An Alternative Algorithm for Vectorization

Michael D. Adams
Dept. of Computer Science
Indiana University

Larisse Voufo
Dept. of Computer Science
Indiana University

1st December 2006

1 Introduction

Vectorization is an important optimization for many scientific computing applications. A traditional solution is to use CodeGen from [2]. However, while that algorithm is well known variations on the theme are possible. In this paper we explore an alternative to CodeGen that works in a bottom-up fashion in stead of the top-down design that CodeGen usually uses. Using a bottom-up design ends up leading to insights into the design of CodeGen and also allows optimizations over the standard that would be be possible in the top-down algorithm.

2 Basic Theory

2.1 Dependence graphs

The task of vectorization at its core requires the identification and preservation of dependencies. These dependencies may be true- (read-after-write), anti- (write-after-read) or output- (write-after-write) dependencies. However all these cases may be handled by the use of a dependence graph where the nodes represent statements and the edges represent dependencies between the statements. For example in the following code there is a dependence from S2 to S1.

```
for i = 1:N,
    C(i) = B(i)   % S1
    B(i+1) = A(i) % S2
end
```

When there are no cycles, these graphs may be trivially linearized to an order that respects the dependencies via topological sort, and those statements vectorized. With the previous example, this would result in:

```
B((1:N)+1) = A((1:N)) % S2
C((1:N)) = B((1:N))   % S1
```

Note that the topological sort has reversed the order of the statements to preserve the dependency.

2.2 Cycles

Many programs however have one or more cycles in their dependency graphs. The statements that make up such a cycle may not be vectorized, but they do not prevent the statements that are not in those cycles from being vectorized. To handle this situation two stages are necessary. The first to identify strongly connected components (i.e. cycles) and collapse them to a single node. The second to topologically sort these collapsed nodes. Each resulting node that was due to a cycle must be re-wrapped in a loop equivalent to the one that they were originally in. Nodes that were not in a cycle may be vectorized.

For example in the following code, S1 depends on S2, S2 depends on S1, and S3 depends on S2 and S3, and finally S4 depends on S3. Thus S1 and S2 form a cycle, and S3 also forms a cycle by itself. After these cycles have been collapsed, there are 3 nodes including the non-cycle node for S4. By topological sorting, they must appear in the order {S1, S2}[1], {S3}, S4.

```
for i = 1:N,
    A(i) = A(i) + B(i) % S1
    A(i) = 2*A(i)      % S2
    B(i) = B(i) + A(i) % S3
    D(i) = 2*B(i)      % S4
end
```

The vectorized form of the code according to the above algorithm is then,

```
for i = 1:N,
    A(i) = A(i) + B(i) % S1
    A(i) = 2*A(i)      % S2
end
for i = 1:N,
    B(i) = B(i) + A(i) % S3
end
D((1:N)) = 2*B((1:N))  % S4
```

Note in particular if even those S3 was alone, it was still in a cycle and thus not vectorized. Also note that the order of the statements in the {S1, S2} loop must be preserved from the original loop in order to remain correct.

[1]The brackets indicate that the label names within them make a strongly connected component a.k.a cycle.

2.3 Loop-Nests

When there are multi-level loop-nests things get a bit more interesting because there are multiple loops which may be causing the dependencies. Consider for example the following program.

```
for i = 1:N,
    B(i) = sqrt(B(i))                          % S1
    for j = 1:M,
       for k = 1:P,

          A(i,j,k) = A(i,j-1,k) / B(i)         % S2
    end
end
```

