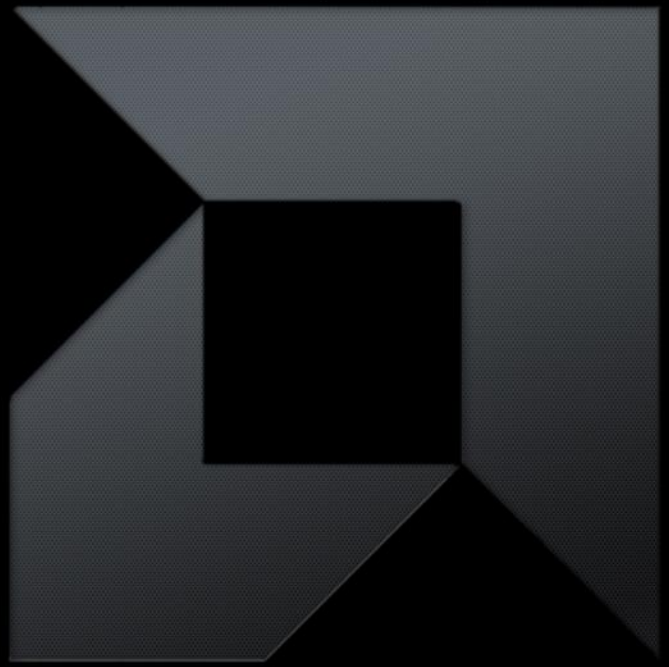


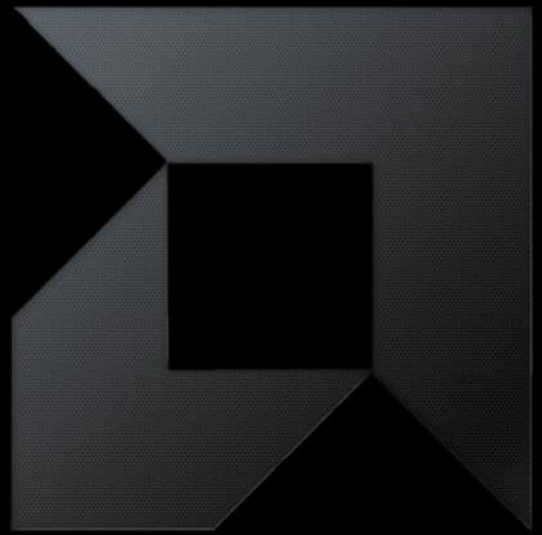
MAKING GPGPU EASIER

OpenCL C++ and a flexible architecture

Lee Howes and Benedict Gaster
AMD Fusion System Software

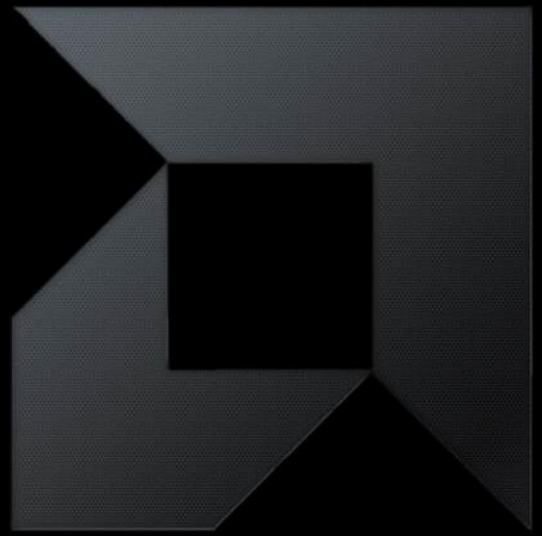
AMD

THE OPENCL C++ KERNEL LANGUAGE AND API



AMD

THE HD7970 AND GRAPHICS CORE NEXT



AMD

GPU EXECUTION AS WAS

- We often view GPU programming as a set of independent threads, more reasonably known as “work items” in OpenCL:

```
kernel void blah(global float *input, global float *output) {
  output[get_global_id(0)] = input[get_global_id(0)];
}
```

