Real-time Hermite Polynomial Characterization of Heartbeats using a Field-Programmable Gate Array

Kartik Lakhotia*, Gabriel Caffarena†, Alberto Gil†, David G. Márquez‡, Abraham Otero† and Madhav P. Desai *
*Indian Institute of Technology (Bombay)
Powai, Mumbai 400076, India
Email: madhav@ee.iitb.ac.in
†Laboratory of Bioengineering
University CEU-San Pablo, Boadilla del Monte 28668, Spain
Email: gabriel.caffarena@ceu.es
‡Centro Singular de Investigacion en Tecnologias da Informacion (CITIUS)
University of Santiago de Compostela, 15782 Santiago de Compostela, Spain
Email: david.gonzalez.marquez@usc.es

***Abstract*—The characterization of ECG heartbeats is a computationally intensive problem, and both off-line and on-line (real-time) solutions to this problem are of great interest. In this paper, we consider the use of a dedicated hardware implementation (using a field-programmable gate-array (FPGA)) to solve a critical component of this problem, namely, the best-fit Hermite approximation of a heartbeat. The implementation is generated using an algorithm-to-hardware compiler tool-chain and the resulting hardware is characterized using an off-the-shelf FPGA card. The single beat best-fit computation latency when using six Hermite basis polynomials is under $0.5ms$ with a power dissipation of 3 watts, demonstrating true real-time applicability.**

The characterization of ECG heartbeats is a computationally intensive problem, and both off-line and on-line (real-time) solutions to this problem are of great interest. In this paper, we consider the use of a dedicated hardware implementation (using a field-programmable gate-array (FPGA)) to solve a critical component of this problem. We describe an implementation of real-time best-fit Hermite approximation of a heartbeat using six Hermite polynomials. The implementation is generated using an algorithm-to-hardware compiler tool-chain and the resulting hardware is characterized using an off-the-shelf FPGA card. The single beat best-fit computation latency is under $0.5ms$ with a power dissipation of 3 watts.

Automatic ECG analysis and characterization can be of great help in patient monitoring. In particular, the characterization of the QRS complex by means of Hermite functions seems to be a reliable mechanism for automatic classification of heartbeats [1]. The main advantages seem to be the low sensitivity to noise and artifacts, and the compactness of the representation (e.g. a 144-sample QRS can be characterized with 7 parameters [2]). These advantages have made the Hermite representation a very common tool for characterizing the morphology of the beats [1], [2], [3], [4], [5].

ECG analysis using Hermite functions has a substantial amount of parallelism. Solutions to the problem have been investigated using processors (and multi-cores) and graphics processing units (GPU's). In this paper, we consider the alternative route of using an FPGA to implement the computations. In particular, our work is motivated by the potential of an FPGA (or eventually, a dedicated application-specific circuit) for low-latency energy efficient heart-beat analysis.

In generating the hardware for heart-beat analysis, we make extensive use of algorithm-to-hardware techniques. By this we mean that the hardware is generated from an algorithmic specification that is written in a high-level programming language (**C** in this case), which is then transformed to a circuit implementation using the AHIRV2 algorithm to hardware compilation tools [6], [7], [8]. The resulting hardware is then mapped to an FPGA card (the ML605 card from Xilinx, which uses a Virtex-6 FPGA). The circuit is then exercised through the PCI-express interface and used to classify beats. The round-trip latency of a single beat classification was found to be under $0.5ms$.

I. QRS approximation by means of Hermite polynomials

The aim of using the Hermite approximation to estimate heartbeats is to reduce the number of dimensions required to carry out the ECG classification, without sacrificing accuracy. The benchmarks used in this work come from the MIT-BIH arrhythmia database [9] which is made up of 48 ECG recordings whose beats have been manually annotated by two cardiologists. Each file from the database contains 2 ECG channels, sampled at a frequency of 360 Hz and with a duration of approximately 2000 beats.

