https://doi.org/10.17979/spudc.9788497498081.0689

Evaluación de la arquitectura ARMv7-A para el desarrollo de HROV's eficientes

Diego Centelles, Rafael Mayo, Eduardo Moscoso, Raúl Marín, Pedro J. Sanz
Dep. de Ingeniería y Ciencia de los Computadores, Universidad Jaume I, Castellón de la Plana, España
{centelld, mayo, rubino, rmarin, sanzp}@uji.es

Resumen

Este trabajo pretende dar a conocer la viabilidad del uso de los mini-ordenadores basados en la arquitectura de procesador ARMv7-A o similar como ordenadores de a bordo de HROV's, destinados a realizar una tarea robótica específica. El bajo consumo energético de los ordenadores basados en estos procesadores y su bajo coste despierta el interés por utilizar estos dispositivos para implementar vehículos submarinos inalámbricos de alta autonomía. Los resultados demuestran que realizando una implementación a medida de los algoritmos de procesado de imagen típicos en robótica, aprovechando las ventajas del juego de instrucciones SIMD del procesador y el multi-núcleo, es posible conseguir una reducción considerable de los tiempos de ejecución.

Palabras clave: Intervención submarina, Autonomía, ARMv7-A, Arquitectura de procesador, Instrucciones SIMD, HROV.

1. Motivación

Una de las dificultades de los sitemas robóticos autónomos es conseguir una técnica para aumentar la autonomía de los robots, especialmente cuando se trata de aplicaciones con necesidades de comunicaciones inalámbricas, tal y como ocurre en el escenario del proyecto MERBOTS (ver Fig. 1). En MERBOTS se pretende conseguir un avance en el ámbito de las intervenciones robóticas submarinas utilizando para ello robots semiautónomos inalámbricos que cooperan entre sí, para conseguir un objetivo común y con un control supervisado de todo el proceso. El elevado precio de las baterías hace necesario que el conjunto de los componentes de los robots submarinos sea lo más eficiente posible, incluído el procesador. Además, en el caso concreto de los robots submarinos inalámbricos se requiere potencia extra para alimentar a los transceptores (normalmente acúticos) que, en general, consumen gran cantidad de energía.

Los procesadores ARM Cortex-A7, que implementan la arquitectura ARMv7-A, son comúnmente utilizados en dispositivos móviles para desempeñar las tareas menos exigentes (scroll sobre una página web, tecleado, reproducir audio, etc.). Su funcionamiento eficiente permite alargar la autonomía del dispositivo en periodos de baja intensidad de procesado. Sólo en los momentos en los que se requiere altas capacidades de procesado (renderizado, cálculo de las físicas de un juego, etc.) entran en funcionamiento los procesadores más potentes Cortex-A15 o el Cortex-A17.

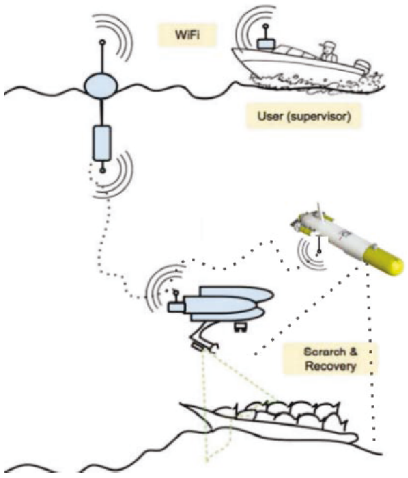

Figura 1: Escenario del proyecto MERBOTS donde se precisa un HROV y un AUV totalmente inalámbricos

En el trabajo presentado en [6] se propone un algoritmo de compresión de imágenes progresivo eficiente que permitiría tener *feedback* visual de alta tasa de refresco utilizando un mini-computador *Raspberry Pi 2 Model B*. Para el cálculo de las frecuencias, este algoritmo utiliza una transformada wavelet implementadada explotando al máximo el juego de instrucciones vectoriales de 128 bits de la arquitectura ARMv7-A (Instrucciones NEON). Los resultados presentados en ese trabajo demuestran que, con la versión optimizada del algoritmo en una *Raspberry Pi 2*, se permite la compresión de imágenes de 1024x768 a muy alta calidad a cerca de 30 *frames* por segundo utilizando sólo uno de los 4 núcleos del procesador. Sin embargo, con una implementación que no usase las instrucciones

vectoriales rondaría los 20 *frames* por segundo.

