Generating Efficient Tensor Contractions for GPUs

Thomas Nelson*, Axel Rivera[†], Prasanna Balaprakash[‡], Mary Hall[†], Paul D. Hovland[‡], Elizabeth Jessup* and Boyana Norris[§]
* *Department of Computer Science, University of Colorado, Boulder, CO 80309*
Email:thomas.nelson@colorado.edu, jessup@cs.colorado.edu
[†] *School of Computing, University of Utah Salt Lake City, UT 84112*
Email: elriv@cs.utah.edu, mhall@cs.utah.edu
[‡] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439*
Email: hovland@mcs.anl.gov, pbalapra@mcs.anl.gov
[§] *Department of Computer and Information Science, University of Oregon, Eugene, OR 97403*
Email: norris@cs.uoregon.edu

***Abstract*—Many scientific and numerical applications, including quantum chemistry modeling and fluid dynamics simulation, require tensor product and tensor contraction evaluation. Tensor computations are characterized by arrays with numerous dimensions, inherent parallelism, moderate data reuse and many degrees of freedom in the order in which to perform the computation. The best-performing implementation is heavily dependent on the tensor dimensionality and the target architecture. In this paper, we map tensor computations to GPUs, starting with a high-level tensor input language and producing efficient CUDA code as output. Our approach is to combine tensor-specific mathematical transformations with a GPU decision algorithm, machine learning and autotuning of a large parameter space. Generated code shows significant performance gains over sequential and OpenMP parallel code, and a comparison with OpenACC shows the importance of autotuning and other optimizations in our framework for achieving efficient results.**

***Keywords*-tensor contraction, autotuning, GPUs**

I. Introduction

Computer architecture is undergoing a significant period of exploration to find new ways for continued performance gains while maintaining energy efficiency and reliability. The result is a diverse landscape of architectures that incorporate features such as massive socket-level parallelism, accelerators, and deep memory hierarchies. Developers of high-performance computational science applications are thus faced with the challenge of maintaining performance portability across diverse architectures.

We are pursuing an approach to performance portability that uses a domain-specific language (DSL) to specify high-level semantics, a transformation and code generation framework to map the DSL to an architecture, and autotuning and machine learning to search among many possible code variants. In this paper, we focus on GPU-based computation of tensor contractions, a multidimensional generalization of matrix-matrix multiplication. Such computations arise frequently in computational science applications. We focus on specific instances from computational fluid dynamics using the spectral element method and electronic structure modeling using coupled cluster theory.

As compared to other DSLs for tensor contraction [4, 27], we focus on a class of tensor computations with small dimensions on GPU architectures. In such cases, mapping the problem to use highly-tuned linear algebra libraries will not achieve high performance as these libraries are optimized for large matrices. Our approach was driven by a desire to improve specific tensor problems not addressed by current tools, but we also view this work as an exemplar for developing highly-tuned applications specialized for individual architectures starting with a mathematical representation of the problem in a DSL. This paper makes the following contributions: (1) a modular domain-specific system design; (2) a new decision algorithm for generating optimized implementations for GPUs; (3) a new machine learning approach customized to the resulting search space of GPU implementations; and, (4) GPU performance results that show significant gains over sequential, OpenMP and OpenACC code. The modular system Barracuda starts from a mathematical representation, generates a large search space of possible implementations, and uses autotuning and machine learning to generate highly-optimized code targeting a specific architecture.

Section II of this paper provides an overview of the problem domain and describes the Barracuda system. Section III describes the high-level module Optimizing Compiler with Tensor OPeration Intelligence (OCTOPI), a DSL and optimizations for tensor contractions. Section IV describes the intermediate module Tensor Contraction Representation (TCR), which performs code generation from a tensor-specific representation and encodes the autotuning search space. Section V presents the new machine learning algorithm SURF, which is used to search the large space of code transformations. Section VI describes the experimental design and results on three generations of NVIDIA GPUs. Section VII describes related work. Section VIII is a conclusion.

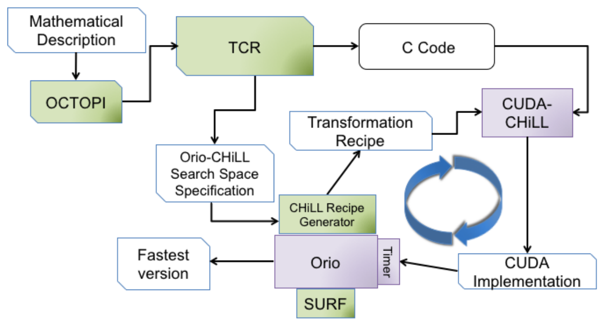

Figure 1. Barracuda framework for tensor contraction code generation.

II. Overview

A. Background

Tensors are a multidimensional generalization of matrices and are a natural way to express many computations arising in scientific computing. The rank of a tensor is the number of dimensions; a vector is a rank-1 tensor and a matrix is a rank-2 tensor. Two types of tensor computation are particularly common: tensor decompositions, a computation frequently used in data analysis, and tensor contractions, a multidimensional analog of matrix-matrix multiplication used in coupled cluster electronic structure calculations [5, 27], in spectral element discretizations of partial differential equations (PDEs) [9], and as a building block for tensor decompositions. In this paper we focus on tensor contractions, which can be expressed as summation along one or more tensor dimensions.

