Using Parallel Programming to Solve Complex Problems

Evan Barnett
Jordan Haskins
Outline

I. Serial Processing

II. Parallel Processing

III. CUDA

IV. Programming Examples
Serial Processing

• Traditionally, software has been written for *serial* computation

• Serial processing is processing that occurs sequentially.
  • Instructions are executed one after another
  • Only one instruction may execute at any moment in time
Serial Processing

www.computing.llnl.gov/tutorials/parallel_comp/
Moore’s Law

- The number of transistors doubles every two years
- Power Wall
Parallel Processing

• The simultaneous use of multiple compute resources to solve a computational problem

• The problem is broken into discrete parts that can be solved concurrently

• Each part is further broken down to a series of instructions

• Instructions from each part execute simultaneously on different CPUs and/or GPUs
Multiprocessors

• Started as multiprocessor systems that were typically used with servers

• Dual-Core and Multi-Core Processors
  • Multiple CPUs on the same chip that run in parallel by multi-threading
Parallel Processing

• Latency vs. Throughput

  • Goal: Higher Performing Processors
    • Minimize Latency
    • Increasing Throughput

• Latency
  • Time (e.g. Seconds)

• Throughput
  • Stuff/Time (e.g. Jobs/Hour)
GPUs

• GPUs are designed to handle parallel processing more efficiently.

• NVIDIA has introduced a way to harness the impressive computational power of the GPU.
CUDA

• CUDA (Compute Unified Device Architecture)
  • A parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).
  • Developed and released by NVIDIA in 2006
  • Can run on all of NVIDIA’s latest discrete GPUs
  • Extension of the C, C++, and Fortran languages
  • Operating Systems that Support it:
    • Windows XP and later, Linux, and Mac OS X
CUDA

- Restricted to certain hardware

- Nvidia cards:
  - compute capability: restrictions to coding depending on compute capability
  - 1.0-3.5

- Hardware we used during our research:
  - Nvidia GeForce 9600M GT: 1.1 compute capability
  - Nvidia GeForce GTX 580: 2.0 compute capability
CUDA Architecture

- By using CUDA functions, the CPU is able to utilize the parallelization capabilities of the GPU.
CUDA

• Kernel function: Runs on the GPU
  • `cudaFxn<<<grid,block>>>(int var, int var)`

• GPU hardware
  • Stream Multiprocessors(SMs)
    • Run in parallel and independently
    • Contains streaming processors and memory
    • Executes one warp per SM at a time
  • Streaming Processors(SPks)
    • Runs one thread each
CUDA

- Threads – Path of execution through program code
- Warps – collection of 32 threads
- Blocks – Made up of threads that communicate within their own block
- Grids – Made up of independent blocks
- Inside to out (threads -> blocks -> grids)
CUDA Functions

- `cudaMalloc(void** devicePointer, size)`
  - Allocates the size bytes of linear memory on the device and the devPtr points to the memory

- `cudaFree(devicePointer)`
  - Frees the memory from cudaMalloc()
CUDA Functions

• cudaMemcpy(destination, source, count, kind)
  • Copies the memory from one source to the destination specified by kind

• __syncthreads();
  • Called in the kernel function
  • Threads wait here until all threads in the same block reach that point
CUDA Memory

• GPU Memory Model
  • Local – Each thread has its own local memory
  • Shared – Each block has its own shared memory that can be accessed by any thread in that block
  • Global – shared by all blocks and threads and can be accessed at any time

• CPU Memory – called Host Memory
  • Normally data is copied to and from the Host Memory to the Global Memory on the GPU

• Speed
  • Local > Shared > Global > Host
CUDA applications

- Medical Imaging
- Computational Fluid Dynamics
- Image modification
- Numerical Calculations
Grayscale Conversion: Parallel Program

- CUDA program that converts a color image into a grayscale image
- Manipulating images and videos is a perfect task for the GPU
- Simultaneously convert each pixel into grayscale using:
  - \( Y' = 0.299R + 0.587G + 0.114B \)
- Conversion is much quicker in parallel compared to sequential
Grayscale Conversion: Parallel Program

