```
1,2d0
< # convert.py
<
21,23c19,21
<     "FEAT8_a_proj_with_mqa": ("wFEAT8_a", None),
<     "FEAT8_a_layernorm": ("FEAT8_norm", None),
<     "FEAT8_b_proj": ("wFEAT8_b", 0),
---
>     "kv_a_proj_with_mqa": ("wkv_a", None),
>     "kv_a_layernorm": ("kv_norm", None),
>     "kv_b_proj": ("wkv_b", 0),
35c33
< def main(hf_ckpt_path, save_path, n_EXPTs, mp):
---
> def main(hf_ckpt_path, save_path, n_experts, mp):
42c40
<         n_EXPTs (int): Total number of EXPTs in the model.
---
>         n_experts (int): Total number of experts in the model.
49c47
<     n_local_EXPTs = n_EXPTs // mp
---
>     n_local_experts = n_experts // mp
61c59
<                 name = name.replace("FEAT2", "ffn")
---
>                 name = name.replace("mlp", "ffn")
70c68
<                     if "EXPTs" in name and "shared_EXPTs" not in name:
---
>                     if "experts" in name and "shared_experts" not in name:
72c70
<                         if idx < i * n_local_EXPTs or idx >= (i + 1) * n_local_EXPTs:
---
>                         if idx < i * n_local_experts or idx >= (i + 1) * n_local_experts:
94c92
<     parser.add_argument("--n-EXPTs", type=int, required=True)
---
>     parser.add_argument("--n-experts", type=int, required=True)
97,103c95,96
<     assert args.n_EXPTs % args.model_parallel == 0
<     main(args.hf_ckpt_path, args.save_path, args.n_EXPTs, args.model_parallel)
<
< # ============================================================================
<
< # fp8_cast_bf16.py
<
---
>     assert args.n_experts % args.model_parallel == 0
>     main(args.hf_ckpt_path, args.save_path, args.n_experts, args.model_parallel)
115c108
< def main(FEAT6_path, bf16_path):
---
> def main(fp8_path, bf16_path):
117c110
<     Converts FEAT6 weights to BF16 and saves the converted weights.
---
>     Converts FP8 weights to BF16 and saves the converted weights.
119c112
<     This function reads FEAT6 weights from the specified directory, converts them to BF16,
---
>     This function reads FP8 weights from the specified directory, converts them to BF16,
124c117
<     FEAT6_path (str): The path to the directory containing the FEAT6 weights and model index file.
---
>     fp8_path (str): The path to the directory containing the FP8 weights and model index file.
131c124
```

```
<     - The function assumes that the FEAT6 weights are stored in safetensor files.
---
>     - The function assumes that the FP8 weights are stored in safetensor files.
137c130
<     model_index_file = os.path.join(FEAT6_path, "model.safetensors.index.json")
---
>     model_index_file = os.path.join(fp8_path, "model.safetensors.index.json")
144c137
<     FEAT6_weight_names = []
---
>     fp8_weight_names = []
162c155
<             file_path = os.path.join(FEAT6_path, file_name)
---
>             file_path = os.path.join(fp8_path, file_name)
166c159
<     safetensor_files = list(glob(os.path.join(FEAT6_path, "*.safetensors")))
---
>     safetensor_files = list(glob(os.path.join(fp8_path, "*.safetensors")))
177c170
<             elif weight.element_size() == 1:  # FEAT6 weight
---
>             elif weight.element_size() == 1:  # FP8 weight
182c175
<                     FEAT6_weight_names.append(weight_name)
---
>                     fp8_weight_names.append(weight_name)
201c194
<     for weight_name in FEAT6_weight_names:
---
>     for weight_name in fp8_weight_names:
211c204
<     parser.add_argument("--input-FEAT6-hf-path", type=str, required=True)
---
>     parser.add_argument("--input-fp8-hf-path", type=str, required=True)
214,218c207
<     main(args.input_FEAT6_hf_path, args.output_bf16_hf_path)
<
< # ============================================================================
<
< # generate.py
---
>     main(args.input_fp8_hf_path, args.output_bf16_hf_path)
337c326
<     tokenizer.decode(generate(model, [tokenizer.encode("ds")], 2, -1, 1.)[0])
---
>     tokenizer.decode(generate(model, [tokenizer.encode("DeepSeek")], 2, -1, 1.)[0])
405,409d393
<
< # ============================================================
<
< # kernel.py
<
413,415c397,399
< import FEAT7
< import FEAT7.language as tl
< from FEAT7 import Config
---
> import triton
> import triton.language as tl
> from triton import Config
418c402
< @FEAT7.jit
---
> @triton.jit
424,426c408,410
<         x_ptr (FEAT7.Pointer): Pointer to the input tensor.
