a SIMT

3. Datový paralelismus SIMD, HW implementace a SW podpora na CPU a GPU

V praxi sa vyskytuje rada výpočetných (pod)problémov, ktoré potrebujú vykonať rovnakú (nezávislú) transformáciu pre množinu prvkov = dátový paralelizmus. Pragmatický prístup výrobcov HW adresuje práve túto spoločnú vlastnosť výpočtov (rovnaká transformácia N prvkov) na dosiahnutie vyššieho výkonu (throughput), a to replikáciou funkčných jednotiek. Keďže operácia nad prvkami je rovnaká, netreba aby sa replikovali plnohodnotné procesory, ktoré obsahujú vlastnú riadiacu jednotku. Pridané procesory sú riadené jednou jednotkou, a teda dochádza k úspore miesta na čipe, resp. ceny. Zdieľanie riadiacej jednotky znamená, že všetky procesory pracujú synchrónne a v danom čase vykonávajú tú istú inštrukciu - a teda sa takéto architektúra označuje Single Instruction, Multiple Data (SIMD).

Silné stránky:

- spracovanie vektorov, matíc, tenzorov

Slabé stránky:

- Transformácie, ktoré obsahujú veľa podmienok , obsahující hodně větvení

Typy paralelizmu v CPU - *kde sa zaraďuje SIMD?*:

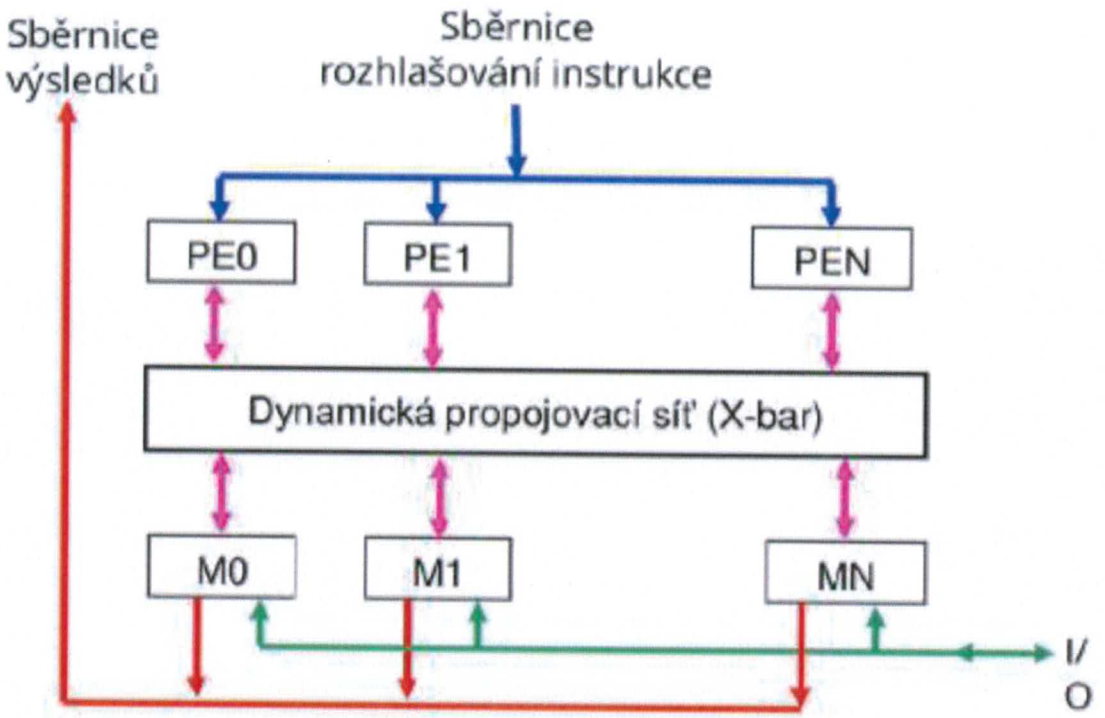
1. Funkčný paralelizmus - rôzne inštrukcie používajú rôzne časti CPU, a preto je ich možné vykonať paralelne (za predpokladu, že medzi nimi nie sú dátové závislosti). Príkladom sú superskalárne procesory (pararelizmus je riadený HW - OoO), a procesory VLIW (Very long instruction word; paralelizmus je riadený kompilátorom)
2. Dátový paralelizmus
    a. V čase - zreťazené spracovanie
    b. V priestoru - replikácia funkčných jednotiek (SIMD)
    c. V oboch dimenziách (paralelná vektorové linky, prípadne SIMT - Single Instruction Multiple Threads)

SIMD Architektúry

- Obsahujú veľký počet výpočetných jednotiek (processing units PE) a jednu riadiaci jednotku
- Počet PE na CPU a GPU je v rozsahu 4-32
- pri návrhu SIMD architektúry je potreba riešiť dilema ako moc zložité budú PE (čo všetko budú vedieť spočítať). Čím zložitejšie, tým viac miesta zaberú a tým menej sa ich na plochu o daných rozmerov zmestí.
    - Typicky umožňujú veľký počet jednotiek, ktoré umožňujú základné výpočty (ADD, SUB, CMP, ...) a malý počet jednotiek pre náročnejšie operácie (DIV, SIN, SQRT, ...)

