Heterogeneous Computing with MATLAB

ECE 6397
Electrical and Computer Engineering
University of Houston
Dr. David Mayerich
CPU Parallelism in MATLAB

- Parallel Computing Toolbox provides options for both task and data parallelism
- `parfor` – data parallelism using FOR loops

```matlab
N = 10000; % number of K x K matrices in an array
for i = 1:N
    [V(i, :, :), D(i, :, :)] = eig(A(i, :, :));
end
```

```matlab
% perform the same operation in parallel
parfor i = 1:N
    [V(i, :, :), D(i, :, :)] = eig(A(i, :, :));
end
```
Controlling Resources with `parfor`

- The number of separate threads can be specified:

```matlab
P = 5;
parfor (i = 1:N, P)  %set the max number of threads P
    [V(i, :, :), D(i, :, :)] = eig(A(i, :, :));
end
```
Parallel Evaluation of Independent Functions

- `parfeval(func, num_outputs, input_1, input_2, ..., input_n)`

```matlab
function R = simulation(X, Y, Z, param)
    %define a function

    [X, Y, Z] = meshgrid(x, y, z);
    %define an array p of N parameters for N simulations:
    %p = [...]
    for i = 1:N
        r(i) = parfeval(@simulation, 1, X, Y, Z, p(i));
    end
```
MATLAB for GPU Computing – Requirements

• Requires compute capability 2.0 or higher

• MATLAB arrays are stored either on the CPU or GPU
  • the CPU is the default location
  • try to keep the data on the GPU – data transfers often limit large applications
  • many functions are designed to take GPU arrays:
  • sin, cos, tan, fft, ifft, legendre, median, svd, eig, etc.
## GPU Device Properties in MATLAB

Device capabilities can be acquired using the `gpuDevice` function with a `parallel.gpu.CUDADevice` structure:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>'GeForce GTX 970'</td>
</tr>
<tr>
<td>Index</td>
<td>1</td>
</tr>
<tr>
<td>ComputeCapability</td>
<td>'5.2'</td>
</tr>
<tr>
<td>SupportsDouble</td>
<td>1</td>
</tr>
<tr>
<td>DriverVersion</td>
<td>8</td>
</tr>
<tr>
<td>ToolkitVersion</td>
<td>5.5000</td>
</tr>
<tr>
<td>MaxThreadsPerBlock</td>
<td>1024</td>
</tr>
<tr>
<td>MaxShmemPerBlock</td>
<td>49152</td>
</tr>
<tr>
<td>MaxThreadBlockSize</td>
<td>[1024 1024 64]</td>
</tr>
<tr>
<td>MaxGridSize</td>
<td>[2.1475e+09 65535 65535]</td>
</tr>
<tr>
<td>SIMDWidth</td>
<td>32</td>
</tr>
<tr>
<td>TotalMemory</td>
<td>4.2950e+09</td>
</tr>
<tr>
<td>FreeMemory</td>
<td>3.5151e+09</td>
</tr>
<tr>
<td>MultiprocessorCount</td>
<td>13</td>
</tr>
<tr>
<td>ClockRateKHz</td>
<td>1329000</td>
</tr>
<tr>
<td>ComputeMode</td>
<td>'Default'</td>
</tr>
<tr>
<td>GPUOverlapsTransfers</td>
<td>1</td>
</tr>
<tr>
<td>KernelExecutionTimeout</td>
<td>1</td>
</tr>
<tr>
<td>CanMapHostMemory</td>
<td>1</td>
</tr>
<tr>
<td>DeviceSupported</td>
<td>1</td>
</tr>
<tr>
<td>DeviceSelected</td>
<td>1</td>
</tr>
</tbody>
</table>
Allocating Memory

- Data is initialized on the CPU and copied to the GPU
- Use the `gpuArray()` function to make the host → device copy
  - essentially performs a `cudaMemcpy()` with `cudaMemcpyHostToDevice`

```matlab
>> A = rand(2000, 2000); %create a matrix of numbers
>> gpu_A = gpuArray(A); %copy them to the GPU
```

- Use the `gather()` function to make the device → host copy

```matlab
>> gpu_B = fft(gpu_A); %perform a gpu-based FFT
>> B = gather(gpu_B); %copy the result to the host
```
Gradient Calculation - CPU

