Loop Vectorization in Java programs using Partial Summaries

Chathur Bommineni cs20b018
Advised by Prof. V. Krishna Nandivada

May 19, 2024

1 Abstract

Modern CPU architectures provide various SIMD instructions. Certain programs can be vectorized(replacing scalar instructions with vector instructions where appropriate) to obtain an improvement in performance. In the case of Java, this may be done by JIT compiler or the programmer. But existing techniques are either not effective or not very practical.

In this project, we propose a new technique for automatic vectorization in Java programs. During initial compilation of the Java program to bytecode, we perform some analysis to determine if there is scope for vectorization. This information is stored into result files, which may be used at run time to replace appropriate scalar instructions with vector instructions.

2 Introduction

2.1 Motivation

Modern CPUs support some SIMD(single instruction multiple data) instructions which can operate simultaneously on multiple inputs. This can be used to improve run-time performance. Such instructions generally take a vector of inputs and generate a vector of outputs. Here, vector means a array of data contiguous in memory.

As we shall see, this is very useful in programs containing loops which perform the same operation on an array of input data. Consider a SIMD instruction whose size of input vector is 4. If in a program there is a loop performing the same operation on an array of input data, lets say of size n, then if appropriately changed to use the vector instruction instead of the scalar instruction, the number of CPU cycles would be down from n to $\frac{n}{4}$ for the "vectorized" binary.

Though the benefits of vectorization are evident, applying the optimization is not very straightforward in the real world. In the next section, we shall see two existing methods of solving the problem which we looked at.

2.2 Existing methods

2.2.1 Vectorization during JIT compilation

For the purposes of this project, we worked with the Eclipse OpenJ9 JVM [2]. In general, a JVM intially runs the program by interpreting the Java bytecode. As the program runs, the JVM assigns "hotness", a measure of how often a section of bytecode is executed. When a certain hotness is reached for a section of code, that section is compiled down to binary instructions. As the code gets hotter, some optimizations may be applied during the generation of the binary.

In OpenJ9 one of the optimizations applied is vectorization. At a certain hotness level, the loops in the function being optimized are analyzed to see if they can be automatically vectorized. This involves checking for a few conditions and then replacing the scalar instructions with vector instructions.

In practice, this was not very effective in our limited testing. We tried running a simple program with a vectorizable loop and enforced maximum optimization, but the vectorization failed due to one of the checks failing. When we reached out to the developers of OpenJ9, they informed us that this optimization was introduced in an experimental state a long time ago and has not been maintined since. It does limited analysis to determine if vectorization is possible and actually performs the optimization in only very few cases.

2.2.2 Vectorization by the programmer

A vector API has been introduced in a JEP by openJDK. The idea is to provide an " ... API to express vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures ... ". The API has been an incubated since JDK 16 and will be fully integrated later.

The API provides a platform agnostic way for the programmer to write code with explicit vectorization. At runtime, it is determined if the current platform supports vector instructions. If so the appropriate instructions are used to generate the binary. Otherwise, the binary generation falls back to using scalar instructions to generate the traditional implementation.

The advantage of this method is that generated code will run on any platform, while exploiting vectorization wherever possible without platform specific intervention by the programmer. But the drawback is that the onus is on the programmer to identify where vectorization is possible and write accurate code themselves. And also for existing codebases to leverage this method will require manual indentification of vectorization opportunities and code refactoring.

2.3 Proposal

As we can see, although techniques exist for applying vectorization in Java, each has their own drawbacks. Here we would like to propose a new technique to address some of these shortcomings. Here is a high level overview of the phases involved:

- During compilation from Java to Java bytecode, we perform analyses to determine which statements in which loops in the input program may be vectorized. In some cases the vectorization may be performed only for certain values of some local variables. There could also be some calls to library functions whose code is not availble at this stage. All of this information, which we call a partial summary (partial because we may not have the source code of certain library calls) is generated and stored into a results file.
- At runtime, the JVM is provided the results files corresonding to the currently executing program. When time comes for certain seciton of code to be compiled down to machine instructions, the JVM will consult the appropriate results files and determine where vectorization is possible by combining partial summaries to get a complete analysis. After determining if the current platform supports the required vector instructions, the compiler generates the vector instructions. If at any point, it is determined that vectorization is not possible, the compiler generates the original scalar implementation.

