Una implementación del algoritmo Exposure Fusion bajo el paradigma SIMD

Kevin Allekotte, Thomas Fischer

19 de marzo de 2018

Resumen

Una cámara captura la intensidad de la luminosidad en cada punto de una imagen. El rango de estas intensidades varía según la *exposición* de la cámara (controlada por los niveles de apertura, tiempo de exposición y sensibilidad), pero el rango de luminosidad -*rango dinámico*- que el sensor o la película de la cámara sabe captar, es acotado. En consecuencia, se saturan las intensidades para valores fuera de ese rango, perdiendose detalle en estas zonas (partes quemadas -blancas- y oscuras -negras-).

Es posible capturar el rango dinámico completo de una imagen tomando múltiples fotos bajo distintas exposiciones y fusionando luego esta información en imágenes de alto rango dinámico (HDR). El siguiente problema que surge, es la visualización de esta gran cantidad de información, ya que los medios que existen para reproducir las imágenes (monitores, impresión), cuentan a su vez con un rango lumínico muy acotado. De aquí la motivación para comprimir la información de manera de conservar la mayor cantidad de detalle y calidad posibles en una imágen de bajo randgo dinámico (LDR).

Construír la imágen HDR tiene un costo computacional alto, y necesita de gran intervención y toma de decisiones por parte del usuario para elegir/combinar algoritmos de fusión y determinar sus parámetros. El objetivo de este trabajo es fusionar dicha información sin computar una imágen HDR, automatizando el proceso bajo un algoritmo de estimación de calidad de las partes de una imágen, reduciendo el costo computacional y la intervención por parte del usuario.

Para esto presentamos una implementación del algoritmo de Exposure Fusion [1] en assembler x64 usando SIMD Extensions.

1. Introducción

El rango dinámico del ojo humano es dificil de calcular, pero bajo condiciones parecidas a las de un sistema de fotografía, es mucho mayor que este último, por lo que captar adecuadamente en una imágen lo que ve un ojo humano es imposible bajo condiciones de mucho rango dinámico.

Las imágenes HDR (High Dynamic Range) son imágenes que tienen un rango dinámico mucho más grande, y por lo tanto resultan más representativas y mas detalladas que las imágenes comunes. HDRI (HDR Imaging) son métodos utilizados para crear imágenes HDR combinando la información de varias imágenes de la misma escena con bajo rango dinámico bajo distintas exposiciones.

Sin embargo, las imágenes HDR no son apropiadas para mostrar en monitores o imprimir, ya que el rango dinámico de estos medios también es acotado y son aptos para ver sólo imágenes con bajo rango dinámico. Una opción, entonces, es volver a **comprimir una imagen HDR a un rango dinámico bajo, mostrando la mayor cantidad de detalle en cada zona y manteniendo la estética y realismo de la misma**. Hay distintos algoritmos que logran esto con distintos resultados, se llaman algoritmos de *Tone Mapping*.

Exposure Fusion [1] es una técnica desarrollada para fusionar varias imágenes de bajo rango dinámico tomadas a distintas exposiciones con un resultado similar a los operadores de tone mapping de HDR. Esta técnica evita pasos como la creación de la imagen HDR, la calibración de la curva de respuesta de la cámara y la selección de operadores de tone mapping, pasos que son costosos computacionalmente e implican una calibración "a ojo" de los parámetros.

Exposure fusion analiza ciertas propiedades de las imágenes que son relevantes para la calidad de la foto, y por cada pixel de la imagen se queda con la "mejor" información que puede recuperar a partir de todas las imágenes originales. Estas propiedades son el contraste, la saturación y la calidad de exposición de un pixel. El resultado final es computado automáticamente. A diferencia de otros algoritmos de tone-mapping, no requiere la creación de la imagen HDR sino que directamente combina las distintas exposiciones. Gracias a esto es más robusto, más rapido, y permite por ejemplo mezclar tomas con y sin flash.

