Comparing Android Runtime with native: Fast Fourier Transform on Android

André Danielsson

anddani@kth.se

Royal Institute of Technology
Computer Science and Communication

May 4, 2017

Outline

1. Introduction
   - Purpose of Thesis
   - Research Question
2. Background
   - Android Development
   - Java Native Interface (JNI)
   - Discrete Fourier Transform (DFT)
   - Fast Fourier Transform (FFT)
   - Related Work
3. Method
   - Experiments
   - Measurements
   - Implementation
4. Results and Discussion
   - JNI
   - Libraries
   - NEON
   - Garbage Collection
   - `float` vs `double`
5. Conclusions

1. Introduction
   - Purpose of Thesis
   - Research Question
2. Background
   - Android Development
   - Java Native Interface (JNI)
   - Discrete Fourier Transform (DFT)
   - Fast Fourier Transform (FFT)
   - Related Work
3. Method
   - Experiments
   - Measurements
   - Implementation
4. Results and Discussion
   - JNI
   - Libraries
   - NEON
   - Garbage Collection
   - `float` vs `double`
5. Conclusions

Purpose of Thesis

- Why is this work important?

Purpose of Thesis

- Why is this work important?
- Where can it be used?

Purpose of Thesis

- Why is this work important?
- Where can it be used?
- Who will benefit from it?

Research Question

Is there a significant performance difference between implementations of a Fast Fourier Transform (FFT) in native code, compiled by Clang, and Dalvik bytecode, compiled by Android Runtime, on Android?

1 Introduction
- Purpose of Thesis
- Research Question

2 Background
- Android Development
- Java Native Interface (JNI)
- Discrete Fourier Transform (DFT)
- Fast Fourier Transform (FFT)
- Related Work

3 Method
- Experiments
- Measurements
- Implementation

4 Results and Discussion
- JNI
- Libraries
- NEON
- Garbage Collection
- `float` vs `double`

5 Conclusions

Android SDK

- Framework for developing applications
- Java
- Dalvik Virtual Machine

Android Runtime

- Replaced Dalvik Virtual Machine
- Ahead-Of-Time instead of Just-In-Time
- This allows for heavier optimizations

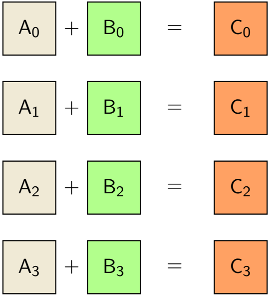
Android NDK

- Tools for building native applications
- Uses Clang and LLVM (as of 2017)
- Uses JNI to communicate between Java and Native

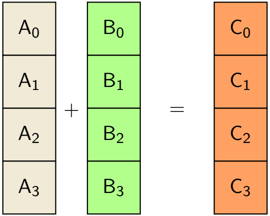
Java Native Interface (JNI)

- Bridge between JVM and binaries
- Run code compiled for a specific architecture from Java
- JVM communication

Vectorization



(a) Four separate instructions



(b) One instruction with SIMD

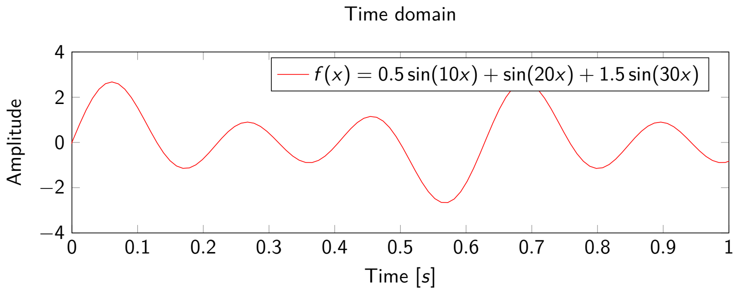
- SSE
- NEON

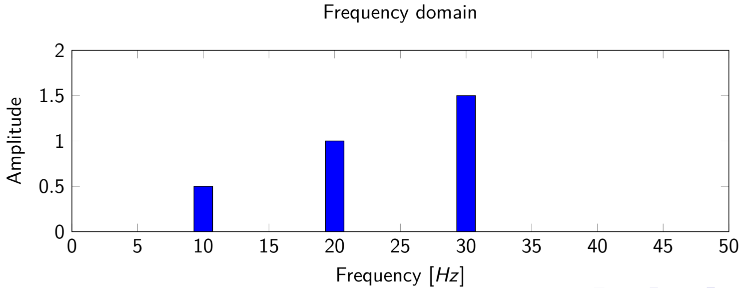
Discrete Fourier Transform (DFT)

- DFT: *Time* $\Rightarrow$ *Frequency*
- Decomposition of a signal
- Example uses:
  - Audio visualization
  - Speech recognition
  - Compression
  - Polynomial multiplication

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, 2, \ldots, N-1$$

Discrete Fourier Transform (DFT)





Fast Fourier Transform (FFT)

- Naive DFT, $O(N^2)$
- Cooley-Tukey FFT[1], $O(N \log N)$
- Trigonometric constants (twiddle factors)

[1] B. J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation Complex Fourier Series", pp. 297–301, 1964.

