HARVEY MUDD COLLEGE

Computer Science Clinic

Final Report for
Sandia National Laboratories

Parallelizing Intrepid Tensor Contractions Using Kokkos

May 5, 2015

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

Advisor
Jeff Amelang

Liaisons
H. Carter Edwards
Robert J. Hoekstra

Abstract

As computer hardware capabilities increase, code parallelism is becoming an increasingly vital part of writing high-performing, computationally expensive code such as that used in scientific computing. Some problems in the scientific computing fields lend themselves to parallelism on a Graphics Processing Unit (GPU) as well as the more standard Central Processing Unit (CPU), but because these two hardware architectures are dramatically different, code written for one architecture does not easily port to the other. Sandia National Labs has developed a new C++ library called Kokkos, which addresses this issue by abstracting away the hardware considerations, allowing code to be written once for either the CPU or GPU. This year, the team used Kokkos to explore parallel algorithms on the CPU and GPU for performing tensor contractions, a class of algebraic operations often used in scientific computing.

Contents

Abstract **iii**

Acknowledgments **xiii**

1 Sandia National Laboratories **1**
- 1.1 Background 1
- 1.2 Problem 2
- 1.3 CPU vs GPU 2
- 1.4 Kokkos 4

2 Intrepid **7**
- 2.1 Tensor Contractions 7
- 2.2 Intrepid Contractions Overview 8
- 2.3 ContractDataDataScalar 10
- 2.4 ContractDataDataVector 10
- 2.5 ContractDataDataTensor 11
- 2.6 ContractDataFieldScalar 11
- 2.7 ContractDataFieldVector 12
- 2.8 ContractDataFieldTensor 13
- 2.9 ContractFieldFieldScalar 13
- 2.10 ContractFieldFieldVector 14
- 2.11 ContractFieldFieldTensor 15

3 Parallelism **17**
- 3.1 Flat Parallelism 17
- 3.2 Reduction 21
  - 3.2.1 Team Depth 1 22
  - 3.2.2 Team Depth 2 24
  - 3.2.3 Teamstride 27
  - 3.2.4 Reduction Special Case 28

3.3 Slicing 32
3.4 Tiling 37
3.4.1 ContractFieldFieldTensor 41

4 Experience with Kokkos **47**
4.1 Performance 47
4.2 Code Snippets 49
4.3 Personal Experience and Thoughts 54

5 Our Performance **55**

List of Figures

2.1 Standard matrix multiplication. The horizontal index of the left matrix and the vertical index of the right matrix are contracted away. 8
2.2 Code from serial `ContractDataDataScalar` 10
2.3 Code from serial `ContractDataDataVector` 11
2.4 Code from serial `ContractDataDataTensor` 11
2.5 Code from serial `ContractDataFieldScalar` 12
2.6 Code from serial `ContractDataFieldVector` 12
2.7 Code from serial `ContractDataFieldTensor` 13
2.8 Code from serial `ContractFieldFieldScalar` 14
2.9 Code from serial `ContractFieldFieldVector` 14
2.10 Code from serial `ContractFieldFieldTensor` 15

3.1 Speedup over serial of the flat parallel algorithm of the `ContractFieldFieldScalar` kernel. At its best this algorithm runs 60 times faster than serial. The closest corner, where speedup is only about 2.5 times, does not fit into Sandia's use case of expected problem sizes. 20
3.2 Code from Team Depth 1 reduction functor for `ContractDataDataTensor` 22
3.3 Code from kernel launch for `ContractDataDataTensor` Team Depth 1 reduction 23
3.4 Speedup of ContractDataDataTensor with Team Depth 1 algorithm over serial 24
3.5 Code from Team Depth 2 reduction functor for `ContractDataDataTensor` 25
3.6 Code from kernel launch for `ContractDataDataTensor` Team Depth 2 reduction 26

3.7 Speedup of ContractDataDataTensor with Team Depth 2 algorithm over serial . . . 26
3.8 Code from Teamstride functor for `ContractDataDataTensor` 27
3.9 Code from kernel launch for `ContractDataDataTensor` Teamstride . . . 27
3.10 Speedup of ContractDataDataTensor with Teamstride algorithm over serial . . . 28
3.11 Code for the special reduction case in `ContractFieldFieldScalar` . . . 30
3.12 A comparison between the reduction algorithm with the special case and the reduction algorithm without the special case. 31
3.13 Demonstration of memory accesses for the first team a slicing implementation of ContractFieldFieldScalar . . . 33
3.14 Demonstration of memory accesses for the second team of a slicing implementation of ContractFieldFieldScalar . . . 33
3.15 Code from slicing algorithm on `ContractFieldFieldScalar` 34
3.16 ContractFieldFieldScalar slicing speedup compared to independent parallelism . . . 35
3.17 Performance of slicing parallelism when compared to serial 36
3.18 Performance of Adaptive Slicing on the same problem, note the increased speedup . . . 36
3.19 Demonstration of first round of memory accesses for a tiling implementation of ContractFieldFieldScalar . . . 37
3.20 Demonstration of second round of memory accesses for a tiling implementation of ContractFieldFieldScalar . . . 37
3.21 Kokkos implementation of tiling for ContractFieldFieldScalar 39
3.22 Raw times for many different algorithms used for ContractFieldFieldScalar. Note that Tiling is the best performing (lowest). This data was generated with $\ell = \mathcal{R} = 125$, $p = 216$ 40
3.23 Raw times for many different algorithms used for ContractFieldFieldScalar. This data was generated with $\ell = \mathcal{R} = 8$, $p = 8$ . . . 41
3.24 Speedup of Tiling for ContractFieldFieldTensor over independent parallelism. This data was generated with $\ell = \mathcal{R} = 16$, $d_1 = d_2 = 4$, and P varying from 16 to 128. The tile size used was 16. . . . 42

3.25 Speedup for serial, independent, and tiling approaches to `ContractFieldFieldScalar`. This data was generated with $\ell = \mathcal{R} = 16$, $d_1 = d_2 = 4$, with p varying from 8 to 128. This graph uses a relatively low memory size, which limits the number of cells 43
3.26 Speedup for serial, independent, and tiling approaches to `ContractFieldFieldScalar`. This data was generated with $\ell = \mathcal{R} = 16$, $d_1 = d_2 = 4$, with p varying from 8 to 128. This data was collected while simulating a memory size an order of magnitude larger than the previous image, leading to a significantly higher cell count. 44
3.27 Code for chokes implementation of tiling for ContractFieldFieldTensor 45

4.1 ContractDataDataScalar Kokkos performance comparison . 48
4.2 ContractFieldFieldScalar Kokkos performance comparison . 49
4.3 Code from Cuda `ContractFieldFieldScalar` 50
4.4 Code from Kokkos Cuda `ContractFieldFieldScalar` . . . 51
4.5 Code from Cuda `ContractFieldFieldScalar` 52
4.6 Code from Kokkos Cuda `ContractFieldFieldScalar` . . . 53

List of Tables

2.1 Summary of the nine Intrepid tensor contraction kernels . . 9
2.2 Intrepid tensor contraction suffixes 9

Acknowledgments

We would like to specially thank our liaisons H. Carter Edwards and Robert J. Hoekstra. We would also like to thank Denis Ridzal for helping us understand the Intrepid kernels and finally our faculty advisor Jeff Amelang for guidance throughout the project.

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.

With the 1996 comprehensive nuclear test ban, Sandia began to focus more heavily on computer simulations. These computationally intense simulations have pushed Sandia to perform more and more optimizations on their codebase, as faster-running programs allow greater throughput on simulation results.

Traditionally, parallel algorithms used the Message Passing Interface (MPI) standard, allowing many machines to work collaboratively on partitioned subdomains of a global problem. The vast majority of scientific software produced and used by the national labs relies on MPI to leverage both inter-node and intra-node parallelism. In the case of intra-node parallelism, this model pretends that the various computational engines within a computer are actually separate computers. 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 had not been high enough to motivate the use 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 for the intra-node parallelism. Unfortunately, this means that much existing code will have to be rewritten, as MPI cannot be used to leverage the parallelism of co-processors such as Graphical Processing

Units (GPUs), which are further discussed in Section 1.3.

1.2 Problem

The task of this clinic project has been to rewrite several kernels included in libraries within Sandia's Trilinos Project. The Trilinos Project is a collection of open source libraries intended for use in large-scale scientific applications. Specifically, this clinic team aimed to rewrite some of these Trilinos kernels to be efficient and thread-scalable on manycore architectures such as GPUs.

As well as presenting Sandia with a set of faster kernels, Sandia has also requested that we present a general approach to parallelizing code. Many of the mathematicians, engineers, and scientists at Sandia have little experience writing code for GPUs or large thread count coprocessors such as Intel's Xeon Phi. Over the course of our clinic project, we have uncovered a number of parallelization pitfalls along with techniques that work well for achieving speedup. By recording our experiences, we hope to make it easier for Sandia's engineers to apply similar techniques to more of their codebase after the termination of this clinic project.

The kernels we focused on during our project were in a tensor manipulation library within Intrepid, a sub-package of Trilinos. The Intrepid package is a library of kernels designed for performing discretizations of partial differential equations. As such, by improving the performance of these kernels within Intrepid, we can improve performance of any calculation-heavy simulation libraries that rely on Intrepid to calculate tensor contractions. See Chapter 2 for more detail on Intrepid.

1.3 CPU vs GPU

Most traditional computer programs run on a Central Processing Unit (CPU). CPUs are characterized by relatively low thread counts. For reference, a modern personal computer typically supports 2-8 hardware threads. Even CPU nodes on scientific computing clusters usually support fewer than 32 threads. Instead of supporting many hardware threads, CPU designers choose to dedicate a large portion of the chip for a CPU to caching and other features that in some ways make up for programming inefficiencies that would otherwise reduce performance.

Our project has mostly focused on writing code that will run on Graphical Processing Units (GPUs). GPUs are characterized by extremely high thread counts[1], decreased memory per thread, and relatively small instruction sets when compared to CPUs. In general, this means that programming on a GPU is less forgiving in the sense that sloppy programs will run significantly slower. For this reason, despite the much higher level of parallelism afforded by the large thread count, it is easy to write parallel GPU code that runs slower than equivalent serial CPU code.

However, GPUs have 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) than CPUs. Additionally, GPUs are more efficient (in terms of FLOPs/watt) than CPUs, which makes them appealing in supercomputers, where power consumption is a major concern.

