Module 4: Hierarchical Parallelism

April 26, 2024
Module 4: Hierarchical Parallelism

Hierarchical Parallelism
▶ How to leverage more parallelism through nested loops.
▶ The concept of Thread-Teams and Vectorlength.

Scratch Space
▶ Getting temporary workspace in kernels.
▶ Leveraging GPU Shared Memory.

Unique Token
▶ How to acquire safely per-thread resources.
Hierarchical parallelism

Finding and exploiting more parallelism in your computations.

Learning objectives:

▶ Similarities and differences between outer and inner levels of parallelism
▶ Thread teams (league of teams of threads)
▶ Performance improvement with well-coordinated teams
(Flat parallel) Kernel:

Kokkos::parallel_reduce("yAx", N,
    KOKKOS_LAMBDa (const int row, double & valueToUpdate) {
        double thisRowsSum = 0;
        for (int col = 0; col < M; ++col) {
            thisRowsSum += A(row, col) * x(col);
        }
        valueToUpdate += y(row) * thisRowsSum;
    }, result);
(Flat parallel) Kernel:

Kokkos::parallel_reduce("yAx",N,
    KOKKOS_LAMBDA (const int row, double & valueToUpdate) {
        double thisRowsSum = 0;
        for (int col = 0; col < M; ++col) {
            thisRowsSum += A(row,col) * x(col);
        }
        valueToUpdate += y(row) * thisRowsSum;
    }, result);

Problem: What if we don’t have enough rows to saturate the GPU?
**Example: inner product (0)**

**Kernel:**
```
Kokkos::parallel_reduce("yAx",N,
    KOKKOS_LAMBDA (const int row, double & valueToUpdate) {
        double thisRowsSum = 0;
        for (int col = 0; col < M; ++col) {
            thisRowsSum += A(row, col) * x(col);
        }
        valueToUpdate += y(row) * thisRowsSum;
    }, result);
```

**Problem:** What if we don’t have enough rows to saturate the GPU?

**Solutions?**
(Flat parallel) Kernel:

Kokkos::parallel_reduce("yAx",N,
    KOKKOS_LAMBDA (const int row, double & valueToUpdate) {
        double thisRowsSum = 0;
        for (int col = 0; col < M; ++col) {
            thisRowsSum += A(row,col) * x(col);
        }
        valueToUpdate += y(row) * thisRowsSum;
    }, result);

Problem: What if we don’t have enough rows to saturate the GPU?

Solutions?
- Atomics
- Thread teams
Atomics kernel:

Kokkos::parallel_for("yAx", N*M,  
KOKKOS_LAMBDA (const size_t index) {  
  const int row = extractRow(index);  
  const int col = extractCol(index);  
  atomic_add(&result, y(row) * A(row,col) * x(col));  
});
**Example: inner product (1)**

**Atomics kernel:**

```cpp
Kokkos::parallel_for("yAx", N*M,
    KOKKOS_LAMBDA (const size_t index) {
        const int row = extractRow(index);
        const int col = extractCol(index);
        atomic_add(&result, y(row) * A(row,col) * x(col));
    });
```

**Problem:** Poor performance
Using an atomic with every element is doing scalar integration with atomics. (See module 3)

Instead, you could envision doing a large number of `parallel_reduce` kernels.

```cpp
for each row
    Functor functor(row, ...);
    parallel_reduce(M, functor);
```
Using an atomic with every element is doing scalar integration with atomics. (See module 3)

Instead, you could envision doing a large number of parallel_reduce kernels.

```cpp
for each row
    Functor functor(row, ...);
    parallel_reduce(M, functor);
}
```

This is an example of hierarchical work.

**Important concept: Hierarchical parallelism**

Algorithms that exhibit hierarchical structure can exploit hierarchical parallelism with thread teams.
Important concept: Thread team

A collection of threads which are guaranteed to be executing *concurrently* and *can synchronize*.
Important concept: Thread team

A collection of threads which are guaranteed to be executing **concurrently** and can synchronize.

