JOURNAL OF OBJECT TECHNOLOGY

Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 2, March-April 2005

Java Optimization for Superscalar and Vector Architectures

Douglas Lyon, Fairfield University, Fairfield CT, U.S.A.

Windows NT crashed.
I am the Blue Screen of Death.
No one hears your screams.
-Anon

Abstract

This paper describes the refactoring of Java code to take advantage of the superscalar and vector architectures available on many modern desktop computers.

The unrolling of Java loops is shown to cause some speed-ups for Java code. However, our benchmarks reveal that Java still lags behind vectorized C code.

The present state-of-the-art in computer hardware has outpaced the current state of the JIT (Just-In-Time) compilers. We have resorted to modifying our Java code to make use of JNI (Java Native Interface) based vector-accelerated C programs to obtain speed-ups from 2 to 10 times.

1 THE SUPERSCALAR G4

The superscalar G4 core (also known as the Motorola MPC74xx series processor) is capable of executing *four instructions per clock cycle*. Superscalar processors are based on pipelined MIMD (Multiple Instruction, Multiple Data) architectures. Simple loop unwinding can take advantage of such architectures. The G4 also has a 128-bit wide vector unit called the *Altivec*. The Altivec has 32 registers with 128 bits each and represents a departure from the pure general purpose CPU and a focus upon a SIMD (Single Instruction, Multiple Data) architecture that enables fine-grained parallelism. A block diagram depicting the PowerPC execution flow is shown in Figure 1.

Cite as follows: Douglas Lyon: "Java Optimization for Superscalar and Vector Architectures", in *Journal of Object Technology*, vol. 4, no. 2, March-April 2005, pp. 27-39

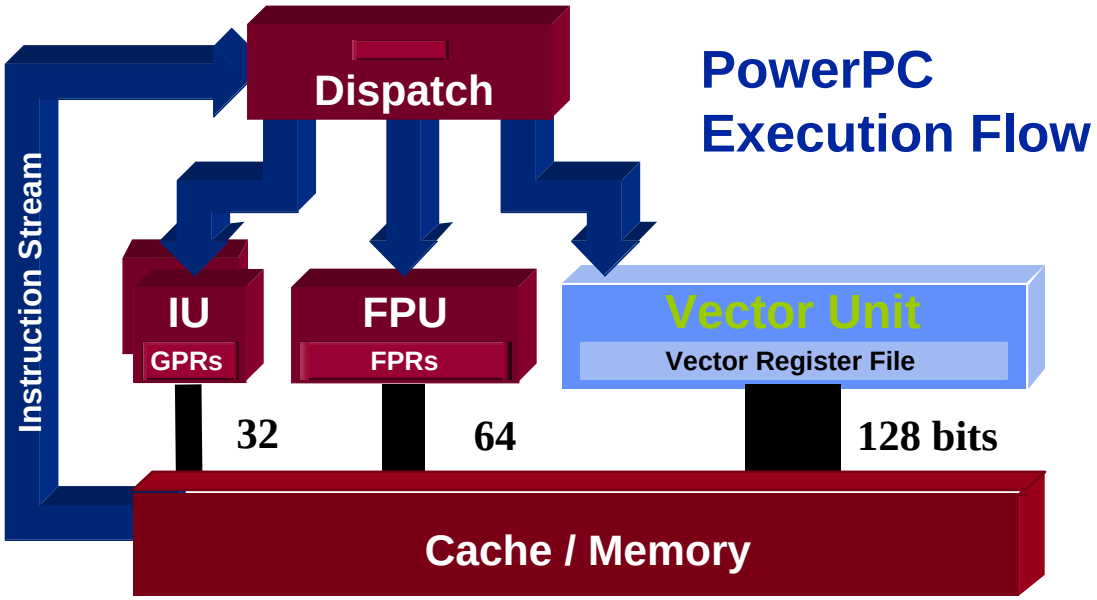


Figure 1. The G4 Execution Flow

Although the PowerPC (PPC) is a RISC (Reduced Instruction Set Computer) system, the addition of the Altivec processor adds over 160 machine instructions. These instructions map directly to C subroutines that can be invoked for the purpose of acceleration. In an ideal world, vectorizing compilers should be able to map data into these instructions and take advantage of the Altivec. In our experience, however, this almost never happens, even with the simplest of code [Freescale].

