Vector, SIMD, & GPUs

Module 10

Module Syllabus

- Vector processors
- Single instruction, multiple data (SIMD) architectures
- SIMD case study
- Graphics processing units (GPUs)
- GPU case study

Motivation

- Multicore processors and multithreading take advantage of thread-level parallelism.
  - Allows us to exploit thread-level parallelism (within a program) and process-level parallelism (across applications)
- They are relatively simple extensions to a conventional core (at least conceptually).
  - For multicore, duplicate the core several times on the chip and add extra logic (e.g., for coherence).
  - For multithreading, add storage for multiple thread contexts and associated pipeline changes.
- However, other architectural choices allow us to extract different forms of parallelism.
  - Such as data-level parallelism
- This module studies these architectures compared to a generic multicore.

Vector Processors

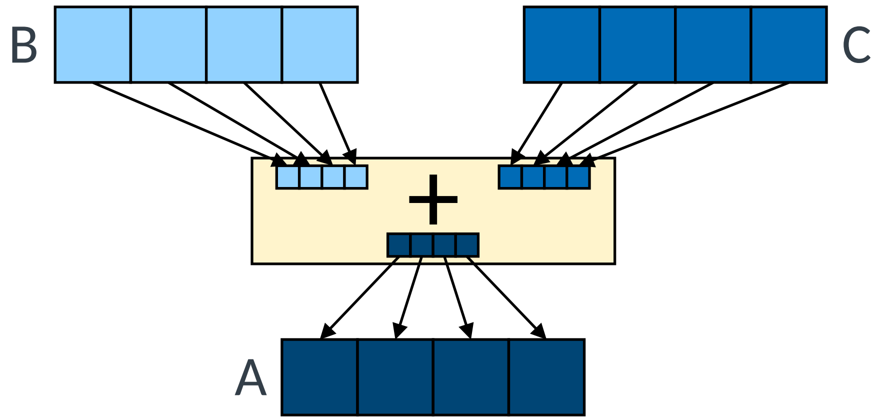
Vector Processors

- Vector processors explicitly exploit data-level parallelism.
  - When instructions are applied to multiple independent data items
- Vector processors gather data from memory into large (vector) registers.
  - Then, conceptually perform operations on these items together
  - Essentially performing many register-register operations with the same opcode
  - Results are scattered back out to memory.
- For vectorizable data, vector processors:
  - Provide energy-efficient computation (amortizing the costs of fetch and decode).
  - Hide memory latencies through pipelining operations.

Example Executing Code on a Vector Processor

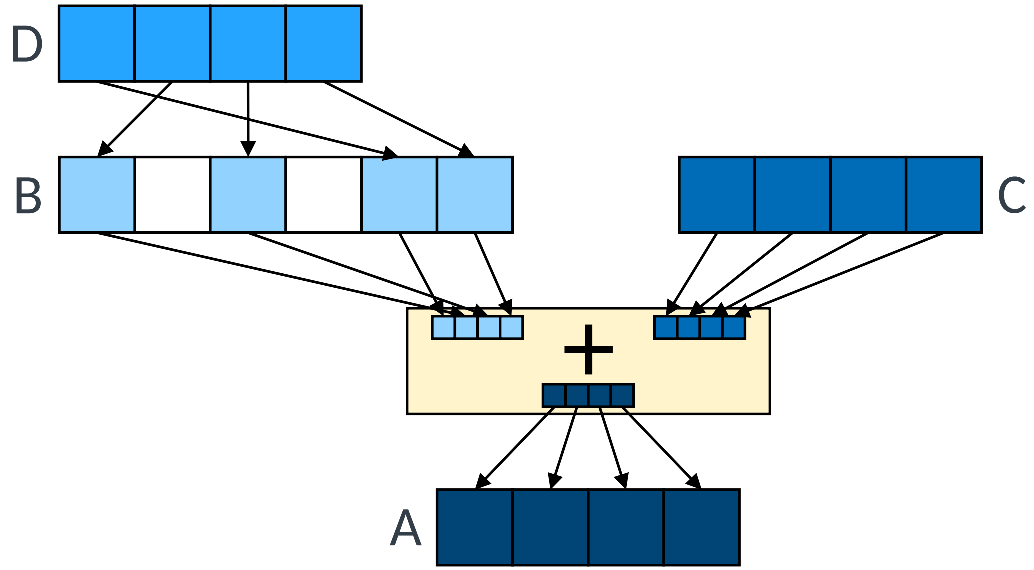
Contiguous data in memory

```
for (int i=0; i<64; ++i) {
  A[i] = B[i] + C[i]
}
```



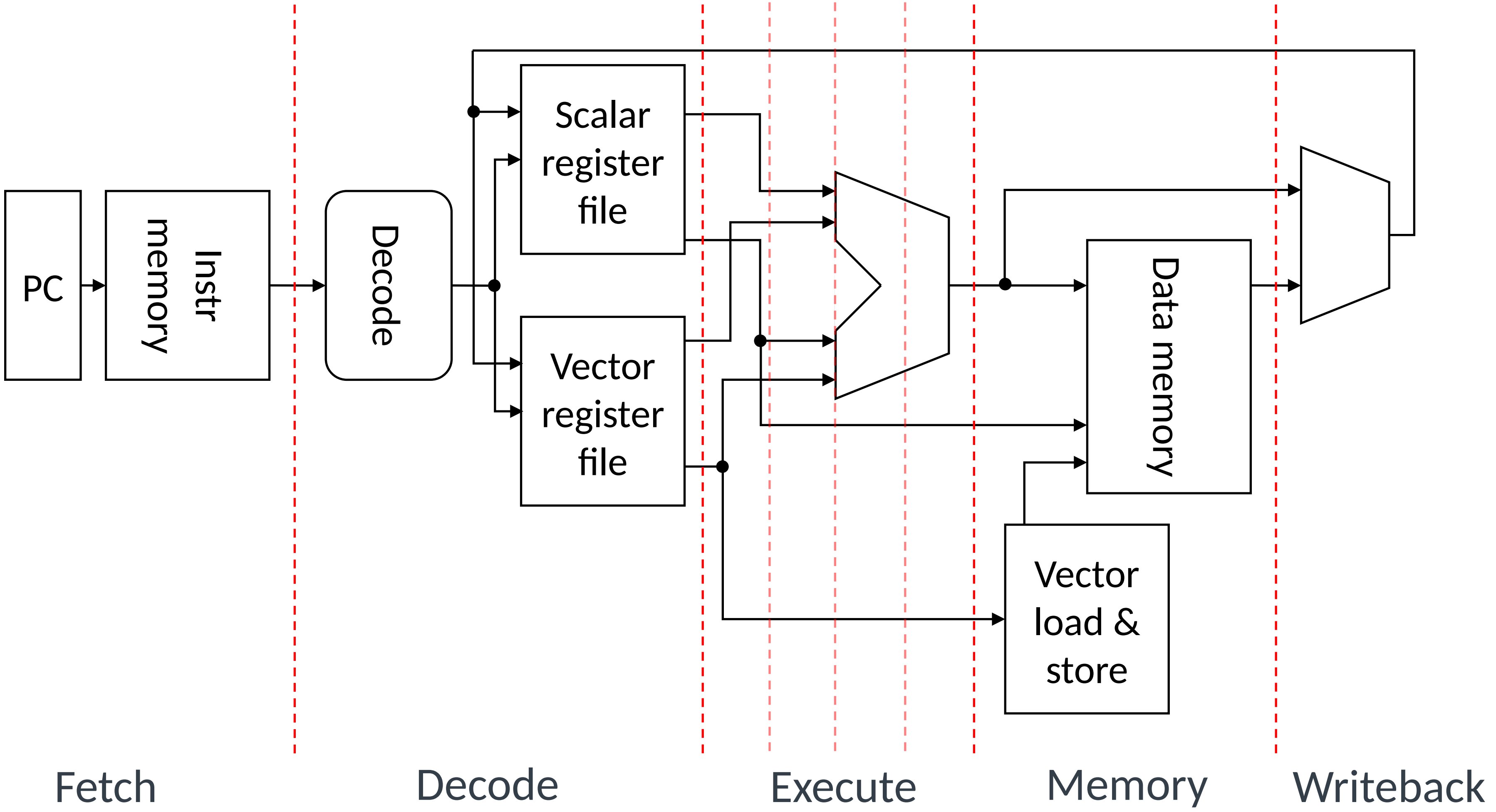
Dispersed data in memory

```
for (int i=0; i<64; ++i) {
  A[i] = B[D[i]] + C[i]
}
```