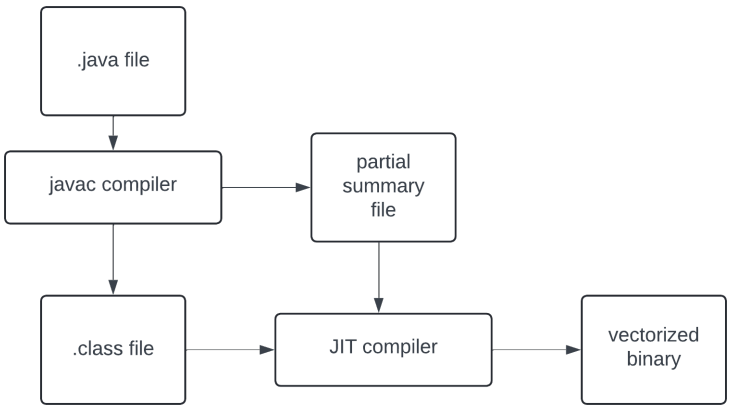

Figure 1: Flowchart

We try to obtain the following advantages over the aforementioned existing techniques:

- There is no intervention required by the programmer. The optimization is applied automatically wherever applicable.
- The generated partial summaries are architecture agnostic. The platform specifics are only considered at runtime.
- The entire analysis is not perfromed every single run. All the information that can be obtained during bytecode generation is encoded in the partial summaries and the only operation performed at run time is the combining of partial summaries to obtain the full analysis.

2.4 Contributions

- We describe a new technique for automatic vectorization using partial summaries in Java programs which solves some of the problems in the existing techniques for achieving the same.
- A description about what a partial summary file might contain and a sample grammar for the same.
- A description of how the technique may be implemented and a basic implementation in the Soot optimization framework.

3 Background

3.1 Vectorization

In the context of loops, vectorization is an optimization that attempts to run as many of the loop iterations as possible at the same time on a SIMD system. A loop using scalar instructions can be converted to use vector instructions as illustrated below.

```
1       // scalar instructions
2       for (i = 0; i < 1024; i++)
3               c[i] = a[i] * b[i];
4
5       // vector instructions
6       for (i = 0; i < 1024; i += 4)
7           c[i:i+3] = a[i:i+3] * b[i:i+3];
```

We can see that the vectorized binary will take 4 times fewer CPU cycles than the scalar binary.

3.2 Loop Distribution

Loop distribution (or loop fission) is an optimization in which a loop is broken into multiple loops over the same index range with each taking only a part of the original loop's body. This is illustrated in the following example.

```
        // original
        for (i = 0; i < 1024; i++){
                a[i] = 1;
                b[i] = 2;
        }

        // distributed
        for (i = 0; i < 1024; i += 4){
                a[i] = 1;
        }
        for (i = 0; i < 1024; i += 4){
                b[i] = 2;
        }
```

In general, loop distribution can improve locality of memory accesses. But in the context of our project, we use loop distribution in order to enable vectorization. In the above example, the original code couldn't be vectorized, but the distributed code can be.

Note that loop distribution is not always sound. One must perform proper dependece analysis to determine if distributing a given loop is sound or not.

3.3 Dependence Analysis

Dependence analysis is used to find out the data dependece between statements in a program. A "dependece" is a relation that constrians the execution order of the instructions. Let two statements is a program be S_1 and S_2.

If S_1 precedes S_2 in execution order, we write $S_1 \triangleleft S_2$. A control dependence arises from control flow of program. If S_2 depends on S_1, we write $S_1 \delta^c S_2$.

A data dependence arises from the flow of data between statements.

- If $S_1 \triangleleft S_2$, S_1 sets a value which S_2 uses, it is flow/true dependence. $S_1 \delta^f S_2$.
- If $S_1 \triangleleft S_2$, S_1 uses a value which S_2 sets, it is anti-dependence. $S_1 \delta^a S_2$.
- If $S_1 \triangleleft S_2$, both set a variable, it is output dependence. $S_1 \delta^o S_2$.
- If $S_1 \triangleleft S_2$, both read a variable, it is input dependence. $S_1 \delta^i S_2$.

