JAVA* FOR HPC:  A STORY OF PERFORMANCE

Razvan Lupusoru, Vivek Deshpande, Sandhya Viswanathan
November 18, 2017
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Agenda

- Background
  - JIT Compiler for performance
- Current Compiler Optimizations
- Vector API for Java

Does not cover:

- Java message passing, GPU programming.
BACKGROUND
Java Popularity

TIOBE Programming Community Index
Source: www.tiobe.com

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
#1 for Cloud

5 million students study Java

10 million Java developers worldwide

15 billion devices run Java

#1 platform for development in the cloud

Netflix powers through 2 billion content requests per day with Java-driven architecture.

NETFLIX

https://go.java/twitter.html

Spotify

https://www.youtube.com/watch?v=UNg9lmk60sg

https://www.youtube.com/watch?v=UNg9lmk60sg

1.6 billion Average number of search queries per day

200+ million Monthly active Twitter users

400+ million Average number of Tweets sent per day

“Reliability and performance are huge goals for us, and that’s why part of our core strategy involves moving to the Java Virtual Machine [JVM] runtime environment. Twitter no longer has the performance issues it previously had, and that’s in large part due to our moving to the JVM.”

- Robert Benson, Senior Director of Software Engineering, Twitter

JAVA AT ALIBABA

CHINA’S SINGLES DAY: THE WORLD’S BIGGEST ONLINE SHOPPING DAY

PEAK TRANSACTIONS PER SECOND

TOTAL SALES IN 2016

12.8

17.79

billions USD
Other names and brands may be claimed as the property of others. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit http://www.intel.com/performance.

1 - Up to 73X faster based on performance ratio of SPECjbb2005 running on Intel® Xeon® X5160 with JDK5u7 versus Intel® Xeon® E5-2697v2 with JDK6u29 and performance ratio of SPECjbb2015-multiJVM running on Intel® Xeon® E5-2697v2 with JDK8u72 versus Intel® Xeon® E5-2669v4.
Why Java?

Ease of Use

- Fast Development
- No need to fine tune and pass compiler flags

Robust

- Automatic Memory management
- Strong type checking

Popular in Enterprise

- Large user community and support via OpenJDK. Many libraries.

Write once, run everywhere

- Portable and performant
How is it performant?

Runtime Just In Time compiler

Profiling of Hot Spots

- Methods/loops

Delivers generated native code specifically for the architecture it runs on!

Use of cpuid to fingerprint system

- Uses best available instructions and technologies in underlying hardware architecture
- Code generation is tailored to different x86 ISA

X86 ISA features enabling and optimizations
OPTIMIZATIONS
## Optimizations since 2011

<table>
<thead>
<tr>
<th>JDK 7 and earlier optimizations</th>
<th>Description</th>
<th>Performance Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction decoding optimization</td>
<td>Removes length-changing prefixes and aligning branch targets.</td>
<td>Allows the Java* virtual machine (JVM) to take advantage of new Intel architecture performance features.</td>
</tr>
<tr>
<td>Out-of-order execution optimization</td>
<td>Eliminates partial flag and partial register stalls.</td>
<td>Allows the JVM to take advantage of new Intel architecture performance features.</td>
</tr>
<tr>
<td>Intrinsic function optimization</td>
<td>Optimizes the most common code paths.</td>
<td>Reduces instruction count and cycles per instruction (CPI), which increases overall performance.</td>
</tr>
<tr>
<td>Allocation pre-fetch</td>
<td>Caches additional memory ahead of the object being allocated.</td>
<td>Increases performance of critical code paths.</td>
</tr>
<tr>
<td>Large-page usage</td>
<td>Uses 2MB large memory pages for both code and data memory.</td>
<td>Increases performance of large analytics jobs.</td>
</tr>
<tr>
<td>Vectorization</td>
<td>Loads, operates on, and stores multiple array elements within a single machine instruction.</td>
<td>Increases execution performance of widely used operations, which improves overall performance.</td>
</tr>
<tr>
<td>Compressed references</td>
<td>Stores 32-bit compressed references for 64-bit pointers.</td>
<td>Reduces memory footprint and cache misses while improving cycles per instruction (CPI).</td>
</tr>
<tr>
<td>Lock optimization</td>
<td>Multi-tiered lock deferrals to avoid or delay inflation to fat locks for coincidental lock contention.</td>
<td>Increases scalability through reduction of fat locks.</td>
</tr>
<tr>
<td>HashMap, TreeMap, BigDecimal</td>
<td>Optimizes commonly used functions.</td>
<td>Improves performance by using caching and other optimizations to reduce path length and object allocation.</td>
</tr>
<tr>
<td>String object allocation optimizations</td>
<td>Optimizes code generation for methods that are frequently used.</td>
<td>Reduces intermediate object allocation for common operations, such as string comparisons.</td>
</tr>
</tbody>
</table>
## Optimizations since 2014

