CUDA algorithm

Since the computation of the integral depends primarily only on the sum of the terms $2f(x_i)$, this task can be divided among a number of processes, computed independently, and the results of those individual tasks can be combined to get the sum. This is the basis for a distributed parallel algorithm approach.

CUDA

The CUDA (Compute Unified Device Architecture) model of parallel computing is based on a set of computing processes housed on a CUDA certified NVIDIA graphics chip [cuda]. A large number of threads can be executed simultaneously on the NVIDIA graphics chip.

In the following subsection, a CUDA program is implemented for the Trapezoidal Rule. The different *device* and *host* functions and their calls will be carefully explained in the context of the Trapezoidal Rule.

CUDA implementation

This algorithm can be easily implemented using *CUDA*. Assign the independent tasks to different computing threads on the graphics chip, and compute the sum from the partial sums on the host. The following example shows a naive implementation; the purpose of this implementation is to maximize clarity of computing using *CUDA*, even at the expense of addition speedup.

On lines 19-23 is the implementation of the *myfunction()*, *f(x)*. the *__device__* designator indicates this function is implemented to run only on the device, graphics, and can be called only by functions running on the device.

On lines 27-40 is the implementation of the kernel function, *integrateKernel()*, which is executed on each thread in the graphics chip. The *__global__* designator specifies the function as an entry point to the GPU. The calling program must specify the number of participating threads to execute this function. This function is passed the address of an array of floats where the partial results will be stored, a floating point value *c* which is the left hand endpoint of the interval of integration, a floating point value *deltaX* and an integer N which together indicate the numerical approximation will be computed over *N* subintervals each with length *deltaX*. One line 30, *idx* is computed to be the thread number (remember there are many executing threads), on line 31, *x* is computed to be the value of the left hand endpoint for this thread, and lines 33-37 compute the partial result, only if the thread corresponds to one of the *N* subintervals.

On lines 46-94 the host function *cudaIntegrate()* is implemented. It is passed the interval of integration $[c,d]$ and the number of subintervals to use, *N*. On line 49, the length of each subinterval, *deltaX* is computed. On line 55, the size in bytes of the array to hold the partial results is computed. On line 58, the space is allocated on the host computer for the partial results, and on lines 60-65 using cudaMalloc() the space is allocated on the device for the partial results. On lines 68-69 the number of threads needed is determined based on the number of blocks and the individual block size. On line 73 the *integratorKernel* on the device is started and is passed four arguments, as well the number of threads are specified as *n_blocks*block_size*. On line 76, the array of partial results computed on the device are copied into an array on the host using cudaMemcpy(). On lines 84-86 the trapezoidal approximation is completed. On lines 89-90 the host and device arrays are released; cudaFree() is used to deallocate the array on the device.

In the main function the host function *cudaIntegrate()* is called on line 111.

```
1   // This program implements trapezoidal integration for a function
2   // f(x) over the interval [c,d] using N subdivisions.  This program
3   // runs on a host and device (NVIDIA graphics chip with cuda
4   // certification).  The function f(x) is implemented as a callable
5   // function on the device.  The kernel computes the sums f(xi)+f(xi+deltaX).
6   // The host function computes of the individual sums computed on the
7   // device and multiplies by deltaX/2.
8
9   #include <iostream>
10  #include <ctime>
11
12  using namespace std;
13  #include <cuda.h>
14  #include <math_constants.h>
15  #include <cuda_runtime.h>
16
17  // function to integrate, defined as a function on the
18  // GPU device
19  __device__ float myfunction(float a)
20  {
21
22          return a*a+2.0f*a + 3.0f;
23  }
24
25  // kernel function to compute the summation used in the trapezoidal
26  // rule for numerical integration
27  __global__ __device__ void integratorKernel(float *a, float c, float deltaX, int N)
28  {
29
30          int idx = blockIdx.x * blockDim.x + threadIdx.x;
31          float x = c + (float)idx * deltaX;
32
33          if (idx<N)
34          {
35                  a[idx] = myfunction(x)+myfunction(x+deltaX);
36
37          }
38
39
40  }
41
42
43  // cudaIntegrate() is the host function that sets up the
44  // computation of the integral of f(x) over the interval
45  // [c,d].
46  __host__ float cudaIntegrate(float c, float d, int N)
47  {
48          // deltaX
49          float deltaX = (d-c)/N;
50
51          // error code variable
52          cudaError_t errorcode = cudaSuccess;
53
54          // size of the arrays in bytes
55          int size = N*sizeof(float);
56
57          // allocate array on host and device
58          float* a_h = (float *)malloc(size);
59
60          float* a_d;
61          if (( errorcode = cudaMalloc((void **)&a_d,size))!= cudaSuccess)
62             {
63                  cout << "cudaMalloc(): " << cudaGetErrorString(errorcode) << endl;
64                  exit(1);
65             }
66
67          // do calculation on device
68          int block_size = 256;
69          int n_blocks = N/block_size + ( N % block_size == 0 ? 0:1);
70          // cout << "blocks: " << n_blocks << endl;
71          // cout << "block size: " << block_size << endl;
72
73          integratorKernel <<< n_blocks, block_size >>> (a_d, c, deltaX, N);
74
75          // copy results from device to host
76          if((errorcode = cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost))!=cudaSuccess)
77              {
78                  cout << "cudaMemcpy(): " << cudaGetErrorString(errorcode) << endl;
79                  exit(1);
80              }
81
82
83          // add up results
84          float sum = 0.0;
85          for(int i=0; i<N; i++) sum += a_h[i];
86          sum *= deltaX/2.0;
87
88          // clean up
89          free(a_h);
90          cudaFree(a_d);
91
92
93          return sum;
94  }
95
96
97  // utility host function to convert the length of time into
98  // micro seconds
99  __host__ double diffclock(clock_t clock1, clock_t clock2)
100 {
101         double diffticks = clock1-clock2;
102         double diffms = diffticks/(CLOCKS_PER_SEC/1000);
103         return diffms;
104 }
105
106
107
108 // host main program
109 int main()
110 {
111         clock_t start = clock();
112         float answer = cudaIntegrate(0.0,1.0,1000);
113         clock_t end = clock();
114
115         cout << "The answer is " << answer << endl;
116         cout << "Computation time: " <<  diffclock(end,start);
117         cout << "  micro seconds" << endl;
118
119         return 0;
120 }
```

Key algorithmic features

The allocation of space on the host and device and the transfer of the partial values from the device to the host are computationally expensive. A good question is can these be significantly improved. The answer is *yes*, but how will not be addressed here, since the technique, at this point, would obfuscate our discussion.

Speed up

For any parallel algorithm, the *speedup* is the ratio of the execution times of the sequential and parallel implementations. All the threads in the same block execute simultaneously and each takes constant time. The number of blocks is $n\_blocks = \frac{n}{256}$. However, on line 85 of *main()* $\Theta(n)$ steps are taken to complete the computation; so the total execution time is the sum of the parallel execution times plus the sequential time to complete the computation. Now speed up can be computed:

$$S(n) = \frac{T(1)}{\Theta(n\_blocks) + \Theta(n)} = \frac{\Theta(n)}{\frac{n}{256} + \Theta(n)} \approx \Theta(1)$$

for reasonable *n*. So it appears that question of *can the allocation of space and transfer to data be improved?* turns out to be very important.

Questions

1. Take time to carefully read through the implementation. What do you **not** understand?
2. Why does the screen flicker when this program runs?

References

[cuda] *Online CUDA Information*, developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/index.html