Before doing the Hermite approximation, the ECG signal is processed to remove the base-line drift. The QRS complexes for each heartbeat are extracted by finding the peak of the beat (e.g. the R wave) and selecting a window of 200 ms centered on the peak. The beat-window is further extended to 400 ms by padding 100-ms sequences of zeros at each side of the complex. Thus, the QRS beat data used as an input to the Hermite polynomial approximation consists of individual beats described as a 144-sample vector $\vec{x} = \{x(t)\}$ of double precision floating point numbers. This vector is to be estimated with a linear combination of N Hermite basis functions (for the work reported in this paper, we use $N = 6$).

The goal then is to find the best minimum-mean-square-

```
void HermiteBestFit()
{
    receiveHermiteBasisFunctions();

    while(1)
    {
        receiveHeartBeat();
        innerProducts();
        findBestFit();
        reportResults();
    }
}
```

Fig. 1. High-level view of algorithm mapped to the FPGA

error (MMSE) approximation to $\{x(t)\}$ as

$$\hat{x}(t) = \sum_{n=0}^{N-1} c_n(\sigma)\phi_n(t,\sigma), \tag{1}$$

with

$$\phi_n(t,\sigma) = \frac{1}{\sqrt{\sigma 2^n n! \sqrt{\pi}}} e^{-t^2/2\sigma^2} H_n(t/\sigma) \tag{2}$$

where $H_n(t/\sigma)$ is the n^{th} Hermite polynomial. These polynomials can be computed recursively as

$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x), \tag{3}$$

where $H_0(x) = 1$ and $H_1(x) = 2x$. The parameter σ is a time-scaling factor in the polynomials which needs to be chosen carefully. In [1] the maximum value of σ for a given order n is estimated. As the value of n increases, the value of σ_{MAX} decreases.

The Hermite polynomials are orthonormal. Thus, the optimal coefficients that minimize the estimation error for a given σ are

$$c_n(\sigma) = \sum_t x(t) \cdot \phi_n(t,\sigma) \tag{4}$$

The best fit is calculated by comparing the MMSE approximation for each σ, and keeping the one with the smallest value. Once the best σ and the corresponding fit coefficients $\vec{c} = \{c_n(\sigma)\}$ $(n \in [0, N-1])$ are found for each heartbeat, it is possible to use only these figures to perform morphological classification of the heartbeats [1].

II. Beginning the FPGA implementation: the algorithm

The algorithm used in the FPGA implementation is illustrated in Figure 1.

The implementation first receives the values of the Hermite polynomial basis functions, and stores them in distinct arrays in the hardware. In the current implementation, we use six arrays to store the basis functions for order $n = 0$ to $n = 5$. For each n, basis functions for ten different values of σ are stored in the corresponding array. The values of σ used range from $1/120$ to $1/90$.

```
void  innerProduct()
{
  int I;
  for (I=0; I < NSAMPLES; I++)
  { // outer-loop
     double x = inputData[I];
     for(SI = 0; SI < NSIGMAS; SI++)
     { // inner-loop
        int I0 = I + Offset[SI];
        double p0 = (x*hF0[I0]);
        double p1 = (x*hF1[I0]);
        double p2 = (x*hF2[I0]);
        double p3 = (x*hF3[I0]);
        double p4 = (x*hF4[I0]);
        double p5 = (x*hF5[I0]);
        dotP0[SI] += p0;
        dotP1[SI] += p1;
        dotP2[SI] += p2;
        dotP3[SI] += p3;
        dotP4[SI] += p4;
        dotP5[SI] += p5;
     }
}
```

Fig. 2. Inner-product loop

After this initialization step, the hardware executes a continuous loop. In the loop body, the hardware first listens for heart beats. When a complete heart-beat (144 samples) is received, the inner products of the heart-beat with all the basis functions are calculated in a double loop. After all inner products are calculated, the inner product coefficients are used to compute the best fit among the different values of σ. The best-fit σ index and coefficients are then written out of the hardware.

The algorithm as described above is purely sequential and does not contain any explicit parallelization. The AHIRV2 compiler (described later in Section III) is intelligent enough to extract parallelism from the two critical loops (in the inner-product and best-fit functions). Even with this simple coding of the hardware algorithm, we observe that excellent real-time performance is observed (in comparison with CPU/GPU implementations). Going further, it is possible to specify explicit parallelism by and exploit it by using multiple function units in hardware in order to reduce the processing latency. These investigations are currently in progress.

