Grado en INGENIERÍA INFORMÁTICA

ARQUITECTURA DE COMPUTADORES

102775

Departamento de Arquitectura de Computadores y Sistemas Operativos (DACSO)

PRÁCTICA nº 3

Paralelismo en el Acceso a Memoria y *Prefetching*

Acelerar el Rendimiento con programas *multi-thread* (Multi-core; Many-core y GPU)

Enunciado y Documentación

Medir, Evaluar y Optimizar Rendimiento en Proc. Multicore y GPUs

ENUNCIADO: Se analiza el efecto de las dependencias entre instrucciones, de las latencias de las operaciones de acceso a memoria, de la ejecución reordenada de instrucciones y del efecto de las instrucciones de *prefetching*.

Se introducirán los lenguajes OpenMP y Cilk+ para programar aplicaciones paralelas multi-thread (MIMD= Multiple-Instruction Multiple-Data) con variables compartidas.

Se mostrará el uso del lenguaje OpenACC, de CUDA y de la librería de funciones Thrust para desarrollar aplicaciones que usan el paralelismo masivo para ser ejecutadas en GPUs.

En todos los casos, se medirá el rendimiento usando contadores H/W para entender mejor el comportamiento de la ejecución y se analizarán fragmentos críticos del código ensamblador.

Procesador: Intel Core i7 950, repertorio x86-64

Clock Frequency: 3 – 3.333 GHz; *Out-of-Order Execution* (Dynamic Reordering)
4 Cores x 2-threads (8 H/W threads) + Capacidad SIMD de 128 bits
Private iL1 & dL1: 32KiB, Private L2: 256KiB, Shared L3: 8MiB (64-Byte cache line), 8 GiB DDR-RAM (25.6 GB/sec.)
CPUs: 0-7, Core0= {CPU0, CPU4}, Core1= {CPU1, CPU5}, Core2= {CPU2, CPU6}, Core3= {CPU3, CPU7}
El sistema operativo considera que hay 8 CPUs virtuales, aunque solamente haya 4 núcleos físicos de cómputo.

Una característica importante del procesador es que la frecuencia de funcionamiento varía según el número de núcleos de ejecución que se estén utilizando (Intel denomina a esta característica TurboBoost). Cuando hay uno o dos threads ejecutándose en un único núcleo de cómputo, el sistema funciona a la máxima frecuencia de reloj (3,333 GHz). En cambio, cuando funcionan todos los núcleos el sistema automáticamente reduce la frecuencia de reloj a 3 GHz, para evitar sobrepasar el límite máximo de potencia eléctrica (y de calor generado por el chip).

Sesión 1: TRABAJO PREVIO

Búsqueda binaria secuencial en un vector ordenado

A continuación se muestra el código de la función `BinarySearch`(), que implementa el algoritmo de búsqueda binaria de un número de tipo entero (`target`) en un vector ordenado (`index`[ ]) de N elementos. El algoritmo define un intervalo del vector (`L`, `R`), que inicialmente referencia a todo el vector, y en cada iteración del bucle se va dividiendo a la mitad, asegurando que el primer valor dentro del vector igual al valor buscado siempre está dentro de este intervalo. La función devuelve la posición del vector `index`[] con el primer valor igual o mayor al valor buscado.

```
unsigned int BinarySearch (const int *index, const int N, const int target) {
  unsigned long L=-1, R= N, M;
  int value;

  M = (R - L)>>1;  // Shift operation used to divide by 2. Assume N is not zero
  do {
       M = M + L;
       value = index[M];
       if (value < target)  L=M;  else  R=M;
       M = (R - L)>>1;                                  // Shift operation used to divide by 2
  } while (M);    // M: size of the [L,R) interval in vector index[] where 'target' may reside
  return R;        // index of the first value in vector index[] equal or higher than 'target'
}
```

Pregunta 1a: Evaluar la complejidad computacional del algoritmo. Es decir, cuánto crece la cantidad total de operaciones que hace el programa al incrementar el tamaño del problema, N.

El siguiente código muestra el programa principal que utiliza la función anterior. La primera parte obtiene parámetros de la línea de comandos (usando **`argc`** y **`argv`**). Observar el código con detenimiento para poder identificar el significado de los tres parámetros que utiliza el programa. Se muestra un mensaje de ayuda si en tiempo de ejecución se usa la opción "H" o "h".

```
int main (int argc, char **argv) {
  int N=0, M=0, Rng= -1, i;
  char o = '1';
  int *A, *B, *C;

  if (argc>1) {  o = argv[1][0];    }
  if (argc>2) {  N = atoi(argv[2]); }
  if (argc>3) {  M = atoi(argv[3]); }

  if (o == 'H' || o == 'h')
    {  std::cout << "Arguments: opt N M" << std::endl;     exit(1);    }
```

Luego se reserva memoria de forma dinámica para los vectores A, B y C, de N, M y M elementos, respectivamente. Se utiliza una función **malloc** especial (disponible para el compilador icc) que reserva la memoria en direcciones cuyo valor es un múltiplo del valor que se le indica como segundo parámetro. Se verifica que el sistema haya logrado reservar memoria suficiente para los vectores (recordad que el tamaño de la memoria DRAM está limitado a 8 GBytes).

```
  // allocate free memory in addresses aligned to 64 Bytes
  A = (int *) _mm_malloc ( N*sizeof(int), 64);   B = (int *) _mm_malloc ( M*sizeof(int), 64);
  C = (int *) _mm_malloc ( M*sizeof(int), 64);

  if (A==NULL || B==NULL || C==NULL) // NULL addresses?
    { std::cout << "Malloc Error: no memory!\n" << std::endl;    return 0;  }
```

Posteriormente se inicializan los vectores A y B con números aleatorios y se ordena el contenido del vector A. La sintaxis **std::** indica el uso de clases de la librería estándar (**cout** representa la salida estándar y **sort** es una función de ordenación). Se muestran mensajes durante la ejecución para indicar el punto de la ejecución en el que está el programa.

```
  std::cout << "N (log)= " << N << " (" << bit_scan_reverse(N) << ")  M= " << M << std::endl;

  init_rand (A, N, 1, Rng);    init_rand (B, M, 2, Rng);

  std::cout << "Sorting " << N << " numbers" << std::endl;
  std::sort (A, A+N);
```

A continuación, según la opción seleccionada en el momento de la ejecución (**o**), se realiza el bucle que busca en el vector A[] los M valores contenidos en el vector B[], y que guarda el resultado de la búsqueda en el vector C[].

```
  switch (o) {  // different search implementations
    case '1':
          std::cout << "Binary Search" << std::endl;
          for (i = 0; i < M; i++)
             C[i] = BinarySearch (A, N, B[i]);
  }
```

Finalmente, si se escribe el carácter 'C' tras la opción de ejecución, el programa verifica que todos los resultados de la búsqueda sean correctos y contabiliza el número total de búsquedas de números aleatorios que han encontrado el valor en el vector A[]. Este número se puede utilizar para verificar que diferentes implementaciones del programa sean funcionalmente idénticas. Este código no es trivial de entender, y no es necesario hacerlo para alcanzar los objetivos de esta práctica.

```
  if (argv[1][1] == 'C')    // check results
  {
    long unsigned int n_matches=0;
    std::cout << "Checking Results ... ";
    for (i = 0; i < M; i++) {
      int pos = C[i], val = B[i];
      if (pos>0 && A[pos-1] >= val)
        std::cout <<A[pos-1]<<" in pos. "<< pos-1 <<" greater than "<< val <<"!!"<< std::endl;
      while (pos<N && A[pos] == val)
        { n_matches++; pos++; }
      if (pos<N && A[pos] < val)
        std::cout << A[pos] << " in pos. "<< pos <<" is lower than "<< val <<"!!"<< std::endl;
    }
    std::cout << " number of Matches = " << n_matches << std::endl;
  }
```

No se debe olvidar liberar la memoria reservada de forma dinámica (con funciones **free** especiales).

```
  _mm_free (A);  _mm_free (B);  _mm_free (C);  return 0;
}
```

