DOI: 10.2507/27th.daaam.proceedings.091

OPTIMIZING HIGH-PERFORMANCE CUDA DSP FILTER FOR ECG SIGNALS

Ervin Domazet, Marjan Gusev & Sasko Ristov

This Publication has to be referred as: Domazet, E[rvin]; Gusev, M[arjan] & Ristov, S[asko] (2016). Optimizing High-Performance CUDA DSP Filter for ECG Signals, Proceedings of the 27th DAAAM International Symposium, pp.0623-0632, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-08-2, ISSN 1726-9679, Vienna, Austria
DOI: 10.2507/27th.daaam.proceedings.091

Abstract

Digital signal processing is usually used in biomedical engineering research area. ECG digital signal filtering serves as a basis for further data analysis, with the intention to eliminate noise. Noise can stem from electrical switching power, radio waves, physical movement and breathing muscle artifacts. Feature extraction and further analysis are successful on noise-free ECG signal.

In many critical times, real-time processing of ECG signal can save lives. In case thousands of connected sensors transfer ECG signals to the Cloud Data Center, processing needs to be fast in order real-time analysis to be done. In this paper, we introduce several optimization approaches to improve the parallel CUDA algorithm for digital signal filtering on GPU cores. Our approach utilizes the shared memory for signal segments and for the impulse response coefficients. Additionally, we consider using constant memory for the impulse response coefficients. Other optimization approaches, including loop unrolling, smaller calculation precision and combined effects were tested and the best solution is discussed. The provided analysis indicates that using shared memory for signal segments optimized the code for 13-78% proportional to kernel length. Considering shared memory for the impulse response coefficients increased the performance on average for 10%, whereas constant memory for 15%. There is a 1-5% performance gain when loop unrolling is considered. Using smaller calculation precision speeds up the code by a factor of 1. The best combination of the proposed optimizations yields nearly 6 times faster code compared to the naïve parallel code.

Keywords: DSP; ECG; Heart Signal; Optimization; Parallelization; CUDA; GPGPU.

1. Introduction

Digital Signal Processing (DSP) area has introduced a powerful set of tools to deal with digital signals, and some of them can be successfully applied to the analysis of ECG signals. Being one of the most important tools of the DSP area, filtering can be used for noise elimination and further on for extraction of signal features. Thus, main features of ECG can be detected and extracted from the hidden information, such as alternating changes of the wave amplitude and subtle deviation of the heart rhythm.

Potential heart risks can be detected before some time making the real-time processing a critical task. This, in turn, can save lives [3]. Sequential algorithms are insufficient of processing ECG signals real time, especially in the case of a

data center intended for real-time processing and analyzing thousands of connected wearable ECG sensors. Our motivation is, therefore, to optimize the high-performance solution for signal processing.

There are alternatives for parallelizing the convolution, such as shared memory OpenMP parallelism, distributed MPI parallelism, Maxeler dataflow parallelism, General Purpose GPU CUDA parallelism and etc.

In this research, we focus on optimizing the parallel version of the filtering algorithm on graphics processing unit (GPU) cores. The goal is to find an optimized solution that speed ups the parallel CUDA solution. The hypothesis set in this paper is to confirm whether the utilization of shared and constant memories on GPU can yield faster execution times on ECG signal filtering. To test the hypothesis we will measure the execution times of naive GPU solution over the optimized solution. We are also interested in determining whether loop unrolling and precision has an effect on the speedup. Moreover, we think it would be worthy to investigate the performance of the element version.

We are aware of the limitations, especially that there is a limit on CUDA-enabled GPU's, on the number of threads per blocks due to the internal synchronization inside a block. Considering the advancements on the GPU arena, we hope that this limit will increase. Yet in ECG signal filtering, current limits on the filter lengths are sufficient.

The paper is organized as follows. The background about ECG signals, DSP processing and CUDA obstacles for the high-performance computing are presented in Section 2. Section 3 presents the optimization methods, while Section 4 the experimental methodology. Obtained results of measured processing time and performance analysis of the optimized code are presented in Section 5. Section 6 discusses the obtained results and compares to other approaches. Finally, the paper is concluded and future work is elaborated in Section 7.

2. Background

This section presents a brief description of ECG signals, DSP processing, and CUDA obstacles in order to better understand the paper.

2.1 ECG signal essentials and DSP processing

Electrocardiogram (ECG) signal contains information about the complex heart condition. It is a collection of the electrical activity of the beating heart muscle over a period of time. The electrical activities are captured over time via attached electrodes, measuring the electrical potential reaching to the human skin. These are detected as tiny electrical signals and then amplified and represented on a specialized paper format.

Since the level of ECG signals is very low, there is a considerable amount of accompanied noise, mostly due to amplification of tiny electrical signals. Radio waves, internal human breathing and physical movement are considered as secondary sources of noise. For precise data analysis and interpretation, data pre-processing is necessary.

