High performance computing on Android devices - a case study.

Robert Fritze\textsuperscript{a,1,}, Claudia Plant\textsuperscript{a}

\textsuperscript{a}University of Vienna, Faculty of Computer Science, Research Group Data Mining, Währinger Straße 29, 1090, Vienna, Austria

Abstract

High performance computing for low power devices can be useful to speed up calculations on processors that use a lower clock rate than computers for which energy efficiency is not an issue. In this trial, different high performance techniques for Android devices have been compared, with a special focus on the use of the GPU. Although not officially supported, the OpenCL framework can be used on Android tablets.

For the comparison of the different parallel programming paradigms, a benchmark was chosen that could be implemented easily with all frameworks. The Mandelbrot algorithm is computationally intensive and has very few input and output operations. The algorithm has been implemented in Java, C, C with assembler, C with SIMD assembler, C with OpenCL and scalar instructions and C with OpenCL and vector instructions. The implementations have been tested for all architectures currently supported by Android.

High speedups can be achieved using SIMD and OpenCL, although the implementation is not straightforward for either one. Apps that use the GPU must account for the fact that they can be suspended by the user at any moment. In using the OpenCL framework on the GPU of Android devices, a computational power comparable to those of modern high speed CPUs can be made available to the software developer.

Keywords: Android, SIMD, OpenCL, GPU, multithreading, Mandelbrot

1. Introduction

Low-power CPUs (central processing units) have become increasingly popular in recent years, especially for mobile computing where energy efficiency is an issue. These CPUs often have a lower clock rate and, therefore, efficient parallelization techniques may be important to optimize runtime for more complex workloads [1].

Android is a widely used Linux-based open-source operating system for mobile devices and allows an easy user interaction. Android Studio provides a free and easy to use IDE (integrated development environment) for the creation of apps. Android supports only a limited set of processor architectures (currently four), which makes it easier to produce different flavors of assembler code.

The Mandelbrot set [2] is the set of complex numbers $c$ for which the sequence of complex numbers $(z_n)_{n \in \mathbb{N}}$ defined by the iteration

$$z_0 = 0$$
$$z_{n+1} = z_n^2 + c$$

is limited. In practice, a number $N \in \mathbb{N}$ is selected. If the absolute value of $z_N$ is still below 2, the sequence is regarded as limited. The calculation of the iteration is computationally intensive and has few input/output operations compared to the number of arithmetic operations. Moreover, this algorithm can be implemented rather easily with different parallel programming paradigms (multithreading, vector parallelism, OpenCL) and assembler code can be produced with moderate effort. Used as a benchmark, this algorithm measures the CPU performance for floating point operations executed serially or in parallel. Implemented with different parallelization paradigms, the performance of the algorithm can be compared in a repeatable and transparent manner. The calculation of the Mandelbrot set has been used several times before as a benchmark [3, 4].

1.1. Contributions

- A survey of how the OpenCL framework can be used on Android mobile devices
- A wrapper library for the use of OpenCL libraries on Android devices
- A locking mechanism to allow the unloading of the wrapper library and memory optimization while the app is not in use
- A locking mechanism to allow calculations performed on the GPU to be safely aborted when the user has stopped the app
- A comparison of parallelization techniques (e.g. SIMD, multithreading) for all currently-supported Android architectures
- A comparison of parallel programming paradigms performed on an ARMv7 tablet GPU (Mali G-71 MP 2)
1.2. Previous literature

Many research projects currently focus on the ability to use lower power processors (especially ARM) for high performance computing but few trials investigated the use of OpenCL on mobile devices and compared it to other parallel programming paradigms. Acosta et al. [5] gave a nice overview of the availability of OpenCL capable GPUs on mobile devices. Ross et al. [6] published a paper in 2014 where they compare different types of graphic cards. One of the devices was a GPU on a mobile device. The Mont Blanc project [11] aims to create a basis for high performance computing on low power CPUs. For this project Pérez et al. [7], [8] tried to create an OpenCL environment, that can be executed on multiple low power devices concurrently.

Wang et al. [4] used the Mandelbrot set to evaluate the performance of an Android Adreno-GPU. They only compared OpenCL and Java. They used a slightly different setting and the speedups were lower compared to those achieved in this trial. In 2013, Wang et al. [9] implemented an image processing application on Android mobile devices with GPUs using OpenCL, that allowed the removal of objects from images. Yokoyama et al. [10] summarize in a 2019 review the efforts that have been made to use ARM processors for high performance computing.

2. Calculations

2.1. Test environment

For this trial three distinct windows of the Mandelbrot set were arbitrarily selected (see Table 1 and Figures 1a-c), representing modest, intermediate and high workload. All three windows had a size of 1600x1072 pixels. For each of the architectures (Intel-x86, Intel-x86_64, Arm-v7, Arm-v8, GPU), each possible number of threads available on the device (except GPU) and each precision available, the wall clock time of the Mandelbrot algorithm has been recorded 50 subsequent times. Every ten cycles a pause of ten seconds was introduced. This pause should allow the processors to cool down and avoid throttling due to overheat. In order to test different parallel execution paradigms (threads, SIMD (single instruction multiple data), GPU) and combinations thereof, the following flavors of the algorithm have been implemented for each architecture: pure Java, pure C, and handwritten scalar assembler, C and handwritten SIMD assembler, C and OpenCL [11] with scalar instructions and C and OpenCL with vector instructions. Single threaded and mutithreaded versions have been used. The mutithreaded versions were implemented using the producer-consumer principle [12] and the workload was divided into 67 chunks of 16 subsequent lines. Assembler instructions were implemented as inline assembler in C. Parts of the program written in C were called from Java using JNI (Java native interface). For all versions that contained C, mutithreading was implemented in C with the pthread library. On all devices, WLAN was disabled before the program execution was started. For the Java, C and OpenCL versions, care has been taken to avoid implicit type casts. Conditional jump instructions have been eliminated wherever possible in the assembler code. See Table 2 for the list of executions environments.

All implementations (independently of the programming language) followed the same scheme: outer loop over the lines, inner loop over pixel of a line and a while loop for the calculations for each pixel (see Listing 1). The C programs were compiled with Clang 8.0 and the default compiler options set by Android Studio. Additionally, the optimization level was set to -O3.

For all assembler programs except for the x86 SIMD variant no local variables were required. Instead, all intermediate results of the calculations were kept in processor registers. The only RAM (random access memory) access was the storage of the results. For the implementation of the SIMD x86 assembler three local variables were necessary, as register pressure was too high. The local variables were accessed only in the outer loops.

2.2. Precision

Not all types of precisions were available for all devices. Single precision was always available. Many assembler languages support half precision as storage format, but among the CPUs available for this trial only Arm-v8 assembler supports half precision arithmetic as optional extension (Arm-v8.2) which was not present on the Arm-v8 CPU used. Half precision was available only on the GPU of the Arm-v7 tablet. Double precision was not available for the Arm-v7 SIMD instructions and on the GPU.

```java
1 float ci = starty;
2 for( int i1=0; i1<HEIGHT; i1++ ){
3   float cr = startx;
4   for( int i2=0; i2<WIDTH; i2++ ){
5     float zi = 0;
6     float zi = 0;
7     int iter = 0;
8     while (((zr*zr + zi*zi)<=4f) && (iter<MAXITER)){
9       float t = zr*zr - zi*zi + cr;
10      zi = 2f*zr*zi + ci;
11      zr = t;
12      iter++;
13    }
14   }
15  }
16  }
17  }
18  }
19  result[i1*WIDTH + i2] = iter;
20  cr += xstepsize;
21  }
22  ci -= ystepsize;
23 }
```

Listing 1: Java implementation of the calculation of a window of the Mandelbrot set for one thread and single precision.
2.3. SIMD

SIMD instructions are available on CPUs and GPUs. On the CPU, vectors can hold four single or two double precision floating point numbers (Arm NEON and Intel SSE). Using OpenCL [11] for accessing the GPU, SIMD vector instructions can have 2, 4, 8 or 16 entries even if the ISA (instruction set architecture) of the GPU does not support all of these vector widths. If the ISA does not support a vector width, the OpenCL compiler must split the vector into chunks of appropriate vector sizes or convert the instruction into a loop.

SIMD instructions apply arithmetical or relational operations on all elements of a vector concurrently. SIMD data parallelism for the calculation of the Mandelbrot set is not straightforward: neighbouring pixels generally will not need the same number of iterations to compute the result. But in the case of the Mandelbrot set, for many parts of the picture the number of iterations for subsequent pixels will be similar. A blending mask was used to exclude vector items that should not be updated any more once their final result has been computed.

Listing 2 shows a part of the innermost loop with the SIMD blending procedure in x86_64 assembler. xmm7 holds four single precision floats that represent the results of the calculation of \( (re^2 + im^2) \) of four subsequent pixels. xmm10 holds four single precision constants (4.0f). In line 21 the SIMD comparison is made. The result of the comparison is written back into the xmm7 register. After the execution of line 21, all bits of each vector entry of xmm7 are zero if the comparison for the corresponding vector entry was false, one else.

xmm5 holds four 32-bit integers and is initialized with 1 before the innermost loop is entered (line 1). This register serves as increment for the iteration counter. In line 25, if the above comparison was false, the corresponding entry in xmm5 will be set to zero and always remain zero.

xmm4 is the iteration counter and holds four 32 bit inte-

|       | Window 1 | Window 2     | Window 3     |
|-------|----------|--------------|--------------|
| \(x_1\) | -2       | -0.739       | -0.737       |
| \(y_1\) | 1        | 0.1448888    | 0.14667      |
| \(x_2\) | 1        | -0.734       | -0.736       |
| \(y_2\) | -1       | 0.1415       | 0.146        |
| iterations | 80       | 1500         | 3000         |

Table 1: Coordinates and maximum number of iterations of the three windows selected. \(x_1 + iy_1\) = left upper corner, \(x_2 + iy_2\) = right lower corner.
ggers. They are initialized with zero before the innermost loop is started (line 4). This register counts the number of iterations needed until the condition in line 21 becomes false. Line 27 adds one or zero to each vector entry of the iteration counter, depending on the values of xmm5. Line 30 tests if all entries of xmm5 were zero. If all entries of xmm5 were zero, in line 34 ecx is loaded with zero. If not, ecx will remain untouched. ecx can become zero if the comparison in line 21 is false for all vector entries or if the maximum number of iterations has been reached. Similar constructs have been used in all SIMD assembler implementations and show that SIMD can be used even if the vector entries need a similar but unequal number of arithmetic operations to calculate the result. The instructions executed in the lines 11-37 in listing 1 executed for four values in parallel).