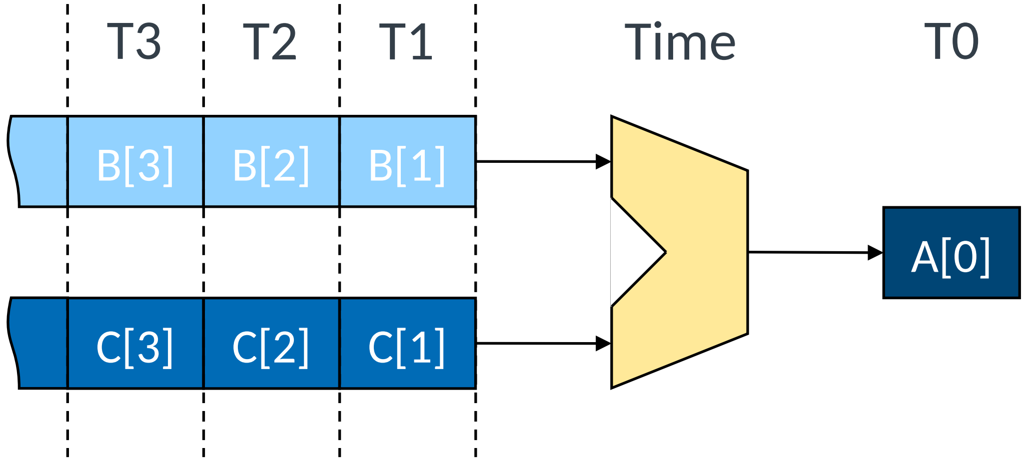
An Example Vector Processor

- Vector functional units fed by
  - Vector registers provide independent data to operate on.
  - Scalar registers provide additional data or memory addresses.
- Vector FUs are fully pipelined.
  - To start processing a new element on each clock cycle



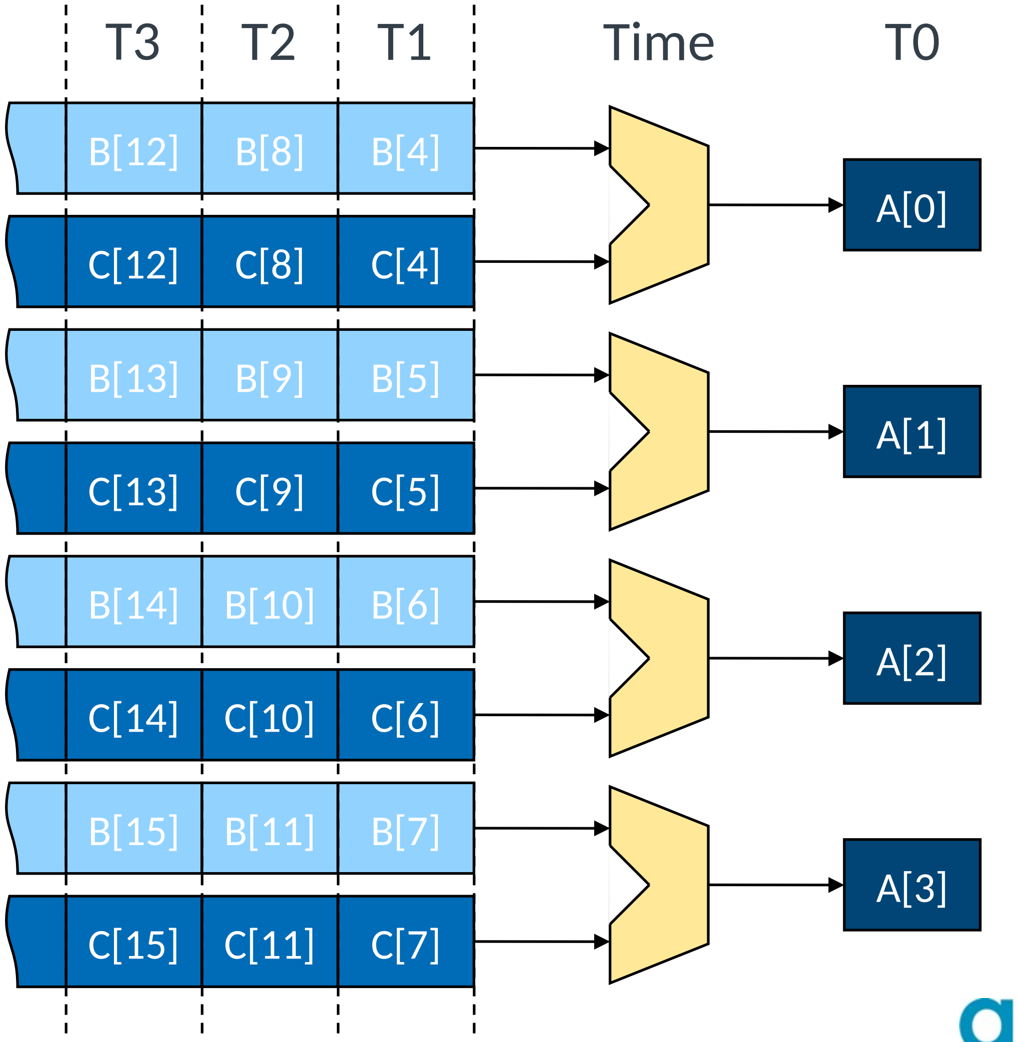
Instruction Execution

- The core operates on vectors of data
  - However, this doesn't mean that the whole vector must be processed at once.
- This is a simple vector functional unit.
  - It can only process one element at a time.
- Operation on a new data element starts on each clock cycle.
  - The FU is pipelined to support multi-cycle operations.
- The ALU needs as many cycles as there are vector elements to process all data.
  - Plus the time to perform one operation



