PORTAFOLIO PRACTICAS CMCP

Alumno: Javier Meliá Sevilla

1. OPENMP

E1. Ejecución de programas

El nthreads1 indica por cada hilo el número de hilos totales que tenemos en este caso 3 con omp_get_num_threads() y cada hilo individual indica que numero de hilo es gracias a omp_get_thread_num()

El nthreads2 ejecuta el hilo principal omp_get_num_threads() por lo cual antes de paralelizar solo hay 1 y después en el bucle paralelizado con la misma sentencia anterior te indica cuantos hilos se han creado en este caso 3 y luego indica que hilo es y la iteración del bucle que está haciendo

El nthreads3 se puede ver cómo hay dos loops paralelizados donde cada uno muestra el hilo que ejecuta cada iteración y no se mezclan nunca por la barrera que tienen hasta que llega el hilo principal al segundo bucle, ósea primero siempre se muestra el loop 1 y después ya el loop 2

E2. Integración numérica

Para paralelizar el bucle es necesario poner el #pragma omp parallel for y luego las variables a y h son compartidas así que no hace falta añadir nada para ellas, pero para la variable s para que cada hilo haga una porción de **la suma y luego se combinen todas en la suma total es necesario poner un reduction(+:s)**

```c
double calcula_integral(double a, double b, int n)
{
   double h, s=0, result;
   int i;

   h=(b-a)/n;
   #pragma omp parallel for reduction(+:s)
   for (i=0; i<n; i++) {
      s+=f(a+h*(i+0.5));
   }

   result = h*s;
   return result;
}
```

E3. Producto matricial

Para paralelizar el bucle solo hace falta privatizar las variables J y la K ya que cuando accedemos a la variable C cada vez es a una posicion diferente y no hay riesgo de que se modifique por diferentes hilos.

```
void matmat(int n,double *A, double *B, double *C)
{
  int i, j, k;
  #pragma omp parallel for private(j,k)
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      C[i*n+j] = 0.0;
      for (k=0; k<n; k++) {
        C[i*n+j] += A[i*n+k]*B[k*n+j];
      }
    }
  }

}
```

Para mostrar el numero de hilos totales hay varias formas, pero he usado que solo el hilo cero muestre el número de hilos totales.

```
/* Multiplicación de matrices */
#pragma omp parallel
if(omp_get_thread_num()==0){
  printf("threads = %d\n",omp_get_num_threads());
}
t1 = omp_get_wtime();
matmat(n,A,B,C);
t2 = omp_get_wtime();
printf("Tiempo transcurrido: %f s.\n", t2-t1);
```

Aquí una muestra de diversos resultados con diferentes hilos y tamaños de las matrices

```
jamese@alumno.upv.es@knights:~/W/CMCP/src/openmp$ OMP_NUM_THREADS=4 ./matmat 1000
threads = 4
Tiempo transcurrido: 2.398809 s.
jamese@alumno.upv.es@knights:~/W/CMCP/src/openmp$ OMP_NUM_THREADS=4 ./matmat 1000
threads = 4
Tiempo transcurrido: 2.565615 s.
jamese@alumno.upv.es@knights:~/W/CMCP/src/openmp$ OMP_NUM_THREADS=4 ./matmat 2000
threads = 4
Tiempo transcurrido: 32.960919 s.
jamese@alumno.upv.es@knights:~/W/CMCP/src/openmp$ OMP_NUM_THREADS=4 ./matmat 500
threads = 4
Tiempo transcurrido: 0.227094 s.
jamese@alumno.upv.es@knights:~/W/CMCP/src/openmp$ OMP_NUM_THREADS=8 ./matmat 1000
threads = 8
Tiempo transcurrido: 1.191531 s.
jamese@alumno.upv.es@knights:~/W/CMCP/src/openmp$ OMP_NUM_THREADS=16 ./matmat 1000
threads = 16
Tiempo transcurrido: 0.602722 s.
```

E4. Fractales–Paralelización con Regiones Paralelas

Para hacer puede hacer con la región paralela como marca el ejercicio o también al ser solo 3 bucles anidados se podría hacer un #pragma omp parallel for private(i,k) que sería lo mismo.

