunks to cover the full  $N_{in}$ . Each iteration uses barrier synchronization to ensure previous partial results are fully computed before loading the next chunk from global memory. This tiling minimizes global memory traffic, each part of  $X$  and  $W$  is read once and reused for multiple arithmetic ops, exploiting data locality[32].
- **Writing Results:** After accumulating over all chunks of the inner dimension, each thread stores its final result to the output matrix  $Y$  in global memory. We convert the 32-bit accumulator to the desired output format. In forward inference, we might convert to FP16 (the next layer expects FP16 input)[33]. In training, for the forward pass we keep outputs in FP16 for subsequent layers and also preserve a copy if needed for backward computations.

This kernel essentially implements an  $INT8 * INT2 \rightarrow INT32$  dot product in a loop. The approach is similar to the Triton kernel HF wrote, which showed comparable performance to PyTorch's half-precision routine[34]. By carefully packing and processing 4 weights per byte, we achieve memory savings and vectorized bit-extraction.

**4.3 Backward Pass Kernels:** For each forward matmul, we need two backward operations: one to propagate gradients to the inputs, and one to compute gradients for the weights.

- **Gradient w.r.t. Inputs (Activation Gradient):** Given the upstream gradient  $\delta Y$  for the outputs (shape  $B * N_{out}$ ), we compute  $\delta X = \delta Y * W$  (since if  $Y = X * W^T$ , then it means  $\delta X = \delta Y * W$  for backprop). This is another matrix multiply:

$$\delta X(B * N_{in}) = \delta Y(B * N_{out}) * W(N_{out} * N_{in}).$$

Here,  $W$  is still in ternary form. We can **reuse the same kernel design** as forward, with  $X \leftarrow \delta Y$  and using  $W$  (transposed in code if needed). Essentially, it's an INT2 weight matmul again. The difference is  $\delta Y$  will be a higher precision (32-bit float or 16-bit) but we can downcast it to 8-bit if we choose an 8-bit representation for activations in the backward pass to optimize (though gradients may have wider dynamic range, so keeping them in 16-bit might be safer). We accumulate into 32-bit and produce  $\delta X'$  in (say) FP16.

- **Gradient w.r.t. Weights:** Given  $\delta Y$  and the input  $X$ , the gradient for weights is  $\delta W = \delta Y^T * X$  (shape  $N_{out} * N_{in}$ ). Expanding  $\delta Y^T$  is  $(N_{out} * B)$  and  $X$  is  $(B * N_{in})$ . This is again a matrix multiplication. But now both operands are higher precision (activations and output grads are floats). We will accumulate into a full precision  $\delta W'$  matrix (F32 elements), since we intend to update weights in a continuous space before re-quantizing. The same tiling strategy can apply, but we don't have ternary weights in this multiplication, it's dense FP16/FP32 multiply-add. We can still write a Vulkan shader for this or possibly use a pre-existing library for dense GEMM. However, given  $N_{out}$  and  $N_{in}$  might be large (thousands), a custom Vulkan kernel might still be needed for performance unless we offload to the CPU for weight gradients (which could be slow). We will likely implement a Vulkan kernel that multiplies  $\delta Y^T$  by  $X$  chunk by chunk. Note this is a different kernel than the ternary one: here both matrices are non-quantized, so we can use standard 16-bit floating multiply-add. The Mali GPU can handle FP16 multiplications at possibly higher throughput than FP32 (Valhall has native support for FP16 ALU).

After computing  $\delta$  was a full-precision matrix, we will reduce or quantize it when applying the weight update (see Section 5 for the strategy). We might accumulate  $\delta W$  across a batch (if using mini-batch gradient descent) before applying the update.

All the above kernels are written in GLSL or HLSL to compile to SPIR-V, or directly in SPIR-V using the Vulkan compute pipeline. We will exploit Vulkan features such as:

- **INT8 Arithmetic Support:** We enable the device feature shader INT8 (core in Vulkan 1.2) so we can use 8-bit types in shaders[35]. We also use the 8-bit and 16-bit storage features to allow buffers of those types. This is crucial for using U8 for weight storage and INT8 for arithmetic on partial results.
- **Subgroup Operations:** Mali G78 warps have 16 threads[36]. We can use Vulkan subgroup operations (like subgroupShuffleXor, etc., via GL\_KHR\_shader\_subgroup extension) to implement cross-lane summations or data shuffles if it helps. For instance, to sum up 16 values within a warp, a tree reduction using subgroup shuffle can be faster than iterative adds. We may use this in the final stage of accumulating a tile or for

reduction in weight gradient computation. The Valhall ISA provides instructions for cross-lane permute (CLPER) to facilitate this[37][38]. We must ensure our shader compiler takes advantage of it when using subgroup intrinsics.

- **Memory Access Patterns:** Global memory accesses are optimized by reading in aligned 32-bit or 64-bit chunks. For example, each thread might read a 32-bit word from  $W_{pack}$  containing 16 weights, then each thread handles the decode of those 16 values (or a subset) to accumulate into its outputs. Similarly, we align  $X$  loads. We also ensure that our workgroup size times load vector width equals a whole cache line to avoid serialization. The Mali-G78, being a tile-based GPU, benefits from sequential access; our packing ensures weight bytes for consecutive  $N_{in}$  indices are contiguous in memory.
- **Compute–Memory Balance:** We choose tile sizes  $(M, N, K_{tile})$  to balance compute and memory. Too large a tile and we may run out of the 128 KB L2 or incur register spilling; too small and we don't fully utilize ALUs. Empirically (drawing from similar GPU kernels), a 16x16 tile with  $K_{tile} = 64$  or 128 is a good start. Each workgroup of 256 threads (if  $256 = 16 \times 16$ , one thread per output) might be heavy; we might use a smaller group of e.g. 64 threads computing an 8x8 output sub-tile each to allow more concurrent groups. The sweet spot will be found via profiling (Section 6).

In essence, the fine-tuning kernels include a **ternary weight matmul shader** and a standard FP16 matmul shader. Both will be heavily optimized for Mali's warp=16, many-core parallelism. The correctness of these will be tested on a single layer (PoC in Section 14) before integrating into the training loop.

## 1.5 MODIFICATION STRATEGY FOR LLAMA.CPP TO ENABLE MALI-G78 GPU FINE-TUNING

To incorporate the above kernels and training mechanism, we must extend llama.cpp (and underlying ggml) significantly. Our strategy:

**5.1 New Quantization Type:** llama.cpp uses quantization schemes like Q4\_0, Q4\_K, Q8, etc., to encode model weights. We will introduce a **Q2\_ternary** (or call it Q1.58) quantization type to represent weights with 2 bits (ternary). If recent versions already have a similar type (they mention 1.5-bit support[27]), we will build upon that. This entails defining how weights are packed in the .gguf file and memory, likely similar to the 2-bit block quantization but ensuring only 3 distinct values are used. The model conversion scripts will need to quantize pretrained weights to this format (if fine-tuning from a FP model). For training, we also need the reverse: the ability to dequantize or apply quantization on the fly when updating weights.

**5.2 Data Structures:** In ggml, each tensor (model weights, activations, etc.) has a storage type. We will add support for a tensor type representing 2-bit packed data. The forward pass routines will recognize this type and route to our custom kernels. For example, currently ggml\_compute\_forward\_mul\_mat() handles matrix multiplication. We will add a case: if the weight tensor is Q2\_ternary and backend is GPU, call ggml\_vulkan\_mul\_mat\_q2() which triggers our Vulkan compute shader instead of the default CPU path.

Additionally, for training we introduce gradient tensors and optimizer state. We might extend the ggml\_context to hold gradient buffers for each trainable weight. Each weight tensor could have a pointer to a gradient tensor of same shape (but in full precision). Alternatively, we manage gradients outside ggml for simplicity, since we can manually call our backward kernels and then update weights in code.

