USING AUTOTUNING FOR ACCELERATING TENSOR CONTRACTION ON GRAPHICS PROCESSING UNITS (GPUS)

by

Axel Y. Rivera

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

December 2014

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Axel Y. Rivera**

has been approved by the following supervisory committee members:

Mary Hall	, Chair	**Oct. 31, 2014** Date Approved
Paul Hovland	, Member	**Aug. 29, 2014** Date Approved
Robert M. Kirby	, Member	**Oct. 24, 2014** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Tensors are mathematical representations of physical entities that have magnitude with multiple directions. Tensor contraction is a form of creating these objects using the Einstein summation equation. It is commonly used in physics and chemistry for solving problems like spectral elements and coupled cluster computation. Mathematically, tensor contraction operations can be reduced to expressions similar to matrix multiplications. However, linear algebra libraries (e.g., BLAS and LAPACK) perform poorly on the small matrix sizes that commonly arise in certain tensor contraction computations. Another challenge seen in the computation of tensor contraction is the difference between the mathematical representation and an efficient implementation. This thesis proposes a framework that allows users to express a tensor contraction problem in a high-level mathematical representation and transform it into a linear algebra expression that is mapped to a high-performance implementation. The framework produces code that takes advantage of the parallelism that graphics processing units (GPUs) provide. It relies on autotuning to find the preferred implementation that achieves high performance on the available device. Performance results from the benchmarks tested, nekbone and NWChem, show that the output of the framework achieves a speedup of 8.56x and 14.25x, respectively, on an NVIDIA Tesla C2050 GPU against the sequential version; while using an NVIDIA Tesla K20c GPU it achieved speedups of 8.87x and 17.62x. The parallel decompositions found by the tool were also tested with an OpenACC implementation and achieved a speedup of 8.87x and 10.42x for nekbone, while NWChem obtained a speedup of 7.25x and 10.34x compared to the choices made by default in the OpenACC compiler. The contributions of this work are: (1) a simplified interface that allows the user to express tensor contraction using a high-level representation and transform it into high-performance code; (2) a decision algorithm that explores a set of optimization strategies for achieving performance; and, (3) a demonstration that this approach can achieve better performance than OpenACC and can be used to accelerate OpenACC.

This work is dedicated to my wife, Zuleika Almanzar, our soon to be born baby and my parents, Isander Rivera and Ruth Rodríguez, for their unconditional support during this journey. They are my source of motivation to accomplish the goals in my life.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Fist of all, I want to thank The Almighty God for all the blessings that were provided during this time and the ones to come.

I want to give my deepest appreciation to my committee chair, Dr. Mary Hall, for her guidance and teaching. She conveyed a spirit of adventure to research and explore the world of science. Without her advice and support this thesis work would not be possible.

I want to extend my gratitude to my committee members, Dr. Paul Hovland (Argonne National Laboratories) and Dr. Robert M. Kirby (University of Utah). They provided important advice that made this work possible.

I would like to thank Thomas Nelson and Dr. Elizabeth Jessup from the University of Colorado, Dr. Boyana Norris from the University of Oregon and Dr. Manu Shantharam at the University of Utah. Their contributions also allowed the production of this thesis.

Finally, I want to include José Sotero and Idalia Ramos from the University of Puerto Rico at Humacao. Both provided me the inspiration to achieve graduate education.

Partial support for this work was provided from the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy Office of Advanced Scientific Computing Research under award number DE-SC0006947 and by a separate DOE award DE-SC0008682. This work was also partially supported by National Science Foundation award CCF-1018881.

CHAPTER 1

INTRODUCTION

Tensors are mathematical representations of physical entities that have magnitude with multiple directions [1]. Tensor contraction is a form of creating tensors using the Einstein summation equation [1, 2, 3]. It is commonly used in physics and chemistry for computing different techniques like spectral element methods [4] and coupled cluster computations [5]. Spectral element methods are used for modeling fluids [4]; while coupled cluster theory is used for making an accurate approximation of the atomic and molecular electronic structure [6].

Mathematically, tensor contraction problems can be reduced to expressions similar to matrix multiplications. While library implementations could be used (e.g., BLAS and LAPACK), these routines perform poorly on the small matrix sizes that commonly arise in certain tensor contraction computations. Another challenge seen in tensor contraction problems is the disconnection between mathematical representation and the implementation.

In this thesis work we present how compiler optimization techniques can be exploited to speed up the computation of tensors on graphics processing units (GPUs). Efficient computation of these objects can be a challenge when they are small in size, but with the right transformations it is possible to achieve high performance. We explore how the optimization techniques used on the CPU can also be applied to these multiprocessing units to achieve high performance [7]. We also present an interface that allows the computation to be expressed mathematically. This representation is used to create optimized code for GPUs.

1.1 Tensors and Tensor Contraction

The elements of a tensor describe a linear mapping between scalars, vectors or other tensors. The order of a tensor is depends on the amount of dimensions; e.g., 0^{th} order is a scalar, 1^{st} order is a vector and 2^{nd} is a matrix [3].

There are numerous ways to produce a tensor; the simplest one is generating a scalar, vector, or an object with higher dimensions, using unit vectors [1]. For example, Figure 1.1 presents a Cartesian plane with a vector that has a magnitude of 8 and an angle of 60^o. A tensor can be composed by finding the unit vectors in this vector by applying traditional trigonometry. The results are 4, $4\sqrt{3}$ and 0 for the x, y and z axis, respectively. These unit vectors compose a tensor of 1^{st} order (vector).

The tensor product, also known as the Kronecker product (denoted by $\bigotimes$), is another way for creating tensors, as defined by Equation 1.1. Tensor C is created from tensors A and B, with sizes of $m \times n$ and $l \times k$, respectively, where matrix B is multiplied with each entry of the matrix A and stored in C [3].

$$C = A \bigotimes B = \begin{bmatrix} a_{1,1}\begin{bmatrix} b_{1,1} & \cdots & b_{1,k} \\ \vdots & \ddots & \vdots \\ b_{l,1} & \cdots & b_{l,k} \end{bmatrix} & \cdots & a_{1,n}\begin{bmatrix} b_{1,1} & \cdots & b_{1,k} \\ \vdots & \ddots & \vdots \\ b_{l,1} & \cdots & b_{l,k} \end{bmatrix} \\ \vdots & \ddots & \vdots \\ a_{m,1}\begin{bmatrix} b_{1,1} & \cdots & b_{1,k} \\ \vdots & \ddots & \vdots \\ b_{l,1} & \cdots & b_{l,k} \end{bmatrix} & \cdots & a_{m,n}\begin{bmatrix} b_{1,1} & \cdots & b_{1,k} \\ \vdots & \ddots & \vdots \\ b_{l,1} & \cdots & b_{l,k} \end{bmatrix} \end{bmatrix} \tag{1.1}$$

Some of the important properties of the operator $\bigotimes$ are given by:

- Noncommutative: $A \bigotimes B \neq B \bigotimes A$
- Transpose is distributive: $(A \bigotimes B)^T = A^T \bigotimes B^T$
- Mixed-product property: $(AB \bigotimes CD) = (A \bigotimes C) \times (B \bigotimes D)$

Another form of producing tensors is using a tensor contraction. Tensor contraction (denoted by $\times$) is the creation of a new tensor by summing the products between the components of the primary tensors [1, 2, 3]. Equation 1.2 presents an example of a tensor contraction between a tensor of 2^{nd} and 1^{st} order (A and b), which produce another tensor

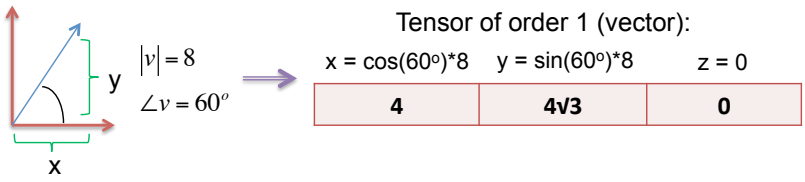

Figure 1.1: Representation of a 1^{st} order tensor using the unit vectors 4, $4\sqrt{3}$ and 0.

of order 1 (c). The contraction will happen when a pair of indices (one subscript and one superscript) in the Einstein notation is set equal in the resultant tensor.

$$\begin{aligned} c &= A \times b \\ c_i &= \sum_k A_k^i * b_k \end{aligned} \tag{1.2}$$

Examples of tensor contraction are:

- Dot product: Two tensors of 1^{st} order produce a 0^{th} order tensor.
- Matrix-vector product: Tensor of 2^{nd} order and tensor of 1^{st} order produces a 1^{st} order tensor.
- Matrix-matrix product: Two tensors of 2^{nd} order produce a 2^{nd} order tensor.

Tensor contraction is not limited to these three examples. Through this thesis work we will see some cases that use tensors of order 2 and 4 to produce tensors of order 6, while other cases perform tensor contraction between tensors of order 2 and 3.

1.2 Applications that use Tensor Contraction

Tensor contractions are widely used in many fields. Physicists use them in applications that compute solutions related to inertia, general relativity or spectral elements. Chemistry applications use them in the coupled cluster computation. This thesis work focuses on two real life applications that use tensor contraction: nek5000 and NWChem. While nek5000 uses small sizes, NWChem partitions the data in small chunks to operate within a single node.

1.2.1 Nek5000: Spectral Elements

Nek5000 is an application that implements a 3-dimensional spectral element discretization technique [8, 9]. It executes a conjugate gradient loop that solves Equation (1.3) [10]. Here A, B and C are 3^{rd} order tensors and $\underline{u}$ is a 2^{nd} order tensor. This equation is composed of tensor products ($A \bigotimes B \bigotimes C$) and a tensor contraction ($C \times \underline{u}$).

$$D = A \bigotimes B \bigotimes C \times \underline{u} \tag{1.3}$$

Traditionally, the solution of Equation 1.3 is given by $d_{i,j,k} = \sum_l \sum_m \sum_n A_k^l * B_j^m * C_i^n * u_{l,m,n}$. This summation is computationally intensive and in this case is $O(n^6)$. The computational order increases with increasing dimensions.

Nek5000 expresses the problem as $D = (A \bigotimes I \bigotimes I) \times (I \bigotimes B \bigotimes I) \times (I \bigotimes I \bigotimes C) \times \underline{u}$. Then it uses partial results for computing the final result.

$$\begin{aligned} \underline{v} &= (I \bigotimes I \bigotimes C)\underline{u} \\ v_{i,j,k} &= \sum_p C_p^k * u_{i,j,p} \end{aligned} \tag{1.4}$$

Equation 1.4 creates a new tensor of 2^{nd} order called $\underline{v}$. This tensor is generated by the tensor contraction between $\underline{u}$ (2^{nd} order) and C^T (3^{rd} order). Usually, this solution is computed as $\underline{v} = \underline{u} \times C^T$, which resembles a $n^2 * n \times n * n$ matrix multiplication. The generated tensor, $\underline{v}$, is used to solve the second partial result.

$$\begin{aligned} \underline{w} &= (I \bigotimes B \bigotimes I)\underline{v} \\ w_{i,j,k} &= \sum_p B_p^j * v_{i,p,k} \end{aligned} \tag{1.5}$$

Equation 1.5 represents tensor contraction between two tensors of 3^{rd} order. This is represented as $\underline{w}(:,:,k) \times B^T$, which is a n batch of $n * n \times n * n$ matrix multiplications.

$$\begin{aligned} D &= (A \bigotimes I \bigotimes I) \times \underline{w} \\ d_{i,j,k} &= \sum_p A_p^i * w_{p,j,k} \end{aligned} \tag{1.6}$$

Finally, Equation 1.6 provides the final result for D. This, same as Equation 1.4, is computed as a matrix multiplication of $n * n \times n * n^2$ between $A \times \underline{w}$.

The optimization presented reduces the amount of computation; in this case, the order decreased from $O(n^6)$ to $O(n^4)$. Not only is the amount of computation reduced, but also it is now expressed as matrix multiplication. Tensor contraction problems are usually expressed in these representations, making it possible to use popular high performance linear algebra libraries like BLAS and LAPACK to achieve high performance [11]. On the other hand, often in practice, the sizes of the tensor are small. Typically, the library implementations for performing matrix multiplication like DGEMM are tailored for much

larger sizes, which lead to poor performance [12]. Nek5000 use a specialized kernel tailored for high performance matrix multiplications of small matrix sizes [7]. When the size is large enough, then nek5000 uses these libraries to compute the solution. The small sizes in nek5000 are related to the order of the discretization polynomial. As it increases, the time required to achieve the convergence point in the computation also increases [10].

1.2.2 NWChem CCSD(T): Coupled Cluster

NWChem is a software package for quantum chemistry and molecular dynamics [13]. We focus on the kernels extracted from the CCSD(T) (coupled cluster theory with full treatment singles, doubles and triples estimated using perturbation theory) computations of NWChem [14]. The tensor contraction is given by two tensors of different orders and produces another tensor with distinct order. This computation resembles a batched matrix multiplication.

The main kernel is divided into three sets where each set has nine functions that explore the tensor contraction in different points of the data. The first set (*sd_t_s1*) performs a tensor contraction between two tensors with order 2 and 4. The second (*sd_t_d1*) and third (*sd_t_d2*) sets work with two tensors where both are order 4. All three cases store the results into a tensor of order 6. The CCSD(T) kernels use small sizes since they represent chunks of a larger tensor contraction problem to be performed in a distributed system. The size of the slab is arbitrary but it should be small enough to make efficient use of the cache hierarchy.

1.3 GPUs and Compilers Optimizations for Achieving Performance

Processes that involve tensor contraction can be computationally and memory intensive, especially when the order of the tensors get higher. Fortunately, in recent years, an emergent architecture allows achieving high performance in a conventional desktop computer: graphic processing units (GPUs). The advantage of GPUs is that the cost and energy consumption are relatively low compared with the amount of performance they provide.

A GPU is an accelerator connected to the motherboard of the computer through a PCI-Express port. These units were originally designed to accelerate graphics applications like games and animated movies. Recently, the high performance computing community has adopted them for more general purpose use due to the simplified architecture and parallelism they provide. Figure 1.2 [15] presents the architectural difference between a CPU and a GPU. Notice that while a CPU has a limited number of registers, GPUs contain a larger

Figure 1.2: Architectural difference between a CPU (left) and a GPU (right). Green squares represent the arithmetic logic units (ALU) registers, yellow are the control registers and orange squares represent memory [15].

amount of simplified registers. GPUs achieve parallelism by using these registers to perform the same computation over different regions of the data.

Another feature that makes GPUs popular is the programming interface. Users benefit from an application programming interface (API) known as compute unified device architecture (CUDA). As an extension of C++, CUDA allows programmers to create kernels for performing the parallel computation in the GPU. It also provides the tools for expressing the data transfer between device (GPU) and host (main memory).