2.4. GPU - OpenCL

For deploying calculations on the GPU, the OpenCL [11] standard has been chosen. OpenCL is a framework that uses a C-style programming language and implements task and data based parallelism. Programs can be compiled and deployed on a great variety of architectures like CPUs, GPUs, FPGAs (Field Programmable Gate Array) and many more. An advantage of OpenCL is its portability across devices [13].

Many Android tablets are shipped with an OpenCL-capable GPU and the necessary libraries, although OpenCL is not officially supported by Android. OpenCL can be used from C linking an appropriate shared library. C compilers require shared libraries to be present at compile-time. This is not the case for the usual Android project build process, which takes place on an external host (e.g., with Android Studio).

Many manuals recommend building and shipping the device’s OpenCL library together with the apk file [6]. This approach has compatibility issues with other GPU vendors and future GPUs of the same manufacturer. For this project another approach has been selected: The POSIX standard provides a method to dynamically load shared libraries and resolve the symbols at runtime (with the ‘dlopen’ command). The native OpenCL libraries do not need to be present at compiletime. Once a library has been loaded at runtime, the symbols (names of the methods) in the library must be resolved manually. In the current implementation of the wrapper library this is not done immediately but only on request if a call to a method is really needed.

Unfortunately there is one obstacle to this approach: Since Android 7 Google does not allow vendor provided libraries to be loaded if they do not appear in a special list. If this list is stored on a read-only filesystem, it can be impossible to create that list for older devices (which was the case for the Arm-v8 tablet).

2.4.1. The wrapper library

For this project an OpenCL wrapper library has been created. This library implements the entire OpenCL 3.0 standard and forwards the calls to its methods to the corresponding method of the native OpenCL shared library of the device. The wrapper library is compatible with any (lower) OpenCL version of the native OpenCL library. If one tries to call OpenCL methods of a standard that is not provided by the native library, an error is returned (the corresponding symbol cannot be resolved). The wrapper library is linked to the app at compile-time.

At the very beginning of the program, a method has to be called to load the native library from the device. The only parameter for this method is the path to the native library on the device. Figure S1 in the supplements shows a sequence diagram of the use of the wrapper library in conjunction with the native library.

Unlike desktops or laptops where the programmer is supposed to have permanent access to all resources (especially the GPU) during the whole lifespan of a program, on Android devices a user can choose to stop an app at any time. Programs that use the GPU have to account for that, because otherwise the resources on the GPU might remain blocked and the GPU might not be fully available by other programs (especially if

| Android Version | x86 | x86_64 | Arm-v7 | Arm-v8 |
|-----------------|-----|-------|-------|-------|
| Android API     | 24  | 24    | 28    | 24    |
| Manufacturer    | Intel | Intel | Samsung | ASUS |
| Model           | NUC7i7BNH | NUC7i7BNH | SM-T510 | ZenPad 10 |
| CPU             | Core i7-7567U | Core i7-7567U | Exynos 7885 | Mediatek MT8163 |
| CPU-speed (MHz) | 400-3500 | 400-3500 | 449-1768 | 600-1300 |
| Virtualization  | qemu/kvm | qemu/kvm | - | - |
| GPU             | - | - | Mali-G71 MP 2 | Mali-T720 (a) |
| Precision CPU   | s+d | s+d | s+d | s+d |
| Precision SIMD  | s+d | s+d | s | s+d |
| Precision GPU   | - | - | h+s | - |
| Cores           | 4 (b) | 4 (b) | 8 (c) | 4 (d) |
| RAM (MB)        | 2048 | 2048 | 2879 | 1953 |
| 32/64 Bit CPU   | 64 | 64 | 64 | 64 |

Table 2: Execution environments. h=half, s=single, d=double. (a) was present but could not be used due to Android operating system restrictions. (b) with Intel Hyper-threading technology (c) 2 x Cortex A73 + 6 x Cortex A53. (d) 4 x Cortex A53.
the GPU has its own memory). Ideally the app should also free any memory not used when it loses the focus. To accomplish these issues two mechanisms have been implemented. The first one is part of every call to an OpenCL method of the wrapper library and the programmer does not have to take care of this mechanism. The second one must be integrated into the app because the resources blocked on the GPU should be freed by the programmer and not the wrapper library.

| Line   | Assembly Instruction          | Comment                                                                 |
|--------|------------------------------|-------------------------------------------------------------------------|
| 1-2    | movdqa xmm5, xmm14; xmm5 <- (1,1,1,1).U32, | ; iteration step                                                       |
| 3-4    | movdqa xmm4, xmm15; xmm4 <- (0,0,0,0).U32, | ; iteration counter                                                     |
| 5-6    | mov rcx, r9; ecx <- MAXITER   |                                                                         |
| 7-9    | .Label2:                     |                                                                         |
| 10-12  | cmp ecx, 0; maximum iterations; reached? | je .Label3; yes -> quit inner; loop                                    |
| 13     | dec ecx; ecx--               |                                                                         |
| 14-15  | (...) ; eleven SIMD floating point instructions to update; real and imaginary part |                                                                         |
| 16-20  | cmpps xmm7, xmm10, 2         | ; (re + im^2) <= 4.0f?                                                   |
| 21     | sxmm10 = (4.0f,4.0f,4.0f,4.0f).F32 | ; blend iteration offset                                                |
| 22     | pand xmm5, xmm7              | ; step vector iteration                                                 |
| 23     | ptest xmm5, xmm9             | ; counter                                                               |
| 24     | cmovz ecx, edx               | ; quit loop if ZF==1                                                   |
| 25     | jmp .Label12                 | ; (edx=0)                                                              |
| 26     | .Label3:                     |                                                                         |

Listing 2: x86-64 Assembler listing with SSE4.1 SIMD extensions of the innermost loop. xmm4 holds four integer iteration counters, xmm5 holds the iteration step (four times zero or one integer values), xmm7 holds the result re^2 + im^2 of four consecutive pixels, xmm9 has all bits set, xmm10 holds four times 4.0f, edx=0; other explanations see [Footnote 2].

2.4.2. The unloading mechanism

Android devices usually have limited memory resources. If an app loses focus, it is put on the so-called 'backstack'. If the dynamically loaded native OpenCL library is temporarily unused, the memory it occupies can be released using a reader/writer lock and a flag. A RW lock is needed to avoid unloading the native library while there are still active calls to the native library. A flag holds the status of the native library (has never been loaded, currently loaded, unloaded) and is protected by the mutex in order to allow concurrent access to it.

Figure 2 depicts the use of the RW locks and the mutex. At the beginning of every call to a method of the wrapper OpenCL library, the reader lock of the reader/writer lock and the mutex that protects the flag are acquired. Deadlocks cannot occur, as the locks are always blocked in the same order. Once exclusive access to the flag has been gained, the method of the wrapper library checks if the native library has ever been loaded. If not, the method returns with an error because the path and filename from where to load the native library have not yet been specified. If path and filename are known, the method of the wrapper library tests if the native library is currently loaded and tries to load it, if it has been unloaded. Furthermore, the wrapper library checks if the symbol (i.e. the method name) in the dynamically loaded library has already been resolved. If not, the wrapper library tries to resolve the symbol.

The mutex is released before the call to the corresponding method of the native OpenCL library and the reader/writer lock afterwards. This mechanism allows the use of OpenCL functions of the native library by multiple threads concurrently. The lock should be reader-preferred, because otherwise, as long as there are readers waiting, the library would be reloaded immediately after it has been unloaded.

The wrapper library has a method that allows one to unload the dynamically loaded native library. Before releasing the dynamically loaded OpenCL library, the method responsible for unloading the library acquires the writer part of the reader/writer lock and gains exclusive access to the native OpenCL library. There are no pending calls to the native library once the writer lock has been acquired. Next the mutex for the flag is acquired and the flag is set accordingly. Afterwards the library is unloaded using 'dlclose'. Finally the mutex and the lock are released. The Android operating system decides if and when the freed shared library is effectively unloaded and the memory freed.

If calculations on the GPU should be resumed, the shared library does not have to be reloaded explicitly but will be loaded implicitly upon the next call to a wrapper library method by the mechanism described above. If the library should not be reloaded anymore, the programmer has to make sure that no more calls to the wrapper library are made once it has been unloaded. This is the purpose of the second locking mechanism described in the next subsection.

2.4.3. GPU memory release upon request

If an app is stopped or killed (by the operating system or the user), the app’s ‘onStop’ and ‘onDestroy’ methods are called by the operating system. The execution of these methods cannot
be delayed for a longer period for finishing calculations on the GPU, because the user would otherwise be notified that the app is not responding. On the other hand, once these methods have been called, resources on the GPU should be freed correctly.

The OpenCL wrapper library is state-less and therefore the resources a software developer has blocked on the GPU have to be freed by the app and cannot be freed by the wrapper library. To address this issue, a writer-preferred reader/writer lock, a mutex and a flag are sufficient. The flag is protected by the mutex. Initially the flag is set to zero.

Figure 3 illustrates a possible mechanism to stop execution of the app and free resources blocked on the GPU. If a method of an app wants to use the GPU, the method should acquire the reader lock of the reader/writer lock before the first call to the wrapper library (except the call to the method that loads the library the very first time). This lock is held until the last call to the wrapper library has been made. After the lock has been acquired, the mutex should be acquired and the flag should be checked. If the flag is not zero, the mutex and the lock should be released and the method should return with an error. If the flag is zero, the mutex should be released and the calculations on the GPU can be carried out. After all calculations have been finished, the reader lock should be released. The reader lock can be acquired multiple times concurrently by different threads, allowing the parallel use of the wrapper library.

The approach described here works well if the kernel execution lasts only for a short period. If calculations on the GPU are more complex and, especially if they involve several subsequent calls to different kernels, the reader lock should be acquired at the very beginning and released once all GPU resources have been freed. The execution of a kernel cannot be easily stopped. But between kernel executions, the flag can be tested (using the mutex) and if the flag meanwhile has been set to one, the resources should be freed, the reader lock should be released and an error should be returned.