- CUDA code of Grayscale conversion (written in C++)

```c++
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>
#include "utils.h"
#include <cuda.h>

__global__
void rgb_to_grayscale(const uchar4* const rgbImage,
                       unsigned char* const greyImage,
                       int numRows, int numCols)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i >= numCols || j >= numRows) {
        return;
    }
    int idx = j * numCols + i;
    uchar4 rgba = rgbImage[idx];
    greyImage[idx] = .299f * rgba.x + .587f * rgba.y + .114f * rgba.z;
}

void your_rgb_to_grayscale(uchar4* const d_rgbImage,
                            unsigned char* const d_greyImage, size_t numRows, size_t numCols)
{
    int M = 16, N = 16;
    const dim3 blockSize(M, N, 1); //TODO
    const dim3 gridSize( numCols/M + 1, numRows/N + 1, 1); //TODO
    rgb_to_grayscale<<<gridSize, blockSize>>>(d_rgbImage, d_greyImage, numRows, numCols);
    cudaDeviceSynchronize();
    checkCudaErrors(cudaGetLastError());
}
Accessible Population: Sequential Program

- **Purpose**: to determine the accessible population within a 300 km radius of most US cities

- Sequential code displaying the populating of the large dists array

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <conio.h>

#define SIZE 988

clock_t begin, end;
double time_spent;
double dists[SIZE * SIZE];
double coordinates[SIZE * 2];
char cities[SIZE * 30];
int topPop[SIZE];
int population[SIZE];
char temp[30];
char buffer[30];

int main(void)
{
    int i, j, n, check = 1, m, temp=0, k = 0, count = 0, popcount = 0;
    FILE *fp;

    for(row = 0; row < SIZE; row++)
    {
        for(col = 0; col < SIZE; col++)
        {
            dists[row * SIZE + col] =
            acos(sin(coordinates[row * 2 + 0]) * sin(coordinates[col * 2 + 0]) +
                cos(coordinates[row * 2 + 0]) * cos(coordinates[col * 2 + 0]) -
                cos(coordinates[col * 2 + 0]) * cos(coordinates[col * 2 + 1]) -
                coordinates[row * 2 + 1]) / 6371;
        }
    }
```
27 out of top 30 most accessible cities are in California

Jackson, TN ranks 827th out of 988
Accessible Population: Parallel Program

- Parallel code displaying the populating of the large dists array within the GPU's kernel

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <conio.h>
#include <cuda_runtime.h>
#include <cuda.h>
#include <device_launch_parameters.h>
#include <device_functions.h>

#define SIZE 998

// CUDA kernel
__global__ void createDistanceTable(float *d_distances, float *d_coordinates)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < SIZE && col < SIZE)
    d_distances[row * SIZE + col] =
    acos(sin(d_coordinates[row * 2 + 0]) * sin(d_coordinates[col * 2 + 0]) + cos(d_coordinates[row * 2 + 0]) * cos(d_coordinates[col * 2 + 0]) - 1) / 5371;
}

int main(void)
{
    int i, j, x, y, n, check = 1, m, temp=0;
    clock_t begin, end;
    double time_spent;
    float coordinates[SIZE * 2];
    char cities[SIZE][30];
    char buffer[30];
    int population[SIZE];
    int topPop[SIZE];
    char temp1[30];
    int popcount = 0;
    FILE *fp;

    float *d_distances;
    float *d_coordinates;
    float *dists;
```
Accessible Population: Parallel Program

- cudaMallocs for all memory that we need to allocate in the GPU kernel
- cudaMemcpy for the memory we need to transfer to kernel

```c
cudaError err;
err = cudaMalloc((void **) &d_distances, SIZE*SIZE*sizeof(float)); // distances
if(ccudaSuccess != err)
    fprintf( stderr, "CUDA error in file \%s' in line \%i : \%s.\n",
            __FILE__, __LINE__, cudaGetErrorString( err ) );

err = cudaMalloc((void **) &d_coordinates, SIZE*2*sizeof(float)); // coordinates
if(ccudaSuccess != err)
    fprintf( stderr, "CUDA error in file \%s' in line \%i : \%s.\n",
            __FILE__, __LINE__, cudaGetErrorString( err ) );

err = cudaMemcpy(d_coordinates, coordinates, SIZE*2*sizeof(float), cudaMemcpyHostToDevice);
if(c(cudaSuccess != err)
    fprintf( stderr, "CUDA error in file \%s' in line \%i : \%s.\n",
            __FILE__, __LINE__, cudaGetErrorString( err ) );

