CUDA

Exercise 1: Enumerate the main architectural parameters of Fermi GPU architecture.

Exercise 2: Enumerate the main architectural parameters of Kepler GPU architecture.

Exercise 3: Enumerate the main architectural parameters of Maxwell GPU architecture.

Exercise 4: Enumerate the main architectural parameters of Tesla and Turing GPU architectures.

Exercise 5: Enumerate the main architectural parameters of Pascal GPU architecture.

Exercise 6: Enumerate the main architectural parameters of Volta GPU architecture.

Exercise 7: Enumerate the main architectural parameters of Ampere GPU architecture.

Exercise 8: Code a CUDA program that computes a matrix *A x B* operation with two matrices *A* and *B* stored at the host address space. Assume the matrices be square (N x N).

Exercise 9: Code a CUDA program that computes a matrix per vector operation *A x v* with the matrix and vector stored at the host address space. Assume the matrix be square (N x N) and vector of size N.

Exercise 10: Code a multi-GPU CUDA program that computes the Jacobi solver we have studied within the LAB assignment.

Exercise 11: Code a multi-GPU CUDA program that computes the Red-Black solver we have studied within the LAB assignment.

Exercise 12: Code a multi-GPU CUDA program that computes the Gauss solver we have studied within the LAB assignment.

Exercise 13: Code a multi-GPU CUDA implementation of the function *CUDA_Scatter* with the same semantic as the *MPI_Scatter* primitive. Assume G the total number of GPUs in the system. Assume peer-to-peer support between the GPUs in the system. Assume all necessary GPU buffers have been already allocated.

```
cudaError_t CUDA_Scatter(void* SendRecv[], unsigned int Root,
                         unsigned int Bytes);
```

where `SendRecv` is a vector of GPU addresses where the receive buffers and send buffer are located, `Root` indicates which GPU scatters the data to the other GPUs and `Bytes` indicates the number of bytes to be transferred to each GPU.

Exercise 14: Code a multi-GPU CUDA implementation of the function *CUDA_Gather* with the same semantic as the *MPI_Gather* primitive. Assume G the total number of GPUs in the system. Assume peer-to-peer support between the GPUs in the system. Assume all necessary GPU buffers have been already allocated.

```
cudaError_t CUDA_Gather(void* SendRecv[], unsigned int Root,
                        unsigned int Bytes);
```

where `SendRecv` is a vector of GPU addresses where the send buffers and receive buffer are located, `Root` indicates which GPU gathers the data from the other GPUs and `Bytes` indicates the number of bytes to be transferred by each GPU.

Exercise 15: Code a multi-GPU CUDA implementation of the function *CUDA_Bcast* with the same semantic as the *MPI_Bcast* primitive. Assume G the total number of GPUs in the system. Assume peer-to-peer support between the GPUs in the system. Assume all necessary GPU buffers have been already allocated.

```
cudaError_t CUDA_Bcast(void* SendRecv[], unsigned int Root,
                       unsigned int Bytes);
```

where `SendRecv` is a vector of GPU addresses where the send buffers and receive buffer are located, `Root` indicates which GPU gathers the data from the other GPUs and `Bytes` indicates the number of bytes to be transferred to each GPU.

Exercise 16: Code a multi-GPU CUDA implementation of the function *CUDA_Scatterv* with the same semantic as the *MPI_Scatterv* primitive. Assume G the total number of GPUs in the system. Assume peer-to-peer support between the GPUs in the system. Assume all necessary GPU buffers have been already allocated.

```
cudaError_t CUDA_Scatterv(void* SendRecv[], unsigned int Root,
                          unsigned int Bytes[]);
```

where `SendRecv` is a vector of GPU addresses where the send buffers and receive buffer are located, `Root` indicates which GPU gathers the data from the other GPUs and `Bytes` indicates the number of bytes to be transferred to each GPU.

Exercise 17: Code a multi-GPU CUDA implementation of the function *CUDA_Gatherv* with the same semantic as the *MPI_Gatherv* primitive. Assume G the total number of GPUs in the system. Assume peer-to-peer support between the GPUs in the system. Assume all necessary GPU buffers have been already allocated.

```
cudaError_t CUDA_Gatherv(void* SendRecv[], unsigned int Root,
                         unsigned int Bytes[]);
```

where `SendRecv` is a vector of GPU addresses where the send/receive buffers are located, `Root` indicates which GPU gathers the data from the other GPUs and `Bytes` indicates the number of bytes to be transferred by each GPU.

Exercise 18: Code a CUDA program that permutes the content of a vector according to the content of another vector. Include in your code the necessary CUDA statements that should be included in the main program regarding memory allocation and data transfers to the GPU. **Do not** use *shared* memory. The CPU version of the computation would be:

```
unsigned int p[N];
int V[N];
int W[N];

int main () {

// CPU code
for (i=0; i<N-RADIUS; i++)
  for (j=0; j<RADIUS; j++) V[p[i]] += W[p[i+j]];

} // main
```

Exercise 19: In the previous exercise, utilize *shared* memory and maximize the coalescing of memory accesses within the warp execution.

Exercise 20: Given the following data structures and function definition, sketch a CUDA version that does the same computation on a device. Include in your code the necessary CUDA statements that should be included in the main program regarding memory allocation and data transfers to the GPU.

```
#define MAX_LENGTH_NAME (32)

struct {
  unsigned int mID;
  char mName[MAX_LENGTH_NAME];
  char mSurnames[MAX_LENGTH_NAME*2];
  unsigned int mAge;
  unsigned int mGender;
  unsigned int mPhone;
} Data;

unsigned int nClients;
Data Clients[MAX_CLIENTS];

unsigned int SearchID(unsigned int ID) {
  unsigned int pos=MAX_CLIENTS;
  for (i=0; i<nClients; i++) {
    if (ID == Clients[i].mID) { pos = i; break; }
  }
  return (pos);
}
```

Exercise 21: In the previous exercise, do you detect a problem with the memory accesses performed over the `Data` and `Clients` data structures (at the warp level)? Are they coalesced? Propose a solution that changes the memory layout of the `Data` struct and the definition of the variable `Clients`. **Note**: think to switch from an *Array of Structs* (AoS) to a *Struct of Arrays* (SoA). This transformation is very usual (when possible) in CUDA programming. Rewrite the GPU code according to this transformation.

Exercise 22: Given the following CUDA program, how many instances will be generated of variable `tmp`? What is the most likely storage used to store these instances: registers, shared memory, global memory? Given the code of the kernel, what should be the value for the non-specified parameter at kernel invocation?

```
__global__ MyKernel (unsigned int* d_V, unsigned int N) {

  __shared__ int tmpV[1024];
  unsigned int tmp;
  ...
}

int main() {
  // Memory alloc at device
  ...

  // Copy data to device
  ...

  // Kernel invocation
  dim3 GridDim(ceal(N/256);
  dim3 BlockDim(256);
  MyKernel<<< GridDim, BlockDim, _____? >>>(d_V, N);

  ...
}
```

Exercise 23: In the previous exercise, what is the relation between the total number of registers in an SM, and the total amount of registers used by a kernel invocation? What happens if the first is less than the second?

Exercise 24: Use the following NVIDIA link https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls to analyze the occupancy of your kernel in the CUDA LAB assignment. What is the flag we use to instruct the *nvcc* compiler to print the number of registers per kernel?