ECE 498 ICC Spring 2020 Jin Yucheng (yucheng9)

Homework 2 Problem 1

Solution: (1) One advantage is that using IP-based hardware design, we can make the computation process faster (by parallel computation, for example). One disadvantage is that using IP-based hardware design, we may sacrifice some flexibility.

(2) For (a), we need 128 compute engine IPs: $\frac{2n}{0.25n} \times \frac{4n}{0.25n} = 128$.

For (b), we need 32 compute engine IPs: $\frac{2n}{0.5n} \times \frac{4n}{0.5n} = 32$.

(3) Since k IPs are instantiated, and each IP is of version (b) from part (2), we have,

- Area $= k \cdot A + 32 \times \frac{1}{4}A = (k+8)A$
- Latency $= (T + \frac{k^2}{32}T) \cdot \frac{32}{k} = (\frac{32}{k} + k)T$
- Area $\times$ Latency $= (k+8)A \cdot (\frac{32}{k} + k)T = (k^2 + 8k + 32 + \frac{256}{k})AT$

The objective function, $f = (k^2+8k+32+\frac{256}{k})$ is minimized at $k = 4$. Because $\frac{df}{dk} = 2k+8-\frac{256}{k^2}$ equals 0 at $k = 4$, and $\frac{df}{dk} > 0$ for $k > 4$, f is monotonously increasing; $\frac{df}{dk} < 0$ for $k < 4$, f is monotonously decreasing, therefore, f is minimized at $k = 4$.

In conclusion, 4 IPs should be instantiated to minimize the product of total latency and total area.

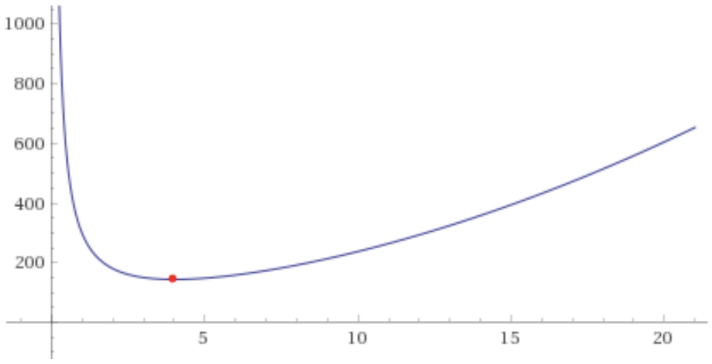
Input interpretation:

minimize	function	$k^2 + 8k + 32 + \frac{256}{k}$
	domain	$0 < k < 20$

Global minimum:

$$\min\left\{k^2 + 8k + 32 + \frac{256}{k} \,\middle|\, 0 < k < 20\right\} = 144 \text{ at } k = 4$$

Plot:



■

ECE 498 ICC Spring 2020 Jin Yucheng (yucheng9)
Homework 2 Problem 2

Solution: (1)

Conv_Layer_One(img[3][28][28], conv_weight[3][3][5][5]):

```
1 : for i ← 1 to 3:
2 :     for j ← 1 to 24:
3 :         for k ← 1 to 24:
4 :             conv1[i][j][k] = 0
5 :                 for x ← 1 to 3:
6 :                     for y ← 1 to 5:
7 :                         for z ← 1 to 5:
8 :                             conv1[i][j][k] += img[x][y + j][z + k] *
9 :                                               conv_weight[i][x][y][z]
10 :                        end
11 :                    end
12 :                end
13 :        end
14 :    end
15 : end
16 : # ReLu as activation function
17 : for i ← 1 to 3:
18 :     for j ← 1 to 24:
19 :         for k ← 1 to 24:
20 :             if conv1[i][j][k] < 0: conv1[i][j][k] = 0
21 :         end
22 :     end
23 : end
```

Output: conv1[3][24][24]

Fully_Connected_Layer_One(pool2[3][6][6], fc_weight[100][108]):

```
1 : flatten = zeros(108)
2 : for i ← 1 to 3:
3 :     for j ← 1 to 6:
4 :         for k ← 1 to 6:
5 :             flatten[6 * 6 * i + 6 * j + k] = pool2[i][j][k]
```