A. The inner product loop

The inner product loop is shown in Figure 2. The outer loop is over the samples, and the inner loop across the σ values. There is a high-level of parallelism in the inner loop which can be further boosted by unrolling the outer loop. When translating this to hardware, the entire function uses one single double-precision multiplier and one single double-precision adder. Further note that the arrays hFn and $dotPn$ are declared on a per-n basis (for $n = 0$ to $n = 5$). This allows the arrays to be mapped to distinct memory spaces, thus increasing the memory access bandwidth in the hardware.

```
void computeMSE()
{
  int I, SI;
  best_mse = 1.0e+20;
  best_sigma_index = -1;
  for (I=0; I<NSAMPLES; I=I+4)
  { // outer-loop
     for (SI=0; SI<NSIGMAS; SI++)
     { // inner-loop
        int fetchIndex0 = I + Offset[SI];
        double p0 = (dotP0[SI]*hF0[fetchIndex0]);
        double p1 = (dotP1[SI]*hF1[fetchIndex0]);
        double p2 = (dotP2[SI]*hF2[fetchIndex0]);
        double p3 = (dotP3[SI]*hF3[fetchIndex0]);
        double p4 = (dotP4[SI]*hF4[fetchIndex0]);
        double p5 = (dotP5[SI]*hF5[fetchIndex0]);
        double diff = (inputData[I]-
                       ((p0+p1) + (p2+p3) + (p4+p5)));
        err[SI] += (diff*diff);
      }
    }
    for (SI=0; SI<NSIGMAS; SI++)
    {
        if(err[SI] <  best_mse)
        {
           best_mse = err[SI];
           best_sigma_index = SI;
        }
    }
}
```

Fig. 3. MMSE calculation loop

B. The minimum-mean-square loop

The MMSE calculation hardware uses the algorithm shown in Figure 3. Note that the inner loop again has considerable parallelism. One pipelined double precision multiplier and one pipelined double precision adder are used to implement the loop. The arrays referred to in the loop $dotPn$ and hFn are all implemented in disjoint memories to give high memory access bandwidth.

C. Optimizations: loop unrolling and loop pipelining

The current implementation uses a simple sequential specification. We have investigated the impact of further optimizations. In particular, we find that outer-loop unrolling (up to four) and inner-loop pipelining have substantial impact on the performance of the generated hardware.

If we look at the inner loop in Figure 2, we observe that because of the accumulation of products in each loop iteration, the time interval between successive loop initiations is equal to the latency of the adder (because the next loop iteration needs to wait for the completion of the sum in the current iteration). This latency is around 20 cycles in our case, and becomes a performance limiter.

If the outer loop in the innner-product function (Section II-A) is unrolled four times, the resulting code will have the form shown in Figure 4. The amount of parallelism in the inner loop increases by a considerable margin. The loop iteration initiation latency stays about the same, but the amount of computation done in each loop iteration is quadrupled, thus leading to higher performance. In Section V, we provide actual measurements in hardware which support this observation.

```
void  innerProduct()
{
  for (I=0; I < NSAMPLES; I += 4)
  { // outer-loop
     double x0 = inputData[I];
     double x1 = inputData[I+1];
     double x2 = inputData[I+2];
     double x3 = inputData[I+3];
     for(SI = 0; SI < NSIGMAS; SI++)
     {
        int I0 = I + Offset[SI];
        I1 = I0+1; I2 = I0+2; I3 = I0+3;
        // compute inner product of
        // (x0,x1,x2,x3) with
        // (hFn[I0], hFn[I1], hFn[I2], hFn[I3])
        // for n=0,1,2,..5.
        //
        // accumulate into dotPn[SI].
        //
     }
}
```

Fig. 4. Inner-product loop unrolled four times.

A similar improvement is observed by unrolling the outer loop in the MMSE computation shown in Figure 3.