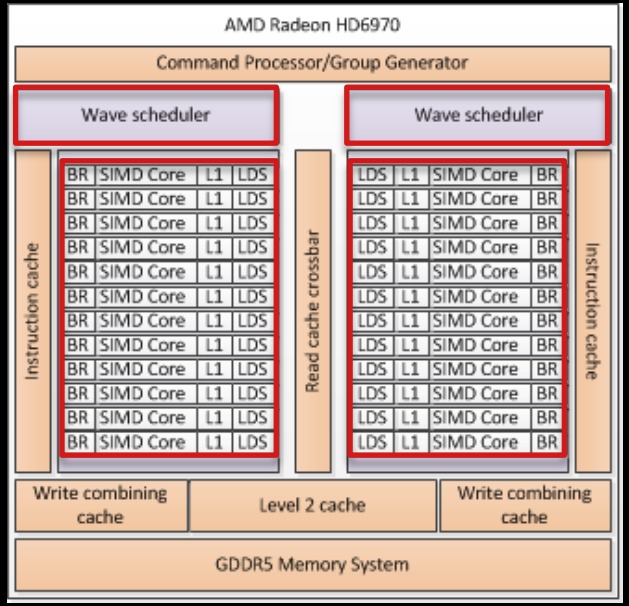
- Which we flatten to an intermediate language known as AMD IL:

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

- Note that AMD IL contains short vector instructions

MULTIPLE CORES

- We can run that IL across multiple cores in the GPU:
  - The HD6970 architecture has 24 vector cores
  - Each half of the device has a wave scheduler
    - This can be seen as a shared, massively threaded, scalar core
- The device as a whole can run up to about 500 threads
  - 500 program counters across the two schedulers
  - The device can execute around 32000 work items concurrently



MAPPING TO THE HARDWARE

- The GPU hardware of course does not execute those work items as threads

- The reality is that high-end GPUs follow a SIMD architecture
  - Each work item describes a lane of execution
  - Multiple work items execute together in SIMD fashion with a single program counter
  - Some clever automated stack management to handle divergent control flow across the vector

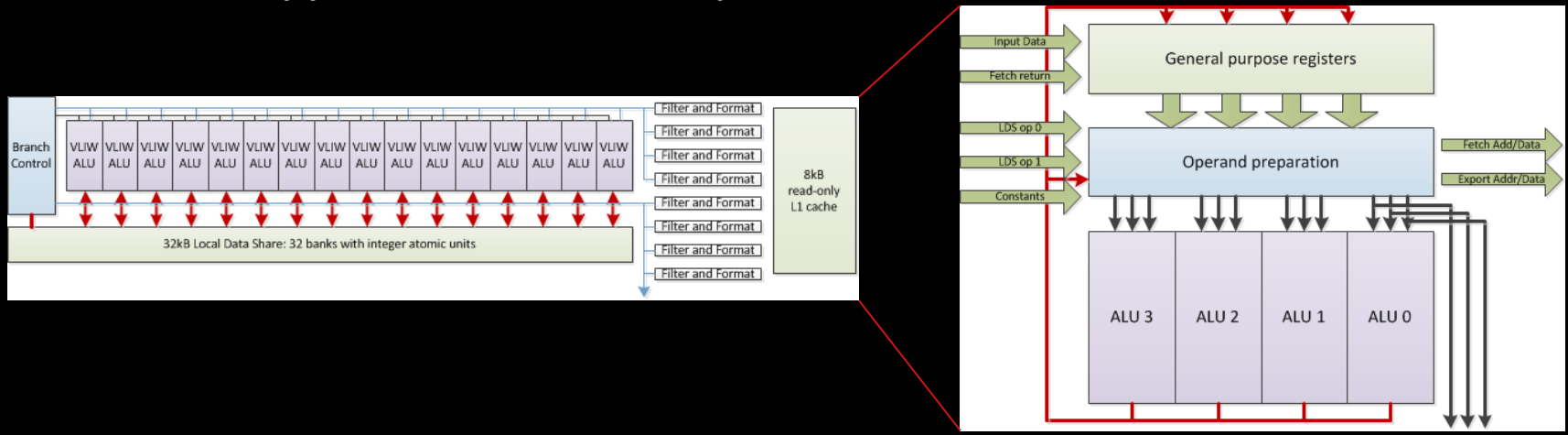
```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```


