Mathematics Clinic

Final Report for
Sandia National Laboratories

Parallelizing Intrepid Tensor Contractions Using Kokkos

April 6, 2015

Team Members

Brett Collins (Project Manager)
Alex Gruver
Ellen Hui
Tyler Marklyn

Advisor

Jeff Amelang

Liaison

H. Carter Edwards

Abstract

Your abstract should be a *brief* summary of the contents of your report. Don't go into excruciating detail here—there's plenty of room for that later.

If possible, limit your abstract to a single paragraph, as your abstract may be used in promotional materials for the Clinic.

Contents

List of Figures

List of Tables

Acknowledgments

Chapter 1

Sandia National Laboratories

1.1 Background

Sandia National Laboratories is a federally funded research and development center owned and managed by the Sandia Corporation. The laboratory's primary focus is the maintenance, management, and development of the United States' nuclear arsenal. Sandia also performs research in the fields of supercomputing and scientific computing.

1.2 CPU vs GPU

The vast majority of scientific software produced and used by the national labs relies on message passing parallelism (MPI) to leverage both inter-node and intra-node parallelism. Until the present, this approach of using MPI across a single node sacrificed some performance for the ease of a monolithic programming model, and the performance penalty has not been high enough to motivate the usage of threads instead of processes. However, as the exascale push hits the power wall, interest has been growing in the area of using higher performing and less power-hungry co-processors on each node, still with message passing across nodes. Unfortunately, this means that much existing code will have to be rewritten, as message passing cannot be used to leverage the parallelism of co-processors such as Graphical Processing Units (GPUs).

GPUs are characterized by high thread counts, decreased memory per thread, and relatively small instruction sets when compared to CPUs. These differences lead to many advantages when it comes to high performance computing. Since GPUs have a smaller instruction set, they can devote

more of their transistors to arithmetic computation. This means that GPUs are capable of executing significantly more floating point operations per second (FLOPS). Additionally, GPUs use less power than CPUs, which makes them appealing for supercomputers, where power consumption is a major concern.

Well implemented GPU code can yield significant speedup in certain processing-heavy applications. However, the ways in which an algorithm must be optimized to run on a GPU are highly dependent on hardware architecture. Additionally, the language used to write code that runs on different hardware might actually differ. Finally, highly parallel code may suffer from race conditions caused by unsafe data accesses. Often, using always safe data structures and memory access patterns can reduce performance, but performance is at the heart of what Sandia, and others in the supercomputing industry are interested in.

1.2.1 WHAT ELSE GOES HERE?

I don't now how in depth you want me to get into GPU architecture. Here is a list of questions I came up with while writing this section:

- Who is my audience?
- Should I go more in depth?
- Should I talk about caches, and how that's different from CPU to GPU? (i.e. coalesced memory accesses)
- Should there be code snippets here? It's the introduction, so my feeling was no, but we might need more later then.
- Should I mention shared memory, threads, warps, etc.?

1.3 Kokkos

The Trilinos Project, developed and maintained by Sandia, is a collection of open source libraries intended for use in large-scale scientific applications. One of the packages in Trilinos is Kokkos, which is designed to aid portability and performance in software written for manycore architectures.

Kokkos attempts to mitigate the issues of architecture dependence by allowing programmers to write their code once, and compile for optimization on a variety of manycore architectures. This is possible because the

library helps manage the allocation of memory across devices for the programmer. Kokkos also allows users to write thread scalable software by providing an API for using fast, non-locking data structures.

However, the Kokkos package is still new and relatively untested. No large-scale projects have yet been written with it. Some small kernels have been rewritten using Kokkos, but there are still many kernels in Trilinos that would benefit from increased thread scalable parallelization.

1.4 Problem

The goal of this clinic has been to rewrite several kernels, using Kokkos, from libraries within Trilinos to be efficient and thread-scalable on many-core architectures. These redesigned and reimplemented kernels will be integrated into Trilinos' production code, as both a performance improvement and as a use case for Kokkos.

We focused primarily on a tensor manipulation library within Intrepid, a sub-package of Trilinos. The Intrepid package is a library of kernels designed for use by developers who want to reuse large parts of their existing code frameworks while gaining access to state of the art tools for compatible discretizations of partial differential equations. As such, by improving the performance of these kernels within Intrepid, we can improve performance of any number of calculation-heavy simulation libraries that rely on Intrepid to calculate tensor contractions.

Chapter 2

Intrepid

Intrepid is a C++ library developed as part of Sandia's Trilinos Project, providing algebraic operations over multi-dimensional arrays. Tensor contractions are one class of tools implemented by Intrepid, widely used in high-performance simulation software.

One particular type of tensor contraction, two-dimensional matrix multiplication, is known to show great speedup when implemented using CPU and especially GPU parallelism. For this reason among others, the team considered Intrepid tensor contractions to have good theoretical potential to derive significant performance gains from parallelism.

2.1 Tensor Contractions

A tensor can be thought of as a multidimensional array. Tensor contractions are algebraic operations over tensors in which pairs of indices, one from each of the two input tensors, are "contracted" together, reducing the dimensionality of the output tensor.

For instance, two matrices (two-dimensional tensors), can have no indices contracted away, yielding a four-dimensional outer product. If the size of one dimension on the left matrix matches the size of one dimension on the right matrix, that dimension can be contracted by summing the products of the left and right matrix entries for each element in the contraction index. This is standard matrix multiplication. If the two matrices are of identical size, then both indices can be contracted away in an inner product, each element in the left matrix multiplied pairwise with the corresponding element int the right matrix and summed to yield a scalar output.

Tensor contractions can be generalized to higher-dimension tensors, such as three or four-dimensional tensors.

2.2 Intrepid Contractions Overview

Intrepid provides nine types of tensor contraction, differing by input dimensionality, number of indices contracted away, and output dimensionality. It is most simple to classify Intrepid's tensor contractions by number of indices contracted away and output dimensionality.

Intrepid tensor contractions contract away one, two, or three indices. The output of a single contraction can be a scalar (zero-dimensional), a vector (one-dimensional), or a matrix (two-dimensional). Each combination of number of contraction indices and output dimension is handled by one Intrepid tensor contraction kernel.

Kernel Name	Left Input	Right Input	Output	Contraction Indices
ContractDataDataScalar	1D	1D	Scalar	One
ContractDataDataVector	2D	2D	Scalar	Two
ContractDataDataTensor	3D	3D	Scalar	Three
ContractDataFieldScalar	2D	1D	Vector	One
ContractDataFieldVector	3D	2D	Vector	Two
ContractDataFieldTensor	4D	3D	Vector	Three
ContractFieldFieldScalar	2D	2D	Matrix	One
ContractFieldFieldVector	3D	3D	Matrix	Two
ContractFieldFieldTensor	4D	4D	Matrix	Three

Table 2.1 Summary of the nine Intrepid tensor contraction kernels

In order to more easily discuss specific tensor contraction kernels in Intrepid, it helps to first understand the naming convention used for the kernel names. Each kernel's name contains two suffixes, where the first indicates the dimensionality of the output and the second indicates the number of contraction indices.

For instance, the first suffix `DataData` is used for kernels that produce scalar outputs, and the second suffix `Scalar` is used for kernels that contract away one dimension. Therefore, the Intrepid kernel `ContractDataDataScalar` produces scalar outputs and contracts away one dimension, and so by necessity the inputs for a single contraction must be vectors. All of the Intrepid tensor contraction kernel suffixes are summarized in Table 2.2. The nine tensor contractions in Intrepid are summarized in Table 2.1.

