**SIMD Enhancement Using Unreal's VectorRegister:**

If the workload and platform support it, Unreal’s SIMD-friendly VectorRegister can further optimize the loop:

**SIMD Vector Processing Example:**

cpp

Copy code

#include "Math/UnrealMathVectorCommon.h"

void ProcessPixelsWithUnrealSIMD(uint8\* PixelPtr, int32 TotalPixels)

{

constexpr int32 BytesPerPixel = 4; // RGBA

constexpr int32 SIMDWidth = 4; // Processes 4 floats at a time (128-bit SIMD)

// Aligned SIMD processing

for (int32 i = 0; i < TotalPixels; i += SIMDWidth)

{

uint8\* CurrentPixelPtr = PixelPtr + i \* BytesPerPixel;

// Load and convert to float vectors

VectorRegister R = VectorLoadAligned(CurrentPixelPtr + 2);

VectorRegister G = VectorLoadAligned(CurrentPixelPtr + 1);

VectorRegister B = VectorLoadAligned(CurrentPixelPtr);

VectorRegister A = VectorLoadAligned(CurrentPixelPtr + 3);

// Convert to 0-1 range

VectorRegister Factor = VectorSetFloat1(1.0f / 255.0f);

R = VectorMultiply(R, Factor);

G = VectorMultiply(G, Factor);

B = VectorMultiply(B, Factor);

// Apply color vision deficiency transformation (pseudo-code, replace with real function)

VectorRegister CorrectedR, CorrectedG, CorrectedB;

ApplySIMDColourCorrection(R, G, B, CorrectedR, CorrectedG, CorrectedB);

// Convert back to 8-bit

VectorRegister MaxValue = VectorSetFloat1(255.0f);

R = VectorMultiply(CorrectedR, MaxValue);

G = VectorMultiply(CorrectedG, MaxValue);

B = VectorMultiply(CorrectedB, MaxValue);

// Store results

VectorStoreAligned(R, CurrentPixelPtr + 2);

VectorStoreAligned(G, CurrentPixelPtr + 1);

VectorStoreAligned(B, CurrentPixelPtr);

VectorStoreAligned(A, CurrentPixelPtr + 3);

}

}

**Why Use SIMD in Unreal Engine?**

1. **Cross-Platform Optimization**:
   * Unreal’s VectorRegister abstracts away platform-specific SIMD intrinsics.
   * Portable across x86 (SSE/AVX) and ARM (NEON).
2. **Integrated Debugging**:
   * SIMD operations in Unreal are easier to debug within its framework compared to raw intrinsics.
3. **Massive Performance Boost**:
   * SIMD + ParallelFor allows Unreal projects to handle high-resolution textures or real-time effects like heatmaps efficiently.

Below is a more Unreal Engine–specific approach using **VectorRegister** (the engine’s abstraction over SSE intrinsics) to process pixels in a partially vectorized manner. This example shows how you can use some of Unreal’s math utilities (UnrealMathSSE.h) to speed up per-pixel operations like scaling from 8-bit to floats (0.0–1.0 range), performing some math, then converting back.

**Note**:

1. VectorRegister is a 128-bit type that can store four 32-bit floats and perform SIMD operations on them.
2. This example processes 4 pixels at a time *for scaling/clamping* in SSE, then applies your color transformation function. The color transform function (ApplyColourCorrectionToCVD) itself may still be scalar unless rewritten for SSE.
3. This example assumes **RGBA 8 bits per channel** (4 bytes per pixel). If you have a different format, adjust accordingly.

**Example: Partially Vectorized Pixel Processing in Unreal**

cpp

Copy code

#include "Math/UnrealMathSSE.h" // For VectorRegister, VectorLoad/Store, VectorMultiply, etc.