Apart from CUDA, OpenACC [16] is a newer API that allows users to develop programs for GPUs and other accelerators. It uses an OpenMP-like interface (directives for FORTRAN and pragmas for C++) where users specify which loop nests are going to be executed on these devices. OpenACC also provides the necessary instructions for copying data and synchronizing with the host.

Writing highly-optimized code for GPUs requires careful management of the memory hierarchy, data copies, synchronizations and parallelism granularity, among others. These strategies are difficult for users to do manually since it requires different transformations that can lead to a variety of implementations. To address this problem we use a concept known as autotuning. Autotuning permits the creation of code variants using different transformations for achieving high performance in a target architecture [17, 7, 18, 19]. It has been widely used in compilers for strategies that produce better code on CPUs and, recently, GPUs [20, 21, 22].

1.4 High Performance Tensor Contraction Interface

Our research goal is to automate the difficult task of code generation for tensor contraction targeting GPUs. The observations that motivated this thesis work are:

- Tensor contraction problems can be expressed as matrix multiplications, which allows the use of high-performance libraries like LAPACK and BLAS. When the sizes of the tensors are small, these libraries may lead to poor performance because they are tailored for large sizes [7]. We need a customized optimization strategy to achieve better performance for these small sizes.

- The same result for a tensor contraction problem can be achieved by different mathematical expressions. The decision of choosing which implementation is going to be used can have an impact on the performance of the application.

- In many cases, the tensor contraction problems are reduced to a batched matrix multiplication. We want to use the parallelism provided by GPUs to accelerate this process.

This thesis proposes an interface resulting from these observations. It presents a framework that transform a tensor contraction expressions into high performance code for GPUs. The tool integrates an interface where the user inputs a high-level pseudo-code that is close to a tensor contraction representation. It automatically generates a collection of implementations and uses autotuning to find possible transformations that generate high performance codes. These implementations are exhaustively explored to select the best version. The focus of this work is oriented to tensor contraction problems that are small but that can still consume a large portion of the computation time of real applications.

Figure 1.3 presents a flowchart of the proposed framework. This framework depends on external information that is an expansion to this work (presented in the gray box area). The user inputs a high-level representation that resembles the mathematical tensor notation, and the external tool applies different transformations for reducing the number of operations, among other optimizations. The output from the external information is a tensor-specialized representation that is used throughout this system to generate high-performance code.

The tensor representation from the external tool generates a basic C++ implementation (Chapter 2). The output, in collaboration with a decision algorithm, is used for creating a series of transformation recipes that describe how to map the code to a GPU (Chapter 3). Each *transformation recipe* is an ordered list of the optimization strategies to be applied on the code for achieving high performance. The set of recipes is used by the CUDA-Composing High-Level Loop Transformations Tool [23, 17] (CUDA-CHiLL) for creating different optimized CUDA codes. These codes are tested exhaustively by Orio [24, 25] to find the best implementation (Chapter 4).

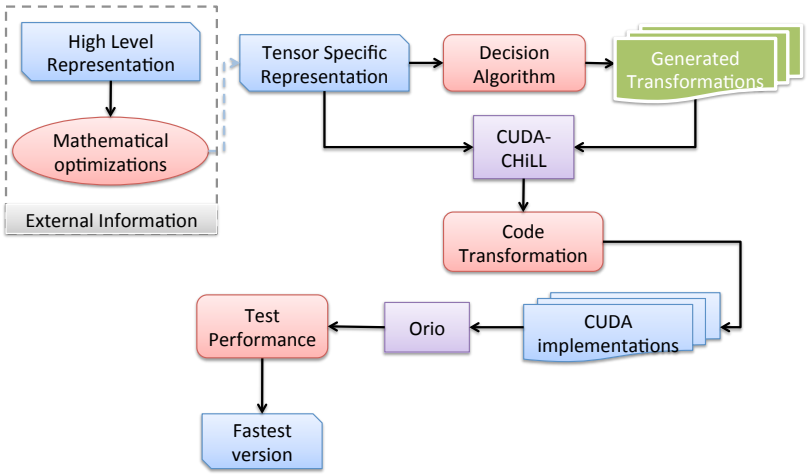

Figure 1.3: Flowchart of the proposed framework. Gray area represents the external information, blue rectangles represent files with code, batched figure are the different transformations and implementations, red rectangles are work done by the framework and purple rectangles are external tools used.

The goal of this framework is to improve the performance of the application that uses tensor contraction. The generated code is used to replace the original implementation as well as, optionally, providing guidance for implementing OpenACC directives.

1.5 Research Contributions

The contributions of this thesis work are:

- Simplifying the implementation of tensor contraction expression using a representation that resembles mathematical notation.
- Providing a decision algorithm based on autotuning that explores different optimization strategies for accelerating small problems related to tensor contraction.
- Demonstrate that this approach can achieve high performance compared to OpenACC and that the result produced by the framework can also be used to accelerate OpenACC.

1.6 Organization of This Thesis

The rest of this thesis is organized as follows. Chapter 2 presents the mathematical representation of tensor contraction and how the C++ version is generated. Chapter 3 explores autotuning for generating different sets of transformations, and Chapter 4 shows how these variants are transformed into code to find the best implementation. Chapter 5

presents the experimental evaluation of this work as well a discussion of the results. Chapter 6 discusses related work, and Chapter 7 presents a summary and conclusions.

CHAPTER 2

MATHEMATICAL REPRESENTATION OF TENSOR CONTRACTION

The best-performing implementation of a tensor contraction computation varies depending on the application context and architecture. Let us consider the tensor contraction in nek5000. Figure 2.1 presents a C++ linearized version of this operation; the original implementation is written in FORTRAN. It captures how the data layout is in column major order in memory, which means that the elements of a column in a multidimensional array are contiguous in memory.

```
for(e = 0; e < nelt; e++){

 for(j = 0; j < lx; j++)
  for(i = 0; i < ly; i++)
   for(k = 0; k < lz; k++)
    for(m = 0; m < lx; m++)
     Ur[e*lx*ly*lz + j*ly*lz + i*lz + k]  +=
       D[m*ly + k] * U[e*lx*ly*lz + j*ly*lz + i*lz + m];

 for(j = 0; j < lx; j++)
  for(i = 0; i < ly; i++)
   for(k = 0; k < lz; k++)
    for(m = 0; m < ly; m++)
         Us[e*lx*ly*lz + j*ly*lz + i*lz + k]  +=
           U[e*lx*ly*lz + j*ly*lz + m*lz + k] * Dt[i*ly + m];

 for(j = 0; j < lx; j++)
  for(i = 0; i < ly; i++)
   for(k = 0; k < lz; k++)
    for(m = 0; m < lx; m++)
     Ut[e*lx*ly*lz + j*ly*lz + i*lz + k]  +=
       U[e*lx*ly*lz + m*ly*lz + j*lz + k] * Dt[i*ly + m];

}
```

Figure 2.1: C++ implementation of the tensor contraction performed in nek5000. `U`, `Ur`, `Us` and `Ut` are *nelt* batch of tensors with dimension $lx \times ly \times lz$, while `D` and `Dt` are $lx \times ly$ tensors.

Although mathematically these tensor contraction computations are similar to matrix multiplication, they may be implemented equivalently in different ways as was described in Chapter 1. This motivated us to create an interface where the user can present the tensor contraction in a high-level representation and generate a high-performance implementation.

This chapter describes the input to the system, which is based on the tensor-specific representation of the problem. The input can be generated either by an external tool or specified by the user. The organization of this chapter is as follows: Section 2.1 presents the tensor-specific representation, and Section 2.2 shows the extra features needed for handling the data representation. Section 2.3 presents a summary of this chapter.

2.1 Tensor-Specific Representation

Considering how tensor contraction problems are presented mathematically, we find some barriers to express them. For example: keyboards do not have a summation key, and a standard text editor does not have a way to write subscripts. This means that we need to express the abstraction in a form where every user can write it and understand it.

The variable declarations in this system are similar to FORTRAN. Figure 2.2 presents how the tensors are expressed in the interface. The user types the name of the tensor, followed by a colon (:) and the list of indices ordered as they are accessed. For example, $c_{i,j}$ is expressed in the interface as `c:(i,j)`.

The interface uses the addition (+=) and subtraction (-=) C++ compound symbols to represent the summation symbol. In this work, the compound symbols are known as assignments. Then, the regular operator for multiplication ($\star$) is used to represent the binary product between two input tensors. Figure 2.3a presents how a mathematical operation is expressed in the framework. It is comprised of the output tensor with the corresponding indices, an assignment represented by the compound symbol and a binary operation containing both inputs in the corresponding indices. Each representation in the system is known as a *statement*.

Figure 2.3b presents an example of a statement based on the `Ur` computation of nek5000. The output tensor, which is $ur_{e,i,j,k}$, is presented as `ur:(e,i,j,k)`, and the input tensors ($d_{j,m}$ and $u_{e,i,m,k}$) are `d:(j,m)` and `u:(e,i,m,k)`. The assignment is a summation ($\sum$) represented by += and the multiplication is handled by $\star$.

```
Variable:(index_1, index_2, ... , index_i, ... , index_n)
```

Figure 2.2: Representation of a variable with the accessed indices in the interface.

```
Output_Var:(index_1, ... , index_m) [assignment]
    Input_Var1:(index_1, ... , index_n) *
    Input_Var2:(index_1, ... , index_p)
```

(a) Composition of a mathematical statement in the interface.

$ur_{e,i,j,k} = \sum_m d_{j,m} \times u_{e,i,m,k} \implies$ `ur:(e,i,j,k) += d:(j,m) * u:(e,i,m,j)`

(b) Computation of *ur* expressed in the interface.

Figure 2.3: Representation of the tensor contraction operation in the interface.

2.2 Complete Structure

Section 2.1 presents only the relationship between mathematics and the input, but the user must also describe the data representation. This work focuses on the following:

- The data layout in memory (row or column major) can have an impact on cache behavior and memory bandwidth.
- It is important to know if the loops access the data in contiguous or strided order because the access order to the data can change; the placement is not modified (e.g., NWChem).

The functionality of the interface was expanded to address these issues.

Figure 2.4 presents the structure of the input file used by the framework. The function name, followed by the access specification (`multidimensional` or `linearized`), data layout (`row` or `column` major), and if the access is `contiguous` or `strided` (`pattern`).

```
     function name
(op) access: multidimensional | linearized
(op) memory: row | column
(op) pattern: contiguous | strided
     define:
       var1 = val
            ⋮
       varN = val
     variables:
       Var_1:orderN | (size_1, ..., size_N)
            ⋮
       Var_n:orderN | (size1, ..., sizeN)
     operations:
       Var_x:(indices) [assignment] Var_y(indices) * Var_z(indices)
            ⋮
       Var_n:(indices) [assignment] Var_t(indices) * Var_r(indices)
```

Figure 2.4: Structure of the user input file. The user specifies the function name, extra features (`op` stands for optional), define values, variables and the operations.

Also, it is possible to define variables that are going to be used internally by the framework, such as the dimension sizes.

In this framework the users must declare a tensor before using it in a statement. A tensor can be declared in two forms, either specifying the order or specifying the size of each dimension. User-defined variables can be used for representing the dimensions. Once the tensors are declared, then the user proceeds with the tensor-specific description. The interface will use this specification to transform the input file into a C++ function. Figure 2.5 presents an example related to nek5000; Figure 2.5a shows the input file and Figure

```
local_grad3
memory: column
access: linearize
define:
 lx = 10
 ly = 10
 lz = 10
 nelt = 1000
variables:
 u:(nelt,lx,ly,lz)
 D:(lx,ly)
 Dt:(lx,ly)
 ur:(nelt,lx,ly,lz)
 us:(nelt,lx,ly,lz)
 ut:(nelt,lx,ly,lz)
operations:
 ur:(e,i,j,k) += D:(k,m)*
    u:(e,i,m,j)
 us:(e,i,j,k) += u:(e,i,k,m)*
    Dt:(m,j)
 ut:(e,j,i,k) += u:(e,m,k,j)*
    Dt:(m,j)
```

(a) Input file representing the computation performed in nek5000.

```
#define lx 10
#define ly 10
#define lz 10
#define nelt 1000
void local_grad3(double *Ut,double *Us,
   double *Ur,double *D,double *U,
   double *Dt){

 int e, i, j, k, m;
 for(e = 0; e < nelt; e++){
  for(i = 0; i < lx; i++)
   for(j = 0; j < ly; j++)
    for(k = 0; k < lz; k++)
     for(m = 0; m < lx; m++)
      Ur[e*lx*ly*lz+i*ly*lz+j*lz+k]+=
       D[m*ly + k] *
       U[e*lx*ly*lz+i*ly*lz+j*lz+m];

  for(i = 0; i < lx; i++)
   for(j = 0; j < ly; j++)
    for(k = 0; k < lz; k++)
     for(m = 0; m < ly; m++)
      Us[e*lx*ly*lz+i*ly*lz+j*lz+k]+=
       U[e*lx*ly*lz+i*ly*lz+m*lz+k] *
       Dt[j*ly + m];

  for(i = 0; i < lx; i++)
   for(j = 0; j < ly; j++)
    for(k = 0; k < lz; k++)
     for(m = 0; m < lx; m++)
      Ut[e*lx*ly*lz+i*ly*lz+j*lz+k]+=
       U[e*lx*ly*lz+m*ly*lz+j*lz+k] *
       Dt[j*ly+m];
 }
}
```

(b) Output produced by the interfaces using the nek5000 example.

Figure 2.5: Example representing the tensor contraction computed in nek5000. The interface uses the input file (a) and produces a C++ output (b).

2.5b the C++ output function. The generated function expects that the input arrays (`Ur`, `Us` and`Ut` for the example presented before) are initialized by the application before using them.

The interface does not apply the optimizations presented in Chapter 1. The strategies presented before are oriented to accelerate the computation on CPUs and our target architecture is a GPU. Since both architectures are different, the framework will explore other optimization strategies to achieve high performance. For this purpose, the code generated for nek5000 presents three tensor contractions between 3^{rd} and 2^{nd} order tensors; rather than two tensor contractions of 2^{nd} order and one operation of 3^{rd} and 2^{nd} order tensors. Chapter 3 explains in detail the optimizations used by the framework.

2.3 Summary and Contributions

This chapter presents a framework where the user can input a high level expression that represents a tensor contraction and produce C++ code that implements it. With this interface the users should not worry about how to implement the tensor contraction and can focus more on the mathematical description of their problem.

The contribution of this chapter is the introduction of a simplified interface that reduces the gap between the mathematical representation and the optimized code. It also provides guidance for code generation.