Well implemented GPU code can yield significant speedup in certain processing-heavy applications. Specifically, a problem may work well on a GPU if it features high levels of arithmetic computation that can be calculated independently; that is, if each calculation does not rely on the results of the other calculations. For example, calculating a sequence of Fibonacci numbers is very difficult for a GPU, as each number relies on results from the previous two numbers. A more ideal problem for a GPU would be to take two large arrays and multiply them element-wise into a third array. When doing element-wise multiplication, there is no reliance on previous results, which means that the threads on the GPU will never have to wait for one another.

Another key consideration when writing high-performance code on any architecture is the memory access pattern. Programs invariably need to retrieve data that is stored in memory (RAM). On traditional CPU architectures, it is best to access this memory sequentially, to take advantage of caching. A single thread of execution on the CPU should ideally access memory locations immediately adjacent to that same thread's previous access.

However, this is not the case on the GPU. On a GPU, groups of threads called *warps* all access memory at the same time. For this reason, among others, threads in a warp should coordinate their memory accesses such that adjacent threads in a warp access adjacent locations in memory, a pattern called 'coalescing'.

These differences between ideal CPU code and GPU code mean that code that is written well for one architecture will usually perform poorly on the other if ported directly without changing the data structures and memory access patterns. Therefore, code tends to be architecture-dependent, so switching to a new architecture often means rewriting an entire codebase. This is not ideal for Sandia, where the mathematicians, engineers and scientists are interested in using the most up-to-date architecture without needing to constantly overhaul their codebase.

---

[1] In order to saturate its cores, a GPU requires a minimum of approximately 15,000 threads, with 150,000+ being required for good performance

1.4 Kokkos

In order to mitigate the effects of architecture dependent code overhauls, Sandia is developing a C++ library called Kokkos, which is included in the Trilinos package. Kokkos attempts to solve the issues of architecture dependence by allowing programmers to write their code one time using Kokkos, and then compile (or make other small tweaks) for optimization on a variety of architectures. This is possible because the library helps manage the allocation and access 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. Finally, Kokkos provides a concept of 'thread teams.' Thread teams are groups of threads that work together to solve a problem, and can be used in nested parallelism algorithms such as those detailed in Section 3.2 and Section 3.4.

Ideally, all of Sandia's codebase would be written using Kokkos, so that no more future large overhauls will be required as new architectures are released. For this reason, all of the parallel code we have written for the clinic project has used Kokkos.

The Kokkos package is still under development and no large-scale projects have yet been fully implemented using Kokkos. As such, another function of this clinic project has been to perform a large-scale test of Kokkos. Thus, we hope to provide both a faster and more future-safe codebase, and also early feedback on the Kokkos project. Sandia hopes for features from Kokkos to eventually be included in the C++17 standard, so any feedback (both positive and negative) we can provide would be useful.

Additionally, Sandia wishes to verify the claim that Kokkos is not slower than other popular methods of thread parallelism for scientific computing, namely OpenMP (for CPU parallelism) and Cuda (for GPU parallelism). In fact, Kokkos uses OpenMP and Cuda backends, for compiling on CPUs and GPUs respectively. We therefore hope to show that there is no overhead to using Kokkos rather than one of the more established parallelism solutions.

Chapter 2

Intrepid

Intrepid is a C++ library developed as part of Sandia's Trilinos Project. Intrepid's main functionality is to perform algebraic operations over multidimensional arrays. Tensor contractions are one class of operations 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, this clinic team considered Intrepid tensor contractions to be good targets for parallelism, as they had 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 multiplicative algebraic operations over tensors in which pairs of indices, one from each of the two input tensors, can be "contracted" together, reducing the dimensionality of the output tensor.

As an example, three different operations can be performed on two matrices (two-dimensional tensors): an outer product, a matrix multiplication, or an inner product. In an outer product, the two matrices are combined to form a four-dimensional tensor. In a matrix multiplication, two of the four resultant indices are contracted together, resulting in a two-dimensional tensor, a visual representation of which can be seen in Figure 2.1. In an inner product, all of the resultant indices are contracted together, resulting in a scalar. In typical terminology, the last two are considered "contractions".

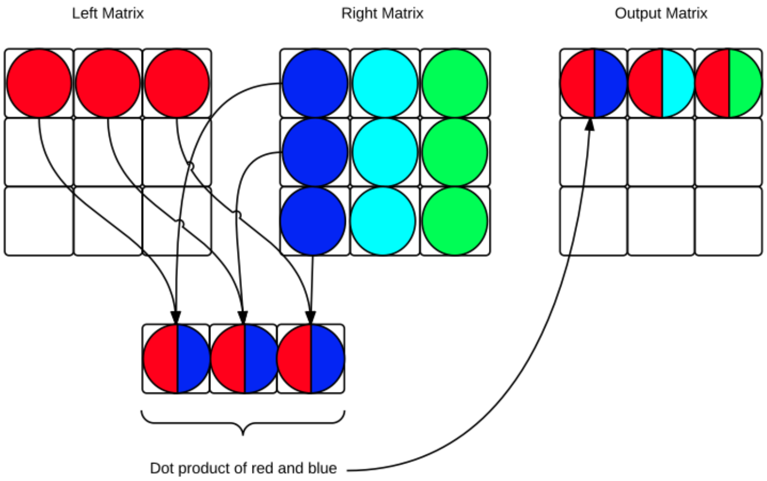


Figure 2.1 Standard matrix multiplication. The horizontal index of the left matrix and the vertical index of the right matrix are contracted away.

In this section, we have discussed the three possibilities for two-dimensional tensor contractions. However, tensor contractions also generalize to tensors with higher dimensionality, such as three or four dimensional tensors. In these higher dimensional tensors, there are more options for how many contraction indices might exist.

2.2 Intrepid Contractions Overview

The Intrepid library 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 ker-

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

nel 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.

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, in the `ContractDataDataScalar` kernel, 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 (one-dimensional tensors). 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.

It is also important to note that the tensor contraction kernels in Intrepid each actually perform many contractions. For instance, `ContractDataDataScalar`, which performs contractions of two input vectors to a single output scalar (commonly known as a 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 one-dimensional tensors (vectors) and outputs an array of scalars. A snippet showing the simple (conceptual) serial implementation of `ContractDataDataScalar` can be seen in Figure 2.2.

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

```

Figure 2.2 Code from serial `ContractDataDataScalar`

As shown in Figure 2.2, this kernel contracts away the `Quadrature Points` dimension, leaving only the `Cell` dimension. It can also be thought of as performing 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.

As shown in Figure 2.3, `ContractDataDataVector` is very similar to `ContractDataDataScalar`, except it contracts two indices instead of one. Again, only the `Cell` dimension is left, so we can think of this as a sort of two-dimensional dot product.

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

```

Figure 2.3 Code from serial `ContractDataDataVector`

2.5 ContractDataDataTensor

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

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

```

Figure 2.4 Code from serial `ContractDataDataTensor`

As shown in Figure 2.4, `ContractDataDataTensor` is very similar to `ContractDataDataVector` and `ContractDataDataScalar`, but has three contraction indices instead of the one or two of the previous cases.

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 2.5 Code from serial `ContractDataFieldScalar`

As shown in Figure 2.5, `ContractDataFieldScalar` has two non-contraction indices, the `Left Basis Function` index (`lbf`) 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.

```
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 2.6 Code from serial `ContractDataFieldVector`

As shown in Figure 2.6, `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 t1 = 0; t1 < iVec1; ++t1) {
        for (int t2 = 0; t2 < iVec2; ++t2) {
          output[c][l] += left[c][l][qp][t1][t2]
                          * right[c][qp][t1][t2];
        }
      }
    }
  }
}

```

Figure 2.7 Code from serial `ContractDataFieldTensor`

As shown in Figure 2.7, `ContractDataFieldTensor` is similar to `ContractDataFieldScalar`, but has three contraction indices.

2.9 ContractFieldFieldScalar

`ContractFieldFieldScalar` takes in two arrays of matrices and contracts away one index to produce an array of matrices. This effectively performs classical matrix multiplication on each pair of input matrices.

As shown in Figure 2.8, `ContractFieldFieldScalar` has two non-contraction indices, so the output is an array of matrices.

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

```

Figure 2.8 Code from serial `ContractFieldFieldScalar`

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.

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

```

Figure 2.9 Code from serial `ContractFieldFieldVector`

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

2.11 ContractFieldFieldTensor

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

