1 Parallel Processing

Why Parallel Processing

There are a number of domains in which problems naturally lend themselves to parallel computation. As silicon technology hits the frequency limit, processors are going wide (multiple cores) to continue satisfying Moore’s law for computational power. Many applications lend themselves to a small amount of parallel processing, while other applications can take full advantage of the large number of small, fast processors available on GPUs.

Parallel programming refers to the programs that simultaneously use multiple processors to execute different threads. A typical application of this, which is used in this project, is to parallelize a ‘loop’ of some sort that is iterating through data independently—that is, each iteration through the loop doesn’t depend on another one.

GPUs are very important in considering parallel programming. They are designed to perform very large problems and operations in parallel. This allows for the massive amounts of rendering that must take place when playing real-time games. NVIDIA has developed a parallel programming language, CUDA, that can be run using their graphics cards.

CUDA

CUDA is a parallel programming language developed by NVIDIA. CUDA is a C-like language, and is relatively easy to implement given a C program to parallelize.

When converting a sequential program to CUDA, there a few things to think about. For example, when programming a loop in CUDA you do not simply iterate over each element. Instead, you create a function called a ‘kernel’. This function does the same work that the loop would do, but it scatters it across different threads that will then run in parallel.

2 Laser Scanning

Laser Scanning – Why Parallel Processing?

Laser scanning is an object recognition method. A laser scanner sends out a pulse of light and then measures the amount of time it takes to return after hitting an object. In this particular application, the laser...
is used for object recognition in a robot. We have a single laser that sends out the light pulse, waits for it to return, the moves to the next line and repeats until an entire sweep is finished. Currently, the application completes approximately 1,000 sweeps. Processing the data in a reasonable amount of time is a large problem, which we intend to fix via parallel programming. The importance of this can be illustrated via other applications as well. For example, when laser scanning is used as object detection on moving vehicles or systems, the amount of time it takes for the system to process the incoming data is extremely critical—hence the importance of parallel programming.

Other Applications using Laser Scanning

There are many applications that use laser scanning that could greatly benefit from processing the incoming data in parallel. To name a few examples, laser scanning is used in factory automation, medical imaging, robotics, and terrain mapping as well as many others.

There are also many different physical devices available to collect the data. Unfortunately, most of them are extremely expensive and aren’t available for under thousands of dollars. For instance, the Swiss Ranger SR400 uses a multi-pixel time of flight sensor and retails for approximately $9,000.

However, the Microsoft Kinect was recently released for the Xbox 360, which offers a much more economical alternative to the more expensive sensors. The Microsoft Kinect sensor for uses a prime sense sensor, which has already been ‘hacked’ by many developers to collect 3d data from sensors that can then be processed and used. The exciting thing about the Microsoft Kinect is that can be purchased for $150, which is far more affordable than any other sensor on the market. The Microsoft Kinect projects a pattern using parallax and reads 480 scan lines 30 times a second—so you get 14,400 scan lines per second. This makes it pretty obvious that parallelizing the processing of these scan lines is incredibly important.

3 Sequential C Code

The following C code is the section that was parallelized. It is a portion of a function called FindObjectsInSingleScanLine. This function goes through an array of scan lines looking for possible objects. Each scanned line has its own array of points (each containing an x, y, and z value). An object is found if the x, y, z points are out of a certain range.
4 Parallel CUDA Code

The code was parallelized on one dimension – ScanLine. So, CUDA will compute the information for each ScanLine in the loop in the above implementation on a different thread.
Here is the CUDA initialization:

```c
void main_cuda()
{
    double TotalExecutionTime = 0.0;
    double Current = 0.0;

    LASER_SCANNER_DATA_T* LaserData;
    StartCounter();
    cudaMemcpy((void**)&LaserData, sizeof(LASER_SCANNER_DATA_T), Current = GetCounter();
    TotalExecutionTime = Current;
    std::cout << "Time to malloc graphics memory: " << Current << std::endl;
    StartCounter();
    cudaMemcpy(LaserData, p_LaserScanLineData, sizeof(LASER_SCANNER_DATA_T), cudaMemcpyHostToDevice);
    Current = GetCounter();
    TotalExecutionTime = Current;
    std::cout << "Time to copy memory to the gpus: " << Current << std::endl;
    StartCounter();
    kernel(<FULL_SCAN_HEIGHT, 1>,(LASER_SCANNER_DATA_T*)LaserData);
    Current = GetCounter();
    TotalExecutionTime = Current;
    std::cout << "Time to run memory to the gpus: " << Current << std::endl;
    StartCounter();
    cudaMemcpy(p_LaserScanLineData, LaserData, sizeof(LASER_SCANNER_DATA_T), cudaMemcpyDeviceToHost);
    Current = GetCounter();
    TotalExecutionTime = Current;
    std::cout << "Time to copy gpu memory back to the cpus: " << Current << std::endl;
    std::cout << "Total time to execute cuda code: " << TotalExecutionTime << std::endl;
    cudaFree(LaserData);
}
```

