### Profiling de CPU

Aplicar el programa GProf, y encontrar la función más pesada del programa. El alumno deberá realizar capturas de pantalla que demuestren la correcta ejecución de las siguientes partes:

* Compilación con los flags “-pg” del código entregado.
* Ejecución y generación del archivo gmon.out.
* Análisis del archivo gmon.out usando el programa gprof.

Después de ejecutar el programa Gprof, se discutirá cual es la función/método más pesado del programa que debería de optimizarse.

Solución:

estudiante@utwks0383:~/Escritorio/ProfilingOpenMPI/EjercicioCPU$ gcc -g -pg fuente.cpp -o fuente

estudiante@utwks0383:~/Escritorio/ProfilingOpenMPI/EjercicioCPU$ ./fuenteestudiante@utwks0383:~/Escritorio/ProfilingOpestudiante@utwks0383:~/Escritorio/ProfilingOpenMPI/EjercicioCPU$

estudiante@utwks0383:~/Escritorio/ProfilingOpenMPI/EjercicioCPU$

estudiante@utwks0383:~/Escritorio/ProfilingOpenMPI/EjercicioCPU$ gprof ./fuente gmon.out > estadisticas.txt

La función que más tiempo se ejecuta es:

% cumulative self self total

time seconds seconds calls ms/call ms/call name

101.03 0.21 0.21 1001000 0.00 0.00 fun3(int)

### Profiling de errores de memoria:

Aplicar Valgrind-Memcheck, y demostrar que el programa está libre de errores de acceso a memoria. Para ello, se mostrarán los comandos usados para ejecutar el programa, junto con una captura de pantalla del resultado obtenido. En caso de que hubiera errores de memoria, indicar los pasos realizados para solucionarlos.

Solución:

Ejecución usando valgrind Memcheck

estudiante@utwks0383:~/Escritorio/ProfilingOpenMPI/EjercicioCPU$ valgrind ./fuente -Memcheck

==3093== Memcheck, a memory error detector

==3093== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.

==3093== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info

==3093== Command: ./fuente -Memcheck

==3093==

==3093==

==3093== Process terminating with default action of signal 27 (SIGPROF)

==3093== at 0x4F42E0F: write\_gmon (gmon.c:354)

==3093== by 0x4F43589: \_mcleanup (gmon.c:422)

==3093== by 0x4E73FF7: \_\_run\_exit\_handlers (exit.c:82)

==3093== by 0x4E74044: exit (exit.c:104)

==3093== by 0x4E5A836: (below main) (libc-start.c:325)

==3093==

==3093== HEAP SUMMARY:

==3093== in use at exit: 40,041,980 bytes in 1,001,001 blocks

==3093== total heap usage: 1,001,001 allocs, 0 frees, 40,041,980 bytes allocated

==3093==

==3093== LEAK SUMMARY:

==3093== definitely lost: 40,040,000 bytes in 1,001,000 blocks

==3093== indirectly lost: 0 bytes in 0 blocks

==3093== possibly lost: 0 bytes in 0 blocks

==3093== still reachable: 1,980 bytes in 1 blocks

==3093== suppressed: 0 bytes in 0 blocks

==3093== Rerun with --leak-check=full to see details of leaked memory

==3093==

==3093== For counts of detected and suppressed errors, rerun with: -v

==3093== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

El tiempo de CPU expiró

Se puede ver que no se hace free de la reserva dinámica del array x en la funcion pruebamalloc

void pruebamalloc(int y)

{

int\* x=(int\*)malloc(10\*sizeof(int));

//x[1000]=5;

x[9]=5;

}

Habría que añadir una sentencia free para liberar la memoria

void pruebamalloc(int y)

{

int\* x=(int\*)malloc(10\*sizeof(int));

//x[1000]=5;

x[9]=5;

// Añadido para evitar memory leaks

free(x);

}

Después de este cambio se puede compilar el programa y realizar la ejecución de nuevo:

estudiante@utwks0383:~/Escritorio/ProfilingOpenMPI/EjercicioCPU$ gcc -g -pg fuente.cpp -o fuente

estudiante@utwks0383:~/Escritorio/ProfilingOpenMPI/EjercicioCPU$ valgrind ./fuente -Memcheck

==3093== Memcheck, a memory error detector

==3093== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.