Al paralelizar hay que tener cuidado con las variables que se calculan en las cabeceras de los for habría que sacarlas fuera y hacer algo semejante, en el caso de la y seria cambiar a y = uly - inc * j; y de la x seria cambiarlo a x = ulx + inc * i

```
#pragma omp parallel
{
  #pragma omp for private(i,k)
  for(j = 0; j < height; j++) {
    y = uly - inc * j;
    /* For each  horizontal line... */
    for(i = 0; i < width; i++) {
      x = ulx + inc * i;
      a = x;
      b = y;

      /* Inner loop for the point (x, y). */
      for(k = 1; k <= maxit; k++) {

        /* The complex equation, which sets the updates to (a, b). */
        u = a * a;
        v = b * b;
        w = 2.0 * a * b;
        a = u - v + x;
        b = w + y;

        /* Check bailout condition, and plot as appropriate. */
        if(u + v > bail) {
          value = (k / idiv + (k % idiv) * (levels / idiv) % levels);
          out[(height-j-1)*width*3+i*3] = value;
          out[(height-j-1)*width*3+i*3+1] = value;
          out[(height-j-1)*width*3+i*3+2] = value;
          break;
        }
      }
    }
  }
}
```

2. MPI

E5 Ejecución de programas

```
jamese@alumno.upv.es@kahan: ~/W/CMCP/src/mpi
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jamese@alumno.upv.es@kahan:~/W/CMCP/src/mpi$ mpicc -Wall -o hellow hellow.c
jamese@alumno.upv.es@kahan:~/W/CMCP/src/mpi$ mpicc -show
gcc -I/opt/openmpi-4.1.1/include -pthread -Wl,-rpath -Wl,/opt/openmpi-4.1.1/lib -Wl,--enable-new-dtags -
L/opt/openmpi-4.1.1/lib -lmpi
jamese@alumno.upv.es@kahan:~/W/CMCP/src/mpi$ mpiexec -n 2 ./hellow
Hello, world, I am 0 of 2
Hello, world, I am 1 of 2
jamese@alumno.upv.es@kahan:~/W/CMCP/src/mpi$ mpiexec -n 3 ./hellow
Hello, world, I am 0 of 3
Hello, world, I am 2 of 3
Hello, world, I am 1 of 3
jamese@alumno.upv.es@kahan:~/W/CMCP/src/mpi$ mpiexec -n 4 ./hellow
Hello, world, I am 3 of 4
Hello, world, I am 1 of 4
Hello, world, I am 2 of 4
Hello, world, I am 0 of 4
jamese@alumno.upv.es@kahan:~/W/CMCP/src/mpi$
```

```
jamese@alumno.upv.es@kahan:~/mpi$ sbatch ej5.sh
Submitted batch job 28476
jamese@alumno.upv.es@kahan:~/mpi$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             28476      mcpd   ej5.sh jamese@a PD       0:00      4 (Resources)
```

E6 Comunicación en anillo

- **Modifica el programa para que el proceso 0 imprima el tiempo transcurrido entre el primer envío y la última recepción, utilizando para ello MPI_Wtime.**

```
if (0 == rank) {
    message = 10;
    t1 = MPI_Wtime();
    printf("Process 0 sending %d to %d, tag %d (%d processes in ring)\n",
           message, next, tag, size);
    MPI_Send(&message, 1, MPI_INT, next, tag, MPI_COMM_WORLD);
    printf("Process 0 sent to %d\n", next);
}
```

```
if (0 == rank) {
    t2 = MPI_Wtime();
    MPI_Recv(&message, 1, MPI_INT, prev, tag_F, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    printf("Tiempo  %f \n", t2-t1);

}
```

- **Modifica el programa para que la finalización se haga por medio de la etiqueta, es decir, usar dos valores de tag, uno normal y otro para indicar la finalización. La operación MPI_Recv debe recibir mensajes de cualquier etiqueta y usar el argumento status para comprobar cuál es la etiqueta del mensaje entrante.**