```

```
<         y_ptr (FEAT7.Pointer): Pointer to the output tensor where quantized values will be stored.
<         s_ptr (FEAT7.Pointer): Pointer to the output tensor where scaling factors will be stored.
---
>         x_ptr (triton.Pointer): Pointer to the input tensor.
>         y_ptr (triton.Pointer): Pointer to the output tensor where quantized values will be stored.
>         s_ptr (triton.Pointer): Pointer to the output tensor where scaling factors will be stored.
459c443
<     grid = lambda meta: (FEAT7.cdiv(x.numel(), meta['BLOCK_SIZE']), )
---
>     grid = lambda meta: (triton.cdiv(x.numel(), meta['BLOCK_SIZE']), )
464c448
< @FEAT7.jit
---
> @triton.jit
512c496
<     grid = lambda meta: (FEAT7.cdiv(M, meta['BLOCK_SIZE']), FEAT7.cdiv(N, meta['BLOCK_SIZE']))
---
>     grid = lambda meta: (triton.cdiv(M, meta['BLOCK_SIZE']), triton.cdiv(N, meta['BLOCK_SIZE']))
517c501
< FEAT6_gemm_configs = [
---
> fp8_gemm_configs = [
522,524c506,508
< @FEAT7.autotune(configs=FEAT6_gemm_configs, key=['N', 'K'])
< @FEAT7.jit
< def FEAT6_gemm_kernel(a_ptr, b_ptr, c_ptr,
---
> @triton.autotune(configs=fp8_gemm_configs, key=['N', 'K'])
> @triton.jit
> def fp8_gemm_kernel(a_ptr, b_ptr, c_ptr,
531c515
<     Performs a matrix multiplication operation on FEAT6 matrices with scaling factors.
---
>     Performs a matrix multiplication operation on FP8 matrices with scaling factors.
579c563
< def FEAT6_gemm(a: torch.Tensor, a_s: torch.Tensor, b: torch.Tensor, b_s: torch.Tensor):
---
> def fp8_gemm(a: torch.Tensor, a_s: torch.Tensor, b: torch.Tensor, b_s: torch.Tensor):
581c565
<     Perform a matrix multiplication using FEAT6 precision.
---
>     Perform a matrix multiplication using FP8 precision.
598,599c582,583
<     grid = lambda META: (FEAT7.cdiv(M, META['BLOCK_SIZE_M']), FEAT7.cdiv(N, META['BLOCK_SIZE_N']))
<     FEAT6_gemm_kernel[grid](a, b, c, a_s, b_s, M, N, K)
---
>     grid = lambda META: (triton.cdiv(M, META['BLOCK_SIZE_M']), triton.cdiv(N, META['BLOCK_SIZE_N']))
>     fp8_gemm_kernel[grid](a, b, c, a_s, b_s, M, N, K)
601,605d584
<
< # ====================================================
<
< # model.py
<
615c594
< from kernel import act_quant, weight_dequant, FEAT6_gemm
---
> from kernel import act_quant, weight_dequant, fp8_gemm
621c600
< gemm_impl: Literal["bf16", "FEAT6"] = "bf16"
---
> gemm_impl: Literal["bf16", "fp8"] = "bf16"
632c611
<         dtype (Literal["bf16", "FEAT6"]): Data type for computations.
---
>         dtype (Literal["bf16", "fp8"]): Data type for computations.
635,636c614,615
```

```
<         inter_dim (int): Intermediate dimension for FEAT2 layers.
<         FEAT1_inter_dim (int): Intermediate dimension for FEAT1 layers.
---
>         inter_dim (int): Intermediate dimension for MLP layers.
>         moe_inter_dim (int): Intermediate dimension for MoE layers.
