Lecture 1: Basics of Parallel Computing

- **What is parallel computing?** The simultaneous use of multiple compute resources to solve a computational problem.
- **Why use parallel computing?** To better use the advanced parallel hardware, and model or simulate large and complex scientific and engineering problem in a more time and cost-efficient way.
- **What is the computer architecture of parallel computing?** SISD, SIMD, MISD, MIMD. Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units. Most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs employ MIMD.
- **What is the memory architecture of parallel computing?** Multi-node with distributed memory, and multiprocessor with shared memory, hybrid distributed-shared memory
- **What kind of parallel computing languages are available?** MPI, OpenMP, CUDA
- **How to design a parallel program?**
    - Partitioning (data-based domain decomposing by block or cyclic, or task-based functional decomposition, or combined)
    - Communication (to reduce communication overhead, minimize communication latency, increase communication bandwidth, improve communication visibility, and etc.),
    - Synchronization (to manage the sequence of work and the tasks performing it) Types of synchronization: barrier, lock/semaphore, synchronous communication operations
    - Data dependencies (loop carried data dependency)
    - Load balancing (equally partition the work each task receives, dynamic work assignment)
    - Granularity (computation to communication ratio: fine-grain parallelism vs. coarse-grain parallelism)
    - I/O (reduce the overall I/O as much as possible)
- **How to evaluate the performance of a parallel program?** Performance Metrics: Execution time, speedup, efficiency, and scalability. How much you parallelize an algorithm, and the size of data set, the synchronization and memory access overhead, and load balancing all influence the performance.

Lecture 2: Introduction to GPGPU

- **What is GPU?** A “Graphics Processing Unit” that initially specialized for processing computer graphics, and recently evolved towards a more flexible architecture capable of general computations.
- **Architecture of GPU**: It is composed of an array of highly threaded streaming multiprocessors (SMs); Each SM has a number of streaming processors (SPs) that share control logic and instruction cache; each GPU comes with a global memory.
- **CPU/GPU comparisons**
    - CPU is designed to optimize sequential code performance with sophisticated control logic and large cache memories.

    - GPU is designed to optimize the execution throughput of massive numbers of threads to maximize floating-point calculation.
    - GPU has simpler memory models and fewer legacy constraints for higher memory bandwidth.
    - Explosive demand for computing on CPU is too expensive and power-consuming.
- **Why should we use GPUs?**
    - Improved FLOPs, Improved bandwidth, energy efficient, easy accessibility, improved floating-point precision, programmable with high-level language, large speedup, relatively cheap
- **CPU+GPU acceleration**: Combining a CPU with a GPU by accelerating the performance-critical functions on the GPUs offers meaningful benefit.
- **GPGPU programming**
    - Maximize parallel execution, minimize data transfers between host and device, and minimize memory latency on the device.

Lecture 3: Introduction to CUDA

- **What is CUDA?** CUDA is a general-purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way.
- **CUDA programing structure:** Integrated host + device application C program, with CPU and its memory as the host, and GPU and its memory as the device. The phases that exhibit little or no data parallelism are implemented in host code, and the phases that exhibit rich amount of data parallelism are implemented in the device code.
- **Processing flow of a CUDA program**
    - 1. Copy input data from CPU memory to GPU memory;
    - 2. Invoke kernels to operate on the data stored in GPU memory;
    - 3. Copy data back from GPU memory to CPU memory.
- **CUDA memory management and data transfer:** The host and device memory are separate entities, with host pointers point to the CPU memory, and device pointers point to the GPU memory. The CUDA runtime provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory. The CUDA C functions for memory operations are similar to standard C function. We have cudaMalloc(), cudaMemcpy(), cudaMemset(), and cudaFree().
- **CUDA function declaration**
    - CUDA extends C function declarations with three qualifier keywords:
        - The __global__ keyword indicates that the function being declared is a CUDA kernel function, which will be executed on the device and can only be called from the host to generate a grid of threads on a device.
        - The __device__ keyword indicates that the function being declared is a CUDA device function, which executes on a CUDA device and can only be called from a kernel function or another device function.
        - The __host__ keyword indicates that the function being declared is a CUDA host function, which is simply a traditional C function that executes on the host and can only be called from another host function.

