Vectorization for Java

Jiutao Nie, Buqi Cheng, Shisheng Li, Ligang Wang, Xiao-Feng Li

Vectorization for Java

Jiutao Nie, Buqi Cheng, Shisheng Li, Ligang Wang, Xiao-Feng Li

China Runtime Technologies Lab, Intel China Research Center
{jiu-tao.nie, bu.qi.cheng, shisheng.li, ligang.wang, xiao.feng.li}@intel.com

Abstract. Java is one of the most popular programming languages in today's software development, but the adoption of Java in some areas like high performance computing, gaming, and media processing is not as universal as in general-purpose computing. A major drawback preventing it from being extensively adopted in those areas is its lower performance than the traditional or domain-specific languages. This paper describes two approaches to improve Java's usability in those areas by introducing vector processing capability to Java. The first approach is to provide a Java vectorization interface (JVI) that developers can program with, to explicitly expose the programs' data parallelism. The other approach is to use automatic vectorization to generate vector instructions for Java programs. It does not require programmers to modify the original source code. We evaluate the two vectorization approaches with SPECjvm2008 benchmark. The performances of scimark.fft and scimark.lu are improved up to 55% and 107% respectively when running in single thread. We also investigate some factors that impact the vectorization effects, including the memory bus bandwidth and the superscalar micro-architecture.

Keywords: Java, vectorization, dependence graph, memory bus bandwidth, superscalar micro-architecture

1 Introduction

Java as a programming language has modern features for software productivity, security and portability. Java also has a comprehensive standard library covering almost all kinds of application needs. It is one of the most widely used programming languages from mobile phone through server.

Due to its position in modern computation environment, a great deal of work has been done to improve Java performance. With the introduction of just-in-time (JIT) compilation, the performance of Java programs has been improved significantly. More and more optimization techniques in static compilers have been adopted by the JIT compilers of Java. In many situations, the execution speed of Java programs can be comparable to the equivalent C programs. However, in terms of performance, there is still one major absence in Java world compared to C/C++ world, i.e., the support to leverage the powerful vector processing units of modern microprocessors. A C/C++ programmer can benefit from the vector instructions in various approaches, such as inserting inline assembly, calling vector intrinsics, writing programs with vector API, etc. However, none of those approaches is available to Java programmers.

To bring the benefit of vector computation capability into Java world, we develop two complementary approaches: a library-based programming approach and a compilation-based automatic approach. In this paper, we describe the two approaches and discuss the issues we meet with Java vectorization. The main contributions of this paper include:

1. We define a generic set of Java vectorization interface (JVI) with Java class library and implement JVI support in a Java virtual machine. JVI covers the vector operation spectrum of common IA32 and EM64T microprocessors.
2. We implement the automatic vectorization in a Java virtual machine that tries to vectorize Java applications automatically.
3. We evaluate the two vectorization approaches with SPECjvm2008 benchmark, and the performance of scimark.fft and scimark.lu is doubled (up to 55% and 107% respectively) when running in single thread.
4. We investigate the factors that impact the vectorization effect. Our study shows that memory bus bandwidth can be a limiting factor for vectorization to scale up on multicore platforms. Superscalar micro-architecture can also hide the benefit of vectorization.

The rest of the paper is organized as follows. Section 2 discusses related work in program vectorization. Section 3 and Section 4 are the main body of this paper describing our Java vectorization work based on JVI and on automatic vectorization respectively. Section 5 gives the experimental results and discusses the issues. We conclude our work in Section 6.

2 Related Work

The latest published vectorization work for Java is an SLP [2] automatic vectorizer implemented in Jikes RVM [3]. It uses a modified tree-pattern matching algorithm to identify similar successive instructions and turn them into equivalent vector instructions. It relies on loop-unrolling to vectorize computations of different iterations as other pure SLP algorithms do. The implementation is in the initial stage. It does not generate real single instruction multiple data (SIMD) instructions. Instead, it uses 32-bit integer instructions to simulate the simple vector computations of 16-bit short and 8-bit byte types.

In recent years, lots of work on automatic vectorization has been devoted into the GCC compiler [4, 5]. Java users can benefit from this work by compiling Java programs into native code with Gnu Compiler for Java (GCJ), which uses GCC's middle and back end to compile both Java source code and Java bytecode into native code. The vectorizer implemented in GCC supports simple loop-based automatic vectorization [6] and interleaved memory accesses in loop [7, 8]. A limitation of the current GCC's vectorizer is that the memory accessing strides must be constants whose values are powers of 2. Our automatic vectorization algorithm does not have this limitation. It unrolls the part of the loop that cannot be vectorized. If the unrolled instructions access consecutive addresses, a complementing SLP vectorizer can further group them into vector instructions. A later important improvement on GCC's vectorizer is to integrate SLP vectorization into the previous loop-based vectorizer, which results in a loop-aware SLP vectorizer [9]. The improved one is more flexible to handle various memory accessing patterns.

The Intel C++ compiler (ICC) provides four levels of vectorization supports: inline assembly, intrinsics, C++ vector library and automatic vectorization. The C++ vector library provides vector classes with overloaded operators. Operations on those types are translated into vector IR by the compiler's front end. Our Java vector class library does the same thing except that no overloaded operators are provided since Java does not support operator overloading.