<table>
<thead>
<tr>
<th>JDK 8 optimizations</th>
<th>Description</th>
<th>Performance benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Advanced Encryption Standard-New Instructions (Intel® AES-NI)</td>
<td>This is hardware-based acceleration of encryption and decryption via new Java* virtual machine (JVM) intrinsic functions for AES-CTR and AES-CBC using Intel AESNI.</td>
<td>Increases the performance of data encryption for storage and network transactions. This enables wide use of encrypted data in transit.</td>
</tr>
<tr>
<td>Intel® AVX/AVX2</td>
<td>New 256-bit instruction set extensions for Intel® Streaming SIMD Extensions (Intel® SSE)</td>
<td>Improves the performance of floating-point and integer operations, as well as operations that contain array and string manipulations.</td>
</tr>
<tr>
<td>Intel® Transactional Synchronization Extensions (Intel® TSX)</td>
<td>Support for Java synchronization through an Intel TSX JVM option.</td>
<td>Provides the benefit of fine-grain lock performance with coarse-grain locks, when you use the -XX:+UseRTMLocking JVM option.</td>
</tr>
<tr>
<td>BMI1 Instructions</td>
<td>The Just-in-time compiler recognizes the patterns for ANDN (X &amp; !Y), BLSR (X &amp; (X - 1)), BLSMSK (X ^ (X - 1)) and BLSI (X &amp; ~X) instructions. The JIT also generates an intrinsic function for Integer.numberOfLeadingZeros using LZCNT instructions. It also generates an intrinsic Integer.numberOfTrailingZeros function using TZCNT instructions.</td>
<td>Improves the performance of bit manipulation code sequences that are used in Java applications.</td>
</tr>
<tr>
<td>Fast cyclic redundancy check (CRC) computation</td>
<td>New CRC generation algorithm using the carry-less multiplication instruction PCLMULQDQ</td>
<td>Increases file checksum and compression/decompression checksum performance, which increases network and file system performance.</td>
</tr>
<tr>
<td>ISO 8859-1 encoding</td>
<td>Uses Intel AVX/Intel AVX2 to increase character encoding to 16 or 32 characters at a time, instead of character by character.</td>
<td>Improves the performance of applications that use international string encoding.</td>
</tr>
<tr>
<td>BigInteger multiply</td>
<td>Uses 64-bit integer multiply instructions with 128-bit results, in order to perform BigInteger Multiply. This optimization also uses ADCX/Aadox instructions where they are available.</td>
<td>Improves the performance of the BigInteger Multiply operation, which is used in RSA cryptography and multi-precision arithmetic.</td>
</tr>
<tr>
<td>1GB large pages</td>
<td>Supports 1GB large memory pages in the JVM runtime for both code and data memory.</td>
<td>Increases the performance of large analytics jobs.</td>
</tr>
</tbody>
</table>
## Optimizations since 2017