CHAPTER 3

DECISION ALGORITHM AND AUTOTUNING

The previous chapter describes how the user expresses the tensor contraction and generates a basic C++ code that will run on a CPU. The next step is a decision algorithm that derives a sequence of transformations to be considered for creating an optimized CUDA implementation. This step involves autotuning, which is a strategy for finding the best performing implementation among a search space of possible versions.

The main motivation for using autotuning is the fact that the optimization strategies used for large tensor contraction problems lead to poor performance over small sizes. This will create a new search space based on different optimization strategies for the computation of small tensors in a GPU. The autotuning focuses on finding those transformations available in the search space that generate high performance code. This work targets reducing the communication to the global memory of the GPU. The first step is creating code that permits contiguous access among consecutive threads to the data located in the global memory. Adjacent data, used by coscheduled threads, can be retrieved from global memory using a single memory transfer. This feature is known as *global memory coalescing*. It reduces expensive communication with global memory. Memory coalescing is directly impacted by the thread and block decomposition. Finding the best thread and block decomposition is the main target for autotuning.

Apart from memory coalescing, there are other optimization strategies that can affect performance, such as accessing reused global data from the cache. Cache behavior is also impacted by thread and block decomposition and loop order. Exploiting the data reuse available in cache also reduces communication with global memory.

The rest of this chapter is organized as follows. Section 3.1 presents a series of steps to perform autotuning, and Section 3.2 explains the output. Section 3.3 is a summary of this chapter.

3.1 Decision Algorithm

Autotuning focuses on evaluating a search space empirically and finds the optimal parameters that achieve an optimization goal (performance, energy, among others) [17, 7, 18, 19]. These parameters depend on the input data, target architecture and back-end compiler to aggressively optimize the implementation. The autotuning proposed in this work adapts the decision algorithm by Khan et al. [21, 22] for creating an implementation that achieves high performance. At a high level it generates the thread and block decomposition (*computation partition*) and the decisions for copying the data to specific memory levels (*data placement*). This information is used to create a set of customized transformation recipes (as defined in Chapter 1) for the specific tensors in the computation. The specialized recipes are the output from the decision algorithm that the interface will use to transform the C++ code from Chapter 2 and test among the multiple implementations.

3.1.1 Data Dependence Analysis

The decision algorithm starts by analyzing data dependences for each statement from the tensor representation. For the tensors described by the framework, this work can use a simplified data dependence analysis. The test examines the indices available in the right hand side of the statement but not in the left hand side. These indices represent a dependence carried by the corresponding loop. The goal of the test is to identify the loops that do not carry dependences and can be performed in parallel. For example, Figure 3.1 presents the tensor representation file for the function *sd_t_d1_1* in the coupled cluster (NWChem) example. The only index listed, for this case, in the right hand side that does

```
sd_t_d1_1
memory: column
pattern: strided
define:
  h3u = 16
  h2u = 16
  h1u = 16
  p6u = 16
  p5u = 16
  p4u = 16
  h7u = 16
variables:
  t3:(h3u,h2u,h1u,p6u,p5u,p4u)
  t2:(h7u,p4u,p5u,h1u)
  v2:(h3u,h2u,p6u,h7u)
operations:
  t3:(h3,h2,h1,p6,p5,p4) -= t2:(h7,p4,p5,h1) * v2:(h3,h2,p6,h7)
```

Figure 3.1: Tensor representation of NWChem sd_t_d1_1.

not appear in the left hand side is *h7*, making it the loop that carries the dependence. The data dependences analysis stores in two different arrays the loops that can be performed in parallel and those that have dependences.

The information from the analysis also identifies which indices of the two input tensors in a statement exhibit contiguous access in memory. For the purpose of generated tensor contraction code in this framework, the tensor where the loop indices are iterated from inner-most to outer-most access contiguous data. We will use the C++ output from the tensor representation of sd_t_d1_1, presented in Figure 3.2, to explain this in details. In this case, the indices in `v2` go from innermost to outermost loops (`h3`, `h2` and then `p6`) and `t2` loops are outermost to innermost (`p4`, `p5` and `h1`). Therefore, the elements of `v2` will be accessed in contiguous order while `t2` will not be contiguous. The tensor implementation that achieves contiguous data access, in this work, is defined as the *contiguous tensor*.

3.1.2 Thread and Block Decomposition

The decision algorithm exposes the important decisions and the thread and block decomposition that makes efficient use of the GPU. In this document we will call $Thread_x$ and $Thread_y$ the threads in the coordinates X and Y in the computational grid of the GPU. Also, $Block_x$ and $Block_y$ refer to the blocks in the respected coordinates. Although it is possible to use the threads available in the Z coordinate, we omit it since it can produce an amount of threads that exceed the quantity supported by the device.

The data analysis is used for generating the computational grid of the GPU. The contiguous tensor will be used for finding the thread and block decompositions for the GPU computation that can achieve memory coalescing. The loop that allows contiguous access to the data in the contiguous tensor is the innermost that does not carry a dependence. This loop is set as $Thread_x$. Allowing this thread to access contiguous data achieves memory coalescing, as shown in Figure 3.3.

The rest of the implementation depends on the number of parallel loops found by the

```
for (int p4=0; p4<p4u; p4++)
 for (int p5=0; p5<p5u; p5++)
  for (int p6=0; p6<p6u; p6++)
   for (int h1=0; h1<h1u; h1++)
    for (int h2=0; h2<h2u; h2++)
     for (int h3=0; h3<h3u; h3++)
      for (int h7=0; h7<h7u; h7++)
       t3[h3+h3u*(h2+h2u*(h1+h1u*(p6+p6u*(p5+p5u*p4))))] -=
        t2[h7+h7u*(p4+p4u*(p5+p5u*h1))] * v2[h3+h3u*(h2+h2u*(p6+p6u*h7))];
```

Figure 3.2: C++ code generated for NWChem sd_t_d1_1.

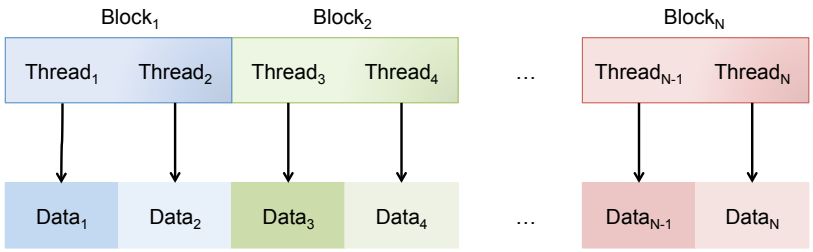

Figure 3.3: Representation of the threads accessing data located in the global memory of the GPU.

dependence test. After setting $Thread_x$, the algorithm starts selecting parallel loops from inner to outer in the contiguous tensor. Once all loops are selected, and if they are less than three, the algorithm proceeds to select parallel indices from the other input tensor (`t2` in Figure 3.2). It will choose from innermost loops (last indices for this tensor) to the outermost. This is because the outermost loops carry data reuse for one of the input tensors, which is more likely to access data that resides in the cache. Accessing consecutive portions of the data linearizes the access among threads and blocks and optimize the data reuse in cache. The data reuse comes from the likelihood that the data for $Thread_i$ is in cache due to the $Thread_{i-1}$ access.

3.1.3 Optimizations at Thread Level

Tensor contraction problems have multiple dimensions (i.e., array dimensions and loop levels). It is possible that, after the thread and block decomposition is created, nested loops can appear inside the kernel of the GPU for performing the operations on multiple regions of the data. The decision algorithm identifies these loops and applies different optimizations to achieve higher performance at the thread level. If the data layout specified by the user is column major order, accessing it can cause an increase in cache misses. This is because CUDA access the data in row major order. The algorithm will permute the parallel loops inside a kernel if the data layout is column major to reduce the misses. Tensor contraction operations can contain data that can be reused at thread levels. The algorithm copies these data to the registers of the GPU to reduce communication with global memory.

In some cases, as in nek5000, the computation of a tensor contraction is divided into multiple kernels. If two or more kernels share the same footprint of the loops and they do not have a fusion-preventing data dependence, the algorithm will fuse them. This optimization reduces the amount of iterations to be performed as well the number of kernel calls. After creating these transformations, the interface tests the different variants of thread and block

decomposition to find the one with the best performance. This test will generate a master recipe that is going to be used to create the specialized recipes.

3.1.4 Master Recipe and Specialized Recipes

Figure 3.4 represents an example of the master recipe generated for sd_t_d1_1. In this example loops `h3` produce the memory coalescing in the data related to tensor `v2`, which is set as $Thread_x$. Loops `h2` ($Block_x$) and `p6` ($Block_y$) allow contiguous access across blocks, while loop `h1` ($Thread_y$) exploits reuse of the data available for the tensor `t2`. The iterations of `p4` and `p5` are permuted since the data layout is column major. The data related to the computation of loop `h7` can be copied into register to exploit reuse.

The information presented by the master recipe describes the possible thread and block decomposition to be considered, as well the data placement. It is possible to apply multiple transformations at the thread level to improve the performance by exploring multiple unroll factors. The different optimization strategies that can be applied at thread level produce distinct versions of the code. For the purpose of this work, the loops that carry dependences are unrolled by unroll factors that evenly divide the size of the iteration space. Only these factors are considered because it prevents unbalanced loops as well as the introduction of conditionals.

Figure 3.5 represents the information for generating a collection of recipes for sd_t_d1_1 with size 16 in each dimension. Since `h7` is the loop to be unrolled, then the multiple recipes are created by the following unroll factors: no unrolling, two, four, eight and fully unroll (16). The tool will test these different parameters to find the fastest implementation. Chapter 4 explains in details how these tests are performed.

```
permute_loops("p5","p4"))
cuda_decomposition(block={"h2","p6"},thread={"h3","h1"})
copy_to_registers("h7")
```

Figure 3.4: Master recipe generated for the sd_t_d1_1 example.

```
permute_loops("p5","p4"))
cuda_decomposition(block={"h2","p6"},thread={"h3","h1"})
copy_to_registers("h7")
unroll_(h7)
unroll_factors(no unroll, 2, 4, 8, full unroll)
```

Figure 3.5: Expression of the specialized recipes generated for sd_t_d1_1.

3.1.5 Shared Memory versus Cache

The steps for performing autotuning in this work do not use other optimization strategies mentioned in the decision algorithm by Khan et al. [21, 22] since the data sizes are small. Loop tiling can change the thread and block decomposition so that there are not enough threads to comprise the GPU scheduling unit (i.e., a warp). Also, the decision algorithm optimizes only for one input tensor (contiguous tensor), allowing the other input to access data in an arbitrary order. This often results in one of the tensors not having coalesced memory accesses.

Khan et al. alleviate the problem related to the noncontiguous tensor by copying such data in coalesced order into shared memory. We do not perform this optimization due to the overhead introduced by copying the data as well thread synchronization. To better understand this, Figure 3.6 presents the code for sd_t_d1_1 performing on the GPU with size

```
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
double newVar;
__device__ __shared__ double _P1[16];
for(int p5 = 0; p5 <= 15; p5 += 1)
 for (int p4 = 0; p4 <= 15; p4 += 1){
   newVar = T3[tx + 16*(ty + 16*(by + 16*(bx + 16*(p5 + 16*p4))))];
   _P1[tx] = T2i[tx + 16*p4 + 4096*by + 256*p5];
   __syncthreads();
   for (int h7 = 0; h7 <= 15; h7 += 1)
     newVar -= _P1[h7] * v2[tx + 16 * (ty + 16 * (bx + 16 * h7))];
   __syncthreads();
   T3[tx + 16*(ty + 16*(by + 16*(bx + 16*(p5 + 16*p4))))] = newVar;
 }
```

(a) CUDA implementation of the sd_t_d1_1 using shared memory.

```
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
double newVar;
for(int p5 = 0; p5 <= 15; p5 += 1)
 for(int p4 = 0; p4 <= 15; p4 += 1){
   newVar = T3[tx + 16*(ty+16*(by+16*(bx+16*(p5+16*p4))))];
   for(int h7 = 0; h7 <= 15; h7 += 1)
      newVar -= T2i[h7 + 16*p4 + 4096*by + 256*p5]
                * v2[tx + 16*(ty + 16*(bx + 16*h7))];
   T3[tx + 16*(ty + 16*(by + 16*(bx + 16*(p5 + 16*p4))))] = newVar;
 }
```

(b) CUDA implementation of the sd_t_d1_1 without using shared memory.

Figure 3.6: CUDA implementations of the sd_t_d1_1 with (a) and without (b) using shared memory.

16. The use of shared memory, as presented in Figure 3.6a, adds a thread synchronization inside the loop nest. This compromises the time spent in the kernel in comparison with the implementation that does not use shared memory (Figure 3.6b).

Optimizing the shared memory to reduce synchronization leads to changing the thread and block decomposition. Figure 3.7 presents the code optimized for shared memory that reduces the synchronization; it sets loop `h3` as $Thread_x$, `p4` as $Thread_y$, `p5` as $Block_x$ and `h1` as $Block_y$. This implementation allows memory coalescing in global memory and places the contiguous data in shared memory. It also performs memory padding to prevent shared memory bank conflicts. This version changes the cache behavior because data in outer dimensions are no longer in contiguous order. Figure 3.8 presents a bandwidth analysis performed by the NVIDIA Visual Profiler [26] for the codes presented in Figure 3.7 (shared memory) and Figure 3.6b (nonshared memory). The bandwidth for the L2 cache is 43.14 GB/s in the shared memory implementation, and the nonshared version is 119.49 GB/s; while in L1 cache we have 226.8 GB/s and 469.44 GB/s, respectively. In device memory we have 36.05 GB/s for the shared memory implementation and 43.12 GB/s for the nonshared version. This demonstrates that the thread and block decomposition of the nonshared memory implementation better uses the cache.

3.2 Output

The master recipe and specialized recipes examples presented before (Figures 3.4 and 3.5, respectively) are representations used by the tool for storing the optimization strategies internally. Figure 3.9 shows an example of the annotation system that will be used with the C++ generated in Chapter 2 for code generation. It presents the annotations generated for the coupled cluster (Figure 3.9a) and the spectral elements (Figure 3.9b).

```
__device__ __shared__ double _P1[16*17];
double newVar;
_P1[tx + 17 * ty] = T2i[tx + 16 * ty + 4096 * by + 256 * bx];
__syncthreads();
for(int p6 = 0; p6 <= 15; p6 += 1) {
 for(int h2 = 0; h2 <= 15; h2 += 1) {
  newVar = T3[tx + 16 * (h2 + 16 * (by + 16 * (p6 + 16 * (bx + 16 * ty))))];
  for(int h7 = 0; h7 <= 15; h7 += 1){
   newVar -= _P1[h7 + 16 * ty] * v2[tx + 16 * (h2 + 16 * (p6 + 16 * h7))];
  }
  T3[tx + 17 * (h2 + 16 * (by + 16 * (p6 + 16 * (bx + 16 * ty))))] = newVar;
 }
}
```

Figure 3.7: CUDA implementation of sd_t_d1_1 optimized for shared memory.

