Grado en INGENIERÍA INFORMÁTICA

ARQUITECTURA DE COMPUTADORES

102775

Departamento de Arquitectura de Computadores y
Sistemas Operativos (DACSO)

PRÁCTICA nº 1

Medida y Optimización del Rendimiento del Procesador en programas *single-thread*

Enunciado y Documentación

Medir y Optimizar el Rendimiento del Procesador

ENUNCIADO: La práctica analiza algunos de los factores que determinan el tiempo de ejecución de un programa: el algoritmo, los datos de entrada y su tamaño, el compilador, la frecuencia de reloj del procesador, y la micro-arquitectura interna del procesador (latencias y capacidades de ejecución). Se trata de determinar numéricamente el tiempo de ejecución y explicarlo a través de medidas de contadores H/W, como el número de instrucciones ejecutadas o el número de ciclos consumidos.

También se realizarán las primeras optimizaciones sencillas en el código y se introducirá la sintaxis o notación de Cilk+ para operaciones vectoriales.

OBJETIVOS:

- Analizar el código ensamblador para estimar el recuento de instrucciones totales y el número de operaciones agrupadas por tipo, y para evaluar las optimizaciones realizadas por el compilador al trasladar el código fuente a código ensamblador.
- Entender el efecto que tienen ciertos parámetros del compilador en la generación de código más eficiente.
- Aprender a usar los contadores H/W de rendimiento mediante el comando `perf`. Determinar empíricamente el IPC de la ejecución.
- Utilizar la herramienta **VTune** para analizar el rendimiento de una aplicación y visualizar el código ensamblador.
- Cuantificar teóricamente y empíricamente el efecto que tienen los datos de entrada al programa (y el tamaño del problema) en el número total de instrucciones ejecutadas (complejidad del programa).
- Entender y aplicar la técnica de optimización de desenrollado de bucles.
- Identificar el uso de instrucciones SIMD o vectoriales, y evaluar el impacto en el rendimiento de su uso.
- Reconocer la sintaxis o notación vectorial de Cilk+

MATERIAL y PREPARACIÓN PREVIA

Procesador Intel i7 950 (Laboratorio)
Quad core i7, CPU clock: 3.33 GHz; Arquitectura x86-64; Compiladores: gcc versión 4.4.7, 4.9.0; icc versión 16.0.0
Las latencias y capacidades de ejecución del procesador se muestran en la siguiente tabla:

TIPO:	CORE	INT	BRN	LOAD	STORE	FAD	FMUL
Latencia:		1 ciclo	1 ciclo	4 ciclos	3 ciclos	3 ciclos	5 ciclos
Throughput:	4 µop / ciclo	3 µop / ciclo	1 µop / ciclo	1 µop / ciclo	1 µop / ciclo	1 µop / ciclo	1 µop / ciclo

Antes de asistir a la sesión de laboratorio, se debe leer con detenimiento todo este documento, y se debe intentar contestar a las preguntas previas a partir de los datos presentados en este documento. La parte de trabajo previo debería ser discutida con el profesor al inicio de la sesión para asegurar que habéis entendido los objetivos y los conceptos importantes. Hay que llevar preparada una planificación del trabajo a realizar durante la sesión (si se improvisa sobre la marcha, suele faltar tiempo al final). Las dudas sobre lo que hay que hacer hay que resolverlas lo antes posible con el profesor de prácticas.

Durante la sesión es importante asegurar que no se están ejecutando procesos que introduzcan "ruido" en la toma de medidas de tiempo. Se puede utilizar el comando **`top`**, que visualiza el uso de la CPU y los procesos en ejecución. Si el porcentaje de uso de CPU es superior al 1 %, se debe avisar al profesor. Presionar la tecla "q" para finalizar el comando **`top`**

```
prompt$ top
top - 18:10:47 up 4 days, 22:33,  1 user,  load average: 0.34, 0.10, 0.03
Tasks:  56 total,   2 running,  54 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni, 100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
```

Los informes de prácticas se deben entregar por correo electrónico (o en el Campus Virtual) según os indique vuestro profesor de prácticas hasta 48 horas después de finalizada la sesión. Cada día adicional a la fecha límite de entrega penalizará un 10% la nota de la práctica (es decir, que entregar más de 10 días tarde supone automáticamente una evaluación de 0 puntos). El documento electrónico enviado debe indicar el nombre completo de los alumnos, el turno de prácticas al que pertenece el grupo, el nombre del profesor de prácticas, y la fecha de entrega. En las sesiones posteriores, el profesor de prácticas hará preguntas a los miembros de cada grupo de prácticas para asegurarse de que todos ellos han alcanzado los objetivos didácticos esperados. La memoria del trabajo de prácticas solamente supone el 30% de la nota total de prácticas: el resto se evalúa a partir del trabajo previo mostrado al profesor y del trabajo realizado durante la sesión.

CONCEPTOS BÁSICOS

Compiladores

En los computadores de sobremesa del laboratorio está instalado el sistema operativo Linux, y el compilador gnu gcc, en dos de sus versiones: 4.4.7 y 4.9.0. Por defecto se usa la versión más antigua (4.4.7); se puede usar el comando **`gcc –v`** para verificar la versión del compilador. Para instalar la nueva versión del compilador se debe usar el comando **`module rm gcc/4.4.7`** para desinstalar la versión actual, y el comando **`module add gcc/4.9.0`** para instalar la versión nueva. La opción de compilación más importante es, probablemente, la que controla el nivel de optimización que realiza el compilador: **`-On`**, donde n es un número del 0 al 3, y cuanto mayor es el número mayor es la cantidad de tipos de optimización que utiliza el compilador. La opción **`gcc --help=optimizers`** nos muestra por separado cada una de las opciones de optimización. La opción –g incluye en el archivo ejecutable información de depuración (*debug*) que permite a las herramientas de perfilado (*profiling*) y depuración, como VTune, mostrar información más detallada.

También se dispone del compilador icc de Intel, que necesita acceso a un servidor de licencias (en caso de problemas al compilar puede ser debido a no tener acceso al servidor de licencias). Para usar el compilador debe ejecutarse el comando **`module add intel/16.0.0`**. Los dos compiladores, gcc e icc, pueden estar instalados y se pueden usar a la vez.

El compilador de Intel tiene soporte nativo para extensiones del lenguaje Cilk+, mientras que el compilador gcc, versión 4.9, requiere el uso del flag **`–fcilkplus`**.

Medida de Rendimiento en el Procesador

Para realizar medidas de rendimiento hay dos técnicas básicas: (1) instrumentar el programa fuente con instrucciones o funciones que midan ciertas métricas, o (2) usar herramientas que interrumpen la ejecución del programa y toman muestras de estas métricas, para luego correlacionarlas con el programa. Ejemplos de esta segunda técnica son los comandos **`time`** y **`perf`** de Linux, y la herramienta gráfica VTune (intel).

Contadores H/W de Rendimiento y PERF

Los procesadores del laboratorio (y todos los procesadores actuales) disponen de contadores H/W internos que se van incrementando durante la ejecución de los programas, y que permiten medir diferentes métricas de rendimiento (ciclos de reloj consumidos en el transcurso de la ejecución, instrucciones ejecutadas de diferente tipo, fallos en las diferentes memorias caché del computador…). Estos contadores se pueden salvar a memoria y recuperarlos de memoria cada vez que hay un cambio de contexto y el proceso/thread en ejecución es desalojado. De esta forma, se puede disponer de información bastante precisa (pero con un cierto margen de error) para cada proceso/thread distinto que está en ejecución, incluso cuando múltiples procesos/threads comparten el procesador.