**5.3 Vulkan Backend Integration:** We will leverage the existing Vulkan backend scaffolding in llama.cpp[39]. This likely provides us with a Vulkan VkDevice, command pool, descriptor set layouts, etc., to run certain operations on GPU. We will insert our new compute shaders into this framework:

- Write GLSL for ternary\_matmul.comp and fp16\_matmul.comp and precompile them to SPIR-V (using glslangValidator) as part of the build. At runtime, create VkShaderModule for each.
- Extend the ggml\_vk code: when scheduling a graph of operations, include the new ops. We may create a function ggml\_vk\_schedule\_mul\_mat\_q2(vk\_context, src0, src1, dst) that records the dispatch of our shader with the correct specialization constants (matrix dims, tile sizes) and descriptor bindings for the buffers. The weight buffer (src1) will be bound as a VkBufferView with format R8\_UINT so we can read bytes easily. The activation (src0) can be bound as FP16 or FP32 buffer. The output (dst) as FP16 or FP32.
- We ensure that all needed Vulkan features are enabled at device creation: e.g., VkPhysicalDeviceFeatures for shader INT8, shader INT16, subgroup operations, etc. The Mali-G78 supports Vulkan 1.1/1.2, and we will require at least 1.1 with the extensions for 8-bit storage. If using an Android device, we might need to check the Vulkan version (most devices with G78 likely support Vulkan 1.1+).
- Memory management: For large models, memory usage is significant but manageable. A 3B param model in 2-bit consumes ~0.75 GB for weights. The Mali-G78 typically has access to up to 6-8 GB of system memory. We allocate Vulkan buffers for each weight tensor (with VK\_BUFFER\_USAGE\_STORAGE\_BUFFER\_BIT). Because the GPU shares memory with CPU, we can map those buffers on the host to initialize them (copying the quantized weights in) and to read back if needed. Gradients and intermediate activations can also reside in such buffers. We might use a staging buffer if copying large data, but since it's unified memory, direct access is fine (just need to flush CPU caches on some architectures).
- **Backprop Implementation:** llama.cpp uses a computation graph for inference. For training, we won't build a static graph easily, as it would need to dynamically incorporate backward ops. Instead, we handle training at a higher level:
- Do forward pass (through each layer, using GPU kernels where possible; store necessary intermediate activations for backward, like the inputs to each layer and perhaps outputs if needed for gradient calcs).
- Compute loss and initial gradient (likely on CPU if loss is simple, or we can code a GPU kernel for say cross-entropy).

- Then for each layer in reverse order, run our backward matmul kernels. For multi-head attention, this includes gradients for  $Q, K, V$  projection matrices and the output projection. For feed-forward, just one linear's weight grad.
- We accumulate weight gradients in full-precision buffers (FP32) on GPU.
- Apply update: For simplicity, initial implementation might do this on CPU: map gradient buffer, do SGD/Adam update on mapped float arrays, re-quantize to nearest ternary and write back to quantized buffer. However, this round-trip CPU may be slow. A more advanced approach: use a **Vulkan kernel for weight update**. We can write a small compute shader that takes the current weight (packed) and the gradient (FP32), and updates an underlying FP32 “shadow weight” plus updates the packed. Alternatively, maintain two GPU buffers per weight: one FP16/32, one quantized. The update kernel reads FP32 weight and grad, performs e.g.  $w_{fp32} -= lr * grad$ , then computes the new quantized value for that weight:
  - We enforce  $weight \in \{-1, 0, 1\}$ . One strategy:
    - if  $w_{fp32} > +t$  then  $weight = +1$ ,
    - if  $w_{fp32} < -t$  then  $weight = -1$ ,
    - if in between  $\pm t$  then  $weight = 0$ ,
    - For some threshold  $t$  (possibly  $t = 0$  or small to avoid flapping). This is effectively rounding to nearest ternary  $\{-1, 0, 1\}$ .
  - Write the appropriate bits to the quantized buffer. (This per-weight logic is divergent, but a single warp could handle many weights; divergence cost is small for 3 branches as long as warp lanes handle different weights.)
  - Also store the updated FP32 back (clamped if needed so it doesn't grow beyond what quantization can represent).
- Using the GPU for weight updates means training can be **almost fully GPU-bound**; the CPU would mainly orchestrate kernel launches and do minimal compute. This is ideal for performance.
- **Control Flow:** We integrate a training loop either as a new mode in llama.cpp or a separate driver. For example, we can add a command llama\_train that takes a dataset and runs epochs. This loop would use the modified ggml to get forward outputs, then invoke backward and update steps. We might not want to fully script this in the ggml graph executor, since it's more straightforward to write an explicit loop in C++ that calls our kernel functions in sequence for each layer. This explicit approach is fine given the relatively small number of layers (~ 32 layers for a 7B model).
- **Maintaining Compatibility:** All these modifications will be guarded by compile-time flags (e.g. GGML\_USE\_VULKAN and GGML\_ENABLE\_TRAINING). If a device doesn't have Vulkan or the user doesn't request training, the code should still run inference normally. We will also provide fallbacks: e.g., if for some reason the Vulkan backend fails or runs out of memory, we can fall back to CPU quantized operations (though slowly). This is important since not all Mali drivers are equal, we want a robust solution.

In summary, we plan to **embed our Vulkan compute pipeline into llama.cpp's workflow**, extending it from an inference engine to a training-capable engine for quantized models. This involves non-trivial development in ggml, but the result will be a single unified framework where

one can load a BitNet model, supply training data, and perform on-device fine-tuning with minimal external dependencies.

## 1.6 PROFILING TOOLS AND PERFORMANCE METRIC COLLECTION FOR MALI-G78

To ensure our implementation meets performance goals and to guide optimizations, we will utilize a suite of **profiling tools** specifically suited to mobile GPUs:

- **Arm Streamline (Mobile Studio):** Arm’s Streamline profiler is essential for Mali GPUs[40]. It can collect both CPU and GPU performance counters on a non-rooted Android device via ADB. We will use the Mali-G78 predefined counter set[41] to monitor:
- GPU Utilization: percentage of time the shader cores are active (helps ensure we keep the GPU busy).
- Warp Occupancy: how many warps are in flight. G78 has 16 threads/warp and multiple warps per core; we can see if our workgroups result in full warps being utilized[42].
- ALU Stalls vs Memory Stalls: counters indicating if shaders are ALU-bound or memory-bound. If memory bandwidth is the bottleneck, we might see high memory load/store unit utilization and idle ALUs, guiding us to optimize memory access patterns.
- Throughput per shader core: The number of arithmetic instructions executed per cycle, etc., to measure if we are nearing hardware limits.
- Bandwidth usage: Mali reports external memory bandwidth. We will see how close we are to the ~50 GB/s theoretical max. If extremely high, we may implement further compression or ensure better data reuse in shaders.

By selecting a timeline region (say one training iteration) in Streamline, we can attribute GPU load to our kernels[43]. If integrated with annotations (Arm provides an API to mark regions[44]), we can label events like “Forward Layer 1” or “Weight Gradient Layer 1” to correlate with GPU activity bursts[45]. This will identify the longest-running stages to focus optimization.

- **Android GPU Inspector (AGI):** Google’s AGI tool also supports Mali GPUs[46]. We will use it to capture per-drawable (per-dispatch in our case) profiling. AGI can show the timeline of Vulkan command buffer execution, the utilization of each queue, and per-shader stats. For example, AGI’s GPU trace can reveal if our dispatch sizes are causing serialization or if multiple dispatches overlap. It also can disassemble the shader to check if the compiler generated efficient code (like using vector operations, etc.).
- **Mali Offline Compiler:** Arm provides an offline shader compiler where we can input our GLSL and see the compiled Valhall instructions and an estimate of cycles. We will use this for micro-optimizations: e.g., adjusting thread group size or loop unrolling and seeing how it affects register usage and instruction count. If the compiler output shows heavy usage of certain instructions (like complex bit ops taking multiple instructions), we might manually optimize (for instance, using bit trick or LUT).
- **Thermal and Power Monitoring:** While not as fine-grained, we will log device temperature and use Android’s BatteryStats or Qualcomm Trepn profiler (if on a device that supports it) to estimate power usage. If our goal is sustained training, we must ensure we don’t throttle severely. We define a target that the GPU can run at least at

50% utilization for 10+ minutes without hitting thermal shutdown. If our initial runs trigger throttling (seen as GPU frequency drop in Streamline's frequency counter or via system logs), we may adjust workload or recommend enabling sustained performance mode in Android (if available via the NDK).

- **End-to-End Metrics:** Apart from low-level profiling, we measure:
  - Iteration time: how many milliseconds per training step (one forward-backward-update) for a given model and batch size. This is the bottom-line performance metric for training speed.
  - Throughput: e.g., tokens processed per second, or examples per second in fine-tuning.
  - Latency: for inference after fine-tuning, how fast can the model generate output on the device (should be similar or slightly worse than original due to added GPU overhead, but quantization keeps it fast).

We will collect these metrics under various scenarios (different batch sizes, sequence lengths, model sizes if possible) to fully characterize performance. For example, running a small batch vs full batch will tell if we're GPU-bound or launch-bound. Also, capturing a profile of **GPU memory usage** is important (AGI can show peak memory used). Mali uses a unified memory, but large allocations might cause paging on some systems if memory is over-subscribed.

