# A Optimizaciones NEON

El código que se ha optimizado es el siguiente:

```
const float factorPI = (float)(CV_PI/180.f);
static void computeOrbDescriptor(const KeyPoint& kpt,
                const Mat& img, const Point* pattern, uchar* desc)
{
    float angle = (float)kpt.angle*factorPI;
    float a = (float)cos(angle), b = (float)sin(angle);

    const uchar* center = &img.at<uchar>(cvRound(kpt.pt.y),
                                         cvRound(kpt.pt.x));
    const int step = (int)img.step;

    #define GET_VALUE(idx) \
     center[cvRound(pattern[idx].x*b + pattern[idx].y*a)*step + \
            cvRound(pattern[idx].x*a - pattern[idx].y*b)]


    for (int i = 0; i < 32; ++i, pattern += 16)
    {
        int t0, t1, val;
        t0 = GET_VALUE(0); t1 = GET_VALUE(1);
        val = t0 < t1;
        t0 = GET_VALUE(2); t1 = GET_VALUE(3);
        val |= (t0 < t1) << 1;
        t0 = GET_VALUE(4); t1 = GET_VALUE(5);
        val |= (t0 < t1) << 2;
        t0 = GET_VALUE(6); t1 = GET_VALUE(7);
        val |= (t0 < t1) << 3;
        t0 = GET_VALUE(8); t1 = GET_VALUE(9);
        val |= (t0 < t1) << 4;
        t0 = GET_VALUE(10); t1 = GET_VALUE(11);
        val |= (t0 < t1) << 5;
        t0 = GET_VALUE(12); t1 = GET_VALUE(13);
        val |= (t0 < t1) << 6;
        t0 = GET_VALUE(14); t1 = GET_VALUE(15);
        val |= (t0 < t1) << 7;

        desc[i] = (uchar)val;
    }

    #undef GET_VALUE
}
```

La optimización comienza en la entrada del bucle de la función. El cálculo de las variables que se encuentran antes del inicio del bucle no pueden ser optimizadas mediante instrucciones NEON ya que dichas variables son comunes para todos los descriptores y solo es necesario calcularlas una vez, por lo que esta parte de la función se mantendrá igual.

En primer lugar se realiza la carga de los datos. Contamos con instrucciones NEON que permiten la carga simultanea de datos contiguos en memoria. También nos ofrecen la posibilidad de realizar cargas con desentrelazado, que consisten en cargar los datos de forma que se distribuyan en varios registros del procesador de forma intercalada como puede verse en la figura **??**. En nuestro caso es perfecto para dejar los datos preparados para la siguiente operación que se realiza con ellos.

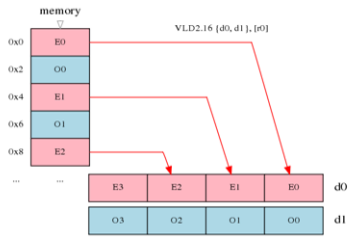


Figure A.1: Carga desentrelazada

Una vez que ya se han cargado los datos del patrón, las x's en un registro y las y'es en otro, ya pueden empezar a realizarse las primeras operaciones múltiples. En primer lugar se realizan las multiplicaciones de pattern[idx].x tanto por a como por b, recordemos que nuestros datos son floats de 32 bytes por lo que el máximo de multipliaciones que podemos realizar al mismo tiempo son 4. El siguiente término a calcular son las multiplicaciones de pattern[idx].y por a y b, NEON cuenta con una instrucción que permite realizar multiplicaciones y además permite realizar una suma o resta del resultado obtenido con otro registro, por lo que de esta forma podemos realizar al mismo tiempo las sumas y restas.