6 : **end**
7 : **end**
8 : **end**
9 : **for** $i \leftarrow 1$ to 100:
10 : fc1[i] = 0
11 : **for** $j \leftarrow 1$ to 108:
12 : fc[i] += fc_weight[i][j] * flatten[j]
13 : **end**
14 : **end**
15 : # ReLu as activation function
16 :**for** $i \leftarrow 1$ to 100:
17 : **if** fc1[i] < 0: fc1[i] = 0
18 : **end**
19 :**end**

Output: fc1[100]

(2) **# of floating-point operations in a convolutional layer** = # of input channels × # of output channels × output width × output height × filter size ×2 (in this case # of floating-point additions = # of floating-point multiplications)

# of floating-point operations in a fully-connected layer = input size × output size ×2 (in this case # of floating-point additions = # of floating-point multiplications)

- For convolutional layers: $Flop_{conv} = 3\times24\times24\times3\times5\times5\times2 + 3\times12\times12\times3\times3\times3\times2 = 282,528$
- For fully-connected layers: $Flop_{fc} = 108\times100\times2+100\times50\times2+50\times10\times2 = 32,600$
- Total # of floating-point operations $= 282,528 + 32,600 = 315,128$
- % of floating-point operations in convolutional layers: $\frac{282,528}{315,128}\times100\% = 89.655\%$
- % of floating-point operations in fully-connected layers: $\frac{32,600}{315,128}\times100\% = 10.345\%$

Therefore, convolutional layers take the majority of floating-point operations.

(3) Assume each parameter takes 32-bit. Assume there is no bias in any layers.

# of parameters in a convolutional layer = # of input channels × # of output channels × filter size

# of parameters in a fully-connected layer = input size × output size

- For convolutional layers: $Para_{conv} = 3\times3\times5\times5\times4\times\frac{1}{1024^2} + 3\times3\times3\times3\times4\times\frac{1}{1024^2} = 0.0011673$ [MB]
- For fully-connected layers: $Para_{fc} = 108\times100\times4\times\frac{1}{1024^2}+100\times50\times4\times\frac{1}{1024^2}+50\times10\times4\times\frac{1}{1024^2} = 0.0621796$ [MB]

- Total size of parameters $= 0.0011673 + 0.0621796 = 0.0633469$ [MB]
- % of parameters in convolutional layers: $\frac{0.0011673}{0.0633469} \times 100\% = 1.84\%$
- % of parameters in fully-connected layers: $\frac{0.0621796}{0.0633469} \times 100\% = 98.16\%$

Therefore, fully-connected layers take the majority of parameters.

(4)

- For convolutional layers: $Ratio = \frac{282,528}{0.0011673} \times 100\% = 242,035,466$
- For fully-connected layers: $Ratio = \frac{32,600}{0.0621796} \times 100\% = 524,288$

Because convolutional layers have a much larger computation-to-memory ratio, so each parameter in convolutional layers involves in more computations.

(5) Convolutional layers take the majority of floating-point operations in terms of computation, fully-connected layers take the majority of parameters in terms of memory, also, because convolutional layers have a much larger computation-to-memory ratio, so each parameter in convolutional layers performs more computations than each parameter in fully-connected layers. In conclusion, for a convolutional layer and a fully-connected layer with similar input sizes and output sizes, the convolutional layer has smaller number of parameters, but each parameter performs more computations, which results in higher amount of computation. ■

ECE 498 ICC Spring 2020 Jin Yucheng (yucheng9)

Homework 2 Problem 3

Solution: (1) We have,

$$\mu_X = \frac{1}{m}\Sigma_{i=1}^{m} X[i]$$
$$\sigma_X^2 = \frac{1}{m}\Sigma_{i=1}^{m}(X[i]-\mu_X)^2$$
$$Y[i] = \gamma\frac{X[i]-\mu_X}{\sqrt{\sigma_X^2+\epsilon}} + \beta$$

where $X[m] = \{x_{1...m}\}$ is a mini-batch, β, γ are parameters to be learned, ϵ is a sufficiently small number to prevent division by zero, and $Y[m] = \{y_{1...m}\}$ is the output after applying batch normalization.

Batch normalization is a training technique that is used to normalize the output of the previous layer for each batch. After batch normalization, the mean of output will be close to 0, and the standard deviation will be close to 1.