El código generado por el compilador icc 16.0 con la opción –O3 se muestra a continuación, y al lado del código ensamblador se muestra el código equivalente. El compilador hace dos optimizaciones: (1) desenrollar el bucle del programa dos veces, y (2) substituir las instrucciones de salto que normalmente se utilizarían para codificar la instrucción **if-then-else**, por instrucciones de movimiento condicional (`cmovle` y `cmovg`). Se puede observar que el código ensamblador de cada una de las dos iteraciones desenrolladas consta de 9 instrucciones escalares: el compilador no ha vectorizado, es decir, no ha usado operaciones SIMD. Las 9 instrucciones se descomponen en 10 µoperaciones: un BRN, 8 INT y un LOAD.

```
BinarySearch: …
```

```
 0,72 | cf5:  add    %r8,%r9
 0,15 |       cmp    (%rdi,%r9,4),%r11d
45,71 |       cmovle %r9,%r10
 1,30 |       cmovg  %r9,%r8
 0,43 |       mov    %r10,%r9
 0,36 |       sub    %r8,%r9
 0,63 |       sar    %r9
 0,61 |       test   %r9,%r9
      |     ↓ je     d2f
 0,54 |       add    %r8,%r9
 0,18 |       cmp    (%rdi,%r9,4),%r11d
43,68 |       cmovle %r9,%r10
 1,19 |       cmovg  %r9,%r8
 0,34 |       mov    %r10,%r9
 0,37 |       sub    %r8,%r9
 0,60 |       sar    %r9
 0,57 |       test   %r9,%r9
      |       jne    cf5
```

```
REPEAT:
1.  M = M + L                 // INT
2.  c1= index[M] < target     // LOAD + INT
3.  L =  c1? M: L             // INT
4.  R = !c1? M: R             // INT
5.  M = R                     // INT
6.  M = M – L                 // INT
7.  M = M >> 1                // INT
8.  c2 = M == M               // INT
9.  if (c2) goto END          // BRN
10. M = M + L                 // INT
11. c1 = index[M] < target    // LOAD + INT
12. L =  c1? M: L             // INT
13. R = !c1? M: R             // INT
14. M = R                     // INT
15. M = M – L                 // INT
16. M = M >> 1                // INT
17. c2 = M == M               // INT
18. if (!c2) goto REPEAT      // BRN
END:
```

La primera optimización (desenrollar) no ofrece ningún beneficio en el rendimiento, pero la tarea de optimizar es muy compleja y los compiladores a veces utilizan estrategias generales que no siempre funcionan (al menos tampoco empeora el rendimiento). La segunda opción (usar instrucciones de movimiento condicional) sí que mejora el rendimiento.

Pregunta 1b: Describir con precisión lo que hacen las instrucciones 3 y 4. Cuando el compilador decide usar las instrucciones de movimiento condicional de datos en lugar de usar instrucciones de salto condicional, ¿qué dos beneficios se logran de cara al rendimiento de la ejecución?

Pregunta 1c: El compilador no ha podido vectorizar el bucle interno. Identificar e indicar las razones que impiden vectorizar.

Se toman medidas de la ejecución para diferentes tamaños del vector ordenado (N), siempre haciendo búsquedas de M= 100 millones de valores. Se mide el tiempo de ejecución del programa en dos casos: (a) “Solo Inicialización y Ordenación”, que se salta la parte de realizar búsquedas y la verificación de resultados, pero que incluye el tiempo de generar N+M números aleatorios y de ordenar N números; y (b) “Ejecución completa con Búsqueda”, realizando todo el proceso y las M búsquedas, sin la parte de verificación. Restando estos dos valores se puede aislar el rendimiento debido únicamente a las búsquedas.

Search	Solo Inicialización y Ordenación		Ejecución completa con Búsqueda	
M=100 Millones	Instructions	Time (seconds)	Instructions	Time (seconds)
N= 100.000	8,7 G	0,46	24,2 G	8,46
N= 1.000.000	9,3 G	0,56	28,2 G	16,63
N= 10.000.000	10,9 G	1,37	33,2 G	38,62
N= 100.000.000	32,5 G	10,8	57,5 G	72,7

Pregunta 1d: Calcular el valor de ICount únicamente para la fase de búsqueda: ¿cuánto crece en función de N y por qué?

Pregunta 1e: A partir de la frecuencia de reloj del procesador (3,333 GHz) calcular el valor de IPC (*Instructions Per Cycle*) de la fase de búsqueda y explicar su comportamiento a medida que crece N.

Search	Solamente la fase de Búsqueda			
M=100 Millones	ICount	Time (seconds)	CCount	IPC
N= …				

Optimización: uso de instrucciones S/W de *prefetch*

Se sospecha que el problema de rendimiento de la búsqueda binaria puede ser debido a la **latencia** de los accesos a memoria, especialmente cuando el vector en el que se buscan los valores es muy grande. Para mejorar el rendimiento se aplica una estrategia de pre-búsqueda (*prefetching*) de datos, que consiste en pedir al sistema de memoria los valores de los dos caminos posibles en el árbol binario de búsqueda, antes de verificar cuál de los dos es el que hay que seguir. Tal como se ve en el código siguiente, la estrategia de *prefetch* no se utiliza en todas las iteraciones del bucle de búsqueda, sino solamente en las iteraciones internas (ni en las primeras, que explotan la localidad temporal de los accesos a memoria, ni en las últimas, que explotan la localidad espacial de los accesos).

```
#include “xmmintrin.h”

unsigned int BinSearchPref (const int *index, const int N, const int target) {
  unsigned long L=-1, R= N, M, Ml, Mr;         int value;
  M = (R - L)>>1;
  while (M>= N/1024)                    // while size of search interval higher than N/1024
  {  // 1st phase: no prefetch (hot positions in index: temporal locality exploited in cache)
        M = M + L;
        value = index[M];
        L= (value < target) ? M : L;    R= (value >= target) ? M : R;
        M = (R - L)>>1;
  }
  while (M>=4)                                          // while size of search interval >= 4
  {         // 2nd phase: prefetch (cold positions in index: very few memory access locality)
        M = M + L;

        // prefetch both possible destinations (Left and Right)
        Ml = (L+M)>>1;   Mr = (R+M)>>1;
        _mm_prefetch( (char const *)(&index[Ml]),_MM_HINT_NTA);
        _mm_prefetch( (char const *)(&index[Mr]),_MM_HINT_NTA);

        value = index[M];
        L= (value < target) ? M : L;    R= (value >= target) ? M : R;
        M = (R - L)>>1;
  }
  while (M)                                       // while size of search interval higher than 0
  {     // 3rd phase: no prefetch (spatial locality: vector now fits into single cache block)
        M = M + L;
        value = index[M];
        L= (value < target) ? M : L;    R= (value >= target) ? M : R;
        M = (R - L)>>1;
  }
  return R;
}
```

El código generado por el compilador icc 16.0 con la opción –O3 se muestra a continuación, y al lado del código ensamblador se muestra el código equivalente. El código corresponde con la segunda fase de la búsqueda, e incluye instrucciones de *prefetch*. En este caso el compilador decide no desenrollar el código. Además, el uso de instrucciones condicionales en lenguaje C simplifica el análisis del compilador para que vuelva a usar instrucciones de movimiento condicional (`cmovle` y `cmovg`).

```
BinarySearchPref: …
```

```
 1,30 |43c:┌─add    %rcx,%rax
 0,35 |     |  lea    (%rcx,%rax,1),%r13
 1,23 |     |  shr    %r13
 1,22 |     |  prefet (%r14,%r13,4)
 5,42 |     |  lea    (%r10,%rax,1),%r13
 0,01 |     |  shr    %r13
      |     |  prefet (%r14,%r13,4)
 1,68 |     |  cmp    (%r14,%rax,4),%r9d
60,08 |     |  cmovle %rax,%r10
 2,46 |     |  cmovg  %rax,%rcx
 1,02 |     |  mov    %r10,%rax
 0,71 |     |  sub    %rcx,%rax
 1,36 |     |  shr    %rax
 1,32 |     |  cmp    $0x4,%rax
      |     └─jae    43c
```

```
REPEAT:
1.  M  = M + L                  // INT
2.  Ml = M + L                  // INT
3.  Ml = Ml >> 1                // INT
4.  PREFETCH ( index[Ml] )      // PREFETCH
5.  Mr = M + R                  // INT
6.  Mr = Mr >> 1                // INT
7.  PREFETCH ( index[Mr] )      // PREFETCH
8.  c1= index[M] < target       // LOAD + INT
9.  L =  c1? M: L               // INT
10. R = !c1? M: R               // INT
11. M = R                       // INT
12. M = M – L                   // INT
13. M = M >> 1                  // INT
14. c2 = M >= 4                 // INT
15. if (c2) goto REPEAT         // BRN
```

Cada iteración de la fase 2 de la búsqueda ejecuta ahora 15 instrucciones escalares, que se descomponen en 16

µoperaciones: un BRN, 12 INT, un LOAD y dos PREFETCH (que usan el mismo recurso MEM que las µoperaciones LOAD). Observar que las instrucciones LEA tienen la apariencia de ser operaciones de acceso a memoria, pero en cambio lo que hacen es calcular una dirección (*Load Effective Address*) pero sin acceder a memoria; se utilizan porque permiten usar dos registros de entrada diferentes entre sí, y diferentes al registro donde de guarda el resultado (mientras que las instrucciones de "normales" de suma siempre usan un registro como entrada y como destino).