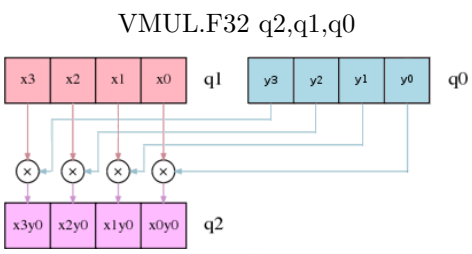


Figure A.2: Multiplicación vector por vector

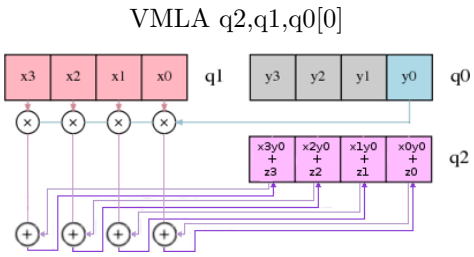


Figure A.3: Vector por escalar mas vector

El siguiente paso es redondear los resultados obtenidos convirtiéndolos en ints para sumar los resultados y así obtener el índice con el que acceder al array denominado "center", antes de realizar la suma es necesario multiplicar el primer término por step. Para realizar la multiplicación se usará una instrucción NEON que multiplica los datos de un registro por un escalar y suma el resultado con otro registro, de esta forma evitamos realizar la suma de los términos en otra instrucción separada.

A continuación es necesario acceder a memoria, en este caso no se puede realizar una carga simultanea de datos ya que los accesos a memoria no son contiguos por lo que será necesario realizar las diferentes cargas de forma independiente.

La última parte del algoritmo consiste en realizar comparaciones de datos dos a dos para obtener el descriptor. Las comparaciones se pueden realizar en grupos de 8 datos ya que son del tipo unsigned char de 8 bits. En primer lugar es necesario colocar los datos que van a ser comparados en un mismo registro ya que al realizar la carga no podemos especificarle el lugar del registro que queremos que ocupe, para ello se van a utilizar las instrucciones del tipo vext que cogen el primer dato de un registro y lo añaden al principio de otro realizando un desplazamiento del registro objetivo y eliminando el último de sus datos.

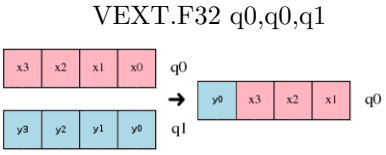


Figure A.4: Extracción

Una vez organizados los registros se puede realizar la comparación de los mismos con una sola instrucción. El último paso para obtener el valor binario(0 ó 1) que corresponde al descriptor consiste en realizar una operación AND sobre los registros ya que los resultados obtenidos tras la comparación son los 8 bits a 0 o 1 y nosotros necesitamos que solo el bit de menor valor este activado o desactivado. Tras esta iteración se ha obtenido el primer byte de los 32 bytes totales que forman el descriptor por lo que es necesario repetir el proceso descrito hasta tener todos los bytes necesarios, para evitar realizar el menor número de escrituras en memoria los datos obtenidos en las comparaciones se guardan en un registro hasta que se llena el registro y así almacenar el mayor número de datos posibles con una sola instrucción.