ECG signal is the representation of impulses generated by beating heart muscle over a period of time (Figure 1) and generally consists 5 features, marked as P, Q, R, S, and T.

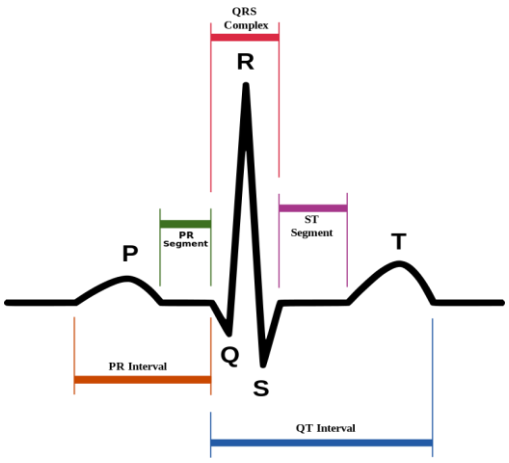

Fig. 1. A general representation of an ECG signal [6]

Characteristic signal of an ECG holds representative and significant information regarding the complex cardiovascular condition of the heart. The general methodology for processing and analyzing ECG signals consist of data pre-processing, feature space reduction and feature extraction steps [3,10]. In this research, we explicitly target the data pre-processing phase.

The power of DSP field rapidly increased during last decades [5]. It is possible to underline that today's most complex systems are built on top of DSP tools. In this research, we are focused on data pre-processing phase of ECG signal analysis.

Convolution, as a mathematical operation, combines the *input data stream x(i)* and *impulse response coefficients h(i)* to generate a new *output data stream y(i)*, for $i = 0, \ldots$ by the calculation expressed in Equation (1).

$$y(i) = \sum_{j=0}^{M-1} h(j)\, x(i-j) \tag{1}$$

Noise elimination on ECG signal is achieved by using three classic filters: low-pass, high-pass and band-pass filters. The main functionality of low-pass filters is to weaken all the frequencies above the cut-off frequency, known as a *stopband* and pass all frequencies below the *passband* [5]. On the other side, high-pass filters are the exact opposite of low-pass. The combination of low-pass and high-pass filters is considered as a band-pass filter, which generally passes a predefined region of frequencies and eliminates other frequencies.

Figure 2 shows the effect of applying all these filters on the input signal. For example, a low-pass filter with a 30 Hz cut-off frequency eliminates the 50Hz noise caused by the electricity. A high-pass filter eliminates the baseline drift, which is caused by breathing and physical movement. A band-pass filter by its definition is the combination of low and high-pass filters and combines both the eliminations of high-frequency noise and baseline drift.

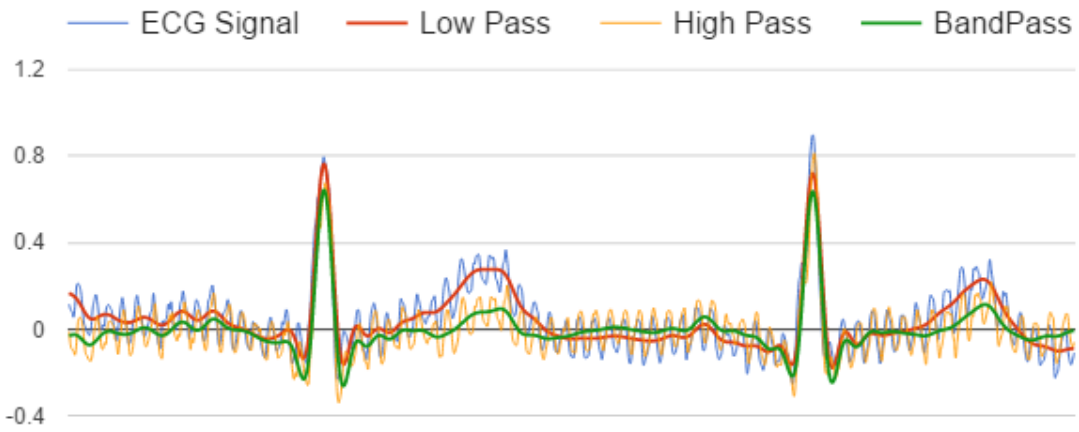

Fig. 2. ECG signal segment with two QRS complexes, filtered with a low-pass filter of 30Hz, a high-pass filter of 0.5Hz and a band pass filter between 0.5Hz and 30Hz.

2.2 CUDA Background

Compute Unified Device Architecture (CUDA) is an application programming interface (API) model developed by NVIDIA. This technology ensures a significant increase in the software performance by using the massive computation power of GPU.

