InstantNGP中的EncodingGrid核心代码

```
std::unique_ptr<Context> forward_impl(
    cudaStream_t stream,
    const GPUMatrixDynamic<float>& input,
    GPUMatrixDynamic<T>* output = nullptr,
    bool use_inference_params = false,
    bool prepare_input_gradients = false
) override {
    auto forward = std::make_unique<ForwardContext>();

    const uint32_t num_elements = input.n();
    if ((!output && !prepare_input_gradients) || padded_output_width() == 0 ||
num_elements == 0) {
        return forward;
    }

    SyncedMultiStream synced_streams{stream, m_n_to_pad > 0 ? 2u : 1u};

    // Take care of padding on the auxiliary stream
    if (output && m_n_to_pad > 0) {
        if (output->layout() == AoS) {
            parallel_for_gpu_aos(synced_streams.get(1), num_elements, m_n_to_pad,
[n_output_dims=m_n_output_dims, out=output->pitched_ptr()] __device__ (size_t
elem, size_t dim) {
                out(elem)[n_output_dims + dim] = 0;
            });
        } else {
            parallel_for_gpu(synced_streams.get(1), num_elements * m_n_to_pad,
[out=output->data() + num_elements * m_n_output_dims] __device__ (size_t i) {
                out[i] = 0;
            });
        }
    }

    // Idea: each block only takes care of _one_ hash level (but may iterate over
multiple input elements).
    // This way, only one level of the hashmap needs to fit into caches at a time
(and it reused for consecutive
    // elements) until it is time to process the next level.

    static constexpr uint32_t N_THREADS_HASHGRID = 512;
    const dim3 blocks_hashgrid = { div_round_up(num_elements,
N_THREADS_HASHGRID), m_n_levels, 1 };

    T* encoded_positions_soa = output ? output->data() : nullptr;
    GPUMemoryArena::Allocation workspace;
    if (output && output->layout() == AoS) {
        workspace = allocate_workspace(synced_streams.get(0), num_elements *
m_n_features * sizeof(T));
        encoded_positions_soa = (T*)workspace.data();
    }

    if (prepare_input_gradients) {
```

```
        forward->dy_dx = GPUMatrix<float, RM>{N_POS_DIMS * m_n_features,
input.n(), synced_streams.get(0)};
    }

    kernel_grid<T, N_POS_DIMS, N_FEATURES_PER_LEVEL, HASH_TYPE>
<<<blocks_hashgrid, N_THREADS_HASHGRID, 0, synced_streams.get(0)>>>(
        num_elements,
        m_n_features,
        m_offset_table,
        m_base_resolution,
        std::log2(m_per_level_scale),
        this->m_max_level,
        this->m_max_level_gpu,
        m_interpolation_type,
        m_grid_type,
        use_inference_params ? this->inference_params() : this->params(),
        forward->positions.data() ? forward->positions.view() : input.view(),
        encoded_positions_soa,
        forward->dy_dx.data()
    );

    if (output && output->layout() == AoS) {
        // Transpose result (was stored row major due to coalescing)
        const dim3 threads_transpose = { m_n_levels * N_FEATURES_PER_LEVEL, 8, 1
};
        const uint32_t blocks_transpose = div_round_up(num_elements,
threads_transpose.y);
        transpose_encoded_position<T><<<blocks_transpose, threads_transpose, 0,
synced_streams.get(0)>>>(
            num_elements,
            encoded_positions_soa,
            output->pitched_ptr()
        );
    }

    return forward;
}
```

