E6895 Advanced Big Data Analytics Lecture 9:

Deep Dive in NVIDIA GPU and GPU on iOS devices

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
IBM Chief Scientist, Graph Computing
Guest Speaker: Chun-Fu (Richard) Chen
Outline

NVIDIA GPU Architecture
  Execution Model
  Resource Allocation
  Memory Type
  Concurrent Processing

Applications on NVIDIA GPU:
  Mandatory Component in Machine Learning:
    Matrix Multiplication with Addition \(Y = A^\top B + C\)

GPU Architecture on iPhone/iPad
  Execution Model
  Metal Programming Examples for Data-Parallel Computation on GPU
    Sigmoid Function
    Sobel Operators for Image Processing
GPU Architecture

built by several Streaming Multiprocessors (SMs)
Single Instruction Multiple Threads (SIMT) 

Every thread behaves identically but data for threads are different.

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

In each device:
L2 Cache
Global Memory

Kepler Architecture, K20X
NVIDIA GPU groups 32 threads into one warp, and then execute warps *sequentially*. Number of *concurrent* warps are based on the number of warp scheduler. (Kepler has 4 warp scheduler)

Relationship between logical view and hardware view

Inefficient way to allocate a thread block: thread number in one block is not multiples of 32.
Execution Model - Warp Divergence

GPU has light-weight control module, complicated control flow will hurt the performance of GPU. In the same warp, e.g., if you allocate 16 threads to do A task; and 16 threads to do B task. A and B will be executed serially.

Example: simpleDivergence.cu
- Test with optimization (-O2) and without optimization (-g)
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></td>
<td>branch</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>divergent_branch</td>
<td>2</td>
<td>2</td>
<td>2</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>divergent_branch</td>
<td>0</td>
<td>0</td>
<td>0</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>divergent_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>
<tr>
<td>Kernel: warmingup(float*)</td>
<td>divergent_branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Metric result:**

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>80.00%</td>
<td>80.00%</td>
<td>80.00%</td>
</tr>
<tr>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>66.67%</td>
<td>66.67%</td>
<td>66.67%</td>
</tr>
<tr>
<td>branch_efficiency</td>
<td>Branch Efficiency</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<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 use lots of resource, fewer threads will be allocated in one SM.

- Registers per SM
  - Kepler: 64K
  - Fermi: 32K

- Shared Memory per SM
  - Kepler: up to 48K
  - Fermi: up to 48K

- More threads with fewer registers per thread
- Fewer threads with more registers per thread
- More blocks with less shared memory per block
- Fewer blocks with more shared memory per block
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

<table>
<thead>
<tr>
<th>Device</th>
<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>&quot;GeForce GT 650M (0)&quot;</td>
<td>Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)</td>
<td>gld_throughput</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>
<td></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>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device</th>
<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>&quot;GeForce GT 650M (0)&quot;</td>
<td>Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)</td>
<td>gld_throughput</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>
<td></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>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device</th>
<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>&quot;GeForce GT 650M (0)&quot;</td>
<td>Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)</td>
<td>gld_throughput</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>
<td></td>
</tr>
<tr>
<td>100</td>
<td>gld_efficiency</td>
<td>Achieved Occupancy</td>
<td>0.821609</td>
<td>0.847440</td>
<td>0.845807</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device</th>
<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>&quot;GeForce GT 650M (0)&quot;</td>
<td>Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)</td>
<td>gld_throughput</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>
<td></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>
<td></td>
</tr>
</tbody>
</table>
Memory Type

1. Register
   • per thread, An automatic variable in kernel function, \textit{low latency, high bandwidth}

2. Local memory
   • per thread, variable in a kernel but can not be fitted in register

3. Shared memory (\texttt{__shared__})
   • all threads, faster than local and global memory, share among thread blocks
   • Use for \texttt{inter-thread communication}, 64KB, physically shared with L1 cache

4. Constant memory (\texttt{__constant__})
   • per device, read-only memory

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

6. Global memory
<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 750M (laptop GPU), there is one copy engine.
  For NVIDIA Tesla K40 (high-end server GPU), there are two copy engines

- **Computation** + **computation** is possible if your computation resource is enough.

Check example, `sumMatrixOnGPUSStream.cu` and `sumMatrixOnGPUNoStream.cu`

Goal: Compute two matrix addition (C1 = A1 + B1 and C2 = A2 + B2)
  The latency in data transfer could be hidden during computing
  or concurrent computation.
Concurrent Processing

Sequential processing

Concurrent processing (data transfer + computation)

Concurrent processing (computation + computation)
In neural network, the most important operation is **inner-product**

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

- **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 intensively regular operations.

