GPU Profiling and Optimizing xRAGE Final Report

Jason Zeng, Ivan Gonzalez
August 18, 2023

1 Introduction

Our project’s objective is to increase the efficiency of GPU-enabled kernels in xRAGE. To do so, we conduct GPU profiling with NSight Systems on xRAGE tests unsplit_sod_1d and unsplit_sedov_2d to identify bottlenecks and understand the behavior of the GPU during code execution. Next, we analyze these generated GPU profiles to locate the lines of code whose optimization have the most potential for improving runtime. We replicate the structure of the code in smaller test problems that are easier to understand, edit, and run quickly. Within these test problems, we implement two different methods of improving performance: transformation of nested loops into a single MDRange-Policy and hierarchical parallelization using teams of threads. Both methods show speedups in the test code, and after transferring them to xRAGE, they both show up to 30x speedups on various computing platforms. Profiling the edited versions of xRAGE reveals that the GPU successfully executed the bottlenecks with greater efficiency.

2 Profiling

We profiled xRAGE on machines with Nvidia GPUs, so we primarily used the profiler provided by Nvidia in addition to the profiling tools provided by Kokkos. Nsight Systems, the profiler provided by Nvidia, is a system-wide performance analysis tool used to profile applications which can be loaded as a module on all the HPC machines we used, i.e., Snow, Darwin, and RZansel, and downloaded onto a local machine to view the reports generated. Kokkos-tools is a set of lightweight profiling and debugging utilities which automatically hooks into the Kokkos runtime, allowing the user to gather information about all Kokkos kernels.

When profiling xRAGE with both the unsplit_sod_1d and unsplit_sedov_2d input decks, Nsight Systems reported that hydro_unsplit_library function getFluxesKokkos was taking up the majority of runtime.

In addition, when using the Kokkos-tools simple-kernel-timer, the unsplit_sedov_2d test showed a similar result, with the getFluxesKokkos function Kokkos kernels get_face_fluxes_internal, get_face_fluxes_LO_side, and get_face_fluxes_HI_side all being within the top twelve most expensive Kokkos kernels.
With this information, we knew that we needed to investigate why the `getFluxesKokkos` function was taking up so much of runtime, and how to optimize it so we could achieve application-wide speedup.
Testing

3.1 Preliminary Testing

Before attempting to optimize the `getFluxesKokkos` function, we investigated why it was taking such a large portion of total runtime. We wrote test code to test our main hypothesis: a `parallel_reduce` is blocked when within a `for` loop. When creating and profiling the test code, the profiler showed that a significant portion of runtime was taken by `CudaStreamSynchronize`, which signifies that the GPU is possibly underutilized. Moreover, Nsight-Systems shows a `CudaStreamSynchronize` is called after every `parallel_reduce` within the `for` loop, which suggests that programs which execute a large number of Kokkos kernels within `for` loops would benefit from combining the reductions or using some other parallel method provided by Kokkos.
Figure 6: Kokkos Blocking Test Code

do not hallucinate.

```c
int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  {
    const int n = std::numeric_limits<int>::max()/2;
    double sum[3];
    for (int k=0; k<3; ++k) {
      Kokkos::parallel_reduce(Kokkos::RangePolicy(0,n),
        KOKKOS_LAMBDA (int i, double& lsum) {
          lsum += std::pow(-1,i)*4.0/(2*i+1);
        }, Kokkos::Sum<double>(sum[k]));
    }
    for (int j=0; j<3; ++j) {
      printf("%.10f\n", sum[j]);
    }
  }
  Kokkos::finalize();
  return 0;
}
```

