THE WORKPAPER: SPEED
MEASUREMENTS OF CONVOLUTION ALGORITHM ON X86 PLATFORM

Project supervisors:
Ole Olsen
Bent Thomsen

Members (group 03gr841):
Vytautas Dusevičius
Agnius Liutkevičius
Rimantas Žukaitis

22 / 04 / 2003
### Table of contents:

1. Document objectives ................................................................. 2
2. General information ................................................................. 2
3. Compilers ................................................................................. 2
4. Results .................................................................................... 3
5. Bibliography ............................................................................ 7
1. Document objectives

The objective of this document is to present the results from the case study: Convolution algorithm. In this document only the original (not optimized) version of the algorithm is described and only time measurements were performed. The measurements were also performed on x86 platform, not DSP.

2. General information

Convolution is one of the most frequently used operations in DSP. For example, it is the basic operation in digital filtering. Given two finite and causal sequences, \( x(n) \) and \( h(n) \), of lengths \( N_1 \) and \( N_2 \), respectively, their convolution is defined as

\[
y(n) = h(n) \otimes x(n) = \sum_{k=-\infty}^{\infty} h(k)x(n - k) \]

where \( M = N_1 + N_2 - 1 \) [1].

The term convolution describes, amongst other things, how the input to a system interacts with the system to produce the output. Generally the system output will be a delayed and attenuated or amplified version of the input [2]. It is particularly useful to consider the output from the system owing to an impulse input. If the input consists of a continuous sequence of impulses, the summations may be replaced by integrals

\[
y(t) = \int_{-\infty}^{\infty} x(\lambda)h(t - \lambda) d\lambda
\]

Properties of convolution:

- Commutative law
  \[ x_1(t) \otimes x_2(t) = x_2(t) \otimes x_1(t) \]
- Distributive law
  \[ x_1(t) \otimes [x_2(t) + x_3(t)] = x_1(t) \otimes x_2(t) + x_1(t) \otimes x_3(t) \]
- Associative law
  \[ x_1(t) \otimes [x_2(t) \otimes x_3(t)] = [x_1(t) \otimes x_2(t)] \otimes x_3(t) \]

There are methods for calculating fast convolution [3] (for example using Fourier transform), but these methods will not be analyzed in this document.

3. Compilers
For compiling C programs we used “Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86”. Options used for optimizations:

- /Od – disable optimizations
- /O2 – maximize speed
- /Te – compile file as .c

For compiling and executing Java programs we used “Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1_01-b01)”. The listed options were used when executing the program:

- -client - to select the "client" VM. The Java HotSpot Client VM is the default virtual machine of the Java 2 SDK and Java 2 Runtime Environment. As its name implies, it is tuned for best performance when running applications in a client environment by reducing application start-up time and memory footprint [4].
- -server - to select the "server" VM. The Java HotSpot Server VM is designed for maximum program execution speed for applications running in a server environment [4].
- -Xint - operate in interpreted-only mode. Compilation to native code is disabled, and all bytecodes are executed by the interpreter. The performance benefits offered by the Java HotSpot VMs' adaptive compiler will not be present in this mode [5].

### 4. Results

In this case only the original implementation of the convolution algorithm was tested. We measured the time of execution. For higher accuracy of the measurements, the method / function was called multiple times (from 100 to 1.000.000, depending on input array size) and then the average execution time value was calculated. The average execution times (in milliseconds) are presented in table 1 and figures 1,2,3. The abbreviations used in tables and figures:

- C – the execution times of the c program with disabled optimizations (/Od option);
- C-O2 - the execution times of the c program, when compiler’s options are set to maximize speed (/O2 option);
- J-C-X - the execution times of the java program, using “client” VM with disabled Just-in-Time compiler (-client and -Xint options);
- J-C - the execution times of the java program, using “client” VM with enabled Just-in-Time compiler (-client option);
- J-S - the execution times of the java program, using “server” VM with enabled Just-in-Time compiler (-server option);
- N – the size of arrays, which were used for calculating convolution.
Table 1. The average times of execution of convolution algorithm on x86 platform:

<table>
<thead>
<tr>
<th>N</th>
<th>C</th>
<th>C-O2</th>
<th>J-C-X</th>
<th>J-C</th>
<th>J-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.000961</td>
<td>0.000510</td>
<td>0.009513</td>
<td>0.000841</td>
<td>0.000561</td>
</tr>
<tr>
<td>8</td>
<td>0.003074</td>
<td>0.001642</td>
<td>0.034860</td>
<td>0.002954</td>
<td>0.001842</td>
</tr>
<tr>
<td>16</td>
<td>0.009924</td>
<td>0.004947</td>
<td>0.134123</td>
<td>0.010044</td>
<td>0.005938</td>
</tr>
<tr>
<td>32</td>
<td>0.046000</td>
<td>0.025000</td>
<td>0.522800</td>
<td>0.036100</td>
<td>0.022000</td>
</tr>
<tr>
<td>64</td>
<td>0.145200</td>
<td>0.071100</td>
<td>2.075000</td>
<td>0.141200</td>
<td>0.095200</td>
</tr>
<tr>
<td>128</td>
<td>0.556800</td>
<td>0.247300</td>
<td>8.242700</td>
<td>0.544800</td>
<td>0.345500</td>
</tr>
<tr>
<td>256</td>
<td>2.102000</td>
<td>0.931300</td>
<td>32.867500</td>
<td>2.144100</td>
<td>1.216800</td>
</tr>
<tr>
<td>512</td>
<td>8.294900</td>
<td>3.583100</td>
<td>131.694200</td>
<td>8.555300</td>
<td>5.014200</td>
</tr>
<tr>
<td>1024</td>
<td>34.040000</td>
<td>15.120000</td>
<td>525.350000</td>
<td>33.950000</td>
<td>70.900000</td>
</tr>
<tr>
<td>2048</td>
<td>141.100000</td>
<td>58.880000</td>
<td>2098.010000</td>
<td>135.700000</td>
<td>136.490000</td>
</tr>
<tr>
<td>4096</td>
<td>573.920000</td>
<td>234.330000</td>
<td>8393.100000</td>
<td>543.280000</td>
<td>373.640000</td>
</tr>
</tbody>
</table>