Example, \( x^T w + b \)

**cuBLAS (GPU) vs. OpenBLAS (CPU)**
- GPU computation includes data transfer between host and device.

**GPU compute a **(4096,4096)** matrix, spent 0.819480 secs**

**CPU compute a **(4096,4096)** matrix, spent 1.527501 secs**
Reference

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

   http://www.wrox.com/WileyCDA/WroxTitle/Professional-CUDA-C-
   Programming.productCd-1118739329,descCd-DOWNLOAD.html

   source code are available on the above website

GTC On-Demand:


Developer Zone:

   http://www.gputechconf.com/resource/developer-zone

NVIDIA Parallel Programming Blog:

   http://devblogs.nvidia.com/parallelforall

NVIDIA Developer Zone Forums:

   http://devtalk.nvidia.com
Characteristics of different GPUs on iDevice (iPhone 6S, iPad Air 2, iPad Pro)

<table>
<thead>
<tr>
<th></th>
<th>iPhone 6S</th>
<th>iPhone Air 2</th>
<th>iPad Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x Twister @ 1.85 GHz</td>
<td>3x Typhone @ 1.5 GHz</td>
<td>2x Twister @ 2.26 GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>PVR GT7600</td>
<td>PVR GXA6850</td>
<td>PVR 12 Cluster Series 7</td>
</tr>
<tr>
<td>RAM</td>
<td>2GB LDDR4</td>
<td>2GB LDDR3</td>
<td>4GB LDDR4</td>
</tr>
<tr>
<td>Memory bus width</td>
<td>64-bit</td>
<td>128-bit</td>
<td>128-bit</td>
</tr>
<tr>
<td>Max # of threads per</td>
<td>512</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>group</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

iDevice’s GPU is embedded in SoC → CPU and GPU share the same physical memory.
No memory transfers between host and device memory.
Note: one memory buffer can not exceed **256 MB**.
GPU Execution 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
Metal Programming, Kernel Function

Compute command

- Two parameters, `threadsPerGroup` and `numThreadgroups`, determines number of threads. \( \rightarrow \) equivalent to grid and thread block in CUDA. They are all 3-D variable.
- The total of all threadgroup memory allocations must not exceed 16 KB. a.k.a. shared memory in NVIDIA GPU.

- Example: sigmoid function, an important activation function in neural network.

```
kern 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]));
}
```
func loadDevice() -> (MTLDevice, MTLCustomCommandQueue, MTLLibrary,
  MTLCustomCommandBuffer, MTLCustomComputeCommandEncoder) {
  // Get access to iPhone or iPad GPU
  let device: MTLDevice! = MTLCustomCreateSystemDefaultDevice()
  // Queue to handle an ordered list of command buffers
  let commandQueue: MTLCustomCommandQueue! = device.newCommandQueue()
  // Access to Metal functions that are stored in Shaders.metal file, e.g. sigmoid()
  let gpuLibrary: MTLLibrary! = device.newDefaultLibrary()
  // Buffer for storing encoded commands that are sent to GPU
  let commandBuffer = commandQueue.commandBuffer()
  // Encoder for GPU commands
  let computeCommandEncoder = commandBuffer.computeCommandEncoder()

  return (device!, commandQueue, gpuLibrary, commandBuffer, computeCommandEncoder)
}

// register functions, sigmoid and normal distribution
let sigmoidFunction = gpuLibrary.newFunctionWithSignature("sigmoid")
var sigmoidComputePipelineFilter: MTLCustomComputePipelineState!
do {
  sigmoidComputePipelineFilter = try device.newComputePipelineStateWithFunction(sigmoidFunction!);
}
catch let error as NSError {
  fatalError("Unexpected error occurred: \\(error.localizedDescription).")
}
Metal Programming, Example in Sigmoid function

```swift
// pin CPU memory location to GPU; note that the memory should be aligned with 16KB
let inVectorBufferNoCopy = device.newBufferWithBytesNoCopy(inMemory, length: Int(size), options: [], deallocator: nil)
computeCommandEncoder.setBuffer(inVectorBufferNoCopy, offset: 0, atIndex: 0) // index is very important, you will access the data in kernel with that index.

let outVectorBufferNoCopy = device.newBufferWithBytesNoCopy(outMemory, length: Int(size), options: [], deallocator: nil)
computeCommandEncoder.setBuffer(outVectorBufferNoCopy, offset: 0, atIndex: 1)

let threadsPerGroup = MTLSIZE(width:maxThreadExeWidth!, height:1, depth:1)
let numThreadgroups = MTLSIZE(width:(Int(maxCount)+maxThreadExeWidth!-1)/maxThreadExeWidth!, height:1, depth:1)

computeCommandEncoder.setComputePipelineState(computePipelineFilter!)
computeCommandEncoder.dispatchThreadgroups(numThreadgroups, threadsPerThreadgroup: threadsPerGroup)
computeCommandEncoder.endEncoding()

commandBuffer.commit()
commandBuffer.waitUntilCompleted()
```

