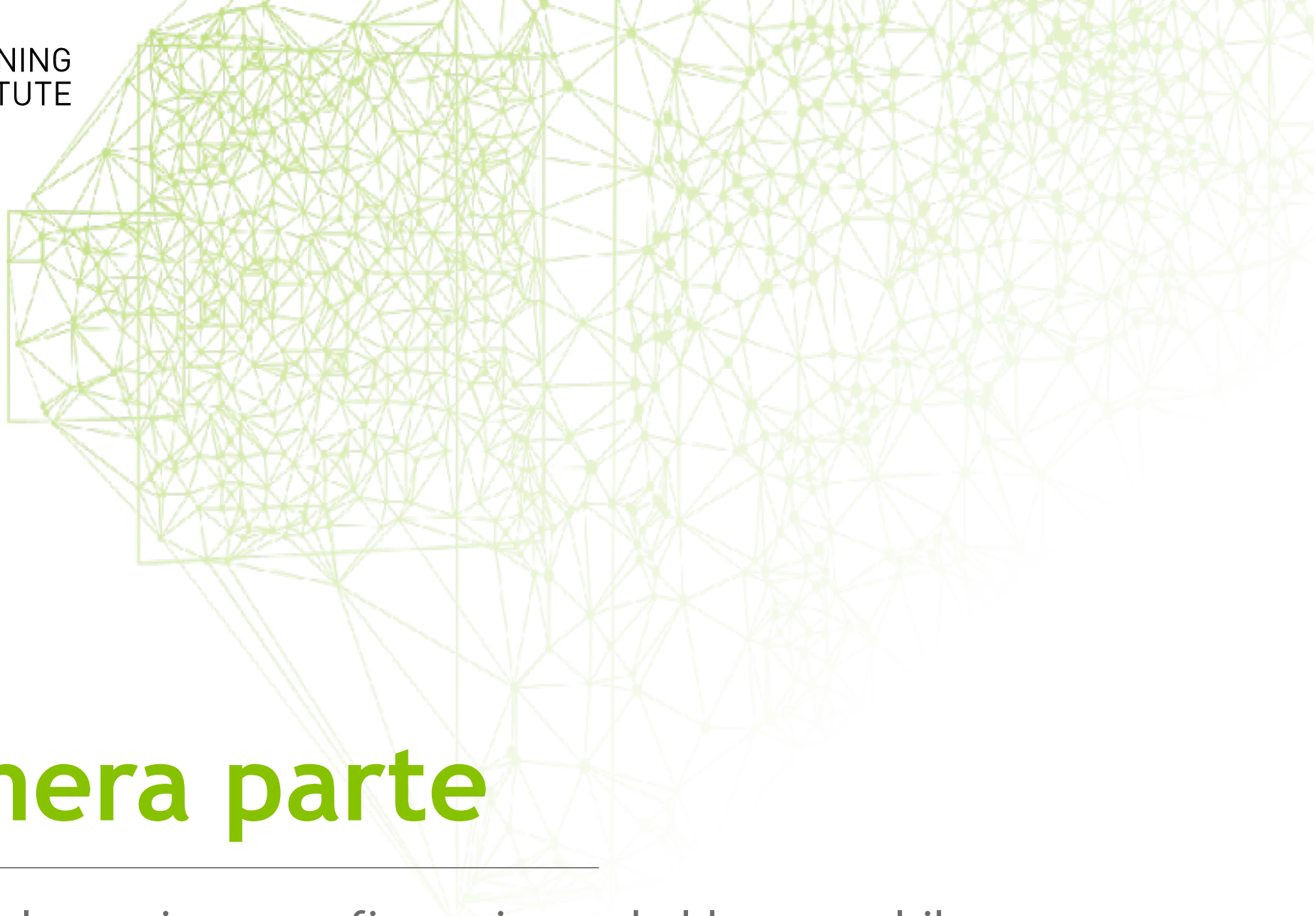
Ejercicios del segundo módulo del DLI

Pertenecientes al workshop
“Fundamentals of Accelerated Computing with CUDA C/C++”