El acceso a los contadores H/W de rendimiento requiere el uso de instrucciones privilegiadas de bajo nivel que sólo son accesibles por procesos de sistema. En esta práctica se utilizará el comando **`perf`** integrado en el paquete *linux-tools*. Se trata de un método de acceder a los contadores H/W de rendimiento, que presenta un interfaz muy simple y sencillo de usar. Proporciona tanto datos de medida de rendimiento como datos de traza de la ejecución (similar a *gprof*). El uso más habitual de **`perf`** es la medida de eventos, que pueden ser tanto eventos software (cambios de contexto, fallos de página, *cpu migrations* …) como eventos hardware que miden eventos ocurridos en la micro-arquitectura del procesador (ciclos de reloj, instrucciones ejecutadas, fallos de caché …). Ver manual de uso en: https://perf.wiki.kernel.org/index.php/Tutorial.

A continuación se muestra parte de la salida de un ejemplo de ejecución. Los valores a la izquierda son medidas reales mientras que los valores a la derecha (después del carácter #) son valores calculados a partir de las medidas de la izquierda. Por ejemplo, se muestra el valor del IPC (Instrucciones ejecutadas Por Ciclo de reloj) calculado al dividir el número de instrucciones ejecutadas entre el número de ciclos de reloj que ha durado la ejecución.

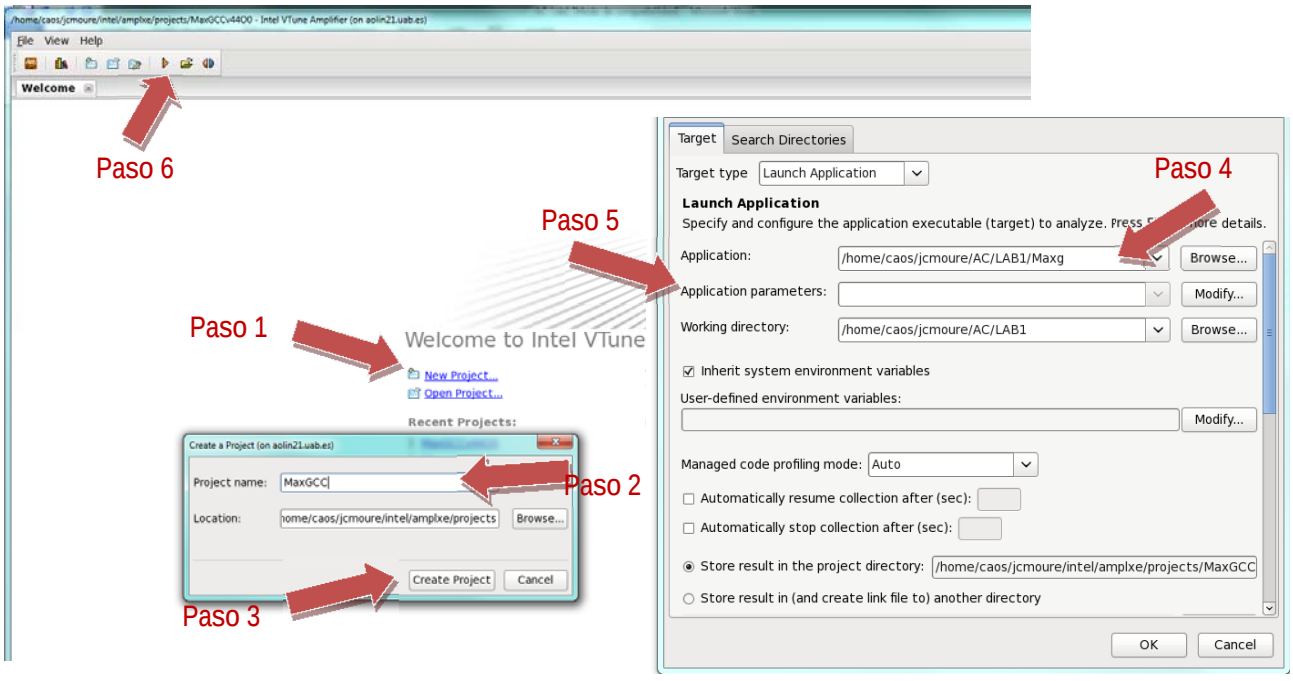
```
ac-tutor@aolin21:~/$ perf stat ./addvec

 Performance counter stats for './addvec':

     898.264674  task-clock-msecs         #      0.997 CPUs
             75  context-switches         #      0.000 M/sec
              0  CPU-migrations           #      0.000 M/sec
            104  page-faults              #      0.000 M/sec
     3003386269  cycles                   #      3,352 GHz
     4006399302  instructions             #      1.334 IPC
               …

    0.901134212  seconds time elapsed
```

Es posible especificar de forma explícita los eventos que se quieren medir con la opción *–e*, y es posible diferenciar entre eventos ocurridos en modo usuario (*:u*) y eventos en modo kernel (*:k*). La opción *–r K* ejecuta la aplicación *K* veces y luego hace la media de todas las medidas de eventos. La opción *list* nos muestra todos los eventos disponibles en el sistema:

```
ac-tutor@aolin21:~/$ perf list

List of pre-defined events (to be used in -e):

  cpu-cycles OR cycles                       [Hardware event]
  instructions                               [Hardware event]
  cache-references                           [Hardware event]
  cache-misses                               [Hardware event]
  branch-instructions OR branches            [Hardware event]
  branch-misses                              [Hardware event]
  bus-cycles                                 [Hardware event]

  cpu-clock                                  [Software event]
  task-clock                                 [Software event]
  page-faults OR faults                      [Software event]
  context-switches OR cs                     [Software event]
  cpu-migrations OR migrations               [Software event]

  L1-dcache-loads                            [Hardware cache event]
  L1-dcache-load-misses                      [Hardware cache event]
  L1-dcache-stores                           [Hardware cache event]
  L1-dcache-store-misses                     [Hardware cache event]
  LLC-loads                                  [Hardware cache event]
  LLC-load-misses                            [Hardware cache event]
  LLC-stores                                 [Hardware cache event]
  LLC-store-misses                           [Hardware cache event]
  LLC-prefetches                             [Hardware cache event]
  LLC-prefetch-misses                        [Hardware cache event]
  dTLB-loads                                 [Hardware cache event]
  dTLB-load-misses                           [Hardware cache event]
  dTLB-stores                                [Hardware cache event]
  dTLB-store-misses                          [Hardware cache event]
  branch-loads                               [Hardware cache event]
  branch-load-misses                         [Hardware cache event]
```

