VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

MODELOVÁNÍ AKCELERÁTORŮ NEURONOVÝCH SÍTÍ

MODELLING OF NEURAL NETWORK HARDWARE ACCELERATORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JAN KLHŮFEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. VOJTĚCH MRÁZEK, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

141019

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Klhůfek Jan, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Vestavěné systémy
Název:	**Modelování akcelerátorů neuronových sítí**
Kategorie:	Umělá inteligence
Akademický rok:	2022/23

Zadání:

1. Seznamte se problematikou konvolučních neuronových sítí (NN) a s možnostmi odhadu parametrů hardwarové akcelerace jednotlivých vrstev NN na specializovaném hardware (např. EyeRiss, Google TPU, NVIDIA NVDLA).
2. Seznamte se s problematikou kvantizace a její implementace v akcelerátorech neuronových sítí.
3. Zpracujte rešerši výše uvedených témat.
4. Navrhněte konfigurovatelný model založený na existujícím modelu umožňující odhad latence a dalších parametrů pro různé architektury NN s odlišnou kvantizací vah i příznaků.
5. Navržený model implementujte.
6. Vyhodnoťte predikci latence a dalších parametrů na různých architekturách NN.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Mrázek Vojtěch, Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	31.10.2022

Abstrakt

Cílem této diplomové práce je zaměřit se na modelování akcelerátorů neuronových sítí s HW podporou kvantizace. Práce nejprve přibližuje koncept výpočtu konvolučních neuronových sítí (CNN) a představuje kategorie různých hardwarových architektur, které slouží k jejich zpracování. Následně jsou shrnuty optimalizační techniky modelů CNN, jejichž cílem je dosáhnout efektivního zpracování na specializovaných hardwarových architekturách. Další část práce obsahuje porovnání existujících analytických nástrojů, jež slouží k odhadu výkonnostních parametrů HW při inferenci, a které jsou rozšiřitelné o implementaci podpory kvantizace. Na základě experimentálního porovnání byl pro účely této práce vybrán nástroj Timeloop. Dále je představen popis fungování tohoto nástroje spolu s návrhem a implementací jeho rozšíření o podporu kvantizace. V závěru práce jsou experimentálně otestovány důsledky různých konfigurací kvantizace na vyhodnocené parametry inference u různých hardwarových architektur.

Abstract

The aim of this master thesis is modeling of neural network accelerators with HW support for quantization. The thesis first focuses on the concept of computation in convolutional neural networks (CNNs) and introduces different categories of hardware architectures that are used for their processing. Following this, optimization techniques for CNN models are summarized, with the goal of achieving efficient processing on specialized hardware architectures. The subsequent part of the thesis involves a comparison of existing analytical tools that are used to estimate hardware performance parameters during inference and which can be expanded to incorporate quantization support. Based on an experimental comparison, the Timeloop tool was selected for the purposes of this thesis. A thorough explanation of this tool's functionality is presented, along with a concept and implementation of its expansion to support quantization. In conclusion, the thesis experimentally tests the impact of various quantization configurations on evaluated inference parameters across different hardware architectures.

Klíčová slova

analytický model, HW akcelerátory, konvoluční neuronové sítě (CNN), kvantizace, prořezávání, optimalizace, energetická účinnost, Timeloop, Accelergy

Keywords

analytical model, HW accelerators, convolutional neural networks (CNN), quantization, pruning, optimization, energy efficiency, Timeloop, Accelergy

Citace

KLHŮFEK, Jan. *Modelování akcelerátorů neuronových sítí*. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vojtěch Mrázek, Ph.D.

Modelování akcelerátorů neuronových sítí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Vojtěcha Mrázka, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jan Klhůfek
10. července 2023

Poděkování

Rád bych tímto poděkoval svému vedoucímu Ing. Vojtěchovi Mrázkovi, Ph.D. za čas strávený při konzultacích i jeho odborné vedení, rady a nápady, jež přispěly ke zhotovení této diplomové práce. Dále bych chtěl moc poděkovat mé rodině za jejich neutuchající podporu. V neposlední řadě děkuji Mirku Šafářovi za dodání referenčních přesností modelů sítě ImageNet. Tato práce byla podpořena Grantovou agenturou České republiky v rámci projektu GA22-02067S.

Obsah

Kapitola 1

Úvod

Umělé neuronové sítě (ANN) zaznamenávají v posledních letech ohromný narůst na popularitě díky jejich široké využitelnosti v mnoha aplikačních doménách [17]. Ukázaly se být efektivní v úlohách zpracování obrazu, zpracování přirozeného jazyka, rozpoznávání řeči či robotice. S rostoucí komplexitou architektur ANN roste jejich přesnost, ovšem za cenu mnohonásobně se zvyšujících výpočetních nároků spojených s jejich trénováním a také s aplikací již natrénovaného modelu nad aplikačními daty (inference). K tomu navíc roste poptávka po zakomponování modelu do nízkopříkonových vestavěných zařízení pro zpracování v reálném čase [6, 59]. Tyto systémy však disponují omezenými výpočetními zdroji a je tedy žádoucí se usilovně věnovat jak optimalizaci modelů sítí, tak také návrhu HW architektur umožňujících jejich efektivní zpracování.

Proces trénování je sice výpočetně náročnější, ale je potřeba k vykonání pouze jednou, zatímco inference je zpracovávána opakovaně nad různými vstupními daty. Nároky na inferenci se úzce pojí s volbou systému, kde má být model nasazen. Buď to se jedná o distribuované cloudové řešení, nebo umístění do vestavěného systému – tzv. edge zařízení. Výběr cílové platformy závisí zejména na účelu použití a systémových požadavcích. Obecně platí že v cloudu se zpracovává velké množství dat a využívá se k tomu flexibilní a dobře škálovatelná distribuovaná infrastruktura s vysoce výkonnými procesními jednotkami. Naopak edge zařízení se používají pro inferenci v reálném čase a tyto systémy disponují omezenými, často málo flexibilními, HW zdroji, což vyžaduje nasazení specializované jednotky pro efektivní zpracování modelu sítě – HW akcelerátoru. Pro optimalizaci aplikace neuronových sítí na konkrétních hardwarových akcelerátorech se často využívají různé matematické modely. Tyto modely slouží k efektivnímu určení vhodného mapování vrstev sítě a jejich výpočetních operací na cílovou architekturu. Cílem této práce je zavést do těchto nástrojů podporu pro kvantizace vstupních dat, se záměrem dosažení efektivnějšího zpracování s ohledem na parametry jako je energie či latence.

Kapitola 2 se věnuje hrubé kategorizaci ANN s hlavním zaměřením se na konvoluční neuronové sítě (CNN), jejich strukturu a princip výpočtu spolu s vzorci pro určení počtu výpočetních operací potřebných ke zpracování a počtem parametrů nutných k uložení do paměti. Pro zpracování neuronových sítí pak existuje celá řada HW achitektur, lišících se škálovatelností, flexibilitou i specializovaností výpočtu. O jejich podrobnější klasifikaci pojednává kapitola 3. V kapitole 4 jsou shrnuty různé optimalizační metody a techniky vedoucí k redukci výpočetní i paměťové náročnosti a tím docílení efektivního zpracování modelů neuronových sítí.

Kapitola 5 se věnuje existujícím analytickým nástrojům umožňující spolehlivě odhadovat výkonnostní parametry systému spojené s procesem inference. Kapitola 6 pak obsahuje

experimentální porovnání dvou takových nástrojů – Timeloop+Accelergy a MAESTRO, s cílem vybrat vhodnější z nich pro účely této práce.

Vybraným nástrojem se nakonec stal Timeloop. Kapitola 7 se tak věnuje podrobnému popisu konfiguračních souborů tohoto nástroje, principů jeho fungování a navrhuje změny pro rozšíření o podporu kvantizace. Konkrétní implementace tohoto rozšíření je pak popsána v kapitole 8.

Kapitola 9 obsahuje experimentální vyhodnocení vlivu různé konfigurace kvantizace modelu na evaluaci parametrů HW inference. V rámci experimentů byl testován celkový počet nalezených platných mapování pro různé konfigurace a také nejlepší dosažené metriky napříč všemi mapováními. Jeden z testovacích scénářů obsahuje srovnání evaluace pro různě uniformně kvantizované modely sítě MobileNet. Pro tento scénář byly natrénovány referenční modely na datasetu Cifar-10, což umožnilo srovnání dosažených HW metrik vůči referenčním hodnotám přesnosti klasifikace.

Kapitola 2

Konvoluční neuronové sítě a princip jejich počítání

Umělé neuronové sítě či jen neuronové sítě (NN) představují výpočetní model inspirovaný strukturou a funkcí lidského mozku [20, 35]. Jejich hlavní předností oproti jiným algoritmům je schopnost zdokonalit se ve vykonávané činnosti učením se z velkého množství trénovacích dat, aniž by k tomu musely být explicitně programovány. Obecně se skládají ze vzájemně propojených vrstev uzlů, z nichž každý představuje umělý neuron a jednotlivé propoje symbolizují synapse sloužící pro přenos informace mezi dvojicí uzlů. Uzly v sobě zahrnují zpracovávané vstupní hodnoty, zatímco propoje obsahují hodnoty vah, které postupným průchodem sítí mění vstupní hodnoty[1]. Za účelem kompaktnější reprezentace dat a zjednodušení náročnosti na výpočet jsou aktivně zkoumány různé topologie sítí se záměrem zvýšit aplikační úspěšnost a zefektivnit způsob zpracování dat napříč vrstvami sítě. Existuje několik podkategorií ANN, různících se strukturou a doménou využití, příkladem jsou:

- Dopředné neuronové sítě (FNN) [2, 48]
- Hluboké neuronové sítě (DNN) [33]
- Konvoluční neuronové sítě (CNN) [31]
- Rekurentní neuronové sítě (RNN) [12]
- Generativní soupeřící sítě (GAN) [3]
- Spiking neuronové sítě (SNN) [38]

Dále se budeme věnovat pouze popisu konvolučních neuronových sítí, které našly své uplatnění zejména v oblasti zpracování obrazu a signálu. Nejprve si v podkapitole 2.1 objasníme strukturu CNN spolu s představením jednotlivých typů vrstev a následně se v podkapitole 2.2 zaměříme na princip výpočtu u jednotlivých vrstev spolu s určením jejich výpočetní náročnosti a také paměťových nároků spojených s uchováním parametrů modelu.

[1] Neurony navíc obsahují aktivační funkci, která zajistí, aby výstupní hodnota neuronu nebyla jen lineární transformací váženého součtu jeho vstupů.

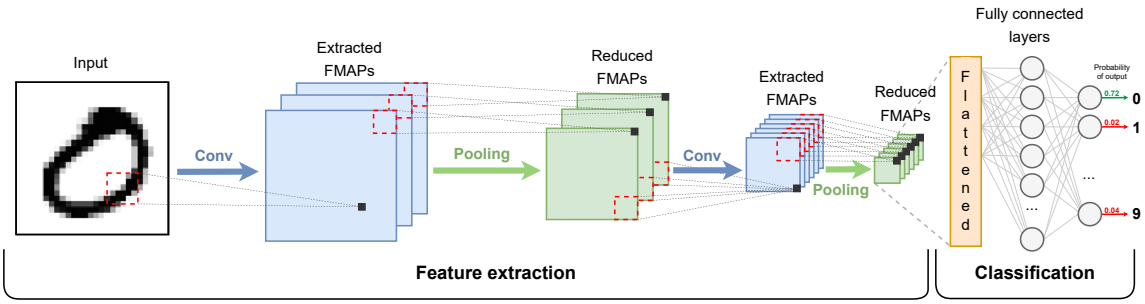
2.1 Struktura CNN

Topologie CNN se skládají ze tří fundamentálních typů vrstev. Patří mezi ně konvoluční vrstvy (*Conv*), pooling vrstvy (*Pooling*) a plně propojené vrstvy (*Fully connected*). Tyto vrstvy mohou být dále upravovány zavedením různých optimalizací, více o nich pojednává kapitola 4. Pokud je vrstev v konvoluční neuronové síti mnoho, hovoříme o nich jako o hlubokých neuronových sítích.

Vzhledem k charakteru vstupních dat pracují CNN s tenzory hodnot, které jsou zpravidla vícedimenzionální, a název *konvoluční* plyne z operace konvoluce, která je nejčastější výpočetní operací prováděnou v konvolučních NN.

- **Konvoluční vrstvy** obsahují sady filtrů, v nichž jsou hodnoty vah, které slouží k extrakci map příznaků[2] ze vstupních dat. Výstupní mapy příznaků zachycují určité prostorové vlastnosti vstupních dat, které se postupným průchodem přes více konvolučních vrstev zjemňují.
- **Pooling vrstvy** provádějí redukci map příznaků extrahovaných konvolučními vrstvami. To umožňuje nejen redukci prostorové dimenzionality a tím i počtu parametrů potřebných k uložení a výpočtu, ale také napomáhají předcházet přetrénování (*overfitting*) a přispívají k eliminaci šumu v datech [15].
- **Plně propojené vrstvy** se nacházejí na konci topologie sítě a slouží ke zpracování map příznaků generovaných zbytkem sítě. Představují hustou síť vzájemně propojených neuronů z jedné vrstvy do druhé. Každý neuron v plně propojené vrstvě tedy obdrží vstup ze všech neuronů předcházející vrstvy. Výstupem je pak výsledek pro danou úlohu pro odpovídající vstupní data.

Za dílčími konvolučními vrstvami a uvnitř jednotlivých neuronů u plně propojených vrstev se navíc vyskytují aktivační funkce pro zachování nelinearity dat. O různých typech aktivačních funkcí se můžete dozvědět více v [19]. Obrázek 2.1 vyobrazuje příklad topologie CNN pro klasifikaci ručně psaných číslic.



Obrázek 2.1: Příklad zjednodušené konvoluční neuronové sítě pro klasifikaci ručně psaných číslic. Šipky mezi tenzory značí substituce za konvoluční a pooling vrstvy a jednotlivé tenzory pak značí výstup z dané vrstvy. Počet výstupních 2D tenzorů map příznaků (dimenze hloubky) po aplikaci konvoluce značí počet filtrů použitých v rámci dané konvoluční vrstvy. První konvoluční vrstva tedy obsahuje celkem 3 filtry a druhá 6 filtrů. Inspirace pro tvorbu obrázku převzata z [33].

[2]Též zvané jako *Feature map* nebo zkráceně *FMAP*, případně jako *Activation map*.

2.2 Výpočetní a paměťové nároky na zpracování CNN

Komplexita zpracovávané sítě je úzce spojena s jejími výpočetními a paměťovými nároky. Síť při zpracování prochází dvěma fázemi – *trénováním* a *inferencí*.

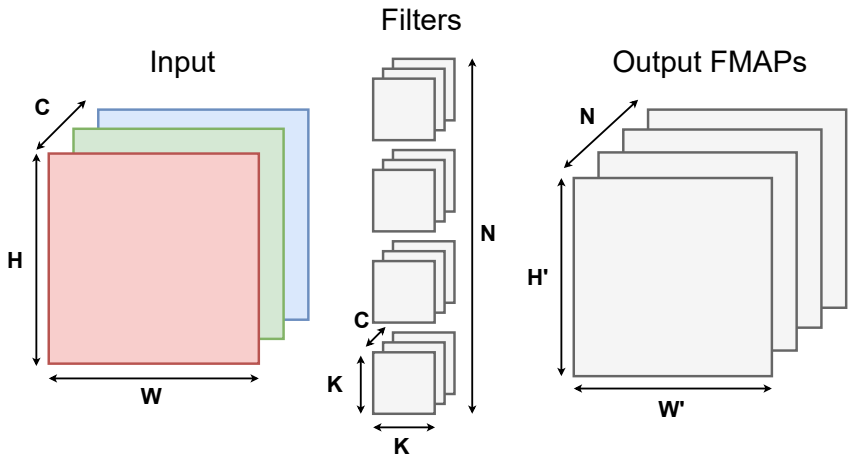
Trénování je výpočetně i paměťově náročnější, vzhledem k velkému počtu parametrů, které je třeba optimalizovat. Toho je docíleno s využitím optimalizačních algoritmů jako je stochastic gradient descent (SGD), které vyžadují několik dopředných a zpětných průchodů sítí k úpravě hodnot parametrů. Proces trénování se provádí pouze jednou a používají se k němu nejčastěji vysoce paralelní grafické procesory GPU, o kterých pojednává sekce 3.1 a tenzorové procesory TPU zmíněné v sekci 3.2. Pro získání dalších informací o výpočetní náročnosti spojené s trénováním viz článek [34].

Naopak zpracování **inference** je vykonáváno opakovaně nad různými vstupními daty v dopředném směru a rychlost zpracování souvisí s volbou systému, kde je model nasazen. Výpočetní náročnost inference konvolučních neuronových sítí je úzce spojena s vlastnostmi vrstev obsažených v dané topologii. Počet operací je závislý od typu vrstev, počtu vrstev a velikosti zpracovávaných dat [50]. Kalkulace se typicky provádí pro každou vrstvu zvlášť, jelikož jsou jednotlivé vrstvy zpracovávány postupně tak, jak jdou v síti za sebou. Je však také možné vrstvy shlukovat, což může být například v případě implementace sítě jakožto jeden obvod na FPGA. Celková komplexita je pak dána sumací dílčích mezivýsledků. Pro volbu systému je tak žádoucí vědět kolik výpočetních operací je třeba provést a také kolik místa v paměti budou parametry modelu zabírat.

Detailněji bude představena pouze inference modelu. Následují podsekce věnující se výpočetním a paměťovým nárokům jednotlivých typů vrstev CNN.

2.2.1 Výpočet konvolučních vrstev

Konvoluční vrstvy jsou z celé sítě výpočetně nejnáročnější, jelikož zpracovávají velké množství dat prostřednictvím operace konvoluce. Ta v režimu posuvného okna postupně násobí váhy filtru napříč vstupními daty, které následně sečte a vyrobí tak novou výstupní hodnotu. Filtr je složen z konvolučního jádra (*kernel*), který je nejčastěji čtvercového tvaru. Výchozí pozice kernelu závisí na hodnotě posunu (*stride*), který je typicky shodný pro vertikální i horizontální směr, a jenž se používá k podvzorkování vstupních dat. Posun kernelu pak ovlivňuje hodnota vyplnění (*padding*), jež slouží k rozšíření původních rozměrů vstupních dat o další hodnoty (často nuly), aby nedošlo k výrazné redukci dimenzí výstupních příznaků. Počet kernelů v jednom filtru je spojen s počtem kanálů vstupních dat pro danou vrstvu. Například při zpracovávání RGB barevných obrázků jsou vstupem pro první konvoluční vrstvu data se třemi kanály. Počet filtrů určuje počet kanálů extrahovaných příznaků. Obrázek 2.2 obsahuje obecnou podobu konvoluční vrstvy spolu s písmenným označením jednotlivých dimenzí, které v textu dále budeme užívat.



Obrázek 2.2: Struktura konvoluční vrstvy spolu s definicí jednotlivých dimenzí. Význam písmen je následující: K značí velikost konvolučního jádra, H, W, H', W' dimenze vstupních a výstupních tenzorů, C značí počet vstupních kanálů a N symbolizuje počet filtrů a počet výstupních kanálů.

Vzorce 2.1 a 8.3 obsahují výpočet dimenzí výšky respektive šířky výstupních tenzorů příznaků.

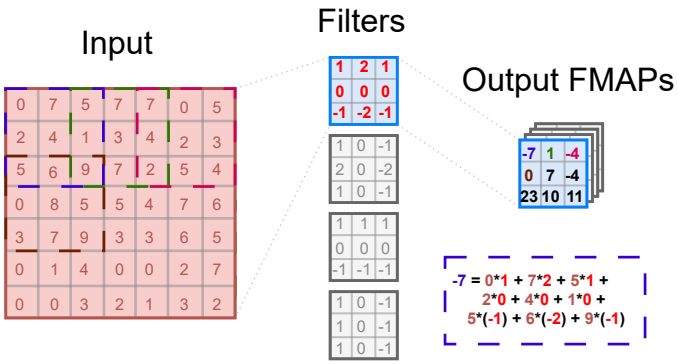
$$H' = \left\lfloor \frac{H - K + 2 * Padding}{Stride} \right\rfloor + 1 \tag{2.1}$$

$$W' = \left\lfloor \frac{W - K + 2 * Padding}{Stride} \right\rfloor + 1 \tag{2.2}$$

Výpočetní nároky

Výpočet konvoluční vrstvy sestává z násobení hodnot filtru s hodnotami vstupu a postupného sčítání získaných mezivýsledků. Každá taková operace typicky pracuje s desetinnými čísly a proto se označuje jako *FLOP*[3]. Souhrnně se pak operace násobení a sčítání nazývá *Multiply-and-accumulate* (*MAC*). Každá MAC operace tedy sestává z dvou FLOP operací. Obrázek 2.3 obsahuje ukázku výpočtu konvoluční vrstvy.

[3] **FL**oating-point **OP**eration – operace s plovoucí desetinnou čárkou



Obrázek 2.3: Ukázka výpočtu konvoluce u sítě s parametry $H = W = 7$, $C = 1$, $N = 4$, $K = 3$, $H' = W' = 3$. $Padding = 0$ a $stride = 2$.

Z obrázku 2.3 jsou navíc vyvoditelné vzorce platné pro určení celkového počtu MAC 2.3 a FLOP 2.4 operací[4].

$$MAC_{CONV} = H' * W' * N * K^2 * C \tag{2.3}$$

$$FLOP_{CONV} = H' * W' * N * K^2 * C * 2 = MAC_{CONV} * 2 \tag{2.4}$$

Paměťové nároky

Jedinými parametry nutnými k uložení do paměti jsou váhy s biasy obsažené ve filtrech. Bias je skalární hodnota přidružená každému filtru, která se po aplikaci konvoluce přičte k výsledku. Vzorec pro celkový počet vah napříč všemi filtry popisuje vzorec 2.5. Celkový počet Bytů zabíraných v paměti pak závisí na počtu Bytů potřebných k uložení hodnot vah a biasů, viz vzorec 2.6.

$$Parameters_{CONV} = (C * K^2 + 1) * N \tag{2.5}$$

$$Bytes_{CONV\_w} = \left\lceil \frac{(C * K^2) * N * size\_of\_weight}{8} \right\rceil$$

$$Bytes_{CONV\_b} = \left\lceil \frac{N * size\_of\_bias}{8} \right\rceil$$

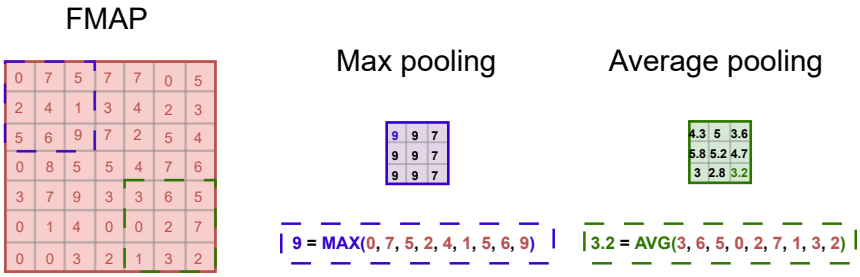
$$Bytes_{CONV} = Bytes_{CONV\_w} + Bytes_{CONV\_b} \tag{2.6}$$

2.2.2 Výpočet pooling vrstev

Pooling vrstvy v sobě obsahují, podobně jako konvoluční vrstvy, filtr. Ten však narozdíl od toho konvolučního neobsahuje žádné hodnoty a je vždy jen jeden pro danou vrstvu. Rozměry filtru jsou zpravidla čtvercového tvaru. Symbolem P zde budeme označovat šířku a výšku pooling filtru. Posun v horizontálním a vertikálním směru pak určuje parametr *stride*. Filtr se používá k vykonání zadané funkce nad odpovídajícími vstupními daty (opět se zde uplatňuje princip posuvného okna jako u konvoluce). Nejčastěji se využívají funkce maxima (*max pooling*) a průměru (*average pooling*). Pro zjištění míry redukce dimenzí lze

[4]Vzorce ve svých kalkulacích neuvažují *bias*, který je často z výpočtu vynechán.

použít vzorce pro výpočet rozměrů výstupních tenzorů konvoluce 2.1 a 8.3 s tím, že hodnota $padding = 0$. Ukázku aplikace max pooling a average pooling funkcí nad vstupní mapou příznaků obsahuje obrázek 2.4.



Obrázek 2.4: Ukázka aplikace MAX a $AVERAGE\ POOLING$ nad vstupní mapou příznaků. Hodnoty parametrů jsou následující $C = 1$, $H' = W' = P = 3$, $H = W = 7$, $stride = 2$.

Výpočetní nároky

Počet operací nutných k vykonání v rámci pooling vrstvy je řádově zanedbatelný oproti náročnosti u konvolučních vrstev a celková složitost se odvíjí od použité funkce. Vzorec 2.7 obsahuje obecný výpočet pro počet potřebných pooling operací s plovoucí desetinnou čárkou.

$$FLOP_{POOL} = H' * W' * C * P^2 \tag{2.7}$$

Paměťové nároky

Pooling vrstvy ve filtru neuchovávají žádné parametry a není tedy potřeba nic ukládat do paměti. Samotné informace o funkci, rozměrech filtru a stridu patří mezi fixní hyperparametry[5] sítě.

2.2.3 Výpočet plně propojených vrstev

V plně propojených vrstvách jsou všechny neurony z aktuální vrstvy napojeny na všechny neurony z vrstvy předcházející, viz fully connected layers na obrázku 2.1. Každý připojující se neuron má tedy přidruženou vlastní váhu a sumací všech takovýchto neuronů se vypočte výstup pro jeden neuron aktuální vrstvy. Na výstupní hodnotu se navíc aplikuje aktivační funkce. Vzorec 2.8 obsahuje výpočet výstupu jednoho neuronu.

$$m_j = f\left(\sum_{i=1}^{N} w_{ij} * n_i + b\right) \tag{2.8}$$

[5]Parametry, které se neučí během trénování a zůstávají fixní po celou dobu trénování a inference. Při inferenci v HW se nahrají z paměti jakožto konfigurační parametry.

kde:

f = aktivační funkce
N = počet vstupů pro aktuální vrstvu (počet neuronů v předcházející vrstvě)
M = počet neuronů v aktuální vrstvě
n_i = i-tý výstup z předcházející vrstvy, kde $i \in [1, N]$
m_j = výstup pro aktuální j-tý zpracovávaný neuron, kde $j \in [1, M]$
w_{ij} = váha propoje mezi neurony n_i a m_j
b = bias

Výpočetní nároky

Počet MAC a FLOP operací závisí na velikosti předcházející a aktuální plně propojené vrstvy. Celkově platí pro výpočet MAC vzorec 2.9 a pro výpočet FLOP vzorec 2.10.

$$MAC_{FC} = N * M \tag{2.9}$$

$$FLOP_{FC} = N * M * 2 = MAC_{FC} * 2 \tag{2.10}$$

Paměťové nároky

Jediné parametry, které je nutné uložit do paměti jsou váhy přidružené k jednotlivým propojům mezi dvojicemi neuronů a také hodnoty biasů pro výstupní neurony. Pro určení počtu parametrů platí vzorec 2.11. Celkový počet Bytů zabíraných v paměti závisí na velikosti vah a biasů, tedy platí vzorec 2.12.

$$Parameters_{FC} = (N + 1) * M \tag{2.11}$$

$$Bytes_{FC\_w} = \left\lceil \frac{N * M * size\_of\_weight}{8} \right\rceil$$

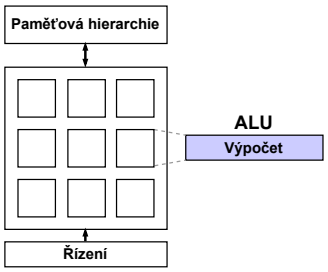
$$Bytes_{FC\_b} = \left\lceil \frac{M * size\_of\_bias}{8} \right\rceil$$

$$Bytes_{FC} = Bytes_{FC\_w} + Bytes_{FC\_b} \tag{2.12}$$

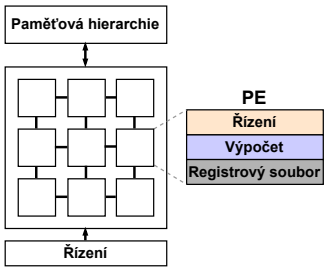
Kapitola 3

Hardwarové architektury pro zpracování konvolučních neuronových sítí

Hardwarové architektury určené ke zpracování neuronových sítí se označují jako akcelerátory [4, 13, 49] a obvykle se skládají ze specializované paměťové hierarchie a pole MAC jednotek realizujících operaci vynásob a sečti. Akcelerátory často využívají vysoce paralelních paradigmat, které je dělí na tzv. časové (*temporal*) a prostorové (*spatial*) architektury. Volba architektury silně závisí na typu zpracování – *trénování* nebo *inference*. Různorodé HW architektury se od sebe odlišují zejména datovými toky (*dataflows*), které určují jak mohou být data rozdělena a naplánována k výpočtu a jak je lze rozmístit napříč jednotlivými úrovněmi paměťové hierarchie. Datové toky mají významný dopad na výkonnost a energetickou účinnost vzhledem k možnostem opakovaného použití dat či míře zužitkování hardwaru [8]. Pro ukázku struktury *temporal* architektury viz obrázek 3.1, ukázku struktury *spatial* architektury pak lze vidět na obrázku 3.2.



