CS4803DGC Design and Programming of Game Console

Spring 2011

Prof. Hyesoon Kim
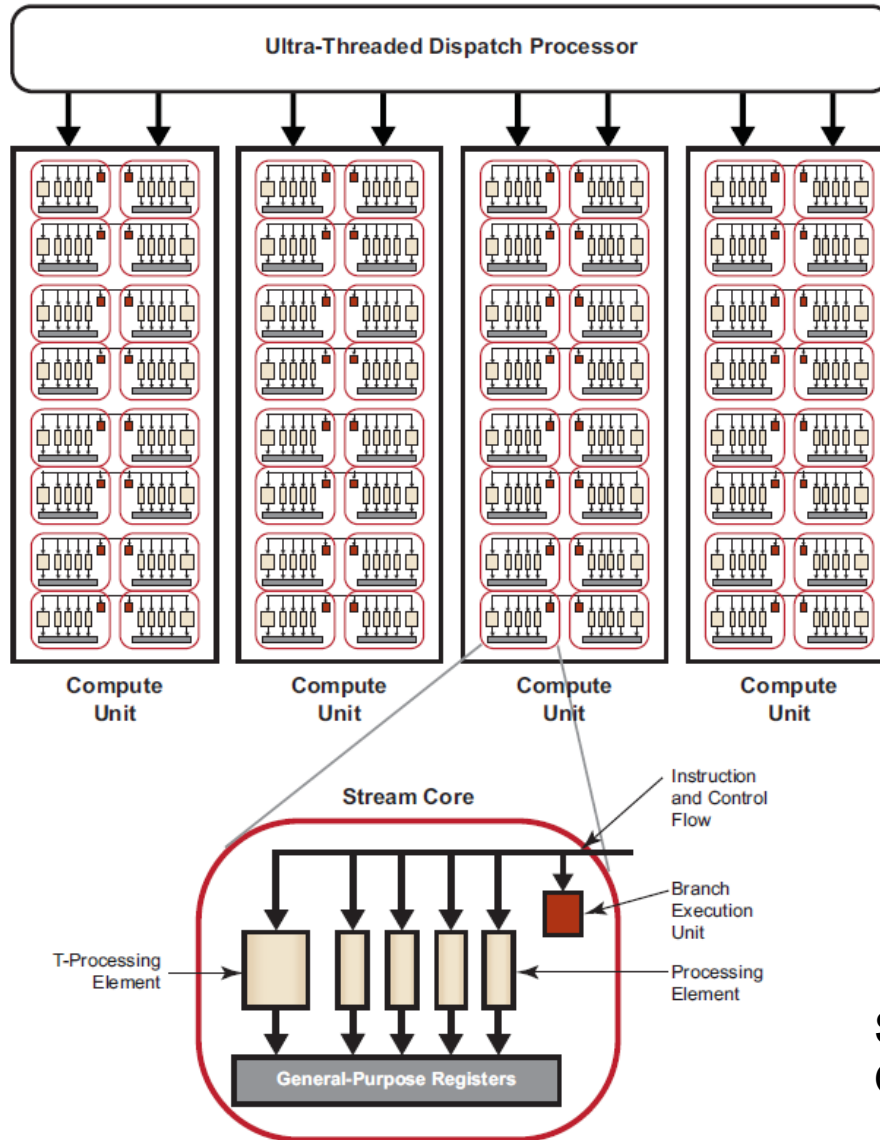
Georgia Tech | College of Computing