Related Work

- S. Lee and J. W. Jeon, "Evaluating Performance of Android Platform Using Native C for Embedded Systems", *International Conference on Control, Automation and Systems*, pp. 1160–1163, 2010.

Related Work

- S. Lee and J. W. Jeon, "Evaluating Performance of Android Platform Using Native C for Embedded Systems", *International Conference on Control, Automation and Systems*, pp. 1160–1163, 2010.
- A. D. D. C. Jr, M. Rosan, and M. Queiroz, "FFT benchmark on Android devices: Java versus JNI", pp. 4–7, 2013.

Related Work

- S. Lee and J. W. Jeon, “Evaluating Performance of Android Platform Using Native C for Embedded Systems”, *International Conference on Control, Automation and Systems*, pp. 1160–1163, 2010.
- A. D. D. C. Jr, M. Rosan, and M. Queiroz, “FFT benchmark on Android devices: Java versus JNI”, pp. 4–7, 2013.
- X. Chen and Z. Zong, “Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation”, *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)*, pp. 485–492, 2016.

1 Introduction
- Purpose of Thesis
- Research Question

2 Background
- Android Development
- Java Native Interface (JNI)
- Discrete Fourier Transform (DFT)
- Fast Fourier Transform (FFT)
- Related Work

3 Method
- Experiments
- Measurements
- Implementation

4 Results and Discussion
- JNI
- Libraries
- NEON
- Garbage Collection
- `float` vs `double`

5 Conclusions

Experiments

1. Cost of using JNI
2. Comparison of smaller FFT libraries
3. Native optimization with NEON
4. Using `float` and `double` data types

Measurements

- Nexus 6P used in all tests
- Time was measured in Java using `SystemClock.elapsedRealtimeNanos()`
- Data sizes varied between $2^4 - 2^{18}$
- 100 executions for each test, 95% confidence interval

Implementation

- **Java**
  - Princeton Recursive
  - Princeton Iterative
  - Columbia Iterative

Implementation

- **Java**
  - Princeton Recursive
  - Princeton Iterative
  - Columbia Iterative
- **C++**
  - KISS (Keep It Simple Stupid) FFT
  - SSE Recursive FFT
  - SSE Iterative FFT

Implementation

- Benchmark application
- Java algorithms translated to C++
- Separate thread, one algorithm at a time
- Time measurements executed on a release build
- Memory measurements executed with an attached debugger

1 Introduction
- Purpose of Thesis
- Research Question

2 Background
- Android Development
- Java Native Interface (JNI)
- Discrete Fourier Transform (DFT)
- Fast Fourier Transform (FFT)
- Related Work

3 Method
- Experiments
- Measurements
- Implementation

4 Results and Discussion
- JNI
- Libraries
- NEON
- Garbage Collection
- `float` vs `double`

5 Conclusions

JNI (Time µs)

