CUDA Lab
William Curran

Background
CPUs are optimized to run sequential code, and this has been fine for years. Now, developers of CPU chips are coming under the problem of using too much power to process code extremely fast on one chip. They are now forced to use multiple cores on a CPU, but this comes with its own problem. In order to use multiple cores, code has to be independent and threaded, and most code to-date has not been built this way, with the exception of one type. Graphics processing is renowned for being inherently parallelizable, and modern day GPUs come with many cores. GPUs are optimized to use many cores to run parallelized code, and this can be used to make parallelizable code run extremely fast. CUDA is the first easy to use parallel processing language developed by NVIDIA.

How to use?
CUDA uses flags to denote code ran on the GPU instead of the CPU. Any function with the __global__ flag before the function denotes a function that can be called by the CPU. A function with the __device__ flag denotes a function that can only be called while inside the GPU.

One main problem with GPU programming is memory. GPUs cannot access memory saved on the CPU, and a CPU cannot access memory saved on a GPU. This problem is alleviated by passing memory back and forth. CUDA comes with a cudaMalloc function that allocates memory to the GPU. Once you are done using the GPU for processing on that variable, CUDA also comes with its own CudaMemcpy function. This allows you to copy memory from the GPU onto the CPU. This is helpful because the GPU only has a few assembly instructions, compared to the CPU, which has many more.

Another handy feature of CUDA is being able to declare up to a 3d array of blocks to send to the GPU. This allows rendering of 2d and 3d images to be done very easy, as you don’t have to keep track of which thread is which dimension. This is done using the dim3 function in the following example.
Comprehensive example of performance

I chose to compare the results of computing a fractal using the NVIDIA GeForce GT 220M graphics card, and the Intel Core 2 Duo T6600 @ 2.20Ghz. In this algorithm the time complexity comes with computing how quickly a complex equation converges or whether or not it diverges. The complex equation is expressed as \( f(z) = z^2 + c \) where \( c \) is a complex constant and \( z \) represents your location in x,y space.

```c
#include "cpu_bitmap.h"
#include <time.h>
#include <stdio.h>

#define DIM 1000
struct Complex {
    float r;
    float i;
    Complex(float a, float b) : r(a), i(b) {}
    __device__ float magnitude2(void) {
        return r * r + i * i;
    }
    __device__ Complex operator*(const Complex & a) {
        return Complex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ Complex operator+(const Complex & a) {
        return Complex(r+a.r, i+a.i);
    }
};

__device__ int julia(int x, int y) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x) / (DIM/2);
    float jy = scale * (float)(DIM/2 - y) / (DIM/2);
    Complex c(-0.8, 0.156);
    Complex z(jx, jy);
    int i = 0;
    Complex zprev(jx, jy);
    for (i=0; i<255; i++) {
        zprev = z;
        z = z * z + c;
        if (z.magnitude2() > 1000)
            return 255;
        if (i > 150) {
            if (abs((float(z.magnitude2()) - float(zprev.magnitude2()))) < 0.5)
                return 255 - (i - 150)*2;
        }
    }
    return 255-i;
}
```
__global__ void kernel( char *bitmapPointer ) {
    // blockIdx contains the index of whatever block is running this code.
    // Since we declared using a two dimensional index earlier, we have an x
    // and a y.
    int x = blockIdx.x;
    int y = blockIdx.y;
    // map from blockIdx to pixel position
    int pointerOffset = x + y * blockDim.x;
    // now calculate the value at that position
    int juliaValue = julia( x, y );
    bitmapPointer[pointerOffset*4 + 0] = 255-juliaValue;
    bitmapPointer[pointerOffset*4 + 1] = 255-juliaValue;
    bitmapPointer[pointerOffset*4 + 2 ] = 255-juliaValue;
    bitmapPointer[pointerOffset*4 + 3 ] = 255;
}

int main( void ) {
    clock_t init, final;
    init = clock();
    CPUBitmap bitmap( DIM, DIM );
    char *cuda_bitmap;
    // Allocate memory on the GPU
    cudaMalloc(( void**)&cuda_bitmap, bitmap.image_size());
    // Declare a 2 dimensional array to use later to tell the GPU to use two
    // dimensional indexing
    dim3 array(DIM, DIM);
    // Call our GPU function, telling it how many threads to run (array
    // variable)
    kernel<<<array, 1>>>( cuda_bitmap );
    // Copy the GPU bitmap to our CPU bitmap to be able to display it.
    cudaMemcpy(bitmap.get_ptr(), cuda_bitmap, bitmap.image_size(), cudaMemcpyDeviceToHost);
    final = clock() - init;
    printf("%f",(double)final / ((double)CLOCKS_PER_SEC));
    bitmap.display_and_exit();
    cudaFree(cuda_bitmap);
}

I ran this algorithm along with its comparative C program and here are the benchmark results of
different sized images.

<table>
<thead>
<tr>
<th>Size of the image n x n</th>
<th>CUDA (s)</th>
<th>C (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>.140</td>
<td>.081</td>
</tr>
<tr>
<td>200</td>
<td>.187</td>
<td>.265</td>
</tr>
<tr>
<td>400</td>
<td>.202</td>
<td>1.03</td>
</tr>
<tr>
<td>600</td>
<td>.296</td>
<td>2.34</td>
</tr>
<tr>
<td>800</td>
<td>.430</td>
<td>4.09</td>
</tr>
<tr>
<td>1000</td>
<td>.546</td>
<td>6.39</td>
</tr>
</tbody>
</table>
Conclusion

As you can see, the GPU severely outperforms the CPU when parallelizable code is used. There are a few things to take into account. The GPU has a much larger overhead in memory allocation, so when there are a very few computations, the CPU will outperform the GPU. You can see this clearly in the 100 x 100 image. The code for CUDA also requires much higher complexity, as you have to juggle around variables and have functions that can only be called by either the CPU or GPU. Even with these fallbacks, it is obvious that in order for performance to increase in CPUs parallelizable code will be needed.