```
for (int c = 0; c < numcells; ++c) {
  for (int l = 0; l < lbf; ++l) {
    for (int r = 0; r < rbf; ++r) {
      for (int qp = 0; qp < quadraturepoints; ++qp) {
        for (int t1 = 0; t1 < iVec1; ++t1) {
          for (int t2 = 0; t2 < iVec2; ++t2) {
              output[c][l][r] += left[c][l][r][qp][t1][t2]
                                   * right[c][l][r][qp][t1][t2];
          }
        }
      }
    }
  }
}

```

Figure 2.10 Code from serial `ContractFieldFieldTensor`

As shown in Figure 2.10, `ContractFieldFieldTensor` is similar to `ContractFieldFieldScalar`, but with three contraction indices.

Chapter 3

Parallelism

3.1 Flat Parallelism

In this section, we will discuss how to write 'flat' parallel code using Kokkos. Flat parallel code is code that does not use Kokkos' thread teams concept. This means that flat parallel code does not make use of GPU shared memory, or intra-team reductions. Flat parallel code does have access to an atomic fetch and add function, but this can have performance costs. These costs occur because atomic fetch and add can cause a bottleneck if too many threads are trying to write to the same memory location. Therefore, we were limited to writing *flat* algorithms in which each thread knew its responsibility and did not have to interact with other threads. In this section we will describe how to write high performing kernels for both the CPU and GPU using this flat parallelism technique. We will also describe some of its shortcomings and which other non-flat parallel algorithms solve these issues.

Over the course of our project, we identified four main factors that affect code's performance. These factors are: the thread count, the amount of work each thread must do, memory access patterns, and how data is laid out in memory. The first two factors (thread count and work per thread) are linked and are inversely proportional to one another.

The thread count and work per thread are generally determined by how far you decide to break down a problem. For example, in `ContractFieldFieldScalar`, which performs many matrix multiplications, a programmer could make each thread do one full matrix multiplication. In this case, the thread count must be equal to the number of cells, or matrix multiplications, that must be calculated. Figuring out the best way to break down the responsibility of a single thread requires looking at the expected problem size.

Representative use cases for the example of `ContractFieldFieldScalar` have from one thousand to tens of thousands of cells, with matrix sizes between eight by eight and sixty-four by one hundred twenty-five. In this case, when writing code for the GPU, it is best to break down the problem into the smallest possible pieces that avoid interaction, which corresponds to one thread per output element in each output matrix. This is because the goal is to minimize the amount of work per thread on the GPU, while still avoiding thread interaction.

The question always becomes: how many threads per contraction is optimal? As we can see from the dimensions described above, our total number of threads will be roughly 1,000 times our number of threads per contraction. Therefore, we would like to have between a couple hundred and a thousand threads per contraction. This will ensure that the GPU is being saturated, which is a necessary condition for high performance. To reiterate Section 1.3, we need to make sure that the GPU always has a *minimum* of 15,000 threads being spawned, with numbers in the 100,000-1,000,000 range being preferable.

When we combine this knowledge with the goal of the threads not being required to interact or write to the same memory location, we discover that the ideal number of threads per contraction we can use for flat parallelism is always the same as the maximum number of threads we can use per contraction, because this maximizes thread count and minimizes the work per thread. The maximum number of threads we can use is equal to of entries in the output tensor, because if we go over that number, then the threads would need to interact to write to the same output location.

Note that the expected output tensor dimensions range from a single number (for any problem that contains `DataData` in its name), to sixty-four squared (the largest practical value for `numLeftFields` and `numRightFields`, which are the dimensions of the output matrix for some problems). This range means that if we use flat parallelism for the `DataData` contractions, we will only be spawning 1,000-10,000 threads on the GPU in the normal use cases, which are below acceptable in terms of saturation. We can see the results of this effect in Figure

Now that the number of threads and their responsibility are known, the best access patterns and data layouts must be calculated. As described earlier, the best access patterns and data layouts for threads on the CPU are ones that utilize the cache. This means that when using CPU, which corresponds to code written with `Kokkos::OpenMP`, we want a thread's next memory read to be next to its current memory read. However, when using the GPU, which corresponds to code written with `Kokkos::Cuda`, a thread's memory read should be adjecent to the reads of threads with nearby indices, as these threads will be in the same warp.

These two memory access patterns are in direct conflict. To solve this problem, Kokkos abstracts out the data layout by introducing a data structure called a View that encodes multidimensional arrays with custom memory layouts. The main layout parameters to a Kokkos View are `LayoutLeft` and `LayoutRight`. In `LayoutLeft`, the left-most indices are adjacent in memory, i.e., incrementing the left-most index by 1 moves to the next memory location. The opposite is true for `LayoutRight`. This means we can use `LayoutLeft` for one architecture, (i.e. `Kokkos::Cuda`), and `LayoutRight` for the other (`Kokkos::OpenMP`) [1].

All that remains is to figure out how to arrange the data, or which order to put the indices so that one layout coalesces the memory while the other uses the cache. This is best shown by an example; in `ContractFieldFieldScalar` there are inputs $A_{c,l,p}$ and $B_{c,r,p}$. Assuming we have a thread per output element in output $C_{c,l,r}$, then we can have the inputs ordered as follows: $A_{c,l,p}$ and $B_{c,r,p}$. When using `Kokkos::OpenMP`, we will assume that the Views are `LayoutRight`, so $A_{i,j,k}$ will be right next to $A_{i,j,k+1}$ in memory, while $A_{i,j,k}$ will be very far from $A_{i+1,j,k}$ in memory. Each thread needs to do a dot product of a row in A with a column in B, so a thread needs to loop through all of p for the same value of c and l in A and also loop through all of p for the same c and r in B. Notice however that all the different values of p in A and B, where c, l, and r are fixed, are next to each other in memory, sicne we are using `LayoutRight`. This means that it will be cache friendly for any thread.

In contrast, for `Kokkos::Cuda`, the memory accesses must be coalesced in the optimal case. This can be achieved by using the same data, but storing it in `LayoutLeft` Views instead of `LayoutRight` views. In this case, c values that differ by 1 are adjacent in memory. Therefore, if thread x is responsible for calculating $C_{i,j,k}$, thread $x+1$ should responsible for calculating $C_{i+1,j,k}$. In this manner, by changing our threading policy and the layout of our Views, our memory accesses are now coalesced, as desired.

Finding the best way to lay out memory to optimize for both the CPU and GPU is not too difficult. One good technique is to first find the best way for caching (CPU), then imagine using the opposite layout (left or right) and seeing if there is any way to coalesce the memory. This way one functor can be used, the data layout can be easily changed, and the performance for both the CPU and GPU will be high. Using this flat parallel technique we have received many good results, an example of which can be seen in Figure 3.1, but there are some issues with flat parallelism.

---

[1] In fact, if the memory layout for a view is not provided, Kokkos will default to `LayoutLeft` on the GPU and `LayoutRight` on a CPU

It is important to find a way to arrange the indices of your data such that caching and coalescing can be achieved by chaning the layout of the view (as we just did). If we had not laid out our indices as we did above, i.e. if c were in the middle rahter than on the left, then it can be difficult, or even impossible to come up with a threading policy for each of CPU and GPU to match it. In this case, we are back to the scenario of writing different code for different architectures, but the whole goal of our project was to avoid doing so.

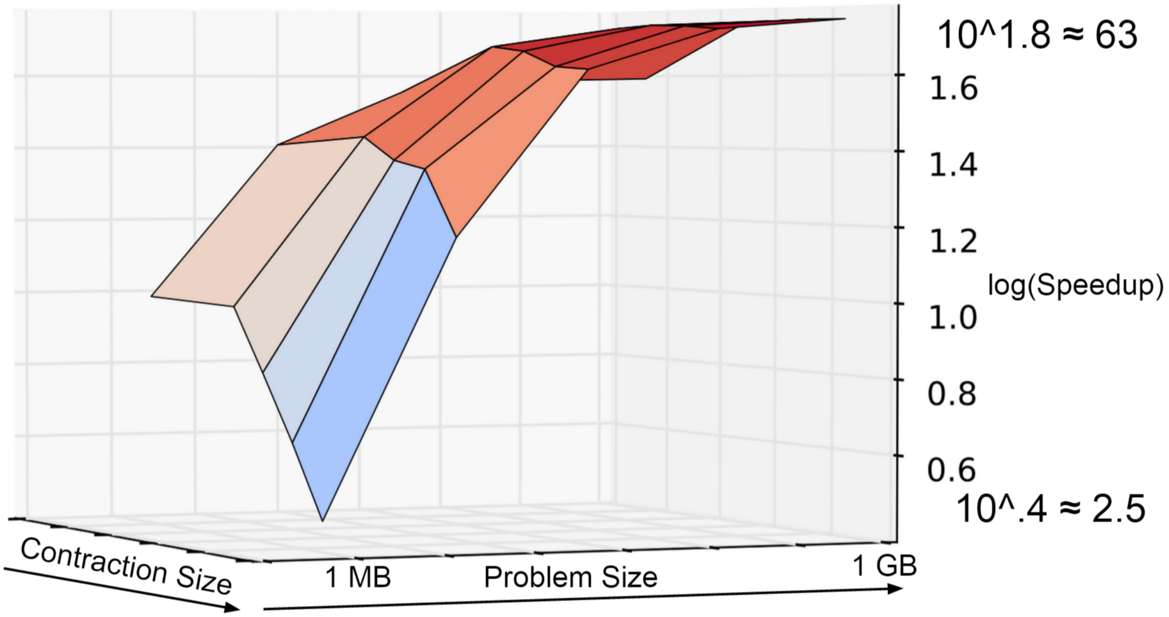


Figure 3.1 Speedup over serial of the flat parallel algorithm of the `ContractFieldFieldScalar` kernel. At its best this algorithm runs 60 times faster than serial. The closest corner, where speedup is only about 2.5 times, does not fit into Sandia's use case of expected problem sizes.

Flat parallelism is an easy-to-implement, high-performing algorithm when there is enough work to saturate the GPU. However, there isn't always enough independent work to use flat parallelism. For example, on all the problems that have `DataData` in their name, the output is simply an array of scalars, meaning that only one thread per contraction can be used. The GPU is great for using tens of thousands of threads to do computation, but when the number of threads is severely limited, the CPU performs better, so other algorithms must be explored on the GPU. For examples of these algorithms, see Section 3.2

Another feature of the GPU that can be leveraged that is outside the scope of flat parallelism is shared memory. Shared memory is essentially a user controlled cache on the GPU. Algorithms that use shared memory will be discussed in Section 3.3 and Section 3.4. The main benefits of not using shared memory are it is easier to code and the speedup of shared memory over flat parallelism is relatively small compared to the speedup flat parallel algorithms reap over serial code.

3.2 Reduction

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, but allows more threads to be created to saturate the GPU.

In Kokkos, threads can be organized into teams[2]. Built-in reduction methods allow teams to merge the contributions of their constituent threads.

---

[2] for seasoned Cuda programmers, teams correspond to Cuda blocks

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 `_dim2` dimension. Each thread therefore performs `_numPoints` × `_dim2` multiplications, and then combines its local sum with that of the other threads in the team.

```
// A team does one cell
const unsigned int cellIndex = thread.league_rank();

float sum = 0;

// Each of the _dim2 threads contracts the qp and d1 dimensions.
Kokkos::parallel_reduce(Kokkos::TeamThreadLoop(thread, _dim2),
// We use an anonymous (lambda) function here:
//      - [&] specifies that all automatic variables in the lambda are
//          passed by reference; this allows us to save time that would be
//          spent copying large arrays.
//      - The lambda takes two arguments, d2 by value and localsum by
//          reference.  The first argument is passed to each thread (this is
//          the index into the _dim2 dimension to be looped over by this
//          thread), and the second is the reduction target.
    [&] (const unsigned int d2, float& localsum) {
      for (unsigned int qp=0; qp < _numPoints; ++qp) {
        for (unsigned int d1=0; d1 < _dim1; ++d1) {
          // Each thread loops over two dimensions and then reduces into
          // localsum
          localsum +=  _leftInput(cellIndex, qp, d1, d2) *
            _rightInput(cellIndex, qp, d1, d2);
        }
      }
  }, sum);

if (thread.team_rank() == 0) {
  _output(cellIndex) = sum;
}

```

Figure 3.2 Code from Team Depth 1 reduction functor for `ContractDataDataTensor`

In Figure 3.2, we can see that each thread loops over the `_numPoints` and `_dim1` dimensions, and then reduces with the other threads in the team. The call to Kokkos' `parallel_for` function, seen in Figure 3.3, specifies an execution policy in which the number of teams launched is `_numCells`, and each team has `_dim2` threads.

As shown in Figure 3.4, this Team Depth 1 algorithm performs generally better than serial, but the speedup is minimal. Therefore, we explored