Obrázek 3.1: Struktura časové (temporal) architektury. Obrázek převzat z [4].



Obrázek 3.2: Struktura prostorové (spatial) architektury. Obrázek převzat z [4].

Se stagnací ve výrazném zlepšení pro zpracování DNN u *temporal* architektur pro obecné použití [21] se větší pozornost upírá směrem k flexibilnějším doménově specifickým *spatial* architekturám. Vzhledem k nepřebernému množství *spatial* architektur a k nim přidruženým možným datovým tokům vzniká velký prostor architektur. Je navíc žádoucí, aby byly architektury co nejvíce flexibilní a tím umožnily odlišné datové toky v rámci dané DNN. Toho je docíleno skrze různé **konfigurace** (**mapování**) HW architektury pro daný **problém** (DNN, vrstvu sítě).

U obou typů architektur se zaměříme na jejich realizaci zpracování a zmíníme jejich hlavní charakteristiky. Bude nás zajímat i výkonnost v podobě FLOPS[1], latence či energetická účinnost (FLOPS/W).

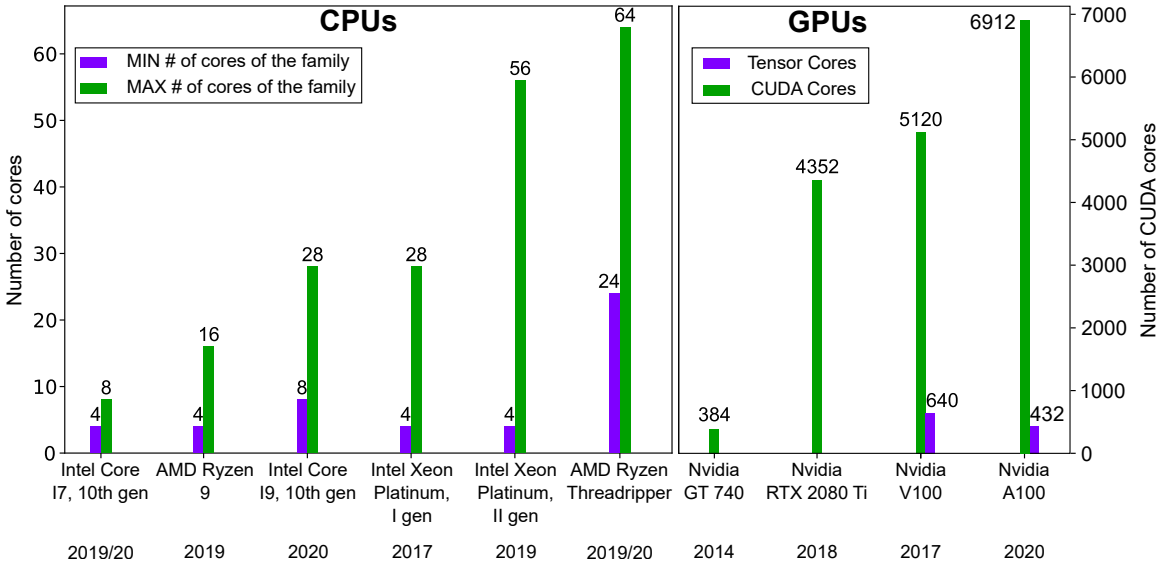
3.1 Temporální hardwarové architektury

Mezi *temporal* architektury se řadí víceúčelové procesory (CPU) a grafické procesory (GPU), u nichž je paralelizace docíleno s využitím vektorových jednotek (SIMD[2]) a paralelních vláken (SIMT[3]). Tyto architektury k samotnému výpočtu používají aritmeticko-logických jednotek (ALU), které si data musejí jednotlivě získat z paměti a nemohou mezi sebou vzájemně komunikovat.

K maximalizaci výkonu temporal architektur je potřeba se zabývat optimalizací sítě a modelu. Cílem je minimalizovat nespojité přístupy do paměti, zvýšit propustnost[4] a co nejvíce zužitkovat výpočetní jádra. Za tímto účelem se používají vysoce optimalizované knihovny jako je Basic Linear Algebra Subroutines (BLAS) nebo CuDNN vyvinutá společností NVIDIA. Tyto knihovny umožňují efektivní výpočet konvoluce převodem na operaci general matrix multiplication (GEMM) [52]. Jinou možností je použít algoritmus pro rychlou Fourierovu transformaci (FFT) převádějící operaci konvoluce z časové domény do domény frekvenční nebo provést redukci modelu pomocí prořezávání (*pruning*) či kvantizace. Zmíněným optimalizačním technikách se podrobněji věnuje kapitola 4.

CPU jádra je za určitých podmínek možné použít k inferenci, ale k trénování nejsou vhodné vzhledem k jejich omezené propustnosti limitované nízkým počtem jader a tím i nízké možnosti paralelizovat výpočet. Naopak GPU jsou velmi užívané jak pro trénování, tak inferenci díky jejich vysokému počtu výpočetních jader (řádově v tisících), což jim umožňuje efektivně vykonávat algoritmy pro maticové násobení. Navíc dnešní vysoce výkonné GPU obsahují speciální tenzorová jádra přímo uzpůsobená k zrychlení maticových výpočtů. Populární frameworky pro hluboké učení jako jsou Caffe, TensorFlow a PyTorch obsahují podporu pro zpracování výpočtu neuronových sítí na GPU[5]. Obrázek 3.3 ukazuje srovnání mezi počtem jader u vybraných CPU a GPU rodin procesorů.

[1] Počet FLOP operací vykonaných za sekundu
[2] Single Instruction, Multiple Data
[3] Single Instruction, Multiple Threads
[4] Množství dat (např. obrázků), které akcelerátor dovede zpracovat za jednotku času.
[5] NVIDIA deep learning frameworks documentation: https://docs.nvidia.com/deeplearning/frameworks/index.html



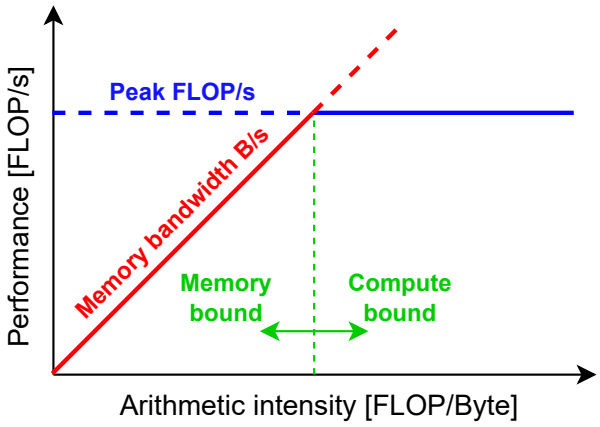
Obrázek 3.3: Porovnání počtu jader u různých modelů CPU a GPU. Fialové a zelené sloupce u CPU značí minimální respektive maximální počet jader pro danou procesorovou rodinu. U GPU reprezentuje fialový sloupec počet tenzorových jader a zelený sloupec počet CUDA jader. Obrázek převzat z [4].

3.1.1 Výkonnostní parametry

Při zpracování DNN je stěžejní dosažená výkonnost (FLOPS), latence zpracování (ms), propustnost systému (FPS[6]), míra využitelnosti výpočetních jednotek, energetická účinnost (FLOPS/W) a s ní spojená celková energetická spotřeba (W), která závisí především na počtu přístupů do vyšších úrovní paměťové hierarchie. Přístupy do paměti totiž zásadním způsobem ovlivňují jak spotřebu energie, tak propustnost systému. K výpočtu jedné MAC operace je potřeba provést tři čtení z paměti, a to konkrétně pro hodnoty vstupního příznaku, váhy filtru a nakonec částečného akumulovaného součtu. Navíc je třeba provést jeden zápis do paměti pro aktualizovaný částečný součet po provedení MAC operace. Je třeba zdůraznit, že přístupy do paměti jsou mnohonásobně energeticky nákladnější než výpočetní operace a je tedy cílem je co nejvíce snížit. Toho můžeme dosáhnout buďto zvýšením přenosového pásma a nebo důkladnějším čtením operandů z paměti, kdy se zohledňuje prostorová a časová lokalita dat.

K pochopení a vizualizaci výkonnosti vícejádrových systému se používá tzv. *roofline model* [55]. Ten vykresluje maximální teoretický výkon systému oproti skutečnému dosaženému při zpracování daného algoritmu, přičemž zohledňuje faktory jako je šířka přenosového pásma a aritmetická intenzita. Aritmetická intenzita udává počet FLOP operací vykonaných danou částí kódu vůči počtu přístupů do paměti (Bytů), které je potřeba k výpočtu dodat. Nízká aritmetická intenzita značí, že je systém vázaný na paměť, která nestačí dodávat data k výpočtu. Naopak vysoká aritmetická intenzita značí efektivní využití načtených dat k provedení operací nad nimi a v takovém případě je systém výpočetně vázaný. Obrázek 3.4 zobrazuje ukázku roofline modelu.

[6] Frames per second



Obrázek 3.4: Vizualizace roofline modelu. Linií pro paměťovou propustnost bývá více a jednotlivé z nich pak zohledňují šířku pásma pro danou úroveň paměťové hierarchie. Podobně tak i linií maximální výkonnosti je více a ty znázorňují míru použité paralelizace.

Obecně se dá říct, že GPU jsou lepší volbou pro zpracování hlubokých neuronových sítí než CPU vzhledem k větší možnosti paralelizace výpočtu, mnohem větší paměťové propustnosti díky technologii High Bandwidth Memory (HBM), a také jelikož obsahují specializovanější paměťovou hierarchii – a sice sdílenou paměť přímo na čipu díky níž je počet přístupů do DRAM podstatně menší. GPU také nabízejí lepší cachování, což je zvláště výhodné pro konvoluční operace, které jsou nejčastější výpočetní operací v CNN.

3.2 Spatial hardwarové architektury

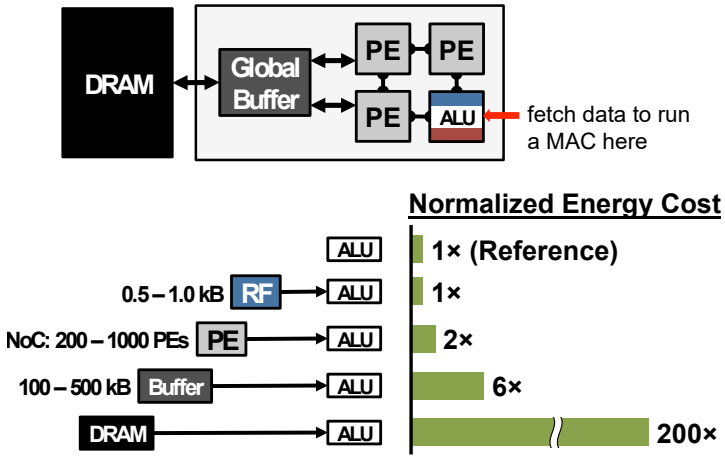
Spatial architektury se využívají v aplikačně specifických obvodech (ASIC) či programovatelných hradlových polích (FPGA). Zde se využívá vzájemného propojení tzv. procesních elementů (PE), které v sobě mimo MAC jednotky často obsahují i řídicí logiku a malý registrový soubor (RF/scratchpad) pro uchování specifických dat k dalšímu výpočtu. Jednotlivé PE jsou navíc vzájemně propojeny pomocí sítě propojů (NoC) a utvářejí tak řetězové zpracování.

FPGA se vyznačují vysokou mírou rekonfigurovatelnosti, což jim umožňuje být flexibilní vůči široké škále možných aplikací. V porovnání s nimi jsou ASIC akcelerátory navrhovány pro předem specifikované použití, umožňující tak zohlednit i konkrétní požadavky a kritéria na výsledný systém týkající se bezpečnosti, energetické účinnosti, výkonnosti a plochy zabírané na čipu. Doba a cena návrhu ASIC architektur je výrazně vyšší, protože zahrnuje tvorbu masky, simulace, verifikace a testování, aby se zajistilo, že obvod bude fungovat správně. Navíc při každé změně návrhu je třeba celý proces opakovat. Na rozdíl od toho u FPGA lze snadno změnit logickou implementaci bez nutnosti fyzického přeprogramování celého obvodu.

Spatial architektury se primárně používají jen k inferenci vzhledem k jejich omezené hrubé výpočetní síle, která je jinak typická pro CPU a GPU. Výjimkou jsou například tenzorové procesní jednotky (TPU), které mohou být nasazeny jak pro trénování v distribuovaných cloudových systémech, tak při zpracování inference v cloudu i edge zařízeních. TPU architektury se skládají ze systolického pole procesních jednotek a podrobněji se jim věnuje článek [24].

3.2.1 Výkonnostní parametry

Jak již bylo zmíněno u temporal architektur, hlavním problémem z hlediska latence a energie jsou přístupy do externí paměti. K jejich snížení disponují spatial architektury několikaúrovňovou paměťovou hierarchií. Ta jim umožňuje uchovávat data určená ke znovupoužití v menších a rychlejších pamětech s nižší energetickou cenou za přístup. Na obrázku 3.5 je znázorněna 3 úrovňová paměť spolu s normalizovanými energetickými nároky spojenými se čtením dat z různé úrovně hierarchie. Z normalizovaných hodnot je patrné, že každý přístup do DRAM je až o 2 řády dražší. Dále budeme obecně uvažovat takovouto 3 úrovňovou paměťovou hierarchii.



Obrázek 3.5: Struktura 3 úrovňové paměťové hierarchie a znázornění znormalizované energetické ceny spojené s přístupem k datům z různých úrovní paměťové hierarchie. Obrázek převzat z [49].

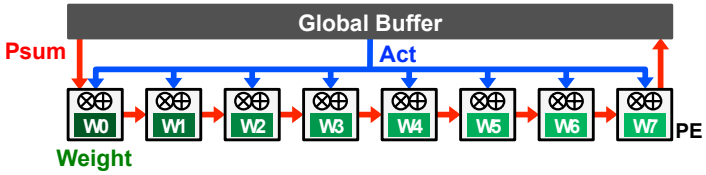
Za účelem co nejvíce snížit přístupy do externí DRAM paměti se začaly prozkoumávat různé možnosti datových toků. Ty určují pořadí v jakém jsou data čtena do menších pamětí a také způsob namapování samotného výpočtu prostřednictvím PEs. Přitom se snaží co nejvíce maximalizovat znovupoužití dat na čipu a celkovou propustnost systému. Pro zpracování konvolučních vrstev se v akcelerátorech uvažují tři podoby znovupoužití vstupních dat:

- Znovupoužití **vah**: kernel vah je použit celkem $H' * W'$ krát v rámci vstupních feature maps
- Znovupoužití **vstupních příznaků**: na stejná vstupní data se aplikují všechny filtry vrstvy (celkem N krát)
- **Konvoluční** znovupoužití: využívá vzájemného prolínání posuvných oken při posunu filtru přes vstupní data, což vede k možnosti výpočtu více výstupů zároveň

V případě plně propojených vrstev lze využít toho, že všechny vstupy předcházející vrstvy se používají k výpočtu všech neuronů vrstvy následující. Datové toky mohou být mapovány fixně podle určitého přístupového vzorce, případně flexibilně s využitím prohledávání prostoru mapování pomocí mapperů. K automatickému prohledávání vhodného mapování spolu s vyhodnocením jeho výkonnostních parametrů slouží analytické nástroje,

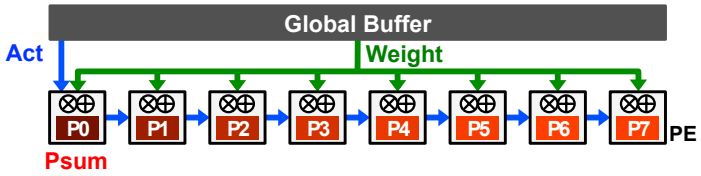
o nichž pojednává kapitola 5. Často se v akcelerátorech implementuje HW podpora určitého druhu datového toku, který pak výrazně omezuje flexibilitu možných mapování výpočtu. Následuje představení nejčastějších HW optimalizací k podpoře různých datových toků.

Weight stationary (WS) datový tok minimalizuje přístupy do paměti spojené se čtením vah tím, že uchovává hodnoty přímo v RF přidružených PE. Zatímco váhy zůstávají po dobu výpočtu stacionární, vstupní data spolu s částečnými součty jsou distribuovaná napříč sítí PE. Snahou je provést co nejvíce MAC operací se stejnými váhami. WS využívá **znovupoužití vah** a **konvoluční znovupoužití filtrů**. Příkladem akcelerátoru používající WS je TPU [18]. Pro představu fungování Weight stationary dataflow, viz obrázek 3.6.



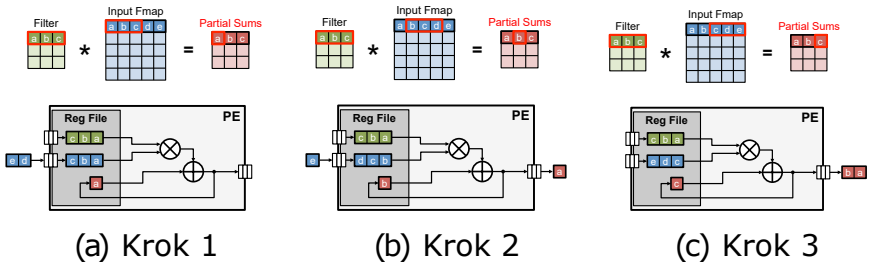
Obrázek 3.6: Ukázka principu datového toku WS. Převzato z [49].

Output stationary (OS) datový tok minimalizuje přístupy do paměti spojené se čtením a zápisem částečných součtů jejich postupnou akumulací v RF jednotlivých PE. K zachování stacionarity částečných součtů v RF jsou vstupní hodnoty proudově dodávány do jednotlivých PE, do nichž je rozeslána (broadcasting) stejná hodnota váhy. OS co nejvíce zužitkovává **konvoluční znovupoužití**. Zástupcem akcelerátorů implementující OS je ShiDianNao [14]. Princip Output stationary dataflow je zachycen na obrázku 3.7.

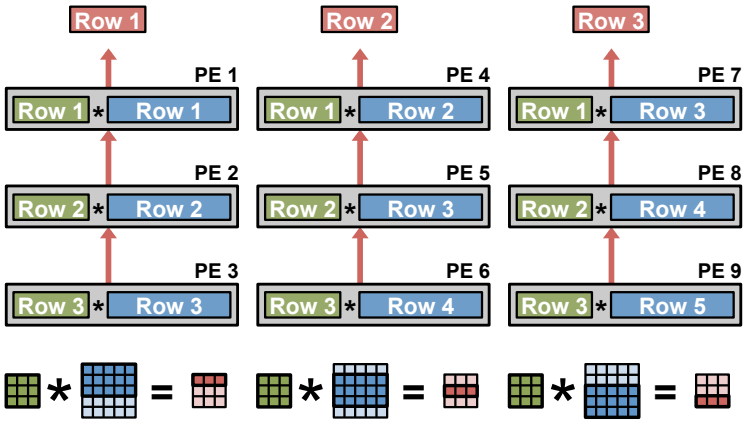


Obrázek 3.7: Ukázka principu datového toku OS. Převzato z [49].

Row stationary (RS) datový tok společně maximalizuje **znovupoužitelnost všech typů dat** (vstupy, váhy, částečné součty). RS přiřazuje každému PE výpočet jednořádkové (1-D) konvoluce. Váhy zůstávají stacionární uvnitř RF. S každou procesní jednotkou zpracovávající 1-D konvoluci, se poté musí agregovat výsledky z více PE, aby realizovaly 2-D, případně 3-D, konvoluci. Eyeriss je populárním akcelerátorem využívající (a představující) RS datový tok [10, 9]. Obrázek 3.8 vyobrazuje princip 1-D konvolučního znovupoužití při výpočtu a obrázek 3.9 pak princip 2-D konvolučního znovupoužití pro výpočet.

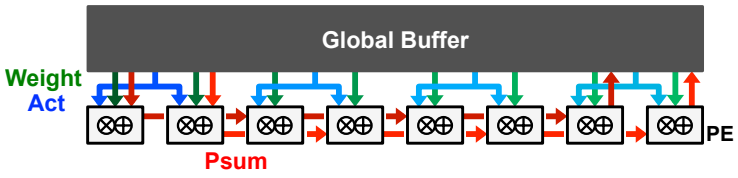


Obrázek 3.8: Znázornění principu 1-D konvolučního znovupoužití v rámci PE u Row Stationary Dataflow. Převzato z [49].



Obrázek 3.9: Znázornění principu 2-D konvolučního znovupoužití napříc prostorovým polem PE u Row Stationary dataflow. Převzato z [49].

No Local Reuse (NLR) odstraňuje RF z procesních jednotek, čímž redukuje plochu čipu, a čte data přímo z globální paměti. Malé registrové soubory jsou totiž sice efektivní z pohledu spotřeby energie (pJ/bit), ale neefektivní z pohledu zabírané plochy (μm^2/bit). Ušetřená plocha se dá využít ke zvětšení kapacity globální paměti. NLR tedy neumožňuje žádné znovupoužití dat a data pro zpracování jsou čtena z globální paměti. Příkladem NLR architektury je DianNao [7]. Fungování No Local Reuse představuje obrázek 3.10.



Obrázek 3.10: Ukázka principu datového toku NLR. Převzato z [49].

3.2.2 Shrnutí

Ke srovnání různých datových toků je třeba uvažovat stejné HW parametry jako jsou celková plocha, počet PE, velikost pamětí, šířka přenosového pásma, apod. Pro podrobnější srovnání viz [49]. Při porovnávání akcelerátorů s rozličnými HW specifikacemi nás zajímají především

parametry plochy (mm^2), spotřeby energie (W), výkonnosti ($FLOPS$) a také technologie použité při výrobním procesu (nm). Srovnání HW architektur k nalezení zde [13].

Datové toky jsou bezpochyby jedním z klíčových faktorů ovlivňujících výkonnostní parametry systému při inferenci. Mimo ně existuje i řada optimalizačních technik, které mohou svou aplikací výrazně zlepšit efektivitu zpracování inference. Patří mezi ně optimalizace HW architektury (například jinou implementací PE), případně použití jiných, nově vznikajících technologií jako je počítání přímo v paměti (in-memory computing) [53]. Další technikou je optimalizace samotného výpočtu použitím efektivních numerických metod pro výpočet konvolučních operací nebo použitím speciálních metod pro řešení lineárních soustav. Pro inferenci je pak stěžejní i redukce velikosti modelu. Ta s sebou může nést i ztrátu na původní přesnosti, a proto se musí provádět obezřetně. Redukce modelu je docíleno například prostřednictvím knowledge distillation, prořezáváním nebo různými metodami kvantizace, které všechny mohou výrazně snížit počty přístupů do paměti. Optimalizacím se souhrnně věnuje následující kapitola 4.

Kapitola 4

Optimalizace konvolučních neuronových sítí pro efektivní inferenci v HW

Vzhledem k rostoucí komplexitě modelů CNN coby do počtu jejich parametrů (řádově stovky milionů až miliardy) a současně s tím zvyšující se poptávce po integraci aplikací CNN do nízkopříkonových vestavěných zařízení pro real-time zpracování [6, 59] je nezbytné se zabývat efektivitou zpracování inference. To vede na různé možnosti optimalizace jak na straně softwaru, tak i hardwaru [56, 11, 16, 32].

Jedním z přístupů k optimalizaci CNN je použití efektivních numerických metod k výpočtu konvoluce, čímž dosáhneme snížení výpočetní komplexity a tím také zrychlení zpracování. Tyto metody zahrnují například *Winograd convolution*, *separable convolution*, *FFT* nebo *GEMM*. Optimalizaci výpočtu CNN se dále věnuje sekce 4.1.

Další možností jsou techniky ke kompresi modelu, které zmenší velikost sítě a s tím i počet parametrů nutných pro výpočet, aniž by byla výrazně ohrožena původní aplikační přesnost. Populárními technikami jsou *knowledge distillation*, *low-rank factorization*, *pruning* a *kvantizace*. Techniky pro kompresi modelu jsou podrobněji popsány v sekci 4.2 s hlavním zaměřením se na pruning a kvantizace.

Pro úplnost uvedeme i názvy významných technik užívaných pro lepší zpracování trénovací fáze, ale nebudeme se jimi blíže zabývat. Ke zrychlení trénování jsou klíčovými technikami *batch normalizace* [22], *dropout* [45] a *data augmentation* [44]. Pro trénování se využívají hlavně GPU a TPU, jak již bylo zmíněno v kapitole 3.

Výše zmíněné optimalizační metody a techniky lze často kombinovat s cílem dosažení větší míry redukce sítě a zefektivnění výpočtu pro inferenci. Následují sekce věnující se výše zmíněným druhům optimalizací modelu pro inferenci.

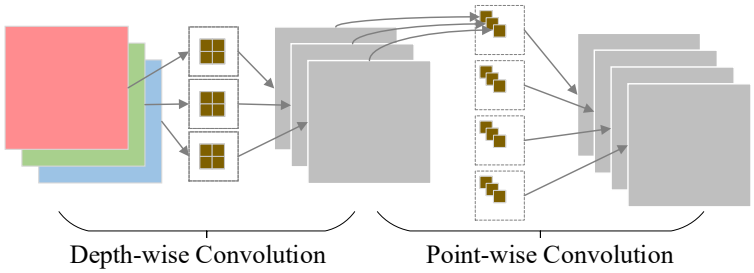
4.1 Optimalizace výpočtu CNN

Optimalizace výpočtu konvoluční neuronové sítě se zabývá metodami a technikami, které zlepšují efektivitu a rychlost výpočtu při provádění operací konvoluce a dalších operací v CNN. Cílem optimalizace je dosáhnout rychlejší inference sítě a snížení nároků na výpočetní prostředky.

4.1.1 Separable konvoluce

Separable konvoluce je technika užívaná k redukci počtu FLOP operací spojených s výpočtem konvoluce. Při běžné konvoluci se konvoluční jádro aplikuje napříč všemi kanály vstupního tenzoru. Naopak při separable konvoluci se kernel rozdělí na dvě části, na tzv. *depthwise* kernel a *pointwise* kernel.

Operace konvoluce je tedy provedena ve dvou postupných krocích. Nejprve se použije depthwise kernel nezávisle na každý vstupní kanál zvlášť a vyprodukuje tak mezivýsledek jehož počet výstupních kanálů odpovídá počtu vstupních kanálů. Následně se aplikuje pointwise konvoluční jádro (s fixní velikostí $1 \times 1 \times C$) na výstup z předcházející depthwise konvoluce k získání finálního výsledku. Počet filtrů je shodný s počtem filtrů původní konvoluce. Díky tomuto procesu je dosaženo snížení celkového počtu vykonaných operací a zároveň je zachována přesnost původního výpočtu. Obrázek 4.1 vyobrazuje fungování obou kroků separable konvoluce na obecném příkladu sítě.



Obrázek 4.1: Zobrazení rozložení běžné konvoluce s parametry $C = 3$, $N = 4$, $K = 2$ do podoby dvoufázové separable konvoluce. Kdybychom však pro jednoduchost uvažovali $H' = W' = 4$, $N = 128$, $stride = 1$, $padding = 0$ (ostatní parametry ponechány viz obrázek) byl by za použití vzorce 2.3 celkový počet MAC operací u klasické konvoluce roven 24 576 MAC, zatímco u separable konvoluce jen 6 336 MAC. Obrázek převzat z [32].

4.1.2 Winograd konvoluce

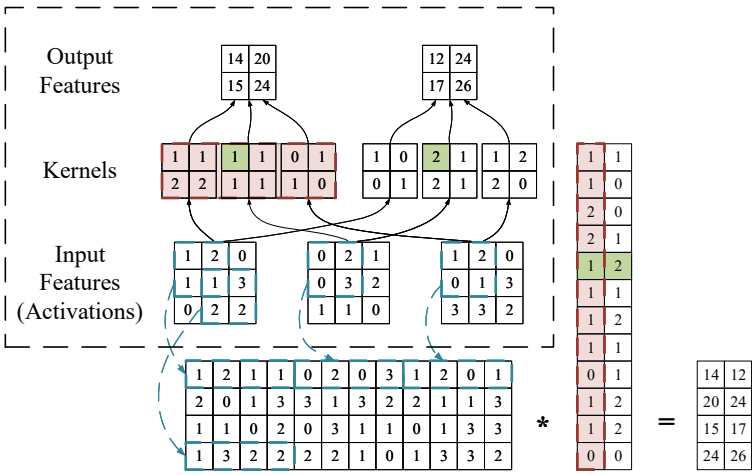
Winograd konvoluce je metoda, jež cílí na snížení počtu násobení nutných k vykonání konvoluce za cenu navýšení počtu sčítání a posuvů. Ty jsou však pro HW efektivně zpracovatelné se zanedbatelnou spotřebou energie [56]. Snížení počtu násobení je dosaženo s pomocí maticových transformací k získání jiné reprezentace vstupních dat a vah filtrů, nad kterými se následně použijí výpočetně efektivnější matematické operace jako je maticové násobení. Přestože vede aplikace Winograd konvoluce ke snížení výpočetní náročnosti, doprovází tuto metodu i chyby s pohyblivou řádovou čárkou. Informace o Winograd konvoluci jsou uvedeny v článku [1].