==3093== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info

==3093== Command: ./fuente -Memcheck

==3093==

==3093==

==3093== Process terminating with default action of signal 27 (SIGPROF)

==3093== at 0x4F42E0F: write\_gmon (gmon.c:354)

==3093== by 0x4F43589: \_mcleanup (gmon.c:422)

==3093== by 0x4E73FF7: \_\_run\_exit\_handlers (exit.c:82)

==3093== by 0x4E74044: exit (exit.c:104)

==3093== by 0x4E5A836: (below main) (libc-start.c:325)

==3093==

==3093== HEAP SUMMARY:

==3093== in use at exit: 40,041,980 bytes in 1,001,001 blocks

==3093== total heap usage: 1,001,001 allocs, 0 frees, 40,041,980 bytes allocated

==3093==

==3093== LEAK SUMMARY:

==3093== definitely lost: 40,040,000 bytes in 1,001,000 blocks

==3093== indirectly lost: 0 bytes in 0 blocks

==3093== possibly lost: 0 bytes in 0 blocks

==3093== still reachable: 1,980 bytes in 1 blocks

==3093== suppressed: 0 bytes in 0 blocks

==3093== Rerun with --leak-check=full to see details of leaked memory

==3093==

==3093== For counts of detected and suppressed errors, rerun with: -v

==3093== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

El tiempo de CPU expiró

### Profiling en GPU

La siguiente parte consistirá en una implementación GPU del algoritmo presentado inicialmente de suma de vectores. Una vez implementado, se mostrarán capturas de pantalla con el resultado de la ejecución, indicando el porcentaje de tiempo del programa en las partes más significativas (funciones GPU y copia de datos a memoria)

Solución:

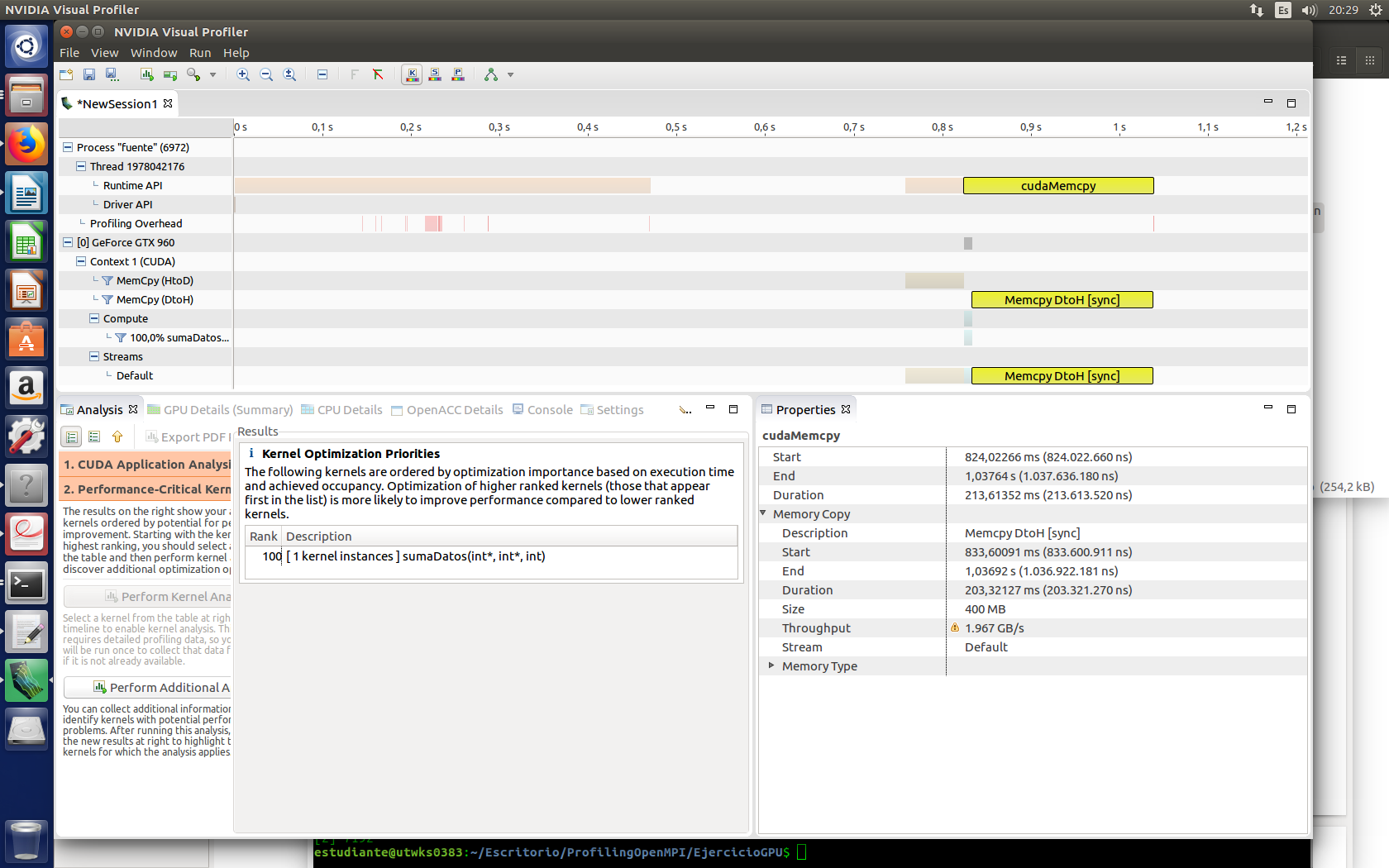
Tiempo copiando los datos desde cpu a device es 66.30337 ms