High-level **strategy**:

1. Do one parallel launch of $N$ teams.
2. Each team handles a row.
3. The threads within teams perform a reduction.
4. The thread teams perform a reduction.
The final hierarchical parallel kernel:

```cpp
parallel_reduce("yAx",
    team_policy(N, Kokkos::AUTO),

    KOKKOS_LAMBDA (const member_type & teamMember, double & update) {
        int row = teamMember.league_rank();
        double thisRowsSum = 0;
        parallel_reduce(TeamThreadRange(teamMember, M),
            [=] (int col, double & innerUpdate) {
                innerUpdate += A(row, col) * x(col);
            }, thisRowsSum);

        if (teamMember.team_rank() == 0) {
            update += y(row) * thisRowsSum;
        }
    }, result);
```
Important point

Using teams is changing the execution *policy*.

“**Flat** parallelism” uses RangePolicy:

We specify a *total amount of work*.

```cpp
// total work = N
parallel_for("Label",
    RangePolicy<ExecutionSpace>(0,N), functor);
```
Important point

Using teams is changing the execution policy.

“**Flat** parallelism” uses RangePolicy:

We specify a *total amount of work*.

```
// total work = N
parallel_for("Label",
    RangePolicy<ExecutionSpace>(0,N), functor);
```

“**Hierarchical** parallelism” uses TeamPolicy:

We specify a *team size* and a *number of teams*.

```
// total work = numberOfTeams * teamSize
parallel_for("Label",
    TeamPolicy<ExecutionSpace>(numberOfTeams, teamSize), functor);
```
Important point

When using teams, functor operators receive a *team member*.

```cpp
using member_type = typename TeamPolicy<ExecSpace>::member_type;

void operator()(const member_type & teamMember) {
    // How many teams are there?
    const unsigned int league_size = teamMember.league_size();

    // Which team am I on?
    const unsigned int league_rank = teamMember.league_rank();

    // How many threads are in the team?
    const unsigned int team_size = teamMember.team_size();

    // Which thread am I on this team?
    const unsigned int team_rank = teamMember.team_rank();

    // Make threads in a team wait on each other:
    teamMember.team_barrier();
}
```
First attempt at exercise:

```cpp
operator() (member_type & teamMember) {
    const size_t row = teamMember.league_rank();
    const size_t col = teamMember.team_rank();
    atomic_add(&result, y(row) * A(row, col) * x(entry));
}
```
First attempt at exercise:

```cpp
operator() (member_type & teamMember ) {
    const size_t row = teamMember.league_rank();
    const size_t col = teamMember.team_rank();
    atomic_add(&result, y(row) * A(row, col) * x(entry));
}
```

- When team size \(\neq\) number of columns, how are units of work mapped to team’s member threads? Is the mapping architecture-dependent?
Second attempt at exercise:

Divide row length among team members.

```cpp
operator() (member_type & teamMember ) {
    const size_t row = teamMember.league_rank();

    int begin = teamMember.team_rank();
    for(int col = begin; col < M; col += teamMember.team_size()) {
        atomic_add(&result, y(row) * A(row,col) * x(entry));
    }
}
```

Still bad because `atomic add` performs badly under high contention, how can team's member threads performantly cooperate for a nested reduction?

On CPUs you get a bad data access pattern: this hardcodes coalesced access, but not caching.
Second attempt at exercise:

Divide row length among team members.

```cpp
operator() (member_type & teamMember ) {
  const size_t row = teamMember.league_rank();

  int begin = teamMember.team_rank();
  for(int col = begin; col < M; col += teamMember.team_size()) {
    atomic_add(&result, y(row) * A(row,col) * x(entry));
  }
}
```

- Still bad because `atomic_add` performs badly under high contention, how can team’s member threads performantly cooperate for a nested reduction?
- On CPUs you get a bad data access pattern: this hardcodes coalesced access, but not caching.
We shouldn’t be hard-coding the work mapping...

```cpp
operator() (member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum = 0;
    // ‘do a reduction’(‘‘over M columns’’,
     [=] (const int col) {
        thisRowsSum += A(row,col) * x(col);
    });
    if (teamMember.team_rank() == 0) {
        update += (row) * thisRowsSum;
    }
}
```

If this were a parallel execution, we’d use Kokkos::parallel reduce.