In summary, we will rely heavily on **Arm Streamline** for Mali-specific insights (as it provides built-in knowledge of G78 counters)[47] and **AGI** for Vulkan-specific timing and debugging. These tools will guide optimizations described in the next section by pinpointing bottlenecks, whether it's at the shader ALU level or memory interface.

## 1.7 REQUIRED FRAMEWORKS, LIBRARIES, AND SDKS

Developing and deploying this solution involves various tools and libraries:

- **llama.cpp / ggml (modified):** The core C/C++ framework that we will modify for training support. We will use the latest version of llama.cpp as a base[9], which already supports quantization and Vulkan. Our contributions (training loop, new kernels) will build on this.
- **Vulkan SDK 1.2+:** The Vulkan Software Development Kit (by LunarG) is required for development. It provides header files, library loaders, and debugging layers. We will write Vulkan API code to manage devices, queues, command buffers, descriptor sets, etc. The Vulkan SDK will also be used to compile shaders offline (via glslangValidator) and to run validation layers during testing to catch any API misusage.
- **Android NDK (for smartphone deployment):** To run on Mali-G78 devices (which run Android OS), we will compile our C++ code with the Android NDK (targeting say API level 30 for Android 11). The NDK provides cross-compilers for ARM64 and Vulkan support. We will likely create an Android test application or binary (via ADB) to launch the fine-tuning process on the phone. This requires linking against Vulkan (which on Android is a system library, accessible via libvulkan.so). The NDK's android.hardware.vulkan.level should be at least 1 (meaning Vulkan 1.1 available).

- **Arm Performance Libraries (optional):** While our primary compute is on GPU, the CPU still might do some work (like softmax in attention or layer norm gradients). For completeness, we may use Arm Compute Library or Eigen for any CPU fallback. However, to keep dependencies minimal (a llama.cpp philosophy[9]), we likely implement required CPU ops in ggml itself using NEON intrinsics if needed.
- **BitNet Model and Converter:** We need access to a pretrained BitNet model or to quantize a LLaMA model to 1.58-bit. Microsoft has released open weights for BitNet b1.58 (2B parameters)[48] and Hugging Face has 8B fine-tuned models[24]. We'll obtain one as our starting point. A conversion script (possibly using PyTorch) will convert the model to ggml's format (with our Q2\_ternary quantization). This script will use the **Transformers** library to load the model and then use our quantization method to write a .gguf file. So, Python (with transformers and bitsandbytes or a custom routine) is needed offline for model preparation.
- **Development Environment:** We will use CMake to build the C++ code for both desktop (for initial testing) and Android. On Android, we may leverage Android Studio for convenience or a simple adb to push binaries. For profiling, we install Arm Mobile Studio tools on a host PC and connect to the device via ADB.
- **Libraries for Vulkan on Desktop:** To expedite development, we will also ensure our Vulkan code runs on desktop GPUs for debugging. This may involve linking with SDL or GLFW if we needed a dummy window (not strictly necessary for compute). The Vulkan loader (through the SDK) and validation layers will be used on desktop for easier debugging. Desktop GPU (like an AMD or Intel iGPU) can mimic the behavior; we specifically might use an Intel iGPU since it also has suboptimal memory like Mali, but any Vulkan device works for functional testing.
- **Profiling Tools:** As discussed: Arm Streamline (available in Arm Mobile Studio) for which we need to integrate the device (installing the companion daemon on device if needed), and Android GPU Inspector (just a client tool on PC plus an agent via GPU driver on device). These aren't libraries but important tools.
- **Vulkan Extensions/Utilities:** We might incorporate the open-source **Vulkan Memory Allocator (VMA)** library to simplify memory allocation (it helps managing device vs host memory, alignment, etc.). This is header-only and will not bloat dependencies.
- **SDKs for ML on device (optional):** If we wanted to consider the phone's NPU, we might look at NNAPI or vendor SDKs. However, integrating with those is complex and out of scope, we stick to Vulkan.
- **IEEE Floating-point Math (FP16):** We rely on GPU's support for FP16, which Mali-G78 has (it supports FP16 ALU at half-rate or full-rate for some operations). We enable VK\_KHR\_shader\_F16\_INT8 to use 16-bit floats in shaders. Also, for safety, we ensure VK\_KHR\_16bit\_storage so that our FP16 activation buffer can be used in storage class.
- **C++17/20 Compiler:** We will use modern C++ (for convenience in writing the host logic), the NDArray structure of ggml is C, but our integration code can use C++ utilities.

In short, the project requires a cross-section of **systems programming tools (Vulkan, NDK)** and **machine learning frameworks (Transformers/PyTorch for data preparation, llama.cpp for runtime)**, plus **Arm's toolchain for profiling and possibly optimization libraries**. All chosen components are open-source or freely available, aligning with the goal of a fully open and reproducible solution.

## 1.8 OPTIMIZATION STRATEGIES AT THE VULKAN SHADER AND KERNEL LEVELS

Achieving good performance on a mobile GPU requires fine-grained optimizations. Based on profiling and known GPU optimization techniques, we will employ:

**8.1 Workgroup Size Tuning:** We choose Vulkan workgroup dimensions carefully to match Mali's architecture. As noted, Mali G78 executes warps of 16 threads[36], and can issue multiple warps per cycle per core. We will use workgroup sizes that are multiples of 16 (e.g., 64 or 128 threads total) to avoid underutilized warps. A 16x16 workgroup (256 threads) might be too large (risking register pressure); 8x16 (128 threads) or 8x8 (64 threads) could strike a balance. We will profile these configurations. The Mali Valhall scheduler will try to keep many warps in flight, we want each core to have enough workgroups to hide latency of memory fetches. If our group is too big, we might use fewer workgroups total, hurting latency hiding. So, prefer smaller groups launched in greater quantity if memory latency is a bottleneck.

**8.2 Memory Coalescing and Alignment:** We ensure that memory accesses in our shader are coalesced: - For weight matrix loads, each thread reads a U8(or better, a UINT32 containing 4 packed bytes) such that consecutive threads read consecutive addresses. This way, the memory system can service them in one transaction. We align  $W_{pack}$  rows to 4 bytes. Our shader will use OpLoad with an INT32 type to load 4 bytes at once (exploiting that each contains 16 weights)[10]. The mask/shift operations then pick out the needed two-bit fields from these 32-bit chunks.

- For  $X$  loads (activations or gradients), we will similarly use 128-bit or 64-bit vectors if possible (packing multiple values). If  $X$  is FP16, we can use f16vec2 (16-bit \* 2) to load two values at once if it aligns.
- Shared local memory is used as much as possible to reuse values. We load a tile of  $X$  and  $W$  into shared and synchronize, then reuse them for multiple multiply-accumulate operations[32]. This dramatically reduces redundant global reads. We choose tile sizes to maximize this reuse given Mali's 128KB L2 and maybe around 32KB L1 per core, our shared memory tile should be small enough to fit in L1 to get low latency.

**8.3 Instruction-Level Optimization:** We rely on the shader compiler to a degree, but we can guide it by:

- **Loop unrolling:** Unroll the fixed-size loops. For example, the loop over  $i=0..3$  for bit extraction (since exactly 4 sub-values per byte) we will fully unroll[10]. Also, the inner dot-product loop over a small  $K_{tile}$  can be unrolled or partially unrolled if it's a fixed constant (by using specialization constants for block sizes, the compiler can unroll known-size loops).

- **Avoid branch divergence:** Use integer arithmetic to handle conditional logic for weights. For example, instead of if(weight==0) skip, we can multiply by a mask (weight being 0 will zero out the product). Our design already does this: weight values are  $\{-1,0,1\}$ , and the multiply-add naturally yields no contribution for 0 without an explicit branch (if we code it as  $(b\_val - 1)$  as in HF code,  $1 - 1 = 0$  for original code 1)[31]. For activation functions or gradient clipping, if needed, we use vectorized min/max rather than branching.
- **Minimize precision conversion:** Keep values in int where possible. For example, in accumulation we do  $INT8 \rightarrow INT32$  adds; only after summing the whole dot product do we convert to F16 for storage[33]. This avoids repeatedly converting in the inner loop.
- **Subgroup reduction:** We might sum partial results across threads in a warp to reduce memory writes. Suppose each thread computed 4 partial sums for an output element due to splitting K. We can use a subgroup add to combine them and have, e.g., one thread write the final result. However, in our design each thread already computes a full output element's sum, so this might not apply unless we change the work assignment. We will consider having each thread compute two outputs and then reduce across two threads if it helps with vectorization.

