GPGPU

```
// Variables
__device__
- Stored in global memory (large, high latency, no cache)
- Read/write by all threads within grid
- Written by CPU via cudaMemcpyToSymbol()
- Lifetime: application

__constant__
- Same as __device__, but cached and read-only by all threads within
grid
- Written by CPU via cudaMemcpyToSymbol()
- Lifetime: application

__shared__
- Stored in on-chip shared memory (very low latency)
- Read/write by all threads in the same thread block
- Lifetime: block

Unqualified variables (in device code)
- Scalars and built-in vector types are stored in registers
- Arrays of more than 4 elements or run-time indices stored in local
memory
- Read/write by thread only
- Lifetime: thread

// must be volatile to use array as shared memory value
otherwise might get optimized away into registers
__device__ __managed__ volatile int global_counter[2];

// global memory on device
   __device__ __managed__ int counter;

__global__ func can be called by host or device
__device__ can only be called by device
no recursion
no return value --> data transfer by cudaMemcpy
printf can come from global func!

// atomic operations
   atomicAdd(&counter, 1);

// calling Syntax
   non_atomic<<<gridDim, blockDim>>>();
   kernel<<<dim3 grid, dim3 block, int smem, int stream>>>(...)
   mykernel<<<blocks, threads, shared_mem, stream>>>(args);


// synchronize different streams
   in host --> cudaDeviceSynchronize();
   barrier in block -->    __syncthreads();

// memory --> copy the data from address start to address device_mem
// allocate memory on device
   cudaMalloc((void **)&device_mem, num_elements * sizeof(int));
// host to device
   // target,source,size
   rc = cudaMemcpy(device_mem, start, num_elements * sizeof(int), cudaMemcpyHostToDevice);
// device to host
   rc = cudaMemcpy(start, device_mem, num_elements * sizeof(int), cudaMemcpyDeviceToHost);


// unified memory --> main() can work directly with device_mem
   cudaMallocManaged((void **)&device_mem, num_elements * sizeof(int));
// unified memory w/o malloc --> must know size at compile time
   __managed__ int result[NUM_ELEMENTS];

// copy from variable
   cudaMemcpyFromSymbol(&host_result, result, sizeof(start));
   cudaMemcpyToSymbol(dResult, &result, sizeof(unsigned long long));


// how to get index using thread ID
// gridDim.xyz              blockDim.xyz
// blockIdx.xyz             threadIdx.xyz
// const unsigned long long id = threadIdx.x + blockIdx.x * blockDim.x;
```

OPEN_MP

```
// variables declared before pragma are implicitly shared
// variables declared in the loop are implicitly private
pragma this and that

// parallel for loop with n_threads, reduce the variable `killed` using the `+` operation
#pragma omp parallel for num_threads(n_threads) reduction(+ : killed) collapse(2)

// explicit shared and private vars
   #pragma omp parallel for shared(a, b, result) private(i, j, k)

// schedule --> how to break up loops
   static distributes in order (block)
   dynamic does round robin (cyclic/block cyclic) --> eg when different iteration might need different
and indeterminate amount of time
   #pragma omp for schedule (static, chunksize)

// sections --> each `section` of code is executed in parallel
   #pragma omp parallel {
      #pragma omp section{}
      #pragma omp section{}
   }

// The nowait clause overrides any synchronization that would otherwise occur at the end of a
construct.

// only 1 thread executes this
   any 1 thread -->    #pragma omp single
   master thread 0 --> #pragma omp master

// create a critical section
   #pragma omp critical
   {}

// barrier --> threads in a parallel region will not execute beyond the omp barrier until all other
threads in the team complete all explicit tasks in the region.
   #pragma omp barrier

// atomic --> makes the next statement atomic
   #pragma omp atomic
   x[index[i]] += y;
```

MPI

An intra-communicator refers to a single group, an inter-communicator refers to a pair of groups.

```
// send recv
int MPIAPI MPI_Send(
 _In_opt_ void        *buf,
        int        count,
        MPI_Datatype datatype,
        int        dest,
        int        tag,
        MPI_Comm     comm
);
int MPIAPI MPI_Recv(
 _In_opt_ void        *buf,
        int        count,
        MPI_Datatype datatype,
        int        source,
        int        tag,
        MPI_Comm     comm,
 _Out_    MPI_Status   *status
);


// Isend Irecv
int MPIAPI MPI_Isend(
 _In_opt_ void        *buf,
        int        count,
        MPI_Datatype datatype,
        int        dest,
        int        tag,
        MPI_Comm     comm,
 _Out_    MPI_Request  *request
);
int MPIAPI MPI_Irecv(
 _In_opt_ void        *buf,
        int        count,
        MPI_Datatype datatype,
        int        source,
        int        tag,
        MPI_Comm     comm,
 _Out_    MPI_Request  *request
);
```

```
// scatter gather
int MPIAPI MPI_Scatter(
  _In_  void        *sendbuf,
        int         sendcount,
        MPI_Datatype sendtype,
  _Out_ void        *recvbuf,
        int         recvcount,
        MPI_Datatype recvtype,
        int         root,
        MPI_Comm    comm
);
int MPIAPI MPI_Gather(
  _In_     void        *sendbuf,
           int         sendcount,
           MPI_Datatype sendtype,
  _Out_opt_ void       *recvbuf,
           int         recvcount,
           MPI_Datatype recvtype,
           int         root,
           MPI_Comm    comm
);
int MPIAPI MPI_Allgather(
  _In_  void        *sendbuf,
  _In_  int         sendcount,
  _In_  MPI_Datatype sendtype,
  _Out_ void        *recvbuf,
        int         recvcount,
        MPI_Datatype recvtype,
        MPI_Comm    comm
);
```

scatter sends `n` elements from the array in root to each receipient (by communicator)
gather in main receives `n` elements from all members on communicator, puts into array
allGather does gather + broadcast, all members have an array to write into

```
// broadcast --> uses a tree-like structure for efficiency
// root sends the data at &data. all others receive it at their &data
  MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)

// barrier
    MPI_Barrier(MPI_Comm communicator)
```

```
// reduce --> reduction on a per-element basis, ie a[0] reduced and a[1] reduced separately
MPI_Reduce(
    void* send_data,
    void* recv_data, <-- only needed on root
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
// MPI_Allreduce --> reduce followed by broadcast
MPI_Allreduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm communicator)
```