String	Position	Meaning
DataData	First Suffix	Scalar Output
DataField	First Suffix	Vector Output
FieldField	First Suffix	Matrix Output
Scalar	Second Suffix	One Contraction Index
Vector	Second Suffix	Two Contraction Indices
Tensor	Second Suffix	Three Contraction Indices

Table 2.2 Intrepid tensor contraction suffixes

Note that the tensor contraction kernels in Intrepid each actually performs many contractions; for instance, `ContractDataDataScalar`, which performs contractions of two input vectors to a single output scalar (dot product), actually calculates an array of dot products. That is, the inputs are both arrays of vectors, and the output is an array of scalars. This is represented in the code using a dummy index, which we call the `Cell` index.

2.3 ContractDataDataScalar

`ContractDataDataScalar` is the simplest tensor contraction in Intrepid. The kernel takes two arrays of vectors and outputs an array of scalars. A snippet showing the simple serial implementation of `ContractDataDataScalar` can be seen in Figure 2.1.

```
1    for (int c = 0; c < numCells; ++c) {
       for (int qp = 0; qp < quadraturePoints; ++qp) {
3        output[c] += leftInput[c][qp] * rightInput[c][qp];
       }
5    }
```

Figure 2.1 Code from serial `ContractDataDataScalar`

As shown in Figure 2.1, this kernel contracts away the `Quadrature Points` dimension, leaving only the `Cell` dimension. It can also be thought of as an array of dot products.

2.4 ContractDataDataVector

`ContractDataDataVector` takes two arrays of two-dimensional tensors (matrices) and computes an array of their inner products.

```
1     for (int c = 0; c < numCells; ++c) {
        for (int qp = 0; qp < quadraturePoints; ++qp) {
3         for (int t = 0; t < iVec; ++t) {
            output[c] += leftInput[c][qp][t] * rightInput[c][qp][t];
5         }
        }
7     }
```

Figure 2.2 Code from serial `ContractDataDataVector`

As shown in Figure 2.2, `ContractDataDataVector` is very similar to `ContractDataDataScalar`, except it contracts two indices instead of one.

2.5 ContractDataDataTensor

`ContractDataDataTensor` takes two arrays of three-dimensional tensors and computes an array of their inner products.

```
1     for (int c = 0; c < numCells; ++c) {
        for (int qp = 0; qp < quadraturePoints; ++qp) {
3         for (int t1 = 0; t1 < iVec1; ++t1) {
            for (int t2 = 0; t2 < iVec2; ++t2) {
5             output[c] += leftInput[c][qp][t1][t2] *
                           rightInput[c][qp][t1][t2];
7         }
        }
9     }
    }
11
```

Figure 2.3 Code from serial `ContractDataDataTensor`

As shown in Figure 2.3, `ContractDataDataTensor` is very similar to

`ContractDataDataVector` and `ContractDataDataScalar`, but has three contraction indices.

2.6 ContractDataFieldScalar

`ContractDataFieldScalar` takes an array of matrices and an array of vectors and contracts away one index.

```
    for (int c = 0; c < numcells; ++c) {
      for (int l = 0; l < lbf; ++l) {
        for (int qp = 0; qp < quadraturepoints; ++qp) {
          output[c][l] += left[c][l][qp] * right[c][qp];
        }
      }
    }

```

Figure 2.4 Code from serial `ContractDataFieldScalar`

As shown in Figure 2.4, `ContractDataFieldScalar` has two non-contraction indices, the `Left Basis Function` index and the dummy `Cell` index. The output for this contraction is therefore an array of vectors instead of an array of scalars.

2.7 ContractDataFieldVector

`ContractDataFieldVector` takes an array of three-dimensional tensors and an array of vectors, and contracts away two indices to produce an array of vectors.

As shown in Figure 2.5, `ContractDataFieldVector` is similar to `ContractDataFieldScalar`, but has two contraction indices.

2.8 ContractDataFieldTensor

`ContractDataFieldTensor` takes an array of four-dimensional tensors and an array of three-dimensional tensors, and contracts away two indices to produce an array of vectors.

```
for (int c = 0; c < numcells; ++c) {
  for (int l = 0; l < lbf; ++l) {
    for (int qp = 0; qp < quadraturepoints; ++qp) {
      for (int t = 0; t < iVec; ++t) {
        output[c][l] += left[c][l][qp][t] * right[c][qp][t];
      }
    }
  }
}
```

Figure 2.5 Code from serial `ContractDataFieldVector`

As shown in Figure 2.6, `ContractDataFieldTensor` is similar to `ContractDataFieldScalar`, but has three contraction indices.

2.9 ContractFieldFieldScalar

`ContractFieldFieldScalar` takes in two arrays of matrices and performs matrix multiplication on each element, yielding an output array of matrices.

As shown in Figure 2.7, `ContractFieldFieldScalar` has two non-contraction indices, so the output is an array of matrices.

2.10 ContractFieldFieldVector

`ContractFieldFieldVector` takes two arrays of three-dimensional tensors and contracts away two indices, keeping the `Cell` dummy dimension as well as the left and right basis function indices.

As shown in Figure 2.8, `ContractFieldFieldVector` is similar to `ContractFieldFieldScalar` but with two contraction indices.

2.11 ContractFieldFieldTensor

`ContractFieldFieldTensor` is the most complex of the tensor contraction kernels in Intrepid. This kernel takes two four-dimensional tensors and and contracts away three indices, keeping the `Cell` dummy dimension as well as the left and right basis function indices.

```
1     for (int c = 0; c < numcells; ++c) {
        for (int l = 0; l < lbf; ++l) {
3         for (int qp = 0; qp < quadraturepoints; ++qp) {
            for (int t1 = 0; t1 < iVec1; ++t1) {
5             for (int t2 = 0; t2 < iVec2; ++t2) {
                output[c][l] += left[c][l][qp][t1][t2] * right[c][qp
    ][t1][t2];
7             }
            }
9         }
        }
11    }
```

Figure 2.6 Code from serial `ContractDataFieldTensor`

As shown in Figure 2.9, `ContractFieldFieldTensor` is similar to `ContractFieldFieldScalar`, but with two contraction indices.

```
1     for (int c = 0; c < numcells; ++c) {
        for (int l = 0; l < lbf; ++l) {
3         for (int r = 0; r < rbf; ++r) {
            for (int qp = 0; qp < quadraturepoints; ++qp) {
5             output[c][l][r] += left[c][l][qp] * right[c][r][qp];
            }
7         }
        }
9     }
```

Figure 2.7 Code from serial `ContractFieldFieldScalar`

```
1     for (int c = 0; c < numcells; ++c) {
        for (int l = 0; l < lbf; ++l) {
3         for (int r = 0; r < rbf; ++r) {
            for (int qp = 0; qp < quadraturepoints; ++qp) {
5             for (int t = 0; t < iVec; ++t) {
                output[c][l][r] += left[c][l][qp][t] * right[c][r][
    qp][t];
7             }
            }
9         }
        }
11    }
```

Figure 2.8 Code from serial `ContractFieldFieldVector`

```
1     for (int c = 0; c < numcells; ++c) {
        for (int l = 0; l < lbf; ++l) {
3         for (int r = 0; r < rbf; ++r) {
            for (int qp = 0; qp < quadraturepoints; ++qp) {
5             for (int t1 = 0; t1 < iVec1; ++t1) {
                for (int t2 = 0; t2 < iVec2; ++t2) {
7                   output[c][l][r] += left[c][l][r][qp][t1][t2] *
                    right[c][l][r][qp][t1][t2];
9               }
              }
11          }
          }
13      }
      }
15
```

Figure 2.9 Code from serial `ContractFieldFieldTensor`

Chapter 3

Parallelism

3.1 Flat Parallelism

3.2 Reduction

As shown in Figure in some cases, flat parallelism can perform very poorly. One problem with flat parallelism is the potential lack of enough parallelism, as in `ContractDataDataTensor`. `ContractDataDataTensor` takes two input arrays of three-dimensional tensors and produces an array of scalars. See Section 2.5 for a more complete description of this kernel.

