

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده مهندسی کامپیوتر

درس فناوری های حافظه
دکتر حامد فربه

رضا آدینه پور
۴۰۲۱۳۱۰۵۵

تمرین شبیه سازی چهارم

تدریسار:

مرتضی عادلخانی (madelkhani@aut.ac.ir)
سارا زمانی (sara.zamani73@aut.ac.ir)

سوالات تئوری

۱. به سوالات زیر پاسخ دهید:

۱. PUM چیست و کدام نوع حافظه ها برای آن بیشتر استفاده می شوند؟ توضیح دهید چرا هر نوع حافظه استفاده می شود.

پاسخ: پردازش در حافظه (PUM) یک کانسپت محاسباتی است که در آن برخی از محاسبات ساده مانند جمع و ضرب به جای انتقال داده ها بین CPU و حافظه، مستقیماً در حافظه انجام می شوند. معمولاً از DRAM، SRAM و NVM ها در PUM استفاده می شود. که در ادامه به بررسی مزایا و معایب استفاده از هر کدام می پردازیم.

DRAM ها به دلیل اینکه رایج ترین نوع حافظه فرار با تراکم بالا و هزینه کم به ازای هر بیت هستند، به طور گسترده استفاده می شود. ویژگی های خازنی سلول های DRAM امکان انجام تکنیک های محاسباتی درون حافظه مانند عملیات منطقی و حسابی را فراهم می کند. مزایا: تراکم بالا، ارزان است.

معایب: فرار، نیاز به تازه سازی دوره ای و معمولاً تأخیر بیشتر نسبت به SRAM. اما در مقابل SRAM زمان های تأخیر کمتری و زمان دسترسی سریعتری نسبت به DRAM دارد و در مواردی که سرعت برای ما بسیار مهم است، (مانند Cache)، استفاده می شود. توانایی حفظ حالت بدون نیاز به تازه سازی، آن را برای عملیات های PUM مناسب می سازد.

مزایا: زمان های دسترسی سریع نسبت به DRAM، عدم نیاز به تازه سازی. معایب: تراکم کمتر و هزینه بیشتر به ازای هر بیت نسبت به DRAM.

در مقابل حافظه های فرار، انواع حافظه های غیر فرار مانند Flash، PCM و ReRAM به دلیل نگه داشتن داده بدون برق، برای ذخیره سازی پایدار و محاسبات مناسب هستند. این حافظه ها می توانند برخی عملیات منطقی را درون سلول های حافظه انجام دهند. مزایا: غیر فرار بودن.

معایب: عموماً سرعت نوشتن کندتر و دوام کمتر نسبت به DRAM و SRAM.

۲. نقاط ضعف UPMEM چیست؟

پاسخ: از نقاط ضعف UPMEM ها می توان به موارد زیر اشاره کرد:

(آ) انعطاف پذیری و قابلیت برنامه ریزی محدود:

معماری PIM UPMEM برای انواع خاصی از عملیات (مانند وظایف داده محور مانند جست و جو در پایگاه داده و تحلیل) است. ممکن است به اندازه CPU یا GPU سستی عمومی و همه منظوره نباشد، که کاربرد آن را به بارهای کاری خاص محدود می کند.

(ب) یکپارچه سازی و سازگاری:

یکپارچه سازی ماژول های PIM UPMEM با سیستم های موجود می تواند چالش برانگیز باشد. ممکن است مشکلات سازگاری با معماری های حافظه و پردازنده فعلی به وجود بیاید که نیاز به اصلاحات در کانفیک نرم افزار و سخت افزار دارد.

(ج) مسائل مربوط به کارایی انرژی:

در حالی که PIM هدفش کاهش مصرف انرژی با حداقل کردن حرکت داده ها بین حافظه و CPU است، صرفه جویی واقعی در انرژی می تواند وابسته به بار کاری باشد. برخی عملیات ممکن است همچنان مصرف انرژی قابل توجهی داشته باشند، به خصوص اگر منطق PIM به طور کامل برای آن کاربرد به خصوص بهینه سازی نشده باشد.

(د) توسعه و اشکال زدایی:

همانطور که در کلاس هم بررسی شد، توسعه برنامه ها برای PIM نیاز به مدل های برنامه نویسی و

ابزارهای جدید دارد. دیباگ و پروفایل کردن برنامه های PIM می تواند به دلیل طبیعت توزیع شده و درون حافظه ای محاسبات سخت تر از برنامه نویسی CPU/GPU سنتی باشد.