Hierarchical parallelism and MDRange policies were considered because both are methods provided by Kokkos to parallelize code, and while the view of views approach did not see improvements in runtime, and ultimately yielded no results, we initially considered this approach because the Kokkos documentation states that “The call is potentially asynchronous if the ReducerArgument is not a scalar type” in regards to parallel_reduce. The hierarchical parallelism and MDRange-Policy approaches both showed improvements in performance, and were ultimately implemented in getFluxesKokkos.
Kokkos::View<Kokkos::View<double, Kokkos::CudaSpace>*, Kokkos::CudaUVMSpace> dtmins
(Kokkos::view_alloc ("dtmins", Kokkos::WithoutInitializing), numOuter);
for (int k = 0; k < numOuter; ++k) {
    const std::string label = std::string("dtmin") + std::to_string(k);
    new (&dtmins(k)) Kokkos::View<double, Kokkos::CudaUVMSpace>
        (Kokkos::view_alloc(label, Kokkos::WithoutInitializing));
}

auto instances = Kokkos::Experimental::partition_space(Kokkos::Cuda(), std::vector<int>(numOuter, 1));
int idx = 0;
for (int dir = 0; dir < num_dim; ++dir) {
    for (int i = 0; i < face_num[dir]; ++i) {
        Kokkos::parallel_reduce("max of sample columns",
            Kokkos::RangePolicy<Kokkos::Cuda, int>(instances[idx], 0, 256),
            KOKKOS_LAMBDA (int k, double &tmax) {tmax = sample(i, dir, k);},
            Kokkos::Min<double, Kokkos::CudaSpace>(dtmins(idx)));
        ++idx;
    }
}

Figure 7: View of Views Test Code

num_dim = 3;
double dtmin = 0;
Kokkos::parallel_reduce("max of sample columns with multidim iteration",
    Kokkos::MDRangePolicy<Kokkos::Cuda, Kokkos::Rank<3>>({0,0,0}, {num_dim-1, face_num_max-1, 256}),
    KOKKOS_LAMBDA (int dir, int face, int i, double &tmax) {
        if (dir==0 and face>face_num[0]) {
            tmax=0.0; // in these cases, we make tmax=0.0 because that will not contribute to the max.
        } else if (dir==1 and face>face_num[1]) {
            tmax=0.0; // in these cases, we make tmax=0.0 because that will not contribute to the max.
        } else if (dir==2 and face>face_num[2]) {
            tmax=0.0; // in these cases, we make tmax=0.0 because that will not contribute to the max.
        } else {
            tmax=sample(face, dir, i);
        },
    Kokkos::Max<double, Kokkos::CudaSpace>(dtmin));

Figure 8: MDRangePolicy Test Code

4 Parallelization Approaches

4.1 Transform Nested Loops into an MDRangePolicy

A Kokkos MDRangePolicy is an execution policy that enables iteration over a multi-dimensional array, where the iteration and resulting calculations can be performed on the GPU. When testing code, this was two of the three methods explored that yielded positive results. However, this approach carries the unique constraint that there must not be any dependencies between the dimensions of the array which are being iterated over.

As seen in Figure 10, the calculations iterate over the variables dir, i, and index. However, on any given run, the range of the variable i depends on the value of dir, and this dependence cannot be declared when using an MDRangePolicy. To get around this issue, we make the dimensions as large as possible to include all relevant points in the iteration, and then filter out the extraneous points with targeted if statements. An example of this can be seen in the MDRangePolicy test
num_dim = 3
Kokkos::parallel_for("iterate over faces and directions",
Kokkos::MDRangePolicy<Kokkos::Cuda, Rank<2>>({0,0},{num_dim-1,face_num_max-1}),
KOKKOS_LAMBDA (int dir, int face) {
  if (dir=0 and face>face_num[0]) {
    continue;
  } else if (dir=1 and face>face_num[1]) {
    continue;
  } else if (dir=2 and face>face_num[2]) {
    continue
  } else {
    Kokkos::parallel_reduce("parallel max of sample columns",
    Kokkos::RangePolicy<Kokkos::Cuda, int>({0,256}),
    KOKKOS_LAMBDA (int k, double &tmax) {tmax = sample(face,dir,k);},
    Kokkos::Min<double, Kokkos::CudaSpace>(dtmins(idx)));
  }
})

Figure 9: Hierarchical Parallelism Test Code

code in Figure 8. The final implementation of the getFluxesKokkos makes use of this filtering as well.