The problem with the `DataData` class of tensor contractions (see Table 2.2 is that they all output an array of scalars – that is, each individual contraction produces a scalar output. Therefore, using flat parallelism, the greatest number of threads we can spawn is one thread per contraction. Each thread must then perform an entire contraction independently, which in the case of `ContractDataDataTensor`, means looping over all three of the contraction indices.

Because of this, we see that when the contraction size is large and the memory size is small – when we cannot spawn enough threads to saturate the GPU and each thread is responsible for a large amount of computation – `ContractDataDataTensor` actually performs worse than serial.

A solution to this problem is to use a parallel reduction algorithm instead of a flat parallelism algorithm. In a reduction, multiple threads contribute to a single output element. This adds the necessary overhead of coordinating between threads and combining their contributions.

In Kokkos, threads can be organized into teams, which correspond to Cuda blocks. Built-in reduction methods allow teams to merge the contri-

butions of its constituent threads.

Using this team-thread paradigm, we explored several methods of implementing `ContractDataDataTensor` using a reduction algorithm.

3.2.1 Team Depth 1

In this reduction algorithm, we assign one team per contraction, and each team has as many threads as there are elements in the `iVec2` dimension. Each thread therefore performs `numPoints` $\times$ `iVec1` multiplications, and then combines its local sum with that of the other threads in the team.

```
2     float sum = 0;
      float tsum = 0;
4
      for (int qp=0; qp < numPoints; ++qp) {
6       for (int t1=0; t1 < iVec1; ++t1) {
          sum +=  left(threadIndex, qp, t1, _t2) *
8           right(threadIndex, qp, t1, _t2);
        }
10    }

12    Kokkos::parallel_reduce(Kokkos::TeamThreadLoop(thread, iVec2),
          [&] (const unsigned int& dim, float& localsum) {
14        localsum += sum;
        }, tsum);
16
      _output(elementIndex) = tsum;
18
```

Figure 3.1 Code from Team Depth 1 reduction functor for `ContractDataDataTensor`

In Figure 3.1, we can see that each thread loops over the `numPoints` and `iVec1` dimensions, and then reduces with the other threads in the team. The call to Kokkos' `parallel_for` function, seen in Figure 3.2, specifies an execution policy in which the number of teams launched is `numCells`, and each team has `iVec1` threads.

As shown in Figure 3.3, this Team Depth 1 algorithm performs generally better than serial, but the speedup is minimal. Therefore, we explored

```
const team_policy reduction_policy(numCells, iVec2);
Kokkos::parallel_for(reduction_policy,
contractDataDataTensorTeamDepth1Functor );
Kokkos::fence();

```

Figure 3.2 Code from kernel launch for `ContractDataDataTensor` Team Depth 1 reduction

other reduction algorithms, which we hoped would yield more impressive results.

3.2.2 Team Depth 2

This reduction algorithm is similar to the previous one. As with the Team Depth 1 reduction, we assign one team per contraction. Each team has `iVec1` $\times$ `iVec2` threads, each responsible for `numPoints` multiplications. Each thread then combines its local sum with that of the other threads in the team.

In Figure 3.4, we can see that each thread loops over the `numPoints` dimension only, and then reduces with the other threads in the team. The call to Kokkos' `parallel_for` function, seen in Figure 3.5, specifies an execution policy in which the number of teams launched is `numCells`, and each team has `iVec1` $\times$ `iVec2` threads.

As shown in Figure 3.6, this Team Depth 2 algorithm always performs better than serial. In addition, the speedup is significant, and for large memory sizes, performs nearly as well as the flat parallel algorithm. Given the necessary overhead of performing a reduction, we believe this to be a fairly good algorithm for good performance across the board.

However, because the team size is fixed based on the size of the `iVec1` and `iVec2` dimensions, this algorithm may suffer performance penalties or perhaps even bugs if these two dimensions are of unexpected sizes. A more generalizable algorithm therefore would be preferable.

3.2.3 Teamstride

A more robust algorithm is one we call the Teamstride algorithm, in which each contraction is still assigned to a team, but a fixed number of threads

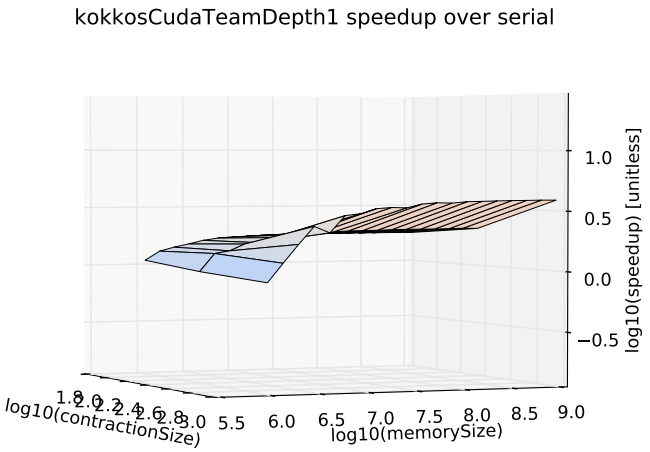


Figure 3.3 Speedup of ContractDataDataTensor with Team Depth 1 algorithm over serial

are spawned for each team. These threads treat the three contraction indices (`numPoints`, `iVec1`, `iVec2`) as if they were a single index, each thread striding forwards by the number of threads on this "combined" index. For instance, if this algorithm were run with a team size of sixty-four threads, then the zeroth thread in a team would sum the product of the zeroth elements, the sixty-fourth, the 128th, and so on.

As seen in Figure 3.7, this algorithm requires more arithmetic on the part of each thread, and the thread is not responsible for looping over a single subset of indices, instead looping over all three contraction indices. The corresponding call in Figure 3.8 uses a fixed team size of 32, unlike the previous two algorithms.

As shown in Figure 3.9, this algorithm always performs better than serial. Like with Team Depth 2, the speedup is significant, and for large memory sizes, performs nearly as well as the flat parallel algorithm. In addition, this algorithm is more robust than Team Depth 2, since the number of threads per team is not determined by the size of the input dimensions. Therefore, the Teamstride reduction algorithm is likely the most generalizable and performant variant of the reduction algorithms, and should be

```
1
    float sum = 0;
3   float tsum = 0;

5   for (int qp=0; qp < numPoints; ++qp) {
        sum +=  left(threadIndex, qp, _t1, _t2) *
7         right(threadIndex, qp, _t1, _t2);
    }
9
    Kokkos::parallel_reduce(Kokkos::TeamThreadLoop(thread, iVec2*
    iVec1),
11      [&] (const unsigned int& dim, float& localsum) {
        localsum += sum;
13    }, tsum);

15  _output(elementIndex) = tsum;

17
```

Figure 3.4 Code from Team Depth 2 reduction functor for `ContractDataDataTensor`

explored in tensor contraction functions in which the inputs may consist of a few large tensor contractions.

3.3 Slicing

Another general method we used for these contractions was Slicing. The first step of this method was to load one full contraction from the left matrix into shared memory. Then, we simultaneously computed every output element that was dependent on that contraction as input. The clearest way to explain the algorithm is by example. Consider one of the matrix multiplications in ContractFieldFieldScalar.