```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

```
mov r255, r1021.xyz0
mov r255, r255.x000
mov r256, l9.xxxx
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r253.x___, r2.xxxx, r255.xxxx
mov r255, r1022.xyz0
mov r255, r255.x000
ishl r255.x___, r255.xxxx, r256.xxxx
iadd r254.x___, r1.xxxx, r255.xxxx
mov r1010.x___, r254.xxxx
uav_raw_load_id(11)_cached r1011.x___, r1010.xxxx
mov r254.x___, r1011.xxxx
mov r1011.x___, r254.xxxx
mov r1010.x___, r253.xxxx
uav_raw_store_id(11) mem.x___, r1010.xxxx, r1011.xxxx
ret
```

IT WAS NEVER QUITE THAT SIMPLE

- The HD6970 architecture and its predecessors were combined multicore SIMD/VLIW machines
  - Data-parallel through hardware vectorization
  - Instruction parallel through both multiple cores and VLIW units
- The HD6970 issued a 4-way VLIW instruction per work item
  - Architecturally you could view that as a 4-way VLIW instruction issue per SIMD lane
  - Alternatively you could view it as a 4-way VLIW issue of SIMD instructions



WHAT DOES THAT MEAN TO THE PROGRAMMER?

- The IL we saw earlier ends up compiling to something like this:

```
; --------  Disassembly --------------------
00 ALU: ADDR(32) CNT(9) KCACHE0(CB1:0-15) KCACHE1(CB0:0-15)
     0  w: LSHL       ____,  R0.x,  2
     1  z: ADD_INT    ____,  KC0[0].x,  PV0.w
     2  y: LSHR       R0.y,  PV1.z,  2
     3  x: MULLO_INT  R1.x,  R1.x,  KC1[1].x
        y: MULLO_INT  ____,  R1.x,  KC1[1].x
        z: MULLO_INT  ____,  R1.x,  KC1[1].x
        w: MULLO_INT  ____,  R1.x,  KC1[1].x
01 TEX: ADDR(48) CNT(1)
     4  VFETCH R2.x___, R0.y, fc153
        FETCH_TYPE(NO_INDEX_OFFSET)
02 ALU: ADDR(41) CNT(7) KCACHE0(CB0:0-15) KCACHE1(CB1:0-15)
     5  w: ADD_INT    ____,  R0.x,  R1.x
     6  z: ADD_INT    ____,  PV5.w,  KC0[6].x
     7  y: LSHL       ____,  PV6.z,  2
     8  x: ADD_INT    ____,  KC1[1].x,  PV7.y
     9  x: LSHR       R0.x,  PV8.x,  2
03 MEM_RAT_CACHELESS_STORE_DWORD__NI: RAT(11)[R0].x___, R2, ARRAY_SIZE(4)  MARK  VPM
04 END
END_OF_PROGRAM
```

Clause header
Work executed by the shared scalar unit

VLIW instruction packet
Compiler-generated instruction level parallelism for the VLIW unit.
Each instruction (x, y, z, w) executed across the vector.

Clause body
Units of work dispatched by the shared scalar unit

Notice the poor occupancy of VLIW slots

WHY DID WE SEE INEFFICIENCY?

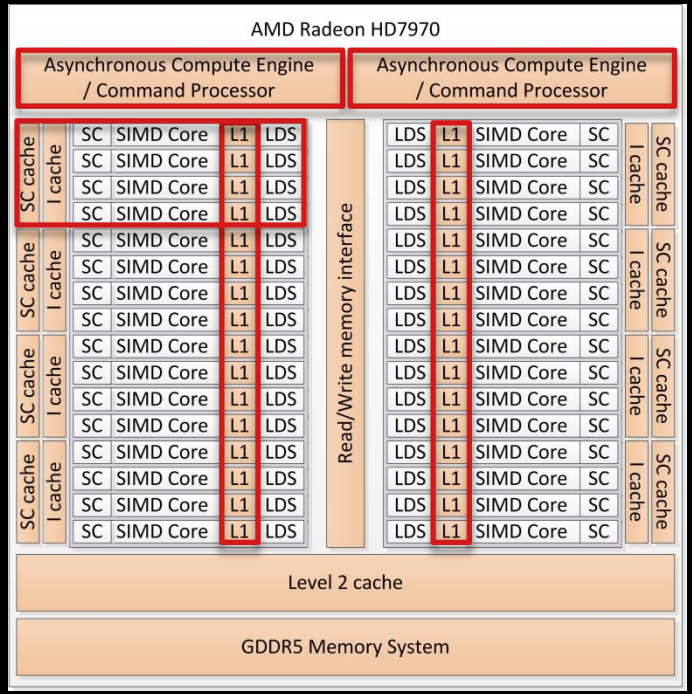
- The architecture was well suited to graphics workloads:
  - VLIW was easily filled by the vector-heavy graphics kernels
  - Minimal control flow meant that the monolithic, shared thread scheduler was relatively efficient

- Unfortunately, workloads change with time.