**8.4 Kernel Fusion and Overlap:** If profiling shows significant overhead in launching many small kernels (which on Vulkan/Android can be non-negligible due to driver scheduling), we consider **fusing some operations**.

- For example, in backprop through a Transformer layer, we have:  $\delta X_{f,fn} = \delta Y * W_{f,fn}$  and  $\delta W_{f,fn} = \delta Y^T * X$ . These share  $\delta Y$  and  $X$ . We could in principle do these in one combined kernel that computes both  $\delta X$  and  $\delta W$  to avoid reading  $\delta Y'$  and  $X$  twice from memory. This would be a larger kernel (more registers), but if memory bandwidth is limiting, fusion could help. We can allocate output memory for both and do atomic adds if necessary (though better to avoid atomics by careful thread assignment).
- We will also pipeline layers: start computing layer 2 forward while layer 1 backward is running, etc., to overlap as long as dependencies allow. This could be done by using multiple Vulkan command buffers or asynchronously launching next forward before previous backward fully done (though backprop has strict sequential dependency, so overlap is mostly between different parts like attention vs feedforward computations).
- Overlap CPU and GPU: Let GPU handle matmul and heavy ops, while CPU computes things like layer normalization gradients or small elementwise ops in parallel. Since Mali G78 is independent of CPU, this concurrency can improve throughput.

**8.5 Utilize Mali-specific features:** If documentation or experimentation shows that certain vector widths are optimal (Mali might have 128-bit vector registers), we will adjust data types accordingly. Also, if the G78 has special instructions (e.g., a dot product that can do 4 INT8 multiplies and accumulate to INT32 in one instruction), we will attempt to pattern-match that by structuring our code (some compilers will emit such for  $INT8 * INT8$  dot with certain shapes). Arm's I8ML (INT8 matrix multiply) is mostly a CPU feature (Neoverse cores)[49], but the GPU might do similar things for 8-bit.

**8.6 Reducing Memory Footprint:** While not exactly speed, memory optimizations can indirectly help performance by avoiding swaps. We will:

- Free intermediate buffers as soon as not needed in backprop (or reuse buffers from forward for backward where safe).
- Use half precision for activations and gradients when possible, to halve memory bandwidth. If we find gradient values fit in FP16 range without overflow, we'll use FP16 for them too.
- Quantize activations to 8-bit on the fly in forward (this is what BitNet inference does[35]). For training, maybe keep them in 16-bit for gradient accuracy. But if memory bandwidth is a problem, we might try INT8 activation even in backward (with careful scaling), though that could hurt convergence.

**8.7 Command Buffer reuse:** On Vulkan, we can record command buffers once and reuse them each iteration (if the sequence of operations is identical). We will explore recording the entire forward-backward as a secondary command buffer and just re-executing it each training step, updating descriptor contents as needed. This avoids CPU overhead of re-recording and could improve throughput.

Through these strategies, we target to maximize utilization of the Mali-G78's **ALUs and texturing units** (if any used for memory) and minimize idle cycles. The end goal is that our shader achieves near peak theoretical throughput: e.g., if G78 has X INT8 ops per cycle per core, we want to come close given enough workload. Achieving this will likely require multiple iterations of tuning guided by the profiling tools from Section 6.

## 1.9 EVALUATION BENCHMARKS AND PERFORMANCE SUCCESS CRITERIA

We define both **functional** and **performance** benchmarks to evaluate success:

### Functional Correctness & Quality:

- Per-layer numerical check: We will verify that our forward and backward kernels produce results matching a high-precision reference. For a given input and weight, the Vulkan forward output should match (within tolerance) a CPU computation of  $X * W^T$  using the ternary weights. Similarly, gradients from our backward pass should align with finite-difference approximations or PyTorch's autograd on an equivalent quantized model. These tests ensure our GPU kernels are correct.
- End-to-end fine-tuning quality: We will fine-tune a BitNet model on a known downstream task (e.g., a small QA dataset or summarization task) on the device. After training, the model's evaluation metrics (accuracy, BLEU, etc., depending on task) should improve and approach the quality obtained by fine-tuning the same model on a desktop. For example, if fine-tuning a 7B model on SQuAD yields 85 F1 off-device, our on-device fine-tuning should achieve results close to that. A significant drop would indicate training instability, possibly due to low precision issues.

## Performance Metrics:

- Throughput (tokens/second): We will measure how many tokens (or training samples) per second the device processes during fine-tuning. For instance, if using sequence length 128 and batch size 1 (for simplicity), and it takes 0.5 seconds per iteration (forward-backward), that's 2 sequences per second, i.e., 256 tokens/sec. We aim to maximize this. Success might be defined as exceeding a certain threshold, e.g., >100 tokens/sec for a 2-3B model, which would be a practical training speed (many thousands of tokens per hour).
- Latency (per inference): For completeness, measure the time per token for inference after fine-tuning. Since our changes primarily affect training, inference should remain very fast (possibly faster than before due to GPU use). If a CPU-only 7B quantized model generated at 5 tokens/sec, the GPU-accelerated should do perhaps 20+ tokens/sec. We expect the **GPU Vulkan inference to be ~3-10x faster** than CPU, based on VulkanLLM results[50]. We'll set concrete targets after initial profiling.
- Utilization: Using the profiling tools, we expect to see high utilization numbers: - GPU core active > 80% during heavy compute phases.
- Memory bandwidth usage close to the theoretical peak (if we are bandwidth-bound, at least we saturate it).
  - If utilization is low, that implies inefficiencies (or CPU bottleneck) to fix.
- Thermal stability: We will run a prolonged fine-tuning session (e.g., 30 minutes) and observe if the device throttles. A success criterion could be “the device maintains at least 70% of initial throughput after 30 minutes of continuous training”. If we see a dramatic slowdown due to thermal throttling, we may need to adjust power management or accept a lower sustained speed. Ideally, with the efficiency of 1.58-bit operations (71x less energy per op for matmul as claimed[7]), the thermal load might be surprisingly manageable, but we need to confirm.
- Benchmarks vs Baselines: We compare our approach to two baselines:
- **CPU-only fine-tuning** on the same device (if possible) using 4 or 8 threads. This will be extremely slow for large models but gives a baseline.
- **Desktop fine-tuning** on a GPU or CPU: to ensure our on-device is within a reasonable factor. We might find our mobile GPU approach is, say, 10x slower than an Nvidia 3090, which might be acceptable given the constraints. But if it's 100x slower, that might be impractical. A qualitative success is if a small fine-tuning (e.g., 1000 training steps) can be completed on device in minutes to an hour (not days).
- Memory footprint: We ensure the memory usage stays within device limits. For a target device with 8GB RAM, we set a criterion that our training process uses <6GB to leave room for OS. If a specific model is too big (e.g., 13B parameters ~ 2.6GB quantized, plus overhead), we would note that and possibly restrict it to 7B or 2B models on 8GB devices. We should demonstrate at least one reasonably sized model (like 2-3B) can fully fine-tune in <4GB, making it feasible on mid-range phones.

**Benchmark Tasks:** We will fine-tune on:

- A language modeling continuation task (to measure loss decrease per token).
- Possibly a multiple-choice QA (to measure accuracy improvement).
- Use the **MMLU or MLPerf Tiny** benchmark to have a standard measure of quality if fine-tuned appropriately (or at least run inference on those to see if model quality holds after fine-tuning).

**Performance Success Criteria:** In summary, we declare success if:

- The fine-tuning process runs to completion on a Mali-G78 phone **without crashes or out-of-memory**, and without severe throttling.
- The model after fine-tuning shows intended accuracy improvements (indicating the training was effective, not broken by quantization).
- The throughput is high enough to be practical (for example, fine-tuning a model on a small dataset in a couple of hours). Concretely, if we achieve  $>5x$  speedup over CPU-only and process on the order of  $10^5$  tokens per hour, that's a success for a first-of-its-kind mobile LLM training demonstration.
- All results and performance data are documented with references to profiling evidence and, if possible, visualizations of the performance counters to demonstrate how close to optimal we are.

We will tabulate results such as “time per iteration vs sequence length” and “GPU utilization vs batch size” to characterize performance. The proposal’s success is not just raw speed but proving that **fine-tuning a 1.58-bit LLM on mobile is feasible and efficient**, meeting the above benchmarks.