On the left, we have the first of the two input matrixes, who's first row elements are labeled $A - E$. On the right we have the second of the two inputs. For the sake of simplicity, assume that we have on block of five threads which are labeled by color. Each of the threads reads in one of the elements on the right and copies it into shared memory. In cases where the number of elements per contraction (row on the left) is unequal to the number of contractions (columns on the right), we set the number of threads

```
const team_policy reduction_policy(numCells, iVec2 * iVec1);
Kokkos::parallel_for(reduction_policy,
contractDataDataTensorTeamDepth2Functor );
Kokkos::fence();

```

Figure 3.5 Code from kernel launch for `ContractDataDataTensor` Team Depth 2 reduction

per block equal to the number of contractions. This causes threads to either sit idle or loop multiple times when reading the elements on the left into shared memory, but this is clearly more efficient than forcing threads to compute more than one element.

After the values of the contraction have been read into shared memory, we have each thread compute the output element corresponding to one contraction. This is shown on the right by the colored columns. Each thread reads every element from shared memory and computes the contraction by multiplying these elements with the columns of the right matrix. We see that throughout this progress, memory accesses will be coalesced within the block, since each thread reads the same element from shared memory then multiplies by an element that is adjacent to the other elements the rest of the block is reading at that time.

For every other block of threads, the approach is similar, if Figure 3.1 represents the first block of the contraction, then the second block will be represented as below.

We see that for the FieldFieldScalar example, where our equation is given by $L_{C,\ell,P} \times R_{C,\mathcal{R},P} = O_{C,\ell,\mathcal{R}}$, the number of blocks initialized by the algorithm will be equal $\ell \times C$, since there are ℓ blocks per matrix, and we have C matrices. Additionally, there will be $\mathcal{R}$ threads per block.

Code for executing the algorithm as described above is included below, although it has been simplified for clarity.

The main advantage of this approach is that it is easily generalizable to tensor contractions of higher dimensions. Unlike tiling, which is significantly less intuitive in higher dimensions, it is easy to implement slicing in higher dimensions by loading a larger slice into shared memory. Because of its reliance on shared memory, there are many use cases in which we would expect slicing to perform poorly. Intuitively, slicing is reliant on large contraction sizes to produce speedup because in situations where the number

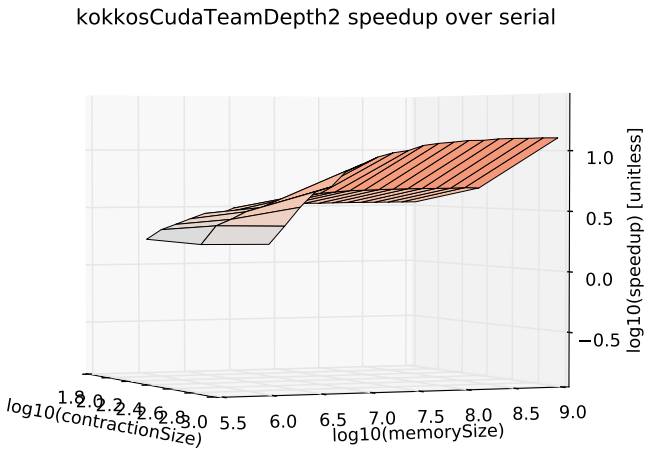


Figure 3.6 Speedup of ContractDataDataTensor with Team Depth 2 algorithm over serial

of threads per block is low it is unable to saturate the GPU. While this problem can be remedied by increasing the number of contractions per block, it can introduce problems with shared memory. Since shared memory is limited by nature, slicing has to balance the amount of work per block with the amount of shared memory available to that block.

In situations where the problem has an inherently large amount of reuse like ContractFieldFieldScalar, this problem can be remedied to some degree, but in contractions without this feature, like ContractDataDataScalar, it seems clear that slicing will not be an efficient algorithm.

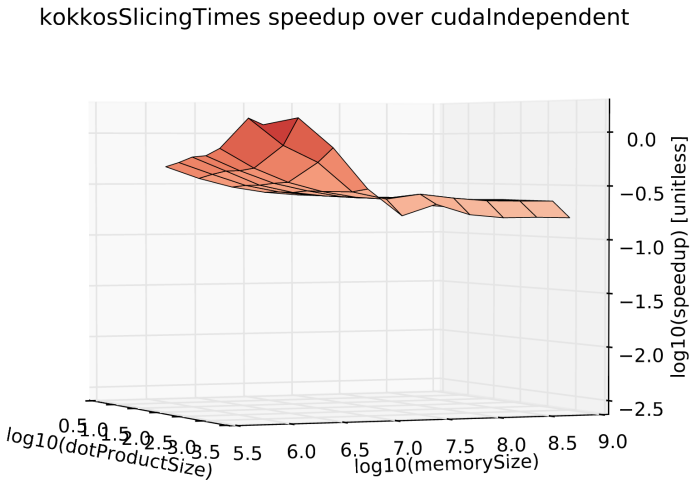
When we compare slicing approaches using one contraction per block to independent flat parallelism on promising problems, we get underwhelming results.

```
    for (int innerIdx = tIndex; innerIdx < cellSize; innerIdx += thread.team_size() ) {
      Sane arithmetic version:
      const unsigned int qp = innerIdx / (iVec1 * iVec2);
      const unsigned int t1 = (innerIdx \% (iVec1 * iVec2)) / iVec2;
      const unsigned int t2 = innerIdx \% iVec2;

      sum +=  _leftInput(c, qp, t1, t2) *
        _rightInput(c, qp, t1, t2);
    }

    Kokkos::parallel_reduce(Kokkos::TeamThreadLoop(thread, thread.team_size()),
        [&] (const unsigned int& dummy, float& localsum) {
        localsum += sum;
      }, tsum);

    _output(cell) = tsum;

```

Figure 3.7 Code from Teamstride functor for `ContractDataDataTensor`

```
    const team_policy reduction_policy(numCells, 32);
    Kokkos::parallel_for( reduction_policy, contractDataDataTensorTeamstrideFunctor );
    Kokkos::fence();

```

Figure 3.8 Code from kernel launch for `ContractDataDataTensor` Teamstride



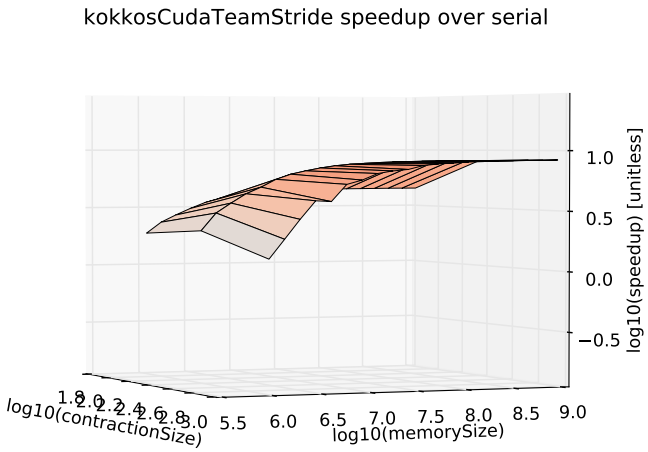


Figure 3.9 Speedup of ContractDataDataTensor with Teamstride algorithm over serial

These results were generated by comparing independent algorithms to slicing on ContractFieldFieldScalar with $\ell = \mathcal{R} = 10, P = 8 - 1024$. We see that in the corner where the memory size is small and contraction size is small we get a small amount of speedup relative to independent Cuda code, which is promising. This is the corner where we would expect slicing to perform the best in comparison to independent parallelism, since in this corner flat parallelism is unable to fully saturate the GPU. The benefits of reuse in this corner are significant enough to outcompete flat parallelism. On the rest of the graph, however, the inability of slicing to saturate the GPU means that it is significantly slower that flat parallelism. Since $\ell = \mathcal{R} = 10$, the algorithm naturally only spawns 10 threads per block, which is not enough to produce good results.

