Mathematics Clinic

Final Report for
Sandia National Laboratories

Parallelizing Intrepid Tensor Contractions Using Kokkos

April 5, 2015

Team Members

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

Advisor

Jeff Amelang

Liaison

H. Carter Edwards

Abstract

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

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

Contents

List of Figures

List of Tables

Acknowledgments

Chapter 1

Sandia National Laboratories

1.1 Background

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.

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

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

2.4 ContractDataDataVector

2.5 ContractDataDataTensor

2.6 ContractDataFieldScalar

2.7 ContractDataFieldVector

2.8 ContractDataFieldTensor

2.9 ContractFieldFieldScalar

2.10 ContractFieldFieldVector

2.11 ContractFieldFieldTensor

Chapter 3

Parallelism

3.1 Flat Parallelism

3.2 Reduction

3.3 Slicing

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

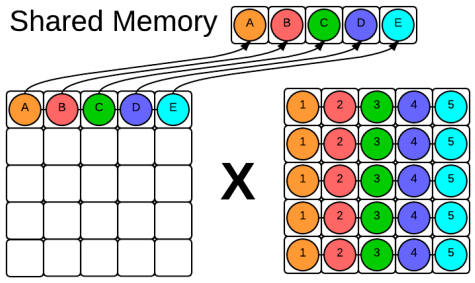


Figure 3.1 Demonstration of memory accesses for a slicing implementation of ContractFieldFieldScalar

On the left, we have the first of the two input matrixes, who's first row elements are labeled $A - E$. On the right we have the second of the two inputs. For the sake of simplicity, assume that we have on block of five threads which are labeled by color. Each of the threads reads in one of the elements on the right and copies it into shared memory. In cases where the number of elements per contraction (row on the left) is unequal to the number of contractions (columns on the right), we set the number of threads per block equal to the number of contractions. This causes threads to either sit idle or loop multiple times when reading the elements on the left into shared memory, but this is clearly more efficient than forcing threads to compute more than one element.

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

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

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

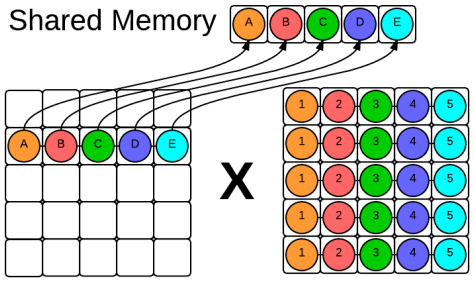


Figure 3.2 Demonstration of memory accesses for the second block of a slicing implementation of ContractFieldFieldScalar

have C matrices. Additionally, there will be $\mathcal{R}$ threads per block.