```
    const team_policy reduction_policy(numCells, _dim2);
    Kokkos::parallel_for(reduction_policy,
   contractDataDataTensorTeamDepth1Functor );
    Kokkos::fence();

```

Figure 3.3 Code from kernel launch for `ContractDataDataTensor` Team Depth 1 reduction

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

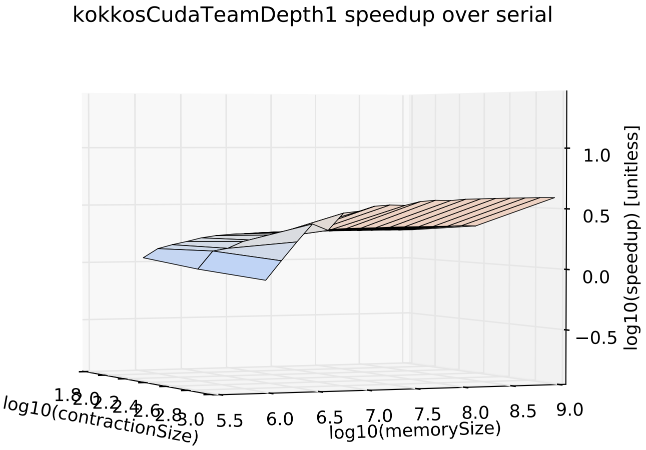


Figure 3.4 Speedup of ContractDataDataTensor with Team Depth 1 algorithm over serial

3.2.2 Team Depth 2

This reduction algorithm is similar to the previous one. In a similar approach to that taken in the Team Depth 1 reduction, we assign one team per contraction. In contrast to Team Depth 1, here each team has `_dim1` × `_dim2` threads, each responsible for `_numPoints` multiplications. Each thread then combines its local sum with that of the other threads in the team.

In Figure 3.5, 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 3.6, specifies an execution policy in which the number of teams launched is `_numCells`, and each team has `_dim1` × `_dim2` threads.

As shown in Figure 3.7, this Team Depth 2 algorithm performs better than serial for the three representative use case.. 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.

```
  // A team does one cell
  const unsigned int cellIndex = thread.league_rank();

  float sum = 0;
  // Each of the _dim1 * _dim2 threads contracts the _numPoints
  dimension
  Kokkos::parallel_reduce(Kokkos::TeamThreadLoop(thread, _dim1 *
   _dim2),
    [&] (const unsigned int threadIndex, float& localsum) {
      const unsigned int d1 = threadIndex / _dim2;
      const unsigned int d2 = threadIndex % _dim2;

      for (unsigned int qp = 0; qp < _numPoints; ++qp) {
        localsum +=  _leftInput(cellIndex, qp, d1, d2) *
          _rightInput(cellIndex, qp, d1, d2);
      }

   }, sum);

  if (thread.team_rank() == 0) {
   _output(cellIndex) = sum;
  }


```

Figure 3.5 Code from Team Depth 2 reduction functor for `ContractDataDataTensor`

However, because the team size is fixed based on the size of the `_dim1` and `_dim2` 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.

```
    const team_policy reduction_policy(numCells, _dim2 * _dim1);
    Kokkos::parallel_for(reduction_policy,
   contractDataDataTensorTeamDepth2Functor );
    Kokkos::fence();

```

Figure 3.6 Code from kernel launch for `ContractDataDataTensor` Team Depth 2 reduction

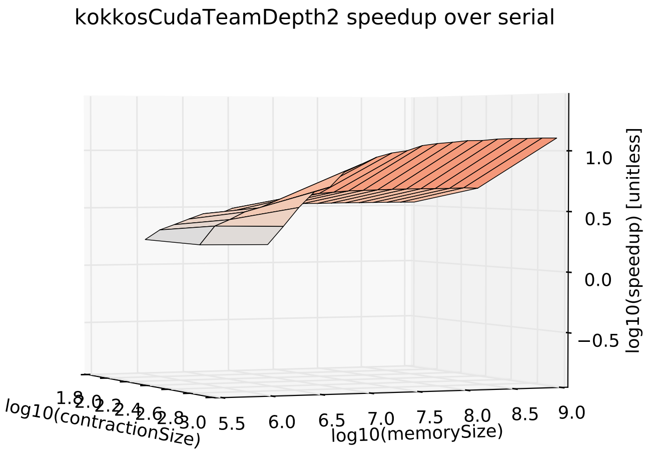


Figure 3.7 Speedup of ContractDataDataTensor with Team Depth 2 algorithm over serial

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 are spawned for each team. These threads treat the three contraction indices (`_numPoints`, `_dim1`, `_dim2`) as if they were a single index, each thread striding forwards by the number of threads on this "combined" index. This technique is similar to that used by OpenMP's collapse clause. 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.

```
// A team does one cell
const unsigned int cellIndex = thread.league_rank();

float sum = 0;

Kokkos::parallel_reduce (Kokkos::TeamThreadLoop(thread,cellSize),
  [&](const unsigned int threadIndex, float& localsum) {
    // Calculate the next element to add (striding by teamsize)
    const unsigned int qp = threadIndex / (_dim1 * _dim2);
    const unsigned int d1 = threadIndex % (_dim1 * _dim2) / _dim2;
    const unsigned int d2 = threadIndex % _dim2;

    localsum +=  _leftInput(cellIndex, qp, d1, d2) *
      _rightInput(cellIndex, qp, d1, d2);
  } , sum );

if (thread.team_rank() == 0) {
  _output(cellIndex) = sum;
}
```

Figure 3.8 Code from Teamstride functor for `ContractDataDataTensor`

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

```

Figure 3.9 Code from kernel launch for `ContractDataDataTensor` Teamstride

As seen in Figure 3.8, this algorithm requires more arithmetic on the part of each thread because the thread is not responsible for looping over a single subset of indices but is instead looping over all three contraction indices. The corresponding call in Figure 3.9 uses a fixed team size of 32, unlike the previous two algorithms. This team size seemed to be the best for performance; a team size of 64 yielded similar speedup, but larger team sizes did not perform as well.

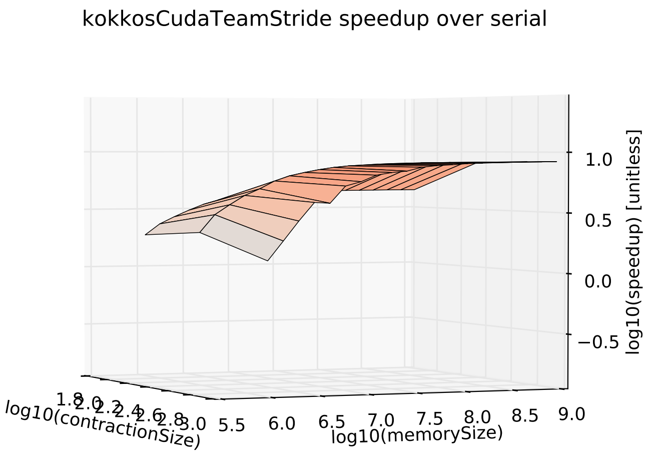


Figure 3.10 Speedup of ContractDataDataTensor with Teamstride algorithm over serial

As shown in Figure 3.10, 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 explored in tensor contraction functions in which the inputs may consist of a few large tensor contractions.

3.2.4 Reduction Special Case

As mentioned above, the parallel reduction algorithm outperforms the flat parallel algorithm when there is a small number of threads but lots of work per thread. We also created parallel reduction algorithms for problems that did not have this issue. An example is ContractFieldFieldScalar, which could have as small as eight multiplies for a single thread since p could be as few as eight. As one may guess, the Teamstride parallel reduction algorithm performs worse than the flat parallel algorithm. This is because we are creating more threads than necessary and dividing up a small amount of work between at least 32 threads. The work is divided between at least 32 threads because, remembering the architecture of the GPU, there are 32 threads in a warp, all of which run in lock step. So in cases where $p < 32$, threads are created that are not used in this reduction algorithm. To mitigate this phenomenon we created a special reduction case.

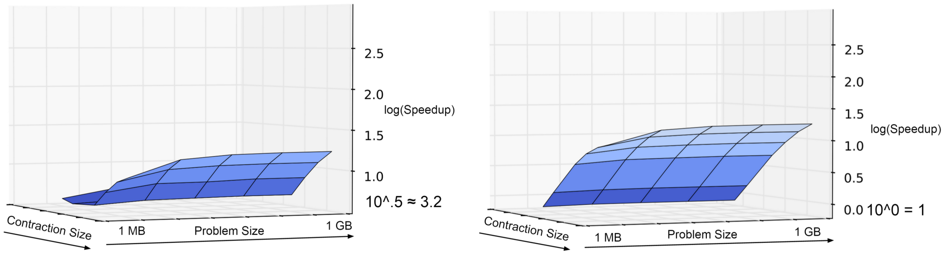
The special reduction case creates fewer teams, giving more work per team, if more than half of the threads in a warp are being wasted. Consider the case of ContractFieldFieldScalar where 24 threads can wasted, because only 8 need to do a multiply then reduction, one quarter of the teams are created, and each team is responsible for calculating four times as many outputs elements. This case adds the code in Figure 3.11 to the functor's operator() function.

In the code of Figure 3.11, the number of multiplies, that is need is numPoints. If that number is less than 16, then we want one team to do more than one output. Increasing the number of active threads per team leads to higher efficiency and speed. One thing that needs to be noted for the code in the special case is that an `atomic_fetch_add` is used instead of a `parallel_reduce`. This is due to the fact that there is no `team_reduce` function where half the threads reduce to one location while the other half reduce to another location. This has the side effect of requiring the output locations to be 0 before the calculation, while in the "normal" reduction algorithm that is not necessary.