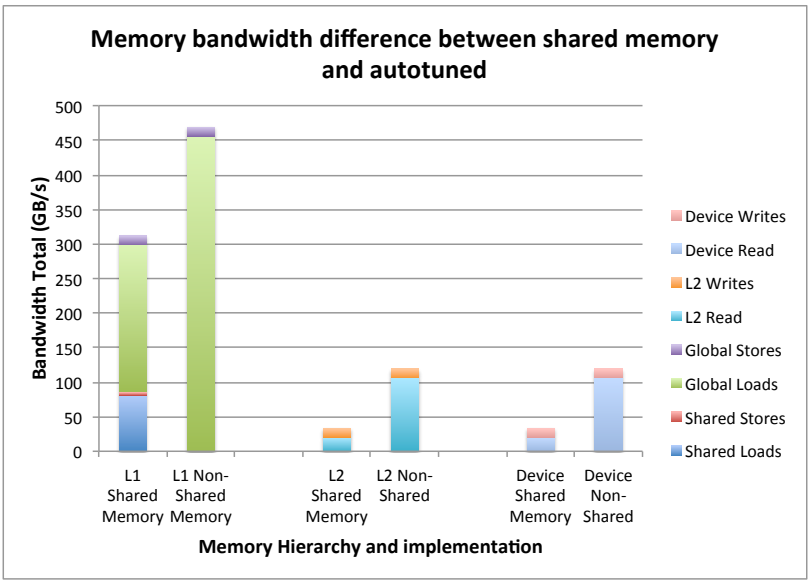

Figure 3.8: Bandwidth difference among memory hierarchy between optimized shared and nonshared memory implementations.

```
UF0[] = [1,2,4,8,16];

permute(0,("p5","p4","p6","h1",
    "h2","h3","h7"))
cuda(0,block={"h2","p6"},
    thread={"h3","h1"})
registers(0,"h7")
unroll(0,"h7",UF0)
```

(a) Output generated by the decision algorithm for sd_t_d1_1.

```
UF0[] = [1,2,5,10];
UF2[] = [1,2,5,10];

cuda(0,block={"e","j"},
       thread={"k","i"})
cuda(2,block={"e","j"},
       thread={"k","i"})

registers(0,"m")
registers(1,"m")
registers(2,"m")

fuse(0,1)
unroll(0,"m",UF0)
unroll(2,"m",UF2)
```

(b) Output generated by the decision algorithm for nek5000.

Figure 3.9: Annotations generated by the decision algorithm for sd_t_d1_1 (a) and nek5000 (b).

Table 3.1 presents the definition of each annotation created. The statement number is assigned by the system automatically depending on the order, from 0 to $N-1$. The transformation will be applied by the order of the numbers. For example, in the nek5000 example it will first work on statement 0 ($ur_{e,i,j} = \sum_k d_{i,k} \times u_{e,k,j}$), transforming it into CUDA. Then the data that can be reused are moved to registers. Statements 0 and 1 are fused since they share the same iteration space and are unrolled by the UF_0 factors. This

Table 3.1: Definitions of the annotations generated by the decision algorithm.

Definitions of the annotations		
Transformation	Parameters	Meaning
UF_N	list with integers	list of unroll factors
`permute`	statement number (integer) new loops order (strings)	permute the loops of statement N to the new order
`cuda`	statement number (integer) loops for blocks (strings) loops for threads (strings)	Transform statement N into CUDA with the proper threads and blocks
`registers`	statement number (integer) loop (string)	Copy the data of statement N at loop level X to registers
`fuse`	statements	Fuse the iterations of the specified statements
`unroll`	statement number (integer) loop level (string) UF_N or amount (integer)	Unroll loop X at statement N with the specified amount

will be performed for statement 2.

The annotation system can be divided into two stages of the decision algorithm. The `permute`, `cuda`, `registers` and `fuse` annotations represent the master recipe, while `unroll` with the list of factors (UF_i) serve as the customized recipes. This annotation will be used by the interface for code generation, which is explained in Chapter 4.

3.3 Summary and Contributions

This chapter presents a decision algorithm that employs autotuning for finding the best-performing implementation in the search space. It focuses on automatically generating a thread and block decomposition optimized for global memory coalescing. This decomposition allows reusing the data located in cache memory rather than optimizing for shared memory. Finally, it applies standard transformations like permute and unroll to further optimize the performance.

The main contribution presented in this chapter is an algorithm that generates the proper decisions for the thread and block decomposition oriented to small tensor contraction problems. It demonstrates that tuning for shared memory on the GPU for these problems does not produce the same performance behavior as expected from large matrix multiplication. The algorithm benefits from having small iterations, which makes the data reuse available in the L1/L2 cache preferable to the transfer from shared memory to registers.

CHAPTER 4

CUDA CODE GENERATION

In the previous chapter we presented how the interface automatically generates loop transformations recipes using a decision algorithm. This chapter describes how it converts the information from the decision algorithm to CUDA code. The code generation in this work is divided in two parts: code transformation and performance measurement.

The rest of this chapter is organized as follows. Section 4.1 shows how the output from the decision algorithm is transformed into CUDA code, and Section 4.2 presents the performance measurement. In Section 4.3 a summary and the contributions from this chapter are presented.

4.1 Code Transformation

The interface uses the Composing High-Level Loop Transformations Tool [23, 17] (CHiLL) to perform code transformation. CHiLL is a source-to-source compiler based on the polyhedral transformation [27, 28] and code generation framework. It uses transformation recipes that specify a series of transformations and applies them over a loop nest. CHiLL has an extension, CUDA-CHiLL [20, 29] that permits users to generate CUDA code from a sequential loop nest. The resultant code will replace the original loop nest to execute the kernel on a GPU. The loop transformations expressed in the recipe can be applied to the GPU code. Figure 4.1 presents an example of a CUDA-CHiLL recipe using the spectral element problem. At a high-level, this recipe fuses the computations of Ur and Us, and generates two CUDA kernels: one for the fused loop nests and another for Ut. Then, it copies the data that have been reused at loops level m to the registers, and unrolls them, for both CUDA kernels, by 2. Table 4.1 presents the annotation that generates the command specified in the recipe.

The output of a CUDA-CHiLL script is a function that contains all the necessary information for CUDA (data allocation/copy, kernels and kernel calls). In case there are multiple statements specified, this will generate a kernel for each statement with the corresponding

```
init("_orio_chill_.c","local_grad3",0)
dofile("cudaize.lua")

nelt=100
lz=10
lx=10
ly=10

distribute({0,2},1)
distribute({0,1},4)
cudaize(0,"local_grad3_GPU_0"{D=lx*ly,ut=nelt*lx*ly*lz,
 us=nelt*lx*ly*lx,ur=nelt*lx*ly*lz,u=nelt*lx*ly*lx,
 Dt=lx*ly},{block={"e","j"},thread={"i","k"}},{})
cudaize(2,"local_grad3_GPU_2"{D=lx*ly,ut=nelt*lx*ly*lz,
 us=nelt*lx*ly*lx,ur=nelt*lx*ly*lz,u=nelt*lx*ly*lx,
 Dt=lx*ly},{block={"e","j"},thread={"i","k"}},{})
copy_to_registers(0,"m","ur")
copy_to_registers(1,"m","us")
copy_to_registers(2,"m","ut")
fuse({0,1,2,3,4,5},4)
unroll(0,4,2)
unroll(2,4,2)
```

Figure 4.1: Example of generated CUDA-CHiLL recipe for nek5000. The recipe uses the unroll factors 2 and 2 to generate CUDA kernels.

Table 4.1: Definitions of the CUDA-CHiLL commands used in the nek5000 recipe.

Definitions of CUDA-CHiLL commands	
Command	Annotation
`distribute`	automatically generated when multiple statements are specified; distributes the loops in different kernels
`cudaize`	`cuda`
`copy_to_registers`	`registers`
`fuse`	`fuse`
`unroll`	`unroll`

data copy. For example, Figure 4.2 presents the function created by CUDA-CHiLL using the recipe presented in Figure 4.1, and Figure 4.3 shows the kernels generated. In the `local_grad3` function, the kernel calls (`local_grad3_GPU_0` and `local_grad3_GPU_2`) refer to the statements of the input file presented in Figure 2.5a.

In the example of Figure 4.2, the computation of each statement has the following structure: starts with memory allocation (`cudaMalloc`), then data copy to device (`cudaMemcpy`), kernel call (`local_grad3_N_GPU`), data copy to host and finally memory free (`cudaFree`). This structure causes a performance degradation due to the costly data copy from/to the host. The information from the arrays `u`, `D` and `Dt` are reused in all kernel calls. In other words, the function is doing two unnecessary memory allocations, copies and frees that

```
void local_grad3(double *ut,double *us,double *ur,double *D,double *u,
 double *Dt){
  cudaMalloc(((void **)(&devO1Ptr)),100000 * sizeof(double ));
  cudaMemcpy(devO1Ptr,ur,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMalloc(((void **)(&devO2Ptr)),100000 * sizeof(double ));
  cudaMemcpy(devO2Ptr,us,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMalloc(((void **)(&devI1Ptr)),100 * sizeof(double ));
  cudaMemcpy(devI1Ptr,D,100 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMalloc(((void **)(&devI2Ptr)),100000 * sizeof(double ));
  cudaMemcpy(devI2Ptr,u,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMalloc(((void **)(&devI3Ptr)),100000 * sizeof(double ));
  cudaMemcpy(devI3Ptr,Dt,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  dim3 dimGrid0 = dim3(100,10);
  dim3 dimBlock0 = dim3(10,10);
  local_grad3_GPU_0<<<dimGrid0,dimBlock0>>>(devO1Ptr,devO2Ptr,devI1Ptr,devI2Ptr,
      devI3Ptr);
  cudaMemcpy(ur,devO1Ptr,100000 * sizeof(double ),cudaMemcpyDeviceToHost);
  cudaMemcpy(us,devO2Ptr,100000 * sizeof(double ),cudaMemcpyDeviceToHost);
  cudaFree(devO1Ptr);
  cudaFree(devO2Ptr);
  cudaFree(devI1Ptr);
  cudaFree(devI2Ptr);
  cudaFree(devI3Ptr);

  cudaMalloc(((void **)(&devO1Ptr)),100000 * sizeof(double ));
  cudaMemcpy(devO1Ptr,ut,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMalloc(((void **)(&devI1Ptr)),100000 * sizeof(double ));
  cudaMemcpy(devI1Ptr,u,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMalloc(((void **)(&devI2Ptr)),100 * sizeof(double ));
  cudaMemcpy(devI2Ptr,Dt,100 * sizeof(double ),cudaMemcpyHostToDevice);
  dim3 dimGrid1 = dim3(100,10);
  dim3 dimBlock1 = dim3(10,10);
  local_grad3_GPU_2<<<dimGrid1,dimBlock1>>>(devO1Ptr,devI1Ptr,devI2Ptr);
  cudaMemcpy(ut,devO1Ptr,100000 * sizeof(double ),cudaMemcpyDeviceToHost);
  cudaFree(devO1Ptr);
  cudaFree(devI1Ptr);
  cudaFree(devI2Ptr);
}
```

Figure 4.2: Function generated by CUDA-CHiLL using the nek5000 recipe.

increase the execution time.

The interface modifies the output from CUDA-CHiLL automatically to reuse the data where safe and reduce the transfers between GPU and host memory. It creates a table with the arrays (host) variables and the corresponding device variables. Those device variables that are repeated will be renamed by just increasing the specified number; for example, since `devI2Ptr` is `u` in the first kernel call, the interface will change `Dt` to `devI3Ptr` in the second call. Followed by this, it will rename the device variable to the first created. This means that, using the same example variables, the interface will rename `u` as `devI2Ptr` in the second kernel. The proper changes will be done to the kernel calls too, for example, `local_grad3_GPU_2` call will be changed to

```
__global__ void local_grad3_GPU_0(double *ur,double *us,double *D,double *u,
 double *Dt){
  newVariable0 = ur[1000 * bx + 10 * ty + 100 * by + tx];
  newVariable1 = us[1000 * bx + 10 * ty + 100 * by + tx];
  for (k = 0; k <= 8; k += 2) {
    newVariable0 = newVariable0  + D[m * 10 + tx]
              * u[bx * 10 * 10 * 10 + by * 10 * 10 + ty * 10 + m];
    newVariable0 = newVariable0 + D[(m + 1) * 10 + tx]
                 * u[bx * 10 * 10 * 10 + by * 10 * 10 + ty * 10 + (m + 1)];
    newVariable1 = newVariable1 + u[bx*10*10*10 + by*10*10 + m*10 + tx]
              * Dt[ty*10 + m];
        newVariable1 = newVariable1 + u[bx*10*10*10 + by*10*10 + (m+1)*10 + tx]
                 * Dt[ty*10 + (m + 1)];
 }
 ur[1000 * bx + 10 * ty + 100 * by + tx] = newVariable0;
 us[1000 * bx + 10 * ty + 100 * by + tx] = newVariable1;
}

__global__ void local_grad3_GPU_2(double *ut,double *u,double *Dt){
 newVariable2 = ut[1000 * bx + 100 * ty + 10 * by + tx];
 for (m = 0; m <= 8; m += 2) {

   newVariable2 = newVariable2 + u[bx*10*10*10 + m*10*10 + by*10 + tx]
                * Dt[ty*10 + m];
   newVariable2 = newVariable2 + u[bx*10*10*10 + (m+1)*10*10 + by*10 + tx]
                * Dt[ty*10 + (m+1)];

  }
  ut[1000 * bx + 100 * ty + 10 * by + tx] = newVariable2;
}
```

Figure 4.3: CUDA kernels generated by CUDA-CHILL using the nek5000 recipe.

`local_grad3_GPU_2<<<dimGrid2,dimBlock2>>>(devO3Ptr,de- vI2Ptr,devI3Ptr).` The repeated entries from the table are removed to prevent the introduction of unnecessary instructions.

