Real-Time GPU Management

Heechul Yun
This Week

• Topic: General Purpose Graphic Processing Unit (GPGPU) management

• Today
  – GPU architecture
  – GPU programming model
  – Challenges
  – Real-Time GPU management
History

• GPU
  – Graphic is **embarrassingly parallel** by nature
  – GeForce 6800 (2003): 53GFLOPs (MUL)
  – Some **PhDs** tried to use GPU to do some general purpose computing, but difficult to program

• GPGPU
  – **Ian Buck** (Stanford PhD, 2004) joined Nvidia and created CUDA language and runtime.
  – General purpose: (relatively) easy to program, many scientific applications
Discrete GPU

- **Add-on PCIe cards on PC**
  - GPU and CPU memories are separate
  - GPU memory (GDDR) is much faster than CPU one (DDR)
Integrated CPU-GPU SoC

- Tighter integration of CPU and GPU
  - Memory is shared by both CPU and GPU
  - Good for embedded systems (e.g., smartphone)
NVIDIA Titan Xp

• 3840 CUDA cores, 12GB GDDR5X
• Peak performance: 12 TFLOPS
NVIDIA Jetson TX2

- 256 CUDA GPU cores + 4 CPU cores

Image credit: T. Amert et al., “GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed,” RTSS17
# NVIDIA Jetson Platforms

<table>
<thead>
<tr>
<th></th>
<th>Jetson TX2</th>
<th>Jetson TX1</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA Pascal™, 256 CUDA cores</td>
<td>NVIDIA Maxwell™, 256 CUDA cores</td>
</tr>
<tr>
<td>CPU</td>
<td>HMP Dual Denver 2/2 MB L2 + Quad ARM® A57/2 MB L2</td>
<td>Quad ARM® A57/2 MB L2</td>
</tr>
<tr>
<td>Video</td>
<td>4K x 2K 60 Hz Encode (HEVC) 4K x 2K 60 Hz Decode (12-Bit Support)</td>
<td>4K x 2K 30 Hz Encode (HEVC) 4K x 2K 60 Hz Decode (10-Bit Support)</td>
</tr>
<tr>
<td>Memory</td>
<td>8 GB 128 bit LPDDR4 58.3 GB/s</td>
<td>4 GB 64 bit LPDDR4 25.6 GB/s</td>
</tr>
<tr>
<td>Display</td>
<td>2x DSI, 2x DP 1.2 / HDMI 2.0 / eDP 1.4</td>
<td>2x DSI, 1x eDP 1.4 / DP 1.2 / HDMI</td>
</tr>
<tr>
<td>CSI</td>
<td>Up to 6 Cameras (2 Lane) CSI2 D-PHY 1.2 (2.5 Gbps/Lane)</td>
<td>Up to 6 Cameras (2 Lane) CSI2 D-PHY 1.1 (1.5 Gbps/Lane)</td>
</tr>
<tr>
<td>PCIe</td>
<td>Gen 2</td>
<td>1x4 + 1x1 OR 2x1 + 1x2</td>
</tr>
<tr>
<td>Data Storage</td>
<td>32 GB eMMC, SDIO, SATA</td>
<td>16 GB eMMC, SDIO, SATA</td>
</tr>
<tr>
<td>Other</td>
<td>CAN, UART, SPI, I2C, I2S, GPIOs</td>
<td>UART, SPI, I2C, I2S, GPIOs</td>
</tr>
<tr>
<td>USB</td>
<td>USB 3.0 + USB 2.0</td>
<td></td>
</tr>
<tr>
<td>Connectivity</td>
<td>1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth</td>
<td></td>
</tr>
<tr>
<td>Mechanical</td>
<td>50 mm x 87 mm (400-Pin Compatible Board-to-Board Connector)</td>
<td></td>
</tr>
</tbody>
</table>
CPU vs. GPGPU