Figure 1. The graphical representation of average execution times:
**Figure 2.** The graphical representation of average execution times with small N:

![Graph showing average execution times with different input array sizes for C, C-O2, J-C, and J-S.](image)

**Figure 3.** The comparison of Java Client VM, with enabled and disabled JIT:

![Graph showing comparison of Java Client VM with enabled and disabled JIT.](image)
In table 2 the speed comparisons between C and Java with different options are presented. For example, value 1,40 in column J-S / C-O2 and row N=512 means, that program execution with Java Server VM is 1,4 times slower than execution with C (optimized for speed), when the size of input arrays is 512. In the last line of the table 2 the average values are presented.

Table 2. The execution rates between C and Java, with different options:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.88</td>
<td>1.50</td>
<td>11.31</td>
<td>0.88</td>
<td>1.10</td>
</tr>
<tr>
<td>8</td>
<td>1.87</td>
<td>1.60</td>
<td>11.80</td>
<td>0.96</td>
<td>1.12</td>
</tr>
<tr>
<td>16</td>
<td>2.01</td>
<td>1.69</td>
<td>13.35</td>
<td>1.01</td>
<td>1.20</td>
</tr>
<tr>
<td>32</td>
<td>1.84</td>
<td>1.64</td>
<td>14.48</td>
<td>0.78</td>
<td>0.88</td>
</tr>
<tr>
<td>64</td>
<td>2.04</td>
<td>1.48</td>
<td>14.70</td>
<td>0.97</td>
<td>1.34</td>
</tr>
<tr>
<td>128</td>
<td>2.25</td>
<td>1.58</td>
<td>15.13</td>
<td>0.98</td>
<td>1.40</td>
</tr>
<tr>
<td>256</td>
<td>2.26</td>
<td>1.76</td>
<td>15.33</td>
<td>1.02</td>
<td>1.31</td>
</tr>
<tr>
<td>512</td>
<td>2.32</td>
<td>1.71</td>
<td>15.39</td>
<td>1.03</td>
<td>1.40</td>
</tr>
<tr>
<td>1024</td>
<td>2.25</td>
<td>0.48</td>
<td>15.47</td>
<td>1.00</td>
<td>4.69</td>
</tr>
<tr>
<td>2048</td>
<td>2.40</td>
<td>0.99</td>
<td>15.46</td>
<td>0.96</td>
<td>2.32</td>
</tr>
<tr>
<td>4096</td>
<td>2.45</td>
<td>1.45</td>
<td>15.45</td>
<td>0.95</td>
<td>1.59</td>
</tr>
<tr>
<td>Avg.</td>
<td>2.14</td>
<td>1.44</td>
<td>14.35</td>
<td>0.96</td>
<td>1.67</td>
</tr>
</tbody>
</table>

After the analysis of the results presented in tables 1 and 2, the following conclusions can be made:

- Java is very slow, when Just-in-Time compiler is disabled. In this case Java without JIT compiler is about 14 times slower, compared with Java with JIT compiler enabled. So JITs are one of the most important features in Java VM.
- C compiler can increase the execution time of the C program more than twice.
- Java Server VM may increase the execution time of the Java program (compared with the Java Client VM) about 1.6 times. The interesting and hard-to-explain results were obtained, when N is 1024 (particularly) and 2048. When N is equal to 1024, Java Server VM is more than twice slower than Java Client VM. Maybe this is a case where Java Server VM somehow changes its behavior and this is the reason of slow-down. This needs further investigations.
- Java Client VM execution speed is almost the same as the execution speed of C program with disabled compiler optimizations. These results are quite interesting, because in this case both (C and Java) implementations can be interpreted as processor-independent. This means, that Java loses some speed because of bytecode interpretation, but can gain some speed because of some more optimal internal implementations (at the level of the VM) and, of course, JIT compiler.
- Java Server VM is only 30% slower (except those cases when N is equal to 1024 and 2048) that C program, that is optimized for speed, using compiler’s option /O2. These results are quite good, having in mind the other advantages of Java language (code writing speed, code portability, understandability, security etc.). But it’s very likely that the results will be worse, when executing these programs on DSP platforms.
5. Bibliography

4. Java Virtual Machines: http://java.sun.com/j2se/1.4.1/docs/guide/vm/
5. Java Application Launcher, options: http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/java.html#options