Another point that needs to be reiterated is that this special case can still perform worse than the flat parallel algorithm, but its purpose is to increase the performance of a pure reduction algorithm in the region of the plot where it performs most poorly. Figure 3.12 shows the effect of using the special case. The flap up for small contraction sizes does not exist for the algorithm not including the special case. The reduction with the special case outperforms the reduction algorithm without the special case by a lot for small contraction sizes.

```
// The if case is where the special reduction case is handled
if (numPoints <= 16) {
  int myID = thread.league_rank()*(threadsPerTeam/numPoints)+thread.team_rank()/numPoints;
  int myMatrix = myID / (numLeftFields * numRightFields);
  int matrixIndex = myID - (myMatrix * (numLeftFields * numRightFields));
  int matrixRow = matrixIndex / numRightFields;
  int matrixCol = matrixIndex - (matrixRow * numRightFields);

  int pointIndex = thread.team_rank() % numPoints;

  float mult = leftView(myMatrix, matrixRow, pointIndex)
    * rightView(myMatrix, pointIndex, matrixCol);

  Kokkos::atomic_fetch_add(&outputView(myMatrix, matrixRow, matrixCol), mult);
}
// This is where the normal reduction case is handled
else {
  int myID = thread.league_rank();
  int myMatrix = myID / (_numLeftFields * _numRightFields);
  int matrixIndex = myID - (myMatrix * (_numLeftFields * _numRightFields));
  int matrixRow = matrixIndex / _numRightFields;
  int matrixCol = matrixIndex - (matrixRow * _numRightFields);

  Kokkos::parallel_reduce(Kokkos::TeamThreadLoop(thread, _numPoints),
    [&] (const unsigned int& i, float& localSum) {
      localSum += _leftView(myMatrix, matrixRow, i) *
        _rightView(myMatrix, i, matrixCol);
      },
      sum);
  if (thread.team_rank() == 0) {
    _outputView(myMatrix, matrixRow, matrixCol) = sum;
  }
}

```

Figure 3.11 Code for the special reduction case in `ContractFieldFieldScalar`



a. Here is a graph showing the special case of Team Reduction algorithm's speedup over serial for ContractFieldFieldScalar. The faster speeds for the smaller contraction size (far left) is where the special case is used.

b. ContractFieldFieldScalar team reduction's speedup over serial without the special case in the reduction. Notice that the smaller contraction sizes continue to perform worse, which was not the case for the reduction algorithm with the special case.

Figure 3.12 A comparison between the reduction algorithm with the special case and the reduction algorithm without the special case.

3.3 Slicing

So far, we've introduced flat methods, which feature no interaction between threads, and reduction methods, which involve communication between threads. Now, we'll move on to two methods featuring even more interaction between threads. These last methods utilize shared memory, which provides a space for threads to store information and share it with other threads.

The first of these methods is a technique we call slicing. The first step of this method is to load one full contraction from the left input tensor 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`.

In Figure 3.13, on the left, we have the first of the two input matrices, whose first row's 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 one block of five threads which are labeled by color. Each of the threads reads in one of the elements on the left 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 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.

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 3.13 represents the first block of the contraction, then the second block will be represented in Figure 3.14.

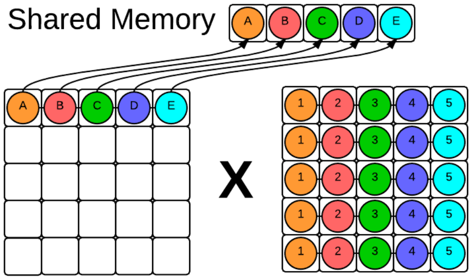


Figure 3.13 Demonstration of memory accesses for the first team a slicing implementation of ContractFieldFieldScalar

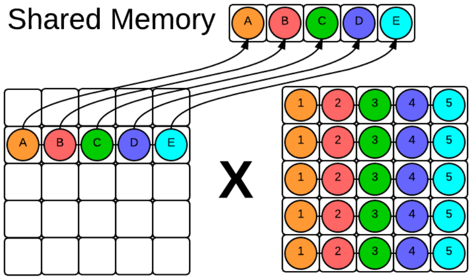


Figure 3.14 Demonstration of memory accesses for the second team of a slicing implementation of ContractFieldFieldScalar

We see that for the ContractFieldFieldScalar example, where our equation is given by $A_{c,l,p} \times B_{c,r,p} = O_{c,l,r}$, the number of blocks initialized by the algorithm will be equal $l \times c$, since there are l blocks per matrix, and we have c matrices. Additionally, there will be r threads per block.

Kokkos code for executing the algorithm as described above is included in Figure 3.15.

Taking a step back, another possible use of shared memory is to use tiles inspired by cache friendly implementations of matrix multiplication. This approach will be discussed in detail in the next section, but it bears mentioning now for contrast with the slicing approach. The main advantage of slicing when compared to a tile based approach is that it is easily generalizable to tensor contractions of higher dimensions. Unlike tiling (described in Section 3.4), 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, we would

```
int r = thread.team_rank();
int c = thread.league_rank() / numLeftFields;
int l = thread.league_rank() - c * numLeftFields; // (mod)

// Load a slice into shared memory, each thread loads as many elements as it needs to.
Kokkos::View<float*, Kokkos::MemoryUnmanaged> shared_slice(thread.team_shmem(), numPoints);
for (int p = thread.team_rank(); p < numPoints; p += thread.team_size()) {
  shared_slice(p) = leftView(c, l, p);
}
thread.team_barrier();

// Do as much as you can using that slice -- The parallel for goes across all the rows if possible.
// This for loop is for the dot product within a row.
float sum = 0;
for (int p = 0; p < numPoints; ++p) {
  sum += shared_slice(p) * rightView(c, p, r);
}
outputView(c, l, r) = sum;


```

Figure 3.15 Code from slicing algorithm on `ContractFieldFieldScalar`

expect it to perform poorly in situations where the memory needed to store slices is a limiting factor. One class of examples of this phenomenon are cases where the size of the contraction large relative to the number of basis functions.

Intuitively, slicing is reliant on large contraction sizes to produce speedup because in situations where the number of threads per block is low it is unable to saturate the GPU. In flat parallelism, we use one thread per element in the output matrix. Critically, these threads are not reliant one one another, so the GPU does not have to satisfy constraints to give threads in the same team access to shared memory. In contrast, slicing does place these types of requirements on the GPU, and when the size of a team is small (less than 100) the GPU suffers performance decreases that lead to slicing becoming less efficient. remedied by increasing the number of contractions per block, it can introduce problems with shared memory. Since shared memory is limited by nature (most NVIDIA GPUs offer only 16KB shared memory), slicing has to balance the amount of workwith the amount of shared memory available.

In situations where the contraction has an inherently large amount of reuse like `ContractFieldFieldScalar`, this problem can be remedied to some degree by loading multiple slices into shared memory and reusing them multiple times. However, some contractions , like `ContractDataDataScalar`, do not reuse any of the elements in the left input matrix to calculate multiple output elements. In these cases, 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 ContractFieldFieldScalar, we get underwhelming results, as shown by Figure 3.16.

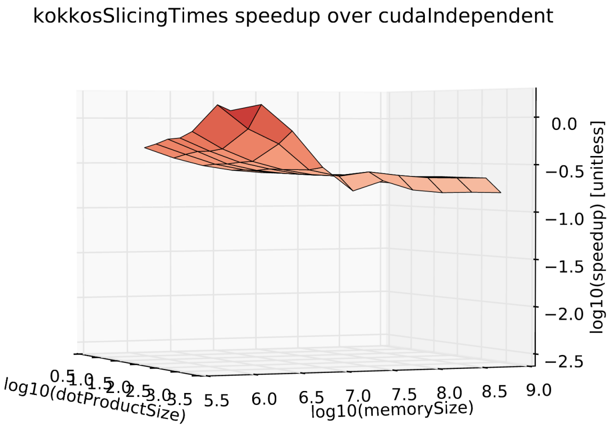


Figure 3.16 ContractFieldFieldScalar slicing speedup compared to independent parallelism

These results were generated by comparing independent algorithms to slicing on `ContractFieldFieldScalar` with $\ell = \mathcal{R} = 10$ and $P \in [8, 1024]$. 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 flat Cuda code, which is promising. This is the corner where we would expect slicing to perform the best with respect 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 out-compete 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, as previously discussed, is not enough.

Fortunately, this can be counteracted to some degree by loading multiple slices into shared memory simultaneously. This places a larger strain on shared memory, but also presents additional opportunities for parallelism. This technique has shown promising results on `ContractFieldFieldTensor`, as shown in Figure 3.18.

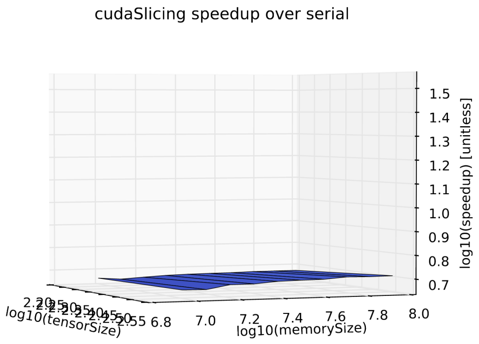


Figure 3.17 Performance of slicing parallelism when compared to serial

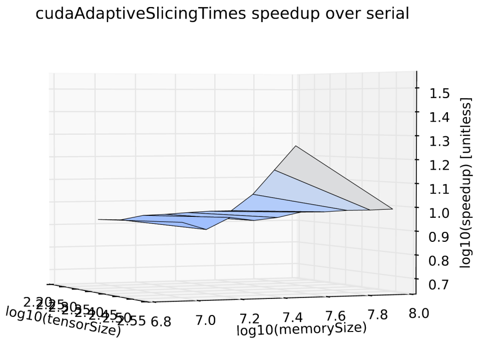


Figure 3.18 Performance of Adaptive Slicing on the same problem, note the increased speedup

This shows that the adaptive slicing approach does have some process in these applications. These results were generated by loading two slices simultaneously in `ContractFieldFieldTensor` with $\ell = \mathcal{R} = 125$, $d_1 = d_2 = 4$, and $p = 216$.

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 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` as shown in Figure 3.20.

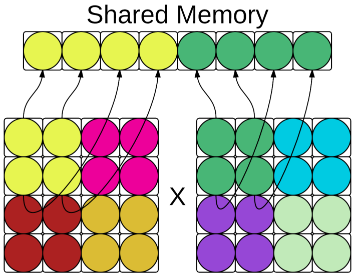


Figure 3.19 Demonstration of first round of memory accesses for a tiling implementation of ContractFieldFieldScalar

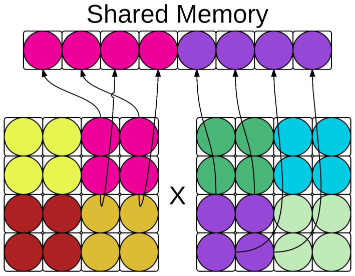


Figure 3.20 Demonstration of second round of memory accesses for a tiling implementation of ContractFieldFieldScalar

For the sake of simplicity we'll consider a team to be four threads, which simplifies our computation since the matrix is four by four. On the left hand side, the team 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 in Figure 3.20. We see that in this case we will have to load two tiles into shared memory before we have computed every output element in the first tile 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 tiling's 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, the algorithm saturates the GPU with both blocks and threads, something that is very difficult to do adaptively with pure slicing.

