# ECE 498 ICC: IoT and Cognitive Computing

# Spring 2020, Homework 2

**SOLUTION**

## Question 1: IP-Based Hardware Design (10 pts)

Matrix-vector multiplications (MVM) are involved in most of deep neural network algorithms. Large MVM computations can be achieved by partitioning large inputs into small portions and feed into small MVM calculation IPs. Suppose we want to develop an IP-based hardware design for MVM.

1. List one advantage and one disadvantage for IP-based hardware design [**2 pts**]

   Advantage: IP-based designs can be reused in other designs;

   Disadvantage: IP-based designs may not expose as much flexibility as flatten designs.

2. If we want to perform MVM on a matrix of size **2n**×**4n** and a vector with size **4n**, how many compute engine IPs do we need if:

   (a) the computer engine IP can calculate MVM with input sizes **0.25n**×**0.25n** and **0.25n**

   (b) the computer engine IP can calculate with input sizes **0.5n**×**0.5n** and **0.5n**

   You may assume that: we do not reuse IPs and there are enough buffers and adders to store and sum the intermediate results. [**3 pts**]

   (i) $8 \times 16 = 128$ instances of the compute engine IP; (ii) $4 \times 8 = 32$ instances.

3. Consider IP version (b) from part (2) and assume we are allowed to reuse the IP. Each instance of the compute engine IP requires an area $A$ and each use of an IP requires an additional $\frac{1}{4}A$ area. The latency of each IP to complete MVM on its input is $T$. You may assume that $k$ IPs are instantiated, and they can work in parallel. However, each of them needs additional time, $\frac{k^2}{32}T$, due to data congestion. How many IPs should be instantiated to minimize the product of total latency and total area? Assume that the number of IPs is divisible by $k$.

   NOTE: You need to justify that the solution which you find is the global minimum of the objective function.

Give reasons and calculate $Latency \times Area$. Suppose there is no pipelining, and all instantiated IPs run together. [**5 pts**]

Suppose $k$ IPs are instantiated,

$Area = \frac{32}{k} \times \frac{A}{4} + kA$

$Latency = \frac{32}{k}(T + \frac{K^2}{32}T)$,

$Latency \times Area = 32AT(\frac{8}{k^2} + \frac{k^2}{32} + \frac{5}{4})$.

Using Cauchy-Schwarz inequality, $\frac{8}{k^2} + \frac{k^2}{32} \geq 1$, when $k = 4$ ,$\frac{8}{k^2} + \frac{k^2}{32} = 1$. Since $k$ is a positive integer, this is the global minimum point of $Latency \times Area$. The minimum is $72AT$.

# Question 2: A Tiny CNN (25 pts)

Consider the CNN we used in Lab 1 as described in Figure 1:

| Layers | Configuration | Activation Function | Output dimensions (channel, height, width) |
|---|---|---|---|
| convolution | 5x5 filter, stride = 1, no padding | ReLU | (3, 24, 24) |
| max pooling | stride = 2 | - | (3, 12, 12) |
| convolution | 3x3 filter, stride = 1, with padding | ReLU | (3, 12, 12) |
| max pooling | stride = 2 | - | (3, 6, 6) |
| fully connected | input size 108, output size 100 | ReLU | (100, 1, 1) |
| fully connected | input size 100, output size 50 | ReLU | (50, 1, 1) |
| fully connected | input size 50, output size 10 | Softmax | (10, 1, 1) |

Figure 1: Architecture of the FashionNet

The input image dimensions are (3, 28, 28). Assume there is no bias in any layers. You do not need to consider the activation layers either. The data type of all the numbers is **float**.

