General purpose processing using embedded GPUs: A study of latency and its variation

Matthias Rosenthal and Amin Mazloumian
May, 2016
Agenda

- General Purpose GPU Computing
- Embedded CPU/GPU versus CPU/FPGA
- CPU – GPU Data Transfer
  - Unified Virtual Addressing (DMA)
  - Memory mapped (Zero Copy)
- Latency Results
- Kernel-Loop Solution avoiding GPU Kernel launch
GPU Computing

Originally used 3D game rendering

GPUs are heavily used in
High Performance Computing
Financial modeling
Robotics
Gas and oil exploration
Cutting-edge scientific research

→ What about embedded systems??
CPU vs. GPU

CPU vs. GPU

- CPUs: Huge cache, optimized for several threads: *Sequential instructions*
- GPUs: 100+ simple cores for huge parallelization: *Intensive parallelization*
Discrete vs Integrated GPU

**Discrete GPU**
- CPU
- GPU
- Cache
- System Memory
- GPU Memory

**Integrated GPU**
- CPU
- GPU
- Cache
- Shared Physical Memory
# CPU/GPU Computing vs. CPU/FPGA

<table>
<thead>
<tr>
<th></th>
<th>CPU/GPU</th>
<th>CPU/FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexibility &amp; Maintenance</td>
<td>High</td>
<td>Mid</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Development Cost</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Latency</td>
<td>Micro seconds</td>
<td>Nano seconds</td>
</tr>
<tr>
<td>Latency variation</td>
<td>?</td>
<td>No variation</td>
</tr>
</tbody>
</table>

(CPU/GPU/DSP/FPGA)
Example: Nvidia TK1

- GPU: 192 Cuda core
- CPU: ARM A-15 Quad-core
- Video decode: Full-HD 60 Hz
- Video encode: Full-HD 30 Hz
- Networking: 1 GB Ethernet
GPU Programming: CUDA

Additional Libraries

Standard Cuda Programm

Linux compilation model

[https://code.msdn.microsoft.com/vstudio/NVIDIA-GPU-Architecture-45c1e6d]
Nvidia TK1

Kepler Memory Hierarchy

TK1

192 Cores

SM-0

Registers

L1

SMEM

Read only

SM-1

Registers

L1

SMEM

Read only

SM-N

Registers

L1

SMEM

Read only

64KByte Configurable
L1 / SMEM / RO

128KByte L2

L2

Global Memory (DRAM)

[GPU performance Analysis, Nvidia (2012)]
Data Transfer on TK1

2 Options for Data Transfer to GPU in Cuda:

- Unified Virtual Addressing (GPU DMA Transfer)
- Memory mapped (Zero Copy)
Cuda Data Transfer

Method 1: Unified Virtual Addressing (with CPU-GPU DMA)

- Allocation in GPU memory
- Local access for first GPU
- No direct CPU access
- DMA Transfer CPU <-> GPU
  cudaMemcpy
Cuda Data Transfer

GPU processing Unified Virtual Addressing (DMA):

Step 1: Copy data to GPU memory

Step 2: Process data in GPU using 1000s of threads

Step 3: Copy results back to host memory
Cuda Data Transfer

// Step 0: allocate memory
cudaMalloc( &dev_a, size );
cudaMalloc( &dev_b, size );
cudaMalloc( &dev_c, size );

// Step 1: copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice ); // GPU-DMA
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice ); // GPU-DMA

// Step 2: launch add() kernel on GPU
add <<< N, M >>>( dev_a, dev_b, dev_c );

// Step 3: copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost )
Method 2: Memory mapped (Zero Copy)

- Allocation in CPU memory
- Local access for CPU
- Memory mapped for GPUs
Cuda Data Transfer

GPU processing Memory Mapped (Zero Copy):

Step 1: Copy data to GPU memory

Step 2: Process data in GPU using 1000s of threads

Step 3: Copy results back to host memory
Typical GPU workflow: Memory-mapped

// Step 0: allocate memory
cudaMalloc(&dev_a, size); cudaMallocHost(&dev_a, size);
cudaMalloc(&dev_b, size); cudaMallocHost(&dev_b, size);
cudaMalloc(&dev_c, size); cudaMallocHost(&dev_c, size);

// Step 1: copy inputs to device
cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

// Step 2: launch add() kernel on GPU
add <<<< N, M >>>( dev_a, dev_b, dev_c );

// Step 3: copy device result back to host copy of c
cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost)
DMA vs. Memory-mapped

Factor 2
__device__ void identity( float *input, float *output, int numElem):
    for (int index = 0; index < numElem; index++) {
        output[index] = input[index]
    }

Tested on Linux-Kernel with PREEMPT_RT / Full Preempt
-There is a huge variation in processing time.

-For 100 bytes data (25 float values) per thread:
  
  -90% of the launches take less than 40 micro sec.
  
  -0.01% of the launches take around 500 micro sec.

-Slow launches drop update rate from 25 KHz to 2KHz.
GPU Latency Variation

Jetson TK1
RT Kernel
identity\texttt{<<<1,1>>>}

Pr\{T \geq t\}

GPU Kernel Time $t$ (\mu s)
Our Solution for Latency Variation

```
... 
    wait_for_input_in_DRAM();
    flag_to_GPU();
...

Kernel Loop: 

while (true) {
    poll_CPU_flag();
    output_data = fct(input_data);
}
```

- Implement kernel-loops in GPU cores
- Memory mapped (zero copy) data access
- Each GPU kernel-loop produces output from its input data (memory-mapped)

- The number of GPU cores limit the number of kernel loops
SoCs with GPU as Industrial Modules

Nvidia TK1 Module

Snapdragon 820 Module

Allwinner A80 Module

Nvidia TX1 Module

Sources: Nvidia, Avionic Design, Toradex, Intrinisic, Theobroma Systems
SoCs with GPU as Industrial Modules

- Video Conferencing
- Android TV
- Medical Imaging
- Driving Assistance
- Lecture recording streaming

Source: Google / PMK
Conclusion

- Our results confirm that for small data chunks memory mapped transfers is more efficient
- We observe a huge but rare variation in GPU processing time
- The variation dramatically reduces update rate by an order of magnitude
- Our solution is to implement GPU kernel-loops and memory-mapped transfer