(2) The Keras API is as follows (see https://keras.io/layers/normalization/),

BatchNormalization [source]

```
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True,
```

Batch normalization layer (Ioffe and Szegedy, 2014).

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Arguments

- **axis**: Integer, the axis that should be normalized (typically the features axis). For instance, after a `Conv2D` layer with `data_format="channels_first"`, set `axis=1` in `BatchNormalization`.
- **momentum**: Momentum for the moving mean and the moving variance.
- **epsilon**: Small float added to variance to avoid dividing by zero.
- **center**: If True, add offset of `beta` to normalized tensor. If False, `beta` is ignored.
- **scale**: If True, multiply by `gamma`. If False, `gamma` is not used. When the next layer is linear (also e.g. `nn.relu`), this can be disabled since the scaling will be done by the next layer.
- **beta_initializer**: Initializer for the beta weight.
- **gamma_initializer**: Initializer for the gamma weight.
- **moving_mean_initializer**: Initializer for the moving mean.
- **moving_variance_initializer**: Initializer for the moving variance.
- **beta_regularizer**: Optional regularizer for the beta weight.
- **gamma_regularizer**: Optional regularizer for the gamma weight.
- **beta_constraint**: Optional constraint for the beta weight.
- **gamma_constraint**: Optional constraint for the gamma weight.

Input shape

Arbitrary. Use the keyword argument `input_shape` (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output shape

Same shape as input.

Therefore, my CNN is (batch normalization at line 8),

```
1  # build CNN model
2  CNN_model = keras.Sequential([
3      # convolution layer; 5x5 filter, stride = 1, no padding; ReLU; output = (3, 24, 24)
4      keras.layers.Conv2D(filters=20, kernel_size=(5,5), strides=1, padding='valid', activation='relu',
5                          input_shape=(28, 28, 1)),
6
7      # batch normalization
8      keras.layers.BatchNormalization()
9
10     # max pooling layer; strides = 2; output = (3, 12, 12)
11     keras.layers.MaxPooling2D(strides=2),
12
13     # convolution layer; 3x3 filter, stride = 1, with padding; ReLU; output = (3, 12, 12)
14     keras.layers.Conv2D(filters=20, kernel_size=(3,3), strides=1, padding='same', activation='relu',
15                         input_shape=(12, 12, 3)),
16
17     # max pooling layer; strides = 2; output = (3, 6, 6)
18     keras.layers.MaxPooling2D(strides=2),
19
20     # flatten
21     keras.layers.Flatten(),
22
23     # fully connected layer; input size = 108, output size = 100; ReLU; output = (100, 1, 1)
24     keras.layers.Dense(units=100, activation='relu'),
25
26     # fully connected layer; input size = 100, output size = 75; ReLU; output = (75, 1, 1)
27     keras.layers.Dense(units=75, activation='relu'),
28
29     # fully connected layer; input size = 75, output size = 50; ReLU; output = (50, 1, 1)
30     keras.layers.Dense(units=50, activation='relu'),
31
32     # fully connected layer; input size = 50, output size = 25; ReLU; output = (25, 1, 1)
33     keras.layers.Dense(units=25, activation='relu'),
34
35     # fully connected layer; input size = 25, output size = 20; ReLU; output = (20, 1, 1)
36     keras.layers.Dense(units=20, activation='relu'),
37
38     # fully connected layer; input size = 20, output size = 15; ReLU; output = (15, 1, 1)
39     keras.layers.Dense(units=15, activation='relu'),
40
41     # fully connected layer; input size = 15, output size = 10; Softmax; output = (10, 1, 1)
42     keras.layers.Dense(units=10, activation='softmax'),
43 ])
```

(3) Yes, we can merge batch normalization into the convolutional layer. What we should do is just to modify weights and biases according to the following pseudocode,

Batch_Normalization_Conv_Layer(image[C_i][H_i][W_i], conv_weight[C_o][C_i][H_o][W_o], conv_bias[C_o], $\mu_X, \sigma_X^2, \beta, \gamma, \epsilon$):

```
: for i ← 1 to C_o:
:     for j ← 1 to C_i:
:         for k ← 1 to H_o:
:             for q ← 1 to W_o:
:                 conv_weight[i][j][k][q] *= γ[m] / sqrt(σ_X^2[m] + ε)
:             end
:         end
:     end
: end
: for i ← 1 to C_o:
```