ATI Architecture Overview

Source: AMD Accelerated Parallel Processing OpenCL Programming Guide
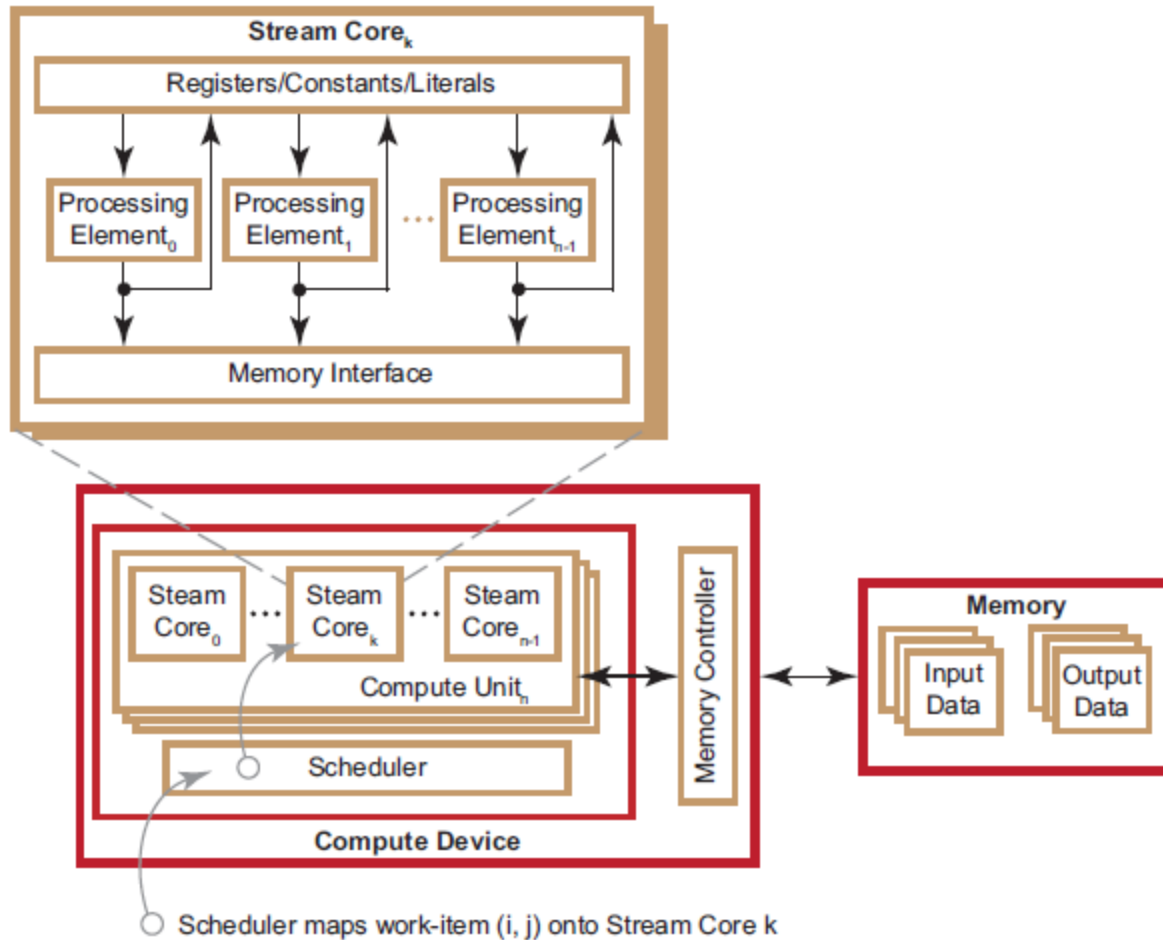
OpenCL and AMD GPUs