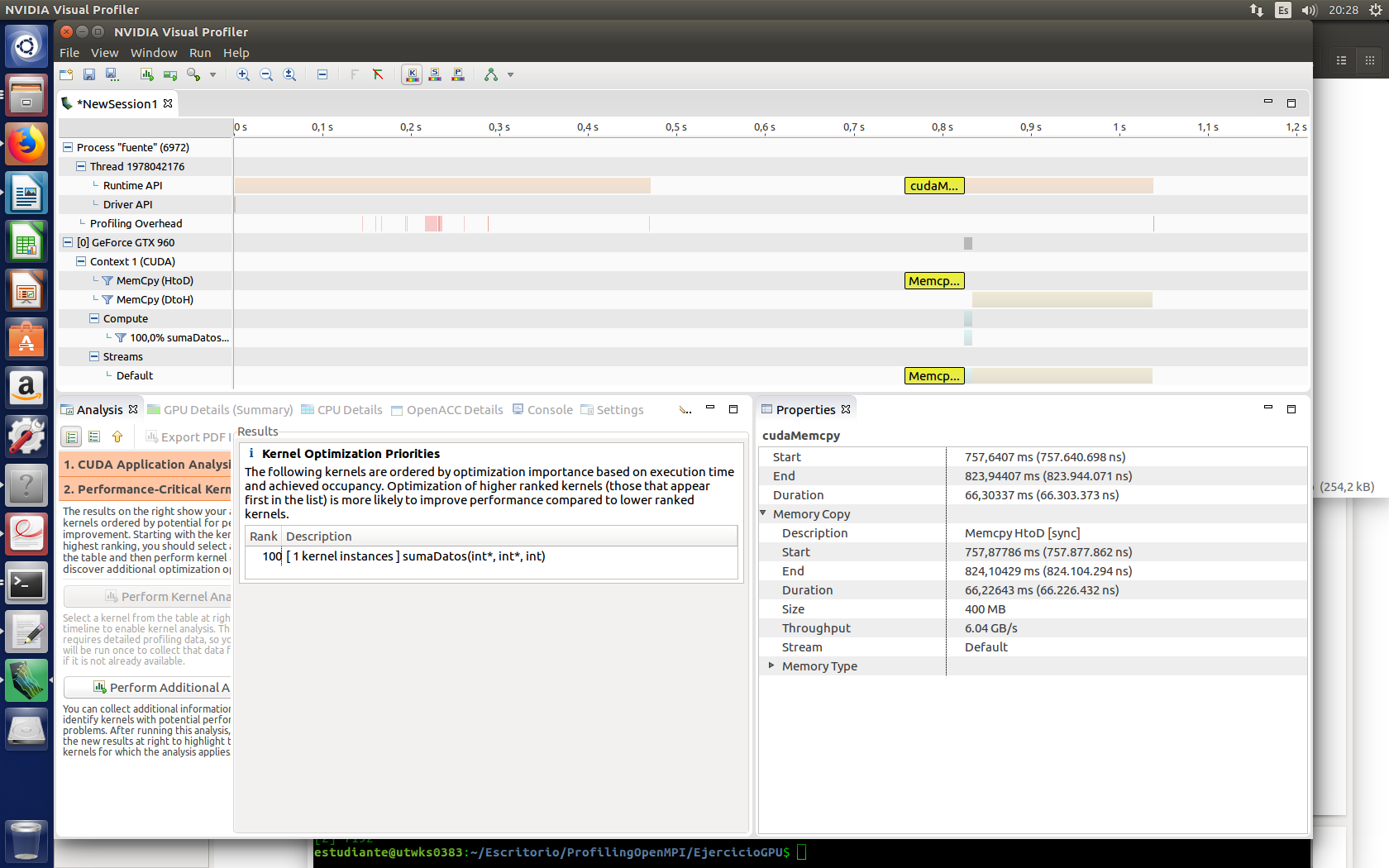
Tiempo copiando los datos de GPU a CPU es de 203,61352ms

