**Vektorski procesori**

Najveći izvor paralelizma su DoAll petlje kako je već definisano u poglavlju o paralelizaciji petlji. Paralelizam tih petlji, ako se razmatraju samo zavisnosti po podacima, je srazmeran broju iteracija koje treba izvršiti. Od ukupnog broja instrukcija koji se izvršavaju u programima, veliki procenat čine instrukcije unutrašnjih petlji (inner most loop) ugnježdenih petlji. Ako su te petlje DoAll tipa, uvođenjem veoma paralelnih procesora, nazvanih vektorski procesori, može se postići veliko ubrzanje. Pod istim terminom vektorski procesori se podrazumevaju različite kategorije procesora koje paralelizuju programske petlje koje mogu da se vektorizuju. Petlje koje mogu da se vektorizuju su nešto širi pojam od DoAll petlji. Tek na kraju ovog poglavlja će biti opisano koje su to petlje što se mogu paralelizovati pomoću vektorskih procesora, a da nisu DoAll petlje. Kroz poglavlje će se objašnjavanje rada vektorskih procesora zasnivati uglavnom na DoAll petljama.

Pretpostavimo sledeću programsku petlju:

```
Do I = 1,256
E(I) = A(I)*B(8*I)+C(257-I)+D(I);                    (VP.1.)
```

Svi nizovi sa desne strane jednakosti nalaze se u memoriji. Posmatrajući nizove, to je DoAll petlja, jer su svi elementi svih nizova izračunati pre same petlje. Petlja je međutim DoAccross zbog inkrementiranja indeksa petlje I. Razmotavanjem petlje K puta i ubacivanjem K indeksa petlje koji startuju u petlji od 1 do K-1 se može napraviti da suma iteracionih distanci za sve cikluse u grafu bude K. Kako je K izabrano proizvoljno, transformacijom petlje se ona može učiniti DoAll petljom, time što se indeksi petlje za sve iteracije dostave pre petlje. Suštinski razlog zašto može da se radi ova transformacija je što u vreme prevođenja znamo indeks petlje za svaku iteraciju. Navedimo kôd transformisane petlje za K = 16.

```
I1 = -15;
I2 = -14;
I3 = -13;
...
I15 = -1;
I16 = 0;
DO I = 1,16
I1 = I1+16;
I2 = I2+16;                                          (VP.2.)
I3 = I3+16;
...
I15 = I15+16;
I16 = I16+16;
E(I1) = A(I1)*B(8*I1)+C(257-I1)+D(I1);
E(I2) = A(I2)*B(8*I2)+C(257-I2)+D(I2);
E(I3) = A(I3)*B(8*I3)+C(257-I3)+D(I3);
...
E(I15) = A(I15)*B(8*I15)+C(257-I15)+D(I15);
E(I16) = A(I16)*B(8*I16)+C(257-I16)+D(I16);
```

ENDLOOP

Izračunavanja pomoćnih indeksa I1..I16 ponovo čini ovu petlju DoAccross petljom. Međutim, sada je suma iteracionih distanci 1 u ciklusima grafa proisteklih iz izračunavanja I1..I16. Ovom transformacijom sa uvođenjem pomoćnih indeksa, ako posmatramo limit za brzinu izračunavanja ove petlje, zbog ciklusa u grafu, dobijeno je 16 puta manje ograničenje. Razlog za to je činjenica da se za 16 puta razmotanu petlju, 16 puta pojavljuju operacije zavisne same od sebe u susednoj iteraciji, ali sve sa istim ograničenjem kao što je bilo i za originalno izračunavanje indeksa petlje.

U daljem delu se razmatra samo ostatak petlje u kome nema izračunavanja pomoćnih indeksa petlje, jer je pokazano da se može proizvoljno povećavati paralelizam u tom izračunavanju.

Načini paralelizacije DoAll petlji

Praktično svi vektorski procesori imaju vektorske registre namenjene pamćenju nizova (vektora), tako da se u susednim lokacijama čuvaju elementi niza u redosledu u kome treba da se radi izračunavanje. Takvo ređanje elemenata niza omogućava da se uz vektorske registre dodaju brojači, koji svakog ciklusa adresiraju sledeći element niza, za narednu iteraciju petlje. U primeru 1 je kod nizova B i C redosled elemenata nizova drugačiji od redosleda elemenata tih nizova u memoriji. Zato je neophodno da sprega vektorskog procesora i memorije (data keš) bude realizovana tako, da može da prepakuje elemente nizova (vektora), da bi redosled elemenata bio odgovarajući za izračunavanje (smer ka vektorskom procesoru), odnosno čuvanje u memoriji (smer ka memoriji). Sprega mora da radi ta prebacivanja tokom prenosa podataka u “letu”, kako bi se minimizovala kašnjenja. Postoje dve osnovne paradigme kako rade vektorski procesori, a koje se međusobno ne isključuju: Chaining (ranije nazivan vector function chaining) i SIMD.