#include "HAL/PlatformMath.h" // For FPlatformMath if needed

#include "Misc/ScopeLock.h" // If needed for threading

#include "Async/ParallelFor.h" // For ParallelFor if you want multi-threaded processing

// Your function that corrects color deficiency. Scalar version assumed here:

static FORCEINLINE FVector3f ApplyColourCorrectionToCVD(

const FVector3f& InColor,

EColorVisionDeficiency DeficiencyType,

float DeficiencyLevel,

bool bCorrectDeficiency)

{

// Implement your color vision deficiency transform.

// This is just a placeholder.

return InColor;

}

void ProcessPixelsWithSIMDAndParallel(

uint8\* InOutPixels,

int32 TextureWidth,

int32 TextureHeight,

EColorVisionDeficiency DeficiencyType,

float DeficiencyLevel,

bool bCorrectDeficiency)

{

// Number of pixels total

const int32 TotalPixels = TextureWidth \* TextureHeight;

constexpr int32 BytesPerPixel = 4; // RGBA8

// We'll process in chunks of 4 pixels at a time,

// because a VectorRegister can hold 4 floats (128 bits).

constexpr int32 PixelsPerBatch = 4;

// Figure out how many full batches (of 4 pixels) we can process

const int32 AlignedPixelCount = (TotalPixels / PixelsPerBatch) \* PixelsPerBatch;

// Optionally do this in parallel if the data set is large:

ParallelFor(AlignedPixelCount / PixelsPerBatch, [&](int32 BatchIndex)

{

// BatchIndex goes from 0 to (AlignedPixelCount / 4) - 1

const int32 PixelStart = BatchIndex \* PixelsPerBatch;

uint8\* BatchPtr = InOutPixels + (PixelStart \* BytesPerPixel);

//

// 1) Load 4 pixels (RGBA8 each) into SSE-friendly floats

// We'll make 4 separate VectorRegisters: one each for R, G, B, A

//

// We have 16 bytes for 4 RGBA pixels each, but they're packed in the order:

// pixel0(RGBA), pixel1(RGBA), pixel2(RGBA), pixel3(RGBA)

// This means R components are not stored contiguously, so we must gather them.

// Let's do a quick gather approach (scalar gather -> SSE):

// Because SSE gather in pure intrinsics form is more complicated,

// we’ll read them into intermediate arrays, then load them into a VectorRegister.

alignas(16) float RData[4];

alignas(16) float GData[4];

alignas(16) float BData[4];

alignas(16) float AData[4];

for (int i = 0; i < 4; ++i)

{

const uint8\* Pixel = BatchPtr + i \* BytesPerPixel;

BData[i] = Pixel[0] \* (1.0f / 255.0f);

GData[i] = Pixel[1] \* (1.0f / 255.0f);

RData[i] = Pixel[2] \* (1.0f / 255.0f);

AData[i] = Pixel[3] \* (1.0f / 255.0f);

}

// Now load those arrays into VectorRegisters

VectorRegister RVec = VectorLoadAligned(RData);

VectorRegister GVec = VectorLoadAligned(GData);

VectorRegister BVec = VectorLoadAligned(BData);

VectorRegister AVec = VectorLoadAligned(AData);

//

// 2) Apply color transform. We'll do it per component in scalar form,

// but you can adapt to SSE if your math is straightforward.

//

// For demonstration, we break them out into 4 scalars, call

// ApplyColourCorrectionToCVD, then store back.

// If you want full SSE, you’d rewrite the color transform to operate

// on VectorRegisters.

//

// Extract from VectorRegister into a temporary array

VectorStoreAligned(RVec, RData);

VectorStoreAligned(GVec, GData);

VectorStoreAligned(BVec, BData);

VectorStoreAligned(AVec, AData);

FVector3f Corrected[4];

for (int i = 0; i < 4; ++i)

{

const FVector3f Source(RData[i], GData[i], BData[i]);

Corrected[i] = ApplyColourCorrectionToCVD(Source, DeficiencyType, DeficiencyLevel, bCorrectDeficiency);

}

//

// 3) Write back to the batch memory in 8-bit RGBA, clamped

//

for (int i = 0; i < 4; ++i)

{

uint8\* Pixel = BatchPtr + i \* BytesPerPixel;