11 : conv_bias[i] = conv_bias[i] * $\frac{\gamma[m]}{\sqrt{\sigma_X^2[m]+\epsilon}}$ + $\beta[m]$ - $\frac{\gamma[m]*\mu_X[m]}{\sqrt{\sigma_X^2[m]+\epsilon}}$

12 : **end**

13 : **for** $i \leftarrow 1$ to C_o:

14 : **for** $j \leftarrow 1$ to $H_i - H_o + 1$:

15 : **for** $k \leftarrow 1$ to $W_i - W_o + 1$:

16 : **for** $x \leftarrow 1$ to C_i:

17 : **for** $y \leftarrow 1$ to H_o:

18 : **for** $z \leftarrow 1$ to W_o:

19 : conv1[i][j][k] += image[x][j+y][k+z] * conv_weight[i][x][y][z]

20 : **end**

21 : **end**

22 : **end**

23 : **end**

24 : **end**

25 : **end**

26 :# Apply activation function ...

27 :# Activation function ends...

Output: conv1$[C_o][H_i - H_o + 1][W_i - W_o + 1]$ ■

ECE 498 ICC Spring 2020 Jin Yucheng (yucheng9)

Homework 2 Problem 4

Solution: (1) See the following table,

	CPU	GPU
Design Orientation	Latency-oriented	Throughput-oriented
Clock Frequency	High clock frequency	Moderate clock frequency
Control	Sophisticated control with branch prediction for reduced branch latency and data forwarding for reduced data latency	Simple control without branch prediction and data forwarding
Caches	Large caches to convert long latency memory accesses to short latency cache accesses	Small caches to boost memory throughput
ALUs	Powerful ALUs: reduced operation latency	Energy efficient ALUs: many, long latency but heavily pipelined for high throughput
Always Used in	Sequential parts where latency matters (10x faster than GPUs for sequential code)	Parallel parts where throughput matters (10x faster than CPUs for parallel code)

(2-a) 12 warps have control divergence, 20 warps are inactive, 300 threads are active, and 724 threads are inactive.

Proof: For the kernel with block dimension of $(64, 16)$, we need $64 \times 16 \times \frac{1}{32} = 32$ warps in total. Because each thread is able to process a subpart of the image of size $(16, 32)$, while the image is of size $(400, 384)$, $\frac{400}{16} = 25$, $\frac{384}{32} = 12$, so the threads active are of size $(25, 12)$, and totally 300 threads are active, 724 threads are inactive. Therefore, 12 warps with blockIdx.y $<$ 12 (blockIdx.y starts with 0) are active, and the rest 20 warps are inactive.

(2-b) Let's have gridDim of $(1, 1, 1)$ and blockDim of $(16, 12, 1)$, and each thread is able to process a subpart of the image of size $(25, 32)$ ($25 \times 16 = 400, 12 \times 32 = 384$) such that there is no control divergence.

```
1   int thread_x = threadIdx.x;
2   int thread_y = threadIdx.y;
3   for (int i = 0 ; i < 25 ; i ++) {
4     int pos_x = thread_x * 25 + i;
5     for (int j = 0 ; j < 32 ; j ++) {
6       int pos_y = thread_y * 32 + j;
7       if (pos_x < 400 && pos_y < 384) {
8       output[pos_x][pos_y] = 255 - input[pos_x][pos_y];
9       }
10    }
11  }
```

(3)

- For regular matrix multiplication of two 4 by 4 matrices: (1) We need to access the global memory for $16 \times 4 = 64$ times (because each thread needs to load elements from matrices for 4 times), and 500 cycles are needed to read from/write to the global memory, 1 cycle is needed to read from/write to the registers, there are totally $64 \times 501 = 32,064$ cycles. (2) To write results back to the global memory, it takes $16 \times 500 = 8,000$ cycles. (3) So in general, $32,064 + 8,000 = 40,064$ cycles are needed.