- **CUDA threads organization**
    - CUDA has a two-level thread hierarchy, which can be decomposed into blocks of threads and grids of blocks.
    - All threads in a grid share the same global memory space. Threads from the same blocks can cooperate with each other by block-local synchronization or block-local shared memory. Threads from different blocks cannot cooperate.
    - There are two unique built-in pre-initialized coordinate variables (blockIdx and threadIdx) being assigned to each thread by the CUDA runtime when a kernel function is executed. Based on the coordinates, you can assign portions of data to different threads.
- **CUDA device properties**
    - We can query each device, report the properties of each device, and choose the GPU with the most desired properties to run our GPU code through the CUDA device property (cudaDeviceProp). Available functions include cudaGetDeviceCount(), cudaGetDeviceProperties(), cudaChooseDevice(), and cudaSetDevice(), etc.

Lecture 4: CUDA Threads

- **CUDA thread organization:** CUDA programs are a hierarchy of concurrent threads. Threads are grouped into thread blocks, and thread blocks conform a grid.
    - All threads within a thread block run in the same SM; threads of the same block can communicate.
    - All threads in a grid execute the same kernel function.
- **Subdivision of a grid**:
    - Choice of subdivision is up to you, but with hardware limitations.
    - The total size of a block is limited to 512 threads, with flexibility in distributing these elements into the three dimensions as long as the number of threads does not exceed 512. The dimension of grid ranges from 1 to 65,535.
- Threads are assigned to execution resources on a block-by-block basis
    - Blocks can be executed in any order relative to each other to enable **transparent scalability** (The ability to execute the same application code on hardware with different number of execution resources).
- **Thread synchronization**:
    - Threads from the same blocks can cooperate with each other by block-local synchronization or block-local shared memory.
    - CUDA allows threads in the same block to coordinate their activities using a **barrier synchronization** function, __syncthreads(), to ensure all threads in a block have completed a phase of their execution of the kernel before any moves to the next phase.
- **Warps**:
    - Warp size is implementation specific. In the GT200 implementation, once a block is assigned to an SM, it is further divided into 32-thread units warps, with threads 0 through 31 form the first wrap, threads 32 through 63 the second wrap, and so on.

    - If the block has 256 threads, then it has 256/32 = 8 warps. Each warp is executed in a SIMD fashion, i.e., all threads within a warp must execute the same instruction at any given time.
    - There is **implicit intra-warp synchronization** after each instruction, which means, We don't need to __syncthreads() for 32 threads or less per block. It saves lots of time and results in faster implementation.
- **Latency hiding**: When an instruction executed by the threads in a warp must wait for the result of a previously initiated long-latency operation, the warp is not selected for execution. Another resident warp that is no longer waiting for results is selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution. This mechanism of filling the latency of expensive operations with work from other threads is referred to as latency hiding.
    - When thread scheduling does not introduce idle time into the execution of timeline through latency hiding, it is called **zero-overhead thread scheduling**.
- **Warp divergence**: All threads in a warp must execute identical instructions on the same cycle, if one thread executes an instruction, all threads in the warp must execute that instruction. This could become a problem if threads in the same warp take different paths through an application (branch divergence)
    - When threads in the same warp take different paths through an application, all instructions in different branches are to be serialized, this is called **warp divergence**.
    - How to avoid different execution paths within the same warp? Partition data to ensure all threads in the same warp take the same control path, e.g., partitioning data to be a multiple of warp size in many cases.