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

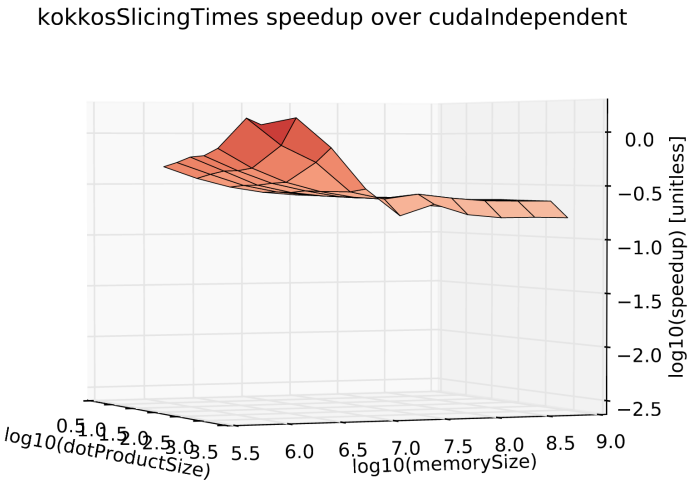
```
extern __shared__ float sliceStorage[];
const unsigned int col = threadIdx.x;
unsigned int currentBlock = blockIdx.x;
unsigned int numBlocks = numBasis*numCells;

syncthreads();
const unsigned int cell = currentBlock / numBasis;
const unsigned int row = currentBlock - cell * numBasis;

for (unsigned int p = threadIdx.x; p < contractionSize; p += blockDim.x) {
   sliceStorage[p] = dev_contractionData_Left[cell*numBasis*contractionSize + row*contractionSize + p];
}
syncthreads();

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

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

Figure 3.3 Code from slicing algorithm on `ContractFieldFieldScalar`

The main advantage of this approach is that it is easily generalizable to tensor contractions of higher dimensions. Unlike tiling, which is significantly less intuitive in higher dimensions, it is easy to implement slicing in higher dimensions by loading a larger slice into shared memory. Because of its reliance on shared memory, there are many use cases in which we would expect slicing to perform poorly. Intuitively, slicing is reliant on large contraction sizes to produce speedup because in situations where the number of threads per block is low it is unable to saturate the GPU. While this problem can be remedied by increasing the number of contractions per block, it can introduce problems with shared memory. Since shared memory is limited by nature, slicing has to balance the amount of work per block with the amount of shared memory available to that block.

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

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



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

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

the data exists for this here: https://github.com/Sandia2014/kokkos-intrepid/tree/slicing+tiling/ContractFieldFieldTensor, will make graph later

We see that while slicing performs better, it is still eclipsed by indepen-

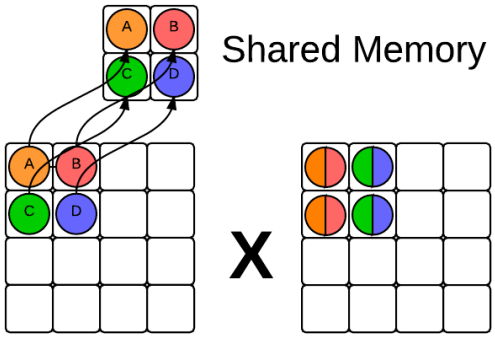


Figure 3.4 Demonstration of memory accesses for a tiling implementation of ContractFieldFieldScalar

dent cuda.

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

3.4 Tiling

The final parallelization technique we used for these tensor contractions was tiling. This technique is similar to the tiled technique for matrix multiplication used in serial operations. Instead of relying on the cache to retain the relevant pieces of information, however, we use shared memory to explicitly store the data we care about. Once again, we will explain this algorithm by example. Consider one of the matrix multiplications in ContractFieldFieldScalar shown above.

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

Tiling can be viewed as a more specialized version of slicing, since they

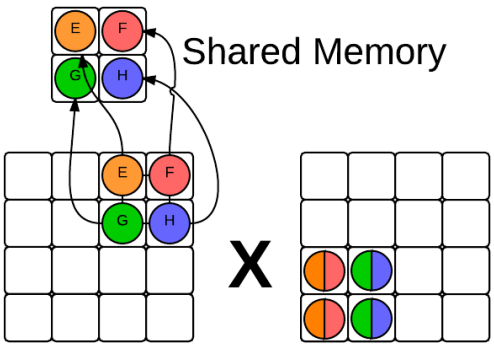


Figure 3.5 Demonstration of memory accesses for a tiling implementation of ContractFieldFieldScalar

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

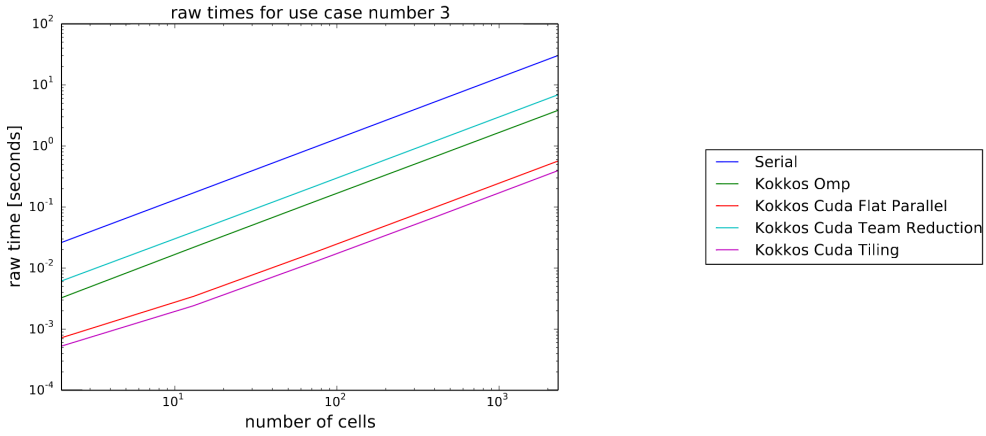
Unfortunately, it is much less clear how exactly to tile in multiple dimensions. Unlike slicing, there seem to be multiple distinct ways of approaching the problem. One could create "tiles" with dimension equal to the contraction size, or any number less than the contraction dimension by unrolling the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

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

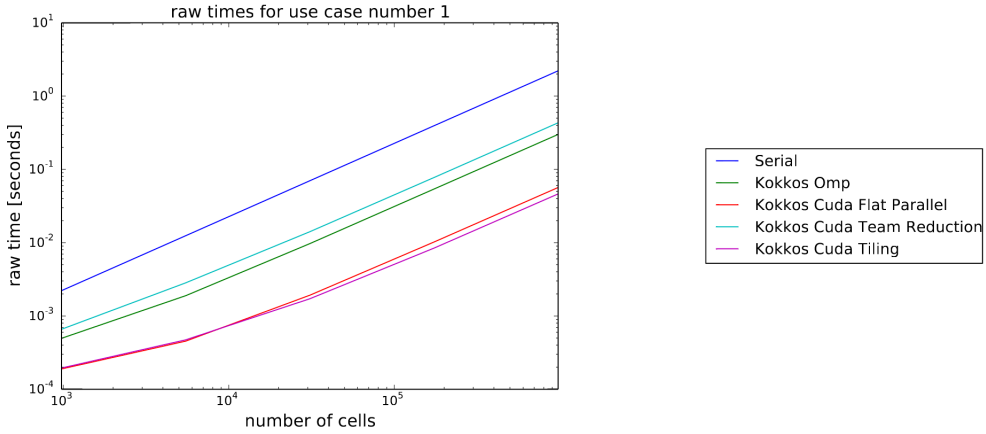
Thus far in our research, we have found tiling to be the most effective algorithm for realizing parallel speedup.

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