Herramienta de Medida de Rendimiento VTune

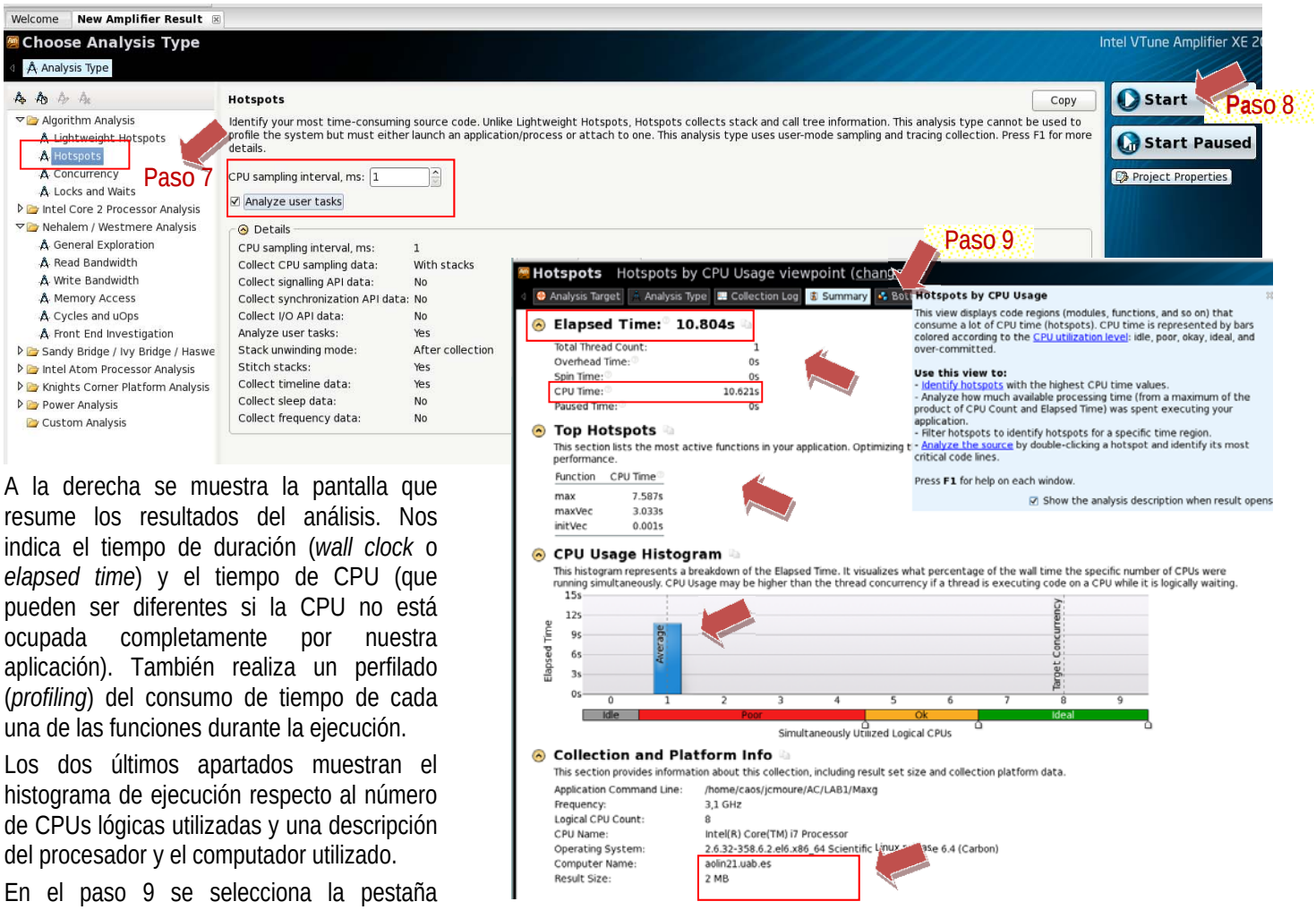
En el computador del laboratorio se dispone del S/W de Intel llamado VTune (se necesita licencia). Para usar VTune se debe haber instalado el entorno del compilador icc (**`module add intel/16.0.0`**). La herramienta gráfica se ejecuta con el comando "**`amplxe-gui &`**": el símbolo '&' sirve para invocar la ejecución del fichero ejecutable y después devolver el control al intérprete de comandos de linux. Es necesario crear un proyecto, utilizando los pasos que se muestran a continuación:



Una vez realizados los pasos anteriores, y seleccionada la opción indicada como paso 6, aparecerá la pantalla "*Choose Analysis Type*", que permite definir el tipo de análisis de rendimiento que se desea realizar. Por defecto se analizan los

hotspots del algoritmo (observar a la izquierda la opción *Algorithm Analysis->Hotspots*), pero se pueden seleccionar otros tipos de análisis. En el ejemplo se ha seleccionado el intervalo de muestreo (*sampling*) a 1 milisegundo, y se ha seleccionado la opción "*analyze user tasks*". Versiones nuevas de VTune pueden incluir nuevas alternativas de análisis. El siguiente paso (8) es clicar en la opción *Start*, y comenzará la ejecución del programa, que irá coleccionando datos de rendimiento.

La ejecución instrumentada se puede demorar algo más que la ejecución real; no se debería realizar a ciegas, sin haber probado antes la ejecución directamente en la línea de comandos. Recordar que se pueden definir los parámetros de la ejecución y las variables de estado del sistema.

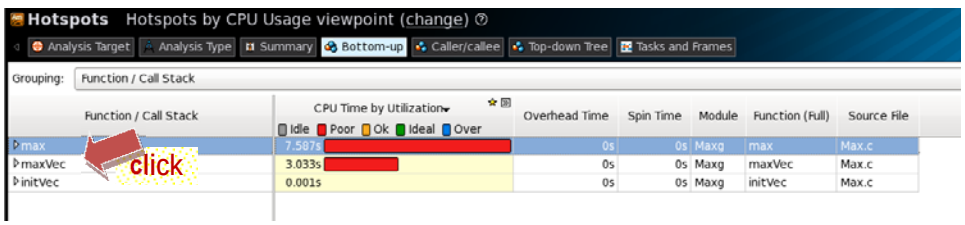


A la derecha se muestra la pantalla que resume los resultados del análisis. Nos indica el tiempo de duración (*wall clock* o *elapsed time*) y el tiempo de CPU (que pueden ser diferentes si la CPU no está ocupada completamente por nuestra aplicación). También realiza un perfilado (*profiling*) del consumo de tiempo de cada una de las funciones durante la ejecución.

Los dos últimos apartados muestran el histograma de ejecución respecto al número de CPUs lógicas utilizadas y una descripción del procesador y el computador utilizado.

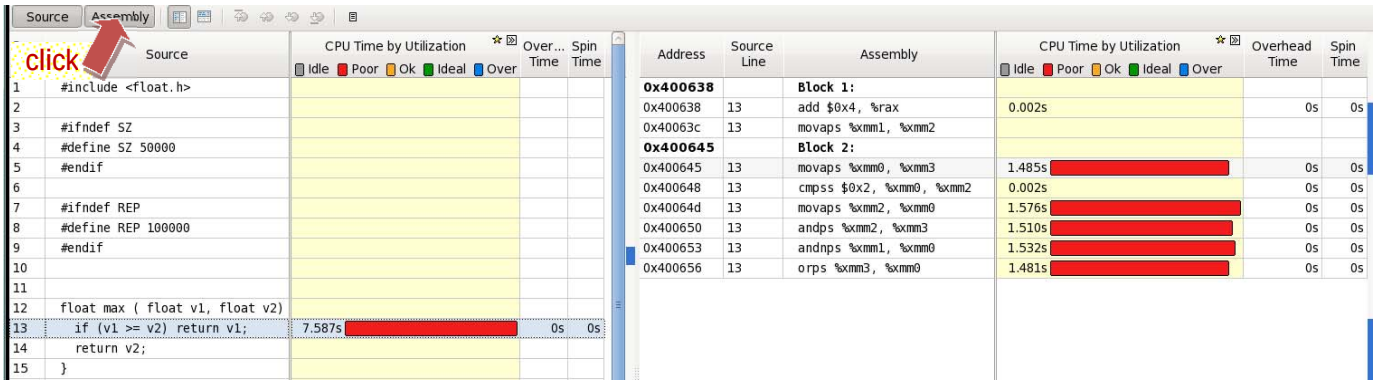
En el paso 9 se selecciona la pestaña *bottom-up* que hace aparecer la información de perfilado clasificada por funciones.

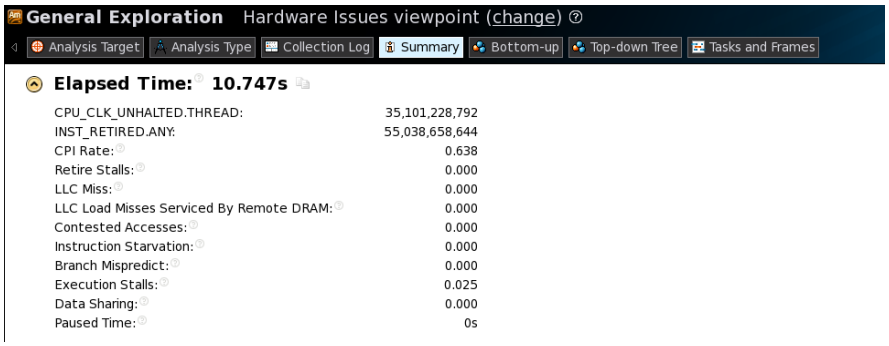
Si se clica sobre la línea correspondiente a una función, aparecerá el código fuente de la función, con la repartición de tiempo de ejecución asignada a cada línea de la función (es necesario haber compilado el programa con la opción –g para incluir información adicional en el archivo ejecutable). Si se selecciona la opción *Assembly*, aparece el código ensamblador asociado a la función, indicando también el tiempo estimado asociado a cada instrucción. Todos los tiempos medidos son **estimaciones obtenidas** a partir de muestras de la ejecución tomadas a un cierto ritmo (en el caso de análisis de *hotspots* las muestras se toman, por defecto, cada milisegundo). Por tanto, no deben tomarse como valores exactos, sino como estimaciones aproximadas, que en la práctica pueden diferir bastante del valor real.