Lecture 5: CUDA Memories

- **Compute to global memory access (CGMA) ratio**: The number of floating-point calculations performed for each access to the global memory within a region of a CUDA program. It influences the highest achievable floating-point calculation throughput. The higher CGMA ratio the higher the performance.
- **CUDA supports several types of device memories**: global memory, shared memory, constant memory, registers, and texture memory.
    - The global memory and constant memory are the memories that the host code can transfer data to and from the device. Global memory can be read and written by device code. It is large and slow, and subject to traffic congestion to prevent all but a few threads from making progress.
    - The constant memory can only be read by the device. It supports short-latency, high-bandwidth when all threads simultaneously access the same location. When used for data that will not change over the course of a kernel execution (read-only), it reduces the required memory bandwidth.
    - The shared memory and register are on-chip memories. Shared memory is restricted to a thread block. Registers are restricted to a thread. Shared memory and registers are small and fast.
- **Scope and lifetime of CUDA variables**

    - Scope is the range of threads that can access the variable, and lifetime is the portion of the program's execution duration when the variable is available for use. In general, by declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.
- **Reducing global memory access enhances performance**
    - Shared memory is a limited resource but can be very useful for reducing global memory bandwidth. (data is reused)
    - And because the size of the shared memory is quite small and one must be careful not to exceed the capacity of the shared memory when loading data element into the shared memory. Otherwise, exceeding the capacity of the memory will become a limiting factor to parallelism.
    - The same thing holds for the other memory types as well.
- **Memory coalescing**
    - A coalesced memory transaction in CUDA is one in which all of the threads in a half-warp access global memory at the same time. Shared memory can also be used to avoid un-coalesced memory accesses by loading and storing data in a coalesced pattern from global memory and then reordering it in shared memory.
- **Shared memory banks**
    - Shared memory is divided into multiple **banks**, which are equally sized memory modules that can be accessed simultaneously. **The bandwidth of shared memory is 32 bits per bank per clock cycle**. The total number of banks is fixed**.** For older GPUs with compute capability 1.x, they have 16 banks, and for modern GPUs with compute capability2.x, they have 32 banks.
    - Any load/store of *n* addresses that spans *n* distinct banks is executed entirely in parallel.
    - If multiple addresses of a memory request map to the same memory bank, it is called bank conflicts. Multiple thread accesses to the same bank with **bank conflict** are serialized. And the **cost = max # of simultaneous accesses to a single bank.** The only exception is when all threads in a warp access the same bank.
    - There are a few ways to avoid bank conflict, such as identifying some common conflict patterns, using odd stride number to make sure it has no common factors with the number of banks, letting thread loads on element in every consecutive group of blockDim elements, and using memory padding approach, etc.
- **Memory can become a limiting factor to parallelism**: Although CUDA registers, shared memory, and constant memory can be extremely effective in reducing the number of access to global memory, we must be careful not to exceed the capacity of these memories. Each CUDA device offers a limited amount CUDA memory, which limits the number of threads that can simultaneously reside in the streaming multiprocessors for a given application. In general, the more memory locations each thread requires, the fewer the number of threads per SM.

Lecture 6: CUDA Memories by Examples

- Global memory example: Julia Set
    - Shows how to change a CPU-based sequential program to GPU-based parallel CUDA program by identifying the most computationally intensive part in the sequential algorithm on CPU and off-shift it to GPU for faster computation using GPU's global memory.
- Shared memory example: Shared memory bitmap
    - Shows that shared memory is meant to be faster for applications that exhibit locality of data access, but we have to correctly place the synchronization barrier upon the completion of all concurrent processing by threads in the same block.
- Constant memory example: Ray tracing
    - Shows the faster execution capability of constant memory-based computation on GPU. Using constant memory for data that will not change over the course of a kernel execution will likely reduce the required memory bandwidth with improved computational efficiency.
- Texture memory example: Heat transfer simulation
    - **Texture memory**: GPU texture memory is designed to accelerate computations with a considerable amount of spatial locality in the memory access.
    - Shows the details on how to use texture memory for applications exhibiting high "spatial locality" in their memory access patterns.

