Duration: July - November 2021,
Intern: Burak Topçu
Sundance Multiprocessor Technology LTD www.sundance.com

HiPEAC Internship Report

Power Profiling a Custom Application on VCS-1

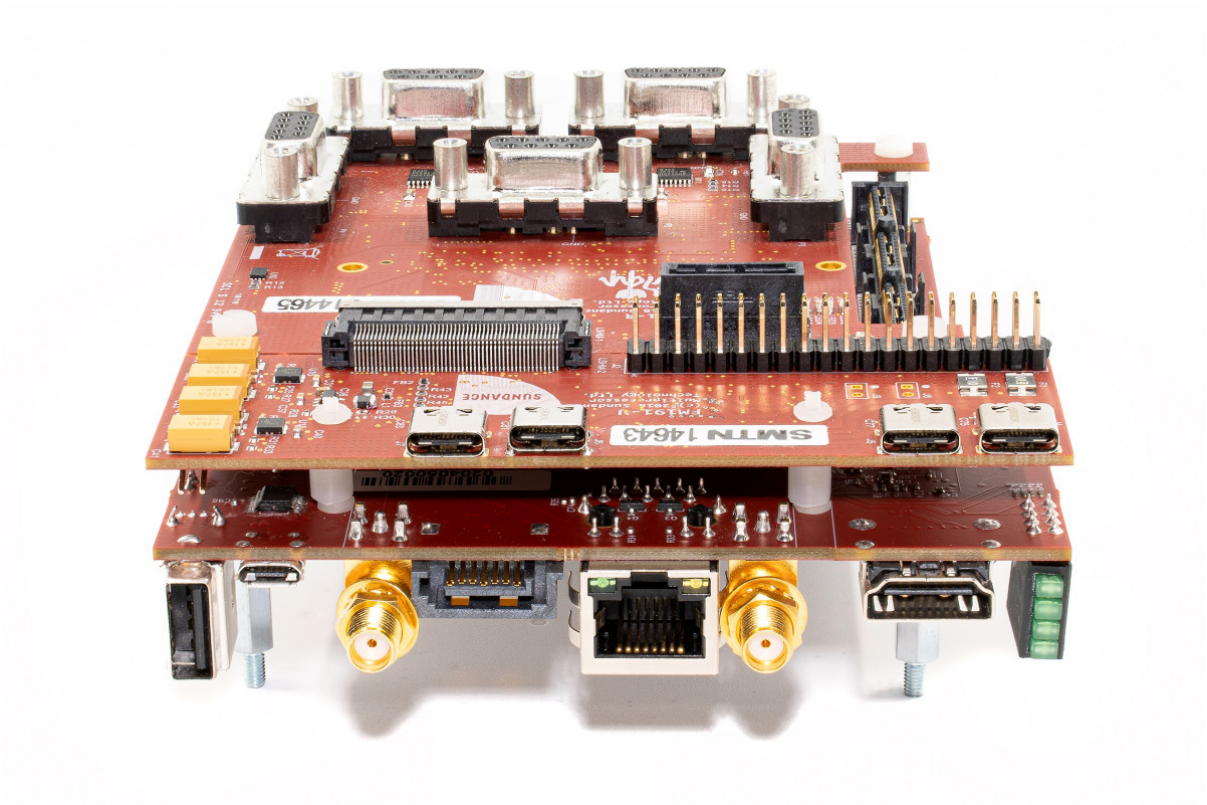

Figure 1: Vision Control and Sensor board (VCS-1).

1. Introduction

a. Contributions of TULIPP Book

Towards Ubiquitous Low-Power Image Processing Platforms (TULIPP) book [1] describes the main reasons for the transition from current processors to the all-in-one platforms as in the VCS-1 board. Major reasons for this platform transition are that data that is processed increases day by day because image processing algorithms spread more and, algorithms become more sophisticated which requires more powerful cores to process increasing data with high performance.

