# Ejercicio de Profiling

El alumno deberá realizar una memoria explicativa aplicando las técnicas vistas en clase. El ejercicio consistirá en una implementación de un algoritmo de multiplicación de matrices de gran dimensión, en el que se aplicarán las siguientes técnicas: Profiling de cpu, Profiling de errores de memoria, Profiling en GPU y Profiling en cluster.

La memoria entregada se estructurará como una guía de la práctica, en la que cada una de las secciones mostrará los pasos seguidos para realizar el perfilado y optimización del código entregado. La memoria entregada contendrá los siguientes capítulos:

### 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.

### 

### 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.

### 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)

## 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.

# Entrega de la práctica

La entrega se realizará a través de blackboard de la asignatura. Se entregará un archivo zip que contendrá el nombre del alumno, el código fuente analizado y la memoria pedida en formato PDF/Docx. Para cualquier duda, ponerse en contacto con el profesor Marcos Novalbos (marcos.novalbos@live.u-tad.com)

# 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);

}

}