Key idea: this is a parallel execution.
⇒ Nested parallel patterns
We shouldn’t be hard-coding the work mapping...

```cpp
operator() (member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    "do a reduction"("over M columns",
    [=] (const int col) {
        thisRowsSum += A(row, col) * x(col);
    });
    if (teamMember.team_rank() == 0) {
        update += (row) * thisRowsSum;
    }
}
```

If this were a parallel execution, we’d use Kokkos::parallel_reduce.
We shouldn’t be hard-coding the work mapping...

```cpp
operator() (member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    "do a reduction"("over M columns",
        [=] (const int col) {
            thisRowsSum += A(row, col) * x(col);
        });
    if (teamMember.team_rank() == 0) {
        update += (row) * thisRowsSum;
    }
}
```

If this were a parallel execution, we’d use Kokkos::parallel_reduce.

**Key idea:** this is a parallel execution.
We shouldn’t be hard-coding the work mapping...

```cpp
operator() (member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    // 'do a reduction' ('over M columns'),
    [=] (const int col) {
        thisRowsSum += A(row, col) * x(col);
    });
    if (teamMember.team_rank() == 0) {
        update += (row) * thisRowsSum;
    }
}
```

If this were a parallel execution,
we’d use Kokkos::parallel_reduce.

**Key idea:** this is a parallel execution.

⇒ **Nested parallel patterns**
TeamThreadRange:

```cpp
operator() (const member_type & teamMember, double & update ) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    parallel_reduce(TeamThreadRange(teamMember, M),
                    [=] (const int col, double & thisRowsPartialSum ) {
                        thisRowsPartialSum += A(row, col) * x(col);
                    }, thisRowsSum);
    if (teamMember.team_rank() == 0) {
        update += y(row) * thisRowsSum;
    }
}
```

The mapping of work indices to threads is architecture-dependent.

The amount of work given to the TeamThreadRange need not be a multiple of the team size.

Intrateam reduction handled by Kokkos.
TeamThreadRange:

```cpp
operator() (const member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    parallel_reduce(TeamThreadRange(teamMember, M),
                    [=] (const int col, double & thisRowsPartialSum) {
                        thisRowsPartialSum += A(row, col) * x(col);
                    }, thisRowsSum);
    if (teamMember.team_rank() == 0) {
        update += y(row) * thisRowsSum;
    }
}
```

- The **mapping** of work indices to threads is architecture-dependent.
- The **amount of work** given to the TeamThreadRange **need not be a multiple** of the team size.
- Intrateam **reduction handled** by Kokkos.
**Anatomy** of nested parallelism:

```cpp
parallel_outer("Label",
    TeamPolicy<ExecutionSpace>(numberOfTeams, teamSize),
    KOKKOS_LAMBDA (const member_type & teamMember[, ...]) {
    
    /* beginning of outer body */

    parallel_inner(
        TeamThreadRange(teamMember, thisTeamsRangeSize),
        [=] (const unsigned int indexWithinBatch[, ...]) {
        /* inner body */
        }[, ...]);
    /* end of outer body */
    }[, ...]);
```

- `parallel_outer` and `parallel_inner` may be any combination of `for` and/or `reduce`.
- The inner lambda may capture by reference, but capture-by-value is recommended.
- The policy of the inner lambda is always a `TeamThreadRange`.
- `TeamThreadRange` cannot be nested.
In practice, you can let Kokkos decide:

```cpp
parallel_something(
    TeamPolicy<ExecutionSpace>(numberOfTeams, Kokkos::AUTO),
    /* functor */);
```
What should the team size be?