Once the variables are replaced, the interface reorganizes the structure of the function. It uses the variables in the table mentioned before to create the data allocation and copy instructions. Since the repeated variables were removed, the new set of instructions reuses the data available in the GPU. Then, it adds the kernel with the proper thread and block declarations and the synchronization call (`cudaThreadSynchronize()`). After all kernels are added, the interface will add the instructions necessary for copying the data to the host and freeing it. Figure 4.4 presents the modified CUDA-CHiLL function created by the interface. This version of `local_grad3` reduces the amount of operations to be performed related to memory allocation and frees from eight to six operations, and the data copy instructions from eleven to nine. In other words, the modification removed four unnecessary operations by modifying the code to exploit data reuse on the GPU.

The interface also performs modifications to the generated kernels from CUDA-CHiLL. It

```
void local_grad3(double *ut,double *us,double *ur,double *D,double *u,
 double *Dt){
  cudaMalloc(((void **)(&devO1Ptr)),100000 * sizeof(double ));
  cudaMalloc(((void **)(&devI1Ptr)),100 * sizeof(double ));
  cudaMalloc(((void **)(&devI2Ptr)),100000 * sizeof(double ));
  cudaMalloc(((void **)(&devO2Ptr)),100000 * sizeof(double ));
  cudaMalloc(((void **)(&devI3Ptr)),100 * sizeof(double ));
  cudaMalloc(((void **)(&devO3Ptr)),100000 * sizeof(double ));
  cudaMemcpy(devO1Ptr,ur,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMemcpy(devI1Ptr,D,100 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMemcpy(devI2Ptr,u,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMemcpy(devO2Ptr,us,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMemcpy(devI3Ptr,Dt,100 * sizeof(double ),cudaMemcpyHostToDevice);
  cudaMemcpy(devO3Ptr,ut,100000 * sizeof(double ),cudaMemcpyHostToDevice);
  dim3 dimGrid0 = dim3(100,10);
  dim3 dimBlock0 = dim3(10,100);
  local_grad3_GPU_0<<<dimGrid0,dimBlock0>>>(devO1Ptr,devI1Ptr,devI2Ptr);
  cudaThreadSynchronize();
  dim3 dimGrid1 = dim3(100,10);
  dim3 dimBlock1 = dim3(10,10);
  local_grad3_GPU_2<<<dimGrid1,dimBlock1>>>(devO2Ptr,devI2Ptr,devI3Ptr);
  cudaThreadSynchronize();
  cudaMemcpy(ur,devO1Ptr,100000 * sizeof(double ),cudaMemcpyDeviceToHost);
  cudaMemcpy(us,devO2Ptr,100000 * sizeof(double ),cudaMemcpyDeviceToHost);
  cudaMemcpy(ut,devO3Ptr,100000 * sizeof(double ),cudaMemcpyDeviceToHost);
  cudaFree(devI1Ptr);
  cudaFree(devO3Ptr);
  cudaFree(devO2Ptr);
  cudaFree(devO1Ptr);
  cudaFree(devI2Ptr);
  cudaFree(devI3Ptr);
}
```

Figure 4.4: CUDA-CHiLL output function modified by the interface related to nek5000.

will search through each kernel generated to linearize the unrolled expressions. For example, rather than `var = var + A[i]` and `var = var + A[i+1]`, it will be changed to one statement as `var = var + A[i] + A[i+1]`. This optimization will allow the compiler to reduce the copy from the register `var` as well as the number of accumulations to be performed. Figure 4.5 shows the kernel calls after the modifications are applied.

4.2 Performance Measurement

As mentioned before, the interface needs to measure the performance of each specified parameter created by the decision algorithm. This means that the interface must create, modify and test a CUDA-CHiLL script for each unroll factor presented in the output from the decision algorithm. The focus of this section is to automatically be able to generate these scripts and measure the output. The interface uses Orio [24, 25] to perform these tasks.

Orio is a tool that allows the user to insert annotations in their codes for specifying

```
__global__ void local_grad3_GPU_0(double *ur,double *us,double *D,double *u,
double *Dt){
 newVariable0 = ur[1000 * bx + 10 * ty + 100 * by + tx];
 newVariable1 = us[1000 * bx + 10 * ty + 100 * by + tx];
 for (k = 0; k <= 8; k += 2) {
    newVariable0 = newVariable0  + D[m * 10 + tx]
      * u[bx*10*10*10 + by*10*10 + ty*10 + m] + D[(m+1)* 10 + tx]
      * u[bx*10*10*10 + by*10*10 + ty*10 + (m+1)];
    newVariable1 = newVariable1 + u[bx*10*10*10 + by*10*10 + m*10 + tx]
      * Dt[ty*10 + m] + u[bx*10*10*10 + by*10*10 + (m+1)*10 + tx]
      * Dt[ty*10 + (m+1)];
 }
 ur[1000 * bx + 10 * ty + 100 * by + tx] = newVariable0;
 us[1000 * bx + 10 * ty + 100 * by + tx] = newVariable1;
}

__global__ void local_grad3_GPU_2(double *ut,double *u,double *Dt){
 newVariable2 = ut[1000 * bx + 100 * ty + 10 * by + tx];
 for (m = 0; m <= 8; m += 2) {
   newVariable2 = newVariable2 + u[bx*10*10*10 + m*10*10 + by*10 + tx]
      * Dt[ty*10 + m] + u[bx*10*10*10 + (m+1)*10*10 + by*10 + tx]
      * Dt[ty*10 + (m+1)];

  }
  ut[1000 * bx + 100 * ty + 10 * by + tx] = newVariable2;
}
```

Figure 4.5: Nek5000 CUDA kernels modified by the interface.

different parameterized transformations. It creates a skeleton code to test these parameters, execute a search over them and determine the best implementation. The search can be specialized by the user by invoking different techniques like exhaustive search, nelder-mead simplex [30], simulated annealing [31], among others.

Orio provides support to work in collaboration with other tools by creating Python modules. We created an external module that calls CUDA-CHiLL for code transformation. Figure 4.6 presents the interaction between CUDA-CHiLL and Orio. The output generated in Chapter 3 is used by this module to generate the annotations required by Orio. Figure 4.7 presents an Orio example for the spectral element problem. The annotations required by Orio are divided in two parts: `PerfTuning` and `CHiLL`. `PerfTuning` handles the definitions and arguments for tuning the experiments, including the compiler building commands, input variables and optimization parameters. In this example, the definition `performance_params` will use the unroll factors created by the autotuner as the parameters to be studied. `CHiLL` transforms each annotation into the proper CUDA-CHiLL instruction as presented in Table 4.1.

Orio automatically generates each script containing the variations specified by the parameters. These scripts will go through the process of code generation and modification

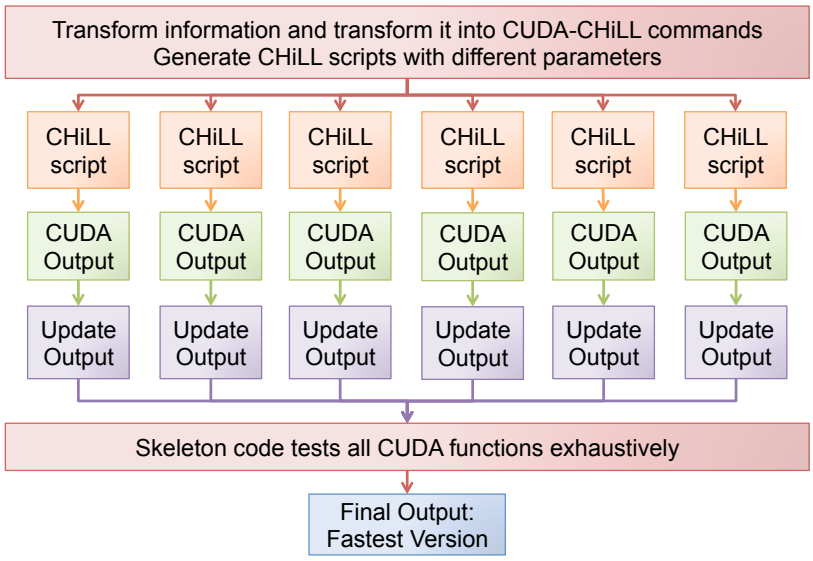

Figure 4.6: Flowchart representing the interaction between Orio and CUDA-CHiLL. Red rectangles present the process performed in Orio, orange is the part performed in CHiLL, green is the output from CHiLL, purple is the modified output and blue is the final output.

as explained in Section 4.1. It uses the skeleton code to measure the performance of each implementation and select the best version. This version will be passed to the interface, which will hand it to the user as the final output. The user must replace the original function on the application with the new implementation.

4.3 Summary and Contributions

This chapter presents a system for selecting the best implementation among the transformations found by the decision algorithm. It uses a combination of two autotuning tools, Orio and CUDA-CHiLL, to evaluate the performance associated with the parameters generated by the decision algorithm. It also automatically optimizes the performance of the generated functions by removing unnecessary data copies, memory allocations and frees, and reusing data available in the GPU. After measuring each implementation exhaustively, it decides the best performing version as the final output to the user.

The contribution in this chapter is a mutual benefit for both tools. Orio was expanded to support CUDA loop optimization as well code transformation by using CUDA-CHiLL. On the other hand, the annotation system allows us to automatically study multiple CUDA-CHiLL scripts using one representation. Also, the interface presents a system for modifying the output from CUDA-CHiLL that reduces the amount of operations to be performed. This feature can be integrated to future versions of CUDA-CHiLL for better code generation.

```
/*@ begin PerfTuning (
 def performance_params {
    param UF0[] = [1,2,5,10];
    param UF2[] = [1,2,5,10];
 }
) @*/
/*@ begin CHiLL (
        cuda(0,block={"e","j"},thread={"k","i"})
        cuda(2,block={"e","j"},thread={"k","i"}
        registers(0,"k")
        registers(1,"m")
        registers(2,"k")
        fuse(0,1)
        unroll(0,"k",UF0)
        unroll(2,"k",UF2)
) @*/

  for(e = 0; e < nelt; e++){
   for(j = 0; j < lx*ly; j++)
    for(i = 0; i < lz; i++)
     for(k = 0; k < lx; k++)
      ur[e*lx*ly*lz + j*lz + i]  +=
           D[k*ly + i] * u[e*lx*ly*lz + j*lz + k];

   for(j = 0; j < lx; j++)
    for(i = 0; i < ly; i++)
     for(k = 0; k < lx; k++)
      for(m = 0; m < ly; m++)
       us[e*lx*ly*lx + j*ly*lx + i*lx + k]  +=
        u[e*lx*ly*lx + j*ly*lx + m*lx + k] * Dt[i*ly+m]);

   for(j = 0; j < lx; j++)
    for(i = 0; i < ly*lz; i++)
     for(k = 0; k < lx; k++)
      ut[e*lx*ly*lz + j*ly*lz + i]  +=
       u[e*lx*ly*lz + k*ly*lz + i] * Dt[j*ly + k];

  }
```

Figure 4.7: Example of the annotation for Orio related to nek5000.

CHAPTER 5

EXPERIMENTAL EVALUATION

This chapter presents the experimental evaluation of the system developed. The results presented are from the two benchmarks based on the scientific applications that have been used throughout this thesis: nekbone and specialized NWChem kernels. Section 5.1 presents the background of the benchmarks, and Section 5.2 discusses the methodology for handling the experiments. Section 5.3 presents the results related to the CUDA code generated by the framework. Section 5.4 discusses the results obtained in OpenACC, and Section 5.5 focuses on the impact in the performance caused by different implementations. Section 5.6 is a summary of this chapter.

5.1 Benchmarks

Nekbone is a mini-app used to study the tensor contraction operations and the communication costs presented in nek5000 [8, 9]. It computes the spectral element solution using the specialized kernel mentioned in Chapter 1 [7]. This computation is performed in a conjugate gradient iteration to solve a 3-dimensional Poisson equation. Each iteration computes the following for each element:

- $Ur_{ijk} = D_{il}U_{ljk}$: recast as a $p \times p$ matrix multiplied by a $p \times p^2$ matrix
- $Us_{ijk} = D_{jl}U_{ilk}$: recast as p matrix-matrix multiplies with $p \times p$ matrices
- $Ut_{ijk} = D_{kl}U_{ijl}$: recast as a $p^2 \times p$ matrix multiplied by a $p \times p$ matrix

After processing the data related to U, nekbone performs the same process in Wr, Ws and Wt. This thesis focuses on the sizes of: $8 \times 8 \times 8$, $10 \times 10 \times 10$ and $12 \times 12 \times 12$, representing $lx \times ly \times lx$ in the code presented in Chapter 2. The *nelt* size used for all experiments was 1000. As with nek5000, nekbone uses tensors of small sizes due to the discretization polynomial [10].

The specialized NWChem kernels are a set of functions created by Jeff Hammond for performance optimization experiments [14]. These kernels recast the computations

of the tensor contraction engine (TCE) coupled cluster for double and triple interactions (CCSD(T)) presented in NWChem [13, 14]. The kernels tested were sd_t_s1, sd_t_d1 and sd_t_d2 with sizes 10, 12 and 16 for each dimension. As commented on Chapter 1, these sizes represent chunks of a larger tensor contraction to be computed in a distributed system.

5.2 Methodology

The experimental evaluation of the codes generated by the system starts by extracting the functions corresponding to the tensor contraction computations of each application. These functions were reproduced by using the user input as explained in Chapter 2. The user inputs were created by hand. The interface automatically generates the different implementations by applying the techniques presented in Chapters 3 and 4; and it selects the fastest code in an average of 100 tests using the skeleton code. For each benchmark, the original kernels were replaced by hand with the outputs produced by the interface. We tested the following versions of each benchmark: sequential, OpenMP (4 Threads), OpenACC, and the CUDA generated code. The average of five runs of each version is used to represent the gigaflops per second. For all versions of each benchmark, the gigaflops and timing information were computed only for the kernels that apply the tensor contraction.

The experiments were conducted on a system with an Intel i7 930-2.8 Ghz and 4 Gigabytes of RAM using Ubuntu 14.04 64-bits. The GPUs tested were Tesla C2050, a Fermi generation architecture, and Tesla K20, a Kepler generation. In this thesis work, the Portland Group compiler (PGI) version 14.3 with OpenACC support and CUDA 5.5 was used to conduct the experiments.

5.3 CUDA Generated Code Results

Results from the experiments related to nekbone are shown in Figure 5.1. Figures 5.2, 5.3, and 5.4 correspond to the three different kernels of NWChem. These results are the measurements of four versions: sequential (blue line, diamond), OpenMP with four threads (red line, square), OpenACC (dashed lines), and the code generated from the system (solid lines) in Fermi (purple lines, circle) and Kepler (green lines, triangles). For the OpenACC version, the OpenACC compiler decided automatically which optimizations should be applied and the thread and block decomposition. This version is referred to as naïve OpenACC. For all cases that use the GPU, the data copying to/from the device was optimized by moving it outside the main computation loops of the benchmarks.