Timoteo Garcia Bertoa, who is the author of chapter 3 in the book, explains how the new SoCs integrated solves the mentioned challenges. SoCs includes lots of application-specific processors with memories inside. To illustrate, Zynq Ultrascale+ includes quad cores of ARM Cortex-A53, real-time processors of ARM-Cortex-R5, a GPU, memory, and controller in addition to lots of connection peripherals such as USB 3.0, PCIe 1,2 and SPI on the PS side. Furthermore, there is a PL side including storage such as BRAM, I/Os, transceivers and programmable logic resources in those SoCs. As one can infer easily, these SoC devices have the capability of processing data in various conditions such as applications requiring highly parallelized implementation or low power consumption. Moreover, EMC2 boards enhance the usage of those SoCs by developing a state-of-art architectural design with many peripherals to be used for lots of applications.

As a result, EMC2 with FM191 extension board enables developers to overcome the bottlenecks of applications with the mentioned properties and benefits.

b. About the Internship

I saw the Beyond TULIPP internship program assigned by Sundance on the HiPEAC job facilities page. Then, I sent a proposal to Sundance by introducing myself, including my background knowledge and explaining my plans for the internship process. Afterwards, we agreed on the internship, and I have started preliminary work of the internship. In this preliminary work, I learnt the Zynq Ultrascale+ chips in terms of architectural manner, unified coding environment of Vitis HLS and some guidelines to optimize the applications. Then, I started the internship on 1st July and completed my internship on 30th October. I explain what I did in the internship duration in the remaining sections of this report.

2. VCS-1 and LynSyn Lite Devices

a. About VCS-1

VCS-1 [2] (Vision, Control and Sensor solution board) consists of an EMC2-DP board [3] and FM191 expansion card shared in figure 2 and figure 3 respectively. EMC2-DP has a PCIe/104 OneBank™ Carrier that is compatible with Trenz SoC modules (mine is Zynq US+), an expansion to access all signals of VITA 57.1 FPGA Mezzanine Card (FMC) compliant low-pin count (LPC) connectors, general-purpose I/O pins and LEDs. Also, there are some external interface connections such as USB 2.0, HDMI, 1Gb Ethernet and SATA. Also, the FM191 expansion card provides FM191-R that provides Analogue and Digital GPIOs for robotics, motors and sensor applications (requiring ADC-DAC conversions). FM191-U that is an expansion of FM191-R adds 4 USB C ports to increase connection capability and 40 I/O pins additionally. This FM191 circuitry can be used with FMC-LPC connectors.

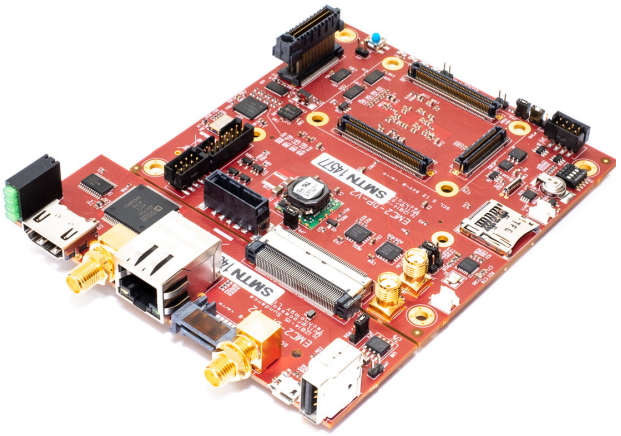

Figure 2: EMC2 board provided by Sundance.

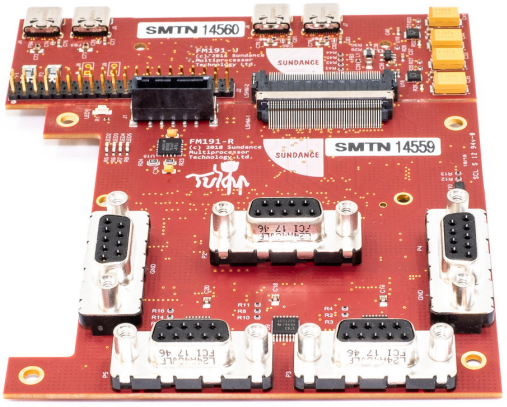

Figure 3: FM191 extension board including many peripherals.