  Para la segunda parte cuando el proceso 0 decrementa el valor y este llega a 0 ejecuta un último Send con otro tag que será un tag final avisando al siguiente de que ya acaba y sale del bucle y el resto de los procesos lo reciben yse van avisando al siguiente saliendo del bucle también. En el primer recibe hay que cambiar a MPI_ANY_TAG

porque ya hay más de un tag y luego el estado a una variable estado para luego hacer las comprobaciones del tag .

```
while (1) {
    MPI_Recv(&message, 1, MPI_INT, prev, MPI_ANY_TAG, MPI_COMM_WORLD,&estado);
    if(estado.MPI_TAG == tag_F)  {
        MPI_Send(&message, 1, MPI_INT, next, tag_F, MPI_COMM_WORLD);
        printf("Process %d exiting\n", rank);
        break;
    } else if( estado.MPI_TAG == tag){
        if (0 == rank) {
            --message;
            printf("Process 0 decremented value: %d\n", message);
            if (message == 0) {
                printf("Process %d exiting\n", rank);
                MPI_Send(&message, 1, MPI_INT, next, tag_F, MPI_COMM_WORLD);
                break;
            }
        }

        MPI_Send(&message, 1, MPI_INT, next, tag, MPI_COMM_WORLD);

    }
}
```

```
jamese@alumno.upv.es@kahan:~/mpi$ mpiexec -n 3 ./ej6
Process 0 sending 10 to 1, tag 201 (3 processes in ring)
Process 0 sent to 1
Process 0 decremented value: 9
Process 0 decremented value: 8
Process 0 decremented value: 7
Process 0 decremented value: 6
Process 0 decremented value: 5
Process 0 decremented value: 4
Process 0 decremented value: 3
Process 0 decremented value: 2
Process 0 decremented value: 1
Process 0 decremented value: 0
Process 0 exiting
Process 1 exiting
Process 2 exiting
Tiempo  0.000217
jamese@alumno.upv.es@kahan:~/mpi$
```

E7 Ecuación de Poisson - Particionado Unidimensional Vertical

Para este ejercicio es importante dividir bien tanto los vectores como la matriz bien por el numero de procesos eso lo hacemos asignándoles con nLocal a cada uno.

```
int main(int argc, char **argv)
{

  int i, j, N=60, M=60, ld, nLocal,size, rank;
  double *x, *b, h=0.01, f=1.5;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  nLocal = N / size;
  /* Extracción de argumentos */
  if (argc > 1) { /* El usuario ha indicado el valor de N */
    if ((N = atoi(argv[1])) < 0) N = 60;
  }
  if (argc > 2) { /* El usuario ha indicado el valor de M */
    if ((M = atoi(argv[2])) < 0) M = 60;
  }
  ld = M+2;  /* leading dimension */

  /* Reserva de memoria */
  x = (double*)calloc((nLocal+2)*(M+2),sizeof(double));
  b = (double*)calloc((nLocal+2)*(M+2),sizeof(double));
```

Luego enviamos la línea 1 a mi vecino de arriba y se almacena en la línea n_local +1 y la última línea al vecino de abajo y la almacenamos en la línea 0.

```
void jacobi_step_parallel(int N,int M,double *x,double *b,double *t)
{
  int i, j, ld=M+2, nLocal, next, prev,size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) prev = MPI_PROC_NULL;
  else prev = rank-1;
  if (rank == size-1) next = MPI_PROC_NULL;
  else next = rank+1;
  nLocal= N/size;

  MPI_Send(&x[ld], ld, MPI_DOUBLE, prev, 0, MPI_COMM_WORLD);
  MPI_Recv(&x[(nLocal+1)*ld], ld, MPI_DOUBLE, next, MPI_ANY_TAG, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  MPI_Send(&x[nLocal*ld], ld, MPI_DOUBLE, next, 0, MPI_COMM_WORLD);
  MPI_Recv(&x[0], ld, MPI_DOUBLE, prev, MPI_ANY_TAG, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  for (i=1; i<=nLocal; i++) {
    for (j=1; j<=M; j++) {
      t[i*ld+j] = (b[i*ld+j] + x[(i+1)*ld+j] + x[(i-1)*ld+j] + x[i*ld+(j+1)] + x[i*ld+(j-1)])/4.0;
    }
  }

}
```

E8 Calculo de Pi.

