E6895 Advanced Big Data Analytics Lecture 14:

NVIDIA GPU Examples and GPU on iOS devices

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
IBM Chief Scientist, Graph Computing Research
Outline

• GPU Architecture
  • Execution Model
    • Examples in tuning configurations of grid and thread
  • Resource Allocation
    • Examples in examining the affect of configuration setting in grid and thread
• Memory Architecture
• Concurrent Processing
• cuBLAS Library for Neural Network
GPU Architecture

built by several streaming multiprocessors (SMs)

In each SM:
- CUDA cores
- Shared Memory/L1 Cache
- Register File
- Load/Store Units
- Special Function Units
- Warp Scheduler

In each device:
- L2 Cache
Execution Model - Warp Concept

NVIDIA GPU groups 32 threads into a warp, and then execute warps sequentially.

- Number of concurrent warps are based on the number of warp scheduler. (Kepler has 4 warp scheduler)
- All threads are arranged one-dimensionally.

Relationship between logical view and hardware view

Inefficient way to allocate a thread block
Execution Model - Warp Divergence

GPU has light-weight control module, complicated *control flow* might hurt the performance of GPU.

In the same warp, e.g., if you allocate 16 threads to do A; and 16 threads to do B.

A and B will be executed serially.

Example: simpleDivergence.cu
- Test with optimization and without optimization
Execution Model - Warp Divergence

compilation — turn off the optimization

```
nvcc -g -G -o simpleDivergence simpleDivergence.cu
```

execute through nvprof to extract profiling information

```
nvprof --metrics branch_efficiency --events branch,divergent_branch .
    simpleDivergence
```

---

```
<table>
<thead>
<tr>
<th>Invocations</th>
<th>Event Name</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;GeForce GT 650M (0)&quot;</td>
<td>branch</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Kernel: mathKernel1(float*)</td>
<td>branch</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Kernel: mathKernel2(float*)</td>
<td>branch</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Kernel: mathKernel3(float*)</td>
<td>branch</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Kernel: mathKernel4(float*)</td>
<td>branch</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Kernel: warmingup(float*)</td>
<td>branch</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

===99767== Metric result:

```
<table>
<thead>
<tr>
<th>Invocations</th>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;GeForce GT 650M (0)&quot;</td>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>80.00%</td>
<td>80.00%</td>
<td>80.00%</td>
</tr>
<tr>
<td>Kernel: mathKernel1(float*)</td>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Kernel: mathKernel2(float*)</td>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>66.67%</td>
<td>66.67%</td>
<td>66.67%</td>
</tr>
<tr>
<td>Kernel: mathKernel3(float*)</td>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Kernel: mathKernel4(float*)</td>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Kernel: warmingup(float*)</td>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
</tbody>
</table>
```
Resource Allocation

A warp is consisted of
- Program counters
- Registers
- Shared memory

If a thread uses lots of resources, fewer threads will be allocated in one SM.
Toolkit, CUDA Occupancy Calculator:
in /usr/local/cuda/tools/CUDA_Occupancy_Calculator.xls
It could assist in measuring the occupancy of your configuration
Occupancy, Memory Load Efficiency, Memory Load Throughput

View number of registers, set the constraints on number of registers per thread

```
    nvcc -g -G -arch=sm_30 --ptxas-options=-v --maxrregcount=31 -o sumMatrix
    sumMatrix.cu
```

Check occupancy, memory load efficiency, memory load throughput to explore the suitable configuration of size of thread block

