Pomen poznavanja računalniške arhitekture

Miha Krajnc

Kje je pomembno znanje arhitekture?

- Če programer ve kako deluje prevajalnik in zbirnik, lahko lažje in hitreje reši napake v kodi.
- Omogoči pisanje **hitrejših** programov.
- Programerji razumejo relativno ceno operacij(**CPI**) in učinke različnega načina pisanja programa

Zakaj potem ne programiramo v zbirniku?

V bistvu bomo programirali z posebnimi metodami, ki se neposredno prevedejo v zbirne ukaze.

Računanje z večimi števili naenkrat

- Osnoven princip SIMD ukazov je paralelno oz. “vektorsko” računanje.
- Podobne operacije poznamo že iz matematike in v bistvu pomenijo da uporabimo enako operacijo nad **n**-timi dimenzijami vektorjev.

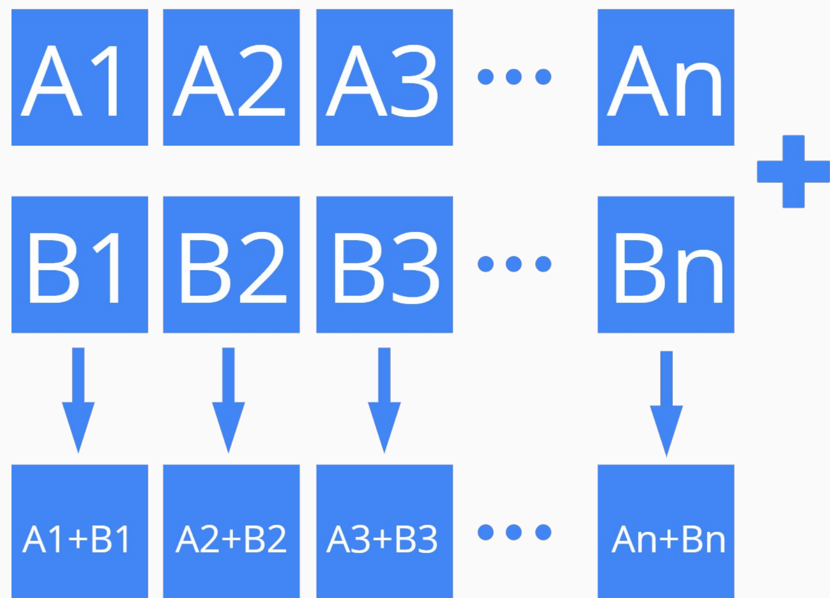
$$\begin{bmatrix} x1 \\ x2 \\ \vdots \\ xn \end{bmatrix} + \begin{bmatrix} y1 \\ y2 \\ \vdots \\ yn \end{bmatrix} \rightarrow \begin{bmatrix} x1+y1 \\ x2+y2 \\ \vdots \\ xn+yn \end{bmatrix}$$

Primer vektorskega seštevanja

Kako pa dejansko pohitrimo algoritem?

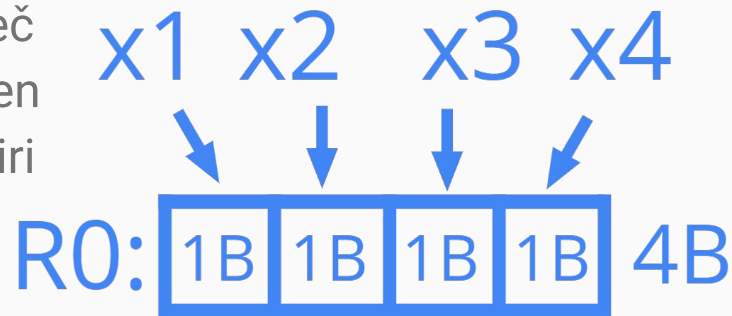
- S pomočjo **SIMD** ukazov lahko z samo enim ukazom izračunamo več različni vrednosti

- Na desni strani imamo primer preprostega paralelnega seštevanja
- Centralno procesna enota lahko sešteje dva vektorja skupaj v enakem času kot dve posamezni števili



Kako vektorje predstavimo CPE-ju?

- Še vedno ostaja goreče vprašanje: **Kako lahko CPE računa z več vrednostmi naenkrat?**
- Kratek odgovor vprašanju bi bil, da v bistvu računamo le z dvema registroma naenkrat ampak z večimi vrednostmi v teh registrih.
- Tukaj imamo preprost primer, kako lahko več manjših spremenljivk shranimo v posamezen register. V en 4B register lahko shranimo štiri 1B vrednosti.