Librerías típicas de procesado de imágenes, como *OpenCV*, ya incluyen versiones optimizadas de algoritmos frecuentes como el detector de bordes *Canny* en las que se aprovecha al máximo la capacidad *multi-core* de estos procesadores, la caché y el juego de instrucciones SIMD.

2. Introducción

En primer lugar se realizará una breve explicación de la arquitectura del procesador utilizado para la experimentación, de su juego de instrucciones SIMD de 128 bits llamadas NEON, y la forma de utilizarlas desde código C a través de las *intrinsics*. En el siguiente apartado se mostrará un ejemplo de implementación del algoritmo del filtrado del ruido (paso previo a *Canny*) y del cálculo del gradiente utilizando este tipo de instrucciones y la paralelización mediante *OpenMP*. En el siguiente apartado se mostrán resultados en tiempos de ejecución. Seguidamente, antes de las conclusiones, se expondrá un ejemplo de tarea autónoma que podría realizar un HROV basado en esta arquitectura de procesador.

3. ARM Cortex-A7

Un procesador ARM Cortex-A7 implementa la arquitectura ARMv7-A y está compuesto de 4 núcleos con memoria caché L1 configurable de 8 a 64KB y una caché L2 de hasta 1MB. La arquitectura contiene un juego de instrucciones vectoriales llamadas NEON, capaces de realizar operaciones sobre vectores de hasta 128 bits, y de las que se habla en la siguiente sección.

4. Instrucciones NEON

La arquitectura ARMv7 extiende el concepto de las instrucciones SIMD (Single Instruction, Multiple Data) definiendo grupos de instrucciones que operan sobre vectores de hasta 128 bits, con elementos de 8 a 64 bytes. Como sugiere el nombre SIMD, este tipo de instrucciones son capaces de realizar al mismo tiempo la misma operación sobre todos los elementos del vector. Para realizar estas operaciones, la arquitectura contiene un banco de 32 registros de 64 bits. Cada registro de 64 bits está mapeado a una mitad de un registro de 128 bits, por lo tanto, podemos decir que el banco tiene un tamaño de 32 registros de 64 bits o de 16 registros de 128 bits. Los registros de 64 bits son llamados registros D, y los registros de 128 bits, registros Q. Esto es por el número de palabras (de 32 bits) que contienen:

- Registros de 64 bits: *Doubleword register*.
- Registros de 128 bits: *Quadword register*.

Existen instrucciones NEON capaces de realizar múltiples operaciones de distinto tipo:

- Acceso a memoria: lectura o escritura en memoria de múltiples datos.
- Conversión de tipos.
- Procesamiento de datos: suma, multiplicación, desplazamiento de bits, etc.

Es posible realizar determinadas operaciones de nuestro programa escrito en C utilizando instrucciones NEON específicas. La mejor manera de hacerlo es utilizando llamadas a funciones de librería de ARM. Cada una de estas funciones de librería suelen corresponderse con una instrucción NEON, en la que se especifica como argumentos los registros a utilizar como operandos. Estas funciones son llamadas *intrinsics* (ver ejemplos 1, 2, 3 y 4).

Ejemplo 1: Ejemplo de instrucción VLD1: Cargar un vector de 4 elementos de enteros con signo de 32 bits en un registro de 128 bits

```
int vector[4] = {...};
int32x4_t registro;
registro = vld1q_s32(vector);
```

Ejemplo 2: Ejemplo de instrucción VADD: Sumar dos vectores de 4 elementos de enteros con signo de 32 bits y guardar los 4 resultados en un registro Q (de 128 bits)

```
int32x4_t registro1, registro2;
...
int32x4_t resultados;
resultados = vaddq_s32(registro1, registro2);
```