<table>
<thead>
<tr>
<th>JDK 9 optimizations</th>
<th>Description</th>
<th>Performance benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® AVX512</td>
<td>Just-in-time (JIT) compiler auto-vectorization is extended to 512 bit in order to generate AVX-512 code. Tail processing is also improved and the reduction operation is supported in a vector loop. The array and string intrinsic methods are enhanced to use AVX-512 bit instructions.</td>
<td>When the Java® virtual machine (JVM) is invoked with the <code>-XX:UseAVX=3</code> option, this optimization improves the performance of floating-point and integer operations, as well as operations that contain array and string manipulation.</td>
</tr>
<tr>
<td>Math Libraries</td>
<td>Optimized trigonometric and transcendental <code>java.lang.Math</code> methods include: <code>exp</code>, <code>pow</code>, <code>log</code>, <code>log10</code>, <code>sin</code>, <code>cos</code>, and <code>tan</code></td>
<td>Improves the performance of applications for Big Data, machine learning, financial option pricing, and high performance computing (HPC).</td>
</tr>
<tr>
<td>Compact Strings</td>
<td>The compact strings optimization uses byte array instead of char array to store characters with an encoding byte field. It changes the internal representation of String class, while preserving compatibility. The characters of String are encoded as either UTF-16 (two bytes per character) or ISO-8859-1/Latin-1 (one byte per character).</td>
<td>Improves space efficiency. Improves space efficiency of the String class and related classes. String methods are highly optimized using Single Instruction Multiple Data (SIMD) instructions where possible. This optimization allows JIT compiled code to perform up to twice the operations per iteration as unoptimized Java applications.</td>
</tr>
<tr>
<td>Cryptography acceleration</td>
<td>Java cryptography architecture (JCA) defines a provider framework, with Oracle SunJCE® as the default provider. This cryptography acceleration optimization provides cryptography stubs for the SunJCE provider, using AESNI, SHA-NI, PCLMULQDQ, ADCX/ADOX and AVX/AVX2; in order to accelerate AES-CTR, AES-CBC, AES-GCM, SHA1, SHA256 and SHA512.</td>
<td>Increases data encryption performance for storage and network transactions. This enables wide use of encrypted data in transit.</td>
</tr>
<tr>
<td>Compression acceleration</td>
<td>JDK 9 uses system Zlib instead of a statically linked (bundled) Zlib. This allows applications to override the system Zlib with a hardware/software accelerated library through <code>LD_PRELOAD</code> and <code>LD_LIBRARY_PATH</code> environment variables*.</td>
<td>Provides a compression acceleration feature for hardware and software, for applications that use compression extensively. Applications that typically use extensive compression include genomics, big data, and enterprise applications.</td>
</tr>
</tbody>
</table>
## JDK 9 optimizations

<table>
<thead>
<tr>
<th>Description</th>
<th>Performance benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC32C support</td>
<td>Increases the performances of checksum calculations for big data and other applications, through the use of the new API.</td>
</tr>
<tr>
<td>Lexicographic array compare</td>
<td>Increases the performance of big data applications that use arrays as keys.</td>
</tr>
<tr>
<td>Floating point multiply Accumulate (FMA)</td>
<td>Increases the performance of HPC, artificial intelligence, and machine-learning applications; through use of the new API.</td>
</tr>
<tr>
<td>Socket Reuse</td>
<td>Improves the scalability and parallelism of network traffic handling. This provides better throughput and lower latency through the use of the new SO_REUSEPORT option in java.net.StandardSocketOptions.</td>
</tr>
</tbody>
</table>
Capabilities added in JDK9

AVX512 support
- auto-vectorization
- Intrinsics for array and string operations

Auto-vectorization enhancements
- reduction support
- super unrolling

Vector FMA
- Generates vector fma instruction by using auto-vectorization
Performance Setup

Platform:
Intel® Xeon® Platinum 8180 @ 2.5GHz, 384 GB RAM

Software:
OS: Red Hat Enterprise Linux Server release 7.2
JDK 8: Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
JDK 9: Java(TM) SE Runtime Environment (build 9+181)
AVX512 FP Performance

FP ARITHMETIC

- 32ByteVector
- 64ByteVector
- 64ByteVectorAligned

FloatAdd  |  FloatMul  |  FloadSub  |  DoubleAdd  |  DoubleMul  |  DoubleSub

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Optimized Math Libraries

- Optimized Trigonometric & Transcendental LIBM methods
- Used in Big Data machine learning, Financial option pricing and HPC applications

**MATH PERFORMANCE**

<table>
<thead>
<tr>
<th>Function</th>
<th>JDK 8</th>
<th>JDK 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cos</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tan</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FSI PERFORMANCE**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>JDK 8</th>
<th>JDK 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monte Carlo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Box Muller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black Scholes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VECTOR API
```c
static void mystery(float[] a, int a_offset, float[] b, int b_offset, float alpha) {
    for (int i = 0; i + a_offset < a.length && i + b_offset < b.length; i++) {
        b[i + b_offset] += alpha * a[i + a_offset];
    }
}
```

```c
static void mysteryUnrolled(float[] a, int a_offset, float[] b, int b_offset, float alpha) {
    for (int i = 0; i + a_offset < a.length && i + b_offset < b.length; i += 4) {
        b[i + b_offset + 0] += alpha * a[i + a_offset + 0];
        b[i + b_offset + 1] += alpha * a[i + a_offset + 1];
        b[i + b_offset + 2] += alpha * a[i + a_offset + 2];
        b[i + b_offset + 3] += alpha * a[i + a_offset + 3];
    }
}
```
static void saxpyUnrolled(float[] a, int a_offset, float[] b, int b_offset, float alpha) {
    for (int i = 0; i + a_offset < a.length && i + b_offset < b.length; i += 4) {
        b[i + b_offset + 0] += alpha * a[i + a_offset + 0];
        b[i + b_offset + 1] += alpha * a[i + a_offset + 1];
        b[i + b_offset + 2] += alpha * a[i + a_offset + 2];
        b[i + b_offset + 3] += alpha * a[i + a_offset + 3];
    }
}