Novell implements a vector class library in its .Net framework, called Mono.Simd, to provide the vector API supports for C# programming. The APIs are mapped directly to the hardware vector operations. Due to the directly mapping, Mono.Simd might be tightly coupled with certain hardware versions.

3 Java Vectorization Interface (JVI)

Both the JVI based vectorization and the automatic vectorization need the support of a just-in-time (JIT) compiler. We implement the two vectorization approaches in Jitrino, the optimizing JIT compiler of Apache Harmony. Jitrino has two levels of intermediate representation (IR), HIR and LIR. It supports most modern facilities for optimization, such as the static single assignment (SSA) form, the control flow graph and the loop tree. It also contains many commonly used compiler optimizations. The infrastructure of Jitrino with our vectorization work is shown in Figure 1. The flow on the right side illustrates the process from the Java source code to the native machine code. Boxes represent transformations, and ellipses represent programs in different forms. The JIT compiler, i.e. Jitrino is encircled by the dotted box, in which, the dashed boxes B, C and F are transformations modified for vectorization, and the other two dashed boxes D and E are modules added for vectorization and depended on by various transformations. The upper-left dashed box A is the JVI class library that can be used by programmers in their Java source code. In the figure, A and B are for JVI based vectorization, and C is for automatic vectorization. D, E and F provide support for both vectorization approaches. This section introduces the design and implementation of the JVI based vectorization.

3.1 JVI design

JVI is designed to be an abstract vector interface independent to concrete vector instruction sets. Programs written with it can be compiled to use different vector instructions, such as SSEx and AVX of Intel processors. JVI comprises a set of classes representing vector types of various primitive Java types, including 8, 16, 32 and 64 bits signed integers, as well as 32-bit and 64-bit floating point numbers. They are packaged into the name space `com.intel.jvi`. Currently, the default size of a vector is 128-bit.

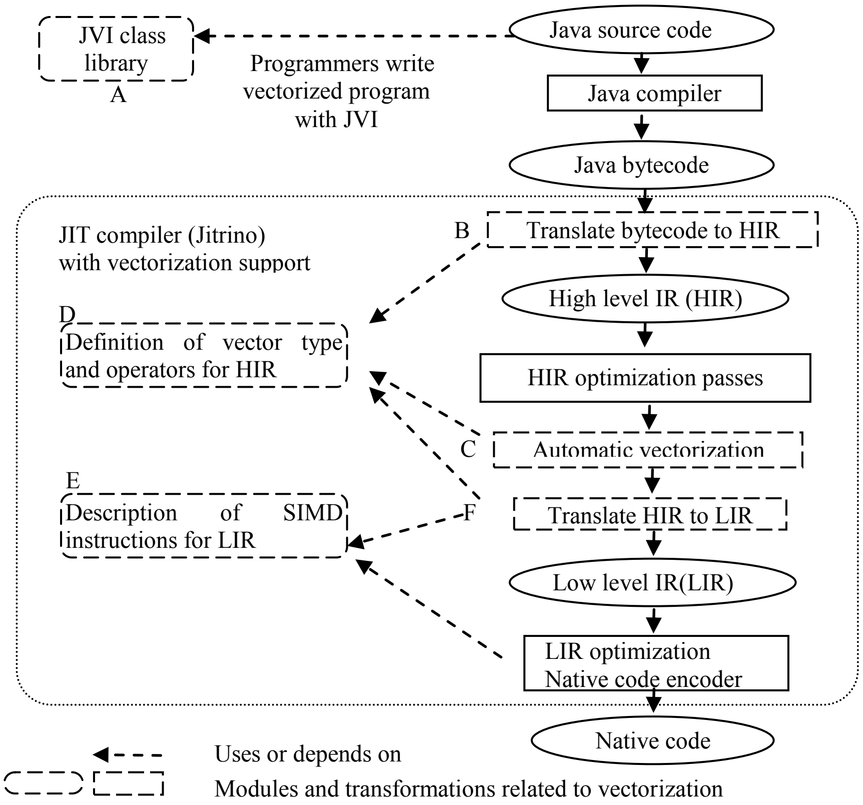

JVI class library
A
Programmers write vectorized program with JVI
Java source code
Java compiler
Java bytecode
JIT compiler (Jitrino) with vectorization support
B
Translate bytecode to HIR
High level IR (HIR)
D
Definition of vector type and operators for HIR
HIR optimization passes
C
Automatic vectorization
E
Description of SIMD instructions for LIR
F
Translate HIR to LIR
Low level IR(LIR)
LIR optimization Native code encoder
Native code
Uses or depends on
Modules and transformations related to vectorization