El siguiente código se ejecuta al seleccionar la opción 'P' en el momento de la ejecución, y realiza el bucle que busca en el vector A[] los M valores contenidos en el vector B[] usando la nueva función de búsqueda usando la estrategia de *prefetch*.

```
    case 'P':
          std::cout << "Binary Search - Prefetch" << std::endl;
          for (i = 0; i < M; i++)
             C[i] = BinSearchPref (A, N, B[i]);
          break;
```

Se vuelven a tomar medidas de la ejecución (completa) para diferentes tamaños del vector ordenado (N), y siempre buscando M= 100 millones de valores. Se puede aislar el tiempo de la búsqueda a partir de los datos de la tabla anterior.

Search M=100 Millones	Prefetch: Ejecución completa con Búsqueda	
	Instructions	Time (seconds)
N= 100.000	21,0 G	7,99
N= 1.000.000	26,3 G	11,82
N= 10.000.000	33,3 G	25,54
N= 100.000.000	60,0 G	49,54

Pregunta 1f: Evaluar y valorar el *speedup* de la nueva versión respecto a la original (solamente la parte de la ejecución del código correspondiente con las M búsquedas).

Sesión 1: TRABAJO DURANTE la SESIÓN

La considerable mejora de rendimiento obtenida al usar la técnica de *prefetching* de datos indica que la latencia de los accesos a memoria es el principal cuello de botella del rendimiento. A continuación se quiere probar una estrategia alternativa que pretende obtener resultados similares a la técnica del *prefetching*. Consiste en combinar dos búsquedas en el mismo bucle, de forma que los accesos a memoria correspondientes a las dos búsquedas se puedan realizar a la vez o de forma solapada. El objetivo, expresado de forma más técnica, es aumentar el paralelismo a memoria de la ejecución.

```
typedef struct S { unsigned int P1; unsigned int P2; } tuple2;  // define a tuple2 data type

tuple2 BinarySearch2 (const int *index, const int N, const int target1, const int target2)
{
  unsigned long L1, R1, M1, L2, R2, M2;
  int value1, value2;

  L1=-1; R1= N;       L2=-1; R2= N;
  M1 = M2 = (R1 – L1)>>1;
  while (M1 && M2)
  {  // combine two searches
    M1 = M1 + L1;          M2 = M2 + L2;
    value1 = index[M1];    value2 = index[M2];
    L2= (value2 < target2) ? M2 : L2;    R2= (value2 >= target2) ? M2 : R2;
    L1= (value1 < target1) ? M1 : L1;    R1= (value1 >= target1) ? M1 : R1;
    M1 = (R1 – L1)>>1;     M2 = (R2 - L2)>>1;
  }

  if (M1) { // one search may take one more step than the other search
    M1 = M1 + L1;    value1 = index[M1];
    L1= (value1 < target1) ? M1 : L1;    R1= (value1 >= target1) ? M1 : R1;
    M1 = (R1 – L1)>>1;
  }
  else if (M2) {
    M2 = M2 + L2;    value2 = index[M2];
    L2= (value2 < target2) ? M2 : L2;    R2= (value2 >= target2) ? M2 : R2;
    M2 = (R2 - L2)>>1;
  }

  tuple2 T;   T.P1= R1; T.P2= R2;   return T;
}
```

El código anterior muestra cómo utilizar una estructura de datos compuesta de dos valores enteros (**tuple2**) para que una función pueda devolver directamente dos valores. Realizar dos búsquedas de forma simultánea solamente requiere duplicar las variables locales necesarias para cada búsqueda y cuidar el caso especial en que dos búsquedas no finalicen exactamente en la misma iteración del bucle. En este último caso, como máximo una de las búsquedas puede necesitar hacer una iteración más que la otra búsqueda. Para simplificar el código, los casos especiales se tratan fuera del bucle principal.

El código siguiente es el que realiza la llamada a la función anterior para realizar las M búsquedas de 2 en 2. Observar que la optimización es equivalente a desenrollar el bucle externo aplicado sobre la lista de valores a buscar, mientras que el bucle de búsqueda no se desenrolla (porque tiene dependencias recurrentes de datos).

```
case '2':  std::cout <<  "Binary Search - fuse 2 searches " << std::endl;
           for (i = 0; i < M; i+=2) {
             tuple2 T;
             T = BinarySearch2 (A, N, B[i], B[i+1]);
             C[i] = T.P1;  C[i+1] = T.P2;
           }
           break;
```

Realizar medidas de tiempo de la ejecución de la versión anterior para M= 100 millones de valores y los valores de N indicados en la siguiente tabla (ejecutar al menos 3 veces y usar el valor más pequeño, porque hay una cierta variabilidad de +/- 0,2 segundos). Recordad que los datos que se miden son de la ejecución completa del programa, incluyendo la inicialización y la ordenación. Los resultados se pueden usar para aislar el tiempo de la parte de la búsqueda a partir de los datos de la primera tabla de este documento. Es necesario compilar y ejecutar tal como se indica:

```
prompt$ icc –O3 binary_search.cpp –o search
prompt$ perf stat ./search 2 100000 100000000
```
(Option=2, N=100.000, M=100.000.000)

Search-Fused2, M=100 Millones	Combine 2 searches: Ejecución completa	
	Instructions	Time (seconds)
N= 100K, 1M, 10M, 100M		

Pregunta 1g: Usar los resultados medidos para calcular la mejora (*speedup*) que se produce respecto a la ejecución single-thread del programa original. Valorar los resultados y comparar esta optimización con la optimización que usaba *prefetching*.

Pregunta 1h: Proponed una optimización que mejore aún más el tiempo de ejecución, quizás combinando o modificando alguna de las propuestas anteriores. Mostrar resultados de tiempo solamente para N= 100 millones y M= 100 millones. Para comprobar que los resultados de la nueva propuesta son idénticos a los anteriores se debe añadir a la opción el carácter 'C' para forzar la comprobación de cada valor.

Alternativamente, en lugar de proponer una versión mejorada, se puede implementar una versión equivalente a la versión con prefetching, pero sin utilizar instrucciones de prefetch, sino utilizando instrucciones "normales" de lectura a memoria. En este caso, se debe realizar una única búsqueda cada vez, en lugar de usar la estrategia de hacer dos búsquedas a la vez.

Sesión 2: TRABAJO PREVIO

Optimización: paralelización de la búsqueda binaria con OpenMP y Cilk+

OpenMP es una extensión de funciones (API) y sentencias especiales integradas en forma de *pragmas*, disponible para algunos lenguajes de programación (C, C++ y Fortran) que permite paralelizar una aplicación secuencial, repartiendo el cómputo entre varios threads (o hilos de ejecución) que **colaboran** compartiendo datos en variables comunes y sincronizándose de forma explícita. En este documento se mostrará el uso de las primitivas más básicas de OpenMP a través de ejemplos sencillos y se usará para ejecutar programas paralelos en procesadores multicore y multithread. OpenMP es un estándar soportado por muchos compiladores, entre ellos los compiladores gcc e icc. Para activar el uso de las directivas o comandos de OpenMP por parte del compilador se usa la opción –fopenmp en el compilador gcc, y de la opción –openmp en el compilador icc.

A continuación se muestra el código paralelizado en forma multithread para CPU, usando las extensiones de OpenMP. Se parametriza el programa para crear NTHR threads, que se reparten el cómputo del programa de forma que cada thread hace la misma cantidad de cómputo (realiza M / NTHR búsquedas).

