2014 Second IEEE Working Conference on Software Visualization

Search Space Pruning Constraints Visualization

Blake Haugen
Innovative Computing Laboratory
University of Tennessee Knoxville
bhaugen@utk.edu

Jakub Kurzak
Innovative Computing Laboratory
University of Tennessee Knoxville
kurzak@icl.utk.edu

***Abstract*—The field of software optimization, among others, is interested in finding an optimal solution in a large search space. These search spaces are often large, complex, non-linear and even non-continuous at times. The size of the search space makes a brute force solution intractable. As a result, one or more search space pruning constraints are often used to reduce the number of candidate configurations that must be evaluated in order to solve the optimization problem.**

If more than one pruning constraint is employed, it can be challenging to understand how the pruning constraints interact and overlap. This work presents a visualization technique based on a radial, space-filling technique that allows the user to gain a better understanding of how the pruning constraints remove candidates from the search space. The technique is then demonstrated using a search space pruning data set derived from the optimization of a matrix multiplication code for NVIDIA CUDA accelerators.

I. Introduction

Many optimization problems require the evaluation of a search space in order to determine optimal solution(s). This search space is often complex, multivariate, nonlinear, and often non-continuous. In software development, and many other fields, there may be a finite number of elements in the search space that can be evaluated. Even in cases where the search space is not finite it may be bounded by practical constraints. In these cases, the most obvious way to find the optimal solution to the problem is to perform a brute force optimization that evaluates each point in the search space and returns the optimal solution.

In the case of simple problems, the search space may be relatively small and evaluating every possible solution is not an unreasonable method for finding the optimal solution. For example, the simplest algorithm for multiplying two matrices is shown in Figure 1. Careful examination of the algorithm reveals that the three nested *for* loops could be reordered while maintaining the correct results. By reordering the loops it is possible to alter the memory access patterns and increase the performance of the matrix multiplication.

The i, j, and k loops can be reordered in six different ways to create a small search space where a brute force exhaustive evaluation of the search space is a reasonable solution. Performing six different matrix multiplications with six different loop orderings should provide enough information to suggest that one particular loop ordering provides the most efficient and best performing variant of the matrix multiplication kernel.

There are many other techniques that can be used to optimize the simple matrix multiplication algorithm shown in Figure 1. Another technique for optimizing the performance of a matrix multiplication code is loop tiling. The most basic implementation of loop tiling is shown in Figure 2. The tiling of the loops in the algorithm allows for the reuse of elements that are in the processor cache and ultimately accelerates the computation. This formulation of the algorithm only adds three *for* loops to the computation but dramatically increases the size of the optimization search space. The two variables in this optimization problem are the order of the loops and the size of the tiles used in the computation. While not all of the 720 possible orderings of the loop will produce the correct results, there are still several ways to reorder the loops and alter the performance. It is also possible to change the size of the blocks used in the loop.

```
for (i = 0; i < n; i++)
 for (j = 0; j < n; j++)
  for (k = 0; k < n; k++)
   C[i][j] += A[i][k] * B[k][j];
```

Fig. 1: $C = C + A * B$ matrix multiplication. A, B and C are $n \times n$ matrices.

```
for (k_ = 0; k_ < n; m_+=blk)
 for (j_ = 0; j_ < n; n_+=blk)
  for (i_ = 0; i_ < n; k_+=blk)
   for (k = k_; k < k_+blk; k++)
    for (j = j_; j < j_+blk; j++)
     for (i = i_; i < i_+blk; i++)
      C[i][j] += A[i][k] * B[k][j];
```

Fig. 2: $C = C + A * B$ matrix multiplication with loop tiling. A, B and C are $n \times n$ matrices. Tile size: blk

While the matrix multiplication with loop tiling shown in Figure 2 only works with block sizes that evenly divide the size of the matrix, it is possible to modify the algorithm to work with any arbitrary tile size. This means that the search space for the optimization problem is $n \times o$ where n is all of the tile sizes that are possible for matrices of size n (it is assumed that tile sizes that are larger than the size of the matrix are impractical) and o is the number of possible orderings for the loops. This search space may be tractable for small matrices but grows beyond the realm of brute force optimization rather

978-0-7695-5305-4/14 $31.00 © 2014 IEEE
DOI 10.1109/VISSOFT.2014.15

quickly.

It becomes apparent that the search space for the problem must be pruned in order to make the brute force optimization technique a practical solution. For the purposes of this paper, we will examine two different types of pruning constraints.

Hard Constraints
: A pruning constraint to eliminate cases where the resulting code variant will produce incorrect results or cause an error

Soft Constraints
: A pruning constraint that eliminates code variants that will be legal and give correct results but are unlikely to result in high performance

In the case of the matrix multiplication kernel with loop tiling, both hard and soft pruning constraints are applied to reduce the search space of the optimization problem. In the case of hard constraints, the six loops can be ordered 720 different ways. However, several of these orderings will produce incorrect results and can be pruned from the search space. The developer may also implement one or more soft pruning constraints based on the tile sizes used in the loop. For example, based on experience with the processor architecture, it is unlikely that odd tile sizes will not perform as well as even tile sizes or the tile size should be a multiple of the cache line size. These constraints can be violated while still producing correct results but are unlikely to yield optimal performance. Multiple pruning constraints can be implemented to reduce the optimization search space to a practical size.

It is simple to understand the effects of a single pruning constraint. The search space contains several code variants that either pass the pruning constraint or are eliminated. This becomes far more challenging when multiple constraints are used and the problem becomes a multi-dimensional problem with several interactions.

It can be challenging to fully analyze the pruning criteria because each of the code variants can be eliminated by one or more pruning constraints. If a single code passes all of the pruning constraints it will be evaluated for performance. A code variant, however, is eliminated if one or more of the pruning constraints is not met. It is simple to evaluate the effectiveness of a single pruning constraint at a time, but this becomes more challenging if the developer wants to evaluate multiple pruning constraints at a time. This work will present a methodology and a tool for evaluating this type of problem using a tree-based, radial, space-filling diagram.