- Typicky realizácie týchto architektúr vystupujú v úlohe koprocesorov (GPU, AVX), prípadne HW akcelerátorov – ich použitie ako general purpose CPU by bolo problematické, keďže pre niektoré druhy výpočtov sú pomalé.

SIMD so zdieľanou pamäťou:



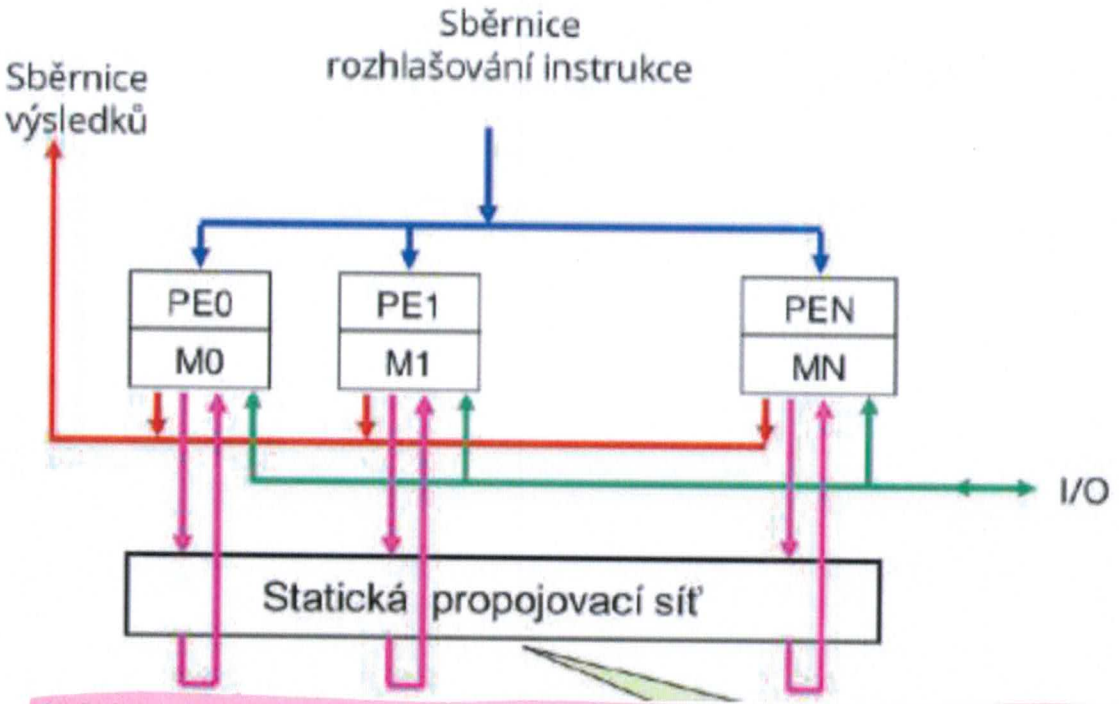
- Každý PE má rovnaký prístup k pamäťovým modulom, ktorý prebieha prostredníctvom dynamickej prepojovacej siete (napr. crossbar - krížový prepínač)
- Komunikácie medzi PE prebieha cez zdieľanú pamäť
- Príklady realizácie: AVX jednotky v CPU, SM procesory v GPU

- kolik dnes využívají procesory a GPUčka
- dá se to modelovat PRAM modelem s CREW

Kdy má vektorizace smysl? - Aritmetická intenzita

- výkonnost jakéhokoli algoritmu je shora omezena buď propustností ALU nebo paměti
- pokud nedosáhneme rozumné aritmetické intenzity (FLOP/B), nemá smysl se do vektorizace vůbec pouštět

SIMD s distribuovanou pamäťou:



- Každý PE má vlastný pamäťový modul - výpočty vykonáva nad lokálnymi dátami (dáta sú blízko pri PE = rýchlosť prístupu)
- Komunikácia s ostatnými PE prebieha cez statickú prepojovaciu sieť (mriežka/torus a pod.)
- Príklady realizácie: IBM Cell v Sony playstation 3

Orientácia v SIMD v moderných CPU - inštrukčné sady

- X86
  - SSE (SIMD Streaming Extensions! - 128b registre. Pôvodne 4x single precision float, s neskoršími vydaniami (SSE2, SSE3, ...) sa pridáva možnosť interpretácií: register = 2x double | 16x byte | 4x 32 bit integer | ...
  - AVX - rozširuje šírku registra na 256b
  - AVX512 - rozširuje šírku registra na 512b
- ARM
  - ARM Neon - inštrukčná sada kombinujúca 64b a 128b operácie

SW podpora

Ako môžem použiť SIMD na urýchlenie výpočtu?