For convenience, we represent tensor contractions using the Einstein summation convention, where whenever the same index appears twice in an expression, once as a superscript and once as a subscript, there is an implied summation over all values of an index. Thus, the vector inner product is represented as $y = u_i v^i$, the matrix-vector product as $y^i = A^i_j x^j$, and the matrix-matrix product as $C^i_k = A^i_j B^j_k$. The contraction of a rank-3 tensor with another rank-3 tensor along one dimension results in a rank-4 tensor

$$C^{ij}_{lm} = A^{ij}_k B^k_{lm} \equiv \sum_k A^{ij}_k B^k_{lm},$$

and the contraction of a rank-3 tensor with another rank-3 tensor along two dimensions results in a rank-2 tensor

$$C^i_l = A^i_{jk} B^{jk}_l \equiv \sum_j \sum_k A^i_{jk} B^{jk}_l.$$

We are interested in computing multidimensional tensor contractions as efficiently as possible. We focus on scenarios featuring computations over thousands of identically-sized small tensors (size O(1)–O(10) in each dimension) because they occur naturally in the spectral element method [9] and provide a building block for computations with large tensors in coupled clustered computations [5, 27].

Consider the case of a pth-order spectral element discretization of a PDE on a mesh with N elements. Each mesh element requires computing tensor contractions of the form

$$V_{ij} = A^l_j B^k_i U_{kl} \equiv \sum_{k=0}^{p} \sum_{l=0}^{p} A^l_j B^k_i U_{kl}$$

in two dimensions or

$$\begin{aligned} V_{ijk} &= A^l_k B^m_j C^n_i U_{lmn} \\ &= \sum_{l=0}^{p} \sum_{m=0}^{p} \sum_{n=0}^{p} A^l_k B^m_j C^n_i U_{lmn} \end{aligned} \qquad (1)$$

in three dimensions. A naive implementation of the two-dimensional contraction requires O(p^4) operations (O(p^2) for each of the p^2 members of V_{ij}). However, this approach ignores redundant subcomputations across columns and rows of V. One can instead compute $W^l_i = B^k_i U_{kl}$ followed by $V_{ij} = A^l_j W_{il}$ at a cost of O(p^3) operations. A similar reorganization of the three-dimensional computation reduces the cost from O(p^6) operations to O(p^4) operations. Tools such as the Tensor Contraction Engine (TCE) [5, 16] and libtensor [10] seek to reorganize tensor contractions in this fashion to minimize the number of floating-point operations [16]. In this paper, we combine such reorganizations with an autotuning compiler so that the search space of possible implementations is much richer.

B. Code Generation

We use the Barracuda system illustrated in Figure 1 to generate many possible variants of a tensor computation and to identify the best-performing implementation. Figure 2 illustrates the input at each stage and final output. The user inputs to OCTOPI a high-level representation of a computation that resembles mathematical tensor notation, as shown in Figure 2(a). This example corresponds to Eqn.(1) above and is adapted from the computation $\underline{v} = C\underline{u}$ on p. 168 of [9]. To this input, OCTOPI applies tensor-specific optimizations to reorganize the computation, as described in Section III, to generate a set of inputs to TCR. For Equation(1), OCTOPI generates fifteen different versions. While six versions all perform the same amount of floating-point computation, their performance on an Nvidia GTX 980 (Maxwell) varies by as much as 9%. The TCR input for what is ultimately the best-performing version is shown in Figure 2(b). TCR generates code annotations to existing tool Orio [15, 21] specifying which transformations and autotuning search space should be explored, an excerpt of which is shown in Figure 2(c). Each code variant is then generated automatically using CUDA-CHiLL, a source-to-source compiler transformation and code generation framework that transforms sequential loop nests to high-performance GPU code [18]. In CUDA-CHiLL, code variants are described by *transformation recipes*, which encode the transformations to be applied to the original sequential computation;

```
V[i j k] += A[l k] B[m j] C[n i] U[l m n]
```

(a) Tensor contraction input to OCTOPI.

```
ex
access: linearize
define:
N = J = M = I = L = K = 10
variables:
temp3:(J,I,L)
A:(L,K)
C:(N,I)
B:(M,J)
U:(L,M,N)
V:(I,J,K)
temp1:(I,L,M)
operations:
temp1:(i,l,m) += C:(n,i)*U:(l,m,n)
temp3:(j,i,l) += B:(m,j)*temp1:(i,l,m)
V:(i,j,k) += A:(l,k)*temp3:(j,i,l)
```

(b) Input to TCR.

```
 def performance_params {
   ...
   param PERMUTE_2_TX2[] = [’m’];
   param PERMUTE_2_TY2[] = [’i’,’1’,’m’,’l’];
   param PERMUTE_2_BX2[] = [’i’,’m’,’l’];
   param PERMUTE_2_BY2[] = [’i’,’1’,’m’,’l’];
   param UF_2[] = [1,2,3,4,5,6,7,8,9,10];
}
/*@ begin CHiLL (
...
cuda(2,block={PERMUTE_2_BX2,PERMUTE_2_BY2},
       thread={PERMUTE_2_TX2,PERMUTE_2_TY2})
registers(2,"n","V")
unroll(2,"n",UF_2)
   ) @*/
...
 for (i=0; i<I; i++){
   for (l=0; l<J; l++){
    for (m=0; m<K; m++){
     for (n=0; n<L; n++){
      V[i*J*K + l*K + m ] = V[i*J*K + l*K + m ] +
           (A[n*K + m ] * temp3[l*I*L + i*L + n ]);
     }
    }
   }
```

(c) Search space excerpt for CUDA-CHiLL and Orio.

```
__global__ void ex_GPU_2
       (double *V,double *A,double *temp3)
{
  // ... declarations ...
  nv2 = V[ty * 100 + bx * 10 + tx] ;
  for (n = 0; n <= 6; n += 3) {
    nv2 = nv2 + A[n * 10 + tx]
          * temp3[bx * 100 + ty * 10 + n];
    nv2 = nv2 + A[(n + 1) * 10 + tx]
          * temp3[bx * 100 + ty * 10 + (n + 1)];
    nv2 = nv2 + A[(n + 2) * 10 + tx]
          * temp3[bx * 100 + ty * 10 + (n + 2)];
  }
  nv2 = nv2 + A[9 * 10 + tx]
        * temp3[bx * 100 + ty * 10 + 9];
  V[ty * 100 + bx * 10 + tx]  = nv2;
}
```

(d) Optimized CUDA (partial) output.