Fig. 1. Infrastructure of vectorization implementation in Jitrino

Each vector class exposes a set of methods acting as vector operators. These “operators” cover most hardware supported vector operations, including initialization from scalar(s), arithmetic and logical computation, bit shifting, comparison, conditional selection, memory accessing and element shuffling. They provide enough flexibility for programmers to write efficient vectorized code. The following is a piece of the source code a JVI class representing the vector type of `double`.

```
public class F64vec2
{
  public static F64vec2 make(double d0, double d1) { return fail (); }
  public static F64vec2 load(double[] a, int i) { return fail (); }
  public F64vec2 add(F64vec2 B) { return fail (); }
  public F64vec2 sub(F64vec2 B) { return fail (); }
  public F64vec2 mul(F64vec2 B) { return fail (); }
  …… ……
}
```

The exposed vector methods are only used to tell the front end of the JIT compiler (part B) how to translate them into the vector IR defined in part D. They should never be actually called, so their bodies only throw an exception (with `fail()`) to indicate that. These methods are designed to be pure functions, i.e. they never rely on or change the state of “this” object. This design forces programmers to treat vector types as primitive types, since internally, these vector types indeed behave as primitive types rather than classes. They can reside in both memory and registers, and can be passed as values in function calls. This is important to avoid dangerous misuse of the JVI library. At the same time, it gives the compiler more freedom to optimize.

3.2 JVI implementation

JVI interface is defined in class library. Programs using JVI must be translated into machine code by JVM. We extend the HIR of Jitrino by defining vector types and operators (see part D) to represent the JVI interface internally. The existing high level optimizations in Jitrino can be applied to the IR directly.

Part B in Figure 1 is the front end of Jitrino. It translates Java bytecode into Jitrino HIR. JVI class references and method callings are correspondingly translated into vector types and vector instructions of the extended HIR in this stage. We will use the following code fragment from scimark.lu of SPECjvm2008 to illustrate the translation process. This hot loop consumes more than 90% execution time of the benchmark.

```
for (int jj = j + 1; jj < N; jj++) Aii[jj] -= AiiJ * Aj[jj];
```

In the code, `Aii` and `Aj` are two arrays of double type values, and `AiiJ` is a double type variable. The manually vectorized version of the loop is as follows:

```
F64vec2 v_aiij = F64vec2.make (AiiJ);
for (int jj = j + 1; jj < N - 1; jj += 2) {
 F64vec2 v_t1 = F64vec2.load (Aj, jj);
 F64vec2 v_t2 = F64vec2.load (Aii, jj);
 v_t2.sub (v_aiij.mul (v_t1)).store (Aii, jj);
}
```

In the above code, `F64vec2.make(AiiJ)` creates a vector containing two double type values of `AiiJ`. `F64vec2.load` loads two consecutive double type values starting at the given index from the given array. `v_t2.sub()` and `v_aiij.mul()` return the results of vector subtraction and vector multiplication correspondingly between the “this” objects and their arguments. `XXX.store(Aii, jj)` stores two elements of `XXX` into `Aii[jj]` and `Aii[jj+1]`. The bytecode of this code fragment contains the vector type `com.intel.jvi.F64vec` and six calls of its methods. After translated by part B (in Figure 1), the six method calls become the following HIR instructions, where `double<2>` is the vector type of `double`. It is the vector IR defined for `com.intel.jvi.F64vec2`:

```
I247: conv t186 -) t187 : double<2>
I262: ldind  [t199] -) t200 : double<2>
I270: ldind  [t207] -) t208 : double<2>
I271: mult187, t200 -) t209 : double<2>
I272: subt208, t209 -) t210 : double<2>
I274: stind  t210 -) [t207]
```

4 Automatic vectorization

Part C in Figure 1 is the automatic vectorization pass we implement in Jitrino. Since most opportunities for data parallelism occur between iterations of loops, traditional vectorization techniques mainly focus on exploiting loop level data parallelism. This kind of techniques is referred to as loop-based vectorization. Another kind of vectorization is called SLP (Superword Level Parallelism) vectorization [9]. It identifies groups of isomorphic instructions exposing superword level parallelism, and combines them into equivalent vector instructions. The loop-based vectorization exploits data parallelism among different executions of the same instruction, while the SLP vectorization exploits data parallelism among different instructions in the straight-line code (usually in the same basic block), so the SLP vectorization can be a complement to the loop-based vectorization. With loop-unrolling, loop level data parallelism can be transformed into superword level parallelism, so the SLP vectorization can also be used to exploit loop level data parallelism with the help of loop unrolling. However, duplicating instructions that can be parallelized multiple times and then re-recognizing them to be isomorphic from all duplicated instructions and finally combining them back into one vector instruction is not as efficient as the loop-based vectorization. The loop-based vectorization only transforms the loop once. To achieve the same effect as the loop-based vectorization, SLP vectorization also needs induction variable analysis and data dependence analysis (it may not do them, but that will cause missing vectorization opportunities). In fact, SLP vectorization is more complex and inefficient than loop-based vectorization for vectoring loops. As a result, we implement the loop-based vectorization in Jitrino and treat SLP vectorization as a complement that may be implemented in the future.

Our loop-based vectorization is composed of two parts: vectorization analysis, and vectorization transformation. The first part analyzes and collects necessary information for all loops, and the second part performs the transformation.