Figure 1.4 Simplified Mapping of OpenCL onto AMD Accelerated Parallel Processing

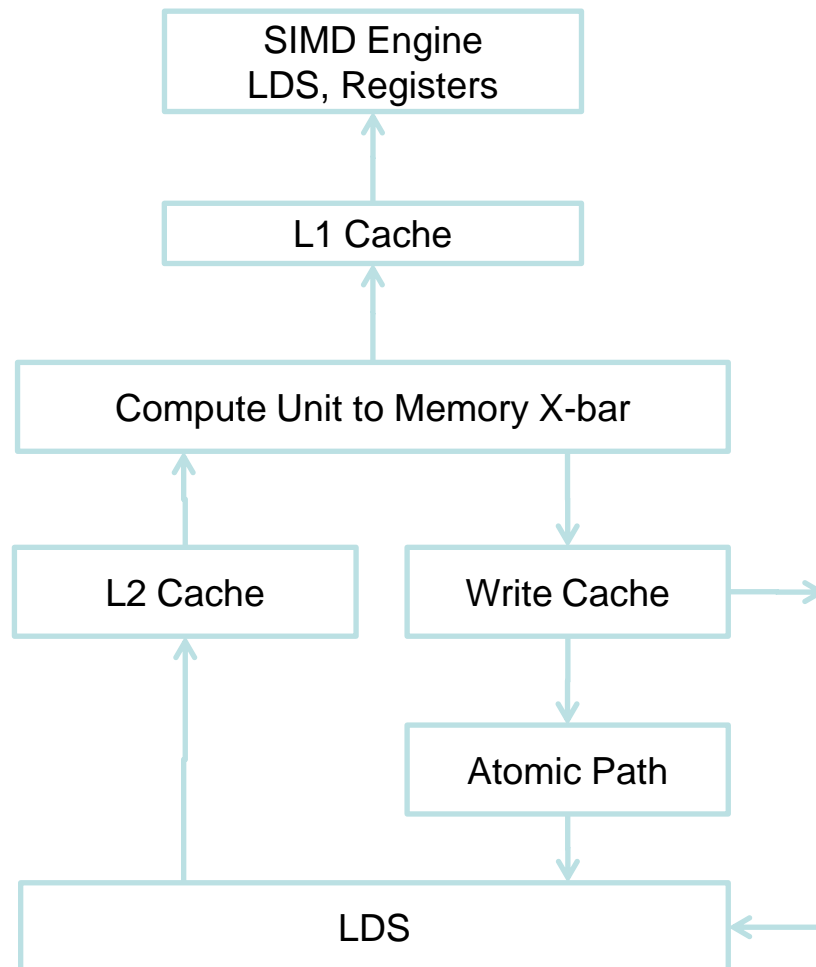# Work-Item, Work-Group, Wavefront
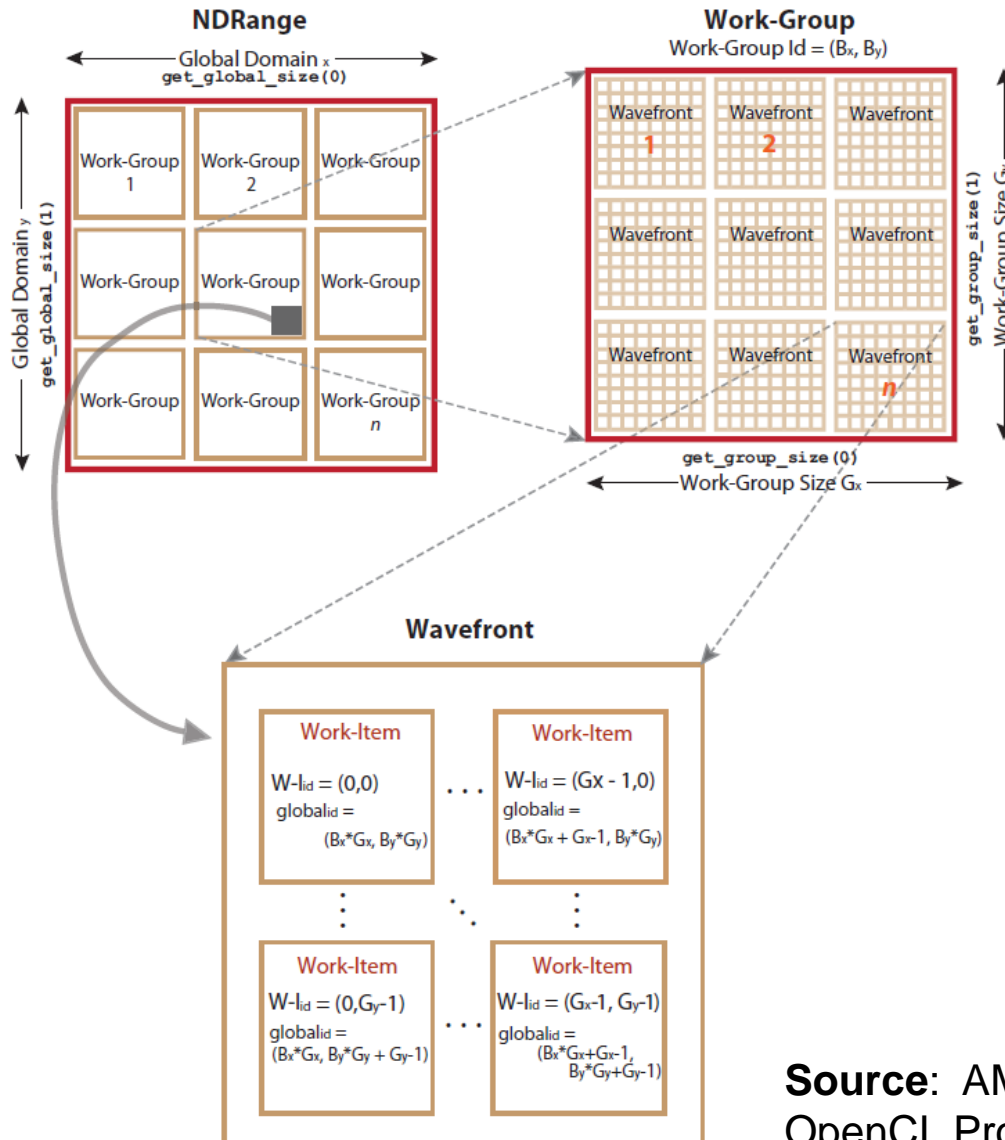
OpenCL : CUDA
Work-item: thread
Work-group: Block