```
    nvprof --metrics gld_throughput,gld_efficiency,achieved_occupancy ./sumMatrix dimX dimY
```

do example on sumMatrix.cu with dim of thread block {4,4}, {4,8}, {8,4}, {8,8}, {16,16}, {32,32} {16,16} is the fast one.
### Profiling Results

#### sumMatrixOnGPU2D <<<(1024,1024), (4,4)>>> elapsed 498.796492 ms

- Profiling application: ./sumMatrix 4 4
- Profiling result:

<table>
<thead>
<tr>
<th>Invocations</th>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;GeForce GT 650M (0)&quot;</td>
<td>Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)</td>
<td>Global Load Throughput</td>
<td>2.5676GB/s</td>
<td>2.5912GB/s</td>
<td>2.5883GB/s</td>
</tr>
<tr>
<td>100</td>
<td>gld_throughput</td>
<td>Global Memory Load Efficiency</td>
<td>50.00%</td>
<td>50.00%</td>
<td>50.00%</td>
</tr>
<tr>
<td>100</td>
<td>gld_efficiency</td>
<td>Achieved Occupancy</td>
<td>0.224382</td>
<td>0.224437</td>
<td>0.224409</td>
</tr>
</tbody>
</table>

#### sumMatrixOnGPU2D <<<(512,512), (8,8)>>> elapsed 209.860447 ms

- Profiling application: ./sumMatrix 8 8
- Profiling result:

<table>
<thead>
<tr>
<th>Invocations</th>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;GeForce GT 650M (0)&quot;</td>
<td>Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)</td>
<td>Global Load Throughput</td>
<td>3.4194GB/s</td>
<td>8.1859GB/s</td>
<td>3.9721GB/s</td>
</tr>
<tr>
<td>100</td>
<td>gld_throughput</td>
<td>Global Memory Load Efficiency</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>100</td>
<td>gld_efficiency</td>
<td>Achieved Occupancy</td>
<td>0.446429</td>
<td>0.451606</td>
<td>0.447626</td>
</tr>
</tbody>
</table>

#### sumMatrixOnGPU2D <<<(256,256), (16,16)>>> elapsed 154.986583 ms

- Profiling application: ./sumMatrix 16 16
- Profiling result:

<table>
<thead>
<tr>
<th>Invocations</th>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;GeForce GT 650M (0)&quot;</td>
<td>Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)</td>
<td>Global Load Throughput</td>
<td>4.8073GB/s</td>
<td>5.0724GB/s</td>
<td>4.9692GB/s</td>
</tr>
<tr>
<td>100</td>
<td>gld_throughput</td>
<td>Global Memory Load Efficiency</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>100</td>
<td>gld_efficiency</td>
<td>Achieved Occupancy</td>
<td>0.821689</td>
<td>0.847440</td>
<td>0.845807</td>
</tr>
</tbody>
</table>

#### sumMatrixOnGPU2D <<<(128,128), (32,32)>>> elapsed 182.770676 ms

- Profiling application: ./sumMatrix 32 32
- Profiling result:

<table>
<thead>
<tr>
<th>Invocations</th>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;GeForce GT 650M (0)&quot;</td>
<td>Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)</td>
<td>Global Load Throughput</td>
<td>4.6957GB/s</td>
<td>5.0193GB/s</td>
<td>4.8495GB/s</td>
</tr>
<tr>
<td>100</td>
<td>gld_throughput</td>
<td>Global Memory Load Efficiency</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>100</td>
<td>gld_efficiency</td>
<td>Achieved Occupancy</td>
<td>0.753967</td>
<td>0.776110</td>
<td>0.772071</td>
</tr>
</tbody>
</table>
Memory Type

1. Register
   • per SM, An automatic variable in kernel function, *low latency, high bandwidth*

2. Local memory
   • Variable in a kernel but can not be fitted in register
   • Cached in per SM L1 cache and per device L2 cache

3. Shared memory (__shared__)
   • per SM, faster than local and global memory, share among thread block
   • Use for inter-thread communication, 64KB, shared with L1 cache

4. Constant memory (__constant__)
   • per device, read-only memory

5. Texture memory
   • perSM, read-only cache, optimized for 2D spatial locality

6. Global memory
## Memory Type

<table>
<thead>
<tr>
<th>MEMORY</th>
<th>ON/OFF CHIP</th>
<th>CACHED</th>
<th>ACCESS</th>
<th>SCOPE</th>
<th>LIFETIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off</td>
<td>†</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>All threads in block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off</td>
<td>†</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Constant</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Texture</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
</tbody>
</table>

† Cached only on devices of compute capability 2.x
Concurrent Processing

Concurrent handle

- **data transfer + computation**
  
  For NVIDIA GT 650M (laptop GPU), there is one copy engine. 
  For NVIDIA Tesla K40 (high-end GPU), there are two copy engines

- **computation + computation** if your CUDA core is enough.

Check example, sumMatrixOnGPUSstream.cu and sumMatrixOnGPUNoStream.cu

The latency in data transfer could be hidden during computing

To handle two tasks, which both are matrix multiplications.

  Copy two inputs to GPU, copy one output from GPU
### Concurrent Processing

#### No concurrent processing

#### Concurrent processing (concurrent computation)

#### Concurrent processing (transfer + computation)
In neural network, the most important operation is \textit{inner-product}.

\[
a = f(x^Tw + b)
\]

\(x\) is a matrix that records the input which is fed to neural network
\(w\) is a matrix that records the weights of network connection
\(b\) is a matrix that records the bias of network connection
\(f\) is an activation function that used to activate the neuron
\(a\) is output

GPU is more suitable for such intensive operations.

Example, \(x^Tw + b\)

\text{cuBLAS (GPU) vs. OpenBLAS (CPU)}

\text{GPU computation includes data transfer between host and device.}

\text{GPU compute a (4096,4096) matrix, spent 0.810978 secs}
\text{CPU compute a (4096,4096) matrix, spent 3.347512 secs}
Check all metrics and events for nvprof, it will also explain the meaning of options