If the 'onStop' and 'onDestroy' methods of an app are called by the operating system, they acquire the mutex (without first acquiring the lock) and set the flag to one (see figure 4). The mutex is released immediately afterwards. Then the writer lock of the reader/writer lock should be acquired. It is important that the lock is writer-preferred, because otherwise the method could starve. Once the writer lock has been acquired, no more
Figure 3: Activity diagram of the mechanism that should be executed every time before a call to the wrapper library is made. This diagram should be implemented in the app. For further explanation see 2.4.3.

Figure 4: Activity diagram of the method that locks the access to the wrapper library. For further explanations see 2.4.3.

readers can hold the reader lock and all GPU resources must have been freed. The writer lock should be released immediately afterwards and the native OpenCL library can be unloaded. No GPU resources will be allocated ever again and no calls to the wrapper library can be made, because the flag has been set to one and is checked before the GPU is used. The native library can be unloaded at this moment because no readers will be waiting any more (all resources have been released, no new readers can start to use the GPU).

If the calculations should be resumed (‘onResume’) it is sufficient to acquire the mutex and set the flag again to zero (and release the mutex afterwards). If the native library has been unloaded, it will be loaded automatically when the next call to a wrapper functions is made.

Besides the initial call to load the native OpenCL library and the locking mechanism for the premature termination of the calculations, any program using OpenCL should run without any modification using the methods of the wrapper library instead of the methods of the native OpenCL library (just the linking process of the C compiler has to be modified accordingly). Of course neither the mechanism of loading/unloading the native OpenCL libraries at runtime nor the mechanism of premature suspension are specific to Android and work on all POSIX compliant operating systems.

2.5. The App

In order to test the mechanisms described above, a small app with three activities has been written and deployed on the tablets. The first activity allows to enter the path and file name of the native OpenCL library on the device. The second activity is responsible for the proper selection of the parameter for the calculations (method, number of threads, vector size, window and precision). The third activity performs the calculations on the device based on the selections made in the second activity and displays the final result.
3. Results

Tables S1 to S9 in the supplements show the results for the calculations on the CPU. The first line of each cell shows the mean (or the median if the sample was not normally distributed) of the wall clock time in milliseconds. The second line shows the confidence interval (if the sample was normally distributed). The third line shows the speedups. The first value is the speedup with respect to Java with the same number of threads, the same window and the same precision (first value of the same row). The second value shows the speedup with respect to the same method, same window, same precision but only one thread (first value in each column). The third value shows the speedup with respect to the single threaded Java version with the same precision and same window (value in the left upper corner for each window). The small letters indicate the method that has been used for the calculation of the significance. Normal distribution was tested with the Shapiro-Wilk test [14] and homoscedasticity was tested with the Levene test [15]. If two samples were normally distributed with equal variances, the student t-test [16] was used to test significance of the difference of the means. If the two samples were distributed normally but had unequal variances, the Welch test [17] was used. If the two samples had similar distributions but were not normally distributed, the Wilcoxon-Mann-Whitney test [18] [19] was used. If none of the cases above were true, the Median test [20] was used. The results displayed in the tables are summarized in the Figures 5, 6, 7 and 8.

For all types of architecture and for all methods, for a given number of threads and a given precision there was a significant (p<0.05) difference in the runtime between the windows. The more one zoomed into the Mandelbrot set, the longer took the calculations. When more threads were used, the execution completed faster. The difference of runtime was highest between one and two threads and declined as more threads were used. Except for the Intel architectures, runtimes can not be directly compared between the different devices as the execution environments differ. Multithreaded SIMD instructions gave had a very good performance. Independently of the architecture the maximum speedups were achieved with SIMD and with four threads they ranged from 5.23 to 15.38 (median 10.485) for single precision and from 4.35 to 5.98 (median 5.75) for double precision.

3.1. Intel based instruction sets (x86 and x86_64)

The Java version was consistently faster on the x86_64 than on the x86 device. For all other versions there was no major runtime difference, independently of the precision, the window and the number of threads (see Tables S1, S2, S3, S4 and Figures 5, 6). The code optimization for both Intel architectures and the fact that the x86 SIMD variant needed three local variables had no influence on the runtime.

With two exceptions, the single threaded Java version was the slowest for all windows. By trend the single precision versions were a bit faster than the double precision calculations, although sometimes, especially for the single threaded versions, double precision was faster than its single precision counterpart.

Plain C was always a bit faster than Java. The runtimes of handwritten scalar assembler lay somewhere between Java and C with two exceptions were handwritten double precision assembler was slower than Java. SIMD was much faster than all other methods. For single precision a maximum speedup with respect to a single threaded Java version of about 15.38 (x86, window 1, SIMD, 4 threads) and for double precision a maximum speedup of about 5.84 (x86, window 1, SIMD, 4 threads) could be achieved. For the x86_64 architecture the maximum speedups were a little bit lower. SIMD single precision instructions were almost twice as fast as double precision instructions.

3.2. Arm based architectures

3.2.1. Arm-v7

Calculations in single precision with Java were slightly slower than double precision (see Tables S3, S6, S7 and Figure 7), whereas for C and handwritten scalar assembler there was not such a difference. SIMD instructions with arm-v7 architecture support only single precision.

A clear explanation for this phenomenon could not be found. Java by default does not convert single precision floating point values to double precision for the standard arithmetic operations (plus, minus, multiplication) if both values have single precision.

As more threads were used, the calculations with Java were performed faster, but the difference in speedup between subsequent numbers of threads became smaller. The decline in speedup depended on the precision and the window: The longer the calculations required, the greater the resulting speedup. The same decline of speedup was observed with the C, scalar assembler and handwritten assembler implementations.

For higher workloads (Window 2 and 3) there is a marked loss of speedup gain if three or more threads are used. This phenomenon can be explained by the fact that the Arm-v7 tablet used has an Exynos 7885 processor, that has two fast Cortex A 73 cores (clock rate 2.3 GHz, out of order execution). The other cores are slower Cortex A 53 processors (clocked at 1.6 GHz, in order execution).

C and handwritten assembler were almost equally fast and always faster than Java by a small amount. SIMD assembler was always much faster than any other method.

The maximum speedups achieved (with respect to a single threaded Java implementation) were 14.58 (SIMD with eight threads) for single precision and 6.44 (C with eight threads) for double precision.

3.2.2. Arm-v8

Java was again slower than any other method. C and assembler were approximately equally fast and both were faster than Java. For all windows SIMD was much faster than all other methods (maximum speedup 11.00 for single precision and 5.98 for double precision with respect to a single-threaded Java version). Again using single precision SIMD instructions was almost twice as fast compared to double precision SIMD instructions.
Figure 5: Median of the runtime of the Mandelbrot algorithm on the virtual Intel x86 tablet. Further explanations see 3.1.

Figure 6: Median of the runtime of the Mandelbrot algorithm on the virtual Intel x86_64 tablet. Further explanations see 3.1.
Figure 7: Median of the runtime of the Mandelbrot algorithm on the Arm-v7 tablet. Further explanations see 3.2.1.

Figure 8: Median of the runtime of the Mandelbrot algorithm on the Arm-v8 tablet. Further explanations see 3.2.2.
3.2.3. Arm-v7 GPU with OpenCL

Table 3 shows the runtime and speedup for different windows, vector sizes and precisions. The first line of each cell shows the mean wall clock time in milliseconds. The second line shows the 95% confidence interval of the mean. If the values were not normally distributed, no confidence interval is displayed and the median, instead of the mean, is shown in the first line. The third line displays the speedup with respect to a single threaded Java version on the GPU (first value) and to the calculation on the GPU with a scalar version of the same window and the same precision (i.e., topmost value in the same column). Measurement of the wall lock time was started in Java immediately prior to the call to the native method that performed the calculations on the GPU.

Half precision worked well for Window 1 but failed to return precise results due to numerical issues for Windows 2 and 3 (see Figure 1d). The results that were rejected are marked grey in Tables 3 and 4.

With respect to Java (single precision) a maximum speedup of 4.65 for half precision and 20.20 for single precision could be achieved. The maximum speedup for half precision was 20.66, but the resulting picture had to be discarded. Greater workloads resulted in a greater speedup.

OpenCL by default offers vector instructions with a length of 2, 4, 8 or 16 elements, even if the GPU natively does not support vectors of all of these sizes. The compiler determines how the vector instructions are implemented. For both precisions tested, scalar operations were faster than any vector operation. For both (single and half precision) performance degraded notably with 16 vector elements. While for the scalar version and Window 1 there was almost no runtime difference between half and single precision, the more elements a vector had and the greater the workload was, the faster the resulting calculations with half instead of single precision.

Table 4 depicts detailed timing of different steps of the algorithm that calculates the Mandelbrot set on the GPU. The same data as in Table 3 was used. The column ‘wall clock time’ is the median runtime measured in Java. The JNI overhead has been measured taking the time at the very beginning and the end of the C method. The lock overhead was the time needed for acquiring the reader writer lock, the semaphore and testing the flag. The OCL overhead is the time needed for setting up and destroying the entire OpenCL environment. This encompasses the compilation of the kernel and the data transfer to and from the GPU.

The JNI overhead is independent of the three variables tested (window, vector size, precision) and is very small (almost always less than 3 milliseconds). The overhead for the locking mechanism is also independent of the three variables and is even smaller (less than 30 microseconds). The OpenCL overhead depends on the workload and the vector size and ranged from 72.99 ms (scalar version, Window 0) to 193.94 ms (vector with 16 elements, Window 2). For smaller workloads the OpenCL framework overhead can attain two thirds of the entire runtime whereas for high workloads the fraction of runtime for the OpenCL framework overhead can drop to less than 10%.

3.2.4. Arm-v7 GPU - Test of the locking mechanisms

Both locking mechanisms described above were implemented and used. The locking events were logged into a file. The app was halted and restarted multiple times. The log showed that the app was working as expected and resources were freed timely after the app had been stopped. The unloading mechanism worked flawlessly.