- For tiled matrix multiplication of two 4 by 4 matrices with TILE_WIDTH = 2. (1) We need to access the global memory and load data to the shared memory for $4 \times 2 = 8$ times (because each thread in a block deals with 2 tiles, and 4 blocks access data one by one), and 500 cycles are needed to read from/write to the global memory, 5 cycle is needed to read from/write to the shared memory, there are totally $8 \times 505 = 4,040$ cycles. (2) We need to access the shared memory to load data for $4 \times 6 = 24$ cycles. (3) To write results back to the global memory, it takes $4 \times 500 = 2,000$ cycles. (4) So in general, $4,040 + 24 + 2,000 = 6,064$ cycles are needed.

(4) The answer is 5.

The number of parallel instructions is $5,120,000 \times 0.75 = 3,840,000$, and the number of sequential instructions is $1,280,000$, so for CPU + GPU computation, CPU needs $12,800,000$ cycles, and GPU needs $3,840,000 \times 250 \times 0.75 \times \frac{1}{1024n} = \frac{937,500}{n}$ cycles.

Compared to pure CPU computation, the relationship is:

$3.94225 = \frac{51,200,000}{12,800,000+\frac{937,500}{n}}$

And we can find that $n \approx 5$. ■

ECE 498 ICC Spring 2020 Jin Yucheng (yucheng9)
Homework 2 Problem 5

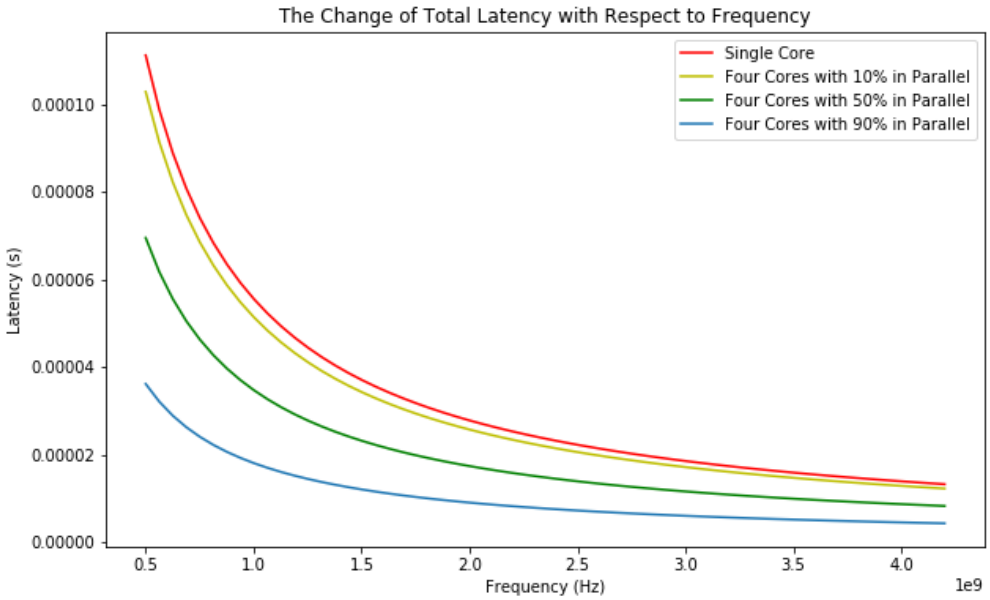
Solution: (A) By Amdahl's Law,

$$Speedup = \frac{T_{new}}{T_{old}} = \frac{1}{(1 - Fraction_{parallel}) + \frac{Fraction_{parallel}}{Speedup_{parallel}}}$$