A continuación se muestra el código desarrollado:

```
const int *p = (int*)pattern;
uint8x8_t results;
for (int i = 0; i < 32; ++i, p += 32){
   //Carga el patron separado por x's e y's
   int32x4x4_t pattern0 = vld4q_s32(p);
   int32x4x4_t pattern1 = vld4q_s32(p + 16);
   int32x4x2_t patternX0 = vzipq_s32(pattern0.val[0],
                                     pattern0.val[2]);
   int32x4x2_t patternY0 = vzipq_s32(pattern0.val[1],
                                     pattern0.val[3]);
   int32x4x2_t patternX1 = vzipq_s32(pattern1.val[0],
                                     pattern1.val[2]);
   int32x4x2_t patternY1 = vzipq_s32(pattern1.val[1],
                                     pattern1.val[3]);
   //Conversion a float
   float32x4_t xPattern0 = vcvtq_f32_s32(patternX0.val[0]);
   float32x4_t yPattern0 = vcvtq_f32_s32(patternY0.val[0]);
   float32x4_t xPattern1 = vcvtq_f32_s32(patternX0.val[1]);
   float32x4_t yPattern1 = vcvtq_f32_s32(patternY0.val[1]);
   float32x4_t xPattern2 = vcvtq_f32_s32(patternX1.val[0]);
   float32x4_t yPattern2 = vcvtq_f32_s32(patternY1.val[0]);
   float32x4_t xPattern3 = vcvtq_f32_s32(patternX1.val[1]);
   float32x4_t yPattern3 = vcvtq_f32_s32(patternY1.val[1]);
   //Multiplicaciones 0
   float32x4_t firstTerm0 = vmulq_n_f32(xPattern0, b);
   float32x4_t secondTerm0 = vmulq_n_f32(xPattern0, a);
   float32x4_t firstTerm2 = vmulq_n_f32(xPattern2, b);
   float32x4_t secondTerm2 = vmulq_n_f32(xPattern2, a);
   firstTerm0 = vmlaq_n_f32(firstTerm0, yPattern0, a);
   secondTerm0 = vmlsq_n_f32(secondTerm0, yPattern0, b);
   firstTerm2 = vmlaq_n_f32(firstTerm2, yPattern2, a);
   secondTerm2 = vmlsq_n_f32(secondTerm2, yPattern2, b);
   //Multiplicaciones 1
   float32x4_t firstTerm1 = vmulq_n_f32(xPattern1, b);
   float32x4_t secondTerm1 = vmulq_n_f32(xPattern1, a);
   float32x4_t firstTerm3 = vmulq_n_f32(xPattern3, b);
   float32x4_t secondTerm3 = vmulq_n_f32(xPattern3, a);
   firstTerm1 = vmlaq_n_f32(firstTerm1, yPattern1, a);
   secondTerm1 = vmlsq_n_f32(secondTerm1, yPattern1, b);
   firstTerm3 = vmlaq_n_f32(firstTerm3, yPattern3, a);
   secondTerm3 = vmlsq_n_f32(secondTerm3, yPattern3, b);

   int32x4_t firstRound0 = vcvtq_s32_f32(firstTerm0);
   int32x4_t secondRound0 = vcvtq_s32_f32(secondTerm0);
   int32x4_t firstRound1 = vcvtq_s32_f32(firstTerm1);
   int32x4_t secondRound1 = vcvtq_s32_f32(secondTerm1);
   int32x4_t firstRound2 = vcvtq_s32_f32(firstTerm2);
```

```
int32x4_t secondRound2 = vcvtq_s32_f32(secondTerm2);
int32x4_t firstRound3 = vcvtq_s32_f32(firstTerm3);
int32x4_t secondRound3 = vcvtq_s32_f32(secondTerm3);
//Multiplicar por step y suma de terminos
firstRound0 = vmlaq_n_s32(secondRound0, firstRound0, step);
firstRound1 = vmlaq_n_s32(secondRound1, firstRound1, step);
firstRound2 = vmlaq_n_s32(secondRound2, firstRound2, step);
firstRound3 = vmlaq_n_s32(secondRound3, firstRound3, step);
//Acceso a la imagen, intercalado
uint8x8_t value0 = vld1_u8(center + firstRound0[0]);
uint8x8_t value1 = vld1_u8(center + firstRound0[1]);
uint8x8_t value2 = vld1_u8(center + firstRound0[2]);
uint8x8_t value3 = vld1_u8(center + firstRound0[3]);
uint8x8_t value4 = vld1_u8(center + firstRound1[0]);
uint8x8_t value5 = vld1_u8(center + firstRound1[1]);
uint8x8_t value6 = vld1_u8(center + firstRound1[2]);
uint8x8_t value7 = vld1_u8(center + firstRound1[3]);
uint8x8_t value8 = vld1_u8(center + firstRound2[0]);
uint8x8_t value9 = vld1_u8(center + firstRound2[1]);
uint8x8_t value10 = vld1_u8(center + firstRound2[2]);
uint8x8_t value11 = vld1_u8(center + firstRound2[3]);
uint8x8_t value12 = vld1_u8(center + firstRound3[0]);
uint8x8_t value13 = vld1_u8(center + firstRound3[1]);
uint8x8_t value14 = vld1_u8(center + firstRound3[2]);
uint8x8_t value15 = vld1_u8(center + firstRound3[3]);
//Reordena los datos
value0 = vrev64_u8(value0); //Toma el primer dato
value1 = vrev64_u8(value1); //Toma el primer dato
value0 = vext_u8(value0, value2, 1);
value1 = vext_u8(value1, value3, 1);
value0 = vext_u8(value0, value4, 1);
value1 = vext_u8(value1, value5, 1);
value0 = vext_u8(value0, value6, 1);
value1 = vext_u8(value1, value7, 1);
value0 = vext_u8(value0, value8, 1);
value1 = vext_u8(value1, value9, 1);
value0 = vext_u8(value0, value10, 1);
value1 = vext_u8(value1, value11, 1);
value0 = vext_u8(value0, value12, 1);
value1 = vext_u8(value1, value13, 1);
value0 = vext_u8(value0, value14, 1);
value1 = vext_u8(value1, value15, 1);
//Comparacion less than
value0 = vclt_u8(value0, value1);
//Transformacion resultados a 0,1
uint8_t one = 1;
uint8x8_t ones = vdup_n_u8(one);
value0 = vand_u8(value0, ones);
//Compone el descriptor
```

```
    value0[0] |= value0[1] << 1;
    value0[0] |= value0[2] << 2;
    value0[0] |= value0[3] << 3;
    value0[0] |= value0[4] << 4;
    value0[0] |= value0[5] << 5;
    value0[0] |= value0[6] << 6;
    value0[0] |= value0[7] << 7;
    //Almacena resultado
    results = vext_u8(results, value0, 1);
    if (i % 8 == 7) {
        //Escribe en memoria
        vst1_u8(desc + i - 7, results);
    }
}
```