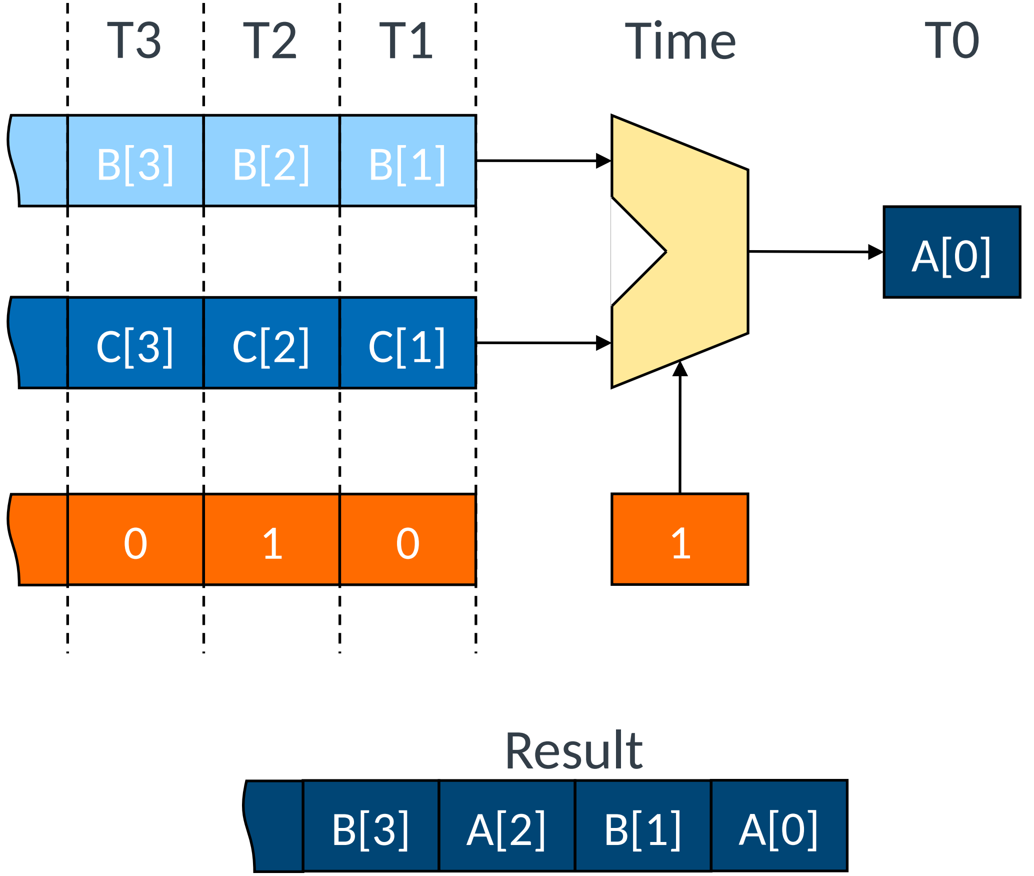
Instruction Execution

- Duplicating the functional units increases performance.
  - Since we now start execution on multiple data elements on each clock cycle
- To help this, we can partition the vector register file into lanes.
  - With one functional unit of each type per lane
  - Data elements are interleaved across lanes to produce data in the correct order.



Predication

- Sometimes you don't want to do computation across the whole vector.
  - Only certain elements of the computation should be performed.
- Predication is one method for achieving this.
- Include one or more predicate registers.
  - Usually, these contain one bit per vector element.
- Bits from the predicate register say whether the operation on the corresponding elements goes ahead.
  - If not, some other value is placed in the destination.
  - E.g., forward the value from the first source



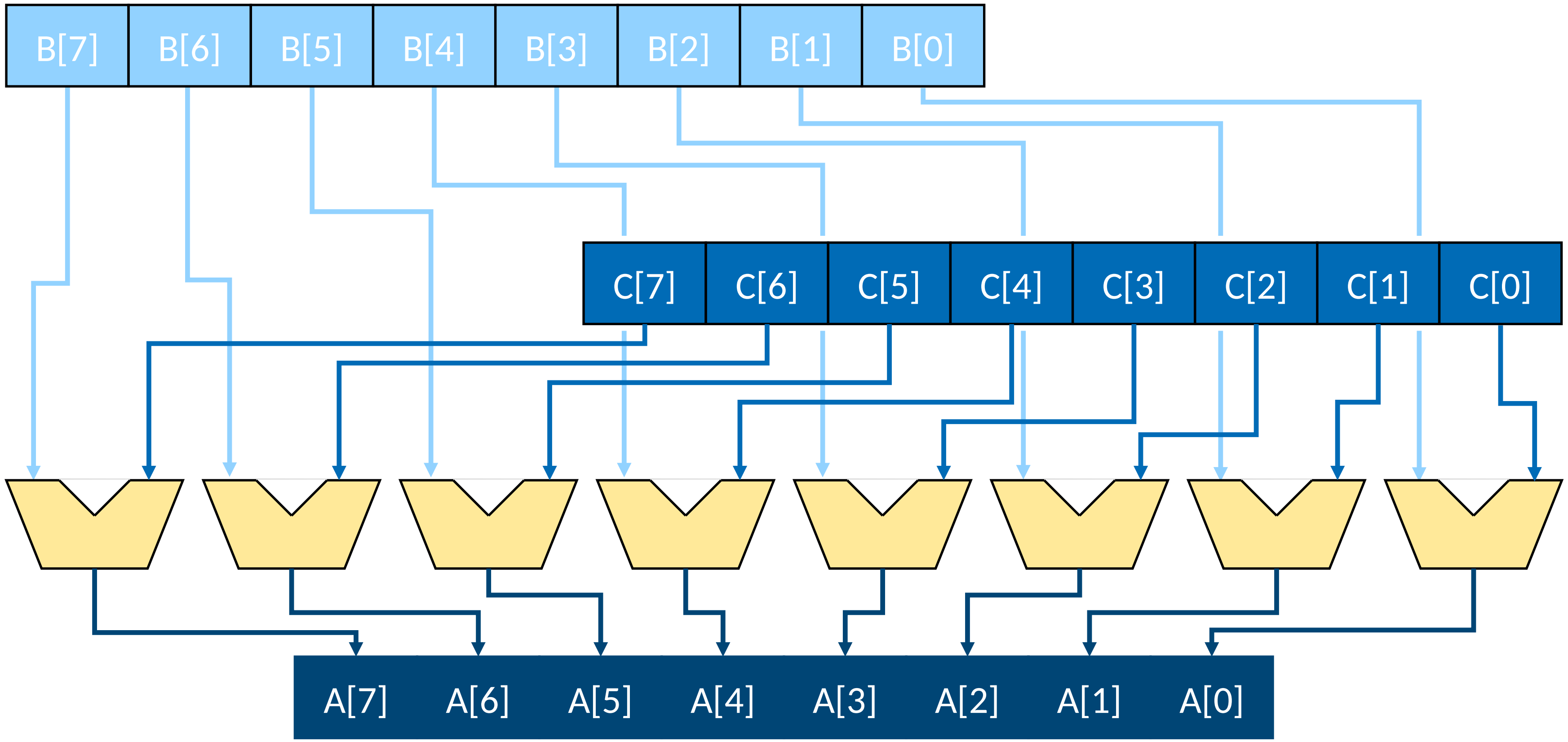
Single Instruction, Multiple Data (SIMD)

SIMD

- SIMD execution brings a (constrained) form of vector processing to general-purpose CPUs.
  - The key idea remains to exploit data-level parallelism.
  - And obtain performance benefits (with energy-efficiency benefits, too, if possible)
- SIMD also provides more efficient processing for certain codes.
  - E.g., multimedia workloads, such as image recognition and object detection, where operations are on pixel color values that are 8 or 16 bits in length
  - Scalar execution would put each value in its own 32-bit register.
  - SIMD packs them into a vector register (e.g., 256-bit vector of 16 * 16-bit data elements).
    - Then operates on each element independently
- However, SIMD typically operates on all elements concurrently.
  - Contrast with vector processing in the previous slides.