Ejemplo 3: Ejemplo de instrucción para conversión de tipo: Reinterpretar un vector de 4 enteros de 16 bits como un vector de 4 enteros sin signo

```
int16x4_t valores;
...
uint16x4_t resultados;
resultados =  vreinterpret_u16_s16(valores);
```

Ejemplo 4: Ejemplo de instrucción VMUL con promoción de tipo: Multiplicar dos vectores con 2 elementos de enteros con signo de 32 bits y guardar los dos resultados en un registro Q, como dos enteros con signo de 64 bits

```
int32x2_t registro1, registro2;
...
int64x2_t resultados;
resultados = vamull_s32(registro1, registro2);
```

Para compilar con *intrinsics* es necesario incluir el fichero de cabeceras *arm_neon.h* en nuestro programa, y añadir la opción *-mfpu=neon* al compilador gcc.

5. Dos ejemplos de aplicación real de las instrucciones NEON

Con el fin de mostrar un ejemplo de aplicación de las instrucciones NEON en una aplicación real y común en procesamiento de imagen, en esta sección se muestra la aplicación del filtrado del ruido con un filtro gausiano de $5x5$ y la realización de las operaciones del gradiente aplicando estas instrucciones a través de las *intrinsics*. En los dos ejemplos (ver algoritmo 5 y 6) se representa tanto la matriz de la imagen como los filtros gausiano y *Sobel* en punto flotante. La mejor solución para ahorrar ciclos de reloj por instrucción sería trabajar sobre números enteros de 16 bits (tal y como suele hacer *OpenCV*), no obstante, con estos ejemplos se pretende dar a conocer la eficacia de las operaciones vectoriales de 128 bits sobre elementos de punto flotante de 32 bits.

5.1. Filtrado del ruido

El código mostrado en 5 ejecuta el filtrado del ruido sobre una imagen utilizando un filtro gausiano separado de tamaño 5.

Algoritmo 5: Filtro del ruido

```
1  static void aplicarFiltro_size5(
2  float ** gsFiltradoM)
3  {
4    ...
5
6    float32x4_t nhfiltro, nvfiltro;
7    nhfiltro = vld1q_f32(hfiltro);
8    nvfiltro = vld1q_f32(vfiltro);
9
10     ...
11
12   float * centroFiltro = hfiltro + foffset;
13   float * fp4 = centroFiltro+2;
14
15     //APLICACION FILTRO FILA
16   omp_set_num_threads(THREADS);
17   #pragma omp parallel for schedule(runtime)
18   for(f=0; f < height; f++)
19   {
20     int c;
21     float * inioptr = oM[f];
22     float * iniaptr = wM[f];
23     for(c=foffset; c < maxWidth; c++)
24     {
25       float * sptr = inioptr + (c-2);
26       float32x4_t tmp;
27
28       tmp = vld1q_f32(sptr);
29
30       tmp = vmulq_f32(tmp, nhfiltro);
31
32       float32x2_t low = vget_low_f32(tmp);
33       float32x2_t high = vget_high_f32(tmp);
34
35       low = vadd_f32(low, high);
36
37       float * dptr = iniaptr + c;
38       *dptr = vget_lane_f32(low,0);
39       *dptr += vget_lane_f32(low,1);
40
41       *dptr += *(sptr+4)**fp4;
42     }
43   }
44
45   centroFiltro = vfiltro + foffset;
46   fp4 = centroFiltro+2;
47
48   float32_t column[4];
49
50     //APLICACION FILTRO COLUMNA
51   #pragma omp parallel for schedule(runtime)
       ↪ private(column)
52   for(f=foffset; f < maxHeight; f++)
53   {
54     int c;
55     float * iniaptr0 = wM[f-2],
56     *iniaptr1 = wM[f-1],
57     *iniaptr2 = wM[f],
58     *iniaptr3 = wM[f+1],
59     *iniaptr4 = wM[f+2];
60
61     for(c=0; c < width; c++)
62     {
63       column[0] = *(iniaptr0 + c);
64       column[1] = *(iniaptr1 + c);
65       column[2] = *(iniaptr2 + c);
66       column[3] = *(iniaptr3 + c);
67
68       float32x4_t tmp;
69       tmp = vld1q_f32(column);
70
71       tmp = vmulq_f32(tmp, nvfiltro);
72
73       float32x2_t low = vget_low_f32(tmp);
74       float32x2_t high = vget_high_f32(tmp);
75
76       low = vadd_f32(low, high);
77
78       float * dptr = &gsFiltradoM[f][c];
79       *dptr = vget_lane_f32(low,0);
80       *dptr += vget_lane_f32(low,1);
81
82       *dptr += *(iniaptr4+c)**fp4;
83     }
84   }
85
86 }
```