4.2 Hierarchical Parallelization with Teams of Threads
Hierarchical parallelization addresses the limitation when there aren’t enough values in a specific dimension (ex. only 5 rows, only 7 faces, etc.) to saturate the GPU. Suppose that the number of rows is the limiting factor suppressing GPU utilization. Using thread teams, which are collections of threads that are guaranteed to execute concurrently and can synchronize, we can enable the GPU to work more efficiently. At a high-level, the strategy works as follows:

1. Do one parallel launch of N teams
2. Each team handles one row
3. The threads within each team performs a reduction
4. The thread teams perform a reduction

Note, to use thread teams requires a policy change to TeamPolicy. Additionally, functor operations receive a team member. We will use the TeamThreadRange policy in the inner loop to execute inner reductions. Using these principles, we wrote the following code:
for (int dir=0; dir < params->numdim; dir++)
{
  for (int i=0; i<params->face_num[dir]; ++i)
  {
    const int nlo = mesh_faces_face_lo(i,dir)-1;
    const int nhi = mesh_faces_face_hi(i,dir);
    if (mesh_faces_face_id(i,dir) > 2)
    {
      Kokkos::parallel_reduce("get_face_fluxes_internal", nhi-nlo,
        KOKKOS_LAMBDA(const int index, int& ierr_reduce, double& dtmin_reduce)
        { ... }, Kokkos::Max<int>(+ierr), Kokkos::Min<double>(dtmin))
    }
    else if (mesh_faces_face_id(i,dir) == 1)
    {
      Kokkos::parallel_reduce("get_face_fluxes_LO_side", nhi-nlo,
        KOKKOS_LAMBDA(const int index, int& ierr_reduce)
        { ... }, Kokkos::Max<int>(+ierr), Kokkos::Min<double>(dtmin))
    }
    else if (mesh_faces_face_id(i,dir) == 2)
    {
      Kokkos::parallel_reduce("get_face_fluxes_HI_side", nhi-nlo,
        KOKKOS_LAMBDA(const int index, int& ierr_reduce, double& dtmin_reduce)
        { ... }, Kokkos::Max<int>(+ierr), Kokkos::Min<double>(dtmin))
    }
  }
}

Figure 10: getFluxesKokkos Current Kernel

Kokkos::parallel_reduce("get_face_fluxes_MDRange",
  Kokkos::MDRange<Kokkos::Rank<3>, Kokkos::DefaultExecutionSpace>({0,0,0},
  {params->numdim, face_num_max, mesh_faces_face_diff_max}),
  KOKKOS_LAMBDA(int dir, int face, int k, int &ierr_reduce, double &dtmin_reduce) {
    if (face < params->face_num[dir]) {
      ...
    if (k < nhi-nlo) {
      ...
    if (mesh_faces_face_id(face, dir) > 2) {
      ...
    } else if (mesh_faces_face_id(face, dir) == 1) {
      ...
    } else if (mesh_faces_face_id(face, dir) == 2) {
      ...
    }
  }, Kokkos::Max<int, Kokkos::DefaultExecutionSpace>(+ierr),
  Kokkos::Min<double, Kokkos::DefaultExecutionSpace>(dtmin))