Table 5.1 shows the speedup achieved by the generated kernels for nekbone over the sequential, OpenMP and OpenACC implementations. These implementations achieved up

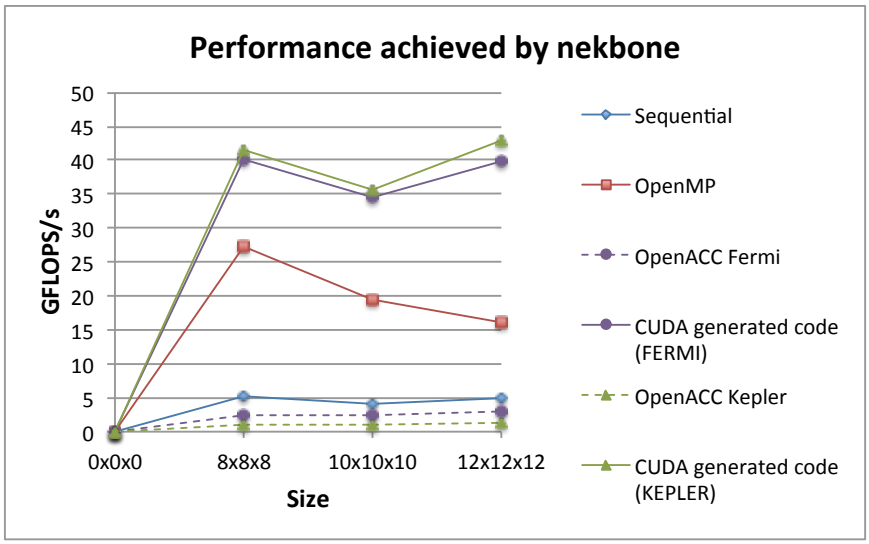

Figure 5.1: Performance achieved by the different implementations of nekbone.

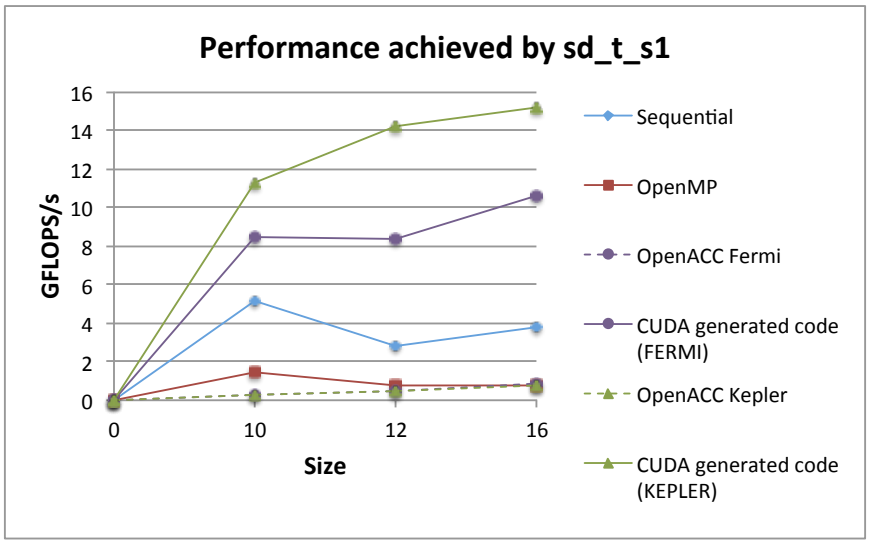

Figure 5.2: Performance achieved by the different implementations of sd_t_s1.

to 8.87x of speedup compared to the sequential implementation. Compared with OpenMP and OpenACC, the generated kernels outperformed these two implementations by 2.68x and 38.18x, respectively.

Table 5.2 presents the speedups obtained by the CUDA code for the kernels related to NWChem. The results of sd_t_s1 show that the generated code increases the performance up to 5.12x compared with the sequential implementation, while OpenMP was surpassed by 18.79x. This CUDA code performed up to 35.21x faster in comparison with OpenACC.

In the case of sd_t_d1, the generated CUDA code improves the performance up to

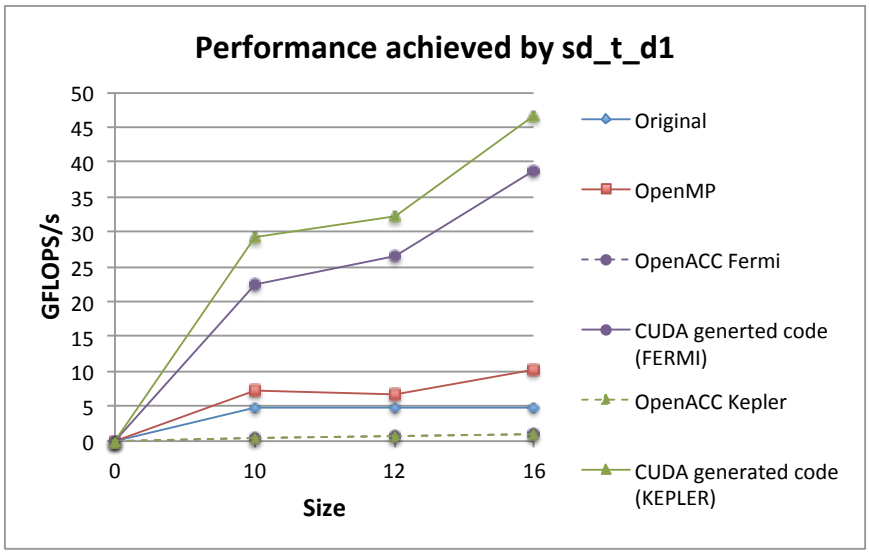

Figure 5.3: Performance achieved by the different implementations of sd_t_d1.

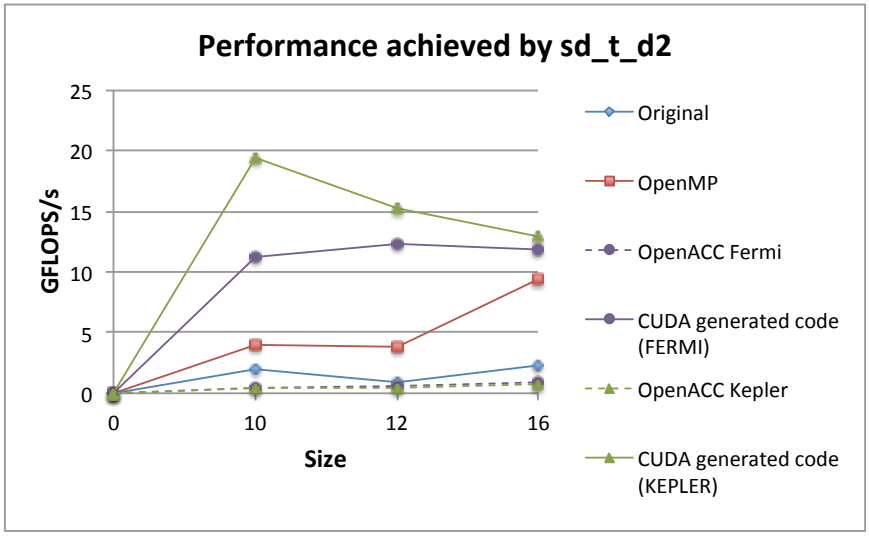

Figure 5.4: Performance achieved by the different implementations of sd_t_d2.

9.78x over the sequential implementation, and 4.49x compared with OpenMP. OpenACC performed up to 70.11x slower than the automatically-produced code. For sd_t_d2, the generated code exceeds the sequential version by 17.62x, and the OpenMP implementation was outperformed by 4.85x. The results also indicate that this implementation performed up to 52.40x faster than the OpenACC version.

Related to the GPU implementations, for most cases the performance scales with tensor size. This relationship can also be seen in the naïve OpenACC implementation. Also, for NWChem, the results show that the benchmarks achieved very different performance

Table 5.1: Speedups achieved by the generated code for nekbone over the sequential, OpenMP and OpenACC implementations.

Speedup achieved by code generated in nekbone				
Size	GPU	Sequential	OpenMP	OpenACC
$8 \times 8 \times 8$	Fermi	7.62x	1.46x	17.16x
	Kepler	7.89x	1.52x	38.18x
$10 \times 10 \times 10$	Fermi	8.56x	1.77x	14.72x
	Kepler	8.81x	1.83x	35.92x
$12 \times 12 \times 12$	Fermi	8.24x	2.48x	13.92x
	Kepler	8.87x	2.68x	36.27x

Table 5.2: Speedups achieved by the code generated for NWChem over the sequential, OpenMP and OpenACC implementations.

Speedup achieved by code generated in NWChem					
Kernel	Size	GPU	Sequential	OpenMP	OpenACC
sd_t_s1	10	Fermi	1.64x	5.85x	25.79x
		Kepler	2.18x	7.79x	35.21x
	12	Fermi	3.02x	10.58x	18.96x
		Kepler	5.12x	17.94x	30.36x
	16	Fermi	2.78x	13.19x	11.98x
		Kepler	3.96x	18.79x	19.06x
sd_t_d1	10	Fermi	4.64x	3.09x	38.86x
		Kepler	6.02x	4.01x	70.11x
	12	Fermi	5.45x	3.91x	33.24x
		Kepler	6.64x	4.77x	53.32x
	16	Fermi	8.15x	3.75x	35.14x
		Kepler	9.58x	4.49x	44.47x
sd_t_d2	10	Fermi	5.79x	2.28x	23.26x
		Kepler	9.95x	4.85x	52.40x
	12	Fermi	14.25x	3.16xx	19.99x
		Kepler	17.62x	3.91x	30.23x
	16	Fermi	5.30x	1.25x	13.53x
		Kepler	5.78x	1.36x	16.65x

ranging from 8.39 GFLOPS/s to 46.57 GFLOPS/s. Although the kernels share the same data structure, each kernel accesses data differently, causing differences in the performance.

5.4 OpenACC Results

For all benchmarks, the naïve OpenACC version was substantially slower than the sequential implementation. The observed performance for nekbone ranges from 0.91 GFLOPS/s to 2.86 GFLOPS/s while the sequential version performed between 4.04 GFLOPS/s to 5.26 GFLOPS/s. The naïve OpenACC implementation of NWChem was also slower than

the sequential version. It ranges from 0.33 GFLOPS/s to 1.10 GFLOPS/s compared with 0.86 GFLOPS/s to 5.17 GFLOPS/s, which is the performance achieved by the sequential version.

The NVIDIA Visual Profiler was used to study what was causing the slowdowns in the naïve OpenACC codes. The results from profiling the benchmarks indicate that the thread and block decomposition generated by the compiler does not use the GPU efficiently. We use the code presented in Figure 5.5, the naïve OpenACC implementation for sd_t_d1_1, as an example to explain this problem. The decomposition generated by the compiler for the case where each dimension has size of 16 is as follows:

	Block	Thread
X	32	1
Y	1	16

The profiler revealed that this decomposition provides only 50% of the warp execution efficiency. In other words, the code does not take advantage of the scheduling unit of the GPU. The performance of OpenACC was optimized by implementing the thread and block decomposition found by the decision algorithm and also explicitly moves the output variable to registers.

Figure 5.6 presents the OpenACC implementation after indicating the following thread and block decomposition:

	Block	Thread
X	16	16
Y	16	16

```
#pragma acc kernels loop present(t2[0:tile4],v2[0:tile4], t3[0:tile6]) {
#pragma acc loop independent
for (int p4=0; p4<p4u; p4++)
#pragma acc loop independent
 for (int p5=0; p5<p5u; p5++)
#pragma acc loop independent
  for (int p6=0; p6<p6u; p6++)
#pragma acc loop independent
   for (int h1=0; h1<h1u; h1++)
#pragma acc loop independent
    for (int h2=0; h2<h2u; h2++)
#pragma acc loop independent
     for (int h3=0; h3<h3u; h3++)
      for (int h7=0; h7<h7u; h7++){
        t3[h3+h3u*(h2+h2u*(h1+h1u*(p6+p6u*(p5+p5u*p4))))] -=
          t2[h7+h7u*(p4+p4u*(p5+p5u*h1))] * v2[h3+h3u*(h2+h2u*(p6+p6u*h7))];
      }
}
```

Figure 5.5: Naïve OpenACC implementation of sd_t_d1_1.

```
#pragma acc kernels loop present(t2[0:tile4],v2[0:tile4], t3[0:tile6]) {
#pragma acc loop independent
for (int p4=0; p4<p4u; p4++)
#pragma acc loop independent
 for (int p5=0; p5<p5u; p5++)
#pragma acc loop independent gang(16)
  for (int p6=0; p6<p6u; p6++)
#pragma acc loop independent gang(16)
   for (int h1=0; h1<h1u; h1++)
#pragma acc loop independent vector(16)
    for (int h2=0; h2<h2u; h2++)
#pragma acc loop independent vector(16)
     for (int h3=0; h3<h3u; h3++)
      for (int h7=0; h7<h7u; h7++)
        t3[h3+h3u*(h2+h2u*(h1+h1u*(p6+p6u*(p5+p5u*p4))))] -=
          t2[h7+h7u*(p4+p4u*(p5+p5u*h1))] * v2[h3+h3u*(h2+h2u*(p6+p6u*h7))];
}
```

Figure 5.6: OpenACC implementation of sd_t_d1_1 with the thread and block decomposition specified.

This optimization allowed OpenACC to increase the memory bandwidth related to the communication between L1 and L2 cache. Also, moving the output variable to the register level, `vard1_1` as presented in Figure 5.7, improved the bandwidth related to device memory. Figure 5.8 presents the difference in memory bandwidth between the naïve implementation and after applying the optimizations. The data reads (red bars) associated with L1/L2 cache increased from 6.593 GB/s to 258.672 GB/s, while the writes (blue) gain

```
#pragma acc kernels loop present(t2[0:tile4],v2[0:tile4], t3[0:tile6]) {
double vard1_1;
#pragma acc loop independent
for (int p4=0; p4<p4u; p4++)
#pragma acc loop independent
 for (int p5=0; p5<p5u; p5++)
#pragma acc loop independent gang(16)
  for (int p6=0; p6<p6u; p6++)
#pragma acc loop independent gang(16)
   for (int h1=0; h1<h1u; h1++)
#pragma acc loop independent vector(16)
    for (int h2=0; h2<h2u; h2++)
#pragma acc loop independent vector(16)
     for (int h3=0; h3<h3u; h3++){
      vard1_1 = t3[h3+h3u*(h2+h2u*(h1+h1u*(p6+p6u*(p5+p5u*p4))))];
      for (int h7=0; h7<h7u; h7++)
       vard1_1 -= t2[h7+h7u*(p4+p4u*(p5+p5u*h1))] *
                  v2[h3+h3u*(h2+h2u*(p6+p6u*h7))];
      t3[h3+h3u*(h2+h2u*(h1+h1u*(p6+p6u*(p5+p5u*p4))))] = vard1_1;
     }
}
```

Figure 5.7: OpenACC implementation of sd_t_d1_1 with computational grid and registers data specified.