On problems with larger basis functions we see better results for slicing. For example, consider the following use case of ContractFieldFieldTensor: $\ell = \mathcal{R} = 125$, $P = 216$, $t_1 = t_2 = 3$.

the data exists for this here: https://github.com/Sandia2014/kokkos-

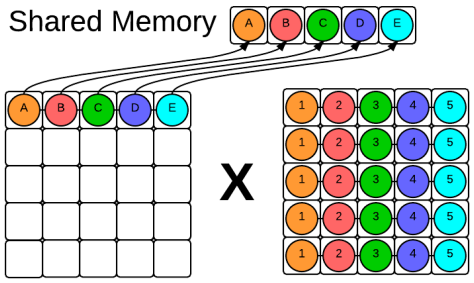


Figure 3.10 Demonstration of memory accesses for a slicing implementation of ContractFieldFieldScalar

intrepid/tree/slicing+tiling/ContractFieldFieldTensor, will make graph later

We see that while slicing performs better, it is still eclipsed by independent cuda.

We're still working on slicing that does two rows per block so hopefully that will be done soon and we'll have data for it.

3.4 Tiling

The final parallelization technique we used for these tensor contractions was tiling. This technique is similar to the tiled technique for matrix multiplication used in serial operations. Instead of relying on the cache to retain the relevant pieces of information, however, we use shared memory

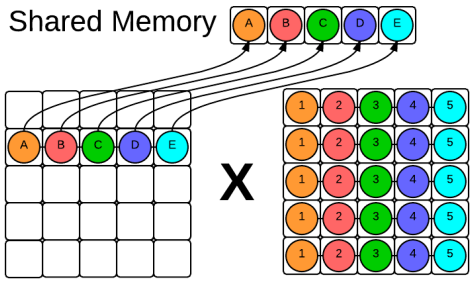


Figure 3.11 Demonstration of memory accesses for the second block of a slicing implementation of ContractFieldFieldScalar

```
1   extern __shared__ float sliceStorage[];
    const unsigned int col = threadIdx.x;
3   unsigned int currentBlock = blockIdx.x;
    unsigned int numBlocks = numBasis*numCells;
5
    syncthreads();
7   const unsigned int cell = currentBlock / numBasis;
    const unsigned int row = currentBlock - cell * numBasis;
9
    for (unsigned int p = threadIdx.x; p < contractionSize; p += blockDim.x) {
11     sliceStorage[p] = dev_contractionData_Left[cell*numBasis*contractionSize + row*
     contractionSize + p];
    }
13  syncthreads();

15  float sum = 0;
    for (int p = 0; p < contractionSize; ++p) {
17    sum += sliceStorage[p] * dev_contractionData_Right[cell*numBasis*contractionSize +
       p*numBasis + col];
19  }

21  dev_contractionResults[cell*numBasis*numBasis + row*numBasis + col] = sum;
```

Figure 3.12 Code from slicing algorithm on `ContractFieldFieldScalar`

to explicitly store the data we care about. Once again, we will explain this algorithm by example. Consider one of the matrix multiplications in ContractFieldFieldScalar shown above.

For the sake of simplicity we'll consider a block to be four threads, which simplifies our computation since the matrix is four by four. On the left hand side, the block loads a four element tile into the shared memory of the threads. Once these elements are loaded into memory, each thread can begin computation of their element in the output matrix. Each thread computes as much of their output element as they can using the elements in shared memory, then we load a new tile into shared memory and continue the process, as shown below. We see that in this case we will have to load two tiles into shared memory before we have computed every output element in its entirety.

Tiling can be viewed as a more specialized version of slicing, since they both use similar access patterns for shared memory. The difference between the two lies in tilings usage of multiple contractions per block, as well as the the distribution of a contractions operations over multiple loops of the routine. Because of these differences, tiling can routinely saturate the GPU in a way that pure slicing cannot, since the algorithm inherently limits the shared memory usage per block by reusing the same shared memory multiple times. Additionally, if we set the dimension of our tiles intelligently, we can reliably saturate the GPU with both blocks and threads, something

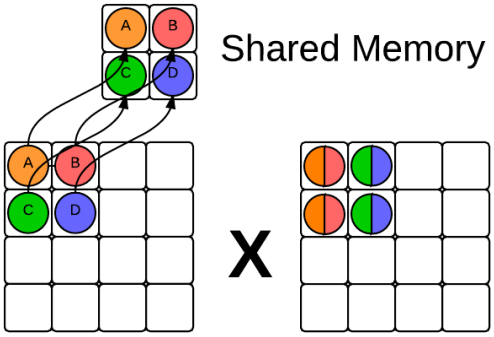


Figure 3.13 Demonstration of memory accesses for a tiling implementation of ContractFieldFieldScalar

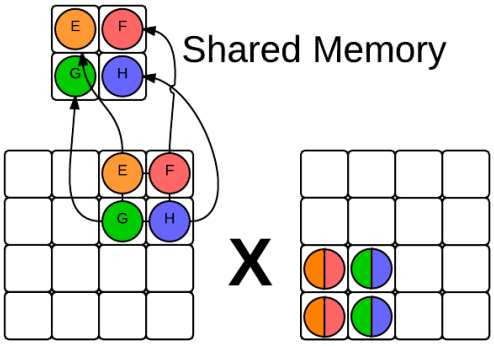


Figure 3.14 Demonstration of memory accesses for a tiling implementation of ContractFieldFieldScalar

that is very difficult to do adaptively with pure slicing.

Unfortunately, it is much less clear how exactly to tile in multiple dimensions. Unlike slicing, there seem to be multiple distinct ways of approaching the problem. One could create "tiles" with dimension equal to the contraction size, or any number less than the contraction dimension by unrolling the contraction to some intermediate degree. We haven't been able to fully explore every possibility in this area, and have simply treated the higher dimensional contractions as a fully unrolled contraction of one dimension. It is possible, however, that in some situations it would be more effective to create tiles with multiple degree. These tiles would have a different layout in memory who's efficiency would vary by situation.

Excerpts from our Cuda implementation of tiling are included below. The code assumes that tileSize (the horizontal and vertical dimensions of a tile) evenly divides both the contraction size and $\ell = \mathcal{R} =$ numBasis.

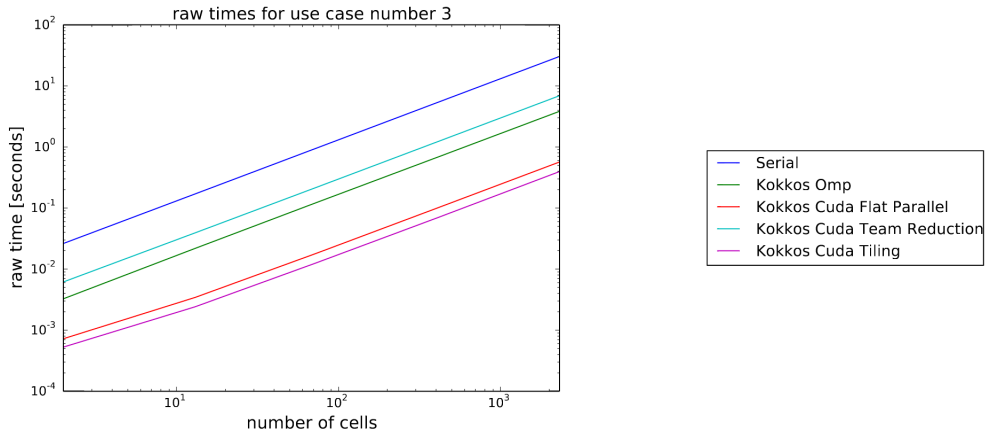
Thus far in our research, we have found tiling to be the most effective algorithm for realizing parallel speedup.