**Chaining**

Osnova chaining-a je realizacija pipeline-a koji izvršava deo ili ceo aciklički graf petlje. Ako veličina petlje i rekonfigurabilni hardver omogućavaju da se napravi pipeline tako da se izvrši cela petlja, elementi nizova za operacije slobodne na vrhu i operacije koje za jedan ulaz imaju spreman niz pre početka petlje se inicijalno smeštaju u vektorske registre. Zatim se povezivanjem vektorskih registara sa ulazima odgovarajućih funkcionalnih jedinica za izvršavanje operacija i međusobnim povezivanjem funkcionalnih jedinica u skladu sa grafom petlje postiže pipeline za izvršavanje dela ili celog acikličkog grafa petlje.

Zbog kompleksnosti rekonfiguracije i potrebnog broja vektorskih registara u toj logici, broj funkcionalnih elemenata je tipično ispod 10, ali je velika prednost ovakve logike da se svakog ciklusa započinje deo nove iteracije ili čak cela nova iteracija. Vektorski registri imaju najmanje dva izlaza (porta za read) i jedan ulaz i njihovim povezivanjem sa odgovarajućim ulazima i izlazima aritmetičkih jedinica se postiže rekonfigurabilnost povezivanja. Startovanje brojača kojim se adresiraju ulazni i izlazni portovi vektorskih registara se obavlja kontrolnim sekvencama, tako da podaci stignu na ulaze funkcionalnih jedinica tačno kada je potrebno. Trenuci starta naravno zavise od broja pipeline stepeni svake jedinice i potrebnih kašnjenja da bi odgovarajući elementi nizova stizali u isto vreme na ulaze funkcionalnih jedinica.

Dakle, nema data tokena iako su funkcionalne jedinice povezane pomoću vektorskih registara u skladu sa grafom zavisnosti po podacima za petlju. Istovremena pojava ekvivalenata usaglašenih data tokena za istu iteraciju realizovana je pomoću kontrolne sekvence.

Broj lokacija vektorskih registara (m) limitira broj iteracija koje mogu da budu u takvom pipeline-u. Ako je ukupan broj potrebnih iteracija veći od broja lokacija u vektorskim registrima, pipeline rekonfigurabilne logike se prvo puni sa prvih m elemenata nizova u skladu sa redosledom u kome treba da se radi izračunavanje. Zatim se u kvantima radi po m iteracija tako što se vektorski registri pune novim elementima. Petlja (VP.1.) bi za slučaj da je m=128 bila realizovana na sledeći način:

```
Do I = 1,128
E(I) = A(I)*B(8*I)+C(257-I)+D(I);                    (VP.3.)
Do I = 129,256
E(I) = A(I)*B(8*I)+C(257-I)+D(I);
```