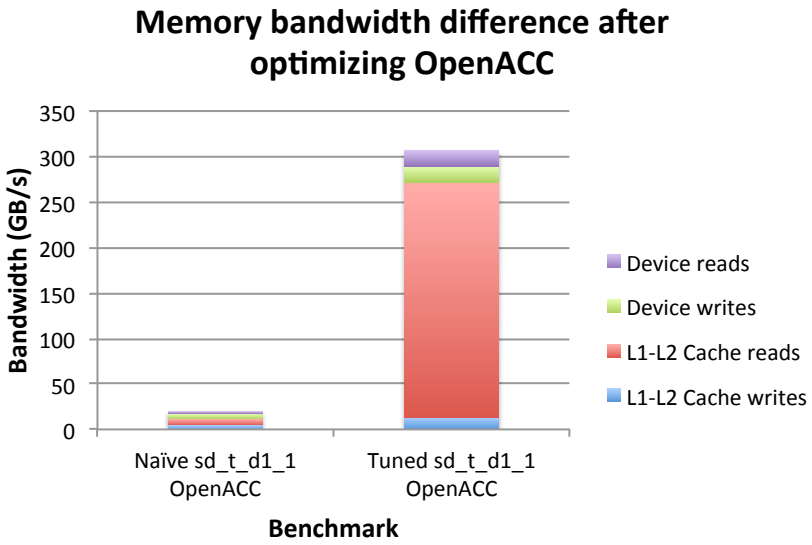

Figure 5.8: Memory bandwidth performance related to naïve and tuned OpenACC implementations.

a boost from 5.023 GB/s to 13.614 GB/s. In the results for the device memory bandwidth, the data reads (purple) grow from 0.408 GB/s to 17.632 GB/s and writes (green) from 6.2 GB/s to 17.001 GB/s. These optimizations improved the data reuse available in cache memory, which reduces the communication with global memory. The profiler also showed that the new implementation uses 100% of the warp execution efficiency. The information copied to registers in the third implementation was specified by hand since the `private` clause in OpenACC did not work properly. It is expected that this problem will be resolved in future OpenACC compiler implementations.

Figure 5.9 presents a comparison between the tuned OpenACC implementations and the codes generated by the interface presented in this thesis. This figure shows the results related to nekbone with size $12 \times 12 \times 12$, and all benchmarks related to NWChem with size 16. In some cases, tuning for OpenACC outperforms the CUDA code created by the framework. For example, the optimized OpenACC version of NWChem sd_t_d1 performed 1.06x faster on Kepler. Also, for sd_t_d2, the optimized OpenACC performed 1.14x faster than the fastest CUDA implementation on Kepler.

Table 5.3 presents the speedups achieved by the different OpenACC implementations over the sequential implementation with the previous sizes. When the computational grid generated by the decision algorithm was specified in the OpenACC implementation of nekbone, the performance increased to 3.74x over the sequential implementation. Moving

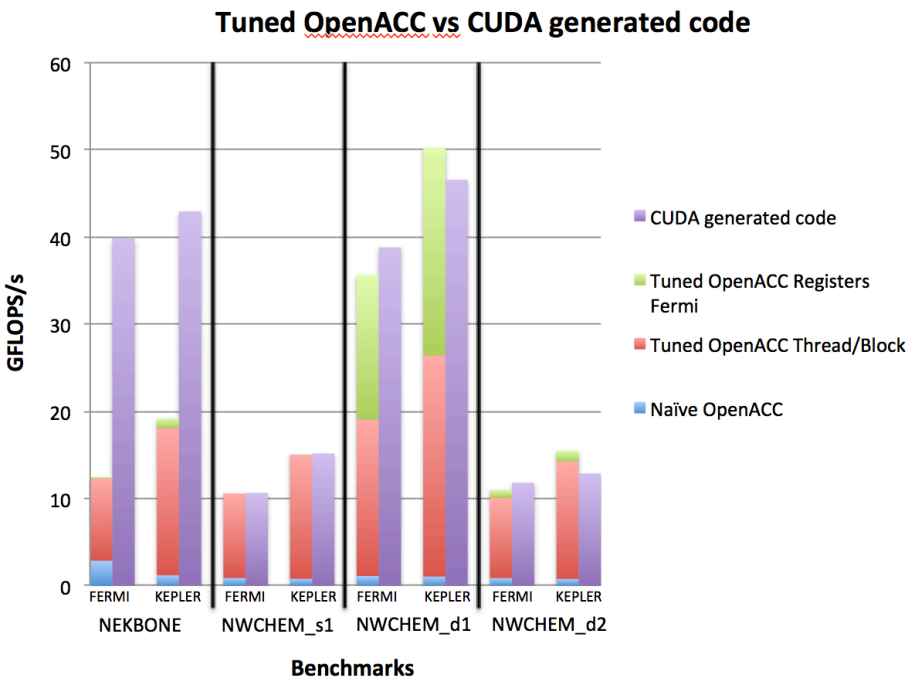

Figure 5.9: Performance difference between optimized OpenACC and the CUDA generated code. The sequential version is used as baseline.

Table 5.3: Speedups achieved by OpenACC after adding the different optimizations.

Speedups achieved by tuning OpenACC				
Benchmark	GPU	Naïve	New Grid	Registers
nekbone	Fermi	0.59x	2.54x	2.56x
	Kepler	0.24x	3.74x	3.97x
sd_t_s1	Fermi	0.23x	2.77x	
	Kepler	0.21x	3.92x	
sd_t_d1	Fermi	0.23x	4.00x	7.25x
	Kepler	0.22x	5.55x	10.34x
sd_t_d2	Fermi	0.39x	4.5x	4.57x
	Kepler	0.35x	6.50x	6.61x

the data that can be reused to the register level increased the performance by 3.97x. In sd_t_s1, providing the thread and block decomposition accelerated the computation up to 3.92x over the sequential implementation. The register level optimization was not used in this benchmark since the output is not accumulated. Specifying the decomposition in sd_t_d1 increased the performance up to 5.55x, and moving the data to registers improved it up to 10.34x over sequential. NWChem sd_t_d2 achieved 6.50x speedup over the sequential implementation after using the thread and block decomposition from the decision algorithm; while moving the reusable data to the registers increased the performance up to 6.65x. These

results suggest that there is a possibility to expand the autotuning for optimizing OpenACC codes.

5.5 Performance Difference due to Implementation

Chapters 1 and 2 mention that the mathematics can have an impact on the performance. For example, the functions that perform tensor contraction on the CPU in nek5000 and nekbone changes the order of the tensors to accelerate the process. The code generated by the interface omits this optimization, as well for the OpenACC code created, since the architecture of the GPU is different. For this reason the generated code treats the computation in nek5000 as a tensor contraction between order 3 and order 2 rather than the computation presented in Section 5.1.

Yet, there is also a difference in the performance related to the code generated by the interface and OpenACC. The results for optimizing OpenACC in nekbone achieved performance of 23.13 GFLOPS/s. Compared with the CUDA implementation (41.50 GFLOPS/s), both using the same optimization strategies, this code performed 1.79x slower. These implementations were profiled with the NVIDIA Visual Profiler and it was found that functions related to the computation of W generated the following decompositions:

	Block	Thread
X	8	8
Y	*nelt*	32
Z	8	-

	Block	Thread
X	10	10
Y	*nelt*	25
Z	10	-

	Block	Thread
X	12	12
Y	12	21
Z	*nelt*	-

These grids present a size of $Thread_Y$ that was not specified, and create a $Block_Z$, which is not supported in CUDA. In other words, although the computational grid was specified, the OpenACC compiler does not always generate the desired grid.

The interface was modified for preventing the loop fusions and generating the same computation as the CPU version of nek5000. The new nekbone code for GPU, referring to CUDA and OpenACC, now performs two tensor contractions of 2^{nd} (Ur, Us) and one tensor contraction of order 3 with order 2 (Us), as presented in Section 5.1. Table 5.4 presents the new thread and block decomposition for OpenACC and the CUDA code generated by the modified framework. Although the new computational grids of OpenACC are more similar compared with the CUDA code, the compiler creates a different grid in some cases.

Figures 5.10 and 5.11 present the performance achieved by the new implementations of nekbone using OpenACC and the new CUDA code generated from the framework. The new version of OpenACC (dotted lines, purple for Fermi and green for Kepler) show it achieved

Table 5.4: Thread and block decomposition generated by the OpenACC compiler and the decision algorithm for nekbone after using the optimizations of CPU.

Thread and block decompositions for nekbone				
Function	Size	Coordinate	OpenACC (Block | Thread)	CUDA (Block | Thread)
Ur (*Wr*)	8	*X*	8 | 8	*nelt* | 1
		Y	*nelt* | 32	8 | 64
	10	*X*	10 | 10	*nelt* | 1
		Y	*nelt* | 25	10 | 100
	12	*X*	12 | 12	*nelt* | 1
		Y	*nelt* | 21	12 | 144
Us (*Ws*)	8	*X*	*nelt* | 8	*nelt* | 8
		Y	8 | 8	8 | 8
	10	*X*	*nelt* | 10	*nelt* | 10
		Y	10 | 10	10 | 10
	12	*X*	*nelt* | 12	*nelt* | 12
		Y	12 | 12	12 | 12
Ut (*Wt*)	8	*X*	*nelt* | 8	*nelt* | 1
		Y	1 | 8	64 | 8
	10	*X*	*nelt* | 10	*nelt* | 1
		Y	1 | 10	100 | 10
	12	*X*	*nelt* | 12	*nelt* | 1
		Y	1 | 12	144 | 12

up to 54.81 GFLOPS/s, making a speedup of 3.27x over the optimized implementation presented in Section 5.4 (dashed lines). In the case of the new CUDA code generated by the framework, the results show that this implementation did not improve the performance. This means that the original code generated by the framework (solid lines) outperformed the new CUDA code (square dots lines) by 1.46x.

Table 5.5 presents the speedup achieved by the new OpenACC implementation over the original CUDA version of nekbone generated by the framework. These results show that the new kernels for OpenACC outperformed the CUDA generated kernels by 1.32x. The CUDA code presents the same behavior related to tensor size and performance: as the work increases, the difference in performance between OpenACC and CUDA reduces. The results for sizes 10 and 12 in Fermi show that the generated CUDA code was up to 1.2x faster than the new OpenACC, while in Kepler this difference is reduced as the size of the problem increases. These results suggest that the OpenACC code requires different optimization strategies compared with CUDA.

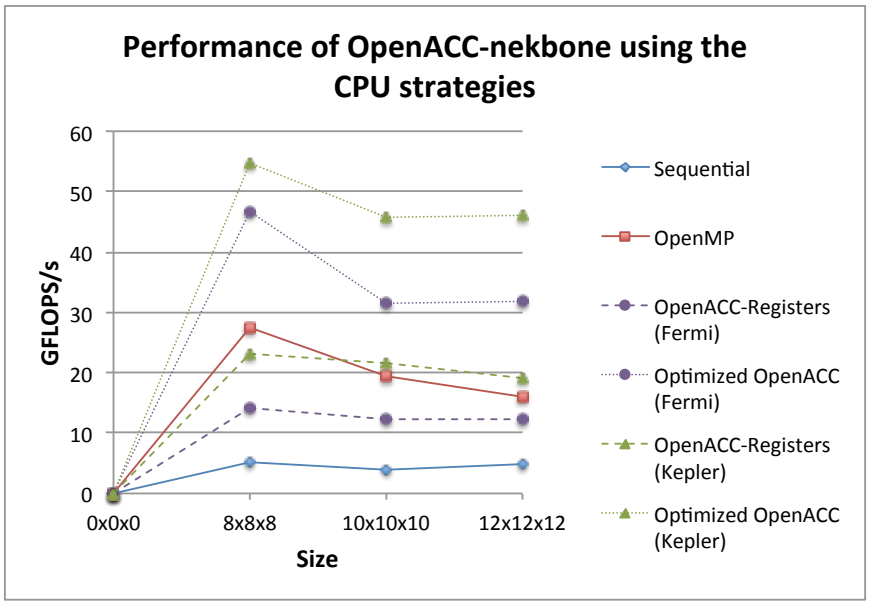

Figure 5.10: Performance achieved by nekbone using the CPU computation strategies in OpenACC.

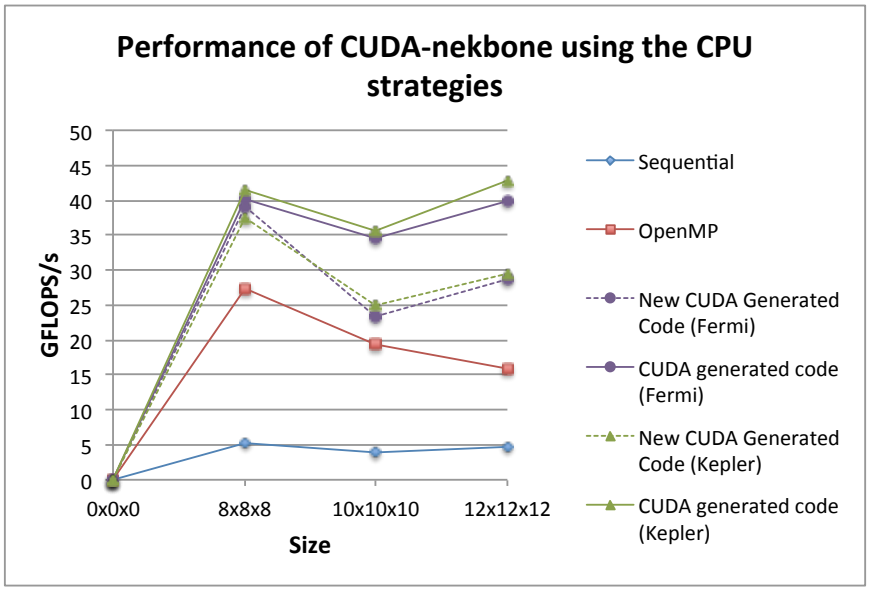

Figure 5.11: Performance achieved by nekbone using the CPU computation strategies in CUDA.

5.6 Summary

This chapter presents the results achieved from the experimental evaluation of the tool presented in this thesis. It demonstrates that when using autotuning with the proper guidance it is possible to achieve high performance in the tensor contraction problems for tensors of small size. Nekbone improved up to 8.87x compared with the sequential

Table 5.5: Speedups achieved by the new OpenACC implementation over the fastest CUDA version of nekbone.