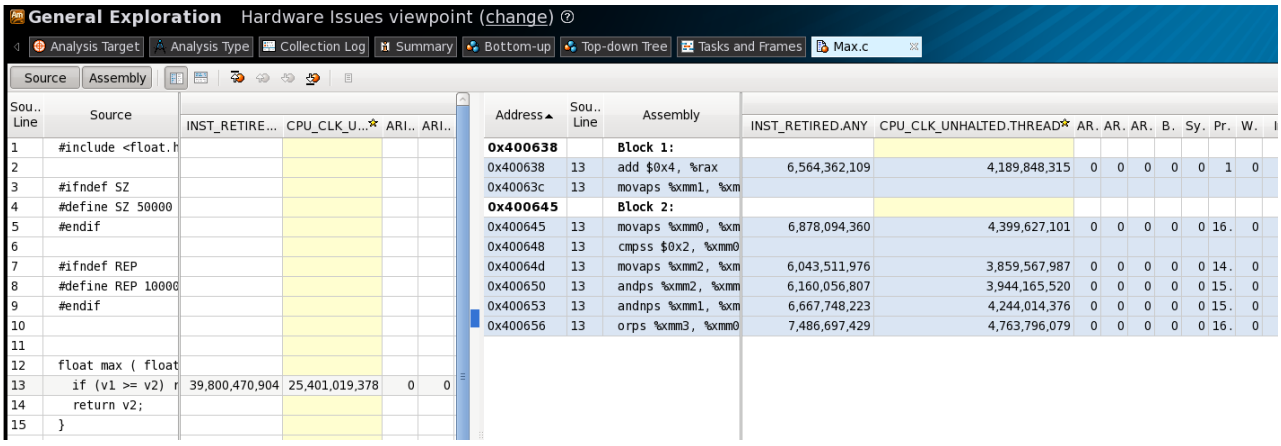


A continuación se muestra un ejemplo de análisis realizado con VTune para medir contadores de rendimiento H/W. Hay que seleccionar el tipo de arquitectura del procesador que queremos analizar (por defecto VTune determina que se debe

seleccionar la arquitectura Nehalem). Los eventos de medida de rendimiento también se pueden seleccionar de forma individual. Las medidas de rendimiento de los contadores se asocian a cada instrucción, pero, de nuevo, hay que tener en cuenta que son medidas estimadas de forma estadística: de otro modo no se entendería que no sea idéntico el número de veces que se ejecutan instrucciones que están juntas dentro del mismo bloque básico







TRABAJO PREVIO

Código: Encontrar el elemento en un vector de valor Máximo

El siguiente código (fichero **`Max.c`**) calcula el máximo de los SZ elementos de un vector de tipo float. El cálculo se realiza REP veces seguidas, para que la toma de medidas de rendimiento sea más precisa. **`max()`** calcula el valor máximo de 2 números. **`maxVec()`** utiliza la función **`max()`** para encontrar el valor máximo del vector. Por último, el programa principal inicializa el vector **`Vect`** con SZ valores generados de forma aleatoria, y luego realiza REP llamadas consecutivas a la función **`maxVec()`**. Tanto SZ como REP son constantes que se pueden definir en tiempo de compilación.

Nota: observar que la variable F se declara utilizando el modificador **`volatile`**, que le indica al compilador que el contenido de la variable puede cambiar de forma asíncrona y que no puede hacer suposiciones sobre su valor. Este modificador se suele usar para variables que pueden ser modificadas en un momento inesperado por parte de una rutina de tratamiento de interrupciones. En este caso se utiliza para evitar que el compilador "optimice" demasiado: si se da cuenta de que el bucle **`for`** que se repite REP veces no sirve para nada, pues puede "eliminar" el bucle y llamar a la función **`maxVec`** solamente una vez.

Max.c:

```
float inline max ( const float v1, const float v2 )
{
  if (v1 >= v2 )
    return v1;
  return v2;
}
float inline maxVec ( const float V[], const int N)
{
  int i;
  float Vmax= FLT_MIN; // minimum float value
  for (i=0; i<N; i++)
    Vmax = max (Vmax, V[i]);
  return Vmax;
}
void initVec (double V[], const int N, const int seed)
{
  int i;
  srand48(seed);  // set initial seed for random number generation
  for (i=0; i<N; i++)
    V[i]= drand48();
}

float Vect[SZ];
float volatile F;  // forces compiler to implement REP loop

void main() {
  int i;
  initVec (Vect, SZ, 0);
  F= 0.0;
  for (i=0; i<REP; i++)
    F = maxVec (Vect, SZ);
  printf("Result: %f\n", F);
}
```

Se compila (versión 4.4.7 del compilador) y se ejecuta el código con los siguientes comandos (la opción –D<variable>=<value> se usa para definir en tiempo de compilación los valores de SZ y de REP, que son constantes dentro del programa):

```
prompt$ gcc –g –DSZ=50000 –DREP=100000 Max.c –o Max
```

Midiendo el Rendimiento y el efecto del Compilador

Se ejecuta el programa con el comando **perf** para obtener el número de instrucciones ejecutadas (ICount) y el nº de ciclos de reloj requeridos para la ejecución (CCount)

```
prompt$ perf stat ./Max
```

```
Result: 0.999987
 Performance counter stats for './Max':

      30311,257800 task-clock                #    0,999 CPUs utilized
               31 context-switches          #    0,001 K/sec
                6 CPU-migrations            #    0,000 K/sec
              148 page-faults               #    0,005 K/sec
    98.462.355.682 cycles                   #    3,378 GHz
    58.320.401.958 stalled-cycles-frontend  #   59,23% frontend cycles idle
     4.027.723.490 stalled-cycles-backend   #    4,09% backend  cycles idle
   135.115.089.787 instructions             #    1,37  insns per cycle
                                            #    0,43  stalled cycles per insn
    25.020.223.000 branches                 #  825,443 M/sec
           638.504 branch-misses            #    0,00% of all branches

      30,382773759 seconds time elapsed
```

Analizando el código fuente del programa, deducimos que la mayor parte del tiempo se dedicará a ejecutar las instrucciones de la función max(), dentro de la función maxVec(), mientras que una pequeña fracción restante del tiempo se utilizará para ejecutar la función initVec(). Ignoraremos la ejecución de initVec() y supondremos que solamente se ejecuta la función max() (si SZ= 50.000 y REP= 100.000, ¿cuántas veces se ejecuta la función max()?). Se quiere medir el efecto de utilizar diferentes opciones de optimización y de variar el compilador. En la tabla siguiente se muestran los resultados de rendimiento para diferentes casos de compilación, donde se indican las opciones añadidas y la versión del compilador. Los valores medidos son una media de las 3 ejecuciones del programa completo con mejor rendimiento, de entre 5.