3.2.5. Arm-v7 GPU in comparison with other GPUs

The median wall clock time of the execution of the OpenCL kernel on the Mali G71 GPU has been compared to an integrated Intel Iris Plus 650, to a and dedicated AMD Radeon VII GPU and to a dedicated Nvidia Titan V GPU. Table 5 in the supplements depicts the results. The same kernel as on
| W | V | P | wall clock time | JNI overhead | JNI % | lock overhead | lock % | OCL overhead | OCL % |
|---|---|---|-----------------|--------------|-------|---------------|-------|--------------|-------|
| 1 | h | s | 98.97           | 0.61         | 0.62% | 0.01          | 0.01% | 73.40        | 74.17%|
|   |   |   | 98.29           | 0.37         | 0.38% | 0.01          | 0.01% | 72.99        | 74.26%|
| 2 | h | s | 111.60          | 0.61         | 0.55% | 0.01          | 0.01% | 84.70        | 75.89%|
|   |   |   | 114.05          | 0.35         | 0.31% | 0.01          | 0.01% | 84.26        | 73.87%|
| 3 | h | s | 108.91          | 0.68         | 0.63% | 0.01          | 0.01% | 85.10        | 78.14%|
|   |   |   | 105.92          | 0.25         | 0.24% | 0.01          | 0.01% | 77.59        | 73.25%|
|   | h | s | 116.12          | 0.36         | 0.49% | 0.01          | 0.01% | 90.04        | 77.54%|
|   |   |   | 126.15          | 0.38         | 0.30% | 0.01          | 0.01% | 94.73        | 75.09%|
|   | h | s | 331.65          | 0.95         | 0.29% | 0.02          | 0.01% | 183.20       | 55.24%|
|   |   |   | 504.44          | 0.40         | 0.08% | 0.02          | 0.00% | 189.24       | 37.51%|
| 4 | h | s | 520.36          | 2.42         | 0.46% | 0.02          | 0.00% | 119.06       | 22.88%|
|   |   |   | 616.49          | 1.00         | 0.16% | 0.02          | 0.00% | 121.26       | 19.67%|
|   | h | s | 568.48          | 1.57         | 0.28% | 0.02          | 0.00% | 132.31       | 23.27%|
|   |   |   | 781.55          | 0.92         | 0.12% | 0.02          | 0.00% | 131.93       | 16.88%|
|   | h | s | 521.65          | 1.54         | 0.29% | 0.02          | 0.00% | 138.14       | 26.48%|
|   |   |   | 793.20          | 1.12         | 0.14% | 0.02          | 0.00% | 138.13       | 17.41%|
|   | h | s | 558.85          | 1.15         | 0.21% | 0.02          | 0.00% | 146.28       | 26.17%|
|   |   |   | 860.08          | 0.98         | 0.11% | 0.02          | 0.00% | 148.34       | 17.25%|
|   | h | s | 2795.65         | 3.00         | 0.11% | 0.03          | 0.00% | 189.25       | 6.77% |
|   |   |   | 10091.85        | 1.00         | 0.01% | 0.02          | 0.00% | 190.92       | 1.89% |
| 5 | h | s | 1418.84         | 2.59         | 0.18% | 0.02          | 0.00% | 120.46       | 8.49% |
|   |   |   | 1339.31         | 1.23         | 0.09% | 0.02          | 0.00% | 122.82       | 9.17% |
|   | h | s | 1521.70         | 1.76         | 0.12% | 0.06          | 0.00% | 132.24       | 8.69% |
|   |   |   | 1703.75         | 0.93         | 0.05% | 0.02          | 0.00% | 133.37       | 7.83% |
|   | h | s | 1309.81         | 1.48         | 0.11% | 0.02          | 0.00% | 137.22       | 10.48%|
|   |   |   | 1685.47         | 1.10         | 0.07% | 0.01          | 0.00% | 138.82       | 8.24% |
|   | h | s | 1326.49         | 1.30         | 0.10% | 0.02          | 0.00% | 147.50       | 11.12%|
|   |   |   | 1830.99         | 0.97         | 0.05% | 0.02          | 0.00% | 149.06       | 8.14% |
|   | h | s | 11917.28        | 1.07         | 0.01% | 0.02          | 0.00% | 191.76       | 1.61% |
|   |   |   | 27448.24        | 1.17         | 0.00% | 0.02          | 0.00% | 193.94       | 0.71% |

Table 4: Median of runtime on Mali G-71 GPU and OpenCL in milliseconds. W=window, V=number of vector elements, P=precision, h=half, s=single. 'wall clock time' = median of wall clock time measured in Java in ms. 'JNI overhead' = median of overhead in ms for JNI interface. 'JNI %' = percentage of 'JNI overhead' with respect to 'wall clock time'. 'lock overhead' = median of overhead in ms for locking mechanism. 'lock %' = percentage of 'lock overhead' with respect to 'wall clock time'. 'OCL overhead' = median of overhead in ms for the setup of the OpenCL environment (creation of command queue, building the program, data transfer and releasing the resources). 'OCL %' = percentage of 'OCL overhead' with respect to 'wall clock time'. Grey color=result not acceptable due to numerical problems.

the tablet has been used, but instead of the Android development environment a plain C program (compiled with GCC 8.2) was used to execute the kernels on the GPUs. Therefore, only the time for the execution of the kernel was tested. On the Intel GPU the program was executed using lightweight virtualization (Docker [21]). Lightweight virtualization is known to have only a very small effect on the runtime [22].

Depending on the workload the Intel GPU is five to ten times faster than the GPU on the tablet. Performance for half precision is significantly better than for single precision. Performance for vectors with 16 entries is significantly worse than other vector sizes. Depending on the workload, the AMD GPU is 50-100 times faster than the GPU on the tablet. There is no performance degradation with larger vector sizes. On the AMD GPU half precision calculations are as fast as single precision calculations. Double precision calculations are two to three times slower than single precision calculations.

Calculations on the Titan V GPU were 120 to 265 times faster than on the GPU of the tablet. Half precision calculations were not available with the OpenCL framework (although the ALUs (arithmetic logical units) of the GPU are capable of performing half precision calculations). Single precision calculations were two to three times faster than double precision calculations.

4. Discussion

C with -O3 optimization and scalar assembler were almost always faster than Java, but the speedup was small. There was no advantage in developing handwritten scalar assembler, especially given the high effort. Perhaps expert level assembler programmers might be able to rearrange the assembler code and increase its performance.
In the case of the Mandelbrot algorithm, calculations can be executed fully in parallel and threads do not need to synchronize. According to Amdahl’s law this should give a high theoretically possible speedup. On the Arm tablets used in this trial the loss of speedup between two and three cores can - in part - be explained with the fact that the scalar versions of the same program use a shorter and faster code, as they do not rely on the overhead of the producer/consumer principle. Moreover, on processors that have unequal cores (e.g., some faster ones and some slower ones) the theoretically possible speedup using multithreading can be much lower than expected, because using more cores results in a higher probability that cores with a lower computing power are used.

Handwritten SIMD assembler was much faster than any other method (except OpenCL). C compilers might not always be able to automatically extract vector parallelism. SIMD instructions are available for all instruction sets currently supported by Android.

If one does not want to rely on auto-vectorization, there are two ways to explicitly use SIMD instructions. The first is the production of assembler code, which may take longer and might be more error prone, but gives full control of the execution of the program. The second is the use of SIMD intrinsic instruction extensions for the C programming language which are available for Intel and ARM CPUs and spare the programmer from using assembler. In both cases program versions for every CPU type and precision needed have to be written.

The OpenMP framework [23] allows to implicitly use vectorization with the #pragma omp simd statements. OpenMP is supported by Android using an appropriate C compiler.

OpenCL kernels are quite easy to write and, contrary to assembler, there is no need to develop versions specific to each architecture. As an extra bonus, the CPU can be used for other purposes while the GPU carries out calculations. OpenCL kernels can be compiled at runtime and therefore can be adapted or programmed at runtime (e.g., the vector size as was done in this trial). Disadvantages are that OpenCL is not officially supported by Android and third party libraries are needed, whose use can be disallowed in future versions of the Android operating system. Vendor provided OpenCL libraries should be compiled in a way that they do not depend on other shared libraries (i.e., linked with static libraries). This avoids compatibility issues. This was the second reason why we were not able to use the OpenCL shared library on the Arm-v8 tablet.

During calculations on the GPU the refresh of the screen might become slower. GPU rendering can be disabled on the device (developer options) or in the app (written in the manifest-XML or in the activity-XML) leaving the entire GPU power for OpenCL. For this trial GPU hardware acceleration was disabled. The very bad performance of vectors with 16 elements may be caused by a lack of registers and the need to store intermediate results in memory. This phenomenon might be specific to the implementation of an algorithm, the precision used and the GPU.

For Java, C and scalar assembler, the selection of the precision is not very important. All platforms used are 64 bit CPUs (the Cortex CPUs of the Arm-v7 tablet would also support Arm-v8). Double precision calculations were even slightly faster than single precision arithmetics on the Arm-v7 tablet with C and Java. SIMD single precision instructions were much faster than double precision, as single precision allows twice as many floating point operations per instruction than double precision.

On the GPU half precision should be preferred over single precision if no numerical issues are expected. Care should be taken regarding the vector size. For half and single precision, scalar operations were fastest. Longer vectors yielded longer runtimes. For small workloads the increase in runtime was almost exclusively caused by a longer compilation time (subtract "OCL overhead" from "wall clock time" in Table 5 see table S10). For higher workloads this effect was less pronounced.

Regarding single precision, for small workloads (Window 1), multithreaded SIMD assembler was faster than the fastest OpenCL version (1.46x). For Window 2 and Window 3 (intermediate and high workload) OpenCL was faster than the fastest SIMD version (1.29x and 1.39x). The overhead for JNI and the locking mechanism in the case of OpenCL were negligible. The overhead for setting up the OpenCL environment can be considerable (up to 79%), if for every call the entire building process has to be executed.

The availability of a pocl (portable computing language) library [24] for Android for the execution of OpenCL kernels on the CPU would be helpful. This would allow to execute the same kernels on the CPU if there were no GPU on the tablet. This would enhance portability and would allow the implicit use of SIMD parallelism on the CPU without the need to write specific assembler versions for each architecture. The use of OpenCL on multicore CPUs in comparison to other parallel programming frameworks has been analyzed by Shen et al. [25].