• CPU
  – Designed to run sequential programs faster
  – High ILP: pipeline, superscalar, out-of-order, multi-level cache hierarchy
  – Powerful, but complex and **big**

• GPGPU
  – Designed to compute math faster for embarrassingly **parallel data** (e.g., pixels)
  – No need for complex logics (no superscalar, out-of-order, cache)
  – Simple, less powerful, but **small**---can put **many** of them
CPU-“style” cores

- Fetch/Decode
- Out-of-order control logic
- ALU (Execute)
- Fancy branch predictor
- Execution Context
- Memory pre-fetcher
- Data Cache (A big one)
Slimming down

Idea #1:
Remove components that help a single instruction stream run fast
Add ALUs

Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

“From Shader Code to a Teraflop: How GPU Shader Cores Work”, Kayvon Fatahalian, Stanford University
Modifying the shader

New compiled shader:

Processes 8 fragments using vector ops on vector registers
128 fragments in parallel

16 cores = 128 ALUs
= 16 simultaneous instruction streams
But what about branches?

Not all ALUs do useful work! Worst case: 1/8 performance

<unconditional shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional shader code>
GPU Programming Model

• Host = CPU
• Device = GPU
• Kernel
  – Function that executes on the device
  – Multiple threads execute each kernel
Example: Increment Array Elements

CPU program

```c
void increment_cpu(float *a, float b, int N) {
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}
```

CUDA program

```c
__global__ void increment_gpu(float *a, float b, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}
```

```c
void main() {
    ..... 
    increment_cpu(a, b, N);
}
```

```c
void main() {
    ..... 
    dim3 dimBlock(blocksize);
    dim3 dimGrid(ceil(N / (float)blocksize));
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Source: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