Speedup of new OpenACC over CUDA		
Size	Fermi	Kepler
8	1.16x	1.32x
10	0.91x	1.28x
12	0.80x	1.07x

implementation, while NWChem demonstrated a speedup of 17.62x.

Using the guidance provided by the tool, OpenACC achieved higher performance too. It achieved up to 16.67x of speedup compared with the sequential version. In some cases, OpenACC outperformed the code generated by our tool. These results show that is possible to combine the optimizations from the OpenACC compiler with the decisions found by the decision algorithm to achieve equal or higher performance. We see the potential for combining OpenACC with autotuning to improve its performance.

Finally, this chapter confirms the statement made in Chapter 1: the GPU requires different optimization strategies compared to the CPU. When the interface produced CUDA code using the CPU strategies, it performed up to 1.46x slower than using the optimizations presented in Chapter 3. On the other hand, the OpenACC compiler applies other strategies that alter the specified thread and block decomposition. This observation suggests that the optimizations required to achieve high performance in CUDA should not necessarily be the same for OpenACC. The autotuning studies related to OpenACC should also consider the transformations applied by the compiler to achieve high performance in the GPU.

CHAPTER 6

RELATED WORK

Tensor contraction is an active research area, especially in mathematics, quantum chemistry and computing. Many libraries and tools have been developed for applying mathematical optimization as explained in Chapter 1. These tools focus on large sizes, multiprocessors and distributed systems. Other works are related to accelerating applications that involve tensor contraction operations.

This chapter presents related research based on the topics that this thesis emphasizes. Section 6.1 shows different tools that focus on optimizing tensor contraction problems. Section 6.2 presents different projects based on accelerating applications that use tensor contraction.

6.1 Optimization Tools

6.1.1 Tensor Contraction Engine

The Tensor Contraction Engine (TCE) is a tool for generating FORTRAN code using an input with a Mathematica-style expression [11, 32]. The main focus of TCE is to create an efficient implementation from the input given. It applies a series of mathematical transformations to reduce the number of operations and minimize the requirements to fit the computation storage by applying loop fusion. The output generated by TCE is optimized for minimizing the communication to CPU memory. TCE also can create code targeting multiprocessor machines.

6.1.2 Super Instruction Assembly Language

The Super Instruction Assembly Language (SIAL, pronounced "sail") [33] is a runtime system developed for accelerating the problems related to coupled clusters. It focuses on large dense tensors and parallel architectures. In SIAL, the algorithms are expressed for computing the data using blocks of numbers instead of performing the operations in the

traditional method. This allows the system to handle the data efficiently, as well as permits the runtime system to overlap the computation with communication.

SIAL programs performs over a Super Instruction Processor (SIP), which is a parallel virtual machine in the runtime. The SIP deals with the complexities of a parallel hardware, like I/O operations, super instructions, among others.

6.1.3 Cyclops Tensor Framework

The Cyclops Tensor Framework (CTF) [34] is a runtime system for coupled clusters problems and distributed systems. It allows the user to input the problem using Einstein notation for tensors. The mathematical transformations are applied to the input for optimizing the number of operations. The runtime system will then focus on generating an implementation that efficiently uses the available resources, optimizing memory usage and communication. This system not only finds the best implementation, but also presents to the user how the data should be decomposed in order to achieve higher performance.

6.1.4 Tensor Library

The libtensor [35] is a C++ library of classes for post-HartreeFock electronic structure methods. It can also be used for computing other methods that requires tensor algebra like coupled clusters and equation of motion. The library performs the computation by splitting the work in smaller blocks and applies a parallel divide-and-conquer algorithm to perform the tensor algebra.

6.1.5 Build To Order Blas

The Build To Order Blas (BTO) compiler [36, 37] is a tool focused on linear algebra, yet it can be used for solving problems related to tensor contraction of 1^{st} and 2^{nd} order. This tool permits the users to express the linear algebra problem at a higher level and generate code from this representation. The user inputs a MATLAB like code used for generating multiple versions in C, where each implementation contains different optimization strategies. It tests all implementations to find the best performing code. Also, BTO can rearrange mathematically the input from the user for achieving higher performance, reducing the order of operations or permitting better reuse of the data available in the different memory hierarchies.

6.1.6 Multi-element problems on GPU

Linh Ha et al. [38] present a survey on different methods used for solving multi-element problems. It also explains an efficient methodology for partitioning the computational grid of a GPU depending on the size of the data. The thread and block decomposition will focus on maximizing the computation across threads, blocks or an intermediate implementation to achieve better performance in the given input. The proposed methodology achieved up to 65% of the theoretical bandwidth in comparison with other approaches. Even for problems that focus on sizes from 2×2 to 32×32, the performance achieved was better compared to libraries like *CUBLAS* and *LAPACK*.

6.2 Accelerating Tensor Contraction Applications

6.2.1 Nekbone and Nek5000 on GPU

The nekbone proxy application also has been optimized on GPUs [39]. The CESAR Co-Desing Center created a series of hand-coded OpenCL kernels for tensors of sizes $8 \times 8 \times 8$ to $12 \times 12 \times 12$. Results from these experiments achieved 100-200 GFlops using an NVIDIA GTX 590 (Fermi generation) in the tensor contraction operations.

The CRESTA project [40] created a version of nek5000 that uses multiple GPUs. Results from the experiments achieved a speedup of 1.59x using 512 NVIDIA TESLA K20x (Kepler architecture) GPUs in comparison with a CPU-only implementation (512 nodes with 8192 cores). The tests were performed on the CRAY XK7 *TITAN* system.

6.2.2 Autotuning on Nek5000

CHiLL was used for accelerating the nek5000 application [7] on a CPU. This work used autotuning and specialization for optimizing the matrix multiplication kernel for small matrices. The kernel contains a series of calls tailored for specific sizes that take advantage of different cache and registers, also single instruction multiple data (SIMD) code generation and instruction level parallelism (ILP). The results from this work present a speedup of 2.2x in the CRAY XT5 *JAGUAR* system.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This chapter summarizes the thesis and the contributions it has made. Also, it discusses future work to expand the capabilities of the tool developed.

7.1 Summary and Conclusions

The main conclusion from this thesis is that the optimization strategies for tensor contraction must take into account three aspects: mathematics, computational device and compilers. The mathematical optimizations reduce the number of operations and/or simplify the interaction. The computational device optimizations explore how to efficiently use the available resources. Using compiler optimizations allows extracting more performance from these resources. In difference with the research work presented in Chapter 6, the contributions of this thesis focus on small size problems and GPUs rather than larger tensors and multiprocessors or distributed systems.

This thesis describes an interface where the users can express tensor contraction problems and generate high-performance code for NVIDIA GPUs. The tool focuses on the idea of parallelizing the batched matrix multiplication using a GPU for small tensors. These sizes present a challenge since the optimization strategies required are different compared to large sizes. Also, even though the size of these tensors are small, they can consume a large portion of the time spent in the computation. The tool is divided in three stages, based on the steps mentioned before: *mathematical representation*, *decision algorithm* and *autotuning and code generation*.

The *mathematical representation* allows the user to input the code using an expression that resembles Einstein notation for tensor contraction. The contribution of this stage is that it simplifies the implementation by allowing the user to focus on optimizing the mathematics rather than spending effort in implementing the solution.

The *decision algorithm* exposes the optimization strategies required to achieve high performance on the computational device, which in this thesis is a GPU. The contribution

of this step is an algorithm that creates a thread and block decomposition focused on achieving memory coalescing. It also generates a search space of specific implementations that can achieve high performance on the GPU. We show that tensor contraction problems on GPUs require different strategies for achieving high performance compared to the CPUs.

The *autotuning and code generation* stage explores the possible compiler optimizations and implementations strategies that achieve better performance. We combined two tools (CUDA-CHiLL and Orio) for studying the search space created by the *decision algorithm* and producing high performance CUDA code. The contribution is a mutual benefit for both tools for studying a search space.

Finally, the tool is not limited to CUDA implementations only. In this thesis, the guidance provided by the interface is also applied to OpenACC. The contribution related to OpenACC is a possibility for expanding the autotuning framework to generate high-performance code for this API. In some cases, OpenACC outperfomed the CUDA versions generated by the tool.

7.2 Future Work

The *mathematical representation* in this work depends on a user input. This framework will be attached to a tool that optimizes the mathematics for reducing the number of operations. The extension is referred to as the gray dashed box in Figure 1.3. This work is a collaboration with Thomas Nelson and Dr. Elizabeth Jessup, both from the University of Colorado.

In Chapter 4, it was mentioned that Orio supports other strategies for pruning the search space. The interface will be expanded for using these methods to reduce the time spent in the exhaustive search. This work is in collaboration with Dr. Boyana Norris from the University of Oregon.

REFERENCES

[1] D. Fleish, *A Student's Guide to Vectors and Tensors*. Cambridge, United Kingdom: Cambridge Press, 2011.

[2] K. Dullemond and K. Peeters, "Introduction to tensor calculus," 2010. [Online]. Available: http://www.ita.uni-heidelberg.de/ dullemond/lectures/tensor/tensor.pdf

[3] V. N. Kaliakin, "Brief review of tensors," 2008. [Online]. Available: http://www.ce.udel.edu/faculty/kaliakin/appendix_tensors.pdf

[4] A. T. Patera, "A spectral element method for fluid dynamics: Laminar flow in a channel expansion," *Journal of Computational Physics*, vol. 54, no. 3, pp. 468 – 488, 1984. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0021999184901281

[5] H. G. Kümmel, "A biography of the coupled cluster method," *International Journal of Modern Physics B*, vol. 17, no. 28, pp. 5311–5325, 2003. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0217979203020442

[6] R. J. Bartlett and M. Musiał, "Coupled-cluster theory in quantum chemistry," *Rev. Mod. Phys.*, vol. 79, pp. 291–352, Feb 2007. [Online]. Available: http://link.aps.org/doi/10.1103/RevModPhys.79.291

[7] J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland, "Speeding up nek5000 with autotuning and specialization," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810120

[8] H. M. Tufo and P. F. Fischer, "Terascale spectral element algorithms and implementations," in *ACM/IEEE Conference on Supercomputing*, Portland, OR, 1999.

[9] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, "nek5000 Web page," 2008. [Online]. Available: http://nek5000.mcs.anl.gov

[10] M. O. Deville, P. F. Fischer, and E. H. Mund, *High-Order Methods for Incompressible Fluid Flow*. Cambridge, United Kingdom: Cambridge Press, 2002.

[11] 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," *J. Parallel Distrib. Comput.*, vol. 72, no. 3, pp. 338–352, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2011.09.006

[12] J. Shin, M. W. Hall, J. Chame, C. Chen, and P. D. Hovland, "Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology," in *In the Fourth International Workshop on Automatic Performance Tuning*, 2009.

[13] 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*, vol. 181, no. 9, pp. 1477 – 1489, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465510001438

[14] J. Hammond, "NWChem TCE CCSD(T) loop-driven kernels." [Online]. Available: https://github.com/jeffhammond/nwchem-tce-triples-kernels

[15] NVIDIA Corporation, "NVIDIA CUDA C programming guide," August 2014. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[16] OpenACC-Standard.org, "The OpenACC application programming interface." [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf

[17] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *IEEE International Symposium on Parallel Distributed Processing*, ser. IPDPS '09, May 2009, pp. 1–12.

[18] C.-K. Lu, "The Intel Software Autotuning Tool (ISAT) user manual (version 1.0.0)," Intel Corportion, Tech. Rep., 2010.

[19] P. Basu, M. Hall, M. Khan, S. Maindola, S. Muralidharan, S. Ramalingam, A. Rivera, M. Shantharam, and A. Venkat, "Towards making autotuning mainstream," *International Journal of High Performance Computing Applications*, vol. 27, no. 4, pp. 379–393, 2013. [Online]. Available: http://hpc.sagepub.com/content/27/4/379.abstract

[20] G. Rudy, "CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation," Master's thesis, Univeristy of Utah, Salt Lake City, UT, 2010.

[21] M. M. Z. M. Khan, "Autotuning, code generation and optimizing compiler technology for GPUs," Ph.D. dissertation, University of Southern California, 2012.

[22] 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.*, vol. 9, no. 4, pp. 31:1–31:25, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400690

[23] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," Tech. Rep., 2008.

[24] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *IEEE International Symposium on Parallel Distributed Processing*, ser. IPDPS '09, May 2009, pp. 1–11.

[25] B. Norris, A. Hartono, and W. Gropp, "Annotations for productivity and performance portability," in *Petascale Computing: Algorithms and Applications*, ser. Computational Science. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007, pp. 443–462, preprint ANL/MCS-P1392-0107. [Online]. Available: http://www.mcs.anl.gov/uploads/cels/papers/P1392.pdf

[26] NVIDIA Corporation, "Profiler user's guide," August 2014. [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide

[27] M. Griebl, C. Lengauer, and S. Wetzel, "Code generation in the polytope model," in *In IEEE PACT*. IEEE Computer Society Press, 1998, pp. 106–111.

[28] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, no. 3, pp. 563–590, Jul. 1967. [Online]. Available: http://doi.acm.org/10.1145/321406.321418

[29] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame, "A programming language interface to describe transformations and code generation," in *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 136–150. [Online]. Available: http://dl.acm.org/citation.cfm?id=1964536.1964546

[30] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965. [Online]. Available: http://comjnl.oxfordjournals.org/content/7/4/308.abstract

[31] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983. [Online]. Available: http://www.sciencemag.org/content/220/4598/671.abstract

[32] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 276–292, Feb 2005.

[33] 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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.3

[34] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *IEEE 27th International Symposium on Parallel Distributed Processing*, ser. IPDPS '13, May 2013, pp. 813–824.

[35] 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*, vol. 34, no. 26, pp. 2293–2309, 2013. [Online]. Available: http://dx.doi.org/10.1002/jcc.23377

[36] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek, "Automating the generation of composed linear algebra kernels," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 59:1–59:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654119

[37] J. Siek, I. Karlin, and E. Jessup, "Build to order linear algebra kernels," in *IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS '08, April 2008, pp. 1–8.

[38] L. Ha, J. King, Z. Fu, and R. M. Kirby, "A high-performance multi-element processing framework on GPUs," University of Utah, Tech. Rep., 2012.

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

[40] P. Schlatter and D. Henningson, "Large-scale fluid dynamics simulations—towards a virtual wind tunnel," 2014. [Online]. Available: http://www.cresta-project.eu/images/CaseStudies/cresta_casestudy2_2014.pdf