- Calculate the gradient of a large image of Haverford Cathedral

\[
\sigma = 20; \quad \text{\%standard deviation of the blur}
\]
\[
x = -3\sigma:3\sigma; \quad \text{\%coordinate of the blur kernel}
\]
\[
k = \text{normpdf}(x, 0, \sigma); \quad \text{\%calculate the 1D blur kernel}
\]
\[
K = k' * k; \quad \text{\%outer product creates the 2D kernel}
\]

\[
I = \text{imread('haverford.ppm')}; \quad \text{\%load the image}
\]
\[
Ig = \text{single(rgb2gray(I))}; \quad \text{\%convert to grayscale, 32-bit}
\]
\[
Ic = \text{conv2(Ig, K, 'valid')}; \quad \text{\%convolve the image}
\]
\[
[dIdx, dIdy] = \text{gradient(Ic)}; \quad \text{\%calculate the gradient}
\]
Gradient Calculation - Results

∇I = \frac{dI}{dx}(x, y)

∇I = \frac{dI}{dy}(x, y)
Gradient Calculation – CPU Profiling

<table>
<thead>
<tr>
<th>algorithm</th>
<th>$\sigma = 10$</th>
<th>$\sigma = 20$</th>
<th>$\sigma = 40$</th>
<th>$\sigma = 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td>%</td>
<td>s</td>
<td>%</td>
</tr>
<tr>
<td>imread()</td>
<td>4.50</td>
<td>22.6</td>
<td>4.62</td>
<td>4.6</td>
</tr>
<tr>
<td>rgb2gray()</td>
<td>0.24</td>
<td>1.2</td>
<td>0.32</td>
<td>0.5</td>
</tr>
<tr>
<td>conv2()</td>
<td>13.8</td>
<td>69.6</td>
<td>63.1</td>
<td>91</td>
</tr>
<tr>
<td>gradient()</td>
<td>1.20</td>
<td>6.0</td>
<td>1.11</td>
<td>1.6</td>
</tr>
<tr>
<td>total</td>
<td>19.7</td>
<td>69.2</td>
<td>252</td>
<td>1238</td>
</tr>
</tbody>
</table>
Gradient Calculation – CPU Profiling

- Convolution operation is the bottleneck
  - Not surprising – $O(n\sigma^2)$ operation, all others are $O(n)$

- All calculations are data parallel

- Once `imread()` is the bottleneck, we’re basically done

<table>
<thead>
<tr>
<th>algorithm</th>
<th>$\sigma = 10$</th>
<th>$\sigma = 20$</th>
<th>$\sigma = 40$</th>
<th>$\sigma = 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td>%</td>
<td>s</td>
<td>%</td>
</tr>
<tr>
<td><code>imread()</code></td>
<td>4.50</td>
<td>22.6</td>
<td>4.62</td>
<td>4.6</td>
</tr>
<tr>
<td><code>rgb2gray()</code></td>
<td>0.24</td>
<td>1.2</td>
<td>0.32</td>
<td>0.5</td>
</tr>
<tr>
<td><code>conv2()</code></td>
<td>13.8</td>
<td>69.6</td>
<td>63.1</td>
<td>91</td>
</tr>
<tr>
<td><code>gradient()</code></td>
<td>1.20</td>
<td>6.0</td>
<td>1.11</td>
<td>1.6</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>19.7</td>
<td>69.2</td>
<td>252</td>
<td></td>
</tr>
</tbody>
</table>
Gradient Calculation – GPU

sigma = 20; %standard deviation of the blur
x = -3*sigma:3*sigma; %coordinate of the blur kernel
k = normpdf(x, 0, sigma); %calculate the 1D blur kernel
K = k' * k; %outer product creates the 2D kernel
gpu_K = gpuArray(K); %copy the kernel to the GPU
I = imread('hereford.png'); %load the image
gpu_I = gpuArray(I); %copy the image to the GPU
gpu_Ig = single(rgb2gray(gpu_I)); %convert to grayscale, 32-bit
gpu_Ic = conv2(gpu_Ig, gpu_K, 'valid'); %convolve the image
[gpu_dIdx, gpu_dIdy] = gradient(gpu_Ic); %calculate the gradient
dIdx = gather(gpu_dIdx); %copy the gradients to the CPU
dIdy = gather(gpu_dIdy);
Gradient Calculation – GPU Profiling