Compilador	Elapsed Time	Instructions	Cycles
gcc (v4.4)	30,38 seg.	135,12 G	98,46 G
gcc –O3 (v4.4)	10,41 seg.	55,04 G	35,08 G
gcc –Ofast (v4.9)	1,113 seg.	5,01 G	3,75 G
icc –O3 (v16.0)	1,113 seg.	2,19 G	3,75 G

Tabla 1. Medidas de Rendimiento promediadas para el programa Max.c con varios compiladores y opciones de compilación. SZ=50.000, REP= 100.000. (G= 10^9)

Pregunta 1a: Rellenar los valores en la tabla siguiente usando los datos anteriores. Calcular: (1) el número total de operaciones max() que realiza el programa, (2) el promedio de operaciones max que ejecuta el programa por cada mil instrucciones máquina ejecutadas, (3) el valor del IPC, (4) el número promedio de ciclos de reloj consumidos por cada operación max ejecutada, y (5) el speedup obtenido respecto al tiempo de ejecución del primero de los casos.

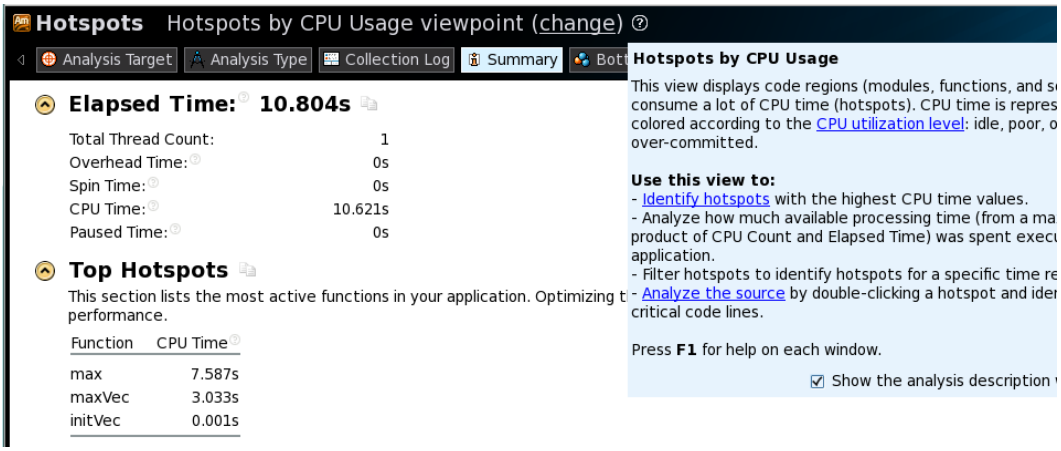
Compilador	OpCount (max)	OpPerInstr	IPC	CyclesPerOper	Speedup
gcc (v4.4)					1,0
gcc –O3 (v4.4)					
gcc –Ofast (v4.9)					
icc –O3 (v16.0)					

Pregunta 1b: Usando los datos de la tabla anterior, explicar qué es lo que consigue cada versión del compilador para mejorar el rendimiento.

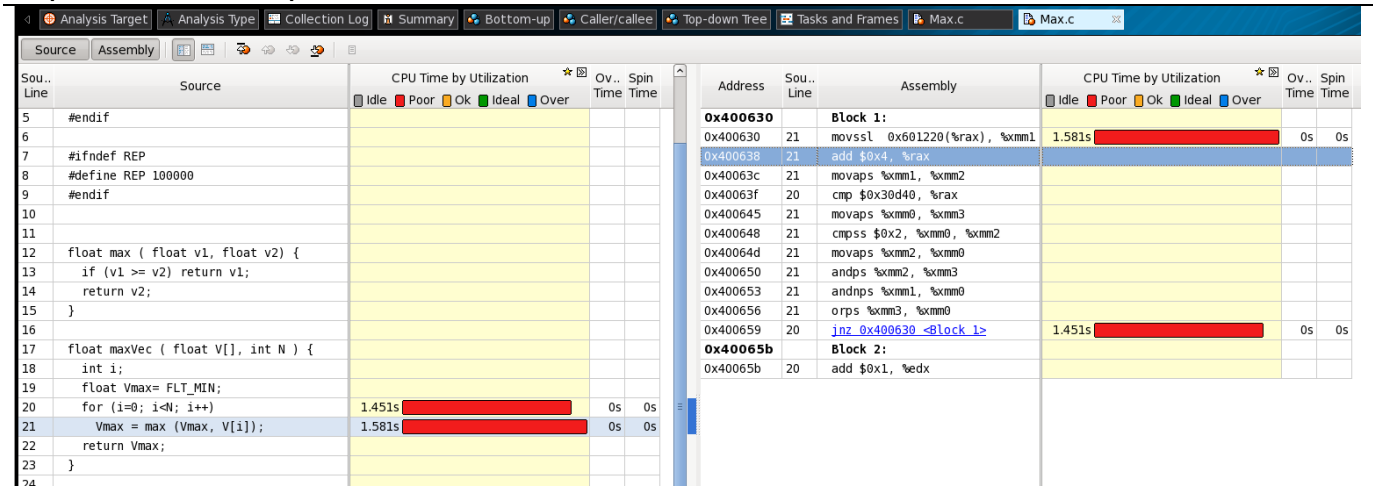
Analizando la ejecución y el código ensamblador

Se utiliza la herramienta VTune con el programa compilado con gcc v4.4 y –O3. Recordad que es necesario generar el fichero ejecutable con información de depuración usando la opción –g, tal como se muestra en la línea siguiente. La siguiente ventana muestra el resumen del análisis de la ejecución (*hotspots*). En ella se muestra que la ejecución de las funciones max() y maxVec() supone la práctica totalidad del tiempo de ejecución.

```
prompt$ gcc –O3 –g –DSZ=50000 –DREP=100000 Max.c –o MaxG    (gcc v4.4)
prompt$ amplxe-gui &
```



La siguiente ventana muestra el detalle de la función maxVec(): a la izquierda el código fuente y a la derecha el código ensamblador correspondiente. Observar que VTune, erróneamente, sólo le atribuye tiempo de ejecución a dos de las instrucciones ensamblador, cuando el resto de las 11 instrucciones del bucle se ejecutan el mismo número de veces que ellas, y por tanto se les debería computar algún tiempo. Esta forma de funcionar tiene que ver con el funcionamiento del muestreo que realiza VTune y del funcionamiento interno del procesador, que ejecuta las instrucciones de forma desordenada. Por tanto, no hemos de tomar estos datos de forma absoluta, y hemos de tener cuidado de no sacar conclusiones precipitadas sobre cómo se reparte el tiempo entre las instrucciones.



A continuación se recompila el código fuente utilizando la opción –S para generar el archivo con el código ensamblador. El siguiente listado muestra el fragmento correspondiente al bucle principal de la función maxVec(), al que se le han añadido comentarios para mostrar el tipo de las micro-operaciones asociadas a cada instrucción ensamblador.