Fichero: bsearchOMP.cpp

```
  case 'O':
    std::cout <<  "Binary Search-fused OpenMP (" << NTHR << " threads)" << std::endl;

  #pragma omp parallel num_threads (NTHR) // create NTHR parallel threads running in parallel the next block of code
    {
      int i;                              // All threads create a local (or private) copy of these variables
      tuple2 T;

    #pragma omp for schedule (static)     // distribute the M / 2 loop iterations evenly among the NTHR threads
      for (i=0; i < M; i+=2) {
        T = BinarySearch2 (A, N, B[i], B[i+1]);
        C[i] = T.P1;  C[i+1] = T.P2;
      }
    }
```

La directiva **`#pragma omp parallel`** define una región paralela: es decir, indica que el bloque de instrucciones que viene a continuación (en este caso, todas las instrucciones entre {}) será ejecutado simultáneamente por *NTHR* threads. La cláusula **`num_threads(NTHR)`** fuerza el nº total de threads a crear (si no se especifica, el sistema operativo "decide" cuántos threads crear, generalmente tantos como el nº de CPUs virtuales de las que dispone el sistema hardware). Las variables definidas dentro de la parte paralela (int i y tuple2 T) son privadas para cada thread, es decir, hay una copia diferente para cada thread. Por defecto, las variables definidas antes de la parte paralela (A[], B[], C[], N y M) son compartidas, es decir, hay una única copia, común para todos los threads.

La directiva **`#pragma omp for`** debe estar siempre dentro de una región paralela (dentro de un **`#pragma omp parallel`**) y justo antes de una sentencia for, e indica que la ejecución del bucle for que viene a continuación se REPARTIRÁ entre todos los threads que están activos dentro de la región paralela. La cláusula **`schedule(static)`** indica que la repartición de iteraciones entre threads se debe hacer de forma estática: si hay *NTHR* threads y se han de ejecutar *M/2* iteraciones del bucle for, cada thread ejecutará *M* / (2*NTHR)* iteraciones consecutivas y diferentes del bucle original. En este caso, si M=200 y NTHR=4, cada thread hará 25 iteraciones: el 1er thread para los valores de i desde el 0 hasta el 48, incluidos y avanzando de 2 en 2, el 2º thread para los valores 50-98, etc. Si no se indica lo contrario, todos los threads deben haber acabado su parte del trabajo para que todos puedan continuar después de la sentencia *for* (lo que se denomina una **sincronización** global de tipo *barrera*).

Para compilar el código es necesario añadir la opción –openmp al compilador icc. Cuando se miden eventos de rendimiento con el comando **perf** hay que tener en cuenta que siempre se muestra la suma de los eventos generados por todos los threads. En la siguiente tabla se muestran los resultados de la **ejecución completa** usando OpenMP con 1 thread y con 4 threads, para 3 valores diferentes de N. Las primeras 2 columnas ("Inicializar y Ordenar") son para el caso en que se ejecuta toda la aplicación pero sin la parte de las búsquedas en el vector. Aunque hay una cierta variabilidad en todas las ejecuciones, mostramos solamente la variabilidad (desviación estándar) en la última columna, donde es más acusada.

Search-Fused2 M=100 Millones	Inicializar y Ordenar		+Búsqueda: 1 thread		+Búsqueda: 4 threads	
	ICount	Time (s)	ICount	Time (s)	ICount	Time (s)
N= 1.000.000	9,0 G	0,54	26,7 G	9,0	27,2 G	2,9 ± 0,1
N= 10.000.000	10,4 G	1,34	32,1 G	23,2	32,5 G	7,65 ± 0,05
N= 100.000.000	32,5 G	10,8	57,0 G	46,0	57,4 G	21,1 ± 0,1

Pregunta 2a: Calcular y valorar la mejora (*speedup*) que se produce en la ejecución multi-thread respecto a la ejecución single-thread, para cada valor de N, y solamente para la parte de las búsquedas en el vector. Analizar y explicar el incremento en el nº total de instrucciones ejecutadas al usar 4 threads respecto al caso de usar un único thread.

Cilk+ es una extensión al lenguaje C++ para poder crear y sincronizar threads que se ejecuten en paralelo, y que colaboren compartiendo datos en variables comunes. El lenguaje está destinado a generar programas ejecutables en una CPU multicore. En este documento se explicará el uso muy básico de Cilk+, en concreto es uso de la primitiva **`cilk_for`**, mediante un ejemplo muy sencillo, que se ejecutará en una CPU multicore. El lenguaje Cilk+ fue creado en el MIT, como un proyecto de investigación y ha sido adoptado por Intel para sus plataformas multicore y sus aceleradores (Intel Phi). El compilador icc de Intel integra automáticamente el uso de primitivas Cilk+. La versión 4.9.0 de gcc (comando g++) soporta el lenguaje Cilk+ utilizando de forma explícita la opción de compilación –fcilkplus.

El siguiente código es la versión multithread del código, pero programado usando Cilk+. Observar el uso de la palabra reservada _Cilk_for, que le dice al compilador que el bucle **`for`** es paralelizable. El funcionamiento de la primitiva _Cilk_for es el siguiente: se dividen las iteraciones del bucle for en 2 partes iguales (de 0 a M/2 - 1, y de M/2 a M-1) y se generan dos tareas independientes, una para cada grupo de iteraciones. El thread que estaba en ejecución asume una de las tareas, y la otra se asignará dinámicamente a un thread ocioso. Cada thread dividirá su tarea en dos nuevas tareas de la mitad de tamaño (mitad de iteraciones del bucle), y se volverán a poner en actividad dos nuevos threads ociosos. En 2 pasos habrá 4 tareas y 4 threads ejecutando de forma simultánea. En 3 pasos habrá 8 tareas y 8 threads en ejecución. El proceso recursivo de partición de tareas finaliza cuando hay suficientes tareas para cubrir, en exceso, todos los threads disponibles. Es decir, si el tamaño del bucle lo permite, es típico que se creen más tareas que threads, para que durante la ejecución se pueda equilibrar la carga de trabajo de forma dinámica entre los threads (e intentar que todos los threads acaben de ejecutarse más o menos a la vez). Todas las variables definidas fuera del bucle for son compartidas por todos los threads, mientras que la variable i del bucle for y todas las que se declaren dentro del cuerpo del bucle for (en la función Binary_Search) son privadas.

```
  case 'C':
    std::cout <<  "Binary Search-Fused - CILK+ (always 8 threads)" << std::endl;
    _Cilk_for (int i= 0; i<M; i+=2) {   // recursively distribute the M/2 loop iterations into tasks assigned to threads
        tuple2 T;
        T = BinarySearch2 (A, N, B[i], B[i+1]);
        C[i] = T.P1;  C[i+1] = T.P2;
    }
    break;
```

En la siguiente tabla se muestran los resultados de la **ejecución completa** usando Cilk+ con 8 threads (valor por defecto).

Search-Fused2 M=100 Millones	CILK+ (8 threads)	
	ICount	Time (s)
N= 1.000.000	27,0 ± 1 G	2,0 ± 0,1
N= 10.000.000	32,0 ± 1 G	5,2 ± 0,1
N= 100.000.000	57,0 ± 1 G	17,5 ± 0,1

Pregunta 2b: Comparad los resultados entre las versiones OpenMP y Cilk+. Intentar explicar las diferencias en el rendimiento obtenido.

Optimización: paralelización de la búsqueda binaria con OpenACC y Thrust

OpenACC es un estándar de programación paralela de sistemas heterogéneos CPU/GPU (*Open ACCelerators*), que ha sido desarrollado por las compañías Cray, Nvidia y PGI. Igual que OpenMP, el programador anota el código fuente en lenguajes C, C++ y Fortran para identificar las áreas del código que pueden ser aceleradas, y lo hace usando directivas del compilador (`#pragma`) y funciones adicionales. El código especificado por el programador será compilado para ser ejecutado en un dispositivo acelerador acoplado a la CPU (que actúa como host y que denominaremos *host*): por ejemplo las GPUs (*Graphical Processing Units*) de Nvidia y AMD, o el acelerador Xeon Phi de Intel (a los que denominamos aceleradores o *devices*). Este código se descargará al device en tiempo de ejecución, en el momento en que el programa que se ejecuta en el Host lo necesite. Está prevista la extensión futura de OpenMP para que también soporte el uso de aceleradores. El código ejecutado en las GPUs se codifica como un conjunto de miles, millones, o incluso miles de millones de *threads* (o hilos de ejecución) que se reparten el trabajo y colaboran, en ocasiones compartiendo datos en variables comunes y sincronizándose de forma explícita.

