Exercises from second DLI module

Workshop
“Fundamentals of Accelerated Computing with CUDA C/C++”

Part 1

Find the best configuration execution (blocks and threads per block) using the Command Line Profiler + the HW info from your GPU model

Get familiar with `nsys` profiler

- Exercise 1: Get familiar with all the information provided by this tool, mainly statistics with respect to:
  - CUDA directives.
  - Execution time for the CUDA kernels.
  - Time required and size of memory access operations.
  - Run-time functions executed by the Operating Systems.
- Answer the following questions:
  - What is the name of the kernel used along this exercise?
  - How many times was the kernel executed?
  - How much time did the kernel take to be executed?

Get familiar with `nsys` profiler (cont.)

- Exercise 2: Launch `addVectorsInto` for many blocks and threads, and check how is affecting the kernel acceleration.
- Exercise 3: Try several execution configurations for:
  - The grid size (number of blocks).
  - The block size (number of threads per block).
  - Hint: Given that the vector size is 2^{25}, good choices within the limits of CUDA Compute Capabilities are:
    - <<<2^{15},1024>>>, <<<2^{16},512>>>, <<<2^{17},256>>>, <<<2^{18},128>>>, <<<2^{19},64>>>
- Try to figure out the best choice, or analyze why is the best once you get the execution results.
- The CUDA Occupancy Calculator tool may give you some good advices.

Query the parameters of your GPU to enable further optimizations

- There are several ways to do it

1. Use the `!nvidia-smi` command learnt in module 1 (take advantage to practice the checkpoints of your Jupyter notebook).

2. Use the `deviceQuery` program included in the CUDA SDK, with the following output:

```
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. Take advantage of `cudaGetDeviceProperties()` - see next.

Use the CUDA runtime support to know the hardware resources available

- There is a number assigned to each GPU available: 0, 1, …
- To query the total number of GPUs available, use:
  - `cudaGetDeviceCount (int* count);`
- To query on which GPU we are executing our code:
  - `cudaGetDevice(int* dev);`
- To choose a particular GPU to execute your kernel:
  - `cudaSetDevice(int dev);`
- To query the CUDA parameters of our GPU:
  - `cudaGetDeviceProperties(cudaDeviceProp* prop, int dev);`
- To know the GPU that better fulfills certain requirements:
  - `cudaChooseDevice(int* dev, const cudaDeviceProp* prop);`

Take advantage of cudaGetDeviceProperties()

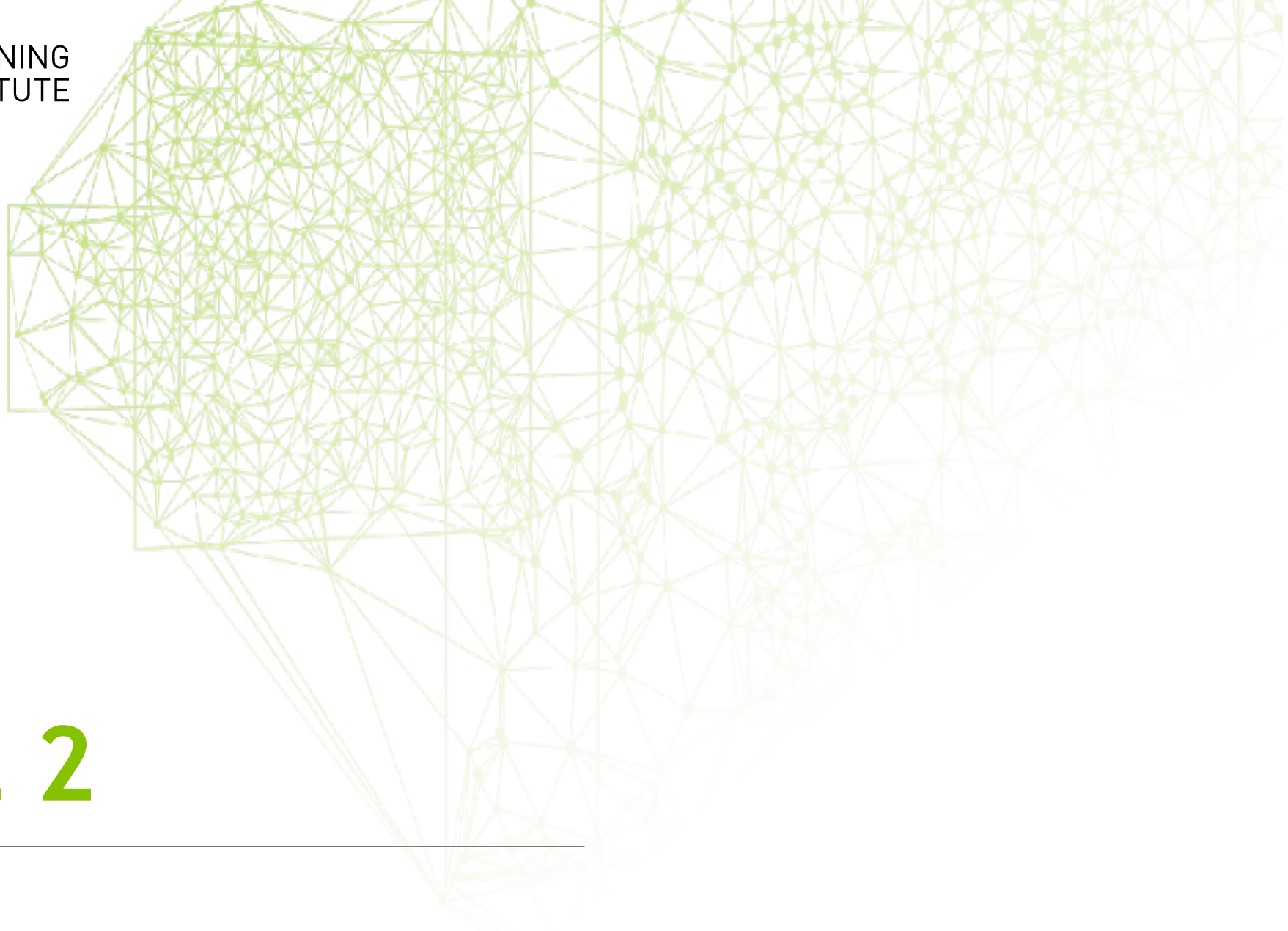
```
int main()
{
  int deviceId;  // Device ID is required first to query the device
  cudaGetDevice(&deviceId);
  cudaDeviceProp props;
  cudaGetDeviceProperties(&props, deviceId); // “props” now contains HW properties