En primer lugar, se cargan en registros el vector fila y el vector columna que representan el filtro gausiano (líneas 7 y 8, respectivamente). En el primer bucle, situado en la línea 18 se realiza la primera pasada aplicando el filtro como vector horizontal. De 25 a 28 se cargan en un registro los 4 primeros píxeles (el máximo posible en un registro) sobre los que se situa el filtro horizontal. En 30 se realiza la multiplicación los 4 primeros píxeles por los 4 primeros elementos del filtro, uti-

lizando sólo una instrucción vectorial. En 32 y 33 se obtiene una referencia al registro D (de 64 bits) más significativo y al registro D menos significativo, respectivamente, del registro Q (de 128 bits) donde se almacenan los resultados de las multiplicaciones. Estas dos *instrinsics* no se traducen en ninguna instrucción máquina, simplmente sirven de guía al compilador. De 35 a 39 se realizan las sumas de las multiplicaciones. Lo último que queda es realizar la última multiplicación que falta, entre el quinto elemento del filtro y el quinto píxel, y sumar al total el resultado (41). En el segundo bucle (52), donde se realiza la pasada con el filtro columna, se realizan unas operaciones similares, pero aplicando el filtro sobre los elementos de la misma columna.

5.2. Cálculo del gradiente

El código mostrado en 6 ejecuta la aplicación de los filtros *Sobel* para obtener el módulo y dirección del gradiente.

Algoritmo 6: Cálculo del gradiente

```
1  static void _computeGradient(
2  float ** sM,
3  float ** xM, float **  yM,
4  float ** xfiltro, float ** yfiltro,
5  float** mgM, uint8_t ** dgdM,
6  unsigned int height, unsigned int width)
7  {
8    ...
9
10   float32x4_t xff0,xff1,xff2, yff0, yff1, yff2;
11   xff0 = vld1q_f32(xfiltro[0]);
12   xff1 = vld1q_f32(xfiltro[1]);
13   xff2 = vld1q_f32(xfiltro[2]);
14   yff0 = vld1q_f32(yfiltro[0]);
15   yff1 = vld1q_f32(yfiltro[1]);
16   yff2 = vld1q_f32(yfiltro[2]);
17
18   ...
19
20   omp_set_num_threads(THREADS);
21   #pragma omp parallel for schedule(runtime)
         ↪ private(mg, dgd)
22   for(f=foffset; f < maxHeight; f++)
23   {
24     int c;
25
26     mg = mgM[f];
27     dgd = dgdM[f];
28
29     int f0=f-1;
30     float * inisptr = sM[f0];
31     float * xptr = xM[f], *yptr = yM[f];
32
33     for(c=foffset; c < maxWidth; c++)
34     {
35       int c0=c-1;
36
37       //GRADIENTE EN X
38       float32x4_t nsum, tmp0,tmp1,tmp2, resf;
39       nsum = vdupq_n_f32(0);
40
41       float * psM, * colptr = inisptr + c0;
42
43       psM = colptr;
44       tmp0 = vld1q_f32(psM);
45       psM += width;
46       tmp1 = vld1q_f32(psM);
47       psM += width;
48       tmp2 = vld1q_f32(psM);
49
50       resf = vmulq_f32(xff0, tmp0);
51       nsum = vaddq_f32(nsum, resf);
52
53       resf = vmulq_f32(xff1, tmp1);
54       nsum = vaddq_f32(nsum, resf);
55
56       resf = vmulq_f32(xff2, tmp2);
57       nsum = vaddq_f32(nsum, resf);
58
59       float32x2_t nsumlow = vget_low_f32(nsum);
         ↪
60       float32x2_t nsumhigh = vget_high_f32(nsum
         ↪ );
61
62       float x;
63       x = vget_lane_f32(nsumlow,0);
64       x += vget_lane_f32(nsumlow,1);
65       x += vget_lane_f32(nsumhigh,0);
66
67       //FIN GRADIENTE EN X
68       //GRADIENTE EN Y
69       nsum = vdupq_n_f32(0);
70
71       resf = vmulq_f32(yff0, tmp0);
72       nsum = vaddq_f32(nsum, resf);
73
74       resf = vmulq_f32(yff1, tmp1);
75       nsum = vaddq_f32(nsum, resf);
76
77       resf = vmulq_f32(yff2, tmp2);
78       nsum = vaddq_f32(nsum, resf);
79
80       nsumlow = vget_low_f32(nsum);
81       nsumhigh = vget_high_f32(nsum);
82
83       float y;
84       y = vget_lane_f32(nsumlow,0);
85       y += vget_lane_f32(nsumlow,1);
86       y += vget_lane_f32(nsumhigh,0);
87
88       //FIN GRADIENTE EN Y
89       //MODULO Y DIRECCION
90       *mg++ = x*x+y*y;
91
92       *dgd++ =  LOOKUP_DEG[(int)round(y)+
         ↪
93           LOOKUPTABLE_DEG_VMAX][(int)
         ↪ round(x)+
94           LOOKUPTABLE_DEG_VMAX];
95     }
96   }
97 }
```