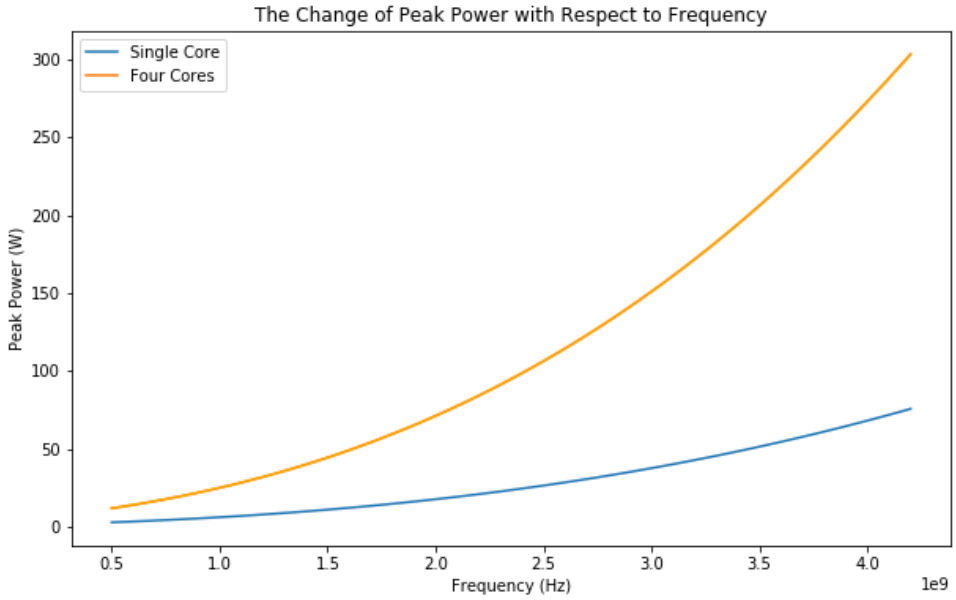
If we want to speed up the application by a factor of 10X, then we have the equation showing the fraction (percentage) of the code to be parallelizable:

$Fraction_{parallel} = \frac{(10X-1)\cdot Speedup_{parallel}}{10X(Speedup_{parallel}-1)}$

(B-1) For total latency,

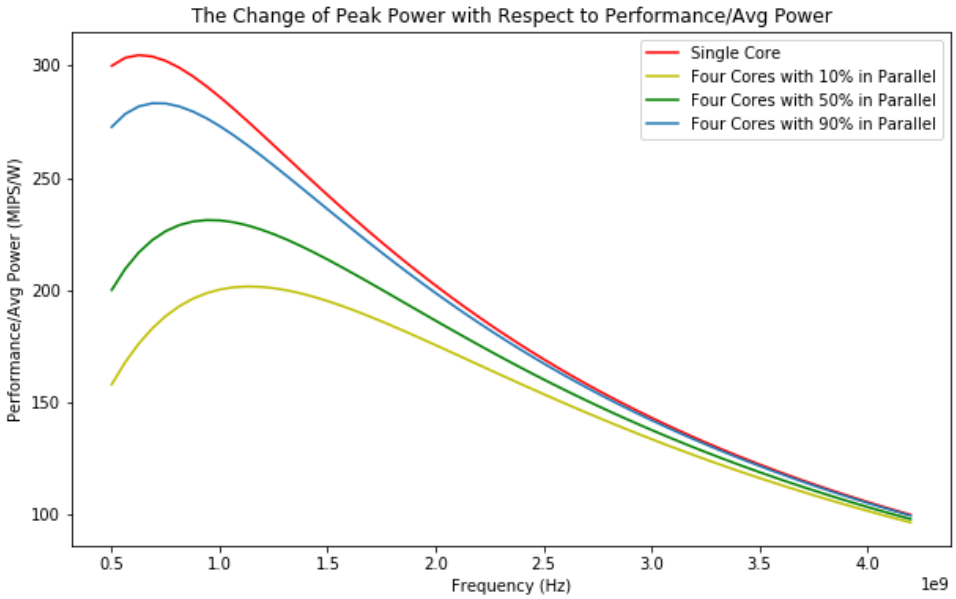


For peak power,



Notice that for four cores with 10%, 50%, and 90% in parallel, they have the same peak power.

