CUDA concept: GPU-CPU memory transfer

CUDA

- Was introduced in 2006 by NVIDIA (1)
- A general purpose parallel computing platform that leverages the engines in their GPUs specifically (2)
- GPUs provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. (3)
- GPU is specialized for highly parallel computations (4)
- Many applications utilize the GPU to run faster (5)
- In turn, CUDA has been employed to solve many complex computational problems in a more efficient way than on a CPU (6)

CPU and GPU are two separate processors; each having their own memory. There are several ways in
which data are transferred between processors which we will be discussing today

- Old method of transferring data between CPU and memory (aka memCUDA copy)
- Method that incorporates Unified memory introduced in CUDA 6
- Method that includes the Prefetching of data with memory advice
- Method with Data transfer or initialization as a CUDA kernel

For each method, we will discuss briefly its concepts, provide a programming demo, and report the results of its execution and transfer time

At the end, we will summarize these findings and analyze what these findings entail

Notes:

- Steps for transition:
    1) Declare that a 1-D convolution program will be used for the whole discussion.
    2) Display program implemented in C with timer.
    3) Declare that it is going to be programmed using CUDA.
        a) Code the program by only changing the 1-D convolution function into a `kernel` that can be run in the GPU by prefixing `__global__` to the 1-D convolution function.
        b) Still not sure about this, but the program would raise an error since the data are not transferred properly to its designated memory locations.
        c) Smoothly transition how data transferring is done using various ways.

GPU-CPU memory transfer

- Old method of transferring data between CPU and GPU memory
- Unified Memory in CUDA 6.0
- Prefetching of data with memory advising
- Data transfer or initialization as a CUDA kernel

Old method of transferring data between CPU and GPU memory

- Utilizes the provided CUDA functions for memory management and memory/data transfer: `cudaMalloc()`, `cudaFree()`, and `cudaMemcpy()`
- `cudaMalloc ( void** devPtr, size_t size )`
    - Allocate memory on the device (GPU).
    - Parameters:
        - `devPtr` - Pointer to allocated device memory
        - `size` - Requested allocation size in bytes
- `cudaFree ( void* devPtr )`
    - Frees memory on the device.
    - Parameters:
        - `devPtr` - Device pointer to memory to free
- `cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind )`
    - Copies data between host and device.
    - Parameters:
        - `dst` - Destination memory address
        - `src` - Source memory address
        - `count` - Size in bytes to copy
        - `kind` - Type of transfer
            - Values:
                - `cudaMemcpyHostToHost = 0` - Host -> Host
                - `cudaMemcpyHostToDevice = 1` - Host -> Device
                - `cudaMemcpyDeviceToHost = 2` - Device -> Host
                - `cudaMemcpyDeviceToDevice = 3` - Device -> Device
                - `cudaMemcpyDefault = 4` - Direction of the transfer is inferred from the pointer values. Requires unified virtual addressing

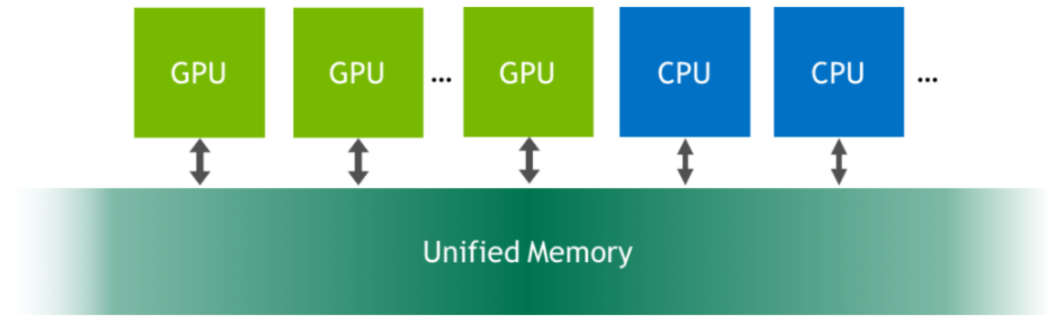
Notes:

- Sample code can be found in the `CUDA Tutorial (Vanilla / Not Unified Memory)`. Link can be found in the References section.
- First version of the program would still be using the loop inside the 1-D convolution CUDA kernel but when calling the said kernel the number of block and thread that would be requested should be both 1
- Second version of the program would be using the grid-stride loop approach.