CUDA enables massively parallel hardware designed to run generic or non-graphic code on a general purpose GPU. The goal of CUDA is to write code that can run on compatible massively parallel SIMD architectures. A GPU consists of thousands of low-frequency cores, intended for efficiently handling multiple similar tasks simultaneously.

2.3 Identifying CUDA GPU obstacles for high-performance convolution

CUDA allows massive parallelism, which should be utilized in an efficient way. In order to design an optimized code, it is important to optimize memory alignment and thus accesses. Generally, most important performance consideration is the coalescing of the memory accesses. Figure 3 illustrates the coalescing concept.

Fig. 3. Coalesced access to memory - all threads access one cached line.

Moreover, utilization of shared memory is another performance improvement. Shared memory is located on-chip and generally provides much higher bandwidth and lower latency than the global memory [4]. This is valid especially when there are no bank conflicts. When multiple addresses of a memory request values of the same memory bank, the accesses are serialized, which degrades the performance significantly.

Constant memory is another CUDA improvement that enables a fast access to data. Particularly, consecutive reads of the same address do not incur any additional memory traffic. A single read from constant memory can broadcast to other nearby threads. This will ensure that no bank conflict will occur in the case of access to constant memory.

A high-performance solution will include a combination of the shared memory and constant memory. Since the filter kernel coefficients are not changing over the execution, a natural solution will be to use the shared memory for storing the input signal and constant memory for the filter kernel, as presented in Figure 4.

Another obstacle is the bank conflict that may occur while accessing the data in the shared memory. An example of 2-way bank conflict is illustrated in Figure 5.

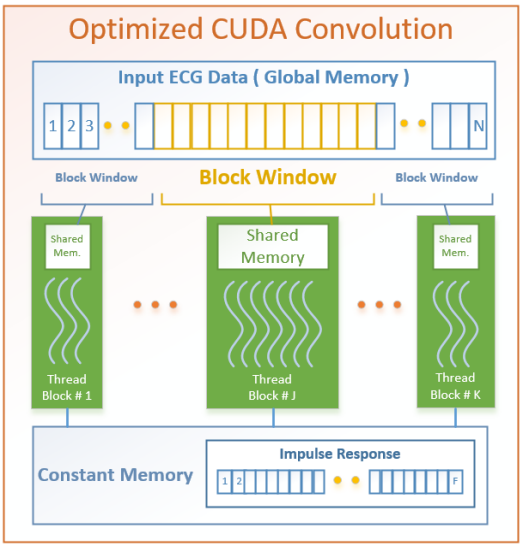

Fig. 4. Data flow in a solution that uses both the shared and constant memory.

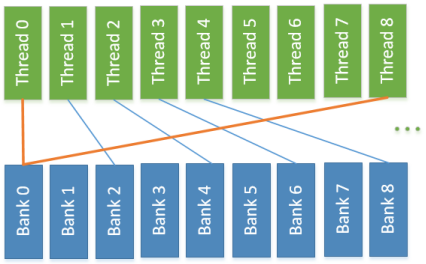

Fig. 5. A two-level bank conflict.

The solution needs a highly accurate filter and fast processing. Therefore the FIR filter is the best candidate for the filtering. A problem will occur if the filter length is bigger than the maximum number of threads per block to eliminate propagation of partial results and introducing internal synchronization inside a block. Currently, NVIDIA devices have 1024 threads per block. In our case, to process an ECG signal, a FIR filter length of 1000 is acceptable, since it generates 30*db* attenuation with a relatively small ripple passband of 0.1db.

3. Optimization approaches

A sequential algorithm for performing convolution is an iterative procedure that repeats the kernel item for each new data element. The complexity of the algorithm *O(nM)* depends on the input *n* and the kernel stream length *M*. The flow on CPU is sequential, where inner loop length depends on the kernel size.

In our previous research [2], we have used a naïve parallel CUDA version of the DSP filtering algorithm running on a GPU. Since it did not include any optimization, we have faced lower performance due to lots of concurrent memory reads. In this research, we target all identified bottleneck problems by optimizing the concurrent memory reads, aligning memory accesses and reorganization of computations.

The following optimization approaches will be tested:

- **O1** – using shared memory for the input signal,
- **O2a** – using shared memory for the filter kernel,
- **O2b** – using constant memory for the filter kernel,
- **O3** – using loop unrolling,
- **O4** – smaller calculation precision, and
- **O5** – element version.

3.1 Utilizing Shared Memory

The O1 optimization approach utilizes the shared memory for storing relevant segment from the actual ECG signal, although the complete input is expected to be stored on the Global Memory.

Let any thread block consist of *TPB* threads, and the filter length is *F*. Each block will eventually require an input segment of length *TPB + F* from the original signal. The first *TPB* threads (within a given thread block) initialize the relevant shared memory. Before each convolution operation, the block threads are synchronized by the intrinsic CUDA synchronization primitive *__syncthreads()* in order to ensure consistent data when initializing the shared memory.