There is a dependency carried by the "j" loop which prevents vectorization in "j", but the "k" loop should be able to be vectorized just fine as it does not carry any dependencies. The CodeGen algorithm handles this situation by first starting at the top level and proceeding with a complete dependency graph. In this example the graph would include edges indicating that S2 depends on S1 and S2 depends on S2. Thus S1 is not in a cycle but S2 is, so S1 may be vectorized but not S2. Then the CodeGen algorithm drills down into the cycles that could not be vectorized and considers only the dependencies that are interior to the cycle while also not considering dependencies caused by the previous loop level being considered. In our example the dependencies in S2 would be considered but ignoring any dependencies caused by the "i" loop. Since the dependency is carried by the "j" loop there is still a cycle. The CodeGen algorithm will drill down again, this time ignoring the dependencies carried by the "i" and "j" loops. At this point S2 will have no dependency cycles caused by the remaining "k" loop so S2 may be vectorized but only in "k". The resulting vectorized code will be,

```
B((1:N)) = sqrt(B((1:N)))                    % S1
for i = 1:N,
    for j = 1:M,

       A(i,j,(1:P)) = A(i,j-1,(1:P)) / B(i) % S2
```

3 Our Solution

Our solution is quite similar to the CodeGen solution in that we use a dependency graph that is checked for cycles and then topologically sorted. However there are a number of critical differences. The dependency graphs that the user must annotate the code with are limited, the graph cycle calculations are done locally and the final vectorization is done in a bottom-up manner as opposed to the top-down CodeGen algorithm.

3.1 Graph Annotations

Though our system requires the user to annotate all loops with dependency information, the dependency information is propagated up from inner loops to outer loops since a dependency on any inner loop will also apply to any outer loop. This means that the user doesn't have to include redundant dependency information in the outer loops about dependencies between statements in inner loops. This information is automatically pushed up the loop-hierarchy by an initial pass of the vectorizer and actually allows for an extra optimization over what is possible with CodeGen which will be discussed later.

The annotations take the form of either statement labels or dependency information. Statement labels are comments that precede a statement and are of the form "%#lab [label_name]"[2]. Dependency information must be placed before any loop that might be vectorized and takes the form "%#dep [label_names...] : [label1],[label2] [label3],[label4]...". Each token is separated by whitespace including the colon. The names before the colon are label names that appear inside the loop. The names after the colon are paired up by a joining comma with no extra space around the comma and represent dependencies. For example label2 depends on label1 and label4 depends on label3. The previous example might be annotated as follows.

```
%#dep S1 S2 : S1,S2
for i = 1:N,
    %#lab S1
    B(i) = sqrt(B(i))                   % S1
    %#dep S2 : S2,S2
    for j = 1:M,
        for k = 1:P,
            %#lab S2
            A(i,j,k) = A(i,j-1,k) / B(i) % S2
    end
end
```

3.2 Graph Cycle Calculation

Once the dependency information has been pushed up the loop-hierarchy, all of the information necessary to locate the dependency cycles and perform the topological sorting for a particular loop is located in a single dependency statement at the top of that loop. This makes it easy to implement a primitive C++ parser for these dependency statements. Once these dependency statements have been imported into C++, the Boost Graph Library [1] may be used which has excellent algorithms for both strongly connected components and topological sorting. Finally this pass outputs information indicating what statements may be vectorized and in what order the statements must be executed.

[2] The brackets in this case indicate optional or meta-syntax

3.3 Doing CodeGen Bottom-Up

Once the dependency information has been calculated and each loop annotated with what statements may be vectorized and what statements are in cycles, the next pass proceeds to actually perform those vectorizations. However unlike the top-down CodeGen algorithm, this pass proceeds in a bottom-up fashion. This provides a number of advantages for making the implementation easier. For example, since all inner loops statements have already been visited and possibly vectorized before the outer statements are visited, their labels may be lifted. This way, the labels referenced by the outer loop dependencies that reference statements nested a few loops down do not require a deep search; since their labels have already been bubbled up through the code.

4 Equivalence of top-down and bottom-up

Our bottom-up algorithm has many similarities and parallels to the normal top-down CodeGen. In fact it should be able to perform exactly the same vectorizations as the regular CodeGen. Where the regular CodeGen keeps reducing the dependence graph as it traverses down the nesting levels, our bottom-up algorithm pushes the dependencies up the nesting levels and achieves the same effect. Cycles and topological sorting serve the same purpose in both algorithms. The final major difference is that one starts with the outer most loops and moves inwards while the other starts with the inner most loops and moves outwards. The bottom-up algorithm does require that a statement be able to be vectorized one index at a time while the top-down CodeGen vectorizes all the indices of a statement at once. However, given that provision, both algorithms are actually calculating the same dependence graphs, cycles and topological sorting. Finally and most importantly they are performing the same vectorizations. The only difference is the order in which these operations are being performed: bottom-up or top-down.