640,645c619,624
<         n_routed_EXPTs (int): Number of routed EXPTs for FEAT1 layers.
<         n_shared_EXPTs (int): Number of shared EXPTs for FEAT1 layers.
<         n_activated_EXPTs (int): Number of activated EXPTs in FEAT1 layers.
<         n_EXPT_groups (int): Number of EXPT groups.
<         n_limited_groups (int): Number of limited groups for FEAT1 routing.
<         score_func (Literal["softmax", "sigmoid"]): Scoring function for FEAT1 routing.
---
>         n_routed_experts (int): Number of routed experts for MoE layers.
>         n_shared_experts (int): Number of shared experts for MoE layers.
>         n_activated_experts (int): Number of activated experts in MoE layers.
>         n_expert_groups (int): Number of expert groups.
>         n_limited_groups (int): Number of limited groups for MoE routing.
>         score_func (Literal["softmax", "sigmoid"]): Scoring function for MoE routing.
648c627
<         FEAT8_lora_rank (int): LoRA rank for key-value projections.
---
>         kv_lora_rank (int): LoRA rank for key-value projections.
650c629
<         qk_FEAT5_head_dim (int): Dimension for query-key projections with rotary embeddings.
---
>         qk_rope_head_dim (int): Dimension for query-key projections with rotary embeddings.
653,654c632,633
<         FEAT5_theta (float): Base for rotary positional encoding.
<         FEAT5_factor (float): Scaling factor for extended sequence lengths.
---
>         rope_theta (float): Base for rotary positional encoding.
>         rope_factor (float): Scaling factor for extended sequence lengths.
661c640
<     dtype: Literal["bf16", "FEAT6"] = "bf16"
---
>     dtype: Literal["bf16", "fp8"] = "bf16"
665c644
<     FEAT1_inter_dim: int = 1408
---
>     moe_inter_dim: int = 1408
669,673c648,652
<     # FEAT1
<     n_routed_EXPTs: int = 64
<     n_shared_EXPTs: int = 2
<     n_activated_EXPTs: int = 6
<     n_EXPT_groups: int = 1
---
>     # moe
>     n_routed_experts: int = 64
>     n_shared_experts: int = 2
>     n_activated_experts: int = 6
>     n_expert_groups: int = 1
677c656
<     # FEAT3
---
>     # mla
679c658
<     FEAT8_lora_rank: int = 512
---
>     kv_lora_rank: int = 512
681c660
<     qk_FEAT5_head_dim: int = 64
---
>     qk_rope_head_dim: int = 64
685,686c664,665
<     FEAT5_theta: float = 10000.0
```

```
<     FEAT5_factor: float = 40
---
>     rope_theta: float = 10000.0
>     rope_factor: float = 40
754c733
<         - For other cases, the function applies quantization to `x` and uses `FEAT6_gemm` for computation.
---
>         - For other cases, the function applies quantization to `x` and uses `fp8_gemm` for computation.
763c742
<         y = FEAT6_gemm(x, scale, weight, weight.scale)
---
>         y = fp8_gemm(x, scale, weight, weight.scale)
909c888
<     dim = args.qk_FEAT5_head_dim
---
>     dim = args.qk_rope_head_dim
913,914c892,893
<     base = args.FEAT5_theta
<     factor = args.FEAT5_factor
---
>     base = args.rope_theta
>     factor = args.rope_factor
998c977
< class FEAT3(nn.Module):
---
> class MLA(nn.Module):
1000c979
<     Multi-Headed Attention Layer (FEAT3).
---
>     Multi-Headed Attention Layer (MLA).
1007c986
<         FEAT8_lora_rank (int): Rank for low-rank key/value projection.
---
>         kv_lora_rank (int): Rank for low-rank key/value projection.
1009c988
<         qk_FEAT5_head_dim (int): Dimensionality of rotary-positional query/key projections.
---
>         qk_rope_head_dim (int): Dimensionality of rotary-positional query/key projections.