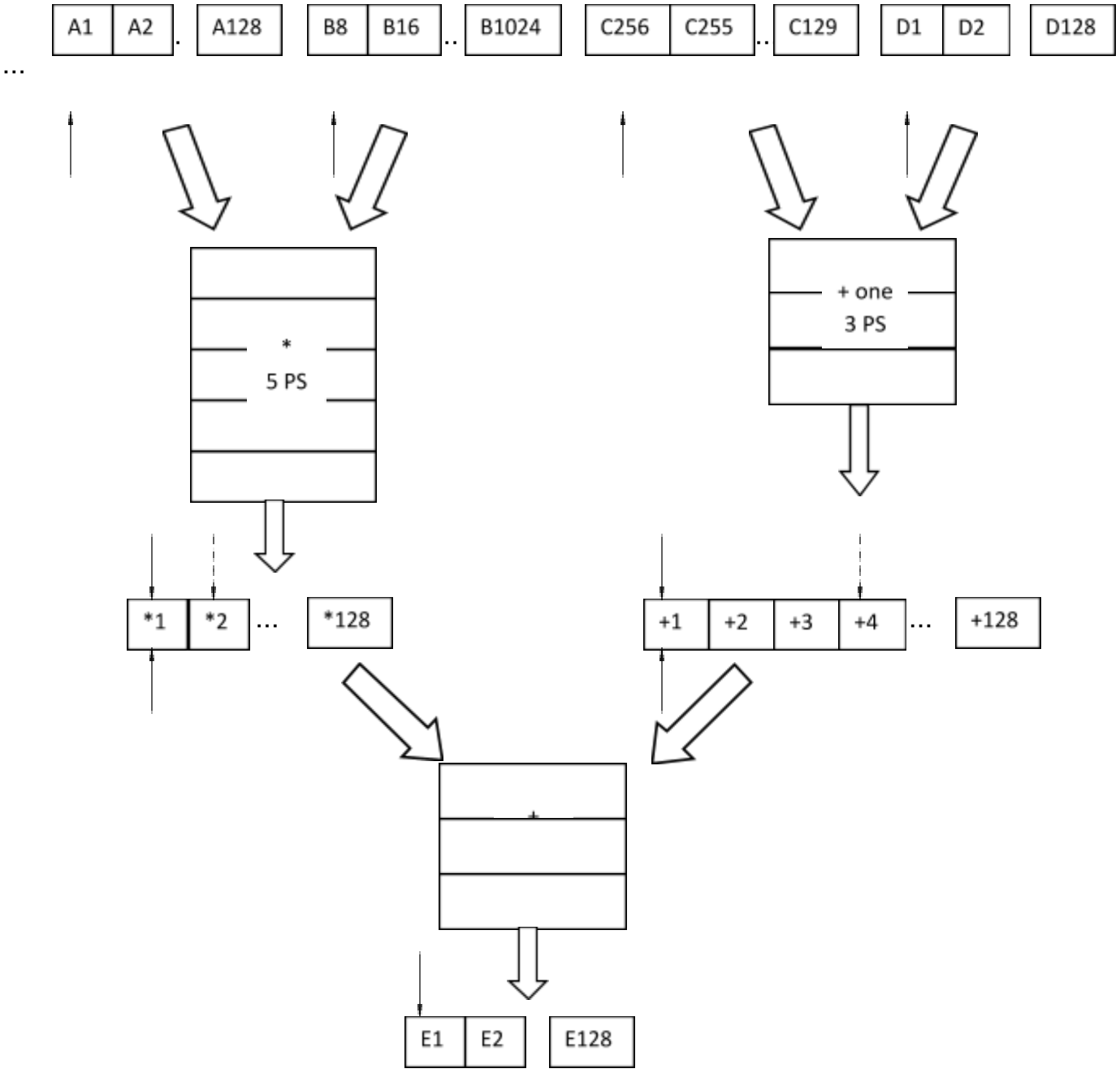
Između ove dve petlje bi se radilo pražnjenje vektorskog registra koji čuva Vektor (niz) E 1..128 i pamćenje tog sadržaja u memoriji. Zatim bi se napunili vektorski registri sa preostalim elementima vektora A,B,C i D u redosledima koji odgovaraju samom računanju iz druge petlje. Odgovarajuće prepakivanje redosleda uvek radi jedinica za spregu vektorskog procesora sa memorijom.

Na Sl. 1. je prikazano kako bi izgledalo izvršavanje prve od dve petlje iz (VP.3.). Pritom neće biti prikazani oni pokazivači (brojači) portova za punjenje i pražnjenje vektorskih registara u interakciji sa memorijom preko sprege. U slučaju petlje iz ovog primera, vektorskim registrima je potreban samo po 1 port za čitanja zbog grafa petlje. Pokazivači (brojači) za adresiranje su prikazani strelicama u početnim položajima, a strelice okrenute naviše pokazuju položaj pokazivača za čitanje elemenata vektorskih registara, a strelice okrenute nadole položaj pokazivača za upis.

Vektorski registri se konfiguracijom veza preko portova vektorskih registara postavljaju na sve ulaze i sve izlaze funkcionalnih jedinica. Oni mogu da se povežu na proizvoljnu funkcionalnu jedinicu i tako se formiraju veze u skladu sa grafom zavisnosti petlje. Njihovi pokazivači/brojači inicijalno imaju položaj na početku vektorskog registra i započinju pomeranje u skladu sa kontrolnom sekvencom. Na primeru izlaza iz množača i sabirača označenih sa * i +one, podaci za istu iteraciju ne stižu u istom trenutku, zbog razlike u broju pipeline stepeni množača i sabirača. Početak pomeranja pokazivača za upis u vektorske registre označene sa * i + je usklađen kontrolnom sekvencom sa pojavom prvog rezultata na izlazu iz množača, odnosno sabirača. Međutim, u trenutku kada je upisan prvi rezultat sabirača, množač još nije izbacio nijedan rezultat u vektorski registar *, zbog razlike u broju pipeline stepeni množača i sabirača. Zato pomeranje pokazivača za upis u vektorski registar * na izlazu množača kreće dva ciklusa kasnije nego pomeranje pokazivača za upis u vektorski registar + na izlazu sabirača.

U dinamičkoj dataflow mašini bi se tražilo da data tokeni budu usklađeni i po vrednosti frame-a. U slučaju chaining-a, kontrolna sekvenca zakašnjava čitanje vektorskog registra + za dva ciklusa, da bi se usaglasili ulazi na +two sabiraču, tako da pripadaju istoj iteraciji. Kako se lokacije vektorskih registara ponašaju kao pipeline registri, čitanje mora da krene najranije jedan ciklus posle upisa u lokaciju vektorskog registra. Čitanje vektorskih registara * i + kreće u istom trenutku, a isprekidanim strelicama okrenutim nadole na registrima * i + je pokazano gde se u tom trenutku nalaze pokazivači za upis u

lokacije tih vektorskih registara. Dakle, ovde je sačekivanje data tokena iz istog frame-a dataflow mašine zamenjeno sa usklađivanjem trenutaka čitanja pomoću pokazivača na port za čitanje vektorskog registra. Ovo usklađivanje je moguće, jer se kašnjenja zavisnosti (odnosno broj stepeni u pipeline-u) znaju u vreme prevođenja, pa kompajler može da isplanira sekvencu.

