Department of Computer Science and Engineering (Data Science)

High Performance Computing Laboratory (DJ19DSL802)

HPC Experiment 6

Bhuvi Ghosh
60009210191

Aim: Array Manipulation on both the Host and Device

Theory:

More recent versions of CUDA (version 6 and later) have made it easy to allocate memory that is available to both the CPU host and any number of GPU devices, and while there are many intermediate and advanced techniques for memory management that will support the most optimal performance in accelerated applications, the most basic CUDA memory management technique we will now cover supports fantastic performance gains over CPU-only applications with almost no developer overhead.

To allocate and free memory, and obtain a pointer that can be referenced in both host and device code, replace calls to malloc and free with cudaMallocManaged and cudaFree as in the following example:

```
// CPU-only

int N = 2<<20;

size_t size = N * sizeof(int);


int *a;

a = (int *)malloc(size);


// Use `a` in CPU-only program.

free(a);

// Accelerated

int N = 2<<20;

size_t size = N * sizeof(int);


int *a;
```

// Note the address of `a` is passed as first argument.

cudaMallocManaged(&a, size);

// Use `a` on the CPU and/or on any GPU in the accelerated system.

cudaFree(a);

More recent versions of CUDA (version 6 and later) introduced **Unified Memory**, which simplifies memory management in GPU-accelerated applications. Traditionally, CUDA required separate memory allocation on the **host (CPU)** and **device (GPU)**, followed by explicit data transfers using cudaMemcpy. However, with Unified Memory, a single allocation can be accessed by both the CPU and GPU without manual data copying.

cudaMallocManaged is the key function that enables Unified Memory. It allocates a shared memory region that is automatically managed by the CUDA runtime. This allows data to be used on the CPU and any GPU in the system without requiring explicit memory transfers. The allocated memory is freed using **cudaFree**, just like in traditional CUDA memory management.

This approach significantly reduces developer overhead while still delivering substantial performance improvements over CPU-only applications. By eliminating manual memory transfers, Unified Memory enables easier debugging and accelerates development while maintaining efficient GPU utilization.

How cudaMallocManaged Works

- Instead of using malloc for CPU memory allocation and cudaMalloc for GPU memory, developers can simply call:

cudaMallocManaged(&pointer, size);

- This function allocates **a unified memory block** that can be accessed by both the CPU and GPU.
- The memory remains accessible throughout execution, and CUDA automatically handles data migration between CPU and GPU.
- The allocated memory must be freed using cudaFree(pointer); once it is no longer needed.

Advantages of Unified Memory (cudaMallocManaged)

1. **Simplified Memory Management**
    - No need for separate memory allocation on the CPU and GPU.
    - Eliminates explicit memory copies (cudaMemcpy).
2. **Automatic Data Migration**

- CUDA automatically moves data between CPU and GPU when accessed, reducing the need for manual optimization.

3. **Easier Debugging and Development**
   - Since data is accessible in both CPU and GPU code, debugging becomes simpler.
   - Developers can write CUDA programs with less boilerplate code.

4. **Multi-GPU Support**
   - Unified Memory enables seamless access to allocated memory across multiple GPUs in the system.

5. **Better Resource Utilization**
   - Unified Memory allows efficient use of system memory when GPU memory is limited, improving performance for large datasets.

Limitations of Unified Memory

- **Performance Overhead:**
  - Although Unified Memory simplifies development, automatic data transfers can introduce latency if not optimized correctly.
- **Hardware Dependency:**
  - Unified Memory works best on **Pascal and newer** GPU architectures. Older GPUs may not fully support it.
- **Less Control Over Data Placement:**
  - Developers have less fine-grained control over where data is stored, which may lead to inefficiencies in some applications.