1020c999
<         self.FEAT8_lora_rank = args.FEAT8_lora_rank
---
>         self.kv_lora_rank = args.kv_lora_rank
1022,1023c1001,1002
<         self.qk_FEAT5_head_dim = args.qk_FEAT5_head_dim
<         self.qk_head_dim = args.qk_nope_head_dim + args.qk_FEAT5_head_dim
---
>         self.qk_rope_head_dim = args.qk_rope_head_dim
>         self.qk_head_dim = args.qk_nope_head_dim + args.qk_rope_head_dim
1032,1034c1011,1013
<         self.wFEAT8_a = Linear(self.dim, self.FEAT8_lora_rank + self.qk_FEAT5_head_dim)
<         self.FEAT8_norm = RMSNorm(self.FEAT8_lora_rank)
<         self.wFEAT8_b = ColumnParallelLinear(self.FEAT8_lora_rank, self.n_heads * (self.qk_nope_head_dim +
self.v_head_dim))
---
>         self.wkv_a = Linear(self.dim, self.kv_lora_rank + self.qk_rope_head_dim)
>         self.kv_norm = RMSNorm(self.kv_lora_rank)
>         self.wkv_b = ColumnParallelLinear(self.kv_lora_rank, self.n_heads * (self.qk_nope_head_dim +
self.v_head_dim))
1038c1017
<             mscale = 0.1 * args.mscale * math.log(args.FEAT5_factor) + 1.0
---
>             mscale = 0.1 * args.mscale * math.log(args.rope_factor) + 1.0
1045,1046c1024,1025
<             self.register_buffer("FEAT8_cache", torch.zeros(args.max_batch_size, args.max_seq_len,
self.FEAT8_lora_rank), persistent=False)
<             self.register_buffer("pe_cache", torch.zeros(args.max_batch_size, args.max_seq_len,
self.qk_FEAT5_head_dim), persistent=False)
```

```
---
>             self.register_buffer("kv_cache", torch.zeros(args.max_batch_size, args.max_seq_len,
self.kv_lora_rank), persistent=False)
>             self.register_buffer("pe_cache", torch.zeros(args.max_batch_size, args.max_seq_len,
self.qk_rope_head_dim), persistent=False)
1050c1029
<         Forward pass for the Multi-Headed Attention Layer (FEAT3).
---
>         Forward pass for the Multi-Headed Attention Layer (MLA).
1068c1047
<         q_nope, q_pe = torch.split(q, [self.qk_nope_head_dim, self.qk_FEAT5_head_dim], dim=-1)
---
>         q_nope, q_pe = torch.split(q, [self.qk_nope_head_dim, self.qk_rope_head_dim], dim=-1)
1070,1071c1049,1050
<         FEAT8 = self.wFEAT8_a(x)
<         FEAT8, k_pe = torch.split(FEAT8, [self.FEAT8_lora_rank, self.qk_FEAT5_head_dim], dim=-1)
---
>         kv = self.wkv_a(x)
>         kv, k_pe = torch.split(kv, [self.kv_lora_rank, self.qk_rope_head_dim], dim=-1)
1075,1077c1054,1056
<             FEAT8 = self.wFEAT8_b(self.FEAT8_norm(FEAT8))
<             FEAT8 = FEAT8.view(bsz, seqlen, self.n_local_heads, self.qk_nope_head_dim + self.v_head_dim)
<             k_nope, v = torch.split(FEAT8, [self.qk_nope_head_dim, self.v_head_dim], dim=-1)
---
>             kv = self.wkv_b(self.kv_norm(kv))
>             kv = kv.view(bsz, seqlen, self.n_local_heads, self.qk_nope_head_dim + self.v_head_dim)
>             k_nope, v = torch.split(kv, [self.qk_nope_head_dim, self.v_head_dim], dim=-1)
1083,1086c1062,1065
<             wFEAT8_b = self.wFEAT8_b.weight if self.wFEAT8_b.scale is None else
weight_dequant(self.wFEAT8_b.weight, self.wFEAT8_b.scale, block_size)
<             wFEAT8_b = wFEAT8_b.view(self.n_local_heads, -1, self.FEAT8_lora_rank)
<             q_nope = torch.einsum("bshd,hdc->bshc", q_nope, wFEAT8_b[:, :self.qk_nope_head_dim])
<             self.FEAT8_cache[:bsz, start_pos:end_pos] = self.FEAT8_norm(FEAT8)
---
>             wkv_b = self.wkv_b.weight if self.wkv_b.scale is None else weight_dequant(self.wkv_b.weight,
self.wkv_b.scale, block_size)
>             wkv_b = wkv_b.view(self.n_local_heads, -1, self.kv_lora_rank)
