Multiprocesadores
% Práctica 1: Fundamentos de Vectorización en x86: Extensiones Vectoriales, Vectorización Automática y Manual
30237 Multiprocesadores. Grado Ingeniería Informática. Esp. en Ingeniería de Computadores

Barea López, Daniel

23-febrero-2017

Tiempo dedicado (aproximado): 4 horas

Plataforma de trabajo

Se han realizado las pruebas sobre la máquina `lab004-071`:

- **CPU:** Intel i5-4570 (soporta las extensiones SSE4.1/4.2 y AVX 2.0)
- **SO:** CentOS 6, kernel 2.6.32
- **Compilador:** GCC versión 6.2.0

Parte 1. Vectorización automática

2. **Compilar con `gcc` las versiones escalares (noavx, noavx512) y vectoriales (avx, avx+fma, avx512) del programa `axpy.c`. ¿Ha vectorizado el bucle en `axpy()`?**

   De acuerdo con la salida del compilador, se ha vectorizado el bucle para las versiones AVX, AVX+FMA y AVX512.

3. **Análisis del ensamblador generado para la versión escalar (noavx):**

```
400bd8:   vmovsd %xmm0,0x8(%rsp)
400bde:   xchg   %ax,%ax
400be0:   vmovss 0x2014a0(%rip),%xmm0        # 602088 <alpha>
400be8:   xor    %eax,%eax
400bea:   nopw   0x0(%rax,%rax,1)
400bf0:   vmulss 0x604100(%rax),%xmm0,%xmm1
400bf8:   add    $0x4,%rax
400bfc:   vaddss 0x6060fc(%rax),%xmm1,%xmm1
400c04:   vmovss %xmm1,0x6060fc(%rax)
```

   Como se puede observar, el compilador ha utilizado las instrucciones `vmulss` y `vaddss` para realizar las operaciones de multiplicación y suma, respectivamente.

   La lectura de los elementos se realiza con las mismas instrucciones `vmulss` y `vaddss`, especificando las direcciones de los vectores como un operando más (`0x604100+%rax` para `x[i]` y `0x6060fc+%rax` para `y[i]`).

   La escritura en memoria se realiza con la instrucción `vmovss`, que almacena en la dirección `0x6060fc+%rax` el resultado de la operación.

   En cada iteración se procesa **1 elemento** de **4 bytes** de cada vector (**4 bytes/iteración**).

4. **Análisis del ensamblador generado para las versiones vectoriales (avx, avx+fma y avx512):**

   a. AVX

```
400beb:   vmovsd %xmm0,-0x18(%rbp)
400bf0:   vmovss 0x201490(%rip),%xmm0        # 602088 <alpha>
400bf8:   xor    %eax,%eax
400bfa:   vbroadcastss %xmm0,%ymm2
400bff:   nop
400c00:   vmulps 0x604100(%rax),%ymm2,%ymm1
400c08:   add    $0x20,%rax
400c0c:   vaddps 0x6060e0(%rax),%ymm1,%ymm1
400c14:   vmovaps %ymm1,0x6060e0(%rax)
```

   La instrucción utilizada para realizar la suma es `vaddps`, y para la multiplicación `vmulps`.

   La lectura de los elementos se realiza con las mismas instrucciones `vaddps` y `vmulps`, especificando las direcciones iniciales de los vectores como un operando más (`0x604100+%rax` para `x[i]` y `0x6060e0+%rax` para `y[i]`).

   La escritura en memoria se realiza con la instrucción `vmovaps`, que almacena a partir de la dirección `0x6060e0+%rax` los resultados de las operaciones.