In the iteration space of nested loops, the k-tuples of values of the loop indexes are called the index vectors. We use $\prec$ to denote the lexical ordering of index vectors, so

$$\langle i1_1, ..., ik_1 \rangle \prec \langle i1_2, ..., ik_2 \rangle$$

$$iff \quad \exists j, 1 \leq j \leq k, i1_1 = i1_2, ..., i(j-1)_1 = i(j-1)_2, ij_1 \leq ij_2$$

We use bracketed subscripts after a statement number to indicate the values of the index variables, for example $S_1[i1_1, ..., 1k_1]$. So $S_1[i1_1, ..., 1k_1] \triangleleft S_2[i1_2, ..., 1k_2]$ iff either S_1 precedes S_2 in the program and $\langle i1_1, ..., ik_1 \rangle \preceq \langle i1_2, ..., ik_2 \rangle$ or S_1 is the same statement as or follows S_2 and $\langle i1_1, ..., ik_1 \rangle \prec \langle i1_2, ..., ik_2 \rangle$.

4 Partial summaries

Now let us look at what information can be stored in a partial summary. We came up with a layout for the results file containing the partial summaries, which we will describe below.

- We will generate one partial summary per method, containing all the information pertaining to it. First, we will have a list of all the loops in that method. Each loop will be referred to by it's BCI (bytecode index).
- Next for each loop, we will have a list of array accesses in the loop that may be vectorized. We will basically look for patterns in the bytecode to determine this. So each vectorizable access is identified by its BCI. We also store the pattern of access so that at run time the JIT compiler will know which vector instruction to use. The examples and the technique that we use (described later) will make this more clear.
- Next for each of the vectorizable accesses, there is a condition under which the statement may by distributed out of the loop. The condition is in generally in terms the local variables in the method. The idea is that during bytecode generation, dependece analysis is performed to determine if the statement can be distributed. Sometimes it is possible to determine if it is distributable at the time of bytecode generation, in which case the partial summary will only contain a "true" or a "false". Otherwise if it is dependent on the local variables, the condition under which it is distributable is stored in the partial summary so that it may be evaluated at runtime.
- Finally, sometimes there might be some extra conditions which are required for vectorization to be performed. So we may specify the condition in the partial summary.

4.1 Partial summary grammar

```
1        PartialSummary  :=  MethodName (LoopSummary)* <EOF>
2        LoopSummary     :=  BytecodeIndex (Vectorizable)*
3        Vectorizable    :=  BytecodeIndex "," TypeOfAccess "," Condition ","
         ↪  Condition
4        TypeOfAccess    :=  "ArrayAddConstant"
5                          | "ArraySubConstant"
6                          | "ArrayMulConstant"
7                          | "ArrayAssignConstant"
8                          | "ArrayAddArray"
9                          | "ArraySubArray"
10                         | "ArrayMulArray"
11                         | "ArrayAssignArray"
12       MethodName      :=  String
13       BytecodeIndex   :=  Integer
14       Condition       :=  BooleanExpr
```

5 Partial analysis

In the project, we developed a basic partial analysis for generating the partial summaries. We will describe it here. We make a few assumptions about the loops to make the analysis easier

- The loops are in canonical form i.e. it is of the form

```
1               for( i : 0 ... n ){
2                       // body
3               }
```

- The only way to exit the loop is for the loop induction variable reaching its maximum allowed value. This means that the loop cannot contain break or continue statements etc.

This analysis is performed using the Soot framework [5]. We added our analysis to the "wjtp" pack of transformations which uses the Jimple IR. In order to get the list of loops in the method we use a toolkit provided by soot out of the box as follows

```
1       LoopFinder loopFinder = new LoopFinder();
2       Set<Loop> loops = loopFinder.getLoops(body);
```

5.1 Identifying vectorizable accesses

Next we traverse trhough the set of loops. For each loop we then try to find statements which can be vectorized. But this may not trivial because the Jimple IR is 3-addressed, meaning a single statement may become multiple statements in Jimple. Consider the follwoing snippet of an array to array assignment inside a loop.

```
1       // original
2       A[x] = B[y+1];
3
4       // jimple transformed
5       temp0 = A;
6       temp1 = x;
7       temp2 = B;
8       temp3 = y;
9       temp4 = temp3 + 1;
10      temp5 = temp2[temp4];       // look for
11      temp0[temp1] = temp5;       // this pattern
```

Here in the original Java code, we can plainly see that the array to array assigment can be vectorized, but it is not clear from the jimple transformed code. To solve this, we try to find the pattern in line 10-11 while traversing the body of the loop. We can similarly find patterns for constant assignments to arrays, constant(or array) addition(or subtraction or multiplication) to array etc and identify them in the partial summary.