```
nvprof --query-metrics
nvprof --query-events
```

Professional CUDA C Programming


source code are available on the above website

GTC On-Demand:


Developer Zone:


NVIDIA Parallel Programming Blog:


NVIDIA Developer Zone Forums:

[http://devtalk.nvidia.com](http://devtalk.nvidia.com)
Metal Programming Model

It integrates the support for both **graphics** and **compute** operations.

Three command encoder:
- Graphics Rendering: Render Command Encoder
- Data-Parallel Compute Processing: Compute Command Encoder
- Transfer Data between Resource: Blitting Command Encoder

Multi-threading in encoding command is supported

Typical flow in compute command encoder
- Prepare data
- Put your function into pipeline
- Command encoder
- Put command into command buffer
- Commit it to command queue
- Execute the command
- Get result back
Metal Programming, Kernel Function

Compute command

- Two parameters, threadsPerGroup and numThreadgroups, determine the number of threads. ➔ equivalent to grid and thread block in CUDA. They are all 3-D variables.
- The total of all threadgroup memory allocations must not exceed 16 KB.

- Kernel function: sigmoid function

```c
kernel void sigmoid(const device float *inVector [[ buffer(0) ]],
    device float *outVector [[ buffer(1) ]],
    uint id [[ thread_position_in_grid ]]) {
    // This calculates sigmoid for _one_ position (=id) in a vector per call on the GPU
    outVector[id] = 1.0 / (1.0 + exp(-inVector[id]));
}
```
Example

```swift
// Initial function to handle Metal
func initMetal() -> (MTLDevice, MTLCommandQueue, MTLLibrary, MTLCommandBuffer, MTLComputeCommandEncoder){
    // Get access to iPhone or iPad GPU
    var device = MTLCreateSystemDefaultDevice()

    // Queue to handle an ordered list of command buffers
    var commandQueue = device.newCommandQueue()

    // Access to Metal functions that are stored in Shaders.metal file, e.g. sigmoid()
    var defaultLibrary = device.newDefaultLibrary()

    // Buffer for storing encoded commands that are sent to GPU
    var commandBuffer = commandQueue.commandBuffer()

