Ejercicios de algoritmos paralelos

Nombre:

Christian Flores Meléndez.

Wilbert Marroquín Caceres

3.1 Suma de matrices

a)

```
void matrixAdd(float* A, float* B, float* C, int n) {
int size = n * n * sizeof(float);
float *d_A, *d_B, *d_C;
 cudaMalloc((void **) &d_A, size);
 cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
cudaMalloc((void **) &d_B, size);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
cudaMalloc((void **) &d_C, size);
matrixAddKernel<< ceil((n*n)/256.0), 256>> (d_A, d_B, d_C, n*n);
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

b)

```
__global__
 void matrixAddKernel(float* A, float* B, float* C, int n){
int i = threadIdx.x + blockDim.x * blockIdx.x;
if(i<n)  C[i] = A[i] + B[i];
}
```

c)

```
__global__
void matrixAddKernel(float* A, float* B, float* C, int n){
int i = threadIdx.x + blockDim.x * blockIdx.x;
```

```
if(i<n)
        for(int j = i * n; j < i * n + n ; j++)
                C[j] = A[j] + B[j];
}
```

d)

```
__global__
void matrixAddKernel(float* A, float* B, float* C, int n){
int i = threadIdx.x + blockDim.x * blockIdx.x;
if(i<n)
        for(int j = i ; j < n*n ; j+=n)
                C[j] = A[j] + B[j];
}
```

e)

El diseño b es mejor para gpu grandes ya que se puede distribuir completamente las tareas entre los threads, en el caso del diseño c y d es peor, pues estamos sobrecargando de tareas a cada thread. Estos últimos se desenvuelven mejor en gpu más pequeñas. La diferencia entre el diseño d y el c es el acceso a memoria. El diseño d tendrá mayor cantidad de cache miss que el diseño c, lo cual lo hace más ineficiente.

3.2 Multiplicación de matrices

```
void vecMult(float* A, float* B, float* C, int n) {
        int size = n * n * sizeof(float);
        int sizevect = n * sizeof(float);
        float *d_A, *d_B, *d_C;
        cudaMalloc((void **) &d_A, size);
        cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_B, sizevect);
        cudaMemcpy(d_B, B, sizevect, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_C, sizevect);
        vecMultKernel<< ceil((n*n)/256.0), 256>> (d_A, d_B, d_C, n);
```

```
        cudaMemcpy(C, d_C, sizevect, cudaMemcpyDeviceToHost);
        cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
__global__
 void vecMultKernel(float* A, float* B, float* C, int n){
int i = threadIdx.x + blockDim.x * blockIdx.x;
if(i<n*n){
        C[i]=0;
        for(int j=0;j<n;j++)
                C[i] += A[i*n+j] + B[j];
          }
}
```

3.3 Un nuevo pasante de verano se siente frustrado con CUDA. Él ha estado quejando de que CUDA es muy tedioso: Él tiene que declarar muchas de las funciones que tiene previsto ejecutar tanto en el host y el dispositivo dos veces, una como una función de host y otra como una función del dispositivo. ¿Cuál es tu respuesta?

Mi respuesta es que use tanto __host__ y __device__ en la declaración de la función, de esta forma el compilador genera dos versiones de la función, una para el dispositivo y otro para el host.

3.4 Complete las Partes 1 y 2 de la función en la Figura 3.5.

```
void vecAdd(float* A, float* B, float* C, int n) {
        int size = n * sizeof(float);
        float *d_A, *d_B, *d_C;
        //Parte 1
         cudaMalloc((void **) &d_A, size);
         cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_B, size);
        cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_C, size);
        //Parte 2
```

```
    vecAddKernel<< ceil(n/256.0), 256>> (d_A, d_B, d_C, n);
}
```

3.5 Si tenemos que utilizar cada hilo para calcular un elemento de salida de una suma de vectores, cuál sería la expresión para el mapeo de los índices de hilo / bloque para indexar los datos:

(C) i = blockIdx.x*blockDim.x + threadIdx.x;

3.6 Queremos usar cada hilo para calcular dos elementos de una suma de vectores (adyacentes), se supone que la variable i debería ser el índice del primer elemento a ser procesado por un hilo. ¿Cuál sería la expresión para el mapeo de los índices hilo / bloque para indexar los datos?

(A) i = blockIdx.x*blockDim.x + threadIdx.x + 2;

3.7 Para obtener una suma de vectores, asuma que la longitud del vector es 2000, cada thread calcula un elemento de salida, y el tamaño de bloque del hilo es de 512 threads.¿Cuántos threads estarán en el grid?

(C) 2048