AMD : NVIDIA
Wavefront: warp

Source: AMD Accelerated Parallel Processing
OpenCL Programming Guide

AMD GPU Memory Architecture

```
        ┌─────────────────────────┐
        │      SIMD Engine        │
        │     LDS, Registers      │
        └─────────────────────────┘
                    ▲
        ┌─────────────────────────┐
        │        L1 Cache         │
        └─────────────────────────┘
                    ▲
  ┌───────────────────────────────────────┐
  │     Compute Unit to Memory X-bar      │
  └───────────────────────────────────────┘
        ▲                    │
        │                    ▼
  ┌────────────┐      ┌────────────────┐
  │  L2 Cache  │      │  Write Cache   │───┐
  └────────────┘      └────────────────┘   │
        ▲                    │             │
        │                    ▼             │
        │             ┌────────────────┐   │
        │             │  Atomic Path   │   │
        │             └────────────────┘   │
        │                    │             │
  ┌───────────────────────────────────┐   │
  │                LDS                │◄──┘
  └───────────────────────────────────┘
```
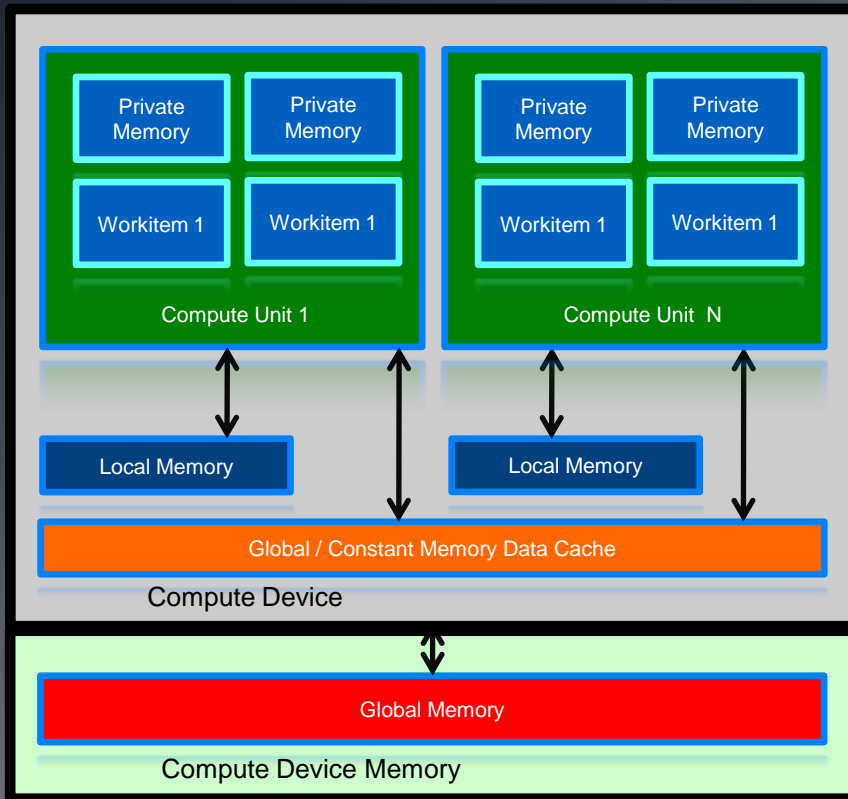
- Memory per compute unit
  - Local data store (on-chip)
  - Registers

  - L1 cache (8KB for 5870) per compute unit

- L2 Cache shared between compute units (512KB for 5870)

- Fast path for only 32 bit operations { no atomic}

- Complete path for atomics and < 32bit operations

Georgia Tech | College of Computing

SIMT Execution Model

- SIMD execution can be combined with pipelining

    - ALUs all execute the same instruction

    - Pipelining is used to break instruction into phases

        - When first instruction completes (4 cycles here), the next instruction is ready to execute

SIMD Width

Wavefront

| Add | Add | Add | Add | Mul | Mul | Mul | Mul | ... |

Cycle	1	2	3	4	5	6	7	8	9	...
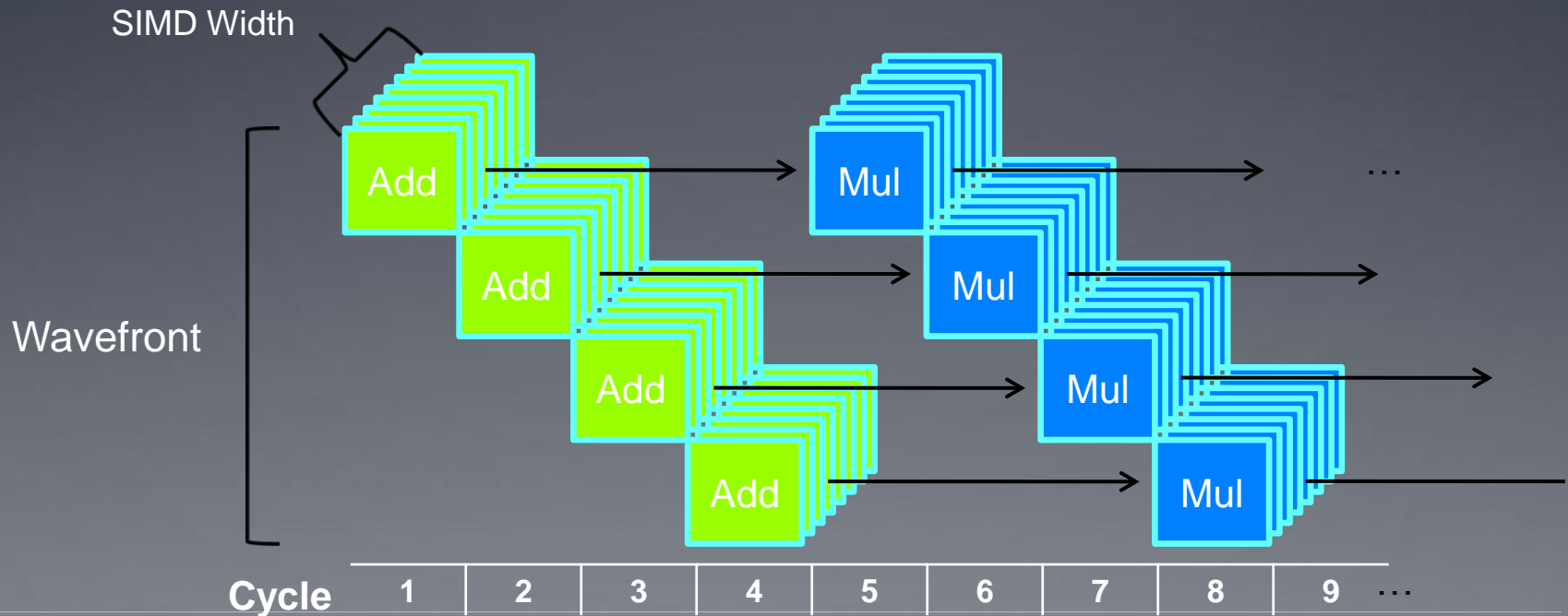
AMD Memory Model in OpenCL