En la Figura 1 se muestra el diagrama de bloques de un sistema híbrido CPU-GPU. Un programa no se puede ejecutar directamente en la GPU (*device*) sino que la GPU debe utilizarse como un coprocesador conectado a una CPU tradicional o *host*. Asimismo, tanto CPU como GPU (*host* y *device*) tienen memorias físicas separadas, optimizadas según diferentes métricas: (1) en el caso de la CPU, se optimiza para reducir la latencia de acceso a memoria y ofrecer el máximo de capacidad de almacenamiento, y (2) en el caso de la GPU, se optimiza para maximizar el ancho de banda para accesos secuenciales y, a causa de ello y por razones de coste, se ha de sacrificar la capacidad de almacenamiento. La gestión de las memorias físicas separadas se puede realizar explícitamente por parte del programador o se puede virtualizar mediante mecanismos de colaboración S/W-H/W, como el que se denomina *memoria unificada*.

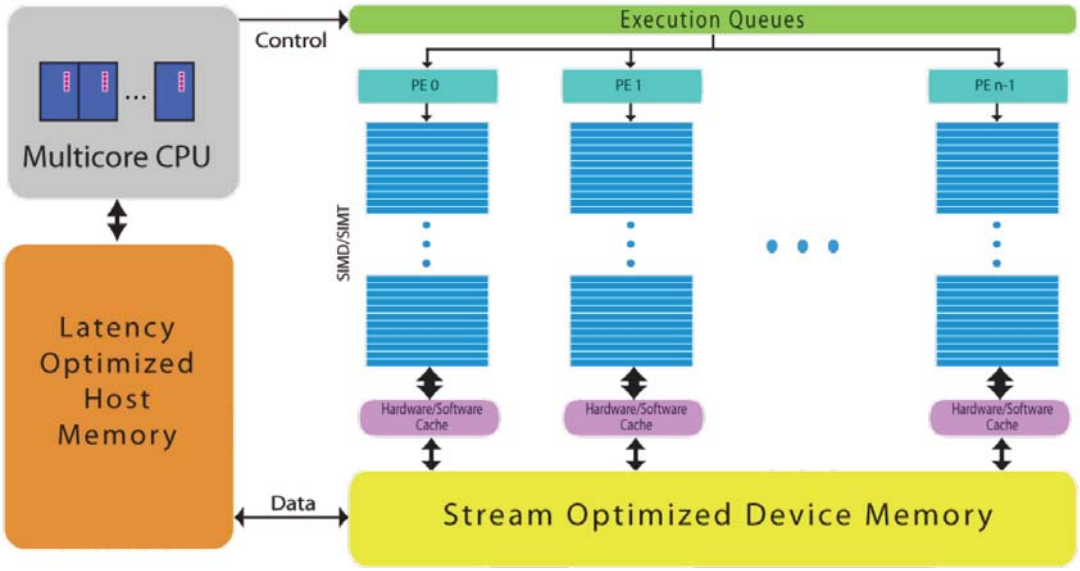


Figura 1. Esquema de un sistema híbrido con una CPU multicore (*host*) y una GPU (acelerador o *device*).

A continuación se muestra el código paralelizado para un sistema híbrido host/device usando las extensiones de OpenACC. La directiva **`#pragma acc data`** define una región de uso del acelerador/device y declara las variables (y su tamaño) que se deben copiar entre host y device. En este caso, al entrar en esta región del código se deben copiar los vectores A y B desde la memoria principal del host hasta la memoria del device, y al salir de esta región del código se deben copiar los vectores A y C desde la memoria del device hasta la memoria del host. A partir de la anterior descripción, debería ser obvio el significado de las cláusulas **`copy()`**, **`copyin()`** y **`copyout()`**.

Fichero: bsearchACC.cpp

```
  #pragma acc data copy( A[0:N] ) copyin( B[0:M] ) copyout( C[0:M] )
                                  // declare the data that must be copied in and out between host and accelerator device
  {
    std::cout << "GPU Sorting (Thrust) " << N << " numbers" << std::endl;

    #pragma acc host_data use_device(A)  // declare A as an accelerator device address to be used by function
    SortThrust( A , N );

    std::cout <<  "Binary Search - OpenACC" << std::endl;

    #pragma acc parallel loop   // distribute the M loop iterations among thousands of threads to be executed in device
    for (int i= 0; i < M; i++)
      C[i] = BinarySearch (A, N, B[i]);
  }
```

Dentro de la región de uso del acelerador hay código que se ejecuta en el host (como las sentencias que muestran mensajes en pantalla usando el *stream* cout de C++) y código que se ejecuta en el device, que está anotado por una directiva. Dejaremos la explicación de la directiva **`#pragma acc host_data`** para más adelante y explicaremos primero la directiva **`#pragma acc parallel loop`**. Es muy similar a la directiva de OpenMP **`omp for`**, también debe ir justo antes de una sentencia for, e indica que la ejecución del bucle for que viene a continuación se REPARTIRÁ entre los miles de threads que se crearán para ser ejecutados en el device. Si no se añaden más cláusulas a la directiva, se delega en el compilador la responsabilidad de decidir cuántos threads hay que generar y cómo repartir las iteraciones entre los threads. Sin un conocimiento profundo del funcionamiento del acelerador (GPU), es una buena decisión dejar que el compilador haga su tarea de forma autónoma.

La directiva `#pragma acc host_data use_device(A)` se utiliza para convertir la referencia al vector A, que es una dirección de memoria (apuntador) en el host, en una dirección de memoria (apuntador) en el device. Sirve para permitir lo que se denomina *interoperabilidad* entre diferentes lenguajes: es decir, para mezclar en la misma aplicación fragmentos diseñados con diferentes lenguajes, como C, OpenACC, C++, Thrust y CUDA. En este ejemplo se utiliza para que la función SortThrust() reciba como parámetro de entrada la dirección en el device del vector A. Esta función se ejecutará en el host, pero en algún momento invocará directamente a una función que se ejecutará en el device para ordenar los elementos del vector A.

Thrust es una biblioteca de *templates* C++ de patrones de ejecución paralela que simplifican la programación de la GPU (también aplicables a la programación de la CPU). Thrust hereda muchas de las características de la biblioteca de C++ denominada STL (*Standard Template Library*). Thrust es completamente compatible con CUDA, el lenguaje básico con el que se programan las GPUs de Nvidia y AMD (se pueden mezclar en un mismo código y fichero instrucciones con la sintaxis de ambos lenguajes) y proporciona una colección de primitivas de operación en paralelo, donde se incluyen operaciones de transformación (*map*), reducción (*reduce*), ordenación, mezcla, etc. Usando Thrust es posible componer diferentes patrones paralelos para implementar ciertos algoritmos complejos de forma simple, concisa y con un código fácil de leer. Estas abstracciones de alto nivel proporcionan a Thrust la libertad de seleccionar de forma automática la implementación más eficiente para las características de la GPU en la que se vaya a ejecutar. También es una opción adecuada para crear prototipos de algoritmos de forma rápida y productiva.

Se puede instalar Thrust junto con todo el CUDA Toolkit, con numerosos ejemplos de código, en la dirección web: https://developer.nvidia.com/cuda-downloads. Thrust se instala automáticamente con el Toolkit de CUDA y para ser usado no se requiere más que incluir en los ficheros de nuestra aplicación la referencia a los ficheros de cabecera para cada uno de los patrones que se utilicen (*#include <thrust/functional.h>*, *#include <thrust/reduce.h>* ...). Los detalles de las funciones-*templates* incluidas en Thrust se pueden encontrar en `/usr/local/cuda/include/thrust`. Todos los ficheros *.h contienen una descripción de las funciones. Todo este código junto con información actualizada está disponible en: https://github.com/thrust/thrust y http://thrust.github.io/.

A continuación mostramos el código correspondiente a la función SortThrust(). Se trata de un *wrapper* (envoltorio) que se ejecuta en el host y que hace un poco de *type-conversion* para poder invocar a la función de ordenación que proporciona Thrust (y que está mucho más optimizada de lo que podría lograr un programador avanzado).

