Scratch memory

Learning objectives:
▶ Understand concept of team and thread private scratch pads
▶ Understand how scratch memory can reduce global memory accesses
▶ Recognize when to use scratch memory
▶ Understand how to use scratch memory and when barriers are necessary
Two Levels of Scratch Space

- Level 0 is limited in size but fast.
- Level 1 allows larger allocations but is equivalent to High Bandwidth Memory in latency and bandwidth.

Team or Thread private memory

- Typically used for per work-item temporary storage.
- Advantage over pre-allocated memory is aggregate size scales with number of threads, not number of work-items.

Manually Managed Cache

- Explicitly cache frequently used data.
- Exposes hardware specific on-core scratch space (e.g. NVIDIA GPU Shared Memory).
Types of Scratch Space Uses

Two Levels of Scratch Space

▶ Level 0 is limited in size but fast.
▶ Level 1 allows larger allocations but is equivalent to High Bandwidth Memory in latency and bandwidth.

Team or Thread private memory

▶ Typically used for per work-item temporary storage.
▶ Advantage over pre-allocated memory is aggregate size scales with number of threads, not number of work-items.

Manually Managed Cache

▶ Explicitly cache frequently used data.
▶ Exposes hardware specific on-core scratch space (e.g. NVIDIA GPU Shared Memory).

Now: Discuss Manually Managed Cache Usecase.
One slice of contractDataFieldScalar:

```c
for (qp = 0; qp < numberOfQPs; ++qp) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
        total += A(qp, i) * B(i);
    }
    result(qp) = total;
}
```
Example: contractDataFieldScalar (2)

```c
for (element = 0; element < numberOfElements; ++element) {
    for (qp = 0; qp < numberOfQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}
```
Example: contractDataFieldScalar (3)

```cpp
for (element = 0; element < numberOfElements; ++element) {
    for (qp = 0; qp < numberOfQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}
```

**Parallelization approaches:**

- Each thread handles an element.
  
  Threads: `numberOfElements`

- Each thread handles a `qp`.
  
  Threads: `numberOfElements * numberOfQPs`

- Each thread handles an `i`.
  
  Threads: `numElements * numQPs * vectorSize`
Parallelization approaches:

- Each thread handles an element.
  Threads: $\text{numberOfElements}$

- Each thread handles a qp.
  Threads: $\text{numberOfElements} \times \text{numberOfQPs}$
Parallelization approaches:

- Each thread handles an element.
  Threads: `numberOfElements`

- Each thread handles a `qp`.
  Threads: `numberOfElements * numberOfQPs`

- Each thread handles an `i`.
  Threads: `numElements * numQPs * vectorSize`
Example: contractDataFieldScalar (3)

```cpp
for (element = 0; element < numberOfElements; ++element) {
    for (qp = 0; qp < numberOfQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}
```

**Parallelization approaches:**

- Each thread handles an element.
  
  Threads: `numberOfElements`

- Each thread handles a qp.
  
  Threads: `numberOfElements * numberOfQPs`

- Each thread handles an i.
  
  Threads: `numElements * numQPs * vectorSize`
Teams kernel: Each team handles an element

```cpp
for (element = 0; element < numberOfElements; ++element) {
    for (qp = 0; qp < numberOfQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}
```

Example: `contractDataFieldScalar (4)`
Teams kernel: Each team handles an element

```c++
operator()(member_type teamMember) {
    int element = teamMember.league_rank();
    parallel_for(
        TeamThreadRange(teamMember, numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < vectorSize; ++i) {
                total += A(element, qp, i) * B(element, i);
            }
            result(element, qp) = total;
        });
}
```

Idea: reduce global memory reads by caching B
Each team has access to a “scratch pad”.

![Diagram showing global memory, shared memory, and scratch pads for each team.]
Scratch memory (scratch pad) as manual cache:

- Accessing data in (level 0) scratch memory is (usually) **much faster** than global memory.

- **GPUs** have separate, dedicated, small, low-latency scratch memories (*NOT subject to coalescing requirements*).

- **CPUs** don’t have special hardware, but programming with scratch memory results in cache-aware memory access patterns.

- Roughly, it’s like a *user-managed* L1 cache.
Scratch memory (scratch pad) as manual cache:

- Accessing data in (level 0) scratch memory is (usually) much faster than global memory.
- **GPUs** have separate, dedicated, small, low-latency scratch memories (*NOT subject to coalescing requirements*).
- **CPUs** don’t have special hardware, but programming with scratch memory results in cache-aware memory access patterns.
- Roughly, it’s like a *user-managed* L1 cache.