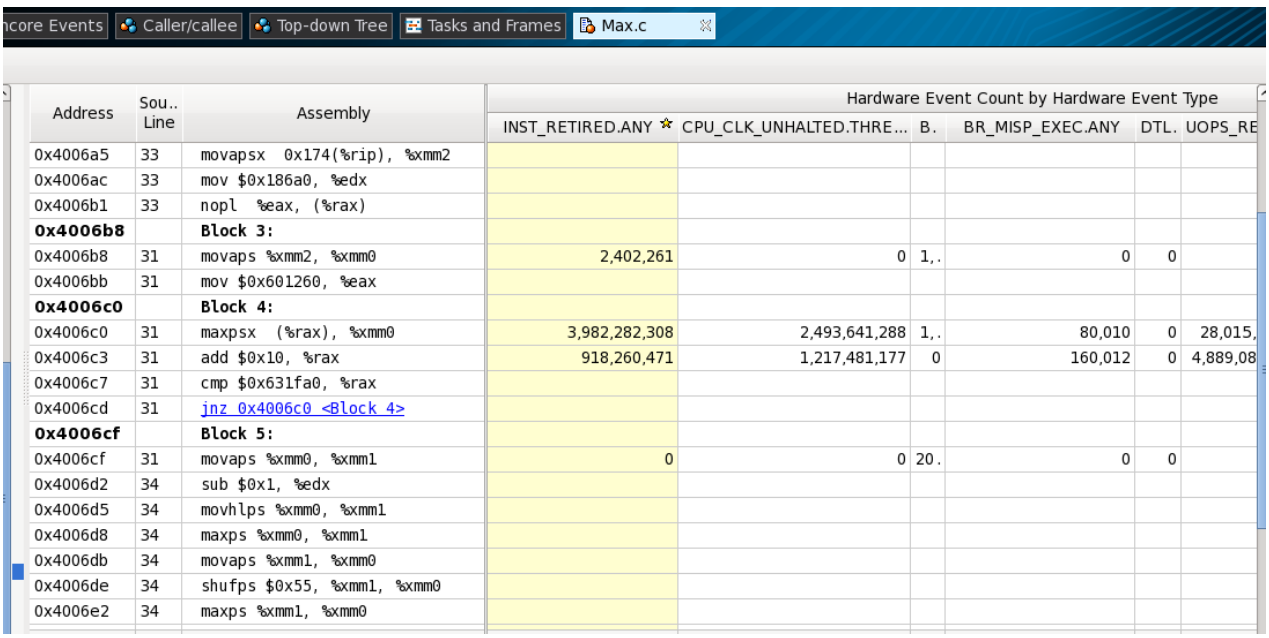
Bucle interno de la función **maxVec()**, compilado con gcc v4.4 opción -O3

```
.L24:  movss    Vect(%rax), %xmm1     // LOAD
       addq     $4,     %rax          // INT
       movaps   %xmm1, %xmm2          // INT
       cmpq     $200000, %rax         // INT
       movaps   %xmm0, %xmm3          // INT
       cmpless  %xmm0, %xmm2          // INT
       movaps   %xmm2, %xmm0          // INT
       andps    %xmm2, %xmm3          // INT
       andnps   %xmm1, %xmm0          // INT
       orps     %xmm3, %xmm0          // INT
       jne    .L24                    // BRN
…
```

Si se compara el código ensamblador que muestra VTune a partir del archivo ejecutable en la parte de la derecha de la ventana, con el que se obtiene al generar el archivo ensamblador (opción –S) se detectan algunas pequeñas diferencias. El valor simbólico **Vect** del archivo ensamblador se convierte en el valor exacto $0x601220 dentro del archivo ejecutable; el proceso de enlazado (*link*) resuelve la dirección simbólica en una dirección virtual constante, expresada en hexadecimal. Asimismo, la etiqueta del archivo ensamblador **.L24** se resuelve en el valor $400630, que representa la dirección de la instrucción a la que se debe saltar.

Se utiliza ahora la herramienta VTune con el programa compilado con gcc v4.9 y –Ofast. En esta ocasión se hace un análisis de rendimiento con contadores hardware.

prompt$ gcc –Ofast –g –DSZ=50000 –DREP=100000 Max.c –o MaxGv9

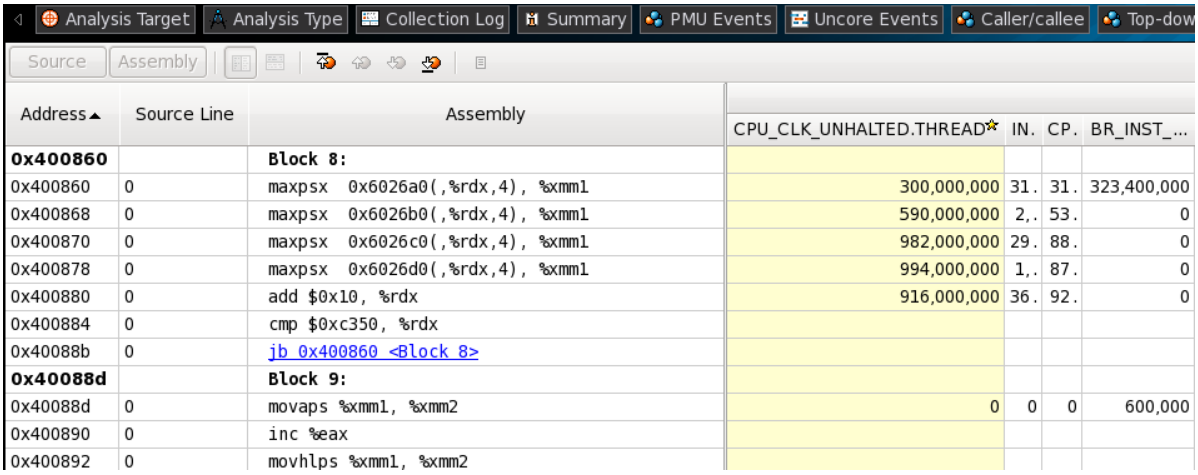


De nuevo, la información sobre instrucciones ejecutadas es un poco extraña: parece que la instrucción **`maxpsx (%rax), %xmm0`** se ejecute casi 4 mil millones de veces, la instrucción **`add $0x10, %rax`** alrededor de mil millones de veces, y en cambio las dos instrucciones siguientes no se ejecuten nunca. De nuevo, la asociación de los contadores de eventos que ocurren en la ejecución a las instrucciones particulares no es demasiado fiable (por ejemplo, a la instrucción **`maxpsx`** se le atribuyen cerca de 80 mil fallos de predicción de salto, y no es una instrucción de salto). Es preferible sumar los contadores asociados a las 4 instrucciones dentro del bucle interno, e interpretarlos de forma conveniente usando la lógica. A continuación se muestra el código ensamblador obtenido al compilar con la opción –S (observad que la primera instrucción máquina se descompone en dos micro-operaciones).

Bucle interno de la función **`main()`**, compilado con gcc v4.9 opción -Ofast

```
…
.L30:
        maxps   (%rax), %xmm0           // Load + FADD
        addq    $16, %rax               // INT
        cmpq    $Vect+200000, %rax      // INT
        jne     .L30                    // BRN
…
```

Finalmente se muestra un fragmento del código ensamblador generado por el compilador `icc`, visualizado usando la herramienta VTune, y extraído del fichero Max.s.



Address	Source Line	Assembly	CPU_CLK_UNHALTED.THREAD	IN.	CP.	BR_INST_...
0x400860		**Block 8:**				
0x400860	0	maxpsx 0x6026a0(,%rdx,4), %xmm1	300,000,000	31.	31.	323,400,000
0x400868	0	maxpsx 0x6026b0(,%rdx,4), %xmm1	590,000,000	2,.	53.	0
0x400870	0	maxpsx 0x6026c0(,%rdx,4), %xmm1	982,000,000	29.	88.	0
0x400878	0	maxpsx 0x6026d0(,%rdx,4), %xmm1	994,000,000	1,.	87.	0
0x400880	0	add $0x10, %rdx	916,000,000	36.	92.	0
0x400884	0	cmp $0xc350, %rdx				
0x40088b	0	jb 0x400860 <Block 8>				
0x40088d		**Block 9:**				
0x40088d	0	movaps %xmm1, %xmm2	0	0	0	600,000
0x400890	0	inc %eax				
0x400892	0	movhlps %xmm1, %xmm2				

Bucle interno de la función **`main()`**, compilado con icc v16.0 opción –O3

```
…
..B1.7:
   maxps       Vect(,%rdx,4), %xmm1      // Load + FADD
   maxps     16+Vect(,%rdx,4), %xmm1     // Load + FADD
   maxps     32+Vect(,%rdx,4), %xmm1     // Load + FADD
   maxps     48+Vect(,%rdx,4), %xmm1     // Load + FADD
   addq      $16,    %rdx                // INT
   cmpq      $50000, %rdx                // INT
   jb        ..B1.7                      // BRN
…
```

Pregunta 1c: Verificar que el recuento de instrucciones (ICount) que obtenemos con los contadores H/W mostrado en la Tabla 1 se corresponde con los tres casos de los que se muestra el código ensamblador. Usando los datos de latencias y capacidades de ejecución del procesador, que se muestran al principio de este documento, y usando el código ensamblador, calcular el tiempo de ejecución de cada uno de los tres casos tanto en el supuesto de ejecución completamente secuencial, como en el de ejecución completamente paralela. **Ayuda:** el tiempo de ejecución medido que se muestra en la Tabla 1 debe estar en medio de los dos tiempos.