De 11 a 13 se cargan las filas del filtro *Sobel* para obtener la dirección del gradiente en x en tres registros diferentes. Seguidemente, de 14 a 16 se cargan las filas del filtro *Sobel* para obtener la dirección del gradiente en y en otros tres registros.

En primer lugar, en 39 se establece a cero un registro mediante una *instrinsic*. En este registro se almacenará el resultado de aplicar el filtro *Sobel* para la intensidad en x, en la posición actual en la imagen. De 43 a 48 se cargan en registros los elementos sobre el filtro en la fila anterior, actual y posterior. De 50 a 57 se realizan las seis multiplicaciones de la aplicación del filtro *Sobel* en x con sólo tres instrucciones. De 59 a 65 se realiza la suma de los resultados para obtener la instensidad del gradiente en x. Del mismo modo, las operaciones aplicando el filtro *Sobel* en y se realizan de 71 a 86. Por último, de 90 a 92 se calcula el módulo (simplificado) y dirección del gradiente.

6. Resultados en *Raspberry Pi 2 Model B*

Se ha realizado una comparativa de tiempos de ejecución de la reducción del ruido y del cálculo del gradiente en las figuras 2 y 3, respectivamente, sobre imágenes submarinas con una alta resolución: 1292x964. En las gráficas se muestran los tiempos de ejecución según el número de hilos empleados en la paralelización de los bucles, usando la versión sin instrucciones NEON y la versión presentada en los ejemplos. Claramente se puede comprobar como, con el aprovechamiento de los cuatro núcleos del mini-computador, y el uso adecuado de las instrucciones *NEON* es posible reducir en gran medida los tiempos de ejecución.

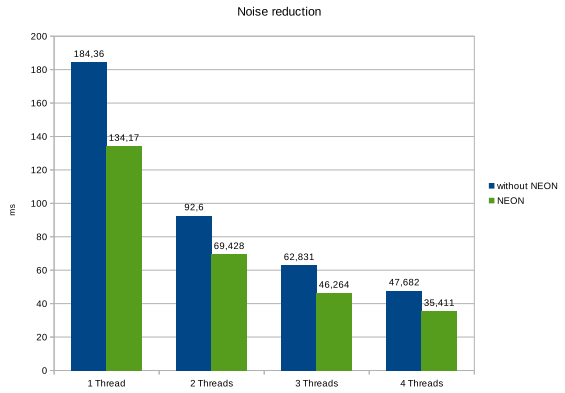

Figura 2: Tiempos de ejecución de la reducción del ruido con paralelización, con NEON y sin NEON