Due to the hardware resource capabilities compatible with application-specific circuitries mentioned above and lots of interfaces such as GPIOs, USB-C, ethernet and so on. This board is preferable for various commercial, industrial and military usage applications.

b. About LynSyn Lite

LynSyn lite [5] shared in figure 4 is a device to be used for measuringconsumed power with 3 different sensors for real-time applications. It samples program counters of the system processor up to 10KHz and correlates the power measurements with source code for mapping power consumption of the applications. There is two power measurement methods with either USB-micro cable or JTAG connection via this device. One can observe power consumption, applied voltage and applied current of each code section for an embedded application by adding breakpoints to the code with the help of a JTAG connection. USB-micro cable connection provides limited power measurement compared to the JTAG connection such that one can just measure applied current, voltage and consumed power in specific time duration. Lynsyn Lite is compatible with both Linux and Windows operating systems and provides an open-source software used for both samples and visualizes measurement results via its GUI in [6].

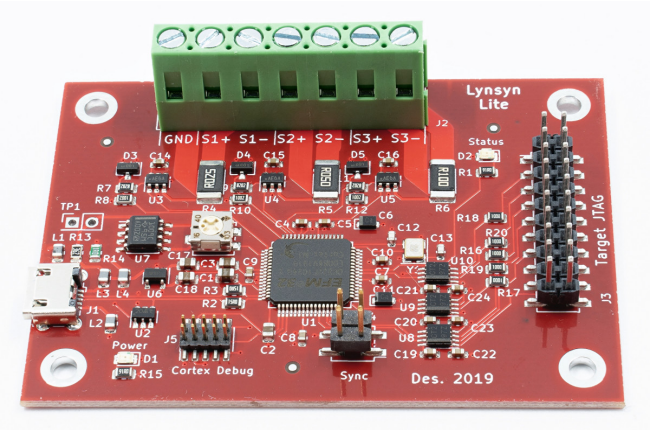

Figure 4: LynSyn Lite device used to profile power consumption of the applications.

3. Deep Neural Network Implementations with MNIST

Deep Neural Network is used commonly by completing popular DNN architectures with its various configuration. In the internship duration, I want to implement 3 different DNN architectures try to observe their power consumption on VCS-1. Table 1 reveals the NN architecture configurations which are inspired from [7] work. First of all, I created the network architectures and trained them with MNIST dataset. I could decrease error values of each architecture as shared in figure [5].

Table 1: Neural Network architecture configuration, activation and loss functions

Layer Amount	Perceptrons	Activation and Loss Function
2 layers	300 perceptrons	Tanh and MSE
3 layers	300+100 perceptrons	Tanh and MSE
3 layers	500+300 perceptrons	Softmax and Cross Entropy

```
burak@MRobot:/media/burak/XilinxSSD/Sundance_Internship/DeepNeuralNetwork$ python3 main.py --2_layered_architecture
The total error for 2 layered architecture configuration:  5.61
burak@MRobot:/media/burak/XilinxSSD/Sundance_Internship/DeepNeuralNetwork$ python3 main.py --3_layered_architecture
The total error for 3 layered architecture configuration:  3.34
burak@MRobot:/media/burak/XilinxSSD/Sundance_Internship/DeepNeuralNetwork$ python3 main.py --3_layered_architecture_wi
th_CE
The total error for 3 layered architecture with cross entropy loss:  2.29
burak@MRobot:/media/burak/XilinxSSD/Sundance_Internship/DeepNeuralNetwork$ python3 main.py
The total error for 2 layered architecture configuration:  5.61
The total error for 3 layered architecture configuration:  3.34
The total error for 3 layered architecture with cross entropy loss:  2.29
```

Figure 5: Achieved errors for DNN for the architectures mentioned as table xx

After that point, I tried to re-implement this code with Vitis HLS unified environment. However, since I attended the HiPEAC Student Challenge event, I could not complete this part. Since I still have the VCS-1, I think I will complete this part of the project for future and share recorded power consumption with the obtained weights and biases on GitHub [8].