In practice, you can let Kokkos decide:

```
parallel_something(
    TeamPolicy<ExecutionSpace>(numberOfTeams, Kokkos::AUTO),
    /* functor */);
```

**GPUs**

- Special hardware available for coordination within a team.
- Within a team 32 (NVIDIA) or 64 (AMD) threads execute “lock step.”
- Maximum team size: **1024**; Recommended team size: **128/256**
What should the team size be?

In practice, you can let Kokkos decide:

```
parallel_something(
    TeamPolicy<ExecutionSpace>(numberOfTeams, Kokkos::AUTO),
    /* functor */);
```

**GPUs**

- Special hardware available for coordination within a team.
- Within a team 32 (NVIDIA) or 64 (AMD) threads execute “lock step.”
- Maximum team size: **1024**; Recommended team size: **128/256**

**Intel Xeon Phi:**

- Recommended team size: `#` hyperthreads per core
- Hyperthreads share entire cache hierarchy
  a well-coordinated team avoids cache-thrashing
Exercise: TeamPolicy

Details:

▶ Location: Exercises/team_policy/
▶ Replace RangePolicy<Space> with TeamPolicy<Space>
▶ Use AUTO for team_size
▶ Make the inner loop a parallel_reduce with TeamThreadRange policy
▶ Experiment with the combinations of Layout, Space, N to view performance
▶ Hint: what should the layout of A be?

Things to try:

▶ Vary problem size and number of rows (-S ...; -N ...)
▶ Compare behavior with Exercise 4 for very non-square matrices
▶ Compare behavior of CPU vs GPU
Exercise 04 (Layout) Fixed Size

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

**Legend:**
- HSW Left
- HSW Right
- KNL Left
- KNL Right
- Pascal60 Left
- Pascal60 Right

**Graph:**
- x-axis: Number of Rows (N)
- y-axis: Bandwidth (GB/s)

**Legend:**
- Coalesced
- Cached
- Uncached
<\text{y|Ax}> Exercise 05 (Layout/Teams) Fixed Size

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

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

- **coalesced**
- **cached**

- HSW Left
- HSW Right
- KNL Left
- KNL Right
- Pascal60 Left
- Pascal60 Right
Exposing Vector Level Parallelism

- Optional **third level** in the hierarchy: ThreadVectorRange
  - Can be used for `parallel_for`, `parallel_reduce`, or `parallel_scan`.

- Maps to vectorizable loop on CPUs or (sub-)warp level parallelism on GPUs.

- Enabled with a **runtime** vector length argument to `TeamPolicy`

- There is **no** explicit access to a vector lane ID.

- Depending on the backend the full global parallel region has active vector lanes.

- TeamVectorRange uses both **thread** and **vector** parallelism.
Three-level parallelism (1)

Anatomy of nested parallelism:

```cpp
parallel_outer("Label",
TeamPolicy<>(numberOfTeams, teamSize, vectorLength),
KOKKOS_LAMBDA (const member_type & teamMember[, ...]) {
    /* beginning of outer body */

    parallel_middle(
        TeamThreadRange(teamMember, thisTeamsRangeSize),
        [=] (const int indexWithinBatch[, ...]) {
            /* begin middle body */

            parallel_inner(
                ThreadVectorRange(teamMember, thisVectorRangeSize),
                [=] (const int indexVectorRange[, ...]) {
                    /* inner body */
                }[, ...]);
            /* end middle body */
        }[, ...]);
    /* end of outer body */

    parallel_middle(
        TeamVectorRange(teamMember, someSize),
        [=] (const int indexTeamVector[, ...]) {
            /* nested body */
        }[, ...]);
    /* end of outer body */
}[, ...]);
```
Question: What will the value of `totalSum` be?

```cpp
int totalSum = 0;
parallel_reduce("Sum", RangePolicy<>\((0, numberOfThreads),
    KOKKOS_LAMBDA (size_t& index, int& partialSum) {
        int thisThreadsSum = 0;
        for (int i = 0; i < 10; ++i) {
            ++thisThreadsSum;
        }
        partialSum += thisThreadsSum;
    }, totalSum);
```

```cpp
totalSum = numberOfThreads * 10
```
Question: What will the value of totalSum be?

```cpp
int totalSum = 0;
parallel_reduce("Sum", RangePolicy<>(0, numberOfThreads),
    KOKKOS_LAMBDA (size_t & index, int & partialSum) {
        int thisThreadsSum = 0;
        for (int i = 0; i < 10; ++i) {
            ++thisThreadsSum;
        }
        partialSum += thisThreadsSum;
    }, totalSum);

totalSum = numberOfThreads * 10
```
**Question:** What will the value of `totalSum` be?

```c++
int totalSum = 0;
parallel_reduce("Sum", TeamPolicy<>)(numberOfTeams, team_size),
  KOKKOS_LAMBDA (member_type& teamMember, int& partialSum) {
    int thisThreadsSum = 0;
    for (int i = 0; i < 10; ++i) {
      ++thisThreadsSum;
    }
    partialSum += thisThreadsSum;
  }, totalSum);
```