Compilador	Tiempo de Ejec. Secuencial	Tiempo de Ejec. Paralela
`gcc –O3 (v4.4)`		
`gcc –Ofast (v4.9)`		
`icc –O3 (v16.0)`		

Pregunta 1d: Explicar la idea general de las optimizaciones que realizan los compiladores en el código. Escribir el código C equivalente de los dos últimos casos (gcc –O3 versión 4.9 e icc –O3 versión 16.0). Para hacer este código equivalente conviene revisar el ejemplo siguiente, que utiliza la nomenclatura de vectores de Cilk Plus.

Sesión 1: TRABAJO DURANTE la SESIÓN

Se optimiza el código anterior usando la nomenclatura vectorial de la extensión al lenguaje C denominada Cilk Plus (cilkplus.org), tal como se muestra a continuación (fichero **Max2.c**).

Max2.c Uso de notación vectorial para ampliar el paralelismo a nivel de instrucción (ILP) y vectorial (SIMD)

```
float max ( float v1, float v2 ) // uso del operador condicional  ()?():()
  { return (v1 >= v2)? v1: v2; }
float maxVec( float V[], int N) {
  int i;
  float Vmax1[4], Vmax2[4];
  Vmax1[0:4] = Vmax2[0:4] = FLT_MIN;  // duplicate value on all elements
  for (i=0; i<N; i+=8)                // assume N is always multiple of 8
  {
    Vmax1[:] = (Vmax1[0:4] > V[i:4]  )? Vmax1[:]: V[i:4];
    Vmax2[:] = (Vmax2[0:4] > V[i+4:4])? Vmax2[:]: V[i+4:4];
  }
  return __sec_reduce_max( max(Vmax1[:], Vmax2[:]));  // vector reduce
}
```

La notación de vectores del lenguaje Cilk Plus permite representar secciones dentro de un vector más grande. La sintaxis es: `array [ lower_bound : length ]`, donde el primer valor dentro de los corchetes indica el elemento inicial de la sección del array, y el segundo elemento indica el nº de elementos consecutivos contenidos en la sección del array, a partir del primer elemento. La notación `array [:]`, sin números dentro de los corchetes, se refiere al vector entero. La instrucción `Vmax2[0:4]=FLT_MIN` copia el valor escalar `FLT_MIN` en cada uno de los 4 elementos del vector `Vmax2[]`. Esta instrucción se podría haber escrito como `Vmax2[:]=FLT_MIN` ya que el vector `Vmax2[]` tiene 4 elementos.

La expresión`(Vmax1[0:4] > V[i:4])? Vmax1[:]: V[i:4]` utiliza el operador condicional "<c>?<v1>:<v2>" que se evalúa como el valor <v1> si <c>==**True**, o como el valor <v2> si <c>==**False**. Como se referencian vectores de 4 elementos, la operación se considera una operación SIMD (Single-Instruction Multiple-Data), en la que se realizan las 4 operaciones de forma simultánea, tal como se muestra a continuación:

```
Vmax1[0] = (Vmax1[0] > V[i])?   Vmax1[0]: V[i]
Vmax1[1] = (Vmax1[1] > V[i+1])? Vmax1[1]: V[i+1]
Vmax1[2] = (Vmax1[2] > V[i+2])? Vmax1[2]: V[i+2]
Vmax1[3] = (Vmax1[3] > V[i+3])? Vmax1[3]: V[i+3]
```

El compilador reconoce la sintaxis y tiene libertad para planificar la ejecución de las cuatro operaciones anteriores en cualquier orden (ya que el programador está diciendo de forma explícita que son operaciones independientes) o las puede planificar para ser ejecutadas a la vez (usando, si las hay, las instrucciones SIMD disponibles en el procesador). Por ejemplo, si el procesador destino de la compilación solamente tiene soporte para instrucciones SIMD de 2 en 2, entonces podría generar un código que realizara las dos primeras operaciones como una única instrucción SIMD y las dos últimas como otra instrucción SIMD.

El compilador substituye la operación `max(Vmax1[:], Vmax2[:])` por la operación `Vmax1[:]>Vmax2[:]? Vmax1[:]: Vmax2[:]`. A esta modificación que hace el compilador se le denomina *substitución en línea*: en lugar de generar una instrucción en ensamblador (*call*) para llamar a la función `max()`poniendo los parámetros `Vmax1[:]` y `Vmax2[:]` en registros o en la pila del procesador, el compilador substituye la llamada a la función `max()` por el código completo que define a la función `max()`, reemplazando los parámetros formales `v1` y `v2`, por los parámetros reales.

La función `__sec_reduce_max()`es una operación especial proporcionada por Cilk Plus que realiza una operación genérica de reducción utilizando la operación máximo. Es equivalente a aplicar la operación `max()` a parejas de elementos del vector hasta obtener un único valor. A continuación se muestran dos formas diferentes de calcular la reducción, de entre las muchas posibles ordenaciones:

```
__sec_reduce_max(V[0:4]) ==   max( max( max( V[0], V[1] ), V[2] ), V[3] ) ==
                              max( max( V[0], V[1] ), max( V[2], V[3] ) ) == …
```

Compilar y ejecutar el código del fichero Max2.c para valores de SZ= 50.000 y REP= 100.000. Probad con los dos compiladores, **gcc** 4.9 con opción –Ofast y –fcilkplus, e **icc** con nivel 3 de optimización. Usar el comando **perf** para tomar medidas de rendimiento, y la herramienta **VTune** para obtener el código ensamblador.

Pregunta 1e: Evaluar de forma cuantitativa el resultado de rendimiento comparando con las versiones previas e identificar el factor más importante para lograr que la optimización del programa Max2.c sea efectiva.

Código: Contar el número de bits con un uno de los elementos de un vector

El siguiente ejemplo muestra un programa que cuenta el número de bits a uno que tienen los elementos de un vector.

Count.c:

```
long int inline count ( long int v)
{  // count number of bits set in v
  int c=0;
  while (V) {          // while V is not zero
    if (V&1) c++;      // if less significant bit of V is not zero, increment c
    V = V >> 1;        // shift value in V one position to the right
  }
  return c;
}

long int inline countBits ( const long int V[], const int N)
{
  int i, cnt= 0;
  for (i=0; i<N; i++)
    cnt += count (V[i]);
  return cnt;
}

long int Vect[SZ];
long int volatile C;  // forces compiler to implement REP loop

void main()
{
  int i;
  initVec (Vect, SZ, 0);
  C= 0;
  for (i=0; i<REP; i++)
    C = countBits (Vect, SZ);
  printf ("Result: %ld\n", C);
}
```

A continuación se muestran dos versiones alternativas de la función count() anterior. Ambas usan unas funciones especiales que se denominan "*intrinsics*" y que son funciones que el compilador entiende muy bien y para las cuales es capaz de generar un código muy optimizado, muchas veces una única instrucción especial en ensamblador. Las funciones *intrinsic* para el compilador icc de intel se pueden encontrar en: https://software.intel.com/sites/landingpage/IntrinsicsGuide. Para encontrar las funciones *intrinsic* para el compilador gcc, un punto de partida puede ser: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. El compilador icc "entiende" muchas de las funciones *intrinsic* de gcc, pero en ocasiones puede generar un código de inferior rendimiento. En todos los casos, puede ser necesario usar algún flag del compilador (como –mpopcnt) para "activar" el uso de instrucciones especiales, que pueden existir únicamente en los modelos nuevos de procesadores.