7. HROV basado en ARMv7-A para el seguimiento autónomo supervisado de una tubería submarina

Una posible aplicación real de un HROV que sea viable de implementar en un mini-computador similar a una Raspberry Pi 2 podría ser el seguimiento autónomo de una tubería submarina supervisado por un operador en la superficie (ver Fig. 5 y 6)

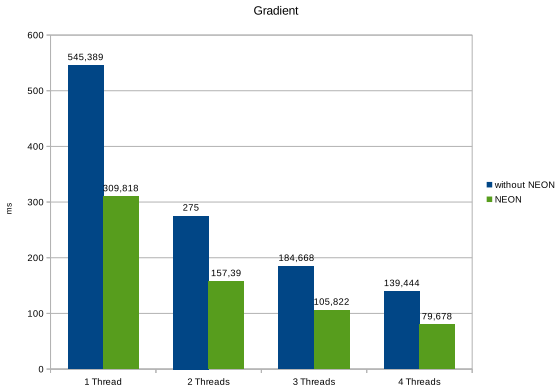

Figura 3: Tiempos de ejecución de las operaciones de gradiente con paralelización, con NEON y sin NEON

Una configuración posible del sistema sería la utilización de uno de los núcleos del procesador para la captura, compresión y envío de la información visual comprimida, además de la recepción de los comandos de alto nivel por parte del operador o supervisor. Según los resultados obtenidos en [6], conseguiríamos entre 20 y 30 *frames* por segundo para una correcta supervisión del proceso, sin considerar un posible cuello de botella en el enlace de comunicación. Los otros tres núcleos restantes servirían para el procesamiento de la imagen necesario para la obtención de la dirección de la tubería y el control automático de los motores. Los pasos requeridos para la obtención de la recta más significativa de una tubería consisten, por este orden, en la reducción del ruido, la aplicación del detector de bordes *Canny* y el cálculo de la transformada de *Hough*, siendo el coste de las dos primeras etapas independiente del contenido de la imagen capturada. La figura 4 muestra los tiempos de ejecución de cada etapa del algoritmo tras realizar una experimentación con 17 imágenes reales, de 646x482, de tuberías submarinas como la mostrada en 5, usando tres de los cuatro núcleos de una *Raspberry Pi 2 Model B*. Tanto en la etapa de la reducción del ruido, como en la del cálculo del gradiente (esta última forma parte de *Canny*) se ha utilizado la implementación con *instrinsics* mostrada en este documento. Los resultados muestran un tiempo total de 55.7 ms, por lo que, muy probablemente, sería viable la implmentación de un HROV basado en esta arquitectura para el seguimiento autónomo y en tiempo real de una tubería con supervisión de un operador. Además, la implementación de *Canny* realizada por *OpenCV* es to-

davía más rápida que la utilizada en este trabajo, puesto que *OpenCV* realiza un uso más eficiente de la caché en su implementación.

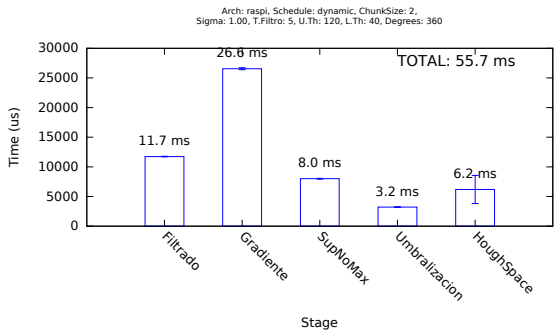

Figura 4: Tiempos de ejecución del algoritmo para la detección de la dirección de la recta más significativa de imágenes de resolución 646x482 usando tres núcleos