// B = Corrected[i].Z

Pixel[0] = FMath::Clamp(FMath::RoundToInt(Corrected[i].Z \* 255.0f), 0, 255);

// G = Corrected[i].Y

Pixel[1] = FMath::Clamp(FMath::RoundToInt(Corrected[i].Y \* 255.0f), 0, 255);

// R = Corrected[i].X

Pixel[2] = FMath::Clamp(FMath::RoundToInt(Corrected[i].X \* 255.0f), 0, 255);

// A = we kept the old alpha in AData[i], or do a transform if needed

Pixel[3] = FMath::Clamp(FMath::RoundToInt(AData[i] \* 255.0f), 0, 255);

}

});

//

// 4) Handle leftover pixels that are not a multiple of 4

//

for (int32 i = AlignedPixelCount; i < TotalPixels; ++i)

{

uint8\* Pixel = InOutPixels + i \* BytesPerPixel;

// 0-1 range

float B = Pixel[0] / 255.0f;

float G = Pixel[1] / 255.0f;

float R = Pixel[2] / 255.0f;

float A = Pixel[3] / 255.0f;

FVector3f Source(R, G, B);

FVector3f DeficientRGB = ApplyColourCorrectionToCVD(Source, DeficiencyType, DeficiencyLevel, bCorrectDeficiency);

// Write back

Pixel[0] = FMath::Clamp(FMath::RoundToInt(DeficientRGB.Z \* 255.0f), 0, 255);

Pixel[1] = FMath::Clamp(FMath::RoundToInt(DeficientRGB.Y \* 255.0f), 0, 255);

Pixel[2] = FMath::Clamp(FMath::RoundToInt(DeficientRGB.X \* 255.0f), 0, 255);

Pixel[3] = FMath::Clamp(FMath::RoundToInt(A \* 255.0f), 0, 255);

}

}

**Explanation of Key Steps**

1. **VectorRegister**:  
   Unreal provides a 128-bit register type called VectorRegister. It can hold 4 floats at a time and perform SIMD operations (e.g., VectorAdd, VectorMultiply).
2. **Gathering RGBA**:
   * Since our pixel data (4 bytes per pixel) is laid out as [B, G, R, A] for each pixel, the components for 4 pixels are interleaved in memory.
   * A direct SSE gather is more complex, so here we read the values into small aligned arrays, then load them into VectorRegister.
3. **Color Transform**:
   * This snippet does the color transformation (ApplyColourCorrectionToCVD) in scalar form for demonstration. If your transform is mostly linear algebra, you could rewrite it using VectorRegister math for a larger speedup.
4. **Storing Back**:
   * After computing the new floats, we clamp and convert them back to 8-bit.
5. **ParallelFor**:
   * We wrap the processing in ParallelFor to distribute the work across multiple cores. Each chunk processes 4 pixels \* N.
   * If your texture is large, multi-threading plus SIMD can significantly improve performance.

**Tips and Further Improvements**

* **Full SSE/AVX Implementation**:  
  If the color transform is a straightforward matrix multiply, you could load RVec, GVec, BVec into a single VectorRegister4Float representation and apply a matrix transform in one go (e.g., 3×3 or 4×4 matrix multiplication).
* **Platform Abstraction**:  
  If you need this to run on non-x86 (e.g., ARM with NEON), consider using UE’s built-in cross-platform vector types (VectorRegister and related functions) rather than raw intrinsics.
* **Data Alignment**:
  + VectorRegister operations can require 16-byte alignment for best performance.
  + If your buffer isn’t guaranteed to be aligned, you can use VectorLoadUnaligned/VectorStoreUnaligned or handle misaligned chunks carefully.
* **LUT / Lookup Table**:  
  If ApplyColourCorrectionToCVD is mostly a simple function, you might accelerate it further using precomputed LUTs (for example, 256-entry LUT per channel).
* **Render Thread vs. CPU**:  
  Depending on your use case, consider using a compute shader (GPU) if you need real-time color transforms on large textures.