Excerpts from our CKokkos implementation of tiling are included in Figure 3.21. The code assumes that `tileSize` (the horizontal and vertical dimensions of a tile) evenly divides both the contraction size and $\ell = \mathcal{R} =$ numBasis.

```
    const unsigned int numBasis = numLeftFields;

    // We use -1, +1 to get the ceiling of numPoints/tile_size and numBasis/tile_size
    const unsigned int numberOfPointTiles = numPoints / tile_size;
    const unsigned int numberOfBasisTiles = numBasis / tile_size;

    const unsigned int numberOfTiles = numCells * numberOfBasisTiles * numberOfBasisTiles;

    const unsigned int subRow = thread.team_rank() / tile_size;
    const unsigned int subCol = thread.team_rank() - subRow * tile_size; // (mod)

    unsigned int resultTileIndex = thread.league_rank();

    // Create our tiles
    Kokkos::View<float**, Kokkos::MemoryUnmanaged> LeftTileStorage(thread.team_shmem(), tile_size,
      tile_size);
    Kokkos::View<float**, Kokkos::MemoryUnmanaged> RightTileStorage(thread.team_shmem(), tile_size,
       tile_size);

    const unsigned int resultMatrix = resultTileIndex / (numberOfBasisTiles * numberOfBasisTiles);
    const unsigned int resultSubmatrixIndex = resultTileIndex - (resultMatrix * numberOfBasisTiles
      * numberOfBasisTiles); // (mod)

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

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

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

      // load the left and right tiles into shared memory
      LeftTileStorage(subRow, subCol) = leftView(resultMatrix, row, tileNumber * tile_size +
     subCol);
      RightTileStorage(subRow, subCol) = rightView(resultMatrix, tileNumber * tile_size + subRow,
     col);

      // make sure everyone's finished loading their pieces of the tiles
      thread.team_barrier();
      for (unsigned int dummy = 0; dummy < tile_size; ++dummy) {
        sum += LeftTileStorage(subRow, dummy) *
            RightTileStorage(dummy, subCol);
        }
        thread.team_barrier();
   }
   outputView(resultMatrix, row, col) = sum;
   resultTileIndex += thread.league_size();
```

Figure 3.21 Kokkos implementation of tiling for ContractFieldFieldScalar

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

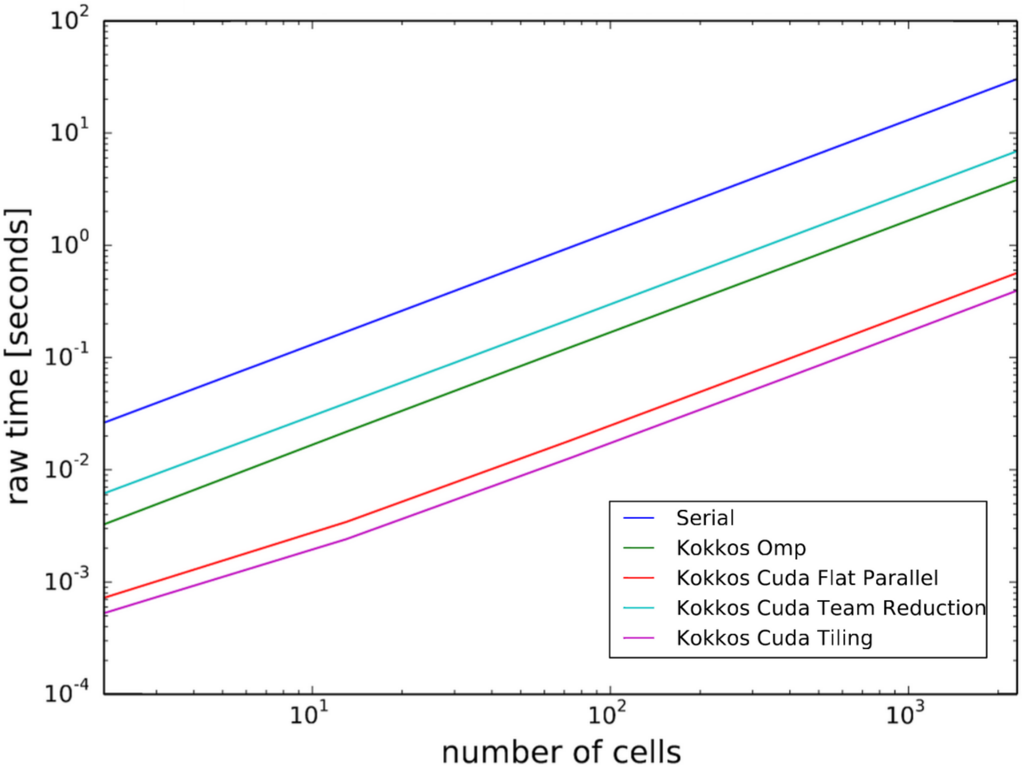


Figure 3.22 Raw times for many different algorithms used for ContractFieldFieldScalar. Note that Tiling is the best performing (lowest). This data was generated with $\ell = \mathcal{R} = 125$, $p = 216$

Consider the graph generated in Figure 3.22 for `ContractFieldFieldScalar`. 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 in Figure 3.23 when $\ell = \mathcal{R} = 8$, $P = 8$.

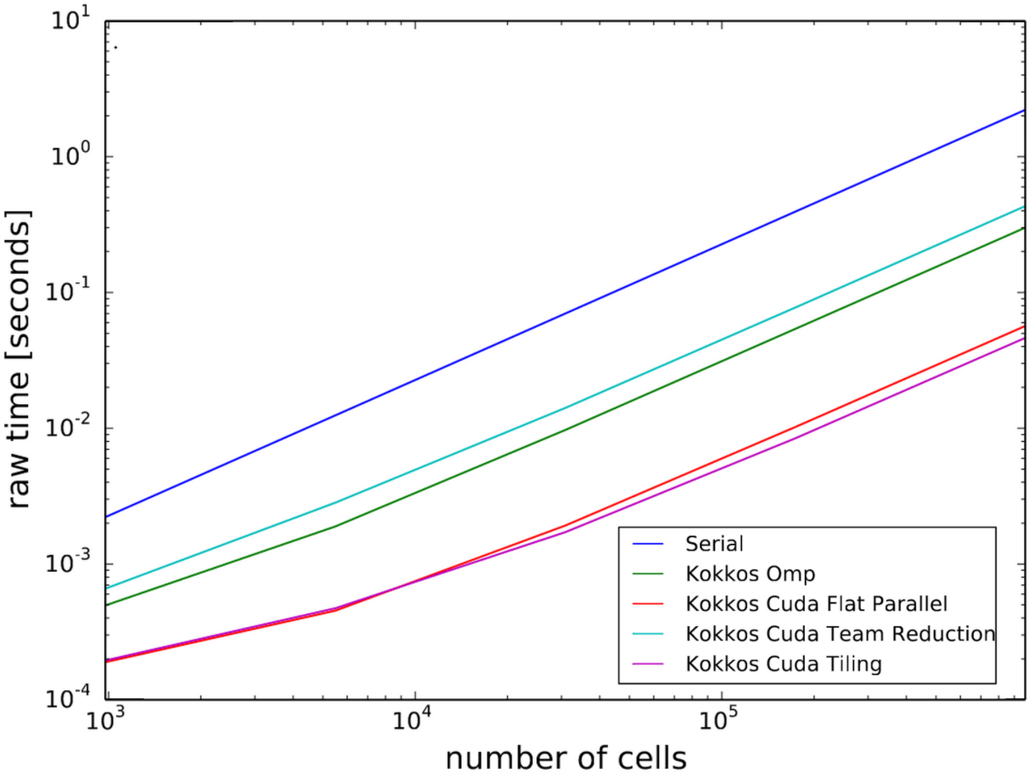


Figure 3.23 Raw times for many different algorithms used for ContractFieldFieldScalar. This data was generated with $\ell = \mathcal{R} = 8$, $p = 8$

3.4.1 ContractFieldFieldTensor

We also created a tiling implementation of `ContractFieldFieldTensor`. Conceptually, it is much more difficult to envision how to approach tiling on multidimensional problems. `ContractFieldFieldTensor`, which can be described by the equation $L_{C,\ell,P,d_1,d_2} \times R_{C,\mathcal{R},P,d_1,d_2} = O_{C,\ell,\mathcal{R}}$, therefore presents an interesting problem for the tiling approach.

Unlike slicing, there seem to be multiple distinct ways of approaching the problem. One can unroll the entire contraction to create a situation where the left and right matrices are rectangular, but can be covered by 2D tiles. Alternatively, algorithms could cover the left and right inputs using multidimentional tiles. While conceptually these two approaches seem distinct, they are functionally equivalent. Ultimately, tiling involves loading pieces of both the input and output matrices into shared memory and operating on them repeatedly. As long as we avoid suboptimal memory access patterns, it makes no difference whether we visualize the tiles as two dimensional or with even higher dimensions. For ease of understanding, we implemented `ContractFieldFieldTensor`'s tiling kernels with unrolled contractions. This essentially, reduces the kernel to a series of long dot products in our vision of the memory layout.

Using the tiling method on `ContractFieldFieldTensor` we were able to generate speedup of between 1.5 and 2 times over flat parallelism, as shown in Figure 3.24.

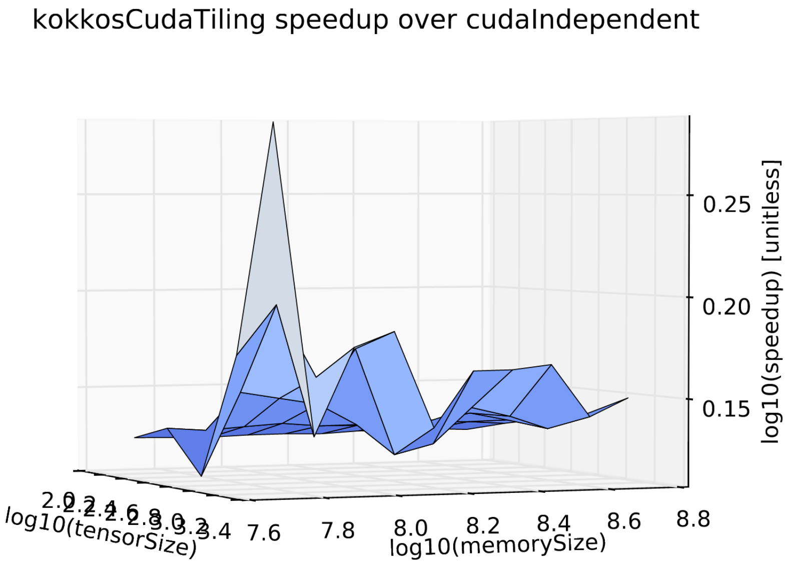


Figure 3.24 Speedup of Tiling for ContractFieldFieldTensor over independent parallelism. This data was generated with $\ell = \mathcal{R} = 16$, $d_1 = d_2 = 4$, and P varying from 16 to 128. The tile size used was 16.