4.1 Vectorization analysis

The vectorization analysis is applied to the leaf nodes in the loop tree. Only countable leaf loops with single entry and single exit are taken as vectorization candidates. The countability of a loop is determined by its exit condition. The exit condition of a countable loop must be a comparison between a loop invariant value and an induction variable. Whether a variable is loop invariant and how a variable changes is analyzed by the scalar evolution analysis. This analysis is called on demand for requested single variable, and is also called by data dependence analysis for analyzing array indices. The data dependence analysis builds data dependence graph (DDG) among all instructions of a loop. The dependences due to explicit variable references can be easily retrieved from the use-define chains incorporated in the SSA form IR. To determine dependences due to accessing aliased array elements, we first use the simplest GCD (greatest common division) test to filter out most non-aliased cases quickly. Then, according to the number of index variables in the array accessing expressions, we call the ZIV (zero index variable) test or SIV (single index variable) test [13]. For the case of multiple index variables (MIV), we simply assume there is a dependence since this case is relatively rare but the testing cost is quite high.

Different from the algorithms described in [4, 7, 8], our vectorization algorithm tries to vectorize the loop even when there exists cyclic dependence in the loop, or some of the instructions can not be vectorized. This is a novel feature of our vectorization algorithm. As we know, other algorithms just give up the vectorization when there is a dependence circle.

With a strongly connected component (SCC) analysis on the DDG, DDG nodes (instructions) can be grouped into different SCCs. Instructions of trivial SCCs (that only contain a single instruction) that can be supported by hardware vector instructions are considered as candidates for vectorization. All other instructions, including those of non-trivial SCCs and those that are not supported by hardware are considered as candidates for loop unrolling. The candidates for loop unrolling and those for vectorization are not split into two separate loops. Two separate loops require additional temporary array and memory accessesing instructions to pass data between them, which may greately degrade the performance. Our algorithm uses pack and extract operations to transfer data among unrolled scalar instructions and the vectorized vector instructions through registers, which is far more efficient than through memory. The pack operation packs a set of scalar values into a vector, which is implemented by a series of SSE instructions. The extract operation extracts a specific scalar element from a vector, which is implemented by a single SSE instruction.

To increase vectorization opportunities, our algorithm tries to break dependence circles through dynamic alias testing. The algorithm finds DDG edges between two array accessing instructions in non-trivial SCCs, and generates alias testing code into the pre-header of the loop for all the pairs of arrays. Then, it removes all such DDG edges. As a result, some dependence circles may be broken. If there is no any alias, the control flow is directed into the vectorized code path. Otherwise, it is directed into the original loop.

Each instruction selected for vectorization is assigned a vectorization factor (VF). The vectorization factor of an instruction is a number denoting how many scalar operations of that instruction can be parallelized in one corresponding vector operation. For example, suppose the size of vector types are all 128 bits, then the VF of a 32-bit integer instruction is four and the VF of a 64-bit floating point instruction is 2. The vectorization factor of a loop is a number denoting how many iterations of the original loop can be executed in one iteration of the vectorized loop. Instructions in a loop may have different vectorization factors. We choose the largest one as the VF for the loop to maximize the data parallelization. An approximate cost model for evaluating the profit of vectorization is given as follows:

$$C_1(I) = \begin{cases} cost(I) \times vf_c & A(I) = U \\ cost(I_V) \times (vf_c / vf(I)) & A(I) = V \end{cases} \qquad (1)$$

$$C_2(I) = \sum_{o \in opnd(I)} \begin{cases} 0 & A(def(o)) = A(I) \\ cost(vec\_pack) & A(def(o)) = U \wedge A(I) = V \\ cost(vec\_extract) \times vf(I) & A(def(o)) = U \wedge A(I) = U \end{cases} \qquad (2)$$

$$P = \sum_{I \in L} (cost(I) \times vf_c) - \sum_{I \in L} (C_1(I) + C_2(I)) \qquad (3)$$

Notations:
- $C_1(I)$: instruction cost of I after transformation;
- $C_2(I)$: operand cost of I after transformation.
- $A(I)$: transformation action to instruction I. U: to unroll; V: to vectorize.
- vf_c: common VF of the loop;
- $vf(I)$: VF of instruction I.
- I_V : vector version of instruction I.
- $def(o)$: defining instruction of variable o.
- $opnd(I)$: operand set of instruction I.
- $cost(I)$: cost of instruction I.

Equation (1) estimates the execution cost of instruction I after the transformation. Equation (2) estimates the operand retrieval cost of I after the transformation. The *vec_pack* is a pseudo operator representing a set of SSE instructions for packing scalar values into a vector operand. The *vec_extract* is another pseudo operator representing the SSE instruction for extracting a scalar operand from a vector value. Equation (3) calculates the performance profit of the action set given by A.

For instructions that cannot be vectorized, we have no choice other than to unroll them. For other instructions that can be vectorized, we have two choices: to vectorize them or to unroll them. Different determinations of transforming actions for these instructions may lead to different profit. We use a simple local policy to try to maximize the profit. Our algorithm first finds all instructions that can be vectorized, marking them as "to-be-vectorized" and marking all others as"to-be-unrolled". Then, it goes through all instructions marked as "to-be-vectorized" and checks for each one of them whether changing the action for the instruction to "to-be-unrolled" will bring more profit (from eliminating operand retrieval cost). If yes, the algorithm changes the action of that instruction. This greedy policy may not generate the optimal result, but it is good enough in practice considering its simplicity and efficiency.