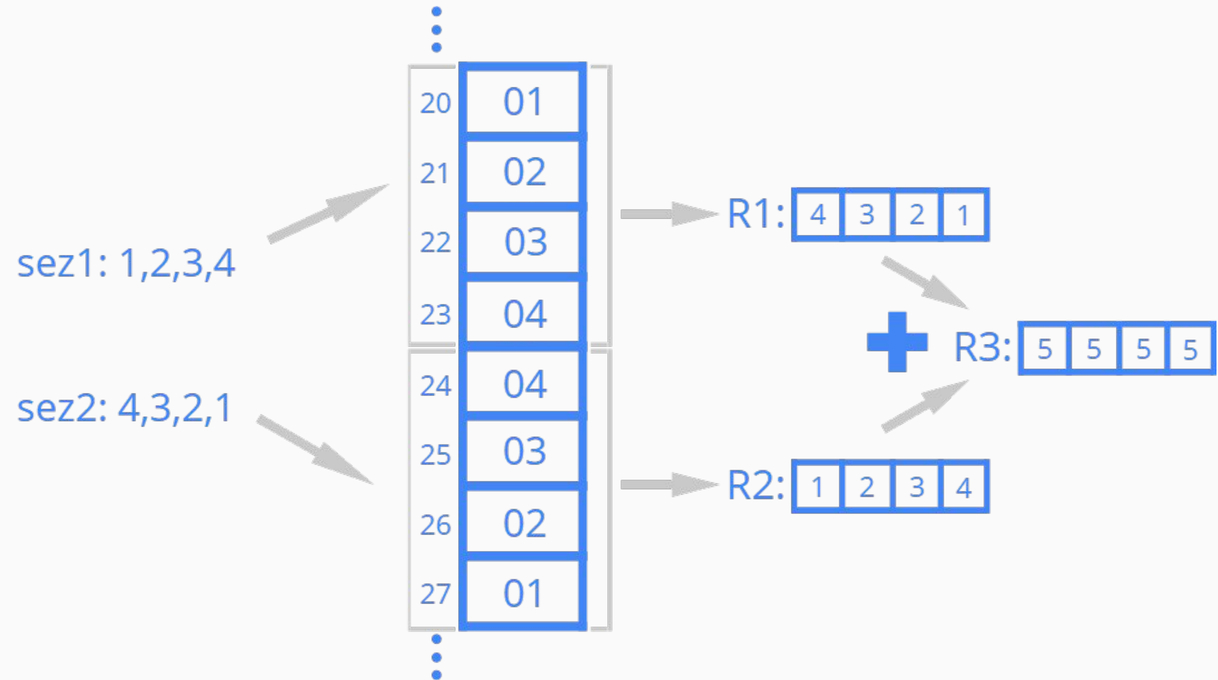
Slabe lastnosti SIMD ukazov

- Navkljub vsem prednostim, pa obstajajo razlogi zakaj takšnih ukazov ne srečujemo v splošni uporabi.
- Glaven razlog za to je, da **izbiro ukazov v večini določa velikost registrov** na posameznem procesorju, oziroma nabor ukazov, ki ga imamo na voljo.
- Za izvedbo algoritma z SIMD ukazi je potrebno napisati **mnogokrat več vrstic kode**.
- Da lahko podpremo vse različne ukazne nabore moramo napisati različne verzije algoritma za posamezne nabore.

Preprost primer - seštevanje seznamov

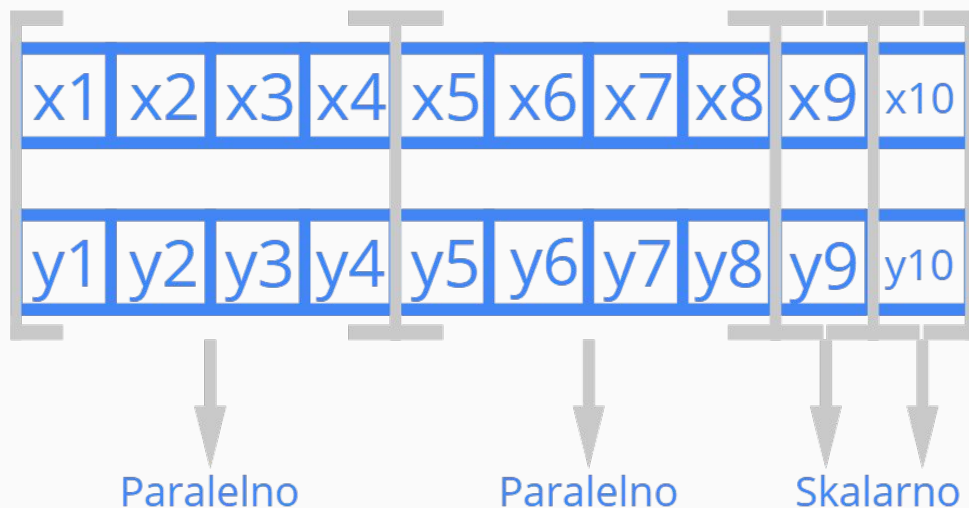
- Za primer uporabe SIMD operacij bomo poskusili sešteti dva seznama števil

```
sez1:   .byte   1,2,3,4
sez2:   .byte   4,3,2,1
adr r0, sez1
ldr r1, [r0]
ldr r2, [r0,#4]
uadd8 r3, r1, r2
```



