Universidad Nacional de San Antonio Abad del Cusco

Escuela Profesional de Ingeniería informática y de Sistemas

Materia: Algoritmos Paralelos y Distribuidos

Profesor: Ray Dueñas Jiménez

Octavo Laboratorio de Algoritmos Paralelos y Distribuidos OpenMP

Buenas prácticas en CUDA

Alumno: Manuel Fernández Mercado

Periodo: 2025-II

Objetivo

Al finalizar esta sesión, el estudiante será capaz de:

1. **Implementar** mecanismos de manejo de errores automáticos para detectar fallos silenciosos en la GPU.
2. **Diferenciar** entre temporizadores de CPU (Wall-clock) y eventos de hardware de GPU (cudaEvent).
3. **Calcular** correctamente las dimensiones de la rejilla (Grid) y bloques para procesar arrays de tamaño arbitrario.
4. **Analizar** el Ancho de Banda Efectivo (Effective Bandwidth) de su kernel.

Código Base:

```
#include <cuda_runtime.h>

#include <stdio.h>

#include <sys/time.h>

// --- MACRO DE MANEJO DE ERRORES ---

// Intercepta cualquier llamada a CUDA que no retorne "Success"

#define CHECK(call)

{

    const cudaError_t error = call;

    if (error != cudaSuccess)

    {

        printf("Error: %s:%d, ", __FILE__, __LINE__);

        printf("code:%d,         reason:         %s\n",         error,
cudaGetErrorString(error));

        exit(1);

    }

}


// Verifica que los resultados de CPU y GPU sean idénticos

void checkResult(float *hostRef, float *gpuRef, const int N) {
```

```
    double epsilon = 1.0E-8;
    bool match = 1;
    for (int i=0; i<N; i++) {
        if (abs(hostRef[i] - gpuRef[i]) > epsilon) {
            match = 0;
            printf("Arrays do not match!\n");
            printf("host %5.2f gpu %5.2f at current %d\n", hostRef[i],
gpuRef[i], i);
            break;
        }
    }
    if (match) printf("Arrays match.\n\n");
}

void initialData(float *ip, int size) {
    time_t t;
    srand((unsigned) time(&t));
    for (int i=0; i<size; i++) {
        ip[i] = (float)( rand() & 0xFF )/10.0f;
    }
}

void sumArraysOnHost(float *A, float *B, float *C, const int N) {
    for (int idx=0; idx<N; idx++)
        C[idx] = A[idx] + B[idx];
}

// Kernel de Suma
```

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

int main(int argc, char **argv) {
    printf("%s Starting...\n", argv[0]);

    // 1. Configurar Dispositivo
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // 2. Configurar tamaño de datos (16 Millones de elementos)
    int nElem = 1<<24;
    printf("Vector size %d\n", nElem);

    // 3. Malloc en Host
    size_t nBytes = nElem * sizeof(float);
    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A     = (float *)malloc(nBytes);
    h_B     = (float *)malloc(nBytes);
    hostRef = (float *)malloc(nBytes);
    gpuRef  = (float *)malloc(nBytes);
```

```
// Inicializar datos
initialData(h_A, nElem);
initialData(h_B, nElem);
memset(hostRef, 0, nBytes);
memset(gpuRef, 0, nBytes);

// Suma en Host (Golden Standard)
sumArraysOnHost(h_A, h_B, hostRef, nElem);

// 4. Malloc en Device
float *d_A, *d_B, *d_C;
CHECK(cudaMalloc((float**)&d_A, nBytes));
CHECK(cudaMalloc((float**)&d_B, nBytes));
CHECK(cudaMalloc((float**)&d_C, nBytes));

// Transferencia Host -> Device
CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice));

// 5. Configuración de ejecución (Grid y Block)
int iLen = 1024;
dim3 block (iLen);
dim3 grid ((nElem+block.x-1)/block.x);

// --- MEDICIÓN CON CUDA EVENTS ---
cudaEvent_t start, stop;
CHECK(cudaEventCreate(&start));
CHECK(cudaEventCreate(&stop));
```

```
    // Grabar evento de inicio
    CHECK(cudaEventRecord(start));

    // Lanzar Kernel
    sumArraysOnGPU <<<grid, block>>>(d_A, d_B, d_C, nElem);

    // Grabar evento de final
    CHECK(cudaEventRecord(stop));