1. Write down the pseudo code for computing the first convolutional layer and the first fully connected layer. The input image is `img[3][28][28]`, the output feature map of the first convolutional layer is `conv1[3][24][24]`, and the convolutional filter is `conv_weight[3][3][5][5]`. The output from the second max-pooling layer is `pool2[3][6][6]`, which needs to be transformed into a 1D array as the input to the first fully-connected layer. The weights of the first fully-connected layer is `fc_weight[100][108]`, and the output of the first fully-connected layer is `fc1[100]`. [**5 pts**]

   Assume all the variables have been declared and initialized. The first convolutional layer is:

```
for (row=0; row<24; row++) {
  for (col=0; col<24; col++) {
    for (to=0; to<3; to++) {
      for (ti=0; ti<3; ti++) {
        for (ki=0; ki<5; ki++) {
          for (kj=0; kj<5; kj++) {
            conv1[to][row][col] +=
                conv_weight[to][ti][ki][kj] *
                img[ti][row+ki][col+kj];
}}}}}}
```

The first fully connected layer is:

```
for (i=0; i<3; i++) {
  for (j=0; j<6; j++) {
    for (k=0; k<6; k++) {
      pool2_1d[i*6*6+j*6+k] = pool2[i][j][k];
}}}
for (m=0; m<100; m++) {
  for (n=0; n<108; n++) {
    fc1[m] += fc_weight[m][n] * pool2_1d[n];
}}
```

2. Calculate the total numbers of floating-point operations (FLOP) in both convolutional layers (2 convolutional layers) and all fully-connected layers (3 fully-connected layers) respectively. What is the total number of floating-point operations in this whole CNN (ignore the max pooling operations)? Compare the percentage of floating-point operations in convolutional layers and fully-connected layers in the whole CNN. [**4 pts**]

   There are $2*5*5*3*3*24*24+2*3*3*3*3*12*12 = 282528$ FLOPs in convolutional layers and $2*108*100+2*100*50+2*50*10 = 32600$ FLOPs in fully connected layers. There are altogether $282528+32600 = 315128$ FLOPs in this CNN. The percentage of FLOP of convolutional layers and fully connected layers in the total FLOP of the whole CNN are 89.65% and 10.35% respectively. Convolutional layers have much more FLOPs.

3. What's the total size of parameters in MB (megabytes) in the convolutional layers? How about the parameter size in all the fully-connected layers? What's the total size of parameters in this whole CNN? Compare the percentages of parameter size for those two parts of this CNN. [**4 pts**]

   The convolutional layers have $(3*3*5*5+3*3*3*3)*4 = 1224$ Bytes $= 0.00012$ MB parameters, and the fully connected layers have $(108*100+100*50+50*10)*4 =$

65200 Bytes = 0.0622 MB parameters. The whole CNN has 66424 Bytes = 0.0633 MB parameters. The convolutional layers and the fully connected layers contain 1.84% and 98.16% of the CNN parameters. The convolutional layers have much less parameters compared to the fully connected layers.

4. Calculate the computation-to-memory ratio (FLOP/MB) for the convolutional layers and fully-connected layers respectively. Compare these two ratios. **[4 pts]**

   The ratios of computation to parameters (FLOPs/MB) for the convolutional part and the fully connected part are 242036013.176 and 347381.006 respectively. The convolutional layers have much higher computation to parameters ratio.

5. Based on the answers above, compare convolutional layers and fully-connected layers in terms of computation and memory. **[4 pts]**

   The convolutional layers have much more floating-point operations than the fully connected layers while the parameter sizes of convolutional layers are much smaller. Both convolutional layers and fully connected layers are memory-bound, but we can see that convolutional layers have much higher computation-to-memory ratio.

## Question 3: Batch Normalization (15 pts)

Batch normalization layers are used in most of today's state-of-the-art convolutional neural networks. Here is the paper (https://arxiv.org/pdf/1502.03167.pdf) which introduces the batch normalization into the convolutional neural networks.

1. Write down the formulation of batch normalization layer. Describe the computation of batch normalization layer. **[5 pts]**

   Batch normalization layer normalizes the inputs of each layer, so that internal covariate shift problem can be avoided. First, the mean and variance of the layers input (among samples in a minibatch) are calculated:

   Batch mean:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{1}$$

   Batch variance:

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2}$$

   where $m$ is minibatch size and $x_i$'s are samples in the minibatch.

Then, normalize the layer inputs using the batch statistics:

$$\overline{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{3}$$

Here $\epsilon$ is a small floating-point number added to variance to avoid dividing by zero.