   En cada iteración se procesan **8 elementos** de **4 bytes** de cada vector (**32 bytes/iteración**).

   b. AVX + FMA

```
400bfb:   vmovsd %xmm0,-0x18(%rbp)
400c00:   vmovss 0x201480(%rip),%xmm0        # 602088 <alpha>
400c08:   xor    %eax,%eax
400c0a:   vbroadcastss %xmm0,%ymm2
400c0f:   nop
400c10:   vmovaps 0x604100(%rax),%ymm1
400c18:   add    $0x20,%rax
400c1c:   vfmadd213ps 0x6060e0(%rax),%ymm2,%ymm1
400c25:   vmovaps %ymm1,0x6060e0(%rax)
```

   La instrucción utilizada para realizar la suma y la multiplicación es `vfmadd213ps`. Esta instrucción multiplica `%ymm2` (vector de `alpha`) y `%ymm1` (vector con `x[i]..x[i+8]`), suma `0x6060e0+%rax` (`&y[i]`) para cada elemento y almacena el resultado en `%ymm1`.

   La lectura de los elementos se realiza con la instrucción `vmovaps`, para leer `x[i]..x[i+8]` en el registro vectorial `%ymm1` y con la misma instrucción `vfmadd213ps` para los elementos del vector `y`.

   La escritura en memoria se realiza con la instrucción `vmovaps`, que almacena a partir de la dirección `0x6060e0+%rax` los resultados de las operaciones (almacenados en el registro `%ymm1`).

   En cada iteración se procesan **8 elementos** de **4 bytes** de cada vector (**32 bytes/iteración**).

   c. AVX-512

```
400beb:   vmovsd %xmm0,-0x38(%rbp)
400bf0:   vmovss 0x201490(%rip),%xmm0        # 602088 <alpha>
400bf8:   xor    %eax,%eax
400bfa:   vbroadcastss %xmm0,%zmm2
400c00:   vmulps 0x604100(%rax),%zmm2,%zmm1
400c0a:   add    $0x40,%rax
400c0e:   vaddps 0x6060c0(%rax),%zmm1,%zmm1
400c18:   vmovaps %zmm1,0x6060c0(%rax)
```

   La instrucción utilizada para realizar la suma es `vaddps`, y para la multiplicación `vmulps`.

   La lectura de los elementos se realiza con las mismas instrucciones `vaddps` y `vmulps`, especificando las direcciones iniciales de los vectores como un operando más (`0x604100+%rax` para `x[i]` y `0x6060c0+%rax` para `y[i]`).

   La escritura en memoria se realiza con la instrucción `vmovaps`, que almacena a partir de la dirección `0x6060c0+%rax` los resultados de las operaciones.

En cada iteración se procesan **16 elementos** de **4 bytes** de cada vector (**64 bytes/iteración**).

La diferencia con el código para AVX es que AVX-512 utiliza los registros vectoriales `%zmm0..%zmm31`, que son más grandes (512 bits).

5. **¿Qué ocurre al ejecutar la versión AVX-512?**

   El procesador i5-4570 no tiene soporte para AVX-512, por lo que al intentar ejecutar las instrucciones de su repertorio produce una excepción. Por ahora solamente los procesadores Intel Xeon Phi tienen soporte para el conjunto de instrucciones AVX-512.

6. **Cálculo de métricas de ejecución:**

   Al ejecutar las diferentes versiones se producen los siguientes resultados:

Versión	Tiempo ejec	Speedup	GFLOPS
No AVX	5.01	1.0	0.399
AVX	0.76	6.592	2.632
AVX+FMA	0.65	7.708	3.077

   La versión escalar es claramente la perdedora de la comparación, mientras que la que mejor resultado obtiene es la versión AVX+FMA. Los *checksums* de todas las versiones coinciden.

Parte 2. Vectorización manual mediante intrínsecos

7. **¿Hay alguna indicación de que el compilador haya vectorizado el bucle en `axpy_intr_SSE()`?**

   Los informes del compilador no indican haber vectorizado el bucle en `axpy_intr_SSE()`.