۳. ساختار Ambit را معرفی کرده و مزایا و معایب آن را توضیح دهید.

پاسخ: Ambit یک معماری PIM است که از بستر DRAM موجود برای انجام عملیات بیتی (مانند AND، OR، NOT) به طور مستقیم درون حافظه استفاده می کند. این معماری از ویژگی های آنالوگ سلول های DRAM و Sense Amplifier برای اجرای این عملیات استفاده می کند. از مزایای آن می توان به موارد زیر اشاره کرد:

(آ) کاهش حرکت داده ها:

با انجام محاسبات مستقیماً درون DRAM به طور قابل توجهی نیاز به حرکت داده بین CPU و حافظه را کاهش می دهد، که منجر به کاهش تاخیر و مصرف انرژی می شود.

(ب) توان محاسباتی بالا:

Ambit می تواند عملیات بیتی را بر روی حجم زیادی از داده ها به طور همزمان انجام دهد، که توان محاسباتی بالایی برای وظایف داده محور مانند جستجوهای پایگاه داده، رمزنگاری و شبکه های عصبی فراهم می کند.

(ج) تغییرات سخت افزاری حداقلی:

Ambit از زیرساخت DRAM موجود با تغییرات حداقلی استفاده می کند، که ادغام آن را با سیستم های فعلی نسبت به معماری های PIM آسان تر می کند.

همچنین از معایب Ambit می توان به موارد زیر اشاره کرد:

(آ) محدودیت در انواع عملیات:

Ambit به طور عمده برای عملیات بیتی طراحی شده است. نمی تواند به طور کارآمد محاسبات پیچیده تر ریاضی یا اعشاری را انجام دهد، که کاربرد آن را به انواع خاصی از عملیات ها محدود می کند.

(ب) پیچیدگی در برنامه نویسی:

برنامه نویسی برای Ambit نیاز به درک مدل عملیاتی خاص و محدودیت های آن دارد. برنامه نویس ها باید الگوریتم های خود را برای استفاده موثر از عملیات بیتی تطبیق دهند، که می تواند پیچیدگی نرم افزاری را افزایش دهد.

(ج) چالش های مقیاس پذیری:

در حالی که Ambit توان محاسباتی بالایی برای عملیات بیتی ارائه می دهد، مقیاس پذیری آن به سیستم های حافظه بزرگ تر یا ادغام آن با واحدهای پردازشی دیگر ممکن است چالش هایی از نظر هماهنگی و مدیریت ایجاد کند.

سوالات شبیه سازی

۲. در این بخش از تکلیف خود، شما با شبیه ساز MNSIM 2.0 برای پیاده سازی یک شتاب دهنده شبکه عصبی سر و کار خواهید داشت. بنابراین، ابتدا باید این شبیه ساز را دانلود کرده و نتایج را طبق درخواست ارائه دهید.

پیکربندی پایه:

۱. پارامترهای شبکه عصبی VGG8 که بر روی مجموعه داده CIFAR-10 آموزش دیده است را دانلود کنید، همانطور که در راهنمای MNSIM 2.0 توضیح داده شده است.

۲. برای هر اجرا، باید VGG8 را به عنوان شبکه عصبی مورد نظر خود انتخاب کنید.

پاسخ: وزن ها را از اینجا دانلود می کنیم.
پس از دانلود به مسیر شبیه ساز می رویم و با دستور

سوال ۱:

همانطور که در کلاس یاد گرفتید، ساختار PIM شامل تعدادی کاشی است و هر کاشی شامل تعدادی PE است. در هر PE، ما مدارهای ضروری و ساختار ضربداری سلول های حافظه را داریم. اگر اندازه ضربداری را کاهش دهیم چه اتفاقی می افتد؟ به عنوان مثال، آیا باید انتظار داشته باشیم که توان، تأخیر و دقت کاهش، افزایش یا تغییر نکند؟ پاسخ خود را به تفصیل توضیح دهید.