- Subset of hardware memory exposed in OpenCL

- Local Data Share (LDS) exposed as local memory
  - Share data between items of a work group designed to increase performance
  - High Bandwidth access per SIMD Engine

- Private memory utilizes registers per work item

- Constant Memory
  - __constant tags utilize L1 cache.

AMD Constant Memory Usage

- Constant Memory declarations for AMD GPUs only beneficial for following access patterns

  - **Direct-Addressing Patterns:** For non array constant values where the address is known initially

  - **Same Index Patterns:** When all work-items reference the same constant address

  - **Globally scoped constant arrays:** Arrays that are initialized, globally scoped can use the cache if less than 16KB

- Cases where each work item accesses different indices, are not cached and deliver the same performance as a global memory read

**Source**: AMD Accelerated Parallel Processing OpenCL Programming Guide

Half-warp vs. Quarter-wavefront

- For the AMD 5870 GPU, memory accesses of 16 consecutive threads are evaluated together and can be coalesced to fully utilize the bus
    - This unit is called a quarter-wavefront

- Both NVIDIA and ATI use 16 consecutive threads as the minimum memory traffics
- 16*4B = 64B 16*8B = 128B

Coalescing Memory Accesses

- Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on an NVIDIA GTX 285
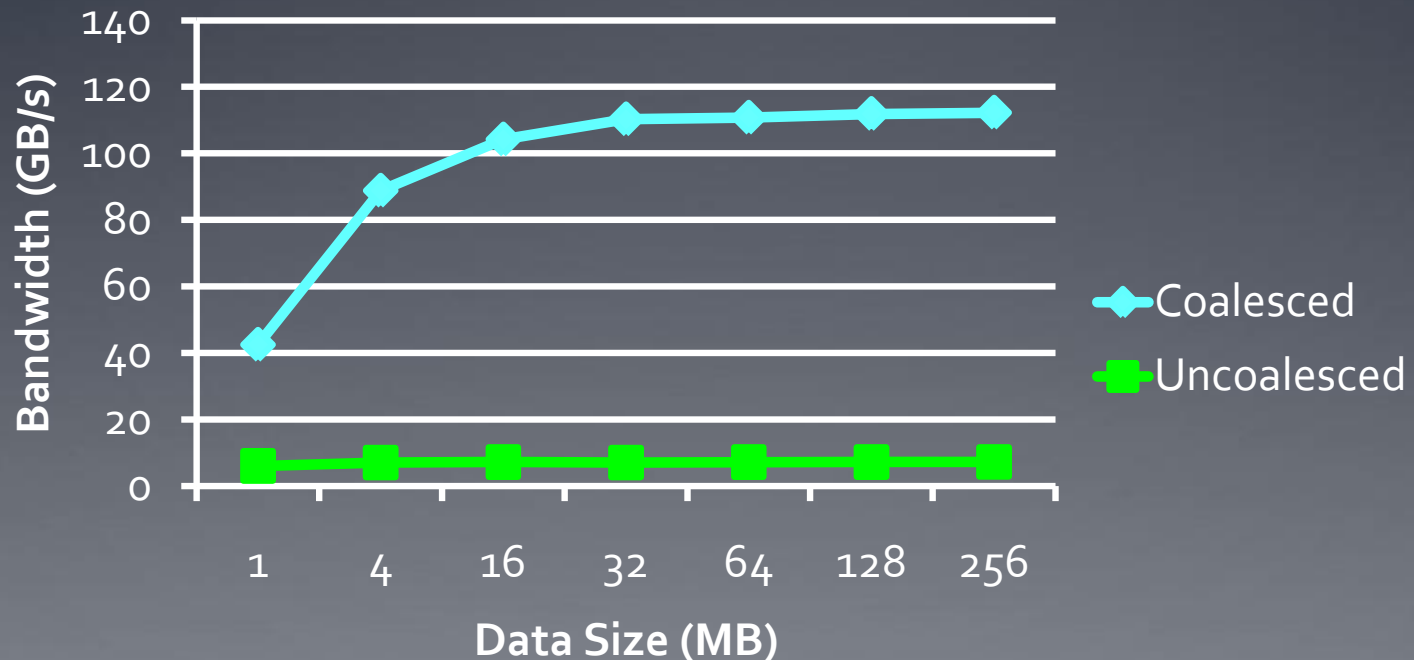
Memory Banks