- Vektorizované knižnice
  - Intel MKL, TensorFlow

- Automatická vektorizace kompilátorem (je nutné uistiť sa, že kompilátor bol schopný kód automaticky vektorizovať)
- Pragma hinty kompilátoru:

```
#pragma omp simd
for (int i = 0; i < K; ++i) { a[i] = 2*a[i] + b[i]; }
```

- Vektorové intrinsic funkcie:
  - Hlavičkový súbor, ktorý pridáva C funkcie, ktoré sa svojím menom podobajú assembleru (napr. `__mm_add_ps`). Tento hlavičkový súbor rovnako pridáva dátové typy, ktoré odrážajú SIMD registre (`__m512`).
- Asembler kód

Automatická vektorizácia kompilátorom - čo dokáže vektorizovať?

- Smyčky, kde je známy a počas iterácie nemenný počet iterácií (kde je předem známý počet iterací)
- Smyčky s jedným vstupom a výstupom (když bude ve smyčce break, bude to problém)
- Smyčky, ktoré neobsahujú zložité vetvenie. Jednoduché podmienky je možné vektorizovať s využitím maskovania, napr. v nasledujúcom kóde sa vyhodnotí podmienka pre celý SIMD register obsahujúci napr. 8 float čísiel pričom výsledok je binárna maska, ktorá hovorí, či táto podmienka platí alebo pre daný element registra. Následne sa táto maska použije a do elementov, ktoré ju spĺňajú sa presunie výsledok `then` vetvy, a do časti ktoré ju nespĺňajú sa presunie výsledok `else` vetvy.

```
for (int i = 0; i < length; i++) {
        if (s >= 0) x[i] = b[i] * b[i] – 4 * a[i] * c[i];
        else x[i] = 0.;
}
```

(jednodušší větvení přes blend)

#omp declare simd

- Iba najvnútornejšie smyčky (pouze nejvnitřnější smyčky)
- Smyčky bez volania funkcií až na intrinsic matematiku, inline funkcie, OMP SIMD funkcie…

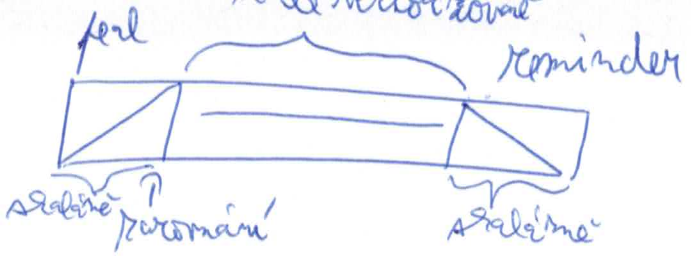
Automatická vektorizácia kompilátorom - kde zlyháva? Kdy to nejde:

- Nejednotkový rozostup - je nutné, aby elementy boli za sebou v pamäti s konštantným rozostupom. Ak je rozostup >1, je možné použiť inštrukcie gather/scatter (typicky obsahujúce penalizáciu).
- Nezarovnané štruktúry
- Dátové závislosti medzi iteráciami
- Pointer aliasing - je nutné testovať za chodu, či sa použite pamäťové lokality môžu prekrývať
- může říct, že je vektorizace možná, ale dle heuristiky compilátoru neefektivní

Zarovnanie pamäte

- Zlé zarovnanie pamäte je jeden z hlavných dôvodov zlého výkonu. Kompilátor preto vkladá do kódu test na zarovnanie, a pokiaľ nie je spracovávané pole korektne zarovnané je vygenerovaný peal, body, reminder - body obsahuje vektorizovaný výpočet na

peal a reminder se chci zbavit
=> proto zarovnání



zarovnanej pamäti, peal obsahuje nevektorizovaný kód spracúvajúci časť poľa pred zarovnanou časťou, obdobne reminder je nevektorizovaný kód spracúvajúci časť za zarovnanou časťou.

- Pre predídenie generovania nevektorizovaných častí (peal, reminder) je nutné explicitne deklarovať zarovnanie, a to pri:
  - Alokácii dát (použiť napr. `aligned_alloc` namiesto `malloc`)
  - Deklarácii nového ukazateľa
  - Deklarácii novej funkcie ktorej parameter je ukazateľ

- když je vstupem funkce ukazatel, tak kompilátor nebude předpokládat zarovnání

  #pragma omp simd align(a:64)

  ↑ řekne compilátoru, že a je zarovnáno

SIMT (Single Instruction Multiple Threads)

- typ SIMD
- většinou GPU
- např: ~~místo smyčky budeme~~

  smyčky úplně rozbalíme a každý jeden prvek v matici bude počítat jedno vlákno (můžeme jich mít až miliardu)

  Programujeme pak kód pro jedno vlákno a pracujeme s jeho indexem pro určení jeho "práce"

- thready se shlukují do WARPů, což je vlastně skupina vláken, které mají společný programový čítač (obdoba AVX), WARPy pak do bloků, které se přiřazují na jednotlivé procesory
- GPU pak vždy zpracovává celý WARP a switchuje mezi nimi pro vyhýbání čekání
- GPU jsou většinou skalární <u>in-order</u>

Programování pro GPU (SIMT)

- nízkoúrovňově - CUDA
- využití OpenMP direktiv (OpenMP 4.5)
  - využívají tasky, protože GPU nedovolují bariéry na pozadí
  - #pragma omp target
  - #pragma omp teams
  - #pragma omp distribute