Perform sigmoid function on $2^{24}$ data points on iPad Pro:

**SpeedGPU:** runtime : 0.01544429166666667 seconds  
**CPU:** runtime : 39.5845222083333 seconds  
**Speedup Ratio:** 2563.05197173713
Sobel operator for detecting edges in Images

\[
\begin{pmatrix}
-1 & 0 & +1 \\
-2 & 0 & +2 \\
-1 & 0 & +1 \\
\end{pmatrix}
\quad +
\quad \begin{pmatrix}
+1 & +2 & +1 \\
0 & 0 & 0 \\
-1 & -2 & -1 \\
\end{pmatrix}
\]
Metal Programming, Example in Sobel operator

Metal codes:

```c
kernel void sobelOperator(const device float *inVector [[ buffer(0) ]],
                         const device int *inParams [[buffer(2)]],
                         device float *outVector [[ buffer(1) ]],
                         uint2 id [[ thread_position_in_grid ]]) {

    int row = inParams[0];
    int col = inParams[1];

    int row_index = id[0];
    int col_index = id[1];

    float3x3 horMask;
    horMask[0][0] = -1.0; horMask[0][1] = -2.0; horMask[0][2] = -1.0;
    horMask[1][0] =  0.0; horMask[1][1] =  0.0; horMask[1][2] =  0.0;
    horMask[2][0] =  1.0; horMask[2][1] =  2.0; horMask[2][2] =  1.0;

    float3x3 verMask;
    verMask[0][0] = -1.0; verMask[0][1] =  0.0; verMask[0][2] =  1.0;
    verMask[1][0] = -2.0; verMask[1][1] =  0.0; verMask[1][2] =  2.0;
    verMask[2][0] = -1.0; verMask[2][1] =  0.0; verMask[2][2] =  1.0;

    float horTemp = 0.0;
    float verTemp = 0.0;

    if (row_index >= 1 && row_index < row - 1 && col_index >= 1 && col_index < col - 1) {
        for (int j = -1; j <= 1; ++j) {
            for (int i = -1; i <= 1; ++i) {
                float pixelTemp = inVector[(col_index + j) * row + (row_index + i)];
                horTemp += (pixelTemp * horMask[j + 1][i + 1]);
                verTemp += (pixelTemp * verMask[j + 1][i + 1]);
            }
        }
        float grad = sqrt(horTemp * horTemp + verTemp * verTemp);
        outVector[col_index * row + row_index] = grad;
    } else {
        outVector[col_index * row + row_index] = 0.0;
    }
}
```
CPU side: Swift language

```swift
var horMask: [[Float]] = [[-1.0,-2.0,-1.0],
                         [0.0,0.0,0.0],
                         [1.0,2.0,1.0]]
var verMask: [[Float]] = [[-1.0,0.0,1.0],
                         [-2.0,0.0,2.0],
                         [-1.0,0.0,1.0]]

start = mach_absolute_time()
for j in 0..<imageHeight {
    for i in 0..<imageWidth {
        if (j >= 1 && j < imageHeight - 1 && i >= 1 && i < imageWidth - 1) {
            var tempGradX: Float = 0.0
            var tempGradY: Float = 0.0
            for m in -1..<2 {
                for n in -1..<2 {
                    let pixelTemp: Float = inMemBuf[(j + m) * imageWidth + (i + n)]
                    tempGradX += (pixelTemp * horMask[m + 1][n + 1])
                    tempGradY += (pixelTemp * verMask[m + 1][n + 1])
                }
            }
            let grad: Double = sqrt(Double(tempGradX * tempGradX + tempGradY * tempGradY))
            h_outMemBuf[j*imageWidth + i] = Float(grad);
        } else {
            h_outMemBuf[j*imageWidth + i] = 0.0;
        }
    }
}
CPU/GPU Performance on iPhone 6 and iPad Pro:
Image size: 4096x3072 (image size captured by 12 MegaPixel Camera)

**iPhone 6**

GPU: runtime : 0.384229375 seconds  
CPU: runtime : 75.0961254166667 seconds  
Speedup Ratio: 195.446080656032

**iPad Pro**

GPU: runtime : 0.02371125 seconds  
CPU: runtime : 39.2896004166667 seconds  
Speedup Ratio: 1657.00249529935

Thanks for colleague, Dr. Larry Lai in helping the app UI.