4.1.3 Rychlá Fourierova transformace

Algoritmus pro rychlou Fourierovu transformaci (FFT) převádí výpočet konvoluce (vstupy a váhy) z prostorové domény do domény frekvenční, kde je výpočet proveden prostřednictvím jednodušší operace násobení. Následně se na výsledek aplikuje inverzní FFT k převodu zpět do prostorové domény. FFT s sebou přináší i vyšší paměťové nároky, jelikož data ve frekvenční doméně mohou zabírat více místa než původní data.

4.1.4 General matrix multiply

General matrix multiply (GEMM) představuje koncept pro paralelní násobení dvou vstupních matic (vstupů a vah) k vyprodukování výstupní matice příznaků. Za tímto účelem jsou původní tenzory transformovány do sloupcových matic prostřednictvím přístupu *im2col* (image to column) [52]. Existuje několik implementací GEMM, včetně BLAS knihovny a podobných. Tyto implementace se snaží optimalizovat výpočty a snížit počet operací potřebných k výpočtu násobení matic, což vede k redukci složitosti celkového výpočtu a tím k urychlení inference zejména u vysoce paralelizovatelných HW architektur (temporal architektury, TPU). Obrázek 4.2 ukazuje průběh výpočtu GEMM včetně transformace im2col.



Obrázek 4.2: Ukázka principu výpočtu GEMM. Ohraničený obdélník obsahuje tradiční reprezentaci konvoluční vrstvy a její výpočet, zbylé části v obrázku zobrazují transformaci pomocí *im2col* a výpočet. Červené výplně označují hodnoty pro 1 filtr, zatímco zelené hodnoty náleží hodnotám z různých filtrů. Obrázek převzat z [32].

4.2 Komprese modelu CNN

Komprese modelu konvoluční neuronové sítě se zabývá metodami a technikami, které redukují velikost a paměťové nároky potřebné k uložení modelu. Cílem komprese je dosáhnout menších modelů, které zabírají méně paměti a jsou snadněji přenositelné. Komprese modelu je klíčová pro efektivní nasazení sítí na zařízeních s omezenými prostředky, jako jsou mobilní telefony a vestavěné systémy.

4.2.1 Knowledge distillation

Technika knowledge distillation (KD) se využívá ke kompresi modelů DNN převedením znalostí z natrénovaného velkého modelu, zvaného *učitelský* model, na menší a jednodušší model, tzv. *studentský* model. Tento proces je realizován natrénováním studentského modelu s cílem dosažení stejné aplikační úspěšnosti jako učitelský model nad danou vstupní datovou sadou. Získaný menší model má pak menší paměťové a výpočetní nároky na inferenci a přitom dosahuje stejné přesnosti jako jeho vzor. O technice knowledge distillation pojednává článek [11].

4.2.2 Low-rank faktorizace

Cílem low-rank faktorizace je rozložit velké tenzory vah na menší aproximované matice a tím snížit celkový počet parametrů. Aproximace menšími maticemi slouží k zachycení klíčových informací (hodnoty vah) v původním tenzoru. Celkově tedy dochází k rozkladu velkých vrstev na několik menších, což přispívá k jednoduššímu přístupu do paměti, komprimaci velikosti modelu a snížení počtu operací. Tato technika se také často spojuje s technikami prořezávání a kvantizace. Low-rank faktorizaci blíže popisuje článek [11].

4.2.3 Prořezávání

Prořezávání neboli pruning je důležitá technika používaná k redukci komplexity natrénovaného modelu odstraněním vah, které významným způsobem nepřispívají k výsledku. K tomu dochází pokud jsou hodnoty vah rovny 0, jsou blízko 0 a nebo jsou replikovány. Cílem je tak redukovat celkový počet parametrů a přitom zachovat původní přesnost. Tím dochází ke snížení paměťových i výpočetních nároků a vytíženosti přenosového pásma.

Pruning se provádí iterativně po skončení fáze trénování, kdy jsou postupně prořezány nejméně významné váhy, neurony či celé části infrastruktury a následně se provede dotrénování (fine-tuning) modelu k navrácení ztracené přesnosti. Větší míra redukce počtu parametrů pochází z plně propojených vrstev [49]. Techniky prořezávání lze klasifikovat na základě několika kritérií [32]:

- **strukturované** a **nestrukturované** – závisí na tom, zda jsou prořezané sítě symetrické nebo ne
- **granualiata prořezávání** – závisí na typu vrstvy a její analýze (filtry, kanály u Conv)
- **statické** a **dynamické** – podle toho jestli se provádí offline nebo za běhu

Pruning je vhodné kombinovat s dalšími technikami komprese modelu, zejména s kvantizací. V následujících podsekcích si přiblížíme jednolivé typy klasifikace prořezání.

Strukturované/Nestrukturované prořezávání

Strukturovaný pruning spočívá v rozdělení vah do skupin na základě geometrických rozměrů, výpočetní náročnosti či míře řídkosti v rámci skupiny. Metody pro strukturovaný pruning pak odstraňují celé skupiny nepotřebných vah, zatímco zachovávají strukturu sítě. Příkladem je group LASSO[1] [62], navržená k uspořádání hodnot do skupin tak, aby je bylo možno buď zcela ponechat nebo zcela prořezat. Řádková nebo sloupcová řídkost v rámci skupiny může být využita k redukci doby výpočtu GEMM.

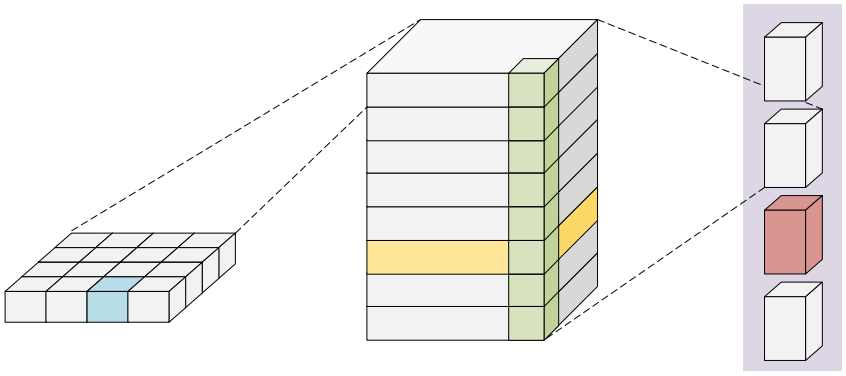
Nestrukturovaný pruning vede k maticím vah, které nelze efektivně a paralelně zpracovat. Například prořezávání po jednotlivých prvcích (element-wise) vede k nestrukturovanému prořezání.

Strukturovanému a nestrukturovanému prořezávání se podrobněji věnuje studie [32].

[1] Least Absolute Shrinkage And Selection Operator

Granualita prořezávání

Pruning se může projevit na různé úrovni granuality v závislosti na typu vrstvy a její analýzy. Pro plně propojené vrstvy se mohou prořezat buď synapse, celé neurony a nebo dokonce celé vrstvy. U konvolučních vrstev je pak možno prořezat jednotlivé váhy, celé kanály, celé vrstvy apod. Obrázek 4.3 znázorňuje možnosti prořezání u konvolučních vrstev.



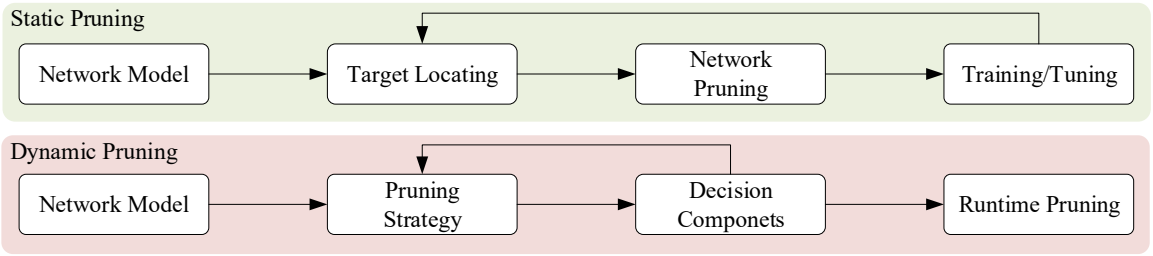
Obrázek 4.3: Granualita prořezání je úzce spojena s mírou řídkosti sítě. Zde se řídkost snižuje zleva doprava. Obrázek převzat z [32].

Statické/Dynamické prořezávání

Statický pruning se provádí po natrénování sítě a před spuštěním běhu inference. Během inference k žádnému dalšímu prořezávání nedochází. Princip funkcionality sestává ze 3 podčástí: (1) výběru parametrů k prořezání, (2) zvolení metody k prořezání, (3) volitelného dotrénování modelu. Dodatečným dotrénováním se může zvýšit výkonnost a dosáhnout přesnosti srovnatelné s původním modelem. Jeho hlavní nevýhodou je, že nevratně dochází k rozbití struktury původní sítě což může mít za následky omezené schopnosti modelu.

Dynamický pruning umožňuje za běhu trénování určit, které váhy, neurony, kanály či celé vrstvy výrazně nepřispívají k výpočtu výsledku. Dovede tak zohlednit měnící se charakter vstupních dat i aktuální stav modelu a na základě toho se rozhodovat při prořezávání. Dynamic pruning je obecně složitější na implementaci, ale může být účinnější než static pruning, protože se dovede přizpůsobit aktuálnímu stavu modelu, čímž se u něj zmenšuje riziko nevratného rozbití struktury sítě.

Obrázek 4.4 znázorňuje rozdílné operační kroky vykonávané statickým a dynamickým prořezáváním.



Obrázek 4.4: Ukázka posloupnosti operací při aplikaci statického respektive dynamického prořezávání. Inspirace pro tvorbu obrázku získána z [32].

4.2.4 Kvantizace

Kvantizace je proces aproximace spojitého signálu s využitím množiny diskrétních hodnot. Ukázala se být dobrým nástrojem pro zvýšení efektivity při trénování i inferenci neuronových sítí [16]. Existují dva základní přístupy kvantizace, které se obecně používají: kvantizace během trénování – **QAT**[2] a kvantizace po trénování – **PTQ**[3].

QAT zahrnuje proces kvantizace jako součást trénování modelu. Tím se model stává kvantizačně citlivým a je schopen se přizpůsobit a minimalizovat chyby způsobené kvantizací. QAT je vhodný pro obě fáze, trénování i inferenci, a může poskytnout modely, které jsou robustnější vůči kvantizačním chybám.

Na druhou stranu, PTQ je proces, který se aplikuje po dokončení trénování modelu. Tento přístup je obecně rychlejší, protože nevyžaduje další trénování. Avšak kvůli tomu může být méně odolný vůči chybám způsobeným kvantizací.

Princip kvantizace spočívá v reprezentaci hodnot vah a aktivací s využitím menšího počtu bitů snížením přesnosti dat a přitom zachováním přesnosti modelu. Příkladem je nahrazení 32-bitových hodnot s plovoucí desetinnou čárkou (FP32) 8-bitovými celočíselnými hodnotami (INT8) [51]. Většina sítí je trénována s využitím FP32 čísel. Výhodou FP32 je velmi dynamický rozsah hodnot, který umožňuje reprezentovat jak velmi malá čísla s vysokou přesností, tak i velmi velká čísla. Tato výhoda se ztrácí po aplikaci kvantizace na nižší bitové hodnoty, jako je INT8. Ovšem málokterá síť opravdu potřebuje tak vysokou přesnost k reprezentaci svých parametrů. Snížení přesnosti tedy vede k redukci výpočetní a paměťové komplexity a také k možnostem větší propustnosti přenosového pásma, což je výhodné zejména pro provádění inference na zařízeních s omezenými výpočetními prostředky.

Existuje několik způsobů jak namapovat data odpovídajícím kvantizovaným diskrétním hodnotám. Nejjednodušší je metoda *lineárního mapování* s uniformní vzdáleností mezi hodnotami. Další možnost je použít *logaritmické mapování*, při němž se vzdálenosti mezi hodnotami různí (vzdálenosti jsou neuniformní). Případně může být mapovací funkce přímo uzpůsobená konkrétním hodnotám dat, například s využitím k-means shlukování.

Kvantizace navíc může být uniformní napříč všemi vrstvami sítě, nebo se může různit pro každou vrstvu zvlášť (neuniformní). Tomuto způsobu kvantizace bude uzpůsobena implementace rozšíření analytického modelu pro návrh a analýzu HW architektur s vestavěnou podporou kvantizace, čemuž se věnuje kapitola 8. Navíc je na místě kombinace spolu s jinými optimalizačními technikami, zejména s prořezáváním.

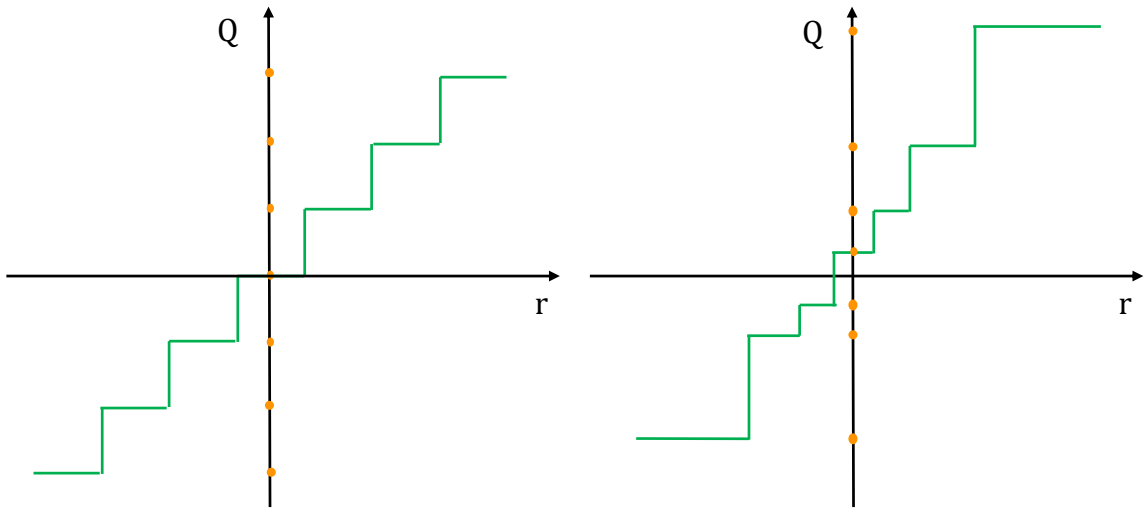
[2] Quantization aware training
[3] Post training quantization

Lineární/nelineární kvantizace

Lineární kvantizace je metoda pro redukci přesnosti vah a vstupních aktivací modelu, která spočívá v rozdělení rozsahu původních možných hodnot do intervalů s fixní uniformní velikostí jednotlivých košů. Následně jsou jednotlivé hodnoty přiřazeny k hodnotám nejbližších košů. Lineární kvantizace je formou ztrátové komprese, jelikož mohou být ztraceny některé informace a může dojít k zanesení kvantizační chyby (měřenou v SQNR[4]). Výhodou lineární kvantizace je její jednoduchá implementace jak pro váhy, tak pro aktivace. Ovšem v případě velkého rozsahu hodnot hrozí zanesení velké kvantizační chyby.

Nelineární kvantizace narozdíl od lineární kvantizace mapuje spojitý prostor na menší množinu diskrétních hodnot s pomocí neuniformní mapovací funkce. Přitom jsou vzdálenosti mezi jednotlivými koši různé. Nelineární kvantizace může poskytnou vyšší kompresní poměr a lepší přesnost než lineární kvantizace vzhledem k možnosti lépe zachytit distribuci hodnot v síti a zohlednit tak jejich přiřazení k odpovídajícím diskrétním hodnotám. Nelineární kvantizace je však výpočetně náročnější pro implementaci v HW.

Obrázek 4.5 znázorňuje lineární a nelineární mapování reálných hodnot k odpovídajícím hodnotám diskrétní množiny kvantizovaných hodnot.



Obrázek 4.5: Srovnání mezi lineární – uniformní (vlevo) a nelineární – neuniformní (vpravo) kvantizací. Reálné hodnoty ze spojité domény r jsou mapovány na diskrétní hodnoty s menší přesností v kvantizované doméně Q. Dílčí diskrétní hodnoty v kvantizované doméně jsou reprezentovány oranžovými tečkami. Vzdálenosti mezi kvantizovanými hodnotami u uniformní kvantizace jsou stejné, zatímco u neuniformní kvantizace se mohou lišit. Obrázek přejat z [16].

[4]Signal-to-quantization-noise ratio

Kapitola 5

Analytické nástroje pro odhad hardwarových parametrů a optimalizaci výkonnosti při inferenci modelu

Abychom mohli efektivně optimalizovat HW architektury pro inferenci neuronových sítí, je důležité mít k dispozici nástroje, které nám umožní analyzovat různé konfigurace a jejich vliv na spotřebu energie, latence a dalších parametrů. V současnosti existuje několik nástrojů, které umožňují automatizované prohledávání různých konfigurací HW architektur s ohledem na zvolené parametry. Některé z nich se však zaměřují jen na konkrétní typ architektur jako jsou FPGA [36, 40, 63, 60].

Timeloop+Accelergy [39, 57, 58] je framework umožňující zkoumat rozdílné HW architektury a jejich konfigurace a analyticky k nim zjistit detailní statistiky týkající se využití HW jednotek, latence, zabírané plochy na čipu, přístupů do paměti na jednotlivých úrovních paměťové hierarchie a mnoho dalšího. Framework poskytuje i kód formou Dockeru[1]. Navíc existují i návody k použití obou nástrojů[2] a jejich dokumentace[3].

MAESTRO [27] je analytický nástroj podobný Timeloopu, který se zaměřuje na evaluaci výkonnosti, energetické účinnosti, latence, využitelnosti HW a dalších parametrů odlišných datových toků pro rozdílné HW specifikace. Jeho záměrem je však především zjistit optimální HW zdroje potřebné ke zpracování daného problému. Je k němu dostupná i dokumentace[4] a vlastní webové stránky[5].

DNN chip predictor [64] se také zaměřuje na rychlé analytické odhady různých HW akcelerátorů a jejich konfigurací a dosahuje takřka stejných odhadů celkové spotřeby energie a latence jako Timeloop pro SOTA[6] ASIC akcelerátor Eyeriss [10, 9], jak je porovnáno v [29]. Nástroj však neumožňuje získat podrobnější statistiky ohledně spotřeby energie jednotlivých vrstev paměťové hierarchie či celkové využití procesních elementů.

Nástroj s názvem **Bifrost** [46] představuje framework pracující s rozličnými HW architekturami a využívající simulátor STONNE [37] pracující na úrovni cyklů (cycle-accurate)

[1] https://github.com/Accelergy-Project/accelergy-timeloop-infrastructure
[2] http://accelergy.mit.edu/tutorial.html
[3] https://timeloop.csail.mit.edu/timeloop
[4] https://maestro.ece.gatech.edu/docs/build/html/index.html
[5] https://maestro.ece.gatech.edu/
[6] State-of-the-art

k průzkumu HW konfigurací. Vzhledem k cycle-accurate simulaci je nástroj pomalejší než nástroje pracující na bázi analytických odhadů. Analytické modely však ve svých odhadech mohou selhávat u komplexních architektur, kde se výpočet neřídí pravidelnými vzory.

K hlubšímu prozkoumání a porovnání byly nakonec zvoleny nástroje Timeloop+Accelergy a MAESTRO z důvodu vysoké míry konfigurovatelnosti HW akcelerátorů, rychlému vyhodnocení parametrů pro zadané datové toky, vysoké úspěšnosti odhadů parametrů vůči post-layout simulaci, která dosahuje v průměru až 95 %, a také široké škále statistických údajů reportovaných k jednotlivým konfiguracím. Timeloop obsahuje i podporu pro prohledávání prostoru možných mapování datových toků k provedení inference na daném HW. MAESTRO žádný takový automatizovaný mapper nativně neobsahuje, ale existuje externí nástroj GAMMA [25, 26] sloužící k prohledávání prostoru možných mapování. GAMMA je založen na genetickém algoritmu a je implementovaný k zaintegrování ke spolupráci s oběma frameworky, čímž se tak nabízí jeho srovnání u obou z nich. Oba nástroje se jeví být užitečné a dobře rozšiřitelné pro účely této práce k přidání podpory pro kvantizace a k následnému zkoumaní jejich dopadu na výkonnostní parametry HW.

5.1 Timeloop+Accelergy

Framework Timeloop+Accelergy se skládá ze dvou samostatných nástrojů: i) Timeloop pro rychlé a flexibilní prohledávání prostoru různých mapování akcelerátorů. ii) Accelergy k odhadu parametrů a výkonnostních statistik jednotlivých HW konfigurací.

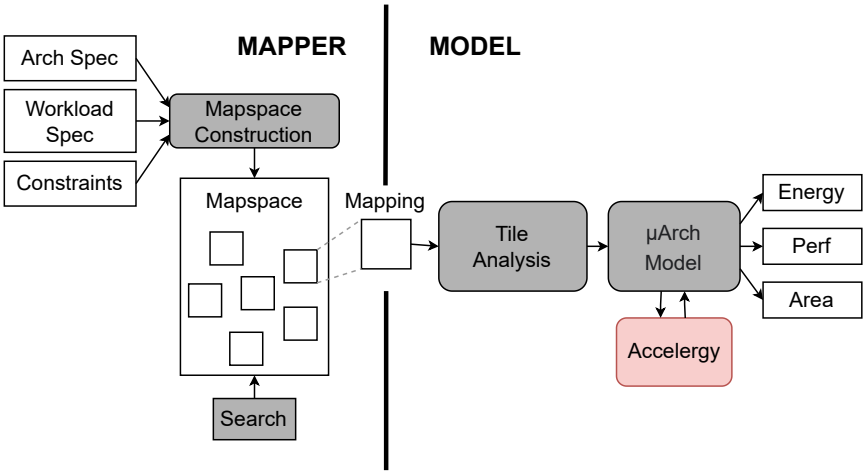
Nástroje pracují s architekturním modelem HW, což umožňuje snadnější vyhodnocení než při práci s modelem na RTL úrovni popisu (t.j., VHDL nebo Verilog). Timeloop navíc umožňuje rychlé analytické vyhodnocení jednotlivých mapování oproti náročné cycle-accurate simulaci, jelikož využívá faktu, že počet operací a vzory pro přenosy dat jsou do značné míry deterministické. Modelování plochy a energie aritmetických jednotek, paměťových modulů a sítě propojů je založeno na 16nm FinFET od TSMC. Framework je schopen zohlednit různé HW implementace a optimalizace (např. Processing-in-Memory, zero-gating PE, Row stationary, apod.). V současné verzi nástroje se optimální mapování hledají vrstvu po vrstvě s tím, že optimalizace celé sítě jsou uvažovány do budoucna[7].

5.1.1 Timeloop

Hlavní úlohou nástroje Timeloop je namapovat daný problém na odpovídající HW architekturu. Mapováním se zde rozumí způsob rozdělení dat problému do různých vrstev paměťové hierarchie tak, aby byl zohledněn charakter dat problému a samotná HW architektura (využití řídkosti, row stationarity, apod.). Nástroj pracuje ve dvou[8] režimech – **MAPPER** a **MODEL**. Vizuální reprezentaci infrastruktury nástroje lze vidět na obrázku 5.1 a obrázek 5.2 znázorňuje fungování Timeloopu v režimu modelu.

[7] Dle zmínky v článku [39] a v rámci tutoriálu dostupném na adrese: http://accelergy.mit.edu/ispass2020/2020_08_23_timeloop_accelergy_tutorial_part1.pdf.

[8] Existuje ještě 3. režim – metrics, o kterém se krátce zmiňuje sekce 7.2.



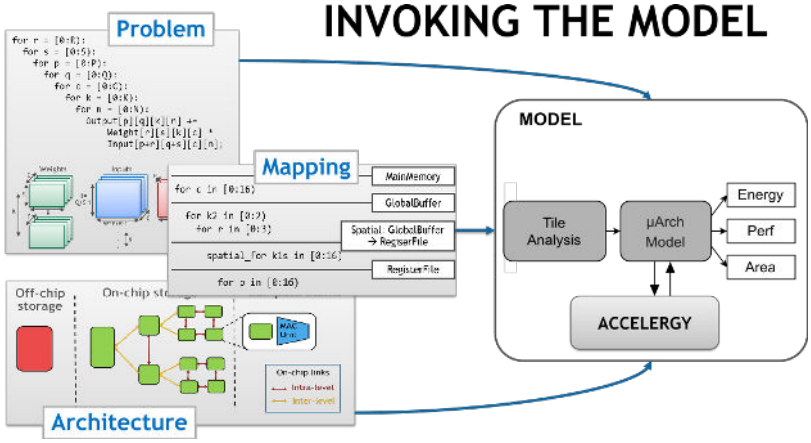
Obrázek 5.1: Tool-flow nástroje Timeloop. Původní obrázek převzat z [39]

MAPPER Pro danou architekturu a problém hledá optimální mapování vzhledem k uživatelem definovaným omezením a optimalizovaným metrikám.
Na vstupu ve formátu YAML vyžaduje:

- specifikaci **HW architektury**
- specifikaci **problému** (vrstvy DNN)
- optimalizace pro podporu zpracování řídkosti (volitelné)
- uživatelská a architekturní omezení pro možná mapování (volitelné)
- **heuristiku pro mapování** (linear, random) spolu s vybranými metrikami k optimalizaci

MODEL Pro danou architekturu a problém použije konkrétní uživatelem specifikované mapování.
Na vstupu ve formátu YAML vyžaduje:

- specifikaci **HW architektury**
- specifikaci problému (vrstvy DNN)
- optimalizace pro podporu řídkosti (volitelné)
- uživatelská a architekturní omezení pro možná mapování (volitelné)
- **mapování** (konfigurace HW), jež se má použít



Obrázek 5.2: Ukázka fungování nástroje Timeloop v režimu modelu. Převzato z tutoriálu.

5.1.2 Accelergy

Accelergy se používá k získání odhadů energetické spotřeby, přístupů do paměti či plochy jednotlivých komponent, kterých pak Timeloop využívá k vyhodnocení parametrů a celkových statistik jednotlivých mapování. Pro strukturu nástroje viz obrázek 5.3.

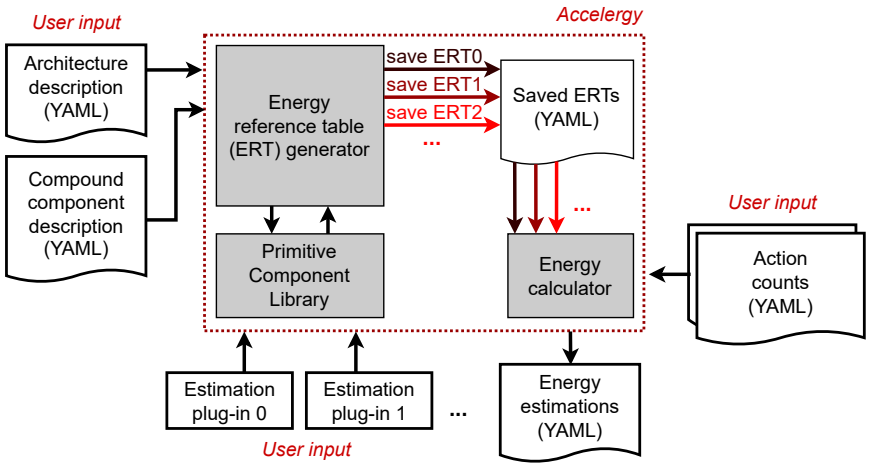
Nástroj interně generuje tzv. Energy Reference Table (**ERT**) a Area Reference Table (**ART**), které poskytují odhady pro spotřebu energie respektive plochy zabírané jednotlivými komponentami použitými v dané HW architektuře. Pro získání odhadů Accelergy využívá externích nástrojů pro odhady (estimation plug-ins) jako jsou např. Aladdin [43] a Cacti [30], jež jsou přímo integrovány do frameworku. Uživatel si však libovolně může přidat vlastní nástroje.

Mimo ERT a ART Accelergy k jednotlivým komponentám navíc potřebuje vědět informaci o počtu provedených akcí (**access counts**). Aby odhady odpovídaly reálnému chování HW, uvažuje se tzv. fine-grained granualita akcí. Ta akcím přiřazuje určité vlastnosti, které je odlišují jak principem chování, tak cenou z pohledu spotřeby energie. Příkladem může být rozdíl mezi akcemi čtení z SRAM v případě čtení náhodných dat z náhodné adresy a při čtení stejných dat ze stejné adresy. Počet akcí Accelergy získává z performance modelu (např. z Timeloopu), což může být analytický či cycle-accurate model.

Uživatel si může vytvořit vlastní ERT, ART nebo access counts a předat je přímo na vstup Timeloopu+Accelergy, který je interně zohlední a nebude je nahrazovat soubory generovanými z Accelergy. V opačném případě se interně využije Accelergy k získání potřebných souborů.