Block size	No params	Vector	Convert	Columbia
16	1.7922 ± 0.1392	1.9333 ± 0.1223	2.6052 ± 0.1004	4.1058 ± 0.3042
32	1.6983 ± 0.0220	2.8130 ± 1.7924	2.6006 ± 0.0370	3.9109 ± 0.0535
64	1.6755 ± 0.0149	1.6344 ± 0.1809	2.6630 ± 0.0425	3.9296 ± 0.0566
128	1.9604 ± 0.4978	1.2349 ± 0.1262	1.9375 ± 0.0843	3.0823 ± 0.0892
256	1.7292 ± 0.0694	1.3276 ± 0.2589	1.8141 ± 0.0276	3.0958 ± 0.0441
512	1.6916 ± 0.0110	1.2567 ± 0.1227	2.2818 ± 0.7011	3.1656 ± 0.0457
1024	2.0228 ± 0.5684	1.3167 ± 0.1341	6.3756 ± 8.4676	3.2896 ± 0.1396
2048	1.7218 ± 0.0288	1.5416 ± 0.1405	1.9099 ± 0.0898	3.4844 ± 0.1113
4096	1.1411 ± 0.0404	1.4010 ± 0.0788	2.0062 ± 0.1562	3.8562 ± 0.3197
8192	1.1105 ± 0.0078	1.4818 ± 0.0759	2.3671 ± 0.1897	3.8474 ± 0.4784
16384	1.1183 ± 0.0280	1.7308 ± 0.1043	2.5833 ± 0.1737	4.9724 ± 0.8955
32768	1.1162 ± 0.0084	2.2099 ± 0.1880	3.2062 ± 0.2029	5.3719 ± 0.2875
65536	1.7463 ± 1.2217	4.7474 ± 3.1960	4.3198 ± 0.2926	6.8136 ± 0.2499
131072	1.1027 ± 0.0141	2.6375 ± 0.1531	5.7004 ± 0.2681	9.6912 ± 1.4337
262144	1.1006 ± 0.0118	3.3172 ± 0.1164	7.4630 ± 0.2309	10.2781 ± 0.2278

JNI (Time μs)

Block size	No params	Vector	Convert	Columbia
16	1.7922 ± 0.1392	1.9333 ± 0.1223	2.6052 ± 0.1004	4.1058 ± 0.3042
32	1.6983 ± 0.0220	2.8130 ± 1.7924	2.6006 ± 0.0370	3.9109 ± 0.0535
64	1.6755 ± 0.0149	1.6344 ± 0.1809	2.6630 ± 0.0425	3.9296 ± 0.0566
128	1.9604 ± 0.4978	1.2349 ± 0.1262	1.9375 ± 0.0843	3.0823 ± 0.0892
256	1.7292 ± 0.0694	1.3276 ± 0.2589	1.8141 ± 0.0276	3.0958 ± 0.0441
512	1.6916 ± 0.0110	1.2567 ± 0.1227	2.2818 ± 0.7011	3.1656 ± 0.0457
1024	2.0228 ± 0.5684	1.3167 ± 0.1341	6.3756 ± 8.4676	3.2896 ± 0.1396
2048	1.7218 ± 0.0288	1.5416 ± 0.1405	1.9099 ± 0.0898	3.4844 ± 0.1113
4096	1.1411 ± 0.0404	1.4010 ± 0.0788	2.0062 ± 0.1562	3.8562 ± 0.3197
8192	1.1105 ± 0.0078	1.4818 ± 0.0759	2.3671 ± 0.1897	3.8474 ± 0.4784
16384	1.1183 ± 0.0280	1.7308 ± 0.1043	2.5833 ± 0.1737	4.9724 ± 0.8955
32768	1.1162 ± 0.0084	2.2099 ± 0.1880	3.2062 ± 0.2029	5.3719 ± 0.2875
65536	1.7463 ± 1.2217	4.7474 ± 3.1960	4.3198 ± 0.2926	6.8136 ± 0.2499
131072	1.1027 ± 0.0141	2.6375 ± 0.1531	5.7004 ± 0.2681	9.6912 ± 1.4337
262144	1.1006 ± 0.0118	3.3172 ± 0.1164	7.4630 ± 0.2309	10.2781 ± 0.2278

JNI (Time µs)

Block size	No params	Vector	Convert	Columbia
16	1.7922 ± 0.1392	1.9333 ± 0.1223	2.6052 ± 0.1004	4.1058 ± 0.3042
32	1.6983 ± 0.0220	2.8130 ± 1.7924	2.6006 ± 0.0370	3.9109 ± 0.0535
64	1.6755 ± 0.0149	1.6344 ± 0.1809	2.6630 ± 0.0425	3.9296 ± 0.0566
128	1.9604 ± 0.4978	1.2349 ± 0.1262	1.9375 ± 0.0843	3.0823 ± 0.0892
256	1.7292 ± 0.0694	1.3276 ± 0.2589	1.8141 ± 0.0276	3.0958 ± 0.0441
512	1.6916 ± 0.0110	1.2567 ± 0.1227	2.2818 ± 0.7011	3.1656 ± 0.0457
1024	2.0228 ± 0.5684	1.3167 ± 0.1341	6.3756 ± 8.4676	3.2896 ± 0.1396
2048	1.7218 ± 0.0288	1.5416 ± 0.1405	1.9099 ± 0.0898	3.4844 ± 0.1113
4096	1.1411 ± 0.0404	1.4010 ± 0.0788	2.0062 ± 0.1562	3.8562 ± 0.3197
8192	1.1105 ± 0.0078	1.4818 ± 0.0759	2.3671 ± 0.1897	3.8474 ± 0.4784
16384	1.1183 ± 0.0280	1.7308 ± 0.1043	2.5833 ± 0.1737	4.9724 ± 0.8955
32768	1.1162 ± 0.0084	2.2099 ± 0.1880	3.2062 ± 0.2029	5.3719 ± 0.2875
65536	1.7463 ± 1.2217	4.7474 ± 3.1960	4.3198 ± 0.2926	6.8136 ± 0.2499
131072	1.1027 ± 0.0141	2.6375 ± 0.1531	5.7004 ± 0.2681	9.6912 ± 1.4337
262144	1.1006 ± 0.0118	3.3172 ± 0.1164	7.4630 ± 0.2309	10.2781 ± 0.2278