Figure 2. Example representing Eqn.(1), displaying one of three loop nests generated automatically by Barracuda.

in Barracuda, these are generated automatically, with an example in Figure 2(c). While Orio supports code transformations, in this work, we rely on CUDA-CHiLL for all code transformations and use Orio to navigate the search space arising from OCTOPI and TCR. We implement SURF, a model-based search method, within Orio to prune the search space, and focus on a small subset of the myriad possibilities of code variants. The resulting tuned GPU code is excerpted in Figure 2(d). In the example, three kernels are generated and individually optimized, corresponding to the three summations in the TCR input, but the data remains on the GPU across these calls. The next sections describe OCTOPI, TCR and SURF, the new modules of Barracuda.

III. OCTOPI Input and Optimization

The input to OCTOPI is a sequence of summation statements. The argument to the sum in Figure 2(a) is the summation indices, and the right hand side expression is computed for the entire (implicit) range for each index. The user can optionally specify the index dimension or a range of dimensions so that the framework can specialize the optimizations it applies for specific tensor sizes.

OCTOPI takes a two-stage approach to optimizing tensor contractions. First, it analyzes the tensor for possible high-level transformations that might improve performance; it then passes each of these transformed variants to TCR. The most important transformation OCTOPI applies at a high level has been previously called strength reduction [4], and involves algebraic simplification to reduce the amount of computation. The pseudocode for the OCTOPI algebraic transformation enumeration is in Algorithm 1. The input to the algorithm is a set of multiplication terms, and the algorithm enumerates possible reorderings of those terms, taking advantage of commutativity and associativity. We now

Algorithm 1 Creating a valid OCTOPI algebraic transformation

```
Input: a set $T$ of $n$ terms, with $T_i$ having indices $I(T_i)$
1  $c \leftarrow 0$
2  $d \leftarrow n$
3  while $n > 1$ do
4      $d \leftarrow d + 1$
5      for $i \leftarrow$ index occurring only in $T_a$ do
6          Create term $T_d[I(T_a) - \{i\}] = \sum_i T_a$
7          $T \leftarrow T \cup \{T_d\} - \{T_a\}$
8          $c \leftarrow a$
9      end for
10     Choose any $a, b$ such that $a < b, b > c$
11     To enumerate exhaustively, perform depth-first
       search on these choices
12     Create term $T_d[I(T_a) \cup I(T_b)] = T_a \times T_b$
13     $T \leftarrow T \cup \{T_d\} - \{T_a, T_b\}$
14     $c \leftarrow b$
15 end while
```

give an example of the performance advantages that can come from this computational reordering. We consider again the tensor example from Figure 2(a).

```
V[i j k] =
 Sum([l m n], A[l k] * B[m j] * C[n i] * U[l m n])
```

The naive implementation of this code creates a 6-deep nested for loop.

```
for i
  for j
    for k
      for l
        for m
          for n
            V[i j k] +=
              A[l k] * B[m j] * C[n i] * U[l m n]
```

These loops can be interchanged, creating $N!$ loop choices for N indices. In the Equation 1 example, there are 720 total arrangements of these loops. Some open opportunities for moving code outside of loops, sometimes called strength reduction [14]. Strength reduction is a reorganization of the sums that takes advantage of partial sums to reduce the total computation. OCTOPI carries out strength reduction by finding all subexpressions that have a smaller iteration space than the full computation. It takes advantage of commutativity to find all reorganizations. In our example, the following are the possible subexpressions for the above input.

```
A[l k] * B[m j] * C[n i] * U[l m n]  →

A[l k] * B[m j]          (l k m j)
B[m j] * C[n i]          (m j n i)
A[l k] * C[n i]          (l k n i)
A[l k] * U[l m n]        (l k m n)
B[m j] * U[l m n]        (m j l n)
C[n i] * U[l m n]        (n i l m)
```

The set of subexpressions depends on the size of the tensor, but for the computations we encounter, it is possible to enumerate them exhaustively. Because each of these subexpressions involves four indices, each requires N^4 operations. For example, if we begin with the last subexpression in the list above, we can replace the single loop nest of order N^6 by three N^4 loop nests, thus reducing the amount of computation and improving performance. The OCTOPI output follows.

```
T1[i l m] += C[n i] * U[l m n]
T2[j i l] += B[m j] * T1[i l m]
V[i j k] += A[l k] * T2[m j]
```

This output corresponds to the following pseudocode.

```
for i
   for l
      for m
          for n
            T1[i l m] += C[n i] * U[l m n]
for j
   for i
       for l
           for m
                T2[j i l] += B[m j] * T1[i l m]
for i
    for j
        for k
            for l
                V[i j k] += A[l k] * T2[m j]
```

We can additionally incorporate loop fusion, with more fusion available if we reorder the loops.

```
for i
    for l
        for m
            for n
                T1[i l m] += C[n i] * U[l m n]
            for j
                T2[j i l] += B[m j] * T1[i l m]
        for k
            for j
                V[i j k] += A[l k] * T2[m j]
```

This variant performs the same number of operations, but has better memory usage and enables more optimizations for CUDA-CHiLL. Choosing different subexpressions to evaluate first will result in different fusion opportunities and sometimes different operation counts. Performance depends on data layout in memory and subsequent transformations. In our example, OCTOPI generates and sends all versions to CUDA-CHiLL for autotuning.

The transformations described here are extremely difficult at the C source level: it requires complex loop interchange, strength reduction, and loop fusion. By considering the operations at the tensor level, we can easily enumerate the possibilities and expose the search space.

