Evaluación de la práctica 1: Fundamentos de Vectorización en x86
30237 Multiprocesadores - Grado Ingeniería Informática
Esp. en Ingeniería de Computadores Universidad de Zaragoza

Sergio García Esteban

3-marzo-2020

Resumen

Para la evaluación de la práctica 1 vais a resolver algunas cuestiones correspondientes a los puntos 1.4, 1.6 y 2.2 del guión de prácticas. Los tiempos y métricas deberán obtenerse para las máquinas del laboratorio L0.04. Sed concisos en las respuestas. Se valorarán las referencias utilizadas.

Notas generales

El trabajo puede presentarse de forma individual o en grupos de máximo dos personas. Podéis trabajar en grupos mayores, pero **cada grupo debe elaborar el material a entregar de forma independiente**. Hacedme llegar vuestros trabajos **en formato pdf** a través de la entrega habilitada en la web de la asignatura (moodle). Incluid vuestro nombre y apellidos en la cabecera del documento y vuestro NIP en el nombre del fichero (p1_NIP.pdf).
Plazo límite de entrega: domingo 8 de marzo, 23h59m59s.

Parte 1. Vectorización automática

4. ¿Cuántas instrucciones se ejecutan en el bucle interno (esc.avx, vec.avx, vec.avxfma y vec.avx512)?

```
for (int i = 0; i < LEN; i++)
    x[i] = alpha*x[i] + beta
```

Calcula la reducción en el número de instrucciones respecto la versión esc.avx.

versión	icount	reducción(%)	reducción(factor)
esc.avx	6144	0	1.0
vec.avx	768	87,5	8.0
vec.avxfma	768	87,5	8.0
vec.avx512	384	93,75	16.0

Indica muy brevemente cómo has calculado los anteriores valores.

`En escalar, 6 instrucciones por iteración y 1024 elementos, 6144 instrucciones. Las versiones avx y avxfma calculan el resultado de 8 elementos del vector en cada iteración del bucle y avx512 16 elementos, necesitarán 8 y 16 veces menos de iteraciones del bucle para calcular todo el vector.`

5. A partir de los tiempos de ejecución obtenidos [...], calcula las siguientes métricas para todas las versiones ejecutadas:

   - Aceleraciones (*speedups*) de las versiones vectoriales sobre sus escalares (vec.avx y vec.avxfma respecto esc.avx).
   - Rendimiento (R) en GFLOPS.
   - Rendimiento pico (R_{pico}) teórico de un núcleo (*core*), en GFLOPS. Para las versiones escalares, considerar que las unidades funcionales trabajan en modo escalar. Considerar asimismo la capacidad FMA de las unidades funcionales solamente para las versiones compiladas con soporte FMA.
   - Velocidad de ejecución de instrucciones (V_I), en Ginstrucciones por segundo (GIPS).

   Indica brevemente cómo has realizado los cálculos.

versión	tiempo(ns)	speed-up	R(GFLOPS)	R_{pico}(GFLOPS)	V_I(GIPS)
esc.avx	466.9	1.0	4,386	7,4	13,15
vec.avx	77.1	6.05	26,562	59,2	9,79
vec.avxfma	70.5	6.62	29,049	118,4	10,89

Notas: GFLOPS = 10^9 FLOPS. GIPS = 10^9 IPS.

```
tiempo   -> medido en ejecución.
speed-up -> tiempo base / nuevo tiempo
R        -> FLOPs / tiempo(ns)
R~pico~  -> UF * ops/UF * CPU~freq~ (2 UF y 3,7 GHz)
V~I~     -> icount / tiempo
```

¿La velocidad de ejecución de instrucciones es un buen indicador de rendimiento?

`No, ya que todas las intrucciones no son igual de productivas ni igual de costosas.`

Parte 2. Vectorización manual mediante intrínsecos

2. Escribe una nueva versión del bucle, `ss_intr_AVX()`, vectorizando de forma manual con intrínsecos AVX. Lista el código correspondiente a la función `ss_intr_AVX()`.

```
__attribute__ ((noinline))
int ss_intr_AVX()
{
  double start_t, end_t;

  init();
  start_t = get_wall_time();

#if PRECISION==0
    __m256 vX, valpha, vbeta;
    for (unsigned int nl = 0; nl < NTIMES; nl++)
    {
        valpha = _mm256_set1_ps(alpha);     // valpha = _mm_load1_ps(&alpha);
        vbeta = _mm256_set1_ps(beta);
        for (unsigned int i = 0; i < LEN; i+= AVX_LEN)
        {
            vX = _mm256_load_ps(&x[i]);
            vX = _mm256_mul_ps(valpha, vX);
            vX = _mm256_add_ps(vX, vbeta);
            _mm256_store_ps(&x[i], vX);
        }
        dummy(x, alpha, beta);
    }
#else
    __m256d vX, valpha, vbeta;
```

```
    for (unsigned int nl = 0; nl < NTIMES; nl++)
    {
        valpha = _mm256_set1_pd(alpha);     // valpha = _mm_load1_ps(&alpha);
        vbeta = _mm256_set1_pd(beta);
        for (unsigned int i = 0; i < LEN; i+= AVX_LEN)
        {
            vX = _mm256_load_pd(&x[i]);
            vX = _mm256_mul_pd(valpha, vX);
            vA = _mm256_add_pd(vX, vbeta);
            _mm256_store_pd(&x[i], vA);
        }
        dummy(x, alpha, beta);
    }
#endif

  end_t = get_wall_time();
  results(end_t - start_t, "ss_intr_AVX");
  check(x);
  return 0;
}
```

Analiza el fichero que contiene el ensamblador de dicha función y busca las instrucciones correspondientes al bucle en ‘ss_intr_AVX()‘.

```
scale_shift()
400740: c5 e4 59 00             vmulps (%rax),%ymm3,%ymm0
400744: 48 83 c0 20             add    $0x20,%rax
400748: c5 fc 58 c2             vaddps %ymm2,%ymm0,%ymm0
40074c: c5 fc 29 40 e0          vmovaps %ymm0,-0x20(%rax)
400751: 48 39 c3                cmp    %rax,%rbx
400754: 75 ea                   jne    400740 <scale_shift+0x50>
ss_intr_AVX()
4008c0: c5 e4 59 00             vmulps (%rax),%ymm3,%ymm0
4008c4: 48 83 c0 20             add    $0x20,%rax
4008c8: c5 fc 58 c2             vaddps %ymm2,%ymm0,%ymm0
4008cc: c5 fc 29 40 e0          vmovaps %ymm0,-0x20(%rax)
4008d1: 48 39 c3                cmp    %rax,%rbx
4008d4: 75 ea                   jne    4008c0 <ss_intr_AVX+0x50>
```

¿Hay alguna diferencia con las instrucciones correspondientes al bucle en ‘scale_shift()‘ (versión vec.avx)?

Las instrucciones del bucle son las mismas.

¿Hay diferencia en el rendimiento de las funciones ‘scale_shift()‘ (versión vec.avx) y ‘ss_intr_AVX()‘?

El tiempo obtenido en la ejecución es 75,1 ns en scale_shift() y 73,3 ns en ss_intr_AVX(), tienen rendimientos muy similares.