II. Related Work

A. *Optimization and Search Visualization*

Androulakis and Vrahatis developed a package called OPTAC (Optimization Analysis and Comparisons) to visualize various optimization methods. [1] They aimed to visualize these methods in order to compare their properties. This work, however, does not appear to address the issue of search space pruning.

One of the areas of interest in pruning visualization is the problem of frequent itemset analysis. This type of analysis is used to understand what items tend to be grouped together. For example, it might be used to determine what types of items are purchased together at a grocery store. In this problem a very large search space is created from all of the possible itemsets and is frequently pruned and visualized. Examples of this work can be found in [2], [3], and [4].

Kuwata and Cohen develop a method for visualizing real-time search algorithms. [5] These search algorithms often perform heuristic pruning of the search space in order to optimize for performance. The visualization is used to better understand the search space and the performance of the heuristics used for pruning. This project seems to be closely related but seems to focus on the performance of the search rather than the specific pruning constraints.

B. *Trees*

The problem of analyzing the pruning constraints can also be viewed as a tree-based problem. The root of the tree represents the entirety of the search space. Each level down the tree represents another pruning constraint. One side of the tree will represent the code variants that pass while the other side represents the code variants that do not pass. The tree will be full if all possible combinations of passing and failure of the pruning constraints are present in the data set, but experiments thus far have shown this to be unlikely.

The visualization of trees has been heavily studied and the methods for visualizing these structures generally fall into two categories: the node-link diagram and the space-filling diagram.

1) Node-Link Diagram: The node-link diagram is often the quintessential visualization method for tree structures and data sets. In this tree representation, each node in the tree is generally represented by some sort visual element such as a circle, rectangle, or even just a text element. The nodes of the tree are then connected with a visual element (in most cases this is a simple line) to signify their relationship in the structure.

The node-link diagram is extremely effective at showing the hierarchy of the objects and their relationship to one another. This class of visualizations begins to break down when the goal is to understand the "weight" of each element of the tree. Typically each node in the tree is given an equal amount of screen space in the visualization. This gives the node-link tree diagrams a very uniform appearance but is less than ideal for visualizing the quantities or weights associated with specific nodes in a tree.

2) Space-Filling Diagram: One of the most common space-filling tree diagrams is the treemap. The treemap was originally designed as a method for visualizing the usage of a hard disk by various users. [6] The general idea is that each of the nodes in the tree will be represented by a rectangle. Each of the rectangles is scaled to show the user the relative size (or some other parameter) of that particular node. In the case of a file system, the size of the rectangle is often used to represent the size of a given file. The nodes are then grouped based on some common data characteristic and placed inside of a larger rectangle and all given the same color to signify that all of the nodes are part of the same class of elements.

The treemap is an excellent tool for visualizing many individual nodes that fit nicely into a simple two-level hierarchy, but they begin to break down when the goal is to visualize a deeper hierarchy. It is possible to create an interactive visualization that would allow the user to “zoom” in on a portion of the treemap and redraw a section of the visualization using a different characteristic for classification. This method, however, still only allows for the visualization of two levels of the hierarchy at a time.

Another method for visualizing the tree structure is the icicle plot which is similar to the idea of castles presented by Kleiner and Hartigan. [7] The icicle plot is an adjacency diagram where each level of the hierarchy is represented by one or more rectangles. As you move down the hierarchy the children of each node in the tree are represented by rectangles that are drawn proportionally to their weight.

The final visualization of the tree diagram, first presented by Yang, was called InterRing. [8] In this method, each successive ring represents another level of the hierarchy. Similar to the icicle diagram, the children are always shown one level below the parent nodes and “divide” their parent’s portion of the circle based on their weight. This methodology visualizes the weight of each node in the tree, like the treemap, while maintaining the clear hierarchy of the node-link diagram. This technique was expanded by Stasko [9] and compared extensively with the treemap diagram. [10] This technique was chosen as the basis for our visualization because of how efficiently the screen space is used to show the content. It gives the most screen space to the lower levels of the hierarchy where the nodes tend to be relatively small and fragmented.

III. Background

A. Automated Empirical Optimization of Software

The ATLAS (Automatically Tuned Linear Algebra Software) library [11] was one of the first to fully implement the automated software optimization approach. In numerical linear algebra, many of the applications are built on top of smaller building blocks often called the BLAS (Basic Linear Algebra Subroutines). If this set of routines is optimized for a given architecture, the applications built on top of them should also achieve high performance. The process of optimizing each of these routines by hand is a tedious and complex task that must be repeated for each particular hardware architecture. The ATLAS library aimed to automate this tuning process making the BLAS routines relatively portable across several computer architectures.

The methodology used in the ATLAS project was given the term AEOS (Automated Empirical Optimization of Software). [12] As the underlying architectures grew in complexity, so did the task of optimizing the software. The AEOS approach automated this process by generating a tremendous number of code variants using techniques like loop reordering, cache blocking, and many more. Each of the code variants was tested to determine the code variant that provides the highest performance.

B. Accelerators

High performance computing (HPC) has recently seen a tremendous growth in interest and adoption of Graphic Processing Units (GPUs) for general purpose processing (GPGPU). NVIDIA and AMD provide GPU products and Intel has released the Xeon Phi coprocessor. These hardware accelerators are an attractive option for many in the HPC community because they generally provide roughly an order of magnitude greater compute power and memory bandwidth than a standard processor. However, these architectures are quite different than the standard CPU. The standard CPU has grown incredibly complex with the addition of pipelines, out-of-order execution, branch prediction, as well as many other technologies. While these standard CPUs have started to place multiple cores on a single chip, they have not grown much past 12 to 16 cores for the most popular architectures at the time of this publication.

Hardware accelerators, however, reverse this trend of a small number of complex cores and replace them with a large number of relatively simple cores. In order to obtain high performance on an accelerator, the developer will need to employ the Single Instruction Multiple Threads (SIMT) paradigm or the Single Instruction Multiple Data (SIMD) paradigm. The OpenCL library provides the developers with a way of expressing code that complies with these computing paradigms for all accelerators. NVIDIA has developed CUDA for their own GPUs but they also support OpenCL.