<table>
<thead>
<tr>
<th>algorithm</th>
<th>$\sigma = 10$</th>
<th>$\sigma = 20$</th>
<th>$\sigma = 40$</th>
<th>$\sigma = 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td>%</td>
<td>s</td>
<td>%</td>
</tr>
<tr>
<td>imread()</td>
<td>4.51</td>
<td>81.2</td>
<td>4.49</td>
<td>62.4</td>
</tr>
<tr>
<td>rgb2gray()</td>
<td>0.13</td>
<td>2.3</td>
<td>0.12</td>
<td>1.7</td>
</tr>
<tr>
<td>conv2()</td>
<td>0.02</td>
<td>0.4</td>
<td>0.02</td>
<td>0.3</td>
</tr>
<tr>
<td>gradient()</td>
<td>0.89</td>
<td>16.0</td>
<td>2.56</td>
<td>35.6</td>
</tr>
<tr>
<td>total</td>
<td>5.55</td>
<td>7.19</td>
<td>13.5</td>
<td>55.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>algorithm</th>
<th>$\sigma = 10$</th>
<th>$\sigma = 20$</th>
<th>$\sigma = 40$</th>
<th>$\sigma = 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td>%</td>
<td>s</td>
<td>%</td>
</tr>
<tr>
<td>imread()</td>
<td>4.50</td>
<td>22.6</td>
<td>4.62</td>
<td>4.6</td>
</tr>
<tr>
<td>rgb2gray()</td>
<td>0.24</td>
<td>1.2</td>
<td>0.32</td>
<td>0.5</td>
</tr>
<tr>
<td>conv2()</td>
<td>13.8</td>
<td>69.6</td>
<td>63.1</td>
<td>91</td>
</tr>
<tr>
<td>gradient()</td>
<td>1.20</td>
<td>6.0</td>
<td>1.11</td>
<td>1.6</td>
</tr>
<tr>
<td>total</td>
<td>19.7</td>
<td>69.2</td>
<td>252</td>
<td>1238</td>
</tr>
</tbody>
</table>

$\approx 20 \times$ speedup
Running CUDA Kernels from MATLAB

- MATLAB allows the execution of CUDA kernels
- The kernel must be compiled to PTX using `nvcc`
- Create a MATLAB CUDAKernel object from the ptx and cu files

Compile the CUDA kernel to PTX code
```
>> nvcc -ptx mykernel.cu
```

Create a CUDAKernel object in MATLAB
```
k = parallel.gpu.CUDAKernel('mykernel.ptx', 'mykernel.cu');
```

Execute the kernel as a function
```
R = feval(k, p0, p1, ...); %returns a gpuArray with the result
Creating a Kernel for MATLAB Use

• A kernel is written in CUDA as a __global__ function

```
//add_array.cu
//-----------------------------------------------------------------------------
//CUDA function adds a constant value to an array
__global__ void add_array(float* a, const float c, const int N){
  //calculate the grid index into the array
  int i = blockIdx.x * blockDim.x + threadIdx.x
  if(i > N) return;  //return if outside the array
  a[i] += c;  //add c to the array element
}
```
Create a Kernel Object in MATLAB

• Compile the kernel using nvcc at the OS command prompt

\[
\text{nvcc -ptx add_array.cu}
\]

  • creates a .ptx file with file name identical to the source file name
  • creates add_array.ptx

• Create a CUDAKernel object in MATLAB and configure

```matlab
%create a GPU kernel k
k = parallel.gpu.CUDAKernel('add_array.ptx', 'add_array.cu);
k.ThreadBlockSize = 1024;  %set # of threads per block
k.GridSize = ceil(N/k.ThreadBlockSize);  %set # of blocks
gpu_x = gpuArray(x);    %copy input to GPU
gpu_y = feval(k, gpu_x, c, N);  %launch kernel
y = gather(gpu_y);       %copy result to GPU
```
Using `feval()`

`feval(KERN, p1, p2, ..., pn)`

- Launches a kernel given by the code and configuration in `KERN`
- Parameters are specified in the same order given in the `.cu` file
- Parameters not specified as `const` in the `.cu` file are returned
  - they are returned in the same order as they are specified
  - it is generally convenient to make the first non-`const` parameter the output

```c
__global__ void add_array(float* a, const float c, const int N)
>> gpu_a = feval(add_array, gpu_a, c, N);

__global__ void mykernel(float* a, const float* b, float* c)
>> [gpu_a, gpu_c] = feval(mykernel, gpu_a, gpu_b, gpu_c);
```