Como se puede ver en el código comentado si era el proceso 0 enviaba a todos los demás procesos que reciban del proceso 0, eso se puede sustituir por una llamada colectiva Broadcast.

```
/* Communication: value of n from process 0 to all other processes */
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
//  if (myid == 0) {
//      for (p=1; p<numprocs; p++) MPI_Send(&n, 1, MPI_INT, p, 11, MPI_COMM_WORLD);
//  } else {
//      MPI_Recv(&n, 1, MPI_INT, 0, 11, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
//  }
```

Aquí en el código comentado, se observa queque todos los procesos le envían al proceso 0 su trozo de pi calculado y luego el proceso 0 que los recibe los guarda en una variable y los va sumando, esto seria sustituible como se ve en la imagen por una llamada colectiva Reduce que haría lo mismo.

```
/* Communication: receive in process 0 the value of mypi from the rest of processes
   and accumulate it on pi in process 0 */
MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
//  if (myid == 0) {
//      pi = mypi;
//      for (p=1; p<numprocs; p++) {
//          MPI_Recv(&aux, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 22, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
//          pi += aux;
//      }
//  } else {
//      MPI_Send(&mypi, 1, MPI_DOUBLE, 0, 22, MPI_COMM_WORLD);
//  }
```

E9 Ecuación de Poisson - Programa Completo

El método jacobi_step_parallel se deja igual que en el ejercicio anterior y ahora consiste en modificar con operaciones colectivas de la siguiente manera.

```
void jacobi_poisson(int N,int M,double *x,double *b)
{
  int i, j, k, ld=M+2, conv, maxit=10000, nLocal, size, rank;
  double *t, s, tol=1e-6,aux;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  nLocal= N/size;
  t = (double*)calloc((nLocal+2)*(M+2),sizeof(double));

  k = 0;
  conv = 0;
  //jacobi_step_parallel(N,M,x,b,t);
  while (!conv && k<maxit) {

    /* calcula siguiente vector */
    jacobi_step_parallel(N,M,x,b,t);

    /* criterio de parada: ||x_{k}-x_{k+1}||<tol */
    s = 0.0;
    for (i=1; i<=nLocal; i++) {
      for (j=1; j<=M; j++) {
        s += (x[i*ld+j]-t[i*ld+j])*(x[i*ld+j]-t[i*ld+j]);
      }
    }

    MPI_Allreduce(&s,&aux,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);


    conv = (sqrt(aux)<tol);
    if(rank==0) {
      printf("Error en iteración %d: %g\n", k, sqrt(aux));
    }
    /* siguiente iteración */
    k = k+1;
    for (i=1; i<=nLocal; i++) {
      for (j=1; j<=M; j++) {
        x[i*ld+j] = t[i*ld+j];
      }
    }

  }

  free(t);
}
```