SIMD Execution

- In SIMD, operations on all data elements occur at the same time.
- The processor provides as many FUs as there are data elements in a vector.



SIMD vs Vector Processing

Vector processing

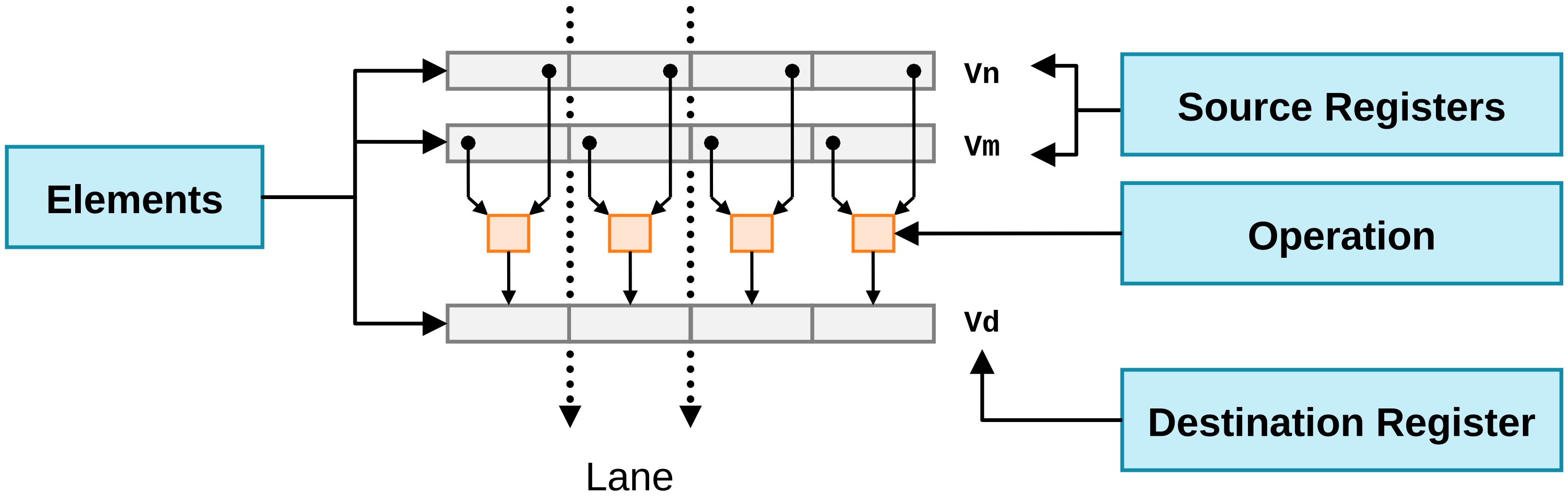
- Operations produce data for elements of the output vector over multiple cycles.
- Maximum vector length can change with each processor generation without requiring ISA changes.
- Best performance when the majority of the code is vectorizable – the scalar architecture is relatively simple.

SIMD

- Operations produce data for elements of the output vector in the same cycle.
- In most SIMD ISAs, the vector length is hardcoded, so increasing it requires new instructions.
- Vector operations are used to increase performance and generally augment a high-performance scalar architecture.

Case Study: AArch64 NEON

- NEON is a wide SIMD architecture developed for multimedia applications.
- Registers are considered as vectors of elements of the same data type (128 bits in size).
  - Integer signed and unsigned 8-bit, 16-bit, 32-bit, 64-bit
  - Floating point half, single and double precision
  - Instructions usually perform the same operation in all lanes.



Case Study: AArch64 SIMD Instruction Types

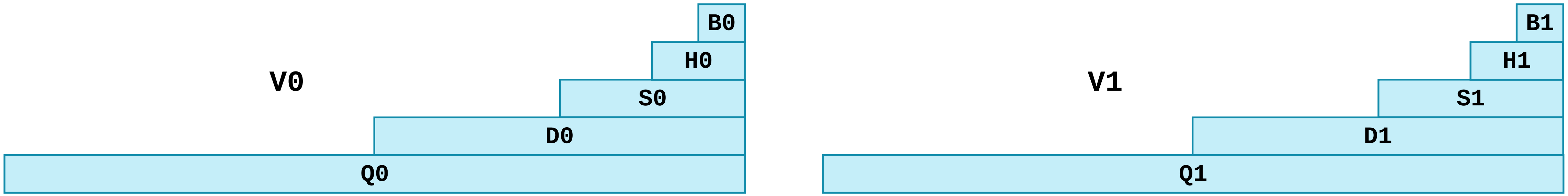
Type	Examples
Arithmetic	**ADD**, **SUB**, **MUL**, **MLS**, **SMIN**, **SMAX**
Saturating math	**UQADD**, **UQRSHL**, **SQDMULL**
Narrowing instructions	**SUBHN**, **ADDHN**, **RSUBHN**
Widening instructions	**SSUBL**, **UMULL2**, **UABDL2**
Integer compare	**CMGT**, **CMHS**, **CMTST**
Logical operations	**ORR**, **AND**, **BIC**, **EOR**
Floating point operations	**FADD**, **FABD**, **FDIV**, **FRINTZ**
Data movement	**DUP**, **INS**, **MOV**, **SMOV**, **UMOV**
Load/store instructions	**LDR**, **LDP**, **LD1**, **ST1**, **LD4R**

More information at https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools

arm

Case Study: AArch64 SIMD Register Bank

- Separate set of 32 registers, 128 bits wide, **`V0 - V31`**
- For access to a scalar
  - **`Qn`** to access 128-bit data (in **`Vn[127:0]`**)
  - **`Dn`** to access 64-bit data (in **`Vn[63:0]`**)
  - **`Sn`** to access 32-bit data (in **`Vn[31:0]`**)
  - **`Hn`** to access 16-bit data (in **`Vn[15:0]`**)
  - **`Bn`** to access 8-bit data (in **`Vn[7:0]`**)



Case Study: AArch64 SIMD Data Types and Sizes

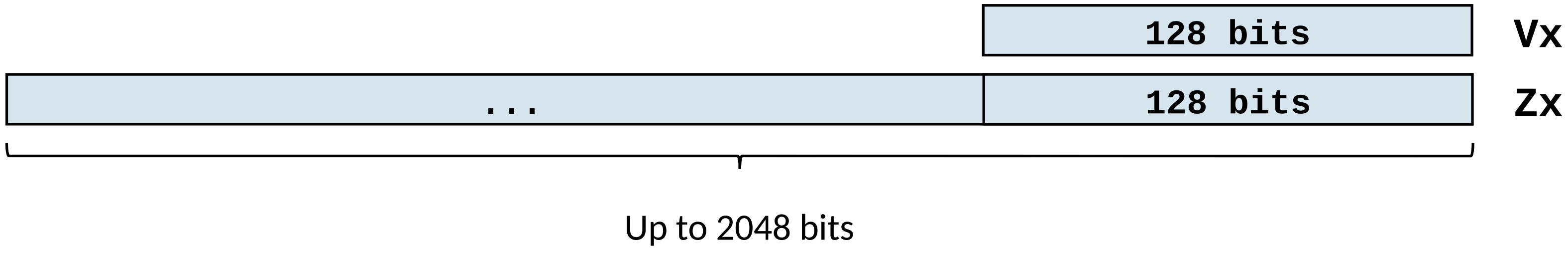
- Data types
  - Unsigned integer (U8, U16, U32, U64) and signed integer (S8, S16, S32, S64)
  - Integer of unspecified sign (I8, I16, I32, I64)
  - Floating point number (F16, F32, F64) and polynomial (P8, P16)
- Data sizes
  - A single scalar value of a floating-point or integer type (Q, D, S, H, B)
  - A 64-bit wide vector containing two or more elements (2S, 4H, 8B)
  - A 128-bit wide vector containing two or more elements (2D, 4S, 8H, 16B)
- Data type is specified as the instruction prefix; size is specified in the operands' type sizes.
  - For example, FADD V2.4S, V0.4S, V1.4S
    - Perform a floating point add of four single-precision values in registers V0 and V1, putting the result in V2
  - But not all combinations of data type size and operation are available (see the Arm Architecture Reference Manual for valid combinations).