    // Encoder for GPU commands
    var computeCommandEncoder = commandBuffer.computeCommandEncoder()

    return (device, commandQueue, defaultLibrary!, commandBuffer, computeCommandEncoder)
}
```

// Set up a compute pipeline with Sigmoid function and add it to encoder
```
let sigmoidProgram = defaultLibrary.newFunctionWithName("sigmoid")
var pipelineErrors = NSErrorPointer()
var computePipelineFilter = device.newComputePipelineStateWithFunction(sigmoidProgram!, error: pipelineErrors)
computeCommandEncoder.setComputePipelineState(computePipelineFilter!)
```

// Set the input vector for the Sigmoid() function, e.g. inVector
```
var inVectorBufferNoCopy = device.newBufferWithBytesNoCopy(memory, length: Int(size), options: nil, deallocator: nil)
computeCommandEncoder.setBuffer(inVectorBufferNoCopy, offset: 0, atIndex: 0)
```

// Create the output vector for the Sigmoid() function, e.g. outVector
```
var outVectorBufferNoCopy = device.newBufferWithBytesNoCopy(outmemory, length: Int(size), options: nil, deallocator: nil)
computeCommandEncoder.setBuffer(outVectorBufferNoCopy, offset: 0, atIndex: 1)
```
Example

```javascript
// hardcoded to 32 for now (recommendation: read about threadExecutionWidth)
var threadsPerGroup = MTLSIZE(width:32, height:1, depth:1)
var numThreadgroups = MTLSIZE(width:(Int(maxcount)+31)/32, height:1, depth:1)
computeCommandEncoder.dispatchThreadgroups(numThreadgroups, threadsPerThreadgroup: threadsPerGroup)
computeCommandEncoder.endEncoding()

commandBuffer.commit()
commandBuffer.waitUntilCompleted()
```
Example: deviceQuery

Understand the hardware constraint via deviceQuery (in example code of CUDA toolkit)

<table>
<thead>
<tr>
<th>Device 0: &quot;GeForce GT 650M&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Driver Version / Runtime Version</td>
</tr>
<tr>
<td>CUDA Capability Major/Minor version number</td>
</tr>
<tr>
<td>Total amount of global memory</td>
</tr>
<tr>
<td>( 2) Multiprocessors, (192) CUDA Cores/MP</td>
</tr>
<tr>
<td>GPU Max Clock rate</td>
</tr>
<tr>
<td>Memory Clock rate</td>
</tr>
<tr>
<td>Memory Bus Width</td>
</tr>
<tr>
<td>L2 Cache Size</td>
</tr>
<tr>
<td>Maximum Texture Dimension Size (x,y,z)</td>
</tr>
<tr>
<td>Maximum Layered 1D Texture Size, (num) layers</td>
</tr>
<tr>
<td>Maximum Layered 2D Texture Size, (num) layers</td>
</tr>
<tr>
<td>Total amount of constant memory</td>
</tr>
<tr>
<td>Total amount of shared memory per block</td>
</tr>
<tr>
<td>Total number of registers available per block</td>
</tr>
<tr>
<td>Warp size</td>
</tr>
<tr>
<td>Maximum number of threads per multiprocessor</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
</tr>
<tr>
<td>Max dimension size of a thread block (x,y,z)</td>
</tr>
<tr>
<td>Max dimension size of a grid size (x,y,z)</td>
</tr>
<tr>
<td>Maximum memory pitch</td>
</tr>
<tr>
<td>Texture alignment</td>
</tr>
<tr>
<td>Concurrent copy and kernel execution</td>
</tr>
<tr>
<td>Run time limit on kernels</td>
</tr>
<tr>
<td>Integrated GPU sharing Host Memory</td>
</tr>
<tr>
<td>Support host page-locked memory mapping</td>
</tr>
<tr>
<td>Alignment requirement for Surfaces</td>
</tr>
<tr>
<td>Device has ECC support</td>
</tr>
<tr>
<td>Device supports Unified Addressing (UVA)</td>
</tr>
<tr>
<td>Device PCI Domain ID / Bus ID / location ID</td>
</tr>
</tbody>
</table>
Homework #3 (due May 22)

Choose 2 of the algorithms you use in your final project. Convert them into a GPU version.

Show your code, and performance measurement comparing to your non-GPU version.

The more innovative/complex your algorithms are, the higher score you will get on your Homework #3.