The results for OpenCL presented here can be applied to other architectures like the Raspberry Pi 3B+, for whose Video-Core IV GPU an (experimental) OpenCL library has been developed. Unfortunately, no OpenCL library exists for the Video-Core VI GPU of the Raspberry Pi 4b.

The use of multiprocessing (fork() in C) has not been investigated as forking new processes on Android is generally not recommended and sharing memory in a multiprocessing environment is more difficult.

5. Conclusions

The following factors were most important for speedup (sorted by the effort needed to implement):

- Multithreading
- SIMD data parallelism
- Use of GPU (via OpenCL)
- Precision (for SIMD and to a lesser extent for OpenCL)

Multithreading is one of the most relevant factors to contribute to a higher speedup on the CPU. Multithreading is well supported by Java and C.
SIMD programming can achieve high speedups, but using data parallelism is not straightforward if one does not want to rely on auto-vectorization (this is valid not only for Android devices).

OpenCL can currently be used on Android devices if the device’s GPU supports this framework. The wrapper library developed for this project allows a comfortable way of accessing the GPU’s resources. The use of OpenCL on Android devices would be easier if OpenCL was officially supported by Android, or at least the possibility of the use of vendor supplied libraries would be guaranteed in the future.

Precision has a less important role for the runtime except for SIMD parallelism on the CPU.

This trial has shown that there are efficient mechanisms to allow OpenCL resources to be freed if the program execution is stopped from outside. Memory can potentially be saved if dynamic libraries are unloaded while an app is not used.

The insights gained here can allow the development of a Java library that implements the locking mechanisms and allows the programmer to use OpenCL without having the need to use C and JNI.

Interesting future developments would be processors that support SIMD instructions with more elements (like the AVX instructions for many Intel processors) and half precision. This would allow doubling of the current speedups for handwritten SIMD assembler, as well as for C compilers and Java interpreters that are able to automatically extract vector parallelism.

On the other hand, more powerful GPUs (currently already available) would allow faster calculations on the GPU. Two GPUs on tablets would decouple the screen related workload from the program arithmetic workload. Using precompiled OpenCL kernels could help to save time.

Not all parallel programming paradigms are suited for all types of problems. The selection of the parallelization type might be driven by the nature of the problem.

Remarks

• This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

• The sequence diagram in the supplements has been created with the online diagram editor available at https://sequencediagram.org/.

• The activity diagrams have been created with the online available diagram editor https://online.visual-paradigm.com/diagrams/solutions/free-activity-diagram-tool/.

• Tables and figures were created using python3 and the libraries scipy, numpy and matplotlib.

References

[1] MontBlanc, https://www.montblanc-project.eu/project, retrieved on july 11th, 2020, 2020.

[2] A. Douady, J. H. Hubbard, Étude dynamique des polynômes complexes, Précépublications mathématiques d’Orsay 2/4 (1984/1985).

[3] A. R. Brodkorb, T. R. Hagen, A comparison of three commodity-level parallel architectures: Multi-core cpu, cell be and gpu, in: Proceedings of the 7th International Conference on Mathematical Methods for Curves and Surfaces, MMCS’08, 2008, p. 70–80.

[4] K. Wang, J. Nurmi, T. Ahonon, Accelerating computation on an android phone with opencl parallelism and optimizing workload distribution between a phone and a cloud service, in: 2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CRDCom/IofP/SmartWorld), 2016, pp. 636–642.

[5] A. Acosta, C. Merino, J. Totz, Analysis of opencl support for mobile gpus on android, in: Proceedings of the International Workshop on OpenCL, IWOCCL ’18, Association for Computing Machinery, 2018, pp. 1–6.

[6] J. A. Ross, D. A. Richie, S. J. Park, D. R. Shires, L. L. Pollock, A case study of opencl on an android mobile gpu, in: 2014 IEEE High Performance Computing Extreme Conference (HPEC), 2014, pp. 1–6.

[7] B. Pérez, E. Stafford, J. Bosque, R. Beivide, S. Mateo, J. Teruel, X. Martorell, E. Ayguadé, Auto-tuned opencl kernel co-execution in ompss for heterogeneous systems, Journal of parallel and distributed computing 125 (2019) 45–57.

[8] B. Pérez, E. Stafford, J. L. Bosque, R. Beivide, S. Mateo, J. Teruel, X. Martorell, E. Ayguadé, Extending ompss for opencl kernel co-execution in heterogeneous systems, in: 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2017, pp. 1–8.

[9] G. Wang, Y. Xiong, J. Yun, J. R. Cavallaro, Accelerating computer vision algorithms using opencl framework on the mobile gpu - a case study, in: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp. 2629–2633.

[10] D. Yokoyama, B. Schulze, F. Borges, et al., The survey on arm processors for hpc, J Supercomput 75 (2019) 7003–7036.

[11] OpenCL, https://www.khronos.org/opencl/, retrieved on march 14th, 2020, 2020.

[12] E. Djikstra, Information streams sharing a finite buffer, Information Processing Letters 1 (1972) 179–180.

[13] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming, Parallel Computing 38 (2012) 391–407.

[14] S. S. Shapiro, M. B. Wilk, An analysis of variance test for normality (complete samples), Biometrika 52 (1965) 591–611.

[15] H. Levene, Robust tests for equality of variances, in: I. Olkin, H. Hotelling, et al. (Eds.), Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling, Stanford University Press, 1960, p. 278–292.

[16] Student, The probable error of a mean, Biometrika 6 (1908) 1–25.

[17] B. L. Welch, The generalization of “Students’s” problem when several different population variances are involved, Biometrika 34 (1947) 28–35.

[18] F. Wilcoxon, Individual comparisons by ranking methods, Biometrics Bulletin 1 (1945) 80–83.

[19] H. Mann, D. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Annals of mathematical Statistics 18 (1947) 50–60.

[20] A. M. Mood, Introduction to the Theory of Statistics, McGraw-Hill, 1950, pp. 394–399.

[21] Docker, https://www.docker.com/, retrieved on october 1st, 2020, 2020.

[22] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance study of opencl on an android mobile gpu, in: 2014 IEEE High Performance Computing Extreme Computing Letters 1 (1972) 179–180.

[23] F. Wilcoxon, Individual comparisons by ranking methods, Biometrics Bulletin 1 (1945) 80–83.

[24] H. Mann, D. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Annals of mathematical Statistics 18 (1947) 50–60.

[25] A. M. Mood, Introduction to the Theory of Statistics, McGraw-Hill, 1950, pp. 394–399.

[26] Docker, https://www.docker.com/, retrieved on october 1st, 2020, 2020.

[27] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers, 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2015) 171–172.

[28] openmp, https://www.openmp.org/, retrieved on august 14th, 2020, 2020.

[29] polcl, http://portablecl.org/, retrieved on july 12th, 2020, 2020.

[30] J. Shen, J. Fang, H. Sips, A. L. Varbanescu, An application-centric evaluation of opencl on multi-core cpus, Parallel Computing 39 (2013) 834–850.
Supplementary material

Figure S1: Sequence diagram of a call to the native OpenCL library on the Android device. Further explanation see 2.4.
| W  | T   | P     | Java       | C         | Assembler | SIMD   |
|----|-----|-------|------------|-----------|-----------|--------|
| 1  | d   | s     | 297.78     | 142.01    | 168.67    | 43.48  |
|    |     |       | (240.86-358.70) | (140.20-143.82) | (166.62-170.72) | (41.65-45.31) |
|    |     |       | [1.00\(^b\), 1.00\(^b\), 1.00\(^b\)] | [2.10\(^b\), 1.00\(^b\), 2.10\(^b\)] | [1.77\(^b\), 1.00\(^b\), 1.77\(^b\)] | [6.88\(^b\), 1.00\(^b\), 6.88\(^b\)] |
| 1  | d   | s     | 191.88     | 145.49    | 166.37    | 80.45  |
|    |     |       | (185.07-198.69) | (143.96-147.01) | (164.34-168.41) | (78.97-81.93) |
|    |     |       | [1.00\(^b\), 1.00\(^b\), 1.00\(^b\)] | [1.32\(^b\), 1.00\(^b\), 1.32\(^b\)] | [1.15\(^b\), 1.00\(^b\), 1.15\(^b\)] | [2.39\(^b\), 1.00\(^b\), 2.39\(^b\)] |
| 2  | d   | s     | 104.20     | 80.28     | 91.13     | 24.75  |
|    |     |       | (102.59-105.82) | (78.84-81.71) | (89.80-92.45) | (23.43-26.08) |
|    |     |       | [1.00\(^b\), 2.28\(^b\), 2.86\(^b\)] | [1.30\(^b\), 1.77\(^b\), 3.71\(^b\)] | [1.14\(^b\), 1.85\(^b\), 3.27\(^b\)] | [4.21\(^b\), 1.76\(^b\), 12.03\(^b\)] |
| 3  | d   | s     | 77.00      | 60.06     | 64.80     | 19.90  |
|    |     |       | (75.80-78.21) | (58.48-61.64) | (63.84-65.76) | (18.78-21.02) |
|    |     |       | [1.00\(^b\), 3.83\(^b\), 3.87\(^b\)] | [1.28\(^b\), 2.36\(^b\), 4.96\(^b\)] | [1.19\(^b\), 2.60\(^b\), 4.60\(^b\)] | [3.87\(^b\), 2.18\(^b\), 14.96\(^b\)] |
| 4  | d   | s     | 69.72      | 53.97     | 58.07     | 19.36  |
|    |     |       | (67.72-71.72) | (52.69-55.26) | (56.51-60.64) | (18.14-20.58) |
|    |     |       | [1.00\(^b\), 2.61\(^b\), 2.61\(^b\)] | [1.24\(^b\), 2.44\(^b\), 3.22\(^b\)] | [1.16\(^b\), 2.61\(^b\), 3.01\(^b\)] | [2.09\(^b\), 2.29\(^b\), 5.45\(^b\)] |
| 1  | d   | s     | 65.45      | 55.24     | 56.21     | 32.88  |
|    |     |       | (63.45-67.45) | (53.73-56.74) | (54.54-57.89) | (31.45-34.32) |
|    |     |       | [1.00\(^b\), 2.93\(^b\), 2.93\(^b\)] | [1.18\(^b\), 2.63\(^b\), 3.47\(^b\)] | [1.16\(^b\), 2.96\(^b\), 3.41\(^b\)] | [1.99\(^b\), 2.45\(^b\), 5.84\(^b\)] |