III. From algorith-to-hardware using AHIRV2, a C-2-VHDL compiler

The AHIRV2 compiler tool-chain [6], [7], [8] provides a pathway from a C-program to actual synthesizable hardware. The tool-chain takes a description of an algorithm (described in C) and produces a VHDL logic circuit description which is equivalent to the algorithm.

The AHIRV2 compiler starts with a C program and produces VHDL. For the The clang-2.8 compiler[1] is used as the C front-end and is used to emit LLVM byte-code[2], which is then transformed to VHDL using the following transformations:

1) The LLVM byte-code is translated to an internal intermediate format, which is itself a static-single assignment centric control-flow language (named **Aa**) which allows the description of parallelism using fork-join structures as well as arbitrary branching.
2) The **Aa** description is translated to a virtual circuit (the model is described in the next section). During this translation, the following major optimizations are performed: declared storage objects are partitioned into disjoint memory spaces using pointer reference analysis, and dependency analysis is used to generate appropriate sequencing of operations in order to maximize the parallelism. Inner loops in the **Aa** code are pipelined so that multiple iterations of a loop can be executed concurrently.
3) The virtual circuit is then translated to VHDL. At this point, decisions about operator sharing are taken. Concurrency analysis is used to determine if a shared hardware unit needs arbitration. Optimizations related to clock-frequency maximization are also carried out

[1] www.clang.org
[2] www.llvm.org

here. The generated VHDL uses a pre-designed library of useful operators ranging from multiplexors, arbiters to pipelined floating point arithmetic units (arbitrary precision arithmetic is supported, and in particular, there is full support for IEEE-754 single precision and double precision add/multiply with all rounding modes).

A. An illustration of the virtual circuit generated by the AHIRV2 compiler

The virtual circuit generated by the AHIRV2 compiler consists of three cooperating components: the control-path, the data-path and the storage system [7], [8].

To illustrate the model, we consider a simple example.

```
float a[1024], b[1024];
float dotp = 0.0;
for(i=0; i < 1024; i++)
{
   dotp += a[i]*b[i];
}
```

The AHIRV2 tool-chain transforms this program to produce a virtual circuit which is depicted in Figure 5. The virtual circuit is then translated to synthesizable VHDL. The virtual circuit consists of three components independent parts, namely the data-path, the storage-subsystem and the control-path.

1) The data-path: The data-path is a directed hyper-graph with nodes being operations and arcs being nets (shown as ovals). Each net has at most one operation which drives it. Further, most operations have a split protocol handshake with the control-path: two pairs of request/acknowledge associations (*sr/*sa for sampling the inputs and *cr/*ca for updating the outputs). The operation samples its inputs on receiving the sr request symbol and acknowledges the completion of this action by emitting the sa acknowledge symbol. After receiving the cr symbol, the operation will update its output net using the newly computed value. The sequencing is required to be

```
sr -> sa -> cr -> ca
```

Note that an operation can be re-triggered while an earlier edition of the operation is in progress (this is important if the operation is implemented using a pipelined operator).

Some data-path operations (such as the multiplexor shown on the top and the decision operation shown at the bottom left in Figure 5) follow a simpler protocol. The multiplexor has a pair of requests and a single acknowledge, with the condition that at most one of the requests is received at any time instant. The input corresponding to the request is then sampled and stored in the output net of the multiplexor. The decision operation has a single request and two acknowledes. Upon receipt of the request symbol, the decision operation checks its input net and emits one of the two acknowledges depending on whether the input is zero/non-zero.

In Figure 5, the following data-path operations are instantiated:

```
mI, mdotP  multiplexors for I, dotP.
INCR       increment for I++
LA         load for a[I]
LB         load for b[I]
FMUL       multiply for p=a[I]*b[I]
FADD       add for dotP += a*b
CMP EQ     compare for COND=(I==1023)
D          decision  COND?
```

Remark Note that the data-path only shows the operations and their interconnection. When the data-path is implemented as hardware, multiple operations may be mapped to a single operator depending on cost/performance tradeoffs. When this is done, multiplexing logic is introduced in the hardware. These decisions and manipulations are performed in the compiler stage which is responsible for transforming the virtual circuit to VHDL.