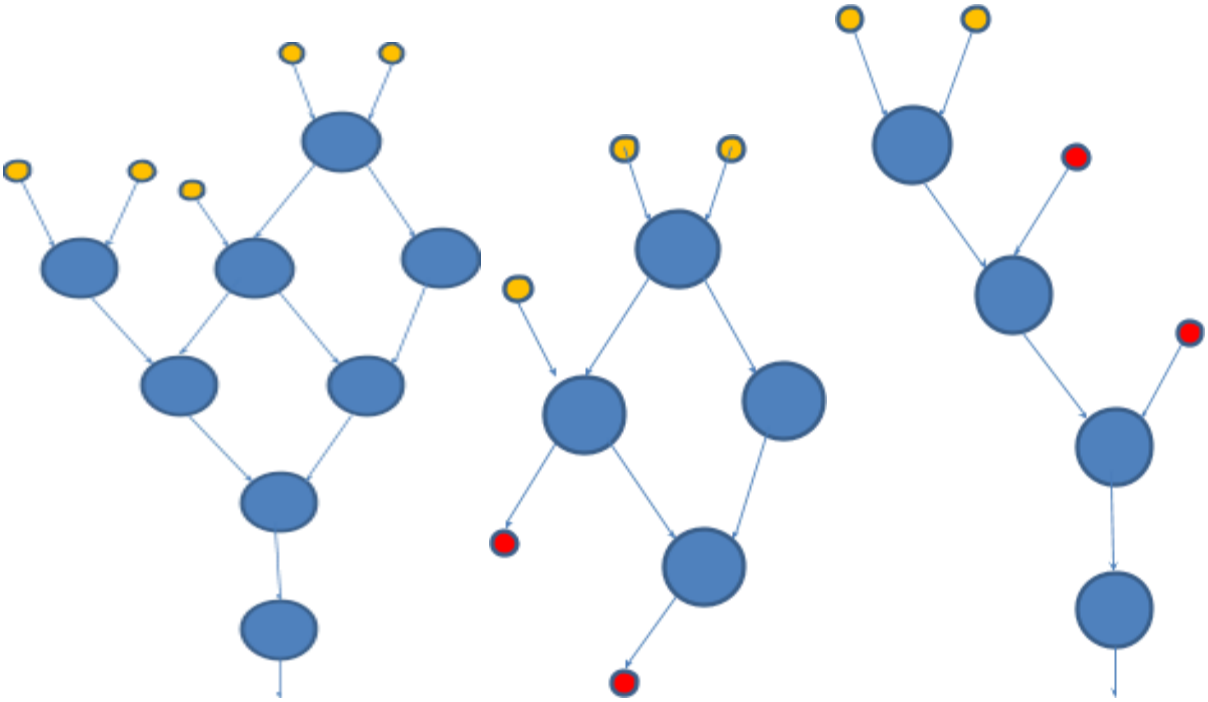



Sl. 1. Izvršavanje prve od dve petlje iz primera (VP.3.)

Chaining je u osnovi primena softverske protočnosti na DoAll petlju tako da se cela petlja ili deo petlje izvršava efektivno za 1 ciklus, kada je jednom napunjen pipeline rekonfigurabilne logike. Osnovni izvor

paralelizma je pipeline koji imitira deo grafa ili ceo graf zavisnosti po podacima za petlje. Kompleksnost rekonfiguracije pomoću vektorskih registara ograničava broj funkcionalnih jedinica i vektorskih registara, a samim tim veličinu grafa petlje, čije se cele iteracije mogu izvršavati u jednom pipeline-u.

Dalje će biti objašnjen koncept kako se radi chaining, kada cela petlja ne može da stane u pipeline. Pođimo od primera DoAll petlje čiji je graf dat Sl. 2. Pretpostavimo radi jednostavnosti da vektorski registri imaju dovoljno lokacija da mogu u njih da stanu vektori za sve iteracije. Takođe, pretpostavimo da su funkcionalne jedinice univerzalne. Ograničenje je broj funkcionalnih jedinica u rekonfigurabilnoj logici i broj vektorskih registara. Pretpostavimo da je broj funkcionalnih jedinica 4 i da je broj vektorskih registara 10. Graf celokupne petlje na delu a. Sl. 2. bi zahtevao 8 funkcionalnih jedinica i ukupno 13 vektorskih registara da bi se mogao uraditi u jedinstvenom pipeline-u. Kako ne postoji toliki broj resursa koji mogu da učestvuju u konfiguracionoj logici, mora se izvršavanje iteracije u opštem slučaju nekako podeliti na dve ili više iteracija. U ovom slučaju može u dve iteracije.



a. Graf originalne petlje

b. Podgraf prvog dela petlje koji je data ready (prvi se izvršava u VP)

c. Podgraf ostatka petlje (mora posle završetka prvog dela petlje)

Sl. 2. Izvršavanje iteracija petlje na vektorkom procesoru kada nije moguće staviti sve operacije u jedan pipeline.