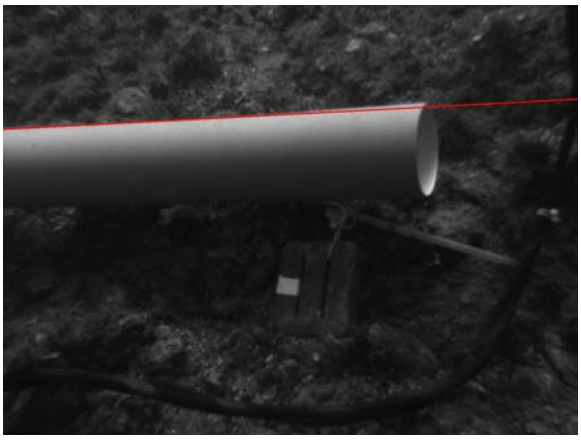

Figura 5: Vista real de una tubería submarina con la recta más significativa detectada tras aplicar la transformada de *Hough*

8. Conclusiones

En este trabajo se ha demostrado la viabilidad del uso de los mini-ordenadores de bajo consumo basados en la arquitectura ARMv7-A o similar para la implementación de robots submarinos cuya misión es realizar una tarea específica y simple de forma semiautónoma. El objetivo de este trabajo ha sido dar a conocer el potencial del juego de instrucciones SIMD de la arquitectura ARMv7-A y el uso de las mismas a través de las *instrinsics*. Además, se han mostrado dos ejemplos de algorítmos típicos donde se combinan las SIMD con las directivas de paralelización de *OpenMP*. También se ha comprobado la viabilidad de un HROV para el seguimiento autónomo y supervisado de una tubería utilizando una *Raspberry Pi 2*

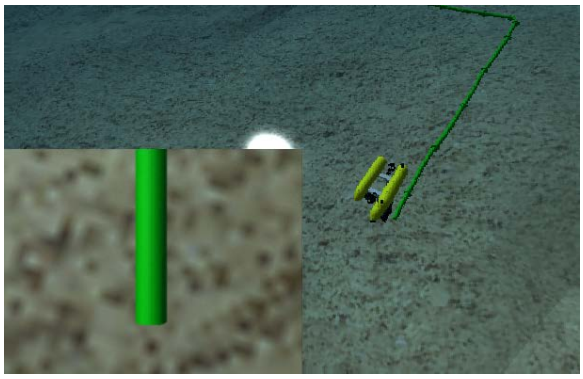

Figura 6: Vista del simulador UWSIM de una tarea de *Pipe Following*

Model o similar como 'cerebro' del robot. Los resultados demuestran que, con un poco de esfuerzo de programación a medida para una arquitectura específica, es posible reducir drásticamente los tiempos de ejecución de los algoritmos típicos de procesado de imagen usados en robótica.

Referencias

[1] Belloch, J. A., González, A., Igual, F. D., Mayo, R., and Quintana-Orti, E. S. (2015). Vectorization of binaural sound virtualization on the arm cortex-a15 architecture. In *Signal Processing Conference (EUSIPCO), 2015 23rd European*, pages 1601–1605.

[2] Mitra, G., Johnston, B., Rendell, A. P., McCreath, E., and Zhou, J. (2013). Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1107–1116.

[3] Pelekanakis, C., Stojanovic, M., and Freitag, L. (2003). High rate acoustic link for underwater video transmission. In *OCEANS 2003. Proceedings*, volume 2, pages 1091–1097 Vol.2.

[4] Pérez, J., Sales, J., Prats, M., Martí, J. V., Fornas, D., Marín, R., and Sanz, P. J. (2013). The underwater simulator UWSim: Benchmarking capabilities on autonomous grasping. In *11th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*.

[5] Ribas, J., Sura, D., and Stojanovic, M. (2010). Underwater wireless video transmission for supervisory control and inspection using acoustic ofdm. In *OCEANS 2010 MTS/IEEE SEATTLE*, pages 1–9.

[6] Rubino, E. M., Centelles, D., Sales, J., Martı,

J. V., Marın, R., and Sanz, P. J. (2015). Image compression with region of interest for underwater robotic archaeological applications. In *XXXVI Jornadas de Automática, CEA-IFAC*, Bilbao, Spain.

[7] Sanz, P. J., Prats, M., Ridao, P., Ribas, D., Oliver, G., and Orti, A. (2010). Recent progress in the RAUVI project. A reconfigurable autonomous underwater vehicle for intervention. In *52-th International Symphosium ELMAR-2010*, pages 471–474, Zadar, Croatia.