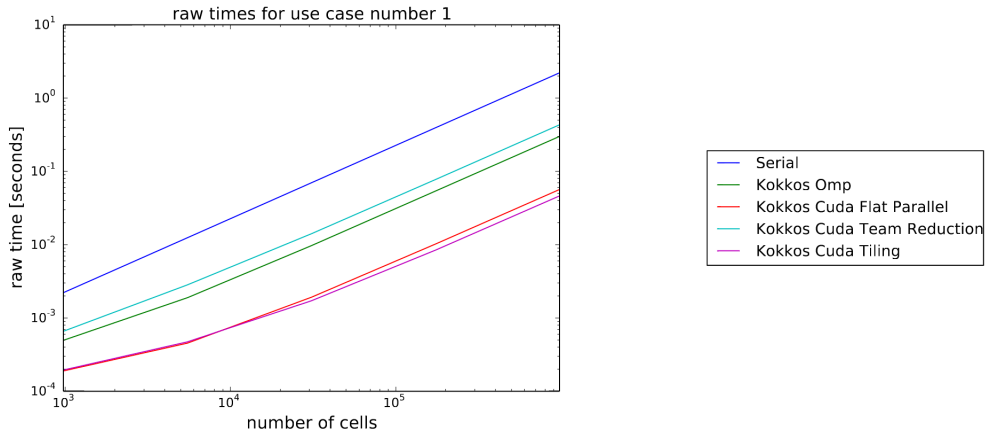
Consider the graph generated above for the following use case of ContractFieldFieldScalar, $\ell = \mathcal{R} = 125$, $P = 216$. We see that Tiling outperforms both flat parallelism and team reductions across the board. This trend continues for smaller use cases as well, as shown below when $\ell = \mathcal{R} = 8$, $P = 8$.

In general, we have found that 2D tiling is the most effective method for achieving parallel speedup on these kernels. While there may be some potential for exploration of higher dimensionality tiles, it seems doubtful that these layouts will be able to accomplish significantly more speedup.

```
extern __shared__ float tileStorage[];
const unsigned int numbersPerTile = tileSize * tileSize;
const unsigned int numberOfHorizontalTiles = contractionSize / tileSize;
const unsigned int numberOfVerticalTiles = numBasis / tileSize;

const unsigned int numberOfTiles = numCells * numberOfVerticalTiles * numberOfVerticalTiles;

const unsigned int subRow = threadIdx.x / tileSize;
const unsigned int subCol = threadIdx.x - subRow * tileSize;

unsigned int resultTileIndex = blockIdx.x;

unsigned int resultSubmatrixIndex = resultTileIndex % (numberOfVerticalTiles *
    numberOfVerticalTiles);
unsigned int resultMatrix = resultTileIndex / (numberOfVerticalTiles * numberOfVerticalTiles);

// for tileNumber in 0...numberOfTilesPerSide
for (unsigned int tileNumber = 0; tileNumber < numberOfHorizontalTiles; ++tileNumber) {

    // calculate result tile indices
    const unsigned int resultTileRow = resultSubmatrixIndex / numberOfHorizontalTiles;
    const unsigned int resultTileCol = resultSubmatrixIndex -
      resultTileRow * numberOfHorizontalTiles;

    // calculate this threads actual output index
    const unsigned int row = resultTileRow * tileSize + subRow;
    const unsigned int col = resultTileCol * tileSize + subCol;

    // these are base indices into the shared memory
    const unsigned int leftBaseIndex = subRow * tileSize;
    const unsigned int rightBaseIndex = numbersPerTile + subCol;

    const unsigned int resultIndex = row * numBasis + col;

    // load the left and right tiles into shared memory
    syncthreads();
    tileStorage[threadIdx.x] = dev_contractionData_Left[resultMatrix * numBasis * contractionSize
      + row * contractionSize + tileNumber * tileSize + subCol];
    tileStorage[threadIdx.x + blockDim.x] = dev_contractionData_Right[resultMatrix * numBasis *
    contractionSize
      + (tileNumber * tileSize + subRow) * numBasis + col];

    // make sure everyone's finished loading their pieces of the tiles
    syncthreads();
    double sum = 0;
    for (unsigned int dummy = 0; dummy < tileSize; ++dummy) {
      sum +=
        tileStorage[leftBaseIndex + dummy] *
        tileStorage[rightBaseIndex + dummy * tileSize];
    }
    dev_contractionResults[resultIndex] += sum;
  }
```

Figure 3.15 Code from tiling algorithm on `ContractFieldFieldScalar`


raw times for use case number 3
raw time [seconds]
number of cells
Serial
Kokkos Omp
Kokkos Cuda Flat Parallel
Kokkos Cuda Team Reduction
Kokkos Cuda Tiling


raw times for use case number 1
raw time [seconds]
number of cells
Serial
Kokkos Omp
Kokkos Cuda Flat Parallel
Kokkos Cuda Team Reduction
Kokkos Cuda Tiling

Chapter 4

Experience with Kokkos

4.1 Performance

A feature that many programmers consider when deciding what the best solution is to solve their problem, is performance. Since Kokkos uses Cuda and OpenMP as a backend we thought that it was important to do some testing to ensure that Kokkos performs as well as these two solutions. If Kokkos's performance was worse then Cuda's or OpenMP's then programmers would use these other solutions instead. The good news for Kokkos is that in our testing it performs almost identically to Cuda and OpenMP. We did not spend an extensive amount of time confirming our results due to project priorities, but after a few pieces of supporting data we assumed that the rest of the tests would give similar results because there is no information or evidence that should give us reason to believe this trend will change. The rest of this section will describe our strategy for testing performance of Kokkos versus Cuda and Kokkos versus OpenMP, present graphs showing the differences observed, and analyze the graphs.

The general method that was used to create performance data for Kokkos, Cuda, and OpenMP was to write algorithmically equivalent code for all three, make sure that the layout of the data is the same, then time the runtime of each (one after another). This process is pretty simple, but there is always noise in timing. That is why we repeated the same exact calculation five times and then use the average time. A couple things that should be noted are we are unsure how Kokkos does a reduction in the team_reduce() function, meaning we could not write a Cuda reduction that we knew was algorithm equivalent, and we can not be sure that the compilers do the same optimizations. Although we could have asked Dr. Carter Edwards

(our liaison and one of the creators of Kokkos), the project's priorities had changed and it was decided to not pursue this further. Regarding the second note, we tried to manually do some code optimizations that compilers can handle in order to make sure the amount of work each algorithm was the same (which is expected). Of course if one compiler has more advanced optimization techniques that is a benefit that should not be overlooked, but the goal of this testing was not to test performance against ease of coding, but rather the overall performance differences of Kokkos to Cuda and Kokkos to OpenMP.

Now we will look at some of the performance differences and similarities of Kokkos, Cuda, and OpenMP. Here is a graph that shows the raw times of Kokkos Cuda, Cuda, Kokkos OpenMP, and OpenMP for ContractDataDataScalar:

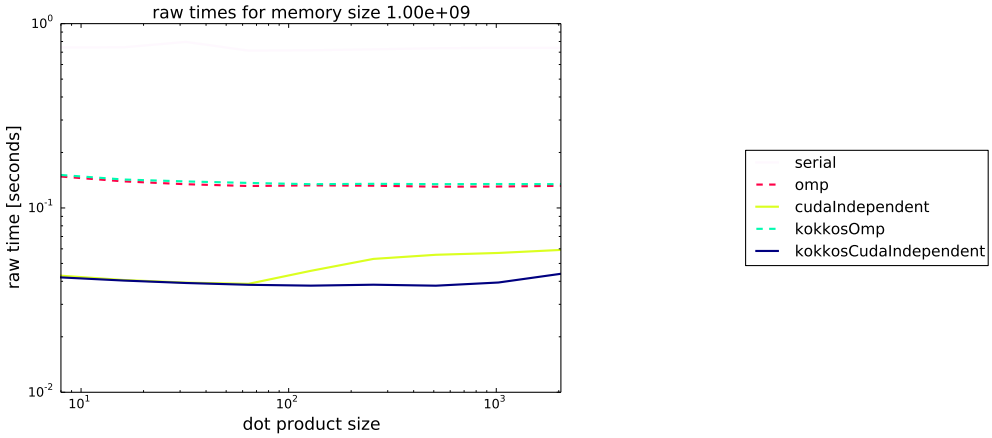


Figure 4.1 This graph plots the performance of Kokkos Cuda, Cuda, Kokkos OpenMP, and OpenMP for ContractDataDataScalar with a memory size of 1 GB. The y-axis is time in seconds, so closer to 0 is better. The x-axis plots different contraction sizes in order to compare multiple points.