4.2 Vectorization transformation

After all SCCs of the DDG have been assigned actions, the transformation algorithm traverses all SCCs in the dependence order, in which depended SCCs appear before depending SCCs. For each of the SCCs, the algorithm emits either a vector instruction or a set of scalar instructions according to the action of that SCC. The key structure for connecting vectorized instructions and unrolled instructions is a map from variables of the original loop to arrays of variables of the transformed loop. For each original variable, the mapped-to array stores its copies in the transformed loop, corresponding to a sequence of iterations. Specifically, in our algorithm, the arrays have VF + 1 elements; the first VF elements of the array stores scalar copies of the original variable corresponding to VF iterations, and the last element of the array stores the vector copy of the original variable. Both scalar and vector copies are created on demand. The transformation algorithm is as follows. The function `unroll` emits unrolled scalar instructions to the loop body for instructions in a given SCC. The function `vectorize` emits a vector instruction to the loop body for the single instruction of the given SCC. The function map returns the variable corresponding to the requested iteration. At the first time a variable is requested, it creates the variable and generates extract or pack instructions to initialize that variable.

```
transform () {
  for each SCC, say s, of the DDG in the dependence order
    switch (action of s) {
      case to_unroll: unroll (s); break;
      case to_vectorize: vectorize (s); break;
    }
}
unroll (s) {
  for (i = 0; i < VF; i++)
    for each instruction "'x' '=' 'y' 'op' 'z'" in s {
      create a new scalar variable, say 't'_i;
      emit "'t'_i '=' map('y',i) 'op' map('z',i)";
      MAP('x')[i] = 't'_i;
    }
}
vectorize (s) {
  let "'x' = 'y' 'op' 'z'" be the single instruction of s;
  create a new vector variable, say 'v';
  emit "'v' '=' map('y',VF) 'op' map('z',VF)";
  MAP('v')[VF] = 'v';
}
map (var, i) {
  if (MAP(var)[i] == NULL) {
    if (i < VF) {
      for (j = 0; j < VF; j++) {
        create a new scalar variable, say 't'_j;
        emit "'t'_j '=' 'vec_extract' MAP(var)[VF], j";
        MAP(var)[j] = 't'_j;
      }
    } else {   // i == VF
      create a new vector variable, say 'v';
      emit "'v' '=' 'vec_pack' MAP(var)[0], MAP(var)[1],
            ..., MAP(var)[VF-1])";
      MAP(var)[i] = 'v';
    }
  }
  return MAP(var)[i];
}
```

For example, in the following code, the arrays and the variable C are of floating point type. Reading from a[i-1] at line #5 and writing to a[i] at line #7 plus the addition at line #6 create a dependence circle with distance 1 and hence cannot be vectorized. Other instructions do not belong in any dependence circles and can be vectorized. The comments in the code indicate which SCC an instruction belongs in.

```
1 for (int i = 1; i < N; i++) {
2    t1 = b[i];   // SCC 0
```

```
  t2 = c[i];     // SCC 1
  t3 = t1 * t2;  // SCC 2
  t4 = a[i - 1]; // SCC 3
  t5 = t4 + t3; // SCC 3
  a[i] = t5;     // SCC 3
  t6 = t5 * C;  // SCC 4
  d[i] = t6;     // SCC 5
}
```

The following is the transformed code commented with changes of the variable map, i.e. the `MAP` in the above algorithm. The SCCs are processed just in the order of SCC numbers given in the comments of the above code. Lines #2 through #4 are instructions generated for SCCs #0 through #2. Lines #5 through #20 are for SCC #3. Lines #21 through #23 are for SCC #4 and line #24 is for SCC #5.

```
1 for (int i = 1; i < N; i++) {
2  v1 = b[i:i+3];  // t1 -> [0, 0, 0, 0, v1]
3  v2 = c[i:i+3];  // t2 -> [0, 0, 0, 0, v2]
4  v3 = v1 * v2;   // t3 -> [0, 0, 0, 0, v3]
5  s1 = a[i - 1];  // t4 -> [s1, 0, 0, 0, 0]
6  s2 = vec_extract(v3, 0);      // t3 -> [s2, 0, 0, 0, v3]
7  s3 = s1 + s2;   // t5 -> [s3, 0, 0, 0, 0]
8  a[i] = s3;
9  s4 = a[i];      // t4 -> [s1, s4, 0, 0, 0]
10 s5 = vec_extract(v3, 1);      // t3 -> [s2, s5, 0, 0, v3]
11 s6 = s4 + s5;   // t5 -> [s3, s6, 0, 0, 0]
12 a[i + 1] = s6;
13 s7 = a[i + 1]; // t4 -> [s1, s4, s7, 0, 0]
14 s8 = vec_extract(v3, 2);      // t3 -> [s2, s5, s8, 0, v3]
15 s9 = s7 + s8;   // t5 -> [s3, s6, s9, 0, 0]
16 a[i + 2] = s9;
17 s10 = a[i + 2];// t4 -> [s1, s4, s7, s10, 0]
18 s11 = vec_extract(v3, 3); // t3->[s2, s5, s8, s11, v3]
19 s12 = s10 + s11; // t5 -> [s3, s6, s9, s12, 0]
20 a[i + 3] = s12;
21 v4 = vec_pack(s3, s6, s9, s12);//t5->[s3,s6,s9,s12,v4]
22 v5 = vec_duplicate(C); // C -> [0, 0, 0, 0, v5]
23 v6 = v4 * v5;  // t6 -> [0, 0, 0, 0, v6]
24 d[i:i+3] = v6;
25 }
```