The value of `totalSum` will be `numberOfTeams * team_size * 10`. This is because each thread calculates the sum of 10 elements and then adds it to the `partialSum`. Since there are `numberOfTeams` threads, each of which performs the same operation, the final value of `totalSum` is the sum of the products of `team_size` and 10, for each team.
**Question:** What will the value of `totalSum` be?

```c
int totalSum = 0;
parallel_reduce("Sum", TeamPolicy<>)(numberOfTeams, team_size),
    KOKKOS_LAMBDA (member_type& teamMember, int& partialSum) {
        int thisThreadsSum = 0;
        for (int i = 0; i < 10; ++i) {
            ++thisThreadsSum;
        }
        partialSum += thisThreadsSum;
    }, totalSum);

totalSum = numberOfTeams * team_size * 10
```
**Question:** What will the value of totalSum be?

```cpp
int totalSum = 0;
parallel_reduce("Sum", TeamPolicy<>(numberOfTeams, team_size),
  KOKKOS_LAMBDA (member_type& teamMember, int& partialSum) {
    int thisTeamsSum = 0;
    parallel_reduce(TeamThreadRange(teamMember, team_size),
      [=] (const int index, int& thisTeamsPartialSum) {
        int thisThreadsSum = 0;
        for (int i = 0; i < 10; ++i) {
          ++thisThreadsSum;
        }
        thisTeamsPartialSum += thisThreadsSum;
      }, thisTeamsSum);
    thisTeamsPartialSum += thisTeamsSum;
  }, totalSum);
```

```cpp
totalSum = numberOf Teams * team_size * team_size * 10
```
**Question:** What will the value of `totalSum` be?

```c++
int totalSum = 0;
parallel_reduce("Sum", TeamPolicy<>)(numberOfTeams, team_size),
    KOKKOS_LAMBDA (member_type& teamMember, int& partialSum) {
        int thisTeamsSum = 0;
        parallel_reduce(TeamThreadRange(teamMember, team_size),
            [=] (const int index, int& thisTeamsPartialSum) {
                int thisThreadsSum = 0;
                for (int i = 0; i < 10; ++i) {
                    ++thisThreadsSum;
                }
                thisTeamsPartialSum += thisThreadsSum;
            }, thisTeamsSum);
        thisTeamsPartialSum += thisTeamsSum;
        partialSum += thisTeamsSum;
    }, totalSum);

    totalSum = numberOfTeams * team_size * team_size * 10
```
The **single** pattern can be used to restrict execution

- Like parallel patterns it takes a policy, a lambda, and optionally a broadcast argument.
- Two policies: PerTeam and PerThread.
- Equivalent to OpenMP **single** directive with **nowait**

```c
// Restrict to once per thread
single(PerThread(teamMember), [&] () {
    // code
});

// Restrict to once per team with broadcast
int broadcastedValue = 0;
single(PerTeam(teamMember), [&] (int& broadcastedValue_local) {
    broadcastedValue_local = special value assigned by one;
}, broadcastedValue);
// Now everyone has the special value
```
The previous example was extended with an outer loop over “Elements” to expose a third natural layer of parallelism.

