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

Abdelbassat Chiguer 871208 y Hugo Cornago 873840

Febrero-2025

Parte 1. Vectorización automática

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

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

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	93.7	16.0

Indica brevemente cómo has calculado los anteriores valores (fórmulas utilizadas). Necesitas conseguir los siguientes valores del desensamblador:

iteraciones = valor_iterador(rax) / incremento_por_iteracion

instrucciones = 6

Despues, calcula cada campo con la siguiente formula expresada en python:

```
esc_avx_cicles = (0x1000//4)*6 # 0x1000 iteraciones, 4 incremento, 6 instrucciones
cicles = instructions*iters
```

```
factor_reduccion = esc_avx_cicles/cicles
reduccion_cien = 100*((esc_avx_cicles-cicles)/esc_avx_cicles)
print(f"cicles: {cicles}, cien: {reduccion_cien}%, factor: {factor_reduccion}")
```

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 (fórmulas utilizadas).

   Se han utilizado las siguientes fórmulas, cabe destacar que para la tabla ha habido que hacer conversiones de las diferentes unidades utilizadas.

   Es importante también recalcar que para el rendimiento, se ha multiplicado el tiempo del bucle interno por el número de veces que se ejecuta el bucle externo, puesto que se nos proporciona el número de flops de todo el programa no solo los ejecutados en el bucle interno.

   rendimiento = flops / tiempo

   rpico = unidades funcionales * frecuencia procesador (4,4 GHZ)

   speedup = tiempo peor / tiempo mejor

   velocidad = instrucciones / tiempo

versión	tiempo(ns)	speed-up	R(GFLOPS)	R_{pico}(GFLOPS)	V_I(GIPS)
esc.avx	505,6	1,0	4,06	8,8	12,1
vec.avx	68,2	7,41	30,07	35,2	11,26
vec.avxfma	45,7	11,06	44,87	70,4	16,8

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

   ¿La velocidad de ejecución de instrucciones es un buen indicador de rendimiento? La velocidad de ejecución de instrucciones no siempre es un buen indicador de rendimiento, ya que no mide cuántas operaciones útiles realiza una instrucción. En cómputo vectorizado, una instrucción puede ejecutar múltiples operaciones a la vez, por lo que un menor VI puede acompañarse de un mayor rendimiento.

Parte 2. Vectorización manual mediante intrínsecos

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

```
#if 1
__attribute__ ((noinline))
int axpy_intr_AVX()
{
  double start_t, end_t;

  init();
  start_t = get_wall_time();

#if PRECISION==0
  __m256 vX, vY, valpha, vaX;
  for (unsigned int nl = 0; nl < NTIMES; nl++) {
      valpha = _mm256_set1_ps(alpha);
      for (unsigned int i = 0; i < LEN; i += AVX_LEN) {
        vX = _mm256_load_ps(&x[i]);
        vY = _mm256_load_ps(&y[i]);
        vaX = _mm256_mul_ps(valpha, vX);
        vY = _mm256_add_ps(vaX, vY);
        _mm256_store_ps(&y[i], vY);
    }
    dummy(x,y,alpha);
  }
#else
  __m256d vX, vY, valpha, vaX;
  for (unsigned int nl = 0; nl < NTIMES; nl++) {
      valpha = _mm256_set1_pd(alpha);
      for (unsigned int i = 0; i < LEN; i += AVX_LEN) {
        vX = _mm256_load_pd(&x[i]);
        vY = _mm256_load_pd(&y[i]);
        vaX = _mm256_mul_pd(valpha, vX);
        vY = _mm256_add_pd(vaX, vY);
        _mm256_store_pd(&y[i], vY);
    }
    dummy(x,y,alpha);
  }
#endif

  end_t = get_wall_time();
  results(end_t - start_t, "axpy_intr_AVX");
  check(y);
```

```
  return 0;
}
#endif
```