Preprost primer - seštevanje seznamov

- Če imamo slučajno podatkov preveč za en register, jih moramo računati iterativno in se pomikati naprej. Lahko se nam tudi zgodi, da nam na koncu nekaj bajtov podatkov ostane prostih. V takšnem primeru moramo preostale podatke izračunati skalarno.



Kako pa v višje-nivojskih jezikih?

- Do sedaj smo SIMD ukaze spoznali le v zbirniku. Kako pa naj takšne operacije uporabljamo v višje-nivojskih jezikih?
- Popularnejši jeziki imajo v ta namen razvite **knjižnice**, ki vsebujejo posebne funkcije, ki jih prevajalnik prevede v točno določene ukaze.
- Zelo razvito podporo SIMD ukazov ima **C++**, saj je **Intel** razvil posebne funkcije(intrinzike) izključno za namen pohitritve.

  Intel® Intrinsics Guide

Podatkovni tipi

- Z intrinziki operiramo nad posebnimi vektorskimi podatkovnimi tipi. V glavnem pa jih ločimo na 3 skupine: **integer, float, double**

__m128   vsebuje 4 32-bitna števila v floating point zapisu

__m128d   vsebuje 2 64-bitni števili v floating point zapisu

__m128i   lahko vsebuje 16x 1B ALI 8x 2B ALI 4x 4B ALI 2x 8B integer števil

__m256   vsebuje 8 32-bitna števila v floating point zapisu

__m256d   vsebuje 4 64-bitni števili v floating point zapisu

__m256i   lahko vsebuje 32x 1B ALI 16x 2B ALI 8x 4B ALI 4x 8B integer števil

Razlike ukaznih naborov

- Različnih verzij ukaznih naborov, ki jih uporabljamo je veliko, ampak jih lahko poenostavimo glede na katere tipe vektorjev podpirajo.

Ukazni nabor	__m64	__m128	__m128d	__m128i	__m256	__m256d	__m256i
SSE	da	da	ne	ne	ne	ne	ne
SSE2	da	da	da	da	ne	ne	ne
SSE3	da	da	da	da	ne	ne	ne
AVX	da	da	da	da	da	da	da
⋮			⋮				

Prevajalnik in SIMD ukazi

- Če pri optimizaciji kode ogromno pomaga tudi prevajalnik(evalvacija konstant, globalna optimizacija, ...), zakaj potem ne morejo sami generirati SIMD strojnih ukazov?
- V resnici prevajalnik lahko in tudi jih generira. Težava je, da so takšne avtomatične optimizacije redke, saj prevajalnik težko prepozna primere, ki se jih da optimizirati z SIMD ukazi.
- Nekakšna vmesna rešitev je tudi “masaža kode”. Izraz pomeni, da programer oblikuje svojo kodo z namenom, da jo prevajalnik prepozna in prevede v SIMD ukaze.

Praktičen primer - računanje ploščin

Podana imamo dva seznama. Eden vsebuje dolžine, drugi pa širine nekih pravokotnih zemljišč. Naša naloga je, da izračunamo ploščine zemljišč in jih shranimo v nov seznam.

Za realizacijo rešitve bomo uporabili jezik **C++** z intrinzičnimi operacijami. Na koncu bomo primerjali “benchmark” rezultate med navadno in pospešeno rešitvijo.

a = [2,4]
b = [1,2]
→ c = [2*1,4*2]

Praktičen primer - postavitev okolice

- Kot že omenjeno, bomo za rešitev primera uporabili programski jezik C++ z intrinzičnimi funkcijami. Njihovo uporabo moramo zato v programu definirati.

  ```
  #include <intrin.h>
  ```

- Pomembno je pa, da tudi preverimo katere ukazne nabore podpira naš procesor. To z lahkoto najdemo na spletu, če poiščemo naziv procesorja ki ga imamo v računalniku npr. “Intel i7-8750H”