5 Experiments and Discussions

We use SPECjvm2008 to evaluate the impact of our vectorization on performance on two representative multi-core platforms:

1. Core i7 965 3.2GHz with 8MB L3, 4.80 GT/s QPI and 3GB DDR3 memory, representing desktop machines.

2. Dual Xeon 5560 2.8 GHz with 8MB L3, 6.4 GT/s QPI and 12G DDR3 memory, representing server machines.

Processors of the machines are both based on the Nehalem micro-architecture with a powerful out-of-order engine and supporting 32-bit and 64-bit scalar instruction set and up to SSE4.2 instruction set. We used 32-bit Linux as the operation system (OS) on both platforms.

The performance measurement includes the JIT compiling time, but the compiling time only occupies a very little portion of the whole running time of the benchmarks. In fact, the additional compiling time of the automatic vectorization comparing to JVI only comes from the automatic vectorization pass, which only applies to few loops satisfying many restrictions, and hence has very little impact on the benchmark scores. Therefore, JVI and automatic vectorization achieve similar performance and we do not distinguish them in the following discussions.

5.1 Performance improvement by vectorization

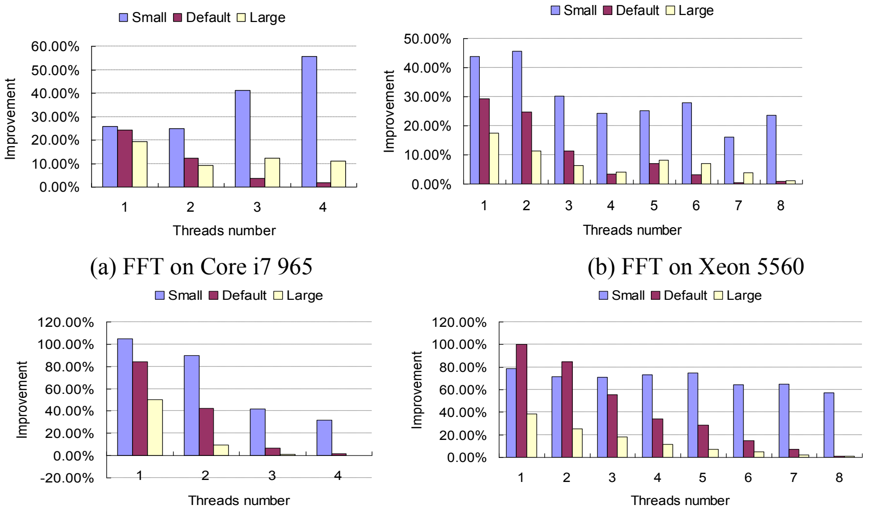
During the execution of SPECjvm2008, there are totally 44 loops of 37 methods within benchmarks or libraries being vectorized. Though many of them only have a very low coverage and hence do not significantly contribute to the performance gain. These numbers at least indicate that there are considerably vectorization opportunities in general Java programs and libraries.

In the performance evaluation, due to their high coverage of the hottest loop, scimark.lu (LU) and scimark.fft (FFT), two sub-benchmarks of SPECjvm2008, are selected as the main workloads for our vectorization evaluation. Both of these benchmarks have three kinds of inputs. They are small, default and large input sets, corresponding to 512KB, 8MB, 32MB input data set per thread. According to the data size requirement, we configure Harmony VM with maximal 600M heap to eliminate the unnecessary overhead from garbage collection.

Figure 2 shows the program level performance improvements of LU and FFT. Multi-threading data are provided to check the vectorization effect in multi-core systems. From the figure we can see that vectorization can get obvious performance gain in almost all scenarios. With single thread, FFT and LU get 55% and 107% performance improvements on the i7 platform, and 45% and 100% on the Xeon platform.

On the other hand, we can see that with the increment of the number of threads, the performance gain is reduced. Especially, for the default and large input data set, the performance gain degradation on the i7 platform is much faster than on the Xeon platform. Figure 3 shows the scalability of LU benchmark. From the data, we can find that, by vectorization, the scalability becomes worse on both platforms.

Based on the fact that the size of the default and large data set of LU and FFT is larger than the whole L3 cache size, we deduce that the limit of memory bus bandwidth is the main reason that restrains the performance gain of vectorization. The higher memory bus bandwidth on the Xeon platform remedies the problem to some degree, and hence causes better scalability comparing with the i7 platform.



