



Mathematics Clinic

Final Report for
Sandia National Laboratories

Parallelizing Intrepid Tensor Contractions Using Kokkos

April 4, 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

Abstract	iii
Acknowledgments	xi
1 Sandia National Laboratories	1
1.1 Background	1
1.2 Problem	1
1.3 CPU vs GPU	1
1.4 Kokkos	1
2 Intrepid	3
2.1 Tensor Contractions	3
2.2 Intrepid Contractions Overview	3
2.3 ContractDataDataScalar	5
2.4 ContractDataDataVector	5
2.5 ContractDataDataTensor	6
2.6 ContractDataFieldScalar	6
2.7 ContractDataFieldVector	7
2.8 ContractDataFieldTensor	7
2.9 ContractFieldFieldScalar	7
2.10 ContractFieldFieldVector	8
2.11 ContractFieldFieldTensor	8
3 Parallelism	11
3.1 Flat Parallelism	11
3.2 Reduction	11
3.3 Slicing	11
3.4 Tiling	11

4	Experience with Kokkos	13
4.1	Performance	13
4.2	Snippets	16
4.3	Personal Experience and Thoughts	18
5	Performance	23

List of Figures

2.1	Code from serial ContractDataDataScalar	5
2.2	Code from serial ContractDataDataVector	5
2.3	Code from serial ContractDataDataTensor	6
2.4	Code from serial ContractDataFieldScalar	6
2.5	Code from serial ContractDataFieldVector	7
2.6	Code from serial ContractDataFieldTensor	8
2.7	Code from serial ContractFieldFieldScalar	9
2.8	Code from serial ContractFieldFieldVector	9
2.9	Code from serial ContractFieldFieldTensor	10
4.1	ContractDataDataScalar Kokkos performance comparison .	14
4.2	ContractFieldFieldScalar Kokkos performance comparison .	15
4.3	Code from Cuda ContractFieldFieldScalar	17
4.4	Code from Kokkos Cuda ContractFieldFieldScalar . . .	17
4.5	Code from Cuda ContractFieldFieldScalar	21
4.6	Code from Kokkos Cuda ContractFieldFieldScalar . . .	22

List of Tables

2.1	Summary of the nine Intrepid tensor contraction kernels . .	4
2.2	Intrepid tensor contraction suffixes	4

Acknowledgments

Chapter 1

Sandia National Laboratories

1.1 Background

1.2 Problem

1.3 CPU vs GPU

1.4 Kokkos

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

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.

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 indi-

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

cates the dimensionality of the output and the second indicates the number of contraction indices.

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

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.

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) {
2       for (int qp = 0; qp < quadraturePoints; ++qp) {
3           output[c] += leftInput[c][qp] * rightInput[c][qp];
4       }
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) {
2       for (int qp = 0; qp < quadraturePoints; ++qp) {
3           for (int t = 0; t < iVec; ++t) {
4               output[c] += leftInput[c][qp][t] * rightInput[c][qp][t];
5           }
6       }
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) {  
2       for (int qp = 0; qp < quadraturePoints; ++qp) {  
3           for (int t1 = 0; t1 < iVec1; ++t1) {  
4               for (int t2 = 0; t2 < iVec2; ++t2) {  
5                   output[c] += leftInput[c][qp][t1][t2] *  
6                               rightInput[c][qp][t1][t2];  
7               }  
8           }  
9       }  
10  }
```

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.

```
1   for (int c = 0; c < numcells; ++c) {  
2       for (int l = 0; l < lbf; ++l) {  
3           for (int qp = 0; qp < quadraturepoints; ++qp) {  
4               output[c][l] += left[c][l][qp] * right[c][qp];  
5           }  
6       }  
7   }
```

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.

```

1  for (int c = 0; c < numcells; ++c) {
2      for (int l = 0; l < lbf; ++l) {
3          for (int qp = 0; qp < quadraturepoints; ++qp) {
4              for (int t = 0; t < iVec; ++t) {
5                  output[c][l] += left[c][l][qp][t] * right[c][qp][t];
6              }
7          }
8      }
9  }

```

Figure 2.5 Code from serial ContractDataFieldVector

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.

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.