Table S1: Mean wall clock time with x86 instruction set and SSE3 extensions in ms. W=window, T=number of threads, P=precision, s=single, d=double. Values in curved brackets are the values of the 95% confidence interval. Values in square brackets are the calculated speedups (see text for explanations). Minima and maxima in bold face. (*)=no normal distribution (median instead of mean, no confidence interval), a=t-test, b=Welch-test, c=Mann-Whitney-U test, d=Median test, underline=statistically significant (p<0.05)
### Table S2: Mean wall clock time with x86 instruction set and SSE3 extensions in ms. W=window, T=number of threads, P=precision, s=single, d=double. Values in curved brackets are the values of the 95% confidence interval. Values in square brackets are the calculated speedups (see text for explanations). Minima and maxima in bold face. (*)=no normal distribution (median instead of mean, no confidence interval), a=t-test, b=Welch-test, c=Mann-Whitney-U test, d=Median test, underline=statistically significant (p<0.05)

| W | T | P   | Java    | C   | Assembler | SIMD |
|---|---|-----|---------|-----|-----------|------|
|   |   |     | 12,299.28 | 9,858.81 | 10,238.09 | 3,430.66 |
| 1 |   | s   | (1.00%, 1.00%, 1.00%) | (1.24%, 1.00%, 1.24%) | (1.19%, 1.00%, 1.19%) | (3.56%, 1.00%, 3.56%) |
|   |   | d   | (12,660.48-13,312.44) | (1.29%, 1.00%, 1.29%) | (1.23%, 1.00%, 1.23%) | (2.23%, 1.00%, 2.23%) |
|   |   |     | 12,986.46 | 10,042.16 | 10,544.26 | 5,829.21 |
| 2 |   | s   | (6,721.05-6,857.07) | (5.645.57) | (5,780.18) | 1,963.43 |
|   |   | d   | (7,366.18-7,706.55) | (5.720.70) | (5,898.06) | 3,292.63 |
|   |   |     | 4,900.31 | 4,114.10 | 4,267.82 | 1,550.22 |
|   |   | s   | (1.00%, 2.46%, 2.46%) | (1.21%, 2.40%, 2.97%) | (1.16%, 2.40%, 2.86%) | (3.20%, 2.21%, 7.88%) |
| 3 |   | d   | (5,820.75-6,000.34) | (1.42%, 2.41%, 3.11%) | (1.35%, 2.40%, 2.96%) | (2.17%, 2.14%, 4.76%) |
|   |   |     | 3,911.83 | 3,507.68 | 3,604.62 | 1,276.68 |
|   |   | s   | (3,877.04-3,946.63) | (1.12%, 2.81%, 3.48%) | (1.09%, 2.84%, 3.39%) | (3.06%, 2.69%, 9.56%) |
| 4 |   | d   | (3,406.63-3,510.91) | (1.40%, 2.90%, 3.75%) | (1.36%, 2.98%, 3.67%) | (2.14%, 2.58%, 5.75%) |

### Table S3: Mean wall clock time with x86_64 instruction set and SSE4.2 extensions in ms. W=window, T=number of threads, P=precision, s=single, d=double. Values in curved brackets are the values of the 95% confidence interval. Values in square brackets are the calculated speedups (see text for explanations). Minima and maxima in bold face. (*)=no normal distribution (median instead of mean, no confidence interval), a=t-test, b=Welch-test, c=Mann-Whitney-U test, d=Median test, underline=statistically significant (p<0.05)

| W | T | P   | Java    | C   | Assembler | SIMD |
|---|---|-----|---------|-----|-----------|------|
|   |   |     | 187.56 | 141.91 | 166.88 | 45.11 |
| 1 |   | s   | (182.52-192.60) | (140.09-143.73) | (166.00-167.76) | (43.23-46.98) |
|   |   | d   | (165.73-170.78) | (137.22-140.66) | (171.44-173.03) | (81.37-84.68) |
|   |   |     | 168.26 | 138.94 | 172.24 | 83.03 |
|   |   | s   | (85.63-88.07) | (77.01-79.27) | (89.48-92.07) | (24.41-26.84) |
| 2 |   | d   | (85.63-88.37) | (77.63-80.67) | (92.69-94.82) | (47.75-50.96) |
|   |   |     | 86.85 | 78.14 | 90.77 | 25.63 |
|   |   | s   | (1.00%, 1.93%, 1.93%) | (1.10%, 1.76%, 2.13%) | (0.93%, 1.84%, 1.97%) | (1.76%, 1.68%, 3.41%) |
| 3 |   | d   | (60.86-64.18) | (57.54-60.00) | (63.71-65.73) | (19.24-21.86) |
|   |   |     | 62.52 | 58.77 | 64.72 | 20.55 |
|   |   | s   | (1.00%, 3.00%, 3.00%) | (1.06%, 2.41%, 3.19%) | (0.97%, 2.58%, 2.90%) | (3.04%, 2.19%, 9.13%) |
| 4 |   | d   | (60.94-63.70) | (56.72-59.65) | (65.67-67.76) | (38.09-41.15) |
|   |   |     | 55.41 | 55.73 | 58.17 | 19.11 |
|   |   | s   | (53.88-56.93) | (*) | (56.23-60.11) | (17.95-20.26) |
|   |   | d   | (55.39-59.29) | (52.07-55.73) | (58.18-61.88) | (34.85-38.21) |
|   |   |     | 57.34 | 53.90 | 60.03 | 36.53 |
|   |   | s   | (1.00%, 2.93%, 2.93%) | (1.06%, 2.58%, 3.12%) | (0.96%, 2.82%, 2.80%) | (1.57%, 2.27%, 4.61%) |

S-iii
| W | T | P | Java | Assembler | SIMD |
|---|---|---|-------|----------|------|
| 1 | s | (4,207.62-4,624.75) | (3,765.17-3,877.32) | (3,914.39-4,040.22) | (*) |
|   | (1,000, 1,000, 1,000) | [1,119, 1,000, 1,119] | (1,072, 1,000, 1,072) | [3,161, 1,000, 3,161] | (*) |
|   | d | 4,021.92 | 3,848.30 | 4,170.62 | 2,152.90 |
|   | (3,949.05-4,094.78) | (*| (*) | (*) |
|   | (1,000, 1,000, 1,000) | [1,055, 1,000, 1,055] | [0,961, 1,000, 0,961] | [1,875, 1,000, 1,875] | (*) |
| 2 | s | 2,252.54 | 2,012.16 | (*) | 699.99 |
|   | (1,000, 1,886, 1,886) | [1,126, 1,906, 2,116] | [1,055, 1,856, 1,926] | [3,224, 1,926, 6,054] | (*) |
|   | d | 2,338.51 | 2,318.18 | 2,441.74 | 1,330.98 |
|   | (*| (*) | (*) | (*) |
| 3 | s | 1,812.41 | 1,530.58 | 1,635.36 | 493.53 |
|   | (1,790.54-1,834.28) | (*| (*) | (*) |
|   | (1,000, 2,344, 2,344) | [1,118, 2,506, 2,776] | [1,086, 2,186, 2,086] | [3,676, 2,726, 8,586] | (*) |
|   | d | 2,098.70 | 1,795.00 | 1,935.61 | 1,128.83 |
|   | (1,000, 1,926, 1,926) | [1,117, 2,146, 2,246] | [1,085, 2,185, 2,085] | [3,656, 1,916, 3,566] | (*) |
| 4 | s | 1,495.92 | 1,279.85 | 1,328.15 | 366.85 |
|   | (1,000, 2,833, 2,833) | [1,117, 2,996, 3,311] | [1,131, 2,996, 3,196] | [4,086, 3,666, 11,556] | (*) |
|   | d | 1,872.47 | 1,476.92 | 1,403.56 | 925.22 |
|   | (1,000, 2,155, 2,155) | [1,126, 2,616, 2,726] | [1,132, 2,976, 2,876] | [2,026, 2,333, 4,353] | (*) |
| 1 | s | 11,326.66 | 9,512.57 | 10,037.57 | 3,276.98 |
|   | (11,245.88-11,407.43) | (9,458.47-9,566.66) | (11,133.41-11,143.31) | (3,466, 1,000, 3,466) | (*) |
|   | (1,000, 1,000, 1,000) | [1,199, 1,000, 1,199] | (1,136, 1,000, 1,136) | (3,466, 1,000, 3,466) | (*) |
|   | d | 11,575.91 | 9,524.42 | 10,492.35 | 5,550.86 |
|   | (11,396.59-11,759.23) | (9,458.47-9,566.66) | (11,133.41-11,143.31) | (3,466, 1,000, 3,466) | (*) |
| 2 | s | 5,825.26 | 5,553.59 | 5,762.30 | 1,786.84 |
|   | (5,745.91-5,910.62) | (5,460.63-5,646.54) | (5,693.84-5,830.76) | (3,266, 1,833, 6,343) | (*) |
|   | (1,000, 1,944, 1,944) | [1,105, 1,714, 2,044] | [1,014, 1,744, 1,974] | (2,096, 1,000, 2,096) | (*) |
|   | d | 6,275.84 | 5,648.08 | 5,924.89 | 3,349.58 |
|   | (1,000, 1,844, 1,844) | [1,111, 1,694, 2,054] | [1,066, 1,764, 1,954] | [1,876, 1,666, 3,466] | (*) |
| 3 | s | 4,229.38 | 3,951.35 | 4,176.13 | 1,356.72 |
|   | (1,000, 2,686, 2,686) | [1,073, 2,414, 2,874] | [1,014, 2,404, 2,714] | [3,124, 2,424, 8,354] | (*) |
|   | d | 5,158.15 | 3,955.32 | 4,392.99 | 2,826.95 |
|   | (1,000, 2,244, 2,244) | [1,300, 2,414, 2,934] | [1,177, 2,384, 2,644] | [1,828, 1,966, 4,104] | (*) |
| 4 | s | 3,405.69 | 3,069.36 | 3,254.17 | 1,087.24 |
|   | (1,000, 3,333, 3,333) | [1,111, 3,104, 3,604] | [1,056, 3,084, 3,484] | [3,124, 3,014, 10,424] | (*) |
|   | d | 3,316.69 | 3,250.89 | 3,619.40 | 2,457.99 |
|   | (1,000, 2,686, 2,686) | [1,339, 2,923, 3,584] | [1,192, 2,904, 3,204] | [1,766, 2,264, 4,714] | (*) |