JNI (Time µs)

Block size	No params	Vector	Convert	Columbia
16	1.7922 ± 0.1392	1.9333 ± 0.1223	2.6052 ± 0.1004	4.1058 ± 0.3042
32	1.6983 ± 0.0220	2.8130 ± 1.7924	2.6006 ± 0.0370	3.9109 ± 0.0535
64	1.6755 ± 0.0149	1.6344 ± 0.1809	2.6630 ± 0.0425	3.9296 ± 0.0566
128	1.9604 ± 0.4978	1.2349 ± 0.1262	1.9375 ± 0.0843	3.0823 ± 0.0892
256	1.7292 ± 0.0694	1.3276 ± 0.2589	1.8141 ± 0.0276	3.0958 ± 0.0441
512	1.6916 ± 0.0110	1.2567 ± 0.1227	2.2818 ± 0.7011	3.1656 ± 0.0457
1024	2.0228 ± 0.5684	1.3167 ± 0.1341	6.3756 ± 8.4676	3.2896 ± 0.1396
2048	1.7218 ± 0.0288	1.5416 ± 0.1405	1.9099 ± 0.0898	3.4844 ± 0.1113
4096	1.1411 ± 0.0404	1.4010 ± 0.0788	2.0062 ± 0.1562	3.8562 ± 0.3197
8192	1.1105 ± 0.0078	1.4818 ± 0.0759	2.3671 ± 0.1897	3.8474 ± 0.4784
16384	1.1183 ± 0.0280	1.7308 ± 0.1043	2.5833 ± 0.1737	4.9724 ± 0.8955
32768	1.1162 ± 0.0084	2.2099 ± 0.1880	3.2062 ± 0.2029	5.3719 ± 0.2875
65536	1.7463 ± 1.2217	4.7474 ± 3.1960	4.3198 ± 0.2926	6.8136 ± 0.2499
131072	1.1027 ± 0.0141	2.6375 ± 0.1531	5.7004 ± 0.2681	9.6912 ± 1.4337
262144	1.1006 ± 0.0118	3.3172 ± 0.1164	7.4630 ± 0.2309	10.2781 ± 0.2278

JNI

- Overhead not significant
- JNI implementation differ between VMs
- Previous studies have reporter larger execution times
- Resolution of Java Timer

Libraries (Java)

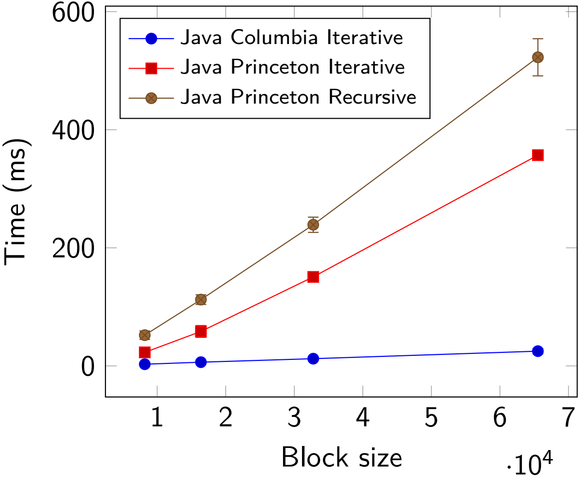


Table: Java line graph for *large* block sizes with standard deviation error bars

Libraries (C++)