Primera parte

Investigar las mejores configuraciones de bloques e hilos utilizando el Command Line Profiler y el modelo de GPU

Toma de contacto con el profiler `nsys`

- Ejercicio 1: Familiarízate con el caudal de información que proporciona esta herramienta, sobre todo estadísticas de uso referentes a:
  - Las directivas CUDA.
  - El tiempo de ejecución para los kernels.
  - El tiempo y tamaño de las operaciones de acceso a memoria.
  - Las funciones en tiempo de ejecución del Sistema Operativo.
- Responde a las siguientes cuestiones:
  - ¿Cómo se llama el kernel que se usa en este ejercicio?
  - ¿Cuántas veces se ejecutó dicho kernel?
  - ¿Cuánto tiempo tardó en ejecutarse dicho kernel?

Toma de contacto con el profiler `nsys` (cont.)

- Ejercicio 2: Lanza `addVectorsInto` para muchos bloques e hilos y comprueba cómo afecta a la aceleración del kernel.
- Ejercicio 3: Prueba varias configuraciones para:
  - El tamaño de la malla o número de bloques.
  - El tamaño del bloque o número de hilos por bloque.
  - Sugerencia: Dado que el tamaño del vector es 2^{25}, buenas opciones dentro de los límites de CUDA Compute Capabilities son:
    - <<<2^{15},1024>>>, <<<2^{16},512>>>, <<<2^{17},256>>>, <<<2^{18},128>>>, <<<2^{19},64>>>
- Intenta adivinar qué combinación es la mejor, o analizar por qué lo es cuando obtengas los resultados de ejecución.
- Puedes afinar más la elección óptima ayudándote del CUDA Occupancy Calculator.

Consulta los parámetros HW de tu GPU para ponerlos al servicio de tus optimizaciones

- Tenemos varias formas de hacerlo:

1. Utiliza el comando `!nvidia-smi` que aprendiste en el módulo 1 (aprovecha para practicar los *checkpoints* de *Jupyter notebooks*).
2. Usa el programa `deviceQuery` del SDK de CUDA, cuya salida es:

```
There are 4 devices supporting CUDA

Device 0: "GeForce GTX 480"
  CUDA Driver Version:                           4.0
  CUDA Runtime Version:                          4.0
  CUDA Capability Major revision number:         2
  CUDA Capability Minor revision number:         0
  Total amount of global memory:                 1609760768 bytes
  Number of multiprocessors:                     15
  Number of cores:                               480
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  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:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.40 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads can use this device simultaneously)
  Concurrent kernel execution:                   Yes
  Device has ECC support enabled:                No
```

3. Aprovecha la llamada a `cudaGetDeviceProperties()` - a cont.

Uso del soporte en tiempo de ejecución para conocer los recursos hardware disponibles

- Cada GPU disponible tiene asignado un número 0, 1, 2, …
- Para conocer el número de GPUs disponibles:
  - `cudaGetDeviceCount (int* count);`
- Para conocer en qué GPU estamos ejecutando el código:
  - `cudaGetDevice(int* dev);`
- Para seleccionar una GPU concreta:
  - `cudaSetDevice(int dev);`
- Para consultar los parámetros CUDA de nuestra GPU:
  - `cudaGetDeviceProperties(cudaDeviceProp* prop, int dev);`
- Para conocer la GPU que mejor reúne ciertos requisitos:
  - `cudaChooseDevice(int* dev, const cudaDeviceProp* prop);`