4. HiPEAC Student Challenge VIII

HiPEAC student challenge is a student challenge event and this year they expected to implement histogram equalization algorithm from the students. In my university, I and some of my friends decided to attend this event to examine performance and power consumption of this algorithm on different platforms. It was an opportunity for me since I had been already a HiPEAC intern and, I would implement the algorithm, make experiments and present my results with my knowledge that I obtained from the internship. I implemented the algorithm and measured the power consumption with a basis image detailed in below [8]. Afterwards, I presented my results in this event in Lyon.

a. Histogram Equalization Algorithm

Histogram equalization [9] is an image processing algorithm to manipulate image pixels with an algorithm to increase the understandability of the grayscale images. The algorithm consists of four main parts.

Firstly, histogram values of each image pixel are calculated by counting the occurrence rate of each pixel value. For example, if the image's pixel depth is 8 bit, we have 255 values in total and calculate the occurrence rates of each pixel value. After calculating the histogram values that can be perceived as probability distributions of each pixel, we need to calculate cumulative distribution functions for those pixel values. CDFs of each pixel value in the range [0,255] is calculated by just adding the histogram value of the previous ones to the current one. After CDFs are calculated, we need to find the non-zero CDF for the minimum pixel value. Lastly, we need to apply the formula given below to calculate equalized histogram values:

$$h(v) = \frac{CDF(v) - CDF_{min}}{M \times N - CDF_{min}} \times (L - 1)$$

where M and N are the dimension parameters of the image and L is the pixel depth. The resultant h(v) values are the histogram equalized result of each pixel of the corresponding image.

b. Implementation

We have carried out work to perform performance and power measurement comparisons for this algorithm among different hardware platforms. To increase performance, I firstly focused on the algorithm sub-patterns that can be parallelized.

Calculation of CDF and finding non-zero pixel value steps result in 255 clock cycles at most because we calculate CDF for each pixel value in the range [0,255] and find non-zero pixel value in this range. However, calculating histogram values by counting occurrence rate of each pixel takes more clock cycles if it is implemented in a sequential way. Similarly, re-mapping the histogram equalized values for those pixels should be implemented in a parallel way to increase algorithm performance.

Firstly, one has to decide how many pixels will be counted in each clock cycle for the histogram calculation. Figure 6 represents the implementation of our experimental work. I have also implemented the algorithm in a more parallelized way. However, those implementations were just tried with the run-time simulation environment of the Vivado tool. Hist is a vector consisting of 256 elements where each element has 24-bit length for a 4096*4096 image.

$$\text{hist}\Big[\text{image}\big[y\big][x\big]\Big] = \text{hist}\big[\text{image}[y][x]\big] + 1;$$

$$\text{hist}\Big[\text{image}\big[y\big][x+1\big]\Big] = \text{hist}\Big[\text{image}\big[y\big][x+1\big]\Big] + 1;$$

$$\text{hist}\Big[\text{image}\big[y\big][x+2\big]\Big] = \text{hist}\Big[\text{image}\big[y\big][x+2\big]\Big] + 1;$$

$$\text{hist}\Big[\text{image}\big[y\big][x+3\big]\Big] = \text{hist}\Big[\text{image}\big[y\big][x+3\big]\Big] + 1;$$

Figure 6: Histogram calculation of the pixel values

Similar to the above mapping to the calculation of the histogram values, resultant histogram equalized values for each pixel is re-mapped as shown in figure 7. In figure 7, I have shared a more parallelized version for the re-mapping such that 16 equalized histogram values re-mapped to the corresponding pixel values in each clock cycle. Implementing the last part in a more parallel way requires more hardware resources since there are division and multiplication operations that are compared to the adder and subtractor circuitry. Hence, I re-mapped 4 equalized pixel values in each clock cycle.

$$\text{image}\big[y\big][x] = \frac{\text{CDF}\Big[\text{image}\big[y\big][x\big]\Big] - \text{CDF}_{\min}}{\text{width} * \text{height} - 1} * (256-1)$$

$$\text{image}\big[y\big][x+1] = \frac{\text{CDF}\Big[\text{image}\big[y\big][x+1\big]\Big] - \text{CDF}_{\min}}{\text{width} * \text{height} - 1} * (256-1)$$

$$\cdots$$
$$\cdots$$

$$\text{image}\big[y\big][x+15] = \frac{\text{CDF}\Big[\text{image}\big[y\big][x+15\big]\Big] - \text{CDF}_{\min}}{\text{width} * \text{height} - 1} * (256-1)$$