Para que todos los procesos envíen su parte de x al proceso 0 se ha creado una variable aux para recibir ahí todos los fragmentos.

```c
int main(int argc, char **argv)
{
  int i, j, N=60, M=60, ld, nLocal, size, rank;
  double *x, *b, h=0.01, f=1.5, *aux2;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  nLocal = N / size;
  /* Extracción de argumentos */
  if (argc > 1) { /* El usuario ha indicado el valor de N */
    if ((N = atoi(argv[1])) < 0) N = 60;
  }
  if (argc > 2) { /* El usuario ha indicado el valor de M */
    if ((M = atoi(argv[2])) < 0) M = 60;
  }
  ld = M+2;  /* leading dimension */

  /* Reserva de memoria */
  x = (double*)calloc((nLocal+2)*(M+2),sizeof(double));
  b = (double*)calloc((nLocal+2)*(M+2),sizeof(double));
  if(rank == 0) {
    aux2 = (double*)calloc((N+2)*(M+2),sizeof(double));
  }
  /* Inicializar datos */
  for (i=1; i<=nLocal; i++) {
    for (j=1; j<=M; j++) {
      b[i*ld+j] = h*h*f;  /* suponemos que la función f es constante en todo el dominio */
    }
  }

  /* Resolución del sistema por el método de Jacobi */
  jacobi_poisson(N,M,x,b);
  MPI_Gather(&x[ld] ,ld*nLocal, MPI_DOUBLE, aux2, ld*nLocal,MPI_DOUBLE,0,MPI_COMM_WORLD);

  /* Imprimir solución (solo para comprobación, se omite en el caso de problemas grandes) */


   if(rank == 0) {
    if (N<=60) {
      for (i=1; i<=N; i++) {
        for (j=1; j<=M; j++) {
          printf("%g ", aux2[i*ld+j]);
        }
        printf("\n");
      }
    }
    free(aux2);
  }

  free(x);
  free(b);
  MPI_Finalize();

  return 0;
}
```

Ej 10 Ecuación de Poisson - Particionado Unidimensional Horizontal

```c
int main(int argc, char **argv)
{
  int i, j, N=60, M=60, ld, mLocal, size, rank;
  double *x, *b, h=0.01, f=1.5, *aux2;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  mLocal = M / size;

  /* Extracción de argumentos */
  if (argc > 1) { /* El usuario ha indicado el valor de N */
    if ((N = atoi(argv[1])) < 0) N = 60;
  }
  if (argc > 2) { /* El usuario ha indicado el valor de M */
    if ((M = atoi(argv[2])) < 0) M = 60;
  }
  ld = mLocal+2;  /* leading dimension */

  /* Reserva de memoria */
  x = (double*)calloc((N+2)*(mLocal+2),sizeof(double));
  b = (double*)calloc((N+2)*(mLocal+2),sizeof(double));
  if(rank == 0) {
    aux2 = (double*)calloc((N+2)*(M+2),sizeof(double));
  }
  /* Inicializar datos */
  for (i=1; i<=N; i++) {
    for (j=1; j<=mLocal; j++) {
      b[i*ld+j] = h*h*f;  /* suponemos que la función f es constante en todo el dominio */
    }
  }

  /* Resolución del sistema por el método de Jacobi */
  jacobi_poisson(N,M,x,b);
  MPI_Datatype column_resized;
  MPI_Type_create_resized(column_not_resized, 0, sizeof(int), &column_resized);
  MPI_Type_commit(&column_resized);

  MPI_Gather(&x[ld] ,ld*nLocal, MPI_DOUBLE, aux2, ld*nLocal,MPI_DOUBLE,0,MPI_COMM_WORLD);

  /* Imprimir solución (solo para comprobación, se omite en el caso de problemas grandes) */
  if (N<=60) {
    for (i=1; i<=N; i++) {
      for (j=1; j<=mLocal; j++) {
        printf("%g ", x[i*ld+j]);
      }
      printf("\n");
    }
  }

  free(x);
  free(b);
  MPI_Finalize();

  return 0;
}
```

```
void jacobi_step_parallel(int N,int M,double *x,double *b,double *t)
{
  int i, j, ld, mLocal, next, prev,size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) prev = MPI_PROC_NULL;
  else prev = rank-1;
  if (rank == size-1) next = MPI_PROC_NULL;
  else next = rank+1;
  mLocal= M/size;
  ld=mLocal+2;

  MPI_Datatype columna;
  MPI_Type_vector(N+2, 1, mLocal+2, MPI_DOUBLE, &columna);
  MPI_Type_commit(&columna);
  MPI_Send(&x[1], 1, columna, prev, 0, MPI_COMM_WORLD);
  MPI_Recv(&x[ld], 1, columna, next, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(&x[ld-1], 1, columna, next, 0, MPI_COMM_WORLD);
  MPI_Recv(&x[0], 1, columna, prev, MPI_ANY_TAG, MPI_COMM_WORLD,MPI_STATUS_IGNORE);

  for (i=1; i<=N; i++) {
    for (j=1; j<=mLocal; j++) {
      t[i*ld+j] = (b[i*ld+j] + x[(i+1)*ld+j] + x[(i-1)*ld+j] + x[i*ld+(j+1)] + x[i*ld+(j-1)])/4.0;
    }
  }

}
```

