Optimizing Compute Kernels for Mobile GPUs

Johan Gronqvist

ARM

ARM 2013 TechCon

Where Intelligence Connects

OpenCL™ and RenderScript

OpenCL

- Khronos standard
- Heterogeneous Systems
- General and Low-level
  - Explicit device selection
  - Explicit memory management
  - Explicit kernel object management

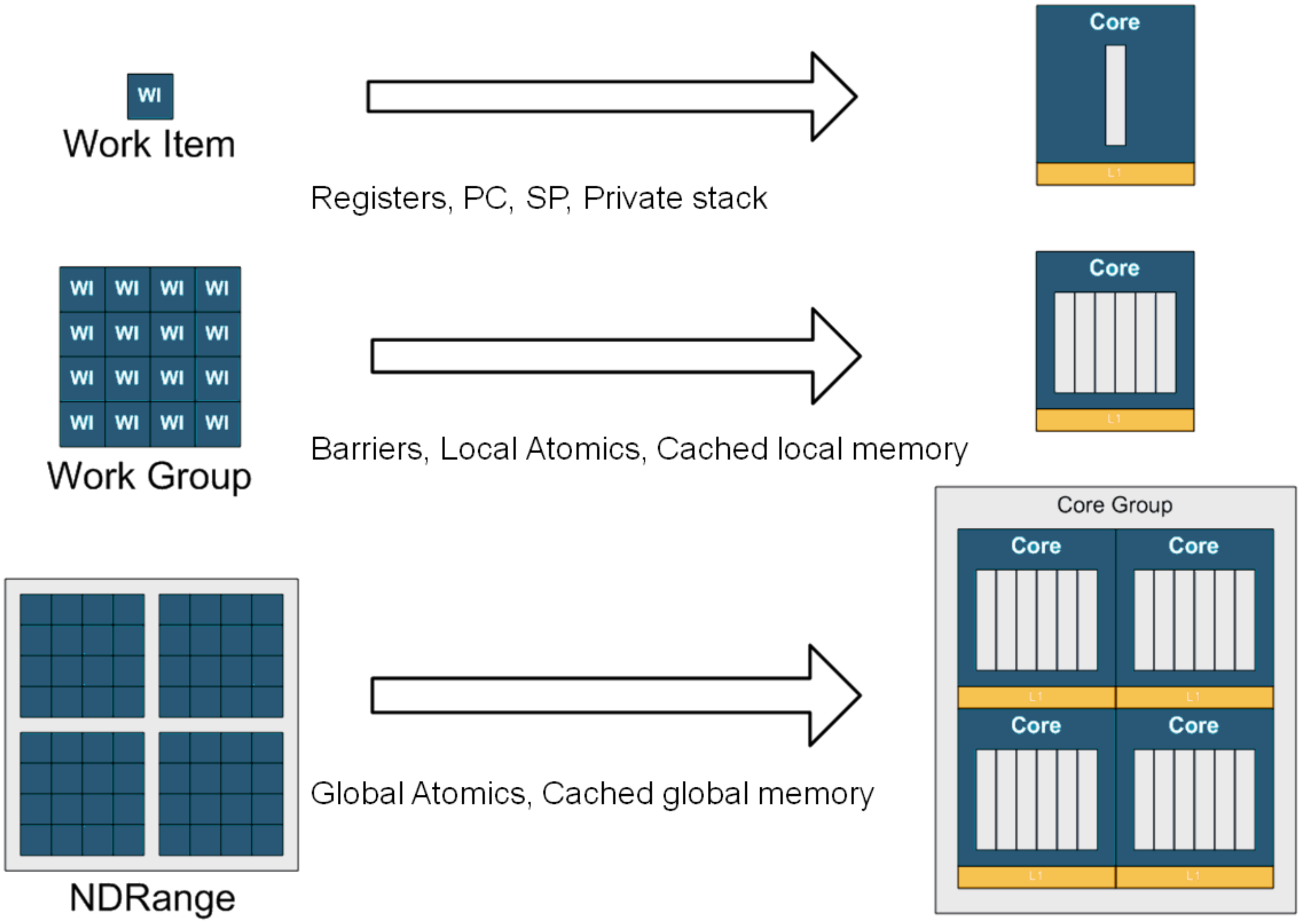
RenderScript

- 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

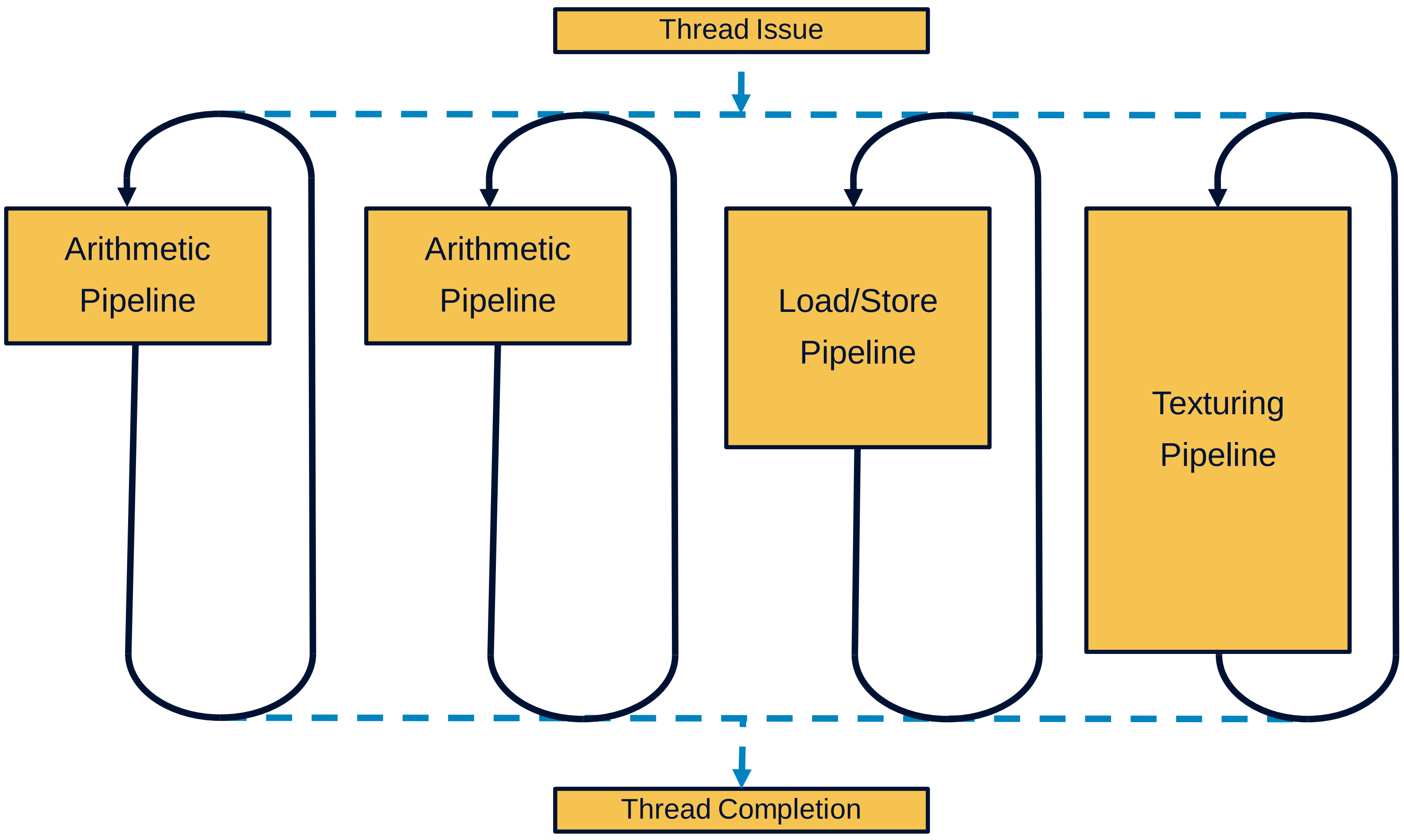


Registers, PC, SP, Private stack

Barriers, Local Atomics, Cached local memory

Global Atomics, Cached global memory

Inside a Core



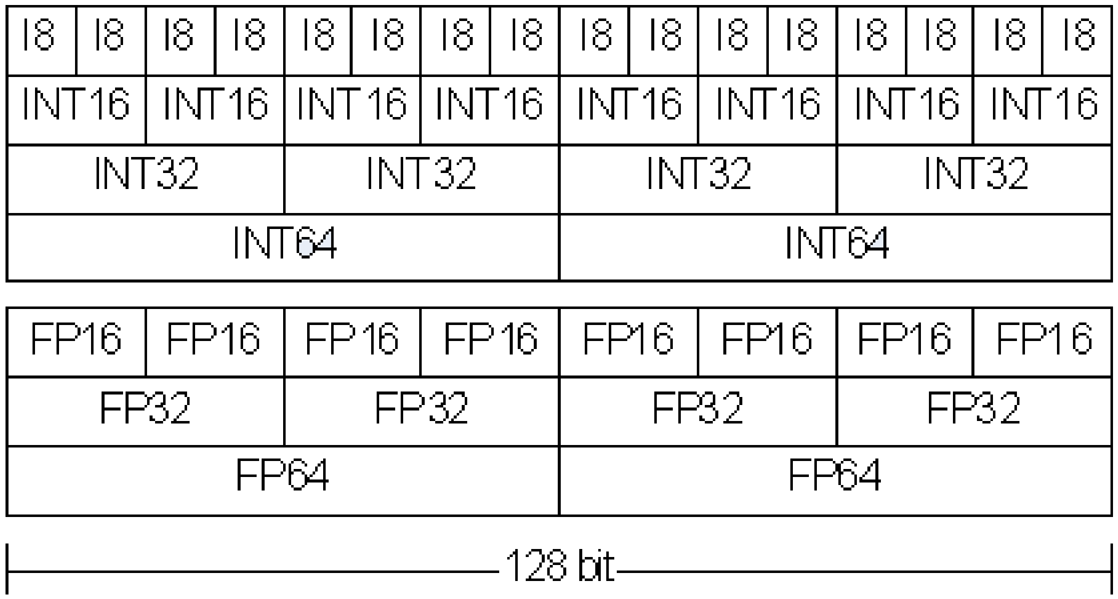
$$T = \max(A_0, A_1, LS, Tex)$$