Figure 7: Applying equalization among calculated CDFs and re-mapping them to the image pixels

c. Experiments and Test Results

I have carried out my experiments with an image that includes 512*512 pixels with 8-bit depth intensities for each pixel. In the experiments, I have used the VCS-1 board which includes a Zynq Ultrascale+ chip integrated with EMC2-DP board and LynSyn Lite circuitry. The clock rate of the Zynq US+ chip is set to 50MHz. The SDRAM memories integrated on-chip operates above 800MHz which ensures that we can fetch data from the memory at each clock rate without stalling. Also, power measurements for the embedded application are done via LynSyn lite.

Performance measurements among the different configurations of the algorithm are shared in table 1 and power consumption metrics measured via USB-micro connection of LynSyn lite are shared in figure 8, 9 and 10.

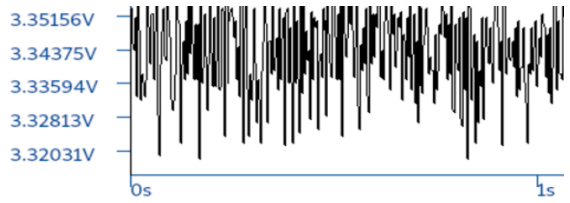

Figure 8: Applied Voltage in the range [0s, 1s]

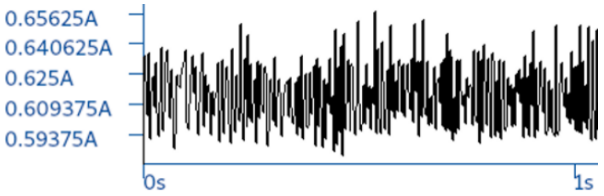

Figure 9: Applied Current in the range [0s, 1s]

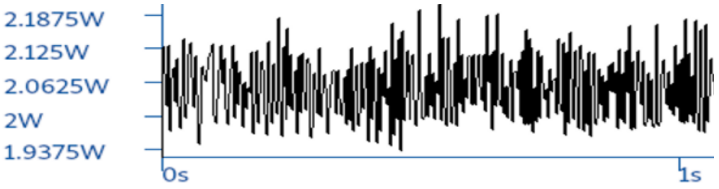

Figure 10: Consumed Power in the range [0s, 1s]

Table 2: Performance measurements of the histogram equalization algorithm with different parallelization configurations (bolded one is the experimental one)

HARDWARE DESIGN SPECIFICATIONS	PERFORMANCE(msec)
• **Histogram calculation -> 4 pixels in each clock** • **CDF_min -> sequential (Calculating 1 pixel value in each clock)** • **CDF calculations -> sequential** • **Histogram equalization -> 4 pixels in each clock**	**10.49**
• Histogram calculation -> 8 pixels in each clock • CDF_min -> sequential (Calculating 1 pixel value in each clock) • CDF calculations -> sequential • Histogram equalization -> 8 pixels in each clock	5.36
• Histogram calculation -> 16 pixels in each clock • CDF_min -> sequential (Calculating 1 pixel value in each clock) • CDF calculations -> sequential • Histogram equalization -> 16 pixels in each clock	2.72
• Histogram calculation -> 32 pixels in each clock • CDF_min -> sequential (Calculating 1 pixel value in each clock) • CDF calculations -> sequential • Histogram equalization -> 32 pixels in each clock	1.39

The bolded result is the experimental one such that the application is created and embedded into the chip with Petalinux. The remaining results are obtained with the run-time simulation environment of the Vivado tool. As a result, the histogram equalization algorithm requires 10.49 milliseconds to process an image that has 512*512 pixels. Also, the belonging hardware consists of Zynq US+ chip consumes 2.047 watt/sec on average.

Implementation details, example images in both .png and .txt (including hexadecimal pixel values) formats are shared in the GitHub repository [8].

d. Performance and Power Measurement Comparisons with Other Devices

Similar to my experiments, we have carried out the same experiment on a CPU and GPU devices. In GPU, experiments are done on the Google Colab which uses Tesla P100 architecture in their cores.