3. CUDA

E11 Información de la GPU

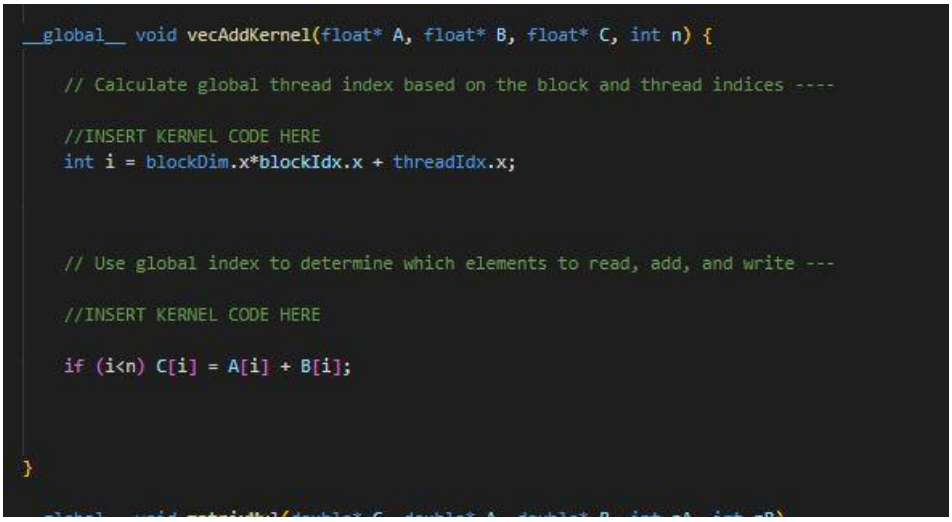
Solo haría falta añadir el método deviceProp.memoryBusWidth para ver la anchura del bus de memoria

```
#endif
    printf("   Memory bus width:                              %d\n",
           deviceProp.memoryBusWidth);
}
printf("\nTEST PASSED\n");
```

```
jamese@alumno.upv.es@gpu:~/W/CMCP/src/cuda/dquery$ ./dquery
There are 2 devices supporting CUDA

Device 0: "Quadro RTX 5000"
  Major revision number:                         7
  Minor revision number:                         5
  Total amount of global memory:                 16908615680 bytes
  Number of multiprocessors:                     48
  Number of cores:                               384
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.81 GHz
  Concurrent copy and execution:                 Yes
   Memory bus width:                                 256

Device 1: "Quadro RTX 5000"
  Major revision number:                         7
  Minor revision number:                         5
  Total amount of global memory:                 16905469952 bytes
  Number of multiprocessors:                     48
  Number of cores:                               384
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.81 GHz
  Concurrent copy and execution:                 Yes
   Memory bus width:                                 256

TEST PASSED
```

E12 Suma de vectores

Primero hay que modifica el archivo kernel.cu



```
__global__ void vecAddKernel(float* A, float* B, float* C, int n) {

    // Calculate global thread index based on the block and thread indices ----

    //INSERT KERNEL CODE HERE
    int i = blockDim.x*blockIdx.x + threadIdx.x;



    // Use global index to determine which elements to read, add, and write ---

    //INSERT KERNEL CODE HERE

    if (i<n) C[i] = A[i] + B[i];



}
```