# B Guía de compilación de ORB-SLAM2 para Android

La librería se ha compilado en un PC con sistema operativo Ubuntu 16.04 y como entorno de desarrollo se ha utilizado Android Studio 2.2.2.
Para la compilación de ORB-SLAM2 se necesita:

- Compilador de C++11 ó C++0x, se ha usado gcc 5.4.0.
- NDK de Android, versión utilizada 13.1.
  `https://developer.android.com/ndk/downloads/index.html`
- OpenCV for Android, versión mínima 2.4., en el proyecto se ha utilizado la versión 2.4.8 proporcionada por NVIDIA, que cuenta con optimizaciones para los procesadores Tegra.
  `http://opencv.org/downloads.html`
  `https://developer.nvidia.com/AndroidWorks-TADP-Archive`
- Eigen, versión mínima 3.1., se ha usado la 3.2.
  `http://eigen.tuxfamily.org/index.php?title=Main_Page`
- DBoW2 y g2o, incluidas en ORB-SLAM2.
  `https://github.com/raulmur/ORB_SLAM2`

**Instrucciones**

Para compilar utilizando el NDK junto con CMake es necesario configurar los siguientes flags a la hora de compilar:

- -DANDROID_NDK: indica la ruta donde se encuentra el NDK.
- -DANDROID_ABI: para especificar la arquitectura para la que se quiere compilar, en nuestro caso "armeabi-v7a with NEON".
- -DANDROID_API_LEVEL: indica la versión de la API utilizada, en el proyecto se ha utilizado la 19.
- -DCMAKE_BUILD_TYPE: para indicar el tipo de compilación, por ejemplo Release.
- -DCMAKE_TOOLCHAIN_FILE: para indicar la ruta de la toolchain a utilizar.
- -DANDROID_TOOLCHAIN_NAME: indica el nombre de la toolchain a utilizar, se ha usado arm-linux-androideabi-gcc-4.9.