5.2 Backward slicing

The Jimple IR snippet also demonstrates another issue we faced. The expressions in the code are seperated into multiple temporaries. In order to get back the expression, we need to do a backward program slice. Program slicing is the computation of the set of program statements that may affect the values at some point of interest. We use a Soot inbuild way for obtaining the definition of a local variable at any program point.

```
1        LocalDefs ld = new SimpleLocalDefs(new CompleteUnitGraph(body));
2        List<Unit> defs = ld.getDefsOfAt((Local)v, (Unit)s);
```

The obtained definition itself might contain temporaries, so we need to do it recursively until we get back the original expression. This way we can obtain all the expressions corresonding to indices of array accesses. We store all of these expressions for future use.

5.3 Distributability

Now that we have the all the vectorizable accesses, we would like to know under what conditions these loops can be distributed. Consider the following pseudocode and let us see how we analyze it.

```
1        // before distribution
2        // e1,e2 are functions of induction variable i
3        for(i : 0 to n){
4                stmt1 (access A[e1])
5                stmt2 (access A[e2])
6        }
7
8        // after distribution
9        for(i : 0 to n)
10               stmt1
11
12       for(i : 0 to n)
13               stmt2
```

There may be other statements between, before or after these two statements but it wont affect the analysis. In order for the distribution to be sound

- both the accesses should only be reads. In this case the data never changes so even if the order of accesses changes, it won't make a difference.
- if one of the accesses is a write, then for an iteration of $stmt2$ we need to make sure that the element accessed by it in that iteration is not accessed by $stmt1$ in a future iteration (because the resulting change would not be visible in the original code but would be in the distributed code), i.e.

$$\forall i_1 > i_2, e_1(i_1) \neq e_2(i_2)$$

This can be extended to more than two accesses of an array. When analyzing a statement for distribution, we need to do the above check for every other statement accessing the same array in the loop (all pairs of accesses). We would like to find the range of induction variable i for which the above condition holds.

One more thing we need to consider is how to analyze distributability when there is a method call in the loop which takes an array as an input which is also in a statement which is a candidate for vectorization elsewhere. Especially, if the method is a library call whose bytecode is not readily availble for analysis. We tried to solve this by adding some more information about the method in its partial summary, but were unable to come up with a simple and practical solution.

5.4 Z3 solver

In order to do the above analysis, we use the Z3 SMT solver [1] (for using the Java bindings, we use a jar distributed by z3-turnkey [6]). We use the following technique to find a range for i. Due to the assumption of canonical loops, we already have the conditions $i_1 >= 0$ and $i_2 >= 0$. We also add the condition $i_1 > i_2$. Next we also add the condition $e_1(i_1) = e_2(i_2)$ but crucially, we add an objective to the solver to $minimize(i_1)$. We use the *Optimize* solver which allows us to specify the objective.

```
1         // using z3
2         o = Optimize();
3
4         o.add(i1 > 0);
5         o.add(i2 > 0);
6         o.add(i1 > i2);
7         o.add(e1 == e2);
8
9         o.minimize(i1);
10
11        if( o.check() != Status.SATISFIABLE ){
12                // always distributable
13        }
14        else{
15                m = o.getModel();
16                i_min = m.eval(i1);
17                // distributable if i < i_min always
18        }
```

- If the model is not satisfiable, then the loop is always distributable. In the partial summary, we can give the condition for distributability as simply *true* since it always possible.
- If the model is satisfiable, then the solver gives us the smallest value m of i_1 for which it is. So if $i < m$ always, then the loop will be distributable. If the bound of the induction variable in the loop is n, in the partial summary, we can give the condition for distributability as $n <= m$.

6 Runtime

At runtime, the JVM has access to the bytecode and the partial summaries. Whenever a method is marked for JIT compilation, during the optimization phase, the compiler can look at the condition under which loop distribution and/or vectorization is allowed. The compiler can then generate two binary instruction sequences; one with vector instructions and one with scalar instructions.

In the generated code, the compiler can instrument code such that if the condition holds true then the vectorized code is reached, otherwise the scalar code is reached (as shown below). Unfortunately we did not have time in this project to create an actual implementation of this, so this may be taken up as future work.

```
        // binary after JIT compilation

        if cond then goto L1        // cond taken from the partial summary

        /* scalar implementation */

        jump L2
        L1

        /* vector implementation */

        L2
```

7 Evaluation

We tested our implementation for generating partial summaries for some basic test cases. Here we provide a few examples.