Lecture 7: Parallel Reduction

- **What is reduction**? Reduction is a general process of taking an input array and performing some computations that produce a smaller array of results. To implement reduction in parallel, we usually decompose the computation into multiple kernel invocations.
- **What is iterative pairwise implementation**? It is a common way to accomplish parallel reduction. Depending on where output elements are stored in-place for each iteration, pairwise parallel implementations can be further classified into Neighbored pair, and Interleaved pair.
    - **Neighbored pair** has all its elements paired with their immediate neighbor. A thread takes two adjacent elements to produce one partial sum at each step.
    - **Interleaved pair** paired elements are separated by a given stride. Usually, the inputs to a thread are strided by half the length of the input on each step.
- Understand all 8 different implementations of parallel reduction in the slides.
- **What is loop unrolling**? It is a technique that attempts to optimize loop execution by reducing the frequency of branches and loop maintenance instructions. It performance gains come from low-level instruction improvements and optimizations that the compiler performs to the unrolled loop.

Lecture 8: Atomics

- **What are race conditions?** Race conditions arise when 2+ threads attempt to access the same memory location concurrently and at least one access is write. Programs with race conditions may produce unexpected, seemingly arbitrary results.

- **What is atomic operation?** It is an uninterruptable read-modify-write operation that forces otherwise parallel threads into a bottleneck, executing the operation one at a time. => slow performance!
- **What kind of atomic functions do we have?** Arithmetic/logical atomic operation, overwriting atomic operation, and compare-and-swap operation. GPU with compute capability 2.0 or above has most of the built-in atomic functions available for atomic operations.
- **What is atomic lock?** a mechanism in parallel computing that forces an entire segment of code to be executed atomically.
- **How to implement lock function?** Use the compare-and-swap atomic function to setup mutex to ensure a safe memory access.

Lecture 9: Streams

- **What is concurrency?** The ability to perform multiple CUDA operations simultaneously. Two levels of concurrency in CUDA C programming:
    - **Kernel level concurrency** (a single task, or kernel, is executed in parallel by many treads on the GPU), and
    - **Grid level currency** (multiple kernel launches are executed simultaneously on a single device). CUDA streams are used to implement grid level concurrency for better device utilization.
- **What are CUDA streams?** A CUDA stream refers to a sequence of asynchronous CUDA operations that execute on a device in the order issued by the host code. In another word, it is a queue of device work. The host places work in the queue and it can continue to work on another task immediately. Meanwhile, device schedules work from streams when resources are free.
- **What is Null stream? And non-Null stream?**
    - The NULL stream is the default stream that kernel launches and data transfers use if you do not explicitly specify a stream.
    - The non-NULL streams are explicitly created and managed. If you want to overlap different CUDA operations, you must use non-NULL streams.
- **Asynchronous, stream-based kernel launches and data transfers enable the following types of concurrency:**
    - Overlapped host computation and device computation
    - Overlapped host computation and host-device data transfer
    - Overlapped host-device data transfer and device computation
    - Concurrent device computation
- **What are the conditions to be satisfied when using streams to overlap device execution with data transfer?**
    - First, the device must support a feature known as device overlap (prop.deviceOverlap);
    - Second, the kernel execution and the data transfer to be overlapped must both occur in different, non- NULL streams;
    - Third, the host memory involved in data transfer must be pinned (page-locked, non-pageable) memory.