## 1.10 CONSTRAINTS OF SMARTPHONE HARDWARE AND Low-POWER OPTIMIZATION STRATEGIES

Smartphones impose unique constraints compared to desktops or servers. Our design must accommodate:

**10.1 Thermal and Power Constraints:** The Mali-G78 GPU in a phone is limited by the device’s ability to dissipate heat. Running the GPU at 100% will heat up the phone quickly. Thermal throttling can reduce GPU frequency dramatically after sustained load, negating speedups. To mitigate:

- We can operate in a **sustained performance mode** if available. Some devices allow a developer mode where the thermal governor is relaxed (assuming user consents to a hot device).
  - If not, we might insert micro-pauses or operate at slightly reduced utilization to avoid triggering throttling aggressively. For example, perform 90% duty cycle (9ms work, 1ms idle) if it helps maintain a constant frequency.

- Use **efficient arithmetic**: INT8 additions and minimal floating ops mean less power per operation. According to BitNet’s paper, the energy per operation is drastically lower[7]. This should help, our approach inherently reduces power per inference/training step, possibly allowing longer operation before throttling compared to an FP16 approach.
- Monitor device temperature via Android APIs and adjust batch size dynamically. If we see temperature crossing a threshold, we could shrink batch or add delay to cool down, albeit at cost of speed.

**10.2 Memory and Storage Limits:** Smartphones have finite RAM and often no swap. If we allocate too much (e.g., >4GB on an 8GB phone), the OS might kill our process. We must:

- Choose model size appropriate to available RAM (likely up to 7B parameters on 8GB device in 2-bit form). For larger, one could offload some layers to CPU (llama.cpp supports hybrid mode[39]). If needed, we implement a scheme where not all layers reside in GPU memory at once, though unified memory means GPU can access CPU memory, performance would suffer. Instead, we ensure our memory allocations (model + optimizer states + activations) fit in the physical memory comfortably.
- Use half precision for optimizer states: if using Adam, it keeps momentum and variance per weight. Storing those in FP32 for billions of weights is costly (~2x model size). We can try storing in FP16 to halve that. Some quality might be lost but often FP16 is okay for momentum.
- Utilize Android’s **large pages or pinned memory** if possible, for large buffers to avoid fragmentation issues.

**10.3 Compute Capability:** Mali-G78, while modern, is not as powerful as desktop GPUs. It has on the order of e.g. 24 cores and a frequency ~750 MHz. It lacks specialized matrix units, so everything is on general-purpose ALUs. This means performance per FLOP is lower. Our optimizations partly address this, but we acknowledge that fine-tuning a huge model might still be slow in absolute terms. Thus, we might lean on **LoRA or adapter fine-tuning** as a strategy:

- Instead of updating all weights, insert low-rank adapters (in higher precision) and only train those. This massively reduces computation and might be more feasible on-device. However, since the question specifically is about fine-tuning BitNet’s weights, we haven’t assumed LoRA. But it’s a viable extension: one could keep base weights fixed and just fine-tune small side matrices in FP16, far less strain on GPU.
- Another strategy: **Gradient Checkpointing** to save memory (recompute forward for gradients instead of storing all activations). This trades compute for memory, which might heat the device more but lower memory usage. If memory becomes the limiter, we enable checkpointing to drop non-critical activations.

**10.4 OS and API constraints:** On Android, long-running background processes might be killed. We likely need to run as a foreground service to avoid this. The user should keep the phone awake (no deep sleep) during training. We’ll provide an interface (maybe a simple app UI) showing progress, to avoid Android terminating the process for inactivity.

**10.5 User Experience Considerations:** Running an LLM fine-tuning will drain battery quickly if not plugged in. We will recommend the device be charging. Also, the phone might become too hot to touch; we'd advise using a stand or cooling fan if doing extended runs. Low-power optimization in this context might mean trading speed for lower heat if needed. We could provide a setting: “Power-saver mode” that reduces GPU usage (by artificially throttling or using smaller batch) to allow hours-long runs without overheating, versus “Performance mode” that finishes faster but with high battery usage.

**10.6 Network and I/O:** If the training data is large, reading it from storage could be an I/O bottleneck (especially if concurrently using the GPU). We'll ensure data is read in large sequential chunks (and consider using memory-mapped storage). And ideally preprocess the data on a PC into a binary format that the phone can load easily to minimize preprocessing load on CPU.

**10.7 Future Hardware:** It's worth noting that newer mobile GPUs (like Mali-G710 or Apple's GPUs) are even more capable. Our design should be forward-compatible. But also, some SoCs have dedicated NPUs that could potentially handle INT8 matrix multiplies extremely efficiently (e.g., Qualcomm Hexagon or Samsung NPU). While our primary target is Mali GPU, an ultimate low-power solution might involve using such NPUs. However, they often require using NNAPI or vendor SDKs, which don't directly support custom operations like 1.58-bit quant. So our approach with Vulkan is currently the most general on GPUs.

In summary, we'll adopt a **thermal-aware, memory-conscious approach**, sometimes sacrificing a bit of raw speed to ensure the process remains within the operational envelope of a smartphone. The combination of quantization (which inherently reduces power per operation) and careful control of workload should allow us to navigate these constraints and still achieve the fine-tuning objective.

## 1.11 PHASED IMPLEMENTATION PLAN WITH MILESTONES AND DELIVERABLES

Delivering this project requires a phased approach:

### Phase 1: Feasibility Study and Environment Setup (Month 1)

- **Task 1.1:** Set up the development environment: compile llama.cpp for desktop and Android. Verify we can run a small model on the Mali-G78 (e.g., LLaMA-7B 4-bit) using the existing Vulkan backend for inference.
  - Deliverable: A report on baseline inference speed on Mali-G78 and confirmation that the Vulkan device is accessible and working.
- **Task 1.2:** Analyze BitNet model architecture and get a pre-trained weight checkpoint (or quantize one). Write a converter to ggml's format with placeholder for Q2\_ternary.
  - Deliverable: A 1.58-bit quantized model file loaded in llama.cpp (perhaps using CPU fallback since kernels not done yet), confirming model can be instantiated.

## Phase 2: Design & Prototype Core Kernel (PoC) (Month 2-3)

- **Task 2.1:** Develop the Vulkan compute shader for forward matmul with 1.58-bit weights (ternary kernel). Test it on desktop GPU with a small matrix. Then deploy on the phone for a small matrix multiplication.
  - Deliverable: Proof-of-Concept result: Given a random small matrix X and random ternary W, the GPU-computed Y matches CPU reference to 1e-3 tolerance. Include a snippet of shader code in documentation and performance on a setting (e.g., 128x128 matmul time).[10][31]
- **Task 2.2:** Integrate this kernel into llama.cpp's inference path for a single linear layer. Possibly hack one layer's computation to call our kernel.
  - Deliverable: Comparison of that layer's output vs original implementation and measured speed for that layer on Mali GPU.

## Phase 3: Extend to Full Forward Pass (Inference) (Month 3-4)

- **Task 3.1:** Implement support for all quantized linear layers in the model (including Q, K, V, O in attention, and FC1, FC2 in feed-forward). Use the Vulkan kernel for each. Also handle any other ops that could benefit from GPU (maybe softmax, though that's minor).
  - Deliverable: The entire model's forward inference runs on GPU and produces correct language generation output identical to CPU (for same random seed).
- **Task 3.2:** Optimize inference: adjust tile sizes, etc., based on profiling. Possibly reach a milestone like X tokens/sec inference speed, demonstrating improvement.
  - Milestone: **Inference GPU-offload complete**, a demo where the phone generates text, with profiling showing GPU acceleration.

## Phase 4: Backward Kernels and Gradient Infrastructure (Month 4-5)

- **Task 4.1:** Implement the backward Vulkan kernels  $\delta X = \delta Y * W$  with ternary W,  $\delta W = \delta Y^T * X$  with floats. Write and test them in isolation on dummy data.
  - Deliverable: Verified gradient kernels (e.g., compared to numeric grad).
- **Task 4.2:** Introduce data structures in ggml for gradients and possibly optimizer state. Ensure each weight has an associated gradient buffer.
  - Deliverable: A simple unit test where we manually call forward kernel, set a fake  $\delta Y$ , call backward kernel, and verify that applying  $\delta W$  to weights improves a dummy loss.

## Phase 5: Weight Update and Optimizer Integration (Month 5)

- **Task 5.1:** Implement a simple optimizer, likely stochastic gradient descent (SGD) with momentum or Adam. This includes the logic for weight update: using either CPU or GPU as decided. Initially, do it on CPU for simplicity: download gradients, update, re-upload weights.
  - Deliverable: End-to-end weight update working for a single step (weights actually change and subsequent forward reflects it).
- **Task 5.2:** If CPU update is too slow, implement the Vulkan weight-update shader to do quantized update on GPU. Test it.
  - Deliverable: GPU-side weight update validated (e.g., compare one step of CPU vs GPU update yields same new weight bits).