Aprovecha la llamada a cudaGetDeviceProperties()

```
int main()
{
  int deviceId;  // Device ID is required first to query the device
  cudaGetDevice(&deviceId);
  cudaDeviceProp props;
  cudaGetDeviceProperties(&props, deviceId); // “props” now contains HW properties

  printf(" Device ID is %d\n There are %d multiprocessors (SMs:)\n
           Max. block size: %d threads\n CUDA Compute Capability: %d.%d\n
           Warp size: %d\n", deviceId, props.multiProcessorCount,
           props.maxThreadsPerBlock, props.major, props.minor, props.warpSize);
  printf(" GPU model: %s\n Clock frequency: %d KHz\n
           Global memory size: %ld bytes\n Clock frequency for memory: %d KHz\n
           Shared memory per block: %ld bytes\n SM registers per block: %d\n”,
           props.name, props.clockRate, props.totalGlobalMem,
           props.memoryClockRate, props.sharedMemPerBlock, props.regsPerBlock);
}
```

- Ejercicio 4 (Query the device): Juega con estas llamadas para consultar los parámetros HW de tu GPU que te interesen.

Demuestra tu conocimiento del hardware y el software de CUDA

- Ejercicio 5: Aprovecha la información que has obtenido en el ejercicio 4 para encontrar la configuración de bloques e hilos/bloque más eficiente para tu kernel `addVectorsInto` según la GPU que te asignaron hoy.

Segunda parte

Analizar el comportamiento de la memoria y minimizar las faltas de página cuando se usa conjuntamente desde la CPU y la GPU

Investiga el movimiento de las páginas en memoria unificada

```
__global__ void deviceKernel(int *a, int N);
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i=idx; i<N; i += stride)
    a[i] = 1;
}
```

```
void hostFunction(int *a, int N)
{
  for (int i=0; i<N, i++)
    a[i] = 1;
}
```

```
int main()
{
  int N = 2<<24;                    // The problem size
  size_t size = N * sizeof(int);    // The memory size (in bytes)
  int *a;
  cudaMallocManaged(&a, size);      // This array to be used jointly by CPU and GPU
  // Call here to CPU functions and/or GPU kernels to analyse page faults with nsys
  deviceKernel<<<256,256>>>(a,N);                    hostFunction(a, N);
                                  cudaDeviceSynchronize();
  cudaFree(a);
}
```

- Ejercicio 6 (Explore UM Migration and Page Faulting): Usa los datos desde la CPU y la GPU para ver cómo migran las páginas entre la memoria principal y la memoria de vídeo.

Investiga el movimiento de las páginas en memoria unificada (II)

```
__global__ void addVectorsInto(float …, int N);
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i=idx; i<N; i += stride)
    result[i] = a[i] + b[i];
}
```

```
void initWith(float *a, int N)
{
  for (int i=0; i<N, i++)
    a[i] = num;
}
void checkElementsAre(float …, int N)
{ }
```

```
int main()
{
  int N = 2<<24;                        // The problem size
  size_t size = N * sizeof(float);  // The memory size (in bytes)
  int *a;
  cudaMallocManaged(&a, size);      // These 3 arrays to be used jointly by CPU and GPU
  cudaMallocManaged(&b, size);
  cudaMallocManaged(&c, size);
  // Call here to CPU functions and/or GPU kernels to analyse page faults with nsys
  cudaFree(a);
}
```

- Ejercicio 7 (Revisit UM Behavior for Vector Add Program): Utiliza el kernel `addVectorsInto` junto a 2 funciones en CPU para analizar cómo afecta el uso de la memoria al rendimiento.

Investiga el movimiento de las páginas en memoria unificada (III)

```
__global__ void addVectorsInto(float …, int N);
{
}
__global__ initWith(float *a, int N)
{
}
```

```
void checkElementsAre(…, int N)
{
}
```

```
int main()
{
  int N = 2<<24;                      // The problem size
  size_t size = N * sizeof(float);  // The memory size (in bytes)
  int *a;
  cudaMallocManaged(&a, size);      // These 3 arrays to be used jointly by CPU and GPU
  cudaMallocManaged(&b, size);
  cudaMallocManaged(&c, size);
  // Call here to CPU functions and/or GPU kernels to analyse page faults with nsys
  cudaFree(a);
}
```

- Ejercicio 8 (Initialize Vector in Kernel): Transforma `initWith` en un segundo kernel que permita inicializar el vector en la GPU y paralelizar este proceso. Analiza la mejora en tiempo y uso de la memoria utilizando `nsys`.