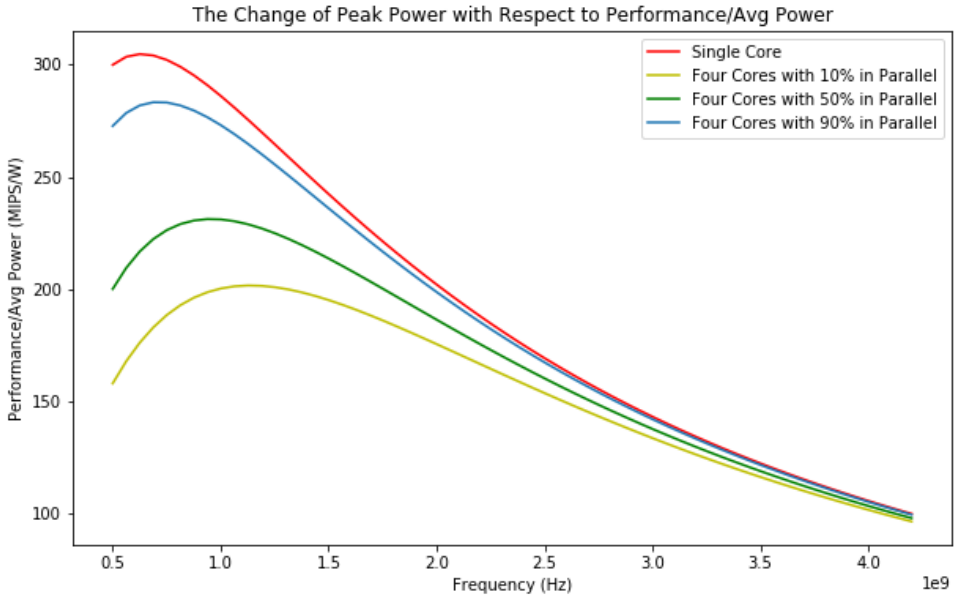
For performance/average power,



(B-2)

	Single core	Four Cores with 10% in Parallel	Four Cores with 50% in Parallel	Four Cores with 90% in Parallel
Average Power (W)	75.80	84.86	123.67	234.14
Peak Power (W)	75.80	303.20	303.20	303.20
Latency (s)	1.32e-5	1.22e-5	8.27e-6	4.30e-6

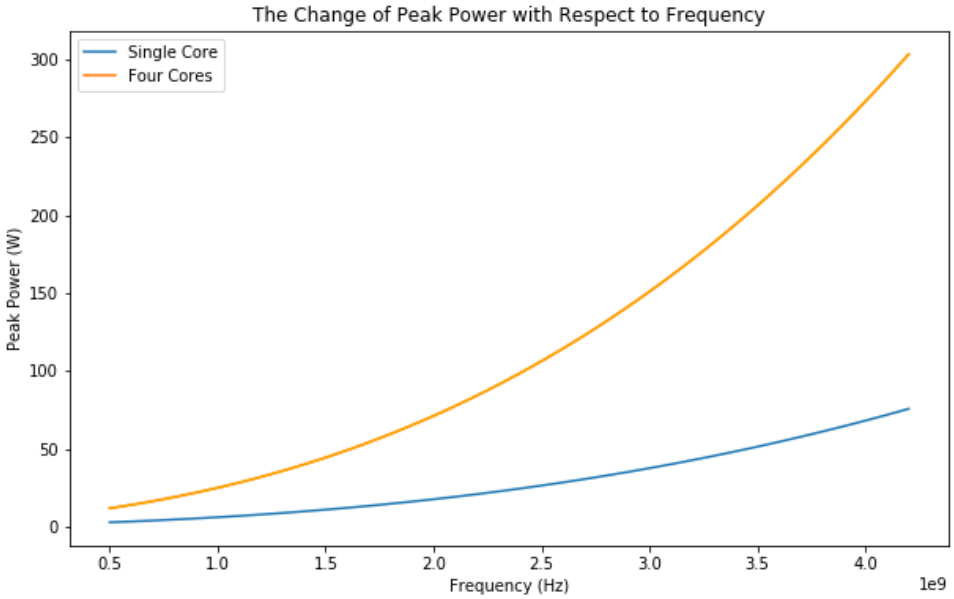
(B-3) Performance per Watt is measured by performance/average power,



From the above plot, we could see at a frequency above 3.5 GHz, the multi-core can achieve similar performance per Watt as the single-core.

If the static power per core is 2W, the multi-core will be less energy-efficient, in order to achieve similar performance per Watt as the single-core, the multi-core should run at a higher frequency than the frequency when static power per core is 1W.

(B-4) The peak power changes as follows,



For the multi-core, parallelism will not influence peak power, so 10%, 50%, 90% are all OK. For shorter latency and higher performance/average power, we should choose 90%, and remember to run the multi-core at a frequency below 1.5 GHz.

(B-5) I will choose the four-core for 90% parallelism, and the two-core for 10% parallelism. Because when parallelism is high, we can exploit the advantage of more cores to reduce latency by choosing the four-core; but when parallelism is low, compared with high power consumption, the advantage of more cores is offset by low energy-efficiency. ■