5 Differences between top-down and bottom-up

Thought the top-down algorithm and the bottom-up algorithm at some level are the same, there are important differences in the motivations for the different aspects of the algorithms and some unique opportunities for improvements. For example, as the bottom-up algorithm proceeds along, any "for" loops it leaves behind must have their entire body in a cycle. In the top-down algorithm this is also true once the algorithm completes, but for the bottom-up algorithm this is true at every intermediate stage. This actually provides a motivation for the phase that does the pushing-up of dependencies. Once a cycle has been detected and a "for" loop constructed, the statements that form that "for" loop must be kept together and cannot be separated. Thus no outer loop should be allowed to separate the statements in the "for" loop. Pushing up the dependencies from the inner loop ensures that any cycles that occur in an inner loop will be seen

by the outer loops and the algorithm will not try to separate those statements when it gets to the outer loops.

The pushing-up of dependencies is of course equivalent to the sub-setting of the dependence graph that the top-down algorithm uses. However looking at the motivation for this stage from the perspective of ensuring that statements inside of a "for" loop don't get separated suggests a possible optimization. If the "for" loop contains a single statement and that statement is thus the single member of a cycle then it is impossible for the statement to be separated from itself and pushing up the dependence may be avoided. This would then mean that outer loops may vectorize that statement even though the inner loop could not vectorize that statement. Readers are encouraged to walk though the cases where this situation may occur in order to satisfy for themselves that this optimization is always valid when a single statement is involved in a dependency cycle. In our implementation this optimization is implemented by only pushing up those dependency edges that start and end at different nodes.

By using this optimization our running example may vectorize S2 in both the "i" and "k" loops thus resulting in the following code.

```
B((1:N)) = sqrt(B((1:N)))
for j = 1:M,

    A((1:N),j,(1:P)) = A((1:N),j-1,(1:P)) ./ B((1:N))
end
```

This optimization would be much less trivial or intuitive in the top-down algorithm. Many other intuitive insights might be gained from viewing the standard CodeGen algorithm from the bottom-up.

6 Limits

Though not an inherent limitation of the bottom-up algorithm, our implementation has not pursued certain features with regard to the mechanics of vectorizing a single statement. Instead we focused on developing the bottom-up algorithm and ensuring that it selects when and what to vectorize correctly. Once it has been decided that a particular statement can be vectorized with regard to a particular index, our implementation uses only a straight forward substitution of the index's range for the index itself. Because MatLab has a highly overloaded syntax this may not always result in equivalent code. Consider this simple example.

```
for i = 1:N,
    A(i) = A(i) * A(i)
```

Obviously this loop should be able to be vectorized. Simply substituting the "(1:N)" for "i" however will not work because the resulting code would be:

```
A((1:N)) = A((1:N)) * A((1:N))
```

In which case the "*" operator is now multiplying two row vectors and will be interpreted as a dot-product instead of an element wise product. In this particular case, the user may avoid this trouble by using ".*" instead of "*" from the start and explicitly request an element-wise product. Another possibility is the following loop.

```
for i = 1:N,
    A(i,2) = B(3,i)
```

In this case a row matrix will be assigned to a column matrix. One could apply a transpose with:

```
for i = 1:N,
    A(i,2) = B(3,i)'
```

However this is not always possible and there are numerous more subtle traps as the following loops show.

```
for i = 1:N,
    for j = 1:M,

        A(i,j) = B(j,i)   % Transpose problem (solvable)
for i = 1:N,
    for j = 1:M,
        for k = 1:P,
            A(i,j*P+k) = B(j,k) + C(k,i)
            % Index arithmetic, and  (unsolvable) transpose problem
for i = 1:N,
    for j = 1:M,

        A(i+j) = 3    % Results in addition of vectors (1:N) and (1:M)
```

Some workarounds may exist by using the arrayfun or cellfun functions or perhaps creative use of repmat and reshape, but since no general solution has presented itself we have chosen to leave it the user's job to ensure that the statements in vector form have the same semantics as they do in non-vector form. Adding more smarts to the code that worked only sometimes would only make life harder on the user.