پاسخ: کاهش اندازه ی ساختار کراسبار در معماری PIM شامل چندین موازنه می شود و بر پارامترهای مختلفی مانند توان مصرفی، تأخیر و دقت تأثیر می گذارد. در ادامه به بررسی و تأثیر هر کدام می پردازیم:

۱ توان مصرفی

- **کاهش توان مصرفی:** کراسبارهای کوچکتر معمولاً به کاهش توان مصرفی منجر می شوند. این امر به این دلیل است که کراسبار کوچکتر تعداد سلول های حافظه و اتصالات کمتری دارد، که منجر به کاهش کلی ظرفیت خازنی می شود. ظرفیت کمتر به معنی شارژ/دشارژ کمتر در طول عملیات است که به توان مصرفی دینامیک کمتری منجر می شود.
- **ملاحظات توان استاتیک:** با این حال، توان مصرفی استاتیک ممکن است به همان اندازه کاهش نیابد، به ویژه اگر به همان تعداد مدارهای جانبی (مانند Sense amplifier و درایورها) برای کراسبار کوچکتر نیاز باشد. کاهش در توان استاتیک عموماً کمتر از کاهش در توان دینامیک است.

۲ تأخیر

- **کاهش تأخیر:** کراسبارهای کوچکتر می توانند تأخیر را بهبود بخشند. زمان خواندن یا نوشتن داده در یک ساختار کراسبار به طول اتصالات و تعداد سلول های حافظه وابسته است. با کراسبار کوچکتر، سیگنال ها باید مسافت کمتری را طی کنند و سلول های کمتری برای شارژ یا دشارژ وجود دارند، که منجر به عملیات سریعتر می شود.

۳ دقت

- **بهبود بالقوه در دقت:** کراسبارهای کوچکتر می توانند منجر به بهبود دقت شوند. کراسبارهای بزرگتر با مشکلاتی مانند افزایش مقاومت و ظرفیت خازنی در طول اتصالات مواجه می شوند، که می تواند باعث تخریب سیگنال و افزایش حساسیت به نویز شود. با کاهش اندازه ی کراسبار، این اثرات به حداقل می رسند، که منجر به انتقال سیگنال و حسگری قابل اعتمادتر می شود و می تواند دقت را بهبود بخشد.
- **نرخ خطا:** احتمال خطا به دلیل تداخل و سایر اثرات نیز در کراسبارهای کوچکتر کمتر است، که به بهبود دقت در عملیات هایی مانند ضرب ماتریسی و عملیات های برداری که به طور معمول در معماری PIM استفاده می شوند، کمک می کند.

سوال ۲:

اگر بخواهیم PUM را به این شبیه‌ساز اضافه کنیم، کدام قسمت باید تغییر کند؟

پاسخ:

تغییرات مورد نیاز برای اضافه کردن PUM به شبیه‌ساز NVSim به صورت زیر است:

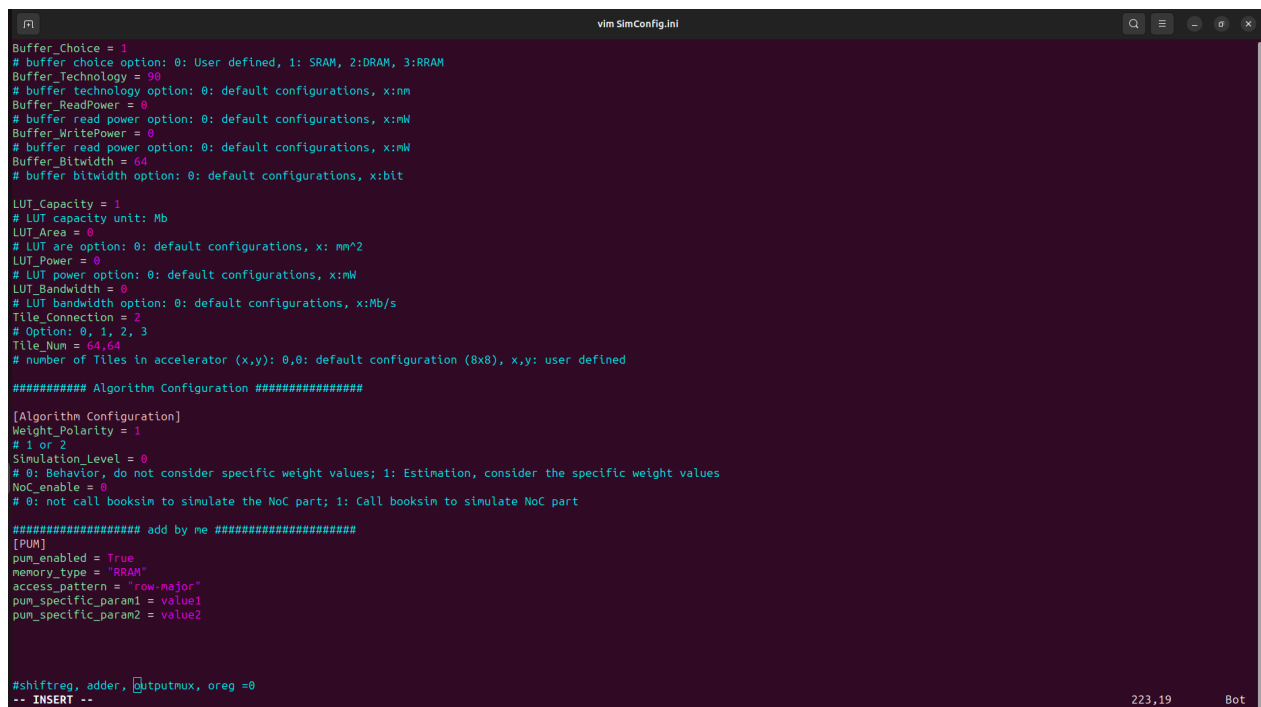
قسمت‌های کلیدی برای تغییر

۱. پیکربندی سخت‌افزار (SimConfig.ini):

- می‌بایست معماری PUM را در فایل پیکربندی سخت‌افزار توصیف کنیم.
- بر اساس معماری مورد نیاز خودمان می‌توانیم بخش‌هایی را به فایل کانفیگ اضافه نموده. این بخش‌ها شامل اضافه نمودن پارامترهای سخت‌افزاری PUM مانند نوع حافظه، الگوهای دسترسی حافظه و ... باشد

برای مثال می‌توان این تنظیمات را در فایل SimConfig.ini انجام داد:

```
[PUM]
um_enabled = True
memory_type = "RRAM"
access_pattern = "row-major"
pum_specific_param1 = value1
pum_specific_param2 = value2
```



```
vim SimConfig.ini

Buffer_Choice = 1
# buffer choice option: 0: User defined, 1: SRAM, 2:DRAM, 3:RRAM
Buffer_Technology = 90
# buffer technology option: 0: default configurations, x:nm
Buffer_ReadPower = 0
# buffer read power option: 0: default configurations, x:mW
Buffer_WritePower = 0
# buffer write power option: 0: default configurations, x:mW
Buffer_Bitwidth = 64
# buffer bitwidth option: 0: default configurations, x:bit

LUT_Capacity = 1
# LUT capacity unit: Mb
LUT_Area = 0
# LUT are option: 0: default configurations, x: mm^2
LUT_Power = 0
# LUT power option: 0: default configurations, x:mW
LUT_Bandwidth = 0
# LUT bandwidth option: 0: default configurations, x:Mb/s
Tile_Connection = 2
# Option: 0, 1, 2, 3
Tile_Num = 64,64
# number of Tiles in accelerator (x,y): 0,0: default configuration (8x8), x,y: user defined

##### Algorithm Configuration #####

[Algorithm Configuration]
Weight_Polarity = 1
# 1 or 2
Simulation_Level = 0
# 0: Behavior, do not consider specific weight values; 1: Estimation, consider the specific weight values
NoC_enable = 0
# 0: not call booksim to simulate the NoC part; 1: Call booksim to simulate NoC part

##### add by me #####
[PUM]
pum_enabled = True
memory_type = "RRAM"
access_pattern = "row-major"
pum_specific_param1 = value1
pum_specific_param2 = value2

#shiftreg, adder, Outputmux, oreg =0
-- INSERT --

223,19 Bot
```

شکل ۱: اضافه کردن کانفیگ PUM

۲. توصیف شبکه (network.py):

اگر فایل network.py را باز کنیم، مشاهده می‌کنیم که همه تنظیمات لایه‌های مختلف شبکه در این فایل تنظیم شده است. می‌بایست این فایل را طوری تغییر دهیم که معماری PUM را به لایه‌های مختلف اضافه کنیم و آن را با لایه‌های مختلف ارتباط دهیم. برای انجام تغییرات باید به مسیر زیر برویم:

MNSIM/Interface/

برای مثال می‌توان مطابق با سایر تنظیمات همین فایل، یک if دیگر به کدها اضافه کرد.

```
if(cate.startswith('pum_net')):
    layer_config_list.append({'type': 'pum_conv', 'in_channels': 3,
                              'out_channels': 64, 'kernel_size': 3, 'padding': 1, 'stride': 2})
    layer_config_list.append({'type': 'relu'})
    layer_config_list.append({'type': 'pooling', 'mode': 'MAX',
                              'kernel_size': 2, 'stride': 2})
```

```
vim network.py
assert 0, f'not support {layer_config["type"]}, only conv and fc layers have weights'
weights_list = torch.split(total_weights, split_len, dim = 1)
# load weights
for i, weights in enumerate(weights_list):
    tmp_state_dict[tmp_key + f'.sublayer_list.{i}.weight'] = weights
# load weights
self.load_state_dict(tmp_state_dict)

def get_net(hardware_config = None, cate = 'lenet', num_classes = 10):
    # define the NN structure
    # initial config
    if hardware_config == None:
        hardware_config = {'xbar_size': 512, 'input_bit': 2, 'weight_bit': 1, 'ADC_quantize_bit': 10, 'DAC_num': 256}
    # layer_config_list, quantize_config_list, and input_index_list
    layer_config_list = []
    quantize_config_list = []
    input_index_list = []
    # layer by layer
    # add new NN models here (conv/fc is followed by one bn layer automatically):
    assert cate in ['lenet', 'vgg16', 'vgg8', 'alexnet', 'resnet18', 'pum_net']

    if cate.startswith('pum_net'):
        layer_config_list.append({'type': 'pum_conv', 'in_channels': 3, 'out_channels': 64, 'kernel_size': 3, 'padding': 1, 'stride': 2})
        layer_config_list.append({'type': 'relu'})
        layer_config_list.append({'type': 'pooling', 'mode': 'MAX', 'kernel_size': 2, 'stride': 2})
        # Add more layers as required by the PUM network architecture
    elif cate.startswith('lenet'):
        layer_config_list.append({'type': 'conv', 'in_channels': 3, 'out_channels': 6, 'kernel_size': 5})
        layer_config_list.append({'type': 'relu'})
        layer_config_list.append({'type': 'pooling', 'mode': 'MAX', 'kernel_size': 2, 'stride': 2})
        layer_config_list.append({'type': 'conv', 'in_channels': 6, 'out_channels': 16, 'kernel_size': 5})
        layer_config_list.append({'type': 'relu'})
        layer_config_list.append({'type': 'pooling', 'mode': 'MAX', 'kernel_size': 2, 'stride': 2})
        layer_config_list.append({'type': 'conv', 'in_channels': 16, 'out_channels': 120, 'kernel_size': 5})
        layer_config_list.append({'type': 'relu'})
        layer_config_list.append({'type': 'view'})
        layer_config_list.append({'type': 'fc', 'in_features': 120, 'out_features': 84})
        layer_config_list.append({'type': 'dropout'})
        layer_config_list.append({'type': 'relu'})
        layer_config_list.append({'type': 'fc', 'in_features': 84, 'out_features': num_classes})
    elif cate.startswith('vgg16'):
        layer_config_list.append({'type': 'conv', 'in_channels': 3, 'out_channels': 64, 'kernel_size': 3, 'padding': 1})
        layer_config_list.append({'type': 'relu'})
        layer_config_list.append({'type': 'conv', 'in_channels': 64, 'out_channels': 64, 'kernel_size': 3, 'padding': 1})
        layer_config_list.append({'type': 'relu'})
        layer_config_list.append({'type': 'pooling', 'mode': 'MAX', 'kernel_size': 2, 'stride': 2})
```

شکل ۲: اضافه کردن PUM به فایل network.py

۳. تغییر ماژول‌های شبیه‌سازی:

پس از مرتبط کردن لایه‌های شبکه با معماری PUM مورد نظر، می‌بایست توابع پیاده سازی آن نیز در سایر فایل‌های برنامه تعریف شود. برای انجام این کار می‌بایست به مسیر زیر برویم:

MNSIM/Hardware_Model/

و فایل های زیر را مطابق با نیازهایمان تغییر دهیم:

- (a) Crossbar.py
- (b) PE.py
- (c) Device.py
- (d) Crossbar.py
- (e) PE.py
- (f) Tile.py
- (g) Buffer.py

سوال ۳:

در اولین پیاده سازی، ابعاد ضربداری (Xbar) را به 256×256 تنظیم کنید. در پیاده سازی دوم، ابعاد ضربداری را به 128×128 تغییر دهید. مجموع تأخیر، توان و انرژی را گزارش دهید و جدول را پر کنید. چه اتفاقی افتاد؟ چرا؟ (برای هر پارامتر به تفصیل توضیح دهید.)

پاسخ:

به صورت پیش فرض Xbar سایز بر روی 256×256 تنظیم است. پس از نصب کتابخانه های مورد نیاز مثل:

1. numpy
2. torch
3. torch vision

در دایرکتوری شبیه ساز دستور زیر را اجرا می کنیم:

```
$ python3 main.py
```

فرایند اجرا چند دقیقه ای به طول خواهد انجامید و سیستم به شدت داغ خواهد شد! پس از اتمام شبیه سازی خروجی به صورت شکل «۳» زیر می شود:

اسکرین شات های خروجی همگی در مسیر images/ قرار دارد.

در گام دوم از فایل SimConfig.ini مقدار Xbar_Size را به 128,128 تغییر می دهیم و مجدداً فایل را شبیه سازی می کنیم. خروجی شبیه سازی در شکل «۴»

با توجه به مقادیر به دست آمده از شبیه سازی، جدول «۱» را تکمیل می کنیم

وضعیت	256*256	128*128	
کاهش	5330703.614503523 ns	5288665.312627485 ns	تأخیر
افزایش	338.62446474552047 W	1297.5432541884213 W	توان
افزایش	3827427.1532042585 nJ	8095960.154017576 nJ	انرژی

جدول ۱: نتایج شبیه سازی

```
reza@r324:/mnt/9636D17436D15639/University/CE/Memory Technologies/HWs/Simulation/HW04/Simulation/MNSIM-2.0

Other digital part energy: 4557.734307916861 nJ
|---adder energy: 0.0 nJ
|---output-shift-reg energy: 280.44595080000005 nJ
|---input-reg energy: 7.689798874130177 nJ
|---input_demux energy: 20.382072351621122 nJ
|---output_mux energy: 1.0883463335608319 nJ
|---joint_module energy: 4247.861500970414 nJ
NoC part energy: 0 nJ

Layer 0 :
  Hardware energy: 122627.03071149802 nJ
Layer 1 :
  Hardware energy: 637504.6088462156 nJ
Layer 2 :
  Hardware energy: 1562.3794680000003 nJ
Layer 3 :
  Hardware energy: 498557.24318860203 nJ
Layer 4 :
  Hardware energy: 997114.4863772041 nJ
Layer 5 :
  Hardware energy: 781.1897344000001 nJ
Layer 6 :
  Hardware energy: 499408.38667660207 nJ
Layer 7 :
  Hardware energy: 948875.9346855439 nJ
Layer 8 :
  Hardware energy: 390.59486720000007 nJ
Layer 9 :
  Hardware energy: 120429.11870449298 nJ
Layer 10 :
  Hardware energy: 48.82435840000001 nJ
Layer 11 :
  Hardware energy: 127.35558530003033 nJ
=====
Accuracy simulation will take a few minutes on GPU
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to /mnt/9636D17436D15639/University/CE/Memory Technologies/HWs/Simulation/HW04/Simulation/MNSIM-2.0/MNSIM/Interface/cifar10/cifar-10-python.tar.gz
100%|#####| 170498071/170498071 [00:52:00:00, 3226821.86it/s]
Extracting /mnt/9636D17436D15639/University/CE/Memory Technologies/HWs/Simulation/HW04/Simulation/MNSIM-2.0/MNSIM/Interface/cifar10/cifar-10-python.tar.gz
Files already downloaded and verified
Original accuracy: 0.9345454545454546
PIN-based computing accuracy: 0.9336363636363636
Mapping time: 2.299008369445801
Hardware modeling time: 0.1382122039794922
Accuracy modeling time: 305.9975235462189
Total simulation time: 308.43474411964417
reza@r324 /mnt/9636D17436D15639/University/CE/Memory Technologies/HWs/Simulation/HW04/Simulation/MNSIM-2.0
```