Generally, codes developed for an accelerator employ a large number of threads in order to perform the computations. In contrast to a standard CPU thread, these threads are extremely lightweight and do not cost many resources to launch. In the CUDA programming model, these threads are organized into warps that contain 32 threads. These warps are then organized by thread blocks that can vary in size. Generally, the problem is mapped onto these threads and thread blocks in a fashion that divides the data among several threads that can operate all at the same time.

One of the simplest examples of how this paradigm works is a common example that adds two vectors shown in the CUDA kernel in Figure 3. The first line shows a vecAdd function that receives three pointers (a, b, and c) as well as the length of the vectors (n). (The __global__ at the beginning of the line is the syntax to tell the compiler that this is a CUDA kernel.) The first line inside the function computes the location of the current thread based on the block it is part of (blockIdx), the size of the block (blockDim), and the particular thread inside of the block (threadIdx). Once the global thread id has been computed, it computes the corresponding element of c after checking that this element is within the array. This kernel is then launched with a specified number of threads and thread blocks. The kernel is then executed by the threads (in parallel) on the device.

```
__global__ void vecAdd(double *a,
                       double *b,
                       double *c,
                       int n)
{
        \\Get global thread ID
        int id = blockIdx.x *
                       blockDim.x +
                       threadIdx.x;

        \\Do not go beyond array
        if (id < n)
                c[id] = a[id] + b[id];
}
```

Fig. 3: A CUDA kernel that adds vectors a and b to fill vector c. All vectors are length n.

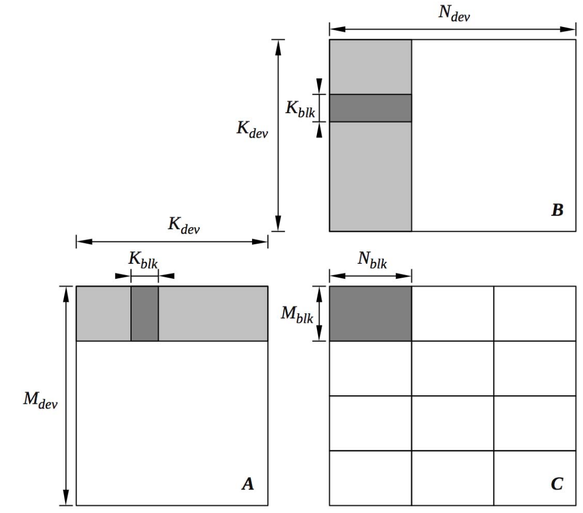

Fig. 4: GEMM at the device level.

C. Matrix Multiplication

In this paper, the case study will examine a data set derived from the optimization of a matrix multiplication (using double precision floating point numbers) written in CUDA for an NVIDIA GPU. The canonical version of the matrix multiplication was shown earlier in Figure 1. In this code, two matrices A and B are multiplied and added to a matrix C. [Note: matrix multiplication and GEMM are used interchangeably throughout this paper.]

In the case of a GPU, the C matrix is overlaid with a 2D grid of thread blocks and each one is responsible for computing a single tile of C. Since the code of a GPU kernel spells out the operation of a single thread block, the two outer loops disappear, and only one loop remains - the loop advancing along the *k* dimension, tile by tile.

Figure 4 shows the GPU implementation of matrix multiplication at the device level. Each thread block computes a tile of C (dark gray) by passing through a stripe of A and a stripe of B (light gray). The code iterates over A and B in chunks of K_{blk} (dark gray). The thread block follows the cycle of:

- making texture reads of the small, dark gray stripes of A and B and storing them in shared memory,
- synchronizing threads with the `__syncthreads()` call,
- loading A and B from shared memory to registers and computing the product,
- synchronizing threads with the `__syncthreads()` call.

After the light gray stripes of A and B are completely swept, the tile of C is read, updated, and stored back to device memory. Figure 5 shows more closely what happens in the inner loop. The light gray area shows the shape of the thread block. The dark gray regions show how a single thread iterates over the tile. The exact details of the code to implement this algorithm are not discussed here for the sake of brevity.

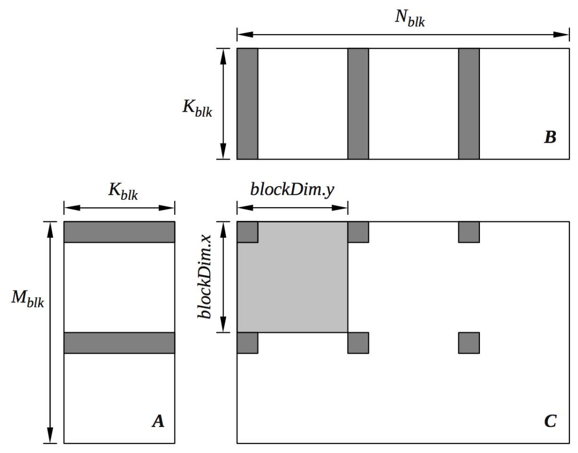

Fig. 5: GEMM at a block level.

D. BEAST

The methodology presented in this paper was developed as part of the BEAST (Bench-testing Environment for Automated Software Tuning) project that aims to employ the basic principles of AEOS in a developer-driven, iterative code tuning process for GPU applications. In this process, the developer first defines a specific kernel template and a set of constraints that must be maintained. The autotuning framework will then generate several code variants that are either eliminated or proceed to the benchmarking phase of the tuning process. Being an iterative process, however, the user can adjust the various pruning constraints after each iteration in order to better tune the code using the user's knowledge of the computation kernel as well as the feedback from the bench-testing framework. For example, the user may set one of the pruning constraints as too harsh and the tuning process may miss a well-performing code variant. This visualization can assist them in finding this error and adjust the pruning constraints accordingly. The visualization described here is one of the key feedback tools that allows the user to understand the effects of the pruning constraints so that he or she may adjust the parameters for each successive iteration of the tuning process.

