OAK RIDGE National Laboratory

Kokkos Tutorial: Advanced Topics

Damien Lebrun-Grandié

ORNL is managed by UT-Battelle LLC
for the US Department of Energy

U.S. DEPARTMENT OF ENERGY

- **Hierarchical Parallelism**
- **Tools: Profiling and Debugging**

Hierarchical parallelism

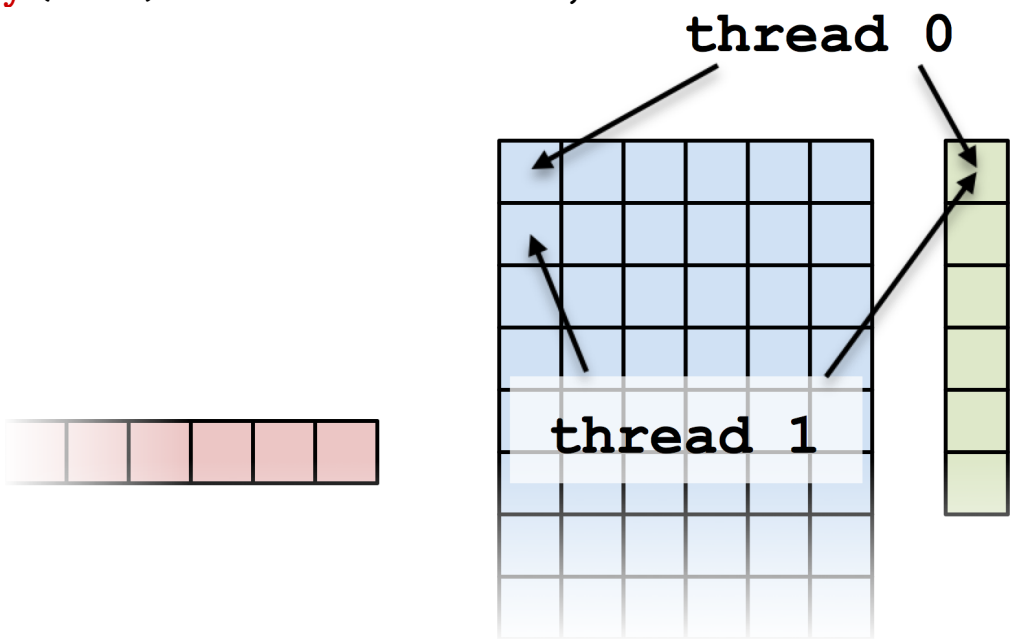
Finding and exploiting more parallelism in your computations.

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:

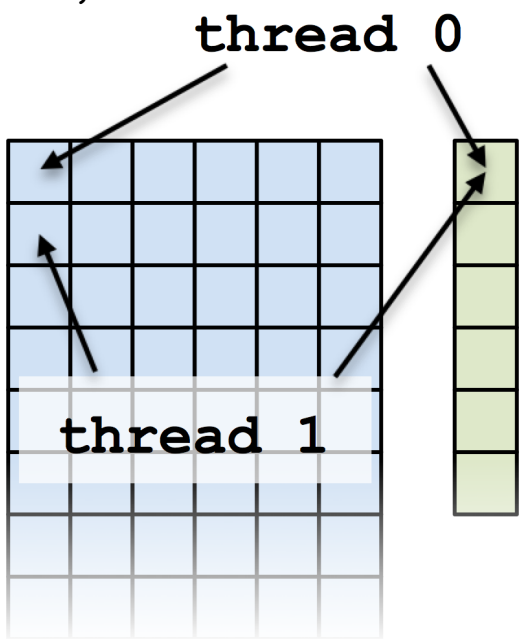
```
Kokkos::parallel_reduce("yAx",N,
  KOKKOS_LAMBDA (const int row, double & valueToUpdate) {
    double thisRowsSum = 0;
    for (int col = 0; col < M; ++col) {
      thisRowsSum += A(row,col) * x(col);
    }
    valueToUpdate += y(row) * thisRowsSum;
  }, result);
```



(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?

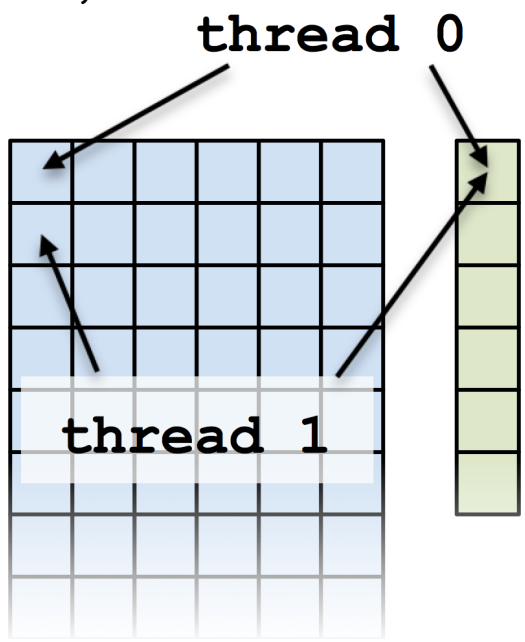


(Flat parallel) Kernel:

```
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);
```

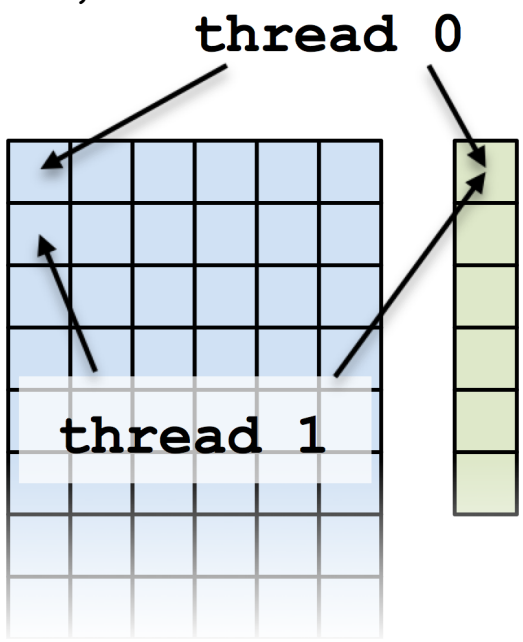
Problem: What if we don't have enough rows to saturate the GPU?

Solutions?



(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);
```



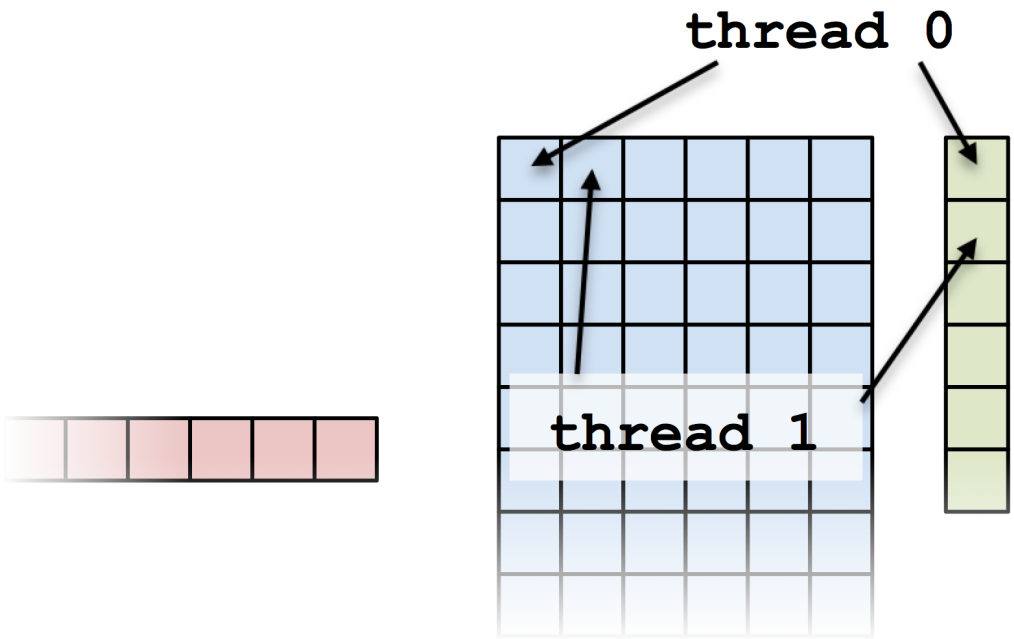
Problem: What if we don't have enough rows to saturate the GPU?

Solutions?

- Atomics
- Thread teams

Atomics kernel:

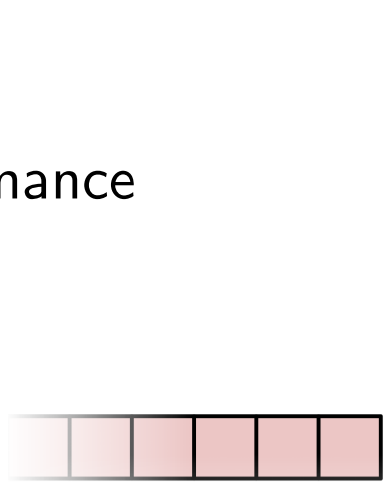
```
Kokkos::parallel_for("yAx", N*M,
  KOKKOS_LAMBDA (const size_t index) {
    const int row = extractRow(index);
    const int col = extractCol(index);
    atomic_add(&result, y(row) * A(row,col) * x(col));
  });
```

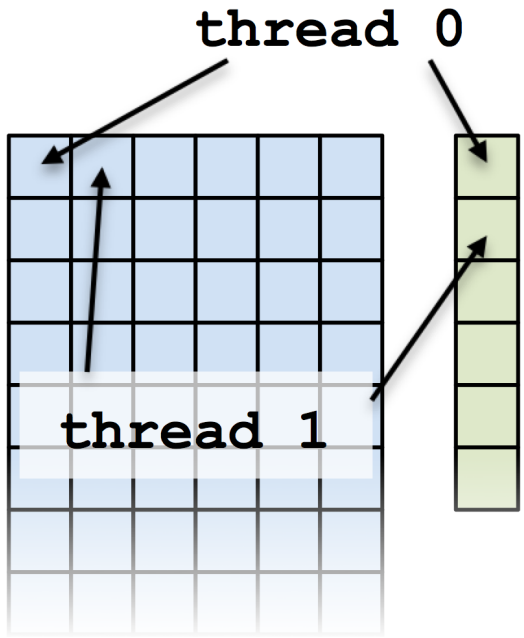


Atomics kernel:

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

Problem: Poor performance





Example: inner product (2)

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.

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

Example: inner product (2)

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.