The trend toward SIMD-style signal processing type architectures is gaining wide acceptance (first with Intel's MMX instructions and lately with the new SSE and SSE2 instructions). The SIMD idea can be traced back to Alan Turing's 1946 parallel computing studies on *VLIW* (Very Long Instruction Word) computers. A key difference between the super-scalar architecture and the VLIW architecture is that programs are typically changed in order to take advantage of the new architecture. In theory, Java obviates the need for the programs to change, in response to a new architecture, since this is the responsibility of the JIT (which should mean recompilation for each run). However, the present JIT's do not take advantage of the modern VLIW machine [Patterson and Hennessy].

The computer industry is motivated to incorporate MIMD architectures into their hardware because they enable the fast computation of homogenous data arrays. These appear in several applications, such as signal processing, image processing, computer graphics, computer vision, communications, etc. In fact, an entire industry has grown up around the creation of chips dedicated to the processing of homogenous data arrays. These are typically called DSP (Digital Signal Processing) chips. It is therefore a matter of an evolutionary trend that we see DSP functions incorporated into general-purpose processors.

2 WHAT ARE C PROGRAMMERS DOING?

To observe just how C programmers optimize their code for the PPC, consider the following function, which is designed to add *n* numbers in an array and return the result:

```
float sum(float a[], int n) {
  int i = 0;
  float s = 0;
  for (i=0; i < n; i++) {
    s = s + a[i];
  }
  return s;
}
```

The kernal of the for-loop translates into the following assembler:

```
L5:
        lwz r0,32(r30)
        slwi r2,r0,2
        lwz r0,88(r30)
        add r2,r2,r0
        lfs f13,36(r30)
        lfs f0,0(r2)
        fadds f0,f13,f0
        stfs f0,36(r30)
        lwz r2,32(r30)
        addi r0,r2,1
        stw r0,32(r30)
        b L2
```

Based on the addition of 1024 floats (32 bits, each), we find that the unoptimized code, running on a G4/400 Mhz, executes in 9 microseconds. In an effort to get the superscalar nature of the processor to kick-in, it is typical for the C programmer to unwind the loop. Since the G4 processor has a 4 stage super-scalar pipe-line, we can obtain some speed-up by using:

```
float sumV(float a[], int n) {
  int i = 0;
  register float s[4] = {0};
  for (i=0; i < n; i=i+4) {
    s[0] = s[0] + a[i];
    s[1] = s[1] + a[i+1];
    s[2] = s[2] + a[i+2];
    s[3] = s[3] + a[i+3];
  }// compaction phase...
  return s[0]+s[1]+s[2]+s[3];
}
```

The kernal of the for-loop translates into the following assembler:

```
L5:
        lwz r0,32(r30)
        slwi r2,r0,2
        lwz r0,104(r30)
        add r2,r2,r0
        lfs f13,48(r30)
```

```
lfs f0,0(r2)
fadds f0,f13,f0
stfs f0,48(r30)
lwz r0,32(r30)
slwi r2,r0,2
lwz r0,104(r30)
add r2,r2,r0
addi r2,r2,4
lfs f13,52(r30)
lfs f0,0(r2)
fadds f0,f13,f0
stfs f0,52(r30)
lwz r0,32(r30)
slwi r2,r0,2
lwz r0,104(r30)
add r2,r2,r0
addi r2,r2,8
lfs f13,56(r30)
lfs f0,0(r2)
fadds f0,f13,f0
stfs f0,56(r30)
lwz r0,32(r30)
slwi r2,r0,2
lwz r0,104(r30)
add r2,r2,r0
addi r2,r2,12
lfs f13,60(r30)
lfs f0,0(r2)
fadds f0,f13,f0
stfs f0,60(r30)
lwz r2,32(r30)
addi r0,r2,4
stw r0,32(r30)
b L2
```

Unwinding the loop by 4 elements gives us a speed up of 2 microseconds (that is a 7 microsecond execution time). However, based on an inspection of the assembler, we find that no vector operations are added to the assembler output. Thus, we have been able to take advantage of the super-scalar architecture, but not the internal vector architecture.