Details:

- Location: Exercises/team_vector_loop/
- Use the single policy instead of checking team rank
- Parallelize all three loop levels.

Things to try:

- Vary problem size and number of rows (-S ...; -N ...)
- Compare behavior with TeamPolicy Exercise for very non-square matrices
- Compare behavior of CPU vs GPU
Exercise: TeamVectorLoop

<y|Ax> Exercise 06 (Three Level Parallelism) Fixed Size

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

Bandwidth (GB/s)

Number of Rows (N)
Hierarchical work can be parallelized via hierarchical parallelism.

Hierarchical parallelism is leveraged using thread teams launched with a TeamPolicy.

Team “worksets” are processed by a team in nested parallel for (or reduce or scan) calls with a TeamThreadRange, ThreadVectorRange, and TeamVectorRange policy.

Execution can be restricted to a subset of the team with the single pattern using either a PerTeam or PerThread policy.
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).
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 Use Case.
```plaintext
for (qp = 0; qp < numberOfQPs; ++qp) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
        total += A(qp, i) * B(i);
    }
    result(qp) = total;
}
```
contractDataFieldScalar:

```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;
    }
}
```
Parallelization approaches:

- Each thread handles an element.
  Threads: numberOfElements

```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`
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`

Requires a parallel_reduce.
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`

Requires a parallel _reduce_.

```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;
    }
}
```
Flat kernel: Each thread handles a quadrature point

```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;
    }
}
```

```cpp
parallel_for("L",MDRangePolicy<Rank<2>>({0,0},{numE,numQP}),
    KOKKOS_LAMBDA(int element, int qp) {
        double total = 0;
        for (int i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
```

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;
    }
}
```

```cpp
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;
        });
}
```
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;
    }
}
```

No real advantage (yet)
Each team has access to a “scratch pad”.

![Diagram of scratch memory]
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.
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.
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 (8)`
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?**

  ```
  scratch(team_rank) = B(element, team_rank);
  ```
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);
    });
```
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? teamSize ≠ vectorSize

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

- TeamVectorRange

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

```cpp
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:

```cpp
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:

```cpp
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 Exercise 6
- Compare behavior of CPU vs GPU
Exercise 07 (Scratch Memory) Fixed Size

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

![Graph showing bandwidth (GB/s) vs number of rows (N). The graph compares different configurations: KNL, HSW, and Pascal60, with data points for 2004 and 2007.]
Allocating scratch in different levels:

```c
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:

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

Using PerThread, PerTeam or both:

```c
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:

```c
policy.set_scratch_size(0, PerTeam(bytes0))
.set_scratch_size(1, PerThread(bytes1));
```
- **Scratch Memory** can be used with the TeamPolicy to provide thread or team *private* memory.
- Use case: 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: large/slow and small/fast.
Unique Token

Learning objectives:

▶ Understand concept of unique tokens and thread-safe resource access.

▶ Learn how to acquire per-team unique ids.

▶ Understand the difference between **Global** and **Instance** scope.
Why do we need a unique token concept?

▶ Within Functor operator / Lambda there is no portable way to identify the active execution resource (thread id)
▶ Some algorithms make efficient use of shared resources by dividing based on execution resource (thread id)
▶ Thread Id is not consistent or portable across all execution environments
▶ Unique Token provides consistent identifier for resource allocations and work division
**Original Example: Random Number Generator Pool**

```csharp
int N = 10000000
int K = ...;
RandomGenPool pool(K, seed);
parallel_for("Loop", N, KOKKOS_LAMBDA(int i) {
    int gen_id = ... 
    auto gen = pool[gen_id]; 
});
```

**How many generators do we need (K)?**
Original Example: Random Number Generator Pool

```c++
int N = 10000000
int K = ...;
RandomGenPool pool(K, seed);
parallel_for("Loop", N, KOKKOS_LAMBDA(int i) {
    int gen_id = ...
    auto gen = pool[gen_id];
});
```

How many generators do we need (K)?
How to get a unique one in the loop (gen_id)?
Original Example: Random Number Generator Pool