Case Study: Scalable Vector Extension (SVE)

- SVE for Armv8-A Arch64
  - Next-generation SIMD instruction set for AArch64
- Motivated by a need for better vectorization of “real-world” applications
  - Enabling different CPUs to implement different vector lengths while sharing a common ISA
    - As a result, a program written for one CPU should work on another with no changes.
  - Add support for data-set lengths that are not a multiple of the vector width
  - Also aims to reduce the initial porting effort to use a new ISA, scalable for future designs
- A vector-length-agnostic architecture
  - Implementations can range from 128 bits up to 2048 bits.
  - Introduces new vectorization techniques

Case Study: SVE Vectors and Predicates

- 32 new scalable vector registers (**`Z0-Z31`**), length determined by the implementation
  - Bottom 128 bits overlay the floating-point & NEON vector register bank (**`V0-V31`**); top bits zeroed on a write



- 16 new scalable predicate registers (**`P0-P15);`** this example is for 256-bit vector registers.
  - Have 1/8th of a vector register’s length: 1 bit of predicate register is mapped to 1 byte of vector register

Zx	8-bit	8-bit	8-bit	8-bit	…	8-bit	8-bit	8-bit	8-bit	8-bit element
Px	1	1	1	1	…	1	1	1	1	32 bits

Zx	16-bit		-		…	16-bit		-		Unpacked 16-bit element
Px		1		0	…		1		0	32 bits

GPUs

GPU Evolution

- GPUs originally developed as specialized hardware for graphics computation
  - Now used as programmable accelerators for highly data-parallel workloads
- GPU hardware evolution closely tied to evolution in usage patterns
  - Desire for improved visual effects driven by games
  - More realism, more effects, more screen resolution, more frames per second
- Originally a fixed-function pipeline, programmability was added gradually to many stages.
  - Now the bulk of the GPU is a programmable data-parallel architecture.
  - Some fixed-function hardware remains for graphics.
  - New hardware being added to cope with modern workloads, e.g., machine learning

GPU Design Principles

- CPU design is about making a single thread run as fast as possible.
  - Pipeline stalls and memory accesses are expensive in terms of latency.
  - So increased logic was added to reduce the probability/cost of stalls.
  - Use of large cache memories to avoid memory misses
- GPU design is about maximizing computation throughput.
  - Individual thread latency not considered important
  - GPUs avoid much of the complex CPU pipeline logic for extracting ILP.
  - Instead, each thread executes on a relatively simple core with performance obtained through parallelism.
    - Single instruction, multiple threads
- Computation hides memory and pipeline latencies.
- Wide and fast bandwidth-optimized memory systems

SIMT vs SIMD

SIMT

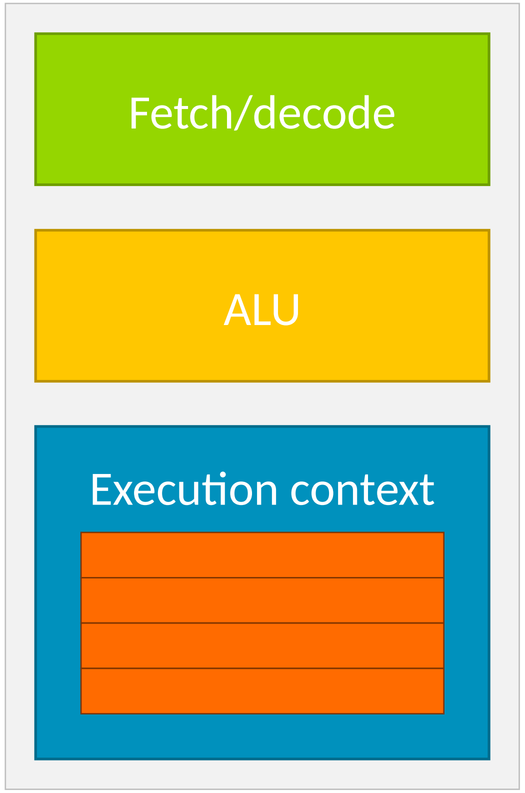
- Single instruction, multiple threads
- Takes advantage of data-level parallelism
- Can be considered a constrained form of multithreading
- Many threads each with their own state
  - Operating on scalar registers
  - With their own local memory

SIMD

- Single instruction, multiple data
- Takes advantage of data-level parallelism
- Can be considered a constrained form of vector processing
- One thread operating on vector registers

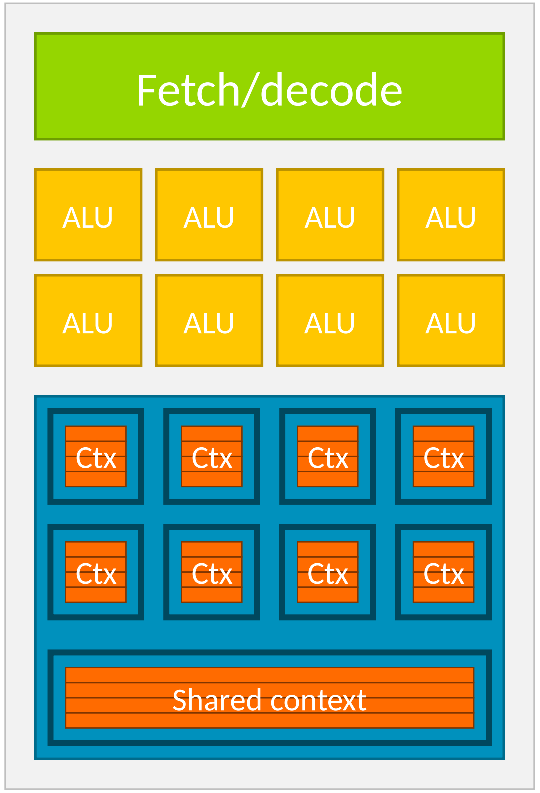
One Processing Element

- A single thread runs on a simple processing element.
- Short pipeline
- The execution context consists of the thread's state.
  - E.g., registers and local memory
- But fetch and decode are costly.