7.1 Example 1

```
        // input program
        for(int i=0; i<n; i++){
                A[i + 2] += temp1;
                A[i] *= 1;
                A[i + 1] = 5;
        }
```

Here there are three vectorizable accesses. For the first, the Z3 solver tells us that the index expression being equal to the other two is unsatisfiable, meaning it will never happen. So in the partial summary, the condition is set to *true*. For the others, their respective conditions are provided.

```
main :
loop 0 :
        9, ArrayAddConstant, true, true
        14, ArrayMulConstant, n <= 1, true
        19, ArrayAssignConstant, n <= 1, true
```

7.2 Example 2

```
// input program
for(int i=0; i<n; i++){
        B[i] = A[3*i + 1];
        A[2*i + 9] *= 2;
}
```

Here there are two vectorizable accesses. The array A is accessed in these statements, so we check of their index expressions can match. The Z3 solver returns the minimum solution as 4. Since the limit of the induction variable is n, we set the condition for distributability in the partial summary as $n <= 4$.

```
// generated partial summary
main :
loop 0 :
        10, ArrayAssignArray, n <= 4, true
        18, ArrayMulConstant, n <= 4, true
```

At runtime, we may expect JIT compiler to use this summary to generate a binary as follows (pseudocode). Assume that the vector instructions take inputs of size 4.

```
// pseudocode of generated binary
if n <= 4 then jump L1
for i : 0 to n
        Set B[i] A[3*i + 1]
        Mul A[2*i + 9] A[2*i + 9] 2
Jump L2
L1
for i : 0 to n/4
        Mul i1 i 4
        VectorSet B[i1] A[3*i1 + 1]
for i : 0 to n/4
        Mul i1 i 4
        VectorMul A[2*i1 + 9] A[2*i1 + 9] 2
L2
```

7.3 Example 3

```
// input program
for(int i=0; i<8; i++){
        B[i] = A[4*i];
        A[2*i + 6] -= 1;
}
```

In this example, the Z3 solver returns the minimum solution as 2. But the loop limit is already availble at the time of bytecode generation as 8. Since the limit is greater than the minimum solution, in the partial summary, the condition is set to $false$.

```
// generated partial summary
main :
loop 0 :
        9, ArrayAssignArray, false, true
        17, ArraySubConstant, false, true
```

8 Related work

- The OpenJ9 JVM has auto-vectorization as one of the optimizations applied by the JIT compiler. It was introduced in an experimental state and hasn't been maintined since. As mentioned, in practice, this wasn't very effective.
- The Vector JEP aims to add an API to allow the programmer to write explicitly vectorized code. At runtime if the platform supports vector instructions then they are used, otherwise it falls back to scalar implementation. The API has been incubated in JDK 16 and is waiting to be integrated. The main problem with this approach is that the programmer needs to identify vectorization opportunities themselves.
- "PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs" [4] presents a framework for points to analysis and escape analysis in which the authors use a technique where expensive partial analysis is performed during static compilation and the partial analyses are combined at run time to obtain the full analysis. The ideas presented in this paper are inspired by this, though we apply them for loop vectorization.

9 Future work

- In this project we were only able to get as far as generating the partial summaries during bytecode generation. The obvious next step would be to implement the runtime component in the JIT compiler. It would read

the partial summaries, determine where vectorization is possible and then use vector instructions and provide a performance gain.

- We must think about when in the order of other optimizations should the partial summaries ge generated. One of the assumptions we made in our basic technique for partial summary generation is that the loops are in canonical form. Loop canonicalization is often used optimization that converts loops which are not in canonical form into canonical form. So it might be a good idea to make sure that loop canonicalization happens before partial summary generation so that more loops may be analyzed.
- Considerations regarding how method calls inside a loop, which take an array as a parameter which is being considered for vectorization, change the dependence analysis and how this information may be encapsulated in partial summaries can be looked. We were not able to find a simple and practical solution for the same in the scope of this project.

References

[1] Leonardo De Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, 337–340. ISBN: 3540787992.

[2] *Eclipse OpenJ9*. URL: `https://projects.eclipse.org/projects/technology.openj9`.

[3] Steven S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[4] Manas Thakur and V. Krishna Nandivada. "PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs". In: *ACM Trans. Program. Lang. Syst.* 41.3 (2019). ISSN: 0164-0925. DOI: 10.1145/3337794. URL: https://doi.org/10.1145/3337794.

[5] Raja Vallée-Rai et al. "Soot - a Java bytecode optimization framework". In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, p. 13.

[6] *z3-turnkey*. URL: `https://github.com/tudo-aqua/z3-turnkey`.