- So how did we change the architecture to improve the situation?

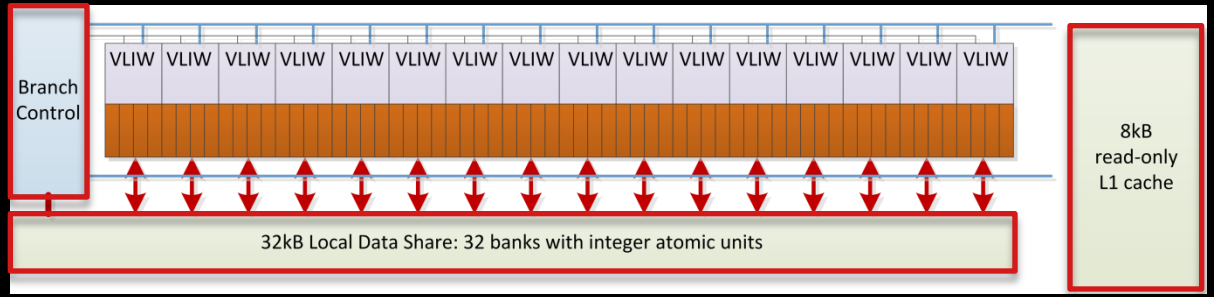
AMD RADEON HD7970 - GLOBALLY

- Brand new – but at this level it doesn't look too different
- Two command processors
  - Capable of processing two command queues concurrently
- Full read/write L1 data caches
- SIMD cores grouped in fours
  - Scalar data and instruction cache per cluster
  - L1, LDS and scalar processor per core
- Up to 32 cores / compute units



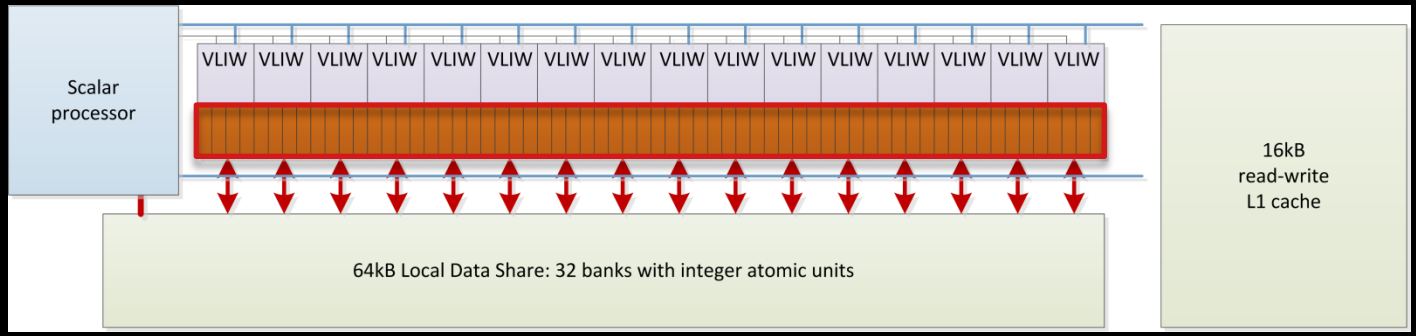
THE SIMD CORE

- The SIMD unit on the HD6970 architecture had a branch control but full scalar execution was performed globally