2) Storage subsystem: The load and store operations in the data-path are associated with memory subsystems. In general, there can be multiple disjoint memory subsystems inferred by the compiler. In this particular case, the arrays a[] and b[] are mapped to disjoint memories, due to which the two loads are allowed to proceed in parallel (the relaxed consistency model is enforced). In order to maintain the relaxed consistency model, the memory subsystems are designed to use a time-stamping scheme which guarantees first-come-first-served access to the same memory location.

3) Control-path: The control-path in the virtual circuit encodes all the sequencing that is necessary for correct operation of the assembly. The control-path (shown on the left in Figure 5) is modeled as a Petri-net with a unique entry point and a unique exit point. The Petri-net is constructed using a set of production rules which guarantee liveness and safeness [7]. Transitions in the Petri-net are associated with output symbols to the data-path (these can be described by the regular expressions *sr and *cr) and input symbols from the data-path (these are of the form *sa and *ca). The *sr symbols instruct an element in the data-path to sample its inputs and the *cr symbols instruct an element in the data-path to update its outputs (all outputs of data-path elements are registered). The *sa and *ca symbols are acknowledgements from the data-path which indicate that the corresponding requests have been served.

The following classes of dependencies are encoded in the control Petri-net:

- Read-after-write (RAW): If the result of operator A is used as an input to operator B, the sr symbol to B can be emitted only after the ca symbol from A has been received.
- Write-after-read (WAR): If B writes to a net whose value needs to have been used by A earlier, for example as in

  ```
  a = (b+c) -- operation A reads c
  c = (p*q) -- operation B writes to c
  ```

  where there is a WAR dependency through c, then the cr request to B can be issued only after the sa acknowledge from A has been received.
- Load-Store ordering: If P,Q are load/store operations to the same memory subsystem, and if at least one