Encouraged by the speed up, we double the unwinding of the loop. Further speed-ups can be had, since there may be a speculative pre-fetch in the CPU that enables the needed data to appear in the L1 cache (which is 32k bytes in size). Thus, we expand the unwinding and note that the array length must be an even multiple of 8, or we will access the array out of bounds:

```
float sumV8(float a[], int n) {
  int i = 0;
  register float b[8] = {0};
  for (i=0; i < n; i=i+8) {
    b[0] = b[0] + a[i];
    b[1] = b[1] + a[i+1];
    b[2] = b[2] + a[i+2];
    b[3] = b[3] + a[i+3];
    b[4] = b[4] + a[i+4];
    b[5] = b[5] + a[i+5];
    b[6] = b[6] + a[i+6];
    b[7] = b[7] + a[i+7];

  }

  return b[0]+b[1]+b[2]+b[3]+
         b[4]+b[5]+b[6]+b[7];
}
```

The unwinding of the loop by 8 does give a small speed-up (executing the sum in only 6 microseconds). It is well to note that unwinding by 16 actually slows the code by a small margin, and thus there are diminishing returns in using the unwinding approach.

One would hope that a vectorizing compiler would see the structure of the above code and take advantage of the obvious parallelism. This appears to be beyond the GCC compilers ability (at the moment). In support of the Altivec processor a *-faltivec* has been added to the GCC compiler. This enables the direct invocation of extensions to the C language. These include the *vector* data type, as well as direct invocation of subroutines that map to Altivec assembler instructions, without the inclusion of header files [Freescale]. For example:

```
float sumVAltivec2(float a[], int n) {
   int i;
   register vector float vb;
   // ensures intermediate sum vb stays in register
   vector float dest;
   // Declare vector in memory to move contents out of register
   float *p = (float *)(&dest);
   // set pointer to it in order to add the four floats
   vector float *input=(vector float *)a;
   // Set up input pointer at array address

   vb=(vector float)vec_splat_u32(0);
   // clear intermediate sum (integer 0 == float 0)
   for (i=0; i < n; i+=4) {
     vb = vec_add(*input++,vb);
     // c = vec_add(a, b);
   }
   dest=vb;
   return *p+*(p+1)+*(p+2)+*(p+3);
   // compacting phase
}
```

The assembler for the kernal of the for-loop is:

```
L5:
        addi r9,r30,68
        lwz r2,0(r9)
        lvx v0,0,r2
        addi r2,r2,16
        stw r2,0(r9)
        addi r2,r30,80
        lvx v1,0,r2
        vaddfp v0,v0,v1
        addi r2,r30,80
        stvx v0,0,r2
        lwz r2,32(r30)
        addi r0,r2,4
        stw r0,32(r30)
        b L2
```

The assembler reveals that *vec_add* has been mapped to the assembler instruction *vaddfp*. Thus, we have finally started to take advantage of the Altivec processor. However, to do it required major modification to the source code.

Even worse, the code has gotten rather much harder to understand. The invocation, *vec_splat_u32* zeros out the vector (a 128 bit quantity). The *vec_add* instruction is adding 128 bits (i.e. 4, 32-bit floats) at a time. The new code now runs in 3 microseconds (about 300% faster then the original *sum* function).