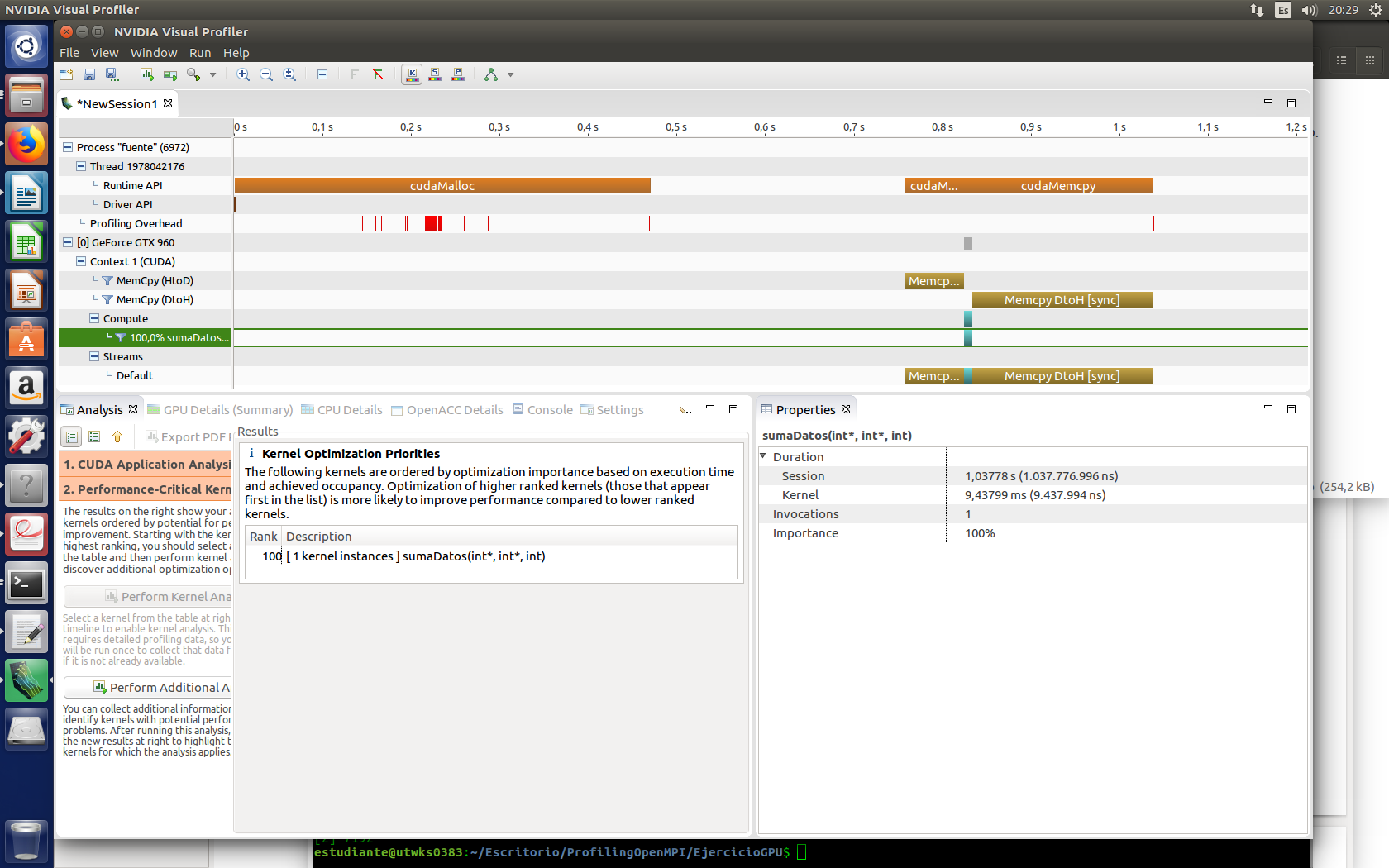
Tiempo de ejecucion del kernel es de 9,43799ms

La reserva de memoria en la GPU tiene un tiempo de 468.25772ms

Tiempo total de ejecucion es de 1,03778s







## Profiling de una aplicación para cluster

Por último se desarrollará una aplicación para cluster usando la librería MPI. Ésta aplicación consistirá en una multiplicación de matrices en paralelo. Se proporciona código al final del documento para su implementación. El alumno deberá medir el tiempo empleado en el envío y recepción de datos, deberá razonar el ancho de banda efectivo de la red usada, y por último se indicará el porcentaje de tiempo que la aplicación ha invertido en el envío/recepción de datos.