In these cases, the size of the contraction will often be larger than the number of basis functions by as much as an order of magnitude. This makes tiling significantly less efficient because the matrices are rectangular instead of square, as we previously discussed. In this case, the number of GPU teams capable of working on a contraction using the tiling algorithm is given by $\frac{\text{number of basis functions}}{\text{size of a tile}}$. While tiling can still perform well in these use cases, as shown by Figures 3.25 and 3.26, it does not perform as well as on square matrices.

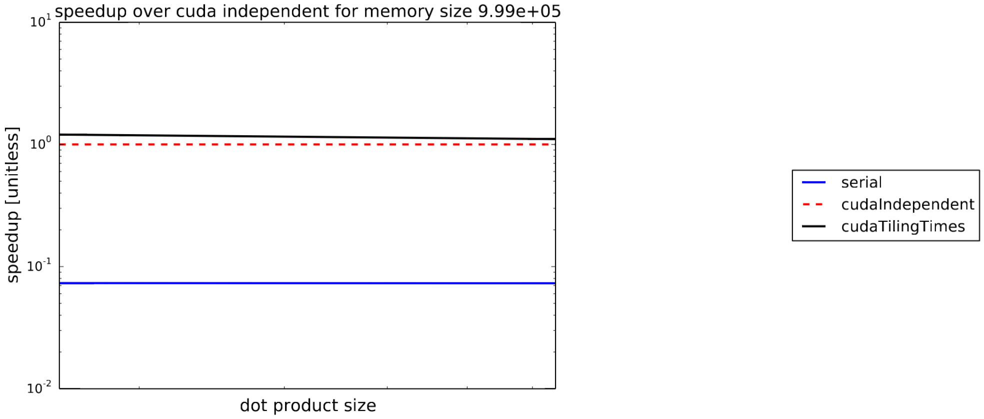


Figure 3.25 Speedup for serial, independent, and tiling approaches to `ContractFieldFieldScalar`. This data was generated with $\ell = \mathcal{R} = 16$, $d_1 = d_2 = 4$, with p varying from 8 to 128. This graph uses a relatively low memory size, which limits the number of cells

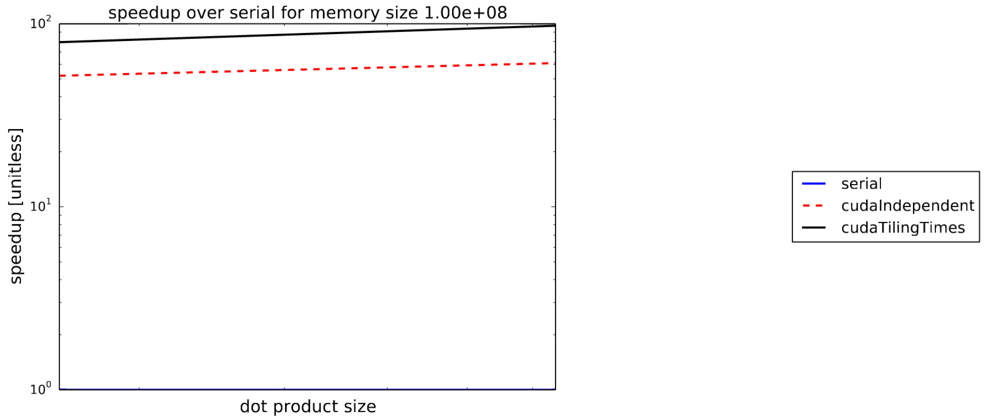


Figure 3.26 Speedup for serial, independent, and tiling approaches to `ContractFieldFieldScalar`. This data was generated with $\ell = \mathcal{R} = 16$, $d_1 = d_2 = 4$, with p varying from 8 to 128. This data was collected while simulating a memory size an order of magnitude larger than the previous image, leading to a significantly higher cell count.

Excerpts from our implementation of tiling for ContractFieldFieldTensor are included in Figure 3.27, which have been simplified using the assumption that $\ell = \mathcal{R}$ as well as the assumption that the size of each tile evenly divides every other dimension for clarity.

```

// Here we pretend that all three contraction dimensions are a single dimension:
  contractionSize
const unsigned int contractionSize = _dimTens1 * _dimTens2 * _numPoints;

const unsigned int numberOfPointTiles = contractionSize / tile_size;
const unsigned int numberOfBasisTiles = numBasis / tile_size;

const unsigned int numberOfTiles = _numCells * numberOfBasisTiles * numberOfBasisTiles;

//These are the subRow and subColumn of the thread within its tile
const unsigned int subRow = thread.team_rank() / _tile_size;
const unsigned int subCol = thread.team_rank() - subRow * _tile_size;

unsigned int resultTileIndex = thread.league_rank();
//Our shared memory for the computation, which fits two full tiles
Kokkos::View<float*, Kokkos::MemoryUnmanaged> tileStorage(thread.team_shmem(), 2 * _tile_size *
   _tile_size);

const unsigned int resultMatrix = resultTileIndex / (numberOfBasisTiles * numberOfBasisTiles);
const unsigned int resultSubmatrixIndex = resultTileIndex - (resultMatrix * numberOfBasisTiles
  * numberOfBasisTiles); // (mod)

// calculate result tile indices
const unsigned int resultTileRow = resultSubmatrixIndex / numberOfBasisTiles;
const unsigned int resultTileCol = resultSubmatrixIndex - resultTileRow * numberOfBasisTiles;
  // (mod)

// calculate this threads actual output index
const unsigned int row = resultTileRow * _tile_size + subRow;
const unsigned int col = resultTileCol * _tile_size + subCol;
float sum = 0;

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

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

    // Here we break it back down so that we can use it
    const unsigned int leftContractionIndex = tileNumber*_tile_size + subCol;
    const unsigned int left_qp = leftContractionIndex / (_dimTens1*_dimTens2);
    const unsigned int left_combinedTens = leftContractionIndex - left_qp * (_dimTens1 *
  _dimTens2); // (mod)
    const unsigned int left_iTens1 = left_combinedTens / _dimTens2;
    const unsigned int left_iTens2 = left_combinedTens - left_iTens1 * _dimTens2; // (mod)

    const unsigned int rightContractionIndex = tileNumber * _tile_size + subRow;
    const unsigned int right_qp = rightContractionIndex / (_dimTens1*_dimTens2);
    const unsigned int right_combinedTens = rightContractionIndex - right_qp * (_dimTens1 *
  _dimTens2); // (mod)
    const unsigned int right_iTens1 = right_combinedTens / _dimTens2;
    const unsigned int right_iTens2 = right_combinedTens - right_iTens1 * _dimTens2; // (mod)

    // load the left and right tiles into shared memory
    tileStorage(thread.team_rank()) = _leftView(resultMatrix, row, left_qp, left_iTens1,
  left_iTens2);
    tileStorage(thread.team_rank() + (_tile_size * _tile_size)) =
        _rightView(resultMatrix, right_qp, right_iTens1, right_iTens2, col);

    // make sure everyone's finished loading their pieces of the tiles
    thread.team_barrier();

    for (unsigned int dummy = 0; dummy < _tile_size; ++dummy) {
      sum +=
        tileStorage(leftBaseIndex + dummy) *
        tileStorage(rightBaseIndex + dummy * _tile_size);
    }
    thread.team_barrier();
  }

  _outputView(resultMatrix, row, col) = sum;


```

Figure 3.27 Code for chokes implementation of tiling for ContractFieldFieldTensor

Chapter 4

Experience with Kokkos

4.1 Performance

Since Kokkos uses Cuda and OpenMP as a backend to achieve faster performance, we chose to do some testing to confirm that Kokkos performs as well as these two solutions. If Kokkos performed worse then Cuda or OpenMP, then programmers might prefer these other solutions instead. Fortunately, we found that Kokkos matches the performance of Cuda and OpenMP in almost all cases. The rest of this section will describe our strategy for testing the performance of Kokkos compared to Cuda and OpenMP, present graphs showing the differences observed, and analyze the graphs.

In order to compare Kokkos, Cuda, and OpenMP, we wrote algorithmically equivalent code using all three paradigms (using the same data layout and memory access pattern) and recorded the runtime of each version. To reduce noise in the timing data, we repeated the same calculations five times and used the average time.

Note that because we were unsure how Kokkos implements the intra-team `team_reduce()` function, we could not write a matching Cuda reduction. Based on our project priorities, we chose not to pursue this further.

These graphs present some of the performance differences and similarities of Kokkos, Cuda, and OpenMP. Figure 4.1 shows the raw times of Kokkos Cuda, Cuda, Kokkos OpenMP, and OpenMP for ContractDataDataScalar.

In this graph, the y-axis is time in seconds, so closer to zero is better. The x-axis plots different contraction sizes. Here, Kokkos OpenMP and OpenMP are almost perfectly overlapping. We are not quite sure why they are not perfectly overlapping, but it appears too consistent to be random

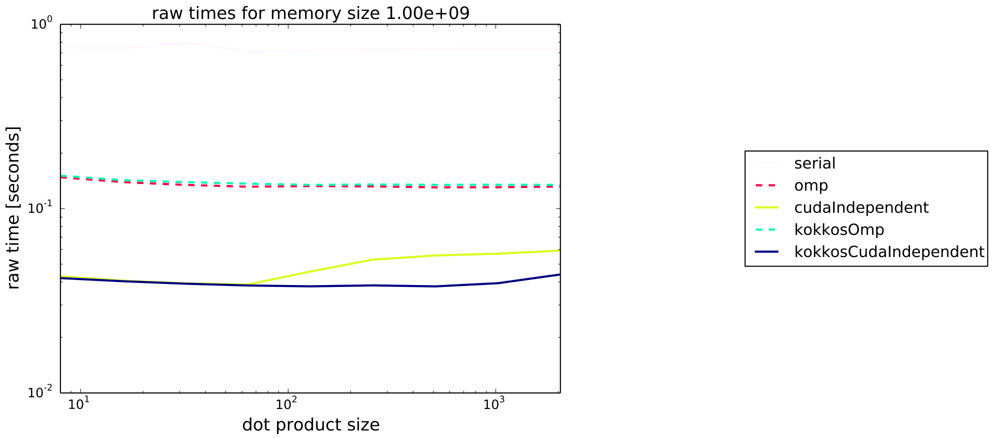


Figure 4.1 Performance of Kokkos Cuda, Cuda, Kokkos OpenMP, and OpenMP for ContractDataDataScalar with a memory size of 1 GB.

noise. However, the difference is small enough as to be fairly insignificant.