  Instruction Set Extensions: Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2

Računanje ploščin - rešitev

- Podan problem z lahkoto rešimo z le eno zanko, v kateri skupaj množimo dve števili na enakem indeksu.
- Očitno je tudi, da kompleksnosti O(n) ne moremo zmanjšati.

```
153     double results[st];
154
155     for(int i = 0; i < st; ++i)
156     {
157         results[i] = a[i] * b[i];
158     }
159 }
```

Računanje ploščin - paralelna rešitev

- Spodaj je primer enake rešitve napisane z intrinzičnimi funkcijami.
- Takoj lahko prepoznamo podobnosti z zbirnim jezikom in da je potrebno napisati kar nekaj več vrstic kode.

```
163     float results[st];
164
165     for(int i = 0; i < (st - 8); i += 8)
166     {
167         __m256 i_a = _mm256_load_ps(&a[i]);
168         __m256 i_b = _mm256_load_ps(&b[i]);
169         __m256 i_c = _mm256_mul_ps(i_a, i_b);
170         _mm256_store_ps(&results[i], i_c);
171     }
172
173     for(int i = (st - 8); i < st; ++i)
174     {
175         results[i] = a[i] * b[i];
176     }
```

Računanje ploščin - rezultati

- Na koncu še primerjamo obe funkciji po časovni kompleksnosti. Spodaj vidimo presenetljive rezultate.
- Intrinzična rešitev, ki **obsega 2x več vrstic kode** kot prva je skoraj **4x hitrejša**.

Experiment	Prob. Space	Samples	Iterations	Baseline	us/Iteration	Iterations/sec
Iterativno	Null	1000	1000	1.00000	2.02500	493827.16
Intrinzicno	Null	1000	1000	0.26321	0.53300	1876172.61

Iz benchmark rezultatov je razvidno, da je intrinzična rešitev hitrejša za 380%

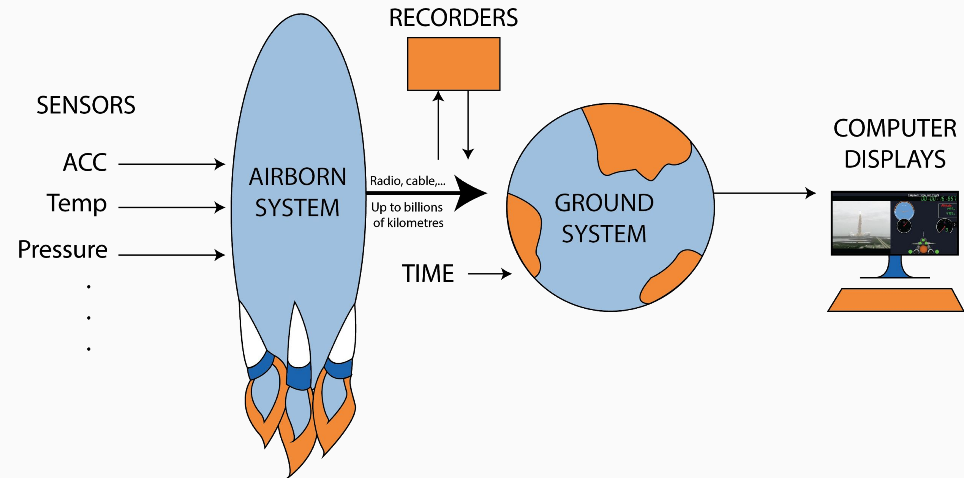
Intrinziki v praksi

- Uporabe intrinzičnih funkcij se v praksi poslužujemo, ko imamo opravka z procesiranjem večjih količin podatkov oz. procesiranje v realnem času.
- Pravilo dela v praksi je ponavadi takšno, da program spišemo z navadnimi skalarnimi operacijami in potem prepoznamo dele, ki jih lahko oz. ki se jih splača optimizirati z SIMD ukazi.
- Deli kode, ki se jih splača pohitriti so ponavadi iterativne operacije z običajno večjim številom iteracij(več 100).

Intrinziki v praksi - primeri

Izris podatkov v realnem času

TELEMETRY SYSTEM



Procesiranje in prenos podatkov v realnem času

Povzetek

- Računalniška arhitektura je ena izmed temeljnih znanj, ki so potrebna za pravilno in učinkovito uporabo virov računalnika.
- Z SIMD ukazi lahko določen program optimiziramo preko vseh pričakovanih meja(tudi do x64).
- Četudi ne nameravamo uporabljati SIMD ukazov, lahko le z osnovnim poznavanjem koncepta učinkovito pišemo kodo, pravilno definiramo podatkovnem vire, ...