Solución:

Ejemplo de ejecucion de la aplicacion:

Ejecucion Esclavo 1 - Tiempo recepcion datos desde Master: 0.051662

CPU!

Ejecucion Master - Tiempo envio datos a esclavo 1: 0.046699

Ejecucion Esclavo 2 - Tiempo recepcion datos desde Master: 0.098608

CPU!

Ejecucion Master - Tiempo envio datos a esclavo 2: 0.046651

Ejecucion Esclavo 3 - Tiempo recepcion datos desde Master: 0.145084

CPU!

Ejecucion Master - Tiempo envio datos a esclavo 3: 0.046573

Ejecucion Esclavo 1 - Envio resultado a Master: 0.011948

Ejecucion Esclavo 1: 0.870442

Ejecucion Master - Tiempo recepcion datos desde esclavo 1: 0.725127

Ejecucion Esclavo 2 - Envio resultado a Master: 0.011901

Ejecucion Esclavo 2: 0.913046

Ejecucion Master - Tiempo recepcion datos desde esclavo 2: 0.042534

Ejecucion Esclavo 3 - Envio resultado a Master: 0.011879

Ejecucion Esclavo 3: 0.955313

Ejecucion Master - Tiempo recepcion datos desde esclavo 3: 0.042372

Ejecucion Master: 0.955265

Las matrices empleadas tienen un tamaño de 1000x1000, siendo los datos de tipo int.

El número de esclavos utilizados, es decir, procesos a los que hay que enviar datos es 3. Esto significa que las matrices se han dividido en tres partes en el máster para que los esclavos realicen el trabajo. El total de datos que se envían desde el máster a los esclavos es:

MPI\_Send(&subMatrizFilas,1,MPI\_INT,slave, TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(&numColumnas,1,MPI\_INT,slave,TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(&numFilas,1,MPI\_INT,slave, TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(&numColumnas,1,MPI\_INT,slave, TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(mat1,subMatrizFilas\*numColumnas,MPI\_INT,slave, TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(mat2,numFilas\*numColumnas,MPI\_INT,slave, TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(&operacion,1,MPI\_INT,slave, TAG\_OPERACION,MPI\_COMM\_WORLD);

4 enteros indicando los tamaños de datos a utilizar

1 entero indicando la operación

333\*1000 enteros pertenecientes a una matriz

1000\*1000 enteros de la otra matriz

Esto hace un total de 1.333.338 enteros a enviar en la transferencia. El tiempo que tarda el máster en enviar a los esclavos es de:

0.046699

0.046651

0.046573

Aproximadamente la velocidad de transmisión es de 218,06Mbps (utilizando el tiempo intermedio de los tres disponibles).

Tiempo total dedicado a enviar y recibir en el máster:

Envío datos = 0.046699, 0.046651, 0.046573 = 0,139923

Recepción datos = 0.725127, 0.042534, 0.042372 = 0,810033

Tiempo total de envío y recepción = 0,949956

Tiempo total de ejecución del máster es de 0.955265

El tiempo de transmisión de datos supone el 99,44% de tiempo de ejecución del máster

Esclavo 1:

Envío y recepción de datos = 0.051662 + 0.011948 = 0,06361

Ejecucion Esclavo 1 = 0.870442

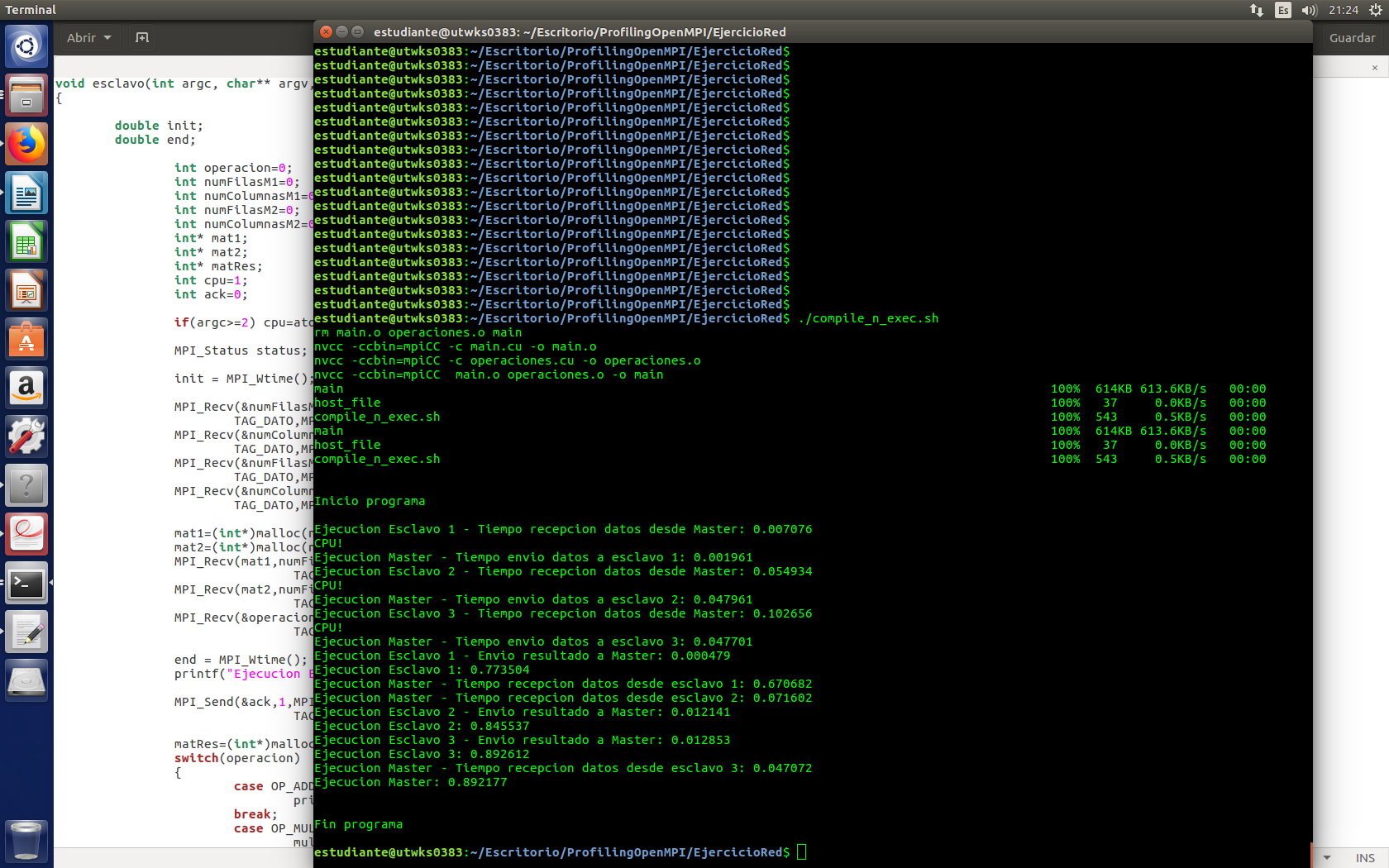
El tiempo de transmisión de datos supone el 7,31% de tiempo de ejecución del esclavo 1

Esclavo 2:

Envío y recepción de datos = 0.098608 + 0.011901 = 0,110509

Ejecucion Esclavo 2 = 0.913046

El tiempo de transmisión de datos supone el 12,1% de tiempo de ejecución del esclavo 2



# ANEXO

Código para testear GPROF

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <math.h>

**void** **pruebamalloc**(**int** y)

{

**int**\* x=(**int**\*)**malloc**(10\***sizeof**(**int**));

x[1000]=5;

}

**int** **fun3**(**int** x)

{

**int** i;

**for**(i=0;i<100;i++)

{

x++;

}

pruebamalloc(x);

**return** x;

}

**int** **fun2**(**int** x)

{

**int** i;

**for**(i=0;i<1000;i++)

{

x+= fun3(x);

}

**return** x;

}

**int** **fun1**(**int** x)

{

**int** i;

**for**(i=0;i<1000;i++)

{

x+=fun2(x);

x+=fun3(x);

}

**return** x;

}

**int** **main**()

{

fun1(0);

**return** 1;

}

Código para testear Nvidia Profiler

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <cuda.h>

**\_\_global\_\_** **void** **sumaDatos**(**int**\* in, **int**\* out, **int** size)

{

**int** IDx=blockIdx.x\*blockDim.x+threadIdx.x;

**if**(IDx>size) **return**;

out[IDx]=in[IDx]+in[IDx];

}

**int** **main**(**int** argc, **char** \*\*argv)

{

**int** datosCount=100000000;

**int**\* h\_datos=(**int**\*)**malloc**(datosCount\***sizeof**(**int**));

**int**\* h\_datosout=(**int**\*)**malloc**(datosCount\***sizeof**(**int**));

**int**\* d\_datos;

**int**\* d\_datosout;

cudaMalloc(&d\_datos,datosCount\***sizeof**(**int**));

cudaMalloc(&d\_datosout,datosCount\***sizeof**(**int**));