Nedostatak resursa se rešava tako što se iz grafa petlje odvoji podgraf grafa petlje slobodan na vrhu (data ready) za koji postoji dovoljno resursa u vektorskom procesoru da se uradi u jedinstvenom pipeline-u. Tada se urade sve iteracije za takav podgraf i zapamte vrednosti svih potrebnih rezultata tog

podgrafa u vektorskim registrima. Primer takvog podgrafa je dat u delu b. Sl. 2. Nakon izvršavanja tog podgrafa, mogu se osloboditi svi vektorski registri izuzev onih koji čuvaju rezultate operacija 1 i 4 iz svih iteracija. Dakle, u fazi izračunavanja podgrafa sa dela slike b. bilo je potrebno 7 vektorskih registara, a nakon izvršavanja podgrafa se oslobađa 5. Rezultat operacija 0 se u vektorskim registrom na izlazu jedinice koja izvršava tu operaciju distribuira putem dva read porta prema operacijama 1 i 2! Podgraf ostatka petlje se izvršava korišćenjem 8 vektorskih registara i 4 funkcionalne jedinice povezanih u skladu sa podgrafom. Inicijalno moraju da budu napunjeni vektorski registri za operande operacija 3. Ujedno, ostaju popunjeni vektorski registri sa rezultatima operacija 1 i 4 iz prvog podgrafa, a konfiguracijom se podešava da jedan njihov port bude vezan za ulaz aritmetičkih jedinica koje izvršava operaciju 5, odnosno 6. Završetkom izvršavanja podgrafa ostatka petlje se obavi cela petlja.

Chaining je bio ključni mehanizam kojim su se vektorizovale petlje kod CRAY računara. Danas je chaining manje zastupljen od SIMD načina paralelizacije petlje.

Veoma često je punjenje i pražnjenje vektorskih registara u interakciji sa memorijom znatno sporije od samog izvršavanja koda petlji. Zato je veoma važno organizovati te prenose podataka da se veoma brzo odvijaju.

**SIMD vektorski procesori**

Današnji vektorski procesori uglavnom koriste SIMD koncept. Logika SIMD koncepta je da podgraf koji se radi za sve iteracije u jednom koraku bude samo jedna instrukcija slobodna na vrhu, tj. data ready, u grafu petlje. Ona se prvo izvršava za sve iteracije i ukloni se iz grafa petlje. Njeni rezultati se čuvaju u vektorskim registrima! Zahvaljujući ovakvom konceptu, pojednostavljuje se kontrolna sekvenca, i takva obrada se može predstaviti kao elementarna vektorska instrukcija. Programer minimalno treba da poznaje funkcionisanje vektorskog procesora, jer vektorska instrukcija obavi kompletno sekvenciranje. Jedino što programer mora da isplanira je zauzeće vektorskih registara. Da bi ovo imalo smisla, neophodno je da se po ciklusu može započinjati jako veliki broj istovetnih instrukcija. Odatle potiču i naziv Single Instruction Multiple Data i skraćenica SIMD.

Već je napomenuto da glavni problem kod vektorskih procesora predstavlja usko grlo ka memoriji. Da bi se taj problem rešio, stalno je povećavana širina magistrale između SIMD vektorskih procesora i data keš memorije. U Intel seriji procesora, ona se povećavala od 128, preko 256 do 512 bita. Elementi nizova (vektora) su istog tipa i samim tim zauzimaju isti broj bita u delovima reči od 512 bita. Kako oni u memoriji postoje na sukcesivnim lokacijama, u 512 bita su se može smestiti 8 64-bitnih elemenata niza, ili 16 32-bitnih, ili ⋯ ili 64 8bitna podatka za ulaz u SIMD izračunavanje. Ako je potrebno prepakivanje podataka u nizu, kao u slučaju vektora C iz primera petlje (VP.1.), to obavlja interfejs između data keša i SIMD procesora.