- Point out the difference between the execution time of C only and CUDA using old method of data transfer (use nprof)

Unified Memory

- A component of the CUDA programming model that was first introduced in CUDA 6.0
- Unified Memory is a single memory address space accessible from any processor in a system



- No need for explicit memory copy calls.
    - When code running on a CPU or GPU accesses data allocated this way (often called CUDA managed data), the CUDA system software and/or the hardware takes care of migrating memory pages to the memory of the accessing processor.
- Benefits:
    - GPU programming is simplified by unifying memory spaces coherently across all GPUs and CPUs in the system and by providing tighter and more straightforward language integration for CUDA programmers.
    - Data access speed is maximized by transparently migrating data towards the processor using it.
- `cudaMallocManaged()` is used to allocate Unified Memory, which replaces the `malloc()` or `new`
- `cudaMallocManaged (void** devPtr, size_t size, unsigned int  flags = cudaMemAttachGlobal)`
    - Allocates memory that will be automatically managed by the Unified Memory system.
    - Parameters:
        - `devPtr` - Pointer to allocated device memory
        - `size` - Requested allocation size in bytes
        - `flags` - Must be either `cudaMemAttachGlobal` or `cudaMemAttachHost` (defaults to `cudaMemAttachGlobal`)

Notes:

- State that we're not going to focus on the `flags` parameter
- Code is similar to the CUDA version 1 code from CEPARCO code discussion
- Implement Unified memory version of the latest 1-D convolution program

- Compare execution time between old method and unified memory method (use nvprof)
- Point out that there are no explicit copy instructions in the code and data transfers are being done by the Unified memory (use nvprof)
- Point out the page faults. Smoothly transition to the prefetching with memory advising.

Data transfer or initialization as a CUDA kernel

- Another type of data transfer which is done through initialization of data in the GPU.
- According to the blog “Unified Memory for CUDA Beginners” by Mark Harris in 2017 under the NVIDIA Developer Technical Blog page, if the data initialization is moved from the CPU to the GPU via kernel, it won't be a page fault.

Notes:
- Sample code of data initialization in a kernel can be seen in the blog, `CUDA Unified Memory for Beginners`. Link can be found in the Reference section.
- Compare if there are any differences with the previous methods of data transferring

Prefetching of data with memory advising

- Behaviors in relation with page faults

Unified Memory	
Pre-Pascal GPU Architectures	**Pascal [and Post-Pascal] GPU Architecture**
• `cudaMallocManaged` physically allocates managed memory in GPU device • Page fault occurs when CPU is accessing allocated memory	• Managed memory may not be physically allocated when `cudaMallocManaged` is called and has returned ○ Physical memory may only be populated on access or prefetching • Page fault occurs if requested page is not in the physical memory of the requesting processor.

- Visual representation of the concept w/ page faults and w/o prefetching with mem advise
- Visual representation of the concept w/ prefetching and memory advice
- Utilizes two of the provided CUDA functions for memory management: `cudaMemPrefetchAsync()` and `cudaMemAdvise()`
    - Additionally, CUDA function for device management is used to get the ID of the GPU device being used, which is the `cudaGetDevice()`
- `cudaMemPrefetchAsync (const void* devPtr, size_t count, int  dstDevice, cudaStream_t stream = 0)`
    - Prefetches memory to the specified destination device.

