Expanding the High Performance Embedded Computing Tool Chest - Mixing Java and C

Nazario Irizarry, Jr.

MITRE Corporation
202 Burlington Rd.
Bedford, MA, 01730

Presented at IEEE High Performance Extreme Computing Conference (HPEC `13), Waltham, MA
©2013-The MITRE Corporation. All rights reserved.
Approved for Public Release; Distribution Unlimited. 13-1117
Outline

- Goals & motivation
- Performance findings
  - CPU experiments
  - GPU experiment
- Helpful frameworks
C is fast but development is meticulous and time consuming
Java is not as fast but development is more expeditious
Java frameworks are very capable
  – Logging, dynamic scheduling, CPU load balancing, elastic grid resizing, automatic data serialization, automatic work failover, etc.
Want to build hybrid apps that leverage the strengths of both
  – Need to understand when Java 7 will and won’t perform
  – Need to understand the cost of interactions between Java and C
  – Need to understand how effectively Java can utilize GPUs
Isn’t Java Slow?

- Java startup is slow
  - Disk I/O
  - Class lookup & validation
  - Static initialization

- Java Hotspot optimizes during runtime
  - Dynamically switch from interpreted to compiled
  - Aggressive method in-lining
Effect of Java Warming on Execution Time

- Iteration 1 (not plotted) = 14 us.
Test Environment

- **Hardware**
  - Intel Core 2 Quad Q9650 @ 2.99GHz
  - 4 GB RAM
  - Four core, not hyper-threaded, SSE4.1

- **Linux 2.6.32**

- **GCC 4.4.6**
  - -O3 optimization

- **Oracle Java 7**
  - Standard edition, 1.7.0_02
Test: Bit-Twiddling

- Workload consists of generating multiple linear feedback shift register sequences

![Execution Time - LFSR Sequence Generation](chart.png)
Test: Structure Building

- Red-black tree key-value insertion and retrieval
- Exercises
  - Memory allocation
  - Integer key comparison
  - Conditional code execution
  - Reference manipulation
Red-Black Tree Data Insertion Time

C (incl free())

Insert (4 bytes) | Insert (512 bytes) | Insert (2048 bytes)
51 | 89 | 93
62 | 121 | 108
36 | 110 | 440

C
Java
Red-Black Tree Data Retrieval Time

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get (4 bytes)</td>
<td>36 msec.</td>
<td>38 msec.</td>
</tr>
<tr>
<td>Get (512 bytes)</td>
<td>53 msec.</td>
<td>56 msec.</td>
</tr>
<tr>
<td>Get (2048 bytes)</td>
<td>48 msec.</td>
<td>39 msec.</td>
</tr>
</tbody>
</table>

This bar chart shows the retrieval time in milliseconds for different data sizes in C and Java.
Test: Simple Iteration and Math

- **Matrix multiply**
  - $10 \leq N \leq 90$

Java

![Graph](image)

- Speed (GFLOPS) vs. Matrix Rank
- C

NxN Matrix Multiply
Test: Intricate Iteration and Math

- Fast Fourier Transform
  - Non-optimized radix 2 butterflies

![FFT Speeds Graph]

- N (Number of points = 2^N)
- Speed (GFLOPS)

- C
- Java
Test Design: Impact of Garbage Collector

Test 1
- 0 GC actions/sec
  - Bit twiddling test
  - Preallocated buffers

Test 2
- 7.5 GC actions/sec.
  - Red black tree test
  - Lots of memory allocation

- 2 scenarios for each
  - 2 compute threads on 2 cores
  - 2 compute thread on 3 cores

Intentionally poorly tuned
Test Results: Impact of Garbage Collector

Test 1
- 0 GC actions/sec

No Impact

Test 2
- 7.5 GC actions/sec.

15% Execution Time Impact

- Key design practices
  - Preallocate buffers and structures
  - Be smart about String concatenation
  - Keep the GC quiet
Investigation: Accessing Native Memory

- Java data is “fenced”
  - Passing reference types (arrays and objects) to C incurs overhead
- Java Native Interface (JNI) is fastest but most tedious
- Alternatives: Java Native Access, BridJ, SWIG, HawtJNI
Comparing 3 Bridging Alternatives

<table>
<thead>
<tr>
<th>Call Arguments</th>
<th>JNI</th>
<th>SWIG</th>
<th>BridJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 int args</td>
<td>11 ns.</td>
<td>17 ns.</td>
<td>200 ns.</td>
</tr>
<tr>
<td>2 read-only Java arrays (double, length 200)</td>
<td>13 ms.</td>
<td>190 ms.</td>
<td>110 ms.</td>
</tr>
<tr>
<td>Pointers to 2 native arrays reference by Java proxies</td>
<td>N/A</td>
<td>58 ns.</td>
<td>10 ns.</td>
</tr>
</tbody>
</table>

- BridJ and SWIG provide good performance
- BridJ is easier to use
  - Uses .h files directly
  - Includes ability to manage C memory from Java
- See paper and final MITRE report for more details and test cases
Test: Utilizing GPUs (1 of 2)

- FFT’s of varying sizes invoked with both synchronous and asynchronous kernel invocation

- Apple’s optimized FFT’s for OpenCL was ported to Java
  - BridJ for native data management
  - JavaCL for OpenCL access
  - Both are subprojects of “nativelibs4java” at Sourceforge
- Java kept the GPU’s as busy as C did
  - Kernel invocation from Java was 2 ms longer than from C (9 ms)
Summary

- **Java 7 (w/ Hotspot) can perform well after warming**
  - Code loops with regular indexes do well
  - Structure allocation and manipulation of moderate sized structures
  - Boolean integer operations
  - Conditional logic

- **Well designed code can incur negligible GC impact**

- **Hybrid performance will depend on:**
  - Granularity of interactions between the Java and C code
  - Whether data is created natively or in Java

- **Frameworks exist to help glue the hybrid Java/C app**

- **GPUs can be effectively managed and utilized if the data is native**

- **A detailed final research report will be available**
  - Parallelism; CPython, Jython, PyPy, Scala; Java grid-computing frameworks
Frameworks and Links

- **BridJ** – Java to C binding
- **JavaCL** – Java tier over OpenCL
- **SWIG** – Multiple languages to C binding
  - [http://www.swig.org/](http://www.swig.org/)
- **CShimToJava** – C to Java binding
  - [http://cshimtojava.sourceforge.net](http://cshimtojava.sourceforge.net)
- **Benchmarks used in this study**
  - [http://jcompbmarks.sourceforge.net](http://jcompbmarks.sourceforge.net)