8. **Escribir una nueva versión del bucle `axpy_intr_AVX()` vectorizando de forma manual con intrínsecos AVX. Análisis del código generado:**

   Para elementos de precisión simple, el código en C que se ha escrito es el siguiente:

```
__m256 vX, vY;
__m256 valpha, vaX;
for (int nl = 0; nl < NTIMES; nl++) {
  valpha = _mm256_set1_ps(alpha);
  for (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, z, alpha);
}
```

   El desensamblado resultante de ese código corresponde a:

```
  400d3b:   vmovsd %xmm0,-0x18(%rbp)
  400d40:   vmovss 0x201340(%rip),%xmm0        # 602088 <alpha>
  400d48:   xor    %eax,%eax
  400d4a:   vbroadcastss %xmm0,%ymm2
  400d4f:   nop
  400d50:   vmulps 0x604100(%rax),%ymm2,%ymm1
  400d58:   add    $0x20,%rax
  400d5c:   vaddps 0x6060e0(%rax),%ymm1,%ymm1
  400d64:   vmovaps %ymm1,0x6060e0(%rax)
```

El código generado vectorizando de forma manual con los intrínsecos AVX es idéntico al generado automáticamente.

La única diferencia (en el código C) es que el bucle más interno itera dando saltos de `AVX_LEN` elementos (`for (int i = 0; i < LEN; i += AVX_LEN)`), mientras que el de la versión automática itera todos los elementos (`for (int i = 0; i < LEN; i++)`). En la versión automática, el compilador se encarga de ajustar los índices del bucle automáticamente, mientras que en la manual se debe ajustar a mano.

Parte 3. Parte optativa

9. **Análisis para vectores con elementos de doble precisión:**

   Se han repetido los cálculos para elementos de doble precisión y los resultados son los siguientes (todos los *checksums* coinciden de nuevo):

Versión	Tiempo ejec	Speedup	GFLOPS
No AVX	5.08	1.0	0.394
AVX	1.50	3.387	1.333
AVX+FMA	1.25	4.064	1.600

   Como se puede apreciar, el *speedup* obtenido al usar las extensiones vectoriales ha disminuido casi a la mitad, tanto en la versión AVX como en la AVX+FMA. La versión escalar, sin embargo, no ha modificado prácticamente su tiempo de ejecución. Esto se debe a que el procesador tarda lo mismo en realizar una operación sobre un elemento de 4 u 8 bytes, pero no puede procesar más de 32 bytes a la vez.

   En este caso, las extensiones vectoriales trabajan en cada iteración con vectores de **4 elementos** de **8 bytes** (**32 bytes/iteración**).