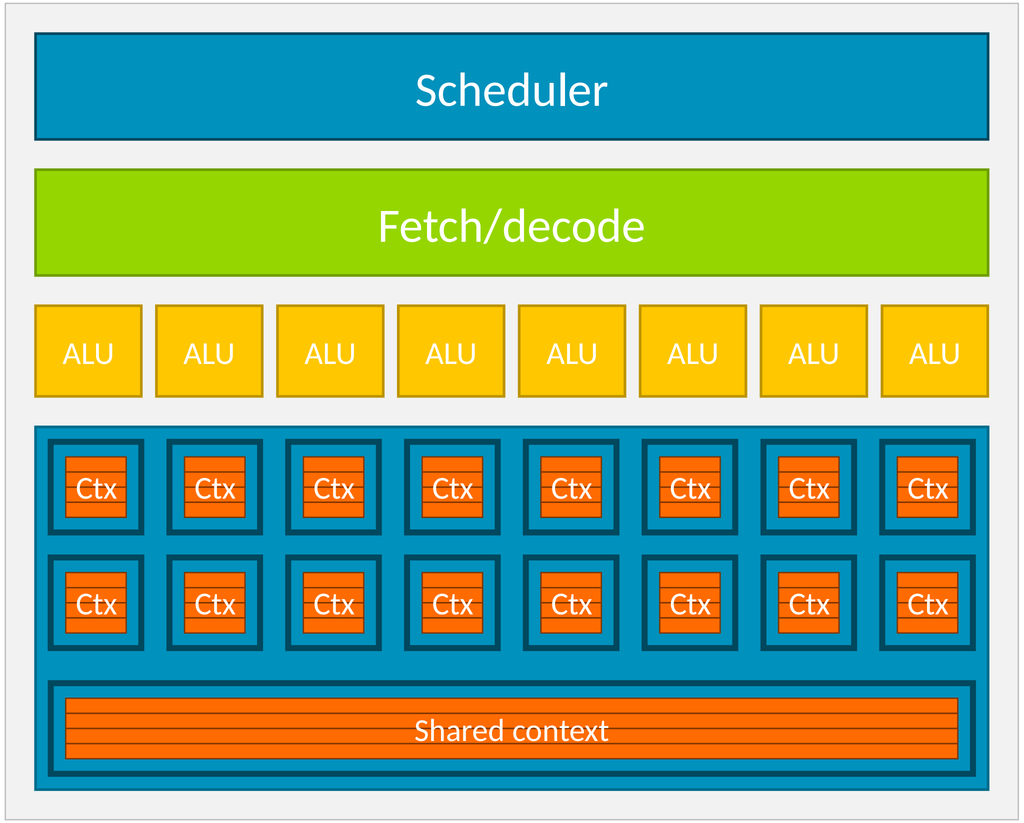
Multiple Processing Elements

- SIMT execution of threads
- Cost of fetch and decode spread across all threads in a work group (OpenCL terminology)
- Each thread has its own context.
  - And some shared context
- Multiple functional units for parallelism
  - One per thread for cheap units (e.g., simple ALU)
  - Fewer expensive units than threads (e.g., sqrt)
- However, stalls are costly.



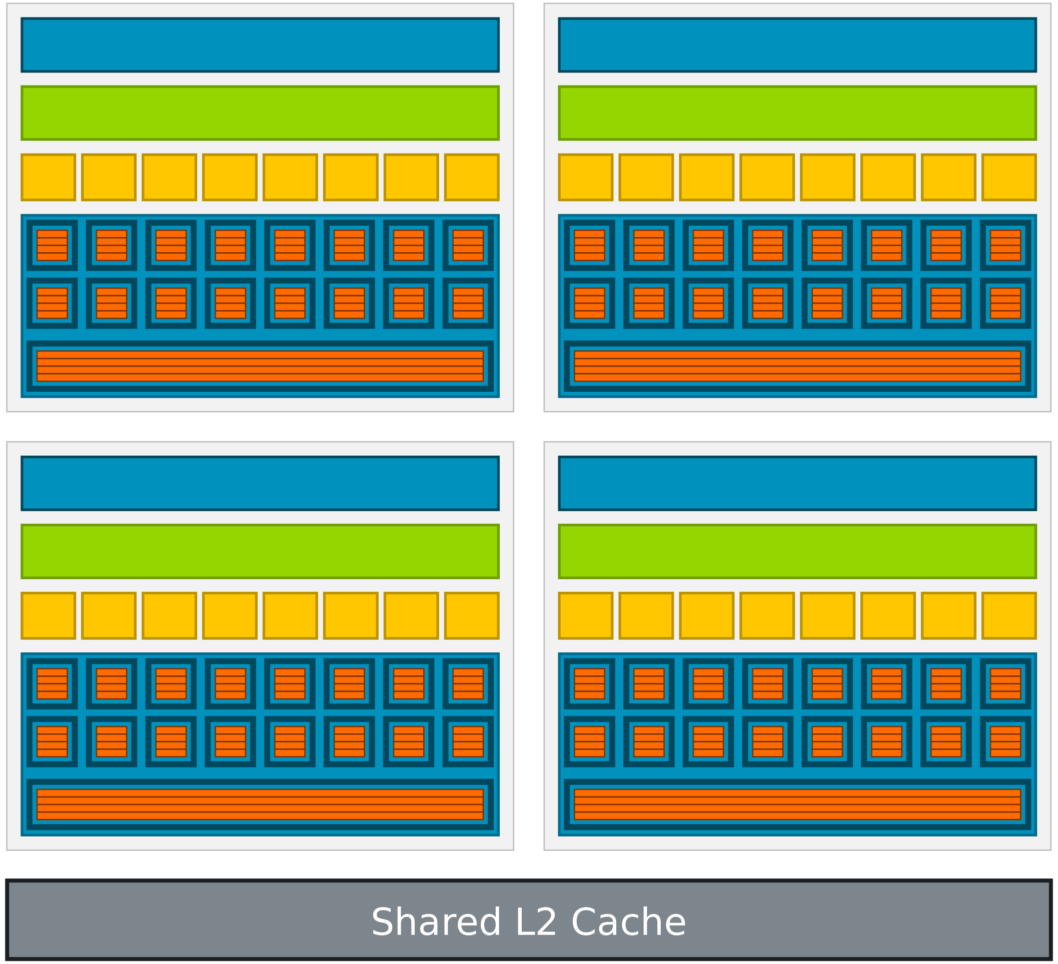
Minimizing Stalls

- Include support for multiple work groups
  - Each is independent of all others.
  - And this is guaranteed by the compiler.
  - Which means there is no fixed ordering of groups.
- A scheduler chooses work groups to run.
  - Maintains a list of ready work groups
  - Makes a choice each cycle
  - Some GPUs can schedule more than one work group per cycle.
  - Hides latency when a work group stalls
- This system is sometimes called a shader core.