E. The Tuning Process

Given a GPU matrix multiplication algorithm as described earlier, there are five dimensions that can be altered to generate variants of the matrix multiplication code. The first two are the dim_m and dim_n dimensions that describe the two dimensions of the thread block. The other three parameters are the blk_m, blk_n, and blk_k dimensions that describe the shape of the tiling sizes.

A sweep through all of the possible values for these five parameters can easily generate millions of GPU matrix multiplication code variants. In order to reduce the search space for this optimization problem, nine pruning constraints are used to eliminate illegal code variants as well as variants that are unlikely to perform highly. The nine pruning constraints are as follows:

Hard Constraints:

cant_reshape_block
The thread block can't be reshaped to cover the tile.

over_max_regs_per_block
This configuration will require more registers per block than is allowed by the architecture. Kernel will fail.

over_max_regs_per_thread
This configuration will require more registers per thread than is allowed by the architecture. Kernel will fail.

over_max_shmem_per_block
This configuration requires more shared memory per block than is allowed by the architecture. Kernel will fail.

over_max_threads
This configuration requires more threads than is allowed by the architecture. Kernel will fail.

Soft Constraints:

low_occupancy_regs
This configuration uses too many registers and is likely to limit the occupancy and decrease performance.

low_occupancy_shmem
This configuration uses too much shared memory and is likely to limit the occupancy and decrease performance.

partial_warps
Warps that are not full are not likely to provide optimal code performance.

short_on_fmas
This configuration defines a kernel where the load/fma ratio is too high and is not likely to produce high performance results.

F. Search Space Pruning Data

Each of the code variants generated by the tuning framework can be described by a number of primary and derived characteristics. The search space of millions of code variants can be pruned to reduce the number of candidate variants for final testing.

In the case of the matrix multiplication kernel for the NVIDIA GPUs, the framework generates 86,731,224 code variants. Table I describes the number of code variants that are eliminated by each of the pruning constraints. Each code variant is evaluated for each pruning constraint in order to determine whether or not the code variant should be removed from the search space. The number of variants eliminated in the table can be summed to a number that is far greater than the total number of code variants in the search space because a single code variant can fail more than one constraint. After the pruning is completed there are a total of 1,223 code variants that pass all nine of the pruning constraints. This is approximately 0.001% of the original search space.

The data presented in Table I is a "rolled-up" version of the data set that is used to generate the visualization. The original data set contains information about every possible combination of passing and failing of the pruning constraints. In other words, the number of elements in the data set is equal to 2^n where n is the number of pruning constraints that are implemented in the automatic tuning framework. These categories are generally represented by a bitmask of length n. For example, a search space that is pruned with four pruning constraints would have a total of 16 data points that correspond to all possible 4-bit bitmasks. Each bitmask uniquely represents a set of code variants in which each pass and fail identical pruning constraints. For example, the set represented by the bitmask 0110 would include all code variants that passed the first and last constraint but failed the second and third constraint.

The data must be stored in this more complete form in order to ensure that all of the necessary information is available for the visualization. The "rolled-up" form of the data set is excellent for providing the data for a visualization like the bar chart in Figure 6. However, it does not contain any information about the fashion in which the pruning constraints overlap that is necessary to generate the hierarchical structure that is used in the visualization.

IV. VISUALIZATION DESIGN

The most obvious way to visualize search space pruning constraints is with a simple bar chart. Figure 6 shows a bar chart of all nine pruning constraints in the case study data set. The user can easily get information about the pruning constraints. For example, the partial_warps constraint eliminates a very large number of the code variants while the over_max_threads constraint eliminates relatively few code variants.

Figure 6 can be useful but it leaves many questions unanswered. There is no way to determine how any of the pruning constraints overlap or interact.

Pruning Constraint	Variants Eliminated	Elimination Percentage
low_occupancy_shmem	73,871,811	85.17%
over_max_shmem_per_block	51,226,914	59.06%
low_occupancy_regs	66,154,080	76.27%
over_max_regs_per_block	33,889,968	39.07%
over_max_regs_per_thread	44,335,656	51.12%
short_on_fmas	27,677,304	31.91%
cant_reshape_block	75,204,971	86.71%
over_max_threads	19,432,248	22.41%
partial_warps	81,665,352	94.16%

TABLE I: This is a table summarizing the effectiveness of each of the pruning constraints. There were a total of 86,731,224 code variants in this particular data set. The table lists the number of code variants that were eliminated by each of the pruning constraints and the percentage of the search space that represents. There were a total of 1,223 code variants that passed all nine of the pruning constraints.