static void saxpySimd(float[] a, int a_offset, float[] b, int b_offset, float alpha) {
    for (int i = 0; i + a_offset < a.length && i + b_offset < b.length; i += 4) {
        b[i + b_offset + 0] b[i + b_offset + 0] alpha a[i + a_offset + 0];
        b[i + b_offset + 1] b[i + b_offset + 1] alpha a[i + a_offset + 1];
        b[i + b_offset + 2] b[i + b_offset + 2] alpha a[i + a_offset + 2];
        b[i + b_offset + 3] b[i + b_offset + 3] alpha a[i + a_offset + 3];
    }
}
```java
static void saxpySimd(float[] a, int a_offset, float[] b, int b_offset, float alpha) {
    for (int i = 0; i + a_offset < a.length && i + b_offset < b.length; i += 4) {
        b[i + b_offset + 0] += alpha * a[i + a_offset + 0];
        b[i + b_offset + 1] += alpha * a[i + a_offset + 1];
        b[i + b_offset + 2] += alpha * a[i + a_offset + 2];
        b[i + b_offset + 3] += alpha * a[i + a_offset + 3];
    }
}
```
```java
static void saxpySimd(float[] a, int a_offset, float[] b, int b_offset, float alpha) {
    for (int i = 0; i + a_offset < a.length && i + b_offset < b.length; i += 4) {
        b[i + b_offset + 0] = b[i + b_offset + 0] - alpha * a[i + a_offset + 0];
        b[i + b_offset + 1] = b[i + b_offset + 1] - alpha * a[i + a_offset + 1];
        b[i + b_offset + 2] = b[i + b_offset + 2] - alpha * a[i + a_offset + 2];
        b[i + b_offset + 3] = b[i + b_offset + 3] - alpha * a[i + a_offset + 3];
    }
}
```

```
int i = 0;
FloatVector<S> alphaVec = fspec.broadcast(alpha);
for (; (i + a_offset + fspec.length()) < a.length && (i + b_offset + fspec.length()) < b.length;
    i += fspec.length()) {
    FloatVector<S> bv = fspec.fromArray(b, i + b_offset);
    FloatVector<S> av = fspec.fromArray(a, i + a_offset);
    bv.add(av.mul(alphaVec)).intoArray(b, i + b_offset);
}
```
int i = 0;
FloatVector<S> alphaVec = fspec.broadcast(alpha);
for (; (i + a_offset + fspec.length()) < a.length && (i + b_offset + fspec.length()) < b.length;
    i += fspec.length()) {
    FloatVector<S> bv = fspec.fromArray(b, i + b_offset);
    FloatVector<S> av = fspec.fromArray(a, i + a_offset);
    bv.add(av.mul(alphaVec)).intoArray(b, i + b_offset);
}

vmovdqu 0x10(%r9,%rsi,4),%ymm1
mov %r8d,%edi
add 0x8(%rsp),%edi
vmovdqu 0x10(%rbx,%rdi,4),%ymm2
vmulps %ymm0,%ymm2,%ymm2
vaddps %ymm2,%ymm1,%ymm1
vmovdqu %ymm1,0x10(%r9,%rsi,4)
mov %r8d,%r11d
add $0x8,%r11d
add $0x10,%edi
cmp %r10d,%edi
jl 0x00007faf5bab7e22
**Vector API performance improvements**

- BLAS I, II algorithms are used in Machine Learning libraries for linear models (logistic and linear regression), collaborative filtering etc. (for example, Spark ML)
- BLAS III routines like GEMM are applicable to neural network and deep learning algorithms.
- Up to 4.5X performance speed-up across BLAS I/II and III algorithms*