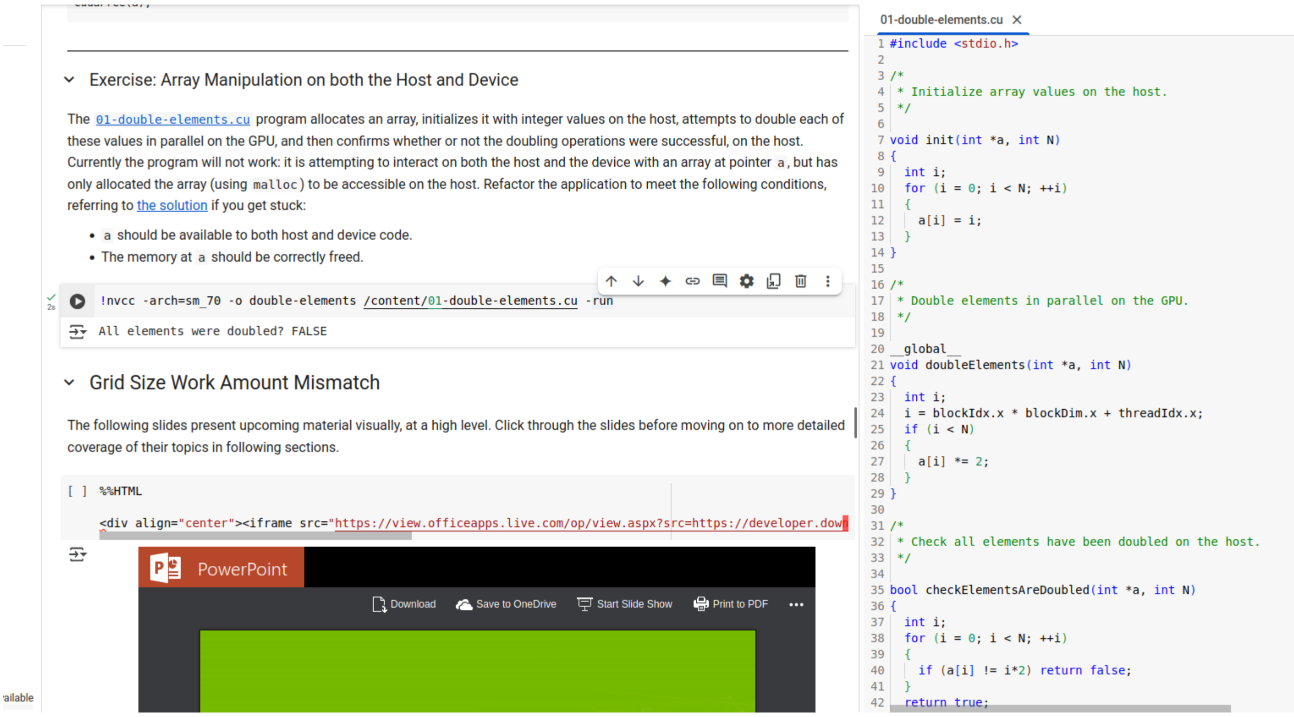
Department of Computer Science and Engineering (Data Science)

High Performance Computing Laboratory (DJ19DSL802)

Lab Experiment to be performed:

The 01-double-elements.cu program allocates an array, initializes it with integer values on the host, tries to double each of these values in parallel on the GPU, and then confirms if the doubling operations were successful, on the host. Currently the program will not work: it is attempting to interact on both the host and the device with an array at pointer a, but has only allocated the array (using malloc) to be accessible on the host. Refactor the application to meet the following conditions :

1. a should be available to both host and device code.
2. The memory at a should be correctly freed.


Exercise: Array Manipulation on both the Host and Device

The 01-double-elements.cu program allocates an array, initializes it with integer values on the host, attempts to double each of these values in parallel on the GPU, and then confirms whether or not the doubling operations were successful, on the host. Currently the program will not work: it is attempting to interact on both the host and the device with an array at pointer a, but has only allocated the array (using malloc) to be accessible on the host. Refactor the application to meet the following conditions, referring to the solution if you get stuck:

- a should be available to both host and device code.
- The memory at a should be correctly freed.

```
!nvcc -arch=sm_70 -o double-elements /content/01-double-elements.cu -run
All elements were doubled? FALSE
```

Grid Size Work Amount Mismatch

The following slides present upcoming material visually, at a high level. Click through the slides before moving on to more detailed coverage of their topics in following sections.

```
%%HTML
<div align="center"><iframe src="https://view.officeapps.live.com/op/view.aspx?src=https://developer.dow
```