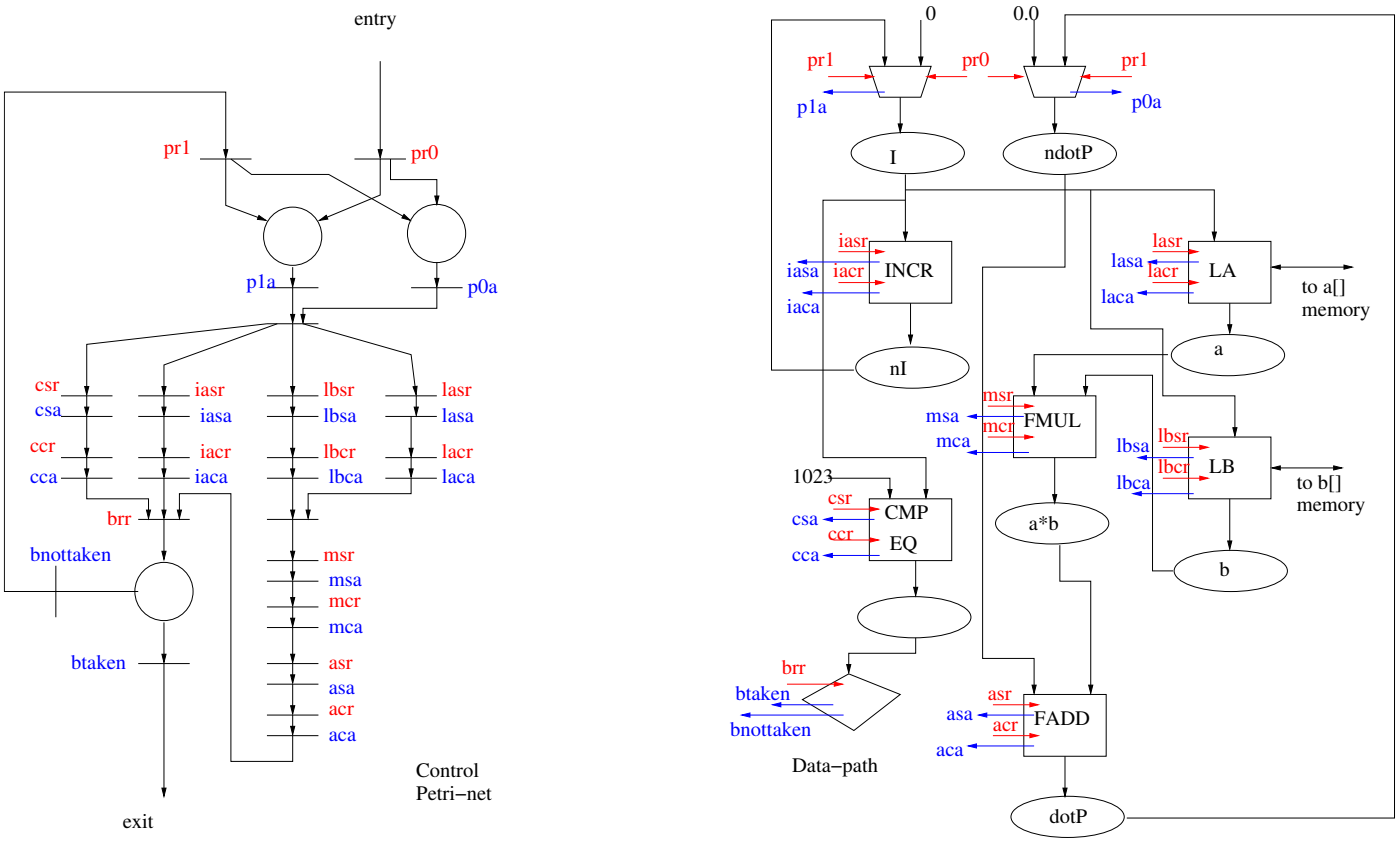

Fig. 5. Control-data-storage virtual circuit model.

of P,Q is a store, and if P is supposed to happen before Q, then the sr request to Q must be emitted only after the sa acknowledge from Q has been received. The memory subsystem itself guarantees that requests finish in the same order that they were initiated. This takes care of WAR, RAW and WAW memory dependencies.

The control-path in Figure 5 shows the sequencing generated by these rules. When pipelining an inner loop, the execution of an operation in a particular iteration is enabled as soon as its dependencies on results from previous iterations are satisfied.

IV. Hardware Implementation Details

The overall system has 3 major components:

- A host computer, which is used to calculate the Hermite basis functions, initialize the FPGA card, send beat data to the FPGA card and receive the best-fit coefficients from the FPGA card.
- The FPGA card, on which the best-fit algorithm is implemented. We use the Xilinx ML605 card which features a Virtex-6 FPGA and an 8-lane PCI express interface.
- The FPGA card driver, which is based on the RIFFA infrastructure [10].

The algorithm mapped to the FPGA is first described in a C program (code fragments descibed in Sections II, II-A and II-B). The architecture of the hardware produced is shown in Figure 6. In the initialization phase, the hardware-side listens on an input FIFO to acquire the Hermite polynomials (these are stored in six disjoint memories, one for each order $n = 0, 1, 2, \ldots 5$). After this step, the unit that receives the beat samples is triggered. This unit listens on the input FIFO and receives a 144 sample heartbeat (coded as 144 double-precision floating point numbers). After receiving the sample, it triggers the inner-product stage. In the inner-product stage, the hardware computes, for each σ and n, an inner product of the received beat with the Hermite polynomials $\phi_n(t, \sigma)$. We are using ten values of σ and 6 values of n. Thus, 60 inner-products are computed in this phase. The inner products are stored in ten disjoint memories, one for each σ. After this is done, the MMSE stage is triggered. In the MMSE stage, the inner-products are used to find the best fit σ. The computed best-fit coefficients are sent back to the host using the output FIFO. The hardware unit which listens for the next beat is then triggered (wait for the next beat).

The VHDL hardware for this design is generated using the AHIRV2 toolchain which was described in Section III. The generated VHDL is instantiated in the FPGA together with the RIFFA wrappers, and the resulting design is synthesized and mapped to the Virtex-6 FPGA using the Xilinx ISE 14.3 toolset.