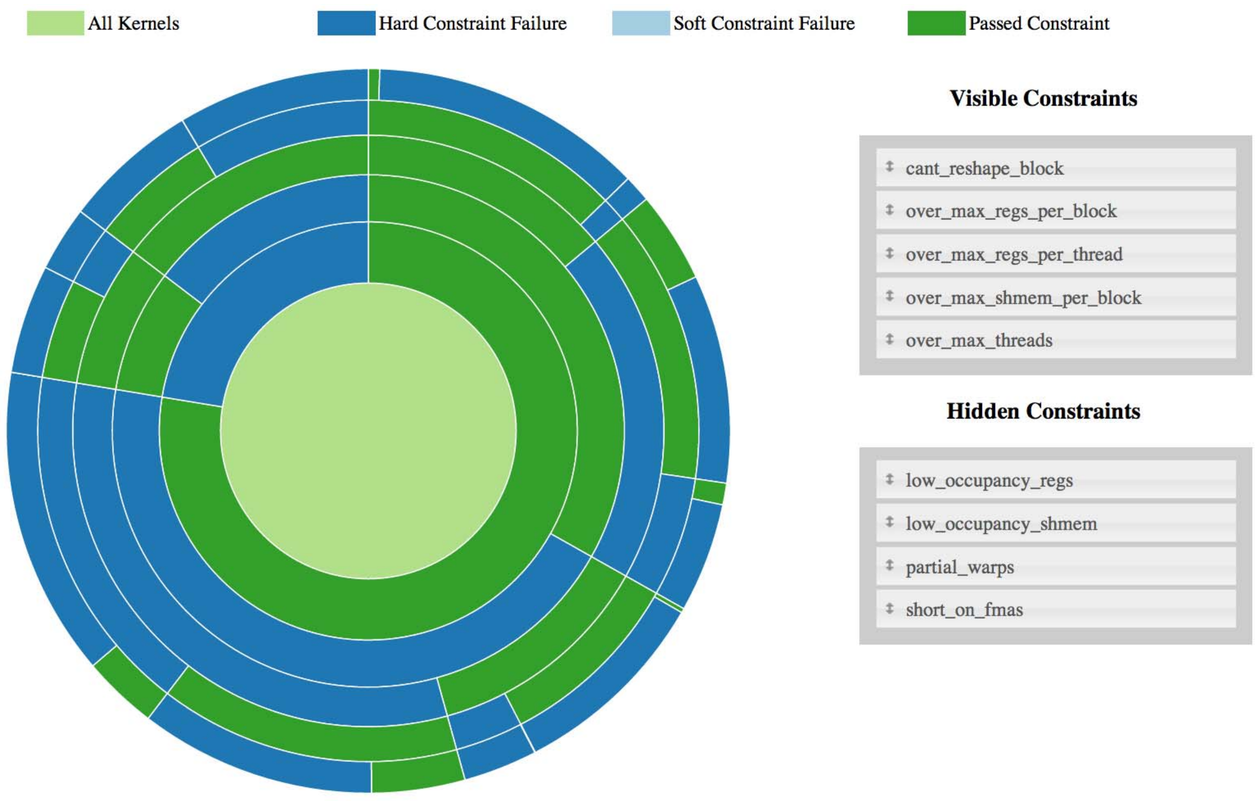

Fig. 7: A screenshot of the visualization showing the hard pruning constraints.

There are three parts of the visualization tool. First, the key defines the colors used in the visualization. The second part is the constraint selection lists, and the final part of the tool is the radial, space-filling visualization.

The key at the top of Figure 7 describes what each color in the visualization represents. There are a total of four colors used in the visualization. The first color is a light green that represents all of the kernels before any pruning has occurred. This color is only used in the center visualization. The darker green represents the portion of variants that pass each particular pruning constraint. There are two shades of blue that represent the portion of the code variants that fail the pruning constraint. The darker blue represents the code variants that failed a hard constraint and the light blue represents the code variants that fail a soft constraint. These two different shades of blue allow the user to visually differentiate the hard and soft pruning constraints.

On the right side of the screen there are two lists labeled “Visible Constraints” and “Hidden Constraints.” This is the primary way the user can interact with the visualization. Early versions of the visualization tool included all nine pruning constraints, but the large amount of information included in with all pruning constraints in a single diagram quickly overwhelmed the user. The two lists allow the user to select any

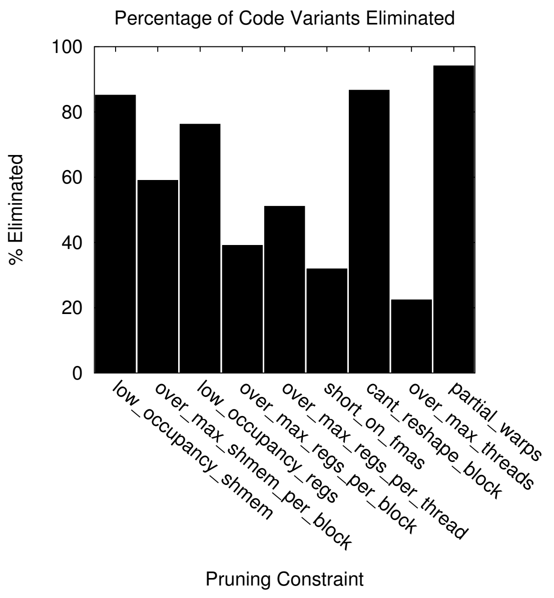

Fig. 6: A bar chart that shows the percentage of code variants that are eliminated by each individual pruning constraint

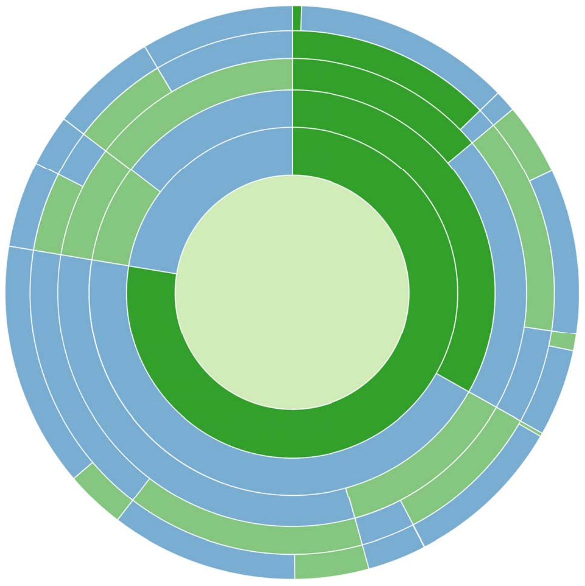

Fig. 8: Each constraint is ordered to show the passing constraints first in a clockwise fashion.