2. Algoritmo

Cada pixel de la imagen final se obtiene como la suma ponderada de los valores de ese pixel en las imágenes de entrada. El peso que se le asigna a cada muestra es la medida de la calidad de ese pixel en esa imagen:

$$R_{i,j} = \sum_{k=1}^{N} \hat{W}_{k_{i,j}} * I_{k_{i,j}}$$

Donde N es la cantidad de imágenes a fusionar, I_k son esas imágenes, $\hat{W}_{k_{i,j}}$ es el peso del pixels $I_{k_{i,j}}$ y $R_{i,j}$ es la imagen resultante.

Para calcular los pesos $\hat{W}_{k_{i,j}}$, primero estimamos la calidad de cada pixel en función de tres propiedades, *Contraste*, *Saturación* y *Exposición*:

$$W_{k_{i,j}} = (C_{k_{i,j}})^{w_C} \times (S_{k_{i,j}})^{w_S} \times (E_{k_{i,j}})^{w_E}$$

donde w_C, w_S y w_E son parámetros que regulan cuanta importancia le asignamos a cada métrica, y luego normalizamos los pesos de las matrices $W_{k_{i,j}}$ para que el resultado sume 1 en cada pixel, y el resultado tenga el mismo depth bit que las imágenes originales.

$$\hat{W}_{k_{i,j}} = W_{k_{i,j}} \times [\sum_{k'=1}^{N} W_{k'_{i,j}}]^{-1}$$

Queda entonces calcular las matrices de los estimadores de calidad.

2.1. Contraste

Aplicamos un filtro laplaciano a la versión en escala de grises de cada imagen y nos quedamos con el valor absoluto de la respuesta. Esto suele asignar pesos altos a elementos que pertenecen a bordes o texturas en la imagen.

$$C_k = laplaciano(I_k)$$

2.2. Saturación

Una medida importante de la calidad de la imágen es la saturación. Buscamos colores vívidos, que son los que tienen más diferencia entre los colores que lo componen, aquellos que se encuentran mas lejos de los grises. Se calcula como el desvío estándar para cada píxel de los componentes rojo, verde y azul, contra el valor esperado en la escala de grises:

$$S_{k_{i,j}} = \frac{\sqrt{(r-\mu)^2 + (g-\mu)^2 + (b-\mu)^2}}{3}$$

Donde r, g y b son los valores de los canales rojo, verde y azul del píxel $I_{k_{i,j}}$ respectivamente, y el valor esperado $\mu = \frac{b+g+r}{3}$.

2.3. Exposición

Un pixel bien expuesto es aquel que no tiene una intensidad cercana al mínimo o al máximo (sobresaturada) del rango dinámico en sus canales. Luego, definimos la calidad de la exposición como la distancia al valor óptimo 0,5 en cada canal, respecto de una curva gaussiana

$$E_{k_{i,j}} = \prod_{i=\{r,g,b\}} exp(-\frac{(i-0{,}5)^2}{2\sigma^2})$$

Donde r, g y b son los valores de los canales rojo, verde y azul del píxel $I_{k_{i,j}}$ respectivamente, y proponemos $\sigma = 0{,}2$ como en [1].

3. Implementación

La aplicación esta hecha para Linux de 64 bits (fue probado en distintas versiones de Ubuntu). El principal lenguaje de programación usado en la implementación fue C. Las operaciones de procesamiento de las imágenes fueron implementadas en ASM x64 para mejorar la performance, y se utilizó GTK para la interfaz gráfica. Para cargar y guardar imágenes se usa la librería OpenCV.

Se abstrayeron las operaciones sobre matrices en el módulo `Matrix`, los algoritmos del Exposure Fusion están en `Expofuse`, y la interfaz gráfica en `Gui`. Los 3 módulos están implementados en C, y `Matrix` y `Expofuse` hacen llamadas a código en assembler en muchas de las funciones.