Fichero: bsGPU.cu

```
extern void SortThrust( int * A_device, const int N )
{
  // convert pointer to int into a pointer to a device vector: this is C++, my friends!
  thrust::device_ptr<int> dev_ptr = thrust::device_pointer_cast(A_device);

  // use device_ptr vector to invoke built-in (very efficient) Thrust sort algorithm
  thrust::sort (dev_ptr, dev_ptr + N);
}
```

A continuación se muestran los comandos necesarios para instalar y utilizar el compilador de OpenACC (un compilador de la compañía PGI, de pago, del que solamente se dispone de una licencia gratuita para universidad que permite compilar en el nodo aolin21. Aunque se puede descargar de forma gratuita una versión para un mes). También es necesario instalar y utilizar el compilador de CUDA (este sí es totalmente gratuito), y así poder utilizar la función de ordenación de la biblioteca de Thrust. A continuación se muestran los resultados de la ejecución completa usando una GPU GTX-680 en el nodo aolin21.

```
$ module add cuda/7.5                       // install cuda compiler (versión 7.5)
$ module add pgi64                          // install OpenACC compiler (versión 2016 – 64 bits)
$ export CUDA_VISIBLE_DEVICES=0,1           // make all GPU cards visible to system
$ nvcc –O3 –arch=sm_30 –lineinfo –c –rdc=true bsGPU.cu
                                             // compile CUDA/Thrust file and generate bsGPU.o
$ pgc++ –O3 –acc -ta=tesla:cc30,cuda7.5 bsearchACC.cpp bsGPU.o -Minfo=accel
        –Mcuda=cuda7.5,lineinfo -o bsACC
                                 // compile C/OpenACC code and link with bsGPU.o (use cuda library)
$ perf stat ./bsACC A 1000000 100000000
                                    // run with OpenACC code, N=1.000.000, M= 100.000.000
```

Search M=100 Millones	OpenACC + Thrust		
	ICount (host)	ICount (device)	Time (s)
N= 1.000.000	11,0 G	35 G	2,6
N= 10.000.000	12,2 G	41 G	3,1
N= 100.000.000	19,7 G	69,7 G	4,4

Pregunta 2c: Comparad los mejores resultados usando el procesador multicore con los resultados anteriores, usando la GPU GTX680. Extraer las conclusiones más importantes.

Sesión 2: TRABAJO DURANTE la SESIÓN

Optimización: paralelización de la búsqueda binaria con OpenMP y Cilk+

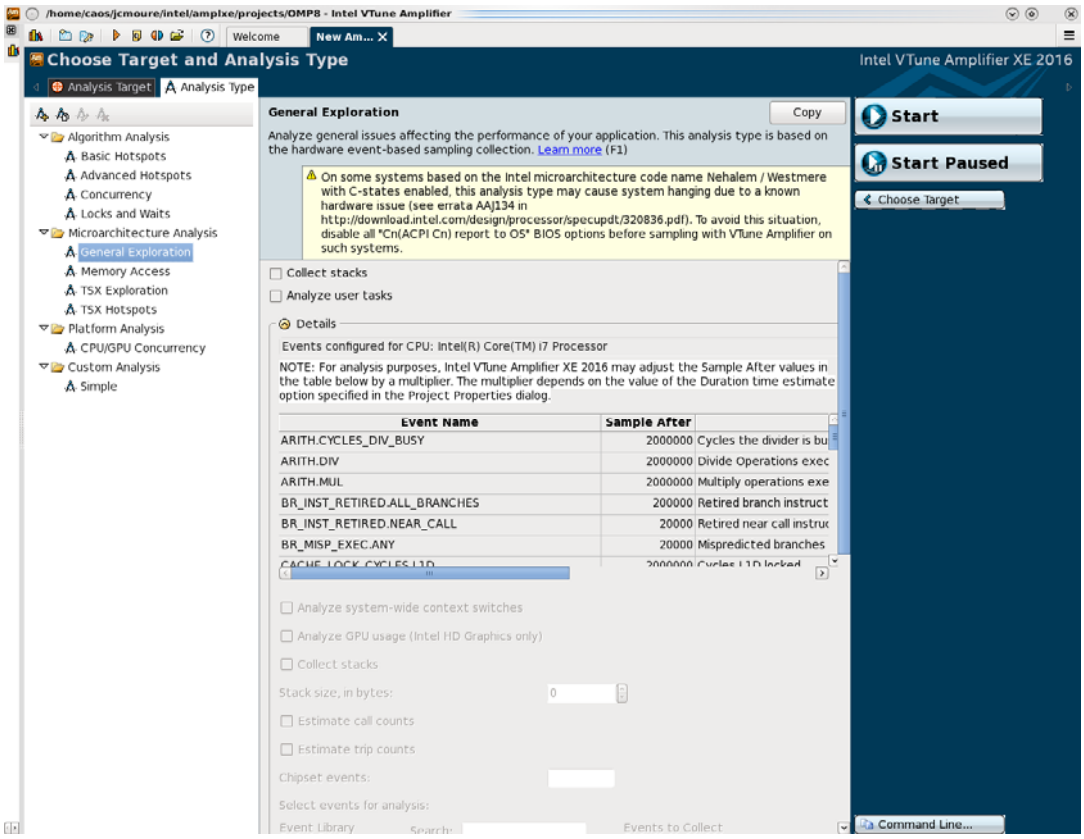
Se debe ejecutar la versión de OpenMP con 8 threads y medir tanto el tiempo de ejecución como el nº total de instrucciones ejecutadas para la **ejecución completa** del programa, para N= 1 millón, 10 millones y 100 millones, y M= 100 millones. Debido a la alta variabilidad en el rendimiento de la ejecución multi-thread, se deben realizar 10 ejecuciones repetidas, y calcular la media de los resultados para las 3 ejecuciones con mejor rendimiento.

```
$ module add intel/16.0.0
$ icc -O3 –g -openmp bsearchOMP.cpp -o bsOMP  // flag –g to include source information
$ perf stat ./bsOMP O 1000000 100000000 8     // versión O (OMP), N= 1M, M= 100M, NTHR=8
```

La diferencia entre usar 4 threads y 8 threads es que en el primer caso se ejecuta un thread diferente en cada núcleo de cómputo de los 4 que tiene el procesador, y en el segundo caso se ejecutan dos threads diferentes en cada núcleo, usando la capacidad de ejecución multi-thread del núcleo (denominada por Intel: *hyperthreading*).

Pregunta 2d: Razonad y explicad la mejora que se produce al pasar de usar 4 threads a usar 8 threads.

Usar la herramienta VTune (`amplxe-gui &`) para visualizar el diagrama de Gantt de la ejecución **tanto con OpenMP para 8 threads como con Cilk+** (siempre se ejecutan 8 threads), para N= 100 millones y M= 100 millones. Revisad el enunciado de la Práctica 1 para recordar cómo se crea una nueva sesión de *profiling*, y cómo en la pestaña etiquetada como "*Analysis Target*" se informa del archivo ejecutable y de los 4 parámetros que se proporcionan en la línea de comandos para definir la ejecución (**`O 100000000 100000000 8`**). En la siguiente imagen se muestra la pestaña etiquetada "*Analysis Type*" que se debe configurar para realizar lo que se denomina "*General Exploration*". Finalmente se debe apretar el botón Start de la derecha y … esperar más de un minuto a que se realice la ejecución y se procesen todos los datos obtenidos.



Pregunta 2e: Visualizar las pestañas "*Summary*", "*Bottom-up*" y "*Platform*", tanto para la ejecución multithread usando 8 threads con OpenMP, como para la ejecución con Cilk +. Explicar los resultados de rendimiento más importantes: IPC, fallos en la caché de último nivel (LLC), uso de los 4 cores del procesador …

Optimización: paralelización de la búsqueda binaria con CUDA y Thrust

En primer lugar, se debe analizar de qué tarjetas gráficas se dispone en el computador del laboratorio. Hay una cierta variedad entre los diferentes computadores, y cada grupo puede encontrar una o dos tarjetas en su computador. Para realizar este análisis se usará una herramienta que viene con el compilador de OpenACC.

```
$ module add pgi64          // install OpenACC compiler (versión 2016 – 64 bits)
$ pgaccelinfo               // analyze all GPU cards visible to system
```

Pregunta 2f: Explicar qué GPU o GPUs existen en el computador y dar los siguientes datos de cada una: GPU *clock rate*, *Memory clock rate*, *Memory bus width*, *Global Memory Size*, *L2 cache size*, número de multiprocesadores y número de núcleos de ejecución. Usando los valores de *Memory clock rate* y *Memory bus width* estimar el máximo ancho de banda de la memoria de la GPU. Estimar también el número de núcleos de cómputo que existen en cada multiprocesador interno a cada GPU.

Desafortunadamente, sólo se dispone de una licencia para utilizar el compilador de OpenACC de la empresa PGI, y la licencia está ligada al nodo aolin21, que se utiliza como nodo de entrada al laboratorio desde una conexión remota. Como no sería práctico que todos los alumnos usaran este nodo de forma simultánea, se va a compilar, ejecutar y analizar una versión equivalente programada usando el lenguaje CUDA y Thrust, la biblioteca C++ de patrones y funciones.