很明显，核心代码在`kernel_grid`中，前面和后面都只是一些初始化之类的操作。

再来看`kernel_grid`：

```
template <typename T, uint32_t N_POS_DIMS, uint32_t N_FEATURES_PER_LEVEL,
HashType HASH_TYPE>
__global__ void kernel_grid(
    const uint32_t num_elements,
    const uint32_t num_grid_features,
    const GridOffsetTable offset_table,
    const uint32_t base_resolution,
    const float log2_per_level_scale,
    float max_level,
    const float* __restrict__ max_level_gpu,
    const InterpolationType interpolation_type,
    const GridType grid_type,
    const T* __restrict__ grid,
```

```
    MatrixView<const float> positions_in,
    T* __restrict__ encoded_positions,
    float* __restrict__ dy_dx
) {
    const uint32_t i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= num_elements) return;

    const uint32_t level = blockIdx.y; // <- the level is the same for all
threads

    if (max_level_gpu) {
        max_level = (max_level_gpu[i] * num_grid_features) /
N_FEATURES_PER_LEVEL;
    } else {
        max_level = (max_level * num_grid_features) / N_FEATURES_PER_LEVEL;
    }

    if (level >= max_level + 1e-3f) {
        if (encoded_positions) {
            TCNN_PRAGMA_UNROLL
            for (uint32_t f = 0; f < N_FEATURES_PER_LEVEL; ++f) {
                encoded_positions[i + (level * N_FEATURES_PER_LEVEL + f) *
num_elements] = (T)0.0f;
            }
        }

        // Gradient is zero for zeroed-out dimensions.
        if (dy_dx) {
            TCNN_PRAGMA_UNROLL
            for (uint32_t f = 0; f < N_FEATURES_PER_LEVEL; ++f) {
                ((vec<N_POS_DIMS>*)dy_dx)[i + (level * N_FEATURES_PER_LEVEL + f)
* num_elements] = {0.0f};
            }
        }

        return;
    }

    grid += offset_table.data[level] * N_FEATURES_PER_LEVEL;
    const uint32_t hashmap_size = offset_table.data[level + 1] -
offset_table.data[level];

    const float scale = grid_scale(level, log2_per_level_scale, base_resolution);
    const uint32_t resolution = grid_resolution(scale);

    float pos[N_POS_DIMS];
    float pos_derivative[N_POS_DIMS];
    uvec<N_POS_DIMS> pos_grid;

    if (interpolation_type == InterpolationType::Nearest || interpolation_type ==
InterpolationType::Linear) {
        TCNN_PRAGMA_UNROLL
        for (uint32_t dim = 0; dim < N_POS_DIMS; ++dim) {
            pos_fract(positions_in(dim, i), &pos[dim], &pos_derivative[dim],
&pos_grid[dim], scale, identity_fun, identity_derivative);
        }
```

```
    } else {
        TCNN_PRAGMA_UNROLL
        for (uint32_t dim = 0; dim < N_POS_DIMS; ++dim) {
            pos_fract(positions_in(dim, i), &pos[dim], &pos_derivative[dim],
&pos_grid[dim], scale, smoothstep, smoothstep_derivative);
        }
    }

    auto grid_val = [&](const uvec<N_POS_DIMS>& local_pos) {
        const uint32_t index = grid_index<N_POS_DIMS, HASH_TYPE>(grid_type,
hashmap_size, resolution, local_pos) * N_FEATURES_PER_LEVEL;
        return *(tvec<T, N_FEATURES_PER_LEVEL, PARAMS_ALIGNED ? sizeof(T) *
N_FEATURES_PER_LEVEL : sizeof(T)>*)&grid[index];
    };

    if (interpolation_type == InterpolationType::Nearest) {
        auto result = grid_val(pos_grid);

        if (encoded_positions) {
            TCNN_PRAGMA_UNROLL
            for (uint32_t f = 0; f < N_FEATURES_PER_LEVEL; ++f) {
                encoded_positions[i + (level * N_FEATURES_PER_LEVEL + f) *
num_elements] = result[f];
            }
        }

        // Gradient is zero when there's no interpolation.
        if (dy_dx) {
            TCNN_PRAGMA_UNROLL
            for (uint32_t f = 0; f < N_FEATURES_PER_LEVEL; ++f) {
                ((vec<N_POS_DIMS>*)dy_dx)[i + (level * N_FEATURES_PER_LEVEL + f)
* num_elements] = {0.0f};
            }
        }

        return;
    }

    if (encoded_positions) {
        // N-linear interpolation
        tvec<T, N_FEATURES_PER_LEVEL, PARAMS_ALIGNED ? sizeof(T) *
N_FEATURES_PER_LEVEL : sizeof(T)> result = {};

        TCNN_PRAGMA_UNROLL
        for (uint32_t idx = 0; idx < (1 << N_POS_DIMS); ++idx) {
            float weight = 1;
            uvec<N_POS_DIMS> pos_grid_local;

            TCNN_PRAGMA_UNROLL
            for (uint32_t dim = 0; dim < N_POS_DIMS; ++dim) {
                if ((idx & (1<<dim)) == 0) {
                    weight *= 1 - pos[dim];
                    pos_grid_local[dim] = pos_grid[dim];
                } else {
                    weight *= pos[dim];
                    pos_grid_local[dim] = pos_grid[dim] + 1;
```

```
                }
            }

            result = fma((T)weight, grid_val(pos_grid_local), result);
        }

        TCNN_PRAGMA_UNROLL
        for (uint32_t f = 0; f < N_FEATURES_PER_LEVEL; ++f) {
            encoded_positions[i + (level * N_FEATURES_PER_LEVEL + f) *
num_elements] = result[f];
        }
    }

    // Gradient
    if (dy_dx) {
        vec<N_POS_DIMS> grads[N_FEATURES_PER_LEVEL] = {0.0f};

        TCNN_PRAGMA_UNROLL
        for (uint32_t grad_dim = 0; grad_dim < N_POS_DIMS; ++grad_dim) {
            TCNN_PRAGMA_UNROLL
            for (uint32_t idx = 0; idx < (1 << (N_POS_DIMS-1)); ++idx) {
                float weight = scale;
                uvec<N_POS_DIMS> pos_grid_local;

                TCNN_PRAGMA_UNROLL
                for (uint32_t non_grad_dim = 0; non_grad_dim < N_POS_DIMS-1;
++non_grad_dim) {
                    const uint32_t dim = non_grad_dim >= grad_dim ?
(non_grad_dim+1) : non_grad_dim;

                    if ((idx & (1<<non_grad_dim)) == 0) {
                        weight *= 1 - pos[dim];
                        pos_grid_local[dim] = pos_grid[dim];
                    } else {
                        weight *= pos[dim];
                        pos_grid_local[dim] = pos_grid[dim] + 1;
                    }
                }

                pos_grid_local[grad_dim] = pos_grid[grad_dim];
                auto val_left = grid_val(pos_grid_local);
                pos_grid_local[grad_dim] = pos_grid[grad_dim] + 1;
                auto val_right = grid_val(pos_grid_local);

                TCNN_PRAGMA_UNROLL
                for (uint32_t feature = 0; feature < N_FEATURES_PER_LEVEL;
++feature) {
                    grads[feature][grad_dim] += weight *
((float)val_right[feature] - (float)val_left[feature]) *
pos_derivative[grad_dim];
                }
            }
        }

        TCNN_PRAGMA_UNROLL
        for (uint32_t f = 0; f < N_FEATURES_PER_LEVEL; ++f) {
```

```
            ((vec<N_POS_DIMS>*)dy_dx)[i + (level * N_FEATURES_PER_LEVEL + f) *
num_elements] = grads[f];
        }
    }
}
```

其输入的`grid`就是模型参数，先经过`grid += offset_table.data[level] * N_FEATURES_PER_LEVEL;`按照目标level进行了一次offset；具体取数的过程在`grid_val`中，`grid_val`作为一个函数给后续过程调用用于从`grid`中取值