2. Explore the batch normalization API in TensorFlow. Consider the CNN we build in Question 3. Please write down the TensorFlow model (with Keras) of the first convolutional layer followed by a batch normalization layer (you don't need to write down the whole model for the whole CNN).[**5 pts**]

```
tf.keras.layers.Conv2D(filters=3, kernel_size=(5,5),
    padding='valid', activation='relu', input_shape=(28,28,1))

tf.keras.layers.BatchNormalization()
```

3. During the inference, the mini-batch mean and variance are fixed and independent of the input data. Consider stacking a convolutional layer and a batch normalization layer. Can you merge the batch normalization layer into the convolutional layer?[**5 pts**]

During the inference, the mini-batch mean and variance are fixed and independent of the input data. Let mean be $u_B$ and var be $\sqrt{\sigma_B^2 + \epsilon}$. This can be merged into the loops of convolution:

```
for (row=0; row<24; row++) {
  for (col=0; col<24; col++) {
    for (to=0; to<3; to++) {
      for (ti=0; ti<3; ti++) {
        for (ki=0; ki<5; ki++) {
          for (kj=0; kj<5; kj++) {
            conv1[to][row][col] +=
                conv_weight[to][ti][ki][kj]*
                img[ti][row+ki][col+kj];
      }}}
      conv1[to][row][col] =
          (conv1[to][row][col] - mean[to]) / var[to];
}}}
```

Based on the code, during inference, the mean and the variance of the batch normalization layer can be "absorbed" into the weights and the bias of the convolutional layer.

# Question 4: GPU and CUDA (20 pts)

1. List at least three significant difference between GPU and CPU [**4 pts**]:

   (a) Extensive use of fine-grained data parallelism

   (b) Highly threaded programming model

   (c) Strong scalability

   (d) Intensive floating-pint computation

   (e) Support of high throughput computations

   (f) Relatively high latency in cycles per operation

2. *Warp* is the basic unit in GPU, and a warp is a set of parallel threads that execute the same instruction together in a SIMT architecture. During execution, Streaming Multiprocessor (SMs) does not touch individual threads but instead takes warps and execute them in lock-step style.

   When threads in the same warp do "different things", we call it *control divergence*. As a good CUDA developer, you should try your best to minimize the control divergence. For this part, all warps have 32 threads linearly in the order of dimensions x >> y >> z.

   (a) Consider the following code:

```
int thread_x = threadIdx.x;
int thread_y = threadIdx.y;
for (int i = 0; i < 16; i ++) {
    int pos_x = thread_x*16 + i;
    for (int j = 0; j < 32; j ++) {
        int pos_y = thread_y*32 + j;
        if (pos_x < 400 && pos_y < 384) {
            output[pos_x][pos_y] = 255 - input[pos_x][pos_y];
        }
    }
}
```

   To revert the color of a grayscale image with a size of (400, 384) using GPU, you instantiate a kernel with a blockDim of (64, 16). Assume each thread processes a sub-part of (16, 32), calculate the number of threads and warps that are in control divergence. How many warps are inactive? Justify you answer [**3 pts**].

   Threads do not have control divergence. [This one does not contribute to your grade since "control divergence in thread" is not a formal term]

   12 warps are in control divergence.

$(12 * 1 + 4 * 2) = 20$ warps are inactive (If students do 32-12, they need to explain.)

(b) Now you are given the freedom to edit gridDim and blockDim. Please give a plan that can avoid any control divergence given the same task as in Part(a). You will need to give the configuration and a short code snippet on how to index the given grayscale image in your solution as in Part(a) **[3 pts]**.

One possible answer(others may also be acceptable):

Declare gridDim$<<<$ $400, 1, 1$ $>>>$ blockDim$<<<$ $384, 1, 1$ $>>>$

Index method: input [blockIdx.x][threadIdx.x]

3. Calculate the total effective latency for the memory access of a regular matrix multiplication of two 4 by 4 matrices. Then, calculate the effective latency for a tiled matrix multiplication of two 4 by 4 matrices with TILE_WIDTH=2. Both latencies are measured in cycles. **[5 pts]**

   Please refer to lecture 10, slide 25 for the steps of calculating the latencies. For latencies in cycles of each read/write operation, refer to slide 3.

   Memory access policy:

   (a) **No** concurrent access to the same memory location is allowed, such as global memory.

   (b) Concurrent access to **different** shared memory is allowed. No concurrent access to same shared memory is allowed.