## Phase 6: Full Training Loop & First Fine-Tuning Run (Month 6)

- **Task 6.1:** Integrate everything into a training loop that iterates over dataset batches. Use a small dataset to test. This involves orchestrating *forward* → *loss* → *backward* → *update* for each batch.
  - Deliverable: Fine-tuning process runs for multiple iterations on device, loss decreases as expected (baseline proof).
- **Task 6.2:** Optimize throughput: overlap transfers, adjust batch size. Achieve stable training. Possibly add checkpoint saving after epochs.
  - Milestone: **On-device fine-tuning demonstrated**, e.g., fine-tune on a small corpus for 1 epoch and show before/after perplexity.

## Phase 7: Performance Tuning & Profiling (Month 7)

- **Task 7.1:** Use Arm Streamline and AGI to profile one training iteration deeply. Identify bottlenecks (e.g., if memory bound, try larger tiles or fusing ops as discussed). Implement changes and measure improvement.
  - Deliverable: Profiling report with charts of GPU utilization, before/after optimization metrics.
- **Task 7.2:** Thermal test: run continuous training for N minutes, log throughput vs time. If severe drop, implement mitigation (as per Section 10).
  - Deliverable: A plot of throughput over time and temperature, showing manageable behavior.
- **Task 7.3:** Accuracy test: Fine-tune on a standard task (e.g., LAMBADA or WebQ) and evaluate. Compare to baseline fine-tune accuracy.
  - Deliverable: Accuracy numbers and confirmation of parity with baseline, validating training effectiveness.

## Phase 8: Documentation, Refactoring, and Deliverables (Month 8)

- **Task 8.1:** Clean up code, add comments, and possibly upstream-friendly changes. Ensure all new code paths are guarded by flags (so inference-only use is unaffected).
- **Task 8.2:** Write detailed documentation: how the fine-tuning can be invoked by users, hardware requirements, and guidelines (like “plug in your phone”). Also document the design of kernels and any API changes.
- **Task 8.3:** Final deliverables:
  - **Technical Report**, an IEEE-style paper (this document) describing the approach, with references to any external tools or libraries used (Vulkan, Arm, etc.)[9][35].
  - **Source Code**, diffs or repository link for the modified llama.cpp/ggml, including shader source.
  - **Example**, a working Android app or CLI binary for a reference device (e.g., a Samsung phone with Mali-G78), so reviewers can reproduce results. Possibly with a tiny demo dataset included.

### Milestones Summary:

- M1 (end of Phase 2): Ternary matmul kernel working (deliverable: PoC demonstration)
- M2 (end of Phase 4): Complete forward/backward GPU kernel suite implemented (deliverable: can compute gradients for a model layer on device)
- M3 (end of Phase 6): End-to-end fine-tuning loop functional (deliverable: device is actually training an LLM with quantized weights)
- M4 (end of Phase 7): Performance tuned and benchmarked (deliverable: report showing achievement of speed/efficiency targets)
- M5 (end of Phase 8): Project deliverables finalized (code, documentation ready for release)

Each phase's output will be reviewed to proceed to the next. This phased approach mitigates risk by verifying core pieces early (for example, if Phase 2 showed the kernel is too slow or inaccurate, we could rethink strategy before building everything on it).

## 1.12 RISK ASSESSMENT AND MITIGATION STRATEGIES

Developing this on-device fine-tuning system involves various risks:

**Risk 1: Performance Shortfall on Mali-G78**, The Mali-G78 GPU might not deliver the expected speedup due to lower frequency or memory bandwidth.

- If our kernels end up memory-bound, the speed gain over CPU might be smaller than anticipated. Mitigation: We have planned extensive optimization (tiling, vectorizing).

- If still limited, we can reduce model size or batch size to at least achieve a workable speed. We also consider enabling partial quantization: e.g.,
- if using 8-bit activations is too slow, try 4-bit (which would pack more into memory at cost of complexity).
- We also could offload some work to the device's DSP/NPU via NNAPI if absolutely necessary for performance (though that adds complexity). As a lower bound, even a 2-3x speedup over CPU might justify the approach, given training on CPU was basically infeasible for large models.

**Risk 2: Numerical Stability and Convergence**, Training with quantized weights can be unstable. The weight updates might oscillate or not converge to good minima, especially if using straight-through estimator for  $\{-1, 0, 1\}$ .

- Mitigation: We will tune the optimization hyperparameters carefully for quantized training (e.g., using smaller learning rate, gradient clipping).
  - We can also try **gradual quantization**, start fine-tuning with weights in higher precision (like 8-bit or float) then project to 1.58-bit slowly[23].
  - However, that may be beyond the scope of on-device (since starting at float doubles memory).
  - Alternatively, use a **shadow FP32 weight** updated with Adam, and only periodically sync to ternary. This could smooth the training. If quality issues persist, as mentioned, LoRA fine-tuning could be a fallback: keep base weights fixed and only learn small FP16 delta weights, easier to converge, then later merge them into ternary base.
  - The risk of quality issues is somewhat lowered by prior research success[24], but we remain cautious and incorporate monitoring of validation loss to early-stop if needed to avoid divergence.

**Risk 3: Device Variability and Compatibility**, Not all Mali-G78 devices have the same driver quality. If our Vulkan shaders hit a driver bug (e.g., certain bit operations not handled), it could crash or give wrong results.

- Mitigation: We will test on multiple devices (say a Huawei vs Samsung implementation of G78). We keep shaders simple and within well-tested features (no exotic extensions beyond INT8, which is common). Also, we can include a fallback to run on CPU if a particular device is problematic, ensuring correctness if not performance.

**Risk 4: Resource Contention**, on a phone, the GPU might also be needed for rendering the UI or other tasks. Running heavy compute might jank the UI.

- Mitigation: Run in a mode without much UI (simple progress text) or use Vulkan compute on a separate queue if the GPU supports it concurrently with graphics. Mali GPUs often have separate "fragment" and "compute" queues[51], which helps if UI and compute share GPU. Also, advising you not to use the phone for other tasks during training is reasonable.

**Risk 5: Memory Overflow or Fragmentation**, If allocations fail on device (esp. big contiguous buffers for model), the training can't even start.

- Mitigation: Pre-split large buffers if needed (e.g., allocate per-layer chunks instead of one monolithic block). Use Vulkan's memory allocator to handle fragmentation. We also ensure to release unused memory promptly (like free gradients of previous layers if doing layerwise training). If memory still insufficient, the ultimate fallback is to use a smaller model or quantize further (e.g., activations to 4-bit to cut memory).

**Risk 6: Timeline and Integration Risk**, Combining all these pieces (custom kernels, llama.cpp integration, Android deployment) is complex. Unknown issues might arise (like synchronization bugs causing incorrect training).

- Mitigation: We have phased milestones to catch issues early (e.g., verifying each backward kernel independently). We will also maintain close alignment with llama.cpp's design (reusing their structures) to ease integration. Possibly, engaging with the open-source community (ggml developers) early by sharing our plan could garner feedback or help, reducing integration friction.

**Risk 7: Ethical/Policy Risks**, On-device fine-tuning could be misused to remove safety filters or create harmful model variants. While not technical, this is worth noting. Llama.cpp is offline, so usage is user-controlled. Our proposal is technical, but one mitigation is to emphasize intended use (personalization, accessibility) and perhaps not fine-tune on harmful data. In any case, this risk is outside the scope of engineering.

We also prepare fallback options: If Mali-G78 proves underpowered, we can pivot to making the solution a demonstration for “next-gen” mobile GPUs (like the Immortalis or Apple M-series in iPads). The techniques remain similar. If full fine-tuning is too slow, we adapt to support only small-scale fine-tunings (like one layer or an adapter), which still achieves personalization goals with less compute.

In conclusion, while there are many risks, our mitigation strategies (algorithmic adjustments, careful resource management, testing) give confidence. The highest priority is ensuring we don't damage devices (thermal) and that training yields meaningful results (accuracy). By addressing performance and convergence risks with equal importance, we aim to deliver a robust solution.