Not to be outdone, an optimized assembler language subroutine is available in a library called *SAL* (Scientific Application Library) [Mercury].

```
// Check out my new sal call:
//   void svex(
//   float *a,                    /* input vector */
//   int i,                    /* address stride for a */
//   float *c,                    /* output scalar */
//   int n,                    /* real element count */
//   int flag                  /* ESAL flag */
//   );
float sumSal(float a[], int n) {
  float sum;
  svex(a,1,&sum,n,0);
  return sum;
}
```

The *svex* based *sumSal* function can sum the 1024 floats in just 1.5 microseconds (some 600% faster than the original *sum* function).

Even faster results can be had when using the VAST compiler [Crescent]. A non-disclosure agreement requires that I remain silent about how much faster VAST is. Also, some of the code had to be altered in order to get the code to work. I am told that my code uses non-standard C extensions.

3 JAVA HAS SPEED PROBLEMS

In Java it is possible to write the same *sum* function as in C:

```
public static final float sum(
          final float[] a) {
     float s = 0;
     for (int i = 0; i < a.length; i++)
         s += a[i];
     return s;
 }
```

```
Using the command:
java -noclassgc -Xint math.Mat1
```

We turn off garbage collection and run our benchmark in interpreted mode. This takes 503 microseconds. The just-in-time compiler can be turned back on using:

```
java -noclassgc -Xmixed math.Mat1
```

With the JIT on, the *sum* method takes 41 microseconds to run (about 4.5 times slower than the corresponding C code). We are using the same G4 processor (running at 400 Mhz). Just as in Section 1, we proceed to unwind the for-loop:

```
public static final float sumV(
          final float a[]) {
     int i = 0;
     int n = a.length;
     float s[] = {0, 0, 0, 0};
     for (i = 0; i < n; i = i + 4) {
         s[0] = s[0] + a[i];
         s[1] = s[1] + a[i + 1];
         s[2] = s[2] + a[i + 2];
         s[3] = s[3] + a[i + 3];
     }// compaction phase...
     return s[0] + s[1] + s[2] + s[3];
 }
```

The above code saves no time at all, and, in fact runs in 48 microseconds (7 microseconds slower). What is going on? Each test is run 10 times, and the fastest time is being reported. This gives the JIT a chance to kick in. What happens if we only run the test once? The *sum* code runs in 246 microseconds (about 30 times slower than the slowest C code).

The following code can take advantage of pipelining, without introducing independence in the computation:

```
public static final float sum4(final float[] a) {
          float s = 0;
          for (int i = 0; i < a.length; i = i + 4) {
              s = a[i] +
                  a[i + 1] +
                  a[i + 2] +
                  a[i + 3] +
                  s;
          }
          return s;
      }
```

It is able to execute in 39 microseconds. Expansion to 8 terms yields:

```
public static final float sum8(final float[] a) {
       float s = 0;
       for (int i = 0; i < a.length; i = i + 8) {
           s = a[i] +
               a[i + 1] +
               a[i + 2] +
               a[i + 3] +
               a[i + 4] +
               a[i + 5] +
               a[i + 6] +
               a[i + 7] +
               s;
       }
       return s;
   }
```

The above program runs in 31 microseconds.

Thus with simple loop unwinding, Java seems able to get a 25% increase in speed, yet it is still 3 times slower than our slowest C code (and 30 times slower than vector-optimized assembler).

4 WHAT IS GOING ON WITH JAVA'S COMPILER?

Our simple-minded example just adds up a list of numbers, yet we see dramatic speed hits when we start to benchmark our code. Also, we find no direct improvement available for vectorization (though there is some improvement with loop unwinding).

What is going on with the Java compiler? Does it understand anything about vectorization? It might be useful for us to examine the assembler output for some of our code. Given the input code:

```
public static final float sum(
          final float[] a) {
     float s = 0;
     for (int i = 0; i < a.length; i++)
         s += a[i];
     return s;
 }
```

The compiler produced:

```
Code:
 0:   fconst_0
 1:   fstore_1
 2:   iconst_0
 3:   istore_2
 4:   iload_2
 5:   aload_0
 6:   arraylength
 7:   if_icmpge       22
 10:  fload_1
 11:  aload_0
```

```
12:  iload_2
13:  faload
14:  fadd
15:  fstore_1
16:  iinc    2, 1
19:  goto    4
22:  fload_1
23:  freturn
```