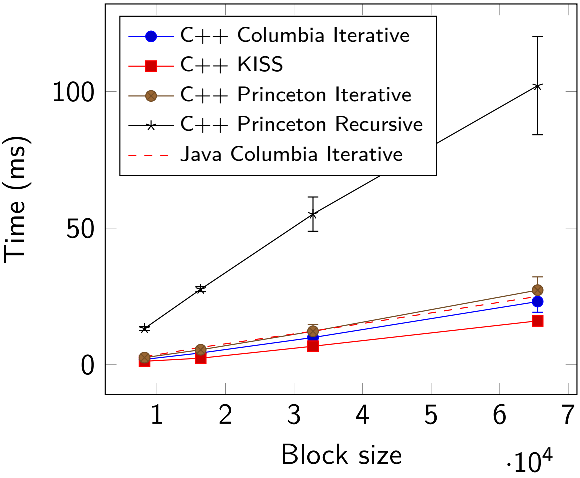


Figure: C++ line graph for *large* block sizes with standard deviation error bars

Libraries

- Recursive algorithm worst
- Java Columbia Iterative comparable with C++
- Translating Java $\rightarrow$ C++

NEON

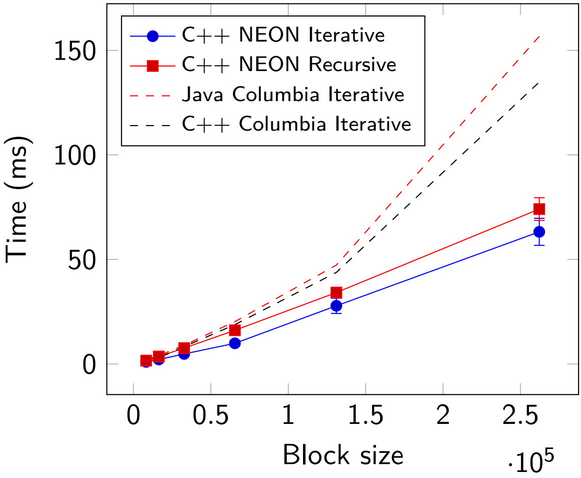


Figure: NEON results table for *extra large* block sizes, Time (ms)

NEON

- Vectorization is clearly faster
- Columbia Iterative diverges for larger execution times
- Vectorization tests require some setup

Garbage Collection

Table: Pauses due to garbage collection

Algorithm	# partial	tot. time (ms)	# sticky	tot. time (ms)
Java Columbia Iterative	0	0	0	0
Java Princeton Iterative	477	2,825.52	406	2,959.24
Java Princeton Recursive	240	602.10	397	887.39
Java Float Columbia Iterative	0	0	0	0
Java Float Princeton Iterative	269	1,541.97	334	2,316.53
Java Float Princeton Recursive	167	313.05	27	71.39
C++ Columbia Iterative	0	0	0	0
C++ Princeton Iterative	0	0	0	0
C++ Princeton Recursive	0	0	0	0
C++ Float Columbia Iterative	0	0	0	0
C++ Float Princeton Iterative	0	0	0	0
C++ Float Princeton Recursive	0	0	0	0
KISS	0	0	0	0
NEON Iterative	0	0	0	0
NEON Recursive	0	0	0	0

Garbage Collection

Table: Pauses due to garbage collection

Algorithm	# partial	tot. time (ms)	# sticky	tot. time (ms)
Java Columbia Iterative	0	0	0	0
Java Princeton Iterative	477	2,825.52	406	2,959.24
Java Princeton Recursive	240	602.10	397	887.39
Java Float Columbia Iterative	0	0	0	0
Java Float Princeton Iterative	269	1,541.97	334	2,316.53
Java Float Princeton Recursive	167	313.05	27	71.39
C++ Columbia Iterative	0	0	0	0
C++ Princeton Iterative	0	0	0	0
C++ Princeton Recursive	0	0	0	0
C++ Float Columbia Iterative	0	0	0	0
C++ Float Princeton Iterative	0	0	0	0
C++ Float Princeton Recursive	0	0	0	0
KISS	0	0	0	0
NEON Iterative	0	0	0	0
NEON Recursive	0	0	0	0

Garbage Collection

Algorithm	Block Size
Princeton Iterative	8192
Princeton Recursive	4096
Float Princeton Iterative	16384
Float Princeton Recursive	8192

- Immutable Complex Class
-
- Pre-allocate Arrays to minimize GC

`float` vs `double`

Table: Java `float` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
32768	8.9846 ± 0.2566	113.1953 ± 3.9572	219.0208 ± 5.5642
65536	20.2833 ± 0.4786	261.9954 ± 9.1987	485.1020 ± 13.3737
131072	47.2950 ± 1.3073	622.4328 ± 24.9022	1039.4937 ± 26.9767
262144	156.7135 ± 1.1934	1728.4640 ± 53.8042	2297.8011 ± 58.2951

