DSP optimization techniques for LCDK with focus on IoT applications

Arjun
Final Design Project
EE113D - Digital Signal Processing Design

***Abstract*—This paper discusses improvements made to the LCDK development workflow through a series of compute optimizations, implementation of I^{2}C, UART and GPIO communication protocols, and the introduction of Deep Neural Networks(DNN) exported from Tensorflow. We have been able to achieve between 5x-7.5x speedup in FFT execution times, 2x speedup in math ops, and 22x speed up in vectorized math ops. In addition, we have implemented support for 400kHz I^{2}C with bus-buffers, USB-UART, UART and GPIO. We have also implemented a 8-layer DNN that classifies human walking "labels = stationary, walking" in real-time with F-measure 0.912 from 2000ms of wrist-worn accelerometer data sampled at 16 samples/sec and transferred over Bluetooth. We hope that these methodologies prove to be a valuable addition to the modern DSP engineer's toolkit.**

***Keywords*—DSPLIB, MATHLIB, IMGLIB, I^{2}C, GPIO, UART, Tensorflow, Deep Learning, Deep Neural Network, Multilayer Perceptron, ReLU activation.**

I. Introduction

A. History

For the better part of this decade, traditional consumer-grade DSP devices have provided end-node capability to cloud services, having played the role of data-acquisition agents or telecommunication systems. But as this landscape changes with the advent of the Internet of Things (IoT), the role of DSP systems will again be redefined as they adapt to the newer constraints that IoT imposes. Faster compute capability and a tighter energy budget, coupled with the need for ubiquitous connectivity and self-agency are already pushing[7] system and application developers to move out of the cloud and back into the users' pocket. For DSP systems to remain relevant in this latent landscape, there is an urgent need to bring state-of-the-art capabilities to the domain of traditional digital signal processing. This paper discusses the implementation of few such technologies that aim to make the LCDK platform faster, better connected and more intelligent. This goal has been subdivided into three categories, each of which will be individually addressed throughout the length of the paper. These categories have been selected based on the most urgent needs today, which include a need for better optimized *Compute Performance*, fast, reliable and redundant *Communication Protocols*, and a sophisticated machine learning approach through *Deep Neural Networks*.

B. Global Constraints

1) Deep Neural Network: For a fully connected network (implemented here), memory and storage consumption and increases by $O(n^2)$ for n additional neurons per layer, where each neuron-weight is stored as a 32-bit float. Although with 128MB DDR2 RAM, approximately 2^{25} weights can be stored in memory, system performance begins to suffer due to page-faults and L2 (256KB, 2^{16} weights) cache-misses. In addition, the initial transfer overhead for larger model sizes is much larger due to low throughput on the FT232 emulator/connector.

2) Communication Protocols: The primary limitation here is hardware-dependant. Since the LCDK multiplexes various channels and protocols through the same I/O controller, simultaneous use of I^{2}C and GPIO is not possible. In addition, GPIO functionality is implemented in software using bit-banging, which makes the system CPU bound and susceptible to errors or undefined behavior caused due to interruption of interrupt-intolerant code. Next, since I^{2}C is a short-distance communication protocol, using cables longer than $15cm$ for high-speed (1000Kbit/s) data transfer renders the system useless due to noise over the transmission line. This issue can be solved by introducing pull up resistors and capacitors on the bus, but may be undesirable due to increased system complexity.

3) Compute Performance: While the use of external libraries such as DSPLIB, MATHLIB and the GNU Scientific Library (GSL) is convenient and effortless in many ways, it is also the introduction of foreign code within the developers' workspace. As such, the burden of responsibility falls upon the developer for keeping up-to-date with community discovered bugs, updates and known issues. Fortunately, all libraries used here are actively maintained and new issues are discovered and fixed regularly. Further information can be found on the community web-page[9].