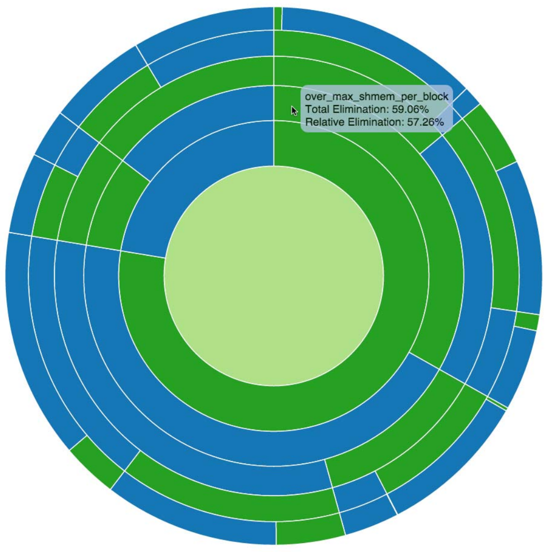

Fig. 9: A mouseover utility adds labels and information about that particular pruning constraint.

pruning constraint and simply drag-and-drop it to either hide the constraint or reorder it within the diagram. The elements in the visible constraint list also act as labels for the various "rings" in the diagram on the left. The elements in the list correspond to the "rings" starting from the outside and moving toward the center of the circle. For example, in Figure 7 the outermost section of the circle represents the portion of the code variants that pass or fail the "cant_reshape_block" constraint while the inner most "ring" (not including the light green center circle) represents the portion of code variants that pass or fail the "over_max_threads" constraint.

The final element of the tool is the radial, space-filling, or sunburst, diagram that depicts the hierarchy of pruning constraints. The visualization can be viewed as a hierarchical tree with each ring representing the next level of the tree. The root of the tree is placed at the center of the sunburst shown as the light green center of Figure 7. The first ring is divided into two sections where the green portion represents the percentage of the kernels that passed the over_max_threads pruning constraint, while the blue represents the percentage of the kernels that are eliminated by this pruning constraint. Each successive ring (moving out from the center) divides each of the categories based on the next pruning constraint in the list. This means that the first ring will have two sections, the second will have four sections, the third will have eight sections, etc. It should be noted that the data set may have several sections that do not have any corresponding code variants and will therefore show fewer segments in that particular ring.

Sections of the diagram are ordered to show the passing code variants before the eliminated code variants. The diagram starts as "12 o'clock" and proceeds in a clockwise fashion. This makes it easy to find the portion of the code variants that pass all of the pruning constraints because they will always start at "12 o'clock." The blocks representing the code variants that pass all pruning constraints are highlighted in Figure 8.

The visualization tool also includes a mouseover feature demonstrated in Figure 9. The tooltip includes the name of the pruning constraint corresponding to that particular ring of the diagram, as well as the total elimination percentage and the relative elimination percentage of that constraint. The total elimination percentage is the percentage of all code variants that are eliminated by that pruning constraint. One of the goals of this project is to understand how the various pruning constraints overlap in pruning the search space. With that in mind, the relative elimination percentage describes the percentage of code variants that would be eliminated but have not been eliminated by the previous constraints. It is important to note that the total elimination percentage will not vary based on the configuration of the visualization, but the relative elimination percentage will change based on the order of the constraints chosen in the visible constraint list. The tooltip in Figure 9 tells the user that the second ring from the center represents the over_max_shmem_per_block constraint and it eliminates 59.06% of the code variants and 57.28% of the code variants that remain after pruning for the first constraint.

A simple example of the visualization can be found in Figure 10 using the partial_warps and over_max_threads pruning constraints. The center of the diagram is the light green section that is labeled as section 1 in the diagram and it represents all of the code variants generated before pruning the search space. The first ring from the center represents the over_max_threads pruning constraint (labeled as constraint A) and it is divided based on the the percentage of code variants that pass (green section labeled as section 2) and fail (bright blue section labeled as section 3) the constraint. Moving outward, the outermost ring is divided into four sections labeled as sections 4, 5, 6, and 7 and represent the partial_warps pruning constraint (labeled as constraint B). Sections 4 and 5 represent the portion of code variants that have passed and failed pruning constraint B, respectively, and have already passed pruning constraint A. Similarly, sections 6 and 7 represent variants that have passed and failed pruning constraint B, respectively, but have not passed pruning constraint A. Also note sections 5 and 7 are light blue in color because the partial_warps (B) pruning constraint is a soft constraint. Section 3 is a bright blue because the over_max_threads (A) pruning constraint is a hard constraint.

This visualization greatly improves on the bar chart shown in Figure 6. The bar chart makes it easy for the user to quickly determine the portion of code variants that are eliminated by any particular pruning constraint. This task is also possible in the new diagram based on the percentage of the ring that passes or fails constraints. It is true that this estimation becomes more challenging for some constraints because of the fragmented nature of the outer rings, but this problem can be solved by a simple reordering of the pruning constraints in the visualization. However, the real improvement presented in this visualization is the ability to determine how the pruning constraints overlap and interact. For example, a user would be unable to determine the overlap of the two constraints in Figure 10 if they were presented in a bar chart. The visualization clearly shows that roughly half of the code variants that pass constraint B would have been eliminated by pruning constraint A.

A. *Implementation Details*

The visualization is implemented in javascript, making use of the d3.js [13] and jquery libraries. This platform was selected because the code is portable to several platforms and does not rely on many dependencies outside of the two javascript libraries.

The tool is also designed to be easily modified for use with different data sets. The only input to the visualization is a CSV file. There is one column for each of the pruning constraints as well as a column that indicates the number of code variants in each category. The header of each column is prepended by "hard:" or "soft:" in order to classify each pruning constraint. When the visualization is first opened, the "hard" constraints are part of the visible list and the "soft" constraints are part of the hidden list.

The table of data provided in the CSV is stored in a flat array of json objects that represent each row of the dataset. This format, however, is not very convenient for the hierarchical nature of the visualization. In order to make the data easier to use in the visualization generation, it is nested in a hierarchical json structure. It is not possible to compute this form of the data in advance because the user is able to reorder the hierarchy of pruning constraints at any time. It is necessary to recompute this data structure and the associated statistics in real time. Luckily, the d3.js library provides a powerful utility that generates the nested data structure based on a list of keys. The nested function is also used to calculate the statistics shown in the mouseover.

V. Case Study Insights

The goal of this visualization is to gain a greater understanding of the nature of pruning constraints in a search space. If a developer can quickly and intuitively gain insight into the process of search space pruning, it is possible to more efficiently create, modify, and deploy search space pruning constraints and ultimately find an optimal solution faster.