- Something to keep in mind:
    - The host memory involved needs to be allocated using **cudaHostAlloc()** since we will queue our memory copies with **cudaMemcpyAsync()** (non-blocking on the host, the call may return before the copy is complete), and asynchronous copies need to be performed with pinned buffers.
    - We need to be aware that **the order in which we add operations to our streams will affect our capacity to achieve overlapping of copies and kernel executions**. The general guideline involves a **breadth-first**, or round-robin, assignment of work to the streams you intend to use.
- **Profiling tools?** nvprof, nvvp, Nsight Compute
    - nvprof is a command-line light-weight GUI-less profiler available for Linux, Windows, and Mac OS. This tool allows you to collect and view profiling data of CUDA-related activities on both CPU and GPU, including kernel execution, memory transfers, etc. Profiling options should be provided to the profiler via the command-line options.
    - The NVIDIA Visual Profiler (nvvp) is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. nvvp can display a timeline of your application's activity on both the CPU and GPU so that you can identify opportunities for performance improvement. It can also analyze your application to detect potential performance bottlenecks and direct you on how to take action to eliminate or reduce those bottlenecks.
    - Nsight Compute is an interactive kernel profiler for CUDA applications. It provides detailed performance metrics and API debugging via a user interface and command line tool. Nsight Compute also provides customizable and data-driven user interface and metric collection that can be extended with analysis scripts for post-processing results.

Lecture 10: Multi-GPU Programming

- **Why do we use multiple GPUs?** The most common reasons for adding multi-GPU support to an application are:
    - Problem domain size: Existing data sets are too large to fit into the memory of a single GPU.
    - Throughput and efficiency: If a single task fits within a single GPU, you may be able to increase the throughput of an application by processing multiple tasks concurrently using multiple GPUs.
- **What's the connectivity of multi-GPU system?**
    - Multiple GPUs connected over the PCIe bus in a single node
    - Multiple GPUs connected over a network switch in a cluster
- **What kind of inter-GPU communication patterns do we have?** Two common inter-GPU communication patterns:
    - No data exchange between problem partitions
        - No data shared across GPUs
        - Each partition can run independently
    - Partial data exchange between problem partitions

        - Need to move data between devices
        - Avoid staging data though host memory
- **How to manage multiple GPUs from a single CPU thread?** To manage multiple GPUs from a single CPU thread, the first step is to determine the number of CUDA-enabled devices available in a system. And we can fetch this information and the properties of those GPUs using device properties API. Next, we iterate over devices using a loop to execute kernels and memory copies from the single host thread.
- **What is CUDA P2P APIs?** The CUDA P2P APIs allows direct communication between GPUs:
    - **Peer-to-peer Access**: Directly load and store addresses within a CUDA kernel and across GPUs.
    - **Peer-to-peer Transfer**: Directly copy data between GPUs.
    - After enabling peer access between two devices, you can copy data between those devices asynchronously using the function **cudaMemcpyPeerAsync**.
    - If peer-to-peer access is not enabled between two GPUs, a peer-to-peer memory copy between those two devices will be staged through host memory, thereby reducing performance.
- **What's the typical workflow for using steams and events in a multi-GPU application?** The typical workflow for using streams and events in a multi-GPU application is:
    - Step 1. Select the set of GPUs this application will use.
    - Step 2. Create streams and events for each device.
    - Step 3. Allocate device resources on each device (for example, device memory).
    - Step 4. Launch tasks on each GPU through the streams (for example, data transfers or kernel executions).
    - Step 5. Use the streams and events to query and wait for task completion.
    - Step 6. Cleanup resources for all devices.
- Something to keep in mind:
    - **You can launch a kernel in a stream only if the device associated with that stream is the current device.**
    - **You can record an event in a stream only if the device associated with that stream is the current device.**
    - **You can query or synchronize any event or stream, even if they are not associated with the current device.**
    - **You can issue a memory copy in any stream at any time, regardless of what device it is associated with or what the current device is.**

Lecture 11: Case Study: Advanced MRI Reconstruction

- Keys to accelerate performance on GPU
    - Determining the kernel parallelism structure
    - **Loop splitting**: takes the body of a loop and splits it into two loops.
    - **Loop interchange**: first swap the inner loop and the outer loop to avoid conflicts between threads

- **Using registers to reduce memory accesses**: load elements into automatic variables before the execution enters the loop, thus converting global memory accesses to register accesses.
- **Chunking data to fit into constant memory**: sequentially process a chunk of the k-space elements that fit into the 64kB capacity of the constant memory.
- **Using hardware trigonometry functions**: change the calls to sin and cos functions into their hardware versions __sin and __cos.