**for**(**int** i=0;i<datosCount;i++)

{

h\_datos[i]=i\*2;

}

**cudaMemcpy**(d\_datos,h\_datos,datosCount\***sizeof**(**int**),cudaMemcpyHostToDevice);

**int** numthreads=256;

**int** numbloques=datosCount/numthreads+1;

**sumaDatos**<<<numbloques,numthreads>>>(d\_datos,d\_datosout,datosCount);

**cudaMemcpy**(h\_datosout,d\_datosout,datosCount\***sizeof**(**int**),cudaMemcpyDeviceToHost);

**printf**("FIN\n");

**return** 0;

}

Código para testear la velocidad de la red del cluster (tres ficheros):

Fichero “main.c”

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include "operaciones.h"

#define TAG\_DATO 0

#define TAG\_OPERACION 1

#define OP\_ADD 0

#define OP\_MUL 1

void master(int argc, char\*\* argv, int rank, int nproc)

{

int numFilas=1000;

int numColumnas=1000;

int\* mat1=(int\*)malloc(sizeof(int)\*numFilas\*numColumnas);

int\* mat2=(int\*)malloc(sizeof(int)\*numFilas\*numColumnas);

int\* matRes=(int\*)malloc(sizeof(int)\*numFilas\*numColumnas);

int operacion=OP\_MUL;

MPI\_Status status;

int subMatrizFilas=numFilas/(nproc-1);

int resto=numFilas%(nproc-1);

subMatrizFilas++;

for(int i=0;i<numFilas;i++)

for(int j=0;j<numColumnas;j++)

{

mat1[i\*numFilas + j]=1;

mat2[i\*numFilas + j]=1;

}

int ack=0;

for(int slave=1;slave<nproc;slave++)

{

if((slave-1)==resto) subMatrizFilas --;

MPI\_Send(&subMatrizFilas,1,MPI\_INT,slave,

TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(&numColumnas,1,MPI\_INT,slave,TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(&numFilas,1,MPI\_INT,slave,

TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(&numColumnas,1,MPI\_INT,slave,

TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(mat1,subMatrizFilas\*numColumnas,MPI\_INT,slave,

TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(mat2,numFilas\*numColumnas,MPI\_INT,slave,

TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Send(&operacion,1,MPI\_INT,slave,

TAG\_OPERACION,MPI\_COMM\_WORLD);

MPI\_Recv(&(ack),1,MPI\_INT,slave,

TAG\_DATO,MPI\_COMM\_WORLD,&status);

}

subMatrizFilas++;

int indexCount=0;

for(int slave=1;slave<nproc;slave++)

{

if((slave-1)==resto) subMatrizFilas --;

MPI\_Recv(&(matRes[indexCount]),subMatrizFilas\*numColumnas,MPI\_INT,slave,

TAG\_DATO,MPI\_COMM\_WORLD,&status);

MPI\_Send(&ack,1,MPI\_INT,slave,

TAG\_DATO,MPI\_COMM\_WORLD);

indexCount+=subMatrizFilas\*numColumnas;

}

}

void esclavo(int argc, char\*\* argv, int rank, int nproc)

{

int operacion=0;

int numFilasM1=0;

int numColumnasM1=0;

int numFilasM2=0;

int numColumnasM2=0;

int\* mat1;

int\* mat2;

int\* matRes;

int cpu=1;

int ack=0;

if(argc>=2) cpu=atoi(argv[1]);

MPI\_Status status;

MPI\_Recv(&numFilasM1,1,MPI\_INT,0,

TAG\_DATO,MPI\_COMM\_WORLD,&status);

MPI\_Recv(&numColumnasM1,1,MPI\_INT,0,

TAG\_DATO,MPI\_COMM\_WORLD,&status);

MPI\_Recv(&numFilasM2,1,MPI\_INT,0,

TAG\_DATO,MPI\_COMM\_WORLD,&status);

MPI\_Recv(&numColumnasM2,1,MPI\_INT,0,

TAG\_DATO,MPI\_COMM\_WORLD,&status);

mat1=(int\*)malloc(numFilasM1\*numColumnasM1\*sizeof(int));

mat2=(int\*)malloc(numFilasM2\*numColumnasM2\*sizeof(int));

MPI\_Recv(mat1,numFilasM1\*numColumnasM1,MPI\_INT,0,

TAG\_DATO,MPI\_COMM\_WORLD,&status);

MPI\_Recv(mat2,numFilasM2\*numColumnasM2,MPI\_INT,0,

TAG\_DATO,MPI\_COMM\_WORLD,&status);