**Important concept**

When members of a team read the same data multiple times, it’s better to load the data into scratch memory and read from there.
Scratch memory for temporary per work-item storage:

- Scenario: Algorithm requires temporary workspace of size $W$.
- **Without scratch memory:** pre-allocate space for $N$ work-items of size $N \times W$.
- **With scratch memory:** Kokkos pre-allocates space for each Team or Thread of size $T \times W$.
- PerThread and PerTeam scratch can be used concurrently.
- Level 0 and Level 1 scratch memory can be used concurrently.
Scratch memory for temporary per work-item storage:

- Scenario: Algorithm requires temporary workspace of size $W$.
- **Without scratch memory:** pre-allocate space for $N$ work-items of size $N \times W$.
- **With scratch memory:** Kokkos pre-allocates space for each Team or Thread of size $T \times W$.
- PerThread and PerTeam scratch can be used concurrently.
- Level 0 and Level 1 scratch memory can be used concurrently.

**Important concept**

If an algorithm requires temporary workspace for each work-item, then use Kokkos’ scratch memory.
Allocating scratch in different levels:

```java
int level = 1; // valid values 0, 1
policy.set_scratch_size(level, PerTeam(bytes));
```
Allocating scratch in different levels:

```cpp
int level = 1; // valid values 0,1
policy.set_scratch_size(level, PerTeam(bytes));
```

Using PerThread, PerTeam or both:

```cpp
policy.set_scratch_size(level, PerTeam(bytes));
policy.set_scratch_size(level, PerThread(bytes));
policy.set_scratch_size(level, PerTeam(bytes1), PerThread(bytes2));
```
Allocating scratch in different levels:

```java
int level = 1; // valid values 0,1
policy.set_scratch_size(level, PerTeam(bytes));
```

Using PerThread, PerTeam or both:

```java
policy.set_scratch_size(level, PerTeam(bytes));
policy.set_scratch_size(level, PerThread(bytes));
policy.set_scratch_size(level, PerTeam(bytes1),
                      PerThread(bytes2));
```

Using both levels of scratch:

```java
policy.set_scratch_size(0, PerTeam(bytes0))
    .set_scratch_size(1, PerThread(bytes1));
```
To use scratch memory, you need to:

1. Tell Kokkos how much scratch memory you'll need.

2. Make scratch memory views inside your kernels.

```cpp
TeamPolicy < ExecutionSpace > policy ( numberOfTeams , teamSize );
// Define a scratch memory view type
using ScratchPadView = View < double *, ExecutionSpace :: scratch_memory_space >;
// Compute how much scratch memory (in bytes) is needed
size_t bytes = ScratchPadView :: shmem_size ( vectorSize );
// Tell the policy how much scratch memory is needed
int level = 0;
parallel_for ( policy . set_scratch_size ( level , PerTeam ( bytes )) ,
    KOKKOS_LAMBDA ( const member_type & teamMember ) {
        // Create a view from the pre-existing scratch memory
        ScratchPadView scratch ( teamMember . team_scratch ( level ), vectorSize );
    });
```
To use scratch memory, you need to:

1. **Tell Kokkos how much** scratch memory you’ll need.
2. **Make** scratch memory **views** inside your kernels.

```cpp
TeamPolicy<ExecutionSpace> policy(numberOfTeams, teamSize);

// Define a scratch memory view type
using ScratchPadView = View<double*, ExecutionSpace::scratch_memory_space>;

// Compute how much scratch memory (in bytes) is needed
size_t bytes = ScratchPadView::shmem_size(vectorSize);

// Tell the policy how much scratch memory is needed
int level = 0;
parallel_for(policy.set_scratch_size(level, PerTeam(bytes)),
    KOKKOS_LAMBDA (const member_type& teamMember) {

    // Create a view from the pre-existing scratch memory
    ScratchPadView scratch(teamMember.team_scratch(level),
                            vectorSize);
    });
```
Kernel outline for teams with scratch memory:

```cpp
operator()(member_type teamMember) {
    ScratchPadView scratch(teamMember.team_scratch(0),
                           vectorSize);

    // TODO: load slice of B into scratch

    parallel_for(
        TeamThreadRange(teamMember, numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < vectorSize; ++i) {
                // total += A(element, qp, i) * B(element, i);
                total += A(element, qp, i) * scratch(i);
            }
            result(element, qp) = total;
        });
}
```
How to populate the scratch memory?