ACCELERGY na vstupu vyžaduje:

- specifikaci modelu architektury a jejích jednotlivých komponent (YAML)
- externí nástroje pro odhad parametrů komponent (volitelně)
- referenční počty akcí jednotlivých komponent (volitelně)

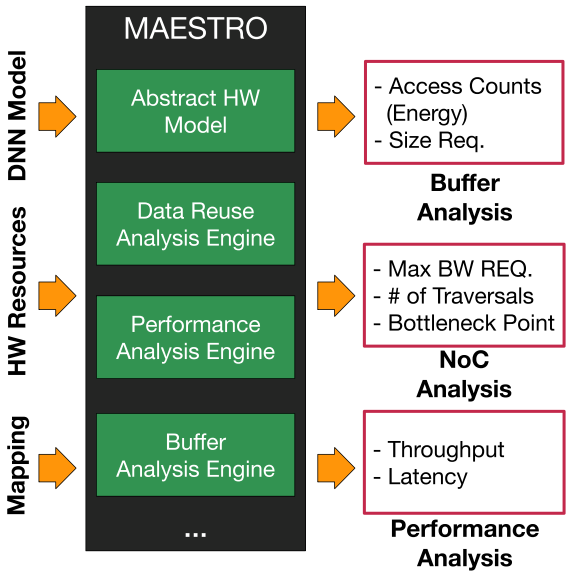


Obrázek 5.3: Block level diagram frameworku Accelergy. Původní obrázek převzat z [57]

5.2 MAESTRO

MAESTRO pracuje s popisem modelu DNN, seznamem HW zdrojů a konkrétním mapováním v data-centrické reprezentaci. V nativní implementaci umožňuje pouze evaluaci ručně předvytvořených mapování a v podstatě tak funkcionalitou odpovídá Timeloop modelu.

Místo hierarchického popisu HW přijímá pouze seznam HW zdrojů jako jsou počet PE, velikosti pamětí pro jednotlivé úrovně paměťové hierarchie, šířku přenosového pásma nebo třeba volbu podpory pro multicasting. Popis mapování je v data-centrické reprezentaci, což umožňuje popsat všechna možná mapování v stručném a kompilátoru přívětivém formátu. Obrázek 5.4 znázorňuje princip fungování nástroje MAESTRO.



Obrázek 5.4: Tool-flow nástroje MAESTRO. Původní obrázek převzat z [27]

Díky práci na architekturní úrovni popisu a konzistentnímu popisu mapování je dosaženo rychlého analytického odhadu více než 20 výkonnostních statistik jako jsou energie, latence, propustnost. Nástroj dovede v průměru dosáhnout až 96% přesnosti odhadu oproti cycle-accurate simulaci. MAESTRO analyticky počítá počet přístupů do paměti a s pomocí Cacti následně určuje odhad celkové spotřeby energie spojené s přístupy do paměti. Při analýze vstupní sítě umožňuje zohlednit různé typy vrstev a jejich optimalizací (1D a 2D konvoluce, Plně propojené vrstvy (FC), Depth-wise separable konvoluce, General Matrix Multiply (GEMM)). Navíc zohledňuje i mezivrstevní optimalizace, tudíž není třeba analyzovat jednotlivé vrstvy zvlášť.

MAESTRO pro zadané HW zdroje a mapování modelu DNN (případně jen vrstvy) vyhodnotí dosažené metriky.
Vstupní parametry:

- seznam HW zdrojů
- specifikace mapování (datového toku) aplikovaného na danou síť, případně vrstvu
- volitelné přepínače pro výpis naměřených statistik a logů

Popis DNN a konkrétní zvolený datový tok je spojen do jednoho souboru specifikující mapování. MAESTRO umožňuje volbu jednoho z 6 předdefinovaných datových toků s možností nadefinovat si vlastní. Některé z nich jsou specifické pro konkrétní typy akcelerátorů jako je Row stationary pro Eyeriss nebo flexibilní datový tok spojený s MAERI akcelerátorem [28]. Propojení modelu DNN s datovým tokem lze provést s využitím předpřipravených skriptů. Ty umožňují si nejprve s pomocí PyTorch či TensorFlow API[9] vygenerovat reprezentaci modelu a k němu následně přiřadit datový tok k obdržení výsledného mapování. Již zmíněná data-centrická reprezentace datového toku a výsledného mapování sestává z použití direktiv *TemporalMap*, *SpatialMap*, které umožňují zohlednit časové a prostorové vlastnosti dat k maximalizaci znovupoužití, jež jsou specifické pro konkrétní datový tok a direktivy *Cluster* pro určení hierarchické organizace procesních jednotek, což v rámci definice mapování umožňuje prozkoumat více paralelních dimenzí zpracovávaného problému.

Význam direktiv pro mapování je následující:

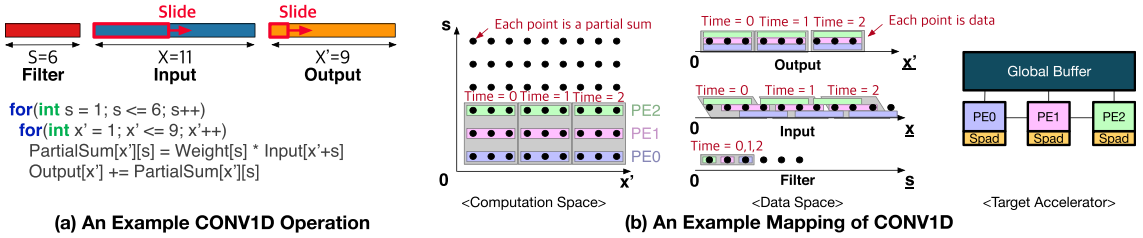
TemporalMap popisuje mapování dat, které se mění v čase (uvnitř PEs). Odpovídá běžné smyčce *for* ve vnořené smyčce zpracování dimenzí sítě.

SpatialMap popisuje mapování dat, které se mění v prostoru (uvnitř PEs). Odpovídá *parallel for* smyčce ve vnořené smyčce zpracování.

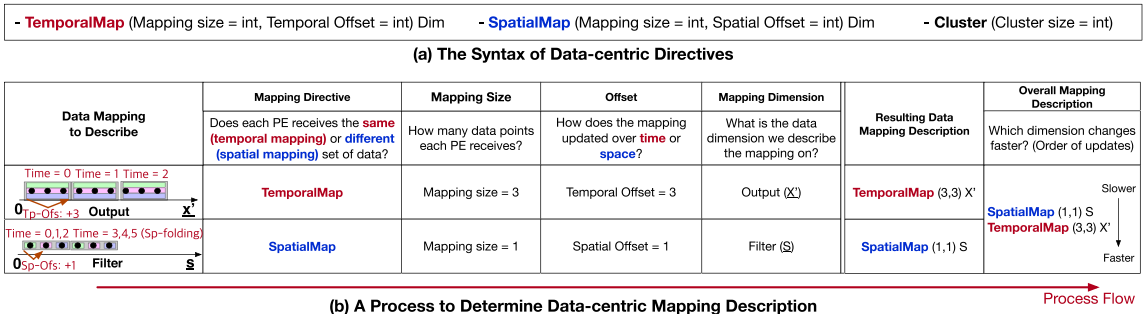
Cluster direktiva specifikuje hierarchickou organizaci PEs, čímž se umožňují prozkoumat větší možnosti paralelizace.

Ukázku výpočtu 1D konvoluce na jednoduché architektuře ukazuje obrázek 5.5 a následně obrázek 5.6 znázorňuje aplikaci direktiv k získání mapování.

[9] Application Programming Interface



Obrázek 5.5: Ukázka 1D konvoluce a mapování na ukázkovém akcelerátoru. Mapování je znázorněno ve výpočetním i datovém prostoru, kde každý bod odpovídá částečnému součtu respektive datu. Obrázek převzat z [27].



Obrázek 5.6: Představení direktiv používaných k definici mapování. (a) Syntaxe spojená s jednotlivými třemi direktivami. (b) Sémantika dvou mapovacích direktiv postavených na příkladu 1D konvoluce z obrázku 5.5. Obrázek převzat z [27].

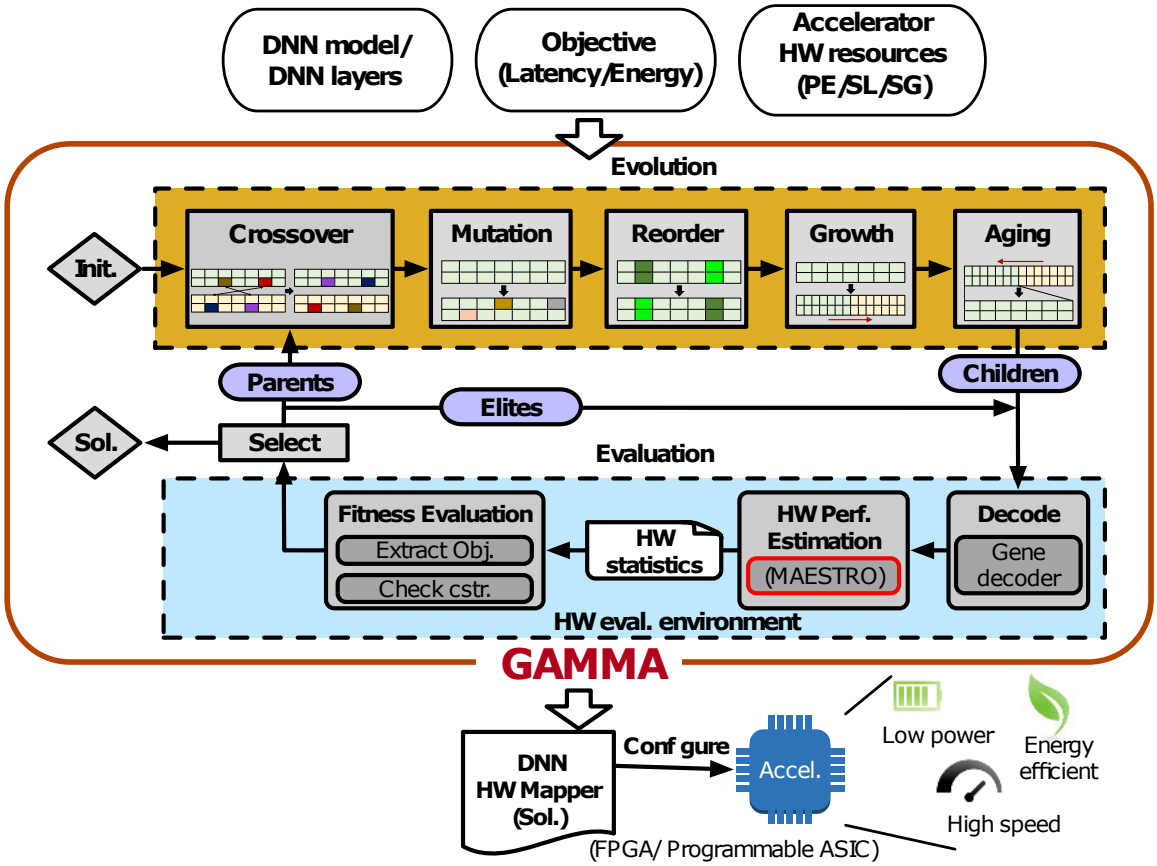
5.3 Nástroj GAMMA rozšiřující Timeloop+Accelergy a MAESTRO k prohledávání prostoru mapování

Zatímco Timeloop v sobě integruje několik heuristik používaných k prohledávání prostoru možných mapování, MAESTRO nativně žádné neobsahuje. Navíc Timeloop se spoléhá na náhodné a lineární prohledávání a dosahuje vysoké efektivity zejména díky paralelizmu užitím více procesorových vláken. GAMMA představuje specializovaný genetický algoritmus, který je ve spolupráci s analytickým modelem schopen prohledávat celý prostor možných mapování.

GAMMA zohledňuje všechny tři klíčové aspekty spojené s HW mapováním: (i) strategii rozdělení dimenzí vstupních tenzorů. (ii) změnu pořadí provedení výpočtu. (iii) paralelizaci výpočtu. Díky tomu je GAMMA schopný prozkoumat až 3 úrovně paralelizmu, umožňující mu hledat mapování i pro akcelerátory s více čipy jako je Simba [42].
Na rozdíl od klasického genetického algoritmu **GAMMA** specializovaně implementuje:

- zakódování všech tří aspektů mapování
- operátory křížení a mutace pro tvorbu nových mapování
- genetické operátory (*Reorder*, *Growth*, *Aging*) k modelování míry paralelizmu

Existují dvě různé implementace GAMMA, které spolupracují se zmiňovanými analytickými nástroji pro prohledávání mapování a jejich vyhodnocení. Obecná struktura takového frameworku je znázorněna na obrázku 5.7. Navíc existuje ještě nástroj DiGamma [26], který dále rozšiřuje funkcionalitu GAMMA o možnost co-optimalizace vůči potřebným HW zdrojům a ideálnímu mapování pro vstupní model DNN.



Obrázek 5.7: Interní struktura frameworku používající geneticky algoritmus GAMMA k prohledávání HW konfigurací akcelerátorů pro dané vstupní specifikace. Framework do sebe integruje i externí analytický model k odhadu výkonnostních parametrů nalezených konfigurací, sloužící jako fitness hodnota určující kvalitu nalezeného řešení. MAESTRO lze v takovémto případě nahradit i Timeloop+Accelergy.

Kapitola 6

Experimentální porovnání Timeloop+Accelergy a Maestro

Následuje experimentální porovnání nástrojů Timeloop+Accelergy a MAESTRO z hlediska možnosti jejich využití pro scénáře NAS[1] (HW-NAS[2]) či modelování HW akcelerátorů zohledňující pokročilé optimalizace modelu jako je kvantizace. Na základě porovnání pak bude zvolen vhodnější nástroj pro rozšíření o podporu kvantizace. Porovnání vlastností těchto nástrojů není k dispozici, a proto provedeme vlastní.

6.1 Návrh experimentů

V této sekci se zaměříme na popis návrhu experimentů pro oba nástroje. Cílem experimentů je poskytnout hlubší porozumění o flexibilitě, konfigurovatelnosti a výkonnosti obou nástrojů. Zajímat nás bude jejich dosažená výkonnost z hlediska doby běhu a odhadu parametrů. Toho bude docíleno prostřednictvím srovnání možných optimalizovaných metrik, dosažených hodnot a výkonnosti běhu. Dále bude předmětem experimentů hodnocení míry konfigurovatelnosti HW akcelerátoru pro modelování v rámci daných nástrojů, včetně možnosti volby bitové šířky slova či technologie výrobního procesu, což nám poskytne lepší představu o tom, jak dobře se každý nástroj může přizpůsobit specifickým požadavkům, které by následně umožnily zkoumat dopad různých nastavení kvantizace datových tenzorů na běh inference.

Přímé porovnání obou nástrojů není jednoduché vzhledem k různé úrovni granuality popisu vstupního HW a odlišného formátu pro popis mapování, čímž se definování stejného a dobře porovnatelného scénáře stává problematičtějším. Navíc se zdá, že MAESTRO není flexibilní vůči volbě bitové šířky dat (ani uniformní) či technologie výrobního procesu a energetické odhady, které podle článku čerpá z nástroje Cacti, jsou pevně definovány v jednom ze zdrojových souborů, odkud se pro vyhodnocení výkonnostních parametrů přečtou a vynásobí příslušným počtem akcí (počet čtení z paměti, MAC operace, apod.). Oba nástroje budou tudíž testovány separátně. U obou nástrojů však nativně chybí možnost specifikace různé bitové šířky datových tenzorů (kvantizace).

Testovanou HW architekturou bude ve všech experimentech akcelerátor, jehož struktura vychází z projektu Eyeriss [9]. Zkoumanými sítěmi budou AlexNet s 5 konvolučními vrstvami

[1] Neural Architecture Search
[2] Hardware-Aware Neural Architecture Search

a VGG16 s 13 konvolučními vrstvami. Všechna měření byla provedena na procesoru AMD Ryzen 9 5900X @3.7GHz, 12 jader, 24 vláken a s 16GB paměti RAM.

6.2 Timeloop+Accelergy: experimenty

Předmětem testování frameworku Timeloop+Accelergy bylo zkoumání flexibility a výkonnosti heuristik pro prohledávání prostoru mapování spolu s jejich schopností nalézt nejlepší řešení vůči optimalizovaným metrikám. Také bylo snahou identifikovat možnosti nástroje pro obezřetné interpretování reportovaných výsledků vedoucí k lepšímu pochopení nástroje a detekování možností jeho rozšíření.

Experimenty 6.2.1 a 6.2.2 se věnují srovnání výkonnosti heuristik k hledání optimálního mapování, experiment 6.2.3 testuje flexibilitu Accelergy pro odhad parametrů HW architektury definované pro různé technologie výrobního procesu a nakonec experiment 6.2.4 zkoumá vliv změny datového typu operandů na celkové výkonnostní parametry.

V rámci experimentů byla použita tzv. *Eyeriss-like* architektura s parametry uvedenými v tabulce 6.1 pokud nebude u konkrétního scénáře specifikováno jinak. Všechny nativní Timeloop heuristiky (kromě exhaustive) používaly nastavení **timeout** = 15000, **victory-condition** = 500, **max-permutations-per-if-visit** = 16, pro více detailů viz dokumentace.

Tabulka 6.1: HW specifikace Eyeriss-like architektury používané v rámci Timeloop experimentů. Specifikace je založena na článku [10]. Použitá architektura modifikuje velikost Shared global bufferu z původních 108 kB na 100 kB, jelikož rozdílových 8 kB podle článku stejně není využito datovými toky a i ukázkové definice v rámci Timeloopu obsahují pouze 100 kB. Také byla zvolena 45nm technologie místo 65nm vzhledem k větší flexibilitě u Open-source nástrojů pro odhady.

Architecture	Eyeriss-like
Technology	45nm
Number of PEs	168
Shared Buffer (SRAM)	100 kB
Weight scratchpad (SRAM)	448 B
Ifmap scratchpad (RF)	24 B
Psum scratchpad (RF)	48 B
Word-bits	16
Data type	int

6.2.1 Srovnání výkonnosti GAMMA a nativních heuristik mapování pro zjednodušenou HW architekturu

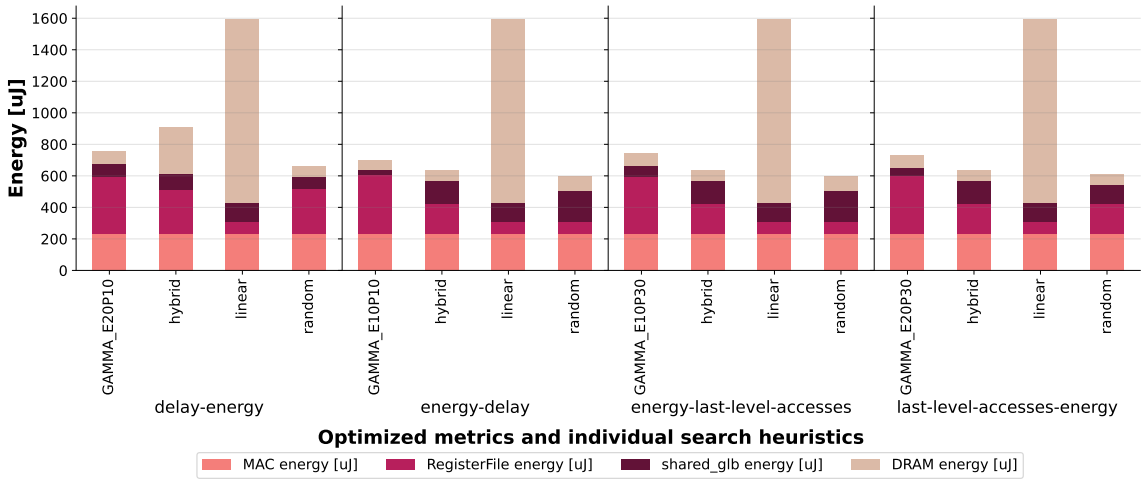
V rámci tohoto experimentu byly srovnány tři nativní Timeloop heuristiky: *linear-pruned*, *random-pruned* a *hybrid* oproti optimalizaci s využitím genetických algoritmů GAMMA [25]. Navíc byly uvažovány 4 dvojice (primární, sekundární) optimalizovaných metrik a GAMMA byla testována v 9 různých konfiguracích nastavení počtu generací a velikosti populace. Všechny heuristiky využívaly paralelizmu 24 vláken k prohledávání prostoru mapování.

Vzhledem k tomu, že GAMMA nedovoluje zohlednit všechny volitelné vstupní soubory jinak podporované Timeloopem jako jsou zejména omezení spojená s interním fungováním

HW (konkrétně pro správnou interpretaci Row stationary dataflow), případně i omezení vedoucí ke zmenšení celého prostoru mapování, rozhodli jsme se v rámci experimentu nepoužít tyto volitelné soubory i pro ostatní heuristiky. Zmenšení prostoru mapování by pro GAMMA neměl být problém, ale nemožnost plně definovat vstupní architekturu mu velmi ubírá na flexibilitě.

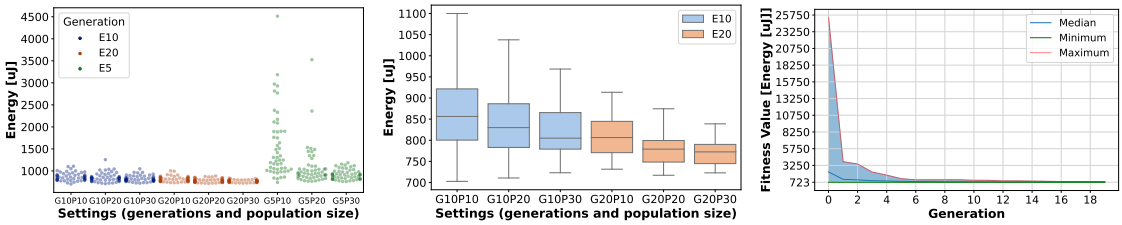
K zachování stejných podmínek pro všechny heuristiky byla architektura upravena a mapovací prostor neomezen. Úprava původní architektury (popsané v tabulce 6.1) sestává z nahrazení tří lokálních scratchpad v PE do jednoho bufferu SRAM. Všechny prezentované výsledky souvisejí se zpracováním první konvoluční vrstvy sítě AlexNet.

Obrázek 6.1 zobrazuje spotřebu energie spojenou s nejlepším mapováním nalezeným pro daný typ heuristiky a optimalizované metriky. Je patrné, že omezení mapovacího prostoru zejména pro linear-pruned způsobilo, že všechna vlákna skončila dříve, než stihla najít rozumné řešení. Změna ukončovacích podmínek prohledávání by tento problém měla vyřešit, avšak vzhledem k neomezenému prostoru mapování by běh trval příliš dlouho. Dále se ukazuje, že random-pruned heuristika z provedených experimentů a s danými nastaveními dosahuje nejlepších výsledků. Volba optimalizovaných metrik má taktéž dopad na výsledné parametry.



Obrázek 6.1: Srovnání heuristik pro hledání vhodného mapování pro různé dvojice optimalizovaných metrik. Na x-ové osy jsou horizontálními štítky označeny optimalizované metriky a vertikálně pak jednotlivé použité heuristiky. Pro GAMMA je uvedena konfigurace, která nalezla nejlepší řešení. Všechna měření jsou porovnávána vůči odhadu celkové spotřeby energie.

Na obrázku 6.2 je pak vidět výkonnost dosažená z 50 běhů pro každou z 9 různých konfigurací GAMMA, tedy celkem z 450 měření, pro metriky energie a zpoždění. Z grafů 6.2(a) a 6.2(b) lze pozorovat, že největší stability výsledků napříč všemi běhy dosahují konfigurace s největším počtem evaluací, čímž je patrný trend, že pro větší počet evaluací roste stabilita nalezených výsledků napříč běhy a zvýšením těchto parametrů by patrně bylo možné dosáhnout ještě lepších výsledků. V tabulce 6.2 pak lze pozorovat detailnější srovnání nejlepších výsledků z jednotlivých konfigurací a také celkové doby všech běhů. Konfigurace s 10 generacemi byly schopné nalézt globálně nejlepší výsledky, avšak rozptyl mezi hodnotami jednotlivých běhů je větší než je tomu pro 20 generací jak napovídají výsledky z obrázku 6.2.



(a) (b) (c)

Obrázek 6.2: Srovnání 450 běhů napříč 9 GAMMA konfiguracemi optimalizovaných vůči energii a zpoždění s využitím nástroje Timeloop+Accelergy. (a) Rozptýlení nalezených hodnot z běhů v rámci konfigurace. (b) Přiblížení na rozložení odhadů spotřeby energie nalezených mapování pro počet generací 10 a 20. (c) Konvergenční křivka pro 50 nezávislých běhů konfigurace s 20 generacemi a počtem populace 30.

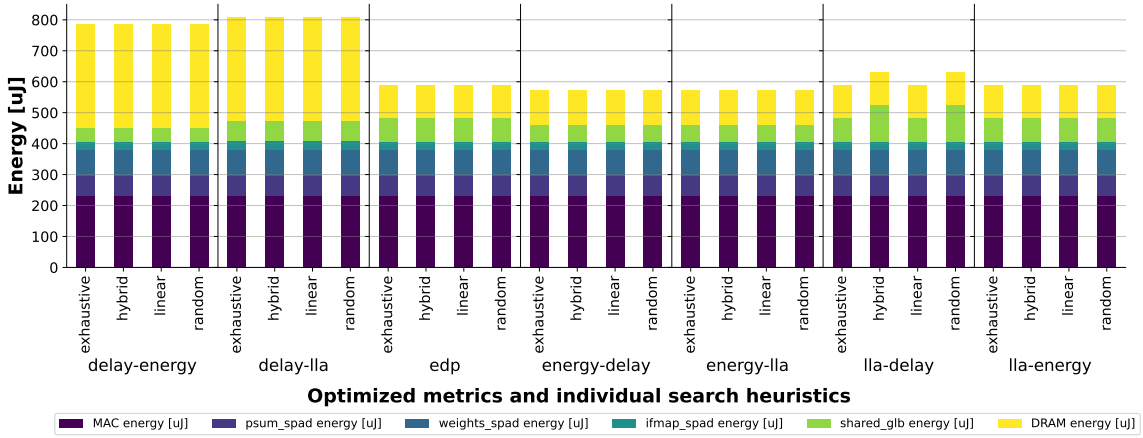
Tabulka 6.2: Doba běhu měření a odhady energie a latence pro nejlepší nalezené mapování z 450 běhů napříč 9 konfiguracemi GAMMA optimalizovanými vůči energii a zpoždění s využitím nástroje Timeloop+Accelergy.

Mapper	Metrics	Runs	Generations	Population size	Energy [μJ]	Cycles	Total runtime [s]	Average runtime [s]
GAMMA	Energy-Delay	50	5	10	739.258	52 562 400	488	~ 9.76
GAMMA	Energy-Delay	50	5	20	766.007	39 494 400	540	~ 10.8
GAMMA	Energy-Delay	50	5	30	759.133	14 361 600	641	~ 12.82
GAMMA	Energy-Delay	50	10	10	**702.738**	14 361 600	658	~ 13.16
GAMMA	Energy-Delay	50	10	20	710.754	14 361 600	677	~ 13.54
GAMMA	Energy-Delay	50	10	30	723.122	14 361 600	975	~ 19.50
GAMMA	Energy-Delay	50	20	10	731.544	14 361 600	900	~ 18.00
GAMMA	Energy-Delay	50	20	20	717.165	14 361 600	1 289	~ 25.78
GAMMA	Energy-Delay	50	20	30	722.867	**13 140 600**	1 601	~ 32.02

6.2.2 Srovnání výkonnosti nativních heuristik mapování pro Eyeriss-like architekturu

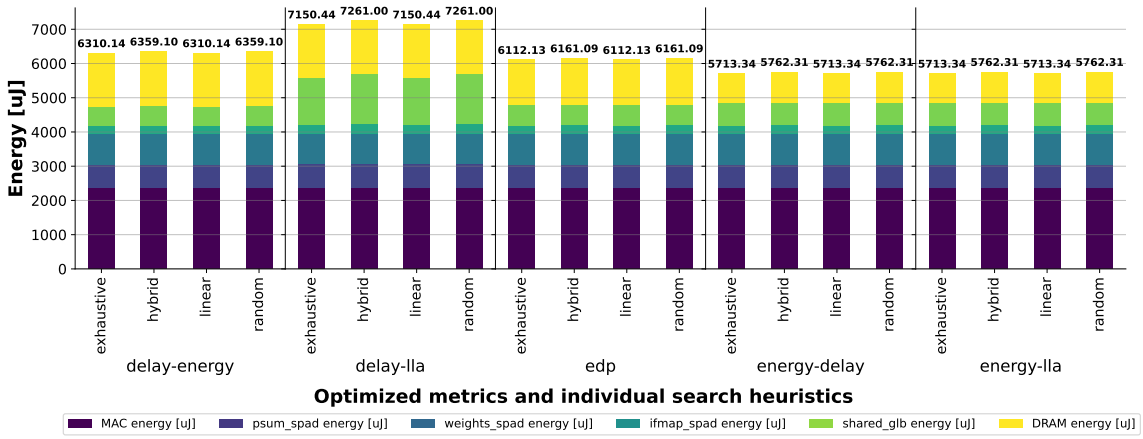
Cílem tohoto experimentu je porovnat nativní Timeloop mappery pro plně specifikovanou Eyeriss architekturu, umožňující zohlednit HW omezení spojené s modelováním Row stationary dataflow a také omezení velikosti mapovacího prostoru s cílem průzkumu pouze validních mapování.