   (c) When copying from the global memory to the shared memory, one action for copying data per tile pays the cost of only 1 read/write operation benefited from Burst Memory Access.

   *The grading for this question would be lenient*

   *Please note that we don't take write back to result matrix into account here because it's constant for both methods. If you do, your answers will also be marked correct*

   **No tiled**: $16(threads) * 2(row + col) * (500 + 1$[register write,no points will be deducted if you miss this, same for following]$) = 16032$ cycles (if burst reading the whole row or column).

   or $16(threads) * 8 * (500 + 1) = 64128$cycles (if each thread need to read 4 elements in row and 4 elements in col). [Both answers are accepted]

   **Tiled**: $4 * 2 * (500 + 5)$[Copy from global to shared] $+4 * (5 + 1)$[access from shared(reusing)]$= 4064$ cycles.

   or $4 * 2 * (500 + 5) + 4 * 4$[Extra 4 here if you think four elements in a subTile share the same memory location]$*(5 + 1)$ [Both answers are accepted]

4. Given a system with one CPU and one GPU, you are asked to deploy a benchmark program to calibrate the performance of the GPU.

   (a) The manual of the single-core CPU tells you that each multiply-add operation requires 10 cycles to complete. Both GPU and CPU are running at the same clock frequency of 2GHz.

   (b) A part of the manual for the GPU is missing, but you are told that each multiply-add operation requires 250 cycles to complete. Each warp has 32 threads, and each Streaming Multiprocessors (SMs) takes up to 1536 threads and spans no more than 8 blocks and 32 warps.

   (c) The benchmark program has the following characteristic:

   - Total multiply-add Operations: 5,120,000
   - The ratio of parallel and sequential code is 3 (75% – 25%)
   - Launch with a blockDim of (64,2,1) and a gridDim set for maximum possible concurrency
   - All data on GPU has been preloaded

   You observe a Speedup of around **3.94225** against running on the same single-core CPU only. What is the number of Streaming Multiprocessor (SM) on this GPU? [**4 pts**]

   Given parallel/sequential = 3, we have portion of 0.75 could be parallel. Next, calculate the maximum concurrency, the limiting factor will be 32 warps or 8 blocks, thus 1024 threads/SM.

   Now, we can calculate the speedup for the parallel part using,

   $Speedup_{parallel}$ = (10 cycles / 250 cycles) * 1024 * (number_of_SM) = 0.75 / (1/3.94225 - 0.25) = 204.792

   Solve the previous equation and we will find $number\_of\_SM = 4.9998 \approx 5$

# Question 5: Processor Performance and Parallelism (20 pts)

You are asked to run an image recognition application on an edge device, and you need to decide what kind of processing device to use. Your initial test device is a single core processor, running at 500MHz and initial operating voltage is 1V. Your machine learning application has 100K instructions.

**PART A:**

If you want to speed up your application by a factor of 10X, what percentage of your code should be parallelizable? [**2 pts**]

**PART B:**

To speed up your application, you can increase the core frequency, but this will increase the power consumption. Therefore, we will explore using multiple cores as well. Consider four scenarios:

1. Single Core.
2. Four cores, and 10% of code is parallel.
3. Four cores, and 50% of code is parallel.
4. Four cores, and 90% of code is parallel.

You are given the following data:

1. Static power per core is 1W
2. Each core has an average IPC (instructions per cycle) of 1.8
3. When you scale frequency, you must increase Vdd (voltage) as well. For every 500MHz above base frequency, you must increase Vdd by 0.15V. You may use the following formula: New Vdd = Base Vdd + (0.15*(RoundDown((New Freq - Base Freq)/Base Freq))
4. You may assume an activity factor (alpha) of 20
5. When executing the serial (non-parallel) portion of code, only one core will consume dynamic power. All cores will consume static power.
6. For the fraction of code that runs in parallel, each core will run the same number of instructions. You may calculate the number of instructions in the parallel section based on the total number of instructions and the amount of parallelism available.

We will perform our analysis with the help of three metrics: Total latency (seconds), Peak Power (Watts), and Performance/Avg Power (MIPS/W), where MIPS = Million Instructions Per Second.