Un ejemplo para realizar la compilación sería:

```
cmake . -DANDROID_NDK=/Android/Sdk/ndk-bundle
        -DANDROID_ABI=armeabi-v7a with NEON
        -DANDROID_NATIVE_API_LEVEL=19
        -DCMAKE_BUILD_TYPE=Release
        -DCMAKE_TOOLCHAIN_FILE=/Android/Sdk/cmake/3.6.3155560/
                               android.toolchain.cmake
        -DANDROID_TOOLCHAIN_NAME=arm-linux-androideabi-gcc-4.9
```

Tras haber descargado ORB-SLAM2 y sus dependencias, hay que seguir los siguientes pasos para lograr su compilación:

1. Modificar el fichero CMakeLists.txt de la librería DBoW2 para incluir los siguientes flags de compilación: -march=armv7-a, -mfpu=neon -mfloat-abi=softfp. También puede incluirse el flag -mtune=cortex-a15 para indicar el procesador exacto para el que se quiere compilar. Además también hay que incluir la nueva librería de OpenCV para Android. Una vez realizados estos ajustes ya puede iniciarse la compilación como se ha indicado al inicio de esta sección.

2. Para la librería g2o el proceso es similar que con DBoW2, hay que modificar también su fichero CMakeLists.txt para incluir los mismos flags de compilación y en este caso también hay que incluir las cabeceras de la librería Eigen descargada. Después iniciar la compilación.

3. Por último queda compilar ORB-SLAM2, de la misma forma que con las dos librerías anteriores en primer lugar se añadirán los flags de compilación a su fichero CMakeLists.txt. También es necesario incluir la librería de OpenCV utilizada y eliminar todas las referencias de Pangolin del código. Por último, antes de lanzar la compilación hay que asegurarse que la ruta donde se encuentran las librerías compiladas de DBoW2 y g2o es la misma que la que se encuentra definida en el fichero CMakeLists.txt y una vez todo esto esta listo se puede iniciar la compilación.

Una vez que ya se tiene la librería compilada es necesario añadirla al proyecto de la aplicación, en el caso de este proyecto se ha realizado con el soporte de CMake que tiene Android Studio. El proceso se puede ver en la siguiente guía:
`https://developer.android.com/studio/projects/add-native-code.html?hl=es-419`

# C Guía de uso

### Pantalla principal

En la pantalla principal el usuario puede acceder a la generación de un nuevo modelo del entorno o iniciar la visualización de un modelo que haya sigo generado previamente.

A la izquierda de la pantalla se incluye una lista con los modelos que se han generado mediante la aplicación. Para acceder a uno de estos modelos es necesario pulsar sobre el nombre del modelo que se quiere cargar, si la pulsación es prolongada se ofrece la posibilidad de borrar el modelo.

En la esquina inferior derecha de la pantalla se encuentra el botón que inicia la generación de un nuevo modelo, al pulsarlo se inicia ORB-SLAM y se comienza el proceso de generación.