Similarly, the O2a optimization approach utilizes the shared memory for storing corresponding filter kernel coefficients. In this case, O1 and O2a will both occupy the shared memory and therefore, the capacity limitation will decrease the level of performance gain.

3.2 Utilizing Constant Memory

The O2b optimization approach, which is complementary to the O2a approach, uses CUDA constant memory to store the read-only filter kernel (weight) coefficients. In this manner, consecutive reads of the same address do not generate any additional memory traffic. It is important to note that no bank conflict happens when using constant memory.

A combination of the O1 and O2b optimization approaches will use shared memory for the input signal segment and constant memory for impulse response coefficients. This avoids the bank conflicts that occur when utilizing only shared memory. Figure 4 presents the idea of this version of the parallel CUDA algorithm.

3.3 Loop unrolling

The O3 optimization approach is based on unrolling the loop in the thread. Eventually, this will eliminate instructions that increment the loop index and test if the limit is reached and combine two or more loop bodies into one loop body. The technique decreases the number of realized operations per thread.

Figure 6 presents a simplified code segment that loops 100 times. Figure 6a shows the basic method, and Figure 6b a code that uses twice unrolled loop. The procedure decreases the number of overall processed instructions, since instead of 100 checks for the looping condition and 100 instructions to increase x, the program is now performing those instructions only 50 times.

```
for (x=0; x<100; x++)
{
        process(x);
}
```

a) Normal loop

```
for (x=0; x<100; x+=2)
{
        process(x);
        process(x+1);
}
```

b) Unrolling by a factor of 2

Fig. 6.A loop unrolling example.

3.4 Precision decrease

The naive parallel CUDA code used double precision as a base type. On devices of computing capability 2.x, each bank has a bandwidth of 32 bits every two clock cycles, and successive 32- bit words are assigned to successive banks [8]. The warp size is 32 threads accessing 32 locations in a memory bank, each with a 32-bit precision. When double precision is used, in the case of accessing 64-bit numbers, the total number of memory locations is 16. Thus, a simultaneous access of 32 threads is not possible in a warp to 32 words of 64-bit length will cause, and only a half warp can be executed.

In order to increase the performance, the O3 approach decreases the precision to 32-bit single precision mode. In such case bank capacity limitation will be eliminated since 32 threads in a warp will simultaneously access 32 words of 32-bit length. Devices with computing capability 3.x allow a bandwidth of 64 bits every clock cycle [4]. Thus, bank capacity limitation arising from using double precision will be eliminated for devices with computing capability 3.x or higher. In addition, processing a single precision instruction is faster than the double precision.

3.5 Element version

Moreover, the element version of convolution is designed and implemented for ECG signal. In this version, instead of rows, a thread computes multiplication and summation of elements. Threads are defined for each combination of input and kernel items. Summation, on the other hand, is performed in logarithmic time. However, this approach leads to use of a lot of synchronization, which can cause a performance decrease.

4. Experimental Methodology

This section describes the conducted experiments.

4.1 Testing Environment

The sequential and parallelized codes are tested on an Amazon EC2 - G2 2xlarge instance. It consists of an 8-core Intel(R) Xeon(R) CPU E5- 2670 2.60GHz 64-bit system with a 15GB of memory. Moreover, the parallelized code is tested on a NVIDIA GRID GPU (Kepler GK104) device, having 1,536 CUDA cores with an 800Mhz system clock and 4GB RAM. The total amount of constant memory is 64KB where each block has 48KB of shared memory. This device has a computing capability 3.x, and input data is 64-bit double precision.

4.2 Experiments and test cases

The experiments were defined by testing the each previously described optimization approach independently and in a combination with the other approaches. Each experiment had several test runs for various sizes of the input data stream that consist of 10.000 up to 500.000 samples of an ECG signal with incremental steps of 10.000 samples.

Since the sampling frequency is 500Hz, the input data stream present an ECG signal from 2 seconds up to 1000 seconds. The filter kernels tested consisted of M=100, 250, 500, 750 and 1000 elements.

Each test run for the experiments was tested at least five times and an average value of measured times was calculated and used for further processing. A functional verification was conducted for each test run to verify if the functional characteristics of the naïve and optimization parallel algorithm executions obtain identical results.

4.3 Test Data

The measured parameters in the experiments are T_{pn} as time required by executing the naïve parallel algorithm using n cores, and T_{on} as time required by the given experiment with an appropriate combination of optimization approaches. The speed-up is calculated by Equation (2) as a ratio of the measured times for execution of the sequential and parallel algorithms.

$$S_n = \frac{T_{pn}}{T_{on}} \tag{2}$$

5. Performance analysis

This section describes the conducted experiments and presents the obtained results.