IV. TCR and Generating the Search Space

The variants generated by OCTOPI are expressed in an an intermediate representation that is input to a lower-level tool (TCR) that encodes a set of parameterized code variants ultimately used by CUDA-CHiLL to generate GPU code. From the representation in the code example in Figure 2(b), TCR creates a *for* loop for each different loop index listed in the operation and uses the tensor equation to generate the statement. In addition, this code is accompanied by a collection of transformations to be applied in CUDA-CHiLL to define the autotuning search space for this specific variant. The search space explored is a simplification of Khan et al.'s algorithm [17, 18]. The algorithm finds the thread and block decomposition and data placement in different levels of the memory hierarchy, along with additional transformations to optimize the thread program.

Data dependence analysis is used to determine the safety of parallelization and other reordering transformations. In general, data dependence analysis requires pairwise comparison of access expressions to the same array, where one of the accesses is a write, within the context of the iteration space of the common loops in which the two accesses are nested [1]. While CUDA-CHiLL incorporates this general approach to dependence analysis, we can rely on a simplified dependence analysis specialized to the domain of tensor contractions that can be specified by TCR's mathematical description. Dependences can be carried only by loops with indices present in the right-hand side but not in the left-hand side of a tensor operation. Loops corresponding to all remaining indices may be executed in parallel. We also analyze the memory access patterns for each of the input tensors.

We use *contiguous tensors* to describe array references whose index expressions refer to loops in the same order as they appear in the code; that is, the array is accessed in memory order (assuming row-major layout). Contiguous tensors are desirable, as they lead to data access orders that achieve global memory coalescing and reuse in the GPU's caches. Nevertheless, in most tensor contraction computations, not all tensors are accessed as contiguous tensors as there does not exist a loop order that is optimal for all data.

We first generate the search space for thread and block decomposition on the GPU. Thread_X and Thread_Y refer to the X and Y thread dimensions on the GPU; Block_X and Block_Y refer to the X and Y block dimensions on the GPU. The X dimension is the leading one, such that adjacent X threads with the same Y value are usually mapped to adjacent GPU cores. Therefore, we choose as candidates for Thread_X any loop such that adjacent elements on an input tensor are accessed by adjacent threads so as to achieve global memory coalescing. Potential choices for Thread_Y, Block_X, and Block_Y are selected by the following rules:

- Select parallel loop indices from the *contiguous tensors* from innermost to outermost loops.
- If the *contiguous tensors* have fewer than four parallel loops, then start selecting parallel indices from the *non-contiguous tensors* from outer to inner.

The search space also consists of different loop orders, which can be realized using loop permutation. Any loops that are inside the GPU kernel that improve memory layout of inner dimensions are considered as candidates for loop permutation. A final optimization included in the search space is the unrolling of inner loops to reduce control flow, enhance register reuse and increase instruction-level parallelism. A number of unroll factors are considered, but these are relatively small because of the small loop iteration counts. Additionally, included with these optimizations, the compiler always applies scalar replacement to explicitly copy the output tensor variable to a scalar temporary so that it is accessed in a register; it is copied back to global memory only at the end of a thread's computation to reduce accesses to global memory.

An earlier version of this decision algorithm created a smaller, pruned search space, which is a subset of the one used in [25]. This prior work also provides additional experiments to motivate the search space we derive. The one example version shown in Figure 2(c) produces a search space that consists of 4 different Thread_Y values (one of which is `1` indicating a 1-dimensional thread block), which can be combined with three possible Block_X dimensions and four possible Block_Y dimensions. The meaning of Orio's `PERMUTE` is that one possible value is selected at a time, and it cannot be the same as a value selected for another parameter in the same block.

V. Search Space Exploration

Enumerating all possible code variants of the autotuning problem posed by OCTOPI and TCR can be computationally prohibitive: for a given tensor computation, OCTOPI can generate a number of *tensor variants*, where each has a number of parameters introduced by TCR for thread, block decomposition, and unroll that in turn produce a large number of *code variants*. Given the decomposition parameters (encoded by `PERMUTE`) and unroll parameters, the number of code variants grow exponentially with respect to the number of parameters. For Lg3t (see Table I), we have 512,000 possible tensor-code variants for empirical evaluation. A promising approach to overcome this difficulty is through the use of a search algorithm that finds high-performing code variants while examining relatively few variants. However, designing a search algorithm to navigate the search space is quite challenging from a mathematical optimization perspective because (decomposition) the resulting variants do not admit a natural ordinal relationship and (unroll) integer parameters cannot be relaxed.

We customize and adapt the model-based search algorithm proposed in [2] to the search problem. We refer to our model-based search as SURF (**s**earch **u**sing **r**andom **f**orest, where random forest is the modeling algorithm adopted within the search). We sample a small number of parameter configurations, empirically evaluating the corresponding code variants to obtain the corresponding performance metrics, and fitting a surrogate model over the input-output space. The surrogate model is then iteratively refined by

Algorithm 2 Pseudo-code for SURF

```
Input: configuration pool $\mathcal{X}_p$, batch size $bs$, max evaluations $n_{\max}$

1  $\mathcal{X}_{\text{out}} \leftarrow$ sample $\min\{bs, n_{\max}\}$ distinct configurations from $\mathcal{X}_p$
2  $\mathcal{Y}_{\text{out}} \leftarrow$ Evaluate_Parallel($\mathcal{X}_{\text{out}}$)
3  $\mathcal{M} \leftarrow$ fit($\mathcal{X}_{\text{out}}$, $\mathcal{Y}_{\text{out}}$)
4  $\mathcal{X}_p \leftarrow \mathcal{X}_p - \mathcal{X}_{\text{out}}$
5  for $i \leftarrow bs + 1$ to $n_{\max}$ do
6      $\mathcal{Y}_p \leftarrow$ predict($\mathcal{M}$, $\mathcal{X}_p$)
7      $x_i^{bs} \leftarrow$ select $bs$ configurations from $\mathcal{X}_p$ with the best performance in $\mathcal{Y}_p$
8      $y_i^{bs} \leftarrow$ Evaluate_Parallel($x_i^{bs}$)
9      retrain $\mathcal{M}$ with $(x_i^{bs}, y_i^{bs})$
10     $\mathcal{X}_{\text{out}} \leftarrow \mathcal{X}_{\text{out}} \cup x_i$; $\mathcal{Y}_{\text{out}} \leftarrow \mathcal{Y}_{\text{out}} \cup y_i$ /* $\cup$ denotes set union */
11     $\mathcal{X}_p \leftarrow \mathcal{X}_p - x_i^{bs}$ /* $-$ denotes set difference */
12 end for

Output: $x \in \mathcal{X}_{\text{out}}$ with the best performance in $\mathcal{Y}_{\text{out}}$
```

obtaining new output metrics at unevaluated input configurations predicted to be high-performing by the model. The main extension handles the decomposition parameters by applying a preprocessing technique called feature binarization [6], where they are transformed into binary vectors to enable surrogate modeling.