MPI\_Recv(&operacion,1,MPI\_INT,0,

TAG\_OPERACION,MPI\_COMM\_WORLD,&status);

MPI\_Send(&ack,1,MPI\_INT,0,

TAG\_DATO,MPI\_COMM\_WORLD);

matRes=(int\*)malloc(numFilasM1\*numColumnasM2\*sizeof(int));

switch(operacion)

{

case OP\_ADD:

printf("no implementada suma\n"); fflush(stdout);

break;

case OP\_MUL:

multiplicaMatrices

(cpu,mat1,mat2,matRes,numFilasM1,numColumnasM1,

numFilasM2,numColumnasM2);

break;

default:

printf("no implementada suma\n"); fflush(stdout);

break;

};

MPI\_Send(matRes,numFilasM1\*numColumnasM2,MPI\_INT,0,

TAG\_DATO,MPI\_COMM\_WORLD);

MPI\_Recv(&ack,1,MPI\_INT,0,

TAG\_DATO,MPI\_COMM\_WORLD,&status);

}

int main (int argc,char\*\* argv)

{

int rank;

int nproc;

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&nproc);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

switch(rank)

{

case 0:

master(argc,argv,rank, nproc);

break;

default:

esclavo(argc,argv,rank, nproc);

break;

};

MPI\_Finalize();

return 0;

}

Fichero “operaciones.h”

#ifndef \_OPERACIONES\_H\_

#define \_OPERACIONES\_H\_

int multiplicaVectores(int\* v1,int\* v2, int size);

void multiplicaMatrices(int cpu,int\* m1,int\* m2, int\* mRes,int numFilasM1, int numColumnasM1,int numFilasM2, int numColumnasM2);

void imprimeMatriz(int\* mat, int numFilas, int numColumnas);

#endif

Fichero operaciones.cu

#include <stdio.h>

#include <stdlib.h>

#include "operaciones.h"

\_\_global\_\_ void multiplicaVectores(int\* mres,int\* m1,int\* m2, int numFilas, int numColumnas)

{

int idx=blockIdx.x\*blockDim.x+threadIdx.x;

int numFila=idx/numColumnas;

int numColumna=idx%numColumnas;

mres[numFila\*numColumnas+numColumna]=0;

for(int i=0;i<numColumnas;i++)

{

mres[numFila\*numColumnas+numColumna]+=m1[numFila\*numColumnas+i]\*m2[numFila\*numColumnas+i];

}

}

void imprimeMatriz(int\* mat, int numFilas, int numColumnas)

{

for(int i=0;i<numFilas;i++)

{

for(int j=0;j<numColumnas;j++)

{

printf("%d,",mat[i\*numFilas+j]);

}

printf("\n");

}

}

int multiplicaVectores(int\* v1,int\* v2, int size)

{

int resultado=0;

for(int i=0;i<size;i++)

{

resultado+=v1[i]\*v2[i];

}

return resultado;

}

void multiplicaMatrices(int cpu,int\* m1,int\* m2, int\* mRes,int numFilasM1, int numColumnasM1,int numFilasM2, int numColumnasM2)

{

if(cpu){

printf("CPU!\n");

for(int i=0;i<numFilasM1;i++)

for(int j=0;j<numColumnasM2;j++)

mRes[i\*numColumnasM2+j]=multiplicaVectores(&(m1[i\*numColumnasM2]),

&(m2[j\*numColumnasM2])

,numColumnasM2);

}else{

printf("CUDA!\n");

int numThreadBloque=128;

int numBloques=(numFilasM1\*numColumnasM2/numThreadBloque)+1;

int\* d\_m1;

int\* d\_m2;

int\* d\_mres;

cudaMalloc((void\*\*)&d\_m1,sizeof(int)\*numFilasM1\*numColumnasM1);

cudaMalloc((void\*\*)&d\_m2,sizeof(int)\*numFilasM2\*numColumnasM2);

cudaMalloc((void\*\*)&d\_mres,sizeof(int)\*numFilasM1\*numColumnasM2);

cudaMemcpy(d\_m1,m1,sizeof(int)\*numFilasM1\*numColumnasM1,cudaMemcpyHostToDevice);

cudaMemcpy(d\_m2,m2,sizeof(int)\*numFilasM2\*numColumnasM2,cudaMemcpyHostToDevice);

multiplicaVectores<<<numThreadBloque,numBloques>>>(d\_mres,d\_m1,d\_m2,numFilasM1, numColumnasM2);

cudaMemcpy(mRes,d\_mres,sizeof(int)\*numFilasM1\*numColumnasM2,cudaMemcpyDeviceToHost);

}

}