THE SIMD CORE

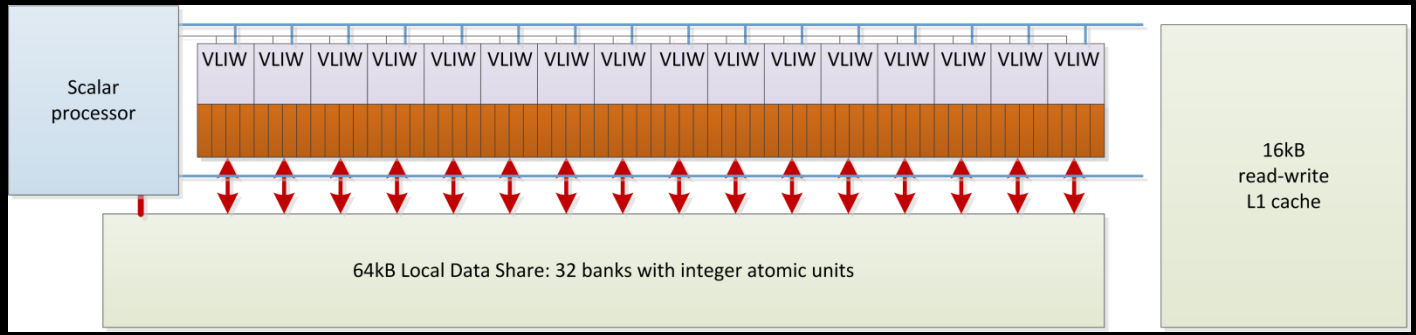
- On the HD7970 we have a full scalar processor and the L1 cache and LDS have been doubled in size
- Then let us consider the VLIW ALUs



AMD

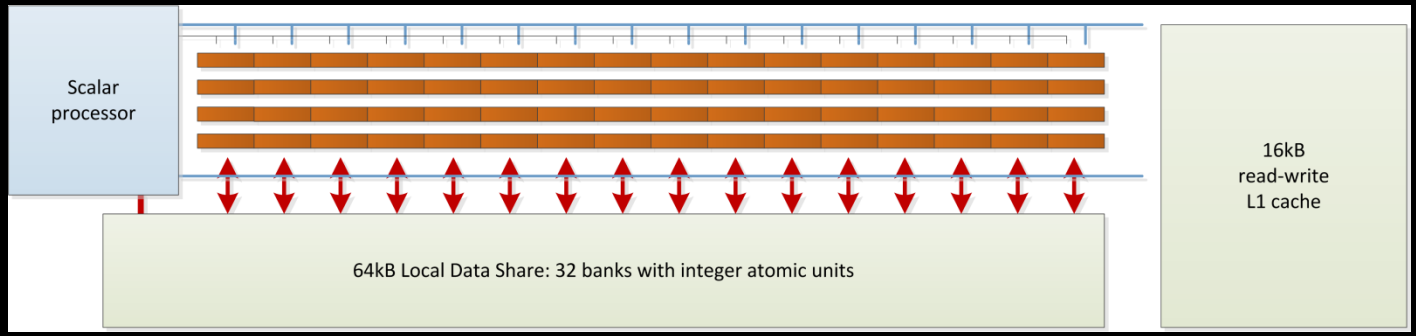
THE SIMD CORE

- Remember we could view the architecture two ways:
  - An array of VLIW units
  - A VLIW cluster of vector units