Table: Java `double` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
32768	12.2634 ± 0.5700	150.7299 ± 1.0864	239.0777 ± 2.5276
65536	24.9874 ± 0.9069	356.9871 ± 1.5864	522.7409 ± 6.1660
131072	85.9483 ± 1.6097	815.8607 ± 3.4304	1144.8802 ± 17.8736
262144	274.5134 ± 5.1129	2108.0771 ± 27.5366	2638.0547 ± 40.5424

`float` vs `double`

Table: Java `float` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
32768	8.9846 ± 0.2566	113.1953 ± 3.9572	219.0208 ± 5.5642
65536	20.2833 ± 0.4786	261.9954 ± 9.1987	485.1020 ± 13.3737
131072	47.2950 ± 1.3073	622.4328 ± 24.9022	1039.4937 ± 26.9767
262144	$\boxed{156.7135 \pm 1.1934}$	1728.4640 ± 53.8042	2297.8011 ± 58.2951

Table: Java `double` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
32768	12.2634 ± 0.5700	150.7299 ± 1.0864	239.0777 ± 2.5276
65536	24.9874 ± 0.9069	356.9871 ± 1.5864	522.7409 ± 6.1660
131072	85.9483 ± 1.6097	815.8607 ± 3.4304	1144.8802 ± 17.8736
262144	$\boxed{274.5134 \pm 5.1129}$	2108.0771 ± 27.5366	2638.0547 ± 40.5424

float vs double

Table: Java `float` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
32768	8.9846 ± 0.2566	113.1953 ± 3.9572	219.0208 ± 5.5642
65536	20.2833 ± 0.4786	261.9954 ± 9.1987	485.1020 ± 13.3737
131072	47.2950 ± 1.3073	622.4328 ± 24.9022	1039.4937 ± 26.9767
262144	156.7135 ± 1.1934	1728.4640 ± 53.8042	2297.8011 ± 58.2951

Table: Java `double` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
32768	12.2634 ± 0.5700	150.7299 ± 1.0864	239.0777 ± 2.5276
65536	24.9874 ± 0.9069	356.9871 ± 1.5864	522.7409 ± 6.1660
131072	85.9483 ± 1.6097	815.8607 ± 3.4304	1144.8802 ± 17.8736
262144	274.5134 ± 5.1129	2108.0771 ± 27.5366	2638.0547 ± 40.5424

`float` vs `double`

Table: Java `float` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
32768	8.9846 ± 0.2566	113.1953 ± 3.9572	219.0208 ± 5.5642
65536	20.2833 ± 0.4786	261.9954 ± 9.1987	485.1020 ± 13.3737
131072	47.2950 ± 1.3073	622.4328 ± 24.9022	1039.4937 ± 26.9767
262144	156.7135 ± 1.1934	1728.4640 ± 53.8042	2297.8011 ± 58.2951

Table: Java `double` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
32768	12.2634 ± 0.5700	150.7299 ± 1.0864	239.0777 ± 2.5276
65536	24.9874 ± 0.9069	356.9871 ± 1.5864	522.7409 ± 6.1660
131072	85.9483 ± 1.6097	815.8607 ± 3.4304	1144.8802 ± 17.8736
262144	274.5134 ± 5.1129	2108.0771 ± 27.5366	2638.0547 ± 40.5424

`float` vs `double`

- Likely due to caching
- `float` $\rightarrow$ 32-bit
- `double` $\rightarrow$ 64-bit

1 Introduction
- Purpose of Thesis
- Research Question

2 Background
- Android Development
- Java Native Interface (JNI)
- Discrete Fourier Transform (DFT)
- Fast Fourier Transform (FFT)
- Related Work

3 Method
- Experiments
- Measurements
- Implementation

4 Results and Discussion
- JNI
- Libraries
- NEON
- Garbage Collection
- `float` vs `double`

5 Conclusions

Conclusions

Conclusion 1

The overhead from JNI does not have a significant effect on performance.

Conclusion 2

Of the tested algorithms, choose Columbia Iterative.

Conclusion 3

Avoid allocating memory in the run-loop of a recurring task.

Conclusion 4

NEON optimization is significantly faster than non-optimized code for larger block sizes.

Questions?