Notice how in this graph Kokkos OpenMP and OpenMP are almost perfectly overlapping with Kokkos OpenMP. We are not quite sure why they are not perfectly overlapping, but it appears that it is not random noise because it is pretty consistent in this graph. However, the difference is so small it seems insignificant.

KokkosCuda versus Cuda, on the other hand, has some big differences. They are identical for the smaller problems but diverge a significant amount

for bigger problems. One of the theories that we have as to why this trend exists is that Kokkos may be launching a different amount of blocks than the number of blocks Cuda launches. We believe the reason this doesn't effect the smaller problem sizes is because the number of blocks that needs to be launched is smaller than the bigger problem sizes, so the upper limit of number of blocks launched is not reached, but clearly is in the bigger problem sizes.

Now here is a graph for ContractFieldFieldScalar that includes the slicing technique (which uses shared memory) for both Kokkos Cuda and Cuda and it includes the normal flat parallel algorithm for Kokkos Cuda and Cuda.

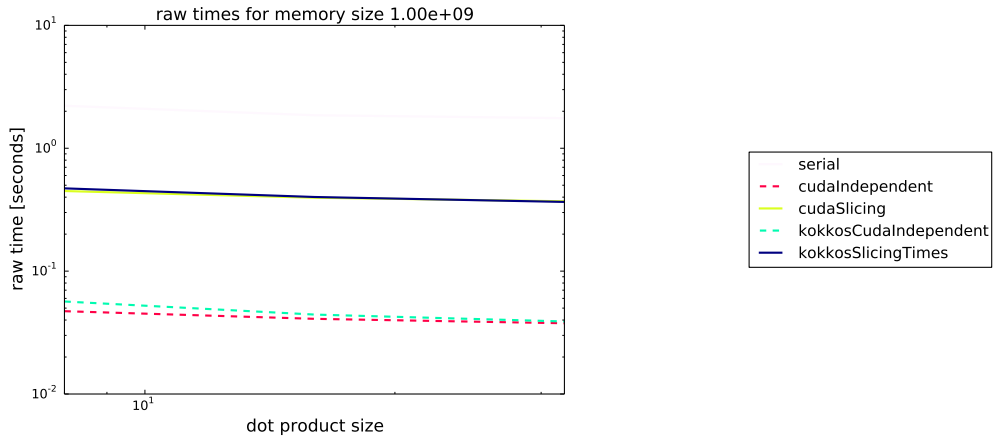


Figure 4.2 This graph shows performance differences (or similarities in our case) of the nested parallelism approach slicing.

In this graph it is clear that the Cuda slicing performance is almost identical to the Kokkos slicing performance. This is important to show that both Kokkos's and Cuda's use of shared memory results in the same performance.

Overall, after seeing a handful of graphs that show Kokkos performs almost identically to Cuda and to OpenMP, we accepted that Kokkos is not adding any unneeded overhead. As stated before, there may be slight differences due to compiler optimizations, but Kokkos seems to perform identically to the other multithreading solutions.

4.2 Snippets

Another major factor that plays into whether or not a programmer uses a certain language, feature, library, etcetera, is code complexity and ease of coding. Although this can be subjective, there are a couple differences between the language we would like to point out, especially regarding amount of code required compared to Cuda or OpenMP (although comparing to OpenMP may be unfair) and intuitiveness of the code (or readability).

Regarding the amount of code required, comparing Kokkos to OpenMP does not seem fair. Comparing OpenMP to almost anything seems unfair, because OpenMP requires very little code and most of the work is done for you. Although OpenMP only works on the CPU which is why it does not require the extra code that Kokkos requires. In general, OpenMP cannot be compared to Kokkos. If the programmer knows the code only needs to be multithreaded on the CPU and will never need more threads then we would strongly advise them to use OpenMP because it is very simple. However, that is not the niche that Kokkos is trying to fill.

Kokkos compared to Cuda however, requires similar amounts of code. Throughout the code comparison of Cuda and Kokkos we will show code snippets and point out the differences and similarities directly. We will start by showing the data setup, because the data needs to get onto the GPU somehow, then we will move to compare and contrast the Cuda kernel and the Kokkos functor.

Here is code that shows the setup of the data on the GPU for Cuda:

```
1 float * dev_leftDataArray;
checkCudaError(cudaMalloc((void **) &dev_leftDataArray,
3   numContractions * numLeftFields * numPoints *
    sizeof(float)));
5
checkCudaError(cudaMemcpy(dev_leftDataArray, &leftDataArray[0],
7   numContractions * numLeftFields * numPoints * sizeof(float),
    cudaMemcpyHostToDevice));
9
```

Figure 4.3 Code from Cuda `ContractFieldFieldScalar`

There are essentially three steps in the process: declaring a pointer to the data on the CPU, creating an array with the correct size on the GPU, then copying the data over to the GPU from wherever the data is currently kept on the CPU. This process is pretty simple and self-explanatory. Now let us compare that to Kokkos:

```
typedef Kokkos::Cuda  DeviceType;
typedef Kokkos::View<float***, Kokkos::LayoutRight, DeviceType>
  ContractionData;
typedef typename ContractionData::HostMirror
  ContractionData_Host;

ContractionData dev_ContractData_Left("left_data",
  numContractions,
  numLeftFields,
  numPoints);

ContractionData_Host contractionData_Left =
  Kokkos::create_mirror_view(dev_ContractData_Left);

for (int cell = 0; cell < numContractions; ++cell) {
  for (int lbf = 0; lbf < numLeftFields; ++lbf) {
    for (int qp = 0; qp < numLeftFields; ++qp) {
      contractionData_Left(cell, lbf, qp) =
        contractionDataLeft[cell*numLeftFields*
        numPoints + lbf*numLeftFields + qp];
    }
  }
}

```

Figure 4.4 Code from Kokkos Cuda `ContractFieldFieldScalar`

The Kokkos code first defines and creates the device and host Views. One of the major differences compared to Cuda is that Kokkos uses its own data structure, a View, instead of an array. This is why we need to use typedefs to define the Views, but the extra work (which honestly is not much of a hassle) allows the programmer much more control over the data. The control also comes at the cost of having to use for loops to copy the data into the host view instead of being able to do a Memcpy. However, this is all initial work that needs to be done once, while the benefit of being able to change the layout of the data by changing the Kokkos::LayoutRight to

Kokkos::LayoutLeft is very useful, especially since this allows the programmer to optimize data layout for both the CPU and GPU. Overall, Kokkos' and Cuda's data setup have different philosophies, which makes sense because Kokkos needs to be easily optimized for both the CPU and GPU while Cuda only runs on the GPU.