II. Motivation

The need for this project the arose due to a requirement for efficient and streamlined development tools which are otherwise inaccessible due to limited documentation or closed-off in a proprietary development environment setting. While other RISC based ARM platforms such Raspberry Pi or BeagleBone already provide much of the functionality developed here (including Tensorflow for ML and SciPy for DSP), they fail to provide low-level access and fine-grain control of hardware

or even the sheer compute capability of the C6748 DSP. On the other hand, although some progress has been made in running Linux kernel on the LCDK, it is important to note that the Linux environment runs on the weaker ARM9 OMAPL138. Thus, any functionality gained by running Linux is immediately lost to sub-power performance due to lack of access to the DSP. It should be noted however, that the DSP can be enabled when running Linux, however this is not trivially achievable, and adds significant overhead to the system complexity. More information about enabling the DSP under Linux can be accessed here[2].

III. Approach

Given that compute performance was a key objective in this project, we adopted a middle-out[1] design strategy by building tools and interfaces around those tools that enhance capability while introducing minimal system overhead. Middle-out design is more complex and requires maturity of knowledge pertaining to the existing system, but allows for deep restructuring without the overhead of a ground-up redesign.

A. ~~Team~~ Project Organization

We strongly felt that middle-out design was the optimal approach due to inter-dependencies between several sub-components. For example, while it would be beneficial to implement a highly optimized matrix-multiply operation, those performance improvements would not translate to other algorithms. In general, improving LCDK performance across the board would mean countless hours of tinkering with a multitude of algorithms, each with it's own unique degree of optimizability. Alternatively, one could heavily optimize only matrix-multiply and then successfully (and approximately!) transform most problems into their matrix-multiply analogs, which is what we did with neural networks (later we will show that neural networks are Turing complete). In general our strategy was to transform problem-sets into neural-network analogs, and process them in real-time using the feed-forward step on our matrix-multiply optimized neural network.

B. Standard

A core component of this project included implementing de jure standards such as I^2C, UART and GPIO communication protocols. Other components required the implementation of well-known methodologies such as feed-forward step in a multilayer-perceptron, or de facto standards such as neuron activation functions in a neural network (tanh, reLU, sigmoid).

C. Theory

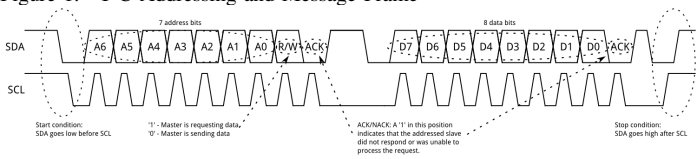
1) Compute Performance: For a given operation, we measure compute performance of as a function of system resources consumed - such as memory, CPU clock-cycles elapsed, and bus usage. Together, these factors contribute to overall system throughput and latency. In order to effectively improve latency, we intend to record and benchmark lower level metrics such as memory and clock cycles for a series of publicly available libraries. These include TI provided assembly implementations of DSPLIB and MATHLIB, along with math.h from the GNU compiler.

2) Communication Protocols: Theoretical basis for the I^2C, UART and GPIO protocols are discussed below.

The I^2C Protocol

The Inter-integrated Circuit (I^2C) Protocol[6] is an asynchronous protocol that allows multiple slave devices to communicate with multiple master devices on the same bus. Operating in either 100kHz or 400kHz mode, the LCDK exposes two independent I^2C buses, multiplexed with SPI on the J15 connector header. I^2C is a intra-bus connection protocol, therefore enabling long distance communication requires the addition of bus-buffers that circumvent the issue of bus capacitance. I^2C messages comprise two types of

Figure 1. I^2C Addressing and Message Frame



frames, namely an address frame and a message frame. Together, these frames provide the receiver all the information it needs to store the incoming byte into the requested location. Data is sent over the SDA channel once SCL has been set to low, and it polled again after SCL is set to high. The address frame is always sent first. I^2C implements 7-bit and 10-bit addressing.

The UART Protocol

Universal Asynchronous Receiver/Transmitter (UART) is a data format and transmission speed configurable serial data transfer protocol. Data is transmitted bit-wise and reassembled at the destination into a complete word. After the transmission of a complete word, a stop bit it sent to the receiver. The data frame in the figure below illustrates this implementation.