3.1. `Matrix`

El tipo Matrix es una `struct` de este tipo:

```
typedef struct {
  int cols;
  int rows;
  double* data;
} Matrix;
```

y implementa las siguientes operaciones:

```
Matrix* NewMatrix(int rows, int cols)
void DeleteMatrix(Matrix* A)
Matrix* CopyMatrix(Matrix* A)
Matrix* Substract(Matrix* A, Matrix* B)
Matrix* AddMatrix(Matrix* A, Matrix* B)
Matrix* AddEqualsMatrix(Matrix* A, Matrix* B)
Matrix* Convolve(Matrix* A, Matrix* kernel, BOUNDARY)
Matrix* Downsample(Matrix* I)
Matrix* Upsample(Matrix* I, int odd_rows, int odd_cols)
void ContrastCurve(Matrix* A, double strength)
```

3.2. Expofuse

Para representar imágenes y procesarlas usamos una `struct` de este tipo:

```
typedef struct {
  Matrix* R;
  Matrix* G;
  Matrix* B;
} ColorImage;
```

con las funciones:

```
ColorImage* NewColorImage()
void DeleteColorImage(ColorImage* image)
ColorImage* CopyColorImage(ColorImage* image)
ColorImage* AddColorImage(ColorImage* A, ColorImage* B)
ColorImage* AddEqualsColorImage(ColorImage* A, ColorImage* B)
ColorImage* LoadColorImage(char* filename, int size)
void SaveColorImage(ColorImage* I, char* filename)
void TruncateColorImage(ColorImage* I)
void SContrast(ColorImage* A, double strength)
```

y para implenetar Exposure Fusion se tienen las funciones:

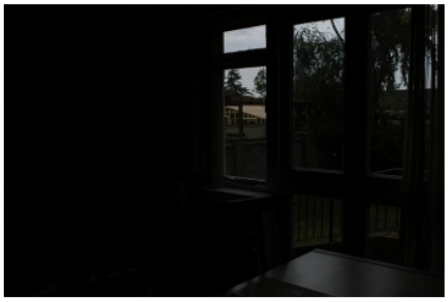
```
Matrix** ConstructWeights(ColorImage** color_images, int n_samples,
double contrast_weight, double saturation_weight, double exposeness_weight,
double sigma)
Matrix* Contrast(Matrix* I)
Matrix* Saturation(ColorImage* I)
Matrix* Exposeness(ColorImage* I, double sigma)
Matrix* Weight(Matrix* contrast, double contrast_weight, Matrix*
saturation, double saturation_weight, Matrix* exposeness, double exposeness_weight)
void NormalizeWeights(Matrix** weights, int n_samples)
void WeightColorImage(ColorImage* color_image, Matrix* weights)
```

```
ColorImage* Fusion(ColorImage** images, Matrix** weights, int n_samples)
Matrix** GaussianPyramid(Matrix* I, int levels)
Matrix** LaplacianPyramid(Matrix* I, int levels)
ColorImage** ColorLaplacianPyramid(ColorImage* I, int levels)
ColorImage* ReconstructFromPyramid(ColorImage** pyramid, int n_levels)
```

Para las partes en Assembler, implementamos la convención de llamadas a funciones de C en x64 y creamos las funciones que nos interesaban. Usamos las extensiones SSE para la mejor performance, y implementamos casos específicos para optimizar las operaciones más demandantes.

4. Resultados

Tal como los resultados de Mertens, Kautz y Van Reeth, obtuvimos imágenes con mucho nivel de detalle en condiciones de mucho contraste y alto rango de luminosidad, que son estéticas y muy realistas. A continuación mostramos algunos resultados con las imágenes de entrada y la composición final.






5. Análisis de performance

A continuación mostrmos una tabla comparativa con los tiempos de ejecución del algoritmo implementado en Assembler y en C, para distinas imágenes de entrada. Se midió sólo el tiempo que tarda la ejecución del procesado, ignorando tiempos de cargado a memoria y guardado de imágenes.

Cantidad y tamaño	**ASM**	**C**
4 × 752×500	2.11s	2.85s
3 × 870×653	2.46s	3.27s
3 × 1200×800	4.15s	5.54s
5 × 1500×750	7.68s	10.34s
3 × 3872×2592	48.96s	63.10s

Se observa que en general la versión en Assembler tarda un 75 % del tiempo de la versión de C.