5.1. Shared Memory

Figure 7 shows the speedup of the O1 optimization using the shared memory for the input signal. The optimized version is more efficient, and the speedup increases with the filter kernel size. An average value of 13% increase is obtained as speedup for the filter length of 100, and 78% for filter length of 1000. We observe that for each filter size, the speedup $S_n > 1$ and the greater speedup is achieved when increasing the filter length, which proves the scalability of the improvement.

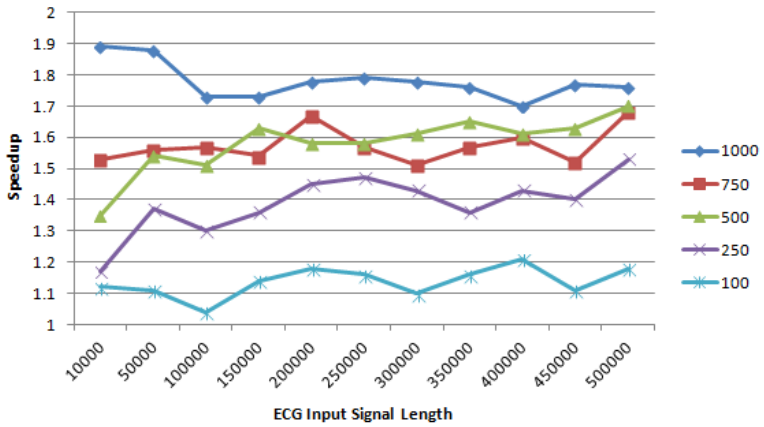

Fig. 7. Speedup of the O1 optimization using shared memory for the input signal in the case of different kernel sizes.

Additionally, Figure 8 shows the speedup of O2a optimization only using the shared memory for filter kernel. It can be noted code is optimized by roughly 10% and that speedup increases with the increasing filter kernel length. The higher the filter kernel is, the better performance gain is obtained. Better results are obtained for the usage of shared memory for storing the input signals, instead of the filter coefficients.

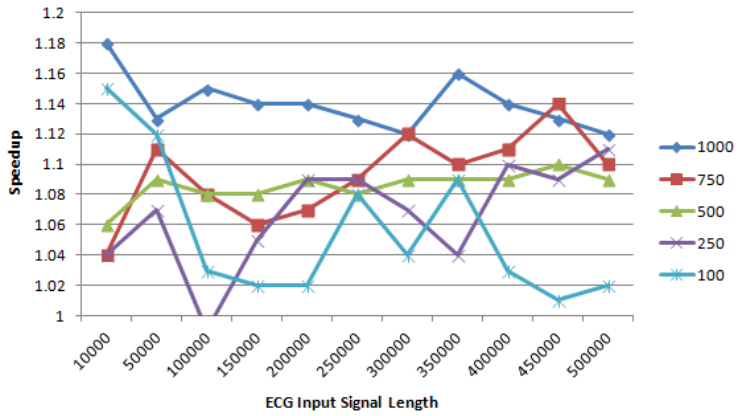

Fig. 8. Speedup of the O2a optimization using shared memory for different kernel sizes

5.2. Constant Memory

Figure 9 presents the analysis made on the constant memory defined by the O2b optimization approach. One can conclude that using a constant memory for the kernel will increase the performance by average 15%. We have compared the O2a and O2b approaches, as presented in Figure 10. The solution using constant memory instead of shared memory for storing the filter kernels obtains a higher performance by an average speedup of 4%.

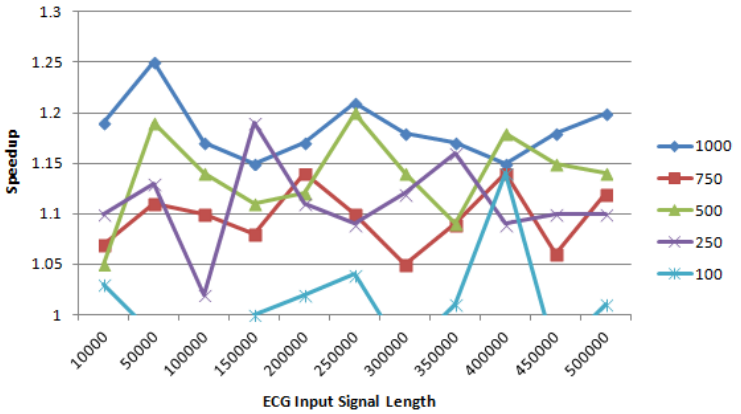

Fig. 9. Speedup of the O2b optimization approach using Constant Memory for the filter kernel.