```
threadID  0  1  2  3  4  5  6  7

... float x = input[threadID];
    float y = func(x);
    output[threadID] = y;
    ...
```
A kernel is executed by a **grid of thread blocks**

- A **thread block** is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution

- Threads from different blocks cannot cooperate

Source: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf
Memory Model

- **Registers**
  - Per thread
  - Data lifetime = thread lifetime

- **Local memory**
  - Per thread off-chip memory (physically in device DRAM)
  - Data lifetime = thread lifetime

- **Shared memory**
  - Per thread block on-chip memory
  - Data lifetime = block lifetime

- **Global (device) memory**
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation

- **Host (CPU) memory**
  - Not directly accessible by CUDA threads

Source: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf
Challenges for Discrete GPU

• Data movement problem
  – Host mem <-> gpu mem
  – Copy overhead can be high

• Scheduling problem
  – Limited ability to prioritize important GPU kernels
  – Most (old) GPUs don’t support preemption
  – New GPUs support preemption within a process
Data Movement Challenge

Nvidia Tesla K80

- 4992 GPU cores
- 480 GB/s
- Graphic DRAM

Intel Core i7

- 4 CPU cores
- 25 GB/s
- Host DRAM

PCIE 3.0

16 GB/s

Data transfer is the bottleneck
An Example

PTask: Operating System Abstractions To Manage GPUs as Compute Devices, SOSP'11
Inefficient Data migration

#> capture | xform | filter | detect &

Acknowledgement: This slide is from the paper's author's slide
Scheduling Challenge

CPU priorities do not apply to GPU

Long running GPU task (xform) is not preemptible delaying short GPU task (mouse update)
Challenges for Integrated CPU-GPU

- Memory is shared by both CPU and GPU
- Data movement may be easier but...

![Diagram of Nvidia Tegra X2 architecture](diagram.png)
Memory Bandwidth Contention

Co-scheduling memory intensive CPU task affects GPU performance on Integrated CPU-GPU SoC

Summary

• GPU Architecture
  – Many simple in-order cores

• GPU Programming Model
  – SIMD

• Challenges
  – Data movement cost
  – Scheduling
  – Bandwidth bottleneck
  – NOT time predictable!
Real-Time GPU Management

• Goal
  – Time predictable and efficient GPU sharing in multi-tasking environment

• Challenges
  – High data copy overhead
  – Real-time scheduling support -- preemption
  – Shared resource (bandwidth) contention
References

• Timegraph: Gpu scheduling for real-time multi-tasking environments. In ATC, 2011.
• Gdev: First-class gpu resource management in the operating system. In ATC, 2012.
• GPES: a preemptive execution system for gpgpu computing. In RTAS, 2015
• A server based approach for predictable gpu access control. In RTCSA, 2017.
Real-Time GPU Scheduling

• Early real-time GPU schedulers
  – Timegraph
  – Gdev
• GPU kernel slicing
  – GPES
• Synchronization (Lock) based approach
  – GPUSync
• Server based approach
  – GPU Server
Acknowledgement: This slide is from the paper author’s slide
• First work to support “soft” real-time GPU scheduling.
• Implemented at the device driver level
TimeGraph Scheduling

- GPU commands are not immediately sent to the GPU, if it is busy
- Schedule high-priority GPU commands when GPU becomes idle
• Implemented at the kernel level on top of the stock GPU driver.

Acknowledgement: This slide is from the paper author’s slide.
TimeGraph Scheduling

- high priority tasks can still suffer long delays
- Due to lack of hardware preemption

Acknowledgement: This slide is from the paper author’s slide
GDev’s BAND Scheduler

- Monitor consumed b/w, add some delay to wait high-priority requests
- Non-work conserving, no real-time guarantee

Load balanced

Acknowledgement: This slide is from the paper author’s slide
GPES

- Based on Gdev
- Implement kernel slicing to reduce latency
GPUSync Approach

- Synchronization-based approach
  - Models each GPU access segment as a critical section
  - Uses a real-time synchronization protocol to handle GPU requests

Benefits
- Does not require any change in GPU device drivers
- Existing schedulability analyses can be directly reused

---


Limitations

1. Busy waiting
   - Critical sections are executed entirely on the CPU
   - No suspension during the execution of a critical section

2. Long priority inversion
   - High priority tasks may suffer from unnecessarily long priority inversion
   - Due to priority boosting used by some protocols, e.g., MPCP and FMLP

---

Common assumptions of most RT synch. protocols, e.g., MPCP*, FMLP†, OMLP‡

---


Server-based Approach

- **GPU server task**
  - Handles GPU access requests from other tasks on their behalf
  - Allows tasks to suspend whenever no CPU intervention is needed

Diagram:
- Task $\tau_i$ with shared memory region
- GPU access segment: Data, Command, Code
- GPU request queue
  - Order requests in task priority
  - Execute the highest priority GPU request
- GPU server suspends during CPU-inactive time

Example under Synch-based Approach

* MPCP is used

Response time of task $\tau_h$: 9

CPU Core 1

$\tau_h$

$\tau_m$

$\tau_l$

GPU

$G_l$

$G_h$

$G_m$

Normal exec. segment

Misc. GPU operation

GPU computation

Busy waiting

Example under Server-based Approach

Response time of task $\tau_h: 6+4\varepsilon$

- **CPU Core 1**
  - $\tau_h$ (normal execution segment)
  - $\tau_m$ (miscellaneous GPU operation)
- **CPU Core 2**
  - $\tau_l$ (normal execution segment)
- **GPU**
  - $\tau_l$ (GPU computation)
  - $\tau_h$ (GPU computation)
  - $\tau_m$ (GPU computation)

Legend:
- Normal exec. segment
- Misc. GPU operation
- GPU computation
- Svr. overhead

Hardware Preemption

• Recent GPUs (NVIDIA Pascal) support hardware preemption capability
  – Problem solved?

• Issues
  – Works only between GPU streams within a single address space (process)
  – High context switching overhead
    • ~100us per context switch (*)

Hardware Preemption

PASCAL PREEMPTION
ENABLES REAL-TIME WORKLOADS

Graphics Preemption – Pixel Level
- First ever pixel-level graphics preemption
- Pixel level graphics + thread-level compute preemption → sub-100us preemption for gaming

Compute Preemption – Thread Level
Discussion

• Long running low priority GPU kernel?

• Memory interference from the CPU?
Challenges for Integrated CPU-GPU

• Memory is shared by both CPU and GPU
• Data movement may be easier but...

Nvidia Tegra X2

PMC
Core1

PMC
Core2

PMC
Core3

PMC
Core4

GPU cores

Shared Memory Controller

16 GB/s

Shared DRAM
References

• SiGAMMA: server based integrated GPU arbitration mechanism for memory accesses, RTNS, 2017
• GPUguard: towards supporting a predictable execution model for heterogeneous SoC, DATE, 2017
• Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms, ECRTS, 2018
SiGAMMA

- Protect PREM compliant real-time CPU tasks
- Throttle GPU when CPU is in a mem-phase
SiGAMMA

- GPU is throttled by launching a high-priority spinning kernel
GPUGuard

- PREM compliant GPU and CPU tasks.
- CPU is throttled using MemGuard
- Need GPU source code modification
• Protect real-time GPU tasks by throttling CPU
• Runtime binary instrumentation overriding CUDA API
  – No code modification is needed
Dynamic Instrumentation

- Begin/stop throttling by instrumenting CUDA