```
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

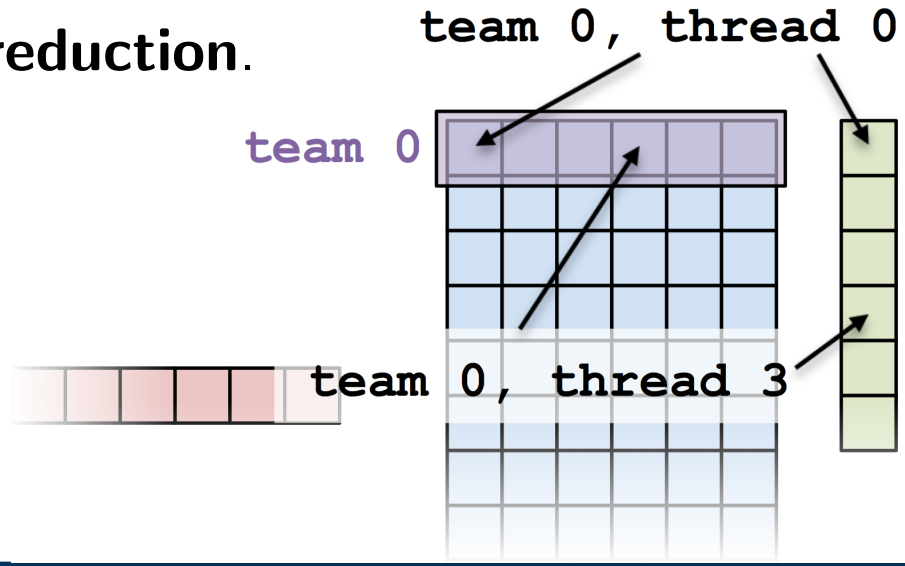
A collection of threads which are guaranteed to be executing **concurrently** and **can synchronize**.

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:

```
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*.

```
// 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*.

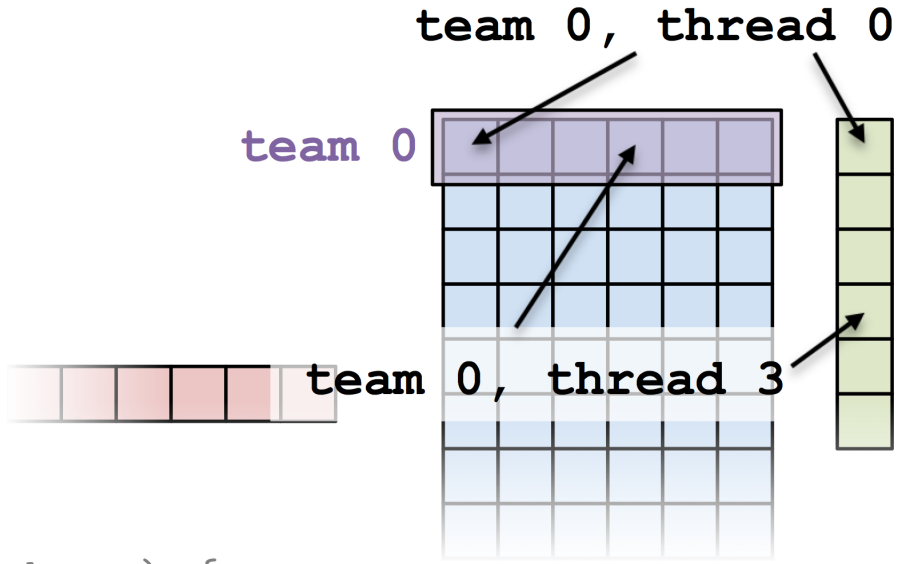
```
// total work = numberOfTeams * teamSize
parallel_for("Label",
  TeamPolicy<ExecutionSpace>(numberOfTeams, teamSize), functor);
```

Important point

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

```
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();
}
```

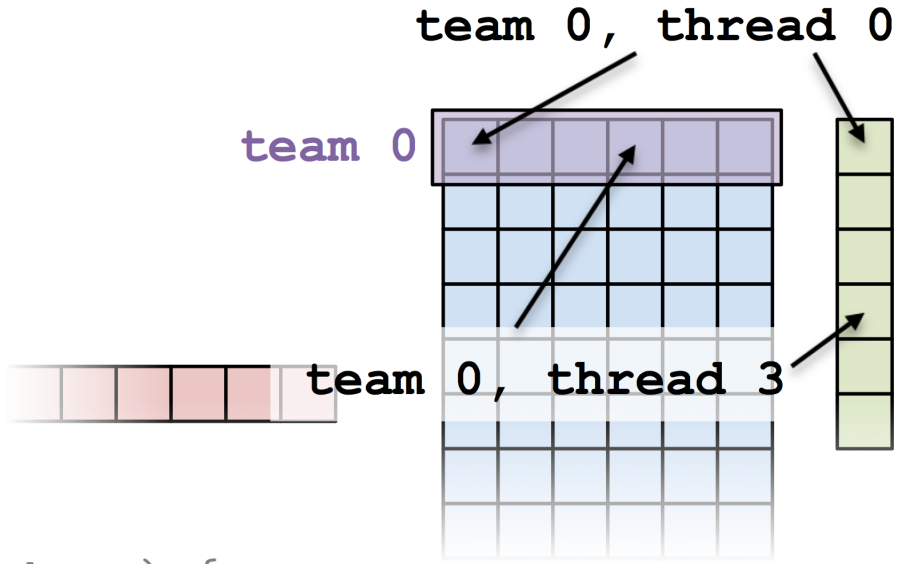
TeamThreadRange (0)



First attempt at exercise:

```
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));
}
```

TeamThreadRange (0)



First attempt at exercise:

```
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.

```
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));
  }
}
```

Second attempt at exercise:

Divide row length among team members.

```
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...

```
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;
  }
}
```

We shouldn't be hard-coding the work mapping...

```
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...

```
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...

```
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.

$\Rightarrow$ **Nested parallel patterns**

TeamThreadRange:

```
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;
  }
}
```

TeamThreadRange:

```
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:

```
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**:

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

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**

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

Details:

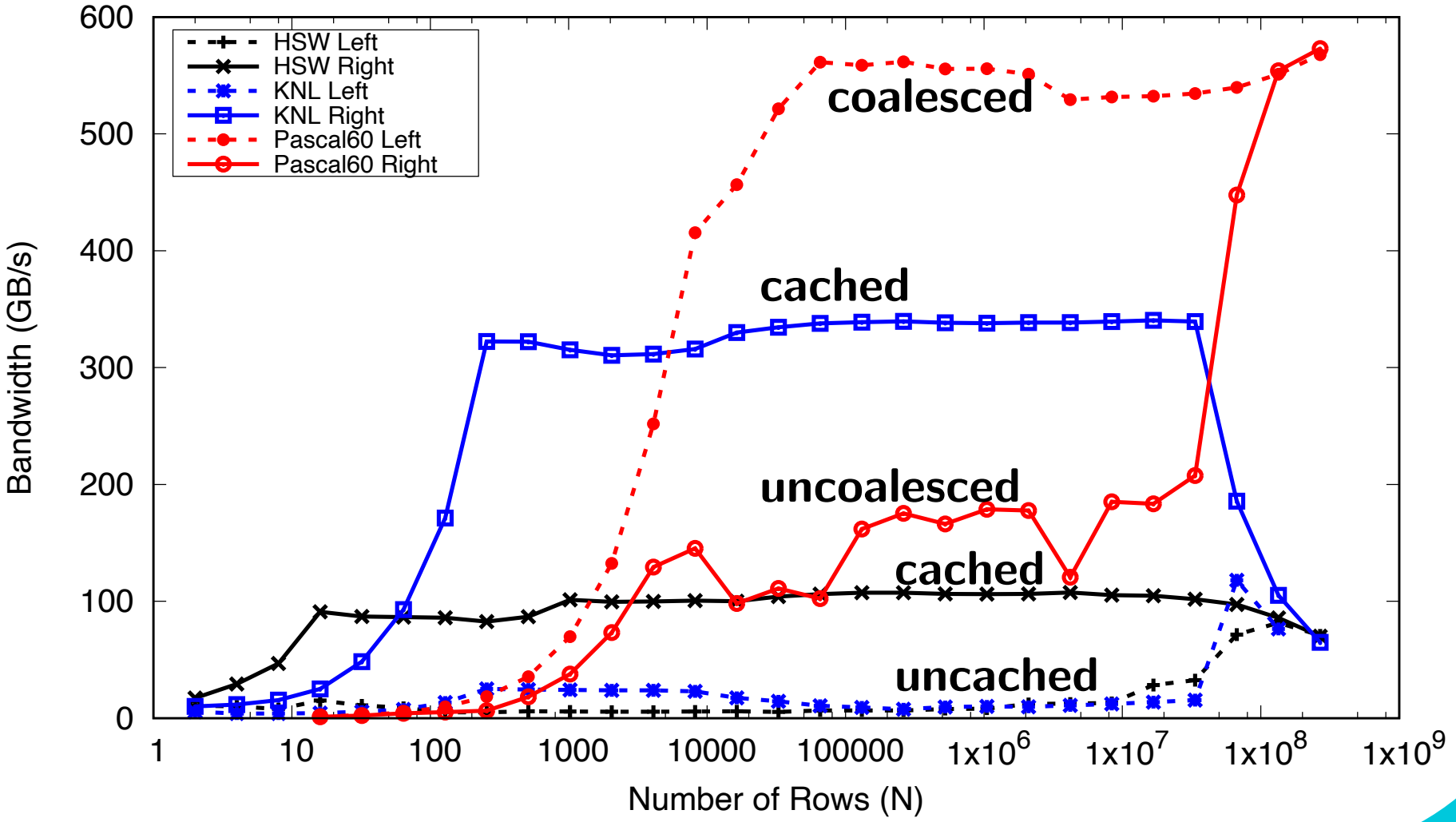
- Location: `Exercises/team_policy/`
- Replace `RangePolicy<Space>` with `TeamPolicy<Space>`
- Use `AUTO` for `team_size`
- Make the inner loop a `parallel_reduce` with `TeamThreadRange` policy
- Experiment with the combinations of `Layout`, `Space`, `N` to view performance
- Hint: what should the layout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

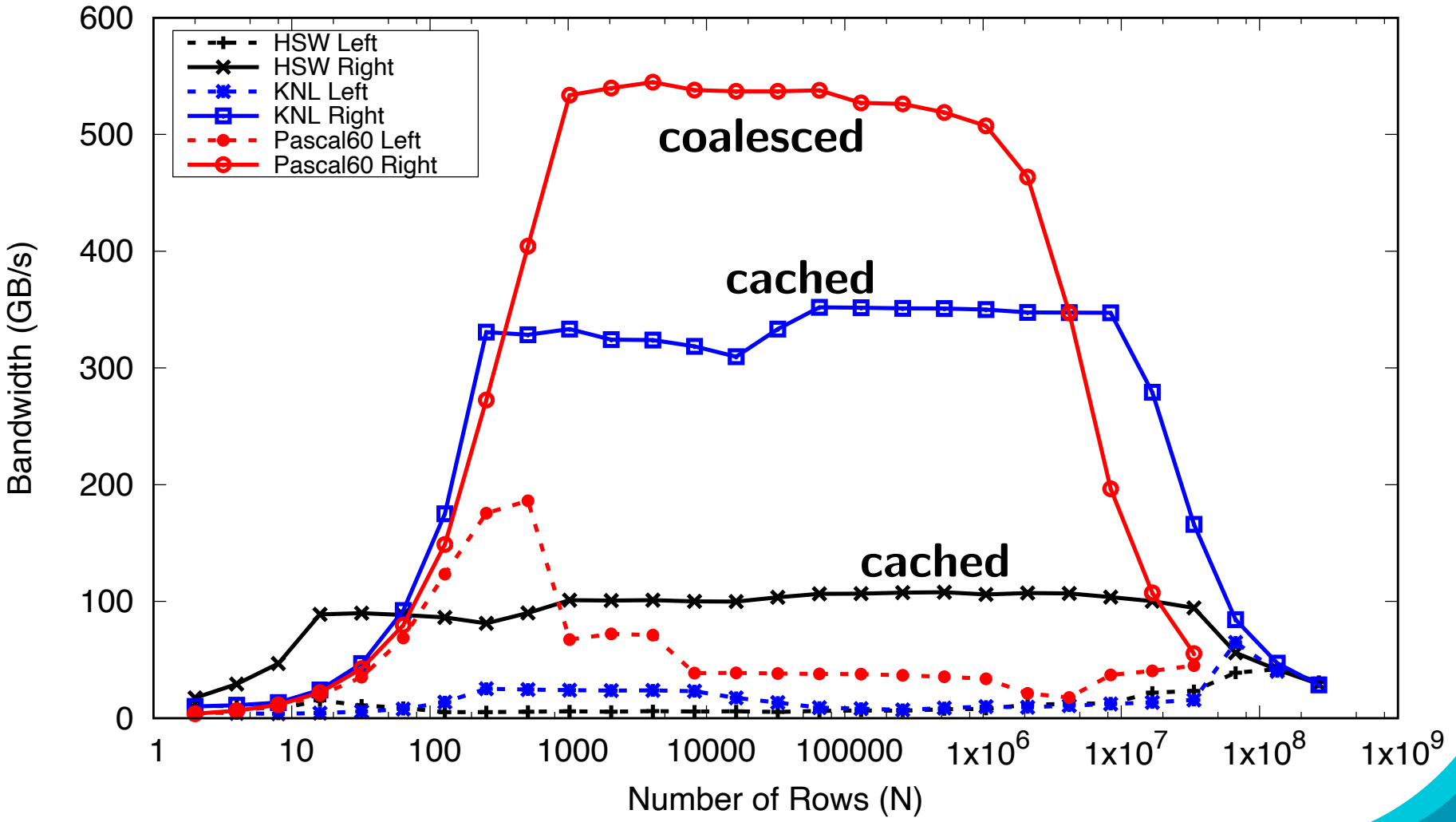
<ylAx> Exercise 04 (Layout) Fixed Size

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



<ylAx> Exercise 05 (Layout/Teams) Fixed Size

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



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.

Anatomy of nested parallelism:

```
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 */
      }[, ...]);
    parallel_middle(
    TeamVectorRange(teamMember, someSize),
      [=] (const int indexTeamVector[, ...]) {
        /* nested body */
      }[, ...]);
    /* end of outer body */
  }[, ...]);
```

Question: What will the value of `totalSum` be?

```
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);
```

Question: What will the value of `totalSum` be?

```
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?

```
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);
```

Question: What will the value of `totalSum` be?

```
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?

```
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);
    partialSum += thisTeamsSum;
}, totalSum);
```

Question: What will the value of `totalSum` be?

```
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);
    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**

```
// 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:

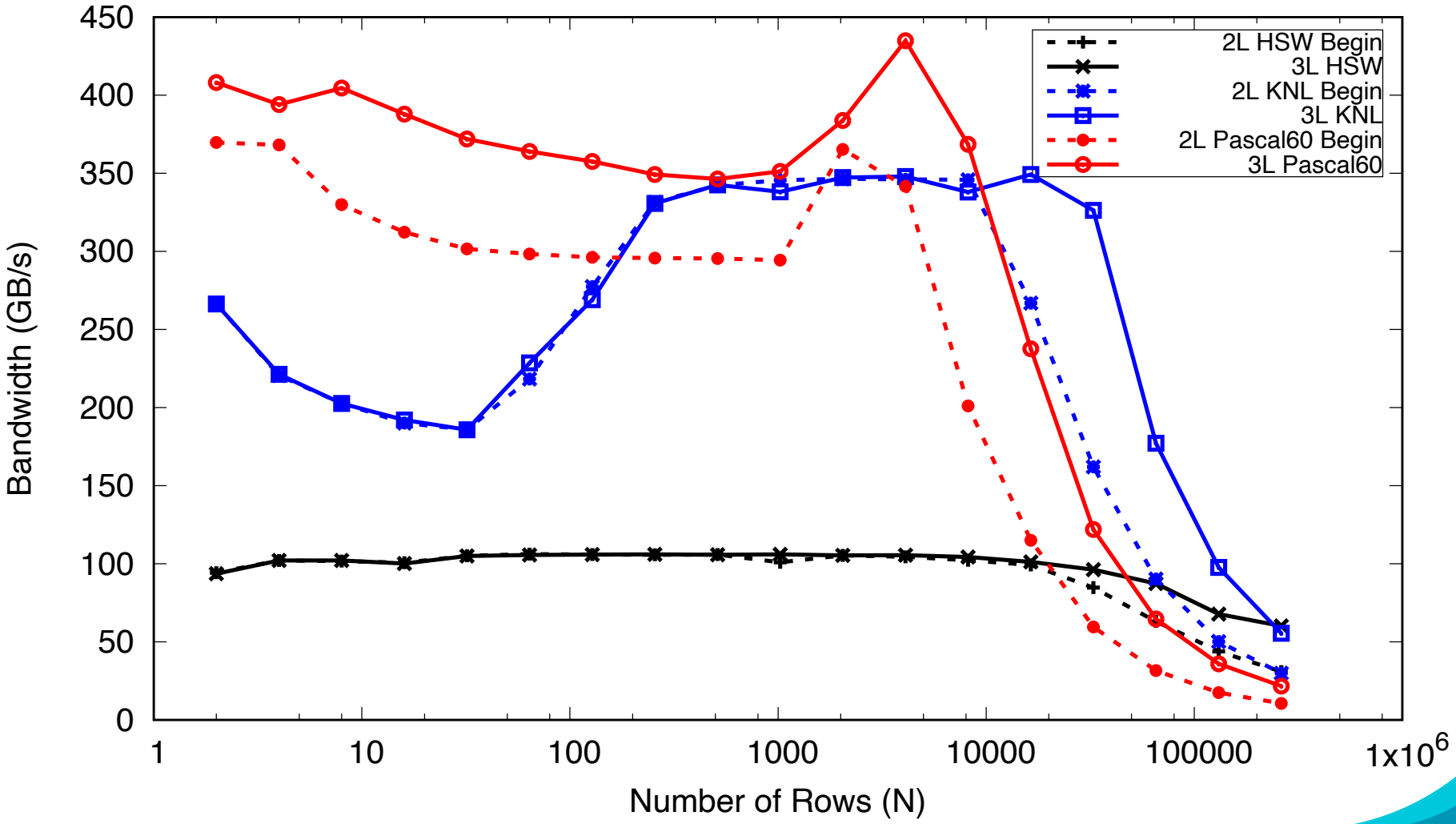
- Location: `Exercises/team_vector_loop/`
- Use the `single` policy instead of checking team rank
- Parallelize all three loop levels.

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

<ylAx> Exercise 06 (Three Level Parallelism) Fixed Size

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



- **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.

Kokkos Tools

Leveraging Kokkos' built-in instrumentation.

Learning objectives:

- The need for Kokkos-aware tools.
- How instrumentation helps.
- Simple profiling tools.
- Simple debugging tools.

Output from NVIDIA NVProf for Trilinos Tpetra