Luego habría que modificar el programa main.cu y ya ejecutarlo con el makefile.

```
int main(int argc, char**argv) {

    Timer timer;
    cudaError_t cuda_ret;

    // Initialize host variables ----------------------------------------------

    printf("\nSetting up the problem..."); fflush(stdout);
    startTime(&timer);

    unsigned int n;
    if(argc == 1) {
        n = 10000;
    } else if(argc == 2) {
        n = atoi(argv[1]);
    } else {
        printf("\n    Invalid input parameters!"
           "\n    Usage: ./vecadd              # Vector of size 10,000 is used"
           "\n    Usage: ./vecadd <m>          # Vector of size m is used"
           "\n");
        exit(0);
    }

    float* h_A = (float*) malloc( sizeof(float)*n );
    for (unsigned int i=0; i < n; i++) { h_A[i] = (rand()%100)/100.00; }

    float* h_B = (float*) malloc( sizeof(float)*n );
    for (unsigned int i=0; i < n; i++) { h_B[i] = (rand()%100)/100.00; }

    float* h_C = (float*) malloc( sizeof(float)*n );

    stopTime(&timer); printf("%f s\n", elapsedTime(timer));
    printf("    Vector size = %u\n", n);

    // Allocate device variables ----------------------------------------------

    printf("Allocating device variables..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HERE
    int nhilos = 256;
    int nbloques = ceil(float(n)/nhilos);
    size_t size = n*sizeof(float);
    float *d_A, *d_B, *d_C;


    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Copy host variables to device ------------------------------------------

    printf("Copying data from host to device..."); fflush(stdout);
    startTime(&timer);
```

```
    //INSERT CODE HERE
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Launch kernel ----------------------------------------------------------

    printf("Launching kernel..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HERE
    vecAddKernel<<<nbloques,nhilos>>>(d_A, d_B, d_C, n);

    cuda_ret = cudaDeviceSynchronize();
    if(cuda_ret != cudaSuccess) FATAL("Unable to launch kernel");
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Copy device variables from host ----------------------------------------

    printf("Copying data from device to host..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HERE
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);


    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Verify correctness -----------------------------------------------------

    printf("Verifying results..."); fflush(stdout);

    verify(h_A, h_B, h_C, n);

    // Free memory ------------------------------------------------------------

    free(h_A);
    free(h_B);
    free(h_C);

    //INSERT CODE HERE
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);



    return 0;

}
```

Ej 13 Multiplicación de matrices

- **Implementa una versión de simple precisión para comparar las prestaciones con respecto de la versión de doble precisión.**

Para este apartado lo que se ha hecho es descargar la carpeta matriMul y realizar las pruebas con simple y doble precisión y los resultados son los siguientes:

Con el formato de simple precisión y unas matrices de tamaño A(512 X 2048) y B (2048X512), el resultado da unos 13 segundos como se puede observar.

```
./matrixMul -wa=512 -ha=2048 -wb=2048 -hb=512
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Turing" with compute capability 7.5

MatrixA(512,2048), MatrixB(2048,512)
Computing result using CUDA Kernel...
done
Performance= 323.35 GFlop/s, Time= 13.283 msec, Size= 4294967296 Ops,
WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
```

Con el formato de Doble precisión y unas matrices de tamaño A(512 X 2048) y B (2048X512), el resultado da menos de 3 segundos como se puede observar.

```
./matrixMul -wa=512 -ha=2048 -wb=2048 -hb=512
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Turing" with compute capability 7.5

MatrixA(512,2048), MatrixB(2048,512)
Computing result using CUDA Kernel...
done
Performance= 1533.17 GFlop/s, Time= 2.801 msec, Size= 4294967296 Ops,
WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
```

- **En /usr/local/cuda/samples/0_Simple hay dos versiones del producto de matrices, una similar a la estudiada en clase (usando memoria compartida) y otra haciendo una llamada a la librería cuBLAS. Compara las prestaciones de las tres versiones para diferentes tamaños de matriz.**

```
./matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Turing" with compute capability 7.5

GPU Device 0: "Quadro RTX 5000" with compute capability 7.5

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
Performance= 5142.28 GFlop/s, Time= 0.038 msec, Size= 196608000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```