Table S4: Mean wall clock time with x86_64 instruction set and SSE4.2 extensions in ms. W=window, T=number of threads, P=precision, s=single, d=double. Values in curved brackets are the values of the 95% confidence interval. Values in square brackets are the calculated speedups (see text for explanations). Minima and maxima in bold face. (*)=no normal distribution (median instead of mean, no confidence interval), a=t-test, b=Welch-test, c=Mann-Whitney-U test, d=Median test, underline=statistically significant (p<0.05)
| W | T   | P     | Java (ms) | C (ms) | Assembler (ms) | SIMD (ms) |
|---|------|-------|-----------|--------|----------------|-----------|
| 1 |      |       | 457.35    | 330.48 | 345.01         | 118.71    |
|   | s    | (450.39-464.31) | (326.90-334.07) | (341.74-348.28) | (115.04-122.38) |
|   |      | [1.00*, 1.00*, 1.00*] | [1.38*, 1.00*, 1.38*] | [1.33*, 1.00*, 1.33*] | (3.85*, 1.00*, 3.85*) |
|   | d    | (433.97-474.34) | (327.26-343.40) | (342.51-349.57) |
|   |      | [1.00*, 1.00*, 1.00*] | [1.37*, 1.00*, 1.37*] | [1.31*, 1.00*, 1.31*] |
| 2 |      |       | 454.16    | 330.83 | 346.04         | -         |
|   |      | (433.97-474.34) | (327.26-343.40) | (342.51-349.57) |
|   |      | [1.00*, 1.00*, 1.00*] | [1.37*, 1.00*, 1.37*] | [1.31*, 1.00*, 1.31*] |
| 3 |      |       | 353.69    | 273.86 | 274.06         | 130.55    |
|   | s    | (329.51-377.87) | (258.48-289.23) | (258.90-289.21) | (125.50-135.61) |
|   |      | [1.00*, 1.20*, 1.20*] | [1.20*, 1.24*, 1.65*] | [1.20*, 1.26*, 1.67*] | (2.71*, 0.91*, 3.50*) |
|   | d    | (261.31-295.12) | (259.30-291.58) | (263.22-296.62) |
|   |      | [1.00*, 1.00*, 1.00*] | [1.02*, 1.20*, 1.65*] | [1.00*, 1.24*, 1.62*] |
| 4 |      |       | 235.47    | 200.68 | 197.73         | 103.09    |
|   | s    | (195.59-219.87) | (163.82-168.40) | (195.59-219.87) | (99.59-106.79) |
|   |      | [1.00*, 1.94*, 1.94*] | [1.17*, 1.65*, 2.28*] | [1.19*, 1.74*, 2.34*] | (2.28*, 1.15*, 4.44*) |
|   | d    | (213.45-216.83) | (195.15-198.69) | (*) |
|   |      | [1.00*, 2.11*, 2.11*] | [1.09*, 1.68*, 2.31*] | [1.08*, 1.74*, 2.29*] |
| 5 |      |       | 201.35    | 172.98 | 171.12         | 87.49     |
|   | s    | (*) | (*) | (*) | (84.30-90.67) |
|   |      | [1.00*, 2.27*, 2.27*] | [1.16*, 1.91*, 2.64*] | [1.18*, 2.02*, 2.67*] | (2.30*, 1.36*, 2.30*) |
|   | d    | (*) | (*) | (*) |
|   |      | [1.00*, 2.40*, 2.40*] | [1.10*, 1.92*, 2.63*] | [1.10*, 2.02*, 2.65*] |
| 6 |      |       | 181.12    | 155.41 | 152.47         | 77.46     |
|   | s    | (*) | (*) | (*) | (69.48-75.88) |
|   |      | [1.00*, 2.53*, 2.53*] | [1.17*, 2.13*, 2.94*] | [1.19*, 2.26*, 3.00*] | (2.34*, 1.53*, 5.90*) |
|   | d    | (*) | (*) | (*) |
|   |      | [1.00*, 2.62*, 2.62*] | [1.13*, 2.16*, 2.97*] | [1.13*, 2.25*, 2.95*] |
| 7 |      |       | 173.55    | 153.03 | 153.84         | -         |
|   | s    | (*) | (*) | (*) | (66.26-71.45) |
|   |      | [1.00*, 2.35*, 2.35*] | [1.17*, 2.34*, 3.23*] | [1.17*, 2.51*, 3.32*] | (2.29*, 1.63*, 6.29*) |
|   | d    | (*) | (*) | (*) |
|   |      | [1.00*, 2.80*, 2.80*] | [1.18*, 2.41*, 3.30*] | [1.18*, 2.50*, 3.29*] |
| 8 |      |       | 156.59    | 134.14 | 129.09         | 68.86     |
|   | s    | (*) | (*) | (*) | (66.17-72.67) |
|   |      | [1.00*, 2.92*, 2.92*] | [1.17*, 2.46*, 3.41*] | [1.21*, 2.67*, 3.54*] | (2.27*, 1.72*, 6.64*) |
|   | d    | (*) | (*) | (*) |
|   |      | [1.00*, 3.05*, 3.05*] | [1.17*, 2.58*, 3.57*] | [1.18*, 2.73*, 3.61*] | (2.16*, 1.71*, 6.59*) |

Table S5: Mean wall clock time with Arm-v7 instruction set and NEON extensions in ms. W=window, T=number of threads, P=precision, s=single, d=double. Values in curved brackets are the values of the 95% confidence interval. Values in square brackets are the calculated speeds (see text for explanations). Minima and maxima in bold face. (*)=no normal distribution (median instead of mean, no confidence interval). a=t-test, b=Welch-test, c=Mann-Whitney-U test, d=Median test, underline=statistically significant (p<0.05). For armv7 there are no double precision SIMD instructions and support for single precision SIMD instructions is optional.
| W | T | P | Java | C | Assembler | SIMD |
|---|---|---|------|---|------------|------|
| 1 | s | (10,203.80-10,400.13) | (7,502.93-7,510.08) | (7,693.94-7,700.89) | (3,102.07-3,109.93) |
|   |   | (1,000, 1,000, 1,000) | (1,37", 1,000, 1,37") | (1,34", 1,000, 1,34") | (3,32", 1,000, 3,32") |
| 2 | s | (9,782.54-10,468.99) | (7,526.72-7,533.80) | (7,790.32-7,797.14) | 1,856.78 |
|   |   | (1,000, 1,000, 1,000) | (1,34", 1,000, 1,34") | (1,30", 1,000, 1,30") | (*) |
| 3 | s | (4,995.67-5,376.21) | (4,119.84-4,575.35) | (4,347.56-5,133.14) | (1,716.11-1,997.44) |
|   |   | (1,000, 1,999, 1,999) | (1,19", 1,73", 2,37") | (1,09", 1,62", 2,17") | (2,79", 1,67", 5,55") |
| 4 | s | (4,014.28-4,571.48) | (4,009.02-4,619.85) | (4,016.75-4,377.76) | (*) |
|   |   | (1,000, 2,36", 2,36") | (1,000, 1,73", 2,35") | (1,002, 1,86", 2,41") | (*) |
| 5 | s | (3,249.96-3,268.31) | (3,130.69-3,178.37) | (3,221.52-3,233.33) | (*) |
|   |   | (1,000, 3,11", 3,11") | (1,03", 2,39", 2,31") | (1,01", 2,41", 3,14") | (*) |
| 6 | s | (2,819.69 | (2,667.35 | (2,669.91 | (*) |
|   |   | (1,000, 3,59", 3,59") | (1,06", 2,82", 3,00") | (1,06", 2,92", 3,79") | (*) |
| 7 | s | (2,868.80 | (2,341.53 | (2,317.27 | (*) |
|   |   | (1,000, 3,59", 3,59") | (1,23", 3,21", 4,40") | (1,24", 2,41", 3,23") | (2,86", 2,23", 7,41") |
| 8 | s | (2,481.74 | (2,294.46 | (2,331.88 | (*) |
|   |   | (1,000, 4,08", 4,08") | (1,08", 3,28", 4,41") | (1,06", 3,34", 4,34") | (*) |
| 9 | s | (2,470.37 | (2,090.37 | (2,068.90 | (*) |
|   |   | (1,000, 4,17", 4,17") | (1,18", 3,59", 4,93") | (1,23", 3,83", 5,13") | (2,65", 3,33", 11,04") |
| 10 | s | (2,222.16 | (2,047.90 | (2,017.07 | (*) |
|   |   | (1,000, 4,56", 4,56") | (1,09", 3,68", 4,94") | (1,10", 3,84", 5,00") | (*) |
| 11 | s | (2,199.47 | (1,881.40 | (1,803.05 | (*) |
|   |   | (2,194.74-2,204.19) | (1,17", 3,99", 5,48") | (1,22", 4,27", 5,71") | (2,59", 3,66", 12,15") |
| 12 | s | (1,996.40 | (1,852.41 | (1,824.95 | (*) |
|   |   | (1,990.50-2,002.29) | (1,08", 4,07", 5,47") | (1,09", 4,27", 5,55") | (*) |
| 13 | s | (2,065.79 | (1,688.52 | (1,638.14 | (*) |
|   |   | (2,057.37-2,074.21) | (1,682.68-1,694.36) | (1,626.65-1,649.63) | (793.82-801.27) |
| 14 | s | (1,928.05 | (1,686.80 | (1,669.85 | (*) |
|   |   | (1,905.01-1,935.10) | (1,676.47-1,697.13) | (1,647.10-1,674.61) | (2,59", 3,80", 12,92") |