- Memory is made up of *banks*
  - Memory banks are the hardware units that actually store data

- The memory banks targeted by a memory access depend on the address of the data to be read/written
  - Note that on current GPUs, there are more memory banks than can be addressed at once by the global memory bus, so it is possible for different accesses to target different banks
    - Bank response time, not access requests, is the bottleneck

- Successive data are stored in successive banks (strides of 32-bit words on GPUs) so that a group of threads accessing successive elements will produce no bank conflicts

Bank Conflicts – Local Memory

- Bank conflicts have the largest negative effect on local memory operations
  - Local memory does not require that accesses are to sequentially increasing elements

- Accesses from successive threads should target different memory banks
  - Threads accessing sequentially increasing data will fall into this category

Bank Conflicts – Local Memory

- On AMD, a wavefront that generates bank conflicts stalls until all local memory operations complete

  - The hardware does not hide the stall by switching to another wavefront

- The following examples show local memory access patterns and whether conflicts are generated

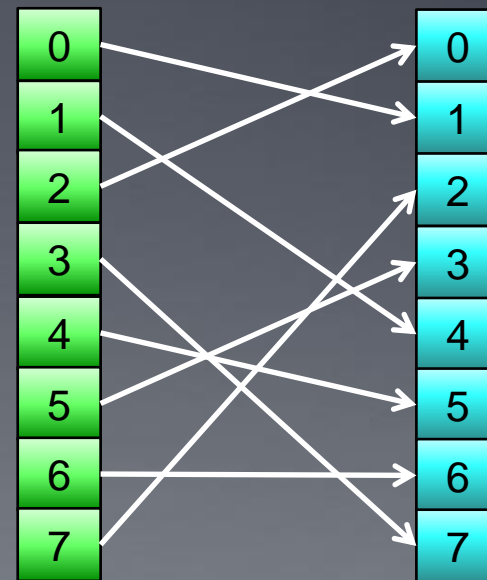 - For readability, only 8 memory banks are shown

Bank Conflicts – Local Memory

- If there are no bank conflicts, each bank can return an element without any delays
  - Both of the following patterns will complete without stalls on current GPU hardware
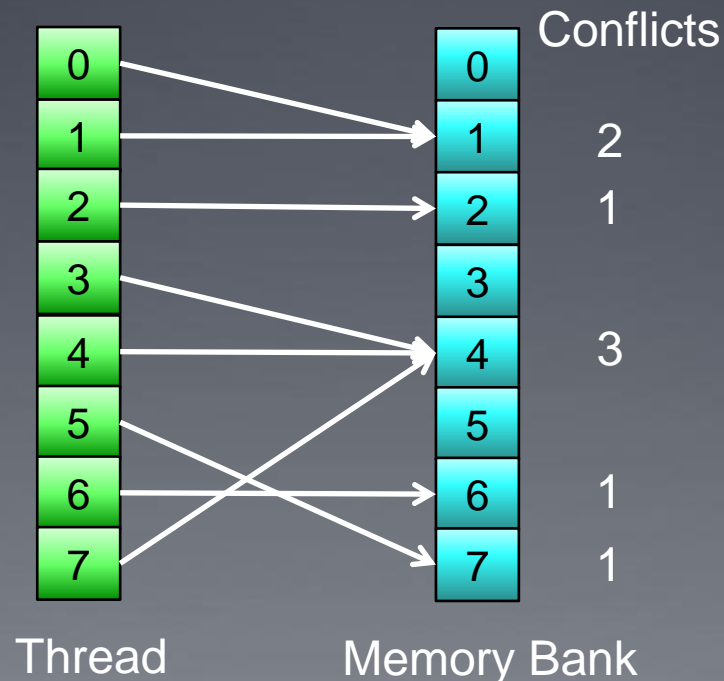