```c
int N = 10000000
int K = ...;
RandomGenPool pool(K,seed);
parallel_for("Loop", N, KOKKOS_LAMBDA(int i) {
    int gen_id = ... 
    auto gen = pool[gen_id];
});
```

How many generators do we need (K)?
How to get a unique one in the loop (gen_id)?
In OpenMP we could use the `thread-id` but what in CUDA?
Motivating Example

OpenMP

```c
int K = omp_get_max_threads();
Kokkos::parallel_for("L", N, KOKKOS_LAMBDA(int i) {
  int tid = omp_get_thread_num();
});
```

CUDA

```c
int K = N; // ??
Kokkos::parallel_for("L", N, KOKKOS_LAMBDA(int i) {
  int tid = threadIdx.x + blockDim.x * blockIdx.x; // i??
});
```
Motivating Example

OpenMP

```c
int K = omp_get_max_threads();
Kokkos::parallel_for("L", N, KOKKOS_LAMBDA(int i) {
    int tid = omp_get_thread_num();
});
```

CUDA

```c
int K = N; // ??
Kokkos::parallel_for("L", N, KOKKOS_LAMBDA(int i) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x; //i??
});
```

Problem: In CUDA there is no way to get hardware thread-id.
Motivating Example

OpenMP

```c
int K = omp_get_max_threads();
Kokkos::parallel_for("L", N, KOKKOS_LAMBDA(int i) {
    int tid = omp_get_thread_num();
});
```

CUDA

```c
int K = N; // ??
Kokkos::parallel_for("L", N, KOKKOS_LAMBDA(int i) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x; //i??
});
```

**Problem**: In CUDA there is no way to get hardware thread-id.

**Solution**: We need a thread-safe and portable way to obtain unique identifier that is per-thread specific.

⇒ UniqueToken
UniqueToken is a pool of IDs

- User acquires an ID and releases it again.

```cpp
UniqueToken<ExecutionSpace> token;
int number_of_unique_ids = token.size();
RandomGenPool pool(number_of_unique_ids, seed);
parallel_for("L", N, KOKKOS_LAMBDA(int i) {
    int id = token.acquire();
    RandomGen gen = pool(id);
    ...
    token.release(id);
});
```
UniqueToken is a pool of IDs

- User acquires an ID and releases it again.

```cpp
UniqueToken<ExecutionSpace> token;
int number_of_unique_ids = token.size();
RandomGenPool pool(number_of_unique_ids, seed);
parallel_for("L", N, KOKKOS_LAMBDA(int i) {
    int id = token.acquire();
    RandomGen gen = pool(id);
    ...
    token.release(id);
});
```

- Do not acquire more than one token in an iteration.
- You must release the token again.
- By default the range of ids is 0 to `ExecSpace().concurrency()`. 
Sometimes you need a **Global UniqueToken**

- Submitting concurrent kernels to CUDA streams
- Shared resource in a multi-threaded environment like Legion
Sometimes you need a **Global UniqueToken**

- Submitting concurrent kernels to CUDA streams
- Shared resource in a multi-threaded environment like Legion

**UniqueToken is Scoped**

UniqueToken has a Scope template parameter which by default is 'Instance' but can be 'Global'.
Sometimes you need a **Global UniqueToken**

- Submitting concurrent kernels to CUDA streams
- Shared resource in a multi-threaded environment like Legion

**UniqueToken isScoped**

UniqueToken has a Scope template parameter which by default is 'Instance' but can be 'Global'.

```cpp
void foo() {
    UniqueToken<ExecSpace, UniqueTokenScope::Global> token_foo;
    parallel_for("L", RangePolicy<ExecSpace>(stream1, 0, N),
             functor_a(token_foo));
}
void bar() {
    UniqueToken<ExecSpace, UniqueTokenScope::Global> token_bar;
    parallel_for("L", RangePolicy<ExecSpace>(stream2, 0, N),
             functor_b(token_bar));
}
```
Sometimes you need a **Global UniqueToken**

- Submitting concurrent kernels to CUDA streams
- Shared resource in a multi-threaded environment like Legion

**UniqueToken is Scoped**

UniqueToken has a Scope template parameter which by default is 'Instance' but can be 'Global'.