>             q_nope = torch.einsum("bshd,hdc->bshc", q_nope, wkv_b[:, :self.qk_nope_head_dim])
>             self.kv_cache[:bsz, start_pos:end_pos] = self.kv_norm(kv)
1088c1067
<             scores = (torch.einsum("bshc,btc->bsht", q_nope, self.FEAT8_cache[:bsz, :end_pos]) +
---
>             scores = (torch.einsum("bshc,btc->bsht", q_nope, self.kv_cache[:bsz, :end_pos]) +
1096,1097c1075,1076
<             x = torch.einsum("bsht,btc->bshc", scores, self.FEAT8_cache[:bsz, :end_pos])
<             x = torch.einsum("bshc,hdc->bshd", x, wFEAT8_b[:, -self.v_head_dim:])
---
>             x = torch.einsum("bsht,btc->bshc", scores, self.kv_cache[:bsz, :end_pos])
>             x = torch.einsum("bshc,hdc->bshd", x, wkv_b[:, -self.v_head_dim:])
1102c1081
< class FEAT2(nn.Module):
---
> class MLP(nn.Module):
1104c1083
<     Multi-Layer Perceptron (FEAT2) used as a feed-forward layer.
---
>     Multi-Layer Perceptron (MLP) used as a feed-forward layer.
1113c1092
<         Initializes the FEAT2 layer.
---
>         Initializes the MLP layer.
1126c1105
<         Forward pass for the FEAT2 layer.
---
>         Forward pass for the MLP layer.
1132c1111
<             torch.Tensor: Output tensor after FEAT2 computation.
```

```
---
>             torch.Tensor: Output tensor after MLP computation.
1139c1118
<     Gating mechanism for routing inputs in a mixture-of-EXPTs (FEAT1) model.
---
>     Gating mechanism for routing inputs in a mixture-of-experts (MoE) model.
1143c1122
<         topk (int): Number of top EXPTs activated for each input.
---
>         topk (int): Number of top experts activated for each input.
1160,1161c1139,1140
<         self.topk = args.n_activated_EXPTs
<         self.n_groups = args.n_EXPT_groups
---
>         self.topk = args.n_activated_experts
>         self.n_groups = args.n_expert_groups
1165,1166c1144,1145
<         self.weight = nn.Parameter(torch.empty(args.n_routed_EXPTs, args.dim))
<         self.bias = nn.Parameter(torch.empty(args.n_routed_EXPTs)) if self.dim == 7168 else None
---
>         self.weight = nn.Parameter(torch.empty(args.n_routed_experts, args.dim))
>         self.bias = nn.Parameter(torch.empty(args.n_routed_experts)) if self.dim == 7168 else None
1176c1155
<             Tuple[torch.Tensor, torch.Tensor]: Routing weights and selected EXPT indices.
---
>             Tuple[torch.Tensor, torch.Tensor]: Routing weights and selected expert indices.
1203c1182
< class EXPT(nn.Module):
---
> class Expert(nn.Module):
1205c1184
<     EXPT layer for Mixture-of-EXPTs (FEAT1) models.
---
>     Expert layer for Mixture-of-Experts (MoE) models.
1214c1193
<         Initializes the EXPT layer.
---
>         Initializes the Expert layer.
1227c1206
<         Forward pass for the EXPT layer.
---
>         Forward pass for the Expert layer.
1233c1212
<             torch.Tensor: Output tensor after EXPT computation.
---
>             torch.Tensor: Output tensor after expert computation.
1238c1217
< class FEAT1(nn.Module):
---
> class MoE(nn.Module):
1240c1219
<     Mixture-of-EXPTs (FEAT1) module.
---
>     Mixture-of-Experts (MoE) module.
1244,1249c1223,1228
<         n_routed_EXPTs (int): Total number of EXPTs in the model.
<         n_local_EXPTs (int): Number of EXPTs handled locally in distributed systems.
<         n_activated_EXPTs (int): Number of EXPTs activated for each input.
<         gate (nn.Module): Gating mechanism to route inputs to EXPTs.
<         EXPTs (nn.ModuleList): List of EXPT modules.