## 1.13 UNDERLYING ASSUMPTIONS AND CRITICAL DEPENDENCIES

Throughout this proposal, we make several key assumptions:

- **Model Availability:** We assume a suitable **1.58-bit model** (BitNet architecture) is available to fine-tune. Specifically, having weights that are already ternary or a procedure to quantize an existing model. This is critical, if no good pre-trained model exists, training from scratch on device is impossible. We mitigate this by using released BitNet weights[48] or quantizing LLaMA-2, etc., as starting point.
- **Mali-G78 Vulkan Capability:** We assume the **Mali-G78 supports Vulkan 1.1+ with INT8 and fp16 features**. This is generally true (Valhall architecture supports subgroup operations and extended precision), and that the Android drivers have no show-stopping

bugs in compute. We also assume the device has a recent driver; if not, we might depend on an OS update (dependency on phone vendor updates).

- **Hardware Resources:** We assume a target device with **at least 8 GB RAM** and that we can use ~4-6 GB for our process. If a device has only 4GB, a 7B model may not fit, so our plan kind of assumes a higher-end phone (flagship from 2021 or so). This is reasonable as Mali-G78 was in high-end devices.
- **User Participation:** Fine-tuning on device implies the user is willing to allocate time, battery, and possibly keep device plugged in. We assume this is a conscious action (i.e., not something that happens without user knowing). Thus, slight inconveniences like plugging in or waiting 1 hour for tuning are acceptable under the “technically proficient reader” scenario.
- **ggml Extensibility:** We assume the llama.cpp ggml library can be extended with relatively moderate effort. If ggml had been written solely for inference, some structures might not directly support gradients. We depend on being able to either extend it or bypass it for training control flow. This is an implementation dependency, if ggml turned out to be too inflexible, we might implement a custom loop outside it.
- **Accuracy of Low-Precision Training:** We assume that fine-tuning in 1.58-bit will work to achieve near FP16-quality results (supported by literature[24]). If this assumption fails (the model accuracy stays poor), the entire proposition of BitNet fine-tuning is undermined. Our fallback assumption is that perhaps more fine-tuning steps or a different fine-tuning method (like knowledge distillation or gradually lowering precision) could eventually get there.
- **Task Suitability:** We assume tasks chosen for fine-tuning are feasible for a smaller model (2B-7B params). If one tried something requiring a 70B model, on-device is unfeasible. So we limit scope to tasks and model sizes that make sense (this is implied in using Mali-G78 anyway).
- **Development Time and Expertise:** It's assumed we have the needed expertise in Vulkan shader programming and low-level optimization. Also, that 8-9 months of development (per the plan) is sufficient. There is a dependency on possibly external knowledge, e.g., understanding Mali hardware counters (from Arm docs)[47], reading BitNet papers[1], etc. We rely on published sources for these (we have many referenced).
- **Dependencies on Tools:** We depend on the proper functioning of the Vulkan SDK, compilers, and profiling tools. If any of those are unavailable or not working on our platform (e.g., GPU inspector needing root, etc.), we have alternative (like using Streamline instead of AGI if needed, or building our own timers in code as rough measure).

In summary, our approach hinges on the assumption that **mobile GPUs with Vulkan can indeed execute these custom low-bit kernels efficiently**. If any critical dependency fails, e.g., Vulkan backend proves unstable, the project could pivot to OpenCL or other means, but that would slow us down. We have tried to ground these assumptions in evidence (like known support of INT8 on Mali, known existence of BitNet model).

Finally, we assume no unforeseen “unknown unknowns” like a Mali driver silently producing wrong results for certain bit ops. We will validate every step to catch such issues early. The plan is designed to be robust as long as the assumptions hold true.

## 1.14 PROOF OF CONCEPT (PoC): 1-BIT MATRIX MULTIPLICATION KERNEL IN VULKAN FOR A GPT LAYER

As a proof-of-concept, we can implement a simplified **1-bit (ternary) matrix multiply** for a single GPT layer on Vulkan. The scenario is a forward pass of a single linear layer in the transformer, using our custom kernel.

**Setup:** We choose a small GPT-style linear layer: let input dimension  $N_{in} = 256$  and output dimension  $N_{out} = 256$  for demonstrative purposes. We pack a random weight matrix  $W$  of shape  $(256 * 256)$  into 2-bit format (where each weight is  $\{-1, 0, 1\}$ ). The input  $X$  is a batch of, say,  $B = 4$  sequences with length 1 (so  $X$  is  $4 * 256$ ).

**Shader pseudocode (GLSL-like):**

```
#version 450
#extension GL_EXT_shader_subgroup : require

layout(local_size_x=16, local_size_y=16) in; // 256 threads per workgroup

layout(std430, binding=0) readonly buffer BufA { U8A_pack[]; }; // packed W, size = N_out * (N_in/4)
layout(std430, binding=1) readonly buffer BufB { uINT16_t B_fp16[]; }; // input X (FP16 values)
layout(std430, binding=2) writeonly buffer BufC { INT32_t C_acc[]; }; // output accumulator (INT32)

layout(push_constant) uniform PushConsts {
    uint N_in;
    uint N_out;
    uint batch;
} pc;

shared U8Asub[16][64]; // tile of packed W (16 rows * 64 bytes covers N_in=256)
shared uINT16_t Bsub[64][16]; // tile of X (64 elements of length, 16 batch elements)

void main(){
    uint out_row = gl_GlobalInvocationID.x; // 0..255
    uint out_col = gl_GlobalInvocationID.y; // 0..3 (since batch=4, maybe pad to 16 global)
    if(out_row >= pc.N_out || out_col >= pc.batch) return;
    // Each workgroup covers 16 output rows and 16 columns (here columns = batch elements)
    uint wg_row = gl_WorkGroupID.x * 16;
    uint wg_col = gl_WorkGroupID.y * 16;
    // Initialize accumulator
    int acc = 0;
    // Loop over K in chunks of 64 (i.e., 64*4 = 256 original length)
    for(uint k0=0; k0 < pc.N_in; k0 += 64){
```

```

// Load a 16x64 tile of W_packed into shared
uint packOffset = wg_row * (pc.N_in/4) + k0/4;
// each thread in y loads a different row piece
for(uint i=threadIdx.y; i < 16; i += local_size_y){
    // each iteration loads 16 bytes (so each thread loads maybe 1 byte for brevity)
    Asub[i][threadIdx.x] = A_pack[packOffset + i*(pc.N_in/4) + threadIdx.x];
}

// Load a 64x16 tile of X into shared (X is not packed, stored as FP16)
uint inOffset = k0 + wg_col;
for(uint j=threadIdx.x; j < 16; j += local_size_x){
    Bsub[j][threadIdx.y] = B_fp16[(inOffset + j) * pc.batch + (wg_col + threadIdx.y)];
}

barrier(); // ensure tile loaded
// Compute on the tile
for(uint k=0; k < 64; ++k){
    // Decode 16 weights from Asub for this output row:
    U8pack_val = Asub[threadIdx.x][k];
    // e.g., process two bits at a time:
    int w0 = (pack_val & 0x3) - 1;
    int w1 = ((pack_val >> 2) & 0x3) - 1;
    int w2 = ((pack_val >> 4) & 0x3) - 1;
    int w3 = ((pack_val >> 6) & 0x3) - 1;
    // multiply with 4 corresponding B values:
    // since B_fp16 is half, we can use half directly if supported
    half x0 = unpackHalf2x16(Bsub[k][threadIdx.y]).x;
    // (for simplicity we assume we can load half as float in code)
    // Actually we might do bitcast of 16-bit to 32 and treat as half etc.
    acc += w0 * float(x0);
    // conceptually, but need to cast half->float for multiply,
    // could accumulate in float then cast to int?
    // ... similarly for w1,w2,w3 with Bsub values ...
}
barrier();
}

// Write out acc to global (and perhaps convert to FP16)
C_acc[out_row * pc.batch + out_col] = acc;
}

```

*Note: The above pseudocode is illustrative; actual code would leverage vector operations and handle half precision properly. Also, each thread ideally computes one output (out\_row,out\_col). We used a 16x16 group; each thread identified by (threadIdx.x, threadIdx.y) would correspond to one output element in the tile.*

## 1.15 INTEGRATION PLAN FOR THE VULKAN KERNEL INTO LLAMA.CPP

Integrating our custom Vulkan kernels into llama.cpp involves modifications at several layers of the framework:

**15.1 Extending Model File Format and Loading:** We add support in llama.cpp's model loading for our quantized weights. The .gguf format will get a new block type for Q2\_ternary data. When reading, if such a block is encountered and the GPU backend is enabled, we allocate a Vulkan buffer and directly load the packed bytes into it (bypassing any CPU unpacking). Each ggml\_tensor representing a weight will have a flag indicating it's on GPU (and perhaps store a pointer/handle to the Vulkan buffer).