Figure 2. UART Data Frame

GPIO

By definition, General-purpose input/output (GPIO) implements a generic protocol-less interface, with ability read or write HIGH/LOW voltages to a given set of pins. In this way, they are a building block for higher-level protocols. The ability to be able to demonstrate fine-control of these pins indicates the potential capability of being able to implement low-bandwidth interfaces of any of type. Although GPIO is quite flexible, it is fully CPU bound, since every event on the bus that generates an interrupt has to be handled by the CPU immediately.

3) Deep Neural Network: The **Universal Approximation Theorem**[4] for artificial neural networks states "A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of R^n, under mild assumptions on the activation function". This indicates that neural networks can represent a wide variety of interesting functions for well chosen weight-parameters. We leverage this powerful capability to transform complex DSP algorithms into weight matrices for neural networks and implement these networks using efficient matrix multiplication operations.

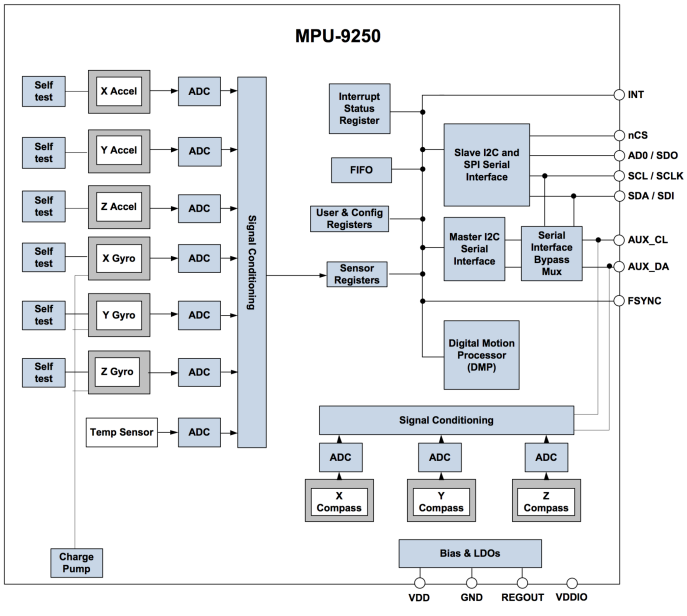
D. Operation - Software/Hardware

1) Compute Performance: Included in the file profiler.h are timing functions that count clock-ticks, and as such can be used for creating small and effective benchmarks (with clock-cycle level accuracy) when profiling LCDK code. Cycle count is independent of clock frequency and therefore a better metric for work-done than elapsed time. This functionality is implemented by keeping track of the clock registers on the processor and plays a pivotal role in bench-marking/profiling code. Effective use of external libraries such as DSPLIB, MATHLIB, and IMGLIB (all of which provide assembly-optimized implementations of commonly used DSP and math functions) mandates precise and accurate bench-marking, which can be achieved using this library.

2) Communication Protocols: In this section, we will omit the overview for UART and GPIO since the theory covers the principle of operation, and the procedure covers the concrete implementation.

I^2C with 9-DOF MPU-9250 Next, we will interface the

Figure 3. MPU-9250 System Overview



InvenSense MPU-9250 9-DOF with the LCDK over I^2C. The figure above illustrates the system diagram for the MPU-9250. The 9250 chipset is a fairly complex set of DSPs in it's own right, with inbuilt capability for motion sensing, step-detection, and threshold detection. Programming the 9250 is fairly straight-forward, it requires setting it's internal register values over I^2C. The register map is illustrated in figure 4, setting specific values leads to different sampling rates, interrupt (on data ready), or interrupt on motion detected. For example, a typical use case is for getting accelerometer and gyroscope data at 184Hz. This can be achieved by writing 0x0 to the PWR-MGMT-1, setting the CLKSEL[2:0] vector to the appropriate mask, and then setting PWR-MGMT-1 to 0x1.

