Práctica 6
Análisis de prestaciones
Procesadores Comerciales
Graduado en Ingeniería Informática

Héctor Lacueva Sacristán\ 869637

Fecha: 26/04/2025

Índice

Preguntas

Todas las preguntas han sido respondidas en base al código generado por el Makefile (nivel de optimización -O2 y compilador gcc) proporcionado en la asignatura para las máquinas del lab102, que cuentan con un procesador Intel i5-9500.

Pregunta 1: ¿Cómo se divide el tiempo de ejecución del programa?

El tiempo de ejecución del programa se divide de la siguiente manera:

50,02%	**49,50%**	**0,48%**
`solution_baseline`	`solution`	`init, compare, rand, ...`

Como se puede apreciar en la tabla, prácticamente todo el tiempo de ejecución lo consumen el `solution_baseline` y el `solution`, acaparando el 99,52% del tiempo de ejecución. El resto del código consume solamente un 0,48%.

Se puede apreciar que un mismo código puede tener diferencias en tiempo de ejecución mostrado en el report, esto puede ser debido a que la precisión no es total u otras razones.

Pregunta 2,3: ¿Cuántas instrucciones ejecuta el programa?, ¿Durante cuántos ciclos se ejecuta el programa?

El comando `perf stat ./build/benchmark -r 1 -s 1024` se obtiene el siguiente resultado:

```
Running benchmark using repetitions = 1 and size = 1024
Solution check is disabled
Execution time: 8.093992 sec

 Performance counter stats for './build/benchmark -r 1 -s 1024':

         8,110.46 msec task-clock                #    1.000 CPUs utilized
               23      context-switches          #    2.836 /sec
                1      cpu-migrations            #    0.123 /sec
            1,604      page-faults               #  197.769 /sec
   35,522,224,509      cycles                    #    4.380 GHz
   14,669,326,631      instructions              #    0.41  insn per cycle
    2,193,910,844      branches                  #  270.504 M/sec
      293,456,874      branch-misses             #   13.38% of all branches

      8.111680045 seconds time elapsed

      8.088515000 seconds user
      0.009948000 seconds sys
```

De aquí obtenemos que se han ejecutado **14.669.326.631 instrucciones** en **35.522.224.509 ciclos**, dando como resultado **0,41 instrucciones por ciclo**.

Pregunta 4: ¿Qué componentes hardware (e.g., TLB, predictor de saltos, caché L1, etc.) están causando paradas en el procesador?

En cuanto a la memoria:

```
BE/Mem     Backend_Bound.Memory_Bound              % Slots      31.9    [33.3%]
BE/Mem     Backend_Bound.Memory_Bound.L1_Bound     % Stalls     0.8   < [33.3%]
BE/Mem     Backend_Bound.Memory_Bound.L2_Bound     % Stalls     -6.0  < [33.3%]
BE/Mem     Backend_Bound.Memory_Bound.L3_Bound     % Stalls     3.6   < [33.3%]
BE/Mem     Backend_Bound.Memory_Bound.DRAM_Bound   % Stalls     39.9    [33.3%]
BE/Mem     Backend_Bound.Memory_Bound.Store_Bound  % Stalls     0.0   < [33.3%]
```

La **DRAM** está provocando la mayoría de las paradas, seguido de la **cache L3** y la **cache L1**.

En cuanto a los saltos:

```
BAD         Bad_Speculation                                    % Slots       33.6
BAD         Bad_Speculation.Branch_Mispredicts                 % Slots       33.6   <==
BAD         Bad_Speculation.Branch_Mispredicts.Other_Mispredicts % Slots      1.3  <
```

El **predictor de saltos** falla bastante.

En cuanto problemas en Backend que no tengan que ver con memoria:

```
BE/Core           Backend_Bound.Core_Bound                     % Slots       27.1    [50.0%]
BE/Core           Backend_Bound.Core_Bound.Divider             % Clocks      38.1    [50.0%]
BE/Core           Backend_Bound.Core_Bound.Serializing_Operation % Clocks     0.1  < [50.0%]
BE/Core           Backend_Bound.Core_Bound.Ports_Utilization   % Clocks      24.7    [50.0%]
```

El **Divider** y el **Ports_Utilization** provocan paradas en el procesador.