CUDA es una ampliación a los lenguajes C y C++ (y otros, como Python), desarrollada por la compañía Nvidia, que permite expresar un algoritmo de forma explícita como la ejecución de millones de threads (o hilos de ejecución) que se reparten el trabajo y colaboran compartiendo datos en variables comunes y sincronizándose de forma explícita. El lenguaje está destinado a generar programas ejecutables en una GPU (*Graphical Processing Unit*) que sea *CUDA-capable*.

A continuación se mostrará el código del algoritmo de búsqueda binaria para GPU utilizando Thrust y CUDA. En primer lugar se muestra el programa principal, que se ejecuta en la CPU (*host*), que utiliza la biblioteca Thrust. Las líneas 1-3 declaran vectores en la CPU (host), que posteriormente son inicializados con datos aleatorios. La sintaxis utilizada es C++, y no es necesario conocer el lenguaje para poder "intuir" que se declaran vectores de datos de tipo integer, y que los valores N o M indican el número de elementos que tiene el vector. La palabra `thrust` seguida de `::` sirve para identificar el "espacio" donde se definen los tipos de datos y funciones que se deben utilizar.

Fichero: bsearchGPU.cu

```
  // Declare & Allocate host (CPU) vectors
1.   thrust::host_vector<int> A(N);
2.   thrust::host_vector<int> B(M);
3.   thrust::host_vector<int> C(M);

  // initialize host (CPU) vectors
4.   init_rand (thrust::raw_pointer_cast(&A[0]), N, 1, Rng);
5.   init_rand (thrust::raw_pointer_cast(&B[0]), M, 2, Rng);

6.   cudaSetDevice(0);  // select first visible GPU

  // Declare device (GPU) vectors and copy A and B from host to device
7.   thrust::device_vector<int> A_CUDA(A.begin(), A.end());
8.   thrust::device_vector<int> B_CUDA(B.begin(), B.end());
9.   thrust::device_vector<int> C_CUDA(M);

10.  std::cout << "Sorting " << N << " numbers in Device (GPU)" << std::endl;
11.  thrust::sort (A_CUDA.begin(), A_CUDA.end());

12.  std::cout <<  "Binary Search on GPU with CUDA" << std::endl;
13.  SearchCUDA(  // convert device_vectors to device pointers for usage in CUDA
         thrust::raw_pointer_cast(A_CUDA.data()),
         thrust::raw_pointer_cast(B_CUDA.data()),
         thrust::raw_pointer_cast(C_CUDA.data()),
         N, M );

  // Copy C data from device (GPU) memory to host (CPU) memory
14.  C = C_CUDA;
```

Las líneas 4-5 llaman a la función de CPU que inicializa un vector con números aleatorios. Para cumplir las estrictas reglas del lenguaje C++ se necesita la función `raw_pointer_cast` () que transforma un `host_vector` en un puntero a entero.

Las líneas 7-9 declaran réplicas de los vectores declarados anteriormente pero en este caso para ser almacenados en GPU (observar el uso de la palabra *device* en lugar de *host*). Además, en las líneas 7 y 8 se especifica que los datos que hay en los vectores host (CPU) A y B deben ser copiados a los dos primeros vectores *device* (A_CUDA y B_CUDA). La biblioteca de plantillas STL de C++ (*Standard Template Library*) utiliza una forma peculiar de referirse a un vector: A.begin() indica el inicio del vector A y A.end() indica el final del vector. De este modo, en lugar de indicar el tamaño del vector, este tamaño está implícito por la posición inicial y final del vector.

La línea 11 ordena los elementos del vector A_CUDA. Como este vector está en memoria de GPU, Thrust "sabe" que se debe invocar el código para su ejecución en GPU. La línea 13 llama a la función de búsqueda binaria dentro de un vector ordenado que se implementará en lenguaje CUDA, y que describiremos a continuación. También es necesaria la función `raw_pointer_cast` () que transforma un `host_vector` en un puntero a entero. Finalmente, se deben mover los valores del vector C_CUDA, en la memoria de la GPU, al vector C, en la memoria de la CPU, es tan fácil como se muestra en la línea 14. No es necesario liberar de forma explícita la memoria de los vectores, pues el entorno de ejecución de C++ ya se encarga de liberar estas estructuras de forma automática cuando ya están fuera del ámbito de su definición.

A continuación se muestra el código de la función SearchCUDA, que se ejecuta en el host y que invoca directamente la ejecución de una función en la GPU (BsearchGPU). La sentencia en la línea 1 usa una sintaxis especial: nombre de la función, entre "<<<" y ">>>" el nº de threads que tiene cada CTA (*Cooperating Thread Array*) o bloque de threads y el nº de CTAs, y los parámetros para la función que se pasan a todos los threads creados. En el ejemplo se invocan 1024 CTAs, cada uno de ellos compuesto por 1024 threads, es decir, un total de 1 Mega threads.

Fichero: bsGPU.cu

```
extern void SearchCUDA( int * A_CUDA, int * B_CUDA, int * C_CUDA, const int N, const int M )
{
     std::cout <<  "Binary Search on GPU with 1024 CTAs of 1024 threads" << std::endl;
1.   BsearchGPU<<< 1024, 1024 >>> (A_CUDA, N, M, B_CUDA, C_CUDA );
}
```

Finalmente, abajo se muestra el código de la función BsearchGPU, que se ejecuta en la GPU (device). Hay varias características especiales en el código CUDA que lo diferencian de código C normal: (1) se usa la palabra reservada `__global__` antes de la declaración de la función para indicar que es un código que se debe ejecutar en la GPU; (2) las funciones CUDA no pueden devolver resultados (son procedimientos en lugar de funciones) y el resultado debe escribirse utilizando la dirección de memoria de GPU contenida en una variable que se le pase como parámetro (en este caso `*result`); y (3) en tiempo de ejecución cada uno de los threads tiene acceso a ciertas variables especiales que le permiten saber "quién es" y cuántos threads se han creado. threadIdx.x es el identificador del thread dentro de un CTA o bloque de threads. Las GPUs utilizadas tienen una restricción de como máximo 1024 threads por CTA. El número total de threads de un CTA se puede obtener durante la ejecución usando la variable especial blockDim.x. La variable que le dice a un thread cuál es el identificador del CTA o bloque al que pertenece es blockIdx.x. Este identificador puede tomar valores enteros entre 0 y el número de CTAs menos 1. El nº total de CTAs o bloques de threads creados es accesible usando la variable gridDim.x.

```
__global__ void BsearchGPU (const int *index, const int N, const int N2, const int *input,
                            int *result)
{
  int L, R, M, value, i = threadIdx.x + blockIdx.x * blockDim.x; // compute first index i

  while (i < N2) {
    int target= input[i];
    L= -1;  R= N;  M = (R - L)>>1;
    do {
       M = M + L;
       value = index[M];
       L= (value < target) ? M : L;
       R= (value >=  target) ? M : R;
       M = (R - L)>>1;
    } while (M);
    result[i] = R;
    i += gridDim.x * blockDim.x;   // iterate on more indexes in input array input[]
  }
}
```

Cada thread que ejecuta la función BsearchGPU usa sus propios identificadores (threadIdx.x y blockIdx.x) y el tamaño del CTA (blockDim.x) para seleccionar el elemento inicial del vector de entrada que debe ser usado para la búsqueda, y para seleccionar el elemento del vector donde escribir el resultado. Luego los threads van iterando sobre los elementos del vector

de entrada para procesar todos los datos: en cada iteración, un thread debe sumar al identificador i el nº total de threads creados, que se calcula a partir del tamaño de cada CTA y del número de CTAs creados (gridDim.x y blockDim.x).

El comando **`nvidia-smi`** nos ofrece información de las tarjetas gráficas instaladas y de sus características. La variable de estado CUDA_VISIBLE_DEVICES indica los identificadores de las GPUs que son visibles para ser usadas. Se puede habilitar el uso de GPUs con el comando **`export CUDA_VISIBLE_DEVICES=0,1`** ...

El compilador de CUDA se denomina `nvcc` y los ficheros CUDA tienen extensión *.cu, aunque pueden contener funciones definidas para ser ejecutadas en el host, que se compilan con el compilador del host. Los siguientes comandos muestran cómo compilar los dos ficheros que incluyen código CUDA/Thrust y código C. La opción **`–arch`** indica el único tipo de arquitectura de GPU para el que va destinado el código (no se podrá ejecutar en arquitecturas más antiguas ni probablemente más nuevas).