---

*Open JDK Project Panama source build 09182017. Java Hotspot 64-bit Server VM (mixed mode). OS version: Cent OS 7.3 64-bit Intel® Xeon® Platinum 8180 processor (using 512 byte and 1024 byte chunk of floating point data).
JVM options: -XX:+UnlockDiagnosticVMOptions -XX:CheckIntrinsics -XX:TypeProfileLevel=121 -XX:+UseVectorApiIntrinsics
Image Processing Application: Sepia Filter

Filter applies Sepia toning to an image.

```c
for (int i = 0; i < width * height; i++) {
    magnitudeR[i] = 0.393f * redFlat[i] + 0.769f * greenFlat[i] + 0.189f * blueFlat[i];
    magnitudeG[i] = 0.349f * redFlat[i] + 0.686f * greenFlat[i] + 0.168f * blueFlat[i];
    magnitudeB[i] = 0.272f * redFlat[i] + 0.534f * greenFlat[i] + 0.131f * blueFlat[i];
    if(255.0f < magnitudeR[i]) magnitudeR[i] = 255.0f;
    if(255.0f < magnitudeG[i]) magnitudeG[i] = 255.0f;
    if(255.0f < magnitudeB[i]) magnitudeB[i] = 255.0f;
}

for (int i = 0; i < width * height; i += 8) {
    FloatVector<Shapes.S256Bit> c1 = fspec.broadcast(0.393f);
    FloatVector<Shapes.S256Bit> c2 = fspec.broadcast(0.769f);
    FloatVector<Shapes.S256Bit> c3 = fspec.broadcast(0.189f);
    FloatVector<Shapes.S256Bit> c4 = fspec.broadcast(0.349f);
    FloatVector<Shapes.S256Bit> c5 = fspec.broadcast(0.686f);
    FloatVector<Shapes.S256Bit> c6 = fspec.broadcast(0.168f);
    FloatVector<Shapes.S256Bit> c7 = fspec.broadcast(0.272f);
    FloatVector<Shapes.S256Bit> c8 = fspec.broadcast(0.534f);
    FloatVector<Shapes.S256Bit> c9 = fspec.broadcast(0.131f);
    FloatVector<Shapes.S256Bit> c10 = fspec.broadcast(255f);
    FloatVector<Shapes.S256Bit> redVec = fspec.fromArray(redFlat, i);
    FloatVector<Shapes.S256Bit> greenVec = fspec.fromArray(greenFlat, i);
    FloatVector<Shapes.S256Bit> blueVec = fspec.fromArray(blueFlat, i);
    FloatVector<Shapes.S256Bit> res1 = redVec.mul(c1).add(greenVec.mul(c2)).add(blueVec.mul(c3));
    FloatVector<Shapes.S256Bit> res2 = redVec.mul(c4).add(greenVec.mul(c5)).add(blueVec.mul(c6));
    FloatVector<Shapes.S256Bit> res3 = redVec.mul(c7).add(greenVec.mul(c8)).add(blueVec.mul(c9));
    res1.blend(c10, res1.lessThan(c10)).intoArray(magnitudeR[i]);
    res2.blend(c10, res2.lessThan(c10)).intoArray(magnitudeG[i]);
    res3.blend(c10, res3.lessThan(c10)).intoArray(magnitudeB[i]);
}
```
SEPIA Filter

```
vmulps ymm3, ymm2, ymm1
vmulps ymm1, ymm4, ymm13
vmulps ymm0, ymm5, ymm4
vaddps ymm0, ymm3, ymm0
vaddps ymm0, ymm0, ymm1
vcmps ps ymm1, ymm0, ymm12, 1h
vblendvps ymm0, ymm12, ymm0, ymm1
vmovdqu ymmword ptr [r12 + r11*8 + 10h], ymm0
```
Parallel Streams + Vector API

```java
IntStream.range(0, SIZE / ISPEC.length()).parallel().forEach(i -> {
    IntVector<Shapes.S256Bit> av = ISPEC.fromArray(ia, i * ISPEC.length());
    IntVector<Shapes.S256Bit> bv = ISPEC.fromArray(ib, i * ISPEC.length());
    av.add(bv).intoArray(ic, i * ISPEC.length());
});
```
More Resources

Accelerating Performance for server-side Java Applications:

https://software.intel.com/sites/default/files/managed/19/ae/JavaPerformanceWP.d46.pdf

Vector API Developer Program

Architecture of Hotspot