Pregunta 5: ¿Qué soluciones nos ayudan a reducir los bottlenecks y por qué?

Backend_Bound:

- **Core_Bound**:
  - **Divider**: al haber una división por un número en todas las iteraciones, se ha calculado la inversa de la división una vez `invAlpha` y ahora en vez de dividir, se multiplica. Con esto consigues una mejoría notable puesto que las divisiones son muy lentas.
- **Memory_Bound**: para mejorar todo lo relacionado con la memoria (tanto cache L1, L2, L3, como DRAM) basta con aplicar técnicas de Tiling ajustando los tamaños de los diferentes Tiles hasta conseguir un buen rendimiento, esto con el fin de reducir el número de accesos a memoria y hacerlos más eficientes.

Bad_Speculation: para reducir esta métrica, basta con evitar las directivas `if/else` para asignar valores a variables, ya que estas por lo general, se transforman en saltos que el predictor no maneja correctamente. Empleando la asignación condicionada podemos conseguir eliminar estos saltos condicionados y, como resultado, se consigue una mejora en el rendimiento.

Además de todo lo mencionado anteriormente, también se ha hecho lo posible para reducir el número de cálculos idénticos realizados.

El código final obtenido es el siguiente:

```
__attribute__((noinline))
void solution(const double *const a,
              const double *const b,
              const double alpha,
              double *const c,
              const int n,
              const double clamp) {

    #define A(i, j) a[i * n + j]
    #define B(i, j) b[i * n + j]
    #define C(i, j) c[i * n + j]

    double aux;
    double invAlpha = 1 / alpha;
    double negCLamp = -clamp;

    const int TILE_I = 8;
    const int TILE_J = (n >> 1);
    const int TILE_K = 4;

    int i_max, j_max, k_max ;

    for (int ii = 0; ii < n; ii += TILE_I) {
        i_max = MIN(ii + TILE_I, n);
        for (int jj = 0; jj < n; jj += TILE_J) {
            j_max = MIN(jj + TILE_J, n);
            for (int kk = 0; kk < n; kk += TILE_K) {
                k_max = MIN(kk + TILE_K, n);
                for (int i = ii; i < i_max; i++) {
```

```
                    for (int j = jj; j < j_max; j++) {
                        aux = C(i,j);
                        aux = (!kk) ? 0.0 : aux ;
                        for (int k = kk; k < k_max; k++) {
                            aux += A(i,k) * B(k,j) * invAlpha;
                            aux = MIN(aux, clamp);
                            aux = MAX(aux, negCLamp);
                        }
                        C(i,j) = aux;
                    }
                }
            }
        }
    }
    #undef A
    #undef B
    #undef C
}
#endif
```

El resultado obtenido de la optimización es el siguiente:

```
FE              Frontend_Bound                      % Slots                      0.7  < [33.3%]
BAD             Bad_Speculation                     % Slots                      0.3  < [33.3%]
BE              Backend_Bound                       % Slots                     19.4  < [33.3%]
RET             Retiring                            % Slots                     79.5    [33.4%]
FE              Frontend_Bound.Fetch_Latency        % Slots                      0.3  < [33.4%]
FE              Frontend_Bound.Fetch_Bandwidth      % Slots                      0.5  < [33.4%]
BAD             Bad_Speculation.Branch_Mispredicts  % Slots                      0.3  < [33.3%]
BAD             Bad_Speculation.Machine_Clears      % Slots                      0.0  < [33.3%]
BE/Mem          Backend_Bound.Memory_Bound          % Slots                      0.9  < [33.3%]
BE/Core         Backend_Bound.Core_Bound            % Slots                     18.5  < [33.3%]
RET             Retiring.Light_Operations           % Slots                     79.2    [33.4%]<==
        This metric represents fraction of slots where the CPU was
        retiring light-weight operations -- instructions that
        require no more than one uop (micro-operation)...
        Sampling events:  inst_retired.prec_dist
RET             Retiring.Heavy_Operations           % Slots                      0.3  < [33.4%]
        This metric represents fraction of slots where the CPU was
        retiring heavy-weight operations -- instructions that
        require two or more uops or micro-coded sequences...
MUX                                                 %                          33.32
```

Pregunta 6: ¿Ha mejorado la métrica de CPI?