Algorithm 2 shows the pseudocode of the model-based search algorithm. The algorithm takes as input a set $\mathcal{X}_p$ of unevaluated configurations, the stopping criterion of maximum number $n_{\max}$ of allowed evaluations, and batch size bs that determines the number of concurrent evaluations at each iteration. In the initialization phase of the algorithm, bs configurations are sampled at random and evaluated in parallel to obtain their corresponding performance metrics. These points are then used as a training set to build a predictive model for performance. The iterative phase predicts the performance of all remaining unevaluated configurations using the models, evaluating bs configurations with best predicted performance, and retraining the model with the evaluation results. The batching allows for a higher degree of parameter space exploration and increases the probability of finding high-quality configurations in fewer iterations [3]. This algorithm reduces the time for the search needed to find high-quality parameter configurations.

We deploy statistical machine learning methods [6] for building surrogate models. In particular, we choose randomized trees [12], a state-of-the-art machine learning algorithm, due to their ability to handle the binarized parameters using recursive partitioning and to model nonlinear interactions among the parameters.

Name	Description
Spectral Element	
Eqn.(1)	example from Figure 2
Lg3	local_grad3 from Nekbone
Lg3t	local_grad3t from Nekbone
Nekbone	Mini-app using optimized Lg3 and Lg3t
Coupled Cluster	
TCE_ex	TCE example tensor[4]
S1 (s1_1-s1_9)	NWChem excerpt: 2 objects with 2&4 dimensions
D1 (d1_1-d1_9)	NWChem excerpt: 2 objects with 4 dimensions
D2 (d2_1-d2_9)	NWChem excerpt: 2 objects with 4 dimensions

Table I
BENCHMARKS USED IN THIS STUDY.

For Lg3t, we ran the model-based search with 100 evaluations, which took 7 minutes (approximately 4 seconds per variant). Assuming the same time per variant, enumeration of 512,000 variants will take approximately 23 days.

VI. PERFORMANCE MEASUREMENTS

We tested the integrated system from Figure 1 on the tensor-contraction computations in Table I. The computations (Eqn.(1), Lg3, Lg3t and TCE_ex) were selected because they allow us to evaluate tensor contraction in isolation, and the Nekbone and NWChem computations let us consider how tensor contractions are used in the context of applications. Nekbone is a 3-dimensional spectral element proxy application derived from Nek5000 [11, 30]. It performs a conjugate gradient loop that operates over a sequence of tensor contractions recast as matrix multiplications, which comprises 60% of the sequential execution time. A problem size of $12 \times 12 \times 12$ was used. These small dimensions result from the small order of the discretization polynomial; as it increases, the time required to converge also increases. NWChem is a software package for quantum chemistry and molecular dynamics simulations [31]. We optimized kernels [7] extracted from the CCSD(T) (coupled cluster theory with full-treatment singles and doubles, and triples estimated by using perturbation theory) computations of NWChem. These kernels are representative of what executes at the socket level, with trip counts of 16 iterations in each dimension, so are appropriate for a single GPU.

We performed experiments on an Intel Haswell CPU and three generations of Nvidia GPUs: TESLA C2050 (Fermi), TESLA K20 (Kepler) and GTX 980 (Maxwell). For the OpenACC results, we used the Portland Group compiler (PGI) version 14.3, but this version does not yet generate code for the GTX 980. The CUDA code was compiled by using the nvcc compiler for CUDA 5.5. For each point (code variant) in the search space, we compute average execution time over 100 repetitions.

A. Individual Tensor-Contraction Computations

We first look at the optimization of the individual tensor-contraction computations summarized in Table VI. This table shows speedup over sequential execution on the Haswell,

		GTX 980		K20		C2050	
	Speedup	GFlops	Search	GFlops	Search	GFlops	Search
Eqn.(1)	0.63×	1.99	3556.0s	1.42	9691.4s	1.89	7111.3s
Lg3	23.74×	42.74	324.8s	41.52	784.6s	42.47	539.8s
Lg3t	22.87×	41.11	356.9s	38.38	849.9s	34.99	581.15s
TCE_ex	29.77×	42.72	276.6s	17.82	768.8s	14.25	577.5s

Table II
RESULTS SUMMARY FOR INDIVIDUAL TENSOR CONTRACTIONS.

absolute performance in terms of GFlops, and time spent in the SURF algorithm to derive the final solution. Three of the four bencchmarks, Lg3, Lg3t and TCE_ex, achieve performance of more than 40 GFlops on the GTX 980 and speedups of more than 20× over the sequential Haswell implementation. Performance on the other GPU platforms is comparable for Lg3 and Lg3t, but quite a bit lower for TCE_ex. The results on Eqn.(1) are quite different. It is a computation that does not speed up compared to the Haswell, and only achieves 1.99 GFlops on the GTX 980, largely because there is insufficient work to compensate for the overhead of copying data to/from the GPU.