Figure 11: getFluxesKokkos MDRangePolicy Implementation
```cpp
struct stat {
  double err;
  double dt;
};

template <class Scalar, class Space>
struct MinMaxData {
  public:
    using reducer = MinMaxData<Scalar, Space>;
    using value_type = std::remove_cv_t<Scalar>;
    static_assert(!std::is_pointer<value_type>::value && !std::is_array<value_type>::value);
    using result_view_type = Kokkos::View<value_type, Space>;
  private:
    result_view_type value;
  public:
    KOKKOS_INLINE_FUNCTION
    MinMaxData(value_type& value_) : value(&value_) {}
    KOKKOS_INLINE_FUNCTION
    MinMaxData(const result_view_type& value_) : value(value_) {}
    KOKKOS_INLINE_FUNCTION
    void join(value_type& dest, const value_type& src) const {
      dest.err = std::max(dest.err, src.err);
      dest.dt = std::min(dest.dt, src.dt);
    }
    KOKKOS_INLINE_FUNCTION
    void init(value_type& val) const {
      val.err = Kokkos::reduction_identity<std::remove_reference_t<decltype(val.err)>>::max();
      val.dt = Kokkos::reduction_identity<std::remove_reference_t<decltype(val.dt)>>::min();
    }
    KOKKOS_INLINE_FUNCTION
    value_type& reference() const { return *value.data(); }
    KOKKOS_INLINE_FUNCTION
    result_view_type view() const { return value; }
};

Figure 12: Custom Reducer for Hierarchical Parallelism
```
```cpp
stat outer_data;
Kokkos::parallel_reduce(Kokkos::TeamPolicy<Kokkos::DefaultExecutionSpace>(params->numdim, 4, 32),
KOKKOS_LAMBDA(Kokkos::TeamPolicy<Kokkos::DefaultExecutionSpace>::member_type member,
stat& outer_data_reduce)
{
    int dir = member.league_rank();
    stat mid_data;
    Kokkos::parallel_reduce(Kokkos::TeamThreadRange(member, params->face_num[dir]),
        [=] (int face, stat& mid_data_reduce)
        {
            ...
            stat inner_data;
            Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(member, nhi-nlo),
                [=] (const int index, stat& inner_data_reduce)
                {
                    ...
                    if (mesh_faces_face_id(face,dir) > 2)
                    {
                        ...
                    }
                    else if (mesh_faces_face_id(face,dir) == 1)
                    {
                        ...
                    }
                    else if (mesh_faces_face_id(face,dir) == 2)
                    {
                        ...
                    }
                }, MinMaxData<stat>(inner_data));
            mid_data_reduce = inner_data;
        }, MinMaxData<stat>(mid_data))
    outer_data_reduce = mid_data;
};

dtmin = outer_data.dt;
*ierr = outer_data.err;
```

Figure 13: getFluxesKokkos Hierarchical Parallelism Implementation

The custom reducer in Figure 12 enables us to perform multiple reductions concurrently. In particular, this reducer finds the max for the double `err` and the min for the double `dt`.

As seen in Figure 13, we implemented hierarchical parallelism structure using a custom reducer to conduct multiple (i.e. two) reductions across the search space.

## 5 Results

Here are the runtimes for the original master branch of xRAGE, the MDRange version, and the hierarchically parallel version.

**Snow:** 36 ranks
- Master: 12.468s
- MDRange: 12.416s
- Hierarchical: 13.185s

**RZAnsel:** 4 ranks (MPS Disabled)
- Master: 48 minutes 8 seconds
• MDRange: 1 minute 40 seconds (29x faster)
• Hierarchical: 1 minute 23 seconds (35x faster)

RZAnsel: 20 ranks (MPS Enabled)
• Master: 16 minutes 28 seconds
• MDRange: 2 min 25 s (7x faster)
• Hierarchical: 43.090s (23x faster)

We observed zero speedups on Snow. This result is expected since Snow does not have GPUs for our parallel code to utilize. On RZAnsel machines, we see notable speedups with both approaches, with hierarchical parallelism performance exceeding MDRange to a moderate extent. RZAnsel machines have 4 GPUs per node that have a peak performance of 7 TFLOP/s with double precision, which explains why we see significant speedups. We could only run 20 ranks on RZAnsel because of the memory required for each MPS instance on the GPU, causing insufficient memory resources for xRAGE.

It is worth noting that running on Snow remains faster than running on other computing clusters, such as RZAnsel, which have GPUs. This result is likely due to the time penalties of moving memory between the CPU and GPU on GPU-enabled platforms.

6 Conclusion

Both methods of parallelization demonstrate significant speedups, on the order of 35x and 29x times for hierarchical and MDRange, respectively. In the future, these methods may be combined for additional optimization. This change would require Kokkos 4.1 to which the XCP-2 group may eventually upgrade. Judging from our results, we successfully achieved our objective to profile and optimize xRAGE utilizing two parallelization techniques.