- One thread loads it all?

```c
if (teamMember.team_rank() == 0) {
    for (int i = 0; i < vectorSize; ++i) {
        scratch(i) = B(element, i);
    }
}
```
Example: contractDataFieldScalar (6)

How to populate the scratch memory?

- One thread loads it all? Serial
  
  ```c++
  if (teamMember.team_rank() == 0) {
    for (int i = 0; i < vectorSize; ++i) {
      scratch(i) = B(element, i);
    }
  }
  ```

- Each thread loads one entry?

  ```c++
  scratch(team_rank) = B(element, team_rank);
  ```
How to populate the scratch memory?

- **One thread loads it all?**  Serial

  ```
  if (teamMember.team_rank() == 0) {
    for (int i = 0; i < vectorSize; ++i) {
      scratch(i) = B(element, i);
    }
  }
  ```

- **Each thread loads one entry?**  teamSize ≠ vectorSize

  ```
  scratch(team_rank) = B(element, team_rank);
  ```

- **TeamVectorRange**

  ```
  parallel_for(
    TeamVectorRange(teamMember, vectorSize),
    [=] (int i) {
      scratch(i) = B(element, i);
    });
  ```
How to populate the scratch memory?

- **One thread loads it all?**  Serial

  ```cpp
  if (teamMember.team_rank() == 0) {
    for (int i = 0; i < vectorSize; ++i) {
      scratch(i) = B(element, i);
    }
  }
  ```

- **Each thread loads one entry?**  `teamSize ≠ vectorSize`

  ```cpp
  scratch(team_rank) = B(element, team_rank);
  ```

- **TeamVectorRange**

  ```cpp
  parallel_for(
    TeamVectorRange(teamMember, vectorSize),
    [=] (int i) {
      scratch(i) = B(element, i);
    });
  ```
(incomplete) Kernel for teams with scratch memory:

```
operator()(member_type teamMember) {
    ScratchPadView scratch(...);

    parallel_for(TeamVectorRange(teamMember, vectorSize),
        [=] (int i) {
            scratch(i) = B(element, i);
        });

    // TODO: fix a problem at this location

    parallel_for(TeamThreadRange(teamMember, numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < vectorSize; ++i) {
                total += A(element, qp, i) * scratch(i);
            }
            result(element, qp) = total;
        });
}
```
(incomplete) Kernel for teams with scratch memory:

```
operator()(member_type teamMember) {
    ScratchPadView scratch(...);

    parallel_for(TeamVectorRange(teamMember, vectorSize),
                [=] (int i) {
                    scratch(i) = B(element, i);
                });
    // TODO: fix a problem at this location

    parallel_for(TeamThreadRange(teamMember, numberOfQPs),
                [=] (int qp) {
                    double total = 0;
                    for (int i = 0; i < vectorSize; ++i) {
                        total += A(element, qp, i) * scratch(i);
                    }
                    result(element, qp) = total;
                });
}
```

Problem: threads may start to use `scratch` before all threads are done loading.
Kernel for teams with scratch memory:

operator()(member_type teamMember) {
    ScratchPadView scratch(...);

    parallel_for(ThreadVectorRange(teamMember, vectorSize),
        [=] (int i) {
            scratch(i) = B(element, i);
        });
    teamMember.team_barrier();

    parallel_for(TeamThreadRange(teamMember, numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < vectorSize; ++i) {
                total += A(element, qp, i) * scratch(i);
            }
            result(element, qp) = total;
        });
}
Use Scratch Memory to explicitly cache the x-vector for each element.

**Details:**
- Location: Exercises/team_scratch_memory/
- Create a scratch view
- Fill the scratch view in parallel using a TeamVectorRange

**Things to try:**
- Vary problem size and number of rows (-S ...; -N ...)
- Compare behavior with Exercises/team_vector_loop/
- Compare behavior of CPU vs GPU
Exercise 07 (Scratch Memory) Fixed Size

KNL: Xeon Phi 68c  HSW: Dual Xeon Haswell 2x16c  Pascal60: Nvidia GPU

Bandwidth (GB/s) vs Number of Rows (N)

06 HSW
07 HSW
06 KNL
07 KNL
06 Pascal60
07 Pascal60
Scratch Memory can be used with the TeamPolicy to provide thread or team private memory.

- Usecase: per work-item temporary storage or manual caching.
- Scratch memory exposes on-chip user managed caches (e.g. on NVIDIA GPUs)
- The size must be determined before launching a kernel.
- Two levels are available: small/fast (level 0) and large/slow (level 1).