Na obrázku 6.3 je ukázáno srovnání tří nativních heuristik a exhaustive metody pro hledání optimálního mapování pro celkem 7 optimalizovaných metrik. Na rozdíl od předešlého experimentu je linear-pruned heuristika schopna nalézt kvalitní řešení.



Obrázek 6.3: Srovnání heuristik pro hledání vhodného mapování pro různé dvojice optimalizovaných metrik u první konvoluční sítě AlexNet. Všechna měření jsou porovnávána vůči odhadu celkové spotřeby energie. Metrika *lla* odpovídá *last-level-accesses*.

Celkovou spotřebu sítě napříč 5 konvolučními vrstvami AlexNetu pro 5 vybraných metrik obsahuje obrázek 6.4. Doby běhu pro metriku energie a zpoždění a jednotlivé heuristiky obsahuje tabulka 6.3. Linear-pruned zde oproti předcházejícímu experimentu dosahuje nejlepšího řešení shodného s tím nalezeným pomocí vyčerpávající heuristiky. Random a hybrid heuristiky pro daný běh nenašly nejlepší mapování, ale odchylka není příliš malá. Je třeba zdůraznit, že výkonnost a rychlost je závislá na počtu použitých vláken a nastavených parametrech pro prohledávání.



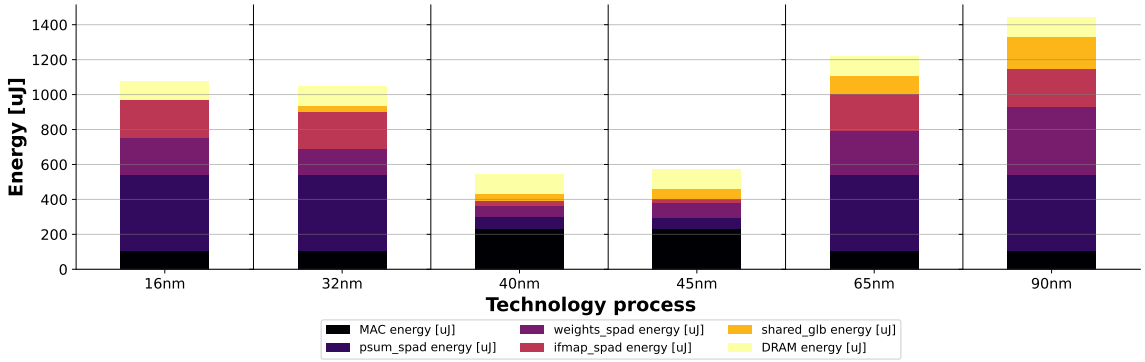
Obrázek 6.4: Celková spotřeba energie pro 5 vybraných optimalizovaných metrik a 5 konvolučních vrstev sítě AlexNet. Metrika *lla* odpovídá *last-level-accesses*.

Tabulka 6.3: Časy pro nalezení optimálního mapování s ohledem na energii a zpoždění pro 4 heuristiky a 5 konvolučních vrstev sítě AlexNet.

Mapper	Energy $[\mu J]$	Last Level Acessess	Runtime $[s]$
Exhaustive	**5 713.345**	**6 781 969**	116
Hybrid	5 762.307	7 041 169	65
Linear-pruned	**5 713.345**	**6 781 969**	44
Random-pruned	5 762.307	7 041 169	66

6.2.3 Srovnání flexibility Accelergy pro modelování architektur s různou technologií výrobního procesu

Následuje experiment zkoumající flexibilitu Accelergy při změně technologie výrobního procesu. Z obrázku 6.5 je patrné, že Accelergy dovede nejlépe zohlednit odhady spojené se 40nm a 45nm techonologií. To je způsobeno tím, že externí open-source nástroje Aladdin a Cacti, kterých nativně Accelergy využívá pro odhady, mají největší flexibilitu právě pro tyto technologie. Cacti dovede zohlednit spotřebu paměti i pro jiné technologie, ale jak lze z grafu vidět, Accelergy nemá k dispozici informace o spotřebě energie spojené s MAC operacemi pro ostatní technologie. To má za následek dosazení hodnoty $1pJ$/MAC operaci.



Obrázek 6.5: Použití různých technologií výrobního procesu pro stejnou HW architekturu na první konvoluční vrstvu sítě AlexNet.

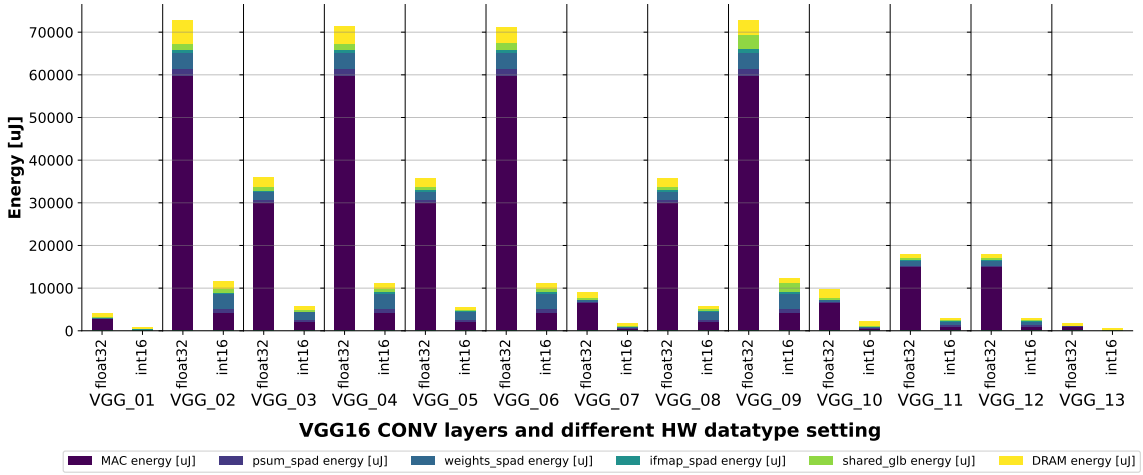
6.2.4 Změna datového typu a přesnosti zpracovávaných dat

Posledním testovaným scénářem bylo porovnání odhadu parametrů pro dvě instance Eyeriss architektury, z nichž jedna pro zpracování uvažuje 16bitové celočíselné operandy a druhá 32bitové hodnoty s pohyblivou řádovou čárkou.

První architektura je zcela nezměněna oproti specifikaci v tabulce 6.1. V případě druhé byly upraveny hodnoty šířky slova v paměti (*word-bits* na 32) a adekvátní přepočty pro počet slov na jednom řádku v paměti (platí vztah $memory\_width = block\text{-}size * word\text{-}bits$). Nakonec pak byla změněna MAC jednotka na *fpmac* (a s tím spojený *datawidth* na 32).

Obrázek 6.6 nabízí srovnání dosažených odhadů pro optimalizaci vůči energy-delay metrikám. Je zřejmé, že přestože byly zachovány stejné velikosti pamětí a šířky přenosových pásem, je spotřeba spojená s výpočtem dat pracující s plovoucí desetinnou čárkou v tomto případě mnohem energeticky náročnější. Nabízí se tedy optimalizace interního fungování

pole PE a také analýza, zda využitelnost PE není brzděna šířkou přenosového pásma (*bandwidth*).



Obrázek 6.6: Srovnání dvou Eyeriss-like architektur pracujících s různou uniformní bitovou šířkou a přesností operandů. Na x-ové ose jsou horizontálně označeny jednotlivé konvoluční vrstvy sítě VGG16 a vertikálně pak dosažené odhady pro zvolený datový typ v rámci architektury.

6.3 MAESTRO: experimenty

V rámci testování nástroje MAESTRO nebylo předmětem experimentování se samotným modelem určeným k vyhodnocení konkrétních datových toků pro dané HW zdroje, ale místo toho byl proveden pouze experiment 6.3.1 založený na použití GAMMA k automatickému prohledávání mapování pro vstupní HW zdroje. Použitý seznam HW zdrojů se skládá z:

$$
\begin{aligned}
NumPEs &= 168 \\
L2Size &= 1\,024\,000 \\
L1Size &= 5\,120 \\
NoC\_BW &= 1\,000\,000\,000
\end{aligned}
$$

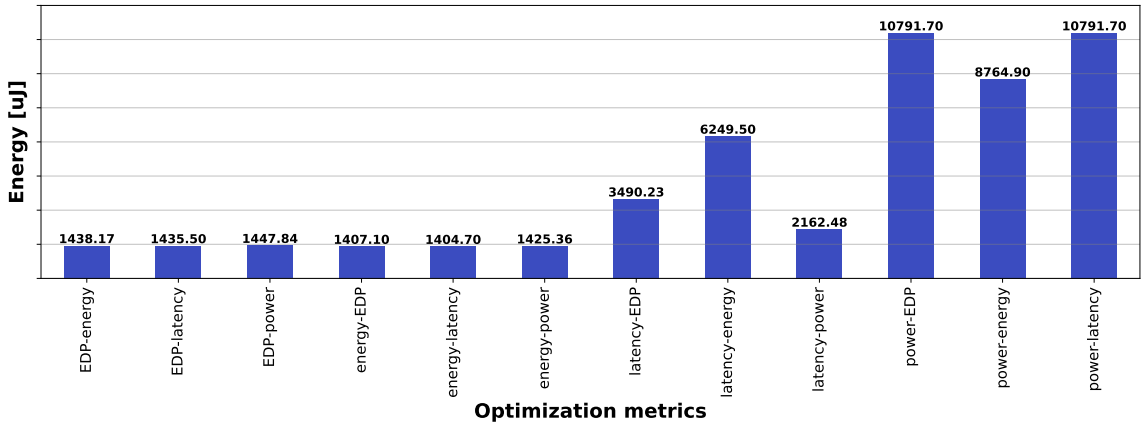
Původně bylo záměrem použít nastavení hodnot L1, L2 a NoC_BW tak, aby umožnily částečně modelovat Eyeriss architekturu, ovšem algoritmus GAMMA nebyl schopen pro různé nastavení konfigurací a vstupní síť AlexNet inicializovat počáteční populaci. Z tohoto důvodu byly vstupní HW zdroje uměle navýšeny. Na vstup algoritmu byly navíc dodány i specifikace pro HW omezení a omezení pro mapování, aby bylo zohledněno modelování Row stationary dataflow. L1 a L2 je patrně v Bytech, ale v dokumentaci o tom není zmínka.

6.3.1 Výkonnost GAMMA algoritmu pro prohledávání prostoru mapování

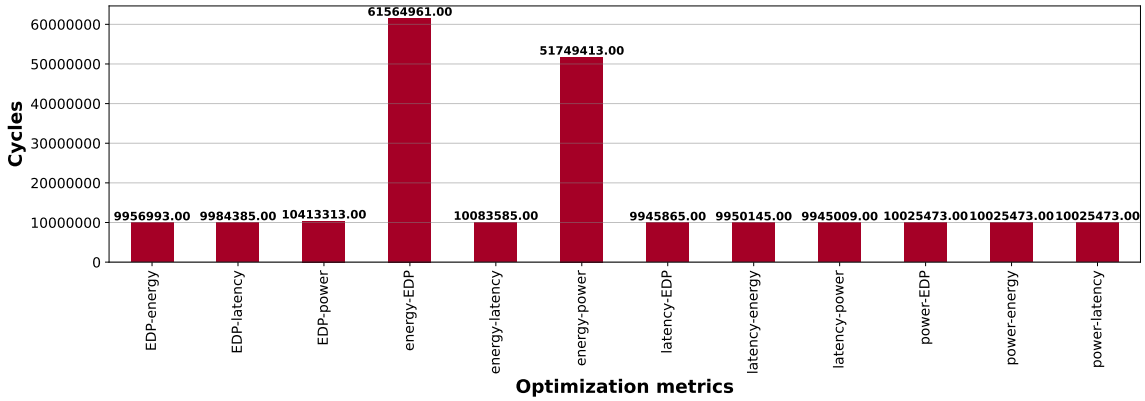
Předmětem experimentu bylo podobně jako v případě Timeloop experimentu 6.2.1 porovnání 9 konfigurací nastavení GAMMA algoritmu pro optimalizaci celkem 12 optimalizovaných metrik.

Obrázek 6.7 obsahuje srovnání energetické spotřeby dosažené z běhu různých konfigurací vůči rozličným optimalizovaným metrikám. Podle očekávání dopadly nejlépe všechny

metriky, které měly jako primární optimalizovanou hodnotu energii nebo EDP[3]. Naopak obrázek 6.8 srovnává stejné běhy vůči nejlepším dosaženým hodnotám počtu cyklů potřebných pro zpracování. Zde naopak dopadly lépe metriky optimalizující primárně vůči latenci.



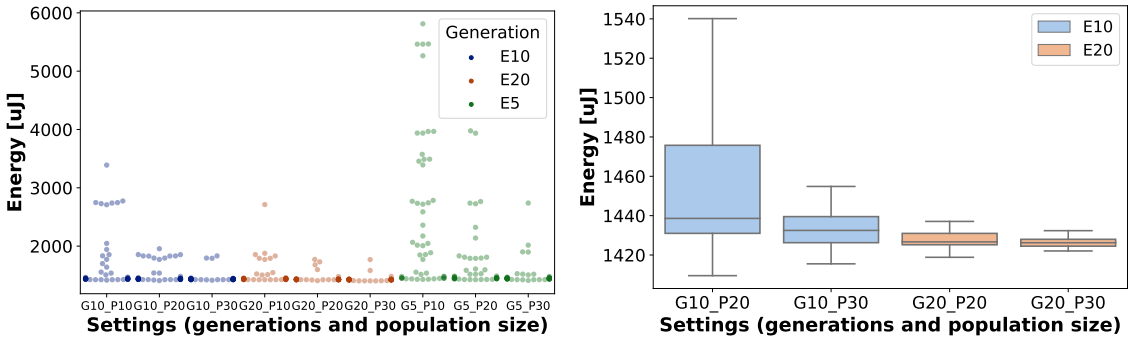
Obrázek 6.7: Srovnání celkové spotřeby energie spojené s nejlepšími mapováními nalezenými pro odlišné cílové optimalizované parametry pro první konvoluční vrstvu sítě AlexNet.



Obrázek 6.8: Srovnání odhadů počtu cyklů pro první konvoluční vrstvu AlexNet sítě pro nejlepší mapování získané z optimalizace vůči odlišným cílovým parametrům.

Srovnání výkonnosti pro optimalizaci vůči energii a latenci mezi 9 rozličnými konfiguracemi GAMMA z celkem 450 běhů obsahuje obrázek 6.9. Podobně jako v případě Timeloop experimentu 6.2.1 lze z grafů 6.9(a) a 6.9(b) pozorovat, že největší stability výsledků napříč všemi běhy dosahují konfigurace s největším počtem evaluací. Konvergenční křivka není pro srovnání poskytnuta vzhledem k tomu, že nebylo jednoduché získat statistiky mezivýsledků z jednotlivých generací běhů. Tabulka 6.4 poskytuje detailní srovnání nejlepších odhadů spotřeby energie a latence, a také celkové doby měření 50 běhů jednotlivých konfigurací.

[3]energy-delay-product



(a) (b)

Obrázek 6.9: Srovnání 450 běhů napříč 9 GAMMA konfiguracemi optimalizovaných vůči energii a latenci s využitím nástroje MAESTRO pro první konvoluční vrstvu AlexNet. (a) Rozptýlení nejlepších nalezených hodnot z běhů v rámci konfigurace. (b) Přiblížení na rozložení nalezených mapování pro počet generací 10 a 20.

Tabulka 6.4: Doba běhu měření a odhady energie a latence pro nejlepší nalezené mapování z 450 běhů napříč 9 konfiguracemi GAMMA optimalizovanými vůči energii a latenci s využitím nástroje MAESTRO pro první konvoluční vrstvu AlexNet. Konfigurace s 20 generacemi byly schopné nalézt globálně nejlepší výsledky pro obě cílové metriky.

Mapper	Metrics	Runs	Generations	Population size	Energy [μJ]	Cycles	Total runtime [s]	Average runtime [s]
GAMMA	Energy-Delay	50	5	10	1 429.30	55 757 121	114	~ 2.28
GAMMA	Energy-Delay	50	5	20	1 424.28	55 757 121	148	~ 2.96
GAMMA	Energy-Delay	50	5	30	1 411.31	57 847 873	196	~ 3.92
GAMMA	Energy-Delay	50	10	10	1 417.04	56 686 337	147	~ 2.94
GAMMA	Energy-Delay	50	10	20	1 409.51	150 543 553	197	~ 3.94
GAMMA	Energy-Delay	50	10	30	1 415.52	225 815 233	266	~ 5.32
GAMMA	Energy-Delay	50	20	10	1 424.49	170 616 001	213	~ 4.26
GAMMA	Energy-Delay	50	20	20	1 408.31	**50 181 313**	291	~ 5.82
GAMMA	Energy-Delay	50	20	30	**1 404.70**	**50 181 313**	408	~ 8.16

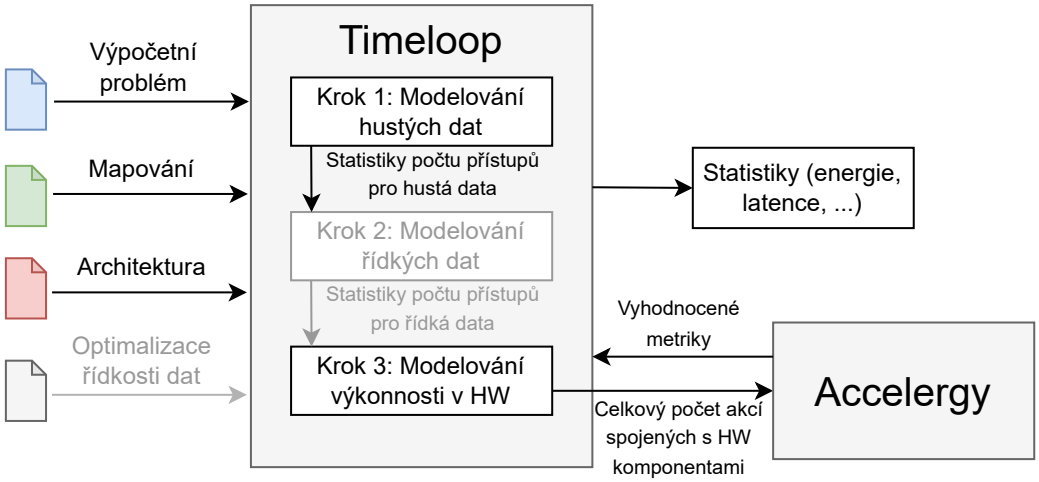
6.4 Závěr porovnání

Z experimentálního porovnání vyplývá, že framework Timeloop+Accelergy poskytuje vysokou míru konfigurovatelnosti a flexibility pro modelování rozličných HW architektur a umožňuje tak aktivně zkoumat dopad modifikace návrhu na odhadnuté parametry. Oproti tomu MAESTRO není zdaleka tak intuitivní a jeho hlavní pole působnosti směřuje vůči co-optimalizaci potřebných HW zdrojů a vhodného mapování k efektivnímu zpracování vstupní neuronové sítě. Ze získaných poznatků a zkušeností s prací s oběma nástroji byl pro implementaci podpory kvantizace zvolen Timeloop+Accelergy. Ten podle mého soudu disponuje větší flexibilitou, kredibilitou a reliabilitou pro jeho aplikaci v oblasti automatizovaného návrhu HW akcelerátorů (HW-NAS).

Kapitola 7

Princip činnosti nástroje Timeloop a návrh jeho rozšíření

Nástroje Timeloop a Accelergy jsou implementovány v jazyce C++ a mohou být nainstalovány a spuštěny buď nativně ve formě zkompilovaných binárních souborů, nebo přímo v prostředí Dockeru. V rámci dalšího popisu a implementace rozšíření budeme uvažovat pouze variantu nativní instalace. V této kapitole se budeme zabývat hlavně interním fungováním nástroje Timeloop s cílem identifikovat místa, která je třeba upravit pro přidání podpory kvantizace. Nástroj Accelergy přitom zůstane beze změn. Důvodem toho je, že Timeloop modeluje přístupy a akce spojené s mapováním výpočetního problému (workload) na zadanou hardwarovou architekturu a bere v úvahu i vlastnosti dat jako jsou řídkost nebo plánovaná podpora bitové šířky. Nástroj Accelergy slouží pouze k získání výsledných parametrů (metrik) inference, které jsou spojené s energetickou spotřebou a plochou jednotlivých komponent. K tomu se využije počet akcí poskytnutých Timeloopem. V následujících sekcích si podrobněji představíme vstupní konfigurační soubory, popíšeme implementační detaily Timeloopu a určíme místa, která bude potřeba rozšířit či upravit pro zohlednění volby bitové šířky operandů vstupních dat. Obrázek 8.1 zobrazuje diagram fungování frameworku Timeloop+Accelergy spolu s potřebnými vstupy a nastíněním interního fungování.



Obrázek 7.1: Diagram zobrazující princip funkčnosti nativního frameworku Timeloop+Accelergy. Šedé části značí volitelná nastavení.

7.1 Vstupní konfigurační soubory

Timeloop na vstupu pro svou funkčnost vyžaduje několik vstupních souborů ve formátu YAML, které specifikují různé parametry a omezení HW architektury a výpočetního problému. Mezi tyto soubory patří soubor popisující hierarchickou strukturu hardwarových komponent a jejich vlastnosti, soubor popisující rozměry a přístupové vzory tenzorů tvořících výpočetní problém, volitelně soubor popisující formát uložení dat spolu s možnostmi optimalizace přístupů spojené s řídkostí dat (gating, skipping) a soubor popisující mapování problému na architekturu nebo informace pro automatické prohledávání a vyhodnocování mapování. Z hlediska kvantizace jsou nejrelevantnější soubory, které popisují výpočetní problém a hardwarovou architekturu. Informace o formátu dat a optimalizacích spojených s řídkostí dat, omezeními či specifikací pro prohledávání mapování nemají přímý vliv na definici kvantizace. Nicméně, zohlednění různých úrovní kvantizace v těchto souborech, zejména pak v mapování a míře řídkosti, může výrazně ovlivnit energetickou a výkonnostní efektivitu modelovaného akcelerátoru.

7.1.1 Popis výpočetního problému

Výpočetní problém reprezentuje typ výpočtu, který chceme modelovat. Příkladem může být konvoluční vrstva, GEMM jádro nebo maticové násobení. Popis výpočetního problému začíná kořenovým klíčem *problem* a obsahuje dva klíče druhé úrovně – *shape* a *instance*.

Klíč *shape* definuje operační prostor a datové prostory související s problémem. Způsob, jakým uživatel specifikuje tvar tenzoru, je inspirován Einsteinovou sumační notací (einsum), o které pojednává například článek [47]. Specifikace zahrnuje název tvaru problému, seznam dimenzí definujících výpočetní (operační) prostor datových tenzorů a nakonec projekci vztahu mezi výpočetním a datovým prostorem (vstupní a výstupní tenzory).

Klíč *instance* pak definuje skutečné velikosti pro každou dimenzi tenzoru a konstantní koeficienty (padding, stride) použité ve specifikaci tvaru problému. V této části lze také specifikovat relativní hustotu dat spolu se způsobem statistického modelování výskytu nulových hodnot v daném tenzoru. Tyto specifikace pak umožňují zohlednit HW optimalizace jako je zero-gating [61] nebo zero-skipping [54].

Ukázka YAML popisu 1. konvoluční vrstvy sítě AlexNet spolu se specifikací relativní četnosti hustoty dat jednotlivých tenzorů a způsobu statistického modelování výskytu nulových hodnot vypadá následovně:

```
# Timeloop workload definition
problem:
  shape:
    name:  CNN-Layer
    dimensions:  [ R, S, P, Q, C, K, N ]
    coefficients:
      - default:  1
        name:  Wstride
      - default:  1
        name:  Hstride
      - default:  1
        name:  Wdilation
      - default:  1
        name:  Hdilation
    data-spaces:
      - name:  Weights
        projection:
          - [ [C] ]
          - [ [K] ]
          - [ [R] ]
```

```
        - [ [S] ]
    - name: Inputs
      projection:
        - [ [N] ]
        - [ [C] ]
        # SOP form: R*Wdilation + P*Wstride
        - [ [R, Wdilation], [P, Wstride] ]
        # SOP form: S*Hdilation + Q*Hstride
        - [ [S, Hdilation], [Q, Hstride] ]
    - name: Outputs
      projection:
        - [ [N] ]
        - [ [K] ]
        - [ [Q] ]
        - [ [P] ]
      read-write: true
instance:
  C: 3 # input channels
  Hdilation: 1
  Hstride: 4
  M: 96 # output channels
  N: 1 # batch size
  P: 55 # output tensor width
  Q: 55 # output tensor height
  R: 11 # kernel width
  S: 11 # kernel height
  Wdilation: 1
  Wstride: 4
  # Optional data density specification
  densities:
    Inputs:
      distribution: fixed-structured
      density: 0.275
    Weights:
      distribution: fixed-structured
      density: 0.346133
    Outputs:
      distribution: fixed-structured
      density: 0.303
```

7.1.2 Popis architektury

Pro specifikaci kompletního návrhu akcelerátoru v nástroji Timeloop se používají tři hlavní klíče ve vstupním YAML souboru, z nichž dva jsou volitelné:

1. **architecture**: základní topologie hardwarové architektury
2. **sparse-optimizations**: (volitelné) specifikace optimalizací pro akceleraci řídkých dat (sparse acceleration features, SAFs) začleněných do návrhu
3. **constraints**: (volitelné) omezení týkající se platných mapování na architekturu, např. limity pro změnu pořadí smyček nebo přeskočení (bypassing)

Architektura

Pro popis architektury je třeba specifikovat organizaci hardwaru, tj. topologii propojených výpočetních a paměťových jednotek pomocí klíče nejvyšší úrovně *architecture*.

V rámci architektury se objevují dva hlavní typy komponent, a to paměťové třídy a výpočetní třídy. Každá z těchto tříd má své vlastní odpovídající atributy.

Paměťové třídy zahrnují například SRAM a DRAM. U těchto tříd je třeba specifikovat atributy, jako je hloubka paměti (počet řádků), šířka paměti (počet bitů na řádek), velikost

bloku (počet slov na řádek), šířka dat (počet bitů na slovo), šířka čtecího a zápisového pásma a další.

Výpočetní třídy, jako je MAC jednotka, mají atributy jako šířka dat (počet bitů na slovo) a počet instancí v prostorovém směru X a Y.

Hardwarová organizace je popsána jako hierarchický strom paměťových komponent, na jehož listech jsou umístěny aritmetické jednotky. V kořeni tohoto stromu se nachází záložní úložiště (jako je DRAM), které obsahuje veškerá data workloadu. Timeloop podporuje libovolný počet úrovní paměti. Každá úroveň může mít celočíselný násobek instancí, což umožňuje rozložení dat v paměťových komponentách nebo prostorový paralelismus výpočtu pomocí aritmetických jednotek.

Základní struktura stromu *architecture* je sdílena s Accelergy, který ji používá k charakterizaci energie a plochy spojené s komponentami architektury. Klíče, které nejsou rozuměny ani Timeloopem, ani Accelergy, jsou ignorovány.

Ukázka YAML popisu jednoduché architektury se 3 úrovňovou paměťovou hierarchií, šířkou slova 8 bitů a počítající s celočíselnými operandy vypadá například následovně:

```yaml
# Timeloop architecture definition
architecture:
  version: 0.3
  subtree:
  - name: System
    local:
    - name: MainMemory # no depth - inferred such that it is large enough to hold all the data
      class: DRAM
      attributes:
        width: 256
        block-size: 32 # width/word-bits
        word-bits: 8
    subtree:
    - name: Chip
      attributes:
        technology: 40nm
      local:
      - name: GlobalBuffer # 256KB buffer (8192*256/8)
        class: SRAM
        attributes:
          depth: 8192
          width: 256
          block-size: 32
          word-bits: 8
      subtree:
      - name: PE
        local:
        - name: RegisterFile
          class: regfile
          attributes:
            depth: 64
            width: 8
            block-size: 1
            word-bits: 8
        - name: MACC
          class: intmac
          attributes:
            datawidth: 8
```

Hardwarové optimalizace jsou klíčové pro zlepšení výkonu a snížení spotřeby energie. Tyto optimalizace zahrnují komprimované tenzorové formáty pro efektivní uložení operandů do paměti a zero-gating a zero-skipping neúčinných operací. Zero-gating a zero-skipping umožňují hardwarovým jednotkám vynechat přístupy do paměti pro řídké prvky vstupního vektoru nebo výpočty, které by vedly k nulovým výstupům, což snižuje energetickou spotřebu. Architekturní omezení definují limity pro mapování a pomáhají zohlednit skutečné omezení

HW tak, aby se zajistilo, že je mapování validní. Omezení mapovacího prostoru může být také specifikováno za účelem vyhnutí se evaluaci neefektivních mapování. Pro více informací spojených s řídkými optimalizacemi a omezeními pro platná mapování se prosím odkažte na dokumentaci zmíněnou při představení Timeloop+Accelergy na začátku kapitoly 5.