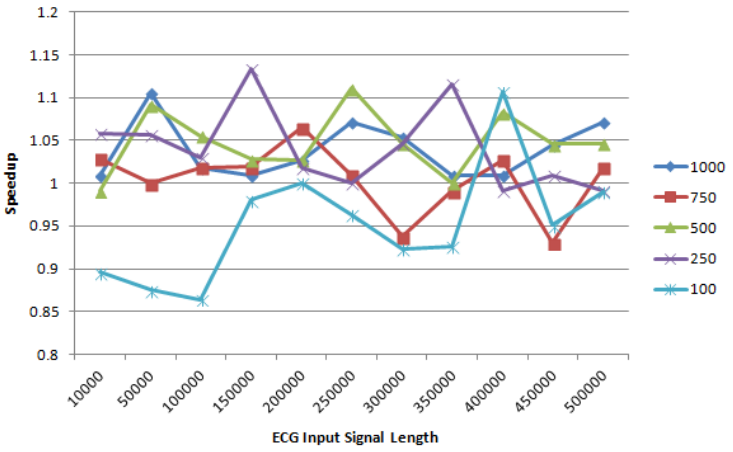

Fig. 10. Speedup of the O2b optimization approach compared to the O2a.

5.3. Loop Unrolling

The speedup obtained by loop unrolling O3 optimization approach is presented in Figure 11. On average, the loop unrolling optimization technique increases the performance for 1 to 5 percent. The best performance gain was obtained for the unrolling length of 64.

5.4. Decreasing the double to single precision

The conducted experiments generally use a 64-bit double precision for storing and calculating the convolution. However, a GPU double precision calculation is a costly operation [7]. We have tested the performance by decreasing the precision to 32-bit, since the functional testing showed difference in both approaches expressed in order of 10^{-5}. Figure 12 presents the effect of the decreased precision on the performance. As input size increases, the single precision version is approximately 40 to 60 percent faster than the pure parallel code.

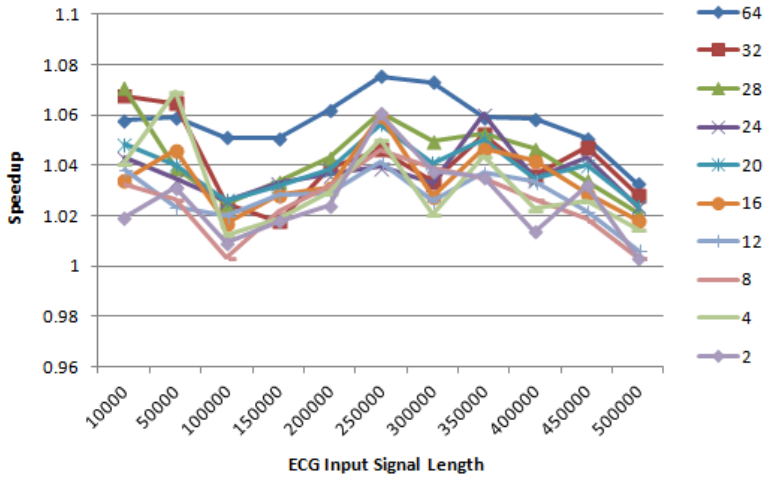

Fig. 11. Speedup of O3 optimization approach using loop unrolling for filter length of 1000.

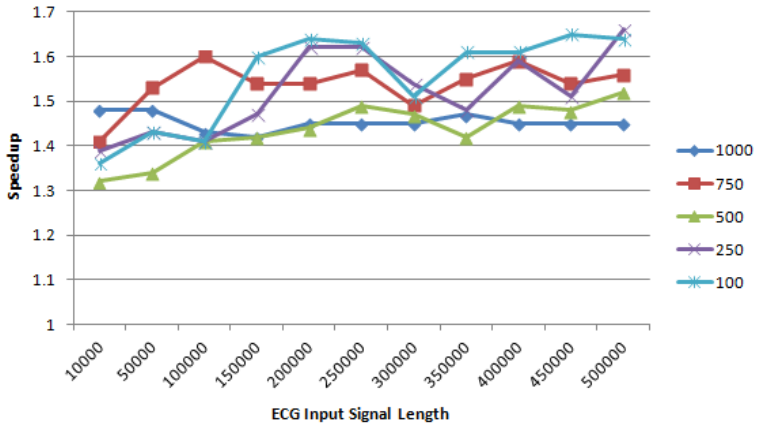

Fig. 12. Speedup obtained by the O4 optimization approach by decreasing the precision

5.5. Element version

The conducted experiments show that the element version actually decreases the performance. This happens due to the synchronization and bank conflicts that need to be performed in each step.

6. Discussion

Figure 13 combines the optimizations compared to the double precision version of the previous code. We observe that Shared Input and Output version speedups the code by 13% and 78%. Using Constant memory version is 15% faster compared to the previous code [2]. Combining these, the maximum speedup of 2.4 is obtained. Thus we can conclude that the hypothesis we set is confirmed.