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”

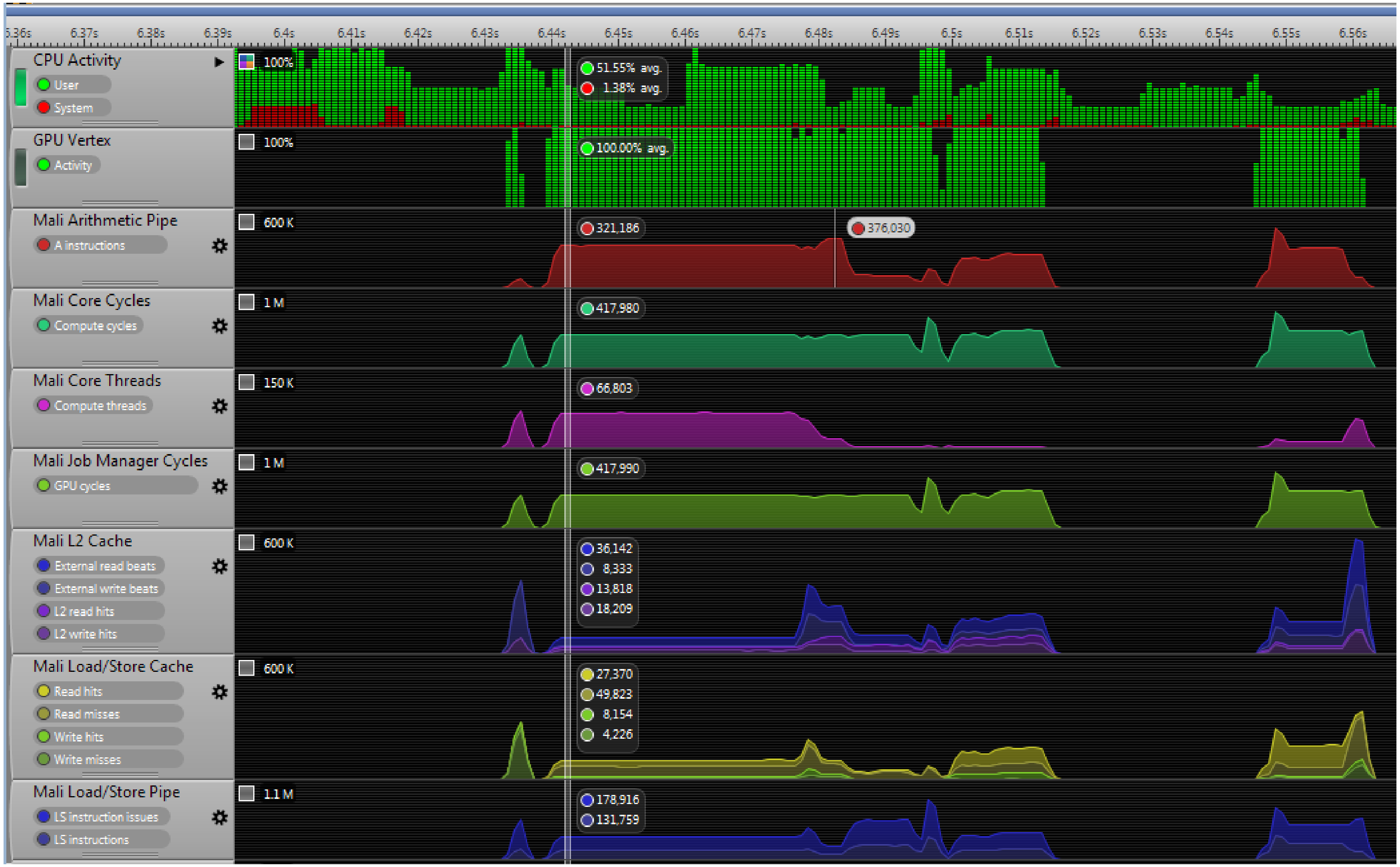

I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8
INT16 INT16 INT16 INT16 INT16 INT16 INT16 INT16
INT32 INT32 INT32 INT32
INT64 INT64
FP16 FP16 FP16 FP16 FP16 FP16 FP16 FP16
FP32 FP32 FP32 FP32
FP64 FP64
128 bit