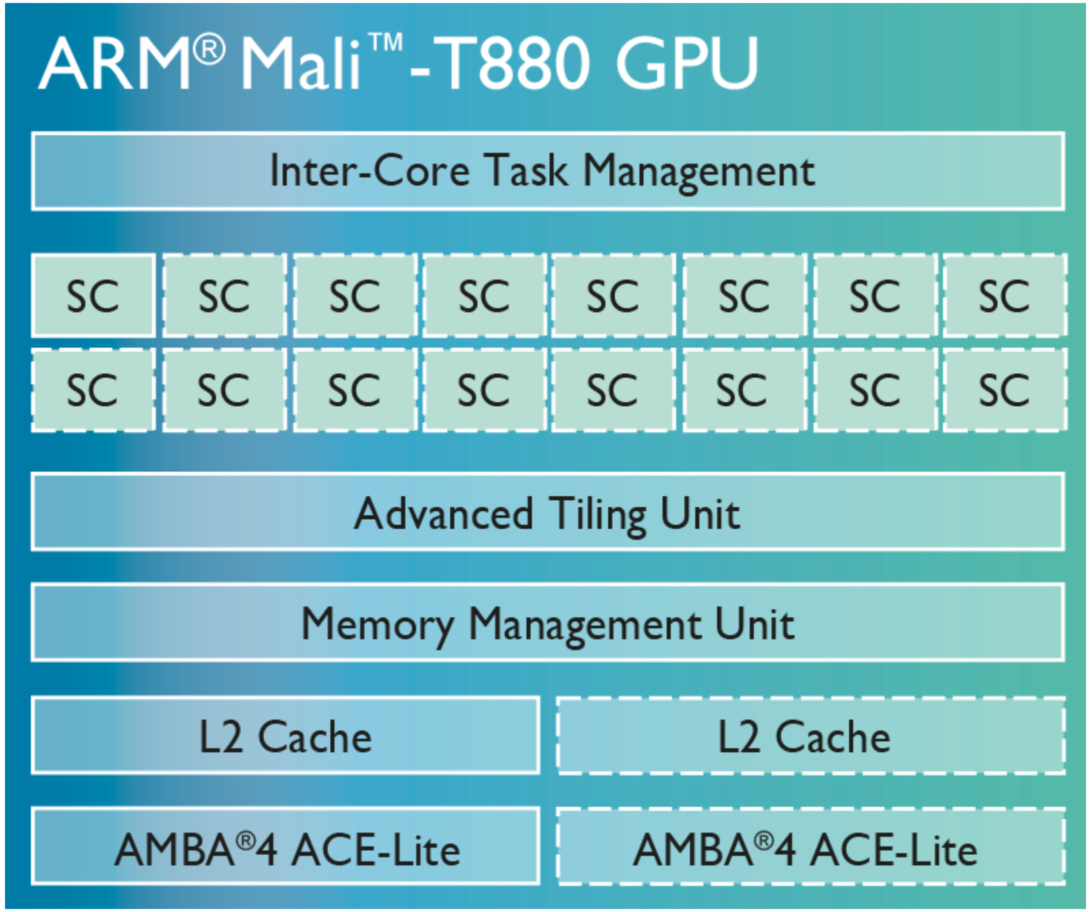
Scaling Out

- Multiple instances of each shader core provided together
  - Each independent of the others
  - Each processes a subset of the work groups
- Massively increases parallelism
- Memory hierarchy provided, too
  - Shared L2 cache reduces memory bandwidth requirements.
  - Smaller caches local to each multithreaded core



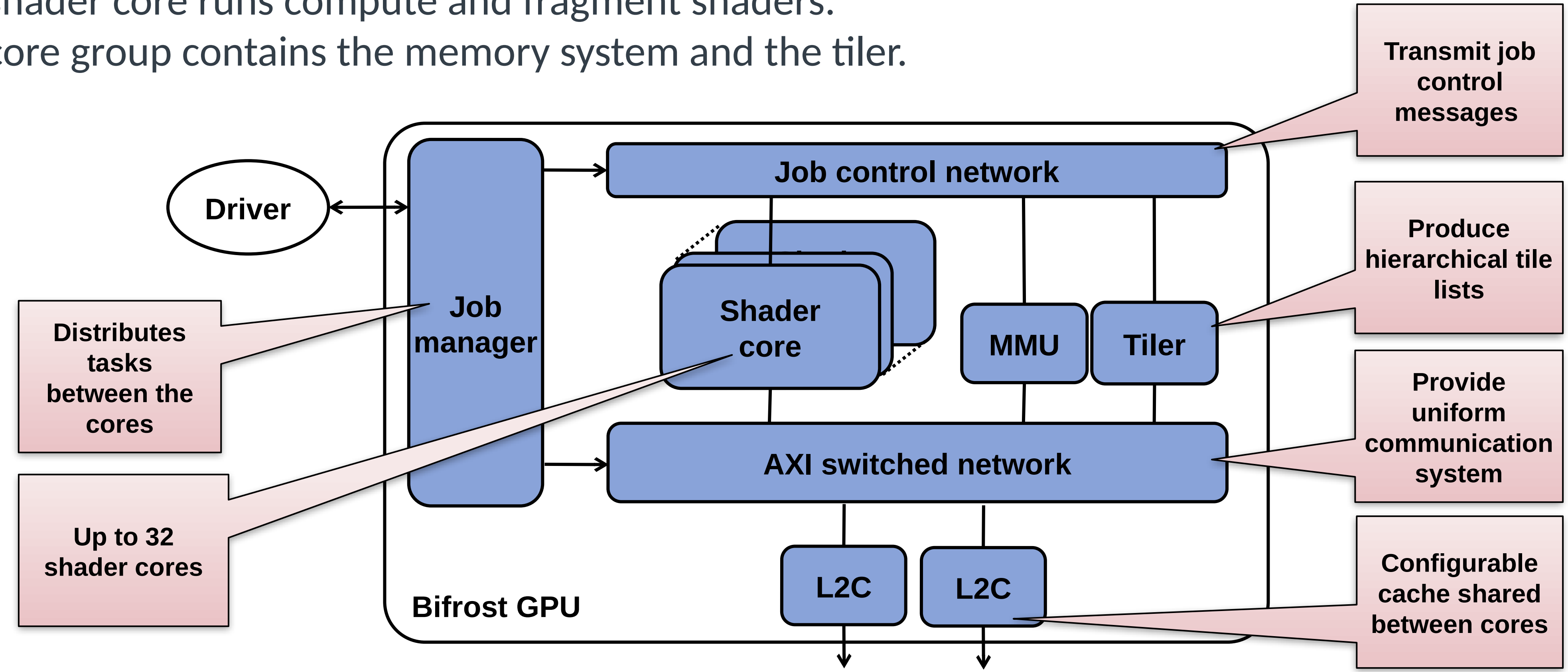
Case Study: Mali T880 GPU

- Up to 16 shader cores (SC)
  - Each core supports multiple threads and operations.
- Performance
  - 30.6G FLOPS at 900 MHz
- API support
  - OpenGL ES 1.1, 1.2, 2.0, 3.0, 3.1
    OpenCL 1.1, 1.2
    DirectX 11 FL11_2
    RenderScript
- Usage in SoCs
  - Exynos 8890, Helio X20 (MT6797), Kirin 950
- Used for both graphics processing and high-performance computing