<         shared_EXPTs (nn.Module): Shared EXPTs applied to all inputs.
---
>         n_routed_experts (int): Total number of experts in the model.
>         n_local_experts (int): Number of experts handled locally in distributed systems.
>         n_activated_experts (int): Number of experts activated for each input.
>         gate (nn.Module): Gating mechanism to route inputs to experts.
>         experts (nn.ModuleList): List of expert modules.
```

```
>         shared_experts (nn.Module): Shared experts applied to all inputs.
1253c1232
<         Initializes the FEAT1 module.
---
>         Initializes the MoE module.
1256c1235
<             args (ModelArgs): Model arguments containing FEAT1 parameters.
---
>             args (ModelArgs): Model arguments containing MoE parameters.
1260,1265c1239,1244
<         assert args.n_routed_EXPTs % world_size == 0
<         self.n_routed_EXPTs = args.n_routed_EXPTs
<         self.n_local_EXPTs = args.n_routed_EXPTs // world_size
<         self.n_activated_EXPTs = args.n_activated_EXPTs
<         self.EXPTs_start_idx = rank * self.n_local_EXPTs
<         self.EXPTs_end_idx = self.EXPTs_start_idx + self.n_local_EXPTs
---
>         assert args.n_routed_experts % world_size == 0
>         self.n_routed_experts = args.n_routed_experts
>         self.n_local_experts = args.n_routed_experts // world_size
>         self.n_activated_experts = args.n_activated_experts
>         self.experts_start_idx = rank * self.n_local_experts
>         self.experts_end_idx = self.experts_start_idx + self.n_local_experts
1267,1269c1246,1248
<         self.EXPTs = nn.ModuleList([EXPT(args.dim, args.FEAT1_inter_dim) if self.EXPTs_start_idx <= i <
self.EXPTs_end_idx else None
<                                     for i in range(self.n_routed_EXPTs)])
<         self.shared_EXPTs = FEAT2(args.dim, args.n_shared_EXPTs * args.FEAT1_inter_dim)
---
>         self.experts = nn.ModuleList([Expert(args.dim, args.moe_inter_dim) if self.experts_start_idx <= i <
self.experts_end_idx else None
>                                     for i in range(self.n_routed_experts)])
>         self.shared_experts = MLP(args.dim, args.n_shared_experts * args.moe_inter_dim)
1273c1252
<         Forward pass for the FEAT1 module.
---
>         Forward pass for the MoE module.
1279c1258
<             torch.Tensor: Output tensor after EXPT routing and computation.
---
>             torch.Tensor: Output tensor after expert routing and computation.
1285,1286c1264,1265
<         counts = torch.bincount(indices.flatten(), minlength=self.n_routed_EXPTs).tolist()
<         for i in range(self.EXPTs_start_idx, self.EXPTs_end_idx):
---
>         counts = torch.bincount(indices.flatten(), minlength=self.n_routed_experts).tolist()
>         for i in range(self.experts_start_idx, self.experts_end_idx):
1289c1268
<             EXPT = self.EXPTs[i]
---
>             expert = self.experts[i]
1291,1292c1270,1271
<             y[idx] += EXPT(x[idx]) * weights[idx, top, None]
<         z = self.shared_EXPTs(x)
---
>             y[idx] += expert(x[idx]) * weights[idx, top, None]
>         z = self.shared_experts(x)
1303,1304c1282,1283
<         attn (nn.Module): Attention layer (FEAT3).
<         ffn (nn.Module): Feed-forward network (FEAT2 or FEAT1).
---
>         attn (nn.Module): Attention layer (MLA).
>         ffn (nn.Module): Feed-forward network (MLP or MoE).
1317,1318c1296,1297
<         self.attn = FEAT3(args)
<         self.ffn = FEAT2(args.dim, args.inter_dim) if layer_id < args.n_dense_layers else FEAT1(args)
---
```

```
>         self.attn = MLA(args)
>         self.ffn = MLP(args.dim, args.inter_dim) if layer_id < args.n_dense_layers else MoE(args)
1362c1341
<         Linear.dtype = torch.float8_e4m3fn if args.dtype == "FEAT6" else torch.bfloat16
---
>         Linear.dtype = torch.float8_e4m3fn if args.dtype == "fp8" else torch.bfloat16
```