شکل ۳: خروجی شبیه سازی با سایز 256x256

توضیحات:

- **تأخیر:** کاهش ابعاد کراسبار از 256x256 به 128x128 باعث کاهش تأخیر شده است. این کاهش تأخیر به این دلیل است که کراسبار کوچک تر دارای مسیرهای انتقال داده کوتاه تر و پیچیدگی کمتری است، که منجر به کاهش زمان دسترسی و اجرای عملیات می شود. در نتیجه، مقدار تأخیر در کراسبار 128x128 کمتر از کراسبار 256x256 است.
- **توان:** افزایش توان در کراسبار 128x128 نسبت به 256x256 به این دلیل است که با کاهش ابعاد کراسبار، تعداد عملیات های پردازشی و تعداد دفعات دسترسی به حافظه افزایش می یابد. این افزایش عملیات های پردازشی باعث افزایش مصرف توان دینامیک می شود. بنابراین، توان مصرفی در کراسبار 128x128 بیشتر از کراسبار 256x256 است.
- **انرژی:** افزایش انرژی مصرفی در کراسبار 128x128 نسبت به 256x256 نتیجه مستقیم افزایش توان مصرفی و تأثیر کمتر کاهش تأخیر است. اگرچه تأخیر کاهش یافته است، اما افزایش توان مصرفی بیشتر بوده و در نتیجه انرژی مصرفی نیز افزایش یافته است. انرژی مصرفی تابعی از توان و تأخیر است و افزایش توان به طور کلی تأثیر بیشتری بر افزایش انرژی دارد. بنابراین، انرژی مصرفی در کراسبار 128x128 بیشتر از کراسبار 256x256 است.

سوال ۴:

در برخی لایه ها، توان ثابت است در حالی که تأخیر متفاوت است. علت چیست؟
پاسخ:


```
reza@r324:/mnt/9636D17436D15639/University/CE/Memory Technologies/HWs/Simulation/HW04/Simulation/MNSIM-2.0
Other digital part energy: 16834.971885395797 nJ
|---adder energy: 0.0 nJ
|---output-shift-reg energy: 621.57176064 nJ
|---input-reg energy: 7.41797148781065 nJ
|---input_demux energy: 19.868282206617604 nJ
|---output_mux energy: 2.412177271750656 nJ
|---joint_module energy: 16182.933413112429 nJ
NoC part energy: 0 nJ
Layer 0 :
Hardware energy: 196622.73067593365 nJ
Layer 1 :
Hardware energy: 1987729.8952463 nJ
Layer 2 :
Hardware energy: 1562.3794688000003 nJ
Layer 3 :
Hardware energy: 994716.091111115 nJ
Layer 4 :
Hardware energy: 1889960.5731111853 nJ
Layer 5 :
Hardware energy: 781.1897344000001 nJ
Layer 6 :
Hardware energy: 946597.4591827926 nJ
Layer 7 :
Hardware energy: 1843373.9994612276 nJ
Layer 8 :
Hardware energy: 390.5948672000007 nJ
Layer 9 :
Hardware energy: 233965.23383505346 nJ
Layer 10 :
Hardware energy: 48.82435840000001 nJ
Layer 11 :
Hardware energy: 211.1829651337925 nJ
=====
Accuracy simulation will take a few minutes on GPU
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to /mnt/9636D17436D15639/University/CE/Memory Technologies/HWs/Simulation/HW04/Simulation/MNSIM-2.0/MNSIM/Interface/cifar10/cifar-10-python.tar.gz
100% | 170498071/170498071 [00:45<00:00, 3778244.42it/s]
Extracting /mnt/9636D17436D15639/University/CE/Memory Technologies/HWs/Simulation/HW04/Simulation/MNSIM-2.0/MNSIM/Interface/cifar10/cifar-10-python.tar.gz to /mnt/9636D17436D15639/Univ
Files already downloaded and verified
Original accuracy: 0.9345454545454546
PIM-based computing accuracy: 0.9309090909090909
Mapping time: 1.2282086448669434
Hardware modeling time: 0.5889298915863037
Accuracy modeling time: 482.5058834552765
Total simulation time: 484.31592199172974
reza@r324 /mnt/9636D17436D15639/University/CE/Memory Technologies/HWs/Simulation/HW04/Simulation/MNSIM-2.0
```