const unsigned int numberOfTiles = numCells * numberOfVerticalTiles * numberOfVerticalTiles;

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

unsigned int resultTileIndex = blockIdx.x;

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

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

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

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

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

    const unsigned int resultIndex = row * numBasis + col;

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

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

Figure 3.6 Code from tiling algorithm on `ContractFieldFieldScalar`



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



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

Chapter 4

Experience with Kokkos

4.1 Performance

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

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

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

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

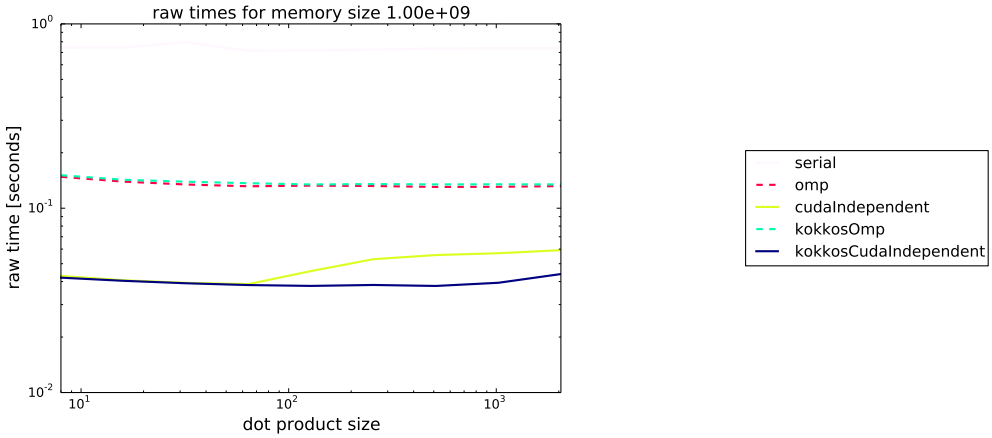


Figure 4.1 This graph plots the performance of Kokkos Cuda, Cuda, Kokkos OpenMP, and OpenMP for ContractDataDataScalar with a memory size of 1 GB. The y-axis is time in seconds, so closer to 0 is better. The x-axis plots different contraction sizes in order to compare multiple points.

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

KokkosCuda versus Cuda, on the other hand, has some big differences. They are identical for the smaller problems but diverge a significant amount for bigger problems. One of the theories that we have as to why this trend exists is that Kokkos may be launching a different amount of blocks than the number of blocks Cuda launches. We believe the reason this doesn't effect the smaller problem sizes is because the number of blocks that needs to be launched is smaller than the bigger problem sizes, so the upper limit of number of blocks launched is not reached, but clearly is in the bigger problem sizes.

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

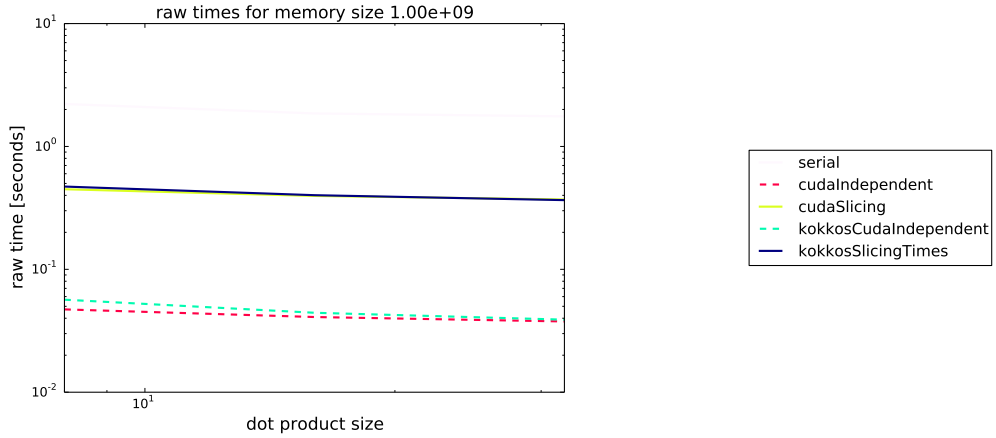


Figure 4.2 This graph shows performance differences (or similarities in our case) of the nested parallelism approach slicing.

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

Overall, after seeing a handful of graphs that show Kokkos performs almost identically to Cuda and to OpenMP, we accepted that Kokkos is not

adding any unneeded overhead. As stated before, there may be slight differences due to compiler optimizations, but Kokkos seems to perform identically to the other multithreading solutions.

4.2 Snippets

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

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

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

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

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

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

Figure 4.3 Code from Cuda `ContractFieldFieldScalar`

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

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

ContractionData_Host contractionData_Left =
        Kokkos::create_mirror_view(dev_ContractData_Left);

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

Figure 4.4 Code from Kokkos Cuda `ContractFieldFieldScalar`

