Parallel Programming using CUDA

Submitted By: UDI T AGRA WAL
ABSTRACT

- Developing a code for fast interpolation for image rotation, required in the numeric’s of a novel 3-degree-of-freedom optical sensor.
OBJECTIVE

• Developing algorithm for image rotation on CPU and then deploying the algorithm to work on GPU parallel threads using CUDA.

Implementation details:
• NVIDIA graphics card (240-core GPU) to be installed on a PC.
• CUDA device driver
• CUDA Software Development Kit
  • Emulator
• CUDA Toolkit.
• Developing algorithm to run on parallel architecture.
Why Parallel Computing??

- Saves time – wall clock time
- Cost savings
- Overcoming memory constraints
- It’s the future of computing
But the problem is...

You will always have more data than cores
ENTER THE GPU

• A graphics processing unit or GPU is a specialized processor that offloads 3D or 2D graphics rendering from the microprocessor.

• Massively parallel.

• Used in embedded systems, mobile phones, personal computers, workstations, and game consoles.
GPU COMPUTING

- Uses graphic hardware for non-graphic computations.

- The excellent floating point performance in GPUs led to the advent of General Purpose Computing on GPU.

- The model for GPU computing is to use a CPU and GPU together in a heterogeneous computing model.
Problems in GPU Programming

- Required graphical languages
- Difficult for users to program applications for GPU
ENTER CUDA

- CUDA is an acronym for Compute Unified device Architecture

- A parallel computing architecture for computing on NVIDIA GPUs.

- CUDA
  - accelerates the computational horsepower of NVIDIA GPUs.
  - Enable general-purpose GPU computing
ADVANTAGES OF USING CUDA

CUDA has following advantages over traditional GPGPU using graphics APIs.

- Shared memory.
- Scattered reads.
- Faster downloads and read backs to and from the GPU.
CUDA PROGRAMMING MODEL

- CUDA with industry-standard C
  - Write a program for one thread
  - Instantiate it on many parallel threads
  - Familiar programming model and language

- CUDA is a scalable parallel programming model
  - Program runs on any number of processors without recompiling

- Threads grouped into thread blocks.
- Grid = all blocks for a given launch.
Processing flow on CUDA

1. Copy processing data
2. Instruct the processing
3. Execute parallel in each core
4. Copy the result
A usual method is a geometry transformation which rotates angle (theta) around center of the image. For example, the original pixel p [x, y] rotates to p’[x’, y’].

A rotation matrix is a matrix that is used to perform a rotation in Euclidean space.

This rotates column vectors by means of the following matrix multiplication:

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix} = \begin{bmatrix}
  \cos \theta & -\sin \theta \\
  \sin \theta & \cos \theta
\end{bmatrix} \begin{bmatrix}
  x \\
  y
\end{bmatrix}
\]
So the coordinates \((x',y')\) of the point \((x,y)\) after rotation are:

\[
x' = x \cos \theta - y \sin \theta
\]
\[
y' = x \sin \theta + y \cos \theta
\]

But some rotations may cause zoom effect because some rotated pixels are not integer. In this case interpolation can be used to figure out the rotated pixel to reduce distortion.
BILINEAR INTERPOLATION

- Bilinear interpolation considers the closest 2x2 neighborhood of known pixel values surrounding the unknown pixel.
- It then takes a weighted average of these 4 pixels to arrive at its final interpolated value. This results in much smoother looking images than nearest neighbor.
IMPLEMENTATION IN CUDA

- The index of a thread and its thread ID relate to each other in a straightforward way:
  For a two-dimensional block of size \((Dx, Dy)\).

\[
\_\text{global\_void Rotate(float Source}[N][N], \text{float Destination}[N][N])
\{
    // Kernel definition
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
    {
        If((i*cos(Q) - j*sin(Q)) not integer || (i*sin(Q) + j*cos(Q) not integer)
        Do
        Interpolation
        else
        Destination[i*\cos(Q) - j*\sin(Q)][i*\sin(Q) + j*\cos(Q)] = Source[i][j];
    }
}
\]
int main()
{
    ...

    // Kernel invocation
    // Indicate the dimension of the block
    dim3 dimBlock(16, 16);
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x, (N + dimBlock.y - 1) / dimBlock.y);
    Rotate<<<dimGrid, dimBlock>>>(Source, Destination);
}

RESULTS & LIMITATIONS

- This is much faster clock-for-clock on a GeForce 8800 GTX GPU than on a CPU.
  - runs more than 50 times as fast as a highly tuned serial implementation.
  - 250 times faster than our portable C implementation.

But there were limitations:

- Copying between host and device memory may incur a performance hit due to system bus bandwidth and latency.
- Maximum number of threads per block: 512, therefore concurrency is limited.
thank you..!!