In the following examples we will examine some of the insights that were obtained through this visualization technique. The data was created in the process of optimizing a matrix multiplication algorithm for an NVIDIA GPU. The algorithm and details of the data set were described earlier in this paper.

One of the goals of this technique was to give software developers a way to understand how the pruning constraints in the optimization problem overlap. It is possible that they frequently occur together or never occur together. Figure 11 examines the pruning performed by the short_on_fmas and over_max_threads constraints. You can see that there are very few code variants that pass the short_on_fmas constraint (the inner ring) and do not pass over_max_thread constraints. In fact, after applying the first constraint, the second only eliminates another 9%.

When pruning the search space, it is also useful to find redundant pruning constraints that may be eliminated in the future. Figure 12 shows pruning constraints for low_occupancy_regs, over_max_regs_per_thread,

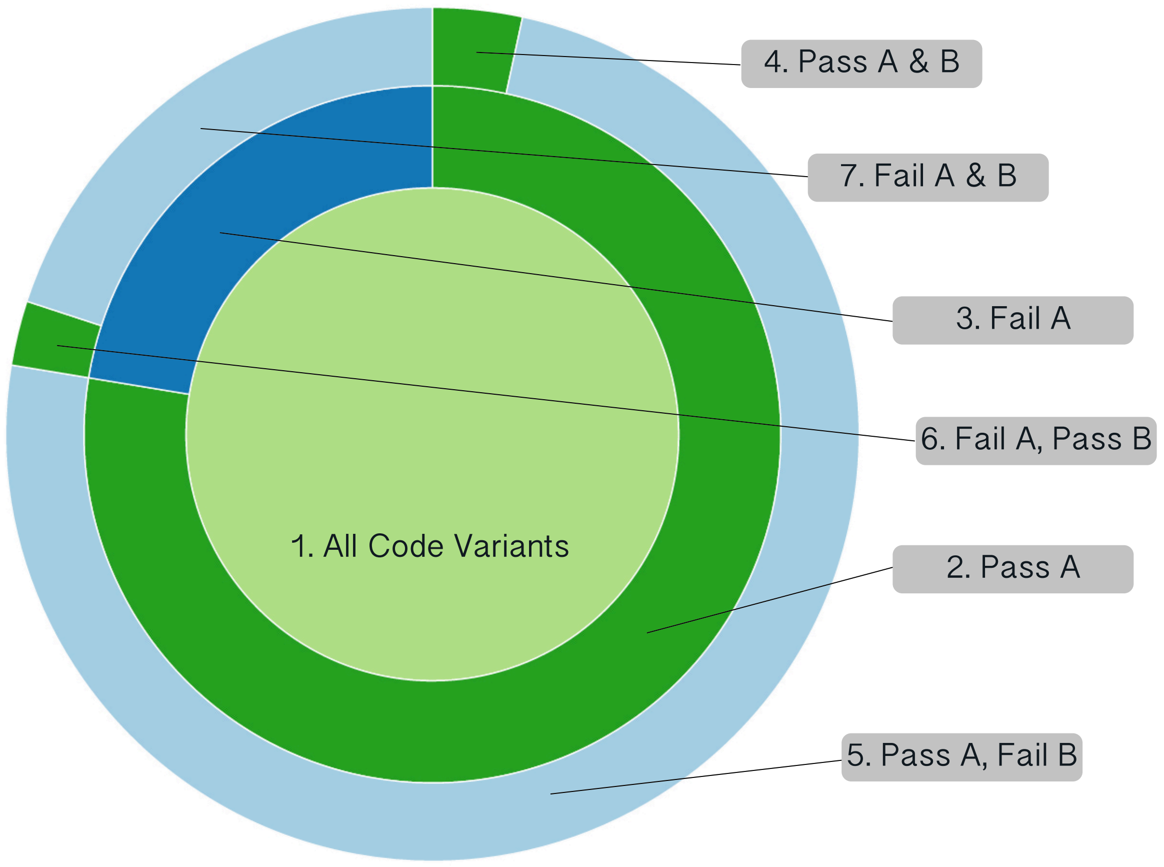

Fig. 10: The outer most ring represents the partial_warps constraint (A) and the inner ring represents the over_max_threads constraint (B).

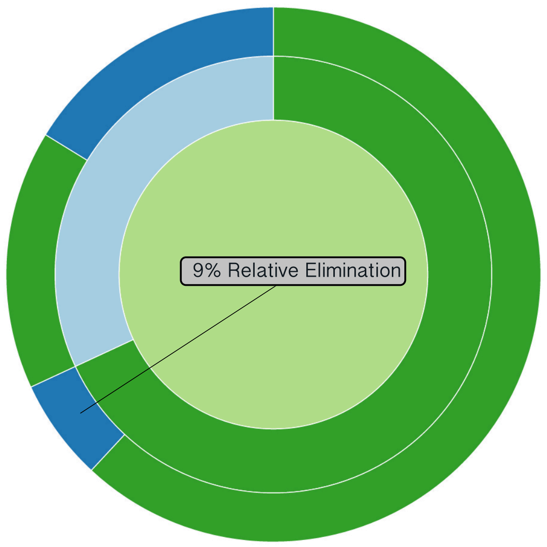

Fig. 11: Inner Ring: short_on_fmas, Outer Ring: over_max_threads

and over_max_regs_per_block (labeled from inside to outside). We can see that after the search space has been pruned using the low_occupancy_regs constraint there are no code variants eliminated by the other two constraints that have not already been eliminated by the first constraint. A similar redundancy occurs with the low_occupancy_shmem and over_max_shmem_per_block constraints.

Figure 13 shows the low_occupancy_shmem and low_occupancy_regs pruning constraints. It appears that there is a rather large number of code variants that pass low_occupancy_regs but fail the low_occupancy_shmem, which suggests that the code might be limited by the shared memory available in the system. This may be of interest to hardware architects in determining what factors are important for code performance when designing the next generation of hardware.