**Please report:**

1. Create three plots for each metric, against increasing frequency. Limit the max frequency to 4.2GHz. [**8 pts**]
2. What is the average power, peak power, and latency in each scenario, at 4.2GHz? [**2 pts**]
3. At what frequency can the multi-core achieve similar Perf/W as a single core? What would happen if the static power per core was 2W? [**2 pts**]

4. If you wanted to use multi-core but limit peak power to 50W, how much parallelism would you need? [**2 pts**]

5. If you have a 50W average power budget, and code that is 90% parallel, should you choose a processor with 2-cores instead of 4-cores? What if your code is only 10% parallel? [**2 pts**]

- Part A

  Use Amdahl's law. You need to at least 90% of code to be parallel for 10X.

- PART B:

  1. You can use excel or python or matlab etc. to make the plots. The key is to get the formulae correctly.

     First, you are given an equation for VDD vs. frequency. Typically, for a given voltage, there is an upper bound on the max clock speed. In this question, we have given you a bound.

     Your table of Voltage vs. Frequency should look like this: (You may use more divisions between steps. 150MHz per step for example.)

| VDD | Freq (MHz) | Cycle Period(s) |
|---|---|---|
| 1 | 500 | 0.000000002 |
| 1.15 | 1000 | 0.000000001 |
| 1.3 | 1500 | 6.66667E-10 |
| 1.45 | 2000 | 5E-10 |
| 1.6 | 2500 | 4E-10 |
| 1.75 | 3000 | 3.33333E-10 |
| 1.9 | 3500 | 2.85714E-10 |
| 2.05 | 4000 | 2.5E-10 |
| 2.2 | 4500 | 2.22222E-10 |

     **To compute latency and performance of the application:**

     Latency = (Num Instructions/IPC) * Clock Period

     If running in parallel, then:

     Num instruction in serial portion = (1-parallelism)*(Total num instructions)

     Num instruction in parallel portion = (parallelism)*(Total num instructions)

     We assume perfect parallelism, so we have:

     Serial portion Latency = (Num Serial Instructions/IPC) * Clock Period

     Parallel portion Latency = ((Num Parallel Instructions/IPC) * Clock Period)/Num cores

Total Latency = Serial Latency + Parallel Latency

MIPS = (Total Instructions)/(Total Latency)

**To compute power of the application:**

Power = Static Power + Dynamic Power

Energy = Power * Time

Static Power = Total Num Cores * Static power per core

Static power is constant and does not change in this setup.

Dynamic Power = Num Cores * Activity Factor * Capacitance * Voltage * Voltage * Frequency

Since num cores, activity factor, and capacitance are constant, the dynamic power will depend only on the voltage (which in turn depends on frequency). When running serial code, num cores = 1.

Your average power will depend on how long all 4 cores are active:

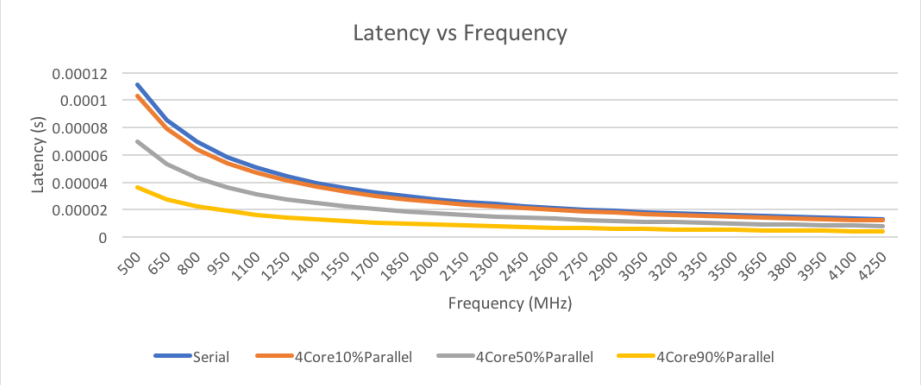
Total Energy = (Serial Latency * Power When Serial) + (Parallel Latency * Power When Parallel)