  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);
}
```

- Exercise 4 (Query the device): Play with these calls to query those GPU parameters you might be interested in.

Prove your GPU knowledge

- Exercise 5: Take advantage of the information you got in exercise 4 to figure out the best execution configuration for your `addVectorsInto` kernel according to the GPU you were assigned in the cloud today.



Part 2

Analyze the memory behavior and minimize page faults when we use unified memory (shared by CPU and GPU)

Investigate page migration within unified memory (I)

```
__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);
}
```

- Exercise 6 (Explore UM Migration and Page Faulting): Use data from CPU and GPU to see how pages migrate between main and video memory.

Investigate page migration within unified memory (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);
}
```

- Exercise 7 (Revisit UM Behavior for Vector Add Program): Use the `addVectorsInto` kernel together with 2 CPU functions to analyze how the use of memory influences performance.

Investigate page migration within unified memory (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);
}
```

- Exercise 8 (Initialize Vector in Kernel): Transform `initWith` into another kernel which allows you to initialize the vector in the GPU and parallelize this process. Analyze with `nsys` the time reduction and the use of memory.

Use calls to
`cudaMemPrefetchAsync(pointer, size, device)`

- It is a resource to **anticipate** data transfers so that page faults may be avoided on demand.
- You can also group the required transfers in order to minimize start-ups in communications and benefit from a extraordinary bandwidth.
- Exercise 9: Try to prefect one, two or three vectors in the GPU **before** they are actually used, and analyze improvements.
- Exercise 10: You can also prefect the output vector in the CPU **before** it validates the correct results in `checkElementsAre`.

Summary of solutions from exercises 7 to 10 (and look ahead into tasks within module 3)

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

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

Exercise 7, 9 y 10

Exercise 2 using Visual Profiler (module 3)

Exercise 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);
```

Exercise 9

Exercise 1 using Visual Profiler

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

Exercise 10

Exercise 3 using Visual Profiler

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

Summary of DLI module 2

- We have used the `nsys` profiler in the command-line to analyze a kernel and find our way to perform optimizations.
- We have exploited our knowledge of the GPU hardware to accelerate a kernel execution.
- We declare unified memory to be shared between CPU and GPU, analyze page faults and activate synchronization mechanisms to guarantee the correct use of data on a concurrent CPU-GPU execution.
- We prefetch memory areas to reduce the number of page faults on demand.
- We use an iterative development cycle to deploy applications and optimize them on the GPU.

Final exercise: Optimize iteratively a SAXPY code accelerated on GPU

CPU version (C code)

```
#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);
}
```

Parallel version on the GPU (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()
{
    < allocate unified memory: cudaMallocManaged >
    < prefetch vectors in memory in case it is needed >
    < initialize vectors: you can choose in CPU or in GPU >
    < prefetch vectors in GPU memory >
    < choose a good configuration execution >
        saxpy<<<dimGrid, dimBlock>>>(a, b, c);
    < free memory >
}
```