The search space pruning constraint visualization tool presented in this paper made this type of pruning constraint analysis much easier understand. It is possible to get all of the insights above based on calculations of the data set but this process has been greatly simplified by this interactive visualization.

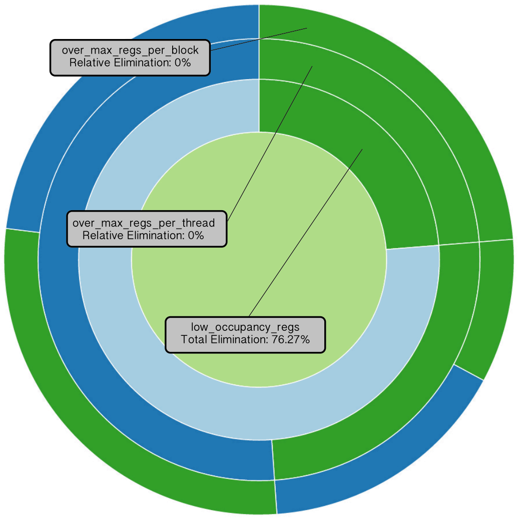

Fig. 12: Inner Ring: low_occupancy_regs, Middle Ring: over_max_regs_per_thread, Outer Ring: over_max_regs_per_block

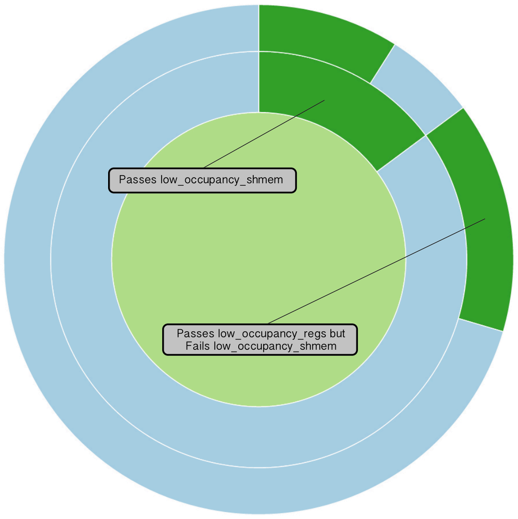

Fig. 13: Inner Ring: low_occupancy_shmem, Outer Ring: low_occupancy_regs

VI. Conclusions

Optimization problems occur in a number of domains, including software development. The space that must be searched is often very large and pruning constraints are an attractive option to accelerate the optimization process. The analysis of these pruning constraints can be a challenging multivariate data analysis problem. The ability to reformulate the raw data set as a tree and interact with a radial, space-filling visualization has given users new insights into the process of search space pruning.

Acknowledgment

This work was funded by NVIDIA as well as the National Science Foundation under grant NSF-CCF-1320603.

References

[1] G. Androulakis and M. Vrahatis, "Optac: a portable software package for analyzing and comparing optimization methods by visualization," *Journal of Computational and Applied Mathematics*, vol. 72, no. 1, pp. 41 – 62, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0377042795002448

[2] L. Yang, "Pruning and visualizing generalized association rules in parallel coordinates," *IEEE Trans. on Knowl. and Data Eng.*, vol. 17, no. 1, pp. 60–70, Jan. 2005. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2005.14

[3] G. Bothorel, M. Serrurier, and C. Hurter, "Visualization of frequent itemsets with nested circular layout and bundling algorithm," in *Advances in Visual Computing*, ser. Lecture Notes in Computer Science, G. Bebis, R. Boyle, B. Parvin, D. Koracin, B. Li, F. Porikli, V. Zordan, J. Klosowski, S. Coquillart, X. Luo, M. Chen, and D. Gotz, Eds. Springer Berlin Heidelberg, 2013, vol. 8034, pp. 396–405. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41939-3_38

[4] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo, "Finding interesting rules from large sets of discovered association rules," in *Proceedings of the Third International Conference on Information and Knowledge Management*, ser. CIKM '94. New York, NY, USA: ACM, 1994, pp. 401–407. [Online]. Available: http://doi.acm.org/10.1145/191246.191314

[5] Y. Kuwata and P. R. Cohen, "Visualization tools for real-time search algorithms," *Computer Science Technical Report*, pp. 93–57, 1993.

[6] B. Shneiderman, "Tree visualization with tree-maps: 2-d space-filling approach," *ACM Trans. Graph.*, vol. 11, no. 1, pp. 92–99, Jan. 1992. [Online]. Available: http://doi.acm.org/10.1145/102377.115768

[7] B. Kleiner and J. A. Hartigan, "Representing points in many dimensions by trees and castles," *Journal of the American Statistical Association*, vol. 76, no. 374, pp. pp. 260–269, 1981. [Online]. Available: http://www.jstor.org/stable/2287820

[8] J. Yang, M. O. Ward, and E. A. Rundensteiner, "Interring: An interactive tool for visually navigating and manipulating hierarchical structures," in *Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, ser. INFOVIS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 77–. [Online]. Available: http://dl.acm.org/citation.cfm?id=857191.857749

[9] J. Stasko and E. Zhang, "Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations," in *Proceedings of the IEEE Symposium on Information Vizualization 2000*, ser. INFOVIS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 57–. [Online]. Available: http://dl.acm.org/citation.cfm?id=857190.857683

[10] J. Stasko, "An evaluation of space-filling information visualizations for depicting hierarchical structures," *Int. J. Hum.-Comput. Stud.*, vol. 53, no. 5, pp. 663–694, Nov. 2000. [Online]. Available: http://dx.doi.org/10.1006/ijhc.2000.0420

[11] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," in *SuperComputing 1998: High Performance Networking and Computing*, 1998, cD-ROM Proceedings. **Winner, best paper in the systems category.**
URL: `http://www.cs.utsa.edu/˜whaley/papers/atlas_sc98.ps` .

[12] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the atlas project," *PARALLEL COMPUTING*, vol. 27, p. 2001, 2000.

[13] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-driven documents," *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011. [Online]. Available: http://vis.stanford.edu/papers/d3