It took 1 millisecond to complete this algorithm on Colab's cloud environment. Since the algorithm takes a too small time to complete its work, the power sampling rate of the nvmlDeviceGetPowerUsage [10] function cannot measure the power consumption of the algorithm reliably.

In the experiments on CPU, the hardware and compiler configurations are as follows:

- C++ 17 is used for the compiler, LLVM OpenMP API version is the 5th version and Ubuntu 5.11.0-38-generic operating system is used.
- Intel Xeon(R) Gold6148 CPUs are used. In the experiments, 20 of those CPUs are used with 40 threads in total.

The algorithm resulted in 6,43 milliseconds with 20 cores for the same image. Figure [xx] represents the performance results of those algorithms on different platforms.

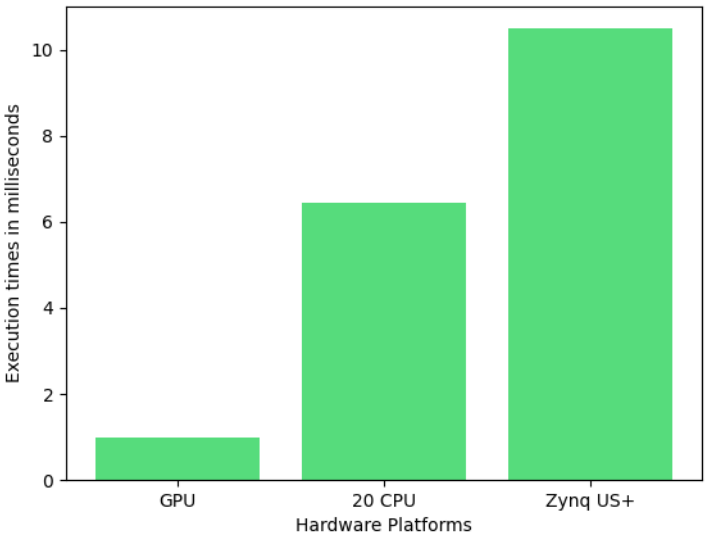

Figure 11: Execution time results for histogram equalization algorithm tested on Colab's Teasla P100 GPU, 20 * Intel CPU and Zynq Ultrascale+

5. AWS - EC2

Amazon Web Server provides an Elastical Compute cloud namely EC2. Users can create their virtual machines with specified resources amount with respect to their demands. In my works, while I was trying to implement more parallelized version of the histogram equalization algorithm, synthesis and implementation did not complete because of the insufficiency of the RAM. Afterwards, Sundance provides me an EC2 account to configure more powerful machine for the experiments.

One of the admins adds additional users' ssh keys to the EC2 account and guest user can create an EC2 instance in this way. After creating the instance, you need to build your machine in terms of dependies, operating system desktop and communication portal. In my work, I have used

remmina application to connect to the my EC2 instance via VNC. VNC has to be installed both local machine and virtual machine. Afterwards, users can access virtual machine via VNC connection and Remmina GUI. Ivica ,who is a member of Sundance, prepared three documentations for AWS EC2 instance creationg, connection between local machine and VM, and building application on VM side for the boards via JTAG tool respectively.

The most sophisticated thing is that, users can debug their applications via JTAG tool between virtual machine and local by connecting device to the local machine and configuring and building the application from the virtual machine.

6. References

[1] M. Jahre, D. Göhringer, and P. Millet, *Towards ubiquitous low-power image processing platforms*. Cham: Springer International Publishing, 2021.

[2] https://www.sundance.com/vcs-1/

[3] https://www.sundance.technology/som-cariers/pc104-boards/emc2-dp/

[4] https://www.sundance.technology/system-on-modules-som/fmc-modules/adc-dac-fmc-modules/fm191/

[5] https://store.sundance.com/product/lynsyn-lite/

[6] https://github.com/EECS-NTNU/lynsyn-host-software/wiki

[7] http://yann.lecun.com/exdb/mnist/

[8] https://github.com/topcuburak/InternshipAtSundance

[9] https://en.wikipedia.org/wiki/Histogram_equalization

[10] https://docs.nvidia.com/deploy/nvml-api/group__nvmlDeviceQueries.html#group__nvmlDeviceQueries_1g7ef7dff0ff14238d08a19ad7fb23fc87