Now consider the *sum4* function:

```
public static final float sum4(final float[] a) {
     float s = 0;
     for (int i = 0; i < a.length; i = i + 4) {
         s = a[i] +
             a[i + 1] +
             a[i + 2] +
             a[i + 3] +
             s;
     }
     return s;
 }
```

Compare the assembler output for *sum* with that of *sum4*:

Code:

```
0:   fconst_0
1:   fstore_1
2:   iconst_0
3:   istore_2
4:   iload_2
5:   aload_0
6:   arraylength
7:   if_icmpge       41
10:  aload_0
11:  iload_2
12:  faload
13:  aload_0
14:  iload_2
15:  iconst_1
16:  iadd
17:  faload
18:  fadd
19:  aload_0
20:  iload_2
21:  iconst_2
22:  iadd
23:  faload
24:  fadd
25:  aload_0
26:  iload_2
27:  iconst_3
28:  iadd
29:  faload
30:  fadd
31:  fload_1
32:  fadd
```

```
33:  fstore_1
34:  iload_2
35:  iconst_4
36:  iadd
37:  istore_2
38:  goto    4
41:  fload_1
42:  freturn
```

Thus we see in the byte codes an output that has expanded in direct correspondence with the expansion in the input code. Yet the code does appear to run somewhat faster, indicating that the JIT/G4 is taking advantage of loop unwinding to exploit pipelining. Wherever the speed up is occurring, we know for sure it is not with Javac.

5 WHAT CAN WE DO TO SPEED UP JAVA?

Several projects have been started to address poor numerical performance in Java. For example, IBM has created the Ninja project, a closed-source C-based system that is compiled only for the PowerPC. Ninja uses C to create a series of JNI (Java Native Interface) assembler calls in order to speed subroutine invocations. If Ninja were an open-source project then we could recompile it for other platforms. As it is, it accelerates PowerPC only.

There are some very high-performance vector accelerated C libraries available from Apple Computer [Apple]. The Apple code is open-source but highly optimized for the PowerPC. Additionally, it has not been interfaced to JNI. To be of use to Java programmers, a more portable version of the vector-accelerated code is needed, so that several platforms can take advantage of the vectorization.

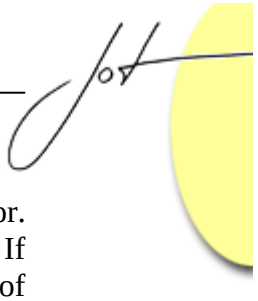
There are portable, open-source, vector-accelerated C libraries available for signal processing [VSIPL]. However, such libraries do not have JNI interfaces, and generating them is a non-trivial exercise.

6 ON THE VECTORIZATION OF JAVA

This section describes a low-level vector-based API that is open-source. In addition, a Java reference implementation is available, for portability to platforms where the C code is not vectorized.

The accelerated C-based API can be recompiled on a per-platform basis, thereby speeding up code that has been refactored to take advantage of the vector-based subroutines. This is going to require that the programmer alter code, in order to get the speed advantage. A nice alternative would be to create a vectorizing JIT compiler.

Luca Lutterotti created a project that contains the source code for implementing all the Altivec operations in Java. Benchmarks show that there can be a 2 to 10 times speed-up over pure Java implementations [Lutterotti].



The Lutterotti library automatically detects the presents of the Altivec processor. Java code is supplied that will compute the result (when no libraries are available). If there are native C-code libraries or if there are native C-code libraries that make use of Altivec, those are used instead. From the programmer's point of view, a single, high-level subroutine is supplied:

```
public static void sqrtf(float[] A, float[] B) {
                switch (libraryType) {
                        case NO_LIBRARY:
                                if (A == null)
                                        break;
                                int size = A.length;
                                if (B == null || (B.length < size))
                                        B = new float[size];
                                for (int i = 0; i < size; i++)
                                        B[i] = (float) Math.sqrt(A[i]);
                        break;
                        case NATIVE_LIBRARY:
                                ssqrtf(A, B);
                        break;
                        case ALTIVEC_LIBRARY:
                                vsqrtf(A, B);
                        break;
                }
        }
```