شکل ۴: خروجی شبیه سازی با سایز 128x128

۱. **نوع عملیات در لایه ها:** لایه های مختلف شبکه عصبی (مانند لایه های کانولوشن، لایه های Fully connected و توابع فعال ساز) ممکن است عملیات های مختلفی را انجام دهند که نیاز به محاسبات و زمان متفاوتی دارند. حتی اگر توان مصرفی ثابت باشد، نوع و پیچیدگی عملیات می تواند باعث تغییر در تأخیر شود. به عنوان مثال، لایه های کانولوشن معمولاً به عملیات محاسباتی پیچیده تری نسبت به توابع فعال سازی نیاز دارند که می تواند تأخیر بیشتری ایجاد کند.
۲. **تعداد عملیات های موازی:** تعداد عملیات هایی که به صورت موازی در یک لایه اجرا می شوند می تواند تأثیر زیادی بر تأخیر داشته باشد. در لایه هایی که موازی سازی بیشتری دارند، تأخیر می تواند کاهش یابد حتی اگر توان مصرفی ثابت بماند. این موازی سازی می تواند وابسته به اندازه کراسبار و نحوه تقسیم وظایف بین واحدهای پردازشی باشد.
۳. **بار حافظه و دسترسی به داده ها:** نحوه دسترسی به داده ها و بار حافظه می تواند باعث تفاوت در تأخیر شود. حتی اگر توان مصرفی ثابت باشد، تأخیر دسترسی به حافظه و انتقال داده ها می تواند متفاوت باشد. لایه هایی که به داده های بیشتری نیاز دارند یا دسترسی های بیشتری به حافظه دارند، ممکن است تأخیر بیشتری داشته باشند.
۴. **تفاوت در الگوهای دسترسی به حافظه:** الگوهای دسترسی به حافظه در لایه های مختلف می تواند متفاوت باشد. به عنوان مثال، لایه هایی که نیاز به دسترسی تصادفی بیشتری دارند، ممکن است تأخیر بیشتری را تجربه کنند در حالی که توان مصرفی ثابت باقی می ماند. این تفاوت در الگوهای دسترسی می تواند به علت ساختار داده ها و نیازهای محاسباتی خاص هر لایه باشد.

سوال ۵:

کدام اندازه کراسبار بهتر است؟ لطفاً با جزئیات توضیح دهید. برای مثال، اگر تصمیم شما بر اساس توان، تأخیر یا انرژی است، باید علت و دلیل این شرایط را ذکر کنید.
با توجه به نتایج بدست آمده از شبیه سازی (جدول «۱») می توان گفت:

۱. کراسبار 128×128 تأخیر کمتری نسبت به کراسبار 256×256 دارد. کاهش تأخیر به معنای سریع تر انجام شدن عملیات ها و پردازش ها است که می تواند در برنامه هایی که نیاز به سرعت بالا دارند، مزیت بزرگی باشد.
 ۲. کراسبار 128×128 توان بیشتری نسبت به کراسبار 256×256 مصرف می کند. افزایش توان مصرفی به معنای نیاز به منابع انرژی بیشتر و احتمال افزایش گرما در سیستم است که می تواند به مشکلات حرارتی و نیاز به خنک سازی منجر شود.
 ۳. انرژی مصرفی در کراسبار 128×128 بیشتر از کراسبار 256×256 است. انرژی مصرفی تابعی از توان و تأخیر است و افزایش انرژی به معنای کارایی کمتر از نظر مصرف انرژی است.
- بنابر این با توجه به معیار ما برای انتخاب می توان یکی از حالت های زیر را انتخاب کرد:

- اگر اولویت تأخیر باشد: کراسبار 128×128 گزینه بهتری است زیرا تأخیر کمتری دارد و عملیات ها را سریع تر انجام می دهد.
- اگر اولویت توان و انرژی مصرفی باشد: کراسبار 256×256 گزینه بهتری است زیرا توان و انرژی کمتری مصرف می کند که می تواند به کاهش هزینه ها و مشکلات حرارتی کمک کند.