    // Esperar a que la GPU llegue a la marca 'stop'
    CHECK(cudaEventSynchronize(stop));

    float milliseconds = 0;
    CHECK(cudaEventElapsedTime(&milliseconds, start, stop));

    printf("sumArraysOnGPU <<<%d,%d>>> Time elapsed %f ms\n", grid.x,
block.x, milliseconds);
    // -------------------------

    // Verificar si hubo errores DENTRO del kernel
    CHECK(cudaGetLastError());

    // Copiar resultados de vuelta
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    // Validar corrección numérica
    checkResult(hostRef, gpuRef, nElem);
```

```
    // Liberar memoria y eventos
    CHECK(cudaFree(d_A)); CHECK(cudaFree(d_B)); CHECK(cudaFree(d_C));
    CHECK(cudaEventDestroy(start)); CHECK(cudaEventDestroy(stop));

    free(h_A); free(h_B); free(hostRef); free(gpuRef);
    return(0);
}
```

Actividad 1: Provocando un error

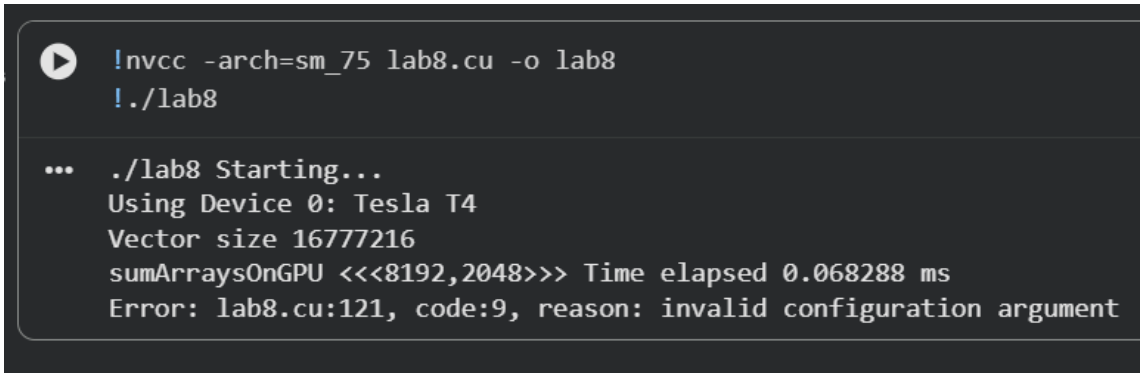
Modifica la línea `int iLen = 1024; a int iLen = 2048;`

- Compila y ejecuta.

Pregunta: ¿Qué mensaje muestra la consola? ¿Gracias a qué parte del código apareció este mensaje descriptivo en lugar de un simple fallo silencioso?

El mensaje que se muestra es: `Error: lab8.cu:127, code:9, reason: invalid configuration argument`

Esto es gracias a la macro CHECK, dado un llamado a una función devuelve el código de error asociado en caso de existir.



Actividad 2: Ancho de banda

Comparación: Busca el ancho de banda teórico de la memoria GDDR6 de la T4 ¿Qué tan lejos estas del máximo teórico?

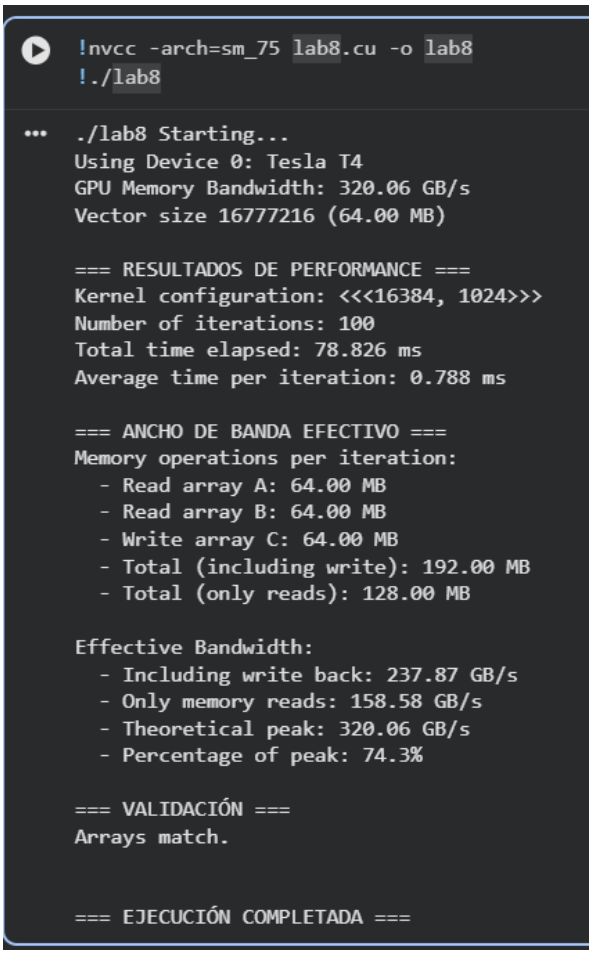
El ancho de banda máximo es de 320 GB/s

La salida del programa fue la siguiente:

```
Using Device 0: Tesla T4

GPU Memory Bandwidth: 320.06 GB/s

Vector size 16777216 (64.00 MB)


=== RESULTADOS DE PERFORMANCE ===

Kernel configuration: <<<16384, 1024>>>

Number of iterations: 100

Total time elapsed: 78.826 ms

Average time per iteration: 0.788 ms


=== ANCHO DE BANDA EFECTIVO ===

Memory operations per iteration:

  - Read array A: 64.00 MB

  - Read array B: 64.00 MB

  - Write array C: 64.00 MB

  - Total (including write): 192.00 MB

  - Total (only reads): 128.00 MB


Effective Bandwidth:

  - Including write back: 237.87 GB/s

  - Only memory reads: 158.58 GB/s

  - Theoretical peak: 320.06 GB/s
```

```
  - Percentage of peak: 74.3%


=== VALIDACIÓN ===

Arrays match.



=== EJECUCIÓN COMPLETADA ===
```

Solo se usó el 74.3% del ancho de banda.



Actividad 3: sincronización

Comenta la línea `CHECK(cudaEventSynchronize(stop));` y vuelve a ejecutar.

- **Pregunta:** ¿El tiempo medido cambió drásticamente o dio error? ¿Por qué es necesario esperar al evento stop antes de pedir el tiempo transcurrido?

Con la sincronización:

```
!nvcc -arch=sm_75 lab8.cu -o lab8
!./lab8

./lab8 Starting...
Using Device 0: Tesla T4
Vector size 16777216
sumArraysOnGPU <<<16384,1024>>> Time elapsed 0.914400 ms
Arrays match.
```

Sin la sincronización:

```
!nvcc -arch=sm_75 lab8.cu -o lab8
!./lab8

./lab8 Starting...
Using Device 0: Tesla T4
Vector size 16777216
Error: lab8.cu:115, code:600, reason: device not ready
```

Ocurrió un error al momento de tratar de ejecutar la línea `CHECK(cudaEventElapsedTime(&milliseconds, start, stop));`

Debido a que la CPU no está esperando a la GPU y se está usando la macro para capturar errores. Significa que al momento de querer medir el tiempo la GPU aún estaba ocupada.

Actividad 4: Desafio de análisis- La ilusión del tiempo (CPU vs GPU)

Ejecuta el programa y anota los dos resultados finales.

- *Ejemplo típico:* CPU = 0.000951 sec, GPU = 0.878304 ms.

1. **Conversión de Unidades:**
    - El timer del CPU está en segundos. El timer de GPU está en milisegundos.
    - Convierte el tiempo del CPU a milisegundos ($segundos \times 1000$).
2. **Cálculo del "Costo Invisible":**
    - Resta: $\text{Tiempo CPU (ms)} - \text{Tiempo GPU (ms)}$.
    - Deberías obtener un número pequeño positivo (aprox. 0.05 a 0.08 ms).
3. **Pregunta Final:**
    - Si el kernel (la suma matemática) tardó exactamente lo que dice el Timer GPU, **¿en qué se gastó esa diferencia de tiempo extra que reporta el CPU?** *Pista:* Considera qué tiene que hacer el CPU con el driver de NVIDIA antes de que el kernel pueda empezar a correr. (Concepto clave: *Kernel Launch Overhead*).

      El tiempo resultante es debido al overhead de comunicación. En un resumen, cuando se ejecuta el kernel esta instrucción pasa por el driver de nvidia y el puerto PCI lo que genera un pequeño retraso.