7 Benchmarks

Experiments where carried out on Intel(R) Xeon(TM) CPU 2.80GHz processors with a cache size of 512 KB and a clock speed of 2.8GHz. The machine used in question (helga.cs.indiana.edu) possesses two physical and two virtual processors; the available main memory was 6.0 GB and the available virtual memory was 8.2 GB.

In order to verify the efficiency of our vectorizing pass, we experimented on two types of cubic-time problems: Cholesky factorization, with vectorization possible with respect to only one loop, and Matrix Multiplication, with vectorization possible with respect to two nested loops. For an even better analysis, a series of tests were run under Matlab version 7.0.1.24704 (R14) Service Pack 1, and then under Octave version 2.1.69 (i686-pc-linux-gnu). The Matlab tests were run with "-nodisplay" and "-nojvm" and the Octave test were run with no special options. The results appeared to be very favorable under Octave, and only restricted ranges of matrix orders showed promising results on Matlab. This yields to indications of hidden super-optimizations being performed by Matlab during compile and run time. Moreover, those super-optimizations, unfortunately, do not favor codes in vectorized form, as much as they do for non-vectorized codes. Figures 1 illustrates the timings observed from all categories of experiments.

8 Conclusion

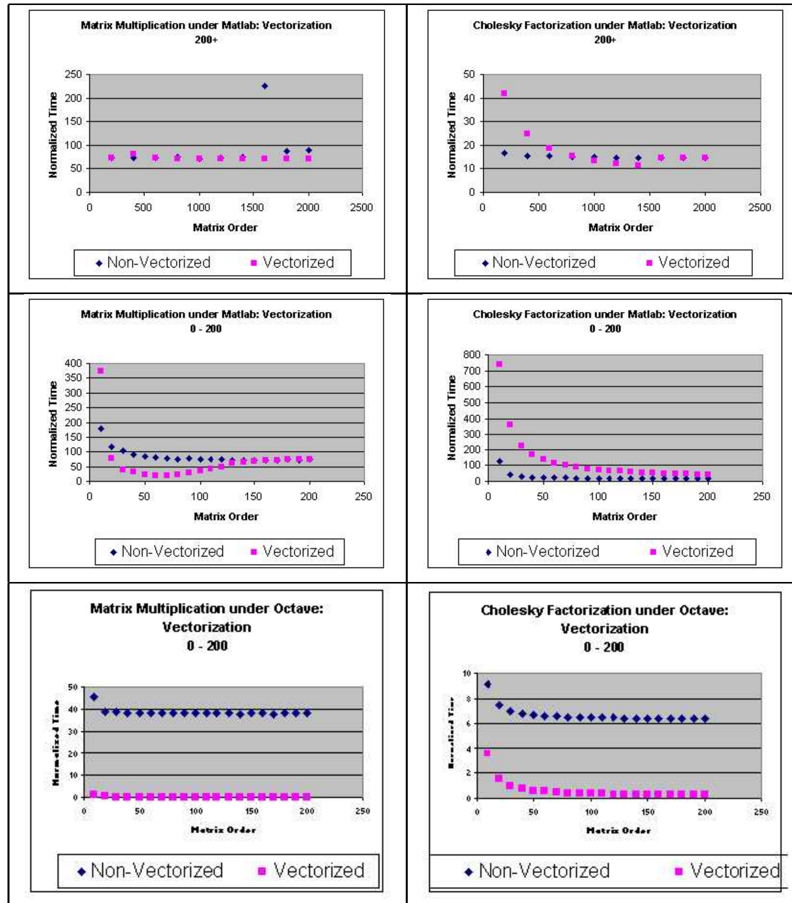
The bottom-up approach to vectorization provides unique insights to a traditionally top-down problem. It provides alternative motivations for equivalent aspects of the algorithm design. In the process it provides insight into opportunities for optimizations beyond the standard top-down CodeGen. As well, in some cases it can make the user's job easier given that the user only needs to specify local dependency graphs. It also allows an easier implementation by providing extra guarantees about the code contained in the loop currently being processed (e.g. any loop inside of it contains only statements that make up a single strongly connected component). Finally, because we had to design the algorithm from scratch and could not blindly implement an existing algorithm, it provided a very effective vehicle for learning and truly understanding the process of vectorization.