```cpp
void foo() {
    UniqueToken<ExecSpace, UniqueTokenType::Global> token_foo;
    parallel_for("L", RangePolicy<ExecSpace>(stream1, 0, N),
                functor_a(token_foo));
}
void bar() {
    UniqueToken<ExecSpace, UniqueTokenType::Global> token_bar;
    parallel_for("L", RangePolicy<ExecSpace>(stream2, 0, N),
                functor_b(token_bar));
}
```

token_foo and token_bar will provide non-conflicting ids.
UniqueToken can also be used for Per-Team resources

There are less teams active than threads. How to get an ID?
**UniqueToken can also be used for Per-Team resources**

There are less teams active than threads. How to get an ID?

---

**Sized UniqueToken**

*UniqueToken supports custom ranges of ids via constructing sized tokens.*
**UniqueToken** can also be used for Per-Team resources

There are less teams active than threads. How to get an ID?

**Sized UniqueToken**

UniqueToken supports custom ranges of ids via constructing sized tokens.

**Acquiring a per-team unique id requires three steps:**

- Compute the range via concurrency and team size.
- Create a sized UniqueToken.
  - For performance reason make it a bit larger than necessary.
- Acquire and broadcast a token in a single pattern.
/ Figure out the team size
int team_size = ...;
/ How many teams are actually in-flight
int num_active_teams = ExecSpace().concurrency()/team_size;
/ Create the token
UniqueToken<ExecSpace> token(num_active_teams * 1.2);

parallel_for("L", TeamPolicy<ExecSpace>(N,team_size),
KOKKOS_LAMBDA(const team_t& team) {
    int id;
    / Acquire an id and broadcast it with a single thread
    single(PerTeam(team),[&](int &lid) {
        lid = token.acquire();
    },id);
    ...
    / Release the id again (likely you want a barrier first!)
    single(PerTeam(team),[&]() {
        token.release(id);
    });
Exercise UniqueToken

- Location: Exercises/unique_token/Begin/
- Assignment: Convert scatter_add_loop to use UniqueToken, removing #ifdef’s
- Compile and run on both CPU and GPU

```make
make -j KOKKOS_DEVICES=OpenMP  # CPU-only using OpenMP
make -j KOKKOS_DEVICES=Cuda   # GPU - note UVM in Makefile
# Run exercise
./uniquetoken.host
./uniquetoken.cud
# Note the warnings, set appropriate environment variables
```

- Compare performance on CPU of the three variants
- Compare performance on GPU of the two variants
- Vary problem size: first and second optional argument
- **UniqueToken** provides a thread safe portable way to divide thread or team specific resources
- **UniqueToken** can be sized, such that it returns only ids within a specific range.
- A **Global** scope UniqueToken can be acquired, allowing safe ids across disjoint concurrent code sections.
Hierarchical Parallelism

- **Hierarchical work** can be parallelized via hierarchical parallelism.
- Hierarchical parallelism is leveraged using **thread teams** launched with a TeamPolicy.
- Team “worksets” are processed by a team in nested parallel for (or reduce or scan) calls with a TeamThreadRange and ThreadVectorRange policy.
- Execution can be restricted to a subset of the team with the single pattern using either a PerTeam or PerThread policy.
- Teams can be used to **reduce contention** for global resources even in “flat” algorithms.
Scratch Space

- **Scratch Memory** can be used with the TeamPolicy to provide thread or team private memory.
- Use case: 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: large/slow and small/fast.

Unique Token

- **UniqueToken** give a thread safe portable way to divide thread specific resources
- **UniqueToken** can be sized to restrict ids to a range.
- A Global UniqueToken is available.