Versión para usar con el compilador gcc. Para generar un código óptimo se debe usar el flag –mpopcnt.

```
long int inline count ( long int v )
{
  return __builtin_popcountll ( v );
}
```

Versión para usar con el compilador icc. Solamente requiere añadir el fichero *.h indicado en el código

```
#include "immintrin.h"
long int inline count ( long int v )
{
  return _mm_countbits_64 ( v );
}
```

Pregunta 1f: Evaluar de forma cuantitativa el uso de la función específica de tipo *intrinsic* en uno de los dos compiladores. Esto incluye comparar el rendimiento cuando se usa y el rendimiento cuando no se usa, y encontrar los factores que nos explican esa mejora de rendimiento. Probar con SZ=50.000 y REP=10.000. Averiguar el nombre de la instrucción ensamblador que cuenta bits.

Sesión 2: Optimización de Código

Transformaciones de tipo Map y Reduce

El código de la siguiente función realiza dos tipos de transformaciones algorítmicas sobre el contenido de tres vectores de entrada de N elementos (X[N], V1[N] y V2[N]) para generar un nuevo vector de salida, también de N elementos (R[N]).

La **transformación de tipo Map** genera una matriz de N^2 valores a partir de una especie de producto cartesiano del vector X[], de N elementos, y de la combinación de los vectores V1[] y V2[], de N elementos cada uno. La función elemental que se aplica a los elementos X_i, $V1_j$ y $V2_j$, para generar cada elemento $t_{i,j}$ es la siguiente:

$$\text{MAP function: } \forall\ i, j\ \{\ t_{i,j} = (X_i - V1_j) * (X_i - V2_j)\ \}$$

La **transformación de tipo Reduce** genera un vector de N valores a partir de N^2 valores usando una función de reducción, que consiste en sumar los elementos. En realidad, podemos pensar que cada elemento del vector resultado R[j] se calcula como la reducción (suma) de todos los valores de la columna j de la matriz tmp[][]. La función elemental que se aplica a los elementos $tmp_{i,j}$ y r_j, para reducir los valores es la siguiente:

$$\text{REDUCE function: } \forall\ i, j\ \{\ r_j = tmp_{i,j} + r_j\ \}$$

MapReduce.c: Fragmento del programa que realiza la parte más significativa del cómputo

```
float *R, *tmp, *X, *V1, *V2;

// Allocate memory for vectors R, X, V1 and V2, and matrix tmp
// Initialize input data: X[], V1[] and V2[]   …

// MAP transformation: Generate 2D matrix from Cartesian product X[] x {V1,V2}[]
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    tmp[i*N+j] = ( X[i] - V1[j] ) * ( X[i] - V2[j] );

// REDUCE transformation: Generate 1D vector from 2D matrix
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    R[j] = R[j] + tmp[i*N+j];
```

Para comprobar que el código anterior sigue siendo correcto tras las optimizaciones que se van a proponer, se ha añadido un código que suma todos los valores del vector resultado y muestra la suma por pantalla.

MapReduce.c: Cálculo de un valor de *CheckSum* para verificar el resultado funcional de la operación

```
for (S=0.0, i=0; i<N; i++)
  S = S + R[i];

printf("CheckSum = %f\n", S);  …
```

Compilar y ejecutar el programa anterior con los compiladores gcc 4.9 e icc 16.0 (opciones –Ofast y –O3, respectivamente) para los tamaños del problema N=20.000, N=40.000 y N=45.000. Utilizar `perf` para medir el tiempo de ejecución, la cantidad de instrucciones ejecutadas y la duración de la ejecución medida en ciclos de reloj (además de otros datos que proporciona y que van a ser útiles para entender el rendimiento del programa).

Pregunta 2a: Rellenar la tabla siguiente, incluyendo el resultado de CheckSum, analizar los resultados, revisar la información extra que proporciona la utilidad `perf` y dar una explicación a estos resultados. **Ayuda**: probar a ejecutar con N=50.000 o N=100.000 para obtener información que nos ayude a entender lo que ocurre.

Compilador	Problem Size	CheckSum	Elapsed Time	Instructions	Cycles
gcc –Ofast (v4.9)	**N=20.000**				
	N=40.000				
	N=45.000				
icc –O3 (v16.0)	**N=20.000**				
	N=40.000				
	N=45.000				

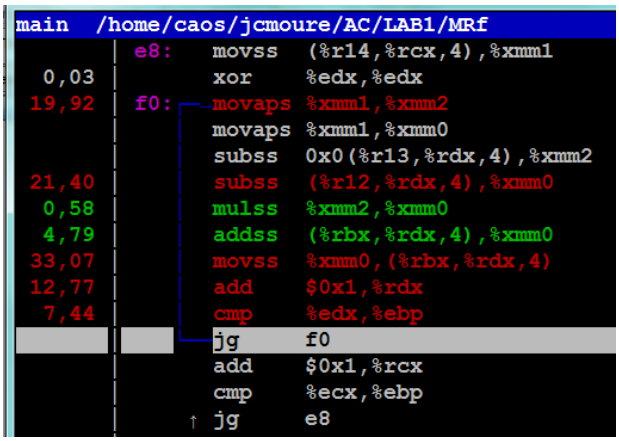
Pregunta 2b: Modificar el código del programa MapReduce.c para corregir el problema grave detectado en el apartado anterior. Con un único tipo de transformación debería ser suficiente para lograr ejecutar el programa para N=50.000 en unos pocos segundos, y para N=200.000 en menos de un minuto. Si no se os ocurre una idea, pedid ayuda al profesor.

Por supuesto, el resultado de *CheckSum* debe utilizarse para verificar que las optimizaciones no afectan (significativamente) a la funcionalidad del programa. En caso de dudas, pedid ayuda al profesor.

Análisis de ensamblador con perf

El comando perf se puede usar como profiler y anotar las instrucciones en ensamblador con el tiempo aproximado que han dedicado a la ejecución. Primero se ejecuta la opción **record**, que genera el archivo perf.data. Después se ejecuta la opción **report**, que por defecto utiliza el archivo perf.data. Con el último comando se puede escoger la opción **annotate** para visualizar el código ensamblador y encontrar aquellas instrucciones que son ejecutadas más veces y que consumen más tiempo de ejecución. Se ha compilado el código modificado propuesto en la pregunta anterior usando el compilador gcc 4.9 con la opción –O2, y el resultado del código ensamblador obtenido usando perf se puede ver a continuación

```
prompt$ gcc -O2 MapRedOptimized -o MRopt
prompt$ perf record ./MRopt 50000
prompt$ perf report
```



Optimizar el código "a mano", usando el compilador con la opción –O1

Pregunta 2c: Optimizar el código del programa para lograr el mejor rendimiento que sea posible utilizando el compilador icc con la opción de optimización –O1 y para N=50.000. La opción de optimización –O1 no genera código vectorizado, ni aunque se use la notación de vectores de Cilk Plus: por tanto no tiene sentido utilizar Cilk Plus para optimizar el código. Se pueden usar cualquiera de las otras ideas analizadas hasta ahora, o se pueden proponer nuevas ideas. Para simplificar las optimizaciones, se puede suponer que el valor de N siempre será múltiplo de 4.

Se debe usar el resultado de *CheckSum* para verificar que las optimizaciones no afectan (significativamente) a la funcionalidad del programa. En caso de dudas, pedid ayuda al profesor.

Una forma de encontrar "ideas", es entender las estrategias de optimización que utilizan los compiladores gcc e icc cuando seleccionamos la opción –O2/–O3. Se puede visualizar y analizar el código ensamblador generado al usar la opción –O2/–O3 para entender estas estrategias.

Importante: se deben explicar las optimizaciones que se proponen, indicando el objetivo y luego evaluando el resultado (utilizando métricas empíricas como IPC, eficiencia de la codificación del programa, etc.)