Análisis del rendimiento del kernel scale and shift en un sistema x86

Jesús Alastruey Benedé y Pablo Ibáñez Marín
Área Arquitectura y Tecnología de Computadores
Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Marzo-2024

1 Introducción

Queremos estudiar las prestaciones del kernel `scale_shift` en un sistema x86:

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

Para ello, vamos a analizar el rendimiento de dos versiones vectorizadas: avx2 y avx2+fma. Ejecutaremos el kernel múltiples veces para facilitar la medida de tiempos y su análisis con `perf`.

2 Compilación

Se ha compilado el código con gcc 12.2.1. El siguiente enlace muestra el código fuente relevante y el ensamblador de ambas versiones:

https://godbolt.org/z/xbfjbrn9r

Las órdenes ejecutadas y los códigos máquina y ensamblador del kernel correspondientes a cada versión son:

2.1 avx2

```
$ gcc -O3 -mavx2 ... -o ss.1k.single.vec.avx.gcc
```

```
$ objdump -Sd ss.1k.single.vec.avx.gcc
```

```
  4012d8:   c5 e4 59 00             vmulps (%rax),%ymm3,%ymm0
  4012dc:   48 83 c0 20             add    $0x20,%rax
  4012e0:   c5 fc 58 c2             vaddps %ymm2,%ymm0,%ymm0
  4012e4:   c5 fc 29 40 e0          vmovaps %ymm0,-0x20(%rax)
  4012e9:   48 39 c3                cmp    %rax,%rbx
  4012ec:   75 ea                   jne    4012d8 <scale_shift+0x58>
```

El código del bucle ocupa 22 bytes. Las instrucciones decodificadas se convierten en 7 uops.

2.2 avx+fma

```
$ gcc -O3 -mavx2 -mfma -ffast-math ... -o ss.1k.single.vec.avxfma.gcc
```

```
$ objdump -Sd ss.1k.single.vec.avxfma.gcc
```

```
  4012f8:   c5 fc 28 c3             vmovaps %ymm3,%ymm0
  4012fc:   c4 e2 6d 98 00          vfmadd132ps (%rax),%ymm2,%ymm0
  401301:   48 83 c0 20             add    $0x20,%rax
  401305:   c5 fc 29 40 e0          vmovaps %ymm0,-0x20(%rax)
  40130a:   48 39 c3                cmp    %rax,%rbx
  40130d:   75 e9                   jne    4012f8 <scale_shift+0x58>
```

El código del bucle ocupa 23 bytes, uno más que la versión avx. Las instrucciones decodificadas se convierten en 7 uops.

3 Ejecución

Cuando ejecutamos ambas versiones en un núcleo de un sistema con un procesador intel i5-9500 [1] (octava generación, Coffee Lake [2]), modelo 158 (0x9E), stepping 10 (0xA), con `x[]` siendo un vector de 1024 elementos de tipo float, se obtienen los siguientes resultados:

Tabla 1: Resultados de ejecución.

Versión	tiempo(ns)	R(GFLOPS)
AVX2	45.1	45.4
AVX2+FMA	64.4	31.8

El rendimiento de la versión avx+fma es un 50% peor de lo esperado.

4 Análisis estático

Los resultados experimentales no concuerdan con los los proporcionados por la herramienta uiCA [3], que estima un rendimiento para ambas versiones de 1.25 ciclos por iteración:

- AVX2: https://bit.ly/3SXesKK
- AVX2+FMA: https://bit.ly/3SFbqd1

Recordar que esta herramienta supone condiciones ideales de ejecución (*frontend* sin atascos).

5 Análisis dinámico

Vamos a analizar la ejecución de el código con la herramienta `toplev` [4], que permite aplicar la metodología TMAM [5]. En primer lugar, recogemos estadísticas de ejecución de ambos códigos:

```
$ toplev.py -l1 -v --no-desc -- binario
```

Tabla 2: Estadísticas de primer nivel. Las unidades de todas las métricas son %slots.