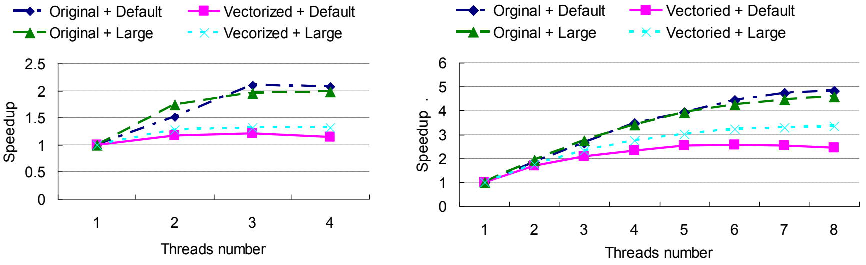
(a) FFT on Core i7 965

(b) FFT on Xeon 5560

(c) LU on Core i7 965

(d) LU on Xeon 5560

Fig.2. the performance improvement of FFT and LU by vectorization



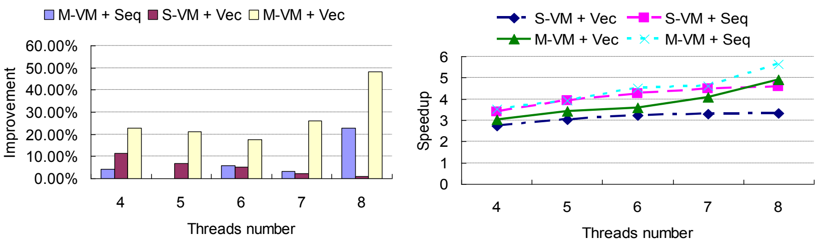
(a) LU on Core i7 965

(b) LU on Xeon 5560

Fig.3. the scalability of LU

To verify the bandwidth effect, we use the NUMA feature of the Xeon 5560 platform. On NUMA enabled platforms, accessing the local memory of the host processor is much faster than accessing the remote memory. Cores accessing local memory do not consume the bus bandwidth of remote memory.

The NUMA feature can not be exploited with current Harmony VM because the VM allocates and binds whole memory heap to single processor in the main thread, although all benchmark threads run in single VM (Single VM mode). To illustrate the impact of NUMA feature on bandwidth with Harmony VM, we simulate the local memory access by running one benchmark thread in one VM instance, and binding the VM with one core. Multiple VM instances (Multiple VM mode) are executed at the same time to simulator multiple benchmark threads.



(a) LU performance improvements (b) LU scalabilities

Fig.4. LU performance improvements and scalabilities in different scenarios

In the experiment, we run both original sequential and vectorized codes in single VM and multiple VMs mode, totally four scenarios are executed. In both execution modes, all the benchmark threads are evenly mapped to different processors, and large data input set is used to make sure that the effect of cache misses is negligible.

Figure 4 shows the data of benchmark LU from Xeon 5560 platform. The data of 4~8 threads are showed to illustrate the scalability problem. Figure 4 (a) shows the performance improvements of benchmark LU with the aids of NUMA and vectorization. In the figure, the sequential execution in single VM mode is used as the base line of the performance. From the data we can find that NUMA can dramatically improve the performance of vectorized programs. Figure 4 (b) compares the scalabilities of the LU benchmark running in four scenarios. From the data we can find that the scalability problem can be largely solved with NUMA architecture.

5.2 Limitation of the partial vectorization

The partial vectorization algorithm described in the last section is powerful in the sense of exploiting data parallelism. However, whether it can improve the performance of a program depends on two factors: how much the inherentparallelism exists in the program, and how well the micro-architecture automatically exploits the parallelism without the vectorization. For example, most modern processors have superscalar micro-architecture that can exploit general instruction-level parallelism. If the inherent data parallelism of a program has been sufficiently exploited by the superscalar engine of the processor, performing vectorization for that program cannot bring any more performance gain.

We observe this phenomenon with the benchmark scimark.sor, whose hot spot is a loop covering more than 90% execution time. The pseudo-code of the loop is given below:

```
for (j = 1; j < N; j++)
  G[j]= A * (Gm[j] + Gp[j] + G[j-1] + G[j+1]) + B * G[j];
```

In this loop, all arrays and variables except `j` and `N` are of `double` type. With our partial vectorization algorithm, the first addition and the last multiplication at the second line are vectorized, and others are unrolled due to the dependence circle. One `double` addition, one `double` multiplication and three `double` loads are saved per two iterations. However, the performance of the vectorized program does not improve compared to the non-vetorized version. The reason is that the inherent parallelism of the loop has been sufficiently exploited by the powerful superscalar engine of Nehalem.

Figure 5.(a) is the data dependence graph of the loop. The least execution time of this loop is determined by the execution time of the circle of the dependence graph in spite of how strong the parallelization ability of a processor is. Suppose `double` type addition, multiplication, load and store requires 3, 5, 4 and 1 processor cycles respectively, then the least execution cycles of the loop is (3 × 3 + 5 + 4 + 1) × N = 19N . During the 19 cycles of each iteration, the Nehalem micro-architecture with six issue ports, three pipelined execution units, one load unit and one store unit can easily arrange the other loads and computations to be executed concurrently. Thus, the execution speed of this loop has reached its toplimit.