```
==278743== Profiling application: ./TpetraCore_Performance-CGSolve.exe --size=200
==278743== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   26.09%  380.32ms         1  380.32ms  380.32ms  380.32ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrsMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>>::pack_functor<K
okkos::View<double*>, Kokkos::View<unsigned long const *>>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
                   22.28%  324.77ms         1  324.77ms  324.77ms  324.77ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<Kokkos::Impl::CudaFunctorAdapter<Tpetra::Details::Impl::ConvertColumnIndicesFromGlobalToLocal<int, __int64, Kokkos::Device<Kokkos
::Cuda, Kokkos::CudaUVMSpace>, unsigned long, unsigned long>, Kokkos::RangePolicy<>, unsigned long, void>, Kokkos::RangePolicy<>, Kokkos::Invali
dType, Kokkos::Cuda>>(int)
                   21.83%  318.26ms        77  4.1332ms  3.8786ms  22.643ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosSparse::Impl::SPMV_Functor<KokkosSparse::CrsMatrix<double const , int const , Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpac
e>, Kokkos::MemoryTraits<unsigned int=1>, unsigned long const >, Kokkos::View<double const *>, Kokkos::View<double*>, int=0, bool=0>, Kokkos::Te
amPolicy<>, Kokkos::Cuda>>(double const )
                   15.51%  226.15ms         1  226.15ms  226.15ms  226.15ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrsMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>>::pack_functor<K
okkos::View<int*>, Kokkos::View<unsigned long const *>>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
                    3.60%  52.486ms       227  231.22us  230.17us  232.93us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::Axpby_Functor<double, Kokkos::View<double const *>, double, Kokkos::View<double*>, int=2, int=2, int>, Kokkos::Ran
gePolicy<>, Kokkos::Cuda>>(double)
                    1.86%  27.174ms        13  2.0903ms  1.0560us  27.157ms  [CUDA memcpy HtoD]
                    1.81%  26.350ms       153  172.22us  138.27us  206.08us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<KokkosBlas::Impl::DotFunctor<Kokkos::View<double>, Kokkos::View<double const *>, Kokkos::View<double const *>, int>, Kokkos::Rang
ePolicy<>, Kokkos::InvalidType, Kokkos::Cuda>>(double)
                    1.61%  23.431ms         1  23.431ms  23.431ms  23.431ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double const *>, Kokkos::View<double const *>, Kokkos::View<double*>, int=2, int=0,
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
                    1.39%  20.299ms         1  20.299ms  20.299ms  20.299ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double const *>, Kokkos::View<double const *>, Kokkos::View<double*>, int=2, int=2,
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
```

Output from NVIDIA NVProf for Trilinos Tpetra

```
==278743== Profiling application: ./TpetraCore_Performance-CGSolve.exe --size=200
==278743== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   26.09%  380.32ms         1  380.32ms  380.32ms  380.32ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrsMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>>::pack_functor<K
okkos::View<double*>, Kokkos::View<unsigned long const *>>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
                   22.28%  324.77ms         1  324.77ms  324.77ms  324.77ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<Kokkos::Impl::CudaFunctorAdapter<Tpetra::Details::Impl::ConvertColumnIndicesFromGlobalToLocal<int, __int64, Kokkos::Device<Kokkos
::Cuda, Kokkos::CudaUVMSpace>, unsigned long, unsigned long>, Kokkos::RangePolicy<>, unsigned long, void>, Kokkos::RangePolicy<>, Kokkos::Invali
dType, Kokkos::Cuda>>(int)
                   21.83%  318.26ms        77  4.1332ms  3.8786ms  22.643ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosSparse::Impl::SPMV_Functor<KokkosSparse::CrsMatrix<double const , int const , Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpac
e>, Kokkos::MemoryTraits<unsigned int=1>, unsigned long const >, Kokkos::View<double const *>, Kokkos::View<double*>, int=0, bool=0>, Kokkos::Te
amPolicy<>, Kokkos::Cuda>>(double const )
                   15.51%  226.15ms         1  226.15ms  226.15ms  226.15ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrsMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>>::pack_functor<K
okkos::View<int*>, Kokkos::View<unsigned long const *>>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
                    3.60%  52.486ms       227  231.22us  230.17us  232.93us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::Axpby_Functor<double, Kokkos::View<double const *>, double, Kokkos::View<double*>, int=2, int=2, int>, Kokkos::Ran
gePolicy<>, Kokkos::Cuda>>(double)
                    1.86%  27.174ms        13  2.0903ms  1.0560us  27.157ms  [CUDA memcpy HtoD]
                    1.81%  26.350ms       153  172.22us  138.27us  206.08us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<KokkosBlas::Impl::DotFunctor<Kokkos::View<double>, Kokkos::View<double const *>, Kokkos::View<double const *>, int>, Kokkos::Rang
ePolicy<>, Kokkos::InvalidType, Kokkos::Cuda>>(double)
                    1.61%  23.431ms         1  23.431ms  23.431ms  23.431ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double const *>, Kokkos::View<double const *>, Kokkos::View<double*>, int=2, int=0,
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
                    1.39%  20.299ms         1  20.299ms  20.299ms  20.299ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double const *>, Kokkos::View<double const *>, Kokkos::View<double*>, int=2, int=2,
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
```

What are those Kernels doing?

Generic code obscures what is happening from the tools

Historically a lot of profiling tools are coming from a Fortran and C world:

- Focused on functions and variables
- C++ has a lot of other concepts:
  - Classes with member functions
  - Inheritance
  - Template Metaprogramming
- Abstraction Models (Generic Programming) obscure things
  - From a profiler perspective interesting stuff happens in the abstraction layer (e.g. `#pragma omp parallel`)
  - Symbol names get really complex due to deep template layers

Instrumentation enables context information to reach tools.

Most profiling tools have an instrumentation interface

- E.g. nvtx for NVIDIA, ITT for Intel.
- Allows to name regions
- Sometimes can mark up memory operations.

Instrumentation enables context information to reach tools.

Most profiling tools have an instrumentation interface

- E.g. nvtx for NVIDIA, ITT for Intel.
- Allows to name regions
- Sometimes can mark up memory operations.

KokkosP

Kokkos has its own instrumentation interface KokkosP, which can be used to write tools.

- Knows about parallel dispatch
- Knows about allocations, deallocations and deep_copy
- Provides region markers
- Leverages naming information (kernels, Views)

There are two components to Kokkos Tools: the KokkosP instrumentation interface and the actual Tools.

KokkosP Interface

- The internal instrumentation layer of Kokkos.
- Always available even in release builds.
- Zero overhead if no tool is loaded.

Kokkos Tools

- Tools leveraging the KokkosP instrumentation layer.
- Are loaded at runtime by Kokkos.
  - Set `KOKKOS_TOOLS_LIBS` environment variable to load a shared library.
  - Compile tools into the executable and use the API callback setting mechanism.

Download tools from
https://github.com/kokkos/kokkos-tools

- Tools are largely independent of the Kokkos configuration
  - May need to use the same C++ standard library.
  - Use the same tool for CUDA and OpenMP code for example.
- We recommend you build the tools with CMake

```
cd kokkos-tools && cmake -B build
cmake --build build --parallel 4
cmake --install build --prefix /where/to/install/the/tools
```

Loading Tools:

- Set `KOKKOS_TOOLS_LIBS` environment variable to the full path to the shared library of the tool.
- Kokkos dynamically loads symbols from the library during `initialize` and fills function pointers.
- If no tool is loaded the overhead is a function pointer comparison to `nullptr`.

An Example Code

```
View<double*> a("A",N);
View<double*, HostSpace> h_a = create_mirror_view(a);

Profiling::pushRegion("Setup");
parallel_for("Init_A",RangePolicy<h_exec_t>(0,N),
  KOKKOS_LAMBDA(int i) { h_a(i) = i; });
deep_copy(a,h_a);
Profiling::popRegion();

Profiling::pushRegion("Iterate");
for(int r=0; r<10; r++) {
  View<double*> tmp("Tmp",N);
  parallel_scan("K_1",RangePolicy<exec_t>(0,N),
    KOKKOS_LAMBDA(int i, double& lsum, bool f) {
      if(f) tmp(i) = lsum;
      lsum += a(i);
  });
  double sum;
  parallel_reduce("K_2",N, KOKKOS_LAMBDA(int i, double& lsum) {
    lsum += tmp(i);
  },sum);
}
Profiling::popRegion();
```

Output of: `nvprof ./test.cuda`

```
==141309== Profiling application: ./test.cuda
==141309== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   40.95%  1.4516ms        20  72.580us  65.215us  81.663us  _ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_12Pa
rallelScanIZ4mainEUliRdbE_NS_11RangePolicyIJNS_4CudaEEEES6_EEEEvT_
                   40.75%  1.4444ms        18  80.246us  1.1520us  1.4186ms  [CUDA memcpy HtoD]
                    8.84%  313.34us        11  28.485us  28.415us  28.703us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos
::Impl::ParallelFor<Kokkos::Impl::ViewValueFunctor<Kokkos::Cuda, double, bool=1>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(Kokkos::Cuda)
                    7.91%  280.25us        10  28.025us  27.423us  29.024us  _ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_14Pa
rallelReduceINS0_18CudaFunctorAdapterIZ4mainEUliRdE_NS_11RangePolicyIJNS_4CudaEEEEdvEES8_NS_11InvalidTypeES7_EEEEvT_
                    1.20%  42.592us        28  1.5210us  1.3440us  2.1760us  [CUDA memcpy DtoH]
                    0.13%  4.5760us         1  4.5760us  4.5760us  4.5760us  Kokkos::_GLOBAL__N__52_tmpxft_0001ee3d_00000000_6_Kokkos_Cu
da_Locks_cpp1_ii_915ea793::init_lock_array_kernel_atomic(void)
                    0.08%  2.8480us         1  2.8480us  2.8480us  2.8480us  Kokkos::Impl::_GLOBAL__N__55_tmpxft_0001ee3b_00000000_6_Kok
kos_Cuda_Instance_cpp1_ii_a8bc5097::query_cuda_kernel_arch(int*)
                    0.08%  2.6880us         1  2.6880us  2.6880us  2.6880us  Kokkos::_GLOBAL__N__52_tmpxft_0001ee3d_00000000_6_Kokkos_Cu
da_Locks_cpp1_ii_915ea793::init_lock_array_kernel_threadid(int)
                    0.06%  2.1440us         2  1.0720us  1.0560us  1.0880us  [CUDA memset]
```

Output of: `nvprof ./test.cuda`