Looking at the "guts" of the programs, Cuda has a kernel that is launched where all the computation is, while Kokkos uses a functor, almost identical to Intel's Thread Building Blocks threading paradigm. However, for programs doing the same calculation, the parenthesis operator function in Kokkos' functor is almost an exact replica of the code in Cuda's kernel. Here is the code for a Cuda kernel for ContractFieldFieldScalar:

While here is the parenthesis operator code for the Kokkos functor:

Although there is some more code for the Kokkos functor (the code required to declare the data members and the constructor), the Kokkos code looks a lot less cluttered. The Kokkos functor does not need to deal with figuring out the thread's ID, because it is an integer given as input, while the Cuda kernel needs to use blockId.x, blockDim.x, etc. Also indexing into the view is easier, especially when changing the layout of the data from LayoutLeft to LayoutRight (or vice versa) because no code changes need to occur in the functor.

4.3 Personal Experience and Thoughts

A task of the project was to document our experiences and thoughts about Kokkos, including any issues that we have run into. Using new tools and learning new syntax always has its tough periods, and getting used to Kokkos definitely had some periods where we had no idea why a program was not compile or giving an incorrect answer (especially in the beginning), but after the initial learning curve everything seemed to flow pretty well and make sense.

Our team has never actually been responsible for installing Kokkos on our machine, instead our liaison, Dr. Carter Edwards, did that for us, so we are unable to talk about the difficulties of downloading and installing the Kokkos library on our machine, but we did have lots of trouble trying to compile and linking against Kokkos originally. This was due to the fact that the same flags need to be used when installing and compiling and

linking against Kokkos. However, since we did not install Kokkos ourselves and the documentation showing how to compile and link against Kokkos used different flags than what were used during our installation, we struggled for a while. Already this shows how Kokkos' documentation is not as developed as one would like, which we will bring up later, but it is understandable since Kokkos is new.

Another obstacle that slowed us down when first using, is Kokkos' use of magic words. For example, Kokkos requires the programmer to typedef Kokkos::Cuda or Kokkos::OpenMP to device_type, and it must be device_type, not some other name. Although the programmer can easily fix this, if the programmer is unaware of this requirement it can cause a lot of hassle for a while. Every team member ran into this at one time or another, but after a while we got used to it. When following examples we learned to use the same names for the typedefs to make sure that we did not run into another bug with the same nature. Once again documentation would have helped in this situation, but there is not much documentation all we have are examples. On the bright side however, since we were able to write all of our programs by simply following a few examples we were able to see some of Kokkos' intuitiveness. Overall we really enjoy Kokkos' philosophy and structure, which as mentioned before, is almost identical to Intel's Thread Building Blocks (TBB). If you are familiar with TBB then learning Kokkos is almost as simple as learning the syntax because they are in the same paradigm.

As previously mentioned, Kokkos has very little documentation. For any emerging technology it is understandable that the creators choose to focus on functionality instead of documentation, but the documentation needs to catch up at some point. The examples were very helpful in getting us to our end goal of working code, but examples are not as helpful in understanding what exactly is happening, the meaning behind some portions of code, or why certain code is necessary. Documentation would have also been helpful in seeing the default values for functions and Views, as well as the other arguments that could have been passed instead. There were many times we tried to use Google to find information about Kokkos, but many times the information would point to uncommented pieces of code, which is not always helpful in determining what is going on. Overall we believe the documentation for Kokkos needs to improve in order for new users to get past the initial learning curve and spread the word about Kokkos.

As a whole, our team's experience with Kokkos has been positive and see that it offers a great alternative to other solutions that allow multithreading on multiple architectures. A quick overview of the benefits of

using Kokkos: Kokkos can create multithreaded code on the CPU, GPU, and XeonPhi, Views can easily change the layout of the data, functors seem to keep the code cleaner and more readable than Cuda's kernels, and the fact that Kokkos is a C++ library and not a a new language adds simplicity. Some of the downsides and changes that we believe would improve Kokkos include Views having more layouts than LayoutRight and LayoutLeft, the use of magic words (or lack of using the right magic words) can create bugs that are hard to find, the example code should include comments to describe what is happening, and finally the documentation needs to improve. However, extended use of Kokkos will solve most of these problems except for Views being limited to two layout types, which is why our team had an overall good experience with Kokkos.

```
__global__ void
cudaContractFieldFieldScalar_Flat_kernel(int numContractions,
  int numLeftFields,
  int numRightFields,
  int numPoints,
  float * __restrict__ dev_contractData_Left,
  float * __restrict__ dev_contractData_Right,
  float * dev_contractResults) {
  int contractionIndex = blockId.x * blockDim.x + threadIdx.x;
  while (contractionIndex < numContractions) {
    int myID = contractionIndex;
    int myCell = myID / (numLeftFields * numRightFields);
    int matrixIndex = myID % (numLeftFields *
      numRightFields);
    int matrixRow = matrixIndex / numRightFields;
    int matrixCol = matrixIndex % numRightFields;

    // Calculate now to save computation later
    int lCell = myMatrix * numLeftFields * numPoints;
    int rCell = myMatrix * numRightFields * numPoints;
    int resultCell = myMatrix * numLeftFields *
      numRightFields;

    float temp = 0;
    for (int qp =0; qp < contractionSize; qp++) {
      temp += dev_contractData_Left[lCell +
        qp*numLeftFields + matrixRow] *
        dev_contractData_Right[rCell +
        qp*numRightFields + matrixCol];
    }

    dev_contractResults[resultCell +
      matrixRow * numRightFields + matrixCol] =
        temp;

    contractionIndex += blockDim.x * gridDim.x;
  }
}
```

Figure 4.5 Code from Cuda `ContractFieldFieldScalar`

```
KOKKOS_INLINE_FUNCTION
void operator() (const unsigned int elementIndex) const {
  int myID = elementIndex;
  int myCell = myID / (_numLeftFields * _numRightFields);
  int matrixIndex = myID % (_numLeftFields * _numRightFields);
  int matrixRow = matrixIndex / _numRightFields;
  int matrixCol = matrixIndex % _numRightFields;

  float temp = 0;
  for (int qp = 0; qp < _numPoints; qp++) {
    temp += _leftFields(myCell, qp, matrixRow) *
      _rightFields(myCell, qp, matrixCol);
  }
  _outputFields(myCell, matrixRow, matrixCol) = temp;
}
```

Figure 4.6 Code from Kokkos Cuda `ContractFieldFieldScalar`

Chapter 5

Our Performance

The original stated goal of this clinic project was to parallelize a number of tensor manipulation kernels in the Intrepid library, and then move on to other kernels that performed more complex computations. However, by the end of our term as a clinic team, we will have only focused our energy on the tensor manipulaiton kernels, without having moved on to any other kernels. The reason for this is twofold:

Firstly, we underestimated the number of obstacles we would encounter over the course of our project. Originally, many of these obstacles stemmed from the fact that no member of our team had written performance-oriented parellel code before beginning work on the project. However, even as we grew more familiar with the concepts involved, we also ran into a number of issues with Kokkos, which is effectively still in an alpha testing stage, and at the beginning of the semester had very little documentation, and a number of other issues that made it difficult to work with. Over winter break, we received a newly updated version of Kokkos that included a few fixes for some of the issues we had been having, along with some new features.

The second reason we never made it to another package was the Kokkos update we received over winter break. With the update, we could write team-oriented parallel code, which allowed us to implement algorithms such as team reductions, and it allowed us access to shared memory on the GPU. At this point, rather than merely writing flat parallel versions of a larger number of the kernels, we decided to focus more heavily on general parallelization techniques using Kokkos teams for tensor contractions in order to find the best way to parallelize our desired kernels. As a result of our shift to focusing on team techniques, we spend the entire spring semester

working on implementing, testing, and plotting results from from various algorithms as applied to the tensor manipulations library, and never moved on to a second kernel (as we had originally intended to do).