6. Compilar y Correr

La implementación tiene como dependencias la librería OpenCV y Nasm, que pueden instalarse en ubuntu con:
`sudo apt-get install libcv-dev libhighgui-dev nasm`.

Para compilar el proyecto, hay que correr `make` desde `/src`.

```
cd /src
make
```

Si compila correctamente se puede correr el programa.

- Por consola, ejemplo:
  `./fuse -o room.jpg room/{A,B,C,D}.jpg`
  Corre el algoritmo con las imágenes `A.jpg`, `B.jpg`, `C.jpg`, `D.jpg` de `src/room/`.
  `./fuse -h`
  Para más información y detalle de los parámetros.

- Con la interfaz gráfica:
  `./gui`
  Inicia la interfaz gráfica donde se deben seleccionar las imágenes (seleccionar múltiples imágenes previamente alineadas y de las mismas dimensiones con Ctrl) y luego se pueden ajustar los parámetros y exportar la imagen final.

6.1. Opcional: Alinear imágenes antes de fusionar

Para que el algoritmo funcione con los resultados esperados, las imágenes de entrada tienen que estar perfectamente alineadas. En general se obtienen las fotos con un trípode, variando la exposición de cada toma, pero sin mover la cámara.

Pero también se pueden obtener las imágenes a mano alzada y luego alinearlas con un software. En este ejemplo mostramos cómo usar `align_image_stack` de Hugin.

Primero, instalar las herramientas de Hugin:
`sudo apt-get install hugin-tools`

Luego, se pueden alinear las imágenes de la siguiente forma:
`align_image_stack A.jpg B.jpg C.jpg D.jpg -a alineadas`
Esto genera las imágenes alineadasXXXX.tif, que pueden ser usadas en la interfaz gráfica o por consola.

A. Rango dinámico

El rango dinámico en una cámara de fotos, se define como la razón entre el nivel de blanco más alto que puede detectar un sensor y el nivel más bajo. Los sensores de cámaras digitales en blanco y negro, consisten de una grilla de fotositos, que se pueden ver como un recipiente con capacidad para cierta cantidad de fotones. Luego de la exposición, se mide la cantidad de fotones que se encuentran en cada fotosito, y eso nos da el valor lumínico de cada pixel. Si nuestro recipiente se llena, decimos que el pixel esta saturado y va a detectar como valores iguales a todos los que esten por encima del máximo, luego, estamos perdiendo detalle en las zonas claras. Si disminuímos el tiempo

de exposición de la imagen para contrarrestar la saturación, estamos dividiendo la cantidad de fotones que inciden en un recipiente por algun factor, con lo cuál los valores que se encuentran en las zonas oscuras son cada vez más parecidos, y son mas indistinguibles del ruido electrónico generado al leer el valor, por lo que perdemos detalle en las zonas oscuras. Claramente el rango dinámico depende del tamaño de estos recipientes, por ejemplo las cámaras compactas, comparadas por ejemplo con SLR's profesionales, al ser mas chicas deben reducir el tamaño del sensor y por lo tanto el de los fotositos, disminuyendo así el rango dinámico. El rango dinámico se mide con distintas unidades, de las cuáles la mas común es el f-stop, una escala logarítmica que describe el rango en potencias de 2, por ejemplo el rango (razón de contraste, o valor-máximo/valor-mínimo de luminosidad) de 1024 valores $1024 : 1 = 1024 = 2^{10} f - stops$. En imágenes a color, simplemente tenemos 4 fotositos por pixel, donde cada uno detecta análogamente solamente en un rango de frecuencias cerca de rojo, verde o azul.

Referencias

[1] Tom Mertens, Jan Kautz, and Frank Van Reeth. Exposure fusion. In Marc Alexa, Steven J. Gortler, and Tao Ju, editors, *Pacific Conference on Computer Graphics and Applications*, pages 382–390. IEEE Computer Society, 2007.