```
==141309== Profiling application: ./test.cuda
==141309== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   40.95%  1.4516ms        20  72.580us  65.215us  81.663us  _ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_12Pa
rallelScanIZ4mainEUliRdbE_NS_11RangePolicyIJNS_4CudaEEEES6_EEEEvT_
                   40.75%  1.4444ms        18  80.246us  1.1520us  1.4186ms  [CUDA memcpy HtoD]
                    8.84%  313.34us        11  28.485us  28.415us  28.703us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos
::Impl::ParallelFor<Kokkos::Impl::ViewValueFunctor<Kokkos::Cuda, double, bool=1>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(Kokkos::Cuda)
                    7.91%  280.25us        10  28.025us  27.423us  29.024us  _ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_14Pa
rallelReduceINS0_18CudaFunctorAdapterIZ4mainEUliRdE_NS_11RangePolicyIJNS_4CudaEEEEdvEES8_NS_11InvalidTypeES7_EEEEvT_
                    1.20%  42.592us        28  1.5210us  1.3440us  2.1760us  [CUDA memcpy DtoH]
                    0.13%  4.5760us         1  4.5760us  4.5760us  4.5760us  Kokkos::_GLOBAL__N__52_tmpxft_0001ee3d_00000000_6_Kokkos_Cu
da_Locks_cpp1_ii_915ea793::init_lock_array_kernel_atomic(void)
                    0.08%  2.8480us         1  2.8480us  2.8480us  2.8480us  Kokkos::Impl::_GLOBAL__N__55_tmpxft_0001ee3b_00000000_6_Kok
kos_Cuda_Instance_cpp1_ii_a8bc5097::query_cuda_kernel_arch(int*)
                    0.08%  2.6880us         1  2.6880us  2.6880us  2.6880us  Kokkos::_GLOBAL__N__52_tmpxft_0001ee3d_00000000_6_Kokkos_Cu
da_Locks_cpp1_ii_915ea793::init_lock_array_kernel_threadid(int)
                    0.06%  2.1440us         2  1.0720us  1.0560us  1.0880us  [CUDA memset]
```

Let us make one larger:

```
_ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0
_14ParallelReduceINS0_18CudaFunctorAdapterIZ4mainEUliRdE
_NS_11RangePolicyIJNS_4CudaEEEEdvEES8_NS_11InvalidTypeES7_EEEEvT_
```

And demangled:

```
void Kokkos::Impl::cuda_parallel_launch_local_memory
<Kokkos::Impl::ParallelReduce<Kokkos::Impl::CudaFunctorAdapter
<main::{lambda(int, double&)#1}, Kokkos::RangePolicy<Kokkos::Cuda>
double, void>, Kokkos::Cuda, Kokkos::InvalidType, Kokkos::RangePol
(Kokkos::Impl::ParallelReduce<Kokkos::Impl::CudaFunctorAdapter<
main::{lambda(int, double&)#1}, Kokkos::RangePolicy<Kokkos::Cuda>,
double, void>, Kokkos::Cuda, Kokkos::InvalidType, Kokkos::RangePol
```

Aaa this is horrifying can't we do better??

Aaa this is horrifying can't we do better??

Lets use SimpleKernelTimer from Kokkos Tools:

- Simple tool producing a summary similar to nvprof
- Good way to get a rough overview of whats going on
- Writes a file `HOSTNAME-PROCESSID.dat` per process
- Use the reader accompanying the tool to read the data

Usage:

```
git clone git@github.com:kokkos/kokkos-tools
cd kokkos-tools/profiling/simple_kernel_timer
make
export KOKKOS_TOOLS_LIBS=${PWD}/kp_kernel_timer.so
export PATH=${PATH}:${PWD}
cd ${WORKDIR}
./text.cuda
kp_reader *.dat
```

Output from SimpleKernelTimer:

```
Regions:

-                                                                                Iterate
 (Region)          0.02977           4         0.00744 147.131  60.772
-                                                                                  Setup
 (Region)          0.00769           4         0.00192  38.010  15.700

---------------------------------------------------------------------
Kernels:

-                                                  Kokkos::View::initialization [A_mirror]
 (ParFor)          0.00878           4         0.00220  43.402  17.927
-                                                                                      K_1
 (ParScan)         0.00651          40         0.00016  32.178  13.291
-                                                   Kokkos::View::initialization [Tmp]
 (ParFor)          0.00191          40         0.00005   9.454   3.905
-                                                                                      K_2
 (ParRed)          0.00169          40         0.00004   8.372   3.458
-                                                                                  Init_A
 (ParFor)          0.00100           4         0.00025   4.965   2.051
-                                                     Kokkos::View::initialization [A]
 (ParFor)          0.00033           4         0.00008   1.629   0.673

---------------------------------------------------------------------
Summary:

Total Execution Time (incl. Kokkos + non-Kokkos):                0.04899 seconds
Total Time in Kokkos kernels:                                    0.02024 seconds
   -> Time outside Kokkos kernels:                               0.02876 seconds
   -> Percentage in Kokkos kernels:                                41.31 %
Total Calls to Kokkos Kernels:                                       132
```

Output from SimpleKernelTimer:

```
Regions:

-                                                                                Iterate
 (Region)          0.02977           4         0.00744 147.131  60.772
-                                                                                  Setup
 (Region)          0.00769           4         0.00192  38.010  15.700

---------------------------------------------------------------------
Kernels:

-                                                  Kokkos::View::initialization [A_mirror]
 (ParFor)          0.00878           4         0.00220  43.402  17.927
-                                                                                      K_1
 (ParScan)         0.00651          40         0.00016  32.178  13.291
-                                                   Kokkos::View::initialization [Tmp]
 (ParFor)          0.00191          40         0.00005   9.454   3.905
-                                                                                      K_2
 (ParRed)          0.00169          40         0.00004   8.372   3.458
-                                                                                  Init_A
 (ParFor)          0.00100           4         0.00025   4.965   2.051
-                                                     Kokkos::View::initialization [A]
 (ParFor)          0.00033           4         0.00008   1.629   0.673

---------------------------------------------------------------------
Summary:

Total Execution Time (incl. Kokkos + non-Kokkos):                0.04899 seconds
Total Time in Kokkos kernels:                                    0.02024 seconds
   -> Time outside Kokkos kernels:                               0.02876 seconds
   -> Percentage in Kokkos kernels:                                41.31 %
Total Calls to Kokkos Kernels:                                       132
```

Will introduce *Regions* later.

Kernel Naming

Naming Kernels avoid seeing confusing Profiler output!

Lets look at Tpetra again with the Simple Kernel Timer Loaded:

At the top we get Region output:

```
Regions:

- CG: global
 (REGION)   0.547101 1 0.547101 26.922698 5.470153
- CG: spmv
 (REGION)   0.323189 77 0.004197 15.904024 3.231379
- CG: axpby
 (REGION)   0.091971 154 0.000597 4.525865 0.919565
- KokkosBlas::axpby[ETI]
 (REGION)   0.055017 228 0.000241 2.707360 0.550081
- KokkosBlas::update[ETI]
 (REGION)   0.030842 2 0.015421 1.517718 0.308370
- CG: dot
 (REGION)   0.028661 153 0.000187 1.410413 0.286568
- KokkosBlas::dot[ETI]
 (REGION)   0.028120 153 0.000184 1.383756 0.281152
```

Then we get kernel output:

```
Kernels:

- Tpetra::CrsMatrix::sortAndMergeIndicesAndValues
 (ParRed)   0.708770 1 0.708770 34.878388 7.086590
- KokkosSparse::spmv<NoTranspose,Dynamic>
 (ParFor)   0.319268 77 0.004146 15.711118 3.192184
- Tpetra::Details::Impl::ConvertColumnIndicesFromGlobalToLocal
 (ParRed)   0.292309 1 0.292309 14.384452 2.922633
- Tpetra::CrsMatrix pack values
 (ParFor)   0.267800 1 0.267800 13.178373 2.677581
- Tpetra::CrsMatrix pack column indices
 (ParFor)   0.157867 1 0.157867 7.768592 1.578422
- KokkosBlas::Axpby::S15
 (ParFor)   0.054251 227 0.000239 2.669699 0.542429
- Kokkos::View::initialization [Tpetra::CrsMatrix::val]
 (ParFor)   0.033584 2 0.016792 1.652666 0.335789
- Kokkos::View::initialization [lgMap]
 (ParFor)   0.033417 2 0.016708 1.644441 0.334118
- KokkosBlas::dot<1D>
 (ParRed)   0.027782 153 0.000182 1.367155 0.277778
```

Understanding MemorySpace Utilization is critical

Three simple tools for understanding memory utilization:

- MemoryHighWaterMark: just the maximum utilization for each memory space.
- MemoryUsage: Timeline of memory usage.
- MemoryEvents: allocation, deallocation and deep_copy.
  - Name, Memory Space, Pointer, Size

```
# Memory Events
# Time      Ptr                 Size        MemSpace      Op          Name
0.000776    0x7f095f600000       8000000             Cuda Allocate    A
0.000910          0x1cb4680      8000000             Host Allocate    A_mirror
0.001571 PushRegion Setup {
0.003754 } PopRegion
0.003756 PushRegion Iterate {
0.004100    0x7f0960000000       8000000             Cuda Allocate    Tmp
0.004451    0x7f0960000000      -8000000             Cuda DeAllocate  Tmp
...
0.010350    0x7f0960000000       8000000             Cuda Allocate    Tmp
0.010605    0x7f0960000000      -8000000             Cuda DeAllocate  Tmp
0.010753 } PopRegion
0.010753          0x1cb4680      -8000000            Host DeAllocate  A_mirror
0.010766    0x7f095f600000      -8000000             Cuda DeAllocate  A
```

Adding region markers to capture more code structure

Region Markers are helpful to:

- Find where time is spent outside of kernels.
- Group Kernels which belong together.
- Structure code profiles.
  - For example bracket *setup* or *solve* phase.

Adding region markers to capture more code structure

Region Markers are helpful to:

- Find where time is spent outside of kernels.
- Group Kernels which belong together.
- Structure code profiles.
  - For example bracket *setup* or *solve* phase.

Simple Push/Pop interface:

```
Kokkos::Profiling::pushRegion("Label");
...
Kokkos::Profiling::popRegion();
```

The simplest tool to leverage regions is the **Space Time Stack**:

- **Bottom Up** and **Top Down** data representation
- Can do MPI aggregation if compiled with MPI support
- Also aggregates memory utilization info.