Search time varies from a few minutes to a few hours, depending on the computation and the architecture. The older GPU architectures typically spend more time in search, and the tiny Eqn.(1) computation spends the longest because the performances of its versions are so similar. Nevertheless, given the enormous search spaces associated with all of these variants, SURF is performing well. We also compared performance for some of these with prior work in [25] which used a brute force search of a smaller search space. We found that the performance resulting from SURF was comparable to and sometimes better than the prior brute force search.

We omitted from this table S1, D1 and D2, because each is comprised of nine kernels. Figure 3 illustrates the speedups achieved by the optimized Barracuda and OpenACC versions of these kernels over the naive OpenACC implementation. To summarize our results, performance ranges from 7 to 20 GFlops for S1, from 20 to 125 GFlops for D1 and 9 to 53 GFlops for D2, as is be shown in the next subsection. Search times for each of the nine kernels ranges from 8 to 32 minutes per kernel.

B. GPU Code Generation Strategies in Context

	OpenACC		
	Naive	Optimized	Barracuda
Tesla K20	2.86	12.39	36.47
Tesla C2050	1.18	19.21	34.65

Table III
NEKBONE PERFORMANCE COMPARISON: OPENACC VS. BARRACUDA.

We now consider how autotuning affects performance and the strategy for generating GPU code. We now consider the core computation from Nekbone, where the optimized Lg3 and Lg3t have been integrated into the code, and the computations representative of NWChem. Performance measurements for Nekbone are summarized in Table VI-B, and for the nine kernels for D1, D2 and S1 of NWChem are shown in Figure 3. We evaluate four different strategies for generating GPU code. OpenACC refers to taking the output of our framework, and then replacing our tool's generated CUDA constructs with OpenACC directives. We produced three OpenACC versions: *Naive* simply includes parallelization directives but no guidance on parallelization decomposition; *Optimized* adds directives on thread and block decomposition that were derived by Barracuda and performs scalar replacement on the output variable since the private designation in OpenACC does not produce the desired result. Overall we see that the *Naive* OpenACC code generation is even slower than sequential execution, but that the *Optimized* OpenACC version sometimes exceeds performance of code generated by Barracuda. Nevertheless, autotuning is essential to achieving high performance.

It is interesting also to ask whether a GPU is the right architecture for these computations. With the OpenMP comparison, we use manually-coded OpenMP versions, parallelizing an outermost loop for nekbone and using the OpenMP directives provided by the author of the NWChem excerpts. Table VI-B shows that the GTX 980 GPU outperforms a 4-thread OpenMP version on the Haswell in all cases for all benchmarks.

	1 core	OpenMP 4 cores	Barracuda
Nekbone	7.79GF	23.97GF	35.70GF
NWCHEM s1	2.47GF	2.61GF	16.14GF
NWCHEM d1	3.90GF	25.29GF	115.37GF
NWCHEM d2	5.60GF	14.90GF	50.00GF

Table IV
NEKBONE AND NWCHEM EXCERPT PERFORMANCE COMPARISONS: OPENMP VS. BARRACUDA.

VII. RELATED WORK

Our project combines compiler optimizations, scientific computation, and search algorithms. Some past research has dealt with tensor computation specifically, and a large body of work has used some form of search to improve code performance, either using a domain-specific language as we do here, or searching over an optimization space for a general purpose language like C or Fortran.

Optimizing Tensor Computations: Many tools and libraries have been developed for optimizing tensor con-

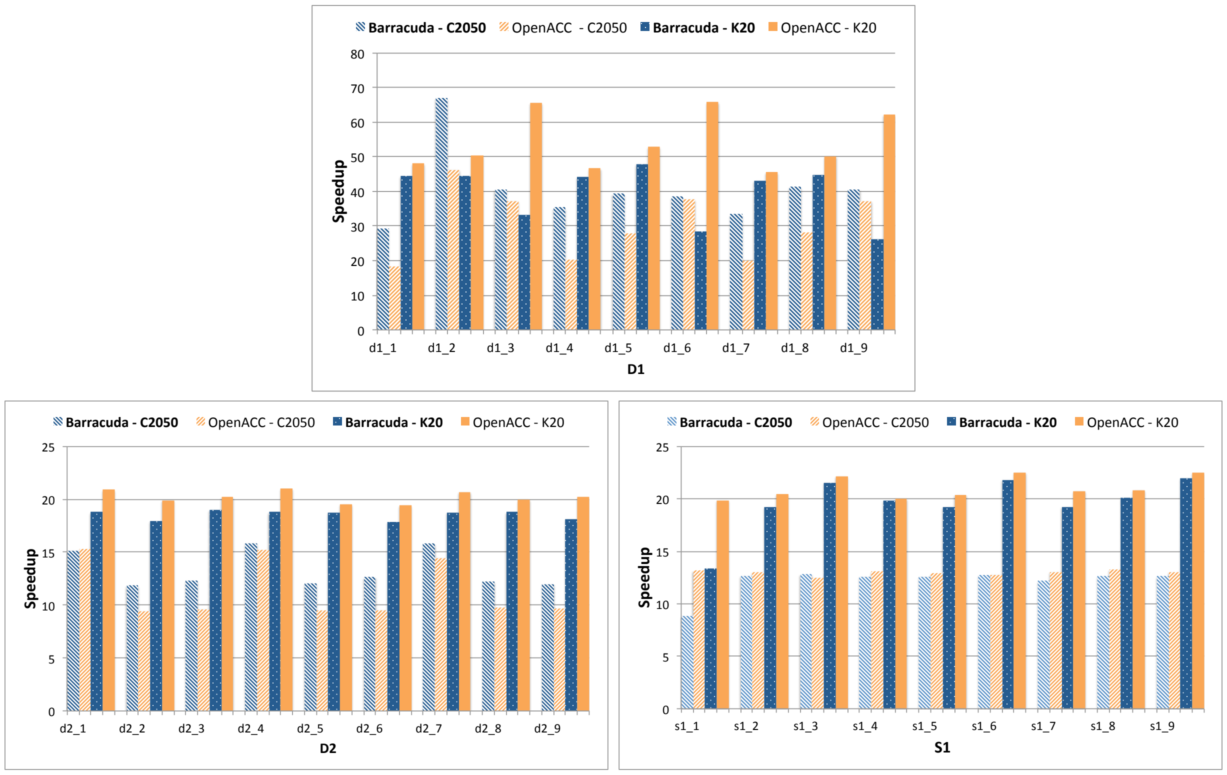

Figure 3. Speedup of the optimized Barracuda and OpenACC code versions over the naive OpenACC implementations of NWChem excerpts.