dim3 dimBlock(32,32,1);
dim3 dimGrid(32,32,1);
createDistanceTable<<<dimGrid, dimBlock>>>(d_distances, d_coordinates);
err = cudaMemcpy(dists, d_distances, SIZE*SIZE*sizeof(float), cudaMemcpyDeviceToHost);
if(c(cudaSuccess != err)
    fprintf( stderr, "CUDA error in file \%s' in line \%i : \%s.\n",
            __FILE__, __LINE__, cudaGetErrorString( err ) );

cudaFree(d_distances);
cudaFree(d_coordinates);
```

- cudaFree the memory in the GPU after the kernel runs
Accessible Population: Sequential Program

<table>
<thead>
<tr>
<th>N</th>
<th>N²</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10000</td>
<td>0.0</td>
</tr>
<tr>
<td>200</td>
<td>40000</td>
<td>3.8</td>
</tr>
<tr>
<td>600</td>
<td>360000</td>
<td>62.8</td>
</tr>
<tr>
<td>988</td>
<td>976144</td>
<td>167.3</td>
</tr>
<tr>
<td>1976</td>
<td>3904576</td>
<td>686.0</td>
</tr>
<tr>
<td>3952</td>
<td>15618304</td>
<td>2706.8</td>
</tr>
<tr>
<td>7904</td>
<td>62473216</td>
<td>10803.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>N²</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10000</td>
<td>105.0</td>
</tr>
<tr>
<td>200</td>
<td>40000</td>
<td>101.3</td>
</tr>
<tr>
<td>600</td>
<td>360000</td>
<td>132.5</td>
</tr>
<tr>
<td>988</td>
<td>976144</td>
<td>148.3</td>
</tr>
<tr>
<td>1976</td>
<td>3904576</td>
<td>347.5</td>
</tr>
<tr>
<td>3952</td>
<td>15618304</td>
<td>1493.8</td>
</tr>
<tr>
<td>7904</td>
<td>62473216</td>
<td>4789.0</td>
</tr>
</tbody>
</table>

- Small values of N: Sequential is the faster choice
- Larger values of N: Parallel becomes the much faster process
Accessible Population: Sequential Program
## Determinant of a Matrix using Minors

- Using the brute force method, this is a $N!$ complexity

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N!$</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0 ms</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>0 ms</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>0 ms</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>0 ms</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>0 ms</td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
<td>0 ms</td>
</tr>
<tr>
<td>8</td>
<td>40320</td>
<td>20 ms</td>
</tr>
<tr>
<td>9</td>
<td>362880</td>
<td>150 ms</td>
</tr>
<tr>
<td>10</td>
<td>3628800</td>
<td>1.4 secs</td>
</tr>
<tr>
<td>11</td>
<td>39916800</td>
<td>17 secs</td>
</tr>
<tr>
<td>12</td>
<td>479001600</td>
<td>3.8 mins</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N!$</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>6.2E+09</td>
<td>53.5 mins</td>
</tr>
<tr>
<td>14</td>
<td>8.7E+10</td>
<td>13.2 hours</td>
</tr>
<tr>
<td>15</td>
<td>1.3E+12</td>
<td>8.3 days</td>
</tr>
<tr>
<td>16</td>
<td>2.1E+13</td>
<td>132.6 days</td>
</tr>
<tr>
<td>17</td>
<td>3.6E+14</td>
<td>6.2 years</td>
</tr>
<tr>
<td>18</td>
<td>6.4E+15</td>
<td>1 century</td>
</tr>
<tr>
<td>19</td>
<td>1.2E+17</td>
<td>2 millennia</td>
</tr>
<tr>
<td>20</td>
<td>2.4E+18</td>
<td>42 millennia</td>
</tr>
<tr>
<td>21</td>
<td>5.1E+19</td>
<td>887 millennia</td>
</tr>
<tr>
<td>22</td>
<td>1.1E+21</td>
<td>2 epochs</td>
</tr>
<tr>
<td>23</td>
<td>2.6E+22</td>
<td>4.5 eras</td>
</tr>
<tr>
<td>24</td>
<td>6.2E+23</td>
<td>21.5 eons</td>
</tr>
</tbody>
</table>

- Using parallel coding, studies have shown a reduction in execution of time of > 40%!
Questions?

• Question and Answer Time