Figure 4. MPU-9250 Register Map

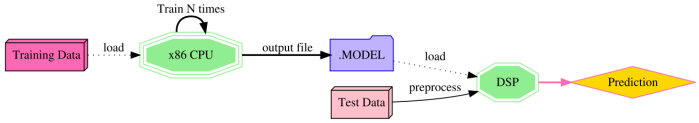
InvenSense | MPU-9250 Register Map and Descriptions | Document Number: RM-MPU-9250A-00 Revision: 1.4 Release Date: 9/9/2013

Addr (Hex)	Addr (Dec.)	Register Name	Serial I/F	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
66	102	I2C_SLV3_DO	R/W	I2C_SLV3_DO[7:0]							
67	103	I2C_MST_DELAY_CTRL	R/W	DELAY_ES_SHADOW	-	-	I2C_SLV4_DLY_EN	I2C_SLV3_DLY_EN	I2C_SLV2_DLY_EN	I2C_SLV1_DLY_EN	I2C_SLV0_DLY_EN
68	104	SIGNAL_PATH_RESET	R/W	-	-	-	-	-	GYRO_RST	ACCEL_RST	TEMP_RST
69	105	MOT_DETECT_CTRL	R/W	ACCEL_INTEL_EN	ACCEL_INTEL_MODE	-		-		-	
6A	106	USER_CTRL	R/W	-	FIFO_EN	I2C_MST_EN	I2C_IF_DIS	-	FIFO_RST	I2C_MST_RST	SIG_COND_RST
6B	107	PWR_MGMT_1	R/W	H_RESET	SLEEP	CYCLE	GYRO_STANDBY	PD_PTAT	CLKSEL[2:0]		
6C	108	PWR_MGMT_2	R/W	-		DIS_XA	DIS_YA	DIS_ZA	DIS_XG	DIS_YG	DIS_ZG
72	114	FIFO_COUNTH	R/W	-		FIFO_CNT[12:8]					
73	115	FIFO_COUNTL	R/W	FIFO_CNT[7:0]							
74	116	FIFO_R_W	R/W	D[7:0]							
75	117	WHO_AM_I	R	WHOAMI[7:0]							
77	119	XA_OFFSET_H	R/W	XA_OFFS [14:7]							
78	120	XA_OFFSET_L	R/W	XA_OFFS [6:0]							-
7A	122	YA_OFFSET_H	R/W	YA_OFFS [14:7]							
7B	123	YA_OFFSET_L	R/W	YA_OFFS [6:0]							-
7D	125	ZA_OFFSET_H	R/W	ZA_OFFS [14:7]							
7E	126	ZA_OFFSET_L	R/W	ZA_OFFS [6:0]							-

Table 1 MPU-9250 mode register map for Gyroscope and Accelerometer

3) Deep Neural Network: Figure 5 illustrates the setup for training, exporting and importing a Tensorflow[8] model directly into the CCS. This setup allows us the flexibility and speed of being able to train deep neural networks using the latest mechanisms on host CPUs+GPUs, and then moving weight matrices from trained models as 32-bit binary-float arrays, nicely wrapped here as a .model file. Source on GitHub provides clean/hassle-free C/python reading and writing utilities for .model files. In CCS, it is as simple as including "DNN.h", and calling DNNRead(), or DNNWrite(). On the

Figure 5. System Overview - Tensorflow + DSPLIB framework



LCDK side, DNNs are read into memory and input data is passed to the DNN using the function int DNNPredict(float *data), which returns an integer value for the predict class, or negative -1 if there is an internal error.

E. Procedure - Plan and Implementation

1) Compute Performance: We began optimization by using TI provided libraries DSPLIB, MATHLIB, and IMGLIB, and comparing their performance with the TI standard provided

headers such as math.h and the infamous fft.h. We then wrote tests bench-marking multiple functions, with varying array sizes and operation-count. The figure below illustrates example usage for the profiler code.

Figure 6. Example Profiler Code Usage

```
#include "profiler.h"

int main(void) {

  const clock_t start_time = getClock();

  function_to_run(); // DO SOME INTERESTING CALCULATIONS HERE

  const clock_t stop_time  = getClock();

  stats(stop_time, start_time);
}
```

2) Communication Protocols: On the LCDK, in order to enable any of the communication protocols, an interrupt service routine has to be set up with an interrupt mask, along with an interrupt handler for the given event. If the interrupt is going to occur on the communication bus, then the pin-mux has to be set as well. Although one could set up multiple interrupt masks for UART and I2C simultaneously, one of the interrupts will not fire due to the multiplexing limitation.