V. Results

We measure the round-trip delay and FPGA core power consumption for processing one beat. The round-trip delay

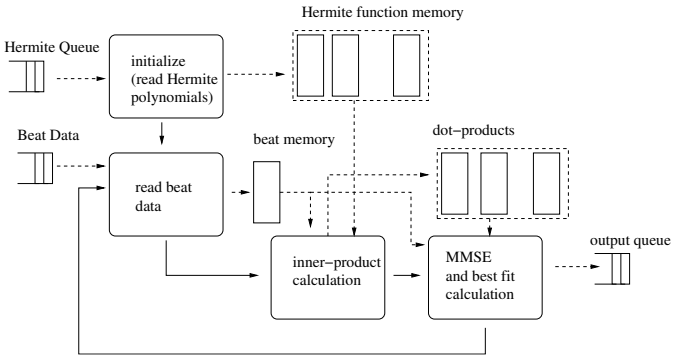

Fig. 6. Hardware Architecture

is the time interval between the beginning of transmission of beat-data from the host to the hardware and the beginning of reception of best fit coefficients from the hardware. The test feeds a single beat at a time to the FPGA and measures the latency.

In the implementation, the two outer-loops described in Sections II-A and II-B were unrolled to different extents to see the impact of unrolling on the system performance. Three levels of unrolling were tried: one-way, two-way and four-way. The four-way unrolling gave the best performance, as expected. The results are summarized in Table I (the reported latency is the average value observed across 100 beats). The minimum latency achieved with four-way-unrolling was observed to be 0.39 ms (for the processing of a single 144-sample beat). The power dissipation values are measured using hardware monitoring while the beats are being processed, and represent peak power dissipation.

TABLE I. RESULTS: FPGA UTILIZATION AND LATENCY FOR DIFFERENT LOOP-UNROLLING LEVELS

Unroll-level	Slice LUT Utilization	Slice Register Utilization	Avg. Processing Latency	FPGA core Power Consumption
1-way	56839	65995	1.39ms	2.75W
2-way	65895	80709	0.80ms	2.88W
4-way	84331	110165	0.44ms	3.09W

It is clear that FPGA can be used for *true* real-time processing since the computation time required to process a pair of beats (corresponding to a heart-beat sampled on two channels) is less than $1ms$, which is much smaller than time-interval between actual heart beats (which is about $1s$). The observed power dissipation is 3.1W. Hardware utilization in 4-way unrolled system is less than 55% of the FPGA resource.

A. Comparison with GPU/CPU implementations

Hermite basis fitting has been evaluated on GPU and CPU implementations as well. For example, in [11], the authors demonstrate a GPU implementation that shows excellent scaling behaviour in real-time, so that 100K beats can be processed in 15.7 seconds. The average processing latency per pair of beats if 0.022 msecs. However, it must be stressed that in general the GPU achieves full parallelization when a huge set of data is processed and real-time requires the processing of small sets of data. Real-time is achieved on the GPU by means of processing chunks of 10 and 100 beats, resulting in latencies of around 10 seconds and around 100 seconds respectively. Thus, *true* real-time processing is not fully acomplished since latencies are too long. On the other hand, a CPU is able to do low-latency real-time wih a latency per beat of around 30 msecs per pair of beats. As for our FPGA implementation, the latency needed to process a pair of beats is under $1ms$. Since the beats are processed one at a time, actual real-time is achieved.

These comparisons are made against the same algorithm executed on Intel-i7 PC(1.6GHz) and NVIDIA TESLA C2050 (1.15GHz) [11]. The operating frequency on the FPGA card was 100MHz which is less than one-tenth of that used on CPUs and GPUs. Further, the peak power consumption (while actually processing the beats) on the FPGA is $3W$ as compared to $100W+$ on Core i7 processors and $200W+$ on the GPU. In summary, the FPGA achieves a better average latency than the CPU and it ourperforms the GPU in term of the low-latency requirements for *actual* real-time capabilities, displaying a power comsumption which is two orders of magnitude lower than the other two technologies. Thus, the FPGA is an attractive option for low energy real-time ECG classification applications in portable health monitoring devices.