Utiliza llamadas a `cudaMemPrefetchAsync(puntero, tamaño, dispositivo)`

- Es un recurso que permite **adelantar** la transferencia de datos para evitar las faltas de página por demanda.
- También suele agrupar las transferencias necesarias, lo que permite beneficiarse del gran ancho de banda de las comunicaciones.
- Ejercicio 9: Trata de precargar uno, dos o los tres vectores en la GPU **antes** de que los use, y analiza las mejoras.
- Ejercicio 10: Puedes precargar también el vector resultado en la CPU **antes** de que ésta valide los resultados correctos en su función `checkElementsAre`.

Soluciones de los ejercicios 7 al 10 (y adelanto de tareas para el módulo 3)

```
cudaMallocManaged(&a, size);
cudaMallocManaged(&b, size);
cudaMallocManaged(&c, size);
```

```
initWith(3, a, N);
initWith(4, b, N);
initWith(0, c, N);
```

Ejercicios 7, 9 y 10

Ejercicio 2 del Visual Profiler (módulo 3)

Ejercicio 8

```
initWith<<<numberOfBlocks, threadsPerBlock>>>(3, a, N);
initWith<<<numberOfBlocks, threadsPerBlock>>>(4, b, N);
initWith<<<numberOfBlocks, threadsPerBlock>>>(0, c, N);
cudaDeviceSynchronize();
```

```
cudaMemPrefetchAsync(a, size, deviceId);
cudaMemPrefetchAsync(b, size, deviceId);
cudaMemPrefetchAsync(c, size, deviceId);
addArraysInto<<<numberOfBlocks, threadsPerBlock>>>(c, a, b, N);
```

Ejercicio 9 Ejercicio 1 del Visual Profiler

```
cudaMemPrefetchAsync(c, size, cudaCpuDeviceId);
checkElementsAre(7, c, N);
```

Ejercicio 10 Ejercicio 3 del Visual Profiler

```
cudaFree(a);
cudaFree(b);
cudaFree(c);
```

Síntesis final del módulo 2

- Utilizamos el profiler `nsys` en la línea de comandos para analizar un kernel y encontrar vías de optimización.
- Aprovechamos nuestro conocimiento del hardware para acelerar la ejecución de un kernel.
- Declaramos memoria que pueden compartir la CPU y la GPU, analizamos las faltas de página y activamos los mecanismos de sincronización que garantizan el uso correcto de los datos en paralelismo biprocesador.
- Precargamos áreas de memoria para reducir el número de faltas de página anticipándonos a su uso.
- Empleamos un ciclo de desarrollo iterativo que nos permite desplegar aplicaciones y optimizarlas en GPU.

Ejercicio final: Optimiza iterativamente un código SAXPY acelerado en GPU

Programa C en CPU

```
#define N 2048 * 2048
void saxpy(int *a, int *b, int *c)
{
        for (int idx = 0; idx<N; idx++)
                c[idx] = 2 * a[idx] + b[idx];
}



void main()
{
    .....
        saxpy(a, b, c);
}
```

Paralelización con CUDA

```
#define N 2048 * 2048
__global__ void saxpy(int *a, int *b, int *c)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if (idx < N)
                c[idx] = 2 * a[idx] + b[idx];
}

void main()
{
    < aloja la memoria unificada: cudaMallocManaged >
    < precarga los vectores en memoria si es necesario >
    < inicializa los vectores: puedes elegir CPU o GPU >
    < precarga los vectores en la memoria de la GPU >
    < elige un buen nº de bloques/malla e hilos/bloque >
        saxpy<<<dimGrid, dimBlock>>>(a, b, c);
    < libera la memoria >
}
```