```
BEGIN KOKKOS PROFILING REPORT:
TOTAL TIME: 0.0100131 seconds
TOP-DOWN TIME TREE:
<average time> <percent of total time> <percent time in Kokkos> <percent MPI imbalance> <remainder> <kernels per second> <number of calls> <name> [type]

==================
|-> 6.90e-03 sec 68.9% 33.9% 0.0% 66.1% 4.35e+03 1 Iterate [region]
|   |-> 1.55e-03 sec 15.5% 100.0% 0.0% ------ 10 K_1 [scan]
|   |-> 4.04e-04 sec 4.0% 100.0% 0.0% ------ 10 Kokkos::View::initialization [Tmp] [for]
|   |-> 3.80e-04 sec 3.8% 100.0% 0.0% ------ 10 K_2 [reduce]
|-> 1.84e-03 sec 18.4% 98.6% 0.0% 1.4% 1.09e+03 1 Setup [region]
|   |-> 1.59e-03 sec 15.9% 100.0% 0.0% ------ 1 "A"="A_mirror" [copy]
|   |-> 2.21e-04 sec 2.2% 100.0% 0.0% ------ 1 Init_A [for]
|-> 6.64e-04 sec 6.6% 100.0% 0.0% ------ 1 Kokkos::View::initialization [A_mirror] [for]
|-> 6.68e-05 sec 0.7% 100.0% 0.0% ------ 1 Kokkos::View::initialization [A] [for]

BOTTOM-UP TIME TREE:
...

KOKKOS HOST SPACE:
==================
MAX MEMORY ALLOCATED: 7812.5 kB
ALLOCATIONS AT TIME OF HIGH WATER MARK:
  100.0% A_mirror

KOKKOS CUDA SPACE:
==================
MAX MEMORY ALLOCATED: 15625.0 kB
ALLOCATIONS AT TIME OF HIGH WATER MARK:
  50.0% A
  50.0% Iterate/Tmp

Host process high water mark memory consumption: 161668 kB

END KOKKOS PROFILING REPORT.
```

Non-Blocking Dispatch implies asynchronous error reporting!

```
Profiling::pushRegion("Iterate");
for(int r=0; r<10; r++) {
  parallel_for("K_1",2*N, KOKKOS_LAMBDA(int i) {a(i) = i;});
  printf("Passed point A\n");
  double sum;
  parallel_reduce("K_2",N, KOKKOS_LAMBDA(int i, double& lsum) {
    lsum += a(i); },sum);
}
Profiling::popRegion();
```

Output of the run:

```
./test.cuda
Passed point A
terminate called after throwing an instance of 'std::runtime_error
  what():  cudaStreamSynchronize(m_stream) error( cudaErrorIllegal
  an illegal memory access was encountered
    Kokkos/kokkos/core/src/Cuda/Kokkos_Cuda_Instance.cpp:312
Traceback functionality not available
Aborted (core dumped)
```

Debugging with Tools

Kokkos Tools can be used to implement Debugging functionality.

Debugging with Tools

Kokkos Tools can be used to implement Debugging functionality.

The KernelLogger is a tool to localize errors and check the actual runtime flow of a code.

- As other tools it inserts fences - which check for errors.
- Prints out Kokkos operations as they happen.

Debugging with Tools

Kokkos Tools can be used to implement Debugging functionality.

The KernelLogger is a tool to localize errors and check the actual runtime flow of a code.

- As other tools it inserts fences - which check for errors.
- Prints out Kokkos operations as they happen.

Output from the above test case with KernelLogger:

```
KokkosP: Allocate<Cuda> name: A pointer: 0x7f598b800000 size: 8000
KokkosP: Executing parallel-for kernel on device 0 with unique exe
KokkosP: Kokkos::View::initialization [A]
KokkosP: Execution of kernel 0 is completed.
KokkosP: Entering profiling region: Iterate
KokkosP: Executing parallel-for kernel on device 0 with unique exe
KokkosP: Iterate
KokkosP:   K_1
terminate called after throwing an instance of 'std::runtime_error
  what():  cudaDeviceSynchronize() error( cudaErrorIllegalAddress)
Traceback functionality not available
```

The standard Kokkos profiling approach

Understand Kokkos Utilization (SimpleKernelTimer)

- Check how much time in kernels
- Identify HotSpot Kernels

Run Memory Analysis (MemoryEvents)

- Are there many allocations/deallocations - 5000/s is OK.
- Identify temporary allocations which could be hoisted

Identify Serial Code Regions (SpaceTimeStack)

- Add Profiling Regions
- Find Regions with low fraction of time spend in Kernels

Dive into individual Kernels

- Use connector tools (next subsection) to analyze kernels.
- E.g. use roof line analysis to find underperforming code.

Analyse a MiniMD variant with a serious performance issue.

Details:

- Location: `Exercises/tools_minimd/`
- Use standard Profiling Approach.
- Find the code location which causes the performance issue.
- Run with `miniMD.exe -s 20`

What should happen:

- Performance should be
- About 50% of time in a Force compute kernel
- About 25% in neighbor list creation

- Kokkos Tools provide an instrumentation interface **KokkosP** and **Tools** to leverage it.
- The interface is **always available** - even in release builds.
- Zero overhead if no tool is loaded during the run.
- Dynamically load a tool via setting `KOKKOS_TOOLS_LIBS` environment variable.
- Set callbacks directly in code for tools compiled into the executable.

Vendor and Independent Profiling GUIs

Connector tools translating Kokkos instrumentation.

Learning objectives:

- Understand what connectors provide
- Understand what tools are available

Kokkos Tools can also be used to interface and augment existing profiling tools.

- Provide context information like Kernel names
- Turn data collection on and off in a tool independent way

There are two ways this happens:

- Load a specific connector tool like `nvprof-connector`
  - For example for Nsight Compute and VTune
- Tools themselves know about Kokkos instrumentation
  - For example Tau

Use the `nvprof-connector` to interact with NVIDIA tools

Translates KokkosP hooks into NVTX instrumentation

- Works with all NVIDIA tools which understand NVTX
- Translates Regions and Kernel Dispatches

Use the `nvprof-connector` to interact with NVIDIA tools

Translates KokkosP hooks into NVTX instrumentation

- Works with all NVIDIA tools which understand NVTX
- Translates Regions and Kernel Dispatches

Initially wasn't very useful since regions are shown independently of kernels

Use the `nvprof-connector` to interact with NVIDIA tools

Translates KokkosP hooks into NVTX instrumentation

- Works with all NVIDIA tools which understand NVTX
- Translates Regions and Kernel Dispatches

Initially wasn't very useful since regions are shown independently of kernels

But CUDA 11 added renaming of Kernels based on Kokkos User feedback!

To enable kernel renaming you need to:

- Load the nvprof-connector via setting `KOKKOS_TOOLS_LIBS` in the run configuration.
- Go to `Tools > Preferences > Rename CUDA Kernels by NVTX` and set it on.

This does a few things:

- User Labels are now used as the primary name.
- You can still expand the row to see which actual kernels are grouped under it.
  - For example if multiple kernels have the same label
- The bars are now named `Label/GLOBAL_FUNCTION_NAME`.

To enable kernel renaming you need to:

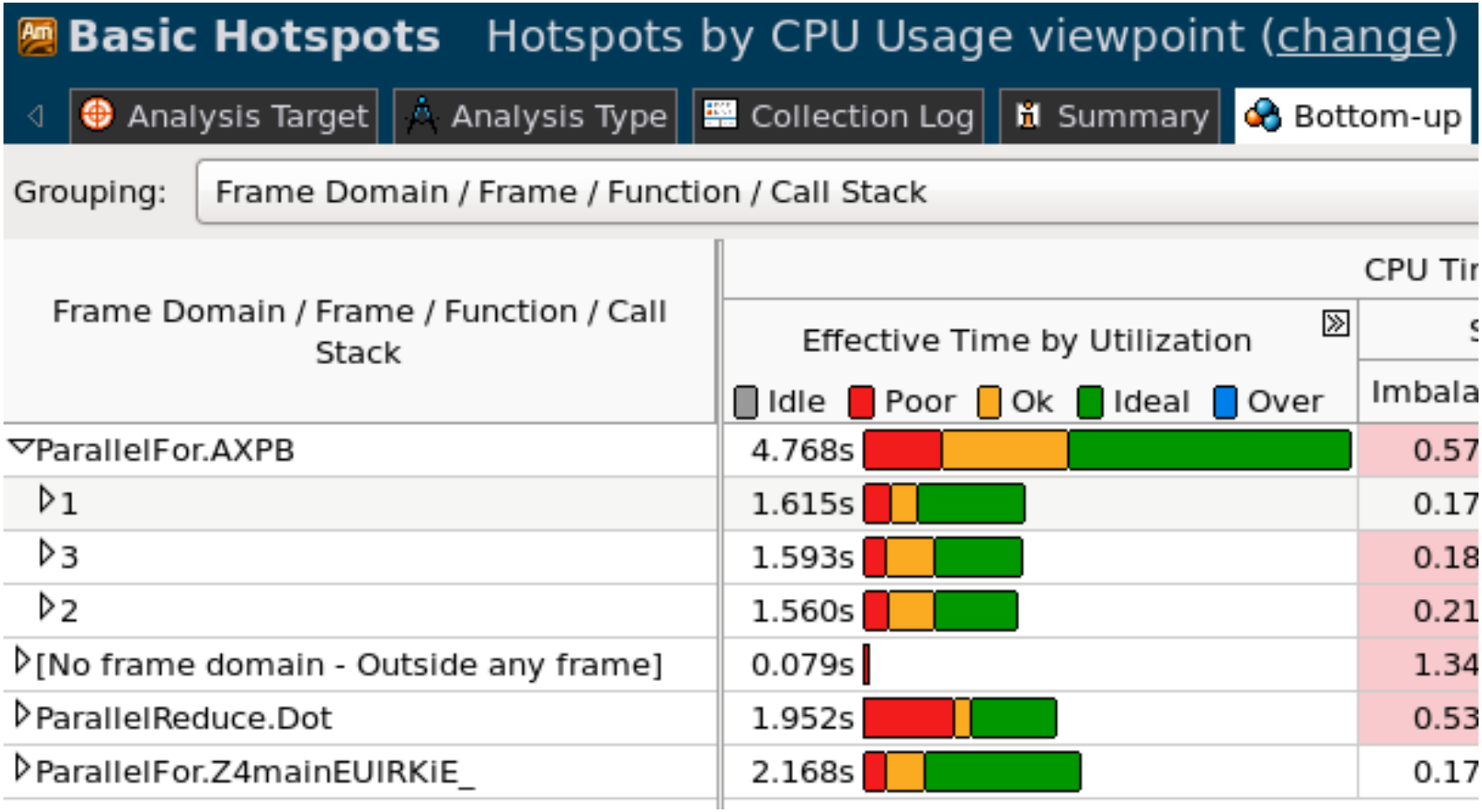
- Load the vtune-connector via setting `KOKKOS_TOOLS_LIBS` in the run configuration.
- Choose the `Frame Domain / Frame / Function / Call Stack` grouping in the bottom up panel.

This does a few things:

- User Labels are now used as the primary name.
- You can expand to see individual kernel invocations
- You can dive further into an individual kernel invocation to see function calls within.
- Focus in on a kernel or individual invocation and do more detailed analysis.

Also available: vtune-focused-connector:

- Used in conjunction with kernel-filter tool.
- Restricts profiling to a subset of kernels.


Basic Hotspots Hotspots by CPU Usage viewpoint (change)
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Grouping: Frame Domain / Frame / Function / Call Stack
Frame Domain / Frame / Function / Call Stack
CPU Tir
Effective Time by Utilization
Idle Poor Ok Ideal Over
Imbala
ParallelFor.AXPB 4.768s 0.57
1 1.615s 0.17
3 1.593s 0.18
2 1.560s 0.21
[No frame domain - Outside any frame] 0.079s 1.34
ParallelReduce.Dot 1.952s 0.53
ParallelFor.Z4mainEUlRKiE_ 2.168s 0.17

TAU is a widely used Profiling Tool supporting most platforms.

Tau supports:

- profiling
- sampling
- tracing

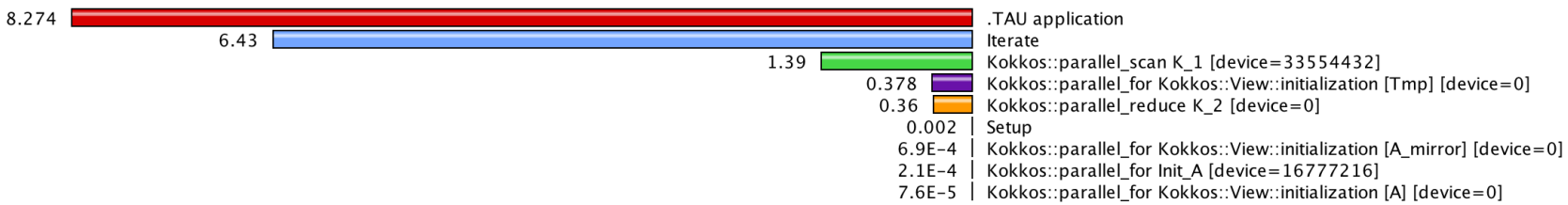
You do not need a connector tool for Tau!

To enable TAU's Kokkos integration simply

- Download and install TAU
- Launch your program with tau_exec (which will set `KOKKOS_TOOLS_LIBS` for you)

For questions contact tau-users@cs.uoregon.edu

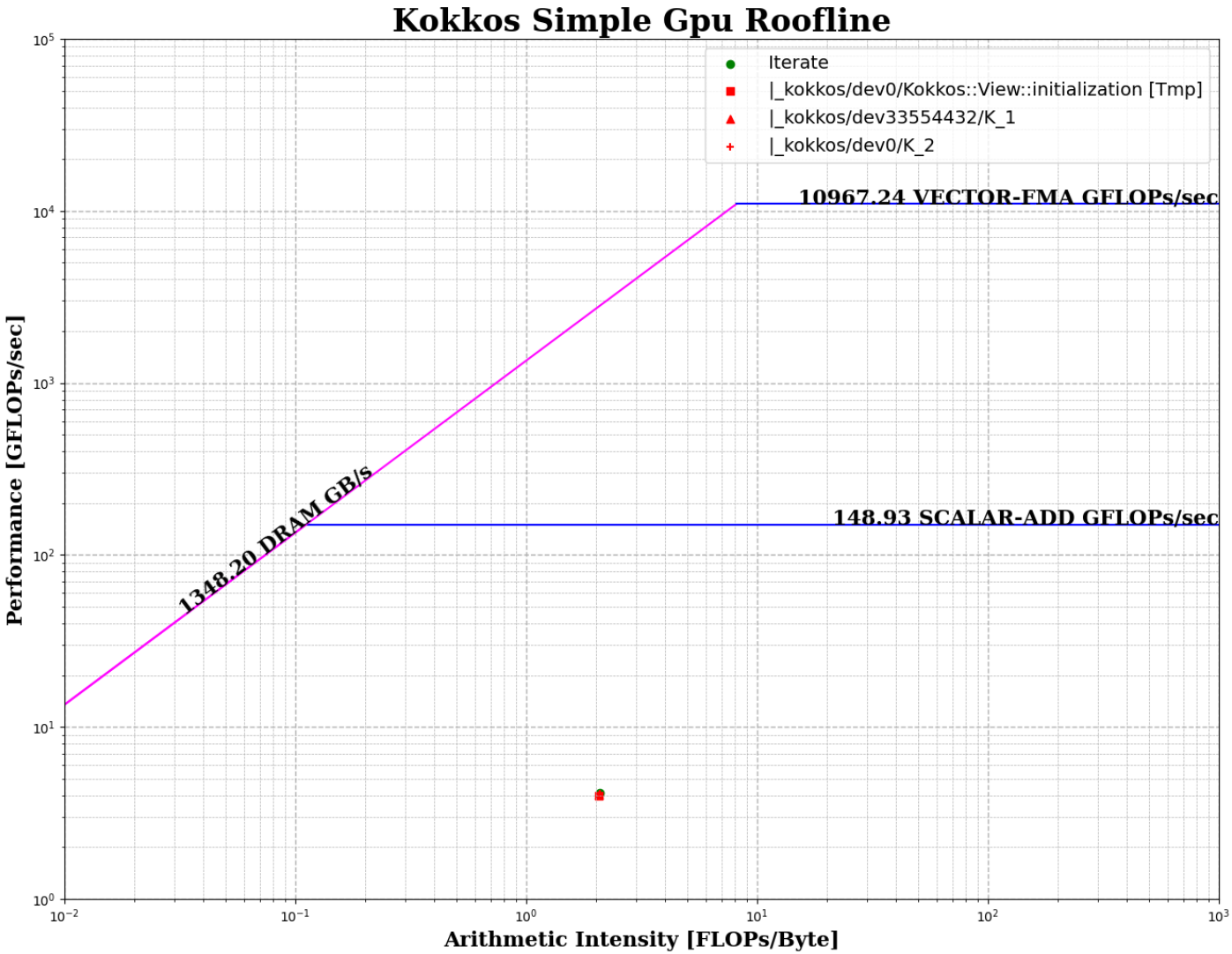
Tau will use Kokkos instrumentation to display names and regions as defined by Kokkos:



Timemory is a modular toolkit provided by NERSC that aims to simplify the creation of performance analysis tools by providing a common design pattern of classes which encapsulate how to perform a start+stop/sample/entry of "something". Each of these components (from timers to HW counters to other profilers) can be used individually with zero overhead from the library. It also provides wrappers and utilities for handling multiple components generically, data storage, writing JSON, comparing outputs, etc.

As a by-product this design, the library provides an large set of individual profiling libraries whose usage comes with the same ease as using the simple-timer tool: setting `KOKKOS_TOOLS_LIBS`.

- It also provides novel capabilities other tools don't, like simultaneous CPU/GPU roofline modeling.
- The usage here is simple:
  - `spack install timemory +kokkos_tools +kokkos_build_config [+mpi +cuda +cupti +papi +caliper ...]`
  - Wait 3 months while spack builds every software package ever from scratch
  - In `<PREFIX>/lib/timemory/kokkos_tools/` there will be 5 to 30+ Kokkos profiling libraries
- Roofline modeling requires one additional setup
  - `timemory-roofline -T "TITLE"-t gpu_roofline -- <CMD>`
  - Where everything after -- is just running your application
- For more information:
  https://github.com/NERSC/timemory


Kokkos Simple Gpu Roofline
Iterate
|_kokkos/dev0/Kokkos::View::initialization [Tmp]
|_kokkos/dev33554432/K_1
|_kokkos/dev0/K_2
10967.24 VECTOR-FMA GFLOPs/sec
148.93 SCALAR-ADD GFLOPs/sec
1348.20 DRAM GB/s
Performance [GFLOPs/sec]
Arithmetic Intensity [FLOPs/Byte]
10^0
10^1
10^2
10^3
10^4
10^5
10^{-2}
10^{-1}
10^0
10^1
10^2
10^3

- Caliper - Broad program analysis capabilities. UVM Profiling.
- HPCToolkit - Not a connector, but a sampling tool with great Kokkos support

- Connectors inject Kokkos specific information into vendor and academic tools.
- Helps readability of profiles.
- Removes your need to put vendor specific instrumentation in your code.
- Growing list of tools support Kokkos natively.

Custom Tools

How to write your own tools for the KokkosP interface.

Learning objectives:

- The KokkosP hooks
- Callback registration inside the application
- Throwaway debugging tools

KokkosTools also allow you to write your own tools!

- Implement a simple C interface.
- Only implement what you want to use!
- Full access to the entire instrumentation.

But why would I want to do that?

- Profiling tools which know about your code structure and properly categorize information.
- Add in situ analysis hooked into your CI system.
- Write debugging tools specific for your framework.
- Write throwaway debugging tools for larger apps, instead of recompiling.

KokkosTools also allow you to write your own tools!

- Implement a simple C interface.
- Only implement what you want to use!
- Full access to the entire instrumentation.

But why would I want to do that?

- Profiling tools which know about your code structure and properly categorize information.
- Add in situ analysis hooked into your CI system.
- Write debugging tools specific for your framework.
- Write throwaway debugging tools for larger apps, instead of recompiling.

We will first walk through the hooks and then illustate with an example.

Some Helper Classes