I²C with 9-DOF MPU-9250

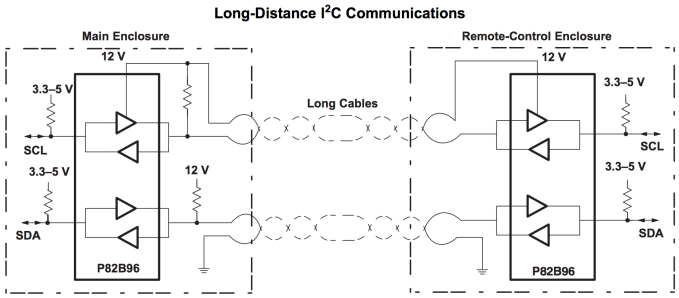
Using StartWare samples for the LCDK, we begin by reverse-engineering the I²C example code, exhaustively isolating and understanding each dependency. Once we have a good handle on the code, we write simple subroutines to send data bytes to the MPU-9250 9-DOF over the I²C bus. The 9-DOF is connected to pins 13 (SDA) and 15(SCL) on the J15 header expansion board. The 9-DOF address is fixed at 0x69. Starting data acquisition from the 9-DOF requires setting up the pin-mux, appropriately covering the Interrupt Service Routine with Interrupt Masks, and enabling the I²C controller on the desired data address. Once the bus is set, data is written to the 9-DOF registers over I²C and an ACK signal is received over the bus.

Long Distance I²C

Although I²C implements clock stretching, bus errors are still fairly typical for transmission lines longer than $15cm$ (@400kHz). These bus errors are a direct consequence of clock skew caused due to bus capacitance and noise on the data channel. In order to mitigate these effects without sacrificing throughput, we propose the addition of six circuit components, which include four pull-up resistors and two TI I²C P82B96 bus buffers. The figure below illustrates the setup, where "Long Cable" is replaced with twisted pairs from a CAT-5E Ethernet cable and "12V" is replaced with "3V3". Since all voltages are a standard 3.3V, all pull up resistors are chosen to be in the range of 1K-2.2K. SDA and SCL pins connected to the 'Main Enclosure' connect with the powered master device, whereas the ones connected to the 'Remote Enclose' connect to the remote slave. With this setup, we are able to receive error-free I²C over $8m$, although the untested theoretical limit is calculated to be $100m$.

UART Protocol

Figure 7. P82B96 + twisted pair I²C Bus Buffer

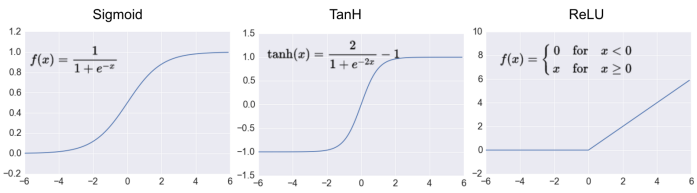


The LCDK hosts three UART controllers, one of which is used by the TI XDS100v2 emulator. We have been able to successfully use the other two. The first UART interface is exposed over USB-UART, and can be connected to through a USB-to-mini-USB cable. In CCS, once the pin-mux, interrupt service routines, and interrupt masks have been set, data can be sent and received at 115.2kbps baud rate over the UART port. The second UART port needs to be manually enabled by shorting out solder pads R206 and R209 on the back of the LCDK and connecting to pins 13 [RXD] and 15 [TXD] on the J15 expansion header.

3) Deep Neural Network: The following section documents the implementation and optimization process for matrix-backed neural networks on the LCDK.

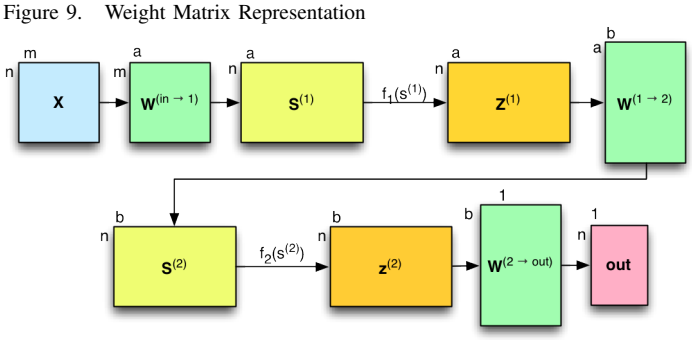
Activation Functions We begin by implementing an activation function. Of the various available options, $reLU(x)$ activations are known to be quite robust to input variance, and are efficient to implement. In interest of completeness, we also implement $tanh(x)$ and $sigmoid(x)$ functions, although these will not be used further, due to the additional need to normalize the inputs when using them and complex implementation. We then further optimize these functions using math ops from MATHLIB. The graph and equations below illustrate the these activation functions.

