OpenCL™ and RenderScript
OpenCL

- Khronos standard
- Heterogeneous Systems
- General and Low-level
  - Explicit device selection
  - Explicit memory management
  - Explicit kernel object management
The GPU-Compute API on Android™

Assumes CPU and GPU only

Transparent device selection
- The driver manages and selects devices

Transparent memory management
- Copying managed by the driver, based on allocation flags

Higher level than OpenCL
- Less explicit control over details
- Know your driver!
ARM® Mali™ GPUs

ARM Mali-T604 GPU and ARM Mali-T628 GPU
GPGPU on the Midgard Architecture

- **Work Item**: Registers, PC, SP, Private stack
- **Work Group**: Barriers, Local Atomics, Cached local memory
- **NDRange**: Global Atomics, Cached global memory
Inside a Core

$T = \max(A_0, A_1, LS, Tex)$
Latency Hiding by Parallelism

- Executing a program on an ARM Cortex®- A15 CPU
  - Execution of consecutive instructions overlap in time
  - Instruction latencies and branch predictions are important

- Executing a kernel on a ARM Mali-T6xx GPU
  - Execution of different threads overlap in time
  - Execution of different instructions of a single thread never overlap

- This leads to latency tolerance
  - No need for branch predictors
  - No need to worry about pipeline latencies
  - Memory latency can still be an issue
**Arithmetic and Load/Store pipes**

- **SIMD:** Several components per operation
  - 128-bit registers

- **VLIW:** Several operations per instruction word
  - Some operations are “free”

<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th>18</th>
<th>18</th>
<th>18</th>
<th>18</th>
<th>18</th>
<th>18</th>
<th>18</th>
<th>18</th>
<th>18</th>
<th>18</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

128 bit
Hardware Counters
Hardware Counters

- Counters per core
  - Active Cycles
  - Pipe activity
  - L1 cache

- Counters per Core Group
  - L2 caches

- Counters for the GPU
  - Active cycles

- Accessed through Streamline™
  - Timeline of all hardware counters, and more
  - Explore the execution of the full application
  - Zoom in on details
Streamline
Optimizing
with Streamline and Hardware Counters
Memories

- Only one programmer controlled memory
  - Many transparent caches

- Memory copying takes time
  - It can easily dominate over kernel execution time

- Use appropriate memory allocation schemes

- Avoid synchronization points
  - Cache maintenance has a cost as well

- Streamline to the rescue
  - Visualize when kernels are executed
  - Many features not covered here
Hiding Pipeline Latency

- Needs enough threads
  - Limited by register usage

- When there are issues
  - Few instructions issued per cycle
  - Spilling of values to memory

- Symptoms
  - Low Max Local Workgroup Size in OpenCL
  - Few instructions issued per cycle in limiting pipe

- Remedy
  - Smaller types → More values per register
  - Splitting kernels
Pipeline Utilization

- Prefer vector operations
  - More components per operation

- Prefer small types
  - More components in 128 bits

- Balance work between the pipes
  - Do less – with the pipe that limits performance

\[ T = \max( A_0, A_1, LS, Tex) \]
Finding the Bottlenecks

Host application or Kernel execution
- Avoid memory copying
- Avoid cache flushes

Which pipe is important?
- Operations in other pipes incur little or no runtime cost

Saving operations or saving registers
- How much register pressure can we handle, and still hide the latencies?

How well are we using the caches
- Are instructions spinning around the LS pipe waiting for data?
Optimizing
Two examples
The Limiting Pipe

- Three hardware counters
  - Cycles active (#C)
  - Number of A instructions (#A)
  - Number of LS instructions (#LS)

- The goal
  - Similar values for #A and #LS → Both pipes used
  - Max(#A, #LS) similar to #C → Limiting pipe used every cycle

- Example:
  - #LS / #A = 5
  - #LS / #A = 1, #C up by < 10%

\[ \bar{y} = a\bar{x} + \bar{y} \]

\[ \bar{y} = 0.05a\bar{x} + 0.05a\bar{x} + 0.05a\bar{x} + ... + 0.05a\bar{x} + \bar{y} \]
Cache Utilization

- The Load/Store pipe hides latency
  - Many threads active

Not always successful
- Insufficient parallelism
- Bad cache utilization
- Failing threads will be reissued

Reissue is a sign of cache-misses
- Instruction words issued
- Instruction words completed

Example
- Inter-thread stride for memory accesses
Kernel saxpy
- Load from x
- Load from y
- Compute
- Store to y

Execution order
- Threads 1 through N load from x
- Threads 1 through N load from y
- Threads 1 through N compute
- Threads 1 through N store to y

How many bytes should we load per thread?

\[ \bar{y} = a\bar{x} + \bar{y} \]
A Single Instruction Word

- We should have one load instruction word
  - The next bytes will be picked up by the next thread

- Loading less is bad
  - Does not utilize the SIMD operations

- Loading more is bad
  - The next bytes will be loaded after all other threads have loaded their first

- Saxpy with different strides
  - 128 bits: 4.5 issues per instruction
  - 256 bits: 5.5 issues per instruction
  - 64 bytes: 9.3 issues per instruction
Optimizing

General Advice
Know your bottleneck

- Use vector operations

- **If you are bandwidth-limited, merge kernels**
  - Avoid reloading data

- **If you are register-limited, split kernels**
  - Easier for the compiler to do a good job

- **If you are Load-Store-limited, do less load-store**
  - Compute complex expressions instead of using lookup-tables

- **If you are Arithmetic-limited, do less arithmetic**
  - Tabulate functions
  - Use polynomial approximations instead of special functions
Optimizing Compute Kernels for Mobile GPUs

Johan Gronqvist
Synchronization between threads

- Two options in OpenCL
  - Barriers inside a work-group
  - Atomics between work-groups

- We like atomics to ensure data consistency
  - But preferably on the same core

- Barriers can be useful to improve cache utilization
  - Limit divergence between threads
  - Keeping jobs small serves the same purpose

- We see examples of large jobs with many barriers
  - We often prefer small jobs with dependencies
Vectorize your operations

- More components per operation
  - For basic arithmetic and memory operations
  - Square roots, trigonometry and atomics are scalar

- Fewer registers used
  - The compiler will only do part of the job

<table>
<thead>
<tr>
<th>I8</th>
<th>I8</th>
<th>I8</th>
<th>I8</th>
<th>I8</th>
<th>I8</th>
<th>I8</th>
<th>I8</th>
<th>I8</th>
<th>I8</th>
<th>I8</th>
<th>I8</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
<td>INT16</td>
</tr>
<tr>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
<td>INT32</td>
</tr>
<tr>
<td>INT64</td>
<td>INT64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
<th>FP16</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
<td>FP32</td>
</tr>
<tr>
<td>FP64</td>
<td>FP64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

128 bit