Kokkos Cuda, however, shows major differences compared to Cuda. The two perform identically for the smaller problems but diverge by a significant amount for bigger problems. This trend exists because Kokkos launches a different number of blocks compared Cuda; Kokkos launches fewer blocks, with the intention of reusing them. We believe this doesn't affect small problem sizes because such problems require fewer blocks than the large problem sizes, so both Kokkos and Cuda launch enough blocks. However, there is clearly a difference in the bigger problem sizes.

Figure 4.2 shows ContractFieldFieldScalar with the slicing technique (which uses shared memory) for both Kokkos Cuda and Cuda. It also includes the flat parallel algorithm in both Kokkos Cuda and Cuda.

In Figure 4.2, the Cuda slicing performance is almost identical to the Kokkos slicing performance. This shows that Kokkos' use of shared memory matches that of Cuda.

Overall, most of our graphs show that Kokkos performs almost identically to Cuda and to OpenMP. We therefore concluded that Kokkos is not adding any major overhead on a variety of problems. There may be slight performance differences due to optimization choices, but Kokkos performs similarly to the native multithreading solutions.

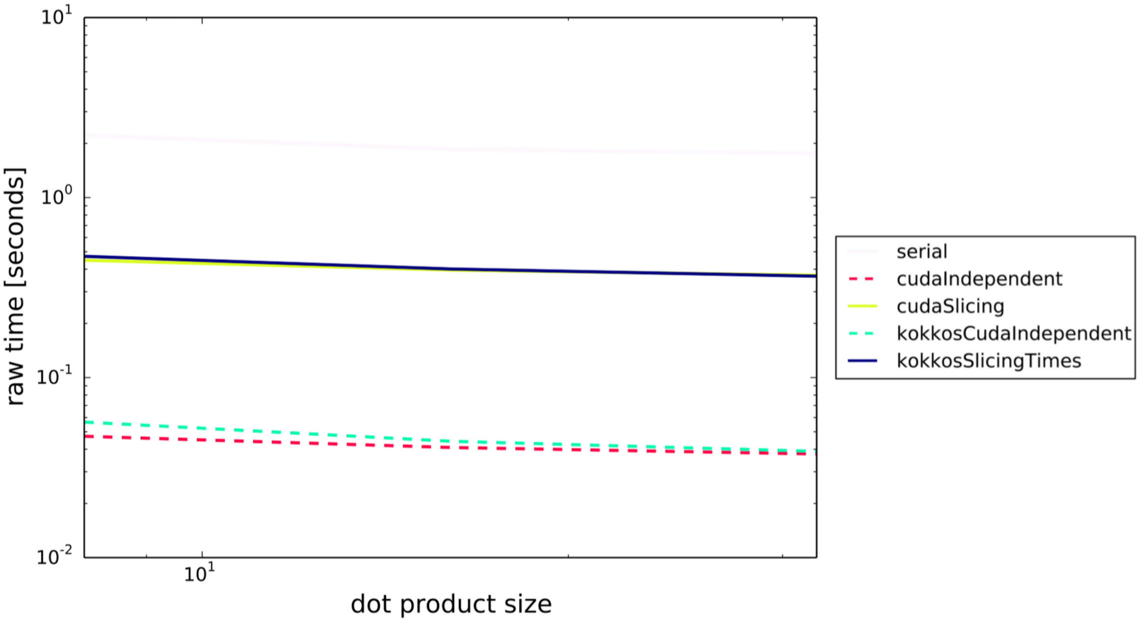


Figure 4.2 Performance of the “slicing” nested parallelism approach.

4.2 Code Snippets

Another major factor that plays into whether programmers will use a language, feature, or library is code complexity and ease of coding. Thus, we also investigated the usability, readability, and intuitiveness of Kokkos compared to Cuda and OpenMP.

Parallelizing code with Kokkos is significantly more complex than doing the same with OpenMP. However, OpenMP is considerably less flexible than Kokkos; it works only on the CPU, and does not generalize to the GPU. Therefore, in the case of code that needs only to run on the CPU, we would strongly advise OpenMP because of its simplicity. However, that is not the niche that Kokkos is trying to fill.

Cuda, however, requires similar amounts of code to Kokkos. The code snippets presented will point out the differences and similarities directly. First, we will show the data setup step of moving data onto the GPU, and then move to comparing and contrasting the Cuda kernel and the Kokkos functor.

Figure 4.3 shows the setup of the data on the GPU for Cuda. There are three steps in the process: declaring a pointer to the data on the CPU, allocating an array with the correct size on the GPU, then copying the data over to the GPU from CPU (host) memory. This process is relatively simple and self-explanatory. Equivalent Kokkos code is shown in Figure 4.4. Note that in a normal case, the data would already reside in the device View. The process in lines 12 - 25 is not necessary if the data is already on the device.

```
float * leftDataArray = (...) // Not shown populating data array
float * dev_leftDataArray;

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

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

```

Figure 4.3 Code from 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 requires typedefs to define the Views, but the small amount of extra work gives the programmer much more control over the data. The control also comes at the cost of having to use loops to copy the data into the host view instead of simply copying raw memory.

However, this initial work is done only once, and allows the user to change the layout of the data by simply changing the `Kokkos::LayoutRight` to `Kokkos::LayoutLeft`. This is useful in optimizing the data layout for both the CPU and GPU. Overall, Kokkos is more verbose, but also more abstract, as it must perform on both the CPU and GPU while Cuda only runs on the GPU.

In a program's computational portion, Cuda uses a kernel while Kokkos uses a functor. However, for programs doing the same calculation, the parenthesis operator in a Kokkos functor is almost an exact replica of the code in the corresponding Cuda kernel. A Cuda kernel for ContractFieldFieldScalar is shown in Figure 4.5.

The parenthesis operator in the corresponding Kokkos functor is shown in Figure 4.6.

```
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) =
        leftArray[cell*numLeftFields*
        numPoints + lbf*numLeftFields + qp];
    }
  }
}

Kokkos::deep_copy(dev_ContractData_Left, contractionData_Left);

```

Figure 4.4 Code from Kokkos Cuda `ContractFieldFieldScalar`

Although there are more lines of code in the Kokkos functor (the code required to declare the data members and the constructor), the Kokkos code is readable and uncluttered.

The Kokkos functor does not need to calculate each thread's ID, while the Cuda kernel has to use built-in constants such as `blockId.x` and `blockDim.x`. Indexing into a View is easier than indexing into a primitive C-style array, especially when changing the layout of the data between `LayoutLeft` and `LayoutRight` because no code changes need to occur in the functor.

```
__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`

4.3 Personal Experience and Thoughts

Part of our project's goal was to document our experiences using Kokkos, including any issues we run into. As with any new tool or language, we initially had some trouble getting used to Kokkos. However, most of our initial issues disappeared once we gained more experience with Kokkos.

Because our liaison, Dr. Carter Edwards, installed Kokkos on our machine, we are unable provide feedback regarding the process of downloading and installing Kokkos. We did, however, encounter some trouble in compiling and linking against Kokkos. Our difficulties were mostly a consequence of insufficient or outdated documentation of the compiler and linker flags Kokkos requires. The lack of documentation due to Kokkos' newness was a recurring problem for the team.

Another obstacle we found was Kokkos' use of "magic words". For example, Kokkos requires the programmer to typedef `Kokkos::Cuda` or `Kokkos::OpenMP` to `device_type`; any other name would cause an error. Though this is a simple line to add, it is not an obvious requirement, and initially caused use some confusion. As with the compiler flags, formal documentation would have helped in this situation; instead, we relied almost exclusively on example programs. In Kokkos' favor, however, the fact that we were able to write all of our programs by following a few examples shows Kokkos' intuitiveness. Overall, we found Kokkos' basic philosophy and structure to be elegant and well-designed.

The main difficulties the team encountered with Kokkos was its lack of documentation. While this is expected in a young project, poor documentation made using Kokkos much more difficult. We expect this to improve in the future, making Kokkos significantly more attractive to users.

As a whole, our team's experience with Kokkos has been positive. It offers a powerful, architecture-independent alternative to other multithreading solutions. Kokkos code is much easier to port between the CPU, GPU, and Xeon Phi and Kokkos Views are powerful abstractions that allow a programmer to easily change the layout of the data. Kokkos functors are cleaner and more readable than Cuda kernels, and the fact that Kokkos is a C++ library and not a new language adds simplicity. Changes we would like to see in Kokkos include View layout options other than `LayoutRight` and `LayoutLeft` and the use of fewer "magic words". Kokkos would also greatly benefit from more documentation as well as better commenting in the example code.

Chapter 5

Our Performance

The original 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 in a different Trilinos library. However, we have instead focused only on the tensor manipulation 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. Before this project, no member of our team had written performance-oriented parallel code, forcing us to spend much of the first semester learning the concepts, techniques, and languages of the field. However, even as we grew more familiar with parallelizing high-performance code, we also ran into a number of issues with Kokkos. Because Kokkos is still a young project, at the beginning of the semester it had very little documentation, as well as a few other issues that gave us difficulties. Over winter break, we received a newly updated version of Kokkos that included fixes for some of these issues, along with some new features.

The second reason we did not move on to another package was the Kokkos update we received over winter break. The update introduced new functionality, allowing us to write team-oriented parallel code and implement algorithms such as team reductions. The new Kokkos package also gave us access to shared memory on the GPU. At this point, rather than merely writing flat parallel versions of other kernels, we decided to focus more heavily on general parallelization techniques using Kokkos teams for tensor contractions, taking the new Kokkos features into account. Thus, because we chose to focus on team techniques, we spent the spring semester implementing, testing, and plotting results from from various algorithms as applied to the tensor manipulations library instead of moving to a different library.

Overall, although our goals diverged from our initial goals as laid out in our statement of work, we accomplished a significant amount of exploratory work this semester. We believe that the work we have accomplished will be useful to Sandia National Laboratories as they continue along the path of making their codebase thread-scalable.