```
1   for (int c = 0; c < numcells; ++c) {  
2       for (int l = 0; l < lbf; ++l) {  
3           for (int qp = 0; qp < quadraturepoints; ++qp) {  
4               for (int t1 = 0; t1 < iVec1; ++t1) {  
5                   for (int t2 = 0; t2 < iVec2; ++t2) {  
6                       output[c][l] += left[c][l][qp][t1][t2] * right[c][qp  
7   ] [t1][t2];  
8   }  
9   }  
10  }  
11 }
```

Figure 2.6 Code from serial `ContractDataFieldTensor`

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 contracts away three indices, keeping the `Cell` dummy dimension as well as the left and right basis function indices.

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

```

1  for (int c = 0; c < numcells; ++c) {
2      for (int l = 0; l < lbf; ++l) {
3          for (int r = 0; r < rbf; ++r) {
4              for (int qp = 0; qp < quadraturepoints; ++qp) {
5                  output[c][l][r] += left[c][l][qp] * right[c][r][qp];
6              }
7          }
8      }
9  }

```

Figure 2.7 Code from serial ContractFieldFieldScalar

```

1  for (int c = 0; c < numcells; ++c) {
2      for (int l = 0; l < lbf; ++l) {
3          for (int r = 0; r < rbf; ++r) {
4              for (int qp = 0; qp < quadraturepoints; ++qp) {
5                  for (int t = 0; t < iVec; ++t) {
6                      output[c][l][r] += left[c][l][qp][t] * right[c][r][
7                      qp][t];
8                  }
9              }
10         }
11     }

```

Figure 2.8 Code from serial ContractFieldFieldVector

```
1  for (int c = 0; c < numcells; ++c) {  
2      for (int l = 0; l < lbf; ++l) {  
3          for (int r = 0; r < rbf; ++r) {  
4              for (int qp = 0; qp < quadraturepoints; ++qp) {  
5                  for (int t1 = 0; t1 < iVec1; ++t1) {  
6                      for (int t2 = 0; t2 < iVec2; ++t2) {  
7                          output[c][l][r] += left[c][l][r][qp][t1][t2] *  
8                              right[c][l][r][qp][t1][t2];  
9                      }  
10                 }  
11             }  
12         }  
13     }  
14 }  
15
```

Figure 2.9 Code from serial ContractFieldFieldTensor

Chapter 3

Parallelism

3.1 Flat Parallelism

3.2 Reduction

3.3 Slicing

3.4 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 than 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:

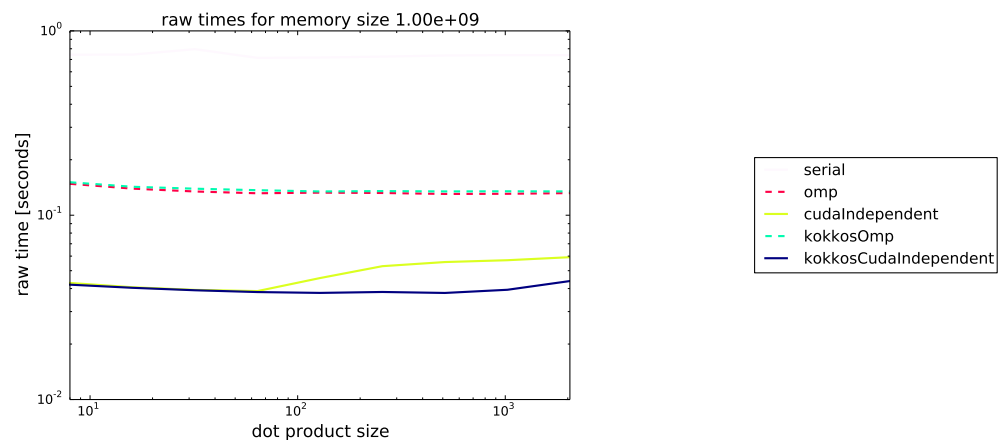


Figure 4.1 This graph plots the performance of Kokkos Cuda, Cuda, Kokkos OpenMP, and OpenMP for ContractDataDataScalar with a memory size of 1 GB. The y-axis is time in seconds, so closer to 0 is better. The x-axis plots different contraction sizes in order to compare multiple points.

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.

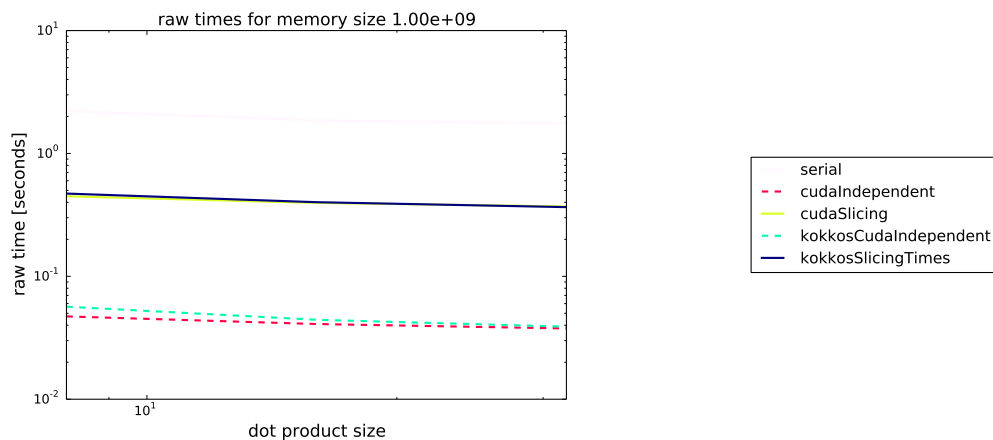


Figure 4.2 This graph shows performance differences (or similarities in our case) of the nested parallelism approach slicing.

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:

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:

```

float * dev_leftDataArray;
2 checkCudaError(cudaMalloc((void **) &dev_leftDataArray ,
   numContractions * numLeftFields * numPoints *
4   sizeof(float)));

6 checkCudaError(cudaMemcpy(dev_leftDataArray , &leftDataArray[0],
   numContractions * numLeftFields * numPoints * sizeof(float),
8   cudaMemcpyHostToDevice));

```

Figure 4.3 Code from Cuda ContractFieldFieldScalar

```

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

6 ContractionData dev_ContractData_Left("left_data",
8   numContractions,
   numLeftFields,
10  numPoints);

12 ContractionData_Host contractionData_Left =
   Kokkos::create_mirror_view(dev_ContractData_Left);
14
16 for (int cell = 0; cell < numContractions; ++cell) {
   for (int lbf = 0; lbf < numLeftFields; ++lbf) {
18     for (int qp = 0; qp < numLeftFields; ++qp) {
       contractionData_Left(cell, lbf, qp) =
20         contractionDataLeft[cell*numLeftFields*
           numPoints + lbf*numLeftFields + qp];
       }
22   }
24 }

```

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 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.


```

1 __global__ void
  cudaContractFieldFieldScalar_Flat_kernel(int numContractions,
3   int numLeftFields,
   int numRightFields,
5   int numPoints,
   float * __restrict__ dev_contractData_Left,
7   float * __restrict__ dev_contractData_Right,
   float * dev_contractResults) {
9   int contractionIndex = blockIdx.x * blockDim.x + threadIdx.x;
  while (contractionIndex < numContractions) {
11    int myID = contractionIndex;
    int myCell = myID / (numLeftFields * numRightFields);
13    int matrixIndex = myID % (numLeftFields *
      numRightFields);
15    int matrixRow = matrixIndex / numRightFields;
    int matrixCol = matrixIndex % numRightFields;
17
    // Calculate now to save computation later
19    int lCell = myMatrix * numLeftFields * numPoints;
    int rCell = myMatrix * numRightFields * numPoints;
21    int resultCell = myMatrix * numLeftFields *
      numRightFields;
23
    float temp = 0;
25    for (int qp = 0; qp < contractionSize; qp++) {
      temp += dev_contractData_Left[lCell +
27        qp*numLeftFields + matrixRow] *
      dev_contractData_Right[rCell +
29        qp*numRightFields + matrixCol];
    }
31
    dev_contractResults[resultCell +
33      matrixRow * numRightFields + matrixCol] =
      temp;
35
    contractionIndex += blockDim.x * gridDim.x;
37  }
39 }

```

Figure 4.5 Code from Cuda ContractFieldFieldScalar

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

9   float temp = 0;
   for (int qp = 0; qp < _numPoints; qp++) {
11    temp += _leftFields(myCell, qp, matrixRow) *
          _rightFields(myCell, qp, matrixCol);
13   }
   _outputFields(myCell, matrixRow, matrixCol) = temp;
15 }
```

Figure 4.6 Code from Kokkos Cuda ContractFieldFieldScalar

Chapter 5

Performance