Figure 8. Neuron Activation Functions



Feed-forward step: Next, we implement the matrix multiplications. Equation (1) depicts the matrix multiplication between the inputs and the first hidden layer. The weights of all hidden layers are stored in memory and are of the size $m * 1 * 4$ bytes where m is the number of features in the input. For all the cases below, $n = 1$ since the batch-size is 1 (since we are making one inference at any moment in a real-time system).

Figure 9. Weight Matrix Representation



$$S^{(1)} = XW^{(in \to 1)} \tag{1}$$
$$Z^{(1)} = f_1(S^{(1)}) \tag{2}$$
$$S^{(2)} = Z^{(1)}W^{(1 \to 2)} \tag{3}$$
$$Z^{(2)} = f_2(S^{(2)}) \tag{4}$$
$$\hat{y} = f_{out}\left(Z^{(2)}W^{(2 \to out)}\right) \tag{5}$$

Training in Tensorflow: With the inference code implementation complete, we proceed to train our neural network in Tensorflow. We have pre-recorded accelerometer data (sampled every $62.5ms$) from a smartwatch. The data contains ambulation information. The table below illustrates some raw-data from the dataset.

$accel_x$	$accel_y$	$accel_z$	label
8.707711	1.149216	1.879448	walking
9.730036	-0.694318	2.047042	stationary

After performing a training/testing split, the raw training data is fed into the neural network and trained using back-propagation[3] with Adam[5] gradient descent. All source code has been made available on GitHub, along with tutorials to create new models. For reference, the table below illustrates the hyper-parameters tuned for our network. For a two second window, $32 * 3 = 96$ samples are recorded at 16samples/sec.

Number of Inputs:	96
Number of Outputs:	2
Hidden Layers:	768, 384, 192, 96, 48, 24, 12, 6
Learning Rate:	0.005
L2 Regularization:	0.001
Batch Size:	114
Epochs:	897

Once the model has been built, it can be exported to the LCDK and mapped directly into memory using the CCS Memory Mapper utility. At this point, we enable our UART code, and read data from the slave device. The slave device is a Bluetooth controller, connected wireless to an Android device (smart-watch). Accelerometer data is streamed in real-time from the smart-watch directly into the input layers of the DNN on our LCDK.

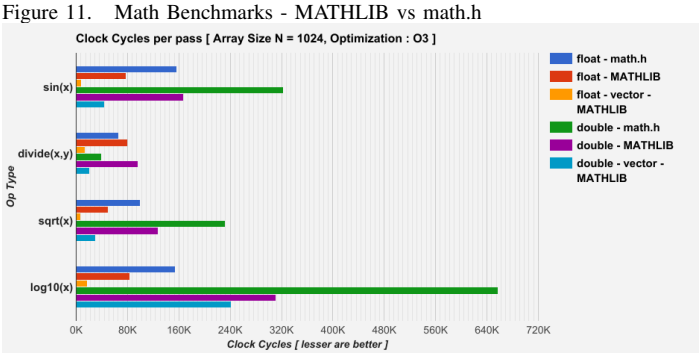
IV. Results

1) Compute Performance: Figure 10 illustrates the performance difference between DSPLIB provided FFT implementations vs the baseline fft.c. DSPLIB provides two separate implementations of FFT, the one in yellow is a decimation in time implementation for radix-2, whereas the one in blue is a mixed radix implementation. Points in read are from the baseline fft.c. The performance gap between the the baseline and optimized versions varies between 5x to 7x. I should be noted that although it was expected that FFT radix-2 perform better than mixed-radix, mix-radix consumes more memory than radix-2, and such benefits where in-place (radix-2) implementation suffers. Another thing to note is the slope of the two groups, which indicates the effect of bin-size on execution time. Since the number of executions are constant overall, this slope is indicative of cache-misses in the baseline code, which are minimal in the DSPLIB optimized versions.