traction computations. Among these are the Tensor Contraction Engine (TCE) [4], the super instruction assembly language (SIAL) [26], and Cyclops [27]. These efforts have focused primarily on contractions involving very large tensors, possibly distributed across a large parallel computer. Performance optimization focuses on reducing the number of operations performed by exploiting symmetry and redundant subexpressions. Often, tensors are transposed so that a high-performance matrix-matrix multiplication can be used. Our focus is on the small tensors that arise in the spectral element method and can be used as a building block for computations involving large tensors. Thus, different optimization priorities apply.

Our work builds from ideas in the similar TCE project which also implements a DSL for tensors. In particular, our fusion and strength reduction algorithm is the same one described by the TCE papers for the single thread case. TCE takes an analytic approach, using a memory model to reduce traffic and exploit space-time tradeoffs [13]. Due to the large tensor problems they solve, the parallelism in TCE is distributed-memory parallelism that applies a generalization of Cannon's algorithm to reduce internode communication [19].

Several efforts have sought to optimize the performance of the Nekbone proxy application on GPUs. The CESAR codesign center reports tensor contraction performance of 100–200 GFlops on a Fermi GTX 590 GPU for tensors of size $8 \times 8 \times 8$ to $12 \times 12 \times 12$ using hand-coded OpenCL kernels [28]. The CRESTA project ported Nekbone to a multi-GPU system and reported a speedup of 1.59x using 512 Nvidia Kepler K20x GPUs versus a CPU-only implementation (512 nodes with 8192 cores) [8]. Although direct comparisons are difficult, our speedup of 1.3x versus OpenMP is encouraging, especially since our results include the time to transfer data back and forth between CPU and device memory.

Domain-Specific Tools: In addition to the tensor languages described above, other projects have used a domain-specific language to focus on high-level optimization and search opportunities.

SPIRAL [22] is a domain-specific language for discrete signal processing. It uses a genetic algorithm as a search strategy for autotuning. SPIRAL translates the search space into trees of rules for breaking discrete transforms into simpler units. The authors develop a unique crossover and mutation scheme for these ruletrees based on swapping and manipulating subtrees. SPIRAL has similarities to our approach, however the difference in domain requires different search and code generation. Signal processing transforms are usually created by calling smaller blocks which handle subproblems, and the primary search challenge is which of these transform decompositions to apply at each stage.

The DxT project [20] is a DSL for distributed-memory dense linear algebra that that uses a cost model based on operation count and communication costs to estimate the performance of many possible implementations. It composes each high-level algorithm mostly out of Level 3 BLAS. The authors use a similar style of search heuristics to narrow the space, focusing on transformations likely to be helpful.

Search and Autotuning: Vuduc, Demmel and Bilmes [32] study the optimization space of applying register tiling, loop unrolling, software pipelining, and software prefetching to matrix multiplication. They show that this search space is difficult (a very small number of combinations achieve good performance), and they present a statistical method for determining when a search has found a point that is close enough to the best.

Tiwari et al [29] describe an autotuning framework that combines ActiveHarmony's parallel search backend with the CHiLL transformation framework. Looptool [23] and AutoLoopTune [24] support loop fusion, unroll-and-jam, and array contraction. AutoLoopTune supports tiling. POET [34] supports a number of loop transformations.

Integer search paramaters, such as loop unroll factors or tile sizes, can take advantage of the integer space in the search strategy. Optimizations like loop fusion or adding SIMD instructions are not easily represented by an integer search parameter, so most search strategies do not apply. Zhao et al [35] use exhaustive search and empirical testing to select the best combination of loop fusion decisions. Qing and Qasem [33] apply empirical search to determine the profitability of optimizations for register reuse, SSE vectorization, strength reduction, loop unrolling, and prefetching. Their framework is parameterized with respect to the search algorithm and includes numerous search strategies.

VIII. Conclusion

This paper describes an autotuning system for tensor contraction computations targeting GPUs. The system uses a tensor-specific mathematical representation as input and generates an autotuning search space that is customized to both the domain of tensors with small dimension sizes, and GPU architectures. We explore the very large search generated by these tools using machine learning, resulting in search times that are practical. We show speedup over sequential and OpenMP execution, as high as 29×, and also demonstrate the necessity of autotuning when using OpenACC to generate efficient code. Our approach was driven by a desire to improve specific tensor problems not solved by current tools, but we also view this work as an exemplar for developing highly-tuned applications specialized for individual architectures starting with a mathematical representation of the problem in a DSL. In the future, we plan to extend this work to further prune the autotuning search space once we develop a better understanding of where pruning does not impact quality of results, and facilitate integration of the generated code into applications. As we expand the approach to surrounding computations, such as jointly optimizing lgrad3, lgrad3t and adjacent code, the search space will grow, and pruning it will be essential to feasibility.

Acknowledgments. The authors wish to thank Jeff Hammond who provided the NWCHEM kernels. The GPUs were donated by Nvidia through the Professor Partnership program. This work is partially supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contracts DE-AC02-06CH11357 and DE-SC0008682, and by National Science Foundation award CCF-1018881.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kauffman, 2002.