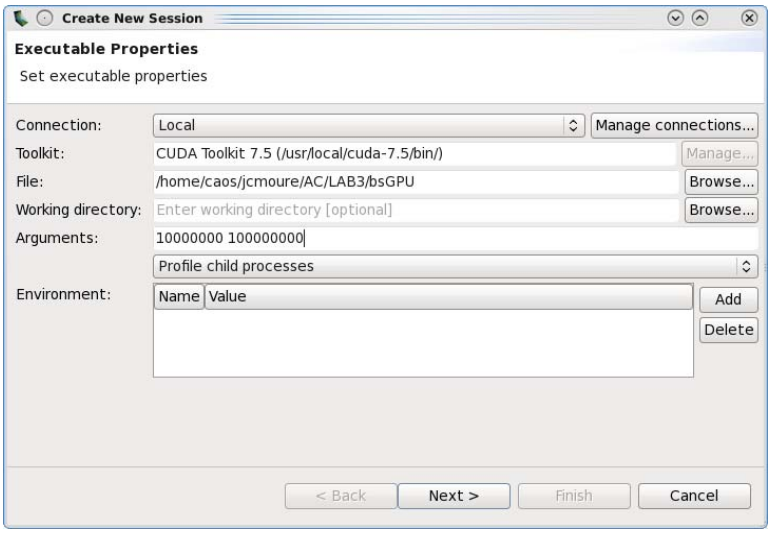
```
$ module add cuda/7.5
$ nvcc –O3 –lineinfo –arch=sm_20 bsGPU.cu bsearchGPU –o bsGPU
                                  // compile CUDA/Thrust file and generate executable bsGPU
$ export CUDA_VISIBLE_DEVICES=0,1       // make all GPU cards visible to system
$./bsGPU 10000000 100000000             // run CUDA code, N=10.000.000, M= 100.000.000
```

Pregunta 2g: Ejecutar el programa usando la GPU con mayor número de núcleos de ejecución que tengáis en vuestro computador. Para ello debéis definir correctamente la variable CUDA_VISIBLE_DEVICES y debéis usar los comandos **`pgaccelinfo`** y **`nvidia-smi`** para comprobar que la única GPU visible es la que realmente queréis utilizar. Haced ejecuciones para N= 1 millón, 10 millones y 100 millones, y M= 100 millones. Comparar el rendimiento usando la GPU con el rendimiento que se obtenía con la versión OpenACC ejecutada en la GPU GTX-680 en el nodo aolin21.

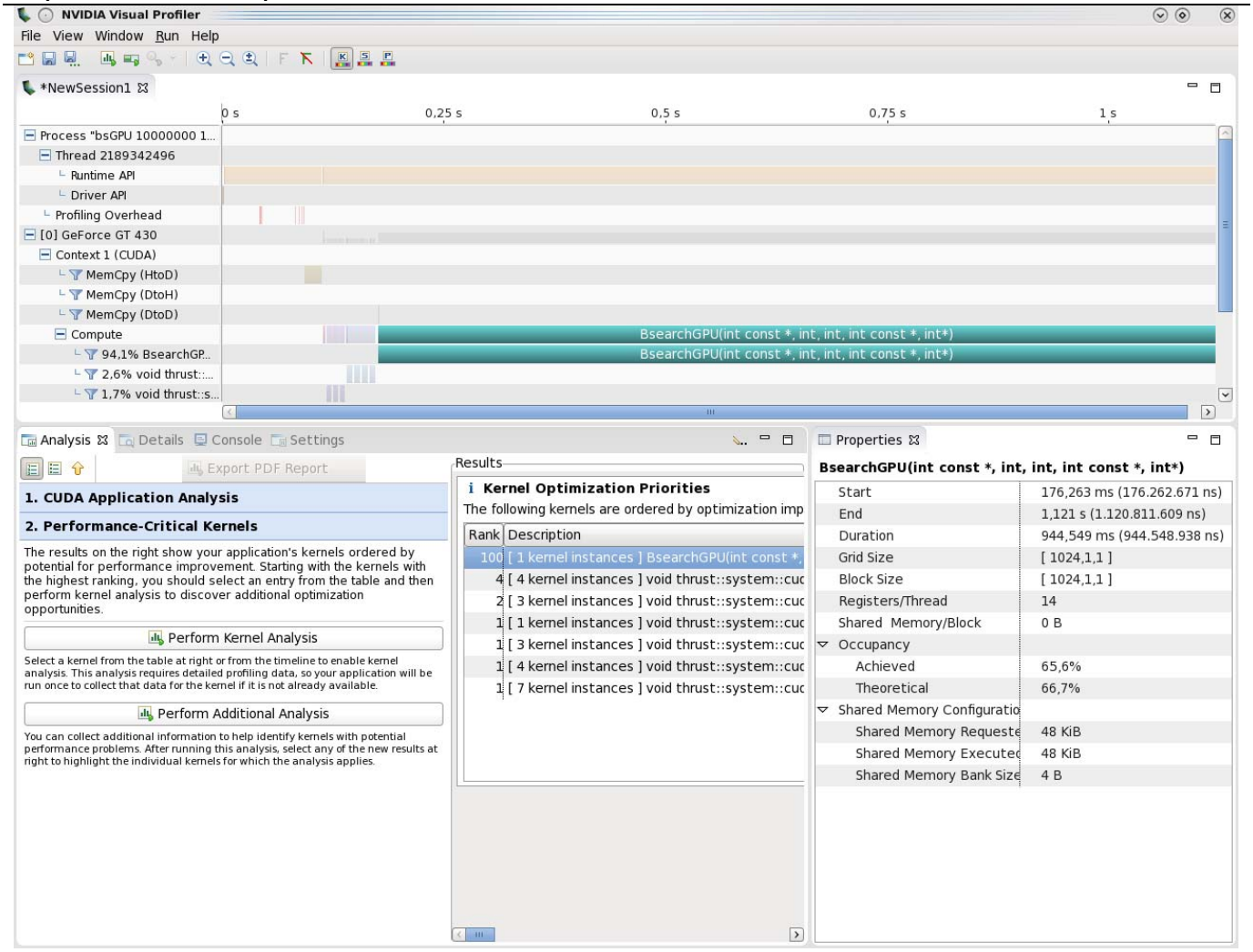
Nota importante: con la tarjeta menos potente tendréis problemas de ejecución. Debéis renunciar a las ejecuciones con tamaños grandes de los vectores. Explicad la causa del problema.

Hay tres herramientas para realizar "*profiling*" de la ejecución de aplicaciones CUDA/Thrust, pero nos limitaremos al uso de la aplicación gráfica `nvvp`. La herramienta `nvvp` tiene un interfaz gráfico similar a VTune y permite visualizar el diagrama de Gantt de la ejecución, que permite identificar qué elementos del sistema se están utilizando a lo largo del tiempo, y qué funciones se están ejecutando. También realiza un análisis automático de los posibles cuellos de botella del rendimiento. El manual de `nvvp` está disponible en **`http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf`**

```
$ nvvp &
```



Pregunta 2h: Ejecutar el *profiler* de GPU y crear una sesión de perfilado tal como se muestra en la ventana de arriba. Usar los parámetros de entrada N= 10 millones, y M= 100 millones, excepto si se tiene que usar una GPU GT 430, en cuyo caso se deben usar los parámetros N= 10 millones, y M= 10 millones. Una vez ejecutada la aplicación, clicar en el nombre de la GPU (izquierda) y buscar en la ventana correspondiente las características de la GPU: comprobar si el ancho de banda máximo de la memoria es el que se calculó previamente. Analizar el diagrama de Gantt de la ejecución y explicar las tareas que se hacen en la GPU (en la página siguiente se muestra el diagrama para la ejecución en la GPU GT 430).



Pregunta 2i: Continuar con el uso del profiler y escoger las siguientes opciones: (1) "*examine individual kernels*"; (2) seleccionar la función BsearchGPU, como en la figura anterior, y escoger la opción "*Perform Kernel Analysis*"; (3) escoger la subopción "*Perform Memory Bandwidth Analysis*"; (4) escoger la opción "*Perform Compute Analysis*" y luego la opción "*Show Kernel Profile – Instruction Execution*", y clicar sobre el nombre de la función para ver el código fuente y el código ensamblador.

Con toda la información generada se debe encontrar: (1) el tiempo total de ejecución en la GPU de la función BsearchGPU, (2) el ancho de banda efectivo a la memoria global (*device memory*) consumido por la función BsearchGPU durante su ejecución, y (3) el nº total de instrucciones ejecutadas (*thread instructions executed*) en la GPU por la función BsearchGPU.

Identificar el cuello de botella de la ejecución para la función BsearchGPU.