Versión	Frontend_Bound	Bad_Speculation	Backend_Bound	Retiring
AVX2	6.7	7.3	1.6	84.5
AVX2+FMA	36.0	3.7	1.3	59

El valor de la métrica FE (Frontend_Bound) es mayor para el código avx+fma.

```
$ toplev --describe Frontend_Bound^
Frontend_Bound
    This category represents fraction of slots where the
    processor's Frontend undersupplies its Backend. Frontend
    denotes the first part of the processor core responsible to
    fetch operations that are executed later on by the Backend
    part. Within the Frontend; a branch predictor predicts the
    next address to fetch; cache-lines are fetched from the
    memory subsystem; parsed into instructions; and lastly
    decoded into micro-operations (uops). Ideally the Frontend
    can issue Pipeline_Width uops every cycle to the Backend.
    Frontend Bound denotes unutilized issue-slots when there is
    no Backend stall; i.e. bubbles where Frontend delivered no
    uops while Backend could have accepted them. For example;
    stalls due to instruction-cache misses would be categorized
    under Frontend Bound.
```

Ahora nos vamos a centrar en la versión avx+fma. `toplev` nos sugiere una nueva ejecución especificando métricas de segundo nivel para obtener más información acerca del problema con el front-end:

```
$ toplev.py --nodes '!+Frontend_Bound*/2,+MUX' -v --no-desc -- ss.1k.single.vec.avxfma.gcc
[...]
```

```
FE       Frontend_Bound                  % Slots                36.0
FE       Frontend_Bound.Fetch_Latency    % Slots                 0.8  <
FE       Frontend_Bound.Fetch_Bandwidth  % Slots                35.2   <==
[...]
Run toplev --describe Fetch_Bandwidth^ to get more information on bottleneck
Add --run-sample to find locations
Add --nodes '!+Fetch_Bandwidth*/3' for breakdown.
```

La métrica señalada como origen del atasco es el ancho de banda del fetch:

```
$ toplev --describe Fetch_Bandwidth^
Frontend_Bound.Fetch_Bandwidth
    This metric represents fraction of slots the CPU was stalled
    due to Frontend bandwidth issues.  For example;
    inefficiencies at the instruction decoders; or restrictions
    for caching in the DSB (decoded uops cache) are categorized
    under Fetch Bandwidth. In such cases; the Frontend typically
    delivers suboptimal amount of uops to the Backend.
```

El esquema del front-end del segmentado puede facilitar el análisis de esta métrica:

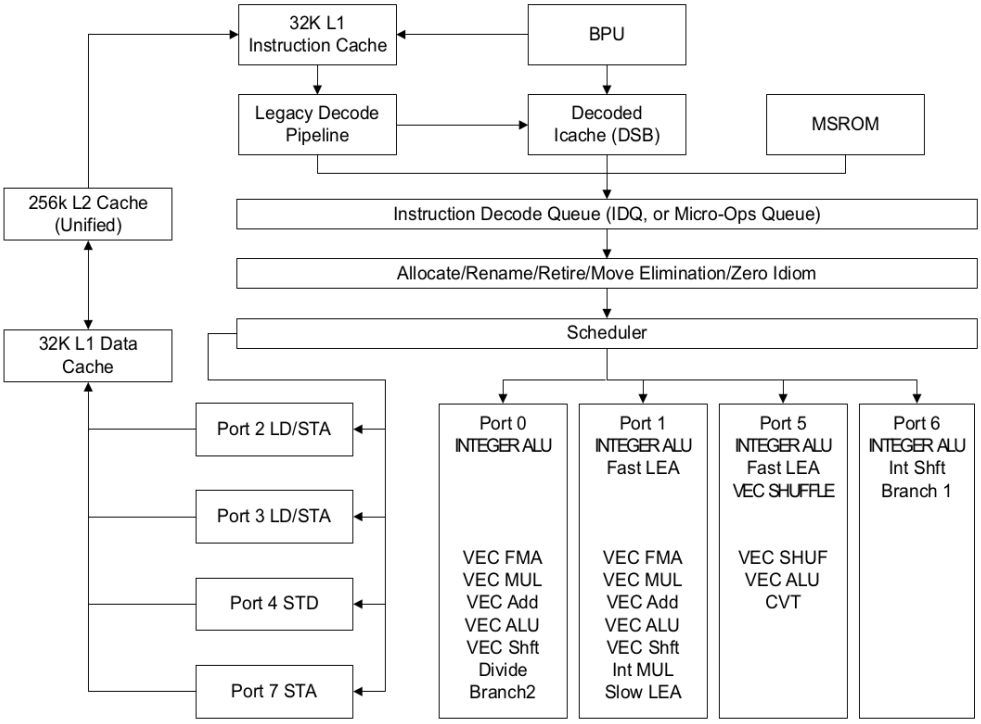

Figura 1: Organización del segmentado en un núcleo Skylake cliente.