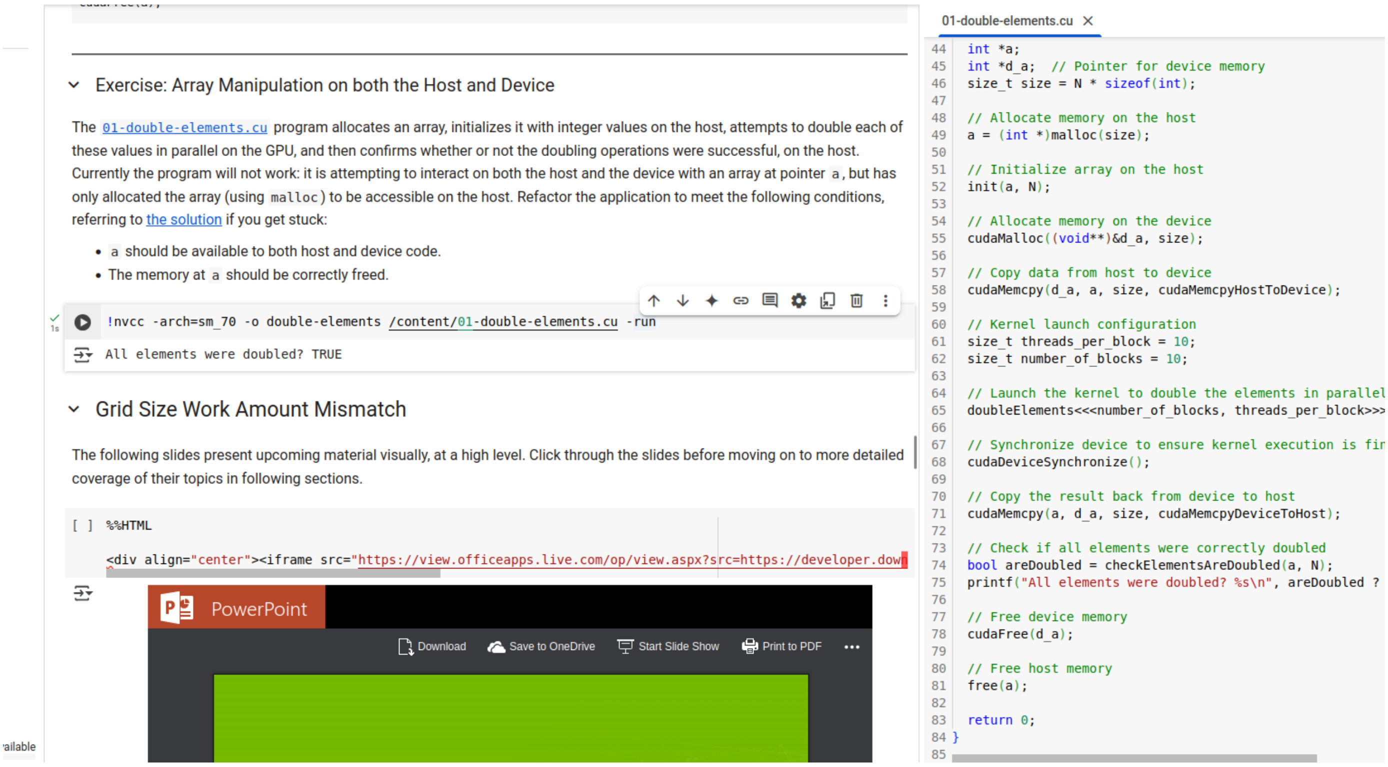
PowerPoint
Download Save to OneDrive Start Slide Show Print to PDF

01-double-elements.cu

```c
#include <stdio.h>

/*
 * Initialize array values on the host.
 */

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

/*
 * Double elements in parallel on the GPU.
 */

__global__
void doubleElements(int *a, int N)
{
  int i;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
  {
    a[i] *= 2;
  }
}

/*
 * Check all elements have been doubled on the host.
 */

bool checkElementsAreDoubled(int *a, int N)
{
  int i;
  for (i = 0; i < N; ++i)
  {
    if (a[i] != i*2) return false;
  }
  return true;
```

This program initializes an array on the host, then doubles its elements in parallel using a CUDA kernel. The `init` function sets each array element to its index. The `doubleElements` kernel runs on the GPU to multiply each element by The `checkElementsAreDoubled` function verifies that all elements were correctly doubled. The code allocates memory for the array on the host but lacks the necessary steps to transfer it to and from the GPU. After the kernel execution, memory is freed.

After Refactoring:



The refactored program allocates memory for the array `a` on both the host and the device. It initializes the array on the host, transfers it to the device using `cudaMemcpy`, and then doubles the elements in parallel using a CUDA kernel. After the kernel execution, the results are copied back from the device to the host. Finally, the memory is freed on both the host (using `free`) and the device (using `cudaFree`). This ensures proper memory management and allows interaction with the array on both the host and device.

Conclusion

CUDA Unified Memory, introduced in CUDA 6, revolutionized memory management by allowing seamless access to memory between the CPU and GPU using **cudaMallocManaged**. It eliminates the need for explicit memory copies, simplifying CUDA programming and reducing development time.However, while Unified Memory makes CUDA programming easier, developers must still be mindful of performance considerations, especially for applications requiring optimal memory management. By understanding when and how to use **cudaMallocManaged**, programmers can achieve a balance between ease of use and high performance in GPU-accelerated applications.