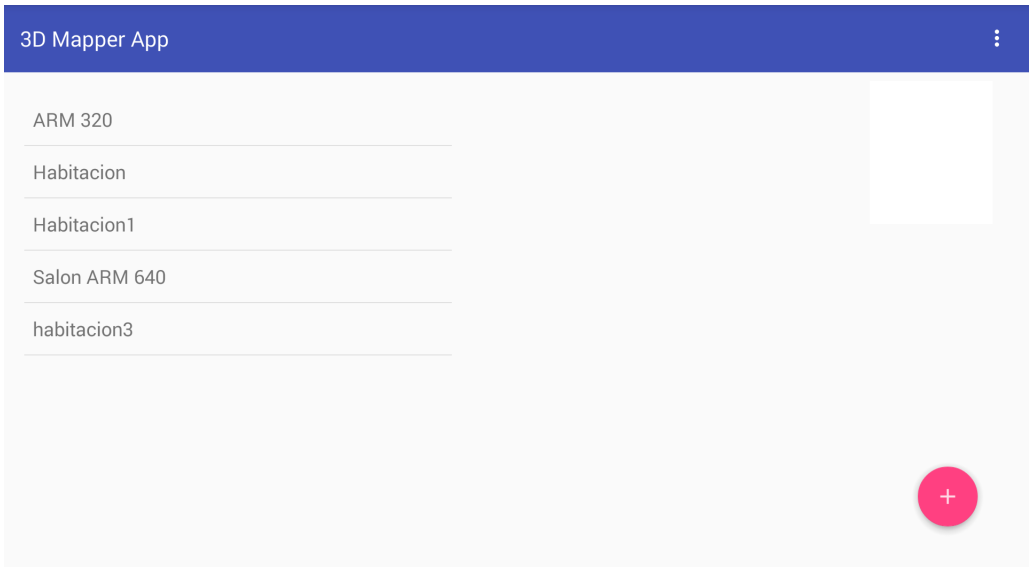


Figure C.1: Pantalla principal de la app

### Pantalla de generación de modelo

La mayor parte de la pantalla la ocupa el visualizador, donde se puede observar el mapa generado. A la hora de visualizar el mapa se permite activar algunas opciones mediante los botones situados en la parte superior de la pantalla y en la esquina inferior del visualizador.

- Follow: permite activar el seguimiento de la trayectoria seguido por la cámara, es decir, si se desea que la visualización del mapa se haga desde el punto donde se encuentra la cámara o desde uno fijo.
- Keyframes: activa el dibujado de los *keyframes*.
- Reconstruction: muestra la nube de puntos que se va generando.
- Pausa: detiene la generación del mapa y el modelo momentáneamente hasta que vuelva a pulsarse el botón.
- Stop: lanza un diálogo que permite terminar la generación del modelo y guardarlo introduciendo un nombre.

En las esquinas superiores derecha e izquierda, se pueden observar respectivamente las imágenes a color capturadas por la cámara en las se pintan los puntos de interés detectados por ORB-SLAM y imagen de profundidad captada por la cámara.

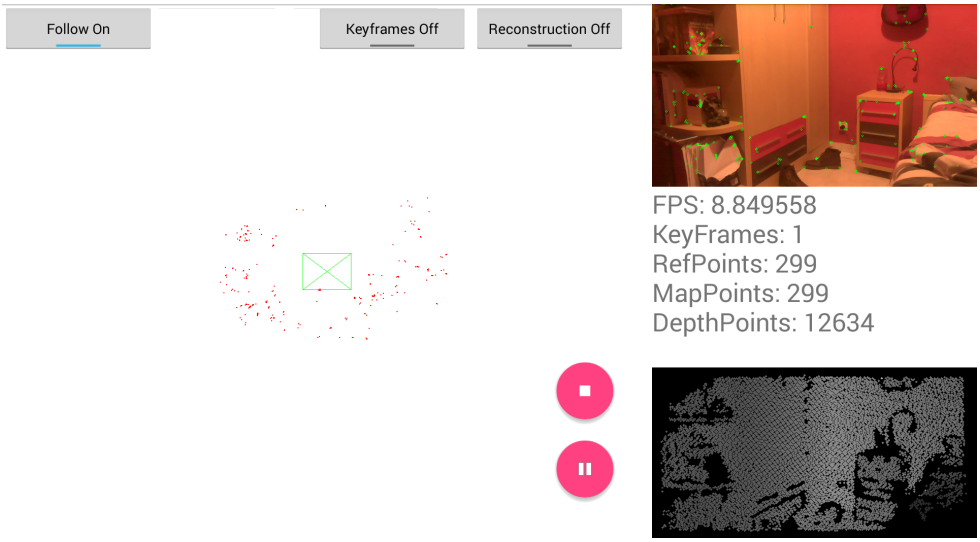


Figure C.2: Pantalla de generación de modelo

### Pantalla de visualización de modelos

La navegación principal se realiza mediante gestos en la pantalla táctil.

El primero de los gestos es el deslizamiento en una dirección, este gesto provoca la rotación de la cámara en la dirección en la que se realice el deslizamiento, si el gesto se realiza con dos dedos la cámara se trasladará en lugar de rotar.

El otro gesto admitido es el de acercar/alejar dos dedos, que producirá un cambio en el nivel de zoom en la visualización dependiendo de si el gesto es de acercamiento o alejamiento.

El resto de acciones se realizan mediante los botones en pantalla:

- Mapa: cambia la visualización a vista cenital.
- Rosa de los vientos: devuelve la cámara a la posición original.
- Flechas: permiten avanzar o retroceder la cámara en la dirección en la que esta observando.

Figure C.3: Pantalla de visualización de modelos de la app