Figure 10. FFT Benchmark - DSPLIB vs Rulph's CS31 fft.c

Figure 11 illustrates a performance evaluation between TI provided MATHLIB and TI compiler provided math.h. The x-axis represents number of clock cycles elapsed, and therefore faster code would be lesser. Comparisons are made for both single and double precision functions, and an oblivious improvement of nearly 2x is seen when using floats instead of doubles. It can also be seen that the MATHLIB implementations are another 2x faster, whereas vectorized implementations of all functions are nearly 15x to 22x faster. This is due to the pipeline and caching optimizations performed by MATHLIB, sources for which are available on this project's GitHub repo and on TI's website.

Figure 11. Math Benchmarks - MATHLIB vs math.h



2) Communication Protocols: All three protocols [I^2C, UART, GPIO] have been implemented and are known to be working well. All source code is available on GitHub, along with steps to reproduce the exact setup used in this project.

3) Deep Neural Network: Next, we test the efficacy of this system on a test data-set, and compare those results with two different instances of a RandomForest Classifier, one with 10 trees and another with 5000 trees. The training/test split (2149sec/114sec) is *identical* for all three classifiers. The table below illustrates the results.

Metrics	**DNN**	RForest (n=10)	RForest (n=5000)
Accuracy	0.912	0.746	0.763
Precision	0.915	0.764	0.777
Recall	0.911	0.749	0.767
F-Measure	**0.912**	0.747	0.762

Over just a two second window-frame, the neural network was able to distinguish between stationary [sitting, standing, laying down] and walking positions with around 91.2% accuracy, while RandomForest(n=10) could manage a meagre 74.7%. Even when given the opportunity to over-fit and memorize the entire training set (n=5000), the RandomForest classifier did not improve. This confirms two distinct hypotheses: first, the test-data is distinct enough from training that memorizing does not seem to improve the prediction score; and second, the neural network is indeed learning, not just memorizing the data-set either.

This test succinctly highlights one of the biggest strengths of neural networks, i.e. the ability to extract features implicitly from within a raw-signal, without the need for additional prepossessing and feature-extraction.

REFERENCES

[1] Autodesk Inventor Blog. "Top-down, bottom-up, middle-out design". In: URL: https://tinyurl.com/ycqn9ls8.

[2] OMAP-L138 LCDK Linux Software Developer's Guide. *Programming the C6748 DSP*. http://processors.wiki.ti.com/index.php/OMAP-L138_LCDK_Linux_Software_Developer's_Guide.

[3] S. i. Horikawa, T. Furuhashi, and Y. Uchikawa. "On fuzzy modeling using fuzzy neural networks with the back-propagation algorithm". In: *IEEE Transactions on Neural Networks* 3.5 (Sept. 1992), pp. 801–806. ISSN: 1045-9227. DOI: 10.1109/72.159069.

[4] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (Jan. 1991), pp. 251–257. DOI: 10.1016/0893-6080(91)90009-t. URL: https://doi.org/10.1016/0893-6080(91)90009-t.

[5] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: http://arxiv.org/abs/1412.6980.

[6] F. Leens. "An introduction to I2C and SPI protocols". In: *IEEE Instrumentation Measurement Magazine* 12.1 (Feb. 2009), pp. 8–13. ISSN: 1094-6969. DOI: 10.1109/MIM.2009.4762946.

[7] TechCrunch. *Tensorflow Lite DSP*. https://techcrunch.com/2017/05/17/googles-tensorflow-lite-brings-machine-learning-to-android-devices/.

[8] Tensorflow. *Getting Started With Tensorflow*. https://www.tensorflow.org/get_started/get_started.

[9] TI Community Forums Wiki. *C674x DSPLIB Known Issues*. http://processors.wiki.ti.com/index.php/C674x_DSPLIB_Known_Issues.