And the kernel function:

```c
__global__ void kernel (LASER_SCANNER_DATA_T* laserScanData)
```

```c
    int ScanLine = blockIdx.x;
    int NumberOfSamples = laserScanData[ScanLine].NumberOfSamples;
    int SampleCount = 0;
    double AverageSum = 0;
    double FloorAverage = 0;
    int Position = laserScanData[ScanLine].LaserAngleTenthDegrees;
    // This code looks for a step in the z direction, by looking for
    // N floor samples -- Gap -- N object samples
    // Note ALL VALUES IN TENTH INCHES unless otherwise indicated
    int SmallObjStartIndex = 0;
    int SmallObjPeakHeight = 0;
    int nDetectedObjects = 0;
    BOOL LeadingEdgeFound = FALSE;
    BOOL TrailingEdgeFound = FALSE;
    double LeftSum = 0;
    int nLeftSamples = 0;
    double RightSum = 0;
    int nRightSamples = 0;
    int PeakDetectOffsetIndex = 0;
    double RightFloorValue = 0;
    double LeftFloorValue = 0;
    for(int i = 0; i < NumberOfSamples; i++)
    {
        RightSum = 0.0;
        nRightSamples = 0;
        LeftSum = 0.0;
        nLeftSamples = 0;
        double RightAve = 0;
        double LeftAve = 0;
        ```
5 Experiment Data & Results

In order to see if the CUDA code ran faster than the sequential code, I inserted timers into the code to measure execution times. The code was run on a machine with the following specifications:

**CPU**
- Intel Core 2 Duo @ 2.5Ghz
- Memory: 4GB DDR2
- Windows 7 64bit

**Graphics Card**
- Quadro NVS 140M
- Video Memory: 512MB GDDR3
- Shared Memory: 1797MB
- 64bit memory interface
- Memory Clock: 600Mhz
- Shader clock: 800Mhz
- Core Clock: 400Mhz
- CUDA Cores: 16

Here is the output for varying the ScanLines (runtime is in milliseconds):

### ScanLines = 640
- Time to initialize: 1241.85
- Time to malloc graphics mem: 5.32838
- Time to copy mem to gpu: 0.0279209
- Time to run mem to gpu: 0.0291527
- Time to copy gpu mem to cpu: 0.0225831
- **Total time to run cuda code:** 5.40804
- **Total time to run sequential:** 1448.12

### ScanLines = 1024
- Time to initialize: 1201.93
- Time to malloc graphics mem: 5.6585
- Time to copy mem to gpu: 0.0262785
- Time to run mem to gpu: 0.0291527
- Time to copy gpu mem to cpu: 0.0225831
- **Total time to run cuda code:** 5.40804
- **Total time to run sequential:** 1448.12

### ScanLines = 2500
- Time to initialize: 1205.33
- Time to malloc graphics mem: 5.37108
- Time to copy mem to gpu: 0.0270997
- Time to run mem to gpu: 0.0324376
- Time to copy gpu mem to cpu: 0.0197089
- **Total time to run cuda code:** 5.40804
- **Total time to run sequential:** 5569.02

### ScanLines = 5000
- Time to initialize: 1222.6
- Time to malloc graphics mem: 8.46333
- Time to copy mem to gpu: 0.0373648
- Time to run mem to gpu: 0.0299739
- Time to copy gpu mem to cpu: 0.0246361
- **Total time to run cuda code:** 5.40804
- **Total time to run sequential:** 11120.8

### ScanLines = 10000
- Time to initialize: 1210.59
- Time to malloc graphics mem: 5.59568
- Time to copy mem to gpu: 0.0234043
- Time to run mem to GPU: 0.0221725
- Time to copy gpu mem to cpu: 0.0238149
- **Total time to run cuda code:** 5.40804
- **Total time to run sequential:** 22634
### Conclusions

It is quite apparent that the CUDA implementation of this code is far more efficient than the sequential version. While this is not a surprise, I did not expect there to be such a drastic difference. While currently there is typically only a data size of 1000 scan lines in the specific application that I parallelized, there is quite a potential for growth—as well as the possibility for this parallelization to be implemented in similar but more advanced applications.

<table>
<thead>
<tr>
<th>Scan Lines</th>
<th>Parallel (CUDA)</th>
<th>Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>640</td>
<td>0.0054</td>
<td>1.448</td>
</tr>
<tr>
<td>1024</td>
<td>0.00545</td>
<td>2.289</td>
</tr>
<tr>
<td>2500</td>
<td>0.00573</td>
<td>5.569</td>
</tr>
<tr>
<td>5000</td>
<td>0.00655</td>
<td>11.12</td>
</tr>
<tr>
<td>10000</td>
<td>0.00566</td>
<td>22.634</td>
</tr>
</tbody>
</table>

![Execution Time Graph](image)

**Number of ScanLines (Data Size)**

- **Parallel (CUDA)**
- **Sequential**