We were also interested to investigate the effect of the loop unrolling. We observe that using a loop unrolling with a length of 64 introduces additional 4 % performance gain. Additionally, when the precision is decreased to 32 bit, the speedup increases up to 6. An interesting case was to see whether the element version performs better. We observed that, due to the internal synchronization and bank conflicts performed on each step, the performance of the element version is not attractive.

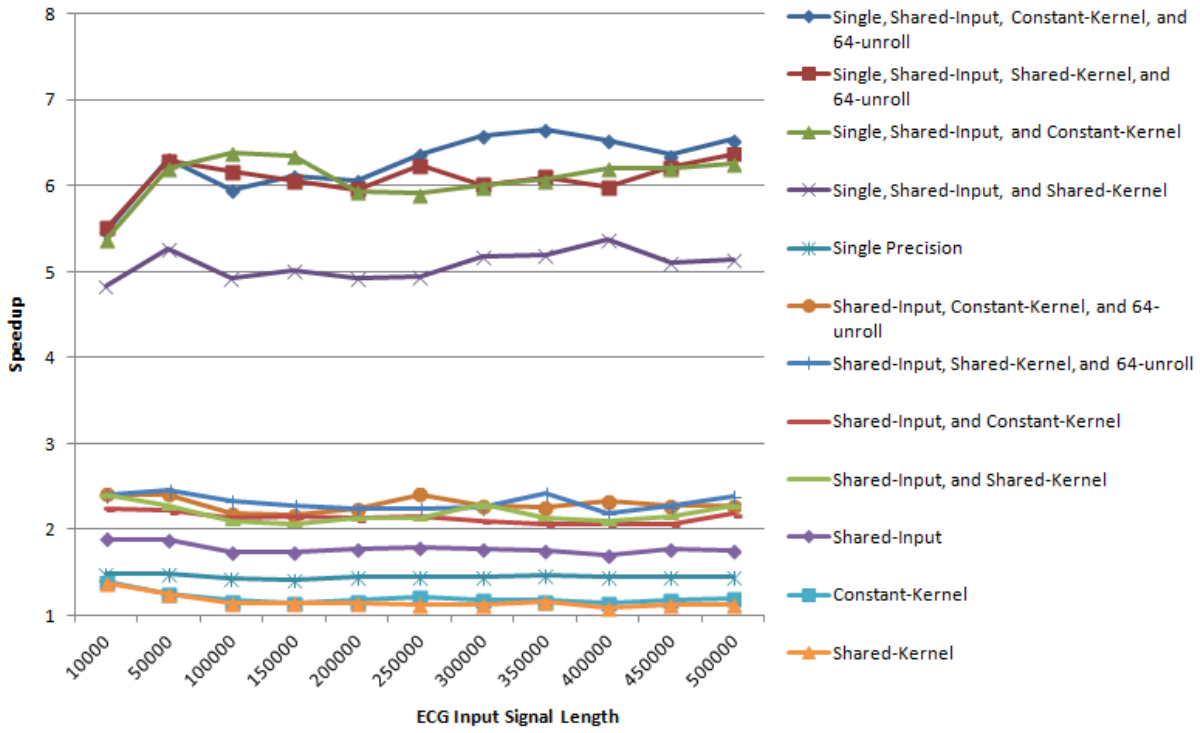

Fig. 13. Performance gain by a combination of optimization approaches.

Anwar et al. [9] have published a solution using shared memory, without analyzing the possibilities of the constant memory. Wefers et al. [12] have concentrated on implementing a real-time convolution on frequency domain. Herdeg et al. [13] have also concentrated on the frequency and time domain fast convolution. On time domain convolution, though, authors have not provided any information related to memory storage, access, and optimizations.

Previously we have considered parallelizing DSP filters on a Maxeler dataflow engine [1]. We obtained promising results, with linear speedups proportional to the kernel length. We calculated speedup by measuring a number of sequential steps needed to convolve the signal, whereas here we compared execution times.

Additionally, we have focused on parallelizing DSP filtering algorithm on CUDA [2], by utilizing thousands of GPU cores. The experiments have shown linear speedups, proportional to the kernel size. Obtained results serve as a basis for this research. We have also observed that threads per block (TPB) should be kept as high as possible by considering the upper limit being provided by the specification of the GPU card.

This research was concentrated on time-domain convolution. Generally, data intensive streams are convolved on the frequency domain, by actually multiplying two signals on the frequency domain. Gained results were much faster, but the results were not accurate. This is because ECG signal is not always periodic; since the heart beat rate changes due to physical activity, emotional state, etc. So, in the case of ECG, frequency-domain convolution is not recommended.

We also investigated the usability of NVIDIA's CUBLAS library to dispatch each of the required convolution calculations as a Level 1 BLAS (Basic Linear Algebra. Subprograms) caxpy (Scalar Alpha X Plus Y) operation. Caxpy computes a constant alpha times a vector x plus a vector y, and finally, overwrites the initial values of vector y. In the case of ECG signal, the impulse response has variable filter elements, making the use of this operation useless.