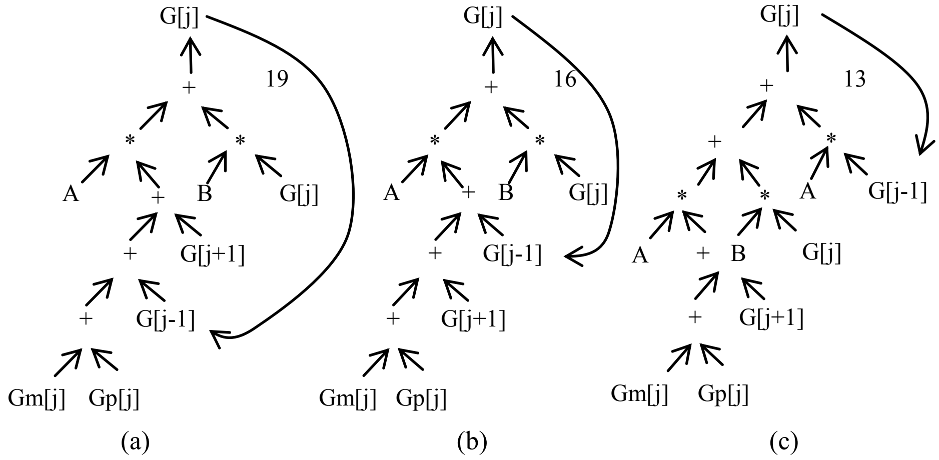


Fig. 5. Data dependence graph of the kernel loop of scimark.sor

In this example, expression re-association can help to expose more parallelism. When floating point precision is not required to be strict, the computation order can be changed to shorten the length of the dependence circle in the loop. The dependence graphs of the following two versions of the loop are shown in Figure 5 (b) and (c).

```
for (j = 1; j < N; j++)
  G[j] = A * (Gm[j] + Gp[j] + G[j+1] + G[j-1]) + B * G[j];
for (j = 1; j < N; j++)
  G[j] = A * (Gm[j] + Gp[j] + G[j+1]) + B * G[j] + A * G[j-1];
```

With the reorder of the computation, the numbers of the approximate cycles required by the dependence circles of version (b) and version (c) are 16 and 13 respectively, and their (non-vectorized) execution speeds are improved by about 30% from (a) to (b) and then 40% from (b) to (c). This means that, the superscalar is still powerful enough for exploiting the increased data parallelism even when the dependence circle is minimized, and there is no further optimization space left for the vectorization technique.

6 Conclusion and Future Work

In this paper, we introduce our Java vectorization work that uses two ways to exploit the data parallelism of Java applications. They can achieve similar vectorization results. Our work shows up to 55% and 107% performance gain for scimark.fft and scimark.lu of SPECjvm2008 when running in one thread. Performance gain was also observed with eight threads.

In the manual vectorization, we define, design and implement a unified and generic set of Java Vectorization Interface so that Java programmers can use the interface for Java programming without considering the specific hardware supports. In the automatic vectorization, we propose a novel algorithm which provides aggressive vectorization supports to maximize the vectorization benefit.

We analyze the result of our work, and our investigation shows that several factors such as memory bus bandwidth, superscalar micro-architecture and code shape need to be considered when applying vectorization techniques.

Currently we are porting this work to JavaScript engine. It is challenging because JavaScript language is dynamically typed. Lots of branches inside code make regular data parallelism hard to be exploited. We are developing type analysis technique to overcome the problem.

References

1. The Apache Software Foundation. Apache Harmony. http://harmony.apache.org
2. Larsen, S., Amarasinghe, S.P.: Exploiting superword level parallelism with multi-media instruction sets. In: PLDI. (2000) 145–156
3. El-Mahdy, S.E.S.A., El-Mahdy, A.: Automatic vectorization using dynamic compi-lation and tree pattern matching technique in jikes rvm. In: ICOOOLPS. (2009) 63–69
4. Free Software Foundation. Auto-vectorization in gcc. http://gcc.gnu.org/projects/tree-ssa/vectorization.html
5. Free Software Foundation. Gcc. http://gcc.gnu.org
6. Naishlos, D.: Autovectorization in gcc. In: GCC Summit. (2004) 105–118
7. Nuzman, D., Rosen, I., Zaks, A.: Auto-vectorization of interleaved data for simd. In Schwartzbach, M.I., Ball, T., eds.: PLDI, ACM (2006) 132–143
8. Nuzman, D., Zaks, A.: Autovectorization in gcc - two years later. In: GCC Summit. (2006) 145–158
9. Rosen, I., Nuzman, D., Zaks, A.: Loop-aware slp in gcc. In: GCC Summit. (2007) 131–142
10. Novell Corporation. Mono.Simd.Namespace. http://www.mono-project.com/
11. Intel Corporation. IA-32 Intel Architecture Optimiztion Refernce Manual. Copyright 1999~2003
12. Pedro V. et al: Automatic loop transformations and parallelization for Java. In ICS'00: 14th Int. Conf. on Supercomputing, pages 1-10, 2000.
13. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)