```
cudaMalloc(...)  
cudaMemcpy(...)  
cudaMemcpy(...)  
kernel<<<...>>>(...)  
cudaFree(...)  
cudaLaunch()  
cudaSynchronize()
```

- Acquire memory bandwidth lock
- Release memory bandwidth lock

Instrument binary using `LD_PRELOAD`

No source code modification is needed
BWLOCK++

- Real-time GPU kernels are protected.
Summary

• Integrated CPU-GPU SoC
  – Shared main memory is a source of interference
• SiGAMMA
  – PREM compliant CPU tasks
  – Throttle GPU to protect real-time CPU tasks
• GPUGuard
  – PREM compliant GPU (and CPU) tasks.
  – Need GPU source code modification
• BWLOCK++
  – Throttle CPU (memguard) to protect real-time GPU tasks
  – Runtime instrumentation (no code modification)
Discussion Papers

• Deadline-based Scheduling for GPU with Preemption Support, RTSS, 2018
• Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference, RTSS, 2019
Deadline-based Scheduling for GPU with Preemption Support

N. Capodieci, R. Cavicchioli, M. Bertogna, A. Paramakuru.

RTSS, 2018
Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference

Yecheng Xiang and Hyoseung Kim

RTSS’19
Cameras in Self-Driving Car
NVIDIA Jetson TX2

- Denver
  - Core
  - Core

- Cortex-A57
  - Core
  - Core
  - Core
  - Core

- Pascal GPU

Shared DRAM
DNN Layer Processing Time

Fig. 10: LeNet layer execution time profile on TX2
Problem

• Processing DNNs on a heterogeneous platform (e.g., TX2) is not efficient
  – Using one type of resource (e.g., GPU) may not be always the best
  – Because CPUs don’t do much (if any) and GPU utilization is low for many layers
  – Also, sometimes CPUs are faster than GPU

• How to process multiple DNNs efficiently on a heterogeneous platform?
Key Ideas

• Group layers into multiple stages
• A stage can be processed in different computing nodes
• Dynamically match make stages and nodes to maximize performance while respecting real-time requirements
Fig. 6: CPU scheduling in the status quo and DART
Fig. 7: GPU scheduling in the status quo and DART
Summary & Discussion

• Efficiently schedule multiple DNN models on a heterogeneous computing platform
• Exploit DNN layer-level differences on best computing resource configurations
• Improve throughput and support real-time scheduling

• Downsides?