De nuevo, `toplev` nos sugiere una ejecución para obtener métricas de tercer nivel relacionadas con el ancho de banda del fetch:

```
$ toplev.py -v --nodes '!+Fetch_Bandwidth*/3' --no-desc -- ss.1k.single.vec.avxfma.gcc
[...]
FE       Frontend_Bound.Fetch_Bandwidth       % Slots              35.2    [50.0%]
FE       Frontend_Bound.Fetch_Bandwidth.MITE  % Slots_est           0.0  < [50.0%]
FE       Frontend_Bound.Fetch_Bandwidth.DSB   % Slots_est          47.5    [50.0%]
FE       Frontend_Bound.Fetch_Bandwidth.LSD   % Slots_est           0.0  < [50.0%]
```

Según estos resultados, el DSB no está suministrando un número suficiente de uops a la IDQ.

```
$ toplev --describe Fetch_Bandwidth.DSB^
[...]
Frontend_Bound.Fetch_Bandwidth.DSB
    This metric represents Core fraction of cycles in which CPU
    was likely limited due to DSB (decoded uop cache) fetch
    pipeline.  For example; inefficient utilization of the DSB
```

```
    cache structure or bank conflict when reading from it; are
    categorized here.
[...]
```

La versión avx no tiene este problema:

```
$ toplev.py -v --nodes '!+Fetch_Bandwidth*/3' --no-desc -- ss.1k.single.vec.avx.gcc
[...]
FE       Frontend_Bound.Fetch_Bandwidth       % Slots                3.8  < [50.0%]
FE       Frontend_Bound.Fetch_Bandwidth.MITE  % Slots_est            0.8  < [50.0%]
FE       Frontend_Bound.Fetch_Bandwidth.DSB   % Slots_est            2.2  < [50.0%]
FE       Frontend_Bound.Fetch_Bandwidth.LSD   % Slots_est            0.0  < [50.0%]
[...]
```

Vamos a concretar el problema con el DSB. Primero vamos a obtener dos métricas relacionadas con esta estructura:

```
$ toplev --describe DSB
[...]
DSB_Coverage
    Fraction of Uops delivered by the DSB (aka Decoded ICache;
    or Uop Cache). See section 'Decoded ICache' in Optimization
    Manual. http://www.intel.com/content/www/us/en/architecture-
    and-technology/64-ia-32-architectures-optimization-
    manual.html
[...]
DSB_Misses
    Total pipeline cost of DSB (uop cache) misses - subset of
    the Instruction_Fetch_BW Bottleneck.
[...]

$ toplev.py --nodes '!+DSB_Coverage,DSB_Misses' -v -- ss.1k.single.vec.avxfma.gcc
[...]
Info.Frontend    DSB_Coverage    Metric                      1.00   [33.4%]
Info.Botlnk.L2   DSB_Misses      Scaled_Slots                0.02   [33.3%]
[...]
```

Los resultados muestran que:

- El DSB sirve la totalidad de las uops decodificadas (DSB_Coverage=1)
- Las peticiones a bloques almacenados en el DSB son servidas con muy pocos fallos (2%)

Así pues, parece que el problema está causado por una ineficiente utilización del DSB. Vamos a confirmarlo con `perf`, para ello vamos a consultar los contadores relacionados con la métrica Frontend_Bound.Fetch_Bandwidth.DSB:

```
$ perf stat -e cycles,idq.all_dsb_cycles_any_uops,idq.all_dsb_cycles_4_uops -- ss.1k.single.vec.avxfma.
[...]
     4.430.039.972      cycles
     4.286.052.577      idq.all_dsb_cycles_any_uops
        79.579.968      idq.all_dsb_cycles_4_uops
[...]
```

Observamos que prácticamente todos los ciclos el DSB no está suministrando suficientes uops a la IDQ. Si lo comparamos con la versión avx:

```
$ perf stat -e cycles,idq.all_dsb_cycles_any_uops,idq.all_dsb_cycles_4_uops -- i5-9500/ss.1k.single.vec
[...]
     3.146.269.244      cycles
     2.461.132.491      idq.all_dsb_cycles_any_uops
     2.252.366.626      idq.all_dsb_cycles_4_uops
[...]
```

En este código podemos observar que en torno a un 25% del tiempo no se suministran uops a la IDQ.