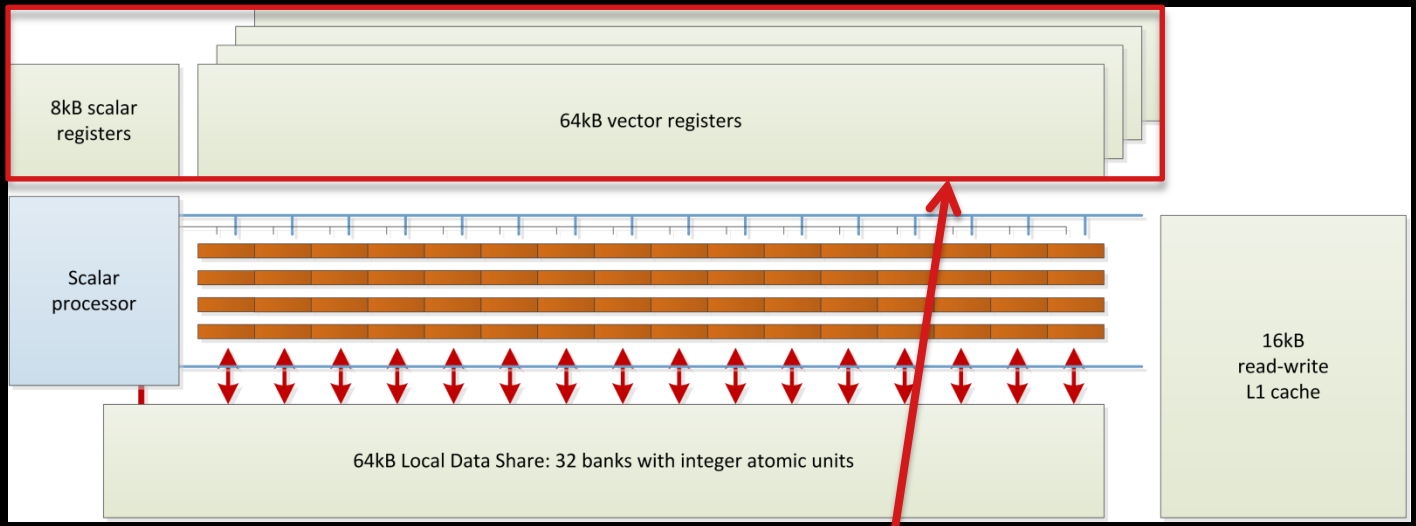
THE SIMD CORE

- Now that we have a scalar processor we can dynamically schedule instructions rather than relying on the compiler
- No VLIW!



- The heart of Graphics Core Next:
  - A scalar processor with four 16-wide vector units
  - Each lane of the vector, and hence each IL work item, is now scalar

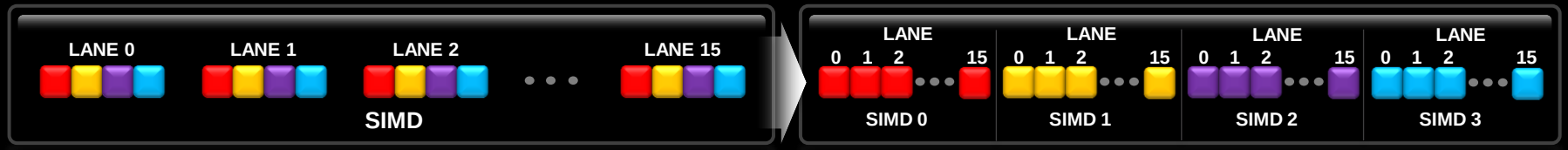
THE SIMD CORE



- The scalar core manages a large number of threads
  - Each thread requires its set of vector registers
  - Significant register state for both scalar and vector storage
  - 10 waves per SIMD, 40 waves per CU (core), 2560 work items per CU, 81920 work items on the HD7970

AMD

VECTOR UNITS



VLIW4 SIMD

- 64 Single Precision multiply-add
- 1 VLIW Instruction × 4 ALU ops ➔ dependency limited
- Compiler manages register port conflicts
- Specialized, complex compiler scheduling
- Difficult assembly creation, analysis, and debug
- Complicated tool chain support
- Careful optimization required for peak performance

GCN Quad SIMD

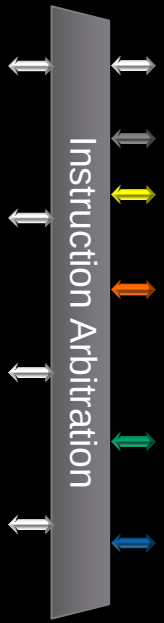
- 64 Single Precision multiply-add
- 4 SIMDs × 1 ALU op ➔ occupancy limited
- No register port conflicts
- Standardized compiler scheduling & optimizations
- Simplified assembly creation, analysis, and debug
- Simplified tool chain development and support
- Stable and predictable performance

THE NEW ISA

- Simpler and more efficient
- Instructions for both sets of execution units inline
- No clauses
  - Lower instruction scheduling latency
  - Improved performance in previously clause-bound cases
  - Lower power handling of control flow as control is closer
- No VLIW
  - Fewer compiler-induced bubbles in the instruction schedule
- Full support for exceptions, function calls and recusion

```
s_buffer_load_dword  s0, s[4:7], 0x04
s_buffer_load_dword  s1, s[4:7], 0x18
s_buffer_load_dword  s4, s[8:11], 0x00
s_waitcnt     lgkmcnt(0)
s_mul_i32     s0, s12, s0
s_add_i32     s0, s0, s1
v_add_i32     v0, vcc, s0, v0
v_lshlrev_b32  v0, 2, v0
v_add_i32     v1, vcc, s4, v0
s_load_dwordx4  s[4:7], s[2:3], 0x50
s_waitcnt     lgkmcnt(0)
tbuffer_load_format_x  v1, v1, s[4:7],
    0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
s_buffer_load_dword  s0, s[8:11], 0x04
s_load_dwordx4  s[4:7], s[2:3], 0x58
s_waitcnt     lgkmcnt(0)
v_add_i32     v0, vcc, s0, v0
s_waitcnt     vmcnt(0)
tbuffer_store_format_x  v1, v0, s[4:7],
    0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
s_endpgm
```