```
// Contains a unique device identifier.
struct KokkosPDeviceInfo { uint32_t deviceID; };

// Unique name of execution and memory spaces.
struct SpaceHandle { char name[64]; };
```

Some Helper Classes

```
// Contains a unique device identifier.
struct KokkosPDeviceInfo { uint32_t deviceID; };

// Unique name of execution and memory spaces.
struct SpaceHandle { char name[64]; };
```

Initialization and Finalization hooks

```
extern "C" void kokkosp_init_library(
  int loadseq, uint64_t version, uint32_t num_devinfos,
  KokkosPDeviceInfo* devinfos);
```

- Called during `Kokkos::initialize`
- Provides device ids used subsequently.
- Use this call to setup tool infrastructure.

```
extern "C" void kokkosp_finalize_library();
```

- Called during `Kokkos::finalize`
- Usually used to output results.

```
extern "C" {
  void kokkosp_begin_parallel_for(const char* name,
                                  uint32_t devid,
                                  uint64_t* kernid);
  void kokkosp_begin_parallel_reduce(const char* name,
                                     uint32_t devid,
                                     uint64_t* kernid);
  void kokkosp_begin_parallel_scan(const char* name,
                                   uint32_t devid,
                                   uint64_t* kernid);
};
```

- Called when a parallel dispatch is initiated.
- `name` is the user provided string or a typeid.
- `kernid` is set by the tool to match up with the end call.

```
extern "C" void kokkosp_end_parallel_for(uint64_t kernid);
extern "C" void kokkosp_end_parallel_reduce(uint64_t kernid);
extern "C" void kokkosp_end_parallel_scan(uint64_t kernid);
```

- Called when a parallel dispatch is done.
- `kernid` is the value the begin call set.

```
extern "C" void kokkosp_begin_deep_copy(
  SpaceHandle dst_hndl, const char* dst_name, const void* dst_ptr,
  SpaceHandle src_hndl, const char* src_name, const void* src_ptr,
  uint64_t size);
```

- Called when a deep_copy is started.
- Provides space handles, names, ptrs and size of allocations.

```
extern "C" void kokkosp_end_deep_copy();
```

- Called when a deep_copy is done.

```
extern "C" void kokkosp_allocate_data(SpaceHandle hndl,
  const char* name, void* ptr, uint64_t size);
extern "C" void kokkosp_deallocate_data(SpaceHandle hndl,
  const char* name, void* ptr, uint64_t size);
```

- Called when allocating or deallocating data.

Sometimes it is useful to build a tool into an executable.

Callback Registration

Kokkos Tools provide a callback setting system to set tool callbacks from within the application.

Takes the form of:

```
void set_HOOK_callback(HOOK_FUNCTION_PTR callback);
```

Where HOOK is one of

```
init finalize push_region pop_region begin_parallel_for
end_parallel_for begin_parallel_reduce end_parallel_reduce
begin_parallel_scan end_parallel_scan begin_fence end_fence
allocate_data deallocate_data begin_deep_copy end_deep_copy
```

One can also store a callback set, reload it and pause tool calls

```
EventSet get_callbacks();   void set_callbacks(EventSet);
void pause_tools();         void resume_tools();
```

Example:

```
#include <Kokkos_Core.hpp>
using Kokkos::Profiling;
using Kokkos::Tools::Experimental;
using Kokkos;

void kokkosp_allocate_data(SpaceHandle space,
  const char* label, const void* const ptr, uint64_t size) {
  printf("Allocate: [%s] %lu\n",label,size);
}
void kokkosp_deallocate_data(SpaceHandle space,
  const char* label, const void* const ptr, uint64_t size) {
  printf("Deallocate: [%s] %lu\n",label,size);
}

int main(int argc, char* argv[]) {
  initialize(argc, argv);
  set_allocate_data_callback(kokkosp_allocate_data);
  set_deallocate_data_callback(kokkosp_deallocate_data);
  ...
  finalize();
}
```

Example: Throwaway Debugging Tool

Sometimes you just need to know what is in a View before and after entering a kernel for the 5th time:

- The view is on the GPU and its on some rank of a large run.
- Recompiling the app takes hours.

Sometimes you just need to know what is in a View before and after entering a kernel for the 5th time:

- The view is on the GPU and its on some rank of a large run.
- Recompiling the app takes hours.

Simple Kokkos tool could do it!

What we need:

- Store the pointer and size of the view with a specific label when it gets allocated.
- Print the View when entering a kernel and before exiting it.
- Make sure the view didn't get deallocated in the mean time.

Store the pointer:

```
int* data; uint64_t N; int count;
extern "C" void kokkosp_allocate_data(SpaceHandle handle,
  const char* name, void* ptr, uint64_t size) {
  if(strcmp(name,"PuppyWeights")==0) {
    data = (int*)ptr+32; N = size; count = 0;
}}
```

Print the View:

```
void print_data() {
  std::vector<int> hcpy(N);
  cudaMemcpy(hcpy.data(),data,N*sizeof(int));
  for(int i=0;i<N;++i) printf("(%d␣%d)",i,hcpy[i]); printf("\n");
}
extern "C" void kokkosp_begin_parallel_for(const char* name,
  uint32_t, uint64_t* kernid) {
  if(strcmp(name,"PuppyOnCouch")==0) {
    count++; if(count==5) print_data(); *kernid=1;
  } else { *kernid = 0; }
}
extern "C" void kokkosp_end_parallel_for(uint64_t kernid) {
  if(kernid == 1 && count==5) print_data();
}
```

```
#include <Kokkos_Core.hpp>
#include <cmath>

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  {
    int N = argc > 1 ? atoi(argv[1]) : 12;
    int R = argc > 2 ? atoi(argv[2]) : 10;
    Kokkos::View<double*> a("PuppyWeights",N);

    for(int r=0; r<R; r++) {
      Kokkos::parallel_for("PuppyOnCouch",N,KOKKOS_LAMBDA(int i)
                           { a(i) = i*r; });
    }
  }
  Kokkos::finalize();
}
```

Output:

```
(0 0) (1 4) (2 8) (3 12)
(0 0) (1 5) (2 10) (3 15)
```

Implementing your own tools is easy!

- Simply implement the needed C callback functions.
- Only implement what you need.
- Goal is to make it simple enough so that one-off tools are a viable debugging help.

Callback registration for applications

- The callback registration system allows to embed tools in applications.
- Store callback sets and restore them.

Hierarchal 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.

Kokkos Tools:

- Kokkos Tools provide an instrumentation interface **KokkosP** and **Tools** to leverage it.
- The interface is **always available** - even in release builds.
- Zero overhead if no tool is loaded during the run.
- Dynamically load a tool via setting `KOKKOS_TOOLS_LIBS` environment variable.
- Set callbacks in code for tools compiled into the executable.

Kokkos Connector Tools:

- Connectors inject Kokkos specific information into vendor and academic tools.
- Helps readability of profiles.
- Removes need to put vendor specific instrumentation in codes.
- Growing list of tools support Kokkos natively.

Implementing your own tools is easy!

- Simply implement the needed C callback functions.
- Only implement what you need.
- The callback registration system allows to embed tools in applications.

The Kokkos Lectures

Watch the Kokkos Lectures for all of those and more in-depth explanations or do them on your own.

- Module 1: Introduction, Building and Parallel Dispatch
- Module 2: Views and Spaces
- Module 3: Data Structures + MultiDimensional Loops
- Module 4: Hierarchical Parallelism
- Module 5: Tasking, Streams and SIMD
- Module 6: Internode: MPI and PGAS
- Module 7: Tools: Profiling, Tuning and Debugging
- Module 8: Kernels: Sparse and Dense Linear Algebra

https://kokkos.link/the-lectures

Online Resources:

- https://github.com/kokkos:
  - Primary Kokkos GitHub Organization
- https://kokkos.link/the-lectures:
  - Slides, recording and Q&A for the Full Lectures
- https://github.com/kokkos/kokkos/wiki:
  - Wiki including API reference
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Can whitelist domains, or invite individual people.