Vamos a estudiar la disposición de ambos códigos en memoria:

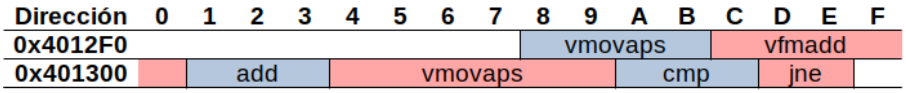

Figura 2: Disposición del código avx+fma en memoria.

El primer nivel de cache de instrucciones (L1I) tiene un tamaño de 32 KiB y está organizada en 64 conjuntos de 8 vías, con bloques de 64 bytes. Por lo tanto, el bucle estará distribuido en dos bloques distintos, los primeros 15 bytes en el conjunto 11 y el resto en el conjunto 12.

Tabla 3: División de direcciones en marca, conjunto, y byte dentro del bloque.

Dirección	Marca	Conjunto	byte/bloque
`0x4012F0`	0100 0000 0001	0010 11	11 0000
`0x401300`	0100 0000 0001	0011 00	00 0000

Según los contadores hardware recogidos, esta disposición del código en L1I no parece afectar al rendimiento. En cuanto al código avx, el bucle está en un único bloque de cache de instrucciones, el 11.

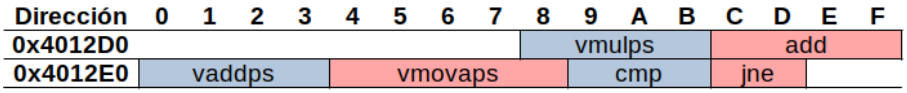

Figura 3: Disposición del código avx en memoria.

Tabla 4: División de direcciones en marca, conjunto, y byte dentro del bloque.

Dirección	Marca	Conjunto	byte/bloque
`0x4012D0`	0100 0000 0001	0010 11	01 0000
`0x4013E0`	0100 0000 0001	0010 11	10 0000

A continuación vamos a estudiar cómo se almacenan las instrucciones decodificadas en el DSB. Para ello, hemos de tener en cuenta la siguiente restricción [6]:

> Todos los uops de una vía deben estar en el mismo bloque de 32 bytes (*Uops in way must be in 32B aligned window*)

La instrucción `vfmadd132ps` cruza una frontera de 32 bytes, por lo que sus dos uops se almacenarán en una vía distinta a las ocupadas por las uops anteriores y posteriores.

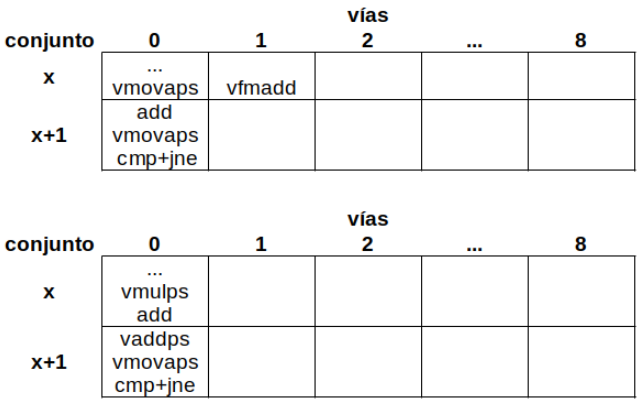

Figura 4: Posible disposición de las instrucciones decodificadas en el DSB.

Por lo tanto, el bucle avx se almacena en dos vías mientra que el bucle avx+fma lo hace en tres. Ahí puede estar la explicación de la pérdida del 50% de rendimiento. En este caso, dado que tenemos un bucle pequeño (23 bytes), podemos evitar que cruce una frontera de 32 bytes alineándolo a dicho tamaño. Para ello, recompilamos la versión añadiendo la opción `-falign-loops=32` [1]. Obtenemos el siguiente código:

```
401325:   66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%rax,1)
40132c:   00 00 00 00
401330:   66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%rax,1)
401337:   00 00 00 00
40133b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

[1] El valor por defecto de la opción `-falign-loops` es 0.