7.1.3 Popis mapování

Mapování určuje, jak jsou datové tenzory problému rozděleny a naplánovány v prostoru a čase napříč strukturou akcelerátoru. YAML formát mapování reprezentuje několikanásobně vnořené smyčky prostřednictvím seznamu direktiv, přičemž každá direktiva popisuje nějaký aspekt smyčky mapované na určité úrovni hardwaru. Klíčové prvky mapování zahrnují cíl (*target*), typ (*type*), permutaci (*permutation*) a faktory (*factors*), které určují, jak je výpočet rozdělen a naplánován napříč různými úrovněmi hardwaru. Mapování musí být legální a musí se vejít do dostupných HW zdrojů, což zahrnuje kapacity vyrovnávacích pamětí a prostorové instance na každé úrovni architektury. Timeloop kontroluje, zda mapování nepřekračuje dostupné hardwarové zdroje, a pokud tak učiní, vyhodí chybu a označí toto mapování za nevalidní.

Ukázka YAML formátu mapování pro výše zmíněnou architekturu a 1. konvoluční vrstvu sítě AlexNet:

```
# Timeloop architecture definition
mapping:
  - target: RegisterFile
    type: datatype
    keep:
      - Weights
      - Inputs
      - Outputs
    bypass:
      []
  - target: GlobalBuffer
    type: datatype
    keep:
      - Weights
      - Inputs
      - Outputs
    bypass:
      []
  - target: MainMemory
    type: datatype
    keep:
      - Weights
      - Inputs
      - Outputs
    bypass:
      []
  - target: RegisterFile
    type: temporal
    factors: C1 M1 R1 S11 N1 P1 Q5
    permutation: SQCMRNP
  - target: GlobalBuffer
    type: temporal
    factors: C3 M96 R11 S1 N1 P1 Q11
    permutation: MCRQSNP
  - target: MainMemory
    type: temporal
    factors: C1 M1 R1 S1 N1 P55 Q1
    permutation: PCMRSNQ
```

Výše specifikovanému mapování popsaného pomocí YAML direktiv odpovídá následující struktura vnořených smyček:

```
MainMemory [ Weights:34848 (34848) Inputs:154587 (154587) Outputs:290400 (290400) ]
-----------------------------------------------------------------------------------
| for P in [0:55)

GlobalBuffer [ Weights:34848 (34848) Inputs:7491 (7491) Outputs:5280 (5280) ]
---------------------------------------------------------------------------
| for Q in [0:11)
| for R in [0:11)
| for C in [0:3)
| for M in [0:96)

RegisterFile [ Weights:11 (11) Inputs:27 (27) Outputs:5 (5) ]
-------------------------------------------------------------
| for Q in [0:5)
| for S~in [0:11)
```

7.1.4 Potřebné dodatky

Pro zohlednění modelování kvantizace v nástroji Timeloop je nutné přidat možnost specifikovat bitovou šířku jednotlivých datových tenzorů do popisu výpočetního problému, podobně jako je zde již zahrnut popis řídkosti dat. V současné implementaci Timeloopu se předpokládá stejná bitová šířka pro všechna data, která se odvozuje z šířky slova v paměti. Implementací techniky bit-packingu by bylo možné uložit více datových operandů s menší šířkou do jednoho slova, což by mělo vliv na celkovou kapacitu pamětí a využití přenosového pásma. Techniku bit-packingu používají například práce [23, 5]. Díky této změně by se mohla stát některá původně nevalidní mapování validními, což by mohlo vést i k lepšímu využití paralelismu systolických polí aritmetických jednotek. Jelikož Timeloop je analytický nástroj, není nutné zahrnout konkrétní kvantizační techniku ani dosaženou přesnost. Tyto informace však mohou být užitečné při propojení Timeloopu s jinými nástroji pro kvantizaci modelů.

7.2 Běh Timeloopu

Jak již bylo zmíněno v podsekci 5.1.1, Timeloop lze spustit ve dvou hlavních režimech: timeloop-model a timeloop-mapper. Mimo ně existuje i režim timeloop-metrics, který pouze instanciuje HW architekturu a pro jednotlivé komponenty s pomocí Accelergy vypíše jejich charakteristiky jako jsou plocha a energetická náročnost za vykonanou akci. Zajímat nás tedy bude hlavně režim modelu a mapperu.

7.2.1 Proces spuštění

Soubory ve složce `src/applications` zajišťují spuštění běhu nástroje. Jak model, tak mapper nejprve zpracovávají vstupní konfigurační soubory, které se týkají architektury, omezení a výpočetního problému. Pro práci s načtenými YAML soubory se používají třídy `CompoundConfig` a `CompoundConfigNode` nacházející se ve složce `src/compound-config`. Za parsování konfiguračního souboru s výpočetním problémem je zodpovědná třída `Workload` ve složce `src/workload`. Kromě toho se v režimu modelu navíc parsuje konkrétní uživatelem předané mapování, zatímco v módu mapperu se provádí faktorizace dimenzí vstupních tenzorů a vytváření všech možných permutací (vnoření smyček), které představují naplánování výpočtu v prostoru a čase v hardwaru (temporální a spatial smyčky). K tomu se použije funkce jmenného prostoru `mapspace` s názvem `ParseAndConstruct()`, která vrací ukazatel

na objekt obsahující informace o vytvořeném prostoru mapování. Poté je prostor rozdělen na vlákna, z nichž každé se řídí uživatelem zvolenými parametry, jako je volba heuristiky, optimalizované metriky pro hodnocení kvality mapování a ukončovací podmínky prohledávání. Po dokončení běhu všech vláken se získá globálně nejlepší mapování z hlediska optimalizovaných metrik a na výstup se vypíší s ním spojené statistiky.

7.2.2 Evaluace mapování

Pro hodnocení kvality mapování na dané architektuře a kontrolu jejich validity se využívá evaluační engine. Ten je implementován ve třídě `Topology` ve složce `src/model`. Tato třída obsahuje metody pro konstrukci hierarchie paměti a aritmetických jednotek architektury, přičemž zohledňuje specifikaci výpočetního problému a HW optimalizace. Dále provádí evaluaci výkonnosti mapování na architektuře. Evaluační engine také shromažďuje statistiky o počtu přístupů do paměti, přenosů dat po přenosových pásmech a operacích na každé úrovni hierarchie. Tyto statistiky se následně využívají v kombinaci s nástrojem Accelergy pro výpočet souhrnných metrik, jako jsou cykly, energie a plocha.

Během evaluace topologie se kontroluje, zda jednotlivé paměťové úrovně mají dostatečnou kapacitu pro uložení dat a zda nedochází k porušení uživatelských omezení. K tomuto účelu jsou implementovány virtuální metody `PreEvaluationChecks()` a `EvaluationChecks()`, jejichž definice závisí na typu komponenty. Metoda `PreEvaluationChecks()` slouží k rychlé kontrole validity mapování, aby se zabránilo spuštění běhu delší a statistiky vyhodnocující metody `EvaluationChecks()`. Stěžejními komponentami topologie jsou třída `BufferLevel`, která popisuje paměťové komponenty, třída `ArithmeticUnits` pro popis aritmetických jednotek a různé třídy spojené s reprezentací propojovacích sítí. Propojovací sítě nemusí být vždy přímo specifikovány na vstupu, a jsou pak následně odvozeny podle komponent umístěných na sousedních úrovních topologie.

7.2.3 Výstupní report statistik

Po dokončení evaluace Timeloop vytváří report statistik, který poskytuje informace o architektuře, mapování a výsledných metrikách. Report statistik obsahuje soubor s příponou `.stats.txt`. Tento report zachycuje chování různých hardwarových komponent v cílovém návrhu a uvádí celkovou energii, plochu a latenci cílového problému, který běží na navrženém hardwaru s použitím mapování uvedeného ve výstupním souboru mapování – soubory s příponou `.map.txt` a `.map.yaml`.

Report statistik se skládá z několika částí. První část popisuje specifikace architektury a příslušné rozložení výkonnosti pro každou úroveň hardwarové hierarchie. Zahrnuje informace o paměťových a aritmetických úrovních, jako jsou jejich kapacity, šířky pásma, energetické náročnosti a počty přístupů. Druhá část obsahuje specifikace mapování a příslušné rozložení výkonnosti pro každou úroveň mapovací hierarchie. Jsou zde informace o temporálních a spatial smyčkách, jako jsou jejich délky, počty iterací, počty instancí a počty operací. Třetí část uvádí souhrnné metriky pro celý návrh, jako jsou celkové cykly, energie a plocha. Tyto metriky jsou vypočítány na základě statistik shromážděných evaluačním enginem a referenčních tabulek energie a plochy vygenerovaných nástrojem Accelergy.

7.2.4 Potřebné modifikace

Zavedení podpory kvantizace do procesu evaluace topologie vyžaduje upravit způsob, jakým jsou zohledňovány kapacity pamětí a přenosová pásma v souvislosti se specifikací smyček

mapování a bitové šířky dat. Při zpracování vstupního problému bude zapotřebí upravit třídu `Workload` tak, aby detekovala definice bitové šířky a uložila je pro pozdější zohlednění při evaluaci. Klíčovými metodami, které je dále třeba upravit, jsou `PreEvaluationChecks()` a `EvaluationChecks()` související s třídou `BufferLevel` a třídami pro reprezentaci propojovacích sítí.

Takové úpravy by měly umožnit akceptování mapování, u kterých by se jinak operandy nemusely vejít do paměti. Zároveň by se měla efektivně snížit kapacita pamětí a počet přístupů do paměti při porovnání stejného mapování před kvantizací a po aplikaci techniky bit-packingu. Je důležité si uvědomit, že informace o vnoření smyček, které jsou součástí samotného mapování, jsou založeny na počtu elementů a neměly by se měnit. Úpravy by se měly týkat pouze způsobu, jak jsou jednotlivé smyčky mapování přiřazovány a plánovány do hardwaru, což by mělo ovlivnit statistiky hardwarových komponent (pamětí, propojovacích sítí).

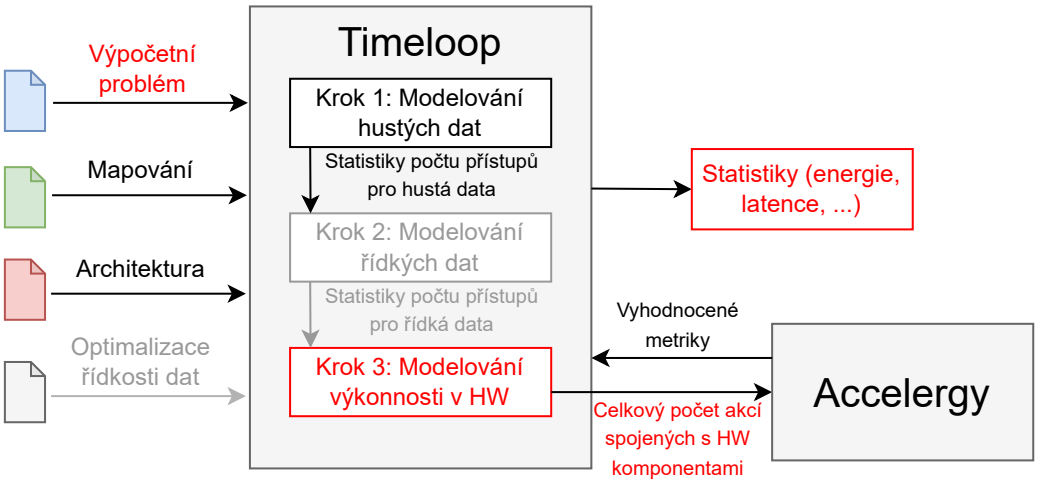
Kromě toho mohou do procesu zasahovat také hardwarové optimalizace, jako je gating a skipping, nebo změna formátu dat. Různé formáty pro kompresi dat (např. RLE[1], bitová maska atd.) mají specifickou strukturu metadat a ukládají se do paměti odlišně než běžná data. Metadata se tedy na kvantizaci přímo nevztahují a je tedy spíše třeba správně definovat vstupní konfigurace formátů, pokud chceme takový scénář správně modelovat.

[1] Run-length encoding

Kapitola 8

Implementace podpory kvantizace do nástroje Timeloop

V této kapitole se zaměříme na implementační úpravy spojené s rozšířením podpory kvantizace do nástroje Timeloop. Tato kapitola navazuje na předchozí kapitolu 7, která se věnovala přiblížení formátu vstupních souborů a principu běhu analytického modelu, a v níž byly identifikovány a navrženy změny potřebné k provedení. Obrázek 8.1 zobrazuje diagram fungování frameworku Timeloop+Accelergy spolu s červeně zvýrazněnými částmi, které bylo třeba upravit pro účely modelování kvantizace.



Obrázek 8.1: Diagram zobrazující princip funkčnosti rozšířeného frameworku Timeloop+Accelergy. Šedé části značí volitelná nastavení. Červené části značí sekce, které byly upraveny pro zohlednění modelování podpory kvantizace.

8.1 Úprava vstupního souboru popisující výpočetní problém

Úprava spočívala v zahrnutí bitové šířky jednotlivých datových tenzorů do popisu výpočetního problému, což umožnilo následné modelování kvantizace při evaluaci topologie pro jednotlivá mapování. Do sekce *instance* vstupního konfiguračního souboru byla po vzoru specifikace density přidána možnost konfigurace bitové šířky dat. Možnosti specifikace bitové šířky jsou pak následující:

- **bitwidths/datawidths/wordwidths**: klíč následovaný YAML mapping collection, která obsahuje páry klíč (název datového tenzoru) a hodnota (bitová šířka)
- **commonBitwidth/commonDatawidth/commonWordwidth**: klíč následovaný hodnotou stejné bitové šířky pro všechny datové tenzory
- **žádná specifikace**: původní chování Timeloopu (odvození podle šířky slova paměti)

Ukázka rozšířené specifikace *instance* vstupního souboru popisující výpočetní problém s konfigurací bitové šířky jednotlivých dat vypadá například následovně:

```
# Enhanced Timeloop workload definition
problem:
  # shape specification ...
  instance:
    # dimension specification
    # ...
    # coefficients specification
    # ...
    # Optional data bitwidth specification
    bitwidths:
      Inputs:  16
      Weights:  4
      Outputs:  16
    # Optional data density specification
    # ...
```

Současně s přidanými specifikacemi byla také provedena úprava parsování vstupního souboru pomocí třídy `Workload`. Zde byla přidána detekce příslušných nových klíčů v rámci sekce *instance* a uložení voleb do vektoru hodnot `bitwidths` a vektoru pravdivostních hodnot `bitwidths_specified` pro snadnější kontrolu uživatelských nastavení.

8.2 Úprava evaluačního algoritmu

Po úpravách vstupního souboru popisující výpočetní problém, jak je popsáno v sekci 8.1, bylo nutné zajistit, aby byly bitové šířky jednotlivých datových tenzorů správně použity při vyhodnocení evaluačního enginu. To zahrnuje následující provedené úpravy:

1. Upravení třídy `BufferLevel` a jejích metod `PreEvaluationChecks()` a `EvaluationChecks()` pro zohlednění bitové šířky dat a vlivu bit-packingu. To zahrnuje snížení kapacity pamětí a počtu přístupů do paměti pro kvantizovaná mapování ve srovnání se stejným mapováním před kvantizací. Zároveň tato úprava zabezpečuje validitu těch mapování, která by bez zohlednění bitové šířky dat neprošla.
2. Upravení tříd pro reprezentaci propojovacích sítí a metod `PreEvaluationChecks()` a `EvaluationChecks()` pro zohlednění bitové šířky dat při přenosu mezi paměťovými úrovněmi podobným způsobem jako při třídě `BufferLevel`. Většinu změn nebylo třeba přímo provádět, jelikož se chování odvíjelo na základě již vyhodnocených výkonnostních statistik paměťových úrovní, které v sobě zohledňují dopady kvantizace. Tyto úpravy zohledňují skutečnost, že kvantizace může ovlivnit propustnost a energetickou náročnost přenosu dat mezi jednotlivými úrovněmi paměti.
3. Přidání statistik reportujících efektivní fragmentaci dat, tedy počtu nevyužitých bitů a počtu operandů na slovo pro danou úroveň smyčky mapování. Tyto statistiky napomáhají lépe porozumět dopadu kvantizace na výkon a energetickou účinnost akcelerátoru v rámci různých úrovní mapování. Současně bylo také přidáno generování

statistického souboru s příponou `.stats.csv` se souhrnnými statistikami běhu modelu/mapperu pro lepší zpracování výsledků.

Úpravy evaluačního enginu zajišťují, že Timeloop správně zohledňuje bitové šířky jednotlivých datových tenzorů a jejich vliv na energetickou účinnost a propustnost při vyhodnocování topologií a mapování. Díky tomu se lze informovaněji rozhodovat při návrhu a optimalizaci architektury akcelerátoru.

8.2.1 Úprava třídy BufferLevel

Do metody `PreEvaluationChecks()` byla přidána kontrola, zda je datový tenzor právě vyhodnocované smyčky kvantizován, či nikoliv. V případě specifikace bitové šířky se provede rychlé vyhodnocení předpokládaného využití paměti pro kontrolu splnitelnosti mapování. Tedy nejprve se pro tenzor kontroluje, jestli platí `workload->IsBitwidthSpecified(tensor)` a zároveň, že šířka slova paměti je různá od specifikované šířky dat, a že šířka dat nepřekračuje šířku slova. Pro výpočet využití paměti je použit následující vzorec:

$$required\_capacity = \left\lceil \frac{working\_set\_size}{\lfloor memory\_word\_bits \div get\_bitwidth(tensor) \rfloor} \right\rceil \tag{8.1}$$

Mapování, která pro všechny své dílčí smyčky nepřekročí celkovou kapacitu jednotlivých úrovní pamětí, definovaných v rámci konfigurace architektury, jsou uznána za validní. Tato mapování jsou následně vyhodnocena výpočetně náročnější metodou `EvaluationChecks()`, v rámci níž dochází k přiřazení jednotlivých smyček k dílčím pamětem a k modelování celkového počtu přístupů do paměti a jiných počtů akcí potřebných pro Accelergy k výpočtu souhrnných statistik mapování.

Výpočet metody `EvaluationChecks()` sestává z volání šesti metod, z nichž každá modeluje klíčový aspekt výkonnosti celého mapování pro danou paměťovou úroveň:

- `ComputeScalarAccesses()`
- `ComputeVectorAccesses()`
- `ComputeBufferEnergy()`
- `ComputeReductionEnergy()`
- `ComputeAddrGenEnergy()`
- `ComputePerformance()`

Pro zohlednění kvantizace datových tenzorů byly přímo upraveny metody `ComputeScalarAccesses()`, `ComputeVectorAccesses()`, které počítají výkonnostní statistiky a počty akcí modelujících běh mapování, a také metoda `ComputeReductionEnergy()`, v níž se počítá energie potřebná pro provedení redukčních operací v paměťové úrovni. Zbylé tři metody pak počítají se statistikami vyhodnocenými zbylými metodami a tudíž žádné přímé úpravy nevyžadovaly.

Metoda ComputeScalarAccesses

Metoda `ComputeScalarAccesses` počítá celkový počet skalárních přístupů do paměti pro danou úroveň a dílčí datové tenzory v ní ukládané. Skalární přístupy do paměti se týkají operací, které manipulují s jednotlivými datovými prvky. Tyto přístupy mohou zahrnovat čtení, zápis nebo aktualizaci skalárních hodnot.

Kvantizace pak zohledňuje tzv. `fine_grained_scalar_accesses`, zatímco `fine_grained_format_scalar_accesses` zůstávají beze změny. Důvodem toho je, že druhé zmiňované přístupy se váží k modelování metadat, které slouží k uchování komprimovaných dat, a které se řídí bitovou šířkou metadat, nikoliv dat samotných. Šířku metadat pak lze specifikovat spolu s popisem formátu pro reprezentaci dat a tato podpora již byla nativně součástí nástroje. Úprava `fine_grained_scalar_accesses` pro zohlednění kvantizace pro například počty skalárních čtení je realizována na základě následujícího vzorce:

$$scalar\_read\_accesses = \left\lceil \frac{required\_scalar\_reads}{\lfloor memory\_word\_bits \div get\_bitwidth(tensor) \rfloor} \right\rceil \tag{8.2}$$

Metoda ComputeVectorAccesses

V metodě `ComputeVectorAccesses` se počítá celkový počet vektorových přístupů do paměti pro danou úroveň a dílčí datové tenzory v ní ukládané. Vektorové přístupy do paměti se týkají operací, které manipulují s více datovými prvky najednou. Tyto přístupy mohou zahrnovat čtení, zápis nebo aktualizaci hodnot vektoru.

Metoda pak provádí kvantizaci vektorových přístupů podobně jako u skalárních přístupů, přičemž zohledňuje `fine_grained_vector_accesses`, zatímco `fine_grained_format_accesses` zůstávají beze změny podobně jako u metody `ComputeScalarAccesses`.

Úprava `fine_grained_vector_accesses` pro zohlednění kvantizace pro například počty vektorových čtení je realizována na základě následujícího vzorce:

$$vector\_read\_accesses = \left\lceil \frac{required\_vector\_reads}{\lfloor memory\_word\_bits \div get\_bitwidth(tensor) \rfloor} \right\rceil \tag{8.3}$$

Metoda ComputeReductionEnergy

V rámci metody `ComputeReductionEnergy` se počítá energie potřebná pro provedení redukčních operací v paměťové úrovni. Redukční operace zahrnují přidání hodnoty přicházející po síti k hodnotě uložené lokálně a je tedy na místě adekvátně uvažovat šířku uložených dat v paměti. Tedy místo šířky slova se pro generování adresy použije `workload->GetBitwidth-(tensor)`.

Mimo provedené implementační úpravy byly navíc do výsledných reportovaných statistik přidány údaje o počtu nevyužitých bitů ve slově vlivem fragmentace – `wasted_bits` a počtu operandů, které se efektivně uloží do jednoho slova paměti – `operands_per_word`. Záznamy jsou součástí výsledného statistického souboru s příponou `.stats.txt`.

8.3 Pomocné skripty

Ve složce `scripts/construct_workloads` jsou navíc skripty umožňující vygenerovat vstupní konfigurační soubory výpočetního problému reprezentující jednotlivé vrstvy z vybraného modelu z populárních knihoven pro práci s neuronovými sítěmi, kterými jsou PyTorch a TensorFlow. Původní skript k tomu určený totiž vyžadoval ruční definování dimenzí jednotlivých vrstev v určeném formátu.

Kapitola 9

Experimenty a testování s kvantizovanými modely

V rámci experimentálního vyhodnocení přidané podpory do nástroje Timeloop bylo záměrem otestovat vliv provedených změn na evaluaci různě kvantizovaných modelů na rozličných HW architekturách.

Experiment 9.1 zkoumá vliv kvantizace u konvolučních vrstvev sítě AlexNet na celkový počet a kvalitu nalezených mapování optimalizovaných vůči EDP na Eyeriss architektuře bez podpory i s podporou zero-gatingu.

Následuje experiment 9.2 zaměřující se na hodnocení kvantizace sítě VGG01 [41] na dvou architekturách: Eyeriss a architektuře inspirovanou Simba architekturou. Cílem je srovnání celkové EDP sítě na jednotlivých architekturách.

Závěrečný experiment 9.3 pak vyhodnocuje výkonnost kvantizovaných sítí MobileNet se stejnými bitovými šířkami pro reprezentaci map příznaků a uniformně různě zvolenou šířkou pro data vah. Pro tento experiment byly také pro účely srovnání získány referenční hodnoty přesností pro jednotlivé konfigurace sítí natrénovaných s využitím QAT metody na datasetu Cifar-10.

Testovaným HW akcelerátorem byl buď *Eyeriss-like*, specifikován v tabulce 6.1, *Simba-like* architektura popsána v tabulce 9.1 nebo modifikace těchto architektur. Zkoumanými sítěmi byly AlexNet s 5 konvolučními vrstvami, VGG01 s 8 konvolučními vrstvami a MobileNet s 28 konvolučními vrstvami.

Všechny Timeloop heuristiky (kromě exhaustive) používaly nastavení **timeout** = 15000, **victory-condition** = 500, **max-permutations-per-if-visit** = 16, pro více detailů viz dokumentace.

Tabulka 9.1: HW specifikace Simba-like architektury používané v rámci Timeloop experimentů s kvantizacemi. Specifikace je založena na základě specifikace poskytnuté v rámci ukázkových designů v rámci Timeloop tutoriálů. Simba implementuje jen 1 chiplet (místo 36) a ve svých PE má 16 místo 64 MAC. Také byla zvolena 45nm technologie vzhledem k větší flexibilitě u Open-source nástrojů pro odhady.

Architecture	Simba-like
Technology	45nm
Number of PEs	16
Shared Global Buffer (SRAM)	64 kB
Input smartbuffer	64 kB
Weight smartbuffer (4 per PE)	8192 words
Psum smartbuffer (4 per PE)	128 words
Register (16 per PE)	1 word
Word-bits	16
Data type	int

Všechna měření byla opět provedena na procesoru AMD Ryzen 9 5900X @3.7GHz, 12 jader, 24 vláken a s 16GB paměti RAM.

9.1 Srovnání kvality a počtu nalezených mapování

Tento experiment se zaměřuje na vyhodnocení šesti konfigurací vstupních konvolučních vrstev sítě AlexNet v kontextu dvou specifikací architektury Eyeriss. První specifikace je nezměněná, zatímco druhá zahrnuje podporu zero-gatingu přístupů k váhám v scratchpadu na základě nulových hodnot čtených operandů vstupních příznaků. Tato technika, zvaná zero-gating, je optimalizační metoda, která zabraňuje zbytečným paměťovým přístupům k operandům dat v paměti, pokud jsou hodnoty jiného datového tenzoru nulové. To vede ke snížení energetické náročnosti a zlepšení výkonnosti systému. Pro každé hardwarové nastavení jsou testovány tři konfigurace. Použitou heuristikou k prohledávání byla exhaustive metoda s cílem prohledat celý omezený prostor mapování.

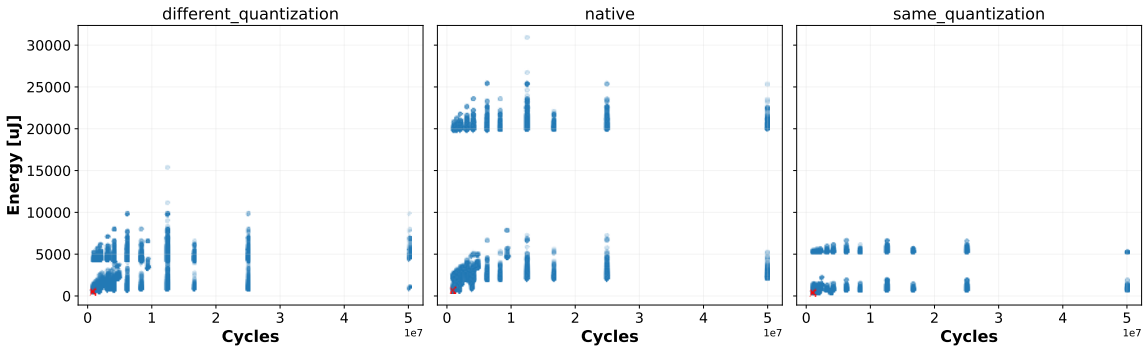
Jednotlivé konfigurace jsou definovány třemi různými nastavení kvantizace: nativním chování Timeloopu s odvozenou bitovou šířkou dat podle šířky slova v paměti (16 bitů), různou kvantizací pro datové tenzory, kde všechny příznaky používají 16 bitů a váhy 3 bity, a konečně kvantizací, kde všechny operandy jsou reprezentovány pomocí 4 bitů.

Tabulka 9.2 ukazuje nastavení relativní hustoty dat pro experimenty se zero-gatingem. Přístupy do těchto pamětí jsou pro váhy vynechány na základě detekce nulových hodnot vstupních příznaků. Lze očekávat, že v případě první konvoluční vrstvy nedojde k žádným optimalizacím vzhledem k absenci řídkosti dat v tenzoru vstupních příznaků.

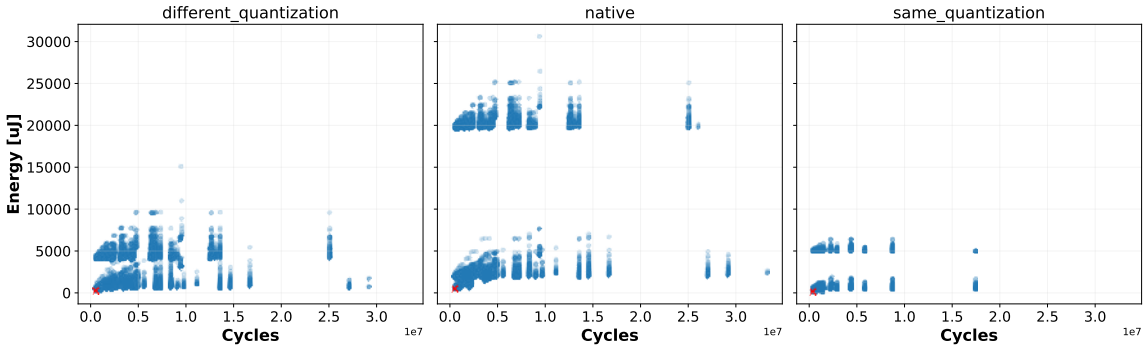
Tabulka 9.2: Nastavení relativní míry hustory jednotlivých datových tenzorů pro konvoluční vrstvy sítě AlexNet, testovaných v rámci experimentu s HW gating optimalizací scratchpad pamětí uchovávající váhy.