[2] P. Balaprakash, Y. Alexeev, S. Mickelson, S. Leyffer, R. Jacob, and A. Craig. Machine-learning-based load balancing for Community Ice CodE component in CESM. In *11th International Conference on High Performance Computing for Computational Science, VECPAR 2014, Revised Selected Papers*, volume 8969 of *Lecture Notes in Computer Science*. Springer, 2015.

[3] P. Balaprakash, R. B. Gramacy, and S. M. Wild. Active-learning-based surrogate models for empirical performance tuning. In *Proceedings of the 2013 IEEE International Conference on Cluster Computing, CLUSTER*, pages 1–8, Indianapolis, Indiana, 2013. IEEE Computer Society.

[4] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, et al. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, 2005.

[5] G. Baumgartner, D. E. Bernholdt, D. Cociorva, C.-C. Lam, J. Ramanujam, R. Harrison, M. Noolijen, and P. Sadayappan. A performance optimization framework for compilation of tensor contraction expressions into parallel programs. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS 2002, pages 33–42, Fort Lauderdale, Florida, 2002. IEEE Computer Society.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning*, volume 1. Springer, New York, 2006.

[7] NWChem TCE CCSD(T) loop-driven kernels. https://github.com/jeffhammond/nwchem-tce-triples-kernels, 2014.

[8] Large-scale fluid dynamics simulations—towards a virtual wind tunnel. ftp://www.mech.kth.se/pub/adam/CRESTA/Case_study/Nek5000_case_study.pdf, 2014.

[9] M. Deville, P. Fischer, and E. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge, 2002.

[10] E. Epifanovsky, M. Wormit, T. Ku, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, and A. I. Krylov. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of Computational Chemistry*, 34(26):2293–2309, 2013.

[11] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier. Welcome to Nek5000 project. http://nek5000.mcs.anl.gov, 2008.

[12] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.

[13] A. Hartono, Q. Lu, X. Gao, S. Krishnamoorthy, M. Nooijen, G. Baumgartner, D. E. Bernholdt, V. Choppella, R. M.

Pitzer, J. Ramanujam, A. Rountev, and P. Sadayappan. Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations. In *Proceedings of the 6th International Conference on Computational Science, ICCS 2006 - Part I*, volume 3991 of *Lecture Notes in Computer Science*, pages 267–275. Springer, 2006.

[14] A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, et al. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *The Journal of Physical Chemistry A*, 113(45):12715–12723, 2009.

[15] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, IPDPS 2009*, pages 1–11, Rome, Italy, 2009. IEEE Computer Society.

[16] A. Hartono, A. Sibiryakov, M. Nooijen, G. Baumgartner, D. Bernholdt, S. Hirata, C.-C. Lam, R. Pitzer, J. Ramanujam, and P. Sadayappan. Automated operation minimization of tensor contraction expressions in electronic structure calculations. In *Proceedings of the 5th International Conference on Computational Science, ICCS 2005 - Part I*, volume 3514 of *Lecture Notes in Computer Science*, pages 155–164. Springer, 2005.

[17] M. Khan. *Autotuning, code generation and optimizing compiler technology for GPUs*. PhD thesis, University of Southern California, 2012.

[18] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Trans. Archit. Code Optim.*, 9(4):31:1–31:25, Jan. 2013.

[19] Q. Lu, X. Gao, S. Krishnamoorthy, G. Baumgartner, J. Ramanujam, and P. Sadayappan. Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions. *Journal of Parallel and Distributed Computing*, 72(3):338–352, 2012.

[20] B. Marker, D. Batory, and R. van de Geijn. A case study in mechanically deriving dense linear algebra code. *International Journal of High Performance Computing Applications*, 27(4):439–452, Nov. 2013.

[21] B. Norris, A. Hartono, and W. Gropp. Annotations for productivity and performance portability. In *Petascale Computing: Algorithms and Applications*, Computational Science, pages 443–462. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007. Preprint ANL/MCS-P1392-0107.

[22] M. Pueschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2):232–275, 2005.

[23] A. Qasem, G. Jin, and J. Mellor-Crummey. Improving performance with integrated program transformations. Technical Report TR03-419, Department of Computer Science, Rice University, October 2003.

[24] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing: Special Issue on Computer Science Research Supporting High-Performance Applications*, 36(9):183–196, 2006.

[25] A. Rivera. Using autotuning for accelerating tensor-contraction on GPUs. Master's thesis, University of Utah, Dec. 2014.

[26] B. A. Sanders, R. Bartlett, E. Deumens, V. Lotrich, and M. Ponton. A block-oriented language and runtime system for tensor algebra with very large arrays. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC10, pages 1–11, New Orleans, Louisiana, 2010. IEEE Computer Society.

[27] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Proceedings of the IEEE 27th International Symposium on Parallel Distributed Processing, IPDPS 2013*, pages 813–824, Boston, Massachusetts, USA, 2013. IEEE Computer Society.

[28] The CESAR Team. The CESAR codesign center: Early results. https://cesar.mcs.anl.gov/content/cesar-codesign-center-early-results, 2012.

[29] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable autotuning framework for compiler optimization. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, IPDPS 2009*, pages 1–12, Rome, Italy, 2009. IEEE Computer Society.

[30] H. M. Tufo and P. F. Fischer. Terascale spectral element algorithms and implementations. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC99, Portland, Oregon, USA, 1999. ACM.

[31] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.

[32] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.

[33] Q. Yi and A. Qasem. Exploring the optimization space of dense linear algebra kernels. In *Languages and Compilers for Parallel Computing: 21st International Workshop, Revised Selected Papers*, LCPC 2008, pages 343–355, Edmonton, Canada, 2008. Springer.

[34] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *Proceedings of the Workshop on Performance Optimization of High-level Languages and Libraries (POHLL) at the 21st IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2007, pages 1–8, Long Beach, California, 2007. IEEE Computer Society.

[35] Y. Zhao, Q. Yi, K. Kennedy, D. Quinlan, and R. Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical Report UCRL-TR-217808, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2005.