Table S6: Mean wall clock time with Arm-v7 instruction set and NEON extensions in ms. W=window, T=number of threads, P=precision, s=single, d=double. Values in curved brackets are the values of the 95% confidence interval. Values in square brackets are the calculated speedups (see text for explanations). Minima and maxima in bold face. (*)=no normal distribution (median instead of mean, no confidence interval), *=t-test, b=Welch-test, c=Mann-Whitney-U test, d=Median test, underline=statistically significant (p<0.05). For armv7 there are no double precision SIMD instructions and support for single precision SIMD instructions is optional.
Table S7: Mean wall clock time with Arm-v7 instruction set and NEON extensions in ms. Window, W = number of threads, P = precision, s = single, d = double. Values in curved brackets are the values of the 95% confidence interval. Values in square brackets are the calculated speedups (see text for explanations). Minima and maxima in bold face. (*) = no normal distribution (median instead of mean, no confidence interval), * = t-test, t = Welch-test, c = Mann-Whitney-U test, d = Median test, underline = statistically significant (p<0.05). For armv7 there are no double precision SIMD instructions and support for single precision SIMD instructions is optional.
Table S8: Mean wall clock time with Arm-v8 instruction set and NEON extensions in ms. W = window, T = number of threads, P = precision, s = single, d = double. Values in curved brackets are the values of the 95% confidence interval. Values in square brackets are the calculated speedups (see text for explanations). Minima and maxima in bold face. (*) = no normal distribution (median instead of mean, no confidence interval), = t-test, = Welch-test, = Mann-Whitney-U test, = Median test, underline = statistically significant (p<0.05)

| W | T | P | Java | C | Assembler | SIMD |
|---|---|---|------|---|-----------|------|
| 1 |   | s | 912.26 | 788.26 | 799.85 | 285.15 |
|   |   |   | [891.92-932.60] | [786.74-789.78] | [797.02-802.68] | [283.63-286.67] |
|   |   | d | 916.20 | 788.59 | 800.82 | 567.00 |
|   |   |   | [887.49-944.91] | [786.51-790.67] | [796.84-804.81] | [564.33-569.67] |
| 2 |   | s | 465.39 | 406.36 | 408.50 | 151.76 |
|   |   |   | [449.52-481.26] | [399.36-413.36] | [400.41-416.58] | [143.96-159.56] |
|   |   | d | 468.27 | 406.95 | 409.02 | 290.05 |
|   |   |   | [448.81-487.73] | [400.20-413.69] | [400.82-417.21] | [283.79-296.31] |
| 3 |   | s | 314.66 | 278.34 | 277.90 | 107.34 |
|   |   |   | [304.09-325.24] | [268.74-287.94] | [267.68-288.12] | [97.83-117.24] |
|   |   | d | 313.31 | 279.42 | 278.21 | 199.47 |
|   |   |   | [303.87-322.74] | [268.80-290.03] | [268.66-287.67] | [190.76-208.19] |
| 4 |   | s | 243.80 | 218.95 | 216.42 | 86.51 |
|   |   |   | [232.54-255.06] | [206.83-231.06] | [204.82-228.02] | [77.14-95.88] |
|   |   | d | 235.75 | 218.22 | 221.16 | 158.92 |
|   |   |   | [233.89-257.61] | [206.22-230.23] | [208.29-234.03] | [147.96-169.89] |

S-viii
Table S9: Mean wall clock time with Arm-v8 instruction set and NEON extensions in ms. W=window, T=number of threads, P=precision, s=single, d=double.
Values in curved brackets are the values of the 95% confidence interval. Values in square brackets are the calculated speedups (see text for explanations). Minima and maxima in bold face. (*)=no normal distribution (median instead of mean, no confidence interval), a=t-test, b=Welch-test, c=Mann-Whitney-U test, d=Median test, underline=statistically significant (p<0.05)

| W | T   | P                | Java        | C        | Assembler | SIMD        |
|---|-----|------------------|-------------|----------|-----------|-------------|
|   | 1   | s 53,794.44      | 48,078.46   | 47,207.33| 19,202.21 |
|   |     | [1.00, 1.00, 1.00] | [1.12, 1.00, 1.12] | [1.14, 1.00, 1.14] | [2.80, 1.00, 2.80] |
|   | 2   | d 55,057.05      | 48,072.52   | 48,075.06| 36,392.40 |
|   |     | [53,622.46-56,491.64] | [48,066.64-48,083.48] | [36,383.03-36,401.77] | (*) |
|   | 3   | s 28,865.83      | 24,125.08   | 23,705.75| 9,657.86  |
|   |     | [28,859.39-28,872.26] | [24,116.87-24,133.28] | [23,695.49-23,716.10] | (9,651.47-9,664.24) |
|   | 4   | d 27,693.69      | 24,128.04   | 24,131.75| 18,273.23 |
|   |     | [26,844.37-26,853.01] | [24,103.35-24,133.93] | [24,124.87-24,138.64] | (18,264.60-18,281.86) |
|   |     | s 19,316.16      | 16,146.52   | 15,860.33| 6,470.98  |
|   |     | [19,305.75-19,326.57] | [16,136.21-16,156.83] | [15,851.05-15,869.61] | (6,462.17-6,479.78) |
|   |     | d 13,735.50      | 12,164.01   | 12,150.13| 9,203.69  |
|   |     | [13,735.03-13,769.97] | [12,140.61-12,187.42] | [12,133.91-12,166.35] | (9,190.81-9,216.57) |

S-ix
Table S10: Comparison of the wall clock time in milliseconds of the GPU on the Arm-v7 tablet, an integrated Intel Iris Plus 650 GPU, a dedicated AMD Radeon VII GPU and a dedicated Nvidia Titan V GPU. \( W \) = window, \( V \) = number of vector elements, \( P \) = precision, \( h \) = half, \( s \) = single, \( d \) = double. \( n/a \) = not available. 'wall clock time' = median of the wall clock time for the execution of the kernel (\texttt{clEnqueueNDRangeKernel}) on the specified GPU in ms. 'speedup' = speedup with respect to the leftmost column. For the execution on the Intel and the AMD GPUs the same OpenCL kernel as for the tablet was used. Instead of the Android development environment a program written in C (GCC 8.2, -O3) was chosen to prepare, enqueue and execute the kernel. For the Intel GPU lightweight virtualization (Docker) has been used. Due to the different execution environments, only the time for the execution of the kernel can be compared. Although the Titan V GPU is able to perform half precision calculations, this capability is not available with the OpenCL framework. Grey color = result not acceptable due to numerical problems.

| W | V | P | Mali-G71 MP 2 | Intel Iris Plus 650 | AMD Radeon VII | Nvidia Titan V | speedup |
|---|---|---|-------------|-----------------|----------------|----------------|--------|
| 1 | h | s | 24.94       | 4.90           | 5.09           | 0.49           | 51.26  |
|   | s | d | 24.92       | 5.84           | 4.27           | 0.40           | 62.51  |
|   | n/a |   | 15.61       | 1.19           |               | 0.20           | 124.28 |
| 2 | h | s | 26.27       | 4.20           | 6.26           | 0.50           | 52.12  |
|   | s | d | 29.43       | 5.87           | 5.01           | 0.67           | 43.65  |
|   | n/a |   | 16.53       | 1.26           |               | 0.21           | 137.30 |
| 4 | h | s | 23.11       | 3.98           | 5.81           | 0.55           | 42.09  |
|   | s | d | 28.07       | 5.87           | 4.78           | 0.66           | 42.74  |
|   | n/a |   | 16.67       | 1.67           |               | 0.21           | 131.75 |
| 8 | h | s | 25.51       | 4.17           | 6.11           | 0.65           | 39.46  |
|   | s | d | 31.03       | 5.72           | 5.42           | 0.57           | 54.26  |
|   | n/a |   | 15.73       | 1.66           |               | 0.22           | 143.91 |
| 16| h  | s | 147.47      | 6.59           | 22.37          | 0.79           | 186.74 |
|   | s | d | 314.79      | 6.33           | 49.70          | 0.83           | 381.55 |
|   | n/a |   | 119.49      | 2.14           |               | 0.27           | 1161.23|
| 1 | h  | s | 398.87      | 80.93          | 4.93           | 5.89           | 67.74  |
|   | s | d | 494.20      | 118.81         | 4.16           | 6.55           | 75.46  |
|   | n/a |   | 150.19      | 19.93          |               | 1.99           | 247.87 |
| 2 | h  | s | 434.58      | 74.53          | 5.83           | 7.32           | 59.36  |
|   | s | d | 648.68      | 120.15         | 5.40           | 8.14           | 79.72  |
|   | n/a |   | 157.57      | 20.26          |               | 2.58           | 251.22 |
| 4 | h  | s | 381.96      | 70.26          | 5.44           | 8.39           | 45.53  |
|   | s | d | 653.93      | 125.87         | 5.20           | 8.68           | 75.35  |
|   | n/a |   | 161.88      | 21.49          |               | 2.68           | 244.06 |
| 8 | h  | s | 411.40      | 76.08          | 5.41           | 10.19          | 40.36  |
|   | s | d | 710.74      | 125.70         | 5.65           | 9.16           | 77.56  |
|   | n/a |   | 158.04      | 22.22          |               | 3.16           | 225.08 |
| 16| h | s | 2603.37     | 117.84         | 22.09          | 11.85          | 219.71 |
|   | s | d | 9899.92     | 159.01         | 62.26          | 9.30           | 1064.67|
|   | n/a |   | 331.79      | 23.06          |               | 3.40           | 2913.67|
| 1 | h  | s | 1295.78     | 130.21         | 9.95           | 15.48          | 83.73  |
|   | s | d | 1215.24     | 144.37         | 8.42           | 14.77          | 82.30  |
|   | n/a |   | 331.79      | 37.00          |               | 4.70           | 258.73 |
| 2 | h  | s | 1387.64     | 128.49         | 10.80          | 16.72          | 82.99  |
|   | s | d | 1569.44     | 149.78         | 10.48          | 16.19          | 96.94  |
|   | n/a |   | 376.49      | 35.74          |               | 6.01           | 261.14 |
| 4 | h  | s | 1177.66     | 130.73         | 9.01           | 17.33          | 67.95  |
|   | s | d | 1545.54     | 153.75         | 10.93          | 15.95          | 105.41 |
|   | n/a |   | 382.03      | 36.54          |               | 6.48           | 259.49 |
| 8 | h  | s | 11724.42    | 182.80         | 64.14          | 18.53          | 632.87 |
|   | s | d | 27253.11    | 192.10         | 141.87         | 16.09          | 1694.22|
|   | n/a |   | 3684.83     | 37.94          |               | 7.90           | 3450.04|

S-x