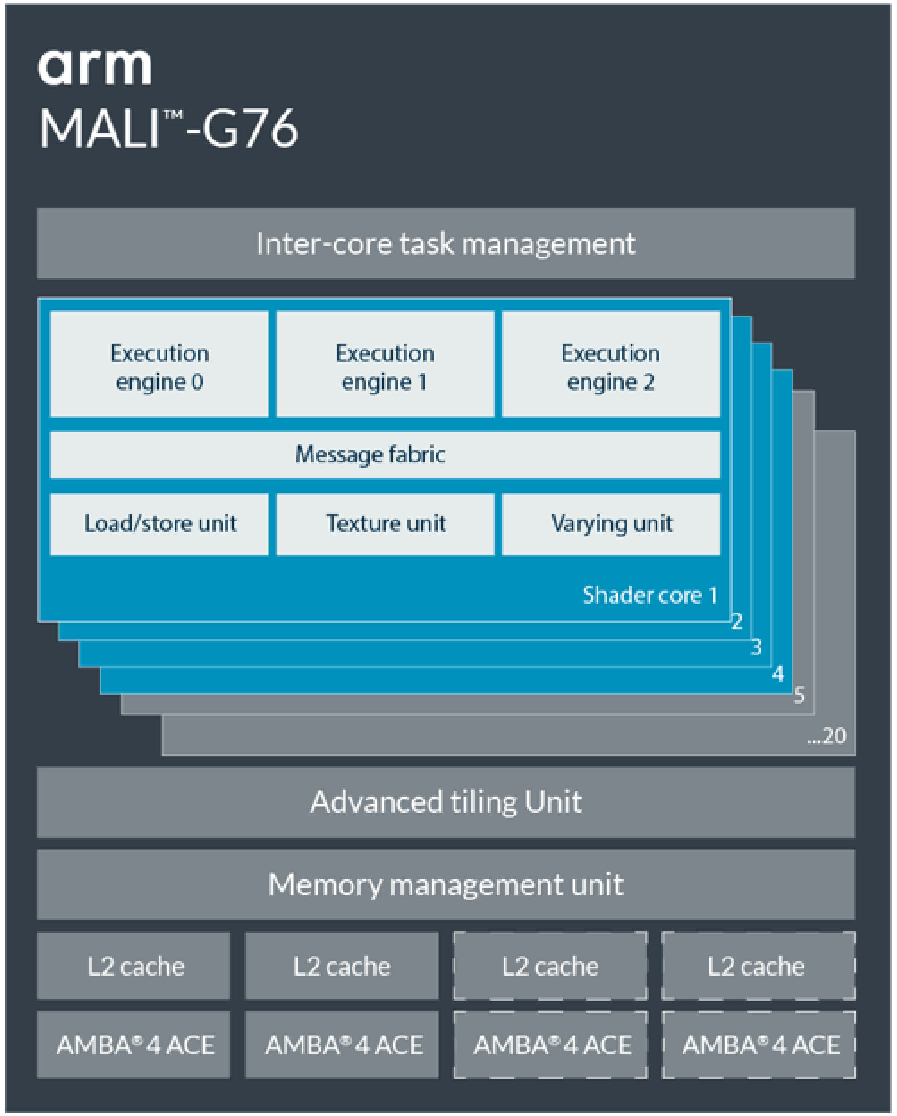
Case Study: Arm Mali Bifrost GPU Architecture

- Mali Bifrost GPU consists of 3 main blocks or groups.
  - The job manager interacts with the driver and controls the GPU HW.
  - The shader core runs compute and fragment shaders.
  - The core group contains the memory system and the tiler.



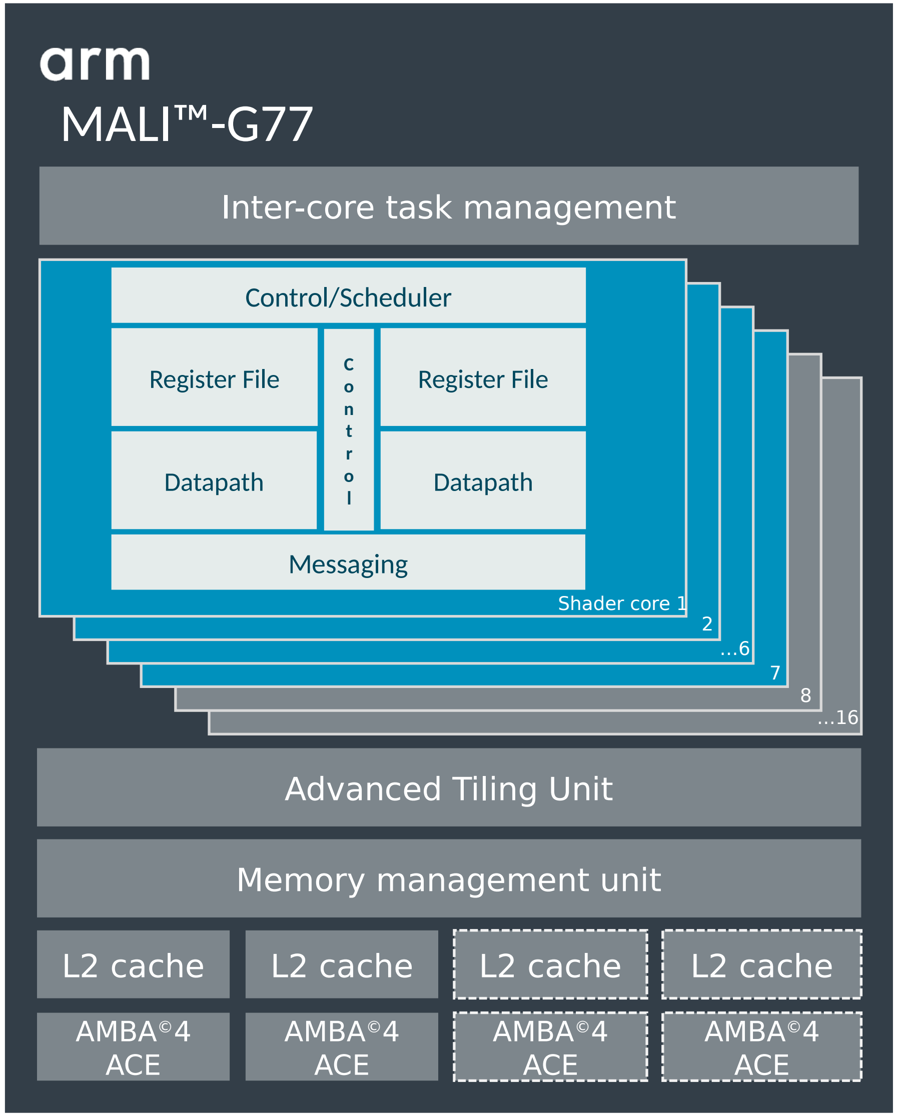
Case Study: Arm Mali-G76 GPU

- Third generation of the Bifrost architecture
- Maximum 20 shader cores (SC)
  - Wider execution engines with double the number of lanes
- Performance
  - Complex graphics and machine-learning workloads
- API support
  - OpenGL ES 1.1, 2.0, 3.1, 3.2
    OpenCL 1.1, 1.2, 2.0 Full profile
    Vulkan 1.1
- Shared L2 cache with 2 or 4 slices



Case Study: Arm Mali-G77 GPU

- First generation of the Valhall architecture
  - Warp-based execution model
  - New instruction set with operational-equivalence to Bifrost
  - Dynamic instruction scheduling in hardware
- Configurable 7 to 16 shader cores
- Single execution engine per shader core
- Configurable 2 to 4 slices of L2 cache
  - 512 KB to 4 MB in total
- Texture mapper – 4 texture element (texel) per cycle
- Supports Vulkan and OpenCL



Summary

- General-purpose CPUs optimize scalar single-threaded performance
  - But there are many domains where data-level parallelism is common.
  - Other architectures can exploit this efficiently.
- Vector processing is highly efficient for regular workloads computing on arrays of values.
  - The basis of early supercomputers
- SIMD processing brings a form of this to the general-purpose CPU.
  - Useful for multimedia codes and data-parallel operations
- GPUs exploit a different form, SIMT, with massive parallelism.
  - Simple cores but multithreading and multiprocessing hide latency