Thread   Memory Bank   Thread   Memory Bank

Bank Conflicts – Local Memory

- If multiple accesses occur to the same bank, then the bank with the most conflicts will determine the latency
  - The following pattern will take 3 times the access latency to complete

Bank Conflicts – Local Memory

- If all accesses are to the same address, then the bank can perform a broadcast and no delay is incurred
  - The following will only take one access to complete assuming the same data element is accessed
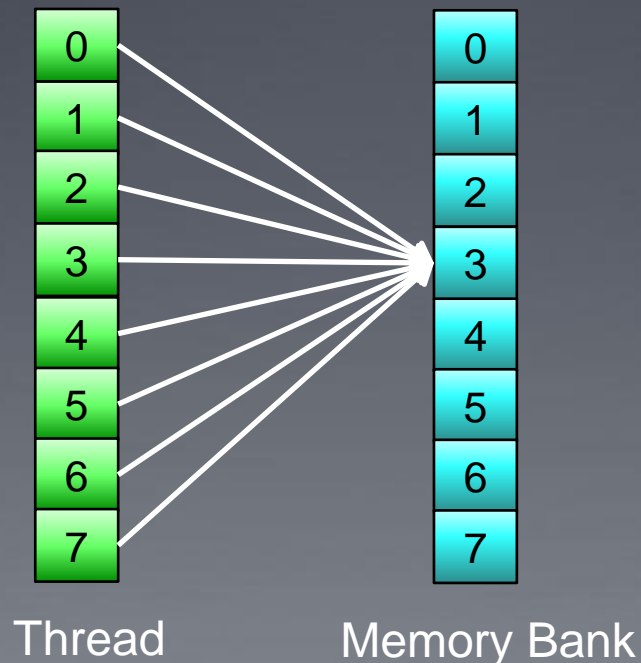
Thread   Memory Bank

Bank Conflicts – Global Memory

- Bank conflicts in global memory rely on the same principles, however the global memory bus makes the impact of conflicts more subtle
  - Since accessing data in global memory requires that an entire bus-line be read, bank conflicts within a work-group have a similar effect as non-coalesced accesses
    - If threads reading from global memory had a bank conflict then by definition it manifest as a non-coalesced access
    - Not all non-coalesced accesses are bank conflicts, however

- The ideal case for global memory is when different work-groups read from different banks
  - In reality, this is a very low-level optimization and should not be prioritized when first writing a program

Occupancy

- On current GPUs, work groups get mapped to compute units
  - When a work group is mapped to a compute unit, it cannot be swapped off until all of its threads complete their execution

- If there are enough resources available, multiple work groups can be mapped to the same compute unit at the same time
  - Wavefronts from another work group can be swapped in to hide latency

- Resources are fixed per compute unit (number of registers, local memory size, maximum number of threads)
  - Any one of these resource constraints may limit the number of work groups on a compute unit

- The term *occupancy* is used to describe how well the resources of the compute unit are being utilized

Occupancy – Registers

- The availability of registers is one of the major limiting factor for larger kernels

- The maximum number of registers required by a kernel must be available for all threads of a workgroup
  - Example: Consider a GPU with 16384 registers per compute unit running a kernel that requires 35 registers per thread
    - Each compute unit can execute at most 468 threads
    - This affects the choice of workgroup size
      - A workgroup of 512 is not possible
      - Only 1 workgroup of 256 threads is allowed at a time, even though 212 more threads could be running
      - 3 workgroups of 128 threads are allowed, providing 384 threads to be scheduled, etc.

Occupancy – Registers

- Consider another example:
  - A GPU has 16384 registers per compute unit
  - The work group size of a kernel is fixed at 256 threads
  - The kernel currently requires 17 registers per thread

- Given the information, each work group requires 4352 registers
  - This allows for 3 active work groups if registers are the only limiting factor

- If the code can be restructured to only use 16 registers, then 4 active work groups would be possible

Occupancy – Local Memory

- GPUs have a limited amount of local memory on each compute unit
  - 32KB of local memory on AMD GPUs
  - 32-48KB of local memory on NVIDIA GPUs

- Local memory limits the number of active work groups per compute unit

- Depending on the kernel, the data per workgroup may be fixed regardless of number of threads (e.g., histograms), or may vary based on the number of threads (e.g., matrix multiplication, convolution)

Occupancy – Threads

- GPUs have hardware limitations on the maximum number of threads per work group
  - 256 threads per WG on AMD GPUs
  - 512 threads per WG on NVIDIA GPUs

- NVIDIA GPUs have per-compute-unit limits on the number of active threads and work groups (depending on the GPU model)
  - 768 or 1024 threads per compute unit
  - 8 or 16 warps per compute unit