A short array can actually slow down the execution of JNI-based computations (due to overhead). On the other hand, given arrays of length 256 floats or longer, the above code can run from 2 to 10 times faster than the Java source code. Size 256 arrays are 10 times faster in Altivec accelerated code than the pure Java code. However, size 1024 arrays are only 2.6 times faster than the pure Java code. What happened? Based on my examinations of the code, there appears to be allocation and freeing of array storage, at run-time. This is sure to cause a speed-hit (and one that might be avoidable, given more effort).

7 CONCLUSION

The creation of application specific API's appears to be a trend in Java. Particularly where speed is critical. For example, there is the Java Advanced Imaging (JAI) API used for image processing, and the Java3D API for graphics. These API's are closed source and typically written in C or C++. We, as Java programmers, must hope that Sun will support our platform with the new API, or Java programs that make use of the new API will not run.

For example, for several years, Windows, Solaris and Linux variants had Java3D and JAI, but the Macintosh did not. As a result, Mac users had to use another platform in order to use these APIs.

Application specific API's are basically frameworks that encourage code reuse. (Altivec is not meant for double precision (i.e., 64 bit) floating point numbers. As a result, vectorization is limited to integers and single-precision floating point numbers.

Calling subroutines outside of the Java environment exchanges reliability for speed. The C code can generate segmentation faults, access memory in ways that Java cannot and generally increases the complexity of the system. We have found that complexity is inversely related to reliability.

It would be far better for reliability and complexity if there were vector-based byte codes that could be created by Javac. These could then map into low-level vector instructions (i.e., Altivec or SSE/SSE2). Clearly, heroic refactoring of legacy code is costly, in terms of programmer time. Even better would be a JIT that understands the vectorization of code (thus leaving the byte codes intact). With the advent of a JIT that can take advantage of pipelined MIMD and SIMD architectures, common on today's desktops, Java would be propelled into the world of high-performance numerical computing [Arvedahl] [Sanseri].

ACKNOWLEDGEMENT

The author is indebted to Rob Distinti, Member of Technical Staff at Norden Systems, for his contribution of the vectorized version of the *sum* function.

REFERENCES

[Apple] http://developer.apple.com/hardware/ve/vector_libraries.html.

[Arvedahl] Svante Arvedahl. *Java Just-In-Time Compilation for the IA-64 Architecture*. 2002. Master's thesis, Linkoping University.

[Crescent] Crescent Bay Software, http://www.crescentbaysoftware.com/compilertech.html

[Freescale] *Altivec Technology Programming Interface Manual*, Motorola/Freescale, 1999, http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

[IBM] Ninja (Numerically Intensive Java) http://alphaworks.ibm.com/tech/ninja

[Lutterotti] Luca Lutterotti, http://www.ing.unitn.it/~luttero/javaonMac/

[Mercury] *SAL Reference Manual*, TC-SAL-RM-570, Mercury Computer Systems, Inc., Chelmsford, MA 01824-2820, May, 2002. http://www.mc.com/.

[Patterson and Hennessy] David Patterson, John Hennessy: *Computer Organization and Design*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.

[Sanseri] Samuel K. Sanseri: *Toward an Optimizing JIT Compiler for IA-64*. 2000. Master's thesis, Portland State University, OR, US.

[VSIPL] Vector Signal Image Processing Library, http://www.vsipl.org/

About the author

After receiving his Ph.D. from Rensselaer Polytechnic Institute, **Dr. Lyon** worked at AT&T Bell Laboratories. He has also worked for the Jet Propulsion Laboratory at the California Institute of Technology. He is currently the Chairman of the Computer Engineering Department at Fairfield University, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. E-mail Dr. Lyon at Lyon@DocJava.com. His website is http://www.DocJava.com.