References

[1] *Boost Graph Library, The: User Guide and Reference Manual.* Addison Wesley Professional, 2002.

[2] Allen, R., and Kennedy, K. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann Publishers, 2002. Chapter 2.

Figure 1: Matlab



A Appendix: Code Guide

The best way to understand the code is to understand the pipeline of stages of transformation that the code goes through. The top level program is bin/vectorize. It is a filter taking a Matlab program on stdin and producing a vectorized form on stdout. It is actually just a short script that pipes together several other programs and its contents should look something like the following.

```
bin/make_graph |
bin/parse |
Vectorize/Vectorize |
Vectorize/StripPragmas |
bin/unparse-canonical
```

The "bin/make_graph" program is also a simple script that runs a pipeline and looks something like:

```
bin/pre-parse |
Vectorize/PrePushUp |
bin/pre-unparse |
Vectorize/fix-dep.pl |
calc_graph/calc_graph
```

In order of actual execution the purpose of each of these program may be summarized by the following table.

Pipeline Program	Purpose
bin/preparse	Parses Matlab code with dependency annotations into ATerms
Vectorize/PrePushUp	Performs the PushUp Phase
bin/pre-unparse	Pretty prints Matlab code with dependency annotations from ATerms
Vectorize/fix-dep.pl	Fixes a few stray white-spaces generated by Stratego's pretty-printer in bin/pre-unparse
calc_graph/calc_graph	C++ code that uses the Boost Graph Library to calculate strongly connected components and do topological sorting. Reads Matlab code with dependency annotations and produces Matlab code with Cycle and Vector annotations.
bin/parse	Parses Matlab code with Cycle and Vector annotations into ATerms
Vectorize/Vectorize	Stratego code to do the bottom-up vectorization.
Vectorize/StripPragmas	Stratego code to remove annotations that bin/unparse-canonical doesn't understand.
bin/unparse-canonical	Pretty prints vectorized matlab code.

B Appendix: Code Used for Experiments

Figure 2: Code for Matrix Multiply Benchmark

```
%#dep a :
for i = 1:rows1,
  %#dep a :
  for j = 1:cols2,
    %#dep a : a,a
    for k = 1:cols1,
      %#lab a
      M(i,j) = M(i,j) + M1(i,k) * M2(k,j);
    end
  end
end
```

Figure 3: Code for Vectorized Matrix Multiply Benchmark

```
for k = 1:cols1,
  M((1 : rows1), (1 : cols2)) = ...
    M((1 : rows1), (1 : cols2)) + ...
      (M1((1 : rows1), k) * M2(k, (1 : cols2)));
end
```

Figure 4: Code for Cholesky Benchmark

```
%#dep sqrt div mult : sqrt,div div,mult mult,sqrt
for k = 1:order,
  %#lab sqrt
  M(k,k) = sqrt( M(k,k) );
  %#dep div :
  for i = (k+1):order,
    %#lab div
    M(i,k) = M(i,k) ./ M(k,k);
  end
  %#dep mult : mult,mult
  for j = (k+1):order,
    %#dep mult :
    for i = j:order,
      %#lab mult
      M(i,j) = M(i,j) - M(i,k) .* M(j,k);
    end
  end
end
```

Figure 5: Code for Vectorized Cholesky Benchmark

```
for k = 1:order
  M(k, k) = sqrt(M(k, k));
  M((k + 1 : order), k) = M((k + 1 : order), k) ./ M(k, k);
  for j = k + 1:order
    M((j : order), j) = M((j : order), j) - (M((j : order), k) .* M(j, k));
  end
end
```