Vrstva	Vstupní fmap	Váhy	Výstupní fmap
AlexNet1	1.0	0.710166	0.491494
AlexNet2	0.372552	0.906464	0.222351
AlexNet3	0.346133	0.527598	0.389361
AlexNet4	0.372449	0.389361	0.429179
AlexNet5	0.368811	0.429179	0.160041

Obrázek 9.1 prezentuje grafy s celkovým počtem nalezených validních mapování optimalizovaných vůči metrice EDP pro 3. konvoluční vrstvu AlexNet na Eyeriss bez podpory zero-gatingu (hustá data). Naopak obrázek 9.2 zobrazuje stejný experiment, ale s řídkými daty a HW podporou gatingu scratchpad vah. Z grafů je zřejmé, že celkový počet detekovaných validních mapování je u kvantizovaných nastavení vyšší než u nativního běhu, přičemž konfigurace se stejnou bitovou šířkou má validních mapování nejvíce. Toto je zapříčiněno skutečností, že díky 4bitovým datům je možné do jednoho slova umístit až čtyři operandy, což jednak zvyšuje efektivitu využití přenosového pásma, ale také šetří kapacitu paměti. V rámci této konfigurace by bylo také vhodné přizpůsobit výpočetní jednotky pro zpracování pouze 4bitových dat. Dále lze pozorovat vliv zero-gatingu na celkové reportované metriky. Vzhledem k řídkosti tenzoru vstupních příznaků v rámci 3. konvoluční vrstvy sítě AlexNet, která má relativní míru hustoty 0.346133, je pokles energie spojený s úsporou přístupů pro čtení operandů vah významný. Pro běh bez HW optimalizací vychází, že nativní konfigurace obsahuje 18079 mapování s nejlepší hodnotou EDP rovnající se 786.67 J*cycle. Kvantizace s různými datovými tenzory zahrnuje celkem 18157 mapování, přičemž nejlepší EDP dosahuje hodnoty 582.65 J*cycle. Graf reprezentující stejnou bitovou šířku datových operandů pak zobrazuje 25600 mapování, z nichž nejlepší dosahuje hodnoty EDP 418.95 J*cycle. Naopak pro konfigurace s HW optimalizacemi zero-gatingu nativní konfigurace obsahuje 18079 mapování s nejlepší hodnotou EDP rovnající se 276.51 J*cycle. Kvantizace s různými datovými tenzory zahrnuje celkem 18157 mapování, přičemž nejlepší EDP dosahuje hodnoty 196.33 J*cycle. Graf reprezentující stejnou bitovou šířku datových operandů pak zobrazuje 25600 mapování, z nichž nejlepší dosahuje hodnoty EDP 67.59 J*cycle.

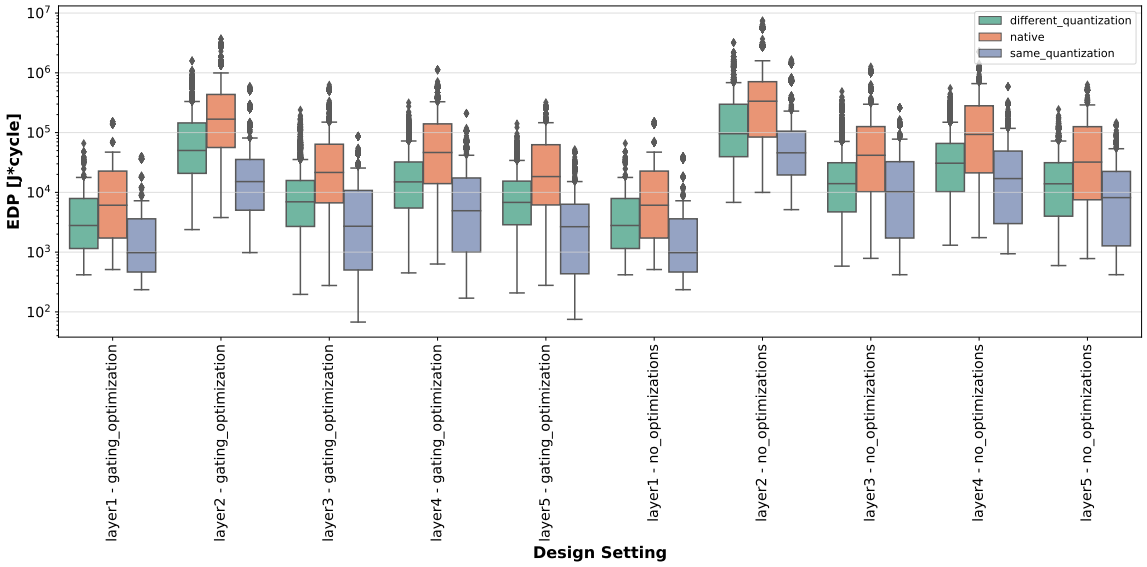


Obrázek 9.1: Srovnání počtu a kvality validních mapování pro různá nastavení kvantizace 3. konvoluční vrstvy sítě AlexNet bez použití HW optimalizace zero-gatingu.



Obrázek 9.2: Srovnání počtu a kvality validních mapování pro různá nastavení kvantizace 3. konvoluční vrstvy sítě AlexNet s HW optimalizací zero-gatingu.

Na obrázku 9.3 jsou vyobrazeny krabicové grafy znázorňující nalezená mapování pro celou síť AlexNet pro všech šest konfigurací napříč dvěma HW nastaveními. Z grafů je patrné, že kvantizace a HW optimalizace mají významný vliv na kvalitu nalezených hodnot EDP, přičemž nejlepší výsledky jsou dosaženy při kombinaci obou těchto přístupů.



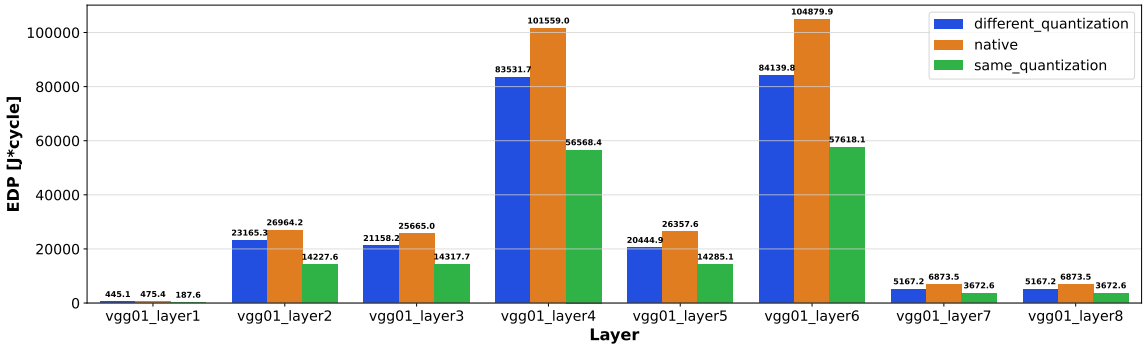
Obrázek 9.3: Srovnání krabicových grafů jednotlivých nastavení kvantizace pro celou síť AlexNet evaluovanou na dvou HW instancích – s podporou a bez podpory zero-gatingu.

9.2 Evaluace VGG01 na různých HW architekturách s rozličnou konfigurací kvantizace

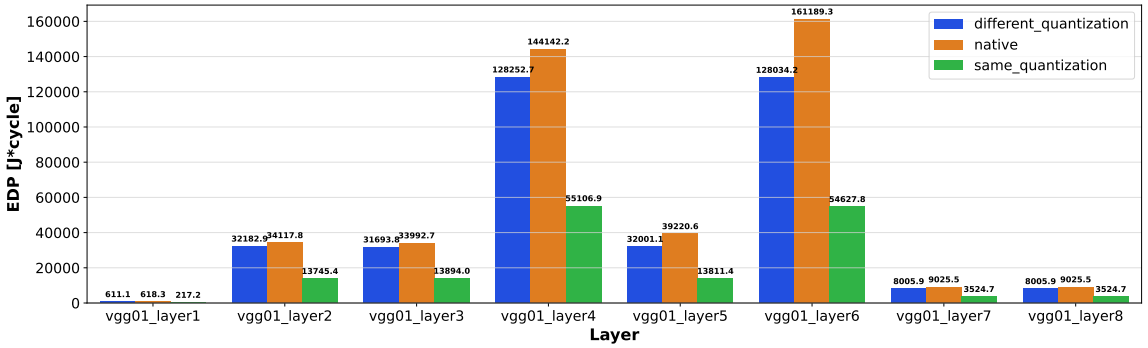
V rámci druhého experimentu byla testována síť VGG01 s 8 konvolučními vrstvami na nepozměněných architekturách Eyeriss_like a Simba_like pro tři různé konfigurace kvantizace dat. Tedy celkem šest různých experimentálních běhů podobně jako v případě experimentu 9.1. Pro prohledávání prostoru mapování byla použita heuristika random-pruned, optimalizována byla metrika EDP.

Jednotlivé konfigurace jsou stejně jako v předchozím experimentu definovány třemi různými nastavení kvantizace: nativním chování Timeloopu s odvozenou bitovou šířkou dat podle šířky slova v paměti (16 bitů), různou kvantizací pro datové tenzory, kde všechny příznaky používají 16 bitů a váhy 3 bity, a konečně kvantizací, kde všechny operandy jsou reprezentovány pomocí 4 bitů.

Obrázek 9.4 ukazuje srovnání běhu kvantizovaných nastavení pro jednotlivé vrstvy sítě VGG01 na architektuře Eyeriss_like a obrázek 9.5 pak srovnání běhu těchto nastavení na architektuře Simba_like. Z grafů je patrné, že i když jsou hodnoty EDP pro architekturu Eyeriss_like výrazně nižší, trend vzájemného srovnání tří nastavení je podobný. Jedinou výjimkou je nižší hodnota EDP pro nastavení pracující s 4bitovou šířkou dat, které je celkově nejlepší pro architekturu Simba_like. Tento jev může být důsledkem tvorby lepších mapování pro tuto konkrétní architekturu a nastavení bitové šířky.



Obrázek 9.4: Srovnání 3 konfigurací nastavení kvantizace pro nejlepší nalezenou hodnotu EDP pomocí random-pruned heuristiky na Eyeriss-like architektuře.



Obrázek 9.5: Srovnání 3 konfigurací nastavení kvantizace pro nejlepší nalezenou hodnotu EDP pomocí random-pruned heuristiky na Simba-like architektuře.

9.3 Výkonnost uniformně kvantizovaných sítí MobileNet na modifikovaných specifikacích HW akcelerátorů

V posledním experimentu se testoval vliv různého nastavení kvantizace na výkonnost modifikovaných architektur akcelerátorů Eyeriss_like a Simba_like. Celkově bylo uvažováno sedm různých nastavení kvantizací modelů sítí MobileNet pracujících s 8bitovými mapami

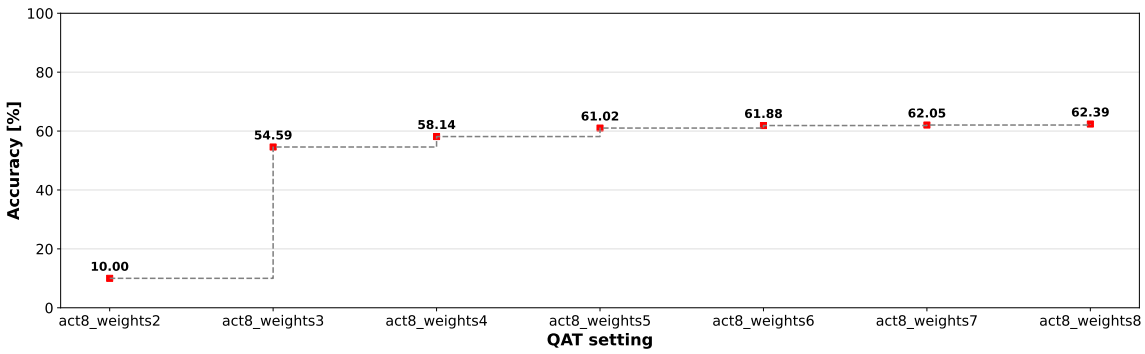
příznaků a $2-8$ bity pro váhy, přičemž volba šířky hodnot vah byla vždy uniformní pro všechny vrstvy sítě. Všechny konfigurace byly následně vyhodnoceny na třech modifikacích architektury Eyeriss_like a třech modifikacích architektury Simba_like.

V rámci architektury Eyeriss_like byla testována původní nezměněná 16bitová architektura, dále 8bitová architektura se stejnými specifikacemi velikostí pamětí a přenosových pásem a nakonec 8bitová architektura se zmenšenými velikostmi scratchpad pamětí pro uchování vah na 0.75 původní velikosti a zdvojnásobenými šířkami přenosových pásem.

Pro architekturu Simba_like se testovala původní nezměněná 16bitová architektura, dále 8bitová architektura se stejnými specifikacemi velikostí pamětí a nakonec 8bitová architektura se zmenšenými velikostmi smartbuffer pamětí pro uchování vah na 0.75 původní velikosti.

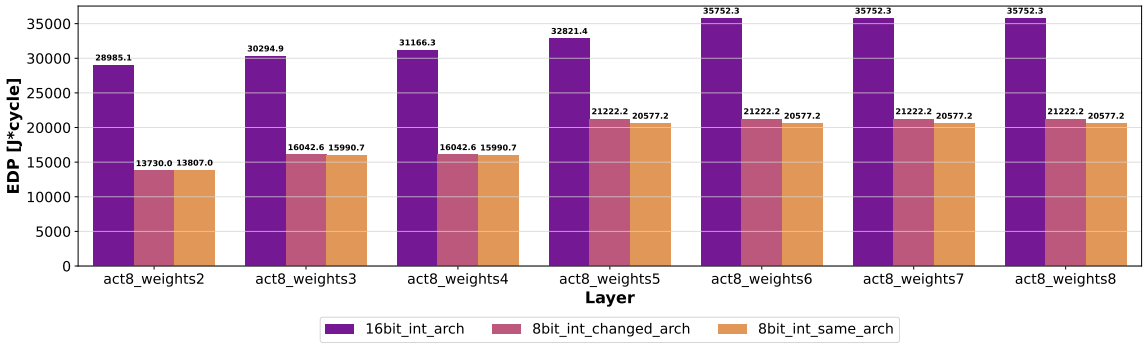
Pro prohledávání prostoru mapování se optimalizovalo vůči EDP a použila se random-pruned heuristika.

Obrázek 9.6 zobrazuje validační přesnosti modelů MobileNet, které byly dotrénovány pomocí metody QAT na původním modelu natrénovaném na datasetu Cifar-10. Pro trénování původního modelu byl použit batch-size 64 a learning rate 0.075. Pro QAT dotrénování se rovněž zvolil batch-size 64 a learning rate o řád nižší, tedy 0.0075. Původní validační přesnost sítě před dotrénováním byla 61.40 %. Z grafu je patrné, že od konfigurace uvažující 6bitové váhy byla validační přesnost původního modelu překonána. Modely byly trénovány na superpočítači Karolina, disponující 8 grafickými kartami NVIDIA A100 se 40 GB pamětí typu HBM2.

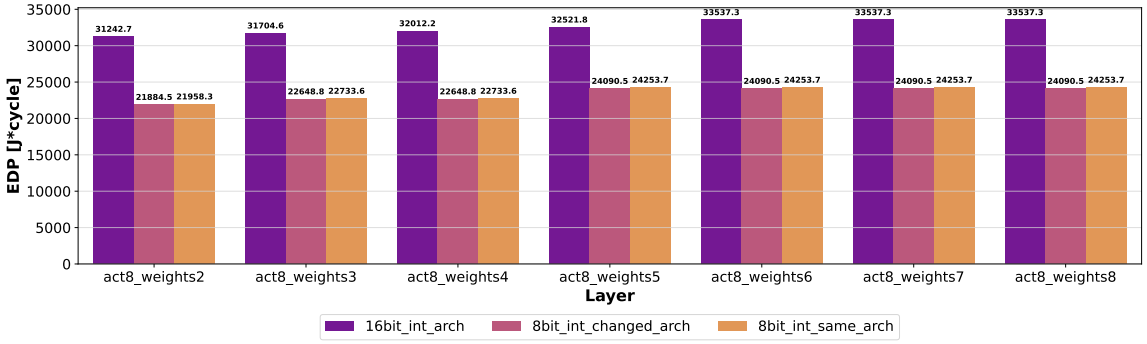


Obrázek 9.6: Graf validační přesnosti modelů MobileNet dotrénovaných pomocí metody QAT na původním modelu natrénovaném na datasetu Cifar-10. Původní model pracuje s 32-bit float daty a dosahuje validační přesnosti 61.40 %. Celkově bylo dotrénováno 7 konfigurací s 8bitovou int reprezentací hodnot příznaků a uniformně $2-8$bitovou int reprezentací hodnot vah.

Obrázky 9.7 a 9.8 zobrazují celkovou spotřebu modelů MobileNet napříč všemi 28 vrstvami pro jednotlivá nastavení kvantizace dat, testovaná na modifikacích architektury typu Eyeriss_like, respektive Simba_like. Volba bitové šířky pro reprezentaci slov v paměti zde hraje klíčovou roli v celkové spotřebě EDP. Přístupy do paměti jsou totiž v případě menší bitové šířky slova energeticky méně náročné. Dále je patrný vliv změny velikosti pamětí uchovávajících váhy. Pro architekturu Eyeriss_like přináší tato změna výhody pouze u konfigurace s 2bitovými váhami. Naproti tomu u architektury Simba_like je změna velikosti pamětí výhodná pro všechny kvantizované modely. Rovněž je zřejmé, že metoda bit-packingu, používaná ke kvantizaci operandů do slova, je těsně spojená s šířkou slova, což způsobuje, že některé konfigurace reportují stejnou celkovou spotřebu.



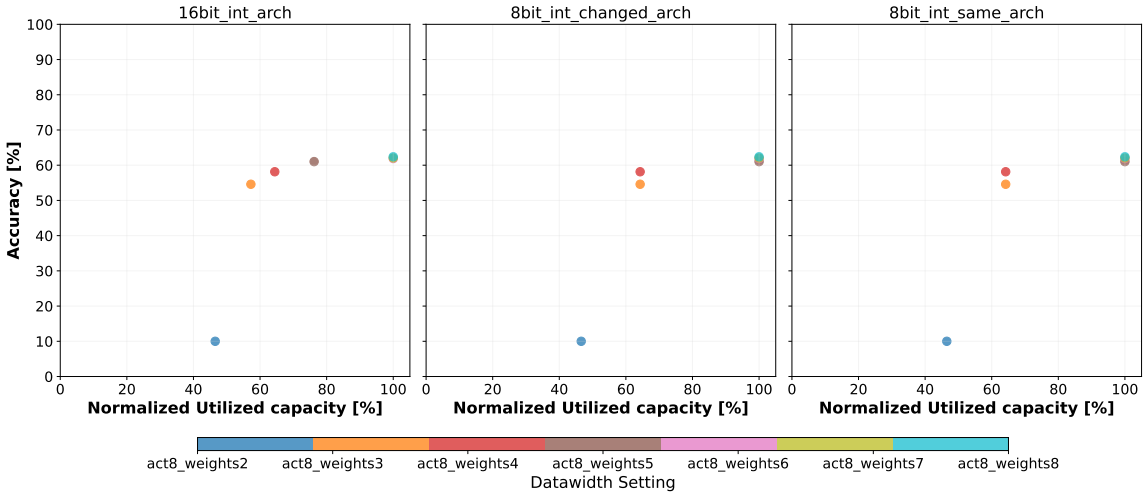
Obrázek 9.7: Graf celkové spotřeby EDP pro všech 28 vrstev sítě MobileNet pro všechna konfigurační nastavení kvantizace testovaná v rámci 3 modifikací architektury Eyeriss_like.



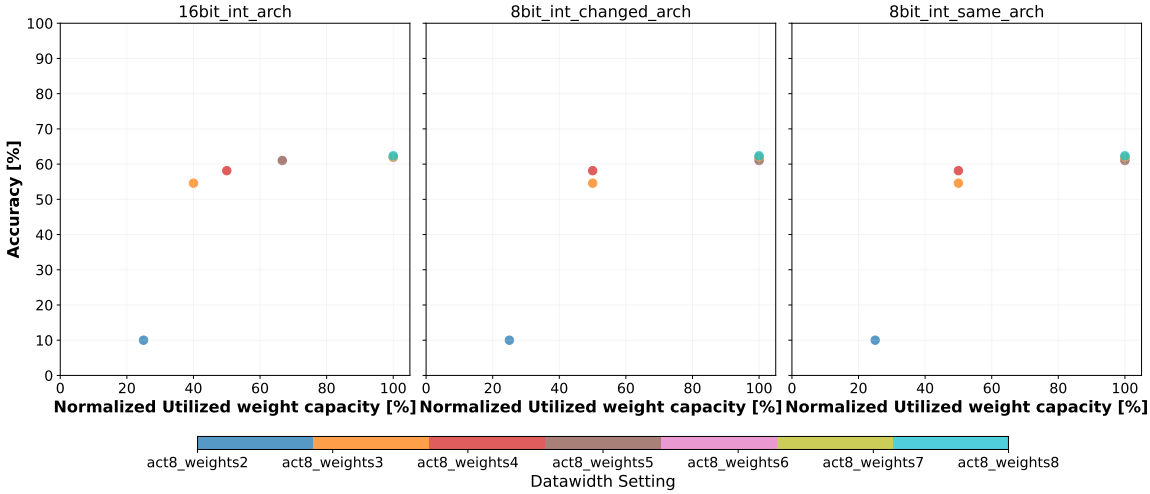
Obrázek 9.8: Graf celkové spotřeby EDP pro všech 28 vrstev sítě MobileNet pro všechna konfigurační nastavení kvantizace testovaná v rámci 3 modifikací architektury Simba_like.

Obrázky 9.9 a 9.10 zobrazují vliv různých nastavení kvantizace na využití kapacity paměti v Eyeriss_like akcelerátoru při zpracování všech 28 vrstev sítě MobileNet. První obrázek se zaměřuje na celkové využití paměti, zahrnující váhy i mapy příznaků, zatímco druhý se zabývá pouze využitím paměti pro uložení vah. Důležitou roli zde hraje nastavení šířky slova v paměti, které má přímý vliv na proces bit-packingu a ovlivňuje kolik operandů lze pro danou bitovou šířku dat uložit do jednoho slova v paměti. To vede k tomu, že v některých případech nedochází k úspoře paměti, jak je pozorováno například pro váhy o šířce $5 - 8$ bitů při použití 8 bitů na slovo. V případě uvažování naivního přístupu pro odhad celkové spotřeby kapacity se totiž uvažuje, že by při použití 5 bitů pro uložení vah na 8 bitové architektuře došlo ke snížení kapacity paměti spojené s uložením vah o 37.5%, k čemuž by došlo při využití pouze 5 bitů z celkových 8. Při použití bit-packingu v HW však k žádné úspoře paměti nedojde, jelikož by to vedlo k rozdělení bitů operandu do 2 slov, což by způsobilo fragmentaci dat, více přístupů do paměti, spojené s potřebou načíst obě slova, a další HW logice navíc pro sestavení celého operandu.

Z grafů vyplývá, že v rámci celkové úspory kapacity paměti došlo například u 16bitové architektury a použití 5 bitů pro uložení vah ke snížení celkového využití paměti o 24% za cenu ztráty 1.4% na validační přesnosti. U 8bitové architektury to pak například pro 4bitové váhy znamenalo úsporu o 36% kapacity paměti za cenu ztráty 4.2% na validační přesnosti.



Obrázek 9.9: Graf srovnávající relativní využití paměti pro uchování všech dat pro 28 vrstev sítě MobileNet při různých nastaveních kvantizace. Hodnoty jsou normalizovány vůči 8bitovému nastavení vah i příznaků pro jednotlivé konfigurace architektury Eyeriss_like, které se liší šířkou slova v paměti, případně velikostí pamětí uchovávající váhy.



Obrázek 9.10: Graf srovnávající relativní využití paměti pro uchování vah pro 28 vrstev sítě MobileNet při různých nastaveních kvantizace. Hodnoty jsou normalizovány vůči 8bitovému nastavení vah i příznaků pro jednotlivé konfigurace architektury Eyeriss_like, které se liší šířkou slova v paměti, případně velikostí pamětí uchovávající váhy.

Kapitola 10

Závěr

Hlavním cílem této diplomové práce bylo zvolit a implementačně rozšířit analytický nástroj pro modelování akcelerátorů neuronových sítí o HW podporu kvantizace vstupních datových tenzorů. Po pečlivém srovnání a zhodnocení kvality existujících nástrojů, jemuž se věnují kapitoly 5 a 6, byl nakonec pro účely této práce zvolen Timeloop. Tento nástroj byl následně upraven tak, aby zohledňoval kvantizaci dat v HW s využitím techniky bit-packingu.

Experimenty provedené s upraveným nástrojem jsou popsány v kapitole 9 a ukazují, že kvantizace tenzorů modelu má významný vliv na kvalitu nalezených mapování z hlediska energetické účinnosti a latence. Rozšířený nástroj tak představuje krok směrem k lepšímu a efektivnějšímu modelování hardwarových akcelerátorů zohledňujících různou bitovou šířku dat modelu, což lze dobře kombinovat s již existující podporou pro práci s řídkými daty, jak prokázal experiment 9.1. Experiment 9.3 navíc ukázal, že naivní přístup pro odhad celkové spotřeby kapacity paměti je nepřesný, a že například pro 8 bitovou architekturu Eyeriss se podařilo uspořit 36% kapacity paměti při ztrátě 4.2% na validační přesnosti.

Do budoucna by bylo vhodné zvážit rozšíření tohoto nástroje i o analytické modelování a optimalizace využívající techniku weight sharingu. Kromě toho by bylo užitečné integrovat tuto implementaci s nástroji pro automatické prohledávání kvantizací modelů. To by umožnilo další optimalizaci a mohlo by vést k vytvoření plně automatizovaného nástroje pro návrh hardwarových akcelerátorů, který by byl schopen automaticky nalézt nejlepší kvantizační schéma pro daný model a zvolenou hardwarovou architekturu.

Takový nástroj by pak mohl být přínosný pro výzkum a vývoj v oblasti hardwarových akcelerátorů pro neuronové sítě a mohl by vést k dalším zlepšením v oblasti energetické efektivity a výkonu. Kombinace těchto přístupů do jednoho nástroje by tak mohla posunout oblast automatizovaného návrhu hardwarových akcelerátorů (HW-NAS) o krok dále.

Literatura

[1] Alam, S. A., Anderson, A., Barabasz, B. a Gregg, D. Winograd Convolution for Deep Neural Networks: Efficient Point Selection. *CoRR*. 2022, abs/2201.10369. Dostupné z: https://arxiv.org/abs/2201.10369.

[2] Bebis, G. a Georgiopoulos, M. Feed-forward neural networks. *IEEE Potentials*. 1994, sv. 13, č. 4, s. 27–31. DOI: 10.1109/45.329294.

[3] Cai, Z., Xiong, Z., Xu, H., Wang, P., Li, W. et al. *Generative Adversarial Networks: A Survey Towards Private and Secure Applications*. arXiv, 2021. DOI: 10.48550/ARXIV.2106.03785. Dostupné z: https://arxiv.org/abs/2106.03785.

[4] Capra, M., Bussolino, B., Marchisio, A., Shafique, M., Masera, G. et al. An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks. *Future Internet*. 2020, sv. 12, č. 7. DOI: 10.3390/fi12070113. ISSN 1999-5903. Dostupné z: https://www.mdpi.com/1999-5903/12/7/113.

[5] Chen, G., He, S., Meng, H. a Huang, K. PhoneBit: Efficient GPU-Accelerated Binary Neural Network Inference Engine for Mobile Phones. *CoRR*. 2019, abs/1912.04050. Dostupné z: http://arxiv.org/abs/1912.04050.

[6] Chen, J. a Ran, X. Deep Learning With Edge Computing: A Review. *Proceedings of the IEEE*. 2019, sv. 107, č. 8, s. 1655–1674. DOI: 10.1109/JPROC.2019.2921977.

[7] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C. et al. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. *SIGARCH Comput. Archit. News*. New York, NY, USA: Association for Computing Machinery. feb 2014, sv. 42, č. 1, s. 269–284. DOI: 10.1145/2654822.2541967. ISSN 0163-5964. Dostupné z: https://doi.org/10.1145/2654822.2541967.

[8] Chen, Y.-H., Emer, J. a Sze, V. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro*. 2017, sv. 37, č. 3, s. 12–21. DOI: 10.1109/MM.2017.54.