Moreover, we designed and implemented the element version of convolution for ECG signal context using explicit synchronization for reduction and summation. Even though the reduction operation has logarithmic complexity, our tests have shown that this approach decreases the performance. We plan to develop an optimized algorithm where multiplications will be distributed across different blocks, and the synchronization will be handled inside threads of a block. We believe that this will optimize the current version of the code.

Higher speedups are obtained when using single instead of double precision. This is directly related to the achieved GFLOPS (billions of floating point operations per second) of the device. On GPUs, double precision GFLOPS is smaller than the single precision [11], which results in faster executions when floating point numbers are used.

In this research, we have also considered using loop unrolling. Approximately 1-5 % performance gain is obtained with double precision. We have also tested to decrease the precision, and nearly doubled the performance of the algorithm.

7. Conclusion

This research contributes to the CUDA GPU optimization strategies for the noise elimination on ECG heart signals. The proposed optimizations, take advantage of intra-block shared memory and the general constant memory. By storing ECG segments on shared memories, we have optimized the memory accesses.

Additionally, we used storing of the filter coefficients on the constant memory, thus consecutive reads of the same address does not generate any additional memory traffic. We are aware of the limitations, where filter length should be smaller than the maximum number of threads per block due to the internal synchronization inside a block.

Results obtained by executing optimized algorithms show they are identical for each of the filter types. From the obtained results, we can conclude that proper usage of shared and constant memory has a positive impact on the performance. Our analysis showed that their combined effect yield 2.4 times faster executions compared to the previous code. We can, therefore, conclude that the hypothesis is confirmed.

Considering loop unrolling speeds up the code by 1-5%. Moreover, we tested the decreased precision effect on the performance and got nearly 1.5 faster code when on 1000 filter length. We observed that the element version is not effective when ported on GPU.

It is important to note that each of the proposed optimization techniques adds up to the combined speedup. We observed that the best-combined effect had a speedup of 6.

As future work, we plan to carry out more tests on multiple GPUs tied together. We will analyze shared memory parallelism by using OpenMP or similar solutions. We also consider developing an efficient algorithm for filter sizes greater than the available maximum number of threads per block. Additionally, we plan to investigate the element algorithm version on CUDA and Intel IPP. In this algorithm, threads are defined on the element level. We expect that the parallelized code will scale linearly with increasing filter size.

8. References

[1] E. Domazet, M. Gushev, and S. Ristov, “Dataflow DSP filter for ECG signals,” in 13th International Conference on Informatics and Information Technologies, Bitola, Macedonia, 2016, in press

[2] E. Domazet, M. Gushev, and S. Ristov, “CUDA DSP filter for ECG signals,” in 6th International Conference on Applied Internet and Information Technologies, Bitola, Macedonia, 2016, in press

[3] T. S. Lugovaya, “Biometric human identification based on ECG,” http://www.physionet.org/pn3/ecgiddb/biometric.shtml, 2005, last visited on 28.05.2016

[4] “Cuda C best practices guide,” http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, licensed under the Creative Commons Attribution 3.0 Unsupported license, last visited on 28.05.2016.

[5] S. W. Smith, Digital signal processing: a practical guide for engineers and scientists. Newnes, 2003

[6] "Schematic diagram of normal sinus rhythm for a human heart as seen on ECG", Wikipedia, https://commons.wikimedia.org/wiki/File:SinusRhythmLabels.svg, released work into the public domain, last visited on 28.05.2016.

[7] Whitehead, Nathan, and Alex Fit-Florea. "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs." rn (A+ B) 21 (2011): 1-1874919424.

[8] “Tuning CUDA Applications for Maxwell,” http://docs.nvidia.com/cuda/maxwell-tuning-guide/, last visited on 28.05.2016.

[9] Anwar, Sajid, and Wonyong Sung. "Digital signal processing filtering with GPU." (2009).

[10] Sörnmo, Leif, and Pablo Laguna. "Electrocardiogram (ECG) signal processing." Wiley Encyclopedia of Biomedical Engineering (2006).

[11] Geeks3D, "AMD Radeon and NVIDIA GeForce FP32/FP64 GFLOPS Table", http://www.geeks3d.com/20140305/amd-radeon-and-nvidia-geforce-fp32-fp64-gflops-table-computing/, last visited on 28.05.2016

[12] Wefers, Frank, and Jan Berg. "High-performance real-time FIR-filtering using fast convolution on graphics hardware." Proc. of the 13th Conference on Digital Audio Effects, 2010

[13] McGraw-Herdeg, Michael P., Douglas P. Enright, and B. Scott Michel. "Benchmarking the NVIDIA 8800GTX with the CUDA Development Platform." HPEC 2007 Proceedings (2007)