El resultado de ejecutar `perf stat ./build/benchmark -r 1 -s 1024` da como resultado lo siguiente:

```
Execution time: 0.886764 sec

 Performance counter stats for './build/benchmark -r 1 -s 1024':

            903.00 msec task-clock                #    0.998 CPUs utilized
                 3      context-switches          #    3.322 /sec
                 0      cpu-migrations            #    0.000 /sec
              3649      page-faults               #    4.041 K/sec
        3954074486      cycles                    #    4.379 GHz
       14398198845      instructions              #    3.64  insn per cycle
        1920816242      branches                  #    2.127 G/sec
            692127      branch-misses             #    0.04% of all branches

       0.904495823 seconds time elapsed

       0.895278000 seconds user
```

```
0.006989000 seconds sys
```

Si tenemos en cuenta la Pregunta 2,3, el número de ciclos por instrucción es mucho mayor en la optimización, **pasa de 0,41 instrucciones por ciclo a 3,64 instrucciones por ciclo**. Por lo tanto, **el CPI de la versión optimizada es mucho menor o mejor**.

CPI v_base	CPI v_opt
2,439	0,274

Pregunta 7: ¿Menor CPI implica siempre menor tiempo de ejecución?

NO. El hecho de que el CPI sea menor, no tiene porque implicar un menor tiempo de ejecución. Existen instrucciones que son rápidas (tardan pocos o fracciones de ciclo) que aumentan el CPI y lentas (varios ciclos) que disminuyen el CPI. No es dificil ver que una instrucción lenta puede ser más rápida para una tarea específica que todo un conjunto de instrucciones rápidas que hacen la misma función y, por lo tanto, la respuesta es NO.

Por no hablar que **si esto fuese cierto, no existirían las instrucciones lentas**.

Pregunta 8: ¿Ha mejorado la versión modificada los MFLOPS del programa?

La formula a utilizar en este apartado es la siguiente:

$$MFLOPS_{programa} = \frac{FLOP_{iter} \times n_{iter}}{T_P}$$

$FLOP_{iter}$ representa el número de FLOP en una sola iteración del bucle más profundo

n_{iter} representa el número de iteraciones totales

T_P es el tiempo de ejecución del programa

Cada una de las dos versiones tiene el mismo número de operaciones, puesto que el cálculo es el mismo, por tanto, con calcular el número de $FLOP_{iter}$ e n_{iter} es el mismo para las dos versiones.

$FLOP_{iter}$ vale 3 ya que se realizan dos multiplicaciones y una suma por iteración.

n_{iter} vale 1.073.741.824, 1024^3.

Los resultados obtenidos son los siguientes:

MFLOPS v_base	MFLOPS v_opt
397,79	3631,13

Como se puede ver en la tabla, ha mejorado en un 912,82% a la versión base.

Pregunta 9: ¿Son los valores de MFLOPS cercanos al valor esperado, teniendo en cuenta la frecuencia del procesador?

Un solo núcleo del procesador (en este caso el i5-9500) tiene una capacidad de:

$$FLOPS = 4.379 \times 10^9 \times 1 = 4.379 \times 10^9 FLOPS$$

$$MFLOPS_{máximos} = 4379 MFLOPS$$

Comparandolos con los resultados obtenidos para la versión optimizada, podemos decir que son relativamente cercanos, teniendo en cuenta el resto de instrucciones ejecutadas por el programa.

Pregunta 10: ¿Qué speedup se consigue?

El speedup conseguido es de:

$$speedup = \frac{Te_{opt}}{Te_{base}} = 9,12$$