[9] Chen, Y., Emer, J. S. a Sze, V. Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks. *CoRR*. 2018, abs/1807.07928. Dostupné z: http://arxiv.org/abs/1807.07928.

[10] Chen, Y.-H., Krishna, T., Emer, J. S. a Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*. 2017, sv. 52, č. 1, s. 127–138. DOI: 10.1109/JSSC.2016.2616357.

[11] Cheng, Y., Wang, D., Zhou, P. a Zhang, T. A Survey of Model Compression and Acceleration for Deep Neural Networks. *CoRR*. 2017, abs/1710.09282. Dostupné z: http://arxiv.org/abs/1710.09282.

[12] De Mulder, W., Bethard, S. a Moens, M.-F. A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*. 2015, sv. 30, č. 1, s. 61–98. DOI: https://doi.org/10.1016/j.csl.2014.09.005. ISSN 0885-2308. Dostupné z: https://www.sciencedirect.com/science/article/pii/S088523081400093X.

[13] Dhilleswararao, P., Boppu, S., Manikandan, M. S. a Cenkeramaddi, L. R. Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey. *IEEE Access*. 2022, sv. 10, s. 131788–131828. DOI: 10.1109/ACCESS.2022.3229767.

[14] Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L. et al. ShiDianNao: Shifting Vision Processing Closer to the Sensor. *SIGARCH Comput. Archit. News*. New York, NY, USA: Association for Computing Machinery. jun 2015, sv. 43, 3S, s. 92–104. DOI: 10.1145/2872887.2750389. ISSN 0163-5964. Dostupné z: https://doi.org/10.1145/2872887.2750389.

[15] Gholamalinezhad, H. a Khosravi, H. Pooling Methods in Deep Neural Networks, a Review. *CoRR*. 2020, abs/2009.07485. Dostupné z: https://arxiv.org/abs/2009.07485.

[16] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W. et al. A Survey of Quantization Methods for Efficient Neural Network Inference. *CoRR*. 2021, abs/2103.13630. Dostupné z: https://arxiv.org/abs/2103.13630.

[17] Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A. et al. Recent Advances in Convolutional Neural Networks. *CoRR*. 2015, abs/1512.07108. Dostupné z: http://arxiv.org/abs/1512.07108.

[18] Guo, C., Zhou, Y., Leng, J., Zhu, Y., Du, Z. et al. Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration. *CoRR*. 2020, abs/2002.08326. Dostupné z: https://arxiv.org/abs/2002.08326.

[19] Gustineli, M. A survey on recently proposed activation functions for Deep Learning. arXiv. 2022. DOI: 10.48550/ARXIV.2204.02921. Dostupné z: https://arxiv.org/abs/2204.02921.

[20] Haykin, S. S. *Neural networks and learning machines / Simon Haykin.* 3rd ed. Prentice Hall, 2009. ISBN 9780131471399.

[21] Hennessy, J. L. a Patterson, D. A. A New Golden Age for Computer Architecture. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. jan 2019, sv. 62, č. 2, s. 48–60. DOI: 10.1145/3282307. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/3282307.

[22] Ioffe, S. a Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*. 2015, abs/1502.03167. Dostupné z: http://arxiv.org/abs/1502.03167.

[23] Jeon, Y., Park, B., Kwon, S. J., Kim, B., Yun, J. et al. BiQGEMM: Matrix Multiplication with Lookup Table For Binary-Coding-based Quantized DNNs. *CoRR*. 2020, abs/2005.09904. Dostupné z: https://arxiv.org/abs/2005.09904.

[24] Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., Agrawal, G. et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR*. 2017, abs/1704.04760. Dostupné z: http://arxiv.org/abs/1704.04760.

[25] Kao, S.-C. a Krishna, T. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. New York, NY, USA: Association for Computing Machinery, 2020. ICCAD '20. DOI: 10.1145/3400302.3415639. ISBN 9781450380263. Dostupné z: https://doi.org/10.1145/3400302.3415639.

[26] Kao, S.-C., Pellauer, M., Parashar, A. a Krishna, T. DiGamma: Domain-aware Genetic Algorithm for HW-Mapping Co-optimization for DNN Accelerators. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022, s. 232–237. DOI: 10.23919/DATE54114.2022.9774568.

[27] Kwon, H., Chatarasi, P., Sarkar, V., Krishna, T., Pellauer, M. et al. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro*. 2020, sv. 40, č. 3, s. 20–29. DOI: 10.1109/MM.2020.2985963.

[28] Kwon, H., Samajdar, A. a Krishna, T. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. mar 2018, sv. 53, č. 2, s. 461–475. DOI: 10.1145/3296957.3173176. ISSN 0362-1340. Dostupné z: https://doi.org/10.1145/3296957.3173176.

[29] Li, C., Yu, Z., Fu, Y., Zhang, Y., Zhao, Y. et al. HW-NAS-Bench: Hardware-Aware Neural Architecture Search Benchmark. *CoRR*. 2021, abs/2103.10584. Dostupné z: https://arxiv.org/abs/2103.10584.

[30] Li, S., Chen, K., Ahn, J. H., Brockman, J. B. a Jouppi, N. P. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2011, s. 694–701. DOI: 10.1109/ICCAD.2011.6105405.

[31] Li, Z., Liu, F., Yang, W., Peng, S. a Zhou, J. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Transactions on Neural Networks and Learning Systems*. 2022, sv. 33, č. 12, s. 6999–7019. DOI: 10.1109/TNNLS.2021.3084827.

[32] Liang, T., Glossner, J., Wang, L. a Shi, S. Pruning and Quantization for Deep Neural Network Acceleration: A Survey. *CoRR*. 2021, abs/2101.09671. Dostupné z: https://arxiv.org/abs/2101.09671.

[33] Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y. et al. A survey of deep neural network architectures and their applications. *Neurocomputing*. 2017, sv. 234, s. 11–26.

DOI: https://doi.org/10.1016/j.neucom.2016.12.038. ISSN 0925-2312. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0925231216315533.

[34] Livni, R., Shalev Shwartz, S. a Shamir, O. On the Computational Efficiency of Training Neural Networks. arXiv. 2014. DOI: 10.48550/ARXIV.1410.1141. Dostupné z: https://arxiv.org/abs/1410.1141.

[35] McCulloch, W. S. a Pitts, W. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*. 1943, sv. 5, č. 4, s. 115–133. DOI: 10.1007/bf02478259.

[36] Motamedi, M., Gysel, P., Akella, V. a Ghiasi, S. Design space exploration of FPGA-based Deep Convolutional Neural Networks. In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2016, s. 575–580. DOI: 10.1109/ASPDAC.2016.7428073.

[37] Muñoz Martínez, F., Abellán, J. L., Acacio, M. E. a Krishna, T. STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators. In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 2021, s. 201–213. DOI: 10.1109/IISWC53511.2021.00028.

[38] Nunes, J. D., Carvalho, M., Carneiro, D. a Cardoso, J. S. Spiking Neural Networks: A Survey. *IEEE Access*. 2022, sv. 10, s. 60738–60764. DOI: 10.1109/ACCESS.2022.3179968. ISSN 2169-3536.

[39] Parashar, A., Raina, P., Shao, Y. S., Chen, Y.-H., Ying, V. A. et al. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2019, s. 304–315. DOI: 10.1109/ISPASS.2019.00042.

[40] Rahman, A., Oh, S., Lee, J. a Choi, K. Design space exploration of FPGA accelerators for convolutional neural networks. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 2017, s. 1147–1152. DOI: 10.23919/DATE.2017.7927162.

[41] Shafiee, A., Nag, A., Muralimanohar, N., Balasubramonian, R., Strachan, J. P. et al. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, s. 14–26. DOI: 10.1109/ISCA.2016.12.

[42] Shao, Y. S., Clemons, J., Venkatesan, R., Zimmer, B., Fojtik, M. et al. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: Association for Computing Machinery, 2019, s. 14–27. MICRO '52. DOI: 10.1145/3352460.3358302. ISBN 9781450369381. Dostupné z: https://doi.org/10.1145/3352460.3358302.

[43] Shao, Y. S., Reagen, B., Wei, G.-Y. a Brooks, D. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, s. 97–108. DOI: 10.1109/ISCA.2014.6853196.

[44] Shorten, C. a Khoshgoftaar, T. M. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*. Jul 2019, sv. 6, č. 1, s. 60. DOI: 10.1186/s40537-019-0197-0. ISSN 2196-1115. Dostupné z: https://doi.org/10.1186/s40537-019-0197-0.

[45] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. a Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. 2014, sv. 15, č. 56, s. 1929–1958. Dostupné z: http://jmlr.org/papers/v15/srivastava14a.html.

[46] Stjerngren, A., Gibson, P. a Cano, J. Bifrost: End-to-End Evaluation and optimization of Reconfigurable DNN Accelerators. In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2022, s. 288–299. DOI: 10.1109/ISPASS55109.2022.00042.

[47] Sullivan, B. *Einstein Summation Notation*. 2017. Dostupné z: http://vucoe.drbriansullivan.com/wp-content/uploads/Einstein-Summation-Notation.pdf.

[48] Svozil, D., Kvasnicka, V. a Pospichal, J. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*. 1997, sv. 39, č. 1, s. 43–62. DOI: https://doi.org/10.1016/S0169-7439(97)00061-0. ISSN 0169-7439. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0169743997000610.

[49] Sze, V., Chen, Y., Yang, T. a Emer, J. S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *CoRR*. 2017, abs/1703.09039. Dostupné z: http://arxiv.org/abs/1703.09039.

[50] Taghavi, M. a Shoaran, M. Hardware Complexity Analysis of Deep Neural Networks and Decision Tree Ensembles for Real-time Neural Data Classification. In: *2019 9th International IEEE/EMBS Conference on Neural Engineering (NER)*. Březen 2019, s. 407–410. DOI: 10.1109/NER.2019.8716983.

[51] Vanhoucke, V., Senior, A. a Mao, M. Z. Improving the speed of neural networks on CPUs. In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*. 2011.

[52] Vasudevan, A., Anderson, A. a Gregg, D. Parallel Multi Channel Convolution using General Matrix Multiplication. *CoRR*. 2017, abs/1704.04428. Dostupné z: http://arxiv.org/abs/1704.04428.

[53] Verma, N., Jia, H., Valavi, H., Tang, Y., Ozatay, M. et al. In-Memory Computing: Advances and Prospects. *IEEE Solid-State Circuits Magazine*. 2019, sv. 11, č. 3, s. 43–55. DOI: 10.1109/MSSC.2019.2922889.

[54] Véstias, M. P., Duarte, R. P., Sousa, J. T. de a Neto, H. C. Fast Convolutional Neural Networks in Low Density FPGAs Using Zero-Skipping and Weight Pruning. *Electronics*. 2019, sv. 8, č. 11. DOI: 10.3390/electronics8111321. ISSN 2079-9292. Dostupné z: https://www.mdpi.com/2079-9292/8/11/1321.

[55] Williams, S., Waterman, A. a Patterson, D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*. New York, NY,

USA: Association for Computing Machinery. apr 2009, sv. 52, č. 4, s. 65–76. DOI: 10.1145/1498765.1498785. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/1498765.1498785.

[56] Wu, R., Guo, X., Du, J. a Li, J. Accelerating Neural Network Inference on FPGA-Based Platforms—A Survey. *Electronics*. 2021, sv. 10, č. 9. DOI: 10.3390/electronics10091025. ISSN 2079-9292. Dostupné z: https://www.mdpi.com/2079-9292/10/9/1025.

[57] Wu, Y. N., Emer, J. S. a Sze, V. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, s. 1–8. DOI: 10.1109/ICCAD45719.2019.8942149.

[58] Wu, Y. N., Tsai, P.-A., Parashar, A., Sze, V. a Emer, J. S. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling. In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, s. 1377–1395. DOI: 10.1109/MICRO56248.2022.00096.

[59] Xu, X., Ding, Y., Hu, S. X., Niemier, M., Cong, J. et al. Scaling for edge inference of deep neural networks. *Nature Electronics*. Apr 2018, sv. 1, č. 4, s. 216–222. DOI: 10.1038/s41928-018-0059-3. ISSN 2520-1131. Dostupné z: https://doi.org/10.1038/s41928-018-0059-3.

[60] Ye, H., Zhang, X., Huang, Z., Chen, G. a Chen, D. HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, s. 1–6. DOI: 10.1109/DAC18072.2020.9218684.

[61] Ye, L., Ye, J., Yanagisawa, M. a Shi, Y. A Zero-Gating Processing Element Design for Low-Power Deep Convolutional Neural Networks. In: *2019 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2019, s. 317–320. DOI: 10.1109/APCCAS47518.2019.8953157.

[62] Yuan, M. a Lin, Y. Model selection and estimation in regression with grouped variables. *Journal of The Royal Statistical Society Series B-statistical Methodology*. Blackwell Publishing Ltd. 2006, sv. 68, č. 1, s. 49–67. DOI: 10.1111/J.1467-9868.2005.00532.X.

[63] Zhang, X., Wang, J., Zhu, C., Lin, Y., Xiong, J. et al. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2018, s. 1–8. DOI: 10.1145/3240765.3240801.

[64] Zhao, Y., Li, C., Wang, Y., Xu, P., Zhang, Y. et al. DNN-Chip Predictor: An Analytical Performance Predictor for DNN Accelerators with Various Dataflows and Hardware Architectures. *CoRR*. 2020, abs/2002.11270. Dostupné z: https://arxiv.org/abs/2002.11270.

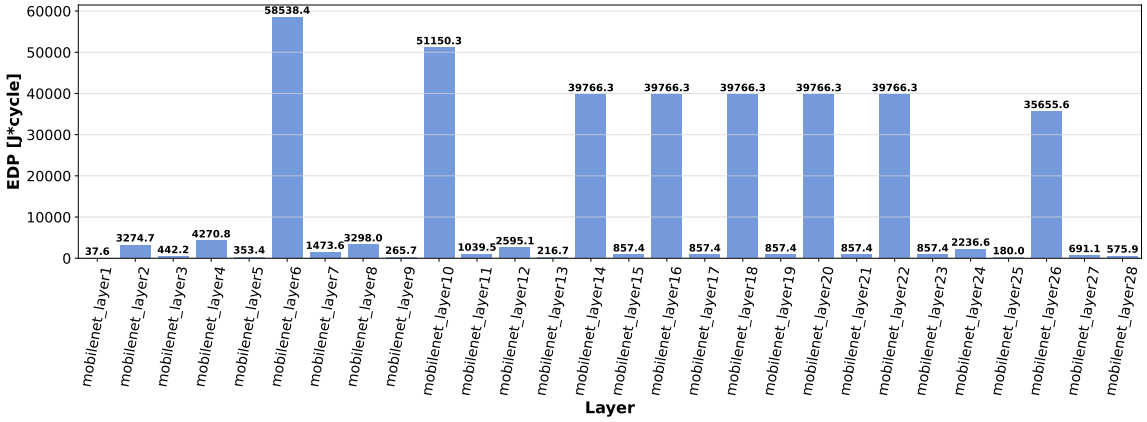
Příloha A

Obsah přiloženého paměťového média

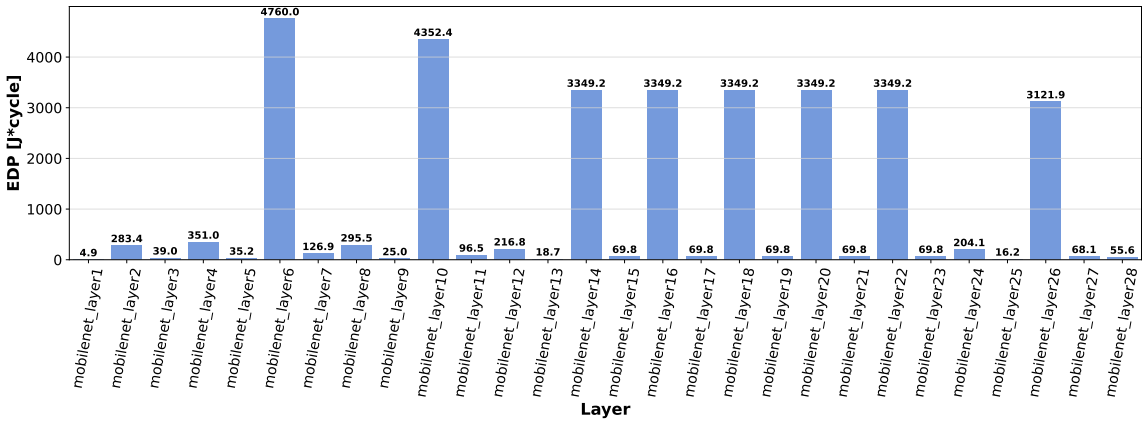
- source_codes – adresář obsahující zdrojové kódy a ukázkové testovací scénáře k experimentování
  - src – adresář obsahující dílčí implementační části frameworku Timeloop+Accelergy, z nichž stěžejní je adresář timeloop obsahující provedené úpravy
  - quantization_exercises – adresář obsahující testovací scénáře pro testování míry kvantizace na evaluaci vrstev CNN v HW, a také pomocné skripty pro zpracování výsledků
  - README.md – textový soubor popisující nástroj Timeloop+Accelergy spolu s návodem ke kompilaci
- master's_thesis – adresář obsahující zdrojové kódy technické zprávy v LaTeXu a její vysázenou verzi ve formátu *pdf*

Příloha B

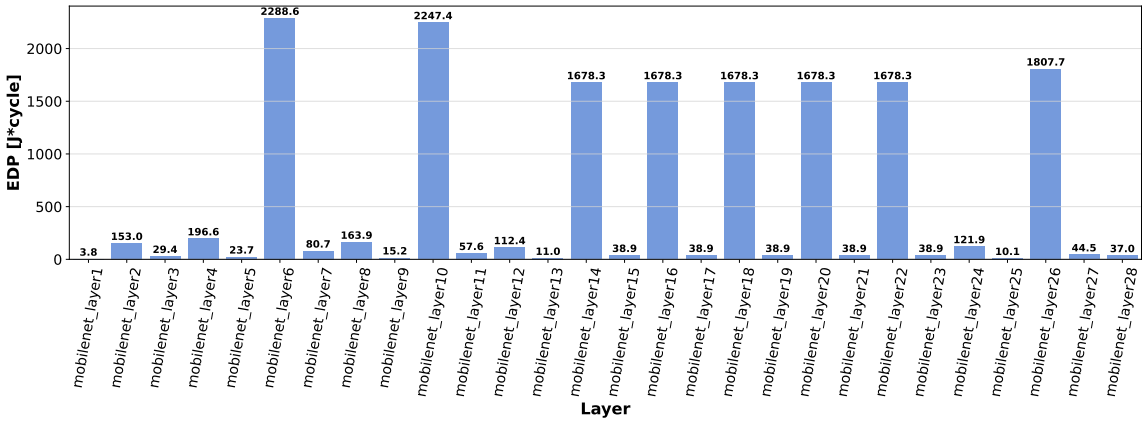
Dodatečné obrázky k experimentu MobileNet pro 4bitové váhy



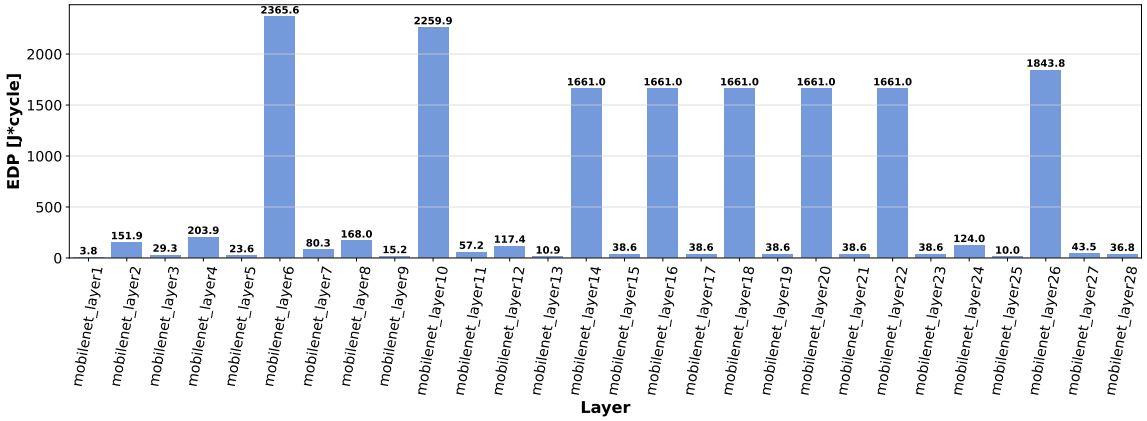
Obrázek B.1: Evaluace optimalizace vůči EDP pro jednotlivé vrstvy sítě MobileNet pro nativní 32-bit float data a adekvátně upravenou architekturu Eyeriss_like.



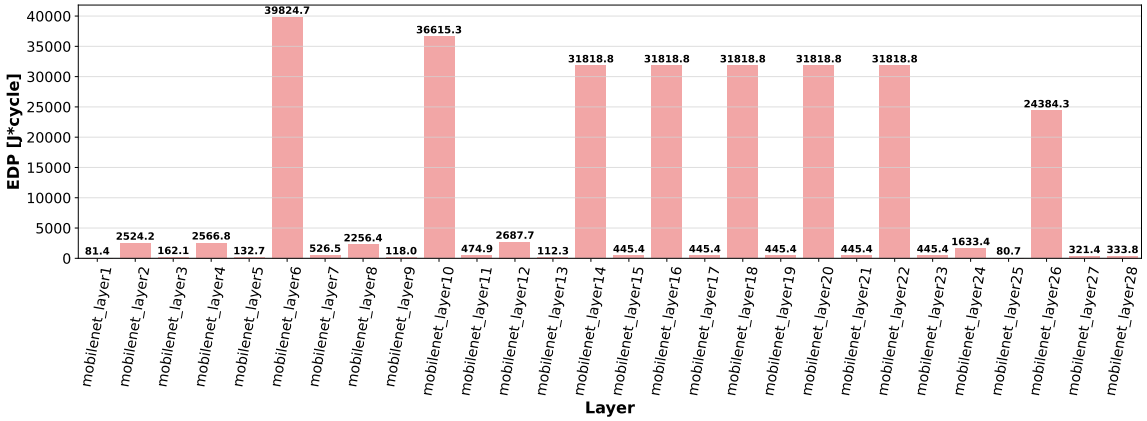
Obrázek B.2: Evaluace optimalizace vůči EDP pro jednotlivé vrstvy sítě MobileNet pro model s 8bitovými operandy příznaků a 4bitovými váhami. Testováno na původní 16bitové int Eyeriss_like architektuře.



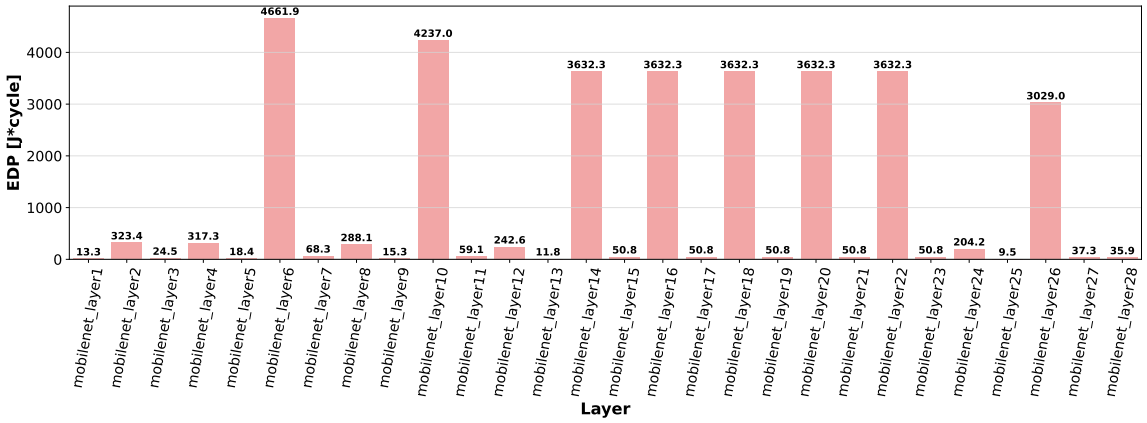
Obrázek B.3: Evaluace optimalizace vůči EDP pro jednotlivé vrstvy sítě MobileNet pro model s 8bitovými operandy příznaků a 4bitovými váhami. Testováno na upravené 8bitové int Eyeriss_like architektuře s ponechanými velikostmi pamětí a šířek přenosových pásem.



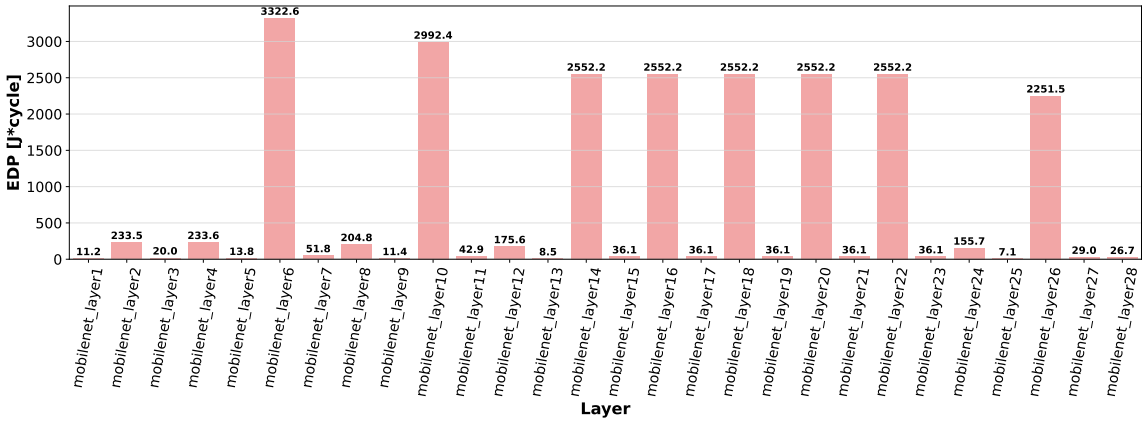
Obrázek B.4: Evaluace optimalizace vůči EDP pro jednotlivé vrstvy sítě MobileNet pro model s 8bitovými operandy příznaků a 4bitovými váhami. Testováno na upravené 8bitové int Eyeriss_like architektuře s pozměněnými velikostmi scratchpad pamětí pro uchování vah na 0.75 původní velikosti a se zdvojnásobenými šířkami přenosových pásem.



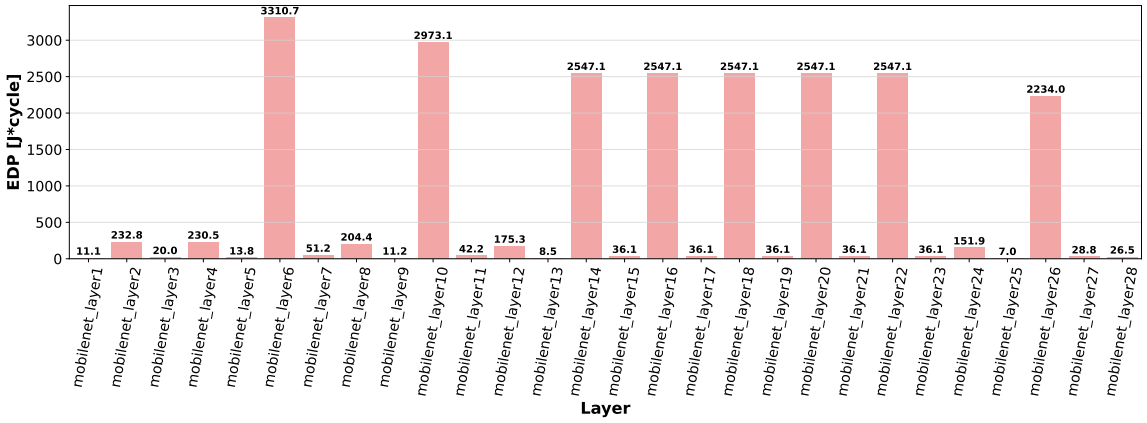
Obrázek B.5: Evaluace optimalizace vůči EDP pro jednotlivé vrstvy sítě MobileNet pro nativní 32-bit float data a adekvátně upravenou architekturu Simba_like.



Obrázek B.6: Evaluace optimalizace vůči EDP pro jednotlivé vrstvy sítě MobileNet pro model s 8bitovými operandy příznaků a 4bitovými váhami. Testováno na původní 16bitové int Simba_like architektuře.



Obrázek B.7: Evaluace optimalizace vůči EDP pro jednotlivé vrstvy sítě MobileNet pro model s 8bitovými operandy příznaků a 4bitovými váhami. Testováno na upravené 8bitové int Simba_like architektuře s ponechanými velikostmi pamětí a šířek přenosových pásem.



Obrázek B.8: Evaluace optimalizace vůči EDP pro jednotlivé vrstvy sítě MobileNet pro model s 8bitovými operandy příznaků a 4bitovými váhami. Testováno na upravené 8bitové int Simba_like architektuře s pozměněnými velikostmi smartbuffer pamětí pro uchování vah na 0.75 původní velikosti a ponechání šířky přenosových pásem.