- AMD GPUs have GPU-wide limits on the number of wavefronts
  - 496 wavefronts on the 5870 GPU (~25 wavefronts or ~1600 threads per compute unit)

Occupancy – Limiting Factors

- The minimum of these three factors is what limits the active number of threads (or occupancy) of a compute unit

- The interactions between the factors are complex
    - The limiting factor may have either thread or wavefront granularity
    - Changing work group size may affect register or shared memory usage
    - Reducing any factor (such as register usage) slightly may have allow another work group to be active

- The CUDA occupancy calculator from NVIDIA plots these factors visually allowing the tradeoffs to be visualized

Vectorization

- On AMD GPUs, each processing element executes a 5-way VLIW instruction
  - 5 scalar operations or
  - 4 scalar operations + 1 transcendental operation

Compute Unit

PE_0 PE_1 PE_2 · · · PE_{n-1}

ALU + T-unit

ALU

Incoming Instruction

Registers

Branch Unit
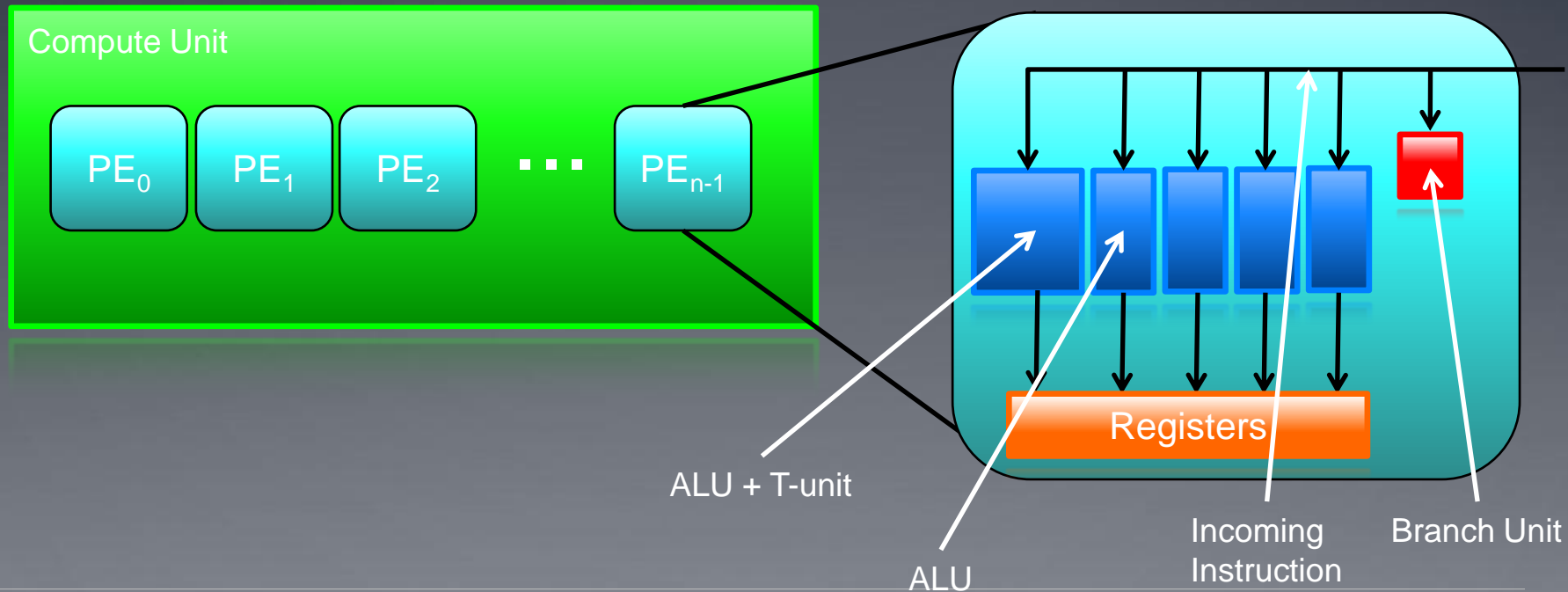
Vectorization

- Vectorization allows a single thread to perform multiple operations at once

- Explicit vectorization is achieved by using vector datatypes (such as `float4`) in the source program
  - When a number is appended to a datatype, the datatype becomes an array of that length
  - Operations can be performed on vector datatypes just like regular datatypes
    - Each ALU will operate on different element of the float4 data

Vectorization

- Vectorization improves memory performance on AMD GPUs

  - The *AMD Accelerated Parallel Processing OpenCL Programming Guide* compares `float` to `float4` memory bandwidth

```
__kernel void
Copy4(__global const float4 * input,
       __global float4 * output)
{
    int gid = get_global_id(0);
    output[gid] = input[gid];
    return;
}

__kernel void
Copy1(__global const float * input,
       __global float * output)
{
    int gid = get_global_id(0);
    output[gid] = input[gid];
    return;
}
```