Maxi

Appropriate CUDA

Hiding Latency

At least 400-600 clock cyles per global memory access
Keep SM busy: 512 per SM

Occupancy: Number of warps running/max warps

Use of __launch_bounds__(xxx1,xxx2)   per kernel
where xxx1 is max threads per block
xxx2 is min blocks launched at once on SM
This restricts number of registers used so more threads

Example:

```
__global__ void __launch_bounds__(maxThreadsPerBlock,
minBlocksPerMultiprocessor) MyKernel(...)
{ ... }
```

CUDA GPU Occupancy Calculator

Multiple GPU

Different GPU cards on same PCIe bus (same workstation)

Multi-GPU:

Provides more processors to speed-up work
and/or provides more memory for on-card storage

Using nvidia-smi (on Linux):

```
++------------------------------------------------------+
| NVIDIA-SMI 331.20     Driver Version: 331.20         |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Quadro K600         Off  | 0000:01:00.0      On |                  N/A |
| 25%   52C    P0    N/A /  N/A |    211MiB /  1023MiB |      1%      Default |
+-------------------------------+----------------------+----------------------
|   1  Tesla K20c          Off  | 0000:02:00.0     Off |                    0 |
| 30%   39C    P8    15W / 225W |     12MiB /  4799MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K20c          Off  | 0000:03:00.0     Off |                    0 |
| 30%   30C    P8    16W / 225W |     12MiB /  4799MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Multiple GPU

Need inter-GPU communication

Checking device information

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    printf("Device %d has compute capability %d.%d.\n",
           device, deviceProp.major, deviceProp.minor);
}
```

Single CPU thread

All CUDA calls are issued to current GPU

cudaSetDevice() sets the current GPU

Example of concurrent execution:

```
cudaSetDevice(0);
kernel <<<…>>> (….);
cudaSetDevice(1)
kernel <<<…>>> (….);
```

Each device has its own stream.

Using GPU-Specific Streams

```
cudaSetDevice(0);              // Set device 0 as current
cudaStream_t s0;
cudaStreamCreate(&s0);         // Create stream s0 on device 0
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 0 in s0
cudaSetDevice(1);              // Set device 1 as current
cudaStream_t s1;
cudaStreamCreate(&s1);         // Create stream s1 on device 1
MyKernel<<<100, 64, 0, s1>>>(); // Launch kernel on device 1 in s1
```

```
// This subsequent kernel launch will fail:
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 1 in s0
```

cudaEventRecord() will fail if the input event and input stream are associated to different devices.

cudaEventElapsedTime() will fail if the two input events are associated to different devices.

Peer-to-Peer Memory Access

```
cudaSetDevice(0);                    // Set device 0 as current
float* p0;
cudaMalloc(&p0, 1024*sizeof(float));             // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0);        // Launch kernel on device 0
cudaSetDevice(1);                    // Set device 1 as current
cudaDeviceEnablePeerAccess(0, 0);   // Enable peer-to-peer access
                                      // with device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
MyKernel<<<1000, 128>>>(p0);
```

Peer-to-Peer Memory Copy

```
cudaSetDevice(0);                    // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);               // Allocate memory on device 0
cudaSetDevice(1);                    // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);               // Allocate memory on device 1
cudaSetDevice(0);                    // Set device 0 as current
MyKernel<<<1000, 128>>>(p0);         // Launch kernel on device 0
cudaSetDevice(1);                    // Set device 1 as current
cudaMemcpyPeer(p1, 1, p0, 0, size); // Copy p0 to p1
MyKernel<<<1000, 128>>>(p1);         // Launch kernel on device 1
```

If cudaDeviceEnablePeerAccess() is enabled, host not involved, so faster