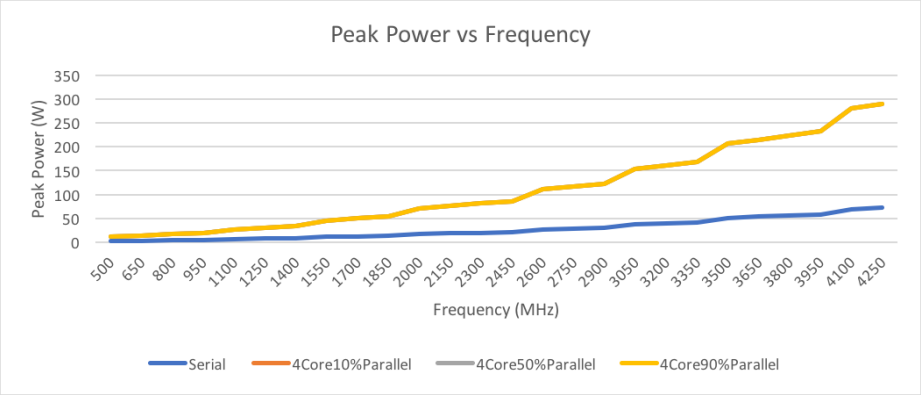
NOTE: Both total serial and total parallel power, include the static power of all cores.

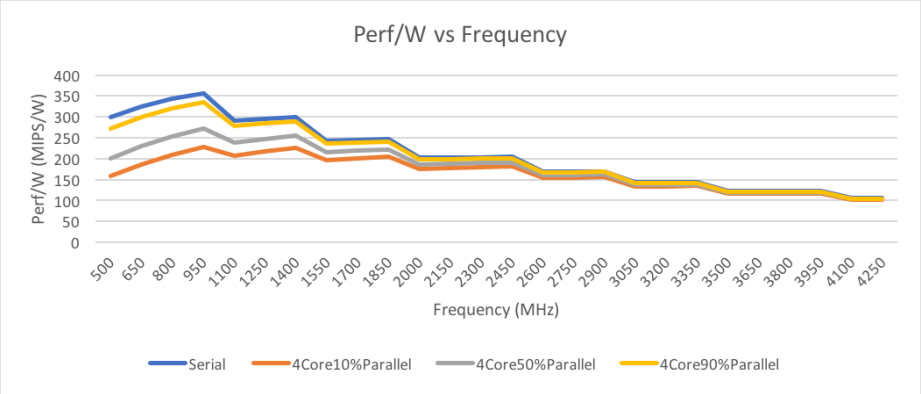
Avg Power = Total Energy / Total Latency

However, Peak Power will always be the power in the parallel region, since this is the time during which the max power is consumed.

**You can now create the tables needed to make the plots:**






2. At 4.25GHZ:

| | Latency(s) | Peak Power (W) | Perf/W (MIPS/W) |
|---|---|---|---|
| Serial | 1.30719E-05 | 72.4425 | 105.6009939 |
| 4Core 10% Parallel | 1.20915E-05 | 289.77 | 101.8065675 |
| 4Core 50% Parallel | 8.16993E-06 | 289.77 | 103.4587686 |
| 4Core 90% Parallel | 4.24837E-06 | 289.77 | 105.165481 |

3. At around 2GHz Perf/W of 90% parallel multi-core is similar to a single core, but for 10% parallelism, it is at 3.5GHz. If static power increases to 2W, then 10% parallel will never match up, and 90% parallel matches up at around 3.1GHz.
4. Peak power is not determined by % parallelism.
5. Find out what frequency will result in 50W avg power for 2 core vs 4 core. Then find out what the latency of the application is. For 4 cores with 90% parallelism, at 1.995GHz, the avg power is 46W (Due to the VDD curve, it crosses 50W after this point). For 2 cores, the limit is 2.515GHz.

At 1.995GHz, the 4 core machine's latency is 9.0504E-06s, while the latency of the 2 core machine at 2.515GHz is 1.21493E-05.

**Clearly, in this case, the 4core machine is superior.**

However, if parallelism is only 10%:

The max frequency of the 4 core machine would be 3.460GHz, and the 2 core machine would be 3.460GHz. The application latency on the 4 core and the 2 core at these frequencies would be 1.48523E-05s and 1.52537E-05s respectively. So there isn't a major performance difference. This is expected, since the latency is dominated by single-thread performance. Thus, we can look at peak power, and we see that the 4core consumes 173.54W, and the 2 core consumes 86.77W.

**Thus, the 2core is better choice.**