The Kokkos code first defines and creates the device and host Views. One of the major differences compared to Cuda is that Kokkos uses its own data structure, a View, instead of an array. This is why we need to use typedefs to define the Views, but the extra work (which honestly is not much of a hassle) allows the programmer much more control over the data. The control also comes at the cost of having to use for loops to copy the data into the host view instead of being able to do a Memcpy. However, this is all initial work that needs to be done once, while the benefit of being able to change the layout of the data by changing the Kokkos::LayoutRight to Kokkos::LayoutLeft is very useful, especially since this allows the programmer to optimize data layout for both the CPU and GPU. Overall, Kokkos' and Cuda's data setup have different philosophies, which makes sense because Kokkos needs to be easily optimized for both the CPU and GPU while Cuda only runs on the GPU.

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

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

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

4.3 Personal Experience and Thoughts

A task of the project was to document our experiences and thoughts about Kokkos, including any issues that we have run into. Using new tools and learning new syntax always has its tough periods, and getting used to

Kokkos definitely had some periods where we had no idea why a program was not compile or giving an incorrect answer (especially in the beginning), but after the initial learning curve everything seemed to flow pretty well and make sense.

Our team has never actually been responsible for installing Kokkos on our machine, instead our liaison, Dr. Carter Edwards, did that for us, so we are unable to talk about the difficulties of downloading and installing the Kokkos library on our machine, but we did have lots of trouble trying to compile and linking against Kokkos originally. This was due to the fact that the same flags need to be used when installing and compiling and linking against Kokkos. However, since we did not install Kokkos ourselves and the documentation showing how to compile and link against Kokkos used different flags than what were used during our installation, we struggled for a while. Already this shows how Kokkos' documentation is not as developed as one would like, which we will bring up later, but it is understandable since Kokkos is new.

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

As previously mentioned, Kokkos has very little documentation. For any emerging technology it is understandable that the creators choose to focus on functionality instead of documentation, but the documentation needs to catch up at some point. The examples were very helpful in getting us to our

end goal of working code, but examples are not as helpful in understanding what exactly is happening, the meaning behind some portions of code, or why certain code is necessary. Documentation would have also been helpful in seeing the default values for functions and Views, as well as the other arguments that could have been passed instead. There were many times we tried to use Google to find information about Kokkos, but many times the information would point to uncommented pieces of code, which is not always helpful in determining what is going on. Overall we believe the documentation for Kokkos needs to improve in order for new users to get past the initial learning curve and spread the word about Kokkos.

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

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

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

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

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

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

Figure 4.5 Code from Cuda `ContractFieldFieldScalar`

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

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

Figure 4.6 Code from Kokkos Cuda `ContractFieldFieldScalar`

Chapter 5

Performance