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


6.36s
6.37s
6.38s
6.39s
6.4s
6.41s
6.42s
6.43s
6.44s
6.45s
6.46s
6.47s
6.48s
6.49s
6.5s
6.51s
6.52s
6.53s
6.54s
6.55s
6.56s
CPU Activity
User
System
100%
51.55% avg.
1.38% avg.
GPU Vertex
Activity
100%
100.00% avg.
Mali Arithmetic Pipe
A instructions
600 K
321,186
376,030
Mali Core Cycles
Compute cycles
1 M
417,980
Mali Core Threads
Compute threads
150 K
66,803
Mali Job Manager Cycles
GPU cycles
1 M
417,990
Mali L2 Cache
External read beats
External write beats
L2 read hits
L2 write hits
600 K
36,142
8,333
13,818
18,209
Mali Load/Store Cache
Read hits
Read misses
Write hits
Write misses
600 K
27,370
49,823
8,154
4,226
Mali Load/Store Pipe
LS instruction issues
LS instructions
1.1 M
178,916
131,759

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%

$$\bar{y} = a\bar{x} + \bar{y}$$

$$\bar{y} = 0.05a\bar{x} + 0.05a\bar{x} + 0.05a\bar{x} + \ldots + 0.05a\bar{x} + \bar{y}$$

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

Execution Order

- Kernel saxpy
  - Load from x
  - Load from y
  - Compute
  - Store to y

$$\bar{y} = a\bar{x} + \bar{y}$$

- 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?

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

ARM®

2013 TechCon™

Where Intelligence Connects

UBM Tech

Optimizing Compute Kernels for Mobile GPUs

Johan Gronqvist

ARM

ARM 2013 TechCon

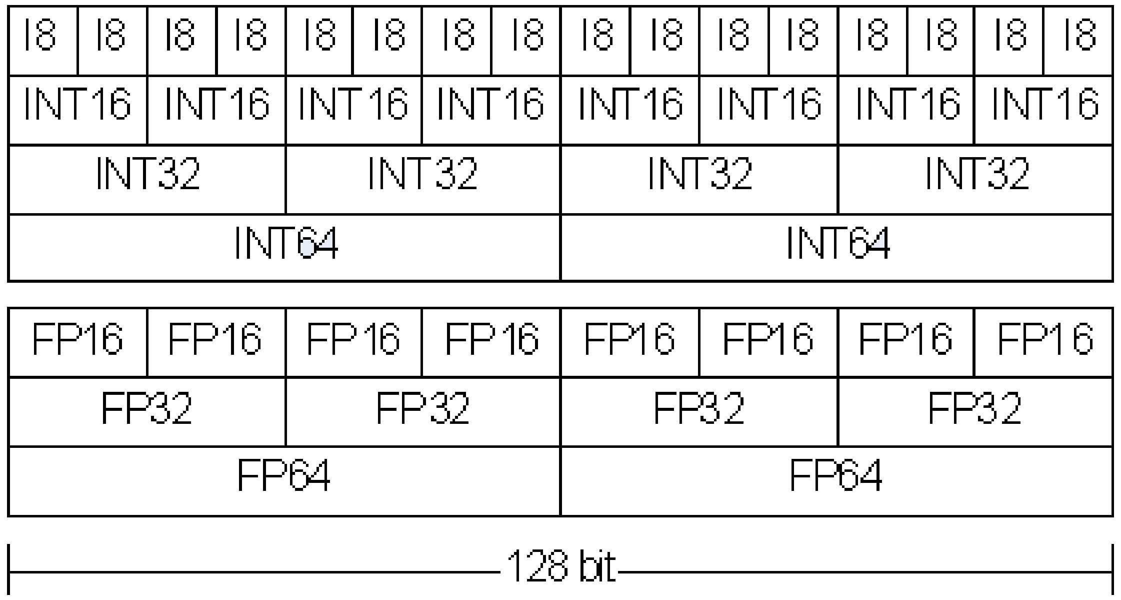
Where Intelligence Connects

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


I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8 I8
INT16 INT16 INT16 INT16 INT16 INT16 INT16 INT16
INT32 INT32 INT32 INT32
INT64 INT64
FP16 FP16 FP16 FP16 FP16 FP16 FP16 FP16
FP32 FP32 FP32 FP32
FP64 FP64
128 bit