    - Parameters:
        - `devPtr` - Pointer to be prefetched
        - `count` - Size in bytes
        - `dstDevice` - Destination device to prefetch to
            - Values:
                - `cudaCpuDeviceId` - Device id that represents the CPU
                - ID of the GPU - can be retrieved using
        - `stream ` - Stream to enqueue prefetch operation
- `cudaMemAdvise (const void* devPtr, size_t count, cudaMemoryAdvise advice, int device)`
    - Advise about the usage of a given memory range.
    - Parameters:
        - `devPtr`
        - `count`
        - `advise`
            - Values:
                - `cudaMemAdviseSetReadMostly = 1` - Data will mostly be read and only occassionally be written to
                - `cudaMemAdviseUnsetReadMostly = 2` - Undo the effect of cudaMemAdviseSetReadMostly
                - `cudaMemAdviseSetPreferredLocation = 3` - Set the preferred location for the data as the specified device
                - `cudaMemAdviseUnsetPreferredLocation = 4` - Clear the preferred location for the data
                - `cudaMemAdviseSetAccessedBy = 5` - Data will be accessed by the specified device, so prevent page faults as much as possible
                - `cudaMemAdviseUnsetAccessedBy = 6` - Let the Unified Memory subsystem decide on the page faulting policy for the specified device
        - `device`
            - Values:
                - `cudaCpuDeviceId` - Device id that represents the CPU
                - ID of the GPU
- `cudaGetDevice (int* device)`
    - Returns which device is currently being used.
    - Parameters:
        - `device` - Returns the device on which the active host thread executes the device code.

Notes:
- Visual representation example can be seen in slides 6 - 17 of `Everything you need to know about Unified Memory`. Link can be found in the References section
- Code is similar to the CUDA version 1 code from CEPARCO code discussion

- Apply prefetching and memory advise on the previous code from the previous method
-
-

REFERENCES

- CUDA documentation: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- CUDA Unified Memory for Beginners: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/
- Maximizing Unified Memory Performance in CUDA: https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/
- Improving GPU Memory Oversubscription Performance: https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/
- Everything you need to know about Unified Memory: https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf
- CUDA Tutorial (Vanilla / Not Unified Memory): https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/
- CUDA Toolkit Documentation
    - Device Management: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html
    - Memory Management: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html
    - Data types used by CUDA Runtime: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__TYPES.html
-

Kernel Average Time

# of elements	Old Method	Unified Memory	Prefetching w/ Mem Advising	Initialization w/ CUDA Kernel

	256	512	1024	256	512	1024	256	512	1024	256	512	1024
2^{20}							61.159us	67.834us	77.791us			
2^{22}							229.92us	254.15us	295.30us			
2^{24}							903.24us	1.0013ms	1.1541ms			

Host to Device data transfer time

# of elements	**Old Method**			**Unified Memory**			**Prefetching w/ Mem Advising**			**Initialization w/ CUDA Kernel**		
	256	**512**	**1024**	**256**	**512**	**1024**	**256**	**512**	**1024**	**256**	**512**	**1024**
2^{20}							352.5070us	365.2030us	351.3430us			
2^{22}							1.409651ms	1.405234ms	1.403221ms			
2^{24}							5.618187ms	5.616589ms	5.602315ms			

Device to Host data transfer time

# of elements	**Old Method**			**Unified Memory**			**Prefetching w/ Mem Advising**			**Initialization w/ CUDA Kernel**		
	256	**512**	**1024**	**256**	**512**	**1024**	**256**	**512**	**1024**	**256**	**512**	**1024**
2^{20}							322.2030us	322.2180us	322.1860us			
2^{22}							1.288979ms	1.288148ms	1.288533ms			
2^{24}							5.153105ms	5.154099ms	5.153647ms			

CPU Page Faults

# of elements	**Old Method**			**Unified Memory**			**Prefetching w/ Mem Advising**			**Initialization w/ CUDA Kernel**		
	256	**512**	**1024**	**256**	**512**	**1024**	**256**	**512**	**1024**	**256**	**512**	**1024**
2^{20}	-	-	-				2	2	2			
2^{22}	-	-	-				8	8	8			
2^{24}	-	-	-				32	32	32			