INSTRUCTION ARBITRATION AND DECODE



- A Kernel freely mixes instruction types (Simplistic Programming Model, no weird rules)
  - Scalar/Scalar Memory, Vector, Vector Memory, Shared Memory, etc.
- Every clock cycle, waves on one SIMDs are considered for instruction issue.
  - Four cycles to execute, four SIMDs…
  - At most one instruction per category
  - At most one instruction per wave
- Up to a maximum of 5 instructions can issue per cycle, not including “internal” instructions.
  - 1 Vector Arithmetic Logic Unit (ALU)
  - 1 Scalar ALU or Scalar Memory Read
  - 1Vector memory access (Read/Write/Atomic)
  - 1 Branch/Message - s_branch and s_cbranch_<cond>
  - 1 Local Data Share (LDS)
  - 1 Export or Global Data Share (GDS)
  - 1 Internal (s_nop, s_sleep, s_waitcnt, s_barrier, s_setprio)

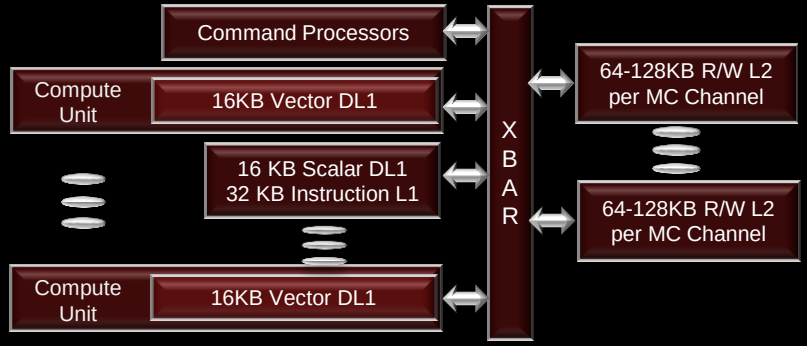
BRANCHING

```
float fn0(float a,float b)
{
   if(a>b)
      return((a-b)*a);
   else
      return((b-a)*b);
}
```

Optional:
Use based on the number of instruction in conditional section.
- Executed in branch unit

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2

v_cmp_gt_f32      r0,r1          //a > b, establish VCC
s_mov_b64         s0,exec        //Save current exec mask
s_and_b64         exec,vcc,exec  //Do "if"
s_cbranch_vccz    label0         //Branch if all lanes fail
v_sub_f32         r2,r0,r1       //result = a - b
v_mul_f32         r2,r2,r0       //result=result * a

    :

s_andn2_b64       exec,s0,exec   //Do "else"(s0 & !exec)
s_cbranch_execz   label1         //Branch if all lanes fail
v_sub_f32         r2,r1,r0       //result = b - a
v_mul_f32         r2,r2,r1       //result = result * b

s_mov_b64         exec,s0        //Restore exec mask
```

R/W CACHE



- Reads and writes cached
  - Bandwidth amplification
  - Improved behavior on more memory access patterns
  - Improved write to read reuse performance
- Relaxed memory model
  - Consistency controls available to control locality of load/store
- GPU Coherent
  - Acquire/Release semantics control data visibility across the machine (GLC bit on load/store)
  - L2 coherent = all CUs can have the same view of data
- Global atomics
  - Performed in L2 cache
- ECC protection for DRAM and SRAM

SUMMARY

- An architecture designed to make GPGPU easier while maintaining high levels of power efficiency
  - Nearly 1 GFLOP/s double precision throughput
  - Higher efficiency of execution
    - No VLIW
    - Efficient dynamic vector/scalar instruction scheduling
  - ECC protection for memory system
  - Read/write caching

- Combined with advances in the OpenCL C++ support to make programming easier

QUESTIONS

Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.