U današnjim SIMD procesorima se reči data keša, ako nema konverzije u interfejsu, direktno pamte u lokacijama vektorskih registara. Osnovna ideja je da se nad takvom širokom reči radi paralelna SIMD obrada nad svim elementima u reči. Koncept je zasnovan na činjenici da za većinu aritmetičkih i logičkih operacija je potreban veoma sličan hardver da se uradi operacija nad 16 32-bitnih reči ili nad npr. 8 64-bitnih podataka. Za takvo istovremeno izračunavanje nad svim elementima u širokom registru se koristi termin SWAR (SIMD Within A Register). SWAR aritmetičko-logičke jedinice moraju da se

prilagođavaju broju bita elemenata niza, na osnovu informacije dostavljene vektorskom procesoru kroz vektorsku instrukciju. Koristi se zato i termin polimorfizam u hardveru, jer se aritmetičke jedinice prilagođavaju dostavljenom tipu podataka elemenata vektora.

Broj nizova (vektora) koje je potrebno čuvati u SWAR registrima je manji nego u slučaju chaining-a, jer za jednu instrukciju je neophodno najviše 3 niza, a eventualno je potrebno pričuvati još poneki niz za kasnije vektorske instrukcije. Kao posledica, često se celim nizovima (za sve iteracije vektorizovane petlje) može alocirati skup raspoloživih vektorskih SWAR registara. Broj ciklusa izračunavanja cele vektorske instrukcije je zavisan od broja iteracija originalne petlje N (dužine vektora), tipa podataka (broja bita) elemenata niza b i širine reči SWAR vektorskih registara W. Ako zanemarimo cikluse punjenja pipeline-a i pretpostavimo da je W deljivo sa b (što je uvek slučaj), može se odrediti orijentaciono koliko ciklusa traje izračunavanje vektorske instrukcije. Tada orijentaciono važi:

$$\text{Br. Ciklusa} = \left\lceil \frac{N}{W/b} \right\rceil \qquad \text{(VP.4)}$$

U izrazu se vidi da poslednji SWAR registar ne mora da bude popunjen do kraja elementima niza. Očigledno je da u hardveru mora da postoji rešenje i za delimično korišćenje SWAR registara, ali se u ovom poglavlju neće detaljnije razmatrati taj problem.

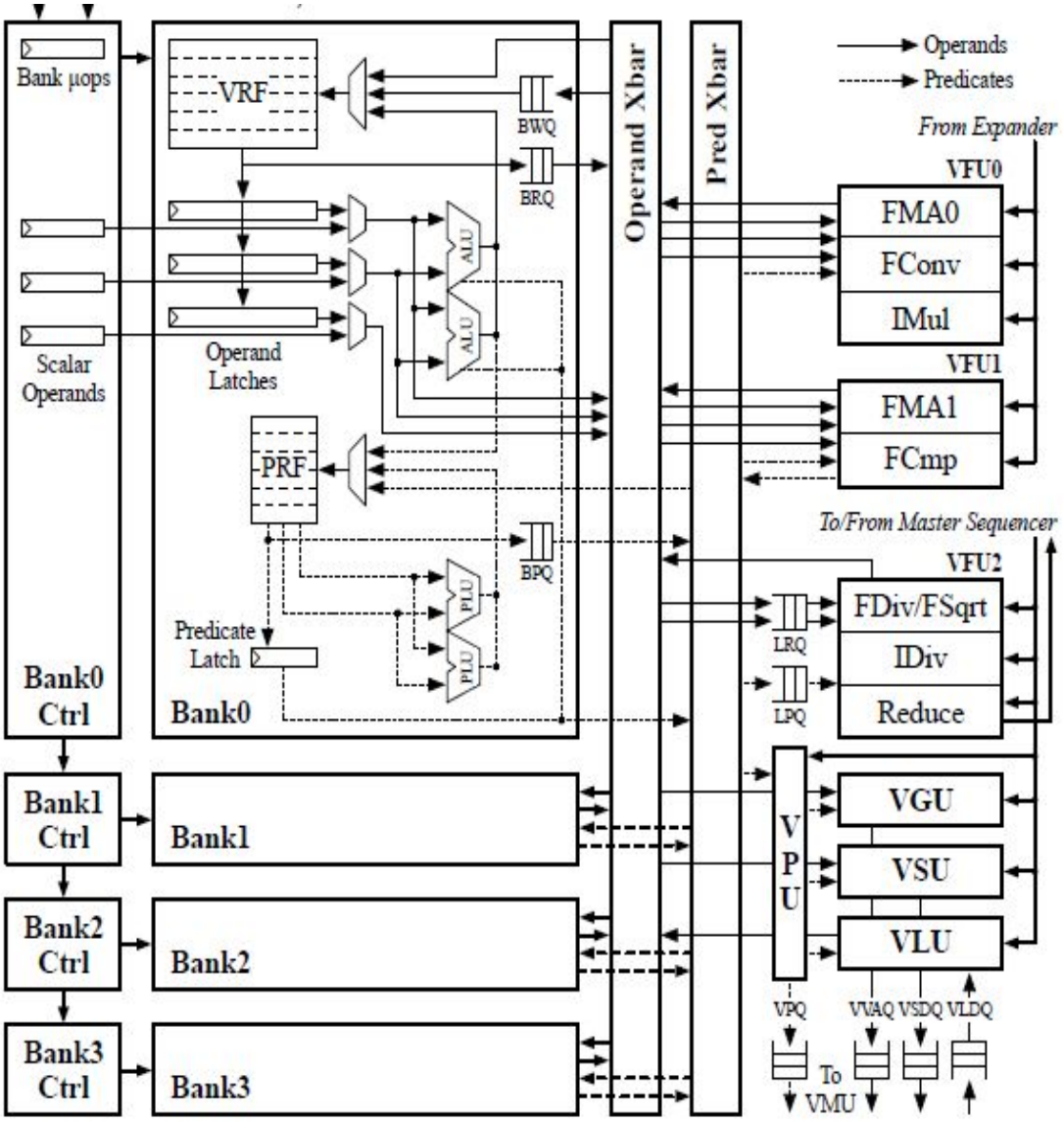
Kako ne postoji nikakav problem da paralelno radi više banki SWAR registara i njima pridruženih aritmetičkih jedinica, može se dodatno smanjiti broj ciklusa. Ako se broj banki i aritmetičkih jedinica poveća npr. 16 puta, tada (VP.4) postaje:

$$\text{Br. Ciklusa} = \left\lceil \frac{N}{16 * W/b} \right\rceil \qquad \text{(VP.5)}$$

Pretpostavimi da originalna petlja ima uslovnih grananja u iteraciji. Tada se nad elementima niza spakovanim u SWAR registar moraju obavljati različite operacije i rezultati upakovati nazad u SWAR registre. Taj naizgled nerešivi problem se rešava pomoću predikatskog izvršavanja (negde to zovu maske). Predikatska logička jedinica treba da izračuna predikate za obradu svih elemenata niza unapred i da radi IF konverziju na osnovu izračunatih predikata. Predikatska ALU (PLU) ima promenljivu širinu magistrale za predikate, jer SWAR registri mogu da sadrže različit broj elemenata niza! Pored predikatske ALU postoji i registarska memorija za predikate, predikatski registarski file (PRF), tako da se mogu pamtiti u njoj predikati za različite SWAR registre istog niza. Širina tih registara PRF je jednaka W/b, za elemente niza sa najmanjim mogućim b koje je podržano. Osim toga, mogu se pamtiti predikati regionskih čvorova svake originalne iteracije i logičkim operacijama nad predikatima, formirati predikati novih regionskih čvorova za petlje sa više grananja i spojeva. Zato je neophodan PLU.

Vektorski procesori su inicijalno primarno pravljeni za floating point operacije. Danas podržavaju i floating point i sve ostale operacije. Interfejs sa magistralom širine 512 bita u današnjoj tehnologiji je

smanjio problem sporosti prenosa podataka iz data keša, pa su je postalo isplativo uraditi značajne dodatne paralelizacije sa više SWAR registara i SWAR ALU jedinica. Umnožavanje svih jedinica izuzev integer množača i svih floating point jedinica ne zauzima veliku površinu na integrisanom kolu. Umnožavanje se radi u trakama (lane) i unutar traka pomoću banaka. Svaka traka ima više vektorskih registarskih memorija SWAR tipa (VRF) sa pridruženim SWAR integerskim ALU i predikatskim ALU jedinicama (PLU) i predikatskim registrima (PRF). Banke su međusobno povezane preko mreže za povezivanje bez blokade (crossbar Xbar), međusobno, na floating point zajedničke SWAR jedinice i integer SWAR množač.




Sl. 3. Traka Hwacha procesora sa 4 banke

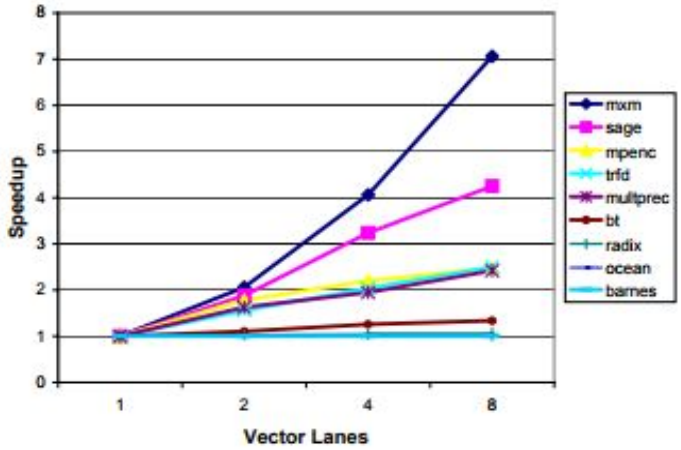
Jedan primer organizacije vektorskog procesora poslednje generacije je primer Hwacha eksperimentalnog vektorskog procesora, koji se razvija od 2016. godine na Univerzitetu Berkley. Njegova organizacija je prikazana na Sl. 3., gde su prikazane 4 banke unutar jedne trake. VPU je interfejsni deo za sve 4 banke iz trake. Floating point zajedničke SWAR jedinice i integer SWAR množač su skoncentrisani u dve grupe zajedničkih jedinica za traku od kojih samo jedna jedinica iz grupe može da pristupa Xbaru za podatke. Svaka od banaka ima sve što je neophodno za autonomno SWAR izračunavanje, dokle god ne zahteva zajedničke jedinice. Orijentacioni broj ciklusa za izračunavanje vektorske instrukcije unutar jedne trake je dato sa:

$$\text{Br. Ciklusa} = \left\lceil \frac{N}{4 * W/b} \right\rceil \qquad \text{(VP.6)}$$

U slučaju Hwacha mašine W je samo 128, jer poenta realizacije te mašine nisu bile performanse, već novi koncepti. U osnovi, to je 4 * SWAR SIMD paralelizam. Kada je u pitanju floating point obrada, po traci ostaje izraz (VP.4), zbog ograničenja u broju izvršnih jedinica.

Trake su umnožene u današnjim vektorskim procesorima tako da se dodatno uvećava paralelizam izvršavanja DoAll petlji. Hwacha ima 4 trake i to je tipična vrednost za današnje vektorske procesore. Trake imaju zajedničku interfejsnu jedinicu prema data kešu. Dok se u jednu traku Hwacha unose reči ili pamte rezultati od 128 bita, u ostalim trakama može da se radi izračunavanje na bazi sadržaja VRF. Za Hwacha mašinu dakle važi orijentacioni izraz (VP.5) za integer vektorske operacije jer u svim trakama ima upravo 16 banaka.

Na Sl. 4. su pokazani rezultati izvršavanja benčmark programa na procesorima sa različitim brojem traka. Rezultati pokazuju da skoro linearno raste brzina obrade na današnjim vektorskim procesorima u kojima su reči širine 512 bita za veoma velike nizove i da interfejsna brzina sa takvom širinom reči nije više značajan ograničavajući faktor.




Sl. 4. Porast brzine izvršavanja sa porastom broja traka za benchmark programe

Intel je sukcesivno za vektorski procesor u poslednjoj deceniji uvodio SWAR reči veličine 128, 256 i 512 bita i za svaku veličinu reči je pravio nove vektorske instrukcije. Tako su se namnožile vektorske instrukcije, ali i hardver koji treba da podržava kompatibilnost za kod vektorskih procesora unazad. Na projektu Hwacha procesora je postignuto da se uradi virtuelizacija hardvera vektorskog procesora uz davanje konfiguracionih informacija kompajleru. Tako se postiže da se sa istim instrukcijama u programima pokriju sve širine SWAR reči i svaki broj traka ne mora menjati kod zbog kompatibilnosti sa novijim i sve moćnijim generacijama vektorskih procesora.

Metoda paralelizacije komponenti jake povezanosti kod petlji omogućava da se komponente jake povezanosti iz grafa petlje, a koje se sastoje od samo jedne instrukcije koja ne zavisi sama od sebe, izvršavaju na vektorskom procesoru. Instrukcije u komponentama jake povezanosti koje imaju više instrukcija, ili jednu instrukciju koja zavisi sama od sebe, treba da se izvršavaju na skalarnom procesoru. Tada mora da se poštuje redosled izvršavanja u skladu sa grafom parcijalnog uređenja komponenti jake povezanosti. Dakle, vektorski procesori se mogu koristiti i za delove DoAccross petlji.

Osim toga, postoji i redak slučaj kako se mogu kompletno vektorizovati DoAccross petlje koje na kritičnom ciklusu u grafu petlje imaju sumu iteracionih distanci k, gde k nije mali broj. Tada se može petlja razbiti na petlje sa k-1 iteracijom, koje se tada ponašaju kao DoAll, jer se zbog ograničenog broja iteracija nije zatvorio nijedan ciklus. Tih k-1 iteracija se mogu izvršiti u SWAR modu, a zatim se dobijeni rezultat operacije koja zatvara ciklus u grafu uzima kao ulaz za sledećih k-1 iteracija u SWAR modu. Ovo liči na ramotavanje k-1 iteracije i dobijanje razmotane (SWAR) petlje koja je sa sumom iteracionih distanci 1 na kritičnom ciklusu. Tako se rade po k-1 iteracija u svakom koraku i dobija paralelizam srazmeran sa k, zato što je DoAccross petlja već imala značajan paralelizam u grafu!

Sa uspehom virtuelizacije hardvera vektorskih procesora će se postići da rastu performanse izvršavanja koda na vektorskim procesorima bez izmene vektorskih instrukcija, pa će se znatno povećati broj programera koji koristi vektorske instrukcije.