VI. CONCLUSIONS

In this paper, a solution to the problem of Hermite polynomial based heart-beat characterization using FPGAs is presented. We have mapped the problem to hardware using algorithm-to-hardware techniques (with the AHIRV2 tools). The Xilinx ML605 card with a Virtex-6 FPGA was used as the platform and the RIFFA host-interface was used to communicate with the FPGA card.

This methodology is presented as an alternative to existing software oriented approaches, for real-time latency-sensitive signal processing. When using the FPGA, a substantial latency reduction in single-beat processing was observed (in comparison with both GPU and CPU implementations of the same algorithm). Further, the peak power dissipation in the FPGA is almost two orders of magnitude lower than that observed in the CPU/GPU case.

The highly parallel GPU and CPU architectures are very effective in off-line processing (processing of a large number of beats, not necessarily in real-time). When it comes to real-time, online beat processing, our work demonstrates that dedicated hardware implementation using an FPGA offers a very competitive platform for ECG signal processing.

ACKNOWLEDGMENT

The authors would like to thank Rakesh Mehta for providing access to the Xilinx ML605 FPGA board.

REFERENCES

[1] Martin Lagerholm, Carsten Peterson, Guido Braccini, Lars Edenbr, and Leif Srnmo, "Clustering ECG complexes using Hermite functions and self-organizing maps," *IEEE Trans. Biomed. Eng*, vol. 47, pp. 838–848, 2000.

[2] David G. Mrquez, Abraham Otero, Paulo Flix, and Constantino A. Garca, "On the Accuracy of Representing Heartbeats with Hermite Basis Functions," in *BIOSIGNALS*, Sergio Alvarez, Jordi Sol-Casals, Ana L. N. Fred, and Hugo Gamboa, Eds. 2013, pp. 338–341, SciTePress.

[3] G. Braccini, L. Edenbrandt, M. Lagerholm, C. Peterson, O. Rauer, R. Rittner, and L. Sornmo, "Self-organizing maps and Hermite functions for classification of ECG complexes," in *Computers in Cardiology 1997*, 1997, pp. 425–428.

[4] Tran Hoai Linh, Stanislaw Osowski, and Maciej Stodolski, "On-line heart beat recognition using Hermite polynomials and neuro-fuzzy network," *Instrumentation and Measurement, IEEE Transactions on*, vol. 52, no. 4, pp. 1224–1231, 2003.

[5] Tran Hoai Linh, Stanislaw Osowski, and Maciej Stodolski, "On-line heart beat recognition using Hermite polynomials and neuro-fuzzy network," *Instrumentation and Measurement, IEEE Transactions on*, vol. 52, no. 4, pp. 1224–1231, 2003.

[6] Sameer D. Sahasrabuddhe, *A competitive pathway from high-level programs to hardware.*, Ph.D. thesis, IIT Bombay, 2009.

[7] S. D. Sahasrabudhe, S. Subramanian, K. Ghosh, K. Arya, and M. P. Desai, "A c-to-rtl flow as an energy efficient alternative to the use of embedded processors in digital systems," in *DSD 2010*, 2010, pp. 147–154.

[8] T. Rinta-Aho, M. Karlstedt, and M.P. Desai, "The clicktonetfpga toolchain," in *USENIX ATC-2012*. 2012, USENIX Association, Berkeley CA.

[9] George B Moody and Roger G Mark, "The impact of the MIT-BIH arrhythmia database," *Engineering in Medicine and Biology Magazine, IEEE*, vol. 20, no. 3, pp. 45–50, 2001.

[10] M. Jacobsen and R. Kastner, "RIFFA 2.0: A reusable integration framework for FPGA accelerators," 2013, vol. 23, pp. 1–8.

[11] A. Gil, G. Caffarena, D. Marquez, and A. Otero, "Hermite polynomial characterization of heartbeats with graphics processing units," *IWBBO 2014*, 2014.