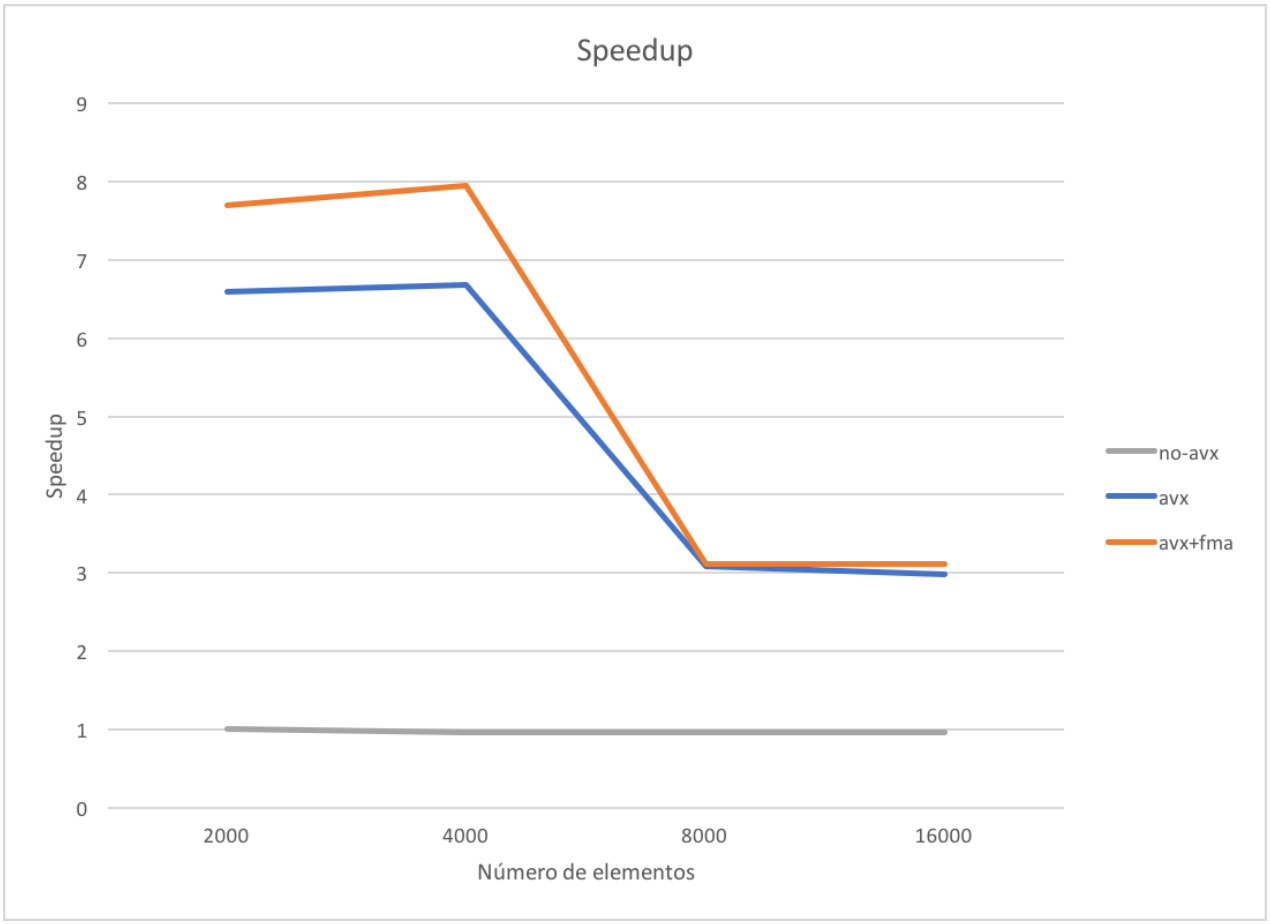
   La diferencia en el código generado para las versiones AVX y AVX+FMA es que se usan las versiones de doble precisión de las instrucciones del repertorio (`vaddpd` en lugar de `vaddps`, `vmuld` en lugar de `vmuls`, `vfmadd213pd` en lugar de `vfmadd213ps`, etc.).