**15.2 ggml Operation Definition:** In ggml's ops list, we introduce something like GGML\_OP\_MATMUL\_Q2 for matrix multiplication where the second operand is quantized 2-bit. We implement the forward function to call our Vulkan routine. Specifically, in ggml\_graph\_compute(), when it sees this op and if GGML\_USE\_VULKAN is true, it will:

- Prepare descriptors: bind the weight buffer (from tensor) and the input and output buffers (which might be CPU or GPU). If input is from a previous GPU layer, it's already in a Vulkan buffer; if it's original embedding from CPU, we may need to copy it to a GPU buffer first.
- Record a dispatch of the Vulkan matmul shader with the appropriate workgroup sizes covering ( $N_{out} \times B$ ) outputs.
- Insert synchronization if needed (though ideally, we keep commands in process and only sync at the very end of inference or at certain points in training).

For backward, we might not integrate deeply into ggml's autograd (since it's minimal or non-existent). Instead, we manage backward calls manually in our training loop code. But we can still encapsulate them: define GGML\_OP\_MATMUL\_Q2\_BACKWARD or similar. However, given time, we might implement backward outside of ggml for simplicity, as long as we have access to the needed buffers.

**15.3 CPU-GPU Memory Management:** llama.cpp had already some CPU-GPU hybrid support[39], which suggests it can copy tensors to GPU memory. We will leverage that: likely there is an API like ggml\_cuda\_transform\_tensor() for the CUDA backend; similarly, we'll have ggml\_vk\_transform\_tensor(). We ensure any tensor that's needed on GPU is copied once and kept there. During training, intermediate activations for forward will be produced on GPU and kept until used for backward, then discarded.

- We also must handle that some operations remain on CPU: e.g., final loss calculation might be easier on CPU if it's scalar. Therefore, transferring the model output from GPU to CPU is needed at iteration end. This is small (just a few values) and won't hurt performance.

**15.4 Command Buffer and Synchronization:** We integrate with llama.cpp's mechanism for command submission. Possibly, llama.cpp uses one Vulkan command buffer and records all inference operations then does vkQueueSubmit. In training, we might use multiple command buffers per iteration (or a single long one). We will likely create a Vulkan fence to know when GPU work is done, so that we can then read results (for loss or for next steps if needed on CPU). The framework will need to wait on this fence at appropriate times (for instance, at the end of backward pass before updating weights on CPU, or at end of iteration).

We also manage multiple GPU queues if beneficial: Mali might only have one general queue, but if it had compute+transfer queues separate, we could copy data concurrently. We'll see if needed.

**15.5 Multi-Device and Fallback:** If a user runs llama.cpp on a device without Vulkan (or chooses not to), our additions should not impede CPU operation. We'll implement fallback CPU computations for any new op (even if slow) to maintain correctness. But these can be extremely slow (e.g., computing with quantized weights on CPU bit by bit). That's acceptable as long as user is warned or as a debug path.

**15.6 Testing Integration:** We will integrate one piece at a time: first just forward inference integration. Test that running llama.cpp -m bitnet.gguf uses Vulkan and produces correct text output. Then integrate backward: we might craft a custom test harness because llama.cpp doesn't have a "train" CLI. Possibly we add a new CLI argument to run one gradient descent step on a dummy input-output pair and print new loss. That would call our backward and update logic internally. This is mostly for development testing.

**15.7 API for Fine-Tuning:** We likely provide a simple API or script interface. For example, a function llama\_finetune(model, dataset, epochs, ...) that runs internally. This could be a C API that we expose, or just an addition to the CLI tool. Perhaps the easiest is adding to the CLI: "--finetune data.txt --steps 100". It would then read in data, tokenize, and loop for 100 steps calling our train routine. This keeps integration self-contained and user-friendly.

**15.8 Merging with llama.cpp updates:** Given llama.cpp is active, we'll keep our code modular to avoid conflict. Possibly our Vulkan training code could live in a separate source (e.g., ggml-vulkan-train.c) to minimize diff. We will document any changes to core structures.

**15.9 After Integration, Sample Run:** As a final integration test, we'll take a small model (say 1.3B) on a Mali-G78 phone, run "llama.cpp --finetune small\_dataset.txt --epochs 1". We expect to see in the console something like "Step 100/100, loss = X, ppl = Y" and then we can use the model to generate text, verifying it learned from the dataset. We also monitor that behind the scenes, Vulkan calls were made (we can enable VK\_LAYER logging or see the GPU usage spiking).

By carefully integrating at each point (model load, forward op, backward routines, weight updates), we ensure the system works end-to-end within the llama.cpp ecosystem. This leverages all existing capabilities (tokenization, sampling for generation, etc.) that llama.cpp already has, while adding the new training capability.

In conclusion, with the integration complete, **llama.cpp becomes a full training-capable framework for ultra-quantized models on mobile GPUs**. A user could take their phone, load a BitNet model, and fine-tune it on their own data with a single command, something that was previously impossible for LLMs. Our thorough integration plan above ensures that the Vulkan kernels we developed are effectively utilized to make this vision a reality.

## 1.16 REFERENCES

1. Ma, Shuming, et al. "The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits." arXiv preprint arXiv:2402.17764 (2024)[1][52].
2. El Khoury, Joe. "BitNet 1.58B Introduction." Medium (2023)[3][7].
3. "1.58-bit large language model." Wikipedia (2025)[5][15].
4. Mekkouri, Mohamed, et al. "Fine-tuning LLMs to 1.58bit: extreme quantization made easy." HuggingFace Blog (Sep 2024)[6][10].
5. "llama.cpp README." GitHub: ggml-org/llama.cpp (2025)[9][27].
6. "Arm Streamline for ML Workloads." Arm Developer Blog (Nov 2021)[47][41].
7. "Reverse-engineering the Mali G78." Collabora Blog (2021)[36].
8. Vulkan 1.2.182 Specification, subgroup operations and shaderINT8 (2020).  
(Additional references to tools and minor details are embedded in text as citations.)

## 1.17 SOURCES

[1] [52] [2402.17764] The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits

<https://arxiv.org/abs/2402.17764>

[2] [5] [12] [15] [16] [22] [23] [25] [48] 1.58-bit large language model - Wikipedia

[https://en.wikipedia.org/wiki/1.58-bit\\_large\\_language\\_model](https://en.wikipedia.org/wiki/1.58-bit_large_language_model)

[3] [4] [7] [11] [13] [14] [17] [18] BitNet 1.58 Bits. Introduction | by Joe El Khoury - GenAI Engineer | Medium

<https://medium.com/@jelkhoury880/bitnet-1-58b-0c2ad4752e4f>

[6] [8] [10] [19] [20] [21] [24] [26] [31] [32] [33] [34] [35] Fine-tuning LLMs to 1.58bit: extreme quantization made easy

[https://huggingface.co/blog/1\\_58\\_llm\\_extreme\\_quantization](https://huggingface.co/blog/1_58_llm_extreme_quantization)

[9] [27] [39] GitHub - ggml-org/llama.cpp: LLM inference in C/C++

<https://github.com/ggml-org/llama.cpp>

[28] Introducing the New OpenCL GPU Backend in llama.cpp for ...

<https://www.qualcomm.com/developer/blog/2024/11/introducing-new-opencl-gpu-backend-llama-cpp-for-qualcomm-adreno-gpu>

[29] [30] [50] VulkanLLM, Run Modern LLMs on Old GPUs via Vulkan (33x Faster on Dell iGPU, 4x on RX 580) : r/LocalLLaMA

[https://www.reddit.com/r/LocalLLaMA/comments/1moa5o0/vulkanllm\\_run\\_modern\\_llms\\_on\\_old\\_gpus\\_via\\_vulkan/](https://www.reddit.com/r/LocalLLaMA/comments/1moa5o0/vulkanllm_run_modern_llms_on_old_gpus_via_vulkan/)

[36] [37] [38] Reverse-engineering the Mali G78

<https://www.collabora.com/news-and-blog/news-and-events/reverse-engineering-the-mali-g78.html>

[40] [41] [43] [44] [45] [47] [51] Arm Community

<https://developer.arm.com/community/arm-community-blogs/b/ai-blog/posts/arm-streamline-for-ml-workloads>

[42] Mali-G78 Performance Counters Reference Guide - Arm Developer

<https://developer.arm.com/documentation/102626/0100/Shader-core-memory-access?lang=en>

[46] Android GPU Inspector

<https://gpuinspector.dev/>

[49] Optimize Llama.cpp with Arm I8MM instruction - Arm Developer

<https://developer.arm.com/community/arm-community-blogs/b/ai-blog/posts/optimize-llama-cpp-with-arm-i8mm-instruction>