```
  401340:   c5 fc 28 c3              vmovaps %ymm3,%ymm0
  401344:   c4 e2 6d 98 00           vfmadd132ps (%rax),%ymm2,%ymm0
  401349:   48 83 c0 20              add    $0x20,%rax
  40134d:   c5 fc 29 40 e0           vmovaps %ymm0,-0x20(%rax)
  401352:   48 39 c3                 cmp    %rax,%rbx
  401355:   75 e9                    jne    401340 <scale_shift+0x70>
```

Puede observarse que al comienzo del bucle se han añadido nops que ocupan 28 bytes. Si medimos el rendimiento de esta nueva versión:

Versión	tiempo(ns)	R(GFLOPS)
AVX2+FMA	43.5	47.1

Este resultado es similar al obtenido por la versión avx.

Los contadores hardware confirman que ahora el DSB sirve 4 uops por ciclo a la IDQ:

```
$ perf stat -e cycles,idq.all_dsb_cycles_any_uops,idq.all_dsb_cycles_4_uops -- ss.1k.single.vec.avxfma.
[...]
     2.989.151.487      cycles
     2.396.426.322      idq.all_dsb_cycles_any_uops
     2.332.210.100      idq.all_dsb_cycles_4_uops
[...]
```

6 Otros entornos

6.1 Compilador ICX

El compilador icx 2023.2 obtiene mejores prestaciones gracias a un desenrollado de grado 4:

Tabla 6: Resultados de ejecución.

comp.	versión	tiempo(ns)	R(GFLOPS)
GCC	AVX2	45.3	45.2
	AVX2+FMA	64.5	31.8
ICX	AVX2	35.0	58.5
	AVX2+FMA	30.2	67.8

6.2 Procesador i5-1240P

Hemos ejecutado las versiones avx y avx+fma con y sin la opción de alineamiento de bucles a 32 bytes en un núcleo de un sistema con un procesador intel i5-1240P (duodécima generación, Alder Lake), modelo 154 (0x9A), stepping 3. Los resultados se muestran en la siguiente tabla.

Tabla 7: Resultados de ejecución en un sistema con un procesador intel i5-1240P.

comp.	versión	tiempo(ns)	R(GFLOPS)
GCC	AVX2	38.8	52.8
	AVX2 alin.	39.5	51.8
	AVX2+FMA	34.9	58.7
	AVX2+FMA al.	34.1	60.1
ICX	AVX2	31.7	64.6
	AVX2+FMA	20.4	100.8

En este procesador la falta de alineamiento del bucle no penaliza el rendimiento de la versión avx+fma.

La versión avx+fma compilada con `icx` supera los 100 GFLOPS.

7 Referencias

[1] Intel® Core™ i5-9500 Processor. https://www.intel.com/content/www/us/en/products/sku/134895/intel-core-i59500-processor-9m-cache-up-to-4-40-ghz/specifications.html

[2] Coffee Lake - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake

[3] uiCA - The uops.info Code Analyzer. https://uica.uops.info/

[4] pmu-tools. https://github.com/andikleen/pmu-tools

[5] A. Yasin, "A Top-Down method for performance analysis and counters architecture". 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, 2014, pp. 35-44, doi: 10.1109/ISPASS.2014.6844459. https://ieeexplore.ieee.org/document/6844459

[6] Causes of Performance Swings Due to Code Placement in IA. https://llvm.org/devmtg/2016-11/Slides/Ansari-Code-Alignment.pdf. https://www.youtube.com/watch?v=IX16gcX4vDQ

8 Bibliografía

- Descripción de eventos de Intel: https://perfmon-events.intel.com/ https://perfmon-events.intel.com/
- How to monitor the full range of CPU performance events. https://bnikolic.co.uk/blog/hpc-prof-events.html
- The mystery of an unstable performance. http://pzemtsov.github.io/2014/05/12/mystery-of-unstable-performance.html
- 32-byte aligned routine does not fit the uops cache. https://stackoverflow.com/questions/61016077/32-byte-aligned-routine-does-not-fit-the-uops-cache
- pmu-tools part I. http://halobates.de/blog/p/245
- From Top-down Microarchitecture Analysis to Structured Performance Optimizations. https://cassyni.com/events/YKbqoE4axHCgvQ9vuQq7Cy
- Top-Down performance analysis methodology. https://easyperf.net/blog/2019/02/09/Top-Down-performance-analysis-methodology
- Code alignment issues. https://easyperf.net/blog/2018/01/18/Code_alignment_issues