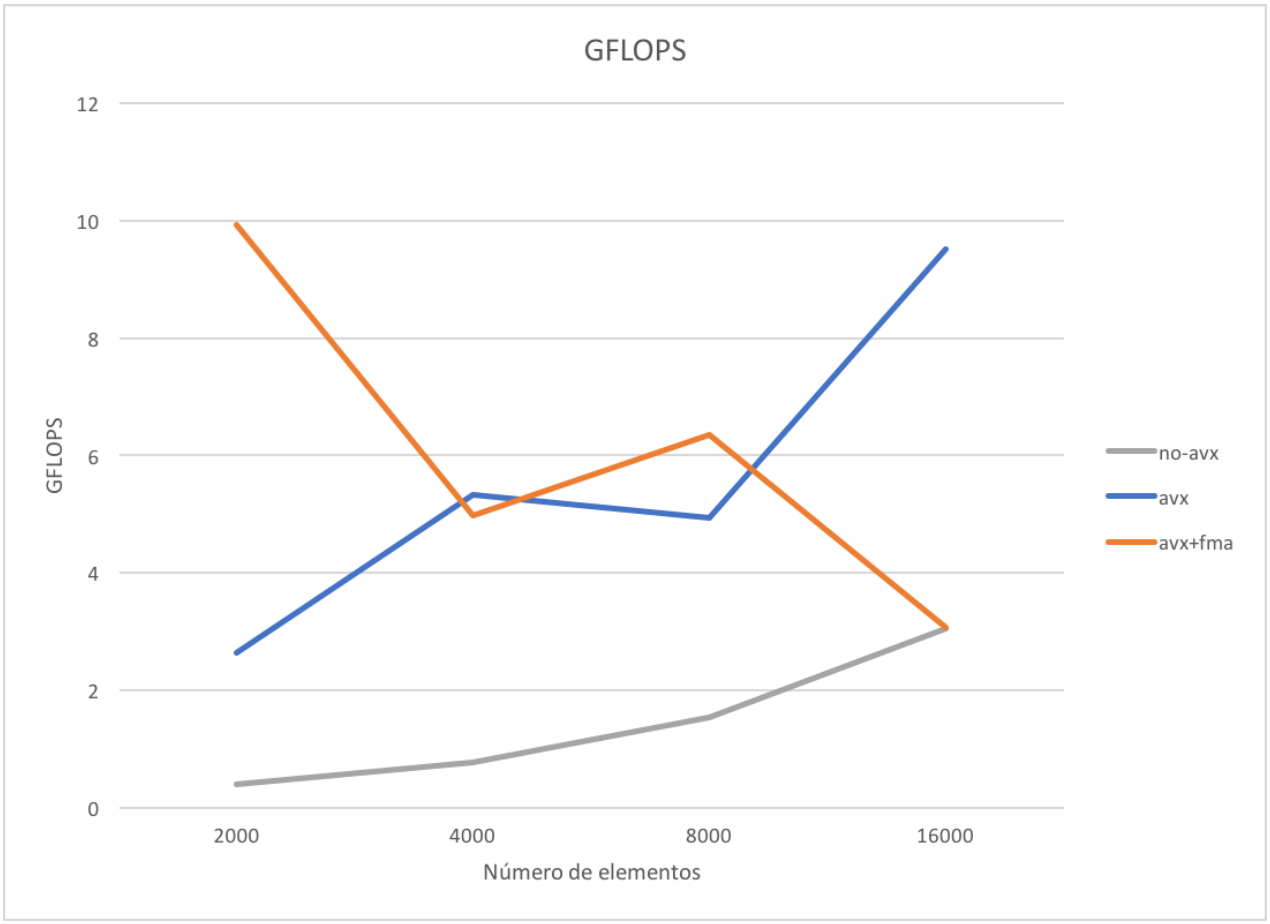
10. **Análisis del rendimiento para diferentes tamaños de vector**

    Para la función `axpy` se analizan vectores de diferentes tamaños (el speedup es en todos los casos en función del tiempo de ejecución de la versión escalar con 2000 elementos):

Núm. elementos	Versión	Tiempo ejec	Speedup	GFLOPS
2000	No AVX	5.01	1.000	0.399
4000	No AVX	5.23	0.958	0.765
8000	No AVX	5.20	1.538	1.538
16000	No AVX	5.23	0.958	3.059
2000	AVX	0.76	6.592	2.632
4000	AVX	0.75	6.680	5.333
8000	AVX	1.62	3.093	4.938
16000	AVX	1.68	2.982	9.524
2000	AVX+FMA	0.65	7.708	3.077
4000	AVX+FMA	0.63	7.952	6.349
8000	AVX+FMA	1.61	3.112	4.969
16000	AVX+FMA	1.61	3.112	9.938

El rendimiento mejora bastante al aumentar el tamaño de vector. Sin embargo, no se han observado una diferencia notable en el tiempo de ejecución (en algunos casos, incluso disminuye al aumentar el número de elementos del vector), por lo que es posible que haya algún error en las mediciones.





Referencias

- Intel Intrinsics Guide